1 Description of the programming project and the first assignment

In the three programming assignments, you will develop a small shell. In this first assignment, you will start the code that you will continue in the following two assignments. The shell that you develop is basically an infinite loop that:

1. prints a *prompt*,
2. reads from the standard input a line of text which includes a command (with its arguments),
3. stores this command in a list of commands,
4. separates the command and its arguments, and
5. processes the command with its arguments.

The *prompt* must be a string of three characters of the form "`@>` ". Note that there is a blank space after the character `>`.

In the first assignment, your shell, named **`myshell`**, should understand basic commands (primarily these that manage the file system). In particular, it should process the following commands:

1. **`author`**: prints your name.
2. **`exit`**: terminates the sell.
3. **`cdir [directory]`**: changes the current working directory of the shell to **`directory`**. When invoked without arguments it prints the current working directory of the shell.
4. **`create [-d] name`**: creates a file or directory in the file system named **`name`**. If argument **`-d`** is provided, then a directory is created. Otherwise an empty file is created.
5. **`delete [-r] name`** deletes a file or directory. If name corresponds to a directory and argument **`-r`** is not provided, the directory will be deleted only if it is empty. If argument **`-r`** is provided and **`name`** is a directory, then the directory and all its content will be deleted.
6. **`list`**: lists all files and directories existing in the file system. (This command implements the same functionality that the command **`ls -a`** implements in Linux.)
7. **`hist [-c]`**: shows the history of the commands executed by this shell. To do this, the commands used within the shell must be stored in some data structure that you have to implement. **`hist -c`** clears the history.

The shell should accept an optional parameter **`--echo`**. That is, the shell can be invoked by typing **`./myshell`** or **`./myshell --echo`** from the system command line. When this parameter is passed to the shell, it should print the input from the user before executing the corresponding command. For instance, if the shell is invoked by typing **`./myshell --echo`** and then the command

```
@> create file.txt
```

is executed in your shell, then the shell should output

```
create file.txt
```

and perform the respective actions. If the shell was invoked by typing **`./myshell`**, then it should just perform the respective actions.

Error handling On processing the input, you may assume that the user command is restricted to be no more than 4095 character long. If the input is longer than that, you may print the error

`command too long`

The shell must not crash if a command longer than expected is given. In general, the shell must not crash if something unexpected occurs or a command cannot be performed for any reason. The shell must not fail to execute a command without reporting a reason. Instead an error message should be produced.

2 Details and clues

The shell must be programed in C. To implement the given functionality you may need to use some system calls and functions available in the standard library. In particular, you may be interested in looking into the functions `printf`, `fgets`, `read`, `write`, `exit`, `getcwd`, `chdir`, `open`, `opendir`, `readdir`, `stat`, `lstat`, `unlink`, `rmdir`, `realpath`, `readlink`... Information about these functions may be obtained by typing `man function` into the Linux system shell.

Most of the functionality that has to be implemented in this exercise is already provided by some Linux system programs. What you are asked to do is to re-implement that functionality to gain the first-hand experience with the system-level programming. The code to be submitted must be implemented in C and only depend on the system calls and functions from the C standard library. The Linux system programs will not be available when the functionality of your implementation will be tested and evaluated. The following are some ideas that may help you in the implementation of your shell.

A shell is basically an endless loop

```
while(1){
    print_prompt();
    get_command(&command);
    process_command(command);
}
```

The only way out is by executing the **`exit`** command or by reaching the end of the file when using a file as standard input (see below the paragraph on *Reading commands from files*). Function `print_prompt()` just prints the prompt. Function `get_command(&command)` may get the command from the user, parse it and store the result in the variable `command`. Then, function `process_command(command)` identifies the command to be executed and calls the function to perform the required functionality.

The first step when processing the input string may consist in splitting this string into words. For this, see the `strtok()` function from the C library. Notice that `strtok()` neither allocates memory nor copy strings, it just breaks the input string by inserting end of string character ('0') in the required places. Check `man strtok` in the system shell for more details. The following function splits the string pointed by `str` (supposedly not NULL) into a NULL terminated array of pointers (`words`). The function returns the number of words in `str`.

```
int split(char* str, char* words[])
{
    int i=1;
    if ((words[0]=strtok(str," \n\t"))==NULL)
        return 0;
    while((words[i]=strtok(NULL," \n\t"))!=NULL)i++;
```

```
    return i;
}
```

Error handling As stated earlier, your shell must not fail to execute a command without reporting a reason. Instead an error message should be produced. When printing errors you should use the standard error file descriptor. When a system call cannot perform (for whatever reason) the task it was asked to do, it returns a special value (usually -1), and sets an external integer variable (`errno`) to an error code. The man page of the system call explains the reason why the system call produced such an error code. A generic message explaining the error can be obtained with any of the following methods:

- the `perror()` function prints that message to the standard error (the screen, if the standard error has not been redirected).
- the `strerror()` function returns the string with the error description if we supply it with the error code.
- the external array of pointers, `extern char* sys errlist[]`, contains the error descriptions indexed by error number so that sys `errlist[errno]` has a description of the error associated with `errno`.

In most cases `perror(command_name)` prints the correct error message when a command cannot be executed. Nevertheless, in some cases, you may need to write your own error messages to adhere to the following specification.

