提取进程信息

热身

通过模块编程，利用task_struct中的进程关系，从当前进程一直找到init的路线：

1. 编写route_struct.c代码，并运行make进行编译
2. 内核模块添加 $sudo insmod route_struct.ko
3. 添加内核模块后dmesg读取内核信息

代码整体已经给出，需要填充细节。

Main函数代码

```
1    #include <linux/kernel.h> /* printk() */
2    #include <linux/module.h>
3    #include <linux/init.h>
4    #include <linux/sched.h> /* task_struct */
5
```

Main函数代码

```c
int route_struct_init(void)
{
    struct task_struct *task = current;
    int i;
    printk(KERN_NOTICE "entering module");
    printk(KERN_NOTICE "0: This task is called %s, his pid is %d\n", task->comm, task->pid);
    for (i = 1; task->pid != 0; ++i)
    {
        //让task指向他的父进程
        //打印这个进程的comm、pid信息
    }
    return 0;
}
```

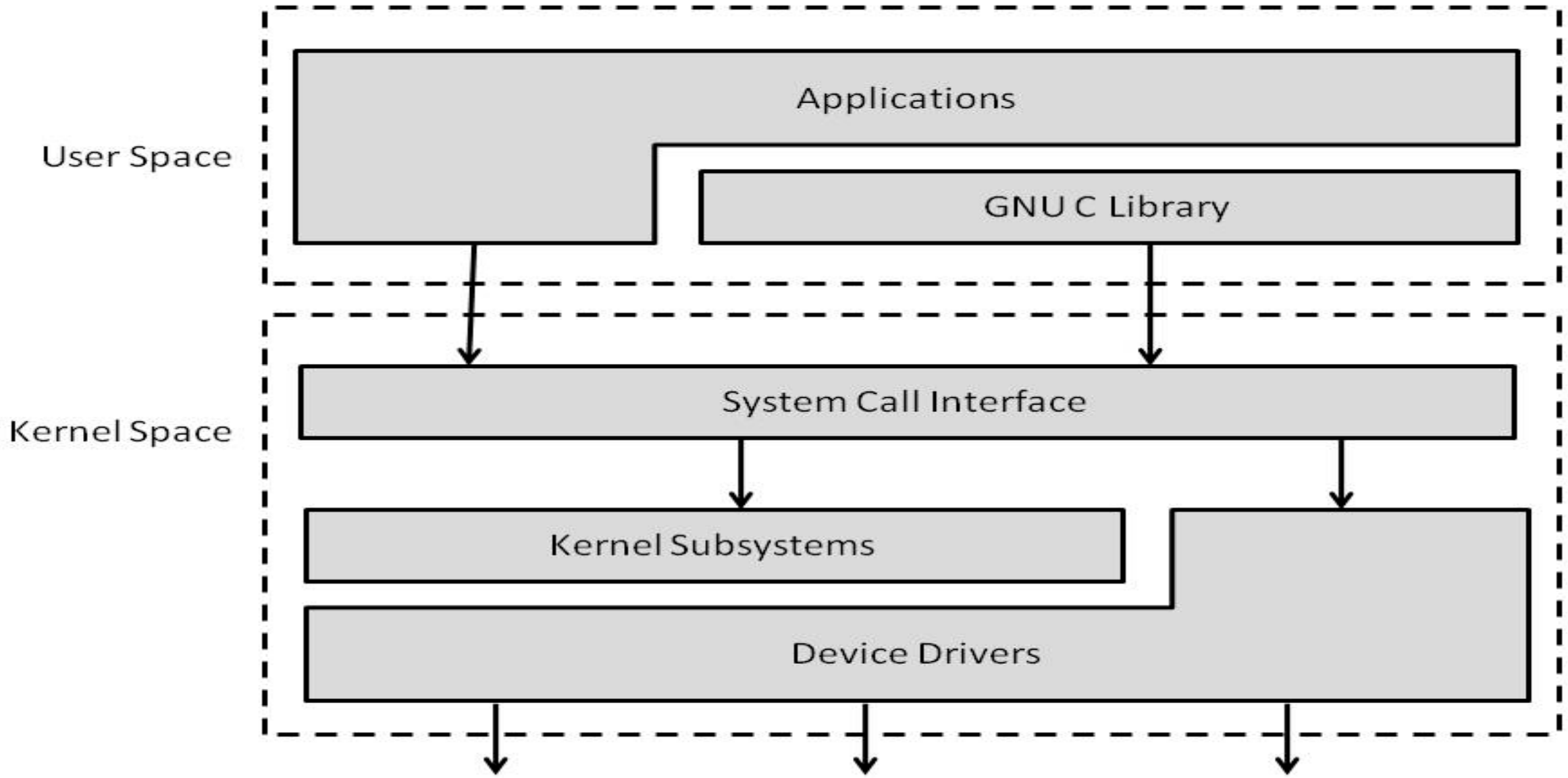
实验结果

```
[ 2165.183155] IPv6: ADDRCONF(NETDEV_CHANGE): ens33: link becomes ready
[ 2165.211295] usb 2-2.1: reset full-speed USB device number 4 using uhci_hcd
[ 2440.371138] entering module
[ 2440.371140] 0: This task is called insmod, his pid is 42898
[ 2440.371198] 1: This task is called bash, his pid is 42888
[ 2440.371201] 2: This task is called su, his pid is 42887
[ 2440.371202] 3: This task is called bash, his pid is 42879
[ 2440.371203] 4: This task is called gnome-terminal-, his pid is 42870
[ 2440.371204] 5: This task is called systemd, his pid is 1488
[ 2440.371205] 6: This task is called systemd, his pid is 1
[ 2440.371205] 7: This task is called swapper/0, his pid is 0
root@ubuntu:/home/amos/Desktop/task struct#
```

