Многопоточное Программирование: CASN

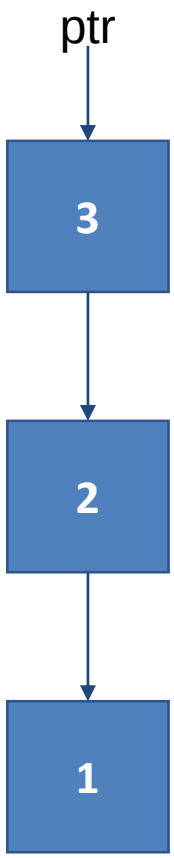
Роман Елизаров, JetBrains, elizarov@gmail.com
Никита Коваль, JetBrains, ndkoval@ya.ru

ИТМО 2020

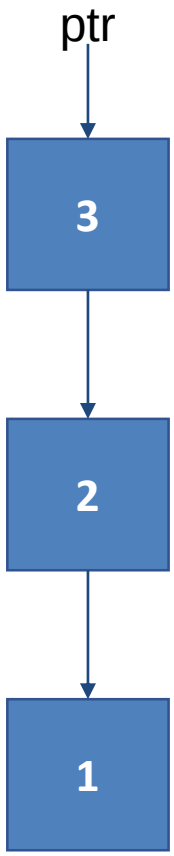
ITMO UNIVERSITY

Списки против Массивов

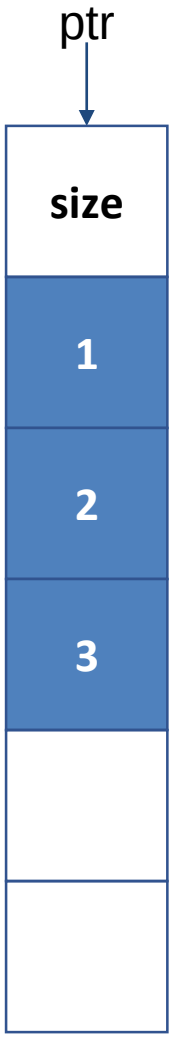
Список



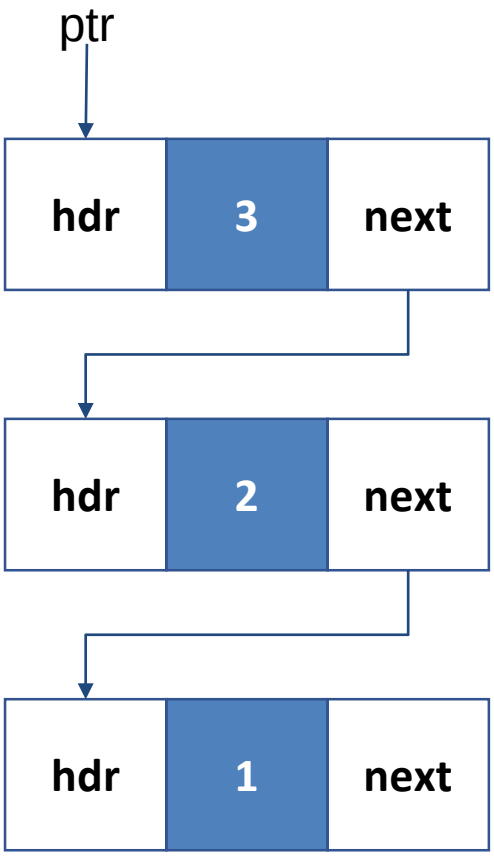
Список



Массив

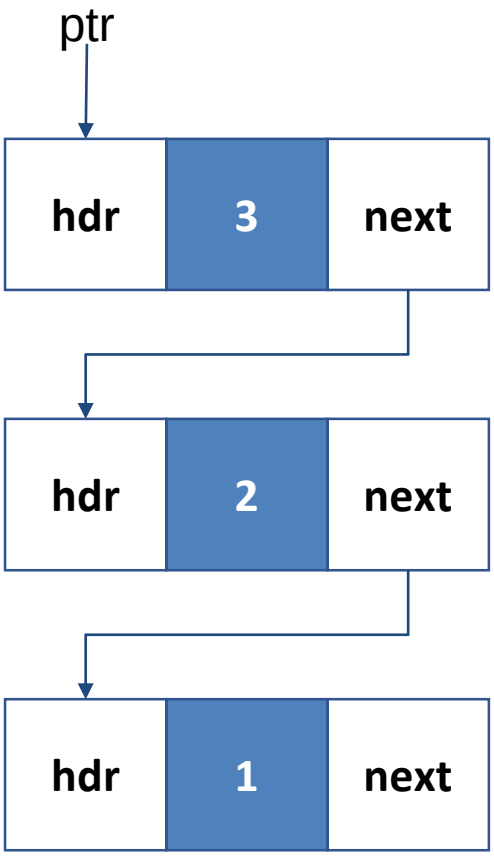


Список



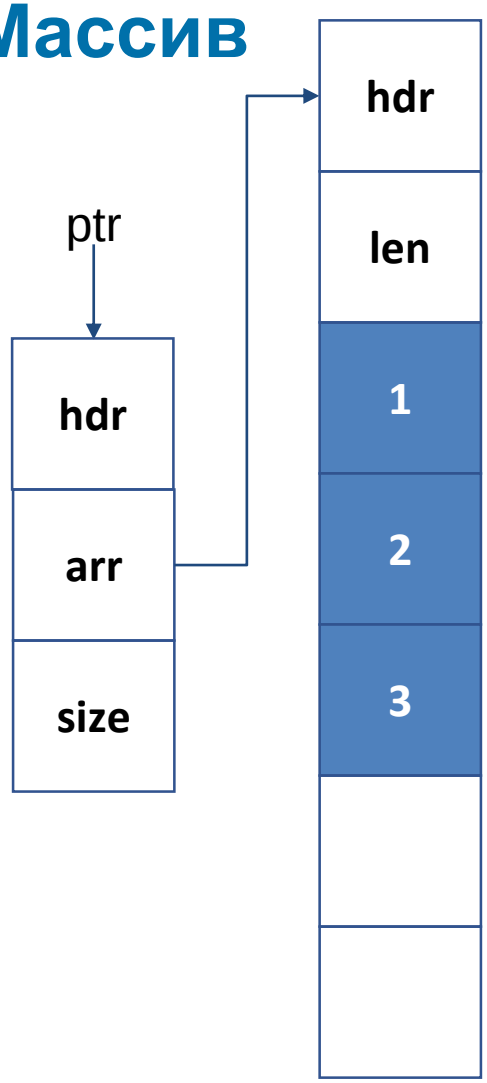
3*N слов памяти

Список



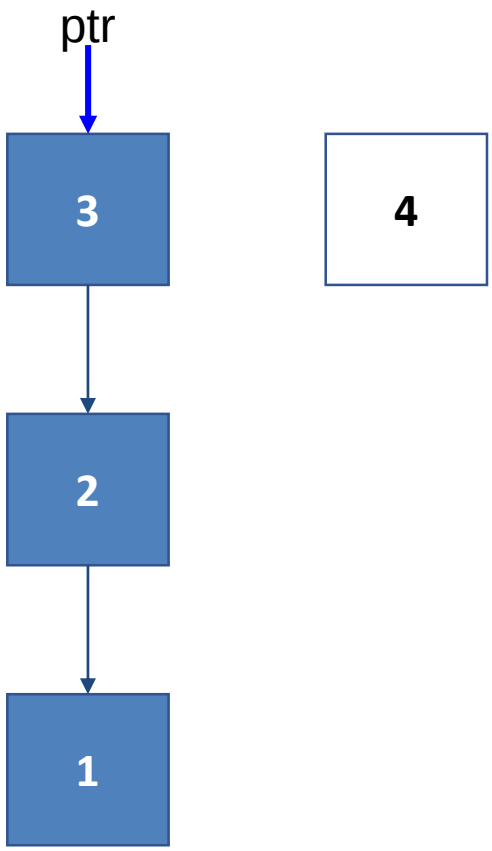
3*N слов памяти

Массив

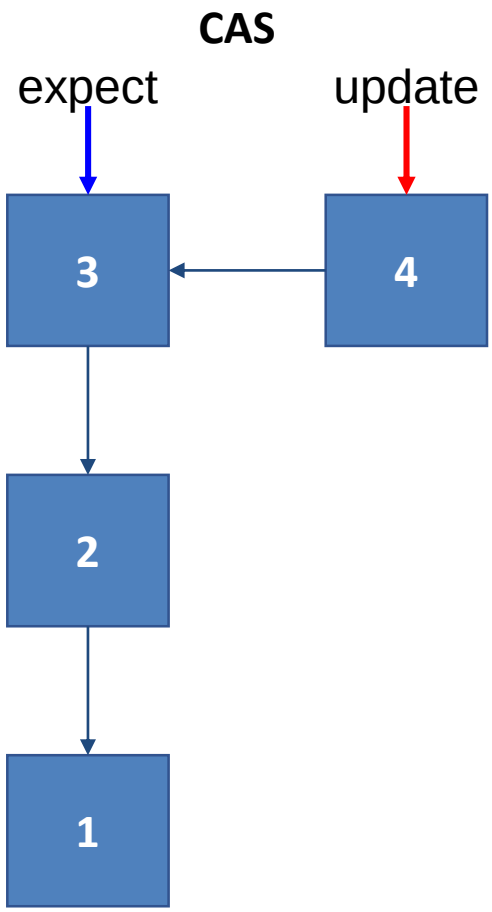


5 + N слов памяти
(до 5 + 2*N при x2 резерве)

