# COMS 352: Introduction to Operating Systems

# Fall 2019

# Project 1: DEVELOPING A UNIX SHELL

### 100 points

### Due: Friday, October 11, 2019, 11:59pm

The shell or command line interpreter is the fundamental user interface to an operating system. A shell interface gives the user a prompt, after which the next command is entered. This project consists of writing a C/C++ program to serve as a shell interface that accepts user commands and then executes each command in a separate process. The goal is to practice some system calls and process management by the operating system.

You are to write a C/C++ program for a simple shell "**myshell**", which has the following properties;

1. The shell MUST support the internal commands like ***date, cal, who, ls, ps, time,*** including the following:
   - i. **cd** <directory> : Change the current default directory to <directory>. If the <directory> argument is not present, report the current directory. If the directory does not exist, an appropriate error message should be displayed. This command should also change the ***PWD*** environment variable.
   - ii. **clr** : Clear the Screen
   - iii. **dir** <directory> : List the contents of directory <directory>
   - iv. **environ** : List all the environment strings
   - v. **echo**<comment> : Display <comment> on the display, followed by a new line (multiple spaces/tabs may be reduced to a single space)
   - vi. **help** : Display the user manual using ***more*** filter
   - vii. **pause** : Pause the operation of the shell until "ENTER/RETURN" key is pressed

viii. **quit** : Quit the shell

ix. The shell environment should contain shell = <pathname> /myshell where <pathname>/myshell is the full path for the shell executable (not a hardwired path back to your directory, but the one from which it was executed).

2. The shell should also support the following:

i) **Handling pipe (|) and more filter**

Your simple shell interface should handle the use of pipe (|) and "more" command as follows;

**-bash-4.3$ ls -l |more**
with the following output:

```
-rwx--x--x.  1 mabdulah domain users   9160 Jan 25 16:28   cll
-rwx------.  1 mabdulah domain users    422 Sep 22 10:43   clone.c
-rwx------.  1 mabdulah domain users    896 Jan 25 15:50   abc.c
-rwx--x--x.  1 mabdulah domain users   8808 Jan 24 22:10   col
-rwx------.  1 mabdulah domain users    842 Jan 24 21:47   xyz.c
-rwx------.  1 mabdulah domain users    842 Jan 24 21:11   xyz.txt
--More—
```

ii) **Handling Multiple Commands**

Your simple shell interface should handle multiple commands entered in the same line separated by "**;**", like

**-bash-4.3$ date;cal;who**

The above should have the following **output** (outputs of the three commands):

```
Wed Sep 13 12:55:51 CDT 2017

   September 2017
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9 10 11
12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27
28 29 30

abi      pts/0    2018-01-28 13:24 (10.64.222.89)
abi      pts/1    2018-01-28 13:31 (10.64.222.89)
mabdulah pts/2    2018-01-28 14:39 (10.25.71.69)
jcu      pts/8    2018-01-19 17:14 (10.24.46.63)
jcu      pts/9    2018-01-19 17:15 (10.24.46.63)
```

3. All command line input is interpreted as program invocation, which should be done by the shell ***fork***ing and ***exec***ing the programs as its own child processes.
   Each command entered at the shell prompt is executed by a Child Process. So, the parent ***fork***s a child process, and this newly created child process executes the command entered by user, using one of the system calls in the ***exec()*** family of C functions.
   The programs should be executed with an environment that contains the entry:

   parent = <pathname>/myshell

   where <pathname>/myshell is as described in 1.(ix) above.

4. The shell must be able to take its command line input from a file. That is, if the shell is invoked with a command line argument:

   ***myshell batchfile***

   then ***batchfile*** is assumed to contain a set of command lines for the shell to process. When the end-of-file is reached, the shell should exit.
   Obviously, if the shell is invoked without a command line argument, it solicits input from the user via a prompt on the display.

5. The shell must support I/O redirection on either or both ***stdin*** and/or ***stdout/stderr***. That is, the command line

   ***programname argl arg2 <inputfile> outputfile***

   will execute the program ***programname*** with arguments ***arg1*** and ***arg2***,
   the ***stdin*** FILE stream replaced by ***inputfile***, and
   the ***stdout/stderr*** FILE stream replaced by ***outputfile***.
   ***stdout*** redirection should also be possible for the internal commands

   **dir**, **environ**, **echo**, and **help**.

   With output redirection; If the redirection character is;

   > ; the ***outputfile*** is created if it does not exist and truncated if it does.

   >> ; the ***outputfile*** is created if it does not exist and appended to if it does.

6. The shell must support background execution of programs. An ampersand (&) at the end of the command line indicates that the shell should return to the command line prompt immediately after launching that program.
7. The command line prompt must contain the pathname of the current directory.

***Note***: You can assume all command line arguments (including the redirection symbols > and >> and the background execution symbol, &) will be delimited from other command line arguments by white space (one or more spaces) and/or tabs.

**Submission**

- You should use C/C++ to develop the code.
- You should work on this project **individually**.
- You need to turn in electronically by submitting a zip file named: Firstname_Lastname_Project1.zip.
  The ZIP file should include:
  Make file
  Readme file
  Source code file(s) (do not include any binary files)
- **Source code must include proper documentation to receive full credit (you will lose 10% of your score, if the code is not well documented as follows).**
  - File Comments: Every .h and .c file should have a high-level comment at the top describing the file's contents, and should include your name(s) and the date.
  - Function Comments: Every function (in both the .h and the .c files) should have a comment describing:
    - what function does;
    - what its parameter values are
    - what values it returns (if a function returns one type of value usually, and another value to indicate an error, your comment should describe both of these types of return values).
  - In-line Comments: Any complicated, tricky code sequences in the function body should contain in-line comments describing what it does (here is where using good function and variable names can save you from having to add comments).
- All projects require the use of a **make file** or a certain script file (accompanying with a **readme file** to specify how to use the script/make file to compile), such that the grader will be able to compile/build your executable by simply typing "make" or some simple command that you specify in your readme file.
- **Source code must compile and run correctly on the department machine "pyrite", which will be used by the TA for grading. If your program compiles, but does not run correctly on pyrite, you will lose 15% of your score. If your program doesn't compile at all on pyrite, you will lose 50% of your score (only for this issue/error, points will be deducted separately for other errors, if any).**
- You are responsible for thoroughly testing and debugging your code. The TA may try to break your code by subjecting it to bizarre test cases.
- You can have multiple submissions, but the TA will grade only the last one.

**Start as early as possible! Ask for explanations/further information early!**