《Mini Shell》

操作系统实践报告

题　　目：	操作系统实践报告
上课时间：	周五早上
授课教师：	李旭东
姓　　名：	张怡桢
学　　号：	2013747
年　　级：	2020级本科生
日　　期：	2023/1/10

Mini Shell

张怡桢，2013747

南开大学软件学院

摘　要：Shell 是一个应用程序，它连接了用户和 Linux 内核，让用户能够更加高效、安全、低成本地使用 Linux 内核，这就是 Shell 的本质。Shell 能够接收用户输入的命令，并对命令进行处理，处理完毕后再将结果反馈给用户，比如输出到显示器、写入到文件等。在本次操作系统实践中，我基本上实现了完整Shell的功能，并在此基础上，做出了拓展，支持多命令串行，多管道运行，后台运行，输入输出重定向，输出追加等功能。

关键词：shell；管道；后台工作；重定向；多命令；环境变量；多管道

1 实验题目

Minishell，实现：

1. 支持后台符号&
2. 支持串行执行多个命令";"
3. 支持管道"|"
4. 支持环境变量

在基础要求上，我还实现了以下部分：

其他要求：

1. 支持输出输出重定向，实现了'>'，'<'，'>>'等功能。
2. 支持多管道运行。
3. 支持用户输入一行命令及其多个参数，并解析执行，并输出结果；
4. 支持 cd 命令，若无参数则回到当前用户的登录目录（见下面提示）；
5. 支持以“当前路径”和“用户名”为提示符；
6. 支持对命令行中空格的自动忽略处理；
7. 支持对命令行中 tab 键的自动忽略处理；
8. 支持一行中以“；”（为标志）分隔的多个命令及多个参数的顺序执行， 即如下：
   (a) MINI SHELL#pwd; ls –l;date

2 原理方法

2.1 shell内置命令和外部命令

内部命令实际上是shell程序的一部分，其中包含的是一些比较简单的linux系统命令，这些命令由shell程序识别并在shell程序内部完成运行，通常在linux系统加载运行时shell就被加载并驻留在系统内存中。内部命令是写在bashy源码里面的，其执行速度比外部命令快，因为解析内部命令shell不需要创建子进程。比如：exit，history，cd，echo等。

外部命令是linux系统中的实用程序部分，因为实用程序的功能通常都比较强大，所以其包含的程序量也会很大，在系统加载时并不随系统一起被加载到内存中，而是在需要时才将其调用内存。通常外部命令的实体并不包含在shell中，但是其命令执行过程是由shell程序控制的。shell程序管理外部命令执行的路径查找、加载存放，并控制命令的执行。外部命令是在bash之外额外安装的，通常放在/bin，/usr/bin，/sbin，/usr/sbin.....等等。可通过“echo $PATH”命令查看外部命令的存储路径，比如：ls、vi等。

用type命令可以分辨内部命令与外部命令：

```
[root@node3 tmp]# type cd
cd is a shell builtin
[root@node3 tmp]# type mkdir
mkdir is hashed (/bin/mkdir)
[root@node3 tmp]#
```

内部命令和外部命令最大的区别之处就是性能。内部命令由于构建在shell中而不必创建多余的进程，要比外部命令执行快得多。因此和执行更大的脚本道理一样，执行包含很多外部命令的脚本会损害脚本的性能。

2.2 Shell重定向

Linux Shell 重定向分为两种，一种输入重定向，一种是输出重定向；从字面上理解，输入输出重定向就是「改变输入与输出的方向」的意思。

一般情况下，我们都是从键盘读取用户输入的数据，然后再把数据拿到程序（C语言程序、Shell 脚本程序等）中使用；这就是标准的输入方向，也就是从键盘到程序。

反过来说，程序中也会产生数据，这些数据一般都是直接呈现到显示器上，这就是标准的输出方向，也就是从程序到显示器。

我们可以把观点提炼一下，其实输入输出方向就是数据的流动方向：

- 输入方向就是数据从哪里流向程序。数据默认从键盘流向程序，如果改变了它的方向，数据就从其它地方流入，这就是输入重定向。
- 输出方向就是数据从程序流向哪里。数据默认从程序流向显示器，如果改变了它的方向，数据就流向其它地方，这就是输出重定向。

Linux 中一切皆文件，包括标准输入设备（键盘）和标准输出设备（显示器）在内的所有计算机硬件都是文件。

为了表示和区分已经打开的文件，Linux 会给每个文件分配一个 ID，这个 ID 就是一个整数，被称为文件描述符（File Descriptor）。

Linux 程序在执行任何形式的 I/O 操作时，都是在读取或者写入一个文件描述符。一个文件描述符只是一个和打开的文件相关联的整数，它的背后可能是一个硬盘上的普通文件、FIFO、管道、终端、键盘、显示器，甚至是一个网络连接。

当执行shell命令时，会默认打开3个文件，每个文件有对应的文件描述符来方便我们使用：

stdin、stdout、stderr 默认都是打开的，在重定向的过程中，0、1、2 这三个文件描述符可以直接使用。

类型	文件描述符	默认情况	对应文件句柄位置	文件名
标准输入（standard input）	0	从键盘获得输入	/proc/slef/fd/0	stdin
标准输出（standard output）	1	输出到屏幕（即控制台）	/proc/slef/fd/1	stdout
错误输出（error output）	2	输出到屏幕（即控制台）	/proc/slef/fd/2	stderr

所以我们平时在执行shell命令中，都默认是从键盘获得输入，并且将结果输出到控制台上。但是我们可以通过更改文件描述符默认的指向，从而实现输入输出的重定向。比如我们将1指向文件，那么标准的输出就会输出到文件中。

2.3 Shell管道

将两个或者多个命令（程序或者进程）连接到一起，把一个命令的输出作为下一个命令的输入，以这种方式连接的两个或者多个命令就形成了**管道（pipe）**。

Linux 管道使用竖线 `|` 连接多个命令，这被称为管道符。Linux 管道的具体语法格式如下：

```
command1 | command2
command1 | command2 [ | commandN... ]
```

当在两个命令之间设置管道时，管道符 `|` 左边命令的输出就变成了右边命令的输入。只要第一个命令向标准输出写入，而第二个命令是从标准输入读取，那么这两个命令就可以形成一个管道。大部分的 Linux 命令都可以用来形成管道。

> !!! 需要注意，command1 必须有正确输出，而 command2 必须可以处理 command2 的输出结果；而且 command2 只能处理 command1 的正确输出结果，不能处理 command1 的错误信息。

2.4 Shell后台运行

在后台执行多个命令是Linux的一个有用的功能，允许用户同时执行多个任务。当运行可能需要一段时间才能完成的长期运行的命令时，这可能特别有帮助，因为它允许用户在后台执行该命令时继续处理其他任务。

在Linux上，在后台运行一个命令的最直接的方法之一是使用”&”操作符。这个操作符被用来在后台运行一个命令，并将终端的控制权还给用户。

要使用”&”运算符，只需将其附加到你想在后台运行的命令的末尾。例如，要在后台运行睡眠命令，你可以输入以下命令

```
$ sleep 45 &
```

该命令将执行sleep命令，使终端暂停45秒，然后将终端的控制权还给用户。该命令将继续在后台运行，直到完成。

2.5 Shell环境变量

2.5.1 环境变量的含义

程序（操作系统命令和应用程序）的执行都需要运行环境，这个环境是由多个环境变量组成的。

2.5.2 环境变量的分类

- 1.按生效的范围分类。

系统环境变量：公共的，对全部的用户都生效。

用户环境变量：用户私有的、自定义的个性化设置，只对该用户生效。

- 2.按生存周期分类。

永久环境变量：在环境变量脚本文件中配置，用户每次登录时会自动执行这些脚本，相当于永久生效。

临时环境变量：使用时在Shell中临时定义，退出Shell后失效。

- 3.Linux环境变量

Linux环境变量也称之为Shell环境变量，以下划线和字母打头，由下划线、字母（区分大小写）和数字组成，习惯上使用大写字母，例如 `PATH`、`HOSTNAME`、`LANG` 等。

