Сказ о том, как мы алгоритм каналов в Kotlin Coroutines делали

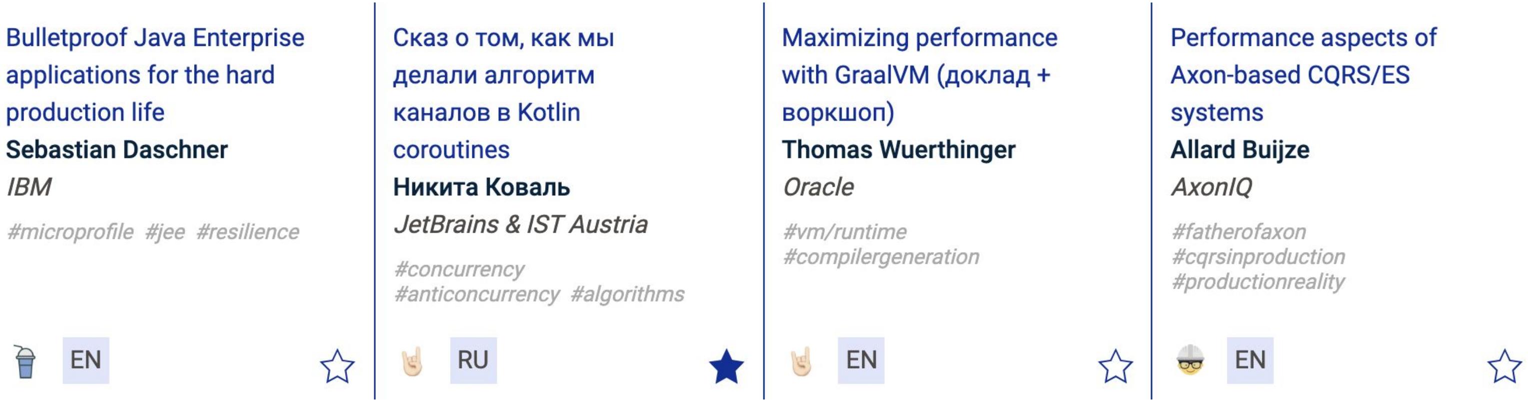
Никита Коваль, JPoint 2019

JET BRAINS

Kotlin

IST AUSTRIA

Attention! This talk is about concurrency and algorithms!



Bulletproof Java Enterprise applications for the hard production life
Sebastian Daschner
IBM
#microprofile #jee #resilience
EN ☆

Сказ о том, как мы делали алгоритм каналов в Kotlin coroutines
Никита Коваль
JetBrains & IST Austria
#concurrency #anticoncurrency #algorithms
RU ★

Maximizing performance with GraalVM (доклад + воркшоп)
Thomas Wuerthinger
Oracle
#vm/runtime #compilergeneration
EN ☆

Performance aspects of Axon-based CQRS/ES systems
Allard Buijze
AxonIQ
#fatherofaxon #cqrsinproduction #productionreality
EN ☆

Speaker: Nikita Koval

🐦 @nkoval_

- Graduated @ ITMO University
- Previously worked as developer and researcher @ Devexperts
- Teaching concurrent programming course @ ITMO University
- **Researcher @ JetBrains**
- PhD student @ IST Austria

What coroutines are

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

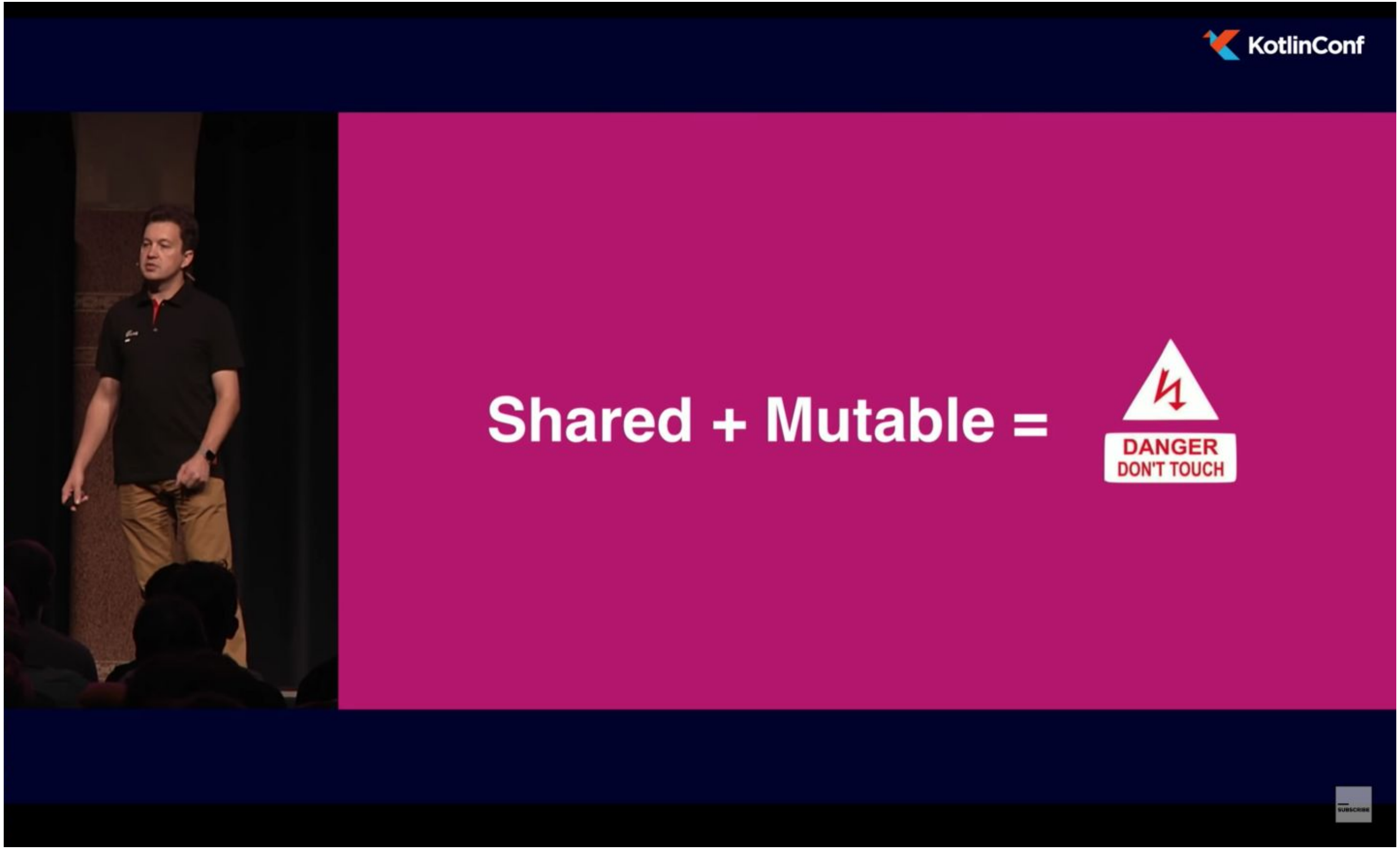
What coroutines are

- 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

```kotlin
suspend fun dbRequest(c: Client, r: Request) {
    val token = requestToken(c)
    val result = doDbRequest(token, r)
    processResult(result)
}
```

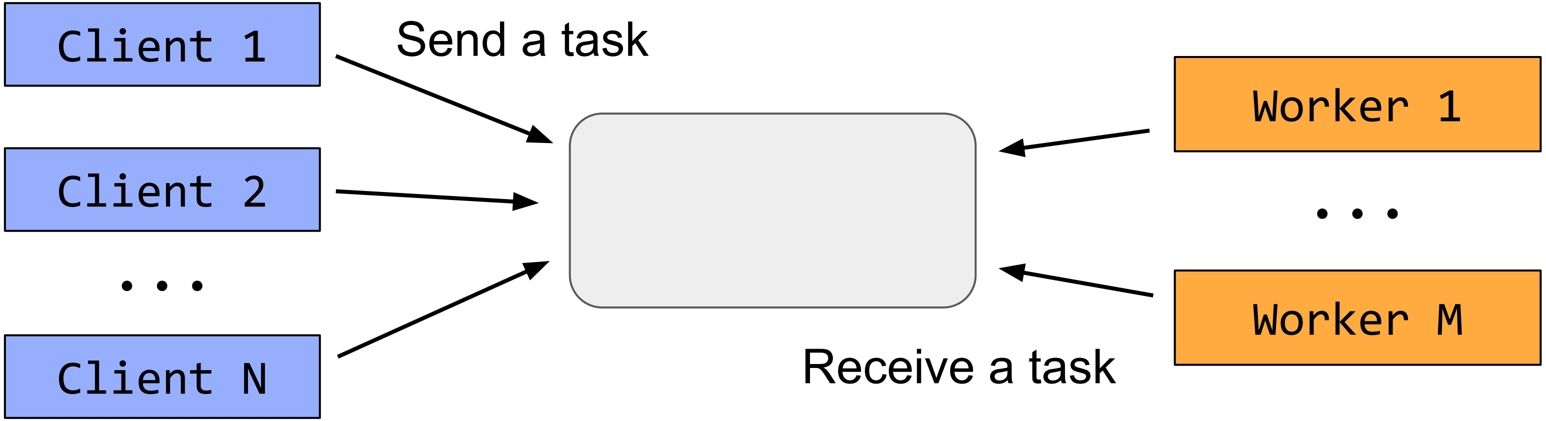
suspend functions



Shared + Mutable =

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

Producer-Consumer Problem



* Both clients and workers 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 an infinite loop

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

Rendezvous Channel Semantics

Client 1

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

Worker

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

Client 2

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

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

Rendezvous Channel Semantics

Client 1

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

Client 2

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

Have to wait for send

Worker

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

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

Rendezvous Channel Semantics

Client 1

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

Client 2

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

Worker

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

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

Rendezvous Channel Semantics

Client 1

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

Worker

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

Client 2

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

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

Rendezvous 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>()
```

Rendezvous 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>()
```

Rendezvous 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>()
```

Rendezvous Channel Semantics

Client 1

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

Client 2

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

Worker

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

`val tasks = Channel<Task>()`

Rendezvous 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>()
```

Coroutines Management

Coroutines Management

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

fun curCoroutine(): Coroutine { ... }

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

Element to be sent

Returns the current coroutine

Functions to manipulate with coroutines

Sequential Rendezvous Channel Implementation

```
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

Sequential Rendezvous Channel Implementation

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

fun curCo

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


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

Check if there is no receiver and suspends

Rendezvous: retrieve 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(r)
    }
}
```

Sequential Rendezvous Channel Implementation

```
suspend fun receive(): T {
    if (senders.isEmpty()) {
        val curCor = curCoroutine()
        receivers.enqueue(curCor)
        suspend(curCor)
        return curCor.element
    } else {
        val s = senders.dequeue()
        val res = s.element
        resume(s)
        return res
    }
}
```

```
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(r)
    }
}
```

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)
   }
}
```

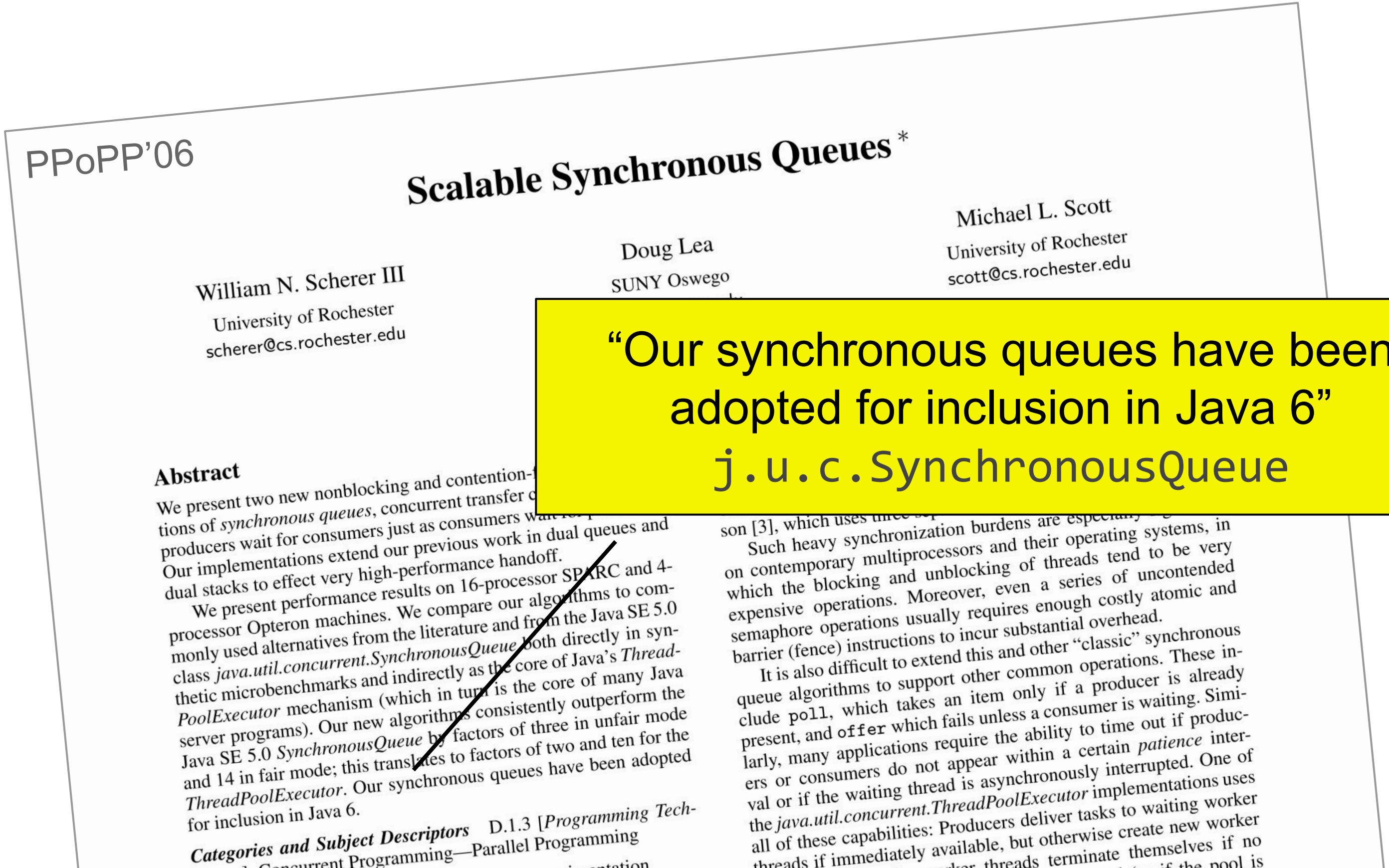
Rendezvous Channel: Golang

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...

Rendezvous Channel: Java



PPoPP'06

Scalable Synchronous Queues *

William N. Scherer III
University of Rochester
scherer@cs.rochester.edu

Doug Lea
SUNY Oswego

Michael L. Scott
University of Rochester
scott@cs.rochester.edu

Abstract

We present two new nonblocking and contention- tions of *synchronous queues*, concurrent transfer producers wait for consumers just as consumers Our implementations extend our previous work in dual queues and dual stacks to effect very high-performance handoff.

We present performance results on 16-processor SPARC and 4-processor Opteron machines. We compare our algorithms to commonly used alternatives from the literature and from the Java SE 5.0 class *java.util.concurrent.SynchronousQueue* both directly in synthetic microbenchmarks and indirectly as the core of Java's *ThreadPoolExecutor* mechanism (which in turn is the core of many Java server programs). Our new algorithms consistently outperform the Java SE 5.0 *SynchronousQueue* by factors of three in unfair mode and 14 in fair mode; this translates to factors of two and ten for the *ThreadPoolExecutor*. Our synchronous queues have been adopted for inclusion in Java 6.

Categories and Subject Descriptors D.1.3 [*Programming Tech-*
Concurrent Programming—Parallel Programming

son [3], which uses

Such heavy synchronization burdens are on contemporary multiprocessors and their operating systems, in which the blocking and unblocking of threads tend to be very expensive operations. Moreover, even a series of uncontended semaphore operations usually requires enough costly atomic and barrier (fence) instructions to incur substantial overhead.

It is also difficult to extend this and other "classic" synchronous queue algorithms to support other common operations. These include `poll`, which takes an item only if a producer is already present, and `offer` which fails unless a consumer is waiting. Similarly, many applications require the ability to time out if producers or consumers do not appear within a certain *patience* interval or if the waiting thread is asynchronously interrupted. One of the *java.util.concurrent.ThreadPoolExecutor* implementations uses all of these capabilities: Producers deliver tasks to waiting worker threads if immediately available, but otherwise create new worker threads terminate themselves if no

"Our synchronous queues have been adopted for inclusion in Java 6"

`j.u.c.SynchronousQueue`

Rendezvous Channel: Java

Based on Michael-Scott lock-free queue algorithm

the simplest known lock-free queue, `j.u.c.ConcurrentLinkedQueue`

Rendezvous Channel: Java

Based on Michael-Scott lock-free queue algorithm

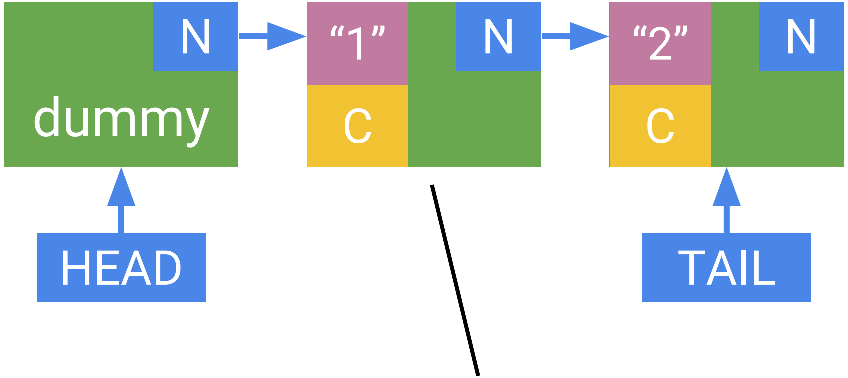
the simplest known lock-free queue, `j.u.c.ConcurrentLinkedQueue`

Either senders or receivers are in the queue!

Rendezvous Channel: Java

Based on Michael-Scott lock-free queue algorithm

the simplest known lock-free queue, `j.u.c.ConcurrentLinkedQueue`

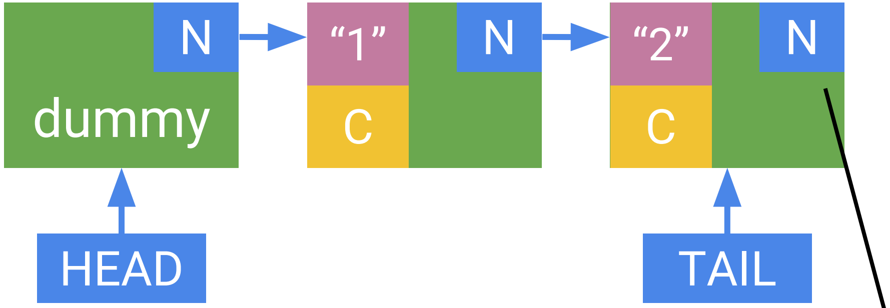


Stores both the element to be sent (`RECEIVE_EL` for `receive`) and the coroutine

Rendezvous Channel: Java

Based on Michael-Scott lock-free queue algorithm

the simplest known lock-free queue, `j.u.c.ConcurrentLinkedQueue`



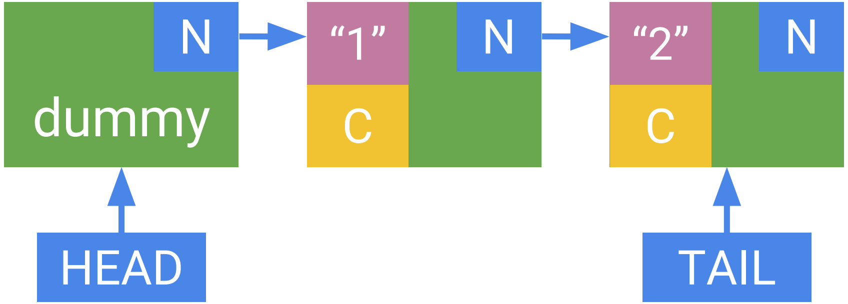
dequeue updates `HEAD`

enqueue updates
`TAIL` and `NEXT`

Rendezvous Channel: Java

Based on Michael-Scott lock-free queue algorithm

the simplest known lock-free queue, `j.u.c.ConcurrentLinkedQueue`



