Fair Locks and Semaphores in Kotlin Coroutines

Никита Коваль, JetBrains
Роман Елизаров, JetBrains

ITMO 2020

Locks and Semaphores

Mutex = **Mut**ual **Ex**clusion, at most 1 thread is in the critical section

Locks and Semaphores

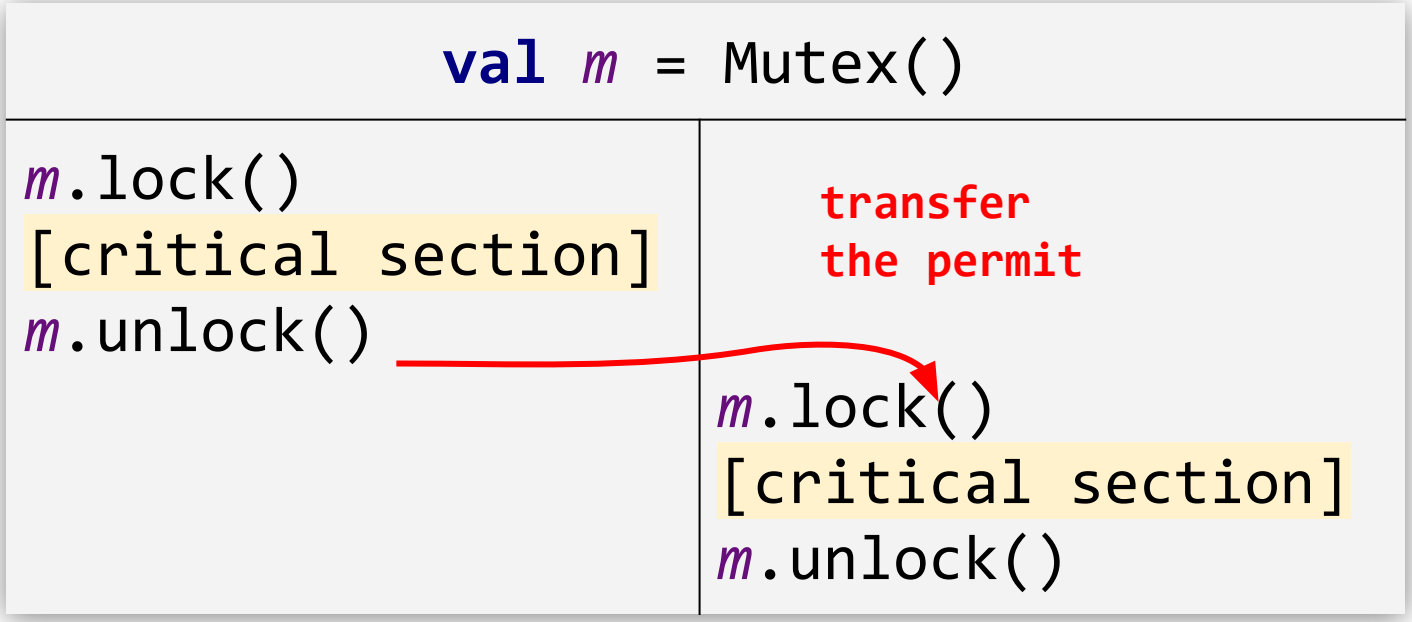
Mutex = **Mut**ual **Ex**clusion, at most 1 thread is in the critical section

```
val m = Mutex()
```

`m.lock()`
`[critical section]`
`m.unlock()`
	`m.lock()`
	`[critical section]`
	`m.unlock()`

Locks and Semaphores

Mutex = **Mut**ual **Ex**clusion, at most 1 thread is in the critical section



Locks and Semaphores

Semaphore = at most **K** threads are in the critical section

```
Mutex = Semaphore(permits = 1)
```

Locks and Semaphores

Semaphore = at most **K** threads are in the critical section

```
Mutex = Semaphore(permits = 1)
```

Semaphore algorithm is the start of this project!

Blocking Calls via `Future`

Threads, coroutines, continuations… Need a simple abstraction!

Blocking Calls via `Future`

Threads, coroutines, continuations… Need a simple abstraction!

```
interface Future<T> {
  fun await(): T
}
```

Blocking Calls via `Future`

Threads, coroutines, continuations… Need a simple abstraction!

```
interface Future<T> {
  fun await(): T
}
```

```
class FutureImmediate<T>(
  val res: T
) {
  fun await(): T = res
}
```

Blocking Calls via Future

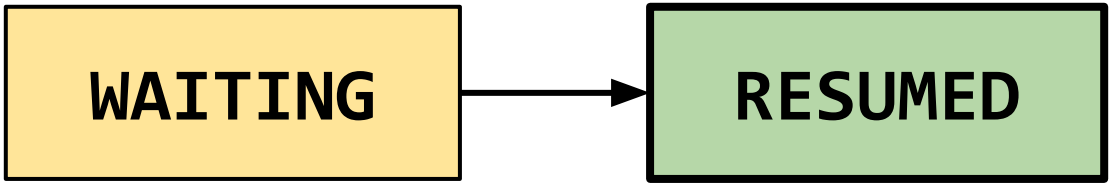
Threads, coroutines, continuations… Need a simple abstraction!

```
interface Future<T> {
  fun await(): T
}
```

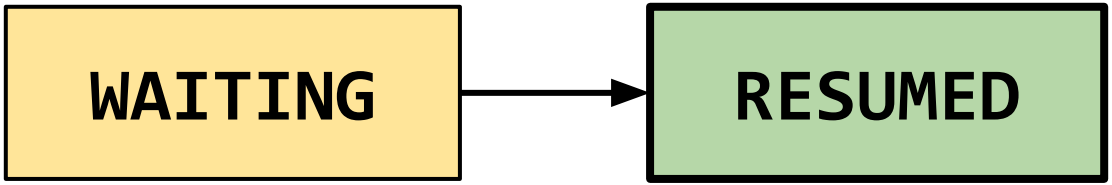
```
class FutureImmediate<T>(
  val res: T
) {
  fun await(): T = res
}
```

```
class FutureSuspended<T> {
  var state: T? = null

  fun await(): T {
    while (state == null) {}
    return state
  }

  fun complete(value: T) {
   state = value
  }
}
```

State Machine for `FutureSuspended`



State Machine for `FutureSuspended`



Further **`FutureSuspended`** updates will be shown via such diagrams

Semaphore API

```
class Semaphore(permits: Int) {
  fun acquire(): Unit
  fun release()
}
```

Semaphore API

```
class Semaphore(permits: Int) {
  fun acquire(): Future<Unit>
  fun release()
}
```

Blocking by design

Straightforward `Semaphore` Implementation

```
class Semaphore(permits: Int) {
  var permits: Int = permits
  ...
}
```

Straightforward `Semaphore` Implementation

```
class Semaphore(permits: Int) {
  var permits: Int = permits
  ...
}
```

< 0 → # waiters

Straightforward `Semaphore` Implementation

```
class Semaphore(permits: Int) {
  var permits: Int = permits
  ...
}
```

```
fun acquire(): Future<Unit> {
  val p = FAA(&permits, -1)
  if p > 0:
    return FutureImmediate(Unit)
  else:
    return suspend()
}
```

Creates a new `FutureSuspended` and puts it into the waiting queue

Straightforward `Semaphore` Implementation

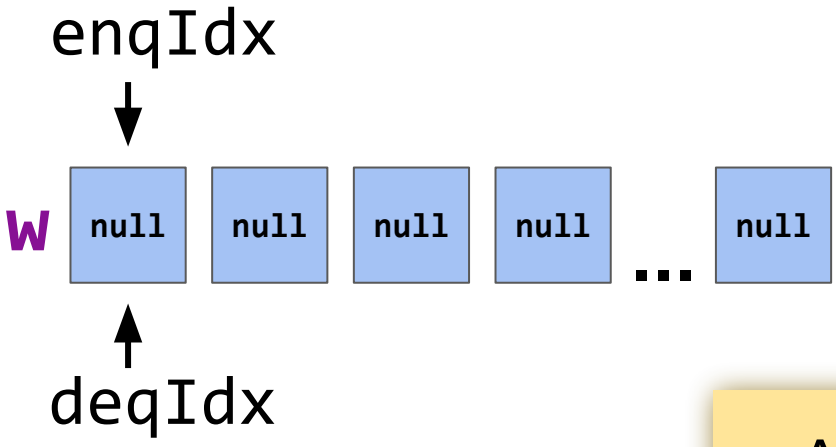
```
class Semaphore(permits: Int) {
  var permits: Int = permits
  ...
```

Retrieves the first waiter and completes it

```
fun acquire(): Future<Unit> {
  val p = FAA(&permits, -1)
  if p > 0:
    return FutureImmediate(Unit)
  else:
    return suspend()
}
```

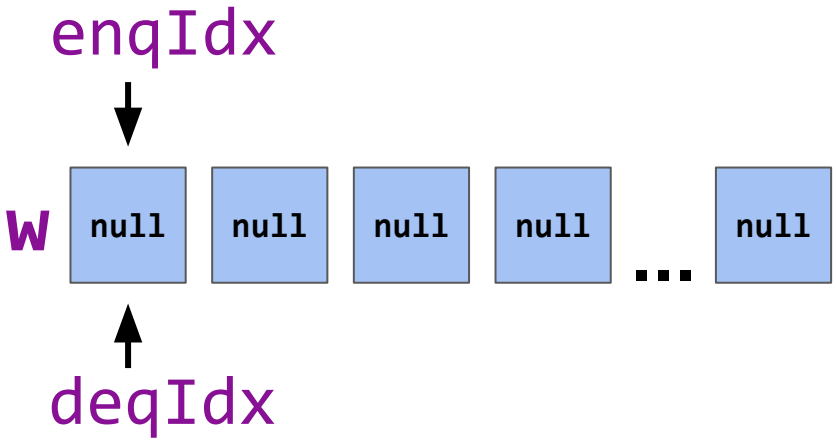
```
fun release() {
  val p = FAA(&permits, +1)
  if p >= 0: return
  resume(Unit)
}
```

SegmentQueueSynchronizer



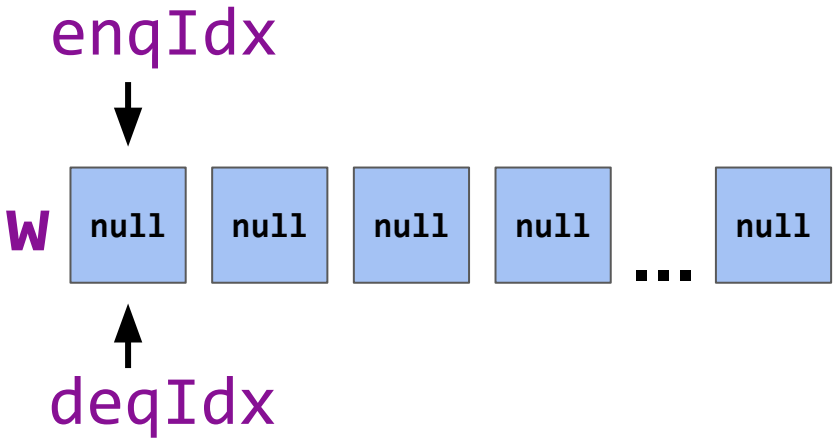
An infinite array with indices for the next addition and retrieval

