Some different algorithms to calculate pi

There are various algorithms for calculating the pi number, but one of the most famous algorithms is the layer-by-layer algorithm or the **Monte Carlo algorithm**. In this algorithm, the number of random points on a circle is used to obtain the pi value.

Advantages:

1. The algorithm is simple and understandable.

2. The Monte Carlo algorithm easily gives approximate values of pi.

3. This algorithm is scalable and can be easily expanded to more points.

Disadvantages:

1. The Monte Carlo algorithm requires a large number of points to increase the accuracy of calculations.

2. This algorithm takes a lot of calculation time due to the use of a large number of random points.

3. The accuracy of Monte Carlo algorithm calculations depends on the number of random points and the final results may not be accurate.

Leibtens Algorithm is another algorithm for calculating the pi number, which estimates the pi number based on the idea of checking the ratio of the area of the circle to the area of the square in which the circle is located. This algorithm works as follows:

1. We draw a square with side length L.

2. We draw a circle inside this square.

3. We generate the number of random points inside the square.

4. We count the number of points inside the circle.

5. The ratio of the number of points inside the circle to the total number of points produced is an estimate of the pi number.

Advantages:

1. This algorithm is simple and understandable.

2. Leibtens algorithm calculations easily give approximate values of pi.

3. The performance of the algorithm depends on the number of random points and can be easily extended to more points.

Disadvantages:

1. The accuracy of Leibtens algorithm calculations depends on the number of random points and the final results may not be accurate.

2. Leibtens algorithm needs a large number of points to increase the accuracy of calculations.

3. This algorithm takes a lot of calculation time due to the use of a large number of random points.

The best calculation algorithm

The **BBP (Bailey–Borwein–Plouffe)** algorithm is an algorithm for calculating the pi number designed by Brian Bailey, Peter Borwein, and Simon Plouffe. This algorithm uses analytical methods to calculate the decimal digits of pi numbers and requires less time to calculate each digit of pi number.

The BBP algorithm works as follows:

1. It uses an analytical formula to calculate pi numbers.

2. Using this formula, it calculates each decimal digit of pi number independently.

3. By repeating this process for all decimal numbers, pi is calculated with the desired accuracy.

Advantages:

1. The BBP algorithm is suitable for accurate calculation of pi numbers and does not require the number of random points or many calculations.

2. Due to the use of analytical formula, this algorithm has less calculation time than approximate methods.

3. The performance of BBP algorithm for calculating pi numbers is very accurate and reliable.

Disadvantages:

1. Implementation of BBP algorithm may be complicated and required for non-specialists.

2. To calculate pi numbers with high accuracy, the BBP algorithm requires precise and complex calculations, which may be time-consuming.

3. This algorithm works analytically to calculate pi numbers and may require approximate accuracy for some cases.

The BBP (Bailey–Borwein–Plouffe) formula for calculating pi numbers is as follows:

$$\pi = \sum_{k=0}^{\infty} \left[\frac{1}{16^k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right)\right]$$

This formula is defined as an infinite series where each term of this series is a function of k integers. This formula is used by geometric series and analytical mathematics to calculate pi numbers with high accuracy. To calculate pi numbers with high accuracy, you can use this formula and calculate the desired number of sentences in the series.

The BBP algorithm uses this formula to calculate pi numbers, and by repeating the calculation of this formula for k different integers, the pi number is calculated with the desired accuracy.

Items used in the code:

"**Semaphore**" is basically a correct variable or a non-negative Integer that is used to solve the "Critical Section Problem" by acting as a signal. Semaphore is a concept in operating system for "Synchronization" of "Concurrent Process".

"Thread Pool" is a design pattern in programming that is used to reduce the cost of creating and destroying threads at runtime.

In this pattern, a number of threads are kept in a "pool" and used to perform different tasks. Once a task is completed, the thread returns to the pool and prepares to execute the next task. This allows programs to run multiple tasks at the same time without having to create or destroy threads each time the task is executed.

In Java, you can use the <java.util.concurrent.ThreadPoolExecutor> class or one of the helper methods in the <java.util.concurrent.Executors> class to create a threadpool. These classes and methods provide facilities that allow you to control the number of threads, the behavior of new tasks when all threads are busy, and how idle threads are handled.

"BigDecimal" is used to handle very large numbers.

Explain this Code:

In this code, we calculate numbers with up to the desired number of decimals, which we initialize using the BBP formula in the run method, and then use the result method to calculate the final value of pi.

The Controller class creates a Semaphore object with a maximum of 2 permits (new Semaphore(2)). This means that only 2 operators can access the resource at the same time.

