Eran Avidor,

Fullstack TL @ Yotpo

eranavidor.com

eran@yotpo.com

@_eranavidor

YOT PO.

What are Go Channels?

... and how understanding them will make you a better Go programmer

Go is trending

What are we gonna talk about?

- “Go in 60 seconds”
- Built-in Concurrency Primitives
  - goroutines
  - channels
- Go Channels
  - Type & Structure
  - Mechanics
  - Design notes

“Go in 60 seconds”, or Why Go?

- Developed at Google
  - Conceived by Rob Pike at 2007
  - Go 1 Released on March 2012
- Objectives:
  - Match today's needs (multi-cores, distributed systems, web oriented)
  - Productiveness
  - Ease of use
  - Reliability
  - Performance orientation

“Go in 60 seconds”, or Why Go?

- Go’s main properties
  - **Concurrent**
  - Compiled
  - Statically typed
  - Garbage Collected

Concurrency

"In programming, **concurrency is the composition of independently executing processes**, while parallelism is the simultaneous execution of (possibly related) computations. **Concurrency is about dealing with lots of things at once.** Parallelism is about doing lots of things at once"

Rob Pike

Go’s Concurrency Primitives

- go**routines**
  - light-weight threads
  - managed by the go runtime env (not OS)
  - independent

Go’s Concurrency Primitives

- go**routines**
  - light-weight threads
  - managed by the go runtime env (not OS)
  - independent

```
func main() {
      myFunc("synchronously")
      // blocked code
}
```

Go’s Concurrency Primitives

- go**routines**
  - light-weight threads
  - managed by the go runtime env (not OS)
  - independent

```
func main() {
      myFunc("synchronously")
      // blocked code
}
```

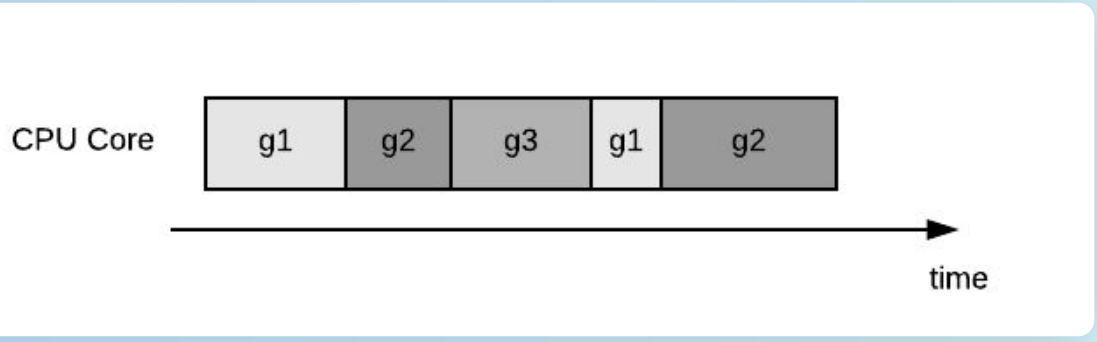
```
func main() {
      go myFunc("concurrently")
      // blocking code
}
```

Go’s Concurrency Primitives

- go**routines**
  - light-weight threads
  - managed by the go runtime env (not OS)
  - independent
  - parallel (potentially)

```
func main() {
      myFunc("synchronously")
      // blocked code
}
```

```
func main() {
      go myFunc("concurrently")
      // blocking code
}
```

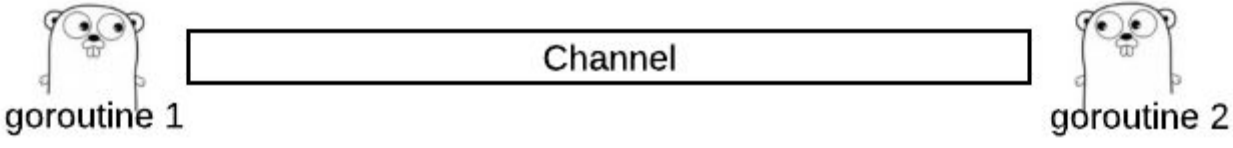


Go’s Concurrency Primitives

- channels
  - pipes that connect goroutines
  - enable communication & synchronization
  - special concept

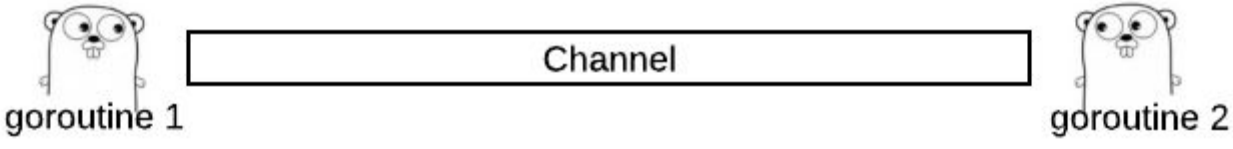
Go’s Concurrency Primitives

- channels
  - pipes that connect goroutines
  - enable communication & synchronization
  - special concept



Go’s Concurrency Primitives

- channels
  - pipes that connect goroutines
  - enable communication & synchronization
  - special concept
- **built-in mechanism**
  - easy to use
  - designed as part of the STL



Simple jobs processing without channels

```
func main() {
	jobs := gimmeWork()

	for _, job := range jobs {
		process(job)
	}
}
```

Simple jobs queue pattern using channels

```
func main() {
	ch := make(chan Job, bufferedChSize)

	for i := 0; i < workersCount; i++ {
		go worker(ch)
	}

	jobs := gimmeWork()

	for _, job := range jobs {
		ch <- job
	}

	// blocking code
}
```

```
func worker(ch <-chan Job) {
	for {
		job := <-ch
		process(job)
	}
}
```

Simple jobs queue pattern using channels

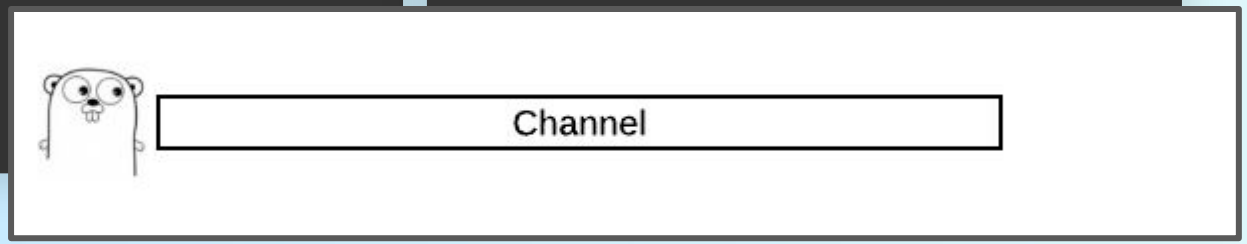
```
func main() {
	ch := make(chan Job, bufferedChSize)

	for i := 0; i < workersCount; i++ {
		go worker(ch)
	}

	jobs := gimmeWork()

	for _, job := range jobs {
		ch <- job
	}

	// blocking code
}
```

```
func worker(ch <-chan Job) {
	for {
		job := <-ch
		process(job)
	}
}
```

Simple jobs queue pattern using channels

```
func main() {
	ch := make(chan Job, bufferedChSize)

	for i := 0; i < workersCount; i++ {
		go worker(ch)
	}

	jobs := gimmeWork()

	for _, job := range jobs {
		ch <- job
	}

	// blocking code
}
```

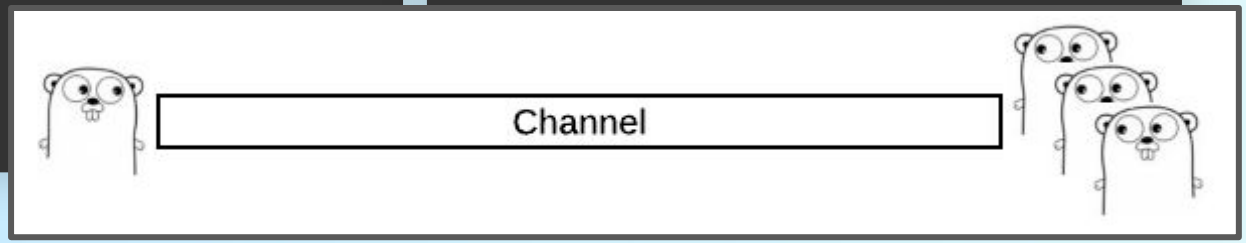
```
func worker(ch <-chan Job) {
	for {
		job := <-ch
		process(job)
	}
}
```



Simple jobs queue pattern using channels

```
func main() {
	ch := make(chan Job, bufferedChSize)

	for i := 0; i < workersCount; i++ {
		go worker(ch)
	}

	jobs := gimmeWork()

	for _, job := range jobs {
		ch <- job
	}

	// blocking code
}
```

```
func worker(ch <-chan Job) {
	for {
		job := <-ch
		process(job)
	}
}
```

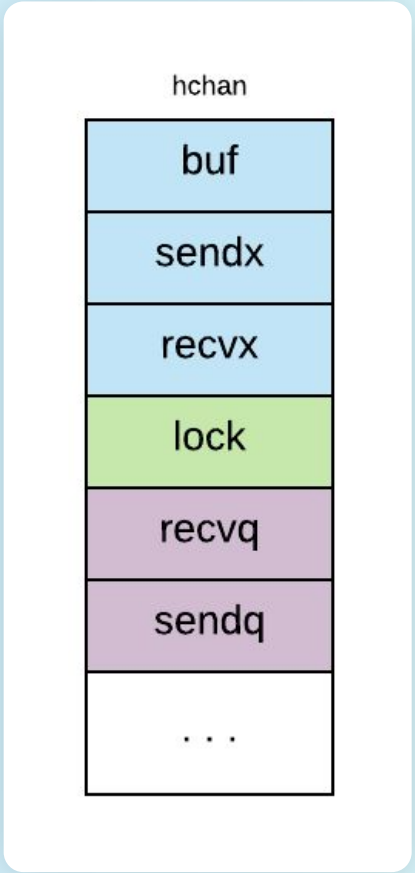


So what do we get?

