**《操作系统》课第09次实验报告**

|  |  |
| --- | --- |
| 学院: | 软件学院 |
| 姓名: | 张怡桢 |
| 学号: | 2013747 |
| 邮箱: | 2662765987@qq.com |
| 时间: | 11/17/2022 |

1. **开篇感言**

“你长大后想成为什么人？”

“什么意思？长大后我就不能成为我自己了吗？”

-- 《阿甘正传》

2. **实验题目**

在 Linux 平台上，采用 C 语言编写一个 Mini Shell 命令解释环境（即类似 Bash Shell 环境）。该环境可以循环接受用户（从标准输入中）输入的（外部和内部） 命令以及若干参数，然后能对上述命令进行解析和执行，最后将用户输入的命令 的执行结果显示在标准输出上。即：

bash-2.03$ ps\_03

\*\*\*\*\*\*\*\*\*\*\*\*\*\*welcome to mini shell\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MINI SHELL#pwd /home/unixmng/oscourses/ps\_prog

MINI SHELL#exit

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* mini shell exit\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

bash-2.03$

3. **实验要求**

1. 支持用户输入一行命令及其多个参数，并解析执行，并输出结果；
2. 支持 cd 命令，若无参数则回到当前用户的登录目录（见下面提示）；
3. 支持以“当前路径”和“用户名”为提示符； 支持对命令行中空格的自动忽略处理；
4. 支持对命令行中 tab 键的自动忽略处理；
5. 支持一行中以“；”（为标志）分隔的多个命令及多个参数的顺序执行， 即如下：
6. MINI SHELL#pwd; ls –l;date
7. 说明：上述三个命令须在本 Mini Shell 下依次顺序执行，最后由 Mini Shell 再次循环接受用户的新命令。

4. **原理方法**

**4.1 相关函数**

1. 相关头文件：

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <pwd.h>

1. 相关参考函数：
2. fork 、execvp、wait ：创建进程
3. strcmp、strcpy、strncpy ：字符串相关操作
4. fopen、fclose、fscanf、fprintf、fgets、fputs：文件 stdio 操作
5. sprintf ：任意类型转换为字符串
6. atoi ：字符串转换为整数类型 int
7. getlogin ：获取当前用户名
8. getcwd ：获取当前路径
9. chdir ：改变当前路径
10. getenv ：获取环境变量
11. 获取当前用户的登录目录

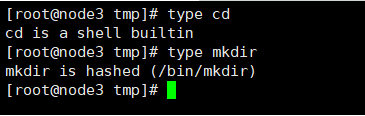
|  |
| --- |
| C++ //////////////////////get user default dir int getuserdir(char \*aoUserDir) {  char \*LoginId;  struct passwd \*pwdinfo;  if (aoUserDir == NULL)  return -9;  if ((LoginId = getlogin()) == NULL)  {  perror("getlogin");   aoUserDir[0] = '\0';   return -8;  }  if ((pwdinfo = getpwnam(LoginId)) == NULL)  {   perror("getpwnam");   return -7;  }  strcpy(aoUserDir, pwdinfo->pw\_dir); } |

4.2 [**shell内置命令和外部命令的区别**](https://www.cnblogs.com/7q4w1e/p/9555680.html)

　　内部命令实际上是shell程序的一部分，其中包含的是一些比较简单的linux系统命令，这些命令由shell程序识别并在shell程序内部完成运行，通常在linux系统加载运行时shell就被加载并驻留在系统内存中。内部命令是写在bashy源码里面的，其执行速度比外部命令快，因为解析内部命令shell不需要创建子进程。比如：exit，history，cd，echo等。

　　外部命令是linux系统中的实用程序部分，因为实用程序的功能通常都比较强大，所以其包含的程序量也会很大，在系统加载时并不随系统一起被加载到内存中，而是在需要时才将其调用内存。通常外部命令的实体并不包含在shell中，但是其命令执行过程是由shell程序控制的。shell程序管理外部命令执行的路径查找、加载存放，并控制命令的执行。外部命令是在bash之外额外安装的，通常放在/bin，/usr/bin，/sbin，/usr/sbin......等等。可通过“echo $PATH”命令查看外部命令的存储路径，比如：ls、vi等。

　　用type命令可以分辨内部命令与外部命令：



　　内部命令和外部命令最大的区别之处就是性能。内部命令由于构建在shell中而不必创建多余的进程，要比外部命令执行快得多。因此和执行更大的脚本道理一样，执行包含很多外部命令的脚本会损害脚本的性能。

