Channels in Kotlin Coroutines*

Nikita Koval, Joker 2018

* A look into the future

Attention! This talk is about concurrency and algorithms!

Micronaut vs Spring Boot, or who's the smallest here?

Kirill Tolkachev
CIAN

Maxim Gorelikov
CIAN

#framework #micro #cloud

RU

Apache Maven supports ALL Java

Robert Scholte
Sourcegrounds

#buildtools #mavenchairman #java11 #jigsaw

EN

Channels in Kotlin coroutines

Nikita Koval
JetBrains

#concurrency #algorithms

RU

How to tune Spark performance for ML needs

Artem Shutak
Grid Dynamics

#bigdata #ml

RU

Speaker: Nikita Koval

@nkoval_

- Graduated at ITMO University
- Previously worked as developer and researcher at Devexperts
- Teaches concurrent programming at ITMO University
- PhD student at IST Austria
- Researcher at JetBrains

What coroutines are

- Lightweight threads, can be suspended and resumed for free
    - You can run millions of coroutines and not die!

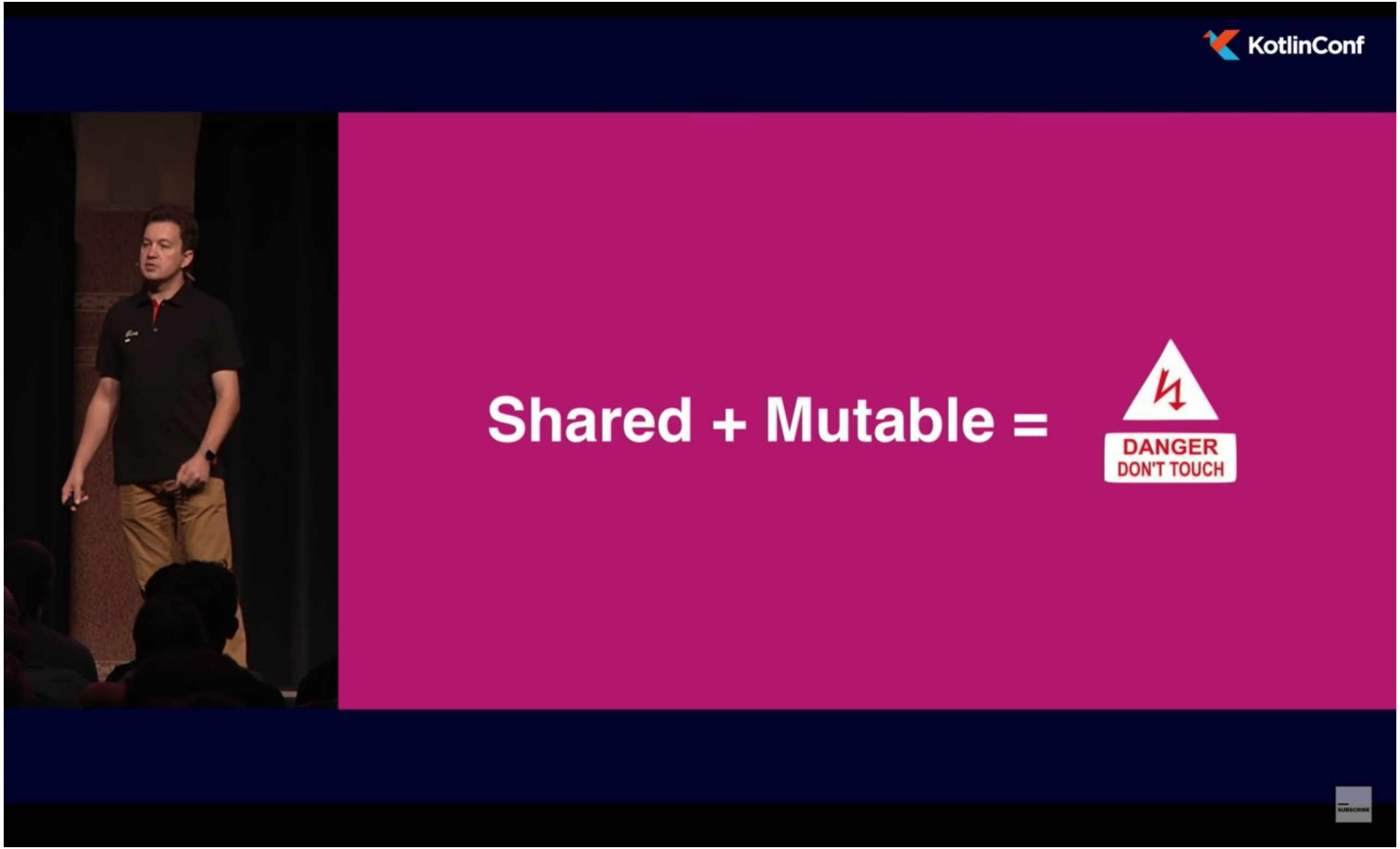
What coroutines are

- Lightweight threads, can be suspended and resumed for free
  - You can run millions of coroutines and not die!

- Support writing an asynchronous code like a synchronous one

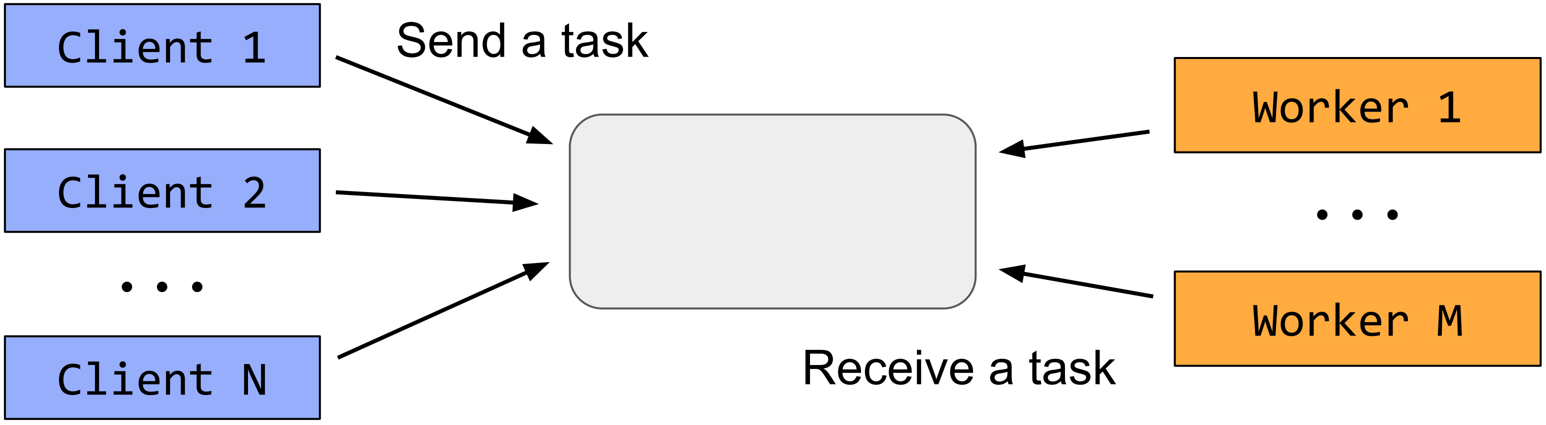
```
fun postItem(item: Item) {
    val token = requestToken()
    val post = createPost(token, item)
    processPost(post)
}
```

suspend functions



“Kotlin Coroutines in Practice” by Roman Elizarov @ KotlinConf 2018

Producer-consumer problem



* Both clients and consumers are coroutines

Producer-consumer problem solution

1. Let’s create a channel

```
val tasks = Channel<Task>()
```

Producer-consumer problem solution

1. Let’s create a channel

```
val tasks = Channel<Task>()
```

2. Clients send tasks to workers through this channel

```
val task = Task(...)
tasks.send(task)
```

Producer-consumer problem solution

1. Let's create a channel

```
val tasks = Channel<Task>()
```

2. Clients send tasks to workers through this channel

```
val task = Task(...)
tasks.send(task)
```

3. Workers receive tasks in infinite loop

```
while(true) {
    val task = tasks.receive()
    processTask(task)
}
```

Channel semantics

Client 1

```
val task = Task(...)
tasks.send(task)
```

Client 2

```
val task = Task(...)
tasks.send(task)
```

Worker

```
while(true) {
  val task = tasks.receive()
  processTask(task)
}
```

```
val tasks = Channel<Task>()
```

Channel semantics

Client 1

```
val task = Task(...)
tasks.send(task)
```

Client 2

```
val task = Task(...)
tasks.send(task)
```

Worker

```
while(true) {
  val task = tasks.receive()
  processTask(task)
}
```

1

Have to wait for send

```
val tasks = Channel<Task>()
```

Channel semantics

Client 1

```
val task = Task(...)
tasks.send(task)
```

Client 2

```
val task = Task(...)
tasks.send(task)
```

Worker


```
while(true) {
  val task = tasks.receive()
  processTask(task)
}
```

1

```
val tasks = Channel<Task>()
```

Channel semantics

Client 1

```
val task = Task(...)
tasks.send(task)
```

Client 2

```
val task = Task(...)
tasks.send(task)
```

Worker


```
while(true) {
  val task = tasks.receive()
  processTask(task)
}
```

1

```
val tasks = Channel<Task>()
```

Channel semantics

Client 1

```
val task = Task(...)
tasks.send(task)
```

2

Rendezvous!

