Теоретическое введение в потоки выполнения и синхронизацию в Java

Поток выполнения (*thread*) представляет собой независимую последовательность выполнения в рамках процесса, работающую параллельно (или квазипараллельно, на одном процессоре) с другими потоками. Java поддерживает многопоточность на уровне языка, позволяя создавать потоки либо путем наследования класса Thread, либо путем реализации интерфейса Runnable. Запуск нового потока выполнения осуществляется с помощью метода start(), который внутренне вызовет метод run() созданного потока, выполняя его одновременно с текущим потоком.

Когда в одной программе выполняется несколько потоков, могут возникнуть ситуации, в которых они *взаимодействуют* или *обмениваются данными* между собой. В частности, если они разделяют **общие ресурсы** (переменные, объекты, файлы и т. д.), существует риск, что их перемежающиеся операции приведут к несогласованности данных. Переключение контекста между потоками может произойти в любой момент, так что два потока могут пытаться одновременно получить доступ к одному и тому же ресурсу, повреждая его состояние. Чтобы предотвратить такие гонки (*race conditions*), необходим механизм, с помощью которого можно контролировать конкурентный доступ к общим ресурсам, то есть обеспечить **взаимное исключение** потоков из критических участков кодabiblioteca.utcluj.ro. **Критический участок** — это сегмент кода, который манипулирует общим ресурсом и который не должен выполняться более чем одним потоком одновременно.

Java предоставляет базовый механизм синхронизации в виде **монитора**. В Java любой объект имеет связанный с ним внутренний монитор, который действует как *блокировка* и *очередь ожидания* для потоков. Когда поток входит в критический участок, помеченный как synchronized, он должен получить монитор соответствующего объекта («взять замок»). Пока монитор удерживается одним потоком, никакой другой поток не может войти в его критические секции, синхронизированные на том же объекте — любые потоки, запрашивающие вход, будут автоматически заблокированы, ожидая освобождения мониторaelse.fcim.utm.mdelse.fcim.utm.md. При выходе из критического участка монитор освобождается («замок» открывается), позволяя другому потоку получить доступ к критическому ресурсуelse.fcim.utm.md.

Помимо взаимного исключения, часто требуется и **условная синхронизация** действий потоков. Это означает, что поток-производитель должен иметь возможность *сигнализировать* потоку-потребителю, что данные были произведены, а потребитель должен *ждать*, пока у него появятся данные для обработки, и наоборот — потребитель может сигнализировать, что он потреблял ресурс, позволяя производителю продолжить работу. Java поддерживает этот тип координации с помощью методов wait() и notify()/notifyAll(), которые являются частью API любого объекта (из базового класса Object). Поток может вызвать wait() на мониторе синхронизированного объекта, чтобы освободить блокировку и перейти в режим ожидания, пока другой поток (обладающий тем же монитором) не вызовет notify() или notifyAll(), чтобы разбудить потоки, находящиеся в режиме ожидания. Этот механизм лежит в основе коммуникации между потоками в модели мониторовelse.fcim.utm.md.

В целом, платформа Java предоставляет несколько инструментов для синхронизации: от ключевого слова synchronized и методов wait/notify, присущих любому объекту, до расширенного API из пакета java.util.concurrent (блокировки, условия, блокирующие очереди и т. д.). В следующих разделах мы опишем проблему производитель-потребитель

и рассмотрим различные решения для синхронизации потоков, чтобы правильно ее реализовать.

4. Описание проблемы «производитель-потребитель»

Проблема производитель-потребитель является классическим примером синхронизации и коммуникации между потоками выполнения. Сценарий предполагает наличие двух типов конкурирующих потоков, которые совместно используют общий ресурс (обычно буфер): поток (или группа потоков) *«Производитель»*, который повторно генерирует данные и помещает их в буфер, и поток (или группа потоков) *«Потребитель»,* который извлекает эти данные из буфера и обрабатывает их.else.fcim.utm.md. Например, производитель может генерировать строку чисел или сообщений, а потребитель поочередно считывает их и отображает или обрабатывает. И производитель, и потребитель работают постоянно (теоретически, в бесконечном цикле): производитель *производит* новые элементы, а потребитель *потребляет* произведенные элементы.

Основная сложность в этой задаче заключается в координации доступа к **общему буферу**. Буфер имеет определенную емкость (может быть минимум 1 элемент или буфер с фиксированным размером). Для поддержания *согласованности данных* и предотвращения ошибок необходимо соблюдать следующие правила синхронизации:

- **Взаимное исключение при доступе к буферу:** в любой момент времени только один поток (производитель или потребитель) должен манипулировать буфером. Другими словами, операции помещения и извлечения в/из буфера являются критическими участками, которые не должны выполняться одновременно. Это требование обеспечивает целостность структуры данных буфера (например, предотвращая одновременное чтение и запись в одни и те же позиции)mobylab.docs.crescdi.pub.ro.
- **Потребитель ожидает, пока буфер** не **опустеет:** если нет доступных элементов (буфер пуст), потребительские потоки должны **блокироваться** до тех пор, пока производитель не создаст новый элемент. В отсутствие этого ожидания несинхронизированный потребитель попытался бы прочитать несуществующие данные, что привело бы к недействительным результатам (или к многократному получению одного и того же элемента)else.fcim.utm.mdmobylab.docs.crescdi.pub.ro.
- **Производитель ожидает, пока буфер не будет заполнен:** если буфер имеет ограниченную емкость и уже заполнен (нет места для новых элементов), поток-производитель должен **приостановить работу** до тех пор, пока потребитель не извлечет хотя бы один элемент, освободив место. Без этой синхронизации несинхронизированный производитель перезаписывал бы еще неиспользованные данные или терял бы элементы (если буфер перезаписывает старые значения)else.fcim.utm.mdmobylab.docs.crescdi.pub.ro.
- **Без потери данных и без взаимных блокировок:** решение должно гарантировать, что **все произведенные элементы в конечном итоге потребляются один раз** (ценности не теряются и не дублируются) и что нити не попадают в состояние постоянного взаимного ожидания (*deadlock*). Потребители никогда не должны пытаться потреблять из пустого буфера, а производители не должны продолжать производство в полном буфере — эти условия обеспечиваются вышеупомянутыми механизмами синхронизацииhabr.com. В то же время, если синхронизация выполнена правильно (например, с использованием соответствующих

