Xiaoshi Wang & Yuchen Sun

February 23, 2018

CS 406

Professor Ordille

Project 1: Write a Shell

Introduction

The goal of this project is to write a mini-shell in C. Here are the requirements we need to achieve in our shell program.

1. The user can set or remove environment variables. And all processes created by the shell should inherit all the shell's current environment variables.
2. Commands in pipelines should be supported.
3. If the command line is not a built-in command we define, we regard that command as a job. Note that command ending with an ampersand runs background. Otherwise, it runs frontgroud. The user can easily change the status of a given job by some built-in commands, or sending signals to our shell.
4. Signals SIGINT, SIGTSTP and SIGCHLD handle appropriately in our shell.
5. The shell has some built-in commands such as quit, jobs, and stats. For stats,

Approach

- **Shell structure**

  We use the structure in Figure 8.23 in the textbook as our basic structure of our shell. The structure consists a main method, an eval method, a built-in method and a parseline method. The main method detects the input of users continuously. When it gets an input command, it passes it to the evaluation method. The evaluation method first passes the received command to the parseline method to break it into parameter pieces holding by an argv array. Argv[0] will always be a command name and the array item after argv[0] are the input variables values. After getting the command array back, the evaluation method divides commands into different categories and implements

commands in each category separately. Built-in is one of the command category, The built-in method is used to determine whether a commands is a built-in one according to their names, and then implement the different built-in commands.

- **Set and call environmental variable**

  To set a variable by “v1=variable”, we first find the position of “=” in the command line. Then we take the substring before “=” as the name of environmental variable and set its value to the substring after “=”. If there is nothing after “=”, we remove the existed variable with the mentioned variable name. To enable the echo method, we change the parseline method in the structure. Whenever a command parameter begins with a “$”, we recognize that it is the name of an environmental variable. Indeed, we will grab out the true value stored with the variable name, and push it, instead of the variable name, into the variable array argv. Thus, the value of the variable is sent back to and can be accessed by the evaluation method. In this section, we mainly learned to break down a C string and use the different part of it differently. We also learned to interact with the external environment by exchanging variables.

- **Pipeline**

  We first check whether a command is a pipeline or not in our eval method. Then, if the line contains at least one “|”, we proceed it in our pipeline method. The given pipeline is divided into three part: the argument before the first pipe, the middle part and the command after the last pipe. To pass result in standard out to be the standard in of the other part, we first initialize a pipe. Then, dup the standard out we will use later to one end of the pipe, and close the other side. Similarly, we dup the standard in to the corresponding side when we want to use data stored in a give pipe.

  One of the difficulties we met is in the close part. Close the side of a used pipe is fairly important because otherwise the pipe doesn’t know when it should stop. Notice that when we finish all instructions of one pipe, we should close both side of it in the parent process. But when we first write code to evaluate pipelines, we only close the pip in child process. This results invalid standard in or out for the later program.

- **Jobs (Foreground & Background)**

Jobs are implemented as a struct with several parameters. To keep tracking on all jobs that are running, we construct a job list. Then, the built-in command *jobs* would list all jobs. Our shell also have bg and fg commands.

The code for jobs is implemented along with the child signal. But we met a serious bug when we run the child signal for the first time. We worked on that for a long time, and finally find out that that was caused by a double for loop in the add job method: we use i in the place it should be j. I learned that regardless how small or simple a method is, we should do some unit tests before we go farther.

- **Signal**

  Construct There are three signals we need to modify in this shell. First, when a child process finishes its evaluation, we delete it from our job list. Since the race may happen, we block all signals during the deletion. The same problem happens in the SIGINT and SIGTSTP handles.

  The context about signals is hard to understand. Also, if you mask the set in a wrong way, or forget unmask it in some place, it is pretty hard to debug your code. For example, we did not fully understand the child signal. Thus, we write codes to delete jobs both in child handler and in int handler. Then, the job will be delete twice. We solve it by adding a if statement inside the child handler.

- **Show Stats Information**

  In this section, we are asked to create a new function for reporting the statistic information of each process. Firstly, we created a built-in method to decide what kinds of statistic information will be outputted by tracking its flags. We hold a collection of flag status as a global variable. Whenever a flag is recognized by the program, we will change the status of the flags to 0/1 to represent “not show” or “show”.

  Then, we learned to use the getrusage() method to collect the resource usage information of the program. Because the result of the getrusage() method seems to be accumulative, we run it once at the start of a new child process to collect the start condition, and another time at the end of the child process to collect the end condition. Considering the requirement of other sections (the pipeline) and the description of our project, we tend to think the problem asking us to find the resource usage of the command itself and all its children. Indeed, we use the RUSAGE_CHILDREN mode as

its parameter. At the end of each process, the flag status are checked. If a certain stats information is wanted to be shown, the difference of the informative values found by the two detection will be calculated and printed out.

Evaluation of Output

- **What we achieved?**

As a whole, our program runs pretty well. We meet most of the requirements. Our shell is able to:

  - Hold a lshprompt environment variable
  - Set environmental variables and, remove echo them
  - Support pipeline commands
  - Run jobs in process groups and print out a list of existed jobs
  - Run jobs on foreground or background according to commands, and restart foreground/background jobs on background/foreground by tracking signals
  - Terminate/suspend a job by tracking signals
  - Break a line input into argument pieces
  - Print out statistic information of processes
  - Be installed by a makefile into correct directories

- **What do the program lack?**

  As we are still unfamiliar with C and Unix Shell, we are not well-prepared and did not make everything perfect. Here are some restrictions of our program:

  - We did not fully consider the interaction of pipeline with other functions. As the original problem 8.26 does not have a pipeline, we did not implement the job struct and methods related to job for the pipeline methods. As there is also no description about the relationship between pipeline and the stats method, we just return the stats report of all commands in the pipeline as a whole. We hope we would have more time, so we can dig deeper into these detailed functions to make our program more completed.

- The Ctrl Z function can be used, but sometimes it is not sensitive. We have to press the key for many times to enable the method. Reason is still not found.
- After multiple (more than three or four) jobs have been run for several seconds, when we want to get the jobs list by command “lsh>jobs”, the shell will get stuck and cannot proceed any more commands. No persuasive reason has been found.
- If the background command runs pretty quickly, its results are sometimes printed directly after the next “lsh>”, which brings users inconvenience. This problem can be partly fixed by adding pause() method before return. However, because our sample commands are limited, we do not know the period length of pause that is the most suitable for the program. More various tests of the program are still needed.

- **What can we do to make it better?**

  To improve our program, we think firstly we should practice our C, so we can implement our ideas more efficiently. At the time, we still cannot handle all data structures and methods in C that can make our program concise and powerful. In order to create a better program, we also have to get a better understanding about shell, pipeline and other features mentioned in the assignment. We should practice by running shells, pipelines and other functions and tracking the process of them to understand their structure better. At last, because the shell is a little bit complex, the uncertainty of its performance is pretty high. However, we code we can used to test our shell is limited. If we try more test commands, we would find out more carelessness in the code and fix them up earlier. We should run a variety of test samples to enhance the performance of our program.

- **Work distribution**
  - Set variable & echo: Yuchen
  - Pipeline: Xiaoshi
  - Job functions: Xiaoshi
  - Signal: Xiaoshi
  - Stats: Yuchen