山东大学 计算机科学与技术 学院

操作系统 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：201818130189 | 姓名：王硕 | | 班级：18级2班 |
| 实验题目：MSH（shell） | | | |
| 实验学时：4学时 | | 实验日期：2020 11 06 | |
| 实验目的：  揭示UNIX系统中的shell是如何使用基础系统调用 | | | |
| 硬件环境：  MacOS、Ubuntu Linux 18.04 | | | |
| 软件环境：  VS Code、Sublime Text | | | |
| 实验内容与设计：  实现一个在 Linux 系统下运行的简单Shell，它能够运行带参数的命令、重定向输入输出、建立双向管道，而且这个Shell能够响应 Linux 中的信号，你可以假定当进程收到 SIGINT 时会被杀死。其余的Shell特性不需要实现。   1. 命令的读入和解析   获取输入的信息，用scanf（）函数按字符进行读取，当读到'\0'和'\n'的时候停止读入，其余的都存储在字符数组中，用于之后的解析；    对命令的解析，利用到了函数strtok\_r（），以“ ”为分割的标志，那么就能得到命令和命令的参数，方便接下来的执行过程；  函数原型如下：  char \*strtok\_r(char \*str, const char \*delim, char \*\*saveptr);  函数的返回值是“排在前面的”被分割出的字符串，或者为NULL；  str是传入的字符串。需要注意的是：第一次使用strtok\_r之后，要把str置为NULL；  delim指向分割依据的字符串，常见的就是“ ”、“,”等；  saveptr保存剩下的待分割的字符串；  特别地，str不能指向常量指针，不然会出现程序崩溃。     1. 处理不包含管道的命令 2. 定义需要的变量      1. 确定重定向符号的个数 2. 利用strcmp（）函数去匹配需要的字符串，即重定向输出"<"和重定向输入符">"；   函数原型如下：  int strcmp(const char\* stri1，const char\* str2);  参数 str1 和 str2 是参与比较的两个字符串；  函数会根据ASCII码依次比较 str1 和 str2 的每一个字符，直到出现不到的字符，或者到达字符串末尾（遇见‘\0’）；  返回值：  如果返回值 < 0，则表示 str1 小于 str2；  如果返回值 > 0，则表示 str2 小于 str1；  如果返回值 = 0，则表示 str1 等于 str2；   1. 出现了重定向符号之后肯定就是输入和输出的目标文件，也需要相应的变量去存储； 2. 对于重定向操作，需要执行的指令到重定向符之前，需要把指令执行的下标进行修改；      1. 可能出现的报错情况 2. 连续的重定向输入、输出； 3. 重定向输入输出的目标文件不存在；      1. 对命令进行处理 2. 创建子进程，用于执行命令 3. 执行过程，如果出现有重定向操作的时候，需要将对应的目标文件按照特定的模式打开，使用到的函数就是open（）函数   函数原型如下：  int open(const char \* pathname, int flags, mode\_t mode);  参数 pathname 指向欲打开的文件路径字符串；  参数flags则是打开文件的方式，例如O\_RDONLY 以只读方式打开文件，O\_WRONLY 以只写方式打开文件，O\_TRUNC 若文件存在并且以可写的方式打开时, 此旗标会令文件长度清为0, 而原来存于该文件的资料也会消失……  参数mode 则有下列数种组合, 只有在建立新文件时才会生效；   1. 将命令存储到字符串数组里面，然后利用函数execvp（）进行执行；   函数原型如下：  int execvp(const char\* file, const char\* argv[]);  第一个参数是要运行的文件，会在环境变量PATH中查找file，并执行；  第二个参数，是一个参数列表，如同在shell中调用程序一样；  失败会返回－1， 成功无返回值，但是，失败会在当前进程运行，执行成功后，直接结束当前进程，可以在子进程中运行。     1. 处理管道指令 2. 考虑特殊情况，‘｜’后面没有指令，那么就等于没后续的指令可以执行； 3. 先定义好管道，然后申请子进程； 4. 需要对输入、输入端进行重定向；   原因如下：  对于第一个和最后一个管道，特殊的地方在于第一个管道的输入端仍是标准输入端，而输出端则需要重定向为管道的输出端，作为下一个管道的输入端；对于最后一个管道，输入端需要从标准输入重定向为管道输入端，输出则是标准输出端，方便最终的结果输出；  对于中间位置的管道，它们的输入输出就都需要重定向为管道的输出和输入，方便从前一个管道读取输入信息，然后将自己执行命令之后的结果输出到下一个管道中；  对于第一个管道的输入端和最后一个管道的输出端的特殊情况，在main函数中，首先将管道的输出和输入端（此时都是标准的）进行一个备份；然后开始执行处理管道指令的操作，执行完之后，此时最后一个管道的输出端应该是被定义成了管道的输出端，在main函数中将备份过的标准输出和输入端进行还原，这样就满足了需求。  dup和dup2函数原型：   1. int dup(int oldfd);   用于复制参数oldfd所指的文件描述符。当复制成功是，返回最小的尚未被使用过的文件描述符，若有错误则返回-1.错误代码存入errno中返回的新文件描述符和参数oldfd指向同一个文件，这两个描述符共享同一个数据结构，共享所有的锁定，读写指针和各项全现或标志位。   1. int dup2(int oldfd, int newfd);   dup2可以用参数newfd指定新文件描述符的数值。若参数newfd已经被程序使用，则系统就会将newfd所指的文件关闭，若newfd等于oldfd，则返回newfd,而不关闭newfd所指的文件。dup2所复制的文件描述符与原来的文件描述符共享各种文件状态。共享所有的锁定，读写位置和各项权限或flags等。若dup2调用成功则返回新的文件描述符，出错则返回-1。     1. 收到SIGINT信号的情况   需要使用到的函数：kill()函数是一个用于向指定进程或进程组发送信号的函数；  函数原型为：  int kill(int pid,int sig);  pid是一个进程或一组进程的标识符，sig是要发送的软中断信号，也就是所要发送的信号量；  当pid > 0时，将sig信号发送给pid进程；  当pid == 0时，将sig信号发送给与发送进程同组的所有进程；  当pid == -1时，将sig信号发送给所有用户标识符真正等于发送进程的有效用户标识符的进程； | | | |
