Многопоточное Программирование: Сложные блокировки

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

ИТМО 2019

ITMO UNIVERSITY

Сложные блокировки и анализ конфликтов

Анализ конфликтов

- Гонка (конфликт) данных (data race): два [несинхронизированных] доступа к одной ячейке данных, один из которых запись.
- *Матрица конфликтов* (X – конфликт):

	R	**W**
R		X
W	X	X

Пример: стек на массиве (однопоточный)

```
public class ArrayStack<T> {
    int top;
    T[] data;
    // конструктор выделяет массив data
    public int size() {
        return top;
    }

    public void push(T item) {
        data[top++] = item;
    }

    public T pop() {
        return data[--top];
    }
}
```

Анализ конфликтов стека

	size	**push**	**pop**
size		X	X
push	X	X	X
pop	X	X	X

Пример: стек с грубой синхронизацией

```
public class ArrayStack<T> {
    private int top;
    private T[] data;
    // конструктор выделяет массив data
    public synchronized int size() {
        return top;
    }

    public synchronized void push(T item) {
        data[top++] = item;
    }

    public synchronized T pop() {
        return data[--top];
    }
}
```

Теперь нет гонок!

Пример: стек с грубой синхронизацией 2

Kotlin

```
import java.util.concurrent.locks.*

// В классе ArrayStack добавляем
private val lock = ReentrantLock()

val size: Int get() = lock.withLock {
    top
}

fun push(item: T) = lock.withLock {
    data[top++] = item
}

fun pop(): T = lock.withLock {
    data[--top]
}
```

Матрица совместимости блокировок с грубой синхронизацией

	size	**push**	**pop**
size	X лишнее	X	X
push	X	X	X
pop	X	X	X

Read-write locks (блокировка чтения-записи)

- Это специальные блокировки, со следующей *матрицей совместимости (*X – несовместимые блокировки)
  - Read aka *Shared* Lock
  - Write aka *Exclusive* Lock
- Они идеально подходят для грубой защиты структур данных в которых есть конфликты (гонки) по данным.

	R	**W**
R		X
W	X	X

Пример: стек с грубой синхронизацией 3

```
// В классе ArrayStack меняем
private val lock = ReentrantReadWriteLock()
```

Пример: стек с грубой синхронизацией 3

```
// В классе ArrayStack меняем
private val lock = ReentrantReadWriteLock()

val size: Int get() = lock.readLock().withLock {
    top
}
```

Пример: стек с грубой синхронизацией 3

```
// В классе ArrayStack меняем
private val lock = ReentrantReadWriteLock()

val size: Int get() = lock.readLock().withLock {
    top
}

fun push(item: T) = lock.writeLock().withLock {
    data[top++] = item
}
```

Пример: стек с грубой синхронизацией 3

```
// В классе ArrayStack меняем
private val lock = ReentrantReadWriteLock()

val size: Int get() = lock.readLock().withLock {
    top
}

fun push(item: T) = lock.writeLock().withLock {
    data[top++] = item
}

fun pop(): T = lock.????????().withLock {
    data[--top]
}
```

Пример: стек с грубой синхронизацией 3

```
// В классе ArrayStack меняем
private val lock = ReentrantReadWriteLock()

val size: Int get() = lock.readLock().withLock {
    top
}

fun push(item: T) = lock.writeLock().withLock {
    data[top++] = item
}

fun pop(): T = lock.writeLock().withLock {
    data[--top]
}
```

Матрица совместимости с блокировками чтения-записи с грубой синхронизацией

	size	**push**	**pop**
size		X	X
push	X	X	X
pop	X	X	X

Полностью повторяет матрицу конфликтов. Лучше сделать нельзя.

