



华中科技大学

操作系统原理课程设计报告

姓 名：

学 院： 计算机科学与技术学院

专 业： 计算机科学与技术

班 级：

学 号：

指导教师：

分数	
教师签名	

目 录

1 实验一 熟悉和理解 Linux 编程环境.....	1
1.1 实验目的.....	1
1.2 实验内容.....	1
1.3 实验设计.....	1
1.3.1 开发环境.....	1
1.3.2 实验设计.....	1
1.4 实验调试.....	4
1.4.1 实验步骤.....	4
1.4.2 实验调试及心得.....	6
附录 实验代码.....	7
2 实验二 掌握添加系统调用的方法.....	17
2.1 实验目的.....	17
2.2 实验内容.....	17
2.3 实验设计.....	17
2.3.1 开发环境.....	17
2.3.2 实验设计.....	17
2.4 实验调试.....	19
2.4.1 实验步骤.....	19
2.4.2 实验调试及心得.....	21
附录 实验代码.....	22
3 实验三 掌握添加设备驱动程序的方法.....	25
3.1 实验目的.....	25
3.2 实验内容.....	25
3.3 实验设计.....	25
3.3.1 开发环境.....	25
3.3.2 实验设计.....	25
3.4 实验调试.....	27
3.4.1 实验步骤.....	27
3.4.2 实验调试及心得.....	29
附录 实验代码.....	30
4 实验四 理解和分析/proc 文件.....	34
4.1 实验目的.....	34
4.2 实验内容.....	34
4.3 实验设计.....	34
4.3.1 开发环境.....	34
4.3.2 实验设计.....	35
4.4 实验调试.....	38
4.4.1 实验步骤.....	38

4.4.2 实验调试及心得.....	43
附录 实验代码.....	43

1 实验一 熟悉和理解 Linux 编程环境

1.1 实验目的

- (1) 掌握 Linux 操作系统的使用方法，包括键盘命令、系统调用。
- (2) 掌握在 Linux 下的编程环境。

1.2 实验内容

- (1) 编一个 C 程序，其内容为实现 cp（文件拷贝）的功能。使用系统调用 open/read/write，实现 cp 命令。
- (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。

1.3 实验设计

1.3.1 开发环境

操作系统：Ubuntu 16.04 LTS
系统类型：64 位
系统内核：Linux version 4.15.0
处理器：Intel® Core™ i5-8250U CPU @ 1.60GHz × 4
内存：3.8GB
编译器及版本：GCC version 5.4.0
图形用户界面库及版本：GTK+ 2.0

1.3.2 实验设计

1、实现文件拷贝功能。

Linux 系统文件拷贝的用户操作命令为 cp，实现指定源文件向目的文件的拷贝功能，若目的文件不存在，则创建后再拷贝。

使用单缓冲区 buffer 实现读写过程中数据的暂存，使用系统调用函数 open、close、read 与 write 实现完整的拷贝功能。另外，为增加拷贝功能的容错性，对

可能出现问题进行查错。具体流程如图 1.1 所示。

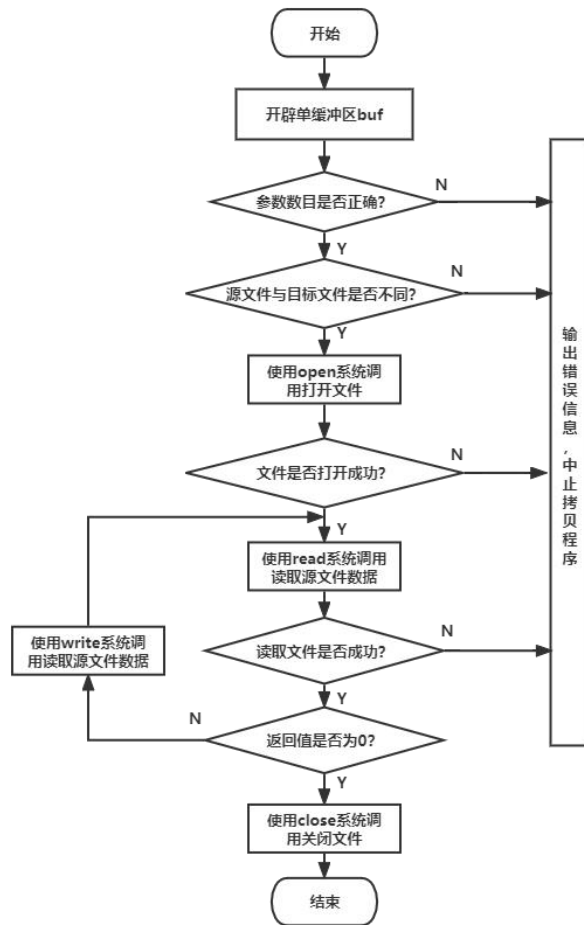


图 1.1 文件拷贝功能设计流程图

其中，输出错误信息与中止程序分别使用 `perror` 与 `exit` 函数。通过 `perror` 进行错误信息的输出显示，通过 `exit(1)` 进行程序中中止并退出。

对于系统功能调用函数的使用，进行下列说明：

(1) `open` 系统调用函数使用

`open` 系统调用函数原型有两种，`int open(const char*path, int oflags);`与 `int open(const char*path, int oflags, mode_t mode);`。

对于返回值，系统调用成功则返回一个文件描述符，为 `int` 类型；失败则返回 -1，可据此查错。对于调用参数，`path` 参数为待打开或待创建文件的路径名，`oflags` 参数为指定文件的创建模式。对于第二种调用方式，其中的 `mode` 参数给出 `C_CREAT` 模式下的访问权限。

以只读模式 `O_RDONLY` 打开源文件，以 `O_WRONLY|O_CREAT` 模式打开目的文件。在 `O_CREAT` 模式下，当目标文件不存在时，进行目标文件的创建。并设置 `mode` 参数为 `0600`，表示仅对于用户有文件的读写权限。

(2) `read` 和 `write` 系统调用函数使用

read 系统调用函数原型为 `ssize_t read(int fd, void* buf, size_t count);`

write 系统调用函数原型为 `ssize_t write(int fd, const void* buf, size_t count);`

对于返回值，为 `ssize_t` 类型，有符号整型。当读写成功时，返回读出与写入的字节数；失败时，则返回-1。当调用 `read` 前已经到达文件末尾，则此次 `read` 调用返回 0，可将此作为读取结束标志。

对于传递参数，`fd` 为待读写文件的文件描述符，`buf` 为缓冲区，`count` 为读写数据的字节数。其中，`buf` 作为单缓冲区，用来暂存源文件的读取数据，并将暂存的数据送入目标文件中。

2、实现并程序的多窗口显示。

该功能实现三个并行进程完成文件誊抄的演示，其中父进程进行共享内存和信号灯组信息的获取与创建，同时使用 `fork` 创建誊抄读写进程，使用 `execv` 函数执行对应程序。并且使用 `GTK+2.0` 函数库实现父进程信息提示窗口显示，显示父进程标志和进程启动按钮。

点击按钮后，进行读与写子进程的执行，并在对应的窗口上显示进程运行信息，在此选择当前文件的读写页数作为显示内容。使用 `gtk_label_set_text` 进行信息更替。

`GTK+2.0` 函数库的使用流程如图 1.2 所示。

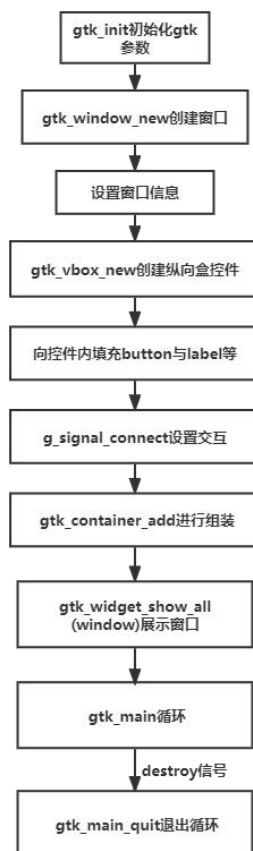


图 1.2 GTK+2.0 图形库编程流程图

在 GTK+2.0 编程流程图中, 包含 4 个阶段, 即初始化阶段、窗口设计阶段、交互设计阶段以及窗口展示阶段。

初始化阶段中, 使用 `gtk_init(&argc, &argv);` 来初始化 `gtk`, 将命令行参数传给 GTK+, 这个函数需要在 GTK+ 函数之前调用。

窗口设计阶段中, 使用 `gtk_window_new()` 来创建主窗口, 其中参数可设置为 `GTK_WINDOW_TOPLEVEL`, 表示生成标准有框架窗口。设置容器控件, 如 `vbox`、`hbox`、`table` 等, 可容纳多个控件; 而 `button` 只能容纳一个控件。在容器控件中添加非容器控件, 如 `label`、`entry` 等实现信息的显示。控件与主窗口之间通过 `gtk_container_add` 函数进行链接, 控件与控件之间则通过对应主控件的链接函数进行关联, 如对于 `box` 控件而言, 可使用 `gtk_box_pack_start`、`gtk_box_pack_end` 等实现链接。

交互设计阶段中, 则通过使用 `g_signal_connect` 函数, 实现可产生信号控件的交互设计。该函数原型为 `gulong g_signal_connect(gpointer instance, const gchar *detailed_signal, GCallback c_handler, gpointer data);` 其中 `instance` 为信号发出控件, 如 `button` 等; `detailed_signal` 为信号标志, 如“`destroy`”关闭窗口信号; `c_handler` 为回调函数名, 表示执行交互后运行的函数, 需要使用 `G_CALLBACK()` 进行转换; `data` 为回调函数传参。`gtk_main` 循环一般通过该函数调用 `gtk_main_quit` 结束, 具体函数实现为 `g_signal_connect(G_OBJECT(window), "destroy", G_CALLBACK(gtk_main_quit), NULL);`。

窗口展示阶段中, 使用 `gtk_widget_show_all (window);` 展示主窗口及其相关链接控件。使用 `gtk_main` 进行显示循环, 并等待控件产生信号。

1.4 实验调试

1.4.1 实验步骤

1、实现文件拷贝功能实验步骤。

(1) 编写文件拷贝程序 `Mycp.c`, 使用 `gcc` 编译, 生成可执行文件 `Mycp`。

(2) 设置源复制文件 `text.txt`, 进行程序测试。

(3) 测试目标文件的存在是否影响复制结果。设置不存在的文件作为目标文件时, 执行 `./Mycp text.txt text-copy.txt`, 程序执行成功, 源文件与生成的目标文件对比如图 1.3 所示。设置已存在的文件为目标文件时, 设置空文件 `text-empty-copy.txt` 作为目标文件, 执行 `./Mycp text.txt text-empty-copy.txt`, 程序执行成功, 源文件与生成的目标文件对比如图 1.4 所示。

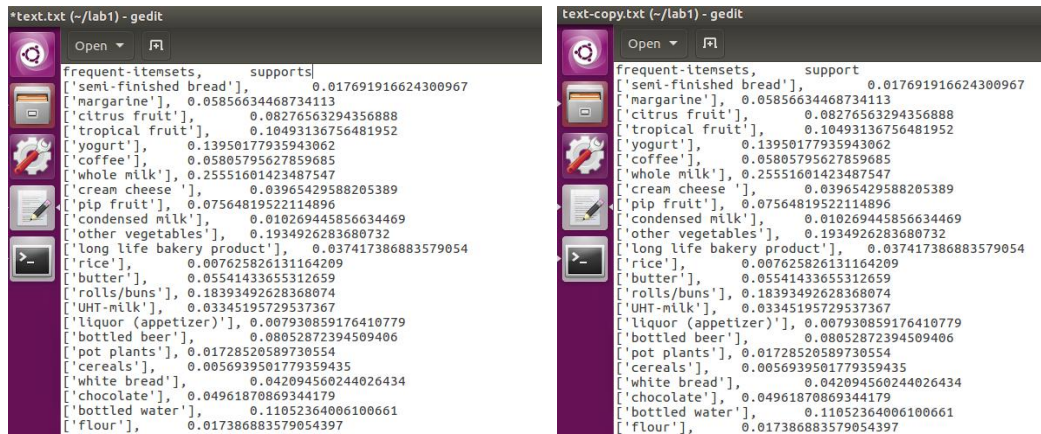


图 1.3 目标文件不存在时的拷贝功能测试

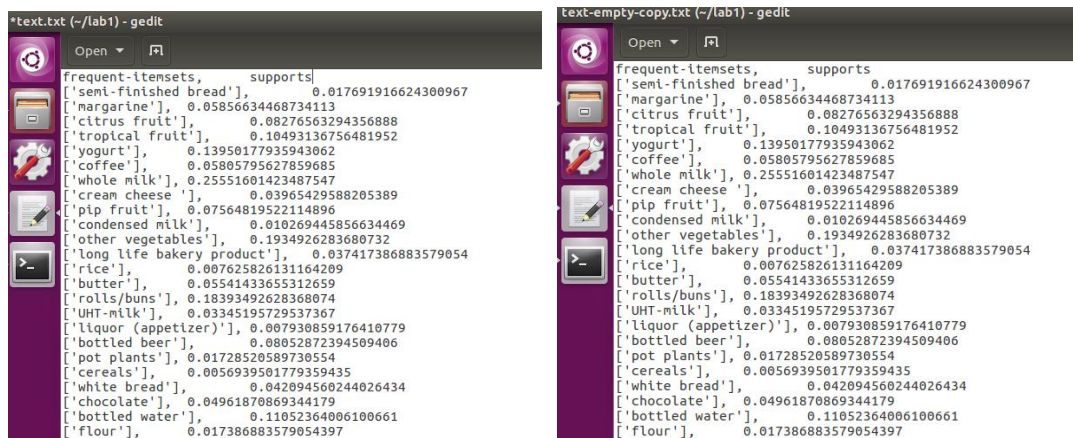


图 1.4 目标文件存在时的拷贝功能测试

(4) 容错性测试, 当填写参数缺少目标文件时, 执行 `./Mycp text.txt` 的程序执行结果如图 1.5 所示, 提示错误信息正确。

```
orange@ubuntu:~/lab1$ ./Mycp text.txt
please input correctly!
```

图 1.5 缺少参数时的拷贝功能测试

源文件名与目标文件名重复时, 执行 `./Mycp text.txt text.txt` 的程序执行结果如所示, 提示错误信息正确。

```
orange@ubuntu:~/lab1$ ./Mycp text.txt text.txt
the two files are same!
```

图 1.6 源文件与目标文件重复时的拷贝功能测试

2、实现并程序的多窗口显示实验步骤。

(1) 编写 3 个并行誊抄程序 `main.c`、`get.c` 和 `put.c`, 分别对 3 个文件使用 `gcc` 编译, 编译时链接 `gtk` 库, 如 `gcc -o main main.c `pkg-config --cflags --libs gtk+-2.0``。

(2) 执行主程序 `./main`, 产生 1 个主窗口。点击按钮 `Start` 后, 运行子进程, 创建 2 个子窗口, 如图 1.7 所示。

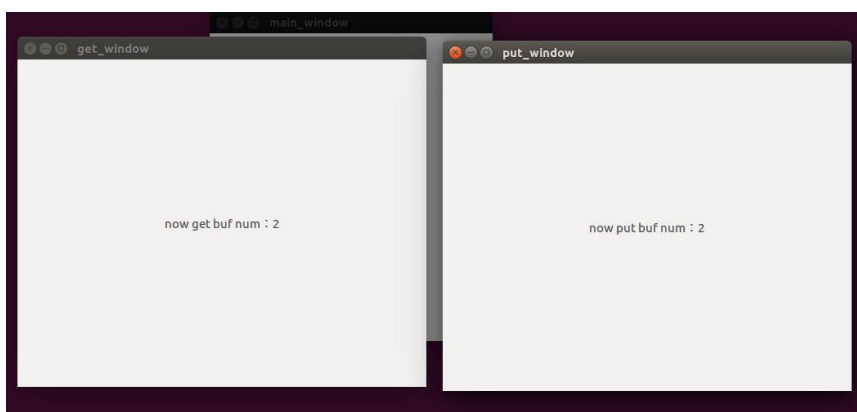


图 1.7 子进程窗口显示

当将子进程窗口关闭后，主窗口信息变换，如图 1.8 所示。再次点击 Again 按钮可继续运行誊抄程序。

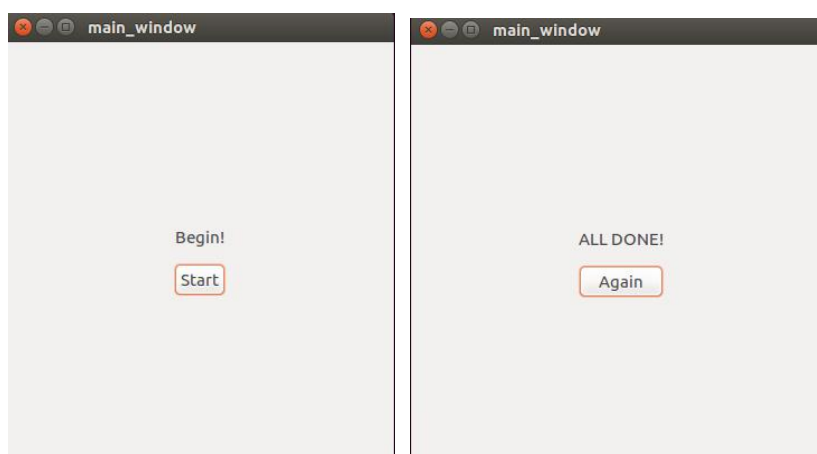


图 1.8 按钮交互前后的主进程窗口显示

1.4.2 实验调试及心得

调试时注意以下几点：

1. 在打开拷贝目标文件时，O_CREAT 模式下一定要加上 mode 权限参数，即使用 open 的第二种函数原型。权限参数采用 4 位十进制表示权限，最高位为 0，后 3 位数字依次表示用户、组用户以及其他用户的权限。数字解释为二进制，从高位到低位依次表示为读、写、执行权限，对应位为 1 时表示具有该权限。如 0600 权限表示仅用户具有文件的读取和写入权限。

