gdb Debugging Full Example (Tutorial): ncurses

09 Aug 2016

I'm a little frustrated with finding "gdb examples" online that show the commands but not their output. gdb is the GNU Debugger, the standard debugger on Linux. I was reminded of the lack of example output when watching the Give me 15 minutes and I'll change your view of GDB talk by Greg Law at CppCon 2015, which, thankfully, includes output! It's well worth the 15 minutes.

It also inspired me to share a full gdb debugging example, with output and every step involved, including dead ends. This isn't a particularly interesting or exotic issue, it's just a routine gdb debugging session. But it covers the basics and could serve as a tutorial of sorts, bearing in mind there's a lot more to gdb than I used here.

I'll be running the following commands as root, since I'm debugging a tool that needs root access (for now). Substitute non-root and sudo as desired. You also aren't expected to read through all this: I've enumerated each step so you can browse them and find ones of interest.

1. The Problem

The bcc collection of BPF tools had a pull request for cachetop, which uses a top-like display to show page cache statistics by process. Great! However, when I tested it, it hit a segfault:

```
# ./cachetop.py
Segmentation fault
```

Note that it says "Segmentation fault" and not "Segmentation fault (core dumped)". I'd like a core dump to debug this. (A core dump is a copy of process memory – the name coming from the era of magnetic core memory – and can be investigated using a debugger.)

Core dump analysis is one approach for debugging, but not the only one. I could run the program live in gdb to inspect the issue. I could also use an external tracer to grab data and stack traces on segfault events. We'll start with core dumps.

2. Fixing Core Dumps

I'll check the core dump settings:

```
# ulimit -c
0
# cat /proc/sys/kernel/core_pattern
core
```

`ulimit -c` shows the maximum size of core dumps created, and it's set to zero: disabling core dumps (for this process and its children).

The `/proc/.../core_pattern` is set to just "core", which will drop a core dump file called "core" in the current directory. That will be ok for now, but I'll show how to set this up for a global location:

```
# ulimit -c unlimited
# mkdir /var/cores
# echo "/var/cores/core.%e.%p" > /proc/sys/kernel/core_pattern
```

You can customize that core_pattern further; eg, `%h` for hostname and `%t` for time of dump. The options are documented in the Linux kernel source, under Documentation/sysctl/kernel.txt.

To make the core_pattern permanent, and survive reboots, you can set it via "kernel.core_pattern" in /etc/sysctl.conf.

Trying again:

```
# ./cachetop.py
Segmentation fault (core dumped)
# ls -lh /var/cores
total 19M
-rw------- 1 root root 20M Aug  7 22:15 core.python.30520
# file /var/cores/core.python.30520
/var/cores/core.python.30520: ELF 64-bit LSB core file x86-64, version 1 (SYSV), SVR4-style, from
```

That's better: we have our core dump.

3. Starting GDB

Now I'll run gdb with the target program location (using shell substitution, "`", although you should specify the full path unless you're sure that will work), and the core dump file:

```
# gdb `which python` /var/cores/core.python.30520
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
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 /usr/bin/python...(no debugging symbols found)...done.

warning: core file may not match specified executable file.
[New LWP 30520]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

warning: JITed object file architecture unknown is not compatible with target architecture i386:x
Core was generated by `python ./cachetop.py'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x00007f0a37aac40d in doupdate () from /lib/x86_64-linux-gnu/libncursesw.so.5
```

The last two lines are especially interesting: it tells us it's a segmentation fault in the `doupdate()` function from the libncursesw library. That's worth a quick web search in case it's a well-known issue. I took a quick look but didn't find a single common cause.

I already can guess what libncursesw is for, but if that were foreign to you, then being under "/lib" and ending in ".so.*" shows it's a shared library, which might have a man page, website, package description, etc.

```
# dpkg -l | grep libncursesw
ii  libncursesw5:amd64                  6.0+20160213-1ubuntu1                    amd64
     shared libraries for terminal handling (wide character support)
```

I happen to be debugging this on Ubuntu, but the Linux distro shouldn't matter for gdb usage.

4. Back Trace

Stack back traces show how we arrived at the point of fail, and are often enough to help identify a common problem. It's usually the first command I use in a gdb session: `bt` (short for `backtrace`):

```
(gdb) bt
#0  0x00007f0a37aac40d in doupdate () from /lib/x86_64-linux-gnu/libncursesw.so.5
#1  0x00007f0a37aa07e6 in wrefresh () from /lib/x86_64-linux-gnu/libncursesw.so.5
#2  0x00007f0a37a99616 in ?? () from /lib/x86_64-linux-gnu/libncursesw.so.5
#3  0x00007f0a37a9a325 in wgetch () from /lib/x86_64-linux-gnu/libncursesw.so.5
#4  0x00007f0a37cc6ec3 in ?? () from /usr/lib/python2.7/lib-dynload/_curses.x86_64-linux-gnu.so
#5  0x00000000004c4d5a in PyEval_EvalFrameEx ()
#6  0x00000000004c2e05 in PyEval_EvalCodeEx ()
#7  0x00000000004def08 in ?? ()
#8  0x00000000004b1153 in PyObject_Call ()
#9  0x00000000004c73ec in PyEval_EvalFrameEx ()
#10 0x00000000004c2e05 in PyEval_EvalCodeEx ()
#11 0x00000000004caf42 in PyEval_EvalFrameEx ()
#12 0x00000000004c2e05 in PyEval_EvalCodeEx ()
#13 0x00000000004c2ba9 in PyEval_EvalCode ()
#14 0x00000000004f20ef in ?? ()
#15 0x00000000004eca72 in PyRun_FileExFlags ()
#16 0x00000000004eb1f1 in PyRun_SimpleFileExFlags ()
#17 0x000000000049e18a in Py_Main ()
#18 0x00007f0a3be10830 in __libc_start_main (main=0x49daf0 <main>, argc=2, argv=0x7ffd33d94838, i
    stack_end=0x7ffd33d94828) at ../csu/libc-start.c:291
#19 0x000000000049da19 in _start ()
```

Read from bottom up, to go from parent to child. The "`??`" entries are where symbol translation failed. Stack walking – which produces the stack trace – can also fail. In that case you'll likely see a single valid frame, then a small number of bogus addresses. If symbols or stacks are too badly broken to make sense of the stack trace, then there are usually ways to fix it: installing debug info packages (giving gdb more symbols, and letting it do DWARF-based stack walks), or recompiling the software from source with frame pointers and debugging information (`-fno-omit-frame-pointer -g`). Many of the above "`??`" entries can be fixed by adding the python-dbg package.

This particular stack doesn't look very helpful: frames 5 to 17 (indexed on the left) are Python internals, although we can't see the Python methods (yet). Then frame 4 is the _curses library, then we're in libncursesw. Looks like `wgetch()`->`wrefresh()`->`doupdate()`. Just based on the names, I'd guess a window refresh. Why would that core dump?

5. Disassembly

I'll start by disassembling the function we segfaulted in, `doupdate()`:

```
(gdb) disas doupdate
Dump of assembler code for function doupdate:
   0x00007f0a37aac2e0 <+0>:   push   %r15
   0x00007f0a37aac2e2 <+2>:   push   %r14
   0x00007f0a37aac2e4 <+4>:   push   %r13
   0x00007f0a37aac2e6 <+6>:   push   %r12
   0x00007f0a37aac2e8 <+8>:   push   %rbp
   0x00007f0a37aac2e9 <+9>:   push   %rbx
   0x00007f0a37aac2ea <+10>:  sub    $0xc8,%rsp
[...]
---Type <return> to continue, or q <return> to quit---
[...]
   0x00007f0a37aac3f7 <+279>: cmpb   $0x0,0x21(%rcx)
   0x00007f0a37aac3fb <+283>: je     0x7f0a37aacc3b <doupdate+2395>
   0x00007f0a37aac401 <+289>: mov    0x20cb68(%rip),%rax        # 0x7f0a37cb8f70
   0x00007f0a37aac408 <+296>: mov    (%rax),%rsi
   0x00007f0a37aac40b <+299>: xor    %eax,%eax
=> 0x00007f0a37aac40d <+301>: mov    0x10(%rsi),%rdi
   0x00007f0a37aac411 <+305>: cmpb   $0x0,0x1c(%rdi)
   0x00007f0a37aac415 <+309>: jne    0x7f0a37aac6f7 <doupdate+1047>
   0x00007f0a37aac41b <+315>: movswl 0x4(%rcx),%ecx
   0x00007f0a37aac41f <+319>: movswl 0x74(%rdx),%edi
   0x00007f0a37aac423 <+323>: mov    %rax,0x40(%rsp)
[...]
```

Output truncated. (I could also have just typed "`disas`" and it would have defaulted to `doupdate`.)