- Processing order & Capacity control
  - FIFO execution order
  - cannot contain more elements than the channel size
- goroutine safety
  - safe memory access from different goroutines
- Easy interface
  - channels can be passed as an argument
- Send & receive blocking
  - send when channel is full ⇒ goroutine is paused
  - receive when channel is empty ⇒ goroutine is paused
  - automatic resume

Let’s break down this magic

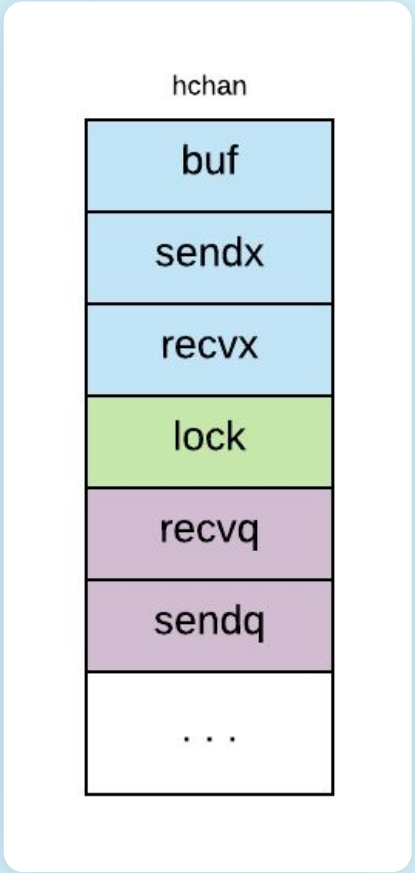
The *hchan* object



The *hchan* object

- *make* constructs an *hchan* struct

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

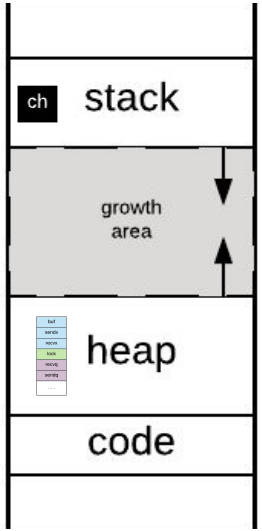


The *hchan* object

- *make* constructs an *hchan* struct

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

- the *hchan* is allocated on the heap

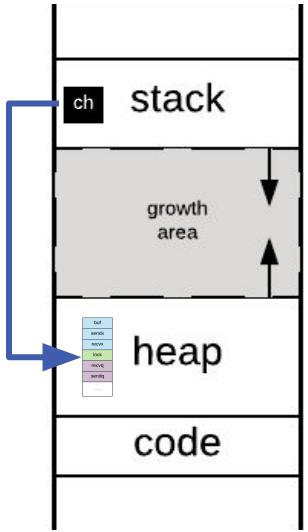


The *hchan* object

- *make* constructs an *hchan* struct

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

- the *hchan* is allocated on the heap
- `ch` is simply a pointer

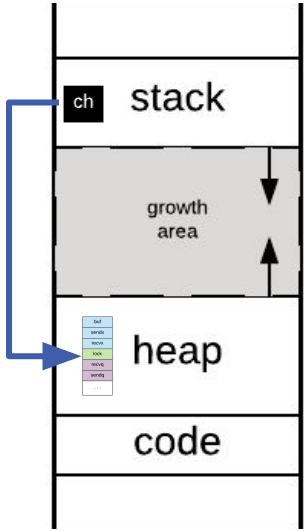


The *hchan* object

- *make* constructs an *hchan* struct

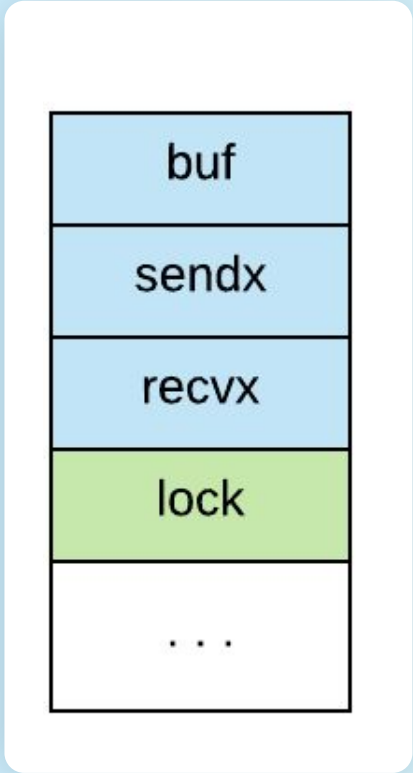
```
ch := make(chan Job, 3)
```

- the *hchan* is allocated on the heap
- `ch` is simply a pointer
- `ch` can be passed as an argument



