**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:***

* Текст программы

public abstract class AbstractIntegerCounter implements Counter<Integer> {  
 private int counter = 0;  
  
 @Override  
 public void increment() {  
 counter++;  
 }  
  
 @Override  
 public void decrement() {  
 counter--;  
 }  
  
 @Override  
 public Integer value() {  
 return counter;  
 }  
}

public class ActionThread extends Thread {  
 private final Runnable action;  
 private final int times;  
  
 public ActionThread(Runnable action, int times) {  
 this.action = action;  
 this.times = times;  
 }  
  
 public void run() {  
 for (int i = 0; i < times; i++) {  
 action.run();  
 }  
 }  
}

public class UnsynchronizedCounterTest {  
 public static void main(String[] args) throws InterruptedException {  
 Counter<Integer> counter = new DefaultCounter();  
 int times = 10000;  
  
 Thread incrementThread = new ActionThread(counter::increment, times);  
 Thread decrementThread = new ActionThread(counter::decrement, times);  
  
 incrementThread.start();  
 decrementThread.start();  
  
 incrementThread.join();  
 decrementThread.join();  
  
 System.*out*.println("Actual counter value: " + counter.value());  
 System.*out*.println("Expected counter value: " + 0);  
 }  
}

* 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

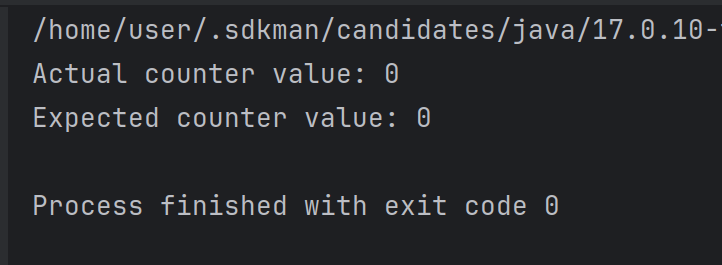
5. Запустить программу

***Отчет:***

* Текст программы

public class SynchronizedCounter extends DefaultCounter {  
 @Override  
 public synchronized void increment() {  
 super.increment();  
 }  
  
 @Override  
 public synchronized void decrement() {  
 super.decrement();  
 }  
}

* Экран вывода



***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

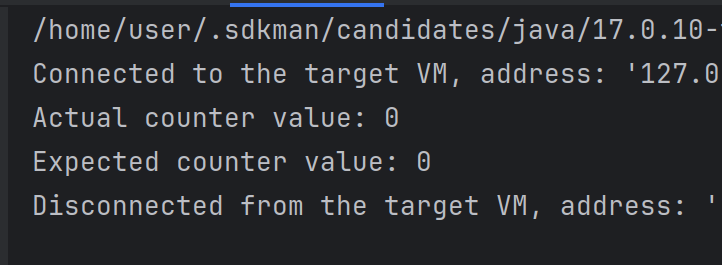
5. Запустить программу

***Отчет:***

* Текст программы

public class AtomicCounter implements Counter<Integer> {  
 private final AtomicInteger counter = new AtomicInteger(0);  
  
 @Override  
 public void increment() {  
 counter.incrementAndGet();  
 }  
  
 @Override  
 public void decrement() {  
 counter.decrementAndGet();  
 }  
  
 @Override  
 public Integer value() {  
 return counter.get();  
 }  
}

* Экран вывода



**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();

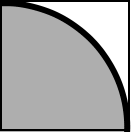
new Thread(c2).start();

}

}

***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

public class SequentialProgram {  
 public static void main(String[] args) {  
 long MAX\_SIZE = 36\_854\_775\_807L;  
 long trialCount = 0;  
 long inCircleCount = 0;  
  
 for (long i = 0; i < MAX\_SIZE; i++) {  
 double x = Math.*random*();  
 double y = Math.*random*();  
 trialCount++;  
 if (x\*x + y\*y < 1)  
 inCircleCount++;  
 }  
 System.*out*.println("Estimate for Pi: " + *estimateForPi*(inCircleCount, trialCount));  
 }  
}

* Текст многопоточной программы

public class MultiThreadedProgram {  
 private final int POINTS\_COUNT = 1\_000\_000\_000;  
 private final BlockingQueue<Point> queue = new ArrayBlockingQueue<>(POINTS\_COUNT);  
  
 public static void main(String[] args) {  
 new MultiThreadedProgram().start();  
 }  
  
 private void start() {  
 final int PRODUCERS\_COUNT = 4;  
 final int CONSUMERS\_COUNT = 2;  
  
 try (ExecutorService executor = Executors.*newFixedThreadPool*(PRODUCERS\_COUNT + CONSUMERS\_COUNT);) {  
 startProducers(executor, PRODUCERS\_COUNT);  
 List<Future<Long>> futures = startConsumers(executor, CONSUMERS\_COUNT);  
  
 executor.shutdown();  
 executor.awaitTermination(20, TimeUnit.*SECONDS*);  
  
 double piEstimate = estimatePi(futures);  
 System.*out*.println("Estimate for pi: " + piEstimate);  
 } catch (InterruptedException | ExecutionException e) {  
 System.*err*.println(e.getMessage());  
 }  
 }  
  
