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：**2018 级　　**上机实践成绩：**

指导教师：翁楚良　　**姓名：**郑佳辰

上机实践名称：Shell 及系统调用　　**学号：**10182100359　　**上机实践日期：**2020/3/17

上机实践编号：　　~2020/4/7

一、 目的

学习 Shell 和系统编程，实现一个基本的 Shell。

二、 内容与设计思想

Shell 主体结构是一个 while 循环，不断地接受用户键盘输入行并给出反馈。Shell 将输入行分解成单词序列，根据命令名称分为二类分别处理，即 shell 内置命令（例如 cd，history，exit）和 program 命令（例如/bin/目录下的 ls，grep 等）。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，Shell 可以不等待进程结束，直接返回。

三、 使用环境

Vmware Workstation pro 15.5
Minix 3.3.0
MobaXterm Personal Edition v20.1
FileZilla 3.47.2.1
Dev-C++ 5.11

四、 实验过程

1.Shell 内置命令

在我的代码中，我使用了 builtin_cmd 函数来判断读入的命令是否为内置命令。读入用户输入的命令行之后，经过 parseline 函数将它解析成 argv[][]数组，然后判断字符串 argv[0]是否是以下内置命令中的一条。如果是，则 builtin_cmd 函数会执行相应内置命令的代码段，并返回 1。如果不是则返回 0 并执行 Program 命令中的运行程序部分。以下是各个内置命令的实现说明。

1.1 工作路径移动命令 cd

cd 命令的作用是改变当前 Shell 程序的工作目录。这里需要用 chdir 系统调用来移动 Shell 的工作目录。主要代码如下。

```
if(chdir(argv[1])<0)
    printf("cd:can't cd to %s\n",argv[1]);
```

其中，argv[1]就是我们需要移动到的工作目录，如果它不存在则会输出错误信息。输出的错误信息如图所示。

```
tsh> cd you
cd:can't cd to you
```

1.2 显示最近执行的指令 history

history 命令要求输出最近执行的 n 条指令。这里可以在读取到输入行之后，立刻将所有的输入行保存到 command_history[][]数组中，并记录当前已经输入的命令行数。

```
strcpy(command_history[(command_num++)%MAXLINE],cmdline);
```

当调用 history 命令时，利用 for 循环输出 command_history[][]中的命令。注意处理输入的 n 大于已输入命令数的边界情况即可。

```
if(n>command_num-1)
    n=command_num-1;
for(i=command_num-n;i<command_num;i++)
    printf(" %d %s",i,command_history[i%MAXLINE]);
```

该命令最终的执行效果如下图所示。

```
tsh> history 7
 8 grep 6 < result.txt &
 9 grep a & < result.txt | grep 6 > result.out
 10 cat result.out
 11 grep a & < result.txt | grep 6 & | grep 8 >> re
 12 cat result.out
 13 mytop
 14 history 7
```

1.3 退出命令 exit

需要退出 Shell 时，直接利用系统调用 exit(0)退出主程序即可。退出命令的效果如图所示。

```
tsh> exit
#
```

1.4 程序运行统计 mytop

内置命令 mytop 的实现包括两部分：内存使用信息和 CPU 使用百分比。由于这个功能实现起来非常复杂，并且包含了多种错误处理函数，为避免错误处理函数中的 exit(1)直接退出 Shell 本身，我们可以新建一个子进程运行 mytop。

获取内存使用信息需要读取文件/proc/meminfo 中的参数，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 。然后根据公式(pagesize * total)/1024，计算出总内存和其他页内存大小。读取文件可以直接使用 fopen 函数，也可以使用 open 和 dup 系统调用。基本框架如下。

```
fp = fopen("/proc/meminfo", "r");
fscanf(fp, "%u %lu %lu %lu %lu", &pagesize, &total, &free,
                  &largest, &cached);
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);
```

CPU 使用百分比的计算很复杂。我们需要先读取/proc/kinfo 中的进程数和任务数，计算进程和任务总数 nr_total。

```
fp = fopen("/proc/kinfo", "r"));
fscanf(fp, "%u %u", &nr_procs, &nr_tasks);
fclose(fp);
nr_total = (int) (nr_procs + nr_tasks);
```

然后遍历/proc 目录下以数字命名的文件夹，它们就是进程的 pid。

```
p_dir = opendir("/proc"));
for (p_ent = readdir(p_dir); p_ent != NULL; p_ent = readdir(p_dir))
{
    pid = strtol(p_ent->d_name, &end, 10);
    if (!end[0] && pid != 0)
        parse_file(pid);
```

```
    }
    closedir(p_dir);
```

