

# 浙江大学实验报告

专业: 计算机科学与技术  
姓名: 吴同  
学号: 3170104848  
日期: 2019年8月15日

课程名称: Linux 程序设计 指导老师: 季江民 电子邮件: wutongcs@zju.edu.cn  
实验名称: 系统程序设计 实验类型: 综合型 联系电话: 18888922355

## 一、实验环境

### 1. 硬件环境

实验所用的计算机为 MacBook Pro (13-inch, 2017, Four Thunderbolt 3 Ports), CPU 为 Intel(R) Core(TM) i5-7267U CPU @ 3.10GHz, 内存为 8 GB 2133 MHz LPDDR3, 显卡为 Intel Iris Plus Graphics 650 1536 MB。

### 2. 操作系统

实验所用的操作系统为 macOS Mojave 10.14.5 Beta。macOS 基于 Darwin BSD 的 Unix 内核, 符合单一 Unix 规范, 在 macOS 的终端下可以正常完成本实验的全部内容。

## 二、实验内容、结果及分析

### 1. makefile

- 所有宏定义的名字: CC OPTIONS OBJECTS SOURCES HEADERS
- 所有目标文件的名字: power main.o stack.o misc.o
- 每个目标的依赖文件:

表 1: 目标文件的依赖文件

| 目标文件    | 依赖文件                         |
|---------|------------------------------|
| power   | main.c main.o stack.o misc.o |
| main.o  | main.c main.h misc.h         |
| stack.o | stack.c stack.h misc.h       |
| misc.o  | misc.c misc.h                |

- 生成每个目标文件执行的命令:

表 2: 生成每个目标文件执行的命令

| 目标文件    | 生成目标文件执行的命令                                |
|---------|--|
| power   | gcc -O3 -o power main.o stack.o misc.o -lm |
| main.o  | gcc -c -o main.o main.c                    |
| stack.o | gcc -c -o stack.o stack.c                  |
| misc.o  | gcc -c -o misc.o misc.c                    |

- 依赖关系树

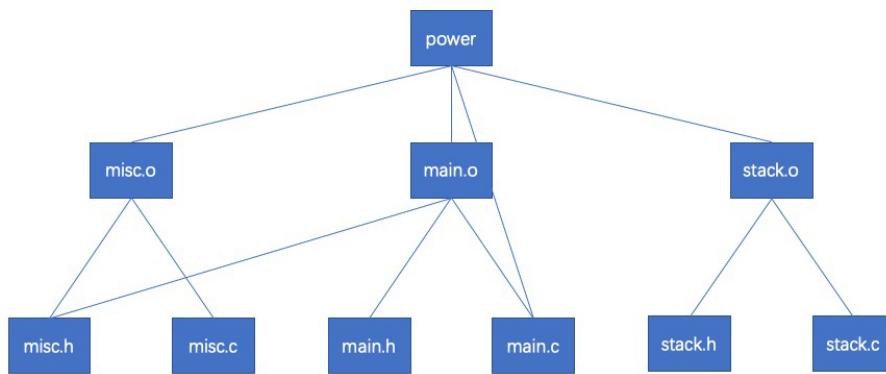


图 1: makefile 对应的依赖关系树

- 生成 main.o stack.o misc.o 时执行的命令

表 3: 生成 main.o stack.o misc.o 时执行的命令

| 目标文件    | 生成目标文件执行的命令               |
|---------|---------------------------|
| main.o  | gcc -c -o main.o main.c   |
| stack.o | gcc -c -o stack.o stack.c |
| misc.o  | gcc -c -o misc.o misc.c   |

## 2. make

- 用 gcc 生成 power 可执行文件

运行步骤：

```
$ gcc -c main.c input.c compute.c
```

```
$ gcc main.o input.o compute.o -o power -lm
```

运行结果：

```
[→ lab3-2 ls
compute.c compute.h input.c input.h main.c main.h
[→ lab3-2 gcc -c main.c input.c compute.c
main.c:6:1: warning: return type defaults to 'int' [-Wimplicit-int]
main()
^~~~
[→ lab3-2 gcc main.o input.o compute.o -o power -lm
[→ lab3-2 ./power
本程序从标准输入获取x和y的值并显示x的y次方。
请输入x的值: 2.4
请输入y的值: 6.5
x的y次方是:296.056
```

图 2: 用 gcc 生成 power 可执行文件

- 用 make 生成 power 可执行文件

修改 Makefile 文件，在结尾添加：

```
clean:
rm -f {*.o,power}
```

运行步骤：

\$ make clean

\$ make

运行结果：

```
[→ lab3-2 make clean
rm -f {*.o,power}
[→ lab3-2 make
gcc -c main.c
main.c:6:1: warning: type specifier missing, defaults to 'int' [-Wimplicit-int]
main()
^
1 warning generated.
gcc -c input.c
gcc -c compute.c
gcc main.o input.o compute.o -o power -lm
[→ lab3-2 ./power
本程序从标准输入获取x和y的值并显示x的y次方。
请输入x的值: 45
请输入y的值: 12
x的y次方是:68952523554931638272.000
```

图 3: 用 make 生成 power 可执行文件

### 3. 进程

源代码：

```
1 /*****
2 * 名称:      prog.c
3 * 作者:      吴同
4 * 学号:      3170104848
5 * 功能:      1、父进程创建子进程p1和p2, p1创建子进程p3
6 *              2、每个进程打印自己的信息, 包括名称, pid, ppid
7 *              3、p1和p2之间实现通信, p3调用ls -l
```



```
56     exit(2);
57 }
58 //将共享内存连接到进程的地址空间
59 *shaeredMemoryAddr = (sharedMemoryStruct*)shmat(shmid, NULL, 0);
60 //如果共享内存连接失败，返回地址为-1
61 if ((void*)-1 == *shaeredMemoryAddr)
62 {
63     fprintf(stderr, "\033[31mshmat failed\033[0m");
64     exit(3);
65 }
66 //设置共享内存中flag数据的值为0，表示没有数据待读取
67 (*shaeredMemoryAddr)->flag = 0;
68 //返回共享内存的标识符
69 return shmid;
70 }
71
72 //向共享内存中写入数据的函数，传入的参数为共享内存的地址和待写入的数据
73 void sharedMemoryWrite(sharedMemoryStruct* shaeredMemory, const char*
    textWrite)
74 {
75     //如果内存处于占用状态，暂不写入
76     while (shaeredMemory->flag)
77     {
78         usleep(200);
79     }
80     //开始写入，将flag设为写数据的进程号
81     shaeredMemory->flag = getpid();
82     //将待写入的数据写入共享内存
83     strncpy(shaeredMemory->text, textWrite, MAXTEXT);
84 }
85
86 //读取共享内存中数据的函数，传入参数为内存地址
87 void sharedMemoryRead(sharedMemoryStruct* shaeredMemory)
88 {
89     //判断共享内存中是否有可读的数据，如果没有数据或数据为本进程自己写入的，则不可读，等待
90     while (getpid() == shaeredMemory->flag || !shaeredMemory->flag)
91     {
92         usleep(200);
93     }
94     //读出数据，显示到屏幕上，提示信息为黄色
95     printf("\033[33m%d read:\033[0m %s\n", getpid(), shaeredMemory->text);
96     //将flag置0，表示没有数据
97     shaeredMemory->flag = 0;
98 }
99
100 //销毁共享内存的函数，传入的参数为共享内存的标识符和内存地址，用在程序结束时
101 void shaeredMemoryDestroy(int shmid, sharedMemoryStruct* shaeredMemory)
102 {
103     //将共享内存从当前进程中分离
104     //若分离失败，返回-1
```

```
105     if (-1 == shmdt((void*)shaeredMemory))
106     {
107         fprintf(stderr, "\033[31mshmdt failed\n\033[0m");
108         exit(4);
109     }
110 //删除共享内存段
111 //若删除失败, 返回-1
112 if (-1 == shmctl(shmid, IPC_RMID, 0))
113 {
114     fprintf(stderr, "\033[31shmctl failed\n\033[0m");
115     exit(5);
116 }
117 }
118 //主函数
119 int main()
120 {
121     //初始化共享内存
122     sharedMemoryStruct* shaeredMemory = NULL;
123     int shmid = sharedMemoryInitialize(KEY, &shaeredMemory);
125
126     //创建子进程p1
127     pid_t pid;
128     pid = fork();
129     switch (pid)
130     {
131         //fork失败
132         case -1:
133             forkError();
134             //子进程p1
135             case 0:
136                 //p1创建子进程p3
137                 pid = fork();
138                 switch (pid)
139                 {
140                     //fork失败
141                     case -1:
142                         forkError();
143                         //子进程p3
144                         case 0:
145                             //显示p3进程信息
146                             procDisplay("p3");
147                             //执行ls -l
148                             printf("\033[33mp3 exec:\033[3m ls -l\033[0m\n");
149                             execlp("ls", "ls", "-l", NULL);
150                             break;
151                         //子进程p1
152                         default:
153                             //显示p1进程信息
154                             procDisplay("p1");
```

```
155 //等待子进程p3结束
156 wait(NULL);
157 //从共享内存中读出数据
158 sharedMemoryRead(shaedMemory);
159 //向共享内存中写入数据
160 sharedMemoryWrite(shaedMemory, "Child process p1 is
161             sending a message!");
162 }
163 break;
164 //主进程
165 default:
166     //主进程创建子进程p2
167     pid = fork();
168     switch (pid)
169     {
170         //fork失败
171         case -1:
172             forkError();
173         //子进程p2
174         case 0:
175             //显示p2进程信息
176             procDisplay("p2");
177             //向共享内存中写入数据
178             sharedMemoryWrite(shaedMemory, "Child process p2 is
179                         sending a message!");
180             //从共享内存中读出数据
181             sharedMemoryRead(shaedMemory);
182             break;
183         //主进程
184         default:
185             //显示主进程信息
186             procDisplay("main");
187             //等待子进程结束
188             waitpid(pid, NULL, 0);
189             //释放分配的共享内存
190             shaeredMemoryDestroy(shmid, shaeredMemory);
191     }
192     //程序结束
193     exit(0);
194 }
```