 private void startProducers(ExecutorService executor, int producersCount) {  
 for (int i = 0; i < producersCount; i++) {  
 executor.submit(new PointProducer((POINTS\_COUNT / producersCount), queue));  
 }  
 }  
  
 private List<Future<Long>> startConsumers(ExecutorService executor, int consumersCount) {  
 List<Future<Long>> futures = new ArrayList<>();  
 for (int i = 0; i < consumersCount; i++) {  
 futures.add(executor.submit(new PointConsumer(queue)));  
 }  
  
 return futures;  
 }  
  
 private double estimatePi(List<Future<Long>> futures) throws ExecutionException, InterruptedException {  
 int totalInsideCircle = 0;  
 for (Future<Long> future : futures) {  
 totalInsideCircle += future.get();  
 }  
  
 return 4.0 \* totalInsideCircle / POINTS\_COUNT;  
 }  
}

public class PointProducer implements Runnable {  
 private final long pointsCount;  
 private final BlockingQueue<Point> queue;  
  
 public PointProducer(long pointsCount, BlockingQueue<Point> queue) {  
 this.pointsCount = pointsCount;  
 this.queue = queue;  
 }  
  
 @Override  
 public void run() {  
 for (long i = 0; i < pointsCount; i++) {  
 try {  
 queue.put(new Point());  
// System.out.println("Thread: " + Thread.currentThread().getName() + ". Point " + i + " added to queue");  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 }  
 }  
  
 System.*out*.println("Thread: " + Thread.*currentThread*().getName() + ". Point count " + pointsCount);  
 }  
}

public class PointConsumer implements Callable<Long> {  
 private final BlockingQueue<Point> queue;  
  
 public PointConsumer(BlockingQueue<Point> queue) {  
 this.queue = queue;  
 }  
  
 @Override  
 public Long call() {  
 long inCircleCount = 0L,  
 i = 0L;  
 while (true) {  
 try {  
 Point point = queue.poll(2, TimeUnit.*SECONDS*);  
 if (point != null) {  
 i++;  
// System.out.println("Thread: " + Thread.currentThread().getName() + ". Point " + i + " is taken from queue. Point: " + point);  
 if (point.isInCircle()) {  
 inCircleCount++;  
 }  
 } else {  
 System.*out*.println(  
 "Thread: " + Thread.*currentThread*().getName() + ". " +  
 i + " points are taken from queue. " +  
 inCircleCount + " are in circle. Percentage for current thread: " + (double) inCircleCount / i);  
 return inCircleCount;  
 }  
 } catch (InterruptedException e) {  
 return inCircleCount;  
 }  
 }  
 }  
}

* Определить, равномерно ли загружены потоки

Потоки в моей программе загруженны равномерно, работают парралельно каждый из потоков берет только свою часть задачи. Прилагаю пример логов на 4 producers & 2 consumers 1\_000\_000\_000 точек

Thread: pool-1-thread-2. Point count 250000000

Thread: pool-1-thread-3. Point count 250000000

Thread: pool-1-thread-4. Point count 250000000

Thread: pool-1-thread-1. Point count 250000000

Thread: pool-1-thread-6. 499097037 points are taken from the queue. 391983043 are in circle. Percentage for current thread: 0.7853844321660439

Thread: pool-1-thread-5. 500902963 points are taken from the queue. 393403304 are in circle. Percentage for current thread: 0.7853882549303267

Estimate for pi: 3.141545388

* Увеличить количество производителей и потребителей

Thread: pool-1-thread-7. Point count 83333

Thread: pool-1-thread-9. Point count 83333

Thread: pool-1-thread-12. Point count 83333

Thread: pool-1-thread-11. Point count 83333

Thread: pool-1-thread-3. Point count 83333

Thread: pool-1-thread-1. Point count 83333

Thread: pool-1-thread-10. Point count 83333

Thread: pool-1-thread-6. Point count 83333

Thread: pool-1-thread-2. Point count 83333

Thread: pool-1-thread-8. Point count 83333

Thread: pool-1-thread-5. Point count 83333

Thread: pool-1-thread-4. Point count 83333

Thread: pool-1-thread-15. 209313 points are taken from queue. 164172 are in circle. Percentage for current thread: 0.7843373321293947

Thread: pool-1-thread-13. 294162 points are taken from queue. 230892 are in circle. Percentage for current thread: 0.7849144349032166

Thread: pool-1-thread-16. 260858 points are taken from queue. 204953 are in circle. Percentage for current thread: 0.785687998834615

Thread: pool-1-thread-14. 235663 points are taken from queue. 185042 are in circle. Percentage for current thread: 0.7851975066090138

Estimate for pi: 3.140236

* Таблица с результатами экспериментов

|  |  |  |  |
| --- | --- | --- | --- |
| Producers\_amount | Counsumers\_amount | Numbers | Pi\_estimation |
| 12 | 4 | 1\_000\_000 | 3.140236 |
| 4 | 2 | 1\_000\_000\_000 | 3.141545388 |
|  |  |  |  |

