操作系统课程设计报告

**专业班级：计算机科学与技术1501班**

**学 号： U201514477**

**姓 名： 华龙**

**指导教师： 阳富民**

**报告日期： 2018年3月28日**

**目 录**

[1 课程设计目的 1](#_Toc510666116)

[2 环境搭建 1](#_Toc510666117)

[2.1 实验平台 1](#_Toc510666118)

[2.2 gcc编译环境搭建 1](#_Toc510666119)

[2.3 gtk编程环境搭建 1](#_Toc510666120)

[2.4 linux内核编译环境搭建 1](#_Toc510666121)

[3 实验一：熟悉Linux编程环境 2](#_Toc510666122)

[3.1 实验目的 2](#_Toc510666123)

[3.2 实验内容 2](#_Toc510666124)

[3.3 实验设计 2](#_Toc510666125)

[3.4 程序清单 2](#_Toc510666126)

[3.5 实验步骤 14](#_Toc510666127)

[3.6 调试记录及运行结果 15](#_Toc510666128)

[4 实验二：增加系统功能调用 16](#_Toc510666129)

[4.1 实验目的 16](#_Toc510666130)

[4.2 实验内容 16](#_Toc510666131)

[4.3 程序清单 16](#_Toc510666132)

[4.4 实验步骤 17](#_Toc510666133)

[4.5 调试记录及运行结果 19](#_Toc510666134)

[5 实验三：添加设备驱动程序 20](#_Toc510666135)

[5.1 实验目的 20](#_Toc510666136)

[5.2 实验内容 20](#_Toc510666137)

[5.3 实验设计 20](#_Toc510666138)

[5.4 程序清单 20](#_Toc510666139)

[5.5 实验步骤 25](#_Toc510666140)

[6 实验四：Proc文件系统的使用 26](#_Toc510666141)

[6.1 实验目的 26](#_Toc510666142)

[6.2 实验内容 26](#_Toc510666143)

[6.3 实验设计 27](#_Toc510666144)

[6.4 程序清单 27](#_Toc510666145)

[6.5 实验步骤 27](#_Toc510666146)

[6.6 调试记录及运行结果 27](#_Toc510666147)

# 课程设计目的

掌握Linux操作系统的使用方法；

了解Linux系统内核代码结构；

掌握实例操作系统的实现方法。

# 环境搭建

## 实验平台

操作系统：ubuntu-16.04

内核版本：kernal 4.10

内存大小：6G

## gcc编译环境搭建

Ubuntu16.04系统自带了gedit文本编辑器，所以只需要安装gcc编译器即可，打开终端，输入如下命令安装gcc：

sudo apt-get install gcc

## gtk编程环境搭建

安装好gcc编译器后，想要开发gtk图形界面程序，还需安装gtk开发环境，打开终端，输入如下命令安装gtk开发环境：

sudo apt-get install libgtk-3-dev

## linux内核编译环境搭建

1. 打开终端，更新软件源代码：

sudo apt-get install update

2. 安装编译需要的软件和依赖包：

sudo apt-get install libncurses5-dev libssl-dev

sudo apt-get install build-essential openssl

sudo apt-get install zlibc minizip

sudo apt-get install libidn11-dev libidn11

# 实验一：熟悉Linux编程环境

## 实验目的

掌握Linux操作系统的使用方法，包括键盘命令、系统调用；

掌握在Linux下的编程环境。

## 实验内容

1. 编一个C程序，其内容为实现文件拷贝的功能(使用系统调用open/read/write..)；

2. 编一个C程序，其内容为分窗口同时显示三个并发进程的运行结果。要求用到Linux下的图形库(gtk/Qt)，如三个进程实现誊抄的演示。

## 实验设计

### 文件拷贝程序：

使用最基本的文件IO系统功能调用open、read、write、close函数实现文件的读写，循环把一段数据读到一个缓存区再写入副本文件，直到读到文件尾（read函数返回0）。具体实现见3.4.1小节的程序清单。

### 多进程文件拷贝程序：

1. 程序一共运行三个进程，并且三个进程各自创建一个图形窗口并显示；

2. 主进程完成源文件的选择和副本文件的文件名设置以及开始拷贝程序；

3. 子进程1将文件中的数据读入环形缓冲区，并在图形窗口中显示读数据的进程；

4. 子进程2将环形缓冲区的数据写入副本文件，并在图形窗口中显示写入进程；

5. 两个子进程都拥有两个线程，主线程处理用户界面，子线程负责文件的读取和写入；

6. 通过信号灯实现两个子进程之间的同步，使用生产者-消费者模型；

具体实现见3.4.2小节的程序清单。

## 程序清单

**1. 文件拷贝程序filecopy.c**

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <errno.h>

#include <string.h>

int filecopy(char \*sname, char \*dname)

