Shell Documentation
Date: 30th September 2020

- Input prompt: the shell shows the following symbols (~~#) that indicate the user to enter a command.
- Shell receives the user input character by character using fgetc in read_command() and further the keyword and its parameters are separated in this command itself and put into different arrays.
- In case a wrong command is entered, the shell shows error.
- The names of all the files that are taken as input throughout the code, in all test cases do not have any spaces in them. eg) file test.txt is not recognised anywhere in the program. Single word names of files are recognised only.
- Input Commands implemented:
    - cd: to change the current working directory.
    - pwd: to print the current working directory.
    - echo: it prints the arguments given along with it.
    - history: prints all the commands which the user has given as input to the shell.
    - exit: closes the shell program.

cd: the following cases can be managed by this command-

1. cd directory_name → if directory_name is present then it shows that the path was successfully changed. If the directory name specified is not there in the current location of the shell, then it shows error.
2. cd .. → goes back to the previous directory.
3. cd . → stays at the current directory.

You can check the working of this command using the c directory which has been made in the shell folder.

Also, the home directory for the shell is where the shell program and all the binary files are present. Hence we cannot cd to directories which are parent to the shell directory.

exit: can be executed by writing exit.
Even if the user enters some parameters with exit, the shell reads only the keyword exit and doesn't pay heed to the parameters that follow it, the shell exits anyway.
eg) exit → exits from the shell
eg) exit random_parameters → still exits from the shell

pwd: the following cases can be managed by this command-

1. pwd → shows the path of the current working directory.
2. pwd random_parameters → shows error because of the parameters.

echo: the following cases can be managed by the shell-

1. echo parameters → prints the parameters that have been entered by the user

In case when no parameters are entered, it shows a blank.
The command also doesn't recognise newline and backslash escapes.

history: can be executed by writing history.
Even if the user enters some parameters with history, the shell reads only the keyword history and doesn't pay heed to the parameters that follow it, showing the history anyway.
eg) history → shows history of inputs from the user.
eg) history random_parameters → still shows history of inputs by user

Sometimes, history also shows some garbage values.

- External commands have their own different compile-able files. On encountering a keyword for an external command, fork() is called to create a parent and a child process, the external command file is called using the execv() command which takes the file name of the binary file of the executable and executes it.

  The parent process waits for this process to finish in the child process of fork() using the waitpid command.

  Error handling of the fork() and waitpid() command has also been done in all external command cases.

  The external commands that were executed are:
  - ls
  - cat
  - date
  - rm
  - mkdir

ls: it lists the files and directories in the current directory. Flags implemented:

1. -a: it prints the hidden files in the directory as well
2. -F : it prints only the directories present with a '/' at the end of their name.

eg) ls -a → prints hidden files and normal files in shell folder
eg) ls -F → prints the directories in parent folder
eg) ls → shows all normal files and directories
eg) ls -s → shows error

I have handled cases where a wrong flag is given as input.
It also handles errors with stat().
ls works only in the home directory of the shell, if we cd into some sub folder where the binary file of ls is not present, then ls doesn't work.

cat: it prints the contents of the files which have been mentioned along with it as parameters.
The files should be written with their extensions in this function otherwise it shows error.
Flags implemented:

1. > → it creates a new file and asks the user to give input to save to that file.
   eg) cat >test.txt
2. -n : shows the line numbers of the content in the file provided as parameter.

eg) cat file1.txt → shows contents of file 1.
eg) cat file1.txt file2.txt → shows the contents of file 1 and then the contents of file 2
eg) cat >file.txt → creates a file by the name file.txt and then a prompt asks input from the user to add to this file.
eg) cat -n file.txt → shows numbering in the content lines

rm: removes the file whose name is mentioned in the parameter. In case of a wrong name, shows error.
Flags implemented:

1. -i : asks for a confirmation before removing the file.
2. -d: to remove empty directories.

The file names should be written with their extension.

eg) rm hello.txt → if hello.txt exists in the current directory then removes it else shows error
eg) rm -d c → if c directory is empty then removes it else shows error
eg) rm -i test.txt → asks for confirmation and on pressing no there, exits from the process.

mkdir: makes a new directory with the specified name in the current directory.
In case no name of the directory is given as a parameter by the user, it just goes back to the parent process and no directory is made.
Flags implemented:

1. -v : to show a confirmation message every time.
2. -m777: to make a directory in 0777 mode i.e. with read-write-execute permissions to all users.

eg) mkdir hello → hello created

date: shows the current date and time of the system. If wrong parameters are entered by the user, then it shows error.
Flags implemented:

1. -u : displays universal time
2. -s followed by the date to be set in the format (dd mm yyyy): displays the date entered in correct format.

eg) date → current date
eg) date -u → universal date
eg) date -s 12 12 2002 → 12/12/2002

TO SEE THE FUNCTIONS USED, REFER TO THE README FILE ATTACHED