UNIX SYSTEM PROGRAMMING LABORATORY (UE15CS355)

TEAM MEMBERS

Nitish J Makam	01FB15ECS197
Parashara R	01FB15ECS202
Pavan Yekbote	01FB15ECS205

IMPLEMENT A SHELL

This project is about implementing our own shell.
Before we dive into the details of implementation, it is important that we know what a shell is.
A shell is a command line interpreter that reads user input and executes commands.
It provides a command line user interface.
It hides the details of the underlying operating system and manages the technical details of the operating system kernel interface, which is the lowest level, or the innermost component of most operating systems.

The shell does four jobs repeatedly :

1. Display a prompt
2. Read a command
3. Process the given command
4. Execute the command

This project was divided into 3 phases :

Phase 1 : Implement the basic shell

Phase 2 : Add more functionality to the basic shell namely -

1. Piping and I/O Redirection
2. History
3. Editor
4. Aliasing

Phase 3 : Implement 2 custom features on our own.

We now begin to describe our approach and a few details of how we implemented this project.

PHASE 1

As we have seen, the first phase of the project was to implement a basic shell. Hence, we had to continuously display a prompt and process the input and then execute the appropriate actions.
Hence, the first step was to write a function that split the input into its constituent parts that would make it easy to execute the appropriate functions and pass the appropriate arguments to them.

- ➔ `char ** split(char * input)`
  This function takes a character array as input and splits the string using space character as the delimiter. It returns an array of character pointers.

The array of pointers returned by the above function is passed to a function called `executor`.

- ➔ `int executor(char **args)`
  This function performs a check on the array of pointers it receives as argument and in turn calls the `execChild` function. Since change directory function is not performed by execvp we had to write our own function that performed this task. The check performed serves this purpose along with a few other things as we'll see later.
  It returns an integer that signifies success or failure.

- ➔ `int execChild(char **args)`
  This function is usually called by the executor function. The executor function passes the arguments it receives, as arguments to this function.
  This function is the heart of the shell. This function performs the most important task of forking a new process, exec - ing this new process by calling execvp with the arguments and performing a wait on this process so that it knows what the exit status of this child process was. It makes sure that the task was performed successfully.
  It returns an integer that signifies success or failure.

This essentially completes Phase 1.

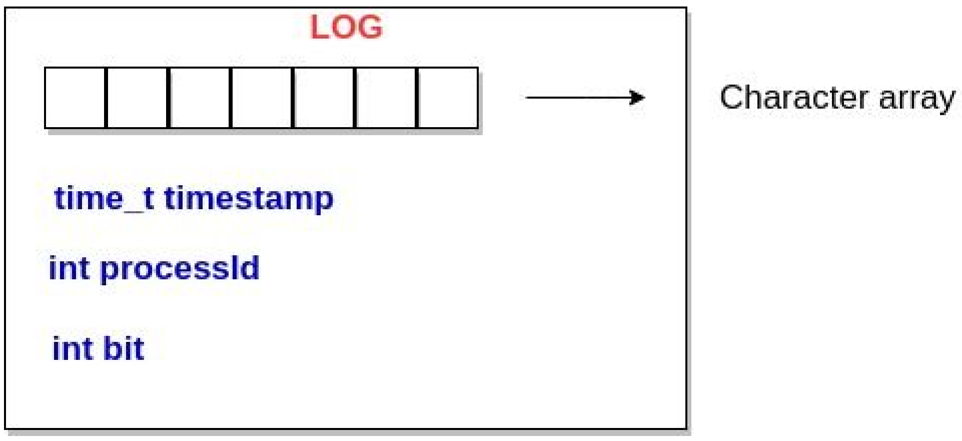
PHASE 2

Now, coming to the second phase. We had to implement 4 additional functionalities to the shell.

- **Piping and I/O Redirection**
  To implement piping we first scan the entire input to check if the pipe '|' character is present in the input. If it is present then we call the `parsePipe` function that will parse the input to retrieve the constituent parts of the input. Next, the `execArgsPiped` function is called that forks two new processes, initializes a pipe between them. It also closes the read end and write for the appropriate processes and performs an exec for these two processes. It then retrieves the exit status of the two child processes to ensure that this task was error free. It then returns an integer that signifies success or failure and returns control back.

  For I/O redirection, we included statements in the `execChild` function that scans the input for '<' and '>' characters. If these characters were found, we called the dup2 function that redirected the input or output or both depending on the characters that were found in the input. The rest was as it was in the first phase.

- **History Feature**
  To implement this, every time a command was entered, it was recorded in array of structures. The structure had the following format.



