

**课 程 设 计**

**课程设计名称：** 操作系统课程设计

**专 业 班 级 ：** 物联网1902班

**学 生 姓 名 ：** 王源

**学 号 ：** 201916070216

**指 导 教 师 ：** 廖庆荣

**课程设计时间：** 2021年6月21日

**操作系统 课程设计任务书**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **学生姓名** | **王源** | **专业班级** | | **物联网1902** | **学院名称** | | **信息科学与工程学院** |
| **题 目** | 操作系统整合性shell用户接口实践 | | | | | | |
| **课题性质** | 操作系统整合实践 | | **课题来源** | | | 教师指定 | |
| **指导教师** | 廖庆荣 | | **同组姓名** | | | **赵浩钧，王众** | |
| **主要内容** | 要能完全理解操作系统，最好的方法就是实践出一个操作系统。因此在这一次的课程设计，主要就是实践操作系统的一些命令。其中shell在UNIX或Linux操作系统中是一种很重要的使用者接口，可以透过命令行的方式输入指令。这个项目希望可以实现shell的操作方式，使用者可以在命令行的后方（>）输入指令，也可以加上（&）符号在背景执行，表示父进程（parent process）和子进程（child process）并行（concurrently）执行。例如编程结果可以执行如下：  cjliaoOSshell>cat myshell.c  或是背景执行  cjliaoOSshell >cat myshell.c &  例如下列五个指令也能在cjliaoOSshell >底下运行：  exit：结束程序执行  history：显示历史纪录  ls：显示目录  mkdir：创建目录  rmdir：删除目录  使用shell功能部份的系统截图如图1所示。就像我们之前在实验课程所进行的将文文件显示列出实践一样，必须从操作系统的数据结构建构好之后，编程完成，如此才能真正理解操作系统是如何运作的。  history功能可以将所有的执行记录下来，保存进程历史纪录。这跟现行所有操作系统保存系统的log档案是一样的道理。  图1：使用shell功能部份的系统截图 | | | | | | |
| **任务要求** | 当完成这个项目之后，运行程序可以显示提示字符cjliaoOSshell >，接着可以下达命令，例如下列所示history、!!、!N（N为数字，代表第几个曾经执行过的指令）等。  1. history：显示最近的十笔历史纪录，如果历史纪录超过十笔，则会显示最近的十笔，但是号码仍会显示出来。例如有三十笔纪录，会显示第21-30笔。如果没有历史纪录，则会显示没有历史纪录的讯息。  2. !!：執行最近運行過的指令，如果没有历史執行纪录，则会显示没有指令可以執行的讯息。  3. !3：執行第三笔历史纪录，如果历史纪录没有第三笔，则会显示错误讯息没有这一笔历史纪录可供再執行一次。  4. 程序中使用fork()可以来启动进程。  5. 这些shell指令主要是在UNIX或Linux 操作系统上运行，如果你使用 Windows的环境运行，除非Windows操作系统完整支持，否则可能会发生程序出错或是shell指令无法运行。  以下这15项请各位同学填上指令功能说明请直接输入在程序中的批注说明，其中使用file功能的系统截图如图2所示。  (1). ps：观察系统所有的进程数据。  (2). ps axjf：连同部分进程树状态显示。  a：不与terminal有关的所有process。  u：有效使用者（effective user）相关的process。  x：通常与a这个参数一起使用，可列出较完整信息。  l：较长、较详细的将该PID的的信息列出。  j：工作的格式（jobs format）。  f：做一个更为完整的输出。  (3). ps aux：ps aux会依照PID的顺序来排序显示。  (4). ls -l： 将文件信息详细输出  (5). ls -l|more： 在ls -l基础上添加more的功能  (6). top： 监控linux的系统情况  (7). cal： 查看公历日历  (8). Whoami： 查看当前用户名  (9). date： 实现系统的日期与时间  (10). pwd： 显示整个路径名  (11). mv： 给文件重命名或移动文件  (12). cp：复制文档  (13). file： 用于辨识文件类型  (14). cat： 连接文件并打印到标准输出设备上  (15). rm：删除文档    图2：使用file功能的系统截图 | | | | | | |
| **参考文献** | 1. 汤小丹、梁红兵、哲凤屏、汤子瀛（2014）。计算机操作系统 第四版，西安电子科技大学出版社。 2. 梁红兵、汤小丹、汤子瀛（2014）。计算机操作系统 第四版 学习指导与题解，西安电子科技大学出版社。 3. Abraham Silberschatz, Greg Gagne, Peter B. Galvin (2013). Operating system concepts. John Wiley & Sons. | | | | | | |
| **审查意见** | **指导教师签字：**  **教研室主任签字： 年 月 日** | | | | | | |

说明：本表由指导教师填写，由教研室主任审核后下达给选题学生，装订在设计（论文）首页

**操作系统整合性shell用户接口和实践**

# 摘要：

要能完全理解操作系统，最好的方法就是实践出一个操作系统。因此在这一次的课程设计，主要就是实践操作系统的一些命令。其中shell在UNIX或Linux操作系统中是一种很重要的使用者接口，可以透过命令行的方式输入指令。这个项目希望可以实现shell的操作方式，并且可以将所有的执行记录下来，保存程历史纪录，这跟现行所有操作系统保存系统的log档案是一样的道理。在这个实践过程，能熟悉Linux的指令操作，同时也能以指令操作了解进程的产生、进程的管理，以及系统的调用等功能，充分了解操作系统的指令运行。

目录

[摘要： 6](#_Toc76142342)

[一、设计目的 6](#_Toc76142343)

[二、设计要求与内容 6](#_Toc76142344)

[三、设计原理 9](#_Toc76142345)

[ls&&ls -l&&ls -l|more 18](#_Toc76142346)

[四、流程圖 52](#_Toc76142347)

[五、 测试结果與說明 64](#_Toc76142348)

[收获、体会和建议 77](#_Toc76142349)

[附录 78](#_Toc76142350)

[1. 源碼 78](#_Toc76142351)

[主要参考文献 78](#_Toc76142352)

# 一、设计目的

1. 能熟悉操作系统的实践
2. 能孰悉Linux的shell指令
3. 能实践整合性的操作系统shell用户接口指令设计
4. 能实现操作系统的历史纪录
5. 能了解和实现进程如何产生以及运作
6. 能实现进程产生
7. 能实现操作系统功能调用

# 二、设计要求与内容

shell在UNIX或Linux操作系统中是一种很重要的使用者接口，可以透过命令行的方式输入指令。这个项目希望可以实现shell的操作方式，使用者可以在命令行的后方（>）输入指令，也可以加上（&）符号在背景执行，表示父进程（parent process）和子进程（child process）并行（concurrently）执行。例如编程结果可以执行如下：

cjliaoOSshell>cat myshell.c

或是背景执行

cjliaoOSshell>cat myshell.c &

　　例如下列五个指令也能在cjliaoOSshell>底下运行：

exit：结束程序执行

history：显示历史纪录

ls：显示目录

mkdir：创建目录

rmdir：删除目录

并且可以将所有的执行记录下来，保存程历史纪录。这跟现行所有操作系统保存系统的log档案是一样的道理。

当完成这个项目之后，运行程序可以显示提示字符cjliaoOSshell>，接着可以下达命令，例如下列所示history、!!、!N（N为数字，代表第几个曾经执行过的指令）等。

1. history：显示最近的十笔历史纪录，如果历史纪录超过十笔，则会显示最近的十笔，但是号码仍会显示出来。例如有三十笔纪录，会显示第21-30笔。如果没有历史纪录，则会显示没有历史纪录的讯息。

2. !!：執行最近運行過的指令，如果没有历史執行纪录，则会显示没有指令可以執行的讯息。

3. !3：執行第三笔历史纪录，如果历史纪录没有第三笔，则会显示错误讯息没有这一笔历史纪录可供再執行一次。

4. 程序中使用fork()可以来启动进程。

5. 这些shell指令主要是在UNIX或Linux 操作系统上运行，如果你使用 Windows的环境运行，除非Windows操作系统完整支持，否则可能会发生程序出错或是shell指令无法运行。

以下这15项请各位同学填上指令功能说明请直接输入在程序中的批注说明，这15个指令必须编程实现出来。

(1). ps： **观察系统所有的进程数据。**

(2). ps axjf：  **连同部分进程树状态显示。**

(3). ps aux：  **ps aux会依照PID的顺序来排序显示。**

(4). ls -l： **将文件信息详细列出来**

(5). ls -l|more：**在ls -l的基础上加上more的功能，自己实现时一条一条按回车输出**

(6). top： **监控linux的系统情况**

(7). cal： **查看公历日历**

(8). whoami： **查看当前用户名**

(9). date： **实现系统的日期与时间**

(10). pwd： **显示当前整个路径名**

(11). mv： **给文件重命名或移动文件**

(12). cp：  **复制文档**

(13). file：  **用于辨识文件类型**

(14). cat： **连接文件并打印到标准输出设备上**

(15). rm：  **删除文档**

# 三、设计原理

**一．exit：结束程序执行**

**1.主要代码实现：**

//退出shell

void quitSHELL()

{

    printf("I`m Chinese! 再见\n");

    exit(1);

}

**2.具体分析：**

在shell.cpp中实现一个函数接口quitSHELL()，当shell.cpp调用该函数时，打印一行“I’m Chinese!再见”,内部再调用exit()函数退出程序。原理很简单，这里无须赘述。

**二．history：显示历史纪录**

**1.主要代码实现：**

//查看history

void historySHELL()

{

    if(\_history.second.size() == 0)

    {

        printf("不存在历史记录\n");

        return;

    }

    auto iter = \_history.second.begin();

    for(int i = 1 + \_history.first; iter != \_history.second.end(); ++iter, ++i)

    {

        printf("%4d %s\n", i, \*iter);

    }

}

//添加history

void historyADD(const char\* pbuf, const int buf\_len)

{

    //new一个char\*出来方便储存

    char\* temp\_buf = new char[buf\_len];

    strcpy(temp\_buf, pbuf);

    temp\_buf[buf\_len-1] = 0;

    //如果已储存消息条数小于10则直接添加

    if(\_history.second.size() < 10)

    { //入表

        \_history.second.push\_back(temp\_buf);

    }

    //否则前端+1 弹表头,入表尾

    else

    {

        //起始序号++

        ++\_history.first;

        //delete掉头节点

        char\* t = \_history.second.front();

        \_history.second.pop\_front();

        delete[] t;

        //入表

    \_history.second.push\_back(temp\_buf);

    }

}

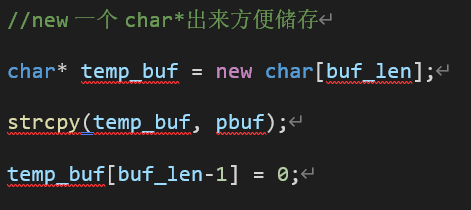
**2．具体分析：**

通过实现一个historySHELL()的函数接口，判断\_history\_second.size()是否为0，若为0，则表明在此之前并没有输出执行过任何命令，那么我们打印“不存在任何历史记录”，否则，也就是\_history\_second.size()不等于0，那么我们进一步判断。通过algorithm内置的.begin()接口得到一个指向开始的迭代器，进而利用for循环从头遍历到尾部，也就是从\_history\_second.begin()到\_history\_second.end(),依次输出在此之前执行过的指令信息。

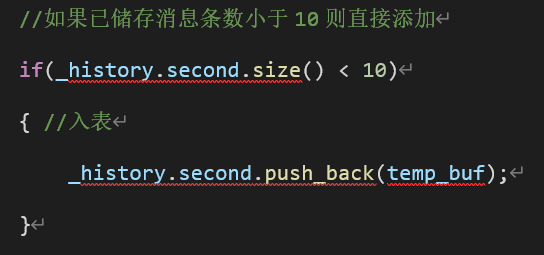
当然，我们也严格根据老师所给的要求实现了如下功能：

创建一个historyADD()函数来添加任何一个执行过的执行信息并存储起来。

我们动态开辟一块空间来存储history的信息。



判断一下已经存储过的信息条数是否小于10，如果小于10条信息，那我们可以直接利用尾插法进行信息的插入。



如果消息条数大于10，让前端表头+1，也就是起始序号++

**，**把原来的头节点删除，创建一个新的头节点，然后再进行入表操作。



**三．!!：执行最近进行过的指令，如果没有历史执行纪录，则会显示没有指令可以执行的讯息。**

**1.主要代码实现：**

//!!

void rollbackSHELL()

{

    char buf[1024];

    //获取最后一条记录

    strcpy(buf, \_history.second.back());

    //执行命令

    runCMD((const char\*)buf);

}

**2.具体分析：**

实现一个rollbackSHELL()函数，首先创建一个buf[]数组来存储信息，利用stdlib.h头文件内置的拷贝函数strcpy()将\_history.second.back()最后一条信息记录拷贝到buf[1024]数组中，随后调用runCMD()执行相关操作。

**runCMD的实现代码如下：**

//执行指定命令

void runCMD(const char\* cmd)

{

    //命令长度

    int buf\_len = strlen(cmd);

    //储存命令

    char buf[1024];

    strcpy(buf, cmd);

    buf[buf\_len] = '\n';

    //储存切割好的命令

    char\* MyArgv[10] = {0};

    //将读取到的字符串分成多个字符串

    int argc = cutCMD(buf, MyArgv);

    //如果是shell自带的命令的话直接执行

    int temp = judgeCMD(MyArgv);

    if(temp > 0)

    {

        switch(temp)

        {

            //退出shell

            case 1: quitSHELL(); break;

            //查看history

            case 2: historySHELL(); break;

            //!相关无法被执行

            case 3:

            case 4: printf("!相关命令无法被重复执行\n"); break;

        }

    }

    //否则fork切换进程执行命令

    else

    {

        //若要求后台执行

        if(!strcmp("&", MyArgv[argc - 1]))

        {

            pthread\_t id;

            pthread\_create(&id, NULL, runADD, (void\*)buf);

            //线程等待

            pthread\_join(id, NULL);

            //printf("后台执行结束    %s\n", buf);

        }

        else

        {

            //利用fork新建进程执行命令

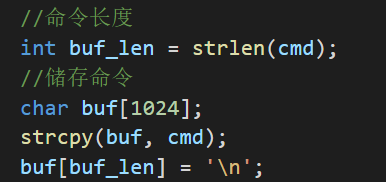
            forkCMD(MyArgv);

