Неблокирующая синхронизация

Гарантии прогресса для мьютекса

- **Свобода от взаимной блокировки** – один из вызовов `mtx.lock()`завершается
- **Свобода от голодания** – каждый вызов `mtx.lock()` завершается

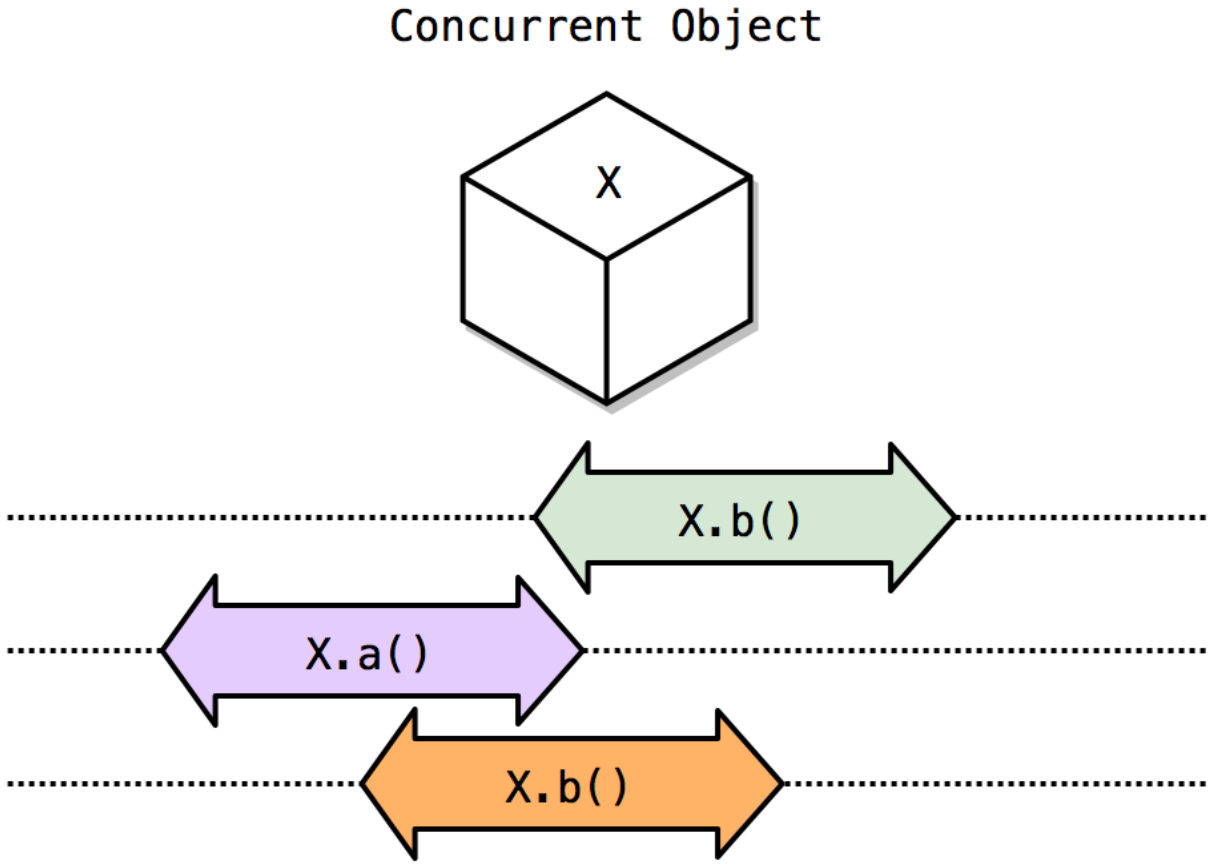
В предположении, что потоки исполняют **конечную** критическую секцию.

Свобода от взаимной блокировки гарантирует **глобальный прогресс всей системы**, но допускает, что конкретный вызов никогда не завершится.

Свобода от голодания гарантирует **прогресс каждому потоку**.

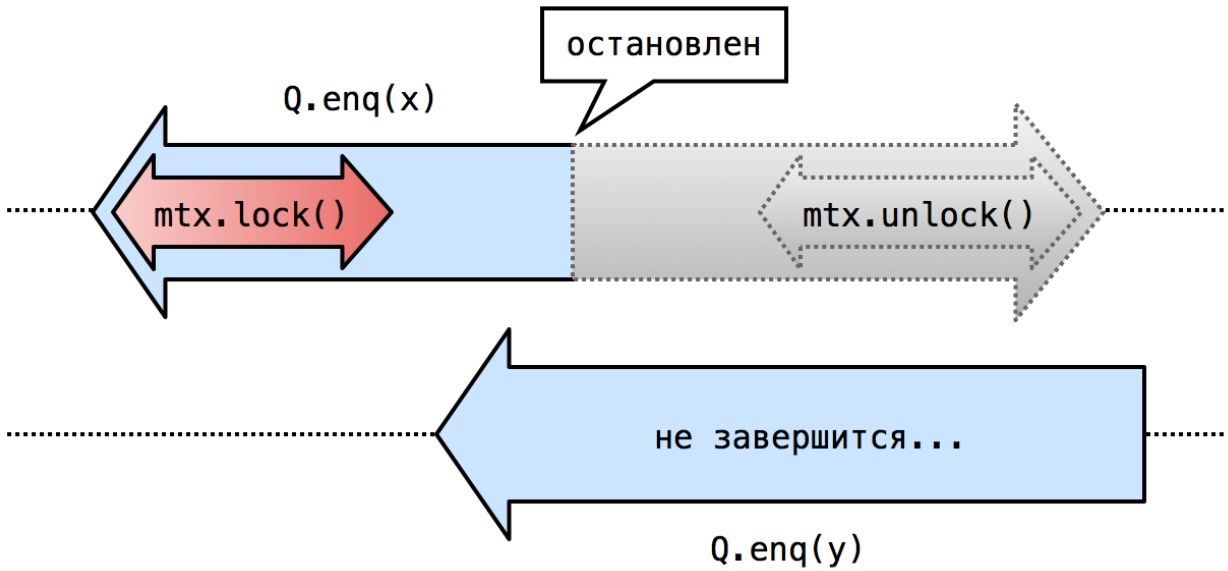
Concurrent Object

Гарантии можно распространить на произвольный объект, который использует блокирующие примитивы синхронизации.



