中 央 民 族 大 学

**操作系统**

**课程设计指导书**

教师姓名：

职 称：

中央民族大学信息工程学院

2020年9月-12月

**目 录**

实验报告格式

实验一 多线程同步实验（难度\*\*）

实验二 添加内核模块（难度\*）

实验三 基于proc系统打印进程树（难度\*\*\*\*）

实验四 遍历内核进程控制块打印进程树（难度\*\*\*\*）

实验五 添加系统调用（难度\*\*\*）

实验六 设备驱动程序（难度\*\*\*\*\*）

实验七 虚拟文件系统（难度\*\*\*\*\*）

**实验报告格式**

**实验一 \*\*\*\*\*\***

1. **实验目的：**

\*\*\*\*

**二、实验原理：**

填写实验基础知识等

1. **实验程序：**

程序代码

1. **实验截图及结果分析:**

\*\*\*\*\*

1. **实验心得：**

\*\*\*\*\*简写

1. **实验日期：**

2020年\*月\*日

1. **报告书写人：**

XXX

**实验一 多线程同步实验**

一、实验目的

1、掌握线程的概念及创建方法

2、掌握线程同步机制

3、掌握生产者消费者原理

4、设计程序，实现多线程生产者消费者同步问题

(备注：至少包括两个生产者线程，两个消费者线程。生产者线程放的数据需要不同，例如：生产者1放1、2、3、4，生产者2放-1，-2，-3，或者3、6、9、12…。备注不写入实验报告)

二、实验基本原理

1.线程的概念

线程（英语：thread）是[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F" \t "_blank)能够进行运算[调度](https://baike.baidu.com/item/%E8%B0%83%E5%BA%A6)的最小单位。它被包含在[进程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B)之中，是[进程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B)中的实际运作单位。一个线程指的是[进程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B)中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

线程是独立调度和分派的基本单位。同一进程中的多个线程将共享该进程中的全部系统资源，如虚拟地址空间，[文件描述符](https://baike.baidu.com/item/%E6%96%87%E4%BB%B6%E6%8F%8F%E8%BF%B0%E7%AC%A6)和[信号处理](https://baike.baidu.com/item/%E4%BF%A1%E5%8F%B7%E5%A4%84%E7%90%86)等等。但同一进程中的多个线程有各自的[调用栈](https://baike.baidu.com/item/%E8%B0%83%E7%94%A8%E6%A0%88)（call stack），自己的寄存器环境（register context），自己的线程本地存储（thread-local storage）。

一个进程可以有很多线程，每个线程并行执行不同的任务。在多核或多CPU，或支持Hyper-threading的CPU上使用多线程程序设计的好处是显而易见，即提高了程序的执行吞吐率。在单CPU单核的计算机上，使用多线程技术，也可以把进程中负责I/O处理、人机交互而常被阻塞的部分与密集计算的部分分开来执行，编写专门的线程执行密集计算，从而提高了程序的执行效率。

2.线程的相关函数

pthread.h头文件中包含的几个基本概念及函数：

(1)pthread\_t：线程ID数据类型，线程ID只在它所属的进程环境中有效。

(2) int pthread\_create(pthread\_t \*tidp, const pthread\_attr\_t \*attr, func(void), arg)

创建新的线程，若成功返回0。

参数1：是一个传出参数，用于保存成功创建线程之后对应的线程id。

参数2：表示线程的属性，通常默认传NULL，如果想使用具体的属性也可以修改具体的参数。

参数3：函数指针，一个指向函数的指针。指向创建线程所执行函数的入口地址，函数执行完毕，则线程结束。

参数4：线程主函数执行期间所使用的参数。

(3) pthread\_t pthread\_self(void)：获取自身线程ID。

(4) void pthread\_exit(void \*rval\_ptr)：终止本线程，用rval\_ptr指向的值作为退出码。

(5) #include <pthread.h>  
int pthread\_join(pthread\_t thread, void \*\* rval\_ptr);

pthread\_t thread: //回收线程的tid  
void \*\* rval\_ptr: //接收退出线程传递出的返回值  
返回值:成功返回0,失败返回错误号

调用该函数的线程将阻塞，直到thread线程调用pthread\_exit、从启动例程返回或被取消，rval\_ptr将包含返回码。

注意：

调用该函数的线程将挂起等待,直到id为thread的线程终止。thread线程以不同的方法终止,通过pthread\_join得到的终止状态是不同的,总结如下:  
如果thread线程通过return返回, rval\_ptr所指向的单元里存放的是thread线程函数的返回值。  
如果thread线程被别的线程调用pthread\_cancel异常终止掉, rval\_ptr所指向的单元里存放的是常数PTHREAD\_CANCELED。  
如果thread线程是自己调用pthread\_exit终止的,retval所指向的单元存放的是传给pthread\_exit的参数。  
如果对thread线程的终止状态不感兴趣,可以传NULL给rval\_ptr参数。

(6) int pthread\_cancel(pthread\_t tid)：该函数用来请求取消统一进程中的其他线程。

3.线程同步————互斥锁，条件变量

1）互斥锁、加锁、解锁

#include <pthread.h>

1. pthread\_mutex\_t \*mutex 定义互斥锁
2. int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

在成功完成之后会返回零。其他任何返回值都表示出现了错误。

[线程](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/103101)调用该函数让互斥锁上锁，如果该互斥锁已被另一个线程锁定和拥有，则调用该线程将阻塞，直到该互斥锁变为可用为止。

1. int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

[pthread\_mutex\_unlock是](https://baike.baidu.com/item/pthread_mutex_unlock%E6%98%AF)可以解除锁定 mutex 所指向的互斥锁的函数。pthread\_mutex\_unlock()可释放mutex引用的互斥锁对象。pthread\_mutex\_unlock()在成功完成之后会返回零。其他任何返回值都表示出现了错误。

#include <pthread.h>

pthread\_mutex\_t mutex;

int ret;

ret = pthread\_mutex\_unlock(&mutex);

2）条件变量

条件变量是利用线程间共享的[全局变量](https://baike.baidu.com/item/%E5%85%A8%E5%B1%80%E5%8F%98%E9%87%8F)进行同步的一种机制，主要包括两个动作：一个线程等待"条件变量的条件成立"而挂起；另一个线程使"条件成立"（给出条件成立信号）。为了防止竞争，条件变量的使用总是和一个[互斥锁](https://baike.baidu.com/item/%E4%BA%92%E6%96%A5%E9%94%81)结合在一起。

条件变量简介

在服务器编程中常用的线程池，多个线程会操作同一个任务队列，一旦发现任务队列中有新的任务，子线程将取出任务；这里因为是多线程操作，必然会涉及到用互斥锁保护任务队列的情况（否则其中一个线程操作了任务队列，取出线程到一半时，线程切换又取出相同任务）。但是互斥锁一个明显的缺点是它只有两种状态：锁定和非锁定。设想，每个线程为了获取新的任务不断的进行这样的操作：锁定任务队列，检查任务队列是否有新的任务，取得新的任务（有新的任务）或不做任何操作（无新的任务），释放锁，这将是很消耗资源的。

而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，它常和互斥锁一起配合使用。使用时，条件变量被用来阻塞一个线程，当条件不满足时，线程往往解开相应的互斥锁并等待条件发生变化。一旦其他的某个线程改变了条件变量，它将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。这些线程将重新锁定互斥锁并重新测试条件是否满足。一般说来，条件变量被用来进行线程间的同步。

对应于线程池的场景，我们可以让线程处于等待状态，当主线程将新的任务放入工作队列时，发出通知（其中一个或多个），得到通知的线程重新获得锁，取得任务，执行相关操作。

条件变量类型为 pthread\_cond\_t。

a) 条件变量和[互斥锁](https://baike.baidu.com/item/%E4%BA%92%E6%96%A5%E9%94%81" \t "_blank)一样，都有静态和动态两种创建方式，

静态方式使用PTHREAD\_COND\_INITIALIZER[常量](https://baike.baidu.com/item/%E5%B8%B8%E9%87%8F)进行初始化，如下：

pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;

动态方式调用[pthread\_cond\_init](https://baike.baidu.com/item/pthread_cond_init" \t "_blank)()函数，API定义如下：

int pthread\_cond\_init(pthread\_cond\_t \*cond, pthread\_condattr\_t \*cond\_attr)

b)注销一个条件变量需要调用pthread\_cond\_destroy()，只有在没有线程在该条件变量上等待的时候，才能注销这个条件变量，否则返回EBUSY。API定义如下：

int pthread\_cond\_destroy(pthread\_cond\_t \*cond)

## c) 等待条件变量pthread\_cond\_wait

#include <pthread.h>

int pthread\_cond\_wait(pthread\_cond\_t \*cond,pthread\_mutex\_t \*mutex);

成功则返回0, 出错则返回错误编号.

这个函数是阻塞等待.等待条件函数等待条件变为真,传递给pthread\_cond\_wait的互斥量对条件进行保护,调用者把锁住的互斥量传递给函数.函数把调用线程放到等待条件的线程列表上,然后对互斥量解锁,这两个操作是原子的.这样便关闭了条件检查和线程进入休眠状态等待条件改变这两个操作之间的时间通道,这样线程就不会错过条件的任何变化.当pthread\_cond\_wait返回时,互斥量再次被锁住.

## d)通知等待条件变量的单个线程pthread\_cond\_signal

#include <pthread.h>

int pthread\_cond\_signal(pthread\_cond\_t \*cond);

成功则返回0, 出错则返回错误编号.

一次唤醒一个线程，如果有多个线程调用 **pthread\_cond\_wait**等待，具体哪个线程被唤醒是不确定的（可以认为是随机的）。

pthread\_cond\_signal函数的作用是发送一个信号给另外一个正在处于阻塞等待状态的线程,使其脱离阻塞状态,继续执行。如果没有线程处在阻塞等待状态,pthread\_cond\_signal也会成功返回。

4.线程同步---Posix信号量

System V信号量和Posix信号量区别

信号量有两种实现：传统的System V信号量和新的POSIX信号量。它们所提供的函数很容易被区分：对于所有System V信号量函数，在它们的名字里面没有下划线。例如，应该是semget()而不是sem\_get()。然而，所有的POSIX信号量函数都有一个下划线。

**POSIX信号量来源于POSIX技术规范的实时扩展方案(POSIX Realtime Extension)，常用于线程；system v信号量，常用于**进程**的同步。**

这两者非常相近，但它们使用的函数调用各不相同。前一种Posix信号量的头文件为semaphore.h，函数调用为sem\_init(),sem\_wait(),sem\_post(),sem\_destory()等等。后一种System V信号量的头文件为<sys/sem.h>,函数调用为semctl(),semget(),semop()等函数。

下面列出了它们提供的所有函数清单：

|  |  |
| --- | --- |
| **Systm V** | **POSIX** |
| semctl() | sem\_getvalue() |
| semget() | sem\_post() |
| semop() | sem\_timedwait() |
|  | sem\_trywait() |
|  | sem\_wait() |
|  |  |
|  | sem\_destroy() |
|  | sem\_init() |
|  |  |
|  | sem\_close() |
|  | sem\_open() |
|  | sem\_unlink() |

另外一个区别是，对于POSIX信号量，可以有命名的信号量，例如，信号量有一个文件关联它们，对于最后三个函数，被用来创建，关闭和删除这样一个命名的信号量。而sem\_init()和sem\_destroy()仅仅供非命名信号量使用。

信号量的函数都以sem\_开头，线程中使用的基本信号量函数有4个，它们都声明在头文件semaphore.h中。

(1)sem\_init函数

该函数用于创建信号量，其原型如下：

int sem\_init(sem\_t \*sem,int pshared,unsigned int value);

该函数初始化由sem指向的信号对象，设置它的共享选项，并给它一个初始的整数值。

pshared控制信号量的类型，如果其值为0，就表示这个信号量是当前进程的局部信号量，否则信号量就可以在多个进程之间共享，value为sem的初始值。调用成功时返回0，失败返回-1.

(2)sem\_wait函数

该函数用于以原子操作的方式将信号量的值减1。原子操作就是，如果两个线程企图同时给一个信号量加1或减1，它们之间不会互相干扰。它的原型如下：

int sem\_wait(sem\_t \*sem);

sem指向的对象是由sem\_init调用初始化的信号量。调用成功时返回0，失败返回-1.

(3)sem\_post函数

该函数用于以原子操作的方式将信号量的值加1。它的原型如下：

int sem\_post(sem\_t \*sem);

与sem\_wait一样，sem指向的对象是由sem\_init调用初始化的信号量。调用成功时返回0，失败返回-1.

(4)sem\_destroy函数

该函数用于对用完的信号量的清理。它的原型如下：

int sem\_destroy(sem\_t \*sem);

成功时返回0，失败时返回-1.

5. 利用多线程实现生产者消费者同步问题

如何利用线程互斥变量机制定义缓冲区结构

如何利用线程同步通信机制协调生产者/消费者进程通信

可采用条件变量或者Posix信号量实现同步。

采用条件变量和互斥锁实现生产者消费者线程同步的提示：

生产者线程：

1. . 调用pthread\_mutex\_lock()对lock上锁，并根据以下条件判断缓冲区是否已满；

(writepos + 1) % BUFSIZE == readpos //认为存放n-1个元素为满。

//若存放n个元素为满，需要定义一个计数器

2）. 若满，调用ptread\_cond\_wait()进入阻塞，等待notfull条件变量；

3）. 写入数据并移动写指针writepos；

4）. 调用pthread\_cond\_signal()向消费者信号通过notempty条件变量;

5）. 调用pthread\_mutex\_unlock()对mutex解锁。

消费者线程：

1）. 调用pthread\_mutex\_lock()对lock上锁，并根据以下条件判断缓冲区是否为空；

writepos == readpos

2）. 若空，调用ptread\_cond\_wait()进入阻塞，等待notempty条件变量；

3）. 读取数据并移动读指针readpos；

4）. 调用pthread\_cond\_signal()向消费者信号通过notfull条件变量;

5）. 调用pthread\_mutex\_unlock()对mutex解锁。

三、 实验样例

//prodcons.c

#include<stdio.h>

#include<pthread.h>

#define BUFSIZE 6

int n=1,m=1;

struct prodcons {

int buffer[BUFSIZE];

pthread\_mutex\_t lock; //互斥LOCK

int readpos , writepos;

pthread\_cond\_t notempty; //缓冲区非空条件判断

pthread\_cond\_t notfull; //缓冲区未满条件判断

};

void init(struct prodcons \* b){

pthread\_mutex\_init(&b->lock,NULL);

pthread\_cond\_init(&b->notempty,NULL);

pthread\_cond\_init(&b->notfull,NULL);

b->readpos=0;

b->writepos=0;

}

struct prodcons buffer;

void \*producer1(void \*data){

while(1){

pthread\_mutex\_lock(&(buffer.lock));

//等待缓冲区未满

if((buffer.writepos + 1) % BUFSIZE == buffer.readpos){

//缓冲区满,生产者将被挂起,直至重新被唤醒

pthread\_cond\_wait(&(buffer.notfull), &(buffer.lock)) ;

}

//写数据,并移动指针

printf("producer1--put Number: ") ;

printf("put pos is %d, put %d\n",buffer.writepos,n);

buffer.buffer[buffer.writepos]=n;

buffer.writepos++;

if(buffer.writepos >= BUFSIZE)

buffer.writepos=0;

//设置缓冲区非空的条件变量

pthread\_cond\_signal(&(buffer.notempty));

pthread\_mutex\_unlock(&(buffer.lock));

n++;

//可打开下边屏蔽语句，观察线程结果，下面两句决定数据显示速度

//if (n %3 == 0)

// sleep(1);

}

}

void \*consumer1(void \* data){

int d;

while(1){

//d = get(&buffer);

pthread\_mutex\_lock(&(buffer.lock));

if(buffer.writepos == buffer.readpos){

//等待缓冲区非空

pthread\_cond\_wait(&(buffer.notempty), &(buffer.lock));

}

//读数据,移动读指针

data = buffer.buffer[buffer.readpos];

printf("first consumer: get pos is %d, get %d\n",buffer.readpos,data);

buffer.readpos++;

if(buffer.readpos >= BUFSIZE)

buffer.readpos=0;

//设置缓冲区未满的条件变量

pthread\_cond\_signal(&(buffer.notfull));

pthread\_mutex\_unlock(&(buffer.lock));

//可打开或关闭下面语句，观看线程执行状况

//sleep(1);

}

return NULL;

}

int main(void){

pthread\_t th\_a, th\_b;

void \*retval;

init(&buffer);

pthread\_create(&th\_a, NULL, producer1, 0);

pthread\_create(&th\_d, NULL, consumer2, 0);

pthread\_join(th\_a, &retval);

pthread\_join(th\_b, &retval);

return 0;