5. **代码分析**

5.1 **全局定义**

全局变量：

current\_dir：当前所在的系统路径

user\_dir：当前所在的系统路径

cmdline：从终端读入的字符串

separator：终端读入多条命令的规定分隔符“；”

commands：解析cmdline后得到的string数组

command：commands的其中一个元素

child\_commands:解析单条命令command的参数

builtins[]:内部命令的结构体数组，用于处理内部命令

全局函数：

void init()：初始化函数

int execute\_command()：执行命令函数

void read\_command()：读取命令函数

void parse\_command()：解析命令函数

int builtin(void）：内部命令

void do\_cd()：内部命令的cd函数

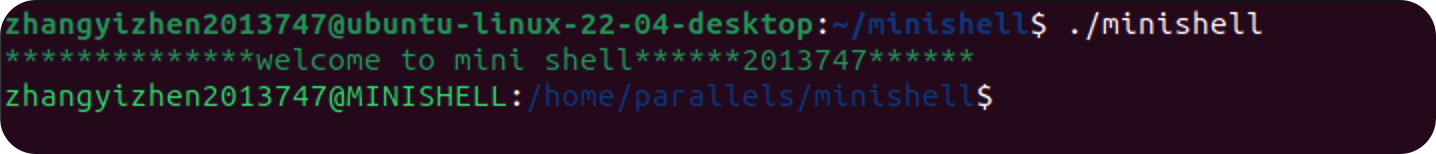
void do\_ls()：内部命令的ls函数

|  |
| --- |
| C++ //全局变量定义  int re\_flag = 0; char current\_dir[100]; char user\_dir[100];  string cmdline; //command input string separator=";"; string command;   vector<string> commands; // 所有命令 vector<string> child\_commands; //仅一个命令  //结构体定义 typedef void (CMD\_HANDLER)(void); typedef struct builtin\_cmd {  char\* name;  CMD\_HANDLER \*handler; } BUILTIN\_CMD;  //全局函数声明 void init(); int execute\_command(); void read\_command(); void parse\_command(); int builtin(void); void do\_cd(); void do\_ls();  //内部命令解析 BUILTIN\_CMD builtins[] = {  {"cd", do\_cd},  {"ls", do\_ls},  {NULL, NULL}  }; |

5.2 **获取当前文件的路径以及系统用户名**

|  |
| --- |
| C++ strcpy(current\_dir, getcwd(NULL, 0)); strcpy(user\_dir, getcwd(NULL, 0)); printf("\033[92m%s@MINISHELL\033[0m:\033[34m%s\033[0m$", getlogin(), current\_dir); |

得到进入minishell后可以获得当前路径以及文件名的提示



5.3 **read\_command()**

读入终端的命令,对终端的命令使用“；”进行划分，对多条命令进行划分；

|  |
| --- |
| C++ void read\_command() {  //处理cmdline  getline(cin, cmdline); // input string with ' '  typedef string::size\_type string\_size;  string\_size i = 0;  while (i != cmdline.size())  {  int flag = 0;  while (i != cmdline.size() && flag == 0)  {  flag = 1;  for (string\_size x = 0; x < separator.size(); ++x)  if (cmdline[i] == separator[x])  {  ++i;  flag = 0;  break;  }  }  flag = 0;  string\_size j = i;  while (j != cmdline.size() && flag == 0)  {  for (string\_size x = 0; x < separator.size(); ++x)  if (cmdline[j] == separator[x])  {  flag = 1;  break;  }  if (flag == 0)  ++j;  }  if (i != j)  {  commands.push\_back(cmdline.substr(i, j - i));  i = j;  }  } } |

5.4 **parse\_command()**

处理命令，对单条命令进行参数解析

|  |
| --- |
| C++ void parse\_command() {  //处理cmds  for (int i = 0; i < commands.size(); i++)  {  child\_commands.clear();  command.clear();  stringstream input2(commands[i]); // string stream initialize 不按照空格划分  while (input2 >> command)  {  child\_commands.push\_back(command);  }  if (command == "exit")  {  printf("\033[32m\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* mini shell exit\*\*\*\*\*\*\*2013747\*\*\*\*\*\*\n\033[0m");  exit(0);  }  if (child\_commands.size())  {  execute\_command();  }  } } |

5.5 **execute\_command**

这一步分成shell内部命令与shell的外部命令

5.5.1 **内部命令**

实现cd

|  |
| --- |
| C++ void do\_cd(){   // 当前系统目录  // char target\_path[100];  // getcwd(target\_path, 100);   if (child\_commands.size() == 1)  {  strcpy(current\_dir, user\_dir);  }  else  {  const char \*rest = child\_commands[1].c\_str();  if (child\_commands[1] == "/")  {  opendir(rest);  strcpy(current\_dir, rest);  chdir(rest);   }  else if (child\_commands[1] == "..")  {  char \*parent\_dir = dirname(current\_dir);  strcpy(current\_dir, parent\_dir);  chdir(parent\_dir);  }  else if (child\_commands[1] == "~")  {  strcpy(current\_dir, user\_dir);  chdir(user\_dir);  }  else  {  char target\_path[1024];   if (strcmp(current\_dir, "/") == 0)  {  snprintf(target\_path, 1024, "%s%s", current\_dir, rest);  }  else  {  snprintf(target\_path, 1024, "%s/%s", current\_dir, rest);  }  if (opendir(target\_path) == NULL)  {   printf("\033[31m没有那个文件或目录.\n\033[0m");  }  strcpy(current\_dir, target\_path);  chdir(current\_dir);  }  cout << current\_dir << endl;  }  } |

