实验二 Linux 内核模块编程

17042127 陶逸群

设计要求

- 设计一个模块，要求列出系统中的所有内核线程的程序名、PID、 进程状态、进程优先级、父进程 PID。
- 设计一个带参数的模块，其参数为某个进程的 PID 号，模块的功 能是列出该进程的家族信息，包括父进程、兄弟进程和子进程的 程序名、PID 号以及进程状态。
- 模块二以树型方式打印输出
- 模块一按列对其输出
- 请根据自身情况，进一步阅读分析程序中用到的相关内核函数的 源码实现。

实验步骤

1. 编写内核模块
    - 打印出系统中所有内核线程信息的内核模块

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched/signal.h>
// 模块在加载的时候会运行init函数
static int __init init_show_all_kernel_thread(void)
{
    // 格式化输出头
    struct task_struct *p;
    printk("%-20s%-6s%-6s%-6s%-6s", "Name", "PID", "State", "Prio", "PPID");
    printk("-------------------------------------------");
    // for_each_process(p)的作用是从0开始遍历进程链表中的所有进程
    for_each_process(p)
    {
        // p最开始指向进程链表中第一个进程，随着循环不断进行p也不断后移直至链表尾
        if (p->mm == NULL){
            // 打印进程p的相关信息
            printk("%-20s%-6d%-6d%-6d%-6d", p->comm, p->pid, p->state, p->prio,
                    p->parent->pid);
        }
    }

    return 0;
}

// 模块在加载的时候会运行exit函数
static void __exit exit_show_all_kernel_thread(void)
{
```

```
    printk("---------所有进程显示完毕---------\n");
}
module_init(init_show_all_kernel_thread);
module_exit(exit_show_all_kernel_thread);

MODULE_LICENSE("GPL");
```

内核函数的具体作用见以上注释部分。

- 以树形打印某进程家族信息的内核模块

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/moduleparam.h>
#include <linux/pid.h>
#include <linux/list.h>
#include <linux/sched.h>

static int pid;
module_param(pid, int, 0644);

static int __init show_task_family_init(void){
    struct pid *ppid;
    struct task_struct *p;
    struct task_struct *pos;
    struct list_head *pos_head;
    int i=0;
    // 通过进程的PID号pid一步步找到进程的进程控制块p
    ppid = find_get_pid(pid);
    if (ppid == NULL){
        printk("pid不存在\n");
        return -1;
    }
    p = pid_task(ppid, PIDTYPE_PID);
    list_for_each(pos_head,&(p->children)){
        pos = list_entry(pos_head,struct task_struct,sibling);
        if(i==0){
            printk("%-10s(%-4d){%-6ld}---%-10s(%-4d){%-6ld}---%-10s(%-4d)
{%-6ld}", p->real_parent->comm,
                    p->real_parent->pid,p->real_parent->state, p->comm, p-
>pid, p->state,pos->comm, pos->pid,pos->state);
        }else{
            printk("%*s|%*s|-%-10s(%-4d){%-6ld}",25,"",26,"", pos->comm,
pos->pid,pos->state);
        }
        i++;
    }
    if(i==0){
                    printk("%-10s(%-4d){%-6ld}---%-10s(%-4d){%-6ld}", p-
>real_parent->comm,
                    p->real_parent->pid,p->real_parent->state, p->comm, p-
>pid, p->state);
    }
    list_for_each(pos_head,&(p->sibling)){
        if(pos_head!= &p->parent->children){
            pos = list_entry(pos_head,struct task_struct,sibling);
            printk("%*s|-%-10s(%-4d){%-6ld}",25,"", pos->comm, pos-
>pid,pos->state);
        }

    }
    return 0;
}
```

```
48
49  static void __exit show_task_family_exit(void)
50  {
51      printk("模块运行完毕\n");
52  }
53  MODULE_LICENSE("GPL");
54  module_init(show_task_family_init);
55  module_exit(show_task_family_exit);
```

在该模块中，首先调用函数 find_get_pid()并传入外部参数——某进 程的 pid 号来获得该进程的进程描述符，并调用函数 pid_task()获得该进 程的进程描述符信息，然后通过使用 Linux 内核中定义的 list_for_each() 宏和 list_entry()宏分别遍历该进程的子进程链表和兄弟进程链表，控制输出为树型。

