



华中科技大学

操作系统原理课程设计报告

姓 名：李响

学 院：计算机科学与技术学院

专 业：计算机科学与技术

班 级：CS1802

学 号：U201814531

指导教师：胡贯荣

分数	
教师签名	

2021 年 04 月 02 日

目 录

1 实验一 Linux 下 C 编程的相关知识.....	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.....	3
1.4.1 实验步骤.....	3
1.4.2 实验调试及心得.....	3
附录 实验代码.....	4
2 实验二 系统调用相关知识.....	7
2.1 实验目的.....	7
2.2 实验内容.....	7
2.3 实验设计.....	7
2.3.1 开发环境.....	7
2.3.2 实验设计.....	7
2.4 实验调试.....	9
2.4.1 实验步骤.....	9
2.4.2 实验调试及心得.....	10
附录 实验代码.....	11
3 实验三 设备驱动相关知识.....	13
3.1 实验目的.....	13
3.2 实验内容.....	13
3.3 实验设计.....	13
3.3.1 开发环境.....	13
3.3.2 实验设计.....	13
3.4 实验调试.....	14
3.4.1 实验步骤.....	14
3.4.2 实验调试及心得.....	16
附录 实验代码.....	17
4 实验四 使用 QT 实现系统监控器.....	20

4.1 实验目的.....	20
4.2 实验内容.....	20
4.3 实验设计.....	21
4.3.1 开发环境.....	21
4.3.2 实验设计.....	21
4.4 实验调试.....	28
4.4.1 实验步骤.....	28
4.4.2 实验调试及心得.....	28
5 实验五 小型文件系统.....	32
5.1 实验目的.....	32
5.2 实验内容.....	32
5.3 实验设计.....	32
5.3.1 开发环境.....	32
5.3.2 实验设计.....	33
5.4 实验调试.....	37
5.4.1 实验步骤.....	37
5.4.2 实验调试及心得.....	38
附录 实验代码.....	43

1 实验一 Linux 下 C 编程的相关知识

1.1 实验目的

掌握 Linux 操作系统的基本使用方法，包括键盘命令、系统调用等方法；熟悉 Linux 下的编程环境，使用 Linux 进行 C 语言编程。

1.2 实验内容

1. 编一个 C 程序，其内容为实现 cp 命令/文件拷贝的功能（注意要求使用系统调用 open/read/write 等，不可使用 C 语言自带的标准文件操作函数）；
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），如三个进程誊抄演示。

1.3 实验设计

1.3.1 开发环境

1. 硬件环境：
 - （1）中央处理器 CPU：AMD Ryzen 7 4800H 2.9 GHz
 - （2）物理内存：16.00 GB
2. 开发编译运行环境（VMware Workstation Pro 16）
 - （1）虚拟机系统版本：Ubuntu18.04 操作系统
 - （2）编译器版本：GCC 7.5.0
 - （3）调试器版本：GNU gdb 8.1.0
 - （4）虚拟机内存：4.00 GB
 - （5）虚拟机磁盘空间：80.00 GB

1.3.2 实验设计

任务 1：实现 cp 命令/文件拷贝的功能

任务 1 的要求是使用系统功能调用实现文件拷贝功能，首先，使用 open 系

统调用函数打开文件，然后使用 read 和 write 系统功能调用函数对文件进行读取与写入，最后使用 close 系统功能调用关闭两个文件即可实现拷贝功能。

为使用系统功能调用，需要添加四个头文件 fcntl.h、unistd.h、sys/types.h 以及 sys/stat.h，四种系统功能调用函数原型声明如下：

```
int open(const char *pathname,int flags);  
int open(const char *pathname,int flags,mode_t mode);  
ssize_t read(int fd, void *buffer,size_t count);  
ssize_t write(int fd, const void *buffer,size_t count);  
int close(int fd);
```

为使得文件拷贝功能更加灵活方便，使用命令行参数的方式输入拷贝的源文件与目标文件名。由于源文件只需要读操作，因此，使用 O_RDONLY 作为 flag，目标文件如果不存在需要进行创建，因此需要使用 O_WRONLY|O_CREAT 作为 flag，文件权限设置为 S_IRUSR|S_IWUSR。

```
/* 拷贝文件 */  
while(bytes_read=read(from_fd,buffer,BUFFER_SIZE))  
{  
    /* 一个致命的错误发生了 */  
    if((bytes_read==-1)&&(errno!=EINTR)) break;  
    else if(bytes_read>0)  
    {  
        ptr=buffer;  
        while(bytes_write=write(to_fd,ptr,bytes_read))  
        {  
            /* 一个致命错误发生了 */  
            if((bytes_write==-1)&&(errno!=EINTR))break;  
            /* 写完了所有读的字节 */  
            else if(bytes_write==bytes_read) break;  
            /* 只写了一部分,继续写 */  
            else if(bytes_write>0)  
            {  
                ptr+=bytes_write;  
                bytes_read-=bytes_write;  
            }  
        }  
        /* 写的时候发生的致命错误 */  
        if(bytes_write==-1)break;  
    }  
}
```

图 1-1 文件拷贝读写过程代码截图

然后，使用双层循环的方式对文件进行拷贝，需要注意的是，在拷贝时需要对文件操作进行如图 1-1 所示的中断等异常检测，以免出现读写错误。

任务 2：分窗口显示三个并发进程运行结果

由于对 QT 的使用比较熟悉，因此本任务使用 QT 进行图形化编程，本任务的要求是分窗口显示三个不同进程的运行结果，由于之前使用过 QT 的图形化编

程，因此可以复用之前的图形化窗口，本次显示的两个进程分别是文件拷贝以及两种不同的资源管理器窗口。使用 `fork` 函数创建子进程，使用 `execv` 函数替换程序段，打开新程序即可实现多窗口。

使用 QT 带 `ui` 编程进行窗口编程，使用 QT 自带的窗口控件对文本和数据进行显示与读取即可。

1.4 实验调试

1.4.1 实验步骤

任务 1：实现 `cp` 命令/文件拷贝的功能

在 Linux 平台下，使用 VS Code 进行代码编辑，然后在命令行中使用 `gcc` 命令进行编译。本次实验实现的 `cp` 命令使用命令行进行源文件名与目标文件名参数的输入，因此需在终端输入 “`./copy 源文件 目标文件`” 来进行文件拷贝。

使用 `gcc` 命令编译 `copy.c` 文件，生成可执行文件 `copy`，在命令行进行测试。

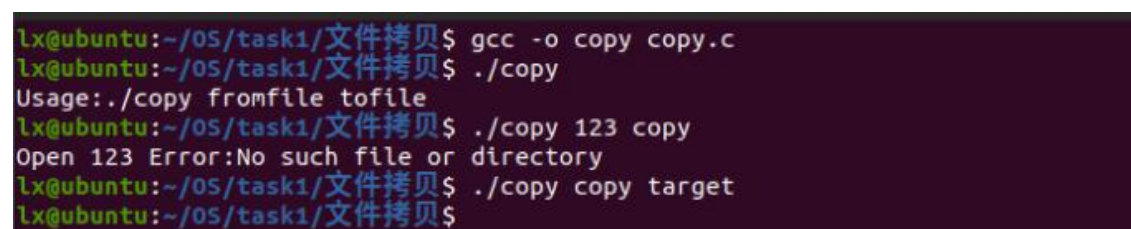
任务 2：分窗口显示三个并发进程运行结果

使用 QT 进行程序的编辑、编译与运行，复用部分实验四与网络代码进行窗口显示，使用 QT 创建项目，编写代码并进行测试，使用 `fork` 函数创建子进程，`execv` 函数替换代码。

1.4.2 实验调试及心得

任务 1：实现 `cp` 命令/文件拷贝的功能

按照实验步骤进行测试，如图 1-2 所示，当输入的参数个数不正确时，系统会报错 “Usage:./copy fromfile tofile”；当输入的源文件 123 不存在时，系统报错 “Open 123 Error:No such file or directory”；如果拷贝成功，系统不会报任何异常。



```
lx@ubuntu:~/05/task1/文件拷贝$ gcc -o copy copy.c
lx@ubuntu:~/05/task1/文件拷贝$ ./copy
Usage:./copy fromfile tofile
lx@ubuntu:~/05/task1/文件拷贝$ ./copy 123 copy
Open 123 Error:No such file or directory
lx@ubuntu:~/05/task1/文件拷贝$ ./copy copy target
lx@ubuntu:~/05/task1/文件拷贝$
```

图 1-2 文件拷贝功能运行结果截图

任务 2：分窗口显示三个并发进程运行结果

使用 QT 进行程序的编译与运行，当点击运行按钮后，出现三个窗口。

如图 1-3 所示，左上角的窗口是一个资源管理器，可以分页面显示不同的信息，是实验四需要完成的内容，本任务复用了实验四的代码。

左下角的窗口是文件拷贝过程监视窗口，从左侧的两个文本输入框输入源文件与目标文件后，点击 **start** 按钮即可开始拷贝，进度条会显示拷贝进度，右侧的提示窗口会显示提示信息，点击 **finish** 即可关闭窗口。

右侧为另一个资源管理器窗口，该窗口是网络上的资源，本次的实验四参考了部分实现思路。

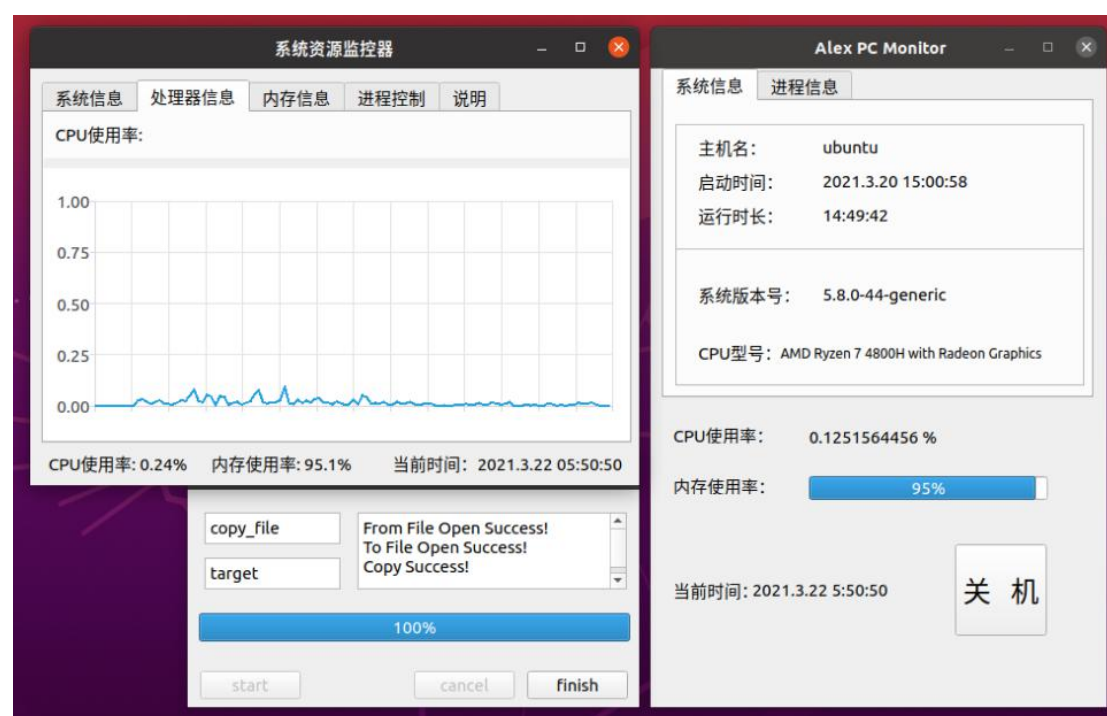


图 1-3 窗口并发进程运行结果显示截图

附录 实验代码

任务 1：

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```

#include <errno.h>
#include <string.h>
#define BUFFER_SIZE 1024

int main(int argc, char **argv)
{
    int from_fd, to_fd;
    int bytes_read, bytes_write;
    char buffer[BUFFER_SIZE];
    char *ptr;
    /* 输入参数错误 */
    if(argc!=3){
        fprintf(stderr, "Usage:%s fromfile tofile\n\a", argv[0]);
        return(-1);
    }
    /* 打开源文件 */
    if((from_fd=open(argv[1], O_RDONLY))== -1){
        fprintf(stderr, "Open %s Error:%s\n", argv[1], strerror(errno));

        return(-1);
    }
    /* 创建目的文件 */
    if((to_fd=open(argv[2], O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR))== -1){
        fprintf(stderr, "Open %s Error:%s\n", argv[2], strerror(errno));

        return(-1);
    }
    /* 拷贝文件 */
    while(bytes_read=read(from_fd, buffer, BUFFER_SIZE)){
        /* 一个致命的错误发生了 */
        if((bytes_read== -1)&&(errno!=EINTR)) break;
        else if(bytes_read>0){
            ptr=buffer;
            while(bytes_write=write(to_fd, ptr, bytes_read)){
                /* 一个致命错误发生了 */
                if((bytes_write== -1)&&(errno!=EINTR)) break;
                /* 写完了所有读的字节 */
                else if(bytes_write==bytes_read) break;
                /* 只写了一部分,继续写 */
                else if(bytes_write>0){
                    ptr+=bytes_write;
                    bytes_read-=bytes_write;
                }
            }
        }
    }
}

```



```

        /* 写的时候发生的致命错误 */
        if(bytes_write==-1)break;
    }
}
close(from_fd);
close(to_fd);
/* 读写错误 */
if(bytes_read==-1||bytes_write==-1){
    fprintf(stderr,"R/W ERROR:%s \n",strerror(errno));
    return(-1);
}
return(1);
}

```

2 实验二 系统调用相关知识

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。

2.2 实验内容

1. 向新内核增添一个新的系统调用，将内核编译、生成，并用新内核启动；
2. 新增系统调用实现：文件拷贝，并编写测试应用程序，调用新系统功能调用进行测试。

2.3 实验设计

2.3.1 开发环境

1. 硬件环境：
 - (1) 中央处理器 CPU: AMD Ryzen 7 4800H 2.9 GHz
 - (2) 物理内存: 16.00 GB
2. 开发编译运行环境 (VMware Workstation Pro 16)
 - (1) 虚拟机系统版本: Ubuntu18.04 操作系统
 - (2) 编译器版本: GCC 7.5.0
 - (3) 调试器版本: GNU gdb 8.1.0
 - (4) 虚拟机内存: 4.00 GB
 - (5) 虚拟机磁盘空间: 80.00 GB

2.3.2 实验设计

系统调用是用户应用程序和操作系统内核之间的功能接口，通过系统调用进程可由用户模式转入内核模式，在内核模式下完成相应服务后再返回用户模式。

由于要求使用编译内核的方式把新系统调用永久性加入内核中，但因为系统

镜像里并不包含内核源码，因此需要额外下载新的内核源码进行内核编译。

然后，将编写好的系统功能调用代码加入内核源码中，并将内核编译即可将新系统调用永久地加入新内核中。

最后，编写测试程序对系统功能调用进行测试。

本次实验中新增系统调用的主要功能是文件拷贝，因此在代码结构上可以参考实验一的文件拷贝的实现，但是，由于系统功能调用的在内核空间中运行，系统功能调用的实现与普通用户状态下的程序实现存在一些不同。

首先，由于程序运行在内核空间，因此不能使用用户态的 `open`、`close` 等系统功能调用函数进行文件操作，也不能使用 `printf` 的其他用户空间函数。在内核空间中进行文件打开、关闭以及读写等操作需要使用 `sys_open`、`sys_close`、`sys_read` 以及 `sys_write` 代替系统调用函数（有些内核使用 `kysys` 函数，不同版本的内核使用的函数存在差异），信息打印函数使用 `printk` 进行输出。

```
asmlinkage long sys_copyfile(const char __user* source, const char __user* target)
{
    int from_fd, to_fd;
    int bytes_read, bytes_write;
    char buffer[1024];
    char *ptr;

    /* 解除内核对用户地址的访问检查 */
    mm_segment_t old_fs = get_fs();
    set_fs(KERNEL_DS);

