EE 4374 Operating Systems Design
Lab #2: Unix Shell
Due Date: Lecture 1 Week 7 (before Midnight) [Mar. 1, 2016]

Objectives:

1) To learn how to use POSIX system calls in C.
2) To learn how to create processes using the *fork()* system call.
3) To learn how to update the memory image of a process with the *execvp()* system call.
4) To learn how to block a parent process until a child process terminates using the *wait() or waitpid()* system calls.
5) To learn how to overwrite a file descriptor with another file descriptor to redirect standard output to a file with the *dup2()* system call.
6) To learn how to reuse software components (e.g., the argument tokenizer from Lab 1).

Unix Shell:
You will develop a shell. A shell is a program that interprets commands and acts as an intermediary between the user and the operating system. There are several Unix shells available for you to use, some of them are:

1. Korn shell (ksh)
2. TC shell (tcs)
3. Bourne-again shell (bash)

You will make use of the following system calls to develop a shell:

a. **fork**(): int fork();
b. **execvp**(): int execvp(const char *file, char *const argv[]);
c. **wait**() or waitpid(): int wait(status);
d. **close**(): int close(file_descriptor);
e. **open**():int open(const char *path, int flags); and
   int open(const char *path, int flags, mode_t modes);
f. **dup2**(): int dup2(int filedes, int filedes2);
g. **read**(): int read(file descriptor, buffer, n_to_read);
h. **write**(): int write(file descriptor, buffer, n_to_write);

To use these system calls, your program might include the following libraries:

- #include <fcntl.h>
- #include <sys/types.h>
- #include <unistd.h>

Tasks:

1) Unpack the Lab 2 template provided by the instructor into your home directory: ‘tar zxvf student_lab2.tgz’. This will create a directory called ‘student_lab2’, please rename this directory to firstinitiallastname_lab2 using the ‘mv’ command. For example, the instructor would rename the directory by executing ‘mv student_lab2 mmcgarry_lab2’. You should replace student_argtok.c and student_argtok.h with your corresponding files from Lab 1. Next, go into the

directory and rename all of the files from 'student_*' to 'firstinitiallastname_*' much like you renamed the directory.

2) Write a main() function that prints a prompt, accepts input, tokenizes the input using the argument tokenizer from Lab 1, and passes the argument vector to an executeCmd() function with the following prototype:

```
int executeCmd(char **args);
```

This function returns -1 on error, otherwise 0.

Your shell should exit when an 'x' is entered by itself at the prompt.

Your main() function will be in the file named *firstinitiallastname_lab2.c* and the executeCmd() function and any supporting functions will be in a file named *firstinitiallastname_exec.c* and the function prototype for executeCmd() should be in a file named *firstinitiallastname_exec.h.* To help the students, the main() function has been provided by the instructor in the Lab 2 template.

3) Write an executeCmd() function that can execute any program in the foreground or background as well as redirect the output of any program to a file. You can write any number of functions to support executeCmd().

More specifically, executeCmd() should be able to:

2.1) Execute any program with any number of arguments, below are some examples
- *ls*: lists your files in your current directory (format: $ ls)
- *ls -l*: lists your files in long format (format: $ ls -l)
- *ls -a*: lists all files (format: $ ls -a)
- *pwd*: displays current working directory (format: $ pwd)
- *wc*: displays the number of lines, words, and characters in a file (format: $wc filename)
- *mkdir*: make a new directory (format: $mkdir dirName)
- *cat*: creates, concatenates and displays text files

2.2) Same as 2.1 but the program should be run in the background, i.e., the shell will not wait for completion before displaying the prompt again.
- *&*: runs job in background (format: $ emacs lab1.c &)
- Note: The instructor has provided a function in student_exec.c called execBackground() that can be used to determine when a job should run in the background. Additionally, that function will strip the '&' character from the argument vector.

2.3) Allow the output of a running program to be redirected to a file
- >: redirects standard output to the filename after >

(format: $ grep 'if' lab1.c > foo)

4) Use a Makefile to build your program. The Makefile provided in the Lab 2 template only has a 'clean' target. You must add the other targets.

5) Submit the deliverables, indicated below, as a single tarball file named firstinitiallastname_lab2.tgz through Blackboard.

Deliverables:

1) Submit all of the source files in your Lab 2 directory as a single tarball file. You can create this by changing to the directory above your Lab 2 directory and execute 'tar zcvf firstinitiallastname_lab2.tgz firstinitiallastname_lab2'. As an example, the instructor would execute 'tar zcvf mmcgarry_lab2.tgz mmcgarry_lab2'. This file will be submitted through Blackboard. Submissions that do not follow these specifications will be ignored!

Scoring:

Lab grades will be based on four criteria that will determine your overall grade. The first criterion determines if your program compiles and executes correctly. The correct result must adhere to the **Tasks** section. The second criterion determines if your program uses the specified libraries and/or function prototypes as specified in the **Task** section. The last criterion determines if your source code is well-documented and adheres to submission guidelines. Your source code must include (at the top) your name, class section, due date, assigned date, and a small description of your program. Also, every function/method must be commented by specifying the purpose of the input values (if any) and their respective output values (e.g. void foo(int x) /* input: x > 0, output: x = x * 2*/).

Operation/Successful Demonstration	**80%**
Does the program compile? 30%	
Does the program support executing any program? 30%	
Does the program support executing any program in the background? 10%	
Does the program support redirecting the output of any program to a file? 10%	
Adherence to Interface Specification	**20%**
Does your program use the libraries/APIs specified in the lab assignment? 10%	
Does your program use the function prototypes specified in the lab assignment? 10%	
Comments/Adherence to Submission Specification	**10%**
Does your submission adhere to the filename guidelines? 5%	
Is the source code well-documented? 5%	
Lateness	**10%** per day (including weekends and holidays)