**实验二 – Shell的实现**

2022年8月22日

1. **实验内容**
2. 了解Shell的概念、基本原理、实现机制等内容；
3. 实现自己的Shell，能够执行简单命令以及具有管道和重定向功能；
4. **实验目的**

实现具有管道、重定向功能的shell，能够执行一些简单的基本命令，如进程执行、列目录等。通过实验，让学生了解Shell实现机制。

1. **设计思路和流程图**
2. 什么是Shell

在设计之前，我们需要首先了解什么是Shell，在计算机科学中，Shell俗称壳（用来区别于核），是指“为使用者提供操作界面”的[软件](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6/12053)（command interpreter，命令解析器）。它类似于[DOS](https://baike.baidu.com/item/DOS/32025)下的[COMMAND.COM](https://baike.baidu.com/item/COMMAND.COM/8063418)和后来的[cmd.exe](https://baike.baidu.com/item/cmd.exe/8192925)。它接收用户命令，然后调用相应的[应用程序](https://baike.baidu.com/item/%E5%BA%94%E7%94%A8%E7%A8%8B%E5%BA%8F/5985445)。

Shell又分为图形式和命令行式、交互式和非交互式等等。

Shell是一个应用程序，提供了界面，用户通过这个界面来访问操作系统内核的服务。它接收用户输入的命令，并对命令进行处理，处理完毕后将结果反馈给用户，Shell本身的功能其实是较弱的，它的功能比如文件操作、输入输出、进程管理都是要依赖于内核的，也就是说Shell为我们提供了一种更便捷的方式来使用内核服务。

**管道：**也就是我们在命令行中经常使用的‘|’符号，它的作用是将一个命令的输出作为另一个命令的输入，可以说是重定向的建议实现。比如说命令“rpm -qa | grep python”就是先列出系统已有的所有软件包，传递给下一个命令，然后下一个命令在这些值中搜索含有“python”的项。

**重定向：**也就是在命令行中使用的‘>’‘>>’‘<’‘<<’符号。我们有些时候想要将命令的输出保存起来，而不仅仅是让它输出到显示器上（终端上）。将命令的输出保存再文件中后方便分析输出的内容。[重定向](https://so.csdn.net/so/search?q=%E9%87%8D%E5%AE%9A%E5%90%91&spm=1001.2101.3001.7020)也可以是输出也可以是输入。

