Многопоточное Программирование: Мониторы и ожидание

Роман Елизаров, JetBrains, elizarov@gmail.com
Никита Коваль, JetBrains, ndkoval@ya.ru

ИТМО 2019

ITMO UNIVERSITY

Объекты как функции

- Операция над объектом как функция
  $f(S, P) = (S', R)$
- Ранее операции были **всюду определены** на паре (S, P)
  - Если операцию нельзя выполнить то результат это ошибка или исключение
- В общем случае операции могут быть **частично определены** множестве пар (S, P)
  - Операцию не может завершиться и *ждет*

Пример: очередь ограниченного размера

- Обычные операции
  - **size(): int** – узнать текущий размер
  - **offer(item): boolean** – вернет **false** если нет свободного места
  - **poll(): item?** – вернет **null** если очередь пуста

Очередь ограниченного размера с ожиданием

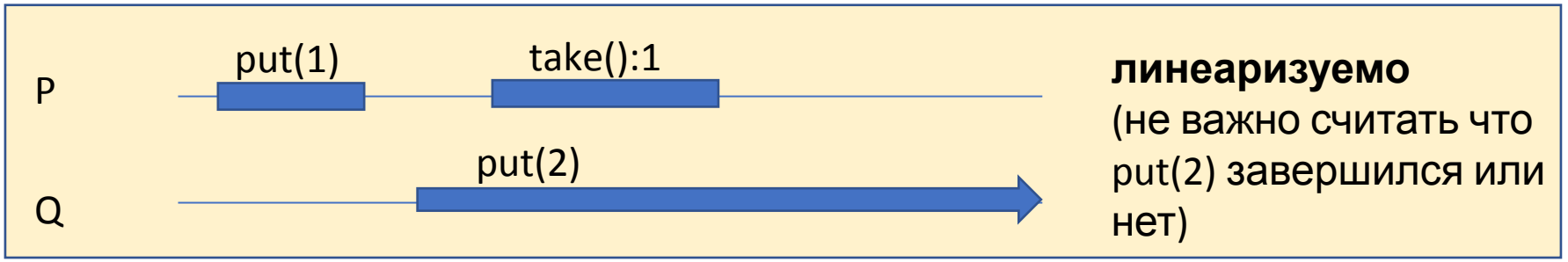
- Операции с ожиданием
  - **put(item)** – положить элемент в очередь, *если есть свободное место* (если же места нет, то операция ждет, в этом состоянии функция операции не определена)
  - **take(): item** – забрать элемент из очереди, *если очередь не пуста* (если пуста – ждет)

Линеаризуемость операций с ожиданием

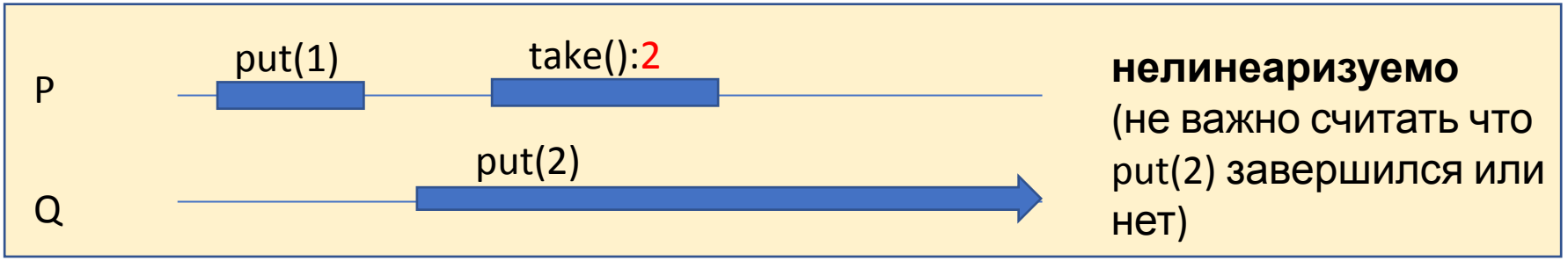
- Расширим понятие исполнения
  - Есть событие вызова inv(A)
  - Но не обязателен ответ res(A)
    - A это **незавершенная операция** если нет res(A)
    - inv(A) это **незавершенный вызов**
- Исполнение **линеаризуемо**
  - Если для незавершенных операций можно
    - Либо добавить ответы
    - Либо выкинуть их из исполнения
  - Чтобы получилось **допустимое последовательное исполнение**

    $inv(A_1) \rightarrow res(A_1) \rightarrow inv(A_2) \rightarrow res(A_2) \ldots$

Примеры для очереди с ожиданием (1)

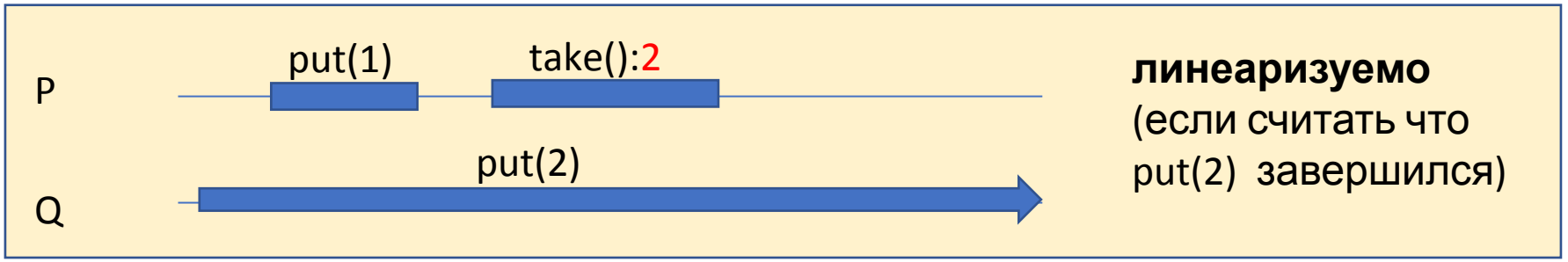


линеаризуемо
(не важно считать что put(2) завершился или нет)



