**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет прикладной математики и информатики**

**Сергиенко Лев Эдуардович**

Отчет по

Лабораторная работа 3

Параллельные вычисления в Java Модели создания и функционирования потоков Модель делегирования: подход 1 и подход 2 Синхронизация Завершение потоков

|  |  | **Преподаватель** |
| --- | --- | --- |
|  | ***Кондратьева О.М.*** |
|  | | |

**2025**

# **Модель делегирования 1**

## **Описание решения**

**Подход:** Управляющий поток последовательно делит диапазон чисел на небольшие чанки размером CHUNK\_SIZE. Для каждого такого диапазона создается отдельный рабочий поток.

**Синхронизация:** Используется объект-мьютекс для контроля над количеством одновременно работающих потоков. Глобальная переменная availableThreads уменьшает число доступных потоков перед запуском нового потока и увеличивается в методе incrementThreadAvailability() после завершения работы.

**Проверка простоты:** Каждый поток проверяет, какие числа в своем диапазоне являются простыми, и обновляет общий потокобезопасный счетчик (AtomicInteger primeCount).

**Ожидание завершения:** Главный поток ожидает, пока все рабочие потоки не завершат работу (то есть пока availableThreads не вернется к начальному значению).

## **Текст программы**

| package com.example;  import java.util.concurrent.atomic.AtomicInteger;  public class **PrimeCounterDelegationModel1** {  private static final int MAX\_THREADS = Runtime.getRuntime().availableProcessors();  private static int availableThreads = MAX\_THREADS;  private static final Object mutex = new Object();   private static final int CHUNK\_SIZE = 1000;   private static final AtomicInteger primeCount = new AtomicInteger(0);   public static void **main**(String[] args) throws Exception {  int N = 1\_000\_000;  if (args.length > 0) {  try {  N = Integer.parseInt(args[0]);  } catch (NumberFormatException e) {  System.err.println("Некорректное число, используется значение по умолчанию: " + N);  }  }   for (int i = 2; i <= N; i += CHUNK\_SIZE) {  synchronized (mutex) {  // Тут можно было бы использовать if, так как у нас только один поток  // уменьшает availableThreads и уведомление от другого потока будет значить  // увеличение availableThreads, но на всякий случай используем while  while (availableThreads == 0) {  mutex.wait();  }  availableThreads--;  }   Thread t = new Thread(new PrimeTask(i, Math.min(i + CHUNK\_SIZE - 1, N)));  t.start();  }   // Ожидаем завершения всех потоков  synchronized (mutex) {  while (availableThreads != MAX\_THREADS) {  mutex.wait();  }  }   System.out.println("Количество простых чисел от 2 до " + N + " = " + primeCount.get());  }   public static void **incrementThreadAvailability**() {  synchronized (mutex) {  availableThreads++;  mutex.notifyAll();  }  }   static class **PrimeTask** implements **Runnable** {  private final int start;  private final int end;   PrimeTask(int start, int end) {  this.start = start;  this.end = end;  }   @Override  public void **run**() {  // Предположим, что произошло исключение во время выполнения задачи  // и мы не уменьшили счетчик доступных потоков. В этом случае программа никогда  // не завершится, так как главный поток будет ждать, пока счетчик доступных  // потоков не станет равным MAX\_THREADS. Поэтому увеличиваем счетчик доступных  // потоков в finally  try {  int count = 0;  for (int i = start; i <= end; i++) {  if (isPrime(i)) {  count++;  }  }  primeCount.addAndGet(count);  } catch (Exception e) {  System.err.println("Exception in PrimeTask: " + e.getMessage());  } finally {  incrementThreadAvailability();  }  }   private boolean **isPrime**(int n) {  if (n < 2) {  return false;  }  for (int i = 2; i \* i <= n; i++) {  if (n % i == 0) {  return false;  }  }  return true;  }  } } |
| --- |

# **Модель делегирования 2**

## **Описание решения**

**Подход:** Управляющий поток заранее формирует пул рабочих потоков и создает очередь задач, где каждая задача представляет собой диапазон чисел размером CHUNK\_SIZE.

**Синхронизация:** Для организации доступа к очереди используется ReentrantLock и условие Condition notEmpty. Это позволяет рабочим потокам ждать появления задач в очереди.