уведомлений), **тупиковая ситуация** предотвращается, и система может работать бесконечно, производя и потребляя данные без полной блокировки.

Проблема производитель-потребитель иллюстрирует необходимость синхронизации как для взаимного исключения, так и для связи между потоками. Далее мы представим различные механизмы синхронизации, предлагаемые Java, и продемонстрируем реализацию проблемы P-C с использованием этих механизмов.

5. Представление используемых классов синхронизации Java

Java предоставляет несколько **механизмов синхронизации** параллельного выполнения, от встроенных в язык до утилит из библиотек высокого уровня. В этой работе мы сосредоточимся на четырех основных подходах: использовании ключевого слова synchronized (внутренние мониторы), использовании класса ReentrantLock (явно реентерантный замок) вместе с интерфейсом Condition (переменные условия) и использовании коллекций типа BlockingQueue из пакета java.util.concurrent. Далее мы кратко опишем каждый из этих механизмов и то, как они способствуют синхронизации потоков в контексте проблемы производитель-потребитель.

synchronized (внутренние мониторы Java): Ключевое слово synchronized является основным механизмом, с помощью которого Java обеспечивает взаимное исключение на критических участках. Блок или метод, объявленный synchronized, защищается *монитором* определенного объекта. Только один поток может владеть монитором объекта в данный момент времени, поэтому другие потоки, пытающиеся войти в синхронизированные секции на том же объекте, будут заблокированы до освобождения монитора. Монитор действует как эксклюзивный *замок*, связанный с общим ресурсом, предотвращая одновременный доступ потоков к нему. Например, в нашей реализации общего буфера мы пометим методы put и get как synchronized, чтобы предотвратить интерференцию производителя и потребителя при чтении и записи значенийelse.fcim.utm.mdelse.fcim.utm.md.

Помимо блокировки доступа, мониторы Java также предоставляют механизмы **ожидания и уведомления** с помощью методов из класса Object: wait(), notify() и notifyAll(). Любой поток, владеющий монитором синхронизированного объекта, может вызвать wait(), чтобы временно освободить монитор и перейти в режим ожидания до наступления определенного события (желаемого условия). Другой поток, после выполнения ожидаемого действия, может вызвать notify() (чтобы разбудить один поток, находящийся в ожидании) или notifyAll() (чтобы разбудить все потоки, ожидающие на этом мониторе). В контексте проблемы P-C производитель, после размещения элемента в буфере, вызовет notifyAll(), чтобы разбудить возможного потребителя, заблокированного в ожидании новых данных. И наоборот, потребитель, после извлечения элемента, может уведомить производителя о том, что в буфере освободилось место. Использование wait() и notifyAll() обеспечивает координацию двух потоков таким образом, чтобы соблюдались условия ожидания (буфер полный/пустой), упомянутые вышеelse.fcim.utm.mdelse.fcim.utm.md. (Примечание: в данном типе задачи предпочтительнее использовать notifyAll() вместо notify(), чтобы избежать **зацикливания** оставшегося потока в ожидании — например, если есть несколько производителей/потребителей, notifyAll() гарантирует, что все, кто может действовать, будут пробуждены).

ReentrantLock (явно реентерантный замок): класс ReentrantLock входит в пакет java.util.concurrent.locks и предлагает более гибкую альтернативу встроенному механизму

мониторов. ReentrantLock ведет себя аналогично неявной блокировке (монитору), связанной с synchronized, но предоставляет явные методы для **блокировки** (lock()) и **разблокировки** (unlock()) ресурса. Таким образом, программист получает более тонкий контроль над моментом и продолжительностью блокировки, например, может попытаться получить блокировку с временем ожидания (tryLock() с таймаутом) или указать политику справедливости (*fairness*) при предоставлении блокировки. Название *reentrant* указывает на то, что один и тот же поток может несколько раз приобретать одну и ту же блокировку (рекурсивно входя в критические секции, защищенные одной и той же блокировкой), не оставаясь заблокированным, но он должен освобождать блокировку столько раз, сколько он ее приобретал.

