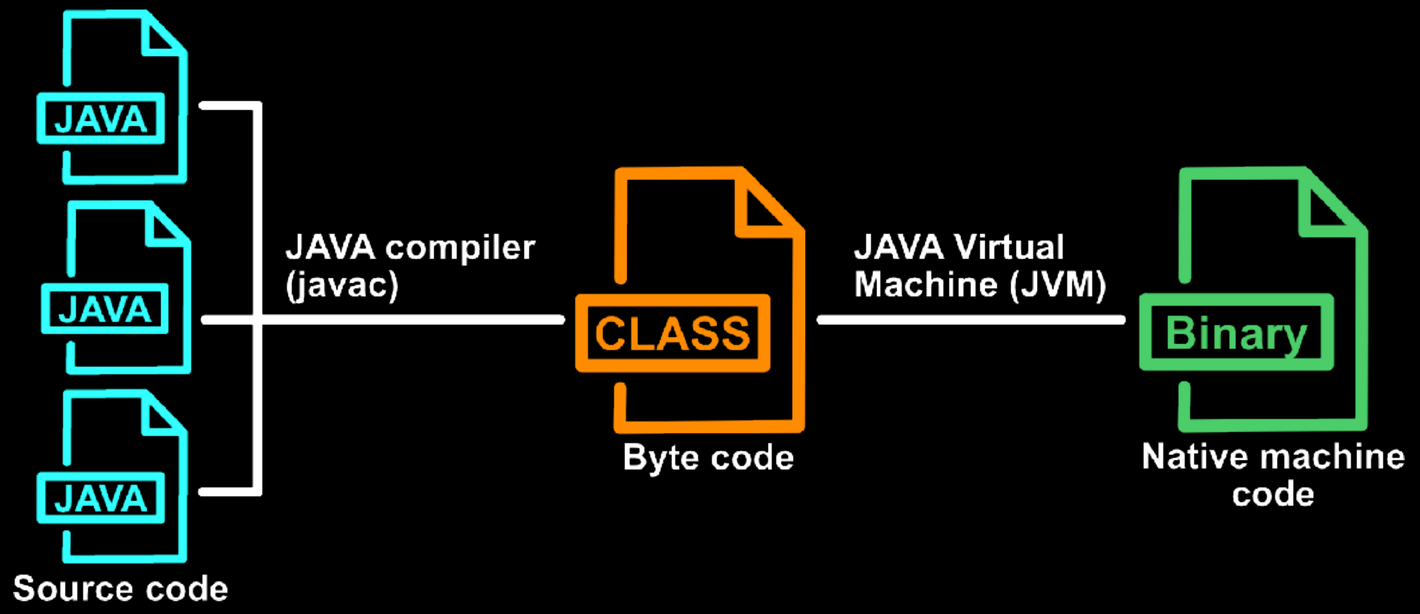
Parallel Programming in Java

“Java is still not dead—and people are starting to figure that out.”

Neco Kriel (n9702806)

Java Programming
The Environment

- Pure Object Orientated Programming (OOP)
- Java compiled code can run on any Java supported platform



Java Programming
Designing Parallel Programs

- The *`java.util.concurrent`* package provides support for concurrency
- *`Locks`* are provided to restrict access to protected resources
    - Code protected by the same lock can only be executed by one thread at a time

```
public synchronized void critcalSection() {
    // critical code goes here
    //
    //
}
```

```
public int add(int val_1, int val_2) {
    synchronized (this) {
        return val_1 + val_2;
    }
}
```

Code Snippets: the keyword *`synchronized`* protects a block of code.