2. 编写模块编译的 Makefile 文件
    - 打印出系统中所有内核线程信息的内核模块的 Makefile 文件

```
1  obj-m := show_all_kernel_thread_states.o
2  KDIR := /lib/modules/$(shell uname -r)/build
3  # 当前路径
4  PWD := $(shell pwd)
5  default:
6      make -C $(KDIR) M=$(PWD) modules
7  clean:
8      make -C $(KDIR) M=$(PWD) clean
```

    - 以树形打印某进程家族信息的内核模块的 Makefile 文件

```
1  obj-m := show_task_family.o
2  KDIR := /lib/modules/$(shell uname -r)/build
3  # 当前路径
4  PWD := $(shell pwd)
5  default:
6      make -C $(KDIR) M=$(PWD) modules
7  clean:
8      make -C $(KDIR) M=$(PWD) clean
```

3. 编译并加载内核模块

结果演示

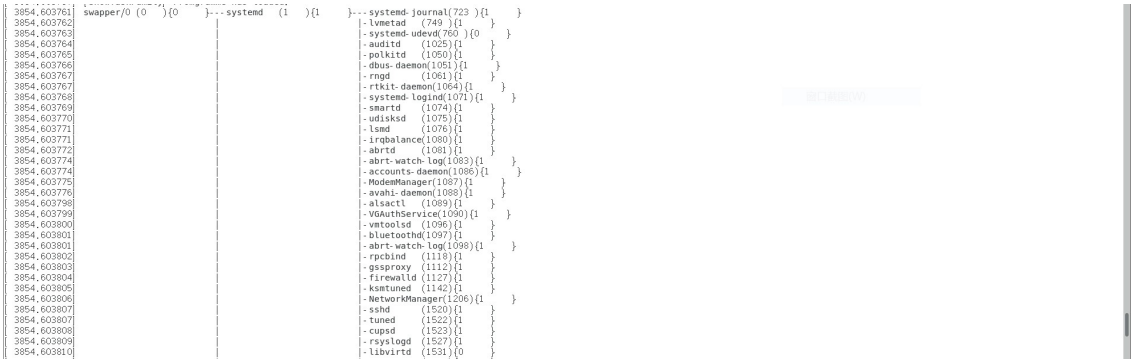
- 打印出系统中所有内核线程信息

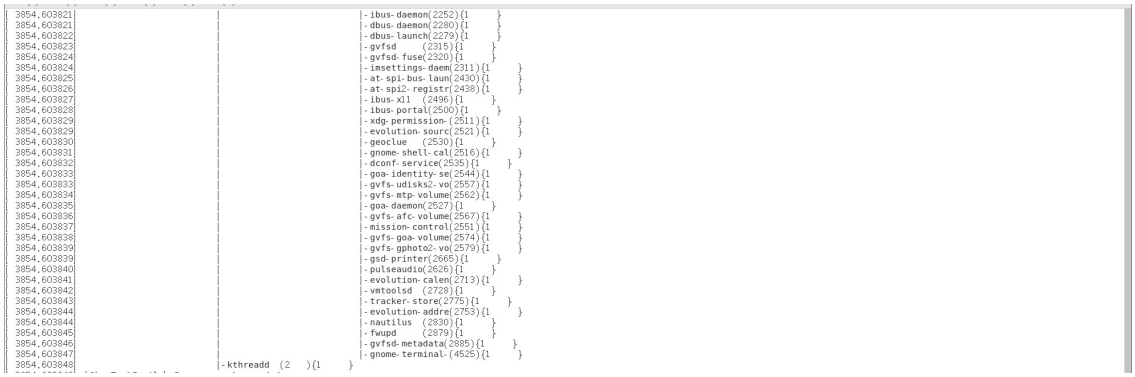
```
[root@localhost module_2]# ps -aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.1  0.2 193808  8536 ?        Ss   19:47   0:04 /usr/lib/syste
root           2  0.0  0.0      0     0 ?        S    19:47   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   19:47   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   19:47   0:00 [rcu_par_gp]
root           6  0.0  0.0      0     0 ?        I<   19:47   0:00 [kworker/0:0H-
root           8  0.0  0.0      0     0 ?        I<   19:47   0:00 [mm_percpu_wq]
root           9  0.0  0.0      0     0 ?        S    19:47   0:00 [ksoftirqd/0]
root          10  0.0  0.0      0     0 ?        I    19:47   0:03 [rcu_sched]
root          11  0.0  0.0      0     0 ?        S    19:47   0:00 [migration/0]
root          13  0.0  0.0      0     0 ?        S    19:47   0:00 [cpuhp/0]
root          14  0.0  0.0      0     0 ?        S    19:47   0:00 [cpuhp/1]
root          15  0.0  0.0      0     0 ?        S    19:47   0:00 [migration/1]
root          16  0.0  0.0      0     0 ?        S    19:47   0:00 [ksoftirqd/1]
root          18  0.0  0.0      0     0 ?        I<   19:47   0:00 [kworker/1:0H-
root          19  0.0  0.0      0     0 ?        S    19:47   0:00 [cpuhp/2]
root          20  0.0  0.0      0     0 ?        S    19:47   0:00 [migration/2]
root          21  0.0  0.0      0     0 ?        S    19:47   0:00 [ksoftirqd/2]
root          23  0.0  0.0      0     0 ?        I<   19:47   0:00 [kworker/2:0H-
root          24  0.0  0.0      0     0 ?        S    19:47   0:00 [cpuhp/3]
root          25  0.0  0.0      0     0 ?        S    19:47   0:00 [migration/3]
root          26  0.0  0.0      0     0 ?        S    19:47   0:00 [ksoftirqd/3]
root          28  0.0  0.0      0     0 ?        I<   19:47   0:00 [kworker/3:0H-
```