    /* 打开源文件与目标文件 */
    from_fd = sys_open(source, O_RDONLY, S_IRUSR);
    to_fd = sys_open(target, O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);

    if (from_fd <= 0){
        printk("source path error!\n");
        set_fs(old_fs);
        return -1;
    }
    if (to_fd <= 0){
        printk("target path error!\n");
        set_fs(old_fs);
        return -1;
    }
}
```

图 2-1 新增系统功能调用部分代码截图

其次，由于内核空间对数据的安全性要求较高，因此会对传入的用户系统参数进行检查，如图 2-1 所示，为实现程序功能需要使用 `set_fs(KERNEL_DS)` 规避这种检查，在程序返回时使用 `set_fs` 恢复原状态避免内核出现问题。

最后，`sys_open` 函数在打开异常时的返回值不总是 -1（可能是其他负数），因此不能使用 -1 作为打开失败的判断标志。

2.4 实验调试

2.4.1 实验步骤

1. 下载内核源码：从 Linux 内核下载官网 <http://www.kernel.org/> 下载稳定版本的内核进行编译，经过多次编译尝试，最终本次实验选择内核版本为 4.14.225 的内核进行系统功能实验。

2. 打开终端，进入源码压缩包所在的目录，在终端输入命令 “tar -xavf linux-4.14.225.tar.xz -C /usr/src” 将内核源码压缩包解压，并移动到 /usr/src 目录下准备开始实验。

3. 使用 apt 命令安装或更新一些编译内核需要使用的工具，具体需要安装的软件工具包如下：

```
sudo apt-get install libncurses5-dev openssl libssl-dev
```

```
sudo apt-get install build-essential openssl
```

```
sudo apt-get install pkg-config
```

```
sudo apt-get install libc6-dev
```

```
sudo apt-get install bison
```

```
sudo apt-get install flex
```

```
sudo apt-get install libelf-dev
```

```
sudo apt-get install zlibc minizip
```

```
sudo apt-get install libidn11-dev libidn11
```

4. 输入命令 “cd /usr/src/linux-4.14.225” 进入源码目录，然后，在终端输入命令 “sudo gedit arch/x86/entry/syscalls/syscall_64.tbl” 查看并修改系统调用表，如图 2-2 所示，按照约定的格式将本次新增的系统功能调用 sys_copyfile 加入到系统调用表中。



333	common	helloworld	sys_helloworld
334	common	copyfile	sys_copyfile

图 2-2 新增系统功能调用号填写图

5. 在终端输入命令 “sudo gedit include/linux/syscalls.h” 查看并修改系统功能调用头文件，将新系统功能调用的函数声明加入到文件末尾，保存并关闭文件。

6. 在终端输入命令“`sudo gedit kernel/sys.c`”查看并修改系统功能调用，将新系统功能调用的函数实现代码加入到文件末尾，保存并关闭文件。

7. 系统调用代码添加完毕，开始编译内核，分别在终端输入命令“`sudo make mrproper`”以及“`sudo make clean`”清除原有编译的痕迹，输入命令“`sudo make menuconfig`”生成编译配置文件.config。

8. 输入命令“`sudo make -j12`”开始进行内核的编译，j 之后的数字为内核的数目，本次给虚拟机分配的内核数量为 12，为加快编译使用 12 个线程编译。

9. 当内核编译完成后，输入命令“`sudo make modules_install`”以及“`sudo make install`”安装内核模块并将内核加入到系统中，编译完成。

10. 在终端输入命令“`sudo gedit /etc/default/grub`”修改 Grub 文件，将配置信息“`GRUB_TIMEOUT_STYLE=hidden`”注释，并修改“`GRUB_TIMEOUT`”为-1 即可在每次启动都进入内核选择界面。

11. 重新启动虚拟机，使用新编译完成的内核进入系统，使用编写好的测试程序对程序进行测试。

2.4.2 实验调试及心得

编写新系统功能调用的测试程序对新增的系统功能调用进行测试，使用 `syscall` 函数调用新系统功能调用，由于新增系统功能调用的系统功能号为 334，且需要两个参数，分别是源文件名以及目标文件名，因此使用“`syscall(334, argv[1], argv[2])`”调用新系统功能调用，测试效果。

```
lx@ubuntu:~/task2$ ./test
Usage:./test fromfile tofile
lx@ubuntu:~/task2$ ./test 1 2
ERROR
lx@ubuntu:~/task2$ ./test test target
```

图 2-3 新增系统功能调用测试结果图

如图 2-3 所示，当输入参数个数不正确时，报错“`Usage:./test fromfile tofile`”；当输入的源文件不存在时，系统功能调用返回-1，程序报错 `ERROR`；当输入的文件名都正确时，文件成功复制。

实验心得：

实验二的复杂程度和困难相较于实验一有着指数级的增长，由于是初次接触

内核编程方面的知识，在实验初上手时我是有一些手足无措的，在查阅了一些资料后，实验逐渐步入正轨。

本次实验的主要难点在于如何进行内核相关的编程以及如何将新增的系统功能调用加入内核中，由于不同版本的系统新增系统功能调用的方式大有不同，在 Ubuntu18.04 系统版本下，新增系统功能调用不再是修改 `entry.S` 等文件，而是修改 `syscall_64.tbl` 系统功能调用表。

此外，内核版本的选择也是个大问题，很多版本的内核存在图形化界面不兼容的问题，因此更换了三个内核后最终确定了 4.14.225 这个版本的内核。

附录 实验代码

```
asmlinkage long sys_copyfile(const char __user* source,
                             const char __user* target)
{
    int from_fd, to_fd;
    int bytes_read, bytes_write;
    char buffer[1024];
    char *ptr;

    /* 解除内核对用户地址的访问检查 */
    mm_segment_t old_fs = get_fs();
    set_fs(KERNEL_DS);

    /* 打开源文件与目标文件 */
    from_fd = sys_open(source, O_RDONLY, S_IRUSR);
    if (from_fd <= 0){
        printk("source path error!\n");
        set_fs(old_fs);
        return -1;
    }
    to_fd = sys_open(target, O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    if (to_fd <= 0){
        printk("target path error!\n");
        set_fs(old_fs);
        return -1;
    }

    /* 拷贝文件 */
    while((bytes_read=sys_read(from_fd, buffer, 1024)))
    {
```

```

        /* 一个致命的错误发生了 */
        if(bytes_read==-1) break;
        else if(bytes_read>0)
        {
            ptr=buffer;
            while((bytes_write=sys_write(to_fd, ptr, bytes_read)))
            {
                /* 一个致命错误发生了 */
                if(bytes_write==-1)break;
                /* 写完了所有读的字节 */
                else if(bytes_write==bytes_read) break;
                /* 只写了一部分,继续写 */
                else if(bytes_write>0)
                {
                    ptr+=bytes_write;
                    bytes_read-=bytes_write;
                }
            }
            /* 写的时候发生的致命错误 */
            if(bytes_write==-1)break;
        }
    }

    sys_close(from_fd);
    sys_close(to_fd);

    /* 读写错误 */
    if(bytes_read==-1||bytes_write==-1){
        printk("read/write error!\n");
        return -1;
    }
    set_fs(old_fs);
    return 1;
}

```

3 实验三 设备驱动相关知识

3.1 实验目的

掌握增加设备驱动程序的方法，通过模块方法，增加一个新设备驱动程序。

3.2 实验内容

通过模块方法，增加一个新的设备驱动程序，其功能可以简单，如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。

3.3 实验设计

3.3.1 开发环境

1. 硬件环境：
 - (1) 中央处理器 CPU: AMD Ryzen 7 4800H 2.9 GHz
 - (2) 物理内存: 16.00 GB
2. 开发编译运行环境 (VMware Workstation Pro 16)
 - (1) 虚拟机系统版本: Ubuntu18.04 操作系统
 - (2) 编译器版本: GCC 7.5.0
 - (3) 调试器版本: GNU gdb 8.1.0
 - (4) 虚拟机内存: 4.00 GB
 - (5) 虚拟机磁盘空间: 80.00 GB

3.3.2 实验设计

设备驱动程序是操作系统内核和机器硬件间的接口。设备驱动程序为应用程序屏蔽了硬件的细节，在应用程序看来，硬件设备只是一个设备文件，应用程序可以像操作普通文件一样对硬件设备进行操作。

设备驱动程序是内核的一部分，它至少需要完成 4 个基本功能，分别是对设

备初始化和释放，把数据从内核传送到硬件和从硬件读取数据，读取应用程序传送给设备文件数据和回送应用程序请求数据以及检测和处理设备出现的错误。

Linux 支持三中不同类型的设备：字符设备（character devices）、块设备（block devices）和网络设备（network interfaces），本次实验将实现字符设备的驱动，演示内核缓冲区的读写操作。

添加新的设备驱动的过程其实是编写函数填充 `file_operations` 的各个域的过程。在设备驱动程序中有一个非常重要的结构 `file_operations`，该结构的每个域都对应着一个系统调用。用户进程利用系统调用在对设备文件进行操作时，系统调用通过设备文件的主设备号找到相应的设备驱动程序，然后读取这个数据结构相应的函数指针，接着把控制权交给该函数。

为实现设备驱动的基本功能，至少需要实现 `read`、`write`、`open` 以及 `release` 这四个子函数的编写，其中 `open` 函数使用 `kmalloc` 内核空间分配函数分配缓冲区，`read` 函数负责从缓冲区中读取数据，`write` 函数负责向缓冲区写入数据，`close` 函数则是使用 `kfree` 函数释放内核缓冲区。

同时对于可卸载的内核模块，至少还需要模块初始化函数以及模块卸载函数这两个基本的模块用于模块的挂载以及卸载。模块初始化函数需要使用系统功能调用函数 `register_chrdev` 将模块挂载到内核上，模块卸载函数则是使用系统功能调用函数 `unregister_chrdev` 将模块卸载。

最后，将完成的源码文件编译，并挂载到内核上即可开始编写测试程序，测试当前的功能。

3.4 实验调试

3.4.1 实验步骤

1. 编写字符驱动程序：本驱动程序利用内核中的一片缓冲区实现数据的写入与读取。首先，按照要求编写 `read`、`write`、`open` 以及 `release` 这四个子函数，并将之与 `file_operations` 的指定域进行绑定；然后，编写驱动设备的初始化与卸载模块，使用 `module_init` 函数以及 `module_exit` 函数将函数与功能绑定；最后，将编写完成的字符驱动源码复制指定目录下，准备开始编译。

2. 模块编译：源码文件复制到当内核源码目录中的 `drivers/misc` 目录下；然后，进入 `drivers/misc` 目录，修改当前目录下 `Makefile` 文件，添加一行命令“`obj-m +=my_drive.o`”；最后，在当前目录下输入命令“`sudo make -C /usr/src/linux SUBDIRS=$PWD modules`”开始编译，如果编译成功将得到 `.ko` 文件。

3. 设备挂载与创建：在当前目录下输入命令“`sudo insmod ./my_drive.ko`”；然后，输入命令“`cat /proc/devices`”查看系统给此设备分配的主设备号，如图 3-1 所示，当前我的设备 `my_drive` 的主设备号为 240；最后，输入命令“`mknod /dev/my_drive C 240 0`”创建虚拟设备，其中第一项参数为当前创建的虚拟设备的路径以及名称，第二个参数 `C` 表示当前创建的设备为字符设备，第三个参数 240 为设备的主设备号，第四个参数 0 为设备的从设备号（可自行分配）。

```
204 ttyMAX
216 rfcomm
226 drm
240 my_drive
241 aux
242 hidraw
```

图 3-1 新增设备驱动主设备号

4. 设备测试：编写测试程序，如图 3-2 所示，使用 `open` 系统功能调用打开设备，系统会自动将控制权交给 `file_operations` 的 `open` 函数指针，执行之前实现的 `open` 子函数，如果当前设备打开失败，则会返回负值；然后，使用 `write` 函数以及 `read` 函数对设备缓冲区进行读写，最后关闭设备。

```
fd = open(DEV_NAME, O_RDWR);
if (fd < 0) {
    printf("open dev fail!\n");
    return -1;
}
do {
    printf("Input some worlds to kernel(enter 'quit' to exit)\n");
    memset(buffer, 0, BUF_SIZE);
    if (fgets(buffer, BUF_SIZE, stdin) == NULL) {
        printf("fgets error!\n");
        break;
    }
    buffer[strlen(buffer) - 1] = '\0';
    if (write(fd, buffer, strlen(buffer)) < 0) {
        printf("write error\n");
        break;
    }
    if (read(fd, buffer, BUF_SIZE) < 0) {
        printf("read error\n");
        break;
    }
    else printf("The read string is from kernel : %s\n", buffer);
} while(strncmp(buffer, "quit", 4));

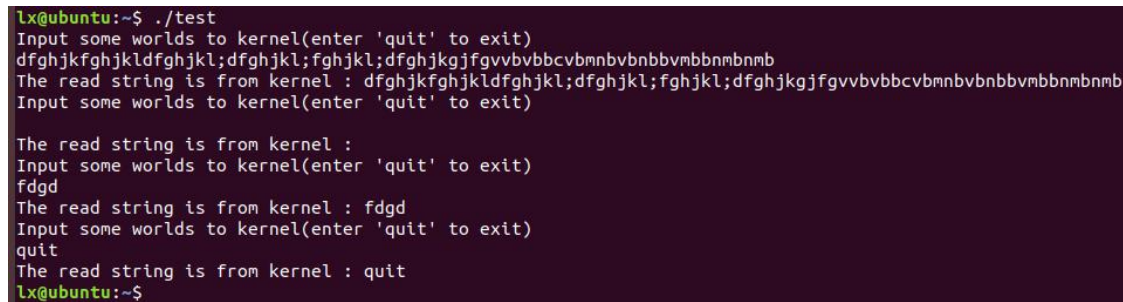
close(fd);
```

图 3-2 新增设备驱动测试程序代码截图

5. 测试完成，卸载设备模块：进入 `drivers/misc` 目录，在当前目录下输入命令“`sudo rmmod my_drive`”将挂载到内核中的模块卸载，然后在当前目录下输入命令“`rm /dev/my_drive`”删除新增的字符设备文件。

3.4.2 实验调试及心得

使用 GCC 命令“`gcc -o test test.c`”编译事先编写完成的测试程序 `test.c` 源文件，然后在命令行输入命令“`./test`”开始进行测试，如图 3-3 所示，测试程序将输入的字符使用 `write` 函数写入字符设备的缓冲区中，然后使用 `read` 函数从字符设备的缓冲区中读取字符，最后显示在终端屏幕上，输入 `quit` 关闭设备。



```
lx@ubuntu:~$ ./test
Input some worlds to kernel(enter 'quit' to exit)
dfghjklfghjkl;dfghjkl;fghjkl;dfghjkgjfgvvbvbcbvmbnbnbbvmbnbnmb
The read string is from kernel : dfghjklfghjkl;dfghjkl;fghjkl;dfghjkgjfgvvbvbcbvmbnbnbbvmbnbnmb
Input some worlds to kernel(enter 'quit' to exit)

The read string is from kernel :
Input some worlds to kernel(enter 'quit' to exit)
fdgd
The read string is from kernel : fdgd
Input some worlds to kernel(enter 'quit' to exit)
quit
The read string is from kernel : quit
lx@ubuntu:~$
```

图 3-3 新增设备驱动测试结果图

实验心得：

通过实验三的实验过程，我初步掌握了如何实现一个简单的字符设备驱动，了解到了设备驱动的基本运行原理，设备驱动的打开关闭及基本的读写等其他操作的实现方式，同时，掌握了如何将编写完成的驱动设备进行挂载和卸载等。

本次实验的主要难点在于掌握设备驱动 `file_operations` 结构体在设备驱动编写中的作用以及如何将编写的模块进行编译与挂载等。

在本次实验中，我并没有遇到特别严重和困难的问题，主要的问题都是在基本的操作方面，这一个实验的实现相较于实验二来说确实算得上顺利。

附录 实验代码

