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：**2019 级　　**上机实践成绩：**

指导教师：翁楚良　　**姓名：**叶秋雨

上机实践名称：Shell Project　　**学号：**10184102103　　**上机实践日期：**2021.3.2

上机实践编号：

一、 目的

理解 shell 程序的设计方法，系统编程，实现一个基本的 shell。

二、 内容与设计思想

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、shell 内置命令

（1）cd +路径名，改变工作路径

shell 本身是一个程序，启动时 Minix 会分配一个当前工作目录，利用 chdir 系统调用改变 shell 的工作目录，并调用 getcwd 函数获取当前工作目录的路径。

（2）history n，显示最近执行的 n 条指令

利用一个二维数组保存 shell 每次输入的命令行，根据指令打印出相应数量的行。

（3）exit，shell 退出

退出 shell 的 while 循环，结束 shell 的 main 函数。

（4）mytop，输出内存使用情况和 CPU 使用百分比

在 minix 系统/proc 文件夹中通过 fopen/fscanf 获取进程信息。

3、program 命令

（1）运行程序

利用 fork 调用创建子进程，利用 execvp 调用加载并运行，对 shell 中的命令进行处理，处理完之后新进程结束，shell 利用 wait/waitpid 调用等待进程结束并回收。

（2）重定向

重定向输入(>)：

调用 open 得到文件描述符 fd，再调用 dup2(fd,1)函数，将文件描述符映射到标准输出。最后，调用 execvp 运行程序。

重定向输出(<)：

调用 open 得到文件描述符 fd，再调用 dup2(fd,0)函数，将文件描述符映射到标准输入。最后，调用 execvp 运行程序。

（3）管道

利用 pipe 函数创建一个管道 fd[2]，再 fork 一个子进程，在子进程中调用 dup2(fd[1],1)函数将管道写端 fd[1]映射到标准输出，调用 execvp 运行程序，将进程的输出写入管道。在父进程中，等待子进程结束并回收，再调用 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。

（4）后台运行

将子进程的标准输入、输出映射到/dev/null，屏蔽键盘和控制台。调用 signal(SIGCHLD, SIG_IGN)，使得 Minix 接管此进程，shell 不用等待子进程结束直接运行下一条命令。

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、实现的主要函数：

函数	功能
void exeCommand(char *cmdline);	实现内置命令、program 命令和后台运行等功能。
int parseline(const char *cmdline, char **argv);	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
void pipeline(char *process1[],char *process2[]);	实现管道，完成进程间的通信。
int builtin_cmd(char **argv);	实现内置命令 exit、history n、cd、mytop 的具体实现。
void getkinfo();	在/proc/kinfo 中查看进程和任务的数量
int print_memory();	在/proc/meminfo 中查看内存信息，计算出内存大小并打印
void get_procs();	创建 struct proc 数组，为每个任务分配一个 struct proc 结构体，保存信息
void parse_dir()	读取目录下每一个文件信息，并获得进程号
void parse_file(pid_t pid);	在/proc/pid/psinfo 中，查看进程 pid 的信息，并保存每个文件对应的结构体 struct proc 中
u64_t cputicks(struct proc *p1, struct proc *p2, int timemode);	利用时间差，计算每个任务、进程的 cputicks
void print_procs(struct proc *proc1,struct proc *proc2,int cputimemode);	计算总体 CPU 使用占比并打印结果

1）int parseline(const char *cmdline, char **argv);

利用 strtok 函数，根据空格对命令行进行划分，得到 argv 参数序列。同时判断命令行最后是否带有参数'&'，若有则为后台作业，反之，则为前台作业。

2）void exeCommand(char *cmdline);

• 调用 parseline 函数解析命令行并判断前、后台作业。

• 调用 builtin_cmd 函数判断是否为内置命令，若是内置命令则执行相应的内置命令操作；若不是内置命令，则返回继续执行。

• 判断命令行中是否包含”>”, “<”, “|”, “&”。在利用 switch 语句，对每种情况做出相应的处理。

（1）命令不包含管道、重定向、后台运行

• 调用 fork 函数，创建一个新的子进程

• 在子进程中，调用 execvp 函数运行程序

• 在父进程中，调用 waitpid 函数，父进程等待子进程结束并回收

```
pid = Fork();
if (pid == 0)
{
    execvp(argv[0], argv);
    exit(0);
}
if (waitpid(pid, &status, 0) == -1)
{
    printf("wait for child process error\n");
}
```