Read->Write Upgrade

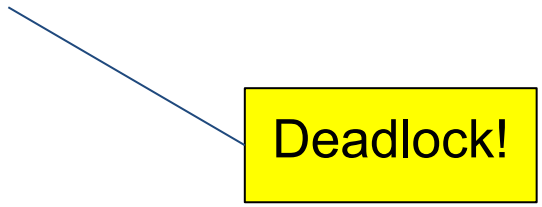
- Где подвох?

```
val lock = ReentrantReadWriteLock()

lock.readLock().withLock {
    println("reading")
    ...
}
```

Read->Write Upgrade

- Где подвох?

```
val lock = ReentrantReadWriteLock()

lock.readLock().withLock {
    println("reading")
    lock.writeLock().withLock {
        println("writing")
    }
}
```



Read->Write Upgrade

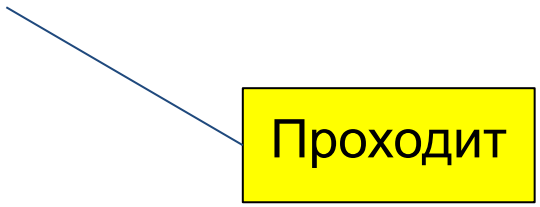
Матрица совместимости

	R	**W**
R		X
W	X	X

UpgradableReadWriteLock

- Ну можем написать….

```
val lock = UpgradableReadWriteLock()

lock.readLock().withLock {
    println("reading")
    lock.writeLock().withLock {
        println("writing")
    }
}
```

