

Linux系统编程实践

实验报告

|  |  |
| --- | --- |
| 实验名称 | Linux系统监测程序-API监控 |
| 学生姓名 | 王羽昕 |
| 学 号 | 8203201225 |
| 专业班级 | 信息安全2002班 |
| 指导教师 | 胡小龙 |
| 学 院 | 计算机学院 |
| 完成时间 | 2022.1.2 |

**目录**

[一 实验目的 3](#_Toc124519673)

[二 实验内容 3](#_Toc124519674)

[三 实验原理 4](#_Toc124519675)

[四 实验步骤 8](#_Toc124519676)

[（1）编写系统调用程序dome.c 8](#_Toc124519677)

[（2）编写内核监视模块syscall.c 8](#_Toc124519678)

[（3）编写Makefile 12](#_Toc124519679)

[（4）编写弹窗shell脚本 12](#_Toc124519680)

[（5）编译并将syscall.ko载入内核 13](#_Toc124519681)

[（6）运行弹窗c.sh脚本 15](#_Toc124519682)

[（7）编译并运行系统调用程序 15](#_Toc124519683)

[（8）查看进程PID记录——xxx文件 16](#_Toc124519684)

[（9）查看内核监视模块的记录 16](#_Toc124519685)

[五 实验总结与心得 17](#_Toc124519686)

一 实验目的

Linux作为一个开源计算机操作系统，广泛应用于服务器、桌面及嵌入式应用领域。本实验旨在锻炼实验者Linux操作系统下的内核级、系统级编程能力，主要包括在Linux操作系统下高级程序设计以及Linux内核程序设计。

**知识：**学习x86体系结构及系统初始化过程，将操作系统基本原理与Linux系统内核的进程管理、存储器管理、设备管理及文件系统的具体实现相结合；了解Linux高级程序设计方法、设计流程、开发环境等；了解Linux字符设备和块设备驱动程序设计方法；了解Linux高级程序设计尤其与系统安全相关API，学习Linux安全相关特性。

**能力：**通过对Linux内核的深入剖析，加深对操作系统原理的理解，并培养对Linux操作系统的系统级分析能力；初步掌握Linux内核模块编程、Linux简单字符设备和块设备编程能力；培养Linux系统安全相关系统设计及程序开发能力；培养能够进行Linux系统级大型应用设计开发能力。

