



华中科技大学

操作系统原理课程设计报告

姓 名：樊俊超

学 院：计算机

专 业：计算机科学与技术

班 级：CS1607

学 号：U201614702

指导教师：胡贯荣

分数	
教师签名	

2019 年 3 月 28 日

目 录

1	实验一 LINUX 用户界面的使用.....	1
1.1	实验目的	1
1.2	实验内容	1
1.3	实验设计	2
1.3.1	开发环境	2
1.3.2	实验设计	2
1.4	实验调试	3
1.4.1	实验步骤	3
1.4.2	实验调试及心得	4
	附录 实验代码	5
2	实验二 掌握系统调用的实现过程	21
2.1	实验目的	21
2.2	实验内容	21
2.3	实验设计	22
2.3.1	开发环境	22
2.3.2	实验设计	22
2.4	实验调试	24
2.4.1	实验步骤	24
2.4.2	实验调试及心得	25
	附录 实验代码	25
3	实验三 掌握增加设备驱动程序的方法。	27
3.1	实验目的	27
3.2	实验内容	27
3.3	实验设计	27
3.3.1	开发环境	27
3.3.2	实验设计	27
3.4	实验调试	28
3.4.1	实验步骤	28
3.4.2	实验调试及心得	30
	附录 实验代码	30
4	实验四 使用 GTK/QT 实现系统监控器	34

4.1	实验目的	34
4.2	实验内容	34
4.3	实验设计	35
4.3.1	开发环境	35
4.3.2	实验设计	35
4.4	实验调试	35
4.4.1	实验步骤	35
4.4.2	实验调试及心得	36
附录	实验代码	36

1 实验一 LINUX 用户界面的使用

1.1 实验目的

掌握 LINUX 操作系统的使用方法。

初步掌握 GTK/QT 的使用方法。

1.2 实验内容

编写 gtk 程序

1. 初始化 GTK。
2. 建立控件
3. 登记消息与消息处理函数
4. 执行消息循环函数 `gtk_main()`

只有 `gtk_main_quit()` 函数才能停止 Gtk+ 的执行，从而最终退出应用程序。

把 `gtk_main_quit()` 函数放在某个消息处理函数之中

编译和执行

程序中用到 Gtk+ 函数或定义的每一部分必须包含 `gtk/gtk.h` 文件，此外，还必须连接若干库。

`gcc hello.c -o hello `pkg-config --cflags` `pkg-config --libs`` 反引号（在键盘上位于字符 ``` 的左边

`chmod -777 hello`” 将 `hello` 设定为可执行的文件。

编一个 C 程序，其内容为实现文件拷贝的功能。基本要求：使用系统调用 `open/read/write`；选择：容错、`cp`。

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`）

基本要求：三个独立子进程，各自窗口显示；

选择：三个进程誊抄演示。

1.3 实验设计

1.3.1 开发环境

操作系统: Ubuntu 4.4.4

编译器: gcc

编辑器: vim、geany

GTK: GTK2.0

1.3.2 实验设计

使用系统调用函数 `open/read/write` 时需要用到 `unistd.h` 库, 拷贝源文件和目标文件参数在调用时传入, 若源文件不存在, 输出错误提示, 拷贝失败。

使用 `gtk2.0` 实现分窗口显示誊抄时, 除了 `main` 文件实现缓冲区创建, 信号灯集创建及赋值、子进程创建调用外, 其他三个子进程 `get`、`copy`、`put` 包括两部分: 执行部分、`gtk` 显示部分。下面列出具体情况。

关于拷贝函数部分:

实验实现需要两个共享存储区充当缓冲区 `S`、`T`, 每当 `get` 程序向 `S` 中写一次, 由 `copy` 程序从 `S` 中把文件拷贝至 `T` 中, 拷贝完成后, 调用 `put` 程序将文件从 `T` 中存进打开的文件里。故 `main` 文件需要创建两个共享存储区, 两个信号灯集负责两个存储区操作控制, 另外创建两个共享存储区存放当前拷贝内容字节数用以结束判断。

由于三个子进程文件的 `gtk` 部分大致相同, 下面显示 `gtk` 部分:

`Gtk` 显示部分包括: 进程名、子进程 `pid` 和父进程 `pid`、当前系统时间、拷贝字节数大小、拷贝块数。拿 `get` 部分举例:

1. 创建 `gtk` 窗口, 设置标题、伸缩属性、大小、关闭属性;
2. 创建垂直容器, 在容器中创建标签输出进程 `id`、创建标签调用函数输出系统时间、创建标签动态更新拷贝字节数和拷贝块数。
3. 创建线程更新每次需要显示的数字, 创建线程调用 `get` 函数执行获取过程。
4. 主事件循环。

关于信号灯:

缓冲区 `S`: `gcsshmId` 管理信号灯, 0 信号灯为 1 表示缓冲区 `S` 可写; 1 信号灯为 1 表示缓冲区 `S` 可读。

缓冲区 T: cpshmId 管理信号集, 0 信号灯为 1 表示缓冲区 T 可写; 1 信号灯为 1 表示缓冲区 T 可读。

当调用 get 时, 写入文件前对 gcshmId 信号集 0 信号灯进行 P 操作, 此时 copy 区陷入阻塞, 拷贝完成后, 更新拷贝字节数并对 1 信号灯进行 V 操作。

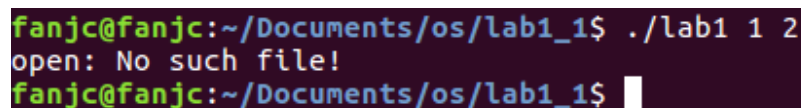
当调用 copy 时, 从 S 缓冲区复制内容到 T 缓冲区时, 对 gcshmId 信号集 1 信号灯进行 P 操作, 对 cpshmId 信号集 0 信号灯进行 P 操作, 读取记录 S 缓冲区拷贝字节数信息 (同样加入 PV 操作), 并据此判断是否需要结束拷贝。每次拷贝到 T 缓冲区后, 对 gcshmId 信号集 0 信号灯进行 V 操作, 对 cpshmId 信号集 1 信号灯进行 V 操作。

当调用 put 时, 写入文件前对 cpshmId 信号集 1 信号灯进行 P 操作, 此时 copy 区陷入阻塞, 拷贝完成后, 读取拷贝字节数做出结束判断, 并对 0 信号灯进行 V 操作。

1.4 实验调试

1.4.1 实验步骤

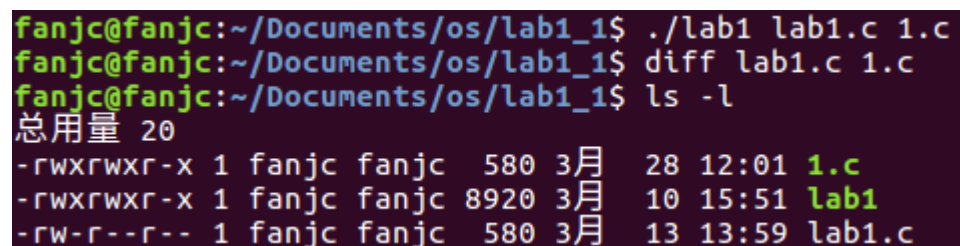
1. 测试文件拷贝, 在终端输入 ./lab1 1 2, 预期结果为输出错误信息。结果如图 1.1, 与预期结果相符。



```
fanjc@fanjc:~/Documents/os/lab1_1$ ./lab1 1 2
open: No such file!
fanjc@fanjc:~/Documents/os/lab1_1$
```

图 1.1

2. 在终端输入 ./lab1 lab1.c 1.c, 使用 diff lab1.c 1.c 及 ls -l 来判断拷贝后文件, 结果如图 1.2



```
fanjc@fanjc:~/Documents/os/lab1_1$ ./lab1 lab1.c 1.c
fanjc@fanjc:~/Documents/os/lab1_1$ diff lab1.c 1.c
fanjc@fanjc:~/Documents/os/lab1_1$ ls -l
总用量 20
-rwxrwxr-x 1 fanjc fanjc 580 3月 28 12:01 1.c
-rwxrwxr-x 1 fanjc fanjc 8920 3月 10 15:51 lab1
-rw-r--r-- 1 fanjc fanjc 580 3月 13 13:59 lab1.c
```

图 1.2

3. 测试誊抄文件演示, 运行 build.sh 脚本编译文件, 在终端输入 ./main 1 2, 结果与预期结果相同, 如图 1.3。

```
fanjc@fanjc: ~/Documents/os/lab1_2
fanjc@fanjc:~/Documents/os/lab1_2$ ./main 1 2
open: No such file!
fanjc@fanjc:~/Documents/os/lab1_2$
```

