UNIX Shell and History Feature

EECE 6029 Final Project Report - Project 1, Group 2

Prudhvi Shedimbi Madhulika Alugubelli Jayanth N. Dungavath

December 6, 2014

Abstract

The aim of the project is to implement UNIX shell and history feature that serves as a shell interface that accepts user commands and then executes each command in a separate process, which can be run on any Linux, UNIX, or Mac OS X system. A shell interface should give the user a prompt and when entered one, the shell should execute the command accordingly. As the separate process (child process) executes the entered command, the parent should wait for the child process to complete it's execution unless otherwise stated to run in the background by using an *&* at the end of the given command. The other aim of the project is to store the commands that have been executed till now, which would enable us to keep track of the history of commands. When *history* command is entered, the shell should display the 10 most recent commands. Shell should also support two other commands *!!* and *!N* (where *N* is an integer). *!!* should execute the most recent command from the history and *!N* should execute the N^{th} command from the history. The should also handle errors accordingly.

Contents

1 Introduction

We have developed this project using *Perl* programming language. We chose *Perl* because of it's extensive library and it's support for regular expressions. We have implemented this project exactly as per the requirement specifications. We not only implemented all the featues, but also have handled corresponding bugs associated with these features. Just like any other advanced shell, we have implemented history feature such that, it stores and remembers all the commands that have been entered till now and can be accessed between different run times as they are permanenly stored in a hidden file in the current directory. We have also implemented this to support *!!* and *!N* (where *N* is an integer). According to the problem statement, N can be any value between 1 and total number of commands in the history (not just the last 10 commands!). In our project, N value ranges from 1 to 99999 (can easily be extended to support even more!). We have thoroughly tested this project in Mac OS X only. Should give the same results in any of the Linux/Unix variants.

2 Environment

- **OS**: Mac OS X Yosemite
- **Language**: Perl v5.12.3
- **Processor**: 2.5 GHz Intel Core i5
- **Memory**: 4 GB 1333 MHz DDR3
- **Editor**: Xcode
- **Documentation**: TeXworks

3 Requirement Specifications

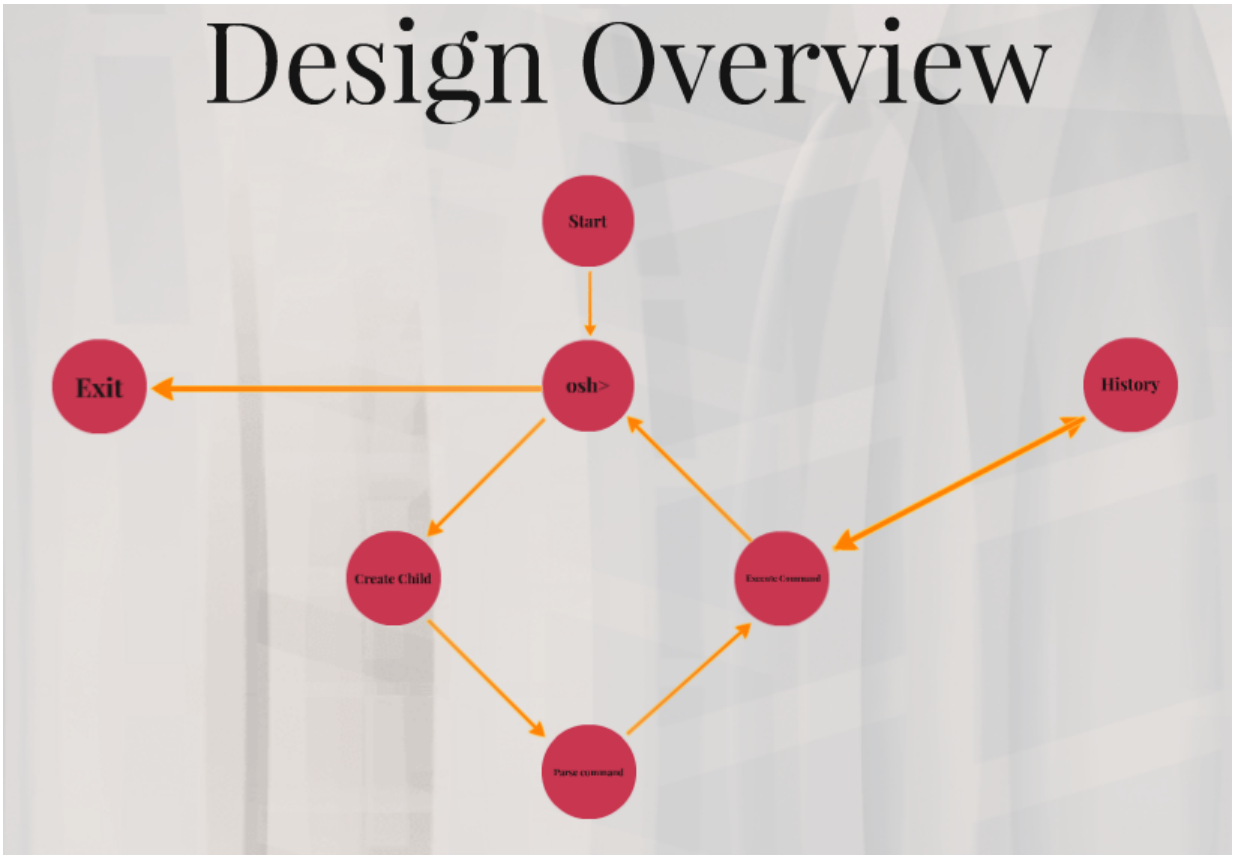
The created shell must meet the following requirements:

- Should execute basic commands with or without arguments
- Commands should be executed by a child process
- Commands should run in the background if the given command is followed by an *&*
- Provide history feature such that last 10 commands can be displayed with *history* command
- *history* should display last 10 commands from most recent to least recent in order
- *!!* should execute the most recent command in the history

- *!N* should execute the Nth command in the history
- Basic error handling

4 Design

Basic work flow of the program is as shown in the following figure.



As soon as the shell starts, it waits for the user to enter a command by displaying the command prompt *osh>*. Once user enters a command, parent process will create a child process which in turn will parse the command. Once the command is parsed, the command will be passed on to the respective code snippet to execute the command accordingly. If the entered command is valid, then the child process will store that command in the history file and will pass on the control to the parent process. Parent process will continue to display command prompt and accepts commands as long as the user enters *exit*. If the user enters *exit*, then shell program ends.

Design of each requirement specification

1. Execute basic commands:

   We have used *system* library function to execute commands which takes *list* (array in perl) as an argument and returns 0 if it is successful and -1 if it fails. In the given *list* array, first element should be the command and rest of the elements can be arguments of that command.

```
print "osh> ";
do{
    $text=<>;
    $text=~ s/^\s+|\s+$//g;
    chomp($text);
}while(!defined $text);
chomp($text);
if($text =~ /history/){
    print $FILE $text;
    print $FILE "\n";
}
@command=split(/ /,$text);
system(@command);
```

   Above code prints the prompt and reads command from the user. It will loop until some valid command(text) is entered which means to ignore *return* at command line. The last but one line tokenizes the command by using space as delimiter and stores command along with it's arguments in a list (array). In the end, *system* exectes the entered command.

2. Commands in child process:

   fork() is used to create child process and the entire processing happens in the child process only.

```
while($command[0] ne "exit") {
    my $pid=fork();
    if (!defined $pid) {
        die "Cannot fork: $!";
    }
    elsif ($pid == 0) {
```

   Above code snippet creates child process and check it's validity. Child process execution starts from the last line.

3. Background processing:

   In case of *system* library function, fork is done first and parent process waits for the child process to exit. But, *exec* library function on the other hand executes system command and never returns. So, if a user enters a command with an *&* in the end, we have used *exec* library function as shown in the following code snippet.

```
elsif($text =~ /&$/) {
    $text=~ s/&//;
    @command=split(/ /,$text);
    exec(@command);
    $text=$text," &";
    print $FILE $text;
    print $FILE "\n";
    if ($? == -1) {
        print "failed to execute: $!\n";
    }
    exit
}
```

   By using *exec* function, parent process will not wait for the child process and will run concurrently with the child. Above code checks if the command ends with an *&* and executes using *exec*.

