FAA-Based Queues and Flat Combining

Никита Коваль, JetBrains
Роман Елизаров, JetBrains

ИТМО, 2020

FAA-Based Queues

Fetch-And-Add

- `FAA(address, delta)` - *атомарно* увеличивает значение на `delta` и возвращает старое значение

- `FAA` гораздо лучше масштабируется, чем `CAS-loop`
  - ведь всегда успешен!

Modern queues use `Fetch-And-Add`


PPoPP'13

Fast Concurrent Queues for x86 Process

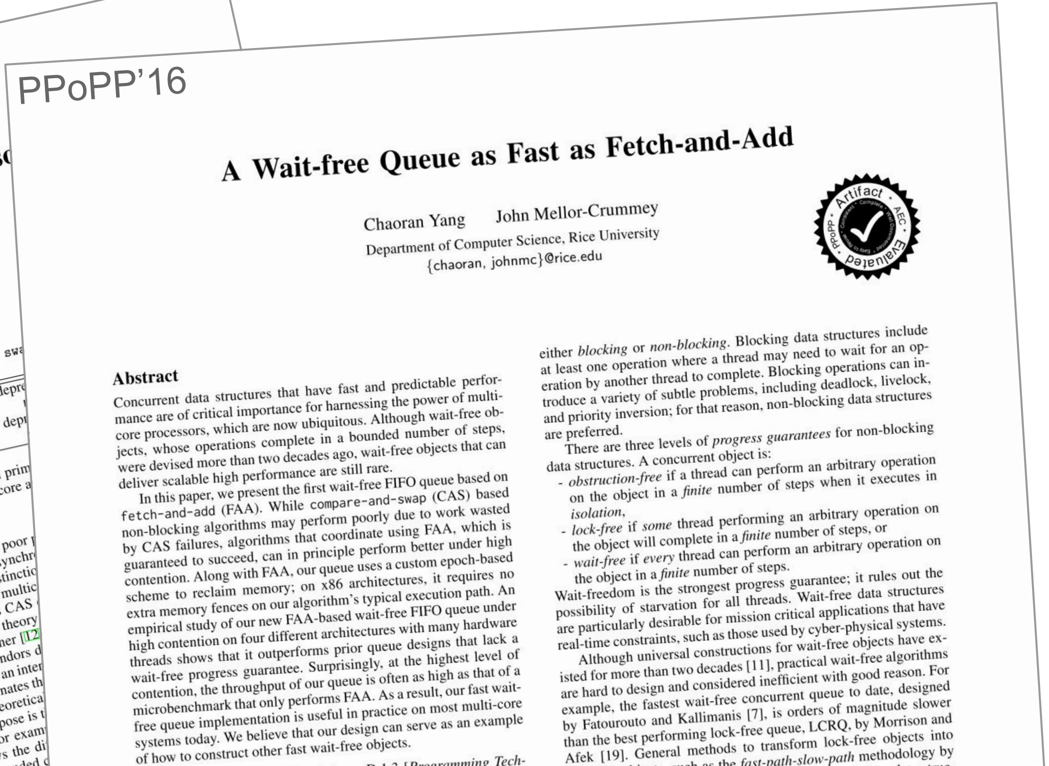
Adam Morrison Yehuda Afek
Blavatnik School of Computer Science, Tel Aviv University

	compare-and-swap
ARM	LL/SC
POWER	LL/SC
SPARC	yes
x86	**yes**

Table 1: Synchronization prim tions on dominant multicore a

Abstract
Conventional wisdom in designing concurrent data structures is to use the most powerful synchronization primitive, namely `compare-and-swap` (CAS), and to avoid contended hot spots. In building concurrent FIFO queues, this reasoning has led researchers to propose *combining-based* concurrent queues.
This paper takes a different approach, showing how to rely on `fetch-and-add` (F&A), a less powerful primitive that is available on x86 processors, to construct a *nonblocking (lock-free) linearizable concurrent FIFO queue* which, despite the F&A being a contended hot spot, outperforms combining-based implementations by 1.5× to 2.5× in all concurrency levels on an x86 server with four multicore processors, in both single-processor and multi-processor executions.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; E.1 [*Data Structures*]: Lists, stacks, and queues


PPoPP'16

A Wait-free Queue as Fast as Fetch-and-Add

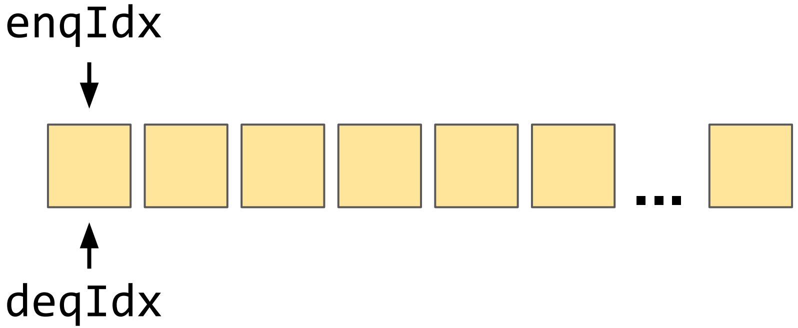
Chaoran Yang John Mellor-Crummey
Department of Computer Science, Rice University
{chaoran, johnmc}@rice.edu

Abstract
Concurrent data structures that have fast and predictable performance are of critical importance for harnessing the power of multi-core processors, which are now ubiquitous. Although wait-free objects, whose operations complete in a bounded number of steps, were devised more than two decades ago, wait-free objects that can deliver scalable high performance are still rare.
In this paper, we present the first wait-free FIFO queue based on `fetch-and-add` (FAA). While `compare-and-swap` (CAS) based non-blocking algorithms may perform poorly due to work wasted by CAS failures, algorithms that coordinate using FAA, which is guaranteed to succeed, can in principle perform better under high contention. Along with FAA, our queue uses a custom epoch-based scheme to reclaim memory; on x86 architectures, it requires no extra memory fences on our algorithm's typical execution path. An empirical study of our new FAA-based wait-free FIFO queue under high contention on four different architectures with many hardware threads shows that it outperforms prior queue designs that lack a wait-free progress guarantee. Surprisingly, at the highest level of contention, the throughput of our queue is often as high as that of a microbenchmark that only performs FAA. As a result, our fast wait-free queue implementation is useful in practice on most multi-core systems today. We believe that our design can serve as an example of how to construct other fast wait-free objects.

either *blocking* or *non-blocking*. Blocking data structures include at least one operation where a thread may need to wait for an operation by another thread to complete. Blocking operations can introduce a variety of subtle problems, including deadlock, livelock, and priority inversion; for that reason, non-blocking data structures are preferred.
There are three levels of *progress guarantees* for non-blocking data structures. A concurrent object is:
- *obstruction-free* if a thread can perform an arbitrary operation on the object in a *finite* number of steps when it executes in *isolation*,
- *lock-free* if *some* thread performing an arbitrary operation on the object will complete in a *finite* number of steps, or
- *wait-free* if *every* thread can perform an arbitrary operation on the object in a *finite* number of steps.

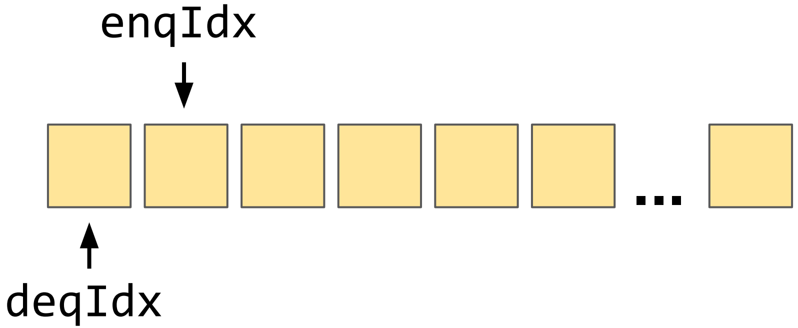
Wait-freedom is the strongest progress guarantee; it rules out the possibility of starvation for all threads. Wait-free data structures are particularly desirable for mission critical applications that have real-time constraints, such as those used by cyber-physical systems.
Although universal constructions for wait-free objects have existed for more than two decades [11], practical wait-free algorithms are hard to design and considered inefficient with good reason. For example, the fastest wait-free concurrent queue to date, designed by Fatourouto and Kallimanis [7], is orders of magnitude slower than the best performing lock-free queue, LCRQ, by Morrison and Afek [19]. General methods to transform lock-free objects into