Преимуществом ReentrantLock является **лучшая масштабируемость** в сценариях с большим количеством потоков, что обеспечивает улучшенную производительность и дополнительные функции по сравнению с synchronizedbiblioteca.utcluj.ro. Например, можно использовать *справедливые блокировки* (опция создания блокировки в режиме *fair*, которая гарантирует, что потоки обслуживаются в том порядке, в котором они запрашивают блокировку, избегая starvation). Кроме того, ReentrantLock позволяет реализовывать схемы неблокирующей или ограниченной по времени блокировки (метод tryLock() может немедленно возвращаться, если блокировка занята, или может ждать в течение ограниченного интервала времени)biblioteca.utcluj.ro. Однако использование явных блокировок требует повышенного внимания со стороны программиста: любой вызов lock() должен обязательно сопровождаться соответствующим вызовом unlock() (обычно помещенным в блок finally), чтобы избежать постоянной блокировки ресурса. Мы проиллюстрируем использование ReentrantLock в решении проблемы P-C, показав, как мы можем получить такое же поведение, как и в варианте с synchronized.

Condition (переменная условия): Интерфейс Condition из пакета java.util.concurrent.locks работает вместе с Lock и представляет собой объектный эквивалент механизма wait/notify. Практически, экземпляр Condition, связанный с ReentrantLock, предоставляет такие методы, как await() (эквивалентный wait(), который освобождает блокировку и приостанавливает текущий поток до сигнала) и signal()/signalAll() (эквивалентные notify()/notifyAll(), для пробуждения потоков, ожидающих этого условия)biblioteca.utcluj.ro. Отличие от обычных мониторов заключается в том, что мы можем иметь **несколько объектов Condition для одного и того же блокировки**, позволяя разным группам потоков ожидать разных условий.ubbcluj.ro. В нашей задаче это полезно для разделения ситуаций ожидания: например, мы можем создать условие notEmpty, которое будут ожидать потребители, когда буфер пуст, и условие notFull, которое будут ожидать производители, когда буфер полон. Таким образом, поток, который производит, может дать signal() на условие notEmpty, когда помещает элемент (пробуждая возможного потребителя), а потребитель может дать signal() на notFull, когда извлекает элемент (пробуждая возможного производителя). Использование Condition обеспечивает **более точную гранулярность** условной синхронизации по сравнению с одним монитором, где все потоки ожидают одного и того же набора условий. Чтобы использовать Condition, объект создается вызовом lock.newCondition() на существующем ReentrantLock. В практическом примере мы увидим, как Condition помогает четко выразить логику ожидания в производителе-потребителе.

BlockingQueue (блокирующая очередь): Интерфейс BlockingQueue (из пакета java.util.concurrent) определяет структуру данных типа thread-safe, специально разработанную для сценариев «производитель-потребитель». Блокирующая очередь

реализует операции вставки и извлечения в **блокирующем** режиме: если поток пытается извлечь из очереди, когда она пуста, поток будет автоматически заблокирован до тех пор, пока элемент не станет доступным; аналогично, если поток пытается вставить элемент в полную очередь (в случае очереди с ограниченной емкостью), поток-производитель будет заблокирован до тех пор, пока не появится свободное местоbiblioteca.utcluj.ro. Это поведение предоставляется по умолчанию, поэтому программисту не нужно явно управлять wait()/notify — они реализуются внутренне методологиями BlockingQueue. Существует несколько доступных реализаций, таких как ArrayBlockingQueue (очередь с фиксированной емкостью, основанная на массивах), LinkedBlockingQueue (очередь с опциональной емкостью, основанная на связанном списке), PriorityBlockingQueue (блокирующая очередь, обслуживающая элементы на основе приоритета) или SynchronousQueue (специальная очередь без емкости, где каждая вставка блокируется до тех пор, пока не будет принята)biblioteca.utcluj.ro.

Использование блокирующей очереди значительно упрощает решение проблемы производитель-потребитель, поскольку больше не нужно вручную писать код условной синхронизации — блокировка и разблокировка потоков осуществляется автоматически за кулисами. Таким образом, BlockingQueue можно рассматривать как *специализированный монитор* для передачи данных между потоками. Как мы покажем в практическом примере, производитель может просто вызвать метод put(element) на очереди, а потребитель — метод take(), зная, что эти вызовы будут ждать, если операция не может быть выполнена немедленно. Кроме того, эти классы хорошо оптимизированы и протестированы, что приводит к более **надежным** и **чистым** реализациям кода concurentbaeldung.com. Практически, вместо того, чтобы заниматься переменными состояния, такими как available, или вызовами wait/notify, мы используем BlockingQueue, который умеет блокировать производителей, пока есть место, и блокировать потребителей, пока есть элементы для потребления.

Чтобы продемонстрировать синхронизацию потоков в контексте проблемы производитель-потребитель, мы представим три варианта реализации, каждый из которых использует другой механизм синхронизации. Во всех вариантах мы будем рассматривать **общий буфер**, в который поток **Производитель** вводит элементы (целые числа), а поток **Потребитель** извлекает их. Производитель будет генерировать, например, числа от 1 до 10 (с небольшой задержкой между ними, чтобы имитировать реальную работу), а потребитель будет их принимать и отображать. Цель состоит в том, чтобы благодаря синхронизации потребитель не пытался читать, когда данные недоступны, а производитель не записывал новые данные поверх еще не потребленных.