了解了Shell的基本原理、实现机制后，下面开始设计所要实现的Shell。

1. 基本的输入输出功能

**设计思路：**

具有交互功能的Shell首先应有基本的输入输出功能，一般来说Shell会先显示给用户当前所在路径，然后提示输入。这里采用的格式为：

SHELL /...*“当前路径”*.../

> *“用户在此处输入命令”*

然后需要能够先读取用户输入的单行命令，使用getline()来读取，下一步再对命令进行解析等。

**具体实现：**

进入/home/seu/EX/EX\_2/路径输入touch shell.c创建文件：





**功能测试：**

测试输入输出功能：



能够正常读取用户输入的命令。

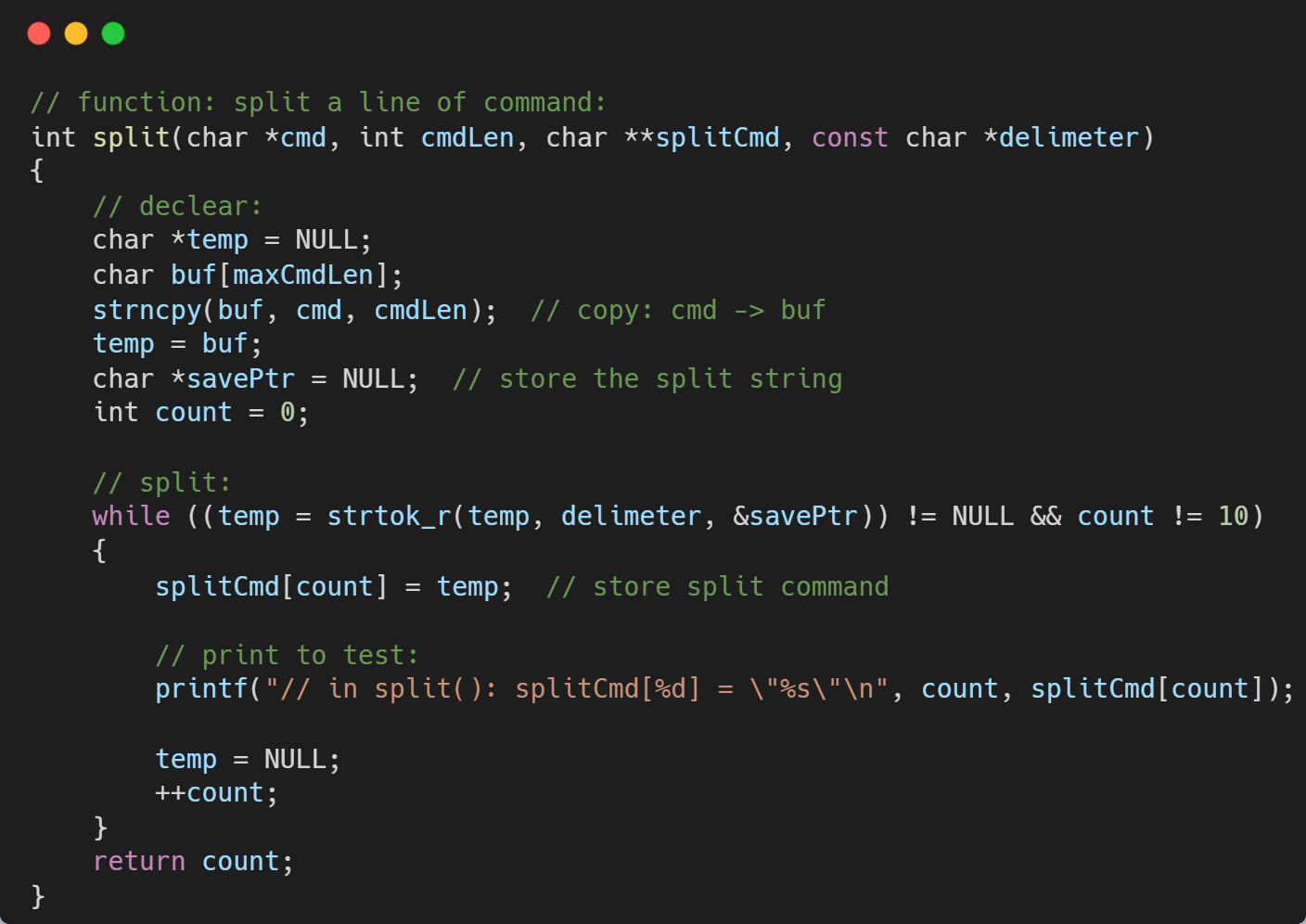
1. 对输入的命令进行分割

**设计思路：**

初步按照空格来对用户输入的命令进行分割，在后面涉及到管道和重定向的部分时再按照‘|’‘<’‘>’这些符号来进行分割。

**具体实现：**

需要#include <string.h>

****

（修改一处注释中的错误，savePtr是不能free的，会导致内存错误）

**功能测试：**

在主函数中调用，测试分割功能：



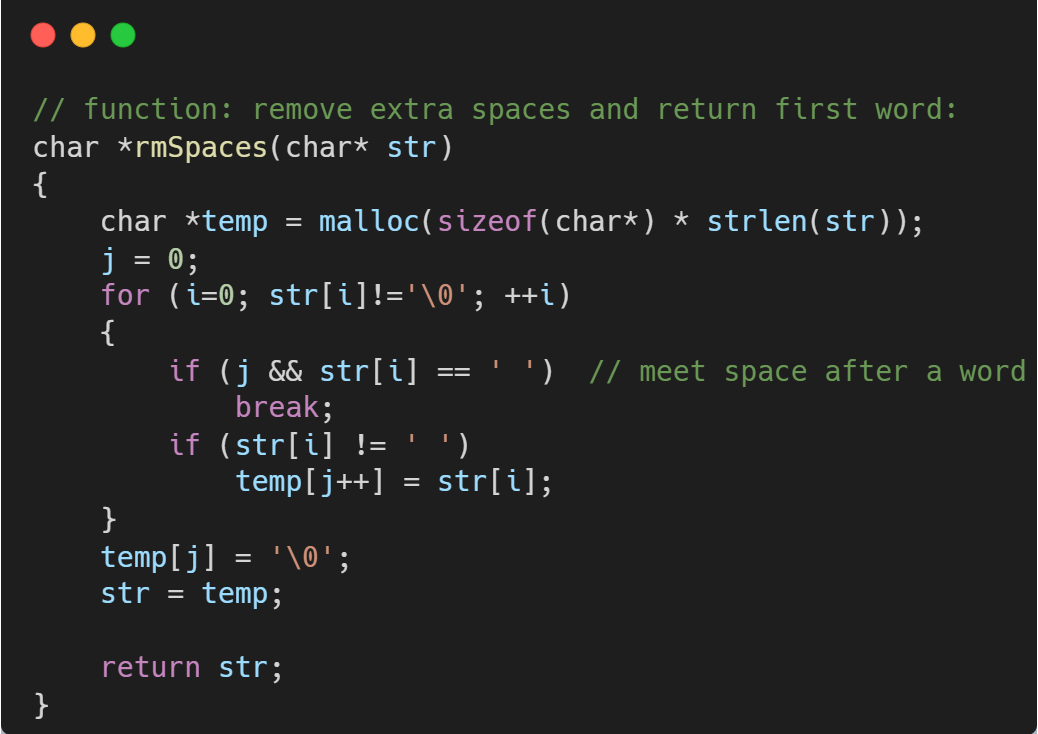
能够按照空格来分割用户输入的命令。

1. 去除分隔后子串中多余的空格

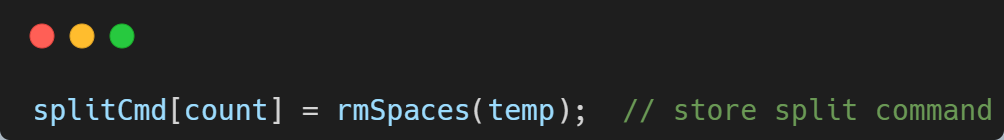
**设计思路：**

用户在输入时可能会在几个命令参数之间误输入多个空格，在对命令进行解析时因为主要是靠字符串比对的方式，所以多余的空格可能会导致命令解析错误，所以需要删除多余的空格。

**具体实现：**

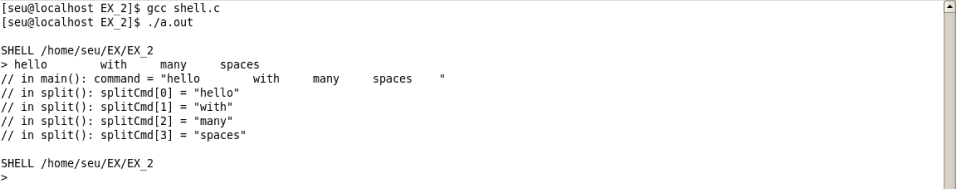


可以修改上一个split()函数，在对字符串分割后，将子串赋值给二维数组的时候调用函数，从而得到的splitCommand中不含多余空格：



**功能测试：**

在命令中加入较多空格，测试功能：



1. 对输入的命令进行解析

**设计思路：**

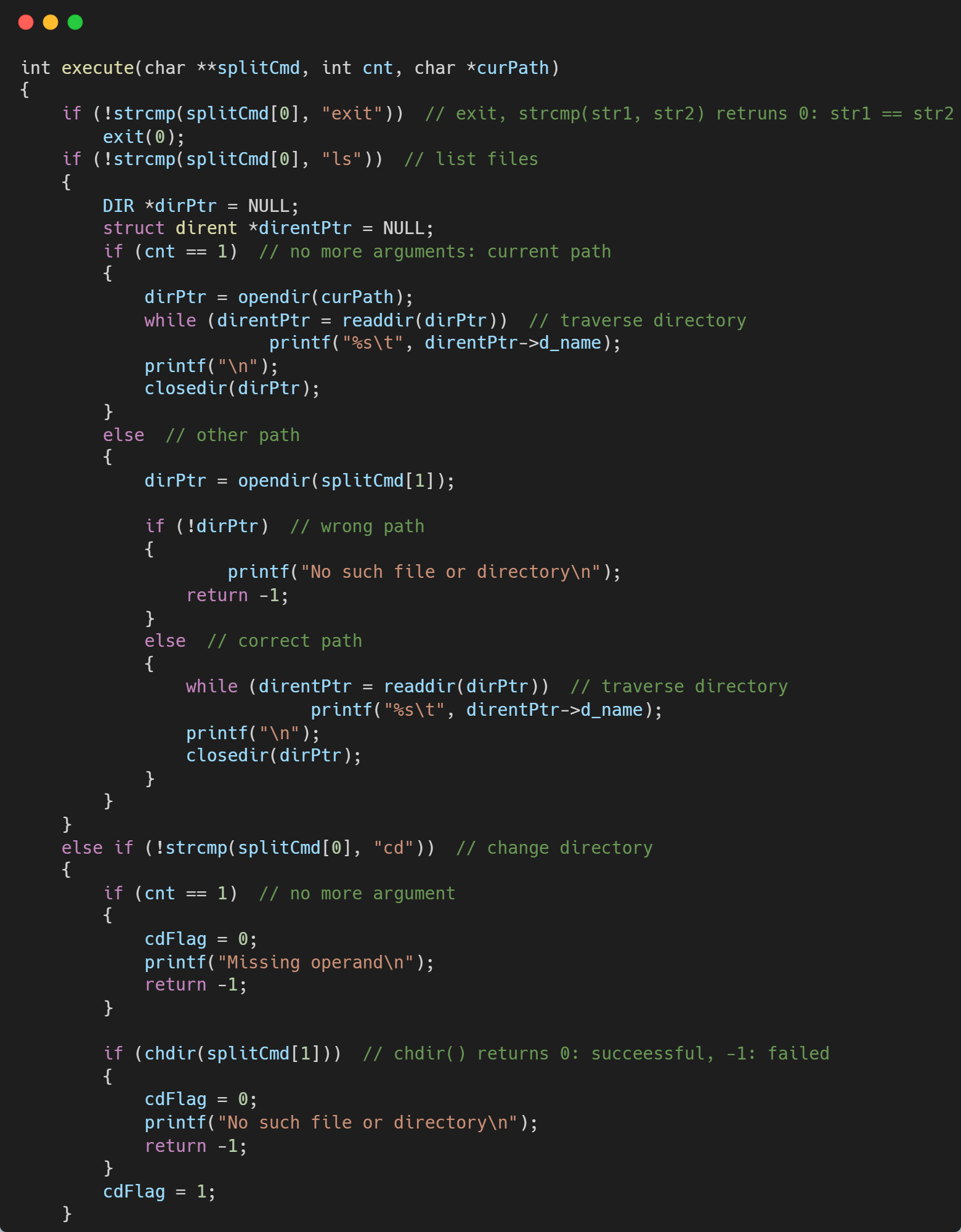
在上一步基础上，对分割后的字符串进行解析，取splitCmd[0]与所支持的命令进行比对，然后跳转到相应的函数或直接实现功能。

