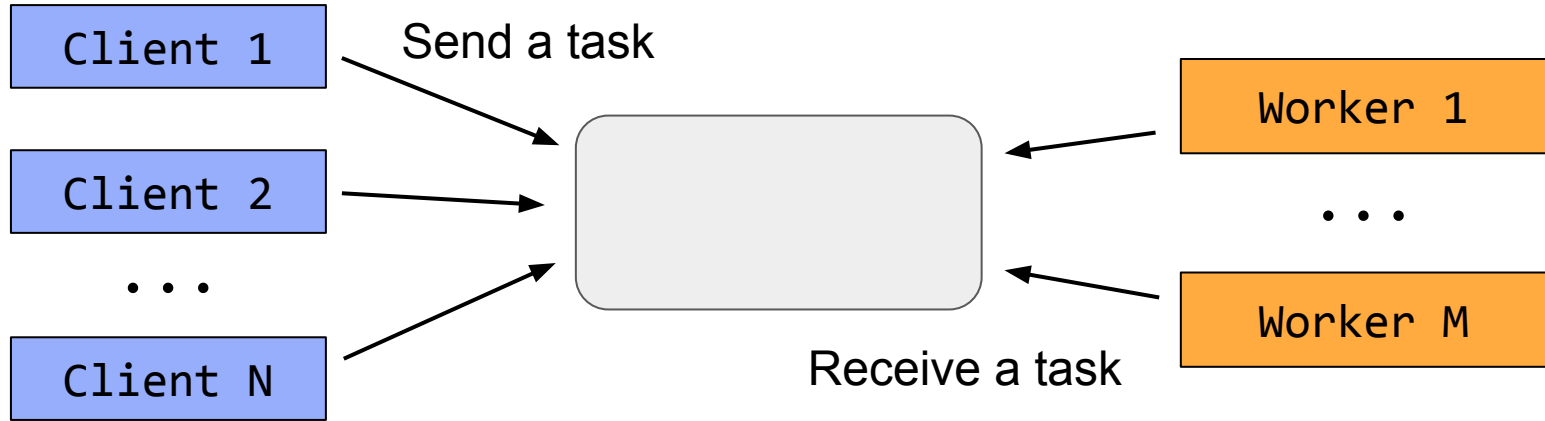


Dual Data Structures

Никита Коваль, JetBrains
Роман Елизаров, JetBrains

ИТМО, 2020

Producer-Consumer Problem



Synchronous Queue aka Rendezvous Channel

```
interface SynchronousQueue<E> {  
    suspend fun send(element: E)  
    suspend fun receive(): E  
}
```

Senders and receivers perform a rendezvous handshake as a part of their protocol (senders wait for receivers and vice versa)

Producer-Consumer Problem Solution

1. Let's create a synchronous queue

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

Producer-Consumer Problem Solution

1. Let's create a synchronous queue

```
val tasks = SynchronousQueue<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 synchronous queue

```
val tasks = SynchronousQueue<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)
```

Client 2

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

Worker

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

```
val tasks = SynchronousQueue<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)
}
```

Have to wait for send



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


Rendezvous Channel Semantics

Client 1

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

Client 2

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

Worker



1

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

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

Rendezvous Channel Semantics

Client 1

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

Client 2

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

Worker



1

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

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

Rendezvous Channel Semantics

Client 1

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

```
2 tasks.send(task)
```

Client 2

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

```
tasks.send(task)
```

Rendezvous!

Worker

```
while(true) {
```

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

```
val tasks = SynchronousQueue<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 = SynchronousQueue<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)  
}
```

Have to wait for receive

```
val tasks = SynchronousQueue<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 = SynchronousQueue<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) {
```

5 1 `val task = tasks.receive()`

3 `processTask(task)`
}

Rendezvous!

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

Coroutines Management

Coroutines Management

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

Element to be sent

```
fun curCoroutine(): Coroutine { ... }
```

Returns the current coroutine

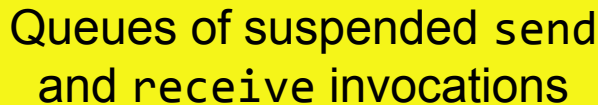
```
suspend fun suspend(c: Coroutine) { ... }  
fun resume(c: Coroutine) { ... }
```

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?  
    ...  
}
```

