myShell说明文档

1.实验内容

- Shell能解析的命令行如下：

1. 带参数的程序运行功能。

   program arg1 arg2 … argN

2. 重定向功能，将文件作为程序的输入/输出。
   1. “>”表示覆盖写

      program arg1 arg2 … argN > output-file
   2. “>>”表示追加写

      program arg1 arg2 … argN >> output-file
   3. “<”表示文件输入

      program arg1 arg2 … argN < input-file

- Shell能解析的命令行如下：

3. 管道符号“|”，在程序间传递数据。

   programA arg1 … argN | programB arg1 … argN

4. 后台符号& ,表示此命令将以后台运行的方式执行。

   program arg1 arg2 … argN &

5. 工作路径移动命令cd。
6. 程序运行统计mytop。
7. shell退出命令exit。
8. history n显示最近执行的n条指令。

2.实验过程

shell是个能解析输入的命令并且执行该命令的程序，即：

1. 用户输入指令
2. shell读取指令
3. shell解析指令
4. shell执行指令
5. 重复1-4直到exit

2.0 准备

2.0.1 宏定义

```
#define ALL_SIZE 10
#define CMD_LENG 8          //指令最大长度
#define PARA_MAX 64         //参数最大长度
#define HISTORY_NUM 20
#define MAX_LINE 100  //每条指令最多包含100个字符
#define STD_INPUT 0
#define STD_OUTPUT 1

#define  USED        0x1
#define  IS_TASK    0x2
#define  IS_SYSTEM    0x4
#define  BLOCKED    0x8
#define TYPE_TASK    'T'
#define TYPE_SYSTEM    'S'
#define STATE_RUN    'R'
//以下指令可以在MINIX3/include/minix/com.h找到
#define MAX_NR_TASKS 1023
#define SELF    ((endpoint_t) 0x8ace)
#define _MAX_MAGIC_PROC (SELF)
#define _ENDPOINT_GENERATION_SIZE (MAX_NR_TASKS+_MAX_MAGIC_PROC+1)
#define _ENDPOINT_P(e) \
((((e)+MAX_NR_TASKS) % _ENDPOINT_GENERATION_SIZE) - MAX_NR_TASKS)
#define  SLOT_NR(e) (_ENDPOINT_P(e) + 5)
#define _PATH_PROC "/proc"
#define CPUTIME(m, i) (m & (1L << (i)))
const char *cputimenames[] = { "user", "ipc", "kernelcall" };
#define NR_TASKS      5
#define IDLE    ((endpoint_t) -4) /* runs when no one else can run */
#define KERNEL  ((endpoint_t) -1) /* pseudo-process for IPC and scheduling
*/
#define CPUTIMENAMES (sizeof(cputimenames)/sizeof(cputimenames[0]))
```

2.0.2 全局变量

```
char history[HISTORY_NUM][MAX_LINE];
char *buff; //动态分配的内存
int history_num=0;
int k=0;
int mark=0; //记录命令history n中的n
int background=0; //前台、后台任务
char currentdir[20];
char *builtinStr[]={"cd","exit","history","mytop"};//list of builtin
commands
cmd_all *cmd_var; //结构体指针cmd_var 方便管理所有指令
struct proc *proc = NULL, *prev_proc = NULL; //mytop中对进程的管理
const char *cputimenames[] = { "user", "ipc", "kernelcall" };
int nr_total=0;
unsigned int nr_procs, nr_tasks;
```

2.0.3 结构体

- 指令

```
typedef struct CMD_STRUCT //每一条指令结构
{
    char *cmd[CMD_LENG]; //数组元素为字符指针每个指针指向命令的首地址
    char cmdStr[CMD_LENG*PARA_MAX];//cmdStr存my_substring得到的子字符串
    char nextSign; // '|' or'>' or '<'
}cmdStruct;
typedef struct CMD_ALL //所有指令的结构
{
    cmdStruct cmd_all[ALL_SIZE];//定义数组包含ALL_SIZE个cmdstruct结构体
    int cmdPtr;//标明对应的是cmd_all的第几条命令
}cmd_all;
```

eg:

命令`ls -a -l > result.txt`

cmd[0]='ls' cmd[1]='-a' cmd[2]='-l' ...

cmd_var->cmd_all[0].cmdStr='ls -a -l'

nextSign='|'

