Concurrent collections .NET

Under the hood

Dmitry Ivanov, JetBrains

Чем мы займемся сегодня?

1. Теорией неблокирующих алгоритмов
2. Иммутабельностью
3. Дизайном и реализацией thread-safe коллекций

А оно точно надо?

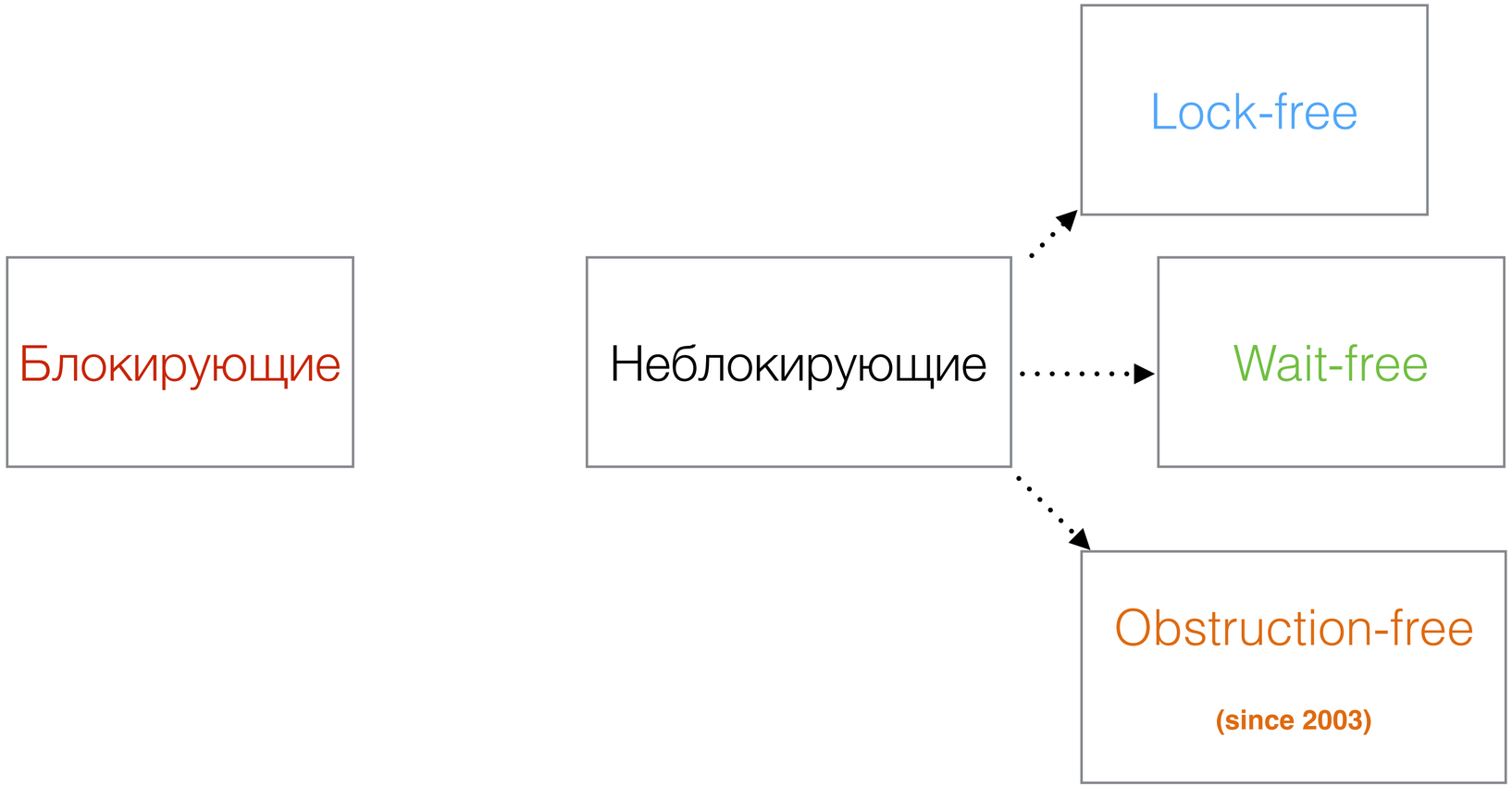
Не точно

Зачем?

1. Стать самым крутым параллельным человеком в компании
2. Написать, пожалуй =), самую быструю в мире неблокирующую очередь
3. Понять как устроен этот многопоточный мир

I. Теория

Алгоритмы бывают…



Non-blocking

```
long L;
long acc;

…

                                                    new    expected old   actual old
while (Interlocked.CompareExchange(ref L, 1,  0) == 0);

acc++;

Interlocked.Exchange(ref L, 0);
```

Non-blocking

```
long L;
long acc;

…


//CAS = Compare-and-swap                                  new   expected old   actual old
while (Interlocked.CompareExchange(ref L, 1,  0) == 0);

acc++;

Interlocked.Exchange(ref L, 0);
```

Non-blocking

```
long L;
long acc++;

…


//CAS = Compare-and-swap
while (!CAS(ref L, 1,  0));
                      new  expected
                             old
acc++;

Interlocked.Exchange(ref L, 0);
```