（2）命令包含重定向输出

• 利用参数列表，得到重定向符后的文件名 file

• 调用 fork 函数，创建一个新的子进程

• 在子进程中调用 open 函数得到 file 的文件描述符 fd；再调用 dup2(fd, 1)，将 file 映射到标准输出；最后，调用 execvp 执行重定向符前的指令

• 在父进程中，调用 waitpid 函数，父进程等待子进程结束并回收

```
pid = Fork();
if (pid == 0)
{
    fd = open(file, O_RDWR | O_CREAT | O_TRUNC, 0644);
    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("wait for child process error\n");
```

```
    }
```

（3）命令包含重定向输入

- 利用参数列表，得到重定向符后的文件名 file
- 调用 fork 函数，创建一个新的子进程
- 在子进程中，调用 open 函数得到 file 的文件描述符 fd；再调用 dup2(fd,0)，将 file 映射到标准输入；最后，调用 execvp 执行重定向符前的指令
- 在父进程中调用 waitpid 函数，父进程等待子进程结束并回收

```
    pid = Fork();
    if (pid == 0)
    {
        fd = open(file, O_RDONLY);
        dup2(fd, 0);
        close(fd);
        execvp(argv[0], argv);
        exit(0);
    }
    if (waitpid(pid, &status, 0) == -1)
    {
        printf("wait for child process error\n");
    }
```

（4）命令包含管道

- 利用参数列表，得到管道符之前的命令参数和管道符之后的命令参数
- 调用 fork 函数，创建一个新的子进程
- 在子进程中调用辅助函数 pipeline 实现管道
- 在父进程中，调用 waitpid 函数，父进程等待子进程结束并回收

```
    if((pid = fork()) < 0){
        printf("fork error\n");
        return ;
    }
    if (pid == 0)
    {
        pipeline(argv,leftargv);
    }
    else
    {
        if (waitpid(pid, &status, 0) == -1)
```

```
        {
            printf("wait for child process error\n");
        }
    }
```

（5）后台任务

- 调用函数 fork 函数，创建一个新的子进程
- 子进程调用 signal(SIGCHLD, SIG_IGN)，使 Minix 接管此进程；在调用 open 函数得到”/dev/null/”的文件描述符，并映射到标准输入输出；最后 execvp 执行命令

```
    pid = Fork();
    if (pid == 0)
    {
        signal(SIGCHLD, SIG_IGN);
        int a = open("/dev/null", O_RDONLY);
        dup2(a, 0);
        dup2(a, 1);
        execvp(argv[0], argv);
        exit(0);
    }
```

3）void pipeline(char *process1[],char *process2[]);

- 调用 pipe 函数创建一个管道 fd[2]
- 调用 fork 函数创建一个子进程
- 在子进程中，分别调用函数 close(fd[0])和 close(1)关闭管道读端 fd[0]和文件描述符 1；再调用 dup(fd[1])将管道的写端映射到标准输出；然后，调用函数 close(fd[1])关闭管道写端，避免堵塞；最后，调用 execvp 执行管道前部分指令，其结果将输出到管道中。
- 在父进程中，分别调用函数 colse(fd[1])和 close(0)关闭管道写端 fd[1]和文件描述符 0；再调用 dup(fd[0])将管道的读端映射到标准输入；然后，调用函数 close(fd[0])关闭管道读端，避免堵塞；最后，调用 execvp 执行管道后部分指令，从管道中读入数据。

```
void pipeline(char *process1[],char *process2[]){
    int fd[2];
    pipe(&fd[0]);
    int status;
    pid_t pid;
    pid=Fork();
    if(pid==0){
        close(fd[0]);
```

```
        close(1);
        dup(fd[1]);
        close(fd[1]);
        execvp(process1[0],process1);
    }else{
        close(fd[1]);
        close(0);
        dup(fd[0]);
        close(fd[0]);
        execvp(process2[0],process2);
    }
}
```

4）int builtin_cmd(char **argv);

（1）exit

• exit(0)，退出 shell 的 while 循环，结束 shell 的 main 函数

```
    if (!strcmp(argv[0], "exit"))
    {
        exit(0);
    }
```