Блокировка одного вызова другим

Остановка потока в неудачный момент может привести к тому, что другой вызов на том же объекте не сможет завершиться:



Такой проблеме подвержен любой объект, в реализации которого используются мьютексы и подобные блокирующие примитивы синхронизации.

А как же гарантии прогресса?

Все это не противоречит сформулированным гарантиям прогресса!

Свобода от взаимного исключения и **свобода от голодания** определяет прогресс потоков в случае, когда потоки **исполняются**.

Попробуем получить гарантии прогресса даже для случаев, когда потоки могут **останавливаться в произвольные моменты**.

Почему это важно?

В реальности поток не останавливается навечно.

Причины пауз в исполнении:

- Вытеснение потока планировщиком по истечении кванта времени
- Промах по локальному кэшу и последующее обращение к оперативной памяти
- Page Fault и подгрузка страницы из оперативной памяти

Параллели с распределенными системами

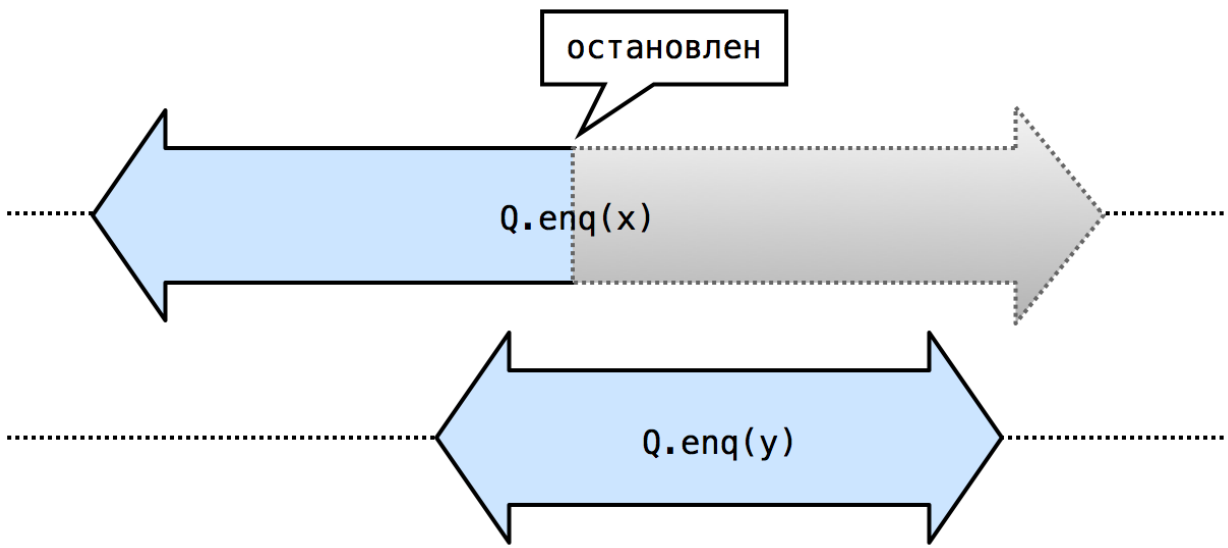
Идея таких гарантий пришла из мира распределенных систем.

В распределенных системах сбой узла или сети – нормальная ситуация.

Устанавливается модель сбоев, и в этой модели решаются (или доказывается невозможность решения) задачи координации узлов.

Гарантия неблокирующей синхронизации

Остановка вызова метода в одном потоке не должна блокировать исполнение конкурентных вызовов в других потоках.



Пример

Мьютекс Петерсона

Поток T0	Поток T1
`flag[0] = true`	`flag[1] = true`
`victim = 0`	`victim = 1`
`while flag[1] and victim == 0:`	`while flag[0] and victim == 1:`
`    pass`	`    pass`

Если поток $T1$ внутри вызов `lock(1)` установит флаг и уснет, не установив `victim`, то поток $T0$ не сможет завершить свой вызов `lock(0)`.

Прогресса в течение времени простоя $T1$ не будет.

Свобода от ожидания (Wait Freedom)

Самая сильная неблокирующая гарантия!

> Метод **свободен от ожидания** (**wait-free**), если **каждый** вызов этого метода завершается за **конечное** число шагов, вне зависимости от поведения других потоков.

Свобода от ожидания гарантирует прогресс **каждому** вызову метода.

Свобода от блокировки (Lock Freedom)

Самая важная с практической точки зрения неблокирующая гарантия!

Не строгое, зато интуитивно понятное определение:

> Метод **свободен от блокировок** (**lock-free**), если хотя бы один из вызовов метода продвигается вперед, т.е. имеет место **глобальный прогресс**, вне зависимости от поведения других потоков.

При этом каждый конкретный вызов может бесконечно голодать, но при условии, что завершаются конкурентные вызовы.

Свобода от блокировки (Lock Freedom)

Чуть более строго:

> При бесконечном исполнении вызовы метода завершаются бесконечно много раз.

Все еще нужно строго определить, что такое бесконечное исполнение...

Гарантии прогресса

Wait-Freedom гарантирует прогресс **каждому** вызову метода

Lock-Freedom – по крайней мере одному вызову.

$$\text{Wait-Freedom} \Rightarrow \text{Lock-Freedom}$$

Гарантии прогресса

Можно провести очевидную параллель между гарантиями прогресса для блокирующей синхронизации и неблокирующей:

Блокирующая

- Deadlock Freedom
- Starvation Freedom

Неблокирующая

- Lock Freedom
- Wait Freedom

Obstruction Freedom

Гарантия, интересная только теоретикам:

> Назовем метод **obstruction-free**, если он завершается за конечное число шагов в ситуации, когда конкурирующие вызовы не исполняются.

Остальные вызовы могут быть остановлены в произвольном месте!

