一、 目的

通过实现一个简单的shell，熟悉c语言环境编程，加深对系统调用的理解。

二、 内容与设计思想

1.命令行输入的读取并执行命令

1.1命令行读取与基础准备

首先用fgets()一次读一行输入。当fgets返回null指针时停止循环。

由于execvp函数要求的参数是以null结尾而不是换行符。故先对结尾的换行符进行处理。

后为配合解析命令的需要，改用my_readline实现

利用my_readline

通过readline读入一行的命令，按char字符一个一个读入。line是一个指向保存一行输入的地址空间的char指针。

基本结构为：

```
int my_readLine(char * line)
{
    int index = 0;
    char c;
    int j=0;
    while (1)
    {
        c = getchar();
        if (c == '\n')
        {
            line[index] = '\n';
```

```
            break;
        }
        else
        {
            line[index] = c;
            j++;
        }
        index++;
    }
    return 0;
}
```

因为内置命令history的需要后在此处将命令存入数组。

1.2 命令行解析

1.2.1初始版本

简单的实现如下所示，但是这个形式略有一些死板。不能处理有多余空格的情况。

```
//命令解析
        char *argv[100];  //分配一个字符指针数组
        int i, len, index = 0;
        argv[index++] = buf;
      for (i = 0, len = strlen(buf); i < len; i++)
      {
          if (buf[i] == ' ')
          {
           buf[i] = 0;
          if (buf[i+1] != ' ') {
            argv[index++] = &buf[i+1];
                                }
          }
       }
            argv[index] = NULL; //argv最后一个为NULL
```

1.2.2 改进版本

为了能够处理多余的空格，调整代码如下。

首先对开头的空格做了处理，直到找到第一个非空格，并通过限制条件保证前一个字符为\0的非空格字符的是参数的起始位置。

```
  while (fgets(buf, MAXLINE, stdin) != NULL) {
            //fgets一次读入一行
        if (buf[strlen(buf) - 1] == '\n')
                buf[strlen(buf) - 1] = 0; // replace
newline with null
       //execlp函数要求的参数是以null结尾而不是换行符
        //命令解析
            char *argv[100];  //分配一个字符指针数组即可
            int i=0;
            argv[0]=buf;//取buf方便后续内置命令的判断
            int  len, index = 0;
            while(buf[i] == ' ') {
              //处理开头的空格直到找到第一个非空格
              i++;
            }
            argv[index++] = buf+i;
            for (len = strlen(buf); i < len; i++) {
              if (buf[i] == ' ') {
                buf[i] = 0;
              } else {
               //前一个字符为\0的非空格字符的是参数的起始位置
                if ( (i-1 >= 0) && (buf[i-1] == 0) ) {
                  argv[index++] = buf+i;
                }
              }
            }
            argv[index] = NULL; //argv最后一个为NULL
```

也可以利用strtok命令利用空格分割。

```
 helper=strtok(buffer," ");//用空格分割
```

1.2.3 最终版本

用my_subString将子字符串复制到一个char数组中。因为在管道和重定向的时候需要将'>''|'前后分别存储。不含管道这些的也要存入子数组才能后面用splitStr空格分开 其中ResultString是目标数组的指针，str是原数组的指针。这样传参后就可以操作resultstring和str这两个char数组了。

再用 my_splitStr来将指令用空格分开，在init和cd中还需要对绝对路径用/来拆分。 resultArr[]是指向分隔好的子字符串的指针数组。strtok首次调用时，str指向要分解的字符串，之后再次调用要把str设成NULL即可。split是分割字符的指针。最后一位设置成NULL因为execvp函数要求的参数是以null结尾。

最后用my_analyCmd解析命令。如果该行有‘&’就将标记是否为后台进程的background置1。当检测到|或<或> ，就将这个之前的字符串存入cmdStr中，再通过空格分割cmd_str存入的指令存入cmd中，再将nextSign后一个子字符串分割后存入cmd。如果没有这些标志，就直接把命令存入substring并分割。

1.3执行program 命令

从标准输入读命令并执行，其中的execvp函数将新程序载入当前进程，并替换了当前进程的代码和数据，新程序开始后，原程序就自动被清除了。shell程序是一个无止境的循环，不能够执行一次命令就终止。为了让原程序执行命令后还能够继续等待命令的完成。fork新建一个进程，让命令在这个新建的进程中调用execvp函数，这样shell函数就不会被清除了。