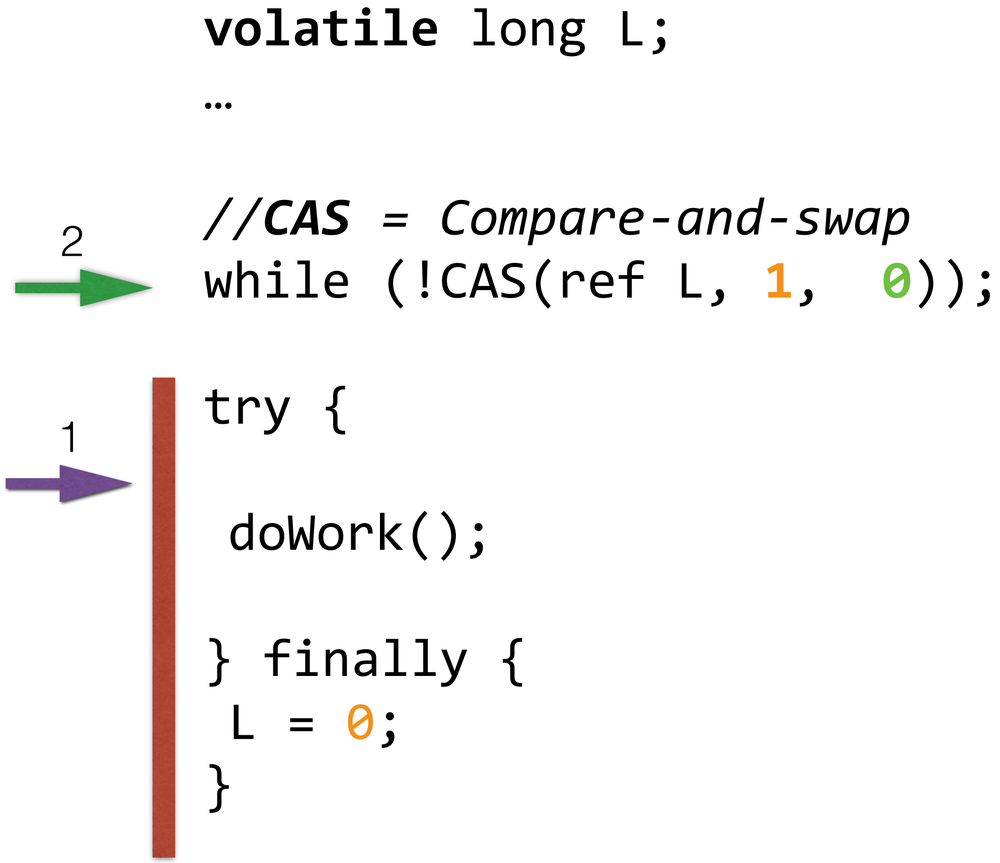
Non-blocking

```
long L;
long acc++;

…


//CAS = Compare-and-swap
while (!CAS(ref L, 1,  0));
                      new  expected
acc++;                       old

//TAS = Test-and-set
Interlocked.Exchange(ref L, 0);
```

Non-blocking

```
volatile long L;
…

//CAS = Compare-and-swap
while (!CAS(ref L, 1,  0));

try {

 doWork();

} finally {
 L = 0;
}
```

Non-blocking

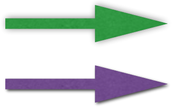


Non-blocking

An algorithm is called **non-blocking**
if failure or suspension
of **any thread**
cannot cause failure or suspension
of **another thread**

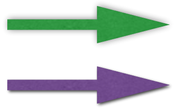
Three algorithms:
Blocking

```
long acc = 0;
object l;

…

lock (l) {
  acc++;
}
```



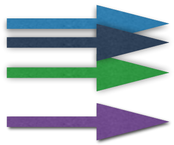
Three algorithms:
Lock-free

```
long acc = 0;


…

do {
  int old = acc;
} while (!CAS(ref acc, old + 1, old))
```



Three algorithms:
Lock-free

```
long acc = 0;


…

do {
  int old = acc;
} while (!CAS(ref acc, old + 1, old))
```



Three algorithms:
Wait-free

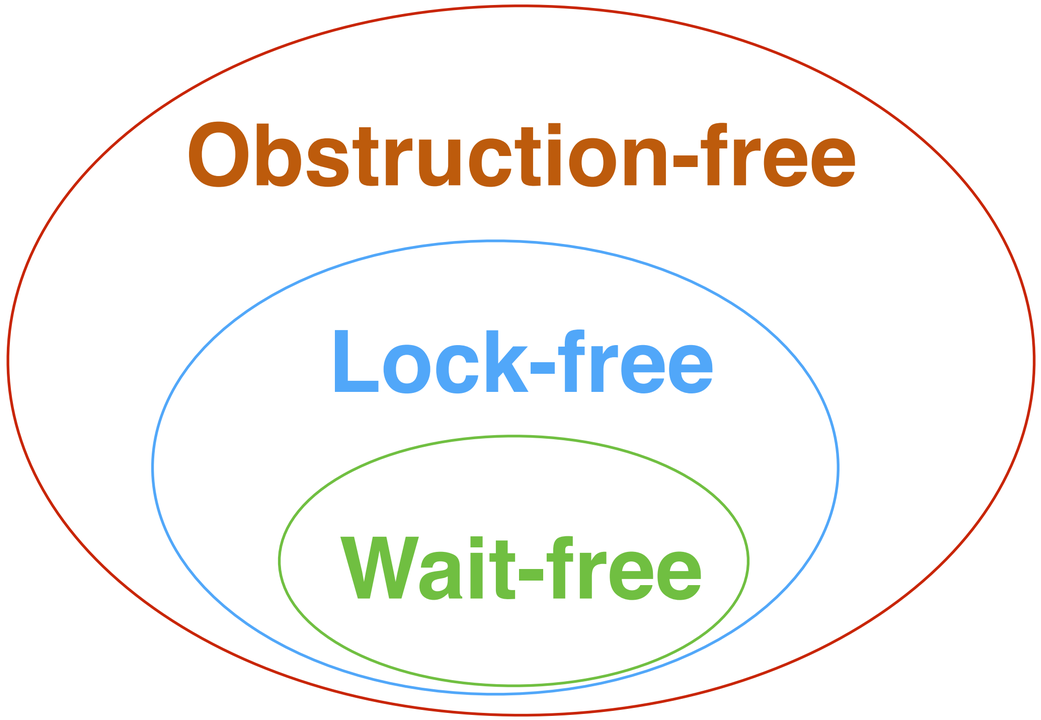
```
long acc = 0;


…

Interlocked.Increment(ref acc);
```

Bonus track:
Obstruction-free

M. Herlihy and company (2003),
Obstruction-Free Synchronization:
Double-Ended Queues as an Example

An algorithm is **obstruction-free**
if at any point,
a single thread executed **in isolation**
for a **bounded** number of steps
will complete its operation.

