



ТИНЬКОФФ

Многопоточность

Использование потоков в реальном коде



О себе



Пришел стажером в 2020 году



Старший разработчик в Тинькофф



Развиваю партнерские сервисы Выгоды



Преподаю в Тинькофф образовании



Программа модуля



Основы работы с потоками и их устройство

Рассмотрим что такое потоки и процессы, когда, как и зачем их использовать



Использование потоков в реальном коде

Разберем что предлагает библиотека Java для работы с потоками



Продвинутые темы

Изучим Java memory model и посмотрим на виртуальные потоки

План лекции



Примитивы синхронизации

Посмотрим, что предоставляет нам пакет
java.util.concurrent



Потокобезопасные коллекции и atomic

Разберемся как и когда использовать коллекции
из java.util.concurrent



ThreadPool и Executors

Изучим что такое и как работать с пулом потоков



CompletableFuture

Как удобно можно запускать многопоточные
программы

План лекции



Примитивы синхронизации и atomic

Посмотрим, что предоставляет нам пакет
java.util.concurrent



Потокобезопасные коллекции

Разберемся как и когда использовать коллекции
из java.util.concurrent



ThreadPool и Executors

Изучим что такое и как работать с пулом потоков



CompletableFuture

Как удобно можно запускать многопоточные
программы



ТИНЬКОФФ

java.util.concurrent.locks

Locks



Зачем нам нужны какие-то Lock есть же synchronized?

```
public interface Lock {  
    void lock();  
  
    void lockInterruptibly() throws InterruptedException;  
  
    boolean tryLock();  
  
    boolean tryLock(long var1, TimeUnit var3) throws InterruptedException;  
  
    void unlock();  
  
    Condition newCondition();  
}
```

Locks Плюсы

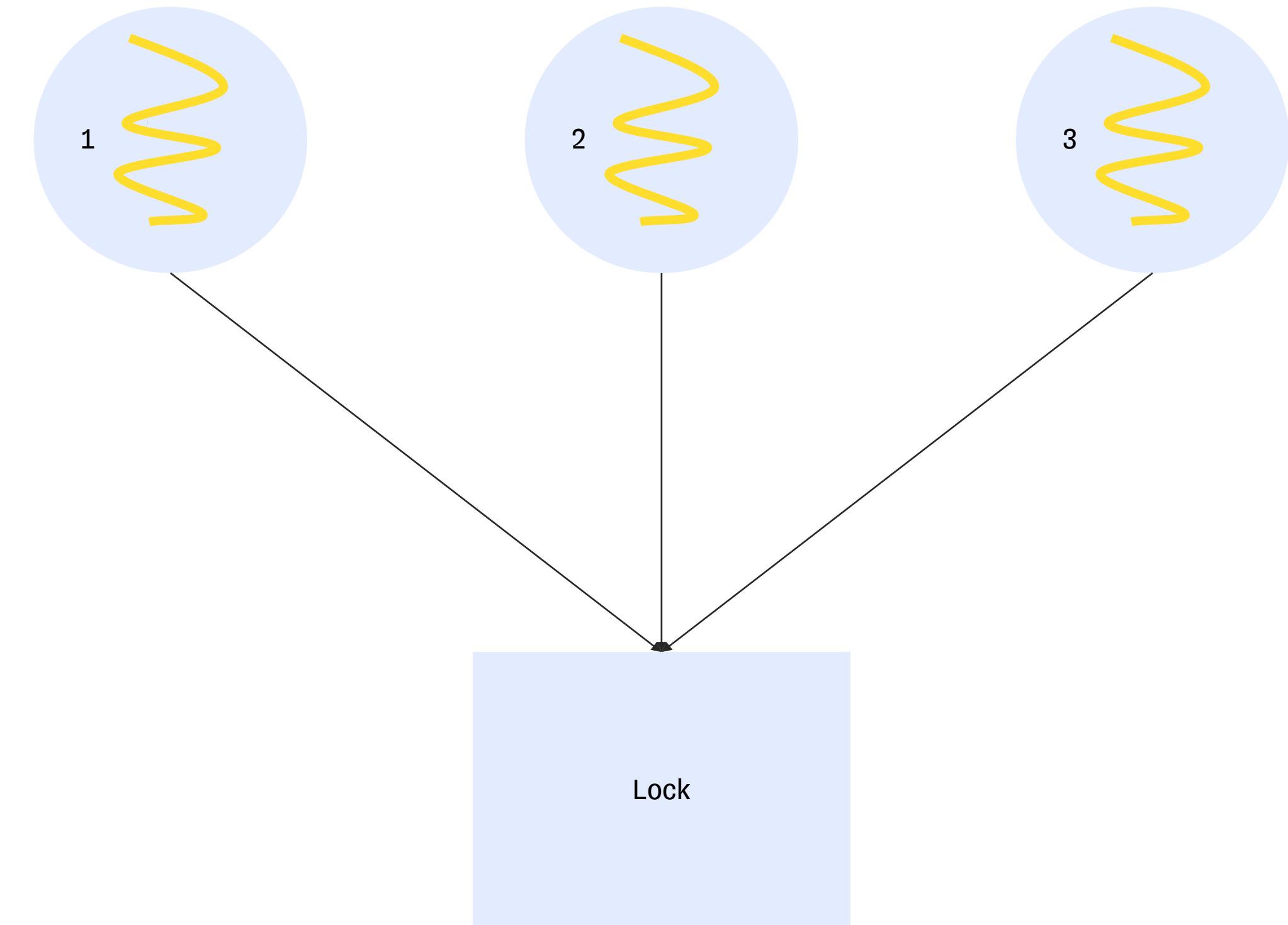
- Locks API дает возможность расширять блок синхронизации
- Synchronized не гарантирует fairness
- tryLock() – позволяет ожидать блокировку с timeout

```
public interface Lock {  
    void lock();  
  
    void lockInterruptibly() throws InterruptedException;  
  
    boolean tryLock();  
  
    boolean tryLock(long var1, TimeUnit var3) throws InterruptedException;  
  
    void unlock();  
  
    Condition newCondition();  
}
```

Fairness



Поочередный захват блокировки
несколькими потоками



Locks



Паттерн использования



```
Lock lock = ...;  
lock.lock();  
try {  
    // access to the shared resource  
} finally {  
    lock.unlock();  
}
```

