Assignment 1

CMPT 300 – Operating Systems – Instructor: Nate Payne

Part 1 – Counting Words (SMP0)

====================

The purpose of this assignment is to help you create, debug and extend C programs that run within a shell environment and utilize basic I/O and string manipulation functions.

The scenario: your well-intentioned-but-inexperienced pair-programming buddy has just written some code for the first assignment. Unfortunately, they dropped the course/were abducted by aliens, and it is now up to you to pick up where they left off.

The accompanying program files (Makefile and main.c) contain examples of good and bad programming practices and includes deliberate errors. Your job is to find and fix the errors, implement missing features, and learn some tricks of the trade in the process.

Luckily, a test program is included to help you along. You can (and should) use it to check your work. See the Self-Evaluation section for details.

There are three parts to this Project.

Please submit a zip folder with the following naming conventions to canvas:

LastName_FirstName_StudentNumber_Assig1

This file should include all code, and a text file called answers.txt that includes answers to all questions. There may be questions for both part 1 and part 2, so make sure that you address all questions.

Note that SMP refers to the acronym short machine problem.

Part One: Crash Course in C

Read the file review.txt. The list of topics it contains is useful for self-assessment and as a study guide.

Answer the questions in 1-Pointers.txt. Try to identify the key to each problem and keep your answers concise and to the point; 2-3 sentences should suffice. These questions bring up important points about pointer usage in C. Keep these in mind when working on the remainder of the problem.

Part Two: Fixing the Bugs

The purpose of the provided program is to count words specified as command-line arguments. Read the description of the program and its functionality in the comment at the top of main.c. Now read through the rest of main.c and the Makefile and understand what each part does.

Finally, compile and run the program from the shell:

```
> make
(ignore the compiler warning for now)
> ./main
```

The program compiles and links... so it must work! But is it really doing what it is supposed to do?

Answer the questions in 2-Debugging.txt and fix the corresponding bugs in main.c. Again, try to keep your answers brief and focused.

Part Three: Enhancements

Now that the bugs have been ironed out, it's time to add some functionality to our word counting program. Follow the instructions in 3-Enhancements.txt to complete the word counter.

Self-Evaluation

The testrunner program is included so that you can check your progress as you implement different parts of the MP.

To run all tests:

```
> make test
```

To run a specific test, e.g., stderr_output:

```
> ./main -test stderr_output
```

As you can see, testrunner is implemented entirely in C. If you found this MP too easy, or are just plain curious, feel free to look at the implementation and see if you can figure out how everything works. The relevant files are: smp0_test.* and testrunner.*.

Note: You should remove or disable any additional debugging output you may have created before running the tests. One way to do this easily is through the use of the preprocessor directive #ifdef:

```
#ifdef DEBUG
fprintf(stderr, "My string %s %d\n", var1, var2);
#endif
```

Then add -DDEBUG to the CCOPTS line in the Makefile during development, and remove it before testing. Make sure that your code still compiles and runs without the debug output!

Submit all code for this part of the assignment.

Reminder: Do not copy or plagiarize any code from any other student in the course and be sure to cite all online references.

Do not copy or plagiarize from any source online. Any student found doing so will receive a 0 for the assignment portion of the course. My goal is to maximize your learning, so please focus on that!

Part 2 – A Simple Shell

INSTRUCTIONS
============

In this MP, you will explore and extend a simple Unix shell interpreter. In doing so, you will learn the basics of system calls for creating and managing processes.

STEP 1: Compile the shell
==========================

```
    chmod +x b.sh
    make
    make test   # Use in Step 5 to test your changes to the MP
    ./shell
```

STEP 2: Try using the shell
============================

Note: You need to specify the absolute paths of commands.

Some commands to try:

```
    /bin/ls
    /bin/ls ..
    cd /
    /bin/pwd
    /bin/bash
    exit
    ./shell      (Note: You need to be in the smp1 directory.)
    ./shell&     (Note: You need to be in the smp1 directory.)
    ./b.sh       (Note: You need to be in the smp1 directory.)
    /bin/kill -s KILL nnnn      (Where nnnn is a process ID.)
```

"./" means the current directory

STEP 3: Study the implementation of the shell
==

In preparation for the questions in Step 4, please explore the source code for the shell contained in 'shell.c'. You needn't understand every detail of the implementation, but try to familiarize yourself with the structure of the code, what it's doing, and the various library functions involved. Please use the 'man' command to browse the Unix manual pages describing functions with which you are unfamiliar.