Obstruction-Free Queue on Infinite Array

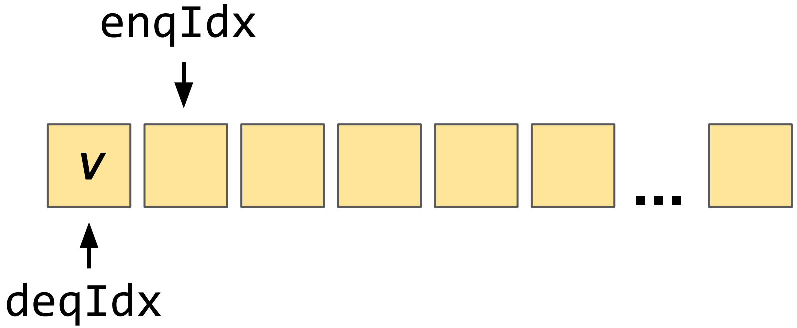


Бесконечный массив и указатели для `enqueue` и `dequeue`.
Сначала увеличиваем индекс, потом пишем/читаем

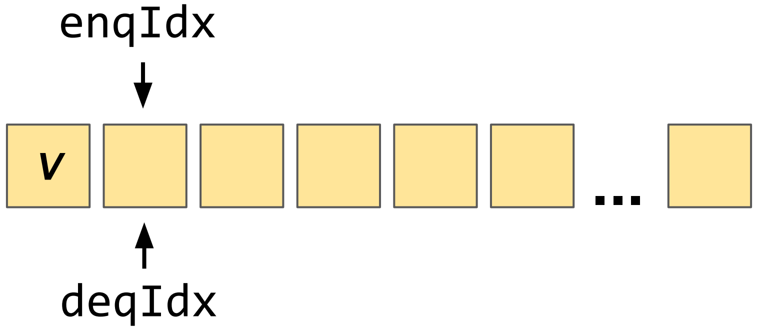
Obstruction-Free Queue on Infinite Array



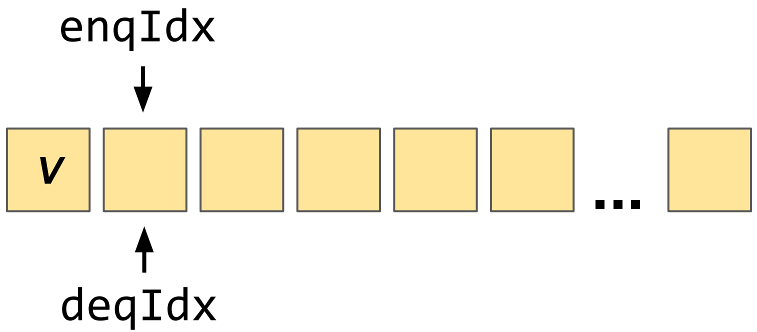
Obstruction-Free Queue on Infinite Array



Obstruction-Free Queue on Infinite Array

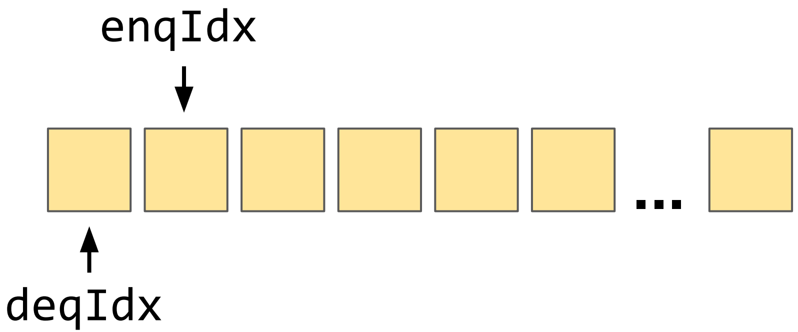


Obstruction-Free Queue on Infinite Array

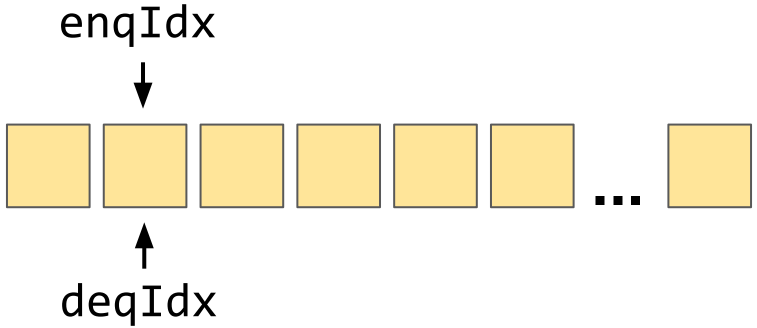


А если dequeu придёт читать раньше, чем произошла запись?

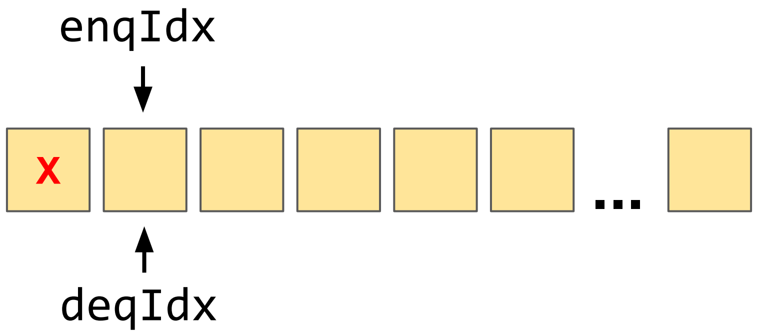
Obstruction-Free Queue on Infinite Array



Obstruction-Free Queue on Infinite Array

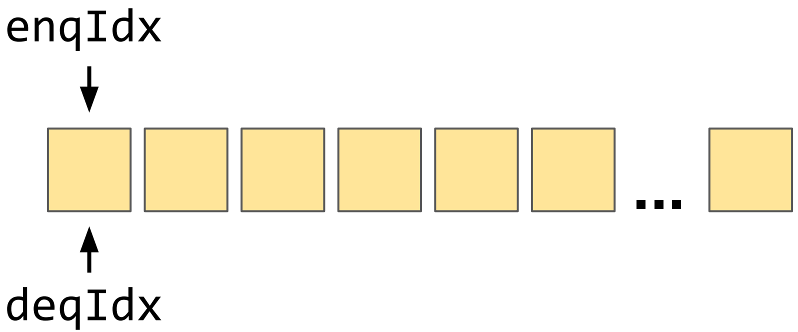


Obstruction-Free Queue on Infinite Array



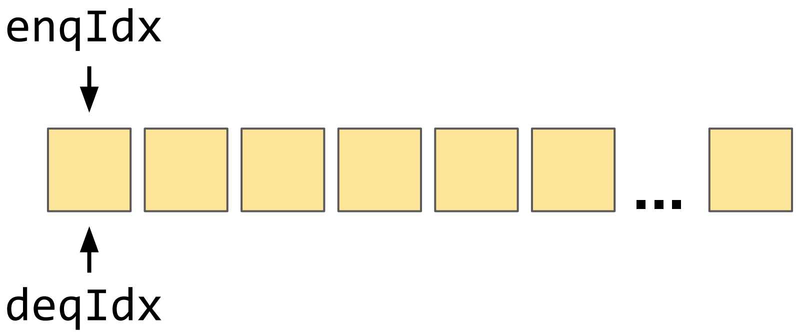
Пометим ячейку как “сломанную”,
обе операции начнутся заново

Obstruction-Free Queue on Infinite Array




```
fun enqueue(x: T)  = while (true) {
  val enqIdx = FAA(&enqIdx, 1)
  if (CAS(&data[enqIdx], null, x))
    return
}
```

Obstruction-Free Queue on Infinite Array



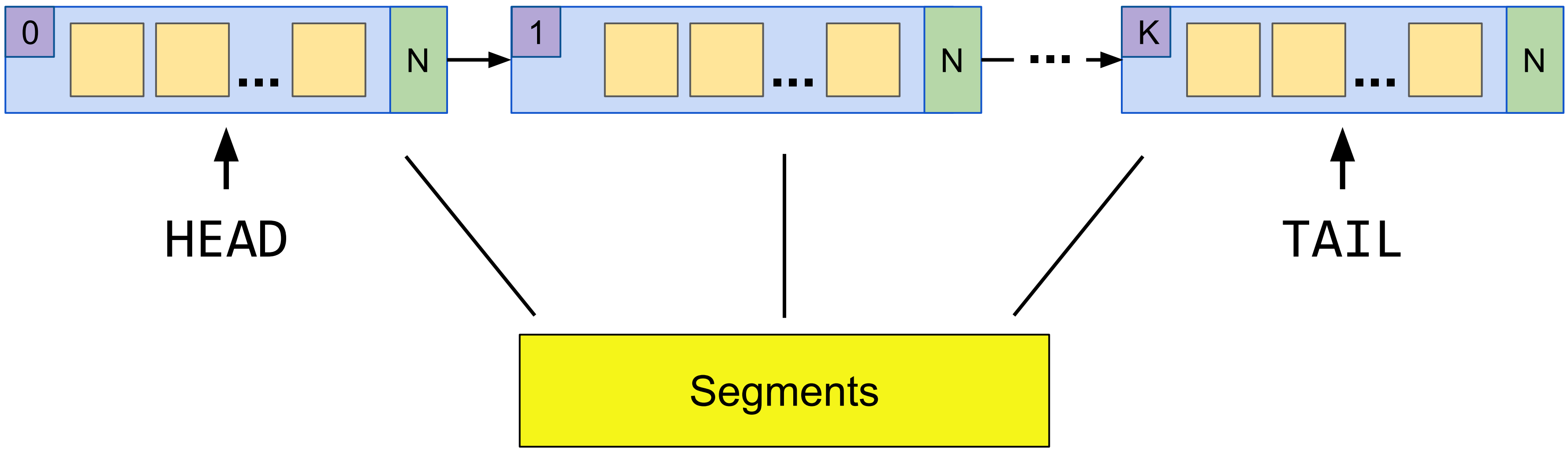

```
fun enqueue(x: T)  = while (true) {
  val enqIdx = FAA(&enqIdx, 1)
  if (CAS(&data[enqIdx], null, x))
    return
}
```

```
fun dequeue() = while (true) {
  if (isEmpty()) return null
  val deqIdx = FAA(&deqIdx, 1)
  val res = SWAP(&data[deqIdx], BROKEN)
  if (res == null) continue
  return res
}

fun isEmpty(): Boolean = deqIdx >= enqIdx
```