设计的命令有：ls（列出目录下所有文件，含参数：指定路径，不含参数：当前路径），cd（改变路径，支持根目录‘/’和上一层目录“..”，home目录‘~’暂时没有实现），create（创建文件，与后面几个命令都支持输入多个参数同时创建、删除多个文件、文件夹），rm（删除文件），mkdir（创建目录），rmdir（删除目录），exit（退出shell）。

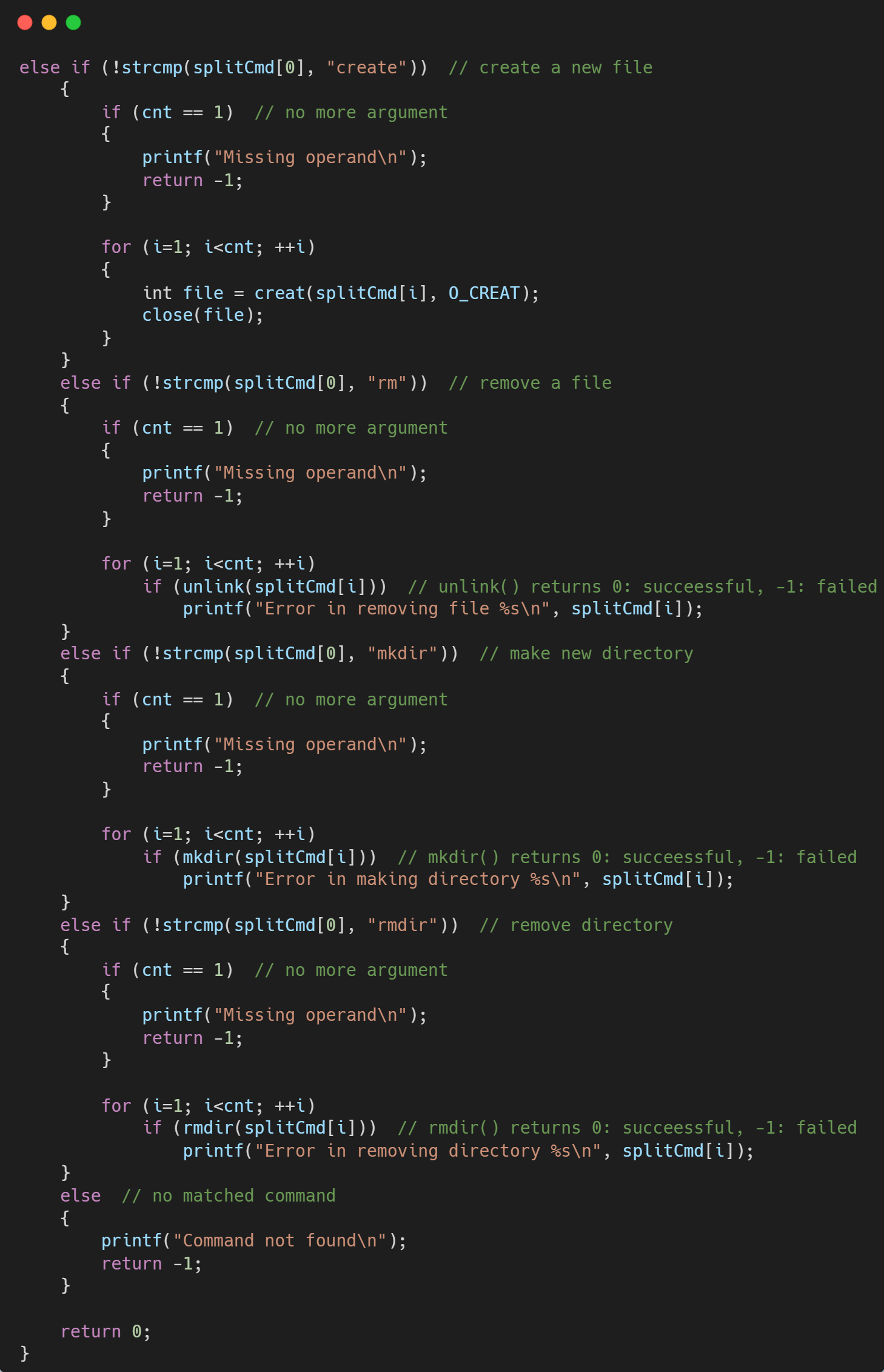
写到后面查到了更多资料之后，才知道有execvp()这样可供直接调用来实现这些功能的函数，功能更全面不需要自己来一个个实现。不过在实现这些功能的时候也学习到了很多系统调用的工作流程，也发现了一些难以解释的bug，在debug过程中也学习到很多。