Нет мьютексов

Неблокирующая синхронизация исключает использование мьютексов и спинлоков.

Нет мьютексов $\Rightarrow$ невозможны дедлоки

Но отсутствие мьютексов не означает, что реализация – LF!

Compare-And-Swap

Главный инструмент неблокирующей синхронизации:

```
CAS(T& var, T& expected, T desired): bool
```

```
atomic {
    current = var
    if (current == expected) {
        var = desired
        return true
    } else {
        expected = current
        return false
    }
}
```

Compare-And-Swap

CAS – самая выразительная RMW-операция!

Теоретическая линейка – **Wait-Free иерархия**

Compare-And-Swap

С помощью CAS легко атомарно выполнить любое преобразование вида $x \to f(x)$

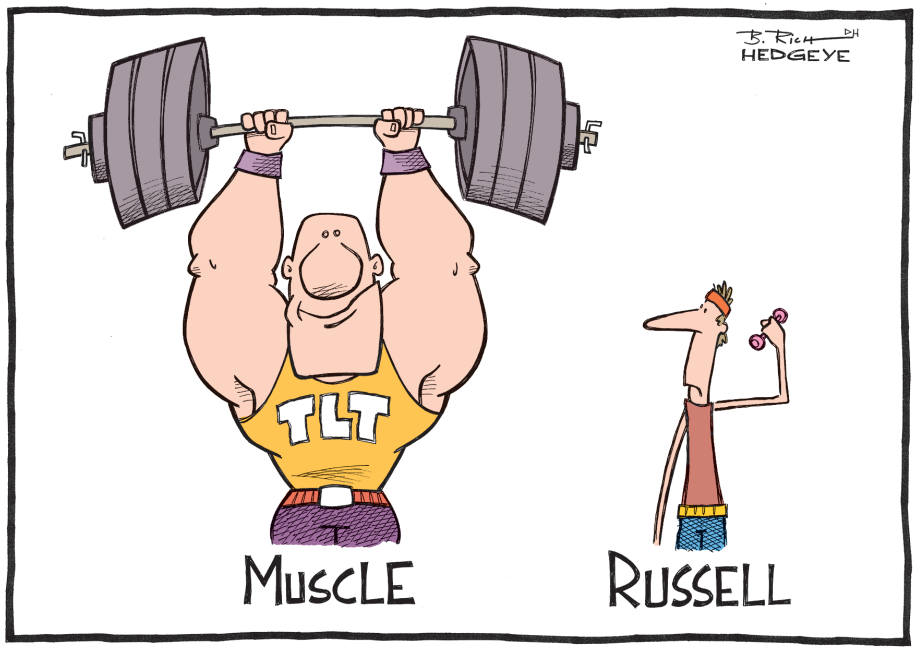
Например, `fetch_and_add`, где $f : x \to x + y$:

```
int fetch_and_add(std::atomic<int> x, int y) {
    int curr_x = x.load();
    while (!x.compare_exchange_weak(x, curr_x, curr_x + y))
        {}
}
```

CAS в C++

В C++ две версии CAS – **слабая** и **сильная**:

- `atomic<T>::compare_exchange_weak(...)`
- `atomic<T>::compare_exchange_strong(...)`



Последствия

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

Гораздо больше промежуточных публичных состояний структуры данных.

Про модель памяти

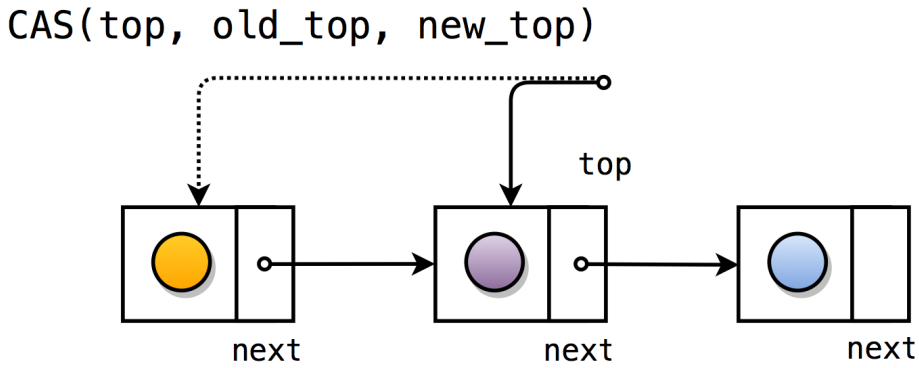
Раньше мьютексы сериализовали обращения к данным из разных потоков.

Каждая последующая критическая секция видела записи в память из предыдущей критической секции.

Теперь нужно думать о гонках и видимости записей через happens-before.

Treiber Lock-Free Stack

Treiber LF Stack: Push




```
push(T item): void
```

```
new_node = new node(item)

curr_top = top
new_node->next = curr_top

while (!CAS(top, new_node->next, new_node)) {
    // wait
}
```

Treiber LF Stack: Pop

```
pop(T& item): bool
```

```
curr_top = top

while (true) {
    if (!curr_top) {
        return false
    }
    if (CAS(top, curr_top, curr_top->next)) {
        item = curr_top->item
        return true
    }
}
```

Treiber LF Stack: Lock Freedom

И `push`, и `pop` перебрасывают всего **один** указатель `top`, причем делают это **атомарно**:

```
push:    CAS(top, new_node->next, new_node)

 pop:    CAS(top, curr_top, curr_top->next)
```

Treiber LF Stack: Освобождение памяти

Нужно освобождать память!