The arrow "=>" is pointing to our segfault address, which is doing a `mov 0x10(%rsi),%rdi`: a move from the memory pointed to in the %rsi register plus an offset of 0x10, to the %rdi register. I'll check the state of the registers next.

6. Check Registers

Printing register state using `i r` (short for `info registers`):

```
(gdb) i r
rax            0x0  0
rbx            0x1993060    26816608
rcx            0x19902a0    26804896
rdx            0x19ce7d0    27060176
rsi            0x0  0
rdi            0x19ce7d0    27060176
rbp            0x7f0a3848eb10   0x7f0a3848eb10 <SP>
rsp            0x7ffd33d93c00   0x7ffd33d93c00
r8             0x7f0a37cb93e0   139681862489056
r9             0x0  0
r10            0x8  8
r11            0x202    514
r12            0x0  0
r13            0x0  0
r14            0x7f0a3848eb10   139681870703376
r15            0x19ce7d0    27060176
rip            0x7f0a37aac40d   0x7f0a37aac40d <doupdate+301>
eflags         0x10246  [ PF ZF IF RF ]
cs             0x33 51
ss             0x2b 43
ds             0x0  0
es             0x0  0
fs             0x0  0
gs             0x0  0
```

Well, %rsi is zero. There's our problem! Zero is unlikely a valid address, and this type of segfault is a common software bug: dereferencing an uninitialized or NULL pointer.

7. Memory Mappings

You can double check if zero is valid using `i proc m` (short for `info proc mappings`):

```
(gdb) i proc m
Mapped address spaces:

      Start Addr           End Addr       Size     Offset objfile
        0x400000           0x6e7000   0x2e7000        0x0 /usr/bin/python2.7
        0x8e6000           0x8e8000     0x2000   0x2e6000 /usr/bin/python2.7
        0x8e8000           0x95f000    0x77000   0x2e8000 /usr/bin/python2.7
  0x7f0a37a8b000     0x7f0a37ab8000    0x2d000        0x0 /lib/x86_64-linux-gnu/libncursesw.so.5.
  0x7f0a37ab8000     0x7f0a37cb8000   0x200000    0x2d000 /lib/x86_64-linux-gnu/libncursesw.so.5.
  0x7f0a37cb8000     0x7f0a37cb9000     0x1000    0x2d000 /lib/x86_64-linux-gnu/libncursesw.so.5.
  0x7f0a37cb9000     0x7f0a37cba000     0x1000    0x2e000 /lib/x86_64-linux-gnu/libncursesw.so.5.
  0x7f0a37cba000     0x7f0a37ccd000    0x13000        0x0 /usr/lib/python2.7/lib-dynload/_curses.
  0x7f0a37ccd000     0x7f0a37ecc000   0x1ff000    0x13000 /usr/lib/python2.7/lib-dynload/_curses.
  0x7f0a37ecc000     0x7f0a37ecd000     0x1000    0x12000 /usr/lib/python2.7/lib-dynload/_curses.
  0x7f0a37ecd000     0x7f0a37ecf000     0x2000    0x13000 /usr/lib/python2.7/lib-dynload/_curses.
  0x7f0a38050000     0x7f0a38066000    0x16000        0x0 /lib/x86_64-linux-gnu/libgcc_s.so.1
  0x7f0a38066000     0x7f0a38265000   0x1ff000    0x16000 /lib/x86_64-linux-gnu/libgcc_s.so.1
  0x7f0a38265000     0x7f0a38266000     0x1000    0x15000 /lib/x86_64-linux-gnu/libgcc_s.so.1
  0x7f0a38266000     0x7f0a3828b000    0x25000        0x0 /lib/x86_64-linux-gnu/libtinfo.so.5.9
  0x7f0a3828b000     0x7f0a3848a000   0x1ff000    0x25000 /lib/x86_64-linux-gnu/libtinfo.so.5.9
[...]
```

The first valid virtual address is 0x400000. Anything below that is invalid, and if referenced, will trigger a segmentation fault.

At this point there are several different ways to dig further. I'll start with some instruction stepping.

8. Breakpoints

Back to the disassembly:

```
   0x00007f0a37aac401 <+289>:   mov    0x20cb68(%rip),%rax        # 0x7f0a37cb8f70
   0x00007f0a37aac408 <+296>:   mov    (%rax),%rsi
   0x00007f0a37aac40b <+299>:   xor    %eax,%eax
=> 0x00007f0a37aac40d <+301>:   mov    0x10(%rsi),%rdi
```

Reading these four instructions: it looks like it's pulling something from the stack into %rax, then dereferencing %rax into %rsi, the setting %eax to zero (the xor is an optimization, instead of doing a mov of $0), and then we dereference %rsi with an offset, although we know %rsi is zero. This sequence is for walking data structures. Maybe %rax would be interesting, but it's been set to zero by the prior instruction, so we can't see it in the core dump register state.

I can set a breakpoint on doupdate+289, then single-step through each instruction to see how the registers are set and change. First, I need to launch gdb so that we're executing the program live:

```
# gdb `which python`
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
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 /usr/bin/python...(no debugging symbols found)...done.
```

Now to set the breakpoint using `b` (short for `break`):

```
(gdb) b *doupdate + 289
No symbol table is loaded.  Use the "file" command.
```

Oops. I wanted to show this error to explain why we often start out with a breakpoint on `main`, at which point the symbols are likely loaded, and then setting the real breakpoint of interest. I'll go straight to `doupdate` function entry, run the problem, then set the offset breakpoint once it hits the function:

```
(gdb) b doupdate
Function "doupdate" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (doupdate) pending.
(gdb) r cachetop.py
Starting program: /usr/bin/python cachetop.py
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
warning: JITed object file architecture unknown is not compatible with target architecture i386:x

Breakpoint 1, 0x00007ffff34ad2e0 in doupdate () from /lib/x86_64-linux-gnu/libncursesw.so.5
(gdb) b *doupdate + 289
Breakpoint 2 at 0x7ffff34ad401
(gdb) c
Continuing.

Breakpoint 2, 0x00007ffff34ad401 in doupdate () from /lib/x86_64-linux-gnu/libncursesw.so.5
```

We've arrived at our breakpoint.

If you haven't done this before, the `r` (`run`) command takes arguments that will be passed to the gdb target we specified earlier on the command line (python). So this ends up running "python cachetop.py".

9. Stepping

I'll step one instruction (`si`, short for `stepi`) then inspect registers:

```
(gdb) si
0x00007ffff34ad408 in doupdate () from /lib/x86_64-linux-gnu/libncursesw.so.5
(gdb) i r
rax            0x7ffff3e8f948   140737285519688
rbx            0xaea060 11444320
rcx            0xae72a0 11432608
rdx            0xa403d0 10748880
rsi            0x7ffff7ea8e10   140737352732176
rdi            0xa403d0 10748880
rbp            0x7ffff3e8fb10   0x7ffff3e8fb10 <SP>
rsp            0x7fffffffd390   0x7fffffffd390
r8             0x7ffff36ba3e0   140737277305824
r9             0x0  0
r10            0x8  8
r11            0x202    514
r12            0x0  0
r13            0x0  0
r14            0x7ffff3e8fb10   140737285520144
r15            0xa403d0 10748880
rip            0x7ffff34ad408   0x7ffff34ad408 <doupdate+296>
eflags         0x202    [ IF ]
cs             0x33 51
ss             0x2b 43
ds             0x0  0
es             0x0  0
fs             0x0  0
gs             0x0  0
(gdb) p/a 0x7ffff3e8f948
$1 = 0x7ffff3e8f948 <cur_term>
```

Another clue. So the NULL pointer we're dereferencing looks like it's in a symbol called "`cur_term`" (`p/a` is short for `print/a`, where "`/a`" means format as an address). Given this is ncurses, is our TERM environment set to something odd?

```
# echo $TERM
xterm-256color
```

I tried setting that to vt100 and running the program, but it hit the same segfault.