**具体实现：**

需要#include <stdlib.h>，#include <dirent.h>，#include <fcntl.h>，#include <unistd.h>



（↓续）

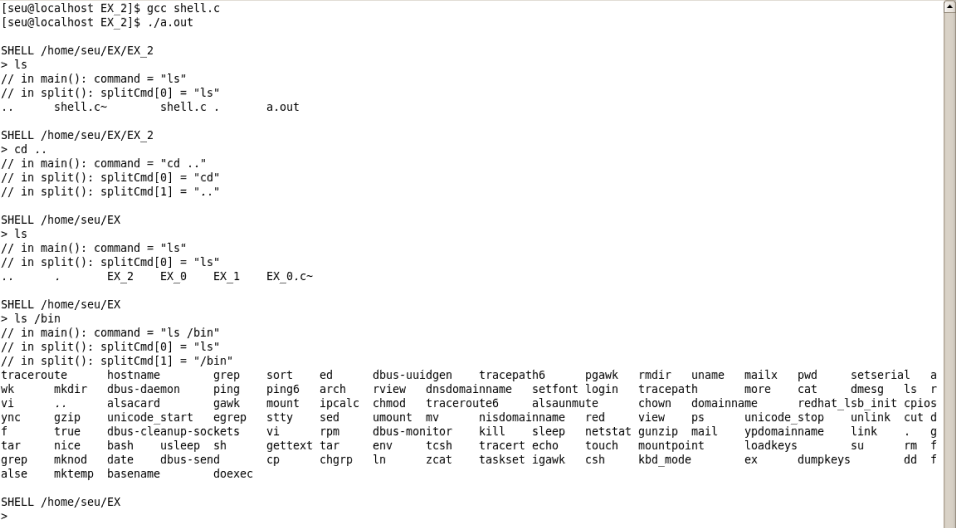


（补充：在创建文件夹和创建文件时应在相应函数中添加设置权限的参数，如0644，否则会出现文件、文件夹无权限的情况）

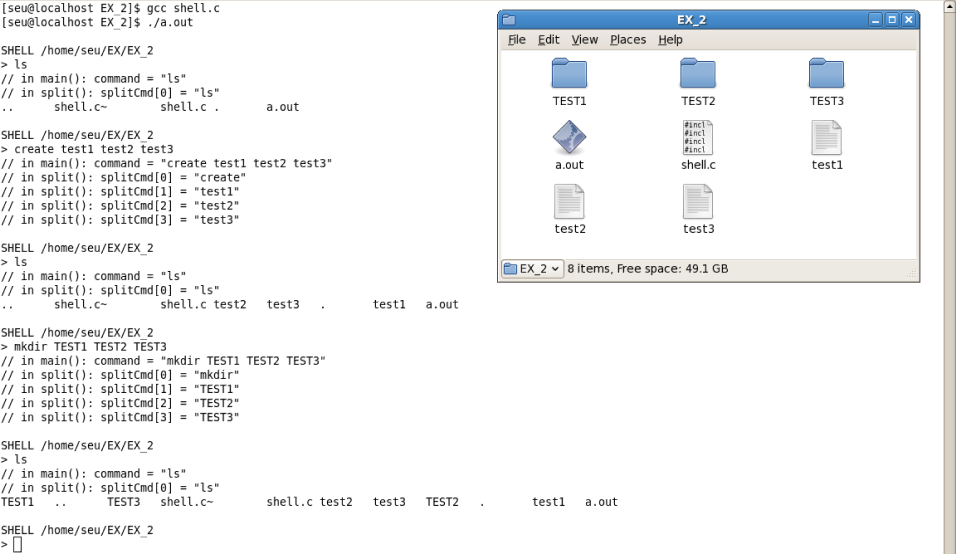
**功能测试：**

在主函数中添加对execute()函数的调用：

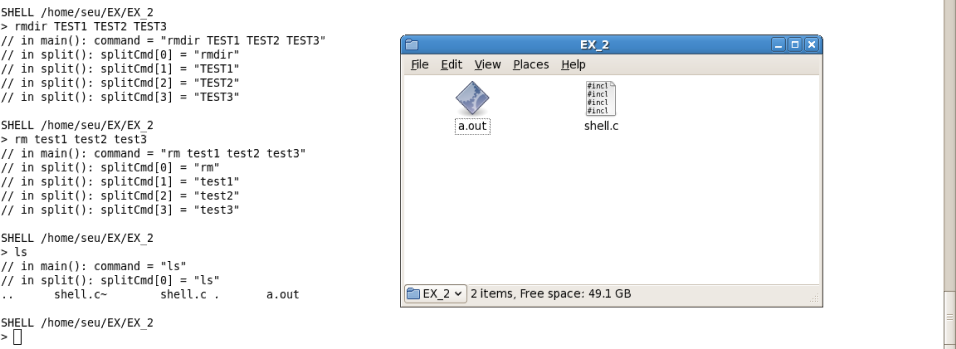
测试ls与cd功能，均可正常跳转与显示：



测试文件、路径创建功能，可以看到创建了相应文件且权限正常：



测试文件、路径删除功能，可以看到均可正常删除：



1. 重定向的实现

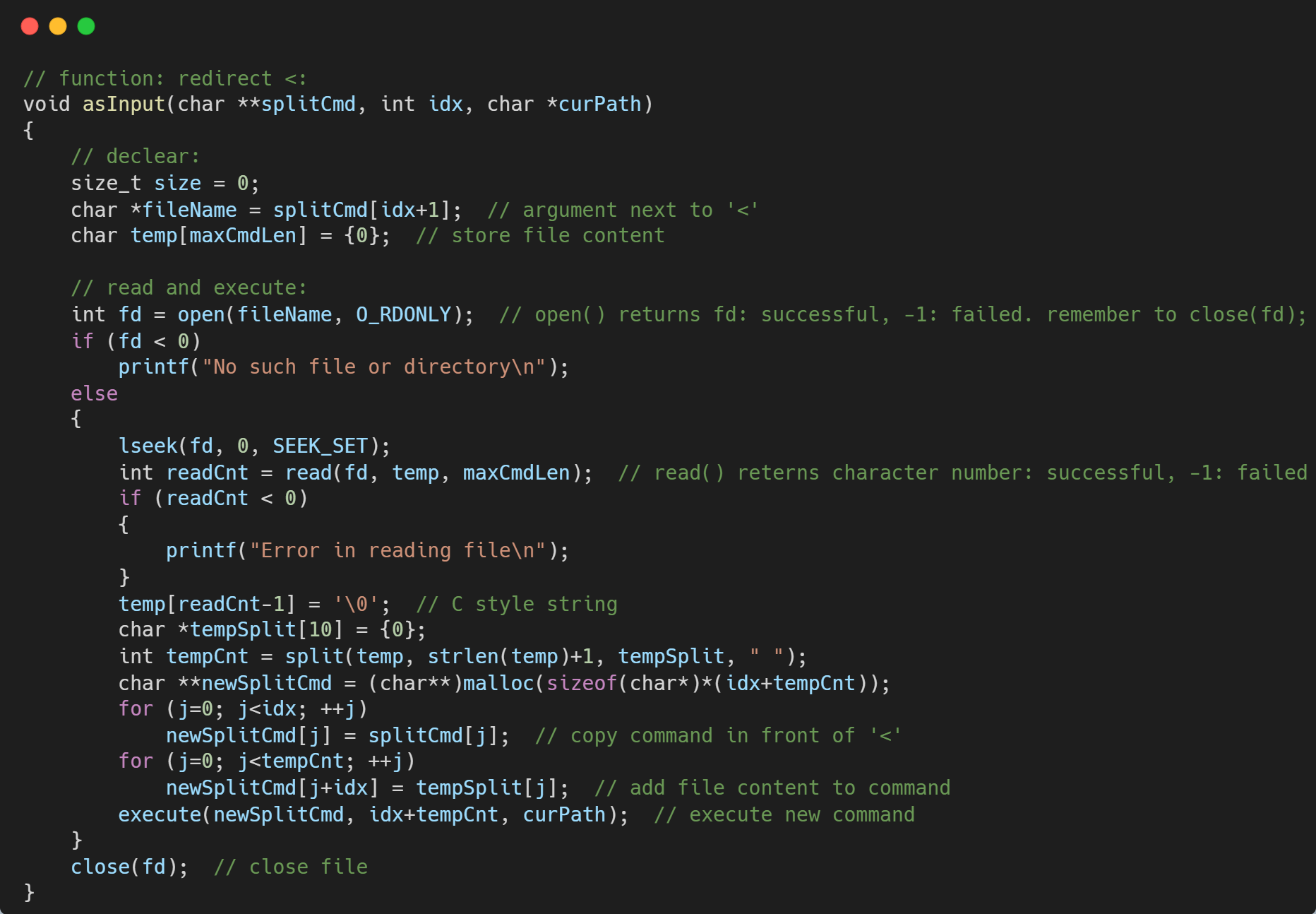
**设计思路：**

将按照空格分割后的命令再按照重定向符号‘<’和‘>’来分为两部分，左侧是命令部分，右侧是文件部分。对于‘<’所要做的是：读出文件的内容，并且要处理其中的空格等作为左侧命令部分的参数输入；对于‘>’所要做的是：将左侧命令的执行结果作为文件的内容写入文件。具体的实现方法见下面的代码部分。

**具体实现：**

需要#include <sys/types.h>，#include <sys/stat.h>

重定向符号‘<’的实现：



重定向符号‘>’的实现：



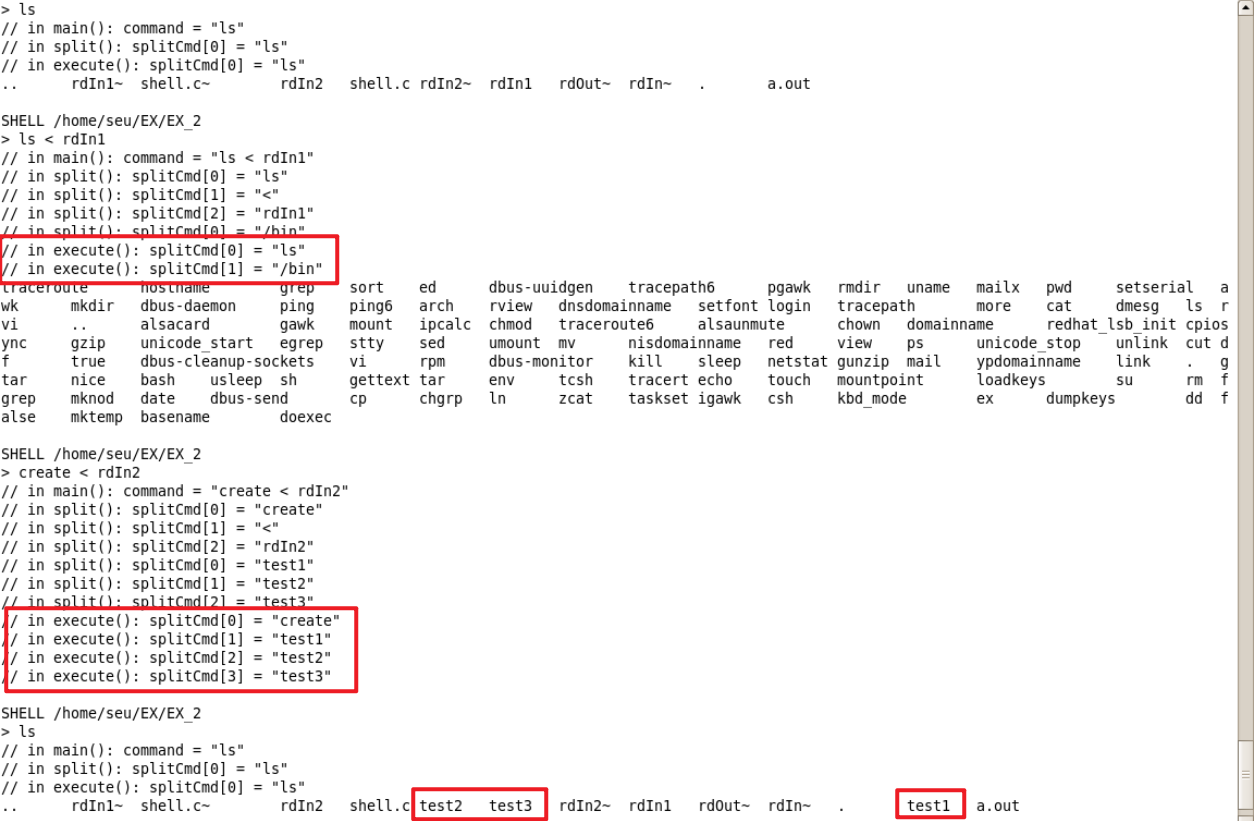
**功能测试：**

创建文件rdIn1，用于接下来测试‘<’的重定向功能，在文件中写入“/bin”，创建文件rdIn2，写入test1 test2 test3并保存，如下：



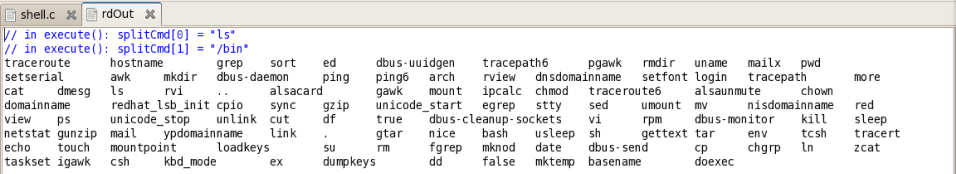


运行shell后依次输入ls < rdIn1和create < rdIn2，结果如下。可以看到rdIn文件中的内容确实作为了‘<’左侧命令的参数执行，结果也正确：