The Operator class extends the Thread class and has its own implementation of the run() method. Each operator has its own name and a reference to the Semaphore object.

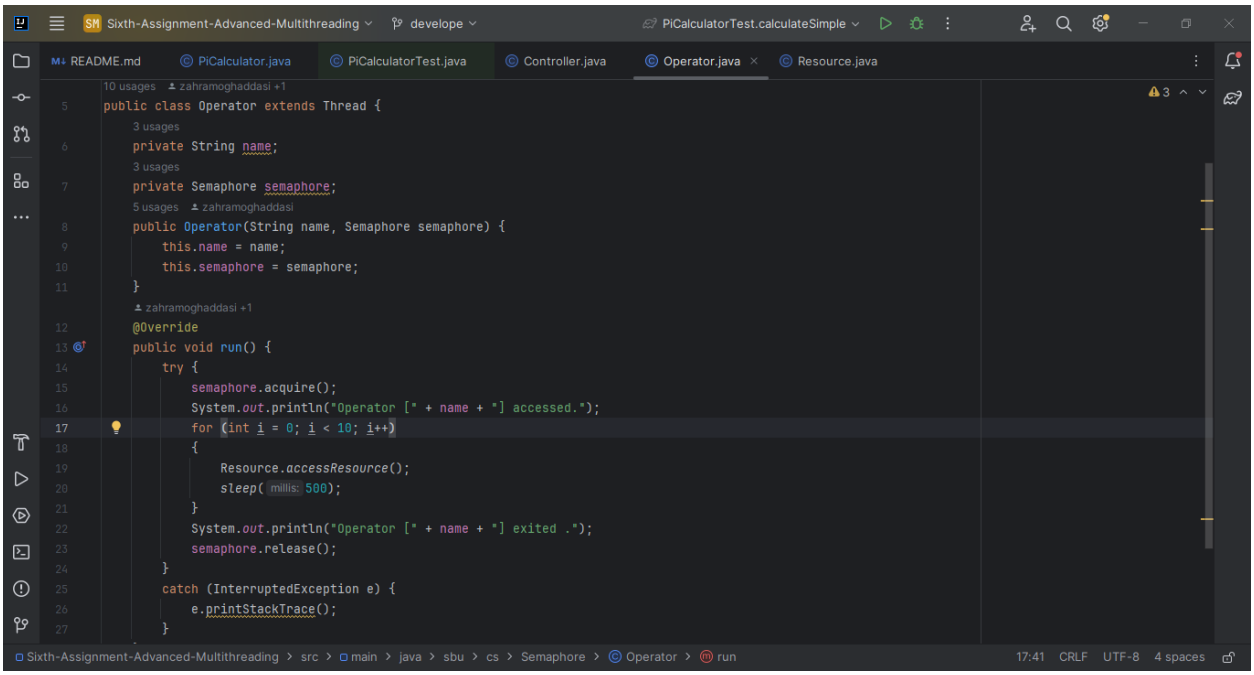
In the run() method, each operator tries to acquire the semaphore using the acquire() method. This blocks the thread if the semaphore is already at its maximum capacity (i.e., 2 operators are already accessing the resource).

Once the semaphore is acquired, the operator accesses the resource using the Resource.accessResource() method and prints a message indicating that it has accessed the resource.