SegmentQueueSynchronizer



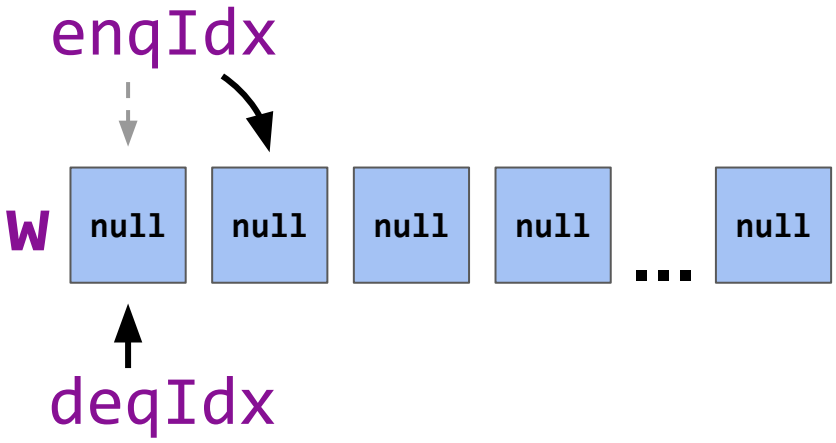

```
fun suspend(): Future<T> {
  val f = FutureSuspended<T>()
  val i = FAA(&enqIdx, +1)
  // store f into w[i]
}
```

SegmentQueueSynchronizer



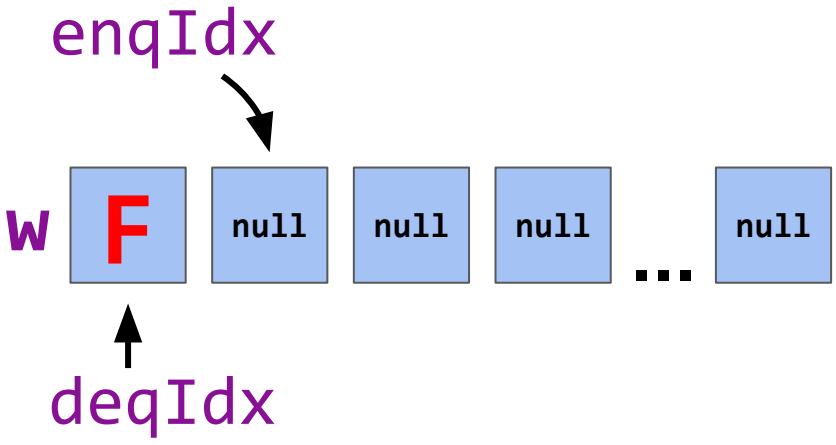

```
fun suspend(): Future<T> {
  val f = FutureSuspended<T>()
  val i = FAA(&enqIdx, +1)
  // store f into w[i]
}
```

SegmentQueueSynchronizer



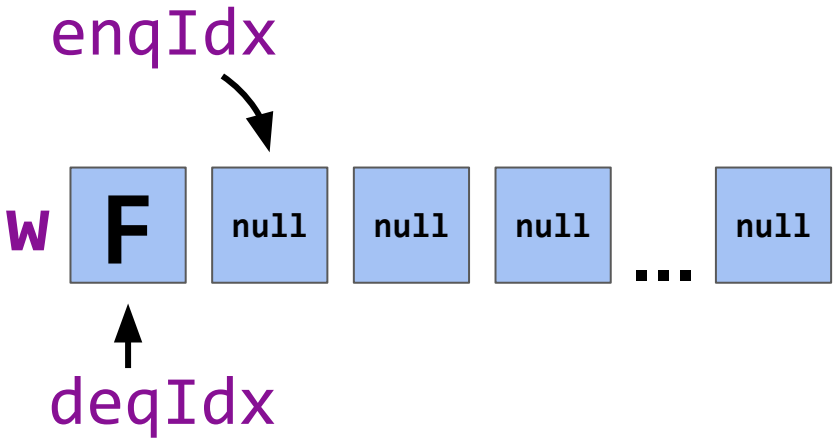

```
fun suspend(): Future<T> {
  val f = FutureSuspended<T>()
  val i = FAA(&enqIdx, +1)   i:0
  // store f into w[i]
}
```

SegmentQueueSynchronizer




```
fun suspend(): Future<T> {
  val f = FutureSuspended<T>()
  val i = FAA(&enqIdx, +1)   i:0
  // store f into w[i]
}
```

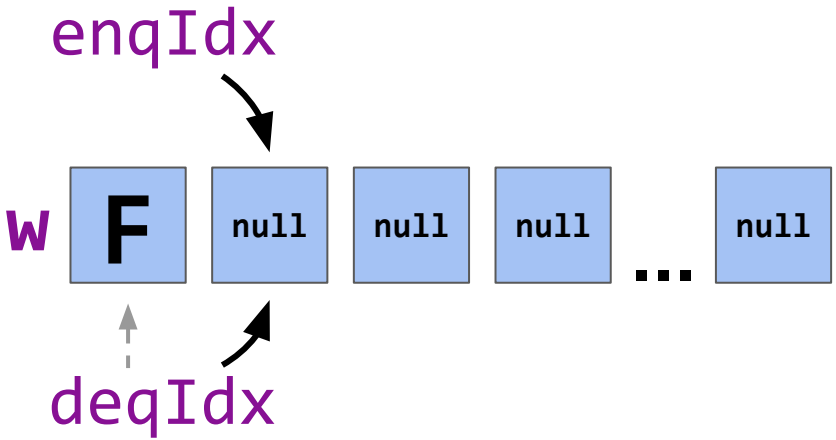
SegmentQueueSynchronizer




```
fun suspend(): Future<T> {
  val f = FutureSuspended<T>()
  val i = FAA(&enqIdx, +1)
  // store f into w[i]
}
```

```
fun resume(value: T) {
  val i = FAA(&deqIdx, +1)
  // complete the future
  // located in w[i]
}
```

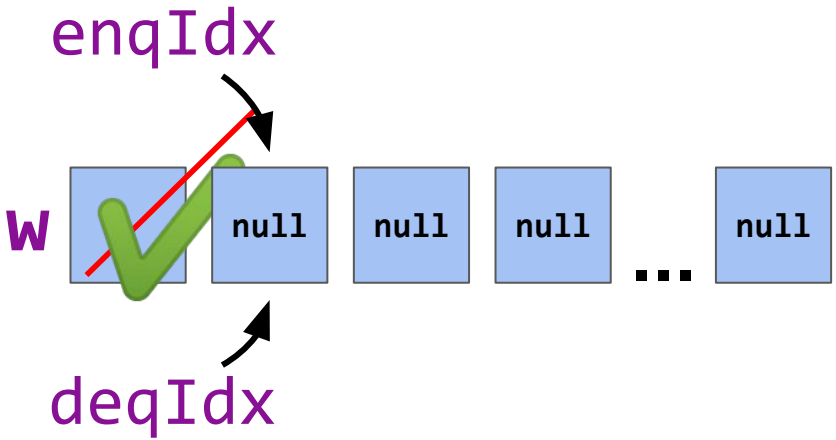
SegmentQueueSynchronizer




```
fun suspend(): Future<T> {
  val f = FutureSuspended<T>()
  val i = FAA(&enqIdx, +1)
  // store f into w[i]
}
```

```
fun resume(value: T) {
  val i = FAA(&deqIdx, +1)   i:0
  // complete the future
  // located in w[i]
}
```