Lock-Free Queue on Infinite Array

Michael-Scott queue of segments



Lock-Free Queue on Infinite Array

```
fun enqueue(x: T) = while (true) {
  val tail = this.tail
  val enqIdx = FAA(&tail.enqIdx, 1)
  if (enqIdx >= NODE_SIZE) {
    // try to insert new node with “x”
  } else {
    if (CAS(&tail.data[enqIdx], null, x))
      return
  }
}
```

Lock-Free Queue on Infinite Array

```
fun enqueue(x: T) = while (true) {
  val tail = this.tail
  val enqIdx = FAA(&tail.enqIdx, 1)
  if (enqIdx >= NODE_SIZE) {
    // try to insert new node with “x”
  } else {
    if (CAS(&tail.data[enqIdx], null, x))
      return
  }
}
```

```
fun dequeue(): T = while (true) {
  val head = this.head
  val deqIdx = FAA(&head.deqIdx, 1)
  if (deqIdx >= NODE_SIZE) {
    val headNext = head.next ?: return null
    CAS(&this.head, head, headNext)
    continue
  }
  val res = SWAP(&head.data[deqIdx], BROKEN)
  if (res == null) continue
  return res
}
```

Flat Combining

Какие есть способы синхронизации?

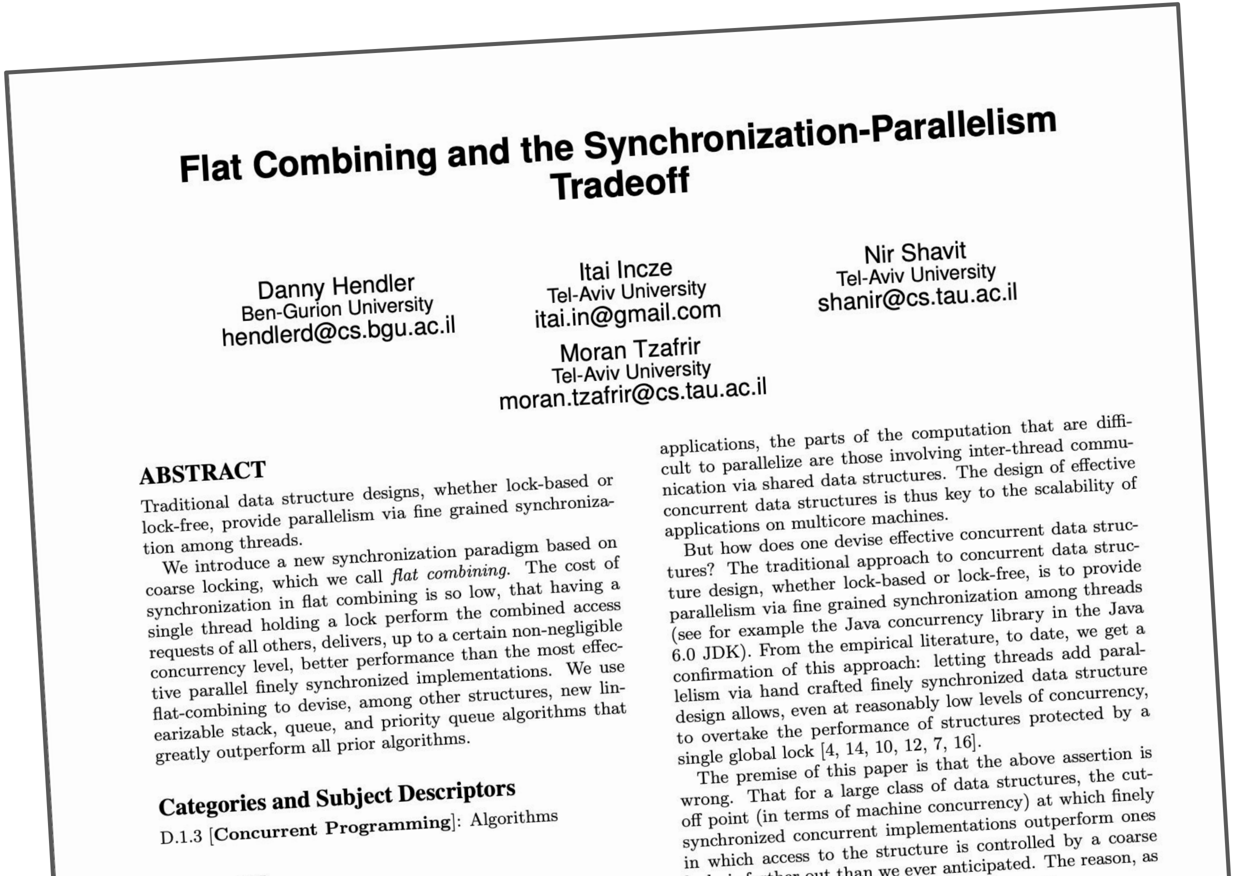
- Грубая блокировка -- просто писать, не масштабируется
- Тонкая блокировка -- сложнее, но и потенциально быстрее
- Lock-free / wait-free -- зачастую лучше, может очень хорошо масштабироваться
    - А может и нет, см. Michael-Scott Queue или Treiber Stack

Какие есть способы синхронизации?

- Грубая блокировка -- просто писать, не масштабируется
- Тонкая блокировка -- сложнее, но и потенциально быстрее
- Lock-free / wait-free -- зачастую лучше, может очень хорошо масштабироваться
  - А может и нет, см. Michael-Scott Queue или Treiber Stack

Хотим писать такой же простой код, как с грубой блокировкой, но быть существенно быстрее

Даёшь Flat Combining!



Flat Combining and the Synchronization-Parallelism Tradeoff

Danny Hendler
Ben-Gurion University
hendlerd@cs.bgu.ac.il

Itai Incze
Tel-Aviv University
itai.in@gmail.com

Nir Shavit
Tel-Aviv University
shanir@cs.tau.ac.il

Moran Tzafrir
Tel-Aviv University
moran.tzafrir@cs.tau.ac.il

ABSTRACT

Traditional data structure designs, whether lock-based or lock-free, provide parallelism via fine grained synchronization among threads.

We introduce a new synchronization paradigm based on coarse locking, which we call *flat combining*. The cost of synchronization in flat combining is so low, that having a single thread holding a lock perform the combined access requests of all others, delivers, up to a certain non-negligible concurrency level, better performance than the most effective parallel finely synchronized implementations. We use flat-combining to devise, among other structures, new linearizable stack, queue, and priority queue algorithms that greatly outperform all prior algorithms.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Algorithms

applications, the parts of the computation that are difficult to parallelize are those involving inter-thread communication via shared data structures. The design of effective concurrent data structures is thus key to the scalability of applications on multicore machines.

But how does one devise effective concurrent data structures? The traditional approach to concurrent data structure design, whether lock-based or lock-free, is to provide parallelism via fine grained synchronization among threads (see for example the Java concurrency library in the Java 6.0 JDK). From the empirical literature, to date, we get a confirmation of this approach: letting threads add parallelism via hand crafted finely synchronized data structure design allows, even at reasonably low levels of concurrency, to overtake the performance of structures protected by a single global lock [4, 14, 10, 12, 7, 16].