{

int fd\_s;

int fd\_d;

char buf[900]; //缓存区

int sizecopid;

//打开源文件

if (-1 == (fd\_s = open(sname, O\_RDONLY, S\_IRWXU | S\_IRWXO | S\_IRWXG)))

{

printf("failed to open %s\n", sname);

return -1;

}

//创建副本文件

if (-1 == (fd\_d = open(dname, O\_CREAT | O\_RDWR, S\_IRWXU | S\_IRWXO | S\_IRWXG)))

{

printf("failed to create %s\n", dname);

return -1;

}

//循环从源文件读取数据和向副本文件写数据

while (1)

{

if (-1 == (sizecopid = read(fd\_s, buf, 900)))

{

printf("failed to read %s\n", sname);

return -1;

}

//如果到达文件尾则跳出循环

if (sizecopid <= 0)

break;

if (-1 == write(fd\_d, buf, sizecopid))

{

printf("failed to write %s:%s\n", dname, strerror(errno));

return -1;

}

//输出拷贝日志信息

printf("copied %d bytes\n", sizecopid);

}

close(fd\_s);

close(fd\_d);

return 0;

}

int main(int argc, char \*const argv[])

{

if (argc != 3)

{

printf("argc != 3 !\n");

return -1;

}

filecopy(argv[1], argv[2]);

printf("file copied!\n");

return 0;

}

**2. 多进程拷贝程序lab1.c**

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <unistd.h>

#include <semaphore.h>

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <string.h>

#include <fcntl.h>

#include <gtk/gtk.h>

#define BUFNUM 5 //缓存区数量

#define BUFSIZE 100 //缓存区大小

void openfilechoosedialog(GtkWidget \*widget, gpointer data);

void set\_s\_name(GtkWidget \*widget, gpointer data);

void start\_copy(GtkWidget \*widget, gpointer data);

void update\_read\_buffer(GtkWidget \*widget, gpointer data);

void update\_write\_buffer(GtkWidget \*widget, gpointer data);

void \*read\_thread(void \*);

void \*write\_thread(void \*);

gboolean set\_read\_bar(gpointer data);

gboolean set\_write\_bar(gpointer data);

typedef struct buffer

{ //缓存区结构

int size; //有效字节数

char buf[BUFSIZE]; //数据缓存区

} BUFFER;

BUFFER \*bufs;

sem\_t \*full, \*empty, \*s1, \*s2;

char \*s\_name, \*d\_name;

pid\_t pid1 = -1, pid2 = -1;

long \*pfilesize;

key\_t shm\_key = (key\_t)14477; //申请共享内存用的键值

//窗口控件指针

GtkWidget \*window1;

GtkWidget \*button\_openfile;

GtkWidget \*dialog\_openfile;

GtkWidget \*entry\_save;

GtkWidget \*button\_start;

GtkWidget \*window2;

GtkWidget \*textview1;

GtkWidget \*progressbar1;

GtkWidget \*window3;

GtkWidget \*textview2;

GtkWidget \*progressbar2;

GtkTextBuffer \*buffer\_read;

GtkTextBuffer \*buffer\_write;

GtkTextIter \*read\_buffer\_end;

GtkTextIter \*write\_buffer\_end;

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

{

//以下是申请共享内存

int segment\_id;

char \*shard\_memory;

printf("%d,%d\n", sizeof(sem\_t), sizeof(BUFFER) \* BUFNUM);

if (-1 == (segment\_id = shmget(shm\_key, sizeof(sem\_t) \* 4 + sizeof(BUFFER) \* BUFNUM + 104, IPC\_CREAT | 0666)))

{ //申请共享内存

perror("shmget error at lab3");

exit(-1);

}

if (-1 == (int)(shard\_memory = (char \*)shmat(segment\_id, 0, 0)))

{ //映射共享内存

printf("\n shmat error in lab3 : %s\n", strerror(errno));

exit(-1);