与 `ps - aux` 输出一致。

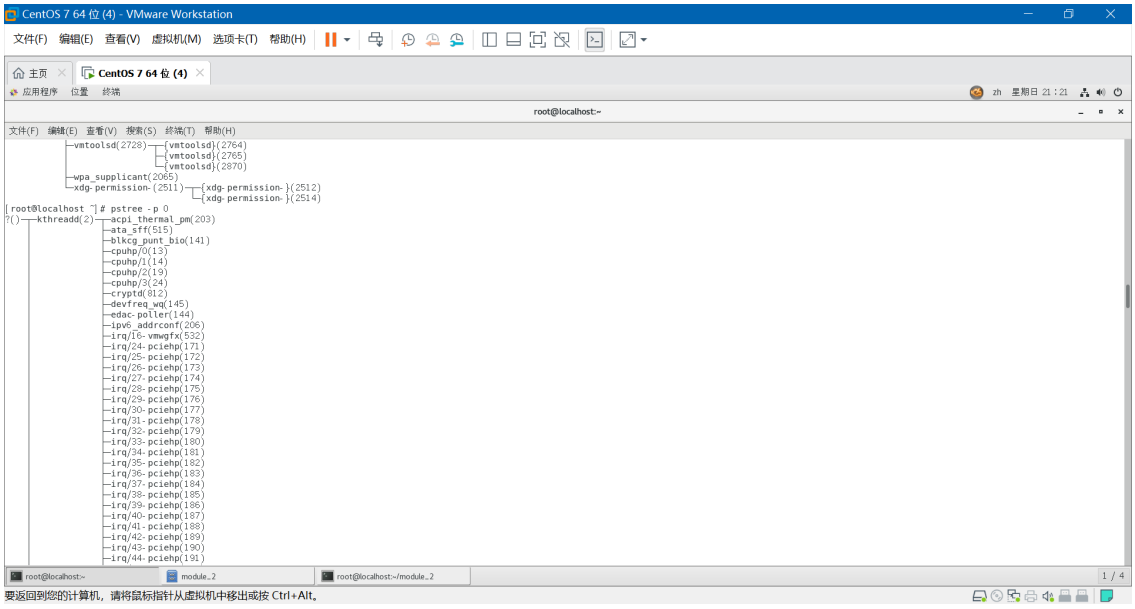

```
[root@localhost module_2]# ps -aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.1  0.2 193808  8536 ?        Ss   19:47   0:04 /usr/lib/syste
root           2  0.0  0.0      0     0 ?        S    19:47   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   19:47   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   19:47   0:00 [rcu_par_gp]
root           6  0.0  0.0      0     0 ?        I<   19:47   0:00 [kworker/0:0H-
root           8  0.0  0.0      0     0 ?        I<   19:47   0:00 [mm_percpu_wq]
root           9  0.0  0.0      0     0 ?        S    19:47   0:00 [ksoftirqd/0]
root          10  0.0  0.0      0     0 ?        I    19:47   0:03 [rcu_sched]
root          11  0.0  0.0      0     0 ?        S    19:47   0:00 [migration/0]
root          13  0.0  0.0      0     0 ?        S    19:47   0:00 [cpuhp/0]
root          14  0.0  0.0      0     0 ?        S    19:47   0:00 [cpuhp/1]
root          15  0.0  0.0      0     0 ?        S    19:47   0:00 [migration/1]
root          16  0.0  0.0      0     0 ?        S    19:47   0:00 [ksoftirqd/1]
root          18  0.0  0.0      0     0 ?        I<   19:47   0:00 [kworker/1:0H-
root          19  0.0  0.0      0     0 ?        S    19:47   0:00 [cpuhp/2]
root          20  0.0  0.0      0     0 ?        S    19:47   0:00 [migration/2]
root          21  0.0  0.0      0     0 ?        S    19:47   0:00 [ksoftirqd/2]
root          23  0.0  0.0      0     0 ?        I<   19:47   0:00 [kworker/2:0H-
root          24  0.0  0.0      0     0 ?        S    19:47   0:00 [cpuhp/3]
root          25  0.0  0.0      0     0 ?        S    19:47   0:00 [migration/3]
root          26  0.0  0.0      0     0 ?        S    19:47   0:00 [ksoftirqd/3]
root          28  0.0  0.0      0     0 ?        I<   19:47   0:00 [kworker/3:0H-
```

