

Project 2: 简易 Unix shell 编程

516030910259 刘欣鹏

1. 实验目的

通过简易 Unix shell 的编写，了解 linux 进程的基本使用。

2. 实验原理

该项目由一个 C 程序组成，它作为接收用户命令并在单独的进程执行每个命令的 Shell 接口。Shell 在下一个命令进入之后为用户提供了提示符。

实现 Shell 接口的一种技术是父进程首先读用户命令行的输入，然后建一个独立的子进程来完成这个命令。除非另作说明，父进程在继续之前等待子进程退出。然而，Unix Shell 一般也允许子进程在后台进行（或并发地运行），通过在命令的最后使用&符号。

用系统调用 `fork()` 来创建独立的子进程，通过使用 `exec()` 族中的一种系统调用来执行用户命令。

3. 实验步骤

3.1. 源代码

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#define MAX_LINE 80
#define BUFFER_SIZE 11
char buffer[BUFFER_SIZE][MAX_LINE/2+1][80]; //历史命令存储
int llength[BUFFER_SIZE]; //每个历史指令参数个数
int l, r; //历史命令指针
int flag; //Ctrl+C 事件是否发生
void handle_SIGINT(){
    int i = l, j;
    char tmp[20];

    strcpy(tmp, "\n");
    write(STDOUT_FILENO, tmp, strlen(tmp));
    while (i != r) {
        strcpy(tmp, " ");
        for (j = 0; j < llength[i]; j++){
            write(STDOUT_FILENO, buffer[i][j],
                strlen(buffer[i][j]));
            write(STDOUT_FILENO, tmp, strlen(tmp));
        }
        strcpy(tmp, "\n");
        write(STDOUT_FILENO, tmp, strlen(tmp));
        i = (i + 1) % 11;
    }
}
```

```

} //Ctrl+C 的事件 handler, 输出最近十个历史命令
strcpy(tmp, "Output Done!\n");
write(STDOUT_FILENO, tmp, strlen(tmp));
flag=1;
}

int setup(char inputBuffer[], char *args[], int *background){
    int length, i, start, ct, j;
    char tmp[50];
    char re[10];
    ct = 0;
    length = read(STDIN_FILENO, inputBuffer, MAX_LINE);
    if (flag==1) return 0; //若发生 Ctrl+C 事件, 忽略当前命令
    start = -1;

    if (length == 0) exit(0);
    else if (length < 0){
        perror("error reading the command\n");
        exit(-1);
    }

    for (i = 0; i < length; i++){
        switch (inputBuffer[i]){
            case ' ':
            case '\t':
                inputBuffer[i]='\0';
                if (start!=-1){
                    args[ct] = &inputBuffer[start];
                    ct++;
                }
                start=-1;
                break;
            case '\n':
                if (start!=-1){
                    inputBuffer[i]='\0';
                    args[ct] = &inputBuffer[start];
                    ct++;
                }
                args[ct] = NULL;
                break;
            default:
                if (start == -1) start = i;
                if (inputBuffer[i] == '&') {
                    *background = 1;
                }
        }
    }
}

```

```

        inputBuffer[i] = '\0';
    }
    break;
}
}
args[ct] = NULL;
if (ct <= 0 || args[ct] != NULL) return 0;
//读入命令
strcpy(re, "r");
if (ct == 1 && !strcmp(re,args[0])){
    for (i=0;i<llength[r-1];i++){
        args[i]=&buffer[r-1][i];
    }
    ct=llength[r-1];
    args[ct]=NULL;
    *background=0;
} //若命令为 r, 将其替换为历史命令中最新一条
else if (ct==2 && !strcmp(re,args[0]) && strlen(args[1])==1){
    i=r;
    int found=0,k;
    while (i!=1){
        i--;
        if (i==-1) i=10;
        if (args[1][0]==buffer[i][0][0]) {found=1;k=i;break;}
    }

    if (!found) return 0;

    for (i=0;i<llength[k];i++){
        args[i]=&buffer[k][i];
    }
    ct=llength[k];
    args[ct]=NULL;
    *background=0;
} //若命令为 r x, 将其按替换为历史命令中最新一条首字母为 x 的指令 ; 若
无相应指令, 忽略该命令

for (i = 0; i < ct; i++){
    for (j = 0; j < strlen(args[i]); j++)
        buffer[r][i][j] = args[i][j];
    buffer[r][i][j] = '\0';
}
llength[r] = ct;
r = (r + 1) % 11;

```

