Understanding channels

@kavya719

kavya

inner workings of channels

concurrency features

goroutines

to execute tasks **independently**,
potentially in parallel.

channels

for **communication**, **synchronization**
between goroutines.

```
func main() {
   tasks := getTasks()

   // Process each task.
   for _, task := range tasks {
      process(task)
   }

   ...
}
```

→ hellaTasks

task queue

```
func main() {
   // Buffered channel.
    ch := make(chan Task, 3)
```

task queue

```
func main() {
   // Buffered channel.
    ch := make(chan Task, 3)

    // Run fixed number of workers.
   for i := 0; i < numWorkers; i++ {
      go worker(ch)
   }
```

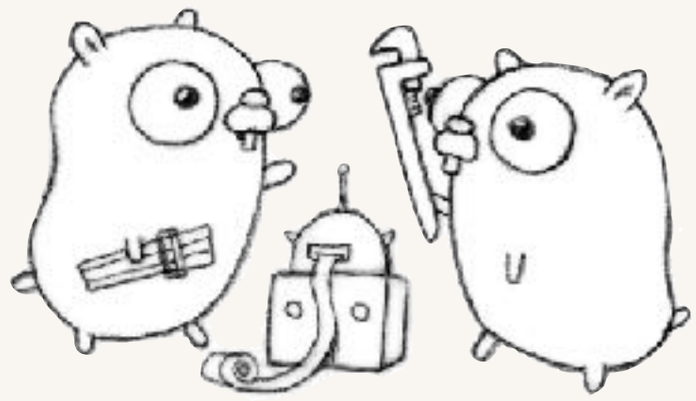
task queue

```
func main() {
   // Buffered channel.
    ch := make(chan Task, 3)

    // Run fixed number of workers.
   for i := 0; i < numWorkers; i++ {
      go worker(ch)
   }

   // Send tasks to workers.
   hellaTasks := getTasks()

   for _, task := range hellaTasks {
      ch <- task
   }

   ...
}
```

```
func worker(ch) {
   for {
      // Receive task.
      task := <-ch
      process(task)
   }
}
```

channels are inherently interesting

- **goroutine-safe**.
- store and **pass values between goroutines**.
- provide **FIFO semantics**.
- can **cause goroutines to block and unblock**.



making channels

the hchan struct

sends and receives

goroutine scheduling

stepping back:

design considerations

making channels

make chan

buffered channel

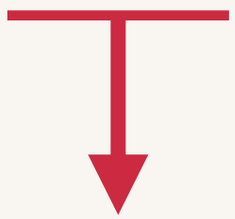
```
ch := make(chan Task, 3)
```

unbuffered channel

```
ch := make(chan int)
```

buffered channel

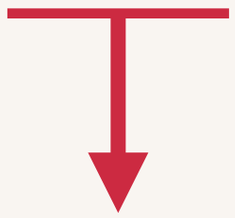
```
ch := make(chan Task, 3)
```



- **goroutine-safe**
- **stores up to capacity elements, and provides FIFO semantics**
- sends values between goroutines
- can cause them to block, unblock

buffered channel

```
ch := make(chan Task, 3)
```

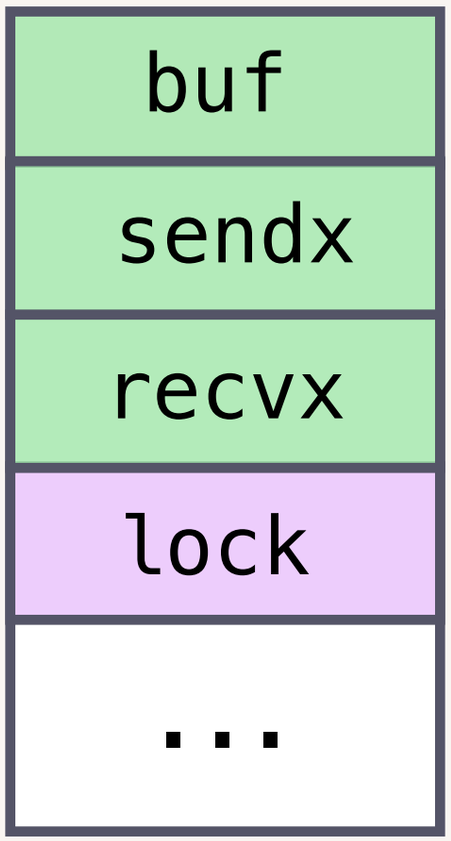


circular queue

send index

receive index

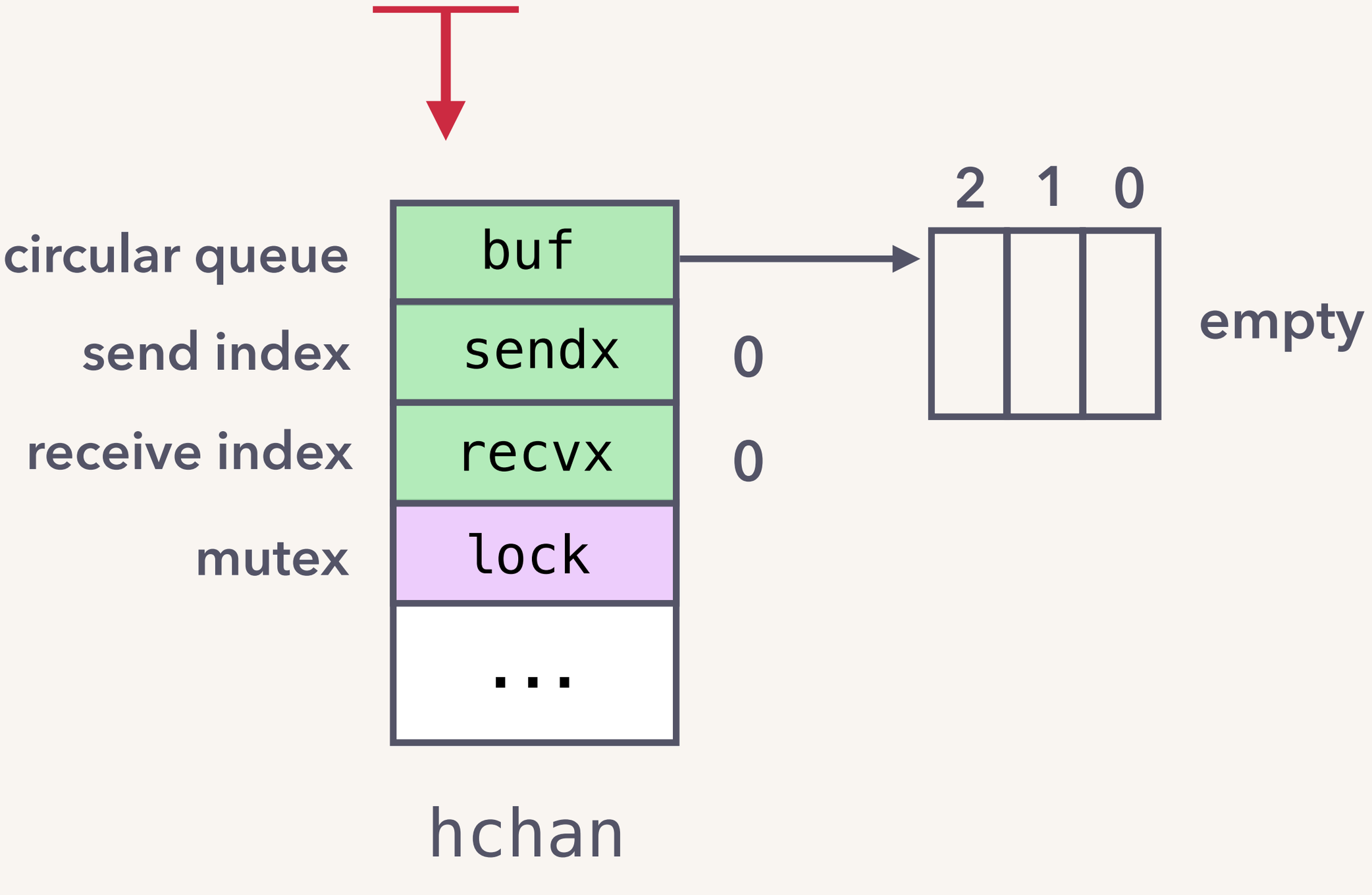
mutex



hchan

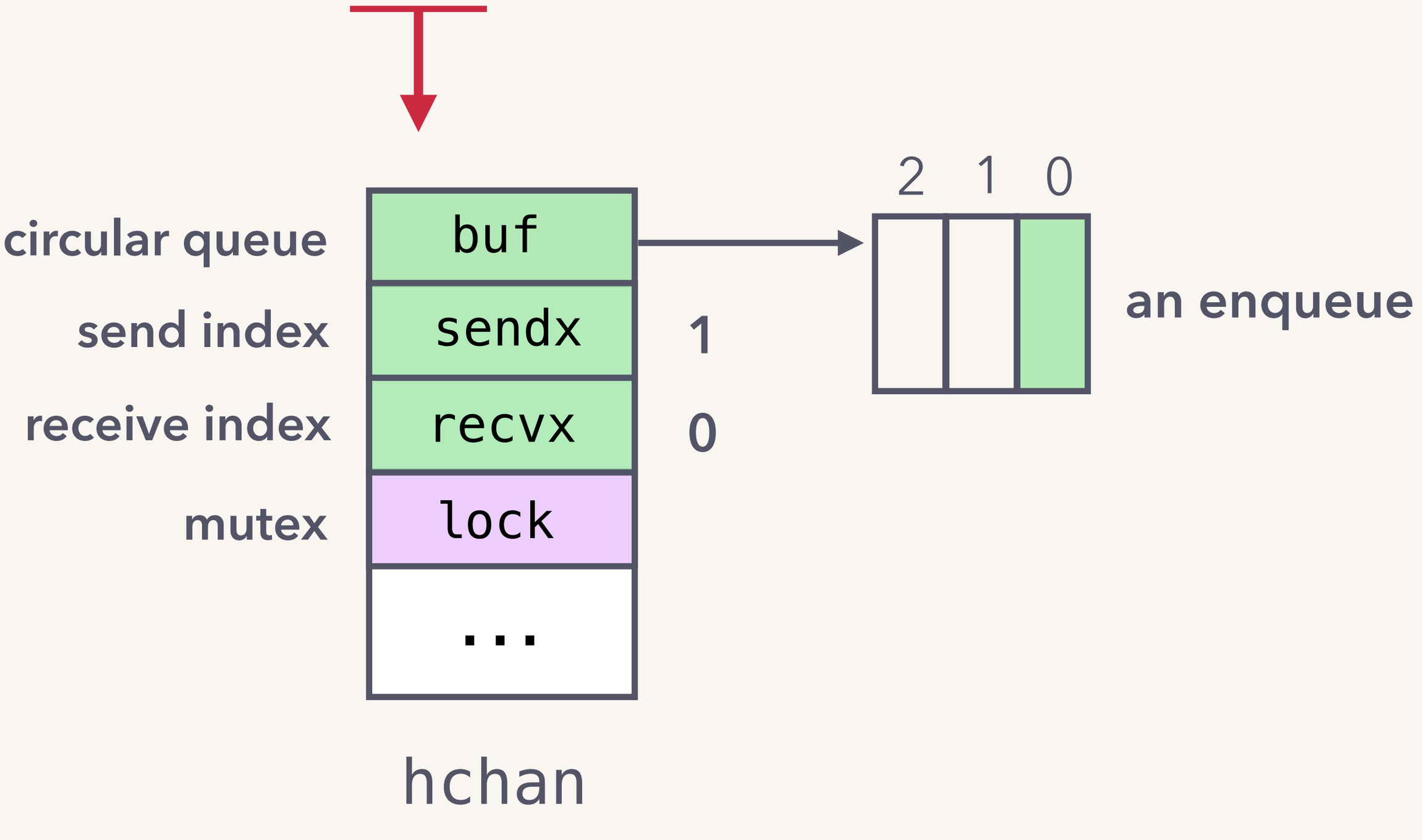
buffered channel

```
ch := make(chan Task, 3)
```



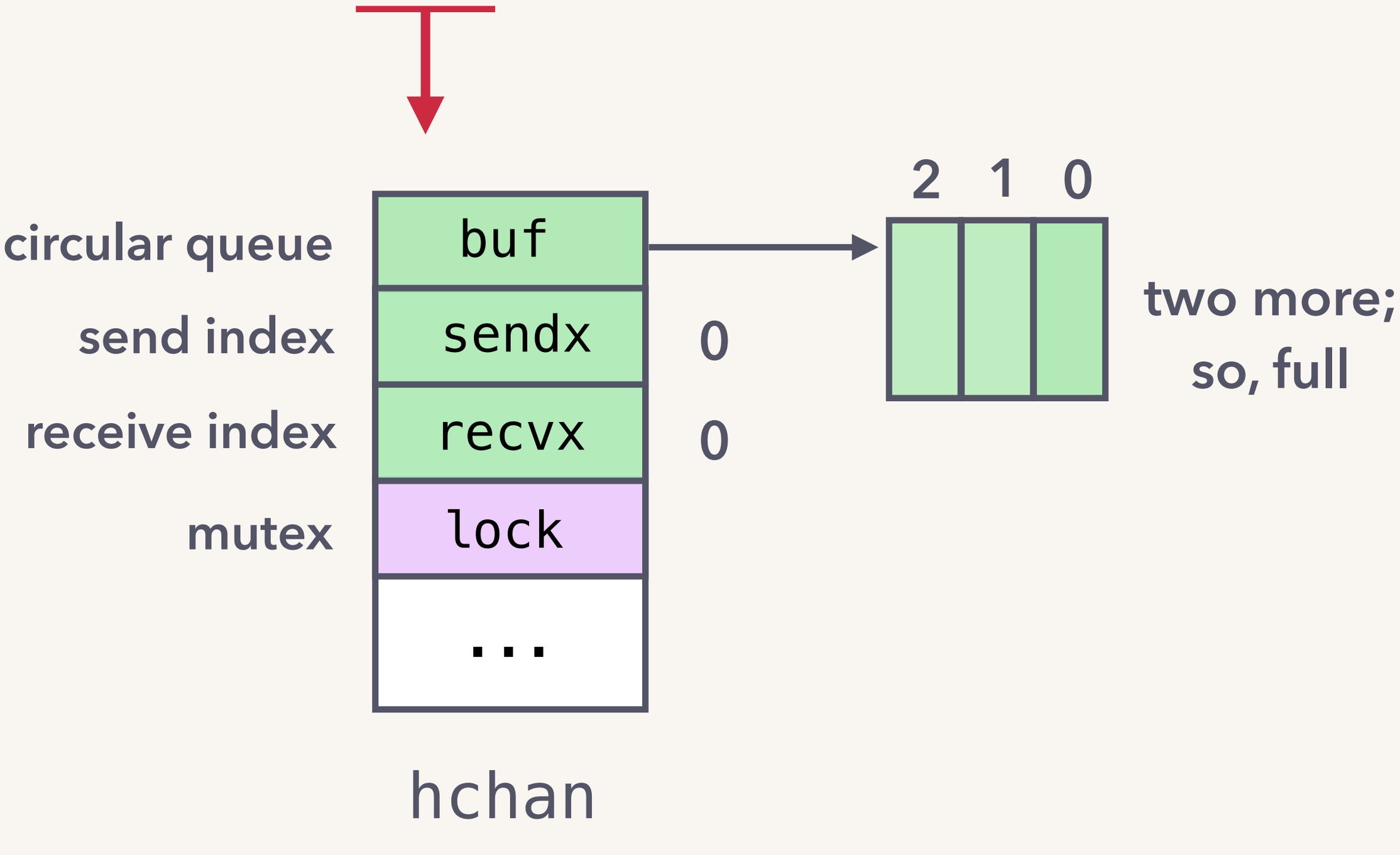
buffered channel

```
ch := make(chan Task, 3)
```



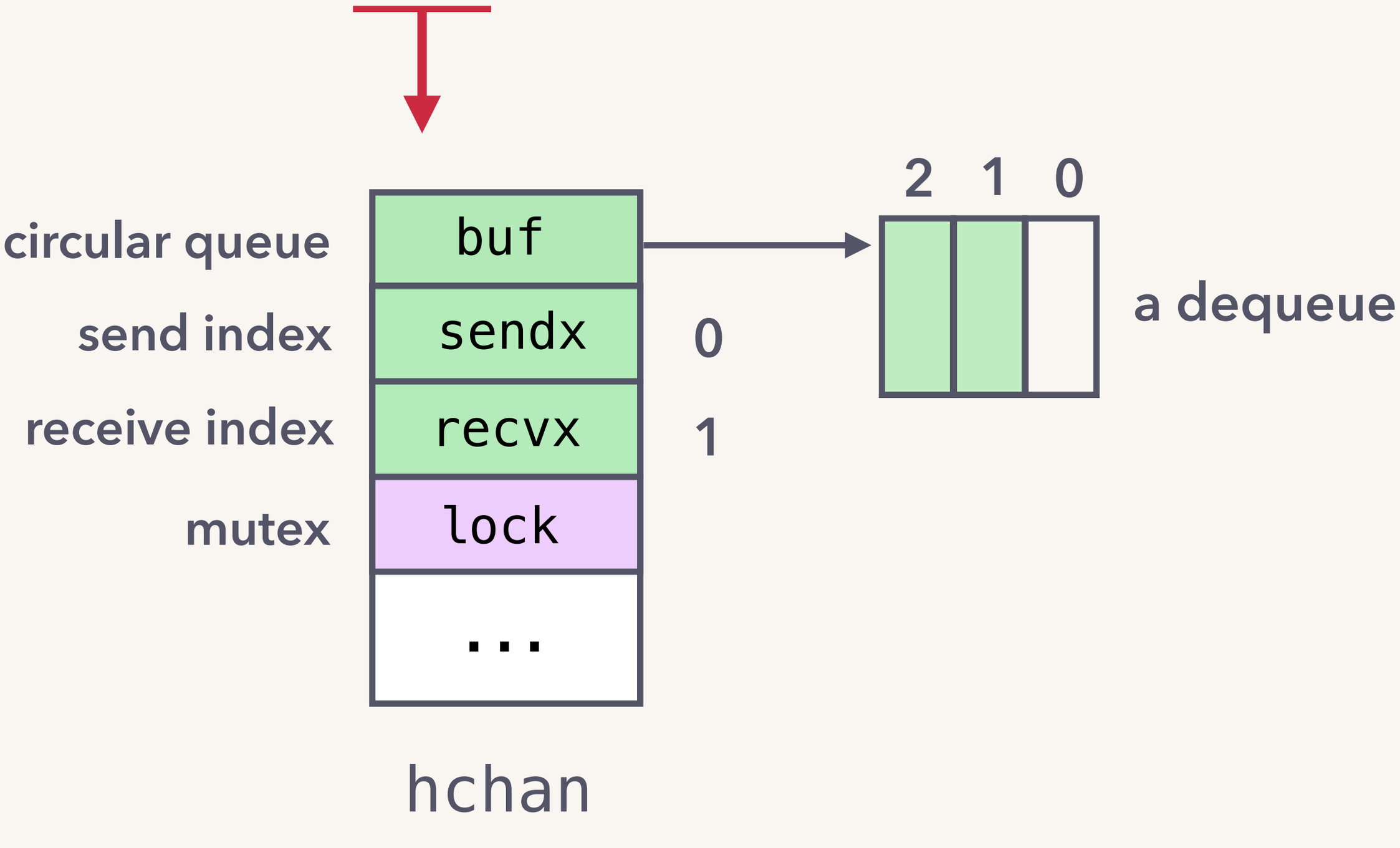
buffered channel

```
ch := make(chan Task, 3)
```