2. 由于系统不自带 GTK+ 库，故在编译时提示库缺失错误。使用 `sudo apt-get install libgtk2.0-dev` 进行 GTK+2.0 图形库下载。使用 `pkg-config --modversion gtk+-2.0` 查看 GTK+2.0 图形库版本。

3. 在 GTK+ 编程时，注意类型的转换。此时需要查询各函数原型传参类型，并将参数转换为对应类型，如回调函数需要进行 G_CALLBACK() 转换。另外，

在控件组装时，需要考虑层级关系，易将主控件套在嵌套控件中，需要仔细设计。在进行信息显示时，为方便查看信息内容，需要进行 sleep 延迟。

实验心得：

该实验让我熟悉 Linux 的编程环境，了解并掌握了 open、read、write 系统调用函数的使用以及 GTK+图形相关编程。

在通过 open、read、write 系统调用的使用过程中，将其与 fopen、fread、fwrite 这些封装函数对比，可以发现 open 返回的是文件描述符，而 fopen 返回的是一个文件指针。通过查找两者的区别可以发现，两者最大的区别在于缓冲的有无。fopen、fread、fwrite 为缓冲文件系统，当处理文件时，会自动在内存开辟缓冲区，将待处理数据读或写入到缓冲区后再进行后续操作，此时访存次数会降低。而 open、read、write 为非缓冲文件系统，每次进行系统调用时，都需要切换状态，开销比缓冲文件系统大。

而在 GTK+编程中，由于是第一次接触前端编写，感到十分新奇。通过设计窗口与交互展示并程序的运行时态，可视化程序的当前信息，这使得任何一位用户都能轻松获取信息，前端的编写对于操作系统的意义巨大。通过三个并行誊抄程序的 GTK+编程，让我感受到可视化的强大，以及操作系统前端设计也同样不易。

附录 实验代码

文件拷贝功能实验代码：

```
//Mycp.c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#define MAX_BUFFER 4096

int main(int argc, char *argv[]){
    int srcfd,dstfd;
    char buf[MAX_BUFFER];
    if(argc != 3){
        perror("please input correctly!\n");
        exit(1);
    }
    if(!strcmp(argv[1],argv[2])){
        perror("the two files are same!\n");
```

```

        exit(1);
    }
    if((srcfd=open(argv[1],O_RDONLY))== -1){
        perror("the source file open fail!\n");
        exit(1);
    }
    if((dstfd=open(argv[2],O_WRONLY|O_CREAT,0600))== -1){
        perror("the destination file open fail!\n");
        exit(1);
    }
    while (1)
    {
        ssize_t readsize,writesize;
        readsize = read(srcfd,buf,MAX_BUFFER);
        if(readsize == 0) break;
        else if(readsize == -1){
            perror("read file error!\n");
            exit(1);
        }
        else{
            writesize = write(dstfd,buf,readsize);
            if(writesize == -1){
                perror("write file error!\n");
                exit(1);
            }
        }
    }
    printf("copy done!\n");
    close(srcfd);
    close(dstfd);
    exit(0);
}

```

誊抄程序的多窗口显示实验代码：

```

//main.c
#include<stdio.h>
#include<stdlib.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<sys/sem.h>
#include<unistd.h>
#include<sys/wait.h>
#include<gtk/gtk.h>

```

```

#define SHMKEY 123    //readbuf 与 writebuf 进程之间的共享内存 key 值
#define SIZEKEY 321  //写进程向读进程传递的放置长度公共内存的 key 值
#define SEMKEY 111   //公共信号灯集 key 值
#define SHMNUM 10    //定义共享内存数

int shmid;
int sizeid;
int semid; //0 号指示 empty, 1 号指示 full
pid_t preadbuf, pwritebuf;
union semun{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};

GtkWidget *window;
GtkWidget *label;
GtkWidget *button;
GtkWidget *box;

void main_process(void){
    //创建共享内存组
    if((shmid = shmget(SHMKEY, 40960, IPC_CREAT|0666)) == -1){    //40960 = 4K * 10
        printf("get shared memory fail!\n");
        exit(1);
    }
    //用于读进程获取剩余的待写字节数
    if((sizeid = shmget(SIZEKEY, 4, IPC_CREAT|0666)) == -1){
        printf("get size shared memory fail! \n");
        exit(1);
    }
    //创建信号灯组
    if((semid = semget(SEMKEY, 2, IPC_CREAT|0666)) == -1){
        printf("get semaphore fail!\n");
        exit(1);
    }
    union semun sem_arg[2];
    sem_arg[0].val = SHMNUM;
    sem_arg[1].val = 0;
    if(semctl(semid, 0, SETVAL, sem_arg[0]) == -1){
        printf("initialize semaphroe 'empty' fail!");
        exit(1);
    }
    if(semctl(semid, 1, SETVAL, sem_arg[1]) == -1){
        printf("initialize semaphroe 'full' fail!");
    }
}

```

```

        exit(1);
    }
    //创建进程 readbuf 和 writebuf
    if ((pwritebuf = fork()) == 0) {        //写共享内存进程
        printf("p_getbuf create!\n");
        execl("./get",NULL);
    }
    else {
        if ((preadbuf = fork()) == 0) { //读共享内存进程
            printf("p_putbuf create!\n");
            execl("./put",NULL);
        }
        else{    //父进程
            //等待子进程结束
            wait(NULL);
            wait(NULL);
            //删除信号灯和共享内存组
            semctl(semid, 0, IPC_RMID, NULL);
            shmctl(shmid, IPC_RMID, 0);
            shmctl(sizeid, IPC_RMID, 0);
            gtk_label_set_text(GTK_LABEL(label),"ALL DONE!");
            gtk_button_set_label(GTK_BUTTON(button),"Again");
        }
    }
}

int main(int argc,char *argv[]){
    gtk_init(&argc, &argv);
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"main_window");
    g_signal_connect (G_OBJECT (window), "destroy",G_CALLBACK(gtk_main_quit), NULL);//关闭窗口操作
    gtk_container_set_border_width (GTK_CONTAINER (window), 150);

    box = gtk_vbox_new(FALSE,0);

    button = gtk_button_new_with_label ("Start");
    g_signal_connect (G_OBJECT (button), "clicked",G_CALLBACK (main_process), NULL);

    label = gtk_label_new("Begin!");
    gtk_box_pack_start (GTK_BOX(box),label,FALSE,FALSE,15);
    gtk_box_pack_start (GTK_BOX(box),button,FALSE,FALSE,0);
}

```

```

gtk_container_add (GTK_CONTAINER (window), box);
gtk_widget_show_all (window);

gtk_main ();
return 0;
}

```

```

//get.c
#include<stdio.h>
#include<stdlib.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<sys/sem.h>
#include<unistd.h>
#include<gtk/gtk.h>

#define SHMKEY 123    //readbuf 与 writebuf 进程之间的共享内存 key 值
#define SIZEKEY 321  //写进程向读进程传递的放置长度公共内存的 key 值
#define SEMKEY 111   //公共信号灯集 key 值
#define SHMNUM 10    //定义共享内存组数
#define SHMSIZE 4096 //共享内存一组为 4K

int shmid;
int sizeid;
int semid; //0 号指示 empty, 1 号指示 full
char *bufaddr;
int *sizeaddr;

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;
}

```

```

char buf[1000];
GtkWidget *label;
GtkWidget *window;
GtkWidget *button;
pthread_t p1;

void *getbuf(void){

    if((shmid = shmget(SHMKEY, 40960, IPC_CREAT|0666)) == -1){
        printf("get: get shared memory fail!\n");
        exit(1);
    }
    bufaddr = shmat(shmid,NULL,0);
    if((sizeid = shmget(SIZEKEY, 4, IPC_CREAT|0666)) == -1){
        printf("get: get size shared memory fail! \n");
        exit(1);
    }
    sizeaddr = shmat(sizeid,NULL,0);
    if((semid = semget(SEMKEY, 2, IPC_CREAT|0666)) == -1)
    {
        printf("get: get semaphore fail!\n!");
        exit(1);
    }

    FILE *fp = fopen("text.txt","rb");
    if(fp == NULL) printf("open 'text.txt' fail\n");
    fseek(fp,0L,SEEK_END);
    int FileLen = ftell(fp); //计算文件长度
    rewind(fp);
    printf("fileLength: %d\n",FileLen);
    int ndnum;
    if(FileLen % SHMSIZE == 0){ //计算所需总缓冲区数
        ndnum = FileLen/SHMSIZE;
    }
    else ndnum = FileLen/SHMSIZE + 1;
    printf("need shared num: %d\n",ndnum);

    int i = 0;
    char *now_in;
    while(1){
        P(semid,0);
        now_in = bufaddr + (i % SHMNUM) * SHMSIZE;
        fread(now_in,1,SHMSIZE,fp);

```

```

printf("now get buf num: %d\n",i + 1);

sprintf(buf,"now get buf num: %d",(int)(i+1));
sleep(1); //方便看清过程变化
gtk_label_set_text(GTK_LABEL(label),buf); //修改记数

if(i + 1 == ndnum){
    *(now_in + SHMSIZE - 3) = 'E';
    *(now_in + SHMSIZE - 2) = 'O';
    *(now_in + SHMSIZE - 1) = 'F';
    *(sizeaddr) = FileLen % SHMSIZE;
    V(semid,1);
    break;
}
i++;
V(semid,1);
}
printf("Getebuf done!\n");
}

int main (int argc, char *argv[])
{
    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);// 生成一个样式为
GTK_WINDOW_TOPLEVEL 的窗口
    gtk_window_set_title(GTK_WINDOW(window),"get_window");//设置窗口标题
    gtk_window_set_default_size(GTK_WINDOW(window), 500, 400);//设置窗口默认大小
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);//设置窗口在显示器中
的位置为居中

    g_signal_connect(G_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);

    label=gtk_label_new("Start!");//创建标签
    int ret=pthread_create(&p1,NULL,&getbuf,NULL);
    gtk_container_add(GTK_CONTAINER(window), label); // 将按钮放在布局容器里

    gtk_widget_show(label);//显示标签
    gtk_widget_show(window);//显示生成的这个窗口
    gtk_main();//进入消息循环

    return 0;
}

```



```

//put.c
#include<stdio.h>
#include<stdlib.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<sys/sem.h>
#include<unistd.h>
#include<gtk/gtk.h>

#define SHMKEY 123    //readbuf 与 writebuf 进程之间的共享内存 key 值
#define SIZEKEY 321  //写进程向读进程传递的放置长度公共内存的 key 值
#define SEMKEY 111   //公共信号灯集 key 值
#define SHMNUM 10    //定义共享内存组数
#define SHMSIZE 4096 //共享内存一组为 4K

int shmid;
int sizeid;
int semid; //0 号指示 empty, 1 号指示 full
char *bufaddr;
int *sizeaddr;

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;
}

char buf[1000];
GtkWidget *label;
GtkWidget *window;
GtkWidget *button;
pthread_t p1;

```

```

void *putbuf(void){
    if((shmid = shmget(SHMKEY, 40960, IPC_CREAT|0666)) == -1){
        printf("putbuf: get shared memory fail!\n");
        exit(1);
    }
    bufaddr = shmat(shmid,NULL,0);
    if ((sizeid = shmget(SIZEKEY, 4, IPC_CREAT|0666)) == -1){
        printf("putbuf: get size shared memory fail! \n");
        exit(1);
    }
    sizeaddr = shmat(sizeid,NULL,0);
    if ((semid = semget(SEMKEY, 2, IPC_CREAT|0666)) == -1)
    {
        printf("putbuf: get semaphore fail!\n!");
        exit(1);
    }

    FILE *fp = fopen("text-copy.txt","w");
    int j = 0;
    char *now_out;
    while(1){
        P(semid,1);
        now_out = bufaddr + (j % SHMNUM) * SHMSIZE;
        if(*(now_out + SHMSIZE - 3) == 'E' && *(now_out + SHMSIZE - 2) == 'O' && *(now_out +
SHMSIZE - 1) == 'F'){
            sprintf(buf,"now put buf num: %d",(int)(j+1));
            sleep(1); //方便看清过程变化
            gtk_label_set_text(GTK_LABEL(label),buf); //修改记数
            printf("now put buf num: %d\n",j + 1);
            if(*(sizeaddr) == 0){
                fwrite(now_out,1,SHMSIZE,fp);
            }
            else{
                fwrite(now_out,1,*(sizeaddr),fp);
            }
            V(semid,0);
            break;
        }
        sprintf(buf,"now put buf num: %d",(int)(j+1));
        sleep(1); //方便看清过程变化
        gtk_label_set_text(GTK_LABEL(label),buf); //修改记数

        printf("now put buf num: %d\n",j + 1);
    }
}

```

```

        fwrite(now_out,1,SHMSIZE,fp);
        j++;
        V(semid,0);
    }
    printf("Putbuf done!\n");
}

int main (int argc, char *argv[])
{
    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);// 生成一个样式为
GTK_WINDOW_TOPLEVEL 的窗口
    gtk_window_set_title(GTK_WINDOW(window),"get_window");//设置窗口标题
    gtk_window_set_default_size(GTK_WINDOW(window), 500, 400);//设置窗口默认大小
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);//设置窗口在显示器中
的位置为居中

    g_signal_connect(G_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);

    label=gtk_label_new("Start!");//创建标签
    int ret=pthread_create(&p1,NULL,&putbuf,NULL);
    gtk_container_add(GTK_CONTAINER(window), label); // 将按钮放在布局容器里

    gtk_widget_show(label);//显示标签
    gtk_widget_show(window);//显示生成的这个窗口
    gtk_main();//进入消息循环

    return 0;
}

```

2 实验二 掌握添加系统调用的方法

2.1 实验目的

- (1) 掌握 Linux 操作系统的系统调用实现过程。
- (2) 掌握在 Linux 环境下编译内核的方法。
- (3) 熟悉 Linux 操作系统内与系统调用相关的内核文件。

2.2 实验内容

- (1) 添加一个新的系统调用，实现文件拷贝功能。
- (2) 采用编译内核的方法，将其增加进内核源码并编译内核。
- (3) 编写一个应用程序，测试新加的系统调用。

2.3 实验设计

2.3.1 开发环境

操作系统：Ubuntu 16.04 LTS

系统类型：64 位

系统内核：Linux version 4.14.267

处理器：Intel® Core™ i5-8250U CPU @ 1.60GHz × 4

内存：3.8GB

编译器及版本：GCC version 5.4.0

2.3.2 实验设计

该实验需要设计系统调用函数，并将其编译进内核中，通过测试程序系统调用增加的调用函数，验证其正确性。

对于文件拷贝功能系统调用函数的编写，需要与实验一进行区分。实验一中的文件拷贝功能运行时的初始状态是在用户态，状态的切换是在调用 `open`、`read`、`write` 等系统调用函数时通过访管中断进行的。而此次实验则需要编写一个系统调用，此时程序运行于管态，故所需的打开、读、写操作同样需要内核提供的函