        }

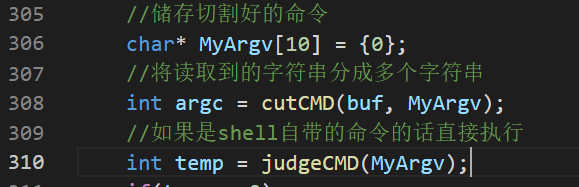
    }

}

runCMD()接受一个传来的指针，首先通过strlen()得到该数组的长度，其次再创建一个buf数组，将得到的信息拷贝到新建的buf数组中去，将buf数组的尾部赋值为’\n’，使其变成一个字符串。



储存好已经分配好的命令，将读取到的字符串分割成多个字符串，判断如果是shell自带的命令则直接执行，如果不是则进行后续操作。



如果得到的返回值大于0，则进入switch操作，当是switch为1时，直接调用quitSHELL函数退出程序，如果switch等于2，则调用historySHELL函数查看历史执行过的命令，返回值为表明执行了重复命令，此时我们给上一句提示：相关命令无法被重复执行！

当接受到的返回值小于等于0时，执行fork切换进程操作。

当!strcmp("&", MyArgv[argc - 1])为真时，执行&实现后台执行，为假时，利用fork新建进程执行命令。



**四．!N：执行第N笔历史纪录，如果历史纪录没有第N笔，则会显示错误讯息没有这一笔历史纪录可供再执行一次。**

**1.主要代码实现：**

//!N

void rollbackN(const char\* cmd)

{

    int t = strlen(cmd);

    //回溯指令id

    int num = 0;

    //获取id

    for(int i = 1; i < t; ++i)

    {

        if(cmd[i] >= '0' && cmd[i] <= '9')

        {

            num = num \* 10 + cmd[i] - '0';

        }

        else

        {

            printf("回溯指令行数输入有误\n");

            return;

        }

    }

    //所记录指令的id起始

    int start = \_history.first + 1;

    int end = start + \_history.second.size() - 1;

    //边缘判定

    if(\_history.second.size() == 0)

    {

        printf("历史记录为空 无法执行\n");

        return;

    }

    if(num < start || num > end)

    {

        printf("没有id为%d的历史记录 无法执行\n", num);

        return;

    }

    //找到这条指令

    auto iter = \_history.second.begin();

    for(int i = 0; iter != \_history.second.end() && i != num - start; ++i)

    {

        ++iter;

    }

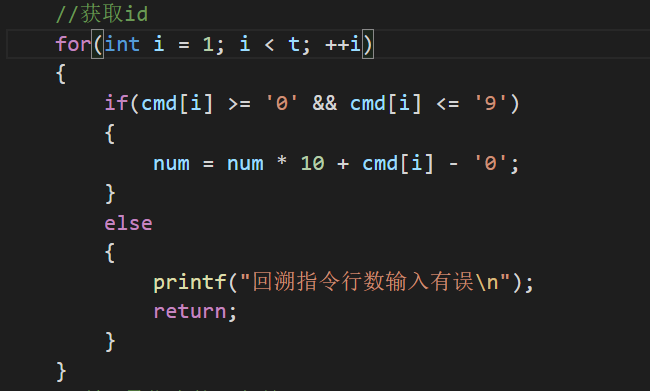
    //执行命令

    runCMD((const char\*)\*iter);

}

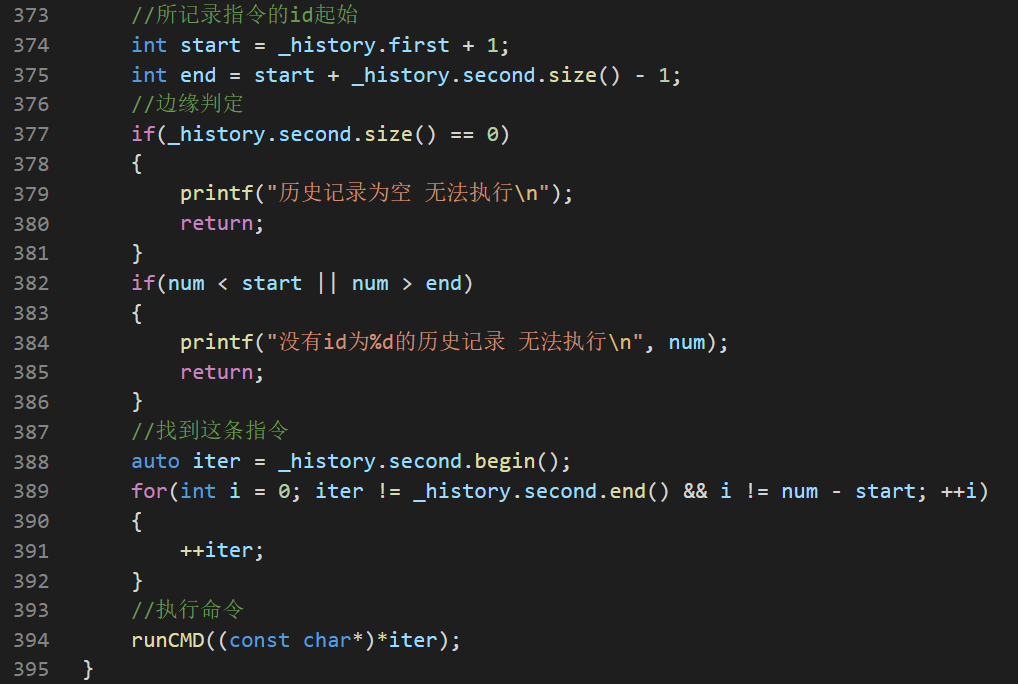
**2.具体分析：**

实现rollbackN函数来接受外部传过来的指针，用strlen函数来得到这个指针指向信息的长度。定义一个回溯变量num，进而获取id。



分别定义变量记录指令的id起点和终点，进行边缘的判定，当判断为0时，我们打印“历史纪录为空 无法执行”，结束掉程序。在不为空的情况下，当num<start或者num>end时，我们打印没有对应的历史记录。

否则便是通过遍历找到了这条指令，迭代器iter再传给runCMD()，执行该指定的命令。这里在！！中已经分析过，二者没有任何区别，故在此不再复述。



以上操作我都在shell.cpp里实现，故在此做一个总结：

通过对linux下shell实现原理的探索，我们决定实现：

1. 以shell系统为核心文件，通过核心内的fork/exec相关函数跳转到键入命令的自实现程序，从而实现shell的命令执行功能。

2. 自实现命令程序相关：通过whereis这个命令，我们可以得知linux下命令程序通常是存储在/usr/bin文件夹下。

故我们组的自实现命令程序被存放在shell文件夹内的./bin目录下。

3. 我们通过每一个独立自实现程序的main函数内的argc与argv参数，传入命令参数，从而根据参数执行不同的内容。

例如在ls命令程序内，通过argv参数得知执行的为ls-l还是ls或是ls -l | more。

4. 实验报告中要求实现history与!相关内容，由于我的shell为独立程序，与自实现命令程序相互独立，故我把history与!的相关内容实现在shell内。

由此shell在接收命令后，首先应该判断为内部命令或是外部命令。若为外部命令则跳转外部程序执行，若为内部命令(history/!!/!N)则直接执行。

5. history的数据结构我采用pair<int,list>，简单明了，int为程序history记录的起始id(因为要求只存10条消息)，list则为历史输入字符串链表。

当list内不足10条时，则直接在list内添加；若已经为10条，则int存储的id++，list链表pop出头信息，把新信息push进链表尾，即可实现history的储存。

当执行history命令时，直接遍历list即可。执行!!与!N时，通过list读取输入内容，随后执行该命令即可。

6. 这里用到了切割命令字符串通过判定空格来切割。

7. fork和exec原理啥的，这是linux下自带的函数，以便于我们去很好的模拟linux的原理操作。

8. &后台执行：如果检测到最后一个参数为&，便执行&后台并行操作。具体使用到create新线程来保证输入不会被阻塞，从而实现后台执行。

**五．Ls，ls -l 及ls -l|more: ls用于显示指定工作目录下之内容（列出目前工作目录所含之文件及子目录)。Ls -l是除了文件名称外，也将文件形态，权限，拥有者，文件大小等详细资讯列出。Ls -l|more用于将ls -l的内容显示出来后，可以按回车enter来显示下一页。**

ls&&ls -l&&ls -l|more

**1.主要代码实现：**

/\*

\* 简单实现ls命令

\* 1509寝W组

\* 2021/6/27

\*/

#include <stdio.h>

#include <dirent.h>

#include <string.h>

#include <sys/ioctl.h>

#include <unistd.h>

#include <sys/stat.h>   // 这个头文件用来得到文件的详细信息

#include <time.h>       // 时间头文件

#include <pwd.h>        // 用来得到用户名

#include <grp.h>        // 用来取得组名

// argv, argc

// 结构体,用来存储要输出的每个属性值

struct attribute

{

    char mode[10];       // 文件属性和权限

    int  links;          // 链接数

    char user\_name[20];  // 用户名

    char group\_name[20]; // 所在的用户组group

    long size;           // 文件大小

    char mtime[20];      // 最后修改的时间

    char filename[255];  // 文件名

    char extra[3];       // 用来显示时候要加 "\*"(可以执行的文件) 或者 "/" (目录) 的额外字符串

};

// 计算整数 n 有几位

int f(long n)

{

    int ret = 0;

    while(n)

    {

        n = n / 10;

        ++ ret;

    }

    return ret;

}

// 得到终端的列数和行数./

void getTerminatorSize(int \*cols, int \*lines)

{

#ifdef TIOCGSIZE

    struct ttysize ts;

    ioctl(STDIN\_FILENO, TIOCGSIZE, &ts);

    \*cols = ts.ts\_cols;

    \*lines = ts.ts\_lines;

#elif defined(TIOCGWINSZ)

    struct winsize ts;

    ioctl(STDIN\_FILENO, TIOCGWINSZ, &ts);

    \*cols = ts.ws\_col;

    \*lines = ts.ws\_row;

#endif /\* TIOCGSIZE \*/

}

// 由 int 型的 mode,得到实际要显示的字符串

void mode2str(int mode, char str[])

{

    strcpy(str, "----------\0");

    if(S\_ISDIR(mode)) str[0] = 'd';

    if(S\_ISCHR(mode)) str[0] = 'c';

    if(S\_ISBLK(mode)) str[0] = 'b';

    if(S\_ISLNK(mode)) str[0] = 'l';

    if(mode & S\_IRUSR) str[1] = 'r';

    if(mode & S\_IWUSR) str[2] = 'w';

    if(mode & S\_IXUSR) str[3] = 'x';

    if(mode & S\_IRGRP) str[4] = 'r';

    if(mode & S\_IWGRP) str[5] = 'w';

    if(mode & S\_IXGRP) str[6] = 'x';

    if(mode & S\_IROTH) str[7] = 'r';

    if(mode & S\_IWOTH) str[8] = 'w';

    if(mode & S\_IXOTH) str[9] = 'x';

}

// 根据用户的 id 值，得到用户名 user name

void uid2str(uid\_t uid, char \*user\_name) /\* 将uid转化成username \*/

{

    struct passwd \*pw\_ptr;

    pw\_ptr = getpwuid(uid);

    if( pw\_ptr == NULL)

    {

        sprintf(user\_name, "%d", uid);

    }

    else

    {

        strcpy(user\_name, pw\_ptr->pw\_name);

    }

}

// 根据用户组的 id 值，得到用户组名 group name

void gid2str(gid\_t gid, char \*group\_name) /\* 将uid转化成username \*/

{

    struct group \*grp\_ptr;

    grp\_ptr = getgrgid(gid);

    if( grp\_ptr == NULL)

    {

        sprintf(group\_name, "%d", gid);

    }

    else

    {

        strcpy(group\_name, grp\_ptr->gr\_name);

    }

}

// 时间的格式化字符串, 注意这里我把前面的星期和后面的年份都去掉了

void time2str(time\_t t, char \*time\_str)

{

    strcpy( time\_str, ctime(&t) + 4);

    time\_str[12] = '\0';

}

// 要显示的某一个文件详细信息,并把信息放在结构体 attribute 中

void ls\_long\_file(char \*dirname, char \*filename, struct attribute \*file\_attri)

{

    // 根据文件夹名和文件名得到全名

    char fullname[256];

    strcpy(fullname, dirname);

    strcpy(fullname + strlen(dirname), filename);

    struct stat mystat;

    if ( stat(fullname, &mystat) == -1)

    {

        printf("ls\_long\_file: stat error\n");

    }

    else

    {

        // 这里参考 <stat.h> 头文件

        int mode   = (int)  mystat.st\_mode;

        int links  = (int)  mystat.st\_nlink;

        int uid    = (int)  mystat.st\_uid;

        int gid    = (int)  mystat.st\_gid;

        long size  = (long) mystat.st\_size;

        long mtime = (long) mystat.st\_mtime;

        char str\_mode[10];          /\* 文件类型和许可权限, "drwxrwx---" \*/

        char str\_user\_name[20];

        char str\_group\_name[20];

        char str\_mtime[20];

        mode2str(mode, str\_mode);

        uid2str(uid, str\_user\_name);

        gid2str(gid, str\_group\_name);

        time2str(mtime, str\_mtime);

        char extra[3] = "\0\0";

        if (str\_mode[0] == 'd')

        {

            extra[0] = '/';

        }

        else if (str\_mode[0] == '-' && str\_mode[3] == 'x')

        {

            extra[0] = '\*';

        }

        // 存储在结构体中

        strcpy(file\_attri->mode, str\_mode);

        file\_attri->links = links;

        strcpy(file\_attri->user\_name, str\_user\_name);

        strcpy(file\_attri->group\_name, str\_group\_name);

        file\_attri->size = size;

        strcpy(file\_attri->mtime, str\_mtime);

        strcpy(file\_attri->filename, filename);

        strcpy(file\_attri->extra, extra);

    }

}

// struct

struct attribute file\_attribute[200]; // maximum 200

void lsLong(char \*dirname, int mode)

{

    DIR \*mydir = opendir( dirname );            /\* directory \*/

    char filename[20];

    int file\_num = 0;

    if (mydir == NULL)

    {

        // 显示单个文件的详细信息

        strcpy(filename, dirname);

        ls\_long\_file("./", filename, &file\_attribute[0]);

        ++ file\_num;

    }

    else

    {

        // 考虑用户输入文件夹没有输入反斜杠的情况

        int len = strlen(dirname);

        if (dirname[len - 1] != '/')

        {

            dirname[len] = '/';

            dirname[len+1] = '\0';

        }

        // 循环得到当前目录下的所有文件名,并存储在自定义的结构体中

        struct dirent \*mydirent;            /\* file \*/

        while ( (mydirent = readdir( mydir )) != NULL)

        {

            char filename[20];

            strcpy(filename, mydirent->d\_name);

            // 不能为隐藏文件

            if (!strcmp(filename, ".") || !strcmp(filename, ".") || filename[0] != '.')