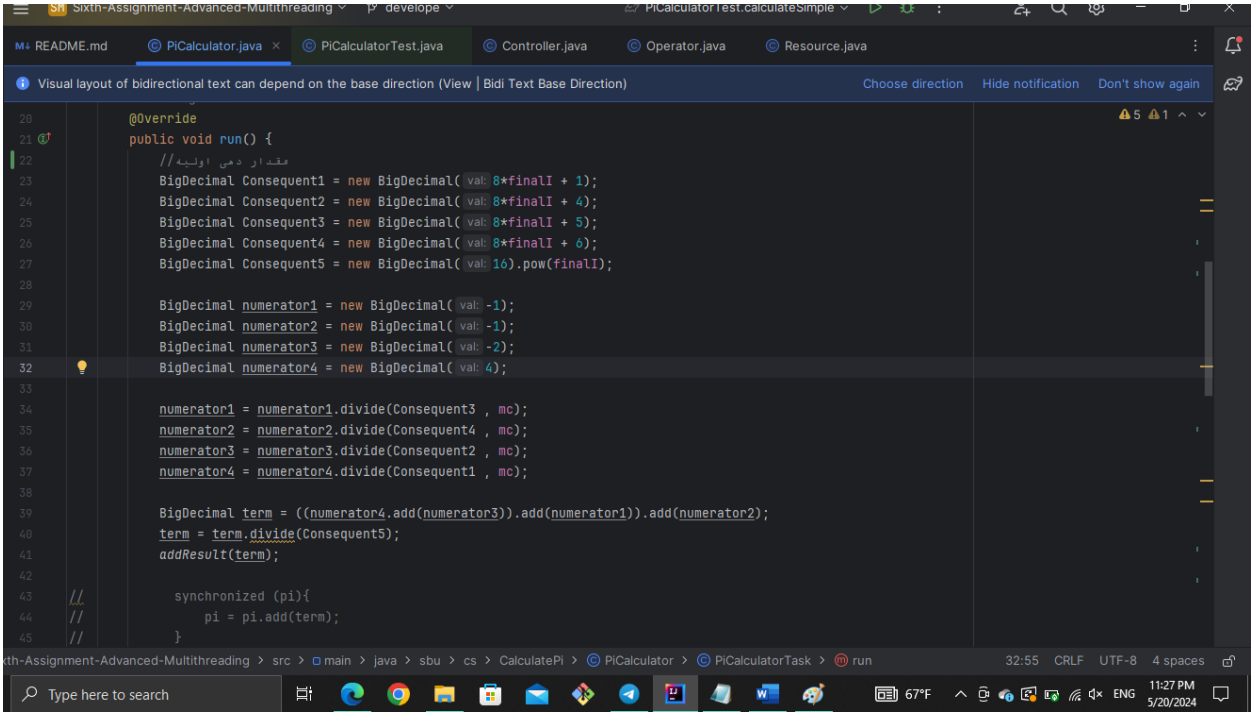
After accessing the resource, the operator releases the semaphore using the release() method, allowing another operator to access the resource.

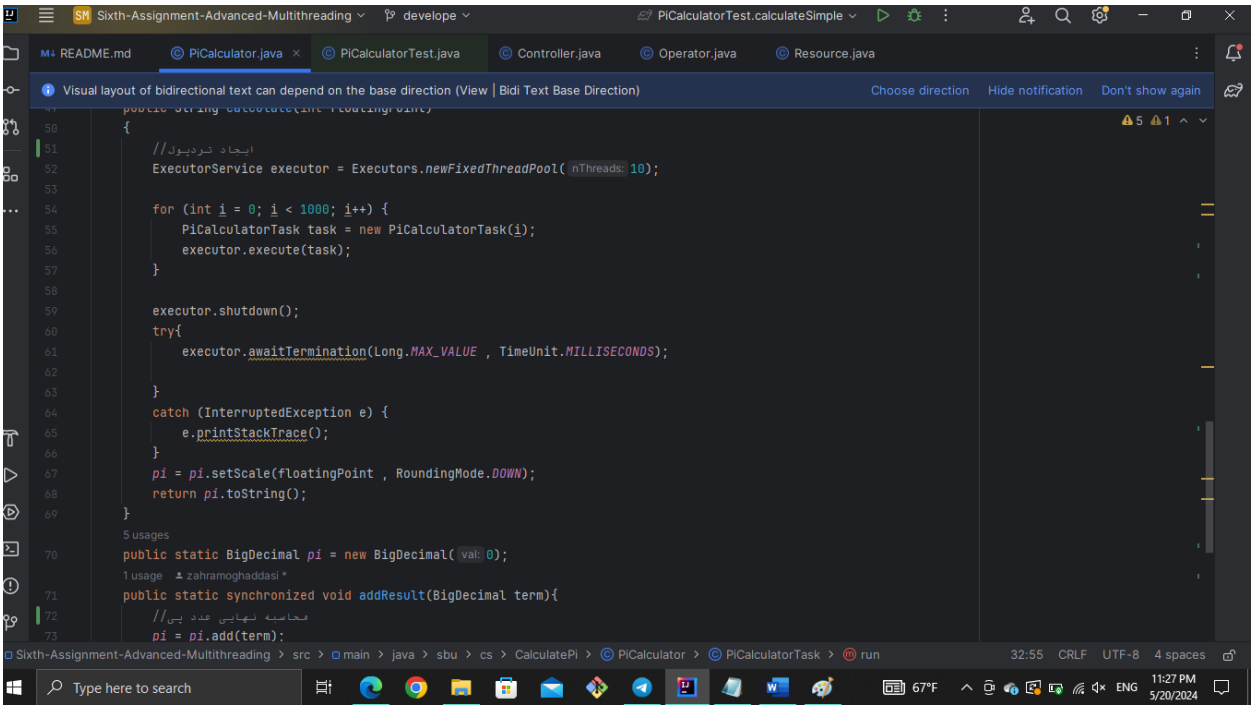
To solve this problem, you need to add code to the Controller class's main() method to print the name and system time every time an operator accesses or exits the resource.