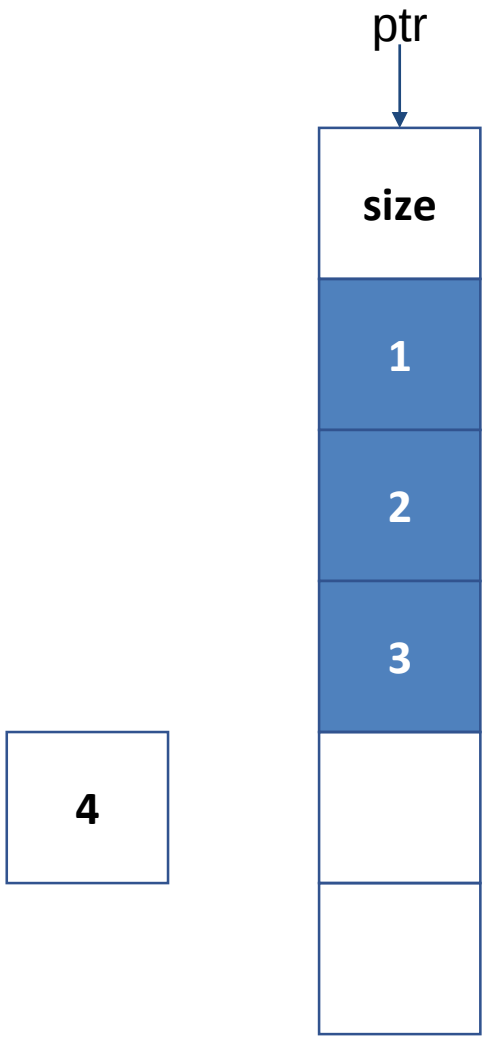
Стек на списке



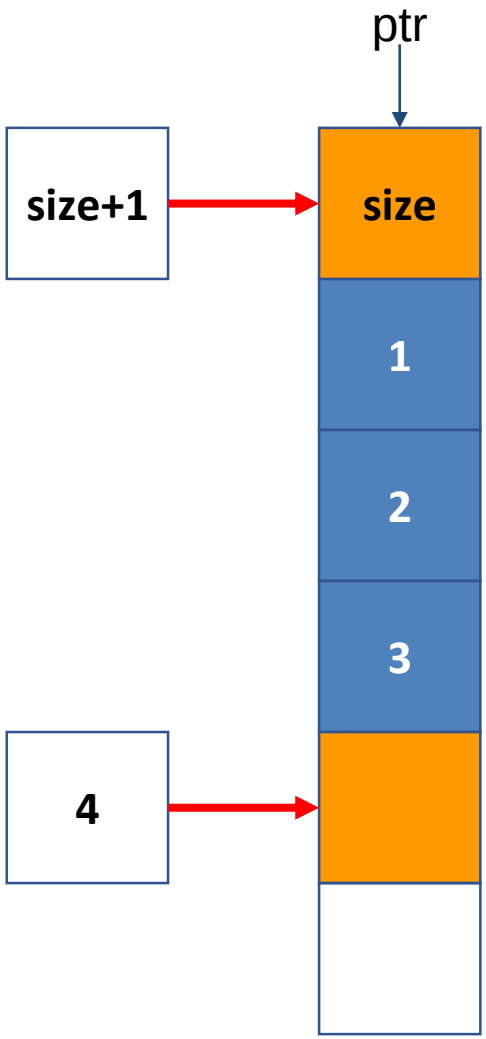
Стек на списке



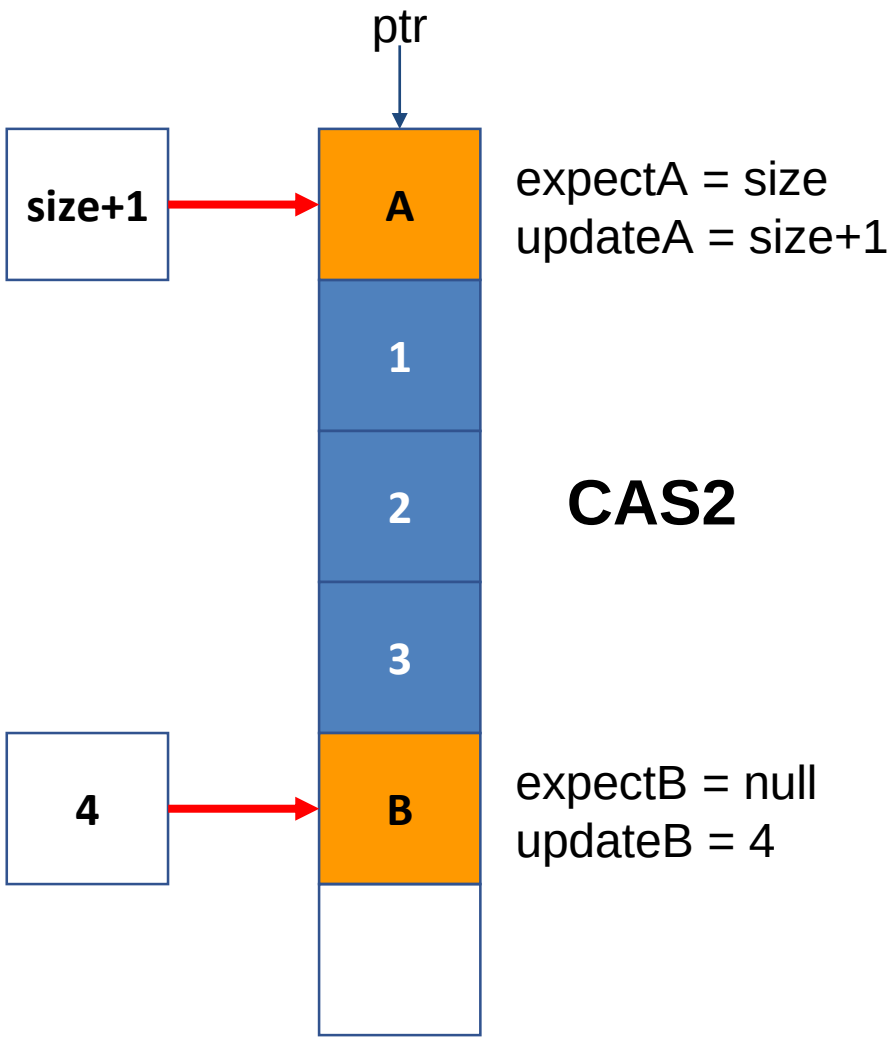
Стек на массиве



Стек на массиве



Стек на массиве



CASn

DISC 2002

A Practical Multi-Word Compare-and-Swap Operation

Timothy L. Harris, Keir Fraser and Ian A. Pratt

University of Cambridge Computer Laboratory, Cambridge, UK
{tim.harris,keir.fraser,ian.pratt}@cl.cam.ac.uk

Abstract. Work on non-blocking data structures has proposed extending processor designs with a compare-and-swap primitive, `CAS2`, which acts on two arbitrary memory locations. Experience suggested that current operations, typically single-word compare-and-swap (`CAS1`), are not expressive enough to be used alone in an efficient manner. In this paper we build `CAS2` from `CAS1` and, in fact, build an arbitrary multi-word compare-and-swap (`CASN`). Our design requires only the primitives available on contemporary systems, reserves a small and constant amount of space in each word updated (either 0 or 2 bits) and permits non-overlapping updates to occur concurrently. This provides compelling evidence that current primitives are not only universal in the theoretical sense introduced by Herlihy, but are also universal in their use as foundations for practical algorithms. This provides a straightforward mechanism for deploying many of the interesting non-blocking data structures presented in the literature that have previously required `CAS2`.

1 Introduction

`CASN` is an operation for shared-memory systems that reads the contents of a series of locations, compares these against specified values and, if they all match, updates the locations with a further set of values. All this is performed atomically with respect to other `CASN` operations and specialized reads. The implementation of a non-blocking multi-word compare-and-swap operation has been the focus of

Общий подход

Общий подход

Реализация на JVM

Kotlin: https://github.com/Kotlin/kotlinx.atomicfu
~Java: **new** AtomicReferece(initial)

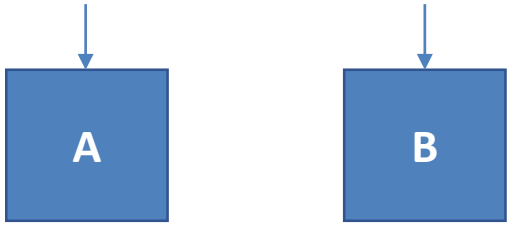
```
class Ref<T>(initial: T) {
    val v = atomic<Any?>(initial)

    var value: T
        get()  {
            return v.value as T /* TODO */
        }

        set(upd) {
            v.value = upd /* TODO */
        }
}
```

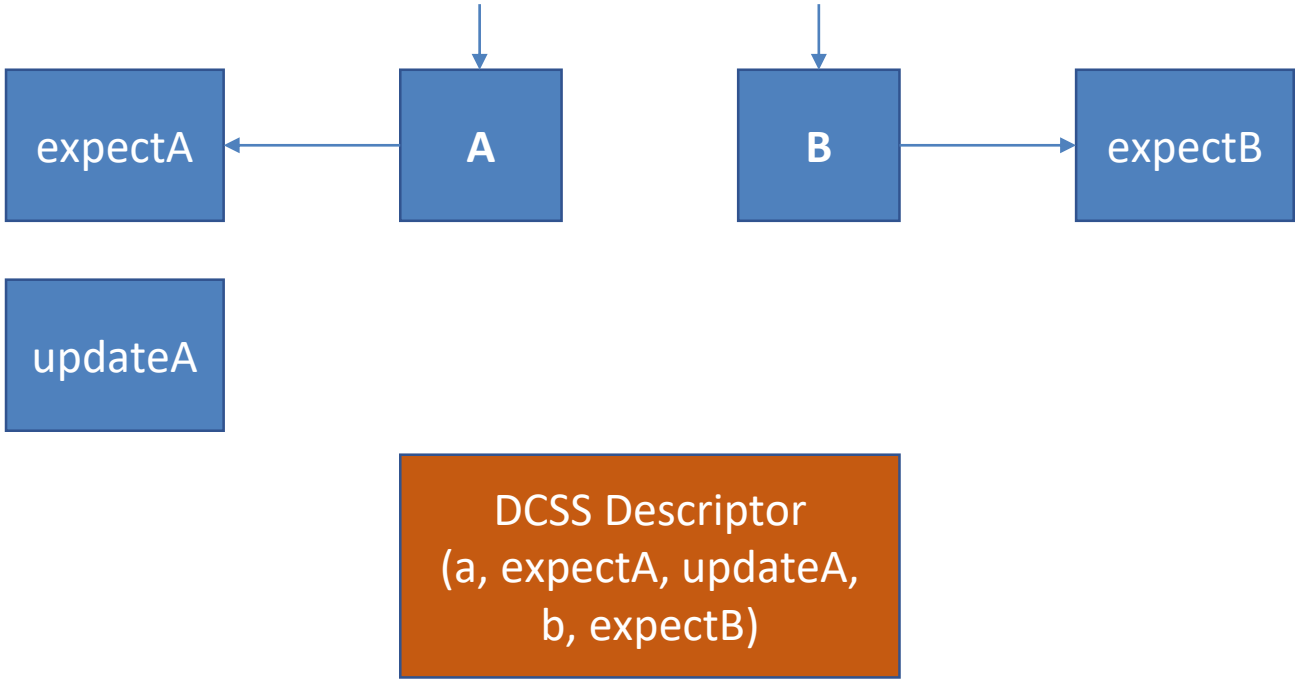
Double-Compare Single-Swap (DCSS)

DCSS в псевдокоде

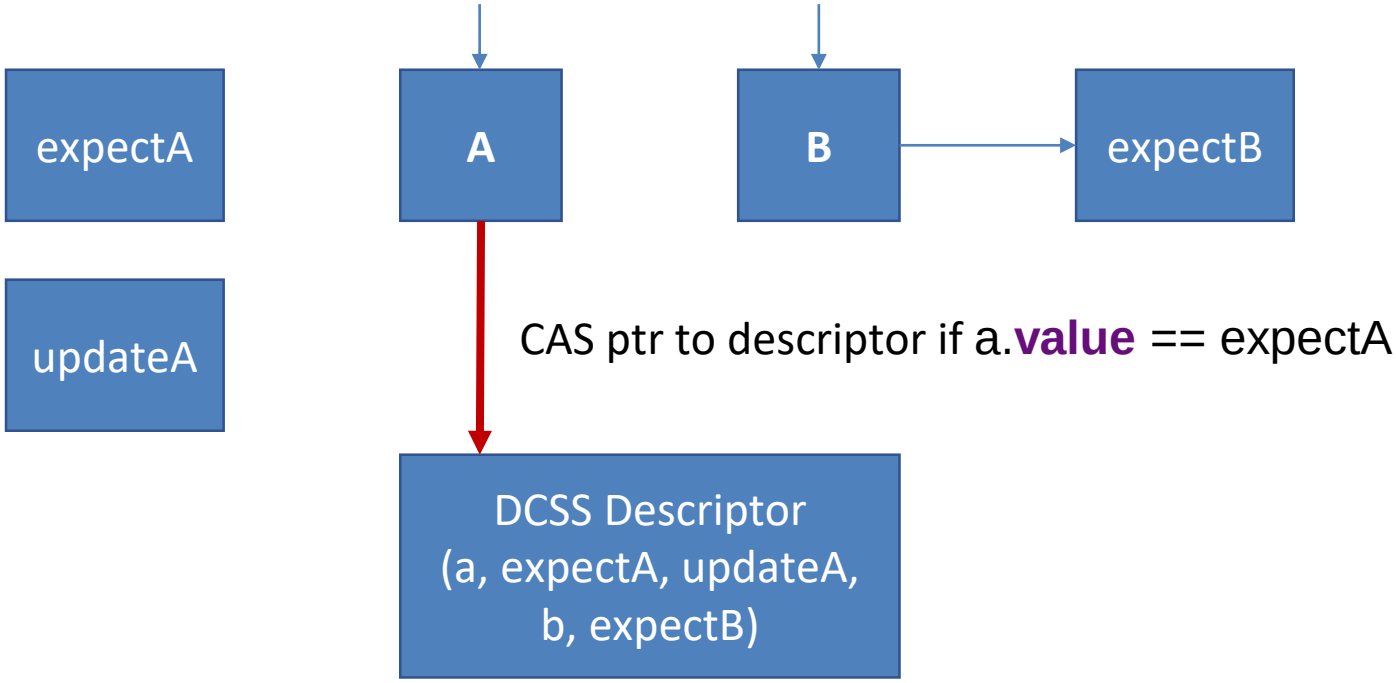



```
fun <A,B> dcss(
1       a: Ref<A>, expectA: A, updateA: A,
2       b: Ref<B>, expectB: B) =
            atomic {
3               if (a.value == expectA && b.value == expectB) {
4                   a.value = updateA
                }
            }
```