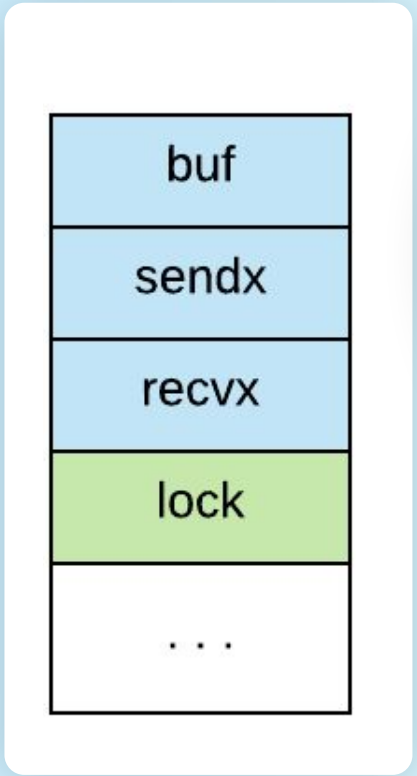
FIFO order & goroutine safety

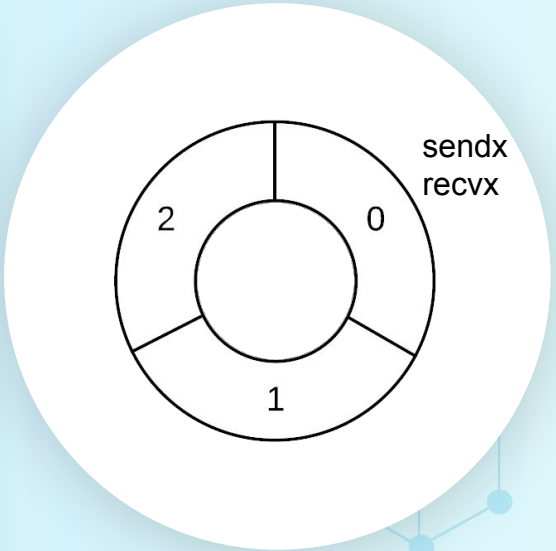
```
ch := make(chan Job, 3)
```



FIFO order & goroutine safety

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

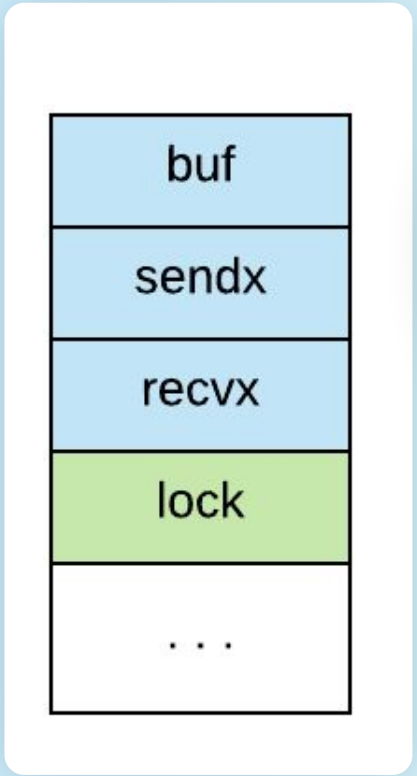


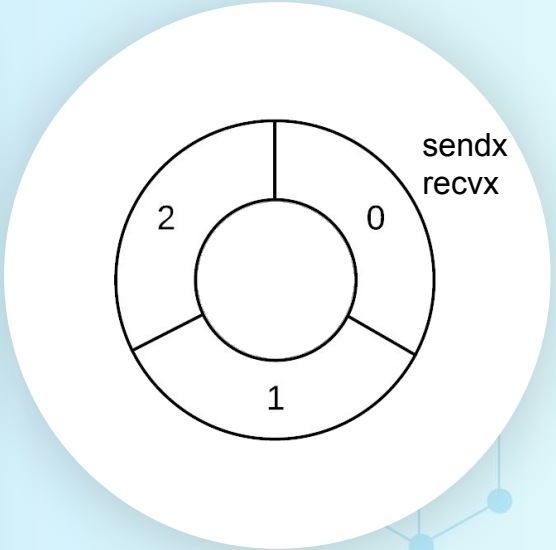


FIFO order & goroutine safety

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

Gmain Gwork





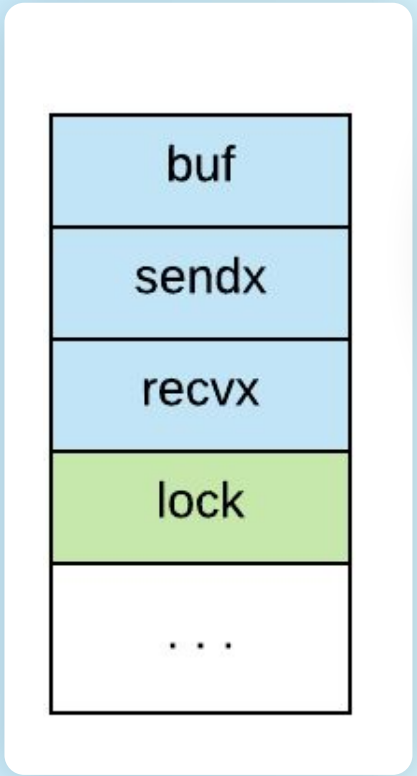
FIFO order & goroutine safety

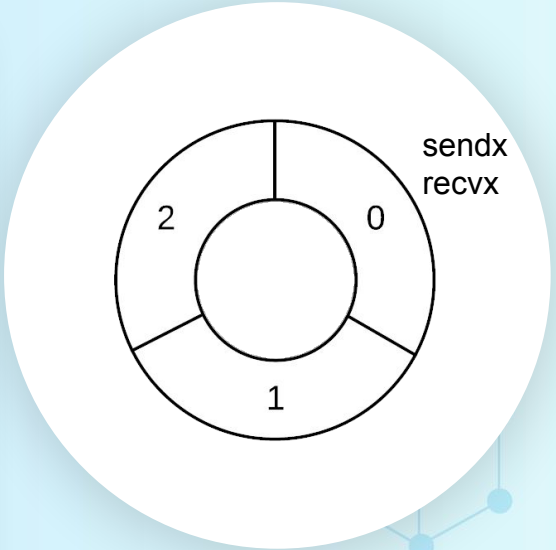
```
ch := make(chan Job, 3)
```

Gmain

```
ch <- job1
```

Gwork





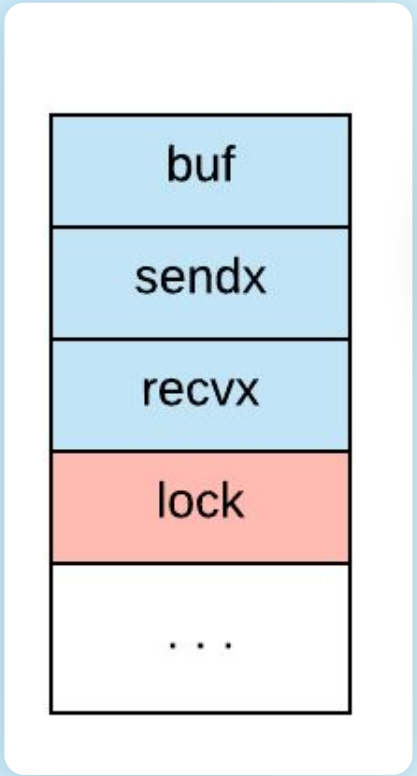
FIFO order & goroutine safety

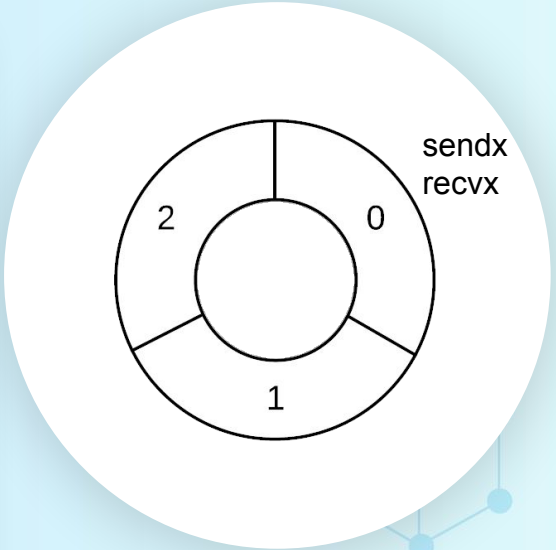
```
ch := make(chan Job, 3)
```

Gmain

```
ch <- job1
```

Gwork





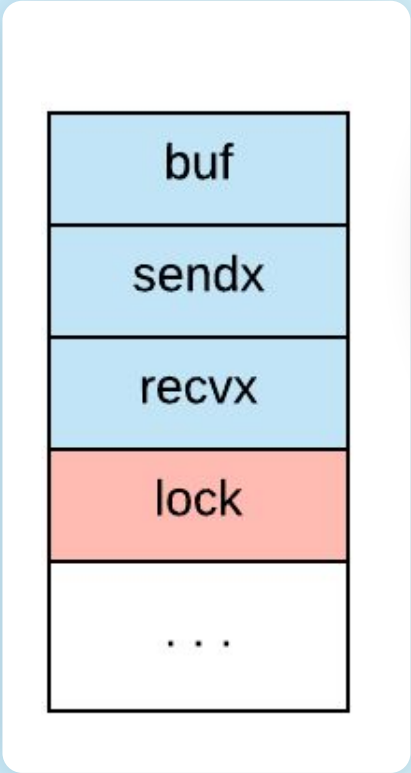
FIFO order & goroutine safety

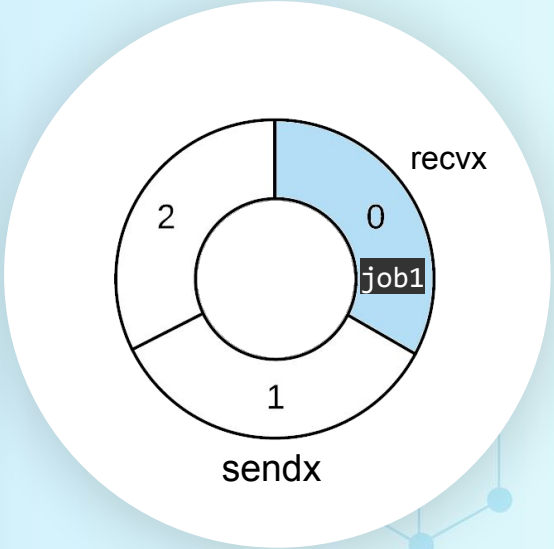
```
ch := make(chan Job, 3)
```

Gmain

```
ch <- job1
```

Gwork





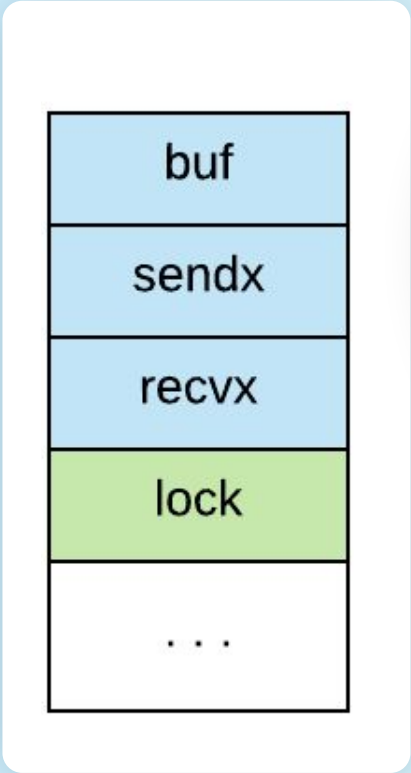
FIFO order & goroutine safety

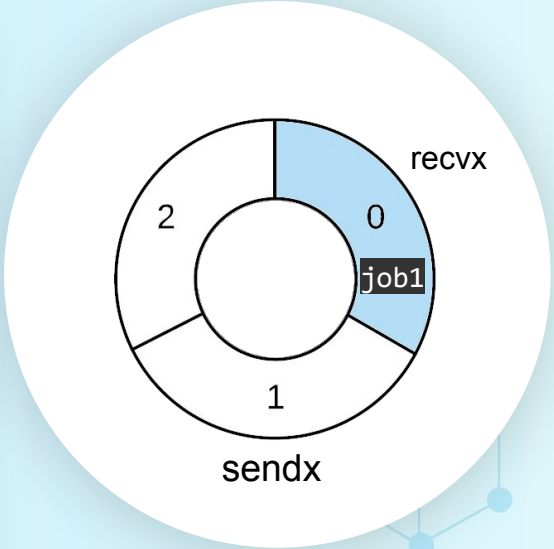
```
ch := make(chan Job, 3)
```

Gmain

```
ch <- job1
```

Gwork





FIFO order & goroutine safety

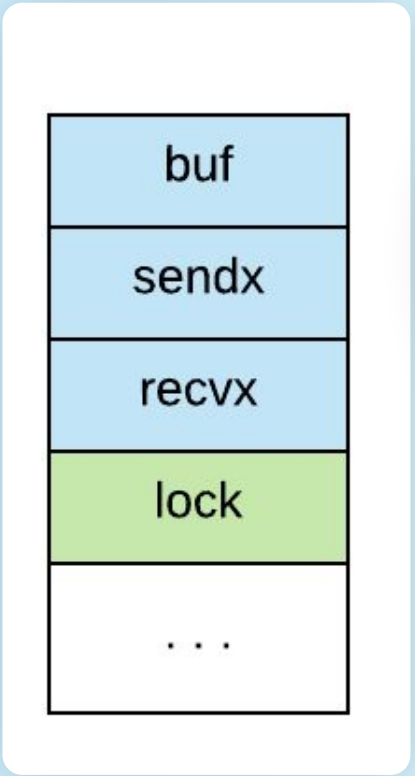
```
ch := make(chan Job, 3)
```

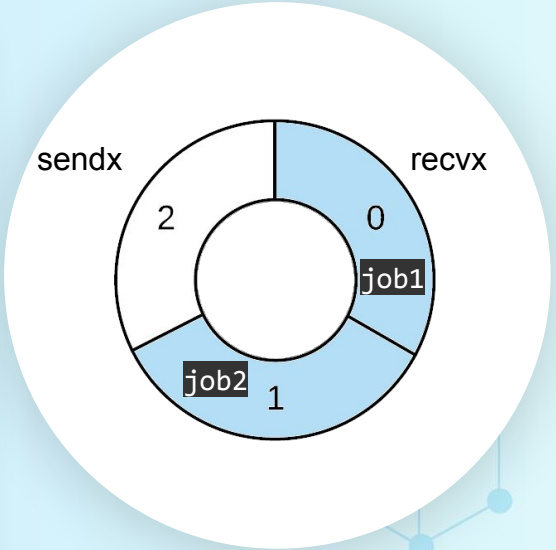
Gmain

```
ch <- job1
```

```
ch <- job2
```

Gwork





FIFO order & goroutine safety

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

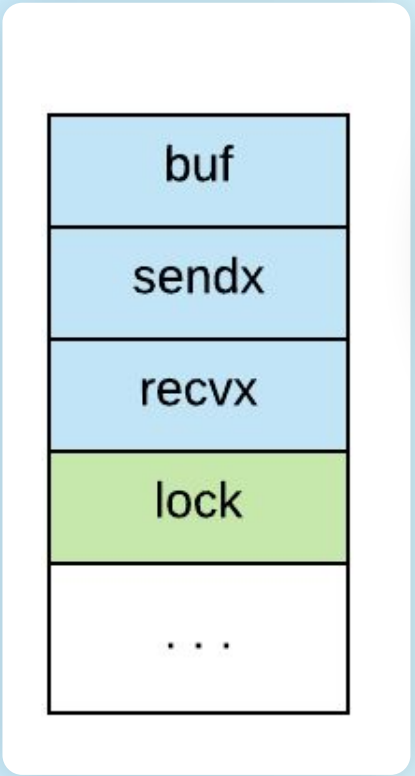
Gmain

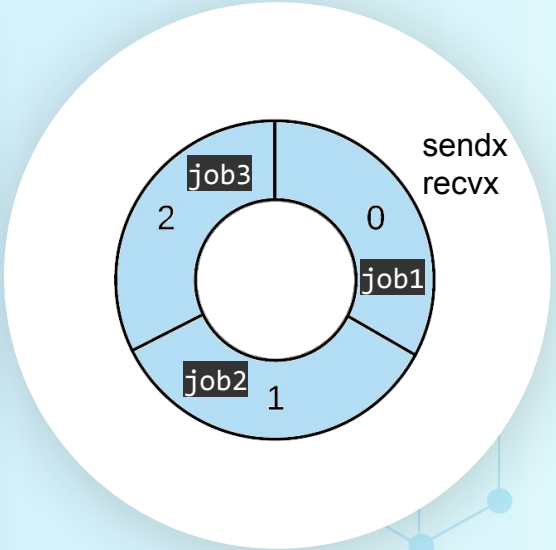
```
ch <- job1
```

```
ch <- job2
```

```
ch <- job3
```

