Модели консистентности для конкурентных объектов
Линеаризуемость

Что такое очередь?

Последовательность элементов с двумя операциями:

- `Q.enq(x)` – Добавить в конец последовательности элемент x.
- `Q.deq()` – Извлечь и вернуть первый элемент в последовательности. Если в последовательности нет элементов, то вернуть 0.

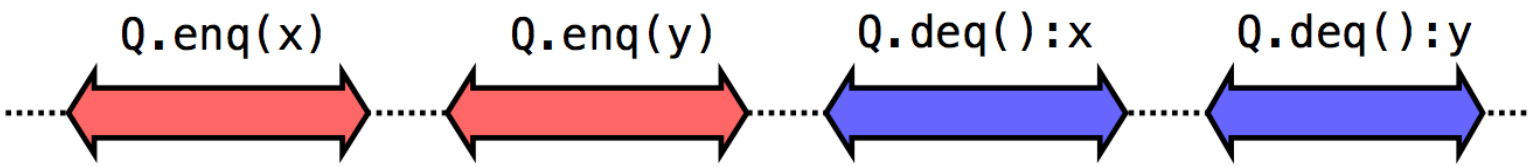
FIFO-семантика – элементы извлекаются в порядке их добавления в очередь

Что такое очередь?

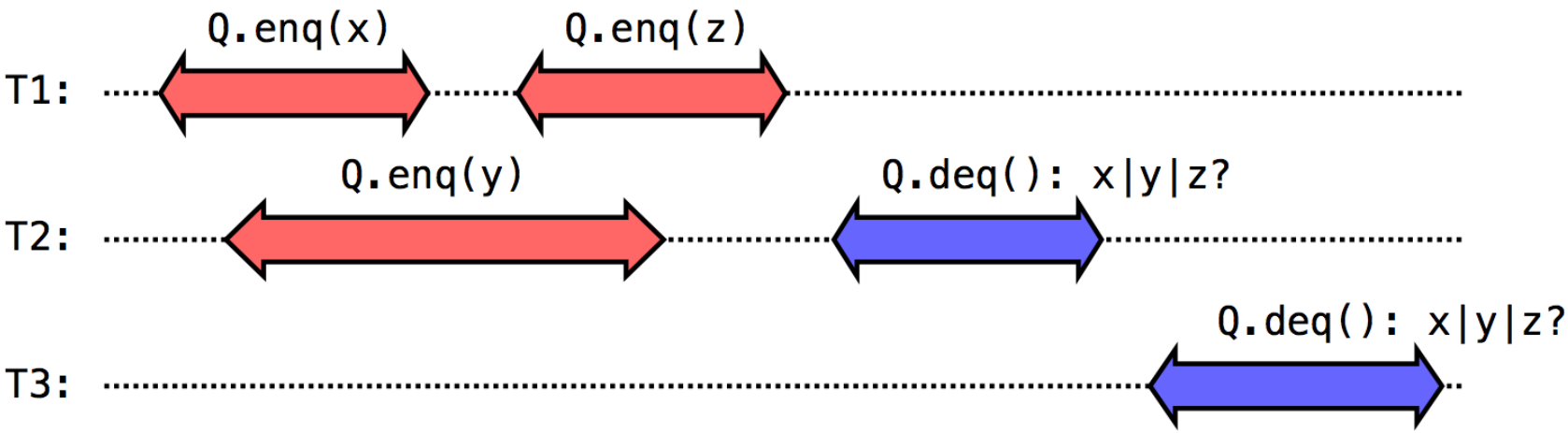
Последовательность элементов с двумя операциями:

- `Q.enq(x)` – Добавить в конец последовательности элемент x.
- `Q.deq()` – Извлечь и вернуть первый элемент в последовательности. Если в последовательности нет элементов, то вернуть 0.

Это **последовательная спецификация** – для любой последовательности вызовов мы знаем, что вернет каждый из них.



А что такое конкурентная очередь?

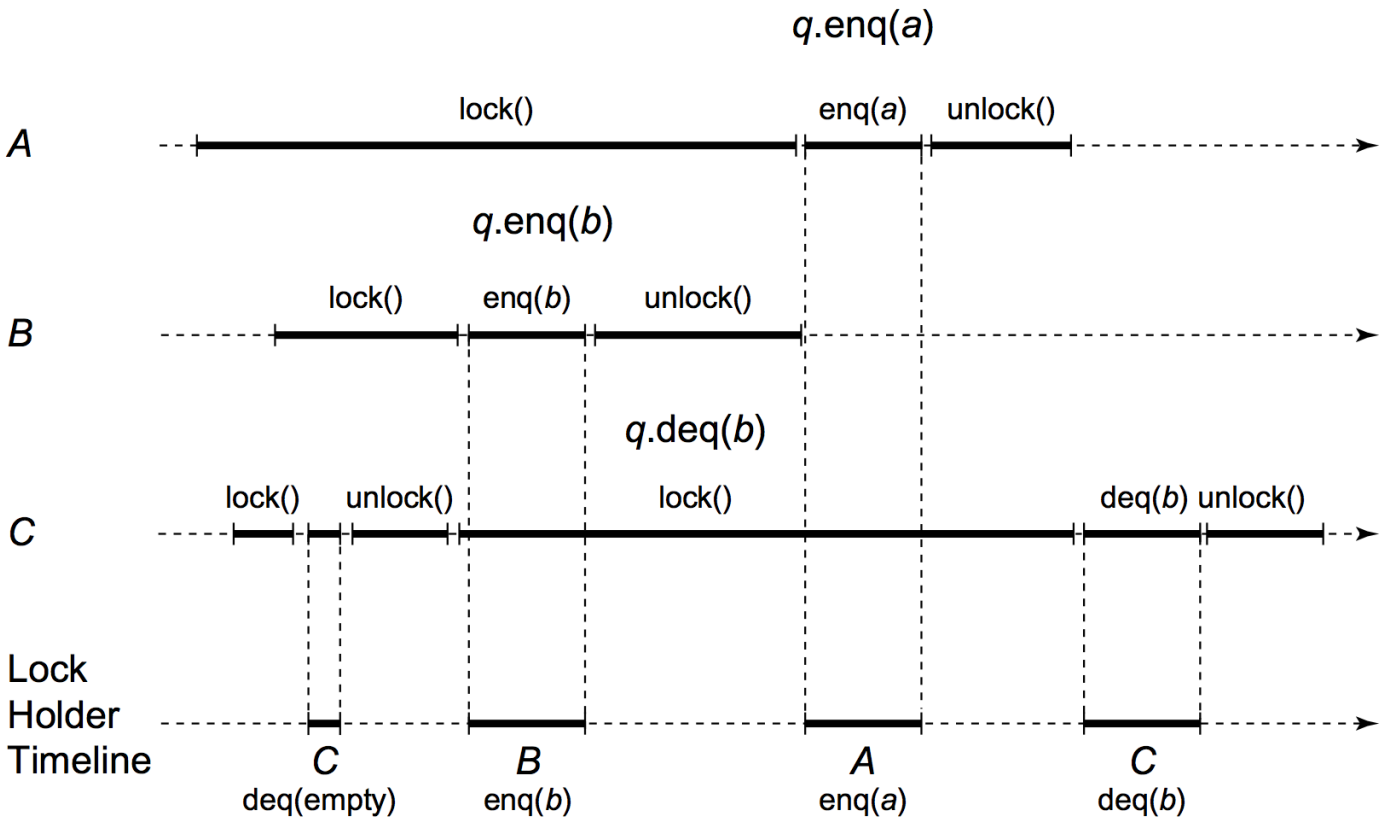


Какие значения могут вернуть вызовы `Q.deq()`, а какие – не могут?

Как вообще определить семантику очереди в этом случае? Что такое FIFO в случае конкурентных операций?

Большой мьютекс

Если конкурентный объект защищен одним мьютексом, то все операции над ним будут сериализованы через критические секции:



Неблокирующая синхронизация

Мы отказываемся от сериализации доступа с помощью мьютексов и спинлоков и позволяем потокам конкурентно работать с внутренностями структуры данных:

Проблемы: утилизация памяти, ABA. Все **очень сложно!**

Sequential Consistency

Последовательная согласованность Лесли Лампорта:

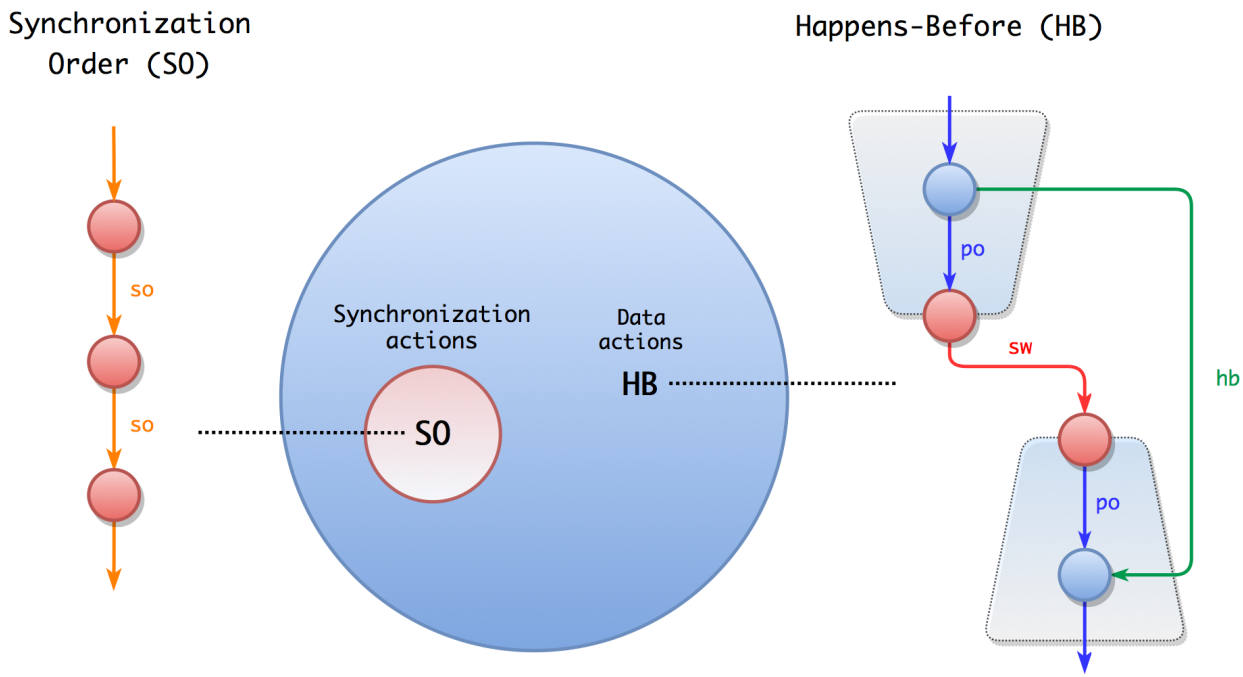
> Результат любого исполнения программы **такой же**, как при некотором последовательном исполнении операций, в котором операции каждого потока исполнялись бы в соответствии с их порядком в программе.

Можно исключить из рассуждений физическую параллельность и анализировать исполнение в модели чередования операций!

Sequential Consistency для памяти

Мы обсуждали, как реализовать такую модель **на уровне ячеек памяти**:

Sequential Consistency for Data-Race-Free Programs



Для высокоуровневых объектов

Не хотим думать о конкурентных исполнениях!

Хотим, чтобы каждому конкурентному исполнению операций над объектом можно было поставить в соответствие эквивалентное исполнение, в котором:

- Все операции происходят последовательно
- Порядок операций согласован с порядком действий каждого потока в исходном исполнении

Напрямую из последовательной согласованности на уровне ячеек памяти это не следует!

Модель исполнения

Конкурентные исполнения будем моделировать с помощью **историй**:

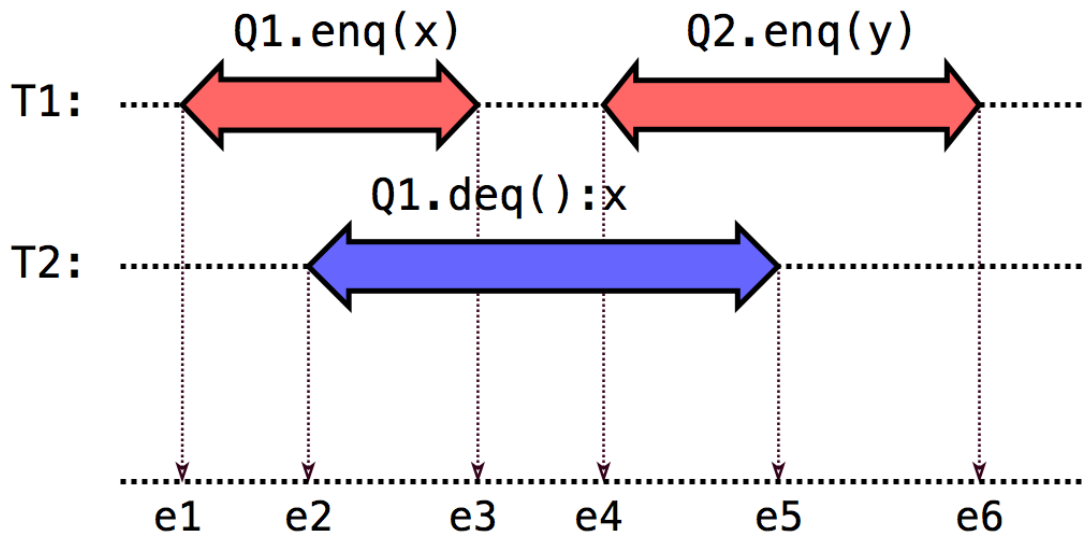
История (**history**) – это последовательность **событий** над одним или несколькими конкурентными объектами.

Событие (**event**) – это либо **вызов** операции (**invocation**), либо **возврат из вызова** (**response**)

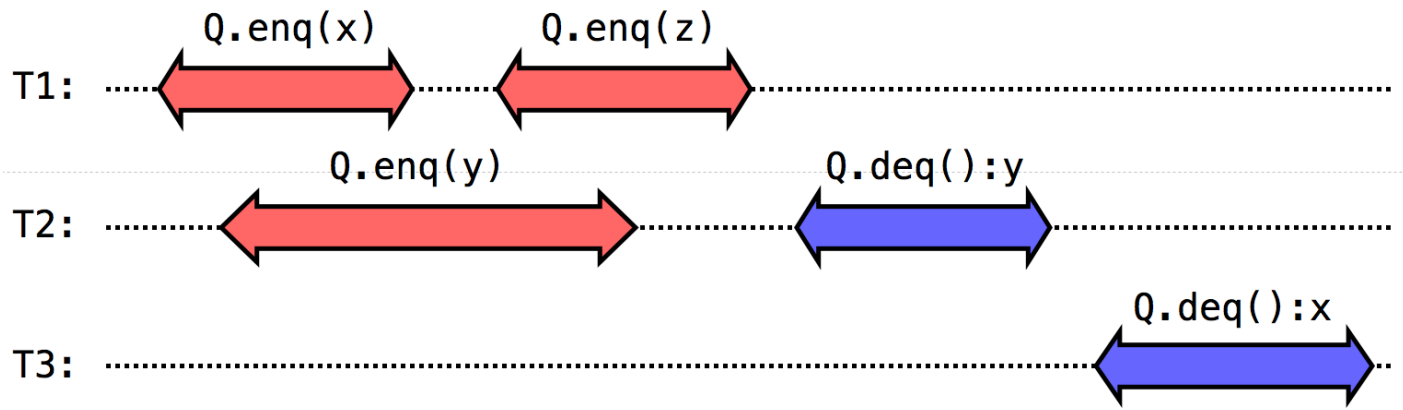
- `T: inv[Q.enq(x)]` – поток `T` начал добавлять в очередь `Q` элемент `x`
- `T': res[Q.deq():y]` – поток `T'` завершил извлечение элемента из очереди `Q`, был извлечен элемент `y`

Пример истории

```
e1  T1: inv[Q1.enq(x)]
e2  T2: inv[Q1.deq()]
e3  T1: res[Q1.enq(x):void]
e4  T1: inv[Q2.enq(y)]
e5  T2: res[Q1.deq():x]
e6  T1: res[Q2.enq(y):void]
```



Пример истории



Проекции (подыстории)

Проекция истории H на поток T – $H|T$ – все события, которые происходят в потоке T

$H|T_1$

```
e1  T1: inv[Q1.enq(x)]
e3  T1: res[Q1.enq(x):void]
e4  T1: inv[Q2.enq(y)]
e6  T1: res[Q2.enq(y):void]
```

Проекция истории H на объект x – $H|x$ – это все события над объектом x.

$H|Q_1$

```
e1  T1: inv[Q1.enq(x)]
e2  T2: inv[Q1.deq()]
e3  T1: res[Q1.enq(x):void]
e5  T2: res[Q1.deq():x]
```

Парные события в истории

Очевидным образом можно определить **парные** события – вызов и соответствующий ему возврат из вызова внутри одного потока.

```
e1  T1: inv[Q1.enq(x)]
e2  T2: inv[Q1.deq()]
e3  T1: res[Q1.enq(x):void]
e4  T1: inv[Q2.enq(y)]
e5  T2: res[Q1.deq():x]
e6  T1: res[Q2.enq(y):void]
```

Последовательные и конкурентные истории

Назовем историю H **последовательной** (**sequential**), если первое событие в ней – вызов метода, и за каждым вызовом сразу же идет парный возврат из вызова.

$e1$ `T1: inv[Q1.enq(x)]`
$e2$ `T1: res[Q1.enq(x):void]`
$e3$ `T2: inv[Q1.deq()]`
$e4$ `T2: res[Q1.deq():x]`
$e5$ `T1: inv[Q2.enq(y)]`
$e6$ `T1: res[Q2.enq(y):void]`

В противном случае назовем историю H **конкурентной** (**concurrent**).

Парные события в истории

Историю назовем **well-formed**, если проекция этой истории на каждый поток – последовательная.

Будем рассматривать только такие истории.

Эквивалентные истории

Две истории H_1 и H_2 назовем **эквивалентными**, если для каждого потока T проекции этих историй на T совпадают.

$$H_1|T = H_2|T \text{ для каждого потока } T$$

Последовательная согласованность

Для высокоуровневых объектов:

История H **последовательно согласованна** (**sequentially consistent**), если существует последовательная история S, эквивалентная H.

Проекции этих историй на каждый из потоков совпадают.