数实现，如 `sys_open`、`sys_read`、`sys_write` 等。拷贝功能系统调用的架构流程图如图 2.1 所示。

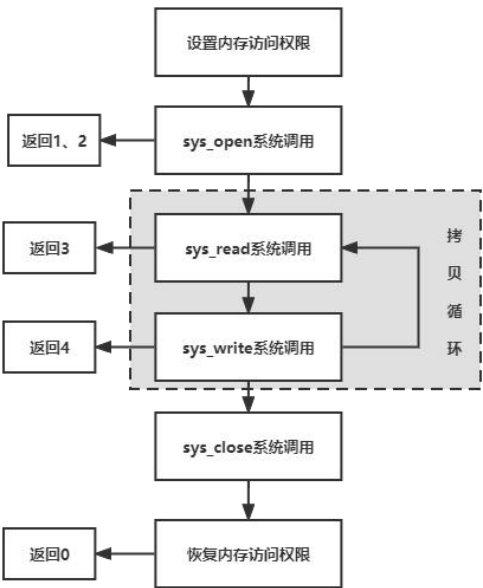


图 2.1 拷贝功能系统调用架构流程图

其中，存在内存访问权限的设置和恢复环节。使用 `get_fs()` 与 `set_fs(get_ds())`；设置内存访问权限大小为 `KERNEL_DS`。`get_fs` 用于获取当前权限状态，存放到 `mm_segment_t` 类型的 `old_fs` 数据中。`get_ds` 获取常量 `KERNEL_DS`，通过 `set_fs` 进行设置，目的是将权限状态设置为内核权限。在内核中使用 `sys_open`、`sys_write` 等系统调用时，存在参数来自用户态，如系统调用 `sys_write` 时传递的 `buf` 参数。为了保护内核空间，一般会用 `get_fs()` 得到的值来和 `USER_DS` 进行比较，从而防止用户空间程序“蓄意”破坏内核空间。为了解决这一问题，使用 `set_fs(KERNEL_DS)` 将其能访问的空间限制扩大到 `KERNEL_DS`，从而在内核内使用需要用到用户态数据的系统调用。在功能完成后，系统调用 `sys_close` 关闭源文件与目标文件，随后将开始存放的原始 `fs` 状态恢复，即 `set_fs(old_fs)`。

测试新增系统调用则通过 `syscall` 来进行对应系统调用程序的测试，通过指定系统调用号，如新增加的系统调用号为 334，设置传递参数，如源文件名和目标文件名 `"text.txt"`、`"text-copy.txt"`，进行测试。测试指令为 `syscall(334, "text.txt", "text-copy.txt")`；，将返回参数进行输出，当返回参数为 0 且拷贝生成目标文件时，表明拷贝功能系统调用添加成功。

2.4 实验调试

2.4.1 实验步骤

1. 在 Linux 官网下载内核压缩包 linux-4.14.267.tar.xz，使用指令 `tar -xvf linux-4.14.267.tar.xz` 解压缩，并将其移入 `/usr/src` 目录下，使用指令 `sudo mv linux-4.14.267 /usr/src`。由于 `/usr/src` 属于内核文件，故需要使用 `sudo` 获取权限。

2. 修改目标内核的系统调用相关文件。首先修改系统调用号表，路径为 `/usr/src/linux-4.14.267/arch/x86/entry/syscalls/syscall_64.tbl`。使用 `sudo` 权限，在表中添加系统调用和系统调用号的映射，如图 2.2 所示。

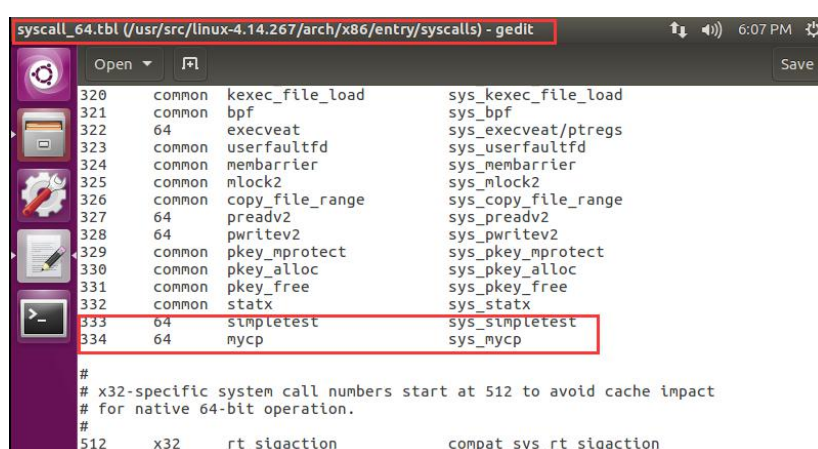


图 2.2 新增系统调用号

其中，`sys_simpletest` 为一个简单的系统调用，`sys_mycp` 为拷贝功能系统调用，分别设置系统调用号为 333 与 334。由于操作系统为 64 位机，可将 ABI 类型设置为 64。

3. 在系统调用声明文件中，路径为 `/usr/src/linux-4.14.267/include/linux/syscalls.h`，添加系统调用函数声明，如图 2.3 所示。

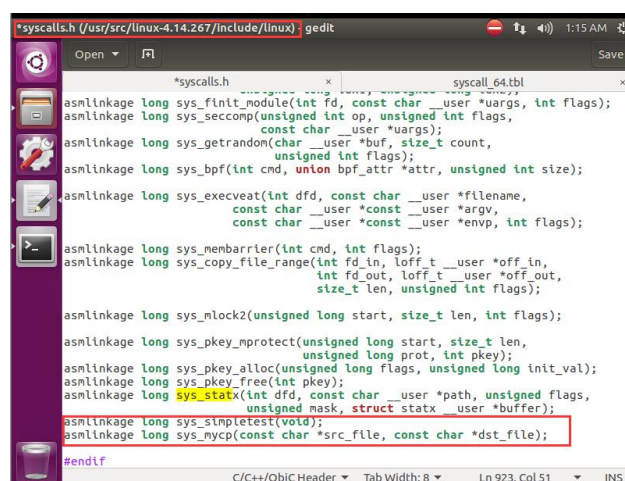


图 2.3 新增系统调用函数声明

其中，`asm linkage` 修饰符表示告知编译器不通过寄存器传递参数，而使用局部栈传参。在系统调用时，用户态的寄存器会压栈保护，相应的相关参数的获取也需要通过局部栈获取，故使用 `asm linkage` 进行限制。另外，函数声明需要放置在 `#endif` 前。

4. 在系统调用实现文件中，路径为 `/usr/src/linux-4.14.267/kernel/sys.c`，添加系统调用实现，如图 2.4 所示。

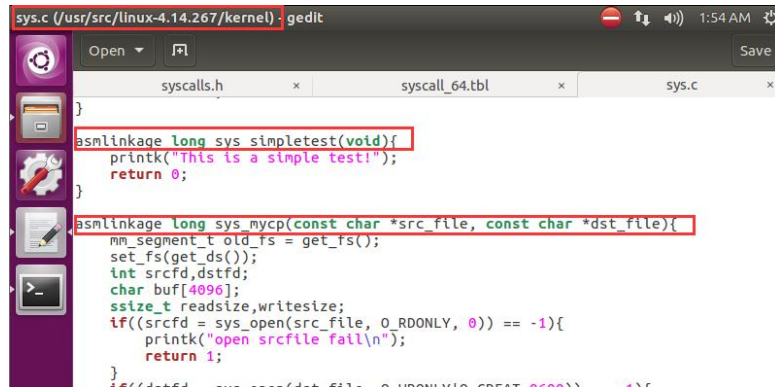


图 2.4 新增系统调用函数定义

5. 在 `make menuconfig` 前，将原内核中的 `.config` 复制到目标内核中，保持原始相关配置。在 `/usr/src/linu-4.14.267` 目录下，使用 `sudo make menuconfig` 指令，进入配置界面，如图 2.5 所示。配置过程中依次执行 `<Load><OK>`、`<Save><OK>`、`<Exit><Exit>`，使用原内核配置，放置原内核配置丢失。

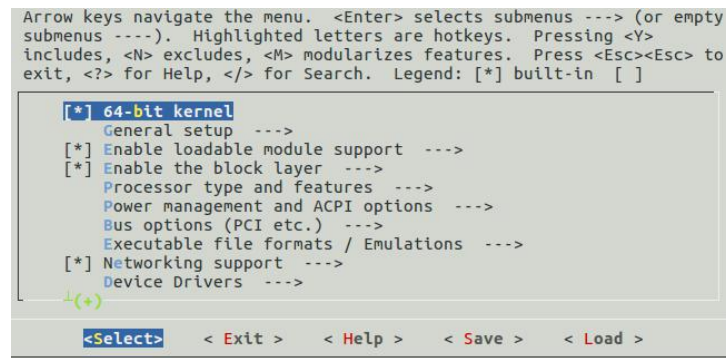


图 2.5 config 配置界面

6. 使用指令 `sudo make -j8 bzImage` 编译启动镜像。在编译过程中，提示缺失文件时，通过 `sudo apt-get install *` 进行对应文件下载后重新编译。指令中 `-j8` 表示 8 线程加速。

7. 使用 `sudo make -j8 modules` 编译模块。编译成功后，使用指令 `sudo make modules_install` 安装模块。

8. 使用指令 `sudo make install` 安装内核。

9. 重启系统，在进入界面按下 `Shift` 按键进入 `grub` 选择界面，如图 2.6 所示。选择目标内核 `linux 4.14.267`。

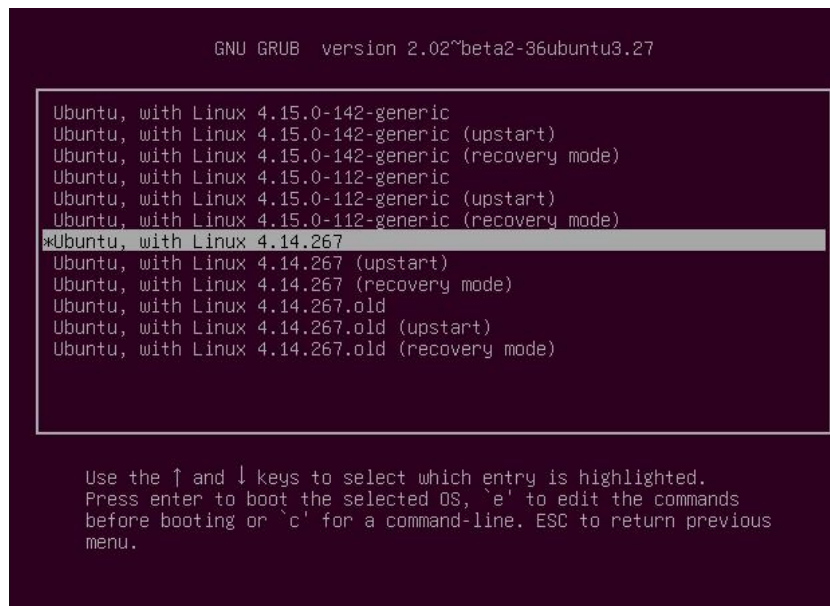


图 2.6 grub 选择界面

10. GCC 编译测试文件 test.c, 对 sys_simpletest 和 sys_mycp 系统调用进行测试, 分别为 syscall(333);与 syscall(334, "text.txt", "text-copy.txt");, 并输出对应的返回值。系统调用返回成功值如图 2.7 所示, 拷贝文件对比如图 2.8 所示。

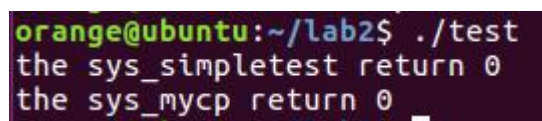


图 2.7 系统调用返回成功值

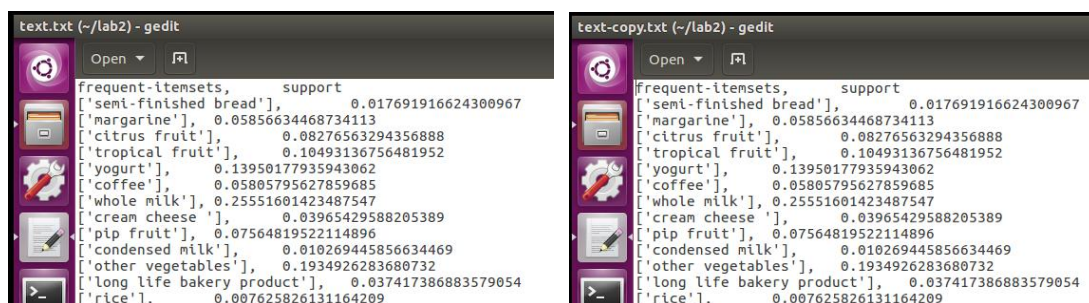


图 2.8 拷贝文件成功

2.4.2 实验调试及心得

调试时注意以下几点:

1. 在进行各种编译时, 会产生缺失文件报错, 此时应该仔细阅读报错信息, 通过搜索报错对应文件来安装相应文件。安装指令为 `sudo apt-get install *`。如在进行 `make menuconfig` 时, 提示错误 `Unable to find the ncurses libraries or the`

required header files. 'make menuconfig' requires the ncurses libraries. Install ncurses (ncurses-devel) and try again. , 根据提示安装 ncurses libraries, 指令为 `sudo apt-get install libncurses*`。其他文件缺失提示错误的处理方式相似。

2. 在进行编译时, 由于有中间文件的生成, 需要预留足够的空间。当提供 40GB 存储时, 在编译过程中会出现存储空间不足报错。故需要开辟更大空间, 当提供 45GB 存储时, 可以容纳编译中间文件。

实验心得:

通过此次实验, 让我掌握了系统调用在内核中的实现, 熟悉了编译内核的操作方式。

在实现系统调用服务时, 由于是在管态的缘故, 所使用的模块函数也必须是内核函数, 如 `sys_open`、`printk` 等。同时, 由于设计到文件操作的系统调用, 需要使用 `get_fs` 与 `set_fs`、`get_ds()` 进行内存访问限制的切换。由于系统调用参数存在来自内核的数据, 如 `sys_write` 中的 `buf`, 在进行编译时会将限制状态与用户态限制常量相比较, 由此防止用户程序篡改内核数据。因此需要将当前限制设置为内核限制, 使用 `get_ds()` 或直接使用常量 `KERNEL_DS` 获取内核限制。`set_fs` 用于设置, 而 `get_fs` 用于保存最初状态, 在结束系统调用时恢复。

另外, 在添加系统调用时, 需要对各个文件进行对应的添加, 添加位置也十分重要, 如若位置错误, 也会导致编译失败。

在进行内核编译时, 需要的时间较长, 而且有时因为多线程的缘故导致错误无法及时发现。所以在第一次编译时, 最好在不修改内核的前提下先试着编译一次, 将所需文件都安装妥当后, 再修改内核的系统调用相关文件。这样查错十分容易, 可将错误逐级解决。若一开始就做修改, 则很难判断是程序错误还是文件缺失错误。

附录 实验代码

拷贝功能系统调用实验代码:

```
//sys_mycp.c
/*系统调用号: 334*/

asmlinkage long sys_mycp(const char *src_file, const char *dst_file);

asmlinkage long sys_mycp(const char *src_file, const char *dst_file){
    mm_segment_t old_fs = get_fs();
    set_fs(get_ds());
    int srcfd, dstfd;
```

```

char buf[4096];
ssize_t readsize,writesize;
if((srcfd = sys_open(src_file, O_RDONLY, 0)) == -1){
    printk("open srcfile fail\n");
    return 1;
}
if((dstfd = sys_open(dst_file, O_WRONLY|O_CREAT,0600)) == -1){
    printk("open dstfile fail\n");
    return 2;
}
while(1){
    readsize = sys_read(srcfd,buf,4096);
    if(readsize == 0) break;
    else if(readsize == -1){
        printk("read file error\n");
        return 3;
    }
    else{
        writesize = sys_write(dstfd,buf,readsize);
        if(writesize == -1){
            printk("write file error\n");
            return 4;
        }
    }
}
sys_close(srcfd);
sys_close(dstfd);
printk("copy done!\n");
set_fs(old_fs);
return 0;
}

```

简单系统调用实验代码：

```

//sys_simpletest.c
/*系统调用号： 333*/

asm linkage long sys_simpletest(void);

asm linkage long sys_simpletest(void){
    printk("This is a simple test!");
    return 0;
}

```

测试文件实验代码：

```
//test.c
#include <unistd.h>
#include <stdio.h>
#include <sys/syscall.h>

int main(){
    int ret = syscall(333);
    printf("the sys_simpletest return %d\n",ret);

    ret = syscall(334, "text.txt", "text-copy.txt");
    printf("the sys_mycp return %d\n",ret);

    return 0;
}
```

3 实验三 掌握添加设备驱动程序的方法

3.1 实验目的

- (1) 掌握 Linux 环境下，通过动态模块加载方式增加设备驱动。
- (2) 掌握与 Linux 设备驱动相关的内核文件。
- (3) 掌握 Linux 内核模块结构。

3.2 实验内容

- (1) 增加一个新的设备驱动程序，实现字符设备的驱动。演示简单字符键盘缓冲区或一个内核单缓冲区。
- (2) 采用动态模块加载方式加载字符设备驱动程序。
- (3) 编写测试程序进行测试。

3.3 实验设计

3.3.1 开发环境

操作系统：Ubuntu 16.04 LTS

系统类型：64 位

系统内核：Linux version 4.14.267

处理器：Intel® Core™ i5-8250U CPU @ 1.60GHz × 4

内存：3.8GB

编译器及版本：GCC version 5.4.0

3.3.2 实验设计

编写设备驱动程序，即是向上层用户程序隐藏设备的具体细节，通过定义功能函数来实现透明性，相当于填充一致的文件系统接口 `file_operations`。由此，设备驱动程序将设备映射为特殊的设备文件，当用户程序对设备进行操作时，通过接口函数直接对设备文件处理即可。