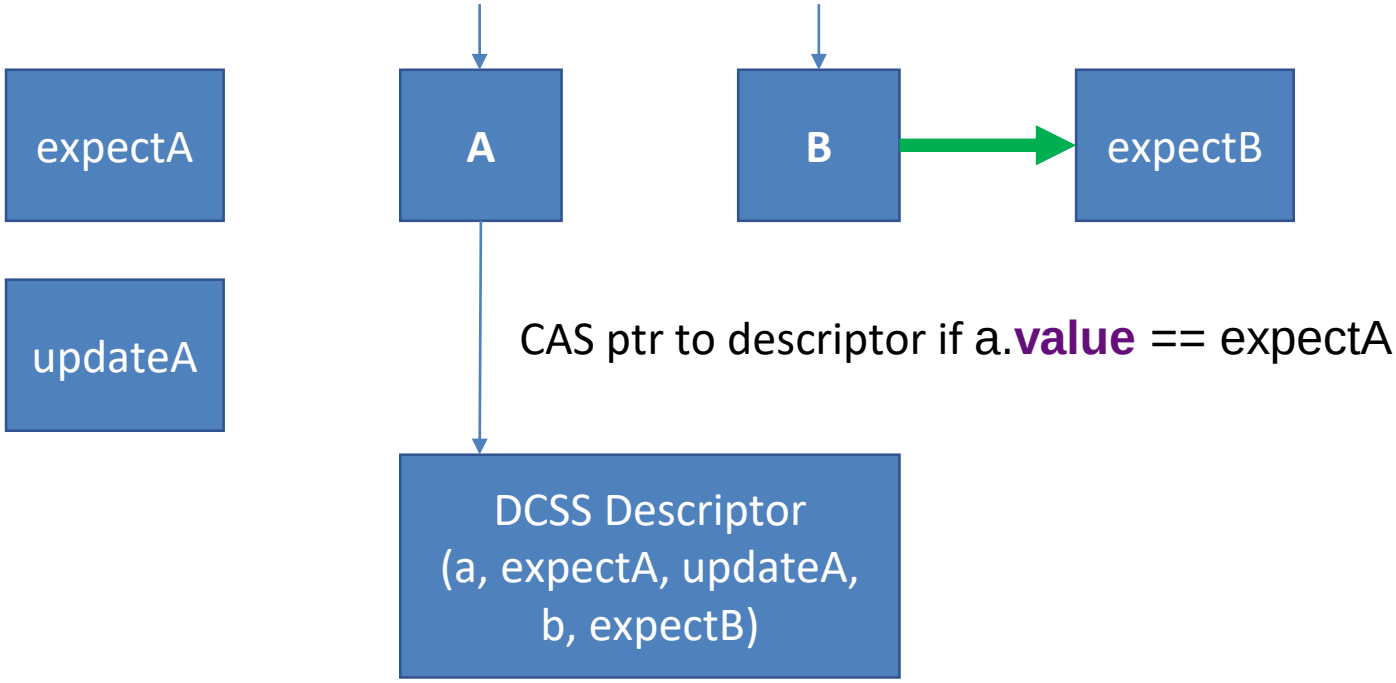
DCSS: init descriptor



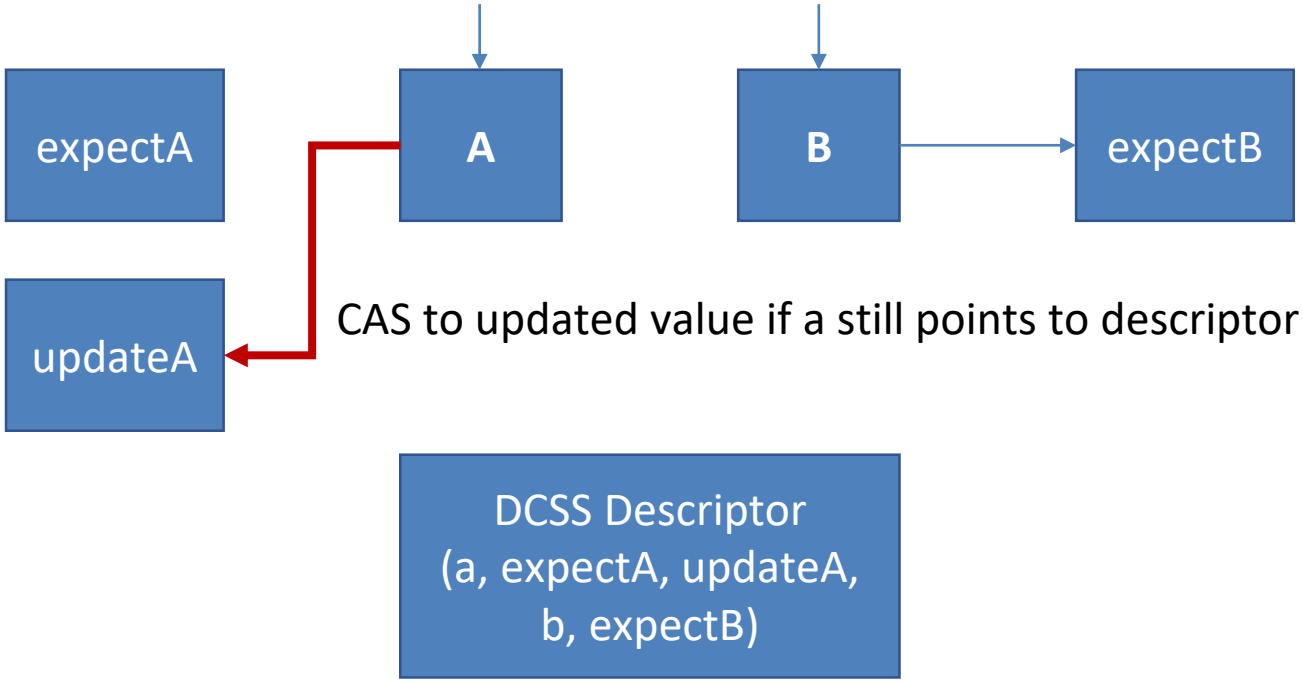
DCSS: prepare



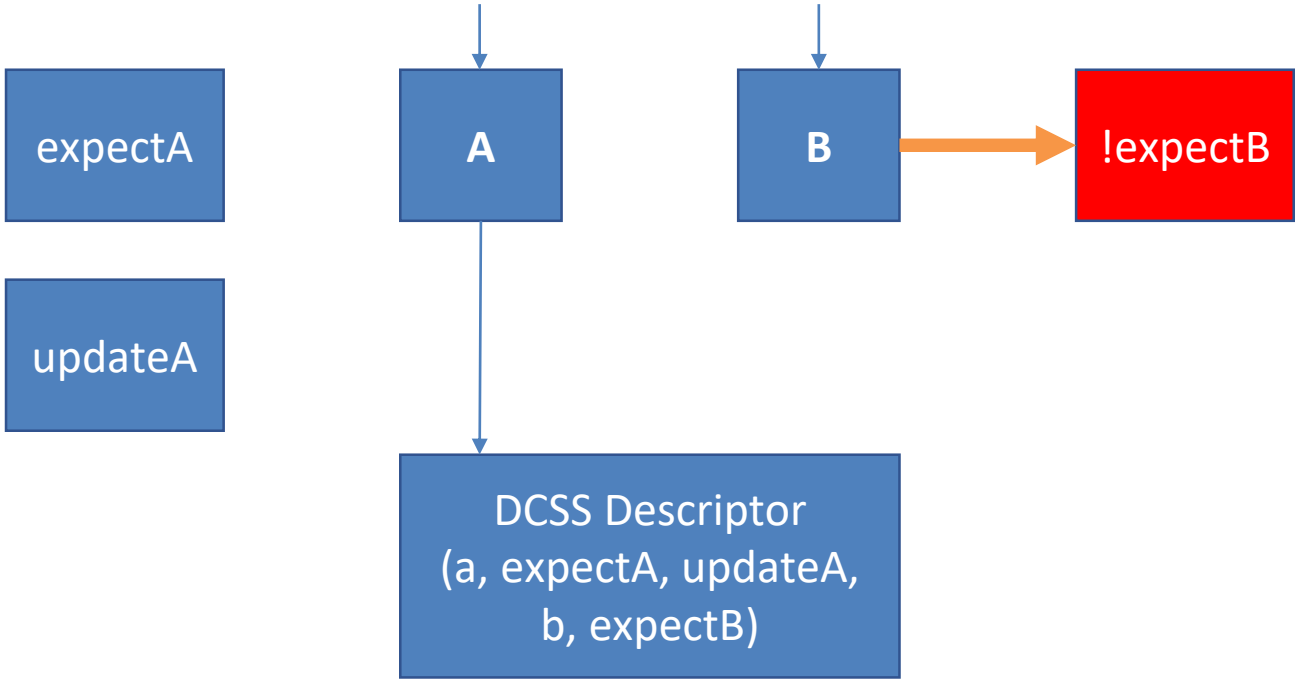
DCSS: read b.value



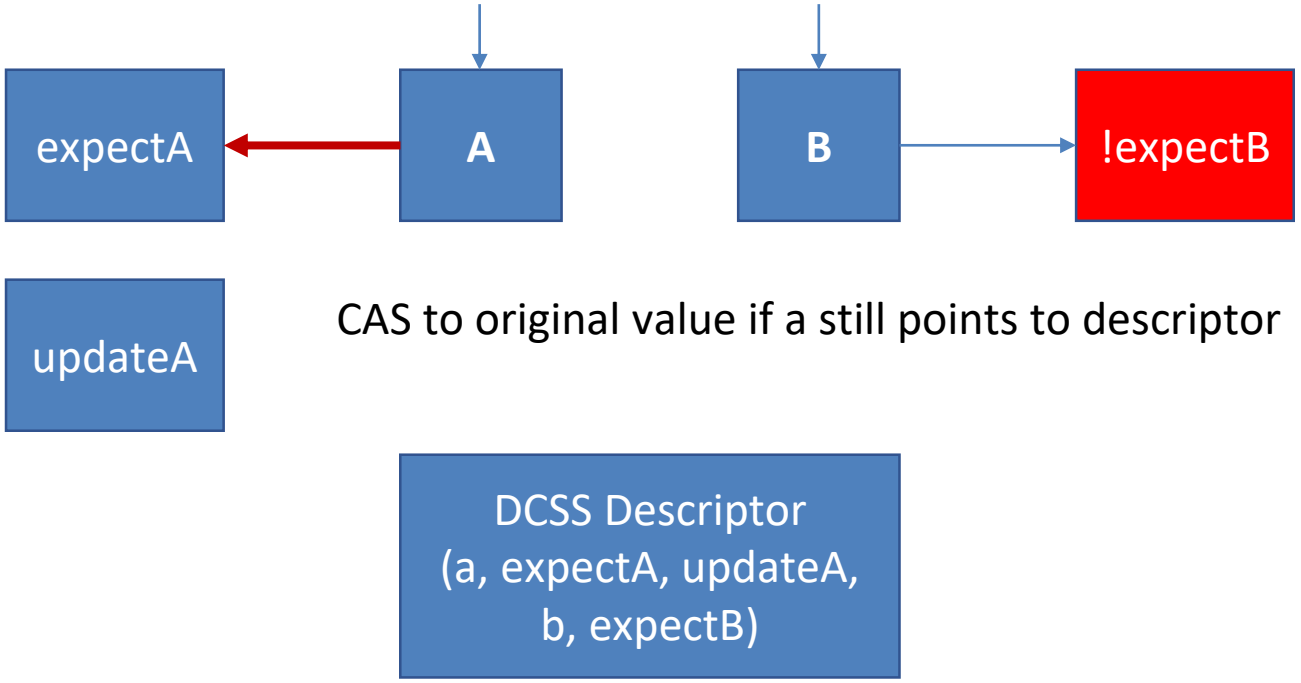
DCSS: complete (when success)