运行结果：

```
[→ prog ./prog
I am main process. My PID is 64181. My PPID is 78712.
I am p2 process. My PID is 64183. My PPID is 64181.
I am p1 process. My PID is 64182. My PPID is 64181.
I am p3 process. My PID is 64184. My PPID is 64182.
p3 exec: ls -l
total 64
-rwxr-xr-x@ 1 wutong staff 9500 8 7 16:15 a.out
-rw-r--r--@ 1 wutong staff 9516 8 13 00:03 prog
-rw-r--r--@ 1 wutong staff 6712 8 13 00:03 prog.c
64182 read: Child process p2 is sending a message!
64183 read: Child process p1 is sending a message!]
```

图 4: prog 运行结果截图

#### 4. 线程

源代码：

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <pthread.h>
5
6 //输出斐波那契数列的最多项数
7 #define N 100
8 //每行最多显示的项数
9 #define MAX_LINE 10
10
11 //用于存储输出结果的全局变量
12 char result[1024];
13
14 //子线程开始执行的位置
15 void* thread_fibonacci(void* n)
16 {
17     //斐波那契数列表，优化运行时间
18     double fibonacci[N + 1] = { 0,
19         1.0, 1.0, 2.0, 3.0, 5.0,
20         8.0, 13.0, 21.0, 34.0, 55.0,
21         89.0, 144.0, 233.0, 377.0, 610.0,
22         987.0, 1597.0, 2584.0, 4181.0, 6765.0,
23         10946.0, 17711.0, 28657.0, 46368.0, 75025.0,
24         121393.0, 196418.0, 317811.0, 514229.0, 832040.0,
25         1346269.0, 2178309.0, 3524578.0, 5702887.0,
26         9227465.0,
27         14930352.0, 24157817.0, 39088169.0, 63245986.0,
28         102334155.0,
29         165580141.0, 267914296.0, 433494437.0, 701408733.0,
30         1134903170.0,
31         1836311903.0, 2971215073.0, 4807526976.0,
32         7778742049.0, 12586269025.0,
33         20365011074.0, 32951280099.0, 53316291173.0,
34         86267571272.0, 139583862445.0,
```

```
30         225851433717.0, 365435296162.0, 591286729879.0,
31             956722026041.0, 1548008755920.0,
32         2504730781961.0, 4052739537881.0, 6557470319842.0,
33             10610209857723.0, 17167680177565.0,
34         27777890035288.0, 44945570212853.0,
35             72723460248141.0, 117669030460994.0,
36             190392490709135.0,
37         308061521170129.0, 498454011879264.0,
38             806515533049393.0, 1304969544928657.0,
39             2111485077978050.0,
40         3416454622906707.0, 5527939700884757.0,
41             8944394323791464.0, 14472334024676220.0,
42             23416728348467684.0,
43         37889062373143904.0, 61305790721611584.0,
44             99194853094755488.0, 160500643816367072.0,
45             259695496911122560.0,
46         420196140727489664.0, 679891637638612224.0,
47             1100087778366101888.0, 1779979416004713984.0,
48             2880067194370816000.0,
49         4660046610375530496.0, 7540113804746346496.0,
50             12200160415121876992.0, 19740274219868225536.0,
51             31940434634990100480.0,
52         51680708854858326016.0, 83621143489848426496.0,
53             135301852344706760704.0,
54             218922995834555203584.0, 354224848179261997056.0
55         };
56     //将待输出的结果保存到全局变量中
57     for (int i = 1; i <= *(int*)n; ++i)
58     {
59         //输出斐波那契数列第i项
60         sprintf(result + strlen(result), "%.0lf", fibonacci[i]);
61         //如果不是待输出的最后一项，输出逗号
62         if (i != *(int*)n)
63         {
64             sprintf(result + strlen(result), ", ");
65         }
66         //如果输出完成，子线程退出
67         else
68         {
69             sprintf(result + strlen(result), "\n");
70             return NULL;
71         }
72         //控制每行输出的项数不超过MAX_LINE个，如果达到MAX_LINE个，换行
73         if (0 == i % MAX_LINE)
74         {
75             sprintf(result + strlen(result), "\n");
76         }
77     }
78 }
```

```
63 //主函数
64 int main()
65 {
66     //黄色字体显示提示信息
67     printf("\033[33mPlease input the length of Fibonacci sequence (1~100):
68         \033[0m");
69     //从标准输入中读入待输出的项数
70     int n;
71     scanf("%d", &n);
72     //检查输入是否合法
73     if (n < 1 || n > 100)
74     {
75         fprintf(stderr, "\033[31millegal input\n\033[0m");
76         exit(1);
77     }
78     //建立子线程
79     pthread_t thread;
80     //函数调用成功时返回0
81     if (pthread_create(&thread, NULL, thread_fibonacci, (void*)&n))
82     {
83         fprintf(stderr, "\033[31mpthread_create failed\n\033[0m");
84         exit(2);
85     }
86     //等待子线程结束
87     if (pthread_join(thread, NULL))
88     {
89         fprintf(stderr, "\033[31mpthread_join failed\n\033[0m");
90         exit(3);
91     }
92     //输出结果
93     printf("%s", result);
}
```

运行结果：

```
[→ fibonacci ./fibonacci
Please input the length of Fibonacci sequence (1~100): 15
1, 1, 2, 3, 5, 8, 13, 21, 34, 55,
89, 144, 233, 377, 610
[→ fibonacci ./fibonacci
Please input the length of Fibonacci sequence (1~100): 100
1, 1, 2, 3, 5, 8, 13, 21, 34, 55,
89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765,
10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040,
1346269, 2178309, 3524578, 5702887, 9227465, 14930352, 24157817, 39088169, 63245986, 102334155,
165580141, 267914296, 433494437, 701408733, 1134903170, 1836311903, 2971215073, 4807526976, 77787420
49, 12586269025,
20365011074, 32951280099, 53316291173, 86267571272, 139583862445, 225851433717, 365435296162, 591286
729879, 95672026041, 1548008755920,
2504730781961, 4052739537881, 6557470319842, 10610209857723, 17167680177565, 27777890035288, 4494557
0212853, 72723460248141, 117669030460994, 190392490709135,
308061521170129, 498454011879264, 806515533049393, 1304969544928657, 211485077978050, 3416454622906
707, 552793970084757, 8944394323791464, 14472334024676220, 23416728348467684,
37889062373143904, 61305790721611584, 99194853094755488, 160500643816367072, 259695496911122560, 420
196140727489664, 679891637638612224, 1100087778366101888, 1779979416004713984, 2880067194370816000,
4660046610375530496, 7540113804746346496, 12200160415121876992, 19740274219868225536, 31940434634990
100480, 5168070885485326016, 83621143489848426496, 135301852344706760704, 218922995834555203584, 35
4224848179261997056
```

图 5: fibonacci 运行结果截图

```
[→ fibonacci ./fibonacci
Please input the length of Fibonacci sequence (1~100): 0
illegal input
[→ fibonacci ./fibonacci
Please input the length of Fibonacci sequence (1~100): 101
illegal input
```

图 6: fibonacci 运行结果截图（非法输入）

## 5. myshell

设计文档：

- 设计思想

mysh 用 C 语言编写，以面向过程的设计思想为主，遵循结构化程序设计的原则，自顶向下，逐步细化，模块化设计。由于 shell 程序是实现用户与操作系统内核的交互，较为底层，在程序中大量运用了 Unix/Linux 的系统调用。

mysh 对于内部命令和外部命令采用了相同的执行方式，即创建子进程，在子进程中执行命令。这样做虽然增加了执行内部命令时的开销，但可以保证接口的统一和一致，避免了实现后台运行、管道和重定向时的麻烦。前台命令的运行采用同步机制，主进程暂停，等待子进程执行结束，再继续执行下去。后台命令的执行采用异步机制，主进程正常运行，接收到子进程结束的信号后，执行信号处理的函数。在进程间通信的处理上，mysh 采用共享内存的方式实现。对于可被多个进程同时读写的数据，将其存储在共享内存中，可以实现进程之间的高效通信。

在内部命令的实现上，mysh 通过对底层文件系统的调用，实现有关目录操作的功能。在管道和重定向的实现中，mysh 也是通过操作文件描述符，达到了改变输入输出的效果。

- 功能模块

mysh 主要有 3 个功能模块：控制器、命令解析器、命令执行器。

控制器负责控制主进程的行为，包括从指定的文件流中读入输入并解析，控制命令的执行，接收信号并进行处理。具体有：启动程序时，为环境变量表和进程表创建共享内存，并关联到进程；判断启动的方式，选择命令行模式或文件模式；从指定的文件流中逐行读入，调用解析器将输入的字符串解析为命令；解析完成后调用执行器进行执行。

命令解析器负责将传入的字符串解析为命令，命令以链表的形式存储。解析器可以识别命令名、选项、参数、重定向符、管道符、后台符。解析器将这些信息存储于命令链表的节点中。

命令执行器负责接收命令，创建子进程，待父进程向进程表内写入信息后开始执行子进程。执行前台命令时，将主进程阻塞，待子进程结束后，进行回收处理；执行后台命令时，注册信号处理函数，结束执行器，返回控制器继续读入输入。

- 数据结构

命令实现为链表。每个链表节点中存储单个命令的信息，包括命令名，命令的参数列表，是否是后台命令，是否有管道和重定向。通过命令链表的形式将多条命令连接起来，实现在一行内输入多条命令。

环境变量采用哈希表的数据结构。根据环境变量的字符串，将其转换为 hash 值，以优化在环境变量表中的查询。冲突处理采用扩展寻址，直接在冲突的位置依次向后寻找，直到找到空位置。

进程表实现为建立在数组上的链表。每个节点中除了有进程的有关信息，还包括指向前一个和后一个节点的指针。当加入新进程时，新的进程被插入到链表的第一个节点处。这样的数据结构，既不用频繁地分配和释放内存，又有助于查询和遍历，并便于访问最新添加的节点。

- 算法

命令解析器所用的算法为有限状态机。有限状态机以空格和 tab 为分隔符号，逐词处理识别。有限状态机的初始状态是准备读入新命令，创建新命令节点并进行初始化，设置命令名，转移到读参数状态。读参数状态下，转态机首先识别该单词是否为特殊符号，包括-、|、<、>、»、&，如果是-，则设置读选项标志；如果是重定向符号，则进入读文件名的模式，如果是|或&，则准备读入新命令。如果不是特殊符号，按参数处理，添加参数列表的节点。

进程控制算法。子进程开始前由主进程写进程表，子进程结束后主进程回收进程表。进程表的第一项保留，其 pid 为当前正在运行的子进程号，其状态值用于 exit 内建命令。调用 exit 命令相关的函数后，该状态值写为 exit，主进程在循环条件判断时退出，起到退出程序的效果。

### 源代码：

#### Makefile

