Operating Systems (CS:3620)

Assignment 4 Total points: 100

Due: 2 Weeks (by 11:59 pm of October 12, 2017)

This assignment will contribute 10% to your final grades.

You can participate in a group of maximum two members.

Please select your group member as soon as you can.

Submission: Please submit your source code through ICON. Zip all the source code before submission. Please make sure you name the zip file in the following way: *<last_name>-<first_name>-Assignment_4.zip*. You should replace <last_name> and <first_name> with your last and first name, respectively. For instance, if a student's name is Bob Lee the submission zip file should be called `Lee-Bob-Assignment_4.zip`.

Grading: Please make sure your source code compiles with GCC, otherwise you will not obtain any points. Please comment your source code so that it is easy for the TA to understand it. Your program will be graded automatically against some TA-generated test cases.

For compiling a C program named `problem1.c` in the terminal using `gcc` and in the process generate the executable file `problem1`, type in the following command: `gcc problem1.c -o problem1`. To run the executable without any command line arguments, you would need to type in `./problem1` in the terminal.

Please make sure you are using an Ubuntu environment for this assignment. We will provide a Ubuntu virtual machine image. Please see the end of the assignment for detailed instructions about setting up the virtual machine. Note: This is the same image as Assignment 1. **If you successfully installed this image, there is no need to go through the installation process again.**

Cheating and Collaboration: ***This is an individual project****, you can discuss with your peers but cannot copy source code. Please do not copy source code from the Internet. Think about the worst-case scenario when you get caught.*

1. (100 points) **Writing a toy UNIX Shell**

   This program should be called "mysh.c"

Objectives

There are three objectives to this assignment:

- To familiarize yourself with the Linux programming environment.
- To learn how processes are created, destroyed, and managed.
- To gain exposure to the necessary functionality in shells.

There is one more informal objective. It is customary for students who are taking the OS class to write a shell in C. Otherwise, there is a chance the instructor may be blamed for not teaching appropriate topics in the OS class.

Overview

In this assignment, you will extend the UNIX toy shell that was part of Assignment 3. The shell should operate in this basic way: when you input a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished.

The shells you implement will be similar to, but simpler than, the one you run every day in Unix. You can find out which shell you are running by typing "echo $SHELL" at a prompt. You may then wish to look at the man pages for csh or the shell you are running (more likely tcsh or bash , or for those few wacky ones in the crowd, zsh or ksh or even fish) to learn more about all of the functionality that can be present. For this project, you do not need to implement too much functionality.

1 Input

1.1 Maximum Length of Input

The maximum length of a line input into the shell is **512 bytes** (excluding the new line character at the end).

1.2 Input Format

The format of the input will vary depending on the command. However, note that the following things will always hold true for any command.

(a) Multiple spaces might precede the first command. For instance, the following two commands should both execute succesfully.

Note: <space> simply denotes a space character.

```
mysh> <space> <space> cd
mysh> cd
```

(b) Multiple spaces might appear after the last command. For instance, the following two commands should both execute succesfully.

Note: <space> simply denotes a space character.

```
mysh> cd <space> <space>
mysh> cd
```

(c) If a command takes an argument there might be more than one space between the argument and the command. For instance, the following two commands should both execute succesfully.

Note: <space> simply denotes a space character.

```
mysh> cd <space> <space> dir
mysh> cd <space> dir
```

(d) Every line input in the shell either through the interactive mode or batch mode (These will be discussed more in depth in the "**Mode of Operation**" section down below) will end with the new line character ('\n').

1.3 Syntatically correct commands

You are safe to assume the input will be syntatically correct. This means that the two special characters (& , >) will be in their appropriate location (These will be explained more in depth in the "**Redirection**" and "**Background Jobs**" sections down below). **Note:** This does **NOT** mean that you can assume the other commands will not contain incorrect number of arguments when your shell is tested. You should apply **"Defensive Programming"** such as described in the "Defensive Programming and Error Messages" section down below.

The & symbol which will stand for a background job (More information about this in the "**Background Jobs**" section below) should always be the last command in a line if present.

