Linux 内核 --- Project 2

1. Project 2A --- 模块编程

这个实验关于模块编程，模块实际上并没有编译到内核里，而是可以通过特定的接口动态地添加与删除，这里的三个实验有关模块的基本操作。

第一个实验即是简单地编译一个模块，关键在于接口，在这里，通过__init 与__exit 表明模块的入口以及出口函数，同时还要通过 module_init 与 module_exit 进行申明，为了表明模块的加载以及卸载，在出口和入口函数中使用 prink 来打印信息，这些信息可以通过 dmesg 命令来显示。

当然，另一个关键的文件是 Makefile，在这里需要表明需要编译为 module 的文件，之后 make -C 利用 linux 源代码顶层的 Makefile 进行编译，编译后会产生*.ko 文件。

```
obj-m := test1.o test2.o test3.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
all:
    make -C $(KDIR) M=$(PWD) modules
clean:
    make -C $(KDIR) M=$(PWD) clean
```

加载模块则是在命令行中使用 insmod，卸载则是使用 rmmod，最后 dmesg 产生的结果如下：

```
[zzs@localhost 1_modules]$ dmesg | tail -5
[   42.506570] nf_conntrack: automatic helper assignment is deprecated and it
. Use the iptables CT target to attach helpers instead.
[   73.023503] r8169 0000:04:00.0 p5p1: link down
[   74.638993] r8169 0000:04:00.0 p5p1: link up
[ 1561.767458] Greeting from a new module...
[ 1567.356344] Bye...
```

第二个实验是为模块添加参数，这些参数可以在 insmod 的时候输入，这里就要使用 module_param 以及相关的函数进行申明，同时定义静态的变量来储存参数，例如，在这里就定义了 who，number，list 三个参数，分别为字符指针，整型以及数组。

```
static char *who = "No-one";
static int number ;
static int  list[] = {1,2,3};

static int the_number = 0;

module_param(who,charp,0664);
module_param(number,int,0664);
module_param_array(list,int,&the_number,0664);
```

在 insmod 的时候输入相应参数即可初始化这些变量：

```
[zzs@localhost 1_modules]$ sudo insmod test2.ko who=zzs number=123 list=1,3,5
[zzs@localhost 1_modules]$ dmesg | tail -5
[ 1561.767458] Greeting from a new module...
[ 1567.356344] Bye...
[ 1934.996549] test2: `5110309684' invalid for parameter `number'
[ 1953.435182] Greeting from a new module,name is zzs,number is 123.
[ 1953.435186] List's length is 3,and they are 1,3,5,
```

第三个实验则是使用 proc 文件系统，proc 文件系统的接口可以从 proc_fs.h 中看到，这里的接口函数与之前的版本有所不同，主要是 proc_create 函数，它的原型如下：

```
static inline struct proc_dir_entry *proc_create(
    const char *name, umode_t mode, struct proc_dir_entry *parent,
    const struct file_operations *proc_fops)
```

其中，proc_fops 的类型是 file_operations，应该是表明文件的读写方式，实际上，linux 内核中还有 seq_file 这样的一层接口，利用它可以简化 proc 文件系统的编程。

```
19 static int hello_proc_show(struct seq_file *m, void *v) {
20     seq_printf(m, "Hello proc! The string is %s...\n",buffer);
21     return 0;
22 }
23
24 static int hello_proc_open(struct inode *inode, struct  file *file) {
25   return single_open(file, hello_proc_show, NULL);
26 }
27
28 static const struct file_operations hello_proc_fops = {
29   .owner = THIS_MODULE,
30   .open = hello_proc_open,
31   .read = seq_read,
32   .write = hello_write,
33   .llseek = seq_lseek,
34   .release = single_release,
35 };
```

在这里，文件的 open 和 read 都是使用的 seq_file 提供的函数，而实际上需要编写的函数即使 hello_open_show 这个函数，它利用 seq_printf 打印相关信息实验结果如下：

```
[zzs@localhost 1_modules]$ sudo insmod test3.ko
[sudo] password for zzs:
[zzs@localhost 1_modules]$ cat /proc/
Display all 277 possibilities? (y or n)
[zzs@localhost 1_modules]$ cat /proc/hello_proc
Hello proc! The string is ...
```

2. **Project 2B --- ctx 的加入**

第二个任务即使修改内核调度部分的代码，为 task 加入被调度次数的信息。虽然这个任务并不改动调度算法，但是修改时还是要理解一些调度的关键函数的位置。

首先，唯每个 task 添加被调度次数的信息，可以添加到 task_struct 的末尾，即在 linux/sched.h 的 task_struct 的声明的末尾加上 int ctx_count;一项。

当然，在进行进程切换的时候需要修改这一项，相关文件是 kernel/sched/core.c，最终的调度利用的是 schedule（ ）函数，这里修改的是其调用的 context_switch（ ）函数，只需要为将要调度的 task 执行 ctx_count++即可。

当然，task 的这一个 field 需要初始化，这些都在 do_fork 的过程之中，具体的可以加在 copy_process 函数之中，置零即可。如果要求在 exec 的时候也置零，同样可以在 exec 的相关函数中将其置零。

实际上，这三行的修改应该就完成了任务，然而为了表现实验的结果，还需要在 proc 文件系统中表明这个结果。

这里对于 proc 文件系统的修改不同于之前，这里需要为每个 task group 创建文件，这里修改的是 fs/proc/base.c 中的 tgid_base_stuff 数组，为它添加一项 INF("ctx", S_IRUGO, proc_pid_ctx)；这里的添加是仿照其他项来的，INF 应该是最简单的接口，表明文件只是简单地打印 information，而 proc_pid_ctx 其实也就是如此，它的任务就是使用 sprintf 打印当前 task 的 ctx_count 的信息：

```
64
65 static int proc_pid_ctx(struct task_struct* task,char *buffer)
66 {
67     return sprintf(buffer,"%d",task->ctx_count);
68 }
69
```

当然，这只是一个简单的实现，如果要显示更多信息或是可以修改 proc 文件系统中的文件，应该可以使用 ONE 以及 REG 提供的接口。

3. 总结

这次实验的内容虽然挺多的，一共有四个实验，但是实际上并不是特别复杂，主要是理解以及运用 module 以及 proc 文件系统的接口以及理解系统的调度的一些过程。