Check if there is no
receiver and suspends

```
fun curCo
```

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

Rendezvous: retrieve the first receiver

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

```
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

“Our synchronous queues have been
adopted for inclusion in Java 6”
`j.u.c.SynchronousQueue`

Abstract

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

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 many Java `ThreadPoolExecutor` mechanisms (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 Techniques]: Concurrent Programming—Parallel Programming

son [3], which uses three... Such heavy synchronization burdens are especially... 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 if the pool is... threads terminate themselves if no... if the pool is

Rendezvous Channel: Java

Based on Michael-Scott lock-free queue algorithm
the simplest known lock-free queue

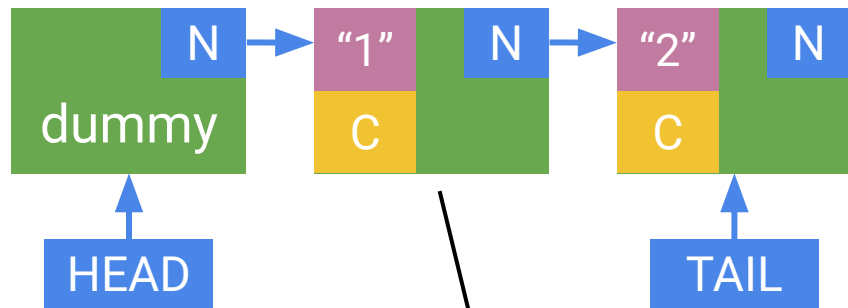
Rendezvous Channel: Java

Based on Michael-Scott lock-free queue algorithm
the simplest known lock-free queue

Either senders or receivers are in the queue!

Rendezvous Channel: Java

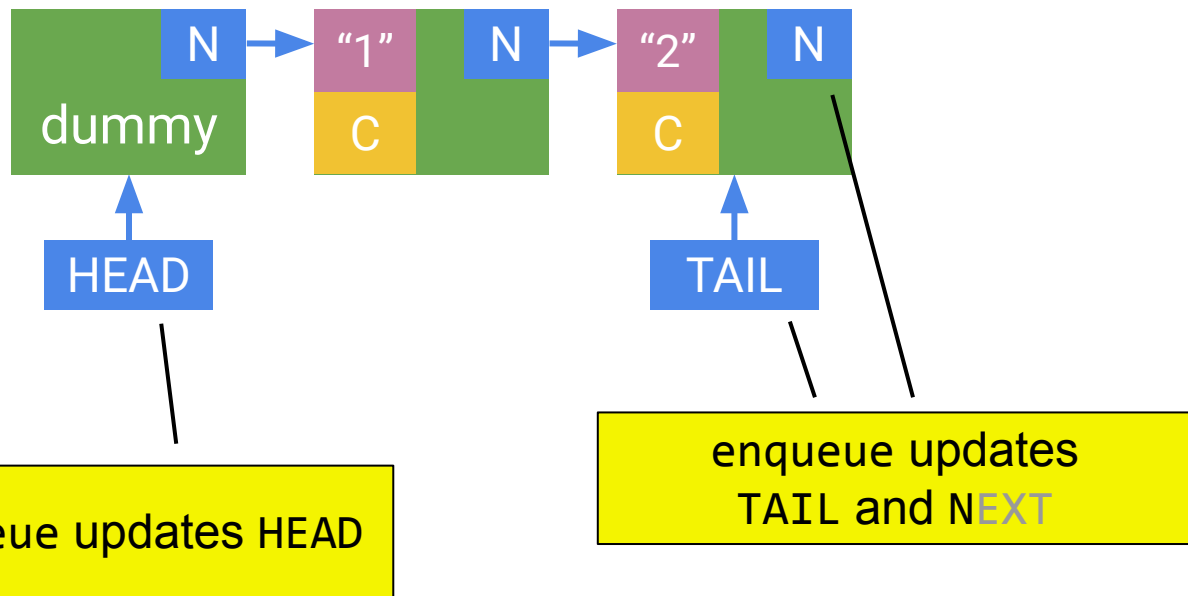
Based on Michael-Scott lock-free queue algorithm
the simplest known lock-free queue



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

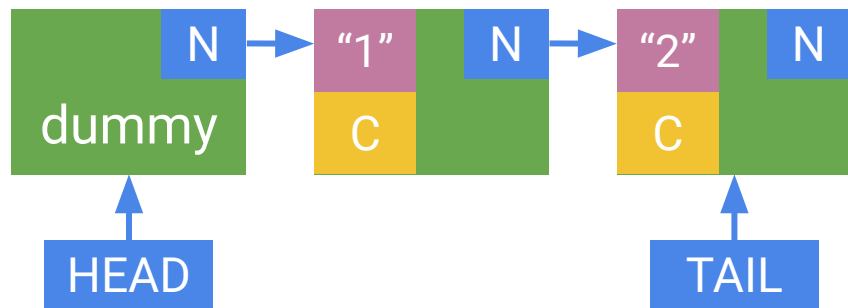
Rendezvous Channel: Java

Based on Michael-Scott lock-free queue algorithm
the simplest known lock-free queue



Rendezvous Channel: Java

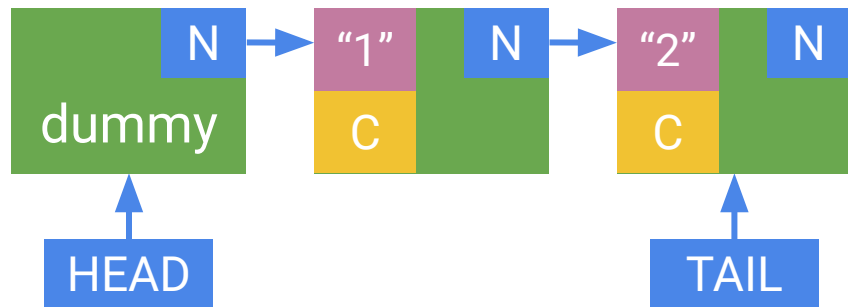
Based on Michael-Scott lock-free queue algorithm
the simplest known lock-free queue



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

Rendezvous Channel: Java

Based on Michael-Scott lock-free queue algorithm
the simplest known lock-free queue



Retry the whole
operation on failures

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

Does It Guarantee Linearizability?

<code>val c = Channel<Int>()</code>	
<code>c.send(4)</code>	<code>c.receive() // S + 4</code>

send waits for receive and vice versa

Does It Guarantee Linearizability?

<code>val c = Channel<Int>()</code>	
<code>c.send(4)</code>	<code>c.receive() // S + 4</code>

Non-linearizable
because of suspension