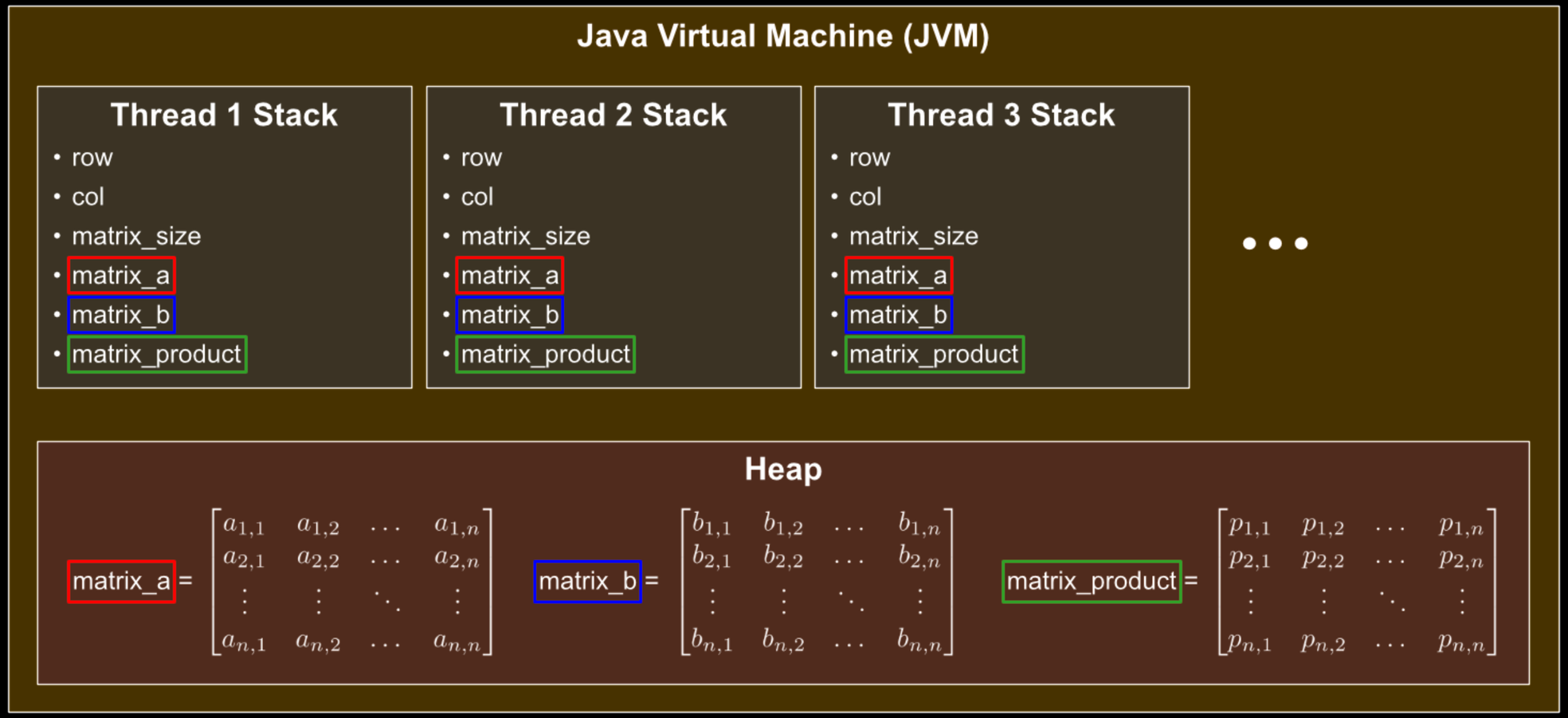
Java Programming

The Java Memory Model

- Resources shared by threads should be protected and the operations on them should be *`atomic`*
- Primitive variables that are local to a thread are stored on the thread stack and are invisible to other threads
- Threads can only pass a ***copy*** of primitive variables to other threads