Реплицированное значение

Коробка для значения с двумя методами put и get

```
replicated_value::put(x) {
    mutex.lock()
    for (int i = 0; i < n; ++i) {
        value[i] = x;
    }
    mutex.unlock()
}

replicated_value::get() {
    i = get_thread_id() // {0, ..., n-1}
    return value[i]
}
```

Реплицированное значение

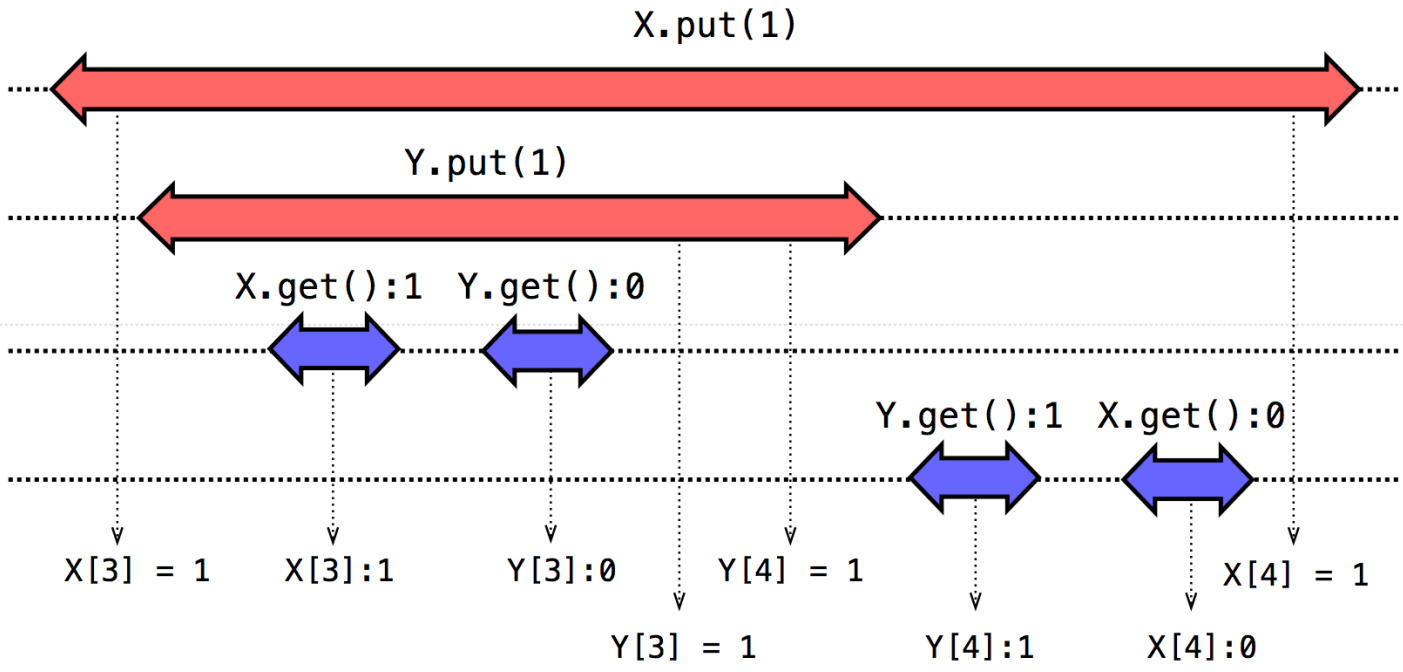
Для любой истории операций на одном таком объекте можно построить эквивалентную последовательную историю!

Все `put`-ы упорядочены между собой критическими секциями.

Все чтения последовательно согласованны, так что `get`-ы можно упорядочить относительно `put`-ов, сохранив порядок вызовов внутри каждого потока.

Короче говоря, любая конкурентная история операций над одним таким объектом **последовательно согласована!**

Реплицированное значение



Потоки видят записи в X и Y в разном порядке!

На уровне ячеек памяти никакого обмана нет – все чтения и записи происходят в последовательном порядке, сохраняется порядок инструкций в программе.

IRIW

Пример – **Independent Reads of Independent Writes**:

Изначально: `x = y = 0`

Поток T1	Поток T2	Поток T3	Поток T4
`x = 1`	`y = 1`	`r1 = x` `r2 = y`	`r3 = y` `r4 = x`

В последовательно согласованной модели памяти **не допускается** результат: `r1 == 1, r2 == 0, r3 == 1, r4 == 0`

Потоки $T3$ и $T4$ **не могут** видеть записи в `x` и `y` в разном порядке!

Проблема Sequential Consistency

На уровне последовательно согласованной памяти невозможна ситуация, когда потоки читают две записи в разном порядке.

Мы комбинируем несколько конкурентных объектов, каждый из которых в отдельности последовательно согласован.

И получаем историю, для которой нет эквивалентной последовательной истории.

Как же быть? Мы не знаем заранее, в каких комбинациях будут использоваться объекты...

Линеаризуемость

Нужна новая модель консистентности, которая позволит комбинировать высокоуровневые конкурентные объекты!

Herlihy, Wing

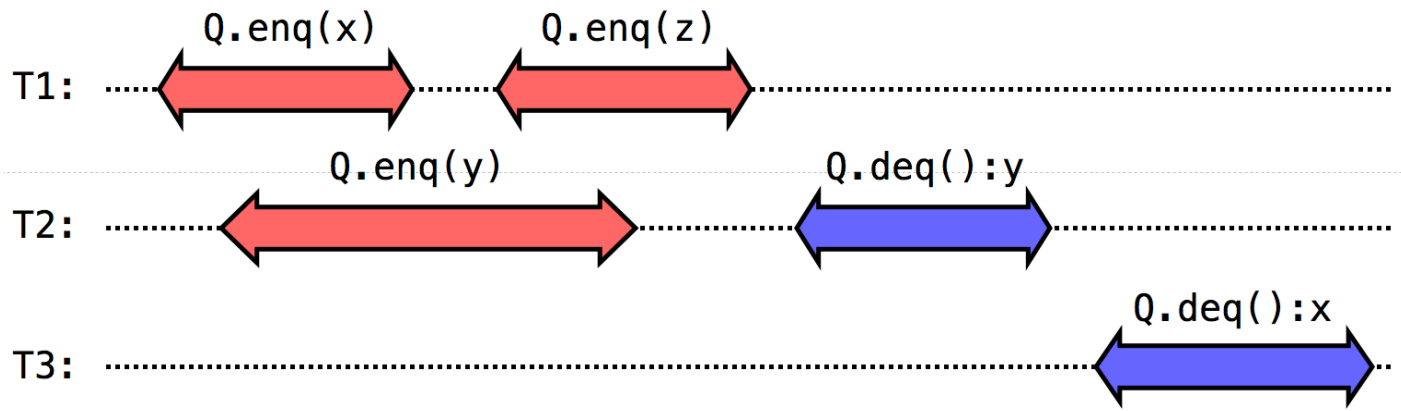
Linearizability: A Correctness Condition for Concurrent Objects

Линеаризуемость

История H индуцирует частичный порядок $<_H$ на операциях:

Положим $o_1 <_H o_2$, если возврат $res(o_1)$ предшествует вызову $inv(o_2)$ в истории H.

Частичный порядок $<_H$ – транзитивный.



Пример: `T1: Q.enq(x)` $<_H$ `T2: Q.deq(y)`

Линеаризуемость

Историю H назовем **линеаризуемой** (**linearizable**), если существует **последовательная** история S:

- Истории H и S эквивалентны, т.е. S сохраняет порядок операций каждого потока в H.
- История S соблюдает частичный порядок $<_H$.

Историю S назовем **линеаризацией** истории H.

Конкурентный объект назовем **линеаризуемым**, если все его истории линеаризуемы.

Отличие от последовательной согласованности: требуем соблюдения частичного порядка $<_H$.

Линеаризуемость $\Rightarrow$ Последовательная согласованность

Линеаризуемость

Интуитивно – для каждой конкурентной истории сущесвует **линеаризация** – эквивалентная ей последовательная история.

- **Конкурентные** операции в линеаризации могут располагаться в произвольном порядке
- **Неконкурентные** операции должны сохранить относительный порядок

Композируемость

Линеаризуемость – **композируемое** (**composable**) свойство!

> История H линеаризуема тогда и только тогда, когда линеаризуема проекция $H|_x$ для каждого конкурентного объекта x из H.

Достаточно доказать линеаризуемость каждого отдельного объекта в системе, тогда об исполнении всей системы можно рассуждать в модели чередования операций!

У нас есть частичный порядок вызовов в исходной истории H, обозначим его $<_H$.

Подыстория $H|x$ для каждого объекта x линеаризуема, обозначим линейный порядок вызовов в этой линеаризации через $<_x$.

Покажем теперь, что можно построить линейный порядок на всех вызовах в H, который сохранял бы частичный порядок $<_H$ вызовов исходной истории и порядок $<_x$ в линеаризации для каждого объекта x.

Построим граф, вершинами которого будут все вызовы в истории.

В качестве дуг этого графа возьмем объединение частичного порядка вызовов $<_H$ в исходной истории и порядка $<_x$ в линаризации подыстории каждого из объектов.

Покажем, что этот граф ацикличный, а значит линеаризацию можно получить топологической сортировкой.

От противного: предположим, что в построенном графе вызовов есть цикл.

Порядок $<_H$ и каждый $<_x$ – **ацикличны**, так что в цикле обязаны быть дуги из разных порядков.