这样写的好处是能够将那些由多条指令组成的指令能够分开来存储，而不是仅仅放在某个二维数组的一行上

- 进程

```
struct proc  //minix3中对进程定义的结构
{
    int p_flags;  //proc的类型：系统/用户
    endpoint_t p_endpoint; //端点
    pid_t p_pid;
    u64_t p_cpucycles[CPUTIMENAMES]; //cpu周期
    int p_priority;
    endpoint_t p_blocked;
    time_t p_user_time; //用户时间
    vir_bytes p_memory;
    uid_t p_effuid;
    int p_nice; //静态优先级
    char p_name[16+1];
};
```

2.0.4 函数声明

```
int my_init(void);
int my_cd(void);
int my_exit(void);
int my_readLine(char *line);
int my_subString(char *ResultString , char *str , int start , int end);
int my_splitStr(char *resultArr[] , char *str , char *split);
int my_analyCmd(char *line);
int my_builtinCmd(void);
```

```
int my_execute(void);
int my_clearCmd(cmd_all *cmd_var );
int my_history(void);
void parse_file(pid_t pid);
void parse_dir(void);
int print_memory(void);
u64_t cputicks(struct proc *p1, struct proc *p2, int timemode);
void print_procs(struct proc *proc1, struct proc *proc2, int cputimemode);
void get_procs(void);
void getkinfo(void);
int mytop();
```

参数为void表明，若在调用该函数是写入参数会报错

2.1 指令输入 & main函数

```
int main()
{
    char line[MAX_LINE];
    int pid;
    buff=(char *)malloc(10240);
    //给指令结构体分配内存
    cmd_var = (cmd_all *)buff;
    my_init();
    while(1)
    {
        printf("%s",currentdir);
        printf("$");
        my_readLine(line);
        my_analyCmd(line);
        //如果是内置命令，成功执行后清理进程，因为执行内置命令时shell不会启用新的进程
        if(0==my_builtinCmd()) //非内置命令return -1
        {
            my_clearCmd(cmd_var);
            continue;
        }
        //fork一个新进程执行program命令
        else
        {
            pid = fork();
            if(background==1) //后台任务
            {
                if (pid==0)
                {//标准输出重定向到/dev/null
                    freopen("/dev/null","w",stdout);
                    my_execute();

                }//父进程 ignore SIGCHLD
                signal(SIGCHLD,SIG_IGN);
                //子进程结束时，父进程会收到SIGCHLD信号
            }
            else
            {
                if(pid==0)
                {
                    my_execute();
                }//父进程等待子进程执行完
```

```
                waitpid(pid,NULL,0);
            }
        }
        //保证所有命令到执行完再进行clear
        sleep(1);
        my_clearCmd(cmd_var);
    }
    return 0;
}
```

2.2 指令读取与解析

2.2.1 int my_splitStr(char *resultArr[],char *str,char *split)

用了c语言的库函数`char *strtok(char *str, const char *delim)`来分解，`resultArr`这个指针数组就是用来存放分解后的指令的每个部分的，数组中的每个元素指向str分割后的每个小部分

2.2.2 int my_readLine(char *line)

通过一个while循环，`char c=getchar()`一个字符一个字符读取，直到读到换行符跳出循环。在循环过程中，也将字符存入history这个二维数组中，方便后续打印。在读到换行符时，判断是否为history命令，若是，则用`全局变量mark`记录下用户要求的数字

2.2.3 int my_subString(char *ResultString,char *str,int start,int end)

该函数用来拆分输入的命令，并且将分解完的命令存入`字符数组ResultString`。每条指令末尾加上`\0`

2.2.4 int my_analyCmd(char *line)

该函数用来识别是否要在后台运行，以及若是重定向或者有管道，进行指令的拆分

遍历line，一个一个字符比对

2.2.5 int my_builtinCmd(void)

上面四个函数其实都是做了解析指令的准备工作，来方便解析指令。

直接用strcmp将输入的指令的第一节一个一个与内置指令比对，如果是内置指令，则执行对应函数并且返回0；如果输入的不是内置指令，则返回-1

2.3 指令执行

2.3.1 内置指令

2.3.1.1 int my_cd(void)

这个函数的作用是将当前路径切换成用户指定的路径。`int chdir(const char *path)`函数将当前的工作目录变成以参数path所指的目录，成功执行返回0，否则返回-1。`char *getcwd(char *buf, size_tsize)`函数将当前的工作目录绝对路径复制到参数buf所指的内存空间，参数size为buf的大小

2.3.1.2 int my_exit(void)