ReentrantLock



ОДИН ПОТОК МОЖЕТ БРАТЬ
БЛОКИРОВКУ НЕСКОЛЬКО РАЗ



Реализация интерфейса Lock



ДЛЯ ГАРАНТИИ FAIRNESS TRUE В
КОНСТРУКТОРЕ



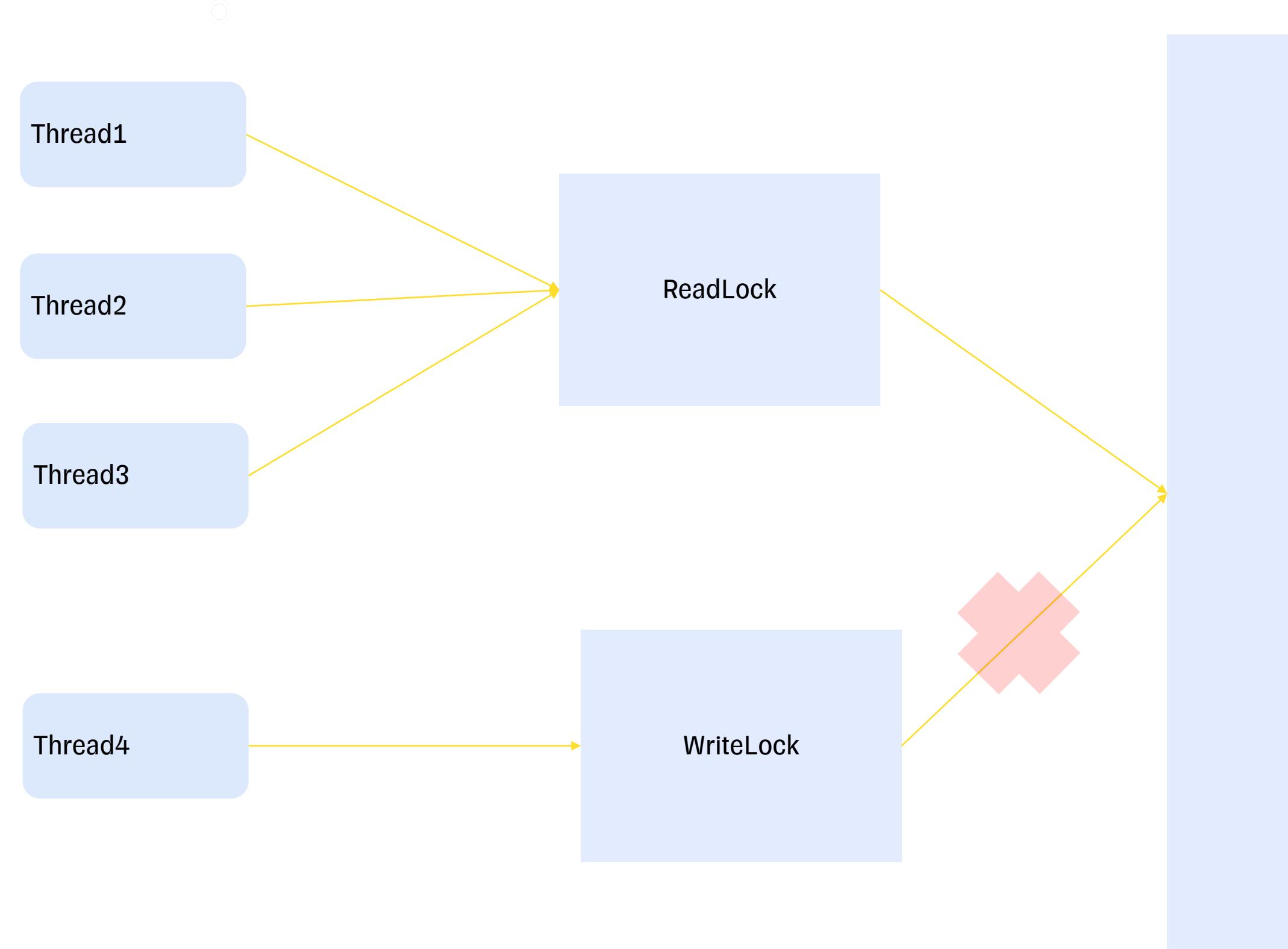
```
private final Lock lock = new ReentrantLock(true);
private final Map<String, String> storage = new HashMap<>();

public void put(String key, String value) {
    lock.lock();
    try {
        storage.put(key, value);
    } finally {
        lock.unlock();
    }
}

public String get(String key) {
    lock.lock();
    try {
        return storage.get(key);
    } finally {
        lock.unlock();
    }
}
```

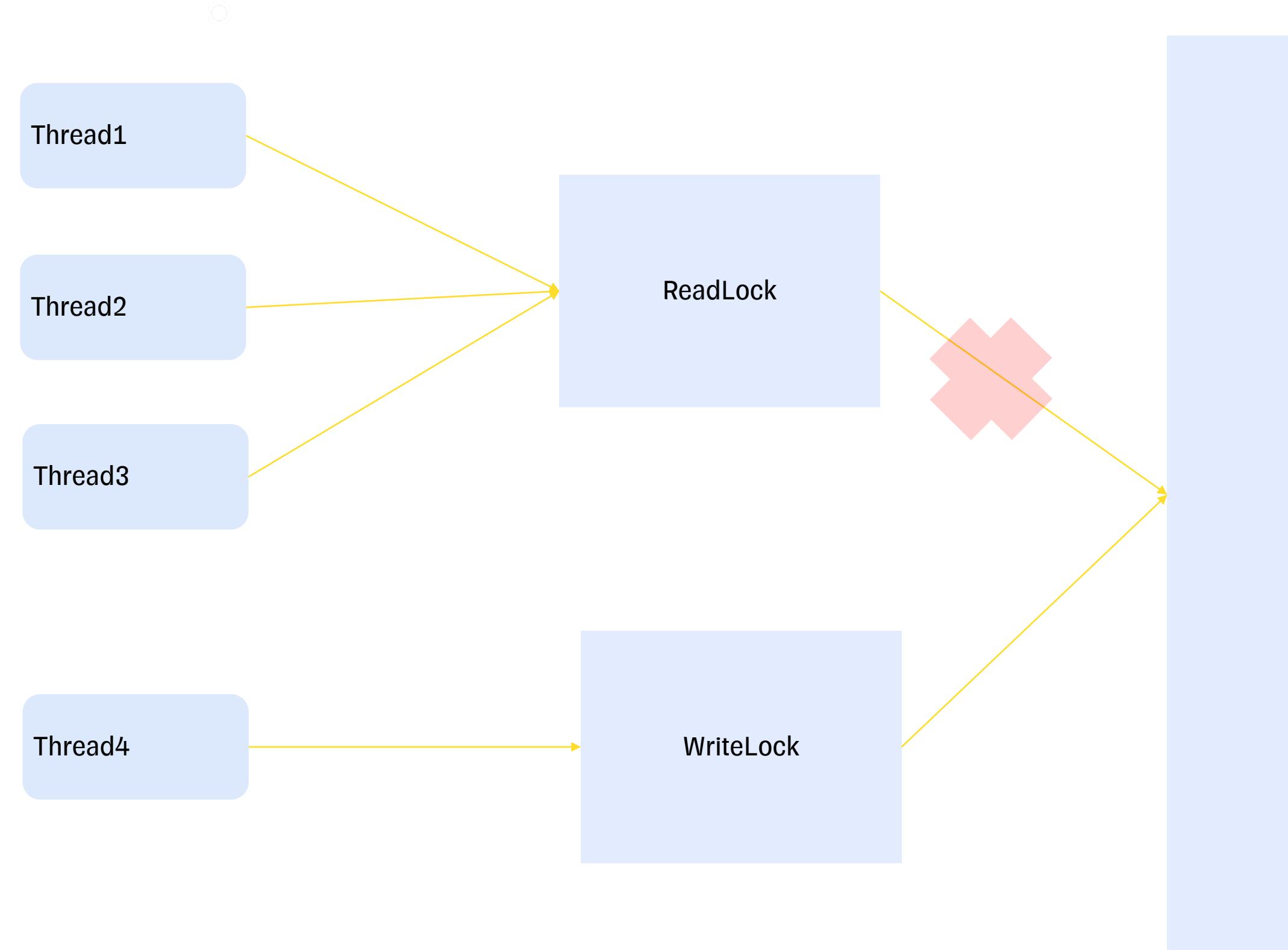
ReadWriteLock

- Несколько потоков могут брать один ReadLock
- Если какой-то поток взял WriteLock, то другие блокируются
- Оптимальнее запросы на чтение