3 Shell设计

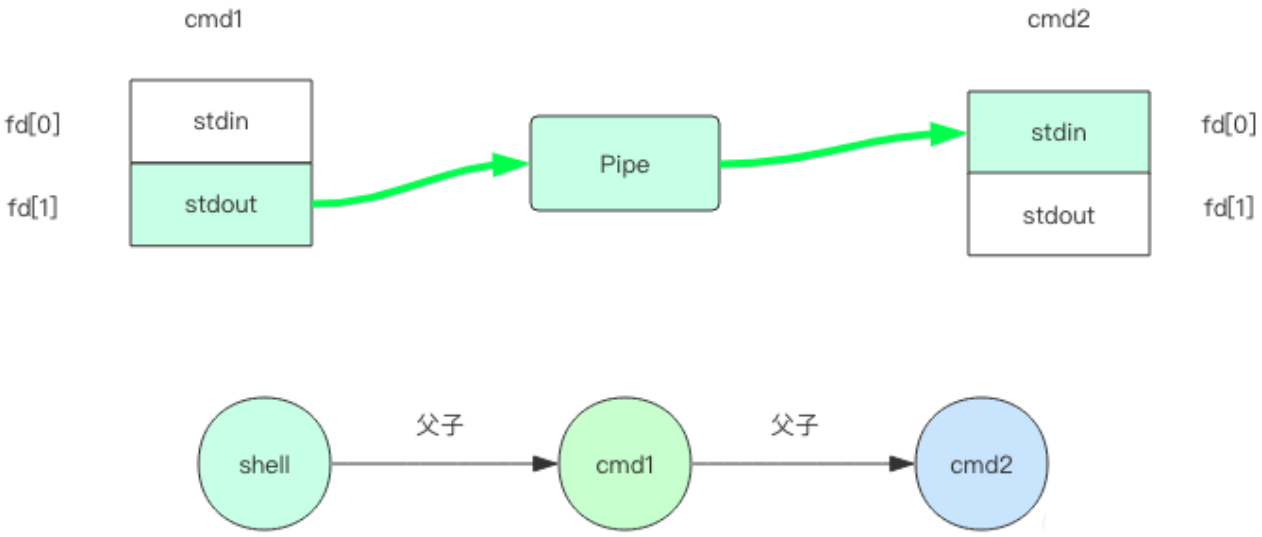
Shell的实现需要理解Shell的原理，并对Shell命令进行设计。

3.1 Shell管道

管道指令的实现原理：

```
$ cmd1 | cmd2
```

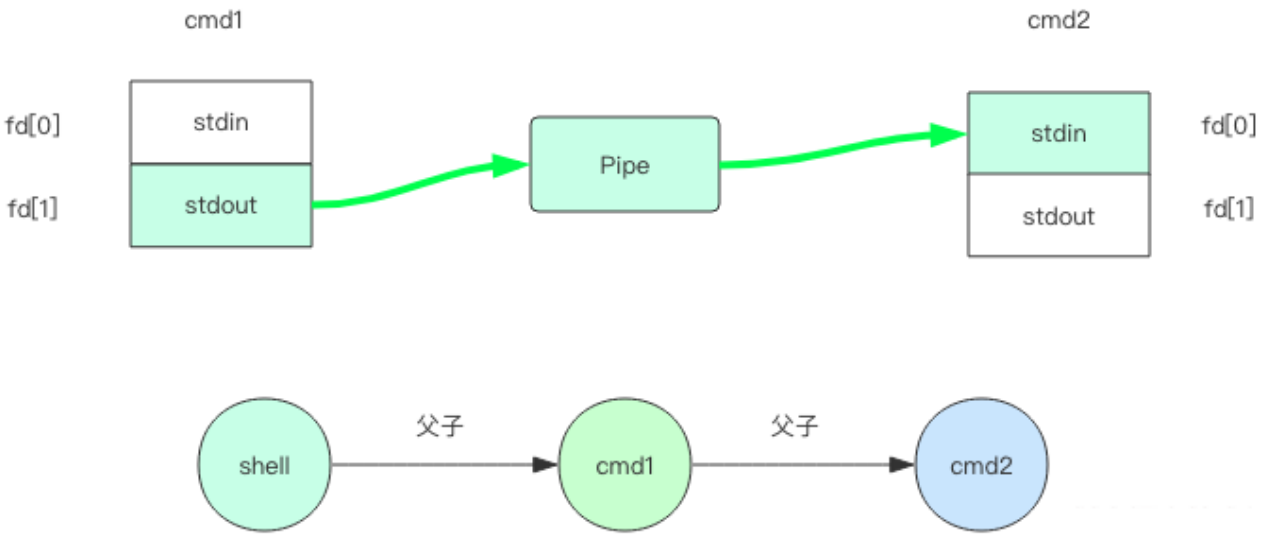
分析上面的这条命令，其实现原理如图所示，其中展示了进程描述符表、管道、进程的父子关系：



3.1.1 fork 和 exec

shell 每次执行指令， 需要 fork 出一个子进程来执行，然后将子进程的镜像替换成目标指令，这又会用到 exec 函数。比如下面这条简单的指令

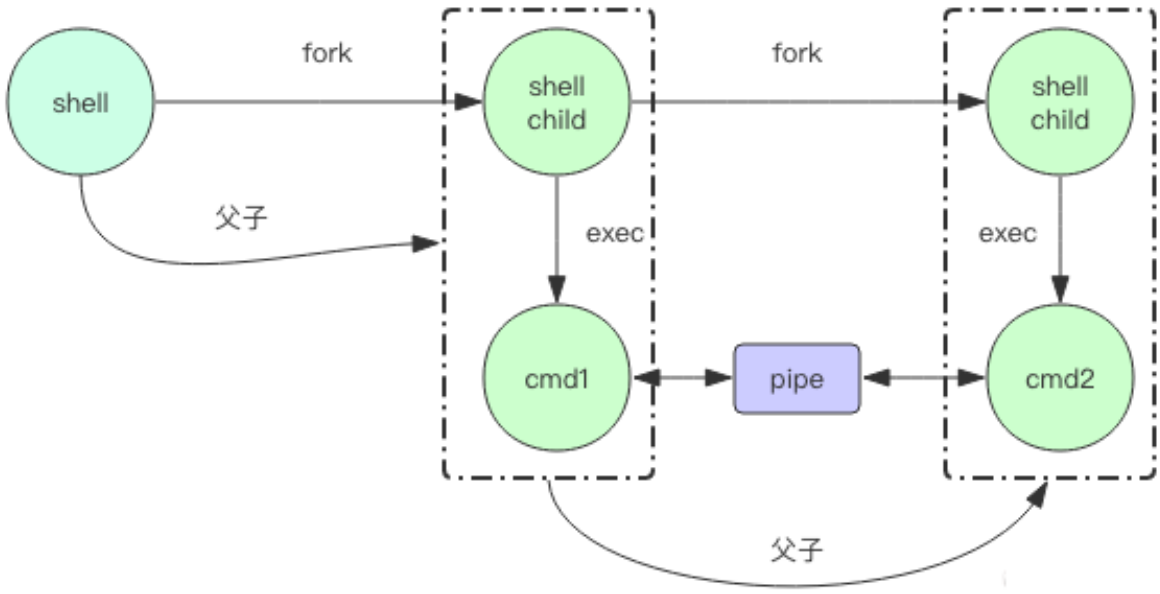
```
$ cmd
```



exec 函数不会改变当前进程的进程号，不会改变进程之间的父子关系。可以将进程看成一个带壳的球体，exec 之后，外面的壳不会变，球里面的东西被完全替换了。而输入输出文件描述符默认在壳上面，这意味着指令 cmd 的输入输出继承了 shell 进程的输入输出。

```
$ cmd1 | cmd2
```