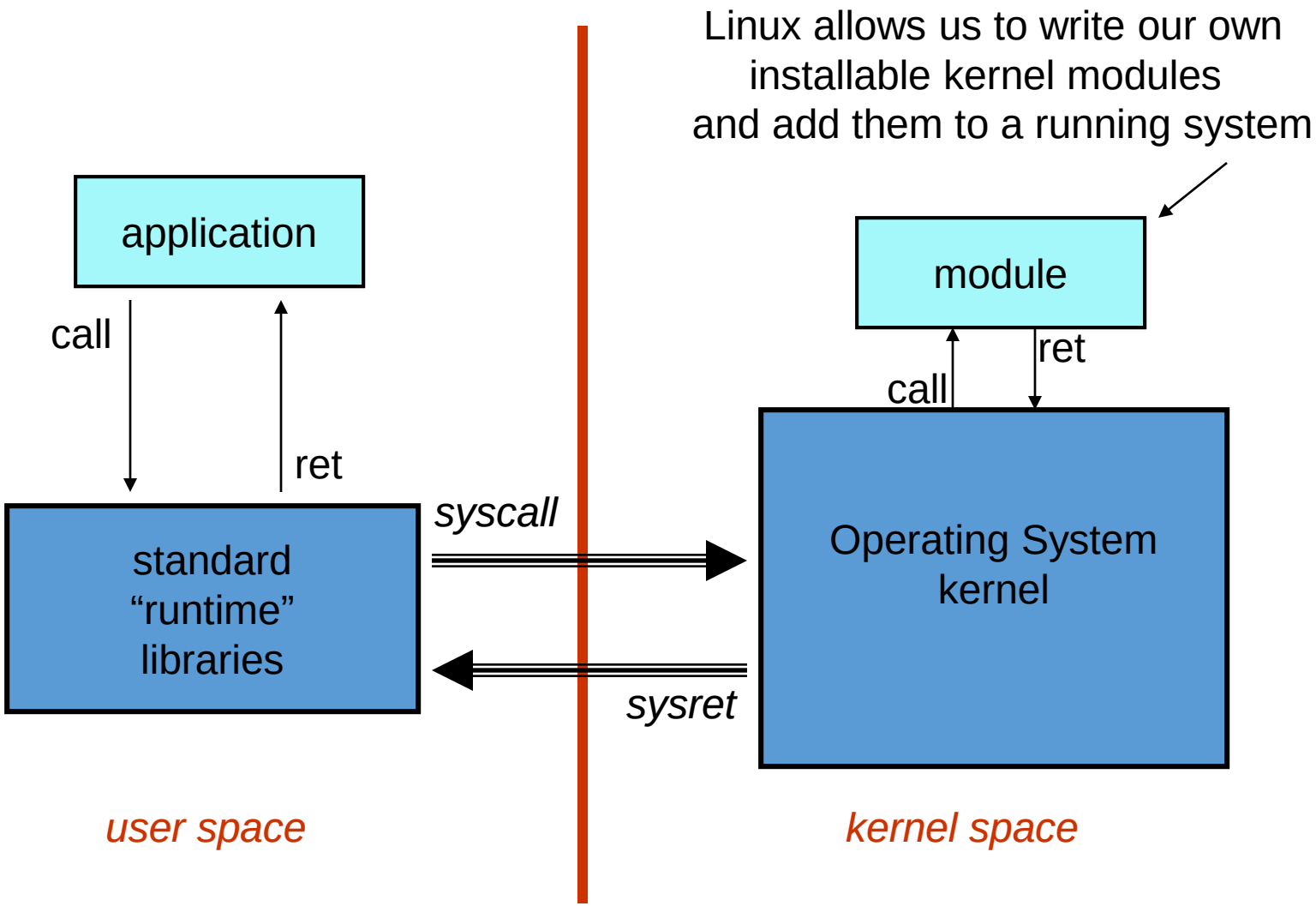
这种方法虽然编程简单，但是缺少用户空间与内核空间的互动性。如果想要再次获得进程信息则需要重新插入模块。