Java Programming

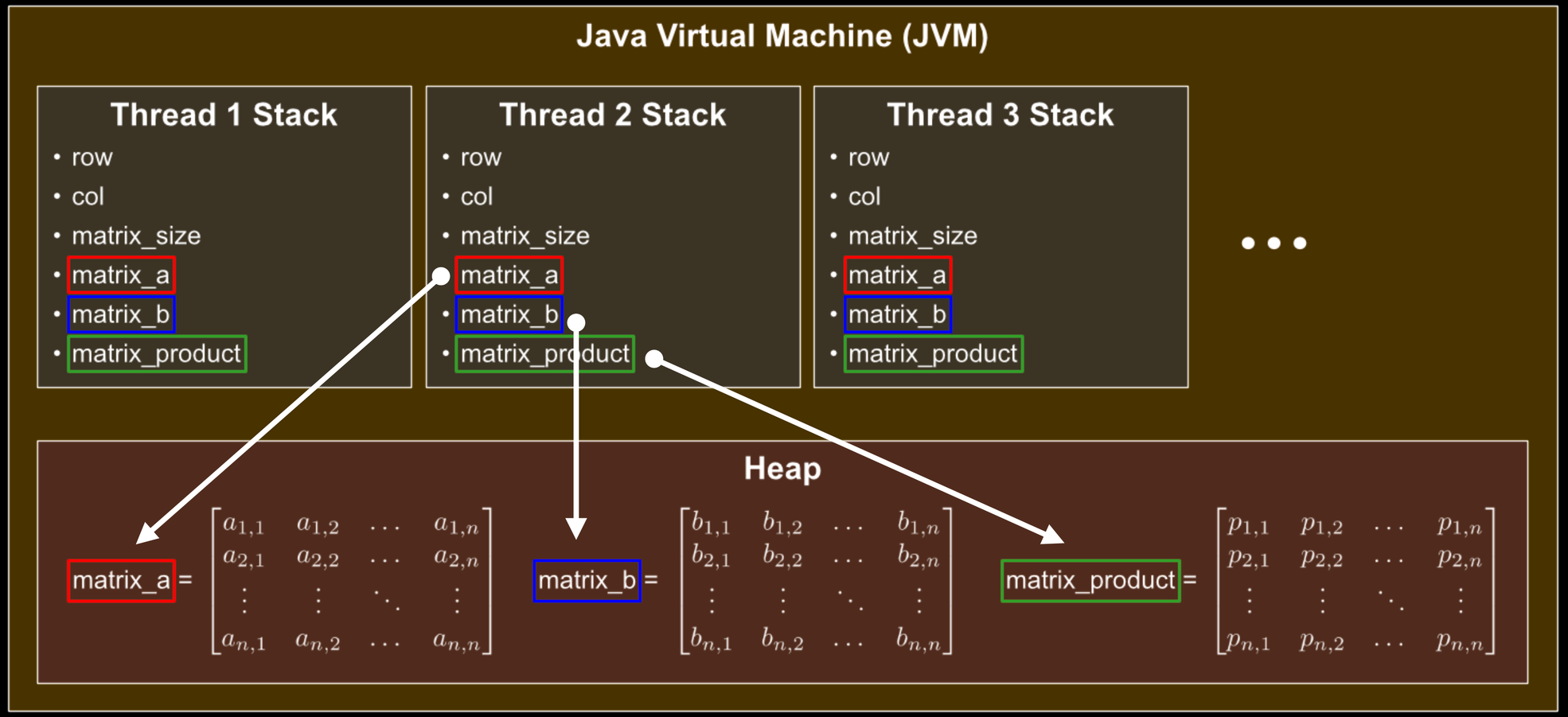
Memory Management



The Java Memory Model

Java Programming

Memory Management



Static variables are stored on the heap.

Useful Data Structures

Atomic Operations

- *`Atomic operation`*: a single unit of work that can't be interrupted
- Reading a variable is atomic (i.e. *`int x = 5`*)
- Incrementing isn't an atomic operation
  - *`x++`* is short hand for: *`x = x + 1`*
- Atomic variables:
  - *`AtomicInteger`*, *`AtomicLong`*, ect
- provide atomic method like:
  - *`get()`*, `set()`, *`getAndIncrement()`*, *`compareAndSet()`*, ect

