MONMOUTH
UNIVERSITY

Department of Computer Science
and Software Engineering

Spring 2019

CS-438-01: Operating Systems Analysis

CS-505-01: Operating Systems Concepts

Java Threads
Multi-threading Exercises

Eugenio Cesario

Divide et impera (1/2)

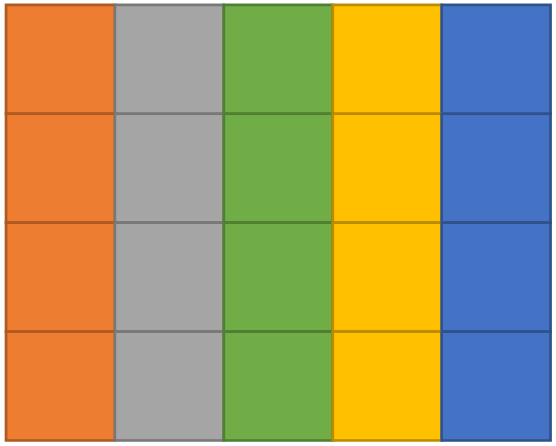
Divide et impera is an algorithm design paradigm based on multi-branched execution. A divide et impera algorithm works by iteratively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

A program, developed on the basis of this technique, is split in three parts:

1. *Divide*: the initial problem is split in several sub-problems (whose size is smaller than the parent problem)
2. *Impera*: each sub-problem is solved
3. *Combine*: all outputs obtained as solutions of the sub-problems are combined to obtain the final result

Divide et impera (1/2)

Problem



Exercise 1.7 – Multi-threaded Scalar Product

Two integer arrays: **a** and **b** (length *n*)

Multi-threaded program computing the scalar product of **a** and **b**

$$\vec{a} = (a_x, a_y, a_z)$$

$$\vec{b} = (b_x, b_y, b_z)$$

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z.$$

$$\begin{bmatrix} A_x & A_y & A_z \end{bmatrix} \begin{bmatrix} B_x \\ B_y \\ B_z \end{bmatrix} = A_x B_x + A_y B_y + A_z B_z = \vec{A} \cdot \vec{B}$$

The program creates *m* thread objects, instances of the *ScalarProductThread* class. Each thread computes the scalar product of a sub-array of length n/m.

Main steps: array and threads initializations, scalar product computation, output of the final result.

m and *n* variables can be defined as numbers in the main.

Exercise 1.7: The ScalarProduct Thread class

```
3  class ScalarProductThread extends Thread {
4      private int begin, end, ris;
5      int [] v1, v2;
6
7      public ScalarProductThread(String name, int [] v1, int [] v2, int begin, int end) {
8          setName(name);
9          this.v1 = v1;
10         this.v2 = v2;
11         this.begin = begin;
12         this.end = end;
13         this.ris = 0;
14     }
15
16     public void run() {
17         System.out.println(Thread.currentThread().getName() + " [" + begin + "," + end + "]  started!");
18         ris = 0;
19         for (int i = begin; i <= end; i++)
20             ris += v1[i] * v2[i];
21     }//run
22
23     public int getResult() {
24         return ris;
25     }
26 }
27
```

Exercise 1.7: a first simple main

```
3
4  //FIRST SIMPLIFIED VERSION: ScalarProduct using only 2 threads
5  public class Ex17_1 {
6      public static void main(String[] args) throws InterruptedException {
7          int n = 6;
8          int [] a = {1,2,3,4,5,6};
9          int [] b = {1,2,3,4,5,6};
10         int result;
11         ScalarProductThread t1 = new ScalarProductThread("T1", a, b, 0, n/2 - 1);
12         ScalarProductThread t2 = new ScalarProductThread("T2", a, b, n/2, n - 1);
13         t1.start();
14         t2.start();
15         t1.join();
16         t2.join();
17         result = t1.getResult() + t2.getResult();
18         System.out.println(result);
19     }
20 }
21
```

Exercise 1.7: towards a more complex main

- Other solutions, having variable *n* and *m*?
- Random initialization of the arrays
- Coding time…

- Can we implement a version by implementing the Runnable interface?
- Coding time…

Exercise 1.7: towards a more complex main

Solution:

1. Implement the ScalarProductThread class
2. Implement the MAIN
    - Create an array of ScalarProductThread threads
    - Initialize each thread, by assigning to it a specific subvector
    - Wait for the completion of all threads (join)
    - Compute the final result