}

编译程序：

gcc prodcons.c -o prodcons.exe -l pthread

#备注：样例程序中，认为存放buffersize-1个元素为满，判空判满条件与循环队列相同。若存储buffersize个元素为满，须加一个计数变量。

若干个线程，可以看到同一个进程地址空间，可以共享一个变量。

**实验二 添加内核模块**

**一、实验目的**

（1）掌握内核模块基本编程技术

（2）向内核中添加一个内核模块，打印进程控制块信息，编译模块

（备注：此模块要求返回当前运行进程（CURRENT）或者始祖进程(init\_task)的进程控制块中的至少20项信息，用中文标记含义。此条备注不放入实验报告）

（3）加载、卸载模块

**二、实验原理**

现代的Linux内核是具有微内核特点的宏内核。Linux内核作为一个大程序在内核空间运行。太多的设备驱动和内核功能集成在内核中，内核过于庞大。

Linux内核引入内核模块机制。通过动态加载内核模块，使得在运行过程中扩展内核的功能。不需要的时候，卸载该内核模块。

1. 模块是Linux精心设计的一种机制，可以用来动态增加内核的功能，模块在内核空间运行。
2. Linux模块可以在内核启动过程中加载，这称为静态加载。也可以在内核运行的过程中随时加载，这称为动态加载。
3. Linux中的大多数设备驱动程序或文件系统都被编译成模块，因为它们数目繁多，体积庞大，不适合直接编译在内核中。而通过模块机制，在需要使用它们的时候，在临时加载，是最合适不过的。
4. 至少两个函数：
5. init\_module()模块加载的时候调用
6. cleanup\_module()模块被卸载前调用
7. 超级用户可以通过insmod和rmmod命令显式地(手工)将模块载入内核或从内核中将它卸载。
8. 也可以在内核需要时请求守护进程（kerneld）加载和卸载模块。当内核发现需要一个模块的时候，内核会请求内核守护进程（kerneld）试图加载合适的模块。说到这里就不能不提到内核守护进程kerneld了，它很聪明，能够主动的把您需要的modules 自动插入kernel，将没用到的 module 从kernel中清退。Kerneld由两个单独的部分构成：一部分工作于Linux的内核，负责向daemon发送请求；另一部分工作于系统的用户数据区，负责调入由内核请求指定的modules。
9. 调试内核代码时，采用模块技术，用户不必每次修改后都重新编译内核和重新引导系统。
10. sudo insmod hello.ko加载这个模块
11. 使用sudo dmesg察看内核日志。可以看到输出的信息写在日志中。
12. lsmod察看系统中所有的模块。
13. 移除模块:sudo rmmod hello.ko //.ko可以去掉

14. 内核使用kbuild构建系统配置编译，kbuild构建系统可用于编译自定义的内核模块。

编译过程首先会到内核源码目录下，读取顶层的Makefile文件，然后再编译模块源码，连接生成的内核模块后缀为.ko

15..编写内核模块的makefile

内核Makefile提供的obj-m表示对象文件（object files）编译成可加载的内核模块

Hello-1.c的Makefile文件 obj-m += hello-1.o

表明有一个模块要从目标文件 hello-1.o 建立，kbuild从该目标文件建立内核模块hello-1.ko。

1. **实验代码**

**1.一个简单模块样例**

**（a）/\*hello.c\*/**

#include <linux/module.h>

#include <linux/kernel.h>

int init\_modules()

{

printk( "The major id is ");//%d\n",result);

return 0;

}

void cleanup\_modules()

{

printk("<1>Goodbye cruel world\n");

}

module\_init(init\_modules);

module\_exit(cleanup\_modules);

/\*printk相当于printf的孪生姐妹，它们一个运行在用户态，另一个则在[内核](https://baike.baidu.com/item/%E5%86%85%E6%A0%B8/108410)态被人们所熟知。但是根据不同的[操作](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C/33052)系统也会有不一样的效果，使用printk这个[函数](https://baike.baidu.com/item/%E5%87%BD%E6%95%B0/18686609" \t "_blank)不一定会将内容显示到终端上，但是一定在内核缓冲区里，可以使用dmesg查看效果。\*/

(b) **hello.c 的 Makefile**

obj-m := hello.o

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

PWD := $(shell pwd)

default:

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

clean:

rm -f hello.ko hello.mod.c hello.mod.o hello.o

//执行make，生成hello.ko

(c) 加载卸载模块

sudo insmod hello.ko加载这个模块

使用sudo dmesg察看内核日志。可以看到输出的信息写在日志中。

lsmod察看系统中所有的模块。

移除模块:sudo rmmod hello.ko //.ko可以去掉

备注：Makefile参数的解释说明：

#makefile的命令行以[tab]键开始

#makefile的目标必须放在第一个命令行，如果有多个目标，可以使用：  
all:target1 target2 target3  
#rm -f 其中的，f参数 （f --force ） 忽略不存在的文件，不显示任何信息不会提示确认信息。

#每个Makefile中都应该写一个清空目标文件（.o和执行文件）的规则，这不仅便于重编译，也很利于保持文件的清洁。这是一个“修养”。一般的风格都是：  
clean:  
rm edit $(objects)

# -C是表示进入$(KERNELDIR)目录执行makefile，而M不是makefile的选项，是内核根目录下的Makefile中使用的变量，modules是Makefile中的目标。

#先解析`pwd`，反引号：`pwd`等同于$(pwd)，在Linux中反引号起着命令替换的作用，将反引号中的字符串做为命令来执行；

pwd命令用于显示工作目录。执行pwd指令可立刻得知您目前所在的工作目录的绝对路径名称。

$(shell pwd)：执行pwd命令

$(pwd)：表示当前所在目录

#uname 是[Linux](https://baike.baidu.com/item/Linux/27050" \t "_blank)命令，用于显示当前操作系统名称。-r 显示操作系统的发行版号。

#obj-m表示把文件hello.o作为"模块"进行编译，不会编译到内核，但是会生成一个独立的 "hello.ko" 文件；

obj-y表示把hello.o文件编译进内核。

（1）obj-m := hello.o

表明有一个模块要从目标文件 hello.o 建立. 在从目标文件建立后结果模块命名为 hello.ko。

（2）KERNELDR := /usr/src/linux-2.6.26

用来定位内核源码目录

（3）PWD := $(shell pwd)

获得当前目录路径

（4）M=$(PWD) M= 选项

使 makefile 在试图建立模块目标前, 回到你的模块源码目录

（5）$(MAKE) -C $(KERNELDR) M=$(PWD) modules

这个命令开始是改变它的目录到用 -C 选项提供的目录下( 就是说, 你的内核源码目录 ). 它在那里会发现内核的顶层 makefile. 这个 M= 选项使 makefile 在试图建立模块目标前, 回到你的模块源码目录. 这个目标, 依次地, 是指在 obj-m 变量中发现的模块列表, 在我们的例子里设成了module.o.

（6）$(MAKE) -C $(KERNELDR) M=$(PWD) modules\_install

用于模块的安装

（7）rm -rf \*.o \*~ core .depend .\*.cmd \*.ko \*.mod.c .tmp\_versions

用于make clean清除上次编译生成的文件

2.样例：编写showPCB模块并加载

**/\*showPCB.c\*/**

/\*此程序需要查看Linux源代码中的进程控制块数据结构typedef struct task\_struct，进程控制块数据结构typedef struct task\_struct所在的头文件sched.h中，可以使用find -name sched.h /usr/src 命令查找位置。位置如下：/usr/src/linux-4.15.3/include/linux/sched.h等等。可以使用gedit /usr/src/linux-4.15.3/include/linux/sched.h查看。可以使用find -name sched.h /usr/src 命令查找\*/

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/sched.h>

//#include <linux/asm-i386/current.h>

#include <linux/list.h>

/\*pid\_t pid;

struct task\_struct \*parent;

struct list\_head children;

struct list\_head sibling;

char comm[16]; \*/

typedef struct task\_struct ts;

//now \*ts;

int init\_modules()

{

ts \*now;

now=current;

printk("comm: %s\n",now->comm);//为用户标识

printk("state: %ld\n",now->pid);

printk("exit\_signal: %d\n",now->exit\_signal);

printk("pdeath\_signal: %d\n",now->pdeath\_signal);

return 0;

}

void cleanup\_modules()

{

printk("<1>Goodbye cruel world\n");

}

module\_init(init\_modules);

module\_exit(cleanup\_modules);

**/\*showPCB 的 Makefile\*/**

obj-m := showPCB.o

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

PWD := $(shell pwd)

default:

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

clean:

rm -f showPCB.ko showPCB.mod.c showPCB.mod.o showPCB.o

加载卸载模块、查看信息同上。

**实验三 基于proc系统打印进程树**

一、实验目的：

1. 掌握process系统
2. 访问/proc目录，编程实现打印进程树

（用户空间编程）

二、实验基本知识：

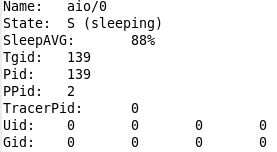
1.proc文件系统详细介绍

proc文件系统是一个伪文件系统，它只存在内存当中，而不占用外存空间。它以文件系统的方式为访问系统内核数据的操作提供接口。用户和应用程序可以通过 proc得到系统的信息，并可以改变内核的某些参数。由于系统的信息，如进程，是动态改变的，所以用户或应用程序读取proc文件时，proc文件系统是动态从系统内核读出所需信息并提交的。它的目录结构如下：  
目录名称目录内容  
apm 高级电源管理信息  
cmdline 内核命令行  
Cpuinfo 关于Cpu信息  
Devices 可以用到的设备（块设备/字符设备）  
Dma 使用的DMA通道  
Filesystems 支持的文件系统  
Interrupts 中断的使用  
Ioports I/O端口的使用  
Kcore 内核核心印象  
Kmsg 内核消息  
Ksyms 内核符号表  
Loadavg 负载均衡  
Locks 内核锁  
Meminfo 内存信息  
Misc 杂项  
Modules 加载模块列表  
Mounts 加载的文件系统  
Partitions 系统识别的分区表  
Rtc 实时时钟  
Slabinfo Slab池信息  
Stat 全面统计状态表  
Swaps 对换空间的利用情况  
Version 内核版本  
Uptime 系统正常运行时间

Cmdline 命令行参数  
Environ 环境变量值  
Fd 一个包含所有文件描述符的目录  
Mem 进程的内存被利用情况  
Stat 进程状态  
Status 进程当前状态，以可读的方式显示出来  
Cwd 当前工作目录的链接  
Exe 指向该进程的执行命令文件  
Maps 内存映象  
Statm 进程内存状态信息  
Root 链接此进程的root目录  
用户如果要查看系统信息，可以用cat命令。例如：  
# cat /proc/interrupts  
  
并不是所有这些目录在你的系统中都有，这取决于你的内核配置和装载的模块。另外，在/proc下还有三个很重要的目录：net，scsi和sys。Sys 目录是可写的，可以通过它来访问或修改内核的参数（见下一部分），而net和scsi则依赖于内核配置。例如，如果系统不支持scsi，则scsi目录不存在。  
除了以上介绍的这些，还有的是一些以数字命名的目录，它们是进程目录。系统中当前运行的每一个进程都有对应的一个目录在/proc 下，以进程的PID号为目录名，它们是读取进程信息的接口。而self目录则是读取进程本身的信息接口，是一个link。Proc文件系统的名字就是由之而起。

2. /proc目录下有一组以进程号pid为目录名的文件夹

每个目录中包含相应进程的信息，其中status文件：包含进程信息，如Pid (进程号), PPid (父进程号), Name(进程名)。如下图所示：

****

利用status文件中的父进程信息等内容，编程实现打印进程树。

（即：达到pstree命令显示样式）。

**三、 实验样例**

可以考虑从当前进程或init进程出发。

实验样例略

**实验四 遍历内核进程控制块打印进程树**

一、实验目的

1.理解task\_struct结构及进程管理机制

2.了解内核进程控制块链表遍历机制

3.掌握Linux内核模块基本编程技术

4.采用添加内核模块技术，打印进程树

二、实验原理

1、基本思想

每个进程都有一个task\_struct结构，包括进程之间的族系成员关系

pid\_t pid;

struct task\_struct \*parent;

struct list\_head children;

struct list\_head sibling;

char comm[16]; //进程名称

编程思路：建立内核模块，利用进程族系成员遍历进程树

（1）定位到根进程

for(cur=current; cur->pid!=1; cur=cur->parent);

（2）递归打印子进程（深度优先算法）

深度优先算法

递归结束条件：子进程pid=0

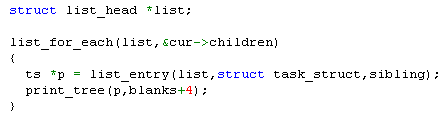
2、获取下一进程地址

途径：利用通用内核链表遍历方法

list\_for\_each(pos, head) //系统函数

list\_entry(ptr, type, member) //系统函数

#define list\_entry(ptr, type, member) container\_of(ptr, type, member)



list\_entry()说明

container\_of宏定义(include/linux/kernel.h)

#define container\_of(ptr, type, member) ({ \

//将链表中的元素ptr转换成结构type中成员member的类型

const typeof( ((type \*)0)->member ) \*\_\_mptr = (ptr); \

// \_\_mptr减去member成员偏移地址正好是type结构地址

(type \*)( (char \*)\_\_mptr - offsetof(type, member) );})

offsetof宏定义(include/linux/stddef.h)

#define offsetof(TYPE, MEMBER) ((size\_t) &((TYPE \*)0)->MEMBER)

size\_t定义为unsigned int(i386)

先计算结构成员在结构中的偏移量，然后根据成员变量的地址反过来得出属主结构变量的地址

((type \*)0)->member将0地址强制“转换”为type结构的指针，再访问到type结构中的member成员

3、**struct list\_head**

在Linux内核中，提供了一个用来创建双向循环链表的结构 list\_head。虽然linux内核是用C语言写的，但是list\_head的引入，使得内核数据结构也可以拥有面向对象的特性，通过使用操作list\_head 的通用接口很容易实现代码的重用，有点类似于C++的继承机制（希望有机会写篇文章研究一下C语言的面向对象机制）。下面就是kernel中的list\_head结构定义：

**struct list\_head {**

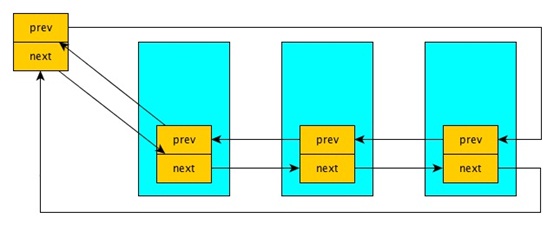
**struct list\_head \*next, \*prev;**

**};**

#define LIST\_HEAD\_INIT(name) { &(name), &(name) }

需要注意的一点是，头结点head是不使用的，这点需要注意。

使用list\_head组织的链表的结构如下图所示：



list\_head这个结构看起来怪怪的，它竟没有数据域！所以看到这个结构的人第一反应就是我们怎么访问数据？

此时list\_head就作为它的父结构中的一个成员了，当我们知道list\_head的地址（指针）时，我们可以通过list.c提供的宏 list\_entry 来获得它的父结构的地址。下面我们来看看list\_entry的实现:

#define list\_entry(ptr,type,member)\

　　container\_of(ptr,type,member)

#define offsetof(TYPE,MEMBER) ((size\_t)&((TYPE \*)0)->MEMBER)

#define container\_of(ptr,type,member) ( {\

　　const typeof( ((type\*)0)->member ) \*\_\_mptr=(ptr);\

　　(type\*)( (char\*)\_\_mptr - offsetof(type,member) );} )

这里涉及到三个宏，还是有点复杂的，我们一个一个来看：

#define offsetof(TYPE,MEMBER) ( (size\_t)& ((TYPE \*)0）-> MEMBER )

我们知道 0 地址内容是不能访问的，但 0地址的地址我们还是可以访问的， 这里用到一个取址运算符(TYPE \*)0 它表示将 0地址强制转换为TYPE类型，((TYPE \*)0）-> MEMBER 也就是从0址址找到TYPE 的成员MEMBER 。

offset宏就是算MEMBER在TYPE中的偏移量的。

我们再看第二个宏

#define container\_of(ptr,type,member) ( {\

　　const typeof( ((type\*)0)->member ) \*\_\_mptr=(ptr);\

　　(type\*)( (char\*)\_\_mptr - offsetof(type,member) );} )

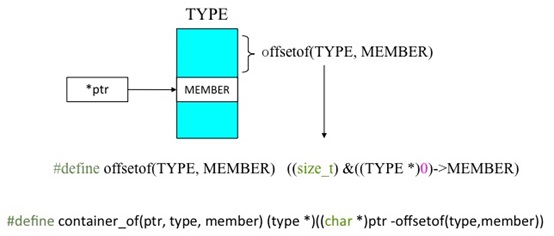
这个宏是由两个语句组成，最后container\_of返回的结果就是第二个表达式的值。这里\_\_mptr为中间变量，这就是list\_head指针类型，它被初始化为ptr的值，而ptr就是当前所求的结构体中list\_head节点的地址。为什么要用中间变量，这是考虑到安全性因素，如果传进来一个ptr++，所有ptr++放在一个表达式中会有副作用，像 (p++)+(p++)之类。

(char\*)\_\_mptr 之所以要强制类型转化为char是因为地址是以字节为单位的，而char的长度就是一个字节。

container\_of的值是两个地址相减，

刚说了\_\_mptr是结构体中list\_head节点的地址，offset宏求的是list\_head节点MEMBER在结构体TYPE中的偏移量，那么\_\_mptr减去它所在结构体中的偏移量，就是结构体的地址。

所以list\_entry(ptr,type,member)宏的功能就是，由结构体成员地址求结构体地址。其中ptr 是所求结构体中list\_head成员指针，type是所求结构体类型，member是结构体list\_head成员名。通过下图来总结一下：



4、**list\_for\_each**

继续列举一些双链表的常用操作：

双向链表的遍历——**list\_for\_each**

//注：这里prefetch 是gcc的一个优化选项，也可以不要

/\*head头指针，pos遍历链表的游走指针\*/

#define list\_for\_each(pos, head) \

         for (pos = (head)->next; prefetch(pos->next), pos != (head); \

                 pos = pos->next)

三、实验样例

1.参考程序

备注：本程序在unbutu 7下可运行。在高版本需要去掉头文件第三行，显示格式不正确

//pstree2.c

#include<linux/module.h>

#include<linux/sched.h>

#include <asm-i386/current.h>

#include<linux/list.h>

typedef struct task\_struct ts;

void print\_tree2(ts \*cur,int blanks){

int i;

for(i=1;i<=blanks;i++)

printk(" ");

printk("%s\n",cur->comm);

struct list\_head \*list;

list\_for\_each(list,&cur->children){

ts \*p=list\_entry(list,struct task\_struct,sibling);

print\_tree2(p,blanks+4);

