UNIX Shell & History Feature

1 IDENTIFYING THE GOALS

To design a C program to function as the shell interface, it will accept user inputs and execute them as separate processes. The main project is organized into two critical parts:

1. Creating the child process and executing the command in the child (command specified by the user)
2. Modifying the shell to allow a history feature

2 DESIGNING THE SHELL

As per our design, the shell invokes the following features:

1. **History buffer [hist]** : A queue to store the commands called/passed to the shell
2. **Argument Parser [args_parse]** : Tokenize the command (and arguments) passed to the shell
3. **History Feature** : Functions designed to perform all history features as per the specifications
4. **Creating a child process and executing the command** : To invoke execvp() as per the specifications provided

2.1 HISTORY BUFFER

For our purposes we have defined the history storage data structure as a circular queue of a fixed size (10). This allows the buffer to be defined as an array of char* pointers, and to track the content and index of the buffer contents using a variable 'current'. By defining the storage buffer as a circular queue, memory waste is nullified. It is a structure that allows data to be passed from one process to another whilst making the most efficient use of memory.

2.2 ARGUMENT PARSER

The argument parser has been designed in such a way that it accepts the buffer in a single pass and iterates over it to until it can identify each parameter as an individual parameter and store the same to $args[\,]$. $args$ is an array of pointers of type char * that stores the command (and the arguments) that were passed to the shell. Example:

$$osh > \ ps - ael$$

The $args$ array is updated to:

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

The parser has been defined such that it identifies a space, a tab, or an end of line character as the termination of a token. Additionally, it has an built-in capability of identifying whether a '&' was passed

as a command to the shell and sets the status of the same as true or false (return call of the function, stored to $childFlag$).

2.3 HISTORY FEATURE

The history feature has been declared to support three distinct features:

1. history : the shell will be display the most recent (maximum) 10 commands that were entered in the shell
2. history !! : the most recent command in the history is executed
3. history !n : the n^{th} command in the history is executed

The $history()$ function takes in two parameters, the history buffer and the value of the current index position of the history buffer (stored as current). It has been defined such that it iterates over the queue and displays the last commands (up to a maximum of 10 commands) that were entered in the shell.

For our consideration, the shell has been designed such that it outputs the commands in the order in which they were entered into the prompt. The indexing of the prompt has been designed to make it more intuitive, such that it can hold values in the range [1, 10] with 1 being the command that was passed the least recently. Example:

Assume the history consists of the commands (from most to least recent):

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

The command history will output:

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

The $history\_recall()$ is passed two parameters, the history buffer and the current index of the history buffer (current). It is designed such that it can handle both the '!!' and the '!n' calls. To do so, it iterates over all the contents of the history buffer from 1 to current, until the index is equal to one less than current (current -1). This has been designed such that:

- $history$!! : current stores the index of the buffer where "$history$!!" resides. To execute the most recent command, the command at (current – 1) has to be executed. Hence, we pass the value of current as is to $history\_recall()$. Error handling has been implemented such that if the buffer is empty, an error will be displayed. Additionally, if two "$history$!! " calls are made in a row, the first of the two will be executed and the second will fail to execute; and will throw an error to the shell. However, in any case the history buffer will be updated automatically, as expected.
- $history$!n : current stores the value of 'n' being passed. As the buffer intuitively runs from [0,9], the goal is to execute the $(n - 1)$ command. Hence, we pass n as is to$history_recall()$. As before, error handling has been incorporated with boundary condition checks being insured such that n can have values in the range of [1,10] only, anything else will throw an error to the shell.

2.4 EXECUTING THE COMMANDS

Once a command was passed to the shell, it gets updated to the buffer. Following that the argument parser ($args\_parse$) tokenizes the inputs and updates $args$ as described earlier. By means of conditional checks, it was determined if a command being passed to the shell was an exit command, a history command or a shell command. Once the nature of $args[0]$ is determined, a $forkCommand()$ was made. $forkCommand()$ is designed such that it does a $fork()$ system call, if the process is in the child thread, the $args$ array will be passed to the $execvp()$ call. Additionally, provision has been made to return an error to the shell if the command entered in the shell is not a recognized one. To account for the '&', the value of $childFlag$ is used. If $childFlag$ is true, this particular child thread will be allowed to run in the background (or concurrently); thus allowing the parent and the child processes to run concurrently.

3 CHALLENGES

The shell has been designed such that it takes into consideration all the requirements that have been outlined in the project description, and aims to incorporate additional features that have not been mentioned in the scope.

- Tokenizing the arguments passed to the shell and updating the same to the $args$ array was a key feature of the shell design, a feature which has was invoked multiple times during execution.
- The use of an appended '&' to the end of a command was a design challenge that we had to overcome. To make provisions for identifying the same, the $args\_parse()$ was modified to return a status flag, which was stored to $childFlag$. If $childFlag$ was true, then an '&' was present and the same information was relayed to the $forkCommand()$. Once a command with a '&' at the end of it was passed, provisions were made for the parent process to wait for the child to execute simultaneously, thereby allowing both the child and parent process to execute concurrently.
- The display of the history command did not seem intuitive as per the instructions provided, as such, it was modified such that the indexing was more appropriate and easily discernable to the user.
- Implementing the history feature was a critical design feature that required several modifications before it was successfully implemented. Considerations had to be made to solve unresolvable errors, such as the case when two "$history$!! " commands were passed in series, it was resolved as explained mentioned above. Additionally, it was important to design the history feature as a minimalistic feature to reduce complexity while optimizing performance during runtime. Overall, the history feature checks for boundary conditions, does a history lookup and a parsing, status checking, and executes the command stored there. Ultimately, the history feature integrated all the features used in the program.
- Our circular buffer does not track the number of commands stored; it was sufficient to store a NULL as a blank value. However, for the $history$! n command to function, it must be known if n is in range, as there could be from 0 to 10 commands stored there. Thus, a variable had to be defined to track the number of commands stored in the history buffer.

4 TESTING

For demonstration purposes, two executable C++ programs have been defined namely $timer.cpp$ and $sieve.cpp$. Timer does a system timer call and counts up the seconds from execution, from 1 to 100. Sieve computes the *Sieve of Erasthonese* for the first 100 integers.

The key aspect of execution with a '&' argument is that the process is launched and concurrently outputs to cout while the host program is using both cin and cout streams, which both write to the screen. A program that periodically outputs to the screen (such as timer) illustrates this effect much more clearly than a program that creates a single surge of output, such as cat. We look forward to our dramatic live demonstration in class.

The Sieve of Erasthonese is a program that creates burst output. Executing ./sieve while ./timer & was running demonstrated that the cout resource is shared by a locking mechanism. Sieve completes its output and closes before timer is allowed to write anything to the display.

We wrote approximately 50 test commands into a test script, which were used to identify bugs and test boundary conditions. A major bug found was that $history$!! and $history$! n commands were displayed incorrectly and inconsistently in the history. It was discovered that parsing algorithm used on a command stored in the history buffer overwrote the contents in the buffer. Therefore, the history buffer was copied to a temporary variable prior to parsing.

This testing also revealed that several boundary cases involving chained history commands required behavior definition, or infinite loops could result.