进阶实验

Overview of Operating Systems and Kernels



Linux Kernel Modules



实验目的

实现一个内核模块，该模块创建/proc/tasklist文件，并且提取系统中所有进程的pid、state和名称(comm)进行显示。

PROC虚拟文件系统

- **/proc 文件系统是一个虚拟文件系统（没有任何一部分与磁盘相关，只存在内存中，不占用外存空间），包含了一些目录和虚拟文件**

- **通过它可以在 Linux内核空间和用户空间之间进行通信：**
  - **可以向用户呈现内核中的一些信息（用cat、more等命令查看/proc文件中的信息）**
  - **也可以用作一种从用户空间向内核发送信息的手段（echo）**

- **LKM是用来展示/proc 文件系统的一种简单方法，因为它可以动态地向 Linux 内核添加或删除代码**

```
root@ubuntu:/proc# ls
1       1345  2409  2601  28    36    59  83           driver           pagetypeinfo
10      1346  2460  2604  2825  37    60  84           execdomains      partitions
1011    1350  2471  2613  2827  3740  61  85           fb               sched_debug
1045    1354  25    2617  2853  3755  62  87           filesystems      schedstat
1079    1367  250   2619  2858  3762  63  888          fs               scsi
1096    1377  2506  2628  2864  38    64  89           interrupts       self
1099    14    2509  2671  2869  388   65  9            iomem            slabinfo
11      140   2510  2674  2875  393   66  966          ioports          softirqs
1104    141   2524  2681  2886  394   67  967          irq              stat
1155    1438  2532  2683  29    398   68  972          kallsyms         swaps
116     15    2534  2687  2910  40    69  974          kcore            sys
1161    1513  2541  2689  2912  406   7   985          key-users        sysrq-trigger
1163    16    2550  2690  2921  407   70  986          kmsg             sysvipc
1166    17    2554  2691  2925  41    71  988          kpagecount       timer_list
1167    1777  2555  27    2982  42    72  acpi         kpageflags       timer_stats
117     18    2556  2715  2983  4247  73  asound       latency_stats    tty
1170    1826  2561  2726  2991  43    74  buddyinfo    loadavg          uptime
1179    1918  2564  2734  3     44    75  bus          locks            version
119     2     2566  2737  30    45    76  cgroups      mdstat           vmallocinfo
12      20    2568  2738  3004  5     77  cmdline      meminfo          vmstat
1220    21    2569  2743  31    54    78  consoles     misc             zoneinfo
1231    2106  2573  2744  32    540   79  cpuinfo      modules
1274    22    2575  2784  33    56    8   crypto       mounts
1275    23    2579  2791  3357  57    80  devices      mpt
1280    2341  2596  2792  34    58    81  diskstats    mtrr
13      238   26    2799  35    586   82  dma          net
root@ubuntu:/proc#
```