| 结论分析与体会：   1. 输入的命令的识别   实际上对输入命令的处理就是一个对字符串的处理，用“ ”进行分割，然后利用strtok\_r函数进行分割，获取要执行的命令和之后的操作；需要注意的就是函数每次切割之后得到的剩余字符串的存储方式；   1. 没有管道指令的命令处理   这一段实际上才是处理程序段的重点。因为其实管道指令也就是一个个命令的处理，只不过是换到了相连的管道中去执行。处理的关键在于是否有重定向操作，如果没有，直接利用execvp函数执行就可以了；如果有的话，那么还需要先用特定的模式操作将目标文件打开，再执行操作。   1. 管道指令的处理   重点即在于对于输出和输入端的定义，需要考虑清楚对于在执行操作的管道，它的输出和输入究竟是需要定向为标准的，还是管道的。最终实现的目标情况就是命令会通过管道将左侧程序的输出作为输入传递给右侧程序。管道支持多级连接，即左侧的标准输出内容会被一次 传递给右侧的下一级，作为它们的标准输入。  4、shell程序的实现过程  建立一个新的进程，然后在那个进程中运行一个程序（如完成ls操作）然后等待那个进程执行结束。然后shell便可读取新的一行输入，建立一个新的进程，在这个进程中运行程序 并等待这个进程结束。所以要写一个shell，需要循环以下过程：  （1）获取命令行  （2）解析命令行  （3）建立一个子进程（fork）  （4）替换子进程（execvp）  （5）父进程等待子进程退出（wait） | | | |

**附录：**

**msh.c原码**

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <unistd.h>
5. #include <sys/types.h>
6. #include <sys/wait.h>
7. #include <signal.h>
8. #include <wait.h>
9. #include <time.h>
10. #include <fcntl.h>
11. pid\_t parentpid, childpid;
12. **typedef** **void** (\*sighandler\_t) (**int**);
14. //信号处理函数
15. **void** sigcat(){
16. **if**(getpid() == parentpid)
17. kill(childpid, SIGINT);
18. }
20. //获取输入的信息
21. **void** getstr(**char** \*str){
22. **char** ch;
23. **int** i = 0;
24. **while**(scanf("%c", &ch) != EOF){
25. **if**(ch != '\0' && ch != '\n')
26. str[i++] = ch;
27. **else**
28. **break**;
29. }
30. }
32. //解析命令
33. **int** parse(**char** \*line, **char** \*\*args){
34. memset(args, 0, **sizeof**(**char**\*)\*64);
35. **int** cnt = 0;//记录切割后的字符串的数目
36. **char** \*str = line;//保存输入的命令
37. **char** \*tmp = NULL;//保存每次分割后的剩余字符串
39. //对输入命令字符串按照“ ”进行切割
40. **while**((args[cnt] = strtok\_r(str, " ", &tmp)) != NULL){
41. cnt++;
42. str = NULL;
43. }
45. **return** cnt;
46. }
48. //执行无管道的指令
49. **void** exec(**char** \*cmd){
50. **size\_t** pid;
51. **char** \*in = NULL, \*out = NULL;//重定向的输入和输出文件的名称
52. **int** rein = 0, reout = 0;//重定向输入和输出的个数
53. **int** status;
54. **char** \*args[256];
55. **char** temp[256];
56. strcpy(temp, cmd);
57. **int** argc = parse(temp, args);//分割后得到的字符串的数目
58. **int** eidx = argc;//对于重定向操作的终止下标
60. **for**(**int** i = 0; i < argc; i++){
61. //确定重定向符“>”和“<”的数目
62. **if**(!strcmp(args[i], "<")){
63. **if**(i + 1 >= argc){
64. printf("File name dose not exist.\n");
65. exit(1);
66. }
67. **else**
68. in = args[++i];//存储文件名
70. rein++;
71. //对于重定向操作,需要执行的操作应该只到倒数第二个字符
72. **if**(eidx == argc) eidx = i - 1;
73. }
75. **if**(!strcmp(args[i], ">")){
76. **if**(i + 1 >= argc){
77. printf("File name dose not exist.\n");
78. exit(1);
79. }
80. **else**
81. out = args[++i];//存储文件名
83. reout++;
84. //对于重定向操作,需要执行的操作应该只到倒数第二个字符
85. **if**(eidx == argc) eidx = i - 1;
86. }
87. }