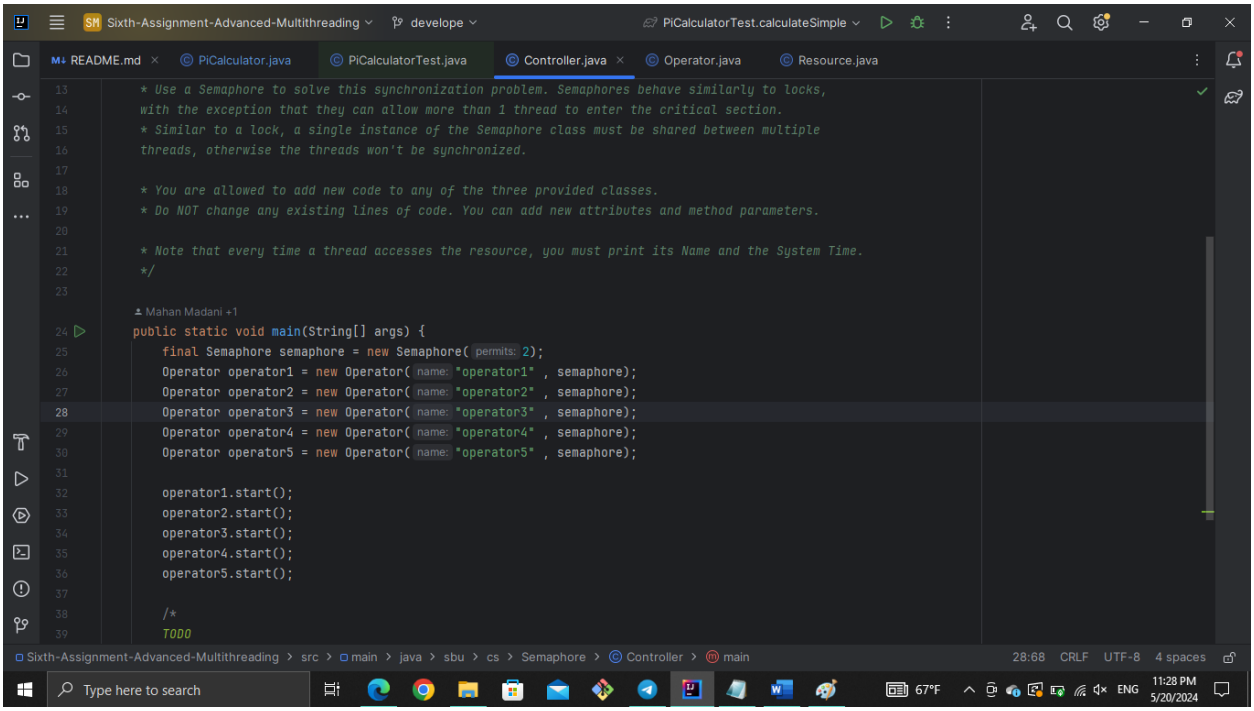
Photo of part of the code:


Sixth-Assignment-Advanced-Multithreading
develope
PiCalculatorTest.calculateSimple
README.md
PiCalculator.java
PiCalculatorTest.java
Controller.java
Operator.java
Resource.java
10 usages zahramoghaddasi +1
public class Operator extends Thread {
3 usages
private String name;
3 usages
private Semaphore semaphore;
5 usages zahramoghaddasi
public Operator(String name, Semaphore semaphore) {
this.name = name;
this.semaphore = semaphore;
}
zahramoghaddasi +1
@Override
public void run() {
try {
semaphore.acquire();
System.out.println("Operator [" + name + "] accessed.");
for (int i = 0; i < 10; i++)
{
Resource.accessResource();
sleep(millis: 500);
}
System.out.println("Operator [" + name + "] exited .");
semaphore.release();
}
catch (InterruptedException e) {
e.printStackTrace();
}
Sixth-Assignment-Advanced-Multithreading > src > main > java > sbu > cs > Semaphore > Operator > run
17:41 CRLF UTF-8 4 spaces


import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
zahramoghaddasi +1 *
public class PiCalculator {
2 usages zahramoghaddasi *
static class PiCalculatorTask implements Runnable {
6 usages
private int finalI;
no usages
public static BigDecimal pi;
4 usages
MathContext mc = new MathContext(setPrecision: 1000);
1 usage zahramoghaddasi *
public PiCalculatorTask(int finalI) {
//کانستراکتور
this.finalI = finalI;
}