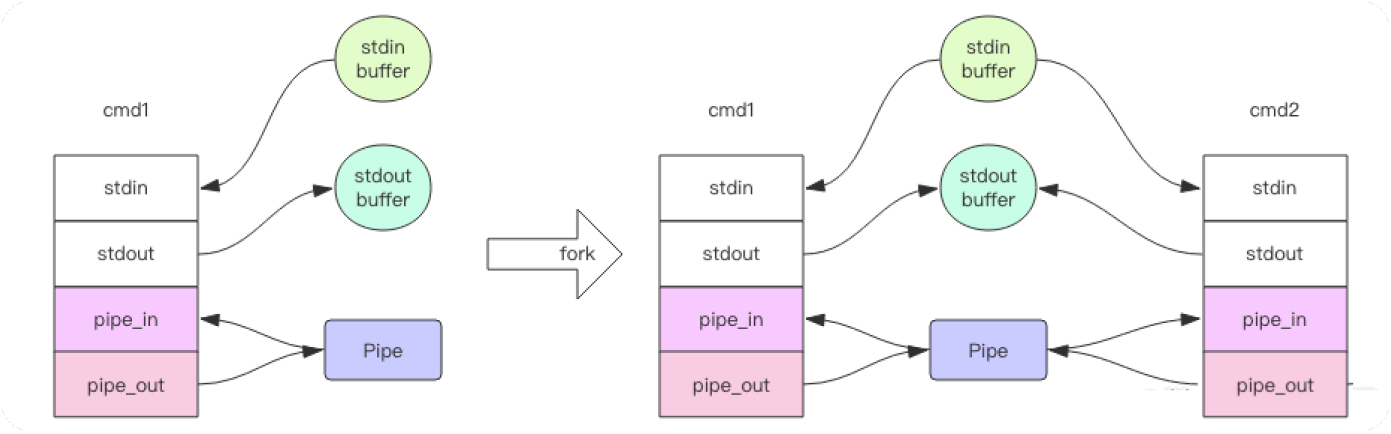
当指令里面包含一个管道符，意味着需要并行执行两个指令，这时候 shell 需要 fork 两次生成两个子进程，然后分别 exec 换成目标指令。



图里面还有一个 pipe，它就是负责父子进程通信的管道。

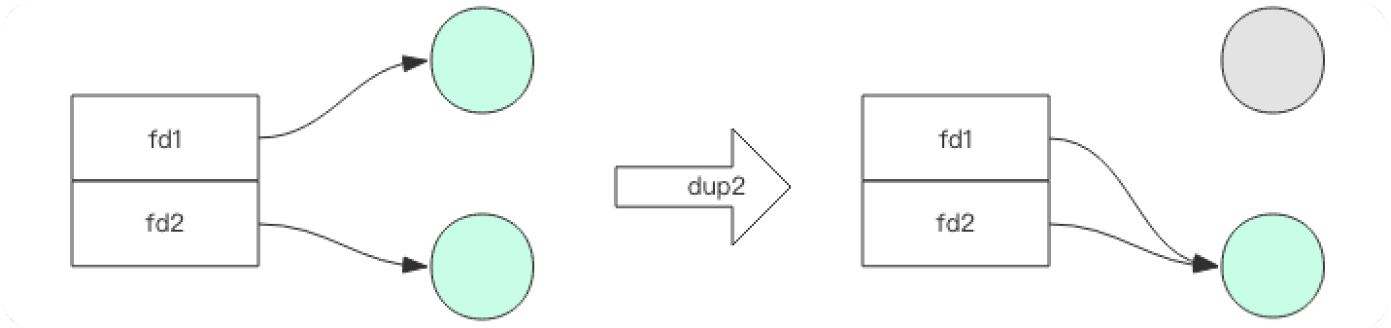
3.1.2 pipe

管道用于父子进程的通信，在 fork 之前创建 pipe，pipe将成为 fork 之后父子进程之间的纽带。pipe 函数会返回两个描述符（pipe_in, pipe_out），一个用于读，一个用于写。

