CS302 · Operating System

Proj2 - Final Report

Xu, Zhang & Qing

May 30, 2021

ESH: An **E**nhanced **SH**ell on Linux System

Botian Xu (11810424) Wenhao Zhang (11812103) Bowen Qing (11812509)

Contents

1 Background

The topic we chose for the Project 2 in the course of CS302 Operating System is *proj12-shell-enhancement-on-SylixOS*[1]. Due to the fact that setting up the environment for *SylixOS* is not feasible, we decided to instead implement a shell on Linux system by C++ following the guideline given in the requirement document.

The main repository of our project is **wtd2/esh**[2] in GitHub.

2 Result Analysis

2.1 Built-in Functions

2.1.1 environment-related

In this part, we are to introduce some built-in commands in our `esh` which are related to the environment.

The first command is `env`, which can show the all the current environment variables in the `esh`. Here is an example.

[1] https://github.com/oscomp/proj12-shell-enhancement-on-SylixOS
[2] https://github.com/wtd2/esh

```
[ubuntu:~/github/esh] env
USER=ubuntu
PWD=/home/ubuntu/github/esh
HOME=/home/ubuntu
OLDPWD=/home/ubuntu/github
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/snap/bin
```

Then there are several commands to add, modify and remove environment variables, which are `setenv`, `resetenv` and `unsetenv` respectively. And we give some examples following.

```
[ubuntu:~/github/esh] setenv ESH_VERSION=0.1
[ubuntu:~/github/esh] env
USER=ubuntu
PWD=/home/ubuntu/github/esh
HOME=/home/ubuntu
OLDPWD=/home/ubuntu/github
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/snap/bin
ESH_VERSION=0.1

[ubuntu:~/github/esh] resetenv ESH_VERSION 0.11
[ubuntu:~/github/esh] env
USER=ubuntu
PWD=/home/ubuntu/github/esh
HOME=/home/ubuntu
OLDPWD=/home/ubuntu/github
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/snap/bin
ESH_VERSION=0.11

[ubuntu:~/github/esh] unsetenv ESH_VERSION
[ubuntu:~/github/esh] env
USER=ubuntu
PWD=/home/ubuntu/github/esh
HOME=/home/ubuntu
OLDPWD=/home/ubuntu/github
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/snap/bin
```

2.1.2 `cd`

We have implemented `cd` function to change our current working directory. The following parameter can be `".."`, `"."`, `"-"`, `" "`, `relative path`, `absolute path`. after changed current directory, it will also update the environment variable `PWD` and `OLDPWD`. And we give some examples following.

```
[ubuntu:~/esh] cd ..
[ubuntu:~] cd -
[ubuntu:~/esh] cd ~
[ubuntu:~] cd /home
[ubuntu:/home] cd ubuntu/esh/
[ubuntu:~/esh] cd .
[ubuntu:~/esh]
```

2.1.3 `history`

We have implemented a built-in function `history` to look up the previous commands user inputted before. by just providing a parameter called `offset`, we can obtain the latest commands and show then in ascending

order with offset. When the require number reached the maximum of records, it will show the maximum available. And we give some examples following.

```
[ubuntu:~] history 3
history 3
cd /home/ubuntu
cd ./github

[ubuntu:~] history 5
history 5
history 3
cd /home/ubuntu
cd ./github
cd ~

[ubuntu:~] history 1000
history 1000
history 5
history 3
...(another 9 previous commands)..
ls
exit
[ubuntu:~]
```

2.1.4 `exit`

Our esh can be shutdown by inputting command `exit` or press keyboard "Ctrl + D". Then it will exit and return to our original shell. And we give some examples following.

```
[ubuntu:~] exit
exit
```

2.2 Pipe & Redirection

2.2.1 pipe

We can put symbol `"|"` between two commands and let the first command's output become the second command's input. Then third one, fourth one... And we give some examples following.

```
[ubuntu:~/github/esh] ls
build CMakeLists.txt esh inc Makefile README.md src
[ubuntu:~/github/esh] ls -a -l | grep src
drwxrwxr-x 6 ubuntu ubuntu 4096 May 28 19:53 src
[ubuntu:~/github/esh] echo "Hello esh!" | cat | cat | cat
Hello esh!
```

2.2.2 redirection

We can use symbol ">" to redirect the standard output to a file, "»" to append to the end of this file. "2>" to redirect the standard error to a file. Use symbol "<" to redirect the content of a file as the standard input of a command. And we give some examples following.

```
[ubuntu:~/esh] cat </etc/hostname
wtd2-vlab1
[ubuntu:~/esh] wc -l < users
8
```

```
[ubuntu:~/esh] eoch
esh: No such file or directory
[ubuntu:~/esh] eoch 2> err.txt
[ubuntu:~/esh] cat err.txt
esh: No such file or directory
```