}

}

int init\_pstree2(void){

ts \*cur;

for(cur=current;cur->pid!=1;cur=cur->parent);

print\_tree2(cur,0);

return 0;

}

void exit\_pstree2(void){

}

module\_init(init\_pstree2);

module\_exit(exit\_pstree2);

2.Makefile文件

obj-m := pstree2.o

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

PWD := $(shell pwd)

modules:

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

clean:

rm -rf \*.o .\*.cmd \*.ko \*.mod.c

**3.运行方法**

编译

make

安装模块

sudo insmod pstree2.ko

运行模块

sudo dmesg

删除模块

sudo rmmod pstree2

**实验五 添加系统调用**

一、实验目的：

1.掌握系统调用添加方法

2.重新编译内核，添加系统调用

二、实验基本知识：

1.下载Linux-\*.\*.\*版本内核，拷贝到/usr/src下， 使用tar xvf linux-\*.\*.\*.tar.xz解压到/usr/src下。

2.编写系统调用函数

输入命令：cd /usr/src/linux-\*.\*.\*/kernel

vi sys.c

在文件最后添加系统函数：

asmlinkage int sys\_mycall(int number)

{

printk(“my first syscall”);

return number;

}

1. 修改相应文件

低版本内核如linux-2.6.\*：

输入命令：cd /usr/src/linux-new/arch/i386/kernel

vi entry.S

.long SYMBOL\_NAME(sys\_mycall)

输入命令：cd /usr/src/linux-new/include/asm-i386

vi unistd.h

#define \_\_NR\_mycall 251

输入命令：cd /usr/include/asm

vi unistd.h

#define \_\_NR\_mycall 251

高版本内核如linux-4.15.\*以上：

(1)打开/usr/src/linux-4.\*.\*/arch/x86/include/asm/的syscalls.h，或者/usr/src/linux-4.\*.\*/include/linux/syscalls.h

插入声明 asmlinkage int sys\_mycall (int number);

（2）在64位系统下，打开/usr/src/linux-4.19.11/arch/x86/entry/syscalls下的syscall\_64.tbl

在64位的调用指令下添加新的系统调用号：

336 common mycall sys\_ mycall

或者添加

336 64 mycall sys\_ mycall

1. 编译新内核和安装内核

回到内核源码目录/usr/src/linux-4.\*.\*，同时使用uname -a查看旧内核版本号

依次输入这四条语句

sudo make mrproper

sudo make clean //编译出错记得前使用两步清除一下

sudo make menuconfig（保存配置文件）或者可以sudo make oldconfig，生成配置文件

sudo make bzImage

sudo make modules //编译模块

sudo make modules\_install //添加模块

或者

sudo make -j8（八线程编译，和虚拟机或者cpu线程数有关，cpu i7-8650是四核八线程，设置j8.如果4线程设置为j4）.此种方法一次性编译所有,相当于前三步。

sudo make install //安装内核

#整个过程中缺什么资源，就用sudo apt-get install安装什么资源。可能需要安装资源有

sudo apt-get install libncurses5-dev

apt-get install bison

apt-get install flex

apt-get install libssl-dev

apt-get install zlib1g-dev

1. 启动新内核

低版本内核：

输入命令：cp /usr/src/linux-new/arch/i386/boot/bzImage /boot/bzImage-jz

cp /usr/src/linux-new/System.map /boot/System.map-jzy

ln –sf /boot/System.map-jzy /boot/System.map

低版本、高版本内核设置grub启动：

系统采用grub引导内核需要在/etc/default/grub中修改相应代码，将GRUB\_HIDDEN\_TIMEOUT=0前面加#，注释掉。这是根据引导方式的不同来修改的，不同的机子引导方式不同，所以不做详细解释。

或者开机时，长按shift ，出现grub引导界面。

启动新内核

uname -a //查看新内核版本号

6. 使用新的系统及系统调用

编写一个程序如下：

#include <linux/kernel.h>

#include <sys/syscall.h>

#include <unistd.h>

#include <stdio.h>

int main(int argc,char \*\*argv)

{

printf("System call sys\_mycall %ld\n",syscall(336, 125));

return 0;

}

输入命令：gcc \*.c –o mycall.exe

./mycall

结果是：125

dmesg 查看打印语句

7. 命令的解释说明：

（1）make mrproper命令会删除所有的编译生成文件、内核配置文件(.config文件)和各种备份文件，所以几乎只在第一次执行内核编译前才用这条命令。

（2）make clean命令则是用于删除大多数的编译生成文件，但是会保留内核的配置文件.config，还有足够的编译支持来建立扩展模块。所以你若只想删除前一次编译过程的残留数据，只需执行make clean命令。

（3）make oldconfig的作用是备份当前.config文件为.config.old，如若make config/menuconfig设置不当可用于恢复先前的.config

Make oldconfig// 命令行配置（不断的问你Y/N/M）  
make menuconfig// 命令行下的简单图形界面配置(集中配置，最后一次性保存), 配置内核选项  
make xconfig// GUI图形界面的配置（集中配置，最后一次性保存）

（4）vmlinuz是可引导的、压缩的内核。“vm”代表“Virtual Memory”。Linux 支持虚拟内存，不像老的操作系统比如DOS有640KB内存的限制。Linux能够使用硬盘空间作为虚拟内存，因此得名“vm”。vmlinuz是可执行的Linux内核，它位于/boot/vmlinuz，它一般是一个软链接。

vmlinuz的建立有两种方式。一是编译内核时通过“make zImage”创建，然后通过：“cp /usr/src/linux-2.4/arch/i386/linux/boot/zImage /boot/vmlinuz”产生。zImage适用于小内核的情况，它的存在是为了向后的兼容性。二是内核编译时通过命令make bzImage创建，然后通过：“cp /usr/src/linux-2.4/arch/i386/linux/boot/bzImage /boot/vmlinuz”产生。bzImage是压缩的内核映像，需要注意，bzImage不是用bzip2压缩的，bzImage中的bz容易引起误解，bz表示“big zImage”。bzImage中的b是“big”意思。

zImage(vmlinuz)和bzImage(vmlinuz)都是用gzip压缩的。它们不仅是一个压缩文件，而且在这两个文件的开头部分内嵌有gzip解压缩代码。所以你不能用gunzip 或 gzip –dc解包vmlinuz。

内核文件中包含一个微型的gzip用于解压缩内核并引导它。两者的不同之处在于，老的zImage解压缩内核到低端内存(第一个640K)，bzImage解压缩内核到高端内存(1M以上)。如果内核比较小，那么可以采用zImage 或bzImage之一，两种方式引导的系统运行时是相同的。大的内核采用bzImage，不能采用zImage。

vmlinux是未压缩的内核，vmlinuz是vmlinux的压缩文件。

vmlinux 是ELF文件，即编译出来的最原始的文件。

vmlinuz应该是由ELF文件vmlinux经过OBJCOPY后，并经过压缩后的文件

zImage是vmlinuz经过gzip压缩后的文件，适用于小内核

bzImage是vmlinuz经过gzip压缩后的文件，适用于大内核

（5）make modules //编译模块，相关的\*.ko驱动模块就在kernel目录下!

下载到板子启动的内核zImage  
选择（\*）编译到zImage里面,开机加载  
选择（M）则不会编译到zImage，编译成模块后（make modules）需要的时候，insmod module\_name

make modules是编译模块，很多驱动还有功能在选的时候选成M的都是modules，不过直接make，不加任何参数，就是make all，包含make modules。不用额外加此make，但是在安装的时候make install只是安装bzimage，Systemmap。没有把modules安装好，还要额外的make modules\_install，把模块放到/lib/modules文件夹一个和内核名一样的文件夹下，并且运行depmod生成模块依赖关系文件，系统启动时加载模块就是从dep里面读取信息加载模块。

（6）make modules\_install：安装模块  
执行命令 make modules\_install 就好了  
（7）make install ：安装内核相关文件  
执行命令 make install安装内核相关文件，主要目的将所有的编译之后的内容拷贝安装到相应的目录。安装bzImage为/boot/vmlinuz-VERSION-RELEASE （去boot目录下查看），生成initramfs文件。

（8）查看配置文件/etc/grub.conf、/etc/lilo.conf，确定是grub还是lilo启动管理器。/boot那个文件夹里面如果有grub那个文件夹，系统就应该使用的是grub。

GNU GRUB（GRand Unified Bootloader简称“GRUB”）是一个来自[GNU](https://baike.baidu.com/item/GNU" \t "_blank)项目的多[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F/192" \t "_blank)启动程序。GRUB是多启动规范的实现，它允许用户可以在[计算机](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA/140338" \t "_blank)内同时拥有多个[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F/192)，并在计算机启动时选择希望运行的操作系统。GRUB可用于选择[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F/192" \t "_blank)分区上的不同[内核](https://baike.baidu.com/item/%E5%86%85%E6%A0%B8/108410)，也可用于向这些[内核](https://baike.baidu.com/item/%E5%86%85%E6%A0%B8/108410)传递启动参数。

GNU GRUB 和GRUB是GRand Unified Bootloader的缩写，它是一个多重[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F" \t "_blank)启动管理器。用来引导不同系统，如windows，linux。在[X86架构](https://baike.baidu.com/item/X86%E6%9E%B6%E6%9E%84" \t "_blank)的机器中，Linux、[BSD](https://baike.baidu.com/item/BSD" \t "_blank) 或其它Unix类的[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F" \t "_blank)中GRUB、[LILO](https://baike.baidu.com/item/LILO" \t "_blank) 是大家最为常用，应该说是主流。

#Initrd简介：

initrd是linux在系统引导过程中使用的一个临时的根文件系统，用来支持两阶段的引导过程。

（9）system.map简介：

内核符号映射表，顾名思义就是将内核中的符号（也就是内核中的函数）和它的地址能联系起来的一个列表。是所有符号及其对应地址的一个列表。之所以这样就使为了用户编程方便，直接使用函数符号就可以了，而不用去记要使用函数的地址。

当你编译一个新内核时，原来的System.map中的符号信息就不正确了。随着每次内核的编译，就会产生一个新的 System.map文件，并且需要用该文件取代原来的文件。

#ln -s = soft，意为软链接 -f = force，表示强制创建

1. 实验样例

本校样例

本校样例1：

一、实验目的：

1.掌握系统调用添加方法

2.重新编译内核，添加系统调用

二、实验原理及过程：

使用ubuntu18.04和4.19.11内核

（1）下载Linux-4.19.11版本内核（已有4.18和5.0），解压到/usr/src下

（2）打开/usr/src/linux-4.19.11/kernel下的sys.c,添加函数

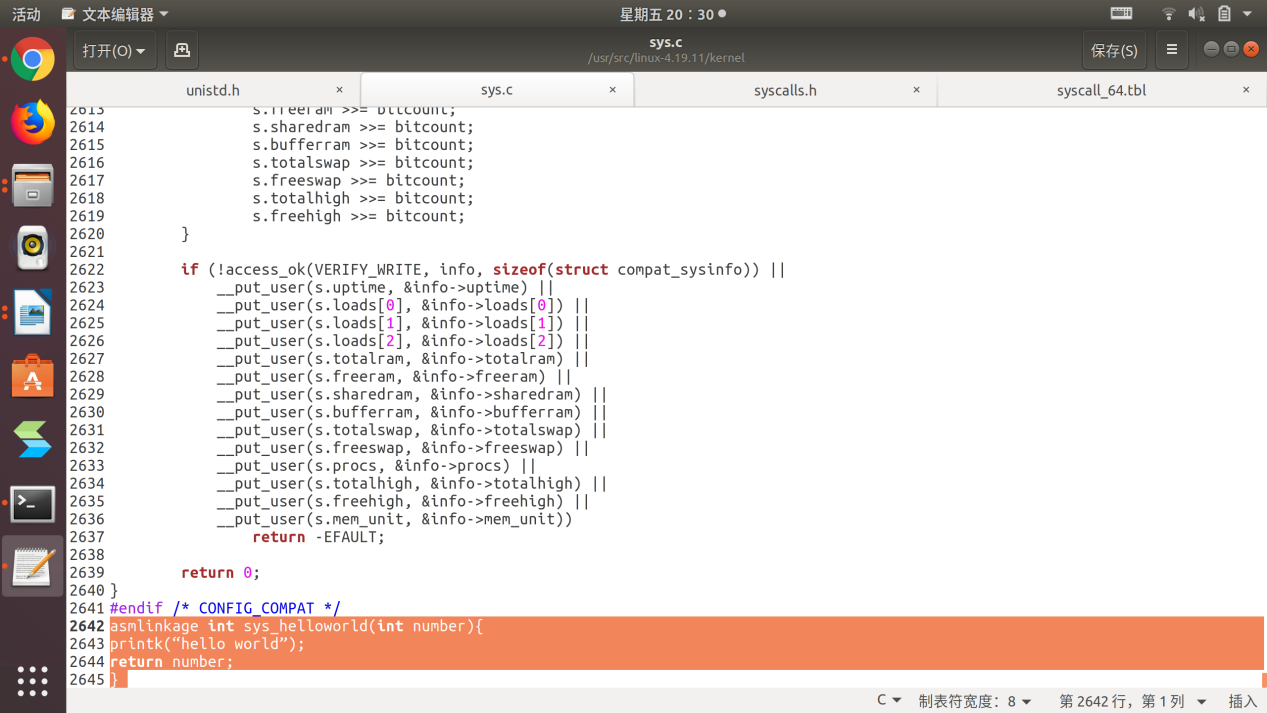
asmlinkage int sys\_helloworld(int number)

{

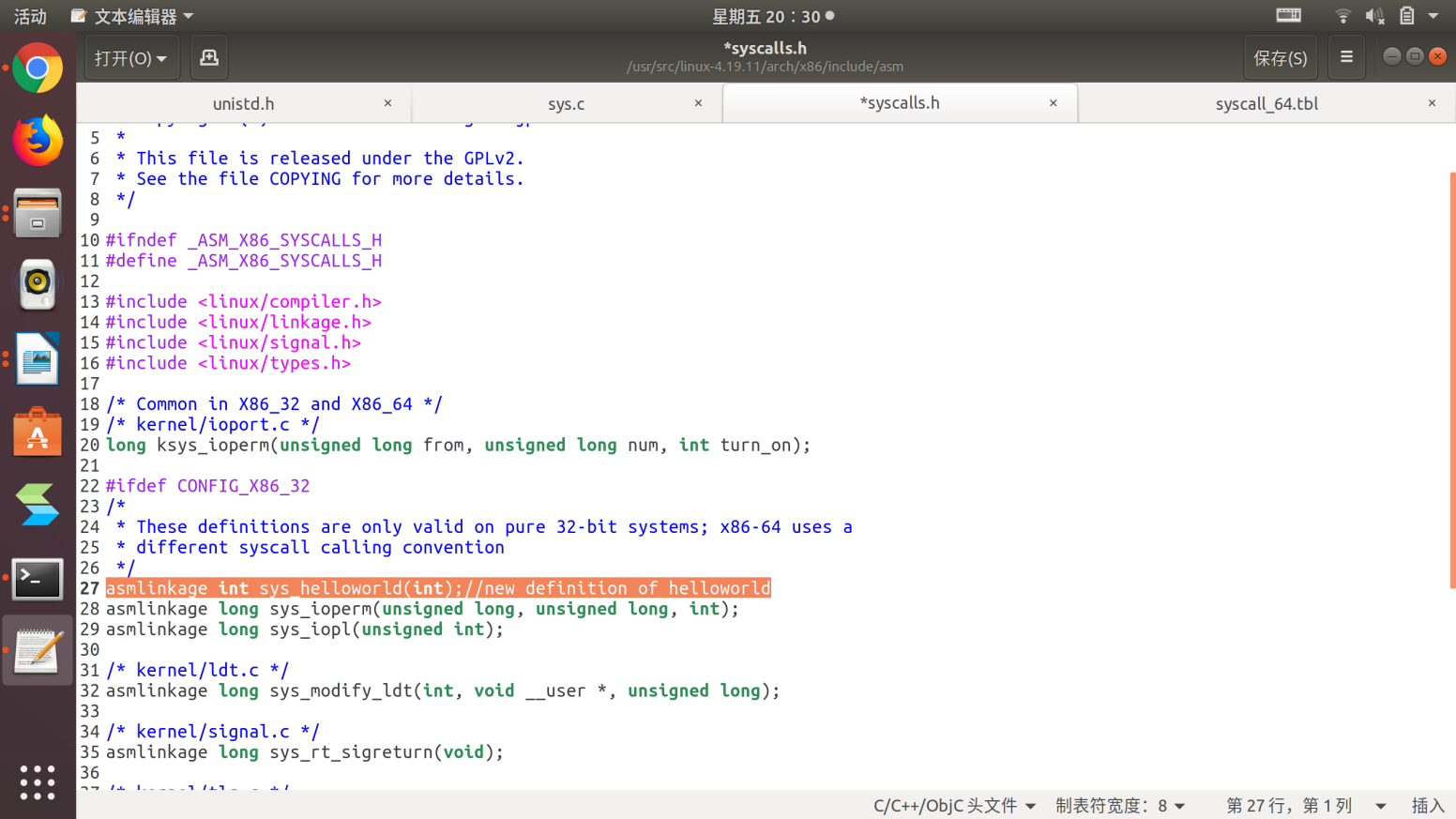
printk("hello world");

return number;