实现ls以及ls的文件重定向

|  |
| --- |
| C++ void do\_ls(){    pid\_t pid;  pid= fork();  int status;  int count = 0;  const char \*rest = child\_commands[0].c\_str();  if (pid == 0)  {  for (int i = 0; i < child\_commands.size(); i++)  {   if (child\_commands[i] == ">")  {  re\_flag = 1;  count = i;  }  if (child\_commands[i] == ">>")  {  re\_flag = 2;  count = i;  }  }  if (re\_flag != 0)  {  char \*\*cmd\_temp = new char \*[count];  for (int i = 0; i < count; i++)  {  cmd\_temp[i] = new char[500];  memset(cmd\_temp[i], 0, sizeof(\*cmd\_temp[i]));  }  for (int i = 0; i < count; i++)  {  strcpy(cmd\_temp[i], child\_commands[i].c\_str());  }  cmd\_temp[count] = current\_dir;  cmd\_temp[count + 1] = NULL;  // 标准输出重定向，将原本要写入标准输出 1 的数据写入新文件(fd)中  int fd = 1;  if (re\_flag == 1)  fd = open(child\_commands[count + 1].c\_str(), O\_CREAT | O\_WRONLY | O\_TRUNC, 0664);  else if (re\_flag == 2)  fd = open(child\_commands[count + 1].c\_str(), O\_CREAT | O\_WRONLY | O\_APPEND, 0664);  dup2(fd, 1);  if (execvp(rest, cmd\_temp) < 0)  {  printf("\033[31m%s:command not found.\n\033[0m", child\_commands[0].c\_str());  }  }  else  {  char \*\*cmd\_temp = new char \*[child\_commands.size() + 1];  for (int i = 0; i < child\_commands.size(); i++)  {  cmd\_temp[i] = new char[500];  memset(cmd\_temp[i], 0, sizeof(\*cmd\_temp[i]));  }  for (int i = 0; i < child\_commands.size(); i++)  {  strcpy(cmd\_temp[i], child\_commands[i].c\_str());  }  cmd\_temp[child\_commands.size()] = current\_dir;  cmd\_temp[child\_commands.size() + 1] = NULL;  if (execvp(rest, cmd\_temp) < 0)  {  printf("\033[31m%s:command not found.\n\033[0m", child\_commands[0].c\_str());  }  }  }   else if (pid > 0)  {  do  {  waitpid(pid, &status, WUNTRACED);  } while (!WIFEXITED(status) && !WIFSIGNALED(status));  }   } |