在用户空间对proc进行读写

- 和正常的文件读写一样
  - open, read, write, close
- 示例

  读：cat /proc/sys/kernel/printk

  写：echo 4096 /proc/sys/net/core/somaxconn

注：通过往/proc中写入来改变内核参数的操作重启之后无效

More ‘/proc’ examples

- **$ cat /proc/cpuinfo**
- **$ cat /proc/modules**
- **$ cat /proc/meminfo**
- **$ cat /proc/iomem**
- **$ cat /proc/devices**

[Read the ‘man-page’ for details: $ man proc]

相关函数

- **proc_create()**
  创建一个文件
- **proc_symlink()**
  创建符号链接
- **proc_mknod()**
  创建设备文件
- **proc_mkdir()**
  创建目录
- **remove_proc_entry()**
  删除文件或目录

创建新的/proc文件

一个虚拟文件或目录叫proc_dir_entry

- 文件：用proc_create函数创建, 函数声明在/include/linux/proc_fs.h文件中
- 目录：用proc_mkdir函数创建

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

- 输入参数：一个文件名、一组权限、所属上一级目录和文件操作函数集
- 返回值：proc_dir_entry 指针
  - 使用这个返回的指针来配置这个虚拟文件的其他参数
  - 为 NULL，说明在 create 时发生了错误

实验流程

1. 编写tasklist.c代码，并运行make进行编译

2. 内核模块添加 $sudo insmod tasklist.ko

3. 添加内核模块后读取tasklist信息： $ cat /proc/tasklist

代码整体tasklist.c已经给出，需要填充细节。

下面部分讲述代码逻辑

1、创建proc文件

使用proc_create函数创建文件。

```
my_proc_entry = proc_create(modname, 0x644, NULL, &my_proc);  //生成proc文件

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

函数参数的其他参数比较好理解，我们针对最后一个参数（file_operations类型参数）进行简单的介绍。

2. File Operations

- **file_operations structure 在 <linux/fs.h> 中定义**

- **Unix：万物皆文件**

通过定义 file_operations 结构来说明设备能完成的功能

大部分域是函数指针，程序设计者需要“完成”这些函数

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*read_iter) (struct kiocb *, const struct iovec *, ...;
        ssize_t (*write_iter) (struct kiocb *, const struct iovec *, ...;
        int (*iterate) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*mremap)(struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
        long (*fallocate)(struct file *file, int mode, loff_t offset,  loff_t len);
        void (*show_fdinfo)(struct seq_file *m, struct file *f);
        #ifndef CONFIG_MMU
           unsigned (*mmap_capabilities)(struct file *);
        #endif
};
```

file_operations参数

当创建proc文件之后需要“实现file_operations中的函数来对proc文件进行操作

对于：

```
static const struct file_operations my_proc =
{ //proc文件操作函数集合
  .owner      = THIS_MODULE,
  .open       = my_open,
  .read       = seq_read,
  .llseek     = seq_lseek,
  .release    = seq_release
};
```

.open指定了打开此设备文件时需要执行的函数

3、 seq_file方式操作proc文件。

seq_file定义了对proc虚拟文件的访问接口，可以方便快速地输出大容量文件内容

```
static int my_open(struct inode *inode, struct file *file)
{
    return seq_open(file, &my_seq_fops);  //打开序列文件并关联my_seq_fops
}
int seq_open(struct file *file, const struct seq_operations *op)

static struct seq_operations my_seq_fops =
{ //序列文件记录操作函数集合
  .start  = my_seq_start,
  .next   = my_seq_next,
  .stop   = my_seq_stop,
  .show   = my_seq_show
};
```

