东南大学计算机科学与工程学院

操作系统专题实践

实验报告2：Shell的实现

|  |  |
| --- | --- |
| 学号： | 09019106 |
| 姓名： | 牟倪 |
| 完成日期： | 2021年10月26日 |

目录

[一、实验内容与目的 3](#_Toc86155726)

[1.1 实验内容 3](#_Toc86155727)

[1.2 实验目的 3](#_Toc86155728)

[二、设计思路与流程图 3](#_Toc86155729)

[2.1 Shell的基本流程与基本代码结构 3](#_Toc86155730)

[2.2 如何处理简单命令 4](#_Toc86155731)

[2.3 如何实现重定向功能 4](#_Toc86155732)

[2.4如何实现管道功能 4](#_Toc86155733)

[2.5 如何组合上述功能 5](#_Toc86155734)

[三、主要数据结构及说明 5](#_Toc86155735)

[四、源程序 5](#_Toc86155736)

[五、程序运行结果 17](#_Toc86155737)

[5.1 测试简单命令 17](#_Toc86155738)

[5.2 测试输入输出重定向 17](#_Toc86155739)

[5.3 测试管道 18](#_Toc86155740)

[六、实验体会 18](#_Toc86155741)

# 一、实验内容与目的

## 1.1 实验内容

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。

* 支持最基本的shell功能：给出命令行提示符，逐次接受命令。命令分成三种：
  + 内部命令（例如help命令、exit命令等）；
  + 外部命令（常见的ls、cp等，以及其他磁盘上的可执行程序等）；
  + 无效命令（不是上述二种命令）。
* 支持管道功能。比如，在shell中输入“dir | more”，能够执行dir命令，并将其输出通过管道传送给more，作为more的输入。
* 支持重定向功能。比如，在shell中输入“dir > direct.txt”，能够执行dir命令，并将结果输出到direct.txt。

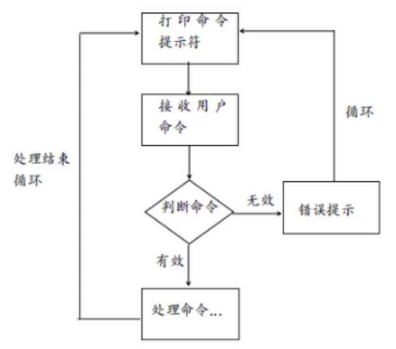
## 1.2 实验目的

通过实验了解Shell实现机制，熟练编写C语言代码。

# 二、设计思路与流程图

## 2.1 Shell的基本流程与基本代码结构

shell的基本流程如图（来自实验指导）所示：



因此，可以给出如下的基本程序结构：

1. while(true){
2. print\_prompt();
3. get\_command();
4. if(command valid)
5. deal\_command();
6. else
7. print\_error();
8. }

## 2.2 如何处理简单命令

strtok\_r函数（strtok函数的Linux安全版本）可以分隔用户输入的内容，解析出命令名称和参数。函数原型为char \*strtok\_r(char \*str, const char \*delim, char \*\*saveptr);。char \*\*saveptr参数是一个指向char \*的指针变量，用来在strtok\_r内部保存切分时的上下文，以应对连续调用分解相同源字符串。第一次调用strtok\_r时，str参数必须指向待提取的字符串；接下来进行连续调用时，saveptr为上次调用后返回的值，不要修改。

可以使用函数execvp实现简单命令，函数原型为int execvp(const char \*file ,char \* const argv []);，execvp()会从PATH 环境变量所指的目录中查找符合参数file 的文件名，找到后便执行该文件，然后将第二个参数argv传给该欲执行的文件。

## 2.3 如何实现重定向功能

freopen函数可以实现重定向，原型为FILE \*freopen( const char \*path, const char \*mode, FILE \*stream )。path：文件名，用于存储输入输出的自定义文件名。 mode：文件打开的模式，和fopen中的模式（如r-只读, w-写）相同。 stream：一个文件，通常使用标准流文件。功能：把预定义的标准流文件定向到由path指定的文件中。

## 2.4如何实现管道功能

pipe函数可以实现管道功能，函数原型为int pipe(int fd[2])。fd[2]：管道的两个文件描述符，调用pipe后可以直接操作这两个文件描述符。管道是半双工的，数据只能向一个方向流动，即一端只能用于读（fd[0]），另一端只能用于写（fd[1]）。一个进程在由 pipe创建管道后，一般再fork一个子进程，然后通过管道实现父子进程间的通信。事实上，只要两个进程具有共同的祖先，就可以采用管道方式来进行通信。