然后读取/proc/pid/psinfo 中的每个进程的信息，有用信息的包括 type，endpoint，pid，cycles_hi,cycles_lo，state。

```
    sprintf(path, "/proc/%d/psinfo", pid);
    fp = fopen(path, "r");
    fscanf(fp, "%d", &version);
    fscanf(fp, " %c %d", &type, &endpt);
    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;
    fscanf(fp, " %*255s %c %*d %*d %*lu %*u %lu %lu"
        ,&state, &cycles_hi, &cycles_lo);
```

连续读 3 次 cycles_hi,cycle_lo（CPUTIMENAMES=3），然后利用 make64 函数拼接成 64 位，放在 p_cpucycles[]数组中。

```
    for(i = 1; i < CPUTIMENAMES; i++) {
        fscanf(fp, " %lu %lu", &cycles_hi, &cycles_lo);
        p->p_cpucycles[i] = make64(cycles_lo, cycles_hi);
    }
```

然后计算每个进程 proc 的 ticks，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(proc->p_endpoint == prev_proc->p_endpoint) {
            t = t + prev_proc->p_cpucycles[i] - proc->p_cpucycles[i];
        } else {
            t = t + prev_proc->p_cpucycles[i];
        }
    }
```

最后计算总的 CPU 使用百分比。遍历所有的进程和任务，判断类型，计算 systemticks，userticks，然后换算成百分比输出即可。由于 kernelticks 和 idleticks 为 0，所以不用计算输出这两项。

```
    for(p = nprocs = 0; p < nr_total; p++) {
        u64_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;
        if(p-NR_TASKS == IDLE) {
            idleticks = uticks;
            continue;
```

```
        }
        if(p-NR_TASKS == KERNEL) {
              kernelticks = uticks;
        }
        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++;
    }
    printf("CPU states: %6.2f%% user, ",
        100.0 * userticks / total_ticks);
    printf("%6.2f%% system, ", 100.0 * systemticks / total_ticks);
```

本功能借鉴了 top.c 的写法。为了显示框架，上述代码均没有包含错误处理。由于不必实现 top 的全部功能，实际代码相比较为简单。最终输出的效果如图所示。

```
tsh> mytop
main memory: 2095548K total, 2031156K free, 2028016K contig free, 33400K cached
CPU states:   0.08% user,   0.50% system
```

2.Program 命令

使用 builtin_cmd 函数判断不是内置命令，则需要运行这个程序。运行程序时，他们的输入输出可以是键盘和控制台，也可以重定向定向到文件中，还可以使用管道作为下一个进程的输入输出。以下是这些功能的实现。

2.1 运行程序

当我们需要运行程序时，我们要先利用 fork 系统调用创建一个子进程，然后在子进程中利用 execvp 调用将 minix 程序装载到该进程，并赋予运行参数。要注意这里不能使用 execve 系统调用，因为 execve 不会调用系统环境变量中配置的路径。然后父进程利用 waitpid 调用等待子进程结束，结束后再读入下一条命令。程序框架如下。

```
    if((pid = Fork())==0)
        if(execvp(argv[0],argv))
            printf("%s: Command not found\n",argv[0]);
    else
        Waitpid(pid,NULL,0);
```

其中大写字母开头的 Fork 和 Waitpid 是包含错误处理的包装函数。此处若 execvp 有返回值，说明加载执行不成功。若 execvp 执行成功则没有返回值。由于还要实现重定向，管道，后台等功能，还需要修改这段代码使其包含上述的其他功能。

程序正常运行时如下图所示。

```
tsh> ls -l
total 104
-rw-r--r--  1 root  operator     44 Sep 14  2014 .exrc
-rw-r--r--  1 root  operator    605 Sep 14  2014 .profile
-rw-r--r--  1 root  operator    456 Apr  5 22:08 result.txt
-rwxr-xr-x  1 root  operator  17304 Apr  5 21:57 tsh
-rw-r--r--  1 root  operator  12513 Apr  5 21:54 tsh.c
drwxr-xr-x  3 root  operator    192 Mar 22 16:08 your
```

找不到要执行的程序时如下图所示。