图 1.3

4. 在终端输入./main main sb, 拷贝过程如图 1.4 所示, 上面显示子进程、父进程 pid, 当前系统时间, 当前拷贝块数, 拷贝完成后, 给新拷贝的 sb 文件添加权限, 用它拷贝 1.txt 文件 (大小 8 字节), 结果如图 1.5。



图 1.4

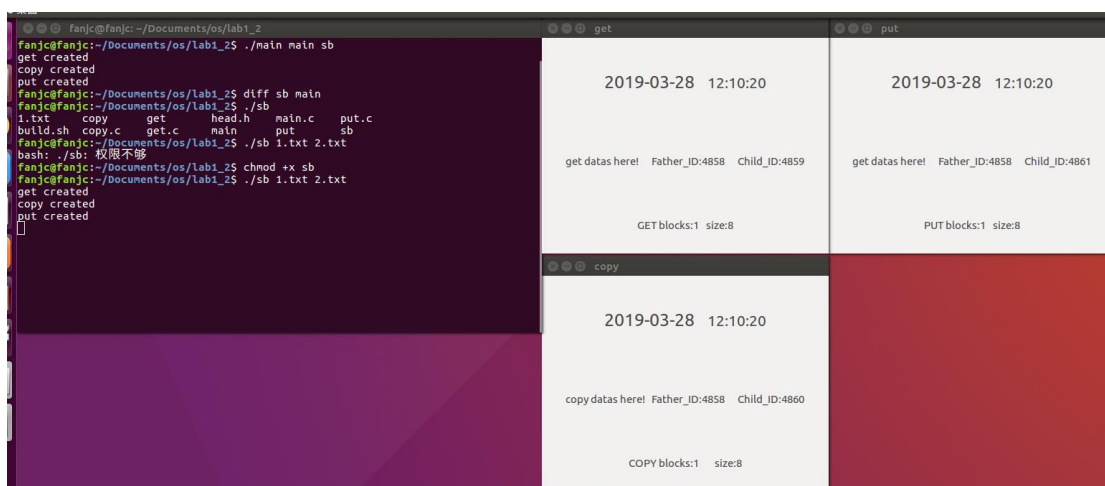


图 1.5

1.4.2 实验调试及心得

文件拷贝很顺利, 但写誊抄演示时由于想要学习如何使用 gtk, 所以就找的网上的 gtk 教程, 慢慢学, 基础的讲的很详细, 包括如何创建窗口, 使用标签, 获取系统时间等。但线程部分就不好写了, 就是如何能在拷贝每一块文件内容后即时的显示在窗口中。查了很多资料, 最后决定采用公共变量, 并把原有的拷贝程序也当作线程调用, 这样就解决了问题。

附录 实验代码

Lab1.c

```
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#define COPY_LENGTH 1024

int main(int argc, char *argv[]){
    char block[COPY_LENGTH];
    if(argc != 3){
        printf("Error occurred for numbers!");
    }
    int fin=open(argv[1],O_RDONLY,S_IRUSR | S_IRGRP | S_IROTH);
    if(fin==-1){
        printf("open: No such file!\n");
        close(fin);
        return 0;
    }

    int fout=open(argv[2],O_WRONLY | O_CREAT,0777);
    int cnt=0;
    while((cnt=read(fin,block,COPY_LENGTH))>0){
        write(fout,block,cnt);
    }
    close(fin);
    close(fout);
    return 0;
}
```

Build.sh

```
gcc -o main main.c head.h `pkg-config --cflags gthread-2.0 --libs gtk+-2.0`
```

```
gcc -o get get.c head.h `pkg-config --cflags gthread-2.0 --libs gtk+-2.0`  
gcc -o copy copy.c head.h `pkg-config --cflags gthread-2.0 --libs gtk+-2.0`  
gcc -o put put.c head.h `pkg-config --cflags gthread-2.0 --libs gtk+-2.0`
```

Head.h

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <time.h>  
#include <sys/types.h>  
#include <gtk/gtk.h>  
#include <sys/ipc.h>  
#include <sys/shm.h>  
#include <sys/sem.h>  
#include <sys/wait.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#define true 1  
#define BUFLLENGTH 100  
#define gcsem_key 100  
#define gcshm_hey 99  
#define gcinfo_key 98  
#define cpsem_key 97  
#define cpshm_key 96  
#define cpinfo_key 95  
key_t key=1;  
union semum{  
    short val; //供 SETVAL 使用  
    struct semid_ds* buf; //semid_ds 结构  
    unsigned short* array; //SETALL 所用数组值  
    struct seminfo *_buf; //为控制 IPC_INFO 提供缓存  
}arg;  
  
void P(int semid,int index){
```

```

    struct sembuf sem;
    sem.sem_num=index;
    sem.sem_op=-1;
    sem.sem_flg=0;
    semop(semid,&sem,1);
    return;
}

```

```

void V(int semid,int index){
    struct sembuf sem;
    sem.sem_num=index;
    sem.sem_op=1;
    sem.sem_flg=0;
    semop(semid,&sem,1);
    return;
}

```

Main.c

```

#include "head.h"

int main(int argc,char *argv[]){
    int fin=open(argv[1],O_RDONLY,S_IRUSR | S_IRGRP | S_IROTH);
    if(fin==-1){
        printf("open: No such file!\n");
        close(fin);
        return 0;
    }
    close(fin);
    pid_t pid1,pid2,pid3;
    int gcshmlId,gcsemId,cpshmlId,cpsemId,gc_num,cp_num;
    gcshmlId=shmget(gcshm_hey,BUFLNGTH,IPC_CREAT|0666); //创建共享
存储区
    cpshmlId=shmget(cpshm_key,BUFLNGTH,IPC_CREAT|0666); //创建共享
存储区
    gc_num=shmget(gcinfo_key,sizeof(int),IPC_CREAT|0666);

```

```
cp_num=shmget(cpinfo_key,sizeof(int),IPC_CREAT|0666);
int *gc_tip=(int *)shmat(gc_num,NULL,0);
int *cp_tip=(int *)shmat(cp_num,NULL,0);
*gc_tip=BUFLNGTH;
*cp_tip=BUFLNGTH;
key=gcsem_key;
gcsemId=semget(key,3,IPC_CREAT|0666); //创建信号灯
if(gcsemId==-1){
    printf("semget error!\n");
    return 0;
}
//设置 0 信号灯，表示缓冲区能写个数
arg.val=1;
semctl(gcsemId,0,SETVAL,arg);
//设置 1 信号灯，表示缓冲区能读个数
arg.val=0;
semctl(gcsemId,1,SETVAL,arg);
arg.val=1;
semctl(gcsemId,2,SETVAL,arg);

key=cpsem_key;
cpsemId=semget(key,3,IPC_CREAT|0666); //创建信号灯
if(cpsemId==-1){
    printf("semget error!\n");
    return 0;
}
//设置 0 信号灯，表示缓冲区能写个数
arg.val=1;
semctl(cpsemId,0,SETVAL,arg);
//设置 1 信号灯，表示缓冲区能读个数
arg.val=0;
semctl(cpsemId,1,SETVAL,arg);
arg.val=1;
semctl(cpsemId,2,SETVAL,arg);
```

```

//创建子进程
if((pid1=fork())==0){
    //get 操作
    printf("get created\n");
    execv("./get",argv);
}else{
    if((pid2=fork())==0){
        //copy 操作
        printf("copy created\n");
        execv("./copy",argv);
    }else{
        if((pid3=fork())==0){
            //put 操作
            printf("put created\n");
            execv("./put",argv);
        }else{
            //退出
            int p;
            waitpid(pid1,&p,0);
            waitpid(pid2,&p,0);
            waitpid(pid3,&p,0);
            shmctl(gcshmId,IPC_RMID,0); //撤销共享存储区
            shmctl(cpshmId,IPC_RMID,0); //撤销共享存储区
            shmctl(gc_num,IPC_RMID,0);
            shmctl(cp_num,IPC_RMID,0);
            semctl(gcsemId,0,IPC_RMID); //删除信号集
            semctl(cpsemId,0,IPC_RMID); //删除信号集
            return 0;
        }
    }
}
}

```

Get.c