```
val c = Channel<Int>()
```

```
c.receive(): // S + 4
```

register as a waiter

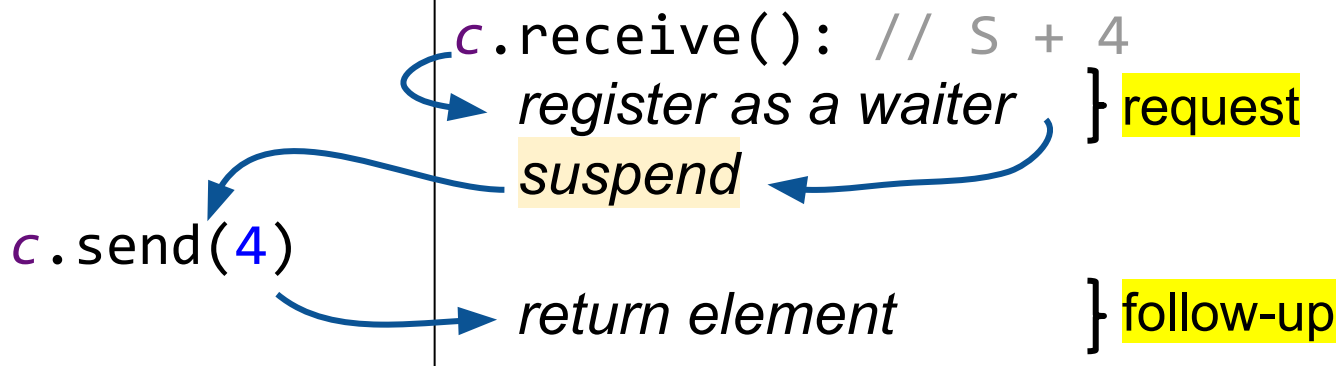
suspend

