

# Многопоточное Программирование: Транзакционная память

Роман Елизаров, JetBrains, [elizarov@gmail.com](mailto:elizarov@gmail.com)

Никита Коваль, JetBrains, [ndkoval@ya.ru](mailto:ndkoval@ya.ru)

ИТМО 2019



ITMO UNIVERSITY

# Композиция многопоточных структур

```
class Business {  
    val working = ThreadSafeSet<Employee>()  
    val vacating = ThreadSafeSet<Employee>()  
  
    operator fun contains(e: Employee) =  
        e in working || e in vacating  
  
    fun startVacation(e: Employee) {  
        working -= e  
        vacating += e  
    }  
}
```

# Композиция многопоточных структур

- Блокировки
  - Грубые блокировки -
    - очень мало параллелизма (здравствуй Амдал!)
  - Тонкие блокировки -
    - необходимость открыть протокол блокировки (инкапсуляция)
    - взаимные блокировки (как обеспечить глобальную иерархию блокировок?)
  - Если есть ожидание / мониторы - ситуация еще хуже
- Алгоритмы без блокировок
  - Эффективный алгоритм для нетривиальной структуры данных - научный результат
  - CASN и универсальная конструкция -- не панацеи

# Транзакционный манифест

```
class Business {  
    val working = ThreadSafeSet<Employee>()  
    val vacating = ThreadSafeSet<Employee>()  
  
    operator fun contains(e: Employee) =  
        atomic {  
            e in working || e in vacating  
        }  
  
    fun startVacation(e: Employee) {  
        atomic {  
            working -= e  
            vacating += e  
        }  
    }  
}
```

# Что такое транзакция?

- Классические транзакции - **ACID** свойства
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - ~~**D**urability~~

# Почему транзакции?

- Не нужно думать о “порядке блокировок”
    - Просто пиши **atomic**
  - Не нужно думать о “тонкой или толстой блокировке”
    - Просто пиши **atomic**
  - Не нужно изобретать структуры данных без блокировки
    - Просто пиши **atomic**
- 
- Появляется **composability** многопоточных абстракций
  - Корректные программы намного проще писать

# Software Transactional Memory

# Software Transactional Memory (STM)

Задачу можно решить чисто программным способом

```
var x: T = initial
```



# Software Transactional Memory (STM)

Транзакционные переменные

~~**var** x: T = initial~~

**val** x = TVar<T>(initial)

# Software Transactional Memory (STM)

## Транзакционные переменные

```
var x: T = initial
```

```
val x = TVar<T>(initial)
```

```
class TVar<T>(initial: T) {  
    fun readIn(tx: Transaction): T  
    fun writeIn(tx: Transaction, x: T)  
}
```

# Software Transactional Memory (STM)

## Транзакции

```
class Transaction {  
  
    fun <T> TVar<T>.read(): T =  
        readIn(this@Transaction)  
  
    fun <T> TVar<T>.write(x: T) =  
        writeIn(this@Transaction, x)  
  
    ...  
}
```

# Software Transactional Memory (STM)

Можем запрограммировать блок **atomic**

```
fun <T> atomic(block: Transaction.() -> T): T {  
    ...  
}
```

# Software Transactional Memory (STM)

И использовать **atomic/read/write**

```
class Foo {  
    val a = TVar(1)  
    val b = TVar(2)  
  
    fun moveAtoB() = atomic { // начали  
        val t = a.read()      // читаем  
        b.write(t)            // пишем  
    }                          // закончили  
}
```

# Software Transactional Memory (STM)

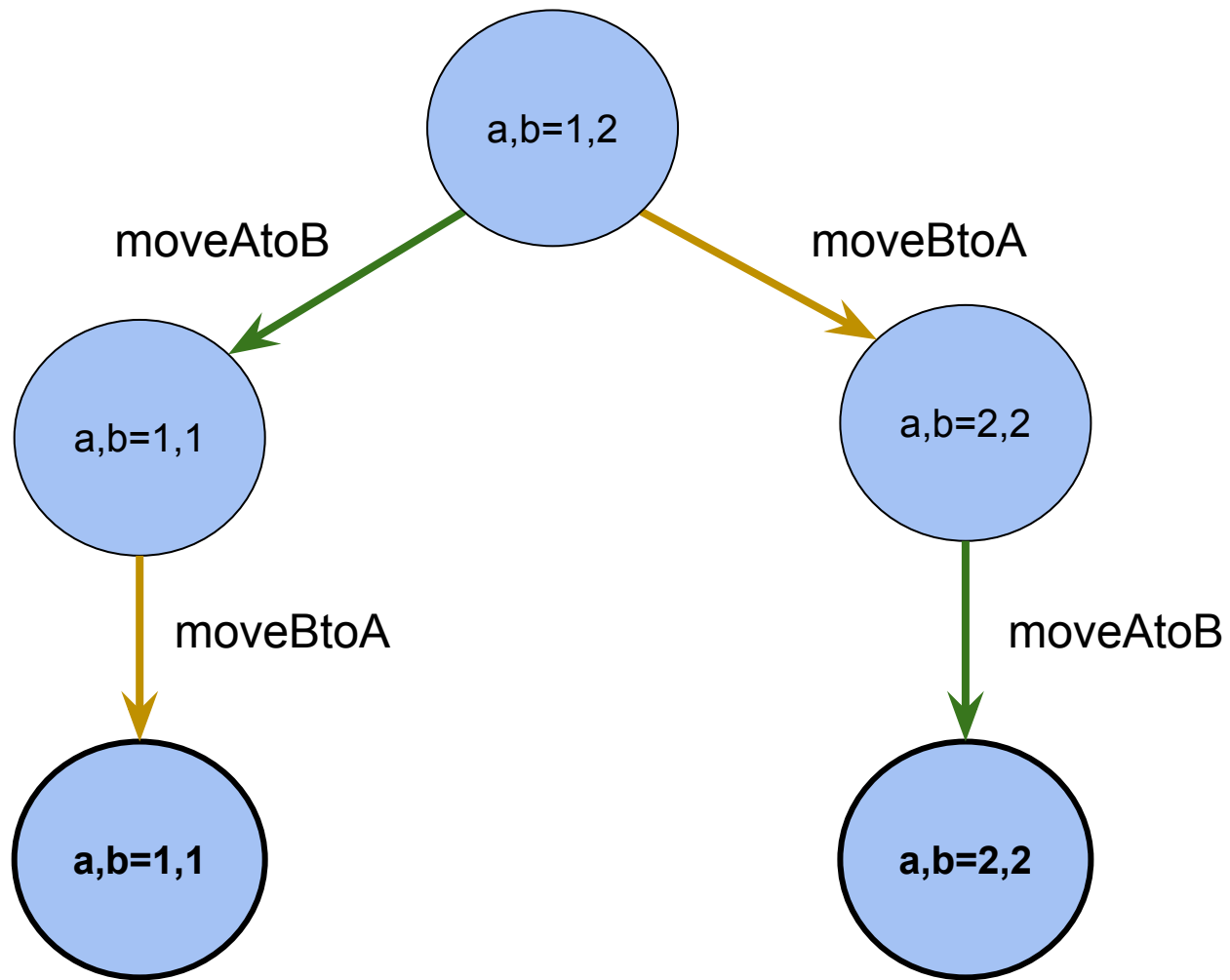
## Анатомия блока **atomic**

```
fun <T> atomic(block: Transaction.() -> T): T {  
    while (true) {  
        val transaction = beginTransaction()  
        try {  
            val result = block(transaction)  
            transaction.commit()  
            return result  
        } catch (e: AbortException) {  
            transaction.abort()  
        }  
    }  
}
```

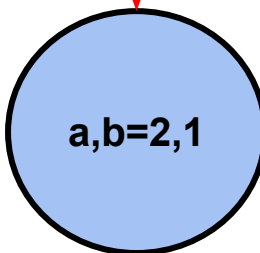
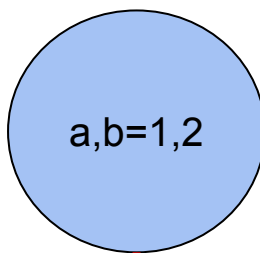
# Зачем abort?

- STM поддерживающий параллелизм на чтение не может гарантировать прогресс без отмены каких-то транзакций.

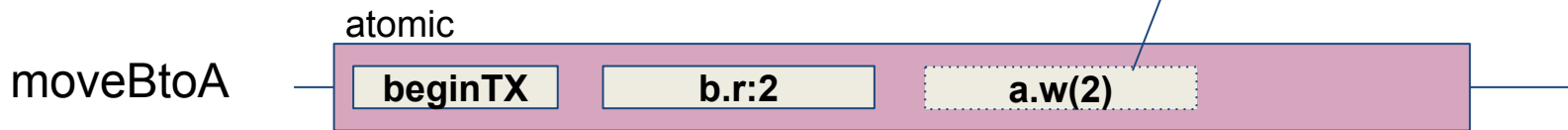
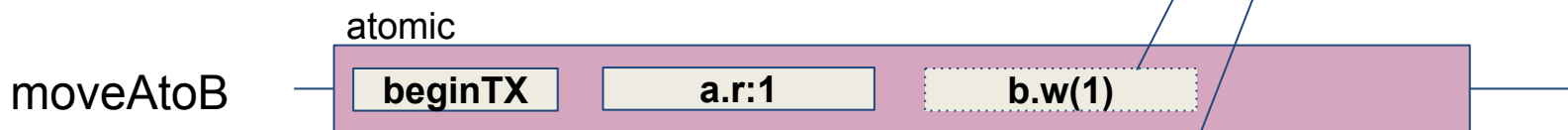
```
class Foo {  
    val a = TVar(1)  
    val b = TVar(2)  
  
    fun moveAtoB() = atomic {  
        val t = a.read()  
        b.write(t)  
    }  
  
    fun moveBtoA() = atomic {  
        val t = b.read()  
        a.write(t)  
    }  
}
```



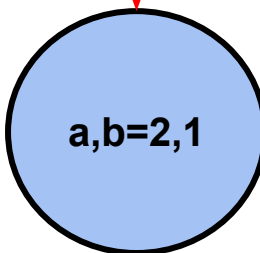
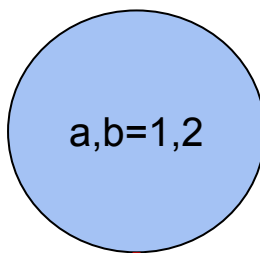




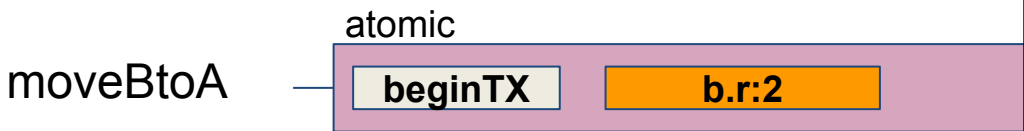
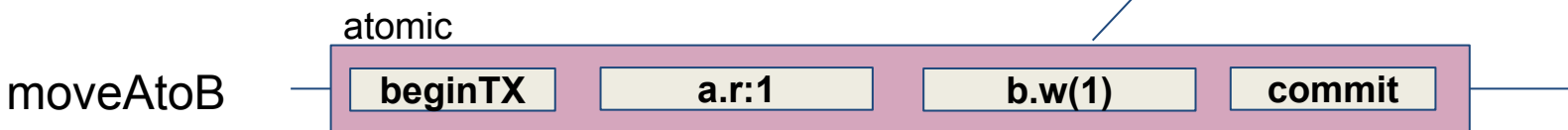
Обе транзакции не  
могут завершиться  
успешно.