```
tsh> grop a
grop: Command not found
```

2.2 重定向

MINIX 为每个进程赋予键盘输入和控制台输出的文件描述符默认为 0 和 1。子进程装载程序前，要实现重定向，先使用 for 循环遍历整个 argv[][]数组。若发现相应的标志符，则利用 open 系统调用，为该文件分配新的文件描述符。然后利用 close 系统调用将默认输入或者输出关闭，再利用系统调用 dup(fd)将某个打开文件的文件描述 fd 映射到标准输入或输出。

```
if(*argv[i] == '<'){
    file[0]=argv[i+1];
    argv[i]=NULL;
    fdn=open(file[0],O_RDONLY);
    if(fdn<0){
        printf("tsh: cannot open %s: no such file\n",file[0]);
        continue;
    }
    close(0);
    dup2(fdn,0);
}
else if((*argv[i] == '>')&&(*(argv[i]+1) != '>')){
    file[1]=argv[i+1];
    argv[i]=NULL;
    fdn=open(file[1],O_WRONLY|O_CREAT,0666);
    close(1);
    dup2(fdn,1);
}
else if((*argv[i] == '>')&&(*(argv[i]+1) == '>')){
    file[1]=argv[i+1];
    argv[i]=NULL;
    fdn=open(file[1],O_WRONLY|O_APPEND|O_CREAT,0666);
    close(1);
    dup2(fdn,1);
}
```

注意在利用 strcmp 比较字符串的时候，>和>>容易混淆，需要尽量避免使用 strcmp 函数，并使用 else if 来避免这种情况。在找不到文件的情况下，我的解决方案是输入重定向报错，输出重定向会创建新文件。所以在输出重定向中调用 open 时使用了 O_CREAT 选项。O_APPEND 选项可以将光标定位到文件末尾，以实现追加写而不是覆盖写。

另外，为了使代码读起来更清晰，这里采用了 dup2 函数，这样可以显示要被赋值的文件描述符。事实上，dup 系统调用总是返回最小的文件描述符。所以如果使用 dup 系统调用的话，将上述代码中的 dup2 函数全部换成 dup(fdn)即可。

输出重定向效果如下图所示。

```
tsh> ls -a -l > result.txt
tsh> cat result.txt
total 112
drwxr-xr-x   3 root  operator    832 Apr  5 21:33 .
drwxr-xr-x  17 root  operator   1408 Mar  4 16:08 ..
-rw-r--r--   1 root  operator     44 Sep 14  2014 .exrc
-rw-r--r--   1 root  operator    605 Sep 14  2014 .profile
-rw-r--r--   1 root  operator      0 Apr  5 21:33 result.txt
-rwxr-xr-x   1 root  operator  17304 Apr  5 21:32 tsh
-rw-r--r--   1 root  operator  12522 Apr  5 21:32 tsh.c
drwxr-xr-x   3 root  operator    192 Mar 22 16:08 your
tsh> ls -a -l >> result.txt
tsh> cat result.txt
total 112
drwxr-xr-x   3 root  operator    832 Apr  5 21:33 .
drwxr-xr-x  17 root  operator   1408 Mar  4 16:08 ..
-rw-r--r--   1 root  operator     44 Sep 14  2014 .exrc
-rw-r--r--   1 root  operator    605 Sep 14  2014 .profile
-rw-r--r--   1 root  operator      0 Apr  5 21:33 result.txt
-rwxr-xr-x   1 root  operator  17304 Apr  5 21:32 tsh
-rw-r--r--   1 root  operator  12522 Apr  5 21:32 tsh.c
drwxr-xr-x   3 root  operator    192 Mar 22 16:08 your
total 120
drwxr-xr-x   3 root  operator    832 Apr  5 21:33 .
drwxr-xr-x  17 root  operator   1408 Mar  4 16:08 ..
-rw-r--r--   1 root  operator     44 Sep 14  2014 .exrc
-rw-r--r--   1 root  operator    605 Sep 14  2014 .profile
-rw-r--r--   1 root  operator    456 Apr  5 21:33 result.txt
-rwxr-xr-x   1 root  operator  17304 Apr  5 21:32 tsh
-rw-r--r--   1 root  operator  12522 Apr  5 21:32 tsh.c
drwxr-xr-x   3 root  operator    192 Mar 22 16:08 your
```

输入重定向效果如图所示。