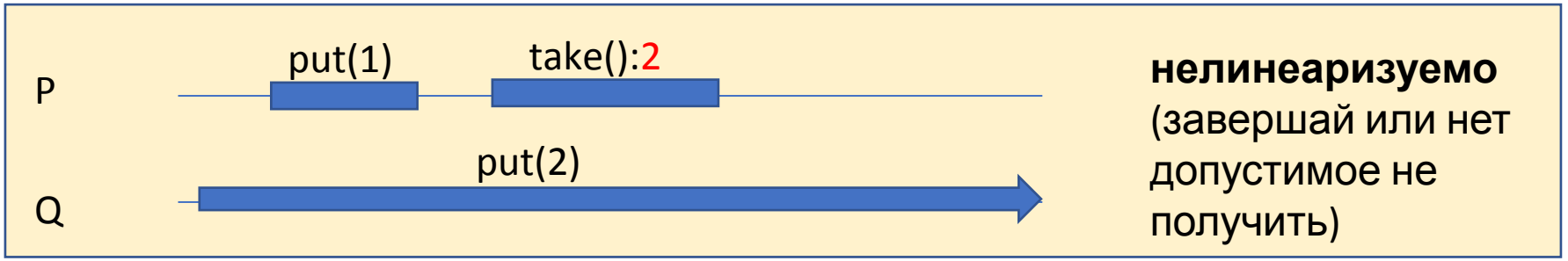
нелинеаризуемо
(не важно считать что put(2) завершился или нет)

Примеры для очереди с ожиданием (2)



линеаризуемо
(если считать что put(2) завершился)

Но если очередь может содержать *только один элемент*, то



нелинеаризуемо
(завершай или нет допустимое не получить)

Реализация через монитор

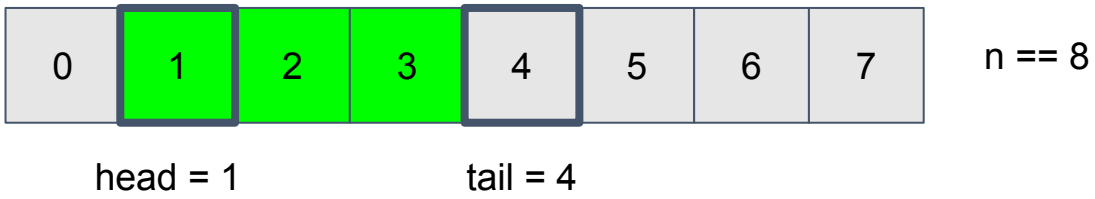
- **Monitor** = **mutex** + **conditional variables**
  - Взаимное исключение для защиты данных от одновременного изменения
  - Условные переменные для ожидания
  - Мониторы придумал Энтони Хоар (1974)
    - Поэтому они так же известны как “*Hoare monitors*”

Монитор в JVM

- В JVM каждый объект имеет монитор с *одной* условной переменной
  - **synchronized**:
    - **monitorenter** (lock)
    - **monitorexit** (unlock)
  - **wait**, **notify**, **notifyAll** – для работы с условной переменной

Пример: структура данных очереди с ожиданием

- Будем писать обычную циклическую очередь на массиве




```
public class BlockingQueue<T> {
   private final T[] items; // элементы
   private final int n;     // == items.length
   private int head;        // голова
   private int tail;        // хвост
}
```

synchronized size

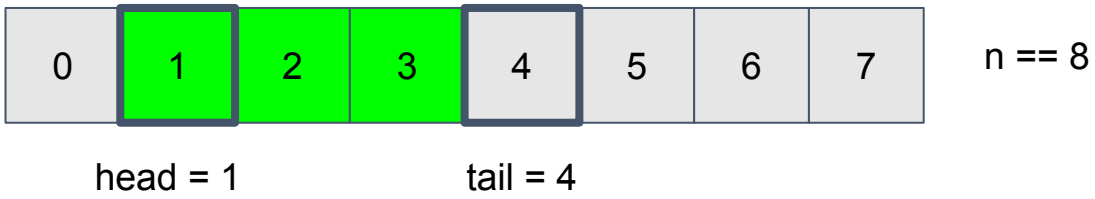
- Используем грубую синхронизацию через встроенный в JVM объекты монитор

```
public class BlockingQueue<T> {
    private final T[] items; // элементы
    private final int n;     // == items.length
    private int head;        // голова
    private int tail;        // хвост

    public synchronized int size() {
        return (tail - head + n) % n;
    }
```