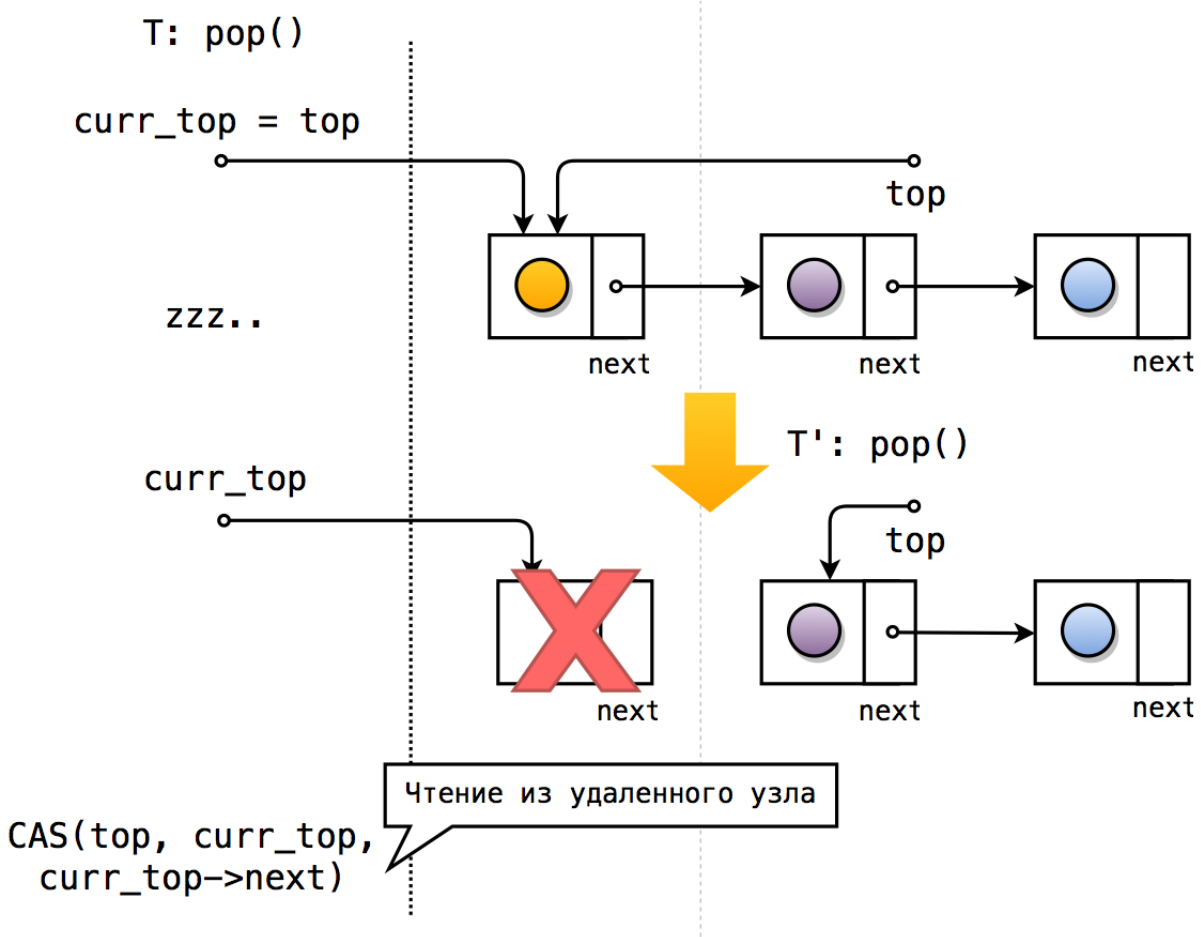
`pop(T& item): bool`

```
curr_top = top

while (true) {
    if (!curr_top) {
        return false
    }
    if (CAS(top, curr_top, curr_top->next)) {
        item = curr_top->item
        // delete curr_top ?
        return true
    }
}
```

Treiber LF Stack: Освобождение памяти

`delete curr_top` после извлечения узла – **плохая идея**!



Про гарантии аллокатора

Большинство lock-free структур данных собраны из узлов, которые ссылаются друг на друга через указатели.

Если структура данных выделяет память в своих методах, то для формального соблюдения гарантии lock-free **аллокатор тоже должен быть lock-free**.

Treiber LF Stack: Освобождение памяти

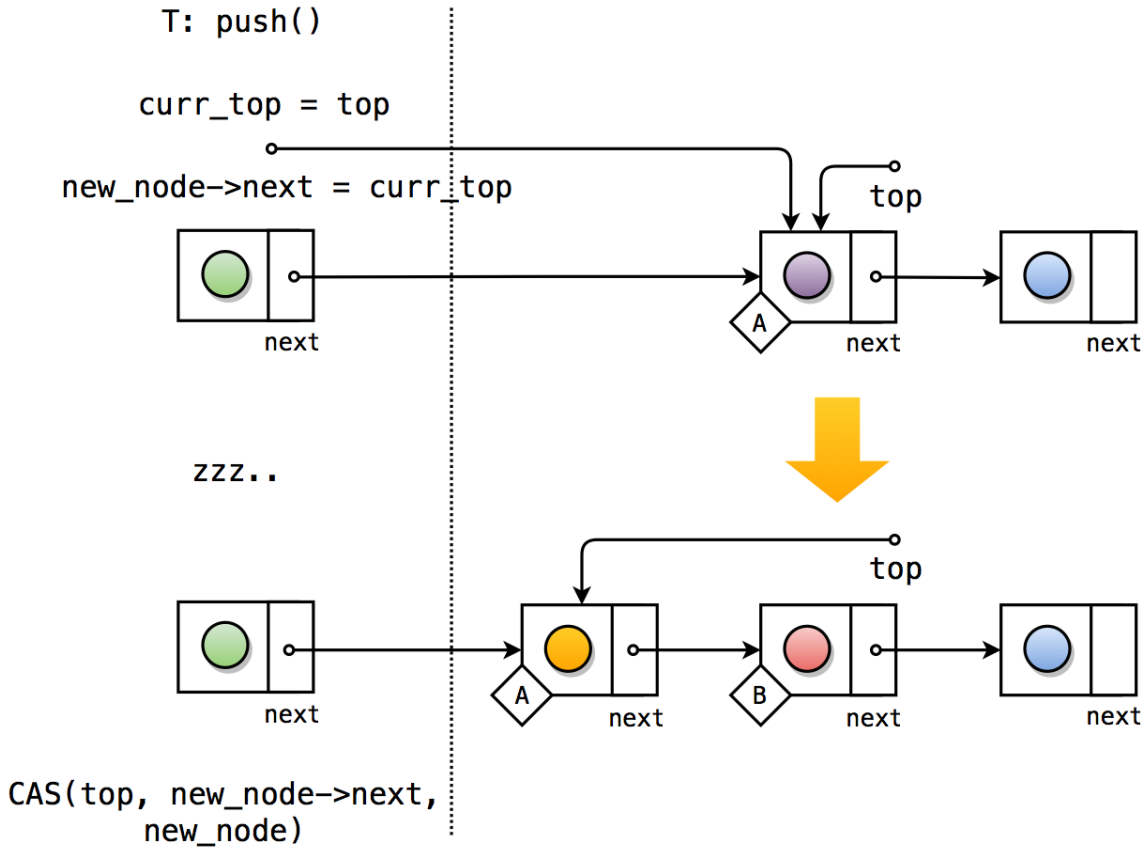
Будем переиспользовать извлеченные узлы – **node recycling**:

- Список свободных узлов
- Потоки в `pop` добавляют в его извлеченные узлы
- Потоки в `push` – забирают из него узлы

Фактически, еще один **lock-free стэк**!

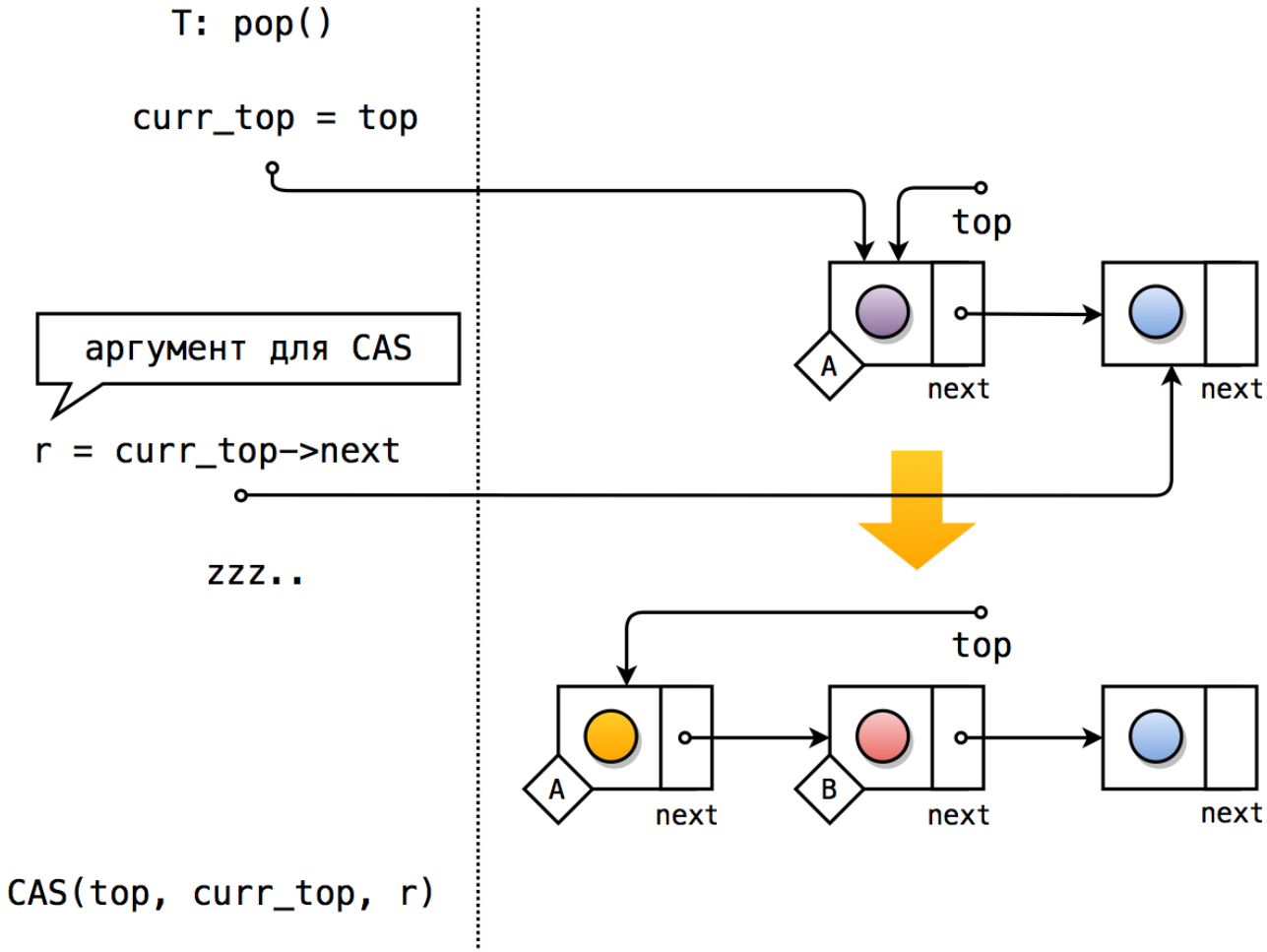
Treiber LF Stack: ABA в Push

CAS проверяет только значение, но не знает о том, менялось ли это значение с момента последнего чтения.



Работает!

Treiber LF Stack: ABA в Pop



Беда!