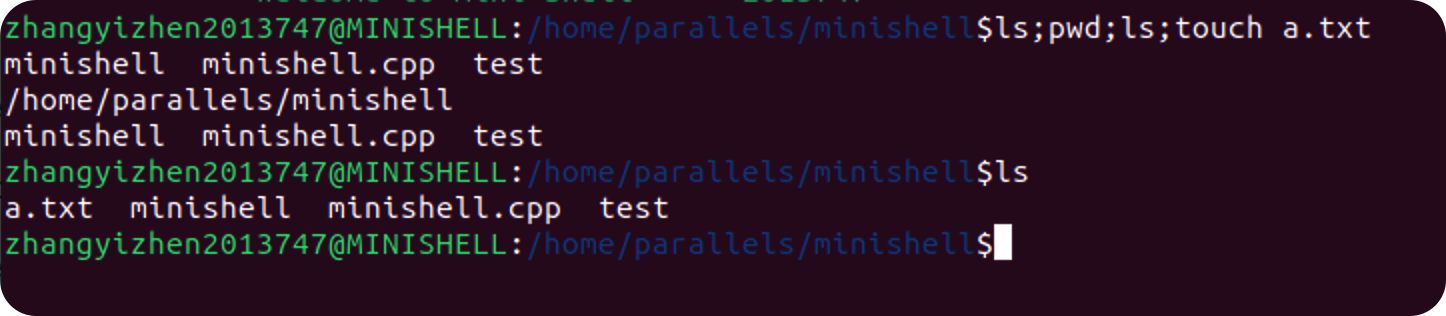
**5.5.2 外部命令**

创建子进程进行外部命令的执行

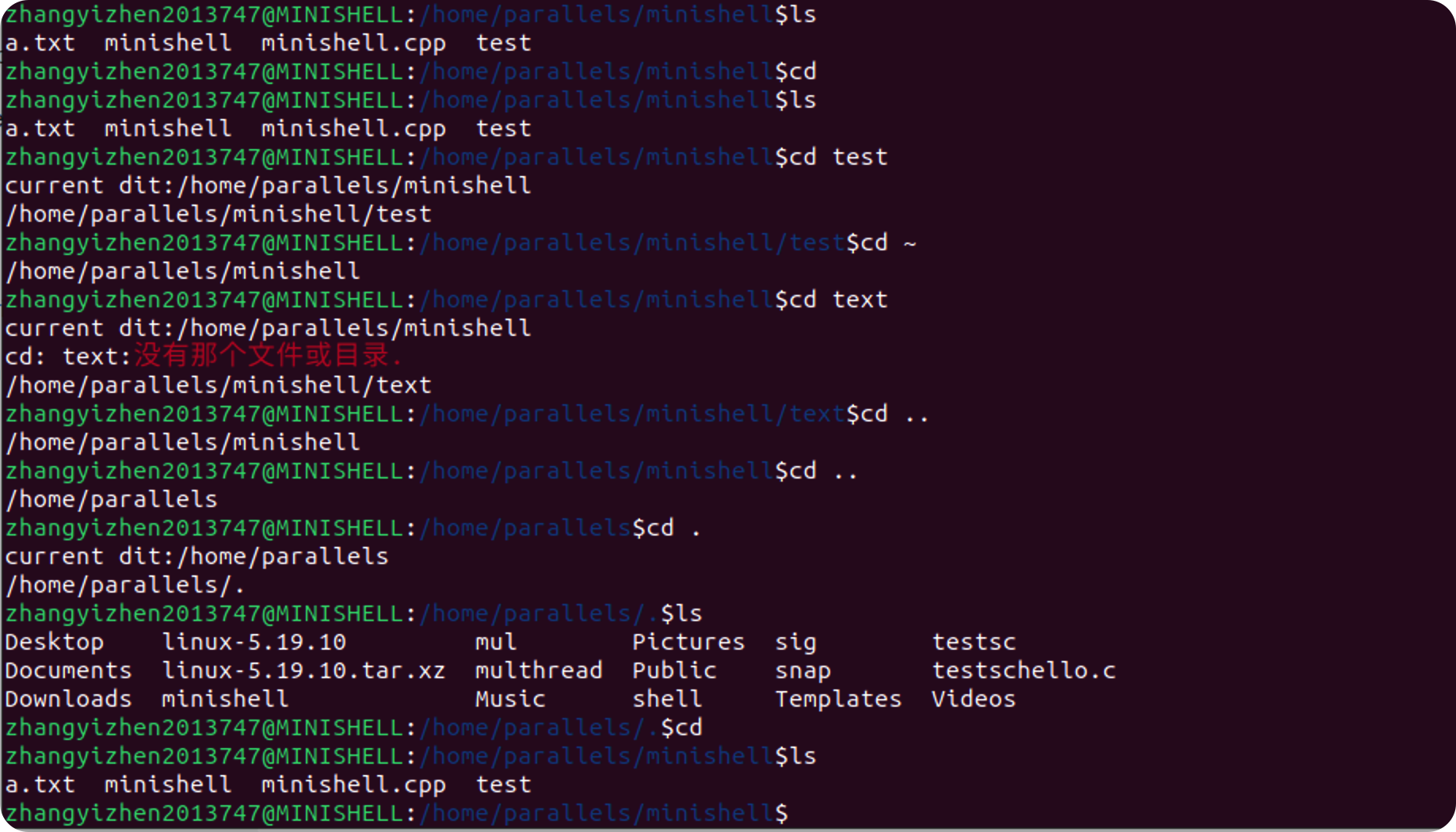
|  |
| --- |
| C++ pid\_t pid; pid = fork(); int status; const char \*rest = child\_commands[0].c\_str(); if (pid == 0) {  //子线程  char \*\*cmd\_temp = new char \*[child\_commands.size()];  for (int i = 0; i < child\_commands.size(); i++)  {  cmd\_temp[i] = new char[500];  memset(cmd\_temp[i], 0, sizeof(\*cmd\_temp[i]));  }  for (int i = 0; i < child\_commands.size(); i++)  {  strcpy(cmd\_temp[i], child\_commands[i].c\_str());  }  cmd\_temp[child\_commands.size()] = NULL;  if (execvp(rest, cmd\_temp) < 0)  {  printf("\033[31m%s:command not found.\n\033[0m", child\_commands[0].c\_str());  }  } else if (pid > 0) {  do  {  waitpid(pid, &status, WUNTRACED);  } while (!WIFEXITED(status) && !WIFSIGNALED(status)); } |

