South China University of Technology

《Operating System》Experiment Report

Experiment Title： Session 4: Implementation of Unix Shell

Name： 谭演锋 Student ID： 202130100456

Class： 计算机全英联合班 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);  2) able to save 10 historical commands  The following system calls can be used to implement the interpreter:  1) fork() —create a new process;  2) execve() —Load a new program and execute it;  3) wait() —waiting for child process to exist  4) chdir() —change the working direction of a process.  【Environment】  Operating System：CentOS7; |
| **Content** |
| 【Procedure】  In order to implement the command interpreter. Let us first define some global variables, the constant HISTORY\_SIZE =10 and the vector history means we store 10 history commands. The string path is used to store the changed path. The char array updatePath is used to show the current working directory.    Fig.1 Global variable  First, in order to execute all the command, we define a **splitString** function, which can split the input string into individual tokens (substrings) based on whitespace characters (spaces, tabs, etc.).  **std::istringstream**: This class is part of the <sstream> header and is used for input stream operations on strings. It allows treating a string as if it were a stream.  **while (iss >> token)** extracts tokens from the input stream (iss) separated by whitespace. Then adds each extracted token to the end of the tokens vectors and return.    Fig.2 Implementation of splitString function  **Function to execute internal commands**  First, the function takes a constant reference to a std::string named command as a parameter.  **Change Directory Command ("cd"):**  If the command is “cd”, the function attempts to change the current working directory to the one specified by the variable path by using the function. The function **chdir(path.c\_str())** attempts to change the current working directory to the path specified by path. **path.c\_str()** converts the std::string variable path to a C-style string (const char\*) using the c\_str() member function. The **chdir** function is a POSIX function that changes the current working directory. It takes a path as an argument and returns 0 on success, and -1 on failure.  If the chdir operation fails, it prints an error message using perror("cd").  If the operation succeeds, it updates the updatedPath variable with the current working directory by using the **getcwd(updatedPath, sizeof(updatedPath))**.The **getcwd** function is used to get the current working directory. It takes two arguments: a buffer to store the current working directory path and the size of that buffer.  **Exit Command ("exit"):**  If the command is “exit”, the program exits with a status code of 0 by calling the function exit**(0)**  **Display Command History ("history"):**  If the command is "history," it displays the command history stored in the history vector.  It prints each command along with its index in the history by using a for loop.    Fig.3 Function to execute Internal Command  **Function to execute external commands**  The function takes a constant reference to a vector of string named args as a parameter.  First, We use **fork** function to create a new process to execute the external program.  **pid > 0**, If the process is the main process, the main process will wait for it until it has exit. We use **wait(nullptr)** here to block the main process. The **wait** function is part of the POSIX standard and is typically used to wait for a child process to finish execution.nullptr argument means the function will not store the child process's exit status.  If **pid < 0**, the fork returns a value less than 0, it indicates an error, and **perror("fork")** is used to print an error message.  **pid == 0** If it the child process, it run the command. First a dynamic 2 dimension array of strings (char\*\* argv) is created to store the command-line arguments for the external command with the size of. Then we use a loop to convert each std::string in the args vector to a string and stores it in argv. The last element of argv is set to nullptr to indicate the end of the argument list for functions **execvp**. Then the **execvp(argv[0], argv)** attempts to replace the current process image with a new one specified by the external command. The first argument (argv[0]) is the path to the command to be executed.The second argument (argv) is an array of pointers to null-terminated strings representing the command-line arguments. If **execvp** returns -1, it indicates an error during the execution. Finally, call the exit function to exit the child process.    Fig.4 Function to execute External Command  **Main function**  In the main function, we use the **std::getline(std::cin, input)** to get a line input as a command. Then add the input to the history vector **history.push\_back(input)**. If the size is larger than 10, erase the first on in the vector by using the **history.erase(history.begin())**.  Call the **splitString** funcion to split the input into tokens and judge whether it is empty. If it is empty **continue**. Else judge the command to find out it is internal of external to call function **executeInternalCommand** or **executeExternalCommand.** If the command is **cd**, we should copy the path token to the variable path.    Fig.5 Details implementation of main function  The running result is showed in Figure 6. 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. We call the internal command **cd c**hange the working path. Then call **ls** and **ls -l for** showing the file in the working directory, which are external command**.** Finally, we call **history** and **exit to** implement the internal command.    Fig.6 Running result of the shell |
| **Conclusion** |
| In this experiment, I first met problem of reading and dealing with a line of command with white space. Then I met problem of getting current working path. And I finally met the problem of executing the external commands.  From the session 4, I have learned about how to implement a simple command interpreter in Linux by c++. After the session 4, I learn how a OS command interpreter commands to be made and run. During the session, I learn how to split the string by using the **istringstream** for command. Then after finishing the internal execution, I learn how to using the **chdir** to change work path andthe **getcwd** to get the current work path. I also using the knowledge learning before like the **exit** and the using of vector in the previous session. In the external command function, I also use the previous learning knowledge the using the function **fork** to get a child process and the **execvp** function to call a process image to run the command. Also we use the **wait** function to block the process. Finally, after using such functions that we successfully accomplishing the **MyShell**.  Form this session, I have a deeply understanding of how the terminal in our computer OS work, it is just a application which get the parameter to run its function or call other application by using the execvp function. The terminal is no longer a obscure thing for me.  The experiment most impressed me that this experiment 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. It impresses me how to implement a operation systems. |
| **Teacher’s Comments and Score** |
| Comment：  Score：           Signature：                                                 Date： |