```
1 all: mysh
2
3 #编译器
4 CC = gcc
5 #编译器选项
6 OPTIONS = -Wall -O3
7
8 #用户级的软件安装路径
9 INSTDIR = /usr/local/bin
10
11 #编译过程
12 mysh: main.c main.h parse.o builtin.o environment.o
```

```

13 $(CC) $(OPTIONS) -o mysh main.c parse.o builtin.o environment.o
14
15 environment.o: environment.c environment.h main.h
16 $(CC) $(OPTIONS) -c environment.c
17
18 parse.o: parse.c parse.h main.h builtin.h environment.h
19 $(CC) $(OPTIONS) -c parse.c
20
21 builtin.o: builtin.c builtin.h main.h
22 $(CC) $(OPTIONS) -c builtin.c
23
24 #清除编译产生的文件
25 clean:
26 -rm -f {*.o,mysh}
27
28 #安装mysh
29 #先将mysh复制到INSTDIR指定的路径中，再赋予执行权限，取消其他用户的写权限
30 install: mysh
31 @ if [ -d $(INSTDIR) ]; then \
32     cp mysh $(INSTDIR); \
33     cp readme $(INSTDIR)/readme_mysh; \
34         chmod a+x $(INSTDIR)/mysh; \
35         chmod og-w $(INSTDIR)/mysh; \
36         chmod og-w $(INSTDIR)/readme_mysh; \
37     fi
38
39 #卸载mysh
40 uninstall: mysh
41 -rm -f $(INSTDIR)/mysh
42 -rm -f $(INSTDIR)/readme_mysh

```

**main.h**

```

1 /**
2 * 名称:          main.h
3 * 作者:          吴同
4 * 学号:          3170104848
5 * 内容:          包含所需的函数库和系统调用
6 */
7
8 #ifndef MAIN_H
9 #define MAIN_H
10
11 //系统调用
12 #include <unistd.h>
13 #include <dirent.h>
14 #include <fcntl.h>
15 #include <termios.h>
16 #include <sys/shm.h>
17 #include <sys/stat.h>

```

```
18 //标准库
19 #include <stdio.h>
20 #include <stdlib.h>
21 #include <string.h>
22 #include <time.h>
23 #include <limits.h>
24 #include <stdbool.h>
25 #include <signal.h>
26
27
28 //该shell程序的名称
29 #define SHELLNAME "mysh"
30
31 //错误代码的宏定义
32 #define FILE_OPEN_FAILED 200
33 #define MALLOC_FAILED 201
34 #define FORK_FAILED 202
35 #define OPENDIR_FAILED 203
36 #define CD_FAILED 204
37 #define SHMGET_FAILED 205
38 #define SHMAT_FAILED 206
39 #define SHMDT_FAILED 207
40 #define SHMCTL_FAILED 208
41 #define DUP_FAILED 209
42
43 //程序运行时允许的最大值，用于分配内存
44 //环境变量的最多个数
45 #define MAX_ENVVAR 64
46 //环境变量的最大长度
47 #define MAX_LENGTH_ENVNAME 16
48 //单个命令的最大长度
49 #define MAX_LENGTH_COMMAND 256
50 //最多同时运行的进程数
51 #define MAX_JOB 8
52 //路径名的最大长度
53 #ifndef _POSIX_PATH_MAX
54 #define _POSIX_PATH_MAX 256
55 #endif
56
57 //等待时每次轮询的暂停时间，单位为毫秒
58 #define WAIT_TIME 500
59
60 //用于控制终端输出的颜色
61 #define REDSTRING "\x1B[31m"
62 #define YELLOWSTRING "\x1B[33m"
63 #define PURPLESTRING "\x1B[35m"
64 #define BLUESTRING "\x1B[36m"
65 #define NORMALSTRING "\x1B[0m"
66
67 //用于显示错误信息并退出程序
```

```
68 void errorExit(char *procName, int errorNo);  
69  
70 //用于调试输出  
71 void debug(int i);  
72  
73 #endif
```

### environment.h

```
1 ****  
2 * 名称:      environment.h  
3 * 作者:      吴同  
4 * 学号:      3170104848  
5 * 内容:      与环境变量、进程表、umask有关的变量和函数的声明及结构体的定义  
6 ****  
7  
8 #ifndef ENVIRONMENT_H //environment.h  
9 #define ENVIRONMENT_H  
10  
11 #include "main.h"  
12  
13 //环境变量的结构, 包括变量名和变量值  
14 typedef struct envVar  
15 {  
16     char name[MAX_LENGTH_ENVNAME];  
17     char value[MAX_LENGTH_COMMAND];  
18 } envVar;  
19  
20 //存储环境变量的起始内存地址的全局变量  
21 extern envVar* envMem;  
22  
23 //初始化环境变量, 用于程序起始时  
24 void envInit();  
25 //根据环境变量名计算hash值, 用于优化查询, 传入参数为变量名  
26 int envHash(char* envName);  
27 //添加环境变量, 传入参数为全局变量的名和值  
28 void envAdd(char* envName, char* envValue);  
29 //获取存储该环境变量信息的内存地址, 传入参数为变量名, 返回指向这段内存的指针  
30 envVar* envAddrGet(char* envName);  
31 //获取环境变量的值, 传入参数为变量名, 返回指向共享内存中存储该变量值的指针  
32 char* envGet(char* envName);  
33 //用于销毁共享内存  
34 void shmDestroy(int id, void* mem);  
35 //释放分配的内存, 用于程序结束时  
36 void envDestroy();  
37  
38 //进程类型的定义, 包括前台和后台  
39 typedef enum jobType { FG, BG } jobType;  
40 //进程状态的定义, 包括运行和暂停  
41 typedef enum jobStatus { RUN, SUSPEND } jobStatus;
```

```
42 //进程变量的结构，包括pid，进程名，类型，状态，指向前一个和后一个进程的指针
43 typedef struct job* jobNode;
44 typedef struct job
45 {
46     pid_t pid;
47     char name[MAX_LENGTH_COMMAND];
48     jobType type;
49     jobStatus status;
50     jobNode lastJob;
51     jobNode nextJob;
52 } job;
53
54 //进程表的起始地址的全局变量
55 extern job* jobMem;
56
57 //初始化进程表
58 void jobInit();
59 //向进程表内添加进程，返回指向该内存段的指针
60 job* jobAdd(pid_t pid, char* name, jobType type, jobStatus status);
61 //获取进程表中某一进程的内存地址
62 job* jobMemAddr(pid_t pid);
63 //从进程表中删除进程
64 void jobDel(pid_t pid);
65
66 //用于实现管道
67 extern int pipeFd[2];
68
69 //用于实现umask命令
70 extern int* mask;
71 //初始化mask变量
72 void maskInit();
73
74 #endif //environment.h
```

## parse.h

```
1 ****
2 * 名称:      parse.h
3 * 作者:      吴同
4 * 学号:      3170104848
5 * 内容:      与命令解析与执行有关的函数声明和结构体定义
6 ****
7
8 #ifndef PARSE_H
9 #define PARSE_H
10
11 #include "main.h"
12
13 //内建命令hash值的宏定义
14 #define PWD_VALUE 55684515
```

```
15 #define EXIT_VALUE 28663136479
16 #define CLEAR_VALUE 27599028185609
17 #define TIME_VALUE 135774263668
18 #define ECHO_VALUE 28585235552
19 #define CD_VALUE 3855
20 #define LS_VALUE 21204
21 #define EXEC_VALUE 28663128758
22 #define ENVIRON_VALUE 1603339966545515189
23 #define SET_VALUE 66778291
24 #define UNSET_VALUE 275297188529899
25 #define UMASK_VALUE 275289977335510
26 #define TEST_VALUE 135759437335
27 #define BG_VALUE 1932
28 #define FG_VALUE 9636
29 #define JOBS_VALUE 64351991592
30 #define SHIFT_VALUE 247733860343257
31
32 //parser的状态定义，有限状态机
33 typedef enum parseStatus {
34     NEWCMDPRE, NEWCMDPRE_PIPE, NEWCMDREADY,
35     INREDIRECTREADY, OUTREDIRECTREADY } parseStatus;
36
37 //重定向类型的定义
38 typedef enum redirecType { NOREDIRECT, COVER, APPEND } redirecType;
39
40 //参数链表的结构体定义，包括option, argument和指向下一个参数的指针
41 typedef struct cmdarg* argNode;
42 typedef struct cmdarg
43 {
44     char option;
45     char argument[MAX_LENGTH_COMMAND];
46     argNode nextArg;
47 } cmdarg;
48
49 //命令链表的结构体定义
50 typedef struct command* cmdNode;
51 typedef struct command
52 {
53     //命令名
54     char commandName[MAX_LENGTH_COMMAND];
55     //参数列表
56     cmdarg* argList;
57     //是否是后台运行
58     bool isBG;
59     //是否用管道
60     bool pipeIn;
61     bool pipeOut;
62     //重定向类型
63     redirecType isInRedirect;
64     redirecType isOutRedirect;
```

```
65 //重定向的文件名
66 char inFile[MAX_LENGTH_COMMAND];
67 char outFile[MAX_LENGTH_COMMAND];
68 //指向下一个命令的指针
69 cmdNode nextCmd;
70 } command;
71
72 //初始化命令行
73 void init_commandline();
74 //初始化执行脚本文件
75 void init_file(char* fileName);
76 //显示当面目录的名称，用于命令行下
77 void displayDirName();
78 //从文件（包括stdin）获取一行输入
79 bool getCommand(FILE* fp);
80 //解析一行输入，传入输入的字符串
81 void parse(char* command);
82 //执行一系列命令，传入参数列表
83 void carryout(cmdNode cmdList);
84 //执行命令时建立子进程，传入单个命令节点
85 void childProcess(cmdNode thisCmd);
86
87 //初始化命令列表，用于开始解析一行输入时，传入二级指针
88 void cmdListInit(cmdNode* cmdList);
89 //新建命令，传入命令节点，直接在下一个节点新建命令
90 cmdNode newCmd(cmdNode cmdList);
91 //销毁命令
92 void commandListDestroy(cmdNode cmdList);
93
94 //初始化参数列表，用于新建命令时，传入二级指针
95 void argListInit(argNode* argList);
96 //新建参数，传入参数节点，直接在下一个节点新建参数
97 argNode newArg(argNode argListNode);
98 //销毁参数列表
99 void argListDestroy(argNode argList);
100
101 //初始化重定向，src为被重定向的文件指针（stdin/stdout），fileName为重定向的文件名
102 //用于执行命令时
103 void redirectInit(FILE* src, char* fileName, redirectType type);
104 //结束重定向，恢复正常状态
105 void redirectEnd(FILE* dst, int oldFd);
106
107 //初始化信号处理
108 void signalInit();
109 //信号处理函数，用于处理接收到SIGCHLD信号
110 void childExit(int sig_no, siginfo_t* info, void* vcontext);
111 //信号处理函数，用于处理接收到ctrl-Z信号
112 void ctrlZHandler(int sig_no);
113
114 #endif
```

---

builtin.h

---