```
#include <linux/kernel.h>
#include <linux/module.h>

#include <linux/fs.h>
#include <linux/init.h>
#include <linux/types.h>
#include <linux/errno.h>
#include <linux/uaccess.h>
#include <linux/kdev_t.h>
#include <linux/cdev.h>
#include <linux/slab.h>

#define MAX_SIZE 1024

int my_open(struct inode *inode, struct file *file);
int my_release(struct inode *inode, struct file *file);
ssize_t my_read(struct file *fileP, char *buf,
                size_t count, loff_t *ppos);
ssize_t my_write(struct file *fileP, const char *buf,
                 size_t count, loff_t *ppos);

char *data = NULL;
int device_num; //设备号
char* devName = "my_drive"; //设备名

/* 注册 */
int _my_init_module(void)
{
    device_num = register_chrdev(0, devName, &pStruct);
    if (device_num < 0)
    {
        printk("failed to register my drive.\n");
        return -1;
    }
    printk("my drive has been registered!\n");
    printk("id: %d\n", device_num);
    return 0;
}
```

```

/* 注销 */
void _my_cleanup_module(void)
{
    unregister_chrdev(device_num, devName);
    printk("unregister successful.\n");
}

/* 打开 */
int my_open(struct inode *inode, struct file *file)
{
    try_module_get(THIS_MODULE);
    printk("module_refcount(module):%d\n", module_refcount(THIS_MODULE));
    ;

    data = (char*)kmalloc(sizeof(char) * MAX_SIZE, GFP_KERNEL);
    if (!data) return -ENOMEM;
    memset(data, 0, MAX_SIZE);
    printk("my_drive open successful!\n");
    return 0;
}

/* 关闭 */
int my_release(struct inode *inode, struct file *file)
{
    module_put(THIS_MODULE);
    printk("module_count:%d\n", module_refcount(THIS_MODULE));
    printk("Device released!\n");
    if (data)
    {
        kfree(data);
        data = NULL;
    }
    return 0;
}

/* 读数据 */
ssize_t my_read(struct file *fileP, char *buf, size_t count, loff_t *p
pos)
{
    if (!buf) return -EINVAL;
    if (count > MAX_SIZE) count = MAX_SIZE;
    if (count < 0 ) return -EINVAL;

```

```

    if (copy_to_user(buf, data, count) == EFAULT)
        return -EFAULT;

    printk("user read data from device!\n");
    return count;
}

/* 写数据 */
ssize_t my_write(struct file *fileP, const char *buf, size_t count, lo
ff_t *ppos)
{
    if (!buf) return -EINVAL;
    if (count > MAX_SIZE) count = MAX_SIZE;
    if (count < 0 ) return -EINVAL;

    memset(data, 0, MAX_SIZE);
    if (copy_from_user(data, buf, count) == EFAULT)
        return -EFAULT;

    printk("user write data to device\n");
    return count;
}

struct file_operations pStruct =
{
    owner:THIS_MODULE,
    open:my_open,
    release:my_release,
    read:my_read,
    write:my_write,
};
module_init(_my_init_module);
module_exit(_my_cleanup_module);

MODULE_AUTHOR("lql");
MODULE_LICENSE("GPL");

```

4 实验四 使用 QT 实现系统监控器

4.1 实验目的

了解 Linux 系统中 `/proc` 文件的特点和使用方法，掌握使用 `/proc` 下的文件查询并监控系统中进程运行情况，同时，学会利用 GTK/QT 等 Linux 图形资源库完成对系统资源的图形化显示与控制。

4.2 实验内容

学习并掌握通过读取 `proc` 文件系统获取系统各种信息的方法，使用 Linux 下图形库 GTK/QT 进行图形化界面的开发，让系统信息以比较容易理解的方式显示出来，其需要完成的功能具体包括：

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. 通过 `pid` 或进程名查询进程，显示该进程详细信息，提供杀掉该进程功能
7. 显示系统所有进程信息，包括 `pid`，`ppid`，占用内存大小，优先级等
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

1. 硬件环境：

- (1) 中央处理器 CPU: AMD Ryzen 7 4800H 2.9 GHz
- (2) 物理内存: 16.00 GB

2. 开发编译运行环境 (VMware Workstation Pro 16)

- (1) 虚拟机系统版本: Ubuntu18.04 操作系统
- (2) 编译器版本: GCC 7.5.0
- (3) 调试器版本: GNU gdb 8.1.0
- (4) 虚拟机内存: 4.00 GB
- (5) 虚拟机磁盘空间: 80.00 GB

4.3.2 实验设计

本次实验要求使用图形界面实现资源管理器，需要使用到 Linux 的图形库，由于对 QT 的使用比较熟悉，因此本次实验使用 QT 进行资源管理器的编写。

Linux 系统中的系统信息保存在 `/proc` 文件系统中，用户和应用程序可以通过 `/proc` 文件系统就可以得到系统的信息，也可以通过 `/proc` 文件系统改变内核的某些参数。由于系统的信息是动态改变的，所以用户或应用程序读取 `proc` 文件时，`proc` 文件系统是动态从系统内核读出所需信息并提交的。

要显示系统信息，只需打开相对应的 `proc` 文件系统下的文件，读取并按照规定显示到 QT 的控件上即可实现系统信息的输出，最后将控件进行组合就可以得到最终的控制器。

由于本次实验需要实现的功能较多，为使得布局美观，首先需要对整体的框架以及界面进行设计，实现设计各个控件的位置与相互作用关系，然后分模块对不同的功能进行实现，具体实现过程如下：

1. 使用 QT 创建基于 Widget 的带 ui 的窗口项目，进入 UI 设计界面进行 UI 设计，将基本的窗口大小调整为 550*370。

2. 插入 Widget 部件，将此部件变型为 QTabWidget 部件（此部件可以实现标签页的切换），插入 5 个标签页，分别命名为系统信息、处理器信息、内存信息、进程控制以及说明，基本窗口结构如图 4-1 所示，各个页面功能如下：



图 4-1 资源管理器窗口基本结构设计图

（1）系统信息：显示功能 1-5 的内容，包括主机名、系统启动时间、系统运行时间、版本号、CPU 的型号与主频；

（2）处理器信息：显示功能 8 的内容，CPU 使用率 2 分钟内纪录曲线；

（3）内存信息：显示功能 9 的内容，内存和交换分区使用率 2 分钟曲线；

（4）进程控制：显示功能 7 的各个进程的信息，并通过排序的方法，实现功能 6 进程的查找与杀死功能；

（5）说明：开发者信息以及关机功能

（6）状态栏：显示功能 9-11 的内容，分别在窗口的最下端始终显示当前的时间、CPU 使用率以及内存使用情况。

3. 按照规划将各个控件放置到设计好的位置上，**系统信息页面**的设计比较简单，需要使用两个 QFrame 部件作为分块显示部件，再向 Frame 上添加 Label 作为显示信息的部件，具体布局如图 4-1 所示；其他几个页面的设计也采用相同的方式，其中，由于需要在**处理器信息**以及**内存信息**页面画曲线，因此需要创建 QVBoxLayot 类的部件作为显示的背板；**进程控制**页面使用 QTableWidgetItem 部件显示各个进程的信息（没选择 QListWidget 的原因主要是 QListWidget 没有办法按照特定的值将表格排序，因此选择 QTableWidgetItem）。

4. 分模块分功能实现填充各个部件，将系统信息显示到指定部件上，并通过事件以及槽函数实现对进程控制操作及信息刷新，各功能具体设计过程如下：

(1) 显示主机名：如图 4-2 所示，打开文件 `/proc/sys/kernel/hostname`，该文件的内容即是本系统的主机名，使用 `readAll` 函数读取主机信息并转化成 `QString` 后使用 `setText` 方法将主机名信息显示在 `Label` 上；

```
/* 设置主机名信息 */
tempFile.setFileName("/proc/sys/kernel/hostname");
tempFile.open(QIODevice::ReadOnly);
allArray = tempFile.readAll();
tempStr = QString(allArray);
tempFile.close();
ui->host_msglabel->setText(tempStr.mid(0,tempStr.length()-1));
```

图 4-2 显示主机信息代码截图

(2) 显示系统启动的时间：可以读取 `/proc/uptime` 文件获取 `uptime`，但是为了简化程序，使用 Linux 的 `sysinfo` 函数获取 `uptime`，将当前时间减去运行时间即是系统开始运行的时间。使用 `localtime` 函数将开始运行的时间从秒转化为可读的时间，并用 `setText` 方法将系统启动时间输出，代码如图 4-3 所示；

```
/* 读取时间信息 */
struct sysinfo info;
time_t cur_time=0, boot_time=0;
struct tm *ptm=NULLptr;
if(sysinfo(&info)) return;
time(&cur_time);
boot_time=cur_time-info.uptime;
char time[30];

/* 设置启动时间信息 */
ptm=localtime(&boot_time);
sprintf(time,"%d.%d.%d %02d:%02d:%02d",ptm->tm_year+1900,
        ptm->tm_mon+1,ptm->tm_mday,ptm->tm_hour,ptm->tm_min,ptm->tm_sec);
ui->start_msglabel->setText(QString(time));
```

图 4-3 显示系统启动时间代码截图

(3) 显示系统运行时间：如图 4-4 所示，使用 Linux 的 `sysinfo` 函数获取 `uptime`，将运行的秒数转化为“天/时/分/秒”的形式输出到标签上；

```
/*显示运行时间信息*/
struct sysinfo info;
if(sysinfo(&info)) return;
sprintf(time,"%ld天%02ld时%02ld分%02ld秒",info.uptime/3600/24,
        info.uptime/3600%24,info.uptime/60%60,info.uptime%60);
ui->run_msglabel->setText(QString(time));
```

图 4-4 显示系统运行时间代码截图

(4) 显示系统版本号：如图 4-5 所示，打开文件/proc/sys/kernel/osrelease，该文件的内容即是本系统版本号，使用相同的方式将信息显示在 Label 上；

```
/*显示系统版本号信息*/
tempFile.setFileName("/proc/sys/kernel/osrelease");
tempFile.open(QIODevice::ReadOnly);
allArray = tempFile.readAll();
tempStr = QString(allArray);
tempFile.close();
ui->ed_msglabel->setText(tempStr.mid(0,tempStr.length()-1));
```

图 4-5 显示系统版本号代码截图

(5) 显示 cpu 的型号和主频大小：打开文件/proc/cpuinfo 查看文件的组织形式，如图 4-6 所示可以发现 cpu 的型号名称处于 model name 以及 stepping 之间，因此，如图 4-7 所示，使用 indexOf 函数查找到 model name 以及 stepping 即可找到 cpu 的型号信息；同理，找到 cpu 的主频信息，使用 setText 方法将主机名信息显示在 Label 上即可；

```
5 model name      : AMD Ryzen 7 4800H with Radeon Graphics
6 stepping        : 1
7 microcode       : 0x8600104
8 cpu MHz         : 2894.562
```

图 4-6 cpuinfo 文件部分内容截图

```
/* 打开cpuinfo文件 */
tempFile.setFileName("/proc/cpuinfo");
tempFile.open(QIODevice::ReadOnly);
allArray = tempFile.readAll();
tempStr = QString(allArray);
tempFile.close();

/* 设置cpu型号信息 */
int from=tempStr.indexOf("model name");
int to=tempStr.indexOf("stepping");
ui->cpu_msglabel->setText(tempStr.mid(from+13,to-from-14));

/*读取cpu主频信息*/
from=tempStr.indexOf("cpu MHz");
to=tempStr.indexOf("cache size");
ui->cpu_fremsglabel->setText(tempStr.mid(from+11,to-from-12)+" MHz");
```

图 4-7 显示 cpu 信息代码截图

(6) 显示进程信息：显示进程信息存在两种选择，第一种是使用 ListWidget 进行显示，第二种是使用 TableWidget 进行显示，由于本次实验需要使用各个信息进行排序，如果使用 ListWidget 比较麻烦，因此使用 TableWidget 进行显示。

为了显示所有进程的信息，首先需要使用 QDir 读取/proc 目录下的所有文件夹，使用 toInt 函数将文件夹的名称转换成数字，如果转换失败则表示进程读取

完成；如果当前文件夹的名称为纯数字，表示当前文件夹内的文件保存着进程信息，打开当前文件夹下的 stat 文件读取当前进程的信息，通过查阅资料可知，如图 4-8 所示，进程名使用小括号包裹，ppid 的值为第 3 个数字，优先级为第 17 个数字，内存大小为第 22 个数字，将信息读取并显示到 TabelWidget 上；使用循环持续读取进程信息显示到表格中即可。

```
1 1 (systemd) S 0 1 1 0 -1 4194560 177470 17839783 136 8106 508 1000 36039
16575 20 0 1 0 8 173203456 2558 18446744073709551615 1 1 0 0 0 0 671173123
4096 1260 0 0 0 17 3 0 0 13 0 0 0 0 0 0 0 0 0 0
```

```
/* 打开进程状态文件 */
QFile tempFile("/proc/" + pro_id + "/stat");
tempFile.open(QIODevice::ReadOnly);
tempStr = tempFile.readAll();

int begin_index = tempStr.indexOf("(");
int end_index = tempStr.indexOf(")");
pro_Name = tempStr.mid(begin_index+1, end_index-begin_index-1);
parent_pro_id = tempStr.section(" ", 3, 3);
pro_Priority = tempStr.section(" ", 17, 17);
pro_Mem = tempStr.section(" ", 22, 22);
number_of_pro++;
pro_status=tempStr.section(' ',2,2);
switch(pro_status.at(0).toLatin1()){
case 'S':number_of_sleep++;break;
case 'Z':break;
case 'R':number_of_run++;break;
}

/* 增加表项 */
ui->tableWidget->insertRow(0);
QTableWidgetItem* pItem = new QTableWidgetItem();
pItem->setData(Qt::EditRole, temp_num);
ui->tableWidget->setItem(0,0,pItem);
ui->tableWidget->setItem(0,1,new QTableWidgetItem(pro_Name));
ui->tableWidget->setItem(0,2,new QTableWidgetItem(parent_pro_id));
ui->tableWidget->setItem(0,3,new QTableWidgetItem(pro_status));
ui->tableWidget->setItem(0,4,new QTableWidgetItem(pro_Priority));
ui->tableWidget->setItem(0,5,new QTableWidgetItem(pro_Mem));
tempFile.close();
```

图 4-8 进程信息文件组织形式以及显示信息代码截图

(7) 查询并杀掉进程：为简化查询操作，本次使用排序加查找的方式进行查询，当用户点击表头时，程序会根据用户点击的表头类进行排序，如点击 pid 表头，资源管理器会调用 TabelWidget 的 sortItems 方法按照 pid 的数值进行排序，在此基础上用户只要拖动页面查找需要查询的进程即可，点击查询的表项，然后点击“结束进程”按钮，系统就会像需要杀死的进程发送 SIGKILL 信号，将需要结束的进程杀死。

(8) cpu 使用率的图形化显示：本次实验使用 QChart 控件显示 cpu 使用率历史曲线，首先，初始化曲线画板，然后，每秒刷新曲线即可实现 cpu 使用率历史曲线的图形化显示，具体过程如下：

```
/*创建画板*/
cpu_chart = new QChart;
/*创建曲线*/
cpu_series = new QLineSeries;
cpu_chart->addSeries(cpu_series);
for(int i=0;i<cpu_series_Size;++i)
    cpu_series->append(i,0);

/*创建cpu坐标轴*/
QValueAxis *axisX = new QValueAxis;
axisX->setRange(0,cpu_series_Size);
axisX->setTickCount(cpu_series_Size/8);
axisX->setLabelsVisible(false);
QValueAxis *axisY = new QValueAxis;
axisY->setRange(0, 1);
axisY->setTickCount(5);
axisY->setLabelFormat("%.2f");

/*将坐标轴添加到画板*/
cpu_chart->setAxisX(axisX,cpu_series);
cpu_chart->setAxisY(axisY,cpu_series);
cpu_chart->legend()->hide();