**Проверка простоты:** Рабочие потоки извлекают задачи из очереди, обрабатывают диапазоны чисел и обновляют общий потокобезопасный счетчик.

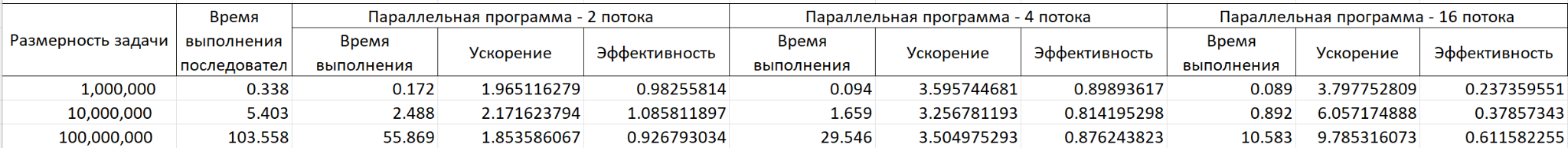
**Завершение работы:** После добавления всех задач в очередь управляющий поток вставляет в очередь специальные сигналы завершения, чтобы уведомить рабочие потоки о завершении работы.

## **Текст программы**

| package com.example;  import java.util.ArrayList; import java.util.LinkedList; import java.util.List; import java.util.Queue; import java.util.concurrent.atomic.AtomicInteger; import java.util.concurrent.locks.Condition; import java.util.concurrent.locks.Lock; import java.util.concurrent.locks.ReentrantLock;  public class **PrimeCounterDelegationModel2** {  private static final Queue<RangeTask> taskQueue = new LinkedList<>();  private static final Lock queueLock = new ReentrantLock();  private static final Condition notEmpty = queueLock.newCondition();  private static final AtomicInteger primeCount = new AtomicInteger(0);   private static final int CHUNK\_SIZE = 1000;   public static void **main**(String[] args) throws Exception {  int N = 1\_000\_000;  if (args.length > 0) {  try {  N = Integer.parseInt(args[0]);  } catch (NumberFormatException e) {  System.err.println("Некорректное число, используется значение по умолчанию: " + N);  }  }   int numWorkers = Runtime.getRuntime().availableProcessors();  List<Thread> workers = new ArrayList<>();   for (int i = 0; i < numWorkers; i++) {  Thread worker = new Thread(new Worker());  worker.start();  workers.add(worker);  }   queueLock.lock();  try {  for (int i = 2; i <= N; i += CHUNK\_SIZE) {  int end = Math.min(i + CHUNK\_SIZE - 1, N);  taskQueue.add(new RangeTask(i, end));  }  notEmpty.signalAll();  } finally {  queueLock.unlock();  }   queueLock.lock();  try {  for (int i = 0; i < numWorkers; i++) {  taskQueue.add(new RangeTask(-1, -1));  }  notEmpty.signalAll();  } finally {  queueLock.unlock();  }   for (Thread worker : workers) {  worker.join();  }   System.out.println("Количество простых чисел от 2 до " + N + " = " + primeCount.get());  }   static class **RangeTask** {  final int start;  final int end;   RangeTask(int start, int end) {  this.start = start;  this.end = end;  }   boolean **isTerminationSignal**() {  return start == -1 && end == -1;  }  }   static class **Worker** implements **Runnable** {  @Override  public void **run**() {  while (true) {  RangeTask task;  queueLock.lock();  try {  while (taskQueue.isEmpty()) {  try {  notEmpty.await();  } catch (InterruptedException e) {  Thread.currentThread().interrupt();  }  }  task = taskQueue.poll();  } finally {  queueLock.unlock();  }   if (task.isTerminationSignal()) {  break;  }   int count = 0;  for (int i = task.start; i <= task.end; i++) {  if (isPrime(i)) {  count++;  }  }  primeCount.addAndGet(count);  }  }   private boolean **isPrime**(int n) {  if (n < 2) {  return false;  }  for (int i = 2; i \* i <= n; i++) {  if (n % i == 0) {  return false;  }  }  return true;  }  } } |
| --- |

# **Таблица с результатами экспериментов**

Модель делегирования 1



Модель делегирования 2

