华东师范大学数据学院上机实践报告

课程名称：操作系统　　**年级**：2022 级　　**上机实践成绩**：

指导教师：翁楚良　　**姓名**：郭夏辉

上机实践名称：Shell 及系统调用　　**学号**：10211900416　　**上机实践日期**：2023 年 3 月 16 日

上机实践编号：No.1　　**组号**：1-416　　**上机实践时间**：2023 年 3 月 16 日

一、 目的

1.安装 MINIX 操作系统

2.学习 Shell，系统编程，实现一个基本的 Shell

二、 内容与设计思想

Shell 能解析的命令行如下:

1．带参数的程序运行功能。

program arg1 arg2 ... argN

2．重定向功能，将文件作为程序的输入/输出。

(1) “>”表示覆盖写

program arg1 arg2 ... argN > output-file

(2) “>>”表示追加写

program arg1 arg2 ... argN >> output-file

(3) “<”表示文件输入

program arg1 arg2 ... argN < input-file

3．管道符号“|”，在程序间传递数据。

programA arg1 ... argN | programB arg1 ... argN

4．后台符号&，表示此命令将以后台运行的方式执行。

program arg1 arg2 ... argN &

5．工作路径移动命令 cd。

6．程序运行统计 mytop。

7．shell 退出命令 exit。

8．history n 显示最近执行的 n 条指令。

三、 使用环境

编辑与开发:Visual Studio Code

虚拟机系统:MINIX 3.3

物理机系统:Windows 10 专业版 19042.1110

虚拟机程序:VMware Workstation 16 Pro

连接虚拟机:MobaXterm Professional v20.0

四、 实验过程

1.安装 MINIX 操作系统

首先在物理机中安装 VMware，然后在网络上下载 MINIX3.3 的镜像

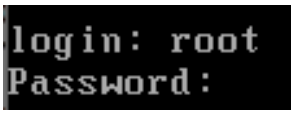
(https://wiki.minix3.org/doku.php?id=www:download:start).下载结束后，要进行解压得到.iso 镜像。在 VMware 中新建虚拟机，设置好需求的内存、硬盘空间等参数，注意此时安装系统的映像文件目录是下载的 MINIX3.3 那个.iso 文件所在的目录，并且网络模式要先设置为 NAT 模式(虚拟机借助物理机的 IP 地址，可以访问外网)。

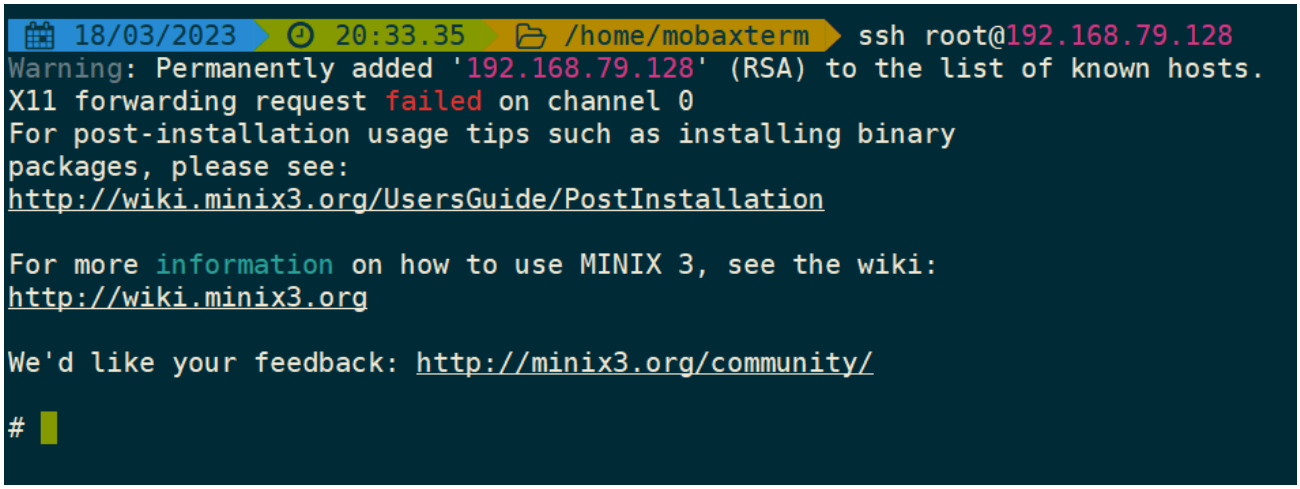
之后一路设置后，成功进入了虚拟机中，根据提示输入 root 注册，然后输入 setup 开始安装，之后一直 Enter(默认配置)完成安装，再 reboot 重启。重启后注意虚拟机要益处 MINIX 镜像文件(防止重复安装)。输入几个基本的命令(pwd,ls 等)发现安装成功。

在安装成功后，依次输入以下的命令安装开发环境

```
1. 在线更新软件仓库元数据，输入 pkgin update
2. 在线安装 git 版本控制器，输入 pkgin install git-base
3. 在线安装 SSH 远程访问，输入 pkgin install openssh
4. 在线安装 VIM 编辑器，输入 pkgin install vim
5. 在线安装 clang 编译器，输入 pkgin install clang
6. 在线安装运行链接库，输入 pkgin install binuti
```

安装完成后，输入 shutdown -h now 后等待出现 MINIX has halted 提示之后再关闭虚拟机(防止突然关机损坏文件)。在关机后将网络模式从 NAT 模式切换到桥接模式，这样虚拟机才有独立的 IP 地址。但是，此时自己设置为桥接模式时，总是卡在一个界面不动，输入什么都没有反应。在助教学长的帮助下，我切换到了 “仅主机模式”后发现内部可以运行了，然后再打开虚拟机，输入 ifconfig 查询 IP 地址。正常运行的情况下我用 passwd 修改了登录密码(SSH 的使用需要提前修改密码)，这时需要重启(openssh 包括了客户端和服务端，但是服务端是需要设置 root 用户密码，不然无法初始化密钥。第一次重启后，ssh 初始化密钥会显示)。一切顺利后我在 MobaXterm 中用 ssh 连接到终端。





至于物理机和虚拟机之间的文件传输问题，在上述内容的基础之上，物理机中通过 FileZilla，通过 sftp 协议(比如我这个的地址是 sftp://192.168.79.128)然后用户名 root，密码是 root 的密码连接到虚拟机之后再进行文件的上传和下载。

2.学习 Shell，系统编程，实现一个基本的 Shell

总而言之，从一个上学期只完成过 CSAPP 的 shell lab 的小白开始，到完成一个简单的 shell 还是十分有挑战性的。

2.0 库函数

为了让我们需要的函数都在库中，我先把可能用到的库放在了源代码最起始位置。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <curses.h>
#include <limits.h>
#include <termcap.h>
#include <termios.h>
#include <time.h>
#include <assert.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <grp.h>
#include <pwd.h>
#include <dirent.h>

#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/times.h>
#include <sys/time.h>
#include <sys/select.h>

#include <minix/com.h>
#include <minix/config.h>
#include <minix/type.h>
#include <minix/endpoint.h>
#include <minix/const.h>
#include <minix/u64.h>
#include <paths.h>
#include <minix/procfs.h>
```

2.1 完成 Shell 的主体 main()

什么是 Shell 的主体？其实就是一个 while 循环，不断地接受用户键盘输入行并给出反馈。然后根据这个原理，可以较轻松地完成 main 函数。

在 main()函数之前，我先写一下需要用到的函数名和宏定义

```
void eval(char *cmdline);
```

```
int cnt=0;//当前读到了第几条命令
char cmd_history[MAXCMD][MAXLINE];
```

```
#define MAXCMD 1024 //最大记录的命令行数
#define MAXLINE 1024 //每行最大长度
```

```
int main(int argc,char **argv){
    char cmdline[MAXLINE];
    fprintf(stdout,"Hello there!I'm tommy and this is the first project of OS in ECNU DASE\n");
    while(1){
        printf("tommyshell  %s >",getcwd(NULL,NULL));
        if(fgets(cmdline,MAXLINE,stdin)==NULL){
            //通过标准输入将命令行读入到 cmdline 中
            printf("Error occurs when input\n");
            continue;
        }
        strcpy(cmd_history[cnt],cmdline); //备份
        ++cnt;
        eval(cmdline);
    }
    return 0;
}
```

2.2 完成输入的每行字符串的转化 parseline();

2.1 中尚未实现的 eval()函数是过分复杂的，我等一下再具体来完善。现在我先设计一个函数 parseline(char *cmdline,char *argv[MAXPIPE][MAXARG],struct Detail *detail),这个函数把每行输入的字符串 cmdline 转化成了

1. 二维数组 argv 这一行中各个指令以及它们各自的参数,以字符串的形式存在;
2. 这一行的具体情况(是否是后台运行，输入模式，输出模式，管道数量等)detail

如果不通过管道，一行输入只能执行一个指令，所以我通过管道的数量来描述这一行指令的数量。首先，我先要定义描述这一行输入之细节的结构体 Detail.