QChartView *chartView = new QChartView(cpu_chart);
ui->verticalLayout->addWidget(chartView);
```

图 4-9 初始化 cpu 使用率历史曲线代码截图

首先，对画板进行初始化，如图 4-9 所示，创建画板并创建一条 120 个点的曲线（每秒刷新一下，两分钟曲线需要 120 个点），将曲线的每个点的利用率初始化为 0；然后，创建坐标轴并将坐标轴添加到画板；最后，将画板添加到 Layout 布局中即可。

然后，需要每秒对曲线进行刷新，首先，使用 `QVector<QPointF>` 将曲线的点转化成由点组成的向量，删除下标为 0 的点；然后，使用 `QFile` 打开 `/proc/stat` 文件，读取并计算 cpu 的运行信息。

经过查阅资料可知，`stat` 文件中除了第 5 项为 cpu 运行信息，其他各项都是 cpu 空闲信息，将当前的运行时间减去上一秒的运行时间，然后除以总运行时间即可得出当前阶段的 cpu 使用率；将得出的数值添加到曲线的末端，并将曲线重新绘制到画板上即可实现曲线的更新与移动。

```

/* 将曲线向前移动一个单位 */
 QVector<QPointF> oldPoints = cpu_series->pointsVector();
 QVector<QPointF> points;
 for(int i=1;i<oldPoints.count();++i)
     points.append(QPointF(i-1 ,oldPoints.at(i).y()));

/*打开/proc/stat文件*/
 QFile tempFile("/proc/stat");
 tempFile.open(QIODevice::ReadOnly);
 QByteArray allArray = tempFile.readLine();
 QString tempStr = QString(allArray);
 tempFile.close();
 QStringList strList = tempStr.split(" ");
/*读取cpu运行信息*/
 double total_time = 0;
 double use_time = 0;
 for(int i=strList.length()-1;i>1;i--)
     total_time += strList[i].toInt();
 use_time = total_time - strList[5].toDouble();
 double usage = (use_time - old_use_time)/(total_time - old_total_time);
 old_total_time = total_time;
 old_use_time = use_time;

points.append(QPointF(oldPoints.count()-1 ,usage));
cpu_series->replace(points);
ui->label->setText("CPU使用率: " + (new QString("%1"))->arg(100*usage).mid(0,4) + "%");

```

图 4-10 刷新 cpu 使用率曲线代码截图

(9) 内存和交换分区使用率的图形化显示：与 CPU 曲线绘制方式相同，唯一不同的是需要读取的文件为/proc/meminfo，数据的处理方式也有所不同，打开文件查看文件的组织形式，如图 4-11 所示，内存使用情况为前两行数据，内存交换分区的使用情况为 15 和 16 两行的数据，按照之前的方式读取并显示即可。

1	MemTotal:	4000756 kB
2	MemFree:	1052516 kB
3	MemAvailable:	1866636 kB
4	Buffers:	62080 kB
5	Cached:	882964 kB
6	SwapCached:	18288 kB
7	Active:	1421728 kB
8	Inactive:	763452 kB
9	Active(anon):	769976 kB
10	Inactive(anon):	479588 kB
11	Active(file):	651752 kB
12	Inactive(file):	283864 kB
13	Unevictable:	0 kB
14	Mlocked:	0 kB
15	SwapTotal:	2097148 kB
16	SwapFree:	1820472 kB

图 4-11 /proc/meminfo 文件组织形式图

(10) 状态栏显示当前时间：在计算运行时间时，同时计算当前时间并显示。

(11) 状态栏显示当前 cpu 使用率：在绘制 cpu 使用率曲线时，同时将计算出的 cpu 使用率显示到对应的标签中即可。

(12) 状态栏显示当前内存使用情况：绘制内存使用情况曲线时，将使用的内存除以总内存即可得到当前的内存使用率。

(13) 新进程运行其他程序：使用 QTextEdit 读取新进程的路径，然后当点击“创建”按钮就会创建一个新进程，使用 fork 创建子进程，然后使用 execv 函数替换程序即可实现新进程的创建。

(14) 关机功能：按下关机按钮，程序发送“shutdown -s”关闭系统。

5. 组合各个部件，当进行窗口初始化时，调用那些不需要刷新的函数，填充部分系统信息，如主机名等；然后创建时钟，使用时钟事件对需要刷新的系统信息进行定期刷新即可。

4.4 实验调试

4.4.1 实验步骤

1. 按照实验设计的流程设计并实现资源管理器。
2. 调试程序并修改问题。

4.4.2 实验调试及心得

1. 测试结果如图 4-12 所示，正确显示基本系统信息，包括主机名 ubuntu、系统的启动时间、运行时间、CPU 的型号主频以及系统内核的版本号；

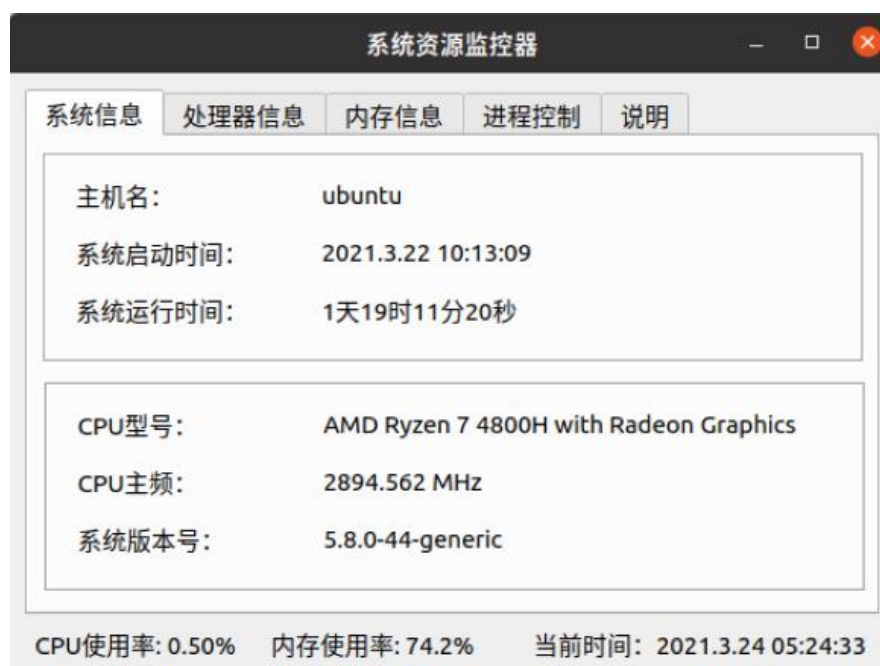


图 4-12 资源管理器系统信息页面效果图

2. cpu 历史使用率曲线的测试结果如图 4-13 所示，由于 Linux 系统的特点，cpu 的使用率并不高，但是可以看到 cpu 的使用率随着时间发生变化，实现了 cpu 使用率的图形化显示功能。同理，内存使用率以及交换区使用率曲线如图 4-14 所示，可以发现成功实现了功能。



图 4-13 资源管理器处理器信息页面效果图

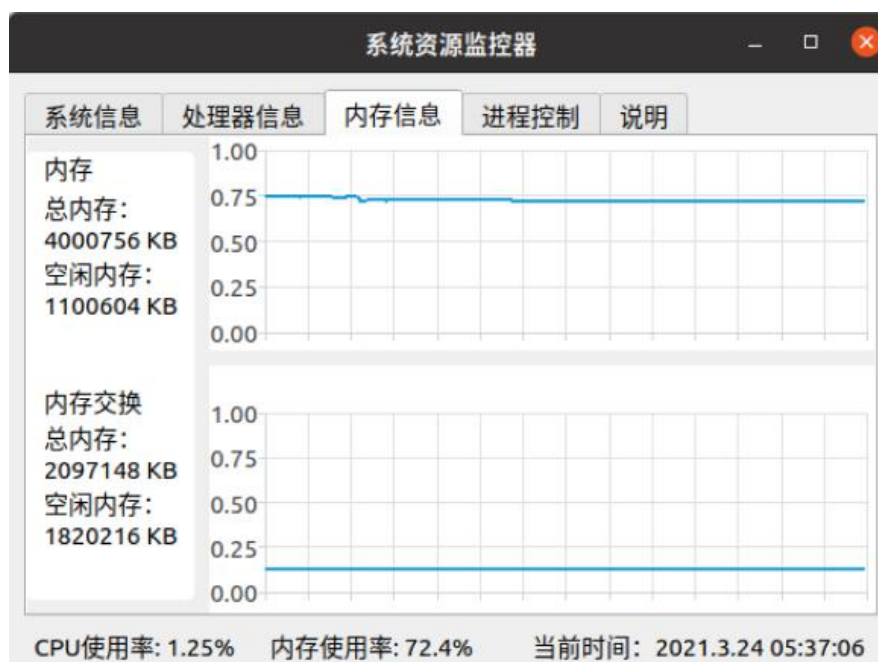


图 4-14 资源管理器内存信息页面效果图

3. 进行进程控制页面的测试，首先打开进程控制界面，如图 4-15 所示，可以看到当前存在 360 个进程，运行中的进程数为 1，睡眠中的进程数为 255；点

击 PID 或者进程名表头，可以发现进程按照该值进行排序，程序功能正常。



图 4-15 资源管理器进程控制页面效果图

4. 说明界面可以启动一个新进程，如图 4-16 所示，当输入新进程的绝对路径后，再点击从“创建”按钮即可打开一个新进程；然后，点击强制关闭即可关闭新创建的进程。



图 4-16 资源管理器说明页面效果图

实验心得:

实验四需要实现一个功能复杂的系统资源管理器，与之前的项目不同，这个

项目需要完成一个具有一定功能的较为复杂的实际项目，是一个比较接近于现实项目设计与实现的场景。

本次实验的主要目的就是掌握 `proc` 文件的特点和使用方法，以及熟悉图形化编程的流程。通过本次实验，我充分了解了 `proc` 文件系统中各个文件的作用以及基本的组织形式，初步掌握了 QT 的图形化编程的流程。

本次实验中我也遇到了很多的问题，由于本次实验涉及到图形化编程，对于图形化编程的不熟悉在编写初期给我制造了相当多障碍，不过通过面向搜索引擎编程后，我很快掌握了基本的图形化编程方法。本次实验我的收获也很多，最重要的收获就是掌握了图形化编程的基本方法。

5 实验五 小型文件系统

5.1 实验目的

使用文件系统的知识，设计并实现一个模拟的文件系统，掌握文件系统的基本结构的实现以及加深对文件系统的了解和掌握。

5.2 实验内容

使用磁盘中的一个文件（大小事先指定）来模拟磁盘，设计并实现一个模拟的文件系统，需要完成如下功能与结构：

1. 设计并实现完善的文件目录项的结构；
2. 实现空白块的管理；
3. 实现文件系统的基本操作，如文件打开删除、目录的创建删除等；
4. 选择支持多用户与树形目录的设计与实现。

5.3 实验设计

5.3.1 开发环境

1. 硬件环境：
 - （1）中央处理器 CPU：AMD Ryzen 7 4800H 2.9 GHz
 - （2）物理内存：16.00 GB
2. 开发编译运行环境（VMware Workstation Pro 16）
 - （1）虚拟机系统版本：Ubuntu18.04 操作系统
 - （2）编译器版本：GCC 7.5.0
 - （3）调试器版本：GNU gdb 8.1.0
 - （4）虚拟机内存：4.00 GB
 - （5）虚拟机磁盘空间：80.00 GB

5.3.2 实验设计

使用 QT 控制台项目开发文件系统，本次实验需要使用一个大文件模拟磁盘进行文件操作，通过规划设计，计划本次将使用大小为 10MB 的文件模拟磁盘，系统磁盘块的大小为 1024B，总磁盘块数为 10240 块，总大小为 10M，可以使用宏定义的方式进行调整，具体文件系统设计如下：

1. 文件系统总体结构：本文件系统支持**多用户与树形目录**，基本磁盘块大小为 1024B，磁盘块数为 10240 个（可以通过宏定义调整），磁盘总大小为 10M，文件组织形式采用串联文件，空白块管理使用空白块链。

使用 0 号块为特殊引导块存放空白块的管理信息以及用户控制信息（多用户），每个用户拥有各自独立的根目录，不可以相互访问，用户之间完全隔离，具体结构如图 5-1 所示。

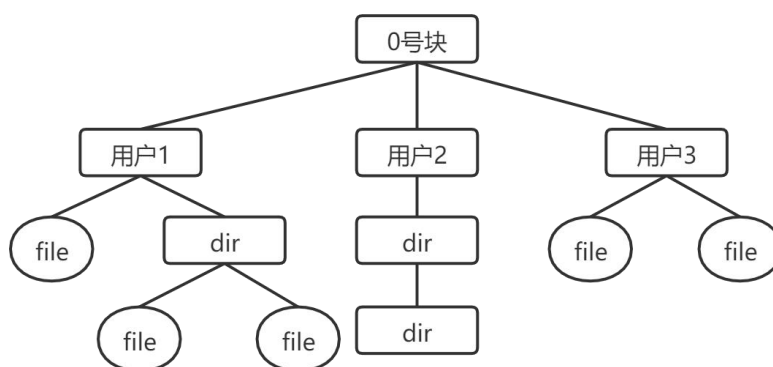


图 5-1 文件系统总体结构示意图

此外，本文件系统支持多项用户、文件以及目录操作，用户操作包括注册、移除、登录以及显示用户信息操作，文件操作包括创建、删除、查看内容以及编辑内容，目录操作包括创建、删除、显示目录下文件以及切换当前目录操作。本系统的所有路径相关操作均支持绝对路径和相对路径操作，且可以解析复杂的文件路径名，操作体验接近于真实的系统。

2. 具体文件组织形式：普通文件的组织形式如图 5-2 所示，采用**串联文件**形式进行组织，每个文件块的头 4 个字节为串联指针，指向下一个文件块，最后一个文件块的串联指针指向 0 号块（NULL）；为简化系统设计与实现，本文件系统规定，每个目录文件最多使用 1 个磁盘块，而普通文件使用的磁盘块个数没有限制，文件可以为任意大小。

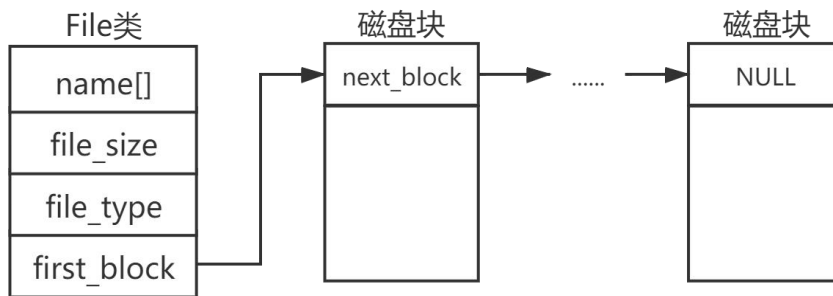


图 5-2 文件组织形式示意图

3. 文件目录项结构：如图 5-2 所示，使用 **File** 类对文件目录项进行封装，每个文件目录项的大小为 32B，文件名占 20 字节，然后文件大小 `file_size`、文件类型 `file_type` (`file_type` 的数值为 `DIR_TYPE` 宏定义表示目录文件，为 `FILE_TYPE` 宏定义表示普通文件) 以及文件块指针 `first_block` 各 4 个字节。

文件目录的基本结构如图 5-3 所示，由于目录最多使用一个磁盘块，因此每一级目录中最多可以容纳 32 个普通文件或目录文件。

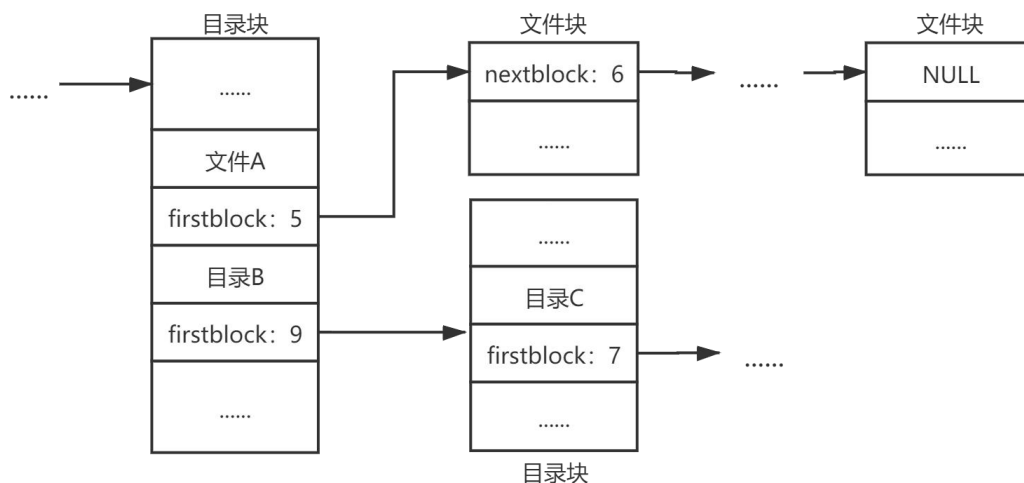


图 5-3 文件目录结构示意图

4. 空白块管理：使用**空白块链**的方式对空白块进行管理，将所有空白块以**链表的形式进行串联**，每个空白块的头 4 个字节作为串联指针，指向下一个空白块，最后一个空白块指向 `NULL` (0 号块)。

使用 0 号块作为特殊引导块管理空白块信息，0 号块的前 4 个字节为空白块链头指针，指向第一个空白块；0 号块的次 4 个字节为空白块的个数，用于管理当前空白块；之后存放用户控制块 `Usr` 的信息，用户控制块定义详见其他数据结构中 (2) 部分定义。

5. 其他数据结构定义如图 5-4 所示，具体定义与作用如下：