            {

                ls\_long\_file(dirname, filename, &file\_attribute[file\_num ++]);

            }

        }

        closedir( mydir );

    }

    // 按照文件名排序

    struct attribute temp;

    char filename1[20];

    char filename2[20];

    int i, j;

    for (i = 0; i < file\_num; ++i)

    {

        for (j = i + 1; j < file\_num; ++ j)

        {

            strcpy(filename1, file\_attribute[i].filename);

            strcpy(filename2, file\_attribute[j].filename);

            if ( strcmp(filename1, filename2) > 0)

            {

                temp = file\_attribute[i];

                file\_attribute[i] = file\_attribute[j];

                file\_attribute[j] = temp;

            }

        }

    }

    // 格式化输出时,考虑每个属性值的范围

    int max\_mode = 0;

    int max\_links = 0;

    int max\_user\_name = 0;

    int max\_group\_name = 0;

    int max\_size = 0;

    int max\_mtime = 0;

    int max\_filename = 0;

    int max\_extra = 0;

    for (i = 0; i < file\_num; ++ i)

    {

        if ( max\_mode < strlen(file\_attribute[i].mode) )

        {

            max\_mode = strlen(file\_attribute[i].mode);

        }

        if (max\_links < f(file\_attribute[i].links))

        {

            max\_links = f(file\_attribute[i].links);

        }

        if ( max\_user\_name < strlen(file\_attribute[i].user\_name) )

        {

            max\_user\_name = strlen(file\_attribute[i].user\_name);

        }

        if ( max\_group\_name < strlen(file\_attribute[i].group\_name) )

        {

            max\_group\_name = strlen(file\_attribute[i].group\_name);

        }

        if (max\_size < f(file\_attribute[i].size))

        {

            max\_size = f(file\_attribute[i].size);

        }

        if ( max\_mtime < strlen(file\_attribute[i].mtime) )

        {

            max\_mtime = strlen(file\_attribute[i].mtime);

        }

        if ( max\_filename < strlen(file\_attribute[i].filename) )

        {

            max\_filename = strlen(file\_attribute[i].filename);

        }

        if ( max\_extra < strlen(file\_attribute[i].extra) )

        {

            max\_extra = strlen(file\_attribute[i].extra);

        }

    }

    //ls普通输出

    if(mode == 1)

    {

        for (i = 0; i < file\_num; ++i)

        {

            char format[50];

            // 定义输出的格式

            sprintf(format, "%%%ds %%%dd %%%ds %%%ds %%%dld %%%ds %%s%%s\n",

                max\_mode, max\_links, max\_user\_name, max\_group\_name, max\_size,

                max\_mtime);

            // 按照定义的输出格式输出

            printf(format, file\_attribute[i].mode, file\_attribute[i].links,

                file\_attribute[i].user\_name, file\_attribute[i].group\_name, file\_attribute[i].size,

                file\_attribute[i].mtime, file\_attribute[i].filename, file\_attribute[i].extra);

        }

    }

    //ls |more 输出

    if(mode == 2)

    {

        //当前默认设置为初始显示10行 随后一行一行显示

        int page = file\_num > 9 ? 9 : file\_num;

        for (i = 0; i < page; ++i)

        {

            char format[50];

            // 定义输出的格式

            sprintf(format, "%%%ds %%%dd %%%ds %%%ds %%%dld %%%ds %%s%%s\n",

                max\_mode, max\_links, max\_user\_name, max\_group\_name, max\_size,

                max\_mtime);

            // 按照定义的输出格式输出

            printf(format, file\_attribute[i].mode, file\_attribute[i].links,

                file\_attribute[i].user\_name, file\_attribute[i].group\_name, file\_attribute[i].size,

                file\_attribute[i].mtime, file\_attribute[i].filename, file\_attribute[i].extra);

        }

        for(i = page; i < file\_num; ++i)

        {

            char format[50];

            // 定义输出的格式

            sprintf(format, "%%%ds %%%dd %%%ds %%%ds %%%dld %%%ds %%s%%s",

                max\_mode, max\_links, max\_user\_name, max\_group\_name, max\_size,

                max\_mtime);

            // 按照定义的输出格式输出

            printf(format, file\_attribute[i].mode, file\_attribute[i].links,

                file\_attribute[i].user\_name, file\_attribute[i].group\_name, file\_attribute[i].size,

                file\_attribute[i].mtime, file\_attribute[i].filename, file\_attribute[i].extra);

            getchar();

        }

    }

}

// 处理不带 -l 参数的 ls 命令

void lsShort(char \*dirname)

{

    DIR \*mydir = opendir(dirname);          /\* directory \*/

    // 用来暂时存储要显示的目录下的所有文件名,可以看到最大可以支持200个文件,但是每个文件名最长为20

    char filenames[200][20];

    int file\_num = 0;

    if (mydir == NULL)

    {

        // 直接显示该文件

        printf("%s\n\n", dirname);

        return ;

    }

    else

    {

        // 循环检查下面有多少文件,并把文件名全部放到filenames数组里

        struct dirent \*mydirent;            /\* file \*/

        while ( (mydirent = readdir( mydir )) != NULL)

        {

            char fname[20];

            strcpy(fname, mydirent->d\_name);

            if (fname[0] != '.' )

            {

                strcpy(filenames[file\_num], mydirent->d\_name);

                file\_num ++;

            }

        }

        closedir( mydir );

    }

    // 文件名排序

    int i, j;

    char temp[20];

    for(i = 0; i < file\_num; ++ i)

    {

        for(j = i+1; j < file\_num; ++ j)

        {

            if(strcmp(filenames[i], filenames[j]) > 0)

            {

                strcpy(temp, filenames[i]);

                strcpy(filenames[i], filenames[j]);

                strcpy(filenames[j], temp);

            }

        }

    }

    // 确定所有文件里面最长的文件名的长度

    int max\_len = 0;

    for(i = 0; i < file\_num; ++ i)

    {

        int len = strlen(filenames[i]);

        if(len > max\_len)

        {

            max\_len = len;

        }

    }

    // 得到当前终端的分辨率

    int cols = 80;

    int lines = 24;

    getTerminatorSize(&cols, &lines);

    char format[20];

    sprintf(format, "%%-%ds  ", max\_len);

    // 格式化输出,当长度大于终端的列数时,换行

    int current\_len = 0;

    for(i = 0; i < file\_num; ++ i)

    {

        printf(format, filenames[i]);

        current\_len += max\_len + 2;

        if(current\_len + max\_len + 2 > cols)

        {

            printf("\n");

            current\_len = 0;

        }

    }

    printf("\n");

}

int main(int argc, char \*\*argv)

{

    int i;

    //ls

    if (argc == 1)

    {

        lsShort("./");

        return 0;

    }

    //ls -l

    if (argc == 2 && !strcmp(argv[1], "-l") )

    {

        lsLong("./", 1);

        return 0;

    }

    //ls -l | more   ls -l |more

    if ( (argc == 4 && !strcmp(argv[1], "-l") && !strcmp(argv[2], "|") && !strcmp(argv[3], "more"))

        || (argc == 3 && !strcmp(argv[1], "-l") && !strcmp(argv[2], "|more")) )

    {

        lsLong("./", 2);

        return 0;

    }

    //ls -l directory name

    if (argc > 2 && !strcmp(argv[1], "-l") )

    {

        for(i = 2; i < argc; ++ i)

        {

            printf("%s:\n", argv[i]);

            lsLong(argv[i], 1);

            if(i != argc - 1)

                printf("\n");

        }

        return 0;

    }

    //ls directory name

    else

    {

        for (i = 1; i < argc; ++ i)

        {

            printf("%s:\n", argv[i]);

            lsShort(argv[i]);

            if(i != argc - 1)

                printf("\n");

        }

        return 0;

    }

    return 0;

}

**2.具体分析：**

新建结构体attribute储存各个系统函数获取的内容

检测argc和argv，若命令为ls则简单遍历目标文件夹内的内容输出，命令为ls-l则通过上面的结构体储存的内容，遍历输出，若为ls -l | more，则对命令的显示做出限制，随着回车来一条一条的显示。

创建一个结构体，在其中分别定义文件属性和权限，链接数，用户名，所在的用户组，文件大小，最后的修改时间，文件名，用来显示的时候要加“\*”(可执行的文件)或者“/”(目录)的额外字符。

写一个f()函数计算传进来的整数n有几位。

实现一个getTerminatorSize()来得到终端的列数和行数。

再由int型的mode，得到实际要显示的字符串，如dcbl，rwx等。

随后又根据用户的id值，得到用户名user name，根据用户组的id值，得到用户组名group name。

实现一个time2str()函数将时间的格式化字符串，注意，\*\*这里我把前面的星期和后面的年份都去掉了\*\*。

要显示的某一个文件的详细信息，并把信息放在结构体attribute中。

最后再判断ls普通输出或者是ls |more输出，再单独处理不带 -l参数的ls命令，将文件名排序后输出打印到屏幕上。

其他小细节实现，注释中已经写的很清楚了，这里不再过多重复。

(ls代码借鉴了老师发的代码，ls -l与ls -l|more为自己实现)

**六．mkdir: 创建一个目录**

**1.主要代码实现：**

/\*

\* 简单实现mkdir命令

\* 1509寝W组

\* 2021/6/28

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <fcntl.h>

#include <sys/stat.h>

//main

int main(int argc,char \*argv[])

{

    if(argc != 2)

        {

        fprintf(stdout, "参数错误:参数数量错误\n");

        return 0;

        }

        //创建权限为0600的文件夹

       if(mkdir(argv[1], O\_CREAT | 0600) == -1)

        {

        fprintf(stdout, "创建失败\n");

        return 0;

        }

    printf("your operation is successful!\n");

    return 0;

}

**2.具体分析：**

因为老师要求实现的功能里面没有mkdir，所以此功能主要是在linux下完成了内置函数mkdir()的调用。Mkdir需要传递两个参数，一个是mkdir命令，另一个是需要创建的文件名。判断一下如果argc不等于2，则打印\*\*参数错误：参数数量错误\*\*，结束掉程序。否则，也就是参数正确，则创建一个权限为0600的文件夹，如果创建失败，抛出\*\*创建失败\*\*，如果成功，抛出\*\*your operation is successful !\*\*。

**七．rmdir: 删除一个空目录**

**1.主要代码实现：**

/\*

\* 简单实现rmdir命令

\* 1509寝W组

\* 2021/6/28

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <sys/types.h>

#include <dirent.h>

#include <unistd.h>

#define PATH\_SIZE 4094

//rmdir

void my\_rmdir(const char \* path);

//main

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

{

    if (argc != 2)

    {

        fprintf(stdout, "参数错误:参数数量错误\n");

        return 0;

    }

    //rmdir

    my\_rmdir(argv[1]);

    return 0;

}

//rmdir

void my\_rmdir(const char \* path)

{

    DIR \*dirp;

    //打开文件夹

    dirp = opendir(path);

    if (NULL == dirp)

    {

        fprintf(stdout, "参数错误:路径出错\n");

        return;

    }

    //rm文件

    struct dirent \*entry;

    int ret;

    while (1)

    {

        entry = readdir(dirp);

        //文件夹为空

        if (NULL == entry)

        {

            break;

        }

        //skip . & ..

        if (0 == strcmp(".", entry->d\_name) || 0 == strcmp("..", entry->d\_name))

        {

            continue;

        }

        //删文件

        char buf[PATH\_SIZE];

        snprintf(buf, PATH\_SIZE, "%s/%s", path, entry->d\_name);

        ret = remove(buf);

        if (-1 == ret)

        {

            //若为目录删子目录

            if (ENOTEMPTY == errno)

            {

                my\_rmdir(buf);

                continue;

            }

            perror(buf);

            return;

        }

        fprintf(stdout, "rm file: %s\n", buf);

    }

    //close

    closedir(dirp);

    //删除文件夹

    ret = rmdir(path);

    if (-1 == ret)

    {

        perror(path);

        return;

    }

    fprintf(stdout, "rm dir: %s\n", path);

}

**2.具体分析：**

与mkdir类似，因为老师要求实现的功能里面没有rmdir，为了配合已经是新年的mkdir，故又再实现了一个rmdir，同理，rmdir需要传递两个参数，一个是rmdir命令，另一个是删除创建的文件名。判断一下如果argc不等于2，则打印\*\*参数错误：参数数量错误\*\*，结束掉程序。否则，也就是参数正确，则执行 my\_rmdir(argv[1])函数，rmdir首先跳过./ ../ ，如果有子文件，snprintf拼接目标文件路径，remove删除文件。如果为子文件夹则递归循环删除，最后删除总文件夹。更细致的分析在注释中都有提到。

**八．(1)ps：观察系统所有的进程数据。**

**(2)ps axjf：连同部分进程树状态显示。**

**a：不与terminal有关的所有process。**

**u：有效使用者（effective user）相关的process。**

**x：通常与a这个参数一起使用，可列出较完整信息。**

**l：较长、较详细的将该PID的的信息列出。**

**j：工作的格式（jobs format）。**

**f：做一个更为完整的输出。**

**(3)ps aux：ps aux会依照PID的顺序来排序显示。**

**1.主要代码实现加具体分析：**

建立一个ps\_info结构体来存储信息

//信息结构体

typedef struct ps\_info

{

    char pname[MAX\_LEN];

    char user[MAX\_LEN];

    int  pid;

    int  ppid;

    char state;

    struct ps\_info \*next;

}mps;

建立一个结构体指针来获取信息，将各种相关的函数布局完成，例如用来read，由进程uid得到进程的所有者user，判断name是否合法，将结果进行显示，最后是对exec的系统调用。

//获取信息

mps\* trav\_dir(char dir[]);

//read

int read\_info(char d\_name[],struct ps\_info \*p1);

//由进程uid得到进程的所有者user

void uid\_to\_name(uid\_t uid, struct ps\_info \*p1);

//判断name是否合法

int is\_num(char \*);

//显示

void print\_ps(struct ps\_info \*head);

//exec

void exec(int argc, char\* argv[]);

trav\_dir()函数的具体实现：调用DIR结构体创建一个结构体指针，建立一个链表，遍历/proc目录下所有进程目录，判断目录名字是否为纯数字，如果是且p1==NULL，则抛出分配失败。如果read\_info(direntp->d\_name,p1)!=0为真，则抛出读取进程信息error，否则便可插入新节点。

//获取信息

mps\* trav\_dir(char dir[])