The premise of this paper is that the above assertion is wrong. That for a large class of data structures, the cutoff point (in terms of machine concurrency) at which finely synchronized concurrent implementations outperform ones in which access to the structure is controlled by a coarse ... further out than we ever anticipated. The reason, as

Flat Combining -- Basic Ideas

1. Операции выполняются последовательно, структура данных защищена блокировкой

2. Поток, который держит блокировку -- **комбайнер**, он выполняет свою операцию и операции других потоков

3. Те, кто не смог взять блокировку, публикуют свои операции для комбайнера и дожидаются или результата этой операции, или блокировки (тогда этот поток становится **комбайнером**)

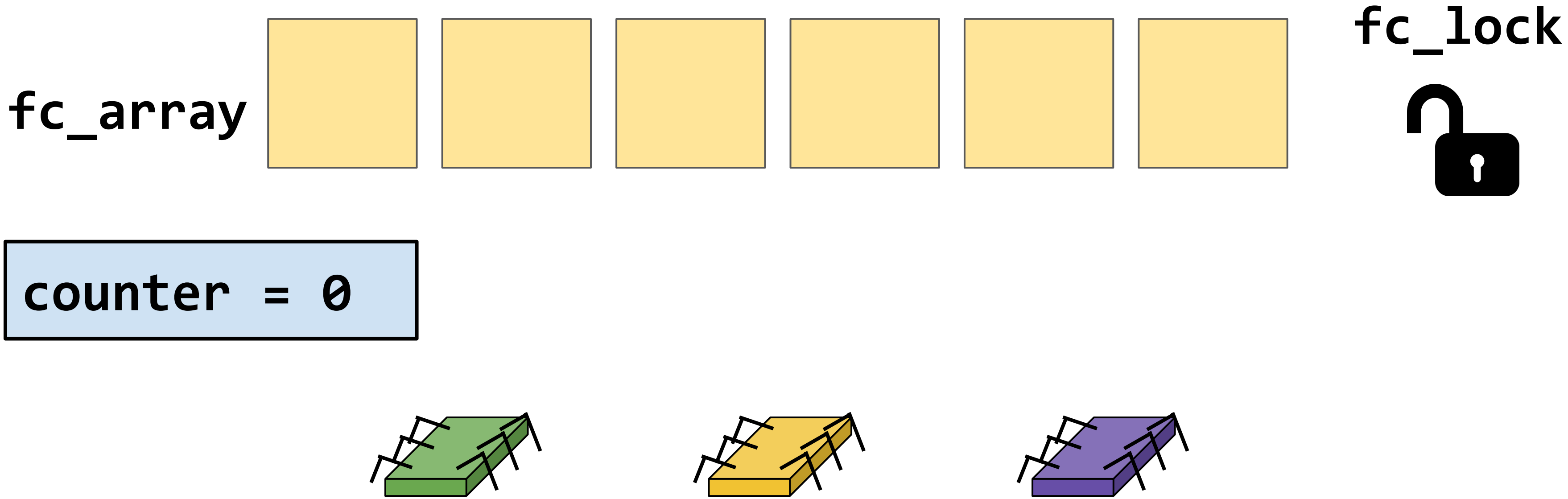

GS

Flat Combining -- пример работы

Поток не смог захватить блокировку => публикует операцию в этот массив и ждёт результат или блокировку

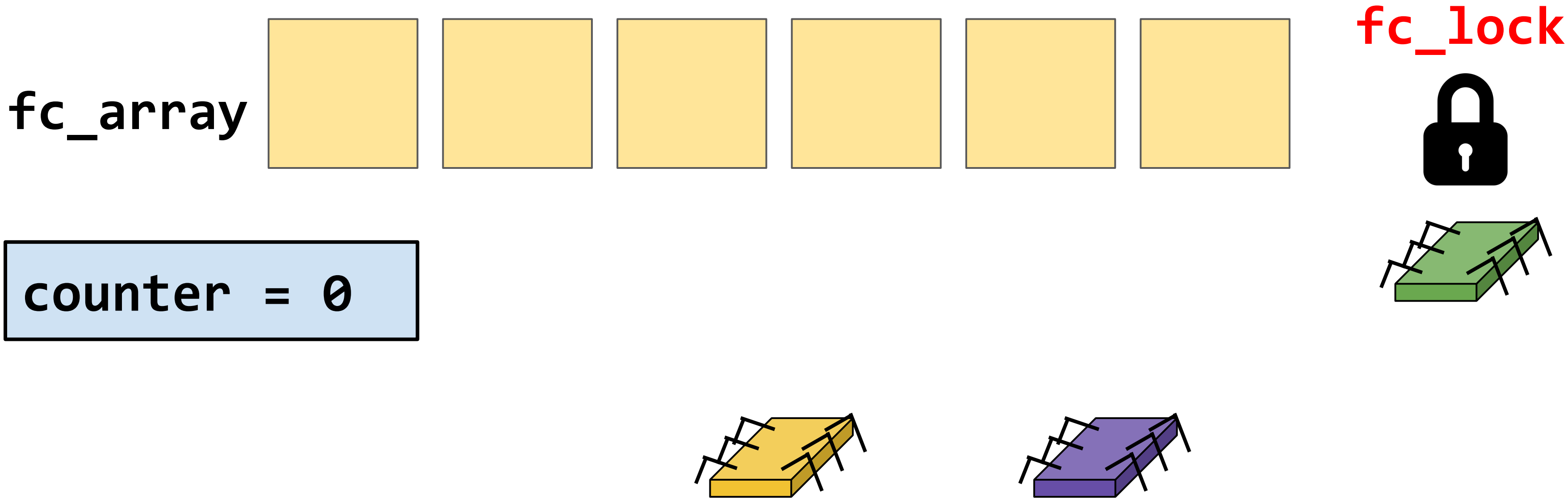
Эта блокировка обеспечивает эксклюзивный доступ к структуре данных

Flat Combining -- пример работы



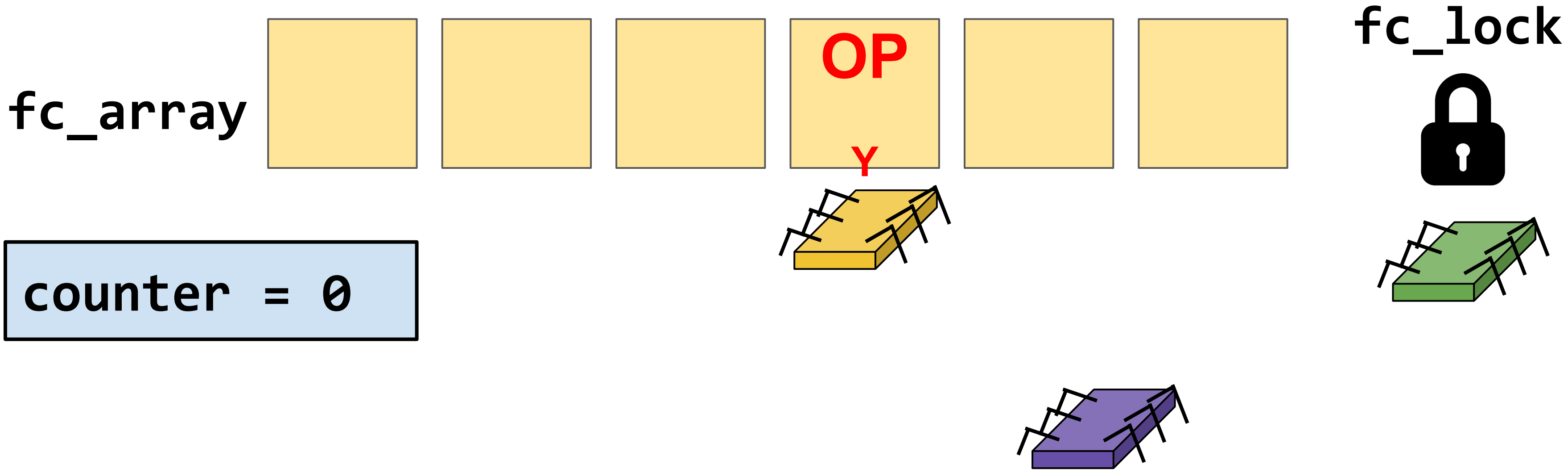
Будем делать счётчик. Три потока, никто ещё ничего не начал.