```
/* 磁盘块类(1024B) */
struct Block{
    char buffer[BLOCK_SIZE];
};

struct Usr {
    char username[MAX_NAME_LENGTH];
    char passname[MAX_PSW_LENGTH];
    int root_size; /* 根目录的大小 */
    int vaild; /* 有效性 */
    int root_block; /* root_block指向根目录块 */
};

struct File_control_block {
    int block_num;
    int file_index;
};
```

图 5-4 其他数据结构定义图

(1) 磁盘块 Block 类：对 char 型的缓冲区进行简单封装，缓冲区大小使用宏定义 BLOCK_SIZE 确定，本次实验使用的缓冲区大小为 1024 字节。在本文件系统中，Block 类的作用是结构化读取磁盘文件以及方便使用指针对文件目录项 File 结构进行操作。

(2) 用户控制块 Usr 类：用户控制块用于存储用户基本信息以及控制用户的登录等操作，包含 5 个成员分别是两个长度为 10 字节的 char 型数组，存储用户名以及用户密码，根目录大小 root_size，用户控制块有效位标记 vaild 以及根目录文件块指针 root_block，指向根目录块。

为方便进行块操作，在设计用户控制块 Usr 类时，特意将其大小设计为 32 字节（与 File 类的大小相同），并且严格限制成员排列方式，将文件大小以及文件块指针的位置设置在相同的位置，方便使用 File 类的指针对其进行操作。

(3) 文件控制块 File_control_block 类：用于文件控制，该类具有两个成员分别是文件目录块号 block_num 以及文件索引号 file_index，分别指向当前文件所存在的文件目录块的块号以及当前文件在此目录中的索引。

6. 文件系统的初始化与磁盘创建：每次进行文件系统初始化时，首先使用 r+ 的模式使用 fopen 函数打开磁盘模拟文件；如果文件不存在，则需要使用 a+ 的方式创建磁盘文件，并将磁盘文件全部初始化为 0；如果文件存在，则直接使用 r+ 的模式使用 fopen 函数打开磁盘模拟文件即可进行操作。

7. 文件系统基本操作：文件系统需要支持一些基本操作，如将指定块从磁盘读到内存中，将内存中指定块的内容重新写入磁盘块中等操作，本文件系统定

义了 7 种基本操作，具体定义如图 5-5 所示，包括磁盘指定块读写，空白块的获取与释放，文件目录项指针设置以及 0 号块的更新功能。

```
/* 文件系统基本操作 */
void read_from_Disc(int block, char* buf); /* 从磁盘读取数据 */
void write_to_Disc(int block, char* buf); /* 向磁盘写入数据 */
int get_block(); /* 获取空白块 */
int release_block(int block_num, int File_type); /* 释放数据块 */
int set_File_Pointer(File**file, int block_num, int index, Block &block);
int get_zero_block();
int update_zero_block();
```

图 5-5 文件系统基本操作定义代码图

8. 文件系统拓展操作：文件系统的拓展操作包括文件操作、目录操作以及用户操作这三类操作，由于本次实现的文件系统拓展操作数量较多，因此无法完全在此进行详细描述，本次报告仅做简述，详细参见代码。

(1) 用户操作：用户操作包括注册、移除、登录以及显示用户信息，以登录操作为例，当系统检测到用户输入“login”命令时，会调用 log_in 函数进行登录，使用循环的方式查询用户信息，然后如图 5-6 所示，对用户名以及密码进行比对，当两者都相同时，表示登录成功需要将 0 号块的引导信息加载到内存中，需要设置包括根目录、当前目录以及空白块指针等数据。

```
if(!strcmp(currUstr[usr_num].username, user.c_str()))
{
    cout << "用户名正确!" << endl;
    if(!strcmp(currUstr[usr_num].passname, pass.c_str()))
    {
        cout << "密码正确!" << endl;
        use_num = usr_num;
        /* 设置当前的根目录 读取引导块 */
        read_from_Disc(0, zero_block.buffer);
        currUstr = (Ustr*)&zero_block.buffer[8];
        rootDir = (File*)&currUstr[usr_num];

        /* 设置当前目录以及临时目录的目录块 */
        currentDir=rootDir;
        currentDir_num=0;
        currentDir_index=usr_num;
        tempDir=rootDir;
        tempDir_num=0;
        tempDir_index=usr_num;

        /* 加载空白块处理逻辑 */
        read_from_Disc(0, buf);
        bk_fhead = *((int*)buf); /* 空白块头指针 */
        bk_nfree = ((int*)buf)[1]; /* 空白块数 */
        usr_name = currUstr[usr_num].username;
        return USR_EXIST;
    }
}
```

图 5-6 用户登录函数部分代码图

(2) 文件操作：文件操作包括创建（touch）、删除（rm）、查看内容（cat）以及编辑内容（gedit），文件操作支持复杂路径的解析，由于文件操作的实现比较复杂，为提高系统的拓展性，本文件系统将常见的文件操作封装成了 4 个基本函数 open_file、close_file、read_from_file 以及 write_to_file。

以 open_file 文件打开函数为例，open_file 函数使用使用文件路径进行文件打开，使用 mode 参数区分创建模式，如图 5-7 所示，当 mode 为 READ_WRITE 时，表示仅查找当前文件，不存在时返回异常；当 mode 为 CREATE 时，表示文件不存在时需要创建文件与文件路径上的目录文件，创建异常分为 3 种，分别是文件名冲突、空白块耗尽以及目录已满。



```
/* 打开文件函数
 * filepath: 文件路径，仅支持相对路径
 * mode: 打开方式控制变量
 *     READ_WRITE: 对文件进行读写，文件不存在时返回异常
 *     CREATE: 创建文件（可读写），不存在时创建文件与目录
 * 正常返回值int: 若当前文件存在，或创建成功则返回对应的编号
 * 异常返回int: block_num为-1表示目录名与文件名冲突，
 *             为-2表示当前空白块耗尽，为-3表示目录已满
 */
File_control_block* File_system::open_file(string &filepath, int mode)
```

图 5-7 open_file 函数函数声明图

其他函数如 close_file、read_from_file 等的实现详见代码注释，函数实现在代码中有详细的注释，在此不再赘述。

(3) 目录操作：目录操作包括创建（mkdir）、删除（rmdir）、显示目录下文件（ls）以及切换当前目录操作（cd），目录操作的函数实现在代码中有详细的注释，在此不再赘述。

5.4 实验调试

5.4.1 实验步骤

1. 按照实验设计进行文件系统的实现，编译并生成可执行文件，编写测试文件，按照测试文件进行测试。
2. 按照测试文件的流程测试程序，找到问题调试程序并修改问题。

5.4.2 实验调试及心得

打开终端，使用命令“g++ -o disc disc.cpp”将文件系统源文件 disc.cpp 编译为可执行文件，然后输入命令“./disc”打开可执行文件，如图 5-8 所示，进入文件系统用户操作界面，在此界面可进行登录、注册等操作，具体功能测试如下：



图 5-8 文件系统初始用户操作界面图

1. 测试用户注册功能：如图 5-9 所示，输入命令“register lx 123456”进行用户注册，系统提示“用户创建成功！”表示当前用户创建成功；这时如果进行重复注册，输入命令“register lx 123”，系统会提示“用户已存在！”；此时，输入“show”即可查看当前的用户信息，可以知道当前空白块个数为 10238 块，用户“lx”的根目录块号为 1。

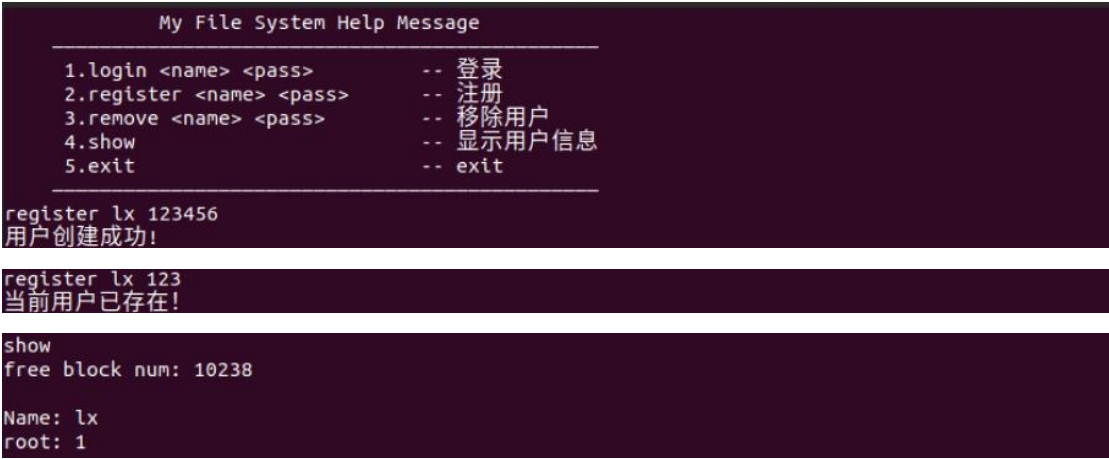


图 5-9 文件系统用户注册功能测试界面图

2. 测试用户登录功能：如图 5-10 所示，如果输入的用户名不存在，系统提示“登录失败！”；如果当前用户存在，但是密码不正确，系统提示“用户名正确！登录失败！”；如果输入的用户名和密码都正确，系统提示“用户名正确！密码正确！”然后进入文件系统的文件及目录操作界面。当正常进入操作界面后，屏幕会显示文件系统信息提示界面，并显示“lx@ubuntu:\$”表示当前用户为 lx，目录为当前用户的根目录。

```
My File System Help Message
1.login <name> <pass>      -- 登录
2.register <name> <pass>   -- 注册
3.remove <name> <pass>     -- 移除用户
4.show                     -- 显示用户信息
5.exit                     -- exit

login llx 123
登录失败!

login lx 123
用户名正确!
登录失败!

login lx 123456
用户名正确!
密码正确!

My File System Help Message
1.touch <filename>         -- 新建文件
2.rm <filename>            -- 删除文件
3.cat <filename>           -- 查看文件
4.gedit <filename>         -- 编辑文件
5.mkdir <dirname>          -- 新建目录
6.rmdir <filename>         -- 移除目录
7.ls                       -- 显示目录文件
8.cd <dirname>             -- 进入目录
9.clear                    -- 清除屏幕内容
10.help                    -- 显示帮助
11.exit                    -- 注销用户

lx@ubuntu:~$
```

图 5-10 文件系统用户登录功能测试界面图

3. 文件创建功能测试，如图 5-11 所示：

```
lx@ubuntu:~$ touch 123
lx@ubuntu:~$ ls
lx/:
FILE 123  0 B
lx@ubuntu:~$ touch a/b/122
lx@ubuntu:~$ cd a/b
lx@ubuntu:~/a/b$ ls
lx/a/b/:
FILE 122  0 B
lx@ubuntu:~/a/b$ touch /111
lx@ubuntu:~/a/b$ ls
lx/a/b/:
FILE 122  0 B
lx@ubuntu:~/a/b$ cd ../../
lx@ubuntu:~$ ls
lx/:
FILE 123  0 B
DIR a    96 B
FILE 111  0 B
lx@ubuntu:~$
```

图 5-11 文件系统文件创建功能测试界面图

(1) 使用单文件名创建文件：输入命令“touch 123”使用 touch 命令创建文件 123，使用命令“ls”查看当前目录下的文件，新文件 123 创建成功，文件类型为 FILE，文件大小为 0；

(2) 使用相对路径创建文件：输入命令“touch a/b/122”在目录 a/b 下创建文件 122，使用命令“cd a/b”进入 a/b 目录后使用命令“ls”即可查看当前目录下文件，文件 122 创建成功；

(3) 使用绝对路径创建文件：输入命令“touch /111”在根目录下创建文件 111，使用 cd 命令进入根目录，并使用 ls 命令查看当前目录下文件，文件 111 创建成功。

4. 文件编辑与查看功能测试：如图 5-12 所示，使用命令“gedit 123”编辑文件 123，输入以回车结尾的字符串，然后使用命令“cat 123”查看文件 123 的内容，可以发现文本内容正确；重新使用 gedit 命令编辑文件，可以发现文件内容被覆盖，文件编辑与查看功能实现成功。

```
lx@ubuntu:~$ gedit 123
请输入需要输入的文本(\n结尾):
123456789123456789123456789123456789123456789
lx@ubuntu:~$ cat 123
123456789123456789123456789123456789123456789
lx@ubuntu:~$ gedit 123
123456789123456789123456789123456789123456789
请输入需要输入的文本(\n结尾):
123456
lx@ubuntu:~$ cat 123
123456
lx@ubuntu:~$
```

图 5-12 文件系统文件查看与编辑功能测试界面图

5. 文件删除功能测试，如图 5-13 所示：

(1) 使用单文件名删除文件：输入命令“rm 123”删除文件 123，系统发出提示“文件删除成功！”，使用 ls 命令查看文件，文件 123 删除成功；

(2) 使用相对路径删除文件：输入命令“rm a/b/122”删除文件 122，系统发出提示“文件删除成功！”，同样使用 cd 命令进入 a/b 目录并使用 ls 查看文件，文件 122 删除成功，使用命令“cd /”回到根目录目录 a 未删除。

```
lx@ubuntu:~$ rm 123
文件删除成功!
lx@ubuntu:~$ ls
lx/:
DIR a    96 B
FILE 111 0 B
lx@ubuntu:~$ rm a/b/122
文件删除成功!
lx@ubuntu:~$ cd a/b
lx@ubuntu:a/b$ ls
lx/a/b/:
lx@ubuntu:a/b$ cd /
lx@ubuntu:~$ ls
lx/:
DIR a    96 B
FILE 111 0 B
lx@ubuntu:~$
```

图 5-13 文件系统文件删除功能测试界面图

6. 目录创建功能测试，如图 5-14 所示：

(1) 使用单目录名创建目录：输入命令“mkdir aaa”创建目录，使用 ls 命

令查看当前目录下的文件，目录 `aaa` 创建成功；若输入“`mkdir 111`”创建于文件名称相同的目录，系统发出提示“目录名与文件名冲突！”，无法创建目录；

(2) 使用相对路径创建目录：输入“`mkdir a/b/ccc`”创建目录，使用 `cd` 命令并使用 `ls` 命令查看文件，目录 `ccc` 创建成功；

(3) 使用绝对路径创建目录：输入命令“`mkdir /bbb`”使用绝对路径创建文件，使用 `cd` 命令进入根目录并使用 `ls` 命令查看文件，目录 `bbb` 创建成功。

