МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

СЕРГИЕНКО ЛЕВ ЭДУАРДОВИЧ

Отчет по
Лабораторная работа 2
Параллельные вычисления в Java Завершение потока

Преподаватель

Кондратьева О.М.

2025

Задание 1.

Вычислить количество простых чисел (в одном потоке). Есть интерфейсный поток (главный, консольный) и вычислительный. По запросу пользователя интерфейсный поток корректно завершает вычислительный. Реализовать в двух вариантах.

Текст программы 1, скриншот.

```
package com.example;

public class PrimeCounterFlag {
    // Флаг прекращения вычислений. Объявление volatile гарантирует,
    // что изменение переменной в одном потоке будет сразу видно другому.
    private static volatile boolean stopRequested = false;

    public static void main(String[] args) throws Exception {
        Thread countingThread = new Thread(() -> {
            long count = 0;
            long number = 2;

            while (!stopRequested) {
                if (isPrime(number)) {
                    count++;
                }
                number++;
            }

            System.out.println("Найдено простых чисел: " + count);
        });

        countingThread.start();

        System.out.println("Вычисление простых чисел запущено. Нажмите
Enter для остановки...");
        System.in.read();

        stopRequested = true;
        // Ждем завершения вычислительного потока
        countingThread.join();
    }

    private static boolean isPrime(long n) {
        if (n < 2)
            return false;
        for (long i = 2; i * i <= n; i++) {
            if (n % i == 0)
```

```
                return false;
            }
            return true;
        }
    }
```

```
Вычисление простых чисел запущено. Нажмите Enter для остановки...

Найдено простых чисел: 992677
```

Текст программы 2, скриншот.

```
package com.example;

public class PrimeCounterInterrupt {
    public static void main(String[] args) throws Exception {
        Thread countingThread = new Thread(() -> {
            long count = 0;
            long number = 2;
            while (true) {
                if (Thread.interrupted()) {
                    System.out.println("Получено прерывание. Завершаем
вычисления...");
                    break;
                }

                if (isPrime(number)) {
                    count++;
                }

                number++;
            }
            System.out.println("Найдено простых чисел: " + count);
        });

        countingThread.start();

        System.out.println("Вычисление простых чисел запущено. Нажмите Enter
для остановки...");
        System.in.read();

        countingThread.interrupt();
        countingThread.join();
    }

    private static boolean isPrime(long n) {
        if (n < 2)
            return false;
```

```
        for (long i = 2; i * i <= n; i++) {
            if (n % i == 0)
                return false;
        }
        return true;
    }
}
```

```
Вычисление простых чисел запущено. Нажмите Enter для остановки...

Получено прерывание. Завершаем вычисления...
Найдено простых чисел: 652391
```

Задание 2

Описание способа решения.

Для решения проблемы бесконечного выполнения потоков были внесены следующие модификации:

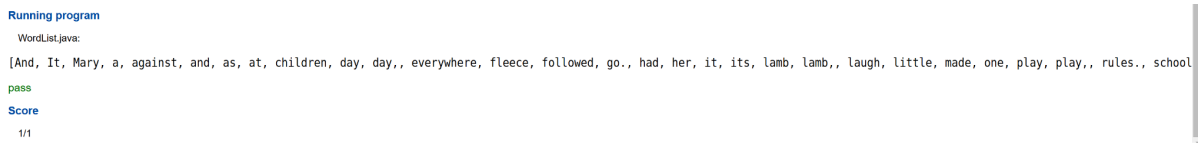
1. **Добавлена переменная volatile boolean sorted**
    - Эта переменная используется для сигнализации о том, что массив полностью отсортирован, и Swapper больше не может выполнять перестановки.
    - Переменная объявлена volatile, чтобы изменения были видны между потоками.
2. **Добавлено условие выхода в классе Swapper**
    - После каждой итерации проверяется, остались ли элементы, требующие перестановки.
    - Если перестановки больше не требуются, устанавливается sorted = true, и поток Swapper завершает работу.
3. **Добавлено условие выхода в классе DupeRemover**
    - Если дубликатов больше не осталось, а sorted == true, поток DupeRemover также завершает выполнение.