ReadWriteLock

- Несколько потоков могут брать один ReadLock
- Если какой-то поток взял WriteLock, то другие блокируются
- Оптимальнее запросы на чтение



ReadWriteLock



Несколько потоков могут брать один ReadLock



Если какой-то поток взял WriteLock, то другие блокируются



Оптимальнее запросы на чтение



```
private final ReadWriteLock lock = new ReentrantReadWriteLock(true);
private final Map<String, String> storage = new HashMap<>();

public void put(String key, String value) {
    lock.writeLock().lock();
    try {
        storage.put(key, value);
    } finally {
        lock.writeLock().unlock();
    }
}

public String get(String key) {
    lock.readLock().lock();
    try {
        return storage.get(key);
    } finally {
        lock.readLock().unlock();
    }
}
```

Condition



Объект с условием, позволяет
блокировать поток



Поток блокируется до выполнения
условия



Нотифицирует другие потоки,
чтобы те разблокировались



```
public interface Condition {  
    void await() throws InterruptedException;  
  
    void awaitUninterruptibly();  
  
    long awaitNanos(long var1) throws InterruptedException;  
  
    boolean await(long var1, TimeUnit var3) throws InterruptedException;  
  
    boolean awaitUntil(Date var1) throws InterruptedException;  
  
    void signal();  
  
    void signalAll();  
}
```

Condition



Реализация блокирующей очереди



В очередь можно вставлять до *capacity* элементов, дальше поток заблокируется



Если очередь пуста то метод *poll()* заблокирует поток

```
private final Queue<T> queue = new ArrayDeque<>();
private final ReentrantLock lock = new ReentrantLock();
private final Condition notEmpty = lock.newCondition();
private final Condition notFull = lock.newCondition();
private final int capacity;

public CustomBlockingQueue(int capacity) {
    this.capacity = capacity;
}

public void push(T e) throws InterruptedException {
    lock.lock();
    try {
        while(queue.size() == capacity) {
            notFull.await();
        }
        queue.add(e);
        notEmpty.signalAll();
    } finally {
        lock.unlock();
    }
}

public T poll() throws InterruptedException {
    lock.lock();
    try {
        while(queue.isEmpty()) {
            notEmpty.await();
        }
        return queue.poll();
    } finally {
        notFull.signalAll();
        lock.unlock();
    }
}
```



ТИНЬКОФФ

Дополнительные примитивы синхронизации

CountDownLatch

Есть счетчик, как только он опускается до 0 блокировка снимается

→ *countDown()* – уменьшает значение счетчика

→ *await()* – блокирует поток, пока счетчик не равен 0

count = 5



Conditions:



CountDownLatch

CountDownLatch

Есть счетчик, как только он опускается до 0 блокировка снимается

Блокируем один thread пока другие threads не закончат свои задачи

```
private static class Worker implements Runnable {

    private final CountDownLatch latch;

    private Worker(CountDownLatch latch) {
        this.latch = latch;
    }

    @Override
    public void run() {
        randomSleep();
        System.out.println("Prepare to start execution " + Thread.currentThread().getName());
        latch.countDown();
        try {
            latch.await();
            randomSleep();
            System.out.println("Do some work " + Thread.currentThread().getName());
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    private void randomSleep() {
        try {
            Thread.sleep(ThreadLocalRandom.current().nextLong(300, 1000));
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

public static void main(String[] args) {
    CountDownLatch latch = new CountDownLatch(5);
    Stream.generate(() -> new Thread(new Worker(latch)))
        .limit(5)
        .forEach(Thread::start);
}
```

CyclicBarrier

→ Похож на CountDownLatch

→ Барьер можно
переиспользовать

parties = 3



CyclicBarrier



Похож на CountDownLatch



Барьер можно
переиспользовать

```
private static class Worker implements Runnable {  
  
    private final CyclicBarrier barrier;  
  
    private Worker(CyclicBarrier barrier) {  
        this.barrier = barrier;  
    }  
  
    @Override  
    public void run() {  
        randomSleep();  
        System.out.println("Prepare to start execution " + Thread.currentThread().getName());  
        randomSleep();  
        System.out.println("Reach common point and waiting other workers " + Thread.currentThread().getName());  
        try {  
            barrier.await();  
            System.out.println("Do some work " + Thread.currentThread().getName());  
        } catch (BrokenBarrierException | InterruptedException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}  
  
public static void main(String[] args) {  
    CyclicBarrier barrier = new CyclicBarrier(3);  
    Stream.generate(() -> new Thread(new Worker(barrier)))  
        .limit(6)  
        .forEach(Thread::start);  
}
```

Semaphore

Есть счетчик, как только он опускается до 0 блокировка снимается



Блокируем один thread пока другие threads не закончат свои задачи



Semaphore

Permits = 3



Semaphore

Есть счетчик, как только он опускается до 0 блокировка снимается



Блокируем один thread пока другие threads не закончат свои задачи



```
private static class Worker extends Thread {  
  
    private final Semaphore semaphore;  
  
    private Worker(Semaphore semaphore) {  
        this.semaphore = semaphore;  
    }  
  
    @Override  
    public void run() {  
        try {  
            semaphore.acquire();  
            System.out.println("acquire semaphore " + Thread.currentThread().getName());  
            System.out.println("work " + Thread.currentThread().getName());  
            threadSleep();  
        } catch (InterruptedException e) {  
            throw new RuntimeException(e);  
        } finally {  
            System.out.println("release semaphore " + Thread.currentThread().getName());  
            semaphore.release();  
        }  
    }  
  
    public static void main(String[] args) {  
        Semaphore semaphore = new Semaphore(2);  
        Stream.generate(() -> new Worker(semaphore))  
            .limit(10)  
            .forEach(Thread::start);  
    }  
}
```

План лекции



Примитивы синхронизации и atomic

Посмотрим, что предоставляет нам пакет
java.util.concurrent



Потокобезопасные коллекции

Разберемся как и когда использовать коллекции
из java.util.concurrent



ThreadPool и Executors

Изучим что такое и как работать с пулом потоков



CompletableFuture

Как удобно можно запускать многопоточные
программы



ТИНЬКОФФ

Atomics

Неблокирующие алгоритмы



Минусы синхронизации

Когда много потоков
конкурируют за блокировку
тратится много ресурсов на их
синхронизацию