```
lx@ubuntu:~$mkdir aaa
lx@ubuntu:~$ls
lx/:
DIR a 96 B
FILE 111 0 B
DIR aaa 64 B
lx@ubuntu:~$mkdir 111
目录名与文件名冲突!
lx@ubuntu:~$mkdir a/b/ccc
lx@ubuntu:~$cd a/b
lx@ubuntu:a/b$ls
lx/a/b/:
DIR ccc 64 B
lx@ubuntu:a/b$cd /
lx@ubuntu:/$cd a/b
lx@ubuntu:a/b$mkdir /bbb
lx@ubuntu:a/b$cd ../../
lx@ubuntu:/$ls
lx/:
DIR a 96 B
FILE 111 0 B
DIR aaa 64 B
DIR bbb 64 B
lx@ubuntu:/$
```

图 5-14 文件系统目录创建功能测试界面图

7. 目录移除功能测试，如图 5-15 所示：

(1) 使用单目录名删除目录：输入命令“`rmdir bbb`”，系统发出提示“目录删除成功！”，使用命令 `ls` 查看文件，目录 `bbb` 删除成功；若输入“`rmdir 111`”删除普通文件，系统发出提示“目录删除失败！”；

(2) 使用相对路径删除目录：输入命令“`rmdir a/b`”，系统发出提示“目录删除成功！”，使用命令 `cd` 进入 `a/b` 目录，使用 `ls` 查看文件，目录删除成功；

```
lx@ubuntu:~$rmdir bbb
目录删除成功!
lx@ubuntu:~$ls
lx/:
DIR a 96 B
FILE 111 0 B
DIR aaa 64 B
lx@ubuntu:~$rmdir 111
目录删除失败!
lx@ubuntu:~$rmdir a/b
目录删除成功!
lx@ubuntu:~$ls
lx/:
DIR a 64 B
FILE 111 0 B
DIR aaa 64 B
```

图 5-15 文件系统目录删除功能测试界面图

8. 用户移除功能测试：如图 5-15 所示，输入命令“remove lx 123”，当密码不正确时，系统提示移除失败，如果输入命令“remove lx 123456”，系统提示“用户名正确！密码正确！”，使用 show 查看当前用户情况，可以发现空白块恢复为初始状态 10239 块，且无用户存在。

```
My File System Help Message
1.login <name> <pass>      -- 登录
2.register <name> <pass>   -- 注册
3.remove <name> <pass>     -- 移除用户
4.show                     -- 显示用户信息
5.exit                     -- exit

remove lx 123
用户名正确！
移除失败！

remove lx 123456
用户名正确！
密码正确！

show
free block num: 10239
```

图 5-15 文件系统用户移除功能测试界面图

实验心得：

实验五需要使用单一大文件模拟磁盘实现小型文件系统，本项目是一个综合性的项目，不仅需要扎实的编程基础，还需要对操作系统文件系统方面的知识有比较深入的了解。

在进行本次文件系统的设计时，刚开始我打算实现一个比较接近于 UNIX 文件系统的小型文件系统，使用索引文件、i 节点表以及空白块成链法管理空白块，但是分析后发现可行性较差，我发现很难在一周内实现这样复杂的文件系统，因此，我只能退而求其次，使用串联文件以及空白块链的方式实现文件系统。

由于文件系统比较复杂，各个操作之间存在着关联，错误的发现和纠错也比较困难，因此花费了大量时间在程序的调试和修复上，但是经过我的不懈努力，最终完成了这个项目，初步实现了文件系统的基本功能。

通过这次实验，我充分了解到了不同文件系统组织形式之间的不同，加深了我对文件系统的理解。

附录 实验代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string>
#define DISC_NAME          "DISE"
#define BLOCK_SIZE          1024
#define STORAGE_SIZE        10240

#define MAX_NAME_LENGTH     10
#define MAX_PSW_LENGTH      10

#define MAX_USR_NUM         32
#define MAX_FILE_NUM        32
#define MAX_USR_NUM         32

#define USR_EXIST            1
#define USR_NOT_EXIST       3
#define USR_FULL             2
#define USR_CREATE_SUCCESS  0
#define USR_CREATE_FAIL     -1

#define FILE_TYPE            0
#define DIR_TYPE             1

#define READ_WRITE           1
#define CREATE               2
#define SEARCH               3

using namespace std;

/* 磁盘块类(1024B) */
struct Block{
    char buffer[BLOCK_SIZE];
};

struct Usr {
    char username[MAX_NAME_LENGTH];
    char passname[MAX_PSW_LENGTH];
    int  root_size; /* 根目录的大小 */
    int  vaild;     /* 有效性 */
    int  root_block; /* root_block 指向根目录块 */
}
```

```

};
struct File_control_block {
    int block_num;
    int file_index;
};
struct File {
    char name[MAX_NAME_LENGTH*2];
    int file_size;
    int file_type;
    int first_block; /* first_block 指向文件第一个块 */
};
class File_system
{
private:
    int use_num;          /* 当前用户编号 */
    FILE* fdisc;          /* 磁盘文件指针 */

    /* 目录管理 */
    File* rootDir;        /* 根目录 */
    File* currentDir;     /* 当前目录 */
    File* tempDir;        /* 临时目录-用于路径解析 */

    Block zero_block;     /* 0号引导块 */
    Block temp_block;     /* 临时块 */

    int currentDir_num;    /* 当前目录块号 */
    int currentDir_index; /* 当前目录编号 */
    int tempDir_num;       /* 临时目录块号 */
    int tempDir_index;     /* 临时目录编号 */

    /* 文件控制块 */
    File_control_block file_control;
    File* currentFile;

    /* 空白块管理 */
    int bk_fhead;          /* 空白块头指针 */
    int bk_nfree;          /* 空白块数 */

public:
    string usr_name;
    string dir_path;

    File_system();
    ~File_system();

```



```

/* 文件系统基本操作 */
void read_from_Disc(int block, char* buf); /* 从磁盘读取数据 */
void write_to_Disc(int block, char* buf); /* 向磁盘写入数据 */
int get_block(); /* 获取空白块 */
int release_block(int block_num, int File_type); /* 释放数据块 */
/* 设置文件指针 */
int set_File_Pointer(File**file, int block_num, int index, Block&block);
int get_zero_block();
int update_zero_block();

/* 文件操作 */
File_control_block* open_file(string &filepath, int mode);
int read_from_file(File_control_block* file,
                  int size, char *buf, int pos);
int write_to_file(File_control_block* file,
                  int size, const char* buf, int pos);
int clear_file(File_control_block* file, int mode);
File_control_block* close_file(File_control_block* file);

/* 目录操作 */
int create_dir(string &filename, int mode);
int remove_dir(string &filename);
int delete_dir_dfs(int block, int file_size);
int list_files();
int cd_dir(string &filepath);

/* 用户操作 */
int log_in(string &user, string &pass);
int log_out();
int Register(string &user, string &pass);
int Remove(string &user, string &pass);
int show_Usr();

/* 拓展操作 */
void touch(string &filepath);
void rm(string &filepath);
void cat(string &filepath);
void gedit(string &filepath);
void mkdir(string &filepath);
void rmdir(string &filepath);
void show_help();
};

```



```

File_system::~~File_system()
{
    fclose(fdisc);
}
File_system::File_system()
{
    /* 创建磁盘文件
     * 0 号块为引导块 */
    fdisc = nullptr;
    if(!(fdisc = fopen(DISC_NAME, "r+"))){
        fdisc = fopen(DISC_NAME, "a+");
        /* 如果磁盘文件未分配空间 */
        char buf[BLOCK_SIZE];
        memset(buf, 0, BLOCK_SIZE);
        /* 文件初始化与空白块成链 */
        for(int i=0; i<STORAGE_SIZE; i++){
            if(i==STORAGE_SIZE-1) *((int*)buf) = 0;
            else *((int*)buf) = i+1;
            fwrite(buf, sizeof(char), BLOCK_SIZE, fdisc);
        }
        fclose(fdisc);
        fdisc = fopen(DISC_NAME, "r+");
        /* 设置空白块的块数 */
        fread(buf, sizeof(char), BLOCK_SIZE, fdisc);
        *((int*)buf)[1] = STORAGE_SIZE - 1;
        fseek(fdisc, 0, SEEK_SET);
        fwrite(buf, sizeof(char), BLOCK_SIZE, fdisc);
    }
    fseek(fdisc, 0, SEEK_SET);
}

/* 文件系统基本操作 */

/* 从磁盘读取数据
 * block: 磁盘块号
 * buf: 磁盘块数据缓冲区
 */
void File_system::read_from_Disc(int block, char* buf)
{
    fseek(fdisc, block*BLOCK_SIZE, SEEK_SET);
    fread(buf, sizeof(char), BLOCK_SIZE, fdisc);
}

```

```

/* 向磁盘写入数据
 * block: 磁盘块号
 * buf: 磁盘块数据缓冲区
 */
void File_system::write_to_Disc(int block, char* buf)
{
    fseek(fdisc, block*BLOCK_SIZE, SEEK_SET);
    fwrite(buf, sizeof(char), BLOCK_SIZE, fdisc);
}

/* 获取空白块
 * 返回值 int: 正常返回可用的空白块号, 异常返回 0
 */
int File_system::get_block()
{
    int temp_bk=bk_fhead;
    if(bk_nfree==0) return 0;
    else
    {
        char temp_buf[BLOCK_SIZE];
        read_from_Disc(temp_bk, temp_buf);
        bk_fhead = *((int*)temp_buf);
        bk_nfree--;
        return temp_bk;
    }
}

/* 释放数据块
 * block_num: 文件的首个磁盘块号
 * File_type: 文件类型
 */
int File_system::release_block(int block_num,int File_type)
{
    Block block;
    if(File_type==DIR_TYPE)
    {
        read_from_Disc(block_num, block.buffer);
        *((int*)block.buffer)=bk_fhead;
        write_to_Disc(block_num, block.buffer);
        bk_fhead=block_num;
        bk_nfree++;
        return 0;
    }
}

```

```

    while(block_num!=0)
    {
        read_from_Disc(block_num, block.buffer);
        int tempnum=((int*)block.buffer);
        *((int*)block.buffer)=bk_fhead;
        write_to_Disc(block_num, block.buffer);
        bk_fhead=block_num;
        block_num=tempnum;
        bk_nfree++;
    }
    return 0;
}

/* 设置文件指针
 * file: 文件双重指针
 * block_num: 文件块号
 * index: 文件索引
 * block: 缓冲区
 */
int File_system::set_File_Pointer(File**file,
                                   int block_num, int index, Block &block)
{
    /* 从磁盘中加载当前文件目录块 */
    read_from_Disc(block_num, block.buffer);
    if(!block_num) /* 当前块为 0 号引导块 */
        *file = &((File*)&block.buffer[8])[index];
    else *file = &((File*)block.buffer)[index];
    return 0;
}

/* 获取 0 号引导块
 * 将 0 号引导块加载到内存
 */
int File_system::get_zero_block()
{
    read_from_Disc(0, zero_block.buffer);
    return 0;
}

/* 更新 0 号引导块
 * 更新空白块情况
 */
int File_system::update_zero_block()
{

```

```

    char buf[BLOCK_SIZE];
    /* 更新引导块 */
    fseek(fdisc, 0, SEEK_SET);
    read_from_Disc(0, buf);
    *((int*)buf) = bk_fhead; /* 空白块头指针 */
    ((int*)buf)[1] = bk_nfree; /* 空白块数 */
    write_to_Disc(0, buf);
    return 0;
}

/* 文件操作 */

/* 打开文件函数
 * filepath: 文件路径，仅支持相对路径
 * mode: 打开方式控制变量
 *     READ_WRITE: 对文件进行读写，文件不存在时返回异常
 *     CREATE: 创建文件（可读写），不存在时创建文件与目录
 * 正常返回值 int: 若当前文件存在，或创建成功则返回对应的编号
 * 异常返回 int: block_num 为-1 表示目录名与文件名冲突，
 *               为-2 表示当前空白块耗尽，为-3 表示目录已满
 */
File_control_block* File_system::open_file(string &filepath, int mode)
{
    /* 设置临时目录为当前目录 */
    tempDir_num=currentDir_num;
    tempDir_index=currentDir_index;
    set_File_Pointer(&tempDir,tempDir_num,tempDir_index,temp_block);

    while (true)
    {
        /* 如果开头为/ */
        if(filepath[0]=='/')
        {
            /* 设置临时目录为根目录 */
            tempDir_num=0;
            tempDir_index=use_num;
            set_File_Pointer(&tempDir,tempDir_num,
                            tempDir_index,temp_block);
            filepath = filepath.substr(1);
        }
        else if(filepath[0]=='.'&&filepath[1]=='/') /* 如果开头为./ */
            filepath = filepath.substr(2);
    }
}

```

```

/* 如果无下级目录 */
if(filepath.find('/')==filepath.npos)
{
    /* filelist 指向文件目录项 */
    Block filelist_block;
    read_from_Disc(tempDir->first_block,
                  filelist_block.buffer);
    File* filelist = (File*)filelist_block.buffer;
    int file_num = tempDir->file_size/32;

    /* 查询是否已经存在该文件 */
    for(int i=0;i<file_num;i++)
    {
        if(!strcmp(filelist[i].name, filepath.c_str()))
        {
            File_control_block* temp_control =
                new File_control_block;
            if(filelist[i].file_type==FILE_TYPE)
            {
                /* 返回文件控制块信息 */
                temp_control->block_num=tempDir->first_block;
                temp_control->file_index=i;
                return temp_control;
            }
            else
            {
                /* 当前文件名称与目录冲突 */
                temp_control->block_num=-1;
                temp_control->file_index=0;
                return temp_control;
            }
        }
    }
    if(mode==READ_WRITE)
    {
        /* READ_WRITE 状态下, 当前文件不存在, 返回异常 */
        File_control_block* temp_control =
            new File_control_block;
        temp_control->block_num=0;
        temp_control->file_index=0;
        return temp_control;
    }
}

```

```

if(tempDir->file_size>=1024)
{
    /* 当前目录已满 */
    File_control_block* temp_control =
        new File_control_block;
    temp_control->block_num=-3;
    temp_control->file_index=0;
    return temp_control;
}

/* 从磁盘申请空白块 */
int newblock = get_block();
if(!newblock)
{
    /* 当前空间已满 */
    File_control_block* temp_control =
        new File_control_block;
    temp_control->block_num=-2;
    temp_control->file_index=0;
    return temp_control;
}
/* 修改文件头文件块的指针 */
Block temp;
read_from_Disc(newblock, temp.buffer);
*((int*)temp.buffer)=0;
write_to_Disc(newblock, temp.buffer);
/* 修改并更新文件目录项 */
tempDir->file_size += 32;
write_to_Disc(tempDir_num, temp_block.buffer);

/* 创建新文件并更新文件块 */
File* newFile = &filelist[file_num];
newFile->first_block=newblock;
newFile->file_size = 0;
newFile->file_type = FILE_TYPE;
strcpy(newFile->name,filepath.c_str());
write_to_Disc(tempDir->first_block,
               filelist_block.buffer);
File_control_block* temp_control =
    new File_control_block;
temp_control->block_num=tempDir->first_block;
temp_control->file_index=file_num;
return temp_control;

