Assignment 2

Available Date: Sunday, 23 October 2016
Due Date: 11.50PM, Sunday, 13 November 2016
Total Mark: 100 marks

Part I [20 marks]
Write a complete C program named **`p1.c`** to perform matrix multiplication by using threads. Given two matrices, *A* and *B*, where matrix *A* contains *m* rows and *n* columns and matrix *B* contains *r* rows and *s* columns, the **matrix product** of *A* and *B* is matrix *C*, where *C* contains *m* rows and *s* columns. The entry in matrix *C* for row *i*, column *j* ($C_{i,j}$) is the sum of the products of the elements for row *i* in matrix *A* and column *j* in matrix *B*. That is,

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \times B_{k,j}$$

For example, if *A* is a 3-by-2 matrix and *B* is a 2-by-3 matrix, element $C_{2,3}$ is the sum of $A_{2,1} \times B_{1,3}$ and $A_{2,2} \times B_{2,3}$.

For this assignment, calculate each element $C_{i,j}$ in a separate child (or ***worker***) thread. This will involve creating $m \times s$ worker threads. The parent thread (i.e., **`main()`** function) will input the matrices *A* and *B* and allocate sufficient memory for each entry $C_{i,j}$ of matrix *C*, which will hold the product of matrices *A* and *B*. These matrices will be declared as global data so that each worker thread has access to *A*, *B*, and *C*.

Matrices *A* and *B* can be initialized statically, as shown below:

```
#define M 3     // m = M
#define K 2     // n = r = K
#define N 3     // s = N
int A[M][K] = {{1, 4},
                {2, 5},
                {3, 6}};
int B[K][N] = {{8, 7, 6},
                {5, 4, 3}};
int C[M][N];
```

Passing Parameters to Each Thread

The parent thread will create $m \times s$ worker threads, passing each worker the values of row *i* and column *j* that are used in calculating the matrix product (i.e., calculating entries $C_{i,j}$). This requires passing two parameters to each thread. The easiest approach with Pthreads is to create a data structure using a `struct`. The members of this structure are *i* and *j*, and the structure appears as follows:

```
/* structure for passing data to threads */
struct v {
   int row; /* row */
   int col; /* column */
};
```

The Pthreads program will create the worker threads using a strategy similar to that shown below:

```
/* We have to create M * N worker threads */
for (i = 0; i < M, i++)
{
    for (j = 0; j < N; j++ )
    { // allocate memory for each entry C[i][j]
        struct v *data= (struct v *) malloc(sizeof(struct v));
        data->row = i;
        data->col = j;
        /* Now create child thread passing it data as a parameter
        pthread_create(&tid, &attr, function name, param);
        // now wait for the child thread to exit
        pthread_join(tid, NULL); */
    }
}
```

The `data` pointer will be passed to the **`pthread_create()`** function, which in turn will pass it as a parameter to the function that is to run as a separate child thread.

Waiting for Threads to Complete

Once all child threads have completed, the parent thread will output the product contained in matrix *C*. This requires the parent thread to wait for all worker threads to finish before it can output the value of the matrix product. Pthreads provides the **`pthread_join()`** function to enable a parent thread to wait for other child threads to finish.

A simple strategy for waiting on several threads using the Pthreads **`pthread_join()`** is to enclose the join operation within a simple `for` loop. For example, you could join on ten threads using the array of worker threads `workers` depicted next.

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

• An execution example of the executable program **`p1.c`** is given below.

```
$ ./p1
Enter the number of rows: 3
Enter the number of columns: 2
Element[0][0] = 1
Element[0][1] = 4
```

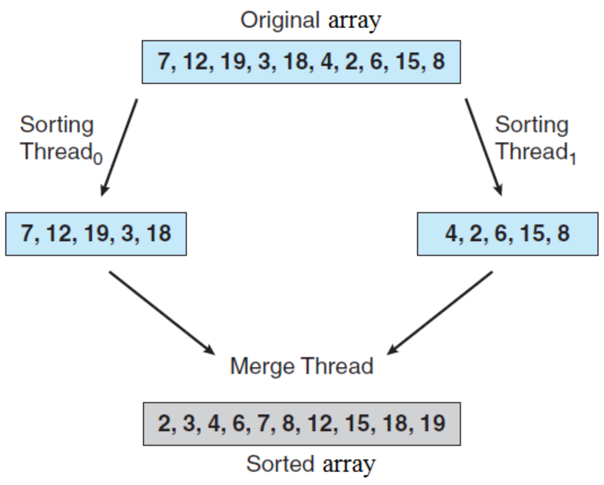
```
Element[1][0] = 2
Element[1][1] = 5
Element[2][0] = 3
Element[2][1] = 6
1       4
2       5
3       6

Enter the number of rows: 2
Enter the number of columns: 3
Element[0][0] = 8
Element[0][1] = 7
Element[0][2] = 6
Element[1][0] = 5
Element[1][1] = 4
Element[1][2] = 3
8       7       6
5       4       3

Result for Matrix Multiplication
28      23      18
41      34      27
54      45      36

$
```

Part II [30 marks] Multithreaded Sorting Application
Write a multithreaded sorting program named **`p2.c`** that works as follows. An *n*-element array of integers is divided into two smaller arrays by using a middle position. Two separate child threads (which we will term sorting threads) sort each subarray using a selection sort algorithm. The two subarrays are then merged by a third child thread (called a merging thread) which merges the two sorted subarrays into a single sorted array.

