CS 3100 Command Interpreter Project
Part 1: C-String Review and Command Parsing

Silberschatz, Galvin, and Gagne describe both the purpose of and the general approach for implementing a command line interpreter (p. 50). Specifically, they state that, "The main function of the command interpreter is to get and execute the next user-specified command." They further suggest the two main approaches for implementation: "In one approach, the command line interpreter itself contains the code to execute the command. . . . An alternative approach. . . implements most commands through system programs." In the latter case, the command line interpreter spawns a new process to execute the system program. In practice, a combination of both techniques are often used (based on the complexity of the command).

The first programming project, which is divided into three separate assignments, will create a simple command line interpreter. (Although a graphical user interface looks quite different from a command line interface, the two are very similar with regard to executing user commands.) Command line interpreters are often called shells, particularly on systems that provide more than one and allow users to choose. We will create a Weber Shell named **`wsh`**, which will contain code to carry out some user commands but will also spawn processes to complete other commands.

Project Overview

- Part 1: C-String Review, Command Parsing, File I/O Review
- Part 2: Simple Operating System Calls (Chapter 2)
- Part 3: Advanced Operating System Calls, Spawning Processes (Chapters 2 and 3), and Recursion Review

Assignment

Operating systems, specifically operating system calls, typically exchange data with application programs through primitive data types and structures. For textual data C-strings are used rather than C++ string objects. A C-string is a primitive data type, which means that C-string operations are carried out with non-member functions and primitive operators; C-strings neither support member functions nor overloaded operators because they are not instances of any class. This assignment uses C- strings to read a command line and to parse it into commands and arguments.

Two C++ files, `wsh.h` and `wsh.cpp`, plus a Makefile are located in `/var/classes/cs3100/lab1/` and should be copied to your `~/cs3100/lab1` folder. Here's how:

```
mkdir ~/cs3100/lab1
cp /var/classes/cs3100/lab1/* ~/cs3100/lab1
```

Complete the `.cpp` file as outlined below, but for Lab 1 do not alter the `.h` file. Place your source files in subdirectory `~/cs3100/lab1` on `icarus`. Compile using `make`, which will call `g++` with the correct parameters. Consider testing each feature as you design and code it.

1. Making the prompt
   a. `wsh.h` declares a C-string named `cwd` (current working directory)
   b. Look up the syntax and usage of the `getcwd` system call (prototyped in `unistd.h`) in the man

pages and use it to initialize `cwd` (this is done at the end of the `wsh` constructor).

c. Use `cwd` as part of the prompt. Print the prompt at the beginning of the next command function.

d. Your prompt will mimic the Linux command line prompt but you must make your `wsh` prompt slightly different than the normal Linux command prompt in some way. The icarus Linux prompt looks like this (for user tedcowan):

```
tedcowan@icarus:~/cs3100/lab1$
```

Your wsh prompt will be different and look like this (using the getcwd system call):

```
/home/tedcowan/cs3100/lab1=>
```

e. Do not print a new-line after the prompt. The cursor should appear one space after the prompt just as it does when the BASH shell prompt appears.

2. Reading and parsing the command line (complete the `next_command` function)

a. Define a <u>local</u> C-string variable into which you will read the unprocessed command line (this <u>variable must be static</u> – why?); use the symbolic constant `PATH_MAX` (declared in `unistd.h`) to size the C-string

b. Prompt the user for input by displaying the prompt created in step 1 above

c. Read the command line

d. Parse the command line into its components and count the components

i. `wsh.h` defines two member variables: `argc` and `argv`. `argc` is the argument count (the number of items in the command line). `argv` is the argument vector (an array of C-strings). The results of the `next_command` function are stored in these variables

ii. The individual components of a command line are separated by "white space" (spaces or blanks and horizontal tabs)

iii. Look up the syntax and usage of the `strtok` function (prototyped in `cstring`) and use it to parse the command line input

iv. At this point, the only "command" that `wsh` will recognize is "exit," so strings of words can be used to test the program (they are printed by the default statement at the end of the `interpret` function)

v. Example input: `see the quick red fox`

vi. Results: `argc is 5; argv[0] is see; argv[1] is the; argv[2] is quick; argv[3] is red; and argv[4] is fox`

3. The copy function (copy one file to another); this command mimics but simplifies the Windows copy and the Unix/Linux cp commands – when in doubt, experiment with one of these commands to see how the `wsh` command should function (subject to the simplifications listed below). <u>NOTE: you will not implement the full `wsh copy` command in this assignment – you need only to validate and possibly correct the destination file name.</u>

a. `wsh` uses the Linux path separator (`'/'`).

b. To simplify programming, `wsh` does <u>not</u> accept file or directory names with spaces

c. On the command line, the copy operation has the form `copy file1 file2` where *`file1`* and *`file2`* represent the names of the two files as entered by the user. This command copies the contents of `file1` to `file2`

d. After the command is read and fully parsed (step 2 above), the three parts of the command are stored in the `wsh` member variable `argv` as follows (i.e., next command initializes these variables to the following values):
   i. the command name, `copy`, is stored in `argv[0]`
   ii. `file1` is stored in `argv[1]`
   iii. `file2` is stored in `argv[2]`
   iv. The file names may be in either relative- or absolute-path formats
   v. For Lab 1, neither file names a directory (directories may appear in the path but the last name must be that of a file)

e. Fixing the `copy` function file names
   i. The second file name may be `'.'` (pronounced "dot"), which denotes the current directory. Although the first file name may take one of many forms, the problem arises when using dot as the second file name. All of the following illustrate legal copy commands (with a space before the single dot):

```
copy ../file1 .
copy ../dir1/dir2/file1 .
copy /file1 .
copy /dir1/dir2/file1 .
```

   ii. Each of these commands creates a file named `file1` in the current directory
   iii. `ifstream` can deal with all of the above forms for file1, but `ofstream` will not accept dot as a file name (you will use instances of these classes in lab 2 to finish the copy command)
   iv. To solve this problem you need to create an appropriate file name to replace the dot as the destination file. Take the last element from the first file name (e.g., `file1`) and use that as the destination file name. You should not change `argv`, so you may need a local temporary variable. Consider using the `strrchr` function and address arithmetic.

f. Do not allow a file to be copied on top of itself (i.e., don't copy if both file names are the same - just return). It is possible to create many different names for the same file by using various relative path names, we simplify the copy command by deliberately neglecting to test for all of these variations – just check for strict equality by comparing the two files names with `strcmp()`.

4. Mark the assignment as complete in Canvas when you are ready for me to grade your work. Please do not upload any source code to Canvas, simply say "Done" in the comment box.

Grading

I often use scripts to help grade programs. Please note the following:

- My scripts use the `g++` compiler on icarus to compile your programs. If you use a different compiler or a different Linux system to create your programs, please test them on icarus before submitting them for grading.
- The automated test bed will fail if you do not follow the naming instructions, so please be sure that the programs are named correctly.

Here is how you earn points for this assignment:

FEATURES	POINTS
Must-Have Features	
Files are named correctly and found in their proper place on `icarus`	10
Compiles without errors or warnings	10
Correctly displays the appropriate prompt	10
Exits properly when `"exit\n"` is typed	10
Required Features	
Echos all words typed on the commandline, one per line	30
Echoes "`copy sourcefile .`" command with destination of the '`.`' replaced with `CWD/sourcefile`	20
Echoes "`copy filea fileb`" on a single line	10
Grand Total	100