**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет прикладной математики и информатики**

**Сергиенко Лев Эдуардович**

Отчет по

Лабораторная работа 5

Параллельные вычисления в Java

Модели создания и функционирования потоков: Производитель и потребитель

Проведение вычислительных экспериментов

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

**2025**

# **Задание 1.**

Изучить корректные решения задачи Производитель/Потребитель.

**Задание 2.**

Изучить различные реализации задачи Производитель/Потребитель на Java.

## **Скелеты реализаций.**

### **1. Реализация с использованием семафоров**

В этом подходе используются три семафора:

* **mutex** – для обеспечения взаимного исключения (двоичный семафор, начальное значение 1);
* **empty** – для подсчёта пустых мест в буфере (начальное значение равно размеру буфера);
* **full** – для подсчёта заполненных мест (начальное значение 0).

Код-скелет:

| #define N 100 /\* Количество мест в буфере \*/ typedef int semaphore; /\* Семафоры представлены как целочисленные переменные \*/  semaphore mutex = 1; /\* Для взаимного исключения \*/ semaphore empty = N; /\* Количество пустых мест \*/ semaphore full = 0; /\* Количество заполненных мест \*/  void **producer**(void) {  int item;  while (TRUE) { /\* Бесконечный цикл производителя \*/  item = produce\_item(); /\* Генерация элемента \*/  down(&empty); /\* Ждём, пока появится пустое место \*/  down(&mutex); /\* Вход в критическую секцию \*/  insert\_item(item); /\* Вставка элемента в буфер \*/  up(&mutex); /\* Выход из критической секции \*/  up(&full); /\* Увеличиваем счетчик заполненных мест \*/  } }  void **consumer**(void) {  int item;  while (TRUE) { /\* Бесконечный цикл потребителя \*/  down(&full); /\* Ждём появления заполненного места \*/  down(&mutex); /\* Вход в критическую секцию \*/  item = remove\_item(); /\* Извлечение элемента из буфера \*/  up(&mutex); /\* Выход из критической секции \*/  up(&empty); /\* Увеличиваем счетчик пустых мест \*/  consume\_item(item); /\* Обработка извлеченного элемента \*/  } } |
| --- |

### **2. Реализация с использованием потоков, мьютексов и условных переменных**

В данном варианте используется один буфер, в который одновременно не может попасть более одного потока. Мьютекс обеспечивает взаимное исключение, а условные переменные позволяют потоку (производителю или потребителю) ждать, пока буфер не станет доступным для записи или чтения.

| #include <pthread.h> #include <stdio.h>  #define MAX 1000000000 /\* Общее количество элементов для производства \*/  pthread\_mutex\_t the\_mutex; pthread\_cond\_t condc, condp; /\* Условные переменные для сигнализации \*/ int buffer = 0; /\* Единичный буфер для обмена \*/  void \*producer(void \*ptr) {  int i;  for (i = 1; i <= MAX; i++) {  pthread\_mutex\_lock(&the\_mutex); /\* Захват мьютекса \*/  while (buffer != 0)  pthread\_cond\_wait(&condp, &the\_mutex); /\* Ждем, пока буфер опустеет \*/  buffer = i; /\* Запись элемента в буфер \*/  pthread\_cond\_signal(&condc); /\* Сигнал потребителю \*/  pthread\_mutex\_unlock(&the\_mutex); /\* Освобождение мьютекса \*/  }  pthread\_exit(0); }  void \*consumer(void \*ptr) {  int i;  for (i = 1; i <= MAX; i++) {  pthread\_mutex\_lock(&the\_mutex); /\* Захват мьютекса \*/  while (buffer == 0)  pthread\_cond\_wait(&condc, &the\_mutex); /\* Ждем появления данных \*/  buffer = 0; /\* Очистка буфера (извлечение элемента) \*/  pthread\_cond\_signal(&condp); /\* Сигнал производителю \*/  pthread\_mutex\_unlock(&the\_mutex); /\* Освобождение мьютекса \*/  }  pthread\_exit(0); }  int **main**() {  pthread\_t pro, con;  pthread\_mutex\_init(&the\_mutex, NULL);  pthread\_cond\_init(&condc, NULL);  pthread\_cond\_init(&condp, NULL);   pthread\_create(&con, NULL, consumer, NULL);  pthread\_create(&pro, NULL, producer, NULL);   pthread\_join(pro, NULL);  pthread\_join(con, NULL);   pthread\_cond\_destroy(&condc);  pthread\_cond\_destroy(&condp);  pthread\_mutex\_destroy(&the\_mutex);  return 0; } |
| --- |