```
send(x):
  t := TAIL
  h := HEAD
  if t == h || t.isSender() {
    enqueueAndSuspend(t, x)
  } else {
    dequeueAndResume(h)
  }
```

Rendezvous Channel: Java

Pros:

- Clear and simple algorithm
- Guarantees lock-freedom for the registration phase

Cons:

- Creates a new node on each suspend
- Cancellation works in *O(N)*
- Non-scalable

Rendezvous Channel: First Solution

Let’s store multiple waiters in node!

Rendezvous Channel: First Solution

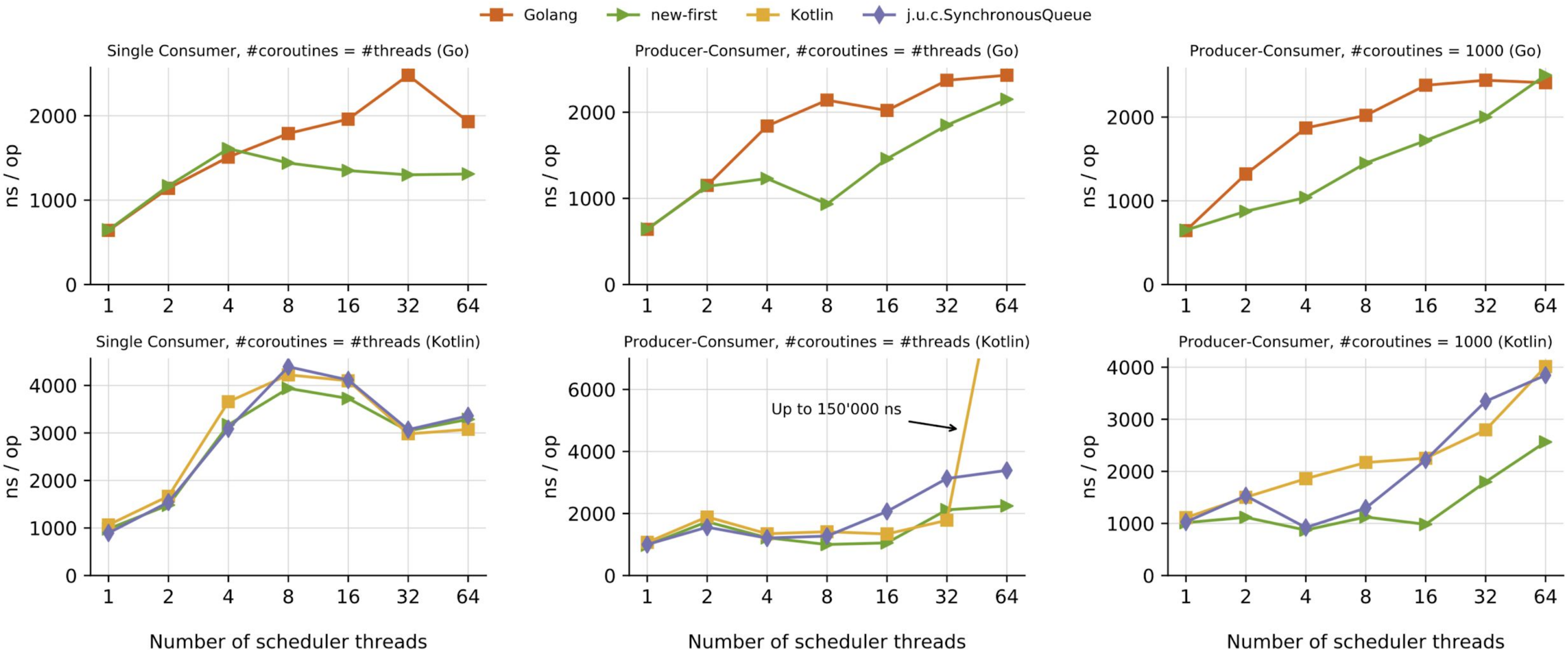
- Each node stores *K* waiters
  - More cache-efficient
  - More GC-efficient

- Node removing works in *O(1)*

- The `select` expression support via descriptors
  - Will be discussed a bit later

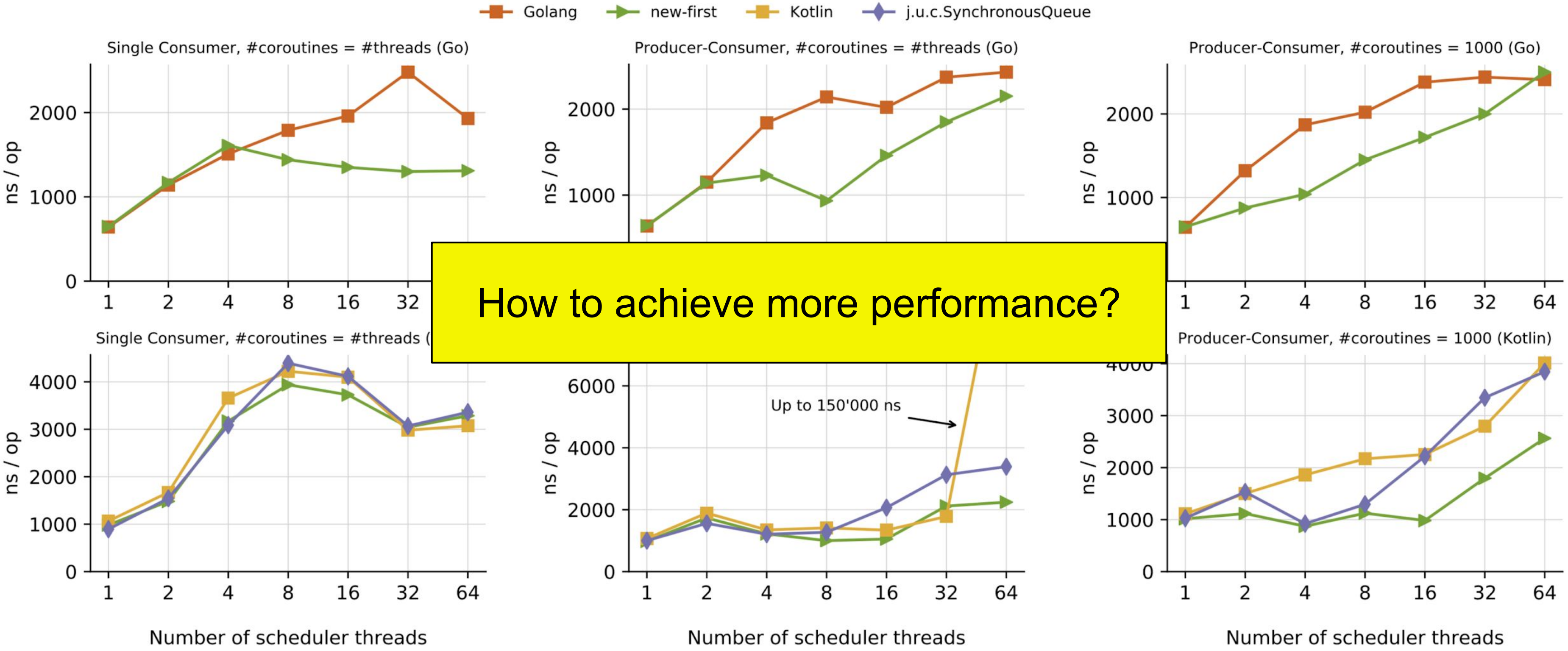
Less is better!

Rendezvous Channel: First Solution



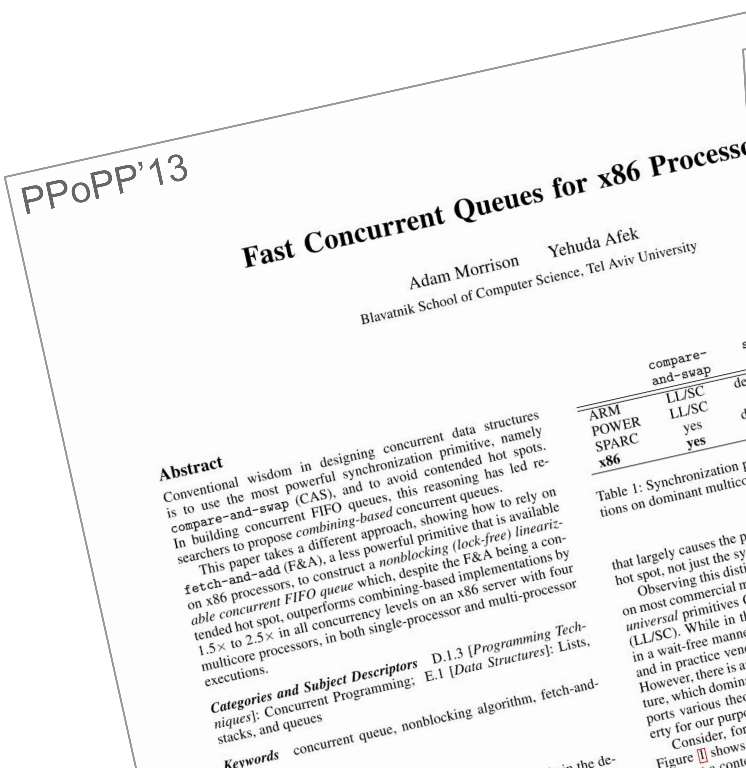
Less is better!

Rendezvous Channel: First Solution



How to achieve more performance?

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



PPoPP'13

Fast Concurrent Queues for x86 Processo

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`
ARM	LL/SC
POWER	LL/SC
SPARC	yes
x86	**yes**

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.

Categories and Subject Descriptors D.1.3 [*Programming Tech-*

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:

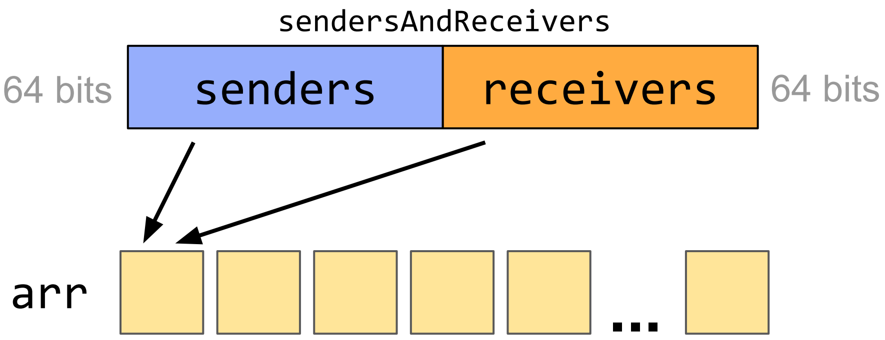
- *obstruction-free* if a thread can perform an arbitrary operation on the object in a *finite* number of steps when it executes in *isolation*,
- *lock-free* if *some* thread performing an arbitrary operation on the object will complete in a *finite* number of steps, or
- *wait-free* if *every* thread 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

Rendezvous Channel: Second Solution

Assume we have an atomic array and an atomic 128-bit register



`sendersAndReceivers`

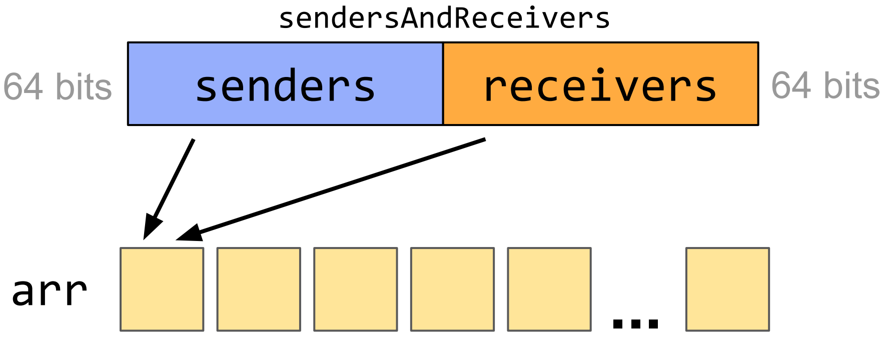
64 bits `senders` `receivers` 64 bits

`arr` …

`senders`   = cell for the next `send`
`receivers` = cell for the next `receive`

Rendezvous Channel: Second Solution

Assume we have an atomic array and an atomic 128-bit register

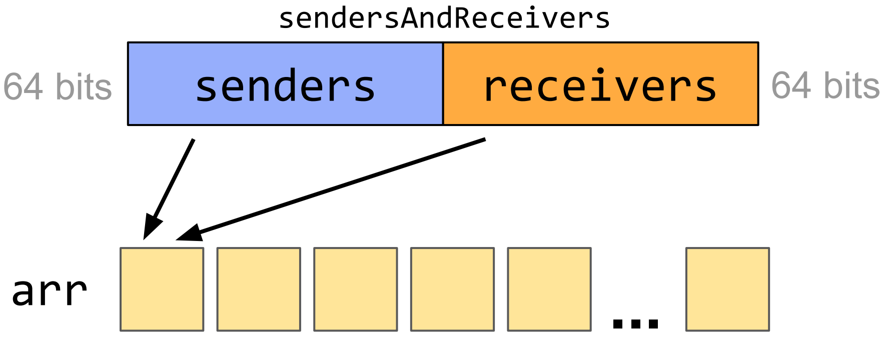


senders   = cell for the next send
receivers = cell for the next receive

```
send(x):
  s, r := incSenders()
  if s >= r {
    arr[s] = Waiter{curCor(), x}
  } else {
    resume(arr[s], x)
  }
```

Rendezvous Channel: Second Solution

Assume we have an atomic array and an atomic 128-bit register



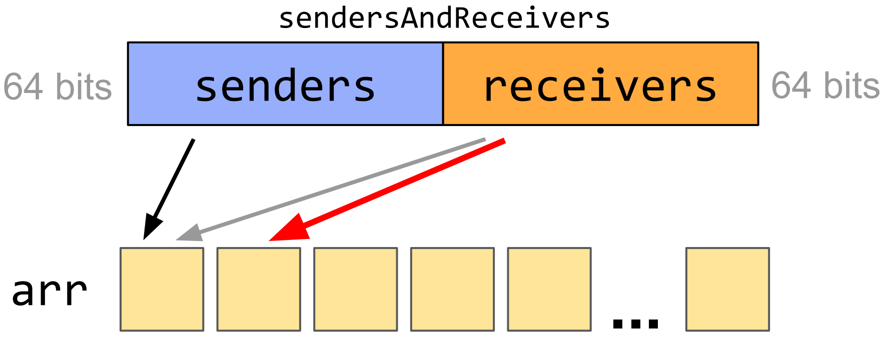
```
send(x):
  s, r := incSenders()
  if s >= r {
    arr[s] = Waiter{curCor(), x}
  } else {
    resume(arr[s], x)
  }
```

`send(1):`

`receive():`

Rendezvous Channel: Second Solution

Assume we have an atomic array and an atomic 128-bit register



```
send(x):
  s, r := incSenders()
  if s >= r {
    arr[s] = Waiter{curCor(), x}
  } else {
    resume(arr[s], x)
  }
```

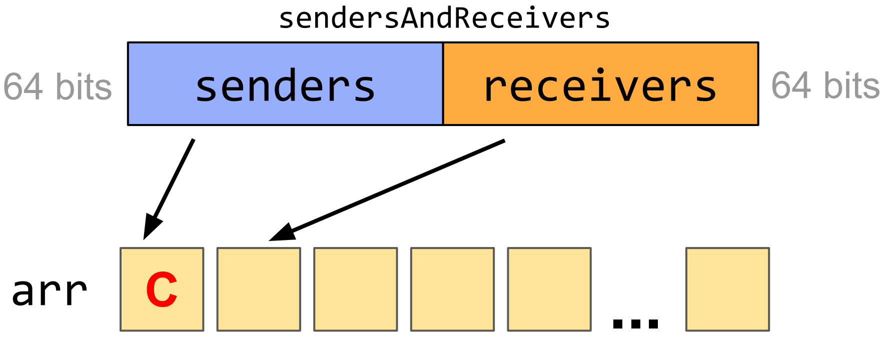
`send(1):`

`receive():`

1. **Inc receivers**

Rendezvous Channel: Second Solution

Assume we have an atomic array and an atomic 128-bit register



```
send(x):
  s, r := incSenders()
  if s >= r {
    arr[s] = Waiter{curCor(), x}
  } else {
    resume(arr[s], x)
  }
```

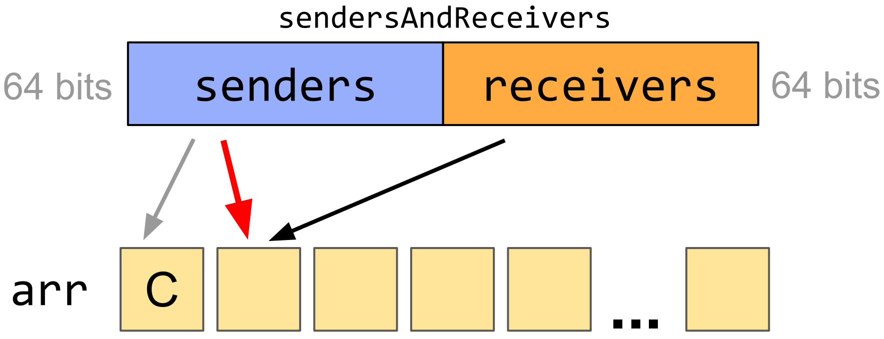
`send(1):`

`receive():`

1. Inc receivers
2. **Store the coroutine**

Rendezvous Channel: Second Solution

Assume we have an atomic array and an atomic 128-bit register



