|  |
| --- |
| 1. 实验目的及实验内容   （本次实验所涉及并要求掌握的知识；实验内容；必要的原理分析） |
| **实验目的：**  内核模块的编写：完成一个Linux/Windows内核/驱动模块的编写，  能够实现对文件访问的监控、或者对键盘设备、USB设备、网络设备、  蓝牙设备等的监控。  **实验内容：**  通过linux内核模块编程，写一个模块使得进程访问文件时会把进程的进程名，进程id，文件名打印到系统日志，然后通过dmesg查看相关信息。 |
| 1. 实验环境及实验步骤   （本次实验所使用的器件、仪器设备等的情况；具体的实验步骤） |
| 实验环境：  VMwareWorkstationPro 15.5.0  Virtual Box 6.1  Ubuntu 12.04.5 desktop i386 32位  bochs 2.6.9  关键技术：  Linux内核编程  实验步骤：   1. 思路   “截获”的过程是：修改系统调用表中调用函数的地址，将其执行我们自己实现的函数，再在我们自己的函数中完成我们想做的事情后，在返回到原来的系统调用执行流程中。  实现文件访问监控的思路为：截获sys\_open系统调用。当进程打开文件时会调用系统中断sys\_open，通过修改中断向量表使得先执行我们编写的函数my\_sys\_open，功能为打印当前访问的进程名，进程号，文件名，然后再执行sys\_open。这样就相当于监控到了信息，信息通过printk打印到系统日志，通过dmesg查看即可。   1. **代码实现**   函数asmlinkage long my\_sys\_open(char \* filename, int flags, int mode)，就是自己实现的调用函数，这里的形参是参考系统原有的open调用函数的原型。  在my\_sys\_open()中，打印了当前是哪个进程在访问（进程名和进程号的信息），访问的是哪个文件（文件的绝对路径），打印完后跳转到原来的系统调用函数。  在模块初始化的过程中，执行start\_hook()函数。在start\_hook()函数中，先获得系统调用表的地址，将系统调用表中的原有open函数的地址保存下来，再将my\_sys\_open()函数的地址赋到系统调用表中。  修改系统调用表时，由于内核中的很多东西，比如系统调用表sys\_call\_table是只读的，需要修改一下权限才能修改。由于控制寄存器CR0的第16位若置位1，则表示禁止系统进程写只有只读权限的文件，所以在修改系统调用表sys\_call\_table之前先将CR0的第16位清零，在修改完后再恢复置位。  代码的close\_cr()函数，是将CR0第16位清零，open\_cr()函数是将CR0第16位恢复。  最后在卸载modu模块的时候，将系统调用表的内容还原。  **细节见注释**  **#include<linux/init.h>**  **#include<linux/module.h>**  **#include<linux/moduleparam.h>**  **#include<linux/unistd.h>**  **#include<linux/sched.h>**  **#include<linux/syscalls.h>**  **#include<linux/string.h>**  **#include<linux/fs.h>**  **#include<linux/fdtable.h>**  **#include<linux/uaccess.h>**  **#include<linux/rtc.h>**  **MODULE\_LICENSE("Dual BSD/GPL");**    **#define \_DEBUG**  **#ifdef \_DEBUG**  **#define kprintk(fmt,args...) printk(KERN\_ALERT fmt,##args)**  **#define kprintf(fmt,args...) printf(fmt,##args)**  **#define kperror(str) perror(str)**  **#else**  **#define kprintk**  **#define kprintf**  **#define kperror**  **#endif**  **/\*Function declaration\*/**  **long \* get\_sys\_call\_table(void);**  **unsigned int close\_cr(void);**  **void open\_cr(unsigned int oldval);**  **void start\_hook(void);**  **asmlinkage long (\*orig\_open)(char \_\_user \*filename, int flags, int mode);**  **long \* g\_sys\_call\_table = NULL; //save address of sys\_call\_table**  **long g\_old\_sys\_open = 0; //save old address of sys\_open**  **long g\_oldcr0 = 0; //save address of cr0**  **struct \_idtr{ //中断描述符表寄存器**  **unsigned short limit;**  **unsigned int base;**  **}\_\_attribute\_\_((packed));**  **struct \_idt\_descriptor{**  **unsigned short offset\_low;**  **unsigned short sel;**  **unsigned char none,flags;**  **unsigned short offset\_high;**  **}\_\_attribute\_\_((packed));**  **unsigned int close\_cr(void){**  **unsigned int cr0 = 0;**  **unsigned int ret;**  **asm volatile("movl %%cr0,%%eax":"=a"(cr0));**  **ret = cr0;**  **cr0 &= 0xfffeffff;**  **asm volatile("movl %%eax,%%cr0"::"a"(cr0));**  **return ret;**  **}**  **void open\_cr(unsigned int oldval){**  **asm volatile("movl %%eax,%%cr0"::"a"(oldval));**  **}**  **/\*Get the address of sys\_call\_table\*/**  **long \* get\_sys\_call\_table(void){ //在idtr寄存器**    **struct \_idt\_descriptor \* idt;**  **struct \_idtr idtr;**  **unsigned int sys\_call\_off;**  **int sys\_call\_table=0;**  **unsigned char\* p;**  **int i;**  **asm("sidt %0":"=m"(idtr)); //汇编，sidt指令获得中断描述符表基地址**  **kprintk(" address of idtr: 0x%x\n",(unsigned int)&idtr);**  **idt=(struct \_idt\_descriptor \*)(idtr.base+8\*0x80); //0x80中断为系统调用中断 这是一个描述符，下面的操作得到描述符指向的具体地址**  **sys\_call\_off=((unsigned int)(idt->offset\_high<<16)|(unsigned int)idt->offset\_low);**  **kprintk(" address of idt 0x80: 0x%x\n",sys\_call\_off); //0x80位**  **p=(unsigned char \*)sys\_call\_off;**  **for(i=0;i<100;i++){**  **if(p[i]==0xff&&p[i+1]==0x14&&p[i+2]==0x85){**  **sys\_call\_table=\*(int\*)((int)p+i+3);**  **kprintk(" address of sys\_call\_table: 0x%x\n",sys\_call\_table);**  **return (long\*)sys\_call\_table;**  **}**  **}**    **return 0;**  **}**  **//My own sys\_open**  **asmlinkage long my\_sys\_open(char \* filename, int flags, int mode){ //打印当前使用sys-open函数的进程信息和文件名**  **kprintk("The process is \"%s\"(pid is %i)\n",current->comm,current->pid);**  **kprintk("The file is being accessed is \"%s\"\n",filename);**  **return orig\_open(filename,flags,mode);**  **}**  **void start\_hook(void){ //得到系统调用表地址，寻找sys-open项，替换为my\_sys-open,cr0寄存器16位置0，可以写只读项**  **g\_sys\_call\_table = get\_sys\_call\_table();**  **if(!g\_sys\_call\_table){**  **kprintk("Get sys\_call\_table error!\n");**  **return;**  **}**  **if(g\_sys\_call\_table[\_\_NR\_close] != (unsigned long)sys\_close){**  **kprintk("Incorrect sys\_call\_table address!\n");**  **return;**  **}**    **g\_old\_sys\_open = g\_sys\_call\_table[\_\_NR\_open];**  **orig\_open = (long(\*)(char \*, int, int))g\_sys\_call\_table[\_\_NR\_open];**    **g\_oldcr0=close\_cr();**  **g\_sys\_call\_table[\_\_NR\_open] = my\_sys\_open;**  **open\_cr(g\_oldcr0);**  **}**  **int monitor\_init(void){ //启动模块**  **kprintk("Monitor init\n");**  **start\_hook();**  **return 0;**  **}**  **void monitor\_exit(void){ //退出模块 恢复系统调用表sys\_open函数所在项地址为原sys\_open，cr0寄存器16位置1，禁止写只读文件**  **if(g\_sys\_call\_table && g\_old\_sys\_open){**  **g\_oldcr0 = close\_cr();**  **g\_sys\_call\_table[\_\_NR\_open] = g\_old\_sys\_open;**  **open\_cr(g\_oldcr0);**  **}**  **kprintk("Monitor exit\n");**  **}**  **module\_init(monitor\_init);**  **module\_exit(monitor\_exit);**   1. **Makefile编写**   获取当前路径和系统内核的路径，然后编译     1. **加载内核模块，卸载内核模块**   通过insmod XX.ko加载模块，rmmod xx.ko卸载模块。  cat /proc/modules查看模块信息，lsmod查看所有的模块。   1. **运行效果**   见实验结果记录 |
| 1. 实验过程分析   （实验分工，详细记录实验过程中发生的故障和问题，进行故障分析，说明故障排除的过程及方法。根据具体实验，记录、整理相应的数据表格等） |
| **遇到的问题及解决：**   1. **编写完makefile 执行Make时报错 nothing to be done for all**   原因：这是因为空格和tab的转换问题  比如下面两个make就不一样。    解决：删掉前面的空格，改成tab   1. **insmod后ubuntu系统卡死**   原因:VMware的问题，换成virtual box就可以正常运行。  **实验结果记录：**  **insmod 后dmesg，查看系统日志，可见记录了一些访问文件的进程名，id和文件名，于是实现了文件监控**    **编译的文件** |