buffered channel

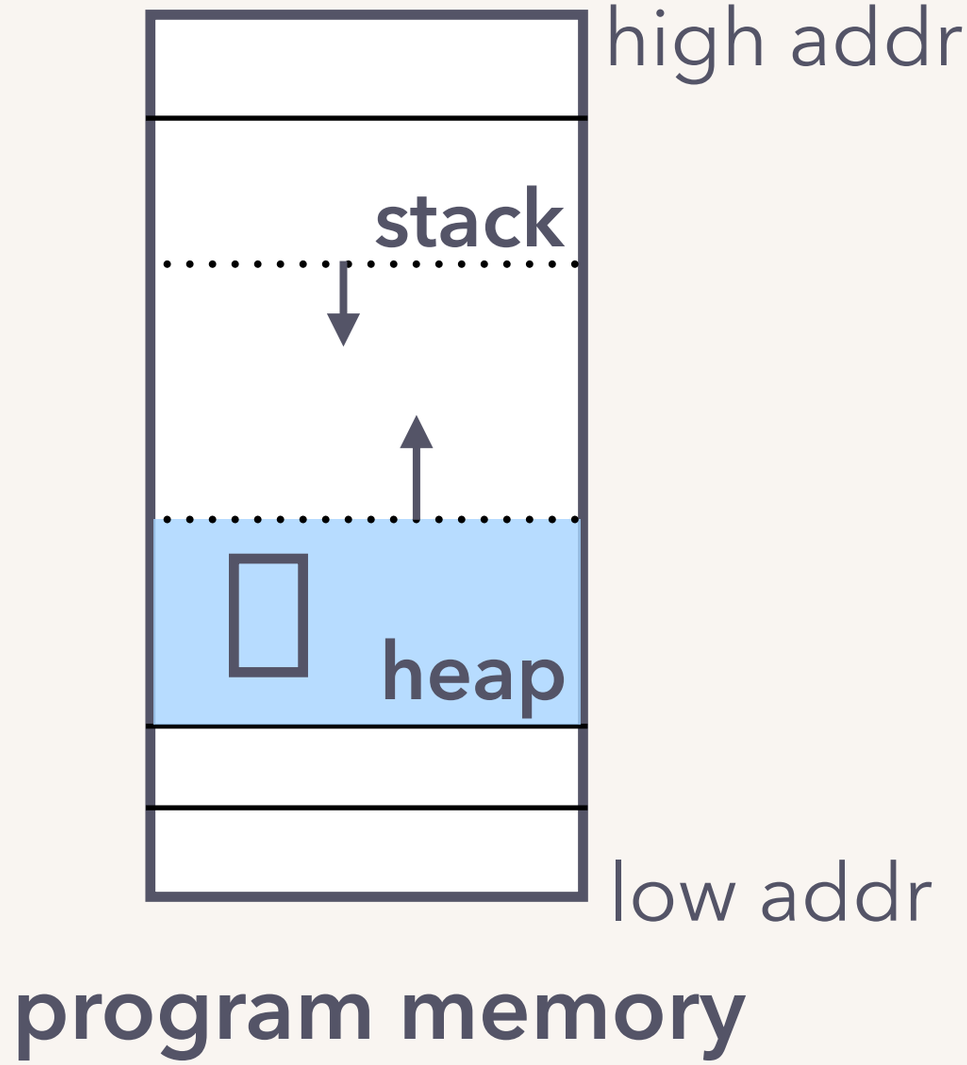
```
ch := make(chan Task, 3)
```



make chan

```
ch := make(chan Task, 3)
```

allocates an hchan struct **on the heap**.

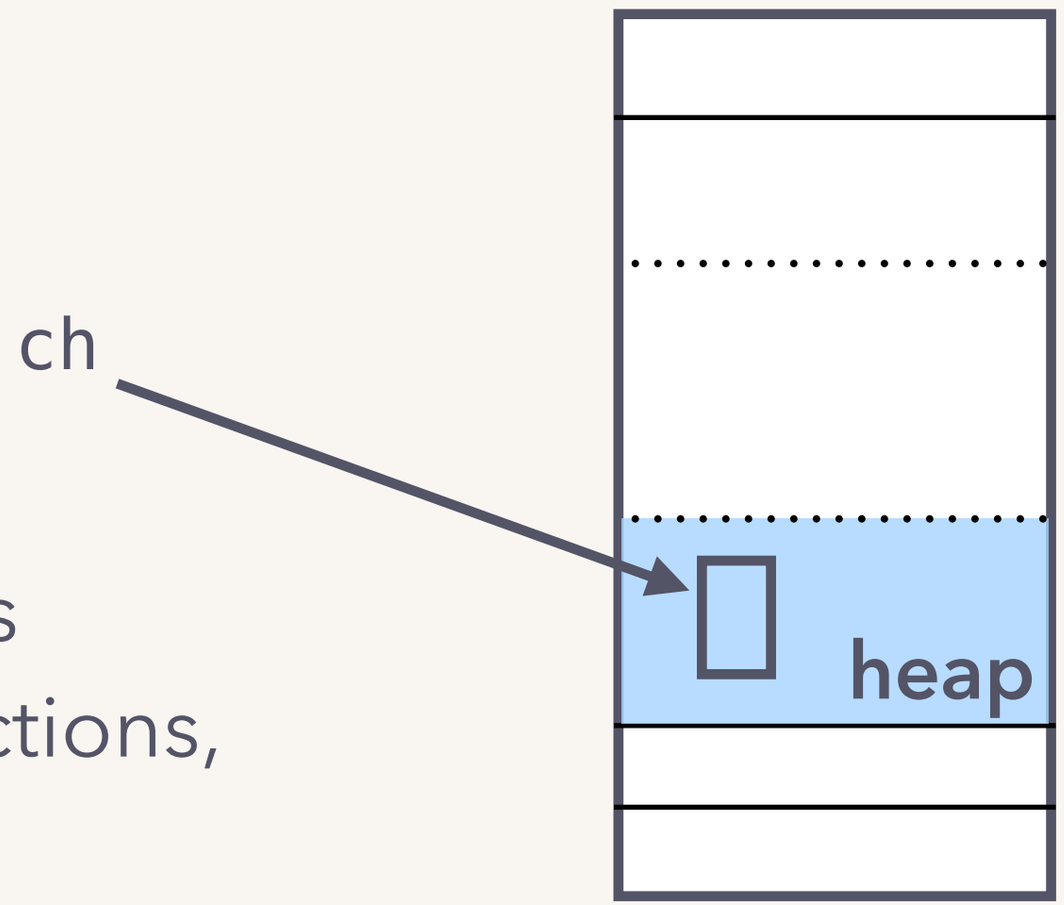


make chan

```
ch := make(chan Task, 3)
```

allocates an hchan struct **on the heap**.
initializes it.
returns a pointer to it.

This is why we can pass
channels between functions,
don't need to pass
pointers to channels.



sends and receives

task queue

G1

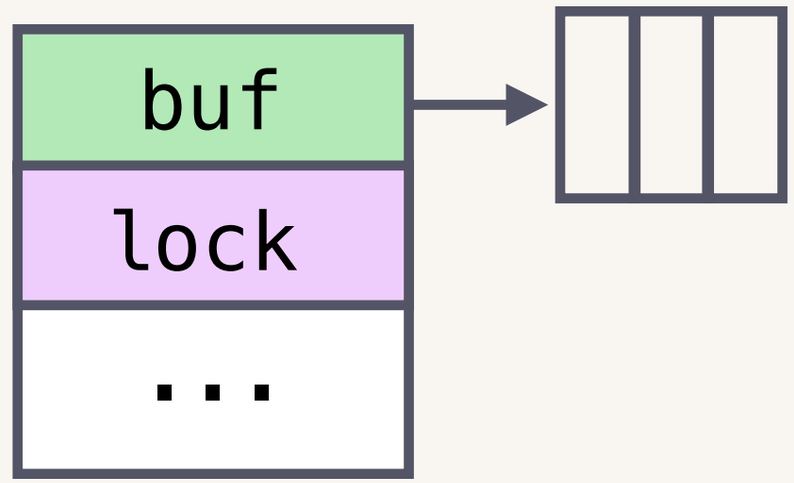
```
func main() {
   ...

   for _, task := range tasks {
      ch <- task
   }

   ...
}
```

G2

```
func worker() {
   for {
      task := <-ch
      process(task)
   }
}
```

G1

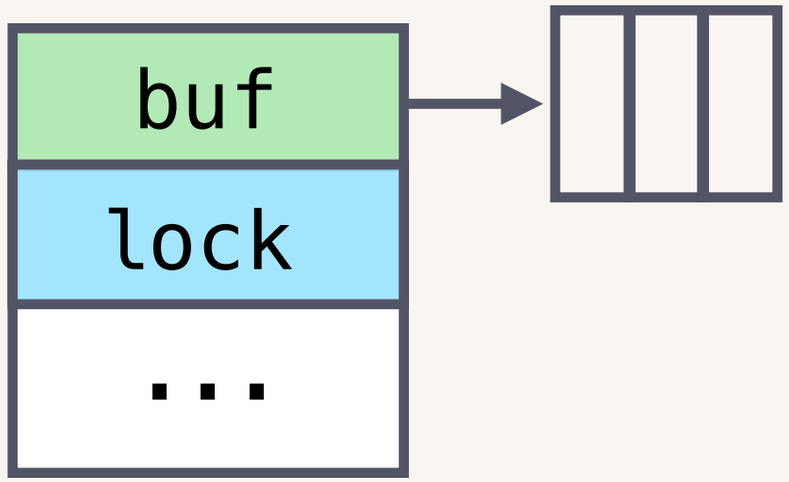
```
ch <- task₀
```



G1

```
ch <- task₀
```

1. acquire

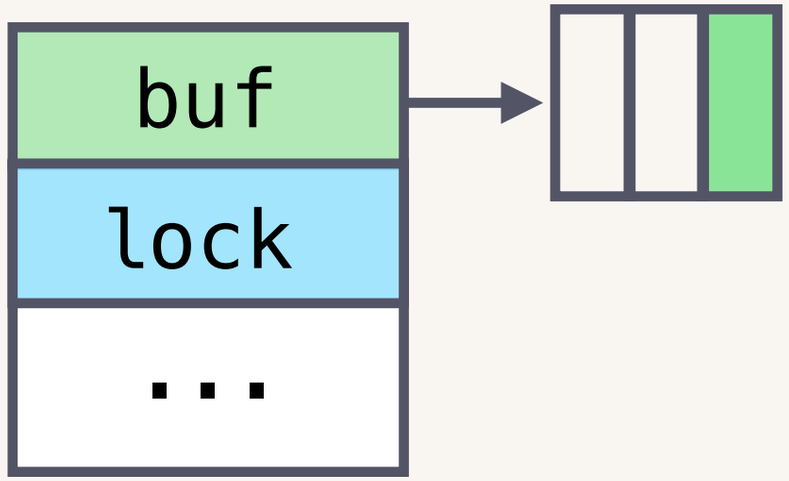


G1

`ch <- task₀`

2. enqueue

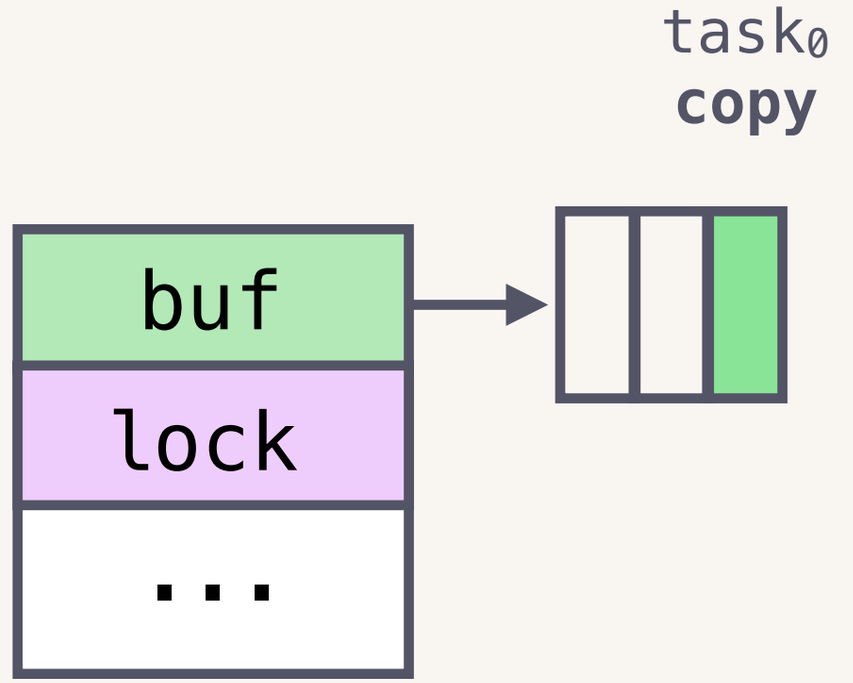
`task₀`
copy



G1

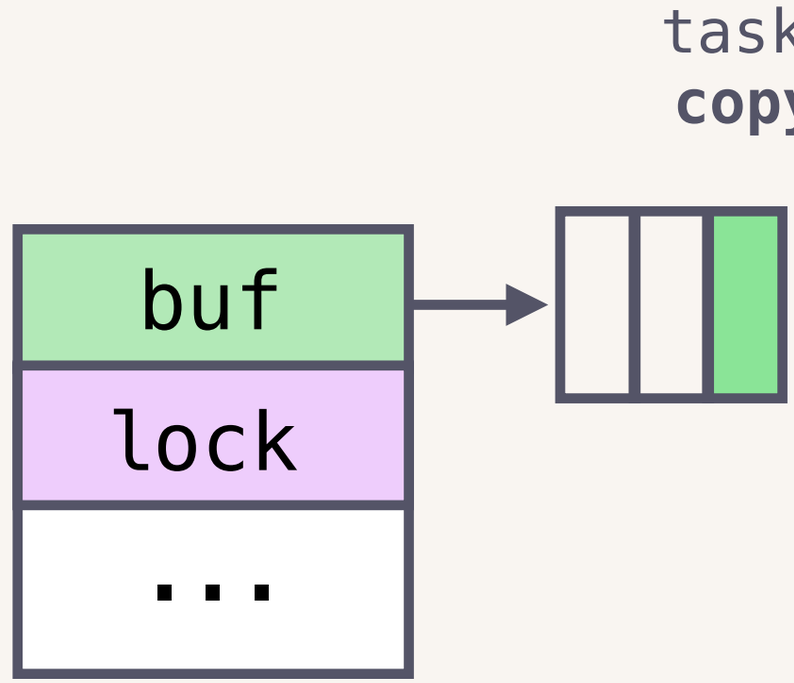
`ch <- task0`

3. `release`



G2

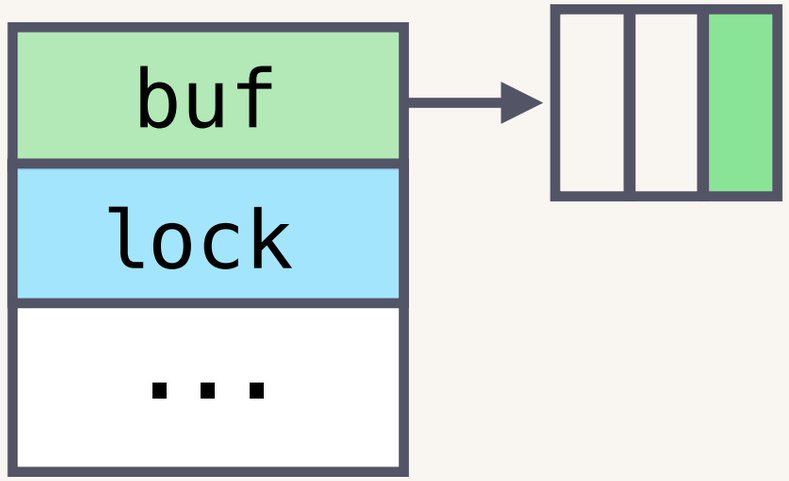
t := <-ch



G2

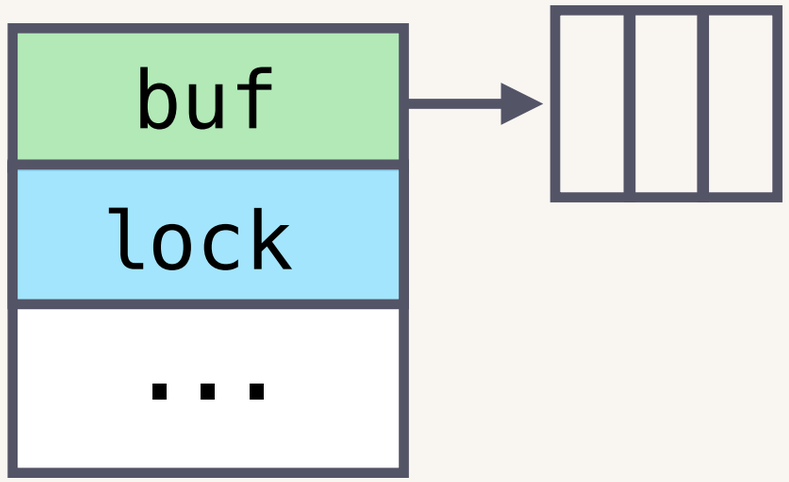
task0
copy

t := <-ch



1. acquire

2. dequeue



G2

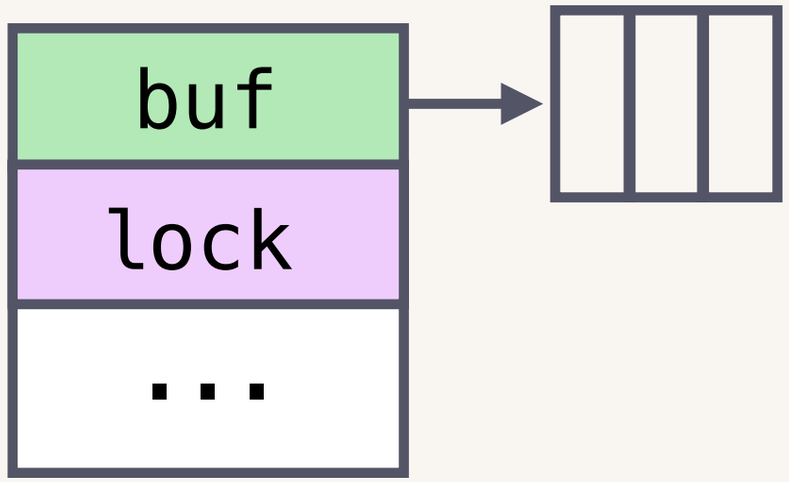
t := <-ch

task₀
copy'