Flat Combining -- пример работы



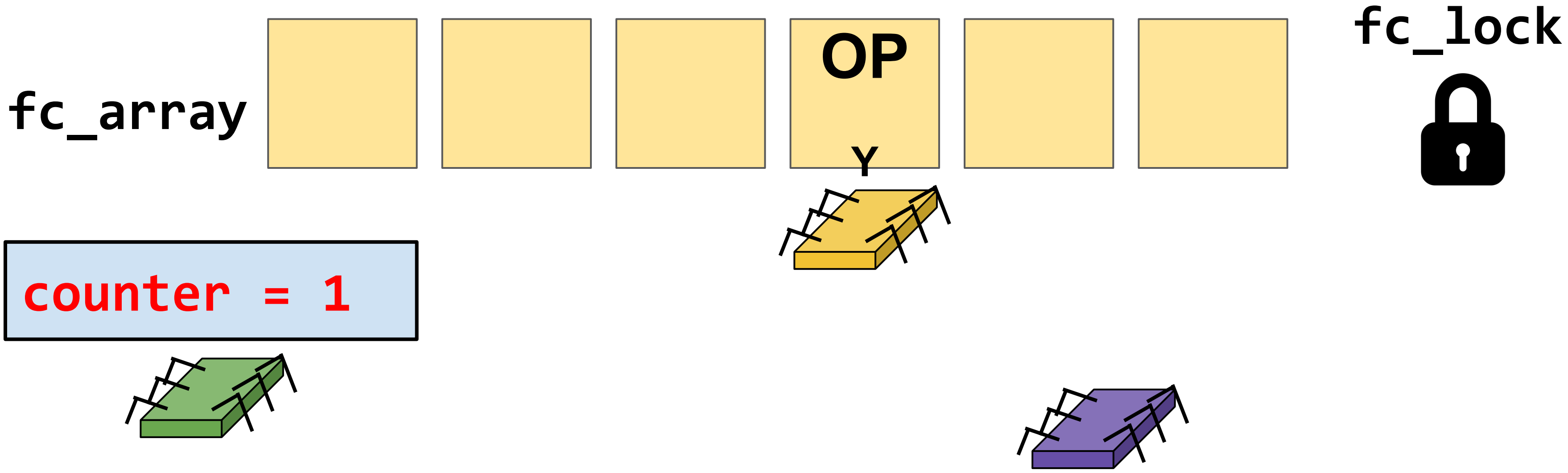
Зелёный начал операцию и успешно взял блокировку

Flat Combining -- пример работы



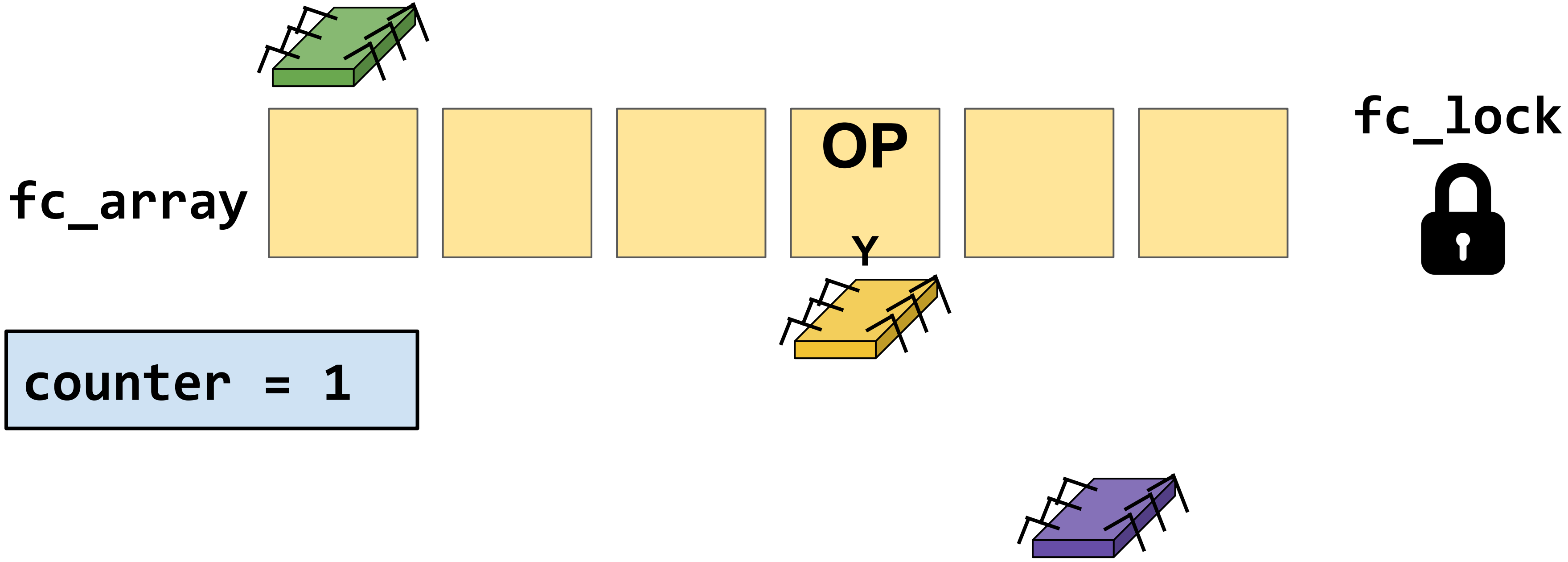
Жёлтый не сумел захватить блокировку
и опубликовал свою операцию

Flat Combining -- пример работы



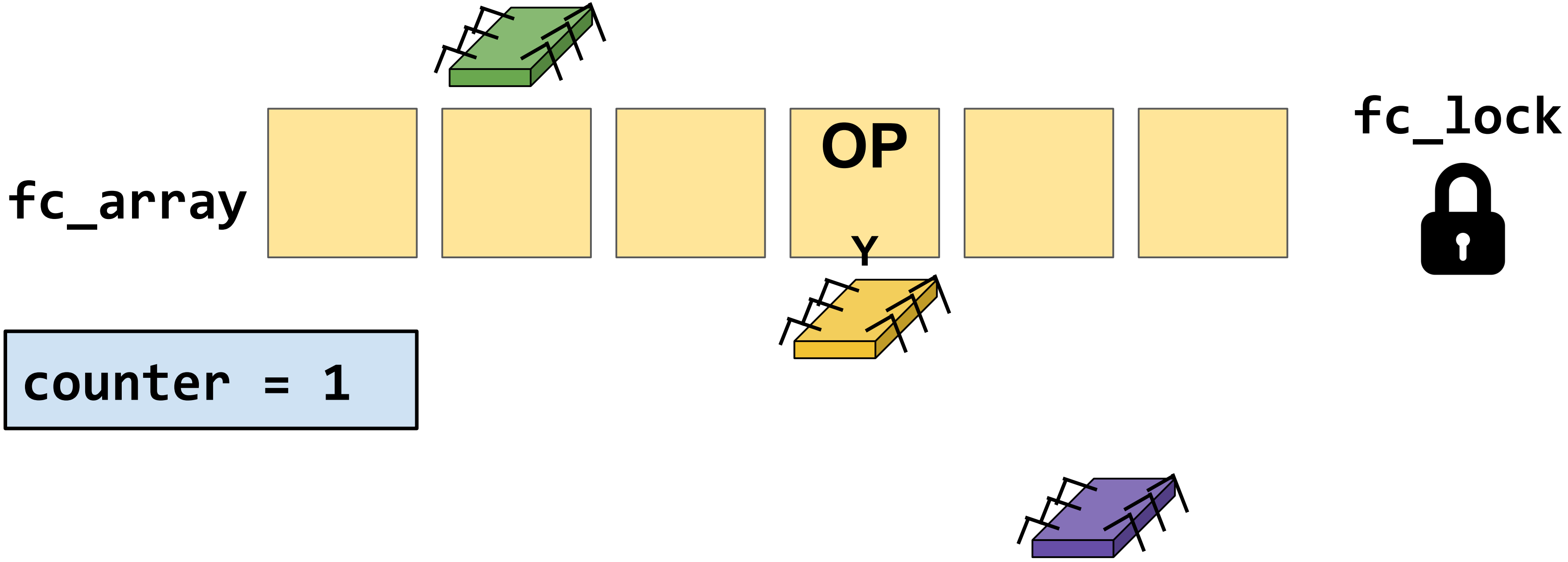
Жёлтый ждёт результат, а зелёный поменял счётчик

Flat Combining -- пример работы



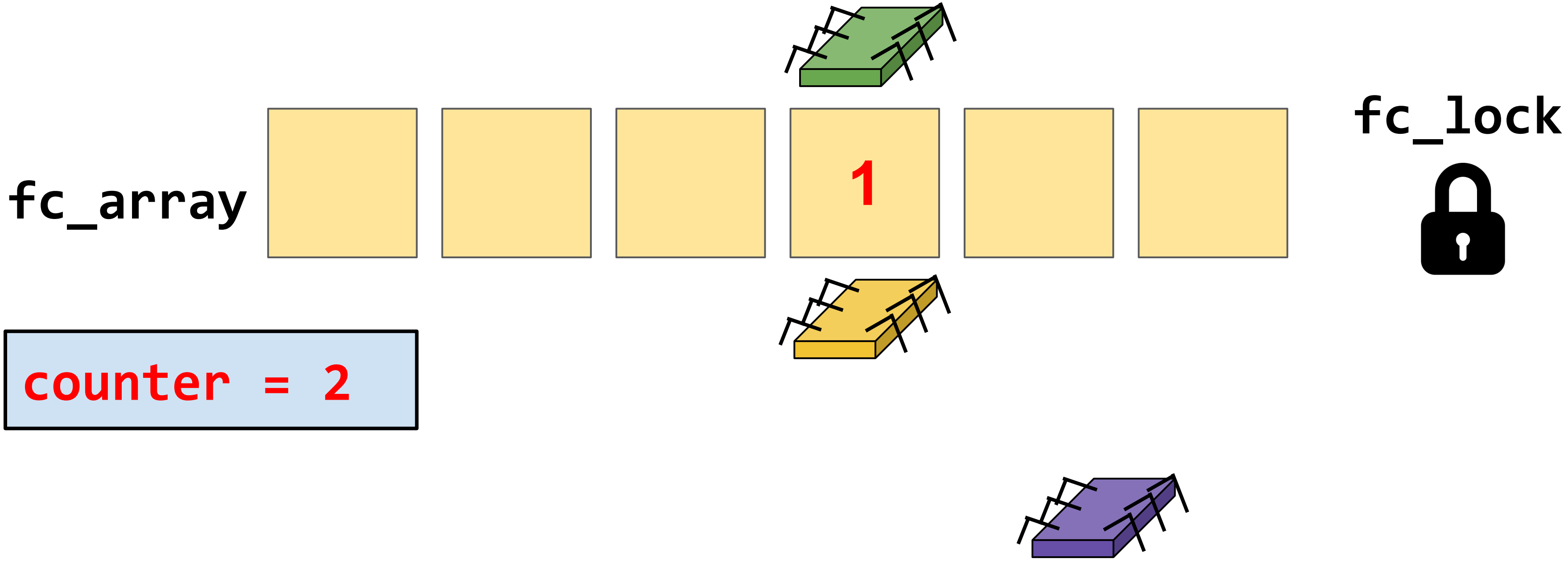
Жёлтый всё ждёт, а зелёный пошёл по массиву