### **3. Реализация на языке Java с использованием монитора**

В Java синхронизация достигается с помощью ключевого слова synchronized и методов wait()/notify(). Здесь монитор инкапсулирует буфер и управляет сигнализацией между потоками производителя и потребителя.

| public class **ProducerConsumer** {  static final int N = 100; // Размер буфера  static Producer p = new Producer(); // Поток-производитель  static Consumer c = new Consumer(); // Поток-потребитель  static Monitor mon = new Monitor(); // Экземпляр монитора   public static void **main**(String args[]) {  p.start(); // Запуск производителя  c.start(); // Запуск потребителя  }   static class **Producer** extends **Thread** {  public void **run**() {  int item;  while (true) { // Бесконечный цикл производства  item = produceItem();  mon.insert(item);  }  }  private int **produceItem**() {  // Логика производства элемента  return 0;  }  }   static class **Consumer** extends **Thread** {  public void **run**() {  int item;  while (true) { // Бесконечный цикл потребления  item = mon.remove();  consumeItem(item);  }  }  private void **consumeItem**(int item) {  // Логика обработки элемента  }  }   static class **Monitor** {  private int[] buffer = new int[N];  private int count = 0, lo = 0, hi = 0;   public synchronized void **insert**(int val) {  if (count == N) goToSleep(); // Буфер полон - ждём  buffer[hi] = val; // Вставка элемента в буфер  hi = (hi + 1) % N;  count++;  if (count == 1) notify(); // Если ранее буфер был пуст - сигнал потребителю  }   public synchronized int **remove**() {  int val;  if (count == 0) goToSleep(); // Буфер пуст - ждём  val = buffer[lo]; // Извлечение элемента из буфера  lo = (lo + 1) % N;  count--;  if (count == N - 1) notify(); // Если ранее буфер был полон - сигнал производителю  return val;  }  private void **goToSleep**() {  try {  wait();  } catch (InterruptedException e) {  // Обработка исключения  }  }  } } |
| --- |

### **4. Реализация с использованием передачи сообщений**

В этой схеме отсутствует общая память для буфера. Вместо этого процессы обмениваются сообщениями фиксированного размера. Потребитель сначала отправляет N пустых сообщений, а затем цикл обмена происходит следующим образом:

1. Производитель ожидает получения пустого сообщения,
2. Заполняет его данными и отправляет потребителю,
3. Потребитель принимает сообщение, извлекает данные,
4. И отправляет обратно пустое сообщение.

| #define N 100 /\* Количество мест (сообщений) \*/  void **producer**(void) {  int item;  message m; /\* Структура сообщения \*/  while (TRUE) {  item = produce\_item(); /\* Производство элемента \*/  receive(consumer, &m); /\* Ожидание пустого сообщения от потребителя \*/  build\_message(&m, item); /\* Формирование сообщения с данными \*/  send(consumer, &m); /\* Отправка заполненного сообщения \*/  } }  void **consumer**(void) {  int item, i;  message m;  /\* Отправка N пустых сообщений для инициализации буфера \*/  for (i = 0; i < N; i++)  send(producer, &m);  while (TRUE) {  receive(producer, &m); /\* Получение сообщения с данными \*/  item = extract\_item(&m); /\* Извлечение элемента из сообщения \*/  send(producer, &m); /\* Возврат пустого сообщения \*/  consume\_item(item); /\* Обработка элемента \*/  } } |
| --- |

# **Задание 3.**

Задача «reverse word».

Задан текстовый файл. Каждое слово этого файла записать в выходной файл в обратном порядке букв.