需要建立管道的两个命令之间，用“|”符号来分隔。解析两个命令后，利用pipe函数生成的读取端和写入端，将第一条命令的输出作为第二条命令的输入，就可以实现管道功能。

## 2.5 如何组合上述功能

在main函数里，首先读入一行用户输入，对用户输入做简单解析；查找“<”“>”“|”符号，由此判断是否存在输出重定向、输入重定向和管道。将处理输出重定向、输入重定向和管道的代码进行封装，按需调用。

# 三、主要数据结构及说明

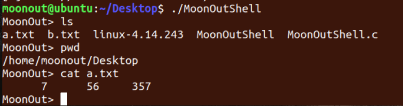
* 使用了数组和指针，存放相关的命令。
* 通过字符串操作，实现基本的逻辑与判断。
* 使用进程的fork。

# 四、源程序

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. #include <sys/types.h>
5. #include <string.h>
6. #include <sys/stat.h>
7. #include <signal.h>
8. #include <fcntl.h>
9. #define hist\_size 1024
10. char \*hist[hist\_size];
11. int f = 0; *// 保存directory变化*
12. int head = 0, filled = 0;
13. *// 输入为字符串，输出为 空格分隔的 第一个单词*
14. char \*trim(char \*string){
15. int i = 0, j = 0;
16. char \*ptr = malloc(sizeof(char\*)\*strlen(string));
17. for (i = 0; string[i] != '\0'; ++i)
18. if (string[i] != ' ')
19. ptr[j++] = string[i];
21. ptr[j] = '\0';
22. return ptr;
23. */\**
24. 为什么要这么做？？
25. string = ptr;
26. return string;
27. \*/
28. }
29. *// 1. 解析用户输入*
30. void parse(char \*word, char \*\*argv){
31. *// 以指针的形式改变argv，使argv存储分隔后的内容，并处理一些简单命令*
32. int count = 0;
33. memset(argv, 0, sizeof(char\*)\*(64)); *// 清零argv中的内容*
34. char \*lefts = NULL;
35. const char \*split = " ";  *// 用空格来分隔*
36. while (1){
37. char \*p = strtok\_r(word, split, &lefts);
38. if (p == NULL){
39. break;
40. }
41. argv[count++] = p;
42. word = lefts;
43. }
44. if (strcmp(argv[0], "exit") == 0) exit(0); *// 直接退出*
45. else if (strcmp(argv[0], "cd") == 0){
46. int ch = chdir(argv[1]); *// 用于改变当前工作目录，参数为目标目录*
47. f = 1;
48. }
49. }
50. *// 2. 执行简单命令*
51. void execute(char \*\*argv){
52. pid\_t pid; int status;
53. if ((pid = fork()) < 0){
54. printf("error: fork failed.\n");
55. exit(1);
56. }
57. else if (pid == 0){ *// 我是子进程*
58. if (execvp(argv[0], argv) < 0 && strcmp(argv[0], "cd"))
59. printf("error: invalid command.\n");
60. exit(0);
61. }
62. else{ *// 我是父进程*
63. while (wait(&status) != pid) *// 自旋等待*
64. ;
65. }
66. }
67. *// 3. 输出的重定向*
68. void  execute\_file(char \*\*argv, char \*output){
70. pid\_t pid;
71. int status;
72. char \*file = NULL;
73. if ((pid = fork()) < 0){
74. printf("error: fork failed.\n");
75. exit(1);
76. }
77. else if (pid == 0) { *// 子进程的工作*
78. if (strstr(output, ">")>0){ *// 返回字符串中 首次出现子串的地址*
80. char \*p = strtok\_r(output, ">", &file);
81. output += 1; *// change2*
82. file = trim(file); *// 得到 > 后的第一个单词，也就是输出重定向的目标文件*
83. int old\_stdout = dup(1);
85. FILE \*fp1 = freopen(output, "w+", stdout);
86. execute\_file(argv, file);
87. fclose(stdout);
88. FILE \*fp2 = fdopen(old\_stdout, "w");
89. \*stdout = \*fp2;
90. exit(0);
91. }
92. if (strstr(output, "<") > 0){ *// 还要输入重定向的……*
93. char \*p = strtok\_r(output, "<", &file);
94. file = trim(file);
95. int fd = open(file, O\_RDONLY);
96. if (fd<0){
97. printf("No such file or directory.\n");
98. exit(0);
99. }
100. }
101. if (strstr(output, "|") > 0){ *// 还有管道的……*
102. fflush(stdout); printf("here"); fflush(stdout);
103. char \*p = strtok\_r(output, "|", &file);
104. file = trim(file);
