

# Assignment cover sheet

Complete all sections of this coversheet

|  |  |
| --- | --- |
| Student name | Student number |
| Mohamed Abuklal | 5902514 |

|  |  |
| --- | --- |
| Subject number and name: | Real-Time Embedded Systems ECTE331 |
| Subject coordinator: | Dr Abdsamad binkrid |
| Title of Assignment: | Project Problems (Completed) |
| Date and time due: | 26/06/2023 |
| Total number of pages: | 24 |

Student declaration and acknowledgement (must be read by all students)

By submitting this assignment online, the submitting student declares on behalf of the team that:

1. All team members have read the subject outline for this subject, and this assessment item meets the requirements of the subject detailed therein.
2. This assessment is entirely our own work, except where we have included fully documented references to the work of others. The material contained in this assessment item has not previously been submitted for assessment.
3. Acknowledgement of source information is in accordance with the guidelines or referencing style specified in the subject outline.
4. All team members are aware of the late submission policy and penalty.
5. The submitting student undertakes to communicate all feedback with the other team members.

Place compressed photo of your structure or team here (optional)

**Table of Content:**

1. **Problem 1………………………..3**

**1.a) Single-Thread code………………….3**

**1.a) Single-Thread fixed Image……….5**

**1.b) Multi-Threads code…………………6**

**1.b) Multi-Threads fixed Image……..10**

**1.c) Time execution code..…………….10**

**1.c) Time execution Output…………..15**

**2) Problem 2…………..…………..16**

**2.a) Final correct values………………….16**

**2.b) The synchronization code………..16**

**2.c) sum of shared variables code….20**

**2.c) sum of shared variables output..24**

**Problem 1:**

**Single-Thread code:**

**package** lolo;

**import** java.awt.image.BufferedImage;

**import** java.io.File;

**import** java.io.IOException;

**import** javax.imageio.ImageIO;

**public** **class** HistogramEqualizationSingleThread {

**public** **static** **void** main(String[] args) {

**try** {

**// Loading and reading the input picture (Rain\_Tree.jpg):**

BufferedImage inputImage = ImageIO.*read*(**new** File("Rain\_Tree.jpg"));

BufferedImage outputImage = *performHistogramEqualization*(inputImage);

**// Saving the fixed picture as (R\_T\_fixed(single-Thread)):**

ImageIO.*write*(outputImage, "jpg", **new** File("R\_T\_fixed(single-Thread).jpg"));

**// Printing the message if the Histogram Equalization compiled successfully:**

System.***out***.println("Histogram Equalization completed successfully.");

} **catch** (IOException e) {

e.printStackTrace();

}

}

**// Finding the width and height of the picture to find out the number of the total pixels:**

**private** **static** BufferedImage performHistogramEqualization(BufferedImage inputImage) {

**int** width = inputImage.getWidth();

**int** height = inputImage.getHeight();

BufferedImage outputImage = **new** BufferedImage(width, height, BufferedImage.***TYPE\_INT\_RGB***);

**int**[] histogram = **new** **int**[256];

**int**[] cumulativeHistogram = **new** **int**[256];

**// Calculating the total pixels of the image:**

**int** totalPixels = width \* height;

**// Calculating the histogram:**

**for** (**int** y = 0; y < height; y++) {

**for** (**int** x = 0; x < width; x++) {

**int** rgb = inputImage.getRGB(x, y);

**int** red = (rgb >> 16) & 0xFF;

histogram[red]++;

}

}

**// Calculating the cumulative of the histogram:**

cumulativeHistogram[0] = histogram[0];

**for** (**int** i = 1; i < 256; i++) {

cumulativeHistogram[i] = cumulativeHistogram[i - 1] + histogram[i];