Gwork





FIFO order & goroutine safety

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

Gmain

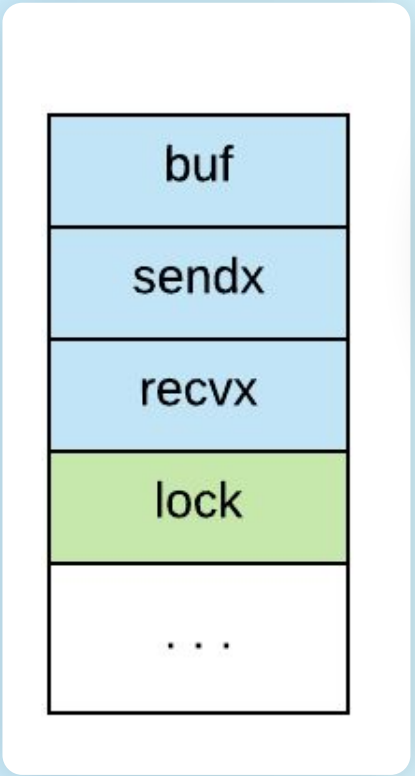
```
ch <- job1
```

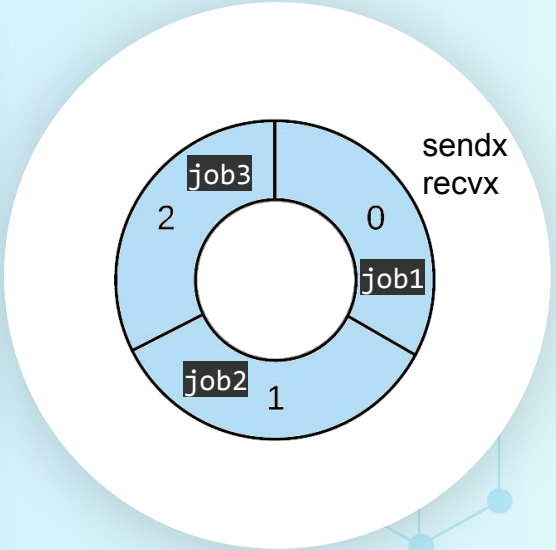
```
ch <- job2
```

```
ch <- job3
```

Gwork

```
job := <-ch
```





FIFO order & goroutine safety

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

Gmain

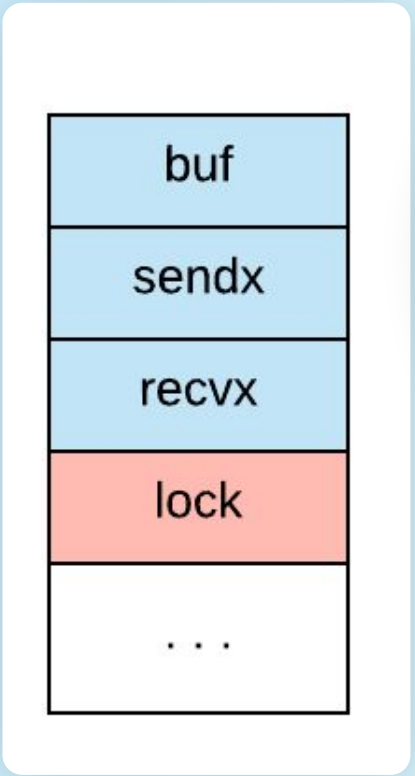
```
ch <- job1
```

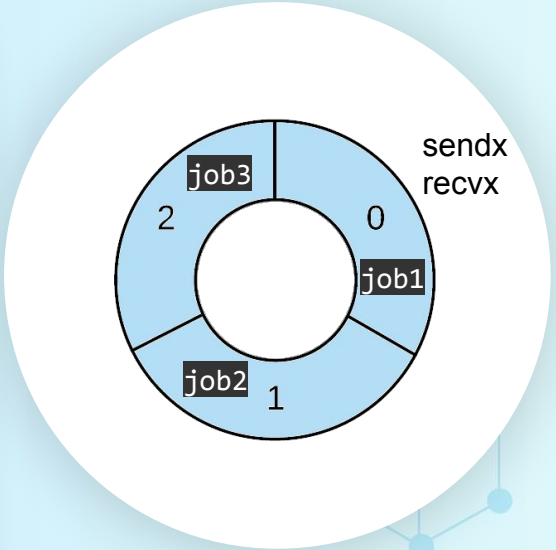
```
ch <- job2
```

```
ch <- job3
```

Gwork

```
job := <-ch
```





FIFO order & goroutine safety

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

Gmain

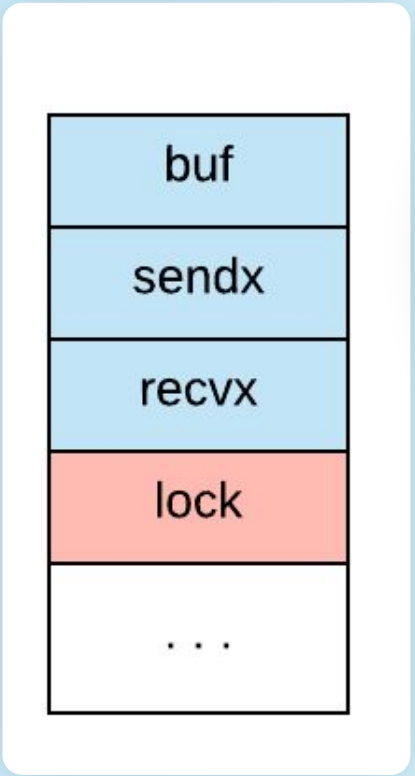
```
ch <- job1
```

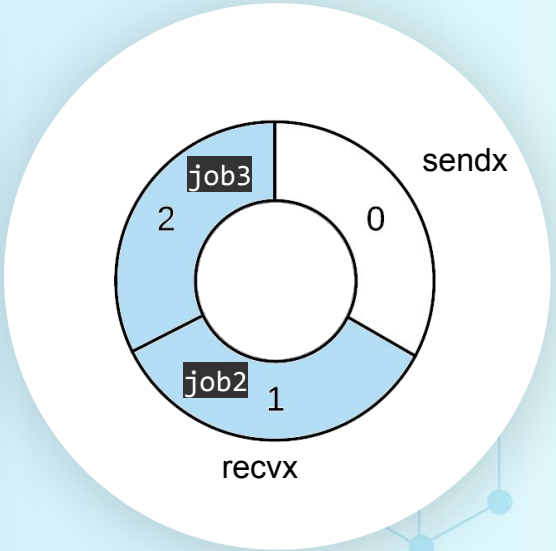
```
ch <- job2
```

```
ch <- job3
```

Gwork

```
job := <-ch
```





FIFO order & goroutine safety

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

Gmain

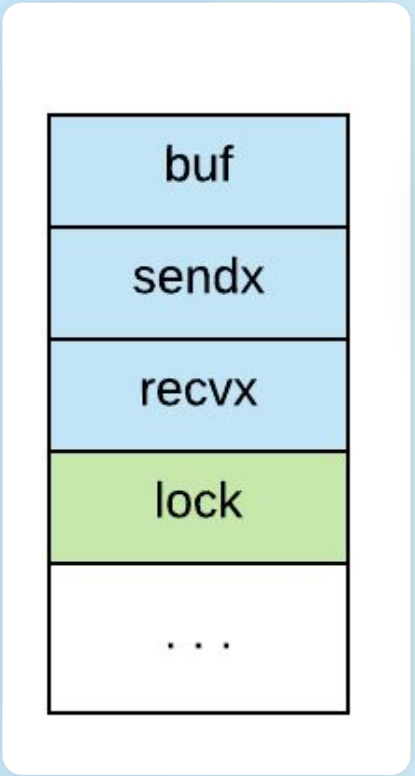
```
ch <- job1
```

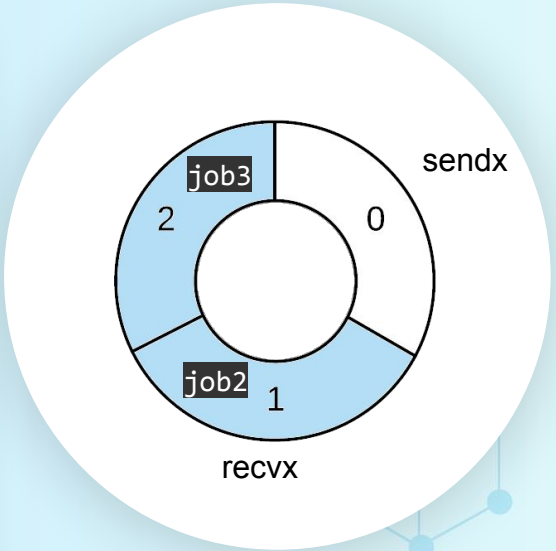
```
ch <- job2
```

```
ch <- job3
```

Gwork

```
job := <-ch
```





Send & Receive blocking nature

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

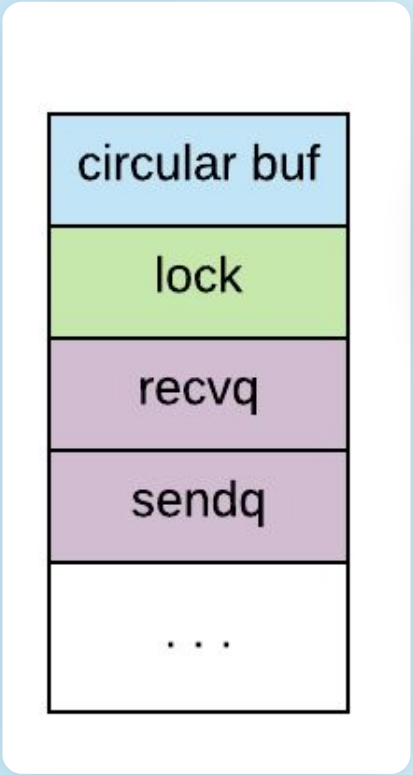
Gmain

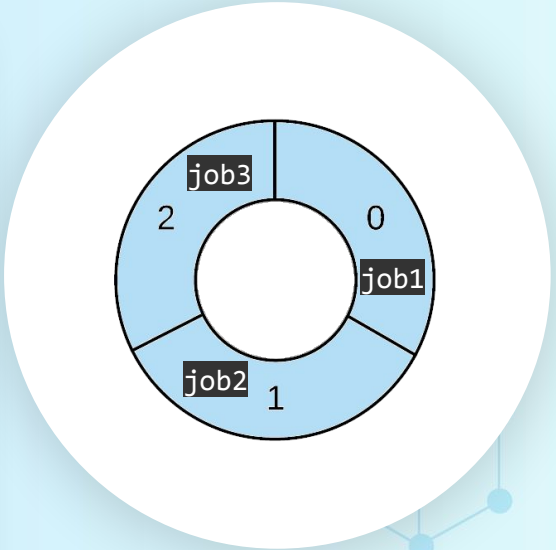
```
ch <- job1
```

```
ch <- job2
```

```
ch <- job3
```

Gwork





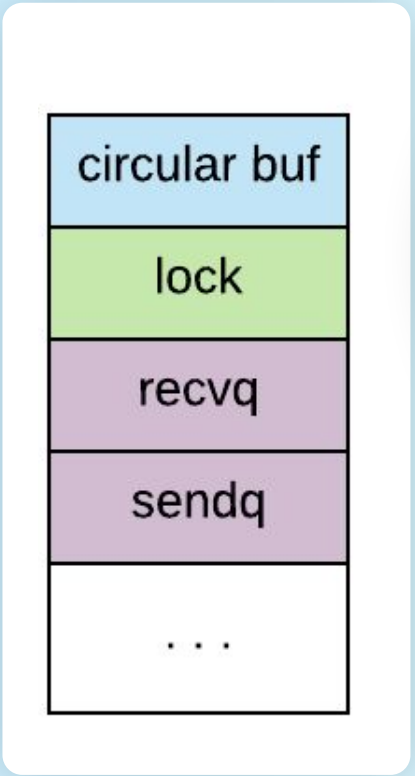
Send & Receive blocking nature

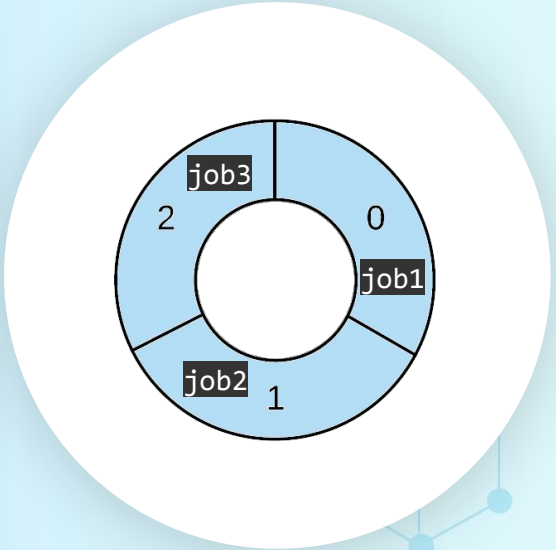
```
ch := make(chan Job, 3)
```

Gmain

```
ch <- job1
ch <- job2
ch <- job3
ch <- job4
```

Gwork





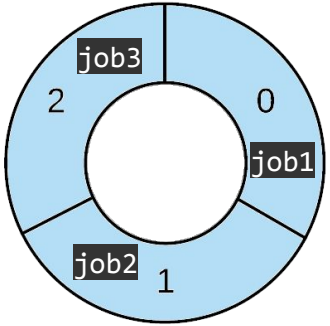
Send & Receive blocking nature

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

Gmain **Gwork**

```
ch <- job1
ch <- job2
ch <- job3
ch <- job4
```

Gm ch is full. execution is paused & resumed after a receive is performed.