**素质：**通过对Linux操作系统具体实现方法的整体分析，建立系统级应用开发概念，体会系统级程序设计及系统安全相关设计的复杂性、关联性、整体性；提高基本理论与具体实现相结合的综合素质。

二 实验内容

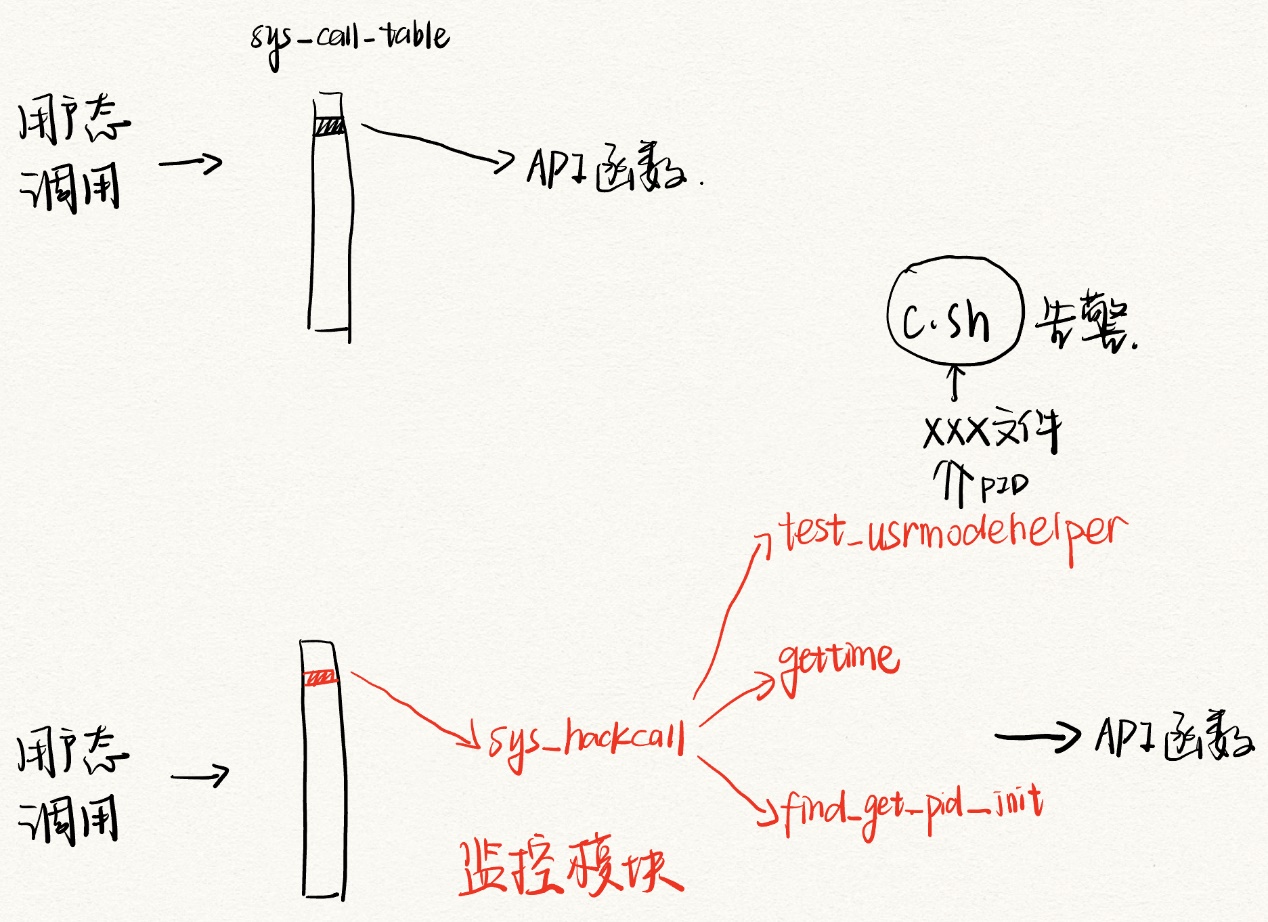
理解Linux API调用过程，掌握定制Linux API相关方法，并从安全角度监控Linux API被进程调用情况。