Если эта транзакция завершилась



Эта транзакция “зомби”. Она еще жива, но уже не может успешно завершиться.

# Транзакции-зомби

- Работа с несогласованными значениями может плохо кончиться
- **Несогласованные** -- нарушающие инвариант

# Транзакции зомби

- Работа с несогласованными значениями может плохо кончиться
- **Несогласованные** -- нарушающие инвариант

```
class ZombieProblem {  
    val a = TVar(1)  
    val b = TVar(2)  
    // invariant a <= b
```

# Транзакции зомби

- Работа с несогласованными значениями может плохо кончиться
- **Несогласованные** -- нарушающие инвариант

```
class ZombieProblem {  
    val a = TVar(1)  
    val b = TVar(2)  
    // invariant a <= b  
  
    fun incA() {  
        atomic {  
            val updateA = a.read() + 1  
            a.write(updateA)  
            // потом восстановили инвариант  
            if (b.read() < updateA) b.write(updateA)  
        }  
    }  
}
```

# Транзакции зомби

- Работа с несогласованными значениями может плохо кончиться
- **Несогласованные** -- нарушающие инвариант

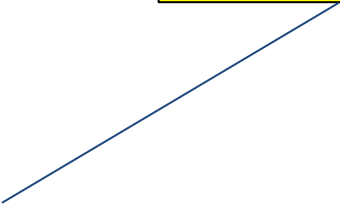
```
class ZombieProblem {  
    val a = TVar(1)  
    val b = TVar(2)  
    // invariant a <= b  
  
    fun doSomething() {  
        atomic {  
            var curA = a.read()  
            val curB = b.read()  
            while (curA++ != curB) {  
                /* something */  
            }  
        }  
    }  
}
```

# Транзакции зомби

- Работа с несогласованными значениями может плохо кончиться
- **Несогласованные** -- нарушающие инвариант

```
class ZombieProblem {  
    val a = TVar(1)  
    val b = TVar(2)  
    // invariant a <= b  
  
    fun doSomething() {  
        atomic {  
            var curA = a.read()  
            val curB = b.read()  
            while (curA++ != curB) {  
                /* something */  
            }  
        }  
    }  
}
```

Нельзя допускать  
несогласованное  
чтение!





# Транзакции зомби

- Работа с несогласованными значениями может плохо кончиться
- **Несогласованные** -- нарушающие инвариант
- Транзакционный менеджер должен обнаруживать и не допускать появления транзакций зомби:
  - Заставляя один транзакции ждать завершения других чтобы предотвратить появление транзакций-зомби (*но 100% предотвратить обеспечив параллельность чтения нельзя*)
  - Прерывая транзакции-зомби (**AbortException**)

**STM с блокировками**

# Самый простой STM - с блокировками

```
class TVar<T>(initial: T) : UpdgradeableLock() {  
    var value: T = initial  
    var oldValue: Any? = UNDEFINED  
  
    ...  
}
```

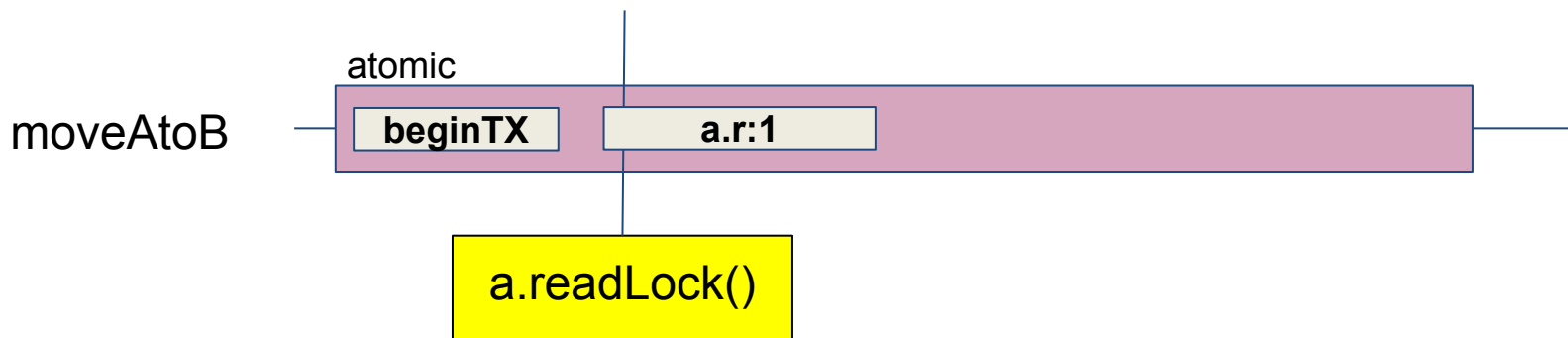
# Идея STM с блокировками

- Общая логика



# Идея STM с блокировками

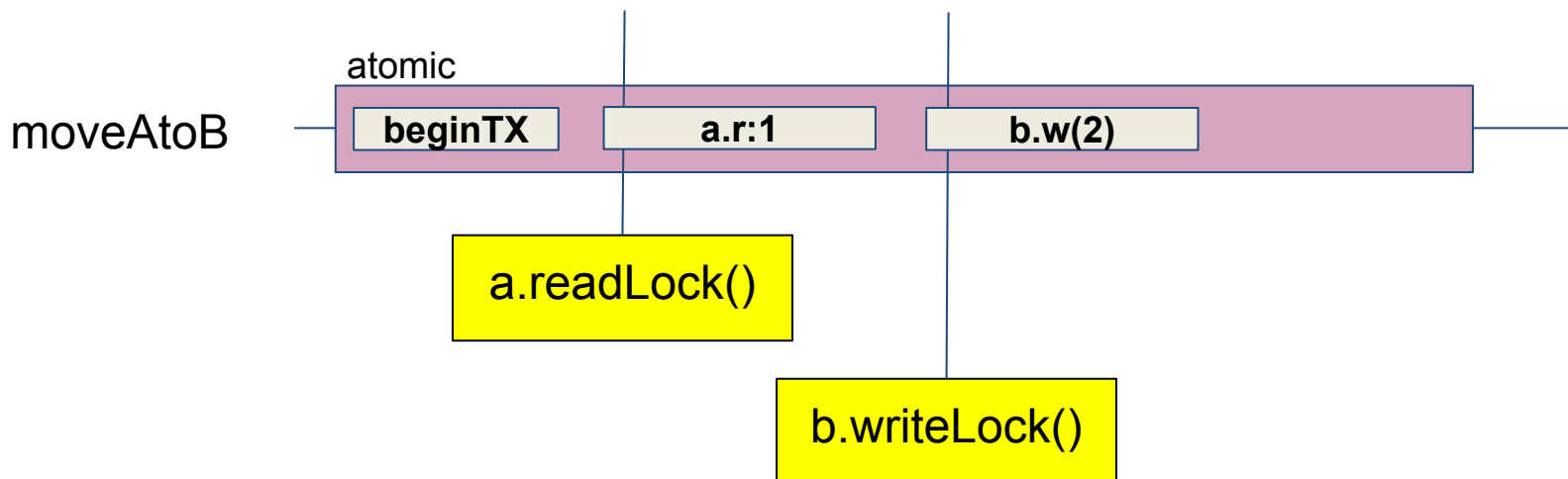
- **Общая логика**
  - При вызове read первый раз берем readLock



# Идея STM с блокировками

- **Общая логика**

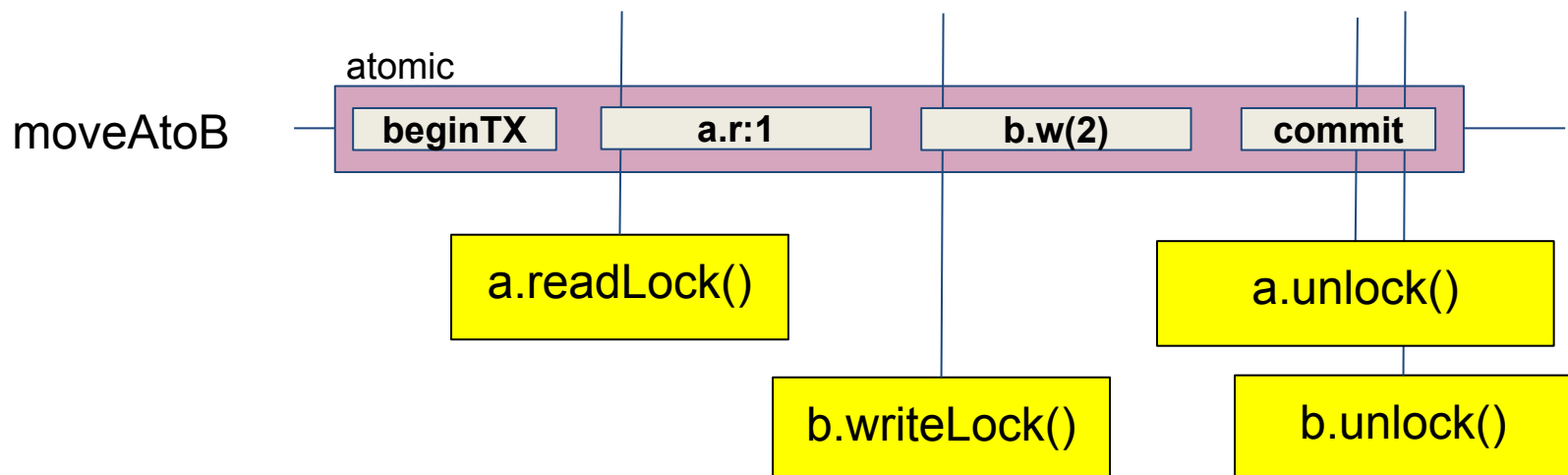
- При вызове read первый раз берем readLock
- При вызове write первый раз берем writeLock и запоминаем oldValue