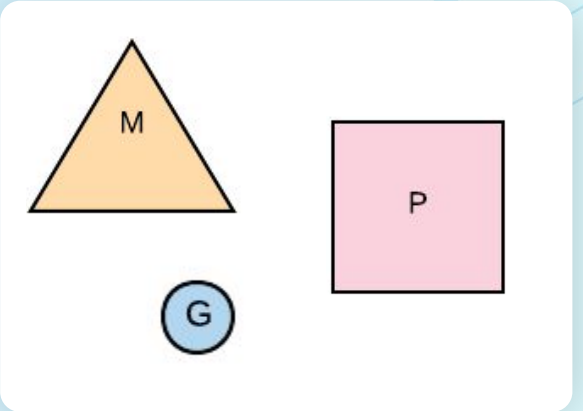
Go’s runtime scheduler (concept)

- Logical scheduling layer on top of the OS’s scheduler
  - part of Go’s runtime environment
  - **Go Scheduler:** allocate goroutines to OS threads
  - **OS scheduler:** allocate OS threads to physical CPUs

Go’s runtime scheduler (concept)

- Logical scheduling layer on top of the OS’s scheduler
  - part of Go’s runtime environment
  - **Go Scheduler:** allocate goroutines to OS threads
  - **OS scheduler:** allocate OS threads to physical CPUs

- Main participants:
  - **M:** Machine (OS thread)
  - **P:** Logical Processor (“holds the goroutines”)
  - **G:** goroutine

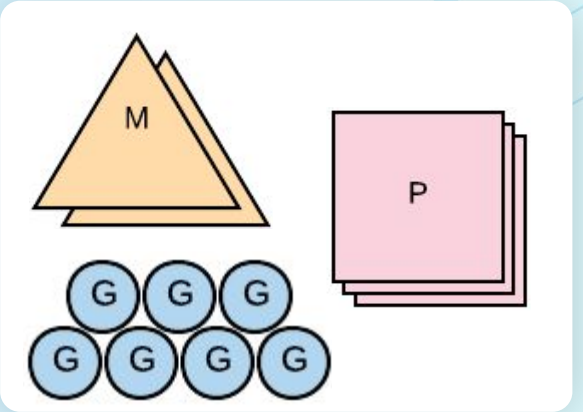


Go’s runtime scheduler (concept)

- Logical scheduling layer on top of the OS’s scheduler
  - part of Go’s runtime environment
  - **Go Scheduler:** allocate goroutines to OS threads
  - **OS scheduler:** allocate OS threads to physical CPUs

- Main participants:
  - **M:** Machine (OS thread)
  - **P:** Logical Processor (“holds the goroutines”)
  - **G:** goroutine

- In general, multiple instances



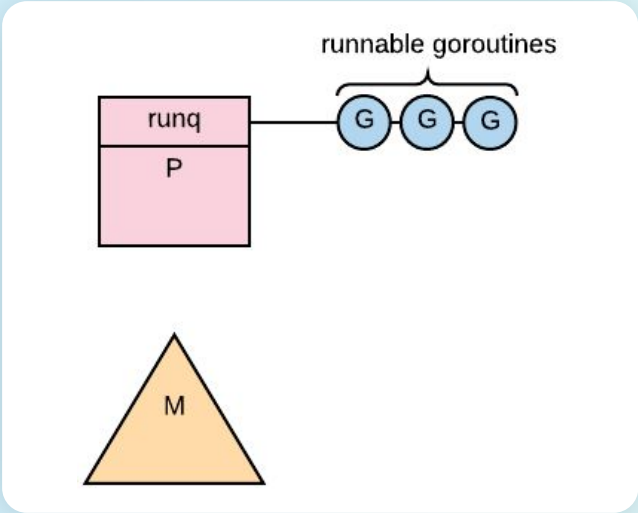
Go’s runtime scheduler (scheduling event)

- The scheduler assigns a goroutine **G** from the logical processor **P** to thread **M**

- Main participants:
  - **M:** Machine (OS thread)
  - **P:** Logical Processor (“holds the goroutines”)
  - **G:** goroutine

Go’s runtime scheduler (scheduling event)

- The scheduler assigns a goroutine **G** from the logical processor **P** to thread **M**
  - **P** holds queue of **G**s (*runq*) that are ready to run

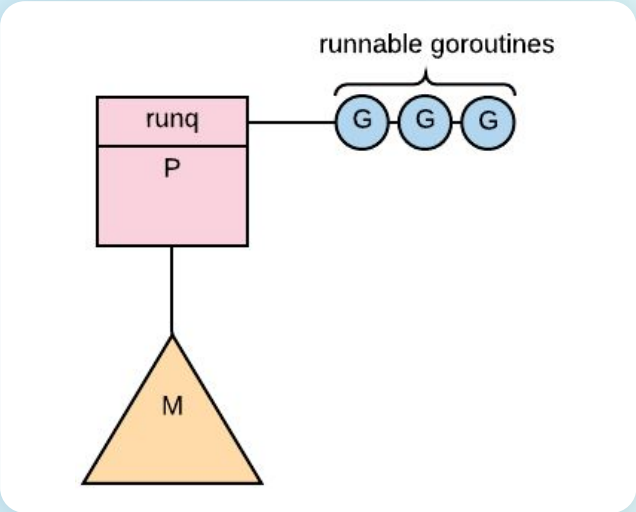


- Main participants:
  - **M:** Machine (OS thread)
  - **P:** Logical Processor (“holds the goroutines”)
  - **G:** goroutine

Go’s runtime scheduler (scheduling event)

- The scheduler assigns a goroutine **G** from the logical processor **P** to thread **M**
  - **P** holds queue of **G**s (*runq*) that are ready to run
  - The scheduler assigns **P** to machine **M**

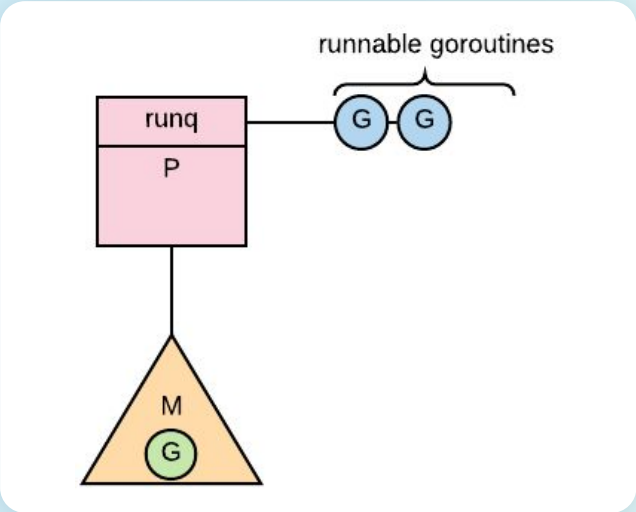
- Main participants:
  - **M:** Machine (OS thread)
  - **P:** Logical Processor (“holds the goroutines”)
  - **G:** goroutine



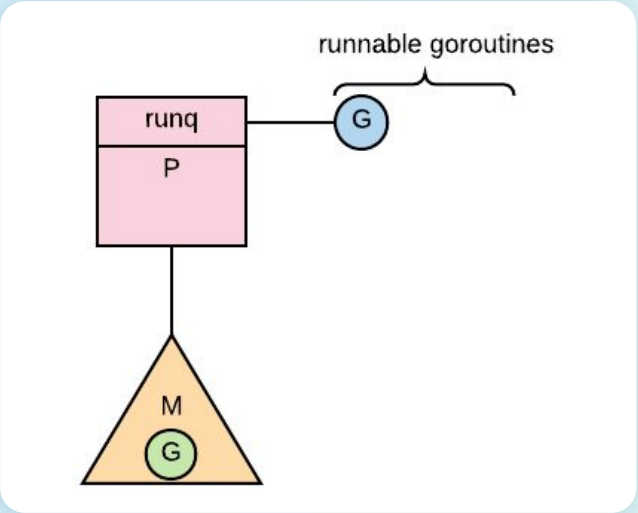
Go’s runtime scheduler (scheduling event)

- The scheduler assigns a goroutine **G** from the logical processor **P** to thread **M**
  - **P** holds queue of **G**s (*runq*) that are ready to run
  - The scheduler assigns **P** to machine **M**
  - Then, the scheduler pops a **G** from the *runq* of **P** and assigns it to **M**

- Main participants:
  - **M:** Machine (OS thread)
  - **P:** Logical Processor (“holds the goroutines”)
  - **G:** goroutine

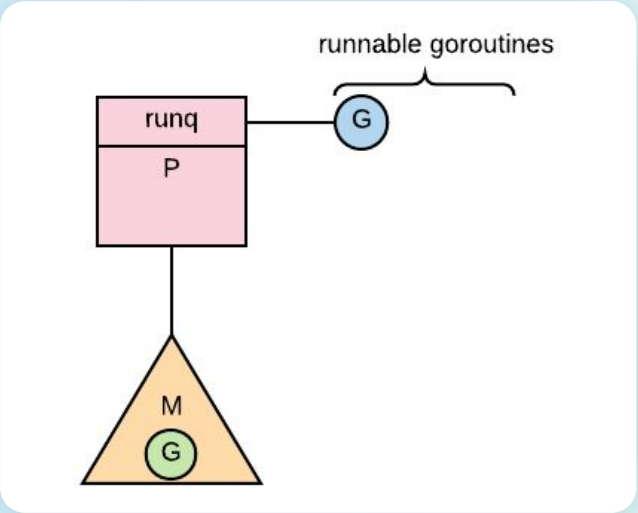


Go’s runtime scheduler (actions)