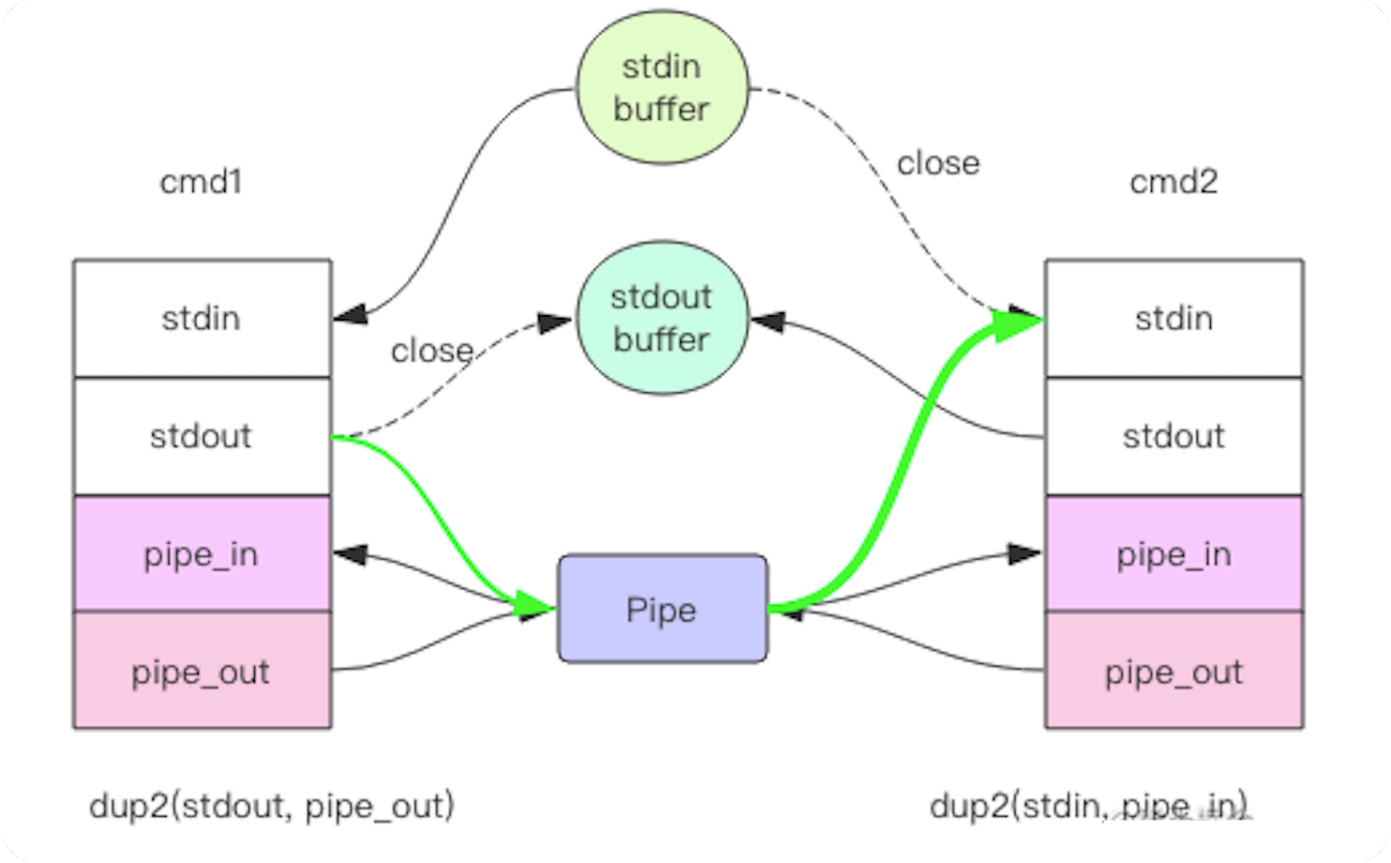


3.1.3 dup2

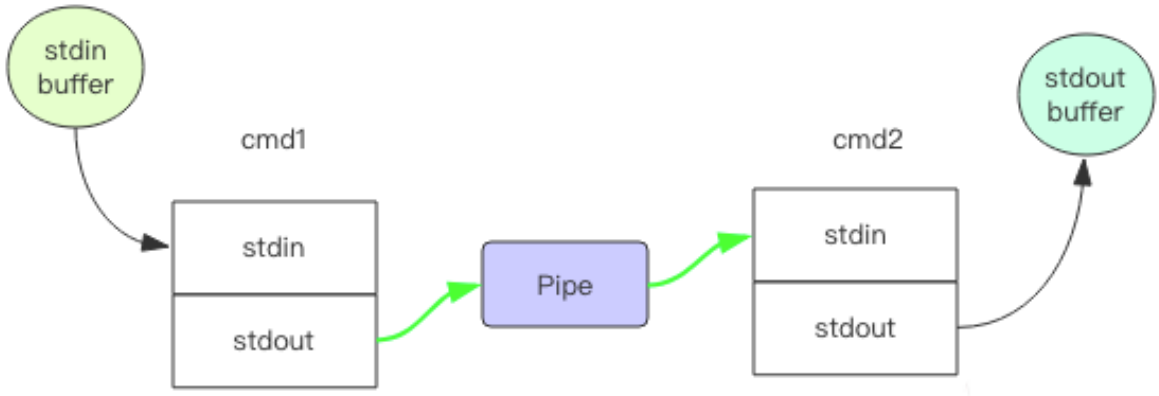
下面我们就需要调整图中描述符的尖头，将 cmd1 进程的 stdout 描述符指向管道写，将 cmd2 进程的 stdin 描述符指向管道读，这就需要神奇的 dup2(fd1, fd2) 函数，它的作用是将 fd1 描述符关联 fd2 指向的内核对象，之前 fd1 指向的内核对象引用计数减一，如果减到零就销毁。注意平时我们调用 close 方法本质上只是递减引用计数，同一个内核对象是可以被多个进程共享的。当引用计数减到零时就会正式关闭。



下面我们将 dup2 函数的规则应用一下，对两个进程分别调用 dup2 方法得到

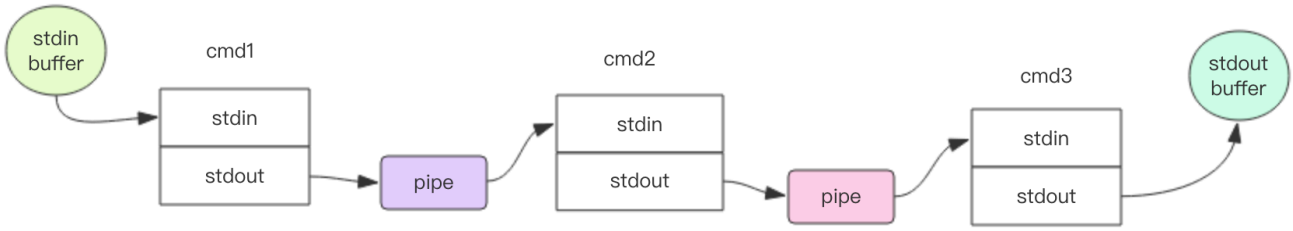


然后再将不需要的描述符关闭掉，就得到了下面的终极图：



如果是两个管道符三个命令如下，就会生成两个管道

```
$ cmd1 | cmd2 | cmd3
```



3.2 Shell文件重定向

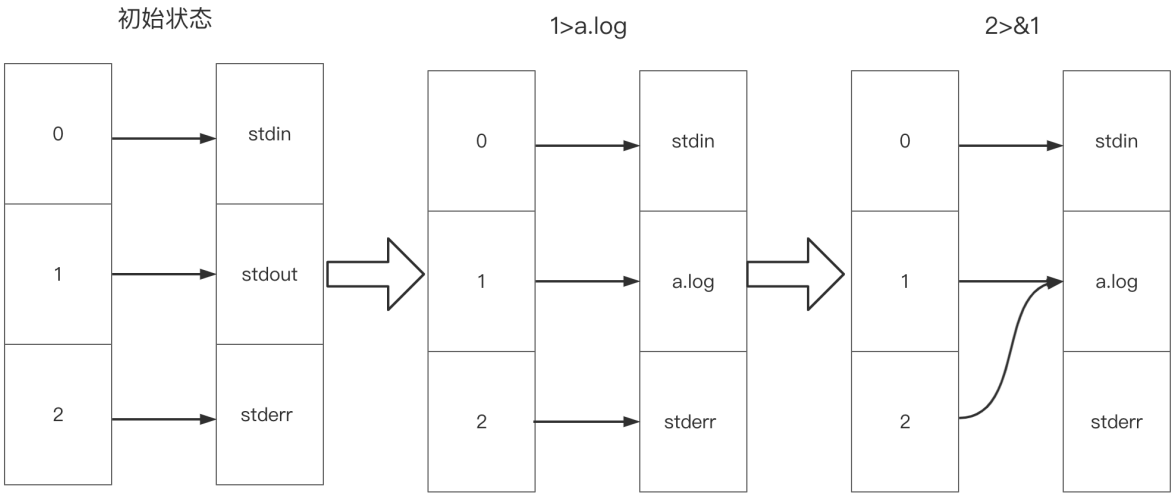
我们使用`>`或者`>>`对输出进行重定向。符号的左边表示文件描述符，**如果没有的话表示1，也就是标准输出**，符号的右边可以是一个文件，也可以是一个输出设备。当使用`>`时，会判断右边的文件存不存在，如果存在的话就先删除，然后创建一个新的文件，不存在的话则直接创建。但是当使用`>>`进行追加时，则不会删除原来已经存在的文件。

我们使用`<`对输入做重定向，从`<`右边的文件或者输入设备获取内容。

默认的`<`的输入为stdin，也就是“从键盘获得输入”，其值为0

默认的`>`的输入为stdout与stderr，stdout是标准的输出，其值为1，stderr是错误的输出，其值为2，都默认输出到“显示器上”。

如下是对文件重定向的解析流程。



`1>a.log 2>&1` 命令解析流程

3.3 Shell后台运行

为了屏蔽键盘和控制台，子进程的标准输入、输出映射成`/dev/null`。子进程调用`signal(SIGCHLD,SIG_IGN)`，使得`Linux`接管此进程。 因此`Shell`可以避免调用`wait/waitpid`直接运行下一条命令。

- 将标准输入与标准输出重定向至`/dev/null`这个特殊的文件夹后再执行命令
- `/dev/null`:Linux系统的垃圾桶，能够将重定向到此的内容全部丢弃，不保存也不显示。

- 父进程直接返回，不等待子进程结束

3.4 Shell环境变量

Linux 的变量可分为两类：环境变量和本地变量

- 环境变量：或者称为全局变量，存在于所有的shell 中，在你登陆系统的时候就已经有了相应的系统定义的环境变量了。Linux 的环境变量具有继承性，即子shell 会继承父shell 的环境变量。
- 本地变量：当前shell 中的变量，很显然本地变量中肯定包含环境变量。Linux 的本地变量的非环境变量不具备继承性。

Linux 中环境变量的文件

当你进入系统的时候，Linux 就会为你读入系统的环境变量，Linux 中有很多记载环境变量的文件，它们被系统读入是按照一定的顺序的。

- 1. `/etc/profile`

此文件为系统的环境变量，它为每个用户设置环境信息，当用户第一次登录时，该文件被执行。并从 `/etc/profile.d` 目录的配置文件中搜集shell 的设置。这个文件，是任何用户登陆操作系统以后都会读取的文件（如果用户的shell 是 `csh`、`tcsh`、`zsh`，则不会读取此文件），用于获取系统的环境变量，只在登陆的时候读取一次。（假设用户使用的是BASH ）

- 2. `/etc/bashrc`

在执行完 `/etc/profile` 内容之后，如果用户的 SHELL 运行的是 bash ，那么接着就会执行此文件。另外，当每次一个新的 bash shell 被打开时，该文件被读取。每个使用 bash 的用户在登陆以后执行完 `/etc/profile` 中内容以后都会执行此文件，在新开一个 bash 的时候也会执行此文件。因此，如果你想让每个使用 bash 的用户每新开一个 bash 和每次登陆都执行某些操作，或者给他们定义一些新的环境变量，就可以在这个里面设置。

- 3. `~/.bash_profile`

每个用户都可使用该文件输入专用于自己使用的 shell 信息。当用户登录时，该文件仅仅执行一次，默认情况下，它设置一些环境变量，执行用户的 `.bashrc` 文件。单个用户此文件的修改只会影响到他以后的每一次登陆系统。因此，可以在这里设置单个用户的特殊的环境变量或者特殊的操作，那么它在每次登陆的时候都会去获取这些新的环境变量或者做某些特殊的操作，但是仅仅在登陆时。