**实验任务：**

（1）编写一个Linux系统调用，或在通用Linux系统调用中增加相关监测机制；

（2）当进程调用该被监测API时，系统对调用情况进行记录（调用进程、调用时间等），并进行报警处理。

三 实验原理



上图是我们总体的思路设计图。

接下来介绍一下linux内核系统调用API函数的原理，以加深理解。

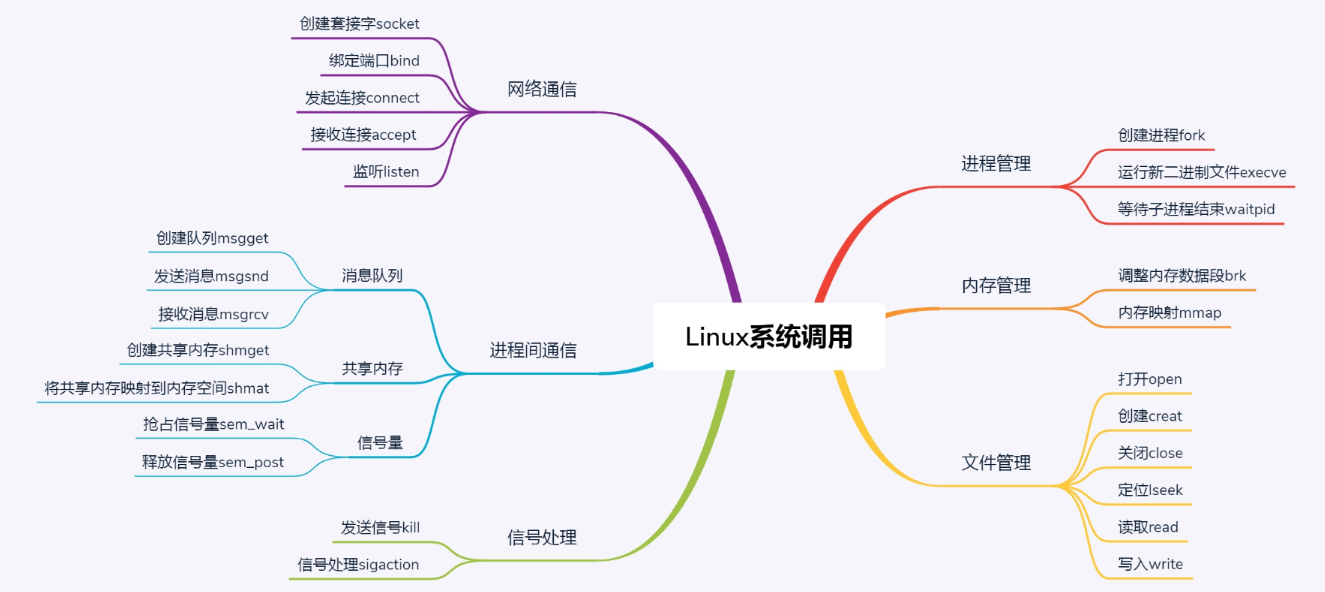
* **系统调用**

系统调用指操作系统提供给用户程序调用的一组“特殊”接口，用户程序可以通过这组“特殊”接口来获得操作系统内核提供的服务。

为什么用户程序不能直接访问系统内核提供的服务呢？这是由于在Linux中，为了更好地保护内核空间，将程序的运行空间分为用户空间和内核空间（也就是常称的用户态和内核态），它们分别运行在不同的级别上，逻辑上是相互分离的。因此，用户进程通常情况下不允许访问内核数据，也无法使用内核函数，它们只能在用户空间操作用户数据，调用用户空间的函数。

但是，在有些情况下，用户空间的进程需要获得一定的系统服务（调用内核空间程序），这时操作系统就必须利用系统提供用户的“特殊接口”———系统调用规定用户进程进入内核空间的具体位置。在进行系统调用时，程序运行空间需要从用户空间进入内核空间，处理完成后再返回用户空间。

Linux系统调用还算精简（只有250个左右），它继承了UNIX系统调用中最基本和最有用的部分。这些系统调用按照功能逻辑大致可分为进程控制、进程间通信、文件系统控制、存储管理、网络管理、套接字控制、用户管理等几类。



一般情况下应用程序通过应用编程接口API，而不是直接通过系统调用来编程。系统调用在用户空间进程和硬件设备之间添加了一个中间层。该层主要作用有三个：

* 它为用户空间提供了一种统一的硬件的抽象接口。
* 系统调用保证了系统的稳定和安全。
* 每个进程都运行在虚拟系统中，而在用户空间和系统的其余部分提供这样一层公共接口，也是出于这种考虑。如果应用程序可以随意访问硬件而内核又对此一无所知的话，几乎就没法实现多任务和虚拟内存，当然也不可能实现良好的稳定性和安全性。
* **API**

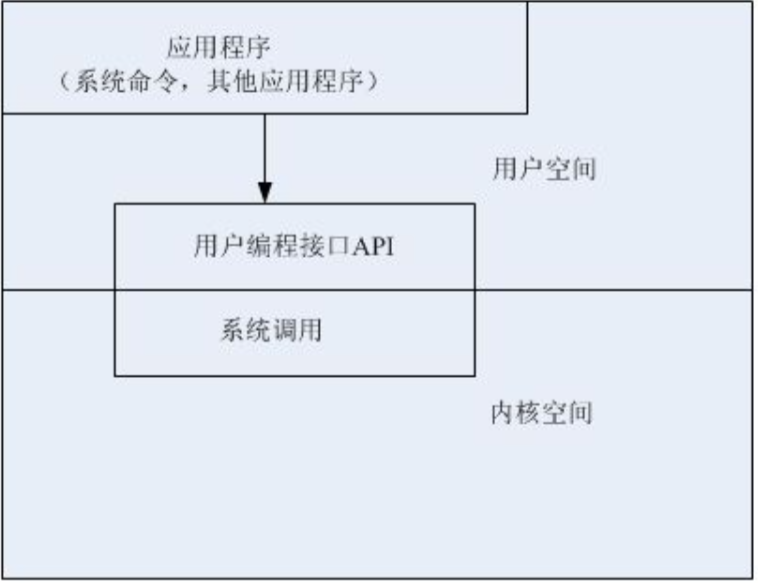
系统调用并不直接与程序员进行交互，它仅仅是一个通过软中断机制向内核提交请求以获取内核服务的接口。实际使用中，程序员调用的通常都是用户编程接口（API）。API（Application Programming Interface,应用程序编程接口），指的是我们用户程序编程调用的如read(),write(),malloc(),free()之类的，调用的是glibc库提供的库函数。