Worker

```
while(true) {
    val task = tasks.receive()
    processTask(task)
}
```

1

Client 2

```
val task = Task(...)
tasks.send(task)
```

```
val tasks = Channel<Task>()
```

Channel semantics

Client 1

```
  val task = Task(...)
2 tasks.send(task)
```

Client 2

```
val task = Task(...)
tasks.send(task)
```

Worker

```
  while(true) {
1   val task = tasks.receive()
3   processTask(task)
  }
```

```
val tasks = Channel<Task>()
```

Channel semantics

Client 1

```
    val task = Task(...)
(2) tasks.send(task)
```

Worker

```
    while(true) {
(1)     val task = tasks.receive()
(3)     processTask(task)
    }
```

Client 2

```
    val task = Task(...)
(4) tasks.send(task)
```

Have to wait for receive

```
val tasks = Channel<Task>()
```

Channel semantics

Client 1

```
  val task = Task(...)
2 tasks.send(task)
```

Worker

```
  while(true) {
1   val task = tasks.receive()
3   processTask(task)
  }
```

Client 2


```
  val task = Task(...)
4 tasks.send(task)
```

```
val tasks = Channel<Task>()
```

Channel semantics

Client 1

```
    val task = Task(...)
(2) tasks.send(task)
```

Worker

```
      while(true) {
(5)(1)  val task = tasks.receive()
(3)     processTask(task)
      }
```

Client 2

```
    val task = Task(...)
(4) tasks.send(task)
```

Rendezvous!

```
val tasks = Channel<Task>()
```

Rendezvous channel

Rendezvous channel

```
class Coroutine {
   var element: Any?
   ...
}

fun curCoroutine(): Coroutine { ... }

suspend fun suspend(c: Coroutine) { ... }
fun resume(c: Coroutine) { ... }
```

Stores an element to be sent

Returns the current coroutine

Functions to manipulate with coroutines

Rendezvous channel

```
class Coroutine {
   var element: Any?
   ...
}

fun curCoroutine(): Coroutine { ... }

suspend fun suspend(c: Coroutine) { ... }
fun resume(c: Coroutine) { ... }

val senders   = Queue<Coroutine>()
val receivers = Queue<Coroutine>()
```

Queues of suspended send and receive invocations

Rendezvous channel

```
class Coroutine {
    var element: Any?
    ...
}

fun curCo

suspend fun suspend(c: Coroutine) { ... }
fun resume(c: Coroutine) { ... }

val senders   = Queue<Coroutine>()
val receivers = Queue<Coroutine>()
```

Checks if there is no receiver and suspends

Rendezvous: retrieves the first receiver

```
suspend fun send(element: T) {
    if (receivers.isEmpty()) {
        val curCor = curCoroutine()
        curCor.element = element
        senders.enqueue(curCor)
        suspend(curCor)
    } else {
        val r = receivers.dequeue()
        r.element = element
        resume(receiver)
    }
}
```

Rendezvous channel: Golang

Rendezvous channel: Golang

Uses per-channel locks

```
suspend fun send(element: T) = channelLock.withLock {
    if (receivers.isEmpty()) {
        val curCor = curCoroutine()
        curCor.element = element
        senders.enqueue(curCor)
        suspend(curCor)
    } else {
        val r = receivers.dequeue()
        r.element = element
        resume(receiver)
    }
}
```

Rendezvous channel: Golang

Uses per-channel locks

```
suspend fun send(element: T) = channelLock.withLock {
    if (receivers.isEmpty()) {
        val curCor = curCoroutine()
        curCor.element = element
        senders.enqueue(curCor)
        suspend(curCor)
    } else {
        val r = receivers.dequeue()
        r.element = element
        resume(receiver)
    }
}
```

Non-scalable, no progress guarantee...

Modern queues use `Fetch-And-Add`... Let's try to use the same ideas for channels!


PPoPP'13

Fast Concurrent Queues for x86 Processo[rs]

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

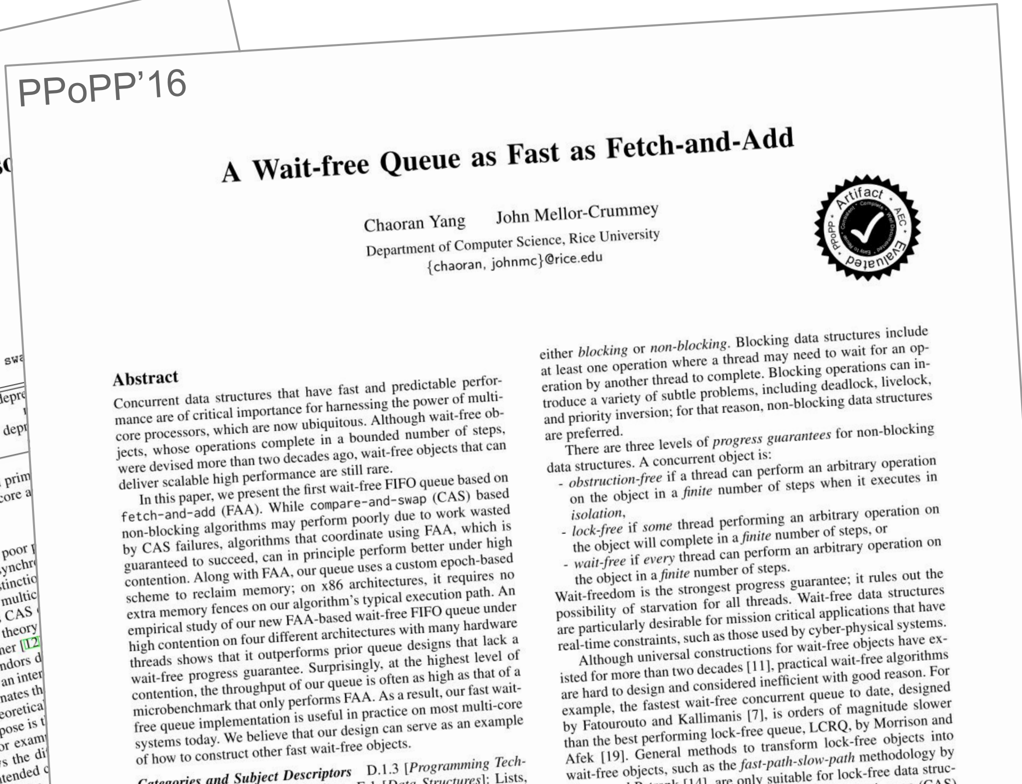
Abstract
Conventional wisdom in designing concurrent data structures is to use the most powerful synchronization primitive, namely compare-and-swap (CAS), and to avoid contended hot spots. In building concurrent FIFO queues, this reasoning has led researchers to propose *combining-based* concurrent queues.
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

Keywords concurrent queue, nonblocking algorithm, fetch-and-

	compare-and-swap	swa…
ARM	LL/SC	depre…
POWER	LL/SC	
SPARC	yes	depr…
x86	**yes**	

Table 1: Synchronization prim… tions on dominant multicore a…


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 multicore 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 wait-free objects, such as the *fast-path-slow-path* methodology by

Concurrent primitives

- `Fetch-And-Add(p, val): Long`
  Atomically increments the located by address `p` register by `val` and returns the new value

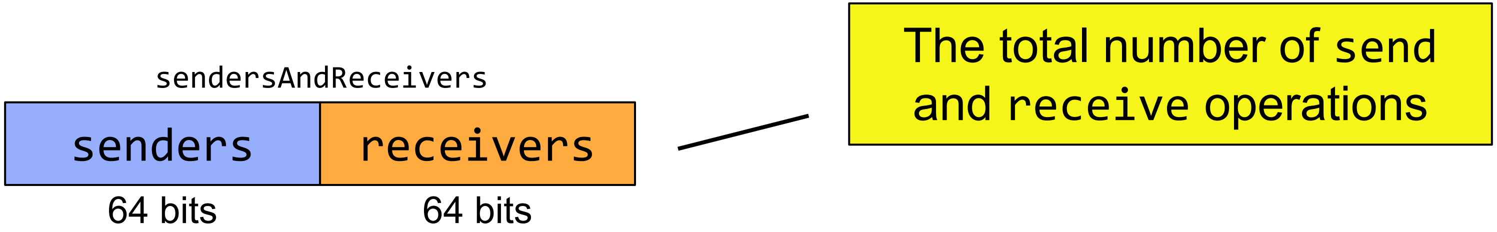
- `Compare-And-Swap(p, old, new): Boolean`
  Atomically checks if the located by address `p` value equals `old` and replaces it with `new`

Rendezvous channel: Kotlin

- Assume we have an atomic 128-bit register
  - That is not true; we will fix this later

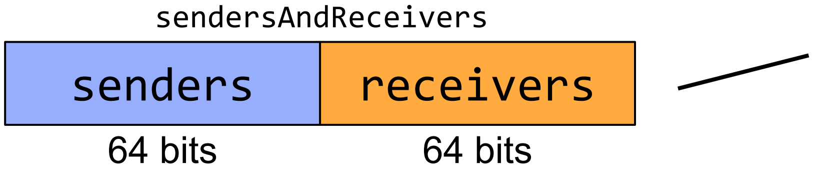
Rendezvous channel: Kotlin

- Assume we have an atomic 128-bit register
  - That is not true; we will fix this later



Rendezvous channel: Kotlin

- Assume we have an atomic 128-bit register
  - That is not true; we will fix this later



The total number of `send` and `receive` operations

- Every `send` and `receive` increments its counter using `FAA`
  - `send` increments the register by `(1 << 64)`
  - `receive` increments the register by `1`

Rendezvous channel: Kotlin

- Each `send-receive` pair works with an unique cell
- This cell id is either `senders` or `receivers` counter after the increment (for `send` and `receive` respectively)

- How to understand if we can make a rendezvous?