```
c.send(4)
```

return element

Dual Data Structures*

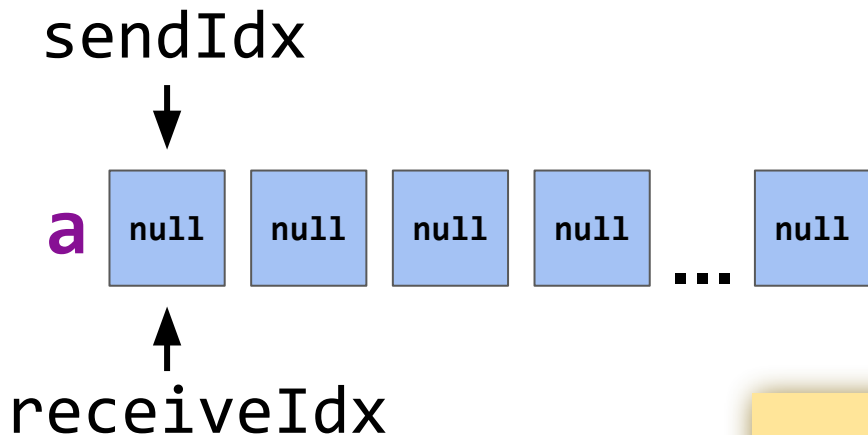
```
val c = Channel<Int>()
```



Channels are similar to queues...

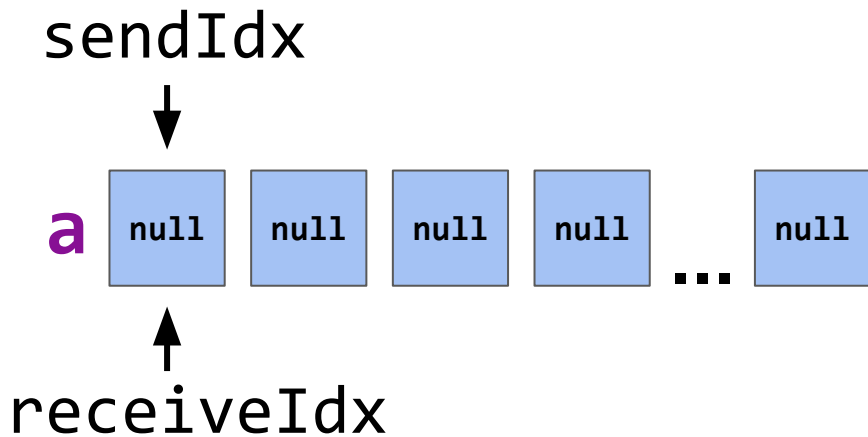
Can we make them faster with
Fetch-And-Add?

FAA-Based Channel on Array



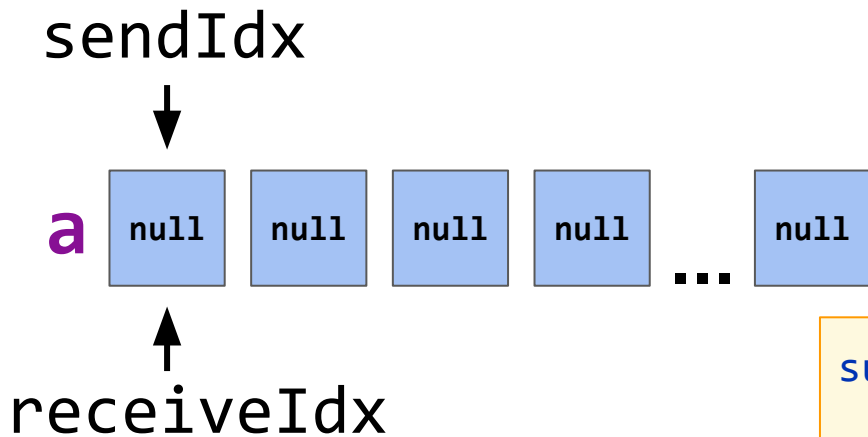
An infinite array with
indices for the next
send and receive

FAA-Based Channel on Array