```

```

    }

    /* 路径切分解析 */
    string spilt=filepath.substr(0, filepath.find('/'));
    filepath=filepath.substr(filepath.find('/')+1);
    tempDir_index=create_dir(spilt, CREATE);
    if(tempDir_index < 0)
    {
        File_control_block* temp_control = new File_control_block;

        temp_control->block_num=-4;
        temp_control->file_index=0;
        return temp_control;
    }

    /* 更新当前文件目录块 */
    write_to_Disc(tempDir_num, temp_block.buffer);
    /* 加载下一级文件目录块 */
    tempDir_num = tempDir->first_block;
    set_File_Pointer(&tempDir, tempDir_num,
                    tempDir_index, temp_block);
}
}

/* 读取函数
 * file: 文件控制块指针，指向文件控制块
 * size: 读取的数据大小
 * buf: 缓冲区
 * pos: 读取数据的偏移
 * 返回值 int: 正常返回读取数据的大小，异常返回-1
 */
int File_system::read_from_file(File_control_block* file,
                                int size, char *buf, int pos)
{
    /* 当前文件无法读取 */
    if(file->block_num <= 0) return -1;
    if(pos < 0 || size < 0 || buf == nullptr) return -1;

    /* 读取数据到缓冲区 */
    Block block;
    read_from_Disc(file->block_num, block.buffer);
    File* pfile = &((File*)block.buffer)[file->file_index];

    int tempblock=pfile->first_block;

```

```

    int tempsize=pfile->file_size;
    read_from_Disc(tempblock, block.buffer);

    int index=0;
    int block_length=BLOCK_SIZE-4;
    while(tempsize!=0)
    {
        if(!size) return index;
        if(!block_length){
            block_length=BLOCK_SIZE-4;
            tempblock=((int*)block.buffer);
            read_from_Disc(tempblock, block.buffer);
        }
        if(!pos){
            buf[index] = block.buffer[BLOCK_SIZE-block_length];
            index++;
            size--;
        }
        else pos--;
        tempsize--;
        block_length--;
    }
    return index;
}

/* 写入函数
 * file: 文件控制块指针，指向文件控制块
 * size: 读取的数据大小
 * buf: 缓冲区
 * pos: 读取数据的偏移，超过文件大小默认为文件末尾
 * 返回值 int: 正常返回读取数据的大小，异常返回-1
 */
int File_system::write_to_file(File_control_block* file,
                               int size, const char* buf, int pos)
{
    /* 当前文件无法写入 */
    if(file->block_num <= 0) return -1;
    if(pos < 0 || size < 0 || buf == nullptr) return -1;

    /* 创建中间缓冲区，读取原数据 */
    Block block;
    read_from_Disc(file->block_num, block.buffer);
    File* pfile = &((File*)block.buffer)[file->file_index];
    int tempsize=pfile->file_size;

```



```

if(pos>tempsize) pos=tempsize; /* 超过文件大小默认为文件末尾 */
int buf_length = ((pos+size) > tempsize) ? (pos+size) : tempsize;
char *tempbuf = new char[buf_length];

read_from_file(file, pos, tempbuf, 0);
for(int i=pos;i<pos+size;i++)
    tempbuf[i]=buf[i-pos];
if((pos+size) < tempsize)
    read_from_file(file, tempsize, tempbuf+pos+size, pos+size);

/* 申请空白块 */
int tempblock=get_block();
if(!tempblock)
{
    delete [] tempbuf;
    return -1;
}

/* 加载磁盘块，将当前块设置为头块 */
read_from_Disc(tempblock, block.buffer);
*((int*)block.buffer)=0;
int head_block=tempblock;

/* 刷新文件 */
tempsize=buf_length;
int block_length=BLOCK_SIZE-4;
while(tempsize)
{
    /* 原空白块已满，申请新空白块 */
    if(!block_length)
    {
        block_length=BLOCK_SIZE-4;
        int temp=get_block();
        /* 空白块申请失败 */
        if(!temp){
            *((int*)block.buffer)=0;
            write_to_Disc(tempblock, block.buffer);
            release_block(head_block, FILE_TYPE);
            delete [] tempbuf;
            return -1;
        }
    }

    /* 更新磁盘块 */
    *((int*)block.buffer)=temp;
}

```

```

        write_to_Disc(tempblock, block.buffer);
        /* 加载磁盘块 */
        read_from_Disc(temp, block.buffer);
        tempblock=temp;
    }

    block.buffer[BLOCK_SIZE-block_length]=
                                tempbuf[buf_length-tempsize];
    block_length--;
    tempsize--;
}

/* 更新磁盘块 */
*((int*)block.buffer)=0;
write_to_Disc(tempblock, block.buffer);

read_from_Disc(file->block_num, block.buffer);
pfile = &((File*)block.buffer)[file->file_index];

/*释放原有的空白块*/
release_block(pfile->first_block, FILE_TYPE);

/* 更新文件目录块 */
pfile->file_size=buf_length;
pfile->first_block=head_block;
write_to_Disc(file->block_num, block.buffer);
delete [] tempbuf;
return 0;
}

/* 文件关闭函数
 * file: 文件控制块指针, 指向文件控制块
 * 返回值 int: 正常返回 nullptr, 异常返回-1
 */
File_control_block* File_system::close_file(File_control_block* file)
{
    delete file;
    return nullptr;
}

/* 文件内容清空函数
 * file: 文件控制块指针, 指向文件控制块

```

```

    * mode: mode 为 1 时表示删除文件, 不保留任何块; mode 为 2 时表示清除内容, 保留头
    块
    * 返回值 int: 正常返回 0, 异常返回 -1
    */
int File_system::clear_file(File_control_block* file, int mode)
{
    Block block;
    read_from_Disc(file->block_num, block.buffer);
    File* pfile = &((File*)block.buffer)[file->file_index];

    /*释放原有的空白块*/
    release_block(pfile->first_block, FILE_TYPE);

    if(mode==1) return 0;
    /* 从磁盘申请空白块 */
    int newblock = get_block();
    if(!newblock) return -1;
    /* 修改文件头文件块的指针 */
    Block temp;
    read_from_Disc(newblock, temp.buffer);
    *((int*)temp.buffer)=0;
    write_to_Disc(newblock, temp.buffer);

    /* 更新文件目录块 */
    pfile->file_size=0;
    pfile->first_block=newblock;
    write_to_Disc(file->block_num, block.buffer);
    return 0;
}

/* 目录操作 */

/* 目录创建函数, 使用之前需要设置 tempDir
* filename: 目录名
* mode: 打开方式控制变量
* SEARCH: 查找当前目录, 不存在时返回异常 -1
* CREATE: 创建目录, 不存在时创建目录
* 正常返回值 int: 若当前文件存在, 或创建成功则返回对应的目录项编号
* 异常返回 int: 返回 -1 表示目录名与文件名冲突, 返回 -2 表示当前空白块耗尽, 返回
-3 表示目录已满
*/
int File_system::create_dir(string &filename, int mode)
{

```

```

int file_num = tempDir->file_size/32;
/* filelist 指向文件目录项 */
Block filelist_block;
read_from_Disc(tempDir->first_block, filelist_block.buffer);
File* filelist = (File*)filelist_block.buffer;

filelist[0].file_size = tempDir_index;
filelist[0].first_block = tempDir_num;
Block block; File* temp_file;
set_File_Pointer(&temp_file, tempDir_num, 0, block);
if(!tempDir_num)
{
    filelist[1].file_size = tempDir_index;
    filelist[1].first_block = tempDir_num;
}
else {
    filelist[1].file_size = temp_file->file_size;
    filelist[1].first_block = temp_file->first_block;
}

write_to_Disc(tempDir->first_block, filelist_block.buffer);

read_from_Disc(tempDir->first_block, filelist_block.buffer);
filelist = (File*)filelist_block.buffer;

/* 查询是否已经存在该目录 */
for(int i=0;i<file_num;i++)
{
    if(!strcmp(filelist[i].name, filename.c_str()))
    {
        if(filelist[i].file_type==DIR_TYPE) return i;
        else return -1;
    }
}

/* SEARCH 状态下, 当前目录不存在, 返回异常 */
if(mode==SEARCH) return -1;

/* CREATE 状态下, 当前目录不存在
 * 当前目录不存在, 需要创建当前目录
 * 若当前目录已满, 则创建失败
 * 否则, 将创建当前目录
 */
if(tempDir->file_size>=1024) return -3;

```

```

/* 从磁盘申请空白块 */
int newblock = get_block();
if(!newblock) return -2;

/* 修改并更新文件目录项 */
tempDir->file_size += 32;
write_to_Disc(tempDir_num, temp_block.buffer);

/* 创建新目录并更新目录块 */
File* newDir = &filelist[file_num];
newDir->file_type = DIR_TYPE;
newDir->first_block = newblock;
newDir->file_size = 64;
strcpy(newDir->name, filename.c_str());
write_to_Disc(tempDir->first_block, filelist_block.buffer);

/* 修正当前目录的内容, 添加 .与..目录 */
Block temp;
read_from_Disc(newblock, temp.buffer);
File* pdir = (File*)temp.buffer; /* 使用文件指针 pdir 进行文件操作 */

strcpy(pdir[0].name, ".");
pdir[0].file_size = file_num;
pdir[0].file_type = DIR_TYPE;
pdir[0].first_block = tempDir->first_block;

strcpy(pdir[1].name, "..");
pdir[1].file_size = filelist[0].file_size;
pdir[1].file_type = DIR_TYPE;
pdir[1].first_block = filelist[0].first_block;
write_to_Disc(newblock, temp.buffer);

return file_num;
}

/* 目录移除函数, 使用之前需要设置 tempDir
* filename: 目录名
* 返回值 int: 返回 0, 删除成功; 返回-1, 目录不存在
*/
int File_system::remove_dir(string &filename)
{
/* 使用 create_dir 函数查找当前目录是否存在 */

```

```

    int file_num = create_dir(filename, SEARCH);
    if(file_num < 0) return -1;

    /* 加载对应的文件目录块 */
    Block block; File* temp;
    int temp_num = tempDir_num;
    int temp_index = tempDir_index;
    set_File_Pointer(&temp, tempDir->first_block, file_num, block);

    /* 递归删除当前目录下的所有文件 */
    delete_dir_dfs(temp->first_block, temp->file_size);

    /* 加载对应的文件目录块 */
    tempDir_num=temp_num;
    tempDir_index=temp_index;
    set_File_Pointer(&tempDir,tempDir_num,tempDir_index,temp_block);

    /* 修改文件目录块 */
    tempDir->file_size-=32;
    write_to_Disc(tempDir_num, temp_block.buffer);

    read_from_Disc(tempDir->first_block, block.buffer);
    File* curr_filelist=(File*)block.buffer;
    while (file_num<tempDir->file_size/32)
    {
        curr_filelist[file_num]=curr_filelist[file_num+1];
        file_num++;
    }
    write_to_Disc(tempDir->first_block, block.buffer);
    return 0;
}

/* 目录递归删除函数，删除该目录下的所有子文件
 * block: 目录名
 * file_size: 目录文件的大小
 * 返回值 int: 返回 0，目录删除成功；返回-1，目录删除失败
 */
int File_system::delete_dir_dfs(int block, int file_size)
{
    /* filelist 指向文件目录项 */
    Block File_block;
    read_from_Disc(block, File_block.buffer);
    File* filelist = (File*)File_block.buffer;

```

```

/* 循环查询每个文件目录项 */
int max_file_num = file_size/32;
for(int i=2;i<max_file_num;i++)
{
    /* 如果当前为文件，则释放所有空白块 */
    if(filelist[i].file_type==FILE_TYPE)
        release_block(filelist[i].first_block, FILE_TYPE);
    /* 如果当前为目录，则递归删除 */
    if(filelist[i].file_type==DIR_TYPE)
        delete_dir_dfs(filelist[i].first_block,
                        filelist[i].file_size);
}

/* 释放当前目录块 */
*((int*)File_block.buffer)=bk_fhead;
bk_fhead=block;
bk_nfree++;
write_to_Disc(block, File_block.buffer);
return 0;
}

/* 显示目录下文件函数
 * 显示当前目录下的所有文件
 */
int File_system::list_files()
{
    /* 加载当前目录 */
    set_File_Pointer(&tDir, currentDir_num, currentDir_index,
                    temp_block);

    Block tempblock;
    read_from_Disc(currentDir->first_block, tempblock.buffer);

    cout << usr_name << "/" << dir_path << ":" << endl;
    for(int i=2;i<currentDir->file_size/32;i++)
    {
        if(((File*)tempblock.buffer)[i].file_type==FILE_TYPE)
            cout << "FILE ";
        else cout << "DIR ";
        cout << ((File*)tempblock.buffer)[i].name << " ";
        cout << ((File*)tempblock.buffer)[i].file_size;
        cout << " B " << endl;
    }
    return 0;
}

```



```

}

/* 切换当前目录函数
 * filepath: 切换到 filepath 指向的目录，支持绝对路径以及相对路径
 * 返回值 int: 返回 0 成功，返回 -1 失败
 */
int File_system::cd_dir(string &filepath)
{
    string path = dir_path;
    /* 从磁盘中加载当前文件目录块 */
    tempDir_num=currentDir_num;
    tempDir_index=currentDir_index;
    set_File_Pointer(&tempDir,tempDir_num,tempDir_index,temp_block);

    while (true) {
        /* 如果开头为/ */
        if(filepath[0]=='/')
        {
            /* 设置临时目录为根目录 */
            tempDir_num=0;
            tempDir_index=use_num;
            set_File_Pointer(&tempDir, tempDir_num,
                            tempDir_index, temp_block);
            filepath = filepath.substr(1);
            if(filepath.length()==0)
            {
                currentDir_num=0;
                currentDir_index=use_num;
                set_File_Pointer(&tempDir, currentDir_num,
                                currentDir_index, temp_block);
                dir_path = "";
                return 0;
            }
        }
        else if(filepath[0]=='.'&&filepath[1]=='/') /* 如果开头为./ */
            filepath = filepath.substr(2);
        else if(filepath[0]=='.'&&filepath[1]!='.') /* 如果开头为.当级
        目录 */
        {
            filepath = filepath.substr(1);
            continue;
        }

        /* 如果无下级目录 */
    }
}

```

```

if(filepath.find('/')==filepath.npos)
{
    /* 使用 create_dir 函数查找当前目录是否存在 */
    int file_num = create_dir(filepath, SEARCH);
    if(file_num < 0) return -1;

    /* 设置为当前目录 */
    currentDir_num=tempDir->first_block;
    currentDir_index=file_num;
    set_File_Pointer(currentDir, currentDir_num,
                     currentDir_index, temp_block);

    /* 如果当前..上级目录 */
    string tempstr = currentDir->name;
    if(tempstr=="..")
    {
        /* 如果需要切换的为根目录 */
        if(!currentDir->first_block)
        {
            currentDir_num=0;
            currentDir_index=use_num;
            set_File_Pointer(currentDir, currentDir_num,
                             currentDir_index, temp_block);
        }
        else
        {
            currentDir_index = currentDir->file_size;
            currentDir_num = currentDir->first_block;
            read_from_Disc(currentDir_num,temp_block.buffer);
            currentDir = &((File*)temp_block.buffer)
                        [currentDir_index];
        }

        /* 修正当前目录 */
        path=path.substr(0,path.length()-1);
        path=path.substr(0,path.find_last_of("/") +1);
    }
    else path += (tempstr + "/");

    dir_path = path;
    return 0;
}

/* 路径切分解析 */

```

```

        string spilt=filepath.substr(0, filepath.find('/'));
        filepath=filepath.substr(filepath.find('/')+1);
        tempDir_index=create_dir(spilt, SEARCH);
        if(tempDir_index < 0) return -1;

        /* 更新当前文件目录块 */
        write_to_Disc(tempDir_num, temp_block.buffer);
        /* 加载下一级文件目录块 */
        tempDir_num = tempDir->first_block;
        set_File_Pointer(&tempDir, tempDir_num,
                        tempDir_index, temp_block);

        /* 如果当前..上级目录 */
        string tempstr = tempDir->name;
        if(tempstr=="..")
        {
            /* 如果需要切换的为根目录 */
            if(!tempDir->first_block)
            {
                tempDir_num=0;
                tempDir_index=use_num;
                set_File_Pointer(&tempDir, tempDir_num,
                                tempDir_index, temp_block);
            }
            else
            {
                tempDir_index = tempDir->file_size;
                tempDir_num = tempDir->first_block;
                read_from_Disc(tempDir_num, temp_block.buffer);
                tempDir = &((File*)temp_block.buffer)[tempDir_index];
            }
            path=path.substr(0,path.length()-1);
            path=path.substr(0,path.find_last_of("/")+1);
        }
        else path += (tempstr + "/");
    }
}

/* 用户操作 */

/* 用户注册函数 */
int File_system::Register(string &user, string &pass)
{