DCSS: complete (alternative)



DCSS: complete (when fail)



DCSS: Состояния

Подход к реализации

```
abstract class Descriptor {
    abstract fun complete()
}
```

Подход к реализации

```
abstract class Descriptor { … }

class Ref<T>(initial: T) {
    var v = atomic<Any?>(initial)

    var value: T
        get()  {
            v.loop { cur ->
                ...
            }
        }

    …
```

```
while (true) {
    val cur = v.value
    ...
}
```

Подход к реализации

```
abstract class Descriptor { … }

class Ref<T>(initial: T) {
    var v = atomic<Any?>(initial)

    var value: T
        get()  {
            v.loop { cur ->
                when(cur) {
                    is Descriptor -> cur.complete()
                    else -> return cur
                }
            }
        }

    …
```

Подход к реализации

```
abstract class Descriptor { … }

class Ref<T>(initial: T) {
    var v = atomic<Any?>(initial)

    var value: T
        set(upd) {
            v.loop { cur ->
                ...
            }
        }
```

Подход к реализации

```
abstract class Descriptor { … }

class Ref<T>(initial: T) {
    var v = atomic<Any?>(initial)

    var value: T
        set(upd) {
            v.loop { cur ->
                when(cur) {
                    is Descriptor -> cur.complete()
                    else -> if (v.compareAndSet(cur, upd))
                        return
                }
            }
        }
```

Подход к реализации

```
class RDCSSDescriptor<A, B>(
    val a: Ref<A>, val expectA: A, val updateA: A,
    val b: Ref<B>, val expectB: B
) : Descriptor() {
    override fun complete() {
        ...
    }
}
```

Подход к реализации

```
class RDCSSDescriptor<A, B>(
    val a: Ref<A>, val expectA: A, val updateA: A,
    val b: Ref<B>, val expectB: B
) : Descriptor() {
    override fun complete() {
        val update = if (b.value === expectB)
                                updateA else expectA
        a.v.compareAndSet(this, update)
    }
}
```

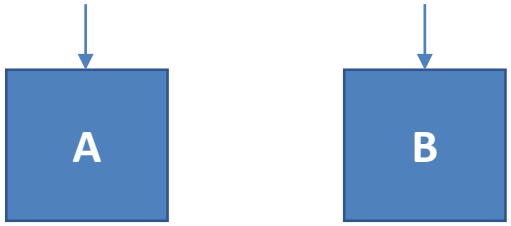
Подход к реализации

```
class RDCSSDescriptor<A, B>(
    val a: Ref<A>, val expectA: A, val updateA: A,
    val b: Ref<B>, val expectB: B
) : Descriptor() {
    override fun complete() {
        val update = if (b.value === expectB)
                                updateA else expectA
        a.v.compareAndSet(this, update)
    }
}
```

Наблюдения и замечания

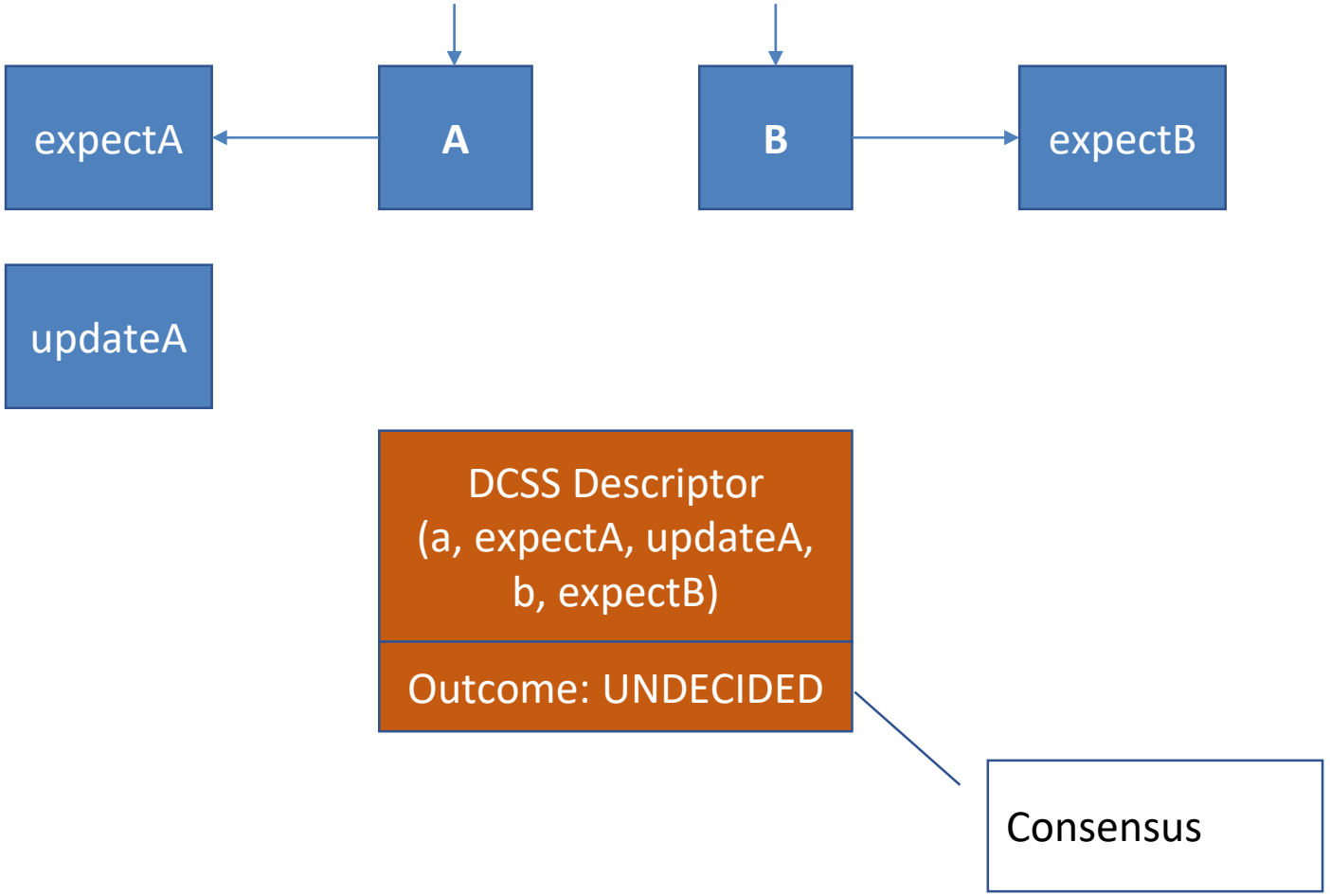
- complete работает без циклов
  - Реализует state machine которая идет только вперед
- Но есть циклы помощи другим потокам
  - Алгоритм lock-free
- Адреса A & B должны быть упорядочены
  - Либо может переполнится стек при помощи
- Один из способов: Restricted DCSS (RDCSS)
  - A и B берутся из разных “регионов” (не пересекающихся)

DCSS Mod: Узнаем ответ

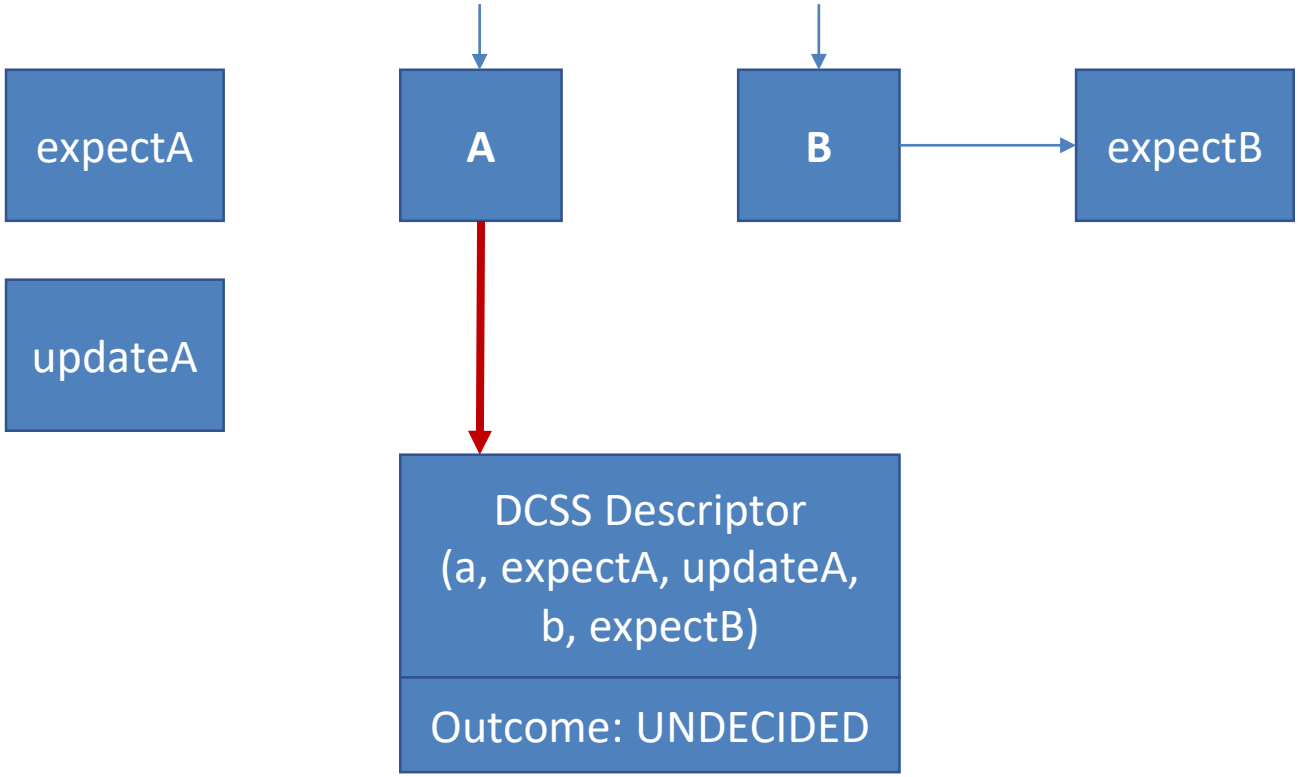



```
fun <A,B> dcssMod(
    a: Ref<A>, expectA: A, updateA: A,
    b: Ref<B>, expectB: B): Boolean =
        atomic {
            if (a.value == expectA && b.value == expectB) {
                a.value = updateA
                true
            } else
                false
        }
```