{

    DIR \*dir\_ptr;

    mps \*head,\*p1,\*p2;

    struct dirent \*direntp;

    struct stat infobuf;

    if((dir\_ptr=opendir(dir))==NULL)

        fprintf(stderr,"dir error %s\n",dir);

    else

    {

        head=p1=p2=(struct ps\_info \*)malloc(sizeof(struct ps\_info));    //建立链表

        while((direntp=readdir(dir\_ptr)) != NULL)               //遍历/proc目录所有进程目录

        {

            if((is\_num(direntp->d\_name))==0)                   //判断目录名字是否为纯数字

            {

                if(p1==NULL)

                {

                    printf("malloc error!\n");

                    exit(0);

                }

                if(read\_info(direntp->d\_name,p1)!=0)         //获取进程信息

                {

                    printf("read\_info error\n");

                    exit(0);

                }

                p2->next=p1;                        //插入新节点

                p2=p1;

                p1=(struct ps\_info \*)malloc(sizeof(struct ps\_info));

            }

        }

    }

    p2->next=NULL;

    return head;

}

由进程uid可以得到进程所有者user

//由进程uid得到进程的所有者user

void uid\_to\_name(uid\_t uid, struct ps\_info \*p1)

{

    struct passwd \*pw\_ptr;

    static char numstr[10];

    if((pw\_ptr = getpwuid(uid)) == NULL)

    {

        sprintf(numstr,"%d",uid);

        strcpy(p1->user, numstr);

    }

    else

        strcpy(p1->user, pw\_ptr->pw\_name);

}

更为具体的分析我都放在了对应的.cpp文件注释里，这里不再过多说明。

总的来说，ps就是新建信息结构体struct ps\_info储存获取的各进程内信息

通过argc和argv的内容，决定输出的内容。

**九.top:用来监控linux的系统状况，是常用的性能分析工具，能够实时显示系统中各个进程的资源占用情况。**

**1.主要代码实现：**

/\*

\* 简单实现top命令

\* 1509寝W组

\* 2021/6/28

\*/

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<sys/types.h>

#include<wait.h>

//main

int main(int argc,char \* argv[])

{

    /\*/fork新的进程

    int id = fork();

    if(id == 0)

            {

        //child,执行替换操作

        //系统调用

        execvp(argv[0], argv);

        perror("error");

        exit(1);

         }

    else

            {

                //father

                    wait(NULL);

            }\*/

            //系统调用

    execvp(argv[0], argv);

    return 0;

}

**2.简单说明：**

top命令我们是直接通过系统调用的，execvp(argv[0],argv);top命令我们尝试了很久，苦于没有办法才出此下策，希望老师能体谅。

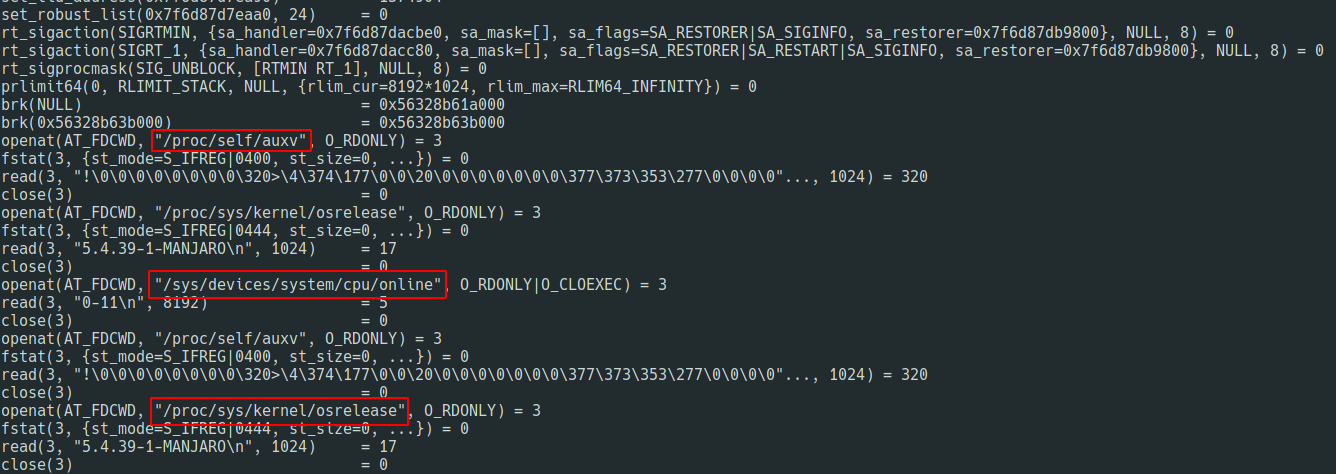
虽然没有通过代码很好的模拟实现出来，但我们有自己的一点想法可以分享：

Linux中的top命令究竟是怎么是实现的呢？

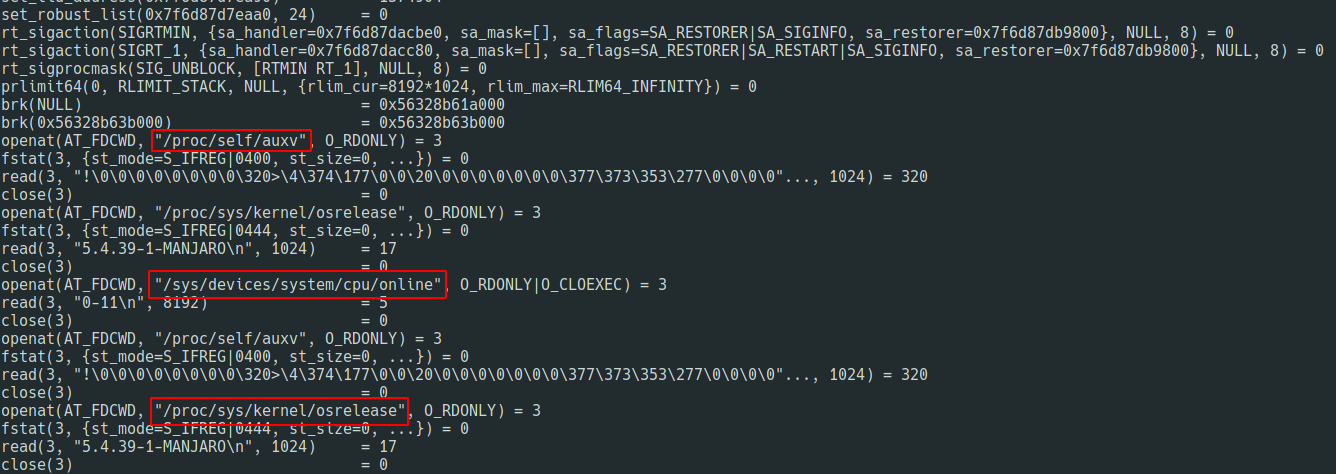
top 是procs的一部分，常用来查看系统的负载情况。Procs中除了top外，还包括ps，free，w，uptime，watch，sysctl等常用的命令。了解top命令除了直接在terminal使用之外，就是top的官方文档和源代码了。

不过在此之前，我们可以用strace top看下top命令到底做了些什么？

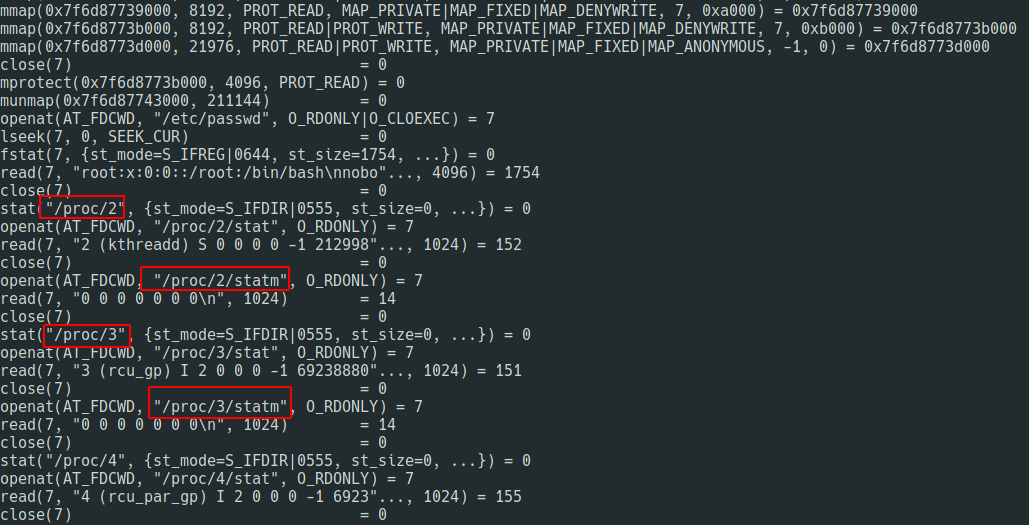
1. 首先会读取一系列的依赖文件：



2.然后会读取一些系统配置文件：



3.最后就是从/proc目录下读取进程的statm信息：



**十.cal: cal命令可以用来显示公历(阳历)的日历**

**1.主要代码实现：**

//获取本月有几天

int monthOfDay(int year, int month);

//获取从1900/1/1到目标日期的天数 (1900/1/1 周一)

int getDay(int year, int month);

//打印表

void showCal(int year, int month, int day, int days);

**2.具体分析：**

实现monthOfDay()函数来获取本月有多少天，定义一个day[]数组用于存放1到12月每个月的天数，也就是31,28,31,30,31,30,31,31,30,31,30,31，这里我们默认是平年，当然我们要考虑到闰年的情况，当年份是4的倍数同时又不是100的倍数，或者年份是400的倍数时，可以判定该年是闰年，此时day[1]=29；最后返回该月的天数。

//获取本月有几天

int monthOfDay(int year, int month)

{

    int day[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    if(year % 400 == 0 || (year % 4 == 0 && year % 100 != 0))

    {

        day[1] = 29;//二月

    }

    return day[month - 1];

}

获取从1900/1/1到目标日期的天数(1900/1/1 周一),时间戳是从1900年一月一日开始的。定义day=0，从1900到当前年份进行for循环，判断当i到达的年份是闰年的话，day+=366天，否则平年+365天。同时，我们也要将当前年份的已经经过的月份遍历,最后day+1就是当前日期，返回该日期，我们用了绿色来使其区别于shell自带的cal与其他日期。

//获取从1900/1/1到目标日期的天数 (1900/1/1 周一)

int getDay(int year, int month)

{

    int days = 0;

    for(int i = 1900; i < year; ++i)//年

    {

        if(i % 400 == 0 || (i % 4 == 0 && i % 100 != 0))

        {

            days += 366;

        }

        else

        {

            days += 365;

        }

    }

    for(int i = 1; i < month; i++)//月

    {

        days = days + monthOfDay(year, i);

    }

    days += 1;//日

    return days;//绿色

}

同时把整个得到的日期表打印出来。获取本月有多少天，整理好打印的位置，循环输出日期，当期日期用绿色标识。

//打印表

void showCal(int year, int month, int day, int days)

{

    int mdays;

    //获取本月有几天

    mdays=monthOfDay(year, month);

    //打印头

    printf("      %02d月 %d      \n", month, year);

    printf("日 一 二 三 四 五 六\n");

    //把1号置于正确的位置

    for(int i = 0; i < days % 7; ++i)

    {

        printf("%2s "," ");

    }

    //循环输出日期

    for(int i = 1; i <= mdays; ++i)

    {

        if(i == day)

        {

            printf("\033[1;40;32m%2d\033[0m ",i);//绿色

        }

        else

        {

            printf("%2d ",i);

        }

        if(days % 7 == 6)

        {

            printf("\n");

        }

           days++;

    }

    printf("\n\n");

}

总结：cal:

通过time.h内的time\_t及tm结构体，获取系统当前时间。

首先计算出1900/1/1到目标日期的天数，从而计算出日历第一行的空格数。

随后计算本月有几天，最后循环输出日期，高亮当前日期即可实现。

**十一. whoami:用于显示自身用户名称**

**1.主要代码实现：**

/\*

\* 简单实现whoami命令

\* 1509寝W组

\* 2021/6/27

\*/

#include<stdio.h>

#include<unistd.h>

#include<pwd.h>

int main(int argc, char\* argv[])

{

    struct passwd\* pass; //passwd结构体，内置name,uid,gid,dir等信息

    pass = getpwuid(getuid());

    printf("%s\n",pass->pw\_name); //pw\_name为用户名

    return 0;

}

**2.具体实现：**

Struct passwd结构体中包含：

#include <sys/types.h>

#include <pwd.h>

struct passwd {

   char \*pw\_name;                /\* 用户登录名 \*/

   char \*pw\_passwd;              /\* 密码(加密后) \*/

   \_\_uid\_t pw\_uid;               /\* 用户ID \*/

   \_\_gid\_t pw\_gid;               /\* 组ID \*/

   char \*pw\_gecos;               /\* 详细用户名 \*/

   char \*pw\_dir;                 /\* 用户目录 \*/

   char \*pw\_shell;               /\* Shell程序名 \*/

};

新建passwd结构体，getuid获取当前进程uid，getpwuid获取passwd结构体

pw\_name即为用户名。

创建一个结构体指针指向该结构体，调用getpwuid()函数，该函数根据传入的用户ID返回指向passwd的结构体 该结构体初始化了里面的所有成员，返回值为一个结构体指针，传入参数为getuid，返回值赋值给创建的指针，最后打印用户名，用pass->pw\_name获取用户名。

**十二.date: date可以用来显示或设定系统的日期与时间。**

**1.主要代码实现：**

/\*

\* 简单实现date命令

\* 1509寝W组

\* 2021/6/27

\*/

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

int main(int argc, char\* argv[])

{

    time\_t t;

    struct tm \*p;

    time(&t);

    p = gmtime(&t);

    printf("%04d年 %02d月 %02d日 星期%d %02d:%02d:%02d CST\n",p->tm\_year+1900, p->tm\_mon+1, p->tm\_mday, p->tm\_wday==0 ? p->tm\_wday+7 : p->tm\_wday, p->tm\_hour+8, p->tm\_min, p->tm\_sec);

    return 0;

}

**2.具体分析：**

date利用time.h内的time\_t和tm结构体获取时间，随后输出。

time\_t为长整型的别名，typedef long time\_t；

struct tm为定义在time.h中的结构体：

struct tm

 {

   int tm\_sec;           /\* Seconds. [0-60] (1 leap second) \*/

   int tm\_min;           /\* Minutes. [0-59] \*/

   int tm\_hour;          /\* Hours.   [0-23] \*/

   int tm\_mday;          /\* Day.     [1-31] \*/

   int tm\_mon;           /\* Month.   [0-11] \*/

   int tm\_year;          /\* Year - 1900.  \*/

   int tm\_wday;          /\* Day of week. [0-6] \*/

   int tm\_yday;          /\* Days in year.[0-365] \*/

   int tm\_isdst;         /\* DST.     [-1/0/1]\*/  
}

gmtime()函数接受一个time\_t类型的参数，返回一个struct tm\* 的结构体指针，该指针指向p，由p->tm\_year,p->tm\_mon,p->tm\_day等可以输出对应的时间信息。

