华东师范大学数据科学与工程学院实验报告

课程名称：操作系统　　**年级**：2020 级　　**上机实践日期**：2022/3/15

指导教师：翁楚良　　**姓名**：张熙翔　　**学号**：10205501427

上机实践名称：Shell Project

一、实验目的

学习 Shell，系统编程，实现一个基本的 Shell

二、实验任务

实现一个基本的 Shell，Shell 能解析的命令行如下:

1. 带参数的程序运行功能（ls，vi，grep 等）。
2. 重定向功能，将文件作为程序的输入/输出
   （1）> 表示覆盖写
   （2）>> 表示追加写
   （3）< 表示文件输入
3. 管道符号 |，能在程序间传递数据。
4. 后台符号&，表示此命令将以后台运行的方式执行。
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

物理机：Windows10
虚拟机：Minix3
虚拟机软件：VMware

四、实验过程

1．前置知识：

（1）Shell 主体：Shell 主体结构是一个 while 循环，不断接受用户键盘输入行并给出反馈。Shell 将输入的命令行进行解析，根据命令名称分为两类分别处理，即 Shell 内置命令和 program 命令。识别 为 shell 内置命令后，执行对应操作。接受 program 命令后，利用 Minix 自带的程序创建一个或多个新进程，并等待进程结束。如果末尾包含&参数，则为后台任务，Shell 不等待进程结束，直接返回。

（2）unix 系统编程：fork, dup2, chdir, getcwd, signal, execvp, open, freopen, pipe, waitpid

等函数；重定向，管道等指令。

2．实验思路：

（1）内置命令：

• cd：利用系统调用 chdir 函数改变目录，调用 getcwd 函数获取当前工作目录。

• exit：退出 Shell 的 while 循环。

• history：将 Shell 中输入的命令行用二维数组存储，根据参数输出相应的历史命令。

• mytop：在 minix 系统/proc 文件夹中通过 fopen/fscanf 获取进程信息，输出内存使用情况和 CPU 使用百分比。

（2）program 命令：

• 重定向 覆盖写>：调用 open 函数得到文件描述符，清空文件内容，调用 dup2（fd，1）函数将文件描述符映射到标准输出。

• 重定向 追加写>>：调用 open 函数得到文件描述符，保留文件内容，调用 dup2（fd，1）函数将文件描述符映射到标准输出。

• 重定向 文件输入<：调用 open 函数得到文件描述符，调用 dup2（fd，0）函数将文件描述符映射到标准输入。

• 后台运行：调用 signal(SIGCHLD, SIG_IGN)，将子进程的标准输入、输出映射到/dev/null，Shell 无需等待子进程结束。

• 管道：调用 pipe 函数创建管道，在子进程中调用 dup2(fd[1],1)函数将管道写端映射到标准输出，进程的输出写入管道。在父进程中，等待子进程结束并回收，调用 dup2(fd[0],0)函数将管道读端 fd[0]映射到标准输入，从管道中读入数据并执行。

3．代码实现：

（1）宏定义，全局变量以及函数声明

#define 最大命令数量，最大输入命令字符数，每条命令最大长度；全局变量 his_cnt 计数历史命令数量，二维数组 his 存放历史命令，path 指针初始化指向 NULL。

```
#define MAXLINE 100
#define MAXARGS 100
#define M 100

char his[M][M];
int his_cnt = 0; //历史命令计数
char *path = NULL;

void doCommand(char *cmdline);
int parseline(const char *cmdline, char **argv);
int builtin_cmd(char **argv);
void guandao(char *process1[], char *process2[]);
void mytop();
```

（2）主函数

获取当前工作路径，打印 Shell 提示符，输入命令存入 cmdline 数组中，如果没输入进入下一循环，将命令存储到历史命令二维数组 his 中，调用 doCommand 函数解析命令，最后刷新缓冲区。

```
int main(int argc, char **argv)
{
    char c;
    char cmdline[MAXLINE];

    while (1)
    {
        path = getcwd(NULL, 0);
        printf("ZhangXixiang_shell>%s# ", path);
        fflush(stdout);
        if (fgets(cmdline, MAXLINE, stdin) == NULL)
        {
            continue;
        }
        for (int i = 0; i < M; i++)
        {
            his[his_cnt][i] = cmdline[i];
        }
        his_cnt = his_cnt + 1;
        doCommand(cmdline);
        fflush(stdout);
    }
    exit(0);
}
```