Kotlin: @Synchronized

Не ждущий poll

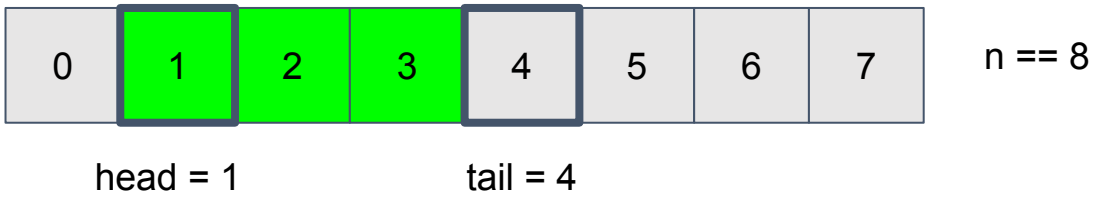
- Полностью определен в любом состоянии
- Если очередь пуста – возвращает **null**




```
public synchronized T poll() {
    if (head == tail) return null;
    T result = items[head];
    items[head] = null;
    head = (head + 1) % n;
    return result;
}
```

Ждущий take

- Не определен для пустой очередь
- Если очередь пуста – ждет

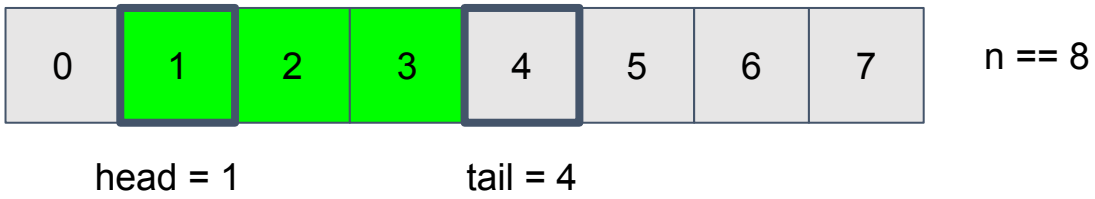



```
public synchronized T take() {
    if (head == tail) wait(); // ждем
    T result = items[head];
    items[head] = null;
    head = (head + 1) % n;
    return result;
}
```

Kotlin:
(this as Object).wait()

Ждущий take

- Кидает **InterruptedException** как признак что ожидание можно прервать через **Thread.interrupt**

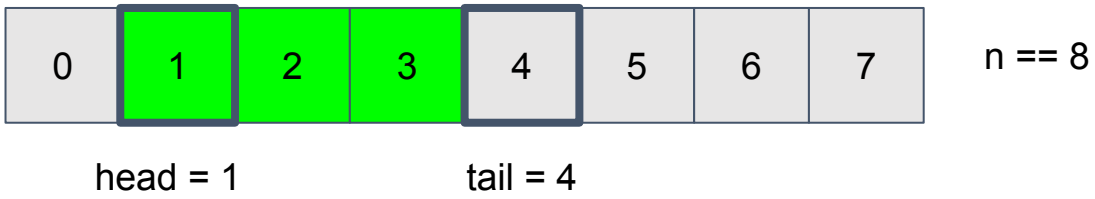



```
public synchronized T take() throws InterruptedException {
    if (head == tail) wait(); // ждем
    T result = items[head];
    items[head] = null;
    head = (head + 1) % n;
    return result;
}
```

Kotlin: не нужно объявлять throws

Spurious wakeup!

- “spurious wakeups are possible, and this method should always be used in a loop”



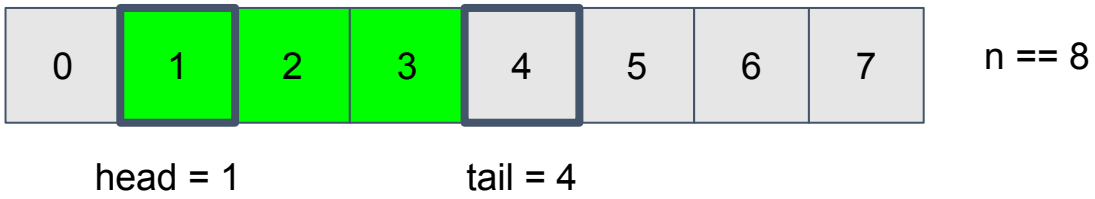

```
public synchronized T take() throws InterruptedException {
    while (head == tail) wait(); // ждем
    T result = items[head];
    items[head] = null;
    head = (head + 1) % n;
    return result;
}
```

Чего ждем?

- Метод **wait** делает такие шаги
  - Выходит из критической секции (из монитора), чтобы другие потоки могли в неё попасть и поменять состояние объекта
  - Дожидается **сигнала** через **условную переменную**
  - Снова входит в критическую секцию (в монитор), чтобы этот поток мог перепроверить состояние объекта и выполнить свою операцию если состояние подходящие
- Сигнал посылается через **notify** (сигнал одном ждущему потоку) и **notifyAll** (сигнал всем ждущим потокам)
  - В случае сомнений всегда используйте **notifyAll**

Не ждущий offer

- Сам метод не ждет
- Но будит потоки, которые могут ждать пока очередь станет *не пуста*




```
public synchronized boolean offer(T item) {
    int next = (tail + 1) % n;
    if (next == head) return false;
    items[tail] = item;
    tail = next;
    notifyAll();
    return true;
}
```

Kotlin:
(this as Object).notifyAll()

Не ждущий offer – лучше

- Нужно будить другие потоки только если очень действительно *становится* не пуста
- **Не важно где в коде стоит notify**

```
public synchronized boolean offer(T item) {
    int next = (tail + 1) % n;
    if (next == head) return false;
    items[tail] = item;
    if (head == tail) notifyAll();
    tail = next;
    return true;
}
```