**十三.pwd: pwd用来查看当前工作目录的完整路径。**

**1.主要代码实现：**

int main(int argc, char\* argv[])

{

    int bufsize = 128;

    char \*buffer = (char\*)malloc(sizeof(char)\*bufsize);//动态开辟一个大小为128字节的空间

        if (!buffer)//当buffer==NULL，分配内存失败，exit(1)退出程序

        {

        printf("allocation error1\n");

        exit(1);

        }

        while (1)

        {

        if(getcwd(buffer, bufsize) == NULL) //getcwd()获取当前工作路径

            {

                    bufsize += bufsize;

                    buffer = (char\*)realloc(buffer, sizeof(char)\*bufsize);

                    if (!buffer)

                    {

                        printf("allocation error2\n");

                        exit(1);

                    }

            }

            else

            {

                    printf("%s\n", buffer);

                    free(buffer);

                    break;

            }

        }

    return 0;

}

**2.具体分析：**

主要通过getcwd获取当前工作路径。首先定义一个变量bufsize用来表示开辟内存的大小，动态开辟该大小的空间，若失败，则抛出error。利用getcwd()函数来获取当前目录路径，若失败，则返回false，成功则返回该路径。若size太小会无法保存该地址，返回NULL，这时候便要扩展bufsize，重新realloc()空间，使其能很好地保存该路径，最后打印该路径，不要忘了释放空间。

**十四.mv: mv用来为文件或目录重命名，或将文件或目录移动到其他目录。**

**1.主要代码实现及具体分析：**

首先获取当前文件名 getFileName()，随后将当前文件名拼接到目标路径上(argv[2])。

//获取文件名

char\* getFileName(char\* fileName)

{

    char tp[100], \*name = (char\*)malloc(sizeof(char));//动态开辟内存

    int i, j=0;

    for(i = strlen(fileName) - 1; i >= 0; --i)

    {

        if(fileName[i] == '/')

        {

            break;

        }

    }

利用rename()函数对文件或文件夹重命名和移动，当新路径文件已存在则会直接覆盖。

    //直接使用rename

    if(rename(argv[1], argv[2]) == -1)

    {

        fprintf(stdout, "移动失败\n");

        return 0;

    }

    return 0;

很容易实现的功能，不用过多分析，都是常规套路。

**十五.cp: cp命令主要用于复制文件或者目录。**

**1．主要代码实现：**

    //打开原文件

    if((in\_fd = open(argv[1], O\_RDONLY)) == -1)

    {

        fprintf(stdout, "打开文件失败\n");

        return 0;

    }

   //新建复制文件

    if((out\_fd = creat(argv[2], COPY\_MODE)) == -1 )

    {

        fprintf(stdout, "创建文件失败\n");

        return 0;

    }

    //复制文件内容

    while((n\_chars = read(in\_fd, buf, BUFF\_SIZE)) > 0 )

    {

        if(write(out\_fd, buf, n\_chars) != n\_chars )

        {

            fprintf(stdout, "复制中出现错误\n");

            return 0;

        }

    }

    //读取文件内容错误

    if(n\_chars == -1)

    {

        fprintf(stdout, "读取文件内容错误\n");

        return 0;

    }

    //关闭文件

    if(close(in\_fd) == -1 || close(out\_fd) == -1)

    {

        fprintf(stdout, "关闭文件出现错误\n");

        return 0;

    }

**2.具体分析：**

Cp：打开原文件，新建复制文件，复制文件内容，关闭原文件，即可实现。

用到creat()函数，可以用来创建一个文件并以只写的方式打开。

用到open()函数，打开和创建文件，可以进行读和写的操作。

用到read()函数，可以读取一个文件，成功则返回读取的字符数，出错返回-1.

**十六.file: file命令用于辨识文件类型，通过file指令，我们得以辨识该文件的类型。**

**1.主要代码实现：**

/\*

\* 简单实现file命令

\* 1509寝W组

\* 2021/6/28

\*/

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<sys/types.h>

#include<wait.h>

//main

int main(int argc,char \* argv[])

{

    /\*/fork新的进程

    int id = fork();

    if(id == 0)

        {

        //child,执行替换操作

        //系统调用

        execvp(argv[0], argv);

        perror("error");

        exit(1);

        }

    else

        {

            //father

                wait(NULL);

        }\*/

    //系统调用

    execvp(argv[0], argv);

    return 0;

}

**2.具体分析：**

file命令我们是直接通过系统调用的，execvp(argv[0],argv);file命令我们尝试了很久，苦于没有办法才出此下策，希望老师能体谅。

虽然没有通过代码很好的模拟实现出来，但我们有自己的一点想法可以分享：

File命令的输出格式是 文件名：文件类型和编码格式，我们可以通过得到文件的文件名与文件类型与编码格式来达到模拟shell的file效果，文件名能很容易的得到，文件类型也应该可以字符串来拼接，唯一没有头绪的是编码格式的get，这个我们小组会好好地想一想解决办法，目前有限的时间内不能实现很抱歉。

**十七. cat ：cat命令用于连接文件并打印到标准输出设备上。**

**1.主要代码实现:**

//获取内容

void copy(int fdin, int fdout)

{

    char buffer[BUFF\_SIZE];

    int size;

    //读取fdin的内容放在fdout的中

    while(size = read(fdin, buffer, BUFF\_SIZE))

    {

        if(write(fdout, buffer, size) != size)

        {

            fprintf(stdout, "写入错误\n");

            exit(1);

        }

    }

    //当读入出现问题

    if(size<0)

    {

        fprintf(stdout, "读入错误\n");

        exit(1);

    }

}

//接收键盘输入与向屏幕输出

    int fd\_in = STDIN\_FILENO;

    int fd\_out = STDOUT\_FILENO;

    //1个argc

    if (argc == 1)

    {

        copy(fd\_in, fd\_out);

        close(fd\_in);

    }

    //n个argc

    else

    {

        for(int i = 1; i < argc; ++i)

        {

                //令输入文件为读入文件

            fd\_in = open(argv[i], O\_RDONLY);

            if(fd\_in < 0)

            {

                fprintf(stdout, "打开%s文件错误\n", argv[i]);

                continue;

            }

            copy(fd\_in, fd\_out);

            close(fd\_in);

        }

    }

**2.具体分析：**

Cat：STDIN\_FILENO;//键盘输入， STDOUT\_FILENO;//屏幕输出

通过上面的两个流

将要获取内容的文件的内容read进键盘输入流

随后流入屏幕输出流显示

即可实现cat功能。

定义一个buffer[]数组来储存信息。通过read()函数和write()函数实现将fdin的内容放入fdout中，copy(fd\_in,fd\_out)接受屏幕输入与输出到屏幕。

**十八.rm: rm命令用于删除一个文件或者目录。**

**1.主要代码实现：**

int rm(char\* file\_name)

{

    char file\_path[128];

    strcpy(file\_path, file\_name);

    struct stat st;

    //找不到文件

    if(lstat(file\_path, &st) == -1)

    {

        return -1;

    }

    //是否为常规文件

    if(S\_ISREG(st.st\_mode))

    {

        //unlink失败

        if(unlink(file\_path) == -1)

        {

            return -1;

        }

    }

    //是否为文件夹

    else if(S\_ISDIR(st.st\_mode))

    {

        fprintf(stdout, "无法删除：其为一个目录\n");

        return -1;

    }

    return 0;

}

**2.具体实现：**

**struct stat为sys/types.h内置的一个结构体，作用是找到目标文件。**

**struct stat**

**{**

**dev\_t       st\_dev;     /\* ID of device containing file -文件所在设备的ID\*/**

**ino\_t       st\_ino;     /\* inode number -inode节点号\*/**

**mode\_t      st\_mode;    /\* protection -保护模式?\*/**

**nlink\_t     st\_nlink;   /\* number of hard links -链向此文件的连接数(硬连接)\*/**

**uid\_t       st\_uid;     /\* user ID of owner -user id\*/**

**gid\_t       st\_gid;     /\* group ID of owner - group id\*/**

**dev\_t       st\_rdev;    /\* device ID (if special file) -设备号，针对设备文件\*/**

**off\_t       st\_size;    /\* total size, in bytes -文件大小，字节为单位\*/**

**blksize\_t   st\_blksize; /\* blocksize for filesystem I/O -系统块的大小\*/**

**blkcnt\_t    st\_blocks;  /\* number of blocks allocated -文件所占块数\*/**

**time\_t      st\_atime;   /\* time of last access -最近存取时间\*/**

**time\_t      st\_mtime;   /\* time of last modification -最近修改时间\*/**

**time\_t      st\_ctime;   /\* time of last status change - \*/**

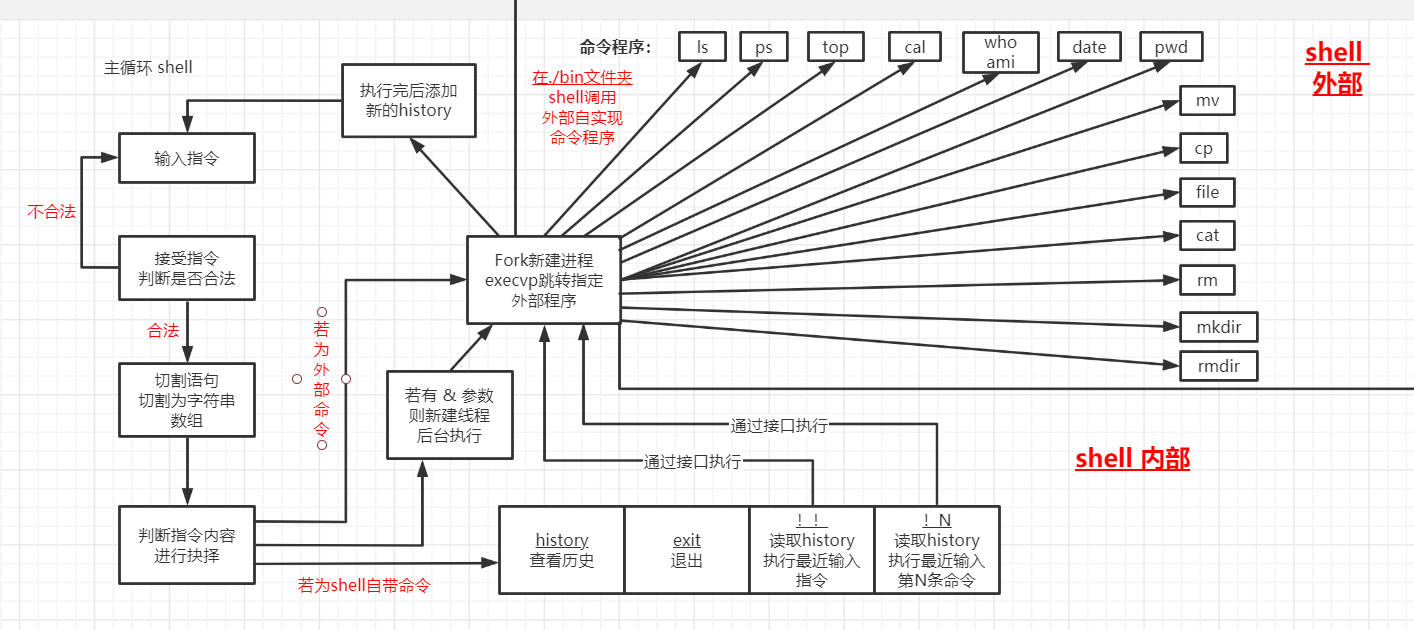
**};**

S\_ISREG判定是否为常规文件，若unlink==-1，则删除失败。

S\_ISDIR判定文件夹(不执行)，为文件夹则不删除。

# 四、流程圖

**主要大纲：**

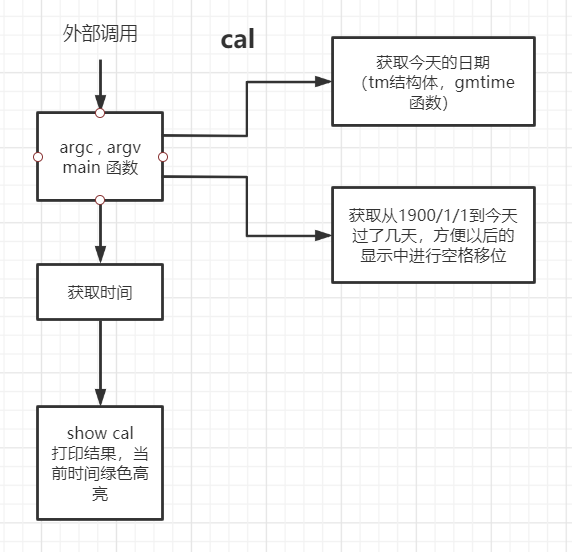


附件pdf：

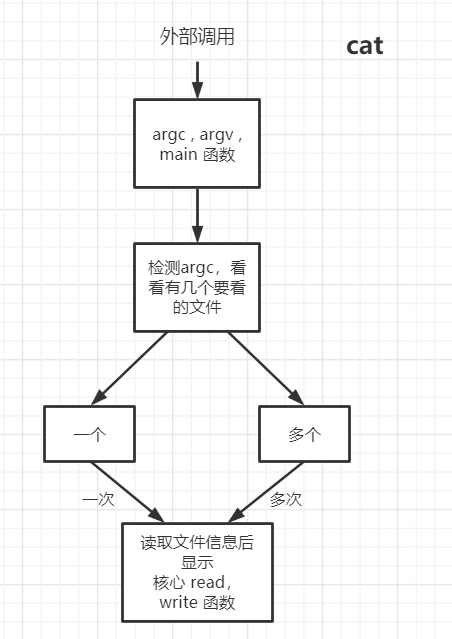


(注：可能看不太清，故这里将.pdf文件上传)