```
send(x):
  s, r := incSenders()
  if s >= r {
    arr[s] = Waiter{curCor(), x}
  } else {
    resume(arr[s], x)
  }
```

`send(1):`

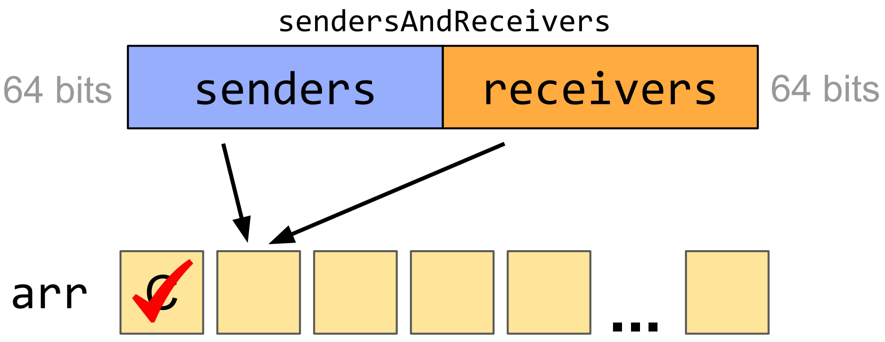
3. **`Inc senders`**

`receive():`

1. `Inc receivers`
2. Store the coroutine

Rendezvous Channel: Second Solution

Assume we have an atomic array and an atomic 128-bit register



```
send(x):
  s, r := incSenders()
  if s >= r {
    arr[s] = Waiter{curCor(), x}
  } else {
    resume(arr[s], x)
  }
```

`send(1):`

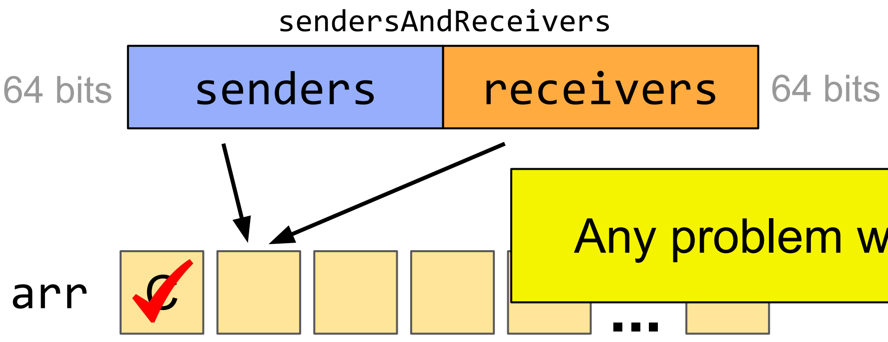
3. Inc `senders`
4. **Make a rendezvous**

`receive():`

1. Inc `receivers`
2. Store the coroutine

Rendezvous Channel: Second Solution

Assume we have an atomic array and an atomic 128-bit register



```
send(x):
  s, r := incSenders()
  if s >= r {
                         Waiter{curCor(), x}

    resume(arr[s], x)
  }
```

Any problem with this solution?

`send(1):`

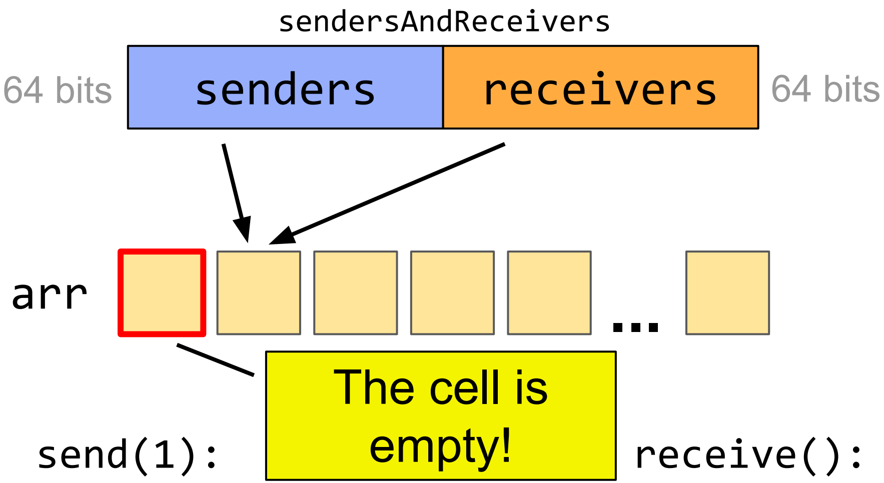
3. Inc `senders`
4. Make a rendezvous

`receive():`

1. Inc `receivers`
2. Store the coroutine

Rendezvous Channel: Second Solution

Assume we have an atomic array and an atomic 128-bit register



```
send(x):
  s, r := incSenders()
  if s >= r {
    arr[s] = Waiter{curCor(), x}
  } else {
    resume(arr[s], x)
  }
```

`send(1):`

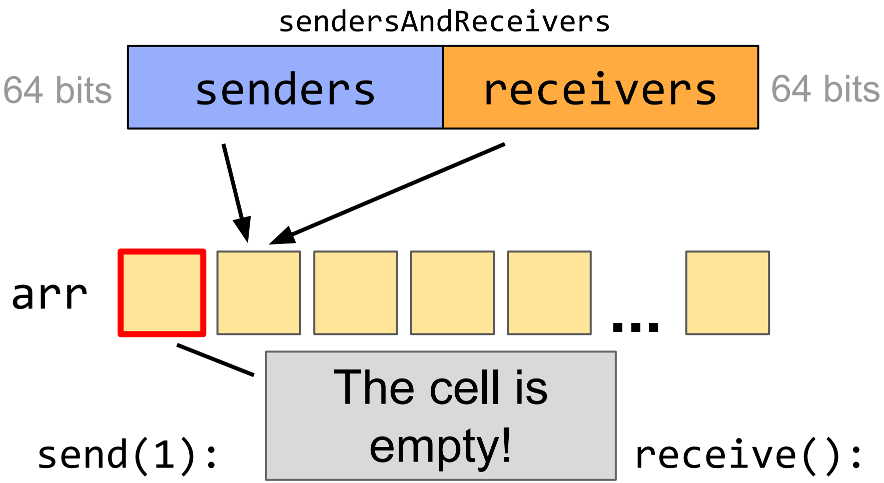
2. Inc `senders`
3. **Make a rendezvous?**

`receive():`

1. Inc `receivers`

Rendezvous Channel: Second Solution

Assume we have an atomic array and an atomic 128-bit register

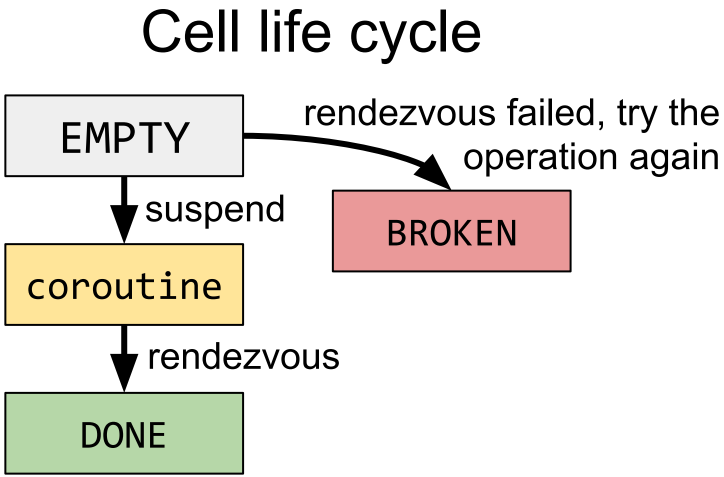


`send(1):`

2. `Inc senders`
3. **Make a rendezvous?**

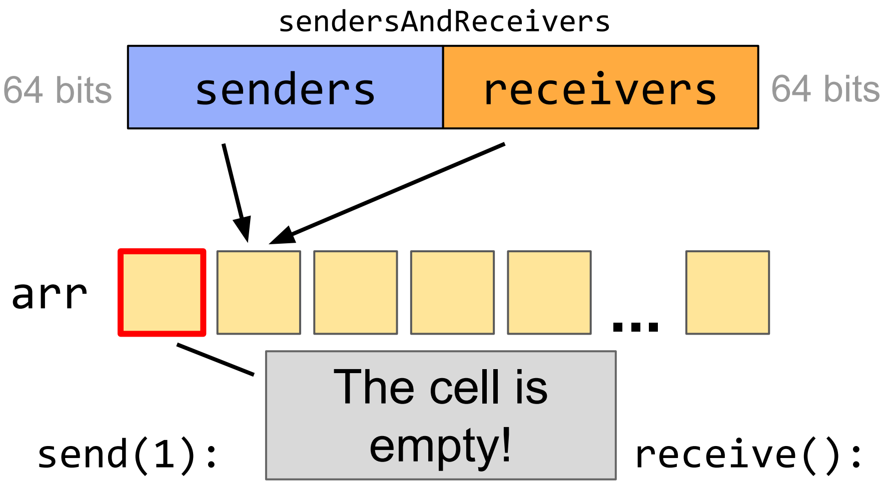
`receive():`

1. `Inc receivers`



Rendezvous Channel: Second Solution

Assume we have an atomic array and an atomic 128-bit register



The cell is empty!

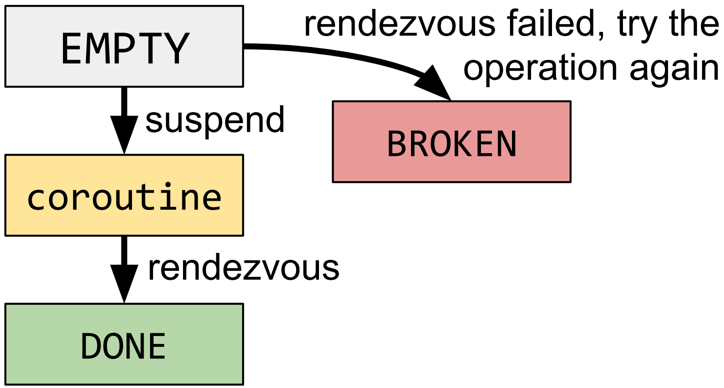
`send(1):`

2. `Inc senders`
3. **Make a rendezvous?**

`receive():`

1. `Inc receivers`

Cell life cycle



Do not need this `BROKEN` state in practice, can just wait

Rendezvous Channel: Second Solution

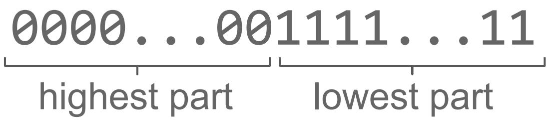
- Each `send-receive` pair works with an unique cell
- This cell id is either `senders` or `receivers` counter after the increment (for `send` and `receive` respectively)

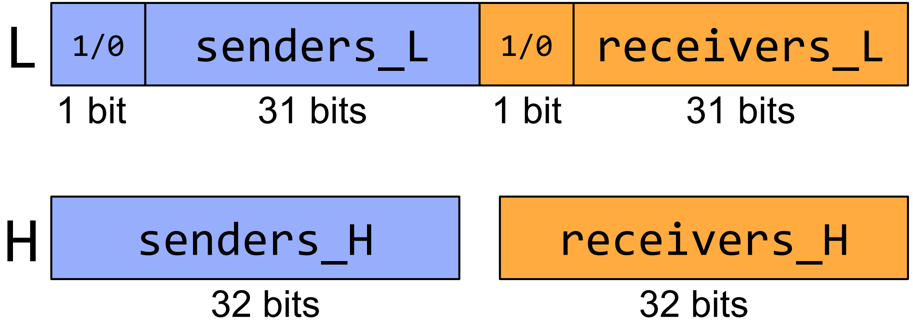
Rendezvous Channel: Second Solution

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

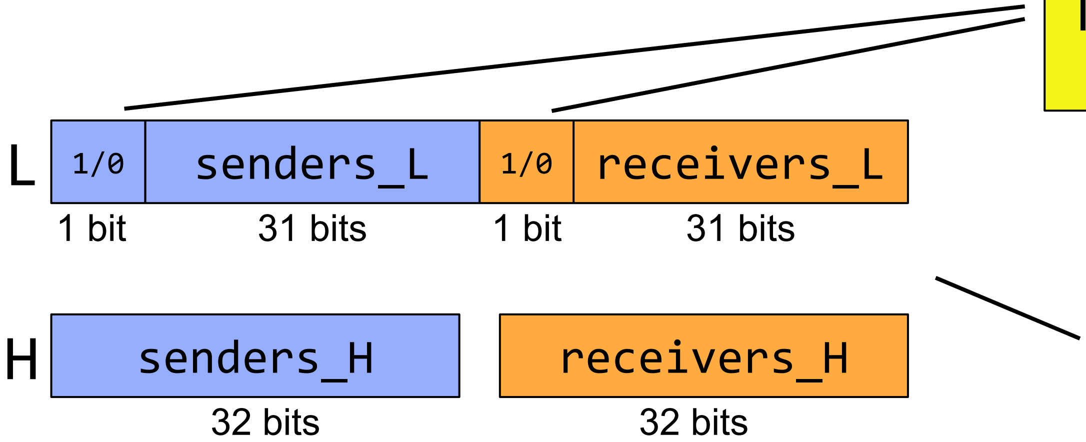
Second Solution: Counters





We maintain highest and lowest parts separately

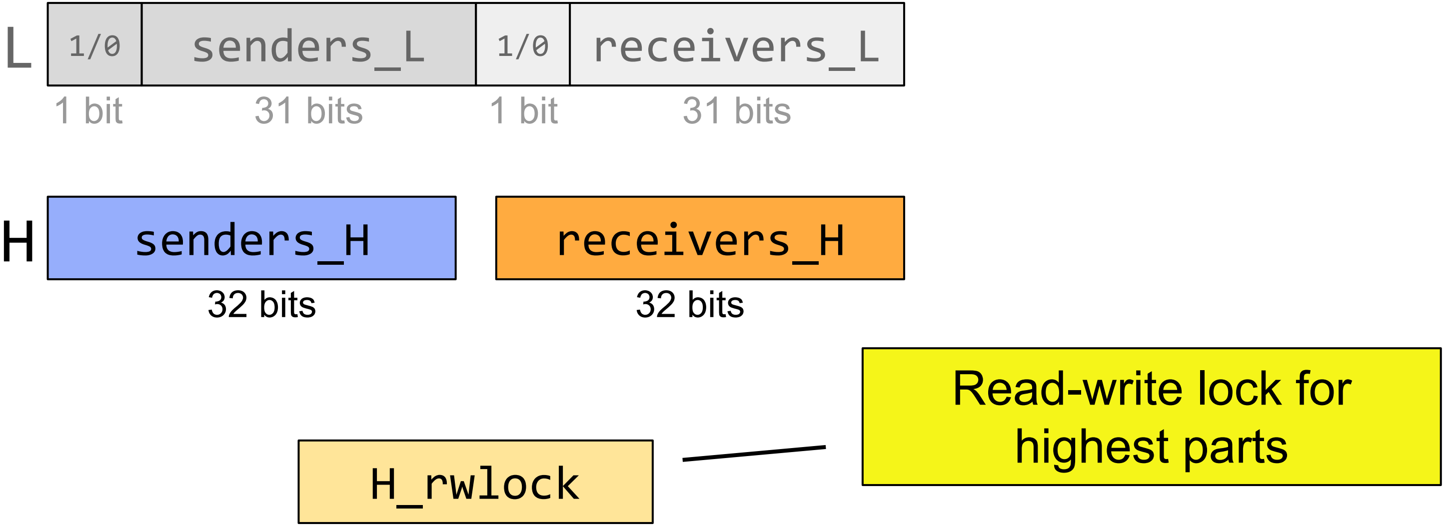
Second Solution: Counters



Indicates that the lowest part is overflowed

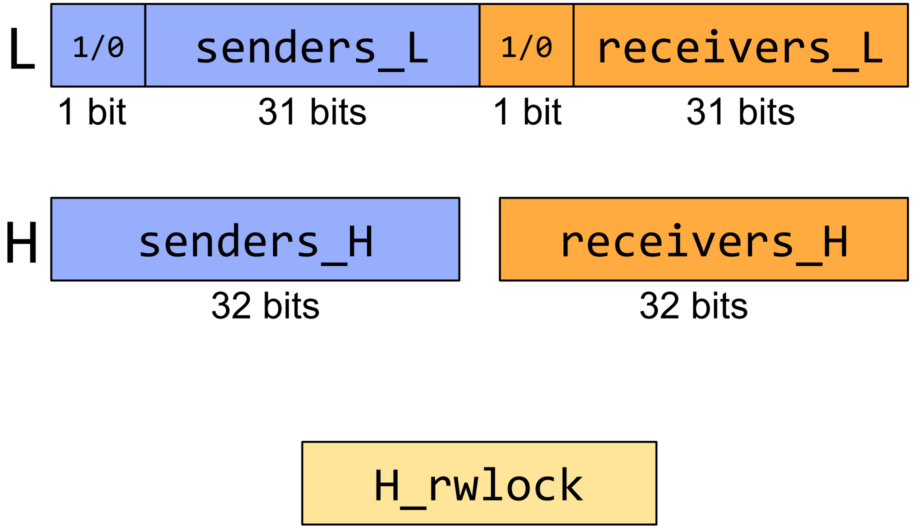
We maintain highest and lowest parts separately

Second Solution: Counters



Read-write lock for highest parts

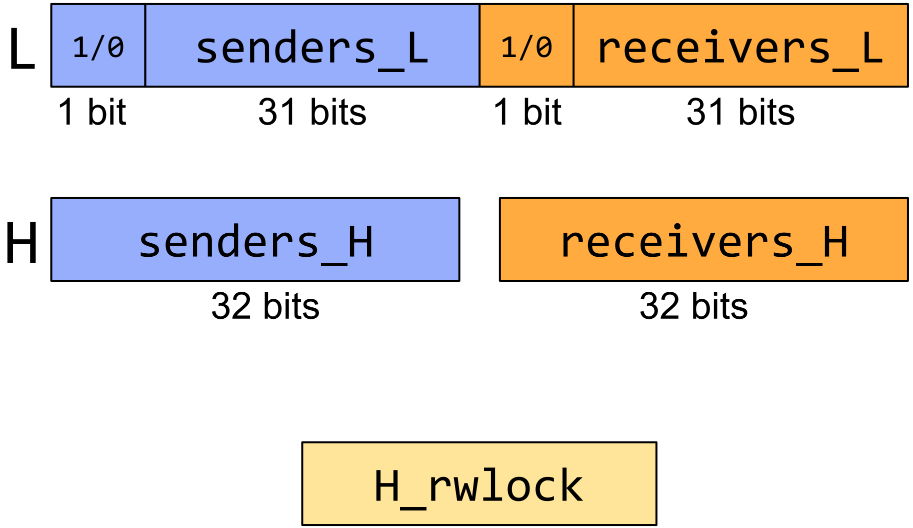
Second Solution: Counters



Increment algorithm:

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

Second Solution: Counters

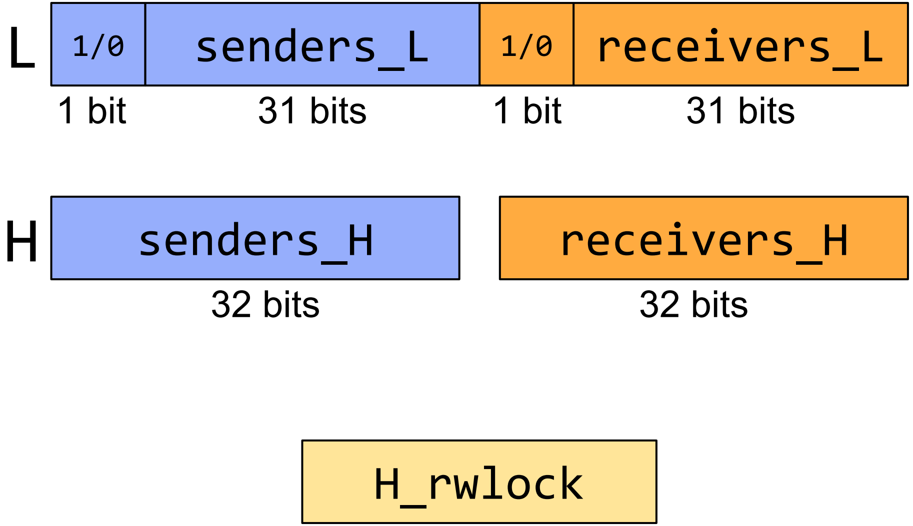


Increment algorithm:

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

Just a FAA

Second Solution: Counters



L `1/0` `senders_L` `1/0` `receivers_L`

1 bit 31 bits 1 bit 31 bits

H `senders_H` `receivers_H`

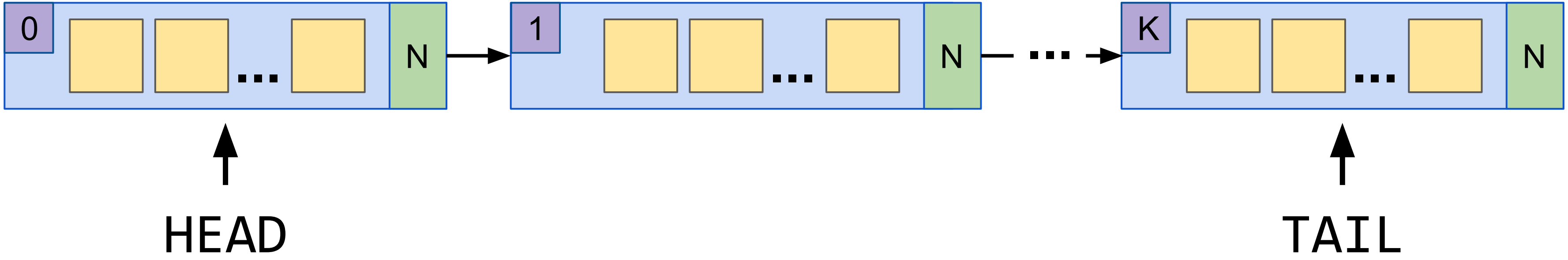
32 bits 32 bits

`H_rwlock`

Increment algorithm:

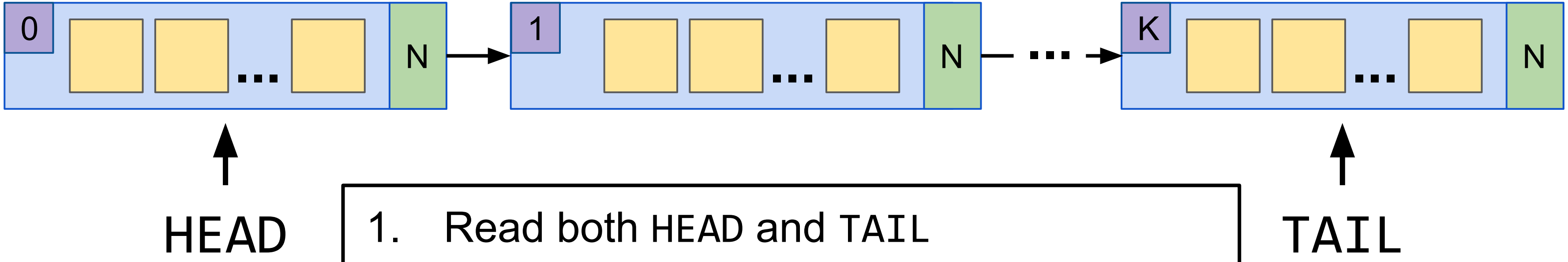
1. Acquire `H_rwlock` for read
2. Read `H`
3. Inc `L` by `FAA`
4. Release the lock
5. If the lowest part is overflowed
    - 5.1. Acquire `H_rwlock` for write
    - 5.2. Reset the bit
    - 5.3. Inc `H`
    - 5.4. Release the lock

Second Solution: Cell Storage



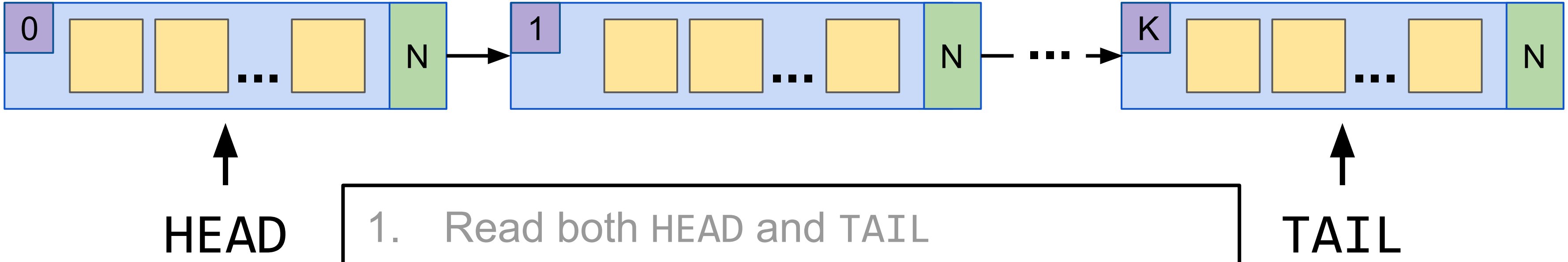
Lock-free Michael-Scott queue of segments

Second Solution: Cell Storage



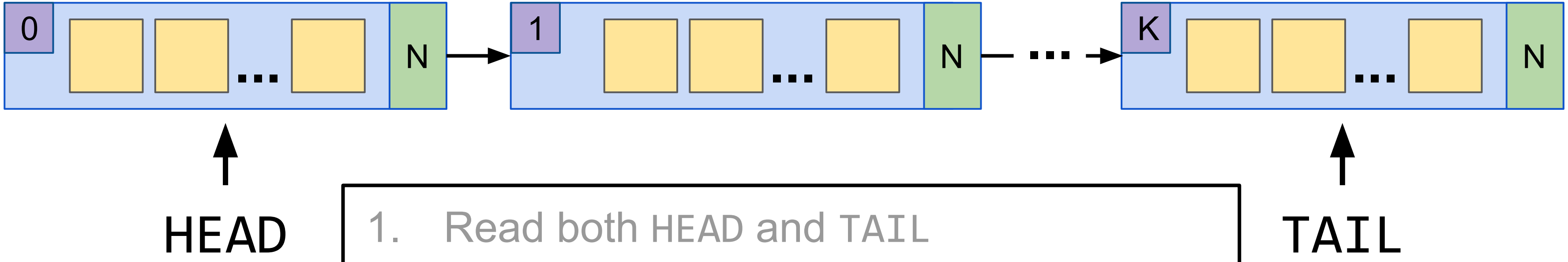
1. Read both `HEAD` and `TAIL`
2. Increment the counter

Second Solution: Cell Storage



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

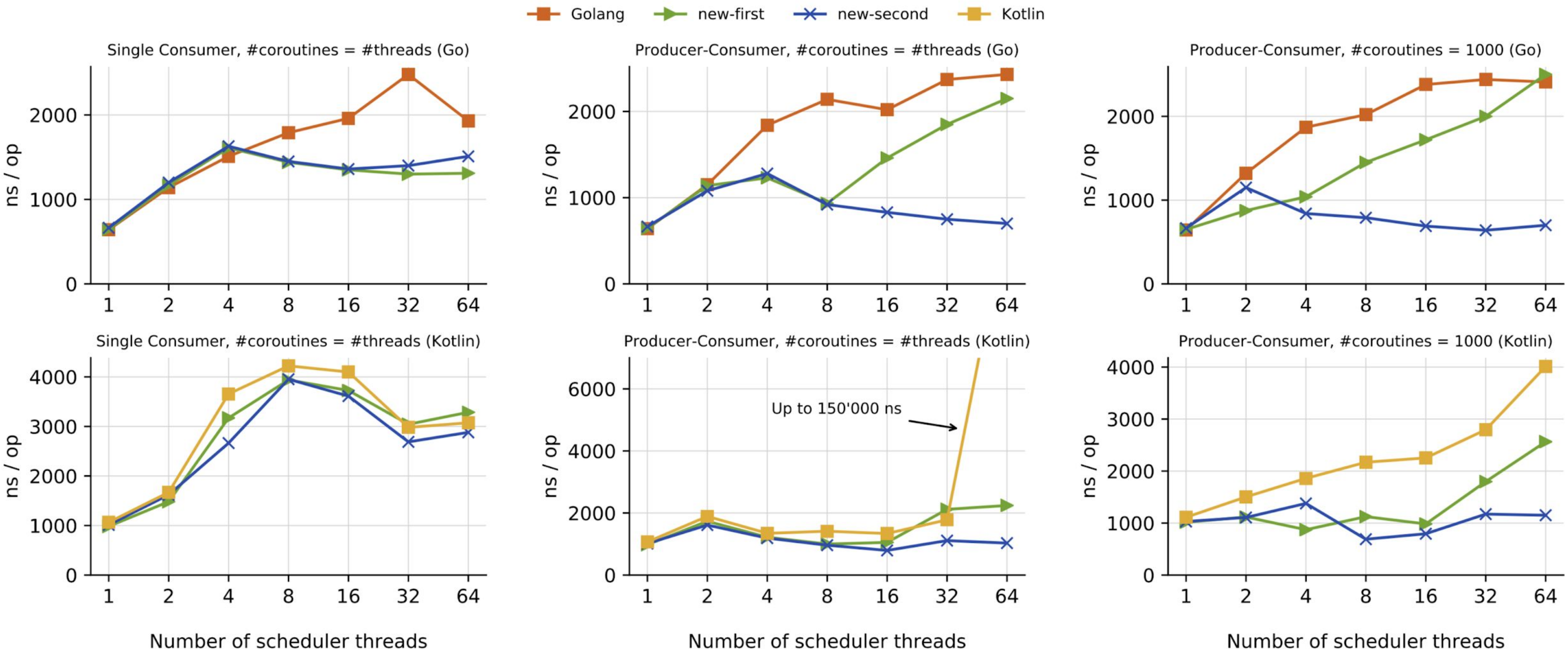
Second Solution: Cell Storage



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

Less is better!

Rendezvous Channel: Second Solution



Buffered Channel Semantics

Client 1

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

Worker

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

Client 2

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

One element can be sent without suspension

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

Buffered Channel Semantics

Client 1

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

Does not suspend!

Worker

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

Client 2

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

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

Buffered Channel Semantics

Client 1

①

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

Worker

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

Client 2

②

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

The buffer is full, suspends

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

Buffered Channel Semantics

Client 1

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

Worker

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

Client 2

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

Receives the buffered element,
resumes the 2nd client,
and moves its task to the buffer

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

Retrieves the 2nd task,
no waiters to resume

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

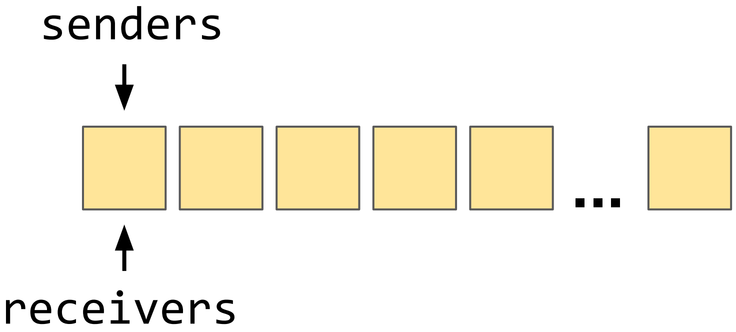
Buffered Channel: Golang

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

- Performs all operations under the channel lock

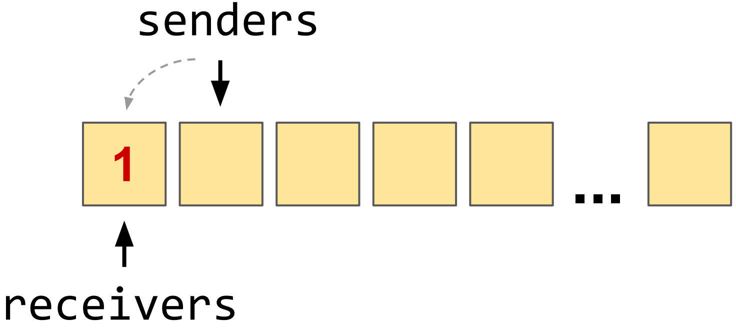
Buffered Channel: Our Solution

Channel with `capacity = 1`



Buffered Channel: Our Solution

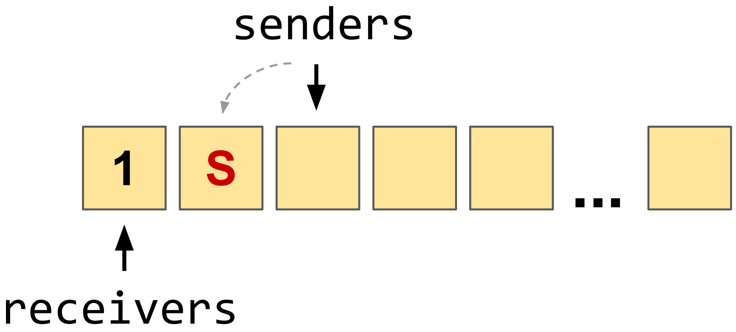
Channel with `capacity = 1`



`send(1): DONE`

Buffered Channel: Our Solution

Channel with `capacity = 1`

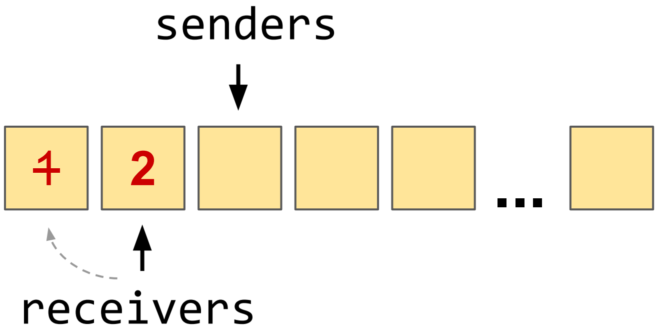


`send(1): DONE`

`send(2): SUSPENDED`

Buffered Channel: Our Solution

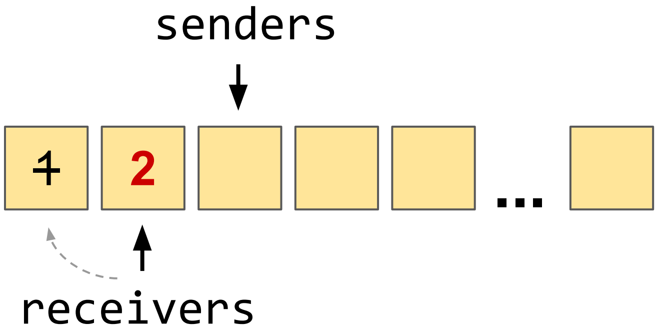
Channel with `capacity = 1`