Rendezvous channel: Kotlin

```
when {
   senders < receivers  -> // make a rendezvous
   senders >= receivers -> // suspend
}
```

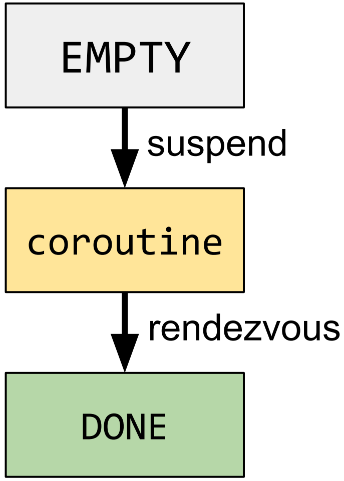
The balance before the send operation

Rendezvous channel: Kotlin

```
when {
   senders < receivers  -> // make a rendezvous
   senders >= receivers -> // suspend
}
```

The balance before the send operation

Cell life cycle

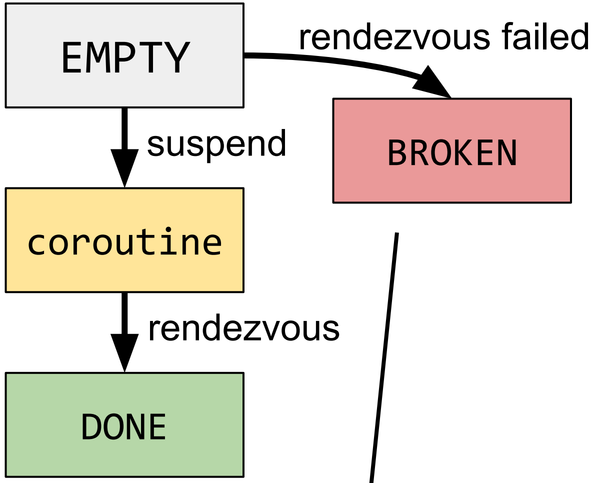


Rendezvous channel: Kotlin

```
when {
    senders < receivers  -> // make a rendezvous
    senders >= receivers -> // suspend
}
```

The balance before the send operation

Cell life cycle



This helps not to block

Rendezvous channel: Kotlin

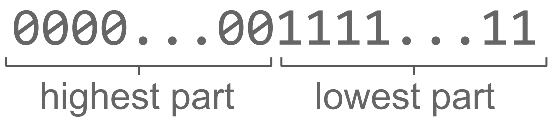
1. How to implement an *atomic* 128-bit counter using 64-bit ones?

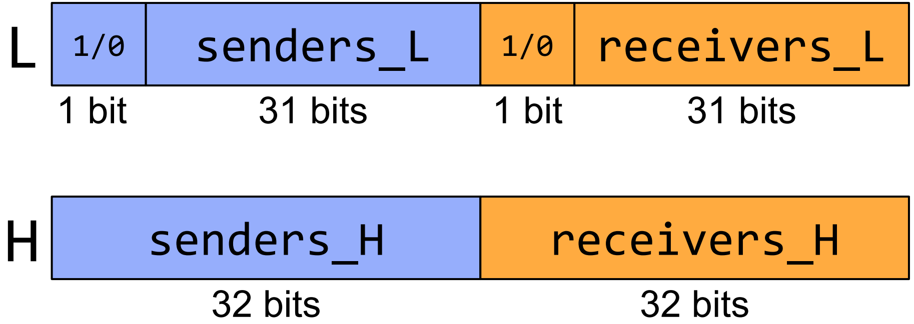
2. How to organize the cell storage?