}



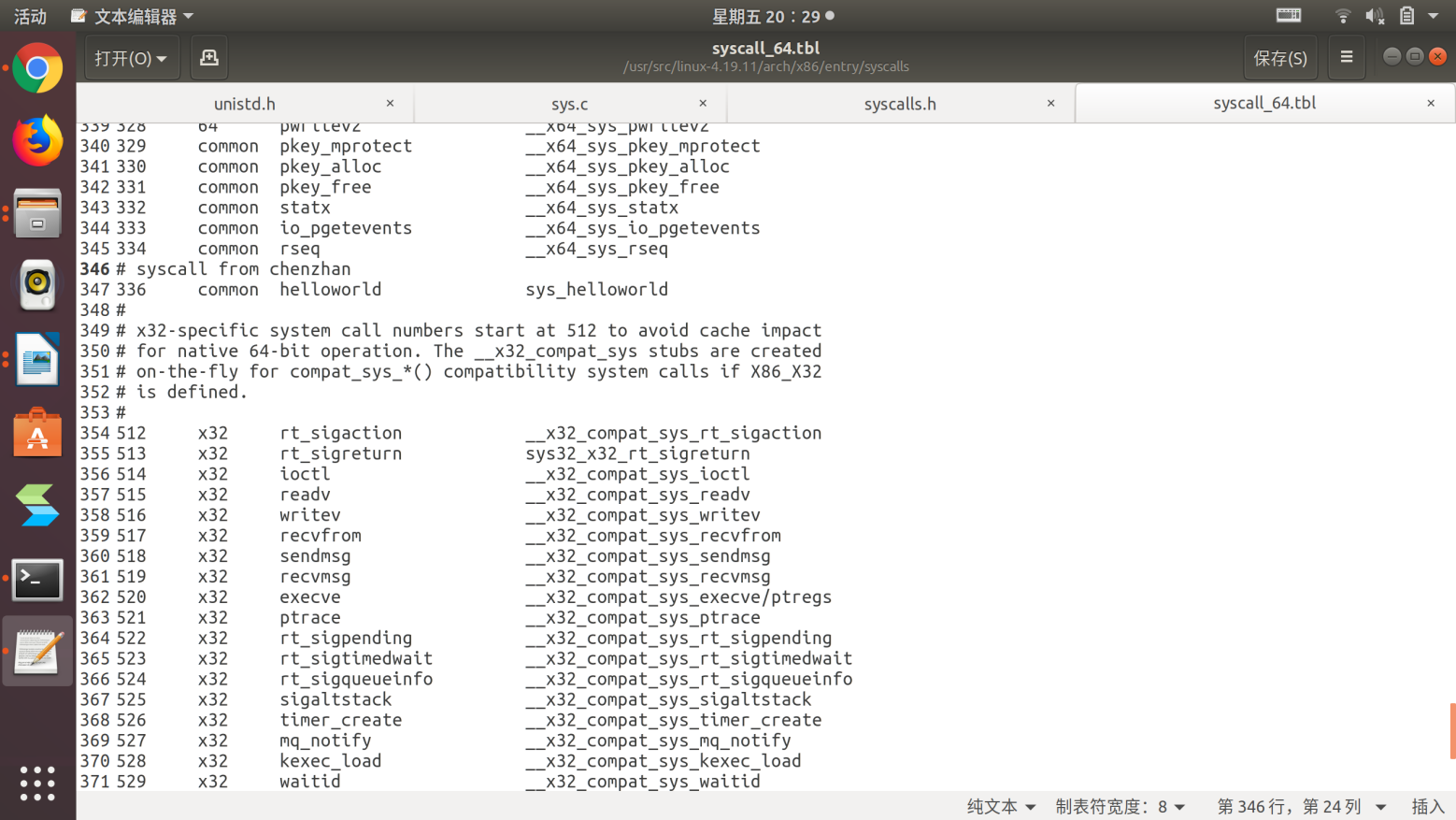
（3）打开/usr/src/linux-4.16.10/arch/x86/include/asm/的syscalls.h，插入声明 asmlinkage int sys\_HelloWorld (int number);



（4）在64位系统下，打开/usr/src/linux-4.19.11/arch/x86/entry/syscalls下的syscall\_64.tbl

在64位的调用指令下添加系统调用号

336 common helloworld sys\_HelloWorld



回到内核目录/usr/src/linux-4.19.11

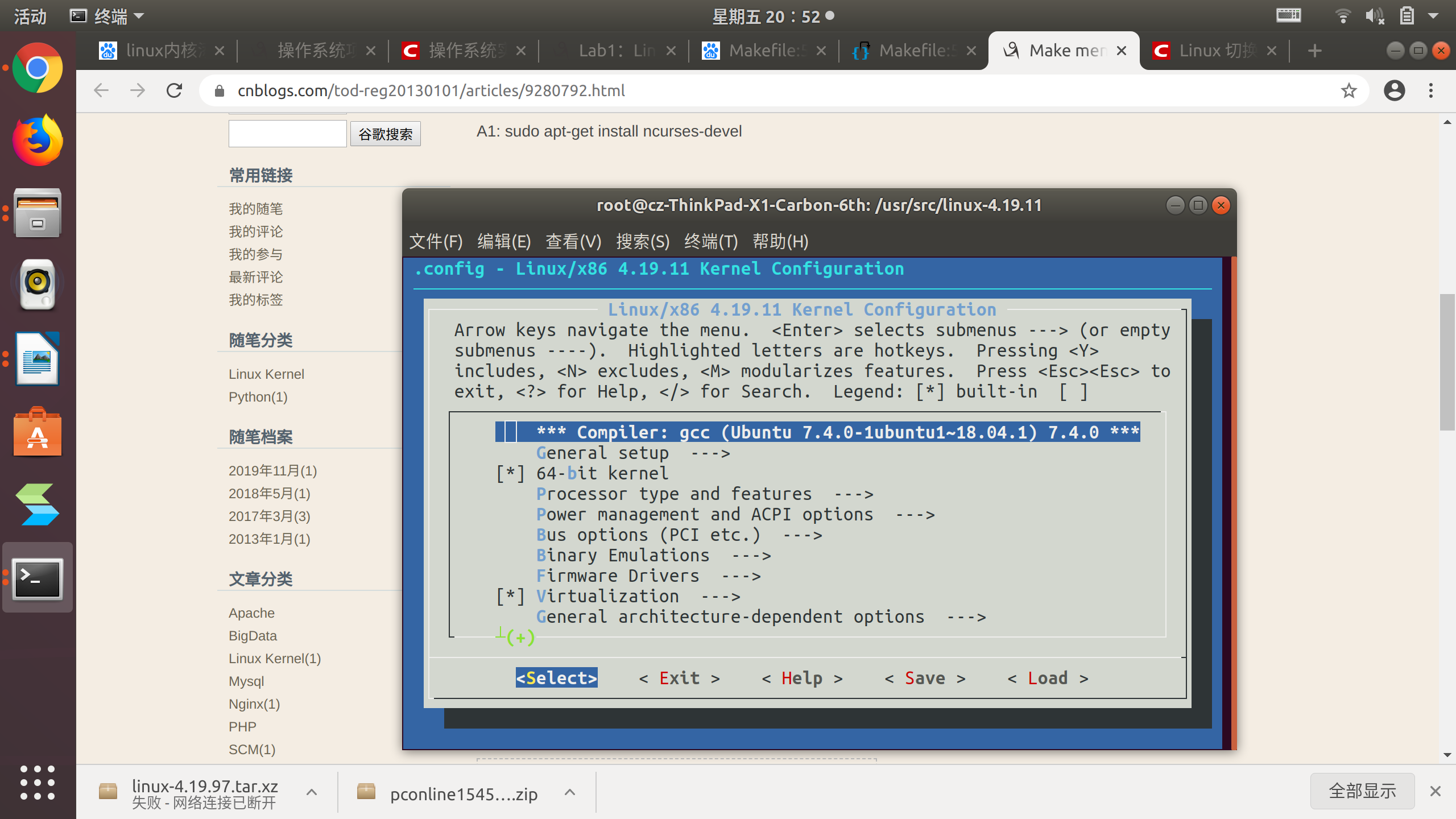
（4）编译内核和安装内核.

依次输入这四条语句

sudo make mrproper

sudo make clean//编译出错记得前两步清除一下

sudo make menuconfig（保存配置文件）/也可以make oldconfig，都是生成配置文件



此时一定记得保存，这里没保存直接退出编译了两次才发现问题

make bzImage（镜像）

make modules（模块）

make modules\_install（模块添加）

或者

sudo make -j8（八线程编译，和虚拟机或者cpu线程数有关，我的i7-8650四核八线程）

此种方法一次性编译所有

建议前一种方式，虽然较慢，但是分步骤编译出问题改起来方便一点，多线程编译的时候很多错误信息会被折叠，等安装模块直接保存不易发现问题，浪费了我四五个小时。

编译完成后输入sudo make install安装内核

重启，在ubuntu高级选项中选择4.19内核（双系统这样处理，虚拟机不清楚）

使用程序调用新加的系统调用

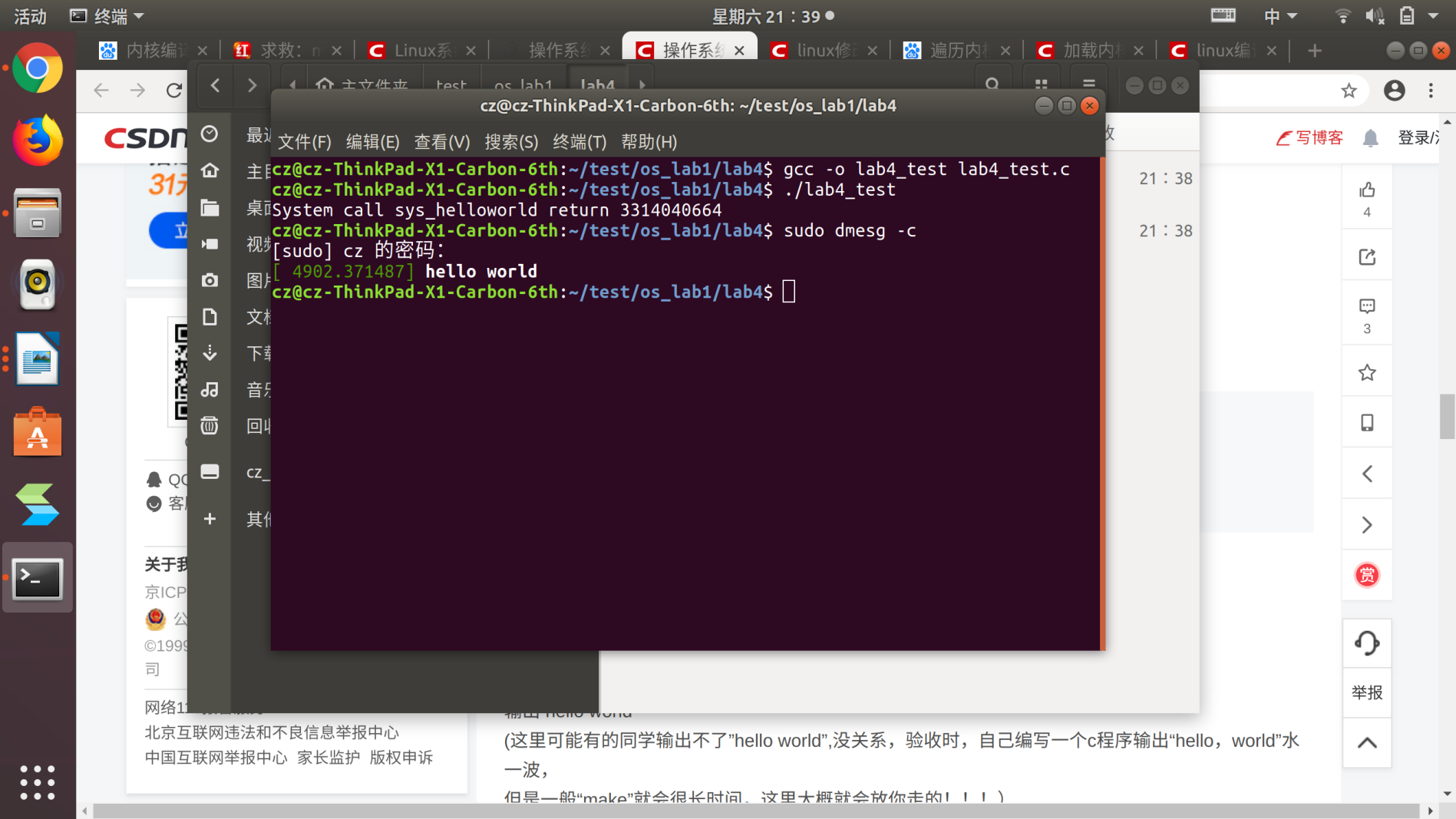
整个过程中需要安装较多的库，建议换国内的清华镜像或者阿里云镜像（华为的给我来个报错最后发现get工具兼容性问题？）

三、实验程序：

见附录

四、实验截图及结果分析:

调用成功，打印缓冲区hello world（printk不能打印到屏幕上，我的调用函数给了传参变量，由于没有给调用函数传值，显示的是缓冲空间一个乱值）



本校样例2：

1. 实验目的

1.掌握系统调用添加方法

2.重新编译内核，添加系统调用

二、实验原理

1. 编写系统函数

输入命令：cd /usr/src/linux-new/kernel

vi sys.c

在文件最后添假系统函数：

asmlinkage int sys\_mycall(int number)

{

return number;

}

1. 修改相应文件

输入命令：cd /usr/src/linux-new/arch/i386/kernel

vi entry.S

.long SYMBOL\_NAME(sys\_mycall)

输入命令：cd /usr/src/linux-new/include/asm-i386

vi unistd.h

#define \_\_NR\_mycall 251

输入命令：cd /usr/include/asm

vi unistd.h

#define \_\_NR\_mycall 251

1. 编译新内核

输入命令：cd /usr/src/linux-new

make mrproper

make oldconfig

make bzImage

make modules

make modules\_install

make install

1. 启动新内核

输入命令：cp /usr/src/linux-new/arch/i386/boot/bzImage /boot/bzImage-jzy

cp /usr/src/linux-new/System.map /boot/System.map-jzy

ln –sf /boot/System.map-jzy /boot/System.map

我的系统采用grub引导内核需要在/boot/grub.conf中修改相应代码。这是根据引导方式的不同来修改的，不同的机子引导方式不同，所以不做详细解释。

1. 使用新的系统

编写一个程序如下：

#include <linux/unistd.h>

\_syscall1(int,mycall,int,ret)

main()

{

printf(“%d \n”,mycall(125));

}

输入命令：gcc –o jzy jzy.c

./jzy

结果是：125

三、实验程序

在/usr/src/linux-2.6.29.6/kernel/sys.c文件中添加源代码，如下所示：

asmlinkage int sys\_mycall(int number)

　　{

　　 return number;

}

自己编写的测试程序如下：

#include <linux/unistd.h>

long sys\_mysyscall1(int，mysyscall,int，ret);

int main()

{

printk("My name is oyz,and my number is %d.\n",mysyscall(0));

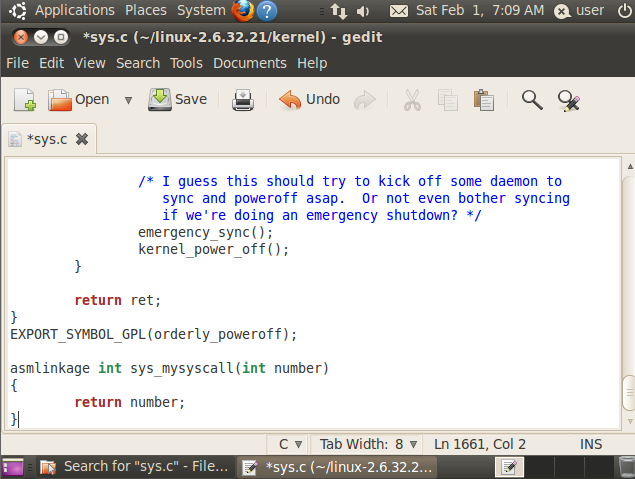
printk("The last of my number is %d.\n",mysyscall(1));

return 0;