直接退出shell

`exit(0)`表示成功执行，正常退出　　`exit(1)`表示未成功执行

2.3.1.3 int my_history(void)

该函数用来打印n条输入的历史指令记录。由于在readLine函数中，已将指令存入了history这个二维数组，因此这里只要做一行行打印这一步就好了。

2.3.1.4 int mytop()

1.输出总体内存大小、空闲块大小、缓存大小

打印meminfo中的内容，并根据公式计算打印即可

由`print_memory()`函数实现

```
# cd /proc
# ls
-1      11      151     175     215     29      59      8       loadavg
-2      118     155     19      245     3       6       9       meminfo
-3      12      163     21      246     32      62      cpuinfo mounts
-4      134     165     211     247     4       7       dmap    pci
-5      139     17      212     250     40      73      hz      uptime
1       143     172     213     251     49      76      ipcvecs
107     147     173     214     252     5       79      kinfo
# cat meminfo
4096 65135 53027 52303 5164
# cat kinfo
256 5
```

2.输出总体CPU占比

- `get kinfo()`读取进程数和任务数来计算nr_total（总和）
- `get procs()`获取每个进程信息并存放入进程的结构体中
- `parse_dir()`获取proc中所有的进程pid
- `parse_file(pid_t pid)`该函数传入的参数是进程的pid，然后读取其中的各种参数

/proc/pid/psinfo中，例如 /proc/107/psinfo文件中，查看pid为107的进程信息。每个参数对应含义依次是：版本version，类型type，端点endpt，名字name，状态state，阻塞状态blocked，动态优先级priority，滴答ticks，高周期highcycle，低周期lowcycle，内存memory，有效用户ID effuid，静态优先级nice等。其中会用到的参数有：类型，状态，滴答。进程时间time=ticks/ (u32_t)60。

```
# cd /proc/118
# ls
cmdline environ map     psinfo
# cat psinfo
0 S 65562 random W 31744 7 0 0 0 4531398 135168 0 0 - 4 12 12 0 0 - 31743 0 0 1077274 0 670156
# 
RSION - Please support MobaXterm by subscribing to the professional edition here: https://mobaxterm.mobatek.net
```

- `u64_t cputicks(struct proc *p1, struct proc *p2, int timemode)`要计算两次取差值，来获得进程的滴答。通过相同的endpoint来判断是否为相同的进程
- `print_procs(struct proc *proc1, struct proc *proc2, int cputimemode)`打印cpu占用比

2.3.2 program指令（管道与重定向

首先开了一个数组`int fd[2];`

fd[0]:存放读操作的文件描述符

fd[1]:存放写操作的文件描述符

2.3.2.1 主要函数：

- `pipe(&fd[0])`该函数用于实现无名管道，将fd[2]数组中的两个文件描述符标记管道读和管道写
- `fork()`创建一个与原来进程完全相同的子进程。调用一次返回两次。子进程返回0，父进程返回子进程的pid
- `dup(fd)` 该函数用来复制文件，识别当前未被使用的最小文件操作符将其定向到到fd所指的文件中
    - `int dup(int oldfd)`函数返回一个新的描述符，这个新的描述符是传给它的描述符的拷贝，若出错则返回 - 1。由dup返回的新文件描述符一定是当前可用文件描述符中的最小数值。这函数返回的新文件描述符与参数 filedes 共享同一个文件数据结构。
    - `int dup2(int oldfd,int newfd)`函数返回一个新的文件描述符，若出错则返回 - 1。与 dup 不同的是，dup2 可以用 newfd参数指定新描述符的数值。如果 newfd已经打开,则先将其关闭。如若 oldfd等于 newfd, 则 dup2 返回 newfd, 而不关闭它,同样，返回的新文件描述符与参数 oldfd同一个文件数据结构
- `int execvp(const char *file, char * const argv [])`;该函数会从当前目录中查找到符合参数file的文件名，然后执行该文件，然后将第二个参数传给file（这个函数执行成功是不会返回的，失败返回-1）

2.3.2.2 具体实现

- 管道：首先创建一个管道，fork出一个子进程，然后将第一个进程的标准输出信息写入到管道中，关闭第一个进程的写端，然后打开子进程的读端
- '>'覆盖写 ： 将重定向的文件复制到开辟的fileName数组中，并且打开新文件文件并写入内容（若原来有内容会被直接覆盖，模式为‘w’
- '<'文件输入 ： 将重定向的文件复制到开辟的fileName数组中，并且打开文件并读出内容
- '>>'追加写：将重定向的文件复制到开辟的fileName数组中，并且打开文件并写入内容（其实和覆盖写一样，但是需要将freopen模式改变成追加模式‘a+’