Rendezvous channel: Kotlin

1. How to implement an *atomic* 128-bit counter using 64-bit ones?
2. How to organize the cell storage?

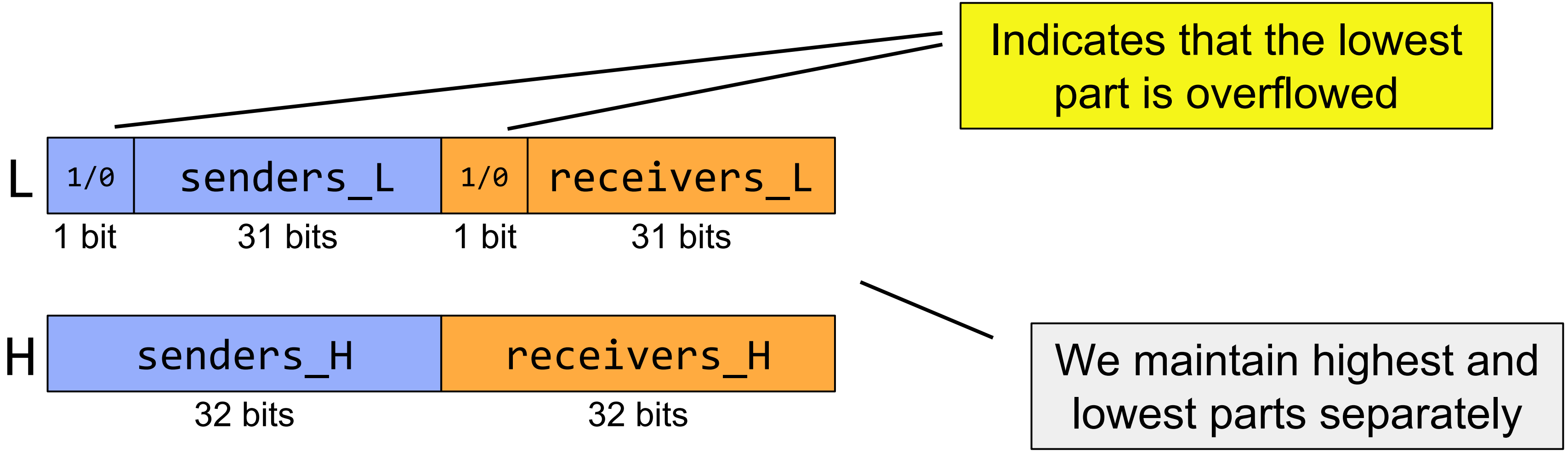
Rendezvous channel: Kotlin



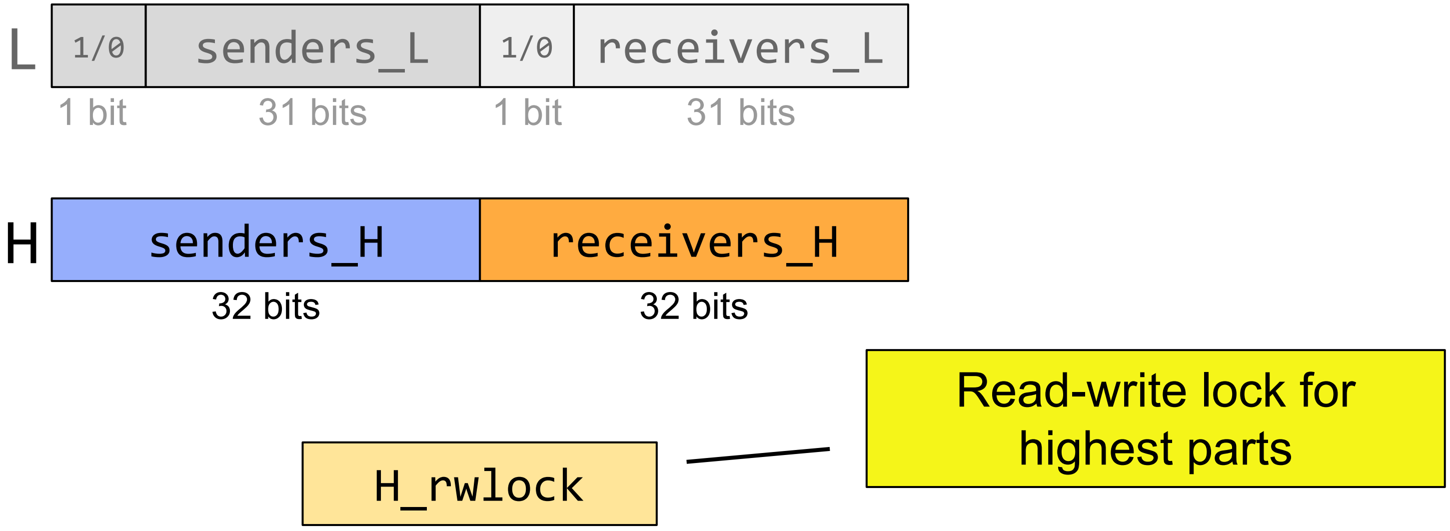


We maintain highest and lowest parts separately

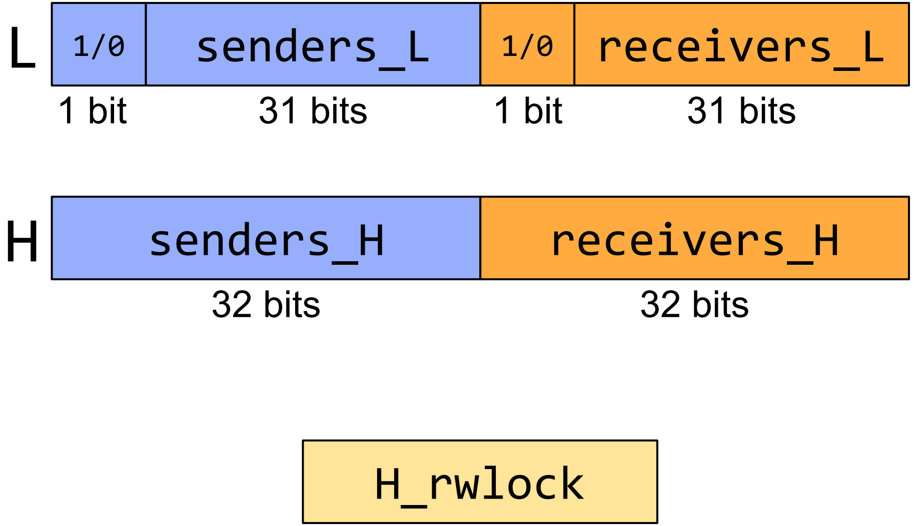
Rendezvous channel: Kotlin



Rendezvous channel: Kotlin



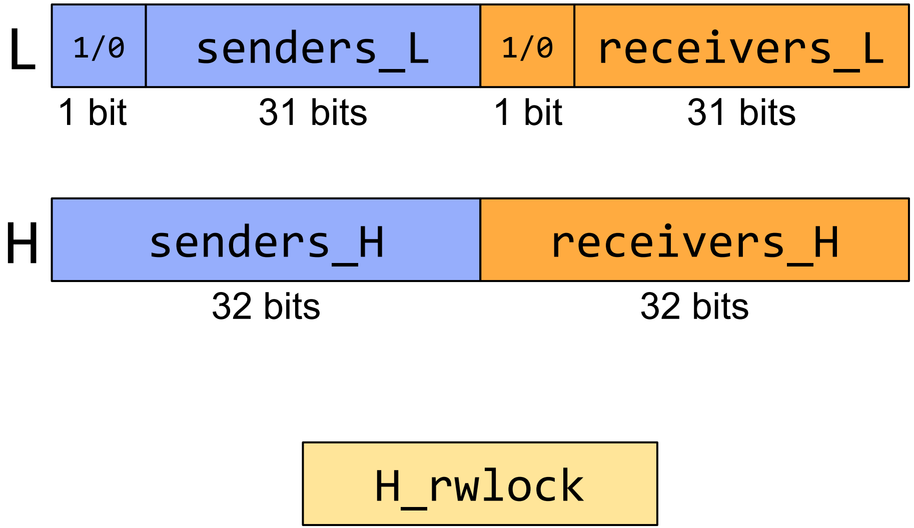
Rendezvous channel: Kotlin



Increment algorithm:

1. Acquire `H_rwlock` for read
2. Read `H`
3. Inc `L` by `FAA`
4. Release the lock

Rendezvous channel: Kotlin

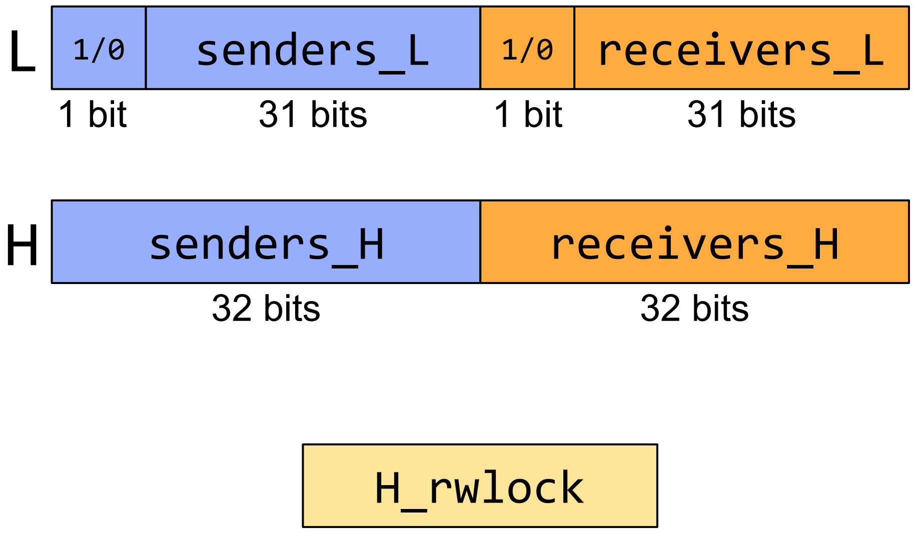


Increment algorithm:

1. Acquire H_rwlock for read
2. Read H
3. Inc L by FAA
4. Release the lock

Just a FAA

Rendezvous channel: Kotlin



Increment algorithm:

1. Acquire `H_rwlock` for read
2. Read `H`
3. Inc `L` by `FAA`
4. Release the lock
5. If the lowest part is overflowed
   5.1. Acquire `H_rwlock` for write
   5.2. Reset the bit
   5.3. Inc `H`
   5.4. Release the lock