（3）doCommand 函数

调用 parseline 函数对命令进行分割，获取命令和参数，并判断是都是后台命令。执行 builtin_cmd 函数，如果是内置命令直接返回。将 program 命令分为六种 case，分别为：

case0: 未出现重定向，管道和后台运行命令

case1: >

case2: <

case3: |

case4: &

case5: >>

根据参数列表 argv 判断所属情况，用 switch-case 结构处理每种情况：

case:0：fork 子进程 execvp 运行。

case1：包含重定向输出，fork 子进程，在子进程中调用 open 函数得到 file 文件描述符 fd，参数中将文件清空，调用 dup（fd，1）将 file 映射到标准输出，调用 execvp 执行重定向符号前的指令，在父进程调用 waitpid 父进程等待子进程结束并回收。

case2：包含重定向输入，利用参数列表得到重定向符号后面的文件名，fork 一个子进程，在子进程中调用 open 得到 file 的文件描述符，调用 dup2（fd，0）将 file 映射到标准输入，调用 execvp 执行符号前面的指令，在父进程中 waitpid 等待子进程结束并回收。

case3：命令包含管道，利用参数列表得到管道符号前面的和后面的命令参数，fork 一个子进程，子进程中调用 pipeline 函数实现管道。

case4：fork 一个子进程，调用 signal(SIGCHLD, SIG_IGN)使 minix 接管此进程，调用 open 得到/dev/null/ 的文件描述符（黑洞，通常被用于丢弃不需要的输出流，或作为用于输入流的空文件），映射到标准输入输出，最后 execvp 执行命令。

case5：包含重定向追加写，fork 子进程，在子进程中调用 open 函数得到 file 文件描述符 fd，参数中保留文件内容，调用 dup（fd，1）将 file 映射到标准输出，调用 execvp 执行重定向符号前的指令，在父进程调用 waitpid 父进程等待子进程结束并回收。

```
void doCommand(char *cmdline)
//内置命令、program 命令、后台运行
{
    char *argv[MAXARGS];
    char buf[MAXLINE];
    int bg;
    pid_t pid;
    char *file;
    int fd;
    int status;
    int case_command = 0;
    strcpy(buf, cmdline);

    if ((bg = parseline(buf, argv)) == 1)
    { //后台
        case_command = 4;
    }
    if (argv[0] == NULL)
    {
        return;
    }

    if (builtin_cmd(argv))
        return; //内置命令返回

    int i = 0;
    for (i = 0; argv[i] != NULL; i++)
    {
        if (strcmp(argv[i], ">") == 0)
        {
            if (strcmp(argv[i + 1], ">") == 0)
            {
                case_command = 5;
```

```
            break;
        }
        case_command = 1;
        file = argv[i + 1];
        argv[i] = NULL;
        break;
    }
}

for (i = 0; argv[i] != NULL; i++)
{
    if (strcmp(argv[i], "<") == 0)
    {
        case_command = 2;
        file = argv[i + 1];
        printf("filename=%s\n", file);
        argv[i] = NULL;
        break;
    }
}
char *leftargv[MAXARGS];
for (i = 0; argv[i] != NULL; i++)
{
    if (strcmp(argv[i], "|") == 0)
    {
        case_command = 3;
        argv[i] = NULL;
        int j;
        for (j = i + 1; argv[j] != NULL; j++)
        {
            leftargv[j - i - 1] = argv[j];
        }
        leftargv[j - i - 1] = NULL;
        break;
    }
}

switch (case_command)
{
case 0:
    if ((pid = fork()) == 0)
```

```
        {
            execvp(argv[0], argv);
            exit(0);
        }
        if (waitpid(pid, &status, 0) == -1)
        {
            printf("error\n");
        }
        break;
    case 1:
        /*包含重定向输出*/
        if ((pid = fork()) == 0)
        {
            fd = open(file, O_RDWR | O_CREAT | O_TRUNC, 7777);
            if (fd == -1)
            {
                printf("open %s error!\n", file);
            }
            dup2(fd, 1);
            close(fd);
            execvp(argv[0], argv);
            exit(0);
        }
        if (waitpid(pid, &status, 0) == -1)
        {
            printf("error\n");
        }
        break;
    case 2:
        /*包含重定向输入*/
        if ((pid = fork()) == 0)
        {
            fd = open(file, O_RDONLY);
            dup2(fd, 0);
            close(fd);
            execvp(argv[0], argv);
            exit(0);
        }
        if (waitpid(pid, &status, 0) == -1)
        {
            printf("error\n");
```

```
        }
        break;
    case 3:
        /*命令包含管道*/
        if ((pid = fork()) == 0)
        {
            guandao(argv, leftargv);
        }
        else
        {
            if (waitpid(pid, &status, 0) == -1)
            {
                printf("error\n");
            }
        }
        break;
    case 4: //后台运行
        signal(SIGCHLD, SIG_IGN);
        if ((pid = fork()) == 0)
        {
            signal(SIGCHLD, SIG_IGN);
            close(0);
            open("/dev/null", O_RDONLY);
            close(1);
            open("/dev/null", O_WRONLY);
            execvp(argv[0], argv);
            exit(0);
        }
        //不等待结束
        break;
    case 5: //追加写
        if ((pid = fork()) == 0)
        {
            fd = open(file, O_RDWR | O_CREAT | O_APPEND, 7777);
            if (fd == -1)
            {
                printf("open %s error!\n", file);
            }
            dup2(fd, 1);
            close(fd);
            execvp(argv[0], argv);
```

```
            exit(0);
        }
        if (waitpid(pid, &status, 0) == -1)
        {
            printf("error\n");
        }
        break;

    default:
        break;
    }
    return;
}
```