```
1  /************************************************************************/
2  * 名称:          builtin.h
3  * 作者:          吴同
4  * 学号:          3170104848
5  * 内容:          执行内建命令所调用函数的声明
6  ************************************************************************/
7
8 #ifndef BUILTIN_H //builtin.h
9 #define BUILTIN_H
10
11 #include "parse.h"
12
13 //以下为执行内建命令所调用的函数, 函数名格式为command_[commandName], 传入参数列表
14 //为统一接口, 再添加新命令时, 传入参数一律为argNode argList
15 void command_pwd();
16 void command_time();
17 void command_exit();
18 void command_clear();
19 void command_environ();
20 void command_jobs();
21 void command_fg();
22 void command_bg();
23 void command_ls(argNode argList);
24 void command_echo(argNode argList);
25 void command_cd(argNode argList);
26 void command_exec(argNode argList);
27 void command_set(argNode argList);
28 void command_unset(argNode argList);
29 void command_umask(argNode argList);
30 void command_test(argNode argList);
31 void command_shift(argNode argList);
32
33#endif //builtin.h
```

---

main.c

---

```
1  /************************************************************************/
2  * 名称:          main.c
3  * 作者:          吴同
4  * 学号:          3170104848
5  * 内容:          主函数的实现
6  ************************************************************************/
7
8 #include "parse.h"
9 #include "environment.h"
10
```

```
11 //主函数
12 int main(int argc, char* argv[])
13 {
14     //初始化运行环境
15     envInit();
16     //无参数传入，进入命令行模式
17     if (1 == argc)
18     {
19         init_commandline();
20     }
21     //有参数传入，运行脚本文件
22     else
23     {
24         init_file(argv[1]);
25     }
26     //结束运行环境
27     envDestroy();
28 }
29
30 //严重错误的处理，以红色字体向stderr中输出错误信息，并退出程序
31 void errorExit(char *procName, int errorNo)
32 {
33     fprintf(stderr, REDSTRING"%s failed\n"NORMALSTRING, procName);
34     exit(errorNo);
35 }
36
37 //用于调试，输出"debug"和整数i
38 void debug(int i)
39 {
40     fprintf(stderr, REDSTRING"debug%d\n"NORMALSTRING, i);
41 }
```

## environment.c

```
1 ****
2 * 名称:      environment.c
3 * 作者:      吴同
4 * 学号:      3170104848
5 * 内容:      运行环境有关函数的实现，包括环境变量表，进程表
6 ****
7
8 #include "environment.h"
9
10 //共享内存，环境变量表
11 int envShmID;
12 envVar* envMem;
13
14 //共享内存，进程表
15 int jobShmID;
16 job* jobMem;
```

```
17 //共享内存，mask值
18 int maskShmID;
19 int* mask;
20
21 //用于管道的实现
22 int pipeFd[2];
23
24 //初始化环境
25 void envInit()
26 {
27     //创建共享内存，获取共享内存标识符
28     //如果共享内存创建失败，标识符返回为-1
29     if (-1 == (envShmID = shmget((key_t)1926, sizeof(envVar) * MAX_ENVVAR,
30                                     0666 | IPC_CREAT)))
31     {
32         errorExit("shmget", SHMGET_FAILED);
33     }
34     //将共享内存连接到进程的地址空间
35     //如果共享内存连接失败，返回地址为-1
36     if ((void*)-1 == (envMem = shmat(envShmID, NULL, 0)))
37     {
38         errorExit("shmat", SHMAT_FAILED);
39     }
40     //初始化环境变量表，全部置0
41     memset(envMem, 0, sizeof(envVar) * MAX_ENVVAR);
42
43     //环境变量表第0位保留，存储shell程序运行状态
44     strcpy(envMem[0].name, "status");
45     strcpy(envMem[0].value, "run");
46
47     //初始化pwd变量
48     char currentDir[_POSIX_PATH_MAX];
49    .getcwd(currentDir, _POSIX_PATH_MAX);
50     envAdd("pwd", currentDir);
51     envAdd("shell", "/usr/local/bin" SHELLNAME);
52
53     //初始化进程表
54     jobInit();
55     //初始化mask变量
56     maskInit();
57 }
58
59 //添加环境变量，传入参数为环境变量的名和值
60 void envAdd(char* envName, char* envValue)
61 {
62     //获取hash值
63     int envNameHash = envHash(envName);
64     while (*envMem[envNameHash].name)
65     {
```

```
66     ++envNameHash;
67 }
68 //将变量名和值存入共享内存
69 strcpy(envMem[envNameHash].name, envName);
70 strcpy(envMem[envNameHash].value, envValue);
71 }
72
73 //获取存储环境变量的内存地址，传入环境变量的名
74 envVar* envAddrGet(char* envName)
75 {
76     //计算hash值
77     int envNameHash = envHash(envName);
78     //如果未查到，则返回空指针
79     int count = 0;
80     while (strcmp(envMem[envNameHash].name, envName))
81     {
82         if (MAX_ENVVAR == count)
83         {
84             return NULL;
85         }
86         ++envNameHash;
87         ++count;
88     }
89     //如果查到，返回存储环境变量的内存地址
90     return &envMem[envNameHash];
91 }
92
93 //获取环境变量的值，传入变量名，返回指向共享内存中变量值的指针
94 char* envGet(char* envName)
95 {
96     //获取存储该变量的内存段
97     envVar* envAddr = envAddrGet(envName);
98     //获取该变量的值
99     return envAddr->value;
100 }
101
102 //计算环境变量的hash值，传入变量名，返回hash值
103 int envHash(char* envName)
104 {
105     //将字符串视为131进制数，转换成十进制数后，对MAX_ENVVAR取模
106     int envNameHash = 0;
107     while (*envName++)
108     {
109         envNameHash = envNameHash * 131 + *envName - 'a';
110     }
111     envNameHash = envNameHash % MAX_ENVVAR;
112     return envNameHash;
113 }
114 //释放共享内存，用于程序结束时
```

```
116 void envDestroy()
117 {
118     //释放环境变量表
119     shmDestroy(envShmID, envMem);
120     //释放进程表
121     shmDestroy(jobShmID, jobMem);
122     //释放mask变量
123     shmDestroy(maskShmID, mask);
124 }
125
126 //初始化mask变量
127 void maskInit()
128 {
129     //创建共享内存，获取共享内存标识符
130     //如果共享内存创建失败，标识符返回为-1
131     if (-1 == (maskShmID = shmget((key_t)64, sizeof(int), 0666 |
132         IPC_CREAT)))
133     {
134         errorExit("shmget", SHMGET_FAILED);
135     }
136     //将共享内存连接到进程的地址空间
137     //如果共享内存连接失败，返回地址为-1
138     if ((void*)-1 == (mask = shmat(maskShmID, NULL, 0)))
139     {
140         errorExit("shmat", SHMAT_FAILED);
141     }
142     //将mask初值设为八进制1000
143     *mask = 512;
144 }
145
146 //初始化进程表
147 void jobInit()
148 {
149     //创建共享内存，获取共享内存标识符
150     //如果共享内存创建失败，标识符返回为-1
151     if (-1 == (jobShmID = shmget((key_t)817, sizeof(jobMem) * MAX_JOB, 0666
152         | IPC_CREAT)))
153     {
154         errorExit("shmget", SHMGET_FAILED);
155     }
156     //将共享内存连接到进程的地址空间
157     //如果共享内存连接失败，返回地址为-1
158     if ((void*)-1 == (jobMem = shmat(jobShmID, NULL, 0)))
159     {
160         errorExit("shmat", SHMAT_FAILED);
161     }
162     //初始化，将进程表中所有进程的pid设为0
163     for (int i = 0; i < MAX_JOB; ++i)
164     {
```

```
164     jobMem[i].pid = 0;
165 }
166
167 //初始化第0个进程，保留
168 jobMem[0].pid = 0;
169 strcpy(jobMem[0].name, SHELLNAME);
170 jobMem[0].type = FG;
171 jobMem[0].status = RUN;
172 jobMem[0].nextJob = NULL;
173 }
174
175 //向进程表中添加进程，返回指向这段内存的指针
176 job* jobAdd(pid_t pid, char* name, jobType type, jobStatus status)
177 {
178     //计算hash值，优化查询，第0个节点保留
179     int i = (pid % MAX_JOB) ? pid % MAX_JOB : 1;
180     while (jobMem[i].pid)
181     {
182         i = (i == MAX_JOB) ? 1 : (i + 1);
183     }
184     //设置这段内存的结构体的值
185     jobMem[i].pid = pid;
186     strcpy(jobMem[i].name, name);
187     jobMem[i].type = type;
188     jobMem[i].status = status;
189     //将这段内存设为链表中的第一个节点，先将第1个节点的nextJob指向第0个节点的nextJob
190     jobMem[i].nextJob = jobMem[0].nextJob;
191     //如果进程表中已经有进程，将新加入的节点的nextJob指向原先的第一个节点
192     if (jobMem[i].nextJob)
193     {
194         jobMem[i].nextJob->lastJob = jobMem + i;
195     }
196     //第1个节点的lastJob指向第0个节点
197     jobMem[i].lastJob = jobMem;
198     //第0个节点的nextJob指向新的第1个节点
199     jobMem[0].nextJob = jobMem + i;
200     //返回指向这段内存的指针
201     return (jobMem + i);
202 }
203
204 //获取进程信息的地址，传入参数为pid，返回指向这段内存的指针
205 job* jobMemAddr(pid_t pid)
206 {
207     //优化查询，计算hash值
208     int i = pid % MAX_JOB;
209     //如果遍历进程表没有找到，则未查询到，返回空指针
210     int count = 0;
211     while (jobMem[i].pid != pid)
212     {
213         if (MAX_JOB == count)
```