STEP 4: Questions
==================

1. Why is it necessary to implement a change directory 'cd' command in the shell? Could it be implemented by an external program instead?

2. Explain how our sample shell implements the change directory command.

3. What would happen if this program did not use the fork function, but just used execv directly? (Try it!)

   Try temporarily changing the code 'pid_from_fork = fork();' to 'pid_from_fork = 0;'

4. Explain what the return value of fork() means and how this program uses it.

5. What would happen if fork() were called prior to chdir(), and chdir() invoked within the forked child process? (Try it!)

   Try temporarily changing the code for 'cd' to use fork:

   ```
   if (fork() == 0) {
       if (chdir(exec_argv[1]))
           /* Error: change directory failed */
           fprintf(stderr, "cd: failed to chdir %s\n", exec_argv[1]);
       exit(EXIT_SUCCESS);
   }
   ```

6. Can you run multiple versions of ./b.sh in the background? What happens to their output?

7. Can you execute a second instance of our shell from within our shell program (use './shell')? Which shell receives your input?

8. What happens if you type CTRL-C while the countdown script ./b.sh is running? What if ./b.sh is running in the background?

9. Can a shell kill itself? Can a shell within a shell kill the parent shell?

   ```
   ./shell
   ./shell
   /bin/kill -s KILL NNN      (Where NNN is the the parent's PID.)
   ```

10. What happens to background processes when you exit from the shell? Do they continue to run? Can you see them with the 'ps' command?

    ```
    ./shell
    ./b.sh&
    exit
    ps
    ```

STEP 5: Modify the MP
======================

Please make the following modifications to the given file shell.c. As in SMP0, we have included some built-in test cases, which are described along with the feature requests below.

In addition to running the tests as listed individually, you can run "make test" to attempt all tests on your modified code.

1. Modify this MP so that you can use 'ls' instead of '/bin/ls' (i.e. the shell searches the path for the command to execute.)

   ```
   Test: ./shell -test path
   ```

2. Modify this MP so that the command prompt includes a counter that increments for each command executed (starting with 1). Your program should use the following prompt format:

   ```
     "Shell(pid=%1)%2> "  %1=process pid %2=counter
   ```

   (You will need to change this into a correct printf format)
   Do not increment the counter if no command is supplied to execute.

   ```
   Test: ./shell -test counter
   ```

3. Modify this MP so that '!NN' re-executes the n'th command entered. You can assume that NN will only be tested with values 1 through 9, no more than 9 values will be entered.

   ```
   Shell(...)1> ls
   Shell(...)2> !1     # re-executes ls
   Shell(...)3> !2     # re-executes ls
   Shell(...)4> !4     # prints "Not valid" to stderr

   Test: ./shell -test rerun
   ```

4. Modify the MP so that it uses waitpid instead of wait.

5. Create a new builtin command 'sub' that forks the program to create a new subshell. The parent shell should run the imtheparent() function just as if we were running an external command (like 'ls').

   ```
   ./shell
   Shell(.n1..)1> sub
   Shell(.n2..)1> exit  # Exits sub shell
   Shell(.n1..)1> exit  # Exits back to 'real' shell
   ```

6. Create a new global variable to prevent a subshell from invoking a subshell invoking a subshell (i.e., more than 3 levels deep):

   ```
   ./shell
   Shell(.n1..)1> sub
   Shell(.n2..)1> sub
   Shell(.n3..)1> sub   # prints "Too deep!" to stderr

   Test: ./shell -test sub
   ```

Submit all code for this part of the assignment. Ensure that the code is properly commented.

Ensure that answers to all questions are included in the answers.txt file.

Reminder: Do not copy or plagiarize any code from any other student in the course and be sure to cite all online references.

Do not copy or plagiarize from any source online. Any student found doing so will receive a 0 for the assignment portion of the course. My goal is to maximize your learning, so please focus on that!