

操作系统PROJECT2

---

# 实现Shell及其历史特征

---

作者：李珉超

学号：515030910361

2017.5.17

## 1 实验原理

- 1.在主函数main中，接受来自用户的输入。
- 2.将用户输入的字符串做处理，将命令及其参数根据空白字符分割开来。
- 3.如果字符串非空，创建新的进程，参数是我们之前处理过的字符串数组。
- 4.根据最后一个参数，决定子进程是否与父进程并行。
- 5.使用大小为10的循环队列，记录最近10条指令记录。
- 6.通过一个信号监听的进程，捕捉信号。
- 7.如果捕捉到信号，进入信号处理函数。

## 2 实验代码

```
1  #include <signal.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <string.h>
5  #define MAXLINE 80
6  #define HISTORY 10
7  #define RX_LEN 10
8  char buffer[HISTORY][MAXLINE];
9  char RX[RX_LEN];
10 char inputBuffer[MAXLINE];
11 int background;
12 char *args[MAXLINE / 2 + 1];
13 int fpid, exitstatus;
14 int head = 0; int tail = -1;
15
16 void handler_SIGINT()
17 {
18     write(STDOUT_FILENO, "\n", 1);
```

```

19     if(tail == -1) {write(STDOUT_FILENO, "No instruction\n", 15);
        fflush(stdin); return ;}

20     else

21     {

22         for(int index = head; index <= tail; index++)

23         {

24             write(STDOUT_FILENO, buffer[index % HISTORY],
                strlen(buffer[index % HISTORY]));

25             write(STDOUT_FILENO, "\n", 1);

26         }

27     }

28     write(STDOUT_FILENO, "\n", 1);

29     gets(RX);

30     if(strlen(RX) == 1 && RX[0] == 'r'){exe(tail % HISTORY); fflush(
        stdin); return ;}

31     else if(strlen(RX) == 3 && RX[0] == 'r')

32     {

33         for(int index = tail; index >= head; index--)

34         {

35             if(buffer[index%HISTORY][0] == RX[2]) {exe(index
                % HISTORY); fflush(stdin); return ;}

36         }

37         //exe(buffer[tail % HISTORY]); return ;

38     }

39     fflush(stdin);

40 }

41

42 void exe(int i)

43 {

44     char inputBuffer[MAXLINE];

```

```

45     int background;

46     char *args[MAXLINE / 2 + 1];

47     strcpy(inputBuffer, buffer[i]);

48     strcpy(buffer[(++tail) % HISTORY], inputBuffer);

49     if(tail - head == HISTORY) head++;

50     int fpid, exitstatus;

51     int index, aindex = 0;

52     int flag = 0;

53     for (index = 0; inputBuffer[index]; index++)
54     {

55         if (!flag)

56         {

57             if (inputBuffer[index] != ' ' && inputBuffer[

                    index] != '\t') flag = 2;

58             else continue;

59         }

60         if (flag)

61         {

62             if (inputBuffer[index] != ' ' && inputBuffer[

                    index] != '\t')

63                 // this one isnot a blank

64                 {

65                     if (flag == 2)

66                         // the former char is a blank character,

                                and this one is not.

67                     {

68                         args[aindex++] = inputBuffer +

                                index;

69                         flag = 1;

70                         continue;

```

```

71         }
72     }
73     else// this one is a blank
74     {
75         if (flag == 1)
76         {
77             inputBuffer[index] = 0;
78             flag = 2;
79         }
80     }
81 }
82 }
83 if (args[aindex - 1][0] == '&' && args[aindex - 1][1] == 0)
84 {background = 0; args[aindex-1] = NULL;}
85 else {background = 1; args[aindex] = NULL;}
86 fpid = fork();
87 switch(fpid)
88 {
89     case -1: perror("fork failed"); exit(1);
90     case 0 : execvp(args[0], args); perror("execvp failed");
91             exit(1);
92     default:
93         if(background) while(wait(&exitstatus)!= fpid);
94 }
95 }
96
97
98 void setup(char inputBuffer[], char *args[], int *background)
99 {

