# Operating Systems and Concurrency

## Week 3 lab exercises

---

### Introduction

In the previous set of exercises, you solved an overall task that could be split into independent subtasks using multithreading. This involved creating separate lists of numbers, each of which could be processed by a different thread. In this case, the threads didn't actually share any data with each other. They were all oblivious to the task being performed by other threads. However, the result of all the threads could be assembled into one final result. So, one approach to parallelising a task is to divide the input up ahead of time and to ask each thread to operate on independent partitions of the data.

Dividing the data up this way might be an unnecessary, expensive or impossible task. In this week's tasks you will look at two other ways of sharing data that don't explicitly divide the data. This means there will be literal sharing of the data but other ways will be found to ensure that threads don't interfere with each other.

In task 1 you will share the data between threads but build in a mechanism for ensuring threads don't interfere with each other. In task 2 you will achieve this goal using Java synchronization.

### Task description

The task that you will be performing is parallel matrix multiplication of square matrices of `double`-precision values[1][2]. This is another example of an embarrassingly parallel task. However, with this week's exercises there will be only one buffer of matrix data that will be shared between threads.

The buffer that you will create is to be organised as follows. All of the matrices to be multiplied are to be stored in the same buffer. The buffer is conceptually (but not literally) divided into two halves. The first half of the buffer contains the left-hand operands in the multiplication, and the second half contains that right-hand operands.

The operands are to be selected for multiplication as follows. If the buffer contains $N$ matrices, this means there will be $N/2$ multiplication operations to be performed. Each left-hand operand matrix $m_n$ will be multiplied by the matrix $m_{n+N/2}$. Here $N/2$ is also the offset that gives the position of the right-hand matrix.

For example, if the buffer contains the following matrices:

$\{M1, M2, M3, M4, M5, M6, M7, M8, M9, M10\}$

The buffer length $N$ is 10 hence the offset into the buffer for the right-hand matrix is $N/2 = 5$.
This means the above buffer represents the following required multiplications:

[1] You can make your program work on rectangular matrices, if you wish, but that's not a requirement for the task.
[2] You may need to refresh your memory about how matrix multiplication works.

$M1 \times M6$
$M2 \times M7$
$M3 \times M8$
$M4 \times M9$
$M5 \times M10$

## Task 1

For this task, you should create a program that can multiply (as described above) the matrices in a buffer.

### The matrix buffer

The buffer can be implemented as an array of matrices, i.e. a three-dimensional array of values of type `double`. The dimensions of the matrices will need to be specified. I suggest at first trying $2 \times 2$ matrices. You can increase the size later. You may need to experiment with how many matrices you can create, but around $2,000,000$ might be achievable. For example, using the values here, the following statement will create such a buffer:

```
double[][][] buffer = new double[2000000][2][2];
```
[3]

This creates an array of length $2,000,000$, each element of which is a 2D array of dimensions 2x2.

The matrices will need to be filled with values. For this I recommend simply assigning them random elements or making them all the identity matrix. If you use the latter it's much easier to see if your actual multiplication code works.

It is advisable to encapsulate the buffer in its own class. That way the buffer can easily be asked to initialise itself. You would also need to think about how access is given to the actual buffer.

You can decide for yourself how to implement this but keep to the requirement that this is a single buffer that contains all the matrices.

### The `Runnable`

Each thread (`Runnable`) will be passed (a reference to) the entire matrix buffer in its constructor. It will also be told, via its constructor, which section of the buffer it should process (this is simply an index into the buffer) and how many matrices it should process.

This means that although the threads are sharing the same buffer, they should not interfere with each other because each of them should be told to work on a different section of the buffer, by passing them a different index.

For this task, when a thread has completed its multiplications it will terminate. This means that full coverage of the buffer is required when creating the threads. This also means that the number of matrices each thread will need to process depends on how many threads are created to perform the task.

Threads (`Runnables`) will need a way of storing their results for later retrieval.

[3] Of course, I don't recommend using magic numbers like this but using constant declarations.

### The program

Write a program that creates the buffer, creates and starts the threads, and then waits for them to terminate. Once the threads have terminated the results of each thread could be gathered. You don't need to print out the results (since there will be rather a lot of them) but you should check somehow that the multiplications have been performed correctly.

You should add some code to your program (as last week) to monitor the run time of the threads. This will involve using `System.nanoTime()` again.

### Discussion points

1. What effect on the run time does increasing the number of threads have? For example, what is the difference between having 1,000 threads and 10,000? What reasons could you give for the these results?

2. What is the effect on the run time of changing the dimensions of the input matrices? What is the difference between having dimension = 4 and dimension = 8? Is this what you would expect?

3. Does making each thread sleep for a short time (e.g. 10ms) make any difference?

## Task 2

For this task you should create a new set of classes based on those in Task 1 so that you can change them to create the behaviour described below.

The key difference for this task is that the buffer will be shared but the threads will not be told in advance which sections of the buffer they will work on. They will, however, always each be given the same number of matrices to work on when they *are* given some to work on, however.

Another difference is that the threads themselves should continually request sections of the buffer to work on until all of the multiplications have been performed. When a thread requests a buffer section, but there are no sections left to work on, it should terminate.

Since the threads do not know which section of the buffer they will work on in advance, they will need to *request* a section from the buffer. The buffer class will require a method that allows threads to request a section of the the buffer. In effect, this will simply be an index into the buffer. The buffer will need to synchronize access to this index otherwise threads will interfere with each other. Think carefully about what it is that actually needs synchronizing here.

The buffer class should be able to tell threads the *size* of the buffer section they should work on. This can be a fixed size. A method could be provided to do this.

Threads will need an internal results buffer into which they put their cumulative results. An open question here is what size should the threads' internal results buffer be? You can think about that as part of your solution. This implies, however, that (depending on the implementation) a thread's internal results buffer might get full. This would need dealing with too.

Again, you should add some code to your program (as last week) to monitor the run time of the threads.

**Discussion points**

1. What is the effect on the run time of having different numbers of threads?
2. What is the effect on the run time of different buffer section sizes? What happens if you make the sections very small or very large?
3. Does making each thread sleep for a short time (e.g. 10ms) make any difference?