

# Linux大作业

## 题目选择：

在Linux内核中增加一个系统调用，并编写对应的linux应用程序。利用该系统调用能够遍历系统当前所有进程的任务描述符，并按进程父子关系将这些描述符所对应的进程id（PID）组织成树形结构显示。

## 前期调研：

关于方法的选择，总体来说，关于添加新的系统调用有两种方法：内核模块法和编译内核法。

重新编译内核：即在内核源码中，找到包含系统调用号的文件，在其中添加系统调用编号、系统调用跳转表和相应历程。

添加内核模块：通过将增加系统调用的所有指令封装成一个模块，并在其中实现新的系统调用的功能函数。

比较两种方法，我们发现修改内核源码来实现系统调用不仅步骤繁琐，更需要重新编译内核源码，需要花费较多时间。和修改内核源码相比，添加内核模块的做法更加符合模块化程序的设计思想，设计思想也相对比较清晰。符合程序设计模块化的思想。操作也更加便捷，因此本实验选择内核模块法来完成系统调用的新增。

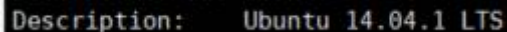
在本实验中，系统的版本及内核版本如下所示

系统版本和内核版本：

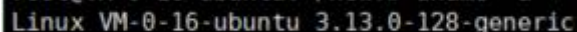
ubuntu 14.04.1 LTS

3.13.0-128-generic

32位



```
Description:      Ubuntu 14.04.1 LTS
```



```
Linux VM-0-16-ubuntu 3.13.0-128-generic
```

## 设计思路

整个程序的构思是将增加系统调用号的所有操作在一个文件中体现，之后将该程序运行得到内核模块，将内核模块加载进入系统内核中，之后利用测试程序测试内核模块是否添加成功以及新增的系统调用的功能是否能够实现。

## 各模块的说明：

1、首先我们先进入内核查看系统调用表，找到空余的系统调用号以及系统调用表所对应的内存地址以用于添加新的系统调用。系统调用表所在路径位 `/boot/System.map-3.13.0-128-generic`

从下图可以发现有两个空余的系统调用号222、223没有被使用，选择222号作为本实验所要求的新的系统调用号。



```
#define __NR_madvise 219
#define __NR_getdents64 220
#define __NR_fcntl64 221

#define __NR_gettid 224
#define __NR_readahead 225
```

在 `/boot/System.map-3.13.0-128-generic` 当我们查看系统调用表时可以看到系统调用表只有“读”权限，因此添加系统调用号之前首先要对内存区域进行权限修改

```
root@VM-0-16-ubuntu:/boot# grep sys_call_table System.map-3.13.0-128-generic
c1672140 R sys_call_table
```

2、关于修改寄存器的权限属性：利用内嵌汇编代码来实现。

`asm("movl %%cr0, %%eax" : "=a"(cr0));` 是Linux C中内嵌的汇编代码，格式位 `"movl %1,%0" : "=r"(result) : "m"(input)`，`"movl %1,%0"` 为指令模板，冒号后面是每个操作数对应的C表达式（括号的内容）和表达式的说明和限制（引号里的内容）。冒号后第一项对应的是%0，第二项对应的是%1。

```
unsigned int clear_and_return_cr0(void) // 寄存器权限修改函数
{
    unsigned int cr0 = 0;
    unsigned int ret;
    asm("movl %%cr0, %%eax" : "=a"(cr0)); // 输入部分为空，即直接从cr0寄存器中取数；输出部分为cr0
    // 变量，a表示将cr0和eax相关联。这句话的作用是cr0寄存器中的值就赋给了变量cr0
    ret = cr0;
    cr0 &= 0xffffefff; // 增加写权限（第17位为0代表内核空间可写）
    asm("movl %%eax, %%cr0" : "a"(cr0)); // 输出部分为空，即直接将结果输出到cr0寄存器。输入部分为
    // 变量cr0，它和eax寄存器相关联。这句话的作用是变量cr0的值赋给了寄存器cr0
    return ret;
}
```