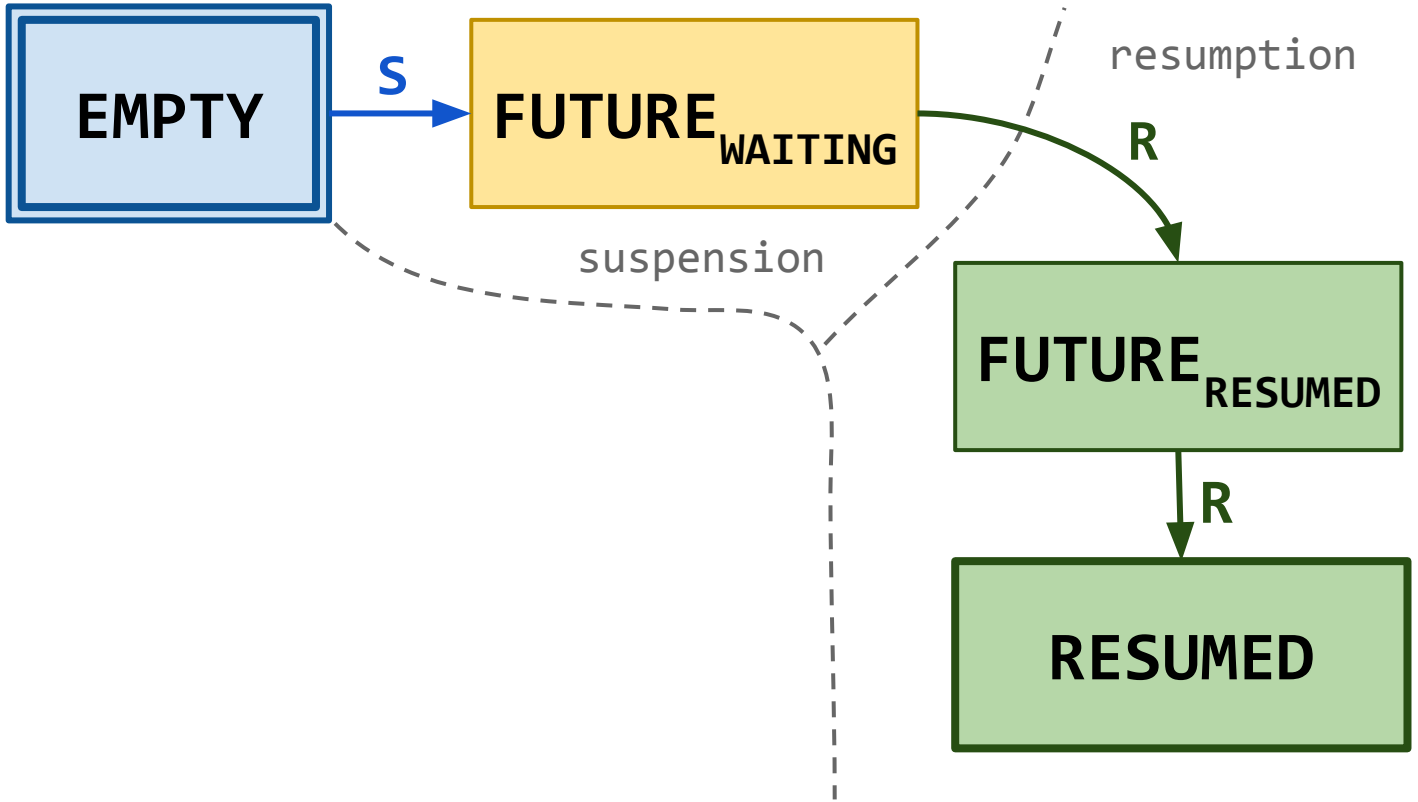
SegmentQueueSynchronizer



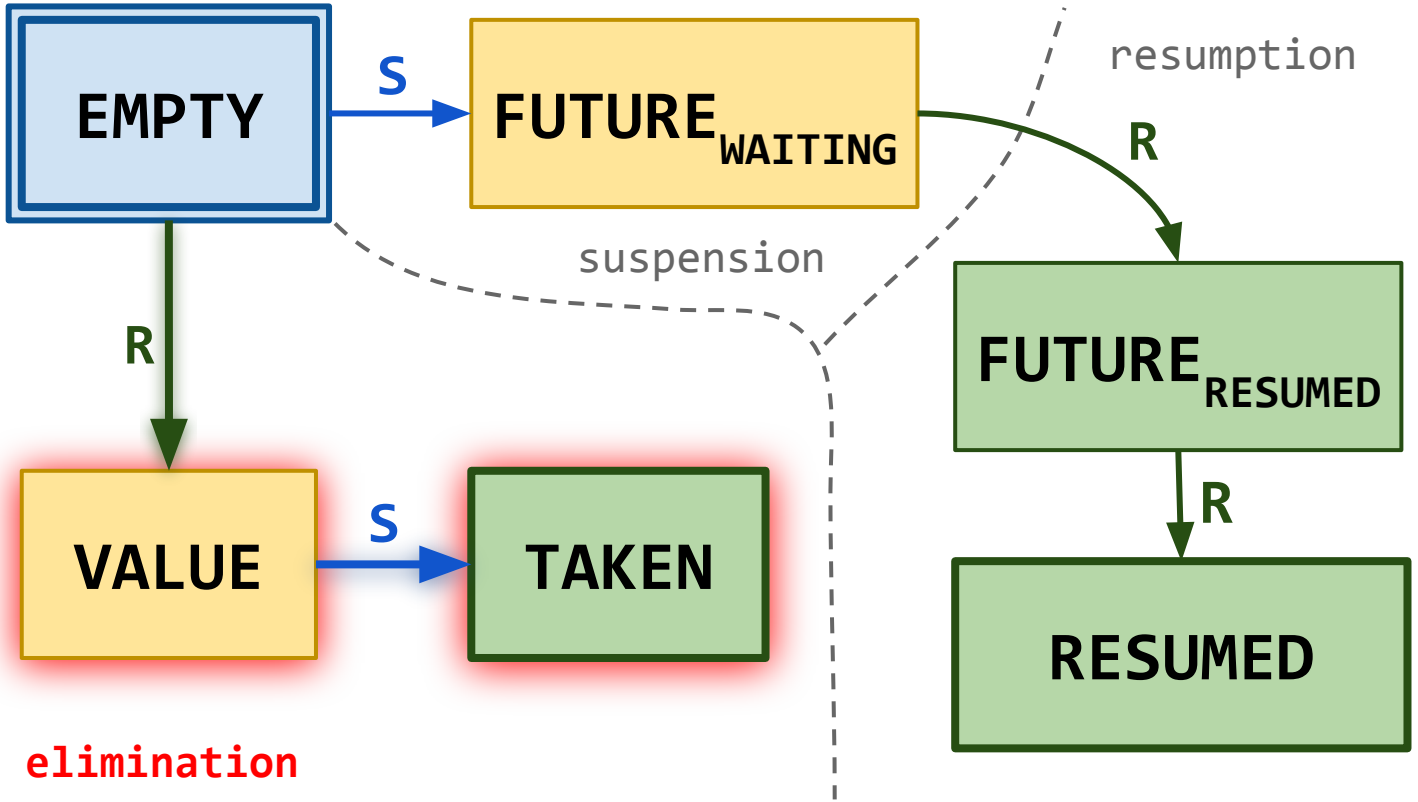

```
fun suspend(): Future<T> {
  val f = FutureSuspended<T>()
  val i = FAA(&enqIdx, +1)
  // store f into w[i]
}
```

```
fun resume(value: T) {
  val i = FAA(&deqIdx, +1)   i:0
  // complete the future
  // located in w[i]
}
```

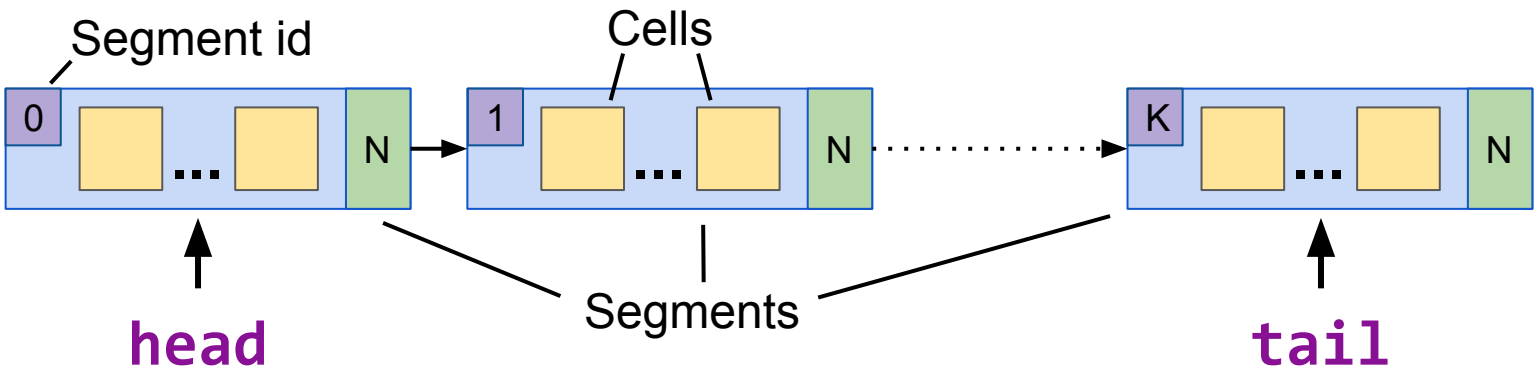
Cell Life-Cycle



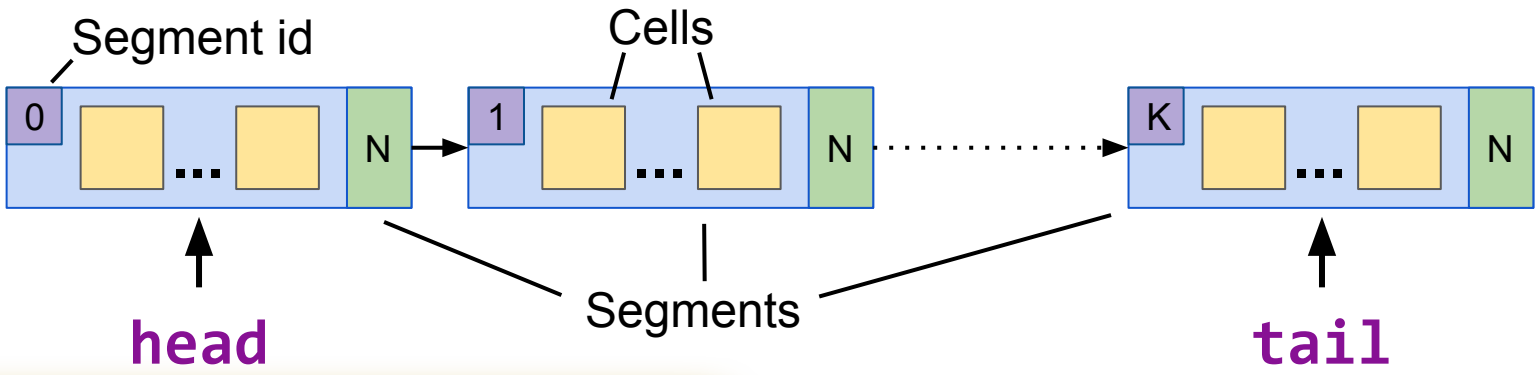
Cell Life-Cycle



Infinite Array Implementation

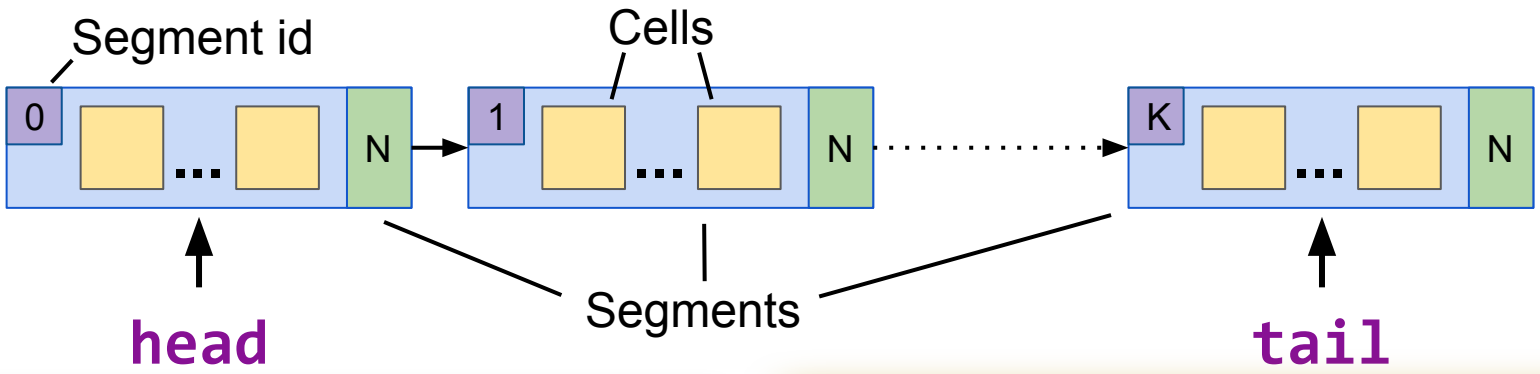


Infinite Array Implementation




```
fun suspend(): Future<T> {
  val t = tail
  val i = FAA(&enqIdx, +1)
  val s = findSegment(start = t,
                      id = i / M)
  moveTailForward(t)
  // w[i] is s[i % M]
}
```