Non-blocking algorithms



progress in isolation

global progress

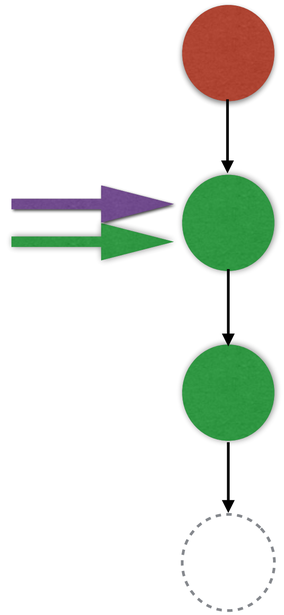
local progress

II. Иммутабельность

Immutable collections

System.Collections.Immutable

- Stack, Queue
- List
- Dictionary (+ Sorted)
- Set (+Sorted)

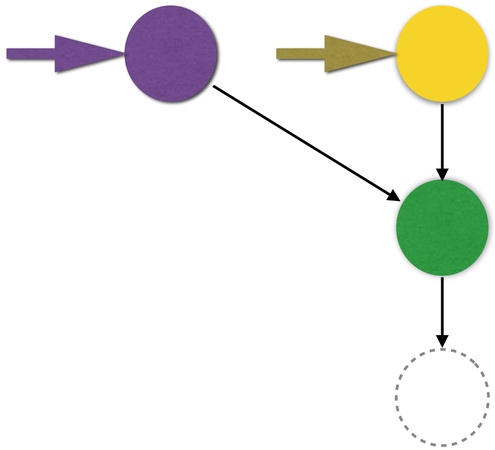


Immutable collections

System.Collections.Immutable

Thread safe

Not linearizable

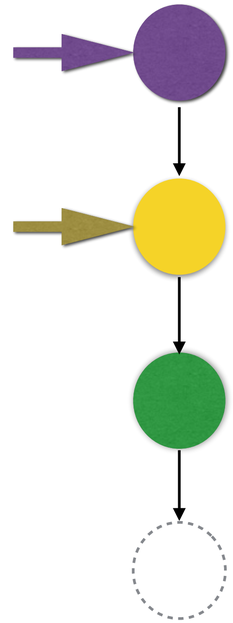


Immutable collections

System.Collections.Immutable

Thread safe

Not linearizable



Could be linearized by lock-free algorithm

Back to **lock-free**

```
long acc = 0;


…

do {
  int old = acc;
} while (!CAS(ref acc, old + 1, old))
```

Back to **lock-free**

```
SomeImmutableDataStructure ds = <initial>;

…

do {
  int old = ds;
  var new = transform(old);
} while (!CAS(ref ds, new, old))
```

Lock-free stack

```
class LockFreeStack<T> {

 ImmutableStack<T> s = <initial>;

 T Pop() {
  do {
   var old = s;
   var new = old.Popped(); //new stack created!
  } while (!CAS(ref s, new, old))

  return old.Top()
 }
 …
}
```

Lock-free queue

```
ImmutableQueue<T> q = <initial>;
```

```
T Dequeue()
{
 do {
  var old = s;
  var new = old.Dequeue(); //new queue created!
 } while (!CAS(ref q, new, old))

 return old.Peek()
}
```

```
Enqueue(T v)
{
 do {
  var old = s;
  var new = old.Enqueue(v); //new queue created!
 } while (!CAS(ref q, new, old))
}
```

III. Практика

Data structures evolution

.NET Framework < 4.0

Synchronized

- `static Synchronized() (e.g. ArrayList.Synchronized())`
  - `ICollection.SyncRoot`

- `SynchronizedCollection<T>`
  - `System.ServiceModel.dll`
  - `Only in .NET Framework`

Data structures evolution

.NET Framework >= 4.0 (.netcore, .netstandard)

Concurrent

- ConcurrentStack **lock-free**
- ConcurrentQueue **lock-free**
- ConcurrentBag
- ConcurrentDictionary (reading is **lock-free**)

Data structures evolution

.NET Framework >= 4.0 (.netcore, .netstandard)

Concurrent

- ConcurrentStack **lock-free**
- ConcurrentQueue **lock-free**
- ConcurrentBag
- ConcurrentDictionary

IProducerConsumerCollection

Always better?

NO (if we talk about performance)

Synchronised *vs* **Concurrent**

- How many producers and consumers?
- Are producers/consumers pure?
- Time between requests
- Number of CPUs

MEASURE YOUR LOAD

API Design

Producer-consumer collections

- ConcurrentStack **lock-free**
- ConcurrentQueue **lock-free**
- ConcurrentBag

```
IProducerConsumerCollection<T> :
  IEnumerable<T>

bool TryAdd(T)

bool TryTake(out T)

T[] ToArray()
CopyTo(T[], offset)
```

API Design

Producer-consumer collections

- ConcurrentStack **lock-free**
- ConcurrentQueue **lock-free**
- ConcurrentBag

```
IProducerConsumerCollection<T> :
  IEnumerable<T>

  bool TryAdd(T)

  bool TryTake(out T)
```

```
BlockingCollection<T> (maxsize, pcCollection)

  void Add(T)

  void Take(out T)
```

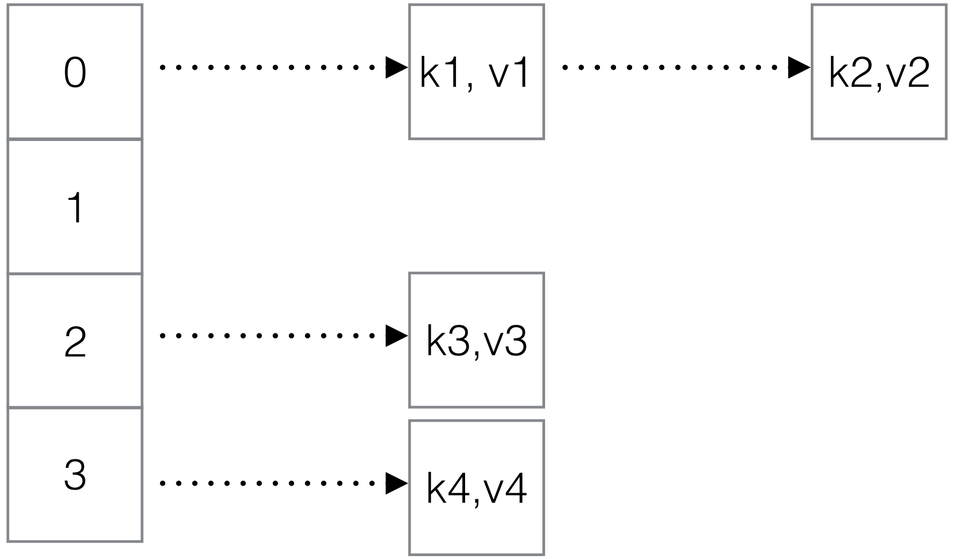
API Design: ConcurrentDictionary

```
ConcurrentDictionary<K,V> : IDictionary<K,V>

  bool TryAdd(K, V)

  bool TryUpdate(K, V)

  bool AddOrUpdate(K, V, Func<K, V, V>)


  bool GetOrAdd(K, V)
```