```
struct Detail{
    //这个结构体是在记录每一行的详细信息
    int bg;
    int pipenum; //通过管道连接的程序的数量
    int input,output; //输入输出模式
    //input 0 标准输入 1 文件输入
    //output 0 标准输出 1 文件输出(覆盖) 2 文件输出(追加)
    char *input_file;//输入文件地址
    char *output_file;//输出文件地址
};
```

然后是相关的宏定义

```
#define MAXARG 64 //命令行中某命令最大参数数量
#define MAXPIPE 64 //通过管道连接的最大程序数量
```

函数名

```
void parseline(char *cmdline,char *argv[MAXPIPE][MAXARG],struct Detail *detail);
```

开始预处理吧!

```
//parseline 读取 cmdline 然后把结果放到 argv 和 detail 中
char *locate;//用于定位
int argc; //一个命令后面所接参数数量
int cnt1; //目前记录了的命令数量

cmdline[strlen(cmdline)-1] = ' '; //把最后一个字符'\n'置为' '
while (*cmdline&&(*cmdline ==' '))cmdline++; //丢弃第一个指令前多余的空格

detail->input=0;detail->output=0;detail->bg=0;
cnt1=0;argc=0;
locate=strchr(cmdline,' ');
```

这里有个细节，就是 strchr()函数查找字符串中的一个字符，并返回该字符在字符串中第一次出现的位置。

接下来就是一个一个去判断了，循环的样子大概是这样的.这里用到的 C 库函数 char *strtok(char *str, const char *delim) 分解字符串 str 为一组字符串，delim 为分隔符。该函数返回被分解的第一个子字符串，如果没有可检索的字符串，则返回一个空指针。strtok 和 strchr 搭配使用，极大地精简了代码。注意每次都要去除多余的空格。

```
while (locate) {
    char *arg_tmp=strtok(cmdline," ");
    //各种而样的判定，主要是在判断一些特殊的情况 >,<,>>,|,&
    *locate='\0';
    cmdline=locate+1;
    while (*cmdline && (*cmdline == ' '))cmdline++;
    locate=strchr(cmdline,' ');
}
```

各种而样的判定虽然繁琐，但每段的操作相似，所以代码量还好.

```
    if(!strcmp(arg_tmp,"<")){
        detail->input=1;
        *locate='\0';
        cmdline=locate+1;
        while (*cmdline && (*cmdline ==' '))cmdline++;
        locate=strchr(cmdline,' ');
        if(!locate){
            printf("Where is the file?!");
            return;
        }
        detail->input_file=strtok(cmdline," ");
    }
    else if(!strcmp(arg_tmp,">")){
        detail->output=1;
        *locate='\0';
        cmdline=locate+1;
        while (*cmdline && (*cmdline==' '))cmdline++;
        locate=strchr(cmdline,' ');
        if(!locate){
            printf("Where is the file?!");
            return;
```

```
            }
            detail->output_file=strtok(cmdline," ");
        }
        else if(!strcmp(arg_tmp,">>")){
            detail->output=2;
            *locate='\0';
            cmdline=locate+1;
            while(*cmdline && (*cmdline==' '))cmdline++;
            locate=strchr(cmdline,' ');
            if(!locate){
                printf("Where is the file?!");
                return;
            }
            detail->output_file=strtok(cmdline," ");
        }
        else if(!strcmp(arg_tmp,"|")){
            ++argc;
            argv[cnt1][argc]=NULL;
            ++cnt1;
            argc=0;
        }
        else{
            argv[cnt1][argc]=arg_tmp;
            argc++;
        }
```

最后不要忘了一些特殊的情况和值。

```
    argv[cnt1][argc]=NULL;
    detail->pipenum=cnt1+1;

    if(argc == 0){
        //如果最后一个命令压根就没有参数
        //如果不加这个，会出错，因为后面的 else if 判断可能越界
        detail->bg = 0;
    }
    else if(!strcmp(argv[cnt1][argc-1],"&"))detail->bg=1;


    if (detail->bg==1){
        argc--;
        argv[cnt1][argc] = NULL;
    }
```

结合上述的一步一步操作，我得到了自认为还算完善的 parseline()函数

```
void parseline(char *cmdline,char *argv[MAXPIPE][MAXARG],struct Detail *detail){
    //parseline 读取 cmdline 然后把结果放到 argv 和 detail 中
    char *locate;//用于定位
    int argc; //一个命令后面所接参数数量
    int cnt1; //目前记录了的命令数量
```

```
    cmdline[strlen(cmdline)-1] = ' '; //把最后一个字符'\n'置为' '
    while (*cmdline&&(*cmdline ==' '))cmdline++; //丢弃第一个指令前多余的空格

    detail->input=0;detail->output=0;detail->bg=0;
    cnt1=0;argc=0;
    locate=strchr(cmdline,' ');
    while (locate) {
        char *arg_tmp=strtok(cmdline," ");
        if(!strcmp(arg_tmp, "<")){
            detail->input= 1;
            *locate='\0';
            cmdline=locate+1;
            while (*cmdline && (*cmdline ==' '))cmdline++;
            locate=strchr(cmdline,' ');
            if(!locate){
                printf("Where is the file?!");
                return;
            }
            detail->input_file=strtok(cmdline," ");
        }
        else if(!strcmp(arg_tmp,">")){
            detail->output= 1;
            *locate='\0';
            cmdline=locate+1;
            while (*cmdline && (*cmdline==' '))cmdline++;
            locate=strchr(cmdline,' ');
            if(!locate){
                printf("Where is the file?!");
                return;
            }
            detail->output_file=strtok(cmdline," ");
        }
        else if(!strcmp(arg_tmp, ">>")){
            detail->output=2;
            *locate='\0';
            cmdline=locate + 1;
            while(*cmdline && (*cmdline == ' '))cmdline++;
            locate=strchr(cmdline,' ');
            if(!locate){
                printf("Where is the file?!");
                return;
            }
            detail->output_file=strtok(cmdline," ");
        }
        else if(!strcmp(arg_tmp, "|")){
            ++argc;
            argv[cnt1][argc]=NULL;
            ++cnt1;
```

```
            argc=0;
        }
        else{
            argv[cnt1][argc]=arg_tmp;
            argc++;
        }
        *locate='\0';
        cmdline=locate+1;
        while (*cmdline && (*cmdline == ' '))cmdline++;
        locate=strchr(cmdline,' ');
    }
    argv[cnt1][argc]=NULL;
    detail->pipenum=cnt1+1;

    if(argc == 0){
        //如果最后一个命令压根就没有参数
        //如果不加这个，会出错，因为后面的 else if 判断可能越界
        detail->bg = 0;
    }
    else if(!strcmp(argv[cnt1][argc-1],"&"))detail->bg=1;


    if (detail->bg==1){
        argc--;
        argv[cnt1][argc] = NULL;
    }
}
```

2.3 完成判断输入命令是否为内置命令的 builtin_cmd(char **argv),若是便执行

```
int builtin_cmd(char **argv);
```

这个函数的完成还是比较简单的，就一直 strcmp 就行(C 标准库中的函数真是帮我大忙)注意 chdir()可以移动 shell 的工作目录。mytop()等主体写完了再写吧！

```
int builtin_cmd(char **argv){
    if(!strcmp(argv[0],"cd")){
        if(chdir(argv[1])<0){
            printf("Error occurs when cd!\n");
        }
        return 1;
    }
    else if(!strcmp(argv[0],"history")){
        int n;
        if(!argv[1])n=cnt-1;
        else n=atoi(argv[1]);

        if(n<cnt){
            for(int i=cnt-n;i<cnt;i++)fprintf(stdout,"%s\n",cmd_history[i]);
        }else{
            printf("Error occurs when history\n");
        }
```

```
        return 1;
    }
    else if(!strcmp(argv[0],"mytop")){
        //too complex!
        mytop();
        return 1;
    }
    else if(!strcmp(argv[0],"exit"))exit(0);

    return 0;
}
```

2.4 完成执行一行命令的 eval(char *cmdline)函数

```
void eval(char *cmdline);
```

初始化阶段，为了后续的实验，我预先定义了一个二维数组 pipegate,其中 pipegate[i][0]和 pipegate[i][1]分别代表第 i 个进程的读端和写端。

```
    char *argv[MAXPIPE][MAXARG];struct Detail detail; pid_t pid;
    parseline(cmdline,argv,&detail);//解析
    int pipegate[MAXPIPE][2];
```

2.4.1 eval 执行的大框架以及后台的信号处理

```
    if(!builtin_cmd(argv[0])){
        if(detail.bg)signal(SIGCHLD, SIG_IGN);
        else signal(SIGCHLD, SIG_DFL);
        /*
        这里要启动各个命令了
        */
    }
```

在输入的指令最后是&时，表示这个任务流是后台执行的，对于父进程，我们要让它不需要等待子进程结束，即面对子进程结束时到来的 SIG_CHLD 信号选择忽略(SIG_IGN)，使得 minix 接管此进程,否则就默认情况(SIG_DFL)杀死子进程。

2.4.2 对于后台执行的指令之具体操作

根据 2.4.1,子进程结束后会成为僵尸进程占用系统资源，所以为了屏蔽键盘和控制台(防止子进程受到终端输入输出的影响)，子进程的标准输入、输出映射成/dev/null