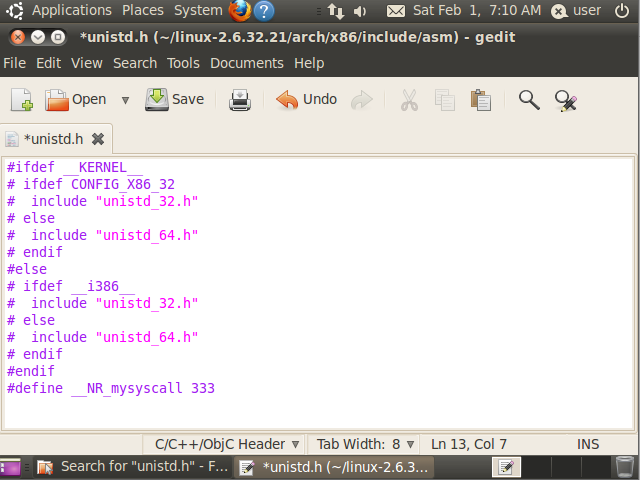
}

四、实验截图及结果分析

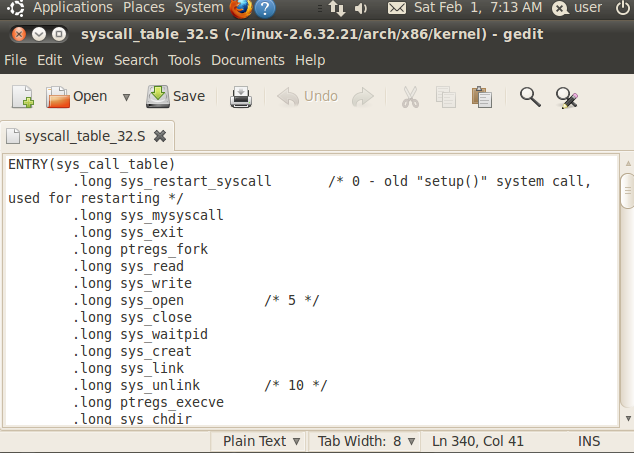
**编写系统函数**



**修改unistd.h系统调用清单**

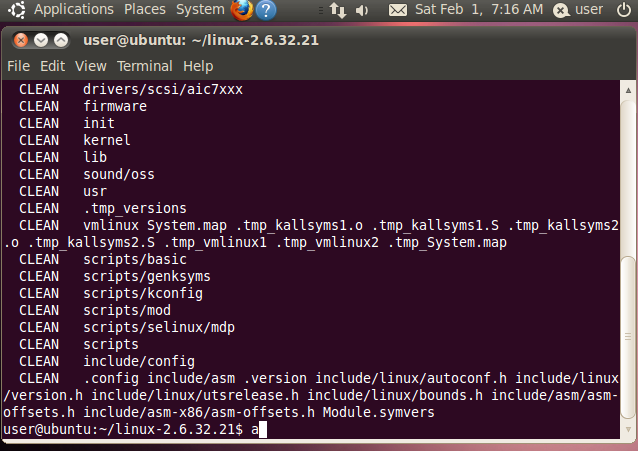


**修改syscall\_table\_32.S文件**



**重建内核：**

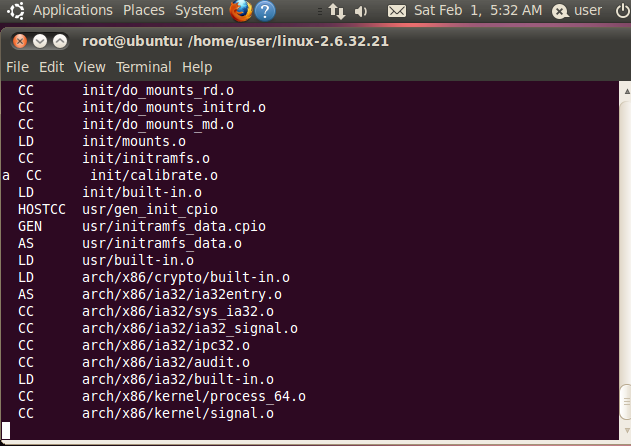
**Make mrprper**

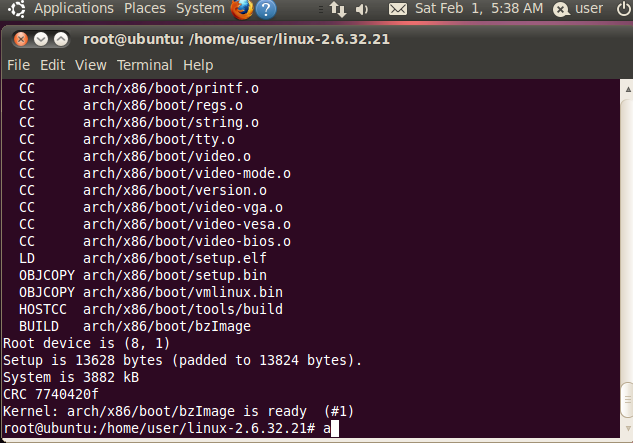


**Make oldconfig**

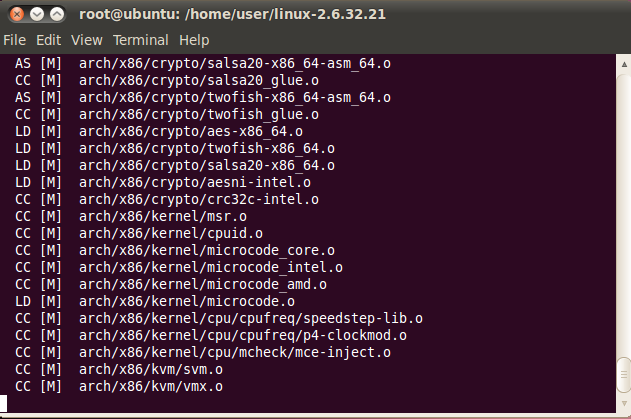


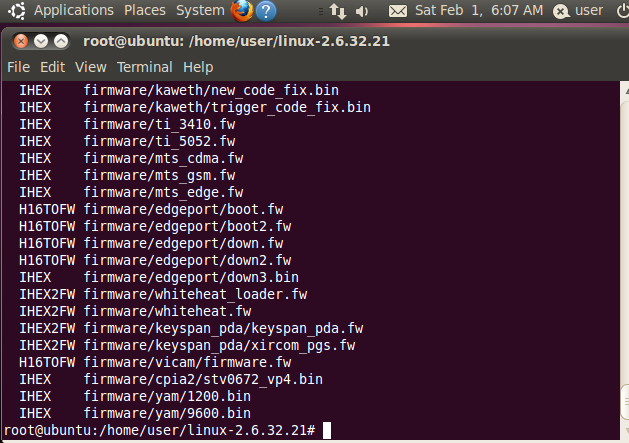
**Make bzImage**



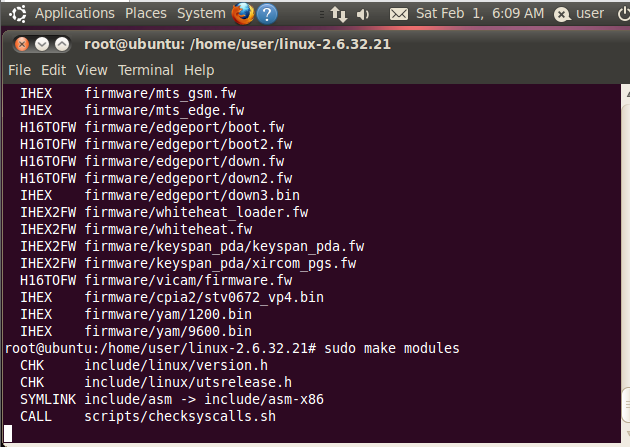


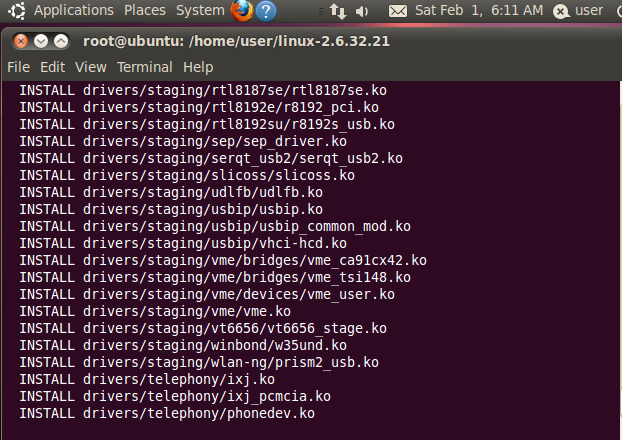
**Make modules**



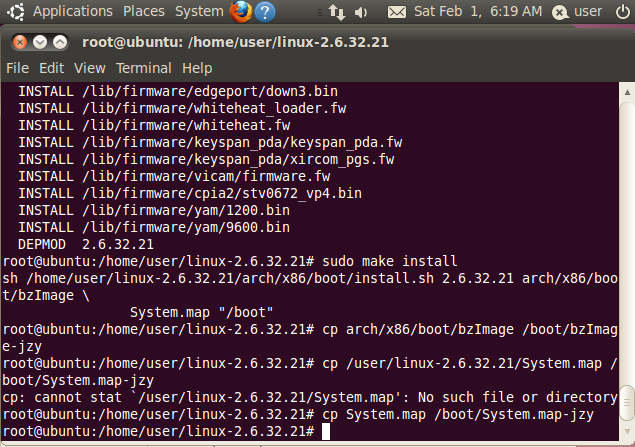


**Make modules\_install**

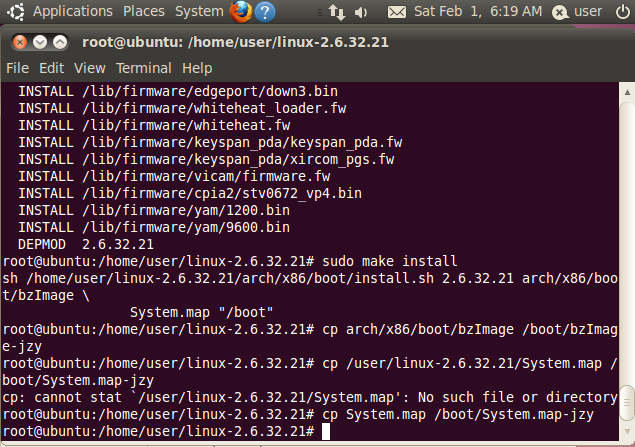


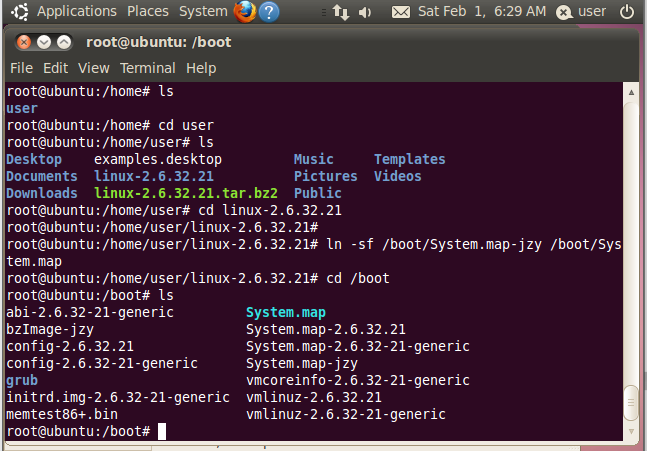


**Make install**

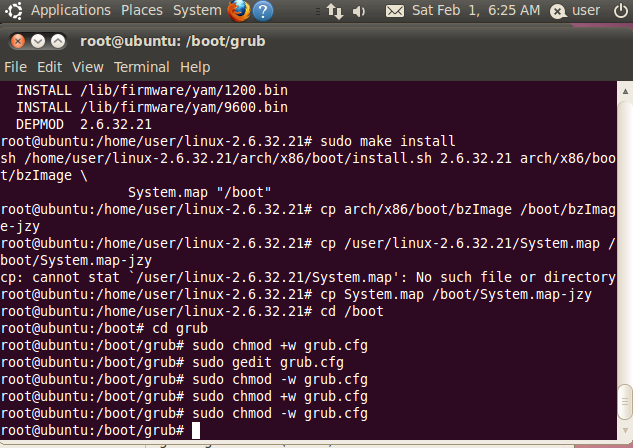


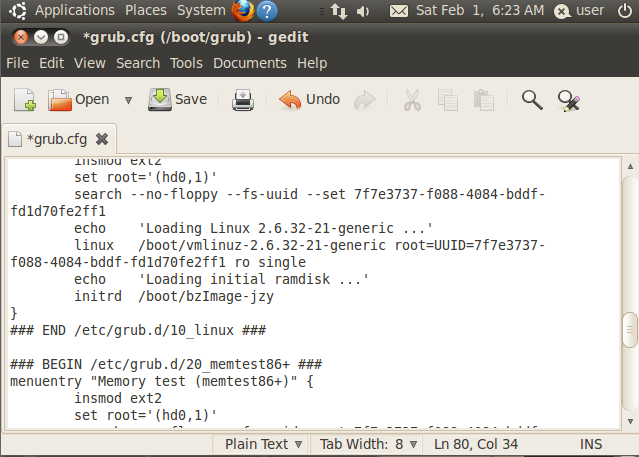
**在启动列表中添加新内核镜像**



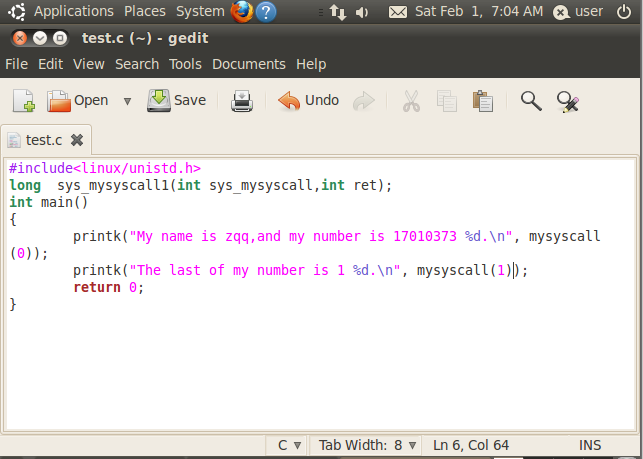


**用grub引导内核挂载：**

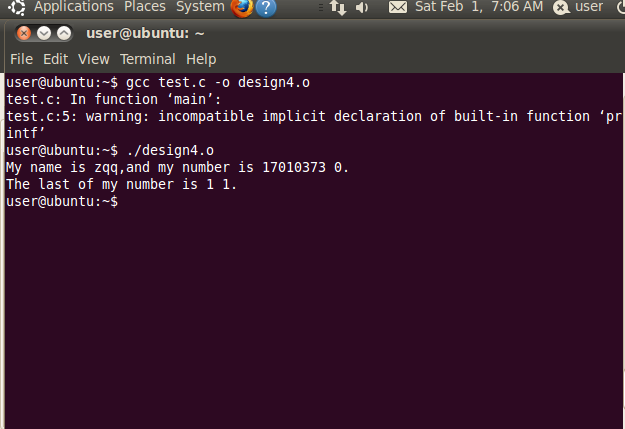




**自己编写一个程序**



**运行结果如下**



**本校样例3：**

**一、实验目的：**

1.掌握系统调用添加方法

2.重新编译内核，添加系统调用

**二、实验原理：**

注：因为和实验指导书的Ubuntu版本差异较大，所以换了个内核和方式

下载Linux-4.19.76版本内核

解压到/usr/src下

打开/usr/src/linux-4.19.76/kernel下的sys.c,

在最后添加asmlinkage long sys\_HelloWorld(void)

{

printk("Damn,you finally made it, bro!");

return 114514;

}

打开/usr/src/linux-4.16.10/arch/x86/include/asm/的syscalls.h

插入声明 asmlinkage long sys\_HelloWorld (void);

因为是64位系统，所以打开/usr/src/linux-4.19.76/arch/x86/entry/syscalls下的syscall\_64.tbl

在64位的调用指令下添加系统调用号

336 common HelloWorld sys\_HelloWorld

回到内核目录/usr/src/linux-4.19.76

编译内核和安装内核.

依次输入这四条语句

sudo make mrproper

sudo make clean

sudo make menuconfig

sudo make -j4（四核编译）

编译完成后输入sudomake install安装内核

重启长按shift后选择写好的内核

使用程序调用新加的系统调用

整个过程中缺什么就用sudo apt-get install安装什么。

**三、实验程序：**

#include <linux/kernel.h>

#include <sys/syscall.h>

#include <unistd.h>

#include <stdio.h>

int main(int argc,char \*\*argv)