例如，创建进程的API函数fork()函数对应于内核空间的sys\_fork()系统调用，但并不是所有的函数都对应一个系统调用。有时，一个API函数会需要几个系统调用来共同完成函数的功能，甚至还有一些API函数不需要调用相应的系统调用（因此它所完成的不是内核提供的服务）。

API直接提供给用户编程使用，运行在用户态。POSIX针对API提出标准，即针对API的函数名，返回值，参数类型进行规范约束，但是并不管API具体如何实现。

* **系统命令**

系统命令相对于API更高了一层，它实际上是一个可执行程序，它的内部引用了用户编程接口（API）来实现相应的功能，它们之间的关系如下图



* **API调用与系统调用区别与联系**

区别：api是函数的定义，规定了这个函数的功能，跟内核无直接关系。而系统调用是通过中断向内核发请求，实现内核提供的某些服务。

联系：程序员调用的是API（API函数），然后通过与系统调用共同完成函数的功能。因此，API是一个提供给应用程序的接口，一组函数，是与程序员进行直接交互的。系统调用则不与程序员进行交互，它是根据API函数，通过一个软中断机制向内核提交请求，以获取内核服务的接口。

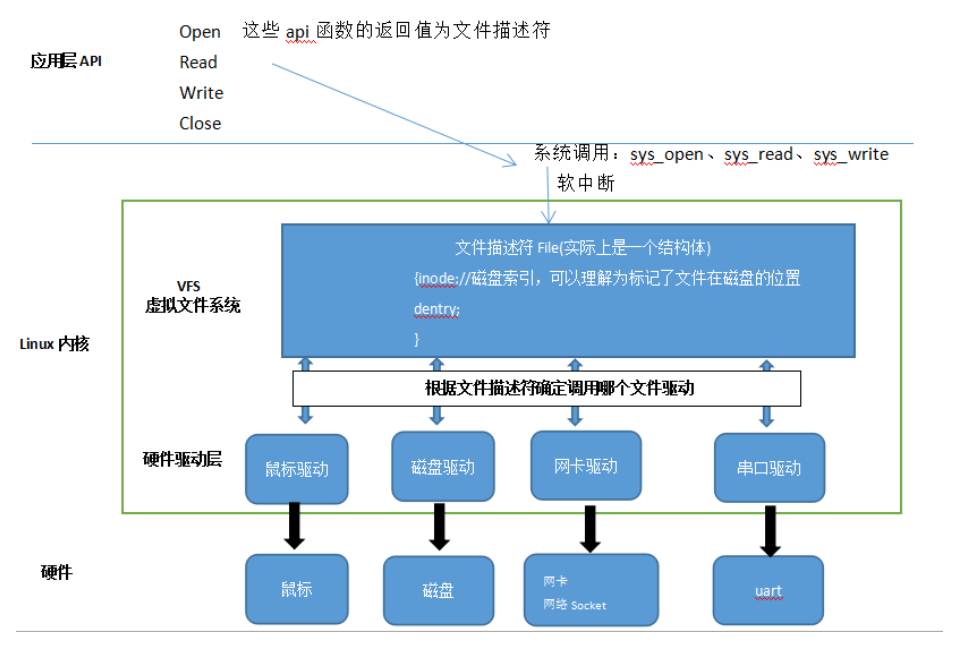
有时候，某些API所提供的功能会涉及到与内核空间进行交互。那么，这类API内部会封装系统调用。而不涉及与内核进行交互的API则不会封装系统调用。也就是说，API和系统调用并没有严格的一一对应关系，一个API可能恰好只对应一个系统调用，比如read()系统调用和read()；一个API也可能由多个系统调用实现；有时候，一个API的功能可能并不需要内核提供的服务，那么此时这个API也就不需要任何的系统调用，比如abs()。另外，一个系统调用可能还被多个API内部调用。

对于编程者来说，系统调用和API都是一组函数，并无什么两样，二者关注的都是函数名、参数类型及返回值的含义；但是事实上，系统调用的实现是在内核完成的，API则是在函数库中实现的。

