随想录（在实践中学习 kernel 代码）

记得我在读书的时候，虽然老师也教过操作系统的课程，但是自己的理解却不是很充分，实践部分的内容就更少。对于课程中的内容，比如说中断、互斥、线程、IO 等概念常常也是一知半解，没有什么特别深刻的体会。等到毕业后，自己开始花钱买一些国外作者写的书，认识上有了很大的改变，但是自己真正动手写代码、调试的部分还是太少。等到去年的时候，自己开始在 pc 上面仿真 ucos 系统的时候，对每一行代码进行单步调试的时候，才真正从本质上理解了系统本身。

当然，我对很多计算机课程的认识，原来都是基本停留在书本上面。不管是计算机网络、编译原理还是人机界面，书本上的知识点虽然大概也知道些，但是你要说理解得有多透彻却说不上来。等到自己真正阅读了 lwip、lua 和 ftk 的相关代码，你才会大呼一声，原来是这么回事，好像也不是很复杂。所以，对于计算机的知识，要想把自己的认识提高一个层次的话，最好的方法就是实践。

在国内，喜欢研究 linux 的人很多，但是大多数朋友对 linux kenel 的理解只是停留在书本上面、视频上面以及 ppt 上面。真正自己动手做实验、把每一行代码都弄懂弄明白的朋友却不是那么多。其实，和以前相比，现在的 linux 版本更多、也更稳定，资源也特别丰富。就我个人认为，linux 系统是学习计算机最好的系统。因为在这么一个系统上面，没有人帮你，很多的困惑都需要自己去解决，只有去克服一个一个难点，你才能感受到自己确实实在进步。这种进步不是停留在学会了某种配置、安装了某种软件、熟悉了某种环境，而在于你对系统本身的认识更高了，看问题的角度发生改变了。

在书店里面，介绍 linux 基本软件开发的书特别多，但是却有很少的书介绍如何在 linux 进行 kernel 的学习、编译、调试，哪些调试工具比较合适等等。他们做的就是直接把答案告诉你，至于为什么这样设计，交代的内容却很少。一方面这方面的需求比较少，另外一方面这些知识点有点难度，不易被大家接受。实际上，这里大家存在一个误区，现在的 linux kernel 开发其实不是那么难，你所需要的其实就是一个 pc、一根网线，这样你就获得了学习的全部资源。下面内容主要是介绍给大家如何学习 linux kernel，特别是如何在实践中学习。

（1）选择一个合适 linux 发行版本，我自身选用的版本是 CentOS 6，使用非常方便

（2）从 www.kernel.org 下载 kernel 代码，学习如何编译 linux 内核，其实步骤也不复杂

a）解压 linux 内核版本
b）cd linux 目录

c）cp /boot/config-2.6.32-220.el6.i686 .config

d） make menuconfig

e）保存，直接 exit 退出

f）make bzImage

g） make modules

h） make modules_install

i） make install

（3） 重启电脑，开机选用新的 linux 内核。开始编写 hello.c 文件，生成模块，利用模块与 linux 内核函数进行互动，用 dmesg -c 查看打印

[cpp] view plaincopyprint?

```
1.  #include <linux/module.h>
2.  #include <linux/init.h>
3.
4.  MODULE_LICENSE("GPL");
5.  MODULE_AUTHOR("feixiaoxing");
6.  MODULE_DESCRIPTION("This is just a hello module!\n");
7.
8.  static int __init hello_init(void)
9.  {
10. printk(KERN_EMERG "hello, init\n");
11. dump_stack();
12. return 0;
13. }
14.
15. static void __exit hello_exit(void)
16. {
17. printk(KERN_EMERG "hello, exit\n");
18. }
19.
20. module_init(hello_init);
21. module_exit(hello_exit);
```

（4）安装 elfutils-devel，systemtap 安装包，用 stap -v *.stp 开始分析内核。你要做的就是插入调试点，编写脚本文件，自由分析内核的代码内容，

a）基础打印

[cpp] view plaincopyprint?

```
1.  probe begin{
2.  printf("hello begin!\n")
3.  }
4.
5.  probe end{
6.  printf("hello end!\n")
7.  }
```

b）定时打印

[cpp] view plaincopyprint?

```
1.  global number
2.
3.  probe begin
4.  {
5.  number = 0
6.  }
7.
8.  probe timer.ms(5000)
9.  {
10. number ++
11. printf ("%d\n", number)
12. }
13.
14. probe end
15. {
16. number = 0
17. }
```

c）统计 syscall 调用

[cpp] view plaincopyprint?

```
1.  global syscalls
2.
3.  function print_top () {
4.
5.  cnt = 0
6.  log ("SYSCALL\t\t\tCOUNT")
7.  foreach ([name] in syscalls-) {
8.  printf("%-20s %5d\n",name, syscalls[name])
9.  if (cnt++ == 20)
10. break
11. }
12.
13. printf("-------------------------------------\n")
14. delete syscalls
15. }
16.
17. probe kernel.function("sys_*") {
18. syscalls[probefunc()]++
19.
20. }
21.
22. # print top syscalls every 5 seconds
23. probe timer.ms(5000) {
24. print_top ()
25. }
```

d）统计 schedule 函数被调用的次数

[cpp] view plaincopyprint?

```
1.  global count
2.
```

```
3.  probe kernel.function("schedule")
4.  {
5.  count ++
6.  }
7.
8.  probe begin{
9.  count = 0
10. }
11.
12. probe end{
13. printf("count = %d\n", count);
14. }
```

e）打印回调堆栈

[cpp] view plaincopyprint?

```
1.  global count
2.
3.  probe kernel.function("schedule").return
4.  {
5.  if(!count)
6.  print_backtrace()
7.
8.  count ++
9.  }
10.
11. probe begin{
12. count = 0
13. }
14.
15. probe end{
16. }
```

f）记录一次进程切换

[cpp] view plaincopyprint?

```
1.  global count
2.
3.  probe begin
4.  {
5.  count = 0
6.  }
7.
8.  probe kernel.function("__switch_to")
9.  {
10. if(!count)
11. {
12. printf("from [%s] to [%s]\n", task_execname($prev_p),
    task_execname($next_p))
13. }
14.
15. exit()
16. }
17.
18. probe end
19. {
20. }
```