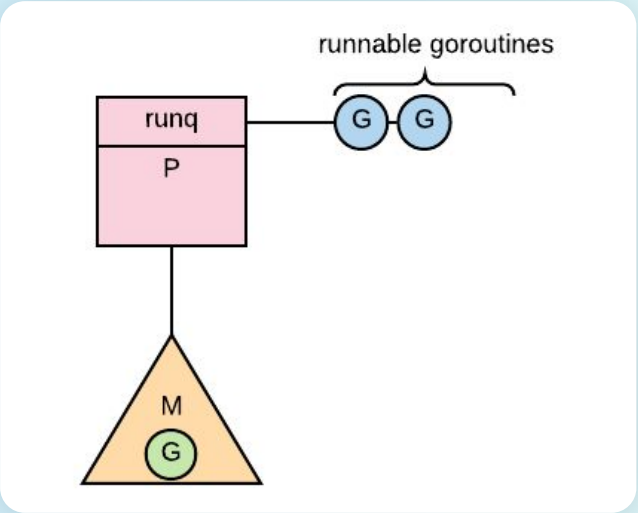
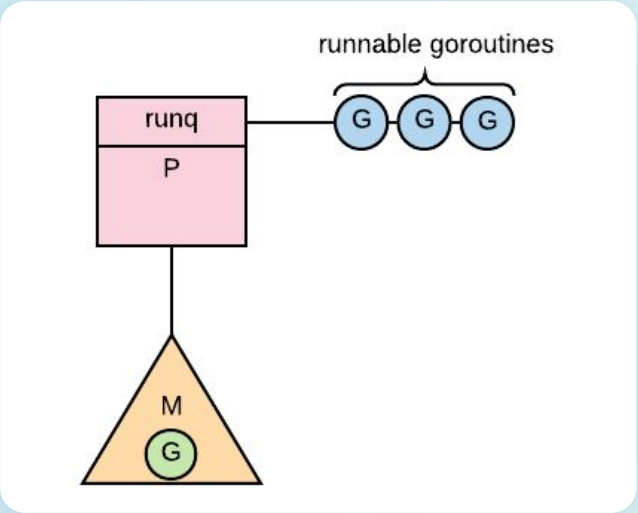
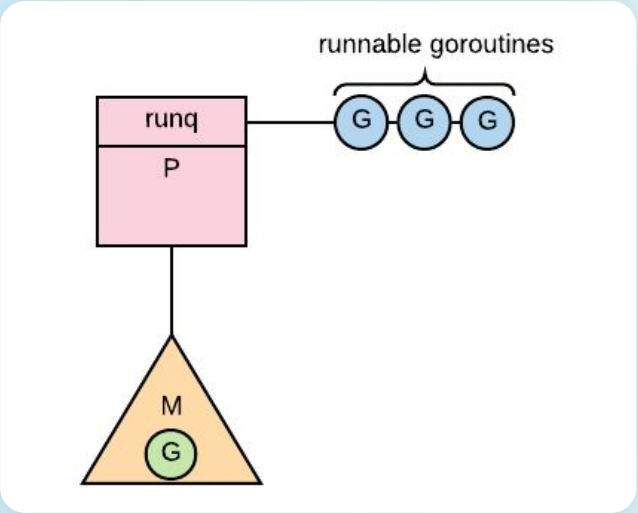
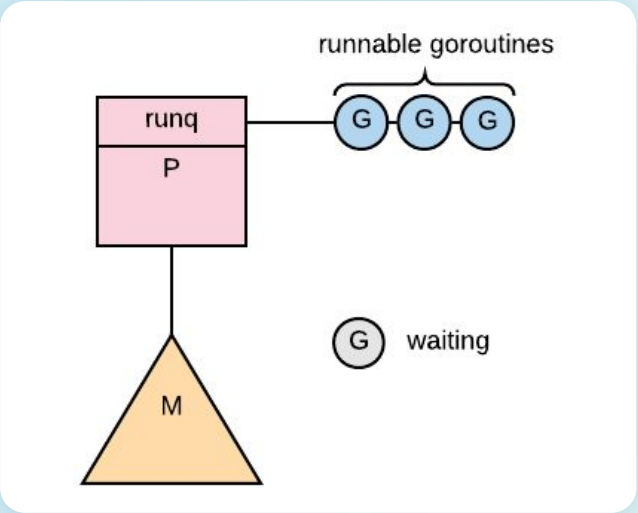
Go’s runtime scheduler (actions)

- Pushing goroutine to the *runq*



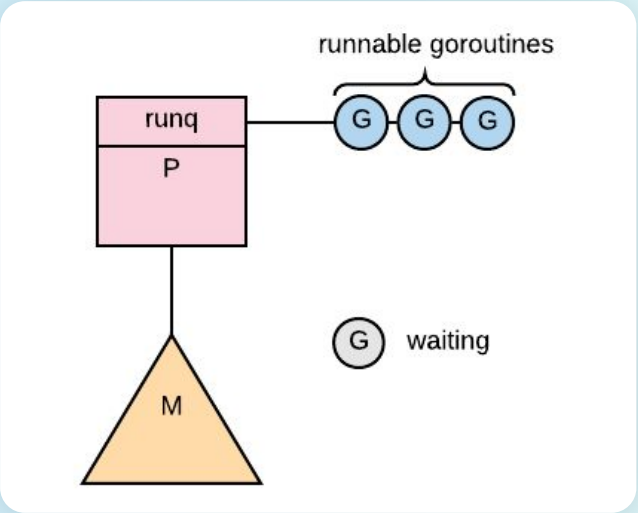
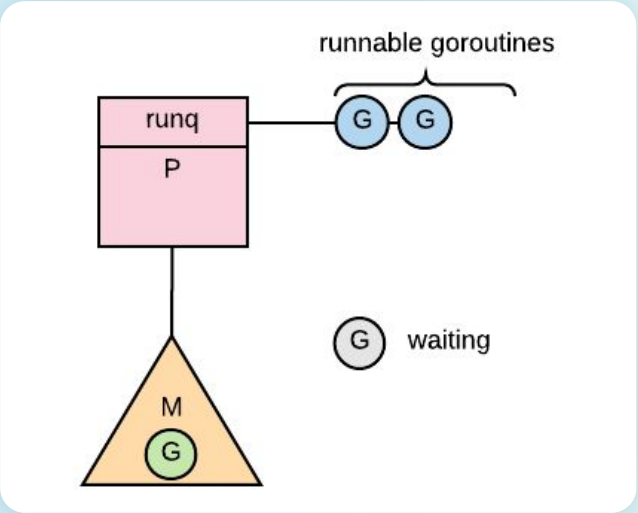
Go’s runtime scheduler (actions)

- Pushing goroutine to the *runq*
  - ```
    go myFunc()
    ```


    ```

Go’s runtime scheduler (actions)

- Pushing goroutine to the *runq*
  - ```
    go myFunc()
    ```
  - ```
    goready(*g, ...)
    ```



Go’s runtime scheduler (actions)

- Pushing goroutine to the *runq*
  - ```
    go myFunc()
    ```
  - ```
    goready(*g, ...)
    ```
- Pausing a running goroutine



Go’s runtime scheduler (actions)

- Pushing goroutine to the *runq*
  - ```
    go myFunc()
    ```
  - ```
    goready(*g, ...)
    ```
- Pausing a running goroutine
  - ```
    gopark(..., *g, ...)
    ```


    ```

Go’s runtime scheduler (actions)

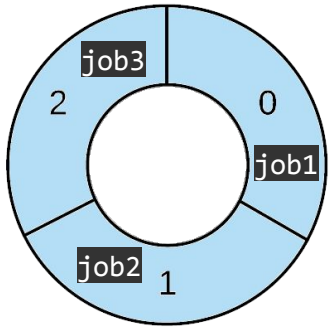
- Pushing goroutine to the *runq*
  - ```
    go myFunc()
    ```
  - ```
    goready(*g, ...)
    ```
- Pausing a running goroutine
  - ```
    gopark(..., *g, ...)
    ```
- Scheduling event


    ```

Go’s runtime scheduler (actions)

- Pushing goroutine to the *runq*
  - ```
    go myFunc()
    ```
  - ```
    goready(*g, ...)
    ```
- Pausing a running goroutine
  - ```
    gopark(..., *g, ...)
    ```
- Scheduling event
  - “context switch”