设计字符设备驱动程序，演示内核单缓冲区的使用从 3 个环节入手，如图 3.1

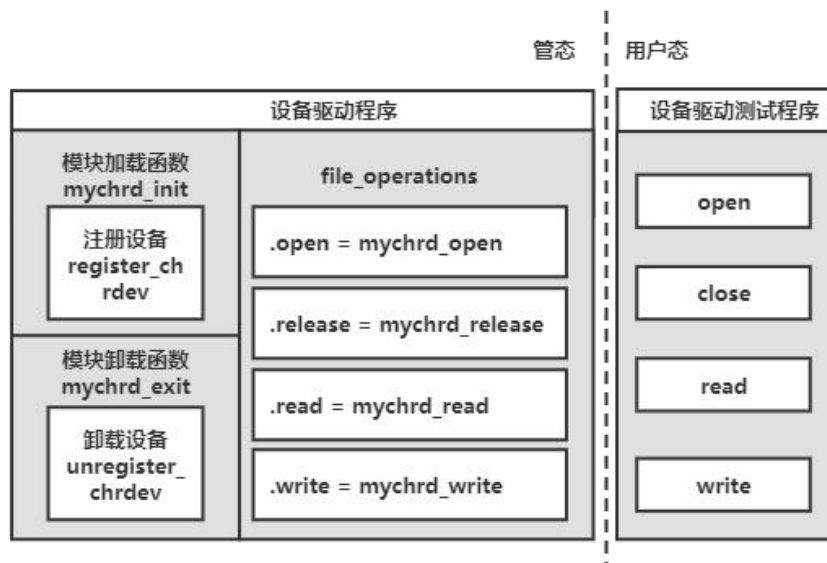


图 3.1 字符设备驱动程序与测试程序模块设计图

所示。其中，需要设计两个程序分别为设备驱动程序、设备驱动测试程序。设备驱动程序中，包含 3 个框架设计，分别为模块加载函数 mychrd_init、模块卸载函数 mychrd_init 与 file_operations 各个函数域的填充。file_operations 中填充 4 个基础功能函数，即 open、release、read 与 write，对应为 mychrd_open、mychrd_release、mychrd_read 与 mychrd_write。而设备驱动测试程序则是对 4 个基础函数进行测试。值得注意的是，两个程序运行所处状态不同，而要使得数据互通，则需要通过 copy_to_user 与 copy_from_user 两个内核函数在设备驱动程序中实现，使用户程序保持设备驱动的透明性。

在模块加载函数的设计中，使用 register_chrdev 注册函数，函数原型为 `int __register_chrdev(unsigned int major, unsigned int baseminor, unsigned int count, const char *name, const struct file_operations *fops);`，其中各参数含义为 major 为主设备号，为 0 时，调用注册函数为动态注册，其他情况为静态注册的设备号，使用动态分配设备号要更优，避免静态设备号冲突。baseminor 为次设备号，从 0 开始。count 为次设备号范围，name 为设备名，fops 为文件系统接口指针。由于是动态分配，函数返回一个主设备号。

在模块卸载函数的设计中，使用 unregister_chrdev 注销函数，函数原型为 `int __unregister_chrdev(unsigned int major, const char *name);`，卸载设备时只需要设备号与设备名即可，此设备号即在模块加载函数中获取的主设备号。当该模块没有被引用时，方可卸载。

设置好模块加载和卸载函数后，进行模块宏定义，`module_init(mydriver_init);` 与 `module_exit(mydriver_exit);`。最后加上 `MODULE_LICENSE("GPL");`；模块的许可声明，"GPL" 指明了 GNU General Public License 的任意版本。

接着填充 file_operations 文件系统接口的各个域。

.open 填充函数原型为 `static int mychrdr_open(struct inode *inode, struct file *file);`。首先添加互斥锁，由于设备同时只能被一个用户程序所用，故设置互斥锁 `open_mutex`。当 `open_mutex` 为 0 时，进行 `open_mutex` 自增上锁。使用 `try_module_get(THIS_MODULE);` 打开模块，将该模块的引用计数加一。

.release 填充函数原型为 `static int mychrdr_release(struct inode *inode, struct file *file);`，`open_mutex` 自减去锁，使用 `module_put(THIS_MODULE);` 将该模块的引用计数减一。

.read 填充函数原型为 `static ssize_t mychrdr_read(struct file *file, char __user *buf, size_t size, loff_t *f_pos);`，`buf` 为用户态的指针。此时设置内核单缓冲区 `buffer_k` 为静态全局变量。要使得内核数据传递到用户，需要使用 `copy_to_user` 内核函数，函数原型为 `unsigned long copy_to_user(void *to, const void *from, unsigned long n);`，其中 `to` 为用户空间目标地址，`from` 为内核空间源地址，`n` 为传递数据字节数。调用为 `copy_to_user(buf, buffer_k, sizeof(buffer_k));`。返回值为 0 时表示传递成功，否则返回传递失败的数据字节数。

.write 填充函数原型为 `static ssize_t mychrdr_write(struct file *file, const char __user *buf, size_t size, loff_t *f_pos);`，`buf` 为用户态的指针。要使得用户数据传递到内核，需要使用 `copy_from_user` 内核函数，函数原型为 `unsigned long copy_from_user(void *to, const void *from, unsigned long n);`，其中 `to` 为内核空间目标地址，`from` 为用户空间源地址，`n` 为传递数据字节数。调用为 `copy_from_user(buffer_k, buf, sizeof(buf));`。同样的，返回值为 0 时表示传递成功，否则返回传递失败的数据字节数。

在测试程序中，需要先打开该设备，其实可将其视作打开文件。`open("/dev/mychrdriver", O_RDWR|O_NOCTTY|O_NONBLOCK);`，返回文件描述符。首先调用 `read` 系统调用读取该设备内核缓冲区的数据。再调用 `write` 向内核缓冲区写入数据，再通过 `read` 读取，查看数据是否缓存成功。

3.4 实验调试

3.4.1 实验步骤

1. 在设计并编写好设备驱动程序后，进行模块添加。首先编写 `makefile`，如图 3.2 所示。`makefile` 需与待编译文件处于同一目录下。
2. 在该目录下进行编译，使用指令 `sudo make`。生成 `mychrdriver.ko` 驱动模块文件。
3. 使用 `insmod` 来显式加载核心模块，指令为 `sudo insmod mychrdriver.ko`。

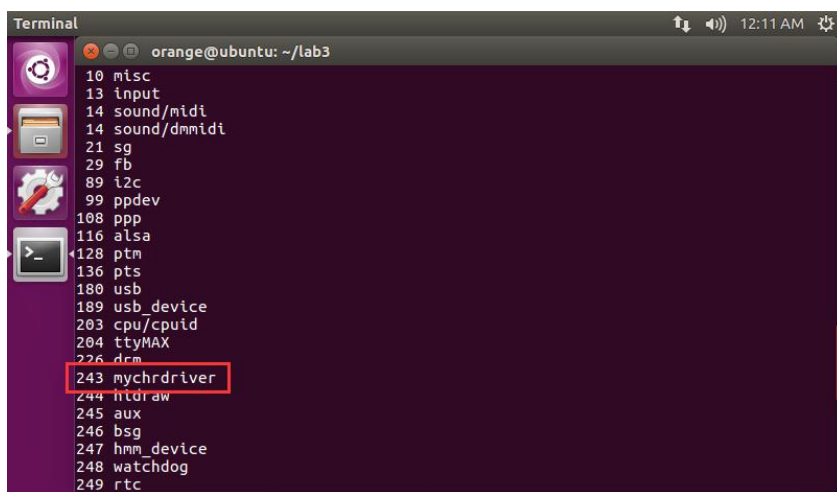
```

*Makefile - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
ifneq ($(KERNELRELEASE),)
#kbuild syntax.
#模块的文件组成
#mymodule-objs := mychrdriver.o
#生成的模块文件名
obj-m := mychrdriver.o
else
PWD := $(shell pwd)
KVER := $(shell uname -r)
KDIR := /lib/modules/$(KVER)/build
all:
$(MAKE) -C $(KDIR) M=$(PWD)
clean:
# rm -f *.cmd *.o *.mod *.ko
rm -rf *.cmd *.o *.mod.c *.ko .tmp_versions
# $(MAKE) -C $(KDIR) M=$(PWD) clean
endif

```

图 3.2 makefile 文件

4. 使用 `cat /proc/devices` 查看动态分配的设备号，如图 3.3 所示。主设备号为 243。



```

Terminal
orange@ubuntu: ~/lab3
10 misc
13 input
14 sound/midi
14 sound/dmide
21 sg
29 fb
89 i2c
99 ppdev
108 ppp
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
203 cpu/cpuid
204 ttyMAX
226 dcm
243 mychrdriver
244 ltrdraw
245 aux
246 bsg
247 hrm_device
248 watchdog
249 rtc

```

图 3.3 查看动态分配设备号

5. 创建设备文件，使用指令 `sudo mknod /dev/mychrdriver c 243 0`，`mknod` 指令后的参数分别为设备文件名、设备类型、主设备号、次设备号。其中设备类型 `c` 表明创建设备为字符设备，主设备号为上述动态分配的设备号，次设备号从 0 开始。查看设备文件是否添加成功，在 `/dev` 目录下找到 `mychrdriver` 文件，添加成功。

6. 进行测试，编译测试文件 `driver_test.c`，运行程序 `sudo ./driver_test`，运行结果如图 3.4 所示。

```

orange@ubuntu:~/lab3$ sudo ./driver_test
The device initial information: the initial information.
please input your information:
ABC
write done.
ABCthe devbuf info is "ABC".

```

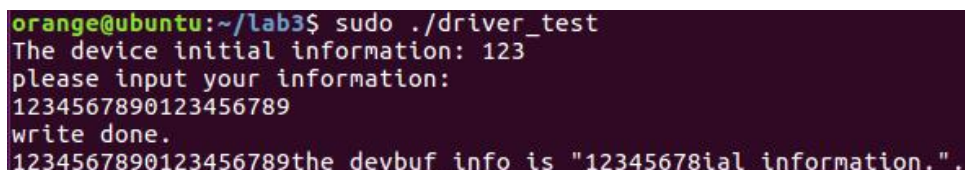
图 3.4 测试结果截图

显示成功内核缓冲区数据。当输入待存放数据“ABC”后，读取数据为 ABC。

3.4.2 实验调试及心得

调试时注意以下几点：

1. 在进行设备驱动程序编写时，涉及到数据在用户空间与内核空间的传递。需要使用到 `copy_to_user` 与 `copy_from_user` 函数，若把参数传递错误，则会产生数据显示问题，如图 3.5 所示。



```
orange@ubuntu:~/lab3$ sudo ./driver_test
The device initial information: 123
please input your information:
1234567890123456789
write done.
1234567890123456789the devbuf info is "12345678ial information."
```

图 3.5 错误缓冲区读取显示

原因在于 `copy_to_user` 与 `copy_from_user` 中 `n` 参数的设置，即传递数据字节数。需要将其设置成 `sizeof(from)`，即源数据长度。因此，对于参数的意义的理解要十分注意。

2. 若字符设备驱动程序编写错误，则需要 `/dev` 下使用指令 `sudo rmmod mychrdriver` 来卸载设备。而不是一味的将其在目录处移除，这样并未达到卸载设备的目的。

实验心得：

通过此次实验，让我掌握了使用动态模块加载方式来增加设备驱动，设备文件的加载与卸载，以及用户数据与内核数据的互通。

通过设计设备驱动程序，明白了设备透明化的原理。则是通过设置同一接口，在统一的结构下填写函数域来完成设备驱动功能。用户通过设备驱动程序提供的接口函数，间接使用对应的物理设备。在用户层面，仅将其视为一个设备文件，通过对设备文件进行 `open`、`read`、`write` 等操作，即可实现相应的设备功能。

用户态数据与管态数据的互通，也同样需要系统调用来实现。因为用户态数据范围权限要小于管态数据范围权限，因此在检测时若有用户态数据填入内核会进行报错。故系统调用在内核中的用处无疑是巨大的，倘若没有系统调用，那么扩展内核将举步维艰。

与实验二进行对比，不难发现通过增加模块的方式对内核进行扩展比重新编译内核要方便很多，这也是将内核模块化的用心之处。

另外，通过此次实验，填补了许多操作系统相关知识的空白，通过查阅书籍、查找资料了解了有关系统调用，设备管理和内核文件的诸多知识，明白了只有不断实践才能有进步的道理。

附录 实验代码

设备驱动程序实验代码:

```
//mychrdriver.c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/uaccess.h>

#define MAIN_DEVICE_NUM 0 //为 0 时，调用注册函数为动态注册；其他情况为静态注册的设备号
#define DEVNAME "mychrdriver"

static int mdevnum = 0;
static char buffer_k[4096] = "the initial information."; //不用 buf 命名因为与 read、write 中要有区分
static int open_mutex = 0;

static int mychr_open(struct inode *inode, struct file *file);
static int mychr_release(struct inode *inode, struct file *file);
static ssize_t mychr_read(struct file *file, char __user *buf, size_t size, loff_t *f_pos);
static ssize_t mychr_write(struct file *file, const char __user *buf, size_t size, loff_t *f_pos);

static struct file_operations mychr_fops = {
    .open = mychr_open,
    .release = mychr_release,
    .read = mychr_read,
    .write = mychr_write
};

static int mychr_open(struct inode *inode, struct file *file){ //驱动子函数 open
    printk("<1>the main device number is %d, the secondary device number is %d.\n", MAJOR(inode->i_rdev), MINOR(inode->i_rdev)); //查看主次设备号
    if(open_mutex){ //进程互斥
        printk("<1>this char device is busy.\n");
        return -1;
    }
    else{
        open_mutex++;
        try_module_get(THIS_MODULE); //打开模块
        printk("<1>open this device successfully.\n");
    }
    return 0;
}
```

```

static int mychrd_release(struct inode *inode, struct file *file){ //驱动子函数 release
    open_mutex--;
    module_put(THIS_MODULE);
    printk("<1>release this device successfully.\n");
    return 0;
}

static ssize_t mychrd_read(struct file *file, char __user *buf, size_t size, loff_t *f_pos){ //驱动子函数 read
    int result = copy_to_user(buf, buffer_k, sizeof(buffer_k));
    if(result != 0){
        printk("<1>read error.\n");
        return -1;
    }
    printk("<1>read success.\n");
    return 0;
}

static ssize_t mychrd_write(struct file *file, const char __user *buf, size_t size, loff_t *f_pos){ //驱动子函数
write
    int result = copy_from_user(buffer_k, buf, sizeof(buf));
    if(result != 0){
        printk("<1>write error.\n");
        return -1;
    }
    printk("<1>write success.\n");
    return 0;
}

static int __init mychrd_init(void){ //模块加载函数
    printk("<1>init my char device.\n");
    int result = register_chrdev(MAIN_DEVICE_NUM, DEVNAME, &mychrd_fops);
    if(result < 0){
        printk("<1>register fail.\n");
        return -1;
    }
    if(mdevnum == 0){
        mdevnum = result;
        printk("<1>register done.\n");
        printk("<1>the main device number is %d.\n", mdevnum);
    }
    return 0;
}

```

```

static void __exit mychrd_exit(void){    //设备卸载函数
    unregister_chrdev(mdevnum, DEVNAME);
    printk("<1>unregister done.\n");
    printk("<1>after unregister,the main device number is %d.\n",mdevnum);
    return 0;
}

module_init(mychrd_init);
module_exit(mychrd_exit);

MODULE_LICENSE("GPL"); //模块许可

```

设备驱动测试程序:

```

//driver_test.c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(){
    char rdbuf[4096], wrbuf[4096];
    int fd = open("/dev/mychrdriver",O_RDWR|O_NOCTTY|O_NONBLOCK);
    if(fd < 0){
        perror("open device fail.\n");
        return -1;
    }
    ssize_t readret = read(fd,rdbuf,sizeof(rdbuf));
    if(readret < 0){
        perror("read devinfo error.\n");
        return -1;
    }
    printf("The device initial information: %s\n", rdbuf); //读取设备初始化信息

    printf("please input your information:\n");
    gets(wrbuf); //写入设备缓冲区的信息
    ssize_t writeret = write(fd,wrbuf,sizeof(wrbuf));
    if(writeret < 0){
        perror("write devinfo error.\n");
        return -1;
    }
    printf("write done.\n");
    readret = read(fd,rdbuf,sizeof(rdbuf));
    if(readret < 0){

```

```
        perror("read devinfo error.\n");
        return -1;
    }
    printf("the devbuf info is \"%s\".\n",rdbuf);
    close(fd);
    return 0;
}
```