Because global data are shared across all threads, perhaps the easiest way to set up the data is to create a global array. Each sorting thread will work on one half of this array. A second global array of the same size as the unsorted integer array will also be established. The merging thread will then merge the two sorted subarrays into this second array. Graphically, this program is structured according to the following figure.

This program will require passing parameters to each of the sorting threads. In particular, it will be necessary to identify the start and end indices from which each thread is to begin sorting. Refer to the instructions in Part I for details on passing parameters to a thread.

The parent thread will output the sorted array once all child threads have finished.



An example of multithreaded sorting

• An execution example of the executable program `a22` is given below.

```
$ ./p2
The given array is
7 12 19 3 18 4 2 6 15 8
Before selection sort of the first half
7 12 19 3 18
Before selection sort of the second half
4 2 6 15 8
After selection sort of the first half
3 7 12 18 19
After selection sort of the second half
2 4 6 8 15
After being merged, the sorted array is
2 3 4 6 7 8 12 15 18 19

$
```

Part III [50 marks] Sudoku solution validator
A Sudoku puzzle uses a 9×9 grid in which each column and row, as well as each of the nine 3×3 subgrids, must contain all of the digits 1, 2, …, 9. An example of a valid Sudoku puzzle is shown in the figure below. You are asked to design a multithreaded program named **`p3.c`** that determines whether the solution to a Sudoku puzzle is valid.

There are several different ways of multithreading this application. One suggested strategy is to create threads that check the following criteria.

- A thread to check that each column contains the digits 1 through 9
- A thread to check that each row contains the digits 1 through 9
- Nine threads to check that each of the 3×3 subgrids contains the digits 1 through 9

6	2	4	5	3	9	1	8	7
5	1	9	7	2	8	6	3	4
8	3	7	6	1	4	2	9	5
1	4	3	8	6	5	7	2	9
9	5	8	2	4	7	3	6	1
7	6	2	3	9	1	4	5	8
3	7	1	9	5	6	8	4	2
4	9	6	1	8	2	5	7	3
2	8	5	4	7	3	9	1	6

The solution to a 9×9 Sudoku puzzle

This would result in a total of eleven separate threads for validating a Sudoku puzzle. However, you are welcome to create even more threads for this application. For example, rather than creating one thread that checks all nine columns, you could create nine separate threads and have each of them check one column.

Passing Parameters to Each Thread

The parent thread will create the worker threads, passing each worker the location that it must check in the Sudoku grid. This step will require passing several parameters to each thread. The easiest approach is to create a data structure using a **`struct`**. For example, a structure to pass the row and column where a thread must begin validating would appear as follows:

```
/* structure for passing data to threads */
struct v {
   int row; /* row */
   int col; /* column */
};
```

The Pthreads program will create the worker threads using a strategy similar to that shown below:

```
struct v *data= (struct v *) malloc(sizeof(struct v));
data->row = i;
data->col = j;
/* Now create child thread passing it data as a parameter
```

The `data` pointer will be passed to the **`pthread_create()`** function, which in turn will pass it as a parameter to the function that is to run as a separate child thread.

Returning Results to the Parent Thread

Each worker thread is assigned the task of determining the validity of a particular region of the Sudoku puzzle. Once a worker has performed this check, it must pass its results back to the parent. One good way to handle this is to create an array of integer values that is visible to each thread. The *i*th index in this array corresponds to the *i*th worker thread. If a worker sets its corresponding value to 1, it is indicating that its region of the Sudoku puzzle is valid. A value of 0 would indicate otherwise. When all worker threads have completed, the parent thread checks each entry in the result array to determine if the Sudoku puzzle is valid.

Assignment Requirements

1. Implement your programs on the UNIX-like platform such as Ubuntu Linux by using the C programming language.
2. The source codes **MUST** be appropriately commented and structured to allow users to understand your codes easily.
3. Your programs should be compiled without any error.
4. Do not hand in any binary (i.e., executable program) files. The submission should contain only the source code files stored in the A2.zip file.
5. At the top of your source programs, include the following information.

```
/*
Student ID:
Full Name:
Email:
Course Code:
*/
```

6. Send your source programs stored in the A2.zip file to the email: uwicomp3100@hotmail.com.