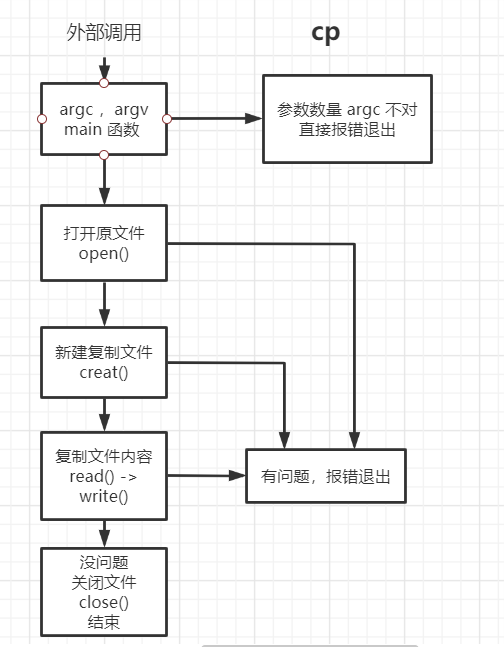
**1.cal指令流程图：**



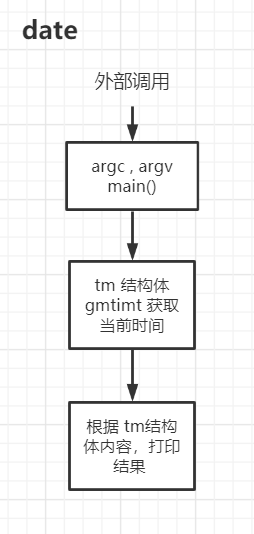
**2．cat指令流程图：**



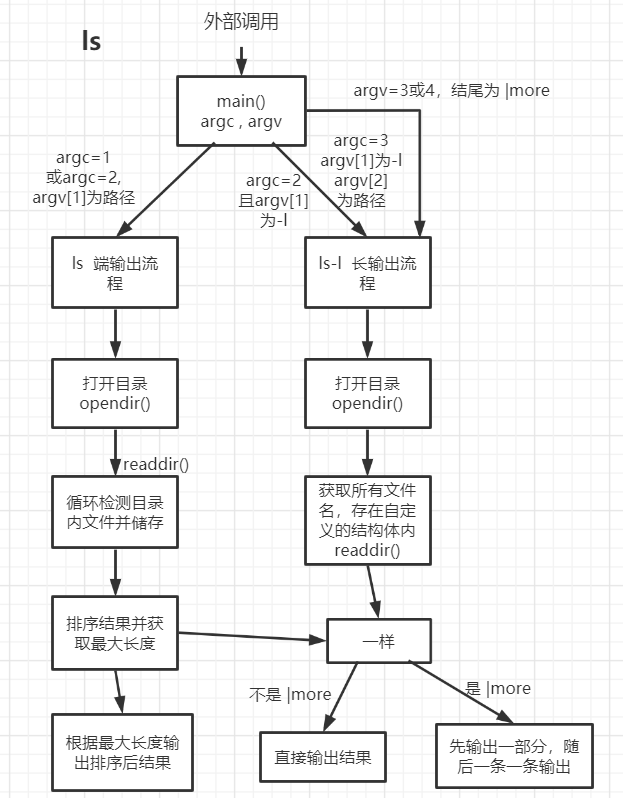
**3.cp指令流程图：**



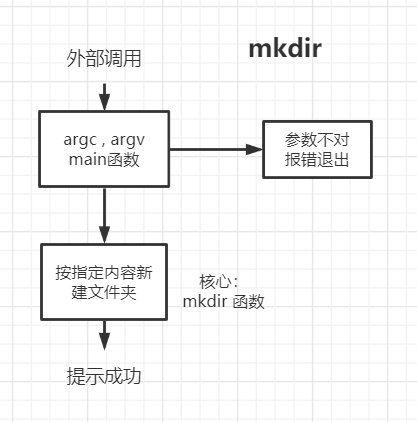
**4.date指令流程图：**



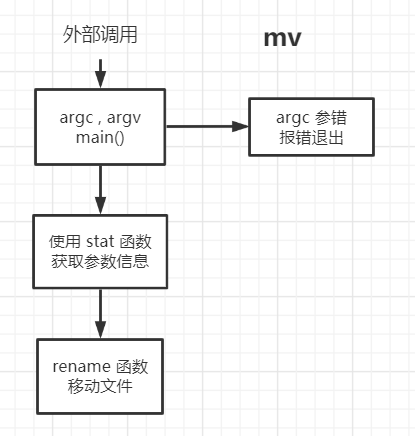
**5. ls及ls -l 及ls -l |more指令流程图：**



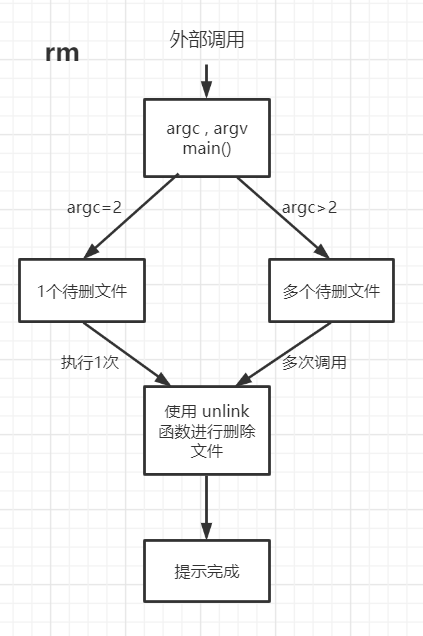
**6.mkdir指令流程图：**



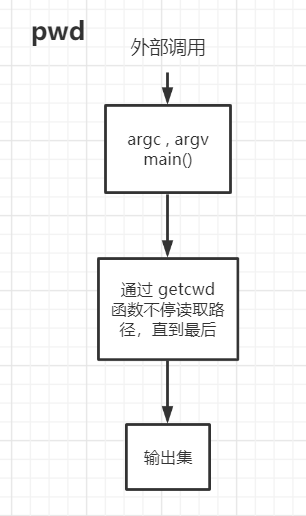
**7.mv指令流程图：**



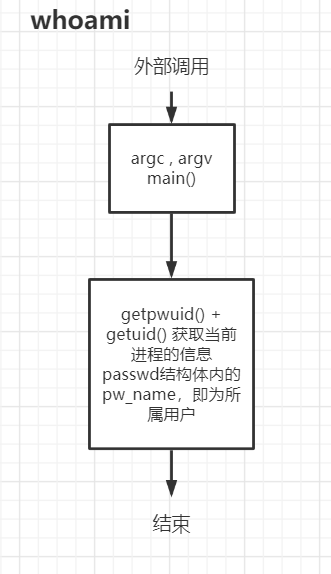
**8.rm指令流程图：**



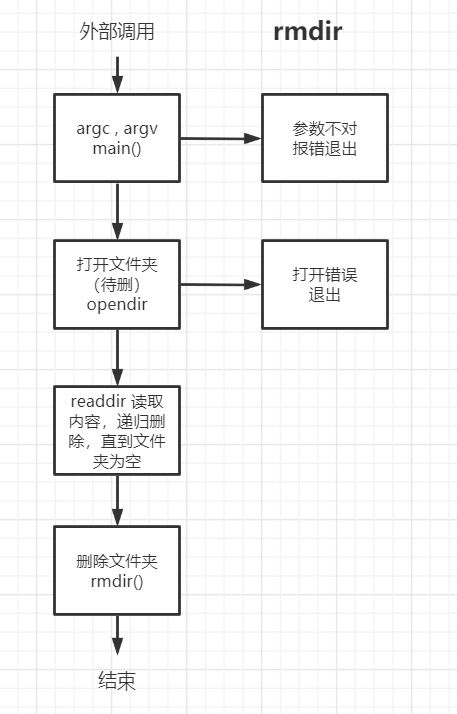
**9.pwd指令流程图：**



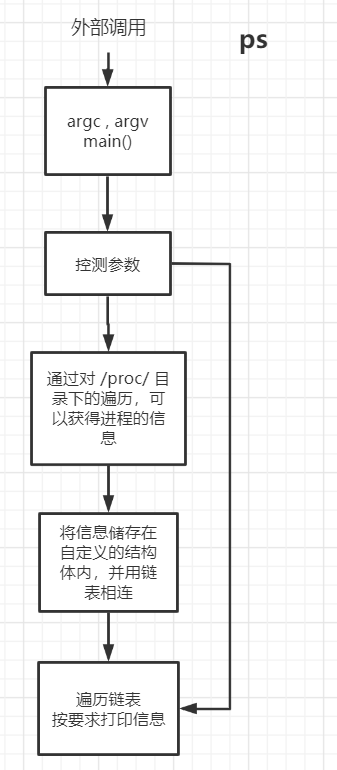
**10.whoami指令流程图：**



**11.rmdir指令流程图：**



**12.ps指令流程图：**



13.exit指令太过简单，没有流程图的必要，file与top直接为execvp(argv[0], argv);实现，这里也不画流程图。

# 测试结果與說明

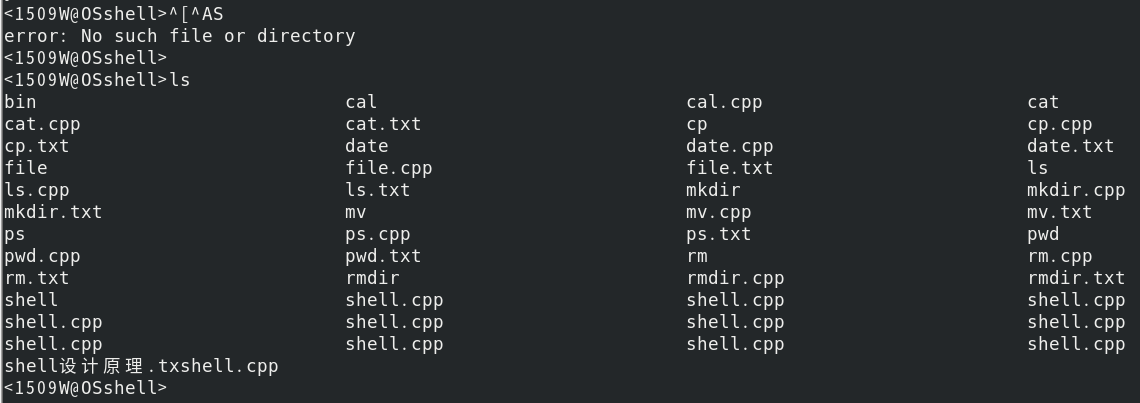
1. **cat功能测试：**

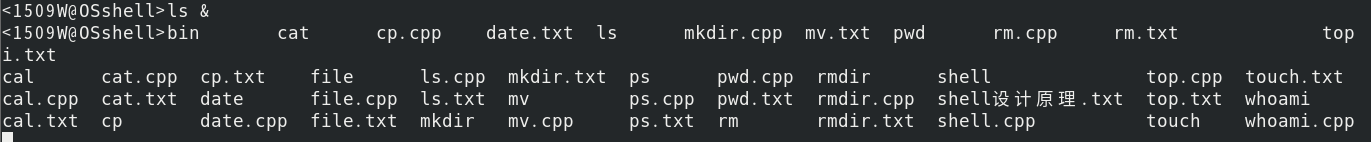
进入shell，调用自己的cat命令查看cp.cpp里的内容



**2.ls功能测试：**

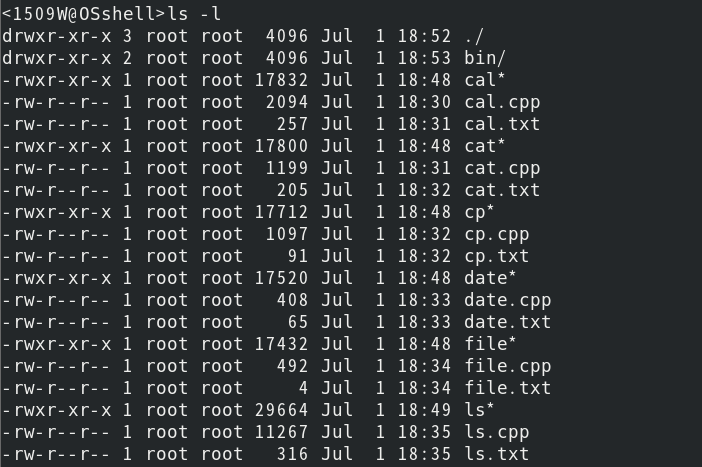
进入自己实现的shell，分别调用自己的ls与&后台运行

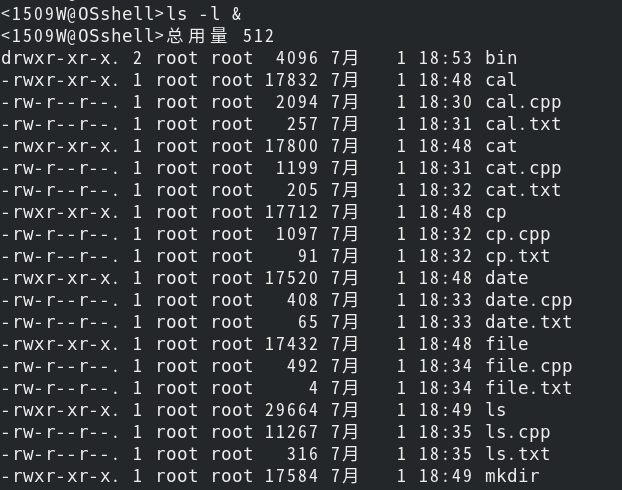




**3.ls -l功能测试：**

进入自己实现的shell，分别调用自己的ls-l与&后台运行

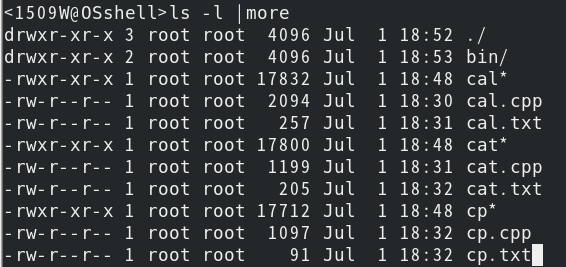


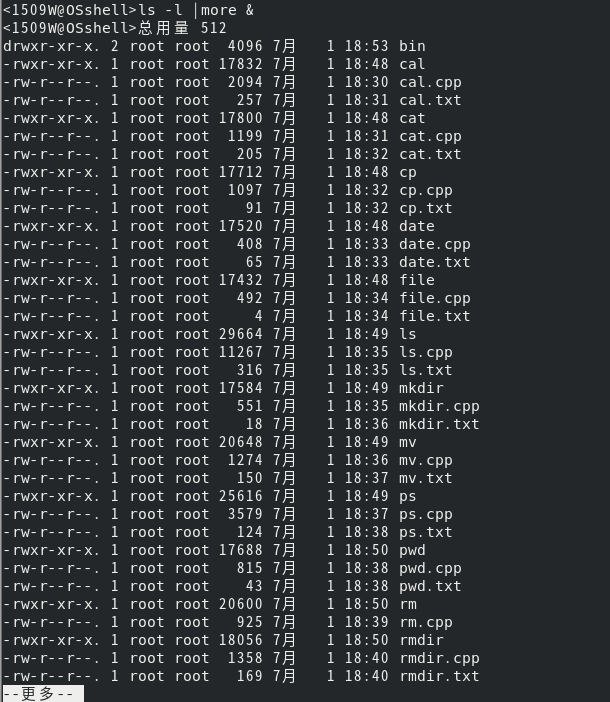


**4.ls -l |more功能测试：**

进入自己实现的shell，分别调用自己的ls -l|more与&后台运行

这里是按照老师要求，先打印出10行，接着按enter回车键一行一行输出后面的信息。

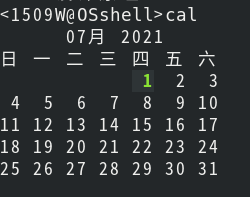


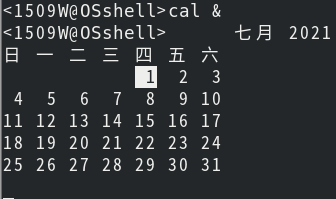
****

**5.cat功能测试：**

进入自己实现的shell，分别调用自己的cat与&后台运行

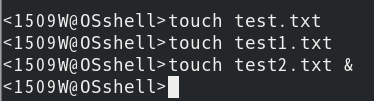
为了有区分度，这里特意把原shell的白底黑字换成了绿色高亮。

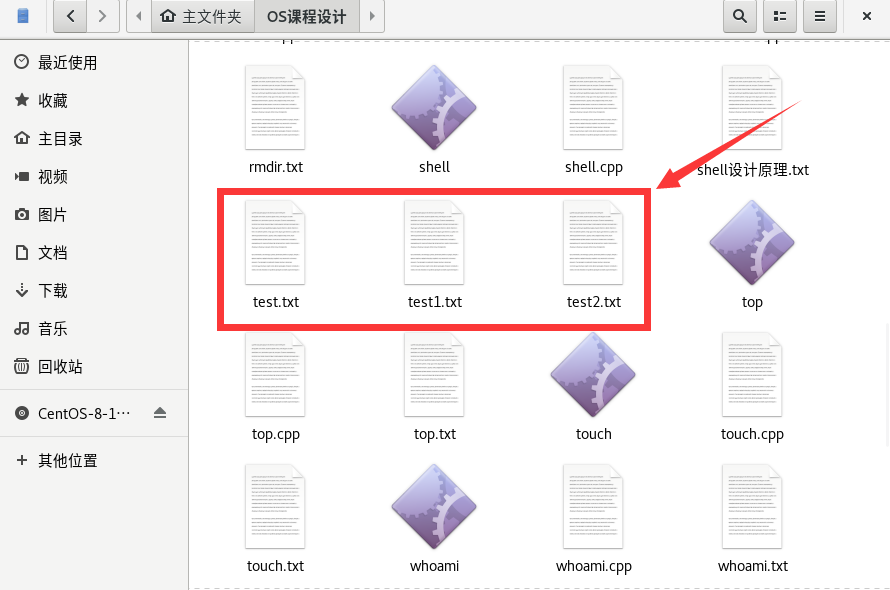




**6.touch功能测试：**

进入自己实现的shell，分别调用自己的touch与&后台运行

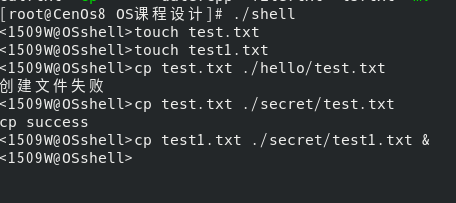


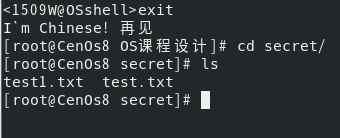


**7.cp功能测试：**

进入自己实现的shell，分别调用自己的cp与&后台运行

操作失败则抛出提示：创建文件失败



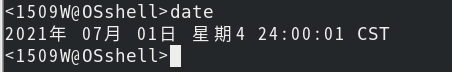


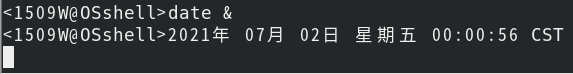


**8.date功能测试：**

进入自己实现的shell，分别调用自己的date与&后台运行

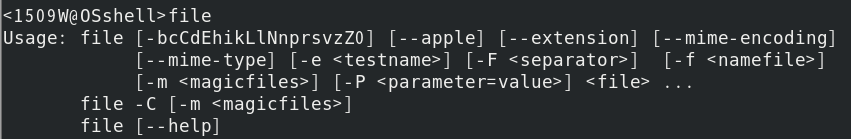
为了有区分度，这里特意把原shell星期几的中文改成了数字。

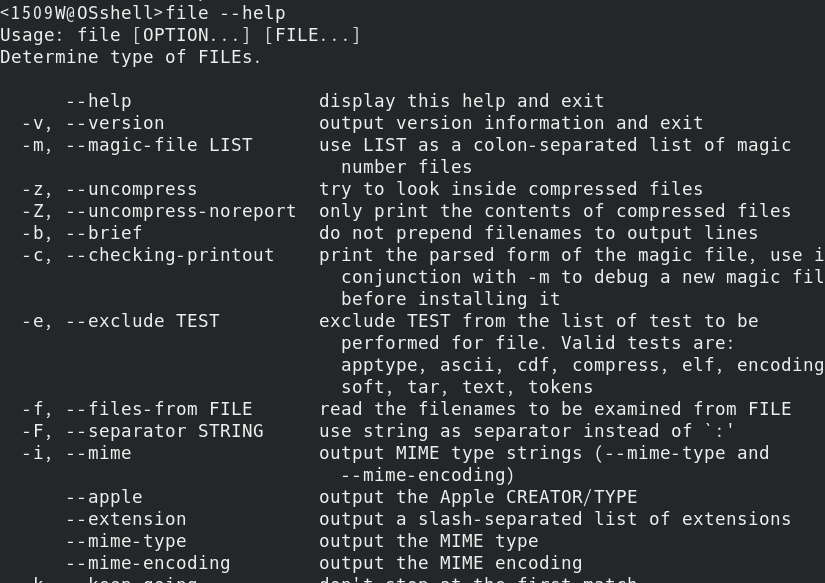


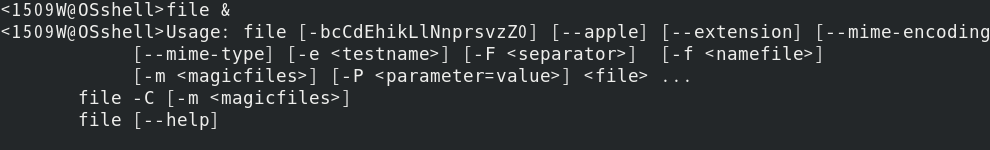


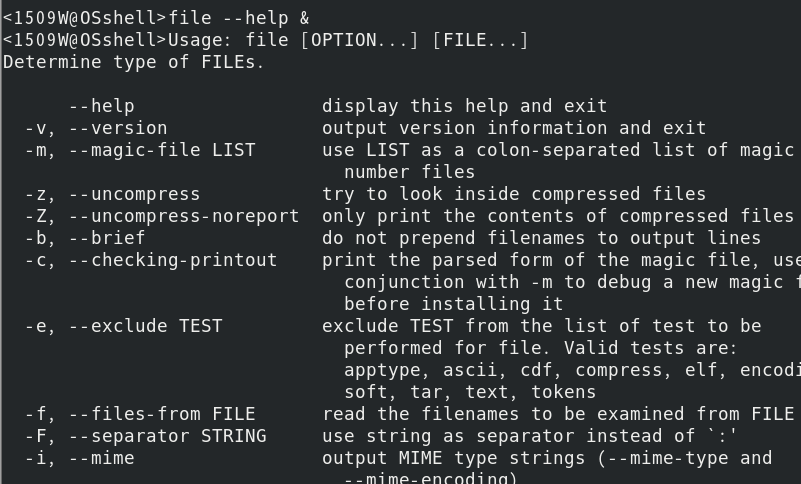
**9.file功能测试：**

进入自己实现的shell，分别调用自己的file与&后台运行