}

full = (sem\_t \*)shard\_memory; //映射信号灯在共享内存中

empty = (sem\_t \*)(shard\_memory + sizeof(sem\_t));

s1 = (sem\_t \*)(shard\_memory + 2 \* sizeof(sem\_t));

s2 = (sem\_t \*)(shard\_memory + 3 \* sizeof(sem\_t));

sem\_init(full, 1, 0); //初始化信号灯,满缓存区数目

sem\_init(empty, 1, BUFNUM); //空缓存区数目；

sem\_init(s1, 1, 0); //控制子进程1能否继续运行

sem\_init(s2, 1, 0); //控制子进程2能否继续运行

bufs = (BUFFER \*)(shard\_memory + 4 \* sizeof(sem\_t)); //映射缓存区在共享内存中

s\_name = (char \*)(shard\_memory + 4 \* sizeof(sem\_t) + sizeof(BUFFER) \* BUFNUM);

d\_name = (char \*)(shard\_memory + 4 \* sizeof(sem\_t) + sizeof(BUFFER) \* BUFNUM + 50);

pfilesize = (long \*)(shard\_memory + 4 \* sizeof(sem\_t) + sizeof(BUFFER) \* BUFNUM + 100);

//子进程1

pid1 = fork();

if (pid1 == 0)

{

printf("entered read process\n");

//1.gtk初始化

gtk\_init(&argc, &argv);

//2.创建GtkBuilder对象，GtkBuilder在<gtk/gtk.h>声明

GtkBuilder \*builder\_read = gtk\_builder\_new();

//3.读取test.glade文件的信息，保存在builder中

if (!gtk\_builder\_add\_from\_file(builder\_read, "reader.glade", NULL)){

printf("connot load file!");

}

//4.获取窗口指针，注意"window1"要和glade里面的标签名词匹配

window2 = GTK\_WIDGET(gtk\_builder\_get\_object(builder\_read, "window2"));

textview1 = GTK\_WIDGET(gtk\_builder\_get\_object(builder\_read, "textview1"));

progressbar1 = GTK\_WIDGET(gtk\_builder\_get\_object(builder\_read, "progressbar1"));

printf("progressbar1:%p\n", progressbar1);

//获取textview的buffer

buffer\_read = gtk\_text\_view\_get\_buffer((GtkTextView \*)textview1);

g\_signal\_connect(G\_OBJECT(window2), "destroy", G\_CALLBACK(gtk\_main\_quit), NULL);

gtk\_widget\_show\_all(window2);

g\_thread\_new("reader", &read\_thread, NULL); //创建读线程

gtk\_main();

g\_object\_unref(G\_OBJECT(builder\_read)); //释放GtkBuilder对象

printf("read process exited\n");

return 0;

}

//子进程2

pid2 = fork();

if (pid2 == 0)

{

printf("entered write process\n");

//1.gtk初始化

gtk\_init(&argc, &argv);

//2.创建GtkBuilder对象，GtkBuilder在<gtk/gtk.h>声明

GtkBuilder \*builder\_write = gtk\_builder\_new();

//3.读取test.glade文件的信息，保存在builder中

if (!gtk\_builder\_add\_from\_file(builder\_write, "writer.glade", NULL)){

printf("connot load file!");

}

//4.获取窗口指针，注意"window1"要和glade里面的标签名词匹配

window3 = GTK\_WIDGET(gtk\_builder\_get\_object(builder\_write, "window3"));

textview2 = GTK\_WIDGET(gtk\_builder\_get\_object(builder\_write, "textview1"));

progressbar2 = GTK\_WIDGET(gtk\_builder\_get\_object(builder\_write, "progressbar2"));

printf("progressbar2:%p\n", progressbar2);

//获取textview的buffer

buffer\_write = gtk\_text\_view\_get\_buffer((GtkTextView \*)textview2);

//定义一个信号

g\_signal\_connect(G\_OBJECT(window3), "destroy", G\_CALLBACK(gtk\_main\_quit), NULL);

gtk\_widget\_show\_all(window3);

g\_thread\_new("writer", &write\_thread, NULL); //创建写线程

gtk\_main();

g\_object\_unref(G\_OBJECT(builder\_write)); //释放GtkBuilder对象

printf("write process exited\n");

return 0;

}

//1.gtk初始化

gtk\_init(&argc, &argv);

//2.创建GtkBuilder对象，GtkBuilder在<gtk/gtk.h>声明

GtkBuilder \*builder\_main = gtk\_builder\_new();

//3.读取test.glade文件的信息，保存在builder中

if (!gtk\_builder\_add\_from\_file(builder\_main, "chooser.glade", NULL)){

printf("connot load file!");