Каждый $<_x$ и $<_H$ – **транзитивны**, так что если в цикле есть две идущие подряд дуги из одного порядка, то их можно заменить на одну.

Теперь можно считать, что в цикле чередуются дуги из разных порядков.

Дуги $e <_x f <_y g$ для разных объектов x и y подряд идти не могут, т.к. $<_x$ определен только на вызовах объекта x, а $<_y$ – на вызовах объекта y.

Значит дуги из линеаризаций могут чередоваться в цикле только с дугами из $<_H$.

Рассмотрим теперь чередование $e <_H f <_x g <_H h$ в цикле.

- $e <_H f$ означает, что $\texttt{res(e)} \prec \texttt{inv(f)}$ в истории H
- $f <_x g$ означает, что $\texttt{inv(f)} \prec \texttt{res(g)}$ в истории H, т.к. в противном случае выполнялось бы $g <_H f$, а порядок $<_x$ не может нарушать $<_H$ по определению линеаризуемости.
- $g <_H h$ означает, что $\texttt{res(g)} \prec \texttt{inv(h)}$ в истории H

Значит $\texttt{res(e)} \prec \texttt{inv(h)}$ в исходной истории H, так что три дуги подряд можно заменить на одну дугу $e <_H h$.

Такими преобразованиями можно свести найденный цикл в построенном графе к дугам только из порядка $<_H$.

Но частичный порядок $<_H$ ацикличный. А значит цикла не могло существовать.

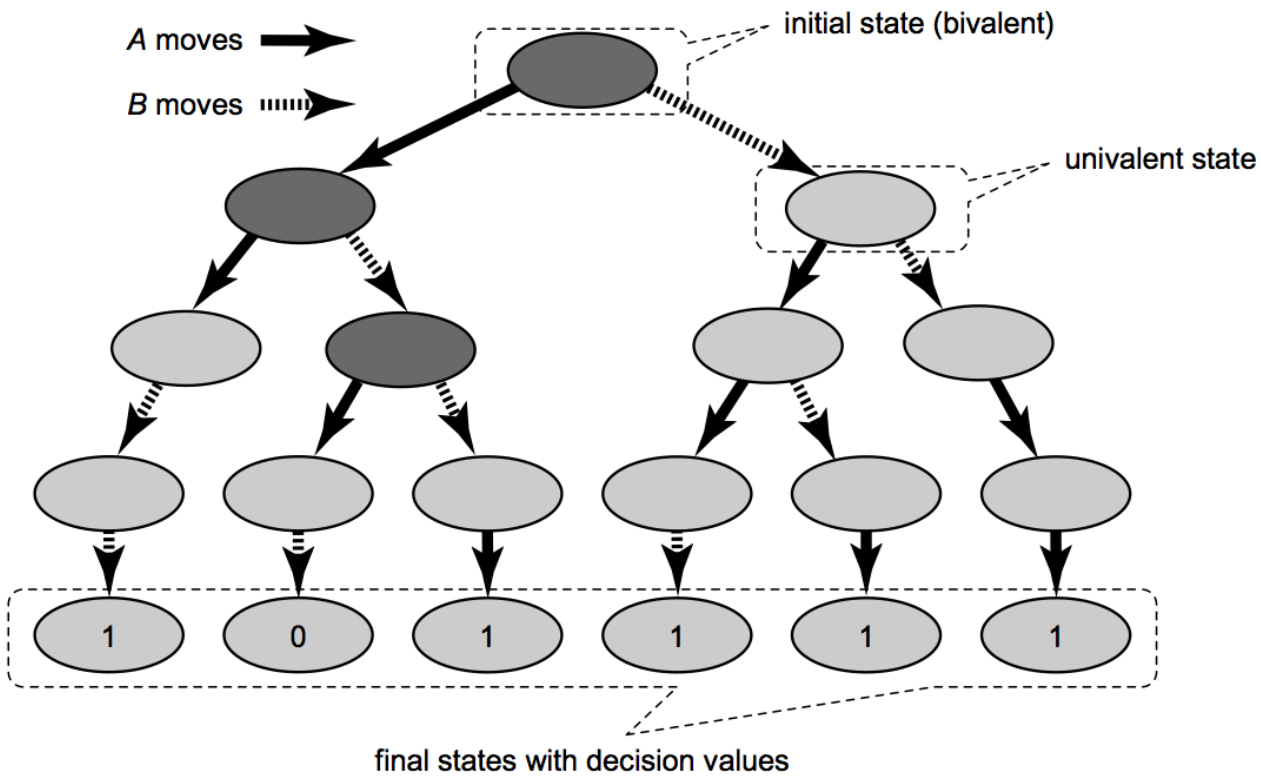
Линеаризуемость

Линеаризуемость позволяет задать для конкурентного объекта только последовательную спецификацию!

Не нужно специально определять, как ведет себя объект в случае, когда несколько потоков конкурентно выполняют операции над ним.

Консенсус

При док-ве невозможности решить консенсус мы предполагали линеаризуемость объектов и их композируемость!



Линеаризуемость

Линеаризуемость = Атомарность

Линеаризуемость – стандарт де-факто для большинства конкурентных структур данных.

Не выходите без неё на улицу!

Доказательство линеаризуемости

Мы построили конкурентный объект и теперь хотим показать, что он **атомарный**, т.е. что каждая конкурентная история вызовов **линеаризуема**.

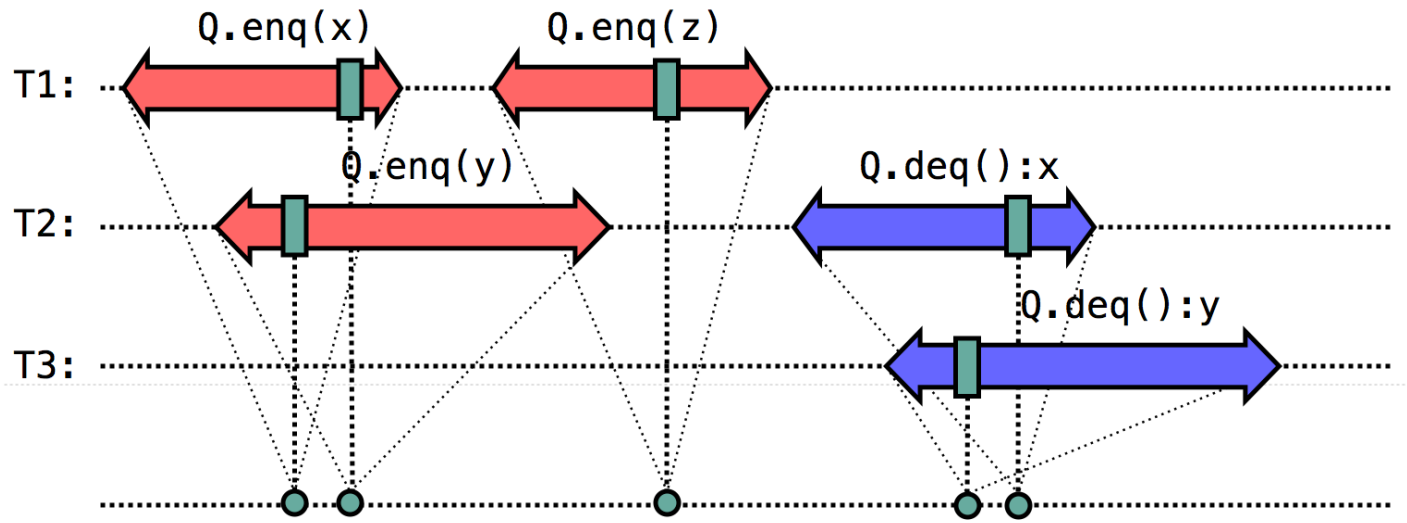
В общем случае простого рецепта **нет**!

Но часто можно обойтись простым методом – указать **точки линеаризации** операций каждой истории.

Точки линеаризации

Метод может последовательно выполнять много отдельных атомарных операций.

Но «происходит» метод в одной из атомарных операций **между вызовом и возвратом из вызова**.



В такой момент «фиксируется» результат всего метода.

Точки линеаризации

Если мы «стянем» каждый вызов в точку линеаризации, то получим эквивалентное последовательное исполнение.