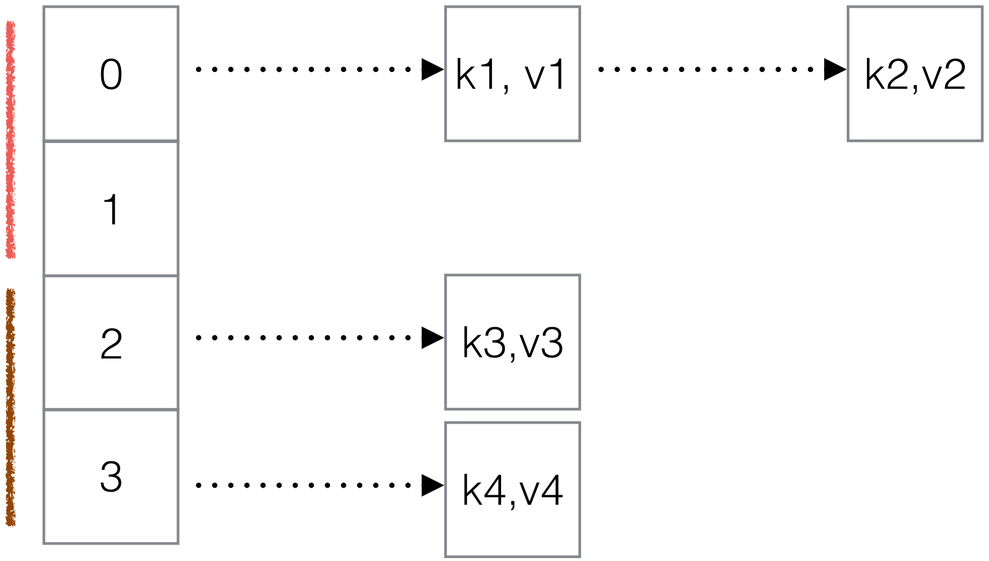
API Design: ConcurrentDictionary

```
ConcurrentDictionary<K,V> : IDictionary<K,V>

  bool TryAdd(K, V)

  bool TryUpdate(K, V)

  bool AddOrUpdate(K, V, Func<K, V, V>)
  bool AddOrUpdate(K, Func<K, V>, Func<K, V, V>)

  bool GetOrAdd(K, V)
  bool GetOrAdd(K, Func<K, V>)
```