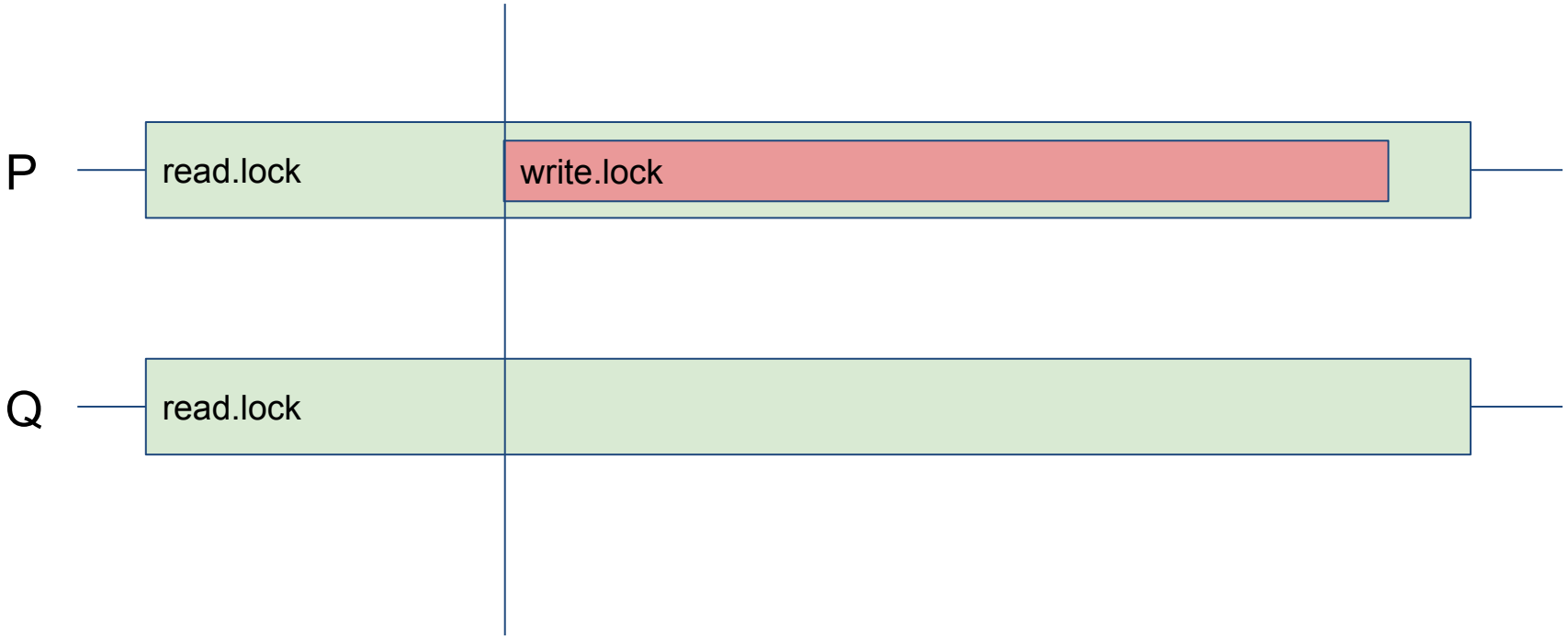


UpgradableReadWriteLock

Матрица совместимости

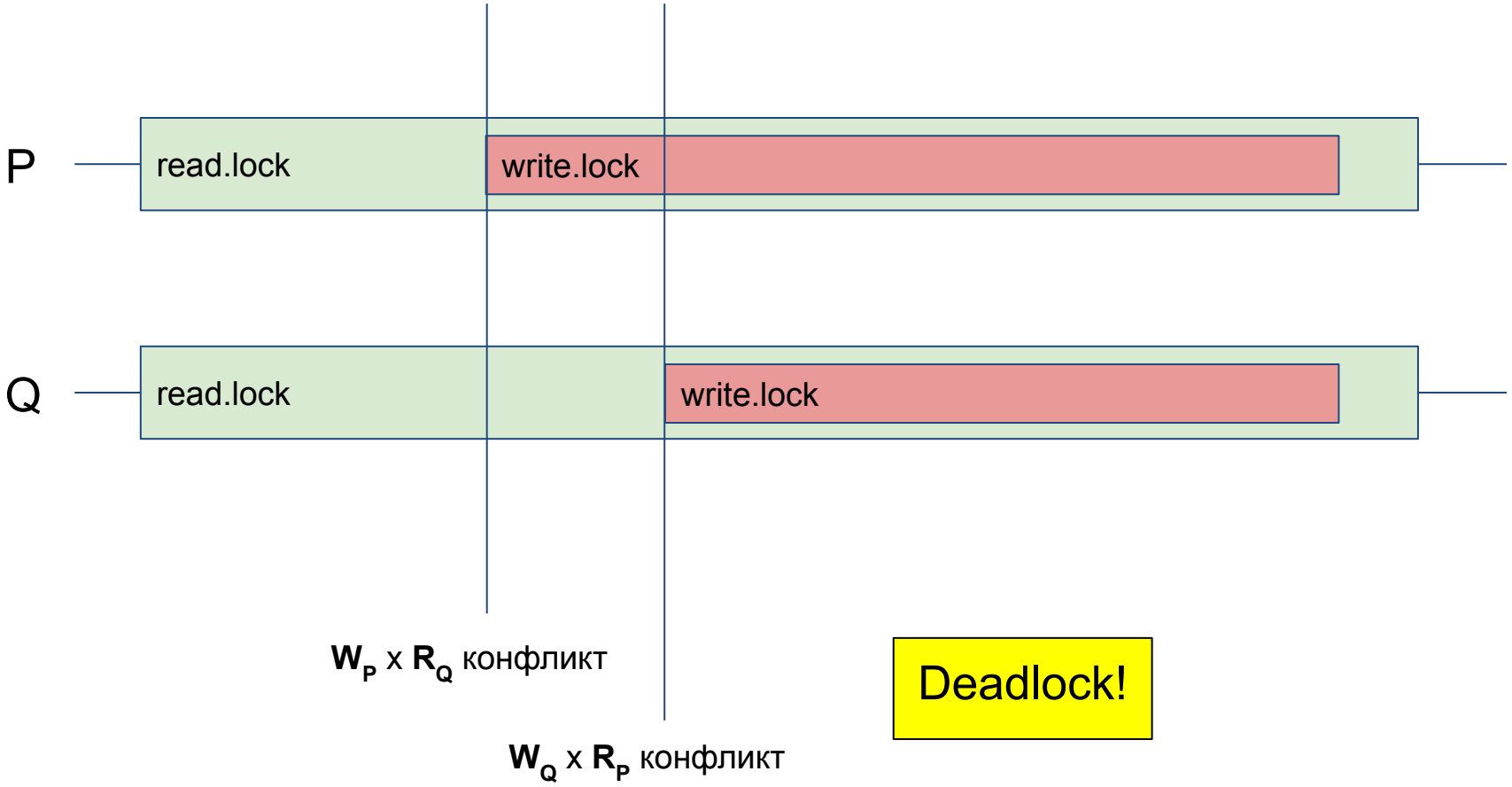
	R_P	R_Q	W_P	W_Q
R_P				
R_Q			X	
W_P		X		X
W_Q			X	

Upgrade на два потока



W_P x R_Q конфликт

Upgrade на два потока



Read-Write locks

- Позволяют увеличить параллелизм в mostly-read случаях
- Надежно работают только при “инкапсулированном” применении
- Если *несколько потоков могут делать upgrade*, то есть опасность взаимной блокировки (**deadlock**)
  - Хм… Есть идея

