实验二 编写内核模块

一、实验目的

Linux 操作系统的内核是单一体系结构(monolithic kernel)的，也就是说，整个内核是一

个单独的非常大的程序。这样，系统的速度和性能都很好，但是可扩展性和维护性就相对比

较差。为了弥补单一体系结构的这种缺陷，Linux操作系统使用了一种全新的机制－—模块

机制，用户可以根据需要，在不需要对内核重新编译的情况下，模块能动态地载入内核或从

内核移出。

本实验通过分析代码，学习Linux 是如何实现模块机制的；通过一个实例，掌握如何编

写模块程序并进一步掌握内核模块的机理。

二、题目内容：

（1）当模块被载入内核时会向系统日志文件中写入“hello,world”；当被卸载时，会向系统写入“goodbye”。

（2）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。

（3）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。

（4）设计一个带参数的模块，编写一个模块函数输入学号、姓名、数学成绩，输出学号、姓名，以及成绩优、良、中、及格、不及格等级。（需要自己参考上面代码来做）

（5）两个模块之间函数调用。

（6）编写一个两个模块，其中Average模块里面有一个模块函数，计算三个成绩（语文、数学、外语）的平均值，有三个形参。其中Student模块要接受三个（语文、数学、外语）的成绩，调用Average模块中的求平均的函数，输出平均值，还有语文、数学、外语成绩。

（需要自己参考上面代码来做）

四、实验内容

（1）

hello.c

#include <linux/module.h>

#include <linux/init.h>

#include <stdio.h>

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("yuming");

MODULE\_DESCRIPTION("Hello World Module");

MODULE\_ALIAS("a simplest module");

static int \_\_init hello\_init()

{

printf(KERN\_EMERG"Hello World!\n");

return 0;

}

static void \_\_exit hello\_exit()

{

printk("<6>hello exit\n");

printf(KERN\_EMERG"good bye!\n");

}

module\_init(hello\_init);

module\_exit(hello\_exit);

Makefile

ifneq ($(KERNELRELEASE),)

obj-m := hello.o

else

KDIR := /lib/modules/2.6.29/build

all:

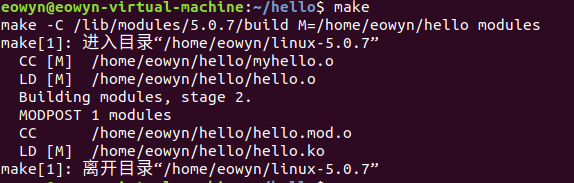
make -C $(KDIR) M=$(PWD) modules

clean:

rm -f \*.ko \*.o \*.mod.o \*.mod.c \*.symvers

endif

结果验证：

输入make

输入insmod ./hello.ko加载模块命令

https://img-blog.csdnimg.cn/20190703134152490.png

输入lsmod查看模块是否被编译

https://img-blog.csdnimg.cn/20190703134152262.png

输入dmesg查看是否在日志输出

https://img-blog.csdnimg.cn/20190703134152575.png

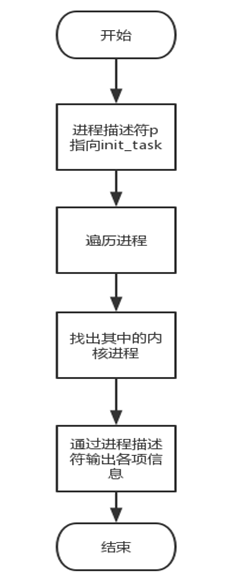
输入rmmod hello卸载模块

https://img-blog.csdnimg.cn/20190703134152588.png

输入dmesg查看结果

https://img-blog.csdnimg.cn/20190703134152583.png

（2）



module1.c

#include <linux/init.h>  
#include <linux/module.h>  
#include <linux/kernel.h>  
#include <linux/sched.h>  
#include <linux/init\_task.h>  
static int hello\_init(void)

{struct task\_struct \*p;

/\*进程描述符指针p\*/

p=&init\_task;

/\*p指向init\_task

内核中init\_task变量就是是进程0（调度程序）使用的进程描述符，也是Linux系统中第一个进程描述符

，贯穿于整个Linux系统的初始化过程，所以不可能不存在\*/

printk(KERN\_ALERT"名称\t进程号\t状态\t优先级\t父进程号\t\n");

for\_each\_process(p)