```
#include "head.h"
```

```

GtkWidget *data_label;
int num=0,read_howmany=0;
void get(char *argv[]){
    int semId,shmId,cnt=0;
    char *data;
    shmId=shmget(gcshm_hey,BUFLNGTH,0666);
    data=(char *)shmat(shmId,NULL,0);
    cnt=shmget(gcinfo_key,sizeof(int),0666);
    int *tip=(int *)shmat(cnt,NULL,0);
    //int num=0,read_howmany=0;
    //获取已存在的信号集
    semId=semget(gcsem_key,3,0666);
    FILE *fp;
    if((fp=fopen(argv[1],"r"))==NULL){
        printf("read file open error\n");
        return;
    }
    while(true){
        P(semId,0);
        num=fread(data,sizeof(char),BUFLNGTH,fp);
        read_howmany+=1;
        //printf("get:%s\n",data);
        //V(semId,1);
        P(semId,2);
        *tip=num;
        V(semId,2);
        V(semId,1);
        usleep(100000);
        if(num!=BUFLNGTH){
            fclose(fp);
            break;
        }
    }
    return;
}

```

```

gboolean settime(gpointer T_data)
{
    time_t times;
    struct tm *p_time;
    time(&times);
    p_time = localtime(&times);
    gchar          *text_data          =          g_strdup_printf("<span
size='xx-large'>%04d-%02d-%02d</span>",\
        (1900+p_time->tm_year),(1+p_time->tm_mon),(p_time->tm_mday));
    gchar          *text_time          =          g_strdup_printf("<span
size='x-large'>%02d:%02d:%02d</span>",\
        (p_time->tm_hour), (p_time->tm_min), (p_time->tm_sec));
    gchar *text_markup = g_strdup_printf("\n%s\t%s\n", text_data, text_time);
    gtk_label_set_markup(GTK_LABEL(T_data), text_markup);

    return TRUE;
}

void show(){
    char buf[32];
    memset(buf,'\0',32);
    while(true){
        sprintf(buf,"GET blocks:%d\tsize:%d",read_howmany,num);
        gdk_threads_enter();
        gtk_label_set_text(GTK_LABEL(data_label),buf);
        gdk_threads_leave();
        usleep(100000);
    }
    return;
}

int main(int argc,char *argv[]){
    //自动完成一些必要的初始化工作， 设置缺省的信号处理函数
    gtk_init(&argc,&argv);

```

```

GtkWidget *window,*button,*vbox,*progress,*tlabel,*label;
//创建窗口并返回指针
window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
//设置窗口标题
gtk_window_set_title(GTK_WINDOW(window),"get");
//设置窗体伸缩属性
gtk_window_set_resizable (GTK_WINDOW (window), TRUE);
//设置窗口位置属性
//gtk_window_set_position          (GTK_WINDOW          (window),
GTK_WIN_POS_CENTER);
//设置窗口大小
gtk_widget_set_size_request(window,400,300);
//设置窗口关闭属性
g_signal_connect(window,"destroy",G_CALLBACK(gtk_main_quit),NULL)
;

//创建一个垂直容器
vbox=gtk_vbox_new(FALSE,10);
//将 vbox 放入窗口中
gtk_container_add(GTK_CONTAINER(window),vbox);
//创建标签,输出进程 id
char str[64];
memset(str,'\0',64);
sprintf(str,"get datas here!\tFather_ID:%d\tChild_ID:%d",getppid(),getpid());
label=gtk_label_new(str);
tlabel=gtk_label_new(NULL);
data_label=gtk_label_new(NULL);
gint s = g_timeout_add (1000, settime, (void *)tlabel);
//加入 vbox 中
gtk_container_add(GTK_CONTAINER(vbox),tlabel);
gtk_container_add(GTK_CONTAINER(vbox),label);
gtk_container_add(GTK_CONTAINER(vbox),data_label);
//创建线程
gdk_threads_init();
g_thread_create((GThreadFunc)get,argv,FALSE,NULL);
g_thread_create((GThreadFunc)show,NULL,FALSE,NULL);

```

```
//显示所有窗口
gtk_widget_show_all(window);
//主事件循环
gdk_threads_enter();
gtk_main();
gdk_threads_leave();
return 0;
}

Copy.c
#include "head.h"
int write_howmany=0,num=0;
GtkWidget *data_label;

void copy(char *argv[]){
    int gcsemId,gcshmId,cpsemId,cpshmId;
    int gc_cnt=0,cp_cnt=0;
    char *gc_data;
    char *cp_data;
    gcshmId=shmget(gcshm_hey,BUFLNGTH,0666);
    cpshmId=shmget(cpshm_key,BUFLNGTH,0666);
    gc_data=(char *)shmat(gcshmId,NULL,0);
    cp_data=(char *)shmat(cpshmId,NULL,0);
    gc_cnt=shmget(gcinfo_key,sizeof(int),0666);
    cp_cnt=shmget(cpinfo_key,sizeof(int),0666);
    int *gc_tip=(int *)shmat(gc_cnt,NULL,0);
    int *cp_tip=(int *)shmat(cp_cnt,NULL,0);
    //获取已存在的信号集
    gcsemId=semget(gcsem_key,3,0666);
    cpsemId=semget(cpsem_key,3,0666);
    int t1=BUFLNGTH;
    while(true){
        P(gcsemId,1);
        P(cpsemId,0);
        P(gcsemId,2);
        t1=*gc_tip;
```

```

        V(gcsemId,2);
        memcpy(cp_data,gc_data,t1);
        P(cpsemId,2);
        *cp_tip=t1;
        V(cpsemId,2);
        write_howmany+=1;
        num=t1;
        V(cpsemId,1);
        V(gcsemId,0);
        usleep(100000);
        if(t1!=BUFLNGTH){
            break;
        }
    }
    return;
}

```

```

gboolean settime(gpointer T_data)
{
    time_t times;
    struct tm *p_time;
    time(&times);
    p_time = localtime(&times);
    gchar          *text_data          =          g_strdup_printf("<span
size='xx-large'>%04d-%02d-%02d</span>",\
        (1900+p_time->tm_year),(1+p_time->tm_mon),(p_time->tm_mday));
    gchar          *text_time          =          g_strdup_printf("<span
size='x-large'>%02d:%02d:%02d</span>",\
        (p_time->tm_hour), (p_time->tm_min), (p_time->tm_sec));
    gchar *text_markup = g_strdup_printf("\n%s\t%s\n", text_data, text_time);
    gtk_label_set_markup(GTK_LABEL(T_data), text_markup);

    return TRUE;
}

```

```

void show(){
    char buf[32];
    memset(buf,'\0',32);
    while(true){
        sprintf(buf,"COPY blocks:%d\tsize:%d",write_howmany,num);
        gdk_threads_enter();
        gtk_label_set_text(GTK_LABEL(data_label),buf);
        gdk_threads_leave();
        usleep(100000);
    }
    return;
}

int main(int argc,char *argv[]){
    //自动完成一些必要的初始化工作，设置缺省的信号处理函数
    gtk_init(&argc,&argv);
    GtkWidget *window,*button,*vbox,*progress,*tlabel,*label;
    //创建窗口并返回指针
    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
    //设置窗口标题
    gtk_window_set_title(GTK_WINDOW(window),"copy");
    //设置窗体伸缩属性
    gtk_window_set_resizable (GTK_WINDOW (window), TRUE);
    //设置窗口位置属性
    //gtk_window_set_position          (GTK_WINDOW          (window),
GTK_WIN_POS_CENTER);
    //设置窗口大小
    gtk_widget_set_size_request(window,400,300);
    //设置窗口关闭属性
    g_signal_connect(window,"destroy",G_CALLBACK(gtk_main_quit),NULL)
;

    //创建一个垂直容器
    vbox=gtk_vbox_new(FALSE,10);

```

```

//将 vbox 放入窗口中
gtk_container_add(GTK_CONTAINER(window),vbox);
//创建标签,输出进程 id
char str[64];
memset(str,'\0',64);
sprintf(str,"copy                                     datas
here!\tFather_ID:%d\tChild_ID:%d",getppid(),getpid());
label=gtk_label_new(str);
tlabel=gtk_label_new(NULL);
data_label=gtk_label_new(NULL);
gint s = g_timeout_add (1000, settime, (void *)tlabel);
//加入 vbox 中
gtk_container_add(GTK_CONTAINER(vbox),tlabel);
gtk_container_add(GTK_CONTAINER(vbox),label);
gtk_container_add(GTK_CONTAINER(vbox),data_label);
//创建线程
gdk_threads_init();
g_thread_create((GThreadFunc)copy,argv,FALSE,NULL);
g_thread_create((GThreadFunc)show,NULL,FALSE,NULL);
//显示所有窗口
gtk_widget_show_all(window);
//主事件循环
gdk_threads_enter();
gtk_main();
gdk_threads_leave();
return 0;
}

```

Put.c