```
                if(detail.bg){
                    int fd=open("/dev/null",O_RDWR);
                    dup2(fd,STDIN_FILENO);
                    dup2(fd,STDOUT_FILENO);
                }
```

这里需要着重讲一下 dup2()和 dup()以及 close()这三个函数。

文件描述符是一个非负整数。可以理解成一个标志，对应的是相应的文件指针。当打开或者创建一个文件时，内核会向进程返回一个文件描述符。原子操作可以理解为无法被细分的操作，这样可以不会出现一段程序中父子进程之间信号混乱使得操作失误之问题。

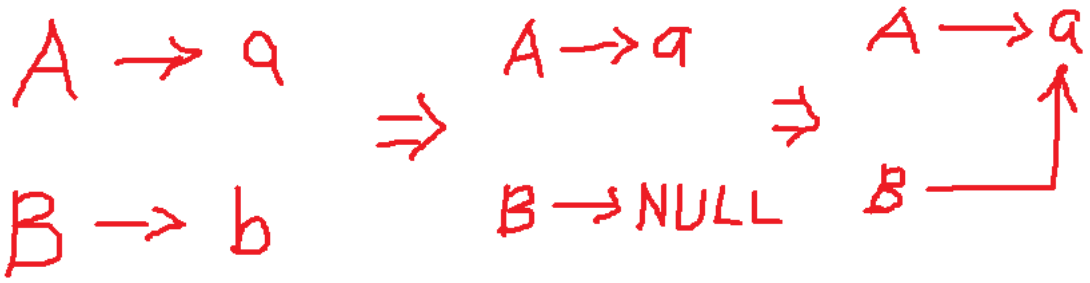
dup(int fd)函数复制现有的文件描述符 fd，返回的新文件描述符是当前可用文件描述符中的最小数值，讲真，这里我最初不是特别理解，如果是“最小”，那怎样确保我们复制的文件描述符对应于我们的需求呢？

close(int fd)函数关闭一个打开的文件，这样对应这个文件的文件描述符就解放了出来。

dup2(int fd，int fd2)函数是一个原子操作，可以较好地用 fd2 参数指定新描述符的值。

根据《UNIX 高级编程》的介绍，一个 dup(fd)等价于 fcntl(fd,F_DUPFD,0) 而一个 dup2(fd,fd2)等价于 close(fd2)和 fcntl(fd,F_DUPFD,fd2) 虽然效果上是类似的，但是 dup2 很清晰、安全且指定了新的文件描述符，所以我还是在本次实验中我大量地使用了它。

下面是我对 dup2()函数的理解，假设 A,B 是两个文件描述符，a,b 是两个文件。这张图展示了 dup2(A,B)的效果



2.4.3 对于无管道情况的实现

先从最简单的没有管道情况开始吧，这里我需要判断输入输出的情况，对于重定向的> < 和>>要有对应的操作。open(const char *path, int mode)函数以对应的模式打开了文件，我在此把本实验中用到的模式给列举一下吧。

O_RDONLY 只读打开 O_WRONLY 只写打开 O_RDWR 读、写打开

O_CREAT 此文件不存在便创建它 O_APPEND 每次写时都追加到文件尾端

```
        //如果没管道
        if((pid=fork())==0){
            if(detail.input==1){
                int fd=open(detail.input_file,O_RDONLY);
                dup2(fd,STDIN_FILENO);
            }

            if(detail.output==1){
                int fd=open(detail.output_file,O_WRONLY|O_CREAT);
                dup2(fd,STDOUT_FILENO);
            }
            else if(detail.output==2){
                int fd=open(detail.output_file,O_WRONLY|O_APPEND);
                dup2(fd,STDOUT_FILENO);
            }

            if(detail.bg){
                int fd=open("/dev/null",O_RDWR);
                dup2(fd,STDIN_FILENO);
                dup2(fd,STDOUT_FILENO);
            }


            if (execvp(argv[0][0],argv[0]) < 0) {
                fprintf(stdout,"%s: Program not found.\n",argv[0][0]);
                exit(0);
            }
```

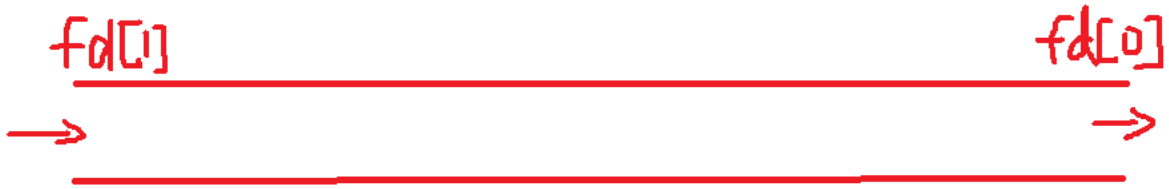
```
        }else{
            if(detail.bg==0){
                int state;
                waitpid(pid, &state, 0);
            }
        }
```

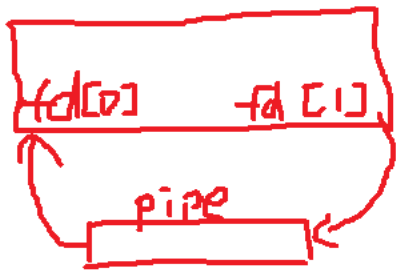
2.4.4 对于多重管道的实现

这应该是本实验中那个最难的部分了吧。其实本质内容还是比较简单的，就是根据parseline()获得的函数，一个一个地运行指令。在这里我有必要详细说一下自己对管道的理解(虽然不一定严谨，但是应该是正确的)。

管道一般是半双工的，即数据只能在一个方向上流动。而且管道只能在具有公共祖先的两个进程之间使用，在这个实验中，我让各个小的指令在一个进程，它们都是兄弟进程，这样便可以实现它们之间的通讯。

pipe(int fd[2])函数创建了管道，执行后 fd 拥有了两个文件描述符 fd[0]读端(只读不写)和 fd[1]写端(只写不读)。在这里发挥了我充分的想象力，就是在 fd[1]端放上了数据，那么管道自动地就把数据移动到了 fd[0],这样其他进程就可以从 fd[0]来获取数据啦。





为了防止保持管道的半双工特性，为了不让管道交互的情况出现，我发现一般就把不需要的那端管道给关闭了，这是设计程序的一个原则。

好的，还有一件事，就是我执行那 pipenum 个指令时，是一个又一个执行的，并且 pipe 创建管道也是随着循环一个又一个创建的。在执行到第 i 个指令时，我们只能控制之前的进程的管道(因为之后的管道压根还没创建呢！)我们怎样设置能让多重管道得以实现？

对于一个指令，本来应该是从 STDIN 输入，然后从 STDOUT 输出。对于一堆管道连接的指令其中的某个指令(先考虑一般情况，不考虑首尾两端)，根据上一段的描述，我们只能控制当前进程的管道和之前进程的描述。我要干两件事:

1. 将上一个进程的输出拿过来取代 STDIN 作为输入
2. 将当前进程的输出取代 STDOUT 作为下一个进程的输入

要完成第一件事，我们要执行读取上一个进程的结果，上一个进程在干它的第二件事已经把它的结果准备好了，只需要当前进程到 pipegate[i-1][0]获取即可，这之后顺便要关闭 pipegate[i-1][1]

```
                    dup2(pipegate[i-1][0],STDIN_FILENO);
                    close(pipegate[i-1][1]);
```

要完成第二件事，我们为了给下一个进程准备结果，只需要把我们的结果放到 pipegate[i][1]即可，这之后顺便要关闭 pipegate[i][0]

```
                    dup2(pipegate[i][1],STDOUT_FILENO);
                    close(pipegate[i][0]);
```

好的，那对于首尾两端的那两个指令怎么办？

首端是从 STDIN 读入，而尾端是从 STDOUT 输出。注意！这里还需要考虑一些特殊的情况，比如后台(尾部考虑)，重定向输入(首部考虑)，重定向输出(尾部考虑)所以代码如下

首部

```
                if(i==0){
                    if(detail.input==1){
                        int fd=open(detail.input_file,O_RDONLY);
                        dup2(fd,STDIN_FILENO);
                    }
                    dup2(pipegate[i][1],STDOUT_FILENO);
                    close(pipegate[i][0]);
                }
```

尾部