For instance, the following example shows where the & symbol will always be

```
mysh> ls &
```

The > symbol which will signify redirection (More information about this in the "**Redirection**" section down below) will always be the 2nd or 3rd command from the last in a line if present.

For instance, the following two examples show where the > will always be.

```
mysh> ls > /temp/ls-output
mysh> ls > /temp/ls-output &
```

2 Functionality

2.1 Mode of Operation

Interactive Mode

Your shell in the `interactive mode` is basically an interactive loop: it repeatedly prints a prompt "**mysh**> " (note the space after the > sign), parses the input, executes the command specified on that line of input, and waits for the command to finish. This is repeated until the user types "`exit`".

You should structure your shell such that it creates a new process for each new command (there are a few exceptions to this, which we will discuss below). There are two advantages of creating a new process. First, it protects the main shell process from any errors that occur in the new command. Second, it allows for concurrency; that is, multiple commands can be started and allowed to execute simultaneously.

Your basic shell should be able to parse a command, and run the program corresponding to the command. For example, if the user types "`ls -la /tmp`", your shell should run the program `ls` with all the given arguments and print the output on the screen.

Note that the shell itself does not "implement" `ls` or really many other commands at all. All it does is find those executables and create a new process to run them. More on this below.

Hints:

Parsing: For reading lines of input, you may want to look at `fgets()`. Be sure to check the return code of this routine for errors! (If you see an error, the routine `perror()` is useful for displaying the problem. But **do not** print the error message from `perror()` to the screen. You should only print the one and only error message that is specified in the "**Defensive Programming and Error Messages**" section below). You may find the `strtok()` routine useful for parsing the command line (i.e., for extracting the arguments within a command separated by whitespace or a tab or a specific character, etc). Some have found `strchr()` useful as well.

Batch Mode

Testing your shell in interactive mode is time-consuming. To make testing much faster, your shell should support `batch mode`.

In interactive mode, you display a prompt and the user of the shell will type in one or more commands at the prompt. In batch mode, your shell is started by specifying a batch file on its command line; the batch file contains the same list of commands as you would have typed in the interactive mode. Each command would be seperated by the new line character.

In batch mode, you should not display a prompt. You should print each line you read from the batch file back to the user before executing it; this will help you when you debug your shells (and us when we test your programs). To print the command line, do not use `printf` because `printf` will buffer the string in the C library and will not work as expected when you perform automated testing. To print the command line, use `write(STDOUT_FILENO, ...)` this way:

```
write(STDOUT_FILENO, cmdline, strlen(cdmline));
```

In both interactive and batch mode, your shell should terminate when it sees the `exit` command on a line or reaches the end of the input stream (i.e., the end of the batch file).

To run in batch mode, your C program must be invoked exactly as follows: `./mysh` [*batchFile*]

The command line arguments to your shell are to be interpreted as follows.

`batchFile`: an optional argument (often indicated by square brackets as above). If present, your shell will read each line of the batchFile for commands to be executed. If not present or readable, you should print the one and only error message (see the "**Defensive Programming and Error Messages**" section below). Implementing the batch mode should be very straightforward if your shell code is nicely structured. The batch file basically contains the same exact lines that you would have typed interactively in your shell. For example, if in the interactive mode, you test your program with these inputs:

```
% ./mysh
mysh> ls
some output printed here
mysh> ls > /temp/ls-out
some output printed here
mysh> notACommand
some error printed here
```

then you could cut your testing time by putting the same input lines to a batch file (for example `myBatchFile`):

```
ls
ls > /tmp/ls-out
notACommand
```

and run your shell in batch mode:

```
./mysh myBatchFile
```

In this example, the output of the batch mode should look like this:

```
ls
some output printed here
ls > /tmp/ls-out
some output printed here
notACommand
some error printed here
```

Hints:

Parsing: To open a file and get a handle with type `FILE *` , look into `fopen()`. Be sure to check the return code of this routine for errors! (If you see an error, the routine `perror()` is useful for displaying the problem. But **do not** print the error message from `perror()` to the screen. You should only print the one and only error message that is specified in the "**Defensive Programming and Error Messages**" section below). You may find the `strtok()` routine useful for parsing the command line (i.e., for extracting the arguments within a command separated by whitespace or a tab or a specific character, etc). Some have found `strchr()` useful as well.