例如：在内核中实现了write系统调用。在库函数中通过宏定义的方式实现write API。可以在unistd.h中看到write到系统调用的转换。

* **API调用流程**

调用过程如图所示：



以应用层调用open()函数打开鼠标文件时的调用过程为例：

当用户通过API中的open()函数试图去打开一个鼠标文件时，open()函数会有一个返回值，这个返回值即为文件描述符，文件描述符被定义为file结构体，file结构体关联到硬件驱动，然后硬件驱动去关联到相应的硬件设备。

再具体一点说，API的open()函数会通过系统调用，而系统调用进入到系统内核即VFS。然后根据文件描述符File来对应硬件驱动。文件描述符实际上是一个结构体，里面是一些函数指针，会提前注册到内核，linux驱动和windows驱动的模型不同，根据不同的模型进行不同的函数封装，当需要调用哪个就是通过回调进行调用驱动。

其中File结构体中有一个inode成员，inode是磁盘索引，可以理解为标记了文件在磁盘的位置。

四 实验步骤

（1）编写系统调用程序dome.c

#include <syscall.h>

#include <stdio.h>

#include <unistd.h>

int main(void)

{

unsigned long ret = syscall(\_\_NR\_perf\_event\_open, NULL, 0, 0, 0, 0);

sleep(5);

printf("%d\n", (int)ret);

return 0;

}

本实验中调用的是\_\_NR\_perf\_event\_open这一系统函数。

（2）编写内核监视模块syscall.c

//注册模块用到的一个结构体

static struct kprobe kp = {

.symbol\_name = "kallsyms\_lookup\_name"

};

// 将进程信息输出到终端xxx文件，与shell脚本配合进行告警

static int test\_usrmodehelper(void){

int result = 0;

//定义从内核函数获取进程文件地址的参数

char buf\_1[TASK\_PATH\_MAX\_LENGTH] = {0};

char \*task\_path = NULL;

task\_path = dentry\_path\_raw(current->mm->exe\_file->f\_path.dentry, buf\_1, TASK\_PATH\_MAX\_LENGTH); //拿进程执行的文件路径放到buf\_1

char shell[52];

sprintf(shell,"echo %d >> /home/demo/桌面/xxx",current->pid);//把pid替换%d，再把字符串放到shell

char cmdPath[] = "/bin/bash";//命令路径

char\* cmdArgv[] = {cmdPath, "-c", shell, NULL};//命令参数

char\* cmdEnvp[] = {"HOME=/", "PATH=/sbin:/bin:/usr/bin", NULL};//命令环境

result = call\_usermodehelper(cmdPath, cmdArgv, cmdEnvp, UMH\_WAIT\_PROC);//拼起来执行一下；调用callusermodehelper执行shell命令

printk("=========================================================");

printk(KERN\_DEBUG"shell is: %s",shell);

printk(KERN\_DEBUG"test\_usrmodehelper exec! The process is \"%s\",pid is %d\n, pidtask\_file\_com is \"%s\"",current->comm,current->pid,task\_path);

return 0;

}

/\*

\* 获取系统时间

\*/

static int gettime(void){

ktime\_t time = 0;

time = ktime\_get\_real(); // 获取到的是纳秒级

printk("time : %lld s \n", time/1000000000);

return 0;

}

/\*\*

\* 打印进程调用api函数返回进程pid信息

\*/

static int find\_get\_pid\_init(void)

{

printk("into find\_get\_pid\_init.\n");

struct pid \* kpid=find\_get\_pid(current->pid); //是个系统函数: 根据进程号，调用函数获取进程描述符信息

printk("the count of the pid is :%d\n", kpid->count);//显示进程描述符信息

printk("the level of the pid is :%d\n", kpid->level);

// 显示进程号

test\_usrmodehelper();

printk("the pid of the find\_get\_pid is :%d\n", kpid->numbers[kpid->level].nr);

printk("out find\_get\_pid\_init.\n");

return 0;

}

/\*\*

\* 设置cr0寄存器的第16位为0，如果为1就不能改sys\_call\_table

\*/

unsigned int clear\_and\_return\_cr0(void){

unsigned int cr0 = 0;

unsigned int ret;

/\* 将cr0寄存器的值移动到rax寄存器中，同时输出到cr0变量中 \*/

asm volatile ("movq %%cr0, %%rax" : "=a"(cr0));