输入ls /bin > rdOut，将“ls /bin”命令的结果写入rdOut文件中，在文本编辑器中查看新出现的rdOut文件，结果如下：





（因为在执行命令前就关闭了标准输出流，而两条调试信息是在execute()函数中打印的，所以被一同写进了文件）

1. 管道的实现

**设计思路：**

管道符号‘|’的作用是将前一个（左侧）命令的输出内容作为后一个（右侧）命令的参数。也可以简单理解为以上两个重定向功能的组合实现。

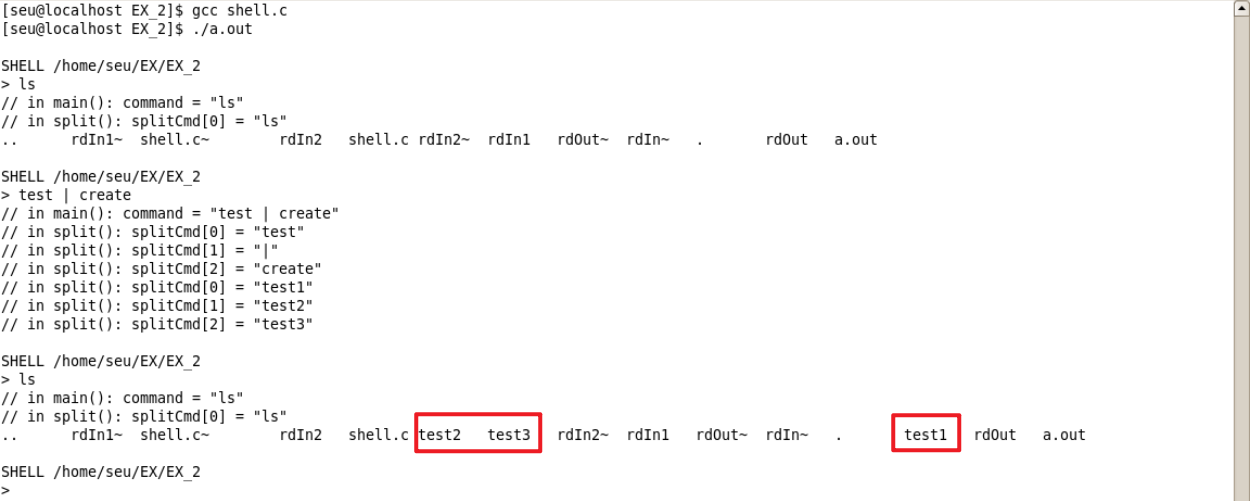
**具体实现：**

****

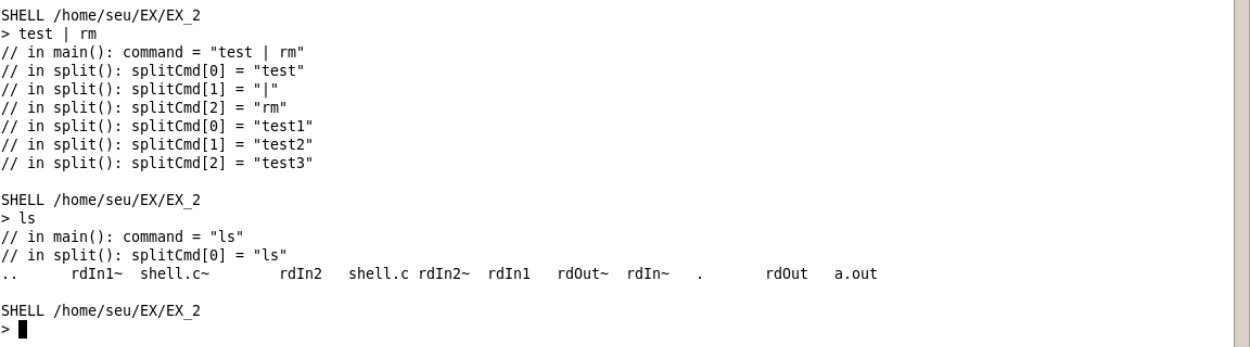
**功能测试：**

因为前面在execute()函数所实现的功能有限，而且大部分不具有内容输出，而具有输出的“ls”命令因为它的输出中含有“..”（上一级目录）和“.”（当前目录），不适合作为其他命令的参数，容易导致错误，所以在execute()中新增了一个test命令，只用于输出内容，所打印的内容也可以更改为所要测试的任何命令。与其他函数相同也是使用printf()来输出内容，所以在原理上是相同的，这里将test命令的输出内容写为“test1 test2 test3”。

输入命令“test | create”，功能应是取test命令的输出内容，即“test1 test2 test3”作为create命令的参数，所以相当于最后执行的是“create test1 test2 test3”，应能在文件夹中看到新建的这几个文件：