Ждущий put

- Ждет пока очередь станет *не полна*
- И будит потоки, которые могут ждать пока очередь станет *не пуста*

```
public synchronized void put(T item) throws Inter…Ex… {
    while (true) { // пока не подходящее состояние
        int next = (tail + 1) % n;
        if (next == head) { wait(); continue; }
        items[tail] = item;
        if (head == tail) notifyAll();
        tail = next;
        return;
    }
}
```

Ждущий put

- Но кто разбудит его?
- Нужно менять **poll** & **take**!

```
public synchronized void put(T item) throws IE {
    while (true) { // пока не подходящее состояние
        int next = (tail + 1) % n;
        if (next == head) { wait(); continue; }
        items[tail] = item;
        if (head == tail) notifyAll();
        tail = next;
        return;
    }
}
```

Доводим до ума poll

```
public synchronized T poll() {
    if (head == tail) return null;
    T result = items[head];
    items[head] = null;
    if ((tail + 1) % n == head) notifyAll(); // была полна
    head = (head + 1) % n;
    return result;
}
```

Доводим до ума take

```
public synchronized T take() throws InterruptedException {
    while (head == tail) wait(); // ждем
    T result = items[head];
    items[head] = null;
    if ((tail + 1) % n == head) notifyAll(); // была полна
    head = (head + 1) % n;
    return result;
}
```

Снова notify vs notifyAll

- Если бы для каждого условия была своя отдельная переменная, то можно использовать **notify**
  - Для *данного* алгоритма поток отработавший по условию так меняет состояние объекта, что условие больше не верно
- Но у Java объекта есть только одна условная переменная на каждый монитор
  - Поэтому при многих ждущих потоках будет очень неэффективно

j.u.c.ReentrantLock спасет

- В пакете **java.util.concurrent** есть интерфейс **Lock**
  - С методами **lock**, **unlock**, **newCondition**
  - И **ReentrantLock** – реализация его
  - Интерфейс **Condition** для условных переменных с методами **await**, **signal**, **signalAll**
  - Можем завести отдельную переменную для каждого условия

```
private final Lock lock = new ReentrantLock();
private final Condition notEmpty = lock.newCondition();
private final Condition notFull = lock.newCondition();
```

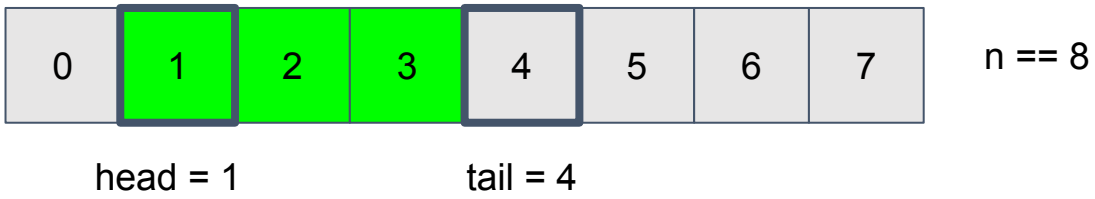
Let’s Kotlin

- Будет удобней работать с локами

```
class BlockingQueue<T>(private val n: Int) {
    private val items = arrayOfNulls<Any>(n)
    private var head = 0
    private var tail = 0

    private val lock = ReentrantLock()
    private val notEmpty = lock.newCondition()
    private val notFull = lock.newCondition()
}
```

Эффективный take

- Теперь можем использовать **signal**
- Но ждать обязаны все-равно в цикле




```
fun take(): T = lock.withLock {
    while (head == tail) notEmpty.await() // ждем
    val result = items[head] as T
    items[head] = null
    if ((tail + 1) % n == head) notFull.signal() // была полна
    head = (head + 1) % n
    result // вернули из withLock
}
```

Эффективный take

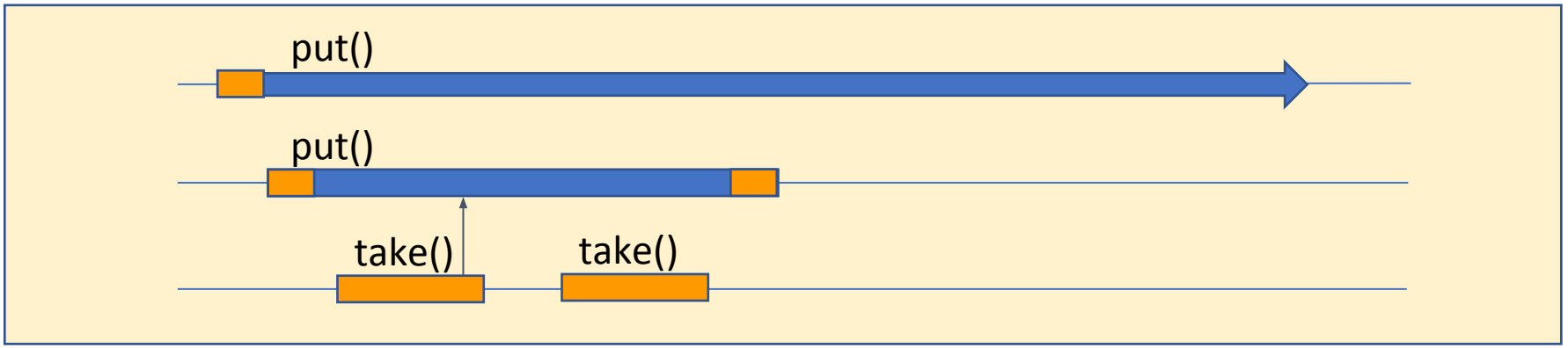
- Не можем использовать **signal** (!!!)

```
fun take(): T = lock.withLock {
    while (head == tail) notEmpty.await() // ждем
    val result = items[head] as T
    items[head] = null
    if ((tail + 1) % n == head) notFull.signal() // была полна
    head = (head + 1) % n
    result // вернули из withLock
}
```