Flat Combining -- пример работы



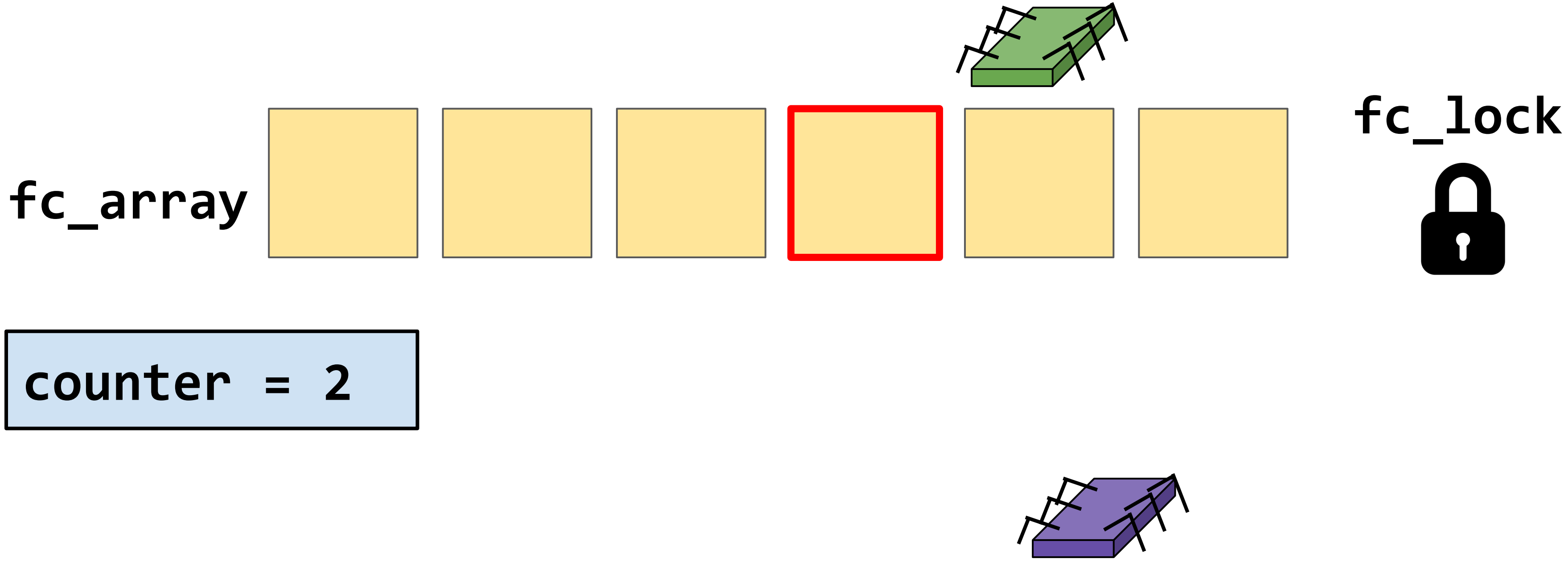
Жёлтый всё ждёт, зелёный всё идёт

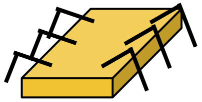
Flat Combining -- пример работы



Зелёный дошёл до жёлтого, ура!
Выполнил операцию и записал результат в ячейку массива.

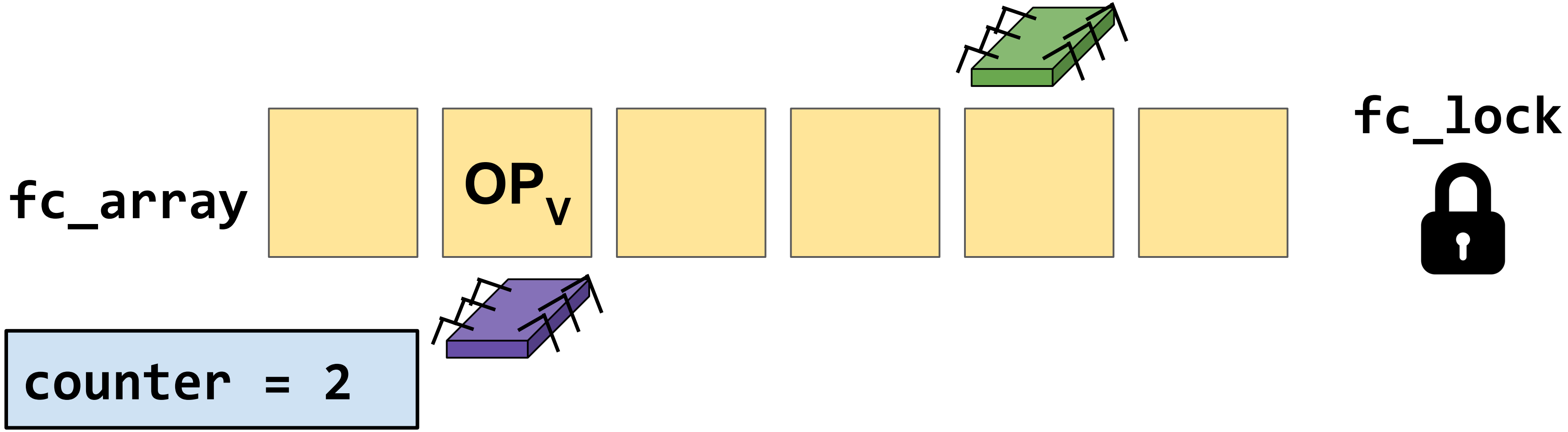
Flat Combining -- пример работы

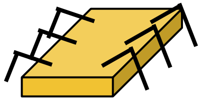




Зелёный пошёл дальше,
а жёлтый вернул результат и обнулил ячейку.