G2

t := <-ch

3. release



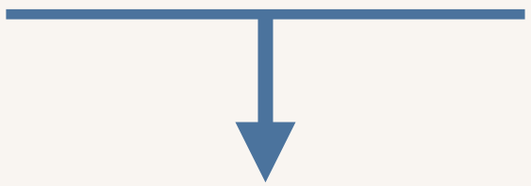
task₀
copy’

no shared memory
(except hchan)

copies

no shared memory
(except hchan)

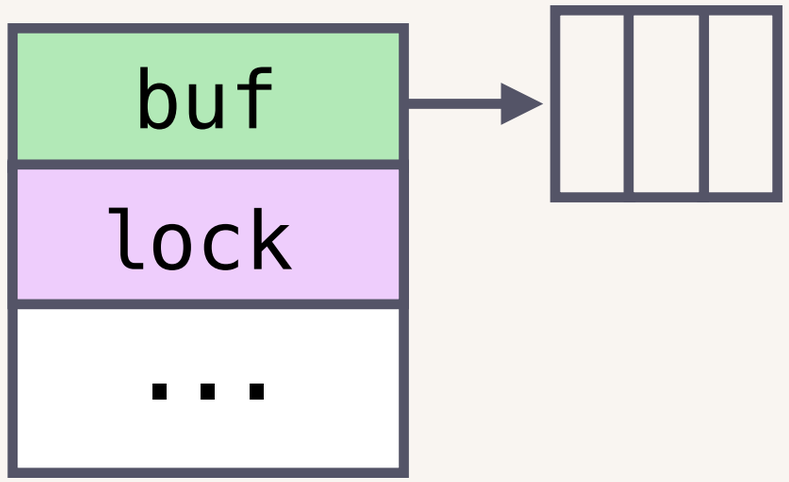
"Do not communicate by sharing memory;
instead, share memory by communicating."



copies

G1

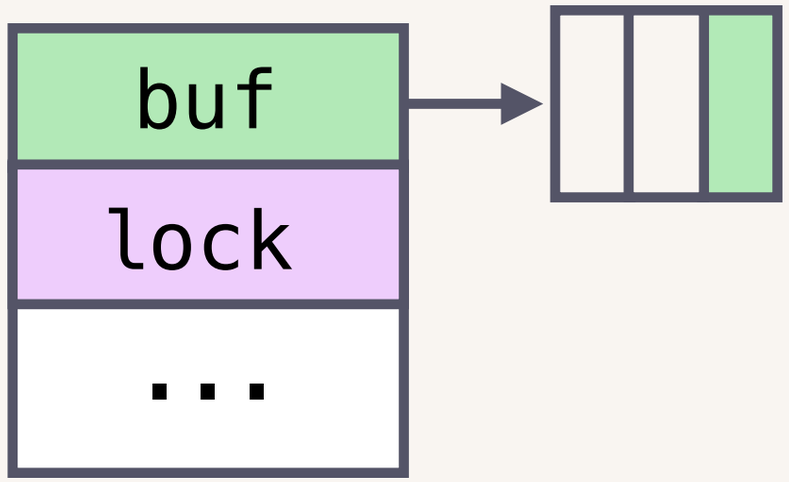
G2



G1

G2

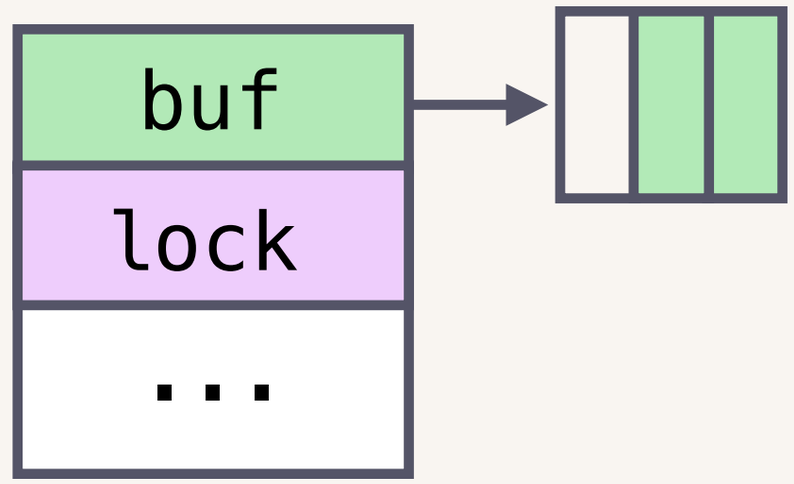
```
ch <- task1
```



G1

G2

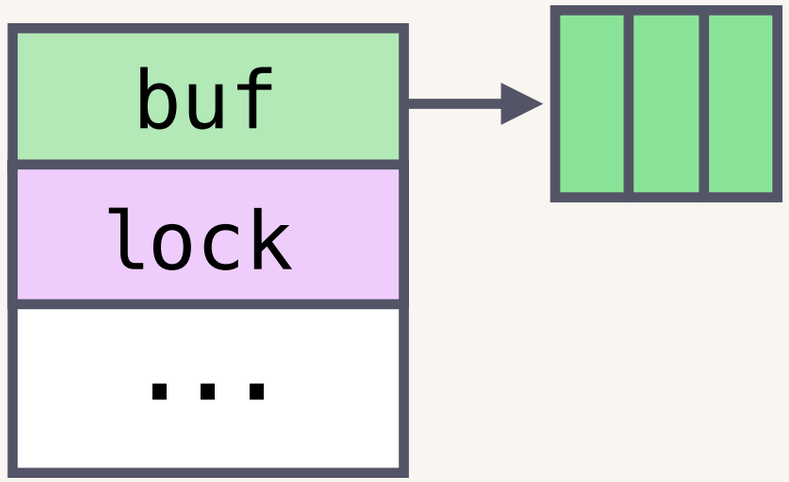
```
ch <- task1
ch <- task2
```



G1

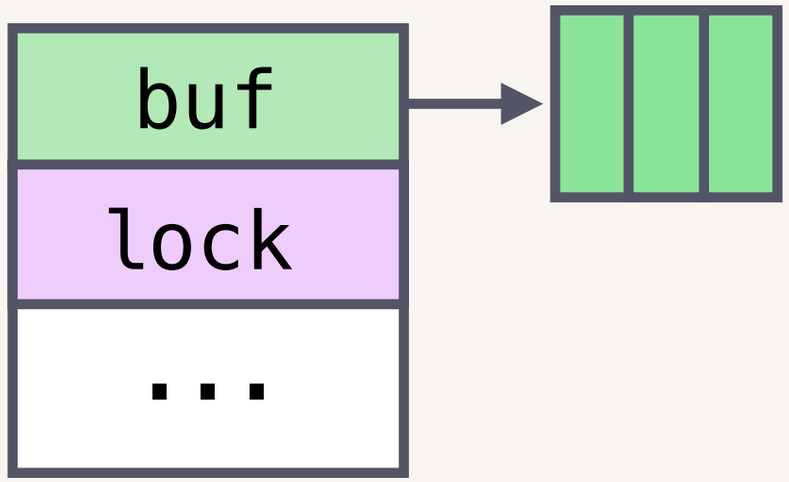
G2

```
ch <- task1
ch <- task2
ch <- task3
```



G1

```
ch <- task1
ch <- task2
ch <- task3
```

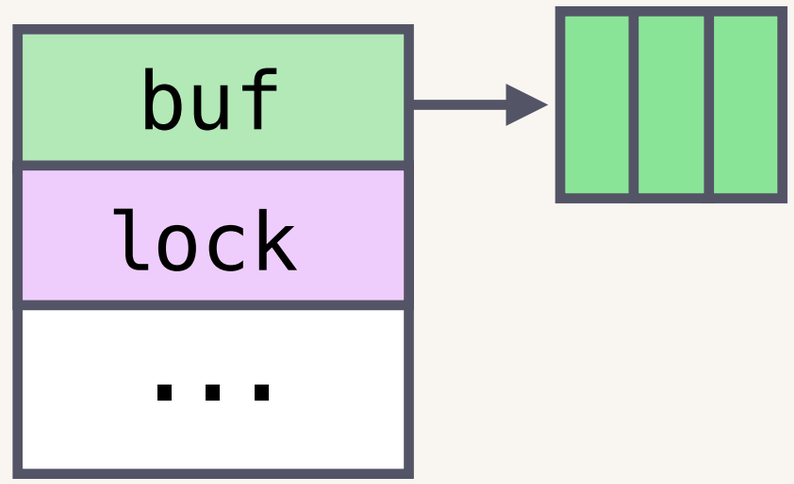


G2

```
ch <- task4
```

G1

```
ch <- task1
ch <- task2
ch <- task3
```



G2

```
ch <- task4
```

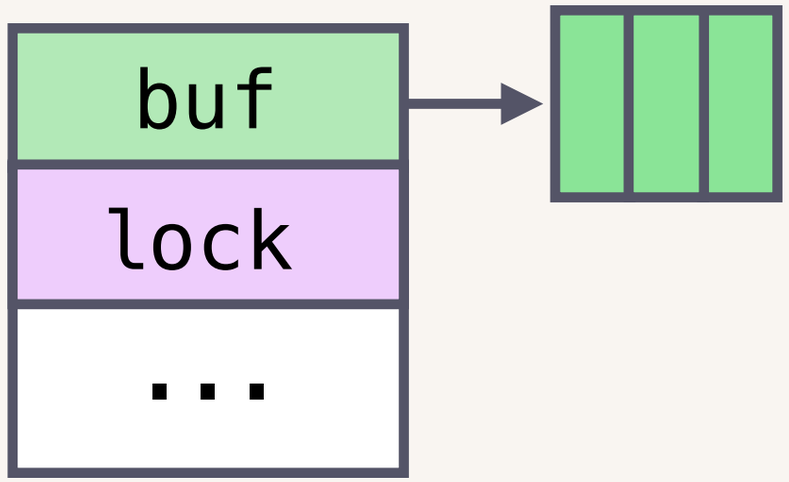
ruh-roh, channel is **full**!

G1's execution is **paused,**

resumed after a receive.

G1

```
ch <- task1
ch <- task2
ch <- task3
```



G2

```
ch <- task4
```

ruh-roh, channel is **full**!
G1's execution is **paused,**
resumed after a receive.

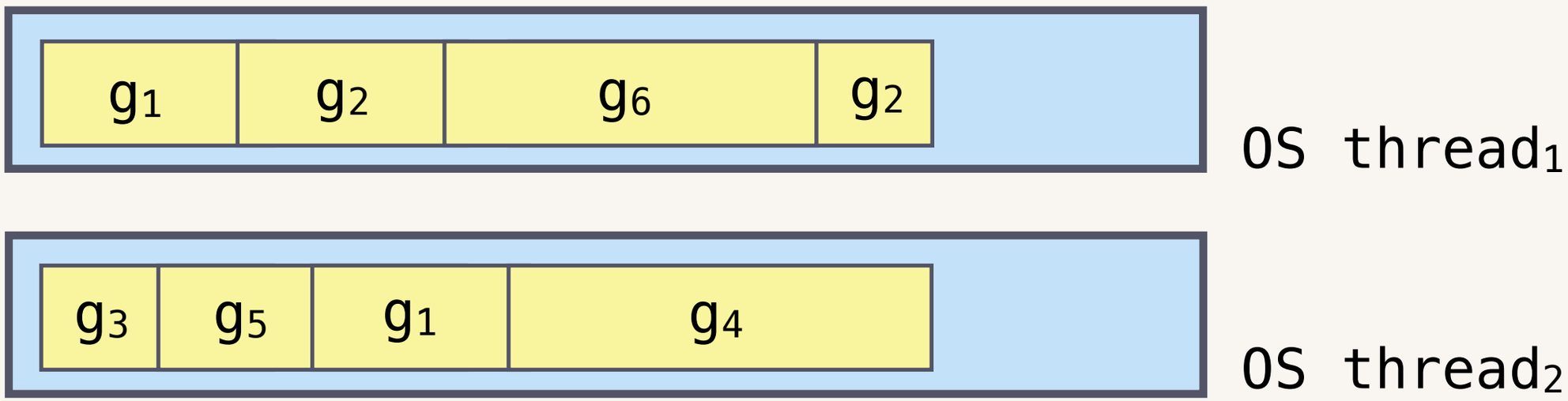
?

interlude: the runtime scheduler

goroutines are user-space threads.

created and managed by the Go runtime, not the OS.
lightweight compared to OS threads.

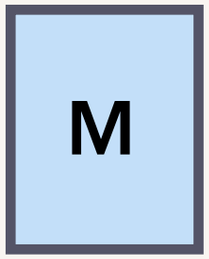
the runtime scheduler schedules them onto OS threads.



M:N scheduling

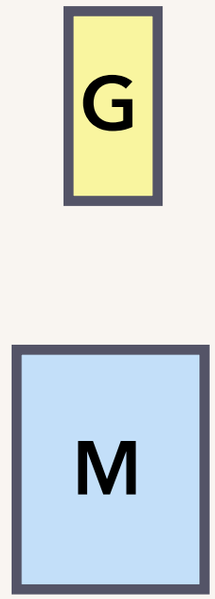
Go’s M:N scheduling is described using three structures:

M: OS thread



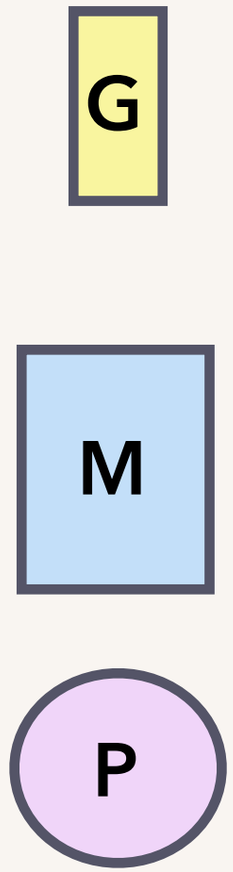
Go’s M:N scheduling is described using three structures:

M: OS thread
G: goroutine



Go’s M:N scheduling is described using three structures:

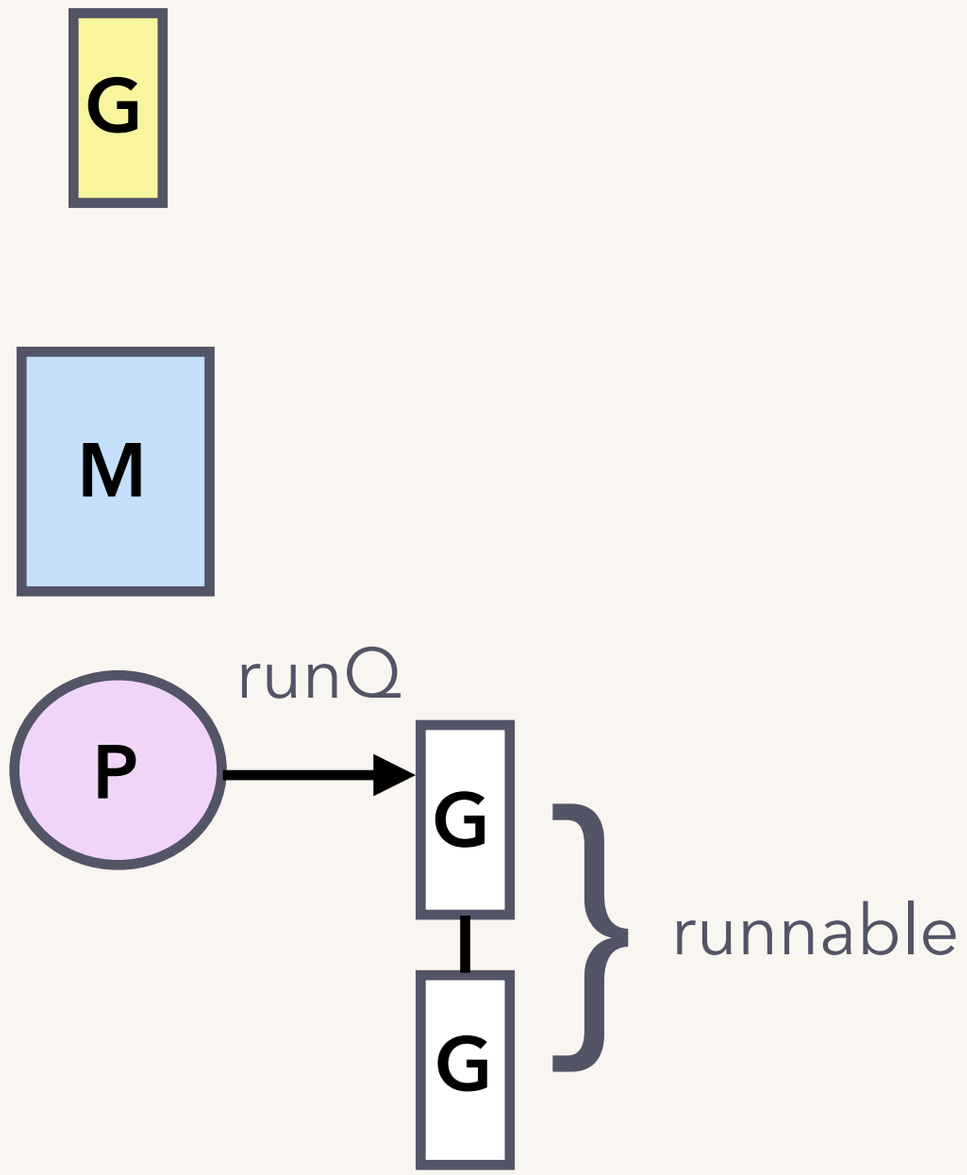
M: OS thread
G: goroutine
P: context for scheduling.



Go's M:N scheduling is described using three structures:

M: OS thread
G: goroutine
P: context for scheduling.