Решение 1: Использование ReentrantLock и Condition

Первое решение реализует ReentrantLock вместо synchronized вместе с двумя условиями: одно для «буфер не пуст» (notEmpty) и другое для «буфер не полон» (notFull). Хотя в нашем примере буфер имеет емкость 1 (аналогично предыдущему решению), мы все же будем использовать два отдельных объекта Condition, чтобы продемонстрировать преимущество этого механизма — в случае расширения до буфера с несколькими слотами два условия позволяют нам выборочно сигнализировать потребителям или производителям, вместо того чтобы каждый раз пробуждать все потоки. Код представлен ниже:

// Решение 1: использование ReentrantLock с Condition

```
import java.util.concurrent.locks.*;
class BufferLock {
    private final Lock lock = new ReentrantLock();
    private final Condition notEmpty = lock.newCondition();
    private final Condition notFull  = lock.newCondition();
    private int значение = -1;
    private boolean доступен = false;

    public void put(int x) throws InterruptedException {
        lock.lock();                    // получает явный замок
        try {
            while (доступно) {
                notFull.await();        // ждет, пока буфер не освободится
            }
            значение = x;
            доступно = true;
            System.out.println("Производитель ввел: " + x);
            notEmpty.signal();          // сигнализирует, что буфер больше не пуст
        } finally {
            lock.unlock();              // освобождает замок в блоке finally
        }
    }

    public int get() throws InterruptedException {
        lock.lock();
        try {
            while (!доступно) {
                notEmpty.await();       // ждет, пока появится значение для потребления
            }
            int результат = значение;
            доступно = false;
            System.out.println("Потребитель взял: " + результат);
```

```
            notFull.signal();          // сигнализирует, что в буфере освободился слот
            return результат;
        } finally {
            lock.unlock();
        }
    }
}

public class TestLockCondition {
    public static void main(String[] args) {
        BufferLock buf = new BufferLock();
        // Создаем поток-производитель с помощью Runnable
        Runnable prodTask = () -> {
            for (int i = 1; i <= 10; i++) {
                try {
                    buf.put(i);
                    Thread.sleep(50);
                } catch (InterruptedException e) { e.printStackTrace(); }
            }
        };
        // Создаем поток-потребитель, используя другой Runnable
        Runnable consTask = () -> {
            for (int i = 1; i <= 10; i++) {
                try {
                    buf.get();
                    Thread.sleep(100);
                } catch (InterruptedException e) { e.printStackTrace(); }
            }
        };
        // Запускаем потоки
        new Thread(prodTask).start();
        new Thread(consTask).start();
```

```
    }
}
```

Пояснения: В этом варианте BufferLock использует объект ReentrantLock с именем lock для защиты критических участков. Методы put и get больше не объявляются как synchronized, вместо этого мы явно вызываем lock.lock() в начале и lock.unlock() в конце (в блоке finally, чтобы гарантировать, что разблокировка произойдет даже в случае исключений). Условия notEmpty и notFull создаются с помощью lock.newCondition(). Они представляют собой отдельные очереди ожидания, связанные с одним и тем же замком:

- notFull используется производителем для ожидания, пока буфер не будет заполнен. В коде, если доступно == true, производитель вызывает notFull.await(), освобождая замок и блокируясь до тех пор, пока другой поток не сигнализирует об этом условии. Когда потребитель извлекает значение, он устанавливает доступно = false и вызывает notFull.signal(), что пробудит один из потоков, ожидающих notFull (в нашем случае, производителя).
- notEmpty используется симметрично потребителем. Если доступно == false (буфер пуст), потребитель выполняет notEmpty.await(), приостанавливаясь до тех пор, пока производитель не поместит значение и не вызовет notEmpty.signal().

Мы замечаем, что логика ожидания и сигнализации очень похожа на логику в предыдущем варианте, только вместо wait()/notifyAll() на мониторе мы используем await()/signal() на объектах условия. Преимущество здесь заключается в том, что мы можем разбудить *именно* тот тип потока, который необходимо разбудить: например, когда буфер становится непустым, мы сигнализируем notEmpty (разбудив потребителя, но не других производителей, которые останутся заблокированными, поскольку их условие notFull все еще ложно). Таким образом, мы избегаем ненужного пробуждения потоков, которые затем обнаруживают, что не могут действовать, и снова входят в режим ожидания. В примере с одним производителем и одним потребителем разница не заметна, но в сценариях с несколькими потоками такое разделение «wait-set» повышает эффективность синхронизации.

Как и в предыдущем случае, мы используем циклы while для проверки условий перед ожиданием и обрабатываем возможные прерывания потоков (выбрасывая InterruptedException из методов put/get вызывающему). Здесь мы использовали синтаксис с лямбда-выражениями для создания потоков производителя и потребителя (Runnable prodTask и consTask), которые имеют поведение, эквивалентное классам Producator и Consumator из первого решения.

Выполнение и результат: Программа TestLockCondition будет производить чередующиеся результаты Производитель поместил X / Потребитель взял X для X от 1 до 10, демонстрируя, что и с ReentrantLock координация является правильной. С точки зрения функциональности результат тот же. Однако внутренне подход с явным блокированием дает возможность более легкого расширения на несколько потоков или буферов с несколькими слотами. Например, если мы изменим буфер, чтобы он содержал циклический вектор размера N > 1, мы сможем настроить условия while для проверки count == N (буфер полный) или count == 0 (буфер пустой), сохранить два Condition (notFull/notEmpty), а сигналы останутся прежними. ReentrantLock и Condition позволяют нам реализовать эту **проблему ограниченного буфера** очень четким образом (практически вручную воспроизводя то, что предлагает следующий класс, BlockingQueue,

но с большей гибкостью и контролем над деталями). В то же время ReentrantLock позволил бы нам также попытаться получить блокировку с таймаутом, если бы мы хотели (например, производитель мог бы отказаться от производства, если не смог бы получить блокировку в течение определенного времени) — такие функции невозможны с synchronized simplubiblioteca.utcluj.ro.