Compare and swap

Можно использовать
неблокирующие алгоритмы,
основанные на cas операциях



Atomics и неблокирующие коллекции

В Java есть Atomic переменные и
специальные неблокирующие
коллекции

Atomics



Основаны на cas операциях



Могут быть использованы, как
volatile переменные

AtomicBoolean

AtomicInteger

AtomicLongArray

AtomicLong

AtomicIntegerArray

AtomicReferenceArray

AtomicReference

Atomics



Основаны на CAS операциях



Могут быть использованы, как
volatile переменные

```
public static void main(String[] args) {
    var value = new AtomicInteger();
    var incrementor = new Thread(() -> {
        for (int i = 0; i < 100_000; i++) {
            value.incrementAndGet();
        }
    });
    var decrementor = new Thread(() -> {
        for (int i = 0; i < 100_000; i++) {
            value.decrementAndGet();
        }
    });
    incrementor.start();
    decrementor.start();

    try {
        incrementor.join();
        decrementor.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println(value.get());
}
```



ТИНЬКОФФ

Потокобезопасные коллекции

Synchronized коллекции



Collections.synchronized...()



Возвращает коллекцию, операции которой используют synchronized



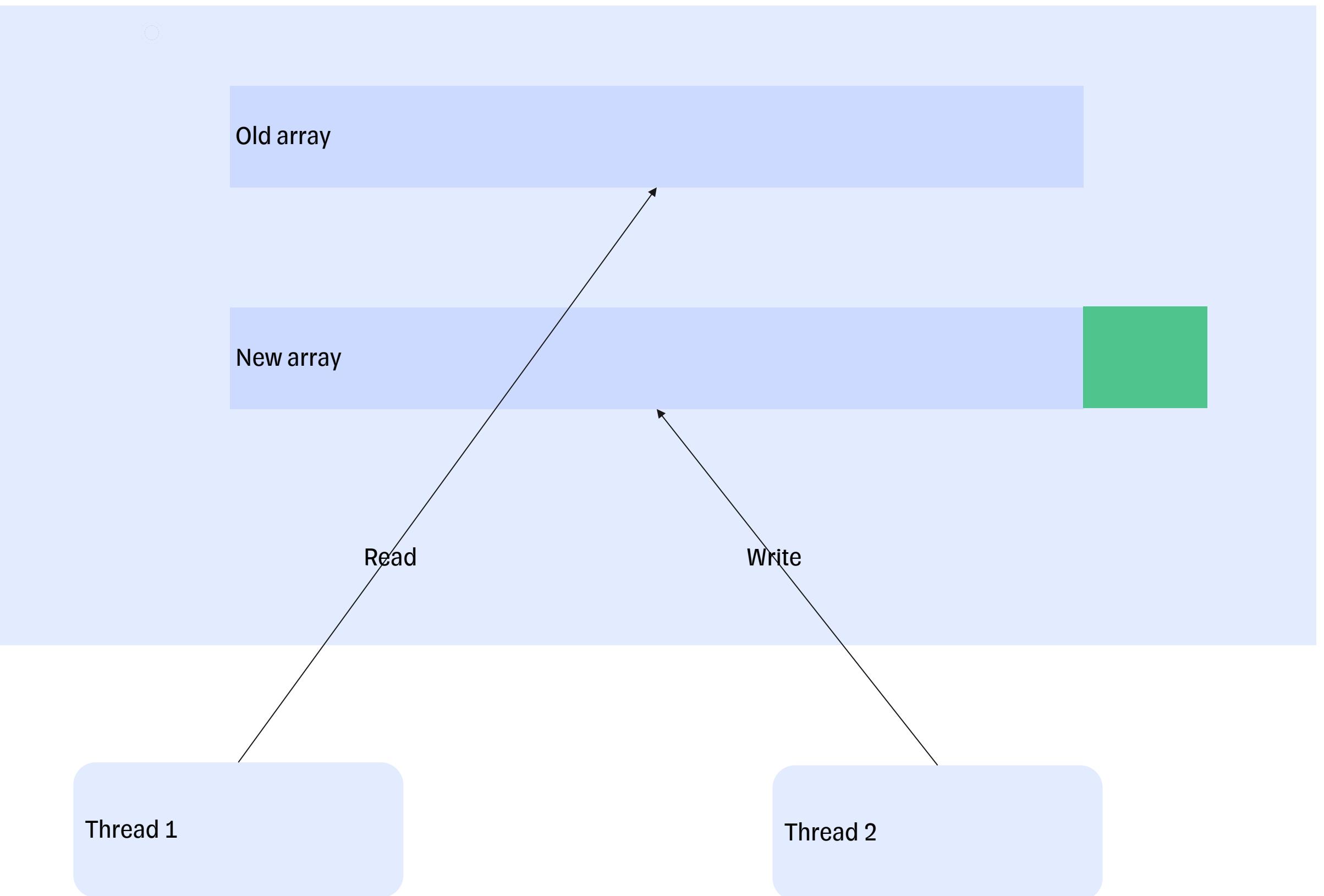
Опять блокировки -> снижение производительности

```
public static void main(String[] args) throws InterruptedException {
    CountDownLatch latch = new CountDownLatch(100000);
    List<Integer> values = Collections.synchronizedList(new ArrayList<>());
    Stream.generate(() -> new Thread(() -> {
        values.add(ThreadLocalRandom.current().nextInt(0, 1000));
        latch.countDown();
    }))
    .limit(100000)
    .forEach(Thread::start);
    latch.await();
    System.out.println(values.size());
}
```

CopyOnWriteArrayList

→ На каждую операцию
модификации add, remove
создается копия списка

→ Хорошо работает, когда на одну
операцию записи приходится
много операций чтения



ConcurrentHashMap



Замена HashMap

Не блокируется при чтении и
редко блокируется при записи



Имеет атомарные методы

- `putIfAbsent(key, value)`
- `remove(key, value)`
- `replace(key, oldValue, newValue)`