2.2 Built-in Commands

Whenever your shell accepts a command, it should check whether or not the given command is one of the built-in ones. Since this assignment is extending the shell written for Assignment 3 Problem 2 all of the commands `(exit, pwd, cd, show-dirs, show-files, mkdir, touch, clear)` should be implemented in this shell as well.

There is however a **minor change** to the format of the commands. The minor change is that now there are optional spaces preceding the given command `(e.g. '' cd Documents'' )`. However their functionality should be the same. For more information about their functionality and in depth description please refer back to the description for Problem 2 in Assignment 3.

Formats for commands:

```
[Optional space(s)] exit [Optional space(s)]
[Optional space(s)] pwd [Optional space(s)]
[Optional space(s)] cd [Optional space(s)]
[Optional space(s)] cd [One or more optional spaces] dir [Optional space(s)]
[Optional space(s)] show-dirs [Optional spaces(s)]
[Optional space(s)] show-files [Optional space(s)]
[Optional space(s)] mkdir [One or more Optional spaces] dir [Optional space(s)]
[Optional space(s)] touch [One or more Optional spaces] file [Optional space(s)]
[Optional space(s)] clear [Optional space(s)]
```

If any error occurs you should print an error message. Please look at the " **Defensive Programming and Error Messages**" section below for more information.

Hints:

pwd - `cwd()` is a way to get the current directory.

cd - You can use `getenv("HOME")` to get the path to the home directory. You can use `chdir()` to change the current working directory.

show-dirs - You can use `readdir()` to get sub-directories and files from the given directory.

show-files - You can use `readdir()` to get sub-directories and files from the given directory.

mkdir - You can use `mkdir()` to create a new directory.

touch - You can create a new file with `fopen()`

clear - Look at how an actual UNIX shell does this.

Chapter 4.22 and 7.9 of the "Advanced UNIX Programming book" might be helpful if you are having problems with these commands.

2.3 Non Built-in Commands

As mentioned above in the **Overview** and the **Mode of Operation** sections there are only a handful of commands that are actually implemented.

For the rest of the commands that are not in the "Built-in" list your program should simply find the executables and create a new process to run these.

Hints:

Executing Commands: Look into `fork`, `execvp`, and `wait/waitpid`. See the UNIX man pages for these functions, and also read the Advance Programming in the UNIX Environment, Chapter 8 (specifically, 8.1, 8.2, 8.3, 8.6, 8.10). Before starting this project, you should definitely play around with these functions to get a better understanding of them.

You will note that there are a variety of commands in the exec family; for this project, you must use `execvp`. You should not use the `system()` call to run a command. Remember that if `execvp()` is successful, it will not return; if it does return, there was an error (e.g., the command does not exist). The most challenging part is getting the arguments correctly specified. The first argument specifies the program that should be executed, with the full path specified; this is straight-forward. The second argument, `char *argv[]` matches those that the program sees in its function prototype:

`int main(int argc, char *argv[]);` Note that this argument is an array of strings, or an array of pointers to characters. For example, if you invoke a program with:

foo 205 535 then argv[0] = "foo", argv[1] = "205" and argv[2] = "535".

Important: The argv array should be terminated with NULL. In the example above, argv[3] = NULL. We strongly recommend that you carefully check that you are constructing this array correctly!

2.4 Redirection

Many times, a shell user prefers to send the output of his/her program to a file rather than to the screen. The shell provides this nice feature with the ">" character. Formally this is named "redirection of standard output". Your shell should include this feature.

For example, if a user types "`ls -la /tmp > output`" , nothing should be printed on the screen. Instead, the output of the `ls` program should be rerouted to the output file.