6. **实现功能**

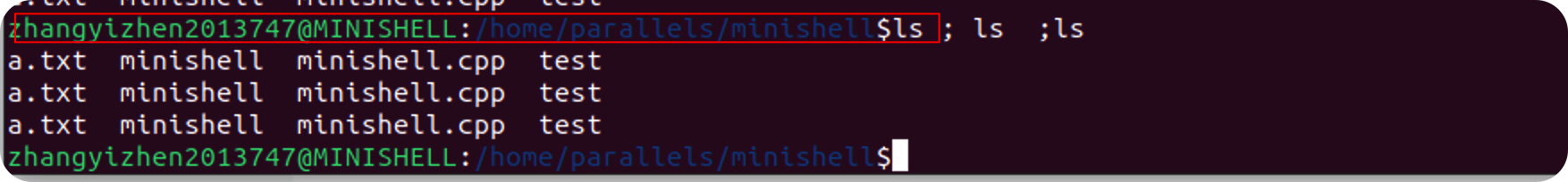
1. 支持用户输入一行命令及其多个参数，并解析执行，并输出结果；



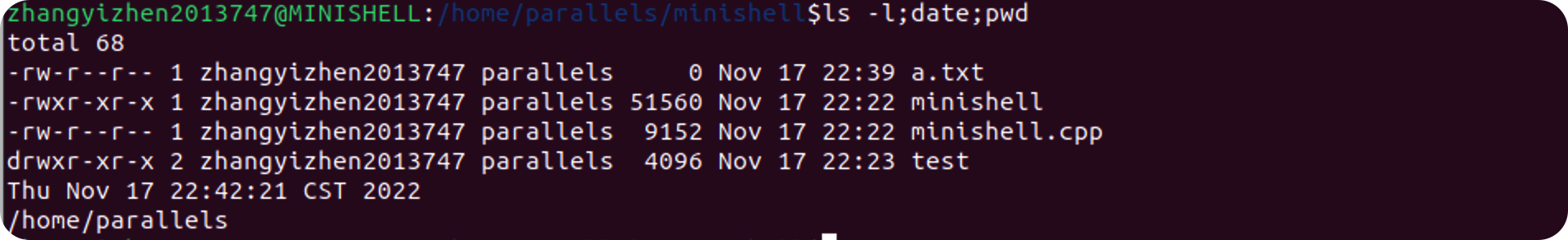
1. 支持 cd 命令，若无参数则回到当前用户的登录目录（见下面提示）；



1. 支持以“当前路径”和“用户名”为提示符； 支持对命令行中空格的自动忽略处理；
2. 支持对命令行中 tab 键的自动忽略处理；



1. 支持一行中以“；”（为标志）分隔的多个命令及多个参数的顺序执行， 即如下：