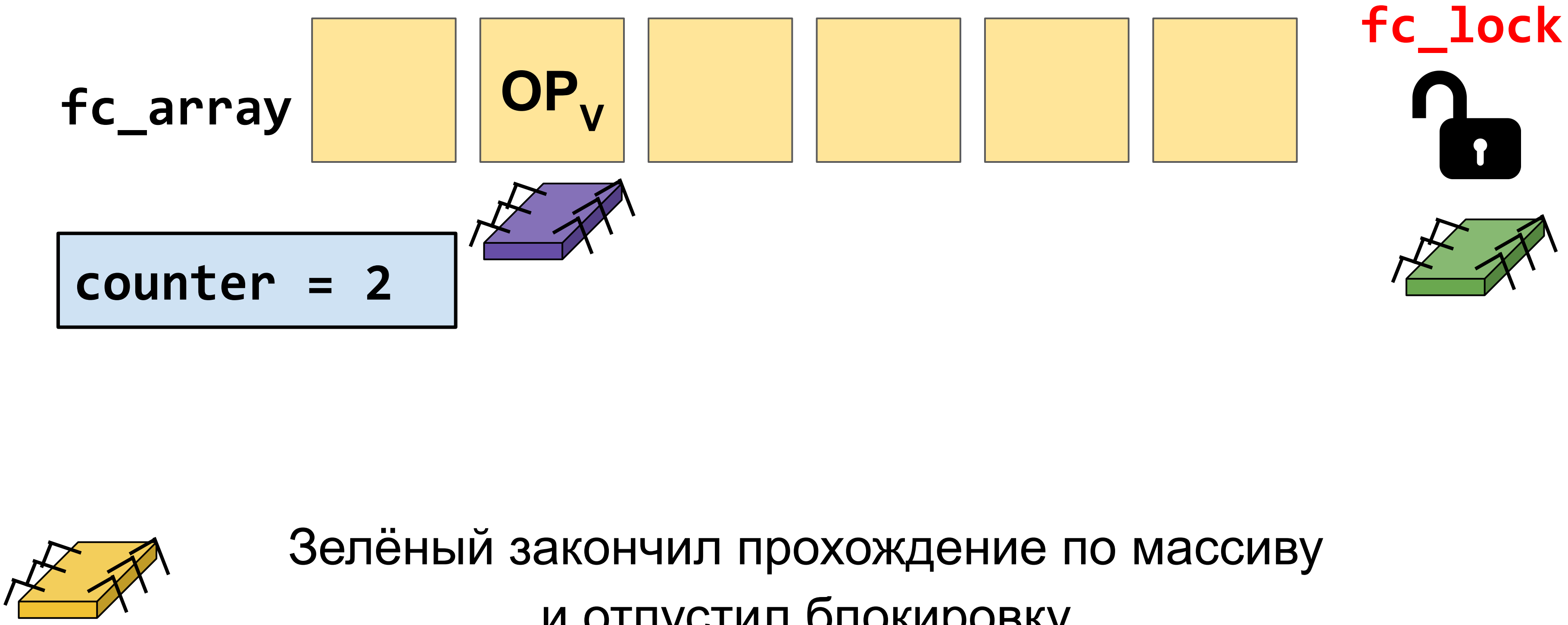
Flat Combining -- пример работы





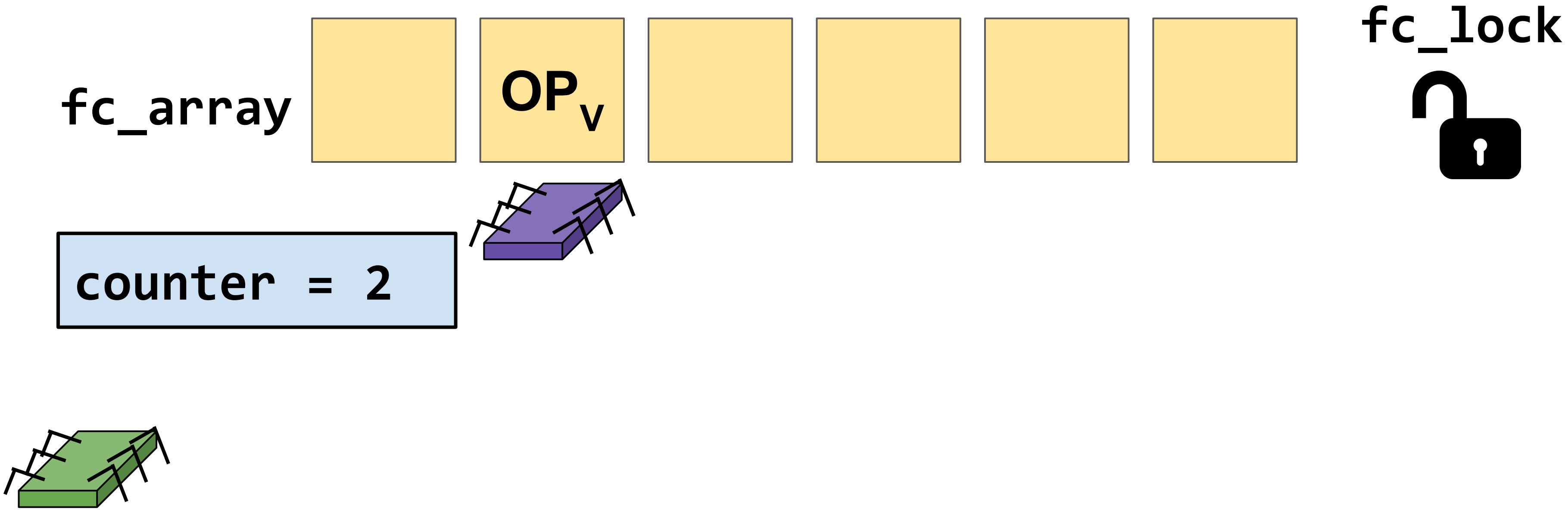
В дело вступает фиолетовый, он тоже не смог захватить блокировку и опубликовал операцию

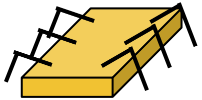
Flat Combining -- пример работы



Зелёный закончил прохождение по массиву
и отпустил блокировку

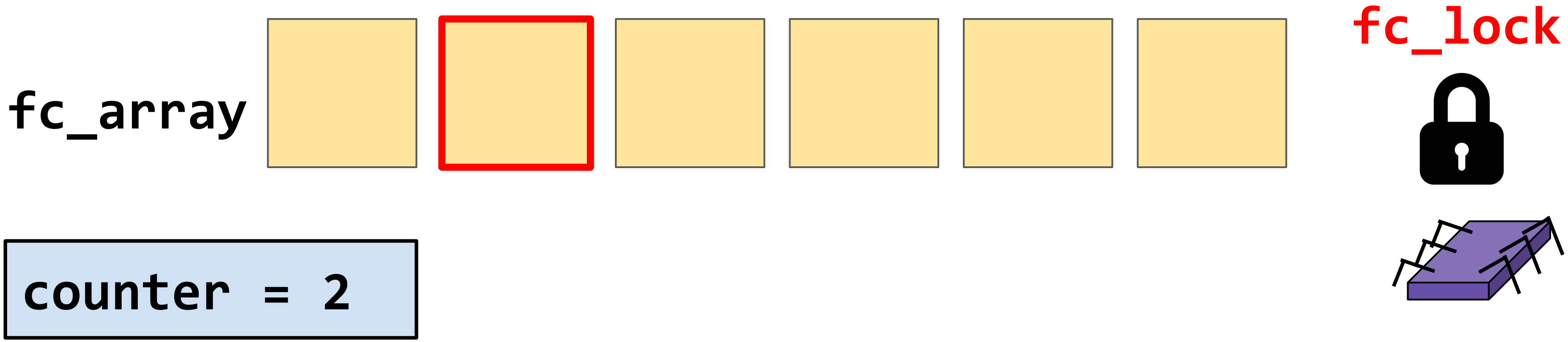
Flat Combining -- пример работы

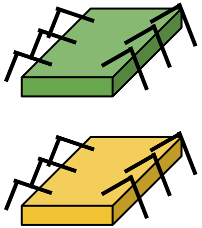




Теперь фиолетовый может захватить блокировку!

