Project II: Operating Systems

1 Writing your own shell

Over the past few TPs, we covered how to use a shell program using UNIX commands. The shell is a program that interacts with the user through a terminal or takes the input from a file and executes a sequence of commands that are passed to the Operating System. In this project, the objective is to write your own shell program.

1.1 Overview

A shell program is an application that allows interacting with the computer. In a shell the user can run programs and also redirect the input to come from a file and output to come from a file. Shells also provide programming constructions such as if, for, while, functions, variables etc. Additionally, shell programs offer features such as line editing, history, file completion, wildcards, environment variable expansion, and programming constructions.
The shell implementation is divided into three parts :

— **The Parser** The Parser is the software component that reads the command line such as `ls -al` and puts it into a data structure that will store the commands to be executed. ***This has been already implemented for you for this project.***
— **The Executer** The executor will take the command table generated by the parser and for every simple command in the array it will create a new process. It will also if necessary create pipes to communicate the output of one process to the input of the next one. Additionally, it will redirect the standard input, standard output, and standard error if there are any redirections.
— **Shell subsystems** Other subsystems that complete your shell are for example, environment variables, wildcards, etc. (More on these later)

1.2 Getting started

Download the bootstrap at the following address :
`https://amritasuresh.github.io/teaching/shell-starter.tar.gz`.

You will find the basic skeleton of the C code that we will use to recode a shell. To compile the project, I suggest you use the `make` command. You will now see an executable, which you can run with the following command : `./shell`. By default, this shell can't do much. The objective is to try and complete the functionality following the steps below.

1. We first consider the base case, which corresponds to the C_PLAIN case. *Hint : think of an example of a command which once parsed returns a cmd object such as* `cmd -> type` == C_PLAIN.
   Now, to execute basic commands of this form, you will need to use the `exec` function. Have a look at `man exec` and see which one is most appropriate for you to use. Why is it the most appropriate ?

2. What is the symbol for the sequence operator in bash ? Give an example of a command where thesequence behaves differently vis-à-vis the `and` operator logic.
   Implement the C_SEQ case.

3. Implement the C_AND and the C_OR case.

4. It is possible in bash to write a command of the form :

   ```
   (cmd1 && cmd2 | cmd3 ...) 2>/dev/null
   ```

   What is the role of parentheses in the above command ? Give an example of a command which (non-trivially) uses these parentheses.
   Implement the C_VOID case.

5. What happens when you type `CTRL + C` in our shell ? Can you suggest a way to recover after typing `CTRL + C` ?

6. What happens when you enter the command

   ```
   ls > dump
   ```

   To correct this problem, I suggest you read the `man stdin` and `man dup` manual pages.
   Using all this information, implement the `apply_redirections` function then modify your implementation for the above command to behave as expected.

7. We finally have to implement the C_PIPE case, for that I suggest you to look at the manual of `man pipe`. Give an example that highlights why we can't just use `dup2` to reimplement the pipe ? Using the `pipe` function, reimplement the C_PIPE case

Note : Be sure to error check all your methods.

BONUS At this point, we have implemented a very rudimentary shell, however it is possible to extend it bymany ways. Here are some possibilities of extensions that can earn you bonus points :
— Reimplement `ls`, `cat`, or `cd` commands.
— Implement wildcard extensions : `ls * .pdf`, etc
— Implement background processes via commands `jobs`, `bg`, `fg`, etc.
— Implement environment variable extensions. Expressions of the form `$VAR` are expanded with the corresponding environment variable. Also the shell should be able to set, expand and print environment vars. For example, in our shell, to have commands like
`NAME="John"; echo $NAME`

Project Guidelines

While submitting, please be sure to send me a file of the form `NOM_PRENOM.tar.gz`, created by command `tar czf`, which unzips a folder with the name `NOM_PRENOM`. This folder should contain a `readme.pdf`, along with a folder `shell` which contains your shell code.

Please ensure your code compiles. Any code that does not compile will not be evaluated.

The `readme` should contain :
— A short description of how to compile and run your code.
— The answers to the questions posed during the shell implementation.
— Examples of commands that demonstrate how your implementation works.
— A description of the bonus implementation(s) you have incorporated.

The zipped file should be sent to `luc.chabassier@inria.fr`. The deadline for the submission is **14h00 CET, Friday, 6rd January 2023**.