```
214     {
215         return NULL;
216     }
217     i = (i + 1) % MAX_JOB;
218     ++count;
219 }
220 //查询到该pid，返回指向这段内存的指针
221 return (jobMem + i);
222 }
223
224 //从进程表中删除进程信息，传入进程的pid
225 void jobDel(pid_t pid)
226 {
227     //优化查询，计算hash值
228     int i = pid % MAX_JOB;
229     while (jobMem[i].pid != pid)
230     {
231         i = (i + 1) % MAX_JOB;
232     }
233     //将该内存单元中的pid置0
234     jobMem[i].pid = 0;
235     //从链表中删除节点
236     if (jobMem[i].lastJob)
237     {
238         jobMem[i].lastJob->nextJob = jobMem[i].nextJob;
239     }
240     if (jobMem[i].nextJob)
241     {
242         jobMem[i].nextJob->lastJob = jobMem[i].lastJob;
243     }
244 }
245
246 //用于释放共享内存的函数，传入标识符和内存地址
247 void shmDestroy(int id, void* mem)
248 {
249     //将共享内存从当前进程中分离
250     //若分离失败，返回-1
251     if (-1 == shmdt(mem))
252     {
253         errorExit("shmdt", SHMDT_FAILED);
254     }
255     //删除共享内存段
256     //若删除失败，返回-1
257     if (-1 == shmctl(id, IPC_RMID, 0))
258     {
259         errorExit("shmctl", SHMCTL_FAILED);
260     }
261 }
```

parse.c



```
50 //初始化信号处理
51 void signalInit()
52 {
53     //捕获ctrl-Z信号后，执行ctrlZHandler函数
54     signal(SIGTSTP, ctrlZHandler);
55     signal(SIGSTOP, ctrlZHandler);
56
57     //将sig_action结构全部清零
58     memset(&sig_action, 0, sizeof(sig_action));
59     //信号处理函数childExit
60     sig_action.sa_sigaction = childExit;
61     sig_action.sa_flags = SA_RESTART | SA_SIGINFO;
62     sigemptyset(&sig_action.sa_mask);
63
64     //SIGCHLD信号处理
65     sigaction(SIGCHLD, &sig_action, &old_sig_action);
66 }
67
68 //捕获SIGCHLD信号执行的函数
69 void childExit(int sig_no, siginfo_t* info, void* vcontext)
70 {
71     //获取发送信号的进程号
72     pid_t pid = info->si_pid;
73     //获取该进程在进程表中的信息
74     job* jobAddr = jobMemAddr(pid);
75     //输出信息
76     if (BG == jobAddr->type)
77     {
78         printf(NORMALSTRING"\n%d\t%s\tend\n", jobAddr->pid, jobAddr->name);
79     }
80     //将进程从进程表中删除
81     jobDel(pid);
82     //清空stdin缓冲区
83     tcflush(fileno(stdin), TCIFLUSH);
84     //开始新的一行，显示当前目录名
85     displayDirName();
86     //返回，等待读入
87     return;
88 }
89
90 //处理ctrl-Z信号，将正在运行的进程挂起
91 void ctrlZHandler(int sig_no)
92 {
93     //获取进程表中第1个进程的信息
94     job* suspJob = jobMem[0].nextJob;
95     if (suspJob)
96     {
97         //向该进程发送SIGSTOP信号
98         kill(suspJob->pid, SIGSTOP);
99         //更新进程表
```

```
100     suspJob->status = SUSPEND;
101     suspJob->type = BG;
102     //输出信息
103     printf(NORMALSTRING"\n%d\t%s\tsuspend\n", suspJob->pid,
104            suspJob->name);
105 }
106
107 //显示当前目录名
108 void displayDirName()
109 {
110     //从环境变量pwd读取目录的绝对路径
111     char* currentDir = envGet("pwd");
112     //获取目录名，新建一个指向字符串结尾的指针
113     char* p_dirName = currentDir + strlen(currentDir) - 1;
114     //当前目录不是根目录
115     if (strlen(currentDir) > 1)
116     {
117         //指针向前移动，直到遇到第一个/
118         while (*p_dirName != '/')
119         {
120             --p_dirName;
121         }
122         ++p_dirName;
123     }
124     //显示目录名
125     printf(BLUESTRING"%s -> "NORMALSTRING, p_dirName);
126     //清空stdout缓冲区
127     fflush(stdout);
128 }
129
130 //读入命令并解析，传入参数为读取命令的文件
131 bool getCommand(FILE* fp)
132 {
133     //从文件中读取一行
134     char currentCommand[MAX_LENGTH_COMMAND];
135     if (fgets(currentCommand, MAX_LENGTH_COMMAND, fp))
136     {
137         //如果命令为help，直接进行字符串替换
138         if (!strcmp(currentCommand, "help\n"))
139         {
140             strcpy(currentCommand, "cat /usr/local/bin/readme_mysh | more\n");
141         }
142         //解析命令
143         parse(currentCommand);
144         return true;
145     }
146     //读入EOF
147     else
148     {
```

```
149         return false;
150     }
151 }
152
153 //解析字符串，实现为有限状态机
154 void parse(char* command)
155 {
156     //新建命令列表
157     cmdNode command_list;
158     cmdListInit(&command_list);
159     cmdNode thisCmd = command_list;
160     argNode thisArg;
161     //parser状态
162     parseStatus status = NEWCMDPRE;
163     //状态及控制信号
164     bool optionReady = false;
165     //以空格和tab为分隔符，分割字符串为单词
166     char* word;
167     while ((word = strsep(&command, " \t\n")))
168     {
169         //连续多个空格或tab的情况
170         if (0 == strlen(word))
171         {
172             continue;
173         }
174         //判断当前状态
175         switch (status)
176         {
177             //准备读取新命令
178             case NEWCMDPRE_PIPE:
179             case NEWCMDPRE:
180                 //行首为空格或tab的情况
181                 if (' ' == *word || '\t' == *word)
182                 {
183                     break;
184                 }
185                 //建立新命令节点
186                 thisCmd = newCmd(thisCmd);
187                 strcpy(thisCmd->commandName, word);
188                 thisArg = thisCmd->argList;
189                 //管道准备接收
190                 if (NEWCMDPRE_PIPE == status)
191                 {
192                     thisCmd->pipeIn = true;
193                 }
194                 //状态转换为正在读取命令
195                 status = NEWCMDREADY;
196                 break;
197                 //正在读取命令
198                 case NEWCMDREADY:
```

```
199         //处理特殊符号
200     if (1 == strlen(word))
201     {
202         //后台运行
203         if ('&' == *word)
204         {
205             thisCmd->isBG = true;
206             status = NEWCMDPRE;
207             break;
208         }
209         //管道
210         else if ('|' == *word)
211         {
212             thisCmd->pipeOut = true;
213             status = NEWCMDPRE_PIPE;
214             break;
215         }
216         //重定向输入
217         else if ('<' == *word)
218         {
219             thisCmd->isInRedirect = COVER;
220             status = INREDIRECTREADY;
221             break;
222         }
223         //重定向输出
224         else if ('>' == *word)
225         {
226             thisCmd->isOutRedirect = COVER;
227             status = OUTREDIRECTREADY;
228             break;
229         }
230     }
231     //处理特殊符号
232     else if (2 == strlen(word))
233     {
234         //重定向输入
235         if ('<' == word[0] && '<' == word[1])
236         {
237             thisCmd->isInRedirect = APPEND;
238             status = INREDIRECTREADY;
239             break;
240         }
241         //重定向输出
242         else if ('>' == word[0] && '>' == word[1])
243         {
244             thisCmd->isOutRedirect = APPEND;
245             status = OUTREDIRECTREADY;
246             break;
247         }
248     }
```

```
249         //准备读入选项
250         if ('-' == *word)
251     {
252         //选项只允许一个字符
253         if (2 != strlen(word))
254         {
255             printf(REDSTRING"illegal option\n"NORMALSTRING);
256             return;
257         }
258         thisArg = newArg(thisArg);
259         thisArg->option = word[1];
260         optionReady = true;
261         break;
262     }
263     //选项后紧接着参数
264     else if (optionReady)
265     {
266         strcpy(thisArg->argument, word);
267         optionReady = false;
268         break;
269     }
270     //没有选项的参数
271     else
272     {
273         thisArg = newArg(thisArg);
274         strcpy(thisArg->argument, word);
275         break;
276     }
277     //准备读入重定向输入的文件
278     case INREDIRECTREADY:
279         strcpy(thisCmd->inFile, word);
280         status = NEWCMDREADY;
281         break;
282     //准备读入重定向输出的文件
283     case OUTREDIRECTREADY:
284         strcpy(thisCmd->outFile, word);
285         status = NEWCMDREADY;
286         break;
287     }
288 }
289 thisCmd = command_list;
290 //初始化管道
291 pipe(pipeFd);
292 //逐个命令执行
293 while ((thisCmd = thisCmd->nextCmd))
294 {
295     carryout(thisCmd);
296 }
297 //销毁命令列表
298 commandListDestroy(command_list);
```

```
299     }
300
301 //执行命令
302 void carryout(cmdNode thisCmd)
303 {
304     //建立新进程
305     pid_t pid = fork();
306     switch (pid)
307     {
308         case -1:
309             errorExit("fork", FORK_FAILED);
310             //子进程
311         case 0:
312             {
313                 //获取主进程在进程表中建立的进程信息
314                 usleep(WAIT_TIME);
315                 job *jobAddr = jobMemAddr(getpid());
316                 //等待主进程初始化完毕
317                 while (SUSPEND == jobAddr->status)
318                 {
319                     usleep(WAIT_TIME);
320                 }
321                 //子进程开始执行命令
322                 childProcess(thisCmd);
323                 exit(0);
324             }
325             //主进程
326         default:
327             {
328                 //在进程表中建立新的进程信息
329                 job *childAddr = jobAdd(pid, thisCmd->commandName,
330                                         (thisCmd->isBG) ? BG : FG, SUSPEND);
331                 //后台进程输出进程信息
332                 if (thisCmd->isBG)
333                 {
334                     printf("%d\t%s\n", pid, thisCmd->commandName);
335                 }
336                 //前台进程
337                 else
338                 {
339                     //SIGCHLD信号处理恢复默认
340                     sigaction(SIGCHLD, &old_sig_action, NULL);
341                     //如果从管道读入，主进程关闭管道，以免影响子进程读取
342                     if (thisCmd->pipeIn)
343                     {
344                         close(pipeFd[0]);
345                         close(pipeFd[1]);
346                     }
347                 }
348             //子进程开始运行
349 }
```