```
send(1): DONE
send(2): DONE
receive(): 1
```

Buffered Channel: Our Solution

Channel with `capacity = 1`

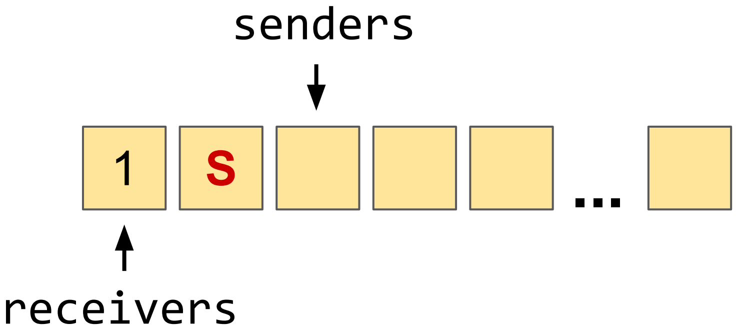


```
send(1): DONE
send(2): DONE
receive(): 1
```

Can we use only `senders` and `receivers` counters to define the current buffer?

Buffered Channel: Our Solution

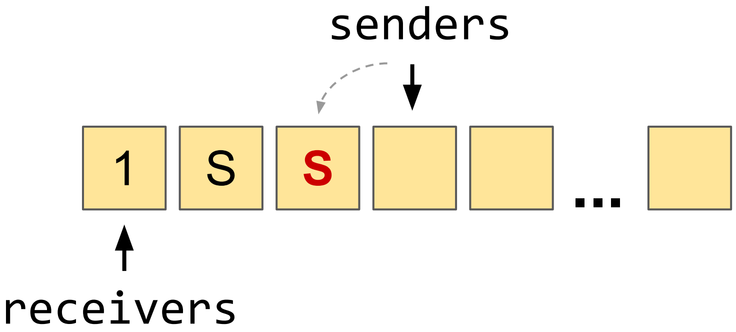
Two counters are not enough!



`send(1): DONE`
`send(2): SUSPENDED`

Buffered Channel: Our Solution

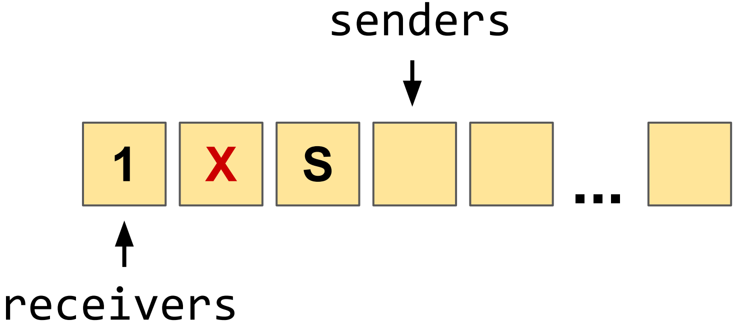
Two counters are not enough!



`send(1): DONE`
`send(2): SUSPENDED`
`send(3): SUSPENDED`

Buffered Channel: Our Solution

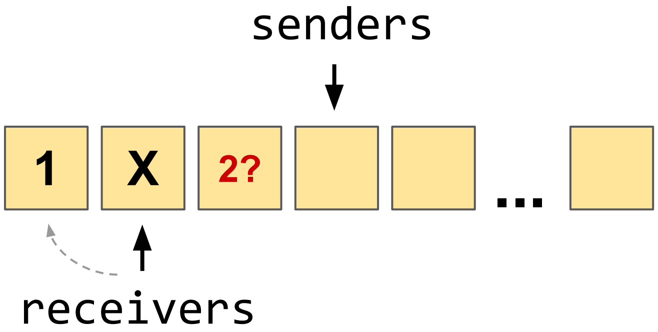
Two counters are not enough!