```
suspend fun send(x: T) {  
    val s = FAA(&sendIdx, +1)  
    if s < receiveIdx:  
        // try to put f into w[s]  
        // or resume the receiver  
    else:  
        // suspend in w[s]  
}
```

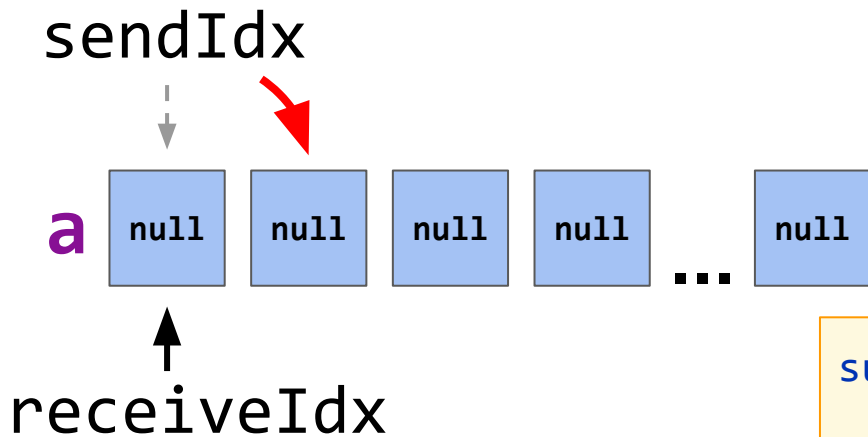
FAA-Based Channel on Array



```
suspend fun send(x: T) {  
    val s = FAA(&sendIdx, +1)  
    if s < receiveIdx:  
        // try to put f into w[s]  
        // or resume the receiver  
    else:  
        // suspend in w[s]  
}
```

```
suspend fun receive(): T {  
    val r = FAA(&receiveIdx, +1)  
    if r < sendIdx:  
        // try to retrieve either  
        // value or sender from w[r]  
    else:  
        // suspend in w[r]  
}
```

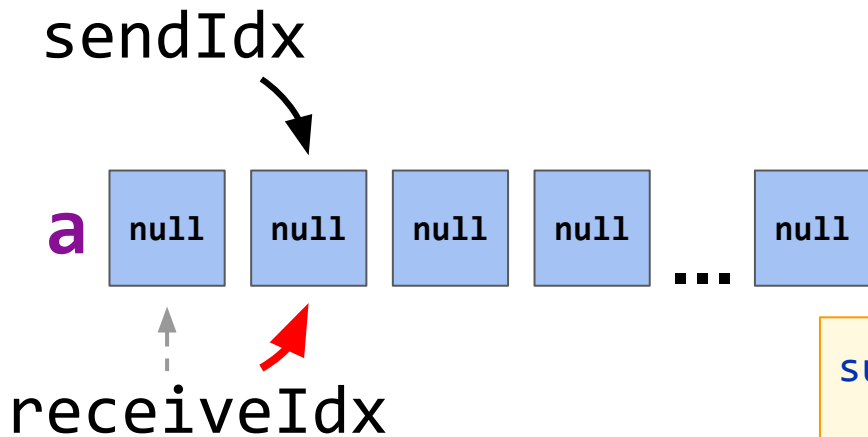
FAA-Based Channel on Array



```
suspend fun send(x: T) {  
    val s = FAA(&sendIdx, +1)  
    if s < receiveIdx:  
        // try to put f into w[s]  
        // or resume the receiver  
    else:  
        // suspend in w[s]  
}
```

```
suspend fun receive(): T {  
    val r = FAA(&receiveIdx, +1)  
    if r < sendIdx:  
        // try to retrieve either  
        // value or sender from w[r]  
    else:  
        // suspend in w[r]  
}
```

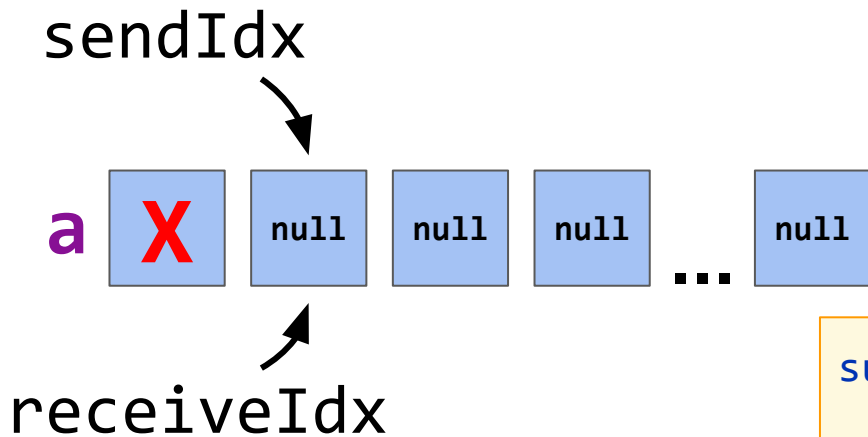
FAA-Based Channel on Array



