Implement your Own Shell

Theory

- Processes are created with the fork system call
- Fork returns to child process with return value 0.
- The child process created by fork is a copy of the original parent process, except that it has its own process ID.
- When parent executes fork it returns to parent with process id of child .
- This helps parent and child to execute different code after creation of new process.
- After forking a child, parent and child execute simultaneously, this behavior can be modified by using wait( ) call and making parent wait till execution of child process gets over.
- To execute a command we use system call 'exec'. Varieties of exec are available.
- Pipe() **has to be used** to communicate between 2 processes.

Input

- Get user input using function's such as getline( ).
- Parse the input line, clean extra white space and store the parameters in an array.

Output Redirection

The UNIX shell provides this nice feature with the ">" character.

If a user types "ls > output" , nothing should be printed on the screen. Instead, the output of the ls program should be re-directed to the output file.

If the "output" file already exists before you run your program, you should simply append at the end of file .

Background Processes

Your shell should able to provide support for background processes. To put a process in the background, the user types & after the command. For example:

mysh> ls &

mysh>

When a user backgrounds a process, control returns immediately to the shell, allowing the user to type more. Redirect the standard output for a command that is being run in the background to a file. This prevents the output from the command appearing on your screen and interrupting your current work. Command "**lsb**" should list the currently running background processes.

Inter Process Communication

You could support | , a pipe, between two processes. For example, ps | grep “root” would send the *STDOUT* of ps to the *STDIN* of grep using a pipe. You may want to start by supporting pipes only between two processes before considering longer chains. Longer chains will probably require something like handle process *n* and then recursively handle the other *n-1*. Eg : ls -l | grep "Aug" | sort +4n

Functions provided by gcc

- **fork( )**
- **exec( )**
- **wait( )**
- **pipe( )**
- **dup2( )**
- **chdir( )**
- **getcwd( )**
- **getlogin( )**
- **waitpid**
- **exit**

Program Errors(Boundary Test Cases)

For the following situation, you should print the error message and continue processing:

- A command does not exist or cannot be executed.
- A very long command line (over 512 characters, excluding the carriage return)

Your shell should also be able to handle the following scenarios below, which are not errors. A reasonable way to check if something is not an error is to run the command line in the real Unix shell.

- An empty command line.
- An empty command between two or more ';' characters.
- Multiple white spaces on a command line.
- White space before or after the ';' character or extra white space in general.

> ls;ls;ls
> ls ; ls ; ls
> ls>a; ls > b; ls> c; ls >d

Simple Shell (Pseudo Code):

```
int     main (int argc, char **argv)
 {
   while (1){
         int childPid;
         char * cmdLine;

         printPrompt();

         cmdLine= readCommandLine(); //or GNU readline("");

         cmd = parseCommand(cmdLine);


         if ( isBuiltInCommand(cmd)){
            executeBuiltInCommand(cmd);
         } else {
             childPid = fork();
             if (childPid == 0){
               executeCommand(cmd); //calls execvp

             } else {
               if (isBackgroundJob(cmd)){
                     record in list of background jobs
               } else {
                     waitpid (childPid);
               }
            }
         }
 }
```

Note : You have to implement your shell using functions listed here.Use of yacc or lex is strictly prohibited.