```

#include "head.h"
int write_howmany=0,num=0;
GtkWidget *data_label;

```

```
void put(char *argv[]){
    int semId,shmId,cnt=0;
    char *data;
    shmId=shmget(cpshm_key,BUFLNGTH,0666);
    data=(char *)shmat(shmId,NULL,0);
    //获取已存在的信号集
    semId=semget(cpsem_key,3,0666);
    cnt=shmget(cpinfo_key,sizeof(int),0666);
    int *tip=(int *)shmat(cnt,NULL,0);
    int t1=BUFLNGTH;
    FILE *fp;
    if((fp=fopen(argv[2],"w+"))==NULL){
        printf("write file open error\n");
        return 0;
    }
    while(true){
        P(semId,1);
        P(semId,2);
        t1=*tip;
        V(semId,2);
        if(t1!=BUFLNGTH){
            fwrite(data,sizeof(char),t1,fp);
            write_howmany+=1;
            num=t1;
            V(semId,0);
            break;
        }
        fwrite(data,sizeof(char),BUFLNGTH,fp);
        write_howmany+=1;
        num=BUFLNGTH;
        V(semId,0);
        usleep(100000);
    }
    fclose(fp);
    return;
```

```

    }

gboolean settime(gpointer T_data)
{
    time_t times;
    struct tm *p_time;
    time(&times);
    p_time = localtime(&times);
    gchar *text_data = g_strdup_printf("<span
size='xx-large'>%04d-%02d-%02d</span>",\
    (1900+p_time->tm_year),(1+p_time->tm_mon),(p_time->tm_mday));
    gchar *text_time = g_strdup_printf("<span
size='x-large'>%02d:%02d:%02d</span>",\
    (p_time->tm_hour), (p_time->tm_min), (p_time->tm_sec));
    gchar *text_markup = g_strdup_printf("\n%s\t%s\n", text_data, text_time);
    gtk_label_set_markup(GTK_LABEL(T_data), text_markup);

    return TRUE;
}

void show(){
    char buf[32];
    memset(buf,'\0',32);
    while(true){
        sprintf(buf,"PUT blocks:%d\tsize:%d",write_howmany,num);
        gdk_threads_enter();
        gtk_label_set_text(GTK_LABEL(data_label),buf);
        gdk_threads_leave();
        usleep(100000);
    }
    return;
}

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

```

```

//自动完成一些必要的初始化工作，设置缺省的信号处理函数
gtk_init(&argc,&argv);
GtkWidget *window,*button,*vbox,*progress,*tlabel,*label;
//创建窗口并返回指针
window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
//设置窗口标题
gtk_window_set_title(GTK_WINDOW(window),"put");
//设置窗体伸缩属性
gtk_window_set_resizable (GTK_WINDOW (window), TRUE);
//设置窗口位置属性
//gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CEN
NTER);
//设置窗口大小
gtk_widget_set_size_request(window,400,300);
//设置窗口关闭属性
g_signal_connect(window,"destroy",G_CALLBACK(gtk_main_quit),NULL);
//创建一个垂直容器
vbox=gtk_vbox_new(FALSE,10);
//将 vbox 放入窗口中
gtk_container_add(GTK_CONTAINER(window),vbox);
//创建标签,输出进程 id
char str[64];
memset(str,'\0',64);
sprintf(str,"get datas here!\tFather_ID:%d\tChild_ID:%d",getppid(),getpid());
label=gtk_label_new(str);
tlabel=gtk_label_new(NULL);
data_label=gtk_label_new(NULL);
gint s = g_timeout_add (1000, settime, (void *)tlabel);
//加入 vbox 中
gtk_container_add(GTK_CONTAINER(vbox),tlabel);
gtk_container_add(GTK_CONTAINER(vbox),label);
gtk_container_add(GTK_CONTAINER(vbox),data_label);
//创建线程
gdk_threads_init();
g_thread_create((GThreadFunc)put,argv,FALSE,NULL);

```

```
g_thread_create((GThreadFunc)show,NULL,FALSE,NULL);
//显示所有窗口
gtk_widget_show_all(window);
//主事件循环
gdk_threads_enter();
gtk_main();
gdk_threads_leave();
return 0;
}
```

2 实验二 掌握系统调用的实现过程

2.1 实验目的

掌握系统调用的实现过程，通过编译内核方法，增加一个新的系统调用。另编写一个应用程序，使用新增加的系统调用。

- (1) 内核编译、生成，用新内核启动；
- (2) 新增系统调用实现：文件拷贝或 P、V 操作。

2.2 实验内容

Linux 内核中设置了一组用于实现各种系统功能的子程序，称为系统调用。用户可以通过系统调用命令在自己的应用程序中调用它们。

系统调用 核心态 操作系统核心提供

普通的函数调用 用户态 函数库或用户自己提供

很多已经被我们习以为常的 C 语言标准函数，在 Linux 平台上的实现都是靠系统调用完成的，如 `open()`, `close()`, `malloc()`, `fork()` .所以如果想对系统的原理作深入的了解，掌握各种系统调用是初步的要求。

系统调用工作原理

不能访问内核所占内存空间也不能调用内核函数。

进程调用一个特殊的指令，这个指令会跳到一个事先定义的内核中的一个位置（当然，这个位置是用户进程可读但是不可写的）。在 Intel CPU 中，这个由中断 `INT0x80` 实现。跳转到的内核位置叫做 `sysem_call`。检查系统调用号，这个号码代表进程请求哪种服务。然后，它查看系统调用表(`sys_call_table`)找到所调用的内核函数入口地址。接着，就调用函数，等返回后，做一些系统检查，最后返回到进程（或到其他进程，如果这个进程时间用尽）。系统调用号表示数组 `sys_call_table[]` 中的位置。

2.3 实验设计

2.3.1 开发环境

操作系统: Ubuntu 16.04

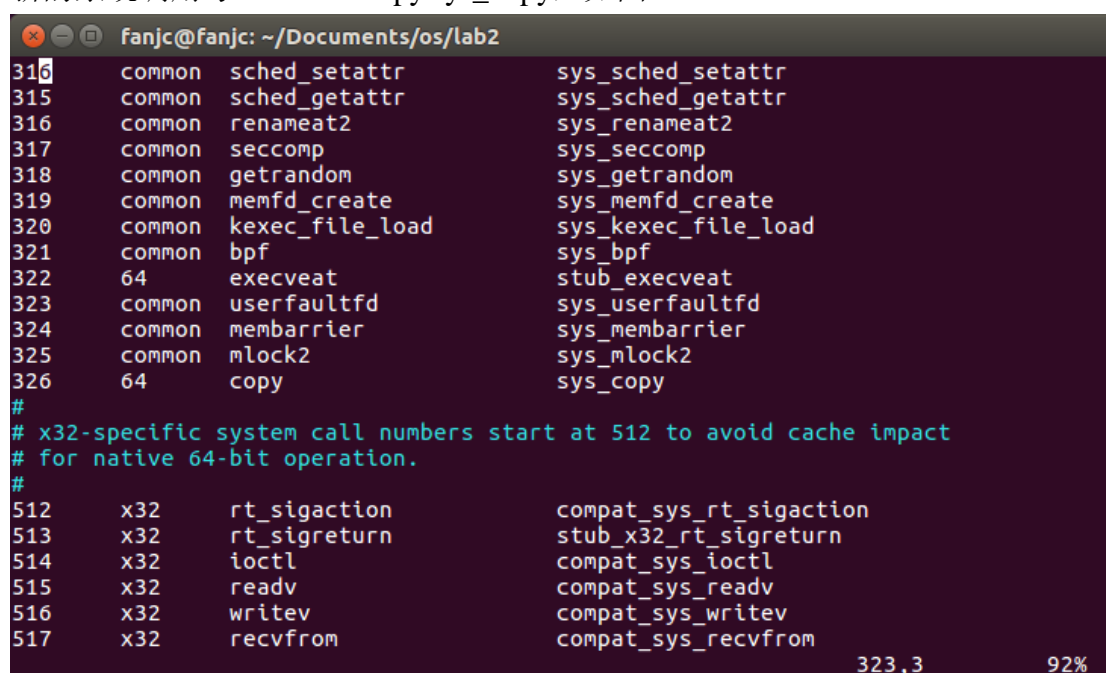
新编译内核: 4.4.4

编译器: gcc

编辑器: vim、geany

2.3.2 实验设计

1. 下载 4.4.4 内核版本, 在 Download 下解压缩, 此后命令均在文件内执行。
2. 了解到系统调用就是让用户使用命令来调用存放在内核中的实现各种功能的子程序。这些程序函数是在内核中实现的, 运行于核心态。所以需要内核中增加自己的函数和系统调用号。
3. 在使用 vim 编辑器打开 arch/x86/entry/syscalls/syscall_64.tbl, 在这里添加新的系统调用号: 326 64 copy sys_copy, 如图 2.1。