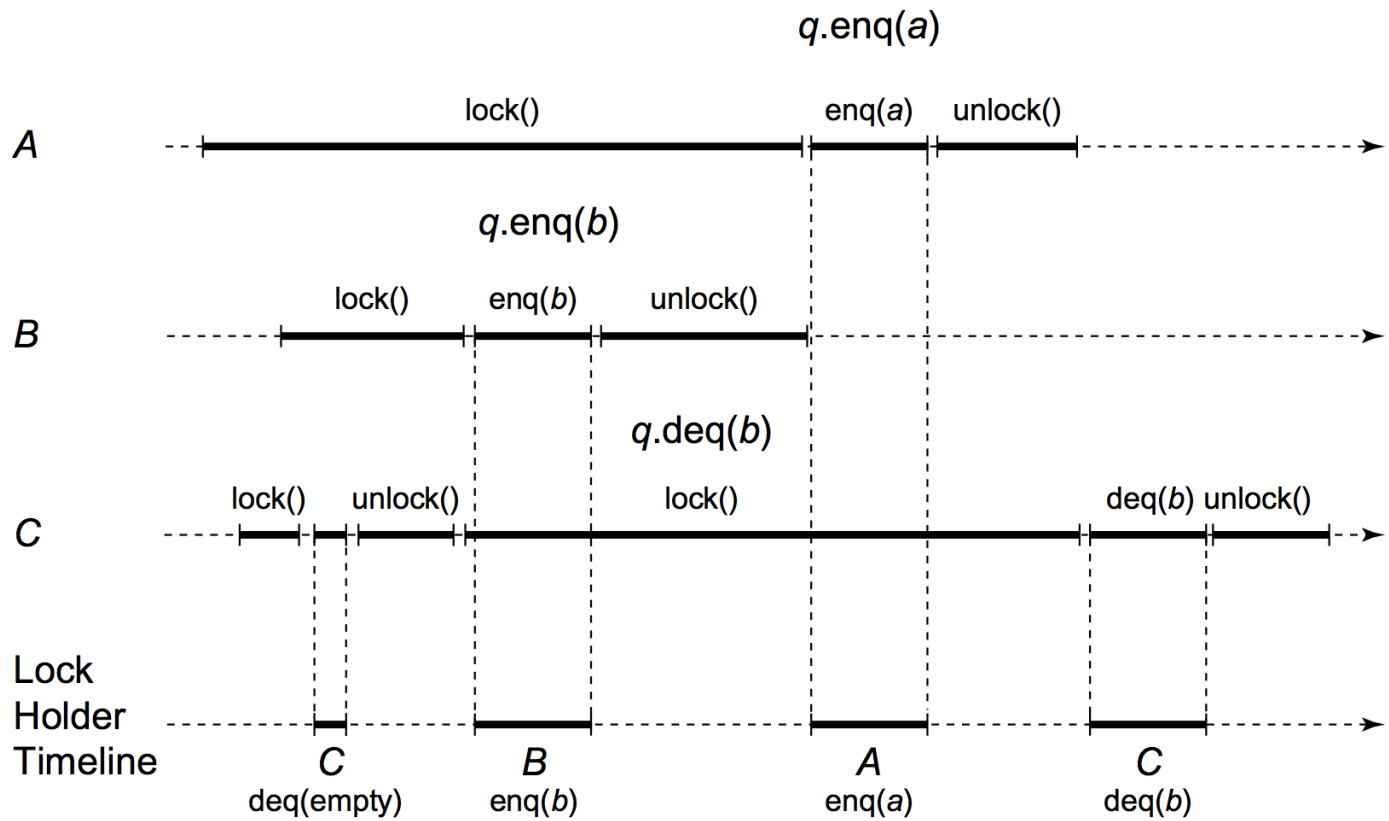
Точки линеаризации задают порядок конкурентных операций.

Чтобы доказать, что конкурентная история линеаризуема, нужно предъявить точки линеаризации.

Сделать это нужно для **каждой** конкурентной истории!

Большой мьютекс

Если конкурентный объект защищен одним мьютексом, то в качестве точек линеаризации операций достаточно выбрать `mtx.unlock()`.



Michael/Scott Lock-Free Queue

enqueue(T item): void

```
new_node = new node(item)

while (true) {
    curr_tail = tail
    curr_tail_next = tail->next

    if (!curr_tail_next) {
        if (CAS(tail->next, curr_tail_next, new_node)) // 1
            break
    } else {
        CAS(tail, curr_tail, curr_tail_next) // helping
    }
}

CAS(tail, curr_tail, new_node) // 2
```

Где точка линеаризации?

Универсальная lock-free конструкция

Добавление узла с операций в лог операций:

```
while (my->seq == 0) {
    before = max-in-head()
    after = before->cons.decide(my)
    before->next = after
    after->seq = before->seq + 1
    head[t] = after
}
```

Где точка линеаризации?

Универсальная wait-free конструкция

Добавление узла с операций в лог операций:

```
announce[t] = my
head[t] = max-in-head()
while (my->seq == 0) {
    before = head[t]
    turn = (before->seq + 1) mod n
    if (announce[turn]->seq == 0) {
        candidate = announce[turn]
    else:
        candidate = my
    after = before->cons.decide(candidate)
    before->next = after
    after->seq = before->seq + 1
    head[t] = after
}
```

Где точка линеаризации?

Универсальная wait-free конструкция

> Узел с операцией одного потока может быть выбран на консенсусе и вставлен в лог операций **другим** потоком!

Точка линеаризации – **первый успешное голосование за узел**.

Соответствующая операция голосования зависит находиться внутри вызова другого потока.

Ничего страшного в этом нет: главное, что точка линеаризации находится **между вызовом и возвратом** операции **в исходном потоке**.

Точки линеаризации

В общем случае указать точки линеаризации **сложно**!

Herlihy/Wing Queue

Пример из оригинальной статьи про линеаризуемость:

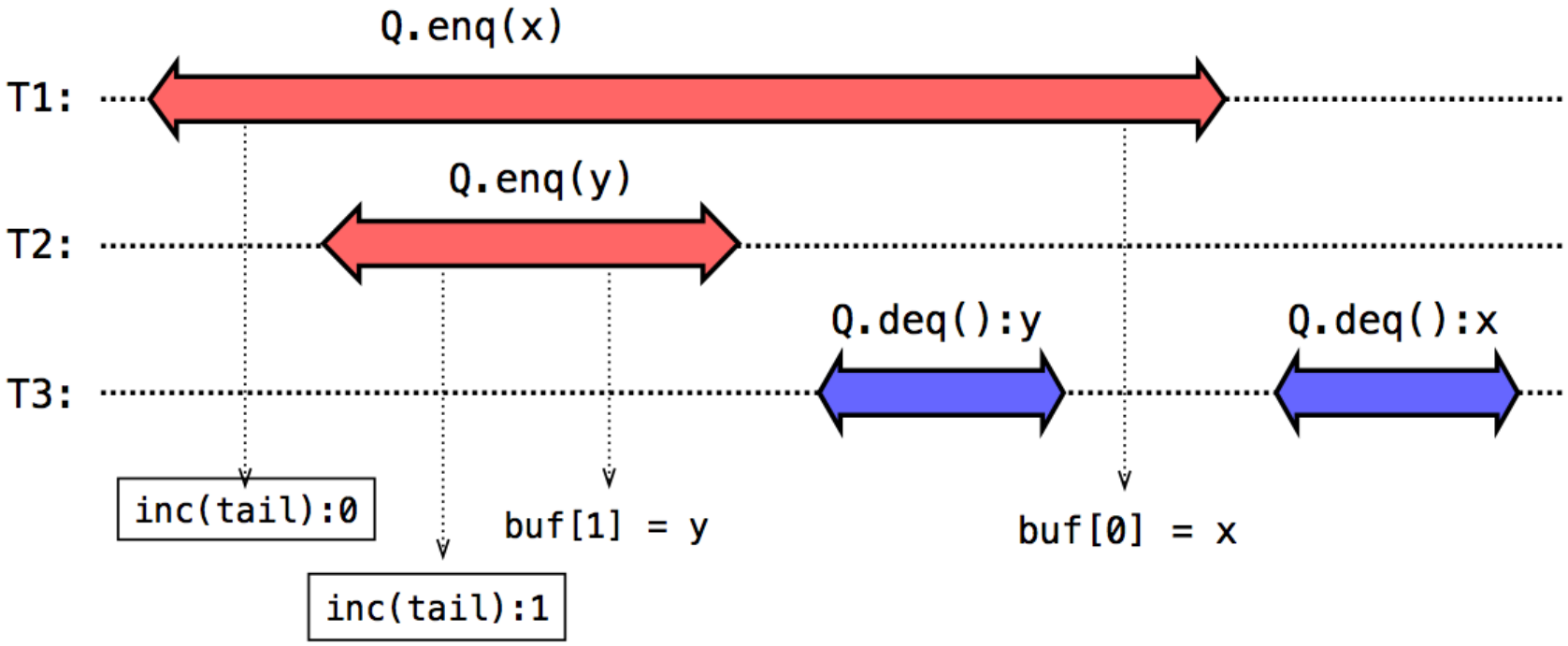
```
void enq(T item) {
    i = fetch-add(tail)
    buf[i] = item
}

T deq() {
    while (true) {
        range = tail
        for (int i = 0; i < range; ++i) {
            item = exchange(buf[i], null)
            if (item != null)
                return item
        }
    }
}
```

1) Это точно очередь? 2) Где точка линеаризации enq?