2. MINI SHELL#pwd; ls –l;date

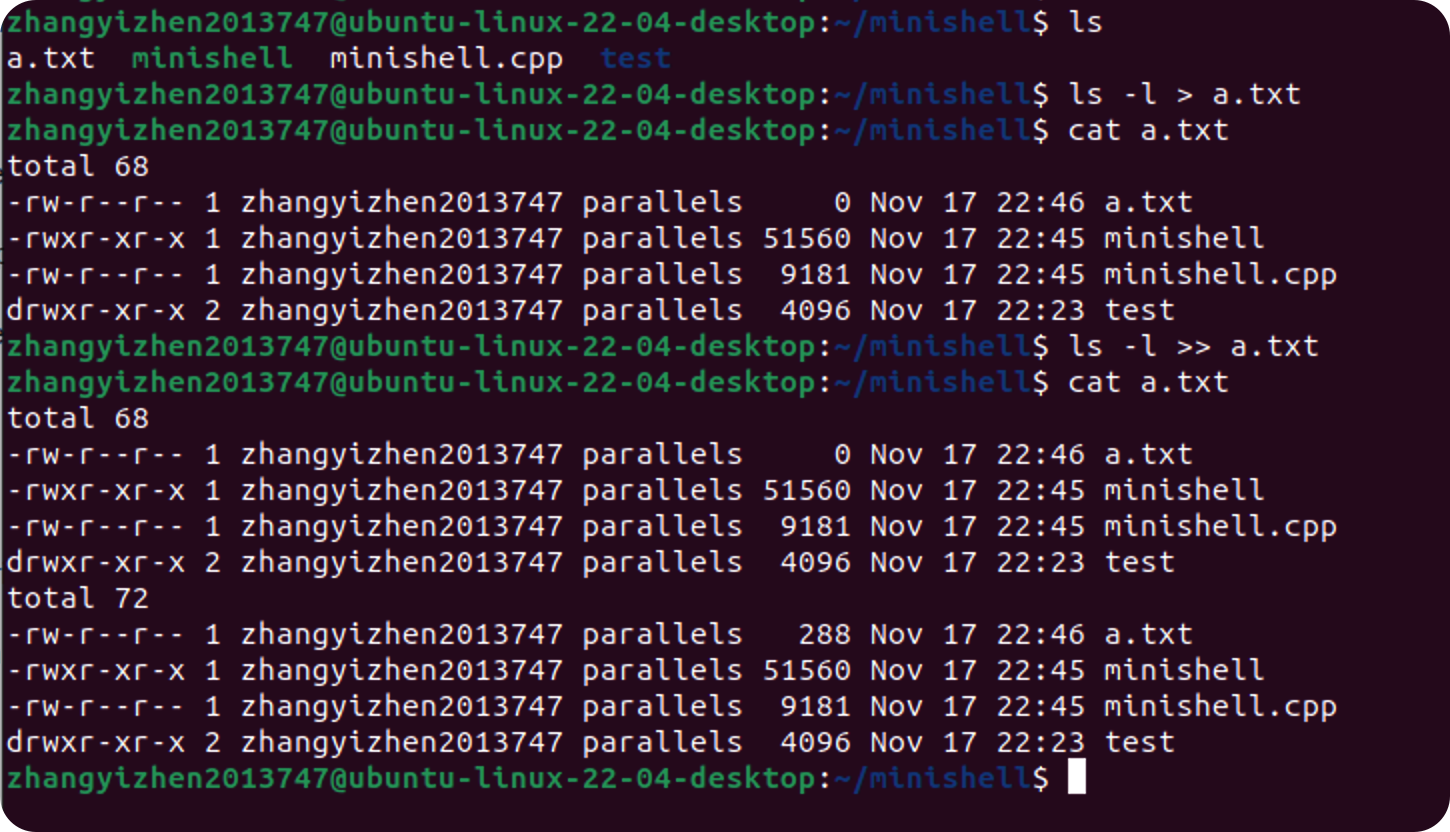


1. 说明：上述三个命令须在本 Mini Shell 下依次顺序执行，最后由 Mini Shell 再次循环接受用户的新命令。

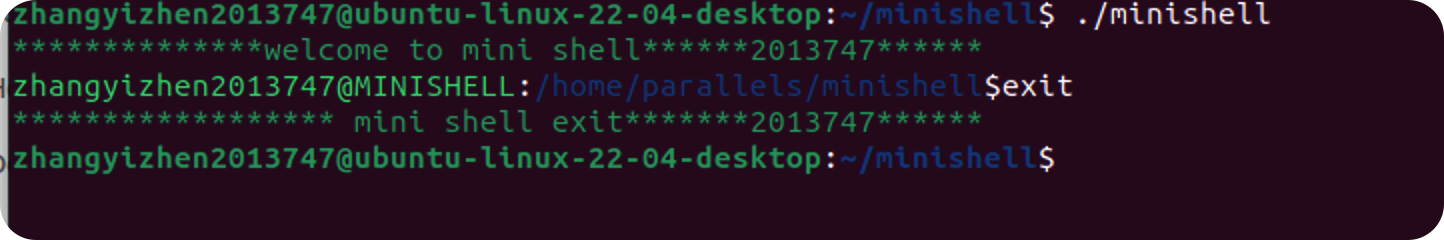
**附加功能：**

ls -l 文件重定向

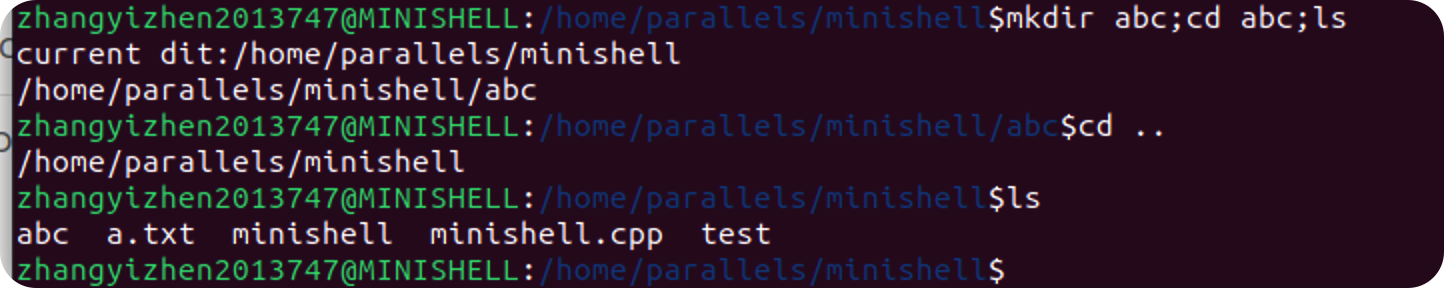
cat查看文件



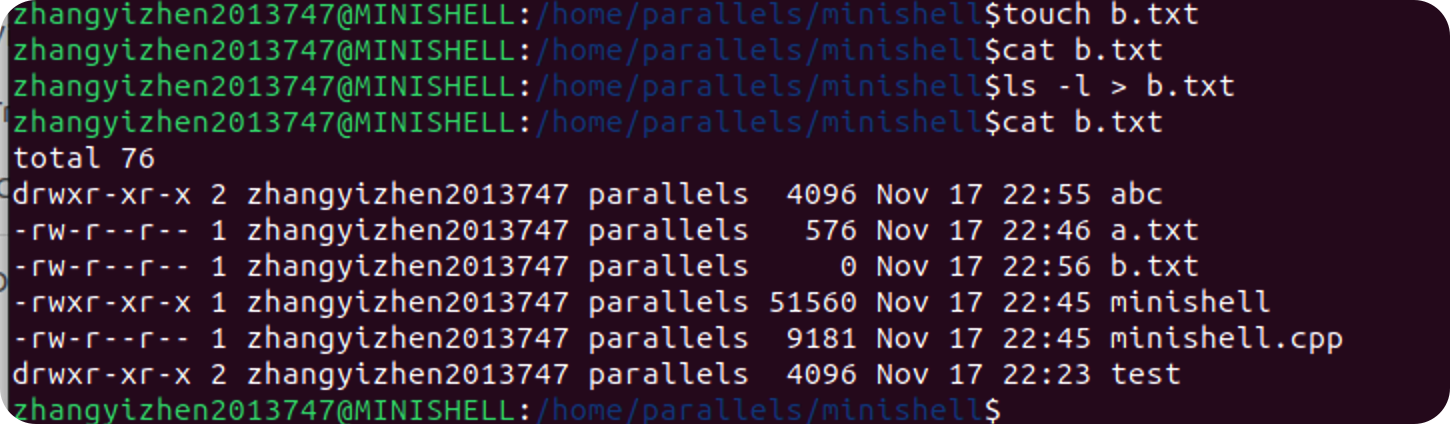
exit()