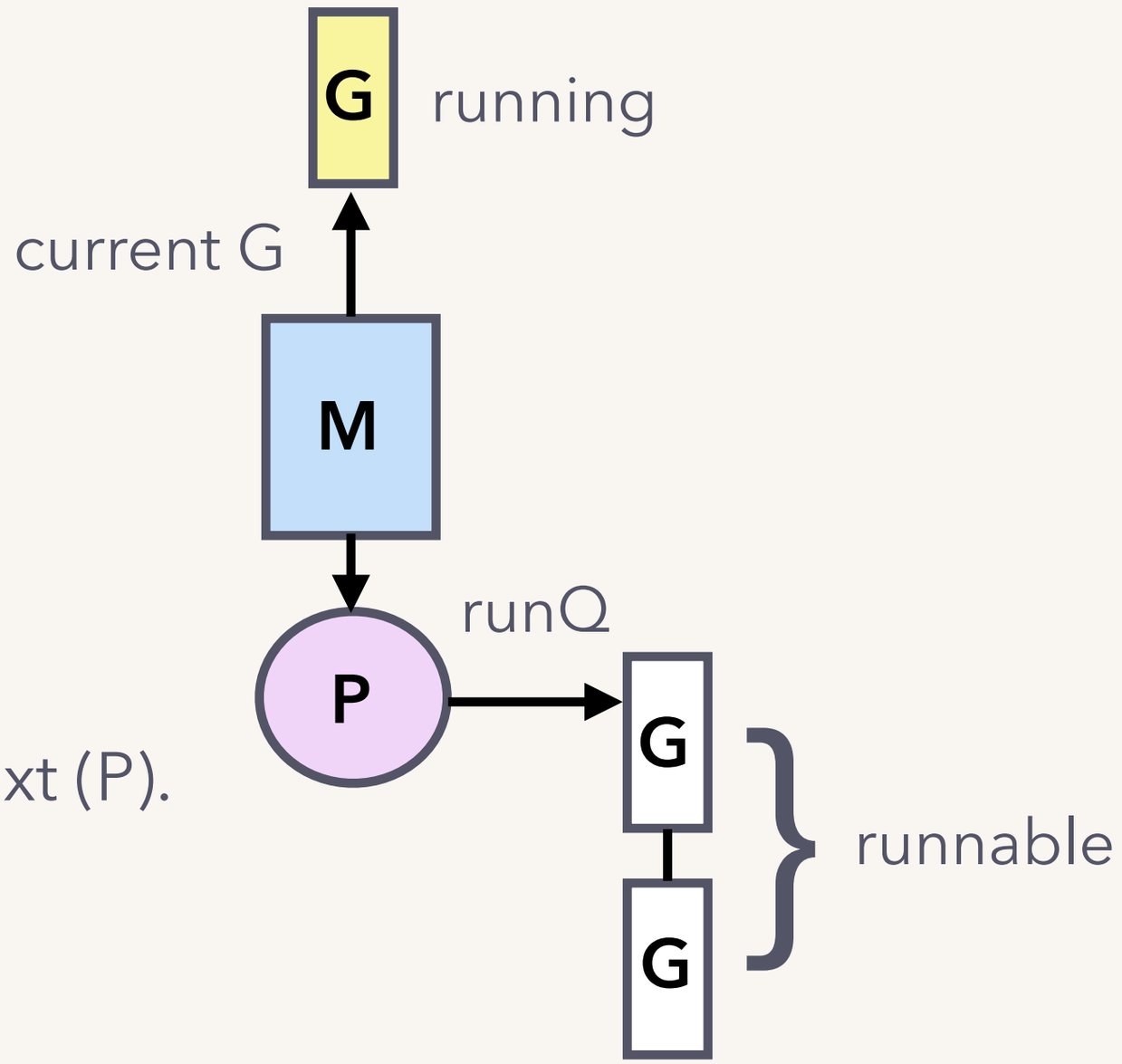
- Ps hold the runqueues.



Go's M:N scheduling is described using three structures:

M: OS thread
G: goroutine
P: context for scheduling.

- Ps hold the runqueues.
- In order to run goroutines (G),
  a thread (M) must hold a context (P).



pausing goroutines

```
ch <- task4
```

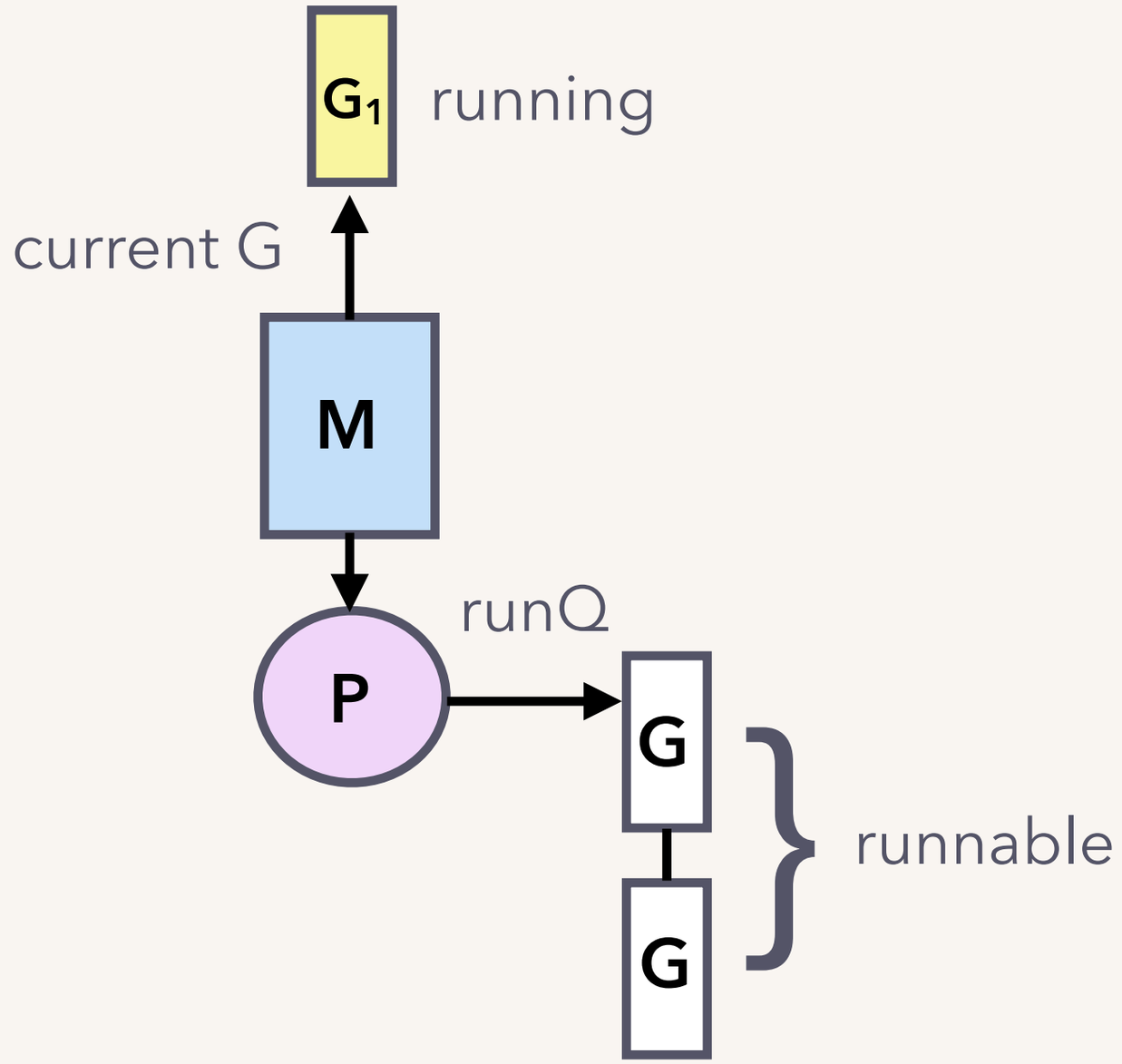
} send on a full channel

G1

`ch <- task4`

`gopark`

calls into the scheduler



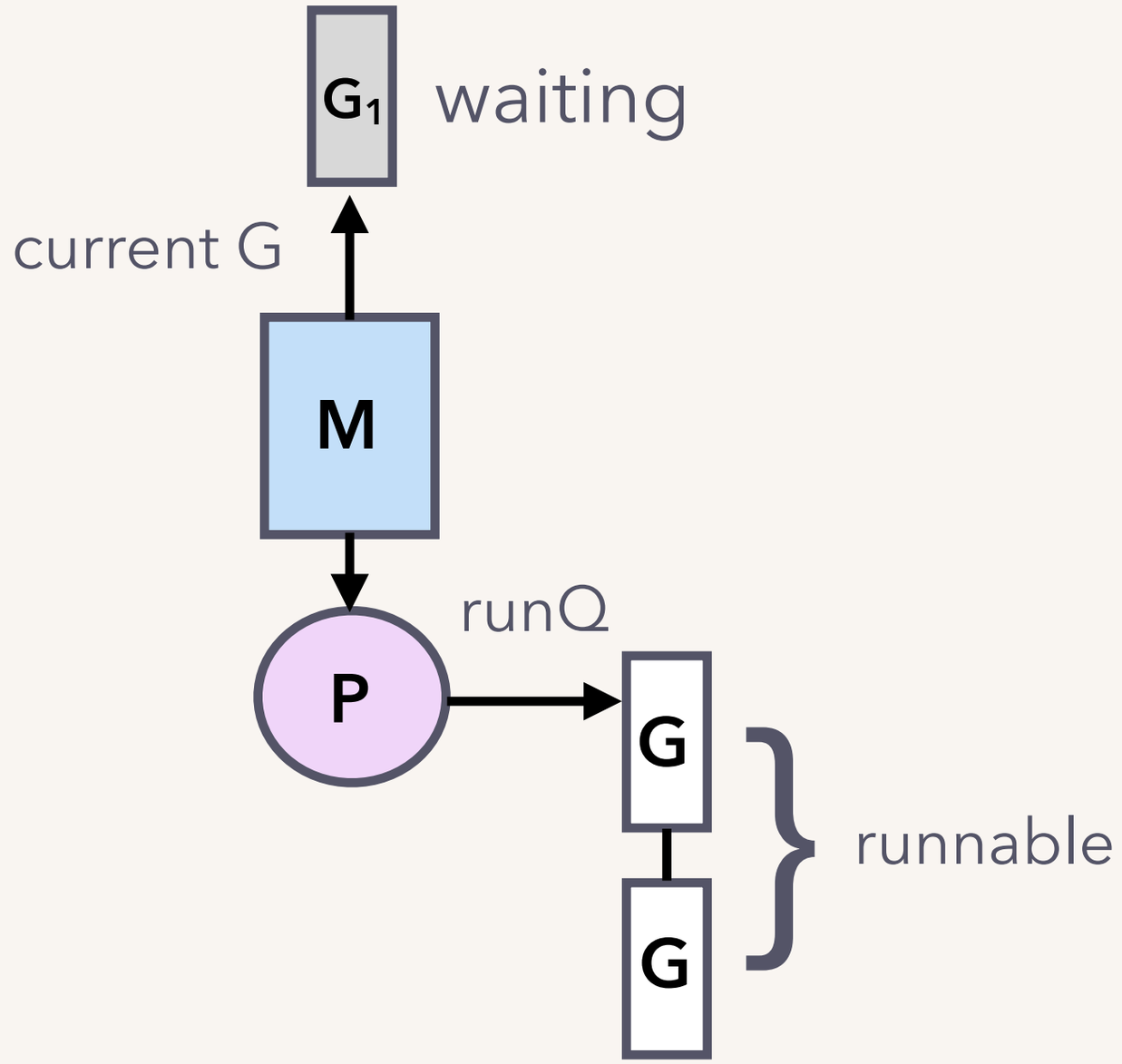
G1

`ch <- task4`

gopark

calls into the scheduler

sets `G1` to waiting



G1

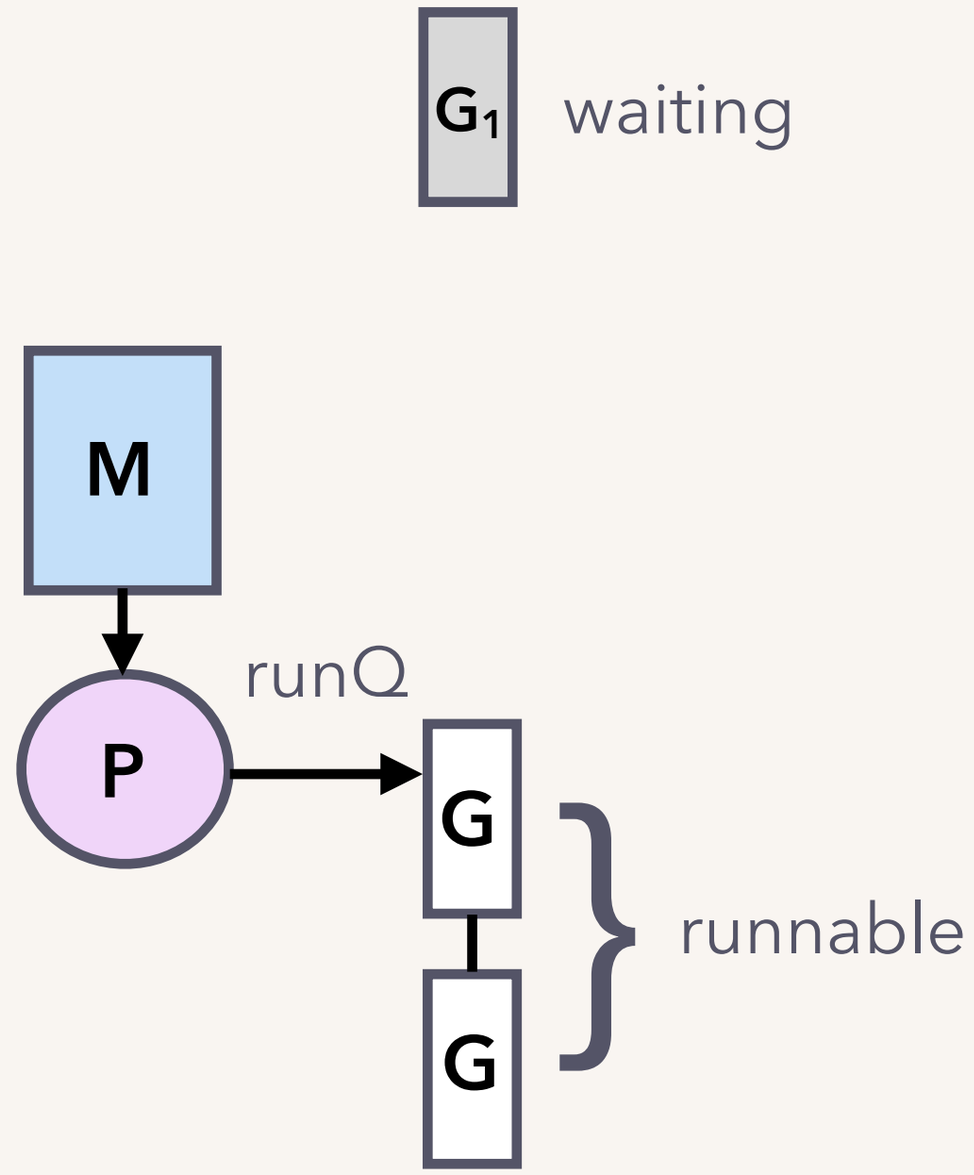
`ch <- task4`

`gopark`

calls into the scheduler

sets `G1` to waiting

removes association
between `G1, M`



G1

`ch <- task4`

gopark

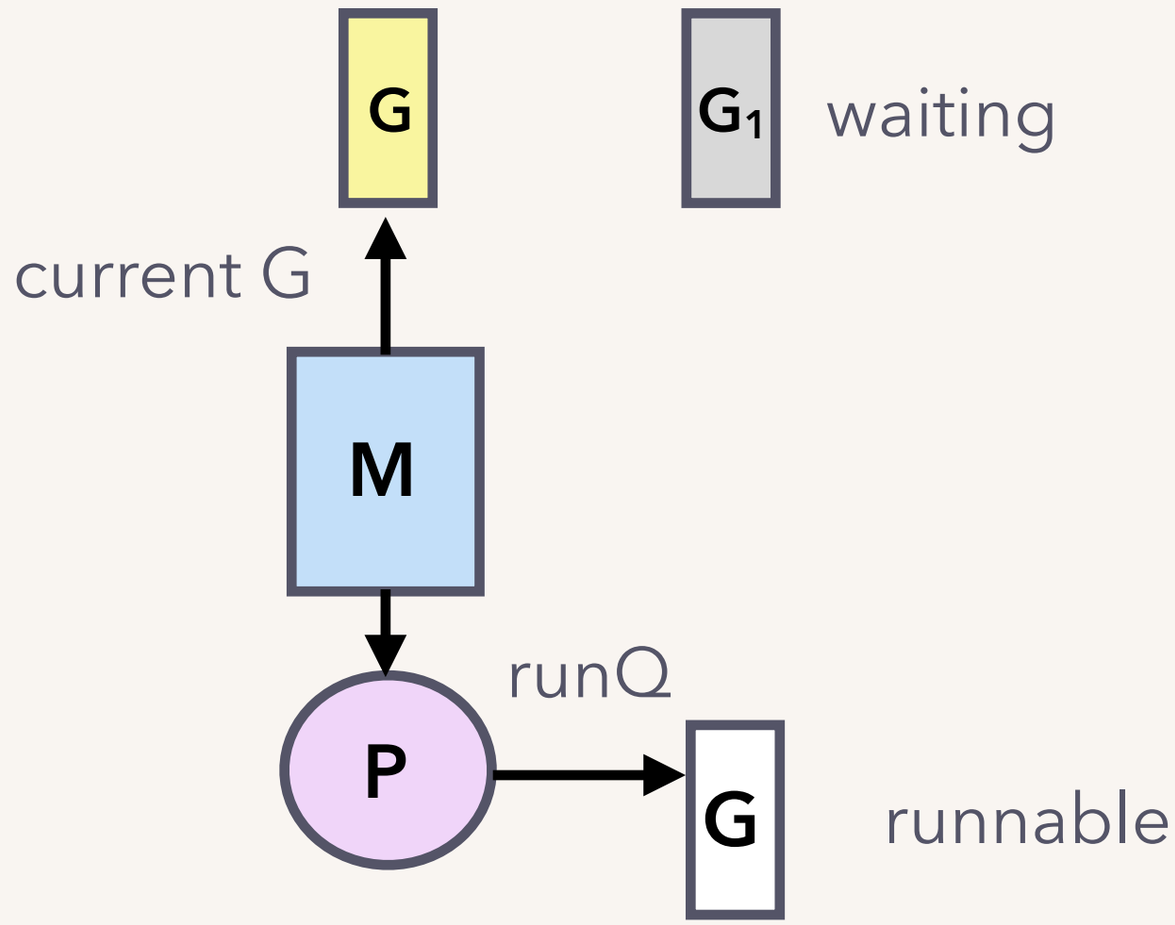
calls into the scheduler

sets `G1` to waiting

removes association between `G1, M`

schedules a runnable `G` from the runqueue

} "returns" to a different G



This is neat.
G1 is blocked as needed, but not the OS thread.

This is neat.
G1 is blocked as needed, but not the OS thread.

…great, but how do we **resume** the blocked goroutine?