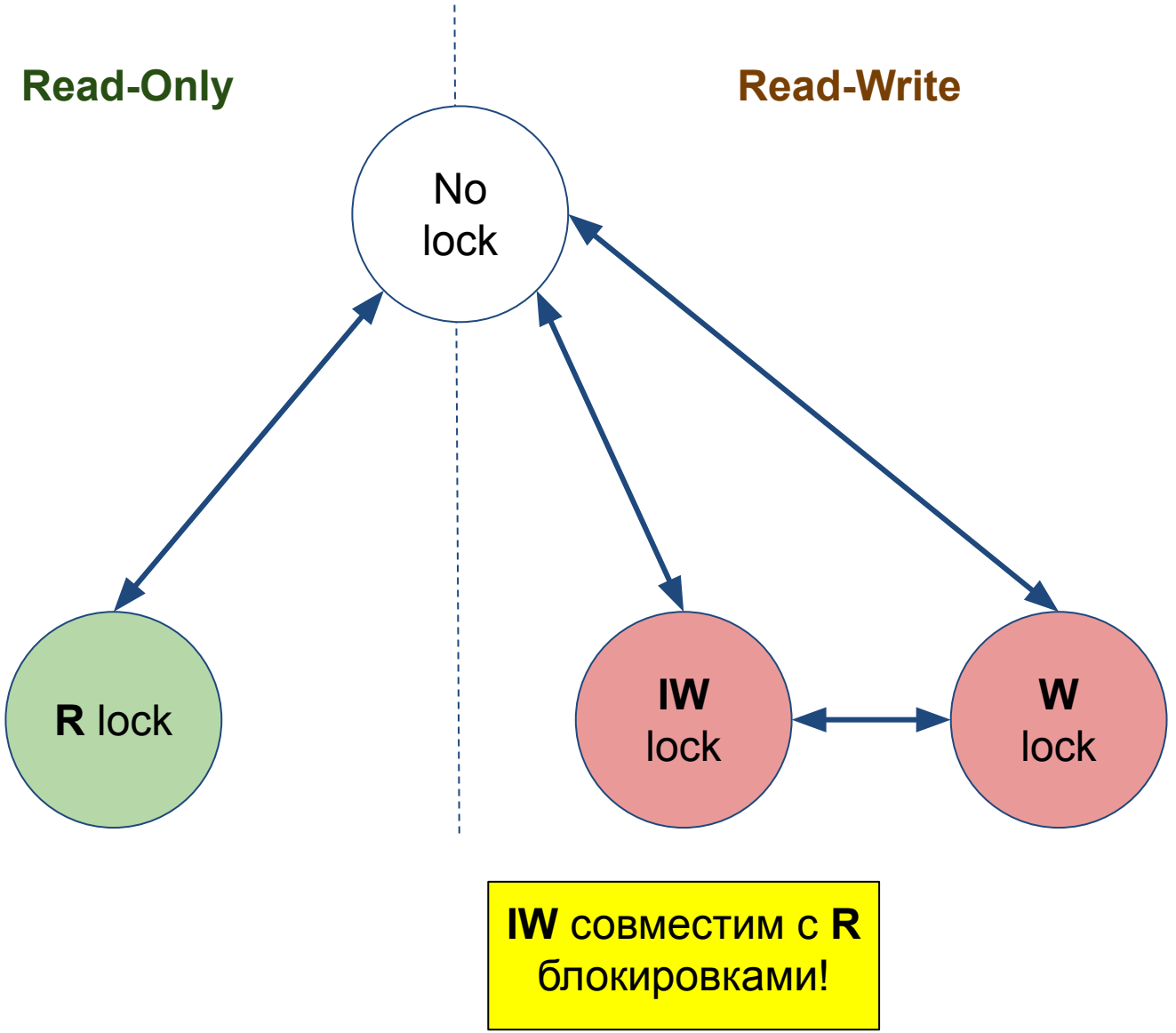
Решение: режим “Intent to Write”