```
349     childAddr->status = RUN;
350     //前台进程
351     if (!thisCmd->isBG)
352     {
353         //阻塞
354         pause();
355         //前台进程结束，在进程表中删除进程信息
356         if (RUN == childAddr->status)
357         {
358             jobDel(pid);
359         }
360         //恢复对SIGCHLD信号的捕获和处理
361         sigaction(SIGCHLD, &sig_action, NULL);
362     }
363     //更新umask
364     if (*mask < 512)
365     {
366         umask(*mask);
367     }
368 }
369 }
370 }
371 //子进程的行为
372 void childProcess(cmdNode thisCmd)
373 {
374     //备份旧的stdin/stdout文件描述符
375     int oldInFd = dup(fileno(stdin));
376     int oldOutFd = dup(fileno(stdout));
377     //管道重定向，从管道接收
378     if (thisCmd->pipeIn)
379     {
380         dup2(pipeFd[0], fileno(stdin));
381         close(pipeFd[0]);
382         close(pipeFd[1]);
383     }
384     //管道重定向，从管道读取
385     if (thisCmd->pipeOut)
386     {
387         dup2(pipeFd[1], fileno(stdout));
388         close(pipeFd[0]);
389         close(pipeFd[1]);
390     }
391     //文件重定向stdin
392     switch (thisCmd->isInRedirect)
393     {
394         case NOREDIRECT:
395             break;
396         case COVER:
397             {
398 }
```

```
399     redirectInit(stdin, thisCmd->inFile, COVER);
400     break;
401 }
402 case APPEND:
403 {
404     redirectInit(stdin, thisCmd->inFile, APPEND);
405     break;
406 }
407 }
408 //文件重定向stdout
409 switch (thisCmd->isOutRedirect)
410 {
411     case NOREDIRECT:
412         break;
413     case COVER:
414     {
415         redirectInit(stdout, thisCmd->outFile, COVER);
416         break;
417     }
418     case APPEND:
419     {
420         redirectInit(stdout, thisCmd->outFile, APPEND);
421         break;
422     }
423 }
424 //命令名hash值
425 unsigned long long commandNameValue = 0;
426 for (int i = 0; thisCmd->commandName[i]; ++i)
427 {
428     commandNameValue = commandNameValue * 1926 + thisCmd->commandName[i]
429         - 'a';
430 }
431 //执行命令
432 switch (commandNameValue)
433 {
434     //内建命令，直接调用函数
435     case PWD_VALUE:
436         command_pwd();
437         break;
438     case EXIT_VALUE:
439         command_exit();
440         break;
441     case CLEAR_VALUE:
442         command_clear();
443         break;
444     case TIME_VALUE:
445         command_time();
446         break;
447     case ECHO_VALUE:
448         command_echo(thisCmd->argList);
```

```
448     break;
449 case CD_VALUE:
450     command_cd(thisCmd->argList);
451     break;
452 case LS_VALUE:
453     command_ls(thisCmd->argList);
454     break;
455 case EXEC_VALUE:
456     command_exec(thisCmd->argList);
457     break;
458 case ENVIRON_VALUE:
459     command_environ();
460     break;
461 case SET_VALUE:
462     command_set(thisCmd->argList);
463     break;
464 case UNSET_VALUE:
465     command_unset(thisCmd->argList);
466     break;
467 case UMASK_VALUE:
468     command_umask(thisCmd->argList);
469     break;
470 case TEST_VALUE:
471     command_test(thisCmd->argList);
472     break;
473 case JOBS_VALUE:
474     command_jobs();
475     break;
476 case SHIFT_VALUE:
477     command_shift(thisCmd->argList);
478     break;
479 case FG_VALUE:
480     command_fg();
481     break;
482 case BG_VALUE:
483     command_bg();
484     break;
485 //外部命令
486 default:
487 {
488     //二维数组，存储参数列表
489     char args[MAX_LENGTH_COMMAND][MAX_LENGTH_COMMAND];
490     //存储指向每个字符串指针的数组
491     char *argp[MAX_LENGTH_COMMAND];
492     argNode thisArg = thisCmd->argList;
493     int i = 0;
494     //数组第一行为调用的文件名
495     strcpy(args[i], thisCmd->commandName);
496     //将参数列表转换为二维数组
497     while ((thisArg = thisArg->nextArg))
```

```
498     {
499         //有选项
500         if ('\0' != thisArg->option)
501         {
502             sprintf(args[++i], "-%c", thisArg->option);
503         }
504         //有参数
505         if ('\0' != thisArg->argument[0])
506         {
507             strcpy(args[++i], thisArg->argument);
508         }
509     }
510     //数组下一行为NULL， 表示结束
511     argp[++i] = NULL;
512     //为指针数组依次赋值
513     while (--i >= 0)
514     {
515         argp[i] = args[i];
516     }
517     //调用exec族执行外部命令
518     execvp(thisCmd->commandName, argp);
519     break;
520 }
521 }
522 //恢复文件重定向
523 if (NOREDIRECT != thisCmd->isInRedirect)
524 {
525     redirectEnd(stdin, oldInFd);
526 }
527 if (NOREDIRECT != thisCmd->isOutRedirect)
528 {
529     redirectEnd(stdout, oldOutFd);
530 }
531 //恢复管道重定向
532 if (thisCmd->pipeIn)
533 {
534     dup2(oldInFd, fileno(stdin));
535 }
536 if (thisCmd->pipeOut)
537 {
538     dup2(oldOutFd, fileno(stdout));
539 }
540 }
541
542 //初始化重定向
543 void redirectInit(FILE* src, char* fileName, redirecType type)
544 {
545     //按照重定向的方式打开文件
546     int fd = 0;
547     switch (type)
```

```
548 {
549     case COVER:
550         fd = open(fileName, O_RDWR | O_CREAT, 0644);
551         break;
552     case APPEND:
553         fd = open(fileName, O_RDWR | O_APPEND | O_CREAT, 0644);
554         break;
555     default:
556         break;
557 }
558 //将文件描述符复制到stdin/stdout
559 if (-1 == dup2(fd, fileno(src)))
560 {
561     errorExit("dup", DUP_FAILED);
562 }
563 //关闭文件描述符
564 close(fd);
565 }

566 //重定向结束，恢复正常状态，传入重定向的目标和旧的文件描述符备份
567 void redirectEnd(FILE* dst, int oldFd)
568 {
569     //将旧的文件描述符备份复制到stdin/stdout
570     if (-1 == dup2(oldFd, fileno(dst)))
571     {
572         errorExit("dup", DUP_FAILED);
573     }
574     //关闭旧的文件描述符
575     close(oldFd);
576 }

577 }

578 //初始化命令链表
579 void cmdListInit(cmdNode* cmdList)
580 {
581     //为哨兵节点分配内存
582     *cmdList = (cmdNode)malloc(sizeof(command));
583     if (NULL == *cmdList)
584     {
585         errorExit("malloc", MALLOC_FAILED);
586     }
587     (*cmdList)->nextCmd = NULL;
588 }

589 }

590 //建立新命令节点
591 cmdNode newCmd(cmdNode cmdListNode)
592 {
593     //为新节点分配内存
594     if (NULL == (cmdListNode->nextCmd = (cmdNode)malloc(sizeof(command))))
595     {
596         errorExit("malloc", MALLOC_FAILED);
```

```
598     }
599     //初始化新节点
600     cmdNode newNode = cmdListNode->nextCmd;
601     argListInit(&newNode->argList);
602     newNode->isInRedirect = NOREDIRECT;
603     newNode->isOutRedirect = NOREDIRECT;
604     newNode->isBG = false;
605     newNode->pipeIn = false;
606     newNode->pipeOut = false;
607     newNode->nextCmd = NULL;
608     return newNode;
609 }
610
611 //销毁命令链表
612 void commandListDestroy(cmdNode cmdList)
613 {
614     //逐个节点释放
615     cmdNode nextCmd;
616     while (cmdList)
617     {
618         nextCmd = cmdList->nextCmd;
619         //释放这一命令的参数链表
620         argListDestroy(cmdList->argList);
621         free(cmdList);
622         cmdList = nextCmd;
623     }
624 }
625
626 //初始化参数链表
627 void argListInit(argNode* argList)
628 {
629     //为哨兵节点分配内存
630     *argList = (argNode)malloc(sizeof(cmdarg));
631     if (NULL == *argList)
632     {
633         errorExit("malloc", MALLOC_FAILED);
634     }
635     (*argList)->nextArg = NULL;
636 }
637
638 //建立新参数节点
639 argNode newArg(argNode argListNode)
640 {
641     //为新节点分配内存
642     if (NULL == (argListNode->nextArg = (argNode)malloc(sizeof(cmdarg))))
643     {
644         errorExit("malloc", MALLOC_FAILED);
645     }
646     //初始化新节点
647     argNode newNode = argListNode->nextArg;
```

```

648     newNode->option = '\0';
649     newNode->argument[0] = '\0';
650     newNode->nextArg = NULL;
651     return newNode;
652 }
653
654 //销毁参数列表, 用于命令执行完毕后
655 void argListDestroy(argNode argList)
656 {
657     argNode nextArg;
658     //逐个节点销毁链表, 释放内存
659     while (argList)
660     {
661         nextArg = argList->nextArg;
662         free(argList);
663         argList = nextArg;
664     }
665 }
```

**builtin.c**

```

1  ****
2  * 名称:      builtin.c
3  * 作者:      吴同
4  * 学号:      3170104848
5  * 内容:      内建命令所调用函数的实现
6  ****
7
8 #include "builtin.h"
9 #include "parse.h"
10 #include "environment.h"
11
12 //pwd命令的实现, 显示当前工作目录的绝对路径
13 void command_pwd()
14 {
15     //从环境变量表中获取名为pwd的变量值并输出
16     char *currentDir = envGet("pwd");
17     printf("%s\n", currentDir);
18 }
19
20 //time命令的实现, 显示当前时刻的时间
21 void command_time()
22 {
23     //获取当前时间并转换为字符串输出
24     time_t currentTime = time(NULL);
25     printf("%s", ctime(&currentTime));
26 }
27
28 //cd命令的实现, 切换工作目录, 如果没有传入参数, 执行pwd
29 void command_cd(argNode argList)
```