2.2.3 separators

We can set symbol ";" between multiple commands, to execute them separately. And we give some examples following.

```
[ubuntu:~/esh/build] ls ; ls -l
CMakeCache.txt CMakeFiles cmake_install.cmake commands compile_commands.json deadloop.c
     esh Makefile out
total 476
-rw-rw-r-- 1 ubuntu ubuntu 14215 May 29 12:08 CMakeCache.txt
drwxrwxr-x 5 ubuntu ubuntu 4096 May 29 16:05 CMakeFiles
-rw-rw-r-- 1 ubuntu ubuntu 1626 May 29 12:06 cmake_install.cmake
-rw-rw-r-- 1 ubuntu ubuntu  79 May 29 16:42 commands
-rw-rw-r-- 1 ubuntu ubuntu 2885 May 29 15:28 compile_commands.json
-rw-rw-r-- 1 ubuntu ubuntu  43 May 29 16:22 deadloop.c
-rwxrwxr-x 1 ubuntu ubuntu 429232 May 29 15:28 esh
-rw-rw-r-- 1 ubuntu ubuntu 14824 May 29 15:28 Makefile
-rw-r--r-- 1 ubuntu ubuntu 624 May 29 16:42 out
```

2.3 Auto-completion

In our esh, we support the auto-completion for both commands and filenames (or name of directory) by press tab, and we also have color theme to make it highlighted. To have a better understanding of the color themes we give examples by an image following.

2.4 Other features

In the esh we also implement some features which are not mentioned in our proposal (while some of them have been shown in the previous sections). To be specific, in this part we will introduce some syntax supported in esh. In the esh it can support and handle backslash (\), environment variables ($), single quote and double quote occurring in the statement. Here are some examples of these syntaxes.

```
[ubuntu:~] cd es\
> h
[ubuntu:~/esh] echo $HOME
/home/ubuntu
[ubuntu:~/esh] echo $HOME $PWD
/home/ubuntu /home/ubuntu/esh
[ubuntu:~/esh] echo '$HOME'
```

```
$HOME
[ubuntu:~/esh] echo "$HOME"
/home/ubuntu
[ubuntu:~/esh] echo "long name with space"
long name with space
```

3 Implementation

We design our shell to be composed of three main components:

1. an interaction unit responsible for displaying prompt, reading and parsing user input;
2. an execution unit that creates subprocesses to execute commands and handles pipe-lining and redirection;
3. a state unit which holds a data structure storing the running state (e.g. environment variables) of the shell.

3.1 Interaction

3.1.1 Prompt

The shell loop repeatedly prints the prompt string which shows the current user and working directory and waits for user input. if multi-line input is indicated by a "\" at the end of the line, it instead prints a "> " at the beginning of each of the following lines until the user finishes input.

3.1.2 Syntax

The input lines are concatenated into a single line which is then tokenized and parsed into commands according to the following rules. These are done using the string processing libraries in C and C++.

1. Words. A word is a single token that appears either between two white spaces (including tabs and blank lines) or two quotes (can be single or double quotes). This is essential for supporting arguments containing white spaces.
2. Command separators. In our shell we accept 3 kinds of separators: **Semicolon ";"**, **logical AND "&&"** and **pipeline "|"**. They are processed sequentially from left to right with each occurrence indicating one creation of a command. Invalid separators cause the whole line to be aborted.
3. Redirections. We support at 3 types of redirections, namely **input ("<" and "<<")**, **output (">" and ">>")** and **error ("2>" and "2>>")**. A command can have at most one of each type. Invalid redirection symbols cause the whole line to be aborted.
4. Variable substitution. When a dollar sign ("$") appears at the beginning of a token, it triggers variable substitution if the token is not single-quoted. The following characters of the token is considered as the name of an environment variable, and the value of that variable is inserted in place of the token.
5. Tilde expansion. If the first character of a word is a "~", it triggers tilde substitution which replaces the "~" with the **HOME** variable for the current user.

3.1.3 Auto-Completion and History

We support auto-completion for both file names and commands. Completions are triggered by consecutive pressing of **tab**:

1. If the cursor is at the first word of the line, the word is considered to be a part of a command. Invoking completion will either insert a suitable match or display all the candidates if there are multiple matches.
2. Otherwise, it performs file name completion.

The possible candidates for command completion are loaded from **$PATH** when initializing the shell. Also, any non-blank input line will be added to history.

3.2 Command Execution

3.2.1 General Commands

We design the result of parsing to be a sequence of **Command** objects holding:

- **path**: the path of the executable of that command;
- **argv**: the argument vector;
- **redirections**: file paths including 3 items (**in_file**, **out_file**, **err_file**) specifying where the command will be reading from/writing to;
- **separator**: by which separator is this command split (**NULL** if no separator). If it is a **"|"**, a new pipe is open.

If a command fails, the error message dumped to **error** is printed to the current error output file descriptor.

3.2.2 Built-in Commands

we designed three function to handle the entrance of `builtin command bool is_builtin(cmd); int check(cmd, aim, real)` and `void exec_builtin(cmd, pram[])`.