```

```

100     gets(inputBuffer);
101
102     if(!strlen(inputBuffer)) return ;
103
104     strcpy(buffer[(++tail) % HISTORY], inputBuffer);
105
106     if(tail - head == HISTORY) head++;
107
108     int index, aindex = 0;
109
110     int flag = 0;
111
112     for (index = 0; inputBuffer[index]; index++)
113     {
114         if (!flag)
115         {
116             if (inputBuffer[index] != ' ' && inputBuffer[
117                 index] != '\t') flag = 2;
118             else continue;
119         }
120
121         if (flag)
122         {
123             if (inputBuffer[index] != ' ' && inputBuffer[
124                 index] != '\t')
125                 // this one is not a blank
126             {
127                 if (flag == 2)
128                     // the former char is a blank character,
129                     and this one is not.
130                 {
131                     args[aindex++] = inputBuffer +
132                         index;
133                     flag = 1;
134                     continue;
135                 }
136             }
137         }
138     }

```

```

126         else// this one is a blank
127         {
128             if (flag == 1)
129             {
130                 inputBuffer[index] = 0;
131                 flag = 2;
132             }
133         }
134     }
135 }
136 if (args[aindex - 1][0] == '&'amp;' && args[aindex - 1][1] == 0)
137 {*background = 0; args[aindex-1] = NULL;}
138 else {*background = 1; args[aindex] = NULL;}
139 }
140
141 int main(void)
142 {
143     struct sigaction handler;
144     handler.sa_handler = handler_SIGINT;
145     sigaction(SIGINT, &handler, NULL);
146
147     while(1)
148     {
149         printf("COMMAND->");
150         setup(inputBuffer, args, &background);
151         if(!strlen(inputBuffer)) continue;
152         fpid = fork();
153         switch(fpid)
154         {
155             case -1: perror("fork failed"); exit(1);

```

```

156         case 0 : execvp(args[0], args); perror("execvp
                failed"); exit(1);
157         default:
158             if(background) while(wait(&exitstatus)!=
                            fpid);
159     }
160     inputBuffer[0] = 0;
161 }
162 return 0;
163 }

```

## 2.1 全局变量

buffer 存放最近10条指令的记录，是一个循环队列。

RX 存放捕捉到信号之后用户输入的运行历史指令的命令。

inputBuffer 存放主函数中用户输入的命令。

background 标记子进程与父进程并行与否。1代表不并行，0代表并行。

args 存放将inputBuffer根据空白字符分割之后的许多字符串，其中包含指令与参数。

fpid 存放子进程号。

exitstatus 标记子进程是否运行完。

head 和tail 用来维护循环队列buffer。

## 2.2 主函数

在main函数中，首先初始化了信号处理函数。

```

1     struct sigaction handler;
2     handler.sa_handler = handler_SIGINT;
3     sigaction(SIGINT, &handler, NULL);

```

然后在一个循环中不断等待用户指令。接受到来自用户的输入后，如果输入非空，创建子进程并执行相关的指令。如果创建失败，则退出程序。否则，作为父进程会通过execvp(args[0], args)创建子进程，而作为子进程则会根据并行方式选择让父进程等待与否。最后清空指令。



```

1  while(1)
2  {
3      printf("COMMAND->");
4      setup(inputBuffer, args, &background);
5      if(!strlen(inputBuffer)) continue;
6      fpid = fork();
7      switch(fpid)
8      {
9          case -1: perror("fork failed"); exit(1);
10         case 0 : execvp(args[0], args); perror("execvp failed");
11                 exit(1);
12         default:
13                 if(background) while(wait(&exitstatus)!= fpid);
14     }
15     inputBuffer[0] = 0;
16 }

```

## 2.3 信号处理函数

如果当前没有历史记录，那么输出“No instructions”，否则按次序列出所以队列中的指令。

```

1  write(STDOUT_FILENO, "\n", 1);
2  if(tail == -1) {write(STDOUT_FILENO, "No instruction\n", 15); fflush(
3      stdin); return ;}
4  else
5  {
6      for(int index = head; index <= tail; index++)
7      {
8          write(STDOUT_FILENO, buffer[index % HISTORY], strlen(
9              buffer[index % HISTORY]));
10         write(STDOUT_FILENO, "\n", 1);
11     }
12 }