3、构造树形图函数

```
void processtree(struct task_struct * p, int b) // 进程树图形结构函数
{
    struct list_head * l;
    a[counter].pid = p -> pid; // 获取进程的序号
    a[counter].depth = b; // 获取进程的深度
    counter++;
    for(l = p -> children.next; l != &(p->children); l = l->next)
    {
        struct task_struct * t = list_entry(l, struct task_struct, sibling);
        processtree(t, b+1);
    }
}
```

4、用 `sys_mycall` 作为新的系统调用，功能是遍历所有进程的并记录任务描述符

```

void processtree(struct task_struct * p,int b)//进程树图形结构函数
{
    struct list_head * l;
    a[counter].pid = p -> pid;
    a[counter].depth = b;
    counter ++;
    for(l = p -> children.next; l != &(p->children); l = l->next)
    {
        struct task_struct *t = list_entry(l,struct task_struct,sibling);
        processtree(t,b+1);
    }
}

```

## 5、将要新增的系统调用号的函数加入到内核中初始化内核

```

static int __init init_addsyscall(void)
{
    printk("add_newkernel\n");
    sys_call_table = (unsigned long *)sys_call_table_address;//获取系统调用服务首地址
    printk("%x\n",sys_call_table);
    anything_saved = (int (*)(void)) (sys_call_table[my_syscall_num]);//保存原始系统调用的地址
    orig_cr0 = clear_and_return_cr0();//调用上面的寄存器修改函数来对系统调用表的内存地址增加写权限
    sys_call_table[my_syscall_num]= (unsigned long)&sys_mycall;//更改原始的系统调用服务地址
    setback_cr0(orig_cr0);//设置为原始的只读cr0
    return 0;
}

```

## 6、内核模块的加载与卸载

对于内核构造函数 `module_init(init_addsyscall);`, 函数原型必须是 `module_init()`, 括号内的是函数指针. 模块析构函数为 `module_exit(exit_addsyscall);`, 函数原型为 `module_exit();`, 此函数在执行卸载内核模块时会被调用。 `MODULE_LICENSE("GPL")`; 是模块许可声明, 是内核程序使用的许可证。

```

module_init(init_addsyscall);
module_exit(exit_addsyscall);
MODULE_LICENSE("GPL");

```

## 7、内核模块卸载函数

```

static void __exit exit_addsyscall(void)//移除内核模块函数
{
    //设置cr0中对sys_call_table的更改权限。
    orig_cr0 = clear_and_return_cr0();//设置cr0可更改
    //恢复原有的中断向量表中的函数指针的值。
    sys_call_table[my_syscall_num]= (unsigned long)anything_saved;
    //恢复原有的cr0的值
    setback_cr0(orig_cr0);
    printk("remove new_kernel_module\n");
}

```

## 遇到的问题及解决方法：

- 内核空间与用户空间不能直接访问，如何在用户空间显示内核中的进程树图？

利用 `copy_to_user` 函数来实现内核与用户空间的数据交换，`copy_to_user(void __user *to, const void *from, unsigned long n)`，这里的 `to` 是目标地址即用户空间地址，`from` 是原地址即内核地址；`From` 源地址，`n` 是将要拷贝的数据的字节数。

- 对于用户空间中用于存储进程信息的数组的大小应该如何选择？

数组大小的设置原则是不让数据溢出，首先查看系统所能支持的最大的进程数，本系统中为

```
14951 root@VM-0-16-ubuntu:~/newf# ulimit -u 14951
```

，但是实际运行的进程数远小于这个数，在实验中，我们选择的数组大小是1000，查看当前执行的程序数量：

```
14951 root@VM-0-16-ubuntu:~/newf# ps -ef|wc -l 80
```

，80远小于1000，故不会存在数据溢出的情况。

- 如何让判断系统调用模块添加到了内核中是否成功？

利用 `dmesg` 命令可查看内核中环形缓冲区信息，看输出是否正确。

## 程序结果：

- 1、加载内核模块：

```
root@VM-0-16-ubuntu:~/folder1227# lsmod |head
Module              Size  Used by
hello                20839  0
crc32_pclmul         12967  0
aesni_intel          18156  0
aes_i586             16995  1 aesni_intel
xts                  12749  1 aesni_intel
lrw                  13057  1 aesni_intel
gf128mul             14503  2 lrw,xts
ablk_helper          13357  1 aesni_intel
cryptd               15578  1 ablk_helper
root@VM-0-16-ubuntu:~/folder1227#
```