3.实验结果

测试用例

1. cd /your/path
2. ls -a -l
3. ls -a -l > result.txt
4. vi result.txt
5. grep a < result.txt
6. ls -a -l | grep a
7. vi result.txt &
8. mytop
9. history n
10. exit

```
root$cd /root/test
cd succeeded
test$cd /root
cd succeeded
root$ls -a -l
total 4920
drwxr-xr-x   4 root  operator    1664 Mar  9 22:03 .
drwxr-xr-x  17 root  operator    1408 Feb 24 14:11 ..
-rw-r--r--   1 root  operator      44 Sep 14  2014 .exrc
-rw-r--r--   1 root  operator     605 Sep 14  2014 .profile
drwx------   2 root  operator     128 Feb 24 16:31 .ssh
-rwxr-xr-x   1 root  operator  733716 Mar  5 21:09 core.314
-rwxr-xr-x   1 root  operator  733716 Mar  5 21:10 core.315
-rwxr-xr-x   1 root  operator  741908 Mar  5 21:11 core.323
-rwxr-xr-x   1 root  operator    5314 Mar  2 21:31 hello
-rw-r--r--   1 root  operator      65 Feb 24 16:55 hello.c
-rwxr-xr-x   1 root  operator   21281 Mar  5 23:05 myshell
-rw-r--r--   1 root  operator   16214 Mar  2 21:48 myshell.c
-rwxr-xr-x   1 root  operator   21710 Mar  5 23:23 myshell1
-rw-r--r--   1 root  operator   20665 Mar  5 23:23 myshell1.c
-rwxr-xr-x   1 root  operator   12713 Mar  7 14:29 myshell2
-rw-r--r--   1 root  operator   10429 Mar  7 14:28 myshell2.c
-rwxr-xr-x   1 root  operator   21710 Mar  7 14:39 myshell6
-rw-r--r--   1 root  operator   22147 Mar  7 14:38 myshell6.c
-rwxr-xr-x   1 root  operator   18555 Mar  9 22:03 myshell7
-rw-r--r--   1 root  operator   20186 Mar  9 22:03 myshell7.c
-rwxr-xr-x   1 root  operator   21255 Mar  2 21:49 myshellno
-rw-r--r--   1 root  operator      13 Mar  2 21:51 res.txt
-rw-r--r--   1 root  operator       2 Mar  5 21:43 result
-rw-r--r--   1 root  operator    1551 Mar  9 15:48 result.txt
drwxr-xr-x   2 root  operator     256 Mar  7 18:54 test
-rw-r--r--   1 root  operator      48 Feb 27 16:06 test.txt
root$ls -a -l > rs
root$ls -a -l >result.txt
```