```
30  {
31      //第一个参数为切换到的目录
32      if (argList->nextArg)
33      {
34          //获取新目录
35          char *newDir;
36          newDir = argList->nextArg->argument;
37          //系统调用，切换工作目录
38          if (chdir(newDir))
39          {
40              errorExit("change directory", CD_FAILED);
41          }
42          else
43          {
44              //切换成功，更新环境变量
45             .getcwd(envGet("pwd"), _POSIX_PATH_MAX);
46          }
47      }
48      //没有传入参数，执行pwd
49      else
50      {
51          command_pwd();
52      }
53  }

54 //echo命令的实现，输出若干字符串，输出的字符串之间一律以一个空格分隔
55 void command_echo(argNode argList)
56 {
57     //获取参数列表，遍历输出
58     argNode thisArg = argList;
59     while ((thisArg = thisArg->nextArg))
60     {
61         printf("%s ", thisArg->argument);
62     }
63     printf("\n");
64 }

65 //clear命令的实现，清屏
66 void command_clear()
67 {
68     //直接输出屏幕控制符，实现清除屏幕
69     printf("\033[2J\033[0;0H");
70 }

71 //exit命令的实现，退出程序
72 void command_exit()
73 {
74     //设置第0个环境变量的值为exit，父进程处理
75     strcpy(envMem[0].value, "exit");
76 }
```

```
80
81 //ls命令的实现，列出当前目录下的所有文件
82 void command_ls(argNode argList)
83 {
84     //第1个参数为目录，如果没有参数，则为当前目录
85     char* dirName = argList->nextArg ? argList->nextArg->argument :
86         envGet("pwd");
87     //打开目录
88     DIR* dp = opendir(dirName);
89     if (!dp)
90     {
91         errorExit("directory open", OPENDIR_FAILED);
92     }
93     //读取目录
94     struct dirent *dirp;
95     while ((dirp = readdir(dp)))
96     {
97         //忽略隐藏文件
98         if ('.' == dirp->d_name[0])
99         {
100             continue;
101         }
102         //根据文件类型以不同颜色输出
103         switch (dirp->d_type)
104         {
105             //普通文件，黄色
106             case DT_REG:
107                 printf(YELLOWSTRING"%s\n"NORMALSTRING, dirp->d_name);
108                 break;
109             //目录文件，蓝色
110             case DT_DIR:
111                 printf(BLUESTRING"%s\n"NORMALSTRING, dirp->d_name);
112                 break;
113             //符号链接，紫色
114             case DT_LNK:
115                 printf(PURPLESTRING"%s\n"NORMALSTRING, dirp->d_name);
116                 break;
117             //其他文件
118             default:
119                 printf("%s\n", dirp->d_name);
120         }
121     }
122     //关闭目录
123     closedir(dp);
124     //清空stdout缓冲区
125     fflush(stdout);
126 }
127 //exec命令的实现，执行命令然后退出shell
128 void command_exec(argNode argList)
```

```
129 {
130     //参数列表为新命令
131     if (argList->nextArg)
132     {
133         //新建命令节点
134         cmdNode newCmd;
135         cmdListInit(&newCmd);
136         //第1个参数为命令名
137         strcpy(newCmd->commandName, argList->nextArg->argument);
138         //后续参数为新命令的参数
139         argListInit(&newCmd->argList);
140         newCmd->isInRedirect = NOREDIRECT;
141         newCmd->isOutRedirect = NOREDIRECT;
142         newCmd->isBG = false;
143         newCmd->argList->nextArg = argList->nextArg->nextArg;
144         newCmd->nextCmd = NULL;
145         //执行命令
146         carryout(newCmd);
147         //执行完命令后，退出shell
148         command_exit();
149     }
150 }
151
152 //environ命令的实现，列出所有环境变量
153 void command_environ()
154 {
155     //遍历环境变量表
156     for (int i = 0; i < MAX_ENVVAR; ++i)
157     {
158         if (*envMem[i].name)
159             printf("%s=%s\n", envMem[i].name, envMem[i].value);
160     }
161 }
162
163 //set命令的实现，设置环境变量，如果没有参数执行environ
164 void command_set(argNode argList)
165 {
166     //如果没有参数，执行environ
167     if (!argList->nextArg)
168     {
169         command_environ();
170         return;
171     }
172     //第一个参数为变量名，第二个参数为变量值
173     char* envName = argList->nextArg->argument;
174     if (!*envName)
175     {
176         return;
177     }
178     //获取变量所在的地址
```

```
179 envVar* envAddr = envAddrGet(envName);
180 //如果该环境变量已经存在，更新其值
181 if (envAddr)
182 {
183     strcpy(envAddr->value, argList->nextArg->nextArg->argument);
184 }
185 //如果该环境变量不存在，添加环境变量
186 else
187 {
188     envAdd(envName, argList->nextArg->nextArg->argument);
189 }
190 }

191 //unset命令的实现，删除环境变量
192 void command_unset(argNode argList)
193 {
194     //如果参数为空，直接返回
195     if (!argList->nextArg)
196     {
197         return;
198     }
199     //第1个参数为环境变量的名
200     char* envName = argList->nextArg->argument;
201     if (!*envName)
202     {
203         return;
204     }
205     //status和pwd变量受保护，禁止删除
206     if (!strcmp(envName, "pwd") || !strcmp(envName, "status"))
207     {
208         return;
209     }
210     //获取环境变量的地址
211     envVar* envAddr = envAddrGet(envName);
212     //如果没有这一变量，直接返回
213     if (!envAddr)
214     {
215         return;
216     }
217     //将环境变量表中的结构体清零
218     *envAddr->name = 0;
219     *envAddr->value = 0;
220 }

221 }

222 //umask命令的实现，设置umask值
223 void command_umask(argNode argList)
224 {
225     //第1个参数为mask值
226     char* mask_str = argList->nextArg->argument;
227     //将八进制字符串形式的mask值转换为mode_t型整数，存储于共享内存中
```

```
229     *mask = (mode_t)strtol(mask_str, NULL, 8);
230 }
231
232 //test命令的实现，用于测试判断条件是否成立
233 void command_test(argNode argList)
234 {
235     //从第1个参数中读入选项
236     char option = argList->nextArg->option;
237     switch (option)
238     {
239         //n选项，判断字符串是否不空
240         case 'n':
241             if (strlen(argList->nextArg->argument))
242             {
243                 printf("true\n");
244             }
245             else
246             {
247                 printf("false\n");
248             }
249             break;
250         //z选项，判断字符串是否为空
251         case 'z':
252             if (!strlen(argList->nextArg->argument))
253             {
254                 printf("true\n");
255             }
256             else
257             {
258                 printf("false\n");
259             }
260             break;
261     }
262 }
263
264 //shift命令的实现，从标准输入读入字符串，将读入的参数左移输出，一般用在管道中
265 void command_shift(argNode argList)
266 {
267     //第一个参数为左移的位数
268     int shift = atoi(argList->nextArg->argument);
269     //从标准输入读入字符串
270     char line[MAX_LENGTH_COMMAND];
271     char* line_p = line;
272     fgets(line, MAX_LENGTH_COMMAND, stdin);
273     //左移，以空格和tab分隔
274     for (int i = 0; i < shift; ++i)
275     {
276         //忽略连续的空格和tab
277         while (' ' == *line_p || '\t' == *line_p)
278         {
```

```
279     ++line_p;
280 }
281 strsep(&line_p, " \t\n");
282 }
283 //输出新字符串
284 while (' ' == *line_p || '\t' == *line_p)
285 {
286     ++line_p;
287 }
288 printf("%s", line_p);
289 }
290
291 //jobs命令的实现，输出所有后台进程的信息
292 void command_jobs()
293 {
294     //遍历进程表
295     for (int i = 1; i < MAX_JOB; ++i)
296     {
297         if (jobMem[i].pid && BG == jobMem[i].type)
298         {
299             //输出正在运行的后台进程
300             if (RUN == jobMem[i].status)
301             {
302                 printf("%d\t%s\trunning\n", jobMem[i].pid, jobMem[i].name);
303             }
304             //输出挂起的进程
305             else
306             {
307                 printf("%d\t%s\tsuspend\n", jobMem[i].pid, jobMem[i].name);
308             }
309         }
310     }
311 }
312
313 //fg命令的实现，将后台进程转入前台
314 void command_fg()
315 {
316     //获取进程表的第一个进程
317     job* bgJob = jobMem[0].nextJob->nextJob;
318     //判断后台进程是否存在
319     if (bgJob)
320     {
321         //输出进程信息
322         printf("%d\t%s\trunning\n", bgJob->pid, bgJob->name);
323         //阻塞，等待进程退出信号
324         while (true);
325     }
326     //后台进程不存在
327     else
328     {
```

```
329         printf("no current job\n");
330     }
331 }
332
333 //bg命令的实现，将挂起的进程转为后台运行
334 void command_bg()
335 {
336     //遍历进程表，查找挂起的进程
337     for (int i = 1; i < MAX_JOB; ++i)
338     {
339         //判断是否为挂起的进程
340         if (jobMem[i].pid && BG == jobMem[i].type && SUSPEND ==
341             jobMem[i].status)
342         {
343             //更新进程表信息
344             jobMem[i].status = RUN;
345             //发送继续执行信号
346             kill(jobMem[i].pid, SIGCONT);
347             //输出进程信息
348             printf("%d\t%s\trunning\n", jobMem[i].pid, jobMem[i].name);
349         }
350     }
}
```

## readme

-----  
This is mysh by Tong Wu from  
Computer Science and Technology College of  
Zhejiang University.  
-----

## Introduction

Mysh is a simple shell for Unix/Linux, which is implemented for homework  
of Linux Course.

A shell is a program that runs between the user layer and the OS kernel.  
It parses the command which  
is input from the keyboard and uses system call or execute other programs,  
and then outputs to the  
screen. A shell usually runs on a virtual terminal in Unix/Linux system.  
Besides, a shell program  
can execute a shell script, which contains a series of commands.

## Installing mysh

To compile:  
\$ make

To clean the files from compiling:

```
$ make clean
```

To move the executable file into \$PATH:

```
$ make install
```

To uninstall mysh from \$PATH:

```
$ make uninstall
```

#### Features

-----

1. Supports inner commands: pwd, time, clear, exit, environ, jobs, fg, bg, ls, echo, cd, exec, set, unset, umask, test, shift.
2. Supports external commands.
3. Supports shell scripts.
4. Supports I/O redirection.
5. Supports pipe.
6. Supports job management, including foreground, background, hangup.

#### Instructions

-----

##### 1. Basic operation

Some basic inner commands in mysh are as followings.