Exercise 1.7: schema of the main

```
public static void main(String[] args) throws InterruptedException {
    int [] a = {1,2,3,4,5,6};
    int [] b = {1,2,3,4,5,6};
    int numOfThreads = 4; //number of threads

    int s = (a.length / numOfThreads); // lenght of subvectors assigned to each thread
    int n = a.length;

    ScalarProductThread threads[] = new ScalarProductThread[numOfThreads];

    int init = -1, end = -1;

    for (int i = 0; i < numOfThreads; i++) {
        init = -1; //TO BE DONE: initialize "init", on the basis of "s" and "n"
        end =  -1; //TO BE DONE: initialize "end", on the basis of "s" and "n"
        threads[i] = new ScalarProductThread("T" + i, a, b, init, end);
        threads[i].start();
    }//for

    //wait for the completion of the threads
    for (int i = 0; i < numOfThreads; i++) {
        threads[i].join();
    }//for

    int result = 0;
    //computation of the final result
    for (int i = 0; i < numOfThreads; i++) {
        System.out.println("Result " + i + " = " + threads[i].getResult());
        result += threads[i].getResult();
    }//for

    System.out.println("Result = " + result);
}
```

Exercise 1.7: schema of the main

```
public static void main(String[] args) throws InterruptedException {
    int [] a = {1,2,3,4,5,6};
    int [] b = {1,2,3,4,5,6};
    int numOfThreads = 4; //number of threads

    int s = (a.length / numOfThreads); // lenght of subvectors assigned to each thread
    int n = a.length;

    ScalarProductThread threads[] = new ScalarProductThread[numOfThreads];

    int init = -1, end = -1;

    for (int i = 0; i < numOfThreads; i++) {
        init = -1; //TO BE DONE: initialize "init", on the basis of "s" and "n"
        end =  -1; //TO BE DONE: initialize "end", on the basis of "s" and "n"
        threads[i] = new ScalarProductThread("T" + i, a, b, init, end);
        threads[i].start();
    }//for

    //wait for the completion of the threads
    for (int i = 0; i < numOfThreads; i++) {
        threads[i].join();
    }//for

    int result = 0;
    //computation of the final result
    for (int i = 0; i < numOfThreads; i++) {
        System.out.println("Result " + i + " = " + threads[i].getResult());
        result += threads[i].getResult();
    }//for

    System.out.println("Result = " + result);
}
```

TO BE DONE

Exercise 1.7: schema of the ScalarProduct Thread

```
class ScalarProductThread extends Thread {
    private int begin, end, ris;
    int [] v1, v2;

    public ScalarProductThread(String name, int [] v1, int [] v2, int begin, int end) {
        setName(name);
        this.v1 = v1;
        this.v2 = v2;
        this.begin = begin;
        this.end = end;
        this.ris = 0;
    }

    public void run() {
        System.out.println(Thread.currentThread().getName() + " [" + begin + "," + end + "]  started!");
        ris = 0;
        for (int i = begin; i <= end; i++)
            ris += v1[i] * v2[i];
    }//run

    public int getResult() {
        return ris;
    }
}
```

Exercise 1.7

Coding time…

Thread Interruption in Java

- Java threads are based on a cooperative organization
- In Java, one thread can not stop another thread. A thread can only **request the other thread to stop**. The request is made in the form of an interruption
- The Thread class provides some methods
  - to ask a thread to be interrupted
  - to know whether a thread has been requested to be interrupted

Thread Interruption in Java

The mechanism to interrupt a thread is implemented using an internal flag known as **interrupt status** (true: to be interrupted):

- the method **void interrupt**() set as *true* the *interrupt status* flag and throws an InterruptedException
- the method **boolean isInterrupted**() tests whether the current thread has been interrupted. It returns the current *value* of the *interrupt status, but it does not modify its value*
- the method **static boolean interrupted()** returns the current *value* of *interrupt status* and reset to false its value (the *interrupted status* of the thread is cleared by this method)

Thread Interruption in Java

- If the Java thread has been implemented through the Runnable interface, it is not possible to invoke the isInterrupted() method, but we must invoke the method Thread.*currentThread*().isInterrupted()

- If the Java thread is in a blocking state (i.e., during a sleep), invoking the interrupt method changes the interrupt status and the thread will receive an **InterruptedException**

Exercise: the Chronometer (using an Interrupt)

```
1  package chronometer;
2
3  public class Chronometer extends Thread {
4
5      public void run() {
6          int numSeconds = 1;
7          while (!isInterrupted()) {
8              try {
9                  Thread.sleep(1000);
10             } catch (InterruptedException e) {
11                 break;
12             }
13             System.out.println("\n" + numSeconds);
14             numSeconds++;
15         }
16     }
17 }
```

Exercise: the Chronometer (using an Interrupt)

```
1  package chronometer;
2
3  import java.util.Scanner;
4
5  public class ChronometerManager {
6
7      public static void main(String[] args) {
8          Scanner in = new Scanner(System.in);
9          Chronometer chronometer = new Chronometer();
10
11         System.out.println("Press ENTER to start...");
12         in.nextLine();
13         chronometer.start();
14
15         System.out.println("Press ENTER to stop...");
16         in.nextLine();
17         chronometer.interrupt();
18
19         System.out.println("END");
20
21     }
22
23 }
24
```

Exercise: the Chronometer (using an Interrupt)

- If the interrupt method is invoked while the thread is sleeping, then its interrupt status will be cleared and it will receive an InterruptedException

- Modify the code to have a 5-second steps chronometer;
- Coding time…
- What do we see in this case?