89. //重定向输入输出的数目大于1的情况
90. **if**(rein > 1){
91. printf("Too many redirection input files\n");
92. }
93. **else** **if**(reout > 1){
94. printf("Too many redirection output files.\n");
95. }
97. //重定向输入的目标文件不存在
98. **else** **if**(rein == 1){
99. **FILE** \*fp = fopen(in, "r");
100. **if**(fp == NULL)
101. printf("The input file dose not exist.\n");
102. fclose(fp);
103. }
105. //处理指令
106. childpid = fork();
107. **if**(childpid < 0){
108. printf("Create Process fail!\n");
109. exit(1);
110. }
111. **else** **if**(childpid == 0){
112. //有重定向输入
113. **if**(rein == 1){
114. close(0);
115. **if**(open(in, O\_RDONLY) != 0){
116. fprintf(stderr, "testsh: open != 0\n");
117. exit(-1);
118. }
120. }
122. //有重定向输出
123. **if**(reout == 1){
124. close(1);
125. **if**(open(out, O\_CREAT|O\_WRONLY|O\_TRUNC, 0644) != 1){
126. fprintf(stderr, "testsh: open != 1\n");
127. exit(-1);
128. }
129. }
131. //无重定向操作
132. **char** \*tmp[256];
133. **for**(**int** i = 0; i < eidx; i++)
134. tmp[i] = args[i];
135. tmp[eidx] = NULL;
137. **if**(execvp(tmp[0], tmp) < 0)
138. printf("ERROR: invalid command.\n");
139. exit(1);
140. }
141. **else**{
142. waitpid(childpid, &status, 0);
143. }
144. }
146. //处理包含管道的指令
147. **void** dealpipe(**char** \*cmd){
148. **size\_t** pid;
149. **char** \*now;//存放多级管道指令中当前需要执行的指令的字符串
150. **char** \*next = NULL;//存储分割后的剩余指令字符串
151. **char** \*args[256];
153. //无管道指令，直接执行
154. **if**(strstr(cmd, "|") == NULL){
155. exec(cmd);
156. **return**;
157. }
159. //有管道指令,对指令开始进行分割
160. now = strtok\_r(cmd, "|", &next);
161. **if**(\*next == 0){
162. printf("There are no follow-up commands after |.\n");
163. exit(1);
164. }
166. **int** tube[2];
167. **if**(pipe(tube) < 0){
168. printf("create pipe failed.\n");
169. }
171. childpid = fork();
172. **if**(childpid < 0){
173. printf("create fork failed.\n");
174. exit(1);
175. }
176. **else** **if**(childpid == 0){
177. close(tube[0]);
178. dup2(tube[1], 1);//将标准输出重定向到管道的输出
179. exec(now);
180. }
181. **else**{
182. **int** status;
183. waitpid(childpid, &status, 0);
185. close(tube[1]);
186. dup2(tube[0], 0);//将标准输入重定向到管道的输入
188. dealpipe(next);//处理下一级指令
190. close(tube[0]);
191. }
192. }
194. **int** main(){
195. parentpid = getpid();
196. signal(SIGINT, (sighandler\_t)sigcat);
197. **char** line[256];
198. **char** \*arglist[256];
200. **while**(1){
201. memset(line, 0, **sizeof**(line));
202. getstr(line);//获取输入的命令
203. fflush(stdin);
205. **if**(\*line == '\n') **continue**;
206. **if**(\*line == '\0') exit(1);
208. **char** temp[256];
209. strcpy(temp, line);
210. parse(temp, arglist);//对输入的指令进行解析
212. **int** stdinput = dup(0);//备份标准输入输出标识符
213. **int** stdoutput = dup(1);
215. dealpipe(line);
217. dup2(stdinput, 0);  //恢复标准输入输出标识符
218. dup2(stdoutput, 1);
219. }
220. **return** 0;
221. }