```

```

int usr_num = 0;
char buf[BLOCK_SIZE];

/* 读取引导块信息 */
read_from_Disc(0, buf);
Usr* currUsr = (Usr*)&buf[8];
bk_fhead = *((int*)buf);
bk_nfree = ((int*)buf)[1];

/* 查找当前是否存在该用户 */
for(;usr_num<MAX_USR_NUM;usr_num++)
{
    if(!currUsr[usr_num].vaild) break;
    if(!strcmp(currUsr[usr_num].username, user.c_str()))
    {
        cout << "当前用户已存在! " << endl;
        return USR_EXIST;
    }
}
if(usr_num==MAX_USR_NUM) return USR_FULL;

/* 创建用户 */
int root = get_block();
if(!root) return USR_CREATE_FAIL;
update_zero_block();

read_from_Disc(0, buf);
currUsr = (Usr*)&buf[8];

currUsr[usr_num].root_block = root;
currUsr[usr_num].root_size = 64;
currUsr[usr_num].vaild=1;
strcpy(currUsr[usr_num].username,user.c_str());
strcpy(currUsr[usr_num].passname,pass.c_str());
write_to_Disc(0, buf);

/* 添加 .与..目录 */
read_from_Disc(currUsr[usr_num].root_block, buf);
File* pdir = (File*)buf; /* 使用文件指针 pdir 进行文件操作 */
strcpy(pdir[0].name, ".");
pdir[0].file_size = usr_num;
pdir[0].file_type = DIR_TYPE;
pdir[0].first_block = 0;

```

```

        strcpy(pdir[1].name, "..");
        pdir[1].file_size = usr_num;
        pdir[1].file_type = DIR_TYPE;
        pdir[1].first_block = 0;

        /* 将修改结果写回 */
        write_to_Disc(root, buf);
        cout << "用户创建成功!" << endl;
        return USR_CREATE_SUCCESS;
    }

    /* 用户移除函数 */
    int File_system::Remove(string &user, string &pass)
    {
        int num = log_in(user, pass);
        if(num != USR_EXIST)
        {
            cout << "移除失败!" << endl;
            return -1;
        }
        delete_dir_dfs(rootDir->first_block, rootDir->file_size);
        log_out();

        char buf[BLOCK_SIZE];
        /* 读取引导块信息 */
        read_from_Disc(0, buf);
        Usr* currUsr = (Usr*)&buf[8];
        int i=use_num;
        for(;i<MAX_USR_NUM;i++)
        {
            if(!currUsr[i+1].vaild) break;
            currUsr[i] = currUsr[i+1];
        }
        currUsr[i].vaild = 0;
        write_to_Disc(0, buf);
        return 0;
    }

    /* 用户登录函数 */
    int File_system::log_in(string &user, string &pass)
    {
        int usr_num = 0;
        char buf[BLOCK_SIZE];

```

```

/* 读取引导块信息 */
read_from_Disc(0, buf);
Usr* currUsr = (Usr*)&buf[8];
/* 查找当前是否存在该用户 */
for(;usr_num<MAX_USR_NUM;usr_num++)
{
    if(!currUsr[usr_num].vaild) return USR_NOT_EXIST;
    if(!strcmp(currUsr[usr_num].username, user.c_str()))
    {
        cout << "用户名正确！" << endl;
        if(!strcmp(currUsr[usr_num].passname, pass.c_str()))
        {
            cout << "密码正确！" << endl;
            use_num = usr_num;
            /* 设置当前的根目录 读取引导块 */
            read_from_Disc(0, zero_block.buffer);
            currUsr = (Usr*)&zero_block.buffer[8];
            rootDir = (File*)&currUsr[usr_num];

            /* 设置当前目录以及临时目录的目录块 */
            currentDir=rootDir;
            currentDir_num=0;
            currentDir_index=usr_num;
            tempDir=rootDir;
            tempDir_num=0;
            tempDir_index=usr_num;

            /* 加载空白块处理逻辑 */
            read_from_Disc(0, buf);
            bk_fhead = *((int*)buf); /* 空白块头指针 */
            bk_nfree = ((int*)buf)[1]; /* 空白块数 */
            usr_name = currUsr[usr_num].username;
            return USR_EXIST;
        }
    }
}

return USR_NOT_EXIST;
}

/* 用户登出函数 */
int File_system::log_out()
{
    /* 更新引导块 */
    update_zero_block();
}

```

```

    usr_name = "";
    dir_path = "";
    return 0;
}

/* 用户显示函数 */
int File_system::show_usr()
{
    int usr_num = 0;
    char buf[BLOCK_SIZE];
    /* 读取引导块信息 */
    read_from_Disc(0, buf);

    cout << "free block num: "<< ((int*)buf)[1] << endl << endl;
    Usr* currUsr = (Usr*)&buf[8];
    /* 查找当前是否存在该用户 */
    for(;usr_num<MAX_USR_NUM;usr_num++)
    {
        if(!currUsr[usr_num].vaild) break;
        cout << "Name: "<< currUsr[usr_num].usrname << endl;
        cout << "root: "<< currUsr[usr_num].root_block << endl;
    }
    return 0;
}

/* 拓展操作 */

void File_system::touch(string &filepath)
{
    File_control_block* file = open_file(filepath, CREATE);
    if(file->block_num==-1) cout << "文件名与目录名冲突!" << endl;
    else if(file->block_num==-2) cout << "空白块耗尽!" << endl;
    else if(file->block_num==-3) cout << "目录已满!" << endl;
    else if(file->block_num==-4) cout << "路径目录创建失败!" << endl;
}

void File_system::rm(string &filepath)
{
    File_control_block* file = open_file(filepath, READ_WRITE);
    if(file->block_num<=0) cout << "当前文件不存在!" << endl;
    else {
        clear_file(file, 1);
    }
}

```



```

        Block block;
        /* 加载对应的文件目录块 */
        read_from_Disc(file->block_num, block.buffer);
        File* filelist = (File*)block.buffer;

        /* 修改文件目录块 */
        tempDir_num=filelist[0].first_block;
        tempDir_index=filelist[0].file_size;

        set_File_Pointer(&tempDir, tempDir_num,
                        tempDir_index, temp_block);
        tempDir->file_size-=32;
        write_to_Disc(tempDir_num, temp_block.buffer);

        int file_num = file->file_index;
        while (file_num<tempDir->file_size/32)
        {
            filelist[file_num]=filelist[file_num+1];
            file_num++;
        }
        write_to_Disc(file->block_num, block.buffer);
        cout << "文件删除成功!" << endl;
    }
}

void File_system::cat(string &filepath)
{
    File_control_block* file = open_file(filepath, READ_WRITE);
    if(file->block_num<=0) cout << "当前文件不存在!" << endl;
    else {
        Block block;
        read_from_Disc(file->block_num, block.buffer);
        File* pfile = &((File*)block.buffer)[file->file_index];

        int size = pfile->file_size;
        char *buf = new char[size];
        read_from_file(file, size, buf, 0);
        for(int i=0;i<size;i++)
            cout << buf[i];
        cout << endl;
        delete[] buf;
    }
}

```

```

void File_system::gedit(string &filepath)
{
    File_control_block* file = open_file(filepath, READ_WRITE);
    if(file->block_num<=0) cout << "当前文件不存在!" << endl;
    else {
        /* 显示原有文本内容 */
        cat(filepath);

        /* 读入新的文本 */
        int size = 0; char c;
        char *buf = nullptr;
        cout << "请输入需要输入的文本(\\n 结尾):" << endl;
        while((c=getchar())!='\\n')
        {
            if(!(size % BLOCK_SIZE))
            {
                if(!buf) buf = new char[BLOCK_SIZE];
                else {
                    int buf_num = size/BLOCK_SIZE;
                    char *tempbuf = new char[buf_num*BLOCK_SIZE];
                    for(int i=0;i<size;i++)
                        tempbuf[i] = buf[i];
                    delete [] buf;
                    buf = tempbuf;
                }
            }
            buf[size++] = c;
        }
        clear_file(file, 2);
        write_to_file(file, size, buf, 0);
    }
}

void File_system::mkdir(string &filepath)
{
    /* 从磁盘中加载当前文件目录块 */
    tempDir_num=currentDir_num;
    tempDir_index=currentDir_index;
    set_File_Pointer(&tempDir,tempDir_num,tempDir_index,temp_block);

    while (true) {
        /* 如果开头为/ */
        if(filepath[0]=='/')
        {

```

```

        /* 设置临时目录为根目录 */
        tempDir_num=0;
        tempDir_index=use_num;
        set_File_Pointer(&tempDir, tempDir_num,
                        tempDir_index, temp_block);
        filepath = filepath.substr(1);
    }
    else if(filepath[0]=='.'&&filepath[1]=='/') /* 如果开头为./ */
        filepath = filepath.substr(2);

    /* 如果无下级目录 */
    if(filepath.find('/')==filepath.npos)
    {
        int file_num = create_dir(filepath, CREATE);
        if(file_num==-1) cout << "目录名与文件名冲突!" << endl;
        else if(file_num==-2) cout << "空白块耗尽!" << endl;
        else if(file_num==-3) cout << "目录已满!" << endl;
        return;
    }

    /* 路径切分解析 */
    string spilt=filepath.substr(0, filepath.find('/'));
    filepath=filepath.substr(filepath.find('/')+1);
    tempDir_index=create_dir(spilt, CREATE);
    if(tempDir_index < 0){
        cout << "路径目录创建失败!" << endl;
        return;
    }
    /* 更新当前文件目录块 */
    write_to_Disc(tempDir_num, temp_block.buffer);
    /* 加载下一级文件目录块 */
    tempDir_num = tempDir->first_block;
    set_File_Pointer(&tempDir,tempDir_num,tempDir_index,temp_block);
}

}

void File_system::rmdir(string &filepath)
{
    /* 从磁盘中加载当前文件目录块 */
    tempDir_num=currentDir_num;
    tempDir_index=currentDir_index;
    set_File_Pointer(&tempDir,tempDir_num,tempDir_index,temp_block);
}

```

```

while (true) {
    /* 如果开头为/ */
    if(filepath[0]=='/')
    {
        /* 设置临时目录为根目录 */
        tempDir_num=0;
        tempDir_index=use_num;
        set_File_Pointer(&tempDir, tempDir_num,
                        tempDir_index, temp_block);
        filepath = filepath.substr(1);
    }
    else if(filepath[0]=='.'&&filepath[1]=='/') /* 如果开头为./ */
        filepath = filepath.substr(2);

    /* 如果无下级目录 */
    if(filepath.find('/')==filepath.npos)
    {
        int status = remove_dir(filepath);
        if(status < 0) cout << "目录删除失败!" << endl;
        else cout << "目录删除成功!" << endl;
        return;
    }

    /* 路径切分解析 */
    string spilt=filepath.substr(0, filepath.find('/'));
    filepath=filepath.substr(filepath.find('/')+1);
    tempDir_index=create_dir(spilt, READ_WRITE);
    if(tempDir_index < 0){
        cout << "路径目录不存在!" << endl;
        return;
    }
    /* 更新当前文件目录块 */
    write_to_Disc(tempDir_num, temp_block.buffer);
    /* 加载下一级文件目录块 */
    tempDir_num = tempDir->first_block;
    set_File_Pointer(&tempDir, tempDir_num,
                    tempDir_index, temp_block);
}
}

```

```

void File_system::show_help()
{
    cout << "          My File System Help Message          " << endl;
    cout << "  _____ " << endl;
    cout << " 1.touch <filename>          -- 新建文件 " << endl;
    cout << " 2.rm <filename>             -- 删除文件 " << endl;
    cout << " 3.cat <filename>            -- 查看文件  " << endl;
    cout << " 4.gedit <filename>          -- 编辑文件  " << endl;
    cout << " 5.mkdir <dirname>           -- 新建目录  " << endl;
    cout << " 6.rmdir <filename>          -- 移除目录  " << endl;
    cout << " 7.ls                        -- 显示目录文件 " << endl;
    cout << " 8.cd <dirname>              -- 进入目录  " << endl;
    cout << " 9.clear                     -- 清除屏幕内容 " << endl;
    cout << " 10.help                     -- 显示帮助  " << endl;
    cout << " 11.exit                     -- 注销用户  " << endl;
    cout << "  _____ " << endl;
}

bool all_blank(string &str)
{
    for(int i=0;i<str.length();i++)
        if(str[i]!=' ') return false;
    return true;
}

int main()
{
    system("clear");
    File_system *file_system = new File_system();

    while (true){
        string cmd, op, sname, dname;
        getline(cin, cmd, '\n');
        system("clear");
        getline(cin, cmd, '\n');
        /* 如果只输入空格 */
        if(all_blank(cmd)) continue;
        /* 删除两端空格 */
        cmd = cmd.substr(cmd.find_first_not_of(' '),
            cmd.find_last_not_of(' ')-cmd.find_first_not_of(' ')+1);
        if(cmd.length() == 0) continue;
    }
}

```

```

/* 切分指令 */
if(cmd.find(" ") == cmd.npos) op = cmd;
else{
    op = cmd.substr(0, cmd.find(" "));
    dname = cmd.substr(cmd.rfind(" ") + 1);

    /* 判断是否为双输入指令 */
    sname = cmd.substr(cmd.find(" "),
                       cmd.rfind(" ") - cmd.find(" ") + 1);
    if(!all_blank(sname)){
        sname = sname.substr(sname.find_first_not_of(' '),
                             sname.find_last_not_of(' ') -
                             sname.find_first_not_of(' ') + 1);
    }
}

if(op == "login"){
    if(file_system->log_in(sname, dname) != USR_EXIST){
        cout << "登录失败! " << endl;
        continue;
    }
}
else if(op == "register"){
    file_system->Register(sname, dname);
    continue;
}
else if(op == "remove"){
    file_system->Remove(sname, dname);
    continue;
}
else if(op == "show"){ file_system->show_usr(); continue;}
else if(op == "exit"){ break;}
else continue;
file_system->show_help();
file_system->dir_path = "";

while (true)
{
    setbuf(stdin, nullptr);
    bool is_cp = false;
    cout << file_system->usr_name << "@ubuntu:"
         << file_system->dir_path << "$";

    getline(cin, cmd, '\n');

```

```

/* 如果只输入空格 */
if(all_blank(cmd)) continue;
/* 删除两端空格 */
cmd = cmd.substr(cmd.find_first_not_of(' '),
                  cmd.find_last_not_of(' ') -
                  cmd.find_first_not_of(' ') + 1);
if(cmd.length() == 0) continue;

/* 切分指令 */
if(cmd.find(" ") == cmd.npos) op = cmd;
else{
    op = cmd.substr(0, cmd.find(" "));
    dname = cmd.substr(cmd.rfind(" ") + 1);

    /* 判断是否为 cp 指令 */
    sname = cmd.substr(cmd.find(" "), cmd.rfind(" ") -
                      cmd.find(" ") + 1);
    if(!all_blank(sname)){
        is_cp = true;
        sname = sname.substr(sname.find_first_not_of(' '),
                             sname.find_last_not_of(' ') -
                             sname.find_first_not_of(' ') + 1);
    }
}

if(op == "touch") file_system->touch(dname);
if(op == "rm") file_system->rm(dname);
if(op == "cat") file_system->cat(dname);
if(op == "gedit") file_system->gedit(dname);
if(op == "mkdir") file_system->mkdir(dname);
if(op == "rmdir") file_system->rmdir(dname);
if(op == "ls") file_system->list_files();
if(op == "cd")
    if(file_system->cd_dir(dname) < 0)
        cout << "当前目录不存在!" << endl;
if(op == "clear") { system("clear");
                    file_system->show_help(); }
if(op == "help") file_system->show_help();
if(op == "exit") { file_system->log_out(); break; }
file_system->update_zero_block();
}
}

delete file_system;
return 0;
}

```