106. char \*args[64];
107. parse(file, args);
108. execute(args);
109. }
110. int old\_stdout = dup(1);
111. FILE \*fp1 = freopen(output, "w+", stdout);
112. if (execvp(argv[0], argv) < 0) printf("error: in exec\n");
113. fclose(stdout);
114. FILE \*fp2 = fdopen(old\_stdout, "w");
115. \*stdout = \*fp2;
116. exit(0);
117. }
118. *// 自旋等待*
119. else{
120. while (wait(&status) != pid)
121. ;
122. }
123. }
124. *// 4. 输入的重定向*
125. void  execute\_input(char \*\*argv, char \*output){
126. pid\_t pid;
127. int fd;
128. char \*file;
129. int flag = 0;
130. int status;
131. if ((pid = fork()) < 0) {
132. printf("error: fork failed\n");
133. exit(1);
134. }
135. else if (pid == 0){ *// 和上一个函数代码基本一样*
136. if (strstr(output, "<") > 0){
137. char \*p = strtok\_r(output, "<", &file);
138. file = trim(file);
139. flag = 1;
140. fd = open(output, O\_RDONLY);
141. if (fd<0){
142. printf("No such file or directory.\n");
143. exit(0);
144. }
145. output = file;
146. }
147. if (strstr(output, ">") > 0){
148. char \*p = strtok\_r(output, ">", &file);
149. file = trim(file);
150. flag = 1;
151. fflush(stdout);
152. fflush(stdout);
153. int old\_stdout = dup(1);
154. FILE \*fp1 = freopen(file, "w+", stdout);
155. execute\_input(argv, output);
156. fclose(stdout);
157. FILE \*fp2 = fdopen(old\_stdout, "w");
158. \*stdout = \*fp2;
159. exit(0);
160. }
161. if (strstr(output, "|") > 0) {
162. char \*p = strtok\_r(output, "|", &file);
163. file = trim(file);
164. flag = 1;
165. char \*args[64];
166. parse(file, args);
167. int pfds[2];
168. pid\_t pid, pid2;
169. int status, status2;
170. pipe(pfds);
171. int fl = 0;
172. if ((pid = fork()) < 0){
173. printf("error: fork failed\n");
174. exit(1);
175. }
176. if ((pid2 = fork()) < 0){
177. printf("error: fork failed\n");
178. exit(1);
179. }
180. if (pid == 0 && pid2 != 0){
181. close(1);
182. dup(pfds[1]);
183. close(pfds[0]);
184. close(pfds[1]);
185. fd = open(output, O\_RDONLY);
186. close(0);
187. dup(fd);
188. if (execvp(argv[0], argv) < 0){
189. close(pfds[0]);
190. close(pfds[1]);
191. printf("error:in exec");
192. fl = 1;
193. exit(0);
194. }
195. close(fd);
196. exit(0);
197. }
198. else if (pid2 == 0 && pid != 0 && fl != 1){
199. close(0);
200. dup(pfds[0]);
201. close(pfds[1]);
202. close(pfds[0]);
203. if (execvp(args[0], args) < 0){
204. close(pfds[0]);
205. close(pfds[1]);
206. printf("error:in exec");
207. exit(0);
208. }
209. }
210. else {
211. close(pfds[0]);
212. close(pfds[1]);
213. while (wait(&status) != pid);
214. while (wait(&status2) != pid2);
215. }
216. exit(0);
217. }
218. fd = open(output, O\_RDONLY);
219. close(0);
220. dup(fd);
221. if (execvp(argv[0], argv) < 0)
222. printf("error: in exec\n");
223. close(fd);
224. exit(0);
225. }
226. else {
227. while (wait(&status) != pid)
228. ;
229. }
230. }
231. *// 5. 管道的实现1*
232. void execute\_pipe(char \*\*argv, char \*output) {
233. int pfds[2], pf[2], flag;
234. char \*file;
235. pid\_t pid, pid2, pid3;
236. int status, status2, old\_stdout;
237. pipe(pfds); *// 创建管道！*