    ```

Send & Receive blocking nature

Gmain **Gwork**

```
ch <- job4
```

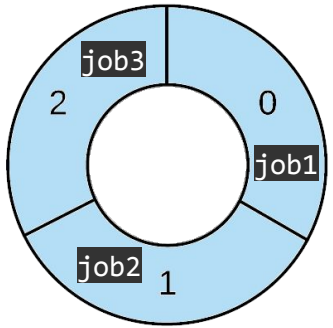


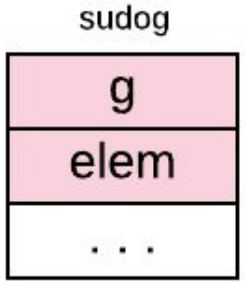
Send & Receive blocking nature

Gmain

```
ch <- job4
```

Gwork





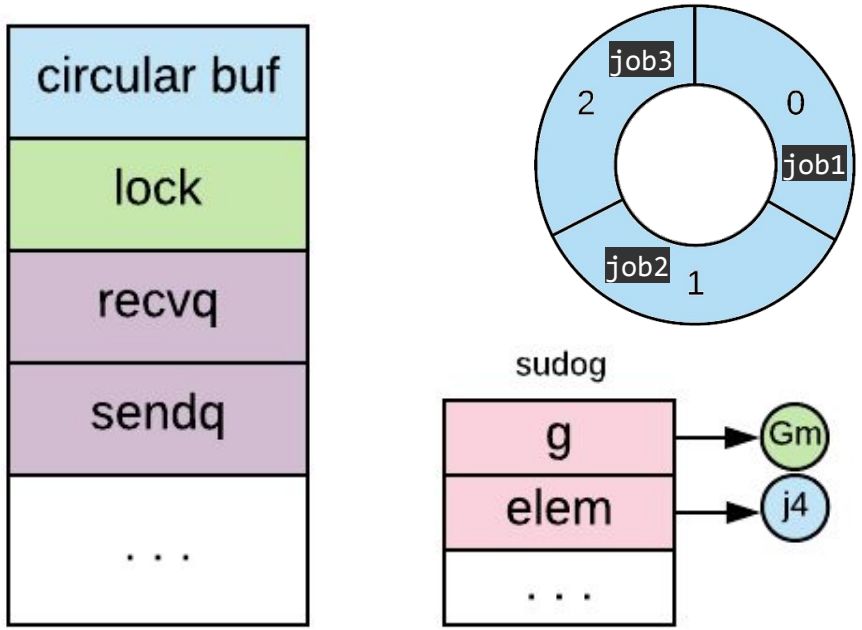
Send & Receive blocking nature

Gmain **Gwork**

```
ch <- job4
```

creates a *sudog* struct

Gm

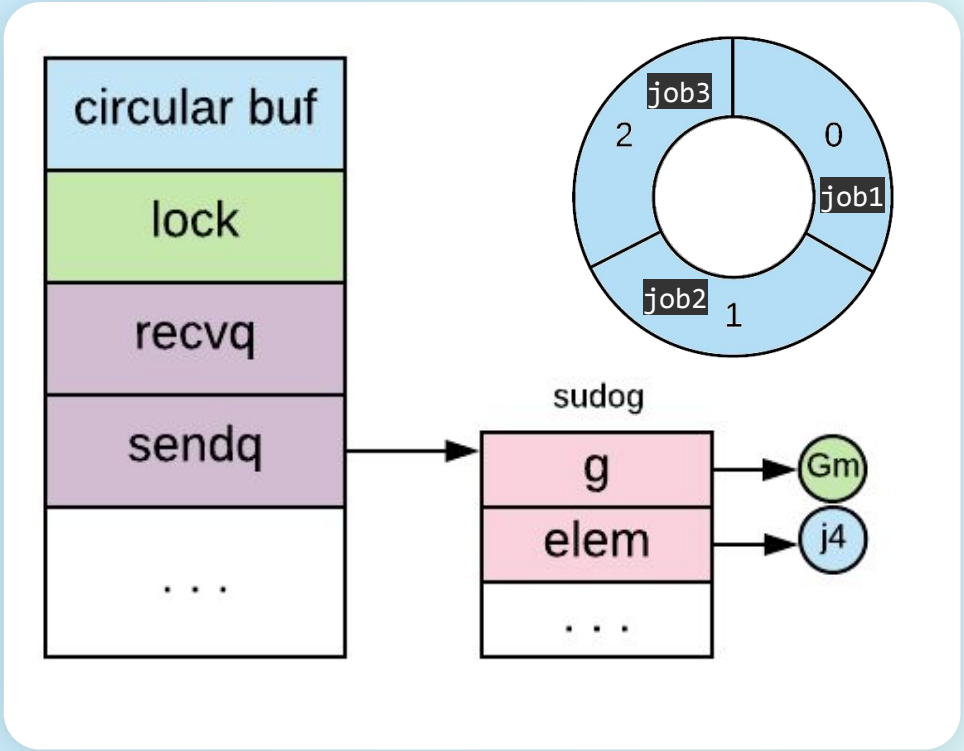


Send & Receive blocking nature

Gmain **Gwork**

```
ch <- job4
```

Gm
- creates a *sudog* struct
- pushes it to the *sendq*

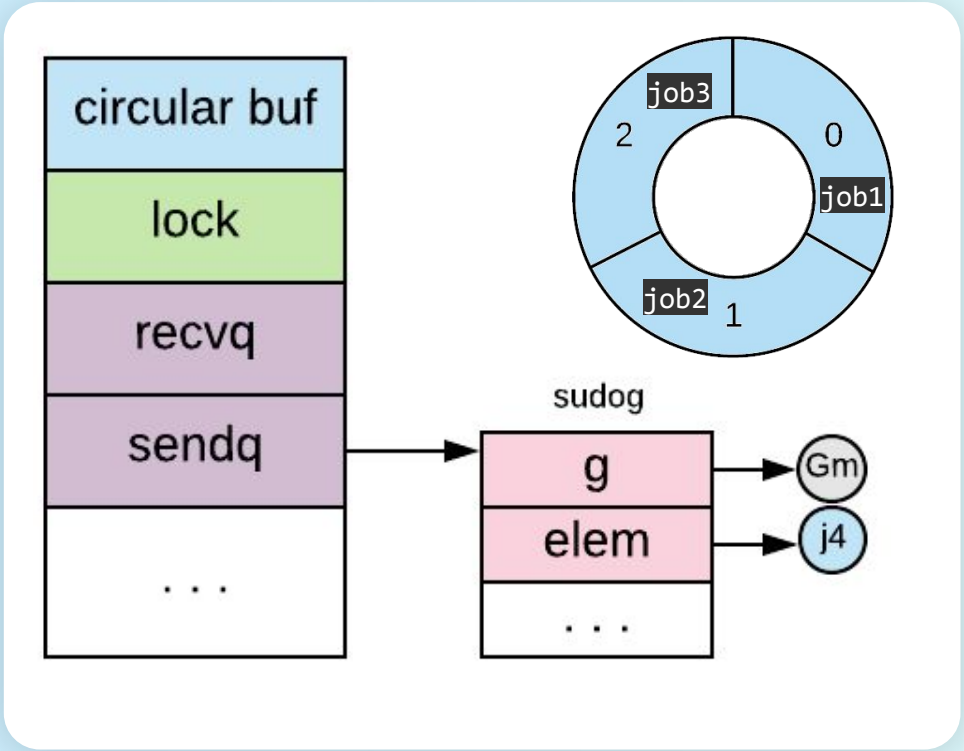


Send & Receive blocking nature

Gmain **Gwork**

```
ch <- job4
```

Gm
- creates a *sudog* struct
- pushes it to the *sendq*
- calls *gopark(Gm)* - gets paused



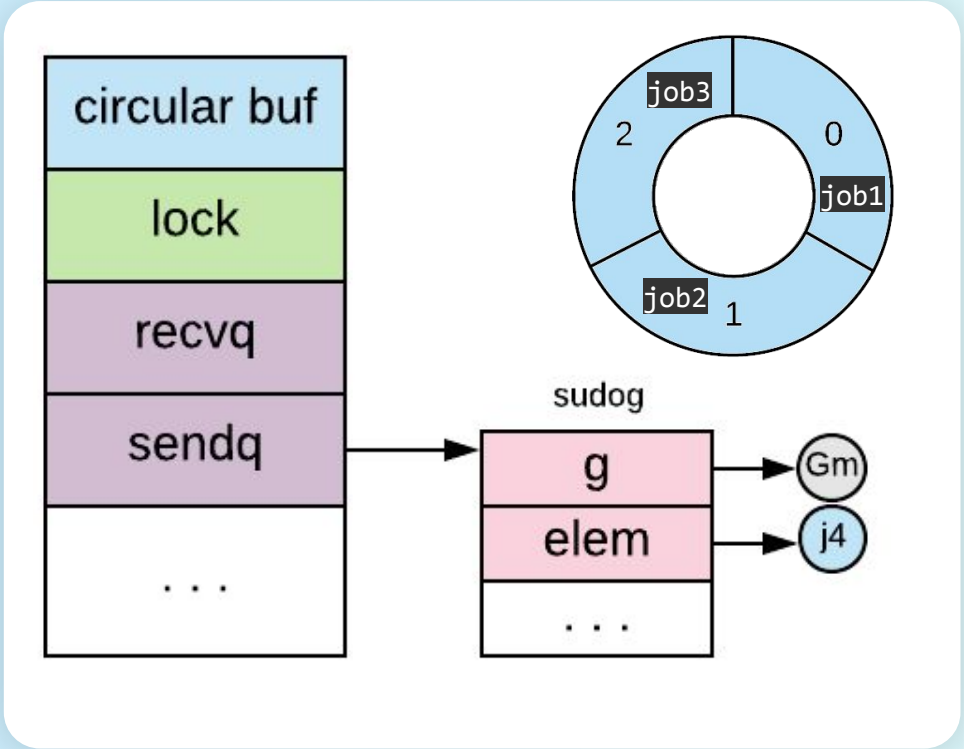
Send & Receive blocking nature

Gmain **Gwork**

```
ch <- job4
```

Gm
- creates a *sudog* struct
- pushes it to the *sendq*
- calls *gopark(Gm)* - gets paused

```
job := <-ch
```



Send & Receive blocking nature

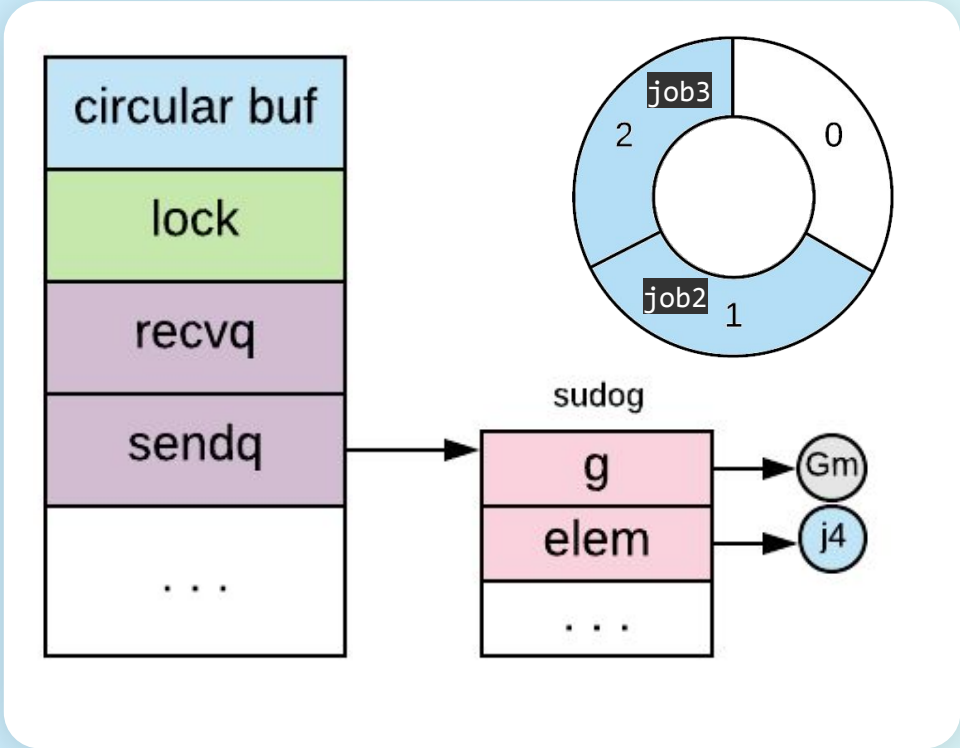
Gmain **Gwork**

```
ch <- job4
```

Gm
- creates a *sudog* struct
- pushes it to the *sendq*
- calls *gopark(Gm)* - gets paused

```
job := <-ch
```

Gw
- dequeues *job1* from the buffer



Send & Receive blocking nature

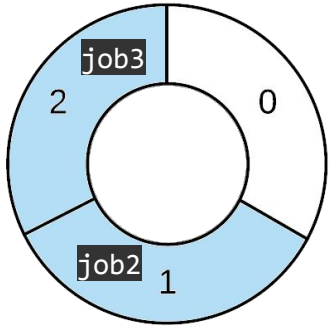
Gmain **Gwork**

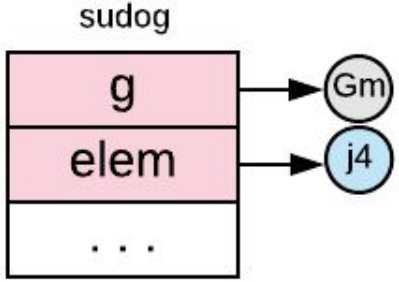
```
ch <- job4
```

Gm
- creates a *sudog* struct
- pushes it to the *sendq*
- calls *gopark(Gm)* - gets paused

```
job := <-ch
```