ConcurrentDictionary

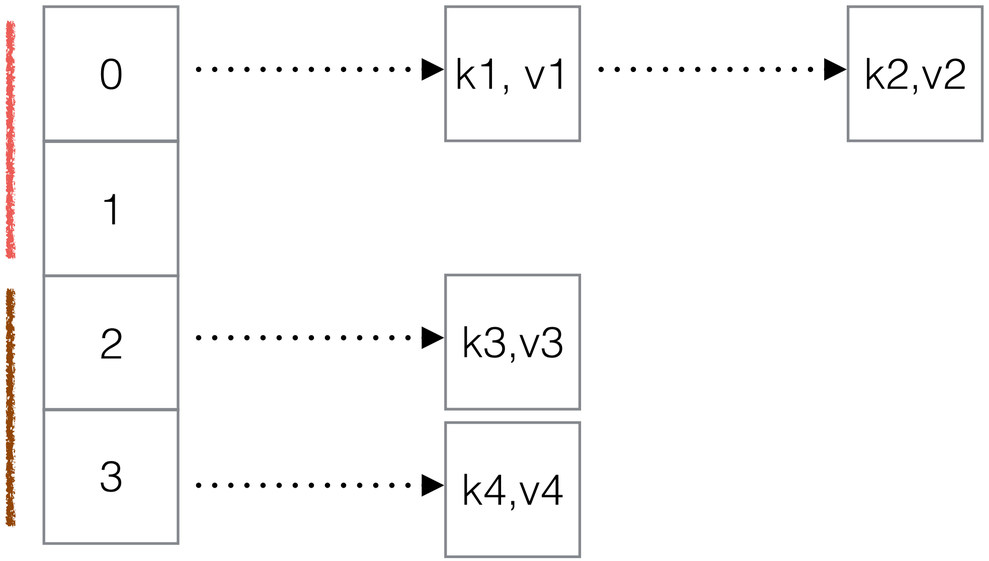


ConcurrentDictionary



granular locking

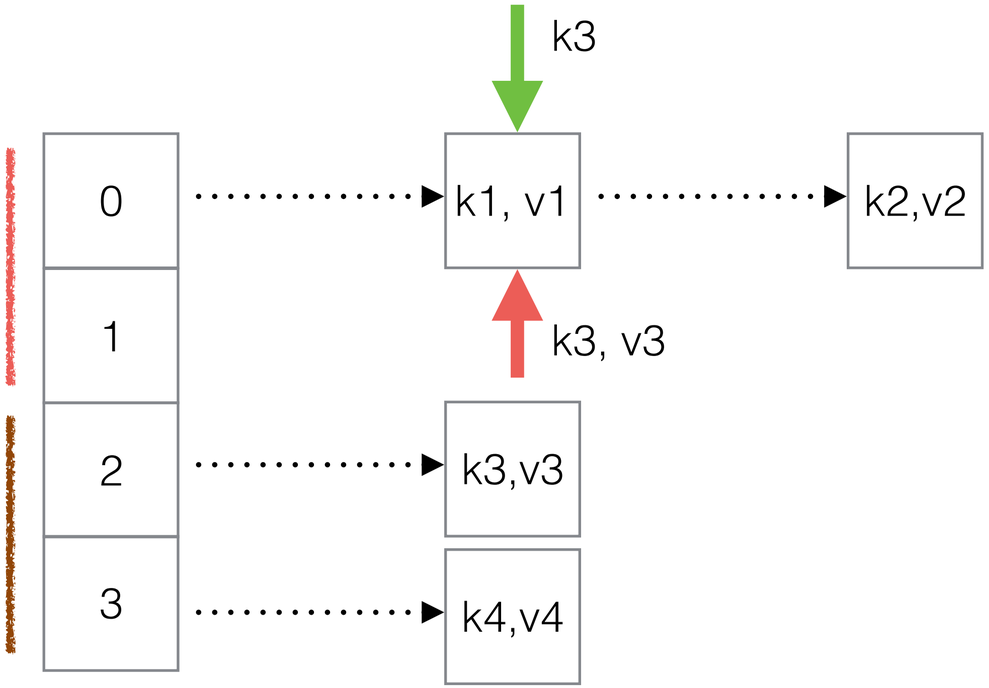
ConcurrentDictionary



All locks operations examples:
Grow,
ToArray,
Values,
Count

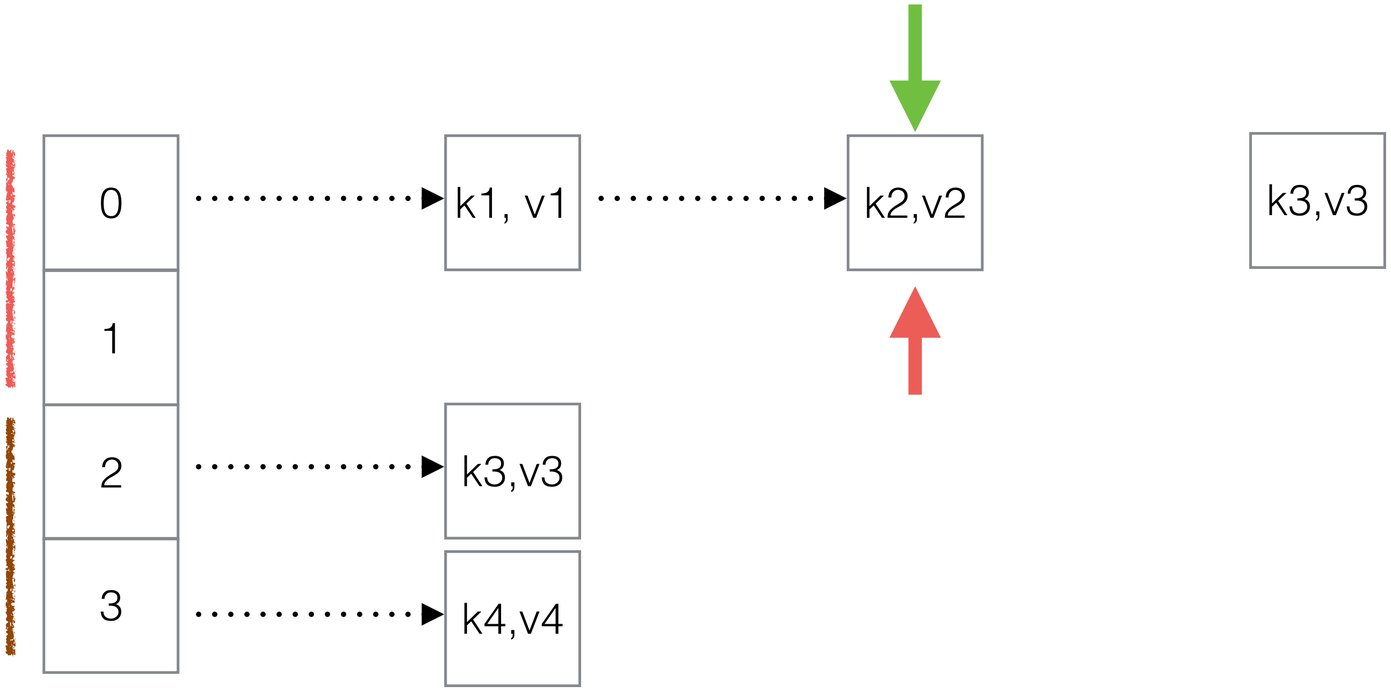
granular locking

ConcurrentDictionary



