Purpose

The purpose of this project is to familiarize you with the mechanics of process control through the implementation of a shell user interface. This includes the relationship between child and parent processes, the steps needed to create a new process, shell variables, and an introduction to user-input parsing and verification.

You must work in a group of two for this project. Groups of three will only be allowed in extreme circumstances. You must implement your shell on the class Linux servers (reachable through address diesburg.cs.uni.edu).

Problem Statement

Design and implement a basic shell interface that supports the execution of other programs and a series of built-in functions, as specified below. The shell should be robust (e.g., it should not crash under any circumstance beyond machine failure).

Part 0: General Shell Structure

The shell (command line) is just a program that continually asks for user input, perhaps does something on the user’s behalf, resets itself, and again asks for user input. Here is an example:

```c
while(1) {
    /* Get user input */
    /* Exit? */
    /* Do something with input */
    /* Reset the shell */
}
```
**Part 1: The Prompt**

At this point, the prompt should indicate that the shell is ready to accept input from the user. Often times, it also shows useful information, such as the name of the user running the shell and the current directory. For now, you just need to implement a simple prompt.

- The prompt should look like the following:
  - prompt$

- There should be a space after the dollar sign so that the user input does not visually run into the prompt.
- Possible extra credit – make it different color(s)!

**Part 2: Command Line Parsing**

Before the shell can begin executing commands, it needs to extract the command name and the arguments into “tokens”. It might be nice to store these tokens into an array so that you can then parse each one in order. In our shell, the first token will always be the name of the program we wish to execute, and all remaining tokens (perhaps including the first token) will be arguments to that program.

Take note of the following assumptions:

- No leading whitespace
- One space separates the command line tokens.
- No trailing whitespace
- You can assume that each token is no longer than 80 characters.
- You can assume that a command will have at most 10 space-separated tokens

Make sure that you can successfully print out your array of tokens through different iterations of your shell loop before moving on. If you see garbage in any of your commands or arguments, try using the C library call memset() or bzero() to clear out your input string and token array before and/or after you are done using them.

The C library call fgets() can gather user input from the screen and save it into a string (C character array). See the man pages for fgets for more information.

**Part 3: Command Execution**

Once the shell understands what commands to execute it is time to implement the execution of simple commands. Since the execution of another program involves creating another process, you will have to use the fork() system call to create another process. Once you have created the new child process, that process must use the execvp() system call to execute the program.
Finally, the parent (shell) process must wait for the child process to complete before releasing the child’s resources using the waitpid() system call.

However, the execvp() system call may return if there is an error. If it does, your shell should print an error, reset, and prompt for new input. Here is an example:

```
prompt$ lalala -a
Error: Command could not be executed
prompt$
```

**Part 4: Built-ins**

Not all commands are actually programs, and your shell must implement two “built-in” commands. In other words, if you encounter any of these two commands, do not execute them using fork(), exec(), and waitpid(). Instead, your shell should call a subroutine that implements the following functionality:

- **exit** – terminates your running shell process and prints 'exit'.
  ```
prompt$ exit
  exit
  (shell exits)
  ```

- **cd [PATH]** – Changes the present working directory. There is actually no ‘cd’ program – the cd command is implemented by the shell. For now, just print “Command not implemented” and re-prompt for new input.
  ```
prompt$ cd ..
  Command not implemented
  prompt$
  ```

  For extra credit, make cd work similar to how it works in bash! You will need to use the chdir() system call and update the PWD environmental variable with setenv().

  ```
prompt$ pwd
  /user/diesburg/os/project1
  prompt$ cd ..
  prompt$ pwd
  /user/diesburg/os
  prompt$ cd project1
  prompt$ pwd
  /usr/diesburg/os/project1
  ```

**Create a README file**

Please create a README text file that contains the following:
- The names of all the members in your group
- A listing of all files/directories in your submission and a brief description of each
- Instructions for compiling your programs
- Instructions for running your programs/scripts
- Any challenges you encountered along the way
- Any sources you used to help you write your programs/scripts

### Grading

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Points Possible</th>
<th>Points Earned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compilation without warnings or errors</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Command parsing</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>No zombies (correct use of waitpid)</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Executes any External Command</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Uses fork correctly</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Built-in: exit</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Built-in: cd</td>
<td>EXTRA 5</td>
<td></td>
</tr>
<tr>
<td>README</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Code:</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Readable with comments (minimum: top of program, above all functions, and in tricky spots)</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Midway Demo:</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

(Parts 1-2 completed, Part 3 fork/exec/waitpid attempted)

In addition, at submission time, you must each individually submit a partner review on eLearning based on the amount of perceived effort your partner put into this project.

### Project Submission

Please turn in your assignment **before your demo time** by zipping an archive of the files and submitting to the appropriate project submission space on eLearning. Include your name(s) in the README and in the comments at the top of any programming files. Do not zip or send any .o, core dumps, or compiled programs (-5 pts for each one I find)! If you fail to submit your project before demo time, I will take off -15 pts.