It's members included a character array that stored the entire command. The second member of the structure recorded the timestamp when the command was executed. The third member stored the processID of the process that executed the command. The last member was an integer that recorded if this structure was free or not.

- `void addLogRecord(char *command, time_t timestamp, int processId)`
  This function took as arguments the command, timestamp and the processID and included this into the table. It does not return anything.
  The number of recent commands recorded was capped at 25.

- `void printRecentCommands()`
  This function printed the recent 25 commands to the output. It scans through the entire table, checks if the structure was used and prints them. There is a global variable that stores the index so that it knows from where to print. It's return value is void.

➔ Editor
Editor can be invoked using the "sedit" command upon which the editor would display the following options :

- R : open a file
- E : edit the currently opened file
- X : close the currently opened file
- Q : quit while discarding any unsaved changes

Upon opening a file using 'R' option, contents of the specified file is displayed along with the following options :

- C : edit the current line
- P : move one line up
- N : move one line down
- D : display the current line
- V : display the contents of the current buffer
- A : add a line immediately after the line at which you are navigating
- S : save changed and exit to the menu
- X : exit without saving changes
- H : show the list of commands

Editor has been implemented by maintaining doubly linked list having a character array, integer index and pointers to the next and previous nodes. Functions used to implement the same :

- `void editcommands(void)` **:** function to print the list of editor commands
- `void addline(struct dll *temp)` **:** function to add a new line via input from the user

- `void inp(void)`: function to take input command provided by the user
- `void printlist(void)`: function to print all the lines stored in the buffer
- `void closer(void)`: function to close the file opened for editing
- `void edit(void)`: function used for editing purposes which inturn calls `editnode(void)` function to edit a specified line
- `void addnode(char t[],struct dll *q)`: function to add a new node after a node q
- `void delnode(struct dll *p)`: function to delete a node
- `void clearlist(void)`: function to clear the list
- `void editnode(struct dll *p)`: function to edit a line
- `void save(void)`: function to save the file

➔ Aliasing

Aliasing has been implemented by maintaining an array of structures where each instance of the structure has a string corresponding to the aliased command as well as the original command being aliased.

```
typedef struct alias
{
        char aliascommand[10];
        char originalcommand[10];
}alias;
```

Usage : alias <existing command> <new command>
Upon using the alias command, void logalias(char * original, char * alias) is called.
Each time a command is called, it is first checked to see if it is an aliased command, if so, the corresponding original command is returned using the and executed.

- void logalias(char* original, char* alias)
  This function takes a character array ‘original’ and a character array ‘alias’ as arguments and stores them within an instance of the alias structure and is invoked when the alias command is called.

- char *getOriginalCommand(char *alias)
  This function takes a character array 'alias' and traverses the array of structures to see if any instance's 'aliascommand' member matches the argument. If so, it returns the corresponding instance's 'originalcommand' else it returns NULL. It is invoked just before any command is executed.

➔ WildCard Feature
To implement this feature we have to use a utility function "**int wildcardcmp(char *pattern,char *text)**" which returns 1 if the given text matches the regex pattern and returns 0 if it doesn't.This function was implemented using "dynamic programming".

The steps for this feature are:

- Get all the files and directories in the current directory using an utility function -
  `char ** getAllMatches(char ** ls,char * regex).`

- Get all the names of the files and subdirectories which match the given pattern in the current directory using the function
  `char ** getAllMatches(char ** ls,char * regex).`

- Store all the matches from the current directory in a list and replace the original command's tokens such as **ls a*** with all the matched names which it would modify the tokens of the original command like so
  ls a1.txt a2.txt a3.txt.
  For achieving this we use the function
  `char ** executeWildCard(char ** tokens)`

PHASE 3

For this phase we decided to implement 2 features:

- ➔ sgown feature

This feature given a directory and a keyword to search it prints out the number of occurrences of the keyword specified, present in every file in the directory specified, recursively until it reaches all of its subdirectories.

- ➔ `void sgown(const char *name,char *searchstring):`

This function prints the output in the format specified below . It opens every file in the current directory and uses a utility function "int getOccurenceCount(char * path,char * word)" which returns the count of the keyword in this file.This function recursively traverses each subdirectory from the current directory until there are no more directories to traverse.

Usage:
sgown <directory path> <keyword>

Output:
file: <path of the file after traversing > count:<count of the keyword>

- ➔ ls -z feature

This is the second of the two features we included. It lists all the files in the current directory that are of zero size.

Usage : ls -z

Output : List of files that have zero size.

To implement this we used the opendir() function, readdir(), stat() functions.
We first get the current working directory, open the directory and read the directory for files one by one. We call the stat function on each of the files. We then check the stat structure for the size. When the size of the file is zero, we print the name of the file onto the output.