再将第二个命令改为“rm”，应会看到几个文件被成功删除：



将test命令输出改为其它内容也可以通过测试。

1. **主要数据结构及其说明**

用到了linux的文件、路径结构，主要是用于路径跳转、读写、新建删除文件和路径时使用。

程序中的函数有以下几个：

1. **int** split(**char** \*cmd, **int** cmdLen, **char** \*\*splitCmd, **const** **char** \*delimeter);
2. **char** \*rmSpaces(**char**\* str);
3. **int** execute(**char** \*\*splitCmd, **int** cnt, **char** \*curPath);
4. **void** asInput(**char** \*\*splitCmd, **int** idx, **char** \*curPath);
5. **void** asOutput(**char** \*\*splitCmd, **int** idx, **char** \*curPath);
6. **void** myPipe(**char** \*\*splitCmd, **int** idx, **int** cnt, **char** \*curPath);
7. 用于分割字符串，输入参数为：用户输入的一行未处理的命令、该命令的长度（含‘\0’）、用于传出分割后的字符串、分隔符，返回值为命令被分割为了几个子串。
8. 用于去除字符串中多余的空格，输入参数为：待处理的字符串，返回值为处理后的字符串，若输入字符串在不同单词中间带有空格，那么只会返回第一个单词。
9. 用于解析输入的命令并执行相应功能，输入参数为：分割好的命令、命令有几个子串，当前路径（因为没有把当前路径写为全局变量，所以需要传入），返回值为执行是否成功，0表示成功，-1表示不成功。
10. 重定向 ‘<’的实现，输入参数为：分割好的命令、重定向符号在命令中的位置（在splitCmd中的下标值）当前路径，无返回值。
11. 重定向 ‘>’的实现，输入参数同上，无返回值。
12. 管道‘|’的实现，参数基本同上，增加了一个cnt传入命令含有几个子串，无返回值。
13. **源程序并附上注释**
14. #include <stdio.h>
15. #include <string.h>
16. #include <stdlib.h>
17. #include <dirent.h>
18. #include <fcntl.h>
19. #include <unistd.h>
20. #include <sys/types.h>
21. #include <sys/stat.h>
23. #define maxCmdLen 512
25. **int** i;
26. **int** j;  // loop variable
28. **int** split(**char** \*cmd, **int** cmdLen, **char** \*\*splitCmd, **const** **char** \*delimeter);
29. **char** \*rmSpaces(**char**\* str);
30. **int** execute(**char** \*\*splitCmd, **int** cnt, **char** \*curPath);
31. **void** asInput(**char** \*\*splitCmd, **int** idx, **char** \*curPath);
32. **void** asOutput(**char** \*\*splitCmd, **int** idx, **char**\* curPath);
33. **void** myPipe(**char** \*\*splitCmd, **int** idx, **int** cnt, **char**\* curPath);
35. **int** main()
36. {
37. **while** (1)
38. {
39. // declear:
40. **size\_t** size = 0;
41. **char** \*curPath = NULL;
42. curPath = getcwd(NULL, 0);  // remember to free(curPath);
43. **char** \*command = NULL;
44. **char** \*splitCommand[10] = {0};
45. **int** rdFlag = 0;  // redirect flag
47. // print:
48. // format: SHELL /...[curPath].../
49. //     > [command]
50. fflush(stdout);
51. printf("\nSHELL ");
52. fflush(stdout);
53. puts(curPath);  // print current path
54. printf("> ");
55. fflush(stdout);
57. // read command of a line:
58. **int** len = getline(&command, &size, stdin);  // remember to free(command);
59. fflush(stdin);
60. **if** (\*command == '\n')  // if only an ENTER
61. {
62. // free:
63. free(curPath);
64. free(command);
65. // restart:
66. **continue**;
67. }
68. command[len-1] = '\0';  // C style string
70. // print to test:
71. printf("// in main(): command = \"%s\"\n", command);
73. // split:
74. **int** splitCount = split(command, len, splitCommand, " ");
76. // redirect and pipe character:
77. **for** (i=0; i<splitCount; ++i)
78. {
79. **if** (splitCommand[i][0] == '<')
80. {
81. rdFlag = 1;
82. asInput(splitCommand, i, curPath);
83. **break**;
84. }
85. **else** **if** (splitCommand[i][0] == '>')
86. {
87. rdFlag = 1;
88. asOutput(splitCommand, i, curPath);
89. **break**;
90. }
91. **else** **if** (splitCommand[i][0] == '|')
92. {
93. rdFlag = 1;
94. myPipe(splitCommand, i, splitCount, curPath);
95. **break**;
96. }
97. }
99. // print to test:
100. //printf("// in main(): splitCmd[1] = \"%s\"\n", splitCommand[1]);
101. //printf("// in main(): splitCnt = %d\n", splitCount);
103. // execute:
104. **if** (!rdFlag)
105. execute(splitCommand, splitCount, curPath);
107. // free:
108. free(curPath);
109. free(command);
110. }
111. **return** 0;
112. }
114. // function: split a line of command:
115. **int** split(**char** \*cmd, **int** cmdLen, **char** \*\*splitCmd, **const** **char** \*delimeter)
116. {
117. // declear:
118. **char** \*temp = NULL;
119. **char** buf[maxCmdLen];
120. strncpy(buf, cmd, cmdLen);  // copy: cmd -> buf
121. temp = buf;
122. **char** \*savePtr = NULL;  // store the split string
123. **int** count = 0;
125. // split:
126. //memset(splitCmd, 0, sizeof(char\*)\*maxCmdLen);  // initialize: fill with 0
127. **while** ((temp = strtok\_r(temp, delimeter, &savePtr)) != NULL && count != 10)
128. {
129. splitCmd[count] = rmSpaces(temp);  // store split command
131. // print to test:
132. //printf("// in split(): splitCmd[%d] = \"%s\"\n", count, splitCmd[count]);
134. temp = NULL;
135. ++count;
136. }
137. **return** count;
138. }
140. // function: remove extra spaces and return first word:
141. **char** \*rmSpaces(**char**\* str)
142. {
143. **char** \*temp = malloc(**sizeof**(**char**\*) \* strlen(str));
144. j = 0;
145. **for** (i=0; str[i]!='\0'; ++i)
146. {
147. **if** (j && str[i] == ' ')  // meet space after a word
148. **break**;
149. **if** (str[i] != ' ')
150. temp[j++] = str[i];
151. }
152. temp[j] = '\0';
153. str = temp;
155. **return** str;
156. }
158. // function: excute basic command:
159. **int** execute(**char** \*\*splitCmd, **int** cnt, **char** \*curPath)
160. {
161. // print to test:
162. //for (j=0;j<cnt;++j)
163. //printf("// in execute(): splitCmd[%d] = \"%s\"\n", j, splitCmd[j]);
165. **if** (!strcmp(splitCmd[0], "exit"))  // exit, strcmp(str1, str2) retruns 0: str1 == str2
166. exit(0);
167. **if** (!strcmp(splitCmd[0], "ls"))  // list files
168. {
169. DIR \*dirPtr = NULL;
170. **struct** dirent \*direntPtr = NULL;
171. **if** (cnt == 1)  // no more arguments: current path
172. {
173. dirPtr = opendir(curPath);
174. **while** (direntPtr = readdir(dirPtr))  // traverse directory
175. printf("%s\t", direntPtr->d\_name);
176. printf("\n");
177. closedir(dirPtr);
178. }
179. **else**  // other path
180. {
181. dirPtr = opendir(splitCmd[1]);
183. **if** (!dirPtr)  // wrong path
184. {
185. printf("No such file or directory\n");
186. **return** -1;
187. }
188. **else**  // correct path
189. {