If the "`output`" file already exists before you run your program, you should simply overwrite the file (after truncating it). If the output file is not specified (e.g. "`ls >` "), you should also print an error message. Please look at the "**Defensive Programming and Error Messages**" section below for more information.

Here are some other examples of redirections that should **not** work:

```
ls > out1 out2
ls > out1 out2 out3
ls > out1 > out2
```

Hints:

Redirection is relatively easy to implement: just use `close()` on `stdout` and then `open()` on a file. More on this during discussion.

With a file descriptor, you can perform read and write to a file. It's possible that you have only used `fopen()` and `fread()`. Unfortunately, these functions work on `FILE*`, which is more of a C library support; the file descriptors are hidden.

To work on a file descriptor, you should use `open()`, `read()`, and `write()` system calls. These functions perform their work by using file descriptors. To understand more about file I/O and file descriptors you should read the Advanced UNIX Programming book Section 3 (specifically, 3.2 to 3.5, 3.7, 3.8, and 3.12). Before reading forward, at this point, you should get yourself familiar with file descriptor.

The idea of redirection is to make the stdout descriptor point to your output file descriptor. First of all, let's understand the `STDOUT_FILENO` file descriptor. When a command "`ls -la /tmp`" runs, the `ls` program prints its output to the screen. But obviously, the `ls` program does not know what a screen is. All it knows is that the screen is basically pointed by the `STDOUT_FILENO` file descriptor. In other words, you could rewrite `printf("hi")` in this way: `write(STDOUT_FILENO, "hi", 2)`.

For more information about **close** and **open** which will be essential to achieve this functionality please refer to **Figure 5.4 (page 43)** of the "Operating Systems: Three Easy Pieces" textbook.

2.5 Background Jobs

Sometimes, when using a shell, you want to be able to run multiple jobs concurrently. You might want to start a process and let it run in the background while doing something else. A single "`&`" character will be used to let your shell know the command passed was meant to be run in the background.

For instance, your shell might take a command such as the following:

```
mysh> ls &
```

By typing a trailing ampersand, the shell knows you just want to launch the job, but not to wait for it to complete. Thus, you can start more than one job by repeatedly using the trailing ampersand.

```
mysh> ls &
msyh> ps &
mysh> find . -name *.c -print &
```

Of course, sometimes you will want to wait for jobs to complete. To do this, in your shell you will simply type `wait`. The built-in command wait should not return until all background jobs are complete. In this example, `wait` will not return until the three jobs (`ls, ps, find`) all are finished.

```
mysh> ls &
msyh> ps &
mysh> find . -name *.c -print &
mysh> wait
```

Hints:

For more information about **wait** and **fork** which will be essential for this part of the assignment please refer to **Figure 5.3 (page 41)** of the "Operating Systems: Three Easy Pieces" textbook

2.6 Fun Feature

Your shell has one fun feature: when you type in the name of a python file where a command should be, your shell recognizes this and instead run the python interpreter (e.g., `/usr/bin/python`) with the file name as an argument. For instance, typing:

```
mysh> hello.py
```

would actually run the python interpreter, with `hello.py` as the argument. If there are other arguments present on the command line, they should also be passed to the running python program as arguments.

The format for this command is the following when no arguments are present:

```
[Optional space(s)] file.py [Optional space(s)]
```

The format for this command with arguments is the following. **Note:** There can be arbitrary number of arguments. In this example we assume 1.

```
[Optional space(s)] file.py [One or more Optional spaces] arg1 [Optional space(s)]
```

Hints:

Please refer back to the hint part of the **Non Built-in Commands** section for help with this.

2.7 Defensive Programming and Error Messages

The one and only error message. In summary, you should print this one and only error message whenever you encounter an error of **any** type:

```
char error_message[30] = "An error has occured\n";
write(STDERR_FILENO, error_message, strlen(error_message));
```

The error message should be printed to `stderr` (standard error). Also, **do not** add whitespaces or tabs or extra error messages.