Herlihy/Wing Queue

Точкой линеаризации не может быть первая строка `Q.enq(x)`:

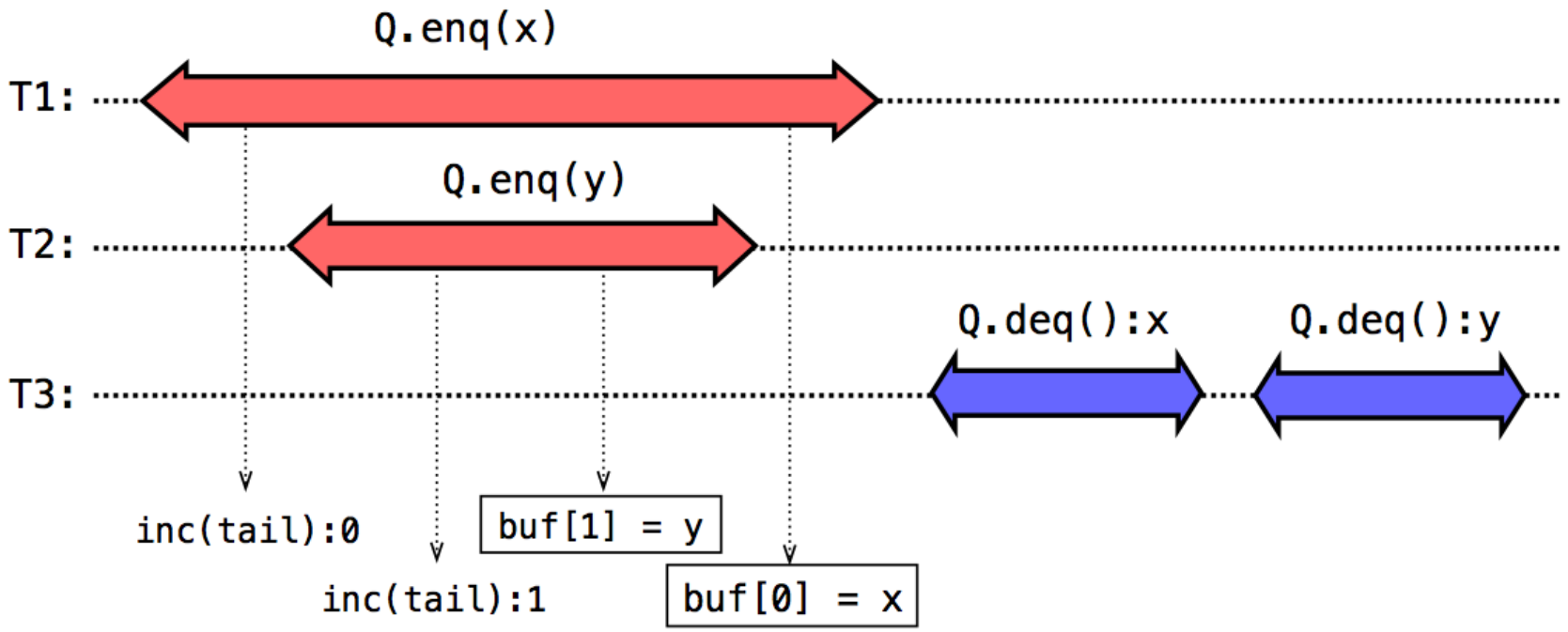
```
i = fetch-add(tail)
```



Herlihy/Wing Queue

Точкой линеаризации не может быть вторая строка `Q.enq(x)`:

```
buf[i] = item
```



Herlihy/Wing Queue

Значит ли это, что очередь Herlihy/Wing нелинеаризуема?

Herlihy/Wing Queue

Значит ли это, что очередь Herlihy/Wing нелинеаризуема?

Нет! Эта очередь линеаризуема!

Доказательство – в оригинальной статье Herlihy и Wing:

Linearizability: A Correctness Condition for Concurrent Objects

Herlihy/Wing Queue

Более того, существует такая конкурентная история, в которой вызов `Q.enq()` завершен, но точку его линеризации назвать нельзя:

Существуют разные продолжения этой истории, в зависимости от которых точка линеаризации этого вызова будет меняться!

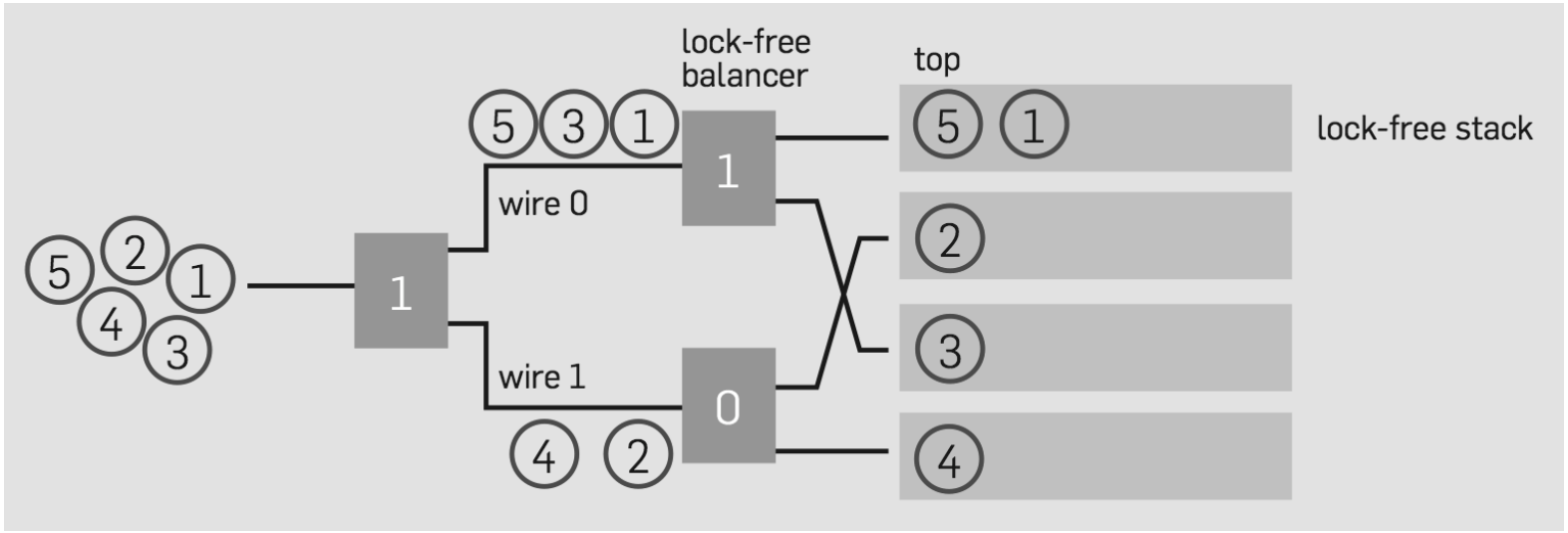
Пример Herlihy и Wing демонстрирует, что точки линеаризации не всегда известны заранее, и их не всегда можно указать по локальному фрагменту истории.

Точки линеаризации – трудности

- Точка линеаризации определяется не просто для операции, а для операции в конкретной истории!
- Точка линеаризации операции одного потока может находится в операции другого потока.
- Точки линеаризации не всегда известны заранее, их не всегда можно задать по локальному фрагменту истории.

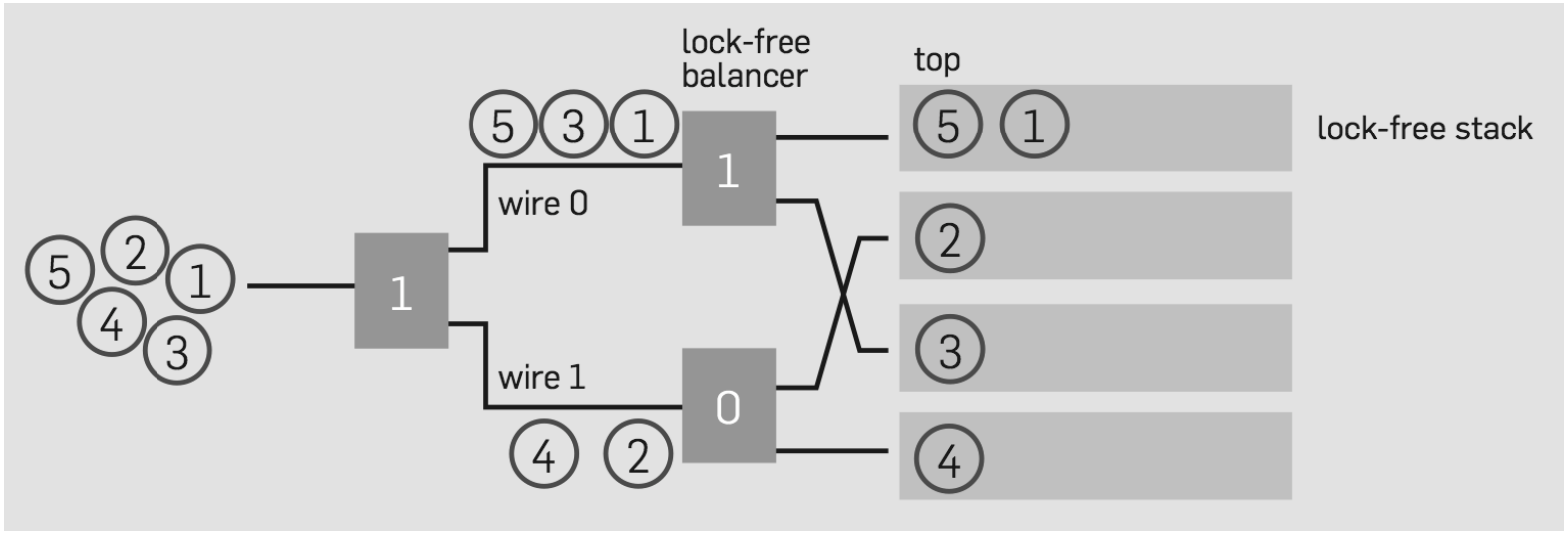
Древесный стек

Заведем несколько стеков, и будем раскидывать между ними элементы с помощью балансирующего дерева:



Древесный стек

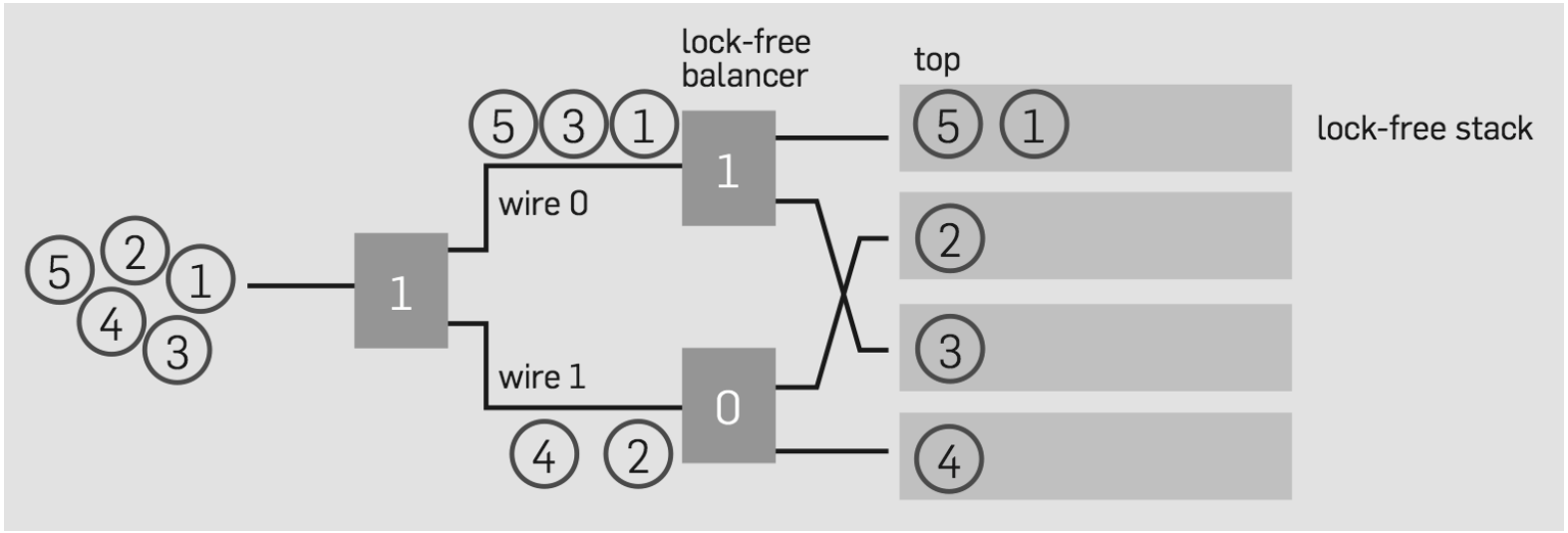
Заведем несколько стеков, и будем раскидывать между ними элементы с помощью балансирующего дерева:



Это точно стек?

Древесный стек

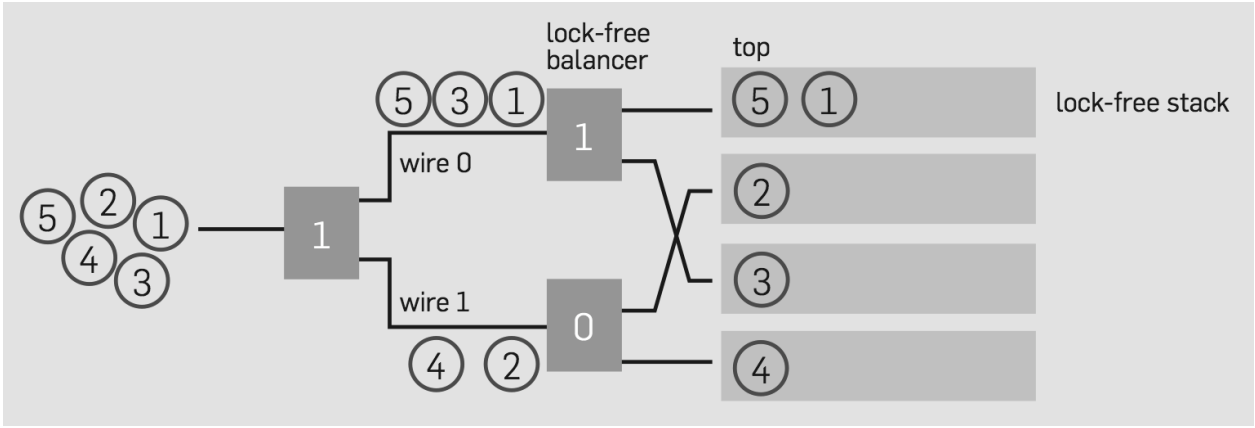
Заведем несколько стеков, и будем раскидывать между ними элементы с помощью балансирующего дерева:



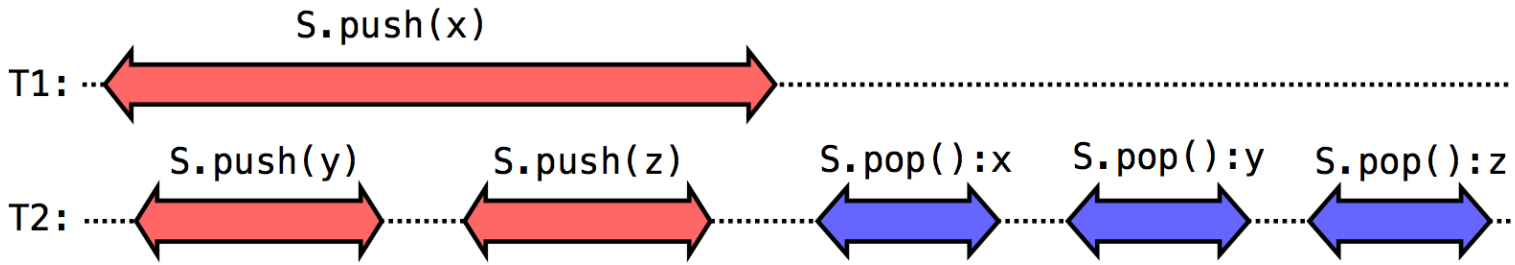
Это точно стек?

В **последовательном** мире - **да**!

Древесный почти-стек



Реализация допускает такую конкурентную историю:

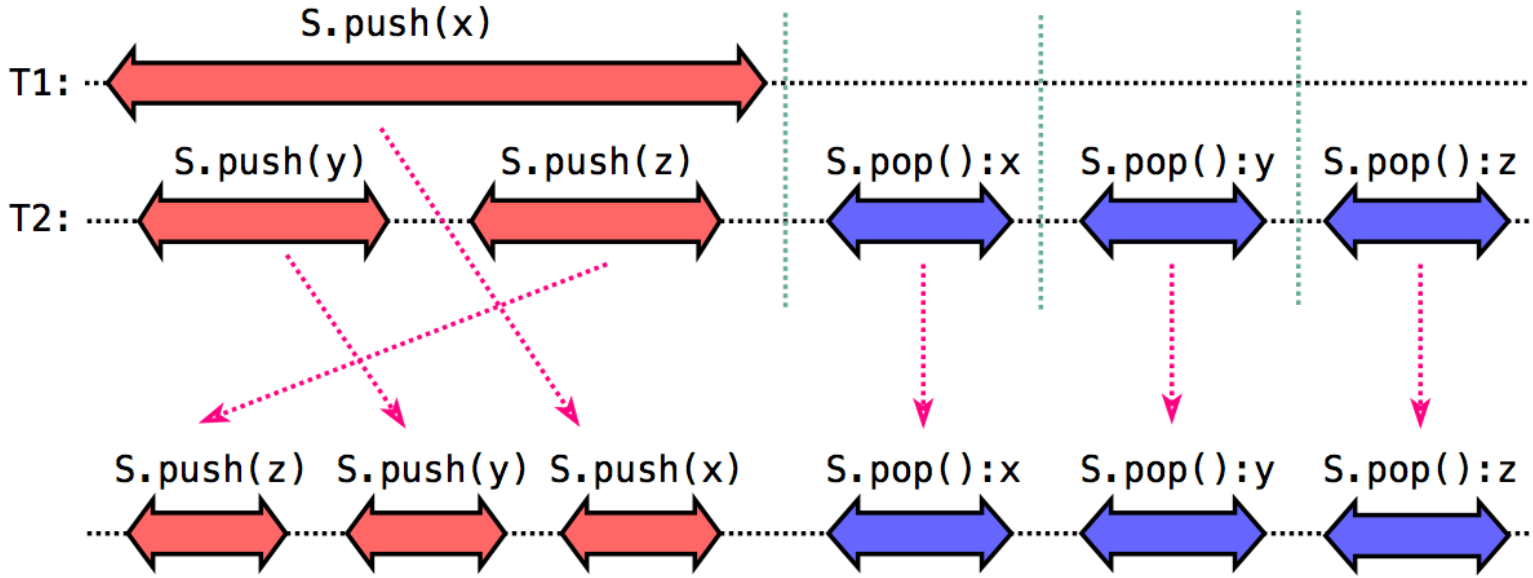


А теперь это совсем не похоже не стек!

Quiescent Consistency

Позволяет мыслить в терминах **последовательных** исполнений.

Но допускает при этом произвольные переупорядочивания вызовов **внутри одной эпохи**:



Quiescent Consistency – композируема!

Релаксация модели консистентности

Quiescent Consistency допускает странные для наблюдателя конкурентные исполнения, но все же гарантирует порядок между операциями из разных эпох.