}

//4.获取窗口指针，注意"window1"要和glade里面的标签名词匹配

window1 = GTK\_WIDGET(gtk\_builder\_get\_object(builder\_main, "window1"));

button\_openfile = GTK\_WIDGET(gtk\_builder\_get\_object(builder\_main, "button1"));

entry\_save = GTK\_WIDGET(gtk\_builder\_get\_object(builder\_main, "savefilename"));

button\_start = GTK\_WIDGET(gtk\_builder\_get\_object(builder\_main, "start"));

printf("entry\_save:%p\n", entry\_save);

g\_signal\_connect(G\_OBJECT(button\_openfile), "clicked", G\_CALLBACK(openfilechoosedialog), NULL);

g\_signal\_connect(G\_OBJECT(button\_start), "clicked", G\_CALLBACK(start\_copy), NULL);

g\_signal\_connect(G\_OBJECT(window1), "destroy", G\_CALLBACK(gtk\_main\_quit), NULL);

gtk\_main();

g\_object\_unref(G\_OBJECT(builder\_main)); //释放GtkBuilder对象

printf("main process exited\n");

kill(pid1, 0); //结束两个子进程

kill(pid2, 0);

sem\_destroy(full); //删除信号灯；

sem\_destroy(empty);

sem\_destroy(s1); //删除信号灯；

sem\_destroy(s2);

shmctl(segment\_id, IPC\_RMID, 0); //释放共享内存

return 0;

}

void openfilechoosedialog(GtkWidget \*widget, gpointer data)

{

printf("openfilechoosedialog called\n");

GtkWidget \*dialog;

GtkFileChooserAction action = GTK\_FILE\_CHOOSER\_ACTION\_OPEN;

gint res;

dialog = gtk\_file\_chooser\_dialog\_new("Open File",

(GtkWindow \*)window1,

action,

"Cancel",

GTK\_RESPONSE\_CANCEL,

"Open",

GTK\_RESPONSE\_ACCEPT,

NULL);

res = gtk\_dialog\_run(GTK\_DIALOG(dialog));

if (res == GTK\_RESPONSE\_ACCEPT){

char \*filename;

GtkFileChooser \*chooser = GTK\_FILE\_CHOOSER(dialog);

filename = gtk\_file\_chooser\_get\_filename(chooser);

if (filename)

strcpy(s\_name, filename);

printf("opened file:%s\n", filename);

g\_free(filename);

}

gtk\_widget\_destroy(dialog);

return;

}

void start\_copy(GtkWidget \*widget, gpointer data)

{

printf("start\_copy called\n");

char buf[30] = "";

printf("entry:%p\n", entry\_save);

if (0 < gtk\_entry\_get\_text\_length((GtkEntry \*)entry\_save))

strcpy(d\_name, gtk\_entry\_get\_text((GtkEntry \*)entry\_save));

else

strcpy(d\_name, "Untitled");

printf("source file:%s\ndest file:%s\n", s\_name, d\_name);

struct stat statbuf;

if (-1 == stat(s\_name, &statbuf))

perror("stat");

\*pfilesize = statbuf.st\_size;

printf("filesize:%ld\n", \*pfilesize);

sem\_post(s1);

sem\_post(s2);

return;

}

gboolean updateread(gpointer data)

{

GtkTextIter read\_buffer\_end;

gtk\_text\_buffer\_get\_end\_iter(buffer\_read, &read\_buffer\_end);

gtk\_text\_buffer\_insert(buffer\_read, &read\_buffer\_end, (char \*)data, -1);

return FALSE;

}

gboolean updatewrite(gpointer data)

{

GtkTextIter write\_buffer\_end;

gtk\_text\_buffer\_get\_end\_iter(buffer\_write, &write\_buffer\_end);

gtk\_text\_buffer\_insert(buffer\_write, &write\_buffer\_end, (char \*)data, -1);

g\_free(data);

return FALSE;

}

void \*read\_thread(void \*data)

{

printf("read\_thread waiting for s1\n");

sem\_wait(s1);

printf("read\_thread started\n");

char \*message;

int fd;

double bytecounts = 0;

message = g\_new0(char, 100);

sprintf(message, "read process started.\n");

gdk\_threads\_add\_timeout(0, updateread, message);

if (-1 == (fd = open(s\_name, O\_RDONLY))){

printf("failed to open src\_file:%s\n", strerror(errno));

exit(-1);