/\* for\_each\_process是宏循环控制语句，内核开发者可它扫描整个进程链表。

#define for\_each\_process(p) \

       for (p = &init\_task ; (p = next\_task(p)) != &init\_task ; )

一直循环下去，直到再次碰上init\_task为止，因为循环链表\*/

{if(p->mm == NULL){

/\* mm是指向被映射的用户地址空间的内存管理结构的指针

       struct mm\_struct \*mm, \*active\_mm;

#ifdef CONFIG\_COMPAT\_BRK

       unsigned brk\_randomized:1;

#endif

#if defined(SPLIT\_RSS\_COUNTING)

       struct task\_rss\_stat rss\_stat;

#endif

内核进程始终在内核空间运行，从来不切换到用户空间去，所以没有用户态地址空间，所以它们的mm成员

总是为NULL\*/

printk(KERN\_ALERT"%s\t%d\t%ld\t%d\n",p->comm,p->pid, p->state,p->normal\_prio,p->parent->pid);

/\*

struct task\_struct {

volatile long state;

int prio, static\_prio, normal\_prio;

pid\_t pid;

char comm[TASK\_COMM\_LEN];

…};

相应的程序名char common[TASK\_COMM\_LEN]

进程标识符pid\_t pid

进程状态volatile long state

常规静态优先级int normal\_prio

\*/

}

}

return 0;

}

// 清理函数

static void hello\_exit(void)

{

printk(KERN\_ALERT"goodbye~\n");

}

// 函数注册

module\_init(hello\_init);

module\_exit(hello\_exit);

// 模块许可申明

MODULE\_LICENSE("GPL");

Makefile

ifneq ($(KERNELRELEASE),)  
obj-m := module1.o  
else  
KDIR := /lib/modules/2.6.29/build  
all:

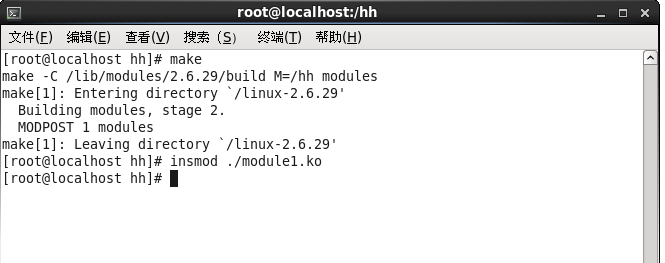
make -C $(KDIR) M=$(PWD) modules

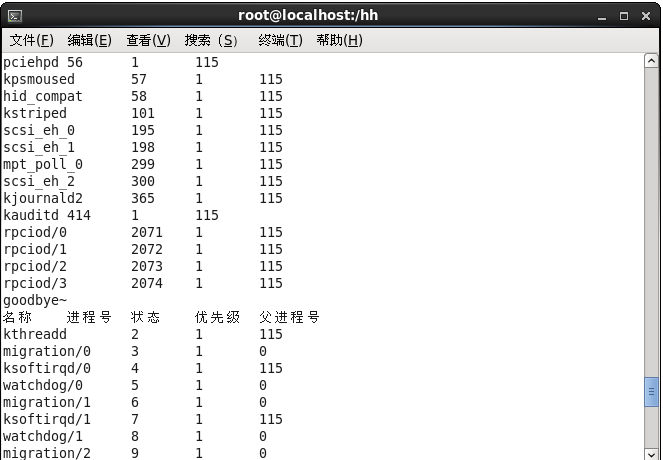
clean:

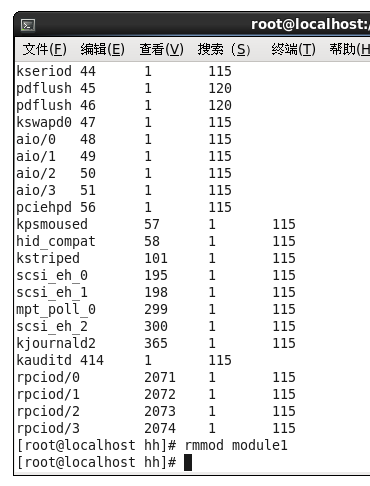
rm -f \*.ko \*.o \*.mod.o \*.mod.c \*.symvers  
endif