Infinite Array Implementation




```
fun suspend(): Future<T> {
  val t = tail
  val i = FAA(&enqIdx, +1)
  val s = findSegment(start = t,
                      id = i / M)
  moveTailForward(t)
  // w[i] is s[i % M]
}
```

```
fun resume(value: T) {
  val t = head
  val i = FAA(&deqIdx, +1)
  val s = findSegment(start = t,
                      id = i / M)
  moveHeadForward(t)
  // w[i] is s[i % M]
}
```

Extend Semaphore with tryAcquire

```
fun acquire(): Future<Unit> {
  val p = FAA(&permits, -1)
  if p > 0:
    return FutureImmediate(Unit)
  else:
    return suspend()
}
```

```
fun release() {
  val p = FAA(&permits, +1)
  if p >= 0: return
  resume(Unit)
}
```

Extend Semaphore with tryAcquire

```
fun acquire(): Future<Unit> {
  val p = FAA(&permits, -1)
  if p > 0:
    return FutureImmediate(Unit)
  else:
    return suspend()
}
```

```
fun release() {
  val p = FAA(&permits, +1)
  if p >= 0: return
  resume(Unit)
}
```

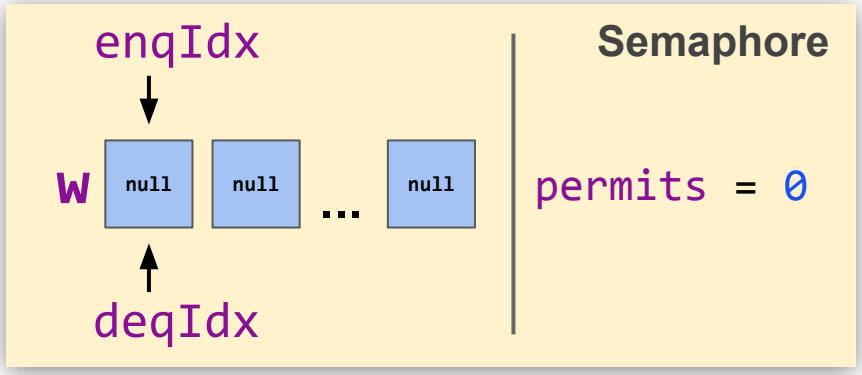
```
fun tryAcquire(): Boolean = while(true) {
  val p = permits
  if p <= 0: return false
  if CAS(&permits, p, p - 1): return true
}
```

Is this `tryAcquire` correct?

```
fun acquire(): Future<Unit> {
  val p = FAA(&permits, -1)
  if p > 0:
    return FutureImmediate(Unit)
  else:
    return suspend()
}
```

```
fun tryAcquire(): Boolean {
 while(true) {
   val p = permits
   if p <= 0: return false
   if CAS(&permits, p, p - 1):
     return false
 }
}
```

```
fun release() {
  val p = FAA(&permits, +1)
  if p >= 0: return
  resume(Unit)
}
```



`val s = Semaphore(1); s.acquire()`	

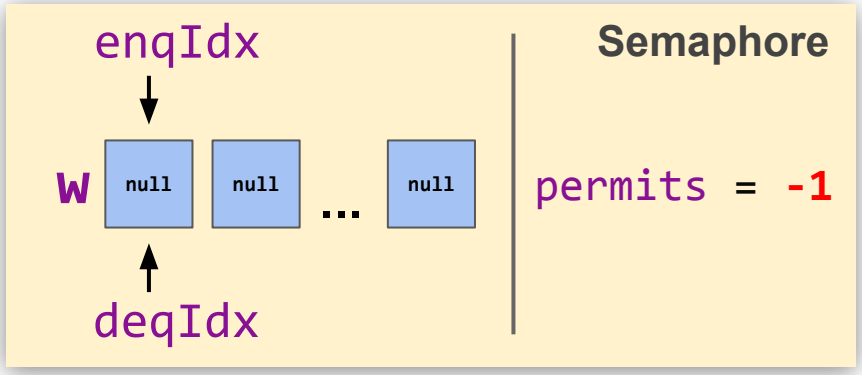
`s.release()`
`s.tryAcquire()`
`s.acquire()`
	`s.acquire():`
	`1. dec permits`
	`2. suspend`

Is this `tryAcquire` correct?

```
fun acquire(): Future<Unit> {
  val p = FAA(&permits, -1)
  if p > 0:
    return FutureImmediate(Unit)
  else:
    return suspend()
}
```

```
fun tryAcquire(): Boolean {
 while(true) {
   val p = permits
   if p <= 0: return false
   if CAS(&permits, p, p - 1):
     return false
 }
}
```

```
fun release() {
  val p = FAA(&permits, +1)
  if p >= 0: return
  resume(Unit)
}
```




```
val s = Semaphore(1); s.acquire()
```

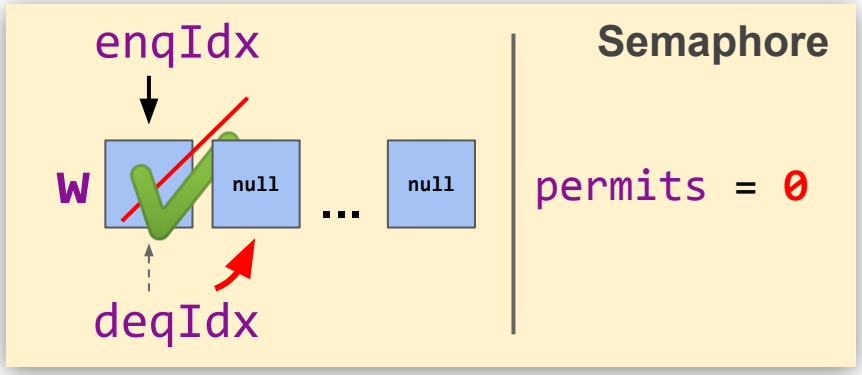
`s.release()`
`s.tryAcquire()`
`s.acquire()`
	`s.acquire():`
	`1. dec permits`
	`2. suspend`

Is this `tryAcquire` correct?

```
fun acquire(): Future<Unit> {
  val p = FAA(&permits, -1)
  if p > 0:
    return FutureImmediate(Unit)
  else:
    return suspend()
}
```

```
fun tryAcquire(): Boolean {
 while(true) {
   val p = permits
   if p <= 0: return false
   if CAS(&permits, p, p - 1):
     return false
 }
}
```

```
fun release() {
  val p = FAA(&permits, +1)
  if p >= 0: return
  resume(Unit)
}
```




```
val s = Semaphore(1); s.acquire()
```

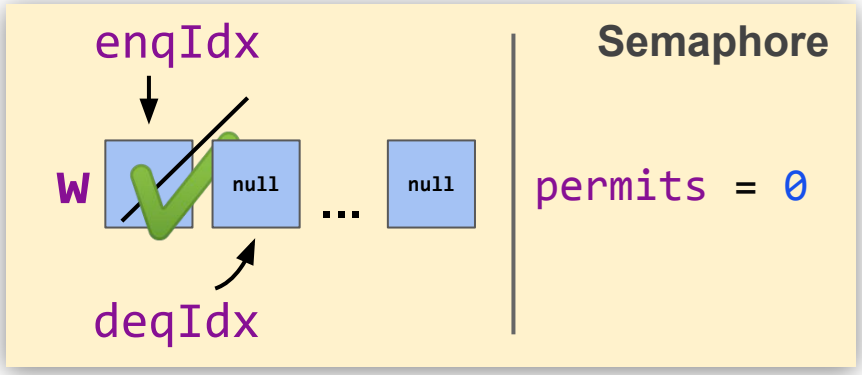
`s.release()` :done
`s.tryAcquire()`
`s.acquire()`
	`s.acquire()`:
	1. dec permits
	2. suspend

Is this `tryAcquire` correct?

```
fun acquire(): Future<Unit> {
  val p = FAA(&permits, -1)
  if p > 0:
    return FutureImmediate(Unit)
  else:
    return suspend()
}
```

```
fun tryAcquire(): Boolean {
 while(true) {
   val p = permits
   if p <= 0: return false
   if CAS(&permits, p, p - 1):
     return false
 }
}
```

```
fun release() {
  val p = FAA(&permits, +1)
  if p >= 0: return
  resume(Unit)
}
```




```
val s = Semaphore(1); s.acquire()
```

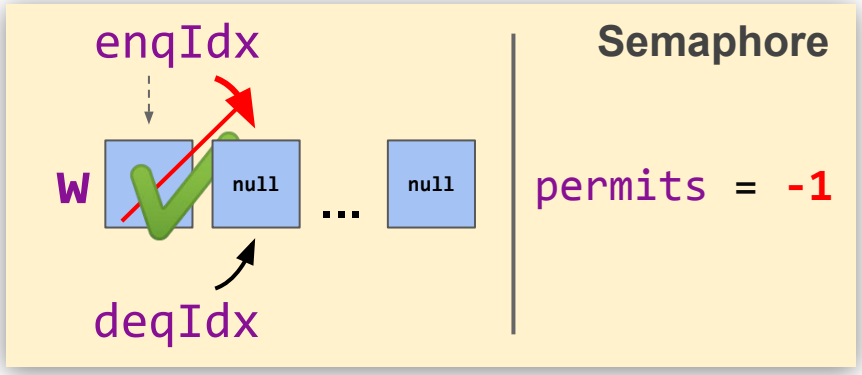
`s.release() :done`
`s.tryAcquire():f`
`s.acquire()`
	`s.acquire():`
	`  1. dec permits`
	`  2. suspend`

Is this `tryAcquire` correct?

```
fun acquire(): Future<Unit> {
  val p = FAA(&permits, -1)
  if p > 0:
    return FutureImmediate(Unit)
  else:
    return suspend()
}
```

```
fun tryAcquire(): Boolean {
 while(true) {
   val p = permits
   if p <= 0: return false
   if CAS(&permits, p, p - 1):
     return false
 }
}
```

```
fun release() {
  val p = FAA(&permits, +1)
  if p >= 0: return
  resume(Unit)
}
```

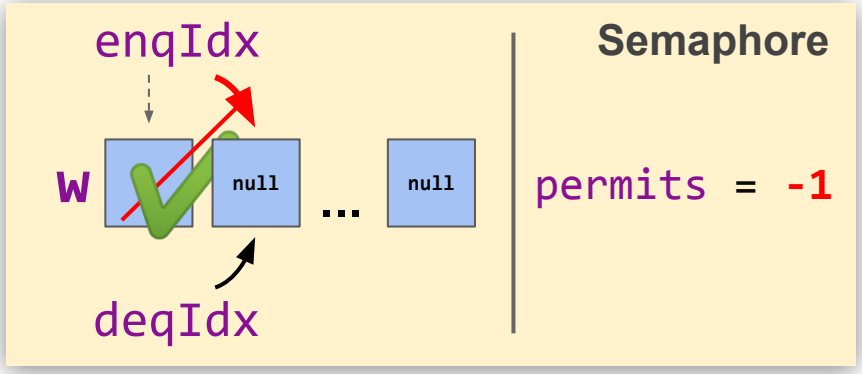



```
val s = Semaphore(1); s.acquire()
```

`s.release()` :done `s.tryAcquire()`:f `s.acquire()` :done	`s.acquire()`: 1. dec permits 2. suspend

Is this `tryAcquire` correct?