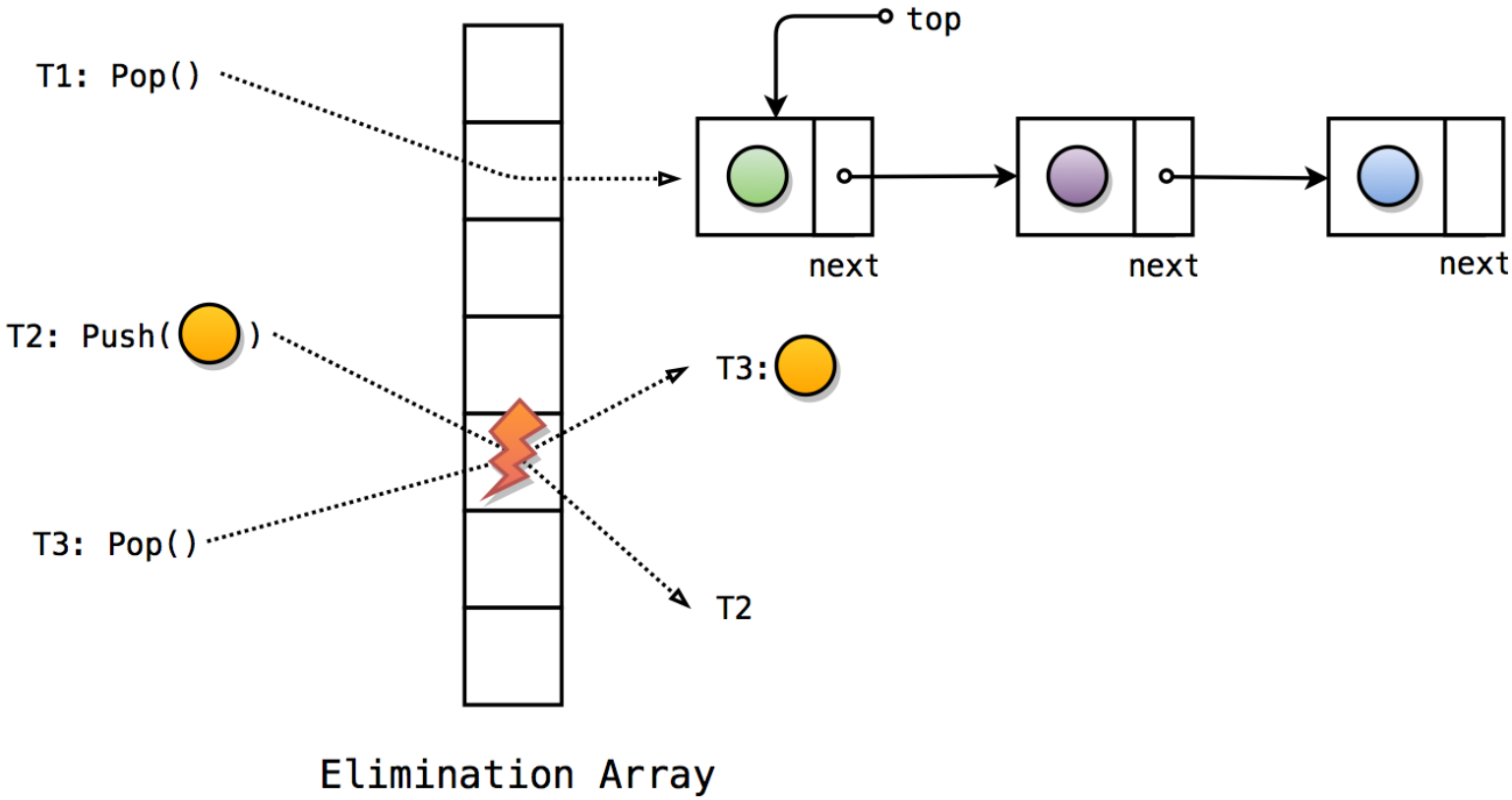
ABA и CAS

На некоторых архитектурах **ABA не бывает**!

Elimination Backoff Stack

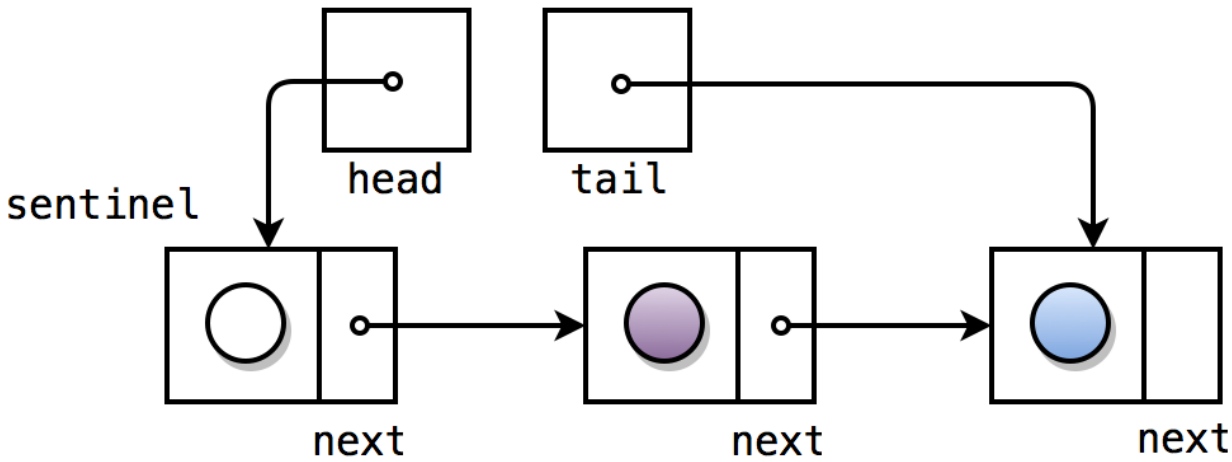
У стека есть **hot spot** – указатель на вершину

Никакой параллельности!