Линеаризуемость, напротив, интуитивна.

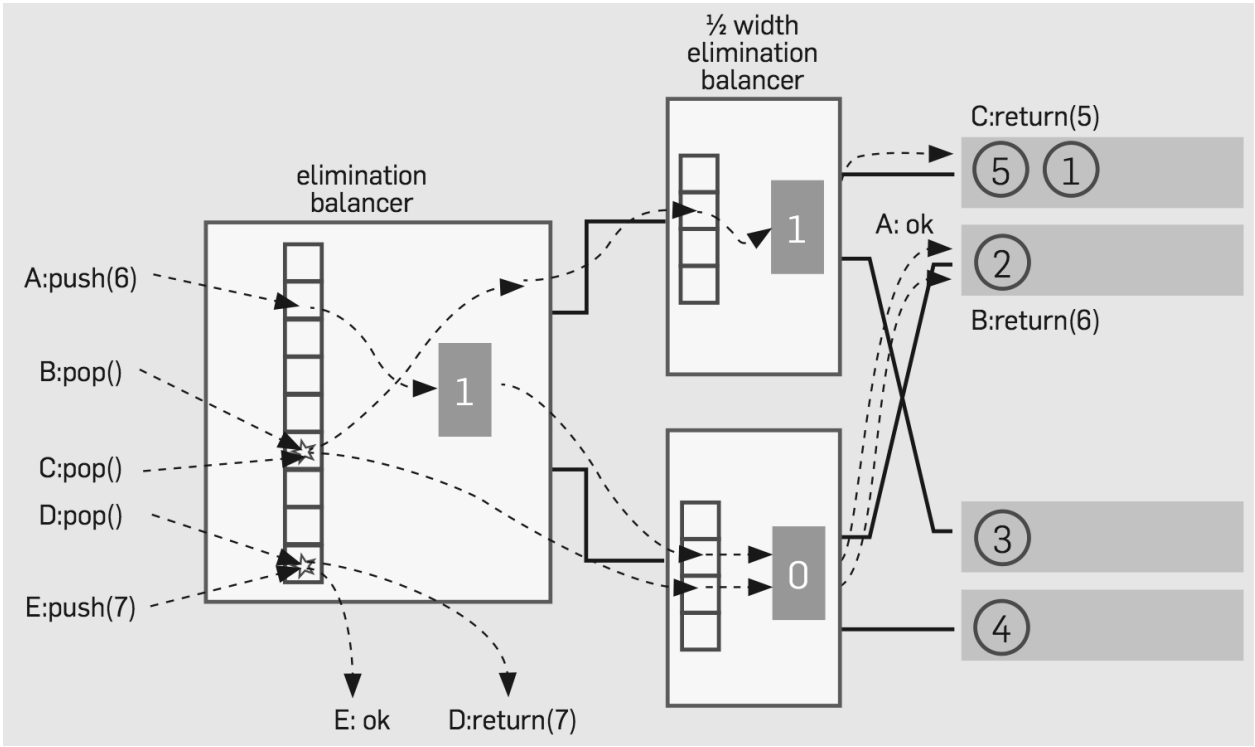
Зачем же ослаблять требования, когда есть линеаризуемость?

Более слабые гарантии позволяют проектировать более масштабируемые структуры данных с меньшими затратами на синхронизацию конкурентных операций.

Древесный стек с призмами

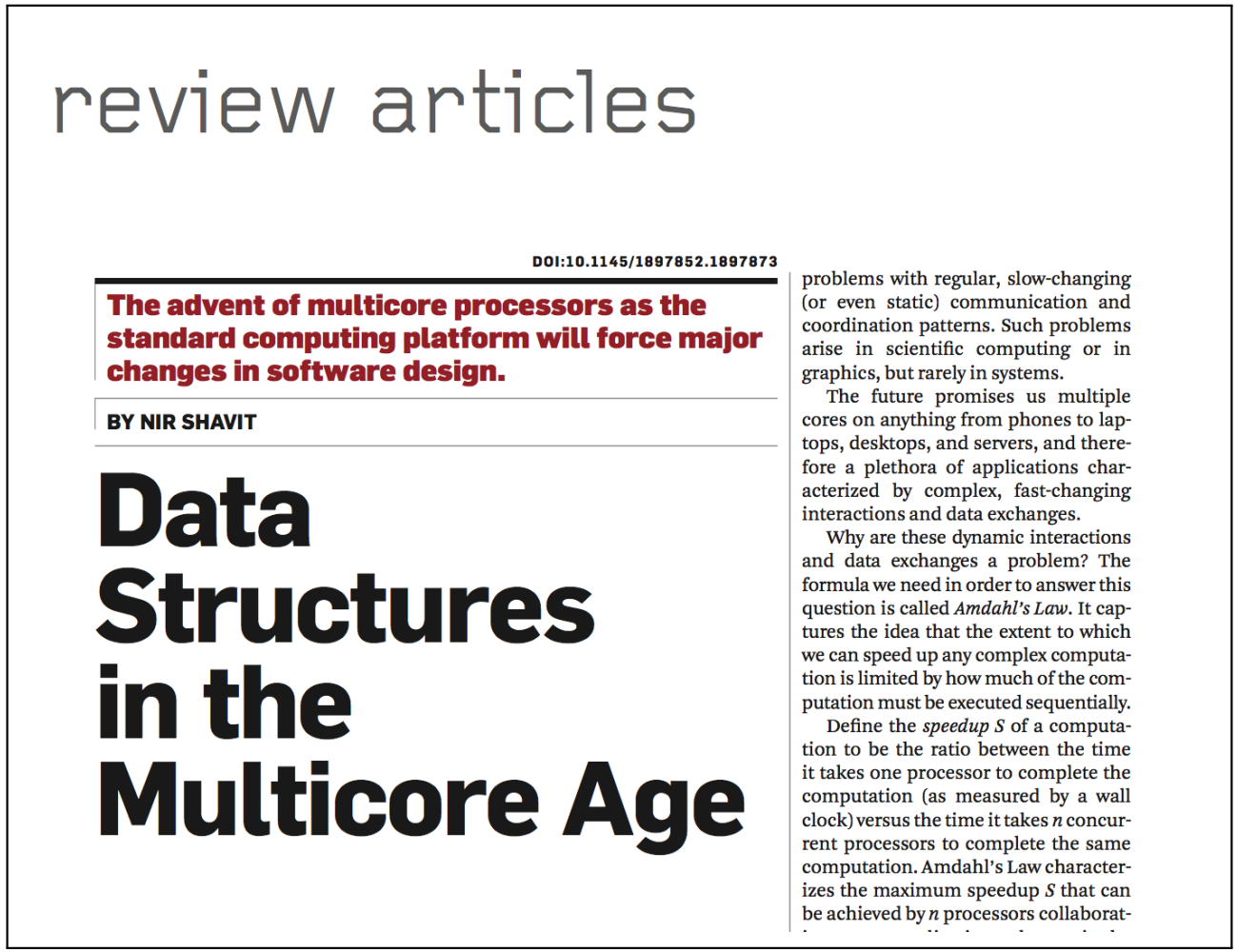
Каждый узел дерева – это точка контеншена.

Поэтому поставим перед каждым узлом дерева по **призме**, которая будет отражать и преломлять операции:



Релаксация гарантий

Nir Shavit - Data Structures in the Multicore Age



review articles

DOI:10.1145/1897852.1897873

The advent of multicore processors as the standard computing platform will force major changes in software design.

BY NIR SHAVIT

Data Structures in the Multicore Age

problems with regular, slow-changing (or even static) communication and coordination patterns. Such problems arise in scientific computing or in graphics, but rarely in systems.

The future promises us multiple cores on anything from phones to laptops, desktops, and servers, and therefore a plethora of applications characterized by complex, fast-changing interactions and data exchanges.

Why are these dynamic interactions and data exchanges a problem? The formula we need in order to answer this question is called *Amdahl's Law*. It captures the idea that the extent to which we can speed up any complex computation is limited by how much of the computation must be executed sequentially.

Define the *speedup S* of a computation to be the ratio between the time it takes one processor to complete the computation (as measured by a wall clock) versus the time it takes *n* concurrent processors to complete the same computation. Amdahl's Law characterizes the maximum speedup *S* that can be achieved by *n* processors collaborat-

Сравнение моделей консистентности

Рассмотренные модели консистентности позволяют рассуждать о конкурентных историях как о последовательных.

Каждая модель консистентности соблюдает при этом частичный порядок на операциях, который заложен в конкурентной истории:

- **Sequential consistency** – порядок между операциями в каждом потоке
- **Linearizability** – порядок между неконкурентными операциями
- **Quiescent Consistency** – порядок между операциями из разных эпох

Сравнение моделей консистентности

Sequential consistency подходит для глобальных монолитных объектов: таких, как совокупность всех ячеек памяти.

Для высокоуровневых объектов нужна **композируемость**, чтобы рассуждать об исполнении **всей системы** в **модели чередования** операций.

Sequential consistency здесь не годится.

Нужно выбирать между **linearizability** и **quiescent consistency**.

Мораль

Сначала мы исключили физическую параллельность из **модели памяти** с помощью **последовательной согласованности**.

Теперь мы перешли к последовательной модели исполнения и для **высокоуровневых объектов** с помощью **линеаризуемости**.

Линеаризуемость

- Позовляет комбинировать атомарные объекты
- Относительно легко достижима на практике