South China University of Technology

《Operating System》Experiment Report

Experiment Title： Session 4: Implementation of Unix Shell

Name： 向天翼 Student ID： 201836020389

Class： 18计算机联合班 Group：

Collaborator：

Teacher： 钟竞辉

|  |
| --- |
| **Description** |
| 【Objective and Requirement】  Implement a simple command interpreter in Linux. The interpreter should:   1. support both internal and external commands, and internal commands support two (cd, exit, history); 2. able to save 10 historical commands   【Environment】  Operating System：Ubuntu 18.04.4 LTS |
| **Content** |
| 【Procedure】  First, we define a *split* function, which can split the string according to the specified delimiter and return the split result. This function uses the *strtok* function to get the next split result from the given string.    Fig.1 Implementation of *split* function  The program in this session contains following part:   1. **Load environment variable and parse input from user.**   We first use *getenv* function to obtain the environment variable of PATH from operating system and use *split* function to get all the paths in it. Then the program will output the current working path as a prompt and wait for the user's input. If the user only press Enter, the program will output the command prompt on the next line and wait for user input again. If the user enters a string and then press Enter, the program will store the string and call the *split* function again to do the parsing, which will split the character string by spaces to obtain the argument vector. If the history vector has achieve its maximum size, the program will delete the first element of it after appending the new command.    Fig.2 Load PATH and parsing input   1. **Analyze internal command and execute**   We implement three types of internal command here, including *cd*, *exit* and *history*.  If the command to be executed is *exit*, then we directly use *exit* function to terminate the shell program.  If the command is *cd*, we first get the current working directory using *getcwd* function and initialize the target path with this working directory, then analysis the second argument. If the second argument is ‘..’, which means we should change current working directory to parent directory. So, we set the character behind the last forward slash of the target path to NUL character. If the second argument is ‘.’, the target path remains the same. If the second argument starts with a forward slash, which means the user want to change to an absolute path. Then we set the target path with the second argument. Otherwise, we directly catenate the second argument to the target path. From the return result of *chcwd* function, we can know whether the operation has been successfully executed. If the return result is not equal to 0, the program will print corresponding error message to user.  If the command is *history*, the program will traverse the history vector and print all the history command in it.  The detail implementation of this part is showed in Figure 3.    Fig.3 Source code of analyzing internal command   1. **Analyze external program command and execute**   We use *fork* function to create a new process to execute the external program. When the child process is running, the main process will wait for it until it has exit. We use *wait* function here to block the main process.  To determine whether a file is existing according to its absolute path, we use *access* function here. When a user wants to execute an external program, we first look up this file in the current working directory. If not find, the program will try to find files in all PATH environment variables. If none is found, the program will print the error message. Otherwise, the program will store the first path where the file is found, then use *execve* function to execute the found program. If the return result of *execve* function is not equal to 0, the program will print corresponding error message to user.  The detail implementation of this part is showed in Figure 4.    