{

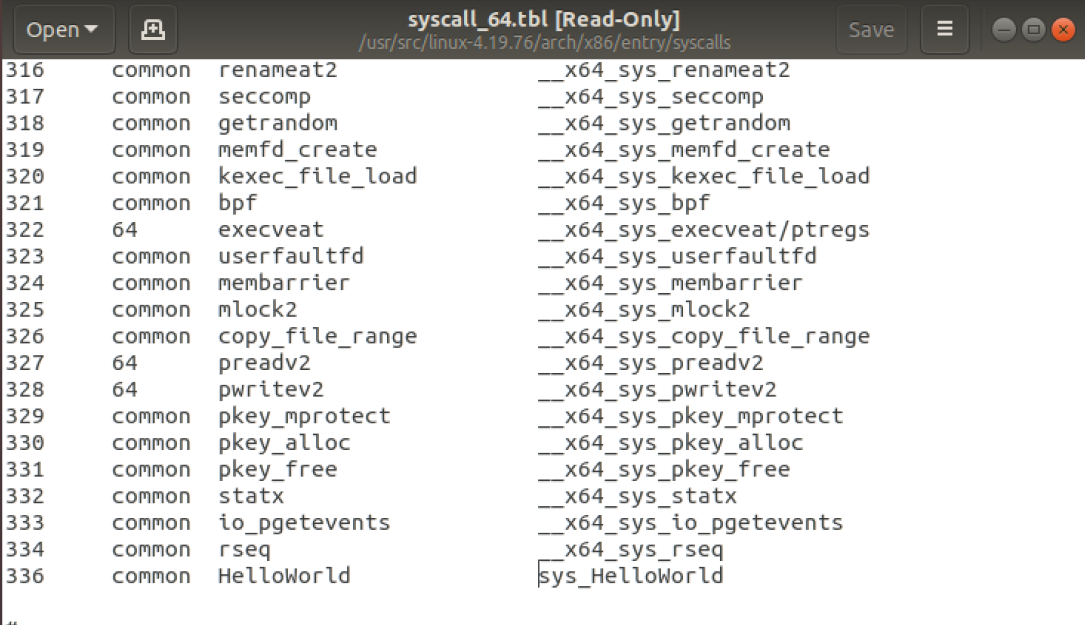
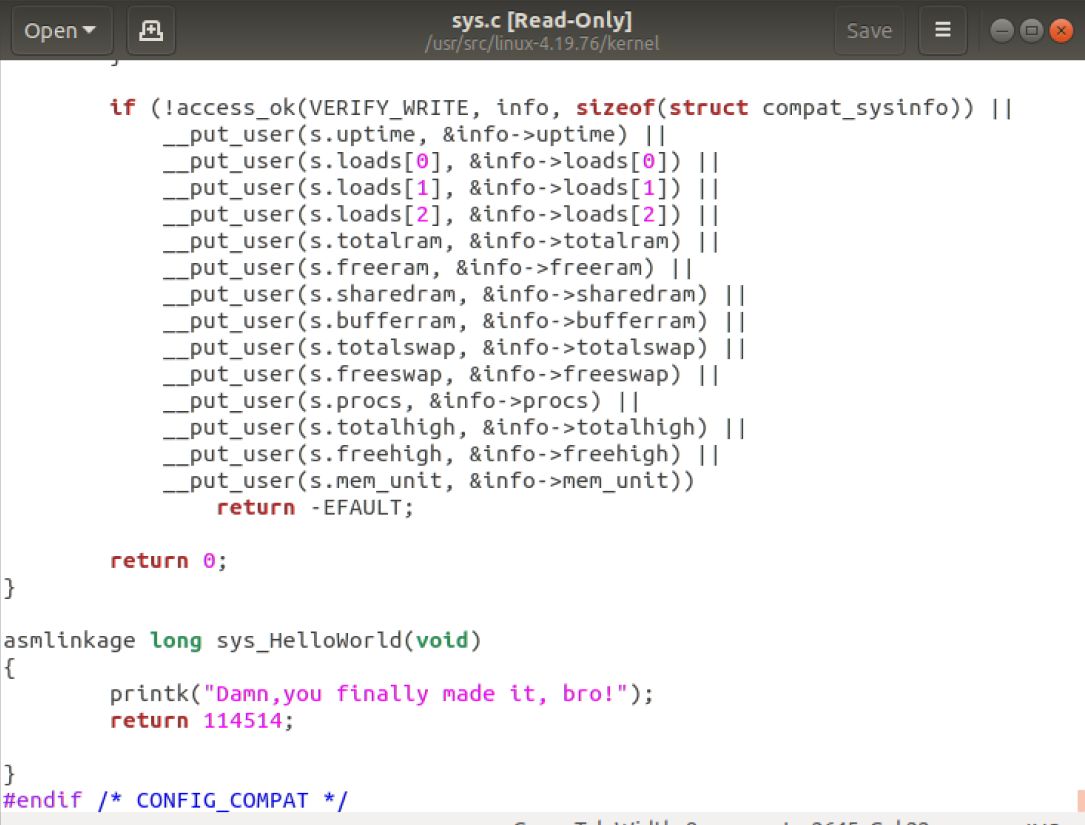
//336:long sys\_helloworld()

printf("System call sys\_helloworld return %ld\n",syscall(336));

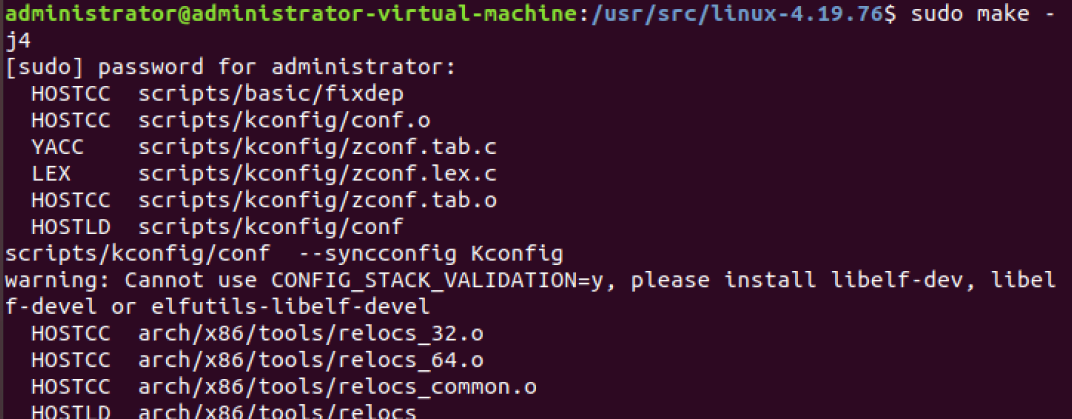
return 0;

}

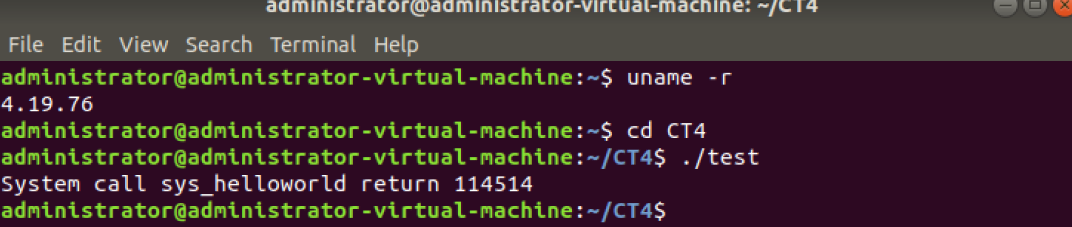
**四、实验截图及结果分析:**

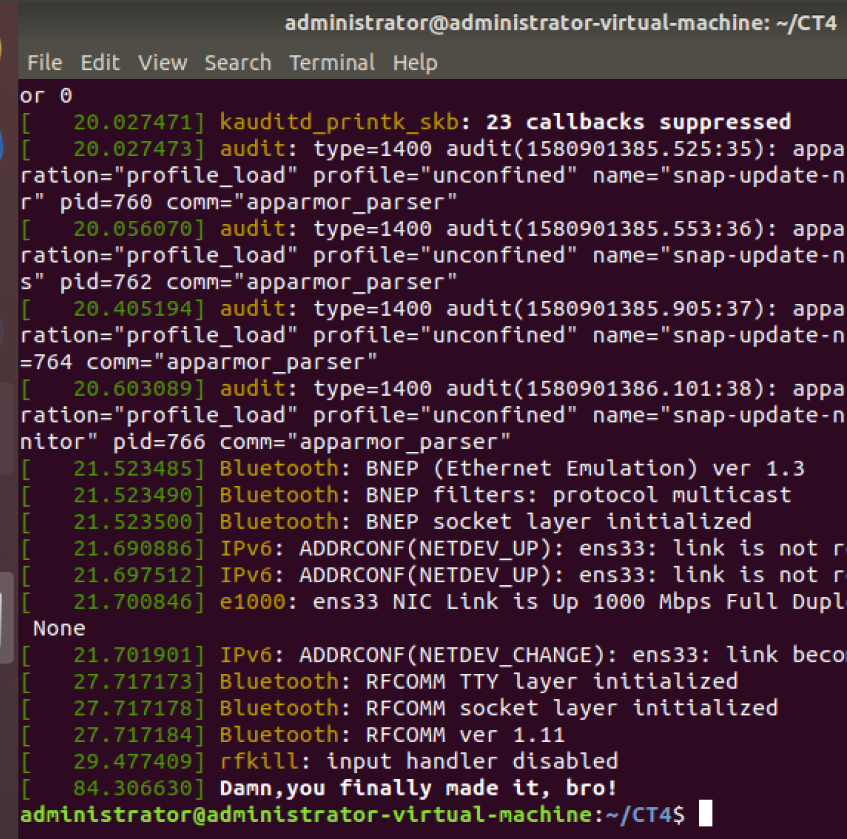


开始编译：



调用成功





**本校样例4：**

**一、实验目的：**

1. 掌握系统调用添加方法
2. 重新编译系统内核，添加系统调用
3. **实验基本知识：**

1.修改相应文件

输入命令：cd linux-4.15.3/arch/x86/entry/syscalls/

ls 修改syscall\_64.tbl,在文件最后添加以下内容：

333 64 mycall sys\_mycll

输入命令：cd linux-4.15.3/include/linux

Ls 修改syscalls.h文件,在最后的#endif之前添加以下内容:

asmlinkage long sys\_mycall(struct timeval \*tv);

2.编写系统调用函数

输入命令：cd linux-4.15.3/kernel

vim sys.c 添加系统调用函数

SYSCALL\_DEFINE1(mycall, struct timeval \*, tv) {

struct timeval ktv;

do\_gettimeofday(&ktv);

if(copy\_to\_user(tv, &ktv, sizeof(ktv) ) ) {

return -EFAULT;

}

return 0;

}

3.编译新内核并安装

输入命令：cd linux-4.15.3

# 清除残留的文件，如果编译失败，需要重新编译则需要清除，第一次编译可以不执行

sudo make mrproper

sudo make clean

sudo make menuconfig

# 编译，这里根据你为虚拟机分配的物理核决定 -jn，相同即可

sudo make –j8

或者 sudo make

# 安装内核模块

sudo make modules\_install

# 安装内核

sudo make install

4.使用新的系统

编写一个程序如下：

#include <unistd.h>

#include <sys/time.h>

#include <sys/syscall.h>

#include <stdio.h>

#define \_\_NR\_mycall 333 //系统调用号

int main() {

struct timeval gettime;

struct timeval mycalltime;

gettimeofday(&gettime, NULL);

syscall(\_\_NR\_mycall, &mycalltime);

printf("gettimeofday:%ld %ld \n", gettime.tv\_sec, gettime.tv\_usec);

printf("mycall : %ld %ld \n", mycalltime.tv\_sec, mycalltime.tv\_usec);

return 0;

}

结果返回当前时间

1. **实验程序：**

下载的内核源代码版本：linux-4.15.3

Ubuntu版本：16.04

系统调用函数如下：

SYSCALL\_DEFINE1(mycall, struct timeval \*, tv) {

struct timeval ktv;

do\_gettimeofday(&ktv);

if(copy\_to\_user(tv, &ktv, sizeof(ktv) ) ) {

return -EFAULT;

}

return 0;

}

新内核系统调用测试函数如下：

#include <unistd.h>

#include <sys/time.h>

#include <sys/syscall.h>

#include <stdio.h>

#define \_\_NR\_mycall 333 //系统调用号

int main() {

struct timeval gettime;

struct timeval mycalltime;

gettimeofday(&gettime, NULL);

syscall(\_\_NR\_mycall, &mycalltime);

printf("gettimeofday:%ld %ld \n", gettime.tv\_sec, gettime.tv\_usec);

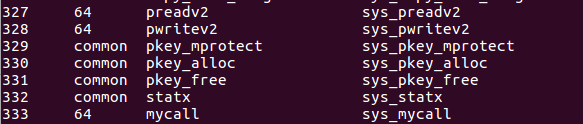
printf("mycall : %ld %ld \n", mycalltime.tv\_sec, mycalltime.tv\_usec);

return 0;

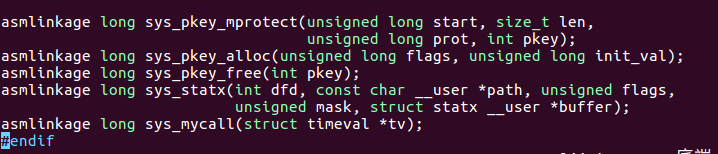
}

**四、实验截图及结果分析：**

修改syscall\_64.tbl



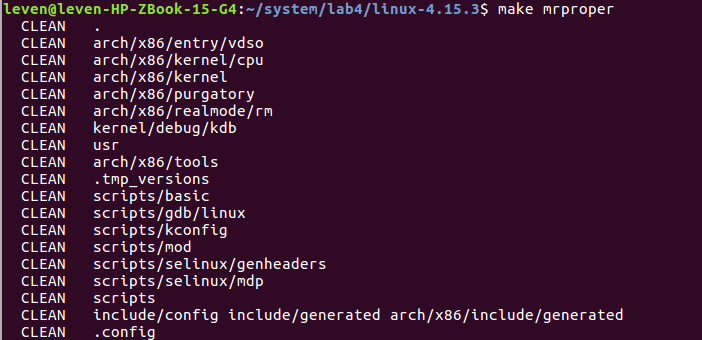
修改syscalls.h



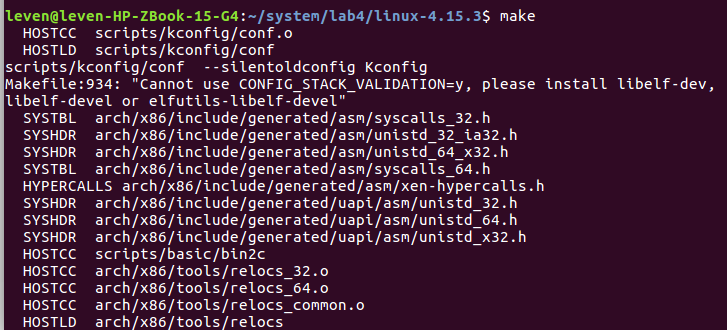
修改sys.c，添加系统调用函数：



清除残留文件 sudo make mrproper成功截图：

****

编译内核 sudo make成功截图



运行测试程序，成功调用新内核自己的系统调用截图：

运行程序

**本校样例5:**

**一、实验目的**

1、掌握系统调用添加方法

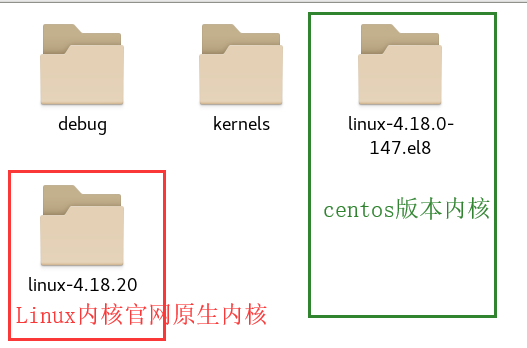
2、重新编译内核，添加系统调用

**二、实验基本原理**

内核本身不是进程，而是进程的管理者，进程/内核模式假定：需要内核服务的进程使用系统调用(system call)机制，每个系统调用都设置了一组参数来识别进程的要求，然后执行与硬件相关的cpu指令完成从用户态到内核态的转换。

**三、实验步骤**

1、下载内核文件，将其解压到/usr/src下。



2、配置系统调用

添加系统调用，在 kernel/sys.c 添加函数

asmlinkage long \_\_x64\_sys\_mycall(long arg)

{

printk("This is \_\_x64\_sys\_mycall %ld",arg);

return arg;

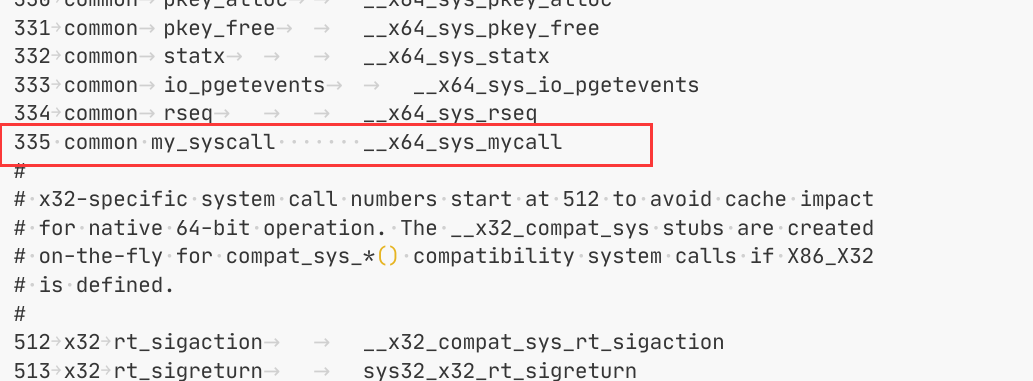
}

添加系统调用声明，在 include/linux/syscalls.h 添加声明

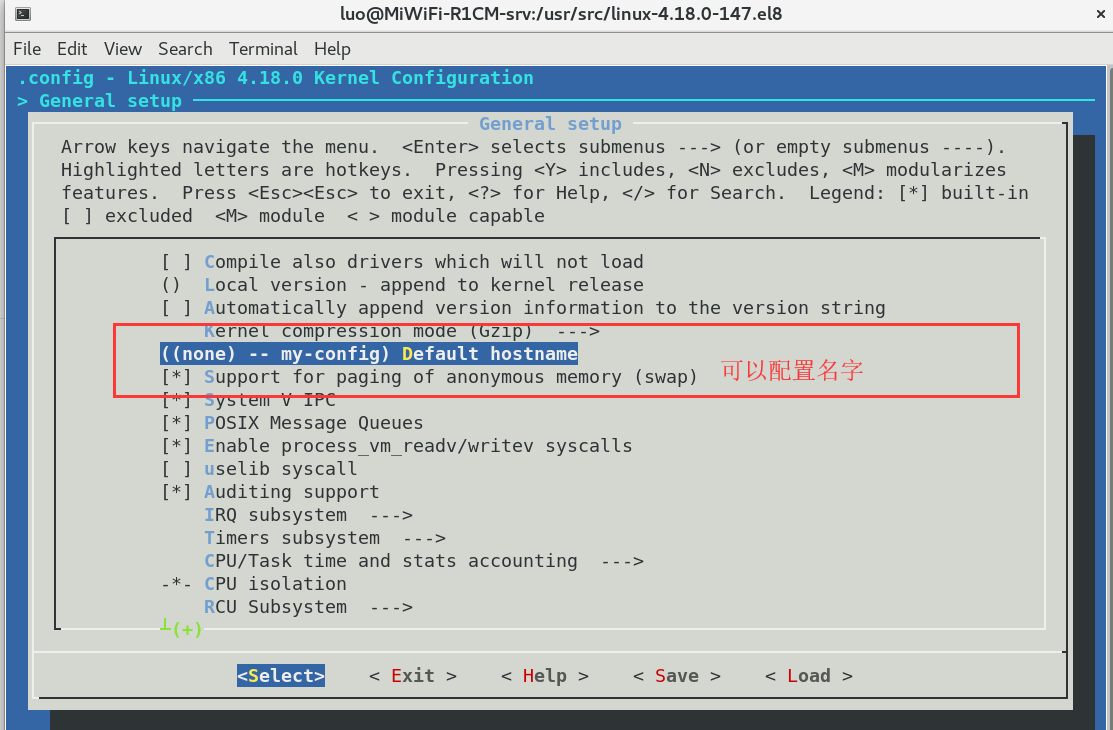
asmlinkage long \_\_x64\_sys\_mycall(long arg);