```
fanjc@fanjc: ~/Documents/os/lab2
316 common sched_setattr sys_sched_setattr
315 common sched_getattr sys_sched_getattr
316 common renameat2 sys_renameat2
317 common seccomp sys_seccomp
318 common getrandom sys_getrandom
319 common memfd_create sys_memfd_create
320 common kexec_file_load sys_kexec_file_load
321 common bpf sys_bpf
322 64 execveat stub_execveat
323 common userfaultfd sys_userfaultfd
324 common membarrier sys_membarrier
325 common mlock2 sys_mlock2
326 64 copy sys_copy
#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation.
#
512 x32 rt_sigaction compat_sys_rt_sigaction
513 x32 rt_sigreturn stub_x32_rt_sigreturn
514 x32 ioctl compat_sys_ioctl
515 x32 readv compat_sys_readv
516 x32 writev compat_sys_writev
517 x32 recvfrom compat_sys_recvfrom
323,3 92%
```

图 2.1

4. 使用 vim 编辑器打开 include/linux/syscalls.h, 在这里添加函数声明: asmlinkage long sys_copy(const char *src_file, const char *copy_file); 如图 2.2。

```
fanjc@fanjc: ~/Documents/os/lab2
const struct iovec __user *lvec,
unsigned long liovcnt,
const struct iovec __user *rvec,
unsigned long riovcnt,
unsigned long flags);

asmlinkage long sys_kcmp(pid_t pid1, pid_t pid2, int type,
    unsigned long idx1, unsigned long idx2);
asmlinkage long sys_finit_module(int fd, const char __user *uargs, int flags);
asmlinkage long sys_seccomp(unsigned int op, unsigned int flags,
    const char __user *uargs);
asmlinkage long sys_getrandom(char __user *buf, size_t count,
    unsigned int flags);
asmlinkage long sys_bpf(int cmd, union bpf_attr *attr, unsigned int size);

asmlinkage long sys_execveat(int dfd, const char __user *filename,
    const char __user *const __user *argv,
    const char __user *const __user *envp, int flags);

asmlinkage long sys_membarrier(int cmd, int flags);

asmlinkage long sys_mlock2(unsigned long start, size_t len, int flags);
asmlinkage long sys_copy(const char *src_file, const char *copy_file);
#endif
892,1 底端
```

图 2.2

5. 使用 vim 编辑器打开 kernel/sys.c 在这里添加函数实现，如图 2.3。

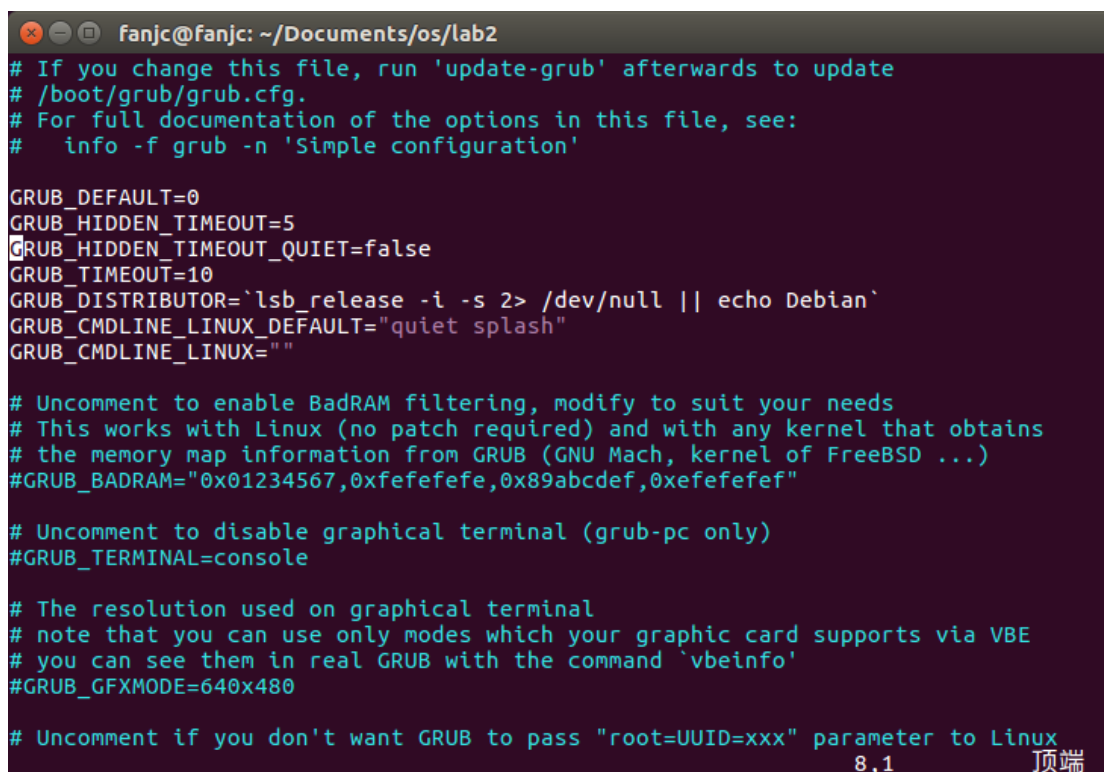
```
fanjc@fanjc: ~/Documents/os/lab2
__put_user(s.mem_unit, &info->mem_unit))
return -EFAULT;

return 0;
}
#endif /* CONFIG_COMPAT */
asmlinkage long sys_copy(const char*src_file, const char*copy_file){
    int fin, fout, cnt;
    char block[512];
    mm_segment_t fs;
    fs = get_fs();
    set_fs(get_ds());
    fin=sys_open(src_file, O_RDONLY, 0);
    if(fin<0){
        printk("src_file open error!\n");
        return 1;
    }
    fout=sys_open(copy_file, O_WRONLY | O_CREAT, 0777);
    while((cnt=sys_read(fin, block, 512))>0){
        sys_write(fout, block, cnt);
    }
    printk("copy successfully!\n");
    sys_close(fin);
    sys_close(fout);
    set_fs(fs);
    return 0;
}
2444,2-9 底端
```

图 2.3

6. 执行下列命令：sudo menuconfig 打开后选择 save 保存生成的.config 文件，然后退出。sudo make 编译内核，sudo make modules_install 安装模块，sudo make install 安装内核。

7. 修改启动项，在终端输入 `sudo vim /etc/default/grub`，设置里面的启动时间项。修改后内容如图 2.4；然后更新(`sudo update-grub`)。



```
fanjc@fanjc: ~/Documents/os/lab2
# If you change this file, run 'update-grub' afterwards to update
# /boot/grub/grub.cfg.
# For full documentation of the options in this file, see:
#   info -f grub -n 'Simple configuration'

GRUB_DEFAULT=0
GRUB_HIDDEN_TIMEOUT=5
GRUB_HIDDEN_TIMEOUT_QUIET=false
GRUB_TIMEOUT=10
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash"
GRUB_CMDLINE_LINUX=""

# Uncomment to enable BadRAM filtering, modify to suit your needs
# This works with Linux (no patch required) and with any kernel that obtains
# the memory map information from GRUB (GNU Mach, kernel of FreeBSD ...)
#GRUB_BADRAM="0x01234567,0xfefefefefe,0x89abcdef,0xefefefef"

# Uncomment to disable graphical terminal (grub-pc only)
#GRUB_TERMINAL=console

# The resolution used on graphical terminal
# note that you can use only modes which your graphic card supports via VBE
# you can see them in real GRUB with the command 'vbeinfo'
#GRUB_GFXMODE=640x480

# Uncomment if you don't want GRUB to pass "root=UUID=xxx" parameter to Linux
```

图 2.4

8. 一切成功后，转入系统调试。
9. 在上列步骤中添加函数实现时需要注意，输出语句使用 `printk`，在内核中对文件的操作函数 `open`、`close`、`read`、`write` 均需加上 `sys_`前缀。还需使用 `get_fs()`、`get_ds()`、`set_fs()`函数来修改地址访问限制值。

2.4 实验调试

2.4.1 实验步骤

1. 根据实验设计完成内核编译后，创建测试文件，在测试文件中使用 `syscall()`函数调用新增加的系统调用。在终端输入命令 `gcc -o lab2 lab2.c` 编译测试文件。
2. 在终端输入 `./lab2 1 2`，在新的终端使用 `dmesg` 查看输出，结果如图 2.5 所示。

```
fanjc@fanjc:~/Documents/os/lab2$ ./lab2 1 2
open: No such files!
fanjc@fanjc:~/Documents/os/lab2$ ./lab2 lab2.c 1.c
sys_copy:0
src_filename:lab2.c,copy_filename:1.c
fanjc@fanjc:~/Documents/os/lab2$
```