//带有C/C++表达式的内联汇编格式为：

//asm volatile("InSTructiON List" : Output: Input: Clobber / Modify);

ret = cr0;

cr0 &= 0xfffeffff; /\* 将cr0变量值的第16位清0,将修改后的值写入到cr0寄存器 \*/

/\* 读取cr0的值到人rax寄存器，再将rax寄存器的值放入cr0中 \*/

asm volatile ("movq %%rax, %%cr0" :: "a"(cr0));

return ret;

}

/\*\*,

\* 还原cr0寄存器的值为val，卸载模块用到

\*/

void setback\_cr0(unsigned int val){

asm volatile ("movq %%rax, %%cr0" :: "a"(val));

}

/\*\*

\* 自己编写的系统调用函数

\*/

static long sys\_hackcall(struct perf\_event\_attr \*attr, pid\_t pid, int cpu, int group\_fd,unsigned long flags){

/\* 打印当前调用该API的PID和当前时间戳 \*/

find\_get\_pid\_init();

gettime();

printk("capted is successful!!!\n");

// 调用原来的API函数

int ret = orig\_syscall(attr, pid, cpu, group\_fd, flags);

printk("----------------------------------------------------------");

printk("orig is running!!!");

return ret;

}

/\*\*

\* 模块的初始化函数，模块的入口函数，加载模块时调用

\*/

static int \_\_init init\_hack\_module(void){

int orig\_cr0;

printk("Looking is starting...\n");

typedef unsigned long (\*kallsyms\_lookup\_name\_t)(const char \*name);

kallsyms\_lookup\_name\_t kallsyms\_lookup\_name;

register\_kprobe(&kp);

kallsyms\_lookup\_name = (kallsyms\_lookup\_name\_t)kp.addr;

unregister\_kprobe(&kp);

/\*\*

\* 获取sys\_call\_table 虚拟内存地址

\*/

sys\_call\_table = (unsigned long \*)kallsyms\_lookup\_name("sys\_call\_table");

/\* 保存原始系统调用 \*/

// orig\_syscall\_saved = (int(\*)(void))(sys\_call\_table[\_\_NR\_perf\_event\_open]);

orig\_syscall\_saved = sys\_call\_table[\_\_NR\_perf\_event\_open];

orig\_syscall = (long(\*)(struct perf\_event\_attr \*, pid\_t , int , int ,unsigned long))sys\_call\_table[\_\_NR\_perf\_event\_open];

orig\_cr0 = clear\_and\_return\_cr0(); /\* 设置cr0寄存器的第16位为0 \*/

sys\_call\_table[\_\_NR\_perf\_event\_open] = (unsigned long)&sys\_hackcall; /\* 替换成我们编写的函数 \*/

setback\_cr0(orig\_cr0); /\* 还原cr0寄存器的值 \*/

return 0;

}

/\*\*

\* 模块退出函数，卸载模块时调用

\*/

static void \_\_exit exit\_hack\_module(void){

int orig\_cr0;

orig\_cr0 = clear\_and\_return\_cr0();

sys\_call\_table[\_\_NR\_perf\_event\_open] = orig\_syscall\_saved; /\* 设置为原来的系统调用 \*/

setback\_cr0(orig\_cr0);

printk("Looking is exited...\n");

}

module\_init(init\_hack\_module);

module\_exit(exit\_hack\_module);

MODULE\_LICENSE("GPL");

（3）编写Makefile

obj-m:=syscall.o

PWD:= $(shell pwd)

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

EXTRA\_CFLAGS= -O0

all:

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

clean:

make -C $(KERNELDIR) M=$(PWD) clean

（4）编写弹窗shell脚本

#!/bin/bash

# ToDo:此shell脚本为了显示弹窗告警！！

b=1;

while true :

do

# 获取文件内容的行数

a=$(awk '{print NR}' ./桌面/xxx | tail -n 1);

#echo $a;

if ((a > b));then

# 获取文件最后一行的内容

pid=$(sed -n '$p' ./桌面/xxx);

echo $pid;

pidinfo=$(ps axhw -o user,pid,cmd | grep -v grep|grep $pid);

echo "$(date "+%Y-%m-%d-%H:%M:%S") $pidinfo";

/usr/bin/notify-send -i face-smile 'Wa:\_\_NR\_perf\_event\_open Running hading '$a' times!!!' "The one time is : $(date "+%Y-%m-%d-%H:%M:%S !!!!")\\nit is: $pidinfo";