结果：

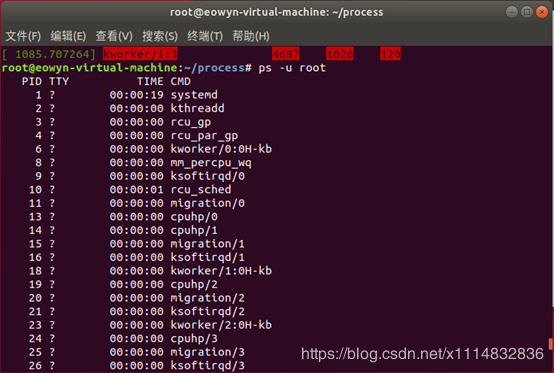






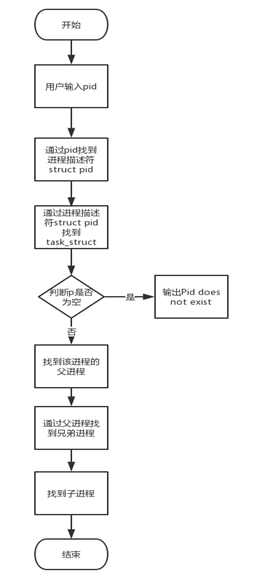


检验：输入ps -u root查看root用户的进程



经检查与结果一致

（3）



list.c

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/sched.h>

#include <linux/moduleparam.h>

#include <linux/list.h>

static int pid;

/\*module\_param(name,type,perm)

name既是用户看到的参数名，有事模块内接收参数的变量

type表示数据的类型，例如：int,double,bool,unit...

perm对于全局可读的变量为0444，对于根目录可写为0644。

\*/

module\_param(pid, int ,0644);

static int se\_init(void)

{

struct task\_struct \*ts;

struct pid \*p;

p=find\_get\_pid(pid);

ts=pid\_task(p,PIDTYPE\_PID);

struct task\_struct \*parent;

printk(KERN\_ALERT "second start!");

parent=ts->parent;

printk(KERN\_ALERT"父名称\t进程号\t状态\t优先级\t\n");

printk(KERN\_ALERT"%s\t%d\t%d\t%d\n",parent->comm,parent->pid,parent->state,parent->prio);

struct task\_struct \*list\_sibling;

struct list\_head \*pos\_sibling;

printk(KERN\_ALERT"兄弟名称\t进程号\t状态\t优先级\t\n");

list\_for\_each(pos\_sibling,&ts->sibling)

{

list\_sibling=list\_entry(pos\_sibling,struct task\_struct,sibling);

printk(KERN\_ALERT"%s\t%d\t%d\t%d\n",list\_sibling->comm,list\_sibling->pid,list\_sibling->state,list\_sibling->prio);

}

struct task\_struct \*list\_children;

struct list\_head \*pos\_children;

printk(KERN\_ALERT"孩子名称\t进程号\t状态\t优先级\t\n");

list\_for\_each(pos\_children,&ts->children)

{ list\_children=list\_entry(pos\_children,struct task\_struct,sibling);

printk(KERN\_ALERT"%s\t%d\t%d\t%d\n",list\_children->comm,list\_children->pid,list\_children->state,list\_children->prio);

} return 0;

}

static void se\_exit(void)

{ printk(KERN\_ALERT "second exit\n"); }

module\_init(se\_init);

module\_exit(se\_exit);

MODULE\_LICENSE("GPL");

Makefile

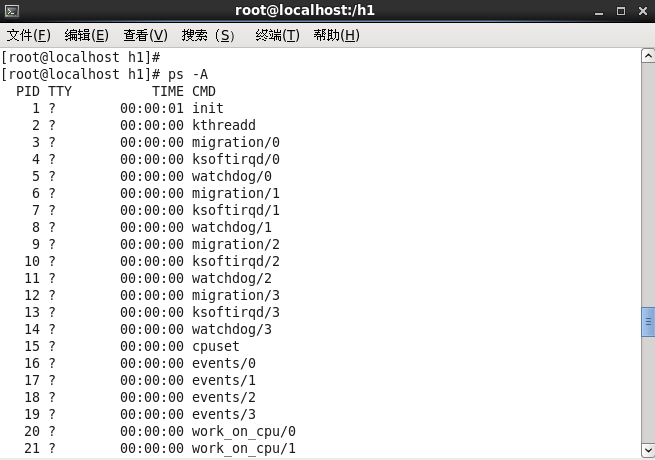
ifneq ($(KERNELRELEASE),)  
obj-m := module1.o  
else  
KDIR := /lib/modules/2.6.29/build  
all:

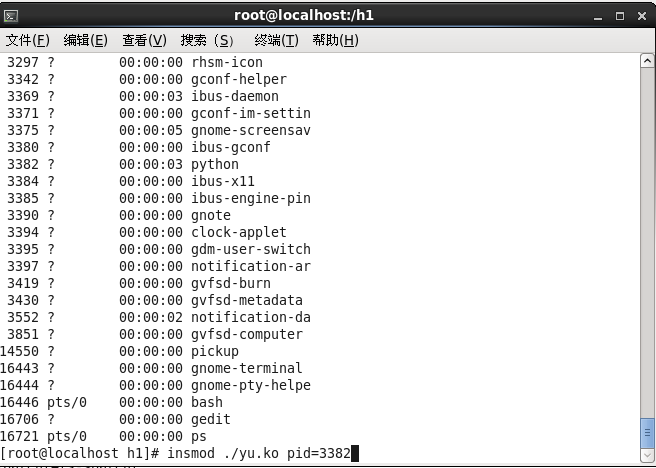
make -C $(KDIR) M=$(PWD) modules

clean:

rm -f \*.ko \*.o \*.mod.o \*.mod.c \*.symvers  
endif

测试：

查看系统中的进程

加载模块，需要给出查看进程的pid



（4）两个模块之间函数调用

模块之间发生调用关系是常有的事情，下面以两个模块A、B，B

使用A模块提供的函数为例，讲解正确使用的方法。

模块A中使用EXPORT\_SYMBOL或EXPORT\_SYMBOL\_GPL将要提供给B

模块的函数导出；

模块A代码和模块B代码放在不同的文件夹里

模块B中用extern 声明需要用到的A模块提供的函数。

代码如下：

模块A的代码 –

A\_func.c

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/init.h>

void A\_print\_messages(void)

{

printk("我是被B模块调用执行的");

return;

}

EXPORT\_SYMBOL(A\_print\_messages);

static int \_\_init A\_init(void)

{

printk("A\_func module init!\n");

return 0;

}

static void \_\_exit A\_exit(void)

{

printk("A\_func module exit!\n");

return;

}

module\_init(A\_init);

module\_exit(A\_exit);

MODULE\_AUTHOR("nefu");

MODULE\_DESCRIPTION("Module A");

MODULE\_VERSION("0.1");

MODULE\_LICENSE("GPL");

模块B的代码 –

B\_func.c

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/init.h>

#include <linux/jiffies.h>

extern void A\_print\_jiffies(void);

static int \_\_init B\_init(void)

{

printk("B\_func module init!\n");

A\_print\_jiffies();

return 0;

}

static void \_\_exit B\_exit(void)

{

printk("B\_func module exit!\n");

return;

}

module\_init(B\_init);

module\_exit(B\_exit);

MODULE\_AUTHOR("nefu");

MODULE\_DESCRIPTION("Module B!");

MODULE\_VERSION("0.1");

MODULE\_LICENSE("GPL");

模块A的Makefile

obj-m := A\_func.o

KERNELDIR := /lib/modules/$(shell uname -r)/build

PWD := $(shell pwd)

default:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

clean:

rm -f \*.o \*.ko \*.mod.c \*.order \*.symvers

模块B的Makefile

obj-m := B\_func.o

KERNELDIR := /lib/modules/$(shell uname -r)/build

PWD := $(shell pwd)

default:

$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

clean:

rm -f \*.o \*.ko \*.mod.c \*.order \*.symvers

接下来，使得模块B编译及加载不出现Warning或Failed.

方法：

A模块在make之后，会产生一个Module.symvers文件，将该文件拷贝到B模块源文件目录中，然后执行make