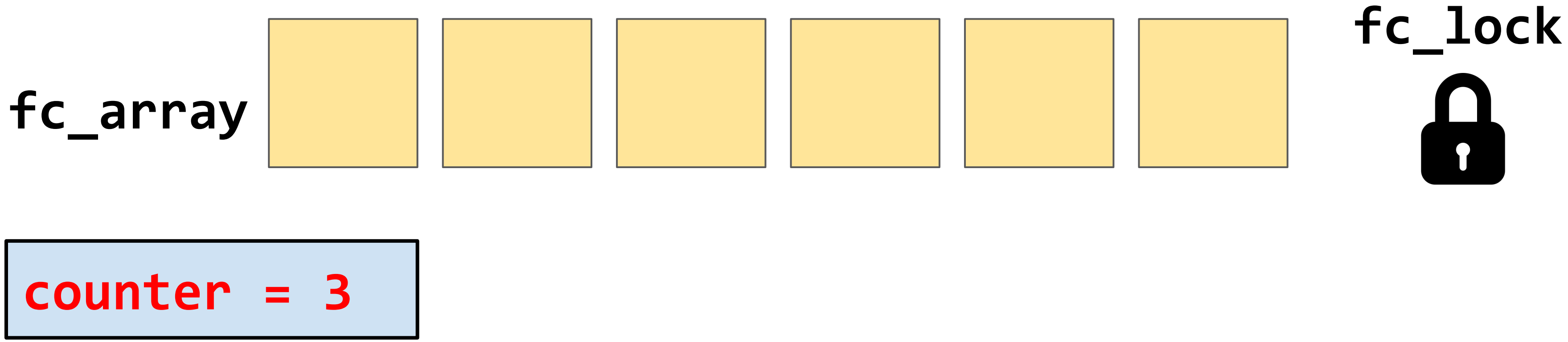
Flat Combining -- пример работы

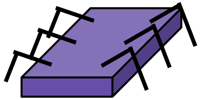


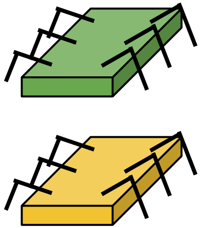


Фиолетовый захватывает блокировку
и очищает ячейку в массиве

Flat Combining -- пример работы





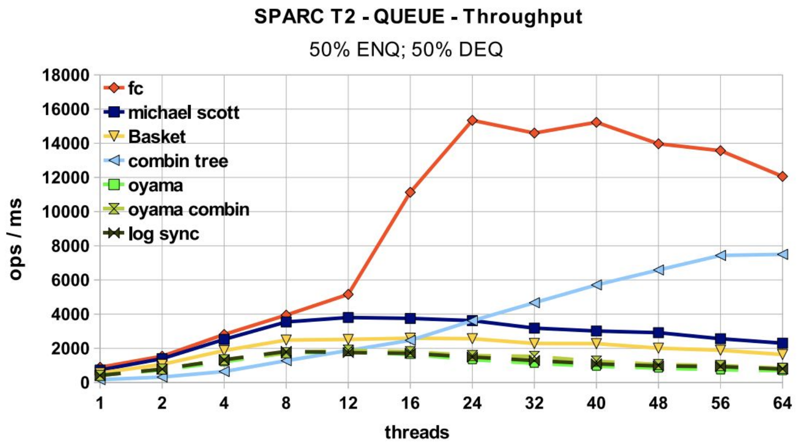


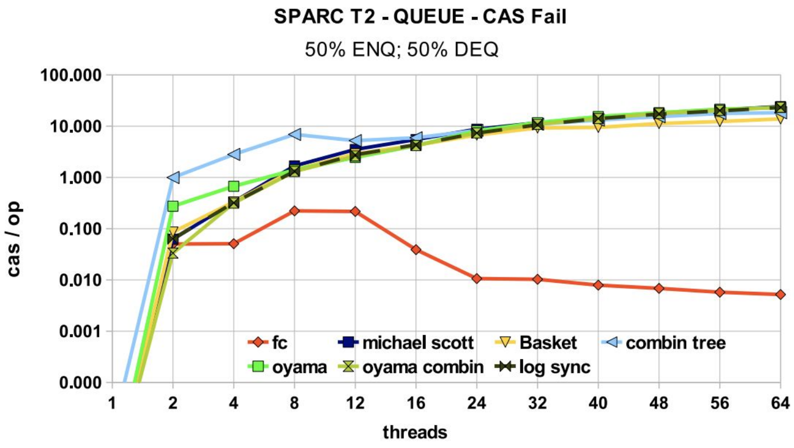
Теперь фиолетовый увеличивает счетчик

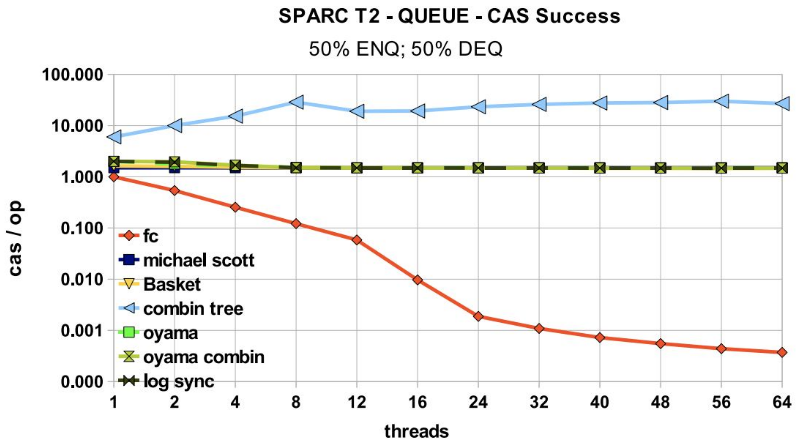
Flat Combining -- пример работы

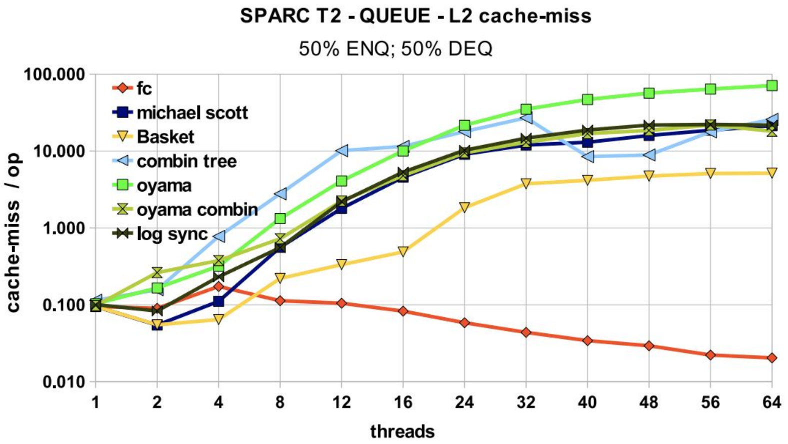
И в конце освобождает блокировку

FC Queue is Much Faster than MSQueue!

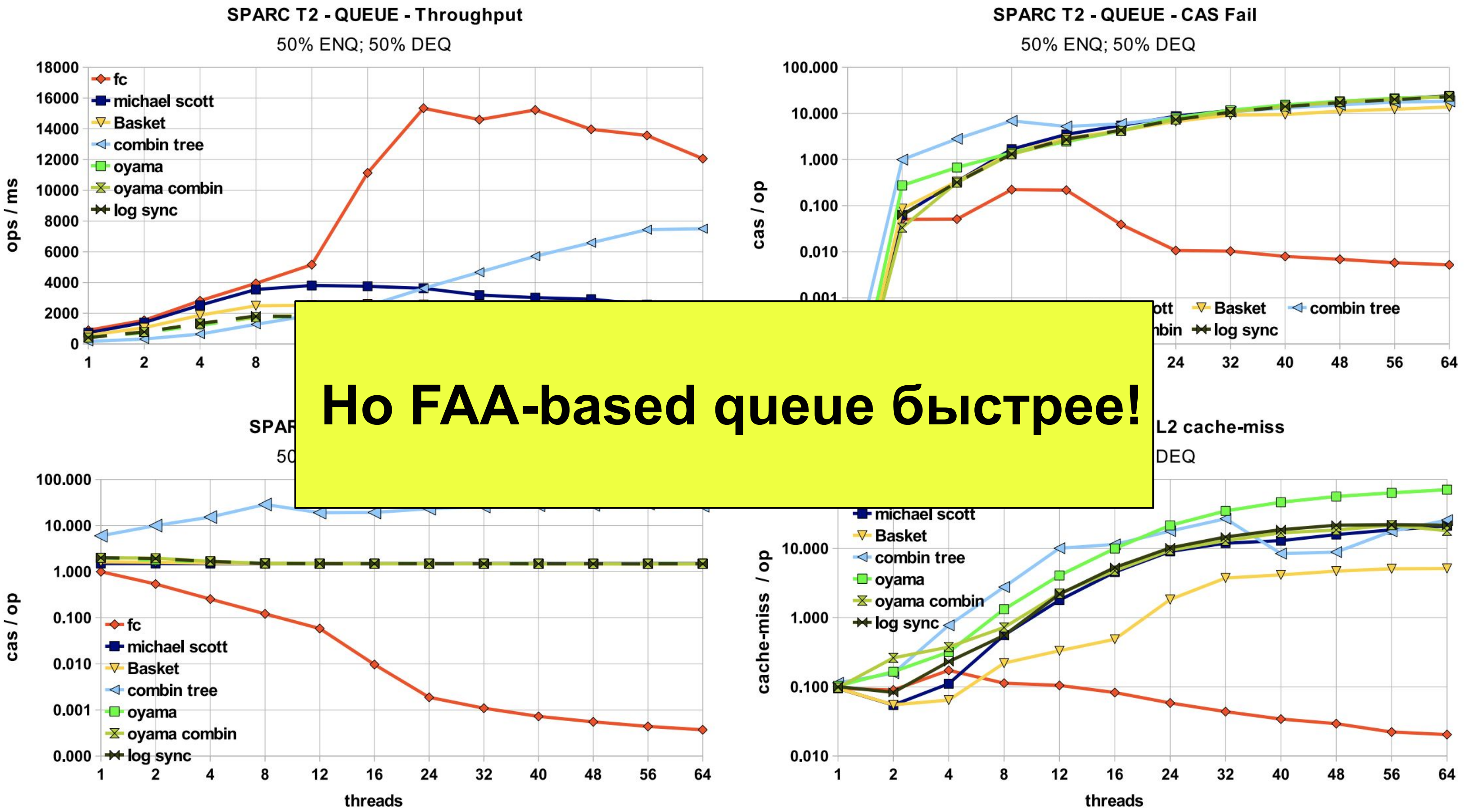








FC Queue is Much Faster than MSQueue!



Но FAA-based queue быстрее!

Flat Combining -- когда полезен?

- Когда производительности грубой/тонкой блокировки недостаточно, а с flat combining -- OK
- Когда последовательная версия структуры данных сильно проще и быстрее (например, priority queue)
- Когда алгоритм плохо масштабируется by design (например, очередь)
- Когда алгоритм может существенно быстрее выполнять запросы пачкой, а не по-отдельности (batch processing)

Flat Combining -- какие плюсы нахаляву?

- Cache Locality -- один поток на одном ядре выполняет несколько операций, нет инвалидаций кеша

- Очень быстрая блокировка -- можем написать вот так:

```
var locked = false
fun tryLock() = CAS(&lock, false, true)
fun unlock() { locked = false }
```