```
suspend fun send(x: T) {  
    val s = FAA(&sendIdx, +1)  
    if s < receiveIdx:  
        // try to put f into w[s]  
        // or resume the receiver  
    else:  
        // suspend in w[s]  
}
```

```
suspend fun receive(): T {  
    val r = FAA(&receiveIdx, +1)  
    if r < sendIdx:  
        // try to retrieve either  
        // value or sender from w[r]  
    else:  
        // suspend in w[r]  
}
```

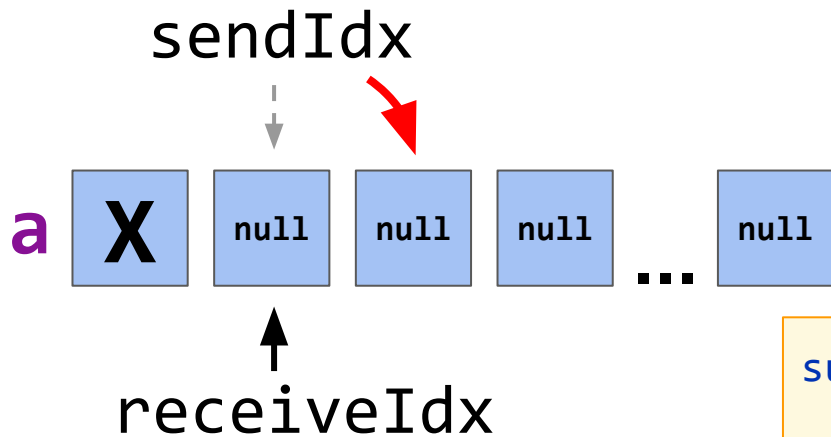

FAA-Based Channel on Array



```
suspend fun send(x: T) {  
    val s = FAA(&sendIdx, +1)  
    if s < receiveIdx:  
        // try to put f into w[s]  
        // or resume the receiver  
    else:  
        // suspend in w[s]  
}
```

```
suspend fun receive(): T {  
    val r = FAA(&receiveIdx, +1)  
    if r < sendIdx:  
        // try to retrieve either  
        // value or sender from w[r]  
    else:  
        // suspend in w[r]  
}
```

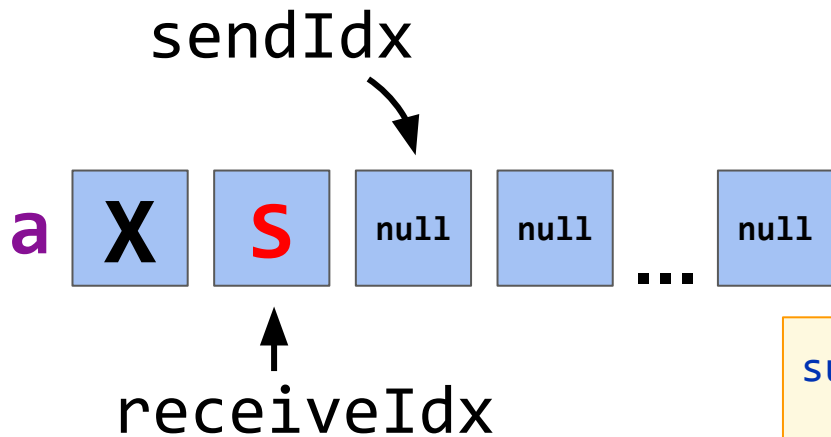
FAA-Based Channel on Array



```
suspend fun send(x: T) {  
    val s = FAA(&sendIdx, +1)  
    if s < receiveIdx:  
        // try to put f into w[s]  
        // or resume the receiver  
    else:  
        // suspend in w[s]  
}
```