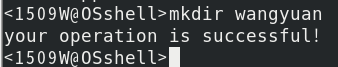


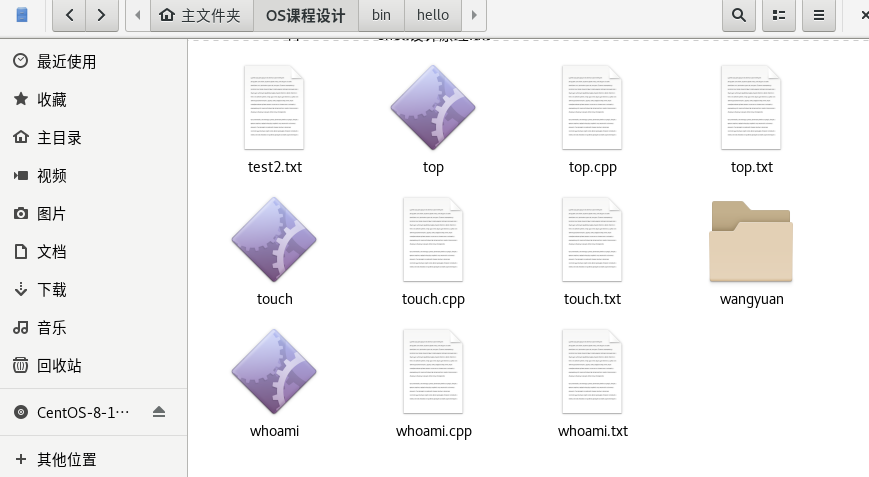


**10.mkdir功能测试：**

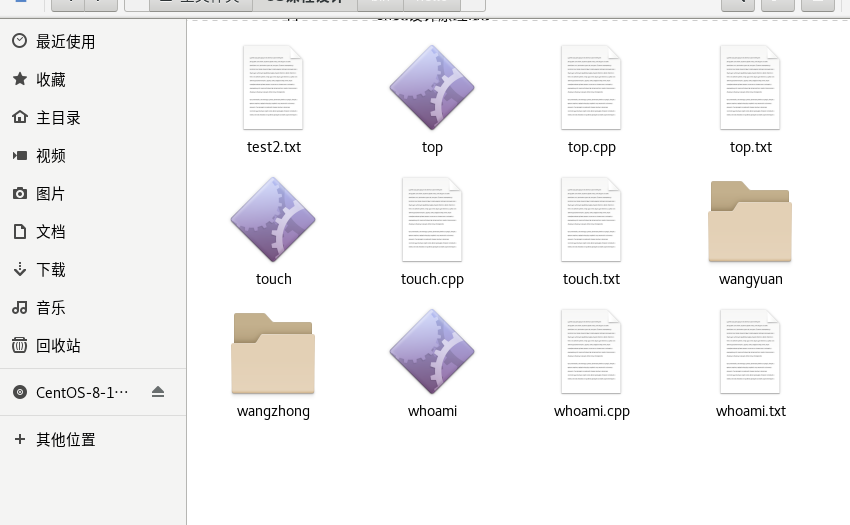
进入自己实现的shell，分别调用自己的mkdir与&后台运行

这里为了跟原shell有一定的区分度，特意在创建文件夹成功时抛出一句提示：your operation is successful！



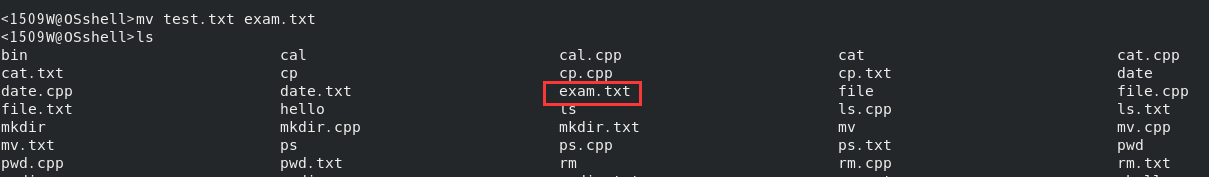




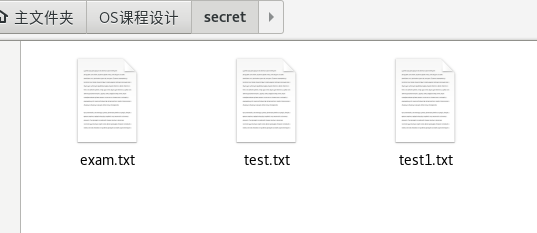


**11.mv功能测试：**

进入自己实现的shell，分别调用自己的mv与&后台运行

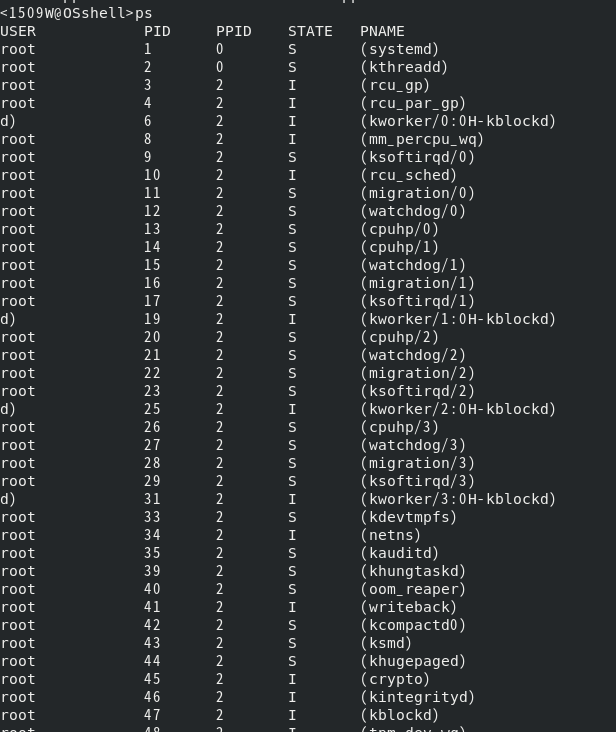


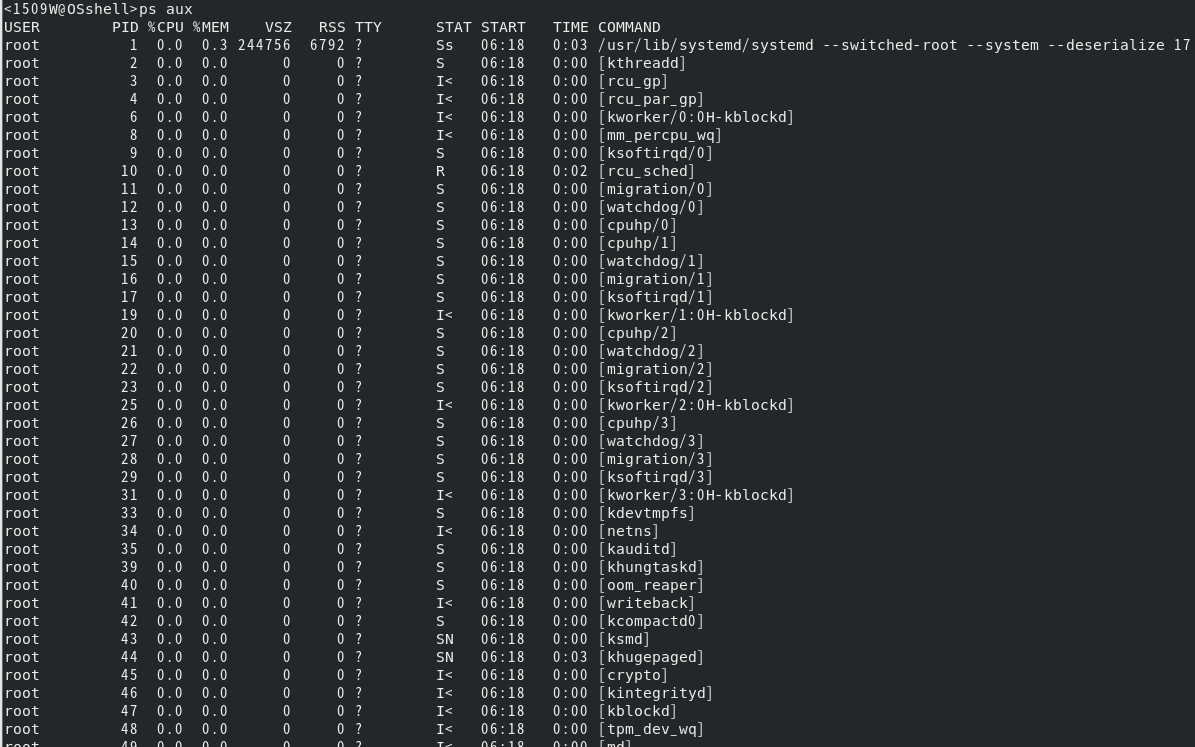




**12.ps功能测试：**

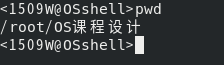
进入自己实现的shell，分别调用自己的ps与&后台运行

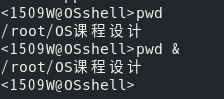




**13.pwd功能测试：**

进入自己实现的shell，分别调用自己的pwd与&后台运行





**14.rm功能测试：**

进入自己实现的shell，分别调用自己的rm与&后台运行



**15.rmdir功能测试：**

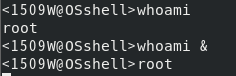
进入自己实现的shell，分别调用自己的rmdir与&后台运行





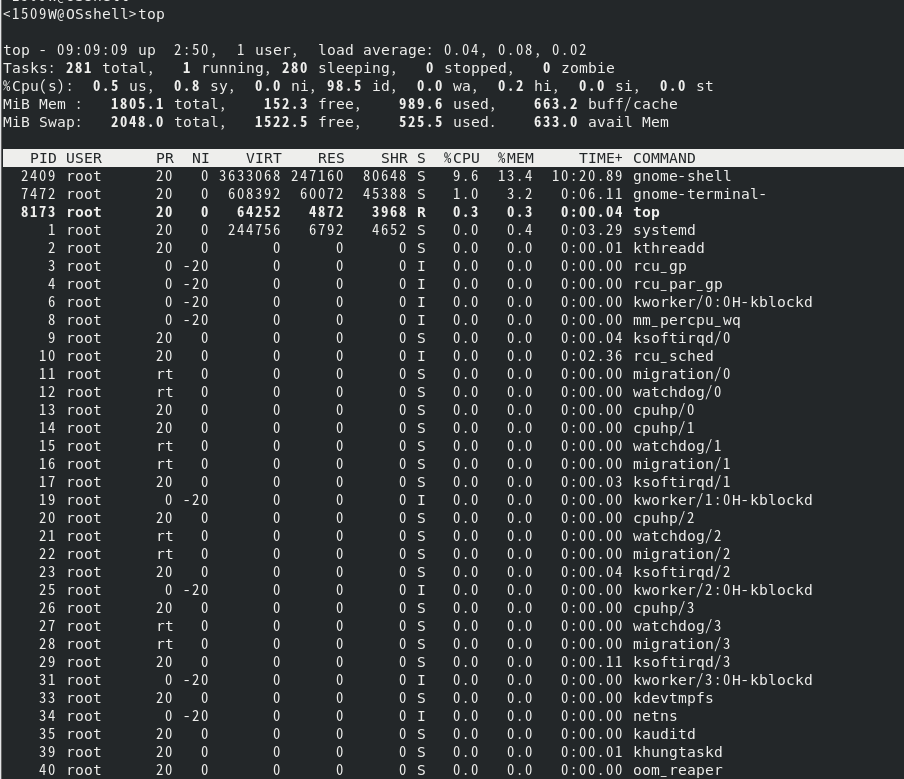
**16.whoami功能测试：**

进入自己实现的shell，分别调用自己的whoami与&后台运行



**17.top功能测试：**

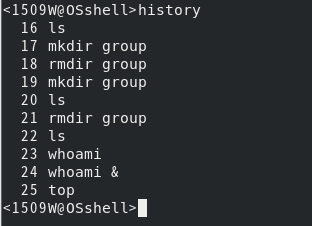
进入自己实现的shell，分别调用自己的top与&后台运行



**18.history+！！+！N功能测试：**

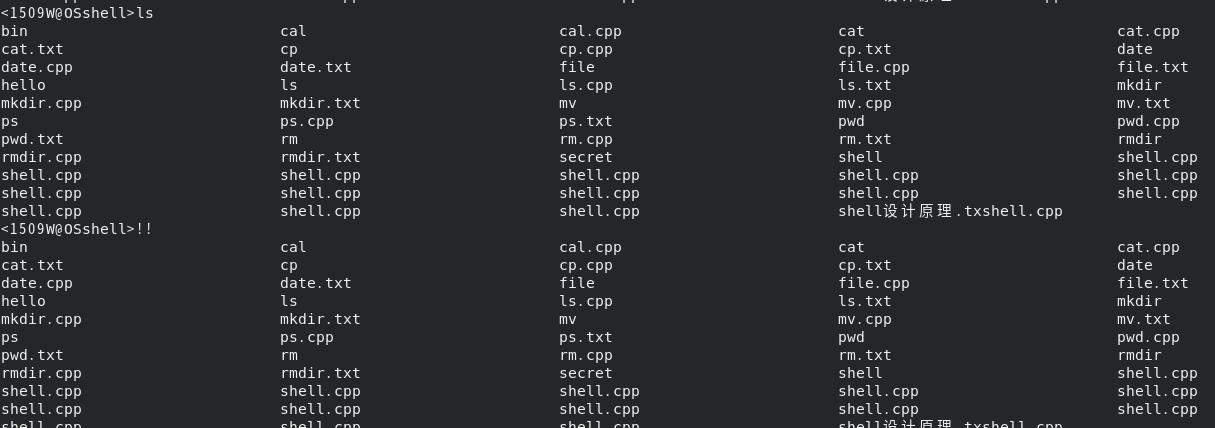
进入自己实现的shell，分别调用自己的history与&后台运行

严格按照老师的要求，拉取最近十条执行过的命令，并标注其id号，若执行的命令数超过十条，则舍弃前面的指令，保留最近十条。



进入自己实现的shell，分别调用自己的！！与&后台运行

！！实现的是执行最近执行过的指令，这里为执行ls。



进入自己实现的shell，分别调用自己的！N与&后台运行

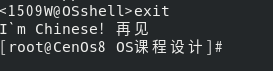
！N为依照id号拉取该id号对应的指令，并执行。



**19.exit功能测试：**

进入自己实现的shell，分别调用自己的exit与&后台运行

抛出一句提示：\*\*I am Chinese！再见\*\*



# 收获、体会和建议

此次操作系统整合性shell用户接口课程设计，我深有感悟：

一开始刚看到老师的题目跟任务说明书时，说实话脑子是很懵的。Shell接口？什么是shell接口呢？我压根没听过这玩意儿，经过上网查阅资料得知，shell接口类似于windows下的CMD，它是linux下的一个命令解释器，它的作用是接受用户输入的命令，然后调用相应的应用程序。它的操作完全是基于命令行实现。

了解了一些基本的情况之后，我们意识到这个课程设计不是单单靠一个人的能力就能在短时间完成的，我们需要分工合作。老师也让我们每四个人一小组，以小组为单位完成该课程设计。

可是上手起来还是会很困难，因为在此之前我们很少接触linux操作系统，更不要提在linux环境下进行编程了。这确实是一个难点，不过万事开头难，我们有信心完成老师布置的任务。

在向老师以及学长学姐们请教过后，我们小组知道了要有一个linux环境，这样才能开始我们的后续步骤，而怎么去搭建一个友好的linux环境呢？他们建议我们使用虚拟机进行一个虚拟的环境搭建，这样即省去了繁杂的安装系统步骤，又不怕系统被我们不小心折腾坏了。于是我们选择了VMware Workstation Pro16。

通过在网页上查找资料，我们在清华大学镜像站下载了centos8的iso镜像，由此来安装centos，至于为什么不选择官网下载呢？因为清华大学镜像站下载国外的软件会比较快。