Используется при кэшировании

В бэкенд сервисах часто
используется для *in memory*
кэша

ConcurrentSkipListMap



Замена TreeMap

Гарантируется среднее время
операций за $O(\log n)$



Имеет доп. методы

- ceilingEntry/Key
- floorEntry/Key



Наследуется от ConcurrentNavigableMap

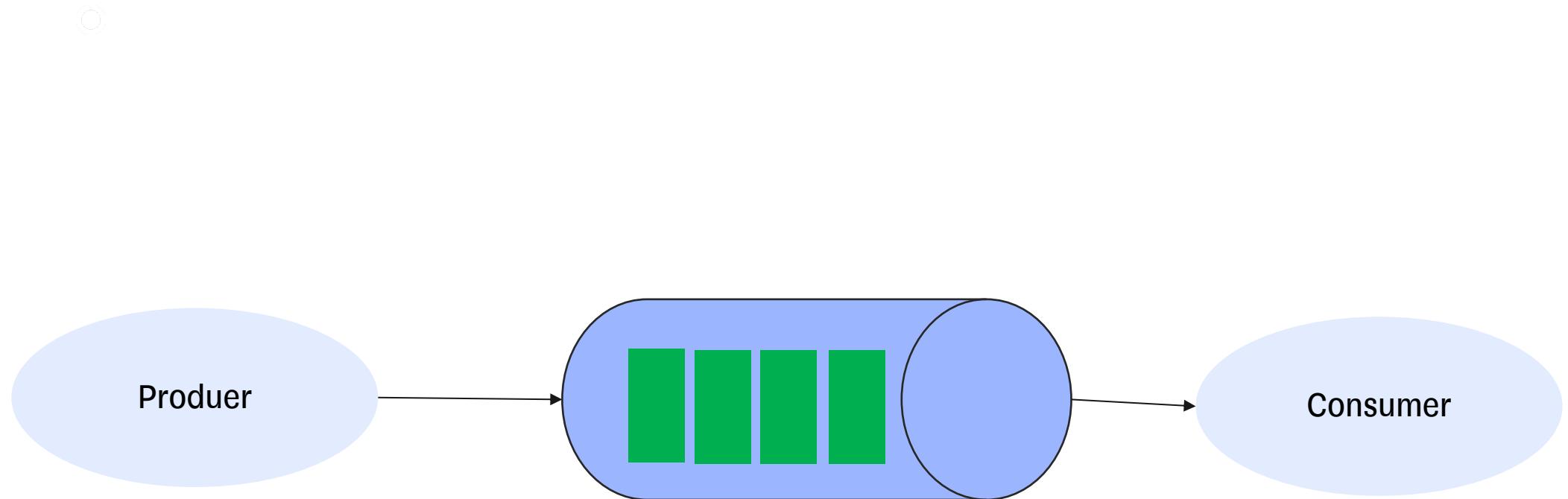


ТИНЬКОФФ

Producer consumer

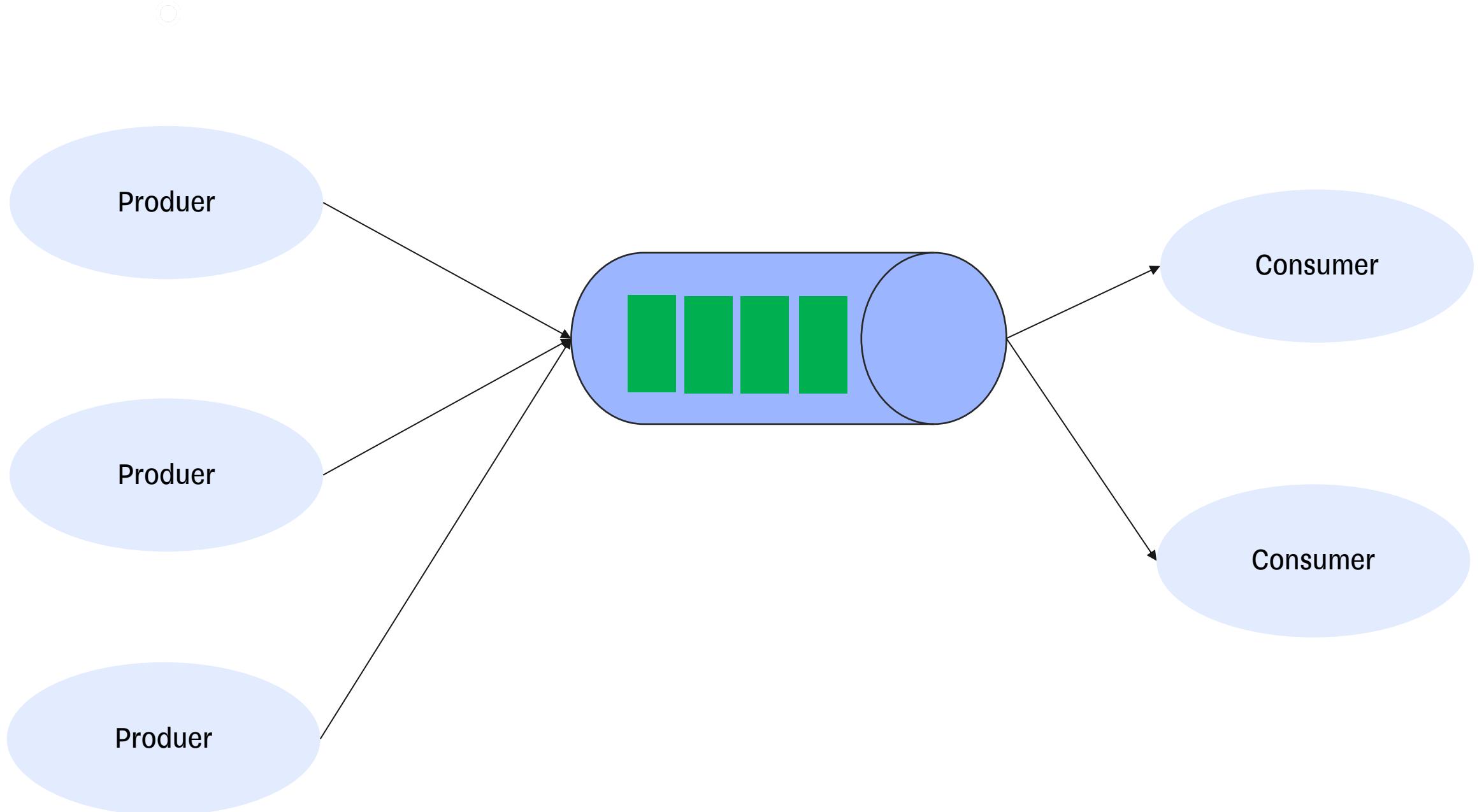
Producer consumer

- Producer – создает и кладет объекты в buffer
- Consumer – читает и обрабатывает объекты из buffer



Сложности

- Producer и Consumer может быть много
- У producer и consumer может быть разная скорость
- Нужно уметь синхронизировать consumer, чтобы они не обрабатывали сообщения 2 раза



BlockingQueue

Интерфейс, позволяющий использовать очередь не беспокоясь о синхронизации

→ ***put()* – блокирует поток если очередь полная**

→ ***take()* – блокирует поток если очередь пустая**

```
private static final BlockingQueue<Integer> queue = new LinkedBlockingDeque<>(5);

public static void main(String[] args) {
    Stream.generate(() -> new Thread(ProducerConsumerExample::produce))
        .limit(3)
        .forEach(Thread::start);
    Stream.generate(() -> new Thread(ProducerConsumerExample::consume))
        .limit(2)
        .forEach(Thread::start);
}

private static void produce() {
    while (true) {
        int value = ThreadLocalRandom.current().nextInt(100, 1000);
        try {
            randomSleep();
            queue.put(value);
            System.out.printf("[%s] Produce value = %d%n", Thread.currentThread().getName(), value);
        } catch (InterruptedException e) {
            break;
        }
    }
}

private static void consume() {
    while (true) {
        try {
            randomSleep();
            int value = queue.take();
            System.out.printf("[%s] Consume value = %d%n", Thread.currentThread().getName(), value);
        } catch (InterruptedException e) {
            break;
        }
    }
}
```

Пример с condition



```
private final Queue<T> queue = new ArrayDeque<>();
private final ReentrantLock lock = new ReentrantLock();
private final Condition notEmpty = lock.newCondition();
private final Condition notFull = lock.newCondition();
private final int capacity;

public CustomBlockingQueue(int capacity) {
    this.capacity = capacity;
}

public void push(T e) throws InterruptedException {
    lock.lock();
    try {
        while(queue.size() == capacity) {
            notFull.await();
        }
        queue.add(e);
        notEmpty.signalAll();
    } finally {
        lock.unlock();
    }
}

public T poll() throws InterruptedException {
    lock.lock();
    try {
        while(queue.isEmpty()) {
            notEmpty.await();
        }
        return queue.poll();
    } finally {
        notFull.signalAll();
        lock.unlock();
    }
}
```