1.3.1基本架构

如下：

```
pid=fork();
if(0==pid)//execute command in child process {
if(0!=execvp(cmd[0],cmd))
//execvp 如果找不到对应的可执行文件就会返回非 0 的错误代码
printf("No such command!\n");
exit(EXIT_SUCCESS);
//结束当前进程 }
else//parent process
{ //等待子进程命令执行完，然后重新执行新的命令 }
```

1.3.2具体实现

在子进程fork成功后调用my_execute()函数。在my_execute分为管道，输入重定向，输出重定向和一般命令四种情况。

管道实现

首先创建一个管道，fork完之后在子进程将标准输入重定向到管道的读端，父进程将标准输出重定向到管道的写端。

在子进程中，关闭写端，关闭标准输入，这是执行dup(fd[0]);因为dup是按文件描述符从0开始找到最小的（也即第一个），这是第一个就是标准输入，所以就能将标准输入重定向到管道指向管道读端。此时fd[0]故close。这里递归调用my_execute 可实现多管道读输入。

在父进程中，关闭读端，关闭标准输出，将将标准输出指向fd[1]，不再需要fd[1],故关闭。接着执行命令 execvp第一个是命令第二个是参数表，如果返回不是0就没有这个命令，返回0就成功结束。

重定向

'>'从标准输出重定向到文件

把文件名复制到fileName,然后从标准输出重定位到fileName输出。这里的核心代码就是

```
freopen(fileName,"w",stdout);
```

FILE *freopen(const char *filename, const char *mode, FILE *stream) 把一个新的文件名 **filename** 与给定的打开的流 **stream** 关联，同时关闭流中的旧文件。其中“w“表示创建一个用于写入的空文件。如果文件名称与已存在的文件相同，则会删除已有文件的内容，文件被视为一个新的空文件。

接下来execvp与管道的父进程类似。

'<'从标准输入重定向到文件

把文件名复制到fileName,然后从标准输出重定位到fileName输出。这里的核心代码就是

```
freopen(fileName,"r",stdin;
```

其中“r“表示打开一个用于读取的文件。该文件必须存在。

接下来execvp与管道的父进程类似。

一般命令

如果没有重定向和管道，先保存当前localPtr，然后 execvp执行命令即可。

1.4执行内置命令

int my_cd(void);用chdir把目录转换到命令中要求的目录。并通过getcwd将当前工作目录的绝对路径复制到参数dir所指的内存空间中，记录当前的目录，一遍模拟系统的cd$前目录变换的情况。

int my_exit(void);释放共享内存并退出shell。

因为使用了共享内存的创建所以在exit中还需要加上：

```
shmdt(shm_buff);

shmctl(shm_id,IPC_RMID,0);
```

shmdt(addr)使进程中的shmaddr指针无效化，不可以使用，但是保留空间。
shmctl(shmid，IPC_RMID，0) 删除共享内存，彻底不可用，释放空间。

int my_history(void);在my_readCMD中已经将指令存入history数组中，所以这里只需要按照history 的n打印最近的n条历史命令即可。

history 定义一个全局的char数组和一个全局的int值。在my_readLine()函数中将每次输入的代码存入数组中,并在my_history函数中按顺序打印出来。

int my_top(void);

My_top实现较为复杂。

print_memory根据老师给的源码，到meminfo里读取输入并打印即可。

CPU_states 在print_procs中实现。

读输入

首先从/proc/kinfo读取nr_procs 和nr_tasks计算进程和任务总数nr_total。

通过get_procs调用parse_dir再调用parse_file读每个pid的状态。
get_procs()函数，负责给每个进程分配一个结构体数组将需要的信息对应地存到结构体各自的数组中。此外，get_procs()函数记录当前进程，赋值给prev_proc，然后通过parse_dir()函数获取到/proc下的所有进程的pid,再通过parse_file()函数获取每一个进程信息，即读取/proc/pid/psinfo文件。

在parse_file中nr_total是261,小于slot。slot是该进程结构体在数组中的位置，需要修改源码slot+=1/2均可，保证每次slot在数组中的位置不重复。
此外，在parse_file中连续读CPUTIMENAMES次cycles_hi,cycle_lo，因为u64_t 是64位，而 high和low32位 ，用make64拼接成64位 high+low，放在p_cpucycles[]数组中。

计算滴答并输出

滴答需要实时更新，所以需要通过当前进程来和该进程一起计算。在u64_t cputicks(**struct** proc *p1, **struct** proc *p2, **int** timemode) 计算每个进程proc的滴答。通过proc和当前进程prev_proc做比较，如果endpoint相等，则在循环中分别计算。