Нельзя так просто взять и использовать signal

Проблематичный сценарий

- Рассмотрим такую последовательность операций
  1. Два **put** ждут **notFull**
  2. Пришел **take** – сделал очередь не полной, послал сигнал **notFull**
  3. Но **put** не успел проснуться и взять **lock** по сигналу
  4. Пришел еще **take**. Очередь уже не полная, так что он не посылает еще один сигнал **notFull**



Проблематичный сценарий

- Итого: один **put** проснулся, а другой продолжает спать, хотя очередь не полна (**notFull**)
- Вывод: в отличие от **notifyAll/signalAll** оптимизировать вызовы **notify/singnal** нужно осторожно (в данном случае – нельзя)

Правильный take

- Надо посылать один сигнал на каждый элемент, который был удален из очереди, чтобы каждый раз будить один ждущий поток

```
fun take(): T = lock.withLock {
    while (head == tail) notEmpty.await() // ждем
    val result = items[head] as T
    items[head] = null
    notFull.signal() // всегда шлем сигнал
    head = (head + 1) % n
    result // вернули из withLock
}
```

Аналогичный put

- Симметрично – ждем **notFull**, сигнализируем **notEmpty**

```
fun put(item: T): Unit = lock.withLock {
    while (true) { // пока не подходящее состояние
        val next = (tail + 1) % n
        if (next == head) { notFull.await(); continue }
        items[tail] = item
        notEmpty.signal() // надо посылать один сигнал
        tail = next
        return@withLock
    }
}
```

Подробней про interrupt

- У каждого потока есть *interrupted* флаг
  - Его ставит метод **Thread.interrupt**
  - Его проверяют методы **wait**, **await** и т.п. методы
  - В случае обнаружения выставленного эти методы
    - Прекращают ждать
    - Сбрасывают флаг
    - Кидают **InterruptedException**

Подробней про interrupt

- Это **кооперативный** способ прерывания заблокированных потоков
  - Не нужно знать что именно ждет поток
  - Но не надейтесь что это будет работать, если вы используете сторонние библиотеки (никто не умеет программировать!)

```
// где-то в глубине чужого Java кода
try {
    wait();
} catch (InterruptedException e) { /* ignore */ }
```

Что делать с ненужным InterruptedException?

- Если нужно реализовать метод который ждет, но не кидает **InterruptedException**, то interrupted флаг надо перевыставить

```
public T takeOrNull() {
    try {
        return take();
    } catch (InterruptedException e) {
        // перевыставим флаг interrupted
        Thread.currentThread().interrupt();
        return null;
    }
}
```

Пишем поток обрабатывающий очередь

- Заводим свой флаг, сигнализирующий что поток надо остановить
  - В отличие от флага interrupted, нет риска что какой-то сторонний метод его случайно сбросит

```
public class DoSomethingThread<T> extends Thread {
   private final BlockingQueue<T> queue; // задачи
   private volatile boolean closed; // флаг останова

   public void close() {
        closed = true; // ставим флаг останова (сначала!)
        interrupt(); // чтобы прервать ожидания
   }
}
```

Главный метод потока

- Метод **run** выходит в случае прерывания
  - Флаг interrupted перевыставлять не надо

```
@Override
public void run() {
    try {
        while (!closed) {
            T item = queue.take();
            doSomething(item);
        }
    } catch (InterruptedException e) {
        // а вот здесь можем проигнорировать -- уже выходим
    }
}
```

Выводы

- Для ожидания определенного состояния структуры данных нужен Hoare’s Monitor
- Для эффективного ожидания нужен Condition Variable на каждое условие которого ждем
- Получается весьма просто, по шаблону
- Но это блокирующий алгоритм (со всеми вытекающими)

Ожидание без блокировки

Пример: обновляемое значение

- Почти очередь на один элемент
- Операции
  - **update(item)** – обновить текущее значение
  - **remove(): item?** – забрать и сбросить текущее значение

Реализация с блокировкой

```
class DataHolder<T> {
    private var value: T? = null
    private val lock = ReentrantLock()

    fun update(item: T) = lock.withLock {
        value = item
    }

    fun remove(): T? = lock.withLock {
        value.also { value = null }
    }
}
```

Ждать новое значение

- **take(): item** – ждет пока значение появится

```
private val updated = lock.newCondition()

fun take(): T = lock.withLock {
    while (value == null) updated.await()
    value!!.also { value = null }
}

fun update(item: T) = lock.withLock {
    value = item
    updated.signal()
}
```

Реализация без блокировки

```
class DataHolder<T> {
    private val v = atomic<T?>(null)

    fun update(item: T) {
        v.value = item // volatile write
    }

    fun remove(): T? {
        v.loop { cur ->
            if (cur == null) return null
            if (v.compareAndSet(cur, null)) return cur
        }
    }
}
```