Текст программы.

```
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class WordList {
    public WordList(Scanner in) {
        words = new ArrayList<String>();
```

```
        myLock = new ReentrantLock();
        while (in.hasNext()) {
            words.add(in.next());
        }
    }

    private class Swapper implements Runnable {
        public void run() {
            while (true) {
                myLock.lock();
                try {
                    // find a pair to swap
                    int i = 0;
                    while (i < words.size() - 1 &&
words.get(i).compareTo(words.get(i + 1)) <= 0) {
                        i++;
                    }
                    if (i < words.size() - 1) {
                        String word1 = words.get(i);
                        String word2 = words.get(i + 1);
                        words.set(i, word2);
                        words.set(i + 1, word1);
                    } else {
                        sorted = true;
                        break;
                    }
                } finally {
                    myLock.unlock();
                }
            }
        }
    }

    private class DupeRemover implements Runnable {
        public void run() {
            while (true) {
                myLock.lock();
                try {
                    // find a duplicate
                    int i = 0;
                    while (i < words.size() - 1 &&
!words.get(i).equals(words.get(i + 1))) {
                        i++;
                    }
                    if (i < words.size() - 1) {
                        words.remove(i);
                    } else if (sorted) {
                        break;
                    }
                } finally {
                    myLock.unlock();
```

```
                }
            }
        }
    }

    public void slowSort() throws InterruptedException {
        Runnable r1 = new Swapper();
        Runnable r2 = new DupeRemover();
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(words);
    }

    private ArrayList<String> words;
    private Lock myLock;
    private static volatile boolean sorted = false;

    public static void main(String[] args) throws IOException,
InterruptedException {
        Scanner in = new Scanner(new FileReader("data.txt"));
        WordList list = new WordList(in);
        list.slowSort();
    }
}
```



Задание 3.

Задача – поиск простых чисел. Проанализировать полученные результаты. Можно ли улучшить?

Решение

Вместо того чтобы каждому потоку выделять заранее определённый участок чисел, программа разбивает всю работу на маленькие куски. Потоки получают эти куски по очереди. Такой подход позволяет равномерно загружать потоки: если один поток закончит раньше, он возьмёт следующий кусок, а не будет простаивать.

```java
package com.example;

import java.util.Scanner;
import java.util.concurrent.atomic.AtomicInteger;

public class BetterPrimaryFinder {
    private final static int MAX = 100_000_000;
    private final static int CHUNK_SIZE = 1_000;

    private static AtomicInteger nextChunk = new AtomicInteger(2);
    private static AtomicInteger primeCount = new AtomicInteger(0);

    private static class CountPrimesThread extends Thread {
        @Override
        public void run() {
            while (true) {
                int start = nextChunk.getAndAdd(CHUNK_SIZE);
                if (start > MAX)
                    break;
                int end = Math.min(start + CHUNK_SIZE - 1, MAX);
                int count = countPrimes(start, end);
                primeCount.addAndGet(count);
            }
        }
    }

    public static void main(String[] args) throws Exception {
        Scanner scanner = new Scanner(System.in);
        int numberOfThreads = 0;
        while (numberOfThreads < 1 || numberOfThreads > 30) {
            System.out.print("How many threads do you want to use (from 1 to
30)? ");
            numberOfThreads = scanner.nextInt();
            if (numberOfThreads < 1 || numberOfThreads > 30)
                System.out.println("Please enter a number between 1 and 30!");
        }

        CountPrimesThread[] threads = new CountPrimesThread[numberOfThreads];
        long startTime = System.currentTimeMillis();

        for (int i = 0; i < numberOfThreads; i++) {
            threads[i] = new CountPrimesThread();
            threads[i].start();
        }

        for (int i = 0; i < numberOfThreads; i++) {
            threads[i].join();
        }

        long elapsedTime = System.currentTimeMillis() - startTime;
        System.out.println("Total primes: " + primeCount.get());
```

```
        System.out.println("Elapsed time: " + (elapsedTime / 1000.0) + "
seconds.");
    }

    private static int countPrimes(int min, int max) {
        int count = 0;
        for (int i = min; i <= max; i++) {
            if (isPrime(i))
                count++;
        }
        return count;
    }

    private static boolean isPrime(int x) {
        if (x < 2)
            return false;
        int top = (int) Math.sqrt(x);
        for (int i = 2; i <= top; i++) {
            if (x % i == 0)
                return false;
        }
        return true;
    }
}
```

Бенчмарки

Старое решение

Размерность задачи	Время выполнения последовательной программы	Параллельная программа - 2 потока			Параллельная программа - 4 потока		
		Время выполнения	Ускорение	Эффективность	Время выполнения	Ускорение	Эффективность
1,000,000	0.141	0.096	1.46875	0.734375	0.067	2.104477612	0.526119403
10,000,000	3.037	1.949	1.558234992	0.779117496	1.297	2.34155744	0.58538936
100,000,000	74.83	62.08	1.205380155	0.602690077	47.737	1.567547186	0.391886796

Новое Решение

Размерность задачи	Время выполнения последовательной программы	Параллельная программа - 2 потока			Параллельная программа - 4 потока		
		Время выполнения	Ускорение	Эффективность	Время выполнения	Ускорение	Эффективность
1,000,000	0.142	0.063	2.253968254	1.126984127	0.041	3.463414634	0.865853659
10,000,000	2.894	1.616	1.790841584	0.895420792	0.83	3.486746988	0.871686747
100,000,000	87.78	58.08	1.511363636	0.755681818	26.9	3.263197026	0.815799257