Note that I've inspected just the first invocation of `doupdate()`, but it could be called multiple times, and the issue may be a later invocation. I can step through each by running `c` (short for `continue`). That will be ok if it's only called a few times, but if it's called a few thousand times I'll want a different approach. (I'll get back to this in section 15.)

10. Reverse Stepping

gdb has a great feature called reverse stepping, which Greg Law included in his talk. Here's an example.

I'll start a python session again, to show this from the beginning:

```
# gdb `which python`
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
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:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /usr/bin/python...(no debugging symbols found)...done.
```

Now I'll set a breakpoint on `doupdate` as before, but once it's hit, I'll enable recording, then continue the program and let it crash. Recording adds considerable overhead, so I don't want to add it on `main`.

```
(gdb) b doupdate
Function "doupdate" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (doupdate) pending.
(gdb) r cachetop.py
Starting program: /usr/bin/python cachetop.py
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
warning: JITed object file architecture unknown is not compatible with target architecture i386:x

Breakpoint 1, 0x00007ffff34ad2e0 in doupdate () from /lib/x86_64-linux-gnu/libncursesw.so.5
(gdb) record
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff34ad40d in doupdate () from /lib/x86_64-linux-gnu/libncursesw.so.5
```

At this point I can reverse-step through lines or instructions. It works by playing back register state from our recording. I'll move back in time two instructions, then print registers:

```
(gdb) reverse-stepi
0x00007ffff34ad40d in doupdate () from /lib/x86_64-linux-gnu/libncursesw.so.5
(gdb) reverse-stepi
0x00007ffff34ad40b in doupdate () from /lib/x86_64-linux-gnu/libncursesw.so.5
(gdb) i r
rax            0x7ffff3e8f948   140737285519688
rbx            0xaea060 11444320
rcx            0xae72a0 11432608
rdx            0xa403d0 10748880
rsi            0x0  0
rdi            0xa403d0 10748880
rbp            0x7ffff3e8fb10   0x7ffff3e8fb10 <SP>
rsp            0x7fffffffd390   0x7fffffffd390
r8             0x7ffff36ba3e0   140737277305824
r9             0x0  0
r10            0x8  8
r11            0x302    770
r12            0x0  0
r13            0x0  0
r14            0x7ffff3e8fb10   140737285520144
r15            0xa403d0 10748880
rip            0x7ffff34ad40b   0x7ffff34ad40b <doupdate+299>
eflags         0x202    [ IF ]
cs             0x33 51
ss             0x2b 43
ds             0x0  0
es             0x0  0
fs             0x0  0
gs             0x0  0
(gdb) p/a 0x7ffff3e8f948
$1 = 0x7ffff3e8f948 <cur_term>
```

So, back to finding the "cur_term" clue. I really want to read the source code at this point, but I'll start with debug info.

11. Debug Info

This is libncursesw, and I don't have debug info installed (Ubuntu):

```
# apt-cache search libncursesw
libncursesw5 - shared libraries for terminal handling (wide character support)
libncursesw5-dbg - debugging/profiling libraries for ncursesw
libncursesw5-dev - developer's libraries for ncursesw
# dpkg -l | grep libncursesw
ii  libncursesw5:amd64                  6.0+20160213-1ubuntu1                        amd64        sha
```

I'll add that:

```
# apt-get install -y libncursesw5-dbg
Reading package lists... Done
Building dependency tree
Reading state information... Done
[...]
After this operation, 2,488 kB of additional disk space will be used.
Get:1 http://us-west-1.ec2.archive.ubuntu.com/ubuntu xenial/main amd64 libncursesw5-dbg amd64 6.0
Fetched 729 kB in 0s (865 kB/s)
Selecting previously unselected package libncursesw5-dbg.
(Reading database ... 200094 files and directories currently installed.)
Preparing to unpack .../libncursesw5-dbg_6.0+20160213-1ubuntu1_amd64.deb ...
Unpacking libncursesw5-dbg (6.0+20160213-1ubuntu1) ...
Setting up libncursesw5-dbg (6.0+20160213-1ubuntu1) ...
# dpkg -l | grep libncursesw
ii  libncursesw5:amd64                  6.0+20160213-1ubuntu1                    amd64        sha
ii  libncursesw5-dbg                    6.0+20160213-1ubuntu1                    amd64        deb
```

Good, those versions match. So how does our segfault look now?

```
# gdb `which python` /var/cores/core.python.30520
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
[...]
warning: JITed object file architecture unknown is not compatible with target architecture i386:x
Core was generated by `python ./cachetop.py'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  ClrBlank (win=0x1993060) at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tty/tty_update
1129        if (back_color_erase)
(gdb) bt
#0  ClrBlank (win=0x1993060) at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tty/tty_update
#1  ClrUpdate () at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tty/tty_update.c:1147
#2  doupdate () at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tty/tty_update.c:1010
#3  0x00007f0a37aa07e6 in wrefresh (win=win@entry=0x1993060) at /build/ncurses-pKZ1BN/ncurses-6.0
#4  0x00007f0a37a99499 in recur_wrefresh (win=win@entry=0x1993060) at /build/ncurses-pKZ1BN/ncurs
#5  0x00007f0a37a99616 in _nc_wgetch (win=win@entry=0x1993060, result=result@entry=0x7ffd33d93e24
    at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/base/lib_getch.c:491
#6  0x00007f0a37a9a325 in wgetch (win=0x1993060) at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/nc
#7  0x00007f0a37cc6ec3 in ?? () from /usr/lib/python2.7/lib-dynload/_curses.x86_64-linux-gnu.so
#8  0x00000000004c4d5a in PyEval_EvalFrameEx ()
#9  0x00000000004c2e05 in PyEval_EvalCodeEx ()
#10 0x00000000004def08 in ?? ()
#11 0x00000000004b1153 in PyObject_Call ()
#12 0x00000000004c73ec in PyEval_EvalFrameEx ()
#13 0x00000000004c2e05 in PyEval_EvalCodeEx ()
#14 0x00000000004caf42 in PyEval_EvalFrameEx ()
#15 0x00000000004c2e05 in PyEval_EvalCodeEx ()
#16 0x00000000004c2ba9 in PyEval_EvalCode ()
#17 0x00000000004f20ef in ?? ()
#18 0x00000000004eca72 in PyRun_FileExFlags ()
#19 0x00000000004eb1f1 in PyRun_SimpleFileExFlags ()
#20 0x000000000049e18a in Py_Main ()
#21 0x00007f0a3be10830 in __libc_start_main (main=0x49daf0 <main>, argc=2, argv=0x7ffd33d94838, i
    stack_end=0x7ffd33d94828) at ../csu/libc-start.c:291
#22 0x000000000049da19 in _start ()
```

The stack trace looks a bit different: we aren't really in `doupdate()`, but `ClrBlank()`, which has been inlined in `ClrUpdate()`, and inlined in `doupdate()`.

Now I really want to see source.

12. Source Code

With the debug info package installed, gdb can list the source along with the assembly:

```
(gdb) disas/s
Dump of assembler code for function doupdate:
/build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tty/tty_update.c:
759 {
   0x00007f0a37aac2e0 <+0>:   push   %r15
   0x00007f0a37aac2e2 <+2>:   push   %r14
   0x00007f0a37aac2e4 <+4>:   push   %r13
   0x00007f0a37aac2e6 <+6>:   push   %r12
[...]
   0x00007f0a37aac3dd <+253>: jne    0x7f0a37aac6ca <doupdate+1002>

1009        if (CurScreen(SP_PARM)->_clear || NewScreen(SP_PARM)->_clear) {   /* force refresh ?
   0x00007f0a37aac3e3 <+259>: mov    0x80(%rdx),%rax
   0x00007f0a37aac3ea <+266>: mov    0x88(%rdx),%rcx
   0x00007f0a37aac3f1 <+273>: cmpb   $0x0,0x21(%rax)
   0x00007f0a37aac3f5 <+277>: jne    0x7f0a37aac401 <doupdate+289>
   0x00007f0a37aac3f7 <+279>: cmpb   $0x0,0x21(%rcx)
   0x00007f0a37aac3fb <+283>: je     0x7f0a37aacc3b <doupdate+2395>

1129        if (back_color_erase)
   0x00007f0a37aac401 <+289>: mov    0x20cb68(%rip),%rax        # 0x7f0a37cb8f70
   0x00007f0a37aac408 <+296>: mov    (%rax),%rsi

1128        NCURSES_CH_T blank = blankchar;
   0x00007f0a37aac40b <+299>: xor    %eax,%eax

1129        if (back_color_erase)
=> 0x00007f0a37aac40d <+301>: mov    0x10(%rsi),%rdi
   0x00007f0a37aac411 <+305>: cmpb   $0x0,0x1c(%rdi)
   0x00007f0a37aac415 <+309>: jne    0x7f0a37aac6f7 <doupdate+1047>
```

Great! See the arrow "=>" and the line of code above it. So we're segfaulting on "`if (back_color_erase)`"? That doesn't seem possible. (A segfault would be due to a memory dereference, which in C would be `a->b` or `*a`, but in this case it's just "`back_color_erase`", which looks like it's accessing an ordinary variable and not dereferencing memory.)