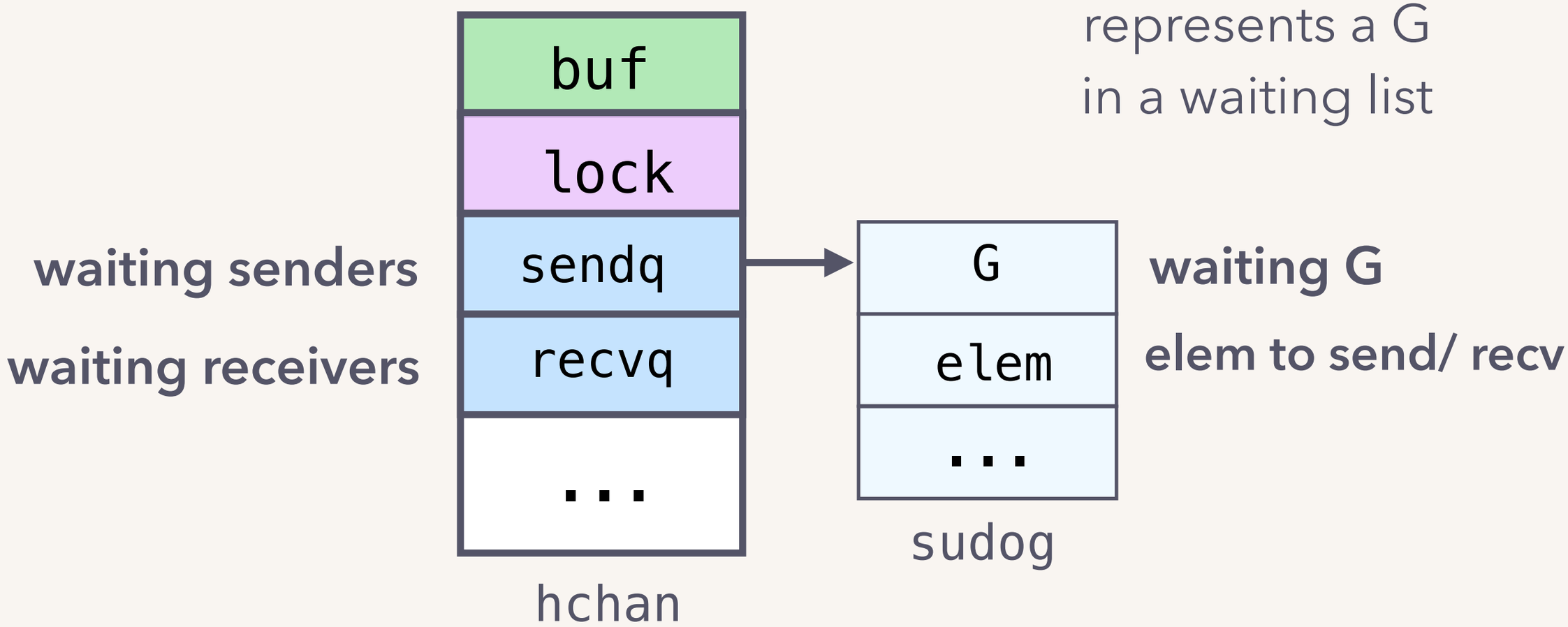
after a channel receive, and the channel is no longer full

resuming goroutines

the hchan struct stores waiting senders, receivers as well.

stored as a sudog

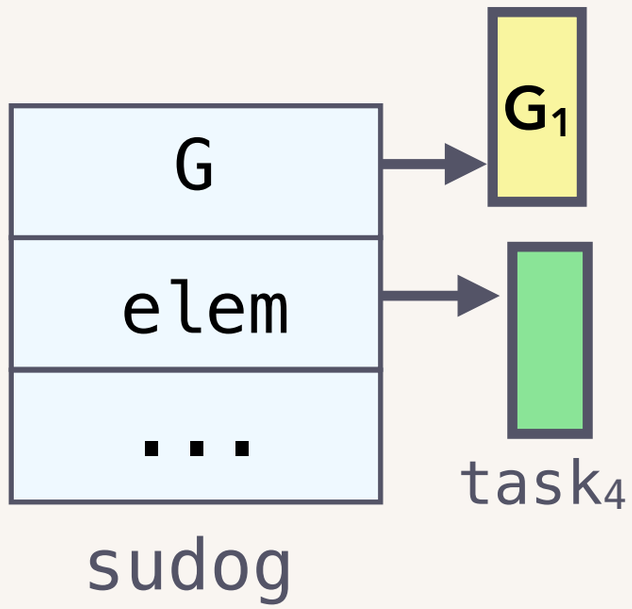
represents a G
in a waiting list



G1

`ch <- task4`

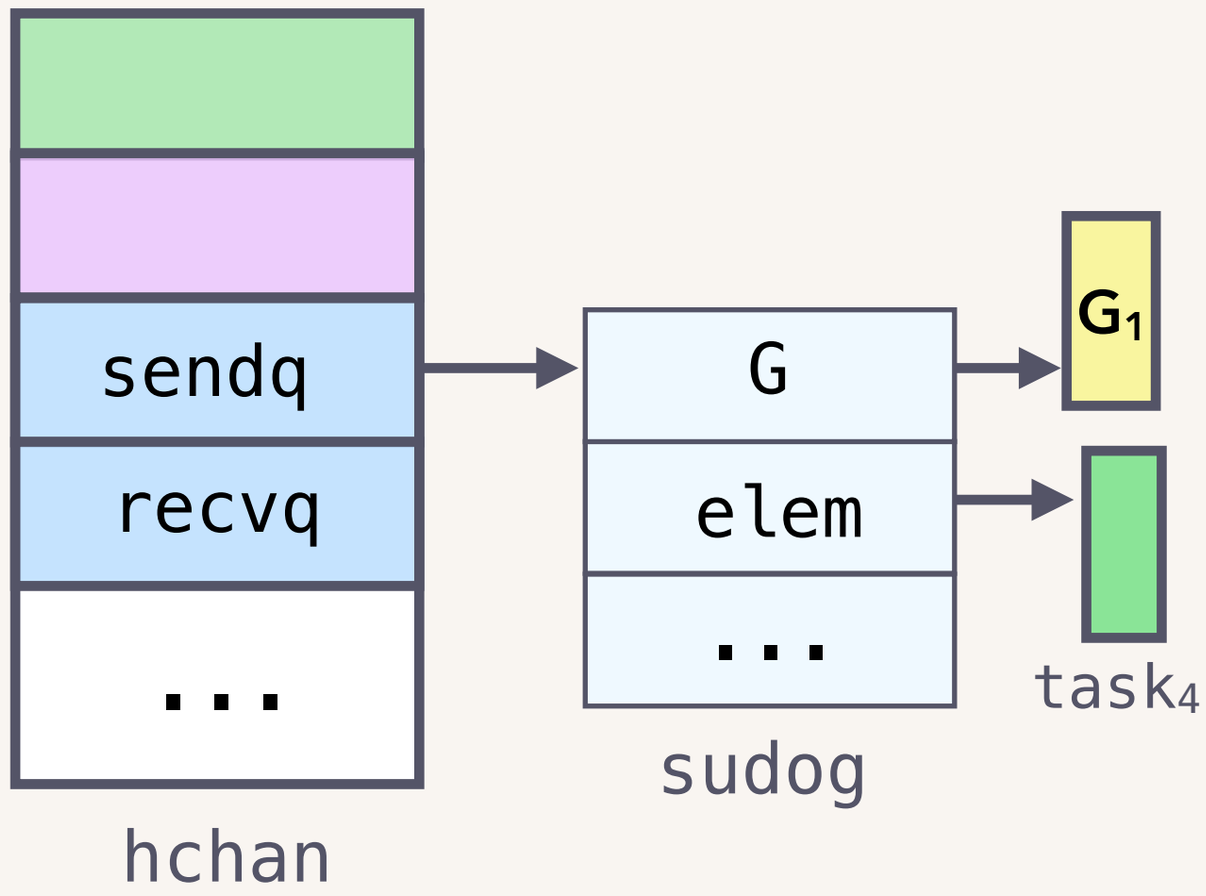
creates a **sudog** for itself



G1

`ch <- task4`

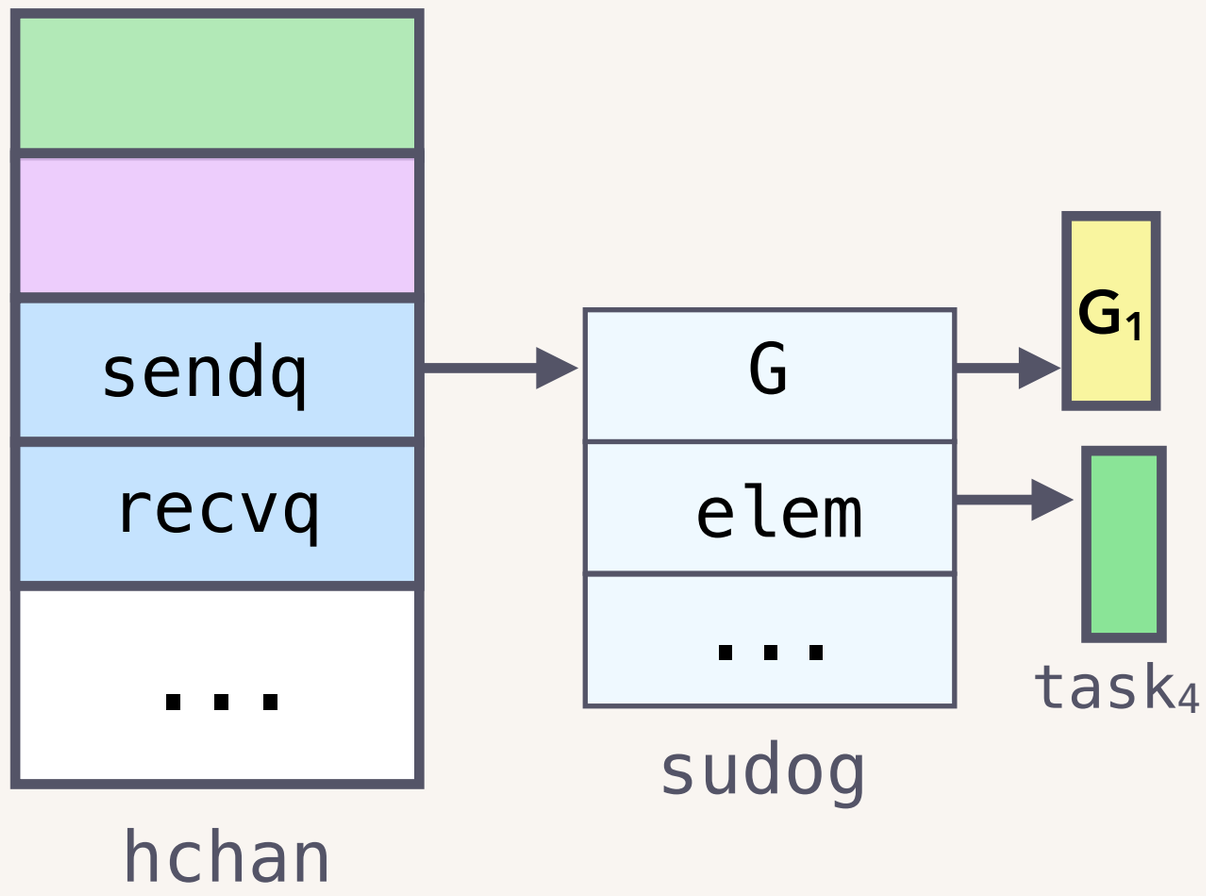
creates a **sudog** for itself, puts it in the **sendq** → **receiver uses it** to **resume** G1.



G1

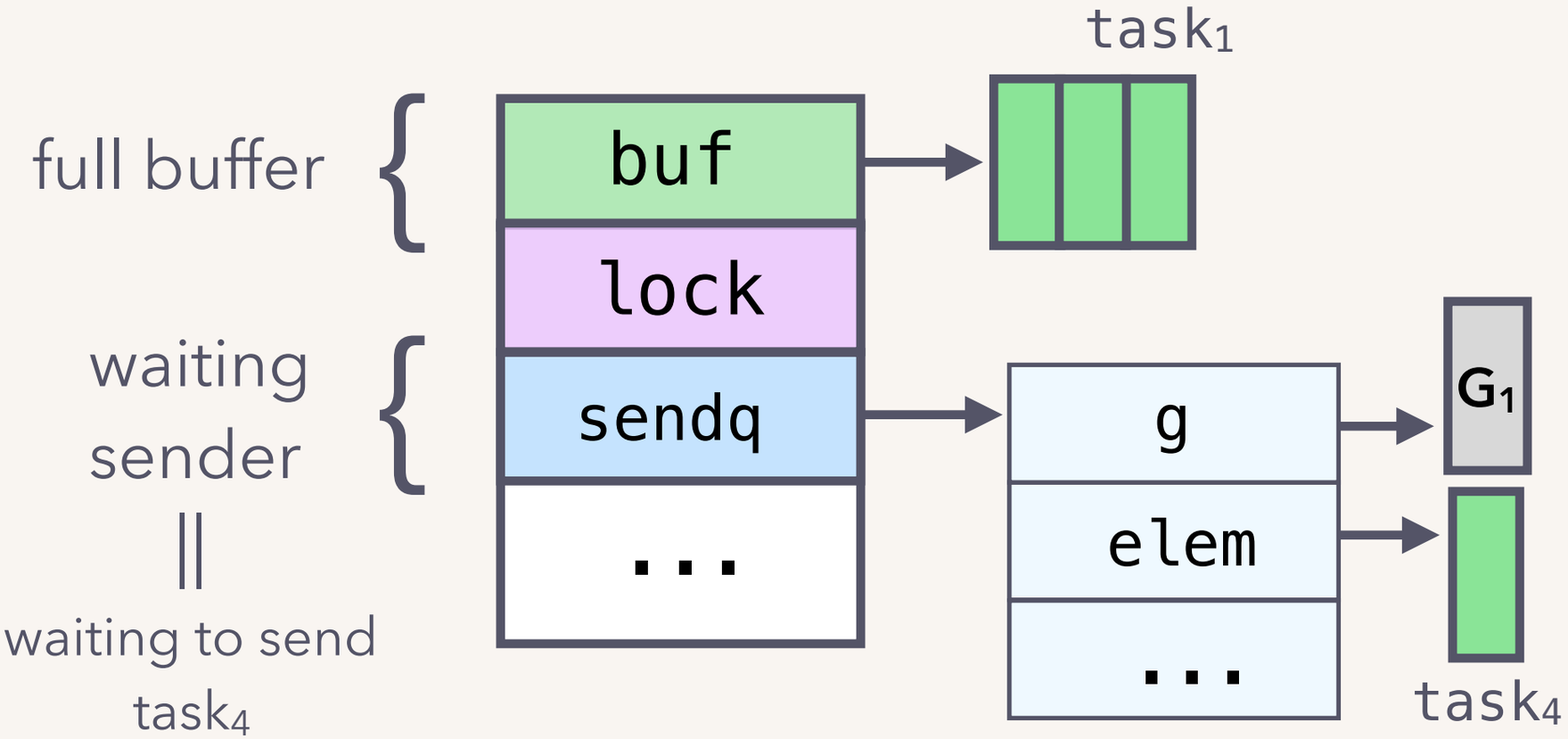
`ch <- task4`

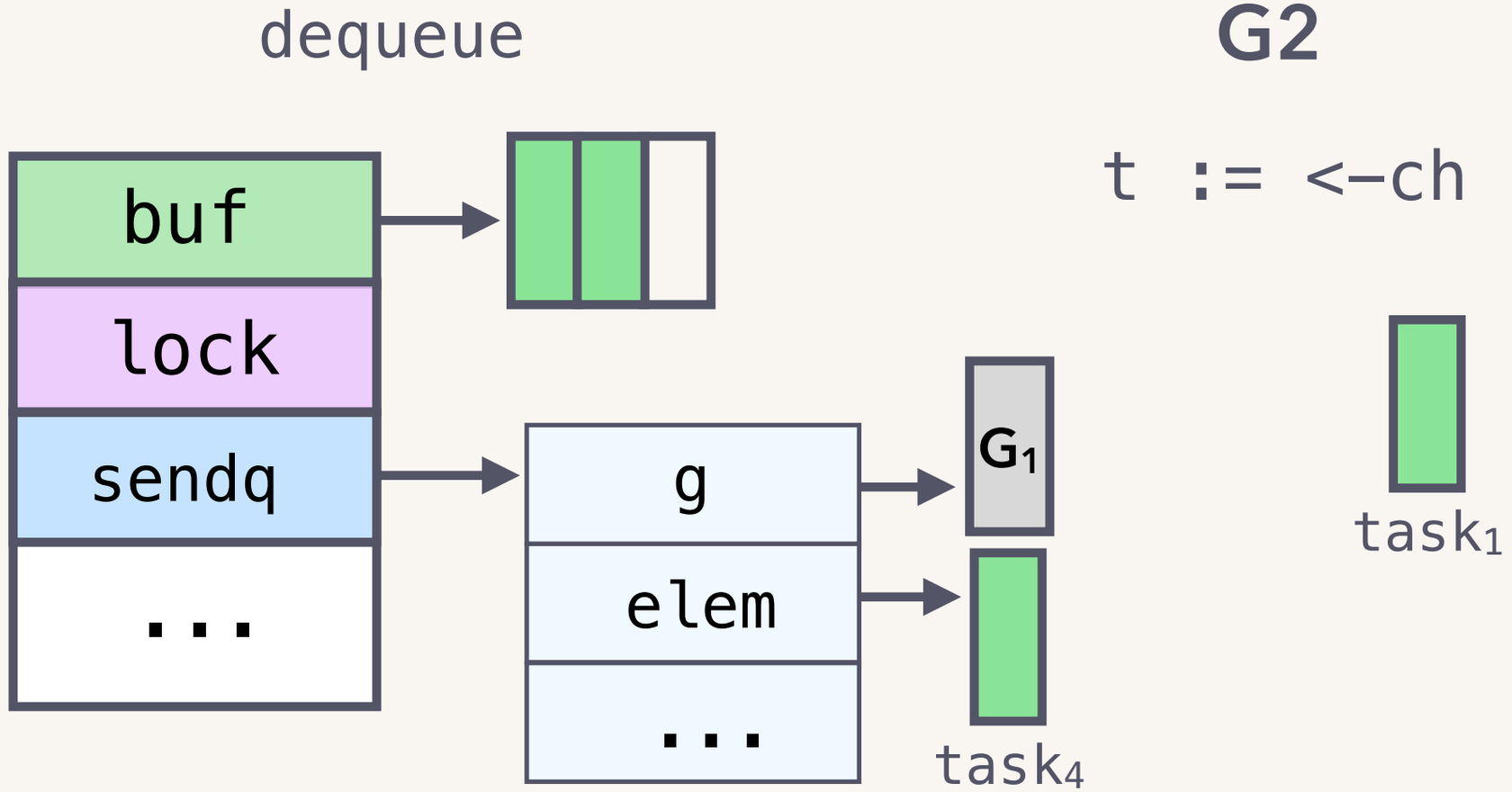
creates a **sudog** for itself, puts it in the **sendq** → **receiver uses it** to **resume** G1.