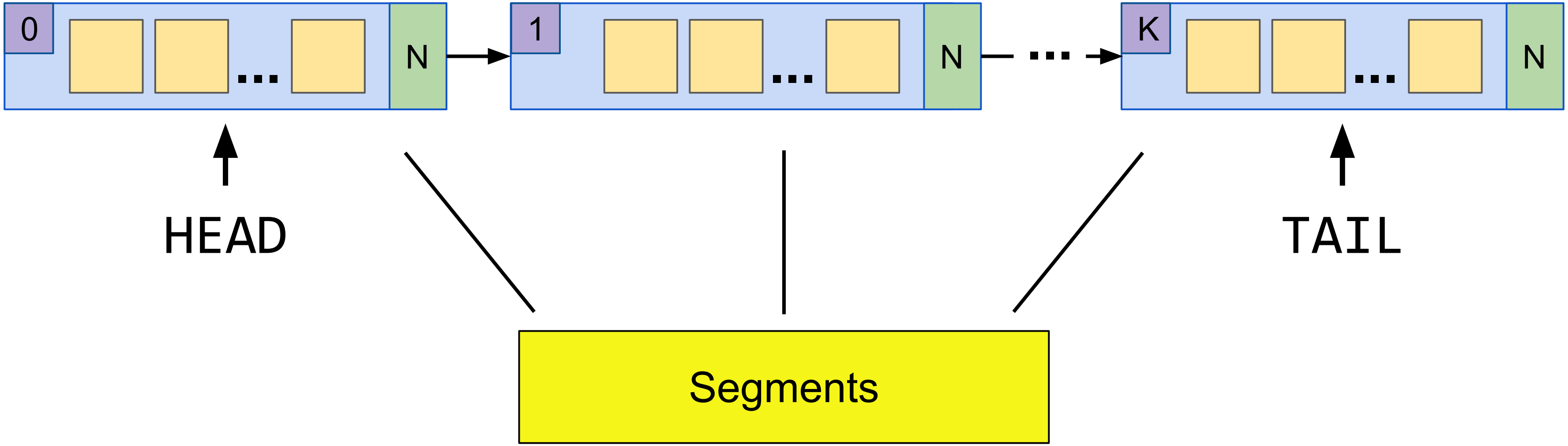
Rendezvous channel: Kotlin

1. How to implement an *atomic* 128-bit counter using 64-bit ones?

2. How to organize the cell storage?

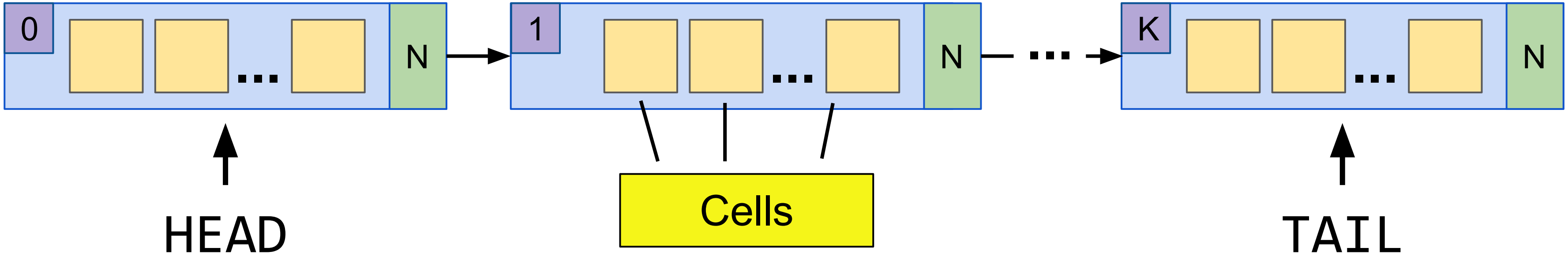
Rendezvous channel: Kotlin

Michael-Scott queue of segments with the fixed number of cells in each



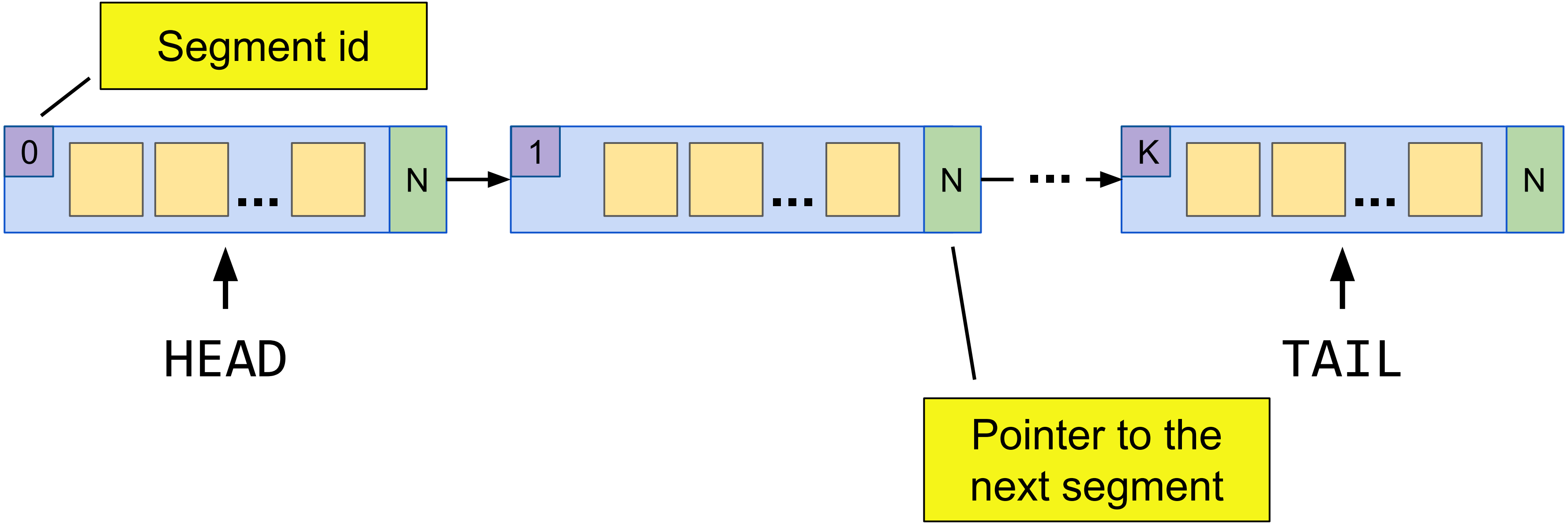
Rendezvous channel: Kotlin

Michael-Scott queue of segments with the fixed number of cells in each

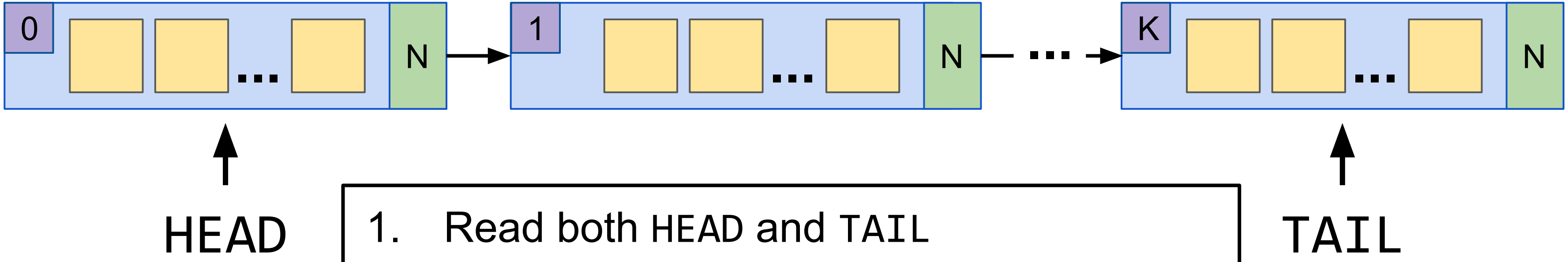


Rendezvous channel: Kotlin

Michael-Scott queue of segments with the fixed number of cells in each

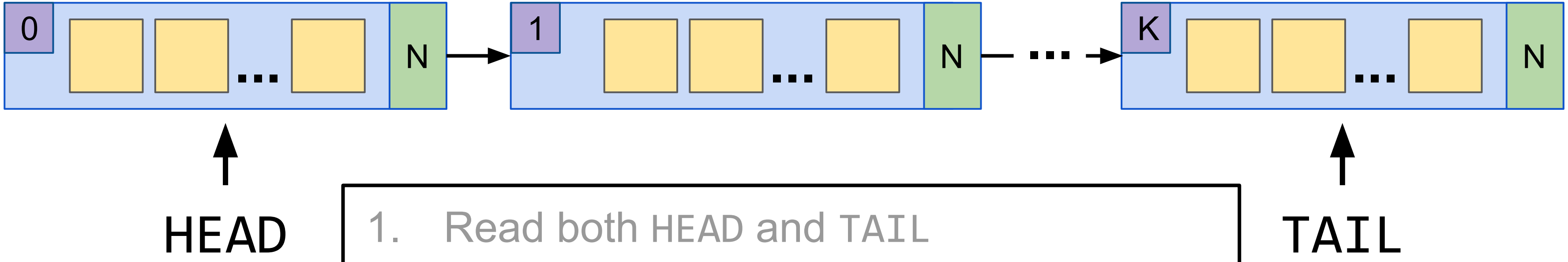


Rendezvous channel: Kotlin



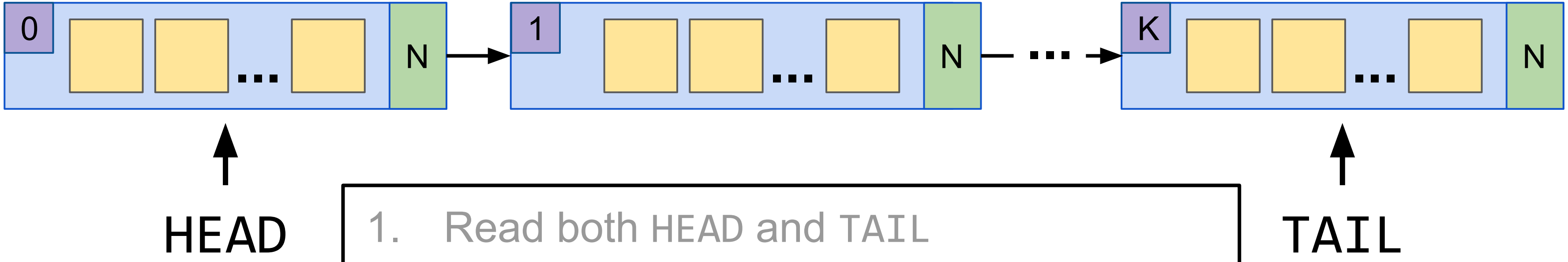
1. Read both `HEAD` and `TAIL`
2. Increment the counter

Rendezvous channel: Kotlin



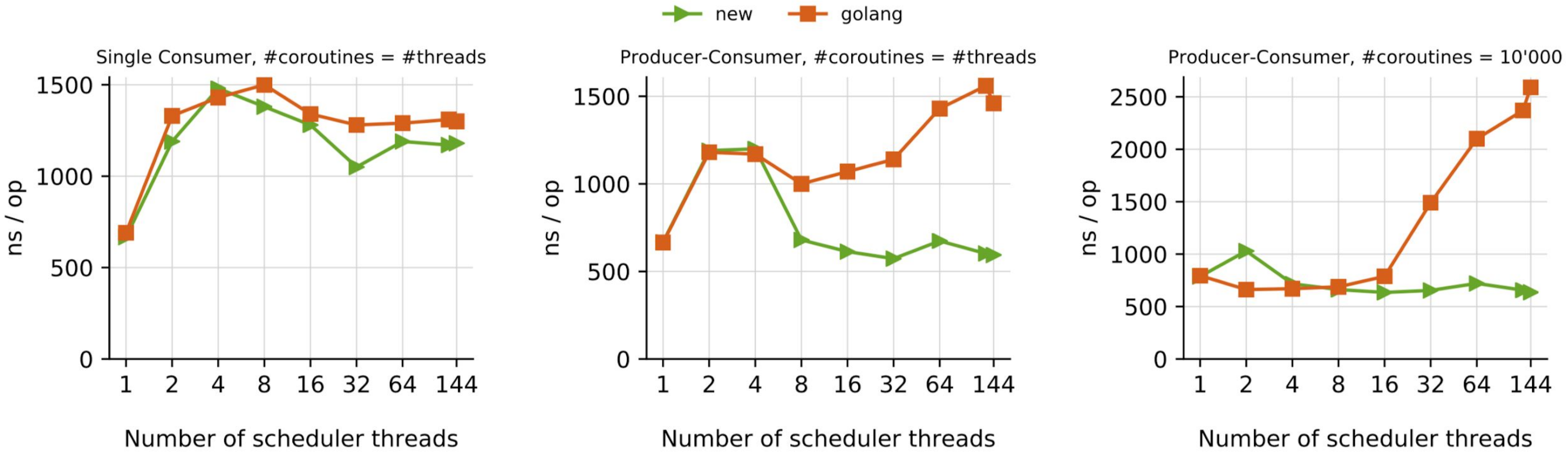
1. Read both `HEAD` and `TAIL`
2. Increment the counter
3. Either make a rendezvous
   3.1. Find the cell starting from the *head*
   3.2. Move `HEAD` forward if needed