```
-time    print the current time
-clear   clear the terminal screen
-echo    print strings onto the screen
-shift   left-shift arguments in a line read from the keyboard
-test    test whether the arguments meets some condition
-exec    execute a command and the exit the shell
-exit    exit the shell
```

##### 2. File system operation

In Unix/Linux system, anything is referred as a file, including regular files, directory files,

symbolic links and so on. Some inner commands are about the file system:

```
-pwd     print the current working directory
-cd      change the current working directory
        example: cd /
-ls      list all the files of a directory
        example: ls /etc
-umask   change the current umask
        example: umask 022
```

Umask is a four-digit octal number. The privilege represented by the mask is banned.

### 3. Job management operation

Jobs in a shell can be divided into foreground job and background job. A foreground job and several background jobs can be carried out at the same time. Commands ending with & is background command.

For example:

```
$ sleep 3 &
```

When a background command is carried out, command \$ fg can convert it into the foreground. After entering command \$ jobs, all the background jobs will be printed to the screen.

When a foreground command is carried out, ctrl-Z can hang it up.

Then enter \$ bg command, this command will be continued in the background.

### 4. Environment variable operation

There are some variables which are stored in the shell environment, instead of the child program.

Variables of this kind are called environment variables. Three inner commands are about Environment variables in mysh:

-environ print the names and values of all environment variables

-set set the value of a variable

example: \$ set wt cj

-unset delete an existed environment variable

example: \$ unset wt

### 5. I/O redirection and pipe

As a typical Unix command, it gets input from the keyboard and the results are displayed at the screen. The input source is referred as stdin and the output source is referred as stdout. However,

the input and output source can be changed, which is called I/O redirection. With a redirection operator, the input/output source can be changed with a file. There are three types of redirection:

input cover(<), output cover(>), output append(>>). For example:

```
$ ls > test.out
```

After this command is operated, the list of files in current working directory will not be displayed

on the screen, but will be written into file test.out instead.

```
$ ls >> test.out
```

After this command is operated, the output will also be redirected to file test.out. But if the file

is not empty, the output will be appended to the tail of the file, different from the behavior of

operator>, which will cover the existed file.

```
$ cat < test.in
```

After this command is operated, the command will not read input from the keyboard, but read from file test.in instead.

A concept closely related to I/O redirection is the concept of pipe. The pipe operator | joins the stdout and stdin of two commands. As for the command \$ command1 | command2 the stdout of command1 is redirected to the pipe and the stdin of command2 is also redirected to the pipe. For example:

```
$ ls -l | wc -l
```

The output of this command is the number of lines in the output of \$ ls -l.

#### Acknowledgements

-----  
Thanks to my teacher, Jiangmin Ji, who gives me good lectures about Linux Programming.  
-----

#### 运行结果:

执行 \$ make install 命令, 将 mysh 安装在/usr/local/bin 中。将 mysh 设为终端的默认启动 shell。



图 7: 设置 terminal 的默认 shell 为 mysh

以下程序测试的截图中包含了实验要求中的全部功能:

```
wutong -> help
-----
This is mysh by Tong Wu from
Computer Science and Technology College of
Zhejiang University.
-----
Introduction
-----
Mysh is a simple shell for Unix/Linux, which is implemented for homework of Linux Course.

A shell is a program that runs between the user layer and the OS kernel. It parses the command which
is input from the keyboard and uses system call or execute other programs, and then outputs to the
screen. A shell usually runs on a virtual terminal in Unix/Linux system. Besides, a shell program
can execute a shell script, which contains a series of commands.

Installing mysh
-----
To compile:
$ make

To clean the files from compiling:
$ make clean
```

图 8: 显示 readme 帮助文件

```
Users -> clear
```

图 9: 测试 clear

```
wutong -> pwd
/Users/wutong
wutong -> cd ..
Users -> pwd
/Users
Users -> ls /etc
emond.d
periodic
manpaths
services-previous
dnsextd.conf
rc.common
csh.logout~orig
auto_master
csh.login
mach_init.d
syslog.conf
rtadvd.conf-previous
syslog.conf-previous
krb5.keytab
sudoers.d
ssl
nanorc
ttys-previous
```

图 10: 测试命令集 1

```
Users -> umask 012
Users -> exit
[进程已完成]
```

图 11: 测试命令集 2

```
wutong -> sleep 5 & sleep 10 &
41015  sleep
41016  sleep
wutong -> jobs
41016  sleep  running
41015  sleep  running
wutong -> ls > test.out
wutong ->
41015  sleep  end
wutong -> fg
41016  sleep  running
wutong -> cat test.out
Music
VirtualBox VMs
OneDrive
Creative Cloud Files
Pictures
Qt
Desktop
weini.jpg
Library
Parallels
```

图 12: 测试命令集 3

```
wutong -> sleep 3 &
41586  sleep
wutong -> ls
41586  sleep  end
wutong ->
wutong ->
```

图 13: 特殊样例：清空输入缓冲区

```
wutong -> sleep 3
^Z
42438  sleep  suspend
wutong -> jobs
42438  sleep  suspend
wutong -> bg
42438  sleep  running
```

图 14: 测试命令集 4

```
wutong -> cat test.in
hello Winnie      West           East          friend
wutong -> cat test.out
123
4556
789
wutong -> shift 3 < test.in >> test.out
wutong -> cat test.out
123
4556
789
East          friend
```

图 15: 测试命令集 5

```
wutong -> echo hello Winnie      West           East          friend
hello Winnie West East friend
wutong -> time | shift 3
23:55:53 2019
wutong -> time
Wed Aug 14 23:55:55 2019
wutong -> exec test -n Winnie
true

[进程已完成]
```

图 16: 测试命令集 6

```
wutong -> environ
status=run
shell=/usr/local/bin/mysh
pwd=/Users/wutong
wutong -> set wt cj
wutong -> set
status=run
shell=/usr/local/bin/mysh
wt=cj
pwd=/Users/wutong
wutong -> unset wt
wutong -> environ
status=run
shell=/usr/local/bin/mysh
pwd=/Users/wutong
wutong -> 
```

图 17: 测试命令集 7

```
[→ ~ cat test.sh
pwd
time
echo hello
cd ..
pwd
[→ ~ mysh test.sh
/Users/wutong
Wed Aug 14 23:58:54 2019
hello
/Users
→ ~ ]
```

图 18: 执行脚本文件（在 zsh 下调用 mysh）

### 三、 讨论与心得

通过本次实验，我初步掌握了 Unix/Linux 的系统编程，掌握了文件 IO、进程、信号、共享内存等系统调用的特点和用法，并学会了 `makefile` 文件的编写。在编写 `myshell` 程序之前，我通读了一遍 Neil & Richard 所著的《Linux 程序设计》一书。虽然是囫囵吞枣，但仍然获益匪浅，了解了 Linux 系统编程的知识。在编写 `myshell` 的过程中，我综合运用所掌握的知识，并在实践中不断改进，从软件的框架到内存中的数据结构，都经过了多次的结构性调整。特别是在加入信号后，每添加一个新功能，都会出现一些莫名的 bug，这个问题直到整个程序快要编写完成时才找到原因并得以解决。在此简述编程时遇到的一些问题，以及解决问题时的思考过程。

首先是软件的架构。最初，我完全采用实验指导中的架构，即内部命令直接调用函数，外部命令新建子进程。但随着程序规模的不断增大，这样的架构带来了很多不便之处。虽然直接调用函数执行内部命令的效率更高，但却造成了内部命令与外部命令接口的不统一。比如后台命令，通过捕获 SIGCHLD 信号来进行命令结束时的处理。对于内部命令，没有创建子进程，无法与外部命令形成统一。这样的程序，代码结构复杂混乱，并且可扩展性不强。最终，我将命令执行方式改为所有命令一律新建子进程，在子进程中执行命令，程序的结构清晰得多。

第二是命令的解析过程。按照所给程序框架的思路，应该是首先判断命令的类型，是否包含后台、重定向等功能。这样的处理方式，会导致对字符串的多次遍历，严重影响了程序的执行效率。我在编写过程中，定义了命令这一结构体类型，采用有限状态机的方式，在遍历一次字符串的过程，同时获取了命令的类型、名称和参数等信息，效率较高。

第三是关于信号的处理。最初我在处理 SIGCHLD 信号时，为了能保证换行时再次显示当前目录名，使用了 `setjmp` 这一跳转功能实现信号处理完毕后的返回。但这样做经常导致程序的阻塞。每次对程序进行改动后，都会出现这一 bug，出现一些自己无法理解的现象，再用一些自己无法理解的方式解决。后来，我放弃了这样的做法，而是直接输出一次目录名后正常返回。这样做，不仅代码结构清晰，而且易于维护，不出现奇怪的错误。

最后是程序所用的数据结构。由于共享内存要提前分配好，所以所有的数据结构都基于数组。我对数组的结构进行了若干的优化，加快了查询的效率和访问的效率。比如，我普遍采用了哈希表的思想，对字符串进行哈希运算，再进行查找。进程表的结构中，我在数组之上建立了链表，这样不仅提高遍历和删除的效率，也便于直接找到最新添加的元素。

由于时间有限，个人水平有限，代码仍有很多不足之处。比如，内部命令调用的函数应统一接口，传入参数列表，返回整数；进程间的通信可以使用信号量；内部命令应该支持更多的选项；应该支持 shell 语言。非常遗憾的是，在通读完《Linux 程序设计》后，我原本想阅读一些现有 shell 程序的代码，如经典的 `bash` 和强大的 `zsh`，但时间已经不允许了。未来若有机会，还是要向优秀的 shell 程序学习一个。

在 Linux 程序设计课程上，我感到收获颇多。通过这门课程的学习，我对 Unix/Linux 系统有了更多的了解，编程能力也得到了锻炼，期待在操作系统课程上学到更多的有关知识。