G2

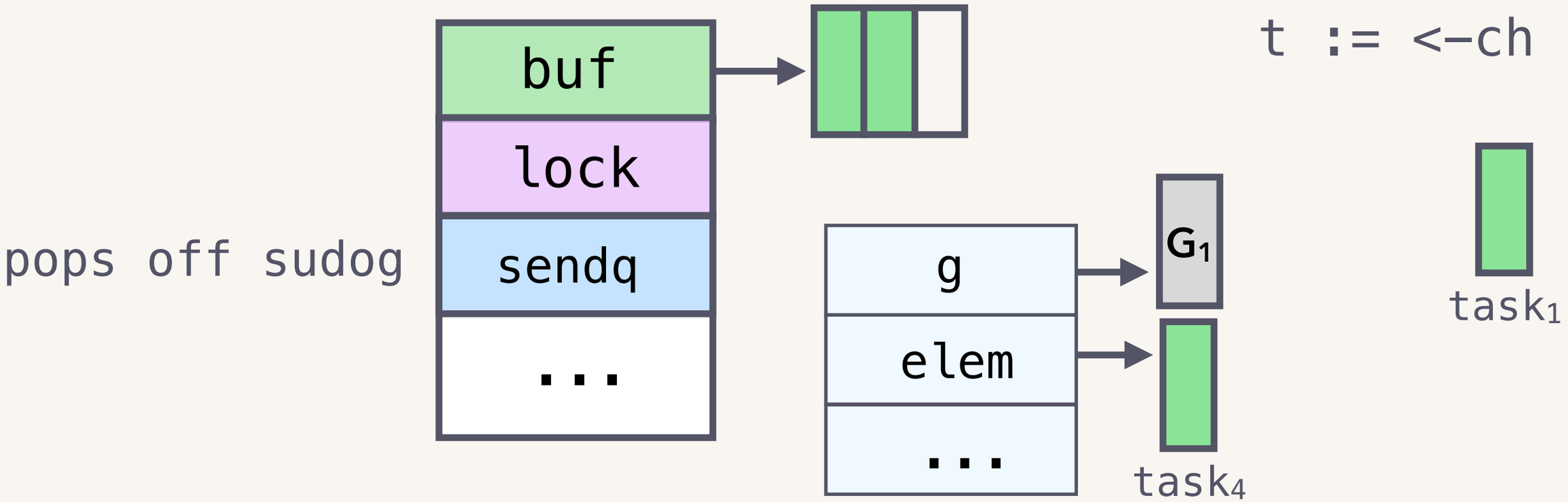
```
t := <-ch
```


task1
full buffer
buf
lock
waiting sender
sendq
...
g
elem
...
G1
task4
waiting to send task4

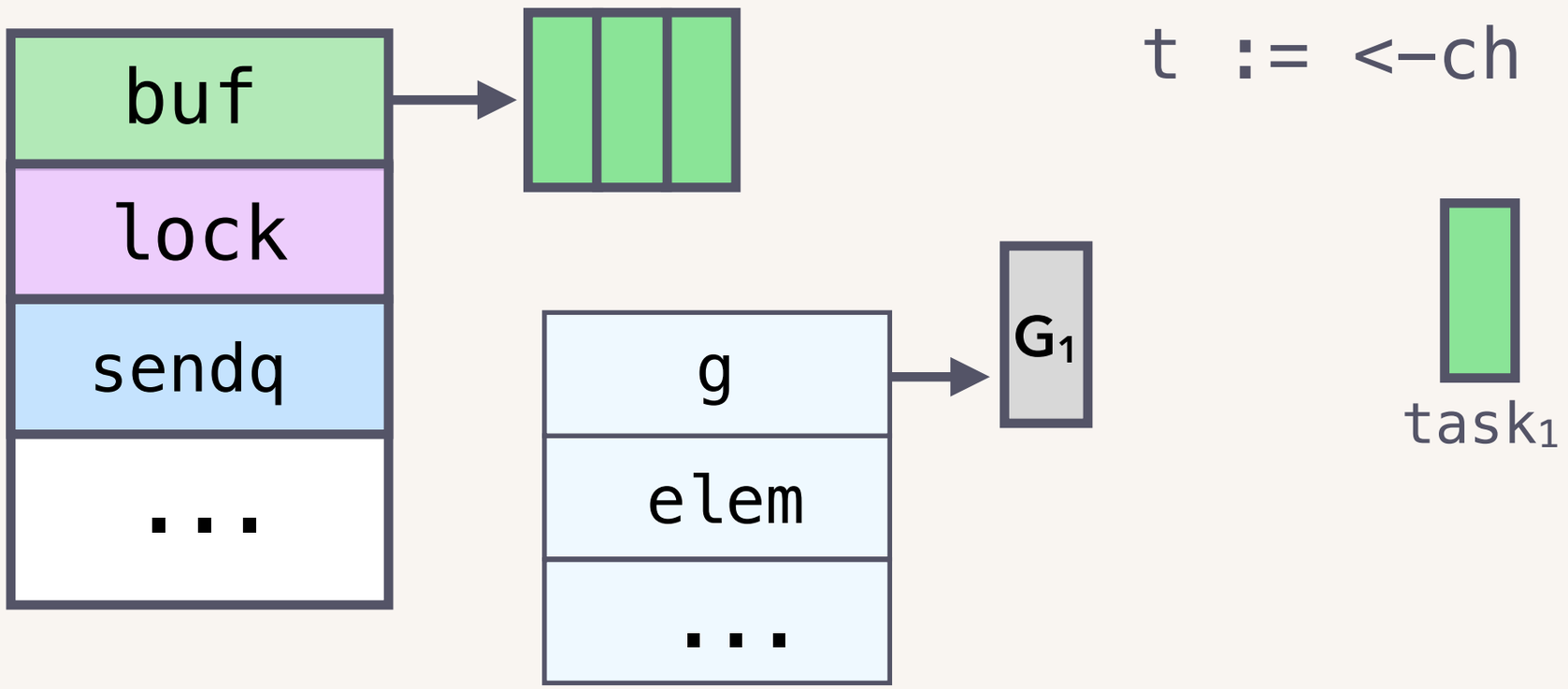

dequeue
G2
t := <-ch
buf
lock
sendq
...
g
elem
...
G1
task4
task1

G2

t := <-ch

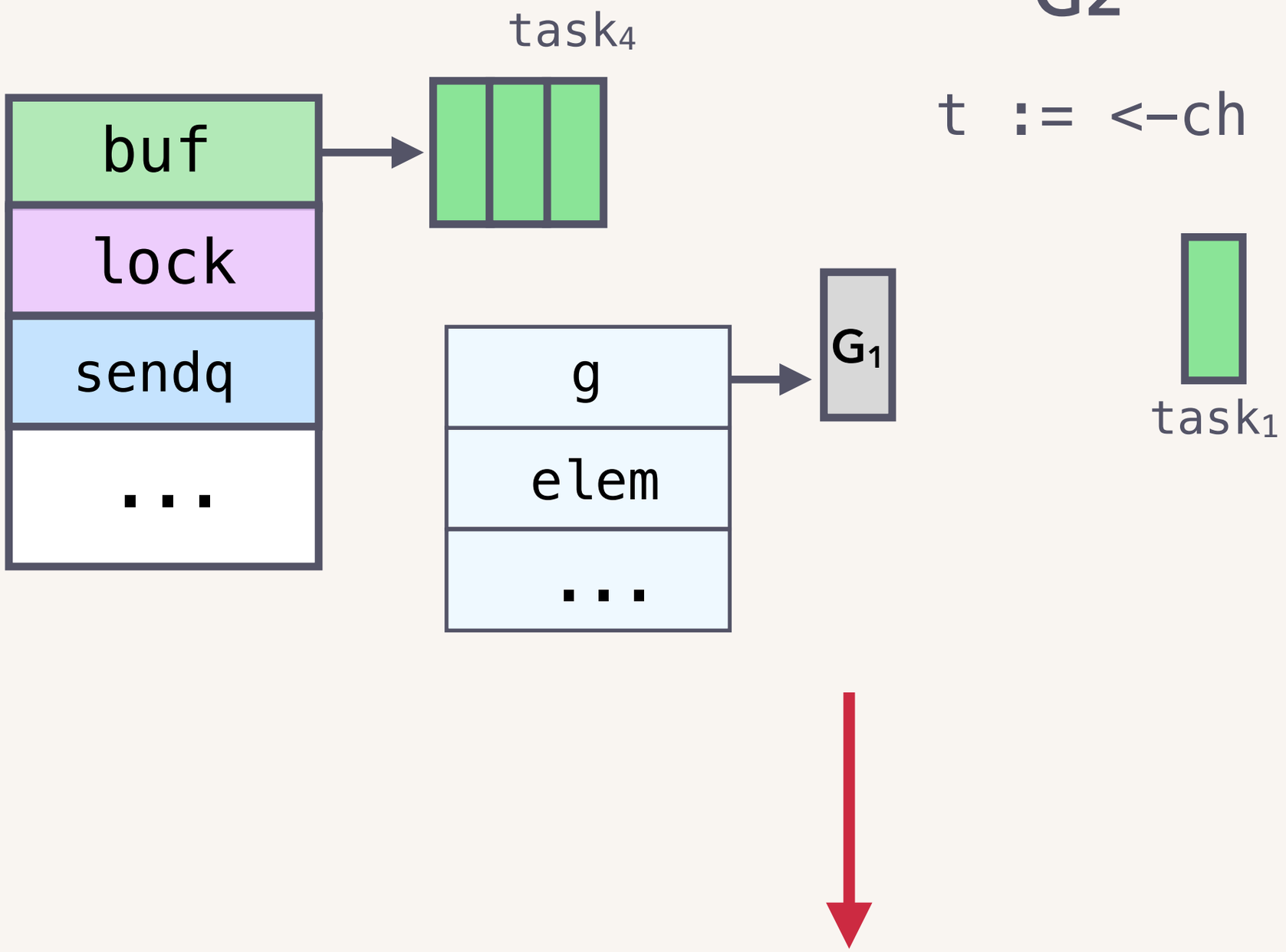


enqueues the sudog's elem:



G2

t := <-ch



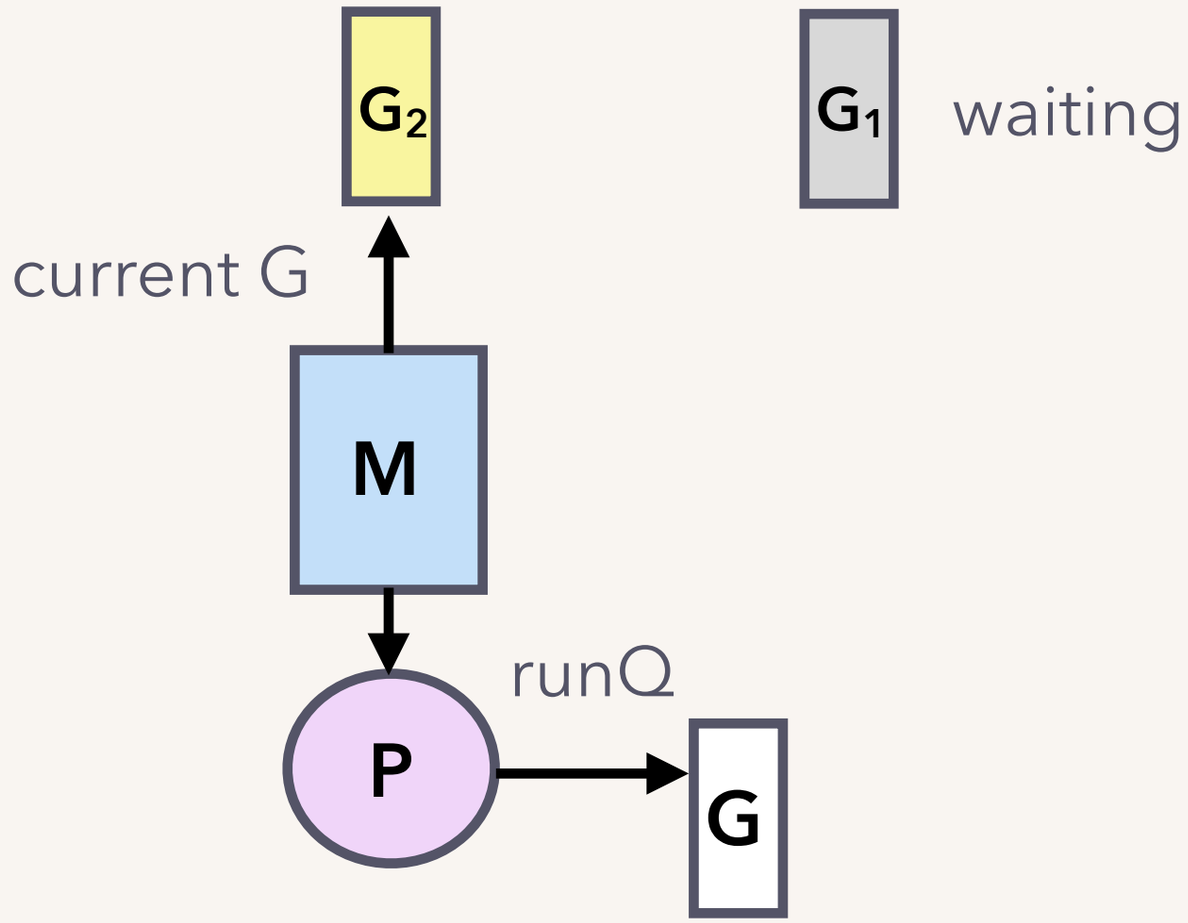
need to set G1 to runnable

G2

```
t := <-ch
```

goready (G1)

calls into the scheduler



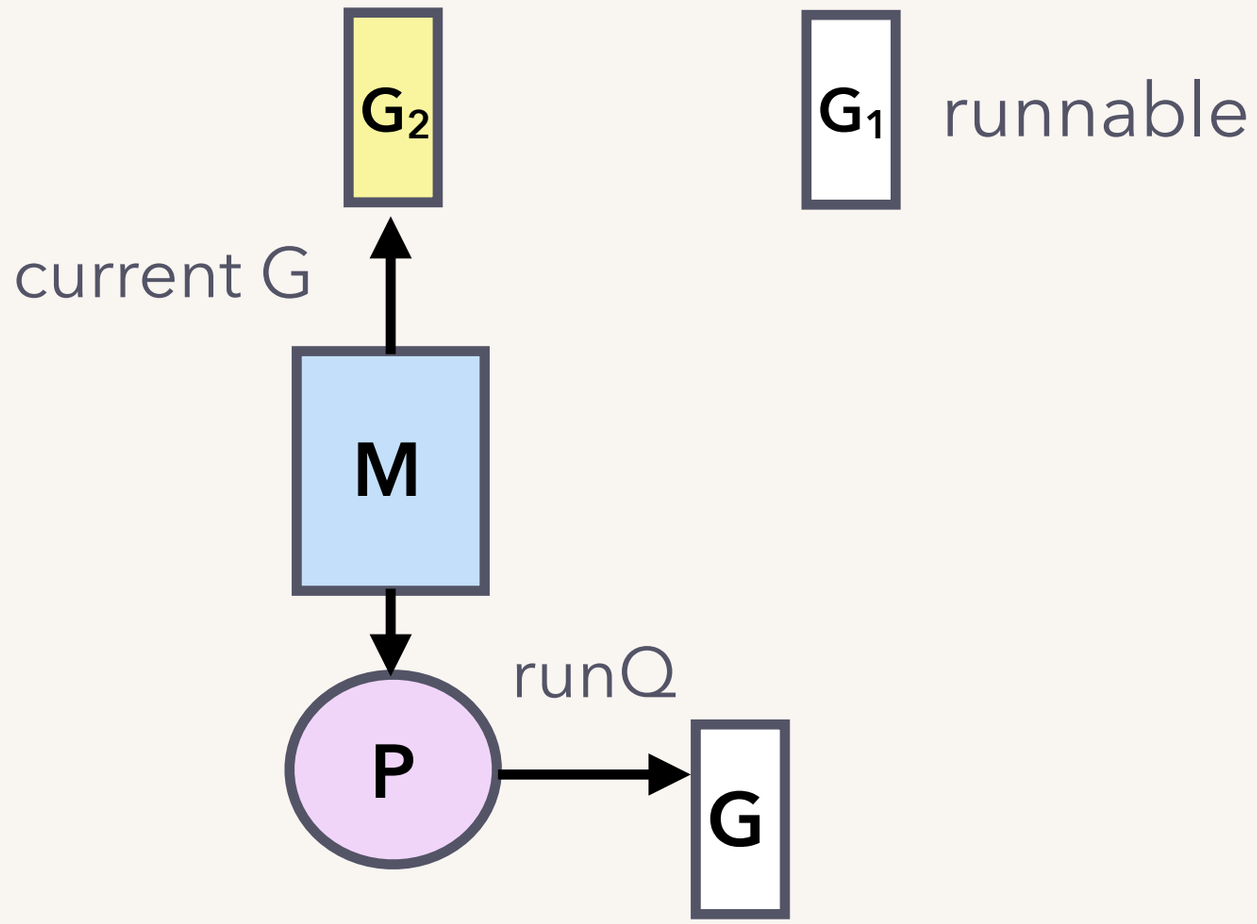
G2

```
t := <-ch
```

goready (G1)