```
                else if (i==detail.pipenum-1){
                    if(detail.output==1){
                        int
fd=open(detail.output_file,O_CREAT|O_WRONLY|O_TRUNC,S_IWRITE|S_IREAD);
                        dup2(fd,STDOUT_FILENO);
                    }
                    else if(detail.output== 2){
                        int
fd=open(detail.output_file,O_CREAT|O_WRONLY|O_APPEND,S_IWRITE|S_IREAD);
                        dup2(fd,STDOUT_FILENO);
                    }
                    dup2(pipegate[i-1][0],STDIN_FILENO);
                    close(pipegate[i-1][1]);

                    if(detail.bg){
                        int fd=open("/dev/null",O_RDWR);
                        dup2(fd,STDIN_FILENO);
                        dup2(fd,STDOUT_FILENO);
                    }
```

```
                    }
```

一般情况

```
                else{
                    dup2(pipegate[i-1][0],STDIN_FILENO);
                    close(pipegate[i-1][1]);
                    dup2(pipegate[i][1],STDOUT_FILENO);
                    close(pipegate[i][0]);
                }
```

在这些判定后，我们终于让子进程可以执行关键的 exec 来执行命令了(为什么要让子进程来执行？这也是一个关键的问题。exec 执行后，会把当前进程用指定的可执行文件给覆盖掉，如果是父进程执行，我们怎么依次执行后面的指令呢？这显然是不可接受的)

还有一个问题，对那 pipenum 个执行用的进程，他们的顺序是怎样的？显然是一个完成了之后才能开始下一个，但是我们怎样实现这个呢？不是说了如果是后台就不 waitpid 了吗？这里我们不管后台还是前台，都要让父进程(执行第 i 个指令的进程的父进程)等待其真的结束才能接着开始第 i+1 个指令。

```
                if((pid=fork())==0){
                    //主体代码区段
                    if(execvp(argv[i][0],argv[i])<0) {
                        fprintf(stdout,"%s: Program not found.\n",argv[i][0]);
                        exit(0);
                    }
                }else{
                    close(pipegate[i][1]);
                    int state;
                    waitpid(pid,&state,0);
                }
```

还有就是在最后别忘了关闭 pipegate[i][1]防止出现管道交互的情况。以及，别忘了 exit(0)!!!!!!这里自己遇到了一些离谱的错误

综合上述的分析和思考，我终于完成了 eval 的功能，下面是 eval 的代码(有些冗长了，中间那个大的 if 和 for 控制语句堆真的需要耐心来调试)

```
void eval(char *cmdline){
    char *argv[MAXPIPE][MAXARG];struct Detail detail; pid_t pid;
    parseline(cmdline,argv,&detail);//解析
    int pipegate[MAXPIPE][2];


    if(!builtin_cmd(argv[0])){
        if(detail.bg)signal(SIGCHLD, SIG_IGN);
        else signal(SIGCHLD, SIG_DFL);

        if(detail.pipenum>1){
            //如果有管道
            if((pid=fork())==0){
                for(int i=0;i<detail. pipenum;i++){
                    pipe(pipegate[i]);
                    if((pid=fork())==0){
                        if(i==0){
                            if(detail.input==1){
```

```
                            int fd=open(detail.input_file,O_RDONLY);
                            dup2(fd,STDIN_FILENO);
                        }
                        dup2(pipegate[i][1],STDOUT_FILENO);
                        close(pipegate[i][0]);
                    }
                    else if (i==detail.pipenum-1){
                        if(detail.output==1){
                            int
fd=open(detail.output_file,O_CREAT|O_WRONLY|O_TRUNC,S_IWRITE|S_IREAD);
                            dup2(fd,STDOUT_FILENO);
                        }
                        else if(detail.output== 2){
                            int
fd=open(detail.output_file,O_CREAT|O_WRONLY|O_APPEND,S_IWRITE|S_IREAD);
                            dup2(fd,STDOUT_FILENO);
                        }
                        dup2(pipegate[i-1][0],STDIN_FILENO);
                        close(pipegate[i-1][1]);

                        if(detail.bg){
                            int fd=open("/dev/null",O_RDWR);
                            dup2(fd,STDIN_FILENO);
                            dup2(fd,STDOUT_FILENO);
                        }
                    }
                    else{
                        dup2(pipegate[i-1][0],STDIN_FILENO);
                        close(pipegate[i-1][1]);
                        dup2(pipegate[i][1],STDOUT_FILENO);
                        close(pipegate[i][0]);
                    }

                    if(execvp(argv[i][0],argv[i])<0) {
                        fprintf(stdout,"%s: Program not found.\n",argv[i][0]);
                        exit(0);
                    }

                }else{
                    close(pipegate[i][1]);
                    int state;
                    waitpid(pid,&state,0);
                }
            }
            exit(0);
        }else{
            if(detail.bg==0){
                int state;
```

```
                    waitpid(pid, &state, 0);
                }
            }
        }
        else{
            //如果没管道
            if((pid=fork())==0){
                if(detail.input==1){
                    int fd=open(detail.input_file,O_RDONLY);
                    dup2(fd,STDIN_FILENO);
                }

                if(detail.output==1){
                    int fd=open(detail.output_file,O_WRONLY|O_CREAT);
                    dup2(fd,STDOUT_FILENO);
                }
                else if(detail.output==2){
                    int fd=open(detail.output_file,O_WRONLY|O_APPEND);
                    dup2(fd,STDOUT_FILENO);
                }

                if(detail.bg){
                    int fd=open("/dev/null",O_RDWR);
                    dup2(fd,STDIN_FILENO);
                    dup2(fd,STDOUT_FILENO);
                }


                if (execvp(argv[0][0],argv[0]) < 0) {
                    fprintf(stdout,"%s: Program not found.\n",argv[0][0]);
                    exit(0);
                }
            }else{
                if(detail.bg==0){
                    int state;
                    waitpid(pid, &state, 0);
                }
            }
        }
    }
    return;
}
```

2.5 程序运行统计 mytop()的实现

这个函数的实现也是十分重量级，不过多亏有老师给出的参考代码(https://github.com/0xffea/MINIX3/blob/master/usr.bin/top/top.c) 作为参考，所以主要还是读懂这个 top.c 在干什么，怎样调用

2.5.1 相关的宏定义、全局变量和结构体

```
//开始 mytop 相关的宏定义
const char *cputimenames[]={ "user", "ipc","kernelcall" }; //CPU cycle types
#define CPUTIMENAMES (sizeof(cputimenames)/sizeof(cputimenames[0]))
#define MAXBUF 1024 //Max buffer size for an file read operation

#define CPUTIME(m, i) (m & (1L << (i)))
#define USED 0x1
#define IS_TASK 0x2
#define IS_SYSTEM 0x4
#define BLOCKED 0x8

//进程类型
#define TYPE_TASK 'T'
#define TYPE_SYSTEM 'S'
#define TYPE_USER 'U'

#define PROC_NAME_LEN 16
```

```
struct proc{
    int p_flags;
    int p_endpoint;
    pid_t p_pid;
    uint64_t p_cpucycles[CPUTIMENAMES];
    int p_priority;
    int p_blocked;
    time_t p_user_time;
    long unsigned int p_memory;
    uid_t p_effuid;
    int p_nice;
    char p_name[PROC_NAME_LEN+1];
};
struct tp {
    struct proc* p;
    u64_t ticks;
};
int slot;
unsigned int nr_procs, nr_tasks;
int nr_total; //Number of process + task
struct proc *proc = NULL, *prev_proc = NULL;
// 由于每个进程中保存的信息是启动时到当前时间的统计信息
// 因此为了获取即时的占用信息，需要在极短的时间内对每个进程信息文件读取两次
// 并计算差值来获得即时的占用信息
```

```
//打印内存和 CPU 信息模块的函数名
void mytop();
void get_memory(int *total_size, int *free_size, int *cached_size);
void getkinfo();
void parse_file(pid_t pid);
void parse_dir();
```

```
void get_procs();
uint64_t make_cycle(unsigned long lo, unsigned long hi);
uint64_t cputicks(struct proc *p1, struct proc *p2, int timemode);
float print_procs(struct proc *proc1, struct proc *proc2, int cputimemode);
```

2.5.2 mytop()主体思路

/proc/meminfo 中， 查看内存信息，每个参数对应含义依次是页面大小 pagesize，总页数量 total ， 空闲页数量 free ，最大页数量 largest ，缓存页数量 cached 。可计算内存大小：(pagesize * total)/1024，同理算出其他页内存大小。

/proc/kinfo 中，查看进程和任务数量。

/proc/pid/psinfo 中，例如 /proc/107/psinfo 文件中，查看 pid 为 107 的进程信息。每个参数对应含义依次是：版本 version，类型 type，端点 endpt，名字 name，状态 state，阻塞状态 blocked，动态优先级 priority，滴答 ticks，高周期 highcycle，低周期 lowcycle，内存 memory，有效用户 ID effuid，静态优先级 nice 等。其中会用到的参数有：类型，状态，滴答。进程时间 time=ticks/ (u32_t)60。

总体 CPU 使用占比。计算方法：得到进程和任务总数量 total_proc，对每一个 proc 的 ticks 累加得到总体 ticks，再计算空闲的 ticks，最终可得到 CPU 使用百分比。

并且一个进程的 CPU 占用时间可以从两次读取到的 CPU 周期之差得到，不过需要注意的是，MINIX 进程信息文件中保存的分别是周期的高 32 位及低 32 位，我们需要首先将他们拼接成一个完整的 64 位整数才能进行后续的计算

2.5.3 mytop 实现中 CPU 使用百分比思路

1. 计算进程和任务总数 nr_total，读取/proc/kinfo
2. 读取每个进程的信息，包括 type，endpoint，pid，cycles_hi,cycles_lo，state。遍历 /proc/pid/psinfo

   如果 type==task，则进程标记 p_flags |=istask

   如果 type==system，则进程标记 p_flags |=issystem

   如果 state！=state_run,则标记 p_flags |=blocked

   连续读 CPUTIMENAMES 次 cycles_hi,cycle_lo（三个 cputimenames），然后拼接成 64 位，放在 p_cpucycles[]数组中。