(`ls -a -l > rs` 我打错了

```
root$ls -a -l >result.txt
root$vi result.txt
root$grep a < result.txt
total 4920
drwxr-xr-x   4 root  operator    1728 Mar  9 22:04 .
drwxr-xr-x  17 root  operator    1408 Feb 24 14:11 ..
-rw-r--r--   1 root  operator      44 Sep 14  2014 .exrc
-rw-r--r--   1 root  operator     605 Sep 14  2014 .profile
drwx------   2 root  operator     128 Feb 24 16:31 .ssh
-rwxr-xr-x   1 root  operator  733716 Mar  5 21:09 core.314
-rwxr-xr-x   1 root  operator  733716 Mar  5 21:10 core.315
-rwxr-xr-x   1 root  operator  741908 Mar  5 21:11 core.323
-rwxr-xr-x   1 root  operator    5314 Mar  2 21:31 hello
-rw-r--r--   1 root  operator      65 Feb 24 16:55 hello.c
-rwxr-xr-x   1 root  operator   21281 Mar  5 23:05 myshell
-rw-r--r--   1 root  operator   16214 Mar  2 21:48 myshell.c
-rwxr-xr-x   1 root  operator   21710 Mar  5 23:23 myshell1
-rw-r--r--   1 root  operator   20665 Mar  5 23:23 myshell1.c
-rwxr-xr-x   1 root  operator   12713 Mar  7 14:29 myshell2
-rw-r--r--   1 root  operator   10429 Mar  7 14:28 myshell2.c
-rwxr-xr-x   1 root  operator   21710 Mar  7 14:39 myshell6
-rw-r--r--   1 root  operator   22147 Mar  7 14:38 myshell6.c
-rwxr-xr-x   1 root  operator   18555 Mar  9 22:03 myshell7
-rw-r--r--   1 root  operator   20186 Mar  9 22:03 myshell7.c
-rwxr-xr-x   1 root  operator   21255 Mar  2 21:49 myshellno
-rw-r--r--   1 root  operator      13 Mar  2 21:51 res.txt
-rw-r--r--   1 root  operator       2 Mar  5 21:43 result
-rw-r--r--   1 root  operator       0 Mar  9 22:05 result.txt
-rw-r--r--   1 root  operator    1605 Mar  9 22:04 rs
drwxr-xr-x   2 root  operator     256 Mar  7 18:54 test
-rw-r--r--   1 root  operator      48 Feb 27 16:06 test.txt
```

```
root$ls -a -l | grep a
total 4928
drwxr-xr-x   4 root  operator    1728 Mar  9 22:04 .
drwxr-xr-x  17 root  operator    1408 Feb 24 14:11 ..
-rw-r--r--   1 root  operator      44 Sep 14  2014 .exrc
-rw-r--r--   1 root  operator     605 Sep 14  2014 .profile
drwx------   2 root  operator     128 Feb 24 16:31 .ssh
-rwxr-xr-x   1 root  operator  733716 Mar  5 21:09 core.314
-rwxr-xr-x   1 root  operator  733716 Mar  5 21:10 core.315
-rwxr-xr-x   1 root  operator  741908 Mar  5 21:11 core.323
-rwxr-xr-x   1 root  operator    5314 Mar  2 21:31 hello
-rw-r--r--   1 root  operator      65 Feb 24 16:55 hello.c
-rwxr-xr-x   1 root  operator   21281 Mar  5 23:05 myshell
-rw-r--r--   1 root  operator   16214 Mar  2 21:48 myshell.c
-rwxr-xr-x   1 root  operator   21710 Mar  5 23:23 myshell1
-rw-r--r--   1 root  operator   20665 Mar  5 23:23 myshell1.c
-rwxr-xr-x   1 root  operator   12713 Mar  7 14:29 myshell2
-rw-r--r--   1 root  operator   10429 Mar  7 14:28 myshell2.c
-rwxr-xr-x   1 root  operator   21710 Mar  7 14:39 myshell6
-rw-r--r--   1 root  operator   22147 Mar  7 14:38 myshell6.c
-rwxr-xr-x   1 root  operator   18555 Mar  9 22:03 myshell7
-rw-r--r--   1 root  operator   20186 Mar  9 22:03 myshell7.c
-rwxr-xr-x   1 root  operator   21255 Mar  2 21:49 myshellno
-rw-r--r--   1 root  operator      13 Mar  2 21:51 res.txt
-rw-r--r--   1 root  operator       2 Mar  5 21:43 result
-rw-r--r--   1 root  operator    1605 Mar  9 22:05 result.txt
-rw-r--r--   1 root  operator    1605 Mar  9 22:04 rs
drwxr-xr-x   2 root  operator     256 Mar  7 18:54 test
-rw-r--r--   1 root  operator      48 Feb 27 16:06 test.txt
root$vi result.txt &
ex/vi: Vi's standard input and output must be a terminal
root$mytop
main memory: 260540K total, 198772K free, 195612K contig free, 34632K cached
1867520855
CPU states:   0.00% user,   0.19% system,   0.00% kernel,   0.00% idle 99.81% to
tal,
```

```
root$history 4
ls -a -l | grep a
vi result.txt &
mytop
history 4
root$exit
exit succeeded
#
```

4.总结

- 一个代码量很大的project。。。而且实现起来比较困难。通过实验，接触了虚拟机，熟悉了shell是如何运行的，以及一些内置命令的实现方法。尤其是在参考minix系统下对top命令的实现方法下了解了CPU占用比的计算方法。当然myshell仅仅实现了一小部分的命令，还有很多命令没有实现。
- 分配内存一定要小心，尤其是指针一定要分配内存。不然就会一直报段错误，其实就是访问的内存超出了系统给这个程序的内存空间。

```
# clear
# clang myshell1.c -o myshell1
# ./myshell1
[1]    Segmentation fault (core dumped) ./myshell1
#  Segmentation fault (core dumped) ./myshell1
```

- 对vi指令进行后台操作会报错，vim的标准输入输出必须在同一个终端中。