Fig.4 Source code of analyzing external program command  The running result is showed in Figure 5. It can be seen from the running results that the program has excellently implemented the functions given by the laboratory requirements, including three internal commands and external program running commands. In addition, user can also input running arguments when running external programs. In addition, the program is also robust. It can print error messages in time for non-existent commands and non-existent directories and files, and will not exit abnormally.    Fig.5 Running result of the shell  **Appendix: Source code for this session.**  // shell.cpp  #include<stdio.h>  #include<unistd.h>  #include<stdlib.h>  #include<vector>  #include<string.h>  #include<string>  #include<sys/wait.h>  #include<errno.h>  #define MAX\_STRING\_LIST\_COUNT 65535  #define HISTORY\_MAX 10  #define MAX\_PATH\_LENGTH 1024  using namespace std;  vector<string> history;  char\*\* split(char\* str, const char\* spliters){  char\*\* results = new char\* [MAX\_STRING\_LIST\_COUNT];  int i = 0;  char\* p = strtok(str,spliters);  while(p){  results[i++] = p;  p = strtok(NULL,spliters);  }  results[i] == NULL;  return results;  }  int main(){  char\*\* paths = split(getenv("PATH"),":");  char buf[1024] = {};  char cwd\_buf[MAX\_PATH\_LENGTH] = {};  while(true)  {  memset(buf, 0, sizeof(buf));  memset(cwd\_buf,0,sizeof(cwd\_buf));  printf("%s> ",getcwd(cwd\_buf,MAX\_PATH\_LENGTH));  //printf("asdfasdf\n");    while(scanf("%[^\n]%\*c", buf) == 0){  printf("%s> ",getcwd(cwd\_buf,MAX\_PATH\_LENGTH));  while(getchar() != '\n');  }  history.push\_back(buf);  if(history.size() > HISTORY\_MAX)  history.erase(history.begin());  char\*\* argv = split(buf," ");  if(!strcmp(argv[0],"exit")) exit(0);  else if(!strcmp(argv[0],"cd")){  int result;  char \*cwd = getcwd(cwd\_buf,MAX\_PATH\_LENGTH);  if(!strcmp(argv[1],"..")){  for (int i = strlen(cwd) - 1; i > 0; i--)  {  if(cwd[i] == '/') {cwd[i+1] = '\0';break;}  }  if(cwd[0] == '\0'){cwd[0] = '/';cwd[1] = '\0';}  }  else if(!strcmp(argv[1],".")){ /\*do nothing\*/ }  else if(argv[1][0] == '/'){ cwd = argv[1]; }  else{  cwd = strcat(strcat(cwd,"/"),argv[1]);  }  result = chdir(cwd);  if(result!=0){  //printf("return code: %d\n",result);  //printf("error code:%d\n",errno);  if(errno == ENOENT){  printf("No such directory: %s\n",argv[1]);  }  else if(errno == EPERM){  printf("Permission denied: %s\n",argv[1]);  }  else{  printf("Unknown error with errno=%d for directory: %s\n",errno,argv[1]);  }  delete[] argv;  }  }  else if(!strcmp(argv[0],"history")){  for (int i = 0; i < history.size(); i++)  {  printf("%d\t%s\n",i+1,history[i].c\_str());  }  }  else{  if(fork() == 0){  bool flag = false;  string path\_prefix;  string wd = getcwd(cwd\_buf,MAX\_PATH\_LENGTH);  if(!access((wd + "/" + argv[0]).c\_str(),F\_OK)){  path\_prefix = wd + "/";  flag = true;  }else{  for (int i = 0; paths[i] != NULL; i++)  {  string i\_path = paths[i];  path\_prefix = i\_path + "/";  if(!access((path\_prefix+argv[0]).c\_str(),F\_OK)){flag = true; break;}  }  }  if(flag){  int result = execve((path\_prefix+argv[0]).c\_str(),argv,NULL);  if(result!=0){  if(errno == ENOENT){  printf("No such file: %s\n",argv[0]);  }  else if(errno == EPERM){  printf("Permission denied: %s\n",argv[0]);  }  else{  printf("Unknown error with errno=%d for file: %s\n",errno,argv[0]);  }  }  }else{  printf("No such file: %s\n",argv[0]);  }  delete[] argv;  exit(1);  }else{  wait(NULL);  //printf("Child process has exited!\n");  }  }  }  } |
| **Conclusion** |
| From this lab session I ‘ve learned the how to implement a simple Unix shell using system call by C++ in Linux operating system. During this lab session, due to my lack of experience in the use of C-related functions, some error like memory leaking occurred several times, but they were all resolved in the end. This process meaningfully enhances my understanding of the working principle and usage of C-related functions and system calls for process operation in issues related to Unix shell in the Linux system. |
| **Teacher’s Comments and Score** |
| Comment：  Score：           Signature：                                                 Date： |