At this point I double checked that I had the right debug info version, and re-ran the application to segfault it in a live gdb session. Same place.

Is there something special about `back_color_erase`? We're in `ClrBlank()`, so I'll list that source code:

```
(gdb) list ClrBlank
1124
1125    static NCURSES_INLINE NCURSES_CH_T
1126    ClrBlank(NCURSES_SP_DCLx WINDOW *win)
1127    {
1128        NCURSES_CH_T blank = blankchar;
1129        if (back_color_erase)
1130        AddAttr(blank, (AttrOf(BCE_BKGD(SP_PARM, win)) & BCE_ATTRS));
1131        return blank;
1132    }
1133
```

Ah, that's not defined in the function, so it's a global?

13. TUI

It's worth showing how this looks in the gdb text user interface (TUI), which I haven't used that much but was inspired after seeing Greg's talk.

You can launch it using `--tui`:

```
# gdb --tui `which python` /var/cores/core.python.30520
    ┌───────────────────────────────────────────────────────────────────────────┐
    │                                                                           │
    │                                                                           │
    │                                                                           │
    │                                                                           │
    │                                                                           │
    │             [ No Source Available ]                                       │
    │                                                                           │
    │                                                                           │
    │                                                                           │
    │                                                                           │
    │                                                                           │
    └───────────────────────────────────────────────────────────────────────────┘
None No process In:                                                  L??   PC: ??
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
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  to continue, or q  to quit---
```

It's complaining about no Python source. I could fix that, but we're crashing in libncursesw. Hitting enter lets it finish loading, at which point it loads the libncursesw debug info source code:

```
    ┌──/build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tty/tty_update.c──────┐
    │1124                                                                       │
    │1125    static NCURSES_INLINE NCURSES_CH_T                                 │
    │1126    ClrBlank(NCURSES_SP_DCLx WINDOW *win)                              │
    │1127    {                                                                  │
    │1128        NCURSES_CH_T blank = blankchar;                                │
  > │1129        if (back_color_erase)                                          │
    │1130            AddAttr(blank, (AttrOf(BCE_BKGD(SP_PARM, win)) & BCE_ATTRS)│
    │1131        return blank;                                                  │
    │1132    }                                                                  │
    │1133                                                                       │
    │1134    /*                                                                 │
    │1135    **      ClrUpdate()                                                │
    │1136    **                                                                 │
    └───────────────────────────────────────────────────────────────────────────┘
multi-thre Thread 0x7f0a3c5e87 In: doupdate            L1129 PC: 0x7f0a37aac40d
warning: JITed object file architecture unknown is not compatible with target ar
chitecture i386:x86-64.
---Type <return> to continue, or q <return> to quit---
Core was generated by `python ./cachetop.py'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  ClrBlank (win=0x1993060)
    at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tty/tty_update.c:1129
(gdb)
```

Awesome!

The arrow ">" shows the line of code that we crashed in. It gets even better: with the `layout split` command we can follow the source with the disassembly in separate windows:

```
  ┌──/build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tty/tty_update.c──────┐
 >│1129        if (back_color_erase)                                            │
  │1130            AddAttr(blank, (AttrOf(BCE_BKGD(SP_PARM, win)) & BCE_ATTRS)  │
  │1131        return blank;                                                    │
  │1132    }                                                                    │
  │1133                                                                         │
  │1134    /*                                                                   │
  │1135    **      ClrUpdate()                                                  │
  └─────────────────────────────────────────────────────────────────────────────┘
 >│0x7f0a37aac40d <doupdate+301>   mov    0x10(%rsi),%rdi                      │
  │0x7f0a37aac411 <doupdate+305>   cmpb   $0x0,0x1c(%rdi)                      │
  │0x7f0a37aac415 <doupdate+309>   jne    0x7f0a37aac6f7 <doupdate+1047>       │
  │0x7f0a37aac41b <doupdate+315>   movswl 0x4(%rcx),%ecx                       │
  │0x7f0a37aac41f <doupdate+319>   movswl 0x74(%rdx),%edi                      │
  │0x7f0a37aac423 <doupdate+323>   mov    %rax,0x40(%rsp)                      │
  │0x7f0a37aac428 <doupdate+328>   movl   $0x20,0x48(%rsp)                     │
  │0x7f0a37aac430 <doupdate+336>   movl   $0x0,0x4c(%rsp)                      │
  └─────────────────────────────────────────────────────────────────────────────┘
multi-thre Thread 0x7f0a3c5e87 In: doupdate            L1129 PC: 0x7f0a37aac40d

chitecture i386:x86-64.
Core was generated by `python ./cachetop.py'.
Program terminated with signal SIGSEGV, Segmentation fault.
---Type <return> to continue, or q <return> to quit---
#0  ClrBlank (win=0x1993060)
    at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tty/tty_update.c:1129
(gdb) layout split
```

Greg demonstrated this with reverse stepping, so you can imagine following both code and assembly execution at the same time (I'd need a video to demonstrate that here).

14. External: cscope

I still want to learn more about `back_color_erase`, and I could try gdb's `search` command, but I've found I'm quicker using an external tool: cscope. cscope is a text-based source code browser from Bell Labs in the 1980's. If you have a modern IDE that you prefer, use that instead.

Setting up cscope:

```
# apt-get install -y cscope
# wget http://archive.ubuntu.com/ubuntu/pool/main/n/ncurses/ncurses_6.0+20160213.orig.tar.gz
# tar xvf ncurses_6.0+20160213.orig.tar.gz
# cd ncurses-6.0-20160213
# cscope -bqR
# cscope -dq
```

`cscope -bqR` builds the lookup database. `cscope -dq` then launches cscope.

Searching for `back_color_erase` definition:

```
Cscope version 15.8b                                         Press the ? key for help











Find this C symbol:
Find this global definition: back_color_erase
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Find assignments to this symbol:
```

Hitting enter:

```
[...]
#define non_dest_scroll_region         CUR Booleans[26]
#define can_change                     CUR Booleans[27]
#define back_color_erase               CUR Booleans[28]
#define hue_lightness_saturation       CUR Booleans[29]
#define col_addr_glitch                CUR Booleans[30]
#define cr_cancels_micro_mode          CUR Booleans[31]
[...]
```

Oh, a `#define`. (They could have at least capitalized it, as is a common style with `#define`'s.)

Ok, so what's `CUR`? Looking up definitions in cscope is a breeze.

```
#define CUR cur_term->type.
```

At least that `#define` is capitalized!

We'd found `cur_term` earlier, by stepping instructions and examining registers. What is it?

```
#if 0 && !0
extern NCURSES_EXPORT_VAR(TERMINAL *) cur_term;
#elif 0
NCURSES_WRAPPED_VAR(TERMINAL *, cur_term);
#define cur_term   NCURSES_PUBLIC_VAR(cur_term())
#else
extern NCURSES_EXPORT_VAR(TERMINAL *) cur_term;
#endif
```

cscope read /usr/include/term.h for this. So, more macros. I had to highlight in bold the line of code I think is taking effect there. Why is there an "`if 0 && !0 ... elif 0`"? I don't know (I'd need to read more source). Sometimes programmers use "`#if 0`" around debug code they want to disable in production, however, this looks auto-generated.