```
send(1): DONE
send(2): CANCELLED
send(3): SUSPENDED
```

Buffered Channel: Our Solution

Two counters are not enough!

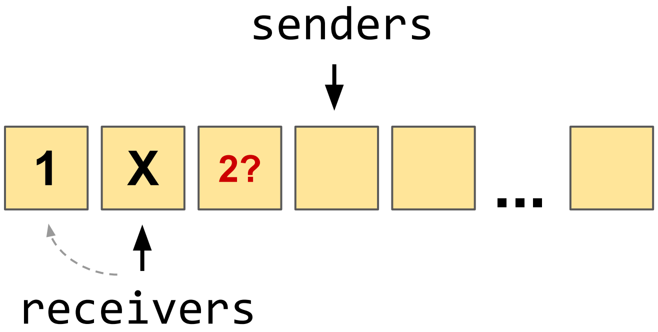


```
send(1): DONE
send(2): CANCELLED
send(3): DONE???
receive(): 1
```

We have to find the first non-cancelled `send` request to resume (put into the buffer)

Buffered Channel: Our Solution

Two counters are not enough!



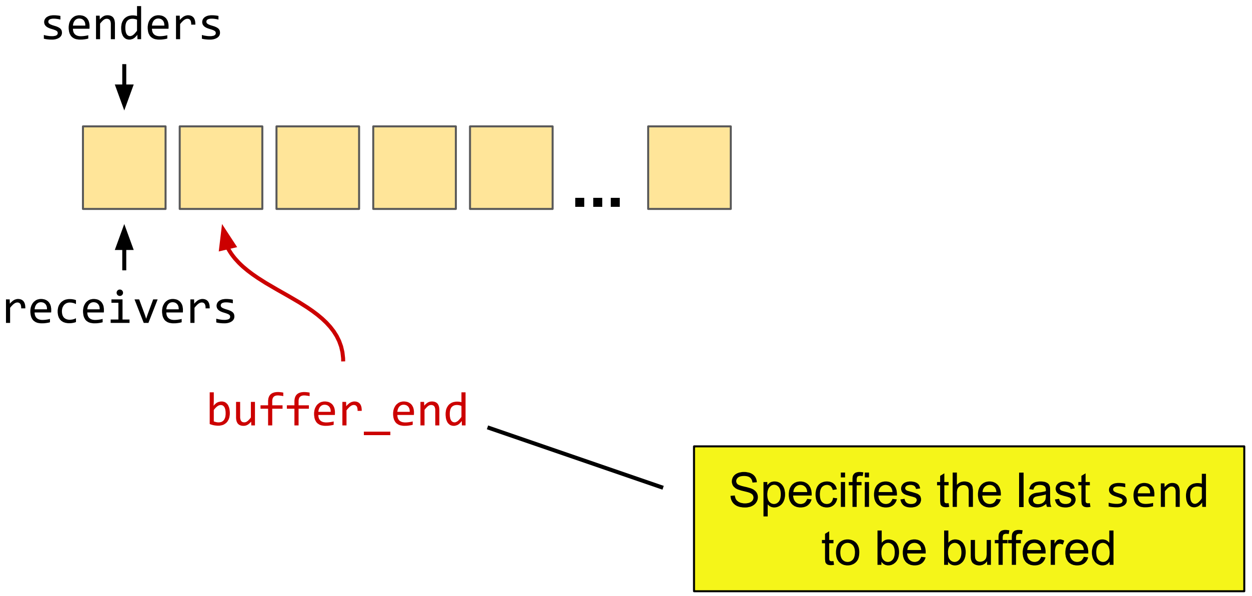
```
send(1): DONE
send(2): CANCELLED
send(3): DONE???
receive(): 1
```

Works in *O(N)*

We have to find the first non-cancelled `send` request to resume (put into the buffer)

Buffered Channel: Our Solution

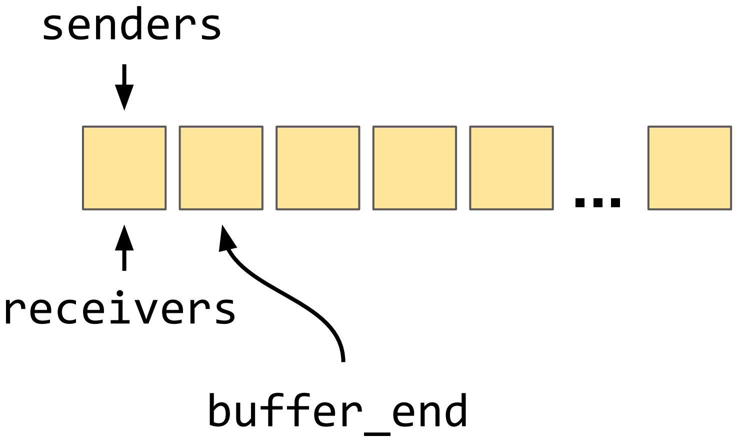
Let’s use three counters!



Specifies the last send to be buffered

Buffered Channel: Our Solution

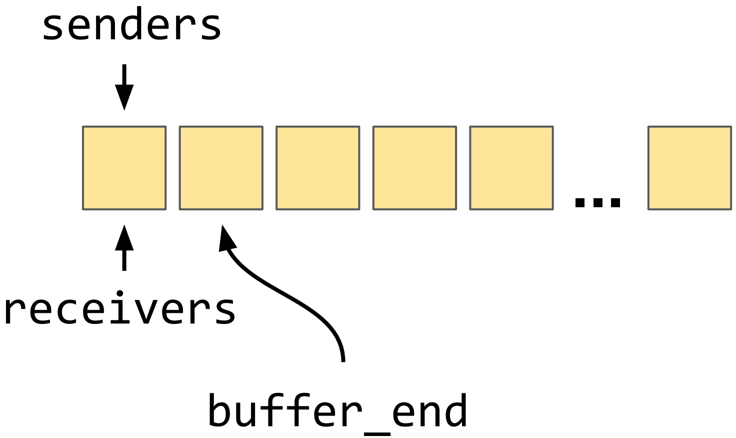
Let’s use three counters!



```
send(x):
  senders++, receivers, buffer_end
  if senders >= receivers {
    if senders < buffer_end {
      storeElement(senders, x) // buffering!
    } else { /* suspend */ }
  } else  { /* rendezvous */ }
```

Buffered Channel: Our Solution

Let’s use three counters!



senders

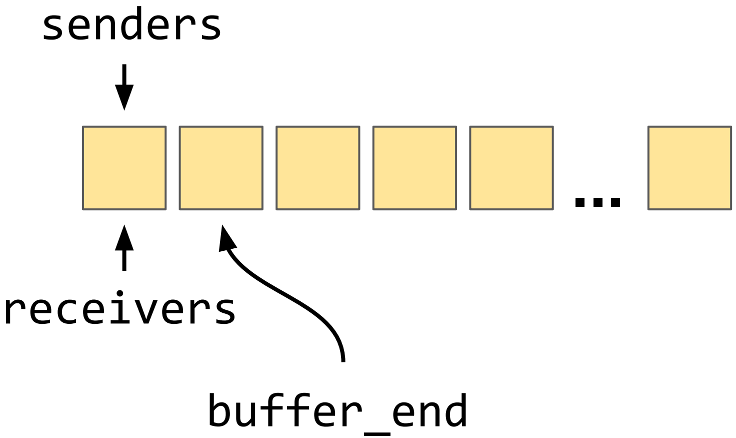
receivers

buffer_end

```
send(x):
  senders++, receivers, buffer_end
  if senders >= receivers {
    if senders < buffer_end {
      storeElement(senders, x) // buffering!
    } else { /* suspend */ }
  } else  { /* rendezvous */ }

receive():
  senders, receivers++, buffer_end++
  receiveImpl(senders, receivers)
  makeBuffered(buffer_end) // inc buffer_end
                           // again on failure
```

Buffered Channel: Our Solution

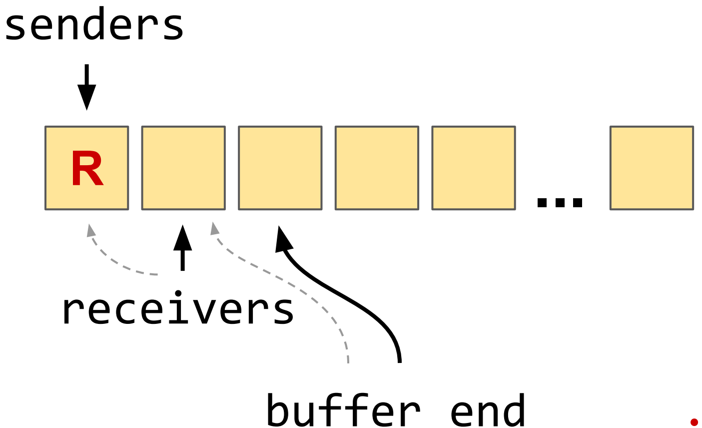
Let's use three counters!



```
send(x):
  senders++, receivers, buffer_end
  if senders >= receivers {
    if senders < buffer_end {
      storeElement(senders, x) // buffering!
    } else { /* suspend */ }
  } else  { /* rendezvous */ }

receive():
  senders, receivers++, buffer_end++
  receiveImpl(senders, receivers)
  makeBuffered(buffer_end) // inc buffer_end
                           // again on failure
```

Buffered Channel: Our Solution

Let’s use three counters!

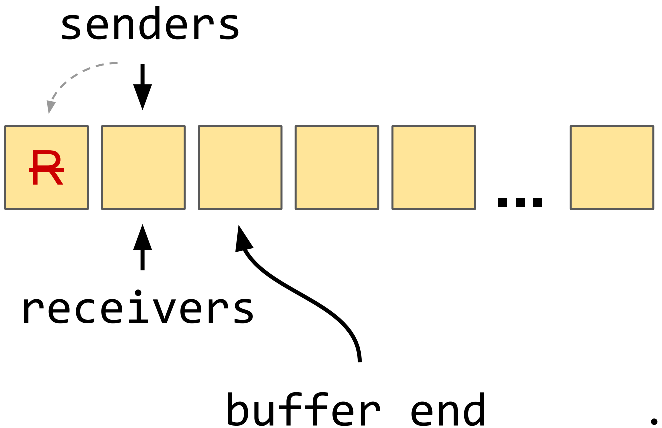


```
send(x):
  senders++, receivers, buffer_end
  if senders >= receivers {
    if senders < buffer_end {
      storeElement(senders, x) // buffering!
    } else { /* suspend */ }
  } else  { /* rendezvous */ }

receive():
  senders, receivers++, buffer_end++
  receiveImpl(senders, receivers)
  makeBuffered(buffer_end) // inc buffer_end
                           // again on failure
```

receive(): SUSPENDED

Buffered Channel: Our Solution

Let’s use three counters!



senders

~~R~~ ...

receivers

buffer_end

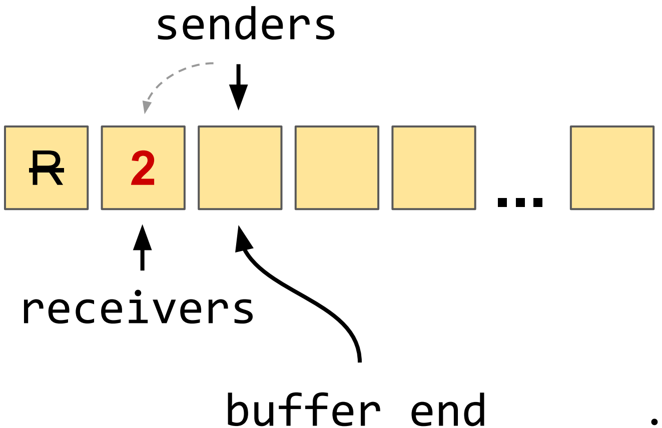
```
send(x):
  senders++, receivers, buffer_end
  if senders >= receivers {
    if senders < buffer_end {
      storeElement(senders, x) // buffering!
    } else { /* suspend */ }
  } else  { /* rendezvous */ }

receive():
  senders, receivers++, buffer_end++
  receiveImpl(senders, receivers)
  makeBuffered(buffer_end) // inc buffer_end
                           // again on failure
```

receive(): 1

send(1): DONE

Buffered Channel: Our Solution

Let’s use three counters!

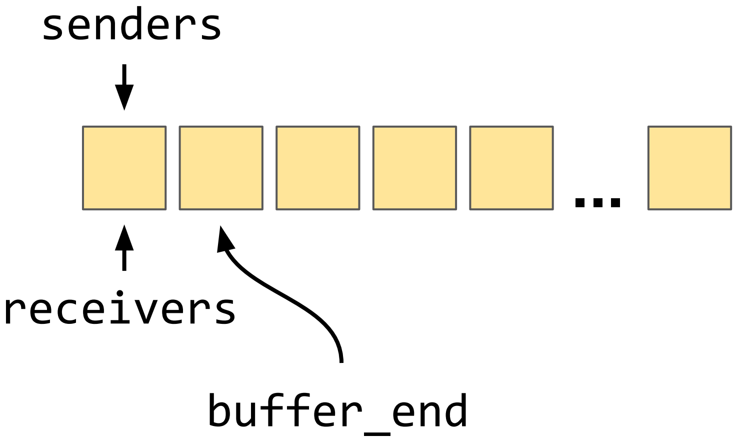


```
send(x):
  senders++, receivers, buffer_end
  if senders >= receivers {
    if senders < buffer_end {
      storeElement(senders, x) // buffering!
    } else { /* suspend */ }
  } else  { /* rendezvous */ }

receive():
  senders, receivers++, buffer_end++
  receiveImpl(senders, receivers)
  makeBuffered(buffer_end) // inc buffer_end
                           // again on failure
```

```
receive(): 1
send(1): DONE
send(2): DONE
```

Buffered Channel: Our Solution

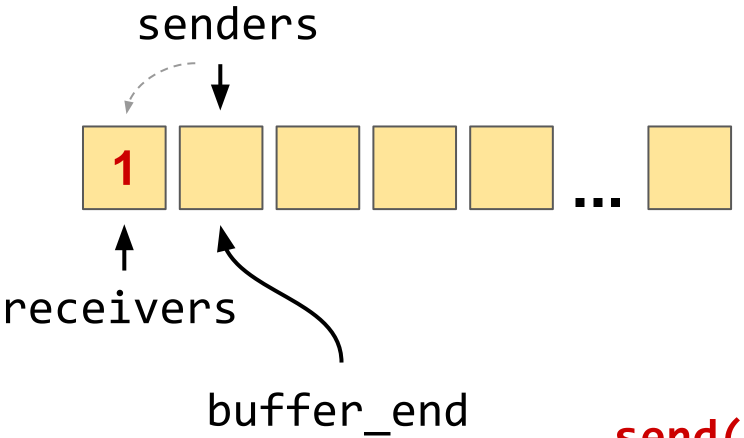
Let’s use three counters!



```
send(x):
  senders++, receivers, buffer_end
  if senders >= receivers {
    if senders < buffer_end {
      storeElement(senders, x) // buffering!
    } else { /* suspend */ }
  } else  { /* rendezvous */ }

receive():
  senders, receivers++, buffer_end++
  receiveImpl(senders, receivers)
  makeBuffered(buffer_end) // inc buffer_end
                           // again on failure
```

Buffered Channel: Our Solution

Let’s use three counters!

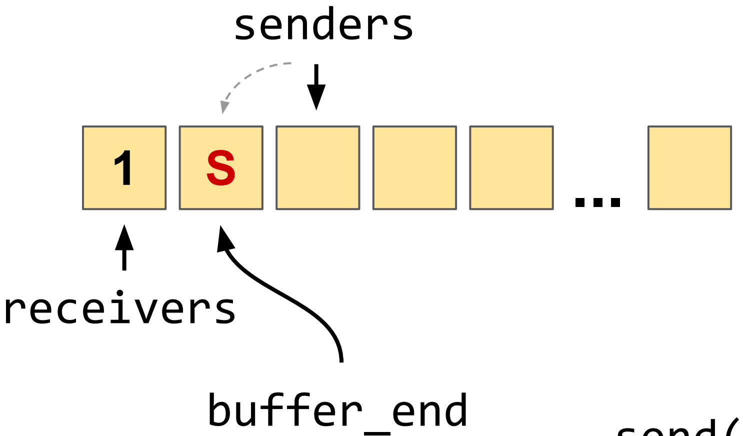


send(1): DONE

```
send(x):
  senders++, receivers, buffer_end
  if senders >= receivers {
    if senders < buffer_end {
      storeElement(senders, x) // buffering!
    } else { /* suspend */ }
  } else  { /* rendezvous */ }

receive():
  senders, receivers++, buffer_end++
  receiveImpl(senders, receivers)
  makeBuffered(buffer_end) // inc buffer_end
                           // again on failure
```

Buffered Channel: Our Solution

Let's use three counters!

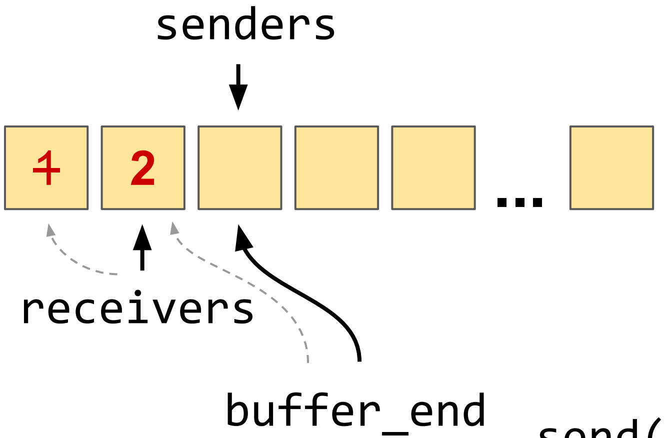


```
send(x):
  senders++, receivers, buffer_end
  if senders >= receivers {
    if senders < buffer_end {
      storeElement(senders, x) // buffering!
    } else { /* suspend */ }
  } else  { /* rendezvous */ }

receive():
  senders, receivers++, buffer_end++
  receiveImpl(senders, receivers)
  makeBuffered(buffer_end) // inc buffer_end
                           // again on failure
```

```
send(1): DONE
send(2): SUSPEND
```

Buffered Channel: Our Solution

Let’s use three counters!

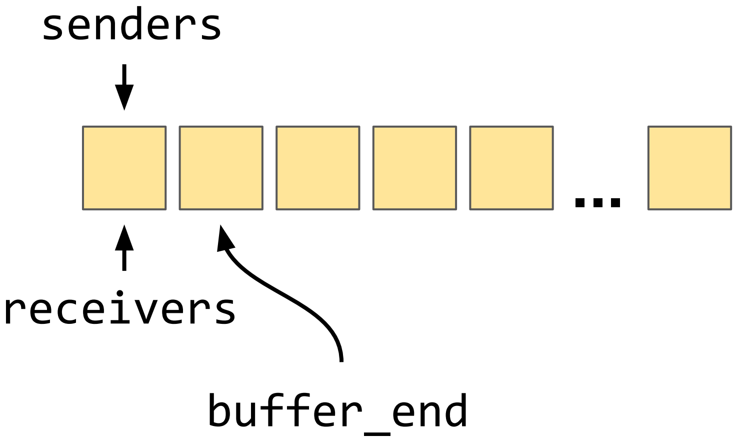


```
send(x):
  senders++, receivers, buffer_end
  if senders >= receivers {
    if senders < buffer_end {
      storeElement(senders, x) // buffering!
    } else { /* suspend */ }
  } else  { /* rendezvous */ }

receive():
  senders, receivers++, buffer_end++
  receiveImpl(senders, receivers)
  makeBuffered(buffer_end) // inc buffer_end
                           // again on failure
```

```
send(1): DONE
send(2): DONE
receive(): 1
```

Buffered Channel: Our Solution

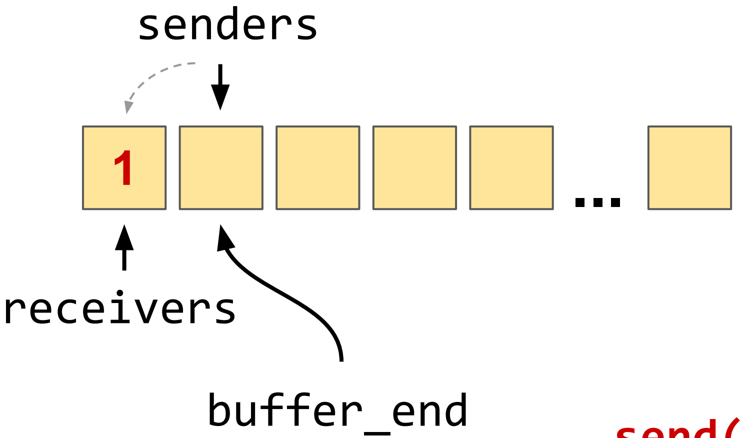
Let’s use three counters!