添加系统调用声明表，在 arch/x86/entry/syscalls/syscall\_64.tbl 添加声明配置

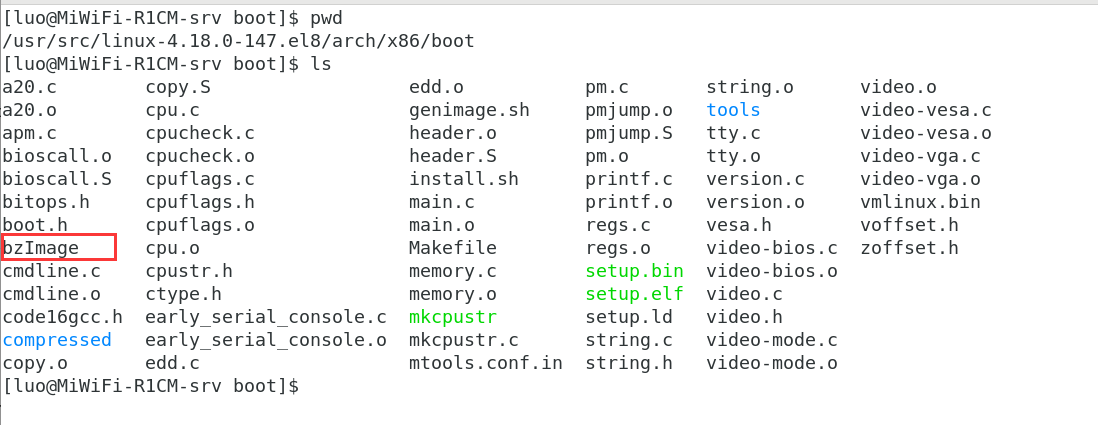
335 common my\_syscall \_\_x64\_sys\_mycall



3、进入内核文件夹，执行 make menuconfig进入配置设置界面，退出并保存。此时会更新文件.config。



4、执行make命令。可以加上参数 -j(数字)，来启用多线程执行命令。过程后会出现bzImage镜像在对应架构文件夹下，例如arch/x86/boot/bzImage。并且会更新系统库函数。

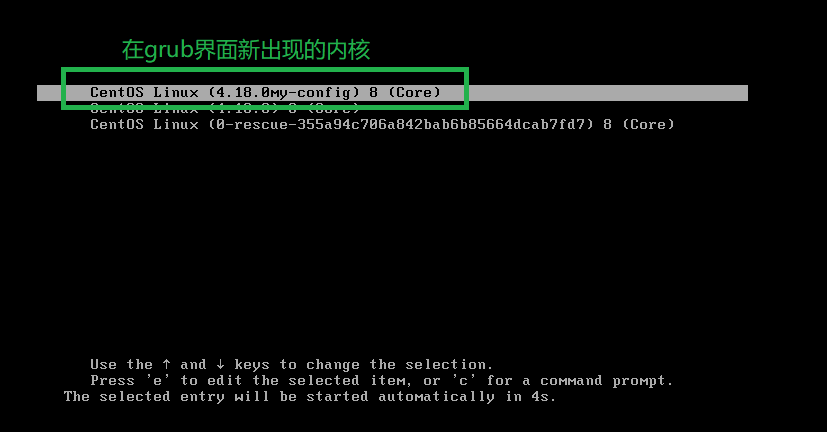




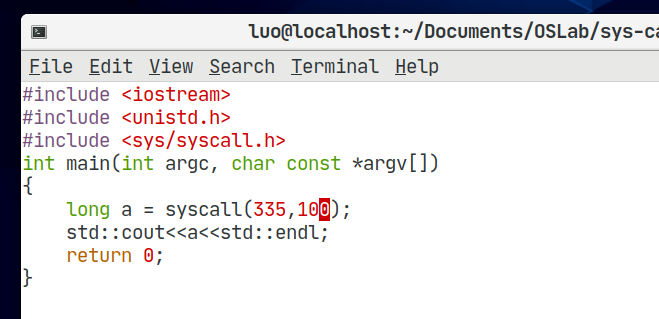
5、执行make modules\_install，将模块生成安装

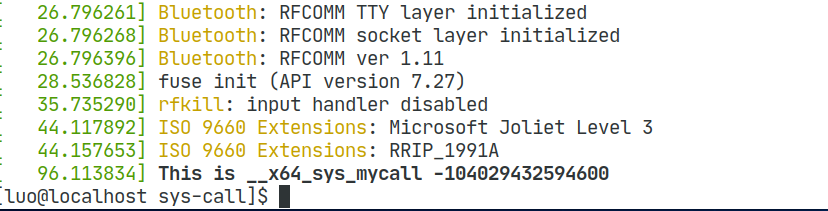
6、执行make install 安装内核镜像。成功后重启系统。不成功需要执行 make mrproper，删除所有配置更新文件，从第3步开始重做。

**四、实验截图与结果分析**



测试代码，335为系统调用表中的调用号，后面的参数是系统调用对应的参数。编译后执行





**其他样例：**

**样例1:**

**下载的内核源代码版本：linux-2.6.29.6**

**Ubuntu 版本：9.0.4**

系统调用是应用程序和操作系统内核之间的功能接口。其主要目的是使得用户可以使用操作系统提供的有关设备管理、输入/输入系统、文件系统和进程控制、通信以及存储管理等方面的功能，而不必了解系统程序的内部结构和有关硬件细节，从而起到减轻用户负担和保护系统以及提高资源利用率的作用。

**（一）Linux系统调用机制**

　　在Linux系统中，系统调用是作为一种异常类型实现的。它将执行相应的机器代码指令来产生异常信号。产生中断或异常的重要效果是系统自动将用户态切换为核心态来对它进行处理。这就是说，执行系统调用异常指令时，自动地将系统切换为核心态，并安排异常处理程序的执行。Linux用来实现系统调用异常的实际指令是： Int $0x80 这一指令使用中断/异常向量号128（即16进制的80）将控制权转移给内核。为达到在使用系统调用时不必用机器指令编程，在标准的C语言库中为每一系统调用提供了一段短的子程序，完成机器代码的编程工作。事实上，机器代码段非常简短。它所要做的工作只是将送给系统调用的参数加载到CPU寄存器中，接着执行int $0x80指令。然后运行系统调用，系统调用的返回值将送入CPU的一个寄存器中，标准的库子程序取得这一返回值，并将它送回用户程序。

　　为使系统调用的执行成为一项简单的任务，Linux提供了一组预处理宏指令。它们可以用在程序中。这些宏指令取一定的参数，然后扩展为调用指定的系统调用的函数。

　　这些宏指令具有类似下面的名称格式：

　　\_syscallN（parameters）

其中N是系统调用所需的参数数目，而parameters则用一组参数代替。这些参数使宏指令完成适合于特定的系统调用的扩展。例如，为了建立调用setuid（）系统调用的函数，应该使用：

　　\_syscall1（ int， setuid， uid\_t， uid ） syscallN（ ）宏指令的第1个参数int说明产生的函数的返回值的类型是整型，第2个参数setuid说明产生的函数的名称。后面是系统调用所需要的每个参数。这一宏指令后面还有两个参数uid\_t和uid分别用来指定参数的类型和名称。

　　另外，用作系统调用的参数的数据类型有一个限制，它们的容量不能超过四个字节。这是因为执行int $0x80指令进行系统调用时，所有的参数值都存在32位的CPU寄存器中。使用CPU寄存器传递参数带来的另一个限制是可以传送给系统调用的参数的数目。这个限制是最多可以传递5个参数。所以Linux一共定义了6个不同的\_syscallN（）宏指令，从\_syscall0（）、\_syscall1（）直到\_syscall5（）。

一旦\_syscallN（）宏指令用特定系统调用的相应参数进行了扩展，得到的结果是一个与系统调用同名的函数，它可以在用户程序中执行这一系统调用。

**（二）添加新的系统调用**

如果用户在Linux中添加新的系统调用，应该遵循几个步骤才能添加成功，下面几个步骤详细说明了添加系统调用的相关内容。

（1） 添加源代码

第一个任务是编写加到内核中的源程序，即将要加到一个内核文件中去的一个函数，该函数的名称应该是新的系统调用名称前面加上sys\_标志。假设新加的系统调用为mysyscall(int number)，在/usr/src/linux-2.6.29.6/kernel/sys.c文件中添加源代码，如下所示：

asmlinkage int sys\_mysyscall(int number)

{

return number;

}

（2） 连接新的系统调用

添加新的系统调用后，下一个任务是使Linux内核的其余部分知道该程序的存在。为了从已有的内核程序中增加到新的函数的连接，需要编辑两个文件。

在我们所用的Linux内核版本（**Ubuntu 9.0.4，内核为2.6.29.6**）中，第一个要修改的文件是：

/usr/src/ linux-2.6.29.6/arch/x86/include/asm /unistd.h

该文件中包含了系统调用清单，用来给每个系统调用分配一个唯一的号码。文件中每一行的格式如下：

#define \_\_NR\_name NNN 其中，name用系统调用名称代替，而NNN则是该系统调用对应的号码。应该将新的系统调用名称加到清单的最后，并给它分配号码序列中下一个可用的系统调用号。我们的系统调用如下：

#define \_\_NR\_mysyscall 333 系统调用号为333，之所以系统调用号是333，是因为Linux-2.6内核自身的系统调用号码已经用到332。

　　第二个要修改的文件是：

/usr/src/ linux-2.6.29.6/arch/x86/kernel 的syscalls\_table\_32.S 文件

在syscalls\_table\_32.S 中添加.long sys\_mysyscall

该文件中有类似如下的清单：

　　.long SYMBOL\_NAME（） 该清单用来对sys\_call\_table[]数组进行初始化。该数组包含指向内核中每个系统调用的指针。这样就在数组中增加了新的内核函数的指针。我们在清单最后添加一行：

.long SYMBOL\_NAME(sys\_mysyscall)

（3） 重建新的Linux内核

　　为使新的系统调用生效，需要重建Linux的内核。这需要以超级用户身份登录。

　　#pwd

　　/usr/src/linux-2.6.29.6

超级用户在当前工作目录（/usr/src/linux-2.6.29.6）下，才可以重建内核。

cd /usr/src/linux-2.6.29.6

make mrproper

make oldconfig

make bzImage

make modules

make modules\_install

make install

编译完毕后，系统生成一可用于安装的、压缩的内核映象文件：/usr/src/linux-2.6.29.6/arch/i386/boot/bzImage

（4）在启动列表中添加新内核镜像

输入命令:cp/usr/src/linux-2.6.29.6/arch/i386/boot/bzImage /boot/bzImage-jzy

cp /usr/src/linux-2.6.29.6/System.map /boot/System.map-jzy

ln –sf /boot/System.map-jzy /boot/System.map

用grub引导内核挂载方法：

在/boot/grub/menu.lst文件中添加如下一段代码：

title Ubuntu jhf

uuid 98b44895-1dd6-46b4-897d-77c01f53c9be(跟程序原来的一样)

kernel /boot/vmlinuz-2.6.29.6

initrd /boot/bzImage-jzy

（5）重启系统,使用新内核引导

（6）自己编写一个程序如下:

#include <linux/unistd.h>

long sys\_mysyscall1(int，mysyscall,int，ret);

int main()

{

printk("My name is oyz,and my number is %d.\n",mysyscall(0));

printk("The last of my number is %d.\n",mysyscall(1));

return 0;

}

（7）输入命令:gcc mhy.c -o mhy

./mhy

**三、实验源程序：**

在/usr/src/linux-2.6.29.6/kernel/sys.c文件中添加源代码，如下所示：

asmlinkage int sys\_mycall(int number)

{

return number;

}

自己编写的测试程序如下：

#include <linux/unistd.h>

long sys\_mysyscall1(int，mysyscall,int，ret);

int main()

{

printk("My name is oyz,and my number is %d.\n",mysyscall(0));

printk("The last of my number is %d.\n",mysyscall(1));

return 0;

}

**四、实验结果：（截图或文本形式）**

**1、内核重建过程中的部分截图：**

make mrproper

显示:

CLEAN /usr/src/ linux-2.6.29.6

CLEAN drivers/scsi/aic7xxx

CLEAN .tmp\_versions

CLEAN scripts/basic

CLEAN scripts/genksyms

CLEAN scripts/kconfig

CLEAN scripts/mod年

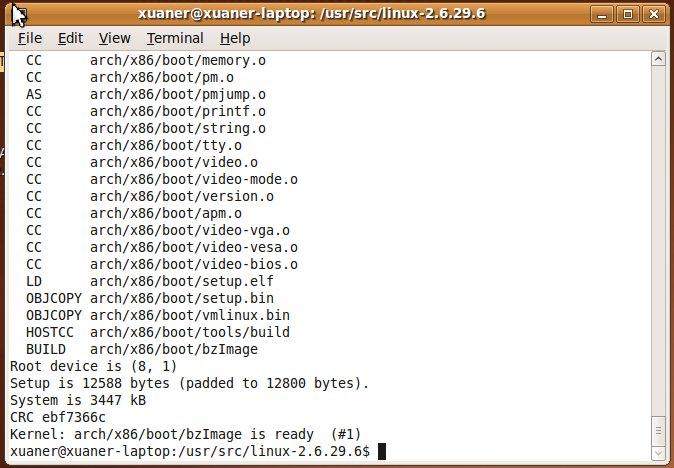
CLEAN scripts

CLEAN include/config

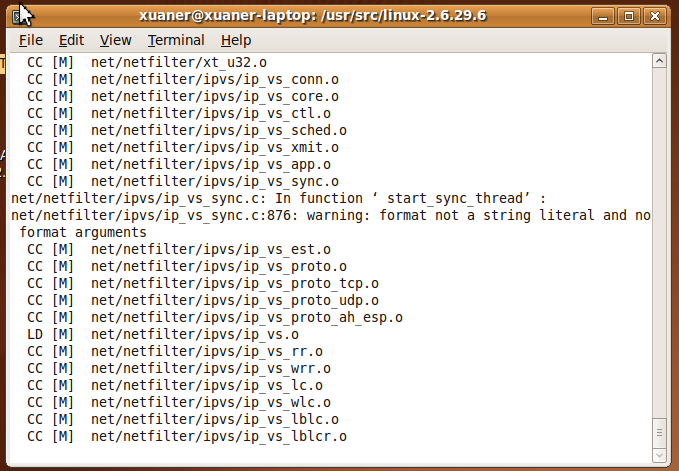
CLEAN .config include/asm include/linux/autoconf.h include/linux/version.h include/linux/utsrelease.h Module.symvers

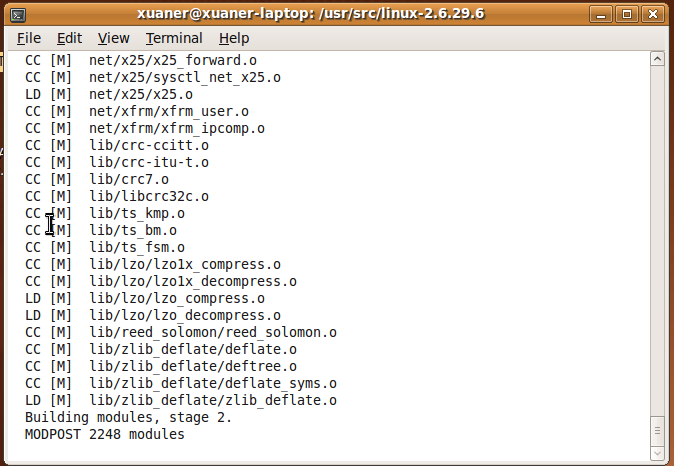
make oldconfig

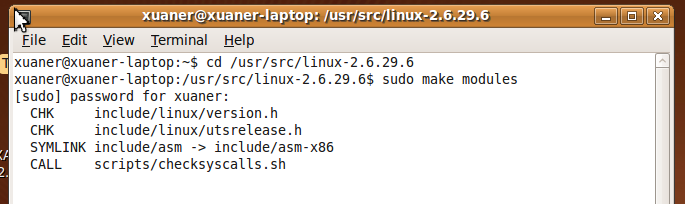
make bzImage

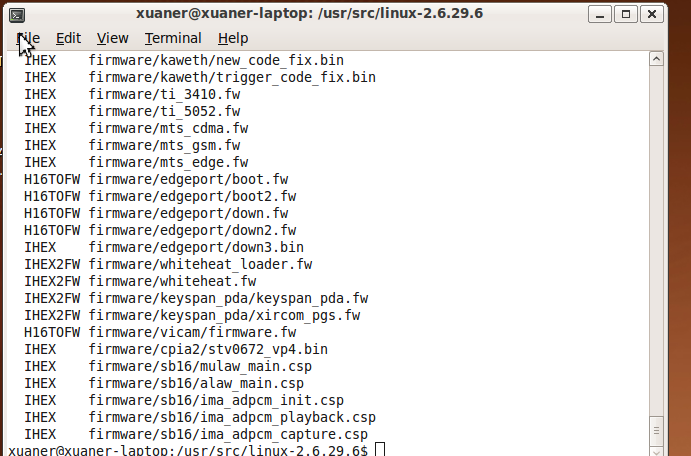


make modules



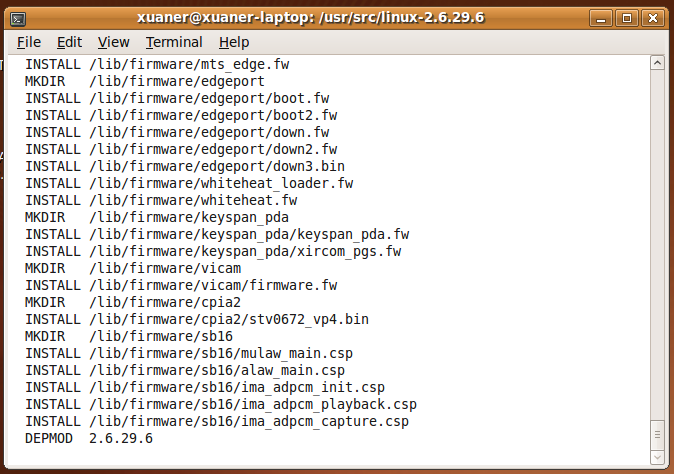




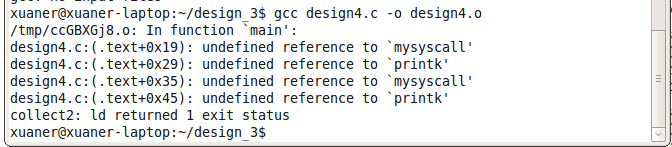


make modules\_install

make install



测试程序编译截图：



结果如图所示，出现错误，原因可能是新编译的内核还是存在问题，相关函数没导入内核。

样例2**（Redhat）**

1.Initrd简介：

initrd是linux在系统引导过程中使用的一个临时的根文件系统，用来支持两阶段的引导过程。

2.system.map简介：

内核符号映射表，顾名思义就是将内核中的符号（也就是内核中的函数）和它的地址能联系起来的一个列表。是所有符号及其对应地址的一个列表。之所以这样就使为了用户编程方便，直接使用函数符号就可以了，而不用去记要使用函数的地址。