# Идея STM с блокировками

- **Общая логика**

- При вызове read первый раз берем readLock
- При вызове write первый раз берем writeLock и запоминаем oldValue
- Во время commit сбрасываем oldValue во всех записанных значениях и отпускаем все блокировки

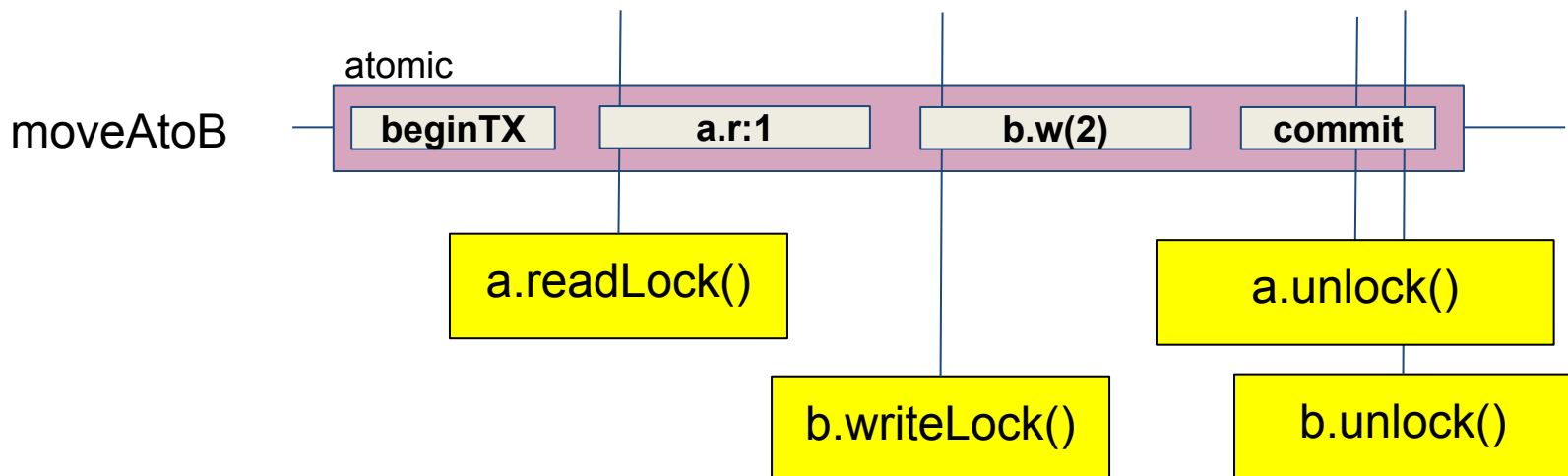


# Идея STM с блокировками

- **Общая логика**

- При вызове read первый раз берем readLock
- При вызове write первый раз берем writeLock и запоминаем oldValue
- Во время commit сбрасываем oldValue во всех записанных значениях и отпускаем все блокировки

- **Автоматом получаем 2PL => исполнение линейризуемо**





# Транзакция хранит список блокировок

```
enum class LockMode { READ, WRITE }
```

```
class Transaction {  
    val vars = HashMap<TVar<*>, LockMode>()
```

# Транзакция освобождает их при commit/rollback

```
fun commit() {  
    for ((v, m) in vars.entries) {  
        v.commit(m)  
    }  
}
```

```
fun abort() {  
    for ((v, m) in vars.entries) {  
        v.abort(m)  
    }  
}
```

# TVar с блокировками

```
fun readIn(tx: Transaction): T {  
    takeLockIn(tx, LockMode.READ)  
    return value  
}
```

# TVar с блокировками

```
fun takeLockIn(  
    tx: Transaction, mode: LockMode  
): LockMode? {  
    tx.vars[this]?.let { return it } // уже есть лок?  
    lock(mode)                        // возьмем  
    tx.vars[this] = mode             // запомним  
    return null  
}
```

# TVar с блокировками

```
fun writeIn(tx: Transaction, x: T) {  
    if (takeLockIn(tx, LockMode.WRITE) == LockMode.READ) {  
        // has read --> upgrade  
        upgradeLock()  
        tx.vars[this] = LockMode.WRITE  
    }  
    if (oldValue === UNDEFINED) oldValue = value  
    value = x  
}
```

# TVar: commit/rollback

```
fun commit(mode: LockMode) {  
    oldValue = UNDEFINED  
    unlock(mode)  
}
```

```
fun abort(mode: LockMode) {  
    if (mode == LockMode.WRITE) value = oldValue as T  
    unlock(mode)  
}
```

# Запись более подробно

**a**

<b>lock</b>	---
<b>value</b>	1
<b>oldValue</b>	UNDEF

# Запись более подробно

a		rd: 1
lock	---	READ
value	1	1
oldValue	UNDEF	UNDEF



# Запись более подробно

a		rd: 1	wr: 2
lock	---	READ	WRITE
value	1	1	2
oldValue	UNDEF	UNDEF	1

Upgrade

Undo Log

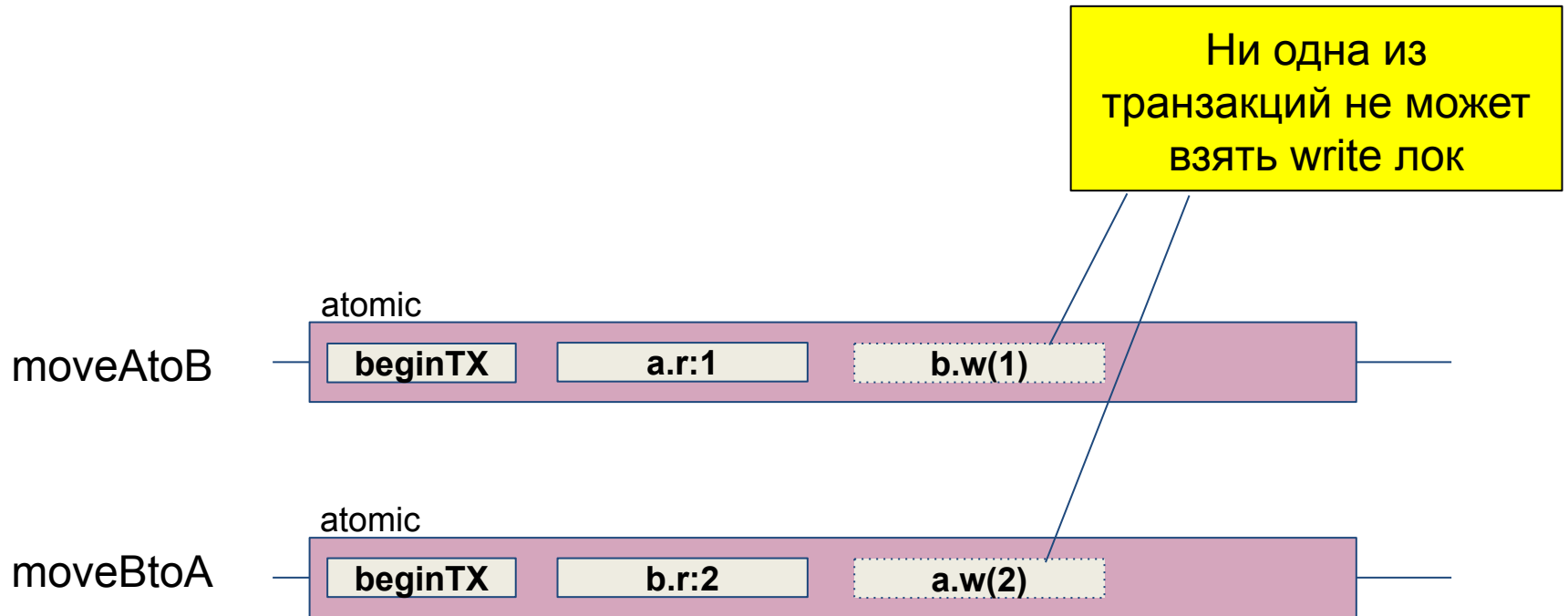
# Запись более подробно

a		rd: 1	wr: 2	commit
lock	---	READ	WRITE	---
value	1	1	2	2
oldValue	UNDEF	UNDEF	1	UNDEF

# Запись более подробно

a		rd: 1	wr: 2	abort
lock	---	READ	WRITE	---
value	1	1	2	1
oldValue	UNDEF	UNDEF	1	UNDEF

# А что если так?



# Deadlock

- Нужно реализовать **deadlock detector**
  - В простейшем случае -- ждать лока не более определенного времени
- В случае обнаружения взаимной блокировки надо одну из транзакций отменить (**AbortException**)

# Deadlock

- Нужно реализовать **deadlock detector**
  - В простейшем случае -- ждать лока не более определенного времени
- В случае обнаружения взаимной блокировки надо одну из транзакций отменить (**AbortException**)
- Полезно иметь **deadlock avoidance**
  - Заранее брать write lock если транзакция планирует писать

# Анализ STM с блокировками

- **Как тонкая блокировка**

- Можно сделать любую “гранулярность” защиты (на поле, на объект, на более сложную структура)
- Те же условные условия прогресса и переключение контекстов когда наткнулись на “занятую” блокировку

- **Преимущества**

- Программисту не нужно думать о порядке блокировок
  - Если что не так, то будет “прозрачный” **abort** и повтор операции
- Работает композиция
  - Можно написать **atomic** на уровне сверху, укрупняя транзакцию, не заводя новых блокировок и не нарушая инкапсуляцию

**STM без блокировок**



# Транзакция

```
enum class TxStatus { ACTIVE, COMMITTED, ABORTED }
```

```
class Transaction {  
    val status = AtomicReference(TxStatus.ACTIVE)
```