```
uint64_t make_cycle(unsigned long lo, unsigned long hi){
    // MINIX 进程信息文件中保存的分别是周期的高 32 位及低 32 位
    // 我们需要首先将他们拼接成一个完整的 64 位整数才能进行后续的计算
    return ((uint64_t)hi << 32) | (uint64_t)lo;
}
```

3. 计算每个进程 proc 的滴答，通过 proc 和当前进程 prev_proc 做比较，如果 endpoint 相等，则在循环中分别计算

```
    for(i = 0; i < CPUTIMENAMES; i++) {
        if(!CPUTIME(timemode, i))continue;
        if(p1->p_endpoint == p2->p_endpoint) {
            t = t + p2->p_cpucycles[i] - p1->p_cpucycles[i];
        } else {
            t = t + p2->p_cpucycles[i];
        }
    }
```

4. 计算总的 cpu 使用百分比，遍历所有的进程和任务，判断类型，计算 systemticks，userticks。

```
        if(!(proc2[p].p_flags & IS_TASK)) {
            if(proc2[p].p_flags & IS_SYSTEM)systemticks = systemticks +
tick_procs[nprocs].ticks;
            else userticks = userticks + tick_procs[nprocs].ticks;
        }
```

2.5.4 需要用到的函数和它们的效果

1. get_procs()，首先记录当前进程，赋值给 prev_proc，然后通过 parse_dir()函数获取到/proc/下的所有进程 pid,再通过 parse_file()函数获取每一个进程信息，即读取/proc/pid/psinfo 文件。
2. parse_file() 遍历/proc/pid/psinfo 读取每个进程的信息
3. cputicks()，计算每个进程的滴答。滴答并不是简单的结构体中的滴答，因为在写文件的时候需要更新。需要通过当前进程来和该进程一起计算，这里需要用到p_cpucycles.
4. print_procs(),输出CPU使用时间。这里创建了一个tp结构体的数组tick_procs。对所有的进程和任务(即上面读出来的nr_total)计算ticks。把kernelticks，userticks，systemticks相加就可以得到CPU的使用百分比。
5. get_memory()打开/proc/meminfo获取内存使用情况。该文件中有 5 个参数，对应页面大小、总页数量、空闲页数量、最大页数量、缓存页数量。
6. getkinfo()打开/proc/kinfo 获得进程相关的数据
7. parse_dir()获取到/proc/下进程的 pid

2.5.5 mytop()主体函数的实现

在知道各个函数大概是有什么效果之后，mytop()还是比较轻易地完成的。

```
void mytop(){
    int total_size,free_size,cached_size; //主存信息
    get_memory(&total_size,&free_size,&cached_size);
    fprintf(stdout, "Total: %dK, Free: %dK, Cached: %dK\n", total_size, free_size,
cached_size);
    getkinfo(); //读取/proc/kinfo 计算进程和任务总数 nr_total
    get_procs(); //读取每个进程的信息
    if (prev_proc == NULL)get_procs();
    float idle = print_procs(prev_proc, proc, 1); //输出 CPU 使用时间
    fprintf(stdout, "CPU Usage: %f%%\n", idle);
}
```

2.5.6 引用 top.c 中需要的代码

```
void get_memory(int *total_size, int *free_size, int *cached_size){
    int mem_f; //File descriptor of memory info
    int bufsize; //Actual size of bytes read
    char buf[MAXBUF]; //Buffer for reading from file
    int page_size, total_page, free_page, largest_page, cached_page;

    mem_f = open("/proc/meminfo", O_RDONLY); //Open memory info file
    bufsize = read(mem_f, buf, sizeof(buf)); //Read memory info
    if(bufsize == -1){
        printf("Error reading memory info!");
```

```
    }
    else{
        page_size = atoi(strtok(buf, " "));
        total_page = atoi(strtok(NULL, " "));
        free_page = atoi(strtok(NULL, " "));
        largest_page = atoi(strtok(NULL, " "));
        cached_page = atoi(strtok(NULL, " "));
        *total_size = (page_size * total_page) / 1024;
        *free_size = (page_size * free_page) / 1024;
        *cached_size = (page_size * cached_page) / 1024;
    }
}

void getkinfo(){
    int fd; //File descriptor of kinfo
    int bufsize; //Actual buffer size
    char buf[MAXBUF], pathbuf[MAXBUF]; //Buffer for file reading

    fd=open("/proc/kinfo", O_RDONLY);
    if (fd == -1) {
        printf("Reading kinfo file error!");
        exit(1);
    }
    bufsize = read(fd, buf, sizeof(buf)); //Read process info
    if(bufsize == -1){
        printf("Error reading total process info!");
    }
    else {
        nr_procs = (unsigned int) atoi(strtok(buf, " ")); //Number of process
        nr_tasks = (unsigned int) atoi(strtok(NULL, " ")); //Number of tasks
        close(fd);
        nr_total = (int) (nr_procs + nr_tasks);
    }
}

void parse_file(pid_t pid)  {
    // 遍历/proc/pid/psinfo  读取每个进程的信息
    /*
        如果 type==task，则进程标记 p_flags |=istask
        如果 type==system，则进程标记 p_flags |=issystem
        如果 state!=state_run,则标记 p_flags |=blocked
    */

    /*
        再连续读 CPUTIMENAMES 次 cycles_hi,cycle_lo（三个 cputimenames）
        然后拼接成 64 位，放在 p_cpucycles[]数组中。
    */
```

```
    char path[MAXBUF], name[256], type, state;
    int version, endpt, effuid;
    unsigned long cycles_hi, cycles_lo;
    FILE *fp;
    struct proc *p;
    int i;

    sprintf(path, "/proc/%d/psinfo", pid);

    if ((fp = fopen(path, "r")) == NULL)
        return;

    if (fscanf(fp, "%d", &version) != 1) {
        fclose(fp);
        return;
    }

    if (fscanf(fp, " %c %d", &type, &endpt) != 2) {
        fclose(fp);
        return;
    }

    slot++;

    if(slot < 0 || slot >= nr_total) {
        fprintf(stderr, "Unreasonable endpoint number %d\n", endpt);
        fclose(fp);
        return;
    }

    p = &proc[slot];

    if (type == TYPE_TASK)
        p->p_flags |= IS_TASK;
    else if (type == TYPE_SYSTEM)
        p->p_flags |= IS_SYSTEM;

    p->p_endpoint = endpt;
    p->p_pid = pid;

    if (fscanf(fp, " %255s %c %d %d %lu %*u %lu %lu",
               name, &state, &p->p_blocked, &p->p_priority,
               &p->p_user_time, &cycles_hi, &cycles_lo) != 7) {

        fclose(fp);
        return;
    }
```

```
    strncpy(p->p_name, name, sizeof(p->p_name)-1);
    p->p_name[sizeof(p->p_name)-1] = 0;

    if (state != STATE_RUN)
        p->p_flags |= BLOCKED;
    p->p_cpucycles[0] = make_cycle(cycles_lo, cycles_hi);
    p->p_memory = 0L;

    if (!(p->p_flags & IS_TASK)) {
        int j;
        if ((j=fscanf(fp, " %lu %*u %*u %*c %*d %*u %u %*u %d %*c %*d %*u",
                    &p->p_memory, &effuid, &p->p_nice)) != 3) {

            fclose(fp);
            return;
        }

        p->p_effuid = effuid;
    } else p->p_effuid = 0;


    for(i = 1; i < CPUTIMENAMES; i++) {
        if(fscanf(fp, " %lu %lu",
                  &cycles_hi, &cycles_lo) == 2) {
            p->p_cpucycles[i] = make_cycle(cycles_lo, cycles_hi); //拼接成 64 位使得后
续计算可以正常进行
        } else  {
            p->p_cpucycles[i] = 0;
        }
    }

    if ((p->p_flags & IS_TASK)) {
        if(fscanf(fp, " %lu", &p->p_memory) != 1) {
            p->p_memory = 0;
        }
    }

    p->p_flags |= USED;

    fclose(fp);
}

void parse_dir(){
    DIR *p_dir; //Pointer of directory
    struct dirent *p_ent; //Info of the directory
    pid_t pid; //Name of sub directory(PID)
    char *end;
```

