Shell Project

The mid-semester course project consists in the implementation of a shell. Your code will rely on the string manipulation modules implemented through the previous homework assignments, and will extend these with additional functionality. The features to include are split in two blocks: minimum requirements, which will give you the highest grade if fully functional, and optional features for extra credit.

You can choose how many optional features you want to include. Also, you are encouraged to come up with your own additional features! Be creative, and don't just stick to what is proposed in this document.

You can work in groups of at most 2 students. If you work in groups, only one of you should submit the assignment. Add a file named `README` in your project where you specify the names of both students.

Please note that all submissions will be carefully inspected for plagiarism. You can discuss and collaborate with your classmates, but you should write your own code. Any pair of suspiciously similar submissions will be automatically discarded, and will receive no credit.

Submission instructions

The shell project should be submitted on Blackboard as two separate files:

- One *tarball* or *zip* file named `project.tgz` or `project.zip`, containing one directory with all your source files. The package should include a `Makefile` which produces an executable called `shell` when you run command `make`. You can produce a tarball by running the following command on the home folder, assuming that your source code is located in a directory named `project`:

  ```
  $ tar -czvf project.tgz project
  ```

  The code should compile and run correctly on the COE Linux machines by simply running the following commands:

  ```
  $ tar -xzvf project.tgz
  $ cd project
  $ make
  $ ./shell
  ```

  If you worked on your local computer, or on any other remote machine, make sure that you test your program on the COE machines before submitting it. Any code that does not compile and run correctly on the COE machines will be discarded.

- One PDF file named `report.pdf` should include a brief project report. This report should be no longer than 2 pages, using 11pt Times font. Your project report should include the following information:
  - Name of the project author(s)
  - List of features implemented in the shell
  - Description of any additional feature for extra credit
  - Execution demo for at least one feature

Minimum Requirements

- The shell should print a prompt when it is ready for the user to input a command. The prompt is a `$` character followed by a space.
- When a command is launched in the foreground, the shell should wait until the last of its sub-commands finishes before it prints the prompt again. A command is considered to be running in the foreground when operator `&` is not included at the end of the command line.
- When a command is launched in the background using operator `&`, the shell should print the *pid* of the process corresponding to the last sub-command. The *pid* should be printed using square bracket. It should then immediately display the prompt and accept new commands, even if any of the child processes are still running. For example:

```
$ ls -l | wc &
[3256]
$
```

- When a sub-command is not found, a proper error message should be displayed, immediately followed by the next prompt. Example:

```
$ hello
hello: Command not found
$
```

- The input redirection operator `<` should redirect the standard input of the first sub-command of a command. If the input file is not found, a proper message should be displayed. Example:

```
$ wc < valid_file.txt
223  551  5288

$ wc < invalid_file
invalid_file: File not found
```

- The output redirection operator `>` should redirect the standard output of the last sub-command of a command. If the output file cannot be created, a proper message should be displayed. Example:

```
$ ls -l > invalid/path
invalid/path: Cannot create file

$ ls -l > hello
$ cat hello
< Contents of "hello" displayed here >
```

- Sub-commands separated by pipes should provide the desired redirections. Assuming N sub-commands, you can create $N - 1$ pipes in the parent process, and let them be inherited by the child processes every time `fork()` is executed. Each process i other than the first should read from pipe $i - 1$, and each process i other than the last should write into pipe i. Remember that all processes, including the shell process itself, should close all unused file descriptors to avoid undesired blocking reads on pipes.

Optional Features

- Modify the command prompt so that it displays the current directory, and implement the functionality of built-in command `cd` to change it. For example:

  ```
  /home/user$ cd test
  /home/user/test$ cd ..
  /home/user$
  ```

- Implement support for process-specific environment variables. These variables can be specified with a pair `VARIABLE=VALUE` appearing as a prefix of the command. For example:

  ```
  $ VAR=Hello ./test
  ```

  Write a program that prints all environment variables available to the process, and launch it from your shell. Verify that the explicit environment variables are indeed being set in the child process.

- Have the shell print a message whenever a process finishes that was launched in the background. This message would be printed only after the user presses *Enter* and before the prompt is displayed again. The message should print, in brackets, the *pid* of the process that finished. Example:

  ```
  $ sleep 2 &
  [5344]
  $
  [5344] finished
  $
  ```

- Modify the way you read the command from the user so that, instead of reading full strings, you read individual characters from the keyboard. Save the history of the previous commands executed by the user, and make the *down* and *up* arrows on the keyboard serve as history navigation keys. You can output special ASCII characters to the terminal in order to overwrite a line, or change the current position of the cursor.

- Think of your own cool extension, and implement it!