Решение 2: Использование блокирующей очереди (BlockingQueue)

Второй подход использует высокоуровневую абстракцию Java: коллекции типа BlockingQueue. Они уже включают в себя необходимые механизмы синхронизации, что позволяет нам реализовать производителя и потребителя **без написания кода блокировки и уведомления вручную**. Мы будем использовать ArrayBlockingQueue<Integer> — блокирующую очередь с фиксированной емкостью (для примера выберем емкость, например 5). Производитель будет использовать метод put() очереди, а потребитель — метод take(). Оба метода выбрасывают InterruptedException и блокируют текущий поток, если операция не может быть выполнена мгновенно: put() блокирует, если очередь полна, а take() блокирует, если она пуста. Вот реализация:

```
// Решение 2: использование блокирующей очереди

import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.BlockingQueue;


public class TestBlockingQueue {

   public static void main(String[] args) {

      BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(5); // буфер с 5 позициями

      // Поток производителя

      Thread producer = new Thread(() -> {

         for (int i = 1; i <= 10; i++) {

            try {

               queue.put(i);  // блокирует, если очередь полна

               System.out.println("Производитель поместил: " + i);

               Thread.sleep(50);

            } catch (InterruptedException e) {

               Thread.currentThread().interrupt();

            }

         }

      });

      // Поток потребителя
```

```
        Thread consumer = new Thread(() -> {
            for (int i = 1; i <= 10; i++) {
                try {
                    int val = queue.take();  // блокирует, если очередь пуста
                    System.out.println("Потребитель взял: " + val);
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });
        producer.start();
        потребитель.start();
    }
}
```

Пояснения: Мы создали блокирующую очередь queue с емкостью 5. Она будет действовать как буфер, совместно используемый потоком producer и потоком consumer. Обратите внимание, что больше не требуется ни блока synchronized, ни блокировки, ни явного условия: методы queue.put() и queue.take() занимаются синхронизацией. Согласно спецификации BlockingQueue, если producer вызывает put(x), а очередь уже заполнена (имеет 5 неиспользованных элементов), то поток producer будет **ждать в блокировке**, пока consumer не извлечет что-либо (освободит место)biblioteca.utcluj.ro. И наоборот, если потребитель вызывает take() на пустой очереди, поток потребителя будет ждать, пока производитель не вставит элемент, и разбудит его. Эта функциональность обеспечивается внутренне ArrayBlockingQueue с помощью собственных блокировок и условий. Практически ArrayBlockingQueue имеет внутренне реализованный механизм, очень похожий на тот, что в решении 2 (использует два условия, notEmpty и notFull, и внутреннюю блокировку)officialcto.comofficialcto.com. Для программиста, однако, детали скрыты, и он имеет в своем распоряжении простой интерфейс типа put/take.

Вышеприведенный код запускает два потока, которые автоматически синхронизируются через очередь. Мы можем напрямую использовать Thread.sleep(), чтобы имитировать разные времена производства/потребления, зная, что необходимые блокировки управляются очередью. Преимущества такого решения — ясность и безопасность: исчезает возможность забыть сделать notify() или неправильно использовать условия, что значительно снижает сложность. Кроме того, решение легко обобщается — если мы хотим иметь больше производителей и больше потребителей, достаточно запустить больше потоков, использующих одну и ту же очередь. Все производители будут автоматически заблокированы, если очередь заполнена (до тех пор, пока потребитель не освободит место), и все потребители будут автоматически ждать, если нет элементов (до тех пор, пока производитель не добавит что-нибудь)biblioteca.utcluj.ro. Эта модель

программирования является **кооперативной** и потокобезопасной по дизайну, предлагаемой библиотеками Java.

Выполнение и результат: При выполнении программы TestBlockingQueue будет наблюдаться аналогичное чередование сообщений «Производитель поместил X» и «Потребитель взял X». Точный порядок может отличаться, но, как и раньше, ни один потребитель не будет пытаться потреблять из пустоты (если потребитель попадает в цикл раньше производителя, он будет блокироваться в take(), пока не появится элемент), и ни один производитель не будет чрезмерно заполнять буфер (если производитель быстрее и заполняет 5 мест, при 6-й вставке он будет блокироваться в put(), пока потребитель не возьмет что-нибудь). Таким образом, инварианты проблемы P-C поддерживаются автоматически с помощью BlockingQueue. Этот подход подчеркивает силу абстракций в java.util.concurrent: разработчик может правильно решить проблему конкуренции, не управляя напрямую условиями синхронизации, что приводит к более лаконичному коду, менее подверженному ошибкамbaeldung.com.

Выводы

В этой лабораторной работе мы исследовали различные методы синхронизации потоков выполнения в Java, примененные к проблеме **«Производитель-Потребитель»**. Мы реализовали эквивалентное решение с использованием явных блокировок (ReentrantLock) и переменных условия (Condition), а в конце прибегли к высокоуровневой конкурентной структуре данных (BlockingQueue), которая значительно упрощает проблему.