Ожидание без блокировки (park)

```
class TakerThread<T> : Thread() {
    // ...

    fun take(): T {
        assert(Thread.currentThread() == this)
        v.loop { cur ->
            if (cur == null) {
                LockSupport.park()
                if (interrupted())
                    throw InterruptedException()
                return@loop // continue loop
            }
            if (v.compareAndSet(cur, null)) return cur
        }
    }
}
```

Идиома прерывания

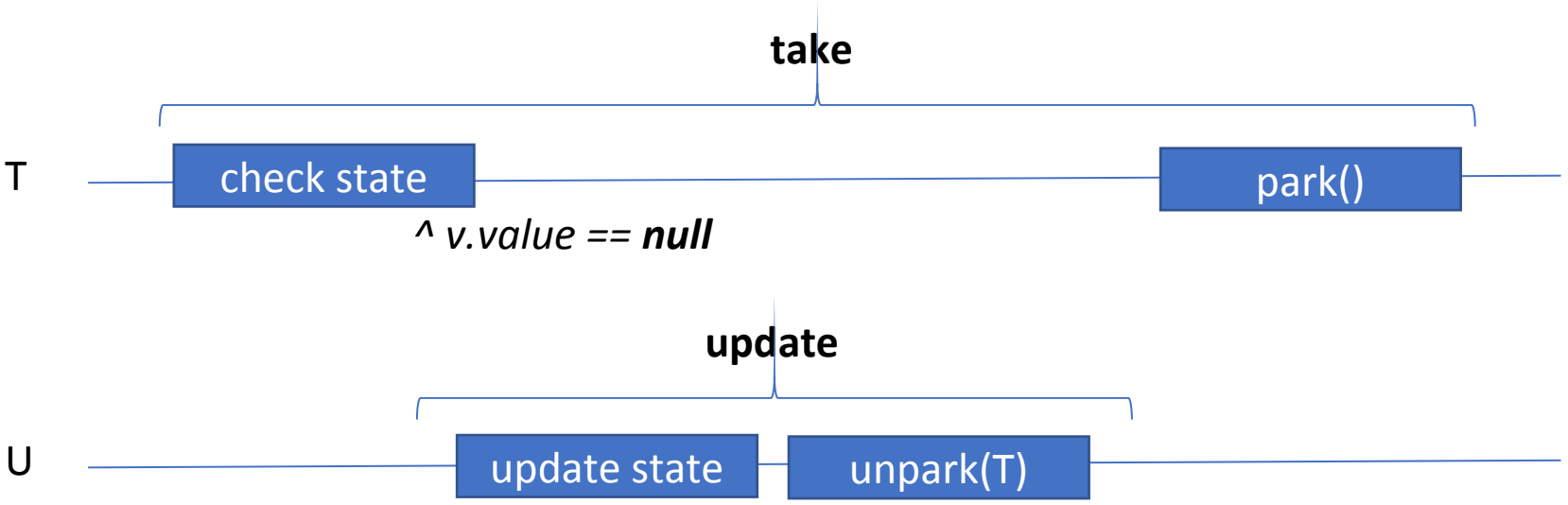
```
class TakerThread<T> : Thread() {
    // ...

    fun take(): T {
        assert(Thread.currentThread() == this)
        v.loop { cur ->
            if (cur == null) {
                LockSupport.park()
                if (interrupted())
                    throw InterruptedException()
                return@loop // continue loop
            }
            if (v.compareAndSet(cur, null)) return cur
        }
    }
}
```

Разбудить (unpark)

- Здесь важен порядок: обновить → unpark

```
fun update(item: T) {
    v.value = item // volatile write
    LockSupport.unpark(this)
}
```