1. If the shell receives a command that it does not recognize, it should print an error message of the form
   `Unrecognized command: command line`
   where **`command line`** is the command line received from the user.
2. If a command is used with the wrong arguments, it should write the error message
   `usage: description`
   where **`description`** is the description of the command provided above. For instance, if the command **`create`** is executed without arguments, the shell should print the error
   `usage: create [-d] name.`
3. If the command cannot execute for any other circumstance, an appropriate error should be printed. This message should be of the form
   `command: explanation`
   where **`command`** is the command being executed and **`explanation`** is a description of the occurred error. This explanation can usually be printed using the function **`perror()`**. Some examples are listed below.
   - If the command **`cdir directory`** is executed and **`directory`** does not exists, the error message
     `cdir: No such file or directory`
     should be printed.
   - If the command **`cdir directory`** is executed and **`directory`** exists but it is not a directory, the error message
     `cdir: Not a directory`
     should be printed.

- If the command **`create name`** is executed and **`name`** already exists, the error message
  `create: File exists`
  should be printed.
- If the command **`delete directory`** is executed and **`directory`** is not empty, the error
  `delete: Directory not empty`
  should be printed.

Recall that the shell should always provide feedback when an error occurs.

To implement the command **`hist`** you may assume that only the last 4096 commands need to be stored. This should be declared as a named constant, and thus it will be easy to modify it at any future point.

Reading commands from files. The shell should be able to run commands provided in a text file using the system redirection. For instance, if **`input.txt`** is a text file where each line is a command, then

`./myshell <input.txt`

should execute all commands, one by one, listed in the file. The output will be printed in the terminal as usual. In particular, if **`input.txt`** contains the lines

```
author
hist
exit
```

typing **`./myshell --echo <input.txt`** in the system shell may produce the following output:

```
@> author
Name:  Jorge Fandinno
@> hist
author
hist
@> exit
```

When reading commands from a file, the **`exit`** command is optional and the shell should terminate when it reaches the end of the file. For instance, if if **`input.txt`** contains the lines

```
author
hist
```

typing **`./myshell --echo <input.txt`** in the system shell may produce the following output:

```
@> author
Name:  Jorge Fandinno
@> hist
author
hist
```

The output can be saved into files **`output.txt`** and **`error.txt`** as follows

`./myshell <input.txt >output.txt 2>error.txt`.

Future extensibility. Note that this exercise will be extended in the future with more commands. Your life will be easier if you build your shell in a way that is easily extensible in the future.

3 Submission

The submission must consist of a single file named **p1.tar.gz** and should be submitted through the **canvas** web platform. Check in canvas the deadline for the submission. The file **p1.tar.gz** can be created with the command

```
tar -czvf p1.tar.gz *
```

After executing this command a file **p1.tar.gz** will be created containing all the files and directories in the current directory. The content of the file can be extracted with the command

```
tar -xzvf p1.tar.gz
```

All the content of **p1.tar.gz** will be extracted in the current directory. The file **p1.tar.gz** must contain a directory called **src/** and a file called **Makefile**. It may contain in addition a directory called **test/**. Inside the directory **src/** should be all the source code of your shell. Executing the command **make** should:

- compile your code,
- create a directory called **bin/**, and
- place a command called **myshell** inside it.

After executing the command **make**, executing the command **./bin/myshell** should execute your shell.

An example file **p1.tar.gz** can be download from the canvas platform. It contains the needed structure, **Makefiles** and a dummy **myshell.c** file. Inside the **test/** directory there is a bash script **compile_test.sh** that can be used to automatically test that these steps work correctly. To check that your **p1.tar.gz** file can be extracted and compiled correctly, copy it together with the script **compile_test.sh** into an empty directory; then execute the script. No error should be reported. **Projects that fail this test will yield no score.**

4 Automatic testing

The sample file **p1.tar.gz** posted on canvas also includes a series of scripts for automatic testing. These scripts are there to help you to check your progress. In particular, there is a script for each but the **author** command. There is also a script named **all_test.sh** that runs all the other scripts and provides a summary of the results. You can run all the test by typing **test/all_test.sh** in the system shell. In case of an error on any of the tests, you can run that test individually. For instance, if test **cdir_test.sh**, then you can invoke **test/cdir_test.sh**. After invoking an individual test, you can check the files **output.txt**, **error.txt**, **expected_output.txt**, **expected_error.txt**, **output_ls.txt** and **expected_ls.txt** in the directory **test/** to see the difference between you **shell** results and the expected ones. These files are not available after running **all_test.sh**. These tests will also be used for evaluating your project (see Section 5 below). If you believe that your shell is producing the correct result according to the above specification and yet the automatic test reports an error, then contact the instructor before submitting the project.

5 Evaluation

Be sure to test your code in the school **odin server** before submitting the exercise: `https://www.unomaha.edu/college-of-information-science-and-technology/about/odin.php` Get an account in the server as soon as possible and be sure to test your code in advance.

Your program will be evaluated by testing a variety of commands. To receive 100 percent, your program must process the entire set of the given commands correctly. If **any of the following errors** occur, the exercise **will not yield any score**:

- The program does not compile or execute on the odin server following the instructions in Section 3. In particular, if the **`compile_test.sh`** does not run correctly as specified in that section.
- The program executes, but it produces a runtime error (like segmentation-fault, bus error...)

The project will be evaluated through (i) automatic testing and (ii) manual/visual code inspection. **Note:** If the provided automatic test for a command fails, then the code implementing that command will not be inspected. That is, if any of the following tests (provided to you in the sample file **`p1.tar.gz`** on canvas) fail, then a 20% of the score will be automatically subtracted from the maximum grade.

`cdir_test.sh`, **`create_test.sh`**, **`delete_test.sh`**, **`hist_test.sh`** and **`list_test.sh`**

Extra tests will be executed to determine your final grade. You are encouraged to exhaustive test your code and write your own automatic tests.