Julia Evans

How to get a core dump for a segfault on Linux

This week at work I spent all week trying to debug a segfault. I'd never done this before, and some of the basic things involved (get a core dump! find the line number that segfaulted!) took me a long time to figure out. So here's a blog post explaining how to do those things!

At the end of this blog post, you should know how to go from "oh no my program is segfaulting and I have no idea what is happening" to "well I know what its stack / line number was when it segfaulted at at least!".

what's a segfault?

A "segmentation fault" is when your program tries to access memory that it's not allowed to access, or tries to . This can be caused by:

- trying to dereference a null pointer (you're not allowed to access the memory address `0`)
- trying to dereference some other pointer that isn't in your memory
- a C++ vtable pointer that got corrupted and is pointing to the wrong place, which causes the program to try to execute some memory that isn't executable
- some other things that I don't understand, like I think misaligned memory accesses can also segfault

This "C++ vtable pointer" thing is what was happening to my segfaulting program. I might explain that in a future blog post because I didn't know any C++ at the beginning of this week and this vtable lookup thing was a new way for a program to segfault that I didn't know about.

But! This blog post isn't about C++ bugs. Let's talk about the basics, like, how do we even get a core dump?

step 1: run valgrind

I found the easiest way to figure out why my program is segfaulting was to use valgrind: I ran

```
valgrind -v your-program
```

and this gave me a stack trace of what happened. Neat!

But I wanted also wanted to do a more in-depth investigation and find out more than just what valgrind was telling me! So I wanted to get a core dump and explore it.

How to get a core dump

A **core dump** is a copy of your program's memory, and it's useful when you're trying to debug what went wrong with your problematic program.

When your program segfaults, the Linux kernel will sometimes write a core dump to disk. When I originally tried to get a core dump, I was pretty frustrated for a long time because – Linux wasn't writing a core dump!! Where was my core dump????

Here's what I ended up doing:

1. Run `ulimit -c unlimited` before starting my program
2. Run `sudo sysctl -w kernel.core_pattern=/tmp/core-%e.%p.%h.%t`

ulimit: set the max size of a core dump

`ulimit -c` sets the **maximum size of a core dump**. It's often set to 0, which means that the kernel won't write core dumps at all. It's in kilobytes. ulimits are **per process** – you can see a process's limits by running `cat /proc/PID/limit`

For example these are the limits for a random Firefox process on my system:

```
$ cat /proc/6309/limits
Limit                     Soft Limit           Hard Limit           Units
Max cpu time              unlimited            unlimited            seconds
Max file size             unlimited            unlimited            bytes
Max data size             unlimited            unlimited            bytes
Max stack size            8388608              unlimited            bytes
Max core file size        0                    unlimited            bytes
Max resident set          unlimited            unlimited            bytes
Max processes             30571                30571                processes
Max open files            1024                 1048576              files
Max locked memory         65536                65536                bytes
Max address space         unlimited            unlimited            bytes
Max file locks            unlimited            unlimited            locks
Max pending signals       30571                30571                signals
Max msgqueue size         819200               819200               bytes
```

```
Max nice priority         0                   0
Max realtime priority     0                   0
Max realtime timeout      unlimited           unlimited           us
```

The kernel uses the **soft limit** (in this case, “max core file size = 0”) when deciding how big of a core file to write. You can increase the soft limit up to the hard limit using the `ulimit` shell builtin (`ulimit -c unlimited`!)

kernel.core_pattern: where core dumps are written

`kernel.core_pattern` is a kernel parameter or a “sysctl setting” that controls where the Linux kernel writes core dumps to disk.

Kernel parameters are a way to set **global** settings on your system. You can get a list of every kernel parameter by running `sysctl -a`, or use `sysctl kernel.core_pattern` to look at the `kernel.core_pattern` setting specifically.

So `sysctl -w kernel.core_pattern=/tmp/core-%e.%p.%h.%t` will write core dumps to `/tmp/core-<a bunch of stuff identifying the process>`

If you want to know more about what these `%e`, `%p` parameters read, see man core.

It’s important to know that `kernel.core_pattern` is a global settings – it’s good to be a little careful about changing it because it’s possible that other systems depend on it being set a certain way.

kernel.core_pattern & Ubuntu

By default on Ubuntu systems, this is what `kernel.core_pattern` is set to

```
$ sysctl kernel.core_pattern
kernel.core_pattern = |/usr/share/apport/apport %p %s %c %d %P
```

This caused me a lot of confusion (what is this apport thing and what is it doing with my core dumps??) so here’s what I learned about this:

- Ubuntu uses a system called “apport” to report crashes in apt packages
- Setting `kernel.core_pattern=|/usr/share/apport/apport %p %s %c %d %P` means that core dumps will be piped to `apport`
- apport has logs in /var/log/apport.log
- apport by default will ignore crashes from binaries that aren’t part of an Ubuntu packages

I ended up just overriding this Apport business and setting `kernel.core_pattern` to `sysctl -w kernel.core_pattern=/tmp/core-%e.%p.%h.%t` because I was on a dev machine, I didn’t care whether Apport was working on not, and I didn’t feel like trying to convince Apport to give me my core dumps.

So you have a core dump. Now what?

Okay, now we know about ulimits and `kernel.core_pattern` and you have actually have a core dump file on disk in `/tmp`. Amazing! Now what??? We still don't know why the program segfaulted!

The next step is to open the core file with `gdb` and get a backtrace.

Getting a backtrace from gdb

You can open a core file with gdb like this:

```
$ gdb -c my_core_file
```

Next, we want to know what the stack was when the program crashed. Running `bt` at the gdb prompt will give you a backtrace. In my case gdb hadn't loaded symbols for the binary, so it was just like `??????`. Luckily, loading symbols fixed it.

Here's how to load debugging symbols.

```
symbol-file /path/to/my/binary
sharedlibrary
```

This loads symbols from the binary and from any shared libraries the binary uses. Once I did that, gdb gave me a beautiful stack trace with line numbers when I ran `bt`!!!

If you want this to work, the binary should be compiled with debugging symbols. Having line numbers in your stack traces is extremely helpful when trying to figure out why a program crashed :)

look at the stack for every thread

Here's how to get the stack for every thread in gdb!

```
thread apply all bt full
```

gdb + core dumps = amazing

If you have a core dump & debugging symbols and gdb, you are in an amazing situation!! You can go up and down the call stack, print out variables, and poke around in memory to see what happened. It's the best.

If you are still working on being a gdb wizard, you can also just print out the stack trace with `bt` and that's okay :)

ASAN

Another path to figuring out your segfault is to do one compile the program with AddressSanitizer ("ASAN") (`$CC -fsanitize=address`) and run it. I'm not going to discuss that in this post because this is already pretty long and anyway in my case the segfault disappeared with ASAN turned on for some reason, possibly because the ASAN build used a different memory allocator (system malloc instead of tcmalloc).

I might write about ASAN more in the future if I ever get it to work :)

getting a stack trace from a core dump is pretty approachable!

This blog post sounds like a lot and I was pretty confused when I was doing it but really there aren't all that many steps to getting a stack trace out of a segfaulting program:

1. try valgrind

if that doesn't work, or if you want to have a core dump to investigate:

1. make sure the binary is compiled with debugging symbols
2. set `ulimit` and `kernel.core_pattern` correctly
3. run the program
4. open your core dump with `gdb`, load the symbols, and run `bt`
5. try to figure out what happened!!

I was able using gdb to figure out that there was a C++ vtable entry that is pointing to some corrupt memory, which was somewhat helpful and helped me feel like I understood C++ a bit better. Maybe we'll talk more about how to use gdb to figure things out another day!