4. *history* festure:

   We have implemented *history* feature by storing all the commands in a hidden text file in the current directory and maintaining two file pointers simultaneously to read and write to that file. One file pointer appends(writes) the commands executed at the end of the file and other pointer reads the contents of the file from bottom to top and stores the read commands in an array, which in turn can be displayed when user enters *history* command.

```
$fh = File::ReadBackwards->new('.hist') or die "can't read file: $!\n";
my $line;
my @hist;
my @last;
my $i=0;
while (defined($line = $fh->readline)){
    chomp($line);
    if( length($line)<1){
        next;
    }
    $hist[$i]=$line;
    $i++;
    #if($i==10){
    #    break;
    #}
}
```

In the above code, file pointer reads the history file from bottom to the top using the library function *ReadBackwrds* and stores all the commands in an array.

5. 10 most recent commands:

```
elsif($command[0] eq "history"){
HIST: if(-z $FILE){
        print "No commands in history!\n";
        print $FILE $text;
        print $FILE "\n";
        exit;
    }
    if($i<10){
        for(my $j=0;$j<$i;$j++){
            print $i-$j," ",$hist[$j],"\n";
        }
    }
    else{
        for(my $j=0;$j<=9;$j++){
            print $i-$j," ",$hist[$j],"\n";
        }
    }
    if($text =~ /^!!$/){
        print $FILE $last[0];
        print $FILE "\n";
        exit;
    }
    if($command[0] =~ /^!/ && $command[0] =~ /[0-9]$/){
        print $FILE $last[0];
        print $FILE "\n";
        exit;
    }
    print $FILE $text;
    print $FILE "\n";
    exit;
}
```

Above code snippet displays the 10 most recent commands from the history. If there are less than 10 commands, it prints accordingly and stores itself(*history*) into the history.

6. *!!* -> executes most recent command:
As we have already read the history file, we just need to execute the command available at index *0* in the array of history commands to execute most recent one.

```
if($command[0] =~ /^!!$/){
    if(-z $FILE){
        print "No commands in history!\n";
        exit;
    }
    $last[0]=$hist[0];
    if($hist[0]=~ /history/){
        goto HIST;
    }
    chomp($last[0]);
    system(@last);
    print $FILE $last[0];
    print $FILE "\n";
    if ($? == -1) {
        print "failed to execute in !!: $!\n";
    }
    exit;
}
```

7. *!N* -> execute Nth command:

```
elsif($command[0] =~ /^!/ && $command[0] =~ /[0-9]$/){
    my $index=0;
    switch(length($command[0])){
        case 2 {
            $command[0]=~ /(\d)/;
            if($1>=$i){
                print "Invalid reference!\n";
                exit;
            }
            $last[0]=$hist[$i-$1];
            if($last[0]=~ /history/){
                goto HIST;
            }
            chomp($last[0]);
            system(@last);
            print $FILE $last[0];
            print $FILE "\n";
            if ($? == -1) {
                print "failed to execute: $!\n";
            }
            exit;
        }
```

Above image contains a segment of code that handles single digit N value after ! and we have developed it to support 5 digit N value.

5 Issues and Challenges

We have started developing the project using *C* programming language and we had tough time handling string in C. It was also very challenging to maintain history. After developing half of the project, we have started developing it in both *C* and as well as *Perl*. We have completed both the versions successfully. We are submitting *Perl* code as we were able to implement all the requirements and also have resolved most bugs that we came accross.

6 Division of work

Initially we three of us started working on different features individually. After completing one cycle of development, we segregated the work such that Prudhvi handled parser implementation, Madhulika worked in designing the workflow & preparing the initial documentation part and I worked on history feature, other features and final documentation. We have also prepared a presentation to present it during demo, which can be found here.

7 Bibliography

[**Abraham Silberschatz, Peter Baer Galvin and Greg Gagne 2013**] Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, *"Operating System Concepts"*, Wiley 2013.

8 Final code

Attached along with this document.