

61 | 搭建操作系统实验环境（下）：授人以鱼不如授人以渔

2019-08-16 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 10:24 大小 9.54M



上一节我们做了一个实验，添加了一个系统调用，并且编译了内核。这一节，我们来尝试调试内核。这样，我们就可以一步一步来看，内核的代码逻辑执行到哪一步了，对应的变量值是什么。

了解 gdb

在 Linux 下面，调试程序使用一个叫作 gdb 的工具。通过这个工具，我们可以逐行运行程序。


例如，上一节我们写的 syscall.c 这个程序，我们就可以通过下面的命令编译。

复制代码

```
1 gcc -g syscall.c
```


其中，参数 -g 的意思就是在编译好的二进制程序中，加入 debug 所需的信息。

接下来，我们安装一下 gdb。

 复制代码

```
1 apt-get install gdb
```

然后，我们就可以来调试这个程序了。

 复制代码

```
1 ~/syscall# gdb ./a.out
2 GNU gdb (Ubuntu 8.1-0ubuntu3.1) 8.1.0.20180409-git
3 .....
4 Reading symbols from ./a.out...done.
5 (gdb) l
6 1      #include <stdio.h>
7 2      #include <stdlib.h>
8 3      #include <unistd.h>
9 4      #include <linux/kernel.h>
10 5      #include <sys/syscall.h>
11 6      #include <string.h>
12 7
13 8      int main ()
14 9      {
15 10          char * words = "I am liuchao from user mode.";
16 (gdb) b 10
17 Breakpoint 1 at 0x6e2: file syscall.c, line 10.
18 (gdb) r
19 Starting program: /root/syscall/a.out
20
21 Breakpoint 1, main () at syscall.c:10
22 10          char * words = "I am liuchao from user mode.";
23 (gdb) n
24 12          ret = syscall(333, words, strlen(words)+1);
25 (gdb) p words
26 $1 = 0x5555555547c4 "I am liuchao from user mode."
27 (gdb) s
28 __strlen_sse2 () at ../sysdeps/x86_64/multiarch/./strlen.S:79
29 (gdb) bt
30 #0  __strlen_sse2 () at ../sysdeps/x86_64/multiarch/./strlen.S:79
31 #1  0x00005555555546f9 in main () at syscall.c:12
```

```
32 (gdb) c
33 Continuing.
34 return 63 from kernel mode.
35 [Inferior 1 (process 1774) exited normally]
36 (gdb) q
```

在上面的例子中，我们只要掌握简单的几个 gdb 的命令就可以了。

l，即 list，用于显示多行源代码。

b，即 break，用于设置断点。

r，即 run，用于开始运行程序。

n，即 next，用于执行下一条语句。如果该语句为函数调用，则不会进入函数内部执行。

p，即 print，用于打印内部变量值。

s，即 step，用于执行下一条语句。如果该语句为函数调用，则进入函数，执行其中的第一条语句。

c，即 continue，用于继续程序的运行，直到遇到下一个断点。

bt，即 backtrace，用于查看函数调用信息。

q，即 quit，用于退出 gdb 环境。

Debug kernel


看了 debug 一个进程还是简单的，接下来，我们来试着 debug 整个 kernel。

第一步，要想 kernel 能够被 debug，需要向上面编译程序一样，将 debug 所需信息也放入二进制文件里面去。这个我们在编译内核的时候已经设置过了，也就是把 “CONFIG_DEBUG_INFO” 和 “CONFIG_FRAME_POINTER” 两个变量设置为 yes。

第二步，就是安装 gdb。kernel 运行在 qemu 虚拟机里面，gdb 运行在宿主机上，所以我们应该在宿主机上进行安装。

第三步，找到 gdb 要运行的那个内核的二进制文件。这个文件在哪里呢？根据 grub 里面的配置，它应该在 /boot/vmlinuz-4.15.18 这里。

另外，为了方便在 debug 的过程中查看源代码，我们可以将 /usr/src/linux-source-4.15.0 整个目录，都拷贝到宿主机上来。因为内核一旦进入 debug 模式，就不能运行了。