Useful Data Structures

Complexities of Atomic Operations

- Only one thread can successfully perform an atomic set operation on the same shared resource at a particular instant
- No other threads are suspended (i.e. like in the case where locks are used), however, instead they are informed that they were unsuccessful
- Complexity comes in handling the scenario where an atomic operation was unsuccessful

Java Programming
Parallelism with Multithreading - Defining the Task

- Two main ways of defining a task to be executed by multiple threads
  - Implementing the *`Runnable`* interface
    - Threads share the same object instance
  - Extending the *`Thread`* class
    - Each thread creates a unique copy of each object
- It is good practice to implement the *`Runnable`* interface
  - A class can only extend one other class in Java (i.e. multiple inheritance is not supported)

Java Programming

Parallelism with Multithreading - Executing the Task

- The Java *`ExecutorService`* abstracts away the low-level complexities associated with **scheduling**, **executing** and **managing** tasks
  - Creates a re-usable pool of threads for running submitted tasks
  - Ensures predictability by letting you define the maximum thread-pool size (i.e. number of threads that can run concurrently)

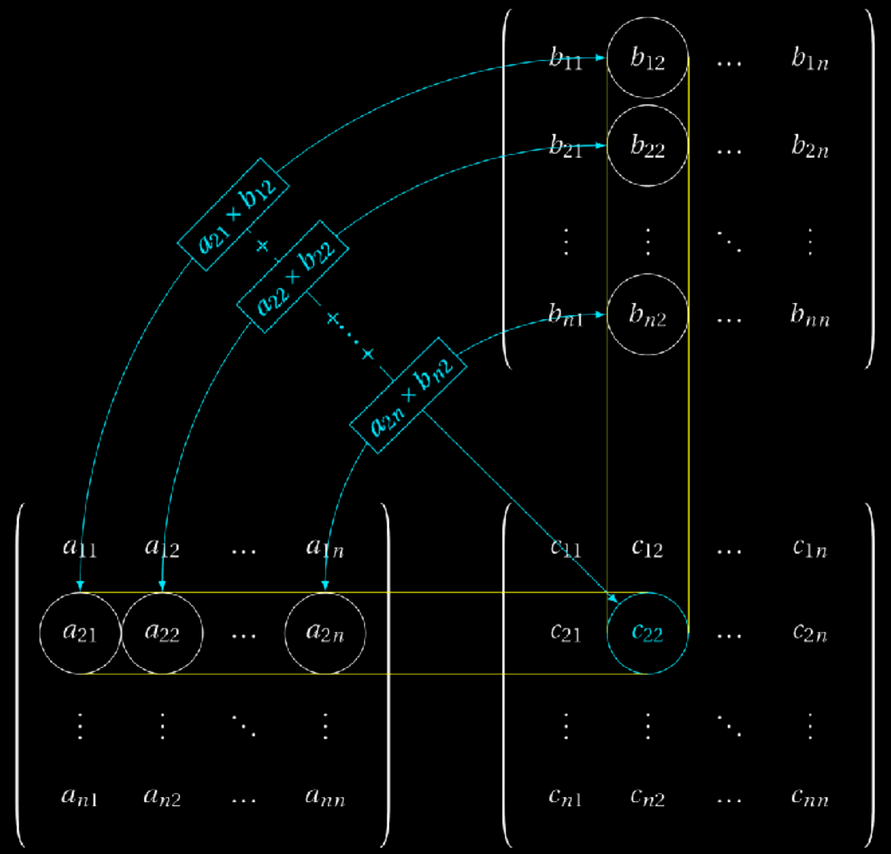
Parallelism with Multithreading

Example: Matrix Multiplication

- Compute each inner-product concurrently

```
for Each matrix A row i do
    for Each matrix B column j do
        Compute the inner-product of matrix A's row i and matrix B's
         column j;
    end
end
```