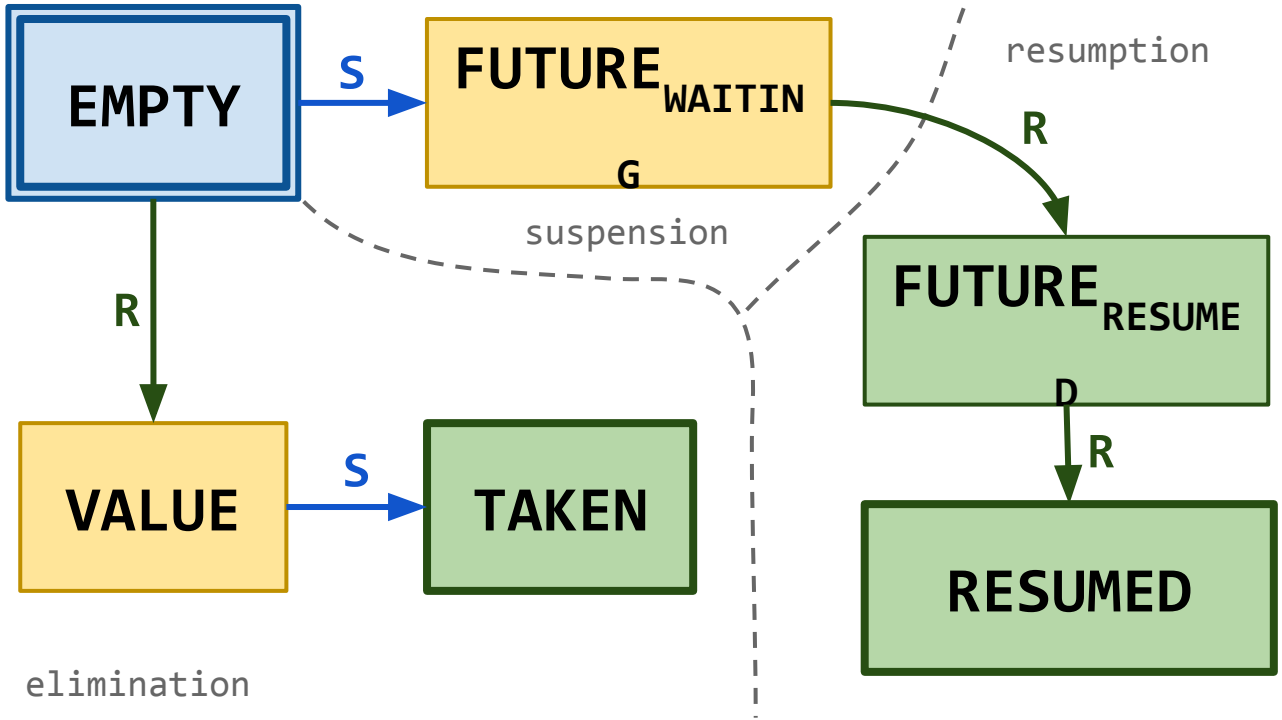
`release` was intended go give a permit to a concurrent **`acquire`**, but gave it to **`acquire`** that *happens after* it



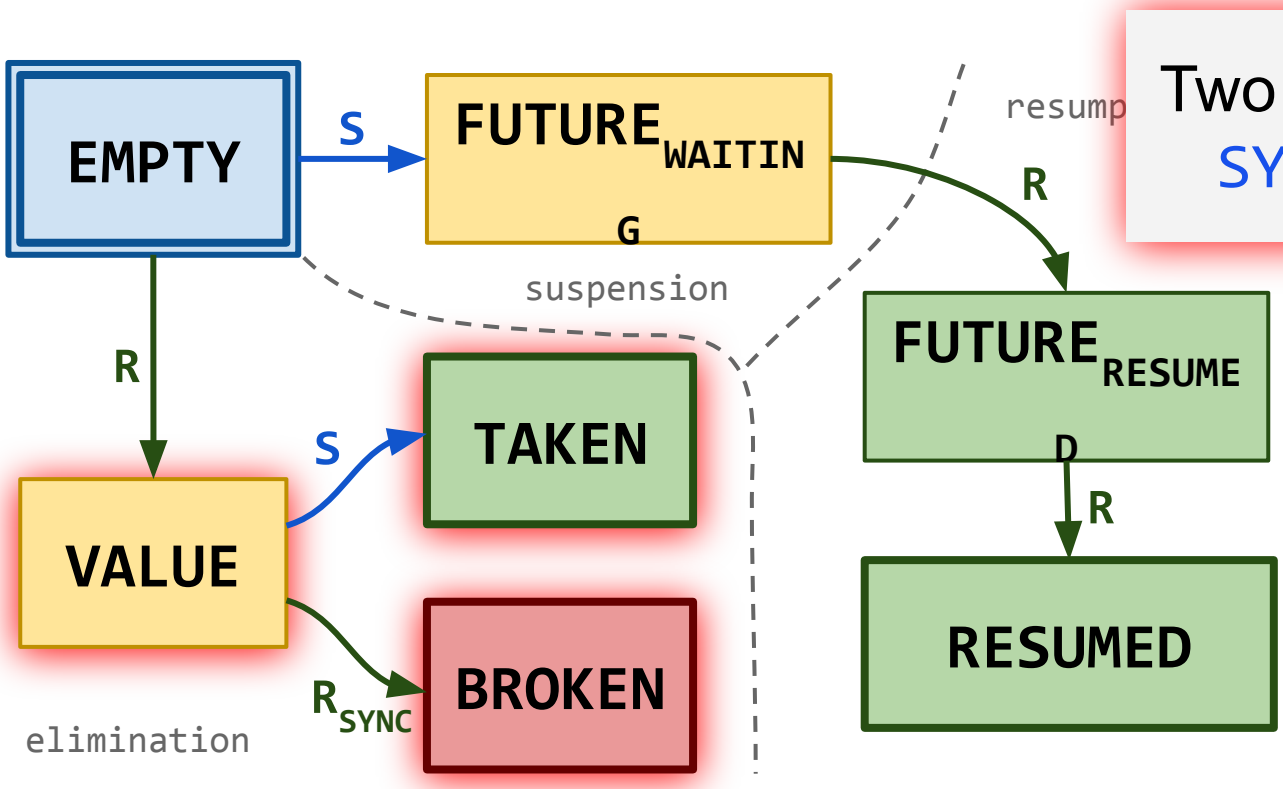

```
val s = Semaphore(1); s.acquire()
```

`s.release() :done`
`s.tryAcquire():f`
`s.acquire() :done`
	`s.acquire():`
	`1. dec permits`
	`2. suspend`

Extend Semaphore with tryAcquire



Extend Semaphore with tryAcquire



Extend Semaphore with tryAcquire

```
class SegmentQueueSynchronizer<T> {
  // can fail on elimination
  fun resume(): Boolean { ... }

  // returns null on broken cell
  fun suspend(): Future<T>? { ... }
}
```

Extend Semaphore with tryAcquire

```
class SegmentQueueSynchronizer<T> {
  // can fail on elimination
  fun resume(): Boolean { ... }

  // returns null on broken cell
  fun suspend(): Future<T>? { ... }
}
```

```
fun acquire(): Future<Unit> {
  while(true) {
    val p = FAA(&permits, -1)
    if p > 0:
      return FutureImmediate(Unit)
    val f = suspend()
    if f != null: return f
  }
}
```

```
fun release() {
  while(true) {
    val p = FAA(&permits, +1)
    if p >= 0: return
    val done = resume(Unit)
    if done: return
  }
}
```

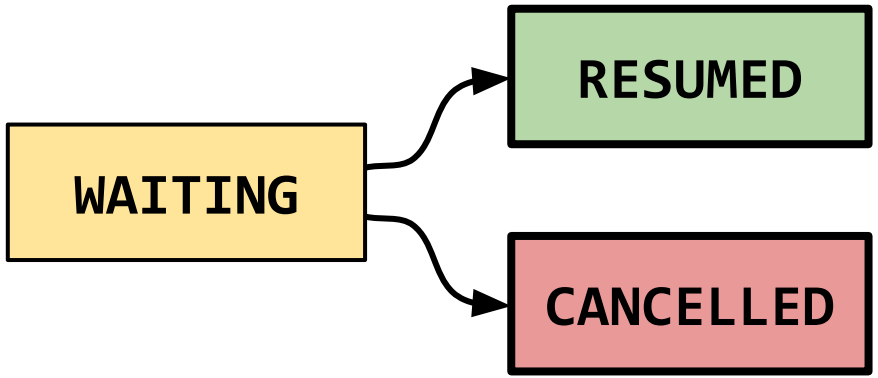
Cancellable `FutureSuspended`

- Cancellation (abortability) is natural for blocking primitives
- Moreover, it is a built-in feature in Kotlin Coroutines