}

**// Applying histogram equalization:**

**for** (**int** y = 0; y < height; y++) {

**for** (**int** x = 0; x < width; x++) {

**int** rgb = inputImage.getRGB(x, y);

**int** red = (rgb >> 16) & 0xFF;

**int** green = (rgb >> 8) & 0xFF;

**int** blue = rgb & 0xFF;

**// Multiplying the cumulative Histogram of the red with 255 and dividing the result with the number of the total pixels:**

**int** newRed = (**int**) (cumulativeHistogram[red] \* 255.0 / totalPixels);

**// Multiplying the cumulative Histogram of the Green with 255 and dividing the result with the number of the total pixels:**

**int** newGreen = (**int**) (cumulativeHistogram[green] \* 255.0 / totalPixels);

**// Multiplying the cumulative Histogram of the Green with 255 and dividing the result with the number of the total pixels:**

**int** newBlue = (**int**) (cumulativeHistogram[blue] \* 255.0 / totalPixels);

**int** newRgb = (newRed << 16) | (newGreen << 8) | newBlue;

outputImage.setRGB(x, y, newRgb);

}

}

**// Returning the fixed image:**

**return** outputImage;

}

}

**Single-Thread fixed Image:**

****

**Multi-Threads code:**

**package** lolo;

**import** java.awt.image.BufferedImage;

**import** java.io.File;

**import** java.io.IOException;

**import** javax.imageio.ImageIO;

**public** **class** HistogramEqualizationMultiThread {

**// The number of threads that have been used:**

**private** **static** **final** **int** ***numOfThreads*** = 4;

**public** **static** **void** main(String[] args) {

**try** {

**// Loading and reading the input picture (Rain\_Tree.jpg):**

BufferedImage inputImage = ImageIO.*read*(**new** File("Rain\_Tree.jpg"));

BufferedImage outputImage = *performHistogramEqualization*(inputImage);

**// Saving the fixed picture as (R\_T\_fixed(multi-Threads)):**

ImageIO.*write*(outputImage, "jpg", **new** File("R\_T\_fixed(multi-Threads).jpg"));

**// Printing the message if the Histogram Equalization compiled successfully:**

System.***out***.println("Histogram Equalization completed successfully.");

} **catch** (IOException e) {

e.printStackTrace();

}

}

**// Converting the Rain\_Tree image to Grayscale:**

**private** **static** BufferedImage convertToGrayscale(BufferedImage inputimage) {

BufferedImage grayscaleImage = **new** BufferedImage(inputimage.getWidth(), inputimage.getHeight(), BufferedImage.***TYPE\_BYTE\_GRAY***);

grayscaleImage.getGraphics().drawImage(inputimage, 0, 0, **null**);

**return** grayscaleImage;

}

**// Finding the width and height of the picture to find out the number of the total pixels:**

**private** **static** BufferedImage performHistogramEqualization(BufferedImage inputImage) {

**int** width = inputImage.getWidth();

**int** height = inputImage.getHeight();

BufferedImage outputImage = **new** BufferedImage(width, height, BufferedImage.***TYPE\_INT\_RGB***);

**int**[] histogram = **new** **int**[256];

**int**[] cumulativeHistogram = **new** **int**[256];

**// Calculating the total pixels of the image:**

**int** totalPixels = width \* height;

**// Calculating the histogram:**

**for** (**int** y = 0; y < height; y++) {

**for** (**int** x = 0; x < width; x++) {

**int** rgb = inputImage.getRGB(x, y);

**int** red = (rgb >> 16) & 0xFF;

histogram[red]++;

}

}

**// Calculating the cumulative of the histogram:**

cumulativeHistogram[0] = histogram[0];

**for** (**int** i = 1; i < 256; i++) {

cumulativeHistogram[i] = cumulativeHistogram[i - 1] + histogram[i];

}

Thread[] threads = **new** Thread[***numOfThreads***];

**int** pixelsPerThread = totalPixels / ***numOfThreads***;

**int** remainingPixels = totalPixels % ***numOfThreads***;

**int** processedPixels = 0;

**// Creating and starting threads:**

**for** (**int** i = 0; i < ***numOfThreads***; i++) {

**int** startPixel = processedPixels;

**int** endPixel = startPixel + pixelsPerThread - 1;

**if** (i == ***numOfThreads*** - 1) {

endPixel += remainingPixels;