Algorithm 1: Calculating the product of matrices A and B



Parallelism with Multithreading

Example: Matrix Multiplication

- Parallelise the inner-loop: compute each inner-product concurrently

```
// Send the information for matrix multiplication in parallel
public static void ParallelMatrixDistributor(ExecutorService executor, int matrix_size,
        AtomicIntegerArray[] matrix_a, AtomicIntegerArray[] matrix_b,
        AtomicIntegerArray[] matrix_product) {
    // for each matrix row
    for (int i = 0; i < matrix_size; i++) {
        // and column elemnt in the row
        for (int j = 0; j < matrix_size; j++) {
            // calculate the product result for the PRODUCT[row, column] value
            ParallelMultiplier multiplyThread = new ParallelMultiplier(i, j, matrix_size, matrix_a[i], matrix_b, matrix_product);
            // submit the thread to pool for row-wise multiplication
            executor.execute(multiplyThread);
        }
    }
}
```

Parallelism with Multithreading

Example: Matrix Multiplication

- Parallelise the inner-loop: compute each inner-product concurrently

```
// Send the information for matrix multiplication in parallel
public static void ParallelMatrixDistributor(ExecutorService executor, int matrix_size,
        AtomicIntegerArray[] matrix_a, AtomicIntegerArray[] matrix_b,
        AtomicIntegerArray[] matrix_product) {
    // for each matrix row
    for (int i = 0; i < matrix_size; i++) {
        // and column elemnt in the row
        for (int j = 0; j < matrix_size; j++) {
            // calculate the product result for the PRODUCT[row, column] value
            ParallelMultiplier multiplyThread = new ParallelMultiplier(i, j, matrix_size, matrix_a[i], matrix_b, matrix_product);
            // submit the thread to pool for row-wise multiplication
            executor.execute(multiplyThread);
        }
    }
}
```

Execute the job

Create a new thread job

Parallelism with Multithreading

Example: Matrix Multiplication

- Calculating the inner-product

Initialise `total sum`;
for *Each element k in matrix A's ith row and matrix B's jth column* **do**
  Sum the product of elements k and `total sum`;
end

Algorithm 2: Calculating the inner-product of row i in matrices A and column j in matrix B

Parallelism with Multithreading

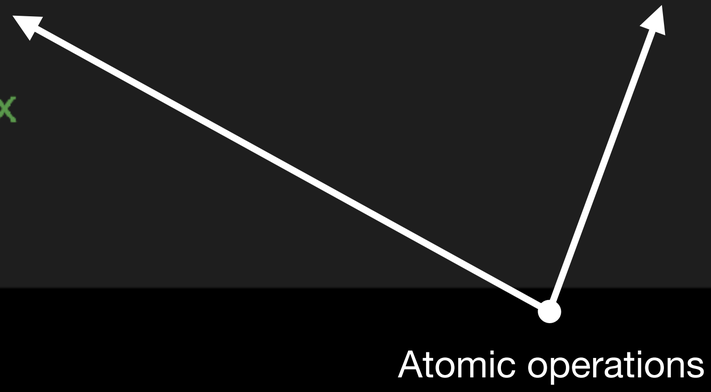
Example: Matrix Multiplication

```
// calculate result stored at matrix_product[row, col]
public void run() {
    // initialise the element total
    int tmp_product = 0;
    // evaluate the dot product of the vectors A[row, :] and B[:, col]
    for (int elem = 0; elem < matrix_size; elem++) {
        // find the total tmp_product
        tmp_product = tmp_product + (matrix_a.get(elem) * matrix_b[elem].get(col));
    }
    // assign the tmp_product to the product matrix
    matrix_product[row].set(col, tmp_product);
}
```