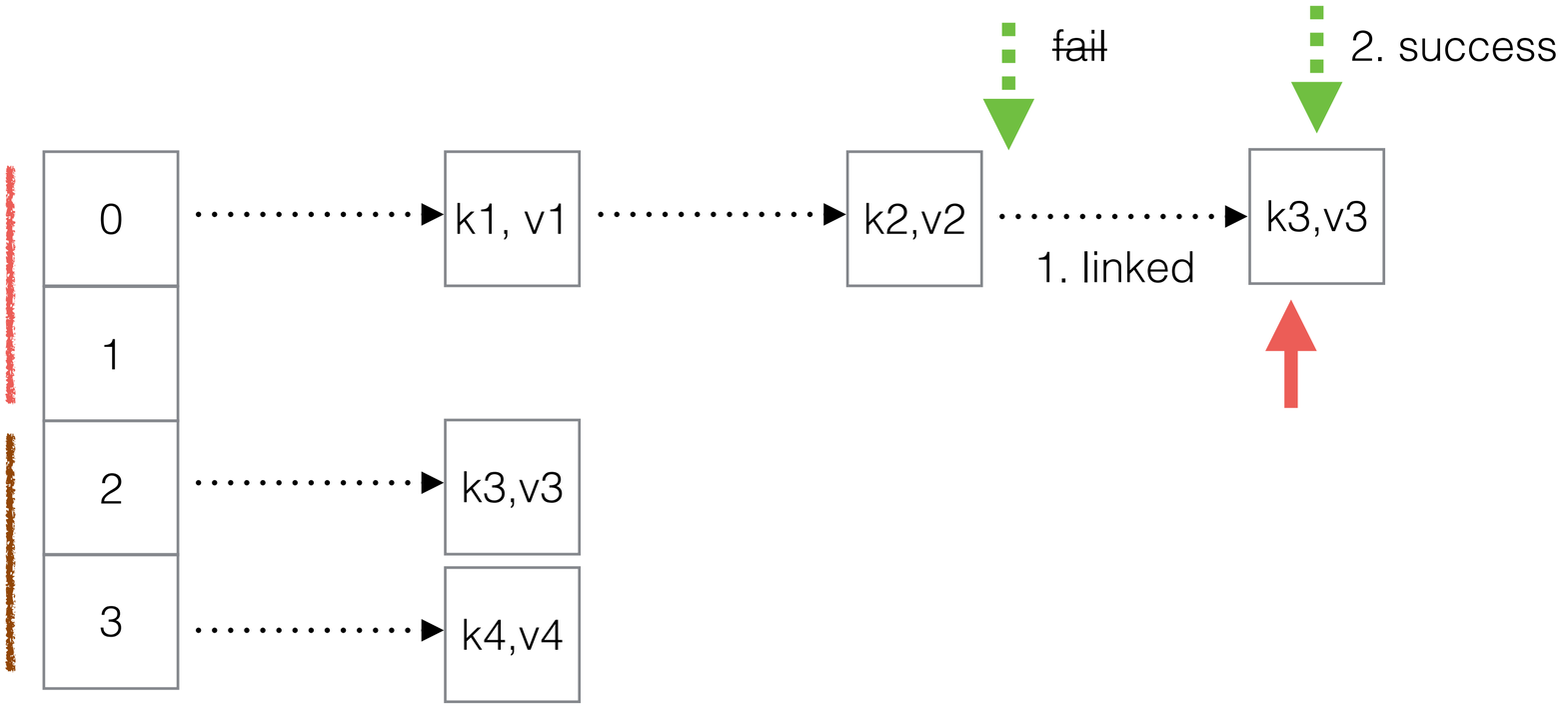
reading could be lock-free

ConcurrentDictionary



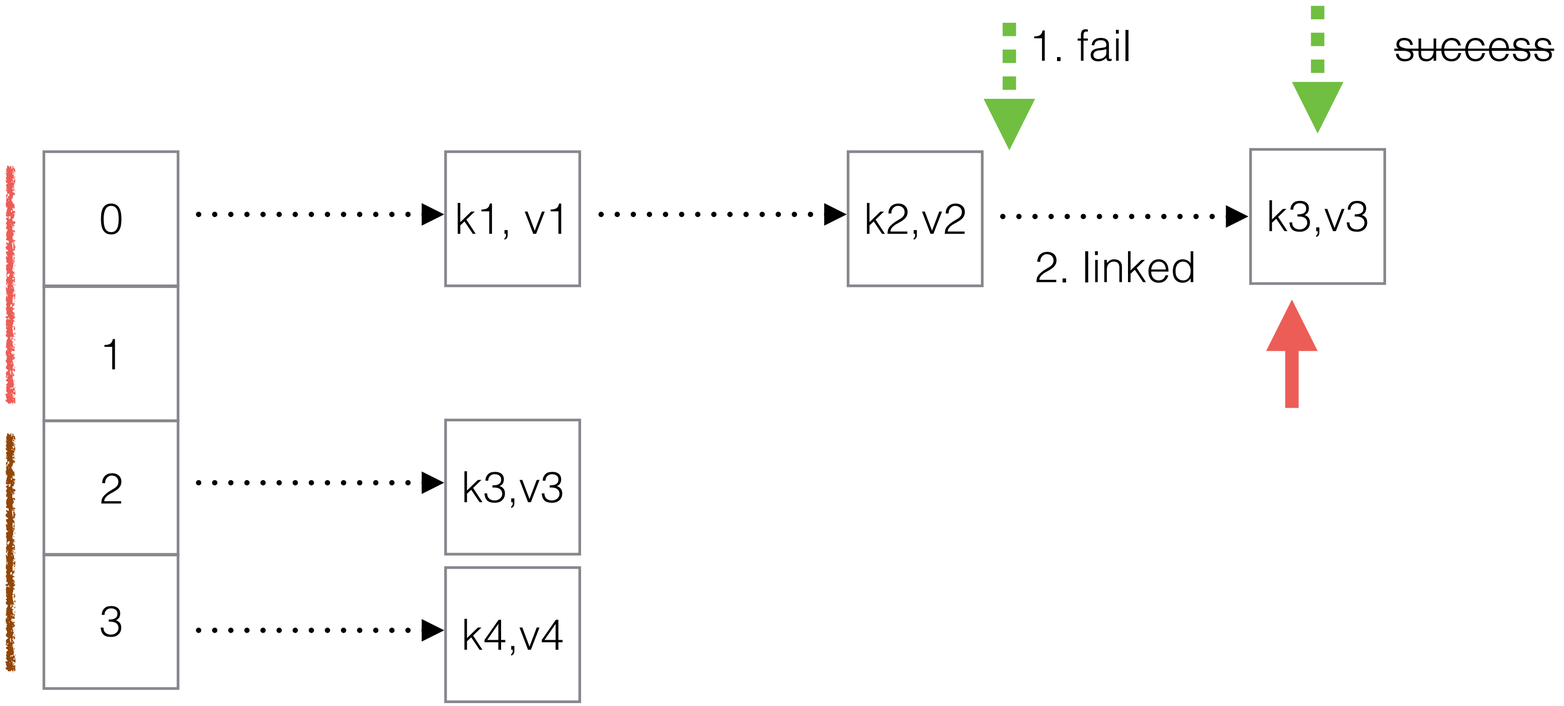
reading could be lock-free

ConcurrentDictionary



reading could be lock-free

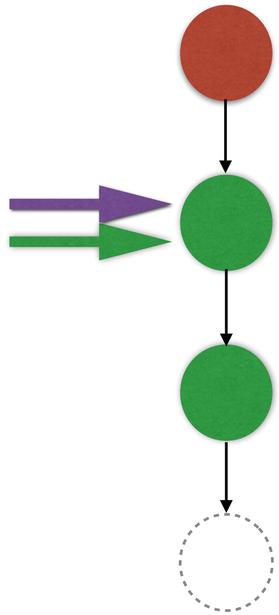
ConcurrentDictionary



reading could be lock-free

ConcurrentStack

Уже рассмотрели!