Все три решения приводят к одному и тому же правильному поведению: производитель и потребитель координируются таким образом, чтобы не терять данные, не вступать в конфликт и не оставаться в постоянной блокировке. Тем не менее, существуют заметные различия между подходами с точки зрения сложности реализации и гибкости:

- Решение с ReentrantLock и Condition немного более **многословно**, но обеспечивает более тонкий контроль. Мы смогли разделить сигналы для разных условий (notEmpty/notFull), что становится очень полезным при расширении проблемы (например, буфер большего размера или несколько производителей/потребителей). Кроме того, ReentrantLock позволяет использовать такие опции, как fairness (справедливые замки) и tryLock(), которые могут предотвратить ситуации *голодания* или предоставить способы избежать полной блокировкиofficialcto.comofficialcto.com. С другой стороны, программист должен вручную управлять разблокировкой и внимательно следить за правильным использованием finally, чтобы не создавать тупиковых ситуаций. В целом, для большинства обычных случаев использования достаточно механизма synchronized по умолчанию, а ReentrantLock используется, когда действительно нужны эти расширенные функцииofficialcto.com.
- Решение с BlockingQueue демонстрирует мощь высокоуровневых абстракций в Java. Это самое **лаконичное** и наименее подверженное ошибкам решение, поскольку оно делегирует задачу синхронизации стандартному компоненту из библиотеки. Блокирующая очередь — это, по сути, специализированная реализация паттерна «производитель-потребитель», предоставляющая *буфер, безопасный для потоков*, и оптимизированные внутренние механизмы блокировки/разблокировки. Используя ее, наш код значительно упрощается, полностью устраняя явные логики ожидания и уведомления. Общая рекомендация заключается в том, чтобы, где это

возможно, использовать такие высокоуровневые инструменты (BlockingQueue, конкурентные коллекции, пулы потоков с исполнителями и т. д.), поскольку они уже были созданы и протестированы для предотвращения тонких ошибок синхронизации. Однако ручное решение с мониторами или блокировками остается важным для понимания основных концепций и для случаев, когда логика синхронизации не соответствует существующей утилите.

В заключение, решение проблемы производитель-потребитель позволило нам продемонстрировать, как различные механизмы синхронизации в Java могут быть использованы на практике. Мы подчеркнули важность правильного использования этих механизмов для получения корректных и надежных параллельных программ. Проведенные эксперименты подтверждают, что все произведенные элементы потребляются надлежащим образом, без незащищенного конкурентного доступа к буферу и без бесконечных блокировок. Сравнивая подходы, студенты могут лучше понять компромиссы между простотой, обеспечиваемой явной синхронизацией, и гибкостью явных блокировок и удобством высокоуровневых структур. Для большинства приложений ключевое слово synchronized и коллекции из java.util.concurrent будут достаточными и предпочтительными, в то время как механизмы, такие как ReentrantLock, становятся полезными, когда требуются дополнительные функции (множественные условия, справедливость, tryLock и т. д.)officialcto.com. Независимо от метода, программисту необходимо соблюдать основные принципы синхронизации (защита критических участков, ожидание условий в цикле, правильное уведомление потоков), чтобы избежать труднообнаружимых ошибок в многопоточных программах.

Лабораторная работа 6.

Синхронизация потоков выполнения с помощью классов синхронизации в задаче «производитель-потребитель»

Цель работы

Цель этой лабораторной работы — познакомить студентов с параллельным программированием на Java, в частности с **синхронизацией потоков выполнения** с помощью специальных механизмов. С помощью классической задачи «производитель-потребитель» работа направлена на демонстрацию как теоретических, так и практических аспектов синхронизации. Студенты поймут необходимость координации потоков, которые обращаются к общим ресурсам, и научатся использовать различные **классы синхронизации**, предлагаемые Java (такие как synchronized, ReentrantLock, Condition, BlockingQueue), для обеспечения правильного взаимодействия между потоками. Работа направлена на развитие навыков выявления критических участков кода, предотвращения гонок и блокировок, а также реализации надежных и эффективных решений для параллельных вычислений.

Цели

Выполняя эту лабораторную работу, студенты достигнут следующих целей:

- **Понимание потоков выполнения (threads)** — что такое поток выполнения, как создается и запускается поток в Java, а также разница между параллельным и конкурентным выполнением.
- **Признание необходимости синхронизации** — выявление ситуаций, в которых конкурентный доступ к общим ресурсам может привести к проблемам (условия гонки, неверные результаты), и мотивация использования механизмов взаимного исключения.
- **Изучение проблемы «Производитель-Потребитель»** – формулировка классической проблемы синхронизации и понимание ее сложностей (координация производства и потребления данных без потерь и конфликтов).
- **Использование мониторов Java (synchronized)** – приобретение навыков защиты критических участков с помощью синхронизированных блоков и использования методов wait()/notify()/notifyAll() для коммуникации между потоками.
- **Использование явных блокировок (ReentrantLock и Condition)** – ознакомление с ручной блокировкой и разблокировкой критических ресурсов и условным ожиданием с использованием объектов типа Condition.
- **Использование блокирующей очереди (BlockingQueue)** – понимание того, как высокоуровневые конкурентные коллекции могут упростить синхронизацию производитель-потребитель путем автоматической блокировки потоков, когда это необходимо.
- **Практическая реализация многопоточности** – разработка Java-приложения, решающего проблему производитель-потребитель с использованием нескольких методов синхронизации и сравнение полученных решений.
- **Анализ и выводы** – оценка преимуществ и ограничений различных используемых подходов к синхронизации, а также обеспечение передовых практик (например, предотвращение тупиковых ситуаций, использование try-finally при разблокировке и т. д.).