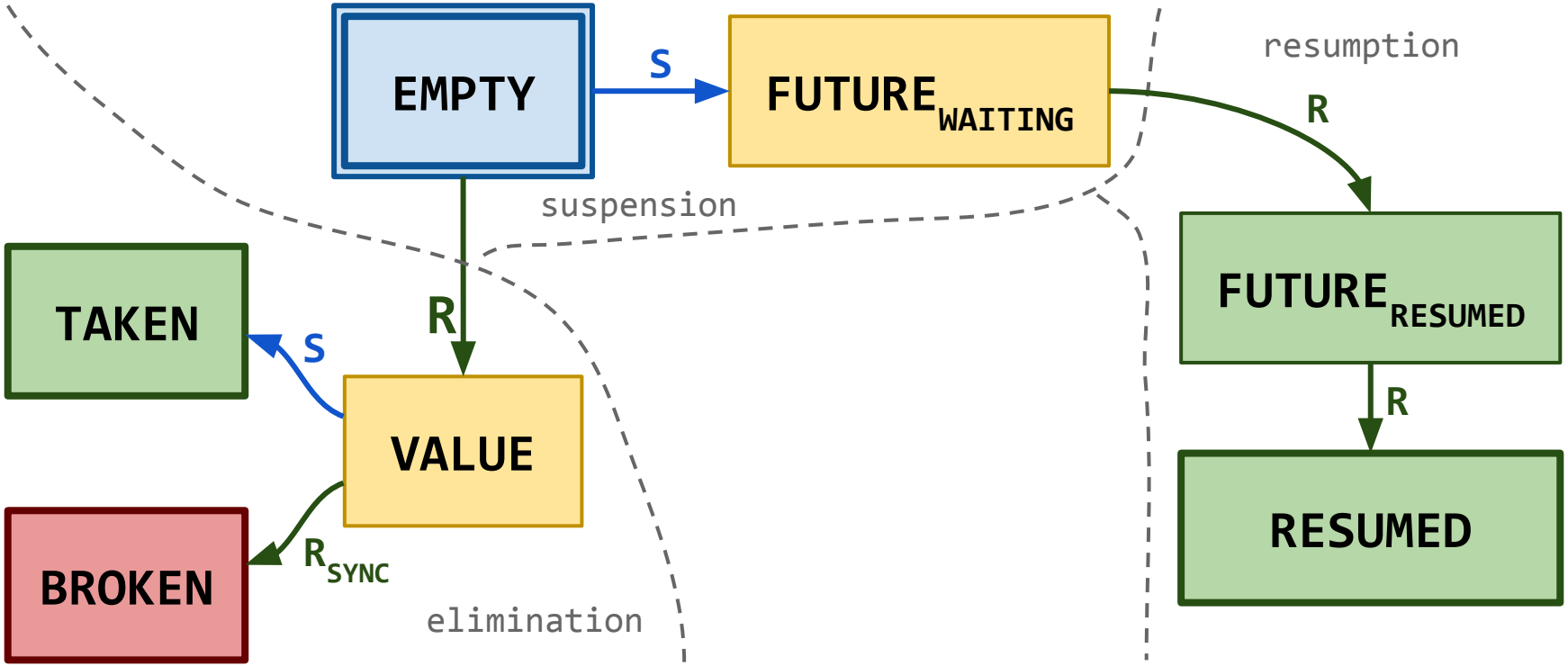
Cancellable `FutureSuspended`

- Cancellation (abortability) is natural for blocking primitives
- Moreover, it is a built-in feature in Kotlin Coroutines

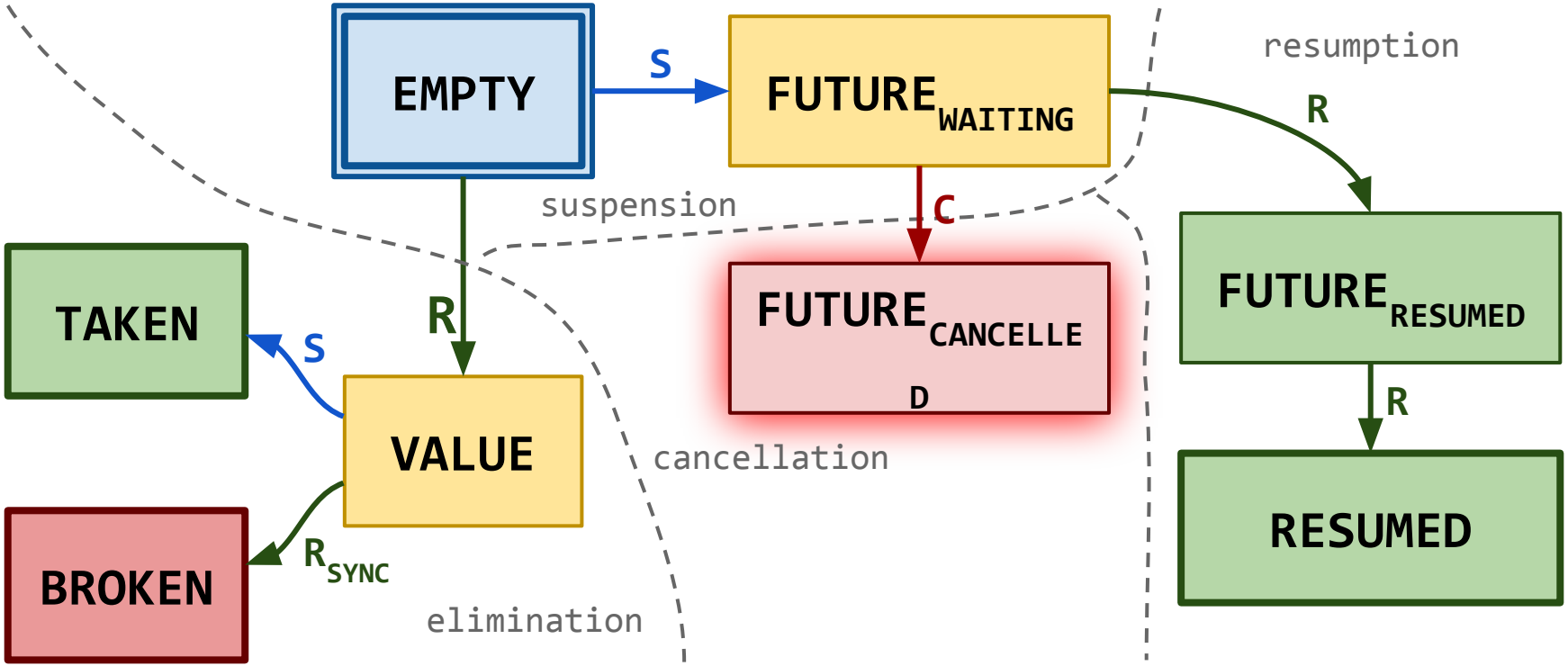
```
class FutureSuspended<T> {
  // returns ⊥ if cancelled
  fun await(): T? { ... }
  fun complete(value: T) { ... }
  fun cancel(): Boolean { ... }

  fun handleCancellation() {
    // TODO: Implement me, please!
  }
}
```



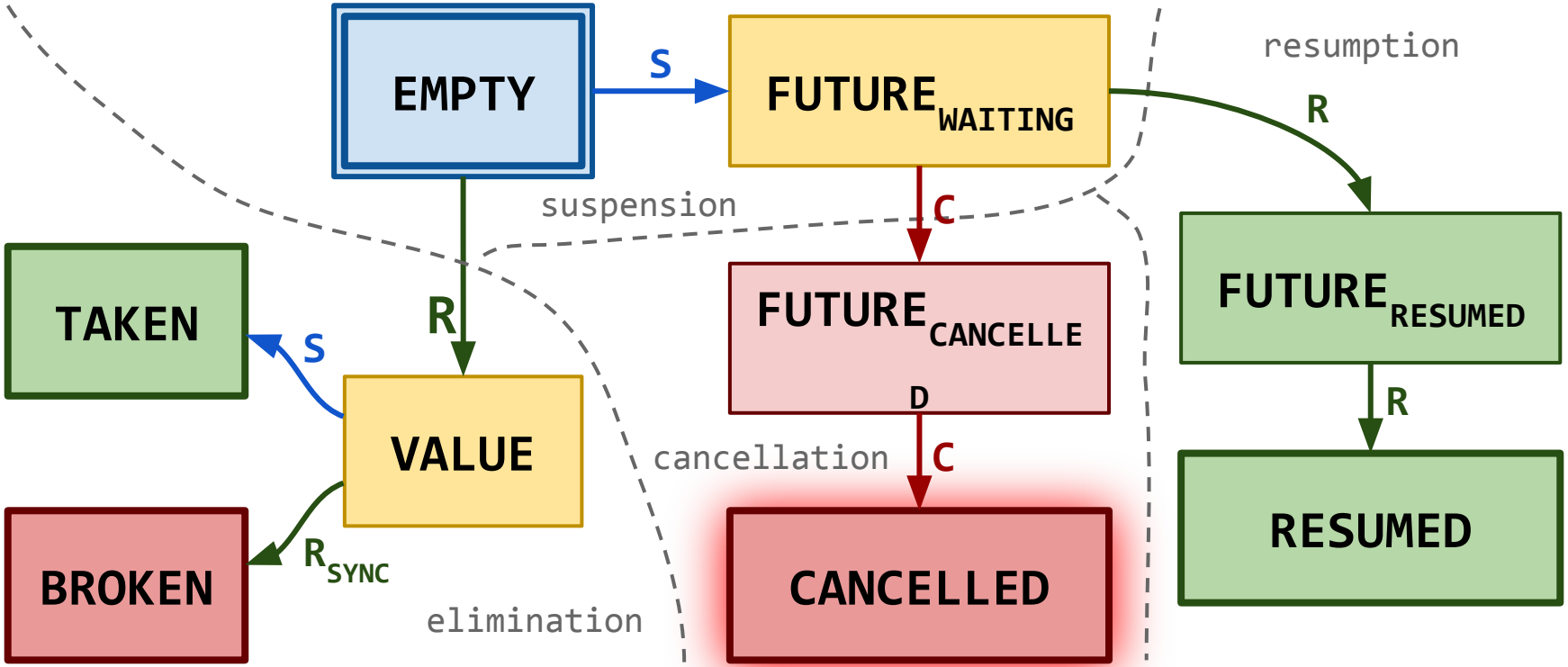
Cancellation Support in the Cell Life-Cycle