Mkdir 文件夹



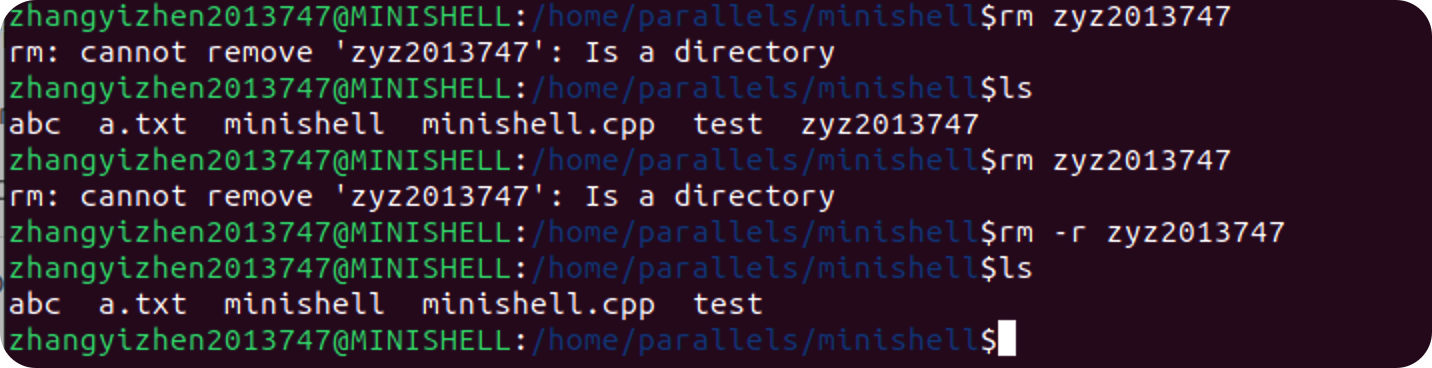
Touch 文件



pwd



rm



7. **参考资料**

[老师的github实验文档](https://github.com/albertleecn/osplab)