# Транзакция

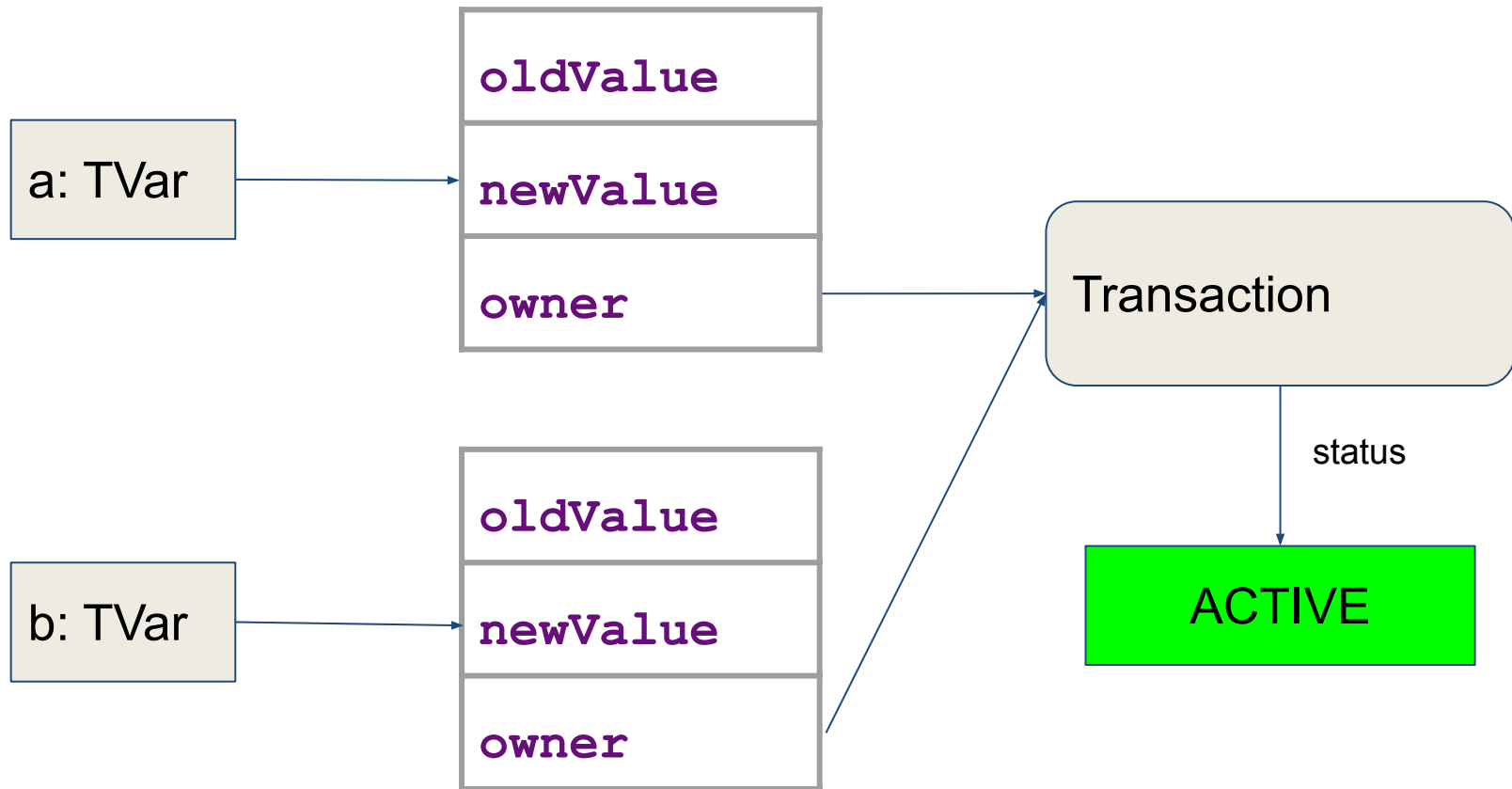
```
enum class TxStatus { ACTIVE, COMMITTED, ABORTED }
```

```
class Transaction {  
    val status = AtomicReference(TxStatus.ACTIVE)
```

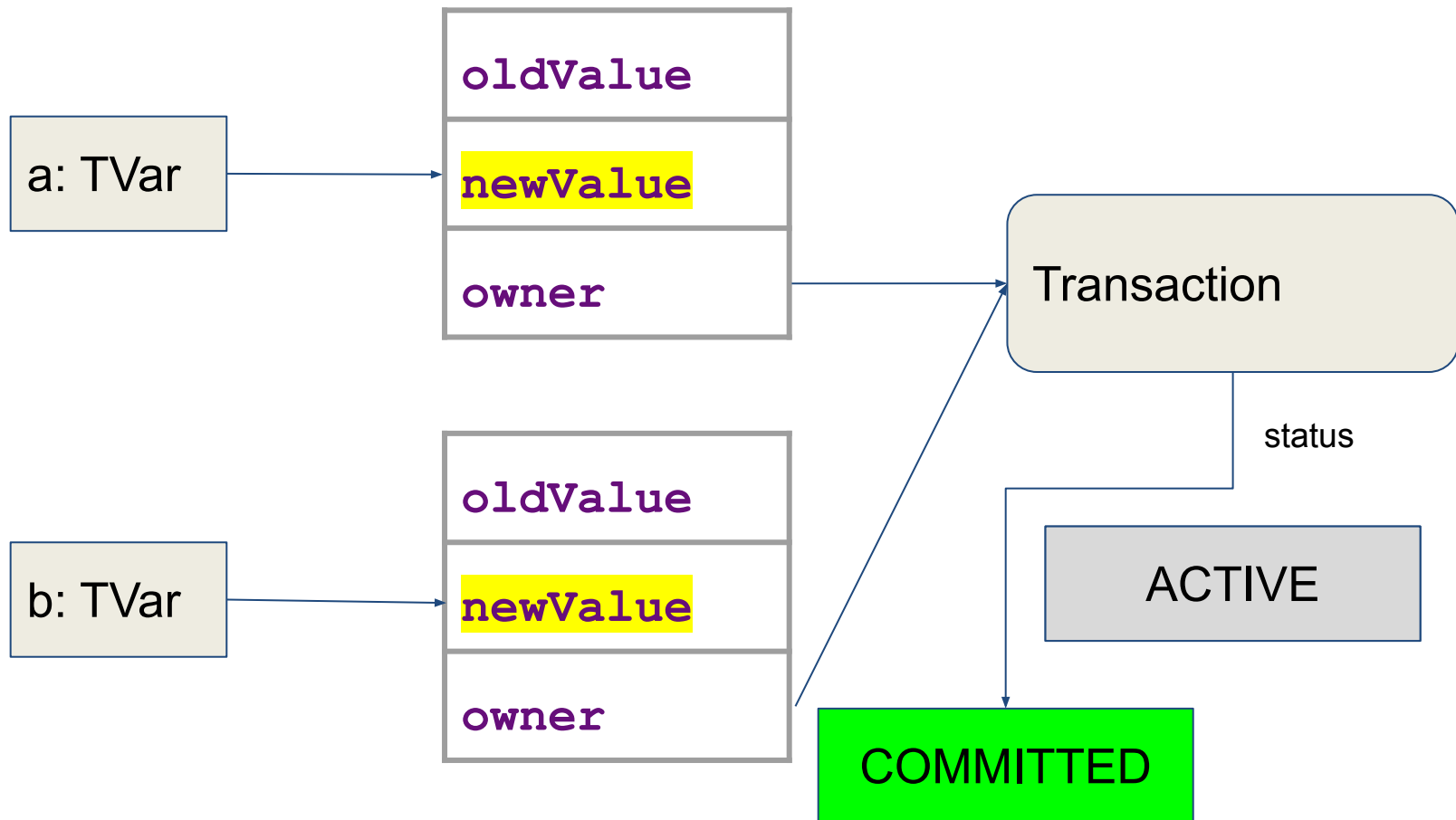
```
    fun commit(): Boolean =  
        status.compareAndSet(  
            TxStatus.ACTIVE, TxStatus.COMMITTED)
```

```
    fun abort() {  
        status.compareAndSet(  
            TxStatus.ACTIVE, TxStatus.ABORTED)  
    }
```

# Идея

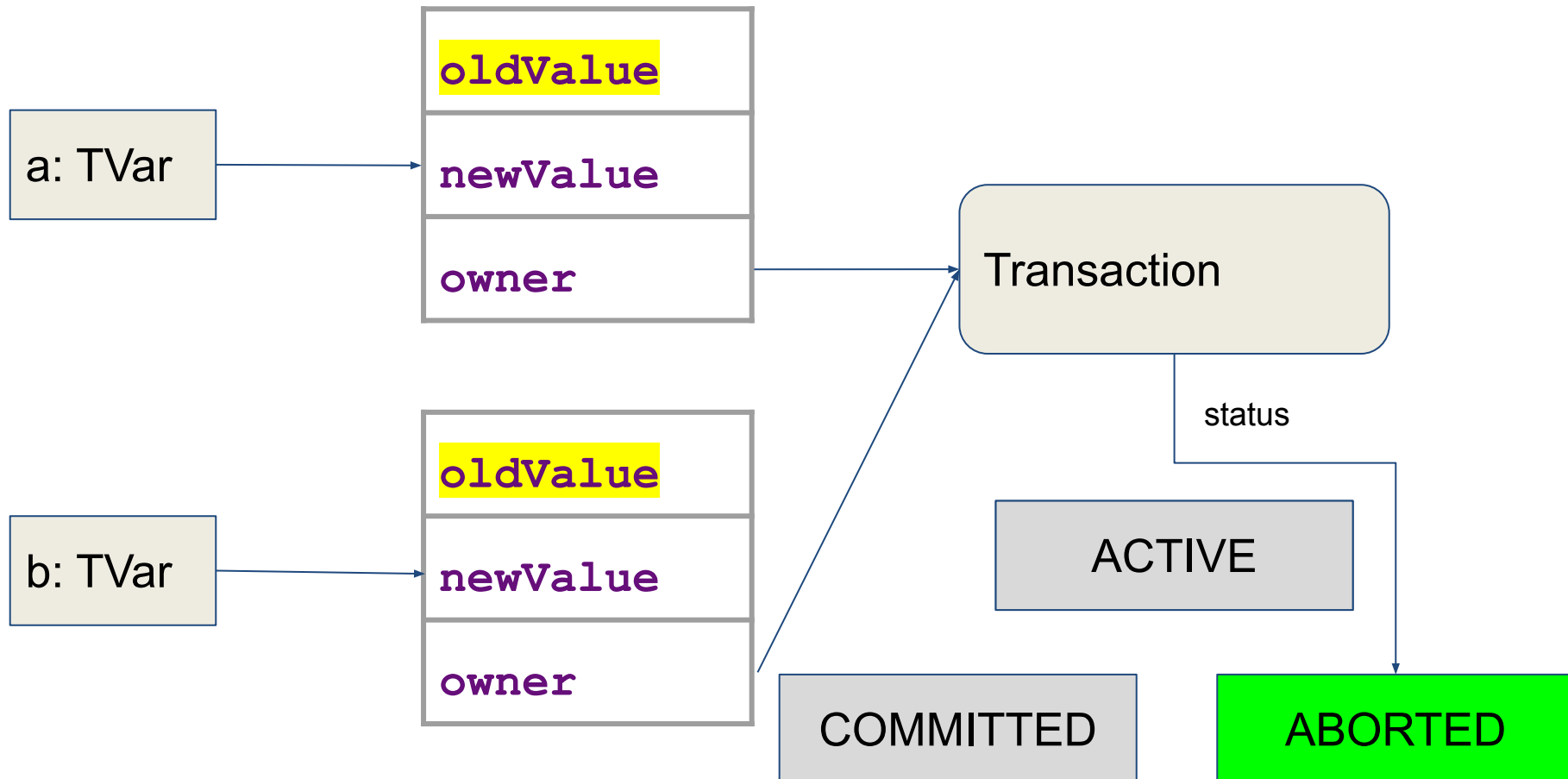


# Идея



**Линеаризация всех изменений в момент успешного  
CAS(ACTIVE,COMMITTED)**

# Идея



# Транзакционные переменные

```
class TVar<T>(initial: T) {  
    private val loc = AtomicReference(...)  
}
```

# Транзакционные переменные

```
class TVar<T>(initial: T) {  
    private val loc = AtomicReference(  
        Loc<T>(initial, initial, rootTx)  
    )  
}
```

```
class Loc<T>(  
    val oldValue: T,  
    val newValue: T,  
    val owner: Transaction  
)
```

```
private val rootTx = Transaction().apply { commit() }
```