搭建好虚拟机环境后，我们开始各自明确的分工。小组进行讨论构架思路的时候，会发生不同意见，一开始大家意见不统一，主要是linux接触的不多，没太理解老师的要求。一开始准备集成文件，后面才知道要用fork转换，去了解了一下，linux的shell也是一个一个文件去调用的。

我们一开始是想把所有代码整合起来，也就是他是一个整体，由main函数往外散开，后来了解到真正的shell实现是由shell这个总指挥调用其中各个小功能，相当于shell提供一个接口，里面的各个小功能像ls，cd，mkdir，rm之类的都通过shell这个平台有施展本领的机会。确定了这个大致方向之后，我们决定每个人分工写一些小功能，最后再都留一个接口在shell上，实现fork调用。

但是每个人思路不同，所使用变量不同，我们最后在进行代码拼接到一起进行运行的时候我们遇到了许多问题，全局变量与局部变量的冲突、代码的执行语序等问题凸显了出来。又比如在ubuntu下跟centos下也有些细微的差距，我们编写的shell在ubuntu 下会发生segmentation fault,并且带有core错误，初步判断应该是越界导致的。随着实践地越发深入，各种问题也逐渐暴露出来。

许多地方由于Linux与windows系统的不同我们必须去寻找不同的函数去替代，或者自己写出来。但同时linux也有各种不同的头文件去供我们使用，并且编译环境比windows好太多。

令人欣慰的是，最后我们都能很好地克服这些困难。

组长规划任务：

框架结构由组长进行规划与撰写，通过小组分工的模式，每个人都能领到自己的任务，同时在遇到问题时，大家一起讨论， 相互合作，相互配合，争取每个人都尽到自己应尽的责任。我们组每个人都很好的完成了自己的任务!

发挥自己的特长

小组成员每个人都有各自擅长的方面，有人负责逻辑结构分析，有人负责寻找资料，有人负责解决编写代码过程出现的BUG,有人负责链表的构建与检查、有人负责代码的整合。总的来说，word, PPT, 代码，编译环境，资料查询，这都是我们需要严格分工合作的地方，-个shell的实现不单单是靠一个人就能很好地完成的。

知识点融合

操作系统文件系统课设之前，我们做了许多老师给的实验报告，Linux下用C语言实现Is,

我们利用之前报告中所积累的经验与教训，结合之前实验报告所学到的知识点，进行融合，用于

实践。之前做过的OS实验如linux下操作系统的调用，磁盘调度都能为之所用。

经过我们小组每个人的努力，老师要求的任务都很好的完成，但我们应该对其中暴露出来的问题有所思考。我们对理论与实践的结合不够深入。逻辑结构不够严密，在实践中小错误频发。在linux环境下对C语言的编写不够熟练，甚至于对其中的头文件的不习惯。调试更是一个大问题，在此之前，我们都是在windows下利用集成开发环境进行语言的编写，如vs2019，dev-c++，code blocks等，但linux下我们没有找到很好地替代品，只能使用gdb进行调试，说实话，gdb很难上手。对vim的操作也觉得很难受，vim的各种模式，对代码的修改保存等，对各段代码之间的跳转等都掌握的不是很好，所以用起来效率很低。当然，最主要的issue是对文件系统的核心原理了解肤浅，代码编写效率低下。

一点总结：本次实验主要对我们对操作系统shell的理解与掌握的考察,我们通过构建链表与结构体去实现shell的各个功能的联系，同时搭建二级链表， 来实现了shell与文件的连接。本次实验是小组合作的形式进行的，非常考验我们的团队合作能力与组长的导指挥协调能力，实验的初期，我们三个人也是大脑一片空白，对整个shell的构建还没有一个清晰的框架结构，后来根据我们不断地相互讨论，查找资料，利用之前上课与实验中学到的东西，我们慢慢地搭起了整个shell的框架结构，通过组长的协调，小组成员的分工配合，每次完成一部分结构体的撰写，大家都热血沸腾，你追我赶的去完成自己的任务,同时还积极讨论，主动分担代码的撰写与BUG的修改，我们充分感受到了小组成员的配合与默契，同时在不断的讨论与修改当中我们对整个文件系统的功能和内部实现有了充分的了解。最大的收获就是在linux环境下的编程功力大幅提升，以后也能多多运用虚拟机来了解更多的操作系统。

# 附录

# 源碼



# 主要参考文献

1. 汤小丹、梁红兵、哲凤屏、汤子瀛（2014）。计算机操作系统 第四版，西安电子科技大学出版社。
2. 梁红兵、汤小丹、汤子瀛（2014）。计算机操作系统 第四版 学习指导与题解，西安电子科技大学出版社。
3. A. Silberschatz, G. Gagne, P. B. Galvin (2013). Operating system concepts. John Wiley & Sons.
4. CSDN社区.实现自己的shell.

https://blog.csdn.net/zhuxinquan61/article/details/47183619