当你编译一个新内核时，原来的System.map中的符号信息就不正确了。随着每次内核的编译，就会产生一个新的 System.map文件，并且需要用该文件取代原来的文件。

**四、实验过程：**

A、内核源码解压：

下载 linux-2.6.18.1.tar.gz的代码，放在/usr/src/ 的目录下，然后解压： 解压后会出现文件夹 linux－2.6.18.1。

B、添加代码：

1：在 /usr/src/linux-2.6.18.1/kernel/sys.c文件中添加下面的代码：

asmlinkage int sys\_mycall(int number)

{

return number;

}

2:在/usr/src/linux-2.6.18.1/include/asm-i386/unistd.h中添加系统调用清单：

＃define \_\_NR\_syscalls 319

3: /usr/src/linux-2.6.18.1/arch/i386/kernel/syscall\_table.s中增加新的内核函数的指针：

.long sys\_youcall

C.配置内核:

在 /boot 的目录下找到一个类似 config-2.6.18-1.2798.fc6的配置文件，将他copy到 /usr/src/linux-2.6.18.1/下，并改名为 .config

[root@localhost linux-2.6.18.1]# make menuconfig

然后直接保存退出。

D.编译内核：

[root@localhost linux-2.6.18.1]# make clean //清除以前编译过的.O文件

[root@localhost linux-2.6.18.1]# make bzImage //编译内核

E.模块编译：

[root@localhost linux-2.6.18.1]# make modules //编译选择的模块

[root@localhost linux-2.6.18.1]# make modules\_install //安装模块

F.配置grub:

1： 将/usr/src/linux/arch/i386/boot/bzImage拷贝到/boot/ 目录下

2:制作initrd文件：

[root@localhost boot]# mkinitrd initrd-2.6.18.1.img 2.6.18.1

3: 配置启动文件 /boot/grub/menu.lst,添加下列内容：

title Linuxtest

root (hd0,6)

kernel /boot/bzImage ro root=LABEL=/ rhgb quiet

initrd /boot/initrd-2.6.18.1.img

G.使用自己添加的系统调用：

重启计算机，选择 linuxtest 进入。

编写测试程序，代码如下：

#include <linux/unistd.h>

\_syscall1(int,mycall,int,ret)

main()

{

printf(“%d \n”,mycall(256));

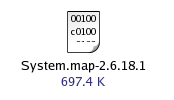
}

**五、实验程序及其结果：**

1.修改好sys.c；unistd.h；syscalls\_table.S；

2.编译内核产生bzImage镜像内核文件、system.map、initrd-2.6.18.1.img。

截图如下：



3．设置menu.lst文件后，增加了内核 Red Hat Linux(2.6.18.1)：

****

样例3.

向系统添加一个自己写的系统调用,并在程序中使用它

(1)下载Linux内核源代码,解压linux-2.6.21.5压缩包得到linux-2.6.21.5文件夹.

(2)以超级用户身分登录,把解压后的inux-2.6.21.5文件夹拷贝到usr/src下.

(3)在终端下输入

vi sys.c 在 sys.c 末尾添加下列代码:

asmlinkage int sys\_mycall(int number)

{

return number;

}

(4)在终端下输入 cd /usr/src/linux\*/include/asm-i386

vi unistd.h 在 unistd.h 中添加#define \_\_NR\_mycall 251 ,添加系统调用类型号;

内核版本在2.4以上, 再输入cd /usr/src/linux\*/arch/i386/kernel

vi syscalls\_table.S

在syscalls\_table.S 中添加.long SYMBOL\_NAME(sys\_mycall)

(5)重新编译内核

执行下面的命令:

cd/usr/src/linux\*

make mrproper

显示:

CLEAN /usr/src/linux-2.6.21.5

CLEAN drivers/scsi/aic7xxx

CLEAN .tmp\_versions

CLEAN scripts/basic

CLEAN scripts/genksyms

CLEAN scripts/kconfig

CLEAN scripts/mod

CLEAN scripts

CLEAN include/config

CLEAN .config include/asm include/linux/autoconf.h include/linux/version.h include/linux/utsrelease.h Module.symvers

make oldconfig

make bzImage

make modules

make modules\_install

make install

(6)在启动列表中添加新内核镜像

输入命令:cp/usr/src/linuxnew/arch/i386/boot/bzImage /boot/bzImage-jzy

cp /usr/src/linux-new/System.map /boot/System.map-jzy

ln –sf /boot/System.map-jzy /boot/System.map

(7)重启系统,使用新内核引导

(8)自己编写一个程序如下:

/\* mhy.c \*/

#include <linux/unistd.h>

\_syscall1(int,mycall,int,ret)

main()

{

printf(“%d \n”,mycall(256));

}

(9)输入命令:gcc mhy.c -o mhy

./mhy

结果为:256

样例4:

A.编写需要添加的系统调用，源代码如下

#include <linux/fcntl.h>

asmlinkage int sys\_wucopy(char \*argv1,char \*argv2){

int newfile,in;

int n,mount=0;

mm\_segment\_t right;

char buffer[32];

if((in=sys\_open(argv1,O\_RDONLY,0))<0){

printk("cannot open infile\n");

sys\_exit(-1);

}

if((newfile=sys\_open(argv2,O\_CREAT|O\_WRONLY|O\_TRUNC,0666))<0){

printk("cannot open outfile\n");

sys\_exit(-1);

}

right=get\_fs();

set\_fs(get\_ds());

while((n=sys\_read(in,buffer,32))>0){

mount+=n;

if(sys\_write(newfile,buffer,n)!=n){

printk("Copy from %s to %s failed\n",argv1,argv2);

sys\_exit(-1);

}

}

set\_fs(right);

sys\_close(in);

sys\_close(newfile);

return mount;

}

B.内核源码解压：

下载 linux-2.6.18.1.tar.gz的代码，放在/usr/src/ 的目录下，然后解压： 解压后会出现文件夹 linux－2.6.18.1。

C.添加代码：

1：在 /usr/src/linux－2.6.18.1/kernel/sys.c文件中添加A中的代码：

2:在/usr/src/linux－2.6.18.1/include/asm-i386/unistd.h中添加系统调用清单：

＃define \_NR\_ wucopy 318

＃define \_\_NR\_syscalls 319

3: /usr/src/linux－2.6.18.1/arch/i386/kernel/syscall\_table.s中增加新的内核函数的指针：

.long sys\_youcall

D.配置内核:

在 /boot 的目录下找到一个类似 config-2.6.18-1.2798.fc6的配置文件，将他copy到 /usr/src/linux-2.6.18.1/下，并改名为 config

[root@localhost linux-2.6.18.1]# make menuconfig

然后直接保存退出。

E.编译内核：

[root@localhost linux-2.6.18.1]# make clean //清除以前编译过的.0文件

[root@localhost linux-2.6.18.1]# make bzImage //编译内核

F.模块编译：

[root@localhost linux-2.6.18.1]# make modules //编译选择的模块

[root@localhost linux-2.6.18.1]# make modules\_install //安装模块

G.配置grub:

1： 将/usr/src/linux/arch/i386/boot/bzImage拷贝到/boot/ 目录下

2:制作initrd文件：

[root@localhost boot]# mkinitrd initrd-2.6.18.1.img 2.6.18.1

3: 配置启动文件 /boot/grub/menu.lst,添加下列内容：

title Linuxtest

root (hd0,6)

kernel /boot/bzImage ro root=LABEL=/ rhgb quiet

initrd /boot/initrd-2.6.18.1.img

H.使用自己添加的系统调用：

重启计算机，选择 linuxtest 进入。

编写测试程序，代码如下：

#include<stdio.h>

#include</usr/src/linux-2.6.18.1/include/asm/unistd.h>

#include<errno.h>

#include<sys/syscall.h>

int main(void){

char infile[30],outfile[30]="knew\_file";

printf("Enter the file name needed to be copied\n");

scanf("%s",infile);

if(syscall(318,infile,outfile)){

printf("success!\n");

printf("The newfile name is knew\_file\n");

}

else

printf("failed");

return 0;

样例5:

<http://topic.csdn.net/u/20090318/09/5ae9ec6e-703a-4d8e-b214-ecd8d9467b68.html>

unbuntu添加系统调用

样例6:

Redhat添加系统调用

http://www.chinalinuxpub.com/read.php?wid=23

1 Linux系统调用机制  
　　在Linux系统中，系统调用是作为一种异常类型实现的。它将执行相应的机器代码指令来产生异常信号。产生中断或异常的重要效果是系统自动将用户态切换为核心态来对它进行处理。这就是说，执行系统调用异常指令时，自动地将系统切换为核心态，并安排异常处理程序的执行。  
　　Linux用来实现系统调用异常的实际指令是：  
　　Int $0x80   
　　这一指令使用中断/异常向量号128（即16进制的80）将控制权转移给内核。为达到在使用系统调用时不必用机器指令编程，在标准的C语言库中为每一系统调用提供了一段短的子程序，完成机器代码的编程工作。事实上，机器代码段非常简短。它所要做的工作只是将送给系统调用的参数加载到CPU寄存器中，接着执行int $0x80指令。然后运行系统调用，系统调用的返回值将送入CPU的一个寄存器中，标准的库子程序取得这一返回值，并将它送回用户程序。  
　　为使系统调用的执行成为一项简单的任务，Linux提供了一组预处理宏指令。它们可以用在程序中。这些宏指令取一定的参数，然后扩展为调用指定的系统调用的函数。  
　　这些宏指令具有类似下面的名称格式：  
　　\_syscallN（parameters）  
　　其中N是系统调用所需的参数数目，而parameters则用一组参数代替。这些参数使宏指令完成适合于特定的系统调用的扩展。例如，为了建立调用setuid（）系统调用的函数，应该使用：  
　　\_syscall1（ int， setuid， uid\_t， uid ）  
　　syscallN（ ）宏指令的第1个参数int说明产生的函数的返回值的类型是整型，第2个参数setuid说明产生的函数的名称。后面是系统调用所需要的每个参数。这一宏指令后面还有两个参数uid\_t和uid分别用来指定参数的类型和名称。  
　　另外，用作系统调用的参数的数据类型有一个限制，它们的容量不能超过四个字节。这是因为执行int $0x80指令进行系统调用时，所有的参数值都存在32位的 CPU寄存器中。使用CPU寄存器传递参数带来的另一个限制是可以传送给系统调用的参数的数目。这个限制是最多可以传递5个参数。所以Linux一共定义了6个不同的\_syscallN（）宏指令，从\_syscall0（）、\_syscall1（）直到\_syscall5（）。  
　　一旦\_syscallN（）宏指令用特定系统调用的相应参数进行了扩展，得到的结果是一个与系统调用同名的函数，它可以在用户程序中执行这一系统调用。  
2 添加新的系统调用  
 　如果用户在Linux中添加新的系统调用，应该遵循几个步骤才能添加成功，下面几个步骤详细说明了添加系统调用的相关内容。  
（1） 添加源代码  
　　第一个任务是编写加到内核中的源程序，即将要加到一个内核文件中去的一个函数，该函数的名称应该是新的系统调用名称前面加上sys\_标志。假设新加的系统  
调用为mycall(int number)，在/usr/src/linux/kernel/sys.c文件中添加源代码，如下所示：  
　　asmlinkage int sys\_mycall(int number)  
　　{  
　　return number;  
　　}  
　　作为一个最简单的例子，我们新加的系统调用仅仅返回一个整型值。  
（2） 连接新的系统调用  
　　添加新的系统调用后，下一个任务是使Linux内核的其余部分知道该程序的存在。为了从已有的内核程序中增加到新的函数的连接，需要编辑两个文件。  
　　在我们所用的Linux内核版本（RedHat 6.0，内核为2.2.5-15）中，第一个要修改的文件是：  
　　/usr/src/linux/include/asm-i386/unistd.h  
　　该文件中包含了系统调用清单，用来给每个系统调用分配一个唯一的号码。文件中每一行的格式如下：  
　　#define \_\_NR\_name NNN  
　　其中，name用系统调用名称代替，而NNN则是该系统调用对应的号码。应该将新的系统调用名称加到清单的最后，并给它分配号码序列中下一个可用的系统调用号。我们的系统调用如下：  
　　#define \_\_NR\_mycall 191  
　　系统调用号为191，之所以系统调用号是191，是因为Linux-2.2内核自身的系统调用号码已经用到190。  
　　第二个要修改的文件是：  
　　/usr/src/linux/arch/i386/kernel/entry.S  
　　该文件中有类似如下的清单：  
　　.long SYMBOL\_NAME（）  
　　该清单用来对sys\_call\_table[]数组进行初始化。该数组包含指向内核中每个系统调用的指针。这样就在数组中增加了新的内核函数的指针。我们在清单最后添加一行：  
　　.long SYMBOL\_NAME(sys\_mycall)  
（3） 重建新的Linux内核  
　　为使新的系统调用生效，需要重建Linux的内核。这需要以超级用户身份登录。  
　　#pwd  
　　/usr/src/linux  
　　#  
　　超级用户在当前工作目录（/usr/src/linux）下，才可以重建内核。  
 　#make config  
　　#make dep  
　　#make clearn  
　　#make bzImage  
　　编译完毕后，系统生成一可用于安装的、压缩的内核映象文件：  
　　/usr/src/linux/arch/i386/boot/bzImage　  
（4） 用新的内核启动系统  
 要使用新的系统调用，需要用重建的新内核重新引导系统。为此，需要修改/etc/lilo.conf文件，在我们的系统中，该文件内容如下：  
　　boot=/dev/hda  
　　map=/boot/map  
　　install=/boot/boot.b  
　　prompt  
　　timeout=50  
  
　　image=/boot/vmlinuz-2.2.5-15  
　　label=linux  
　　root=/dev/hdb1  
　read-only  
  
　　other=/dev/hda1  
　　label=dos  
　　table=/dev/had  
　　首先编辑该文件，添加新的引导内核：  
 　image=/boot/bzImage-new  
　　label=linux-new  
　　root=/dev/hdb1  
　　read-only  
　　添加完毕，该文件内容如下所示：  
　　boot=/dev/hda  
　　map=/boot/map  
　　install=/boot/boot.b  
　　prompt  
　　timeout=50  
  
　　image=/boot/bzImage-new  
　　label=linux-new  
　　root=/dev/hdb1  
　　read-only  
  
　　image=/boot/vmlinuz-2.2.5-15  
　　label=linux  
　　root=/dev/hdb1  
　　read-only  
  
　　other=/dev/hda1  
　　label=dos  
　　table=/dev/hda  
　　这样，新的内核映象bzImage-new成为缺省的引导内核。  
　　为了使用新的lilo.conf配置文件，还应执行下面的命令：  
　　#cp /usr/src/linux/arch/i386/boot/zImage /boot/bzImage-new  
　　其次配置lilo:  
　　# /sbin/lilo  
　　现在，当重新引导系统时，在boot:提示符后面有三种选择：linux-new 、  
linux、dos，新内核成为缺省的引导内核。  
　　至此，新的Linux内核已经建立，新添加的系统调用已成为操作系统的一部分，重新启动Linux，用户就可以在应用程序中使用该系统调用了。  
（5）使用新的系统调用  
　　在应用程序中使用新添加的系统调用mycall。同样为实验目的，我们写了一个简单的例子xtdy.c。  
　　/\* xtdy.c \*/  
　　#include  
　　\_syscall1(int,mycall,int,ret)  
　　main()  
　　{  
　　printf("%d n",mycall(100));  
　　}  
　　编译该程序：  
　　# cc -o xtdy xtdy.c  
　　执行：  
　　# xtdy  
　　结果：  
　　# 100  
　　注意，由于使用了系统调用，编译和执行程序时，用户都应该是超级用户身份