ConcurrentStack

```
class ConcurrentStack<T> {

 Node<T> head = null;

 T Pop() {
  var res;
  do {
   var old = head;
   res = head.Info;
   var new = head.Next;
  } while (!CAS(ref head, new, old))

  return res;
 }
}
```

ConcurrentStack

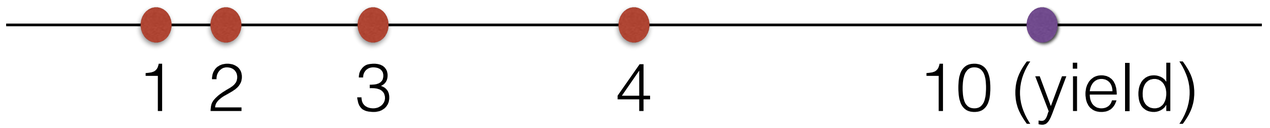
```
class ConcurrentStack<T> {

 Node<T> head = null;

 void Push(T info) {
  do {
   var old = head;
   var new = new Head(info, old);
  } while (!CAS(ref head, new, old))

 }
}
```

ConcurrentStack

```
// А теперь инженерное решение!
class ConcurrentStack<T> {

 Node<T> head = null;

 void Push(T v) {
  var sw = new SpinWait();
  do {
   var old = head;
   var new = new Head(v, old);
   sw.SpinOnce();
  } while (!CAS(ref head, new, old))
 }
}
```

SpinWait

- `Thread.SpinWait()` with exponential back-off strategy
- Goes to kernel sometimes (`yield`,`sleep`)
- Depends of #cores
- *Good latency and CPU load*



SpinWait

```
SpinWait.SpinUntil(Func<bool> condition)
```

Use it in your code right now!

ConcurrentQueue

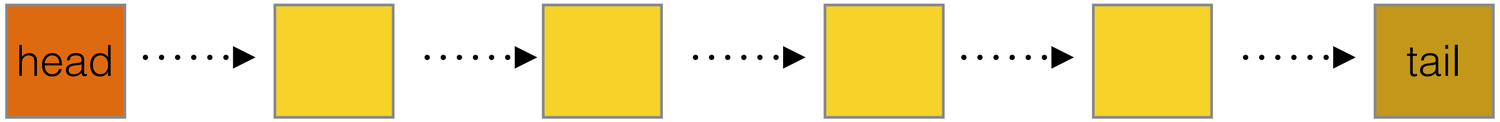
RingBuffer

ConcurrentQueue

RingBuffer

ConcurrentQueue

Michael, Maged; Scott, Michael (1996)
Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms

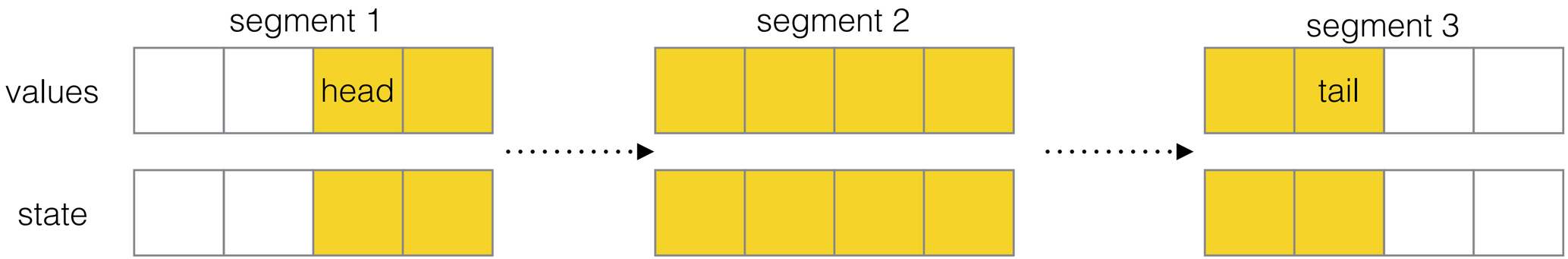


ConcurrentQueue

.net special way

ConcurrentQueue

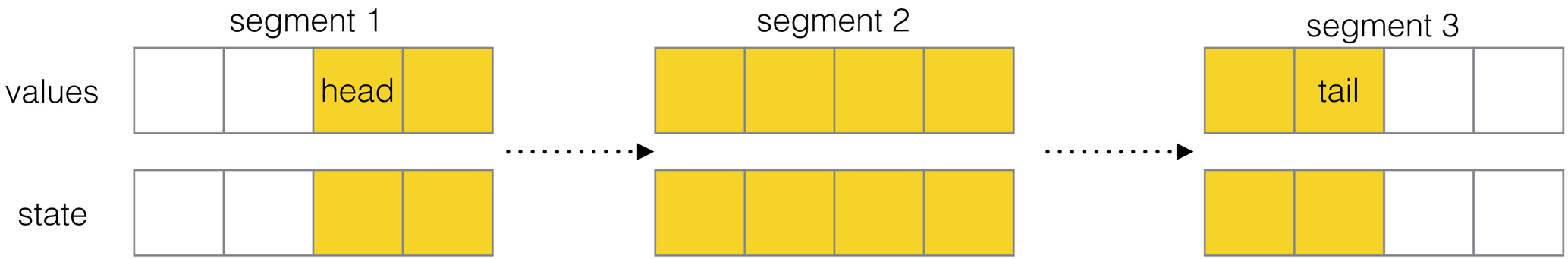
.net special way




```
struct VolatileBool {
  volatile bool m_value;
}
```

ConcurrentQueue

.net special way

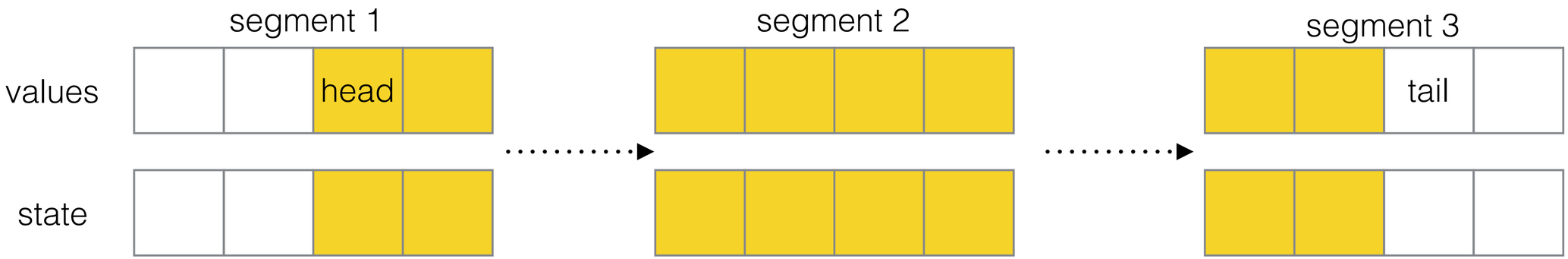


Enqueue:

1. tail ++
2. write value
3. state = 1

ConcurrentQueue

.net special way

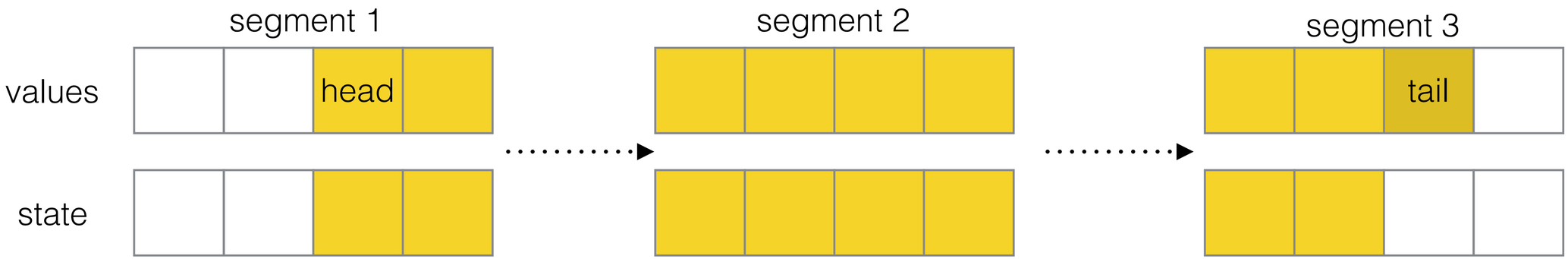


Enqueue:

1. tail ++
2. write value
3. state = 1

ConcurrentQueue

.net special way

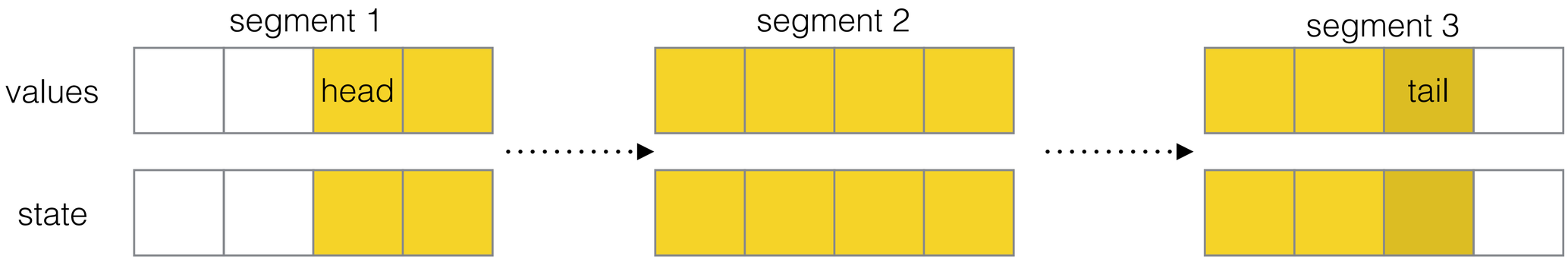


Enqueue:

1. tail ++
2. write value
3. state = 1

ConcurrentQueue

.net special way

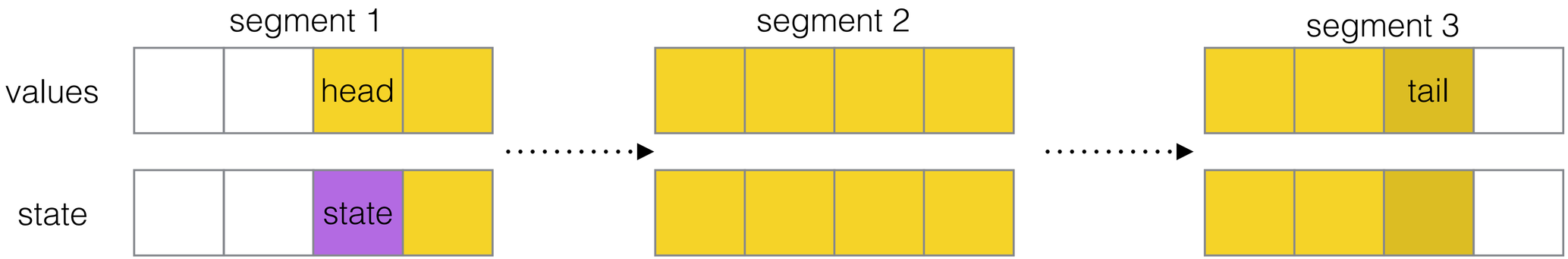


Enqueue:

1. tail ++
2. write value
3. state = 1

ConcurrentQueue

.net special way



Enqueue:

1. tail ++
2. write value
3. state = 1

Deque:

1. h = CAS(head => head+1) *(if head<=tail)*
2. SpinWaitFor(state==1) // e.g. head == tail
3. res = value;
4. value = null; state = 0
5. return res;

Резюме

1. Теперь вы знаете как отличить **блокирующий** алгоритм от неблокирующего, а **wait-free** от **lock-free**
2. Теперь вы можете легко написать **lock-free** коллекцию поверх **иммутабельной** (но она может слегка тормозить)
3. Теперь вы легко выберете самую подходящую системную коллекцию или напишете свою. Но это не точно…

Если вы вдохновились теорией

http://neerc.ifmo.ru/sptcc/

Если вы заинтересовались практикой

- https://docs.microsoft.com/en-us/dotnet/standard/collections/thread-safe/when-to-use-a-thread-safe-collection

- https://blogs.msdn.microsoft.com/pfxteam/2010/04/26/performance-of-concurrent-collections-in-net-4

Вопросы и ответы