}

threads[i] = **new** Thread(**new** HistogramEqualizationTask(inputImage, outputImage, cumulativeHistogram, startPixel, endPixel));

threads[i].start();

processedPixels = endPixel + 1;

}

**// Waiting for all threads to finish the job:**

**for** (**int** i = 0; i < ***numOfThreads***; i++) {

**try** {

threads[i].join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**// Returning the fixed image:**

**return** outputImage;

}

**// Defining private variables that is related to the input image inside a private class:**

**private** **static** **class** HistogramEqualizationTask **implements** Runnable {

**private** **final** BufferedImage inputImage;

**private** **final** BufferedImage outputImage;

**private** **final** **int**[] cumulativeHistogram;

**private** **final** **int** startPixel;

**private** **final** **int** endPixel;

**// Defining the constructor of the HistogramEqualizationTask private class:**

**public** HistogramEqualizationTask(BufferedImage inputImage, BufferedImage outputImage, **int**[] cumulativeHistogram, **int** startPixel, **int** endPixel) {

**this**.inputImage = inputImage;

**this**.outputImage = outputImage;

**this**.cumulativeHistogram = cumulativeHistogram;

**this**.startPixel = startPixel;

**this**.endPixel = endPixel;

}

@Override

**public** **void** run() {

**int** width = inputImage.getWidth();

**int** height = inputImage.getHeight();

**for** (**int** pixelIndex = startPixel; pixelIndex <= endPixel; pixelIndex++) {

**int** y = pixelIndex / width;

**int** x = pixelIndex % width;

**int** rgb = inputImage.getRGB(x, y);

**int** red = (rgb >> 16) & 0xFF;

**int** green = (rgb >> 8) & 0xFF;

**int** blue = rgb & 0xFF;

**// applying histogram equalization by:**

**// Multiplying the cumulative Histogram of the red with 255 and dividing the result with the number of the total pixels:**

**int** newRed = (**int**) (cumulativeHistogram[red] \* 255.0 / (width \* height));

**// Multiplying the cumulative Histogram of the Green with 255 and dividing the result with the number of the total pixels:**

**int** newGreen = (**int**) (cumulativeHistogram[green] \* 255.0 / (width \* height));

**// Multiplying the cumulative Histogram of the Green with 255 and dividing the result with the number of the total pixels:**

**int** newBlue = (**int**) (cumulativeHistogram[blue] \* 255.0 / (width \* height));

**int** newRgb = (newRed << 16) | (newGreen << 8) | newBlue;

outputImage.setRGB(x, y, newRgb);

}

}

}

}

**Multi-Threads fixed Image:**



**Time execution code between single-thread and multi-threads:**

**package** lolo;

**import** java.awt.image.BufferedImage;

**import** java.io.File;

**import** java.io.IOException;

**import** javax.imageio.ImageIO;

**public** **class** HistogramEqualizationComparison {

**// The number of threads that have been used:**

**private** **static** **final** **int** ***numOfThreads*** = 4;

**public** **static** **void** main(String[] args) {

**try** {

**// Loading and reading the input picture (Rain\_Tree.jpg):**

BufferedImage inputImage = ImageIO.*read*(**new** File("Rain\_Tree.jpg"));

**// Applying single-threaded histogram equalization:**

**long** startTimeSingleThread = System.*currentTimeMillis*();

BufferedImage outputImageSingleThread = *performHistogramEqualizationSingleThread*(inputImage);

**long** endTimeSingleThread = System.*currentTimeMillis*();

**long** executionTimeSingleThread = endTimeSingleThread - startTimeSingleThread;

**// Applying multi-threaded histogram equalization:**

**long** startTimeMultiThread = System.*currentTimeMillis*();

BufferedImage outputImageMultiThread = *performHistogramEqualizationMultiThread*(inputImage);

