Project 3—UNIX Shell and History Feature

Deadline: Monday, Oct. 30 11:59 pm.
Test your code on a virtual machine.
Submission: **Electronic** copy: submit your **code** and **screenshots** on Blackboard

This project consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process. A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt osh> and the user's next command: ls (This command displays the items in the current folder on the terminal using the UNIX ls command.)

```
osh> ls
```

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, ls), and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. However, UNIX shells typically also allow the child process to run in the background, or concurrently. To accomplish this, we add an ampersand (&) at the end of the command. Thus, if we rewrite the above command as

```
osh> ls &
```

the parent and child processes will run concurrently.

The separate child process is created using the fork() system call, and the user's command is executed using one of the system calls in the exec() family (as described in Section 3.3.1).
A C program that provides the general operations of a command-line shell. The main() function presents the prompt osh-> and outlines the steps to be taken after input from the user has been read. The main() function continually loops as long as shouldrun equals 1; when the user enters exit at the prompt, your program will set shouldrun to 0 and terminate.
This project is organized into two parts: (1) creating the child process and executing the command in the child, and (2) modifying the shell to allow a history feature.

Part I— Creating a Child Process

The first task is to modify the main() function so that a child process is forked and executes the command specified by the user. This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings. For example, if the user enters the command ps -ael at the osh> prompt, the values stored in the
args array are:

```
args[0] = "ps"
args[1] = "-ael"
args[2] = NULL
```

This args array will be passed to the execvp() function, which has the following prototype:

```
execvp(char *command, char *params[]);
```

Here, command represents the command to be performed and params stores the parameters to this command. For this project, the execvp() function should be invoked as execvp(args[0], args). Be sure to check whether the user included an & to determine whether or not the parent process is to wait for the child to exit.

Part II—Creating a History Feature

The next task is to modify the shell interface program so that it provides a ***history*** feature that allows the user to access the most recently entered commands. The user will be able to access up to 9 commands by using the feature.
The user will be able to list the command history by entering the command
history
at the osh> prompt. As an example, assume that the history consists of the commands (from most to least recent):

ps, ls -l, top, cal, who, date

The command history will output:

```
6       ps
5       ls -l
4       top
3       cal
2       who
1       date
```

Your program should support two techniques for retrieving commands from the command history:

1. When the user enters !!, the most recent command in the history is executed.
2. When the user enters a single ! followed by an integer *N*, the *N*th command in the history array is executed.

Continuing our example from above, if the user enters !!, the ps command will be performed; if the user enters !3, the command cal will be executed. Any command executed in this fashion should be echoed on the user's screen. The command should also be placed in the history buffer as the next command.

The program should also manage basic error handling. If there are no commands in the history, entering !! should result in a message "No commands in history." If there is no command corresponding to the number entered with the single !, the program should output "No such command in history."

Implementation Guideline

This project implements a simple version of Unix shell. Some code has been provided to you. You need fill in the blanks in the C file.

Part I

To implement Part I, you need work on the following functions.

1. `int setup(char inputBuffer[], char *args[],int *background)`

   Function description: The setup function will **read** in the next command line; **separate** it into distinct arguments (using blanks as delimiters), and set the args array entries to point to the beginning of what will become null-terminated, C-style strings.

   Return value: When it returns **1**, it means the user input is in the right format and the shell program will continue to create a process. Otherwise, the shell program will simply terminate.

1.1 **read** is provided for you.

```
length = read(STDIN_FILENO,inputBuffer,MAX_LINE);
```

read() reads up to count bytes from the given file descriptor into the buffer pointed to by buf.
It returns the number of characters read, and advances the file position by that many bytes, or returns −1 if an error occurred. Check and use this return value. It is otherwise impossible to safely use the buffer contents.
read() blocks waiting for input: it does not return until there is data available for reading. When reading from standard input, read() returns when the user types enter or CTRL-D. These situations can be distinguished by examining the contents of the buffer: typing enter causes a new-line character (\n) to be written at the end of the line, whereas typing CTRL-D does not cause any special character to appear in the buffer. If a user types CTRL-D on a line by itself, read will return 0, indicating that no more data is available to be read—a condition called end of file. In this case, your shell should exit.