```
    if ((p_dir = opendir("/proc")) == NULL) {
        exit(1);
    }

    //Traverse the directory
    for (p_ent = readdir(p_dir); p_ent != NULL; p_ent = readdir(p_dir)) {
        pid = strtol(p_ent->d_name, &end, 10); //Get the name of sub directory
        if (!end[0] && pid != 0)parse_file(pid);
    }
    closedir(p_dir);
}

void get_procs(){
    struct proc *p;
    int i;
    p = prev_proc;
    prev_proc = proc;
    proc = p;

    if (proc == NULL) {
        proc = malloc(nr_total * sizeof(proc[0])); //Allocate a new process
structure
        //Allocate failed
        if (proc == NULL) {
            fprintf(stderr, "Out of memory!\n");
            exit(1);
        }
    }
    //Initialize all the entry ranging in the total process+task num
    for (i = 0; i < nr_total; i++)proc[i].p_flags = 0;
    parse_dir();
}

uint64_t make_cycle(unsigned long lo, unsigned long hi){
    // MINIX 进程信息文件中保存的分别是周期的高 32 位及低 32 位
    // 我们需要首先将他们拼接成一个完整的 64 位整数才能进行后续的计算
    return ((uint64_t)hi << 32) | (uint64_t)lo;
}

uint64_t cputicks(struct proc *p1, struct proc *p2, int timemode){
    int i;
    uint64_t t = 0;

    // 计算每个进程 proc 的滴答，通过 proc 和当前进程 prev_proc 做比较
    // 如果 endpoint 相等，则在循环中分别计算
    for(i = 0; i < CPUTIMENAMES; i++) {
        if(!CPUTIME(timemode, i))continue;
        if(p1->p_endpoint == p2->p_endpoint) {
```

```
            t = t + p2->p_cpucycles[i] - p1->p_cpucycles[i];
        } else {
            t = t + p2->p_cpucycles[i];
        }
    }
    return t;
}


float print_procs(struct proc *proc1, struct proc *proc2, int cputimemode){
    int p, nprocs;
    uint64_t systemticks = 0;
    uint64_t userticks = 0;
    uint64_t total_ticks = 0;
    static struct tp *tick_procs = NULL;

    if (tick_procs == NULL) {
        tick_procs = malloc(nr_total * sizeof(tick_procs[0]));
        if (tick_procs == NULL) {
            fprintf(stderr, "Out of memory!\n");
            exit(1);
        }
    }

    for (p = nprocs = 0; p < nr_total; p++) {
        uint64_t uticks;
        if (!(proc2[p].p_flags & USED))continue;
        tick_procs[nprocs].p = proc2 + p;
        tick_procs[nprocs].ticks = cputicks(&proc1[p], &proc2[p], cputimemode);
        uticks = cputicks(&proc1[p], &proc2[p], 1);
        total_ticks = total_ticks + uticks;
        // 计算总的 cpu 使用百分比，遍历所有的进程和任务
        // 计算 systemticks，userticks
        if(!(proc2[p].p_flags & IS_TASK)) {
            if(proc2[p].p_flags & IS_SYSTEM)systemticks = systemticks +
tick_procs[nprocs].ticks;
            else userticks = userticks + tick_procs[nprocs].ticks;
        }
        nprocs++;
    }

