**EE 463** **Operating Systems** Date: 10/06/1444
**Winter 2023** **Homework #3** 03/01/2023

**Dr. A. M. Al-Qasimi** **Due: 10/01/2023**

**Objectives:**
The purpose of this homework is to get hands-on experience with multithreaded programming, using Java thread API library, under the Windows operating system.

**Description**:
Your assignment consists of designing and programming a multithreaded solution to perform a fast matrix transposition (assuming that you have enough CPU/GPU cores). Write your multithreaded program in Java, using the Java thread API library.

Calculating the transpose of a matrix is essential to many important applications. Given matrix $\boldsymbol{A}$, of $M$ rows and $N$ columns, its transpose is the matrix $\boldsymbol{A^T}$ of $N$ rows and $M$ columns, where $\boldsymbol{A^T}$ can be calculated by copying the element in row $i$, column $j$ of matrix $\boldsymbol{A}$ (i.e. $\boldsymbol{A}_{i,j}$) to row $j$, column $i$ of matrix $\boldsymbol{A^T}$ (i.e. $\boldsymbol{A^T}_{j,i}$). That is,

$$A^T_{j,\,i} = A_{i,\,j}\ ; \quad i = 0\,..\,M-1, \quad j = 0\,..\,N-1$$

In a system that have *only* one CPU core it takes ($M$ * $N$) copying operations to get the resulting matrix $\boldsymbol{A^T}$. But we can do much better if more CPU cores were available, for example we can assign one-element copying operation to each core, then finding $\boldsymbol{A^T}$ would take the time of one scalar copying operation. However, this means each core does a trivial operation with high overhead. To reduce system overhead one can have coarser granularity by assigning one-vector copying operation to each core, instead.

For better efficiency and scalability, you should design your algorithm such that it processes one-vector per core (i.e. one row or one column is processed in each separate ***worker*** thread). This involves creating $M$ or $N$ worker threads, whichever is nearest to the number of cores you have. The ***main*** thread shall initialize matrix $\boldsymbol{A}$ and allocate sufficient memory for matrix $\boldsymbol{A^T}$. These matrices must be declared as *global* data so that each worker thread shall have access to both $\boldsymbol{A}$ and $\boldsymbol{A^T}$.

Matrix $\boldsymbol{A}$ shall be populated by reading-in values from a file. Then, the ***main*** thread shall call the ***worker*** threads and pass to each (***worker***)$_i$ or (***worker***)$_j$, the value of row $i$ and its *size* or the value of column $j$ and its *size* respectively, which the worker thread should use in its operation. This requires passing two parameters to each thread. Because each worker thread can do its operation and store the corresponding element(s) of $\boldsymbol{A^T}$ **independent** of the other worker threads, we can expect to have a correct solution without the need to setup any *shared memory protection* or *synchronization* between the threads.

**Waiting for Threads to Complete**
During the execution of the worker threads, the ***main*** thread will have to wait for all the workers to finish before continuing. Once all ***worker*** threads have completed, the ***main*** thread shall output $\boldsymbol{A^T}$.

Read the Java multi-threaded example given in the lecture; the Java Thread class and the Runnable interface documentation to learn how to create threads, pass the required parameters to them and how to make the main thread wait until the other worker threads are finished.

**What to do**:
Write a multithreaded program in Java under Windows using the Java user-thread API library to calculate the matrix transpose as described above. Your program should read one command-line parameter which is the name of a file that contains the following data:

1. In the first line, two integers indicating the dimensions of $\boldsymbol{A}_{MxN}$ : $M$ for rows and $N$ for columns.
2. The matrix $\boldsymbol{A}$ values, one row per line (i.e. $M$ lines, each with $N$ values).
3. This means that the file must contain $2+(M \times N)$ values in $1+M$ lines.

The program (***main*** thread) must setup the shared matrices: $\boldsymbol{A}$, and $\boldsymbol{A^T}$, read the command line parameter, check and read the values in the input file, populate and print out $M$, $N$, and matrix $\boldsymbol{A}$ then it shall create the ***worker*** threads and wait for all of them to finish. Each ***worker*** thread must write one output line identifying itself in terms of the parameters passed to it and printing what it is doing. The ***main*** thread shall then print the resulting matrix $\boldsymbol{A^T}$ and terminate. Your program must check for all possible errors and if there are some errors, it shall attempt correcting them by interacting with the user or terminate gracefully after printing a useful error message.

**What to turn in**:
A single .zip file that contains ONLY three files:

a) Your Java source program, fully documented and commented,
b) Your test data file as described above, and
c) Your main program output including: $M$, $N$, $\boldsymbol{A}$, $\boldsymbol{A^T}$; and the output generated by the worker threads.

**Sample Output**:

```
M = 6, N = 3

Thread T1 operates on column #1 of A, Writes 6 values to row #1 of AT.
Thread T3 operates on column #3 of A, Writes 6 values to row #3 of AT.
Thread T2 operates on column #2 of A, Writes 6 values to row #2 of AT.

A =
 1.00  3.00  7.00
 3.00  2.00 11.00
 2.00  4.00 13.00
 3.00  4.00  6.00
 2.00  3.00  1.00
 0.00  1.00 22.00

AT = A Transposed
 1.00  3.00  2.00  3.00  2.00  0.00
 3.00  2.00  4.00  4.00  3.00  1.00
 7.00 11.00 13.00  6.00  1.00 22.00
```

**Notes**:
1) You must **test your program thoroughly** before submitting it. It should have no errors,
2) Make sure that your program can handle **small**, **huge** and **erroneous** inputs without problems,
3) Any **programs that do not work** properly will get a **zero** mark. No excuses,
4) Your submitted file must be a **.zip** file named as your name (i.e. `FirstInitial_LastName.zip`).