План лекции



Примитивы синхронизации и atomic

Посмотрим, что предоставляет нам пакет
java.util.concurrent



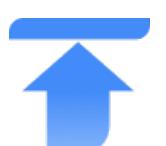
Потокобезопасные коллекции

Разберемся как и когда использовать коллекции
из java.util.concurrent



ThreadPool и Executors

Изучим что такое и как работать с пулом потоков



CompletableFuture

Как удобно можно запускать многопоточные
программы



ТИНЬКОФФ

ThreadPool и Executors

Threads in real life



Просто скачиваем картинки



Что не нравится в этом коде?

```
public static void main(String[] args) throws IOException, InterruptedException {
    List<Thread> workers = Stream.generate(() -> new Thread(ImagesExample::downloadImage))
        .limit(5)
        .toList();
    workers.forEach(Thread::start);
    for (var worker : workers) {
        worker.join();
    }
    System.out.println("Finished download images");
}

private static void downloadImage() {
    try {
        URL url = new URL("https://source.unsplash.com/featured/300x201");
        InputStream in = new BufferedInputStream(url.openStream());
        Path path = Paths.get("image-" + UUID.randomUUID() + ".jpg");
        Files.write(path, in.readAllBytes());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Threads in real life

→ Приходится писать однотипный код

→ Каждый раз создается новый поток

→ Поток создавать дорого

→ А если бы картинок было бы 1К

```
public static void main(String[] args) throws IOException, InterruptedException {
    List<Thread> workers = Stream.generate(() -> new Thread(ImagesExample::downloadImage))
        .limit(5)
        .toList();
    workers.forEach(Thread::start);
    for (var worker : workers) {
        worker.join();
    }
    System.out.println("Finished download images");
}

private static void downloadImage() {
    try {
        URL url = new URL("https://source.unsplash.com/featured/300x201");
        InputStream in = new BufferedInputStream(url.openStream());
        Path path = Paths.get("image-" + UUID.randomUUID() + ".jpg");
        Files.write(path, in.readAllBytes());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

ThreadPool

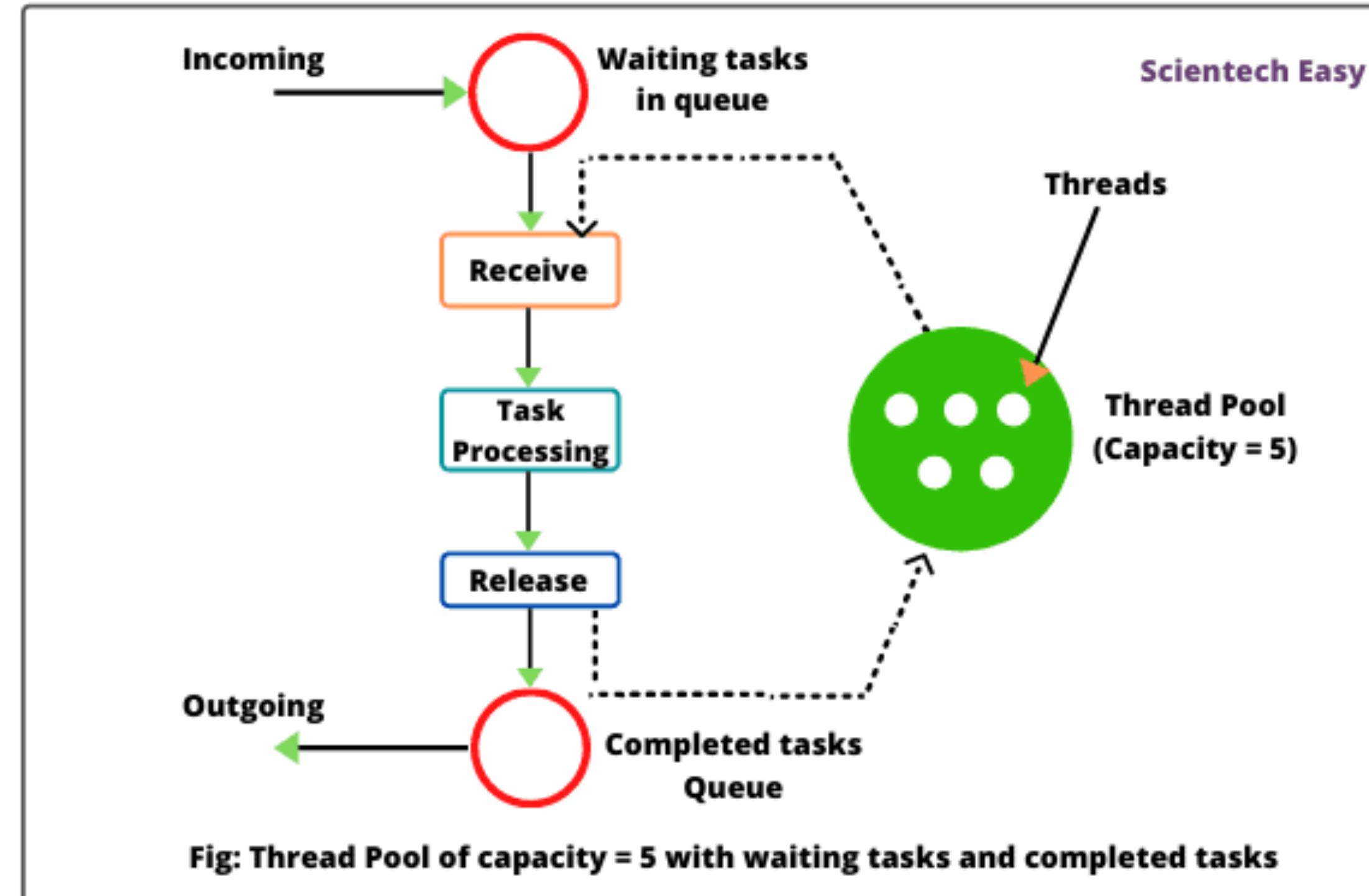


Что за бассейн?



