CS39002: Operating Systems Lab.
Assignment 1
Floating date: 11/1/2016
Due date: 18/1/2016

Q1. **Distributed Linear Search**

Objective: In this assignment, your will be implementing **Distributed Linear Search (DLS)** using the concepts Linux System calls. In DLS, the array is split up into subarrays and searching the sub-arrays in delegated to other processes (hence distributed).

Submission: Submit a file *dls.c* which can compiled and run. The program should take two arguments as input.

Input: An array of integers to be read from a file a search integer. Path to the file is first argument to the program and the search integer is the second argument.

Input file will have integers separated by whitespace.

Output: Print index of the searched number, or number not found.

Method:

The input array can be read as global array. Each process is given a segment of the array to handle and the number to find. It creates 2 processes and gives one half each of the segment to the 2 child processes to search in. If the size of the segment is small enough (say <= 5), then it searches in the segment by itself. The main process starts with the complete array.

If any process finds the required number, then it sends a signal to the main process and also returns the index. After this, the main process kills all the other processes (Why is this beneficial?) and prints the index.

Your code should also handle the case when the number is not present in the array.

Q2. **Basic Shell**

Objective: The shell is a program that interprets commands and acts as an intermediary between the user and the inner workings of the operating system and as such is arguably one of the most important parts of a Unix system.

In this assignment, we shall start making our very own version of a Unix shell.

Submission: Write a simple shell in C. Call it *simplesh.c.*

The shell should implement following features:

- The shell should run continuously, and display a prompt (>, similar to $) when waiting for input. Include the full path of the current directory also in the prompt followed by the “>” sign (for ex., /usr/home/me/Desktop>).
- The shell should read a line from input one at a time.
- After parsing and lexing the command, the shell should execute it.
- Implement the following Built-in Commands,
- clear: clear the screen
- env: displays all environment parameters
- cd <dir>
- pwd: prints the current directory
- mkdir <dir>: creates a directory called "dir"
- rmdir <dir>: removes the directory called "dir"
- ls: lists files in current directory (ls –l: option also need to be supported)
- history: displays the last commands the user ran, with an offset next to each command. Last commands can be stored in a file and may be displayed to user when the command is issued.
- history <argument>: displays the given number of commands as specified
- exit: exits the shell

The commands are the same as the corresponding Linux commands by the same name. Do "man" to see the descriptions. You can use the standard C library functions chdir, getcwd, mkdir, rmdir, readdir, stat etc. to implement the calls.

All calls should handle errors properly, with an informative error message. Look up the perror call.

These commands are called builtin commands since your shell program will have a function corresponding to each of these commands to execute them; no new process will be created to execute them. (Note that all these commands are not builtin commands in the bash shell, but we will make them so in our shell).

● Any other command typed at the prompt should be executed as if it is the name of an executable file. For example, typing "a.out" should execute the file a.out. The file can be in the current directory or in any of the directories specified by the PATH environment variable (use getenv to get the value of PATH). The file should be executed after creating a new process and then executing the file onto it. The parent process should wait for the file to finish execution and then go on to read the next command from the user. The command typed can have any number of command line arguments.
● Support background execution of commands. Normally when you type a command at the shell prompt, the prompt does not return until the command is finished. For background executions, the prompt returns immediately, the command continues execution in the background. Typing an "&" at the end of a command (for ex., a.out&) should make it execute in the background. (Note: Background execution needn’t be supported for builtin commands).