}

else

printf("openfilename : %s\n", s\_name);

int buf\_index = 0;

BUFFER readbuf;

while (1){

sleep(1); //为观察程序执行过程而设置每秒拷贝一段数据

int sizeread = readbuf.size = read(fd, readbuf.buf, BUFSIZE);//从文件中读数据

printf("read %d bytes.\n", sizeread);

message = g\_new0(char, 100);

sprintf(message, "read %d bytes.\n", sizeread);

gdk\_threads\_add\_timeout(0, updateread, message);

if (sizeread < 0)

printf("read error %d:%s \n", errno, strerror(errno));

if (readbuf.size <= 0) //如果读到最后一块数据跳出循环

break;

buf\_index = buf\_index % BUFNUM; //缓存区索引，根据缓存区数量循环

sem\_wait(empty);//向缓存区存数据

memcpy((void \*)&bufs[buf\_index], (void \*)&readbuf, sizeof(BUFFER));

sem\_post(full);

bytecounts += sizeread;

double \*d\_read = g\_new0(double, 1);

\*d\_read = bytecounts;

gdk\_threads\_add\_timeout(0, set\_read\_bar, d\_read);

buf\_index++;

}

close(fd);

printf("read completed.\n");

message = g\_new0(char, 100);

sprintf(message, "read completed.\n");

gdk\_threads\_add\_timeout(0, updateread, message);

return 0;

}

void \*write\_thread(void \*data)

{

window3;

printf("write\_thread waiting for s2\n");

sem\_wait(s2);

printf("write\_thread started\n");

char \*message;

int fd;

double bytecounts = 0.0;

message = g\_new0(char, 100);

sprintf(message, "write process started.\n");

gdk\_threads\_add\_timeout(0, updatewrite, message);

if (-1 == (fd = open(d\_name, O\_CREAT | O\_RDWR, S\_IRWXU | S\_IRWXO | S\_IRWXG))){

printf("failed to create dest\_file:%s\n", strerror(errno));

exit(-1);

}

else

printf("savefilename : %s\n", d\_name);

int buf\_index = 0;

BUFFER writebuf;

while (1){

sleep(1); //为观察程序执行过程而设置每秒拷贝一段数据

buf\_index = buf\_index % BUFNUM; //缓存区索引，根据缓存区数量循环

sem\_wait(full); //从缓存区取数据

memcpy((void \*)&writebuf, (void \*)&bufs[buf\_index], sizeof(BUFFER));

sem\_post(empty);

int sizewrited = write(fd, writebuf.buf, writebuf.size); //向文件中写数据

printf("writed %d bytes.\n", sizewrited);

message = g\_new0(char, 100);

sprintf(message, "writed %d bytes.\n", sizewrited);

gdk\_threads\_add\_timeout(0, updatewrite, message);

if (sizewrited < 0)

printf("write error %d:%s \n", errno, strerror(errno));

if (writebuf.size < BUFSIZE) //如果读到最后一块数据跳出循环

break;

bytecounts += sizewrited;

double \*d\_write = g\_new0(double, 1);

\*d\_write = bytecounts;

gdk\_threads\_add\_timeout(0, set\_write\_bar, (gpointer)d\_write);

buf\_index++;

}

close(fd);

printf("write completed.\n");

message = g\_new0(char, 100);

sprintf(message, "write completed.\n");

gdk\_threads\_add\_timeout(0, updatewrite, message);

return 0;

}

gboolean set\_read\_bar(gpointer data)

{

double \*done = data;

double progress = \*done / (double)\*pfilesize;

gtk\_progress\_bar\_set\_fraction((GtkProgressBar \*)progressbar1, progress);

g\_free(data);

return FALSE;

}

gboolean set\_write\_bar(gpointer data)

{

double \*done = data;

double progress = \*done / (double)\*pfilesize;

gtk\_progress\_bar\_set\_fraction((GtkProgressBar \*)progressbar2, progress);

g\_free(data);

return FALSE;

}

## 实验步骤

### 文件拷贝程序

1. 编译程序，在终端中执行命令 gcc filecopy.c –o filecopy ；

2. 测试程序，在终端中执行命令 ./filecopy filecopy.c filecopy.backup ，其中程序的第一个参数为源文件的文件名，第二个参数为副本文件的文件名；