# Узнать текущее значение

```
class Loc<T>(...) {  
    fun valueIn(tx: Transaction,  
               onActive: (Transaction) -> Unit): Any? =  
        if (owner === tx) newValue else  
            ...  
}  
}
```



# Узнать текущее значение

```
class Loc<T>(...) {  
    fun valueIn(tx: Transaction,  
                onActive: (Transaction) -> Unit): Any? =  
        if (owner === tx) newValue else  
            when (owner.status.get()!!) {  
                TxStatus.ABORTED -> oldValue  
                ...  
            }  
}  
}
```

# Узнать текущее значение

```
class Loc<T>(...) {  
    fun valueIn(tx: Transaction,  
                onActive: (Transaction) -> Unit): Any? =  
        if (owner === tx) newValue else  
            when (owner.status.get()!!) {  
                TxStatus.ABORTED -> oldValue  
                TxStatus.COMMITTED -> newValue  
                ...  
            }  
    }  
}
```

# Узнать текущее значение

```
class Loc<T>(...) {  
    fun valueIn(tx: Transaction,  
                onActive: (Transaction) -> Unit): Any? =  
        if (owner === tx) newValue else  
            when (owner.status.get()!!) {  
                TxStatus.ABORTED -> oldValue  
                TxStatus.COMMITTED -> newValue  
                TxStatus.ACTIVE -> {  
                    onActive(owner)  
                    TxStatus.ACTIVE  
                }  
            }  
    }  
}
```

# “Открываем” переменную при любой операции

```
class TVar<T> {  
    fun openIn(tx: Transaction, update: (T) -> T): T {  
        while (true) {  
            val curLoc = loc.get()  
            ...  
        }  
    }  
}
```

# “Открываем” переменную при любой операции

```
class TVar<T> {  
    fun openIn(tx: Transaction, update: (T) -> T): T {  
        while (true) {  
            val curLoc = loc.get()  
            val curValue = curLoc.valueIn(tx) { ... }  
        }  
    }  
}
```

# “Открываем” переменную при любой операции

```
class TVar<T> {  
    fun openIn(tx: Transaction, update: (T) -> T): T {  
        while (true) {  
            val curLoc = loc.get()  
            val curValue = curLoc.valueIn(tx) { owner ->  
                contention(tx, owner)  
            }  
        }  
    }  
}
```

# Что делать если работает другая транзакция?

```
fun contention(tx: Transaction, owner: Transaction)
```

- **Abort:** Отменить другую транзакцию (`owner.abort()`)
- **Backoff:** Подождать немного
- **Priority:** Нумеровать транзакции: отметить более новые, более старые ждут.
- Отслеживать “сколько работы транзакция сделала”
- И т.п.

# “Открываем” переменную при любой операции

```
class TVar<T> {  
    fun openIn(tx: Transaction, update: (T) -> T): T {  
        while (true) {  
            val curLoc = loc.get()  
            val curValue = curLoc.valueIn(tx) { ... }  
            if (curValue === TxStatus.ACTIVE) continue  
        }  
    }  
}
```



# “Открываем” переменную при любой операции

```
class TVar<T> {  
    fun openIn(tx: Transaction, update: (T) -> T): T {  
        while (true) {  
            val curLoc = loc.get()  
            val curValue = curLoc.valueIn(tx) { ... }  
            if (curValue == TxStatus.ACTIVE) continue  
            val updValue = update(curValue as T)  
        }  
    }  
}  
  
fun readIn(tx: Transaction): T = openIn(tx) { it }  
fun writeIn(tx: Transaction, x: T) = openIn(tx) { x }
```

# “Открываем” переменную при любой операции

```
class TVar<T> {  
    fun openIn(tx: Transaction, update: (T) -> T): T {  
        while (true) {  
            val curLoc = loc.get()  
            val curValue = curLoc.valueIn(tx) { ... }  
            if (curValue === TxStatus.ACTIVE) continue  
            val updValue = update(curValue as T)  
            val updLoc = Loc(curValue, updValue, tx)  
            if (loc.compareAndSet(curLoc, updLoc)) {  
                ...  
            }  
        }  
    }  
}
```

# “Открываем” переменную при любой операции

```
class TVar<T> {  
    fun openIn(tx: Transaction, update: (T) -> T): T {  
        while (true) {  
            val curLoc = loc.get()  
            val curValue = curLoc.valueIn(tx) { ... }  
            if (curValue == TxStatus.ACTIVE) continue  
            val updValue = update(curValue as T)  
            val updLoc = Loc(curValue, updValue, tx)  
            if (loc.compareAndSet(curLoc, updLoc)) {  
                if (tx.status.get() == TxStatus.ABORTED)  
                    throw AbortException()  
                return updValue  
            }  
        }  
    }  
}
```

Валидация  
(против зомби-транзакций)

# Анализ STM без блокировок

- Открываем “эксклюзивно” при любой операции
  - Но можно модернизировать реализацию для поддержки множества параллельных чтений
- Любая модификация исключает параллельное чтение
  - Но можно сделать так, чтобы читатели ждали, как было с блокировками
- И вообще, меняя реализацию алгоритма contention можно добиться поведения как с локами (всегда ждать если транзакционная переменная уже открыта/заблокирована)
  - Но самое интересное -- это приоритет более старой транзакции (гарантия ее завершения)

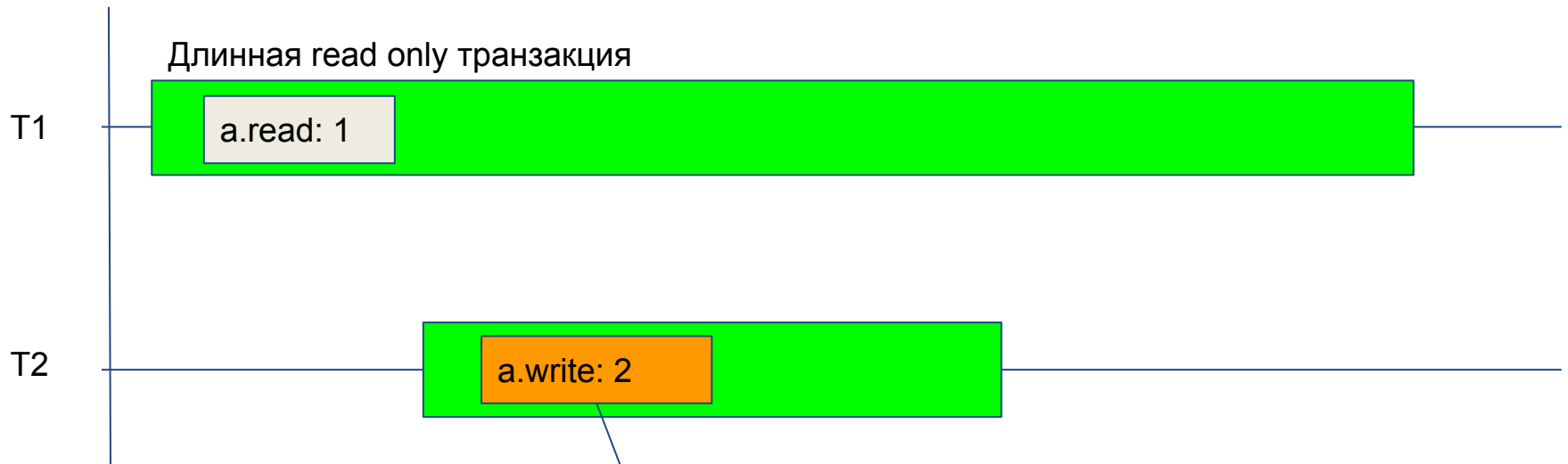
# Длинные Read Only транзакции

- Либо они не могут завершиться (ждут других)
- Либо мешают работать другим транзакция (мешают писать)
- Выход есть: **MVCC (Multi-Version Concurrency Control)**