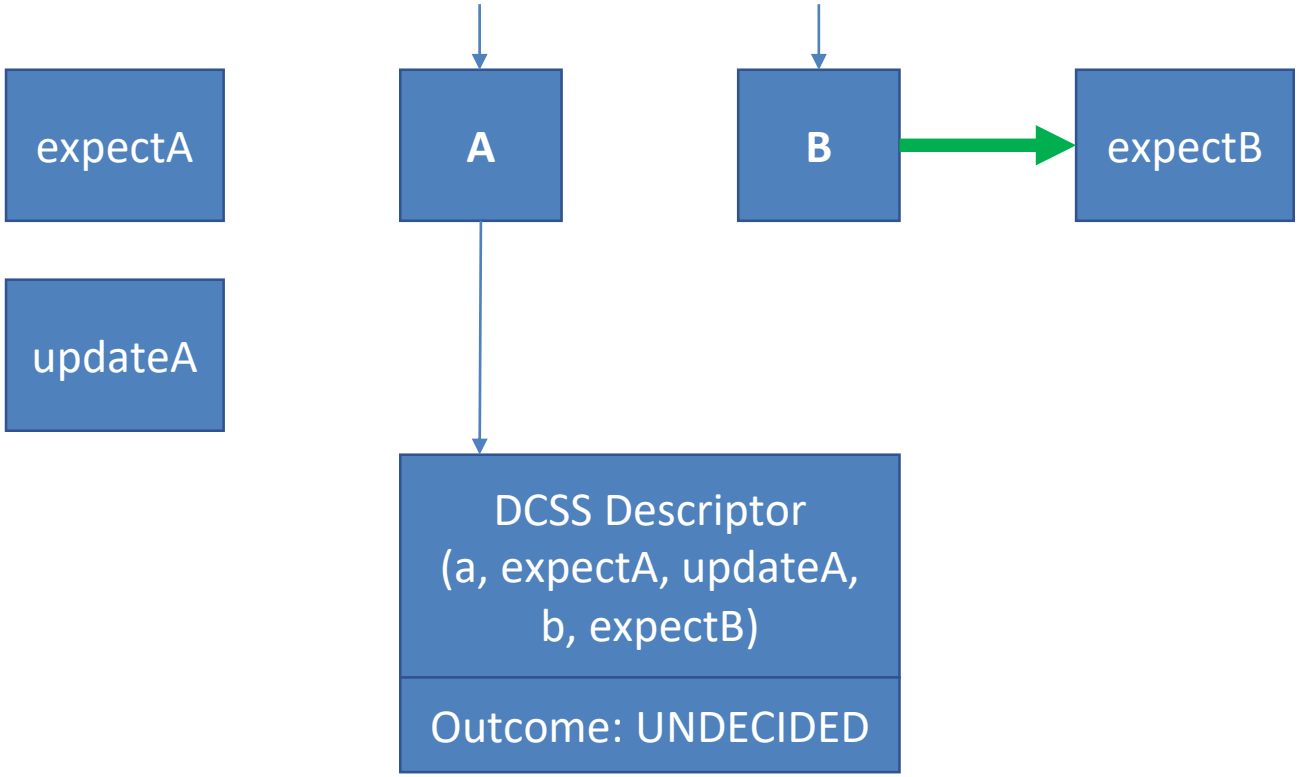
DCSS Mod: init descriptor



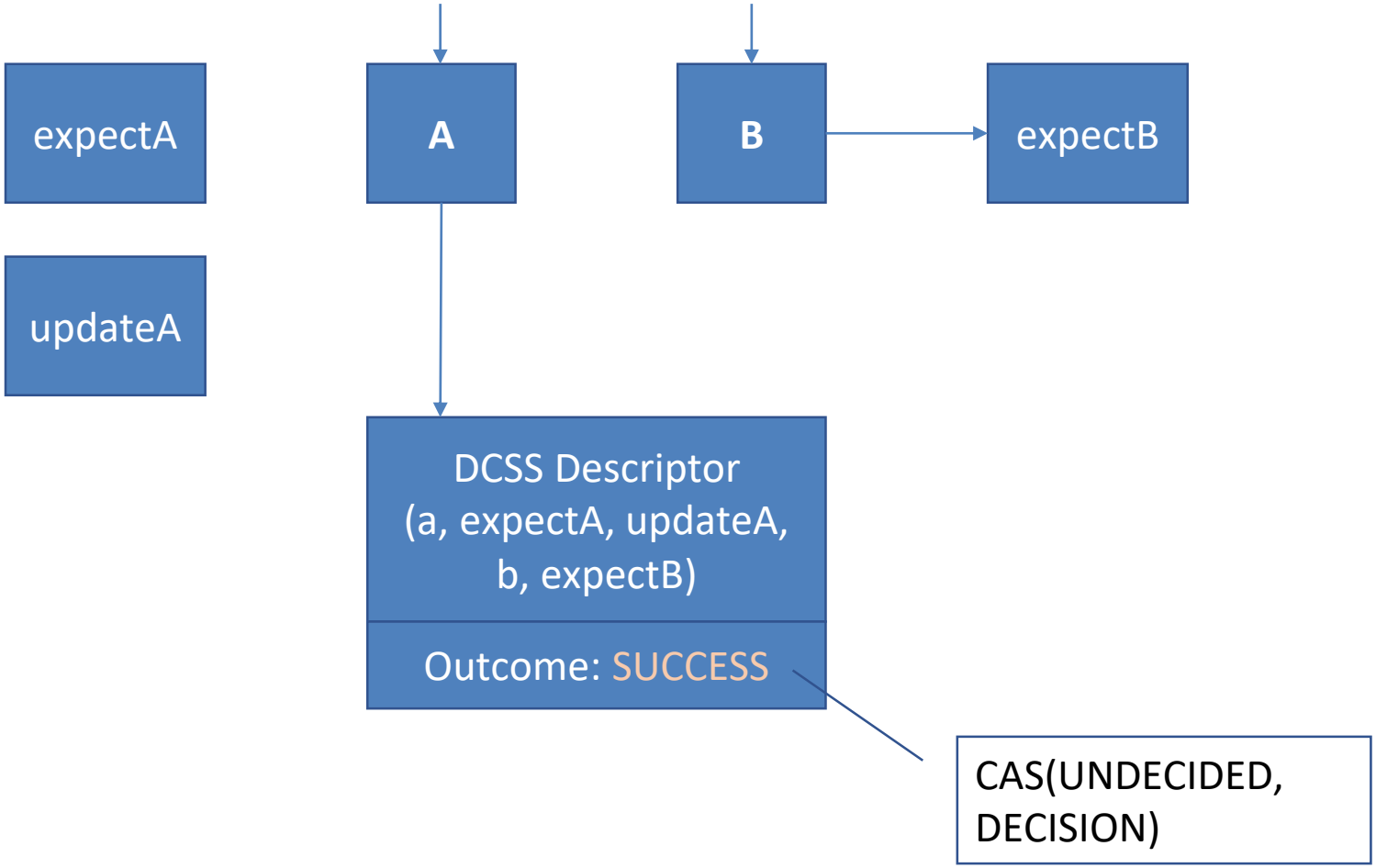
DCSS Mod: prepare



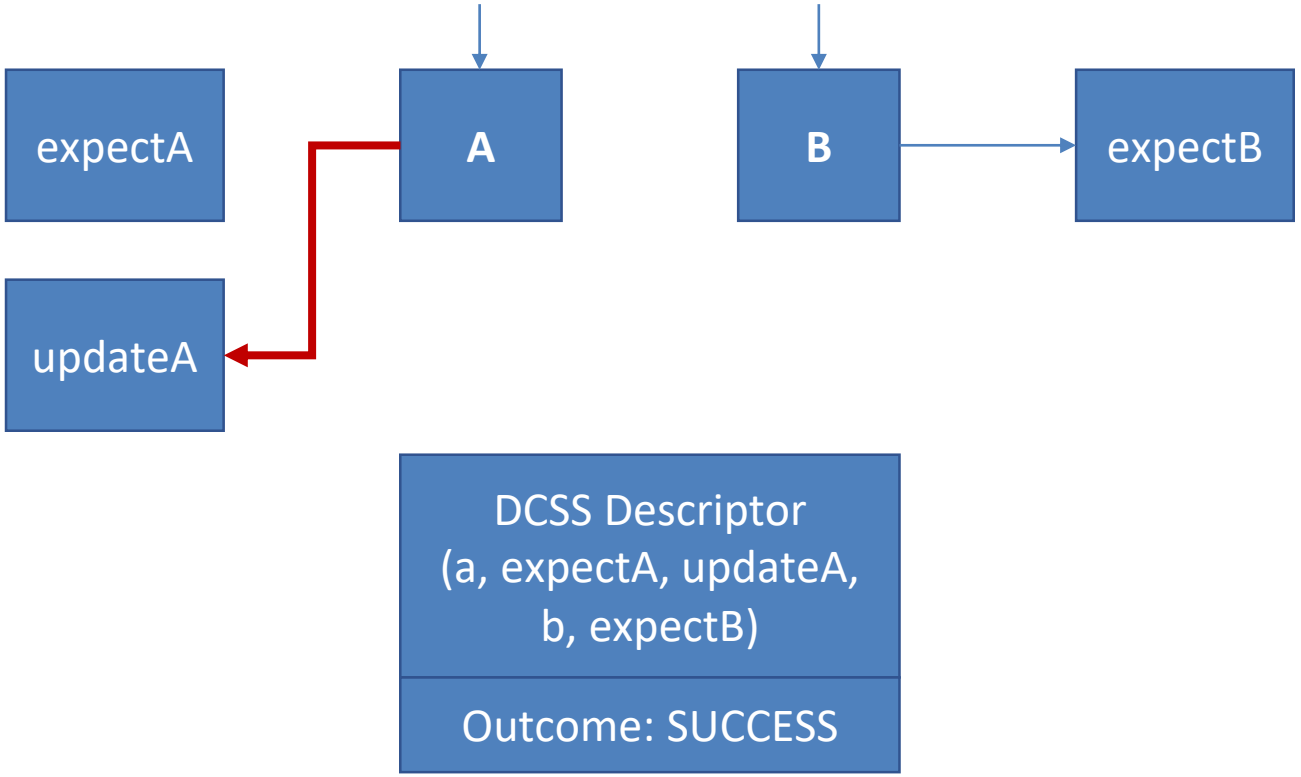
DCSS Mod: read b.value



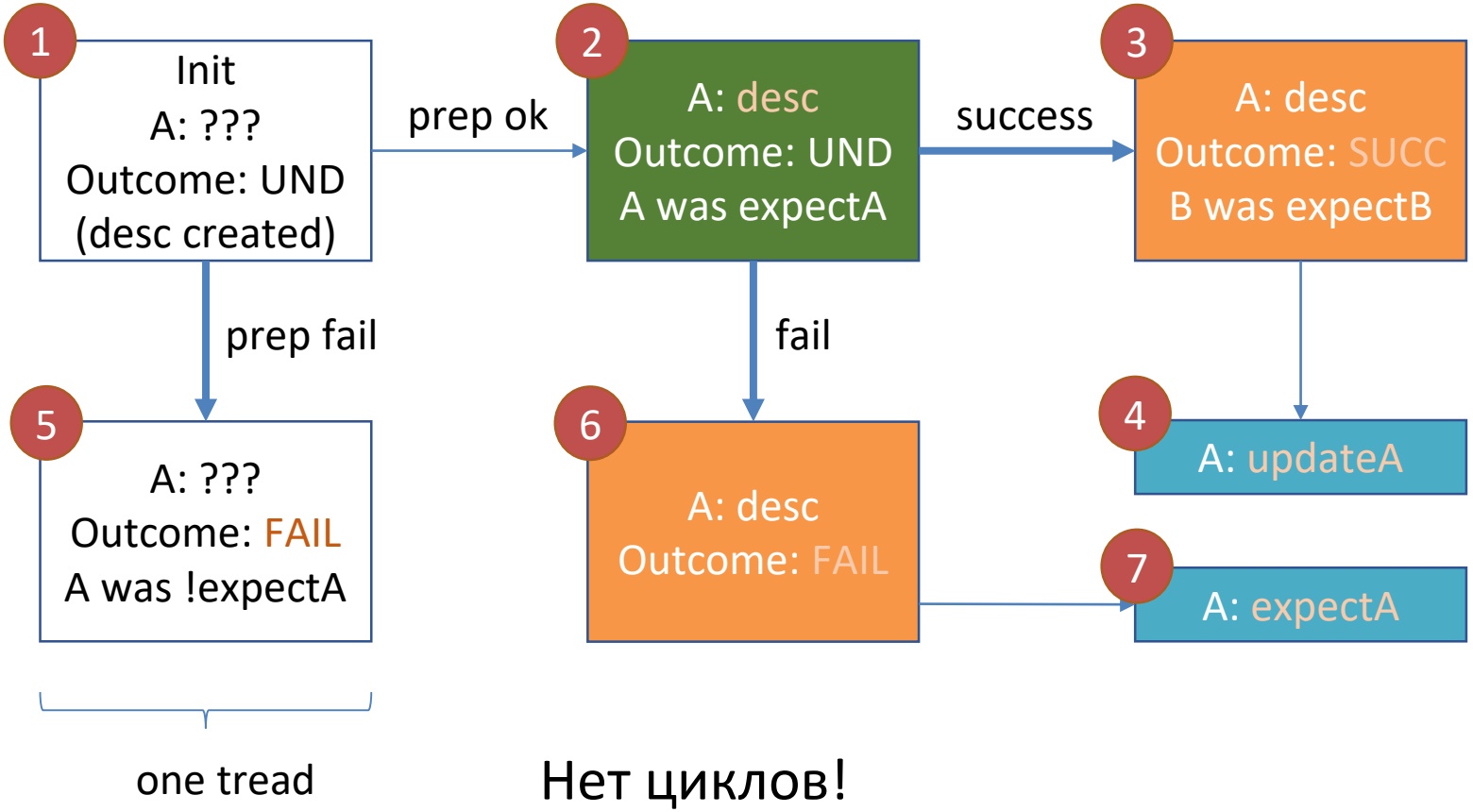
DCSS Mod: reach consensus



DCSS Mod: complete

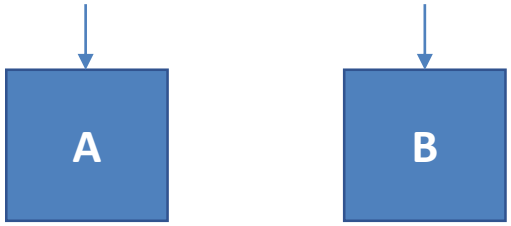


DCSS Mod: Состояния



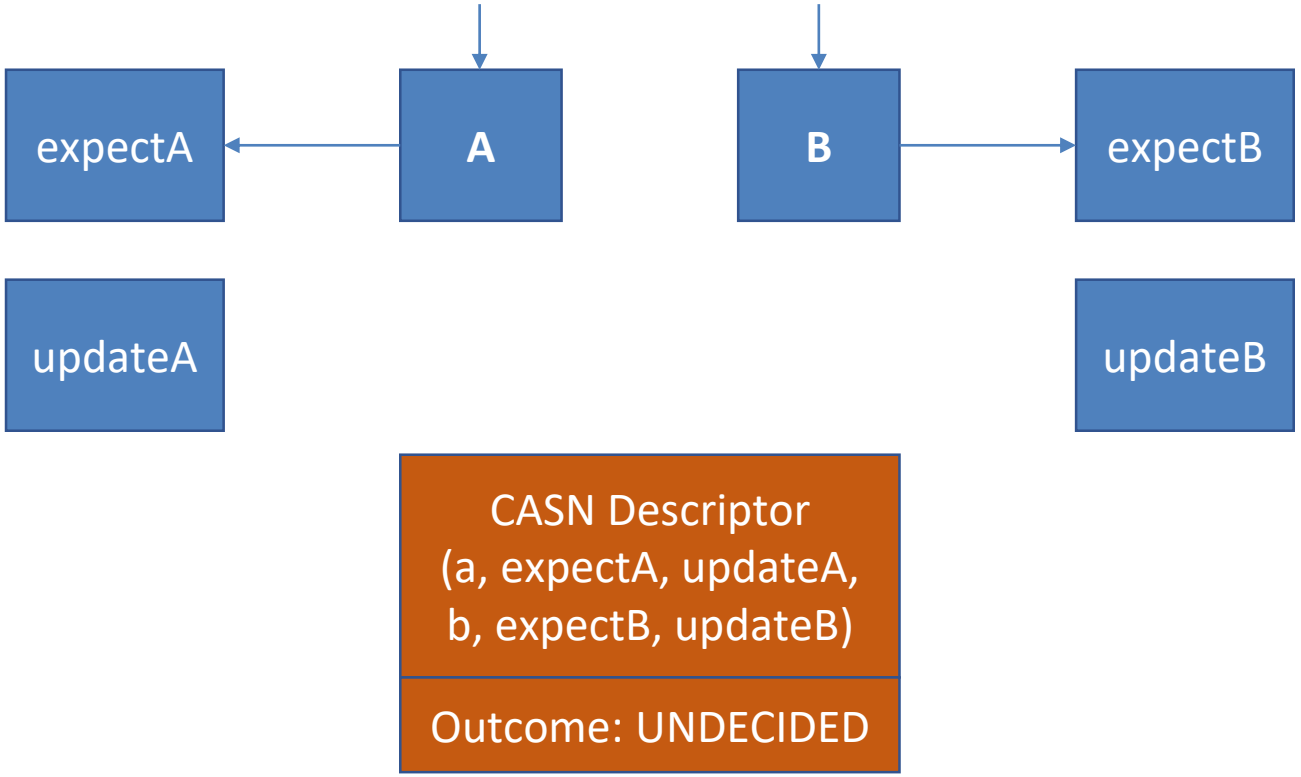
Compare-And-Swap N-words (CASN)