3. 验证测试是否正确，执行命令 cmp filecopy.c filecopy.backup ，如果无任何输出，说明两个文件无差别，即文件拷贝程序功能正确。

### 多进程拷贝程序

1. 编译程序，在终端中执行 gcc lab1.c –o lab1 –pthread `pkg-config --cflags --libs gtk+-3.0` ；

2. 运行程序，点击选择文件按钮，选择要拷贝的源文件，在文本输入框中输入副本文件的文件名，然后点击开始按钮开始拷贝文件；

3. 比较源文件和副本文件的差异，验证程序正确性。

## 调试记录及运行结果

### 文件拷贝程序

按照实验步骤测试，程序的运行结果如图3.1所示。从结果可以看出，程序功能正确。

图3.1 文件拷贝程序运行结果

### 多进程拷贝程序

1. 按照实验步骤运行程序，程序拷贝过程显示如图3.2所示。从图中可看到文件拷贝的过程显示功能正确。

图3.2 拷贝过程中显示内容

2. 文件拷贝结束后，程序运行结果如图3.3所示。

图3.3 拷贝完成后显示内容

3. 用cmp指令比较源文件和副本文件的差异，结果如图3.4所示，结果表明源文件与副本文件内容无差异，即程序拷贝功能正确。

图3.4 cmp比较结果

# 实验二：增加系统功能调用

## 实验目的

掌握系统调用的实现过程；

掌握编译linux内核的基本方法。

## 实验内容

1. 通过编译内核方法，增加一个新的系统调用，实现文件拷贝功能；

2. 另编写一个应用程序，调用新增加的系统调用以测试其功能正确性。

## 程序清单

**1. 功能调用sys\_filecopy函数定义：**

asmlinkage int sys\_filecopy(char \*sname,char \*dname)

{

int fd\_s;

int fd\_d;

char buf[911];

int sizecopid;

mm\_segment\_t fs;

fs = get\_fs();

set\_fs(get\_ds());

if (-1 == (fd\_s = sys\_open(sname, O\_RDONLY, S\_IRWXU | S\_IRWXO | S\_IRWXG))){

printk("failed to open %s\n", sname);

return -1;

}

if (-1 == (fd\_d = sys\_open(dname, O\_CREAT | O\_RDWR, S\_IRWXU | S\_IRWXO | S\_IRWXG))){

printk("failed to create %s\n", dname);

return -1;

}

while (1){

if (-1 == (sizecopid = sys\_read(fd\_s, buf, 900))){

printk("failed to read %s\n", sname);

return -1;

}

if (sizecopid <= 0)

break;

if (-1 == sys\_write(fd\_d, buf, sizecopid)){

printk("failed to write %s\n", dname);

return -1;

}

printk("copied %d bytes\n", sizecopid);

}

sys\_close(fd\_s);

sys\_close(fd\_d);

set\_fs(fs);

return 0;}

**2. 测试程序syscall\_test.c：**

#include <stdio.h>

#include <stdlib.h>

#include <sys/syscall.h>

int main(int argc,char \*const argv[]){

//调用新的系统功能调用

syscall(223,argv[1],argv[2]);

printf("file copied!\n");

return 0;

}

## 实验步骤

1. 安装好编译环境后，到官方网站<www.kernel.org>下载Linux内核4.10，并以管理员权限解压内核到/usr/src/目录下；

2. 打开linux-4.10/kernel/sys.c文件，加入linkage.h的头文件，并在文件末尾加入4.3节中的sys\_filecopy系统调用函数，如图4.1和图4.2；

