## 实验二报告：Linux内核模块编程

### 一、实验目的

Linux提供的模块机制能动态扩充Linux功能而无需重新编译内核，已经广泛应用在Linux内核的许多功能的实现中。在本实验中讲学习模块的基本概念、原理及实现技术，然后利用内核模块编程访问进程的基本信息，加深对进程概念的理解，掌握基本的模块编程技术。

### 二、实验内容

* 设计一个模块，要求列出系统中所有内核线程的程序名、PID、进程状态、进程优先级、父进程的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号及进程状态。
* 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。

### 三、设计方案

* 列出系统中所有内核现成的程序名、PID、进程状态、进程有限级、父进程的PID
  + 通过for\_each\_process函数遍历所有进程
  + 通过mm判断是否为内核线程，如果为null则为内核线程
* 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、进程状态
  + 父进程：就一个，用parent指针
  + 孩子进程：用list\_for\_each遍历，遍历孩子的兄弟的兄弟的兄弟…
  + 兄弟进程：遍历父亲的孩子的兄弟的兄弟的兄弟…，注意要判断是否是自身

### 四、实验过程

#### 第一个模块

* **模块代码**
* //头文件声明  
  #include <linux/init.h> //模块初始化和清理函数的定义  
  #include <linux/module.h> //加载模块所需要的的函数和符号定义  
  #include <linux/kernel.h> //printk函数需要  
  #include <linux/init\_task.h> //引入task\_struct定义  
  #include <linux/sched.h> //引入for\_each\_process()方法  
    
    
  //内核模块没有main函数，必须定义两个函数  
  //初始化函数用来完成模块注册和申请资源  
  //static使这个函数不会在特定文件之外可见  
  //如果只是初始化使用一次的话可以在生命语句中加\_\_init标识，则模块加载后会丢弃释放其内存空间  
  static int show\_all\_thread\_init(void)  
  {  
   int count=0;  
   struct task\_struct \*p;  
   printk(KERN\_ALERT"程序名\tPID\t状态\t优先级\t");  
   //线性遍历方式访问所有进程  
   /\*   
   for\_each\_process是宏循环控制语句，在/include/linux/sched/signal.h中  
   #define for\_each\_process(p) \  
   for (p = &init\_task ; (p = next\_task(p)) != &init\_task ; )  
   其中next\_task也是宏循环控制语句，在/include/linux/sched/signal.h中：  
   #define next\_task(p) \  
   list\_entry\_rcu((p)->tasks.next, struct task\_struct, tasks)  
   而task\_struct中有：  
   struct task\_struct {   
   ...  
   struct list\_head tasks;  
   ...  
   }  
  \*/  
   for\_each\_process(p)  
   {  
   //对Linux来说，用户进程和内核线程（kernel thread)都是task\_struct的实例，  
   //唯一的区别是kernel thread是没有进程地址空间的，内核线程也没有mm描述符的，所以内核线程的tsk->mm域是空（NULL）  
   //mm指向进程用户地址空间描述符，指向内存块  
   if(p->mm==NULL){  
   printk(KERN\_ALERT"%s\t%d\t%ld\t%d\n",  
   p->comm, p->pid, p->state, p->prio);  
   count++;  
   }  
   }  
   printk("一共有:%d个内核线程\n",count);  
   return 0;  
  }  
    
  //退出函数用来完成注销和释放资源  
  static void show\_all\_thread\_exit(void)  
  {  
   printk(KERN\_ALERT"show over\n");  
  }  
    
  module\_init(show\_all\_thread\_init);  
  module\_exit(show\_all\_thread\_exit);  
  MODULE\_LICENSE("GPL"); //宏生命此模块的许可证 GNU General Public License
* **Makefile文件**
* obj-m :=all\_thread.o  
  KDIR :=/lib/modules/$(shell uname -r)/build  
  PWD :=$(shell pwd)  
    
  default:  
   make -C $(KDIR) M=$(PWD) modules  
    
  clean:  
   make -C $(KDIR) M=$(PWD) clean

#### 第二个模块

* 模块代码
* #include<linux/init.h>  
  #include<linux/module.h>  
  #include<linux/kernel.h>  
  #include <linux/sched.h>  
  #include <linux/moduleparam.h>  
    
  static pid\_t pid=1;  
  module\_param(pid,int,0644);  
    
  //写的一个快速输出进程的信息的函数  
  void quick\_print(struct task\_struct\* p, char\* description){  
   printk("%-10s\tpid: %-9d 进程名: %-20s 进程状态: %ld\n",  
   description,p->pid, p->comm, p->state);  
  }  
    
  static int show\_family\_init(void)  
  {  
   struct task\_struct \*p;  
   struct list\_head \*pos;  
   struct task\_struct \*temp\_p;  
    
   //打印自己的pid与名称  
   p = pid\_task(find\_vpid(pid), PIDTYPE\_PID);  
   quick\_print(p, "自身信息");  
    
   // 父进程  
   if(p->parent == NULL) {  
   printk("无父进程\n");  
   }  
   else {  
   quick\_print(p->parent, "父进程");  
   }  
    
    
   list\_for\_each(pos, &p->parent->children)  
   {  
   temp\_p = list\_entry(pos, struct task\_struct, sibling);  
   //排除自己  
   if (temp\_p != p)  
   {  
   quick\_print(temp\_p, "兄弟进程");  
   }  
   }  
    
   // 子进程  
   list\_for\_each(pos, &p->children)  
   {  
   temp\_p = list\_entry(pos, struct task\_struct, sibling);  
   quick\_print(temp\_p, "子进程");  
   }  
   return 0;  
  }  
    
  static void show\_family\_exit(void)  
  {  
   printk(KERN\_ALERTs"show family over\n");  
  }  
    
  module\_init(show\_family\_init);  
  module\_exit(show\_family\_exit);  
    
  MODULE\_LICENSE("GPL");
* **Makefile文件**
* obj-m :=thread\_family.o  
  KDIR :=/lib/modules/$(shell uname -r)/build  
  PWD :=$(shell pwd)  
    
  default:  
   make -C $(KDIR) M=$(PWD) modules  
    
  clean:  
   make -C $(KDIR) M=$(PWD) clean

### 五、问题记录和总结

* module\_init()
* 一开始没看清想当然得理解成了module对象里面的init方法，结果就写成了module.init()，后来才发现，再想一下c语言里面也对象的思想啊
* 配置文件
  + Makefile文件复制过去的时候少了第一个字母，以为是程序的问题一直在检查.c文件，要注意vim打开文件的时候需要先敲一个字母a之类的才会进入insert模式，所以复制的时候经常会开头少掉一些字母
  + 第一行里面的obj-m :=show.o，我以为.o文件可以自己随便取名字就随便取了一个show.o结果一直既不报错也没编译成功，后来用排除法终于找到了原因，再后来发现指导书里面好像也有写。。没仔细看