Cancellation Support in the Cell Life-Cycle

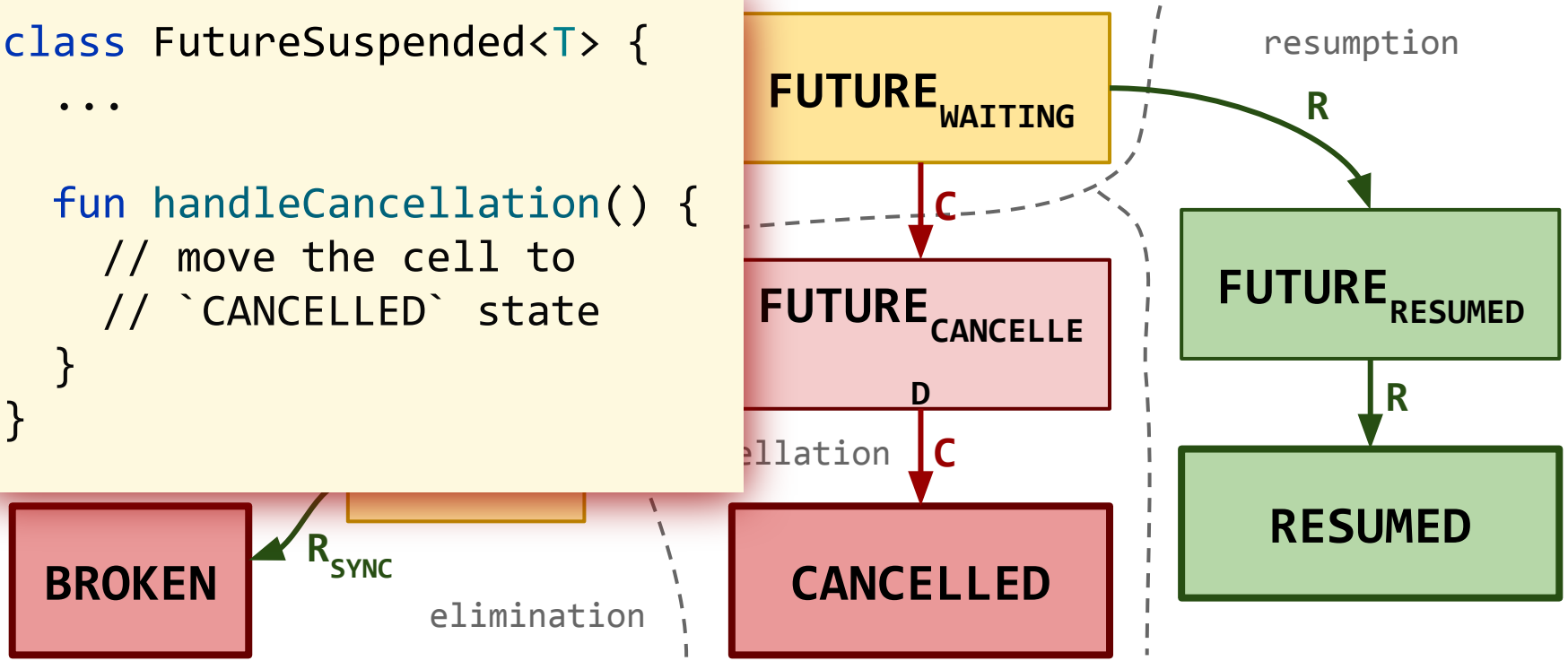


Cancellation Support in the Cell Life-Cycle



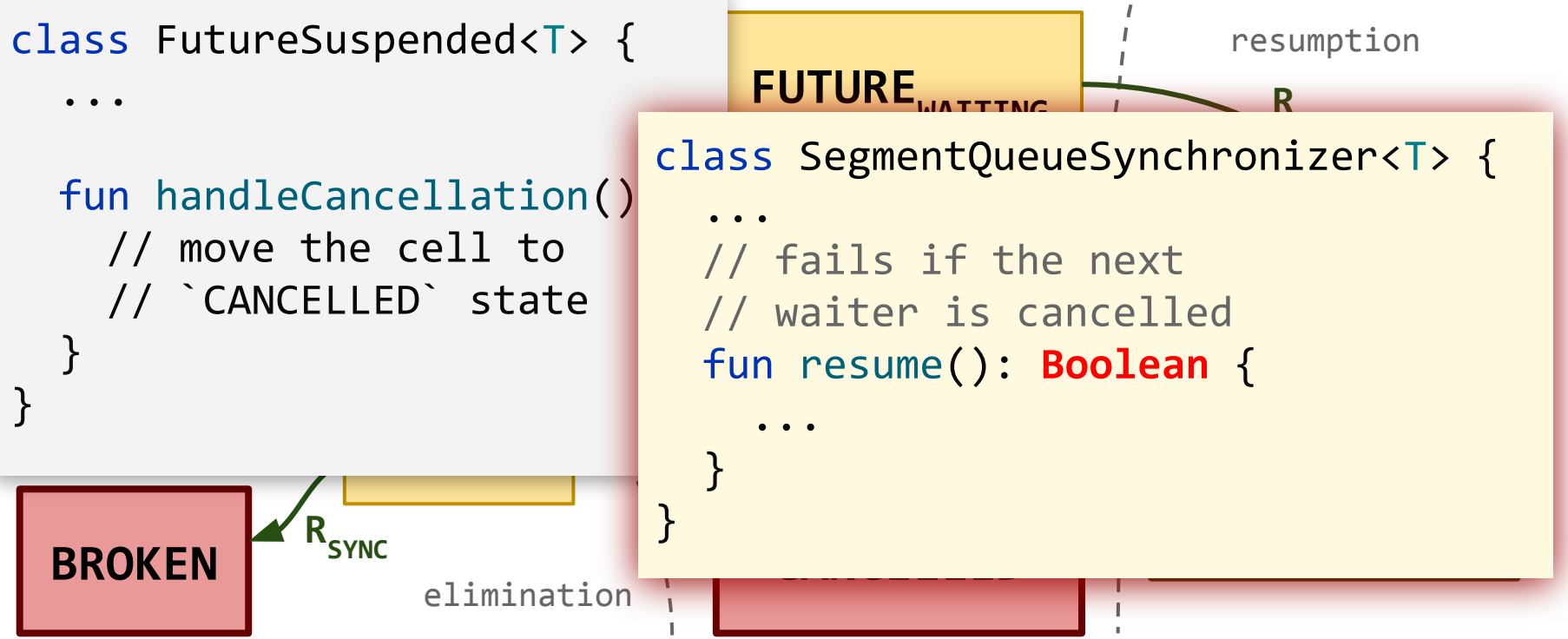
Cancellation Support in the Cell Life-Cycle

```
class FutureSuspended<T> {
  ...

  fun handleCancellation() {
    // move the cell to
    // `CANCELLED` state
  }
}
```



Cancellation Support in the Cell Life-Cycle

```
class FutureSuspended<T> {
  ...

  fun handleCancellation()
    // move the cell to
    // `CANCELLED` state
  }
}
```



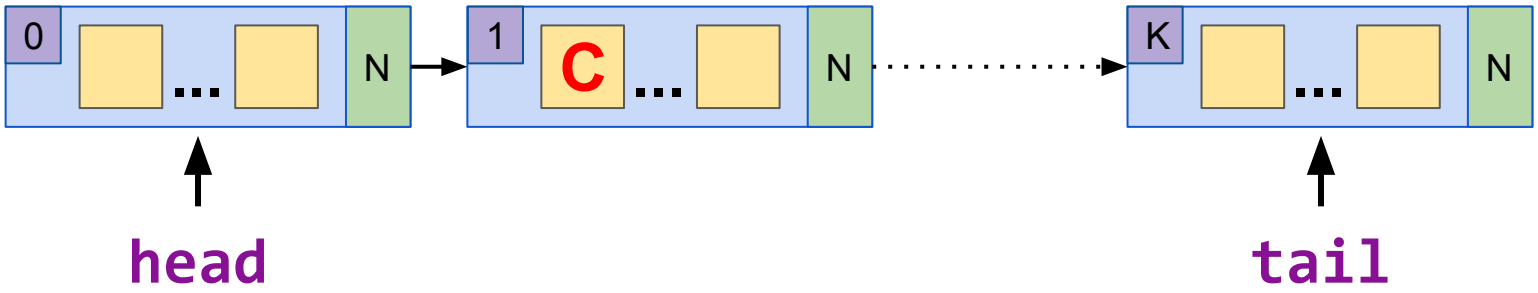

```
class SegmentQueueSynchronizer<T> {
  ...
  // fails if the next
  // waiter is cancelled
  fun resume(): Boolean {
    ...
  }
}
```

Cancellation Support in General

- **`handleCancellation`** moves the cell state to **`CANCELLED`** to avoid memory leaks
- Can we remove the cells themselves for the same reason?

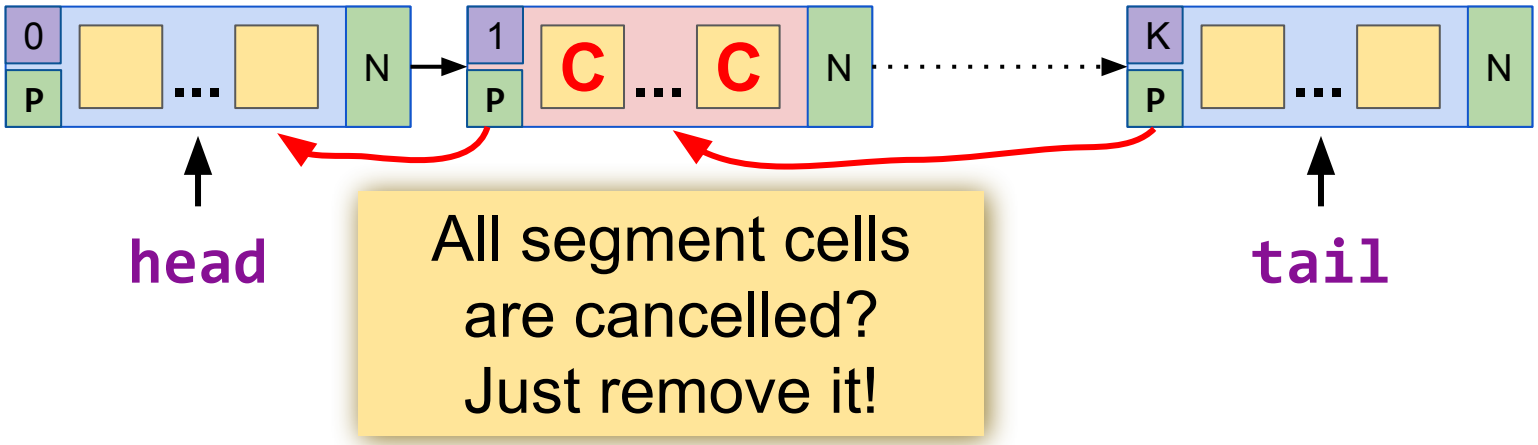
Cancellation Support and Memory Leaks

- **`handleCancellation`** moves the cell state to **`CANCELLED`** to avoid memory leaks
- Can we remove the cells themselves for the same reason?



Cancellation Support and Memory Leaks

- **`handleCancellation`** moves the cell state to **`CANCELLED`** to avoid memory leaks
- Can we remove the cells themselves for the same reason?