```
tsh> grep 6 < result.txt
drwxr-xr-x  17 root  operator   1408 Mar  4 16:08 ..
-rw-r--r--   1 root  operator    605 Sep 14  2014 .profile
drwxr-xr-x   3 root  operator    192 Mar 22 16:08 your
drwxr-xr-x  17 root  operator   1408 Mar  4 16:08 ..
-rw-r--r--   1 root  operator    605 Sep 14  2014 .profile
-rw-r--r--   1 root  operator    456 Apr  5 21:33 result.txt
drwxr-xr-x   3 root  operator    192 Mar 22 16:08 your
```

也可以同时将输入和输出重定向。

```
tsh> grep 9 < result.txt > result.out
tsh> cat result.out
drwxr-xr-x   3 root  operator    192 Mar 22 16:08 your
drwxr-xr-x   3 root  operator    192 Mar 22 16:08 your
```

输入重定向出错时如图所示。

```
tsh> grep a < result.out
tsh: cannot open result.out: no such file
```

2.3 管道

若有 n 个子进程组成管道流，Shell 应当用 pipe 调用创建 n-1 对管道描述符，关闭不需要的读写端。每个子进程利用 dup 将前一个管道的读端映射到标准输入，后一个管道的写端映射到标准输出。

```
if(*argv[i] == '|'){
    argv[i]=NULL;
    int fd[2];
    pipe(&fd[0]);
    if((pid = Fork())==0){
        file[1]="pipe";
        close(fd[0]);
        close(1);
```

```
            dup2(fd[1],1);
            close(fd[1]);
            break;
        }
        else{
            process_pos=i+1;
            file[0]="pipe";
            bg=0;
            close(fd[1]);
            close(0);
            dup2(fd[0],0);
            close(fd[0]);
        }
```

在我的实现中，当检测到管道的标识符时，首先创建 fd 数组，并使用 pipe 调用。之后我们使用 fork 调用创造一个新的子进程，并且在父子进程之间建立管道。我们让子进程往管道里写入，让父进程从管道里读入，这样做的好处是我们可以让父进程使用 waitpid 系统调用等到子进程执行完之后再执行。这样方便回收进程，让 Shell 等管道流中的所有进程执行完之后再读入下一条指令。

对于要向管道中写入的子进程来说，我们先关闭不需要的管道读端 fd[0]，然后使用 dup2 函数将文件描述符 1，也就是标准输出重定向到管道的写端 fd[1]，然后我们关闭原先管道的写端所代表的文件描述符 fd[1]。由于符号之后的内容是下一个进程的运行程序的参数数组 argv[][]。所以在做完这些之后，我们应该使用 break 跳出循环，将后面的参数留给父进程，也就是接下来执行的进程来解析参数。

对于要向管道中读入的父进程来说，写法大致相似。我们先关闭不需要的管道写端，我们先关闭不需要的管道读端 fd[0]，然后 dup2 函数将标准输出重定向到管道的写端 fd[1]，然后我们关闭原先管道写端 fd[1]。由于这是下一个要执行的进程，之前解析出来的参数不一定在适用于这个进程。所以我们应该利用 process_pos 记录下当前参数在 argv 数组中的位置，并修改 execvp 函数的参数。此外，这个进程不一定是后台执行，所以我们将 bg 的重新赋值为 0，并在接下来检测下一个后台运行的标志符。

为了兼容管道，加载执行的部分应当修改如下。

```
    if(pid!=0)
        Waitpid(pid,NULL,0);
    if(execvp(argv[process_pos],(argv+process_pos))){
        printf("%s: Command not found\n",argv[process_pos]);
        exit(1);
    }
```

这样一来，Shell 就会按管道流的顺序来执行程序。多个管道的执行如下。

```
tsh> ls -a -l | grep a | grep 0
drwxr-xr-x  17 root  operator   1408 Mar  4 16:08 ..
-rw-r--r--   1 root  operator     44 Sep 14  2014 .exrc
-rw-r--r--   1 root  operator    605 Sep 14  2014 .profile
-rwxr-xr-x   1 root  operator  17304 Apr  5 21:32 tsh
drwxr-xr-x   3 root  operator    192 Mar 22 16:08 your
```

管道和重定向也可以一起使用。

```
tsh> grep 6 < result.txt &
tsh> grep a & < result.txt | grep 6 > result.out
tsh> cat result.out
drwxr-xr-x  17 root  operator   1408 Mar  4 16:08 ..
-rw-r--r--   1 root  operator    605 Sep 14  2014 .profile
drwxr-xr-x   3 root  operator    192 Mar 22 16:08 your
drwxr-xr-x  17 root  operator   1408 Mar  4 16:08 ..
-rw-r--r--   1 root  operator    605 Sep 14  2014 .profile
-rw-r--r--   1 root  operator    456 Apr  5 21:33 result.txt
drwxr-xr-x   3 root  operator    192 Mar 22 16:08 your
```

管道中 execvp 出错的情况如下图。