calls into the scheduler

sets G1 to runnable



G2

```
t := <-ch
```

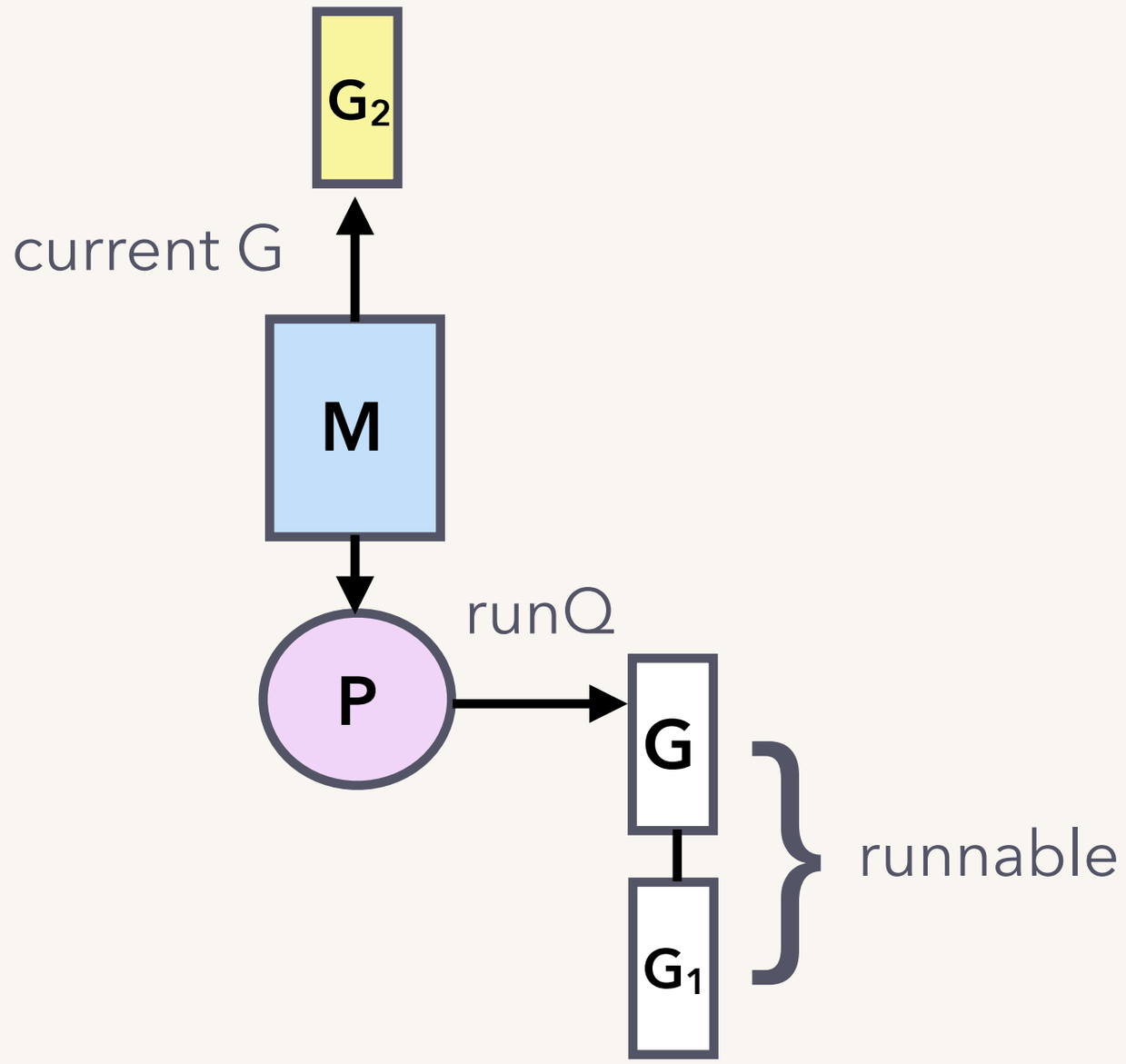
goready (G1)

calls into the scheduler

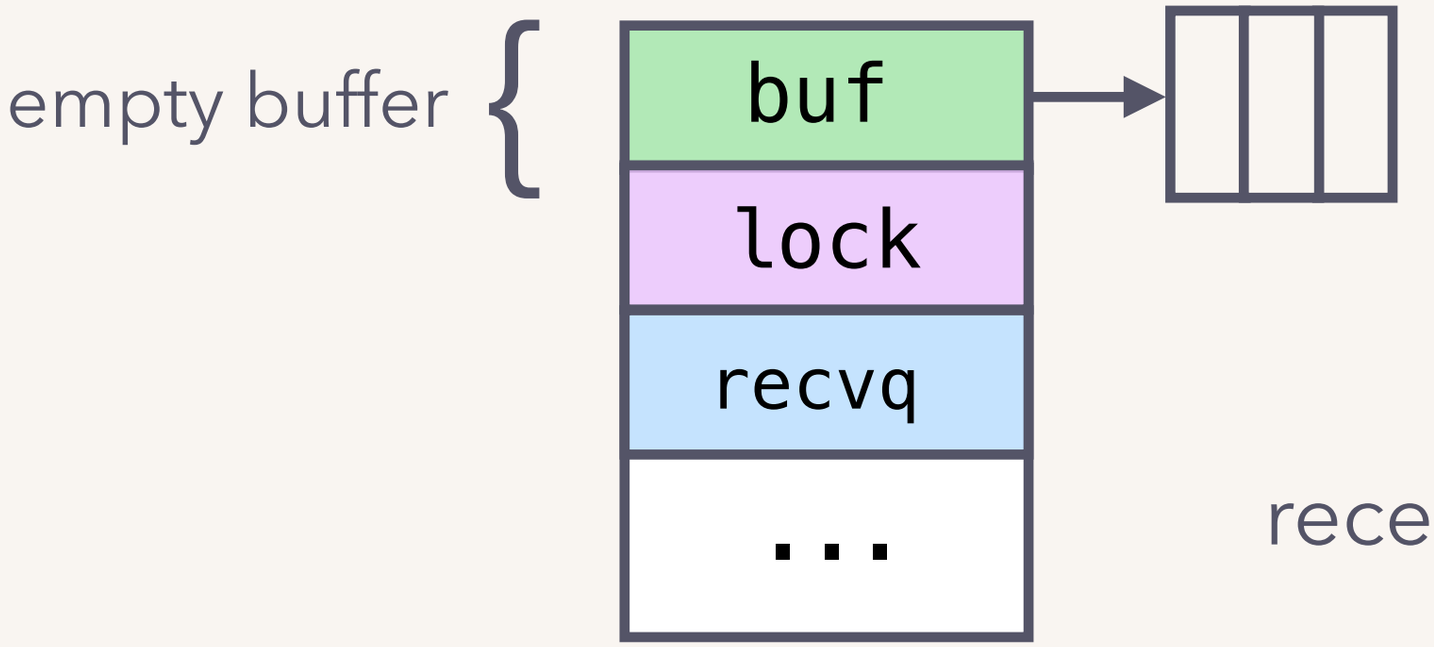
sets G1 to runnable

puts it on runqueue

returns to G2



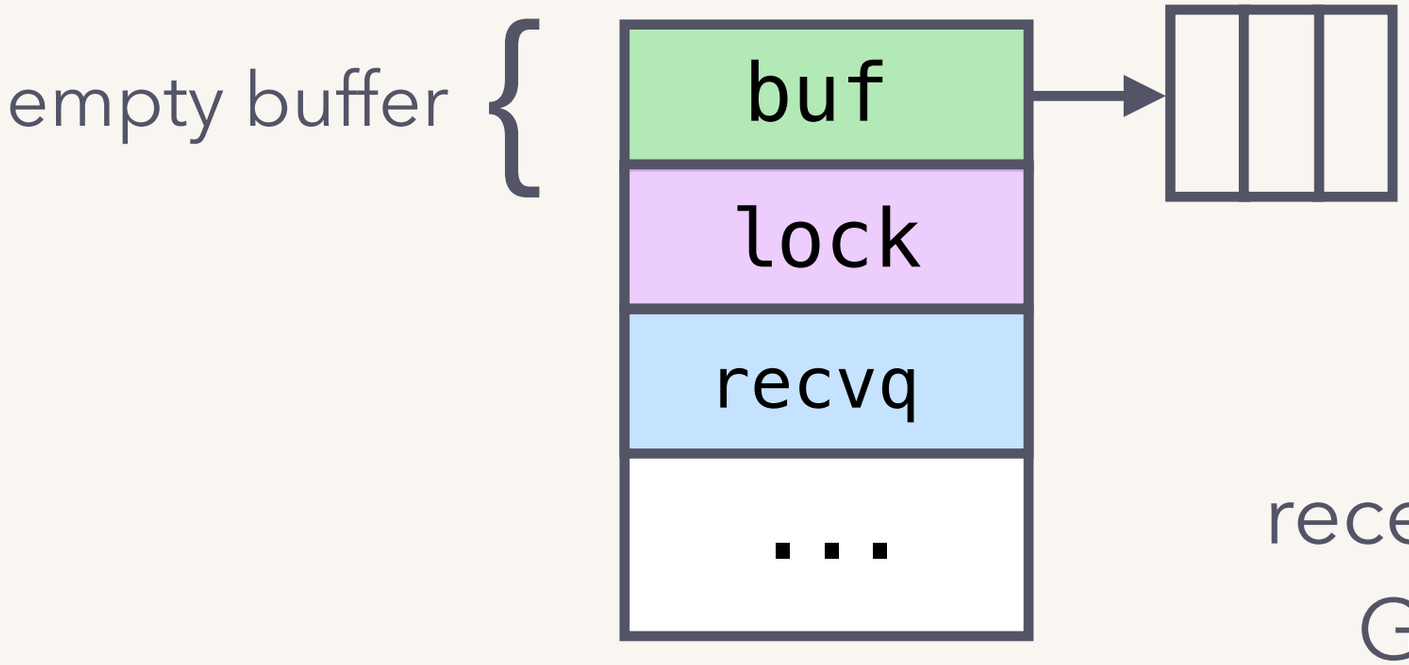
sends and receives
when the **receiver**
comes first



G2

```
t := <-ch
```

receive on an **empty** channel:



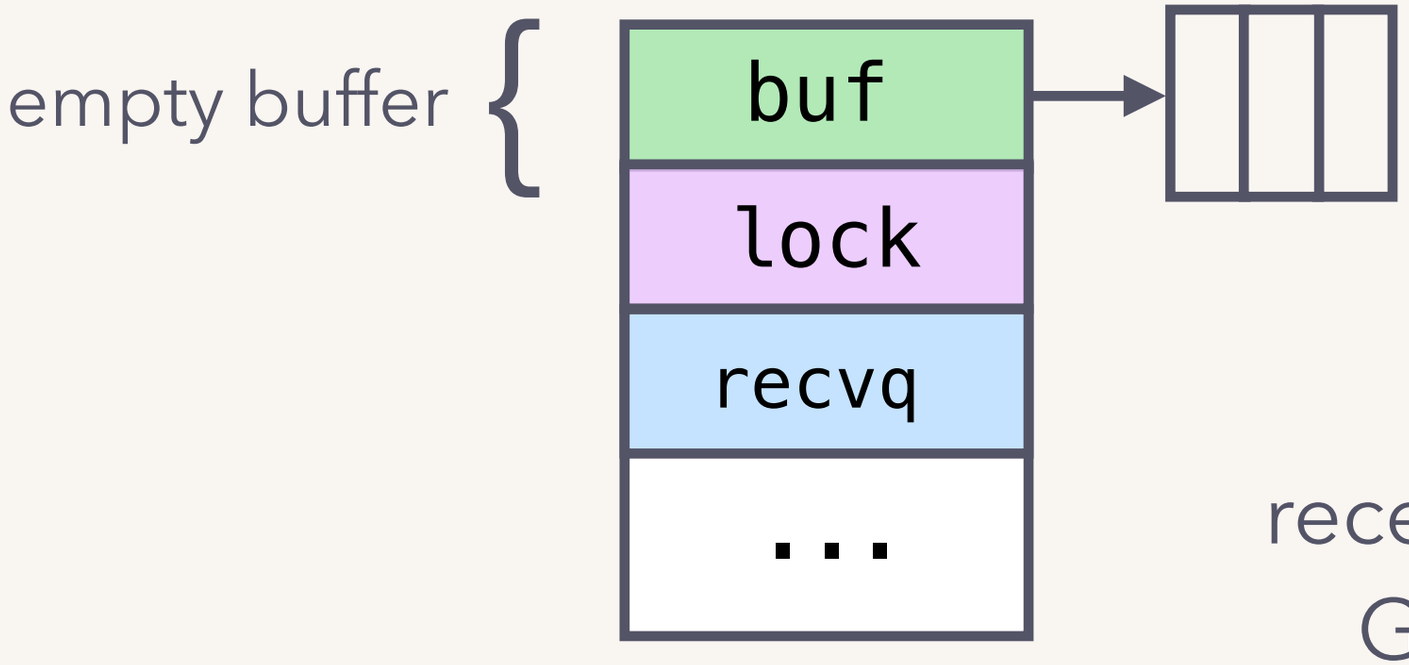
G2

```
t := <-ch
```

receive on an **empty** channel:
G2's execution is **paused**
resumed after a send.

G2

```
t := <-ch
```



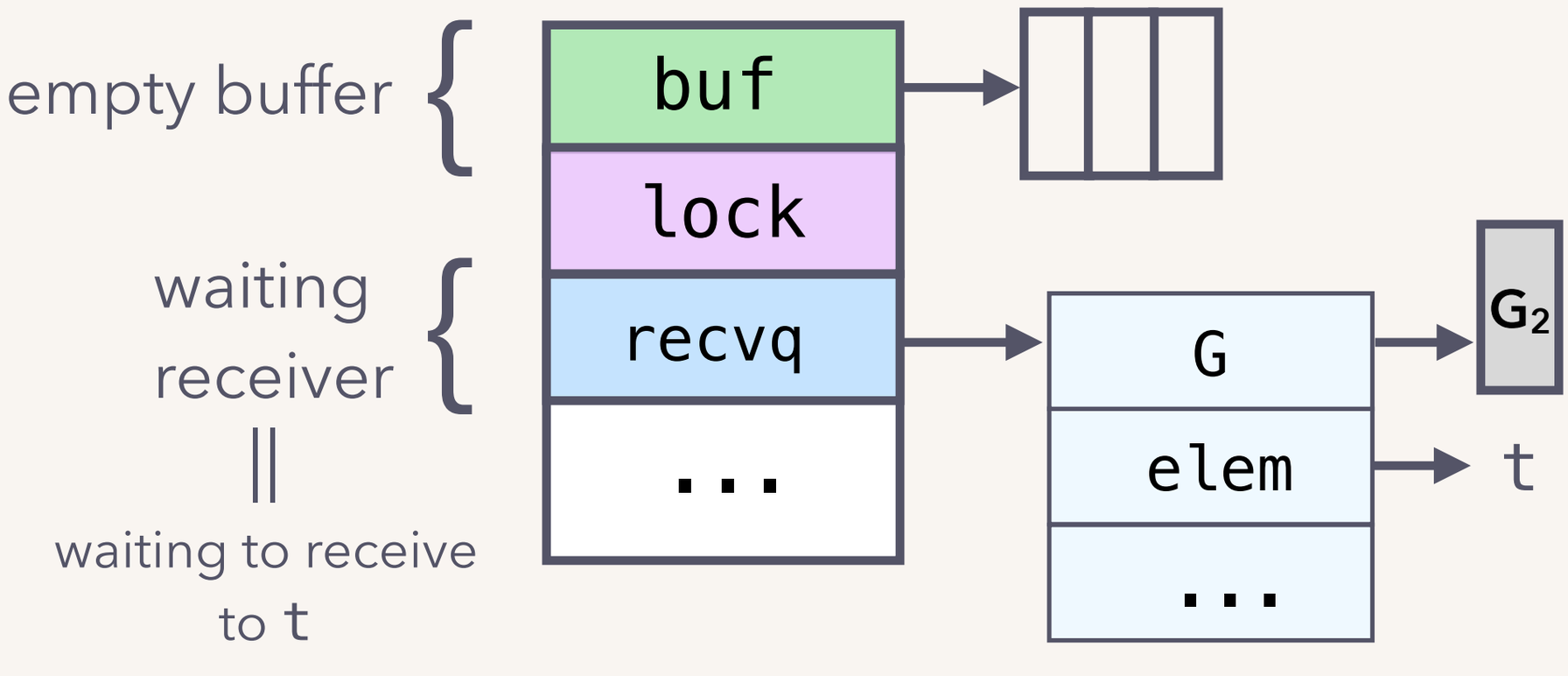
receive on an **empty** channel:
G2's execution is **paused**
resumed after a send.

