**Лабораторная работа 4**

**28.03.2024**

**Уважаемые студенты!**

**Отчет по этому занятию следует представить в двух вариантах:**

**1) в конце занятия, чтобы подтвердить свое присутствие и активность на занятии;**

**2) к следующему занятию в завершенном виде.**

**Синхронизация**

**Модели создания и функционирования потоков: Производитель и потребитель**

**Проведение вычислительных экспериментов**

**Задания**

***Задание 1.*** Необходимость синхронизации. Доказательство с помощью эксперимента.

**Читать**

[**https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html**](https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html)

[**https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html**](https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html)

1. class Counter

2. Напишите класс потока IncrementThread, который будет неоднократно вызывать метод increment() для объекта типа Counter. Параметры конструктора этого класса:

* объект типа Counter
* количество вызовов метода increment()

3. Напишите класс потока DecrementThread, который будет неоднократно вызывать метод Decrement() для объекта типа Counter. Параметры конструктора этого класса:

* объект типа Counter
* количество вызовов метода decrement()

4. class UnsynchronizedCounterTest, method main()

Реализовать эксперимент, описанный в <https://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html>

Thread A invokes increment at about the same time Thread B invokes decrement

Использовать join() для ожидания завершения потока.

Вывести итоговое значение счетчика.

5. Чтобы увидеть ошибку, может потребоваться довольно большое количество приращений и экспериментов.

***Отчет:***

class Counter {  
 private int c = 0;  
  
 public void increment() {  
 c++;  
 }  
  
 public void decrement() {  
 c--;  
 }  
  
 public int value() {  
 return c;  
 }  
}  
  
class IncrementThread extends Thread {  
 private Counter counter;  
 private int numIncrements;  
  
 public IncrementThread(Counter counter, int numIncrements) {  
 this.counter = counter;  
 this.numIncrements = numIncrements;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < numIncrements; i++) {  
 counter.increment();  
 }  
 }  
}  
  
class DecrementThread extends Thread {  
 private Counter counter;  
 private int numDecrements;  
  
 public DecrementThread(Counter counter, int numDecrements) {  
 this.counter = counter;  
 this.numDecrements = numDecrements;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < numDecrements; i++) {  
 counter.decrement();  
 }  
 }  
}  
  
public class UnsynchronizedCounterTest {  
 public static void main(String[] args) throws InterruptedException {  
 Counter counter = new Counter();  
 int numIncrements = 1000000;  
 int numDecrements = 1000000;  
  
 IncrementThread incrementThread = new IncrementThread(counter, numIncrements);  
 DecrementThread decrementThread = new DecrementThread(counter, numDecrements);  
  
 incrementThread.start();  
 decrementThread.start();  
  
 incrementThread.join();  
 decrementThread.join();  
  
 System.*out*.println("Final counter value: " + counter.value());  
 }  
}

Экран вывода с ошибкой

***Задание 2. Synchronized Methods***

**Читать**

[**https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html**](https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html)

1. class SynchronizedCounter

2. class IncrementThread

3. class DecrementThread

4. class SynchronizedCounterTest, method main() – отличается от class UnsynchronizedCounterTest только тем, что использует SynchronizedCounter

5. Запустить программу

***Отчет:***

class SynchronizedCounter {  
 private int c = 0;  
  
 public synchronized void increment() {  
 c++;  
 }  
  
 public synchronized void decrement() {  
 c--;  
 }  
  
 public synchronized int value() {  
 return c;  
 }  
}  
  
class IncrementThread extends Thread {  
 private SynchronizedCounter counter;  
 private int numIncrements;  
  
 public IncrementThread(SynchronizedCounter counter, int numIncrements) {  
 this.counter = counter;  
 this.numIncrements = numIncrements;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < numIncrements; i++) {  
 counter.increment();  
 }  
 }  
}  
  
class DecrementThread extends Thread {  
 private SynchronizedCounter counter;  
 private int numDecrements;  
  
 public DecrementThread(SynchronizedCounter counter, int numDecrements) {  
 this.counter = counter;  
 this.numDecrements = numDecrements;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < numDecrements; i++) {  
 counter.decrement();  
 }  
 }  
}  
  
public class SynchronizedCounterTest {  
 public static void main(String[] args) throws InterruptedException {  
 SynchronizedCounter counter = new SynchronizedCounter();  
 int numIncrements = 1000000;  
 int numDecrements = 1000000;  
  
 IncrementThread incrementThread = new IncrementThread(counter, numIncrements);  
 DecrementThread decrementThread = new DecrementThread(counter, numDecrements);  
  
 incrementThread.start();  
 decrementThread.start();  
  
 incrementThread.join();  
 decrementThread.join();  
  
 System.*out*.println("Final counter value: " + counter.value());  
 }  
}

***Задание 3. Atomic Variables***

**Читать**

[**https://docs.oracle.com/javase/tutorial/essential/concurrency/atomicvars.html**](https://docs.oracle.com/javase/tutorial/essential/concurrency/atomicvars.html)