**long** endTimeMultiThread = System.*currentTimeMillis*();

**long** executionTimeMultiThread = endTimeMultiThread - startTimeMultiThread;

**// Loading the fixed image of the single-thread code (R\_T\_fixed(Single-Thread)):**

ImageIO.*write*(outputImageSingleThread, "jpg", **new** File("R\_T\_fixed(Single-Thread).jpg"));

**// Loading the fixed image of the multi-thread code (R\_T\_fixed(multi-Thread)):**

ImageIO.*write*(outputImageMultiThread, "jpg", **new** File("R\_T\_fixed(Multi-Thread).jpg"));

**//Printing the execution time of the single-thread code in milliseconeds:**

System.***out***.println("Single-threaded Histogram Equalization execution time: " + executionTimeSingleThread + "ms");

**//Printing the execution time of the multi-thread code in milliseconds:**

System.***out***.println("Multi-threaded Histogram Equalization execution time: " + executionTimeMultiThread + "ms");

} **catch** (IOException e) {

e.printStackTrace();

}

}

**// Finding the width and height of the picture to find out the number of the total pixels of the single-thread:**

**private** **static** BufferedImage performHistogramEqualizationSingleThread(BufferedImage inputImage) {

**int** width = inputImage.getWidth();

**int** height = inputImage.getHeight();

BufferedImage outputImage = **new** BufferedImage(width, height, BufferedImage.***TYPE\_INT\_RGB***);

**int**[] histogram = **new** **int**[256];

**int**[] cumulativeHistogram = **new** **int**[256];

**int** totalPixels = width \* height;

**// Calculating histogram:**

**for** (**int** y = 0; y < height; y++) {

**for** (**int** x = 0; x < width; x++) {

**int** rgb = inputImage.getRGB(x, y);

**int** red = (rgb >> 16) & 0xFF;

histogram[red]++;

}

}

**// Calculating cumulative histogram:**

cumulativeHistogram[0] = histogram[0];

**for** (**int** i = 1; i < 256; i++) {

cumulativeHistogram[i] = cumulativeHistogram[i - 1] + histogram[i];

}

**// Applying histogram equalization:**

**for** (**int** y = 0; y < height; y++) {

**for** (**int** x = 0; x < width; x++) {

**int** rgb = inputImage.getRGB(x, y);

**int** red = (rgb >> 16) & 0xFF;

**int** green = (rgb >> 8) & 0xFF;

**int** blue = rgb & 0xFF;

**int** newRed = (**int**) (cumulativeHistogram[red] \* 255.0 / totalPixels);

**int** newGreen = (**int**) (cumulativeHistogram[green] \* 255.0 / totalPixels);

**int** newBlue = (**int**) (cumulativeHistogram[blue] \* 255.0 / totalPixels);

**int** newRgb = (newRed << 16) | (newGreen << 8) | newBlue;

outputImage.setRGB(x, y, newRgb);

}

}

**return** outputImage;

}

**// Finding the width and height of the picture to find out the number of the total pixels of the multi-thread:**

**private** **static** BufferedImage performHistogramEqualizationMultiThread(BufferedImage inputImage) {

**int** width = inputImage.getWidth();

**int** height = inputImage.getHeight();

BufferedImage outputImage = **new** BufferedImage(width, height, BufferedImage.***TYPE\_INT\_RGB***);

**int**[] histogram = **new** **int**[256];

**int**[] cumulativeHistogram = **new** **int**[256];

**int** totalPixels = width \* height;

**// Calculating histogram:**

**for** (**int** y = 0; y < height; y++) {

**for** (**int** x = 0; x < width; x++) {

**int** rgb = inputImage.getRGB(x, y);

**int** red = (rgb >> 16) & 0xFF;

histogram[red]++;

}

}

**// Calculating cumulative histogram:**

cumulativeHistogram[0] = histogram[0];