- 4. `~/.bashrc`

该文件包含专用于单个人的 bash shell 的 bash 信息，当登录时以及每次打开一个新的 shell 时，该文件被读取。单个用户此文件的修改会影响到他以后的每一次登陆系统和每一次新开一个 bash。因此，可以在这里设置单个用户的特殊的环境变量或者特殊的操作，那么每次它新登陆系统或者新开一个 bash ，都会去获取相应的特殊的环境变量和特殊操作。

- 5. `~/.bash_logout`

当每次退出系统(退出bash shell) 时, 执行该文件。

用户登录后加载 `profile` 和 `bashrc` 的流程如下：

```
1)/etc/profile --> /etc/profile.d/*.sh
2)$HOME/.bash_profile --> $HOME/.bashrc --> /etc/bashrc
```

bash首先执行 `/etc/profile` 脚本，`/etc/profile` 脚本先依次执行 `/etc/profile.d/*.sh`。

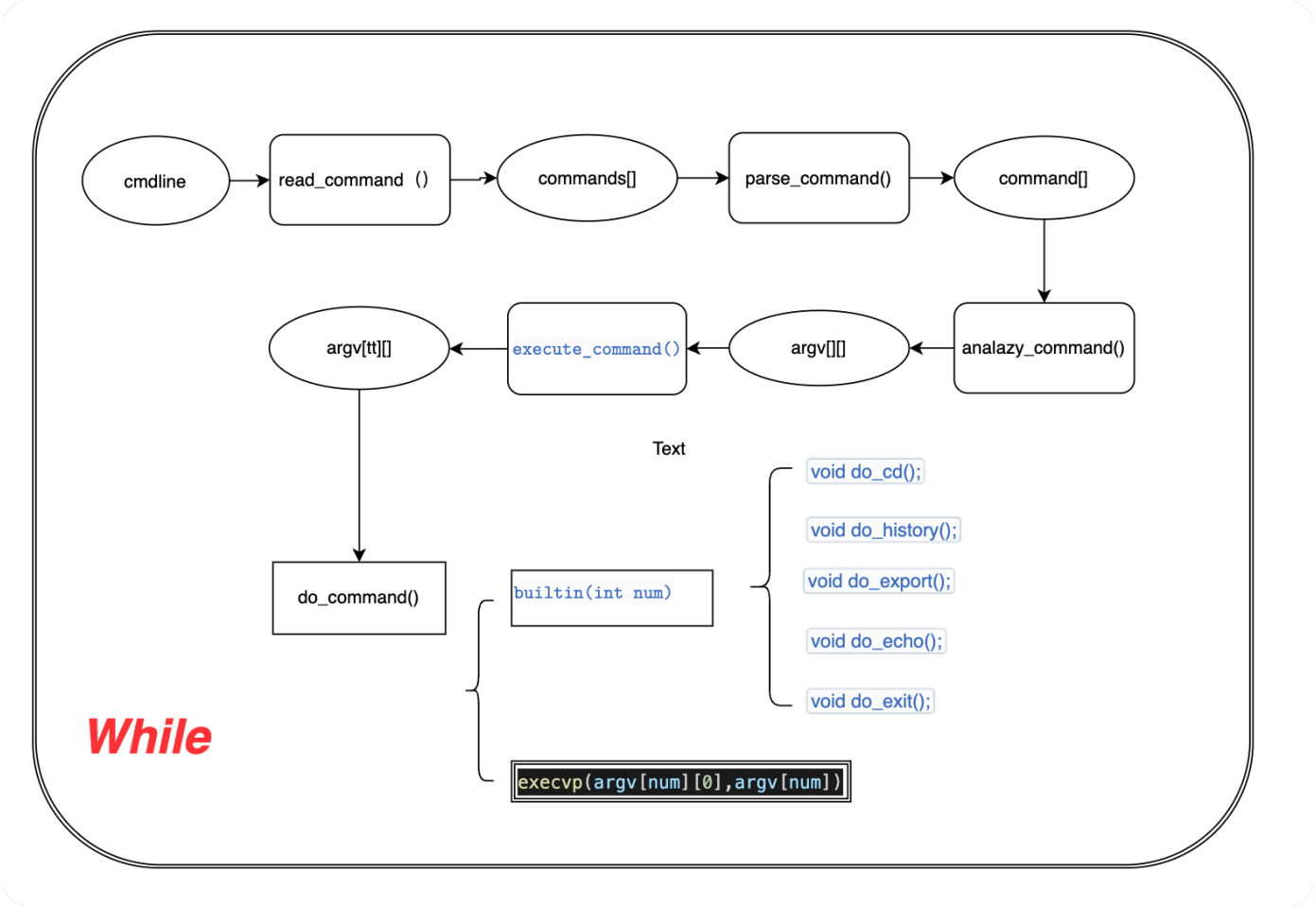
随后bash会执行用户主目录下的 `.bash_profile` 脚本，`.bash_profile` 脚本会执行用户主目录下的 `.bashrc` 脚本，

而 `.bashrc` 脚本会执行 `/etc/bashrc` 脚本。至此，所有的环境变量和初始化设定都已经加载完成。

bash随后调用 `terminfo` 和 `inputrc`，完成终端属性和键盘映射的设定。

4 实现思路

全局实现思路图如下：



4.1 全局定义

4.1.1 全局变量：

//全局常量定义

#define `SUCCESS` 0

#define `BUFFSIZE` 100

//全局变量定义

char `current_dir[BUFFSIZE]`; //当前所在的系统路径

char `user_dir[BUFFSIZE]`; //当前所在的系统路径

uid_t `uid`; //获得用户id

string `cmdline`; //从终端读入的字符串

string `separator`=";"; //终端读入多条命令的规定分隔符“；”

vector `history`; //记录所有历史命令

vector `commands`; // 所有命令,解析cmdline后得到的string数组

char *`command[100]`;//切分后一条命令的字符串数组

int `arg`=0;//切分后一条命令的参数个数

char *`argv[100][100]`;//解析单条命令command的参数

int `ar`=0;//管道个数

int `tt`;//记录当前遍历到第几个命令

int `flag[100][3]`;//管道的输入输出以及追加重定向标记

char *`file[100][2]={0}`;//对应两个重定向的文件

char * `f`=(char*)"temp.txt";//共享文件

int `pid`;//设为全局变量，方便获得子进程的进程号

bool `back_flag`; //是否后台执行

4.1.2 全局函数：

//全局函数声明

`void init();` //初始化函数

`void read_command();` //读取多种命令函数，主要处理“；”

`void parse_command();` //处理一条命令command

`void analazy_command();` //处理一条命令的多个参数，包括是否含有管道，输入输出定向

`int execute_command();` //处理多管道，多线程，后台执行等

`int do_command(int num);` //执行命令，包括内建以及外部

//内部命令

`int builtin(int num);`//执行内部命令

`void do_cd();`

`void do_history();`

`void do_echo();`

`void do_export();`

`void do_exit();`

4.2 Show Shell *显示终端提示符*

主要执行函数在`main()`函数中，输出好看的终端页面。

```
int main()
{

    printf("\033[32m----------------welcome to mini shell--------2013747------ \n\033[0m");
    /* 获取用户id */
  uid = getuid();
    strcpy(current_dir, getcwd(NULL, 0));
    strcpy(user_dir, getcwd(NULL, 0));
//    printf("PATH : %s\n", getenv("PATH"));
    while (1)
    {
        printf("\033[92m%s@MINISHELL\033[0m:\033[34m%s\033[1;35m", getlogin(),
current_dir);
         /* 打印用户提示符 */
        if (0 == uid)
        {
            printf("# \033[0m");    // 超级用户
        }
```

```
        else
        {
            printf("$ \033[0m");     // 普通用户
        }
        init();
        read_command();
        parse_command();

    }
    return 0;
}
```