```
fanjc@fanjc: ~
[ 2114.026983] QNX4 filesystem 0.2.3 registered.
[ 2114.942101] raid6: sse2x1 gen() 3716 MB/s
[ 2115.010098] raid6: sse2x1 xor() 2921 MB/s
[ 2115.078102] raid6: sse2x2 gen() 4787 MB/s
[ 2115.146103] raid6: sse2x2 xor() 3116 MB/s
[ 2115.214109] raid6: sse2x4 gen() 4965 MB/s
[ 2115.282107] raid6: sse2x4 xor() 3511 MB/s
[ 2115.350099] raid6: avx2x1 gen() 7026 MB/s
[ 2115.418109] raid6: avx2x2 gen() 7939 MB/s
[ 2115.486108] raid6: avx2x4 gen() 8904 MB/s
[ 2115.486111] raid6: using algorithm avx2x4 gen() 8904 MB/s
[ 2115.486113] raid6: using avx2x2 recovery algorithm
[ 2115.489650] xor: automatically using best checksumming function:
[ 2115.526108]   avx      : 26171.000 MB/sec
[ 2115.635458] Btrfs loaded
[ 2992.631354] atkbd serio0: Unknown key pressed (translated set 2, code 0xd8 on isa0060/serio0).
[ 2992.631367] atkbd serio0: Use 'setkeycodes e058 <keycode>' to make it known.
[ 2992.639174] atkbd serio0: Unknown key released (translated set 2, code 0xd8 on isa0060/serio0).
[ 2992.639187] atkbd serio0: Use 'setkeycodes e058 <keycode>' to make it known.
[ 6847.862781] src_file open error!
[ 6909.206412] copy successfully!
fanjc@fanjc:~$
```

图 2.5

3. 在终端输入`./lab2 lab2.c 1.c`，使用 `diff` 及 `ls` 命令比较结果，同时在新的终端使用 `dmesg` 查看输出，结果如图 2.5 所示。

2.4.2 实验调试及心得

实验二有老师的 ppt 还是很好做的，基本没遇到什么问题。但也有让我印象深的地方，就是不知道怎么进入内核更换界面，“无意“碰到了 `esc` 键，成功解决，然后发现老师 ppt 那一页说，请使用 `esc` 进入键显示菜单，很尴尬。

附录 实验代码

Test.c

```
#include <stdio.h>
#include <fcntl.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>
```

```
int main(int argc,char *argv[]){
    long int b;
    b=syscall(326,argv[1],argv[2]);
    if(b==0){
        printf("sys_copy:%ld\n",b);
        printf("src_filename:%s,copy_filename:%s\n",argv[1],argv[2]);
    }else
        printf("open: No such files!\n");
    return 0;
}
```

函数实现 sys_copy

```
asm linkage long sys_copy(const char*src_file,const char*copy_file){
    int fin,fout,cnt;
    char block[512];
    mm_segment_t fs;
    fs = get_fs();
    set_fs(get_ds());
    fin=sys_open(src_file,O_RDONLY,0);
    if(fin<0){
        printk("src_file open error!\n");
        return 1;
    }
    fout=sys_open(copy_file,O_WRONLY | O_CREAT,0777);
    while((cnt=sys_read(fin,block,512))>0){
        sys_write(fout,block,cnt);
    }
    printk("copy successfully!\n");
    sys_close(fin);
    sys_close(fout);
    set_fs(fs);
    return 0;
}
```

3 实验三 掌握增加设备驱动程序的方法。

3.1 实验目的

通过模块方法，增加一个新的字符设备驱动程序，其功能可以简单,基于内核缓冲区。

3.2 实验内容

内核模块

LKM Loadable Kernel Modules

Linux 核心是一种 monolithic 类型的内核，即单一的大核心。另外一种形式是 MicroKernel，核心的所有功能部件都被拆成独立部分，这些部分之间通过严格的通讯机制进行联系。linux 内核是一个整体结构，因此向内核添加任何东西，或者删除某些功能，都十分困难。为了解决这个问题，引入了模块机制，从而可以动态的在内核中添加或者删除模块。模块一旦被插入内核,他就和内核其他部分一样。

基本要求：演示实现字符设备读、写； 选择：键盘缓冲区，不同进程、追加、读取。

3.3 实验设计

3.3.1 开发环境

操作系统：Ubuntu 16.04

新编译内核：4.4.4

编译器：gcc

编辑器：vim、geany

3.3.2 实验设计

模块的实现机制：

Linux 为我们提供了两个命令：使用 insmod 来显式加载核心模块，使用

rmmod 来卸载模块。同时核心自身也可以请求核心后台进程 kerneld 来加载与卸载模块。对于每一个内核模块来说，必定包含两个函数：

int init_module() 在插入内核时启动,在内核中注册一定的功能函数。

int cleanup_module() 当内核模块卸载时，调用它将模块从内核中清除。

一个典型的驱动程序,大体上可以分为这么几个部分：

注册设备：

在系统初启,或者模块加载时候,必须将设备登记到相应的设备数组,并返回设备的主设备号；

定义功能函数：

对于每一个驱动函数来说，都有一些和此设备密切相关的功能函数。以最常用的块设备或者字符设备来说，都存在着诸如 open()、read()这一类的操作。当系统调用这些调用时，将自动的使用驱动函数中特定的模块。来实现具体的操作；

卸载设备：

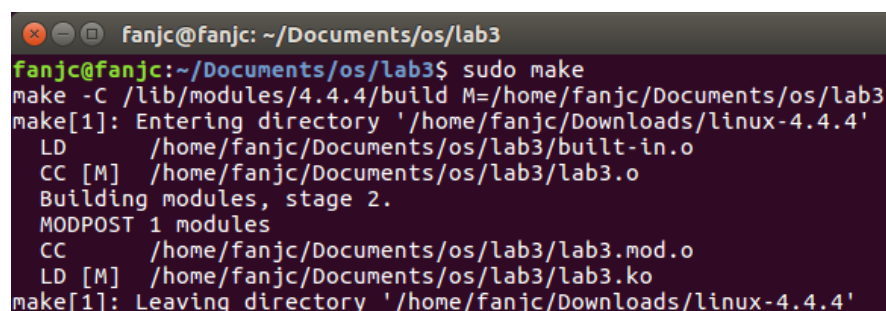
在不用这个设备时,可以将它卸载，主要是从/proc 中取消这个设备的特殊文件。

1. 首先使用 geany 编辑器编写驱动程序文件，然后利用老师给的 makefile 模板，修改后保存为 Makefile 文件。
2. 在 lab3 目录下生成.ko 文件后，加载并测试程序，使用 dmesg 来查看信息。

3.4 实验调试

3.4.1 实验步骤

1. 在终端输入 sudo make 生成.ko 文件，并使用 insmod 命令加载模块，如图 3.1 所示。



```
fanjc@fanjc: ~/Documents/os/lab3
fanjc@fanjc:~/Documents/os/lab3$ sudo make
make -C /lib/modules/4.4.4/build M=/home/fanjc/Documents/os/lab3
make[1]: Entering directory '/home/fanjc/Downloads/linux-4.4.4'
LD      /home/fanjc/Documents/os/lab3/built-in.o
CC [M]  /home/fanjc/Documents/os/lab3/lab3.o
Building modules, stage 2.
MODPOST 1 modules
CC      /home/fanjc/Documents/os/lab3/lab3.mod.o
LD [M]  /home/fanjc/Documents/os/lab3/lab3.ko
make[1]: Leaving directory '/home/fanjc/Downloads/linux-4.4.4'
```

图 3.1

2. 在加载模块前使用 `cat` 命令查看 `/proc/devices` 下未被占用的数字，结果如图 3.2 所示。

```
89 i2c
99 ppdev
108 ppp
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
242 new_dev
243 media
244 mei
245 hidraw
246 nvme
247 bsg
248 watchdog
249 ptp
250 pps
251 rtc
252 dimmctl
253 ndctl
254 tpm