可以看到出现了一个hello模块

- 2、运行测试程序

```
ubuntu@VM-0-16-ubuntu:~/folder1227$ vim hello_test.c
ubuntu@VM-0-16-ubuntu:~/folder1227$ cc hello_test.c
ubuntu@VM-0-16-ubuntu:~/folder1227$ ./a.out
the result is:8000
0
|→-1
|→-|→-253
|→-|→-307
|→-|→-323
|→-|→-332
|→-|→-339
|→-|→-493
|→-|→-524
|→-|→-701
|→-|→-753
|→-|→-756
|→-|→-761
|→-|→-762
|→-|→-764
|→-|→-785
|→-|→-|→-10202
|→-|→-|→-|→-10250
|→-|→-|→-|→-|→-10251
|→-|→-|→-|→-|→-|→-10525
|→-|→-|→-|→-10523
|→-|→-|→-|→-10524
|→-|→-789
|→-|→-798
|→-|→-799
|→-|→-986
|→-|→-1570
|→-|→-|→-1586
|→-|→-|→-1587
|→-|→-1888
|→-|→-2076
|→-|→-2129
|→-|→-2130
```

```
|→- |→- 2129
|→- |→- 2130
|→- |→- 2140
|→- 2
|→- |→- 3
|→- |→- 5
|→- |→- 7
|→- |→- 8
|→- |→- 9
|→- |→- 10
|→- |→- 11
|→- |→- 12
|→- |→- 13
|→- |→- 14
|→- |→- 15
|→- |→- 16
|→- |→- 17
|→- |→- 18
|→- |→- 19
|→- |→- 20
|→- |→- 21
|→- |→- 22
|→- |→- 23
|→- |→- 25
|→- |→- 26
|→- |→- 27
|→- |→- 28
|→- |→- 29
|→- |→- 30
|→- |→- 31
|→- |→- 32
|→- |→- 44
|→- |→- 46
|→- |→- 47
|→- |→- 48
|→- |→- 70
|→- |→- 71
```

```
|→- |→- 71
|→- |→- 117
|→- |→- 118
|→- |→- 125
|→- |→- 126
|→- |→- 237
|→- |→- 374
|→- |→- 1500
|→- |→- 24039
|→- |→- 19546
ubuntu@VM-0-16-ubuntu:~/folder1227$ █
```

### 3、卸载内核模块

```
root@VM-0-16-ubuntu:~/folder1227# lsmod | head
Module                Size  Used by
crc32_pclmul          12967  0
aesni_intel           18156  0
aes_i586              16995  1 aesni_intel
xts                   12749  1 aesni_intel
lrw                   13057  1 aesni_intel
gf128mul              14503  2 lrw,xts
ablk_helper           13357  1 aesni_intel
cryptd                15578  1 ablk_helper
cirrus                24370  1
root@VM-0-16-ubuntu:~/folder1227# █
```

#### 4、卸载内核后运行测试程序

```
root@VM-0-16-ubuntu:~/folder1227# ./hello_test
the result is:-1
0
```

#### 5、利用 `dmesg` 指令查看输出是否正确

```
root@VM-0-16-ubuntu:~/folder1227# dmesg
[1001668.373012] creat_newkernel
[1001668.373511] c1672140
[1001675.179748] add_newkernel!
[1001834.050239] remove new_kernel_module
root@VM-0-16-ubuntu:~/folder1227#
```

### 源代码：

hello.c

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/unistd.h>
#include <asm/uaccess.h>
#include <linux/sched.h>

#define my_syscall_num 222 //新增系统调用的调用号
#define sys_call_table_address 0xc1672140//系统调用表的地址

static int counter = 0;
struct process
{
    int pid;
    int depth;
};

struct process a[1000];

unsigned int clear_and_return_cr0(void);
void setback_cr0(unsigned int val);
asmlinkage long sys_mycall(char __user *buf);//asmlinkage的作用是告诉编译器，函数参数不是用寄存器来传递，而是用堆栈来传递的
int orig_cr0;
unsigned long *sys_call_table = 0;
static int (*anything_saved)(void);

unsigned int clear_and_return_cr0(void)//寄存器权限修改函数
{
    unsigned int cr0 = 0;
    unsigned int ret;
    asm("movl %%cr0, %%eax":"=a"(cr0)); //输入部分为空，即直接从cr0寄存器中取数；输出部分为cr0
    //变量，a表示将cr0和eax相关联。这句话的作用是cr0寄存器中的值就赋给了变量cr0
    ret = cr0;
```

