Operating Systems Lab (CS 470):

Lab 1: Create a mini shell using C/C++ programming language.

Overview

The OS command interpreter is the program that people interact with in order to launch and control programs. On UNIX systems, the command interpreter is usually called *the shell*: it is a user-level program that gives people a command-line interface to launching, suspending, and killing other programs. `sh`, `ksh`, `csh`, `tcsh`, and `bash` are all examples of UNIX shells.

The example below illustrates the use of the cwushell, the shell that you will create. The example shows a prompt `cwushell>` and the user's next command: `cat Prog.c`. This command displays the file *Prog.c* on the terminal using the UNIX `cat` command.

```
cwushell> cat Prog.c
```

Every shell is structured as a loop that includes the following:

1. print a prompt
2. read a line of input from the user
3. parse the line into the program name, and an array of parameters
4. use the `fork`() system call to spawn a new child process
    - the child process then uses the `exec`() system call to launch the specified program
    - the parent process (the shell) uses the `wait`() system call to wait for the child to terminate
5. once the child (i.e. the launched program) finishes, the shell repeats the loop by jumping to 1.

Although most of the commands people type on the prompt are the name of other UNIX programs (such as `ls` or `cat`), shells recognize some special commands (called internal commands) which are not program names. For example, the `exit` command terminates the shell, and the `cd` command changes the current working directory. Shells directly make system calls to execute these commands, instead of forking child processes to handle them.

Instructions

Write a mini shell program (in C/C++) called `cwushell`. The shell has the following features:

- It recognizes 4 internal commands:

1. `exit [n]` terminates the shell, either by calling the `exit`() standard library routine or causing a return from the shell's `main`(). If an argument (n) is given, it should be the exit value of the shell's execution. Otherwise, the exit value should be the value returned by the last executed command (or 0 if no commands were executed.)
2. `mkdir [dir]` uses the `mkdir()` standard library to create a DEFAULT directory. If a parameter (dir) is provided, the directory with the given name (dir) should be considered.
3. `ls` list the content of the current directory.

4. `cmp <filename1> <filename2>` will compare two files and will return which is the first byte where is a difference if any.

Your shell should uses some flavor of `exec`(), to invoke the executable, passing it any command line arguments.

Notes

- Please take a look at the manual pages for `execvp`, and `getenv`.
- To allow users to pass arguments to executables, you will have to parse the input line into words separated by whitespace (spaces and '\t' (the tab character)), and then create an array of strings pointing at the words. You might try using `strtok`() for this (`man strtok` for a very good example of how to solve exactly this problem using it).
- You'll need to pass the name of the command as well as the entire list of tokenized strings to one of the other variants of exec, such as `execvp`(). These tokenized strings will then end up as the `argv`[] argument to the main() function of the new program executed by the child process. Try `man execv` or man `execvp` (under Linux) for more details.
- Error handling should be considered.
- It is not required to implement the 4 commands considering fork()!

Rubric

Task	**Points**
Error handling	1
exit function implementation	1
mkdir function implementation	2
ls function implementation	2
cmp function implementation	4