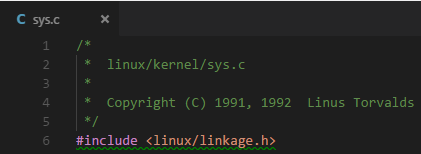


图4.1 在sys.c中加入linkage.h头文件

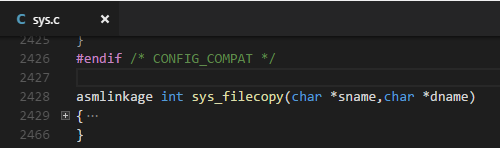


图4.2 在sys.c中加入sys\_filecopy函数定义

3. 打开linux-4.10/arch/x86/include/asm/syscalls.h文件，加入系统调用函数声明，如图4.3所示；

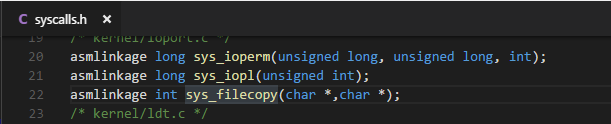


图4.3 在syscalls.h中加入sys\_filecopy函数声明

4. 打开linux-4.10/arch/x86/entry/syscalls/syscall\_32.tbl文件，加入系统功能调用号，如图4.4所示；

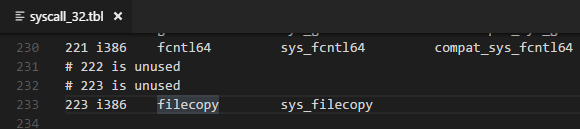


图4.4 在syscall\_32.tbl文件中加入系统功能调用号

5. 编译并安装内核，在终端依次执行：

sudo make menuconfig

sudo make –j8

sudo make modules\_install

sudo make install

6. 重启计算机，启动菜单中选择高级选项，加载刚刚安装的内核版本进入操作系统；

7. 编译4.3中的测试程序syscall\_test.c ：

gcc –o test syscall\_test.c

8. 运行测试程序（需要管理员权限）：

sudo ./test syscall\_test.c file.backup

9. 比较源文件和副本文件的内容，以验证功能调用正确性：

cmp syscall\_test.c file.backup

## 调试记录及运行结果

1. 按照实验步骤进性测试，运行测试程序后，用dmesg命令查看日志信息如图4.5所示。

图4.5 测试文件拷贝系统功能调用的日志信息

2. 比较源文件和副本文件的内容，结果如图4.6所示，结果表明源文件与副本文件内容无差异，即程序拷贝功能正确。

图4.6 cmp比较源文件与副本文件的结果

# 实验三：添加设备驱动程序

## 实验目的

掌握增加设备驱动程序的基本方法。

## 实验内容

通过模块方法，增加一个新的字符设备驱动程序，实现简单的功能。

## 实验设计

模块名定义为hualong，模块功能为一个50字节大小的缓存区设备的驱动程序，测试程序可对该缓存区读写。实现的功能函数有open、close、read、write和lseek。模块功能的具体实现见5.4节中的模块定义源程序。

## 程序清单

**1. 模块定义hualong.c**

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/init.h>

#include <asm/uaccess.h>

#define MY\_MAJOR 231

#define DEVICE\_NAME "HuaLong"

#define buf\_size 50

static char buffer[buf\_size] = "";

int pos;

static int hualong\_open(struct inode \*inode, struct file \*file)

{

printk(KERN\_ALERT "hualong is opened.\n");

pos = 0;

return 0;

}

static int hualong\_read(struct file \*file, char \_\_user \*buf, size\_t count, loff\_t \*ppos)

{

printk(KERN\_ALERT "hualong is being read.\n");

if (\*ppos >= buf\_size)

return 0;

if (count > (buf\_size - \*ppos))

count = buf\_size - \*ppos;

if (copy\_to\_user((void \*)buf, (void \*)(buffer + \*ppos), count))

{

return -EFAULT;

}

\*ppos = \*ppos + count;

return count;

}

static int hualong\_write(struct file \*file, const char \_\_user \*buf, size\_t count, loff\_t \*ppos)

{

printk(KERN\_ALERT "hualong is being writede.\n");

if (\*ppos >= buf\_size)

return 0;

if (count > (buf\_size - \*ppos))

count = buf\_size - \*ppos;

if (copy\_from\_user((void \*)(buffer + \*ppos), (void \*)buf, count))

{

return -EINVAL;

}

\*ppos = \*ppos + count;

return count;

}

loff\_t hualong\_llseek(struct file \*filp, loff\_t off, int whence)

{

loff\_t newpos;

switch (whence)

{

case 0: /\* SEEK\_SET \*/

newpos = off;

break;

case 1: /\* SEEK\_CUR \*/

newpos = filp->f\_pos + off;

break;

case 2: /\* SEEK\_END \*/

newpos = buf\_size + off;

break;

default: /\* can't happen \*/

return -EINVAL;

}

if (newpos < 0 || newpos > buf\_size)

return -EINVAL;

filp->f\_pos = newpos;

return newpos;

}

/\*关闭设备文件\*/

int hualong\_close(struct inode \*inode, struct file \*filp)

{

printk(KERN\_ALERT "hualong is closed.\n");

return 0;

}

static struct file\_operations hualong\_flops = {

.owner = THIS\_MODULE,

.open = hualong\_open,

.read = hualong\_read,

.write = hualong\_write,

.llseek = hualong\_llseek,

.release = hualong\_close,

};