Searching for `NCURSES_EXPORT_VAR` finds:

```
#  define NCURSES_EXPORT_VAR(type) NCURSES_IMPEXP type
```

... and `NCURSES_IMPEXP`:

```
/* Take care of non-cygwin platforms */
#if !defined(NCURSES_IMPEXP)
#  define NCURSES_IMPEXP /* nothing */
#endif
#if !defined(NCURSES_API)
#  define NCURSES_API /* nothing */
#endif
#if !defined(NCURSES_EXPORT)
#  define NCURSES_EXPORT(type) NCURSES_IMPEXP type NCURSES_API
#endif
#if !defined(NCURSES_EXPORT_VAR)
#  define NCURSES_EXPORT_VAR(type) NCURSES_IMPEXP type
#endif
```

... and `TERMINAL` was:

```
typedef struct term {       /* describe an actual terminal */
    TERMTYPE    type;       /* terminal type description */
    short   Filedes;    /* file description being written to */
    TTY     Ottyb,      /* original state of the terminal */
        Nttyb;      /* current state of the terminal */
    int     _baudrate;  /* used to compute padding */
    char *      _termname;      /* used for termname() */
} TERMINAL;
```

Gah! Now `TERMINAL` is capitalized. Along with the macros, this code is not that easy to follow...

Ok, who actually sets `cur_term`? Remember our problem is that it's set to zero, maybe because it's uninitialized or explicitly set. Browsing the code paths that set it might provide more clues, to help answer why it isn't being set, or why it is set to zero. Using the first option in cscope:

```
Find this C symbol: cur_term
Find this global definition:
Find functions called by this function:
Find functions calling this function:
[...]
```

And browsing the entries quickly finds:

```
NCURSES_EXPORT(TERMINAL *)
NCURSES_SP_NAME(set_curterm) (NCURSES_SP_DCLx TERMINAL * termp)
{
    TERMINAL *oldterm;

    T((T_CALLED("set_curterm(%p)"), (void *) termp));

    _nc_lock_global(curses);
    oldterm = cur_term;
    if (SP_PARM)
    SP_PARM->_term = termp;
#if USE_REENTRANT
    CurTerm = termp;
#else
    cur_term = termp;
#endif
```

I added the highlighting. Even the function name is wrapped in a macro. But at least we've found how `cur_term` is set: via `set_curterm()`. Maybe that isn't being called?

15. External: perf-tools/ftrace/uprobes

I'll cover using gdb for this in a moment, but I can't help trying the uprobe tool from my perf-tools collection, which uses Linux ftrace and uprobes. One advantage of using tracers is that they don't pause the target process, like gdb does (although that doesn't matter for this cachetop.py example). Another advantage is that I can trace a few events or a few thousand just as easily.

I should be able to trace calls to `set_curterm()` in libncursesw, and even print the first argument:

```
# /apps/perf-tools/bin/uprobe 'p:/lib/x86_64-linux-gnu/libncursesw.so.5:set_curterm %di'
ERROR: missing symbol "set_curterm" in /lib/x86_64-linux-gnu/libncursesw.so.5
```

Well, that didn't work. Where is `set_curterm()`? There are lots of ways to find it, like gdb or objdump:

```
(gdb) info symbol set_curterm
set_curterm in section .text of /lib/x86_64-linux-gnu/libtinfo.so.5

# objdump -tT /lib/x86_64-linux-gnu/libncursesw.so.5 | grep cur_term
0000000000000000      DO *UND*  0000000000000000  NCURSES_TINFO_5.0.19991023 cur_term
# objdump -tT /lib/x86_64-linux-gnu/libtinfo.so.5 | grep cur_term
0000000000228948 g    DO .bss   0000000000000008  NCURSES_TINFO_5.0.19991023 cur_term
```

gdb works better. Plus if I took a closer look at the source, I would have noticed it was building it for libtinfo.

Trying to trace `set_curterm()` in libtinfo:

```
# /apps/perf-tools/bin/uprobe 'p:/lib/x86_64-linux-gnu/libtinfo.so.5:set_curterm %di'
Tracing uprobe set_curterm (p:set_curterm /lib/x86_64-linux-gnu/libtinfo.so.5:0xfa80 %di). Ctrl-C
          python-31617 [007] d... 24236402.719959: set_curterm: (0x7f116fcc2a80) arg1=0x1345d70
          python-31617 [007] d... 24236402.720033: set_curterm: (0x7f116fcc2a80) arg1=0x13a22e0
          python-31617 [007] d... 24236402.723804: set_curterm: (0x7f116fcc2a80) arg1=0x14cdfa0
          python-31617 [007] d... 24236402.723838: set_curterm: (0x7f116fcc2a80) arg1=0x0
^C
```

That works. So `set_curterm()` *is* called, and has been called four times. The last time it was passed zero, which sounds like it could be the problem.

If you're wondering how I knew the %di register was the first argument, then it comes from the AMD64/x86_64 ABI (and the assumption that this compiled library is ABI compliant). Here's a reminder:

```
# man syscall
[...]
       arch/ABI      arg1  arg2  arg3  arg4  arg5  arg6  arg7  Notes
       ──────────────────────────────────────────────────────────────
       arm/OABI      a1    a2    a3    a4    v1    v2    v3
       arm/EABI      r0    r1    r2    r3    r4    r5    r6
       arm64         x0    x1    x2    x3    x4    x5    -
       blackfin      R0    R1    R2    R3    R4    R5    -
       i386          ebx   ecx   edx   esi   edi   ebp   -
       ia64          out0  out1  out2  out3  out4  out5  -
       mips/o32      a0    a1    a2    a3    -     -     -     See below
       mips/n32,64   a0    a1    a2    a3    a4    a5    -
       parisc        r26   r25   r24   r23   r22   r21   -
       s390          r2    r3    r4    r5    r6    r7    -
       s390x         r2    r3    r4    r5    r6    r7    -
       sparc/32      o0    o1    o2    o3    o4    o5    -
       sparc/64      o0    o1    o2    o3    o4    o5    -
       x86_64        rdi   rsi   rdx   r10   r8    r9    -
[...]
```

I'd also like to see a stack trace for arg1=0x0 invocation, but this ftrace tool doesn't support stack traces yet.

16. External: bcc/BPF

Since we're debugging a bcc tool, cachetop.py, it's worth noting that bcc's trace.py has capabilities like my older uprobe tool:

```
# ./trace.py 'p:tinfo:set_curterm "%d", arg1'
TIME     PID    COMM         FUNC             -
01:00:20 31698  python       set_curterm      38018416
01:00:20 31698  python       set_curterm      38396640
01:00:20 31698  python       set_curterm      39624608
01:00:20 31698  python       set_curterm      0
```

Yes, we're using bcc to debug bcc!

If you are new to bcc, it's worth checking it out. It provides Python and lua interfaces for the new BPF tracing features that are in the Linux 4.x series. In short, it allows lots of performance tools that were previously impossible or prohibitively expensive to run. I've posted instructions for running it on Ubuntu Xenial.

The bcc trace.py tool should have a switch for printing user stack traces, since the kernel now has BPF stack capabilities as of Linux 4.6, although at the time of writing we haven't added this switch yet.

17. More Breakpoints

I should really have used gdb breakpoints on `set_curterm()` to start with, but I hope that was an interesting detour through ftrace and BPF.

Back to live running mode:

```
# gdb `which python`
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
[...]
(gdb) b set_curterm
Function "set_curterm" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (set_curterm) pending.
(gdb) r cachetop.py
Starting program: /usr/bin/python cachetop.py
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, set_curterm (termp=termp@entry=0xa43150) at /build/ncurses-pKZ1BN/ncurses-6.0+20160
80  {
(gdb) c
Continuing.

Breakpoint 1, set_curterm (termp=termp@entry=0xab5870) at /build/ncurses-pKZ1BN/ncurses-6.0+20160
80  {
(gdb) c
Continuing.

Breakpoint 1, set_curterm (termp=termp@entry=0xbecb90) at /build/ncurses-pKZ1BN/ncurses-6.0+20160
80  {
(gdb) c
Continuing.

Breakpoint 1, set_curterm (termp=0x0) at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tinfo
80  {
```

Ok, at this breakpoint we can see that `set_curterm()` is being invoked with a termp=0x0 argument, thanks to debuginfo for that information. If I didn't have debuginfo, I could just print the registers on each breakpoint.