#zenity 调用的弹窗会线程中止，可能会漏掉通知！！，演示时弹出后及时点掉再继续运行程序问题不大!

zenity --warning --title=Warning:Pert\_Event\_Open\_is\_Running --text="\_\_NR\_perf\_event\_open\_API is Running Now $(date)\\n Info: $pidinfo" --width=1 --timeout=60 --modal;

b=$a;

echo "info:$b";

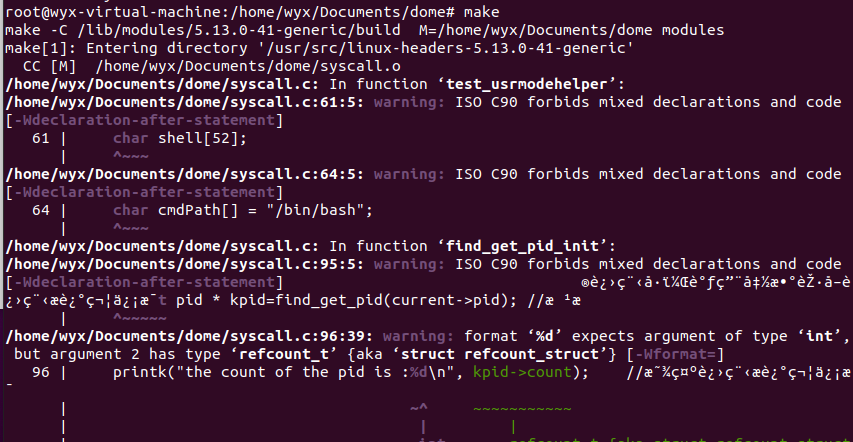
fi

#echo "over!"