在print_procs创建了一个tp结构体的数组tick_procs。利用nr_total的for循环对所有的进程和任务计算ticks。

计算idle代码

```
if(p-5 == 317) {

     idleticks = uticks;

       continue;

  }
```

因为p-5一直不为317（由前面算出nr_total为261可知）故循环一直continue，故这样得出的idleticks恒为0。把kernelticks，userticks，systemticks相加就可以得到CPU的使用百分比。

2 设计思想

2.1基本架构

shell 的工作流程是这样的：

1、打印命令提示符；

2、读取并分析命令；

3、执行命令；

4、执行完命令后，重复 1-3；

基本实现如main函数所示，先对在my_init对指针初始化，然后写一个while循环。读入输入的命令并分析。判断是内置命令还是program命令。

如果是内置命令，成功执行后清理进程修改的所有数组。

如果是program命令，fork一个新进程执行program命令，（这里fork的原因是execvp函数将新程序载入当前进程，并替换了当前进程的代码和数据，新程序开始后，原程序就自动被清除了。这样while循环就失效了），然后判断是否为后台进程。

如果是后台进程，将标准输出重定向到/dev/null 这样做的目的是为了防止屏蔽键盘和控制台，父进程中语句：signal(SIGCHLD,SIG_IGN);表示父进程忽略SIGCHLD信号，该信号是子进程退出的时候向父进程发送的。

如果不是后台进程，在子进程中调用my_execute();用waitpid将父进程挂起直到子进程执行完再执行。

waitpid(pid,NULL,0);此处父进程pid>0，只等待进程ID等于pid的子进程，不管其它已经有多少子进程运行结束退出了，只要指定的子进程还没有结束，waitpid就会一直等下去。NULL, 表示 waitpid() 忽略子进程返回状态。

最后等一次读入的所有命令执行完再清理命令改变的所有数组。

```
int main()
{
    char line[MAX_LINE];
    my_init();
    while(1)
    {
        //打印当前命令
        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
        if(0==my_builtinCmd())
        {
            my_clearCmd(cmd_var);
            continue;
        }
```

```
        //fork一个新进程执行program命令
        else
        {
            pid = fork();
            if(background==1)
            {
                if (pid==0)
                {//标准输出重定向到/dev/null
                    freopen("/dev/null","w",stdout);
                    my_execute();

                }//父进程ignoreSIGCHLD
                signal(SIGCHLD,SIG_IGN);

            }
            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保存命令的结构体数组的设计

```
typedef struct CMD_STRUCT
{
    char *cmd[CMD_LENG]; //数组元素为字符指针
    char cmdStr[CMD_LENG*PARA_MAX];
    char nextSign; // '|' or'>' or '<'
}cmdStruct;
typedef struct CMD_ALL
{
    cmdStruct cmd_all[ALL_SIZE];//定义数组包含ALL_SIZE个
cmdstruct结构体
    int cmdPtr;
}cmd_all;
```

如上为一个指令结构体。char * cmd为一个元素为字符指针的数组，每个指针指向命令的首地址。cmdStr存my_substring得到的子字符串。nextSign存 '|' or'>' or '<'。

cmd_all数组包含所有个cmdstruct结构体。cmdPtr用以指示对应的是cmd_all的第几个命令。

举例,如对ls -l -a | grep a

Cmd 中指向的是cmd[0]为ls、cmd[1]为-l、cmd[2]为-a这样一个一个命令，是通过splitStr根据空格拆解开的。

nextSign存'|'

而由于管道符号的分割，前后算两个大的字符串。存入两个cmdStr,分别对应cmd_var->cmd_all[0].cmdStr和cmd_var->cmd_all[1].cmdStr。

这里的ls对应cmd_var->cmd_all[0].cmd[0]，grep对应cmd_var->cmd_all[1].cmd[0].

三、 使用环境

Xcode

VMware fusion 虚拟机

本地终端terminal

四、 实验过程

代码运行过程中的部分错误处理

4.1段错误

一开始，本地编译succeeded.但是在终端运行shell的时候会出现段错误。

查询后，分析原因是因为程序中的指针有几个没有分配内存导致的。而本地的Mac端对内存的管理非常严格。故修正程序如下：

添加代码