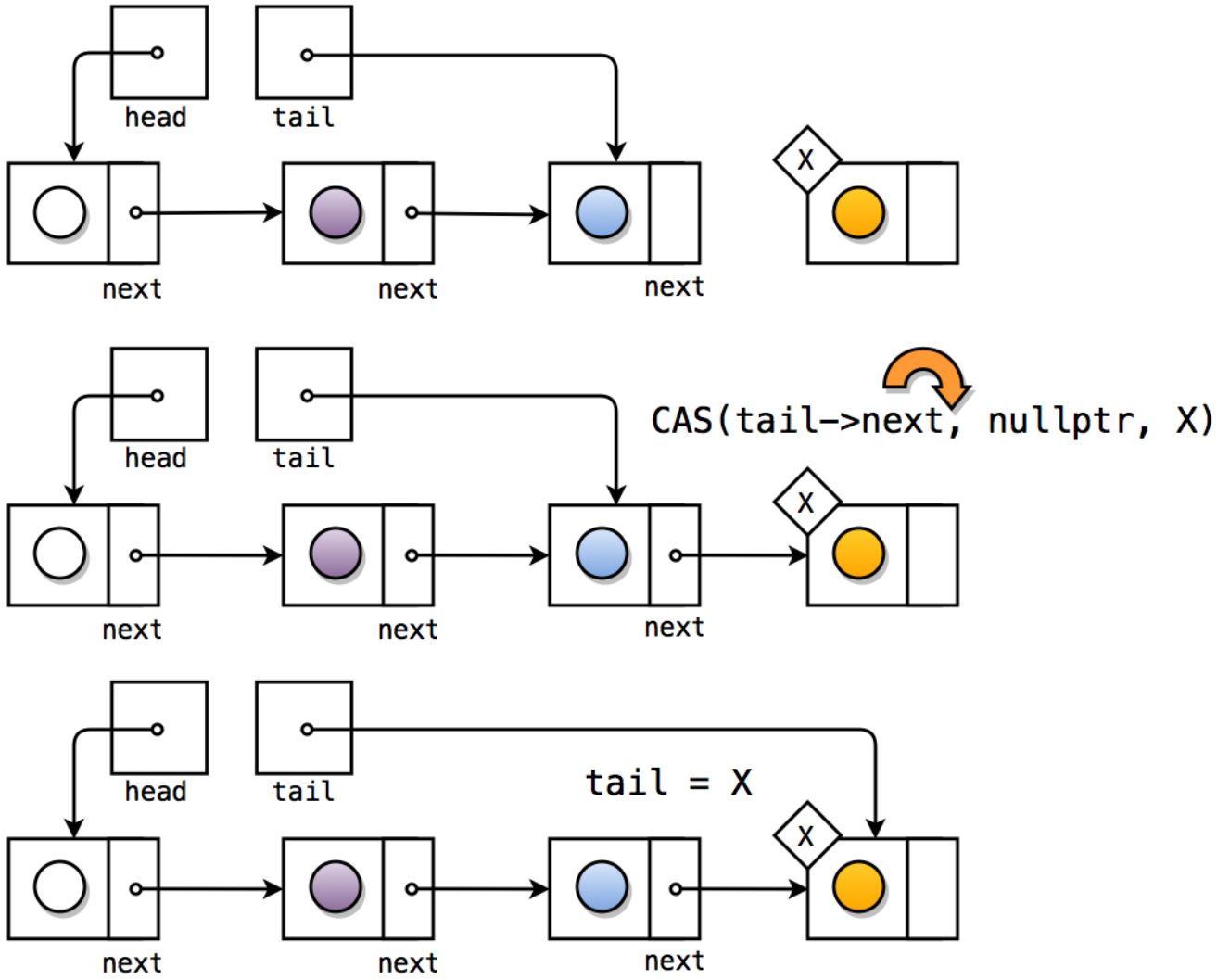


Michael & Scott Lock-Free Queue

Первый подход к очереди



Первый подход к очереди



Michael & Scott LF Queue

Метод `enqueue` делает **две** публичные модификации:

1. Цепляет новый узел
2. Переставляет указатель `tail`

Если поток уснет **между** этими двумя шагами, то другие потоки не смогут завершить свои вызовы `enqueue`.

Не удовлетворяет гарантии non-blocking!

В реализации стека такой проблемы не возникало: только одна публичная модификация контейнера с помощью CAS.

Michael & Scott LF Queue

Основная идея – **взаимопомощь** (**helping**)

> Если другой поток видит вставку в промежуточном состоянии, то он сам пробует завершить ее.

Продвигать вперед `tail` теперь могут несколько потоков, так что вместо обычной записи нужно использовать CAS.

Делать при этом CAS в цикле **не нужно**: если CAS на продвижение `tail` не сработал, то значит кто-то другой успел передвинуть `tail`.

M&S LF 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
```

M&S LF Queue

dequeue(T& item): bool

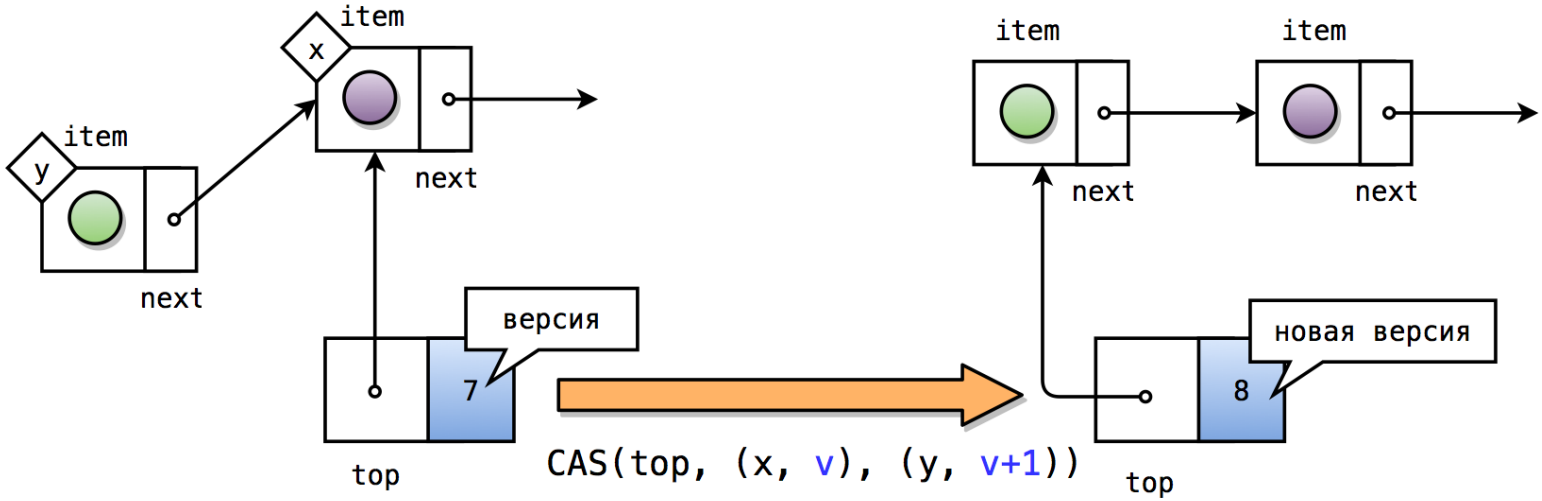
```
while (true) {
    curr_head = head
    curr_tail = tail
    curr_head_next = curr_head->next