## **Последовательная программа.**

| package com.example;  import java.io.IOException;  public class **ReverseWordSequential** {  public static void **main**(String[] args) {  String inputFile = "in.txt";  String outputFile = "out.txt";  try {  long startTime = System.currentTimeMillis();  java.util.List<String> lines = java.nio.file.Files.readAllLines(java.nio.file.Paths.get(inputFile));  StringBuilder outputBuilder = new StringBuilder();   for (String line : lines) {  String[] words = line.split(" ");  for (String word : words) {  String reversed = new StringBuilder(word).reverse().toString();  outputBuilder.append(reversed).append(" ");  }  outputBuilder.append("\n");  }   java.nio.file.Files.write(java.nio.file.Paths.get(outputFile), outputBuilder.toString().getBytes());  long endTime = System.currentTimeMillis();  System.out.println("Время выполнения: " + (endTime - startTime) + " мс");  } catch (IOException e) {  e.printStackTrace();  }  } } |
| --- |

## **Многопоточная программа 1.**

| package com.example;  import java.io.BufferedWriter; import java.io.IOException; import java.nio.file.Files; import java.nio.file.Paths; import java.util.concurrent.ArrayBlockingQueue; import java.util.concurrent.BlockingQueue; import java.util.concurrent.TimeUnit;  public class **ReverseWordBlockingQueue** {  private static final int QUEUE\_CAPACITY = 1000;  private static volatile boolean producerFinished = false;   public static void **main**(String[] args) {  String inputFile = "in.txt";  String outputFile = "out.txt";   BlockingQueue<String> queue = new ArrayBlockingQueue<>(QUEUE\_CAPACITY);   // Поток-производитель  Thread producer = new Thread(() -> {  try {  java.util.List<String> lines = java.nio.file.Files.readAllLines(java.nio.file.Paths.get(inputFile));  for (String line : lines) {  String[] words = line.split(" ");  for (String word : words) {  queue.put(word);  }  queue.put("\n");  }  producerFinished = true;  } catch (IOException | InterruptedException e) {  e.printStackTrace();  }  });   // Поток-потребитель  Thread consumer = new Thread(() -> {  try (BufferedWriter writer = Files.newBufferedWriter(Paths.get(outputFile))) {  while (true) {  String word = queue.poll(100, TimeUnit.MILLISECONDS);  if (word != null) {  if (word.equals("\n")) {  writer.newLine();  } else {  String reversed = new StringBuilder(word).reverse().toString();  writer.write(reversed + " ");  }  } else if (producerFinished) {  break;  }  }  writer.flush();  } catch (IOException | InterruptedException e) {  e.printStackTrace();  }  });   long startTime = System.currentTimeMillis();   producer.start();  consumer.start();   try {  producer.join();  consumer.join();  long endTime = System.currentTimeMillis();  System.out.println("Время выполнения: " + (endTime - startTime) + " мс");  } catch (InterruptedException e) {  e.printStackTrace();  }  } } |
| --- |

## **Многопоточная программа 2.**

| **package com.example;  import java.io.BufferedWriter; import java.io.IOException; import java.nio.file.Files; import java.nio.file.Paths; import java.util.LinkedList;  public class ReverseWordMonitor {  private static final String END\_MARKER = "###END###";   public static void main(String[] args) {  String inputFile = "in.txt";  String outputFile = "out.txt";  WordQueue queue = new WordQueue(1000);   // Поток-производитель  Thread producer = new Thread(() -> {  try {  java.util.List<String> lines = java.nio.file.Files.readAllLines(java.nio.file.Paths.get(inputFile));  for (String line : lines) {  String[] words = line.split(" ");  for (String word : words) {  if (!word.isEmpty()) {  queue.put(word);  }  }  queue.put("\n");  }  queue.put(END\_MARKER);  } catch (IOException e) {  e.printStackTrace();  queue.put(END\_MARKER);  }  });   // Поток-потребитель  Thread consumer = new Thread(() -> {  try (BufferedWriter writer = Files.newBufferedWriter(Paths.get(outputFile))) {  while (true) {  String word = queue.take();   if (END\_MARKER.equals(word)) {  break;  }   if (word.equals("\n")) {  writer.newLine();  } else {  String reversed = new StringBuilder(word).reverse().toString();  writer.write(reversed + " ");  }  }  writer.flush();  } catch (IOException e) {  e.printStackTrace();  }  });   long startTime = System.currentTimeMillis();  producer.start();  consumer.start();  try {  producer.join();  consumer.join();  long endTime = System.currentTimeMillis();  System.out.println("Время выполнения: " + (endTime - startTime) + " мс");  } catch (InterruptedException e) {  e.printStackTrace();  }  } }  class WordQueue {  private final LinkedList<String> queue = new LinkedList<>();  private final int capacity;   public WordQueue(int capacity) {  this.capacity = capacity;  }   public synchronized void put(String word) {  while (queue.size() == capacity) {  try {  wait();  } catch (InterruptedException e) {  Thread.currentThread().interrupt();  return;  }  }  queue.add(word);  notify();  }   public synchronized String take() {  while (queue.isEmpty()) {  try {  wait();  } catch (InterruptedException e) {  Thread.currentThread().interrupt();  return null;  }  }  String word = queue.remove();  notify();  return word;  } }** |
| --- |