190. **while** (direntPtr = readdir(dirPtr))  // traverse directory
191. printf("%s\t", direntPtr->d\_name);
192. printf("\n");
193. closedir(dirPtr);
194. }
195. }
196. }
197. **else** **if** (!strcmp(splitCmd[0], "cd"))  // change directory
198. {
199. **if** (cnt == 1)  // no more argument
200. {
201. cdFlag = 0;
202. printf("Missing operand\n");
203. **return** -1;
204. }
206. **if** (chdir(splitCmd[1]))  // chdir() returns 0: succeessful, -1: failed
207. {
208. cdFlag = 0;
209. printf("No such file or directory\n");
210. **return** -1;
211. }
212. cdFlag = 1;
213. }
214. **else** **if** (!strcmp(splitCmd[0], "create"))  // create a new file
215. {
216. **if** (cnt == 1)  // no more argument
217. {
218. printf("Missing operand\n");
219. **return** -1;
220. }
222. **for** (i=1; i<cnt; ++i)
223. {
224. **int** file = open(splitCmd[i], O\_CREAT, 0644);
225. close(file);
226. }
227. }
228. **else** **if** (!strcmp(splitCmd[0], "rm"))  // remove a file
229. {
230. **if** (cnt == 1)  // no more argument
231. {
232. printf("Missing operand\n");
233. **return** -1;
234. }
236. **for** (i=1; i<cnt; ++i)
237. **if** (unlink(splitCmd[i]))  // unlink() returns 0: succeessful, -1: failed
238. printf("Error in removing file %s\n", splitCmd[i]);
239. }
240. **else** **if** (!strcmp(splitCmd[0], "mkdir"))  // make new directory
241. {
242. **if** (cnt == 1)  // no more argument
243. {
244. printf("Missing operand\n");
245. **return** -1;
246. }
248. **for** (i=1; i<cnt; ++i)
249. **if** (mkdir(splitCmd[i], 0644))  // mkdir() returns 0: succeessful, -1: failed
250. printf("Error in making directory %s\n", splitCmd[i]);
251. }
252. **else** **if** (!strcmp(splitCmd[0], "rmdir"))  // remove directory
253. {
254. **if** (cnt == 1)  // no more argument
255. {
256. printf("Missing operand\n");
257. **return** -1;
258. }
260. **for** (i=1; i<cnt; ++i)
261. **if** (rmdir(splitCmd[i]))  // rmdir() returns 0: succeessful, -1: failed
262. printf("Error in removing directory %s\n", splitCmd[i]);
263. }
264. **else** **if** (!strcmp(splitCmd[0], "test"))  // test for pipe
265. printf("test1 test2 test3\n");  // change this line to test different command
266. **else**  // no matched command
267. {
268. printf("Command not found\n");
269. **return** -1;
270. }
272. **return** 0;
273. }
275. // function: redirect <:
276. **void** asInput(**char** \*\*splitCmd, **int** idx, **char** \*curPath)
277. {
278. // declear:
279. **char** \*fileName = splitCmd[idx+1];  // argument next to '<'
280. **char** temp[maxCmdLen] = {0};  // store file content
282. // read and execute:
283. **int** fd = open(fileName, O\_RDONLY);  // open() returns fd: successful, -1: failed. remember to close(fd);
284. **if** (fd < 0)
285. printf("No such file or directory\n");
286. **else**
287. {
288. lseek(fd, 0, SEEK\_SET);
289. **int** readCnt = read(fd, temp, maxCmdLen);  // read() reterns character number: successful, -1: failed
290. **if** (readCnt < 0)
291. {
292. printf("Error in reading file\n");
293. }
294. temp[readCnt-1] = '\0';  // C style string
295. **char** \*tempSplit[32] = {0};
296. **int** tempCnt = split(temp, strlen(temp)+1, tempSplit, " ");
297. **char** \*\*newSplitCmd = (**char**\*\*)malloc(**sizeof**(**char**\*)\*(idx+tempCnt));
298. **for** (j=0; j<idx; ++j)
299. newSplitCmd[j] = splitCmd[j];  // copy command in front of '<'
300. **for** (j=0; j<tempCnt; ++j)
301. newSplitCmd[j+idx] = tempSplit[j];  // add file content to command
302. execute(newSplitCmd, idx+tempCnt, curPath);  // execute new command
303. }
304. close(fd);  // close file
305. }
307. // function: redirect >:
308. **void** asOutput(**char** \*\*splitCmd, **int** idx, **char**\* curPath)
309. {
310. // declear:
311. **char** \*fileName = splitCmd[idx+1];  // argument next to '>'
312. **char** \*\*newSplitCmd = (**char**\*\*)malloc(**sizeof**(**char**\*)\*(idx+1));
313. **for** (j=0; j<idx; ++j)
314. newSplitCmd[j] = splitCmd[j];  // copy command in front of '>'
316. // write and execute:
317. **int** oldStdout = dup(1);
318. **FILE** \*filePtr1 = freopen(fileName, "w+", stdout);
319. execute(newSplitCmd, idx, curPath);  // execute command without "> ..."
320. fclose(stdout);
321. **FILE** \*filePtr2 = fdopen(oldStdout, "w");
322. \*stdout = \*filePtr2;
323. }
325. // function: implement of pipe:
326. **void** myPipe(**char** \*\*splitCmd, **int** idx, **int** cnt, **char**\* curPath)
327. {
328. // declear:
329. **char** \*fileName = "/home/seu/EX/EX\_2/pipeTemp";
330. **char** temp[maxCmdLen] = {0};  // store file content
331. **char** \*\*leftCmd = (**char**\*\*)malloc(**sizeof**(**char**\*)\*(idx+1));  // left command
332. **for** (j=0; j<idx; ++j)
333. leftCmd[j] = splitCmd[j];  // copy command in front of '|'
335. // write:
336. **int** oldStdout = dup(1);
337. **FILE** \*filePtr1 = freopen(fileName, "w+", stdout);
338. execute(leftCmd, idx, curPath);  // execute command without "| ..."
339. fclose(stdout);
340. **FILE** \*filePtr2 = fdopen(oldStdout, "w");
341. \*stdout = \*filePtr2;
343. // read:
344. **int** fd = open(fileName, O\_RDONLY);
345. lseek(fd, 0, SEEK\_SET);
346. **int** readCnt = read(fd, temp, maxCmdLen);
347. temp[readCnt-1] = '\0';
348. **char** \*tempSplit[maxCmdLen] = {0};
349. **int** tempCnt = split(temp, strlen(temp)+1, tempSplit, " ");
350. **char** \*\*rightCmd = (**char**\*\*)malloc(**sizeof**(**char**\*)\*(cnt-idx-1+tempCnt));  // right command
351. **for** (j=0; j<(cnt-idx-1); ++j)
352. rightCmd[j] = splitCmd[j+idx+1];
353. **for** (j=0; j<tempCnt; ++j)
354. rightCmd[j+cnt-idx-1] = tempSplit[j];  // add file content to command
356. // execute:
357. close(fd);  // close file
358. unlink(fileName);  // remove temp file
359. execute(rightCmd, cnt-idx-1+tempCnt, curPath);  // execute new command
360. }
361. **程序运行结果及分析**