Block devices:
259 blkext
7 loop
8 sd
```

图 3.2

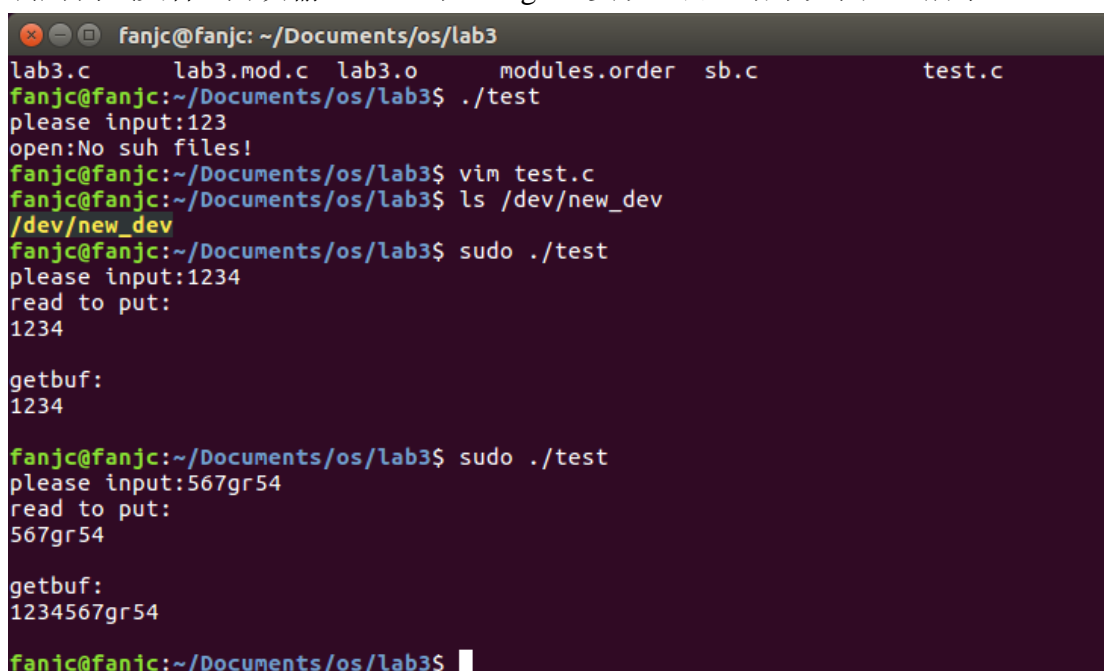
3. 在终端输入 `sudo mknod /dev/new_dev c 242 0`，再次查看 `/proc/devices`，结果如图 3.3 所示。

```
fanjc@fanjc: ~/Documents/os/lab3
108 ppp
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
242 new_dev
243 media
244 mei
245 hidraw
246 nvme
247 bsg
248 watchdog
249 ptp
250 pps
251 rtc
252 dimmctl
253 ndctl
254 tpm

Block devices:
259 blkext
7 loop
8 sd
```

图 3.3

4. 调用测试文件，两次输入 1234 和 567gr54 实现追加，结果如图 3.4 所示。



```
fanjc@fanjc: ~/Documents/os/lab3
lab3.c      lab3.mod.c  lab3.o      modules.order  sb.c        test.c
fanjc@fanjc:~/Documents/os/lab3$ ./test
please input:123
open:No suh files!
fanjc@fanjc:~/Documents/os/lab3$ vim test.c
fanjc@fanjc:~/Documents/os/lab3$ ls /dev/new_dev
/dev/new_dev
fanjc@fanjc:~/Documents/os/lab3$ sudo ./test
please input:1234
read to put:
1234

getbuf:
1234

fanjc@fanjc:~/Documents/os/lab3$ sudo ./test
please input:567gr54
read to put:
567gr54

getbuf:
1234567gr54

fanjc@fanjc:~/Documents/os/lab3$
```

图 3.4

3.4.2 实验调试及心得

实验三开始没看懂怎么增加字符设备驱动，还好老师的 ppt 很详细，总的过程很顺利，就是编译内核时间有点长。一开始使用的 manjrao 系统，还有最新的内核版本，但最后发现不能用，这就很烦。索性就装回了 ubuntu16.04 版本，新内核使用 4.4.4 这次到没出什么问题。但后来又碰到了污点内核，在网上百度说是签名问题，但也没找到解决办法，最后我把之前修改的部分改了部分，解决了这个问题。

附录 实验代码

Lab3.c

```
#include <linux/module.h>
#include <linux/types.h>
#include <linux/fs.h>
#include <linux/sched.h>
#include <linux/init.h>
#include <linux/cdev.h>
#include <linux/slab.h>
#include <asm/uaccess.h>
```

```

#include <linux/uaccess.h>

#define BUFLNGTH 1024
struct cdev cdev;//字符设备结构体
static int major = 0;//主设备号变量

struct MemD
{
    unsigned long size;
    char *buffer;
}*memdev;

static int new_open(struct inode *inode, struct file *file)
{
    file->private_data = memdev;
    return 0;
}

static ssize_t new_read(struct file *file, char __user *buf, size_t count, loff_t
*p_cfo)
{
    struct MemD *dev = file->private_data;
    if (copy_to_user(buf, (void*)(dev->buffer + file->f_pos), count)){
        file->f_pos += count;
        return count;
    }else
        return -EFAULT;
}

static ssize_t new_write(struct file *file, const char __user *buf, size_t count,
loff_t *p_cfo)
{
    struct MemD *dev = file->private_data;
    if (!copy_from_user(dev->buffer + file->f_pos, buf, count)){
        memdev->size+=count;

```

```

        file->f_pos = memdev->size;
        return count;
    }
    else
        return -EFAULT;
}

static loff_t new_llseek(struct file *file, loff_t offset, int whence)
{
    loff_t now_off=0;
    switch(whence){
        case SEEK_SET:
            now_off=offset;break;
        case SEEK_CUR:
            now_off=file->f_pos+memdev->size;break;
        case SEEK_END:
            now_off=BUFLNGTH-1+offset;break;
        default:
            return -EINVAL;
    }
    if ((now_off<0) || (now_off>BUFLNGTH))
        return -EINVAL;
    file->f_pos = now_off;
    return now_off;
}

struct file_operations fops={
    .owner = THIS_MODULE,
    .open = new_open,
    .read = new_read,
    .write = new_write,
    .llseek = new_llseek,
};

int init_module(void)

```

```

{
    major = register_chrdev(0,"new_dev",&fops);
    if (major < 0) {
        printk(KERN_INFO "Newdev: FAIL to get major number\n");
        return major;
    }
    cdev_init(&cdev, &fops);
    cdev.owner = THIS_MODULE;
    cdev.ops = &fops;
    cdev_add(&cdev,MKDEV(major, 0), 1);
    memdev = kmalloc(sizeof(struct MemD), GFP_KERNEL);
    if (!memdev)
    {
        major = -ENOMEM;
        return major;
    }
    memdev->size = 0;
    memdev->buffer = kmalloc(BUFLNGTH, GFP_KERNEL);
    memset(memdev->buffer, 0, BUFLNGTH);
    return 0;
}

void cleanup_module(void)
{
    cdev_del(&cdev);
    kfree(memdev);
    kfree(memdev->buffer);
    unregister_chrdev(major,"new_dev");
}

MODULE_LICENSE("GPL");

```

4 实验四 使用 GTK/QT 实现系统监控器

4.1 实验目的

1. 了解/proc 文件的特点和使用方法;
2. 监控系统中进程运行情况;
3. 用图形界面实现系统资源的监控。

4.2 实验内容

Linux 的 PROC 文件系统是进程文件系统和内核文件系统的组成的复合体, 是将内核数据对象化为文件形式进行存取的一种内存文件系统, 是监控内核的一种用户接口. 它拥有一些特殊的文件(纯文本), 从中可以获取系统状态信息。

(1) 系统信息

与进程无关,随系统配置的不同而不同.

命令 `procinfo` 可以显示这些文件的大量信息

(2) 进程信息

系统中正在运行的每一个用户级进程的信息。

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。

使用 `GTK+ /Qt` 下的 `c` 语言开发。

具体包括:

- (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, 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)`cpu` 使用率的图形化显示(2 分钟内的历史纪录曲线)

-
- (9)内存和交换分区(swap)使用率的图形化显示(2 分钟内的历史纪录曲线)
 - (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)用新进程运行一个其他程序
 - (14)关机功能

4.3 实验设计

4.3.1 开发环境

操作系统: Ubuntu 4.4.4

编译器: `gcc`

编辑器: `vim`、`geany`

4.3.2 实验设计

本次实验我只实现了 1、2、3、4、5、10 六个最基本的显示功能,原理仿照实验一的誊抄演示部分。主要读取 `proc` 文件系统,对获取的数据做处理之后输出即可。

`/proc/cmdline`:内核启动的命令行

`/proc/cpuinfo`:CPU 信息

`/proc/stat`:CPU 的使用情况、磁盘、页面、交换、所有的中断、最后一次的启动时间等。

`/proc/meminfo`:内存状态的有关信息。

实验首先创建主窗口,在主窗口中创建 `notebook`,

4.4 实验调试

4.4.1 实验步骤

4.4.2 实验调试及心得

由于做实验四之前没想着实验四会有多耗费时间，想着只是难度比较大，然而我做了才发现，我 gtk 学的不足以让我把实验四在两三天内写完。所以最后只能继续找新的 gtk 教程，最后只把显示部分给做完并调试好。这次试验是我莽了，一开始没有准备好 GTK 的知识，导致只能边学边写实验，虽然调试过程的错误都是小错误，但是 gtk 更深层次的东西我没有拿来实践，这让我感到遗憾。

附录 实验代码

Lab4.c