```
tsh> ls -a -l | grop a
grop: Command not found
```

2.4 后台运行

要实现程序的后台运行，首先需要判断相应的调用命令中是否有&符号。若有，则需要屏蔽键盘和控制台，方法是将子进程的标准输入、输出映射成/dev/null。然后子进程调用 signal(SIGCHLD,SIG_IGN)，使得 minix 接管此进程。这样父进程可以避免调用 waitpid 直接运行下一条命令。

```
if(bg){
    if(!file[0]){
        file[0]="/dev/null";
        fdn=open(file[0],O_RDWR);
        close(0);
        dup2(fdn,0);
    }
    if(!file[1]){
        file[1]="/dev/null";
        fdn=open(file[1],O_RDWR);
        close(1);
        dup2(fdn,1);
    }
    Signal(SIGCHLD,SIG_IGN);
}
```

为了和管道，重定向等功能兼容，需要判断子进程是否已经有除键盘和控制台之外的标准输入输出。所以采用了 file 数组记录输入输出，file[0]记录标准输入，file[1]记录标准输出。默认的键盘控制台输入输出为 NULL，重定向或使用管道后将把路径字符串记录到 file 数组中，这样调用后台运行时就不会覆盖掉已经更改过的标准输入和输出。

部分进程如 vim 无法使用后台运行，重定向时会出错，如下图。

```
tsh> vi result.txt
tsh> vi result.txt &
ex/vi: Vi's standard input and output must be a terminal
```

当进程在管道流中或使用重定向时，Shell 不会将这些进程的输入输出再次重定向。这些进程正常输入输出。如下图所示。

```
tsh> grep a & < result.txt | grep 6 > result.out
tsh> cat result.out
drwxr-xr-x  17 root  operator   1408 Mar  4 16:08 ..
-rw-r--r--   1 root  operator    605 Sep 14  2014 .profile
drwxr-xr-x   3 root  operator    192 Mar 22 16:08 your
drwxr-xr-x  17 root  operator   1408 Mar  4 16:08 ..
-rw-r--r--   1 root  operator    605 Sep 14  2014 .profile
-rw-r--r--   1 root  operator    456 Apr  5 21:33 result.txt
drwxr-xr-x   3 root  operator    192 Mar 22 16:08 your
```

这些功能全部放在一起是这样的。

```
tsh> grep a & < result.txt | grep 6 & | grep 8 >> result.out
tsh> cat result.out
drwxr-xr-x  17 root  operator   1408 Mar  4 16:08 ..
-rw-r--r--   1 root  operator    605 Sep 14  2014 .profile
drwxr-xr-x   3 root  operator    192 Mar 22 16:08 your
drwxr-xr-x  17 root  operator   1408 Mar  4 16:08 ..
-rw-r--r--   1 root  operator    605 Sep 14  2014 .profile
-rw-r--r--   1 root  operator    456 Apr  5 21:33 result.txt
drwxr-xr-x   3 root  operator    192 Mar 22 16:08 your
drwxr-xr-x  17 root  operator   1408 Mar  4 16:08 ..
drwxr-xr-x   3 root  operator    192 Mar 22 16:08 your
drwxr-xr-x  17 root  operator   1408 Mar  4 16:08 ..
drwxr-xr-x   3 root  operator    192 Mar 22 16:08 your
```

五、总结

这次实验用到的主要是进程管理和文件管理的有关系统调用和它们的相关函数。进程管理的系统调用包括：创建一个与父进程相同的子进程 fork，等待一个子进程结束 waitpid 和 wait，替换一个进程的内核映像 exceve，终止进程的执行 exit。文件管理的系统调用包括：打开一个文件进行只读或只写或读写 open，关闭一个文件 close，为打开文件分配一个新的文件描述符 dup，创建一个管道 pipe。此外还包括改变当前工作目录的 chdir。

通过这次实验，我掌握了 Shell 和系统调用和的知识，并能利用系统调用编写 C 语言程序。系统调用是操作系统与应用程序之间的接口，大多数现代操作系统都有功能相似的系统调用。理解系统调用有助于我们理解之后要学的的操作系统原理的的有关知识。