1. class AtomicCounter

2. class IncrementThread

3. class DecrementThread

4. class AtomicCounterTest, method main() – отличается от предыдущего только тем, что использует AtomicCounter

5. Запустить программу

***Отчет:***

import java.util.concurrent.atomic.AtomicInteger;  
  
class AtomicCounter {  
 private AtomicInteger c = new AtomicInteger(0);  
  
 public void increment() {  
 c.incrementAndGet();  
 }  
  
 public void decrement() {  
 c.decrementAndGet();  
 }  
  
 public int value() {  
 return c.get();  
 }  
}  
  
class IncrementThread extends Thread {  
 private AtomicCounter counter;  
 private int numIncrements;  
  
 public IncrementThread(AtomicCounter counter, int numIncrements) {  
 this.counter = counter;  
 this.numIncrements = numIncrements;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < numIncrements; i++) {  
 counter.increment();  
 }  
 }  
}  
  
class DecrementThread extends Thread {  
 private AtomicCounter counter;  
 private int numDecrements;  
  
 public DecrementThread(AtomicCounter counter, int numDecrements) {  
 this.counter = counter;  
 this.numDecrements = numDecrements;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < numDecrements; i++) {  
 counter.decrement();  
 }  
 }  
}  
  
public class AtomicCounterTest {  
 public static void main(String[] args) throws InterruptedException {  
 AtomicCounter counter = new AtomicCounter();  
 int numIncrements = 1000000;  
 int numDecrements = 1000000;  
  
 IncrementThread incrementThread = new IncrementThread(counter, numIncrements);  
 DecrementThread decrementThread = new DecrementThread(counter, numDecrements);  
  
 incrementThread.start();  
 decrementThread.start();  
  
 incrementThread.join();  
 decrementThread.join();  
  
 System.*out*.println("Final counter value: " + counter.value());  
 }  
}

**Задача производитель-потребитель** (Producer-Consumer problem) является классическим примером синхронизации потоков в параллельном программировании. В этой задаче у нас есть производитель (Producer), который создает данные, и потребитель (Consumer), который потребляет или обрабатывает эти данные. Задача состоит в том, чтобы обеспечить правильную и безопасную передачу данных между производителем и потребителем, учитывая возможность конфликтов при доступе к данным.

Условие:

- Есть ограниченный буфер, который может хранить только ограниченное количество элементов.

- Производитель должен помещать элементы в буфер.

- Потребитель должен извлекать элементы из буфера.

- Если буфер пуст, потребитель должен ждать, пока в буфере не появятся элементы.

- Если буфер заполнен, производитель должен ждать, пока потребитель не извлечет элемент.

Подход к решению:

Для решения этой задачи можно использовать механизм синхронизации, такой как мьютексы, семафоры или условные переменные, чтобы обеспечить безопасное взаимодействие между производителем и потребителем.

**Реализации задачи Производитель-Потребитель на Java**

**Вариант 1**

***BlockingQueue*** имеет подходящий интерфейс <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html>

class Producer implements Runnable {

private final BlockingQueue queue;

Producer(BlockingQueue q) { queue = q; }

public void run() {

try {

while (true) { queue.put(produce()); }

} catch (InterruptedException ex) { ... handle ...}

}

Object produce() { ... }

}

class Consumer implements Runnable {

private final BlockingQueue queue;

Consumer(BlockingQueue q) { queue = q; }

public void run() {

try {

while (true) { consume(queue.take()); }

} catch (InterruptedException ex) { ... handle ...}

}

void consume(Object x) { ... }

}

class Setup {

void main() {

BlockingQueue q = new SomeQueueImplementation();

Producer p = new Producer(q);

Consumer c1 = new Consumer(q);

Consumer c2 = new Consumer(q);

new Thread(p).start();

new Thread(c1).start();

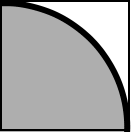
new Thread(c2).start();

}

}

***Задание 4. Вычисление числа Пи методом Монте-Карло***

It is possible to get an estimate of the mathematical constant π by using a random process. The idea is based on the fact that the area of a circle of radius 1 is equal to π, and the area of a *quarter* of that circle is π/4. Here is a picture of a quarter of a circle of radius 1, inside a 1-by-1 square:



The area of the whole square is one, while the area of the part inside the circle is π/4. If we choose a point in the square at random, the probability that it is inside the circle is π/4. If we choose N points in the square at random, and if C of them are inside the circle, we expect the fraction C/N of points that fall inside the circle to be about π/4. That is, we expect 4\*C/N to be close to π. If N is large, we can expect 4\*C/N to be a good estimate for π, and as N gets larger and larger, the estimate is likely to improve.

We can pick a random point in the square by choosing numbers x and y in the range 0 to 1 (using Math.random()). Since the equation of the circle is x\*x+y\*y=1, the point lies inside the circle if x\*x+y\*y is less than 1. One trial consists of picking x and y and testing whether x\*x+y\*y is less than 1. To get an estimate for π, you have to do many trials, count the trials, and count the number of trials in which x\*x+y\*y is less than 1.

for (int i = 0; i < MAX\_SIZE; i++) {

double x = Math.random();

double y = Math.random();

trialCount++;

if (x\*x + y\*y < 1)

inCircleCount++;

}

double estimateForPi = 4 \* ((double)inCircleCount / trialCount);

1. class SequentialProgram, method main()

Вычисляем число Пи, используя приведенные выше подсказки.