 复制代码

```
1 scp -r popsuper@192.168.57.100:/usr/src/linux-source-4.15.0 ./
```

在 /usr/src/linux-source-4.15.0 这个目录下面，vmlinux 文件也是内核的二进制文件。


第四步，修改 qemu 的启动参数和 qemu 里面虚拟机的启动参数，从而使得 gdb 可以远程 attach 到 qemu 里面的内核上。

我们知道，gdb debug 一个进程的时候，gdb 会监控进程的运行，使得进程一行一行地执行二进制文件。如果像 syscall.c 的二进制文件 a.out 一样，就在本地，gdb 可以通过 attach 到这个进程上，作为这个进程的父进程，来监控它的运行。

但是，gdb debug 一个内核的时候，因为内核在 qemu 虚拟机里面，所以我们无法监控本地进程，而要通过 qemu 来监控 qemu 里面的内核，这就要借助 qemu 的机制。

qemu 有个参数 -s，它代表参数 -gdb tcp::1234，意思是 qemu 监听 1234 端口，gdb 可以 attach 到这个端口上来，debug qemu 里面的内核。

为了完成这一点，我们需要修改 ubuntutest 这个虚拟机的定义文件。

 复制代码

```
1 virsh edit ubuntutest
```

在这里，我们能将虚拟机的定义文件修改成下面的样子，其中主要改了两项：

在 domain 的最后加上了 qemu:commandline，里面指定了参数 -s；

在 domain 中添加 xmlns:qemu。没有这个 XML 的 namespace，qemu:commandline 这个参数 libvirt 不认。


```

1 <domain type='qemu' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
2   <name>ubuntutest</name>
3   <uuid>0f0806ab-531d-6134-5def-c5b4955292aa</uuid>
4   <memory unit='KiB'>8388608</memory>
5   <currentMemory unit='KiB'>8388608</currentMemory>
6   <vcpu placement='static'>8</vcpu>
7   <os>
8     <type arch='x86_64' machine='pc-i440fx-trusty'>hvm</type>
9     <boot dev='hd' />
10  </os>
11  <clock offset='utc' />
12  <on_poweroff>destroy</on_poweroff>
13  <on_reboot>restart</on_reboot>
14  <on_crash>restart</on_crash>
15  <devices>
16    <emulator>/usr/bin/qemu-system-x86_64</emulator>
17    <disk type='file' device='disk'>
18      <driver name='qemu' type='qcow2' />
19      <source file='/mnt/vdc/ubuntutest.img' />
20      <backingStore />
21      <target dev='vda' bus='virtio' />
22      <alias name='virtio-disk0' />
23      <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0' />
24    </disk>
25    .....
26    <interface type='bridge'>
27      <mac address='fa:16:3e:6e:89:ce' />
28      <source bridge='br0' />
29      <target dev='tap1' />
30      <model type='virtio' />
31      <alias name='net0' />
32      <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0' />
33    </interface>
34    .....
35  </devices>
36  <qemu:commandline>
37    <qemu:arg value='-s' />
38  </qemu:commandline>
39 </domain>

```


另外，为了远程 debug 成功，我们还需要修改 qemu 里面的虚拟机的 grub 和 menu.list，在内核命令行中添加 nokaslr，来关闭 KASLR。KASLR 会使得内核地址空间布局随机化，从而会造成我们打的断点不起作用。