Rendezvous channel: Kotlin



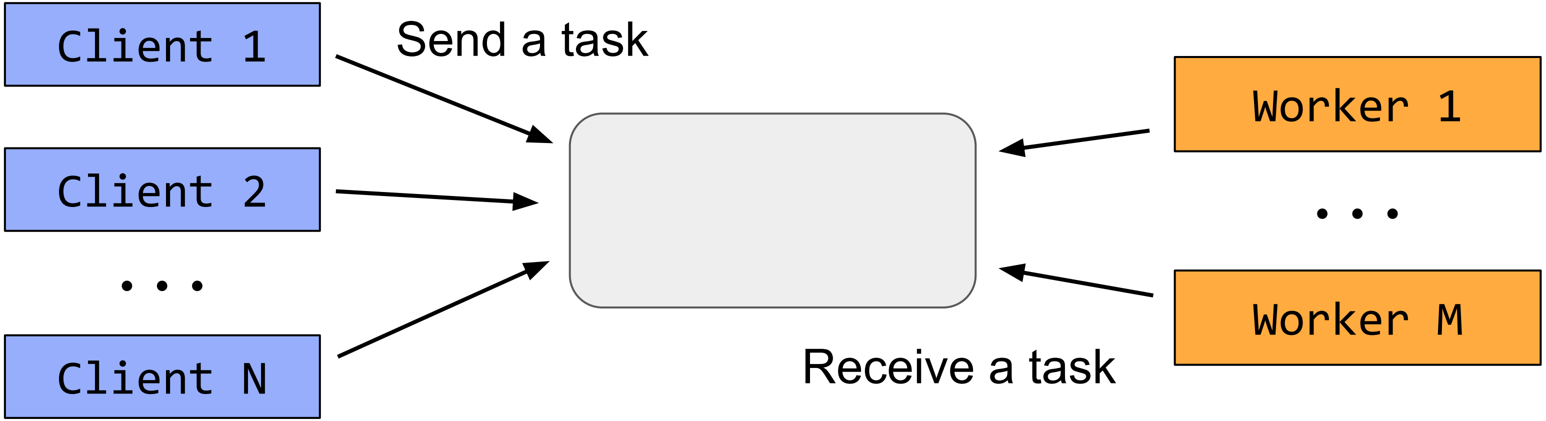
1. Read both `HEAD` and `TAIL`
2. Increment the counter
3. Either make a rendezvous
   3.1. Find the cell starting from the *head*
   3.2. Move `HEAD` forward if needed
4. or suspend
   4.1. Find the cell starting from the *tail*
   4.2. Create new segments if needed

Rendezvous channel: Kotlin vs Golang



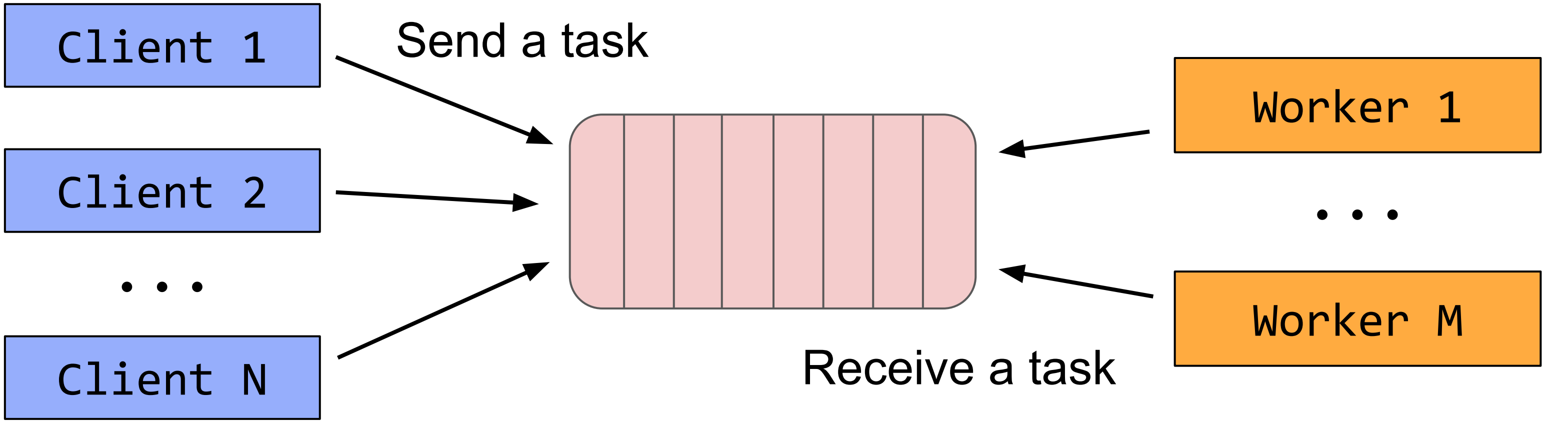
4 x 2 x Intel Xeon Gold 6150 (Skylake) 2.7GHz = (144 virtual cores)

Producer-consumer problem: buffering



We don't want to wait on every send...

Producer-consumer problem: buffering



Let’s use a fixed-size buffer!

Buffered channel semantics

Client 1

```
val task = Task(...)
tasks.send(task)
```

Client 2

```
val task = Task(...)
tasks.send(task)
```

Worker

```
while(true) {
  val task = tasks.receive()
  processTask(task)
}
```

One element can be sent without suspension

```
val tasks = Channel<Task>(capacity = 1)
```

Buffered channel semantics

Client 1

```
val task = Task(...)
tasks.send(task)
```

1

Does not suspend!

Client 2

```
val task = Task(...)
tasks.send(task)
```

Worker

```
while(true) {
  val task = tasks.receive()
  processTask(task)
}
```

```
val tasks = Channel<Task>(capacity = 1)
```

Buffered channel semantics

Client 1

```
    val task = Task(...)
1   tasks.send(task)
```

Worker

```
while(true) {
  val task = tasks.receive()
  processTask(task)
}
```

Client 2

```
    val task = Task(...)
2   tasks.send(task)
```

The buffer is full, has to suspend

```
val tasks = Channel<Task>(capacity = 1)
```

Buffered channel semantics

Client 1

```
val task = Task(...)
```

1 `tasks.send(task)`

Client 2

```
val task = Task(...)
```

2 `tasks.send(task)`

Worker

```
while(true) {
  val task = tasks.receive()
  processTask(task)
}
```

3 (at `val task = tasks.receive()`)

Receives the buffered element at first

```
val tasks = Channel<Task>(capacity = 1)
```

Buffered channel semantics

Client 1

1
```kotlin
val task = Task(...)
tasks.send(task)
```

Worker

4 3
```kotlin
while(true) {
  val task = tasks.receive()
  processTask(task)
}
```

Client 2

2
```kotlin
val task = Task(...)
tasks.send(task)
```

Makes a rendezvous with the 2nd client

```kotlin
val tasks = Channel<Task>(capacity = 1)
```

Buffered channel: Golang

- Maintains an additional fixed-size buffer
  - Tries to send to this buffer instead of suspending

- Performs all operations under the channel lock

Buffered channel: Kotlin

Rendezvous channel

```
when {
   senders - receivers < 0  -> // make a rendezvous
   senders - receivers >= 0 -> // suspend
}
```

The balance before the send operation

Buffered channel: Kotlin

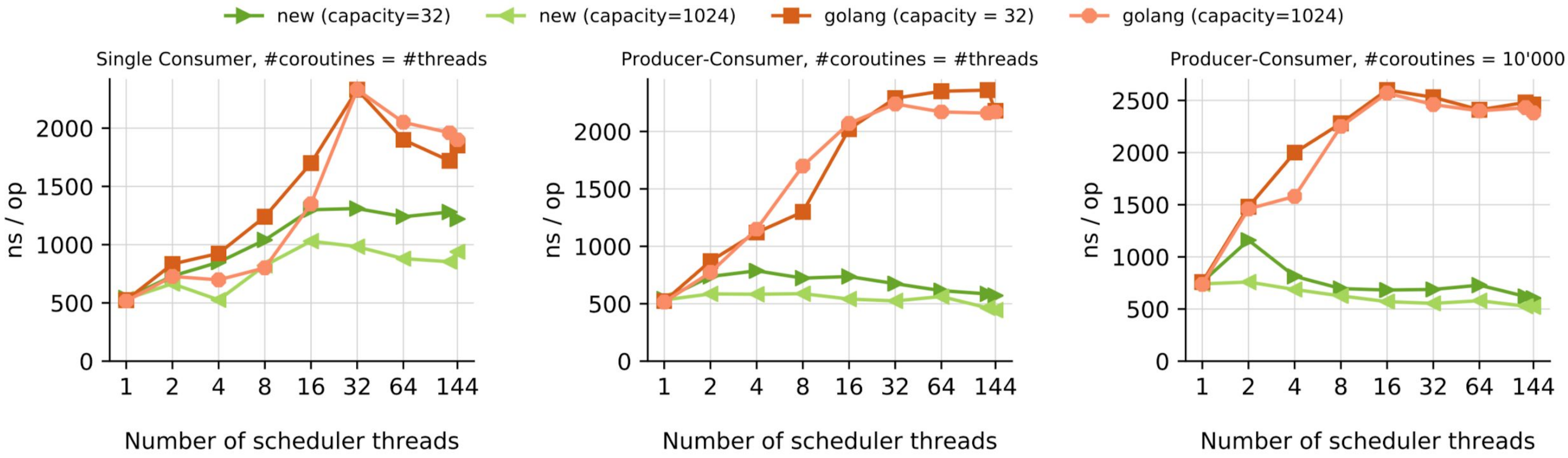
Rendezvous channel

```
when {
   senders - receivers < 0  -> // make a rendezvous
   senders - receivers >= 0 -> // suspend
}
```

Buffered channel

```
when {
        senders - receivers < 0          -> // make a rendezvous
   0 <= senders - receivers < capacity  -> // send the element without suspension
        senders - receivers >= capacity -> // suspend
}
```

Buffered channel: Kotlin vs Golang



4 x 2 x Intel Xeon Gold 6150 (Skylake) 2.7GHz = (144 virtual cores)

The `select` expression

Client

```
val task = Task(...)
tasks.send(task)
```

Suspends here

The `select` expression

Client

```
val task = Task(...)
tasks.send(task)
```

The `select` expression

~~Client~~


```
val task = Task(...)
tasks.send(task)
```

The client was interrupted while waiting for a worker

The `select` expression

~~Client~~

```
val task = Task(...)
tasks.send(task)
```

The client was interrupted while waiting for a worker

Do we need to process the task anymore?