**MVCC**

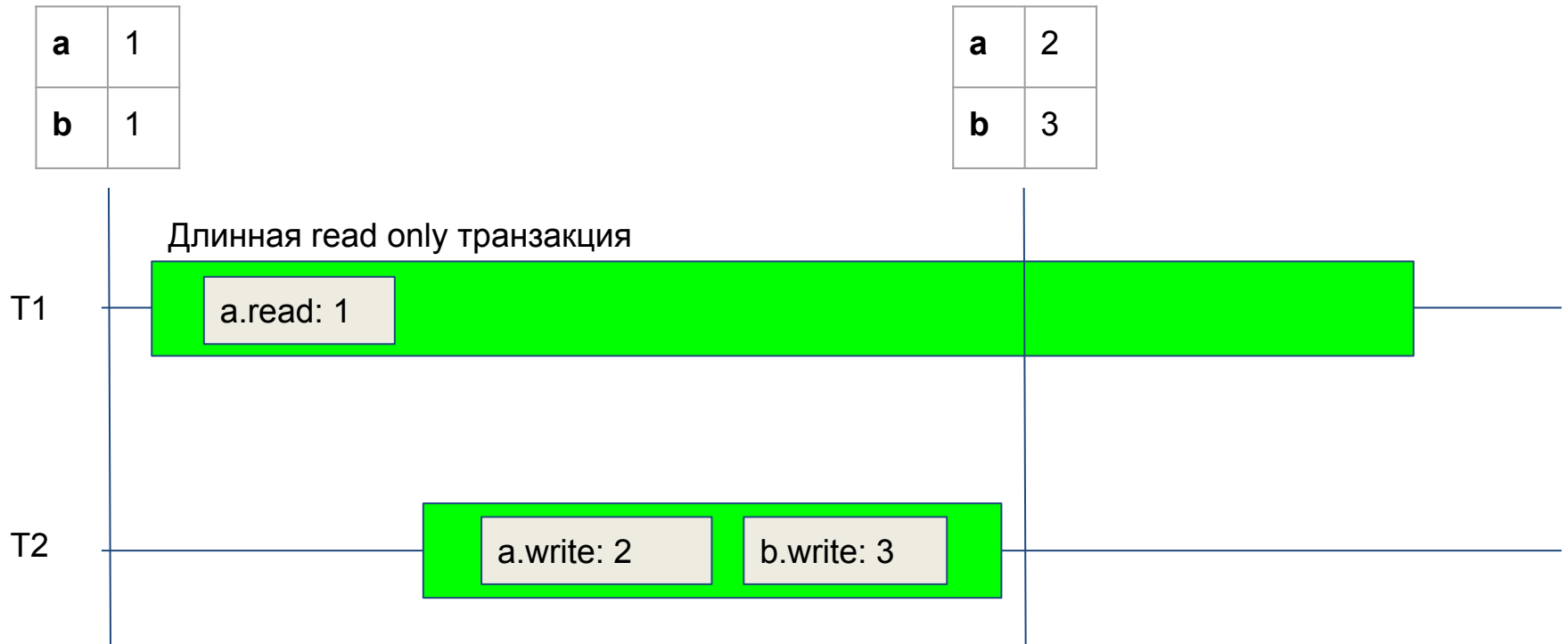
# Общая проблема

<b>a</b>	1
<b>b</b>	1



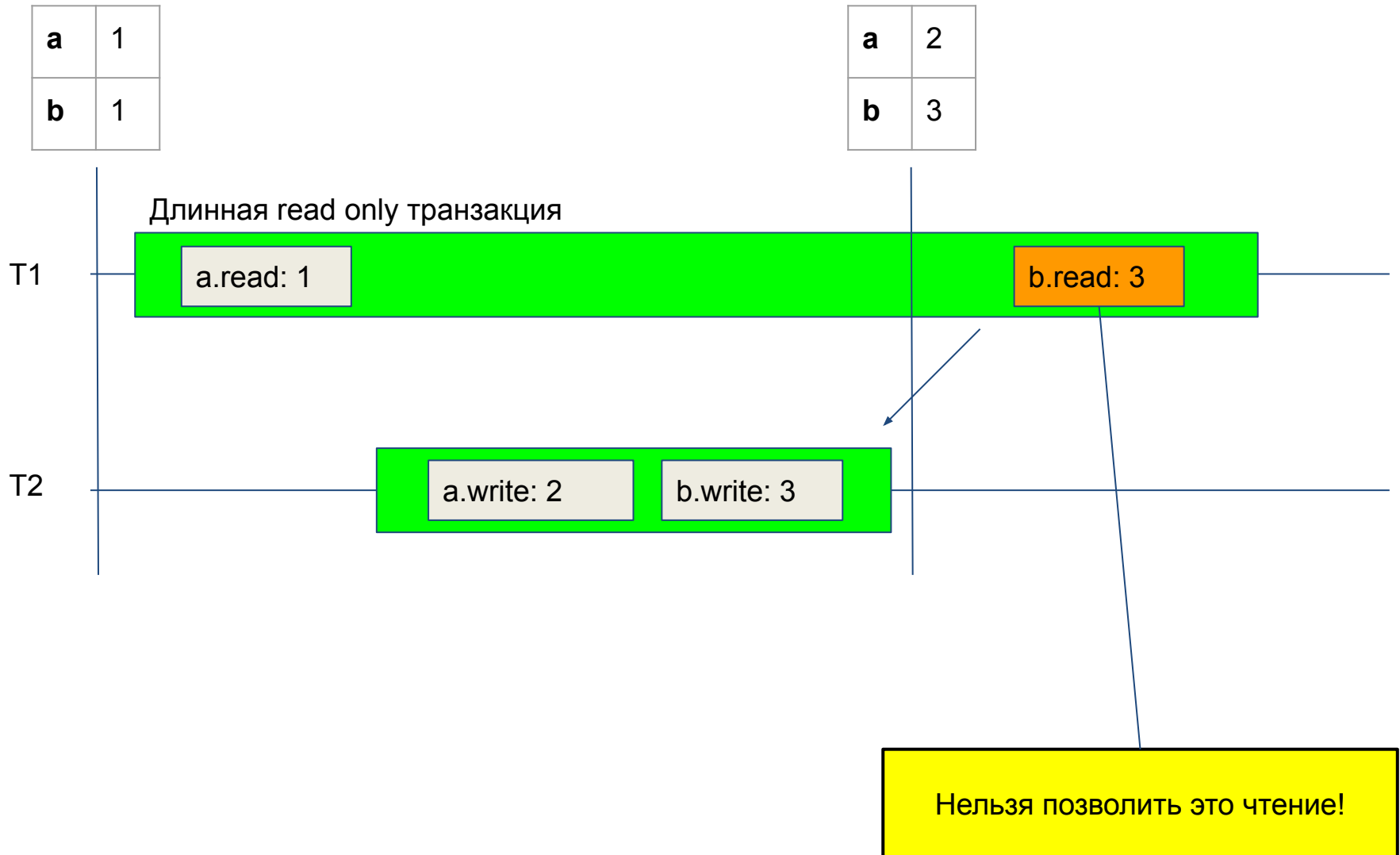
Надо либо ждать завершения T1 либо откатить T1/T2.  
**Почему нельзя позволить ей записать?**

# Общая проблема

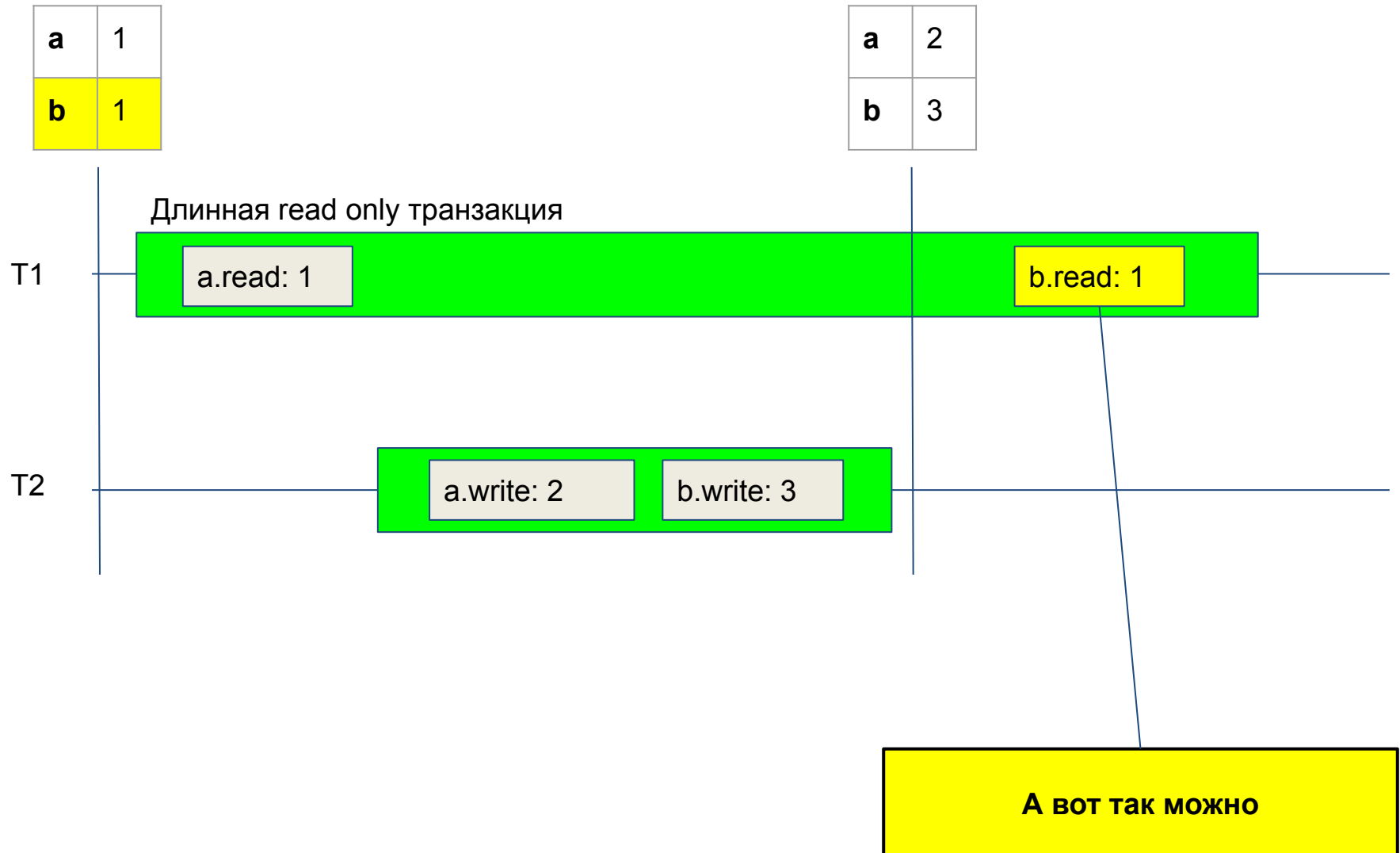




# Несогласованная картина мира



# MVCC: Согласованная картина мира



# MVCC: Основная идея

- Заведём глобальный номер транзакции  $\sim$  текущее время
- При начале транзакции запоминаем её время.
- Все чтения возвращают значения на начало транзакции