得到进入minishell后可以获得当前路径以及文件名的提示

```
zhangyizhen2013747@ubuntu-linux-22-04-desktop:~/minishell$ g++ -o zyzshell zyzshell.cpp
zhangyizhen2013747@ubuntu-linux-22-04-desktop:~/minishell$ ./zyzshell
----------------welcome to mini shell--------2013747------
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$
```

- 终端提示符组成：[用户名+@+主机名+当前目录]+用户提示符

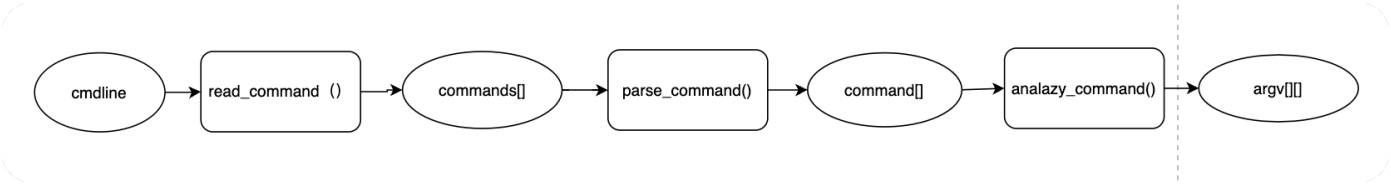
4.3 Pase Input 解析命令

设计到解析命令的函数有以下3种：

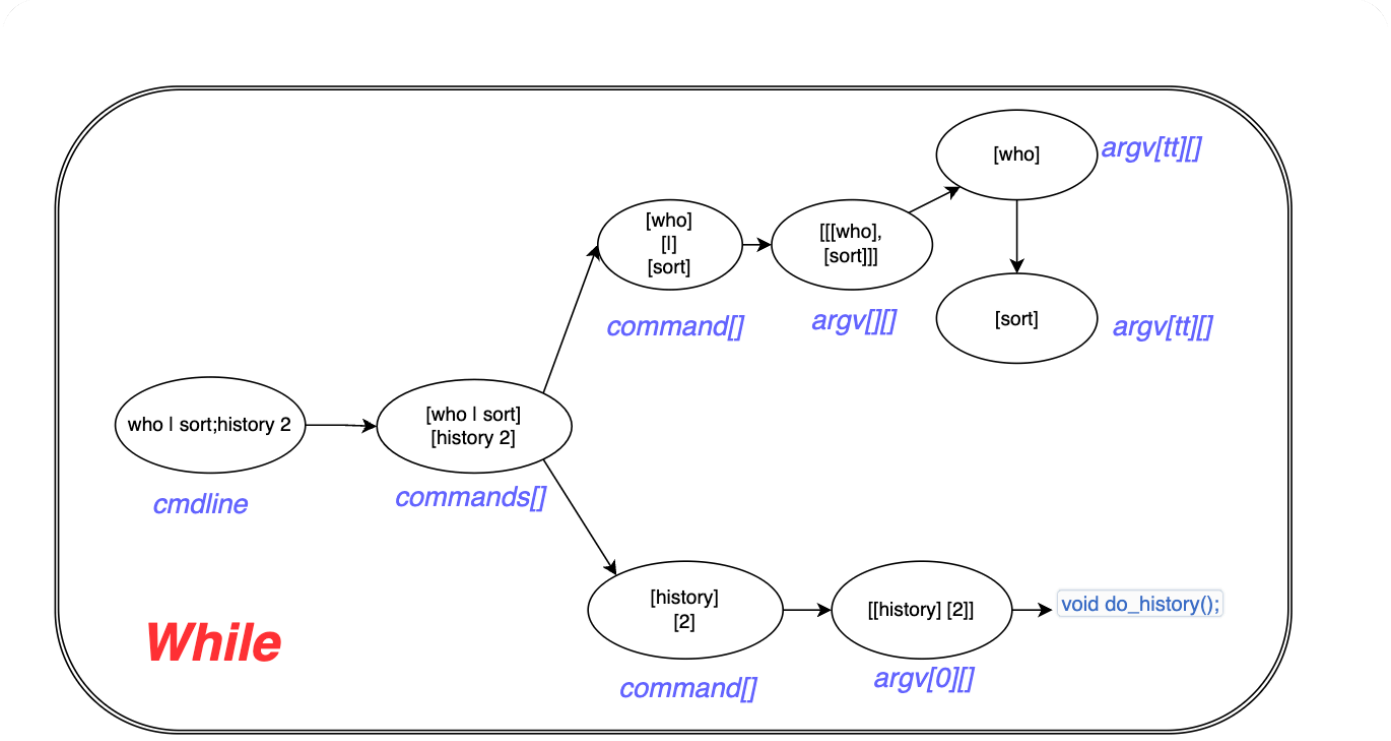
`void read_command();` //读取多种命令函数，主要处理“；”

`void parse_command();` //处理一条命令command

`void analazy_command();` //处理一条命令的多个参数，包括是否含有管道，输入输出定向



其中命令解析的过程与对应变量如下：



4.3.1 read_command() 处理多命令separator=";"

读入终端的命令,对终端的命令使用“；”进行划分，对多条命令进行划分；

```
void read_command()
{
    //处理cmdline，根据separator把命令变成commands【】
    getline(cin, cmdline); // input string with ' '
    typedef string::size_type string_size;
    history.push_back(cmdline);
    string_size i = 0;
    while (i != cmdline.size())
    {
        int flag = 0;
        while (i != cmdline.size() && flag == 0)
        {
            flag = 1;
            for (string_size x = 0; x < separator.size(); ++x)
                if (cmdline[i] == separator[x])
                {
                    ++i;
                    flag = 0;
                    break;
                }
        }
```

```
            flag = 0;
            string_size j = i;
            while (j != cmdline.size() && flag == 0)
            {
                for (string_size x = 0; x < separator.size(); ++x)
                    if (cmdline[j] == separator[x])
                    {
                        flag = 1;
                        break;
                    }
                if (flag == 0)
                    ++j;
            }
            if (i != j)
            {
                commands.push_back(cmdline.substr(i, j - i));
                i = j;
            }
        }
    }
```

4.3.2 parse_command() 单命令解析

串行处理cmds，对单条命令进行参数解析，从commands到command

```
    void parse_command()
    {
        //串行处理处理cmds
        for (int i = 0; i < commands.size(); i++)
        {
            for(int i=0;i<100;i++) command[i]=0;
        arg=0;//初始化参数个数
            string temp_command;
            stringstream input2(commands[i]); // string stream initialize 不按照空格划分
            //处理每一条commands【】，每一条命令可能有多参数，用child_commands【】来解析一条命令多
    参数
            while (input2 >> temp_command)
            {
                command[arg]=new char[temp_command.size()+1];
                strcpy(command[arg++],temp_command.c_str());
            }
            if(temp_command.compare("exit")==0){
                do_exit();
            }
            if (arg>0)
```

```
        {
            //这个函数就是处理分开参数之后的一条命令child_commands
            analazy_command();
            execute_command();
        }
    }
}
```