**for** (**int** i = 1; i < 256; i++) {

cumulativeHistogram[i] = cumulativeHistogram[i - 1] + histogram[i];

}

Thread[] threads = **new** Thread[***numOfThreads***];

**int** pixelsPerThread = totalPixels / ***numOfThreads***;

**int** remainingPixels = totalPixels % ***numOfThreads***;

**int** processedPixels = 0;

**// Creating and starting threads:**

**for** (**int** i = 0; i < ***numOfThreads***; i++) {

**int** startPixel = processedPixels;

**int** endPixel = startPixel + pixelsPerThread - 1;

**if** (i == ***numOfThreads*** - 1) {

endPixel += remainingPixels;

}

threads[i] = **new** Thread(**new** HistogramEqualizationTask(inputImage, outputImage, cumulativeHistogram, startPixel, endPixel));

threads[i].start();

processedPixels = endPixel + 1;

}

**// Waiting for all threads to finish:**

**for** (**int** i = 0; i < ***numOfThreads***; i++) {

**try** {

threads[i].join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**return** outputImage;

}

**// Defining private variables that is related to the input image inside a private class:**

**private** **static** **class** HistogramEqualizationTask **implements** Runnable {

**private** **final** BufferedImage inputImage;

**private** **final** BufferedImage outputImage;

**private** **final** **int**[] cumulativeHistogram;

**private** **final** **int** startPixel;

**private** **final** **int** endPixel;

**// Defining the constructor of the HistogramEqualizationTask private class:**

**public** HistogramEqualizationTask(BufferedImage inputImage, BufferedImage outputImage, **int**[] cumulativeHistogram, **int** startPixel, **int** endPixel) {

**this**.inputImage = inputImage;

**this**.outputImage = outputImage;

**this**.cumulativeHistogram = cumulativeHistogram;

**this**.startPixel = startPixel;

**this**.endPixel = endPixel;

}

@Override

**public** **void** run() {

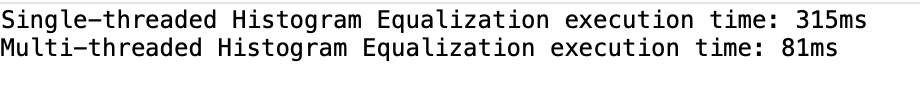
// **TODO** Auto-generated method stub

}

}

}

**Time execution output:**

****

* The single-thread takes a time of 315 millisecond to execute.
* The multi-threads take a time of 81 millisecond to execute.

End of **problem 1**.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Problem 2:**

1. **The final correct values of the shared variables:**

* FuncA1: A1= = 5050
* FuncB1: B1=A1+ = 25150
* FuncB2: B2= = 45150
* FuncA2: A2= 𝐵2 + = 125350
* FuncB3: B3= 𝐴2 + = 250600
* FuncA3: A3= B3 + = 430900

1. **The synchronization code:**

**package** lolo;

**public** **class** SharedVariables {

**private** **static** **final** Object ***lock*** = **new** Object();

**private** **static** **final** String ***A1*** = **null**;

**private** **static** **final** String ***B1*** = **null**;

**private** **static** **final** String ***B2*** = **null**;

**private** **static** **final** String ***A2*** = **null**;

**private** **static** **final** String ***B3*** = **null**;

**private** **static** **final** String ***A3*** = **null**;

**private** **static** **boolean** *funcA1Executed* = **false**;

**private** **static** **boolean** *funcB1Executed* = **false**;

**private** **static** **boolean** *funcB2Executed* = **false**;

**private** **static** **boolean** *funcA2Executed* = **false**;

**private** **static** **boolean** *funcB3Executed* = **false**;

**private** **static** **boolean** *funcA3Executed* = **false**;

**public** **static** **void** main(String[] args) {

Thread funcA1Thread = **new** Thread(() -> {

**synchronized** (***lock***) {

**// Executing Function A1:**

**int** A1 = 0;

**for** (**int** i = 0; i <= 100; i++) {

A1 += i;