Sixth-Assignment-Advanced-Multithreading
develope
PiCalculatorTest.calculateSimple
README.md
PiCalculator.java
PiCalculatorTest.java
Controller.java
Operator.java
Resource.java
Visual layout of bidirectional text can depend on the base direction (View | Bidi Text Base Direction)
Choose direction
Hide notification
Don't show again
@Override
public void run() {
//مقدار دهی اولیه
BigDecimal Consequent1 = new BigDecimal(val: 8*finalI + 1);
BigDecimal Consequent2 = new BigDecimal(val: 8*finalI + 4);
BigDecimal Consequent3 = new BigDecimal(val: 8*finalI + 5);
BigDecimal Consequent4 = new BigDecimal(val: 8*finalI + 6);
BigDecimal Consequent5 = new BigDecimal(val: 16).pow(finalI);
BigDecimal numerator1 = new BigDecimal(val: -1);
BigDecimal numerator2 = new BigDecimal(val: -1);
BigDecimal numerator3 = new BigDecimal(val: -2);
BigDecimal numerator4 = new BigDecimal(val: 4);
numerator1 = numerator1.divide(Consequent3 , mc);
numerator2 = numerator2.divide(Consequent4 , mc);
numerator3 = numerator3.divide(Consequent2 , mc);
numerator4 = numerator4.divide(Consequent1 , mc);
BigDecimal term = ((numerator4.add(numerator3)).add(numerator1)).add(numerator2);
term = term.divide(Consequent5);
addResult(term);
// synchronized (pi){
// pi = pi.add(term);
// }
Sixth-Assignment-Advanced-Multithreading › src › main › java › sbu › cs › CalculatePi › PiCalculator › PiCalculatorTask › run
32:55 CRLF UTF-8 4 spaces
Type here to search
67°F
ENG
11:27 PM
5/20/2024


Sixth-Assignment-Advanced-Multithreading
develope
PiCalculatorTest.calculateSimple
README.md
PiCalculator.java
PiCalculatorTest.java
Controller.java
Operator.java
Resource.java
Visual layout of bidirectional text can depend on the base direction (View | Bidi Text Base Direction)
Choose direction
Hide notification
Don't show again
{
//ایجاد ترد‌یول
ExecutorService executor = Executors.newFixedThreadPool(nThreads: 10);
for (int i = 0; i < 1000; i++) {
PiCalculatorTask task = new PiCalculatorTask(i);
executor.execute(task);
}
executor.shutdown();
try{
executor.awaitTermination(Long.MAX_VALUE , TimeUnit.MILLISECONDS);
}
catch (InterruptedException e) {
e.printStackTrace();
}
pi = pi.setScale(floatingPoint , RoundingMode.DOWN);
return pi.toString();
}
5 usages
public static BigDecimal pi = new BigDecimal(val: 0);
1 usage zahramoghaddasi *
public static synchronized void addResult(BigDecimal term){
//محاسبه نهایی عدد پی
pi = pi.add(term);
Sixth-Assignment-Advanced-Multithreading › src › main › java › sbu › cs › CalculatePi › PiCalculator › PiCalculatorTask › run
32:55 CRLF UTF-8 4 spaces
Type here to search
67°F
ENG
11:27 PM
5/20/2024


Sixth-Assignment-Advanced-Multithreading
develope
PiCalculatorTest.calculateSimple
README.md
PiCalculator.java
PiCalculatorTest.java
Controller.java
Operator.java
Resource.java
* Use a Semaphore to solve this synchronization problem. Semaphores behave similarly to locks,
with the exception that they can allow more than 1 thread to enter the critical section.
* Similar to a lock, a single instance of the Semaphore class must be shared between multiple
threads, otherwise the threads won't be synchronized.
* You are allowed to add new code to any of the three provided classes.
* Do NOT change any existing lines of code. You can add new attributes and method parameters.
* Note that every time a thread accesses the resource, you must print its Name and the System Time.
*/
Mahan Madani +1
public static void main(String[] args) {
final Semaphore semaphore = new Semaphore(permits: 2);
Operator operator1 = new Operator(name: "operator1" , semaphore);
Operator operator2 = new Operator(name: "operator2" , semaphore);
Operator operator3 = new Operator(name: "operator3" , semaphore);
Operator operator4 = new Operator(name: "operator4" , semaphore);
Operator operator5 = new Operator(name: "operator5" , semaphore);
operator1.start();
operator2.start();
operator3.start();
operator4.start();
operator5.start();
/*
TODO
Sixth-Assignment-Advanced-Multithreading › src › main › java › sbu › cs › Semaphore › Controller › main
28:68
CRLF
UTF-8
4 spaces
Type here to search
67°F
ENG
11:28 PM
5/20/2024