The code for "read" from user input is provided for you. Basically, everything typed in from keyboard will be stored in '**inputBuffer**'.

1.2 **parse** the contents of **inputBuffer**.

a. As listed in the C file, to parse the inputBuffer, your code needs identify ' ' and '\t' as the argument separators. Your code needs identify '\n', which means the final character of the input. Since ' ' and '\t' are separators, you want to use '\0' to replace them to make sure that's the end of a string, which becomes a real separate string.
b. For all other characters (**default** in the C file), you need identify the first character after ' ' or '\t'. It is the first character of a string. We care about this character because we need the address of this character to fill in the array **args[x],** which holds the address of the ***x-th*** argument. You may consider using a flag to mark the beginning character of each argument.
c. In the **default** case, you also need identify '**&**', which you will use it to set up '***backgroud'**.

1.3 return

Now it is time to return, before you return, make sure that '&' is not written into the argument list **args.**

2. **int** main()
   Function description: This is your shell program. Your shell program will always wait for reading from the standard input (the keyboard). In the implementation, the shell program will call the "**setup**" function in a while loop.

   shouldrun = **setup**(inputBuffer,args,&background);

   2.1 If the user input is '**exit'**, the user wants to stop using the shell. Simply return **0**.
   Use the function 'strncmp' to compare **inputBuffer** and 'exit'
   2.2 The return value **shouldrun** from "**setup"** works like a flag.
   If **shouldrun** is true, the shell will continue to create a child process and call **execvp** to run the command from user. Otherwise, shell will terminate.

Part II

To implement Part II, you need work on the following functions.

1. `addtohistory`

```
void addtohistory(char inputBuffer[]);
```

Function description: This function maintains the history of commands. In the implementation, this function will update three global values:

```
char history[MAX_COMMANDS][MAX_LINE];
char display_history[MAX_COMMANDS][MAX_LINE];

int command_count = 0;
```

1.1 update the array **history** and **`command_count`**
You need call the function **strcpy**(str1, str2) to copy the string **inputBuffer** into the array **history.** You also need to be careful with the index of **history**. It won't be larger than **MAX_COMMANDS**. The **history** buffer will only store the most recent 9 commands.

1.2 update the array **display_history**
You also need update another array: **display_history**. Since your shell need call **printf** to print out the history of commands**.** You don't want to keep the characters like '\n' and '\0' in **display-history.**

2. **int** setup(char inputBuffer[], char *args[],int *background)
We revisit this function to deal with **history.**

2.1 check if they are using **history**
If the user input starts with **'!'** (inputBuffer[0]=='!'), it means user want to use the history commands in the array **history.**
You need use the corresponding command from the array **history** to replace the **inputBuffer.**

a. You want to check whether there is no history. In this case, print out an error message and **return**. What return value you want to put here? If **return 1,** it means shell will continue to wait for input from user. If you **return 0**, it will cause your shell program to terminate.

b. Your program will continue to check the second character. If user input is **'!!',** (inputBuffer[1]=='!'), the most recent history will be used to replace **inputBuffer.** Don't forget to update **length** after you update **inputBuffer.**

c. User input asks for the ***nth*** command, for example '!5'. You need first check whether the second character of **inputBuffer** is a digit. Call the function **isdigit**(inputBuffer[1]), which returns true if it is a digit. Then continue to replace **inputBuffer** and update **length**, **if the digit is valid**.

2.2 Add the command to history

Call the function addtohistory(inputBuffer);
2.2 must be after 2.1

3. int main(**void**)

We revisit this function to support a new command "**history"**. This project requires your shell to print out all stored commands when a user inputs 'history' as a command. You will **printf** every item in the array **display_history**. After that, do you still need create a process or call execvp? If no, how to skip the child process part and continue your shell program with the **while** loop.