Semaphore with Cancellation

Our implementation is already correct!

```
fun acquire(): Future<Unit> {
  while(true) {
    val p = FAA(&permits, -1)
    if p > 0:
      return FutureImmediate(Unit)
    val f = suspend()
    if f != null: return f
  }
}
```

```
fun release() {
  while(true) {
    val p = FAA(&permits, +1)
    if p >= 0: return
    // can fail due to cancellation,
    // adjusted `permits` then
    val done = resume(Unit)
    if done: return
  }
}
```

Evaluation

- Google Cloud machine with 96 CPUs
- Comparison against standard Java implementations:
  - `j.u.c.Semaphore` and `j.u.c.ReentrantLock`

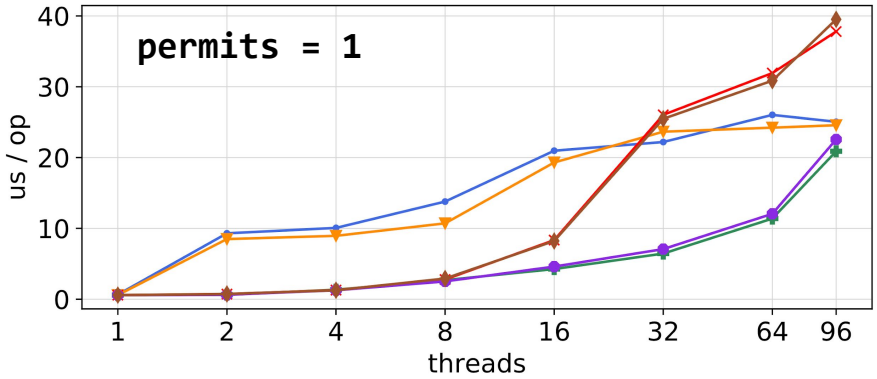
Evaluation

- Google Cloud machine with 96 CPUs
- Comparison against standard Java implementations:
  - `j.u.c.Semaphore` and `j.u.c.ReentrantLock`

```
fun operation()
{
    semaphore.acquire()

doSomeGeomDistrWork()

semaphore.release()
```

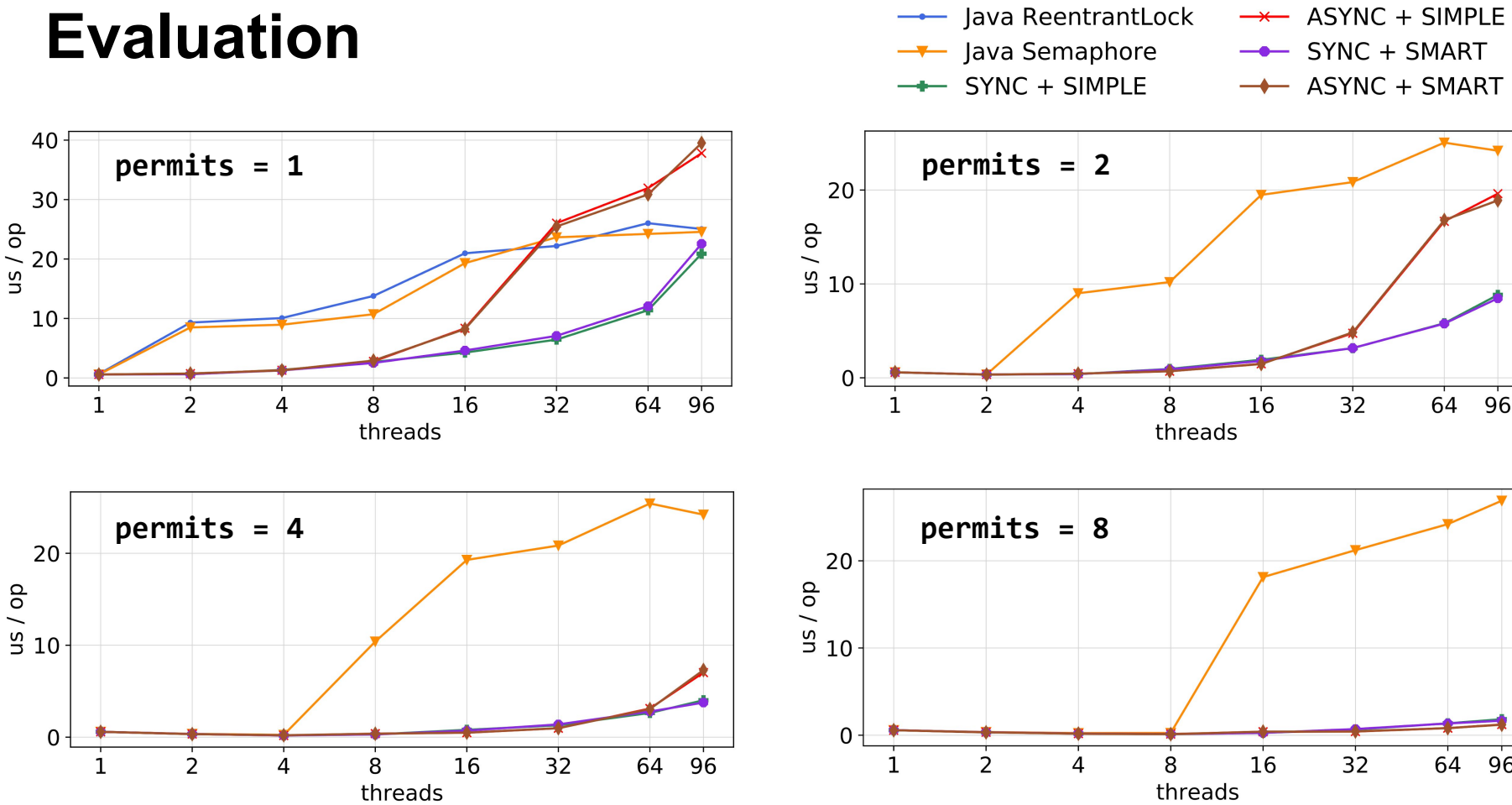
Let's run this code on different numbers of threads!

Evaluation



Java ReentrantLock
Java Semaphore
SYNC + SIMPLE
ASYNC + SIMPLE
SYNC + SMART
ASYNC + SMART

Evaluation


Java ReentrantLock
Java Semaphore
SYNC + SIMPLE
ASYNC + SIMPLE
SYNC + SMART
ASYNC + SMART
permits = 1
permits = 2
permits = 4
permits = 8
us / op
threads
0
10
20
30
40
1
2
4
8
16
32
64
96

Let's use `SegmentQueueSynchronizer` for primitives other than `Semaphore`!

CountDownLatch

Allows to wait until several operations are completed

```
class CountDownLatch(count: Int) {

  fun countDown() { ... }

  fun await(): Future<Unit> { ... }
}
```

CountDownLatch

```
resume_mode = ASYNC
```

```
class CountDownLatch(count: Int) {
  val count = count
  val waiters = 0
  ...
}
```

CountDownLatch

`resume_mode = ASYNC`

```
class CountDownLatch(count: Int) {
  val count = count
  val waiters = 0
  ...
}
```

```
fun countDown() {
  val c = FAA(&count, -1)
  if c <= 0: resumeWaiters()
}
```

CountDownLatch

`resume_mode = ASYNC`

```
class CountDownLatch(count: Int) {
  val count = count
  val waiters = 0
  ...
}
```

```
fun countDown() {
  val c = FAA(&count, -1)
  if c <= 0: resumeWaiters()
}

fun resumeWaiters() = while (true) {
  val w = waiters
  if w & DONE_BIT != 0: return
  if CAS(&waiters, w, w & DONE_BIT) {
    repeat(w) { resume(Unit) }
  }
}
```

CountDownLatch

`resume_mode = ASYNC`

```
class CountDownLatch(count: Int) {
  val count = count
  val waiters = 0
  ...
}
```

```
fun await(): Future<Unit> {
   if count <= 0:
     return FutureImmediate(Unit)
   val w = FAA(&waiters, +1)
   if w & DONE_BIT != 0:
     return FutureImmediate(Unit)
   return suspend()
}
```

```
fun countDown() {
  val c = FAA(&count, -1)
  if c <= 0: resumeWaiters()
}

fun resumeWaiters() = while (true) {
  val w = waiters
  if w & DONE_BIT != 0: return
  if CAS(&waiters, w, w & DONE_BIT) {
    repeat(w) { resume(Unit) }
  }
}
```

Pools

```
resume_mode = ASYNC
```

```
class BlockingPool() {
  // < 0 => # waiters
  var elements = 0
  ...
}
```

Pools

```
resume_mode = ASYNC
```

```
class BlockingPool() {
  // < 0 => # waiters
  var elements = 0
  ...
}
```

```
fun put(element: T) {
  val e = FAA(&elements, 1)
  if e < 0 {
    resume(element)
  } else {
     insertIntoPool(element)
  }
}
```

Pools

```
resume_mode = ASYNC
```

```
class BlockingPool() {
  // < 0 => # waiters
  var elements = 0
  ...
}
```

```
fun put(element: T) {
  val e = FAA(&elements, 1)
  if e < 0 {
    resume(element)
  } else {
     insertIntoPool(element)
  }
}
```

```
fun retrieve(): Future<T> {
   val e = FAA(&elements, -1)
   if (e > 0) {
     val elem = retrieveFromPool()
     return FutureImmediate(elem)
   } else {
     return suspend()
   }
}
```

Pools

```
resume_mode = ASYNC
```

```
class BlockingPool() {
  // < 0 => # waiters
  var elements = 0
  ...
}
```

```
fun put(element: T) {
  val e = FAA(&elements, 1)
  if e < 0 {
    resume(element)
  } else {
     insertIntoPool(element)
  }
}
```

```
fun retrieve(): Future<T> {
   val e = FAA(&elements, -1)
   if (e > 0) {
     val elem = retrieveFromPool()
     return FutureImmediate(elem)
   } else {
     return suspend()
```

`insertIntoPool` and **`retrieveFromPool`** can use any data structure under the hood, e.g. queue, stack, or just bag.

Pools

```
resume_mode = ASYNC
```

```
class BlockingPool() {
  // < 0 => # waiters
  var elements = 0
  ...
}
```

Smart cancellation mode can be used here as well

```
fun put(element: T) {
  val e = FAA(&elements, 1)
  if e < 0 {
    resume(element)
  } else {
     insertIntoPool(element)
  }
}
```

```
): Future<T> {
  val e = FAA(&elements, -1)
  if (e > 0) {
    val elem = retrieveFromPool()
    return FutureImmediate(elem)
  } else {
    return suspend()
```

insertIntoPool and **retrieveFromPool** can use any data structure under the hood, e.g. queue, stack, or just bag.

Conclusion and Links

- Fair synchronization primitives can be simple and fast
- Stronger guarantees ⇒ more complicated code
  -

Conclusion and Links

- Fair synchronization primitives can be simple and fast
- Stronger guarantees ⇒ more complicated code

- Kotlin Coroutines project
  github.com/Kotlin/kotlinx.coroutines
- The experiments (`sqs-experiments` branch)
  github.com/Kotlin/kotlinx.coroutines/tree/segment-queue-synchronizer

Questions?