4 实验四 理解和分析/proc 文件

4.1 实验目的

- (1) 了解/proc 文件的特点和使用方法。
- (2) 了解系统监控器的实现方式。
- (3) 掌握 GTK/Qt 的编程模式。

4.2 实验内容

- (1) 读取 proc 文件系统，监控系统状态，获取系统各种信息。
- (2) 用图形界面显示系统监控状态，要求参照 Windows 的任务管理器，利用 GTK 或 Qt 实现图形界面编程。
- (3) 实现功能详细如下：获取并显示主机名；获取并显示系统启动的时间；显示系统到目前为止持续运行的时间；显示系统的版本号；显示 cpu 的型号和主频；通过 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；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；cpu 使用率的图形化显示（2 分钟内的历史记录曲线）；内存和交换分区（swap）使用率的图形化显示（2 分钟内的历史记录曲线）；在状态栏显示当前时间；在状态栏显示当前 cpu 使用率；在状态栏显示当前内存使用情况；用新线程运行一个其他程序；关机功能。

4.3 实验设计

4.3.1 开发环境

操作系统：Ubuntu 16.04 LTS

系统类型：64 位

系统内核：Linux version 4.15.0

处理器：Intel® Core™ i5-8250U CPU @ 1.60GHz × 4

内存：3.8GB

编译器及版本：GCC version 5.4.0

图形用户界面库及版本：GTK+ 2.0

4.3.2 实验设计

编写 GTK+2.0 程序实现系统监控器的图形化显示，需要了解系统监控器的所需功能，以及对应功能的实现模块和相关文件。



图 4.1 系统监控器功能模块图

系统监控器设计包含 5 个部分，分别为状态栏模块设计、系统信息显示模块设计、CPU 信息显示模块设计、内存和交换分区信息显示模块设计、进程处理模块设计。模块设计包含两个部分，界面布局设计和功能模块设计。设计系统监控器的功能模块设计界面如图 4.1 所示。

界面布局设计：

在整体界面布局架构设计中，通过笔记本控件 `notebook` 实现系统信息显示模块、CPU 信息显示模块、内存和交换分区信息显示模块和进程处理模块的组合。而状态栏则在系统监控器中持续显示，应将其放置于顶层控件内，实际设计为 3 个 `label` 文本标签。因此设计主窗口 `window` 下填充 `table` 表格布局容器控件，用来分配 `notebook` 和状态栏 `label` 文本标签的相对位置与大小。`notebook` 开辟 4 个页面。

1. 系统信息显示界面布局设计。提供系统信息信息，放置在 `scrolled_window` 滑动窗口中，而关机和重启功能，则分别通过两个按钮交互实现，"clicked" 点击信号传来时，运行各自的回调函数。

2. CPU 信息显示界面布局设计。在 `frame` 控件下填充 CPU 利用率可视化曲线图。绘图采用绘图区控件 `draw` 实现，初始化为 `cpuuse_draw = gtk_drawing_area_new()`。同样的，采用 "expose-event" 曝光事件信号来对绘图回调函数进行连接，由于曝光事件信号是自动发生的，故可实现每秒实时显示 CPU

的变化曲线。

3. 内存和 swap 信息显示界面布局设计。与 CPU 图形化界面相同，采用 "expose-event" 曝光事件信号来对相应的绘图回调函数进行连接。另外，为提供更多信息，在曲线图下提示相对应的详细信息，通过 label 文本标签显示。文本标签填充在 hbox 横盒控件中，实现横向排布。

4. 进程处理显示界面布局设计。包含信息显示和交互操作部分。信息显示顶层控件为 scrolled_window 滑动窗口便于填充超出窗口大小的文件。接着填充 clist 列表控件，用于生成表格显示各进程的 pid、进程名、状态、ppid、优先级以及占用内存大小。在交互操作部分中，包含 entry 信息输入栏控件，用于输入查询或终结进程的进程号以及运行进程的进程名等。提供 4 个功能按钮，Search 查找、Kill 终结、Refresh 刷新和 Run 运行。当点击对应按钮发起 "clicked" 点击信号后，进行回调函数的运行。

5. 状态栏显示界面布局设计。包含 3 个 label 文本标签。由于要实现实时的信息更新，使用 g_timeout_add 函数进行相关操作，传递刷新间隔时间、定时运行函数以及运行函数所需的参数。如实现时间显示功能的函数为 gboolean settime(gpointer data);，则对其进行定时运行，函数实现为 g_timeout_add(1000, settime, (gpointer)time_label);。

功能模块设计：

1. 状态栏功能模块设计。包含显示当前时间功能、显示当前 CPU 利用率功能以及当前内存使用情况功能。时间显示则未使用 proc 文件系统，而当前 CPU 利用率需要查看 /proc/stat 下的信息，内存利用率则查看 /proc/meminfo 信息。

在获取当前时间功能时，需要使用 time 文件。通过 time 函数获取 time_t 时间结构体，再由 localtime 将其转化为 tm 本地时间信息结构，从而获取日期时间信息。

使用 cat /proc/stat 指令查看 CPU 信息，如图 4.2 所示。其中一行信息的参数依次为 CPU 名称、用户态 CPU 时间、nice 值为负的进程所占用的 CPU 时间、核心时间、除硬盘 IO 等待时间以外其它空闲时间、硬盘 IO 等待时间、硬中断时间、软中断时间。实时 CPU 利用率计算可以采用两个时刻的 CPU 运行时间差除以时间间隔。计算运行时间差则可用两个时刻 CPU 总的时间差减去 CPU 空闲等待时间差计算。值得注意的是，时间单位为 jiffies，1jiffies=0.01s。

```
orange@ubuntu:~$ cat /proc/stat
cpu 7345 2266 10512 3983452 1351 0 344 0 0 0
cpu0 1829 174 2394 997155 317 0 127 0 0 0
cpu1 1784 1068 3087 995208 338 0 60 0 0 0
cpu2 1894 827 2636 994652 313 0 76 0 0 0
cpu3 1836 195 2393 996435 382 0 80 0 0 0
```

图 4.2 /proc/stat CPU 时间信息

使用 `cat /proc/stat` 指令查看内存使用信息，如图 4.3 所示。内存使用大小通过 `MemTotal-MemFree` 得到，获取对应参数则使用 `fgets` 获取一行信息，再使用 `sscanf` 提取信息参数。

```
orange@ubuntu:~$ cat /proc/meminfo
MemTotal:      4015916 kB
MemFree:       2528200 kB
MemAvailable:  3095092 kB
Buffers:       40452 kB
Cached:        728128 kB
SwapCached:    0 kB
```

图 4.3 /proc/meminfo 内存信息

2. 系统信息显示功能模块设计。包含功能：获取主机名、获取并显示系统启动的时间、显示系统到目前为止持续运行的时间、显示系统的版本号、显示 `cpu` 的型号和主频以及关机、重启功能。

主机名通过 `/proc/sys/kernel/hostname` 获取。系统启动时间和持续运行时间通过 `/proc/uptime` 获取，如图 4.4 所示。其中第一个参数为从系统启动到当前的时间，第二个参数为系统空闲的时间，单位为秒。而系统空闲时间是记录的多核的空闲时间和，由于此系统为 4 核，故需除以 4 方可得到单核的平均空闲时间。

```
orange@ubuntu:~$ cat /proc/uptime
11528.82 45873.70
```

图 4.4 /proc/uptime 运行时间信息

接着在 `/proc/sys/kernel/ostype` 获取系统内核，在 `/proc/sys/kernel/osrelease` 获取内核版本号。在 `/proc/cpuinfo` 中获取 CPU 的型号和主频，如图 4.5 所示。通过循环 `fgets` 获取行字符串，分别将名称字符与“`model name`”和“`cpu MHz`”进行比对，当成功时表示查找到对应行，通过 `sscanf` 提取即可。

```
orange@ubuntu:~$ cat /proc/cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 142
model name     : Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
stepping       : 10
microcode      : 0xc6
cpu MHz        : 1800.000
cache size     : 6144 KB
physical id    : 0
siblings       : 4
core id        : 0
```

图 4.5 /proc/cpuinfo CPU 信息

而关机、重启功能，通过设计回调函数来实现。对于关机功能，使用 `system("shutdown -h now");` 系统指令。重启功能则使用 `system("shutdown -r now");` 实现。

3. CPU 信息、内存信息和 `swap` 信息显示功能模块设计。包含功能：`cpu` 使用率的图形化显示（2 分钟内的历史纪录曲线）、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）。由于三类曲线制作方法相似，故详细介绍 CPU 信息可视化图形设计。首先进行图形编程参数的初始化，如

GdkColor 参数等。接着使用 `gdk_draw_rectangle` 填充背景色，颜色的设置由 `gdk_gc_set_rgb_fg_color(gc, &color);` 实现。为方便图形显示和信息获取，通过增加方格的方式实现。`gdk_draw_line` 函数实现曲线的绘制，以图形左右为 x 端点，y 方向间隔 20 像素画一条横线，以设置的动态变化起点为 x 端点，图形上下为 y 端点画竖线，实现方格的动态移动。设置 120 个数据存储点，随着时间的增加，进行循环填充，即记数超过 120 后取模覆盖旧数据，数据在状态栏生成 CPU 使用率时产生，故无需再次获取，通过存储数据画线。

4. 进程处理功能模块设计。包含功能：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、用新线程运行一个其他程序。设计包括两个部分，进程信息获取和交互功能实现。进程信息获取则通过查看各个进程状态文件 `/proc/(pid)/stat`，然而由于 pid 并非连续，故通过逐个文件打开的方式不成立，需要通过操作文件目录的方式打开。首先打开目录文件 `opendir("/proc");`，通过循环读取目录表信息，获取各 pid 文件路径 `sprintf(pid_path, "/proc/%s/stat", dirinfo->d_name);`。以打开 `pid=1` 的状态文件为例，如图 4.6 所示。

```
orange@ubuntu:~$ cat /proc/1/stat
1 (systemd) S 0 1 1 0 -1 4194560 9763 602340 52 664 39 1113 4687 3104 20 0 1 0 1
8 189861888 1492 18446744073709551615 1 1 0 0 0 0 671173123 4096 1260 0 0 0 17 2
0 0 42 0 0 0 0 0 0 0 0 0 0 0
```

图 4.6 /proc/(pid)/stat 示例

其中所需数据为 `pid=1` 第一个数据。接着是进程名。接着是任务状态 S，如 S 代表 sleeping 状态，R 代表 running 状态等。接着是 `ppid=0` 父进程 id。优先级 `priority=20` 为第 18 个数据。占用控件大小为第 23 个参数，单位为 page，通过 `getpagesize()` 获取，单位为 B。对于数据相对位置固定，可以通过空格数的计数获取对应数据。

其次是交互功能的实现。包含 4 个交互，search、kill、refresh 和 run。各交互 pid 通过 entry 文本框的输入获取，使用函数 `gtk_entry_get_text`。Search 通过显示详细信息窗口来实现，提供查询错误提示。Kill 则通过 `kill` 函数实现，同样提供终结失败窗口提示。Refresh 通过 `gtk_clist_freeze`、`gtk_clist_clear` 进行清空窗口，再调用进程信息获取模块函数显示。Run 则使用 `system` 函数，并提供运行错误窗口提示。

4.4 实验调试

4.4.1 实验步骤

1. 编写好系统监控器文件 `sys_monitor.c`，进行编译 `gcc -o sys_monitor monitor.c `pkg-config --cflags --libs gtk+-2.0`。`

2. 运行执行文件 `sys_monitor`。显示窗口界面。
3. 查看状态栏和系统信息显示界面，如图 4.7 所示。显示正确，当前时间、CPU 使用率和内存使用清空以及系统持续运行时间都能实时动态变化。