（4）parseline 函数

解析命令行，得到参数序列，并判断是前台作业还是后台作业。调用 strtok 函数根据空格对命令进行分割，并根据最后一个字符时候是&判断是否为后台命令。

```
int parseline(const char *cmdline, char **argv)
{
    static char array[MAXLINE];
    char *buf = array;
    int argc = 0;
    int bg;

    strcpy(buf, cmdline);
    buf[strlen(buf) - 1] = ' ';
    while (*buf && (*buf == ' '))
        buf++;

    char *s = strtok(buf, " ");
    if (s == NULL)
    {
        exit(0);
    }
    argv[argc] = s;
    argc++;
    while ((s = strtok(NULL, " ")) != NULL)
    {
        argv[argc] = s;
        argc++;
    }
    argv[argc] = NULL;
```

```
    if (argc == 0)
        return 1;

    if ((bg = (*argv[(argc)-1] == '&')) != 0)
    {
        argv[--(argc)] = NULL;
    }
    return bg;
}
```

（5）builtin_cmd 函数

exit：退出 main 函数的 while 循环。

mytop：调用 mytop 函数。

cd：chdir 函数根据参数改变工作目录，path 指向 getcwd 获得的当前的工作目录。

history：打印历史命令记录，如果无参数则全部打印，参数过大进行提示。

```
int builtin_cmd(char **argv)
{
    //内置命令
    if (!strcmp(argv[0], "exit"))
    {
        exit(0);
    }
    if (!strcmp(argv[0], "mytop"))
    {
        mytop();
        return 1;
    }
    if (!strcmp(argv[0], "cd"))
    {
        if (!argv[1])
        { // cd 后面啥也没有
            argv[1] = ".";
        }
        int ret = chdir(argv[1]); //改变目录
        if (ret < 0)
        {
            printf("No such directory!\n");
        }
        else
        {
```

```
            path = getcwd(NULL, 0); //当前目录
        }
        return 1;
    }

    if (!strcmp(argv[0], "history"))
    {
        if (!argv[1])
        { //只输入 history
            for (int j = 1; j <= his_cnt; j++)
            {
                printf("%d ", j);
                puts(his[j - 1]);
            }
        }
        else
        {
            int t = atoi(argv[1]);
            if (his_cnt - t < 0)
            {
                printf("please confirm the number below %d\n", his_cnt);
            }
            else
            {
                for (int j = his_cnt - t; j < his_cnt; j++)
                {
                    printf("%d ", j + 1);
                    puts(his[j]);
                }
            }
        }
        return 1;
    }

    return 0;
}
```

（6）mytop 函数

在/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。在/proc/kinfo 中，查看进程和任务数量。

```
void myTop() {
```

```
FILE *fp = NULL;
char buff[255];

/* 获取内容总体内存大小，空闲内存大小，缓存大小。 */
fp = fopen("/proc/meminfo", "r");   // 以只读方式打开 meminfo 文件
fgets(buff, 255, (FILE*)fp);        // 读取 meminfo 文件内容进 buff
fclose(fp);

// 获取 pagesize
int i = 0, pagesize = 0;
while (buff[i] != ' ') {
    pagesize = 10 * pagesize + buff[i] - 48;
    i++;
}

// 获取 页总数 total
i++;
int total = 0;
while (buff[i] != ' ') {
    total = 10 * total + buff[i] - 48;
    i++;
}

// 获取空闲页数 free
i++;
int free = 0;
while (buff[i] != ' ') {
    free = 10 * free + buff[i] - 48;
    i++;
}

// 获取最大页数量 largest
i++;
int largest = 0;
while (buff[i] != ' ') {
    largest = 10 * largest + buff[i] - 48;
    i++;
}

// 获取缓存页数量 cached
i++;
```

```
    int cached = 0;
    while (buff[i] >= '0' && buff[i] <= '9') {
        cached = 10 * cached + buff[i] - 48;
        i++;
    }

    printf("totalMemory  is %d KB\n", (pagesize / 1024 * total));
    printf("freeMemory   is %d KB\n", pagesize / 1024 * free));
    printf("cachedMemory is %d KB\n", (pagesize / 1024 * cached));

    /* 2. 获取进程和任务数量 */
    fp = fopen("/proc/kinfo", "r");     // 以只读方式打开 meminfo 文件
    memset(buff, 0x00, 255);           // 格式化 buff 字符串
    fgets(buff, 255, (FILE*)fp);       // 读取 meminfo 文件内容进 buff
    fclose(fp);

    // 获取进程数量
    int processNumber = 0;
    i = 0;
    while (buff[i] != ' ') {
        processNumber = 10 * processNumber + buff[i] - 48;
        i++;
    }
    printf("processNumber = %d\n", processNumber);

    // 获取任务数量
    i++;
    int tasksNumber = 0;
    while (buff[i] >= '0' && buff[i] <= '9') {
        tasksNumber = 10 * tasksNumber + buff[i] - 48;
        i++;
    }
    printf("tasksNumber = %d\n", tasksNumber);

    return;
}
```