```

    cr0 &= 0xfffffff; //增加写权限（第17位为0代表内核空间可写）
    asm("movl %%eax, %%cr0"::"a"(cr0)); //输出部分为空，即直接将结果输出到cr0寄存器。输入部分为
    变量cr0，它和eax寄存器相关联。这句话的作用是变量cr0的值赋给了寄存器cr0
    return ret;
}

void processtree(struct task_struct * p, int b) //进程树图形结构函数
{
    struct list_head * l;
    a[counter].pid = p -> pid;
    a[counter].depth = b;
    counter ++;
    for(l = p -> children.next; l != &(p->children); l = l->next)
    {
        struct task_struct * t = list_entry(l, struct task_struct, sibling);
        processtree(t, b+1);
    }
}

asmlinkage long sys_mycall(char __user * buf)
{
    int b = 0;
    struct task_struct * p;
    printk("creat_newkernel!\n");

    for(p = current; p != &init_task; p = p->parent ); //遍历所有进程并调用进程树图函数得到树
    状图
        processtree(p, b);

    if(copy_to_user((struct process *)buf, a, 1000*sizeof(struct process))) //将内核空间的数
    据拷贝到用户空间
        return -EFAULT; // (为什么不返回0??)
    else
        return sizeof(a); //拷贝失败返回拷贝地址的字节数
}

void setback_cr0(unsigned int val)
{
    asm volatile("movl %%eax, %%cr0"::"a"(val)); //读取val的值到eax寄存器，再将eax寄存器的值
    放入cr0中
}

static int __init init_addsyscall(void)
{
    printk("add_newkernel\n");
    sys_call_table = (unsigned long *)sys_call_table_address; //获取系统调用服务首地址
    printk("%x\n", sys_call_table);
    anything_saved = (int (*)(void)) (sys_call_table[my_syscall_num]); //保存原始系统调用的
    地址
    orig_cr0 = clear_and_return_cr0(); //调用上面的寄存器修改函数来对系统调用表的内存地址增加写权
    限
    sys_call_table[my_syscall_num] = (unsigned long)&sys_mycall; //更改原始的系统调用服务地址
    setback_cr0(orig_cr0); //设置为原始的只读cr0
    return 0;
}

```



```

static void __exit exit_addsyscall(void)//移除内核模块函数
{
    //设置cr0中对sys_call_table的更改权限。
    orig_cr0 = clear_and_return_cr0();//设置cr0可更改
    //恢复原有的中断向量表中的函数指针的值。
    sys_call_table[my_syscall_num]= (unsigned long)anything_saved;
    //恢复原有的cr0的值
    setback_cr0(orig_cr0);
    printk("remove new_kernel_module\n");
}

module_init(init_addsyscall);
module_exit(exit_addsyscall);
MODULE_LICENSE("GPL");

```

## hello\_test.c

```

#include <linux/unistd.h>
#include <syscall.h>
#include <sys/types.h>
#include <stdio.h>

struct process//获取进程的序号以及深度
{
    int pid;
    int depth;
};

struct process a[1000];

int main()
{
    int i,j;

    printf("the result is:%d\n",syscall(222,&a));

    for(i = 0; i < 1000; i++)
    {
        for(j = 0; j < a[i].depth; j++)
            printf("|-->-");
        printf("%d\n",a[i].pid);
        if(a[i+1].pid == 0)
            break;
    }
    return 0;
}

```

## Makefile

```

KVERS = $(shell uname -r)

# Kernel modules
obj-m += hello.o

# Specify flags for the module compilation.
#EXTRA_CFLAGS=-g -O0

```

```
build: kernel_modules user_test

kernel_modules:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) modules
user_test:
    gcc -o hello_test hello_test.c

clean:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) clean
```