Пример реализации:

```
import java.util.ArrayDeque;

import java.util.Deque;

import java.util.Random;

import java.util.concurrent.Phaser;

import java.util.concurrent.locks.ReentrantLock;


public class ProdConsPhaser59 {

   static final int CAPACITY = 9;

   static final int TARGET_TOTAL = 59;

   static final int PRODUCERS = 3;
```

```
    static final int CONSUMERS = 4;
    // Четная фаза = FILL (заполнение), нечетная фаза = DRAIN (опорожнение)
    static class Depot {
        final Deque<Integer> buffer = new ArrayDeque<>(CAPACITY);
        final ReentrantLock lock = new ReentrantLock(true);
        final Random rnd = new Random();
        int producedTotal = 0;
        int consumedTotal = 0;
        volatile boolean done = false;

        void printFullIfNeeded() {
            if (buffer.size() == CAPACITY) {
                        System.out.printf(">>> Хранилище заполнено (%d/%d)%n", buffer.size(),
CAPACITY);
            }
        }
        void printEmptyIfNeeded() {
            if (buffer.isEmpty()) {
                System.out.printf("<<< Хранилище пусто (0/%d)%n", CAPACITY);
            }
        }

        // возвращает true, если что-то было произведено в этой попытке
        boolean tryProduce(String name) {
            lock.lock();
            try {
                if (producedTotal >= TARGET_TOTAL) return false; // мы закончили производство
                if (buffer.size() == CAPACITY)   return false;   // полный -> ждем фазу DRAIN
                int value = 31 + rnd.nextInt(59); // [31..89], только для удобства чтения
                buffer.addLast(value);
                producedTotal++;
                System.out.printf("%s произвел %d | запас=%d/%d | totalProd=%d%n",
```

```
                name, value, buffer.size(), CAPACITY, producedTotal);
            if (buffer.size() == CAPACITY) printFullIfNeeded();
            return true;
        } finally {
            lock.unlock();
        }
    }
    // возвращает true, если что-то было потреблено в этой попытке
    boolean tryConsume(String name) {
        lock.lock();
        try {
            if (buffer.isEmpty()) return false;
            int value = buffer.removeFirst();
            consumedTotal++;
            System.out.printf("%s потреблял %d | запас=%d/%d | totalCons=%d%n",
                    name, value, buffer.size(), CAPACITY, consumedTotal);

            if (buffer.isEmpty()) printEmptyIfNeeded();
            if (consumedTotal >= TARGET_TOTAL && buffer.isEmpty()) {
                done = true; // мы достигли цели и опустошили склад
            }
            return true;
        } finally {
            lock.unlock();
        }
    }
}
static class Producer extends Thread {
    private final Depot depot;
    private final Phaser phaser;
    Producer(Депо depot, Фазер phaser, Строка name) {
        super(name);
```

```
        this.depot = depot;
        this.phaser = phaser;
    }
    @Override public void run() {
        try {
            while (!depot.done) {
                int phase = phaser.getPhase();
                if (phase % 2 == 0) { // FILL
                    // Производим, пока склад не будет заполнен или не будет достигнута цель
                    boolean madeSomething = true;
                    while (!depot.done) {
                        madeSomething = depot.tryProduce(getName());
                        if (!madeSomething) break; // заполнен или произведено все
                    }
                    // сигнализируем барьер: мы закончили вклад в этой фазе
                    phaser.arriveAndAwaitAdvance();
                } else {
                    // Фаза DRAIN -> производители останавливаются
                    phaser.arriveAndAwaitAdvance();
                }
            }
        } catch (Исключение игнорируется) {
        } finally {
            // в конце, грациозная отмена регистрации
            try { phaser.arriveAndDeregister(); } catch (IllegalStateException ignored) {}
        }
    }
}

static class Consumer extends Thread {
    private final Depot депо;
    private final Phaser phaser;
```

```
    Consumer(Депо депо, Фазер фазер, Строка имя) {
        super(name);
        this.depot = depot;
        this.phaser = phaser;
    }
    @Override public void run() {
        try {
            while (!depot.done) {
                int phase = phaser.getPhase();
                if (phase % 2 == 1) { // DRAIN
                    // Потребляем, пока склад не опустеет (или не достигнем цели)
                    boolean tookSomething = true;
                    while (!depot.done) {
                        tookSomething = depot.tryConsume(getName());
                        if (!tookSomething) break; // пусто -> переходим к следующей фазе
                    }
                    phaser.arriveAndAwaitAdvance();
                } else {
                    // Фаза FILL -> потребители остаются
                    phaser.arriveAndAwaitAdvance();
                }
            }
        } catch (Исключение игнорируется) {
        } finally {
            try { phaser.arriveAndDeregister(); } catch (IllegalStateException ignored) {}
        }
    }
}
public static void main(String[] args) throws InterruptedException {
    Depot depot = new Depot();
    // Фазер с 3 производителями + 4 потребителями (=7 «партий»)
    Phaser phaser = new Phaser(PRODUCERS + CONSUMERS);
```

```
Thread[] producers = new Thread[PRODUCERS];
Thread[] consumers = new Thread[CONSUMERS];
for (int i = 0; i < PRODUCERS; i++) {
   producers[i] = new Producer(depot, phaser, "Producator-" + (i + 1));
}
for (int i = 0; i < CONSUMERS; i++) {
   consumers[i] = new Consumer(depot, phaser, "Потребитель-" + (i + 1));
}
// На старте: фаза 0 (FILL). Потребители будут ждать, производители заполняют.
for (Thread t : producers) t.start();
for (Thread t : consumers) t.start();
// Ждем логического завершения
for (Thread t : producers) t.join();
for (Thread t : consumers) t.join();
 System.out.println("=== Готово: было произведено и потреблено " + TARGET_TOTAL
+ " объектов в циклах FILL→DRAIN. ===");
}}
```

Задачи для решения:

Даны X производителей, которые случайным образом генерируют F объектов, которые потребляются Y потребителями. Необходимо отобразить информацию о производстве и потреблении объектов, сообщения о случаях, когда «склад пуст или полон». Размер склада равен D. Производители заполняют склад D объектами. Потребители не могут потреблять, пока склад не будет полон. После этого потребители потребляют. Производители не могут производить, пока склад не будет пуст. Все операции выполняются до тех пор, пока не будут произведены и потреблены Z объектов.