8. **附件**

📎

|  |
| --- |
| C++ #include <iostream> #include <cstdio> #include <string.h> #include <unistd.h> #include <libgen.h> #include <sys/wait.h> #include <vector> #include <sstream> #include <dirent.h> #include <fcntl.h> using namespace std;  //全局常量定义 #define SUCCESS 0  //全局变量定义  int re\_flag = 0; char current\_dir[100]; char user\_dir[100];  string cmdline; //command input string separator=";"; string command;   vector<string> commands; // 所有命令 vector<string> child\_commands; //仅一个命令  typedef void (CMD\_HANDLER)(void); typedef struct builtin\_cmd {  char\* name;  CMD\_HANDLER \*handler; } BUILTIN\_CMD;  //全局函数声明 void init(); int execute\_command(); void read\_command(); void parse\_command(); int builtin(void); void do\_cd(); void do\_ls();  //内部命令解析 BUILTIN\_CMD builtins[] = {  {(char\*)"cd", do\_cd},  {(char\*)"ls", do\_ls},  {NULL, NULL}  };    int main() {   printf("\033[32m\*\*\*\*\*\*\*\*\*\*\*\*\*\*welcome to mini shell\*\*\*\*\*\*2013747\*\*\*\*\*\* \n\033[0m");  strcpy(current\_dir, getcwd(NULL, 0));  strcpy(user\_dir, getcwd(NULL, 0)); // printf("PATH : %s\n", getenv("PATH"));  while (1)  {  printf("\033[92m%s@MINISHELL\033[0m:\033[34m%s\033[0m$", getlogin(), current\_dir);  init();  read\_command();  parse\_command();   }  return 0; }  void init(){  for (int i = 0; i < child\_commands.size(); i++)  {  child\_commands[i].clear();  }  for (int i = 0; i < commands.size(); i++)  {  commands[i].clear();  }  cmdline.clear();  command.clear();  child\_commands.clear();  commands.clear(); }  void read\_command() {  //处理cmdline  getline(cin, cmdline); // input string with ' '  typedef string::size\_type string\_size;  string\_size i = 0;  while (i != cmdline.size())  {  int flag = 0;  while (i != cmdline.size() && flag == 0)  {  flag = 1;  for (string\_size x = 0; x < separator.size(); ++x)  if (cmdline[i] == separator[x])  {  ++i;  flag = 0;  break;  }  }  flag = 0;  string\_size j = i;  while (j != cmdline.size() && flag == 0)  {  for (string\_size x = 0; x < separator.size(); ++x)  if (cmdline[j] == separator[x])  {  flag = 1;  break;  }  if (flag == 0)  ++j;  }  if (i != j)  {  commands.push\_back(cmdline.substr(i, j - i));  i = j;  }  } }  void parse\_command() {  //处理cmds  for (int i = 0; i < commands.size(); i++)  {  child\_commands.clear();  command.clear();  du stringstream input2(commands[i]); // string stream initialize 不按照空格划分  while (input2 >> command)  {  child\_commands.push\_back(command);  }  if (command == "exit")  {  printf("\033[32m\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* mini shell exit\*\*\*\*\*\*\*2013747\*\*\*\*\*\*\n\033[0m");  exit(0);  }  if (child\_commands.size())  {  execute\_command();  }  } }  //内部命令解析 //执行返回1，没有表示0 int builtin(void){  int i = 0;  int found = 0;  while(builtins[i].name != NULL){  if(builtins[i].name == child\_commands[0]) {  builtins[i].handler();  found=1;  break;  }  i++;  }  return found; }  int execute\_command() {   if(builtin())  return SUCCESS;  else  {  pid\_t pid;  pid = fork();  int status;  const char \*rest = child\_commands[0].c\_str();  if (pid == 0)  {  //子线程  char \*\*cmd\_temp = new char \*[child\_commands.size()];  for (int i = 0; i < child\_commands.size(); i++)  {  cmd\_temp[i] = new char[500];  memset(cmd\_temp[i], 0, sizeof(\*cmd\_temp[i]));  }  for (int i = 0; i < child\_commands.size(); i++)  {  strcpy(cmd\_temp[i], child\_commands[i].c\_str());  }  cmd\_temp[child\_commands.size()] = NULL;  if (execvp(rest, cmd\_temp) < 0)  {  printf("\033[31m%s:command not found.\n\033[0m", child\_commands[0].c\_str());  }   }  else if (pid > 0)  {  do  {  waitpid(pid, &status, WUNTRACED);  } while (!WIFEXITED(status) && !WIFSIGNALED(status));  }  }  return SUCCESS; }  void do\_cd(){   // 当前系统目录  // char target\_path[100];  // getcwd(target\_path, 100);   if (child\_commands.size() == 1)  {  strcpy(current\_dir, user\_dir);  chdir(user\_dir);  }  else  {  const char \*rest = child\_commands[1].c\_str();  if (child\_commands[1] == "/")  {  opendir(rest);  strcpy(current\_dir, rest);  chdir(rest);   }  else if (child\_commands[1] == "..")  {  char \*parent\_dir = dirname(current\_dir);  strcpy(current\_dir, parent\_dir);  chdir(parent\_dir);  }  else if (child\_commands[1] == "~")  {  strcpy(current\_dir, user\_dir);  chdir(user\_dir);  }  else  {  char target\_path[1024];   if (strcmp(current\_dir, "/") == 0)  {  snprintf(target\_path, 1024, "%s%s", current\_dir, rest);  }  else  {  snprintf(target\_path, 1024, "%s/%s", current\_dir, rest);  }  if (opendir(target\_path) == NULL)  {   printf("\033[31m没有那个文件或目录.\n\033[0m");  }  strcpy(current\_dir, target\_path);  chdir(current\_dir);  }  cout << current\_dir << endl;  }  }  void do\_ls(){    pid\_t pid;  pid= fork();  int status;  int count = 0;  const char \*rest = child\_commands[0].c\_str();  if (pid == 0)  {  for (int i = 0; i < child\_commands.size(); i++)  {   if (child\_commands[i] == ">")  {  re\_flag = 1;  count = i;  }  if (child\_commands[i] == ">>")  {  re\_flag = 2;  count = i;  }  }  if (re\_flag != 0)  {  char \*\*cmd\_temp = new char \*[count];  for (int i = 0; i < count; i++)  {  cmd\_temp[i] = new char[500];  memset(cmd\_temp[i], 0, sizeof(\*cmd\_temp[i]));  }  for (int i = 0; i < count; i++)  {  strcpy(cmd\_temp[i], child\_commands[i].c\_str());  }  cmd\_temp[count] = current\_dir;  cmd\_temp[count + 1] = NULL;  // 标准输出重定向，将原本要写入标准输出 1 的数据写入新文件(fd)中  int fd = 1;  if (re\_flag == 1)  fd = open(child\_commands[count + 1].c\_str(), O\_CREAT | O\_WRONLY | O\_TRUNC, 0664);  else if (re\_flag == 2)  fd = open(child\_commands[count + 1].c\_str(), O\_CREAT | O\_WRONLY | O\_APPEND, 0664);  dup2(fd, 1);  if (execvp(rest, cmd\_temp) < 0)  {  printf("\033[31m%s:command not found.\n\033[0m", child\_commands[0].c\_str());  }  }  else  {  char \*\*cmd\_temp = new char \*[child\_commands.size() + 1];  for (int i = 0; i < child\_commands.size(); i++)  {  cmd\_temp[i] = new char[500];  memset(cmd\_temp[i], 0, sizeof(\*cmd\_temp[i]));  }  for (int i = 0; i < child\_commands.size(); i++)  {  strcpy(cmd\_temp[i], child\_commands[i].c\_str());  }  cmd\_temp[child\_commands.size()] = current\_dir;  cmd\_temp[child\_commands.size() + 1] = NULL;  if (execvp(rest, cmd\_temp) < 0)  {  printf("\033[31m%s:command not found.\n\033[0m", child\_commands[0].c\_str());  }  }  }   else if (pid > 0)  {  do  {  waitpid(pid, &status, WUNTRACED);  } while (!WIFEXITED(status) && !WIFSIGNALED(status));  }   } |