# MVCC

	val	tst
<b>a</b>	1	0
<b>b</b>	1	0

T1



stamp = 1

# MVCC

	val	tst
<b>a</b>	1	0
<b>b</b>	1	0

T1

a.read: 1

stamp = 1



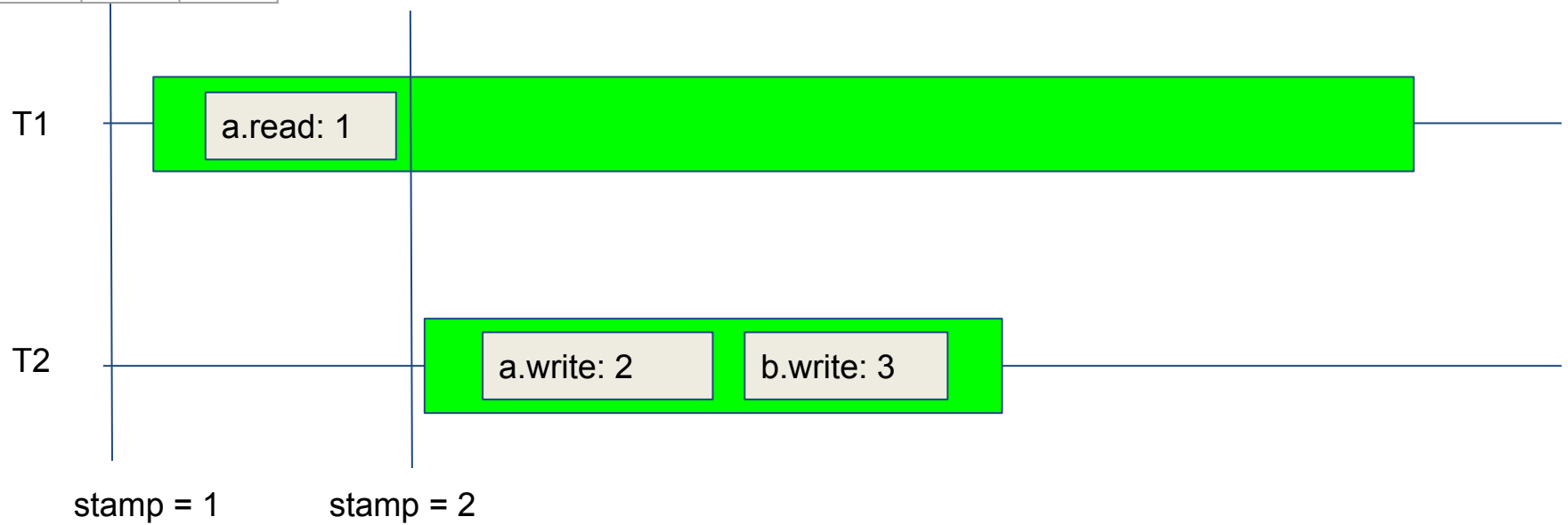
# MVCC

	val	tst
<b>a</b>	1	0
<b>b</b>	1	0



# MVCC

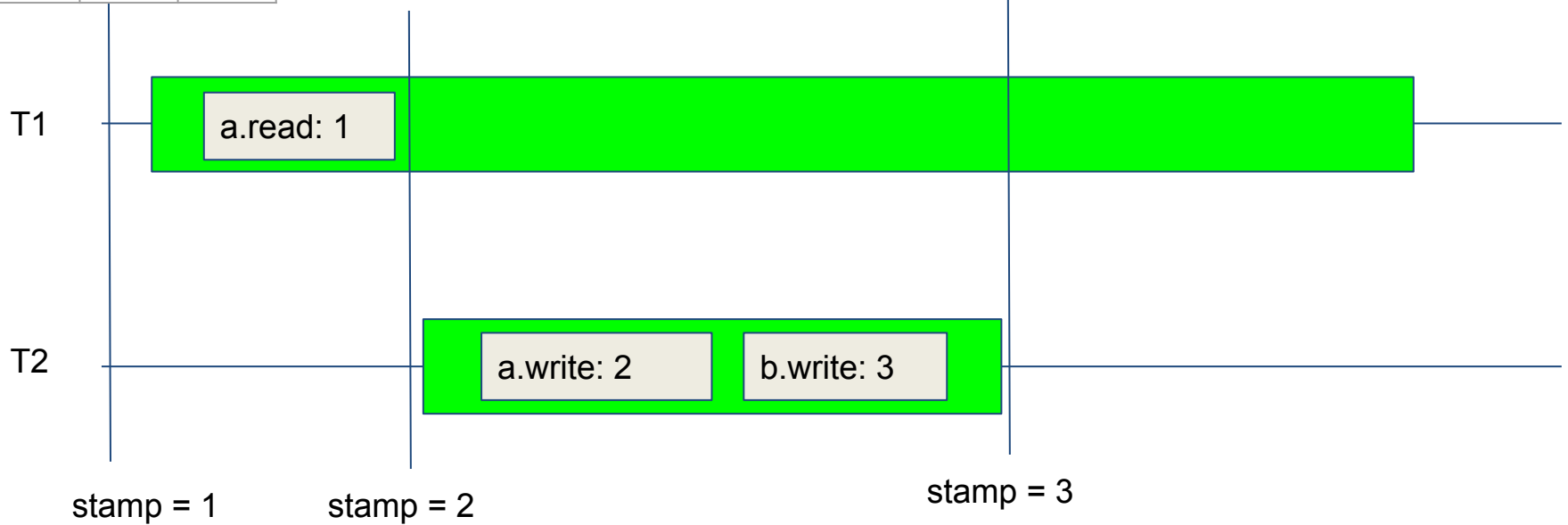
	val	tst
<b>a</b>	1	0
<b>b</b>	1	0



# MVCC

	val	tst
<b>a</b>	1	0
<b>b</b>	1	0

	val	tst
<b>a</b>	2	3
<b>b</b>	3	3



**Отдельное время завершения  
транзакции!  
(Время линеаризации)**



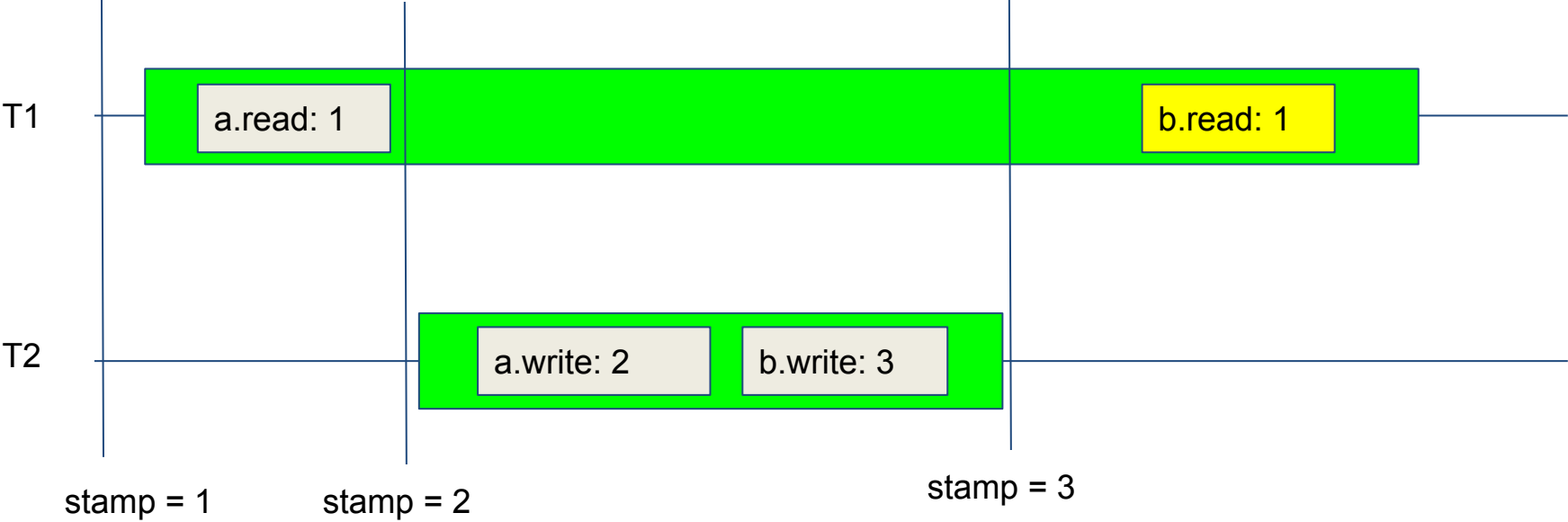
# MVCC



# MVCC

	val	tst
<b>a</b>	1	0
<b>b</b>	1	0

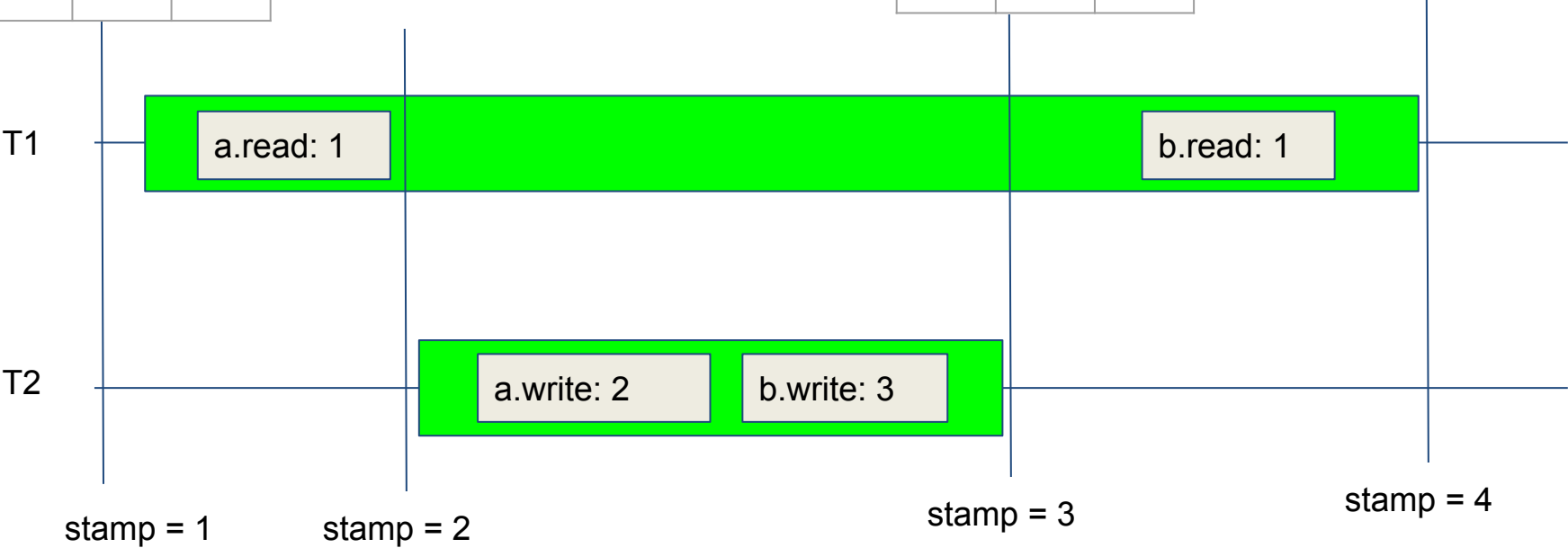
	val	tst
<b>a</b>	2	3
<b>b</b>	3	3



# MVCC

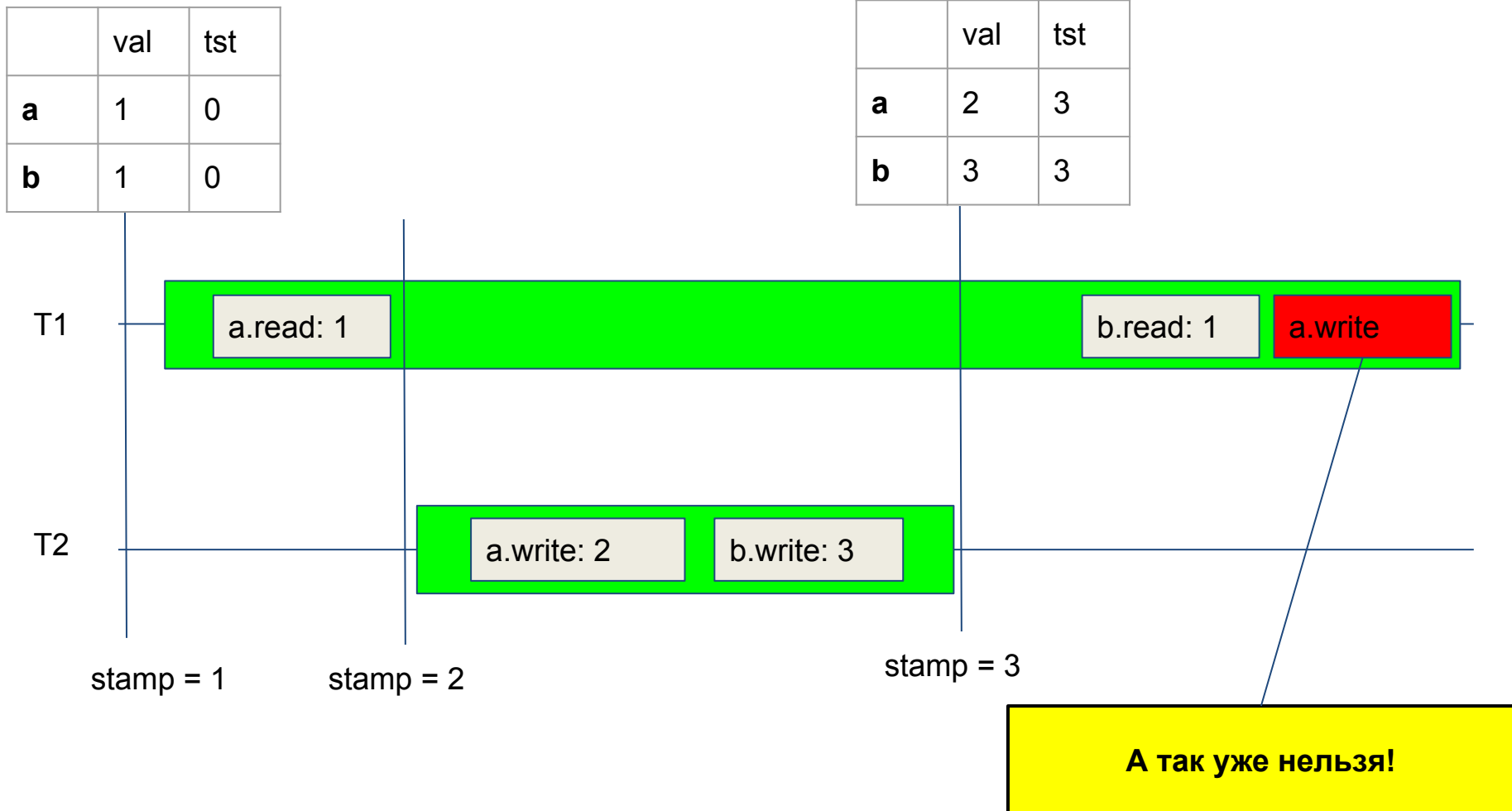
	val	tst
<b>a</b>	1	0
<b>b</b>	1	0

	val	tst
<b>a</b>	2	3
<b>b</b>	3	3



**Можем завершить читающую транзакцию**

# MVCC



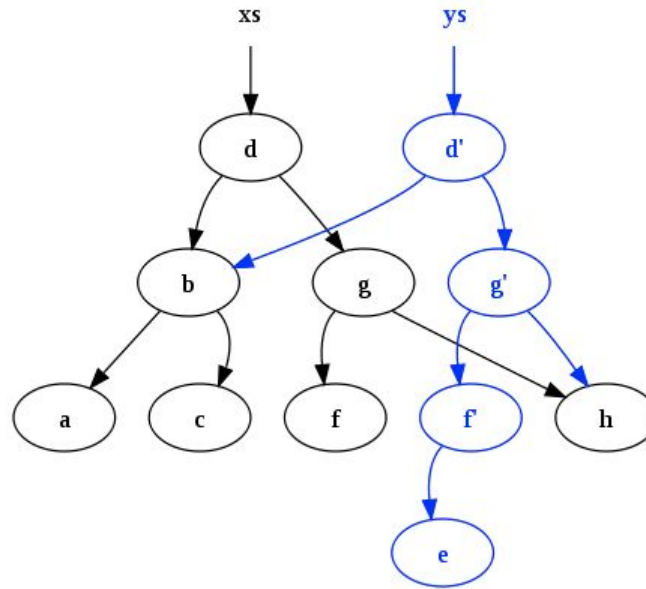
Но если позволить этой транзакции работать, то это безопасный “зомби” -- можно проверять согласованность записей только во время завершения транзакции.

# Обзор STM

# Общий анализ STM

- STM это очень медленно
  - Каждый read/write это всегда какой-то indirection
- STM без MVCC могут быть реализованы так, что накладные расходы есть только на значения “участвующие в транзакции”
  - Каждое значение это либо “реальное значение” либо указатель на некий дескриптор хранящий транзакцию владелец, локи, старые значения и т.п.
  - Но без MVCC и без блокировок длинные readonly транзакции
- STM+MVCC всегда вынужден хранить как минимум пару (значение, версия) + еще значения для старых версий (для которых еще есть активные транзакции)

# Персистентные структуры данных



- Персистентные структуры = immutable + древовидные
  - Обновление обычно за  $O(\log N)$
  - Существуют “эффективные” реализации большинства классических структур
  - Иногда удается добиться  $O(1)$  но большая константа

# STM + Персистентные структуры данных =

- А что если всё состояние приложения хранить в одной или нескольких персистентных структурах данных?
  - Тогда можно спокойно использовать STM
  - Можно использовать STM+MVCC
  - Но, платим логарифмом + выделения памяти при всех обновлениях



# Hardware Transactional Memory

# Hardware Transactional Memory (HTM)

- Intel Transactional Synchronization Extensions (TSX)
  - XBEGIN - начинает транзакцию