CAS2 в псевдокоде

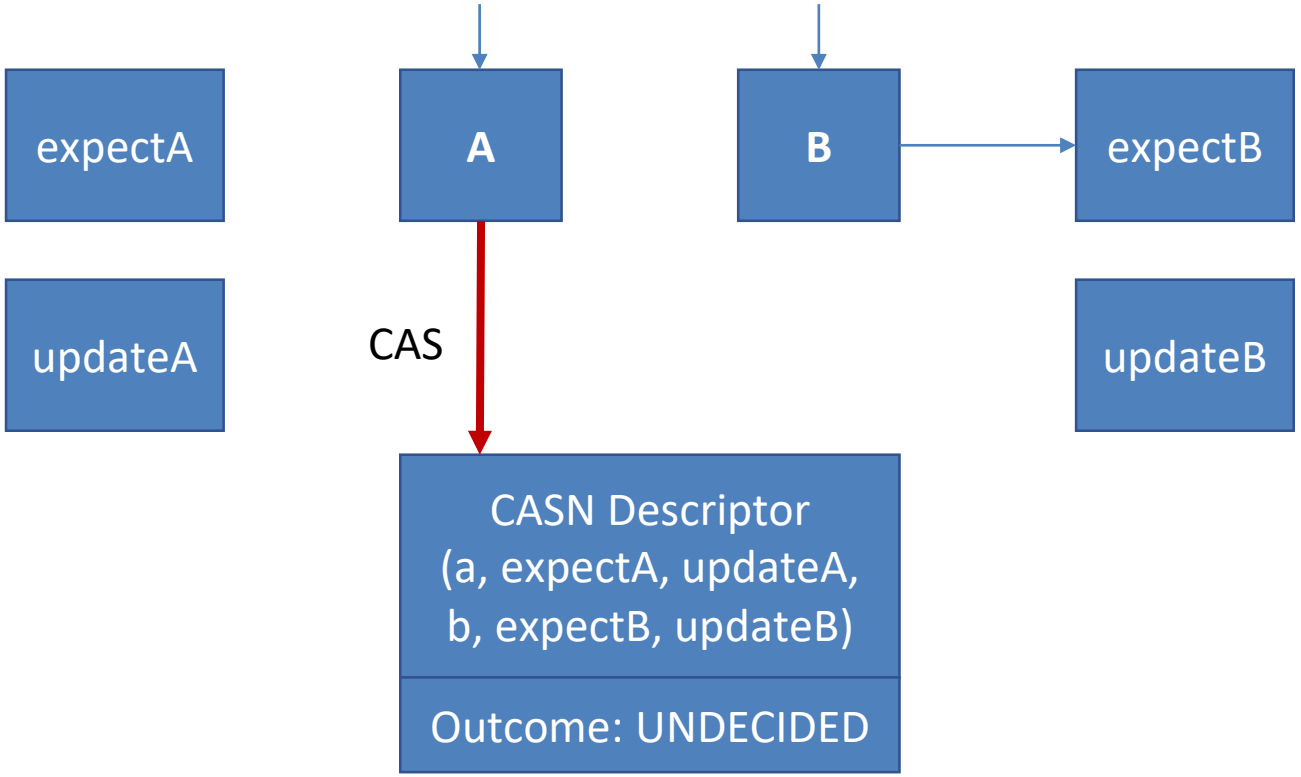



```
fun <A,B> cas2(
1       a: Ref<A>, expectA: A, updateA: A,
2       b: Ref<B>, expectB: B, updateB: B): Boolean =
            atomic {
3               if (a.value == expectA && b.value == expectB) {
                    a.value = updateA
                    b.value = updateB
4                   true
                } else
5                   false
            }
```