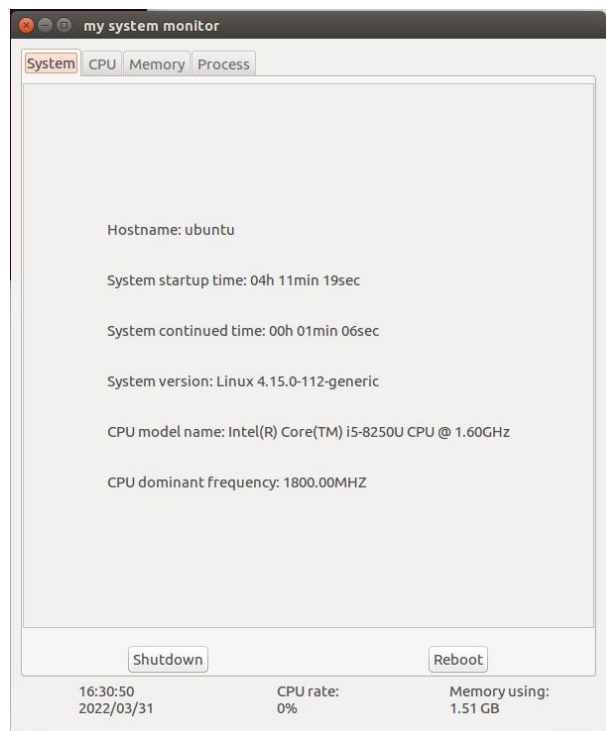


图 4.7 状态栏和系统信息显示界面

4. 查看 CPU 信息显示界面，如图 4.8 所示。在打开浏览器后测试 CPU 的利用率，实现动态变化。

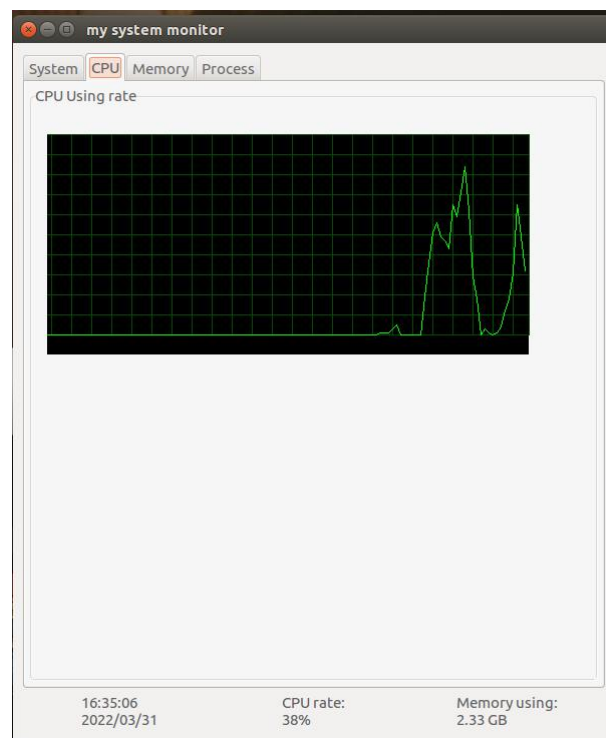


图 4.8 CPU 信息显示界面

5. 查看内存和交换分区信息显示界面，如图 4.9 所示。在打开浏览器后测试内存和交换分区利用率，实现动态变化。

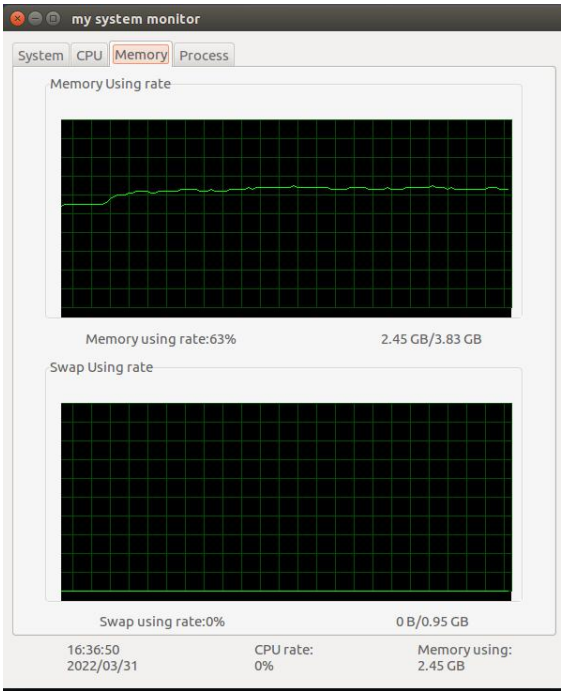


图 4.9 内存和交换分区信息显示界面

6. 查看进程处理显示界面，如图 4.10 所示。显示进程信息，包括所需属性：pid、进程名、进程状态、ppid、优先级以及存储大小。

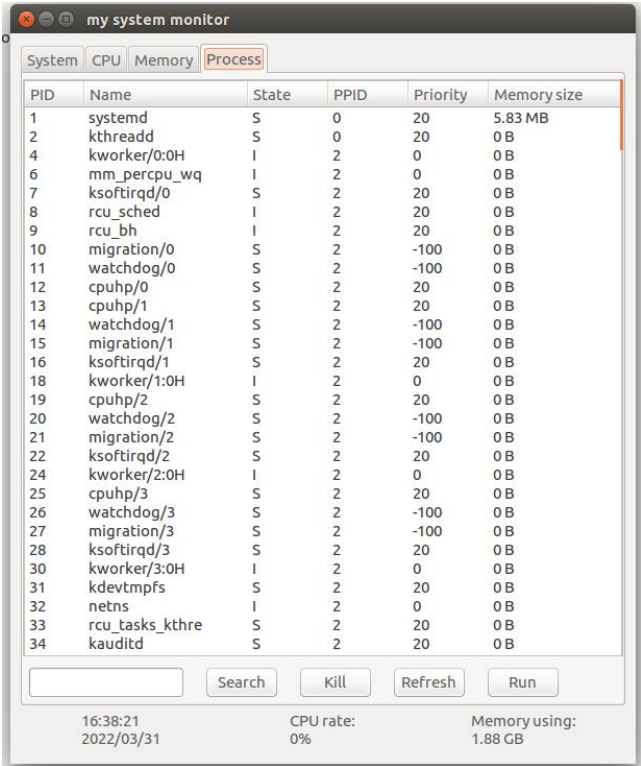


图 4.10 进程处理显示界面

7. 进程信息查询功能测试。在输入栏输入 1 时，显示信息如图 4.11 所示。

成功打开详细信息显示界面。

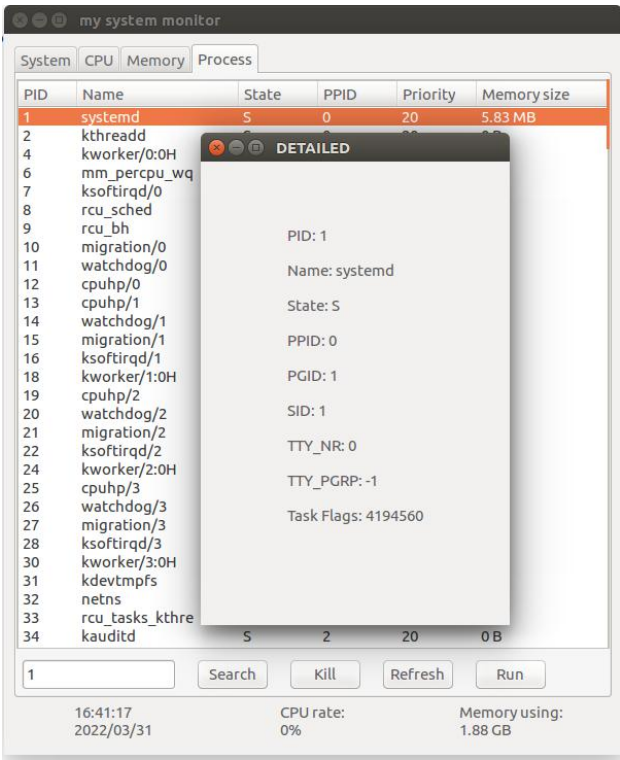


图 4.11 search 功能测试

- 8. 终结进程功能测试。关闭浏览器进程。输入浏览器进程的进程号 4448，点击 Kill 按钮，浏览器关闭，功能测试成功。
- 9. 刷新功能测试。点击 Refresh 按钮，界面刷新成功。
- 10. 运行功能测试。输入 firefox，点击 Run 按钮，打开浏览器成功，如图 4.12 所示。

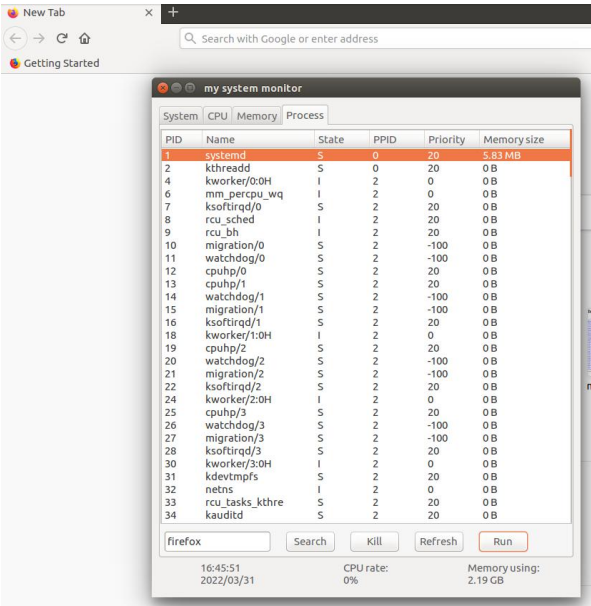


图 4.12 run 功能测试

11. 容错性测试。输入错误类型的pid，如输入 a，点击 Search 按钮。提示错误信息窗口，如图 4.13 所示，测试成功。

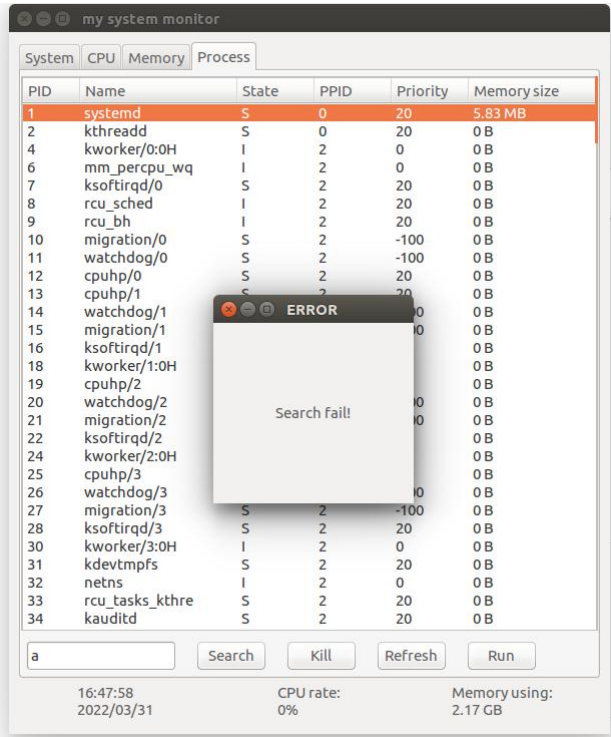


图 4.13 数据类型容错性测试

输入内核处理进程pid，点击 Kill 按钮，不允许终结该进程。显示错误错误，如图 4.14 所示，测试成功。

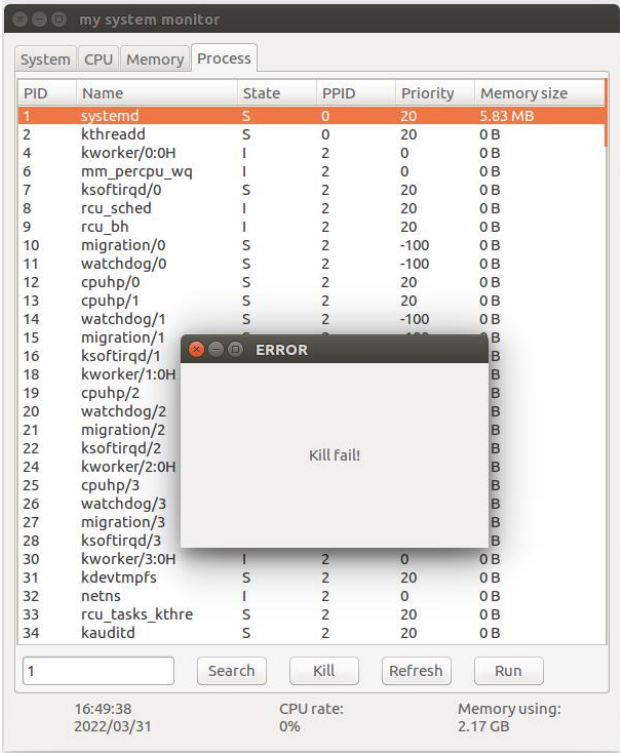


图 4.14 功能性执行失败提示测试

4.4.2 实验调试及心得

调试时注意以下几点：

1. 在进行 GTK+2.0 开发时，需要绘制控件模式图来辅助编程。因为进行程序编写时，常常需要进行控件嵌套，而且变量也很多。通过画图的方式理解各控件的嵌套关系十分重要，不然常常会因为控件无关联或者控件反复嵌套进行报错。

2. 在开辟存储空间时，需要考虑文件内所含数据大小，若开辟空间过小，将导致数据越界。如处理系统信息的获取时，开辟的空间过小，读取数据超过缓存区存放范围，造成数据越界，导致图形输出时产生重复、乱码的问题。

3. 在进行绘图操作时，需要不断调整参数实现图形的动态变化。因此采用循环记数方式处理最为妥当，若不采用循环记数，随着时间的推移会产生数据溢出的问题。因此为显示 2 分钟的可视化界面，采用 120 循环计数的方法，既不会溢出，也不会出现错误，这是工程处理的重要环节。

实验心得：

通过此次实验，让我熟练掌握了 GTK+2.0 编程模式以及 /proc 文件的具体内容。

在 GTK+2.0 编程中，十分体现出控件间的嵌套关系。这类前端编程的思想第一次接触感到十分有趣，体会到了操作系统不仅有内核，外层软件也十分重要，能够极大提高用户体验。

另外，通过实现从系统信息到 CPU 信息，再到内存、交换区信息的获取，让我深刻体会到 Linux 包含一个强大的文件系统，/proc 这一冰山一角也显得十分庞大了。通过查看 /proc 目录下的各个文件，提取所需信息的过程，让我掌握了通过 Linux 文件 cat 查看的指令方式获取系统信息的方法。在不查看显式提供的信息时，也可以通过查看系统内核获取关键信息。这在熟悉一个操作系统的过程中具有重要意义。

最后，系统监控器的实现工作量相比与之前几十行的代码有明显提示。对于掌握工程代码编写能力有很大作用。

附录 实验代码

系统监控器实验代码：

```
//sys_monitor.c
#include <stdio.h>
#include <stdlib.h>
```

```

#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <glib.h>
#include <dirent.h>
#include <time.h>
#include <gtk/gtk.h>

gboolean settime(gpointer data);
gboolean setcpurate(gpointer data);
gboolean setmemuse(gpointer data);
char* B_2_higher(long b);
void myshutdown(void);
void myreboot(void);
gboolean setsysinfo(gpointer label);
gboolean drawcpuusing_callback(GtkWidget* widget);
gboolean drawcpuusing(gpointer widget);
gboolean setmem_rat(gpointer data);
gboolean setmem_fra(gpointer data);
gboolean drawmemusing_callback(GtkWidget* widget);
gboolean drawmemusing(gpointer widget);
gboolean setswap_rat(gpointer data);
gboolean setswap_fra(gpointer data);
gboolean drawswapusing_callback(GtkWidget* widget);
gboolean drawswapusing(gpointer widget);
void setprocinfo(void);
void setpidstat(char(*info)[1000], char* stat_info);
void searchproc(void);
void killproc(void);
void refreshproc(void);
void runproc(void);
char* gettxtddetailed(const char* pid);

GtkWidget* main_window;
GtkWidget* scrolled_window;
GtkWidget* vbox,* hbox;
GtkWidget* table;
GtkWidget* cpuuse_frame;
GtkWidget* cpuuse_draw;
GtkWidget* memuse_frame;
GtkWidget* memuse_draw;

```

```

GtkWidget* swapuse_frame;
GtkWidget* swapuse_draw;
GtkWidget* clist;
GtkWidget* entry01;
GtkWidget* notebook;
GtkWidget* button01;
GtkWidget* button02;
GtkWidget* button03;
GtkWidget* button04;
GtkWidget* time_label,* cpu_label,* mem_label;
GtkWidget* title_label;
GtkWidget* label01;
GtkWidget* label02;
long tt,newtt,idle,newidle; //cpu 各类时间
char titleinfo[100]; //页面标题
int cpu_curve_start = 20; //cpu 曲线
long usage,usage_data[120]; //cpu 数据记录
int mem_curve_start = 20; //mem 曲线
long memusage, memusage_data[120]; //mem 数据记录
int swap_curve_start = 20; //swap 曲线
long swapusage, swapusage_data[120]; //swap 数据记录
int proc_num;

int main(int argc, char *argv[]){
    gtk_init(&argc, &argv);
    /*创建主窗口*/
    main_window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(main_window), "my system monitor");
    gtk_window_set_default_size(GTK_WINDOW(main_window), 600, 700);
    gtk_container_set_border_width(GTK_CONTAINER(main_window), 10);
    gtk_window_set_position(GTK_WINDOW(main_window), GTK_WIN_POS_CENTER);
    gtk_window_set_policy(GTK_WINDOW(main_window),TRUE, TRUE, TRUE);
    g_signal_connect(G_OBJECT(main_window), "delete_event", G_CALLBACK(gtk_main_quit), NULL);

    /*创建笔记本控件*/
    notebook = gtk_notebook_new();
    gtk_notebook_set_tab_pos(GTK_NOTEBOOK(notebook), GTK_POS_TOP);

    /*页面 1： 主机信息*/
    /*显示主机名称， 系统启动时间， 持续运行时间， 系统版本号， CPU 型号， CPU 主频*/
    /*关机功能， 重启功能*/
    vbox = gtk_vbox_new(FALSE, 10);
    hbox = gtk_hbox_new(FALSE, 10);

```



```

        scrolled_window = gtk_scrolled_window_new(NULL, NULL); //信息显示滑动窗口
        gtk_widget_set_size_request(scrolled_window, 500, 550);
        gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW(scrolled_window),
GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);
        label01 = gtk_label_new(" ");
        gtk_scrolled_window_add_with_viewport(GTK_SCROLLED_WINDOW(scrolled_window), label01);
        g_timeout_add(1000, (GtkFunction)setsysinfo, (gpointer)label01);
        button01 = gtk_button_new_with_label("Shutdown"); //关机按钮
        g_signal_connect(G_OBJECT(button01), "clicked", G_CALLBACK(myshutdown), NULL);
        button02 = gtk_button_new_with_label("Reboot"); //重启按钮
        g_signal_connect(G_OBJECT(button02), "clicked", G_CALLBACK(myreboot), NULL);
        gtk_box_pack_start(GTK_BOX(hbox), button01, TRUE, FALSE, 5);
        gtk_box_pack_end(GTK_BOX(hbox), button02, TRUE, FALSE, 5);
        gtk_box_pack_start(GTK_BOX(vbox), scrolled_window, TRUE, FALSE, 5);
        gtk_box_pack_end(GTK_BOX(vbox), hbox, TRUE, FALSE, 0);

        sprintf(titleinfo, "System");
        title_label = gtk_label_new(titleinfo);
        gtk_notebook_append_page(GTK_NOTEBOOK(notebook), vbox, title_label);

        /*页面 2: CPU 使用率图形化*/
        cpuuse_frame = gtk_frame_new("CPU Using rate");
        gtk_container_set_border_width(GTK_CONTAINER(cpuuse_frame), 5);
        gtk_widget_set_size_request(cpuuse_frame, 520, 300);
        cpuuse_draw = gtk_drawing_area_new();
        gtk_widget_set_size_request(cpuuse_draw, 0, 0);
        g_signal_connect(G_OBJECT(cpuuse_draw), "expose_event", G_CALLBACK(drawcpuusing_callback),
NULL);
        gtk_container_add(GTK_CONTAINER(cpuuse_frame), cpuuse_draw);

        sprintf(titleinfo, "CPU");
        title_label = gtk_label_new(titleinfo);
        gtk_notebook_append_page(GTK_NOTEBOOK(notebook), cpuuse_frame, title_label);

        /*页面 3: 内存使用率图形化和 swap 使用率图形化*/
        vbox = gtk_vbox_new(FALSE, 0);

        hbox = gtk_hbox_new(FALSE, 0); //构造 memuse 控件
        gtk_box_pack_start(GTK_BOX(vbox), hbox, TRUE, FALSE, 5);
        memuse_frame = gtk_frame_new("Memory Using rate");
        gtk_container_set_border_width(GTK_CONTAINER(memuse_frame), 5);
        gtk_widget_set_size_request(memuse_frame, 520, 270);
        gtk_box_pack_start(GTK_BOX(hbox), memuse_frame, TRUE, FALSE, 5);