I'll print the stack trace so that we can see *who* was setting `curterm` to 0.

```
(gdb) bt
#0  set_curterm (termp=0x0) at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tinfo/lib_cur_t
#1  0x00007ffff5a44e75 in llvm::sys::Process::FileDescriptorHasColors(int) () from /usr/lib/x86_6
#2  0x00007ffff45cabb8 in clang::driver::tools::Clang::ConstructJob(clang::driver::Compilation&,
#3  0x00007ffff456ffa5 in clang::driver::Driver::BuildJobsForAction(clang::driver::Compilation&,
#4  0x00007ffff4570501 in clang::driver::Driver::BuildJobs(clang::driver::Compilation&) const ()
#5  0x00007ffff457224a in clang::driver::Driver::BuildCompilation(llvm::ArrayRef<char const*>) ()
#6  0x00007ffff4396cda in ebpf::ClangLoader::parse(std::unique_ptr<llvm::Module, std::default_del
#7  0x00007ffff4344314 in ebpf::BPFModule::load_cfile(std::__cxx11::basic_string<char, std::char_
   from /usr/lib/x86_64-linux-gnu/libbcc.so.0
#8  0x00007ffff4349e5e in ebpf::BPFModule::load_string(std::__cxx11::basic_string<char, std::char
   from /usr/lib/x86_64-linux-gnu/libbcc.so.0
#9  0x00007ffff43430c8 in bpf_module_create_c_from_string () from /usr/lib/x86_64-linux-gnu/libbc
#10 0x00007ffff690ae40 in ffi_call_unix64 () from /usr/lib/x86_64-linux-gnu/libffi.so.6
#11 0x00007ffff690a8ab in ffi_call () from /usr/lib/x86_64-linux-gnu/libffi.so.6
#12 0x00007ffff6b1a68c in _ctypes_callproc () from /usr/lib/python2.7/lib-dynload/_ctypes.x86_64-
#13 0x00007ffff6b1ed82 in ?? () from /usr/lib/python2.7/lib-dynload/_ctypes.x86_64-linux-gnu.so
#14 0x00000000004b1153 in PyObject_Call ()
#15 0x00000000004ca5ca in PyEval_EvalFrameEx ()
#16 0x00000000004c2e05 in PyEval_EvalCodeEx ()
#17 0x00000000004def08 in ?? ()
#18 0x00000000004b1153 in PyObject_Call ()
#19 0x00000000004f4c3e in ?? ()
#20 0x00000000004b1153 in PyObject_Call ()
#21 0x00000000004f49b7 in ?? ()
#22 0x00000000004b6e2c in ?? ()
#23 0x00000000004b1153 in PyObject_Call ()
#24 0x00000000004ca5ca in PyEval_EvalFrameEx ()
#25 0x00000000004c2e05 in PyEval_EvalCodeEx ()
#26 0x00000000004def08 in ?? ()
#27 0x00000000004b1153 in PyObject_Call ()
#28 0x00000000004c73ec in PyEval_EvalFrameEx ()
#29 0x00000000004c2e05 in PyEval_EvalCodeEx ()
#30 0x00000000004caf42 in PyEval_EvalFrameEx ()
#31 0x00000000004c2e05 in PyEval_EvalCodeEx ()
#32 0x00000000004c2ba9 in PyEval_EvalCode ()
#33 0x00000000004f20ef in ?? ()
#34 0x00000000004eca72 in PyRun_FileExFlags ()
#35 0x00000000004eb1f1 in PyRun_SimpleFileExFlags ()
#36 0x000000000049e18a in Py_Main ()
#37 0x00007ffff7811830 in __libc_start_main (main=0x49daf0 <main>, argc=2, argv=0x7fffffffdfb8, i
    stack_end=0x7fffffffdfa8) at ../csu/libc-start.c:291
#38 0x000000000049da19 in _start ()
```

Ok, more clues...I think. We're in `llvm::sys::Process::FileDescriptorHasColors()`. The llvm compiler?

18. External: cscope, take 2

More source code browsing using cscope, this time in llvm. The FileDescriptorHasColors() function has:

```
static bool terminalHasColors(int fd) {
[...]
  // Now extract the structure allocated by setupterm and free its memory
  // through a really silly dance.
  struct term *termp = set_curterm((struct term *)nullptr);
  (void)del_curterm(termp); // Drop any errors here.
```

Here's what that code used to be in an earlier version:

```
static bool terminalHasColors() {
  if (const char *term = std::getenv("TERM")) {
    // Most modern terminals support ANSI escape sequences for colors.
    // We could check terminfo, or have a list of known terms that support
    // colors, but that would be overkill.
    // The user can always ask for no colors by setting TERM to dumb, or
    // using a commandline flag.
    return strcmp(term, "dumb") != 0;
  }
  return false;
}
```

It became a "silly dance" involving calling `set_curterm()` with a null pointer.

19. Writing Memory

As an experiment and to explore a possible workaround, I'll modify memory of the running process to avoid the `set_curterm()` of zero.

I'll run gdb, set a breakpoint on `set_curterm()`, and take it to the zero invocation:

```
# gdb `which python`
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
[...]
(gdb) b set_curterm
Function "set_curterm" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (set_curterm) pending.
(gdb) r cachetop.py
Starting program: /usr/bin/python cachetop.py
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, set_curterm (termp=termp@entry=0xa43150) at /build/ncurses-pKZ1BN/ncurses-6.0+20160
80      {
(gdb) c
Continuing.

Breakpoint 1, set_curterm (termp=termp@entry=0xab5870) at /build/ncurses-pKZ1BN/ncurses-6.0+20160
80      {
(gdb) c
Continuing.

Breakpoint 1, set_curterm (termp=termp@entry=0xbecb90) at /build/ncurses-pKZ1BN/ncurses-6.0+20160
80      {
(gdb) c
Continuing.

Breakpoint 1, set_curterm (termp=0x0) at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tinfo
80      {
```

At this point I'll use the `set` command to overwrite memory and replace zero with the previous argument of `set_curterm()`, 0xbecb90, seen above, on the hope that it's still valid.

WARNING: Writing memory is not safe! gdb won't ask "are you sure?". If you get it wrong or make a typo, you will corrupt the application. Best case, your application crashes immediately, and you realize your mistake. Worst case, your application continues with silently corrupted data that is only discovered years later.

In this case, I'm experimenting on a lab machine with no production data, so I'll continue. I'll print the value of the %rdi register as hex (`p/x`), then `set` it to the previous address, print it again, then print all registers:

```
(gdb) p/x $rdi
$1 = 0x0
(gdb) set $rdi=0xbecb90
(gdb) p/x $rdi
$2 = 0xbecb90
(gdb) i r
rax            0x100    256
rbx            0x1  1
rcx            0xe71    3697
rdx            0x0  0
rsi            0x7ffff5dd45d3   140737318307283
rdi            0xbecb90 12503952
rbp            0x100    0x100
rsp            0x7fffffffa5b8   0x7fffffffa5b8
r8             0xbf0050 12517456
r9             0x1999999999999999   1844674407370955161
r10            0xbf0040 12517440
r11            0x7ffff7bb4b78   140737349634936
r12            0xbecb70 12503920
r13            0xbeaea0 12496544
r14            0x7fffffffa9a0   140737488333216
r15            0x7fffffffa8a0   140737488332960
rip            0x7ffff3c76a80   0x7ffff3c76a80 <set_curterm>
eflags         0x246    [ PF ZF IF ]
cs             0x33 51
ss             0x2b 43
ds             0x0  0
es             0x0  0
fs             0x0  0
gs             0x0  0
```

(Since at this point I have debug info installed, I don't need to refer to registers in this case, I could have called `set` on "`termp`", the variable name argument to `set_curterm()`, instead of `$rdi`.)