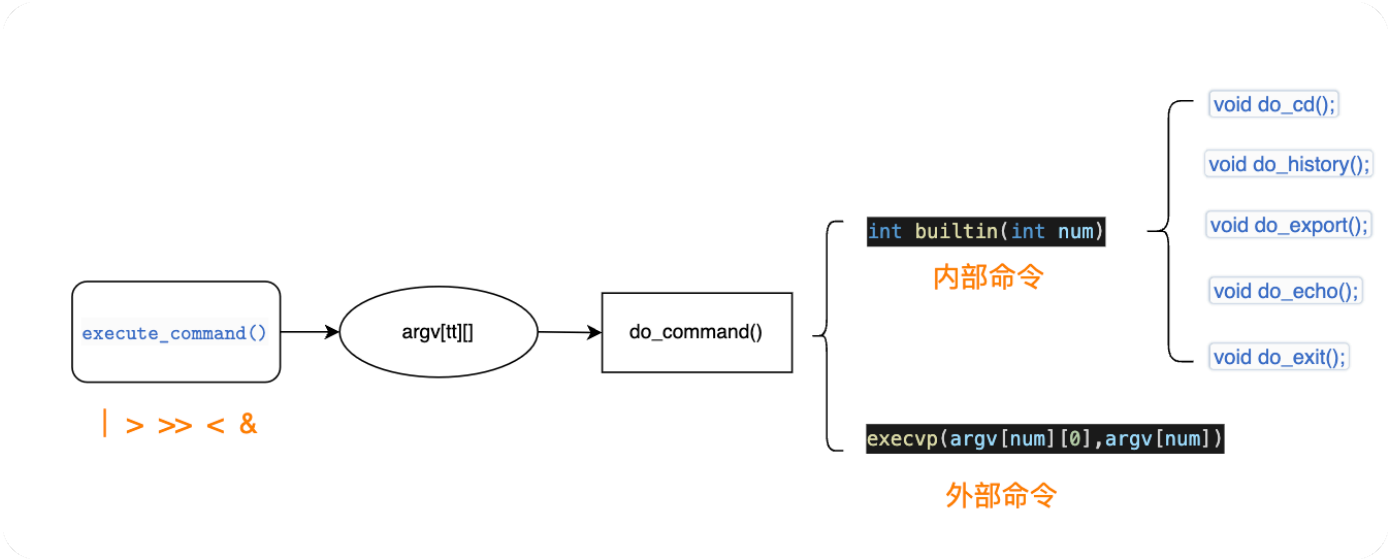
4.3.3 analazy_command() 解析命令是否存在管道；后台；重定向

```
//解析命令
void analazy_command() {
  ar=0;
    tt=0;
  for(int i=0;i<100;i++) {
    flag[i][0]=flag[i][1]=flag[i][2]=0;
    file[i][0]=file[i][1]=0;
        back_flag=0;
    for(int j=0;j<100;j++) {
      argv[i][j]=0;
    }
  }
  for(int i=0;i<arg;i++) argv[0][i]=command[i];//初始化第一个参数
  argv[0][arg]=NULL;
  int a=0;//当前命令参数的序号
  for(int i=0;i<arg;i++) {
        //判断是否存在管道
    if(strcmp(command[i],"|")==0) {//c语言中字符串比较只能用strcmp函数
      //printf("遇到 | 符号\n");
      argv[ar][a++]=NULL;
      ar++;
      a=0;
    }
    else if(strcmp(command[i],"<")==0) {//存在输入重定向
      flag[ar][0]=1;
      file[ar][0]=command[i+1];
      argv[ar][a++]=NULL;
    }
    else if(strcmp(command[i],">")==0) {//没有管道时的输出重定向
      flag[ar][1]=1;
      file[ar][1]=command[i+1];
      argv[ar][a++]=NULL;//考虑有咩有输入重定向的情况
    }
        else if(strcmp(command[i],">>")==0){
            flag[ar][2]=1;
```

```
            file[ar][1]=command[i+1];
            argv[ar][a++]=NULL;
        }
        else if(strcmp(command[i],"&")==0){
            back_flag=1;
            argv[ar][a++]=NULL;
        }
        else argv[ar][a++]=command[i];
    }

}
```

4.4 Execute Input 执行命令



4.4.1 execute_command()实现多管道，后台，重定向

- 先判断是否存在管道，如果有管道，则需要用多个命令参数，并且创建新的子进程;
- 再判断是否存在后台运行符号，存在则重定向输入输出，对父进程发出信号，后台运行。

```
int execute_command()
{
    int pid;
    pid=fork();//创建的子进程
  if(pid<0) {
    perror("fork error\n");
        exit(0);
  }
  //先判断是否存在管道，如果有管道，则需要用多个命令参数，并且创建新的子进程。否则一个命令即可
  else if(pid==0) {
        if(back_flag){
```

```
        //再判断是否存在后台运行符号，存在则重定向输入输出，对父进程发出信号，后台运行。
        freopen( "/dev/null", "w", stdout );
        freopen( "/dev/null", "r", stdin );
        signal(SIGCHLD,SIG_IGN);
    }
if(!ar) {//没有管道
  if(flag[0][0]) {//判断有无输入重定向
    close(0);
    int fd=open(file[0][0],O_RDONLY);
  }
  if(flag[0][1]) {//判断有无输出重定向
    close(1);
    int fd2=open(file[0][1],O_WRONLY|O_CREAT|O_TRUNC,0666);
  }
        if(flag[0][2]){
            close(1);
    int fd2=open(file[0][1],O_WRONLY|O_CREAT|O_APPEND,0666);
        }
  if(do_command(0)==-1){
            printf("command error! please retry!\n");
            exit(0);

        }
}
else {//有管道
  for(tt=0;tt<ar;tt++) {
    int pid2=fork();
    if(pid2<0) {
      perror("fork error\n");
      exit(0);
    }
    else if(pid2==0) {
      if(tt) {//如果不是第一个命令，则需要从共享文件读取数据
                    close(0);
          int fd=open(f,O_RDONLY);//输入重定向
                }
                if(flag[tt][0]) {
        close(0);
        int fd=open(file[tt][0],O_RDONLY);
      }
      if(flag[tt][1]) {
        close(1);
        int fd=open(file[tt][1],O_WRONLY|O_CREAT|O_TRUNC,0666);
      }
                if(flag[tt][2])
```

```
                {
                    close(1);
          int fd=open(file[tt][1],O_WRONLY|O_CREAT|O_APPEND,0666);
                }
                close(1);
                remove(f);//由于当前f文件正在open中，会等到解引用后才删除文件
                int fd=open(f,O_WRONLY|O_CREAT|O_TRUNC,0666);
        if(do_command(tt)==-1) {
                    printf("command error! please retry!\n");
                    exit(0);
                }
      }
      else {//管道后的命令需要使用管道前命令的结果，因此需要等待
        waitpid(pid2,NULL,0);
      }
    }
          //接下来需要执行管道的最后一条命令
    close(0);
    int fd=open(f,O_RDONLY);//输入重定向
    if(flag[tt][1]) {
      close(1);
      int fd=open(file[tt][1],O_WRONLY|O_CREAT|O_TRUNC,0666);
    }
          else if(flag[tt][2])
          {
              close(1);
              int fd=open(file[tt][1],O_WRONLY|O_CREAT|O_APPEND,0666);
          }
    if(do_command(tt)==-1) {
              printf("command error! please retry!\n");
              exit(0);
          }
  }
}
//father
else {
      if(back_flag)
      {
          printf("[process id %d]\n",pid);
          return 1;
      }
      waitpid(pid,NULL,0);
}
  return 1;
```

```
}
```

4.4.2 do_command()执行命令

这一步分成shell内部命令与shell的外部命令

```
int do_command(int num){
    if(builtin(num))
        return SUCCESS;
    int result=execvp(argv[num][0],argv[num]);
    return result;
}
```

4.4.2.1 内部命令builtin(num)

使用句柄实现内部命令。

主要实现的功能有：`exit`,`cd`,`echo`,`export`,`history`等

```
typedef void (CMD_HANDLER)(void);
typedef struct builtin_cmd
{
    char *name;
    CMD_HANDLER *handler;
} BUILTIN_CMD;

//内部命令解析,内部命令的结构体数组，用于处理内部命令
BUILTIN_CMD builtins[] = {
        {(char*)"cd", do_cd},
        {(char*)"history",do_history},
        {(char*)"echo",do_echo},
        {(char*)"export",do_export},
        {(char*)"exit",do_exit},
        {NULL, NULL}

};

//内部命令解析
//执行返回1，没有表示0
int builtin(int num){
    int i = 0;
    int found = 0;
```

```
    while(builtins[i].name != NULL){
        if(strcmp(builtins[i].name,argv[num][0])==0) {
            builtins[i].handler();
            found=1;
            break;
        }
        i++;
    }
    return found;
}
```

4.4.2.2 外部命令 execvp()

外部命令的执行:

```
execvp(argv[num][0],argv[num]);
```

5 功能展示

5.1 支持后台符号&

```
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ python3 main.py & > result.txt
```

测试后台执行文件`Main.py`

```
import time

print('...front...')
time.sleep(50)
print('...end...')
```

```
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ ls
main.py  test  zyzshell  zyzshell.cpp
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ python3 main.py & > result.txt
[process id 205136]
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ ps
    PID TTY          TIME CMD
 204265 pts/8    00:00:00 bash
 204323 pts/8    00:00:00 zyzshell
 205136 pts/8    00:00:00 python3
 205161 pts/8    00:00:00 ps
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$
```