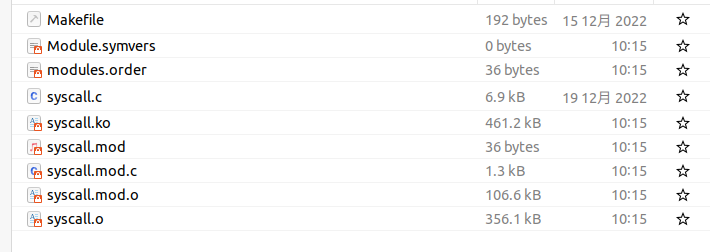
done

（5）编译并将syscall.ko载入内核

Make命令编译：



编译生成的文件结构：

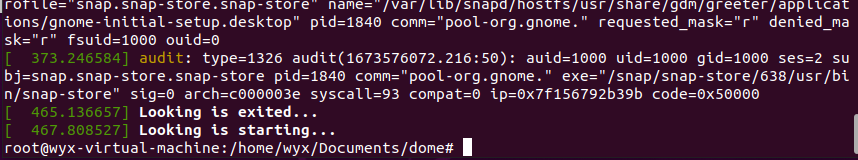


载入syscall.ko内核：

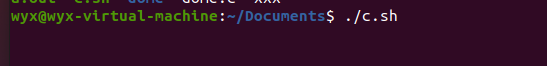


查看系统日志，监视模块是否生效：

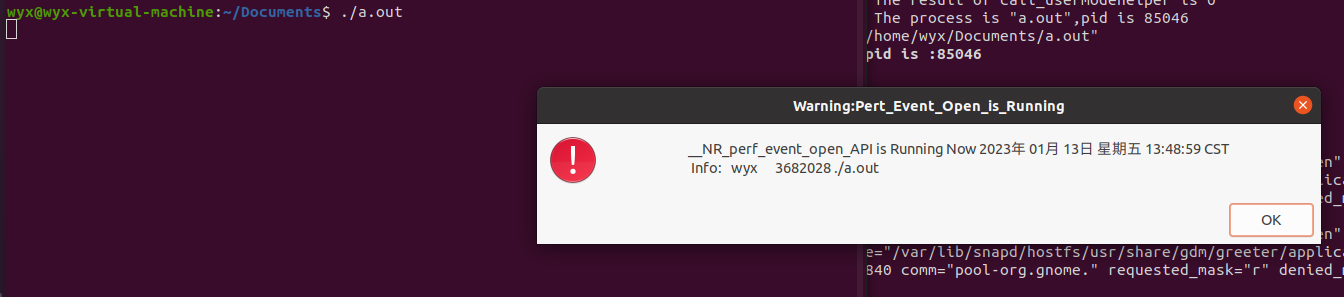
**命令：dmesg**



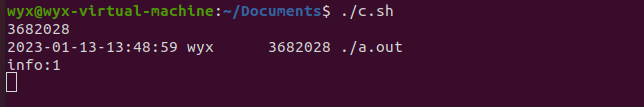
（6）运行弹窗c.sh脚本



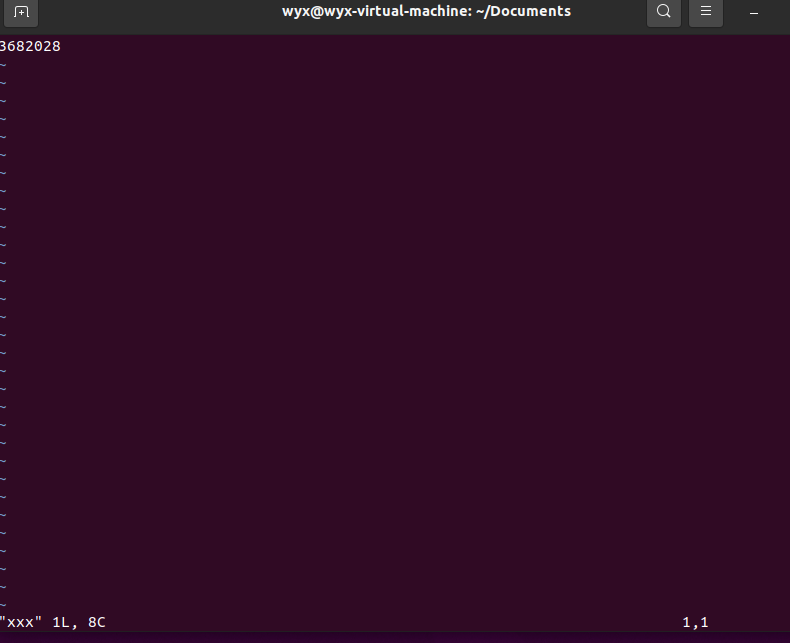
（7）编译并运行系统调用程序



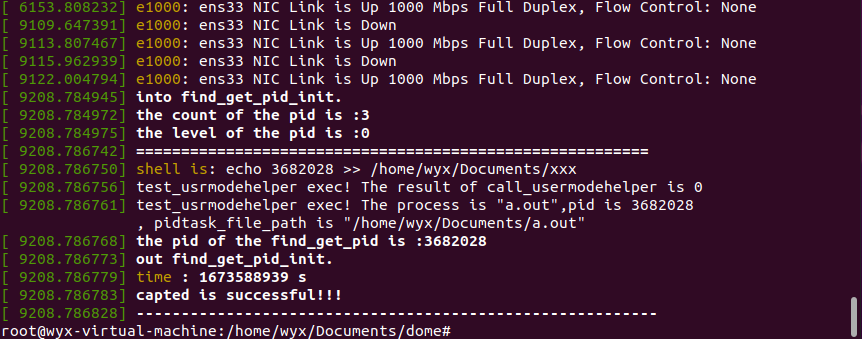
有告警弹窗显示。



（8）查看进程PID记录——xxx文件



（9）查看内核监视模块的记录



五 实验总结与心得

内核是操作系统的关键组件。它借助进程间通信和系统调用，在硬件级别上充当应用程序和数据处理之间的桥梁。每当将操作系统加载到内存中时，首先，将加载内核并将其保留在那里，直到操作系统关闭。内核负责处理低级任务，例如任务管理，内存管理，风险管理等。

而API是一个提供给应用程序的接口，一组函数，是与程序员进行直接交互的。系统调用则不与程序员进行交互的，它根据API函数，通过一个软中断机制向内核提交请求，以获取内核服务的接口。并不是所有的API函数都一一对应一个系统调用，有时，一个API函数会需要几个系统调用来共同完成函数的功能，甚至还有一些API函数不需要调用相应的系统调用（因此它所完成的不是内核提供的服务）。

因为内核与系统调用联系紧密，而API接口参与了系统调用，为了内核的安全防止内核被恶意入侵，内核监控Linux API被进程调用情况是十分有必要的，这在恶意代码入侵检测中起到重要的作用。因此我们必须格外关注API函数的调用安全，因为攻击者很有可能从API下手，而对内核进行监控有利于及时发现风险，从而防护免于被攻击。