238. int blah = 0;
239. char \*args[64];
240. char \*argp[64];
241. int fl = 0;
242. if ((pid = fork()) < 0){
243. printf("error: fork failed\n");
244. exit(1);
245. }
246. if ((pid2 = fork()) < 0){
247. printf("error: fork failed\n");
248. exit(1);
249. }
250. if (pid == 0 && pid2 != 0) { *// 现在是哪一个进程*
251. close(1);
252. dup(pfds[1]);
253. close(pfds[0]);
254. close(pfds[1]);
255. if (execvp(argv[0], argv) < 0){ *// 还是要执行的*
256. close(pfds[0]);
257. close(pfds[1]);
258. printf("error: in exec");
259. fl = 1;
260. kill(pid2, SIGUSR1);
261. exit(0);
262. }
263. }
264. if (pid2 == 0 && pid != 0) { *// 现在是哪一个进程*
265. if (fl == 1) exit(0);
266. if (strstr(output, "<") > 0) { *// 输入重定向*
267. char \*p = strtok\_r(output, "<", &file);
268. file = trim(file);
269. flag = 1;
270. parse(output, args); *// 把output parse一下，存到args里*
271. execute\_input(args, file);
272. close(pfds[0]);
273. close(pfds[1]);
274. exit(0);
275. }
276. if (strstr(output, ">") > 0) { *// 输出重定向*
277. char \*p = strtok\_r(output, ">", &file);
278. file = trim(file);
279. flag = 1;
280. parse(output, args);
281. blah = 1;
282. }
283. else {
284. parse(output, args);
285. }
286. close(0);
287. dup(pfds[0]); *// 复制文件描述词，新的文件描述词和 参数 指的是同一个文件, 共享所有的锁定、读写位置和各项权限*
288. close(pfds[1]);
289. close(pfds[0]);
290. if (blah == 1) { *// 输出重定向*
291. old\_stdout = dup(1);
292. FILE \*fp1 = freopen(file, "w+", stdout);
293. }
294. if (execvp(args[0], args) < 0) {
295. fflush(stdout);
296. printf("error: in exec %d\n", pid);
297. kill(pid, SIGUSR1);
298. close(pfds[0]);
299. close(pfds[1]);
300. }
301. fflush(stdout);
302. if (blah == 1) { *// 输出重定向*
303. fclose(stdout);
304. FILE \*fp2 = fdopen(old\_stdout, "w");
305. \*stdout = \*fp2;
306. }
307. }
308. else {
309. close(pfds[0]);
310. close(pfds[1]);
311. while (wait(&status) != pid)
312. ;
313. while (wait(&status2) != pid2)
314. ;
315. }
316. }
317. *// 6. 管道的实现2*
318. void execute\_pipe2(char \*\*argv, char \*\*args, char \*\*argp){
319. int status;
320. int i;
321. int pipes[4];
322. pipe(pipes);
323. pipe(pipes + 2);
324. if (fork() == 0) {
325. dup2(pipes[1], 1);
326. close(pipes[0]);
327. close(pipes[1]);
328. close(pipes[2]);
329. close(pipes[3]);
330. if (execvp(argv[0], argv) < 0) {
331. fflush(stdout);
332. printf("error:in exec");
333. fflush(stdout);
334. close(pipes[0]);
335. close(pipes[1]);
336. close(pipes[2]);
337. close(pipes[3]);
338. exit(1);
339. }
340. }
341. else {
342. if (fork() == 0) {
343. dup2(pipes[0], 0);
344. dup2(pipes[3], 1);
345. close(pipes[0]);
346. close(pipes[1]);
347. close(pipes[2]);
348. close(pipes[3]);
349. if (execvp(args[0], args) < 0) {
350. fflush(stdout);
351. printf("error:in exec");
352. fflush(stdout);
353. close(pipes[0]);
354. close(pipes[1]);
355. close(pipes[2]);
356. close(pipes[3]);
357. exit(1);
358. }
359. }
360. else {
361. if (fork() == 0) {
362. dup2(pipes[2], 0);
363. close(pipes[0]);
364. close(pipes[1]);
365. close(pipes[2]);
366. close(pipes[3]);
367. if (execvp(argp[0], argp) < 0) {