Parallelism with Multithreading

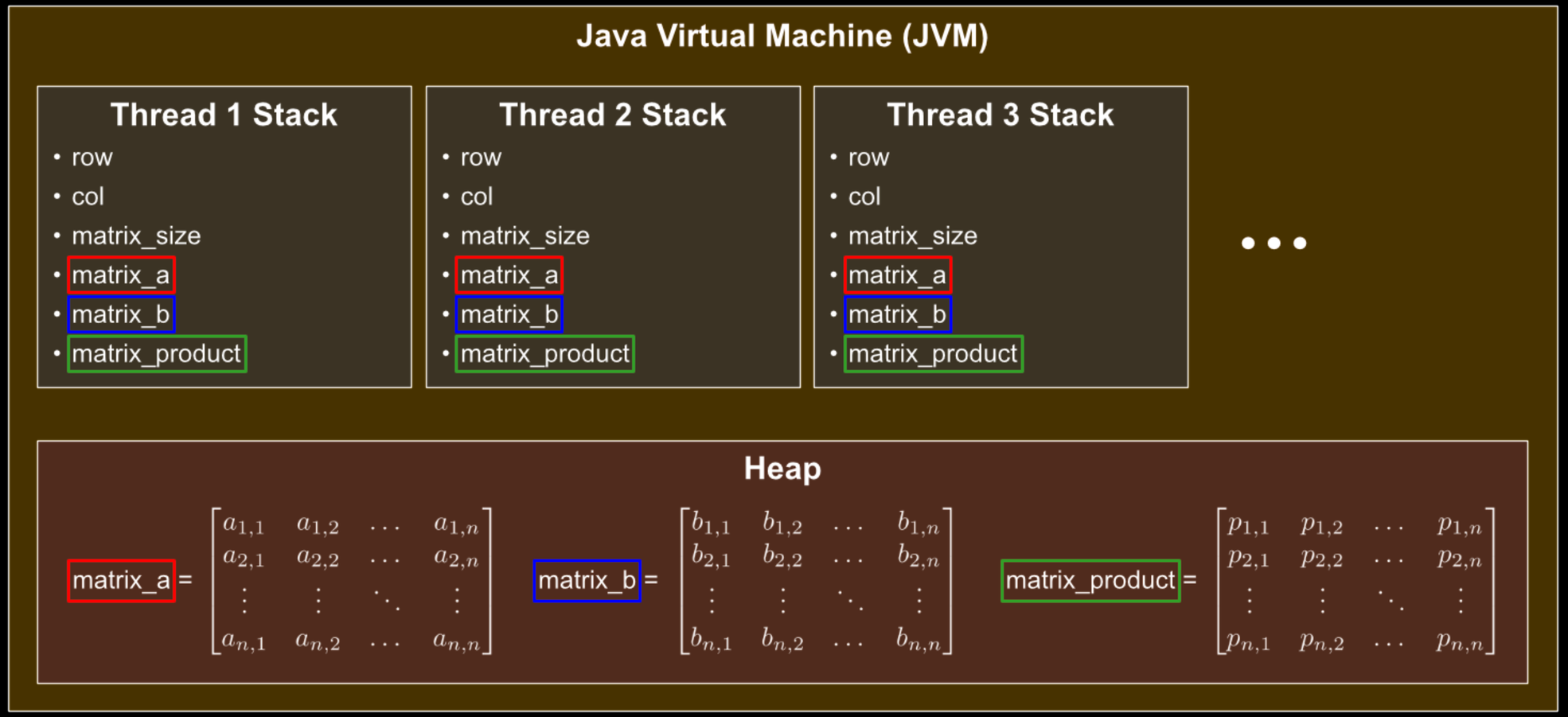
Example: Matrix Multiplication

```
// calculate result stored at matrix_product[row, col]
public void run() {
    // initialise the element total
    int tmp_product = 0;
    // evaluate the dot product of the vectors A[row, :] and B[:, col]
    for (int elem = 0; elem < matrix_size; elem++) {
        // find the total tmp_product
        tmp_product = tmp_product + (matrix_a.get(elem) * matrix_b[elem].get(col));
    }
    // assign the tmp_product to the product matrix
    matrix_product[row].set(col, tmp_product);
}
```