ThreadPool

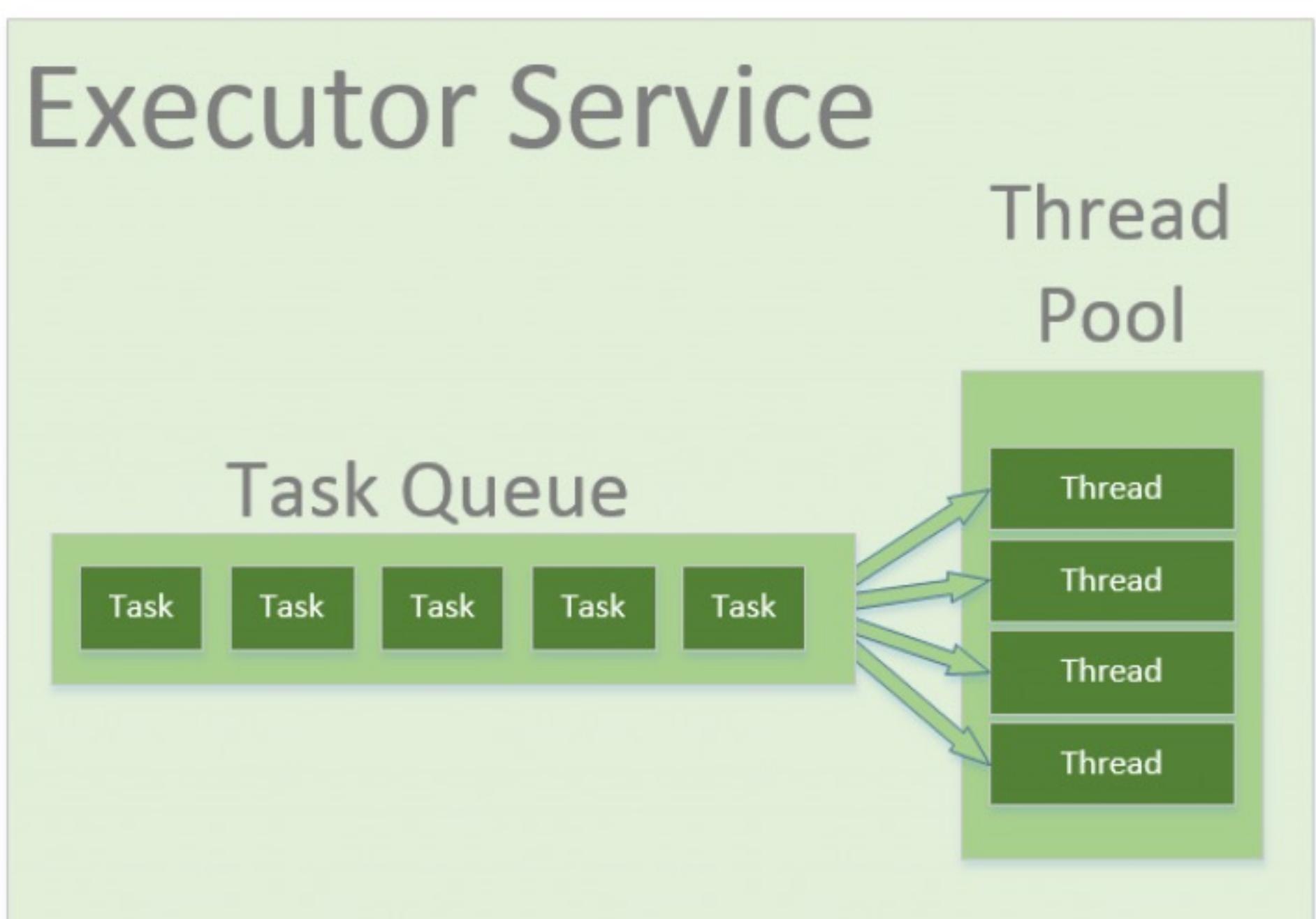
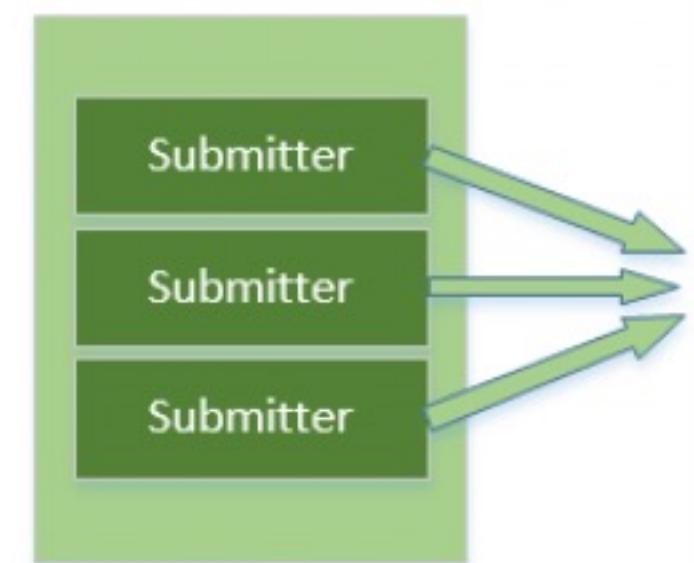
- Набор потоков, который создается сразу при старте
- Потоки переиспользуются
- После использования поток опять кладется в pool и не создается по новой



ExecutorService

- Использует пул потоков
- Использует блокирующую очередь
- Задачи помещаются в очередь и распределяются по потокам

Task Submitters



ExecutorService

- *Future<?> submit(Runnable)* – отправляет задачу на выполнение без возвращаемого значения
- *Future<T> submit(Callable<T>)* – отправляет задачу на выполнение с возвращаемым значением
- *List<Future<T>> invokeAll(Collection<? Extends Callable<T>>)* – отправляет несколько задач на выполнение
- *Future<T> invokeAny(Collection<? Extends Callable<T>>)* – отправляет несколько задач на выполнение и возвращает результат одного которое закончится успешно

ExecutorService



***future.get()* – блокирует поток,
пока задача не завершится**

```
public static void main(String[] args) {  
    ExecutorService executorService = Executors.newSingleThreadExecutor();  
    Future<?> future = executorService.submit(ImagesExecutorsExample::downloadImage);  
    future.get();  
    System.out.println("Finish downloading image");  
}
```

Future



- *boolean isDone()* – выполнилась ли задача у ExecutorService
- *boolean cancel(boolean)* – говорит ExecutorService прекратить выполнение операции
- *boolean isCancelled()* – отменена ли была задача
- *get()* – блокирует поток, пока ExecutorService не выполнит задачу
- *get(long, TimeUnit)* – выбросит TimeoutException, если задача не выполнится за переданное время

ExecutorService

```
ExecutorService executorService = Executors.newFixedThreadPool(5);
Callable<Void> callable = () -> {
    downloadImage();
    return null;
};
var tasks = Stream.generate(() -> callable).limit(5).toList();
List<Future<Void>> futures = executorService.invokeAll(tasks);
for (var future: futures) {
    future.get();
}
executorService.shutdown();
System.out.println("Finish download images");
```

Executors

→ Класс с методами создания разных реализаций ExecutorService

→ Для разных целей используются свои реализации

→ FixedThreadPool – Пул с фиксированным кол – вом потоков

→ SingleThreadPool – Пул с одним потоком

→ CachedThreadPool – Кэширует потоки в рамках небольшого отрезка времени

→ ScheduledThreadPool – Ставит задачу на время

План лекции



Примитивы синхронизации и atomic

Посмотрим, что предоставляет нам пакет
java.util.concurrent



Потокобезопасные коллекции

Разберемся как и когда использовать коллекции
из java.util.concurrent



ThreadPool и Executors

Изучим что такое и как работать с пулом потоков



CompletableFuture

Как удобно можно запускать многопоточные
программы



ТИНЬКОФФ

CompletableFuture

CompletableFuture



Позволяет еще легче писать
ассинхронный код



Есть возможность писать с
использованием ExecutorService

```
ExecutorService executorService = Executors.newFixedThreadPool(5);
        var tasks = Stream.generate(() ->
CompletableFuture.runAsync(ImagesExecutorsExample::downloadImage, executorService))
        .limit(5)
        .toArray(CompletableFuture[ ]::new);
CompletableFuture.allOf(tasks).join();
executorService.shutdown();
System.out.println("Finish download images");
```