Магия park/unpark

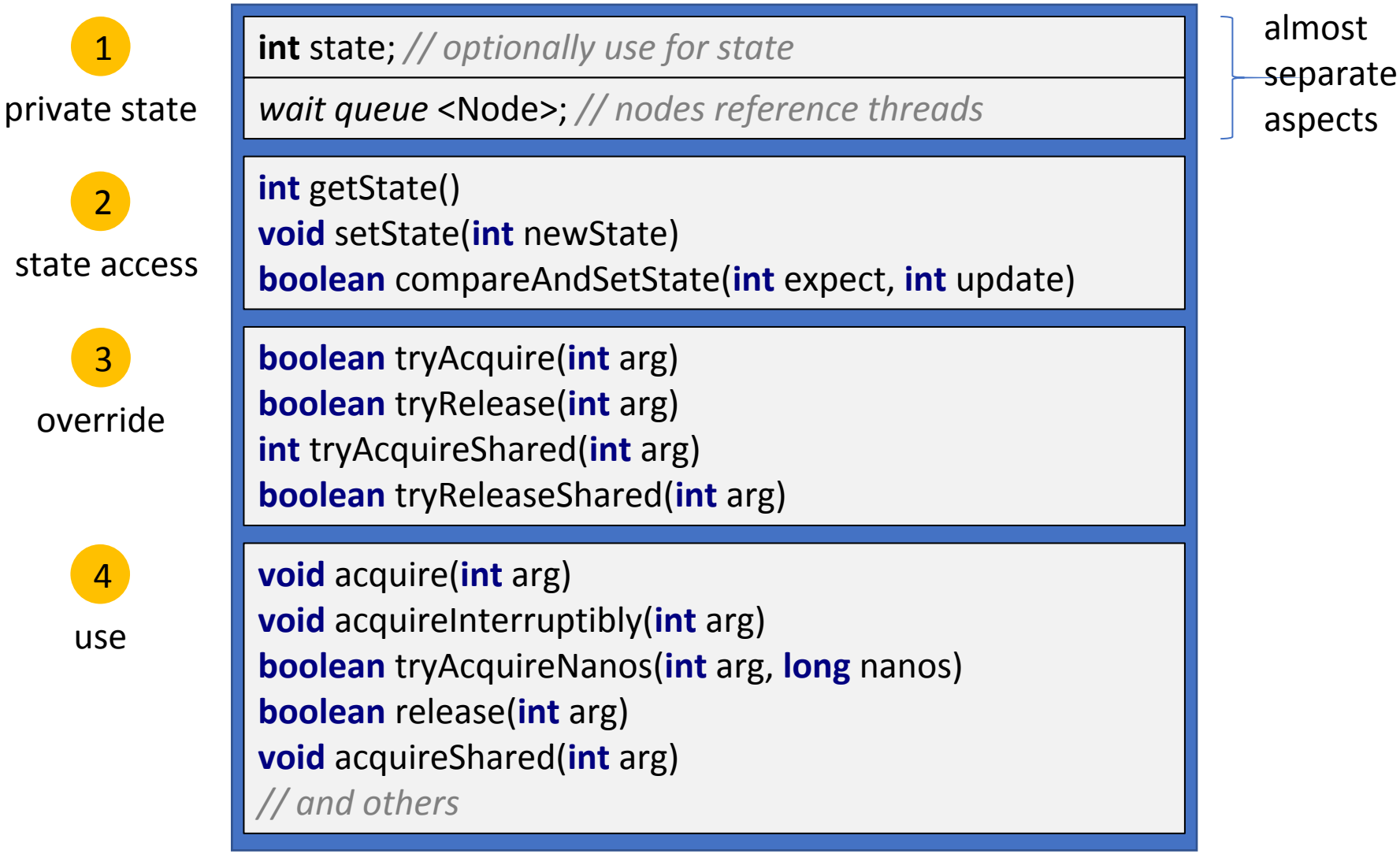


LockSupport.unpark(T): “Makes available the permit for the given thread, if it was not already available. If the thread was blocked on park then it will unblock. *Otherwise, its next call to park is guaranteed not to block.*”

Ожидание из многих потоков

- Нужна очередь ждущих потоков
  - Нетривиально написать
- Взять готовый!
- **j.u.c.l.AbstractQueuedSynchronizer**
  - **ReentrantLock**
  - **ReentrantReadWriteLock**
  - **Semaphore**
  - **CountDownLatch**

Анатомия AbstractQueuedSynchronizer



Анатомия AbstractQueuedSynchronizer (2)

```
public final void acquireInterruptibly(int arg)
        throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    if (!tryAcquire(arg))
        doAcquireInterruptibly(arg);
}

public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

1

2 adds to wait queue

3

4 unlinks from wait queue

Пишем свой внутренний синхронайзер

```
inner class Sync : AbstractQueuedSynchronizer() {
    override fun tryAcquire(arg: Int): Boolean {
        val cur = v.value ?: return false
        if (!v.compareAndSet(cur, null)) return false
        // надо как-то вернуть значение отсюда
        results[arg] = cur
        return true
    }

    // всегда "освобождаем" -- будим следующего
    override fun tryRelease(arg: Int): Boolean = true
}
```

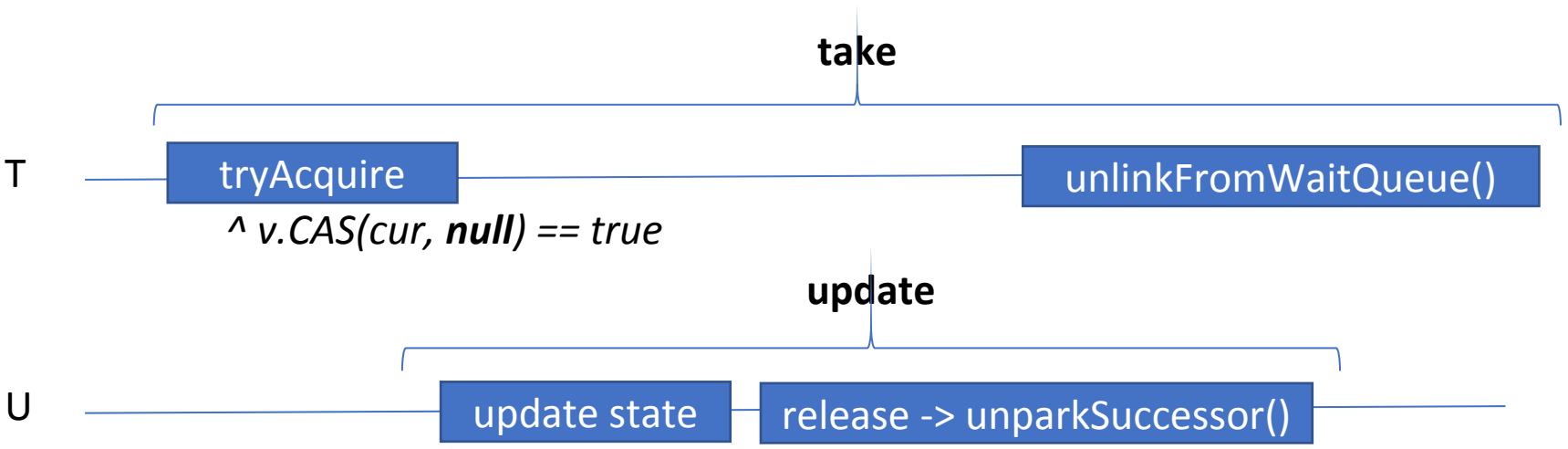
Используем его