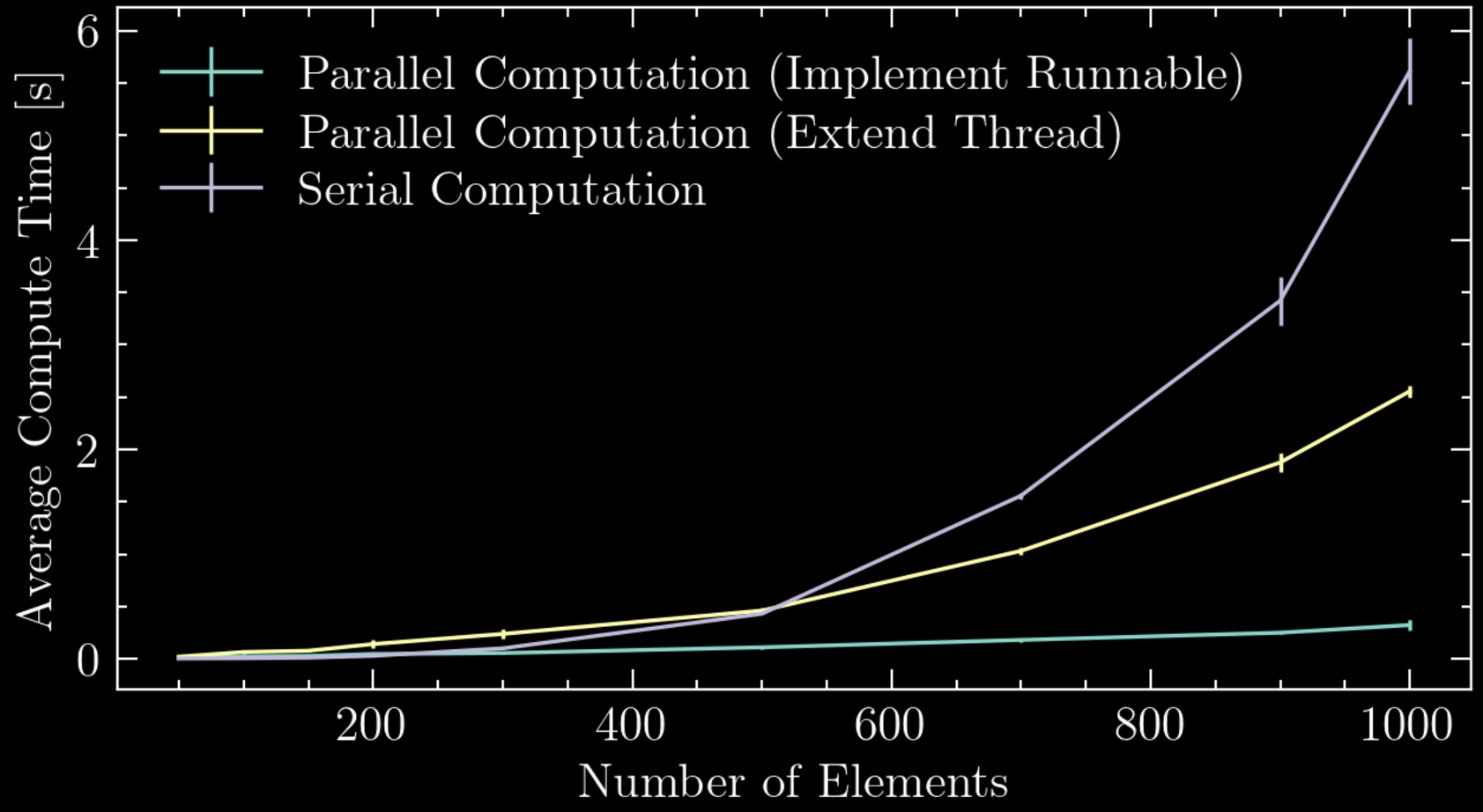
Java Programming

Example: Matrix Multiplication



Performance & Scalability

Example: Matrix Multiplication



Summary

Key Results & Takeaways

- Resources shared by threads should be protected and the operations on them should be *`atomic`*
- Implement the *`Runnable`* interface instead of extending the *`Thread`* class
    - There is performance overhead associated with coupling your class with the Thread class (i.e. multiple object creations)
- Use the *`ExecutorService`* to manage the execution of tasks