（测试与分析部分写在了前面的详细流程之下）

1. **实验体会**
2. 实验中遇到的问题及解决过程

本次实验的工程量较大，一开始会比较容易不知道从何下手，先做的是将问题划分为多个模块、分层并按照一定顺序逐个实现，将程序的功能慢慢丰富起来。

比如说我的顺序是这样：

先在主函数中实现出基本的提示信息输出，看起来像一个shell，以及用户的输入、对用户输入的读取。这一部分主要的难点是当前路径的获取、输出，用户输入的读取，以及内存空间的分配与回收。

然后要对命令进行解析，首先需要按照空格划分为不同的几部分，这时出现的问题是用户在输入时可能会误输入多个空格，那么所留下的多空格就会影响到后面命令的比对结果，需要除去多余空格。这一部分全是在对字符串进行操作，还包括char \*\*这样容易出现内存错误的变量，遇到的问题基本是字符串的操作、作为参数传入传出时的各种错误。

接下来已经得到了分割好的命令，需要进行字符串比对，然后实现相应功能。困难主要在功能的实现上，一开始不知道execvp()这样可以直接调用的现成函数，全在自己实现，包括ls、cd、文件、路径的新建删除，花了很多时间在找这些功能的实现方法，好在大多数都有函数供我们直接调用。

之后是重定向和管道的实现，需要在分割命令时加入对三个符号的判断，需要跳转到相应的函数中执行功能，与前面的基本功能执行应是独立的。所涉及到的主要是输入输出流的控制以及文件的读写。

1. 实验中产生的错误及原因分析

其中出现错误较多的部分主要是内存分配与回收、字符串的定界、数组越界、文件读写部分，这也反映出了在编程时对内存的管理做的不够好，以及一些危险操作的后续处理做的不到位。没有编译器能够即时给出提示和变量当前值的查看，很多时候需要打印查看变量的内容，比较不便。

1. 实验的体会及收获

这次实验能够清晰的了解shell的结构、功能，在自己动手编写出来之后会感觉原来某些功能也没有想象的那么复杂，也会感觉有些功能在实现起来是真的很复杂，比如说ls中我只简单实现了内容的显示，而排序、权、时间、用户信息以及不同类型以不同颜色显示等功能在实现上实在过于复杂。

而且因为实验中对字符串的操作、处理较多，也在不断出错中总结出了一些容易造成错误的操作和解决方法。

1. 对做好今后实验提出建设性建议等。

涉及到内存的操作一定需要谨慎，很多错误在编译上是完全不会有问题的，而在实际运行时，尤其是在没有编译器做保护的状态下，很容易造成内存错误，造成程序的崩溃，所以在内存相关的危险操作上需要多加检查。