2. Решаем задачу, используя два потока один Производитель и один Потребитель и ***BlockingQueue***.

Производитель генерирует случайные точки, а потребитель проверяет, попадает ли точка внутрь области.

***Отчет:***

* Текст последовательной программы
* class SequentialProgram {  
   public static void main(String[] args) {  
   final int MAX\_SIZE = 1000000;  
   int inCircleCount = 0;  
   int trialCount = 0;  
    
   for (int i = 0; i < MAX\_SIZE; i++) {  
   double x = Math.*random*();  
   double y = Math.*random*();  
   trialCount++;  
   if (x \* x + y \* y < 1)  
   inCircleCount++;  
   }  
    
   double estimateForPi = 4 \* ((double) inCircleCount / trialCount);  
   System.*out*.println("Estimate for Pi: " + estimateForPi);  
   }  
  }
* Текст многопоточной программы
* import java.util.concurrent.BlockingQueue;  
  import java.util.concurrent.ArrayBlockingQueue;  
    
  class Producer implements Runnable {  
   private final BlockingQueue<Boolean> queue;  
   private final int maxTrials;  
    
   Producer(BlockingQueue<Boolean> q, int maxTrials) {  
   queue = q;  
   this.maxTrials = maxTrials;  
   }  
    
   public void run() {  
   try {  
   for (int i = 0; i < maxTrials; i++) {  
   double x = Math.*random*();  
   double y = Math.*random*();  
   if (x \* x + y \* y < 1)  
   queue.put(true);  
   else  
   queue.put(false);  
   }  
   } catch (InterruptedException ex) {  
   }  
   }  
  }  
    
  class Consumer implements Runnable {  
   private final BlockingQueue<Boolean> queue;  
   private final int maxTrials;  
   private int inCircleCount = 0;  
    
   Consumer(BlockingQueue<Boolean> q, int maxTrials) {  
   queue = q;  
   this.maxTrials = maxTrials;  
   }  
    
   public void run() {  
   try {  
   for (int i = 0; i < maxTrials; i++) {  
   if (queue.take())  
   inCircleCount++;  
   }  
   double estimateForPi = 4 \* ((double) inCircleCount / maxTrials);  
   System.*out*.println("Estimate for Pi: " + estimateForPi);  
   } catch (InterruptedException ex) {  
   }  
   }  
  }  
    
  class MultithreadedProgram {  
   public static void main(String[] args) {  
   final int MAX\_SIZE = 1000000;  
   BlockingQueue<Boolean> queue = new ArrayBlockingQueue<>(MAX\_SIZE);  
   int numProducers = 1;  
   int numConsumers = 1;  
    
   for (int i = 0; i < numProducers; i++) {  
   Thread producerThread = new Thread(new Producer(queue, MAX\_SIZE));  
   producerThread.start();  
   }  
    
   for (int i = 0; i < numConsumers; i++) {  
   Thread consumerThread = new Thread(new Consumer(queue, MAX\_SIZE));  
   consumerThread.start();  
   }  
   }  
  }
* Определить, равномерно ли загружены потоки

В данном коде каждый производитель и каждый потребитель работают с одной и той же очередью queue. Загрузка потоков считается равномерной, если они имеют примерно одинаковое количество работы. В данном случае, каждый производитель и потребитель выполняют maxTrials итераций, поэтому загрузка считается равномерной.

* Увеличить количество производителей и потребителей

int numProducers = 2;

int numConsumers = 2;

* Таблица с результатами экспериментов

***Последоват:***

Estimate for Pi: 3.14074

Execution Time: 57 ms

***Паралл.:***

Producer Thread: 272 ms

Consumer Thread: 271 ms

Estimate for Pi: 3.138496

***Задание 5. Реализация BlockingQueue***

**Читать**

[**https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html**](https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html)

class Drop – вариант реализации BlockingQueue

Вычисляем Пи, используя class Drop

***Отчет:***

* Текст многопоточной программы
* Таблица с результатами экспериментов
* Сравнить с результатами экспериментов, полученными в предыдущем задании

***Задание 6. Используем AtomicInteger***

***Задача*.** Посчитать количество простых чисел в заданном диапазоне.

***ИДЕЯ.*** Модель делегирования 2. Вместо очереди заданий Рабочие потоки используют общий счетчик типа AtomicInteger, у которого запрашивают очередное число для проверки на простоту.

***Отчет:***

* Главный поток (текст программы)
* Рабочий поток (текст программы)
* Таблица с результатами экспериментов

Размерности задачи: 10 000 000, 50 000 000, 100 000 000.

Количество потоков: 2 потока, 4 потока.

Количество подзадач: 10, 100, 1 000.