## **Таблица с результатами экспериментов.**

| Размерность задачи, символов | Время выполнения последовательной, мc | Многопоточная программа 1 | | | Многопоточная программа 2 | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Время выполнения | Ускорение | Эффективность | Время выполнения | Ускорение | Эффективность |
| 1 000 000 | 36 | 112 | 0,3214285714 | 0,1607142857 | 42 | 0,8571428571 | 0,4285714286 |
| 10 000 000 | 101 | 213 | 0,4741784038 | 0,2370892019 | 141 | 0,7163120567 | 0,1790780142 |
| 100 000 000 | 309 | 691 | 0,4471780029 | 0,2235890014 | 712 | 0,433988764 | 0,108497191 |

## **Сравнение реализаций**.

### **1. Последовательная реализация**

**Описание:**В последовательном варианте программа читает весь файл, разбивает строки на слова, переворачивает каждое слово и записывает результат в выходной файл. Все операции выполняются в одном потоке.

### **2. Многопоточная реализация с использованием блокирующей очереди**

Эта версия разделяет задачу на два потока:

**Производитель** читает слова из файла и помещает их в BlockingQueue.

**Потребитель** извлекает слова из очереди, переворачивает их и записывает в выходной файл.

**Плюсы:**

* Использование готового класса ArrayBlockingQueue из пакета java.util.concurrent упрощает работу с потоками.
* Такая схема удобна, если потоков несколько и они должны работать параллельно.

**Минусы:**

* Накладные расходы на синхронизацию и переключение потоков довольно велики.
* При небольшой нагрузке затраты на межпотоковое взаимодействие могут значительно замедлить выполнение по сравнению с последовательным подходом.

### **3. Многопоточная реализация с использованием монитора (synchronized, wait/notify)**

Здесь реализован свой класс-монитор, который с помощью synchronized, wait() и notify() синхронизирует работу производителя и потребителя.

Производитель помещает слова (а также специальный маркер конца) в очередь.

Потребитель извлекает слова, переворачивает их и записывает в файл.

**Плюсы:**

* Более тонкий контроль над синхронизацией, можно настроить работу очереди под конкретные нужды.

**Минусы:**

* Реализация монитора требует большего количества кода.

### **Сравнение результатов**

* **Последовательная реализация** показывает наилучшее время выполнения для данной задачи, так как работа с каждым словом не требует больших вычислительных затрат. Здесь отсутствуют дополнительные издержки, связанные с управлением потоками.
* **Многопоточные реализации** (и с блокирующей очередью, и с монитором) добавляют накладные расходы на синхронизацию и переключение контекста между потоками. В эксперименте видно, что при объеме в 1 000 000 символов версия с монитором почти сопоставима с последовательной (42 мс против 36 мс), а вариант с блокирующей очередью значительно медленнее (112 мс). При увеличении объема данных (10 000 000 и 100 000 000 символов) ситуация сохраняется – ускорение остается менее 1, что означает, что параллельное выполнение не дает выигрыша, а наоборот, замедляет выполнение из-за дополнительных издержек.

Для задачи reverse word, где основная операция разворот символов в слове, последовательная реализация оказывается наиболее эффективной. Многопоточные решения могут быть полезны для более трудоемких операций, но в данном случае их преимущества нивелируются накладными расходами, что приводит к худшим результатам по времени выполнения и низкой эффективности параллелизма.