（2）cd

• 调用 chdir 函数，改变工作目录

• 调用 getcwd 函数，获取当前所在目录

```
if (!strcmp(argv[0], "cd"))
    {
        if (!argv[1])
        {
            argv[1] = ".";
        }
        int ret;
        ret = chdir(argv[1]); //改变工作目录
        if (ret < 0)
        {
            printf("No such directory!\n");
        }
        else
        {
            path = getcwd(NULL, 0); //利用 getcwd 取当前所在目录
        }
        return 1;
    }
```

（3）history n

• 在 main 函数中将输入的指令保存在二维数组 history 中

• 根据参数 n，打印出最近输入的 n 条指令；若仅输入 history 未带参数 n 则打印出所有历史指令；若 n 大于历史指令数量则输出错误信息 history error

```
if (!strcmp(argv[0], "history"))
    {
        if (!argv[1])
        { //当只输入 history 时，打印已有的所有指令
            for (int j = 1; j <= n_his; j++)
            {
                printf("%d ", j);
                puts(history[j - 1]);
            }
        }
        else
        {
            int t = atoi(argv[1]);
            if (n_his - t < 0)
            { //如果 history 后未带参数或带的参数大于已有指令数
                 printf("history error\n");
            }
            else
            {
                for (int j = n_his - t; j < n_his; j++)
                {
                    printf("%d ", j + 1);
                    puts(history[j]);
                }
            }
        }
        return 1;
    }
```

（4）mytop

• 调用辅助函数 getkinfo，查看进程和任务总数

• 调用辅助函数 print_memory，查看内存信息，从计算出总体内存大小、空闲内存大小、缓存大小

• 根据进程和任务总数，分别调用两次辅助函数 get_procs 为每一个进程和结构体分配一个 struct proc 结构体，得到两个 struct proc 数组 proc1、proc2

• 调用辅助函数 print_procs，打印出 CPU 使用百分比情况

```
if (!strcmp(argv[0], "mytop"))
    {
        int cputimemode = 1;
        getkinfo();
        print_memory();
        //得到 prev_proc
        get_procs();
        if (prev_proc == NULL)
        {
            get_procs();//得到 proc
        }
        print_procs(prev_proc, proc, cputimemode);
        return 1;
    }
```

5）void getkinfo();

调用 fopen 函数打开文件"/proc/kinfo"，读入进程和任务数量，相加得到进程和任务总数 nr_total。

```
    if ((fp = fopen("/proc/kinfo", "r")) == NULL)
    {
        fprintf(stderr, "opening /proc/kinfo failed\n");
        exit(1);
    }

    if (fscanf(fp, "%u %u", &nr_procs, &nr_tasks) != 2)
    {
        fprintf(stderr, "reading from /proc/kinfo failed");
        exit(1);
    }
    fclose(fp);
    nr_total = (int)(nr_procs + nr_tasks);
```

6）int print_memory();

调用 fopen 函数打开文件"/proc/meminfo"，查看内存信息，分别读入页面大小 pagesize、总页数量 total、空闲页数量 free、最大页数量 largest、缓存页数量 cached。根据公式 $(pagesize * total)/1024$ 算出内存大小，同理可算出其他页内存的大小，并打印。

```
    if ((fp = fopen("/proc/meminfo", "r")) == NULL)
    {
        return 0;
    }
```

```
if (fscanf(fp, "%lu %lu %lu %lu %lu", &pagesize, &total, &free, &largest, &cached) != 5)
{
    fclose(fp);
    return 0;
}
fclose(fp);
printf("main memory: %ldk total,%ldk free,%ldk contig free,%ldk cached\n", (pagesize * total) / 1024,
       (pagesize * free) / 1024, (pagesize * largest) / 1024, (pagesize * cached) / 1024);
```

7）void get_procs();

根据进程和任务总数 nr_total，为每个任务和进程都分配一个 struct proc 结构体来保存相关信息，得到一个 struct proc 数组 proc。

```
    proc = malloc(nr_total * sizeof(proc[0])); //struct proc 的大小
    if (proc == NULL)
    {
        fprintf(stderr, "Out of memory!\n");
        exit(0);
    }
```

8）void parse_dir()

