## Programming Exercise E - Parallel Search using Threads

### Assignment Objectives

* Practice implementing source code that is part of a larger software development involving multiple source code files
* Practice using POSIX threads as a solution for solving a parallel searching problem

### Assignment Summary

In this assignment, you will complete the implementation of a partially-written program. Given a word search puzzle and a list of words, the program will search through the puzzle and print out the list of words and their locations in the puzzle. The program will be able to perform both a sequential search and a parallel search of the puzzle. The parallel search will be performed using POSIX threads.

Your finished source code, when combined with the provided source code, should compile, link and run in a POSIX standard environment with no error messages. It should also run correctly when given expected input.

### Program Summary

The program looks for four entries on the command line: the program name, the search type (-s for sequential or -p for parallel), a word puzzle file name, and a word list file name. This is illustrated in the following usage message:

Usage: a.exe search\_type puzzle\_file\_name word\_list\_file\_name

search\_type : -s for sequential search, -p for parallel search

Minimum puzzle size: 4 rows and 4 columns  
Maximum puzzle size: 100 rows and 100 columns  
Maximum number of words: 100  
Maximum word size: 40

The format of the word puzzle file consists of two integers on the first line representing the number of rows and columns. This line is followed by rows of upper-case characters representing the word puzzle. The characters start at the beginning of the line and are separated from each other by one blank space. For examples of this format, refer to the **furniture-puzzle.txt** and **state-names-puzzle.txt** files.

The format of the word list file consists of one integer on the first line representing the number of words in the file, followed by one word per line. The words contain all upper-case letters and contain no blank spaces. For example, "Hall stand" appears as "HALLSTAND". For examples of this format, refer to the **furniture-list.txt** and **state-names-list.txt** files.

The program searches the puzzle for each of the words in the list. **Four** different kinds of searches occur: horizontally to the right, horizontally to the left, vertically down, and vertically up. These searches are performed either sequentially or in parallel depending on the command line option.

After completing the search, the program prints the list of words. It follows each word with its row and column starting position and its direction in which the word extends in the puzzle. In the program output, the first row or column is referred to as row 1 or column 1. For examples of this format, refer to the **furniture-locations.txt** and **state-names-locations.txt** files.

### Assignment Directions

1. As stated earlier, the program is only partially completed. It consists of three files described below. Download each of these files from the assignment page on Blackboard:
   * **program-driver.c** - This source code file contains the code that starts the program, parses the command line, opens the two files, reads the puzzle file, reads the word list file, and displays the search results. In addition, it contains the code that calls the functions to perform a sequential search or a parallel search of the puzzle. Make no changes in this file.
   * **puzzleSearcher.h** - This source code header file contains #define constants, a type definition, and function prototypes. Make no changes to this file.
   * **puzzleSearcher-module-student.c** - This source code file contains #define constants, static global variables, static function prototypes, and global and static function definitions. The **searchHorizontallyRight()** function contains the complete algorithm to search for words that run from left to right in the puzzle. You may copy and modify this algorithm as needed to do other kinds of searches in the other functions. As an alternative, you may design and implement your own search algorithm as long as the function prototypes and the reading from and assignment to the global data strutures remain the same. Also, no other data structures should be created.
2. Change the name of the search-module-student.c file to search-module.c
3. Study the source code in each of the three files to get an understanding of how the program works. Especially notice the source code for the following functions:
   * **void performSequentialSearch(void)**
   * **static void \*searchHorizontallyRight(void \*parameter)**

These functions contain the source code to perform a sequential search horizontally to the right; consequently, the program in its current condition performs only a single kind of sequential search. These two functions serve as a guide for you in coding the remaining functions needed in the file.

1. Complete the definition of the **performSequentialSearch()** function so that it contains function calls to search three other directions: horizontally to the left, vertically down, and vertically up. You may want to do this by creating a copy of the searchHorizontallyRight() function call and making the necessary name changes to adapt to the other three kinds of search. Below are names and prototypes for you to use for the other three functions. (Note: Although these function prototypes appear to look like threads, this is only for convenience so that they can be used that way later. Do not create any threads at all when performing the sequential search.) :
   * **static void \*searchHorizontallyLeft(void \*parameter)**
   * **static void \*searchVerticallyDown(void \*parameter)**
   * **static void \*searchVerticallyUp(void \*parameter)**
2. Complete the definition of the **performParallelSearch()** function. This function shall perform a parallel search of the puzzle by assigning each of the four search functions (i.e., directions) to a separate POSIX thread. (When creating each thread, use NULL as the value passed for the thread argument.) The performParallelSearch function shall then join each thread (i.e., wait for each thread to end) after all threads have been started.
3. If you haven't done so already, complete the definitions of each of the **three** other search functions listed above in Step #4 (i.e., horizontally to the left, vertically down, and vertically up) The code in these search functions should not contain any output statements (e.g, no printf(), etc.)
4. When compiling and linking the files, you may need to designate the pthread library on the command line depending on the default libraries that your development environment uses. This is shown below:

gcc program-driver.c searcher-module.c

1. Perform tests on the program using the data in the two sample puzzle files
2. Only submit your search-module.c file to Blackboard

### Optional Additions to the Program

In addition to the four horizontal and vertical search functions listed above, complete the definitions of the **four** diagonal search functions listed below. These functions shall also be called by the performSequentialSearch() and performParallelSearch() functions.

* + **static void \*searchDiagonallyDownRight(void \*parameter)**
  + **static void \*searchDiagonallyUpLeft(void \*parameter)**
  + **static void \*searchDiagonallyUpRight(void \*parameter)**
  + **static void \*searchDiagonallyDownLeft(void \*parameter)**

### Design and Implementation Constraints

* Follow the same coding standards given in the previous assignments.