%rdi is now populated, so those registers look ok to continue.

```
(gdb) c
Continuing.

Breakpoint 1, set_curterm (termp=termp@entry=0x0) at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/n
80  {
```

Ok, we survived a call to `set_curterm()`! However, we've hit another, also with an argument of zero. Trying our write trick again:

```
(gdb) set $rdi=0xbecb90
(gdb) c
Continuing.
warning: JITed object file architecture unknown is not compatible with target architecture i386:x

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff34ad411 in ClrBlank (win=0xaea060) at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurs
1129        if (back_color_erase)
```

Ahhh. That's what I get for writing memory. So this experiment ended in another segfault.

20. Conditional Breakpoints

In the previous section, I had to use three continues to reach the right invocation of a breakpoint. If that were hundreds of invocations, then I'd use a conditional breakpoint. Here's an example.

I'll run the program and break on `set_curterm()` as usual:

```
# gdb `which python`
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
[...]
(gdb) b set_curterm
Function "set_curterm" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (set_curterm) pending.
(gdb) r cachetop.py
Starting program: /usr/bin/python cachetop.py
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, set_curterm (termp=termp@entry=0xa43150) at /build/ncurses-pKZ1BN/ncurses-6.0+20160
80  {
```

Now I'll turn breakpoint 1 into a conditional breakpoint, so that it only fires when the %rdi register is zero:

```
(gdb) cond 1 $rdi==0x0
(gdb) i b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00007ffff3c76a80 in set_curterm at /build/ncurses-pKZ1BN/ncurse
	stop only if $rdi==0x0
	breakpoint already hit 1 time
(gdb) c
Continuing.

Breakpoint 1, set_curterm (termp=0x0) at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tinfo
(gdb)
```

Neat! `cond` is short for `conditional`. So why didn't I run it right away, when I first created the "pending" breakpoint? I've found conditionals don't work on pending breakpoints, at least on this gdb version. (Either that or I'm doing it wrong.) I also used `i b` here (`info breakpoints`) to list them with information.

21. Returns

I did try another write-like hack, but this time changing the instruction path rather than the data.

WARNING: see previous warning, which also applies here.

I'll take us to the `set_curterm()` 0x0 breakpoint as before, and then issue a `ret` (short for `return`), which will return from the function immediately and not execute it. My hope is that by not executing it, it won't set the global `curterm` to 0x0.

```
[...]
(gdb) c
Continuing.

Breakpoint 1, set_curterm (termp=0x0) at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tinfo

(gdb) ret
Make set_curterm return now? (y or n) y
#0  0x00007ffff5a44e75 in llvm::sys::Process::FileDescriptorHasColors(int) () from /usr/lib/x86_6
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
                                                    _nc_free_termtype (ptr=ptr@entry=0x100) at /b
52      FreeIfNeeded(ptr->str_table);
```

Another crash. Again, that's what I get for messing in this way.

One more try. After browsing the code a bit more, I want to try doing a `ret` twice, in case the parent function is also involved. Again, this is just a hacky experiment:

```
[...]
(gdb) c
Continuing.

Breakpoint 1, set_curterm (termp=0x0) at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tinfo
80  {
(gdb) ret
Make set_curterm return now? (y or n) y
#0  0x00007ffff5a44e75 in llvm::sys::Process::FileDescriptorHasColors(int) () from /usr/lib/x86_6
(gdb) ret
Make selected stack frame return now? (y or n) y
#0  0x00007ffff45cabb8 in clang::driver::tools::Clang::ConstructJob(clang::driver::Compilation&,
(gdb) c
```

The screen goes blank and pauses...then redraws:

```
07:44:22 Buffers MB: 61 / Cached MB: 1246
PID      UID      CMD              HITS     MISSES   DIRTIES  READ_HIT%  WRITE_HIT%
    2742 root     systemd-logind       3        66        2       1.4%      95.7%
   15836 root     kworker/u30:1        7         0        1      85.7%       0.0%
    2736 messageb dbus-daemon          8        66        2       8.1%      89.2%
       1 root     systemd             15         0        0     100.0%       0.0%
    2812 syslog   rs:main Q:Reg       16        66        8       9.8%      80.5%
     435 root     systemd-journal     32        66        8      24.5%      67.3%
    2740 root     accounts-daemon    113        66        2      62.0%      36.9%
   15847 root     bash               160         0        1      99.4%       0.0%
   15864 root     lesspipe           306         0        2      99.3%       0.0%
   15854 root     bash               309         0        2      99.4%       0.0%
   15856 root     bash               309         0        2      99.4%       0.0%
   15866 root     bash               309         0        2      99.4%       0.0%
   15867 root     bash               309         0        2      99.4%       0.0%
   15860 root     bash               313         0        2      99.4%       0.0%
   15868 root     bash               341         0        2      99.4%       0.0%
   15858 root     uname              452         0        2      99.6%       0.0%
   15858 root     bash               453         0        2      99.6%       0.0%
   15866 root     dircolors          464         0        2      99.6%       0.0%
   15861 root     basename           465         0        2      99.6%       0.0%
   15864 root     dirname            468         0        2      99.6%       0.0%
   15856 root     ls                 476         0        2      99.6%       0.0%
[...]
```

Wow! It's working!

22. A Better Workaround

I'd been posting debugging output to github, especially since the lead BPF engineer, Alexei Starovoitov, is also well versed in llvm internals, and the root cause seemed to be a bug in llvm. While I was messing with writes and returns, he suggested adding the llvm option `-fno-color-diagnostics` to bcc, to avoid this problem code path. It worked! It was added to bcc as a workaround. (And we should get that llvm bug fixed.)

23. Python Context

At this point we've fixed the problem, but you might be curious to see the stack trace fully fixed.

Adding python-dbg:

```
# apt-get install -y python-dbg
Reading package lists... Done
[...]
The following additional packages will be installed:
  libpython-dbg libpython2.7-dbg python2.7-dbg
Suggested packages:
  python2.7-gdbm-dbg python2.7-tk-dbg python-gdbm-dbg python-tk-dbg
The following NEW packages will be installed:
  libpython-dbg libpython2.7-dbg python-dbg python2.7-dbg
0 upgraded, 4 newly installed, 0 to remove and 20 not upgraded.
Need to get 11.9 MB of archives.
After this operation, 36.4 MB of additional disk space will be used.
[...]
```

Now I'll rerun gdb and view the stack trace:

```
# gdb `which python` /var/cores/core.python.30520
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
[...]
Reading symbols from /usr/bin/python...Reading symbols from /usr/lib/debug/.build-id/4e/a0539215b
[...]
(gdb) bt
#0  ClrBlank (win=0x1993060) at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tty/tty_update
#1  ClrUpdate () at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tty/tty_update.c:1147
#2  doupdate () at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/tty/tty_update.c:1010
#3  0x00007f0a37aa07e6 in wrefresh (win=win@entry=0x1993060) at /build/ncurses-pKZ1BN/ncurses-6.0
#4  0x00007f0a37a99499 in recur_wrefresh (win=win@entry=0x1993060) at /build/ncurses-pKZ1BN/ncurs
#5  0x00007f0a37a99616 in _nc_wgetch (win=win@entry=0x1993060, result=result@entry=0x7ffd33d93e24
    at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/ncurses/base/lib_getch.c:491
#6  0x00007f0a37a9a325 in wgetch (win=0x1993060) at /build/ncurses-pKZ1BN/ncurses-6.0+20160213/nc
#7  0x00007f0a37cc6ec3 in PyCursesWindow_GetCh.lto_priv.109 (self=0x7f0a3c57b198, args=()) at /bu
#8  0x00000000004c4d5a in call_function (oparg=<optimized out>, pp_stack=0x7ffd33d93f30) at ../Py
#9  PyEval_EvalFrameEx () at ../Python/ceval.c:2987
#10 0x00000000004c2e05 in PyEval_EvalCodeEx () at ../Python/ceval.c:3582
#11 0x00000000004def08 in function_call.lto_priv () at ../Objects/funcobject.c:523
#12 0x00000000004b1153 in PyObject_Call () at ../Objects/abstract.c:2546
#13 0x00000000004c73ec in ext_do_call (nk=0, na=<optimized out>, flags=<optimized out>, pp_stack=
    at ../Python/ceval.c:4662
#14 PyEval_EvalFrameEx () at ../Python/ceval.c:3026
#15 0x00000000004c2e05 in PyEval_EvalCodeEx () at ../Python/ceval.c:3582
#16 0x00000000004caf42 in fast_function (nk=0, na=<optimized out>, n=<optimized out>, pp_stack=0x
    at ../Python/ceval.c:4445
#17 call_function (oparg=<optimized out>, pp_stack=0x7ffd33d943f0) at ../Python/ceval.c:4370
#18 PyEval_EvalFrameEx () at ../Python/ceval.c:2987
#19 0x00000000004c2e05 in PyEval_EvalCodeEx () at ../Python/ceval.c:3582
#20 0x00000000004c2ba9 in PyEval_EvalCode (co=<optimized out>, globals=<optimized out>, locals=<o
#21 0x00000000004f20ef in run_mod.lto_priv () at ../Python/pythonrun.c:1376
#22 0x00000000004eca72 in PyRun_FileExFlags () at ../Python/pythonrun.c:1362
#23 0x00000000004eb1f1 in PyRun_SimpleFileExFlags () at ../Python/pythonrun.c:948
#24 0x000000000049e18a in Py_Main () at ../Modules/main.c:640
#25 0x00007f0a3be10830 in __libc_start_main (main=0x49daf0 <main>, argc=2, argv=0x7ffd33d94838, i
    stack_end=0x7ffd33d94828) at ../csu/libc-start.c:291
#26 0x000000000049da19 in _start ()
```