The `select` expression

~~Client~~

```
val task = Task(...)
tasks.send(task)
```

The client was interrupted while waiting for a worker

Do we need to process the task anymore?

It would be better to cancel the request and detect this

The `select` expression

Client

```
val task = Task(...)
val cancelled = Channel<Unit>()
```

`Unit` is sent to this channel if the client is interrupted

The `select` expression

Client

```
val task = Task(...)
val cancelled = Channel<Unit>()
select<Unit> {
    tasks.onSend(task) { println("Task has been sent") }
    cancelled.onReceive { println("Cancelled") }
}
```

Waits simultaneously, at most one clause is selected *atomically*.

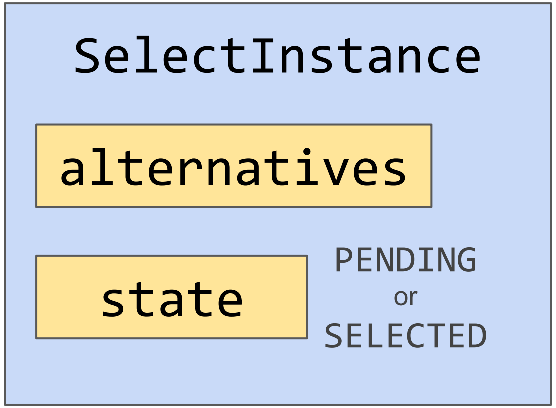
The `select` expression: Golang

- Fine-grained locking

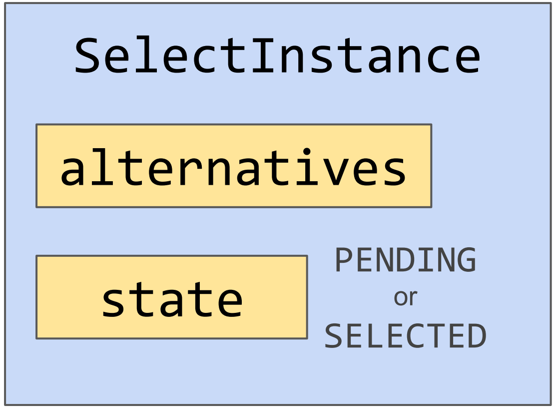
- Acquires all involved channels locks to register into the queues
  - Uses hierarchical order to avoid deadlocks

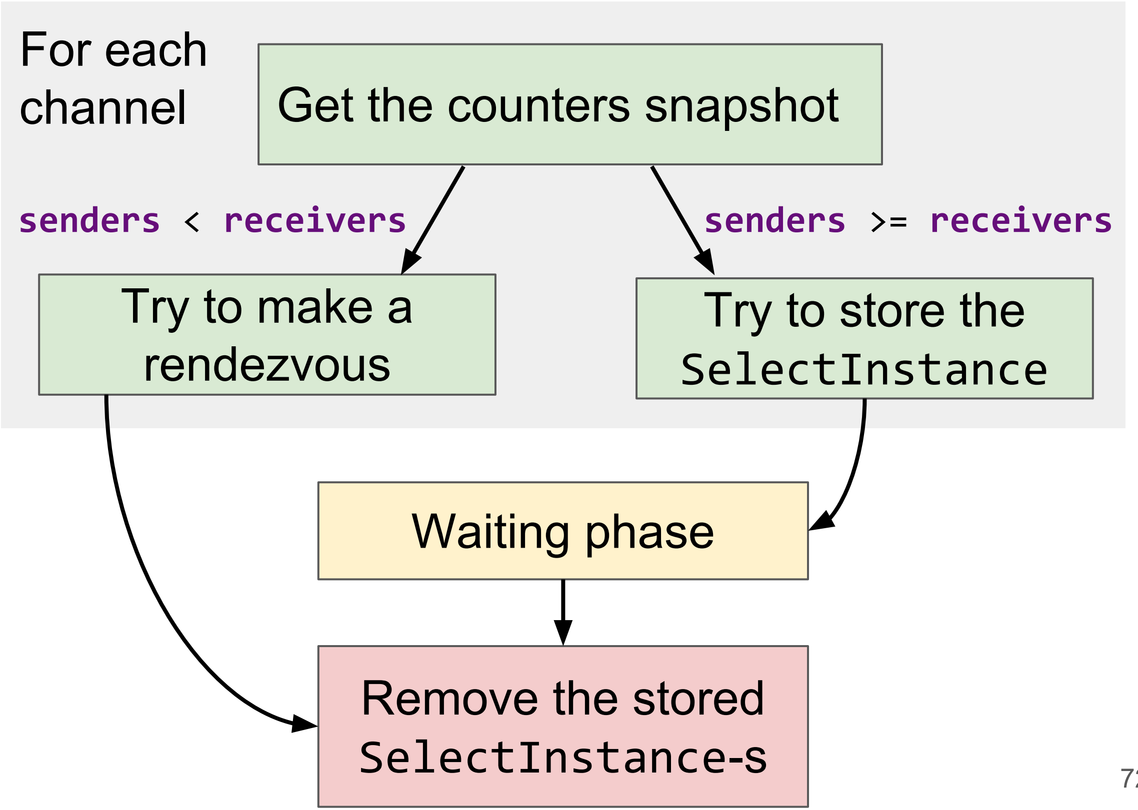
- Acquires all these locks again to resume the coroutine
  - Otherwise, two select clauses could interfere