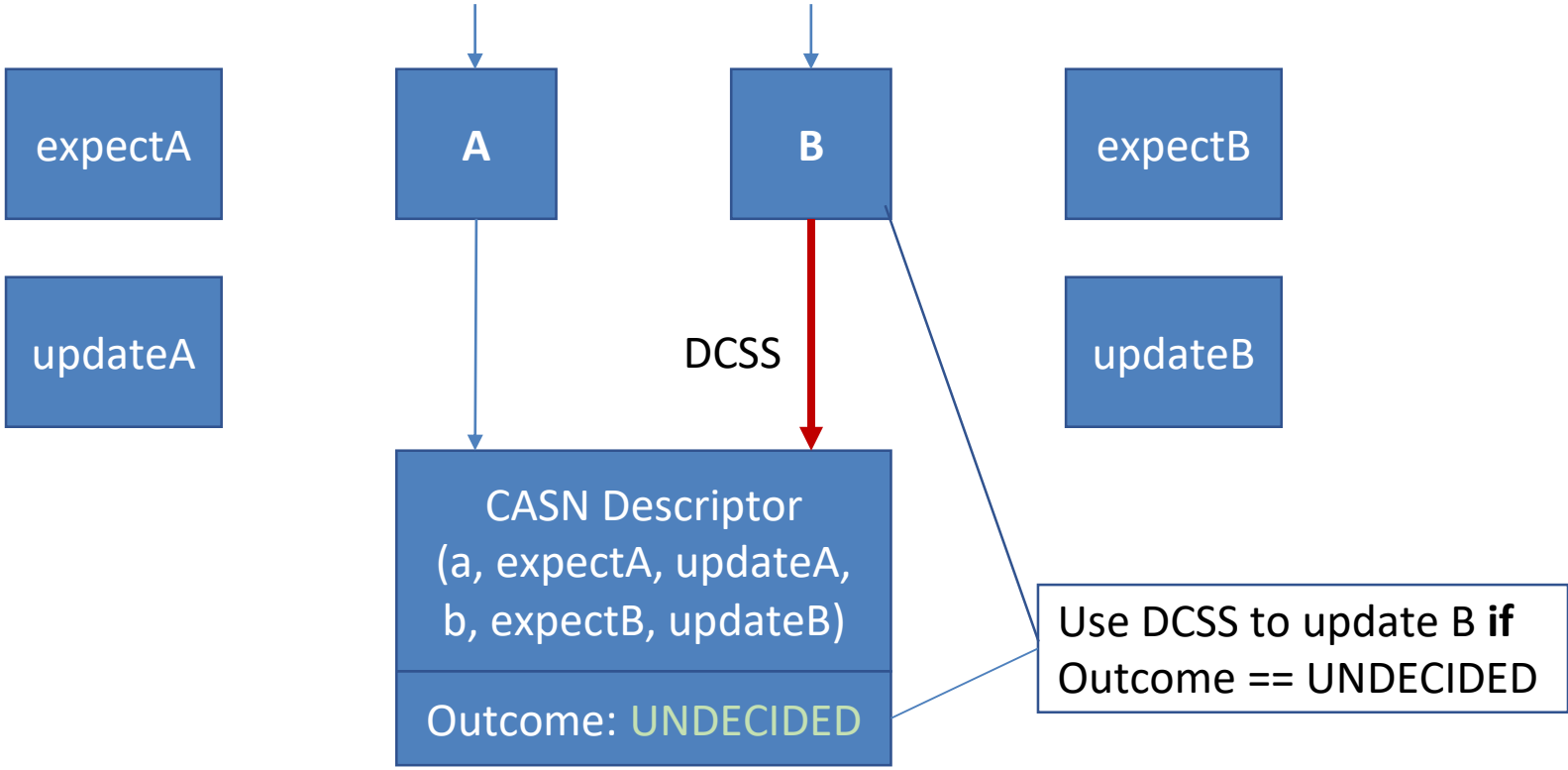
CASN: init descriptor



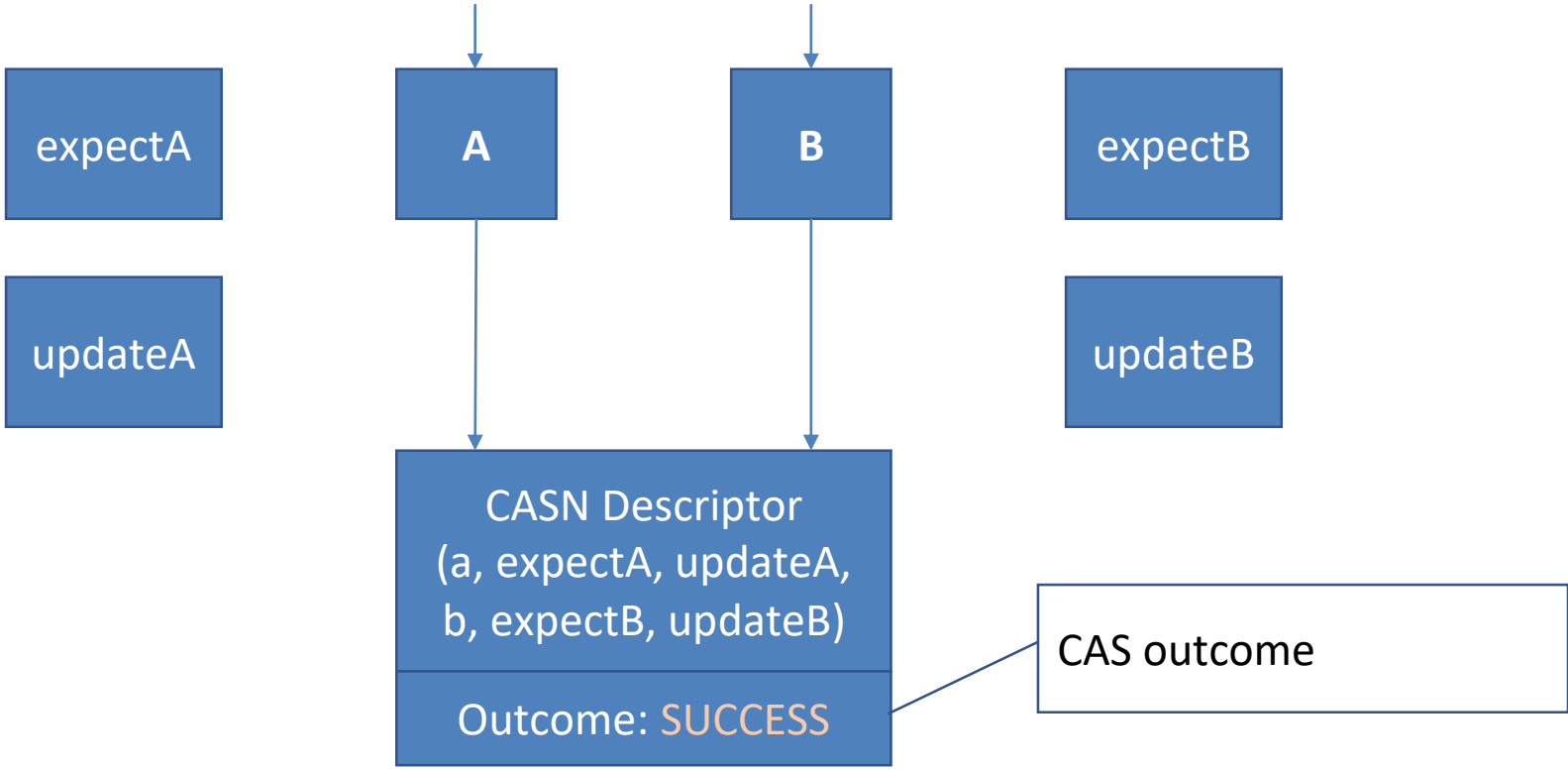
CASN: prepare (1)



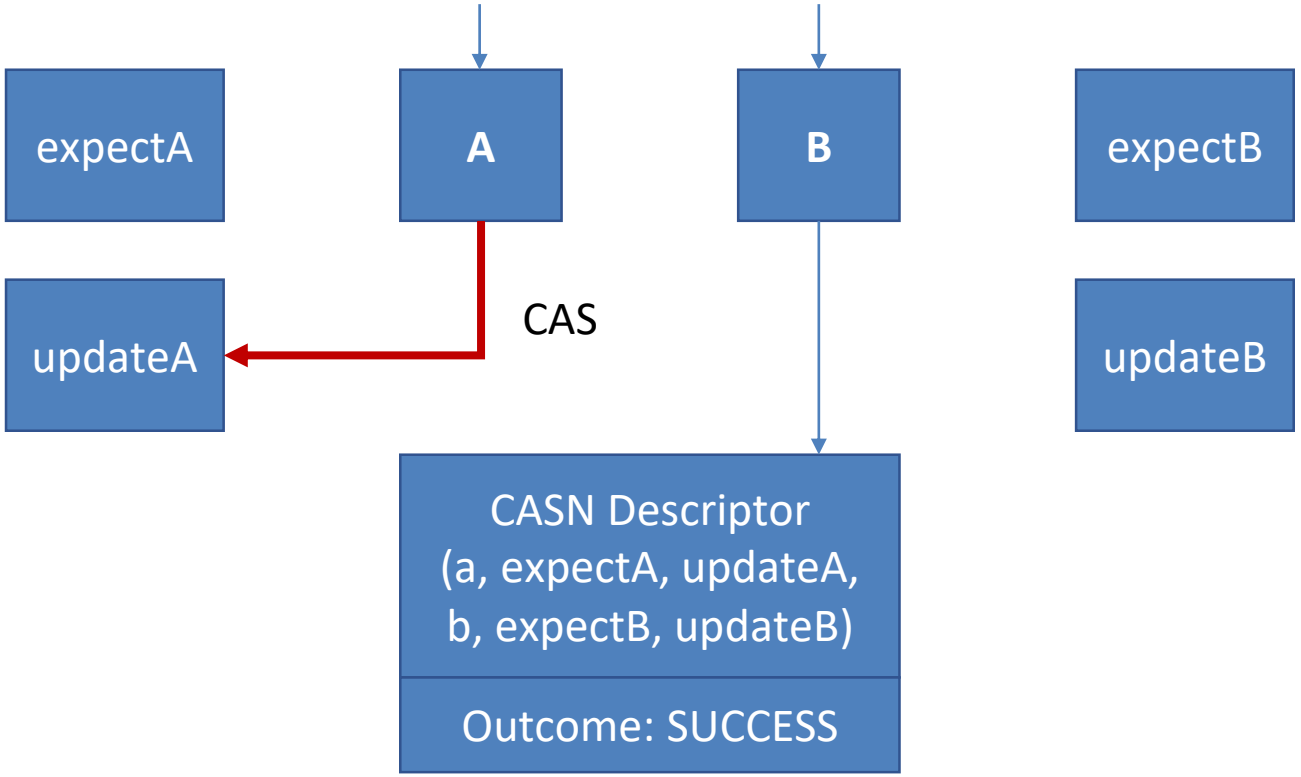
CASN: prepare (2)



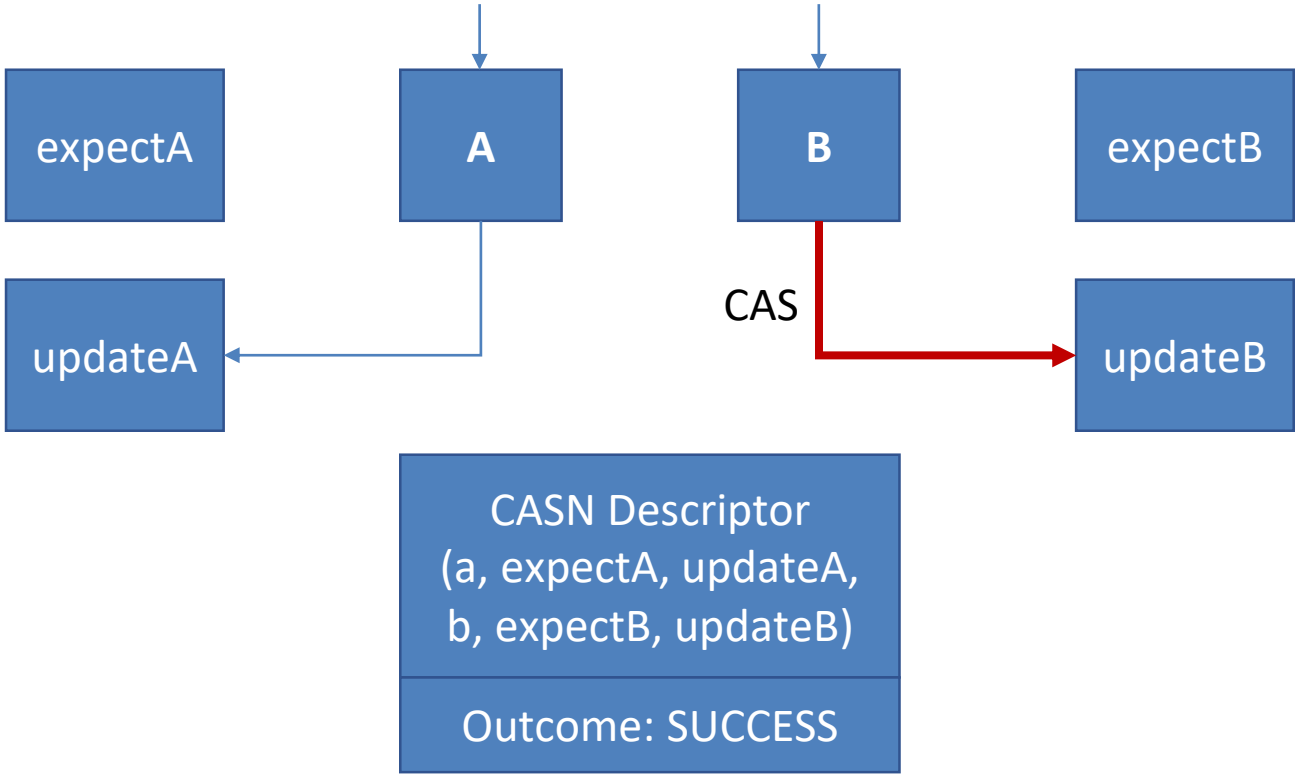
CASN: decide


expectA
A
B
expectB
updateA
updateB
CASN Descriptor
(a, expectA, updateA,
b, expectB, updateB)
Outcome: SUCCESS
CAS outcome