```
send(x):
  senders++, receivers, buffer_end
  if senders >= receivers {
    if senders < buffer_end {
      storeElement(senders, x) // buffering!
    } else { /* suspend */ }
  } else  { /* rendezvous */ }

receive():
  senders, receivers++, buffer_end++
  receiveImpl(senders, receivers)
  makeBuffered(buffer_end) // inc buffer_end
                           // again on failure
```

Buffered Channel: Our Solution

Let’s use three counters!

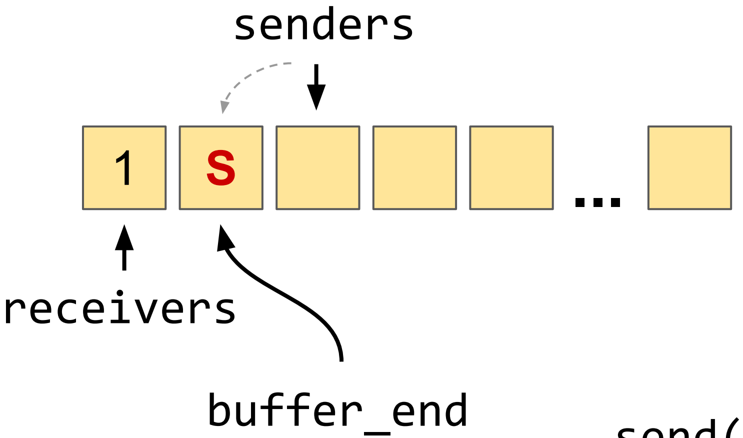


```
send(x):
  senders++, receivers, buffer_end
  if senders >= receivers {
    if senders < buffer_end {
      storeElement(senders, x) // buffering!
    } else { /* suspend */ }
  } else  { /* rendezvous */ }

receive():
  senders, receivers++, buffer_end++
  receiveImpl(senders, receivers)
  makeBuffered(buffer_end) // inc buffer_end
                           // again on failure
```

send(1): DONE

Buffered Channel: Our Solution

Let's use three counters!

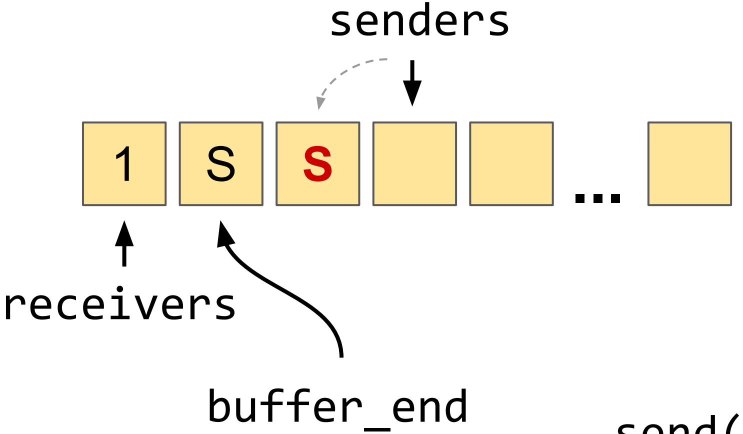


```
send(x):
  senders++, receivers, buffer_end
  if senders >= receivers {
    if senders < buffer_end {
      storeElement(senders, x) // buffering!
    } else { /* suspend */ }
  } else  { /* rendezvous */ }

receive():
  senders, receivers++, buffer_end++
  receiveImpl(senders, receivers)
  makeBuffered(buffer_end) // inc buffer_end
                           // again on failure
```

```
send(1): DONE
send(2): SUSPEND
```

Buffered Channel: Our Solution

Let’s use three counters!

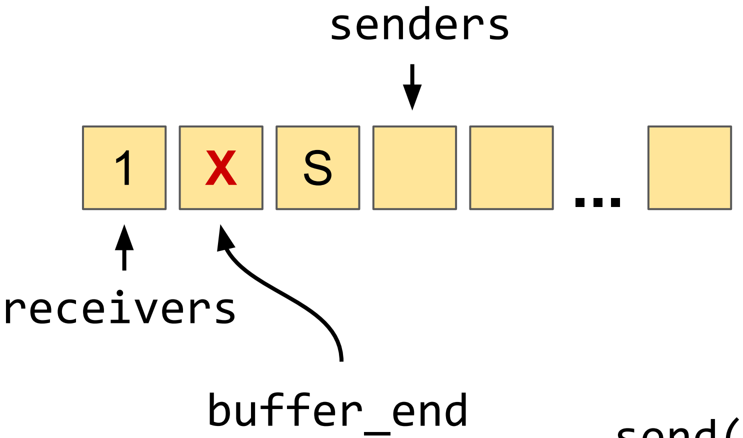


```
send(x):
  senders++, receivers, buffer_end
  if senders >= receivers {
    if senders < buffer_end {
      storeElement(senders, x) // buffering!
    } else { /* suspend */ }
  } else  { /* rendezvous */ }

receive():
  senders, receivers++, buffer_end++
  receiveImpl(senders, receivers)
  makeBuffered(buffer_end) // inc buffer_end
                           // again on failure
```

```
send(1): DONE
send(2): SUSPEND
send(3): SUSPEND
```

Buffered Channel: Our Solution

Let’s use three counters!

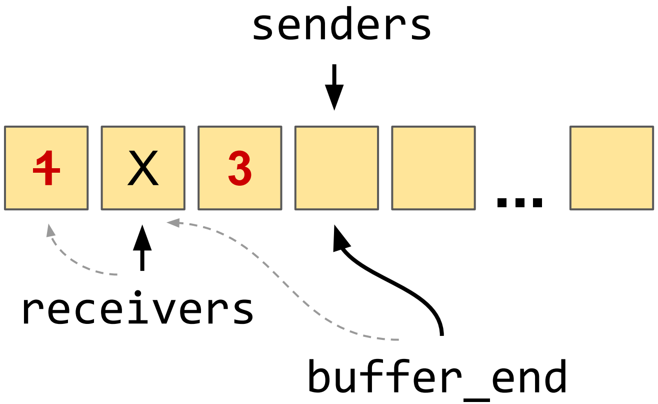


```
send(x):
  senders++, receivers, buffer_end
  if senders >= receivers {
    if senders < buffer_end {
      storeElement(senders, x) // buffering!
    } else { /* suspend */ }
  } else  { /* rendezvous */ }

receive():
  senders, receivers++, buffer_end++
  receiveImpl(senders, receivers)
  makeBuffered(buffer_end) // inc buffer_end
                           // again on failure
```

```
send(1): DONE
send(2): CANCELLED
send(3): SUSPEND
```

Buffered Channel: Our Solution

Let’s use three counters!


senders

~~1~~ X 3 ...

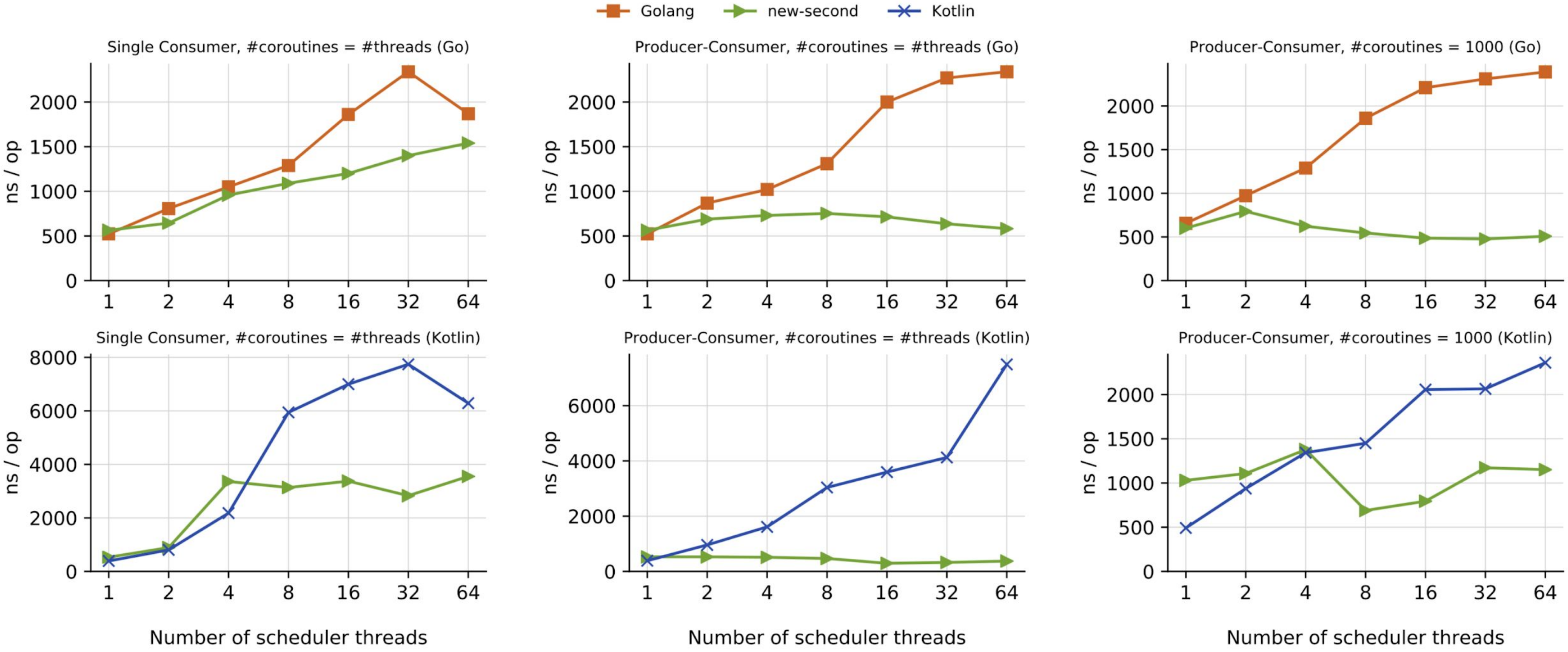
receivers

buffer_end

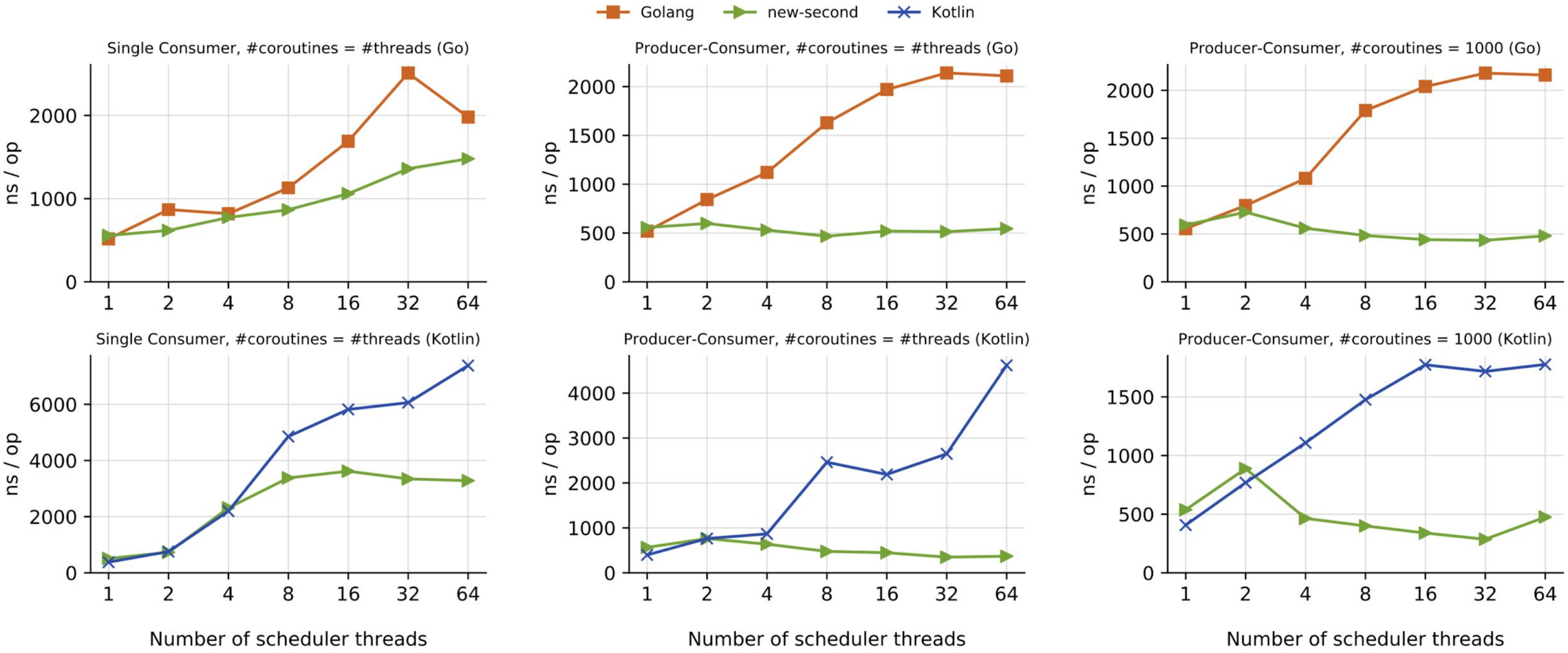
```
send(x):
  senders++, receivers, buffer_end
  if senders >= receivers {
    if senders < buffer_end {
      storeElement(senders, x) // buffering!
    } else { /* suspend */ }
  } else  { /* rendezvous */ }