对于 grub.conf，修改如下：

 复制代码

```
1 submenu 'Advanced options for Ubuntu' $menuentry_id_option 'gnulinux-advanced-470f3a42-7a97-4b9d-aaa0-26deb3d'
2     menuentry 'Ubuntu, with Linux 4.15.18' --class ubuntu --class gnu-linux --class
3         recordfail
4         load_video
5         gfxmode $linux_gfx_mode
6         insmod gzio
7         if [ x$grub_platform = xxen ]; then insmod xzio; insmod lzopio; fi
8         insmod part_gpt
9         insmod ext2
10        if [ x$feature_platform_search_hint = xy ]; then
11            search --no-floppy --fs-uuid --set=root 470f3a42-7a97-4b9d-aaa0-26deb3d
12        else
13            search --no-floppy --fs-uuid --set=root 470f3a42-7a97-4b9d-aaa0-26deb3d
14        fi
15        echo    'Loading Linux 4.15.18 ...'
16        linux   /boot/vmlinuz-4.15.18 root=UUID=470f3a42-7a97-4b9d-aaa0-26deb3d
17        echo    'Loading initial ramdisk ...'
18        initrd  /boot/initrd.img-4.15.18
19    }
```

对于 menu.list , 修改如下 :


 复制代码

```
1 title                Ubuntu 18.04.2 LTS, kernel 4.15.18
2 root                 (hd0)
3 kernel               /boot/vmlinuz-4.15.18 root=/dev/hda1 ro nokaslr console=hvc0 console=tty
4 initrd               /boot/initrd.img-4.15.18
```

修改完毕后, 我们需要在虚拟机里面 `shutdown -h now` , 来关闭虚拟机。注意不要 `reboot` , 因为虚拟机里面运行 `reboot` , 我们改过的那个 XML 会不起作用。

当我们在宿主机上发现虚拟机关机之后, 就可以通过 `virsh start ubuntu` 启动虚拟机, 这个时候我们添加的参数 `-s` 才起作用。

第五步, 使用 `gdb` 运行内核的二进制文件, 执行 `gdb vmlinux`。

 复制代码

```
1 /mnt/vdc/linux-source-4.15.0# gdb vmlinux
```

```

2 GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
3 .....
4 To enable execution of this file add
5     add-auto-load-safe-path /mnt/vdc/linux-source-4.15.0/vmlinux-gdb.py
6 .....
7 (gdb) b sys_sayhelloworld
8 Breakpoint 1 at 0xffffffff8109e2f0: file kernel/sys.c, line 192.
9 (gdb) target remote :1234
10 Remote debugging using :1234
11 native_safe_halt () at ./arch/x86/include/asm/irqflags.h:61
12 61     }
13 (gdb) c
14 Continuing.
15 [Switching to Thread 2]
16 Thread 2 hit Breakpoint 1, sys_sayhelloworld (words=0x563cbfa907c4 "I am liuchao from u:
17 192     {
18 (gdb) bt
19 #0  sys_sayhelloworld (words=0x55b2811537c4 "I am liuchao from user mode.", count=29) at
20 #1  0xffffffff810039f7 in do_syscall_64 (regs=0xffffc9000133bf58) at arch/x86/entry/comr
21 #2  0xffffffff81a00081 in entry_SYSCALL_64 () at arch/x86/entry/entry_64.S:237
22 (gdb) n
23 195         if(count >= 1024){
24 (gdb) n
25 198         copy_from_user(buffer, words, count);
26 (gdb) n
27 199         ret=printk("User Mode says %s to the Kernel Mode!", buffer);
28 (gdb) p buffer
29 $1 = "I am liuchao from user mode.\000\177\000\000\...
30 (gdb) n
31 200         return ret;
32 (gdb) p ret
33 $2 = 63
34 (gdb) c
35 (gdb) n
36 do_syscall_64 (regs=0xffffc9000133bf58) at arch/x86/entry/common.c:295
37 295         syscall_return_slowpath(regs);
38 (gdb) s
39 syscall_return_slowpath (regs=<optimized out>) at arch/x86/entry/common.c:295
40 (gdb) n
41 268         prepare_exit_to_usermode(regs);
42 (gdb) n
43 do_syscall_64 (regs=0xffffc9000133bf58) at arch/x86/entry/common.c:296
44 296     }
45 (gdb) n
46 entry_SYSCALL_64 () at arch/x86/entry/entry_64.S:246
47 246         movq    RCX(%rsp), %rcx
48 .....
49 (gdb) n
50 entry_SYSCALL_64 () at arch/x86/entry/entry_64.S:330
51 330         USERGS_SYSRET64