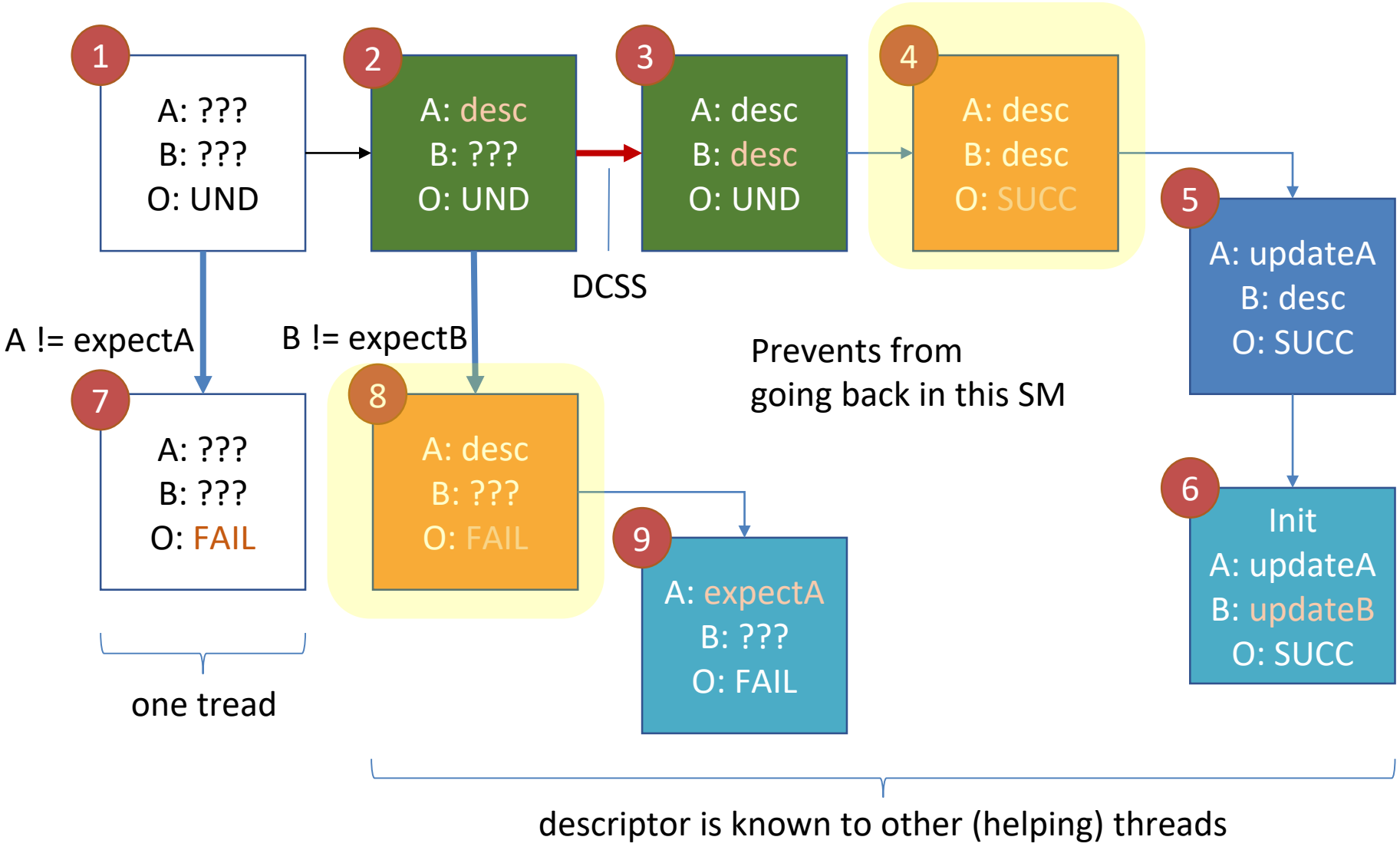
CASN: complete (1)



CASN: complete (2)

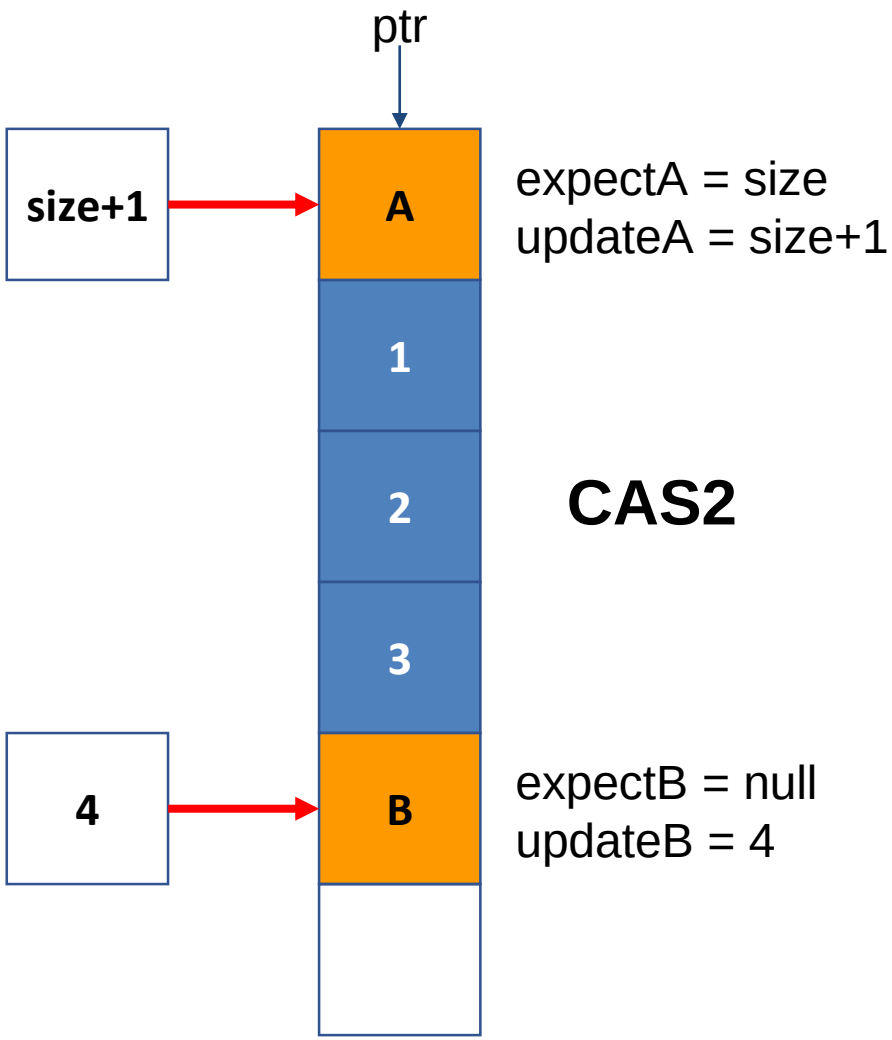


CASN: Состояния



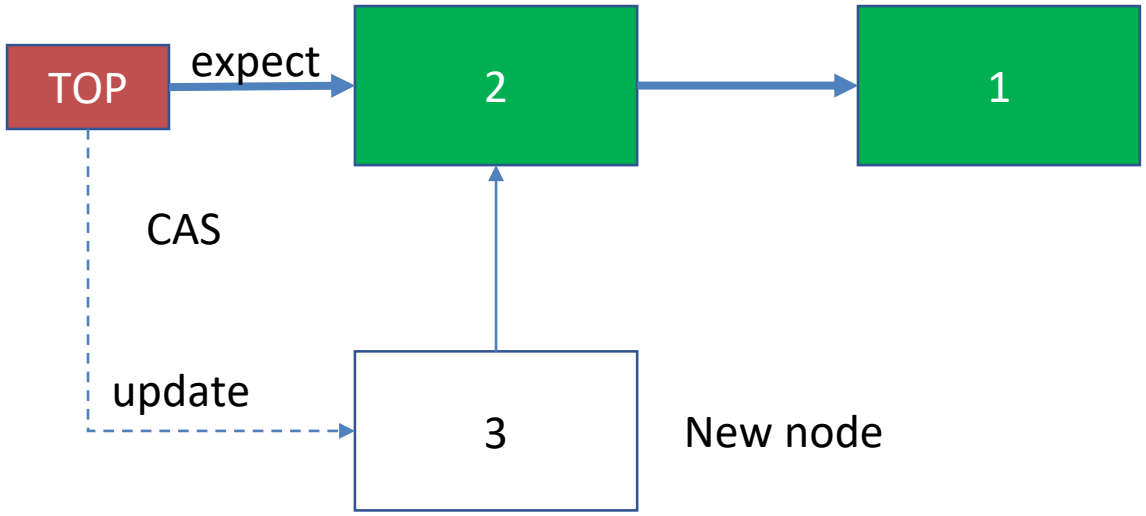
Применяем Раз

Стек на массиве



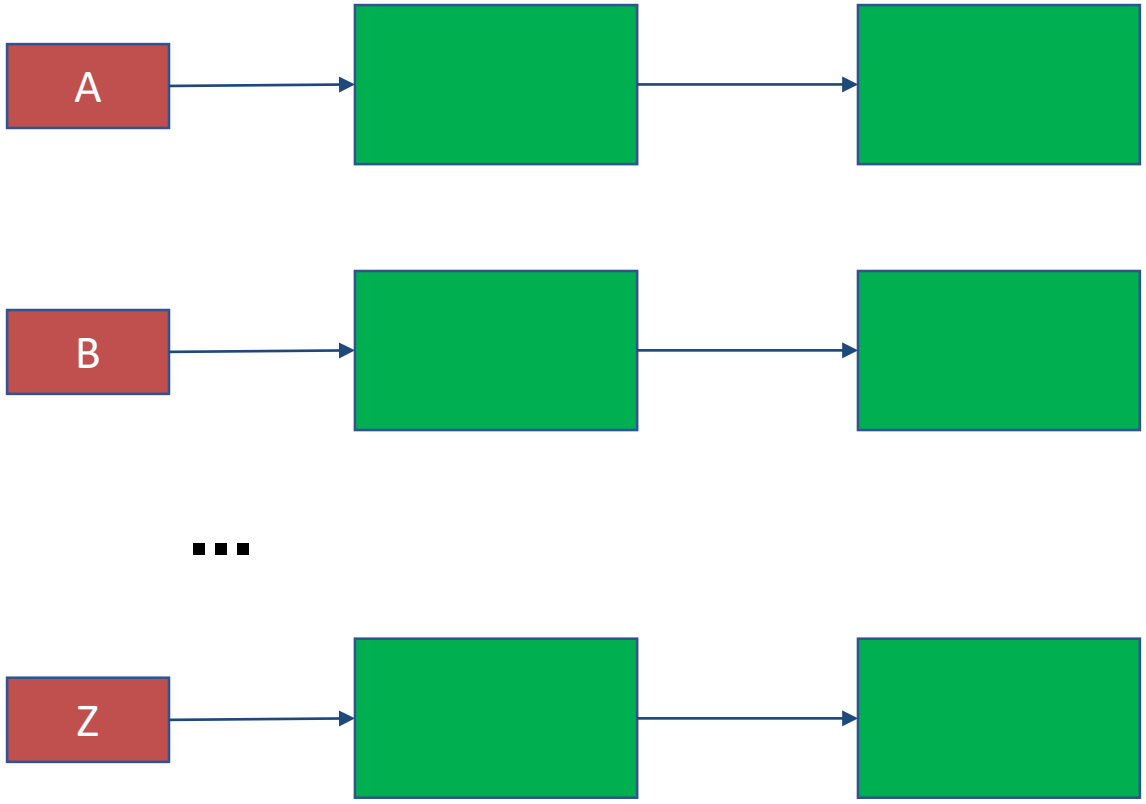
Применяем Два

Стек на списках



Можем узнать все параметры для CAS (ref, expect, update) до выполнения операции

Много стеков



Можем атомарно сделать набор операций над различными стеками