```
cmd_var->cmdPtr=0;
```

此外还有段错误是由空指针引起的。在实现mytop 的CPU使用情况时出现了空指针的段错误报错。查明原因是在parse_file读如proc下的文件时的读取出了错误。

4.2 Thread 1: EXC_BAD_ACCESS (code=1, address=0x40)

此运行错误在测试中出现多次，此处描述第一次的处理策略。其他几次问题原因类似，不做赘述。代码编译正常后，本地运行出现Thread 1: EXC_BAD_ACCESS (code=1, address=0x40)的情况。

> “Thread 1: EXC_BAD_ACCESS” 的出现一般都是因为内存泄漏的问题导致。
> 当需要给一个对象发送消息，但是该对象已经被提前释放时，就可能会出现这种情况，但是编译器往往会crash到其他位置

```
cmd_var->cmdPtr=0;
```

定义了一个数组来存放指令。

```
cmd_all * cmd_var ;
```

```
typedef struct CMD_ALL
{
    cmdStruct cmd_all[POOL_SZ];
    int cmdPtr;
}cmd_all;
```

ps:此处头文件用typedef struct 便于定义一个结构类型，并给这个新的结构起名字。

这里报错的原因是cmd_var并没有分配一段内存。采用共享内存分配。因为这个shell要实现管道这样的进程间通信方式。

> 对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据： 一次从输入文件到共享内存区，另一次从共享内存区到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建 立共享内存区域。而是保持共享区域，直到通信完毕为止。

共享内存实现如下：

```
key = ftok(".",0);

  shm_id = shmget(key,SHM_MEM_SIZE,IPC_CREAT|0666);

   shm_buff = (char *)shmat(shm_id,NULL,0);

  cmd_var = (cmd_all *)shm_buff;
```

key_t ftok(const char * fname, int id)，"."本处设置为当前目录。id为子序列号.

int shmget(key_t key, size_t size, int shmflg);shmget用来开辟/指向一块共享内存

shmflg是权限标志，它的作用与open函数的mode参数一样，与IPC_CREAT做或操作,可以在key标识的共享内存不存在时，创建它。0666表示：表示文件拥有者有读写权限，但没有执行权限，文件拥有者同组用户有读写权限，但没有执行权限，其它用户有读写权限，但没有执行权限。

shmat()函数的作用就是用来启动对该共享内存的访问，并把共享内存连接到当前进程的地址空间。

这样就完成共享内存的创建和启动的初始工作。断开映射关系和删除内存片段在my_exit()中完成。

4.3 Process finished with exit code 11

在my_history()测试时出现 Process finished with exit code 11，debug调试发现原因是history数组中只存入了一行读入的命令。故当my_history()中的printf想打印出第二行及以后的命令就无法打印。my_readline()每次读入一行就break，而表示行号的变量为局部变量。这会导致将第二次读入的命令存入数组时，想存入数组第二行。而实际上存的是第一行，与原来存入的值发生冲突，出现乱码。将代码中表示行号的变量定义为全局变量即可。

4.4 Vim: Warning: Output is not to a terminal

在进行vi result.txt 出现Vim: Warning: Output is not to a terminal

打开result.txt文件发现里面都是乱码，最后一行显示Error reading input, exiting...，疑似输入重定向中出现了问题。

发现应该每次读完执行完指令后要将所有记录该指令的变量数组还原，变回最开始的样子。

```
int my_clearCmd(cmd_all * cmd_var)
{
    int i,j;

    cmd_var->cmdPtr=0;
    for(i=0;i<ALL_SIZE;i++)
```

```
    {
        for(j=0;j<CMD_LENG;j++)
        {
            cmd_var->cmd_all[i].cmd[j]=NULL;
            cmd_var->cmd_all[i].cmdStr[0]='\0';
        }
        cmd_var->cmd_all[i].nextSign=0;
    }

    return 0;
}
```

五、 总结

在做类似这样几百行的shell之初，要先构建基本架构思路，再逐一实现。

在这次实验中明白了一个基本shell的实现，熟悉了C语言环境高级编程。本次实验学到了管道和重定向的操作，结构体指针的运用，以及空指针出现的段错误，如何避免野指针（指针变量一定要初始化，如果free或delete指针后一定要置NULL，并且要尽量避免指针操作超出变量的作用范围）以及局部变量和全局变量运用的搭配，在history数组打印时有比较突出的体现。初步体会到了模块化的重要意义。

学习了出现错误除了分析代码以外，通过printf打印，进行二分法搜索出错的那行的代码的步骤，在调试mytop的代码中有比较明显的体现。

分析了minix top实现的源码，了解了github双击变量会自动出现define,了解了minix /proc下的基本架构，以及CPUstates的计算方法和实时更新的实现。

熟悉了虚拟机的使用和Cyberduck传输文件的方法。