```
#include <gtk/gtk.h>
#include <linux/kernel.h>
#include <stdio.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <stdlib.h>
#include <signal.h>

void show_sysinfo(GtkWidget *);
gint time_engine( gpointer);
gint time_run( gpointer);
gint show_time(gpointer);

int main(int argc,char *argv[]){
    GtkWidget *notebook,*notebook_label,*notebook_frame,*window_table;
    //自动完成一些必要的初始化工作，设置缺省的信号处理函数
    gtk_init(&argc,&argv);
    GtkWidget *window,*tlabel;
```

```

//创建窗口并返回指针
window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
//设置窗口标题
gtk_window_set_title(GTK_WINDOW(window),"资源监视器");
//设置窗体伸缩属性
gtk_window_set_resizable (GTK_WINDOW (window), TRUE);
//设置窗口位置属性
gtk_window_set_position          (GTK_WINDOW          (window),
GTK_WIN_POS_CENTER);
//设置窗口大小
gtk_widget_set_size_request(window,960,600);
gtk_widget_show(window);

//创建 window_table
window_table=gtk_table_new(10,1,TRUE);
gtk_container_add(GTK_CONTAINER(window),window_table);
gtk_widget_show(window_table);
//显示时间
tlabel=gtk_label_new(NULL);
show_time(tlabel);
gtk_timeout_add(1000, show_time,tlabel);
gtk_table_attach_defaults(GTK_TABLE(window_table),tlabel,0,1,9,10);
gtk_widget_show(tlabel);

//创建 notebook
notebook=gtk_notebook_new();
gtk_notebook_set_tab_pos(GTK_NOTEBOOK(notebook),GTK_POS_TOP)
;

gtk_table_attach_defaults(GTK_TABLE(window_table),notebook,0,1,0,10);
gtk_widget_show(notebook);
notebook_frame=gtk_frame_new(NULL);
notebook_label=gtk_frame_new("系统");
//gtk_container_set_border_width(GTK_CONTAINER(notebook_frame),0);
gtk_widget_show(notebook_frame);
gtk_widget_show(notebook_label);

```

```

        gtk_notebook_append_page(GTK_NOTEBOOK(notebook),notebook_frame,
notebook_label);
        show_sysinfo(notebook_frame);

        //设置窗口关闭属性
        g_signal_connect(window,"destroy",G_CALLBACK(gtk_main_quit),NULL)
;

        gtk_main();
        return 0;
    }

```

```

void show_sysinfo(GtkWidget *frame){
    GtkWidget *frame_table;
    GtkWidget *label1,*label2,*label4,*label3;
    GtkWidget *engine_label,*run_label;
    int fd,i;
    char buf[1000],buf1[150];
    char *pbuf;
    frame_table=gtk_table_new(8,1,TRUE);
    gtk_container_add(GTK_CONTAINER(frame),frame_table);
    gtk_widget_show(frame_table);

```

//显示主机名

```

    fd=open("/proc/sys/kernel/hostname",O_RDONLY);
    read(fd,buf,20);
    pbuf=strtok(buf,"\n");
    sprintf(buf1,"主机名:%10s",pbuf);
    label1=gtk_label_new(buf1);
    //放入第一行
    gtk_table_attach_defaults(GTK_TABLE(frame_table),label1,0,1,0,1);
    gtk_widget_show(label1);
    close(fd);

```

//显示系统版本号

```

    fd=open("/proc/sys/kernel/ostype",O_RDONLY);

```

```
read(fd,buf,20);
pbuf=strtok(buf,"\n");
memset(buf1,'\0',sizeof(buf1));
strcpy(buf1,pbuf);
fd=open("/proc/sys/kernel/osrelease",O_RDONLY);
read(fd,buf,20);
pbuf=strtok(buf,"\n");
strcat(buf1,pbuf);
memset(buf,'\0',sizeof(buf));
sprintf(buf,"系统版本:%s",buf1);
label2=gtk_label_new(buf);
//放入第二行
gtk_table_attach_defaults(GTK_TABLE(frame_table),label2,0,1,1,2);
gtk_widget_show(label2);
close(fd);
```

//显示 CPU 型号和主频大小

```
fd=open("/proc/cpuinfo",O_RDONLY);
read(fd,buf,1000);
pbuf=strtok(buf,"\n");
for(int i=0;i<4;i++)
    pbuf=strtok(NULL,"\n");
sprintf(buf,"CPU %s",pbuf);
label3=gtk_label_new(buf);
gtk_table_attach_defaults(GTK_TABLE(frame_table),label3,0,1,2,3);
gtk_widget_show(label3);
close(fd);
```

//显示系统启动时间

```
enginlabel=gtk_label_new(NULL);
time_engine(enginlabel);
//gtk_timeout_add(1000,time_engine,(void *)enginlabel);
gtk_table_attach_defaults(GTK_TABLE(frame_table),enginlabel,0,1,3,4);
gtk_widget_show(enginlabel);
```

```

//显示系统运行时间
runlabel=gtk_label_new(NULL);
time_run(runlabel);
gtk_timeout_add(1000,time_run,(void *)runlabel);
gtk_table_attach_defaults(GTK_TABLE(frame_table),runlabel,0,1,4,5);
gtk_widget_show(runlabel);

return;
}

/*系统启动时间*/
gint time_engine( gpointer data )
{
    int fd,swtime;
    char buf[150];
    time_t ts;
    char *pbuf,*uptime;
    time(&ts);
    struct tm *ptm=localtime(&ts);
    GtkWidget *enginlabel = (GtkWidget *) (data);
    fd=open("/proc/uptime",O_RDONLY),swtime=0;
    int day,hour,min,sec;
    read(fd,buf,30);
    pbuf=strtok(buf,"\n");
    uptime=strtok(pbuf," ");
    for(int i=0;i<strlen(uptime)-3;i++)
        swtime=swtime*10+*(uptime+i)-48;
    day=swtime/(24*3600);
    hour=swtime/3600-24*day;
    min=(swtime%3600)/60;
    sec=swtime%60;
    sprintf(buf,"系统启动时间:%d 年%d 月%d 日%d 时%d 分%d 秒",ptm->tm_year+1900,ptm->tm_mon,ptm->tm_mday-day,ptm->tm_hour-hour,ptm->tm_min-min,ptm->tm_sec-sec);
    gtk_label_set_text(GTK_LABEL(enginlabel),buf);
}

```

```

        close(fd);
        return TRUE;
    }

/*系统持续运行时间*/
gint time_run( gpointer data )
{
    int fd=open("/proc/uptime",O_RDONLY),swtime=0;
    long int day,hour,min,sec;
    char buf[150];
    char *pbuf,*uptime;
    GtkWidget *runlabel = (GtkWidget*)(data);
    read(fd,buf,30);
    pbuf=strtok(buf,"\n");
    uptime=strtok(pbuf," ");
    for(int i=0;i<strlen(uptime)-3;i++)
        swtime=swtime*10+*(uptime+i)-48;
    day=swtime/(24*3600);
    hour=(swtime%(24*3600))/3600;
    min=(swtime%3600)/60;
    sec=swtime%60;
    sprintf(buf,"系统持续运行时间:%ld 日 %ld 时 %ld 分 %ld 秒",day,hour,min,sec);

    gtk_label_set_text(GTK_LABEL(runlabel),buf);
    close(fd);
    return TRUE;
}

//显示当前时间
gint show_time(gpointer data){
    time_t times;
    struct tm *p_time;
    time(&times);
    p_time = localtime(&times);

```

```
        gchar          *text_data          =          g_strdup_printf("<span
size='xx-large'>%04d-%02d-%02d</span>",\
        (1900+p_time->tm_year),(1+p_time->tm_mon),(p_time->tm_mday));
        gchar          *text_time          =          g_strdup_printf("<span
size='x-large'>%02d:%02d:%02d</span>",\
        (p_time->tm_hour), (p_time->tm_min), (p_time->tm_sec));
        gchar *text_markup = g_strdup_printf("\n%s\t%s\n", text_data, text_time);
        gtk_label_set_markup(GTK_LABEL(data), text_markup);

        return TRUE;
    }
```