368. fflush(stdout);
369. printf("error:in exec");
370. fflush(stdout);
371. close(pipes[0]);
372. close(pipes[1]);
373. close(pipes[2]);
374. close(pipes[3]);
375. exit(1);
376. }
377. }
378. }
379. }
380. close(pipes[0]);
381. close(pipes[1]);
382. close(pipes[2]);
383. close(pipes[3]);
384. for (i = 0; i < 3; i++)
385. wait(&status);
386. }
388. int  main() {
389. *// 初始化*
390. char line[1024];
391. char \*argv[64];
392. char \*args[64];
393. char \*left;
394. size\_t size = 0;
395. char ch;
396. int count = 0;
397. char \*tri;
398. char \*second;
399. char \*file;
400. int i;
401. for (i = 0; i < hist\_size; i++){
402. hist[i] = (char \*)malloc(150);
403. }
405. *// 主循环*
406. while (1){
407. count = 0;
408. int flag = 0;
409. char \*word = NULL;
410. char \*dire[] = { "pwd" }; *// print work directory 用于显示工作目录*
411. fflush(stdout); *// 强制马上输出，避免错误*
412. printf("MoonOut> ");
413. fflush(stdout);
414. execute(dire); *// 打印现在的directory*
415. printf("$");
416. int len = getline(&word, &size, stdin); *// 得到一行内容*
417. if (\*word == '\n') *// 是回车…*
418. continue;
419. word[len - 1] = '\0';
420. char \*file = NULL;
421. int i = 0;
422. char \*temp = (char \*)malloc(150);
423. strcpy(temp, word); *// 把word的内容粘到temp里*
424. parse(temp, argv); *// 解析命令*
426. *// 存储directory*
427. strcpy(hist[(head + 1) % hist\_size], word);
428. head = (head + 1) % hist\_size;
429. ++filled;
430. if (strcmp(word, "exit") == 0) exit(0); *// 请直接退出吧*
431. *// 对字符进行遍历查找，修改flag，决定接下来的策略*
432. for (i = 0; word[i]!='\0'; ++i){
433. if (word[i] == '>'){ *// 输出重定向*
434. char \*p = strtok\_r(word, ">", &file);
435. file = trim(file);
436. flag = 1;
437. break;
438. }
439. else if (word[i] == '<'){ *// 输入重定向*
440. char \*p = strtok\_r(word, "<", &file);
441. file = trim(file);
442. flag = 2;
443. break;
444. }
445. else if (word[i] == '|'){ *// 管道*
446. char \*p = strtok\_r(word, "|", &left);
447. flag = 3;
448. break;
449. }
450. }
451. *// 对每种情况进行分别处理*
452. if (flag == 1){ *// 输出重定向*
453. parse(word, argv);   *//parsed command stored in argv*
454. execute\_file(argv, file);
455. }
456. else if (flag == 2){ *// 输入重定向*
457. parse(word, argv);
458. execute\_input(argv, file);
459. }
460. else if (flag == 3){ *// 管道*
461. char \*argp[64];
462. char \*output, \*file;
463. if (strstr(left, "|") > 0){
464. char \*p = strtok\_r(left, "|", &file);
465. parse(word, argv);
466. parse(left, args);
467. parse(file, argp);
468. execute\_pipe2(argv, args, argp);
469. }
470. else{
471. parse(word, argv);
472. execute\_pipe(argv, left);
473. }
474. }
475. else{ *// 简单命令*
476. parse(word, argv);
477. execute(argv);
478. }
479. }
480. }

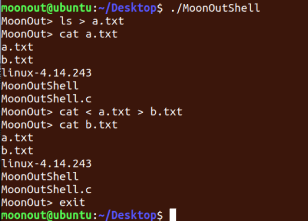
# 五、程序运行结果

## 5.1 测试简单命令



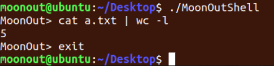
可以看出，shell可以实现简单命令的正确执行。

## 5.2 测试输入输出重定向



可以看出，基本的输入输出重定向功能已经实现。

## 5.3 测试管道



可以看出，基本的管道功能已经实现。

# 六、实验体会

在本次实验的过程中，我最深的感受是：着手写代码之前，要做好整个程序逻辑的设计与规划。在大二下学期，我已经在操作系统课程的实验中写过简易的shell，因此对于fork、execvp、freopen等函数的应用有一定经验；但由于在程序的设计中没有安置好管道功能的位置，导致代码冗长、重复代码较多。