}

*funcA1Executed* = **true**;

***lock***.notifyAll();

}

});

Thread funcB1Thread = **new** Thread(() -> {

**synchronized** (***lock***) {

**while** (!*funcA1Executed*) {

**try** {

***lock***.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**// Executing Function B1:**

String B1 = ***A1*** + 0;

**for** (**int** i = 0; i <= 200; i++) {

B1 += i;

}

*funcB1Executed* = **true**;

***lock***.notifyAll();

}

});

Thread funcB2Thread = **new** Thread(() -> {

**synchronized** (***lock***) {

**while** (!*funcB1Executed*) {

**try** {

***lock***.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**// Executing Function B2:**

**int** B2 = 0;

**for** (**int** i = 0; i <= 300; i++) {

B2 += i;

}

*funcB2Executed* = **true**;

***lock***.notifyAll();

}

});

Thread funcA2Thread = **new** Thread(() -> {

**synchronized** (***lock***) {

**while** (!*funcB2Executed*) {

**try** {

***lock***.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**// Executing Function A2:**

String A2 = ***B2*** + 0;

**for** (**int** i = 0; i <= 400; i++) {

A2 += i;

}

*funcA2Executed* = **true**;

***lock***.notifyAll();

}

});

Thread funcB3Thread = **new** Thread(() -> {

**synchronized** (***lock***) {

**while** (!*funcA2Executed*) {

**try** {

***lock***.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**// Executing Function B3:**

String B3 = ***A2*** + 0;

**for** (**int** i = 0; i <= 500; i++) {

B3 += i;

}

*funcB3Executed* = **true**;

***lock***.notifyAll();

}

});

Thread funcA3Thread = **new** Thread(() -> {

**synchronized** (***lock***) {

**while** (!*funcB3Executed*) {

**try** {

***lock***.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**// Executing Function A3:**

String A3 = ***B3*** + 0;

**for** (**int** i = 0; i <= 600; i++) {

A3 += i;

}

*funcA3Executed* = **true**;

***lock***.notifyAll();

}

});

**// Starting all threads functions:**

funcA1Thread.start();

funcB1Thread.start();

funcB2Thread.start();

funcA2Thread.start();

funcB3Thread.start();

funcA3Thread.start();

**// Waiting for all threads to complete successfully:**

**try** {

funcA1Thread.join();

funcB1Thread.join();

funcB2Thread.join();

funcA2Thread.join();

funcB3Thread.join();

funcA3Thread.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**// Print the final correct values of the shared variables:**

System.***out***.println("A1 = " + ***A1***);

System.***out***.println("B1 = " + ***B1***);

System.***out***.println("B2 = " + ***B2***);

System.***out***.println("A2 = " + ***A2***);

System.***out***.println("B3 = " + ***B3***);

System.***out***.println("A3 = " + ***A3***);

}

}

1. **The sum of all shared variables code:**

**package** ll;

**class** SumUtility {

**public** **static** **int** *A1*;

**public** **static** **int** *B1*;

**public** **static** **int** *B2*;

**public** **static** **int** *A2*;

**public** **static** **int** *B3*;

**public** **static** **int** *A3*;

**public** **static** **int** calculateSum(**int** n) {

**long** sum=0;

**for** (**int** i = 0; i<=n ; i++) {

sum += i;