    if (total_ticks == 0)return 0.0;
    return 100.0 * (systemticks + userticks) / total_ticks;
}
```

3. 完整代码

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <curses.h>
#include <limits.h>
#include <termcap.h>
#include <termios.h>
#include <time.h>
#include <assert.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <grp.h>
#include <pwd.h>
#include <dirent.h>

#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/times.h>
#include <sys/time.h>
#include <sys/select.h>

#include <minix/com.h>
#include <minix/config.h>
#include <minix/type.h>
#include <minix/endpoint.h>
#include <minix/const.h>
#include <minix/u64.h>
#include <paths.h>
#include <minix/procfs.h>

//开始宏定义
#define MAXCMD 1024 //最大记录的命令行数
#define MAXLINE 1024 //每行最大长度

#define MAXARG 64 //命令行中某命令最大参数数量
#define MAXPIPE 64 //通过管道连接的最大程序数量

//开始 mytop 相关的宏定义
const char *cputimenames[]={ "user", "ipc","kernelcall" }; //CPU cycle types
#define CPUTIMENAMES (sizeof(cputimenames)/sizeof(cputimenames[0]))
```

```
#define MAXBUF 1024 //Max buffer size for an file read operation

#define CPUTIME(m, i) (m & (1L << (i)))
#define USED 0x1
#define IS_TASK 0x2
#define IS_SYSTEM 0x4
#define BLOCKED 0x8

//进程类型
#define TYPE_TASK 'T'
#define TYPE_SYSTEM 'S'
#define TYPE_USER 'U'

#define PROC_NAME_LEN 16

//全局变量和数据类型
int cnt=0;//当前读到了第几条命令
char cmd_history[MAXCMD][MAXLINE];
struct Detail{
    //这个结构体是在记录每一行的详细信息
    int bg;
    int pipenum; //通过管道连接的程序的数量
    int input,output; //输入输出模式
    //input 0 标准输入 1 文件输入
    //output 0 标准输出 1 文件输出(覆盖) 2 文件输出(追加)
    char *input_file;//输入文件地址
    char *output_file;//输出文件地址
};


struct proc{
    int p_flags;
    int p_endpoint;
    pid_t p_pid;
    uint64_t p_cpucycles[CPUTIMENAMES];
    int p_priority;
    int p_blocked;
    time_t p_user_time;
    long unsigned int p_memory;
    uid_t p_effuid;
    int p_nice;
    char p_name[PROC_NAME_LEN+1];
};
struct tp {
    struct proc* p;
    u64_t ticks;
};
int slot;
unsigned int nr_procs, nr_tasks;
```

```
int nr_total; //Number of process + task
struct proc *proc = NULL, *prev_proc = NULL;
// 由于每个进程中保存的信息是启动时到当前时间的统计信息
// 因此为了获取即时的占用信息，需要在极短的时间内对每个进程信息文件读取两次
// 并计算差值来获得即时的占用信息

//开始各种函数名
void eval(char *cmdline);
void parseline(char *cmdline,char *argv[MAXPIPE][MAXARG],struct Detail *detail);
int builtin_cmd(char **argv);



//打印内存和 CPU 信息模块的函数名
void mytop();
void get_memory(int *total_size, int *free_size, int *cached_size);
void getkinfo();
void parse_file(pid_t pid);
void parse_dir();
void get_procs();
uint64_t make_cycle(unsigned long lo, unsigned long hi);
uint64_t cputicks(struct proc *p1, struct proc *p2, int timemode);
float print_procs(struct proc *proc1, struct proc *proc2, int cputimemode);




int main(int argc,char **argv){
    char cmdline[MAXLINE];
    fprintf(stdout,"Hello there!I'm tommy and this is the first project of OS in
ECNU DASE\n");
    while(1){
        printf("tommyshell  %s >",getcwd(NULL,NULL));
        if(fgets(cmdline,MAXLINE,stdin)==NULL){
            //通过标准输入将命令行读入到 cmdline 中
            printf("Error occurs when input\n");
            continue;
        }
        strcpy(cmd_history[cnt],cmdline); //备份
        ++cnt;
        eval(cmdline);
    }
    return 0;
}

void eval(char *cmdline){
    char *argv[MAXPIPE][MAXARG];struct Detail detail; pid_t pid;
    parseline(cmdline,argv,&detail);//解析
    int pipegate[MAXPIPE][2];
```

```
    if(!builtin_cmd(argv[0])){
        if(detail.bg)signal(SIGCHLD, SIG_IGN);
        else signal(SIGCHLD, SIG_DFL);

        if(detail.pipenum>1){
            //如果有管道
            if((pid=fork())==0){
                for(int i=0;i<detail. pipenum;i++){
                    pipe(pipegate[i]);
                    if((pid=fork())==0){
                        if(i==0){
                            if(detail.input==1){
                                int fd=open(detail.input_file,O_RDONLY);
                                dup2(fd,STDIN_FILENO);
                            }
                            dup2(pipegate[i][1],STDOUT_FILENO);
                            close(pipegate[i][0]);
                        }
                        else if (i==detail.pipenum-1){
                            if(detail.output==1){
                                int
fd=open(detail.output_file,O_CREAT|O_WRONLY|O_TRUNC,S_IWRITE|S_IREAD);
                                dup2(fd,STDOUT_FILENO);
                            }
                            else if(detail.output== 2){
                                int
fd=open(detail.output_file,O_CREAT|O_WRONLY|O_APPEND,S_IWRITE|S_IREAD);
                                dup2(fd,STDOUT_FILENO);
                            }
                            dup2(pipegate[i-1][0],STDIN_FILENO);
                            close(pipegate[i-1][1]);

                            if(detail.bg){
                                int fd=open("/dev/null",O_RDWR);
                                dup2(fd,STDIN_FILENO);
                                dup2(fd,STDOUT_FILENO);
                            }
                        }
                        else{
                            dup2(pipegate[i-1][0],STDIN_FILENO);
                            close(pipegate[i-1][1]);
                            dup2(pipegate[i][1],STDOUT_FILENO);
                            close(pipegate[i][0]);
                        }

                        if(execvp(argv[i][0],argv[i])<0) {
                            fprintf(stdout,"%s: Program not found.\n",argv[i][0]);
```

```
                        exit(0);
                    }

                }else{
                    close(pipegate[i][1]);
                    int state;
                    waitpid(pid,&state,0);
                }
            }
            exit(0);
        }else{
            if(detail.bg==0){
                int state;
                waitpid(pid, &state, 0);
            }
        }
    }
    else{
        //如果没管道
        if((pid=fork())==0){
            if(detail.input==1){
                int fd=open(detail.input_file,O_RDONLY);
                dup2(fd,STDIN_FILENO);
            }

            if(detail.output==1){
                int fd=open(detail.output_file,O_WRONLY|O_CREAT);
                dup2(fd,STDOUT_FILENO);
            }
            else if(detail.output==2){
                int fd=open(detail.output_file,O_WRONLY|O_APPEND);
                dup2(fd,STDOUT_FILENO);
            }

            if(detail.bg){
                int fd=open("/dev/null",O_RDWR);
                dup2(fd,STDIN_FILENO);
                dup2(fd,STDOUT_FILENO);
            }


            if (execvp(argv[0][0],argv[0]) < 0) {
                fprintf(stdout,"%s: Program not found.\n",argv[0][0]);
                exit(0);
            }
        }else{
            if(detail.bg==0){
                int state;
```

```
                    waitpid(pid, &state, 0);
                }
            }
        }
    }
    return;
}

void parseline(char *cmdline,char *argv[MAXPIPE][MAXARG],struct Detail *detail){
    //parseline 读取 cmdline 然后把结果放到 argv 和 detail 中
    char *locate;//用于定位
    int argc; //一个命令后面所接参数数量
    int cnt1; //目前记录了的命令数量

    cmdline[strlen(cmdline)-1] = ' '; //把最后一个字符'\n'置为' '
    while (*cmdline&&(*cmdline ==' '))cmdline++; //丢弃第一个指令前多余的空格

    detail->input=0;detail->output=0;detail->bg=0;
    cnt1=0;argc=0;
    locate=strchr(cmdline,' ');
    while (locate) {
        char *arg_tmp=strtok(cmdline," ");
        if(!strcmp(arg_tmp, "<")){
            detail->input= 1;
            *locate='\0';
            cmdline=locate+1;
            while (*cmdline && (*cmdline ==' '))cmdline++;
            locate=strchr(cmdline,' ');
            if(!locate){
                printf("Where is the file?!");
                return;
            }
            detail->input_file=strtok(cmdline," ");
        }
        else if(!strcmp(arg_tmp,">")){
            detail->output= 1;
            *locate='\0';
            cmdline=locate+1;
            while (*cmdline && (*cmdline==' '))cmdline++;
            locate=strchr(cmdline,' ');
            if(!locate){
                printf("Where is the file?!");
                return;
            }
            detail->output_file=strtok(cmdline," ");
        }
        else if(!strcmp(arg_tmp, ">>")){
            detail->output=2;
```

```
            *locate='\0';
            cmdline=locate + 1;
            while(*cmdline && (*cmdline == ' '))cmdline++;
            locate=strchr(cmdline,' ');
            if(!locate){
                printf("Where is the file?!");
                return;
            }
            detail->output_file=strtok(cmdline," ");
        }
        else if(!strcmp(arg_tmp, "|")){
            ++argc;
            argv[cnt1][argc]=NULL;
            ++cnt1;
            argc=0;
        }
        else{
            argv[cnt1][argc]=arg_tmp;
            argc++;
        }
        *locate='\0';
        cmdline=locate+1;
        while (*cmdline && (*cmdline == ' '))cmdline++;
        locate=strchr(cmdline,' ');
    }
    argv[cnt1][argc]=NULL;
    detail->pipenum=cnt1+1;

    if(argc == 0){
        //如果最后一个命令压根就没有参数
        //如果不加这个，会出错，因为后面的 else if 判断可能越界
        detail->bg = 0;
    }
    else if(!strcmp(argv[cnt1][argc-1],"&"))detail->bg=1;


    if (detail->bg==1){
        argc--;
        argv[cnt1][argc] = NULL;
    }
}

int builtin_cmd(char **argv){
    if(!strcmp(argv[0],"cd")){
        if(chdir(argv[1])<0){
            printf("Error occurs when cd!\n");
        }
        return 1;
    }
```

```
    else if(!strcmp(argv[0],"history")){
        int n;
        if(!argv[1])n=cnt-1;
        else n=atoi(argv[1]);

        if(n<cnt){
            for(int i=cnt-n;i<cnt;i++)fprintf(stdout,"%s\n",cmd_history[i]);
        }else{
            printf("Error occurs when history\n");
        }
        return 1;
    }
    else if(!strcmp(argv[0],"mytop")){
        //too complex!
        mytop();
        return 1;
    }
    else if(!strcmp(argv[0],"exit"))exit(0);

    return 0;
}



void mytop(){
    int total_size,free_size,cached_size; //主存信息
    get_memory(&total_size,&free_size,&cached_size);
    fprintf(stdout, "Total: %dK, Free: %dK, Cached: %dK\n", total_size, free_size,
cached_size);
    getkinfo(); //读取/proc/kinfo 计算进程和任务总数 nr_total
    get_procs(); //读取每个进程的信息
    if (prev_proc == NULL)get_procs();
    float idle = print_procs(prev_proc, proc, 1); //输出 CPU 使用时间
    fprintf(stdout, "CPU Usage: %f%%\n", idle);
}

void get_memory(int *total_size, int *free_size, int *cached_size){
    int mem_f; //File descriptor of memory info
    int bufsize; //Actual size of bytes read
    char buf[MAXBUF]; //Buffer for reading from file
    int page_size, total_page, free_page, largest_page, cached_page;

    mem_f = open("/proc/meminfo", O_RDONLY); //Open memory info file
    bufsize = read(mem_f, buf, sizeof(buf)); //Read memory info
    if(bufsize == -1){
        printf("Error reading memory info!");
    }
    else{
```

```
        page_size = atoi(strtok(buf, " "));
        total_page = atoi(strtok(NULL, " "));
        free_page = atoi(strtok(NULL, " "));
        largest_page = atoi(strtok(NULL, " "));
        cached_page = atoi(strtok(NULL, " "));
        *total_size = (page_size * total_page) / 1024;
        *free_size = (page_size * free_page) / 1024;
        *cached_size = (page_size * cached_page) / 1024;
    }
}

void getkinfo(){
    int fd; //File descriptor of kinfo
    int bufsize; //Actual buffer size
    char buf[MAXBUF], pathbuf[MAXBUF]; //Buffer for file reading

    fd=open("/proc/kinfo", O_RDONLY);
    if (fd == -1) {
        printf("Reading kinfo file error!");
        exit(1);
    }
    bufsize = read(fd, buf, sizeof(buf)); //Read process info
    if(bufsize == -1){
        printf("Error reading total process info!");
    }
    else {
        nr_procs = (unsigned int) atoi(strtok(buf, " ")); //Number of process
        nr_tasks = (unsigned int) atoi(strtok(NULL, " ")); //Number of tasks
        close(fd);
        nr_total = (int) (nr_procs + nr_tasks);
    }
}

void parse_file(pid_t pid)  {
    // 遍历/proc/pid/psinfo  读取每个进程的信息
    /*
        如果 type==task，则进程标记 p_flags |=istask
        如果 type==system，则进程标记 p_flags |=issystem
        如果 state!=state_run,则标记 p_flags |=blocked
    */

    /*
        再连续读 CPUTIMENAMES 次 cycles_hi,cycle_lo（三个 cputimenames）
        然后拼接成 64 位，放在 p_cpucycles[]数组中。
    */


    char path[MAXBUF], name[256], type, state;
    int version, endpt, effuid;
```

```
    unsigned long cycles_hi, cycles_lo;
    FILE *fp;
    struct proc *p;
    int i;

    sprintf(path, "/proc/%d/psinfo", pid);

    if ((fp = fopen(path, "r")) == NULL)
        return;

    if (fscanf(fp, "%d", &version) != 1) {
        fclose(fp);
        return;
    }

    if (fscanf(fp, " %c %d", &type, &endpt) != 2) {
        fclose(fp);
        return;
    }

    slot++;

    if(slot < 0 || slot >= nr_total) {
        fprintf(stderr, "Unreasonable endpoint number %d\n", endpt);
        fclose(fp);
        return;
    }

    p = &proc[slot];

    if (type == TYPE_TASK)
        p->p_flags |= IS_TASK;
    else if (type == TYPE_SYSTEM)
        p->p_flags |= IS_SYSTEM;

    p->p_endpoint = endpt;
    p->p_pid = pid;

    if (fscanf(fp, " %255s %c %d %d %lu %*u %lu %lu",
               name, &state, &p->p_blocked, &p->p_priority,
               &p->p_user_time, &cycles_hi, &cycles_lo) != 7) {

        fclose(fp);
        return;
    }

    strncpy(p->p_name, name, sizeof(p->p_name)-1);
    p->p_name[sizeof(p->p_name)-1] = 0;
```

```
    if (state != STATE_RUN)
        p->p_flags |= BLOCKED;
    p->p_cpucycles[0] = make_cycle(cycles_lo, cycles_hi);
    p->p_memory = 0L;

    if (!(p->p_flags & IS_TASK)) {
        int j;
        if ((j=fscanf(fp, " %lu %*u %*u %*c %*d %*u %u %*u %d %*c %*d %*u",
                    &p->p_memory, &effuid, &p->p_nice)) != 3) {

            fclose(fp);
            return;
        }

        p->p_effuid = effuid;
    } else p->p_effuid = 0;


    for(i = 1; i < CPUTIMENAMES; i++) {
        if(fscanf(fp, " %lu %lu",
                  &cycles_hi, &cycles_lo) == 2) {
            p->p_cpucycles[i] = make_cycle(cycles_lo, cycles_hi); //拼接成 64 位使得后
续计算可以正常进行
        } else  {
            p->p_cpucycles[i] = 0;
        }
    }

    if ((p->p_flags & IS_TASK)) {
        if(fscanf(fp, " %lu", &p->p_memory) != 1) {
            p->p_memory = 0;
        }
    }

    p->p_flags |= USED;

    fclose(fp);
}

void parse_dir(){
    DIR *p_dir; //Pointer of directory
    struct dirent *p_ent; //Info of the directory
    pid_t pid; //Name of sub directory(PID)
    char *end;

    if ((p_dir = opendir("/proc")) == NULL) {
        exit(1);
```

```
    }

    //Traverse the directory
    for (p_ent = readdir(p_dir); p_ent != NULL; p_ent = readdir(p_dir)) {
        pid = strtol(p_ent->d_name, &end, 10); //Get the name of sub directory
        if (!end[0] && pid != 0)parse_file(pid);
    }
    closedir(p_dir);
}

void get_procs(){
    struct proc *p;
    int i;
    p = prev_proc;
    prev_proc = proc;
    proc = p;

    if (proc == NULL) {
        proc = malloc(nr_total * sizeof(proc[0])); //Allocate a new process
structure
        //Allocate failed
        if (proc == NULL) {
            fprintf(stderr, "Out of memory!\n");
            exit(1);
        }
    }
    //Initialize all the entry ranging in the total process+task num
    for (i = 0; i < nr_total; i++)proc[i].p_flags = 0;
    parse_dir();
}

uint64_t make_cycle(unsigned long lo, unsigned long hi){
    // MINIX 进程信息文件中保存的分别是周期的高 32 位及低 32 位
    // 我们需要首先将他们拼接成一个完整的 64 位整数才能进行后续的计算
    return ((uint64_t)hi << 32) | (uint64_t)lo;
}

uint64_t cputicks(struct proc *p1, struct proc *p2, int timemode){
    int i;
    uint64_t t = 0;

    // 计算每个进程 proc 的滴答，通过 proc 和当前进程 prev_proc 做比较
    // 如果 endpoint 相等，则在循环中分别计算
    for(i = 0; i < CPUTIMENAMES; i++) {
        if(!CPUTIME(timemode, i))continue;
        if(p1->p_endpoint == p2->p_endpoint) {
            t = t + p2->p_cpucycles[i] - p1->p_cpucycles[i];
        } else {
```

```
            t = t + p2->p_cpucycles[i];
        }
    }
    return t;
}


float print_procs(struct proc *proc1, struct proc *proc2, int cputimemode){
    int p, nprocs;
    uint64_t systemticks = 0;
    uint64_t userticks = 0;
    uint64_t total_ticks = 0;
    static struct tp *tick_procs = NULL;

    if (tick_procs == NULL) {
        tick_procs = malloc(nr_total * sizeof(tick_procs[0]));
        if (tick_procs == NULL) {
            fprintf(stderr, "Out of memory!\n");
            exit(1);
        }
    }

    for (p = nprocs = 0; p < nr_total; p++) {
        uint64_t uticks;
        if (!(proc2[p].p_flags & USED))continue;
        tick_procs[nprocs].p = proc2 + p;
        tick_procs[nprocs].ticks = cputicks(&proc1[p], &proc2[p], cputimemode);
        uticks = cputicks(&proc1[p], &proc2[p], 1);
        total_ticks = total_ticks + uticks;
        // 计算总的 cpu 使用百分比，遍历所有的进程和任务
        // 计算 systemticks，userticks
        if(!(proc2[p].p_flags & IS_TASK)) {
            if(proc2[p].p_flags & IS_SYSTEM)systemticks = systemticks +
tick_procs[nprocs].ticks;
            else userticks = userticks + tick_procs[nprocs].ticks;
        }
        nprocs++;
    }

    if (total_ticks == 0)return 0.0;
    return 100.0 * (systemticks + userticks) / total_ticks;
}
```