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

{

int ret;

ret = register\_chrdev(MY\_MAJOR, DEVICE\_NAME, &hualong\_flops);

if (ret < 0)

{

printk(KERN\_ALERT DEVICE\_NAME " can't register MY\_MAJOR number.\n");

return ret;

}

printk(KERN\_ALERT DEVICE\_NAME " initialized.\n");

return 0;

}

static void \_\_exit hualong\_exit(void)

{

unregister\_chrdev(MY\_MAJOR, DEVICE\_NAME);

printk(KERN\_ALERT DEVICE\_NAME " removed.\n");

}

module\_init(hualong\_init);

module\_exit(hualong\_exit);

MODULE\_LICENSE("GPL");

**2. Makefile文件**

ifneq ($(KERNELRELEASE),)

MODULE\_NAME = mymodule

$(MODULE\_NAME)-objs := hualong.o

obj-m := $(MODULE\_NAME).o

else

KERNEL\_DIR = /lib/modules/`uname -r`/build

MODULEDIR := $(shell pwd)

.PHONY: modules

default: modules

modules:

make -C $(KERNEL\_DIR) M=$(MODULEDIR) modules

clean distclean:

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

rm -rf .tmp\_versions

endif

**3. 测试程序hualongtest.c**

#include <fcntl.h>

#include <stdio.h>

#include <unistd.h>

int main(void)

{

int fd,myseek,readsize,writesize;

char buf[200] = "";

fd = open("/dev/hualong", O\_RDWR);

if(fd < 0){

printf("can't open!\n");

}

const char c[] = "0123456789abcdefghijklmnopqrstuvwxwzABCDEFGHIJKLMNOPQRSTUVWXYZ";

writesize = write(fd, c, 55);

printf("write: %d bytes\n",writesize);

myseek = lseek(fd,0,SEEK\_CUR);

printf("myseek = %d\n",myseek);

readsize = read(fd,buf,20);

printf("read: %d bytes\n",readsize);

myseek = lseek(fd,0,SEEK\_CUR);

printf("myseek = %d\n",myseek);

myseek = lseek(fd,10,SEEK\_SET);

printf("myseek = %d\n",myseek);

readsize = read(fd,buf,20);

printf("read: %d bytes\n",readsize);

myseek = lseek(fd,0,SEEK\_CUR);

printf("myseek = %d\n",myseek);

for(int i = 0;i<20;i++)

putchar(buf[i]);

putchar('\n');

close(fd);

return 0;

}

## 实验步骤

1. 编译驱动：

sudo make

2. 编译测试程序

gcc hualongtest.c -o hualongtest

3. 安装模块

sudo insmod hualong.ko

4. 查看安装日志

dmesg -T -l 1

5. 查看是否成功加载模块

cat /proc/devices |grep 231

6. 新建对应设备文件

sudo mknod /dev/hualong c 231 0

7. 查看设备文件是否建立成

ll /dev/hualong

8. 运行测试程序

./hualongtest

## 调试记录及运行结果

按照实验步骤操作的结果如图5.1所示，最后测试程序的运行结果表明设备驱动的功能都正确。

图5.1 模块测试过程和测试结果

# 实验四：Proc文件系统的使用

## 实验目的

了解和掌握proc文件系统的特点和使用方法。

## 实验内容

1. 通过读取proc文件系统，获取系统各种信息，并以比较容易理解的方式显示出来；

2. 使用 GTK+下的c语言开发，实现图形界面；

3. 功能清单：

(1) 获取并显示主机名；

(2) 获取并显示系统启动的时间；

(3) 显示系统到目前为止持续运行的时间；

(4) 显示系统的版本号；

(5) 显示CPU的型号和主频大小；

(6) 同过pid或者进程名查询一个进程，并显示该进程的详细信息，及提供杀掉该进程的功能；

(7) 显示系统所有进程的一些信息，包括pid，ppid，占用内存大小，优先级等等；

(8) 图形化显示2分钟内CPU使用率的历史记录曲线；

(9) 图形化显示2分钟内内存使用率的历史纪录曲线；

(10) 在状态栏显示当前时间；

(11) 在状态栏显示当前CPU使用率；

(12) 在状态栏显示当前内存使用情况；

(13) 用新线程运行一个其他程序；

(14) 关机功能。

## 实验设计

1. 使用glade

## 程序清单

## 实验步骤

## 调试记录及运行结果