set up state for resumption, and pause

put a **sudog** in the **recvq** `gopark G2.`

G2

```
t := <-ch
```

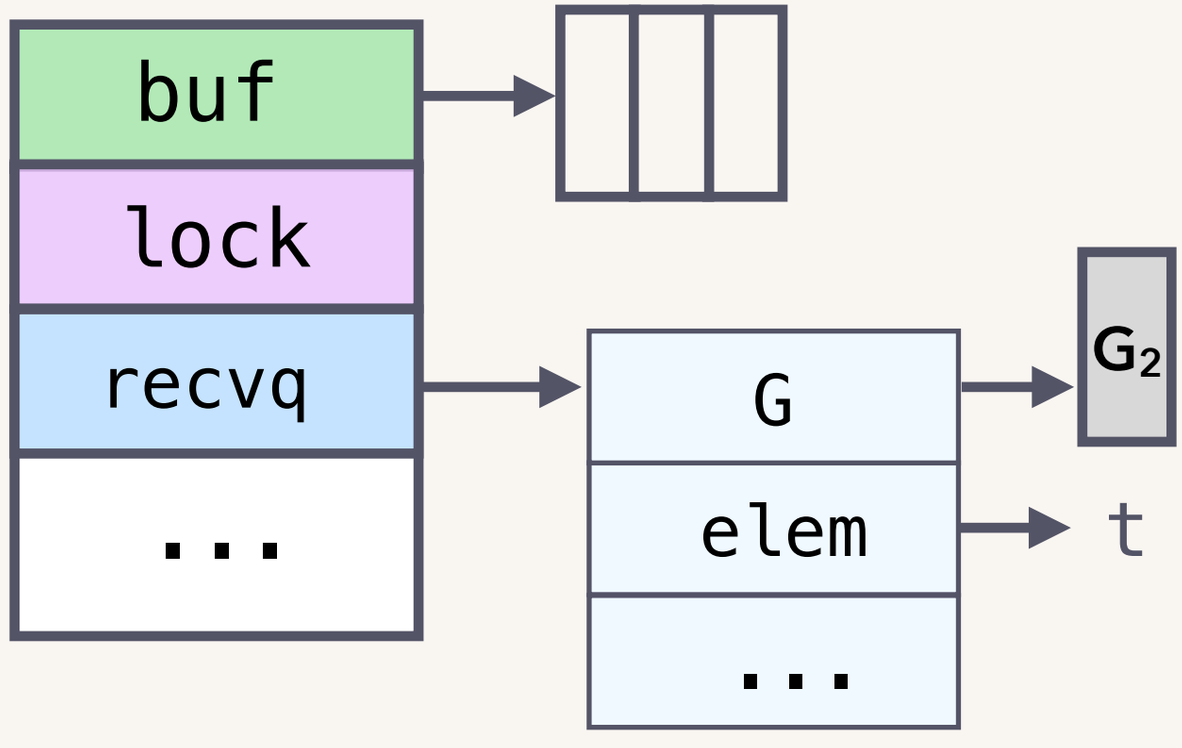


set up state for resumption, and pause

put a **sudog** in the **recvq** `gopark G2.`

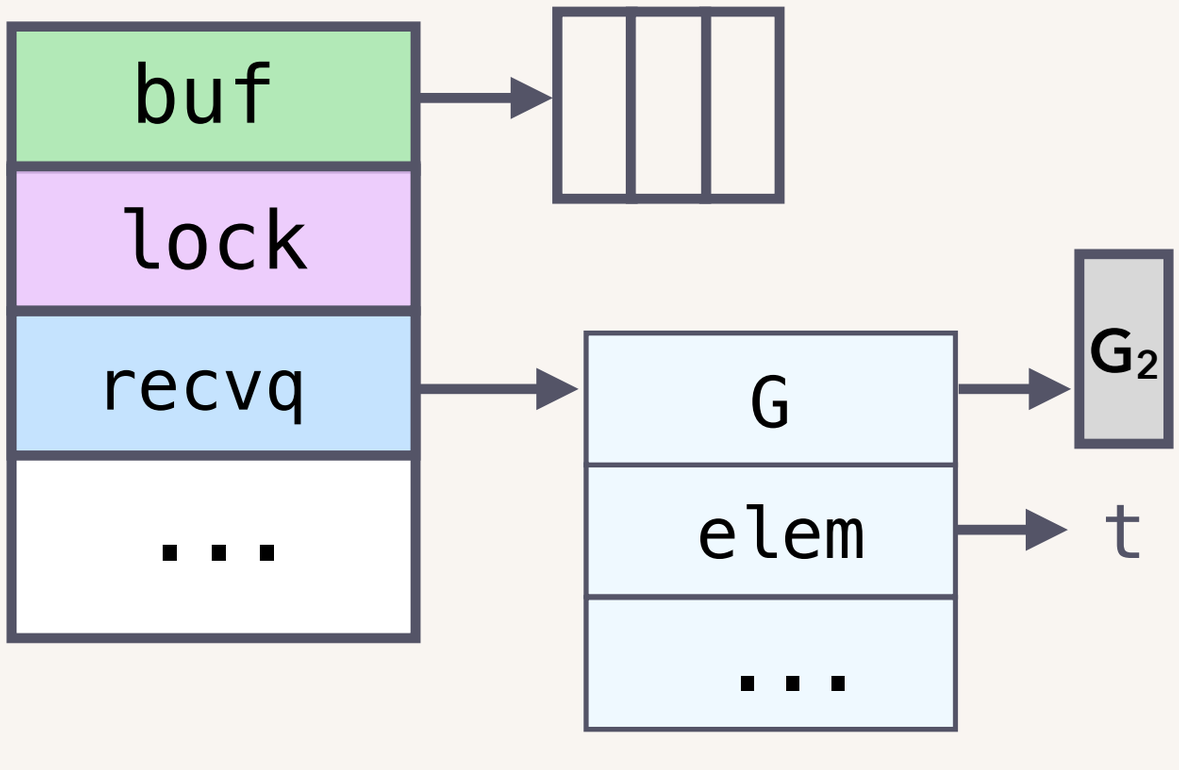
G1

```
ch <- task
```



G1

```
ch <- task
```

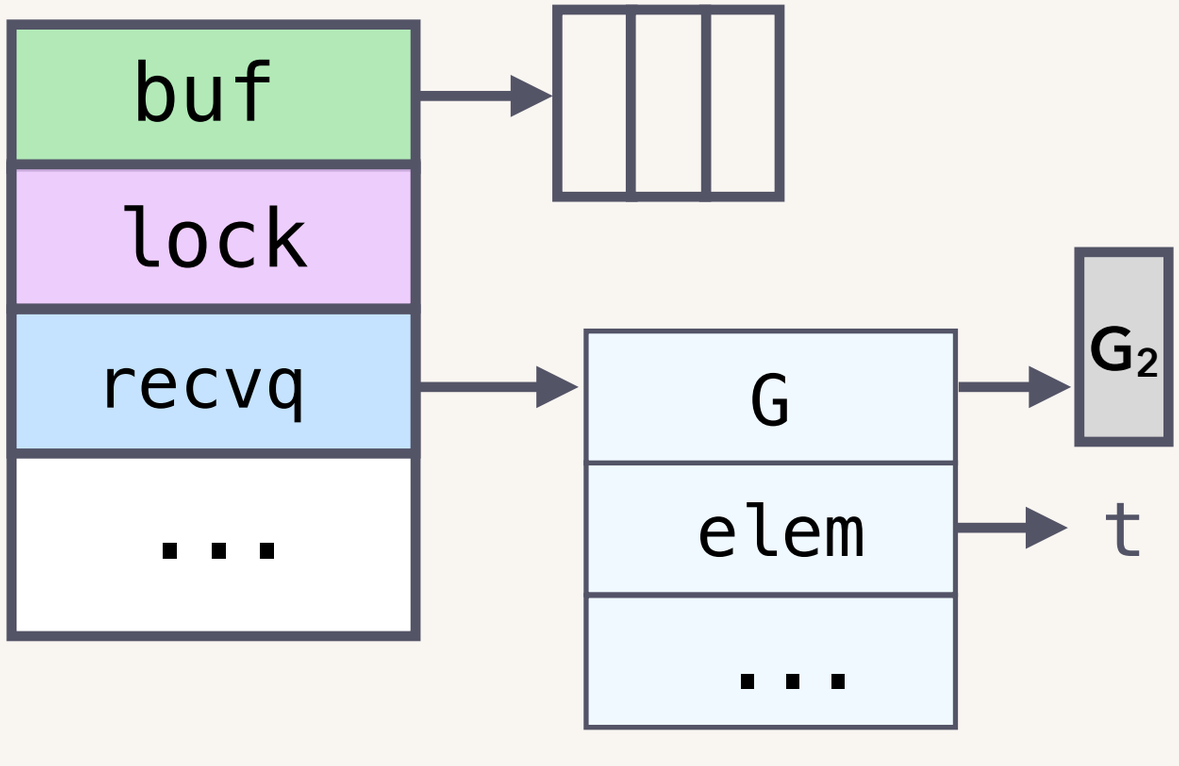


could:

- enqueue `task` in the buffer,
- `goready(G2)`

G1

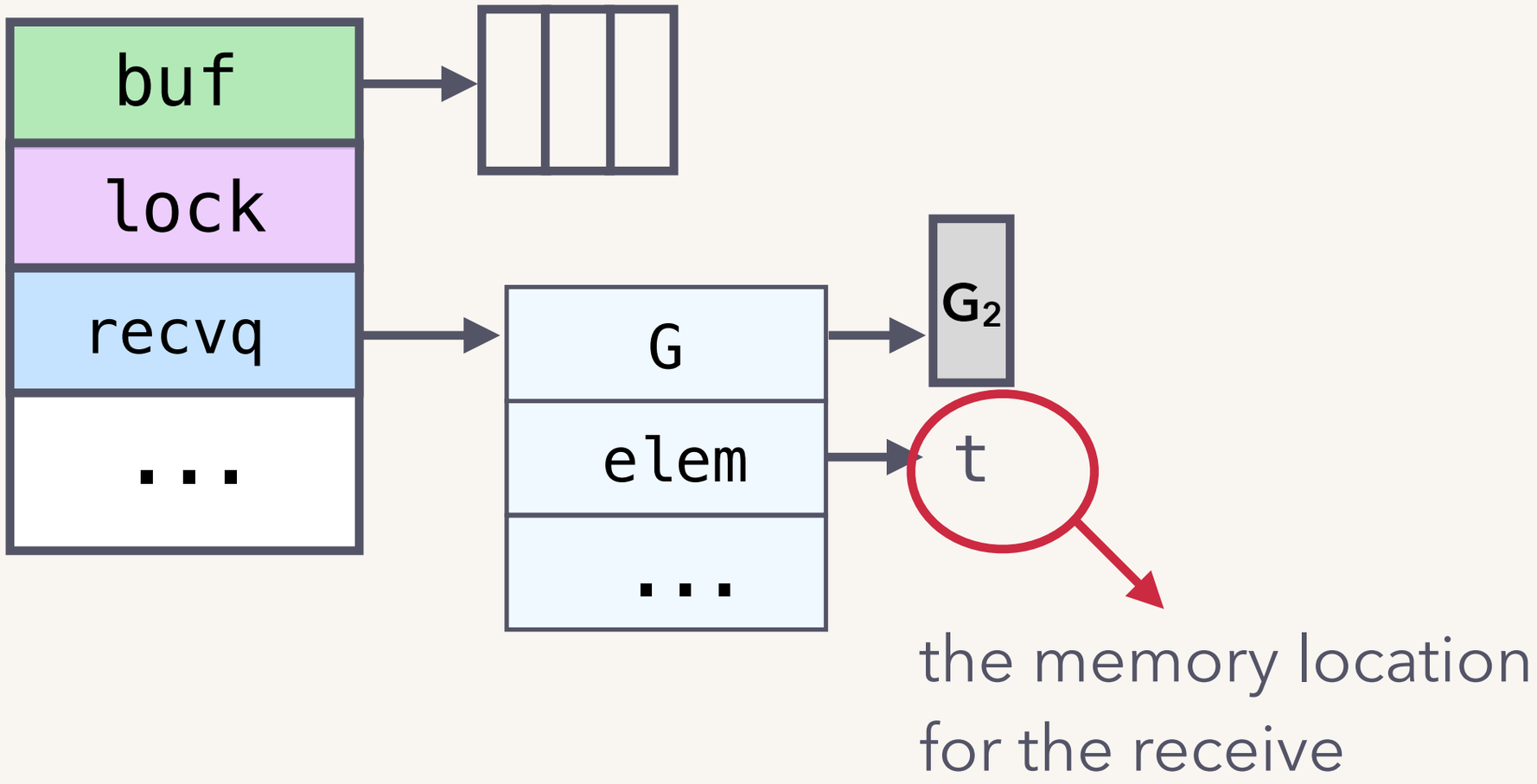
```
ch <- task
```



or we can be **smarter.**

G1

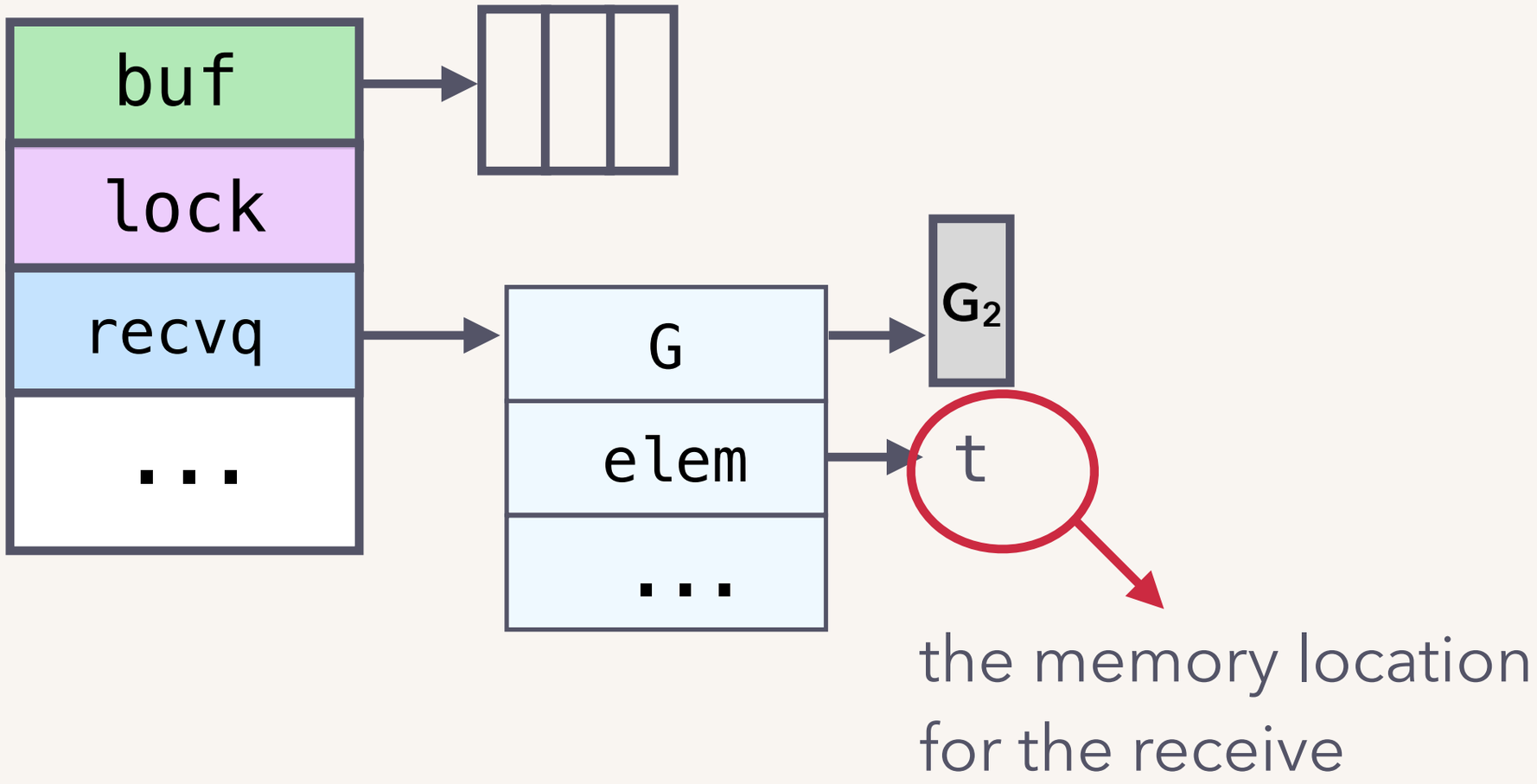
```
ch <- task
```



or we can be **smarter.**

G1

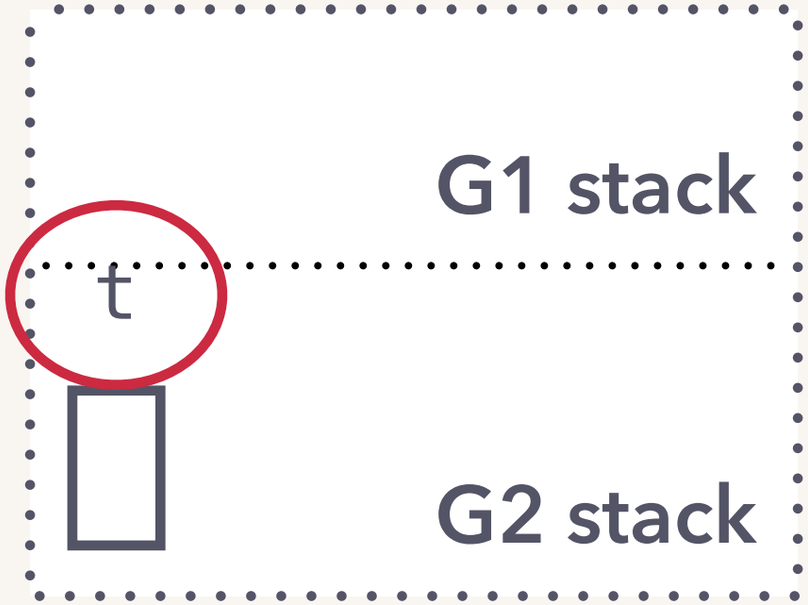
```
ch <- task
```



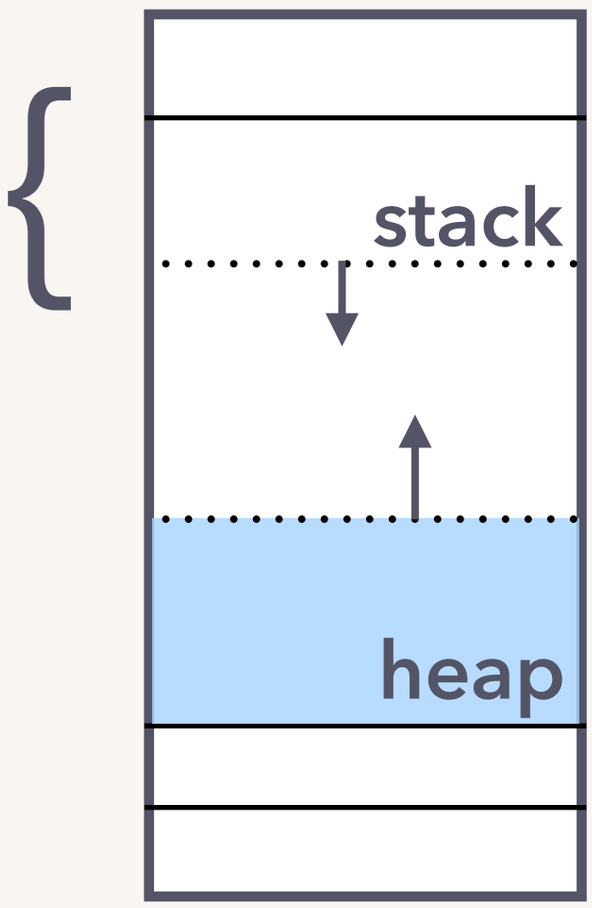
G1 writes to t **directly**.

!

direct send



per-goroutine stacks



G1 writes to G2's stack!

only operations in runtime where this happens.

This is clever.
On resuming, G2 does not need to acquire channel lock and manipulate the buffer.

Also, one fewer memory copy.

we now understand channels (sorta)...

goroutine-safe

- hchan **mutex**

store values, pass in FIFO.

- copying into and out of hchan **buffer**

can cause goroutines to pause and resume.

- hchan **sudog queues**
- calls into the **runtime scheduler** (`gopark`, `goready`)

a note (or two)...

unbuffered channels

unbuffered channels <u>always</u> work like the **“direct send” case**:

- receiver first –> sender writes to receiver’s stack.
- sender first –> receiver receives directly from the sudog.

select (general-case)

- **<u>all</u>** channels locked.
- a sudog is put in the sendq /recvq queues of **<u>all</u>** channels.
- channels unlocked, and the `select`-ing G is **paused**.
- CAS operation so there’s one winning case.
- **resuming** mirrors the pause sequence.

stepping back...

simplicity and **performance**

simplicity

queue with a lock preferred to lock-free implementation:

"The **performance improvement** does not materialize from the air, **it comes with code complexity increase**." — dvyokov

performance

calling into the runtime **scheduler**:

- OS thread remains unblocked.

cross-goroutine stack reads and writes.

- goroutine wake-up path is lockless,
- potentially fewer memory copies

} need to account for memory management:

garbage collection,
stack-shrinking

astute trade-offs
between
simplicity and **performance**

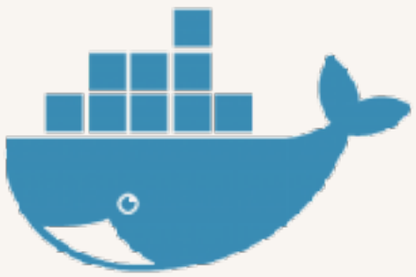
“The noblest pleasure is
the joy of understanding.”

- Leonardo da Vinci

@kavya719

speakerdeck.com/kavya719/understanding-channels

Special thanks to Eben Freeman, Jelle van den Hooff for reading drafts of this.



Docker: software container platform

…Go is geared for distributed computing.
It has many built-in features to support concurrency…

Railgun: CloudFlare's web proxy

We chose to use Go because Railgun is inherently highly concurrent…
Railgun makes extensive use of goroutines and channels.

Doozer: Heroku's distributed data store

Fortunately, Go's concurrency primitives made the task much easier.

Kubernetes: Google's container orchestration platform

Built in concurrency. Building distributed systems in Go is helped tremendously by being able to fan out …