receive():
  senders, receivers++, buffer_end++
  receiveImpl(senders, receivers)
  makeBuffered(buffer_end) // inc buffer_end
                           // again on failure
```

send(1): DONE
send(2): CANCELLED
send(3): **DONE**
receive(): 1

Buffered Channel: Our Solution (capacity = 32)



Buffered Channel: Our Solution (capacity = 128)



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

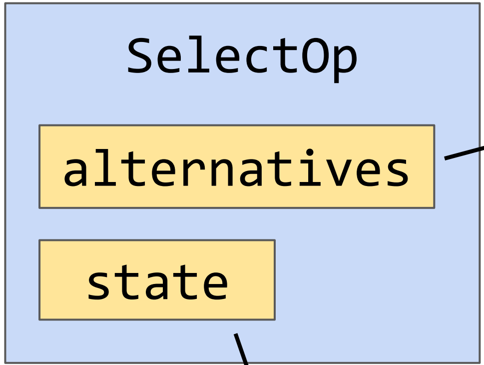
- Acquires all involved channels locks to register into the queues
    - Uses hierarchical order to avoid deadlocks

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

The First Solution

Operation
Descriptors

The `select` Expression: Second Solution



`SelectOp`

`alternatives`

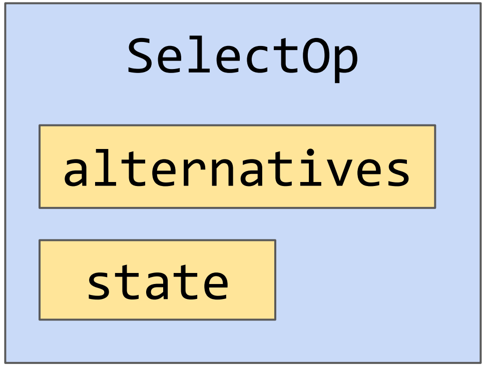
`state`

Each alternative contains:

- element to be sent (`RECEIVE_EL` for `receive`)
- channel
- action

Progress state of this `select` instance

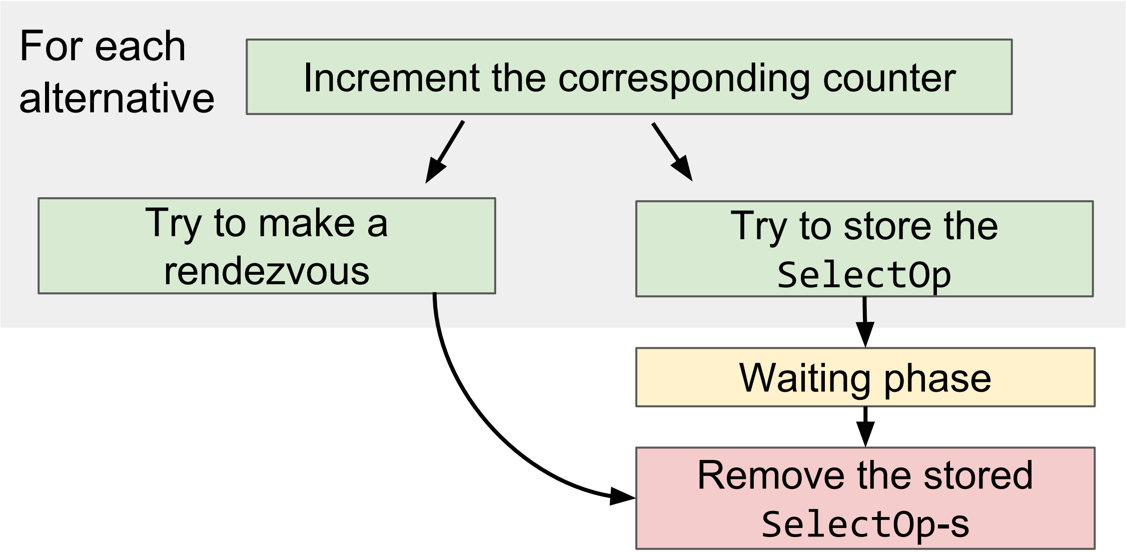
The `select` Expression: Second Solution



`SelectOp`

`alternatives`

`state`



For each alternative

Increment the corresponding counter

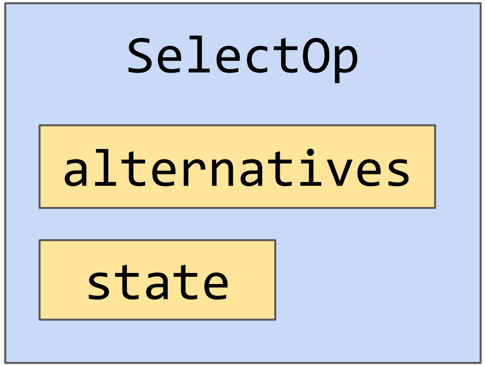
Try to make a rendezvous

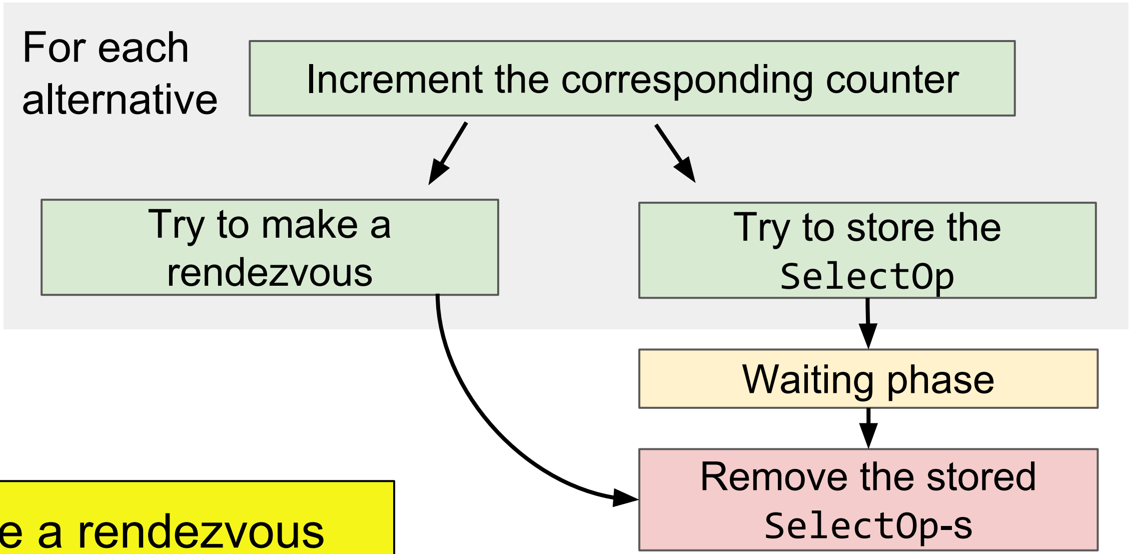
Try to store the `SelectOp`

Waiting phase

Remove the stored `SelectOp`-s

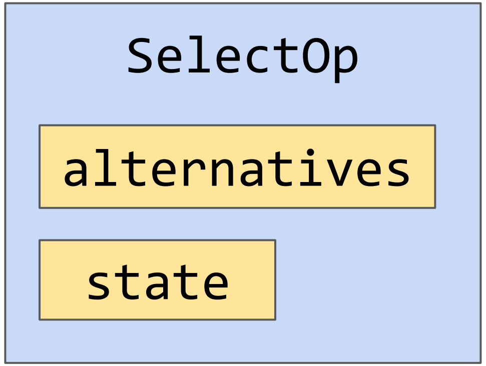
The `select` Expression: Second Solution





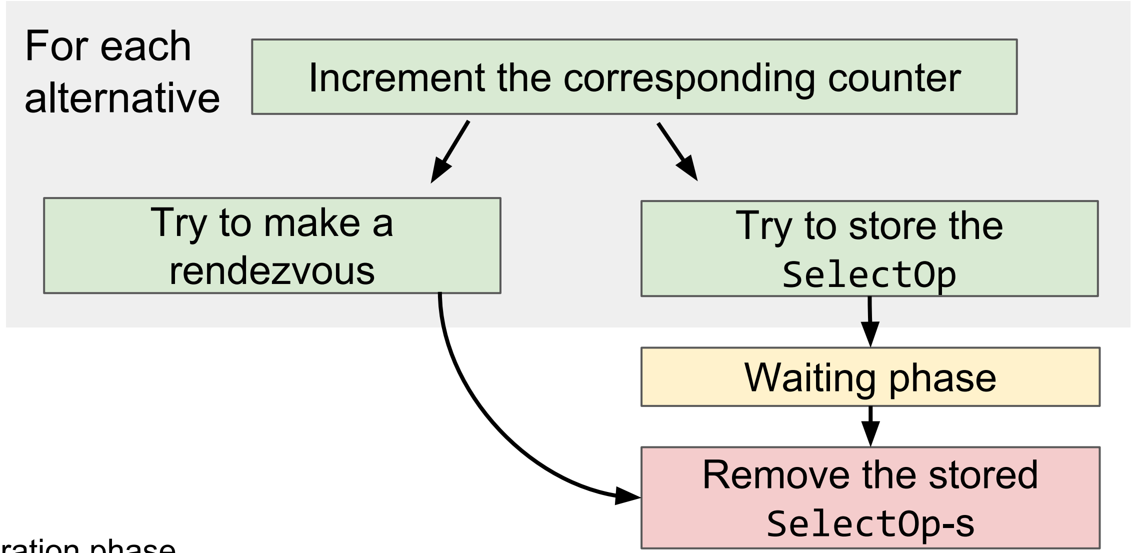
How to make a rendezvous with this `select` instance?

The `select` Expression: Second Solution


`SelectOp`

`alternatives`

`state`


For each alternative

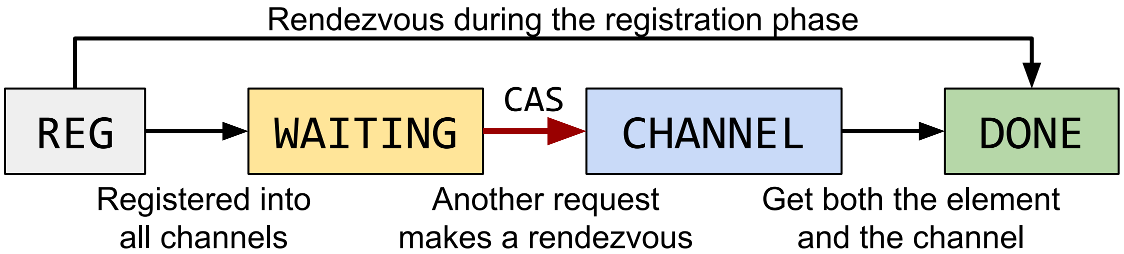
Increment the corresponding counter

Try to make a rendezvous

Try to store the `SelectOp`

Waiting phase

Remove the stored `SelectOp`-s


Rendezvous during the registration phase

`REG` → `WAITING` → CAS → `CHANNEL` → `DONE`

Registered into all channels

Another request makes a rendezvous

Get both the element and the channel

The `select` Expression: Second Solution

Client:

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

Worker:

```
val task = tasks.receive()
processTask(task)
```

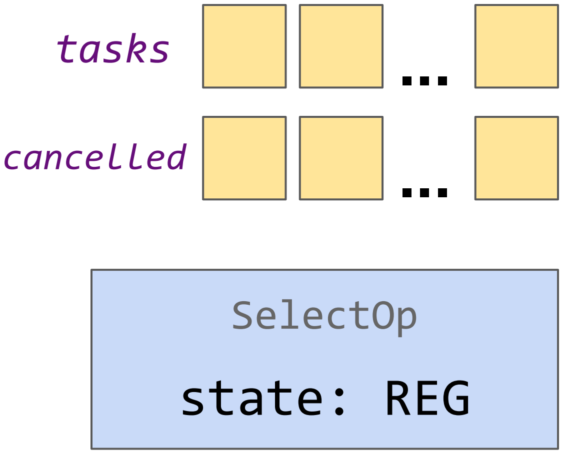
The `select` Expression: Second Solution

Client:

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

Worker:

```kotlin
val task = tasks.receive()
processTask(task)
```



tasks ...

cancelled ...

SelectOp

state: REG

The `select` Expression: Second Solution

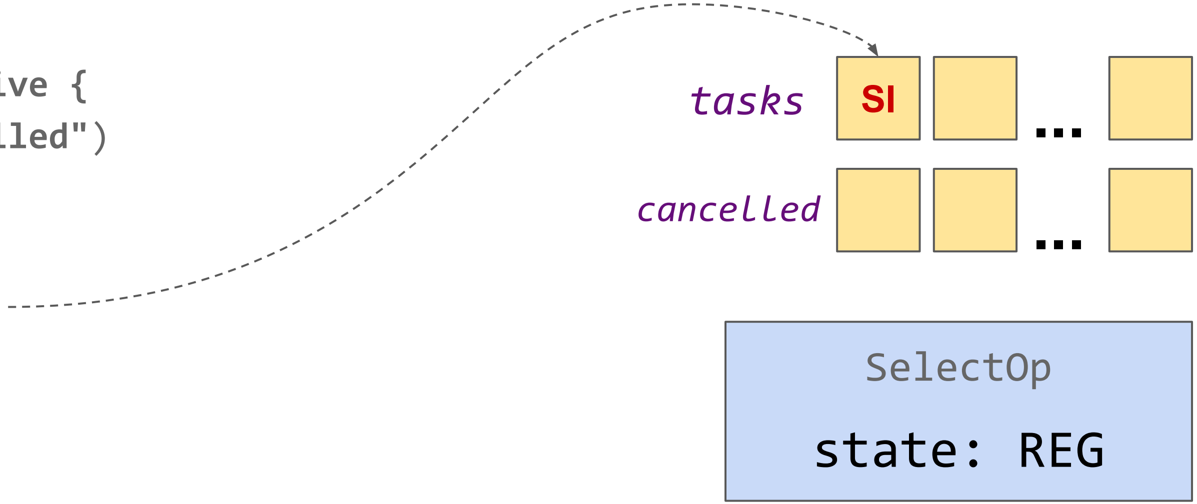
Client:

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

`C:` **Register in *tasks***

Worker:

```
val task = tasks.receive()
processTask(task)
```



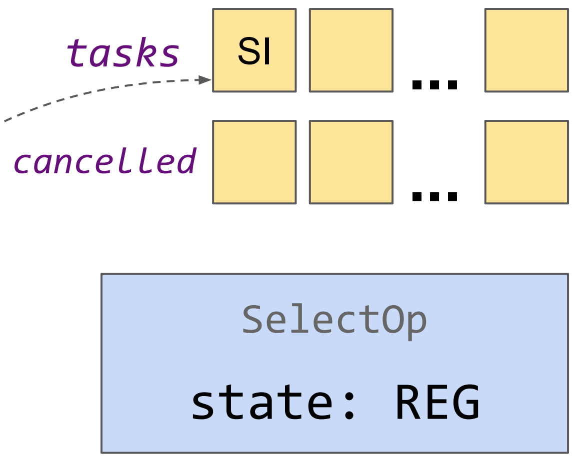
The `select` Expression: Second Solution

Client:

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

Worker:

```
val task = tasks.receive()
processTask(task)
```



`C:` Register in *tasks*

`W:` Rendezvous attempt in *tasks*, wait for `state != REG`

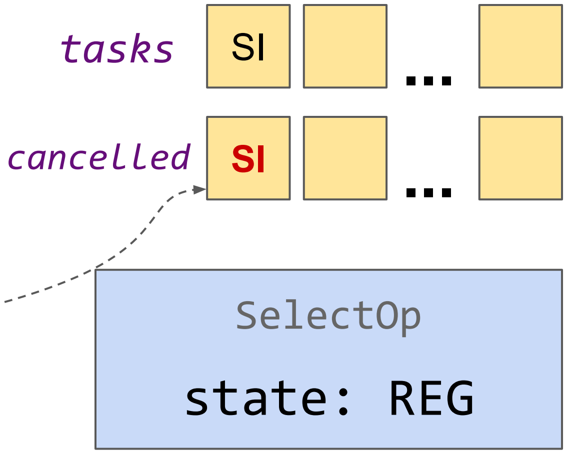
The `select` Expression: Second Solution

Client:

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

Worker:

```
val task = tasks.receive()
processTask(task)
```



`C:` Register in *tasks*

`W:` Rendezvous attempt in *tasks*, wait for `state != REG`

`C:` **Register in *cancelled***

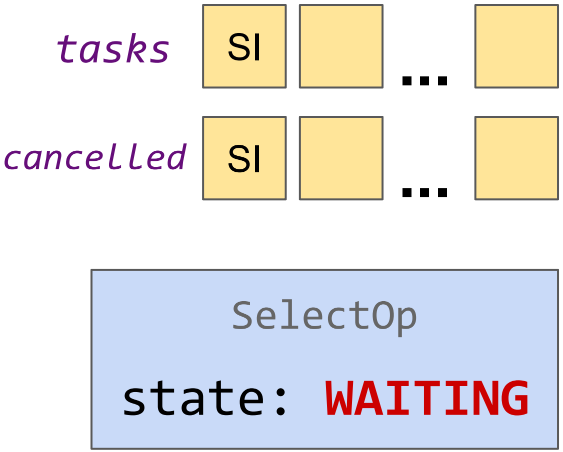
The `select` Expression: Second Solution

Client:

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

Worker:

```
val task = tasks.receive()
processTask(task)
```



`C:` Register in *tasks*

`W:` Rendezvous attempt in *tasks*, wait for `state != REG`

`C:` Register in *cancelled*

`C:` Change state to `WAITING`

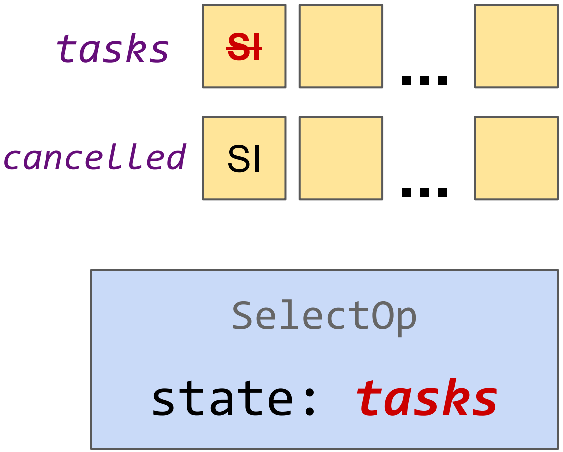
The `select` Expression: Second Solution

Client:

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

Worker:

```
val task = tasks.receive()
processTask(task)
```



tasks ~~SI~~ …

cancelled SI …

`SelectOp`

`state:` ***`tasks`***

`C:` Register in *`tasks`*

`W:` Rendezvous attempt in *`tasks`*, wait for `state != REG`

`C:` Register in *`cancelled`*

`C:` Change `state` to `WAITING`

`W:` Change state to *`tasks`*, the rendezvous done

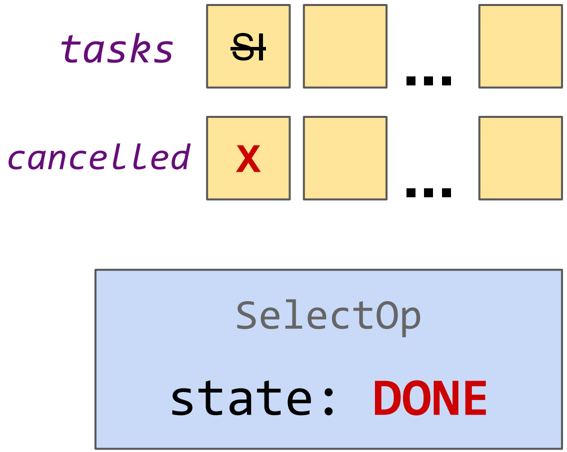
The `select` Expression: Second Solution

Client:

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

Worker:

```
val task = tasks.receive()
processTask(task)
```



`C:` Register in *`tasks`*
`W:` Rendezvous attempt in *`tasks`*, wait for `state != REG`
`C:` Register in *`cancelled`*
`C:` Change `state` to `WAITING`
`W:` Change `state` to *`tasks`*, the rendezvous done
`C:` Selected, change `state` to `DONE`

The `select` Expression: Deadlock Avoidance

Coroutine 1:

```
select<Unit> {
  chan_1.onSend(task) { ... }
  chan_2.onReceive { ... }
}
```

Coroutine 2:

```
select<Unit> {
  chan_2.onSend(task) { ... }
  chan_1.onReceive { ... }
}
```

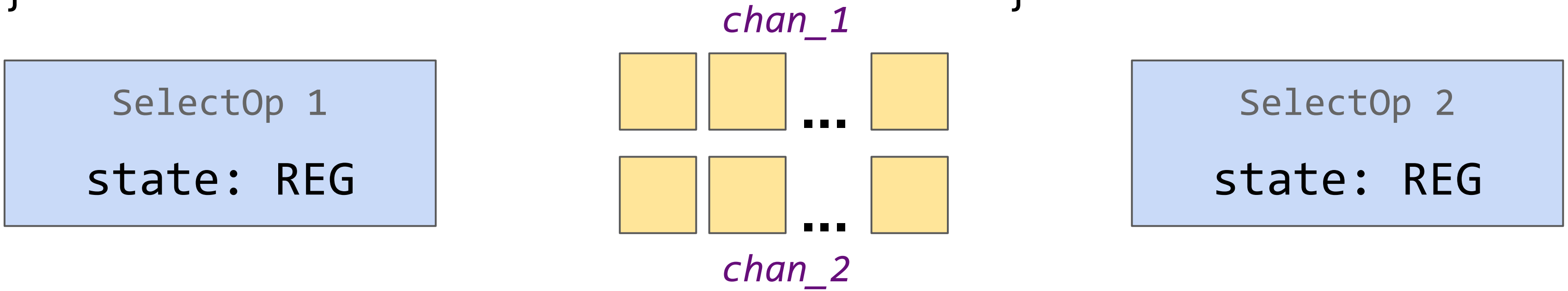
The `select` Expression: Deadlock Avoidance

Coroutine 1:

```
select<Unit> {
  chan_1.onSend(task) { ... }
  chan_2.onReceive { ... }
}
```

Coroutine 2:

```
select<Unit> {
  chan_2.onSend(task) { ... }
  chan_1.onReceive { ... }
}
```



SelectOp 1

state: REG

chan_1

...

...

chan_2

SelectOp 2

state: REG

The `select` Expression: Deadlock Avoidance

Coroutine 1:

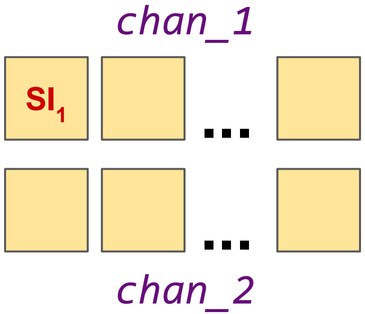
```
select<Unit> {
  chan_1.onSend(task) { ... }
  chan_2.onReceive { ... }
}
```

Coroutine 2:

```
select<Unit> {
  chan_2.onSend(task) { ... }
  chan_1.onReceive { ... }
}
```

`SelectOp 1`

`state: REG`



chan_1

SI_1 ...

...

chan_2

`SelectOp 2`

`state: REG`

`1. C1:` **Register in *chan_1***

The `select` Expression: Deadlock Avoidance

Coroutine 1:

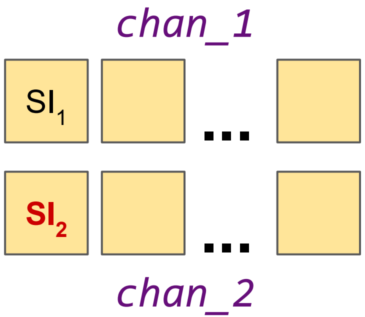
```
select<Unit> {
  chan_1.onSend(task) { ... }
  chan_2.onReceive { ... }
}
```

Coroutine 2:

```
select<Unit> {
  chan_2.onSend(task) { ... }
  chan_1.onReceive { ... }
}
```

SelectOp 1

state: REG



SelectOp 2

state: REG

`1. C1:` Register in *chan_1*

`2. C2:` Register in *chan_2*

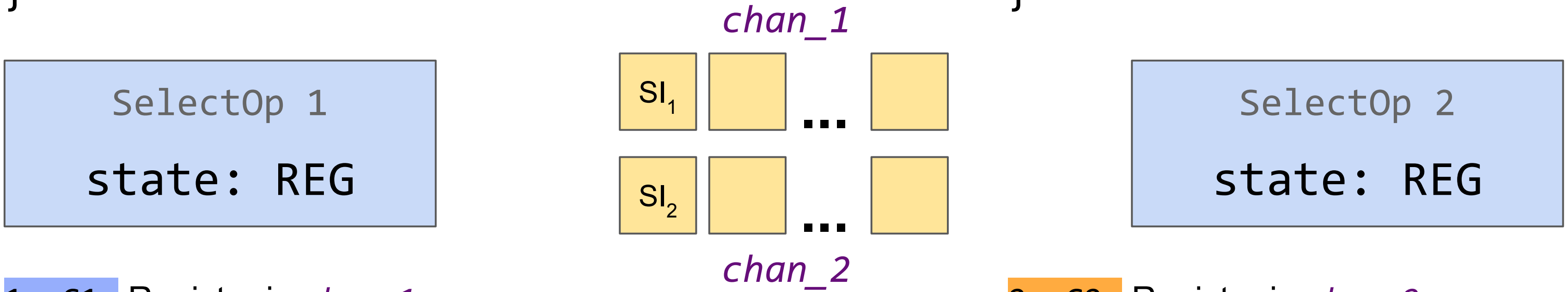
The `select` Expression: Deadlock Avoidance

Coroutine 1:

```
select<Unit> {
  chan_1.onSend(task) { ... }
  chan_2.onReceive { ... }
}
```

Coroutine 2:

```
select<Unit> {
  chan_2.onSend(task) { ... }
  chan_1.onReceive { ... }
}
```



`1. C1:` Register in *chan_1*

`2. C2:` Register in *chan_2*

`3. C1:` **Rendezvous attempt in *chan_2*,**
wait for `state != REG`

The `select` Expression: Deadlock Avoidance

Coroutine 1:

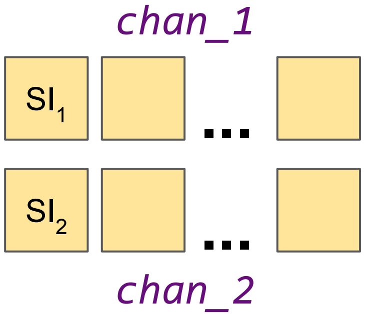
```
select<Unit> {
  chan_1.onSend(task) { ... }
  chan_2.onReceive { ... }
}
```

Coroutine 2:

```
select<Unit> {
  chan_2.onSend(task) { ... }
  chan_1.onReceive { ... }
}
```

`SelectOp 1`

`state: REG`



`SelectOp 2`

`state: REG`

`1. C1:` Register in *chan_1*

`3. C1:` Rendezvous attempt in *chan_2*, wait for `state != REG`

`2. C2:` Register in *chan_2*

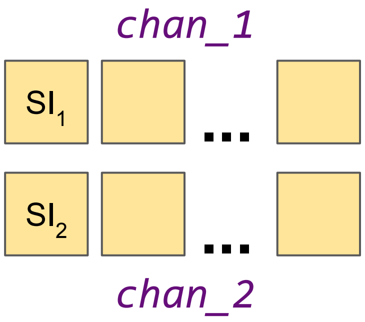
`4. C2:` Rendezvous attempt in *chan_1*, wait for `state != REG`

Deadlock!

The `select` Expression: Deadlock Avoidance

`SelectOp 1`

`state: REG`



`SelectOp 2`

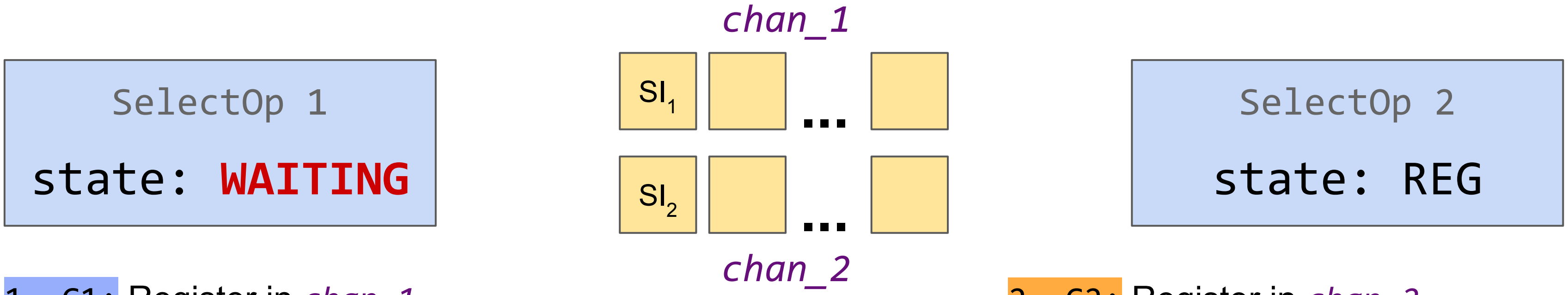
`state: REG`

`1. C1:` Register in *chan_1*
`3. C1:` Rendezvous attempt in *chan_2*, wait for `state != REG`

`2. C2:` Register in *chan_2*
`4. C2:` Rendezvous attempt in *chan_1*, wait for `state != REG`

1. Each `select` instance has unique id
2. Change the state of the select instance of minimal id in a waiting cycle from REG to WAITING

The `select` Expression: Deadlock Avoidance



`1. C1:` Register in *chan_1*
`3. C1:` Rendezvous attempt in *chan_2*,
wait for `state != REG`
`5. C1:` Deadlock, change state to `WAITING`

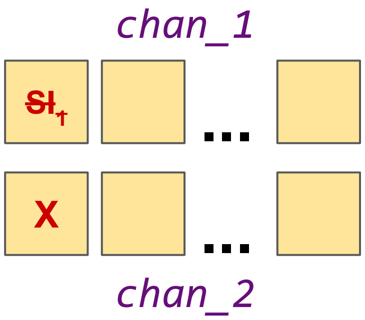
`2. C2:` Register in *chan_2*
`4. C2:` Rendezvous attempt in *chan_1*,
wait for `state != REG`

1. Each `select` instance has unique id
2. Change the state of the select instance of minimal id in a waiting cycle from REG to WAITING

The `select` Expression: Deadlock Avoidance

`SelectOp 1`

`state:` ***chan_1***



`SelectOp 2`

`state:` **`DONE`**

`1. C1:` Register in *chan_1*

`3. C1:` Rendezvous attempt in *chan_2*,
wait for `state != REG`

`5. C1:` Deadlock, change state to `WAITING`

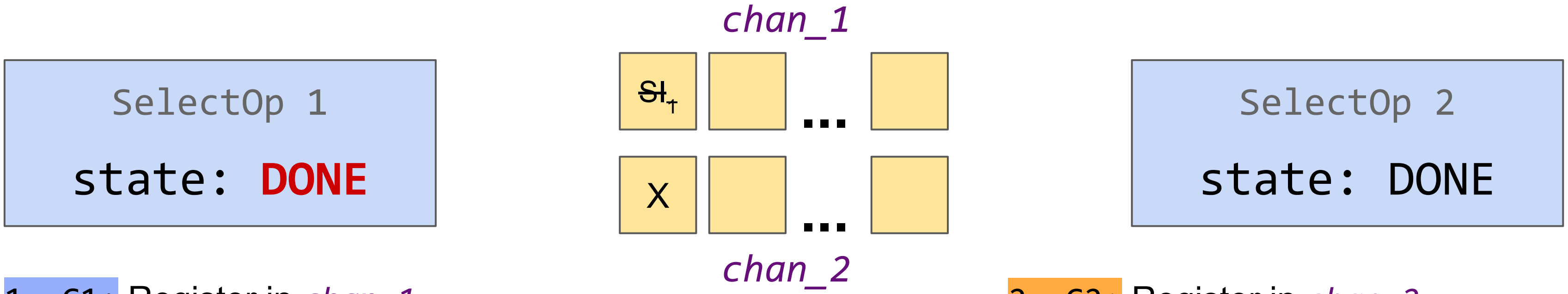
`2. C2:` Register in *chan_2*

`4. C2:` Rendezvous attempt in *chan_1*,
wait for `state != REG`

`6. C2:` Change 1st state to *chan_1*, rendezvous done

1. Each `select` instance has unique id
2. Change the state of the select instance of minimal id in a waiting cycle from REG to WAITING

The `select` Expression: Deadlock Avoidance

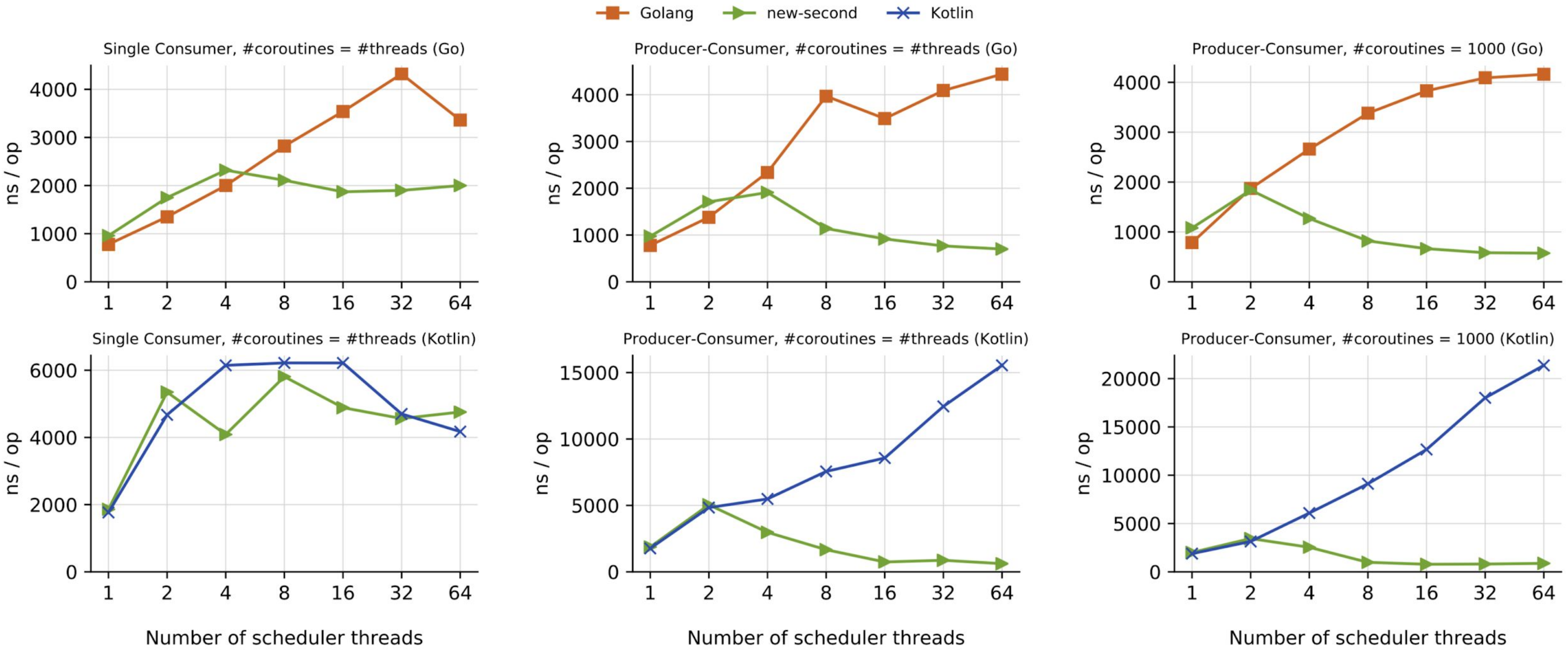


`1. C1:` Register in *chan_1*
`3. C1:` Rendezvous attempt in *chan_2*, wait for `state != REG`
`5. C1:` Deadlock, change state to `WAITING`
`7. C1:` Selected, change state to `DONE`

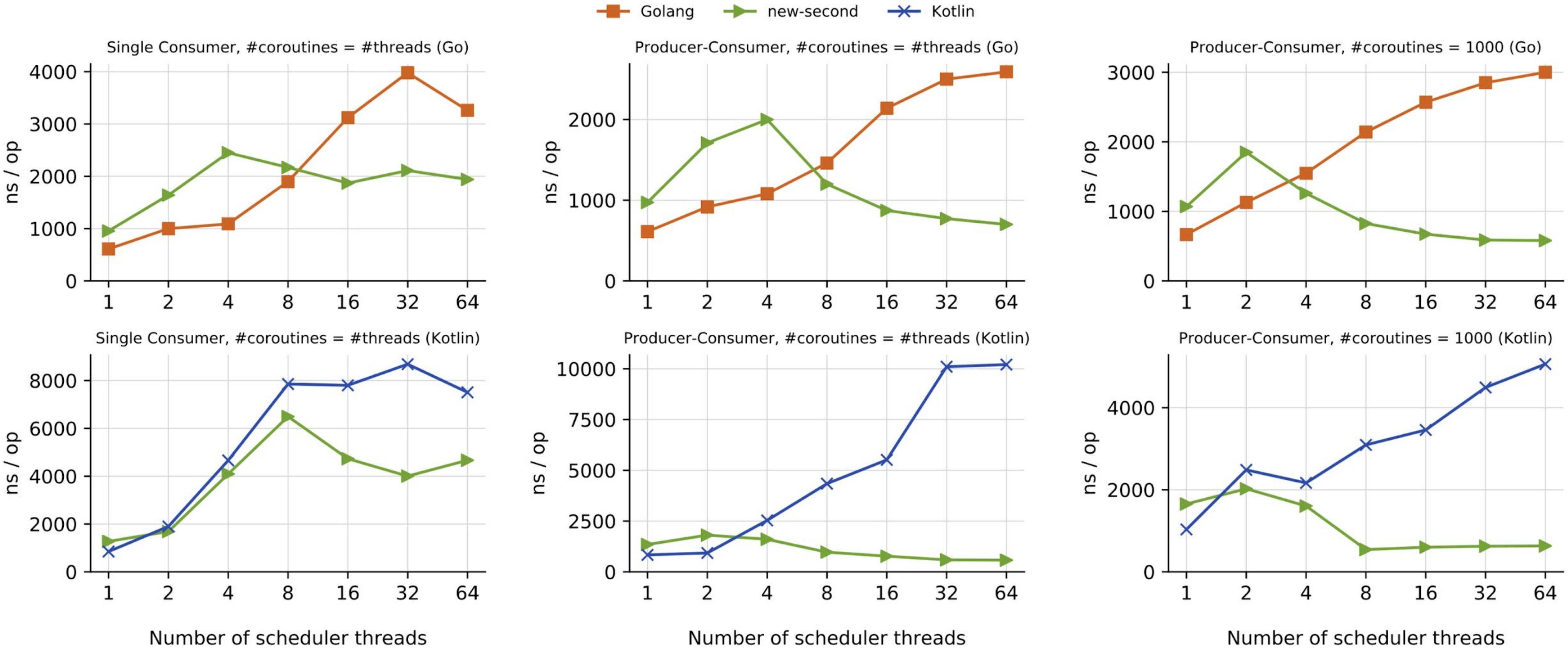
`2. C2:` Register in *chan_2*
`4. C2:` Rendezvous attempt in *chan_1*, wait for `state != REG`
`6. C2:` Change 1st state to *chan_1*, rendezvous done

1. Each `select` instance has unique id
2. Change the state of the select instance of minimal id in a waiting cycle from REG to WAITING

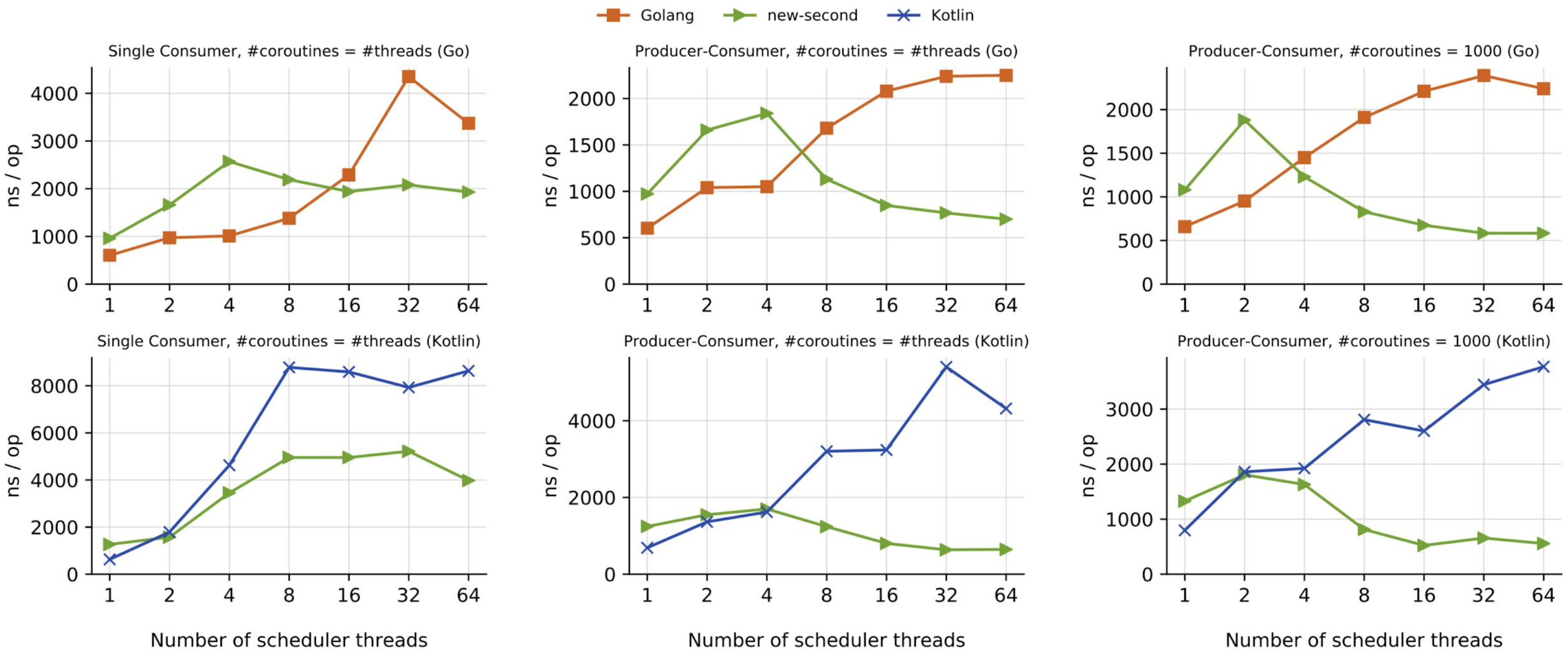
The `select` Expression (capacity = 0)



The `select` Expression (capacity = 32)



The `select` Expression (capacity = 128)



Instead of Summary

- Locks != bad
- Non-blocking != scalable
- Nowadays concurrent programming is full of trade-offs

Channels in Kotlin Coroutines are the best in the world 😉
https://github.com/Kotlin/kotlinx.coroutines/tree/channels

Questions