UNI CS 3430, Section 1
Operating Systems

Project 1 - Implementing a Shell (worth 60pts)

Half-way turn in date (Parts 1-2 completed, Part 3 attempted): 2/6
Final submission (All parts completed): 2/13

Language Restrictions: C only
Additional Restrictions: The system() system call may not be used.

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 may work in a group of two for this project. You must implement your shell on the class Linux server (reachable through address student.cs.uni.edu). Your project will be tested on the Linux server.

Grading

The entire project grade is worth 12% of your class grade (60/500 total class points). At the half-way turn-in date, you should have parts 1-2 completed and part 3 attempted. The half-way turn in is graded on effort and worth 1/3 (20 points) of your project grade. The final submission should have everything working and is graded on the correctness of your solution worth 2/3 (40 points) of your project grade.

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:

```
while(1)
{
```

```
                */ Get user input */
                */ Exit? */
                */ Do something with input */
        }
```

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.

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. The function strtok() will be helpful to do this, but it is a bit tricky to use. Be sure to look at the class notes.

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 strtok, fgets, memset, or bzero 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. 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
```

- **showpid** – shows the last 5 ***child*** process IDs created by your shell.

```
prompt$ showpid
4987
4992
5001
```

```
5002
5004
```

Part 5: Better Prompt

Let's make your shell more usable.

- Modify the prompt so that it displays the current working directory before the $ sign:
  - `/home/xkcd/$`
- Make the prompt a different color. Optionally, you can make other parts of your shell different colors as well.

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 Rubric (out of 40 points)

Requirement	Points Possible	Points Earned
Command parsing	5	
No zombies (correct use of waitpid)	5	
Executes any External Command	5	
Uses fork correctly	5	
Built-in: cd	5	
Built-in: showpid	5	
Part 5: prompt shows current working directory and is a different color	5	
README	5	

If the program does not compile, I will assign a zero to your submission grade.

Academic Integrity

All work turned in by you must be your own. I have a database of similar previous student work and similar Internet solutions. I also automatically run a plagiarism checker against similar work and other submissions in the current class. If I find that you have misrepresented other work as your own, you will receive a grade of 0 for this project and optionally a letter to the Provost to go on your academic record.

Project Submission

Both the C code file and the README text file should be turned in for both the half-way and final submission dates. Navigate to the class eLearning page and click on the Project Submission link on the left. Follow the links to submit your C code file and README file separately.

Half-way submissions cannot be late. Final submissions can be up to two days late, but note the penalty on the syllabus for every day late.