```

```

9         }
10    }
11    write(STDOUT_FILENO, "\n", 1);

```

然后获得用户的命令，用户通过输入” r x” 可以运行之前10 个命令中的任何一个，其中”x”为该命令的第一个字母。如果有命令以”x” 开头，则执行最近的一个。同样，用户可以通过仅输入”r”来再次运行最近的命令。可以假定只有一个空格来将”r” 和第一个字母分开，且该字母后面跟着’\n’。而且，如果希望执行最近的命令，单独的”r” 将紧跟’\n’。

结束之后清理缓冲区。

```

1    gets(RX);
2    if(strlen(RX) == 1 && RX[0] == 'r'){exe(tail % HISTORY); fflush(stdin
    ); return ;}
3    else if(strlen(RX) == 3 && RX[0] == 'r')
4    {
5        for(int index = tail; index >= head; index--)
6        {
7            if(buffer[index%HISTORY][0] == RX[2]) {exe(index %
            HISTORY); fflush(stdin); return ;}
8        }
9        //exe(buffer[tail % HISTORY]); return ;
10   }
11   fflush(stdin);

```

## 2.4 setup函数

首先读取用户的输入，并加入到记录历史的循环列表buffer中。

```

1    gets(inputBuffer);
2    if(!strlen(inputBuffer)) return ;
3    strcpy(buffer[(++tail) % HISTORY], inputBuffer);
4    if(tail - head == HISTORY) head++;

```

对输入的字符串根据空白字符进行分割。具体方法是从左至右遍历输入的字符串，维护一个指针，指向空白字符后或者是开头的第一个非空白字符。当遇到非空白字符后的第一个空白字符，将它赋值为0，并将之前维护的指针赋给args的一个元素，所以自然我们也需要维护args的长度。如此遍历完整个字符串，我们就能够得到根据空白字符分割的字符串数组了。

```
1      int index, aindex = 0;
2
3      int flag = 0;
4
5      for (index = 0; inputBuffer[index]; index++)
6      {
7          if (!flag)
8          {
9              if (inputBuffer[index] != ' ' && inputBuffer[index] != '\t') flag = 2;
10             else continue;
11         }
12         if (flag)
13         {
14             if (inputBuffer[index] != ' ' && inputBuffer[index] != '\t')
15             // this one is not a blank
16             {
17                 if (flag == 2)
18                 // the former char is a blank character, and this
19                 // one is not.
20                 {
21                     args[aindex++] = inputBuffer + index;
22                     flag = 1;
23                     continue;
24                 }
25             }
26         }
27         else // this one is a blank
```

```

24         {
25             if (flag == 1)
26             {
27                 inputBuffer[index] = 0;
28                 flag = 2;
29             }
30         }
31     }
32 }

```

继上一步之后，我们查看字符串的最后一个参数是否为&，来决定子进程的并行方式，然后对应的给background赋值。

特别地，如果最后一个参数是&，我们需要将这个参数从字符串数组中删除，因为它不是指令的参数。

```

1     if (args[aindex - 1][0] == '&' && args[aindex - 1][1] == 0)
2     { *background = 0; args[aindex-1] = NULL; }
3     else { *background = 1; args[aindex] = NULL; }

```

## 2.5 exe函数

集合了之前部分的很多功能，我之所以这么做是因为考虑到信号处理函数的特殊性。因为它是主函数的并行进程，所以信号处理函数中的exe需要申请自己的子进程，而主函数则没有将这一功能写入setup。另外它不需要用户的输入，因为信号处理函数的输入在handler\_SIGINT中实现。基于这些不同，所以不能单纯的调用setup函数，所以我重新定义了这个函数。但是其主要的思想与代码都借鉴了其他的模块。