```

    if (l == r) l = (l + 1) % 11; //将当前指令加入历史命令缓存中
    return 1;
} //读取命令，若输入指令有效返回 1，否则返回 0

```

```

int main(void){
    char inputBuffer[MAX_LINE];
    char consts[40];
    char cd[4];
    char ch;
    int background;
    char *args[MAX_LINE/10];
    int valid;
    signal(SIGINT, handle_SIGINT);
    l = 0; r = 0;
    flag = 0;
    strcpy(cd, "cd");
    //初始化
    while (1){
        background = 0;
        strcpy(consts, "COMMAND->");
        write(STDOUT_FILENO, consts, strlen(consts));
        valid = setup(inputBuffer, args, &background);
        if (flag == 1) {
            flag = 0;
            continue;
        } //若发生 Ctrl+C 事件，忽略当前指令直接进入下一循环
        if (!valid){
            strcpy(consts, "Invalid instruction!\n");
            write(STDOUT_FILENO, consts, strlen(consts));
            continue;
        } //若当前命令不合法，忽略当前指令直接进入下一循环
        pid_t pid;
        if (!strcmp(args[0], cd)){
            int xx=r-1;
            if (xx==-1) xx=10;
            if (llength[xx]==2) chdir(args[1]);
        } //若当前指令为 cd，直接通过系统调用 chdir() 执行
        else{
            pid=fork();
            if (pid == 0) {
                execvp(args[0], args);
                write(STDOUT_FILENO, consts, strlen(consts));
                exit(0); //在子进程中执行命令
            }
        }
    }
}

```

```

        else if (background == 0) waitpid(pid); //若指令为非后台执行
        则等待子进程终止再进入下一循环
    }
}
}

```

3.2. 测试

```

ubuntu@ubuntu:~/Desktop$ ./shell
COMMAND->date
Tue May  1 05:27:51 PDT 2018
COMMAND->cd ..
COMMAND->pwd
/home/ubuntu
COMMAND->cd Desktop
COMMAND->pwd
/home/ubuntu/Desktop
COMMAND->ls
hwtree matrix matrix.c shell shell.c test test.c tree.py tree.pyc tt.c
COMMAND->pwd
/home/ubuntu/Desktop
COMMAND->cd ..
COMMAND->cd ..

COMMAND->pwd
/home
COMMAND->cd -
COMMAND->ls
ubuntu
COMMAND->pwd
/home
COMMAND->cd ubuntu
COMMAND->pwd
/home/ubuntu
COMMAND->^C
ls
pwd
cd ..
cd ..
pwd
cd -
ls
pwd
cd ubuntu
pwd
Output Done!

COMMAND->date
Tue May  1 05:29:15 PDT 2018
COMMAND->r
Tue May  1 05:29:23 PDT 2018
COMMAND->r p
/home/ubuntu
COMMAND->gedit tt.c&
COMMAND->date
Tue May  1 05:30:02 PDT 2018
COMMAND->

```

上两图展示了该简易 Shell 程序的指令执行、历史命令缓存、历史纪录执行、指令后台执行功能，并额外实现了 cd 命令的执行。

4. 心得与体会

通过本实验，我对 linux 下 fork() 有了更深入的认识，对 linux 中的进程有了更深入地了解。

在实现 cd 命令的过程中，遇到了一个问题：不能执行该命令。经过查阅相关资料，我了解到 cd 命令不能与其他命令一样地通过创建子进程来执行，因为这样只改变了子进程的工作目录，而父 Shell 进程的工作目录并没有改变，且 execvp() 函数并不支持 cd 命令。而在 linux 自带的 shell 中，cd 命令是直接由 shell 自身进行解析、执行的。

于是我增加了判断 cd 命令的代码，若判定当前命令为 cd 命令，则直接调用 chdir() 系统调用在父进程中转换工作目录，最终解决了问题，在简易 Shell 程序中成功实现了 cd 命令。