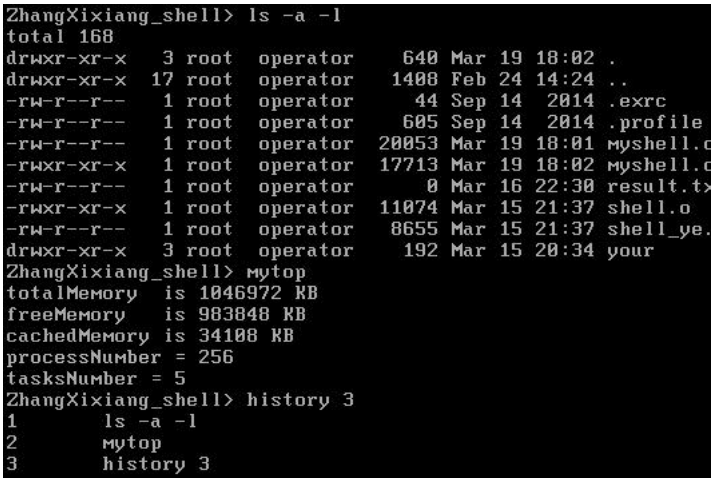
（7）pipe_line 管道函数

调用 pipe 函数创建一个管道 fd[2]，fork 一个子进程，关闭管道读端 fd[0]和文件描述符 1，fd[1]管道写入端，映射到标准输出 1，关闭写端避免堵塞，执行前部分指令，结果输出到管道，父进程中关闭管道读端 fd[1]和文件描述符 0，fd[0]管道读入端，映射到标准输入 0，关闭读端避免堵塞，等待子进程结束，继续执行。

```
void pipe_line(char *process1[], char *process2[])
{
    int fd[2];
    pipe(&fd[0]);

    int status;
    if ((pid_t pid = fork()) == 0)
    {
        close(fd[0]);
        close(1);
        dup(fd[1]);
        close(fd[1]);
        execvp(process1[0], process1);
    }
    else
    { //父进程中
        close(fd[1]);
        close(0);
        dup(fd[0]);
        close(fd[0]);
        waitpid(pid, &status, 0);
        execvp(process2[0], process2);
    }
}
```

五、实验结果



```
# ./shell.o
ZhangXixiang_shell> /root# cd your/path
ZhangXixiang_shell> /root/your/path# ls -a -l
total 24
drwxr-xr-x  2 root  operator  192 Mar 15 21:38 .
drwxr-xr-x  3 root  operator  192 Mar 15 20:34 ..
-rw-r--r--  1 root  operator  166 Mar 19 17:52 result.txt
ZhangXixiang_shell> /root/your/path# ls -a -l > result.txt
ZhangXixiang_shell> /root/your/path# vi result.txt_
```

```
~
~
~
~
~
~
~
~
~
~
~
:q
ZhangXixiang_shell> /root/your/path# grep a < result.txt
filename=result.txt
total 16
drwxr-xr-x  2 root  operator  192 Mar 15 21:38 .
drwxr-xr-x  3 root  operator  192 Mar 15 20:34 ..
-rw-r--r--  1 root  operator    0 Mar 19 17:52 result.txt
ZhangXixiang_shell> /root/your/path# ls -a -l | grep a
total 24
drwxr-xr-x  2 root  operator  192 Mar 15 21:38 .
drwxr-xr-x  3 root  operator  192 Mar 15 20:34 ..
-rw-r--r--  1 root  operator  166 Mar 19 17:52 result.txt
ZhangXixiang_shell> /root/your/path# cat result.txt &
```