代码就不另外贴出了，因为代码的思想一致。代码在本section(实验代码)的开头中可找到，实现的思路可借鉴之前的模块。

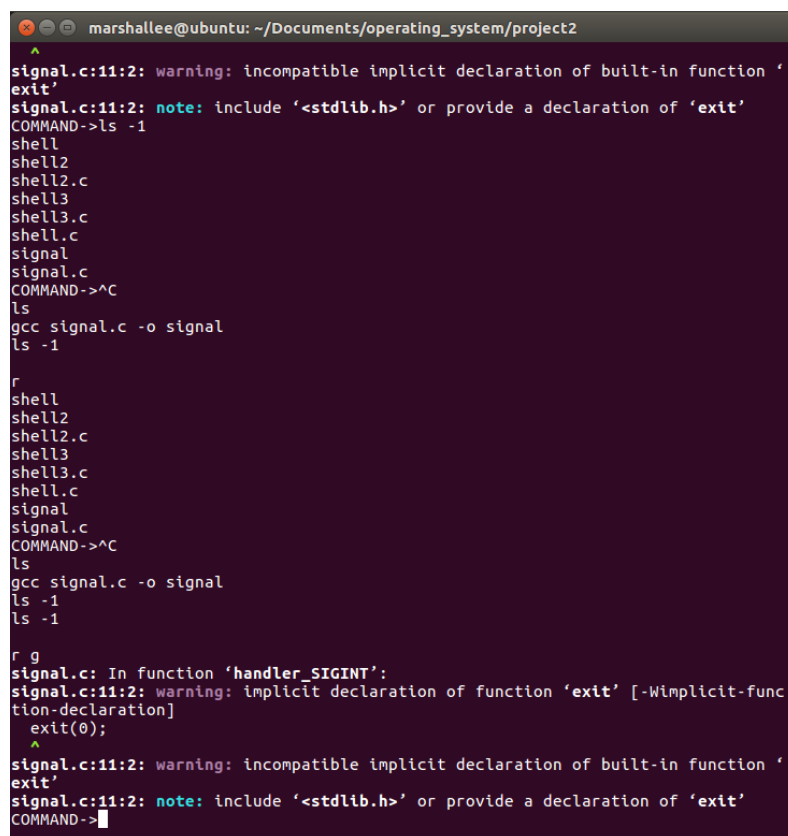
### 3 实验结果

输入指令，可以正确显示结果。如果指令结尾是&，则子进程是并行的(下图中最后一条指令)。



```
marshalllee@ubuntu: ~/Documents/operating_system/project2
marshalllee@ubuntu:~/Documents/operating_system/project2$ ./shell3
COMMAND->ls
shell shell2 shell2.c shell3 shell3.c shell.c signal signal.c
COMMAND->ls -l
shell
shell2
shell2.c
shell3
shell3.c
shell.c
signal
signal.c
COMMAND->ls &
COMMAND->shell shell2 shell2.c shell3 shell3.c shell.c signal signal.c
```

在输入若干指令后，按[Ctrl+C]，进入历史记录。历史记录被输出在命令行中。如果只输入r，然后回车，则执行最近的一条指令。如果输入r后，输入其他字母，则执行以该字母为指令首字母的最近的一条指令。



```
marshalllee@ubuntu: ~/Documents/operating_system/project2
signal.c:11:2: warning: incompatible implicit declaration of built-in function 'exit'
signal.c:11:2: note: include '<stdlib.h>' or provide a declaration of 'exit'
COMMAND->ls -l
shell
shell2
shell2.c
shell3
shell3.c
shell.c
signal
signal.c
COMMAND->^C
ls
gcc signal.c -o signal
ls -l

r
shell
shell2
shell2.c
shell3
shell3.c
shell.c
signal
signal.c
COMMAND->^C
ls
gcc signal.c -o signal
ls -l
ls -l

r g
signal.c: In function 'handler_SIGINT':
signal.c:11:2: warning: implicit declaration of function 'exit' [-Wimplicit-function-declaration]
    exit(0);
    ^
signal.c:11:2: warning: incompatible implicit declaration of built-in function 'exit'
signal.c:11:2: note: include '<stdlib.h>' or provide a declaration of 'exit'
COMMAND->
```

## 4 实验心得

对于命令行的运行原理以及shell的历史特性有了大概的了解。

另外也实践了上课所说的进程的创建，对父子进程的运作模式有了初步的了解。知道了子进程如何继承父进程，然后通过exec开始一个全新的进程。在写代码的过程中也对并行与等待有了更深的理解。