调用 opendir 函数打开目录"/proc"，调用 readdir 函数读取目录下每个文件的信息，并保存在每个文件对应的结构体 struct proc 中。每个文件都会得到一个 struct dirent 结构体的返回值储存了文件信息，结构体成员包括索引节点号、在目录文件中的偏移、文件名长、文件类型、文件名。再调用 strtol 函数由文件名获取进程号，最后调用 parse_file 函数查看进程信息。循环执行以上步骤，直至目录中的所有文件都被读取。

9）void parse_file(pid_t pid);

调用 fopen 函数打开文件"/proc/pid/psinfo"，查看进程 pid 的信息。依次读入版本 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 等。利用&proc[slot]获取相应的 struct proc，并将信息存储在结构体中。

若 type 是 task 则将 struct proc 的结构体成员 p_flags 倒数第二位标记为 1(p->p_flags |= IS_TASK)，若 type 是 system 则将 p_flags 倒数第三位标记为 1(p->p_flags |= IS_SYSTEM)。若进程状态 state 不是在运行状态则将 p_flags 倒数第四位标记为 1(p->p_flags |=BLOCKED)。

然后，调用 make64(cycles_lo, cycles_hi)根据高低频计算出 cputicks。最后，将 p->flags 最后一位标记为 1，表示已访问过该进程信息(p->p_flags |=USED)。

10）u64_t cputicks(struct proc *p1, struct proc *p2, int timemode);

(p2->p_cpucycles[i] — p1->p_cpucycles[i])计算出 task i 在一段时间内的 cputicks

```
if (p1->p_endpoint == p2->p_endpoint)
{
    t = t + p2->p_cpucycles[i] - p1->p_cpucycles[i];
}
else
{ //否则 t 直接加上 p2
    t = t + p2->p_cpucycles[i];
}
```

11）void print_procs(struct proc *proc1,struct proc *proc2,int cputimemode);

对 struct proc 数组中每个 proc，调用函数 cputicks 计算他们的 CPU 时钟周期。并将数组中每个任务的 cputicks 累加得到 total cputicks。在利用 p_flags 标记来判断是 systemticks 还是 usertickes。最后分别除以 total cputicks 得到 CPU 使用百分比。

2、实验结果

```
# ./myshell
myshell> /root# cd your/path
myshell> /root/your/path# ls -a -l
total 24
drwxr-xr-x  2 root  operator  1024 Mar 24 15:49 .
drwxr-xr-x  3 root  operator   256 Mar 24 15:39 ..
-rw-r--r--  1 root  operator     0 Mar 24 14:54 grep
-rw-r--r--  1 root  operator   840 Mar 24 15:44 result.txt
myshell> /root/your/path# ls -a -l > aaa.txt
myshell> /root/your/path# vi aaa.txt
myshell> /root/your/path# grep aaa < aaa.txt
filename=aaa.txt
-rw-r--r--  1 root  operator     0 Mar 24 15:50 aaa.txt
myshell> /root/your/path# ls -a -l | grep aaa
-rw-r--r--  1 root  operator   278 Mar 24 15:50 aaa.txt
myshell> /root/your/path# cat aaa.txt &
myshell> /root/your/path# mytop
main memory: 1046972k total,980700k free,853300k contig free,35856k cached
CPU states:   0.05% user,   0.36% system,   0.00% kernel,   0.00% idle
myshell> /root/your/path# history 4
6 ls -a -l | grep aaa

7 cat aaa.txt &

8 mytop

9 history 4

myshell> /root/your/path# exit
#
```

五、总结

通过本次的 project，基本掌握了简单的 Minix 环境下的系统编程，也让我对 shell、操作系统和内核的关系有了更好的理解，除此之外对管道、重定向、创建子进程等实现原理都有了更清楚的认识，也能更熟练地使用一些系统调用。

在刚动手开始编写时，觉得十分困难，不知道如何下手。但仔细了解学习了 shell 的工作原理和每个命令的实现后，也逐步完成了整个 shell 的编写。不过本次实验还未直接深入与操作系统打交道，只是利用系统调用请求操作系统服务，但是也让我学习到了很多。并且，在实现 mytop 内置命令时，阅读源码的过程也让算是对后续的 project 有一些初尝试。虽然完成的整个过程是艰难的，尤其是 debug 的时候，但是实验成功实现的最终成果也是很有成就感的。对于操作系统今后的学习，除了教材内容的书面理解，自己动手进行操作实践可能是更加重要的。