No more "??"'s, but not hugely more helpful, yet.

The python debug packages have added other capabilities to gdb. Now we can look at the python backtrace:

```
(gdb) py-bt
Traceback (most recent call first):
  File "./cachetop.py", line 188, in handle_loop
    s = stdscr.getch()
  File "/usr/lib/python2.7/curses/wrapper.py", line 43, in wrapper
    return func(stdscr, *args, **kwds)
  File "./cachetop.py", line 260, in
    curses.wrapper(handle_loop, args)
```

... and Python source list:

```
(gdb) py-list
 183        b.attach_kprobe(event="mark_buffer_dirty", fn_name="do_count")
 184
 185        exiting = 0
 186
 187        while 1:
>188            s = stdscr.getch()
 189            if s == ord('q'):
 190                exiting = 1
 191            elif s == ord('r'):
 192                sort_reverse = not sort_reverse
 193            elif s == ord('<'):
```

It's identifying where in our Python code we were executing that hit the segfault. That's really nice!

The problem with the initial stack trace is that we're seeing Python internals that are executing the methods, but not the methods themselves. If you're debugging another language, it's up to its complier/runtime how it ends up executing code. If you do a web search for "*language name*" and "gdb" you might find it has gdb debugging extensions like Python does. If it doesn't, the bad news is you'll need to write your own. The good news is that this is even possible! Search for documentation on "adding new GDB commands in Python", as they can be written in Python.

24. And More

While it might look like I've written comprehensive tour of gdb, I really haven't: there's a lot more to gdb. The `help` command will list the major sections:

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

You can then run `help` on each command class. For example, here's the full listing for breakpoints:

```
(gdb) help breakpoints
Making program stop at certain points.

List of commands:

awatch -- Set a watchpoint for an expression
break -- Set breakpoint at specified location
break-range -- Set a breakpoint for an address range
catch -- Set catchpoints to catch events
catch assert -- Catch failed Ada assertions
catch catch -- Catch an exception
catch exception -- Catch Ada exceptions
catch exec -- Catch calls to exec
catch fork -- Catch calls to fork
catch load -- Catch loads of shared libraries
catch rethrow -- Catch an exception
catch signal -- Catch signals by their names and/or numbers
catch syscall -- Catch system calls by their names and/or numbers
catch throw -- Catch an exception
catch unload -- Catch unloads of shared libraries
catch vfork -- Catch calls to vfork
clear -- Clear breakpoint at specified location
commands -- Set commands to be executed when a breakpoint is hit
condition -- Specify breakpoint number N to break only if COND is true
delete -- Delete some breakpoints or auto-display expressions
delete bookmark -- Delete a bookmark from the bookmark list
delete breakpoints -- Delete some breakpoints or auto-display expressions
delete checkpoint -- Delete a checkpoint (experimental)
delete display -- Cancel some expressions to be displayed when program stops
delete mem -- Delete memory region
delete tracepoints -- Delete specified tracepoints
delete tvariable -- Delete one or more trace state variables
disable -- Disable some breakpoints
disable breakpoints -- Disable some breakpoints
disable display -- Disable some expressions to be displayed when program stops
disable frame-filter -- GDB command to disable the specified frame-filter
disable mem -- Disable memory region
disable pretty-printer -- GDB command to disable the specified pretty-printer
disable probes -- Disable probes
disable tracepoints -- Disable specified tracepoints
disable type-printer -- GDB command to disable the specified type-printer
disable unwinder -- GDB command to disable the specified unwinder
disable xmethod -- GDB command to disable a specified (group of) xmethod(s)
dprintf -- Set a dynamic printf at specified location
enable -- Enable some breakpoints
enable breakpoints -- Enable some breakpoints
enable breakpoints count -- Enable breakpoints for COUNT hits
enable breakpoints delete -- Enable breakpoints and delete when hit
enable breakpoints once -- Enable breakpoints for one hit
enable count -- Enable breakpoints for COUNT hits
enable delete -- Enable breakpoints and delete when hit
enable display -- Enable some expressions to be displayed when program stops
enable frame-filter -- GDB command to disable the specified frame-filter
enable mem -- Enable memory region
enable once -- Enable breakpoints for one hit
enable pretty-printer -- GDB command to enable the specified pretty-printer
enable probes -- Enable probes
enable tracepoints -- Enable specified tracepoints
enable type-printer -- GDB command to enable the specified type printer
enable unwinder -- GDB command to enable unwinders
enable xmethod -- GDB command to enable a specified (group of) xmethod(s)
ftrace -- Set a fast tracepoint at specified location
```

```
hbreak -- Set a hardware assisted breakpoint
ignore -- Set ignore-count of breakpoint number N to COUNT
rbreak -- Set a breakpoint for all functions matching REGEXP
rwatch -- Set a read watchpoint for an expression
save -- Save breakpoint definitions as a script
save breakpoints -- Save current breakpoint definitions as a script
save gdb-index -- Save a gdb-index file
save tracepoints -- Save current tracepoint definitions as a script
skip -- Ignore a function while stepping
skip delete -- Delete skip entries
skip disable -- Disable skip entries
skip enable -- Enable skip entries
skip file -- Ignore a file while stepping
skip function -- Ignore a function while stepping
strace -- Set a static tracepoint at location or marker
tbreak -- Set a temporary breakpoint
tcatch -- Set temporary catchpoints to catch events
tcatch assert -- Catch failed Ada assertions
tcatch catch -- Catch an exception
tcatch exception -- Catch Ada exceptions
tcatch exec -- Catch calls to exec
tcatch fork -- Catch calls to fork
tcatch load -- Catch loads of shared libraries
tcatch rethrow -- Catch an exception
tcatch signal -- Catch signals by their names and/or numbers
tcatch syscall -- Catch system calls by their names and/or numbers
tcatch throw -- Catch an exception
tcatch unload -- Catch unloads of shared libraries
tcatch vfork -- Catch calls to vfork
thbreak -- Set a temporary hardware assisted breakpoint
trace -- Set a tracepoint at specified location
watch -- Set a watchpoint for an expression

Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

This helps to illustrate how many capabilities gdb has, and how few I needed to use in this example.

25. Final Words

Well, that was kind of a nasty issue: an LLVM bug breaking ncurses and causing a Python program to segfault. But the commands and procedures I used to debug it were mostly routine: viewing stack traces, checking registers, setting breakpoints, stepping, and browsing source.

When I first used gdb (years ago), I really didn't like it. It felt clumsy and limited. gdb has improved a lot since then, as have my gdb skills, and I now see it as a powerful modern debugger. Feature sets vary between debuggers, but gdb may be the most powerful text-based debugger nowadays, with lldb catching up.

I hope anyone searching for gdb examples finds the full output I've shared to be useful, as well as the various caveats I discussed along the way. Maybe I'll post some more gdb sessions when I get a chance, especially for other runtimes like Java.

It's `q` to quit gdb.

You can comment here, but I can't guarantee your comment will remain here forever: I might switch comment systems at some point (eg, if disqus add advertisements).