    if (curr_head == curr_tail) {
        if (!curr_head_next)
            return false
        else
            CAS(tail, curr_head, curr_head_next) // helping
    } else {
        if (CAS(head, curr_head, curr_head_next)) {
            item = curr_head_next->item
            return true
        }
    }
}
```

Safe Memory Reclamation и ABA

Tagged Pointers

Для каждого указателя, на котором делаем CAS, храним **версию**. Вместе с каждым CAS-ом увеличиваем версию.



Теперь можно спокойно переиспользовать извлеченные узлы.

Но как делать CAS на двух машинных словах?

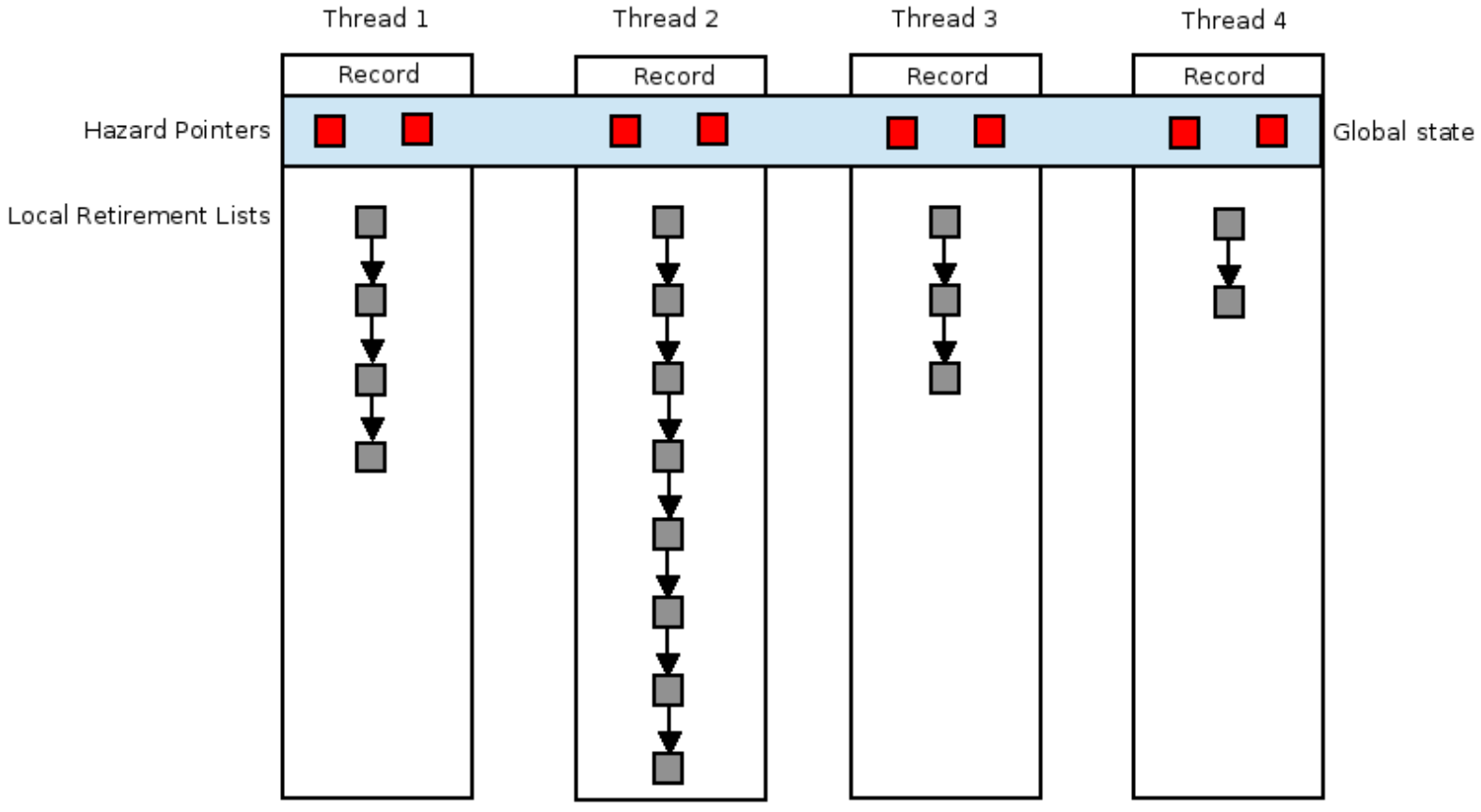
M&S Lock-Free Queue + Tagged Pointers

При реализации с помощью tagged pointers возникают интересные топологические эффекты из-за переиспользования узлов.

Смотри оригинальную статью:

> Maged Michael, Michael Scott – Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms

Hazard Pointers



Заблуждения

Lock-Free – это когда нет мьютексов и есть CAS-ы

Нет! Например, в случае использования спинлока.

Lock-Free – это про производительность

В первую очередь, неблокирующая синхронизация – **про гарантии прогресса**!

Во-вторых, нужно делать бенчмарки.

Резюме

- Неблокирующие гарантии синхронизации – остановка одного потока не должна блокировать глобальный прогресс
- Две неблокирующие гарантии - Wait Freedom и Lock Freedom
- Wait Freedom – теоретическая, Lock Freedom – практическая
- CAS - главный инструмент в построении неблокирующих структур данных
- Trieber LF стек и Michael & Scott LF Queue
- Трудности с управлением памятью: Safe Memory Reclamation и ABA