  - XEND - завершает транзакцию
  - XABORT - откатывает транзакцию

# Но как?

- Протоколы когерентности кэша (MESI)
- Значение, которое write в транзакции храним Exclusive/Modified
  - Если другой процессор хочет его читать - abort
- Значение, которые read в транзакции храним Shared/E/M
  - Если другой процессор его хочет менять - abort
- Сбрасываем в основную память только при успешном окончании транзакции
  - Если кто-то хочет значение после хенд, но до того, как успели сбросить -- не проблема (умеет передавать по шине)
- По сути всё очень просто - нужен еще один бит о том, что значение это часть транзакции

# Анализ НТМ

- Очень быстро работает (вообще нет накладных расходов)
- Будет надежно работать только для очень быстрых транзакций (сложно успешно завершить длинную из-за aborts)
- Ограничено физическим размером кэша
  - Что хуже - ограничено ассоциативностью

# HTM для избежания блокировок (Lock Elision)

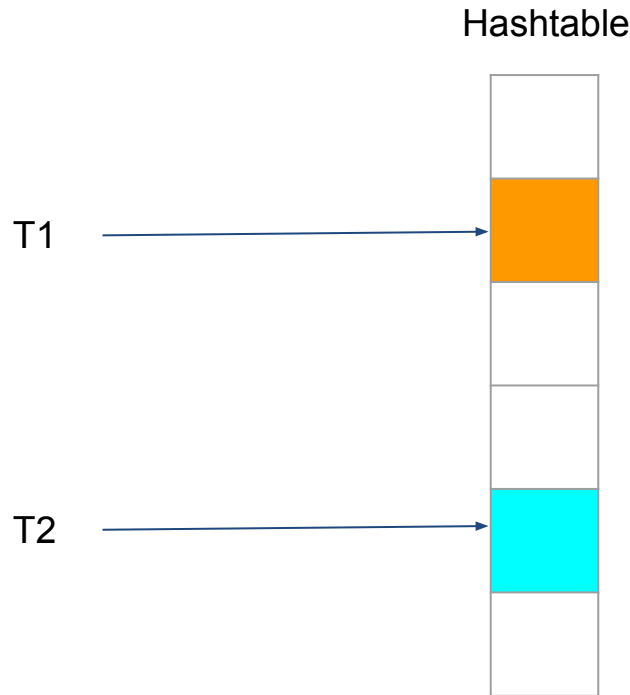
```
class Lock {  
    val state = atomic(0)  
  
    fun lock() {  
        while (true) {  
            if (state.compareAndSet(0, 1)) break  
            // wait & retry  
        }  
    }  
  
    fun unlock() {  
        state.value = 0  
    }  
}
```

# HTM для избежания блокировок (Lock Elision)

```
fun lock() {  
    if (xbegin()) {  
        if (state.value == 0) return  
        xabort() // иначе берем лок обычно  
    }  
    while (true) {  
        if (state.compareAndSet(0, 1)) break  
        // wait & retry  
    }  
}  
  
fun unlock() {  
    if (state.value == 0) {  
        xend()  
        return  
    }  
    state.value = 0  
}
```

# НТМ для избежания блокировок (Lock Elision)

- Можно писать код с грубой блокировкой
- А получать параллелизм как с тонкой блокировкой



# HTM LE: Практические проблемы

- В реальных структурах данных есть какой-нибудь size из-за изменений которого параллелизма транзакций не будет
- Как только одна транзакция пошла по software пути (по любой причине), всё сваливается на STM



# HTM для специфичных структур данных

- DCSS/CASN можно эффективно реализовать через HTM
- Всегда можно сделать fallback на программную реализацию
  - Алгоритм Хариса “HTM ready”

# Hybrid Transaction Manager

- Пытаемся делать `atomic { ... }` через HTM
- Если не получилось -- fallback на STM