```

```

memuse_draw = gtk_drawing_area_new();
gtk_widget_set_size_request(memuse_draw, 0, 0);
g_signal_connect(G_OBJECT(memuse_draw), "expose_event", G_CALLBACK(drawmemusing_callback),
NULL);
gtk_container_add(GTK_CONTAINER(memuse_frame), memuse_draw);

hbox = gtk_hbox_new(FALSE, 0); //构造文字说明控件
label01 = gtk_label_new("memory using rate");
label02 = gtk_label_new("memory using fraction");
gtk_box_pack_start(GTK_BOX(hbox), label01, TRUE, FALSE, 5);
gtk_box_pack_start(GTK_BOX(hbox), label02, TRUE, FALSE, 5);
g_timeout_add(1000, (GtkFunction)setmem_rat, (gpointer)label01);
g_timeout_add(1000, (GtkFunction)setmem_fra, (gpointer)label02);
gtk_widget_set_size_request(hbox, 550, 20);
gtk_box_pack_start(GTK_BOX(vbox), hbox, TRUE, FALSE, 5);

hbox = gtk_hbox_new(FALSE, 0); //构造 swapuse 控件
gtk_box_pack_start(GTK_BOX(vbox), hbox, TRUE, FALSE, 5);
swapuse_frame = gtk_frame_new("Swap Using rate");
gtk_container_set_border_width(GTK_CONTAINER(swapuse_frame), 5);
gtk_widget_set_size_request(swapuse_frame, 520, 270);
gtk_box_pack_start(GTK_BOX(hbox), swapuse_frame, TRUE, FALSE, 5);
swapuse_draw = gtk_drawing_area_new();
gtk_widget_set_size_request(swapuse_draw, 0, 0);
g_signal_connect(G_OBJECT(swapuse_draw), "expose_event", G_CALLBACK(drawswapusing_callback),
NULL);
gtk_container_add(GTK_CONTAINER(swapuse_frame), swapuse_draw);

hbox = gtk_hbox_new(FALSE, 0); //构造文字说明控件
label01 = gtk_label_new("swap using rate");
label02 = gtk_label_new("swap using fraction");
gtk_box_pack_start(GTK_BOX(hbox), label01, TRUE, FALSE, 5);
gtk_box_pack_start(GTK_BOX(hbox), label02, TRUE, FALSE, 5);
g_timeout_add(1000, (GtkFunction)setswap_rat, (gpointer)label01);
g_timeout_add(1000, (GtkFunction)setswap_fra, (gpointer)label02);
gtk_widget_set_size_request(hbox, 550, 20);
gtk_box_pack_start(GTK_BOX(vbox), hbox, TRUE, FALSE, 5);

sprintf(titleinfo, "Memory");
title_label = gtk_label_new(titleinfo);
gtk_notebook_append_page(GTK_NOTEBOOK(notebook), vbox, title_label);

/*页面 4: 进程信息*/
vbox = gtk_vbox_new(FALSE, 0);

```

```

scrolled_window = gtk_scrolled_window_new(NULL, NULL); //信息显示滑动窗口
gtk_widget_set_size_request(scrolled_window, 500, 550);
gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW(scrolled_window),
GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);

clist = gtk_clist_new(6); //构造列信息框架
setprocinfo();
gtk_scrolled_window_add_with_viewport(GTK_SCROLLED_WINDOW(scrolled_window), clist);
gtk_box_pack_start(GTK_BOX(vbox), scrolled_window, TRUE, TRUE, 5);
hbox = gtk_hbox_new(FALSE, 10);
entry01 = gtk_entry_new(); //输入框与相关按钮
gtk_entry_set_max_length(GTK_ENTRY(entry01), 0);
button01 = gtk_button_new_with_label("Search");
button02 = gtk_button_new_with_label("Kill");
button03 = gtk_button_new_with_label("Refresh");
button04 = gtk_button_new_with_label("Run");
g_signal_connect(G_OBJECT(button01), "clicked", G_CALLBACK(searchproc), NULL);
g_signal_connect(G_OBJECT(button02), "clicked", G_CALLBACK(killproc), NULL);
g_signal_connect(G_OBJECT(button03), "clicked", G_CALLBACK(refreshproc), NULL);
g_signal_connect(G_OBJECT(button04), "clicked", G_CALLBACK(runproc), NULL);
gtk_widget_set_size_request(entry01, 150, 30);
gtk_widget_set_size_request(button01, 70, 30);
gtk_widget_set_size_request(button02, 70, 30);
gtk_widget_set_size_request(button03, 70, 30);
gtk_widget_set_size_request(button04, 70, 30);
gtk_box_pack_start(GTK_BOX(hbox), entry01, FALSE, FALSE, 5);
gtk_box_pack_start(GTK_BOX(hbox), button01, FALSE, FALSE, 5);
gtk_box_pack_start(GTK_BOX(hbox), button02, FALSE, FALSE, 5);
gtk_box_pack_start(GTK_BOX(hbox), button03, FALSE, FALSE, 5);
gtk_box_pack_start(GTK_BOX(hbox), button04, FALSE, FALSE, 5);
gtk_box_pack_start(GTK_BOX(vbox), hbox, FALSE, FALSE, 5);

sprintf(titleinfo, "Process");
title_label = gtk_label_new(titleinfo);
gtk_notebook_append_page(GTK_NOTEBOOK(notebook), vbox, title_label);

/*创建状态栏*/
time_label = gtk_label_new("Time");
cpu_label = gtk_label_new("CPU rate");
mem_label = gtk_label_new("Memory using");

/*创建表格布局容器*/
table = gtk_table_new(3, 3, TRUE);
gtk_table_attach_defaults(GTK_TABLE(table), notebook, 0, 3, 0, 28);
gtk_table_attach_defaults(GTK_TABLE(table), time_label, 0, 1, 28, 30);

```

```

gtk_table_attach_defaults(GTK_TABLE(table), cpu_label, 1, 2, 28, 30);
gtk_table_attach_defaults(GTK_TABLE(table), mem_label, 2, 3, 28, 30);

gtk_container_add(GTK_CONTAINER(main_window), table);
g_timeout_add(1000, settime, NULL);
g_timeout_add(1000, setcpurate, (gpointer)cpu_label);
g_timeout_add(1000, setmemuse, (gpointer)mem_label);
gtk_widget_show_all(main_window);
gtk_main();
return 0;
}

gboolean settime(gpointer data){
    time_t times;
    struct tm* p_time;
    time(&times);
    p_time = localtime(&times);
    gchar* text_data = g_strdup_printf("%04d/%02d/%02d", \
        (1900 + p_time->tm_year), (1 + p_time->tm_mon), (p_time->tm_mday));
    gchar* text_time = g_strdup_printf("%02d:%02d:%02d", \
        (p_time->tm_hour), (p_time->tm_min), (p_time->tm_sec));
    gchar* text_markup = g_strdup_printf("%s\n%s", text_time, text_data);
    gtk_label_set_markup(GTK_LABEL(time_label), text_markup);
    return TRUE;
}

gboolean setcpurate(gpointer data) {
    FILE* fp;
    char cpuinfo[100];
    fp = fopen("/proc/stat", "r");
    fgets(cpuinfo, sizeof(cpuinfo), fp);
    fclose(fp);
    char cpuname[50];
    long Nuser, Nnice, Nsystem, Nidle, Niowait, Nirq, Nsoftirq, ext1, ext2, ext3;
    sscanf(cpuinfo, "%s%ld%ld%ld%ld%ld%ld%ld%ld%ld", cpuname, &Nuser, &Nnice, &Nsystem,
        &Nidle, &Niowait, &Nirq, &Nsoftirq, &ext1, &ext2, &ext3);
    newidle = Nidle;
    newtt = Nuser + Nnice + Nsystem + Nidle + Niowait + Nirq + Nsoftirq + ext1 + ext2 + ext3;
    if (tt > 0) usage = ((newtt - tt) - (newidle - idle)) * 100 / (newtt - tt);
    char cpurate[50];
    sprintf(cpurate, "CPU rate:\n%ld%%", usage);
    gtk_label_set_text(GTK_LABEL(data), cpurate);
    tt = newtt;
}

```

```

        idle = newidle;
        return TRUE;
    }

gboolean setmemuse(gpointer data) {
    FILE* fp;
    char totalmeminfo[50], freememinfo[50];
    char temp1[20], temp2[20];
    long ttm, frem;
    fp = fopen("/proc/meminfo", "r");
    fgets(totalmeminfo, sizeof(totalmeminfo), fp);
    fgets(freememinfo, sizeof(freememinfo), fp);
    fclose(fp);
    sscanf(totalmeminfo, "%s%ld%s", temp1, &ttm, temp2);
    sscanf(freememinfo, "%s%ld%s", temp1, &frem, temp2);
    long usem = ttm - frem; //目前为 KB
    char cum[20];
    strcpy(cum, B_2_higher(usem * 1024));
    char usingmem[50];
    sprintf(usingmem, "Memory using:\n%s", cum);
    gtk_label_set_text(GTK_LABEL(data), usingmem);
    return TRUE;
}

gboolean setmem_rat(gpointer data) {
    FILE* fp;
    char totalmeminfo[50], freememinfo[50];
    char temp1[20], temp2[20];
    long ttm, frem;
    fp = fopen("/proc/meminfo", "r");
    fgets(totalmeminfo, sizeof(totalmeminfo), fp);
    fgets(freememinfo, sizeof(freememinfo), fp);
    fclose(fp);
    sscanf(totalmeminfo, "%s%ld%s", temp1, &ttm, temp2);
    sscanf(freememinfo, "%s%ld%s", temp1, &frem, temp2);
    memusage = (long)((ttm - frem)*100 / ttm);
    char usingmem[50];
    sprintf(usingmem, "Memory using rate:%ld%%", memusage);
    gtk_label_set_text(GTK_LABEL(data), usingmem);
    return TRUE;
}

gboolean setmem_fra(gpointer data) {
    FILE* fp;

```

```

char totalmeminfo[50], freememinfo[50];
char temp1[20], temp2[20];
long ttm, frem;
fp = fopen("/proc/meminfo", "r");
fgets(totalmeminfo, sizeof(totalmeminfo), fp);
fgets(freememinfo, sizeof(freememinfo), fp);
fclose(fp);
sscanf(totalmeminfo, "%s%ld%s", temp1, &ttm, temp2);
sscanf(freememinfo, "%s%ld%s", temp1, &frem, temp2);
long usem = ttm - frem; //目前为 KB
char charum[20], chartm[20];
strcpy(charum, B_2_higher(usem * 1024));
strcpy(chartm, B_2_higher(ttm * 1024));
char usingmem[50];
sprintf(usingmem, "%s/%s", charum, chartm);
gtk_label_set_text(GTK_LABEL(data), usingmem);
return TRUE;
}

gboolean setswap_rat(gpointer data) {
    long tswap, freswap;
    char temp1[20], temp2[20];
    FILE* fp;
    fp = fopen("/proc/meminfo", "r");
    char info01[30] = "SwapTotal";
    char info02[30] = "SwapFree";
    char temp[50];
    int flag = 0;
    while (flag != 2) {
        fgets(temp, sizeof(temp), fp);
        if (fp == NULL) break;
        if (strncmp(temp, info01, 9) == 0) {
            sscanf(temp, "%s%ld%s", temp1, &tswap, temp2);
            flag++;
        }
        else if (strncmp(temp, info02, 7) == 0) {
            sscanf(temp, "%s%ld%s", temp1, &freswap, temp2);
            flag++;
        }
    }
    fclose(fp);
    swapusage = (long)((tswap - freswap) * 100 / tswap);
    char usingswap[50];
    sprintf(usingswap, "Swap using rate:%ld%%", swapusage);

```

```

    gtk_label_set_text(GTK_LABEL(data), usingswap);
    return TRUE;
}

gboolean setswap_fra(gpointer data) {
    long ttswap, freswap;
    char temp1[20], temp2[20];
    FILE* fp;
    fp = fopen("/proc/meminfo", "r");
    char info01[30] = "SwapTotal";
    char info02[30] = "SwapFree";
    char temp[50];
    int flag = 0;
    while (flag != 2) {
        fgets(temp, sizeof(temp), fp);
        if (fp == NULL) break;
        if (strncmp(temp, info01, 9) == 0) {
            sscanf(temp, "%s%ld%s", temp1, &ttswap, temp2);
            flag++;
        }
        else if (strncmp(temp, info02, 7) == 0) {
            sscanf(temp, "%s%ld%s", temp1, &freswap, temp2);
            flag++;
        }
    }
    fclose(fp);
    long useswap = ttswap - freswap;
    char charus[20], charts[20];
    strcpy(charus, B_2_higher(useswap * 1024));
    strcpy(charts, B_2_higher(ttswap * 1024));
    char usingswap[50];
    sprintf(usingswap, "%s/%s", charus, charts);
    gtk_label_set_text(GTK_LABEL(data), usingswap);
    return TRUE;
}

char* B_2_higher(long b){
    static char g[10];
    if (b > 999999999)
        sprintf(g, "%.2f GB", (float)b / 1073741824);
    else {
        if (b > 999999) sprintf(g, "%.2f MB", (float)b / 1048576);
        else {
            if (b > 999) sprintf(g, "%.2f KB", (float)b / 1024);
        }
    }
}

```

```

        else sprintf(g, "%ld B", b);
    }
}
return g;
}

void myshutdown(void) {
    system("shutdown -h now");
    return;
}

void myreboot(void) {
    system("shutdown -r now");
    return;
}

gboolean setsysinfo(gpointer label) {
    char hostname[50];
    char system_startup_time[50];
    char system_continued_time[50];
    char system_version[50];
    char CPU_model[100];
    char CPU_dominant_frequency[50];
    char temp01[50];
    char temp02[50];
    char temp03[50];
    char temp04[50];
    char temp05[100];
    int len;
    FILE* fp;
    fp = fopen("/proc/sys/kernel/hostname", "r"); //获取主机名称
    fgets(temp01, sizeof(temp01), fp);
    fclose(fp);
    len = strlen(temp01);
    temp01[len - 1] = '\0';
    sprintf(hostname, "Hostname: %s", temp01);
    fp = fopen("/proc/uptime", "r"); //获取系统启动时间和运行时间
    fgets(temp02, sizeof(temp02), fp);
    fclose(fp);
    double st, ft, ct;
    sscanf(temp02, "%lf%lf", &st, &ft);
    ct = st - ft/4;
    long h, m, s;
    h = (long)(st / 3600);

```



```

m = (long)(st - h * 3600) / 60;
s = (long)(st - h*3600 - m*60);
sprintf(system_startup_time, "System startup time: %02ldh %02ldmin %02ldsec", h,m,s);
h = (long)(ct / 3600);
m = (long)(ct - h * 3600) / 60;
s = (long)(ct - h * 3600 - m * 60);
sprintf(system_continued_time, "System continued time: %02ldh %02ldmin %02ldsec", h, m, s);
fp = fopen("/proc/sys/kernel/ostype", "r");    //获取系统内核
fgets(temp03, sizeof(temp03), fp);
fclose(fp);
fp = fopen("/proc/sys/kernel/osrelease", "r");    //获取系统版本号
fgets(temp04, sizeof(temp04), fp);
fclose(fp);
len = strlen(temp03);
temp03[len-1] = '\0';
len = strlen(temp04);
temp04[len-1] = '\0';
sprintf(system_version, "System version: %s %s", temp03,temp04);
fp = fopen("/proc/cpuinfo", "r");    //获取系统版本号
char info01[30] = "model name";
char info02[30] = "cpu MHz";
int flag = 0;
while (flag != 2) {
    fgets(temp05, sizeof(temp05), fp);
    if (fp == NULL) break;
    if (strncmp(temp05, info01, 10) == 0) {
        int i;
        len = strlen(temp05);
        for (i = 13; i <= len; i++) {
            temp05[i - 13] = temp05[i];
        }
        len = strlen(temp05);
        temp05[len-1] = '\0';
        sprintf(CPU_model, "CPU model name: %s", temp05);
        flag++;
    }
    else if (strncmp(temp05, info02, 7) == 0) {
        double hz;
        sscanf(temp05, "%s%s%s%lf", temp01, temp02,temp03,&hz);
        sprintf(CPU_dominant_frequency, "CPU dominant frequency: %.2lfMHZ", hz);
        flag++;
    }
}
fclose(fp);

```

```

char totalinfo[300];
sprintf(totalinfo,
        "%s\n\n%s\n\n%s\n\n%s\n\n%s\n\n%s\n\n",
hostname,system_startup_time,system_continued_time,system_version,CPU_model,CPU_dominant_frequency);
gtk_label_set_text(GTK_LABEL(label), totalinfo);
return TRUE;
}

gboolean drawcpuusing_callback(GtkWidget* widget) {
    static int flag = 0;
    drawcpuusing((gpointer)widget);
    if (flag == 0) {
        g_timeout_add(1000, (GtkFunction)drawcpuusing, (gpointer)widget);
        flag = 1;
    }
    return TRUE;
}

gboolean drawcpuusing(gpointer widget) {
    GtkWidget* cpu_curve = (GtkWidget*)widget;
    GdkColor color;
    GdkGC* gc = cpu_curve->style->fg_gc[GTK_WIDGET_STATE(widget)];
    static int flag = 0;
    static int now_pos = 0;
    int draw_pos = 0;

    color.red = 0;
    color.green = 0;
    color.blue = 0;
    gdk_gc_set_rgb_fg_color(gc, &color);
    gdk_draw_rectangle(cpu_curve->window, gc, TRUE, 15, 30, 480, 220); //填充背景

    color.red = 0;
    color.green = 20000;
    color.blue = 0;
    gdk_gc_set_rgb_fg_color(gc, &color);
    for (int i = 30; i <= 230; i += 20)
        gdk_draw_line(cpu_curve->window, gc, 15, i, 495, i); //画横线
    for (int i = 15; i <= 480; i += 20)
        gdk_draw_line(cpu_curve->window, gc, i + cpu_curve_start, 30, i + cpu_curve_start, 230); //画纵线

    cpu_curve_start -= 4; //重复画线
    if (cpu_curve_start == 0)
        cpu_curve_start = 20;
}