There is a difference between errors that your shell catches and those that the program catches. Your shell should catch all the syntax errors specified in this project page. If the syntax of the command looks perfect, you simply run the specified program. If there is any program-related errors (e.g. invalid arguments to `ls` when you run it, for example), let the program print its specific error messages in any manner it desires (e.g. could be `stdout` or `stderr`).

Defensive programming is required. Your program should check all parameters, error-codes, etc., before it trusts them. In general, there should be no circumstances in which your C program will core dump, hang indefinitely, or prematurely terminate. Therefore, your program must respond to all input in a reasonable manner; by "reasonable", we mean print the error message (as specified up above) and either continue processing or exit, depending upon the situation.

You should consider the following situation as an error; in which case, your shell should print the error message to `stderr` (as specified up above) and **exit** gracefully:

(a) An incorrect number of command line arguments to your shell program.

For the following situation, you should print the error message to `stderr` and **continue** processing:

(a) A command does not exist or cannot be executed.

(b) A very long command line (over 512 characters, excluding the new line character).

Your shell should also be able to handle the following scenarios below, which are not errors. A reasonable way to check if something is not an error is to run the command line in the real Unix shell.

(a) An empty command line.

(b) Multiple white spaces on a command line. All of these requirements will be tested extensively.

2.8 Miscellaneous

Writing your shell in a simple manner is a matter of finding the relevant library routines and calling them properly. To simplify things for you in this assignment, we have suggested a few library routines which you may want to use to make your coding easier. (Do not expect this detailed of advice for future assignments!) You are free to use these routines if you want or to disregard our suggestions. To find information on these library routines, look at the manual pages (using the Unix command man).

Remember to get the basic functionality of your shell working before worrying about all of the error conditions and end cases. For example, first get a single command running (probably first a command with no arguments, such as "`ls`"). Then try adding more arguments.

Next, try working on multiple commands. Make sure that you are correctly handling all of the cases where there is miscellaneous white space around commands or missing commands. Finally, you add built-in commands and redirection supports.

We strongly recommend that you check the return codes of all system calls from the very beginning of your work. This will often catch errors in how you are invoking these new system calls. And, it's just good programming sense.

Beat up your own code! You are the best (and in this case, the only) tester of this code. Throw lots of junk at it and make sure the shell behaves well. Good code comes through testing – you must run all sorts of different tests to make sure things work as desired. Don't be gentle – other users certainly won't be. Break it now so we don't have to break it later.

Keep versions of your code. More advanced programmers will use a source control system such as SVN or GitHub. Minimally, when you get a piece of functionality working, make a copy of your .c file (perhaps a subdirectory with a version number, such as v1, v2, etc.). By keeping older, working versions around, you can comfortably work on adding new functionality, safe in the knowledge you can always go back to an older, working version if need be.

List of helpful functions

The following list are functions that will be helpful for this project

- open
- close
- read
- write
- perror
- execvp
- wait
- waitpid
- fork
- stdup
- fread
- fopen
- fwrite
- fclose
- strtok
- strchr
- fgets
- sscanf
- sprintf

2.9 Mysh (Bonus 10 points)

This command is **optional**. Your shell should accept the command "mysh". When this command is executed your shell should print out the current user name.

The format for the command is the following:

```
[Optional space(s)] mysh [Optional space(s)]
```

Virtual Machine Instructions

1. Please download and install **VirtualBox** from the following link:
   https://www.virtualbox.org/wiki/Downloads
2. Please download the **sec.ova** images from the following link:
   `https://www.dropbox.com/sh/hpgt3tuyev8hlt6/AADkefjEtdzp9mP4YXU1DvFNa?dl=0`
3. Run **VirtualBox**
4. Go to **File/Import Appliance**
5. In the *Appliance to import* window choose *sec.ova* image file and press **Continue**
6. Press **Import** and then the import process will begin
7. After the *Import* process finishes you can see the imported virtual machine (VM) on the left-side window. Select the VM and then press **Start** in the toolbar.
8. Whenever prompted for username and password use the following credential
   - **Username:** sec
   - **Passowrd:** ces
9. After logging in. Go to ICON from the VM and download Assignment 4 found in ICON.