- 对proc_create生成的proc文件执行cat命令时，会调用open函数，执行seq_open

open→sys_open→…→file_operation的open函数（my_open），→seq_open

- seq_read 读 取 seq_file: start->show->next->show...->next->show->next->stop

$ strace cat /proc/tasklist

……

```
openat(AT_FDCWD, "/proc/tasklist", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|S_ISGID|S_ISVTX|0104, st_size=0, ...}) = 0
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
mmap(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f713a8f5000
read(3, "#2050\t 0\t0\tswapper/0\t\n", 131072) = 22
write(1, "#2050\t 0\t0\tswapper/0\t\n", 22#2050    0          0          swapper/0
) = 22
read(3, "#2051\t 1\t1\tsystemd\t\n", 131072) = 20
write(1, "#2051\t 1\t1\tsystemd\t\n", 20#2051      1          1          systemd
) = 20
read(3, "#2052\t 2\t1\tkthreadd\t\n", 131072) = 21
write(1, "#2052\t 2\t1\tkthreadd\t\n", 21#2052     2          1          kthreadd
) = 21
```

……

- my_seq_start, 从init_task开始遍历进程

```c
static void * my_seq_start(struct seq_file *m, loff_t *pos)
{
    //printk(KERN_INFO"Invoke start\n");    //可以输出调试信息
    if ( *pos == 0 )  // 表示遍历开始
    {
        task = &init_task; //遍历开始的记录地址
        return &task;    //返回一个非零值表示开始遍历
    }
    else //遍历过程中
    {
        if (task == &init_task )  //重新回到初始地址，退出
            return NULL;
        return (void*)pos ;//否则返回一个非零值
    }
}
```

/linux/v5.6/source/init/init_task.c
EXPORT_SYMBOL(init_task);

几个输出函数

- int seq_printf(struct seq_file *sfile, const char *fmt, ...);
  - 把给定参数按照给定的格式输出到seq_file文件，类似 printf，如果返回-1，意味着缓存区已满，部分输出被丢弃。
- int seq_putc(struct seq_file *sfile, char c);
  - 把一个字符输出到seq_file文 件
- int seq_puts(struct seq_file *sfile, const char *s);
  - 把一个字符串输出到seq_file文 件
- int seq_write(struct seq_file *seq, const void *data, size_t len)
  - 将data指向的数据写入seq_file缓存，数据长度为len。用于非字符串数据。

接着my_seq_show将我们需要的信息打印到seq_file，也就是tasklist中

```
static int my_seq_show(struct seq_file *m, void *v)
{ //获取进程的相关信息
  //printk(KERN_INFO"Invoke show\n");
  //输出进程序号
  seq_printf( m,  "#%-3d\t ", taskcounts++ );
  //输出进程pid?
  //输出进程state?
  //输出进程名称(comm)?
  seq_puts( m, "\n" );
  return 0;
}
```

然后my_seq_next将task指向下一个task_struct

```
static void * my_seq_next(struct seq_file *m, void *v, loff_t *pos)
{
  //printk(KERN_INFO"Invoke next\n");
  (*pos)++;
  //task指向下一个进程?
  return NULL;

}
```

然后内核提供的序列文件操作函数自动开始下一轮的操作：

实验结果

最后cat命令读取到的tasklist的内容如下

```
amos@ubuntu:~/Desktop/src$ sudo insmod tasklist.ko
amos@ubuntu:~/Desktop/src$ cat /proc/tasklist
#0         0        0          swapper/0
#1         1        1          systemd
#2         2        1          kthreadd
#3         3        1026       rcu_gp
#4         4        1026       rcu_par_gp
#5         6        1026       kworker/0:0H
#6         9        1026       mm_percpu_wq
#7         10       1          ksoftirqd/0
#8         11       1026       rcu_sched
#9         12       1          migration/0
#10        13       1          idle_inject/0
#11        14       1          cpuhp/0
#12        15       1          cpuhp/1
#13        16       1          idle_inject/1
#14        17       1          migration/1
#15        18       1          ksoftirqd/1
```

实验三

1.阅读并填充route_struct.c代码，完成热身小实验。要求：这次不用proc伪文件系统，直接在内核消息中打印出来（printk）并用dmesg查看即可。

2. 阅读并填充tasklist.c代码，然后完成实验。

3. 尝试在pid、state、comm后边添加打印出你感兴趣的task_struct中的字段（可以与作业相结合）

4. 观察seq_file输出过程

附加：阅读seq_file.c，解释实际输出过程