五、总结

```
# clang myshell.c -o myshell
```

```
# ./myshell
Hello there!I'm tommy and this is the first project of OS in ECNU DASE
tommyshell  /root/project-shell >
```

cd testshell

```
tommyshell  /root/project-shell >cd testshell
tommyshell  /root/project-shell/testshell >
```

ls -a -l

```
tommyshell  /root/project-shell/testshell >ls -a -l
total 16
drwxr-xr-x  2 root  operator   128 Mar 18 23:29 .
drwxr-xr-x  3 root  operator  2048 Mar 18 23:30 ..
```

ls -a -l > result.txt

```
tommyshell  /root/project-shell/testshell >ls -a -l > result.txt
tommyshell  /root/project-shell/testshell >cat result.txt
total 16
drwxr-xr-x  2 root  operator   192 Mar 18 23:36 .
drwxr-xr-x  3 root  operator  2048 Mar 18 23:30 ..
--w---S---  1 root  operator     0 Mar 18 23:36 result.txt
tommyshell  /root/project-shell/testshell >
```

vi result.txt

```
total 1632
drwxr-xr-x  2 root  operator     256 Mar 18 23:42 .
drwxr-xr-x  3 root  operator    2048 Mar 18 23:30 ..
-rwxr-xr-x  1 root  operator  815700 Mar 18 23:42 core.253
--w---S---  1 root  operator     169 Mar 18 23:36 result.txt
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
result.txt: unmodified: line 1
```

grep a < result.txt

```
tommyshell  /root/project-shell/testshell >grep a < result.txt
total 16
drwxr-xr-x  2 root  operator   192 Mar 18 23:36 .
drwxr-xr-x  3 root  operator  2048 Mar 18 23:30 ..
--w---S---  1 root  operator     0 Mar 18 23:36 result.txt
```

ls -a -l | grep a

```
tommyshell  /root/project-shell/testshell >ls -a -l | grep a
total 24
drwxr-xr-x  2 root  operator   192 Mar 18 23:36 .
drwxr-xr-x  3 root  operator  2048 Mar 18 23:30 ..
--w---S---  1 root  operator   169 Mar 18 23:36 result.txt
tommyshell  /root/project-shell/testshell >
```

```
tommyshell  /root/project-shell >ls
myshell   myshell.c testshell
tommyshell  /root/project-shell >ls | grep c
myshell.c
tommyshell  /root/project-shell >
```

vi result.txt &

```
tommyshell  /root/project-shell/testshell >vi result.txt &
tommyshell  /root/project-shell/testshell >ex/vi: Vi's standard input and output must be a terminal
```

mytop

```
tommyshell  /root/project-shell >mytop
Total: 1046972K, Free: 991816K, Cached: 26392K
CPU Usage: 0.142393%
tommyshell  /root/project-shell >
```

history n

```
tommyshell  /root/project-shell/testshell >history
ls -a -l

ls -a -l > result.txt

vi result.txt

grep a < result.txt

ls -a -l | grep a

vi result.txt &

shit

mytop

history

tommyshell  /root/project-shell/testshell >
```

```
tommyshell  /root/project-shell/testshell >history 3
mytop

history

history 3
```

exit

```
tommyshell  /root/project-shell/testshell >exit
#
```

多重管道测试

```
tommyshell  /root/project-shell >cd /
tommyshell  / >ls
bin           boot_monitor  home          proc          sbin          tmp
boot          dev           lib           result.txt    service       usr
boot.cfg      etc           mnt           root          sys           var
tommyshell  / >ls | grep b | wc -l
       6
tommyshell  / >
```

在完成这个实验之后我十分有成就感，不仅是因为自己从小白开始实现了一个建议的Shell,而且因为自己对于相关的系统调用有了更深刻的理解。基本的函数是经典的，调试的过程是枯燥的，在逐步完善和突破的过程中，我对未来学习操作系统充满了信心。