```
private val sync = Sync()

fun update(item: T) {
    v.value = item // volatile write
    sync.release(0) // шлем сигнал
}

fun take(): T {
    val arg = reserveResultsSlot() // приходится крутиться
    sync.acquireInterruptibly(arg) // ждет внутри
    // нужна перепроверка чтобы не потерять unpark
    if (v.value != null) sync.release(0)
    return releaseResultsSlot(arg)
}
```

Зачем двойная проверка?

Упрощенно!

```
void doAcquireXXX(int arg) {
    addToWaitQueue();
    for (;;) {
        if (isFirstInQueue() && tryAcquire(arg)) {
            unlinkFromWaitQueue(); return;
        }
        doPark();
    }
}
```

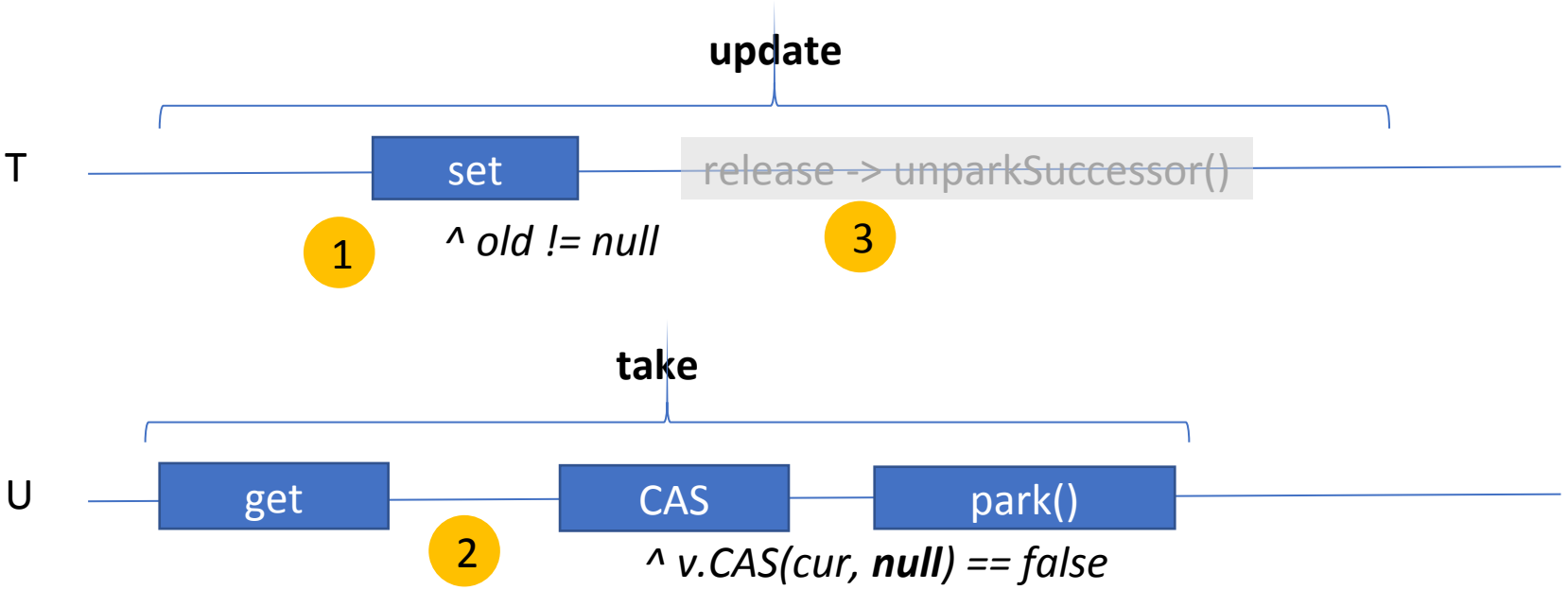


“Улучшим” производительность?

- Идея: будить поток только когда обновляем **null** → значение
- Не работает очень тонким образом...

```
fun update(item: T) {
    val old = v.getAndSet(item)
    if (old == null) sync.release(0) // шлем сигнал
}
```

Fail



Из-за **update** параллельный **tryAcquire** может обломаться на CAS и запарковаться, но так как мы не вызываем больше **release** то никто его больше никогда не разбудит

Исправляем tryAcquire

- Используем цикл и повторяем если CAS упал

```
override fun tryAcquire(arg: Int): Boolean {
    while (true) {
        val cur = v.value ?: return false
        if (!v.compareAndSet(cur, null)) continue
        // надо как-то вернуть значение отсюда
        results[arg] = cur
        return true
    }
}
```

Выводы

- Ждать можно без блокировок через park/unpark
- Как обычно, без блокировок надо аккуратно думать, легко ошибиться
- AbstractQueuedSynchronizer вообще-то предназначен для написания блокировок