```


我们先设置一个断点在我们自己写的系统调用上 `b sys_sayhelloworld`，通过执行 `target remote :1234`，来 attach 到 qemu 上，然后，执行 `c`，也即 `continue` 运行内核。这个时候内核始终在 `Continuing` 的状态，也即持续在运行中，这个时候我们可以远程登录到 qemu 里的虚拟机上，执行各种命令。

如果我们在虚拟机里面运行 `syscall.c` 编译好的 `a.out`，这个时候肯定会调用到内核。内核肯定会经过系统调用的过程，到达 `sys_sayhelloworld` 这个函数，这就碰到了我们设置的那个断点。

如果执行 `bt`，我们能看到，这个系统调用是从 `entry_64.S` 里面的 `entry_SYSCALL_64()` 函数，调用到 `do_syscall_64` 函数，再调用到 `sys_sayhelloworld` 函数的。这一点和我们在[系统调用](#)那一节分析的过程是一模一样的。

我们可以通过执行 `next` 命令，来看 `sys_sayhelloworld` 一步一步是怎么执行的，通过 `p buffer` 查看 `buffer` 里面的内容。在这个过程中，由于内核是逐行运行的，因而我们在虚拟机里面的命令行是卡死的状态。

当我们不断地 `next`，直到执行完毕 `sys_sayhelloworld` 的时候，会看到，`do_syscall_64` 会调用 `syscall_return_slowpath`。它会调用 `prepare_exit_to_usermode`，然后会回到 `entry_SYSCALL_64`，然后对于寄存器进行操作，最后调用指令 `USERGS_SYSRET64` 回到用户态。这个返回的过程和系统调用那一节也一模一样。

看，通过 `debug` 我们能够跟踪系统调用的整个过程。你可以将我们这一门课里面学得的所有过程都 `debug` 一下，看看变量的值，从而对于内核的工作机制有更加深入的了解。

总结时刻

在这个课程里面，我们写过一些程序，为了保证程序能够顺利运行，我一般会将代码完整地放到文本中，让你拷贝下来就能编译和运行。如果你运行的时候发现问题，或者想了解一步一步运行的细节，这一节介绍的 `gdb` 是一个很好的工具。

这一节你尤其应该掌握的是，如何通过宿主机上的 `gdb` 来 `debug` 虚拟机里面的内核。这一点非常重要，会了这个，你就能够返回去，挨个研究每一章每一节的内核数据结构和运行逻辑了。

在这门课中，进程管理、内存管理、文件管理、设备管理、网络管理，我们都介绍了从系统调用到底层的整个逻辑。如果你对我前面的代码解析还比较困惑，你可以尝试着去 debug 这些过程，只要把断点打在系统调用的入口位置就可以了。

从此，开启你的内核 debug 之旅吧！

课堂练习

这里给你留一道题目，你可以试着 debug 一下文件打开的过程。

欢迎留言和我分享你的疑惑和见解，也欢迎你收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习、进步。

 极客时间

趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院
云计算技术部首席架构师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 60 | 搭建操作系统实验环境（上）：授人以鱼不如授人以渔

下一篇 62 | 知识串讲 | 用一个创业故事串起操作系统原理（一）

精选留言 (3)

 写留言



LDxy

2019-08-17

打断点的指令b后面是不是既可以跟行号也可以跟函数名？



kxxue

2019-08-16

赞👍

展开▼



许童童

2019-08-16

跟着老师一起精进。

展开▼