- **is_builtin**: Checking the input command and comparing them with our builtin key words and return whether there are our builtin commands.
- **check**: Checking the input parameter and compare them with our aim length, if they mismatch, print stderr to inform the user.
- **exec_builtin**:if these function above passed successfully, we execute these builtin commends by calling the builtin function we introduce below.

We designed a builtin command system to handle the common commands provided by our shell.

- **env**:Loop through our shell's member variables **env** print line of all the stored environment variable in the format of **<key> = <value>**.
- **setenv**: we use reallocate()to reallocate a new variable array with larger size to store the new added variable provided by user in the format of **<key> = <value>** at the end of this array.
- **unsetenv**: :Loop through our shell's member variables **env** find the environment variable that need to be delete according the key user provided. Copy other variables to another new environment variable array, then overwrite the original one.
- **resetenv**:Loop through our shell's member variables **env** find the environment variable that need to be updated according the key user provided. replace its value with the new value user provided. Finally copy it and other variables to another new environment variable array, then overwrite the original one.
- **cd**: Check the path before executing function **chdir()**, if path contains "-", or " ", replace it with the environment variable "OLDPWD", or "HOME". Then we call system function **chdir()** to change our current working directory. Finally we use function resetenv() to update the environment variable "OLDPWD", "PWD".
- **exit**: Set the variable exit to true and the while loop will exit automatically.

3.2.3 Pipeline and Redirection

Pipelining and redirection are achieve by manipulating file descriptors (**fd**). Upon execution of each command, file descriptors are set according to the options specified. File redirections have higher priority than pipelines and would overwrite pipes when they appear together.

1. If **err_file** is provided, set the error output **fd** to it.
2. If **out_file** is provided, set the output **fd** to it. Create the file if it doesn't exist. Can output in either truncate (">") or append (">>") mode. Otherwise, if **pipeline** is used, create a new pipe with a pair of input and output **fd**s and set the output. Record that **pipeline** is used.
3. If **in_file** is provided, set the input **fd** to it. The specified file is required to exist, or an error is thrown. Otherwise, if the previous command has used pipeline, set the input **fd** to it.

By default, if no redirection or pipeline presents, the file descriptors are **stdin**, **stdout** and **stderr** at the beginning and are reset after every execution.

3.3 Coordination

3.3.1 Running Environment

Our shell accepts the environment variables passed in when it starts and create from them a new environment. By doing this, we support setting and unsetting environment variables during run time. The variables can be used with substitution signified by a "$" at the beginning of a token.

It also enables some built-in commands such as "cd -" which changes to the previous working directory.

3.3.2 Signal Handling

This part handles **SIGINT** triggered by pressing Ctrl+C by setting a handler at initialization.

- If the shell is executing a command, i.e., waiting for the subprocess to finish, it sends a **KILL** signal to the subprocess to terminate it.
- Otherwise, abort the current input and start a new line.

4 Difficulties Encountered

Here we list the three major difficulties we've encountered doing this project.

4.1 Memory management

Since we are less familiar with C/C++ programming language, manually managing memory allocation is a subtle problem for us. Implementing a shell involves a lot of string processing where the use of pointers **char*** and **char**** often needs appropriate **free** of allocated spaces. This put us always into the risk of memory leak. Also, dealing with **NULL** pointers incurs segmentation fault from time to time.

4.2 Parsing and execution logic

The challenge of this part is deciding a consistent rule about how to parse the input line into commands and how to execute them. Because the syntax is not clearly defined in many cases, it frequently runs into unexpected or unreasonable behavior that needs to be resolved specifically with care.

4.3 String processing

C/C++ doesn't provide a satisfactory functionality for string processing. Many operations such as splitting and comparing becomes tricky especially when C++ **string** and C **char*** strings are used in mix.

5 Future Direction

In our proposal, we select the task to make our shell as an interpreter (support command line scripting) as an advanced task. However, since we are not familiar with compilers and interpreters, it is hard for us

to complete this task in such a short period of time. Maybe we can implement it after we have a sufficient comprehension of compilers.

Another shortcoming of our project is that we do not focus on the memory management, thus this shell may lead to memory leak. In the future if we have enough time, it is worthwhile to implement memory management meticulously.

6 Summary

To do this project is quite meaningful for us to have a comprehensive understanding of Linux operating system. For the built-in functions part, we mainly focus on the environment variables, which is quite important to support a shell system. For the pipe and redirection part, having an understanding of file descriptors is the key to implement these functions. For the auto-completion part, we need to scan all the executions in the environment variable `PATH` as well as the built-in commands. Also we need many conditions and string operations to support the syntax for shell (pipe, redirection, separator, etc,).

To have a better teamwork, it is important to have meetings frequently and code together. Also, we are more familiar with git commands (coding in different branches) by this project.

7 Division of Labor

- **Botian Xu (11810424):** Programmer & Tester
- **Wenhao Zhang (11812103):** Designer & Programmer
- **Bowen Qing (11812509):** Programmer & Tester