CompletableFuture

- ➡ *boolean isDone() – проверяет записан ли результат задачи в completableFuture*
- ➡ *T get() – блокирует поток и ждет результата выполнения задачи*
- ➡ *T join() – то же что и get(), но бросает CompletionException*
- ➡ *T getNow(T valueIfAbsent) – возвращает значение либо valueIfAbsent если его еще нет*
- ➡ *V get(long, TimeUnit) – get() с timeout*

CompletableFuture запуск задач



static <U> CompletableFuture<U> supplyAsync(Supplier<U>) - Запускается задача с функцией supplier, и результат выполнения записывается во фьючерс



static <U> CompletableFuture<U> supplyAsync(Supplier<U>, ExecutorService) - Запускается задача с функцией supplier, на переданном ExecutorService



static CompletableFuture<Void> runAsync(Runnable)

static CompletableFuture<Void> runAsync(Runnable, ExecutorService) - аналоги supplyAsync, но с Runnable

CompletableFuture обработка результата



CompletableFuture<U> thenApply(Function<? super T, ? extends U> fn) – функция, которая принимает значение из первого future, обрабатывает его и записывает новое значение в результирующий future



CompletableFuture<Void> thenAccept(Consumer<? super T> block) – функция принимает значение из первого future и обрабатывает его без возвращаемого значения



static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs) – Возвращает новый future, который завершится когда хотя-бы один из данных завершается



static CompletableFuture<Object> allOf(CompletableFuture<?>... cfs) – Возвращает новый future, который завершится когда все из данных завершатся

CompletableFuture



Позволяет еще легче писать
ассинхронный код



Есть возможность писать с
использованием ExecutorService



```
ExecutorService executorService = Executors.newFixedThreadPool(5);
var tasks = Stream.generate(
    () -> CompletableFuture.runAsync(ImagesExecutorsExample::downloadImage, executorService))
    .limit(5)
    .toArray(CompletableFuture[]::new);
CompletableFuture.allOf(tasks).join();
executorService.shutdown();
System.out.println("Finish download images");
```

CallbackHell



Происходит, когда много колбеков передаются в лямбы



CompletableFuture делает такой код легче для восприятия

```
1  function hell(win) {
2    // For listener purpose
3    return function() {
4      loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
5        loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
6          loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
7            loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
8              loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {
9                loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
10               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
11                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
12                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
13                     async.eachSeries(SCRIPTS, function(src, callback) {
14                       loadScript(win, BASE_URL+src, callback);
15                     });
16                   });
17                 });
18               });
19             });
20           });
21         });
22       });
23     });
24   );
25 };
26 }
```



CallbackHell



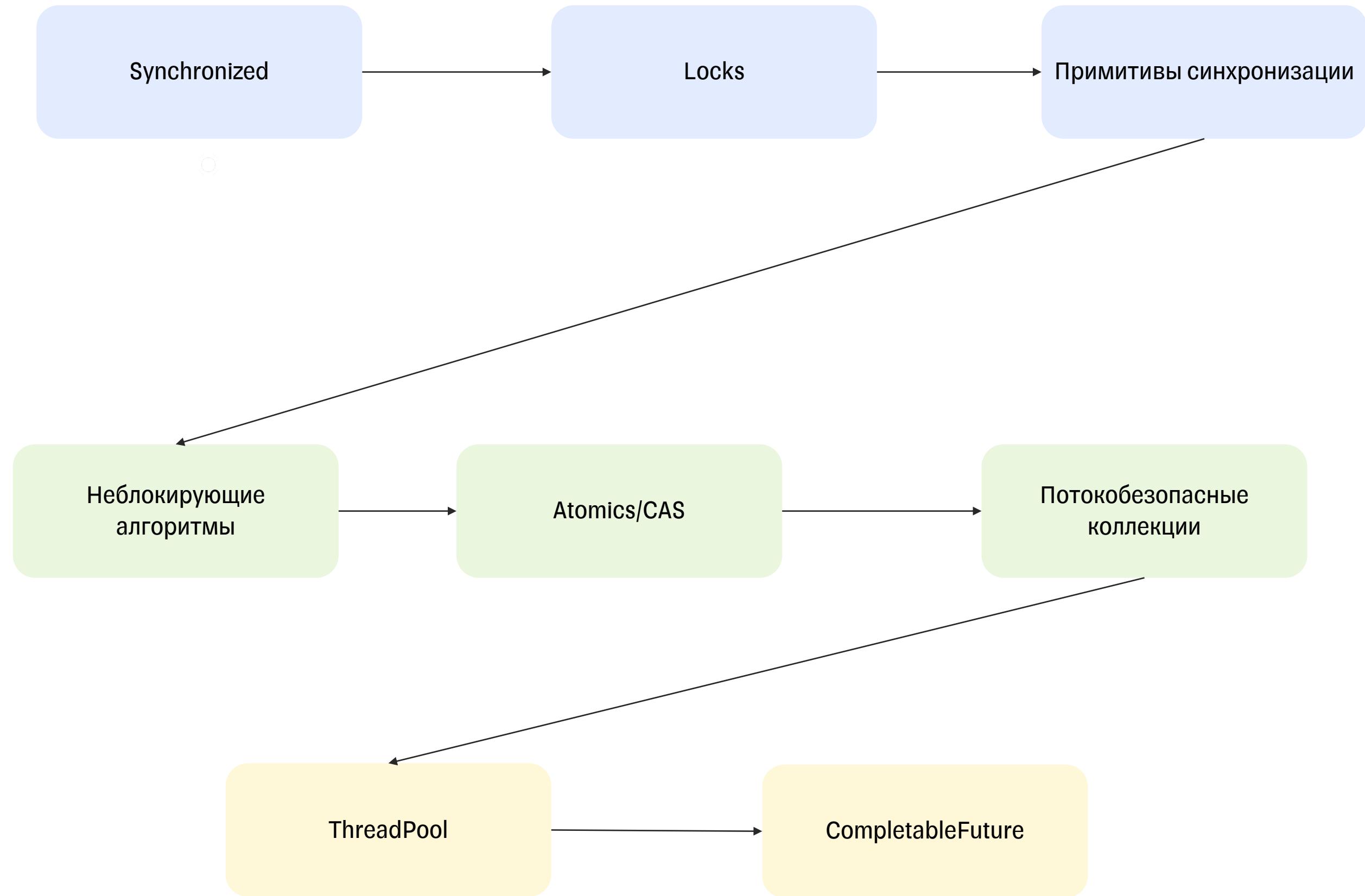
Происходит, когда много колбеков
передаются в лямбы



CompletableFuture делает такой код
легче для восприятия

```
CompletableFuture.runAsync(compute0, executorService)
    .thenApply(compute1)
    .thenApply(compute2)
    .thenApply(compute3)
    .thenAccept(compute4)
    .join();
```

Итоги



Доп. ресурсы

- <https://habr.com/ru/articles/277669> - статья про синхронизаторы
- <https://www.baeldung.com/java-concurrent-locks> - статья про Locks
- <https://habr.com/ru/articles/213319/> - статья про CompletableFuture
- <https://www.baeldung.com/java-completableFuture> - статья про CompletableFuture
- <https://www.baeldung.com/java-util-concurrent> - обзор java.util.concurrent
- <https://www.baeldung.com/java-executor-service-tutorial> - статья про ExecutorService



ТИНЬКОФФ

Он такой один



Обратная связь