}

**return** (**int**) sum;

}

**public** **static** **int** calculateSum(**int**[] is) {

// **TODO** Auto-generated method stub

**return** 0;

}

}

**public** **class** SharedVariables {

**private** **static** **final** Object ***lock*** = **new** Object();

**private** **static** **boolean** *funcA1Executed* = **false**;

**private** **static** **boolean** *funcB1Executed* = **false**;

**private** **static** **boolean** *funcB2Executed* = **false**;

**private** **static** **boolean** *funcA2Executed* = **false**;

**private** **static** **boolean** *funcB3Executed* = **false**;

**private** **static** **boolean** *funcA3Executed* = **false**;

**// Assigning the shared variables to a public integer:**

**public** **static** **int** *A1*;

**public** **static** **int** *B1*;

**public** **static** **int** *B2*;

**public** **static** **int** *A2*;

**public** **static** **int** *B3*;

**public** **static** **int** *A3*;

**public** **static** **void** main(String[] args) {

**// Creating thread to A1 function:**

Thread funcA1Thread = **new** Thread(() -> {

**synchronized** (***lock***) {

*A1* = SumUtility.*calculateSum*(100);

*funcA1Executed* = **true**;

***lock***.notifyAll();

}

});

**// Creating thread to B1 function:**

Thread funcB1Thread = **new** Thread(() -> {

**synchronized** (***lock***) {

**while** (!*funcA1Executed*) {

**try** {

***lock***.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

*B1* = *A1* + SumUtility.*calculateSum*(200);

*funcB1Executed* = **true**;

***lock***.notifyAll();

}

});

**// Creating thread to B2 function:**

Thread funcB2Thread = **new** Thread(() -> {

**synchronized** (***lock***) {

**while** (!*funcB1Executed*) {

**try** {

***lock***.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

*B2* = SumUtility.*calculateSum*(300);

*funcB2Executed* = **true**;

***lock***.notifyAll();

}

});

**// Creating thread to A2 function:**

Thread funcA2Thread = **new** Thread(() -> {

**synchronized** (***lock***) {

**while** (!*funcB2Executed*) {

**try** {

***lock***.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

*A2* = *B2* + SumUtility.*calculateSum*(400);

*funcA2Executed* = **true**;

***lock***.notifyAll();

}

});

**// Creating thread to B3 function:**

Thread funcB3Thread = **new** Thread(() -> {

**synchronized** (***lock***) {

**while** (!*funcA2Executed*) {

**try** {

***lock***.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

*B3* = *A2* + SumUtility.*calculateSum*(500);

*funcB3Executed* = **true**;

***lock***.notifyAll();

}

});

**// Creating thread to A3 function:**

Thread funcA3Thread = **new** Thread(() -> {

**synchronized** (***lock***) {

**while** (!*funcB3Executed*) {

**try** {

***lock***.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

*A3* = *B3* + SumUtility.*calculateSum*(600);

*funcA3Executed* = **true**;

***lock***.notifyAll();

}

});

**// Starting all threads in the code:**

funcA1Thread.start();

funcB1Thread.start();

funcB2Thread.start();

funcA2Thread.start();

funcB3Thread.start();

funcA3Thread.start();

**// Waiting for all threads to complete:**

**try** {

funcA1Thread.join();

funcB1Thread.join();

funcB2Thread.join();

funcA2Thread.join();

funcB3Thread.join();

funcA3Thread.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**// Assigning sum to be the addition of all shared variables:**

**int** sum = SumUtility.*calculateSum*(**new** **int**[]{*A1*, *B1*, *B2*, *A2*, *B3*, *A3*});

**// Print the final correct values of the shared variables and the sum of these shared variables:**

System.***out***.println("A1 = " + *A1*);

System.***out***.println("B1 = " + *B1*);

System.***out***.println("B2 = " + *B2*);

System.***out***.println("A2 = " + *A2*);

System.***out***.println("B3 = " + *B3*);

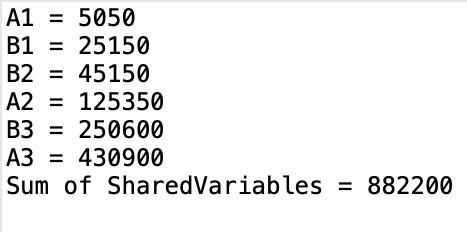
System.***out***.println("A3 = " + *A3*);

System.***out***.println("Sum of SharedVariables = " + sum);

}

}

* **The sum of all shared variables output:**

****