```

```

if(flag == 0) { //初始化数据
    for (int i = 0; i < 120; i++) {
        usage_data[i] = 0;
        flag = 1;
    }
}
usage_data[now_pos] = usage; //添加数据
now_pos++;
if (now_pos == 120)
    now_pos = 0;

color.red = 0;
color.green = 65535;
color.blue = 0;
gdk_gc_set_rgb_fg_color(gc, &color);
draw_pos = now_pos;
for (int i = 0; i < 119; i++) { //画线
    gdk_draw_line(cpu_curve->window, gc,
        15 + i * 4, 230 - 2 * usage_data[draw_pos % 120],
        15 + (i + 1) * 4, 230 - 2 * usage_data[(draw_pos + 1) % 120]);
    draw_pos++;
    if (draw_pos == 120)
        draw_pos = 0;
}

color.red = 25000;
color.green = 25000;
color.blue = 25000;
gdk_gc_set_rgb_fg_color(gc, &color);
return TRUE;
}

gboolean drawmemusing_callback(GtkWidget* widget) {
    static int flag = 0;
    drawmemusing((gpointer)widget);
    if (flag == 0) {
        g_timeout_add(1000, (GtkFunction)drawmemusing, (gpointer)widget);
        flag = 1;
    }
    return TRUE;
}

gboolean drawmemusing(gpointer widget) {
    GtkWidget* mem_curve = (GtkWidget*)widget;

```

```

GdkColor color;
GdkGC* gc = mem_curve->style->fg_gc[GTK_WIDGET_STATE(widget)];
static int flag = 0;
static int now_pos = 0;
int draw_pos = 0;

color.red = 0;
color.green = 0;
color.blue = 0;
gdk_gc_set_rgb_fg_color(gc, &color);
gdk_draw_rectangle(mem_curve->window, gc, TRUE, 15, 30, 480, 220); //填充背景

color.red = 0;
color.green = 20000;
color.blue = 0;
gdk_gc_set_rgb_fg_color(gc, &color);
for (int i = 30; i <= 230; i += 20)
    gdk_draw_line(mem_curve->window, gc, 15, i, 495, i); //画横线
for (int i = 15; i <= 480; i += 20)
    gdk_draw_line(mem_curve->window, gc, i + mem_curve_start, 30, i + mem_curve_start, 230); //画
纵线

mem_curve_start -= 4; //重复画线
if (mem_curve_start == 0)
    mem_curve_start = 20;

if (flag == 0) { //初始化数据
    for (int i = 0; i < 120; i++) {
        memusage_data[i] = 0;
        flag = 1;
    }
}
memusage_data[now_pos] = memusage; //添加数据
now_pos++;
if (now_pos == 120)
    now_pos = 0;

color.red = 0;
color.green = 65535;
color.blue = 0;
gdk_gc_set_rgb_fg_color(gc, &color);
draw_pos = now_pos;
for (int i = 0; i < 119; i++) { //画线
    gdk_draw_line(mem_curve->window, gc,

```

```

        15 + i * 4, 230 - 2 * memusage_data[draw_pos % 120],
        15 + (i + 1) * 4, 230 - 2 * memusage_data[(draw_pos + 1) % 120]);
draw_pos++;
if (draw_pos == 120)
    draw_pos = 0;
}

color.red = 25000;
color.green = 25000;
color.blue = 25000;
gdk_gc_set_rgb_fg_color(gc, &color);
return TRUE;
}

gboolean drawswapusing_callback(GtkWidget* widget) {
    static int flag = 0;
    drawswapusing((gpointer)widget);
    if (flag == 0) {
        g_timeout_add(1000, (GtkFunction)drawswapusing, (gpointer)widget);
        flag = 1;
    }
    return TRUE;
}

gboolean drawswapusing(gpointer widget) {
    GtkWidget* swap_curve = (GtkWidget*)widget;
    GdkColor color;
    GdkGC* gc = swap_curve->style->fg_gc[GTK_WIDGET_STATE(widget)];
    static int flag = 0;
    static int now_pos = 0;
    int draw_pos = 0;

    color.red = 0;
    color.green = 0;
    color.blue = 0;
    gdk_gc_set_rgb_fg_color(gc, &color);
    gdk_draw_rectangle(swap_curve->window, gc, TRUE, 15, 30, 480, 220); //填充背景

    color.red = 0;
    color.green = 20000;
    color.blue = 0;
    gdk_gc_set_rgb_fg_color(gc, &color);
    for (int i = 30; i <= 230; i += 20)
        gdk_draw_line(swap_curve->window, gc, 15, i, 495, i); //画横线

```

```

        for (int i = 15; i <= 480; i += 20)
            gdk_draw_line(swap_curve->window, gc, i + swap_curve_start, 30, i + swap_curve_start, 230); //画
纵线

        swap_curve_start -= 4; //重复画线
        if (swap_curve_start == 0)
            swap_curve_start = 20;

        if (flag == 0) { //初始化数据
            for (int i = 0; i < 120; i++) {
                swapusage_data[i] = 0;
                flag = 1;
            }
        }
        swapusage_data[now_pos] = swapusage; //添加数据
        now_pos++;
        if (now_pos == 120)
            now_pos = 0;

        color.red = 0;
        color.green = 65535;
        color.blue = 0;
        gdk_gc_set_rgb_fg_color(gc, &color);
        draw_pos = now_pos;
        for (int i = 0; i < 119; i++) { //画线
            gdk_draw_line(swap_curve->window, gc,
                15 + i * 4, 230 - 2 * swapusage_data[draw_pos % 120],
                15 + (i + 1) * 4, 230 - 2 * swapusage_data[(draw_pos + 1) % 120]);
            draw_pos++;
            if (draw_pos == 120)
                draw_pos = 0;
        }

        color.red = 25000;
        color.green = 25000;
        color.blue = 25000;
        gdk_gc_set_rgb_fg_color(gc, &color);
        return TRUE;
    }

void setprocinfo(void) {
    gtk_clist_set_column_title(GTK_CLIST(clist), 0, "PID");
    gtk_clist_set_column_title(GTK_CLIST(clist), 1, "Name");
    gtk_clist_set_column_title(GTK_CLIST(clist), 2, "State");
}

```

```

gtk_clist_set_column_title(GTK_CLIST(clist), 3, "PPID");
gtk_clist_set_column_title(GTK_CLIST(clist), 4, "Priority");
gtk_clist_set_column_title(GTK_CLIST(clist), 5, "Memory size");
gtk_clist_set_column_width(GTK_CLIST(clist), 0, 50);
gtk_clist_set_column_width(GTK_CLIST(clist), 1, 150);
gtk_clist_set_column_width(GTK_CLIST(clist), 2, 70);
gtk_clist_set_column_width(GTK_CLIST(clist), 3, 70);
gtk_clist_set_column_width(GTK_CLIST(clist), 4, 70);
gtk_clist_set_column_width(GTK_CLIST(clist), 5, 100);
gtk_clist_column_titles_show(GTK_CLIST(clist));
DIR* dir;
struct dirent* dirinfo;
int fd;
char pid_path[50];
char pid_info[1000];
char* one_file;
char ttinfo[6][1000];
gchar* listinfo[6];
proc_num = 0;
dir = opendir("/proc");
while ((dirinfo = readdir(dir)) != NULL) {
    if (((dirinfo->d_name)[0] >= '0') && ((dirinfo->d_name)[0] <= '9')) { //只打开数字 pid 的文件
        sprintf(pid_path, "/proc/%s/stat", dirinfo->d_name);
        fd = open(pid_path, O_RDONLY);
        read(fd, pid_info, 1000);
        close(fd);
        one_file = pid_info;
        setpidstat(ttinfo, one_file);
        for (int i = 0; i < 6; i++) {
            listinfo[i] = g_locale_to_utf8(ttinfo[i], -1, NULL, NULL, NULL);
        }
        gtk_clist_append(GTK_CLIST(clist), listinfo);
        proc_num++;
    }
}
closedir(dir);
return;
}

void setpidstat(char(*info)[1000], char* stat_info) {
    int i;

    for (i = 0; i < 1000; i++) { //获取 pid
        if (stat_info[i] == ' ') break;
    }
}

```

```

}
stat_info[i] = '\0';
strcpy(info[0], stat_info);
i += 2;
stat_info += i;

for (i = 0; i < 1000; i++) { //获取 name
    if (stat_info[i] == ')') break;
}
stat_info[i] = '\0';
strcpy(info[1], stat_info);
i += 2;
stat_info += i;

for (i = 0; i < 1000; i++) { //获取 state
    if (stat_info[i] == ' ') break;
}
stat_info[i] = '\0';
strcpy(info[2], stat_info);
i += 1;
stat_info += i;

for (i = 0; i < 1000; i++) { //获取 ppid
    if (stat_info[i] == ' ') break;
}
stat_info[i] = '\0';
strcpy(info[3], stat_info);
i += 1;
stat_info += i;

int j = 0;
for (i = 0; i < 1000; i++) { //获取 priority
    if (stat_info[i] == ' ') j++;
    if (j == 13) break;
}
stat_info[i] = '\0';
i += 1;
stat_info += i;
for (i = 0; i < 1024; i++) {
    if (stat_info[i] == ' ') break;
}
stat_info[i] = '\0';
strcpy(info[4], stat_info);
i += 1;

```



```

stat_info += i;

j = 0;
for (i = 0; i < 1000; i++) { //获取 size
    if (stat_info[i] == ' ') j++;
    if (j == 5) break;
}
stat_info[i] = '\0';
i += 1;
stat_info += i;
for (i = 0; i < 1024; i++) {
    if (stat_info[i] == ' ') break;
}
stat_info[i] = '\0';
long size = atoi(stat_info); //单位为 pagesize
int page = getpagesize(); //单位为 B
size = size * page;
strcpy(info[5], B_2_higher(size));
return;
}

void searchproc(void) {
    const gchar* entry_txt;
    gchar* list_txt;
    entry_txt = gtk_entry_get_text(GTK_ENTRY(entry01));
    gint i = 0;
    while (gtk_clist_get_text(GTK_CLIST(clist), i, 0, &list_txt)) {
        if (strcmp(entry_txt, list_txt) == 0) break;
        i++;
    }
    if (i >= proc_num) {
        GtkWidget* win = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        GtkWidget* label = gtk_label_new("Search fail!");
        gtk_widget_set_size_request(win, 200, 180);
        gtk_container_add(GTK_CONTAINER(win), label);
        gtk_window_set_title(GTK_WINDOW(win), "ERROR");
        gtk_window_set_position(GTK_WINDOW(win), GTK_WIN_POS_CENTER);
        gtk_widget_show_all(win);
        return;
    }
    else {
        GtkWidget* win = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        char detailed[1024];
        strcpy(detailed, gettxt_detailed(entry_txt));
    }
}

```

```

        GtkWidget* label = gtk_label_new(detailed);
        gtk_widget_set_size_request(win, 300, 450);
        gtk_container_add(GTK_CONTAINER(win), label);
        gtk_window_set_title(GTK_WINDOW(win), "DETAILED");
        gtk_window_set_position(GTK_WINDOW(win), GTK_WIN_POS_CENTER);
        gtk_widget_show_all(win);
    }
    gtk_clist_select_row(GTK_CLIST(clist), i, 0);
    return;
}

void killproc(void) {
    const gchar* entry_txt;
    gchar* list_txt;
    entry_txt = gtk_entry_get_text(GTK_ENTRY(entry01));
    gint i = 0;
    while (gtk_clist_get_text(GTK_CLIST(clist), i, 0, &list_txt)) {
        if (strcmp(entry_txt, list_txt) == 0) break;
        i++;
    }
    gtk_clist_select_row(GTK_CLIST(clist), i, 0);
    if (entry_txt == NULL) return;
    int ret = kill(atoi(entry_txt), SIGKILL);
    if (ret) {
        GtkWidget* killfailwin = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        GtkWidget* killfaillabel = gtk_label_new("Kill fail!");
        gtk_widget_set_size_request(killfailwin, 300, 180);
        gtk_container_add(GTK_CONTAINER(killfailwin), killfaillabel);
        gtk_window_set_title(GTK_WINDOW(killfailwin), "ERROR");
        gtk_window_set_position(GTK_WINDOW(killfailwin), GTK_WIN_POS_CENTER);
        gtk_widget_show_all(killfailwin);
    }
    return;
}

void refreshproc(void) {
    gtk_clist_freeze(GTK_CLIST(clist));
    gtk_clist_clear(GTK_CLIST(clist));
    setprocinfo();
    gtk_clist_thaw(GTK_CLIST(clist));
    gtk_clist_select_row(GTK_CLIST(clist), 0, 0);
    gtk_entry_set_text(GTK_ENTRY(entry01), "");
    return;
}

```

```

void runproc(void) {
    const gchar* entry_txt;
    entry_txt = gtk_entry_get_text(GTK_ENTRY(entry01));
    char txt[30];
    sprintf(txt, "%s&", entry_txt);
    int ret = system(txt);
    if (ret == -1 || ret == 127) {
        GtkWidget* win = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        GtkWidget* label = gtk_label_new("Run fail!");
        gtk_widget_set_size_request(win, 200, 180);
        gtk_container_add(GTK_CONTAINER(win), label);
        gtk_window_set_title(GTK_WINDOW(win), "ERROR");
        gtk_window_set_position(GTK_WINDOW(win), GTK_WIN_POS_CENTER);
        gtk_widget_show_all(win);
        return;
    }
    return;
}

char* gettxt_detailed(const char* pid) {
    char temp[1024] = "";
    char* p = temp;
    char pid_path[50];
    char pid_info[1000];
    char* stat_info;
    sprintf(pid_path, "/proc/%s/stat", pid);
    int fd = open(pid_path, O_RDONLY);
    read(fd, pid_info, 1000);
    close(fd);
    stat_info = pid_info;
    int i;

    for (i = 0; i < 1000; i++) { //获取 pid
        if (stat_info[i] == ' ') break;
    }
    stat_info[i] = '\0';
    strcat(temp, "PID: ");
    strcat(temp, stat_info);
    strcat(temp, "\n\n");
    i += 2;
    stat_info += i;

    for (i = 0; i < 1000; i++) { //获取 name

```

```

        if (stat_info[i] == ')') break;
    }
    stat_info[i] = '\0';
    strcat(temp, "Name: ");
    strcat(temp, stat_info);
    strcat(temp, "\n\n");
    i += 2;
    stat_info += i;

    for (i = 0; i < 1000; i++) { //获取 state
        if (stat_info[i] == ')') break;
    }
    stat_info[i] = '\0';
    strcat(temp, "State: ");
    strcat(temp, stat_info);
    strcat(temp, "\n\n");
    i += 1;
    stat_info += i;

    for (i = 0; i < 1000; i++) { //获取 ppid
        if (stat_info[i] == ')') break;
    }
    stat_info[i] = '\0';
    strcat(temp, "PPID: ");
    strcat(temp, stat_info);
    strcat(temp, "\n\n");
    i += 1;
    stat_info += i;

    for (i = 0; i < 1000; i++) { //获取 pgid
        if (stat_info[i] == ')') break;
    }
    stat_info[i] = '\0';
    strcat(temp, "PGID: ");
    strcat(temp, stat_info);
    strcat(temp, "\n\n");
    i += 1;
    stat_info += i;

    for (i = 0; i < 1000; i++) { //获取 sid
        if (stat_info[i] == ')') break;
    }
    stat_info[i] = '\0';
    strcat(temp, "SID: ");

```

```

    strcat(temp, stat_info);
    strcat(temp, "\n\n");
    i += 1;
    stat_info += i;

    for (i = 0; i < 1000; i++) { //获取 tty_nr
        if (stat_info[i] == ' ') break;
    }
    stat_info[i] = '\0';
    strcat(temp, "TTY_NR: ");
    strcat(temp, stat_info);
    strcat(temp, "\n\n");
    i += 1;
    stat_info += i;

    for (i = 0; i < 1000; i++) { //获取 tty_pgrp
        if (stat_info[i] == ' ') break;
    }
    stat_info[i] = '\0';
    strcat(temp, "TTY_PGRP: ");
    strcat(temp, stat_info);
    strcat(temp, "\n\n");
    i += 1;
    stat_info += i;

    for (i = 0; i < 1000; i++) { //获取 task_flag
        if (stat_info[i] == ' ') break;
    }
    stat_info[i] = '\0';
    strcat(temp, "Task Flags: ");
    strcat(temp, stat_info);
    strcat(temp, "\n\n");
    i += 1;
    stat_info += i;

    return p;
}

```