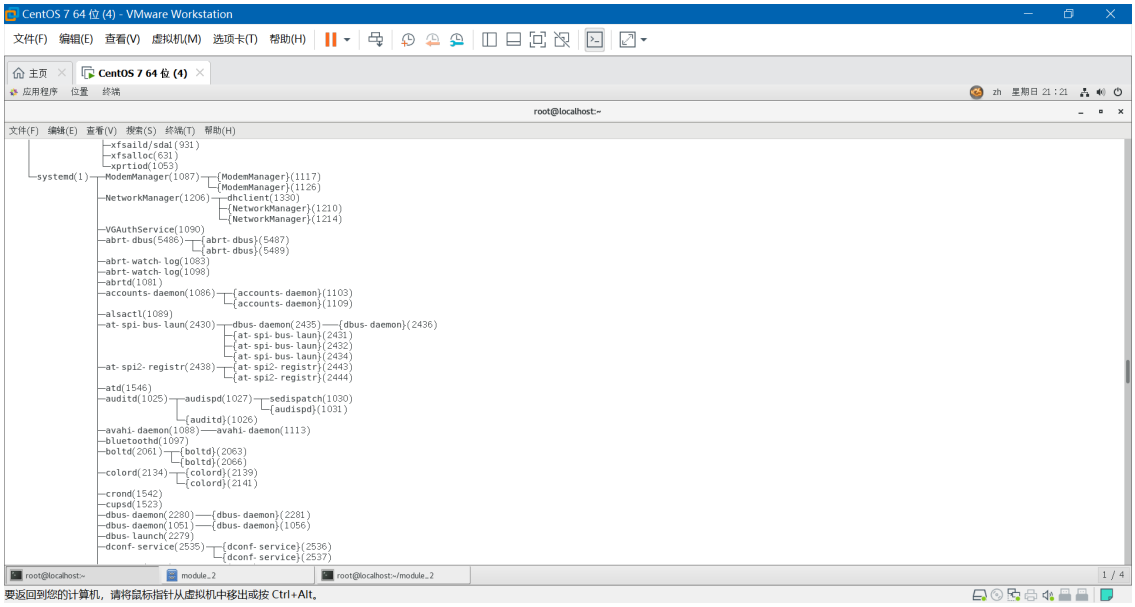

- 以树形打印某进程家族信息





这里的pid为1，与 `pstree - p 0` 结果一致





体会

本次实验让我对 Linux 系统以及 Linux 内核有了更加深入的认识，编写内核模块可以实现对内核功能的动态添加， 与通过直接修改内核添加系统调用实现添加内核功能相比更加灵活。 同时，通过本次实验我还了解了一些 Linux 内核中定义的宏，比如 list_f or_each()、for_each_process()、以及 list_entry()。