The `select` expression: Kotlin

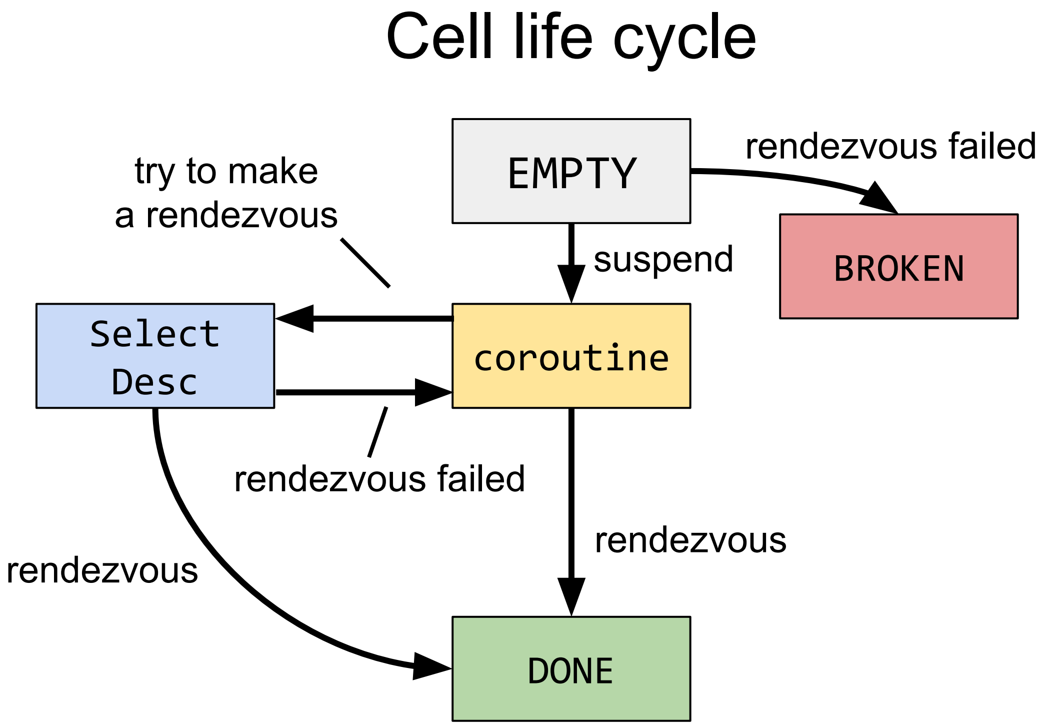


The `select` expression: Kotlin





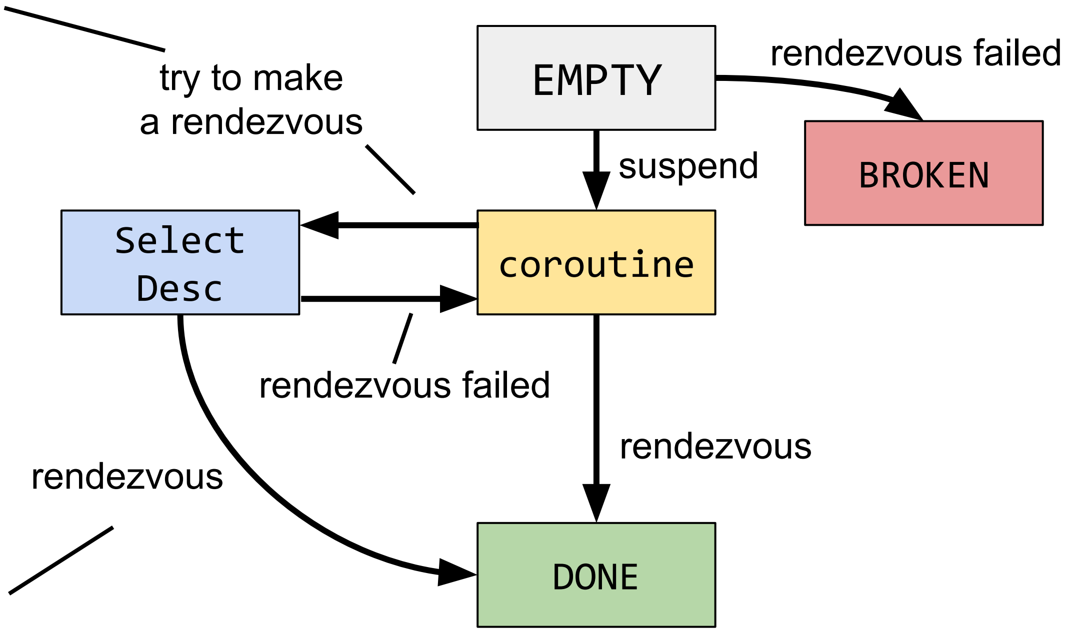
The `select` expression: Kotlin



The `select` expression: Kotlin

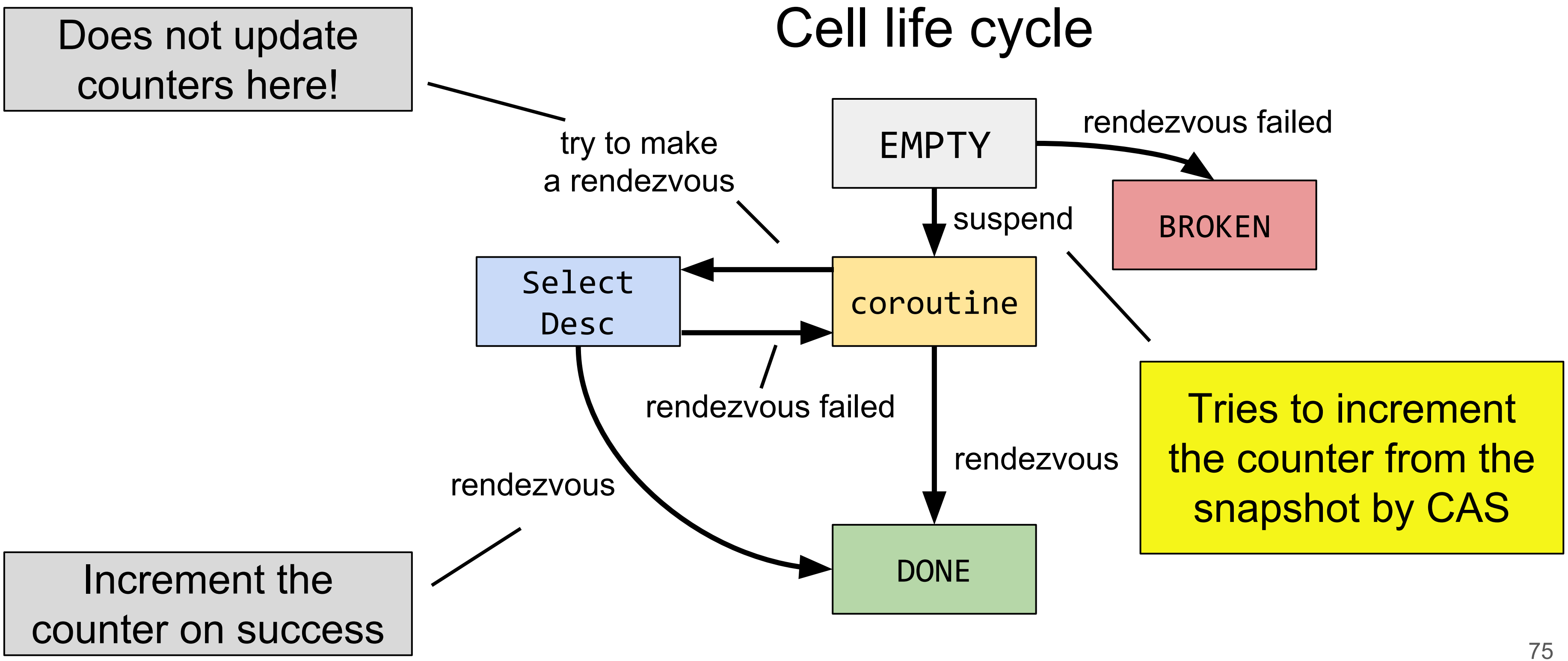
Does not update counters here!

Cell life cycle



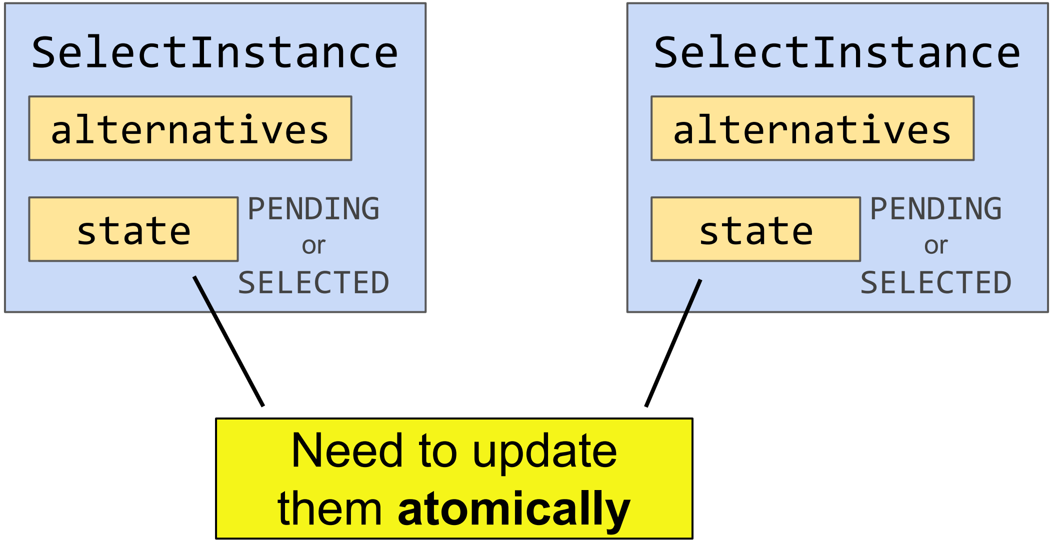
Increment the counter on success

The `select` expression: Kotlin



The `select` expression: Kotlin

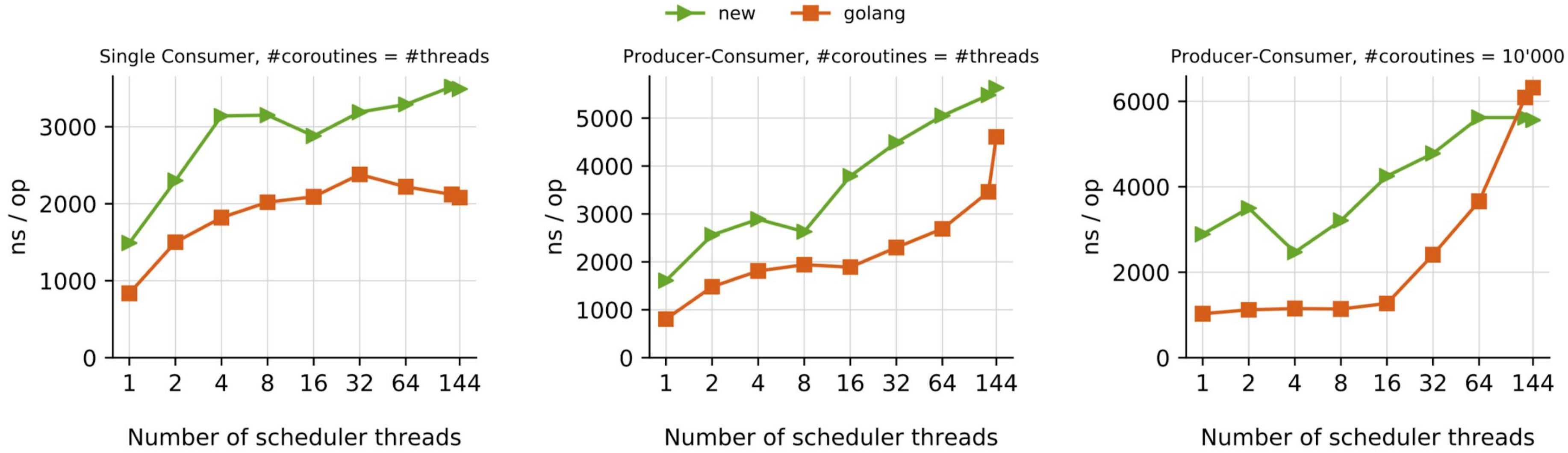
A rendezvous between two selects:



Let's update them similarly to the Harris multiword CAS*

* "A practical multi-word compare-and-swap operation" by Harris et al. @ DISC'02

The `select` expression: Kotlin vs Golang



4 x 2 x Intel Xeon Gold 6150 (Skylake) 2.7GHz = (144 virtual cores)

Cancellation in Kotlin

- Cancellation is a built-in feature in Kotlin
  - However, the previous pattern is widely used in Golang

```
val job = GlobalScope.launch { ... }
job.cancel()
```

Removes the coroutine from all channels as well

Cancellation in Kotlin

- Cancellation is a built-in feature in Kotlin
  - However, the previous pattern is widely used in Golang

```
val job = GlobalScope.launch { ... }
job.cancel()
```

Invokes `job.cancel()` after the timeout

```
GlobalScope.launch {
    withTimeout(time = 1, unit = TimeUnit.SECONDS) {
        ...
    }
}
```

There are more message passing primitives

- `BroadcastChannel`
  Sends to multiple receivers
- `ConflatedChannel`
  Receivers always get the most recently sent element
- `ConflatedBroadcastChannel`
  Mix of the previous ones

- `Mutex`

Industry

+

Academia

=