***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

*/\*\**  
 *\* A class that implements the {@link BlockingQueue} interface for Points.*  
 *\* It uses a {@link Queue} to store Points and provides thread-safe operations to put and poll Points.*  
 *\*/*  
public class PointDrop implements BlockingQueue<Point> {  
  
 */\*\**  
 *\* Queue to store Points.*  
 *\*/*  
private final Queue<Point> points = new ConcurrentLinkedQueue<>();  
  
 */\*\**  
 *\* {@link Boolean} flag to indicate if the queue is empty.*  
 *\* If {@code true}, consumer threads should wait for producer threads to put Points.*  
 *\* If {@code false}, producer threads should wait for consumer threads to retrieve Points.*  
 *\*/*  
private boolean empty = true;  
  
 */\*\**  
 *\* Puts a Point into the queue.*  
 *\* If the queue is not empty, it waits until the queue is empty.*  
 *\**  
 *\* @param point the Point to be put into the queue*  
 *\* @throws InterruptedException if any thread has interrupted the current thread*  
 *\*/*  
@Override  
 public void put(Point point) throws InterruptedException {  
 while (!empty) {  
 wait();  
 }  
 empty = false;  
 this.points.add(point);  
 notifyAll();  
 }  
  
 */\*\**  
 *\* Retrieves and removes a Point from the queue, waiting up to the specified wait time if necessary for a Point to become available.*  
 *\* If the queue is empty, it waits until a Point is available or the timeout expires.*  
 *\**  
 *\* @param timeout the maximum time to wait*  
 *\* @param unit the time unit of the timeout argument*  
 *\* @return the head of the queue, or {@code null} if the specified waiting time elapses before a Point is available*  
 *\* @throws InterruptedException if any thread has interrupted the current thread*  
 *\*/*  
@Override  
 public Point poll(long timeout, TimeUnit unit) throws InterruptedException {  
 long timeoutInMillis = unit.toMillis(timeout);  
 long end = System.*currentTimeMillis*() + timeoutInMillis;  
  
 while (empty) {  
 long remaining = end - System.*currentTimeMillis*();  
 if (remaining <= 0) {  
 return null;  
 }  
 wait(remaining);  
 }  
 empty = points.isEmpty();  
 notifyAll();  
 return points.poll();  
 }

* Таблица с результатами экспериментов

|  |  |  |  |
| --- | --- | --- | --- |
| Producers\_amount | Counsumers\_amount | Numbers | Pi\_estimation |
| 1 | 1 | 1\_000 | 3.161325 |
| 1 | 1 | 1\_000\_000 | 3.14123819 |
|  |  |  |  |

* 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)

public class PrimeCounter {  
 private final int RANGE = Integer.*MAX\_VALUE*; // Заданный диапазон  
 private final int COUNTERS\_AMOUNT = 1;  
  
 private final AtomicInteger counter = new AtomicInteger();  
 private final AtomicInteger primeCounter = new AtomicInteger();  
  
 public static void main(String[] args) {  
 new PrimeCounter().start();  
 }  
  
 private void start() {  
 List<Thread> counters = IntStream.*range*(0, COUNTERS\_AMOUNT)  
 .boxed()  
 .map(i -> new Thread(new PrimeChecker(counter, primeCounter, RANGE)))  
 .toList();  
  
 counters.forEach(Thread::start);  
  
 counters.forEach(counter -> {  
 try {  
 counter.join();  
 } catch (InterruptedException e) {  
 System.*err*.println("Thread " + Thread.*currentThread*().getName() + " interrupted");  
 }  
 });  
  
  
 System.*out*.println("Количество простых чисел в диапазоне от 0 до " + RANGE + ": " + primeCounter.get());  
 }  
}

* Рабочий поток (текст программы)

public class PrimeChecker implements Runnable {  
  
 private final AtomicInteger counter;  
 private final AtomicInteger primeCounter;  
 private final int range;  
  
 public PrimeChecker(AtomicInteger counter, AtomicInteger primeCounter, int range) {  
 this.counter = counter;  
 this.primeCounter = primeCounter;  
 this.range = range;  
 }  
  
 @Override  
 public void run() {  
 while (counter.get() < range) {  
 int currentNumber = counter.getAndIncrement();  
 if (*isPrime*(currentNumber)) {  
 primeCounter.incrementAndGet();  
 }  
 }  
 }  
}

* Таблица с результатами экспериментов

|  |  |  |  |
| --- | --- | --- | --- |
| Размерность задачи | Кол-во потоков | Время, ms | Результат |
| 10 000 000 | 2 | 4267 | Простые чисел в диапазоне от 0 до 10000000: 664578 |
| 10 000 000 | 4 | 2309 | Простые чисел в диапазоне от 0 до 10000000: 664578 |
| 50 000 000 | 2 | 43069 | Простые чисел в диапазоне от 0 до 50000000: 3001133 |
| 50 000 000 | 4 | 23008 | Простые чисел в диапазоне от 0 до 50000000: 3001133 |

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

Количество потоков: 2 потока, 4 потока.