Значения для X, Y, Z, D, F указаны в таблице 3.

Задача выполняется в группе. Один член группы выполняет задачу и синхронизацию производителя. Второй член группы выполняет задачу и синхронизацию потребителя.

Задача может быть выполнена с помощью классических потоков или пулов потоков. Синхронизация выполняется только с помощью классов синхронизации из Java.

Таблица 1 Варианты выполнения задачи

№	X	Y	Z	D	Тип Объекты
1	2	3	40	8	Четные числа
2	3	4	45	5	Нечетные числа

3	4	3	50	10	Вокал
4	3	2	66	11	Согласные
5	2	5	60	12	Четные числа
6	2	4	42	7	Нечетные числа
7	3	3	42	6	Вокал
8	4	2	45	5	Согласные
9	5	2	44	4	Четные числа
10	3	3	48	8	Нечетные числа
F - каждый производитель производит по 1 предмету за раз для всех вариантов.					

Критерии оценки:

1. Создание и инициализация потоков для выполнения задач.
2. Выбор форм синхронизации потоков.
3. Синхронизация потоков с подходящими формами.
4. Создание графического интерфейса программы.
5. **Корректность кода** — проверка правильности кода, отсутствие ошибок в работе.
6. **Соблюдение инструкций и требований** — проверка правильности требований задачи, таких как количество потоков выполнения.
7. **Оптимизация кода** — оценка эффективности кода в использовании ресурсов и избегание кода.
8. **Соблюдение срока сдачи** — оценка баллов в зависимости от пунктуальности, сдана ли работа в установленный срок.
9. **Оценка знаний** — объяснения, данные о процессе выполнения работы, что может включать описание основных функций и использованной логики.
10. Использование студентом ИИ.

Для получения оценки 5-6 обязательны критерии 1,2,3,5,8,9

Для получения оценки 7-8 обязательны критерии 1,2,3,4,5,6,8,9

Для получения оценки 9-10 обязательны критерии 1-9

Если был использован критерий 10, оценка снижается на 2 балла, только если студент разъяснил функционирование кода. В противном случае лабораторная работа не принимается.

8. Контрольные вопросы:

Чтобы проверить понимание теоретических и практических аспектов, охваченных в этой лабораторной работе, ответьте на следующие вопросы:

1. Объясните концепцию *критического участка*. Почему необходима синхронизация, когда несколько потоков обращаются к одному и тому же ресурсу?

2. Кратко сравните механизмы synchronized и ReentrantLock. Назовите два преимущества ReentrantLock по сравнению с synchronized и одну дополнительную ответственность, которую он налагает на программиста.

3. Что такое объект Condition и как его получить? Чем отличается использование await()/signal() от wait()/notify()?

4. В примере проблемы производитель-потребитель, почему мы использовали два отдельных объекта Condition (notEmpty и notFull)? Что произошло бы, если бы мы использовали один объект Condition для обоих типов ожидания?

5. Что такое блокирующая очередь (*blocking queue*) и как она облегчает коммуникацию между производителями и потребителями? Объясните своими словами, как работают методы put() и take() в BlockingQueue.

6. Приведите пример как минимум двух конкретных реализаций интерфейса BlockingQueue в Java и укажите различия между ними (например, связанные с емкостью или типом порядка).

7. В коде решения (с ReentrantLock), что может произойти, если мы забудем вызвать lock.unlock() в блоке finally метода get()? А если мы вызовем notFull.signal() перед освобождением блокировки?

8. Кратко объясните, как вы бы расширили созданное приложение для случая, когда производитель и потребитель должны работать бесконечно (а не только с 10 элементами). Как бы вы сигнализировали об окончании потоков при отсутствии конечного числа элементов? (Подсказка: *poison pill* — специальный элемент окончания в конце очереди)

Библиография

1. **FCIM, UTM Кишинев** – *Классическая проблема производитель-потребитель – Синхронизация потоков выполнения*, лабораторная работа, Факультет вычислительной техники, информатики и микроэлектроникиelse.fcim.utm.mdelse.fcim.utm.md (румынский язык).

2. **Cosmina Ivan** – *Технологии и приложения в параллельных и распределенных вычислениях*, Издательство U.T. Pres, Клуж-Напока, 2013biblioteca.utcluj.robiblioteca.utcluj.ro (на румынском языке).

3. **Кшиштоф Дымек** – *Java: продвинутая конкурентность* (перевод на русский язык), Habr, 2022habr.com (русский язык).

4. **Official CTO Blogs** – *Explicit Locks in Java – ReentrantLock*, онлайн-статьяofficialcto.comofficialcto.com (на английском языке).

5. **Baeldung** – *Guide to java.util.concurrent.BlockingQueue*, онлайн-статьябaeldung.com (английский язык).