```
suspend fun receive(): T {  
    val r = FAA(&receiveIdx, +1)  
    if r < sendIdx:  
        // try to retrieve either  
        // value or sender from w[r]  
    else:  
        // suspend in w[r]  
}
```

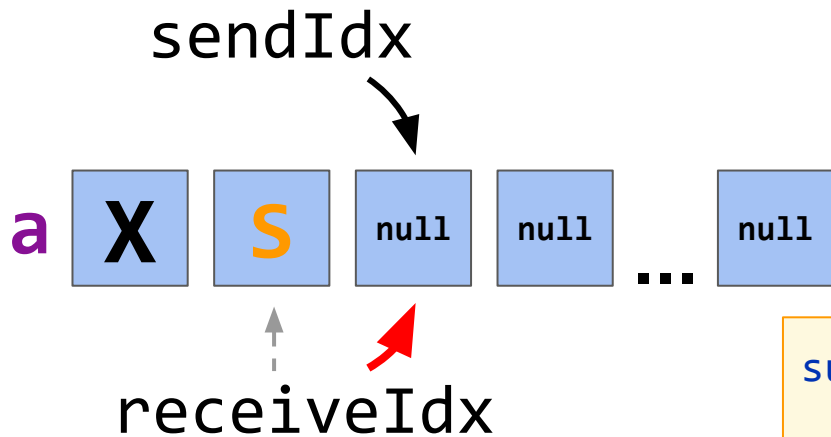
FAA-Based Channel on Array



```
suspend fun send(x: T) {  
    val s = FAA(&sendIdx, +1)  
    if s < receiveIdx:  
        // try to put f into w[s]  
        // or resume the receiver  
    else:  
        // suspend in w[s]  
}
```

```
suspend fun receive(): T {  
    val r = FAA(&receiveIdx, +1)  
    if r < sendIdx:  
        // try to retrieve either  
        // value or sender from w[r]  
    else:  
        // suspend in w[r]  
}
```

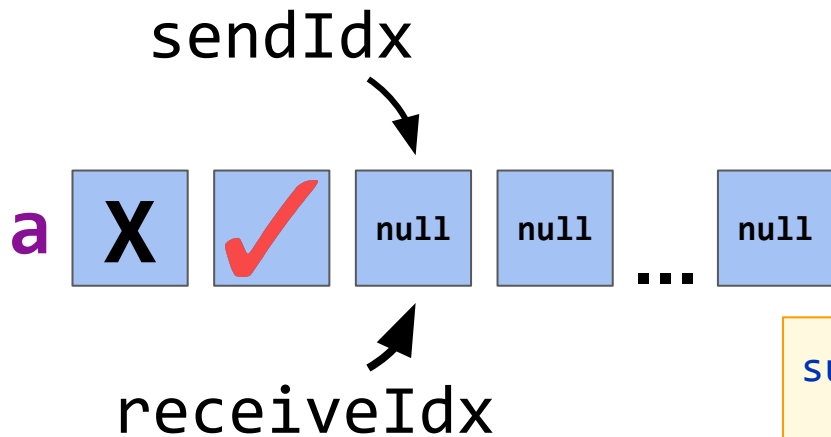
FAA-Based Channel on Array



```
suspend fun send(x: T) {  
    val s = FAA(&sendIdx, +1)  
    if s < receiveIdx:  
        // try to put f into w[s]  
        // or resume the receiver  
    else:  
        // suspend in w[s]  
}
```

```
suspend fun receive(): T {  
    val r = FAA(&receiveIdx, +1)  
    if r < sendIdx:  
        // try to retrieve either  
        // value or sender from w[r]  
    else:  
        // suspend in w[r]  
}
```

FAA-Based Channel on Array



```
suspend fun send(x: T) {  
    val s = FAA(&sendIdx, +1)  
    if s < receiveIdx:  
        // try to put f into w[s]  
        // or resume the receiver  
    else:  
        // suspend in w[s]  
}
```

```
suspend fun receive(): T {  
    val r = FAA(&receiveIdx, +1)  
    if r < sendIdx:  
        // try to retrieve either  
        // value or sender from w[r]  
    else:  
        // suspend in w[r]  
}
```