Матрица совместимости

R хоторый хочет сделать *upgrade* на **W**: *разрешаем не больше одного такого*

	R	**IW**	**W**
R			X
IW		X	X
W	X	X	X

Два разных типа операций



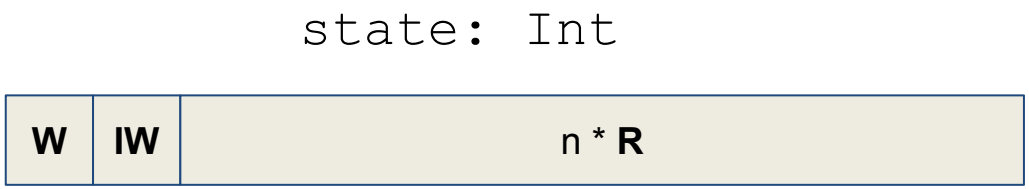
IW Блокировки

- “Решает” проблему upgrade deadlock

- Но надо *заранее* знать тип операции
  - Read-Only
  - Read-Write

Как это всё реализовать?

Состояние блокировки



AbstractQueuedSynchronized

```
private const val R_BIT = 1
private const val R_MASK = (1 shl 30) - 1
private const val IW_BIT = 1 shl 30
private const val W_BIT = 1 shl 31


class Sync : AbstractQueuedSynchronizer() {
    ...
}
```

Немного AQS терминологии

- Shared lock -- могут быть другие
  - **R**
  - **IW**

- Exclusive lock -- не может быть других
  - **W**

Sync: R and IW locks (shared)

```
override fun tryAcquireShared(arg: Int): Int {
    while (true) {
        val state = this.state
        if (state and W_BIT != 0)
            return -1
        if (arg == IW_BIT && state and IW_BIT != 0)
            return -1
        val update = state + arg
        if (compareAndSetState(state, update))
            return 1
    }
}
```

Sync: R and IW locks (shared)

```
override fun tryReleaseShared(arg: Int): Boolean {
    while (true) {
        val state = this.state
        val update = state - arg
        if (compareAndSetState(state, update))
            return update and (R_MASK or IW_BIT) == 0
    }
}
```

Sync: W lock (exclusive)

```
override fun tryAcquire(arg: Int): Boolean {
    while (true) {
        val state = this.state
        if (state != 0)
            return false
        if (compareAndSetState(state, state or W_BIT))
            return true
    }
}
```

Sync: W lock (exclusive)

```
override fun tryRelease(arg: Int): Boolean {
    while (true) {
        val state = this.state
        val update = state and W_BIT.inv()
        if (compareAndSetState(state, update))
            return true
    }
}
```

Read/IW/Write Lock

```
class RIWLock {
   private val sync = Sync()

   val readLock = RLock(sync)
   val IWriteLock = IWLock(sync)
   val writeLock = WLock(sync)

   ...

}
```

RLock

```
private class RLock(val sync: Sync) : Lock {

    override fun lock() =
        sync.acquireShared(R_BIT)
    override fun lockInterruptibly() =
        sync.acquireSharedInterruptibly(R_BIT)
    override fun tryLock(): Boolean =
        sync.tryAcquireShared(R_BIT) > 0
    override fun unlock() {
        sync.releaseShared(R_BIT)
    }

    ...
}
```

IWLock

```
private class IWLock(val sync: Sync) : Lock {

    override fun lock() =
        sync.acquireShared(IW_BIT)
    override fun lockInterruptibly() =
        sync.acquireSharedInterruptibly(IW_BIT)
    override fun tryLock(): Boolean =
        sync.tryAcquireShared(IW_BIT) > 0
    override fun unlock() {
        sync.releaseShared(IW_BIT)
    }

    ...
}
```

lWLock

```
private class WLock(val sync: Sync) : Lock {

    override fun lock() =
        sync.acquire(W_BIT)
    override fun lockInterruptibly() =
        sync.acquireInterruptibly(W_BIT)
    override fun tryLock(): Boolean =
        sync.tryAcquire(W_BIT)
    override fun unlock() {
        sync.release(W_BIT)
    }

    ...
}
```

ReentrantXxxLock

- Что если лок просит тот же поток?

ReentrantXxxLock

- Что если лок просит тот же поток?
- Дополнительно считаем в **ThreadLocal** состоянии

```
class ThreadState {
    var rCount = 0
    var iwCount = 0
    var wCount = 0
}
```