程序执行结束后

```
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ ls
main.py  result.txt  test  zyzshell  zyzshell.cpp
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ ps
    PID TTY          TIME CMD
 204265 pts/8    00:00:00 bash
 204323 pts/8    00:00:00 zyzshell
 205136 pts/8    00:00:00 python3 <defunct>
 205520 pts/8    00:00:00 ps
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ cat result
cat: result: No such file or directory
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ cat result.txt
...front...
...end...
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$
```

可以看到，后台执行成功。

5.2 支持串行执行多个命令";"

```
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ ls;ls > a.txt;pwd >> a.txt;ls;cat a.txt
```

运行结果如图：

```
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ ls;ls > a.txt;pwd >> a.txt;ls;cat a.txt
main.py  result.txt  test  zyzshell  zyzshell.cpp
a.txt  main.py  result.txt  test  zyzshell  zyzshell.cpp
a.txt
main.py
result.txt
test
zyzshell
zyzshell.cpp
/home/parallels/minishell
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$
```

5.3 支持管道"|"

使用 tr 命令从 b.txt 文件中获取输入，然后通过管道将输出发送给 sort 与 uniq：

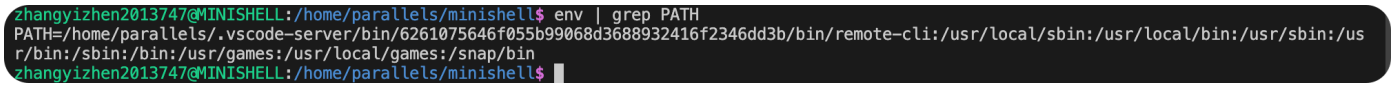
```
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ tr a-z A-Z < b.txt | sort | uniq
```

```
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ cat b.txt
zhangyizhen2013747
zhangyizhen2013747
zhangyizhen2013747
apple
Ubunte
apple
Os
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ tr a-z A-Z < b.txt | sort | uniq
APPLE
OS
UBUNTE
ZHANGYIZHEN2013747
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$
```

5.4 支持环境变量env echo export

5.4.1 env

```
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ env | grep PATH
```

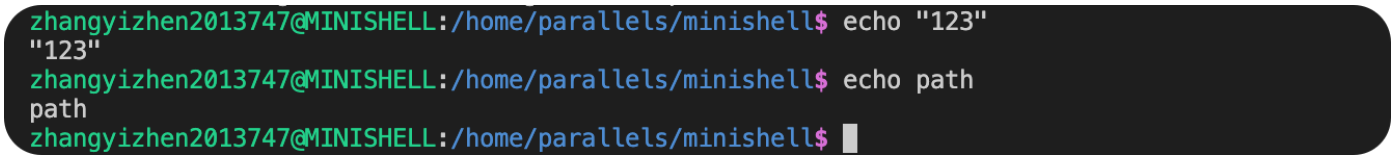


5.4.2 echo 回显

```
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ echo $LANG
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ echo $SHELL
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ echo $PATH
```

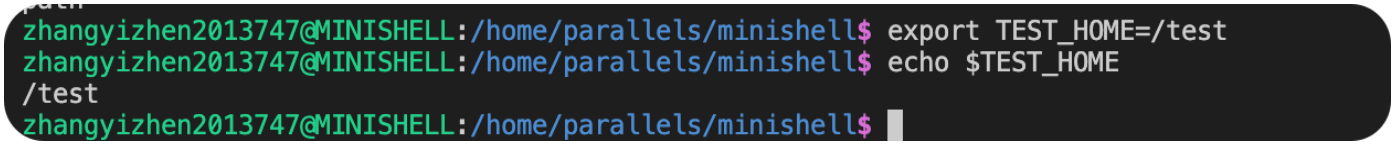
echo回显函数，当没有&符号时或者变量未定义，则会直接输出echo后面的内容

```
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ echo "123"
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ echo path
```



5.4.3 export 设置环境变量

```
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ export TEST_HOME=/test
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ echo $TEST_HOME
```



5.5 支持重定向'>' '>>' '<'

```
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ tr a-z A-Z < b.txt | sort | uniq > b.txt.new
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ cat b.txt.new
```

```
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ tr a-z A-Z < b.txt | sort | uniq > b.txt.new
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ cat b.txt.new
APPLE
OS
UBUNTE
ZHANGYIZHEN2013747
```

zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ tr a-z A-Z < b.txt | sort | uniq >> b.txt.new;cat b.txt.new

```
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ tr a-z A-Z < b.txt | sort | uniq >> b.txt.new;cat b.txt.new
APPLE
OS
UBUNTE
ZHANGYIZHEN2013747
APPLE
OS
UBUNTE
ZHANGYIZHEN2013747
```

5.6 其他内建命令builtin_cmd

```
BUILTIN_CMD builtins[] = {
        {(char*)"cd", do_cd},
        {(char*)"history",do_history},
        {(char*)"echo",do_echo},
        {(char*)"export",do_export},
        {(char*)"exit",do_exit},
        {NULL, NULL}

};
```

5.6.1 do_cd()

zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ cd abc

zhangyizhen2013747@MINISHELL:/home/parallels/minishell/abc$ cd ..

```
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ ls
abc  a.txt  b.txt  b.txt.new  main.py  result.txt  temp.txt  test  zyzshell  zyzshell.cpp
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ cd abc
current dit:/home/parallels/minishell
/home/parallels/minishell/abc
zhangyizhen2013747@MINISHELL:/home/parallels/minishell/abc$ cd ..
/home/parallels/minishell
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$
```

5.6.2 do_history()

zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ history

zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ history 3

```
zhangyizhen2013747@ubuntu-linux-22-04-desktop:~/minishell$ ./zyzshell
----------------welcome to mini shell--------2013747------
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ ls
abc  a.txt  b.txt  b.txt.new  main.py  result.txt  temp.txt  test  zyzshell  zyzshell.cpp
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ ls
abc  a.txt  b.txt  b.txt.new  main.py  result.txt  temp.txt  test  zyzshell  zyzshell.cpp
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ ls
abc  a.txt  b.txt  b.txt.new  main.py  result.txt  temp.txt  test  zyzshell  zyzshell.cpp
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ pwd
/home/parallels/minishell
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ ls;pw
abc  a.txt  b.txt  b.txt.new  main.py  result.txt  temp.txt  test  zyzshell  zyzshell.cpp
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ ls;pwd
abc  a.txt  b.txt  b.txt.new  main.py  result.txt  temp.txt  test  zyzshell  zyzshell.cpp
/home/parallels/minishell
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ history
1 ls
2 ls
3 ls
4 pwd
5 ls;pw
6 ls;pwd
7 history
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ history 3
6 ls;pwd
7 history
8 history 3
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$
```

5.6.3 do_exit()

```
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ exit
```

```
----------------welcome to mini shell--------2013747------
zhangyizhen2013747@MINISHELL:/home/parallels/minishell$ exit
 exit
zhangyizhen2013747@ubuntu-linux-22-04-desktop:~/minishell$
```

5.7 其他

支持`mkdir`,`touch`,`top`,`cat`,`ls`,`ps`,`kill`,`pwd`,`rm`,`date`等命令。

6 实验总结

在本次实践中，我学到了很多新的知识，也巩固了许多知识，对于fork信息，信号处理，shell的内建命令，shell的多管道运行，文件输入输出重定向，后台执行，环境变量的设置与回显等的原理，操作，实践等都又了进一步的理解，经过这学期的操作系统课，以及多次的操作系统实践，包括此次的minishell升级版的实践，我对于操作系统的理解更加深刻，不再畏惧使用无ui的操作系统，对于linux的使用更加熟悉，了解了linux内核与应用之间的关系，对于计算机的运行有了更加深刻的体验。

本次实验实现的shell，实现了shell的基本逻辑：接收用户输入的命令，并对命令进行处理，处理完毕后再将结果反馈给用户，完成度较高。