Gw
- dequeues *job1* from the buffer
- pops a *sudog* from *sendq*





Send & Receive blocking nature

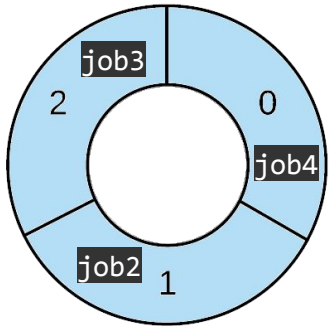
Gmain **Gwork**

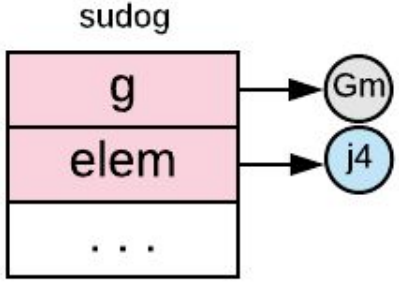
```
ch <- job4
```

Gm
- creates a *sudog* struct
- pushes it to the *sendq*
- calls *gopark(Gm)* - gets paused

```
job := <-ch
```

Gw
- dequeues *job1* from the buffer
- pops a *sudog* from *sendq*
- enqueues *job4* to the buffer





Send & Receive blocking nature

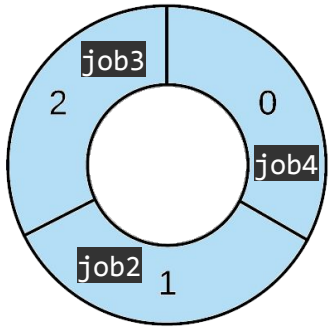
Gmain **Gwork**

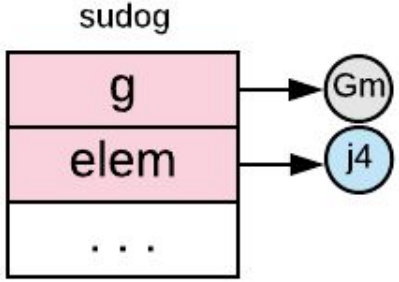
```
ch <- job4
```

Gm
- creates a *sudog* struct
- pushes it to the *sendq*
- calls *gopark(Gm)* - gets paused

```
job := <-ch
```

Gw
- dequeues *job1* from the buffer
- pops a *sudog* from *sendq*
- enqueues *job4* to the buffer
- calls *goready(Gm)* - gets runnable





Send & Receive blocking nature

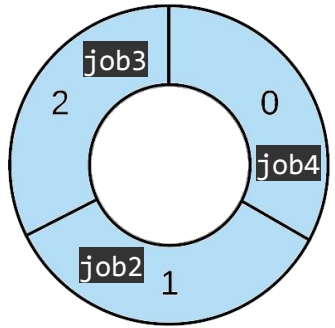
Gmain **Gwork**

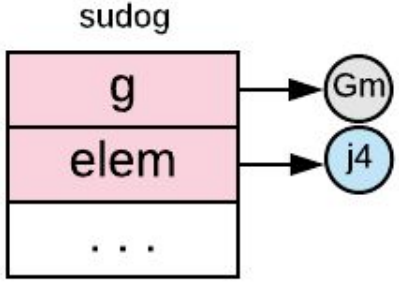
```
ch <- job4
```

Gm
- creates a *sudog* struct
- pushes it to the *sendq*
- calls *gopark(Gm)* - gets paused

```
job := <-ch
```

Gw
- dequeues *job1* from the buffer
- pops a *sudog* from *sendq*
- enqueues *job4* to the buffer
- calls *goready(Gm)* - gets runnable
- resumes (processing *job1*)





Send & Receive blocking nature

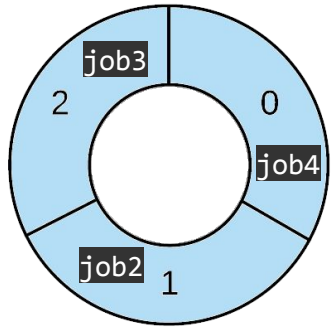
Gmain **Gwork**

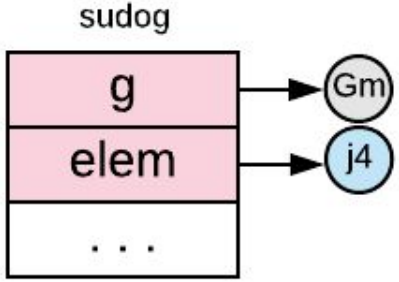
```
ch <- job4
```

Gm
- creates a *sudog* struct
- pushes it to the *sendq*
- calls *gopark(Gm)* - gets paused

```
job := <-ch
```

Gw
- dequeues *job1* from the buffer
- pops a *sudog* from *sendq*
- enqueues *job4* to the buffer
- calls *goready(Gm)* - gets runnable
- resumes (processing *job1*)





Send & Receive blocking nature

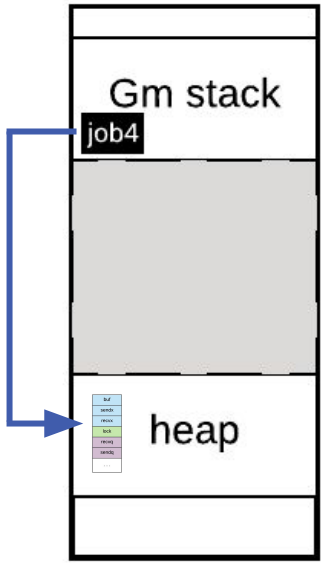
Gmain **Gwork**

```
ch <- job4
```

Gm
- creates a *sudog* struct
- pushes it to the *sendq*
- calls *gopark(Gm)* - gets paused

```
job := <-ch
```

Gw
- dequeues *job1* from the buffer
- pops a *sudog* from *sendq*
- enqueues *job4* to the buffer
- calls *goready(Gm)* - gets runnable
- resumes (processing *job1*)

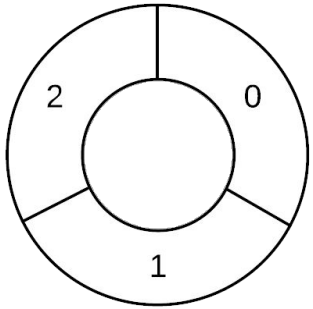


Send & Receive blocking nature

Gmain

Gwork

```
job := <-ch
```

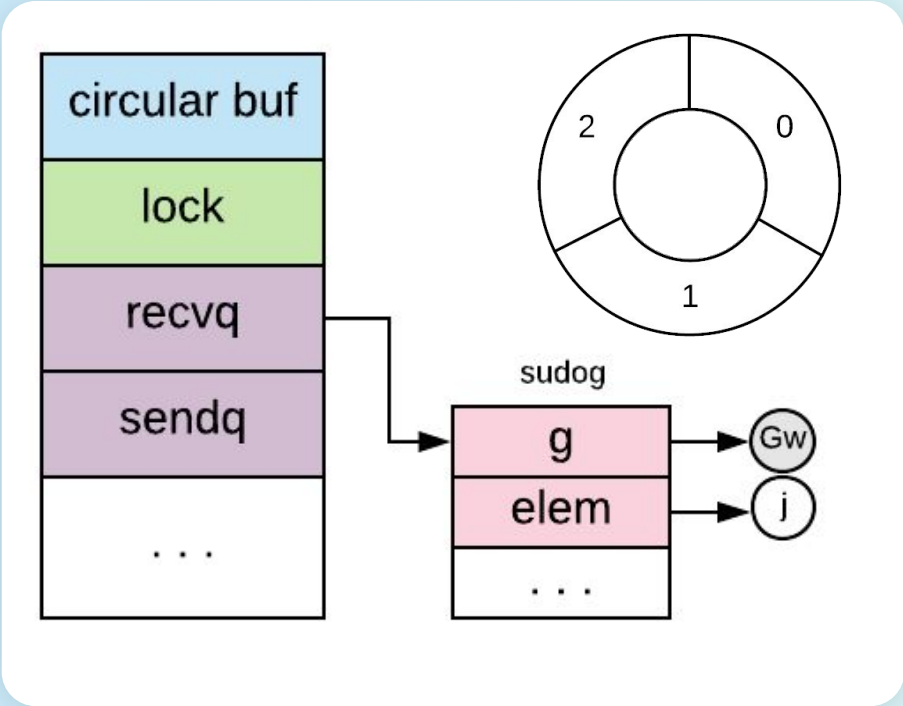


Send & Receive blocking nature

Gmain **Gwork**

```
job := <-ch
```

Gw
- creates a *sudog* struct
- pushes it to the *recvq*
- calls *gopark(Gw)* - gets paused



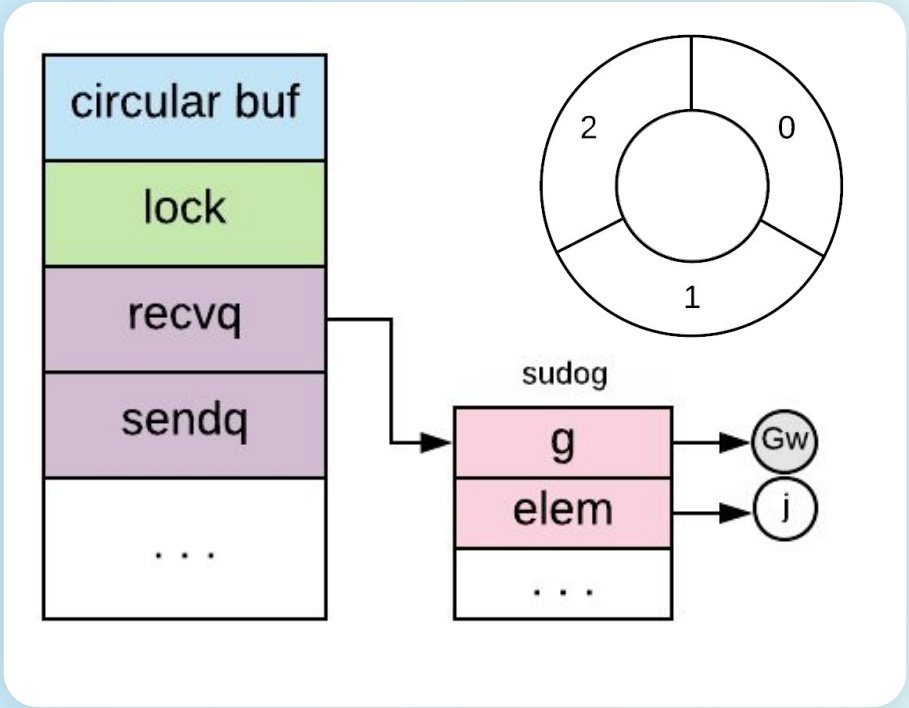
Send & Receive blocking nature

Gmain **Gwork**

```
job := <-ch
```

Gw
- creates a *sudog* struct
- pushes it to the *recvq*
- calls *gopark(Gw)* - gets paused

```
ch <- job1
```



Send & Receive blocking nature

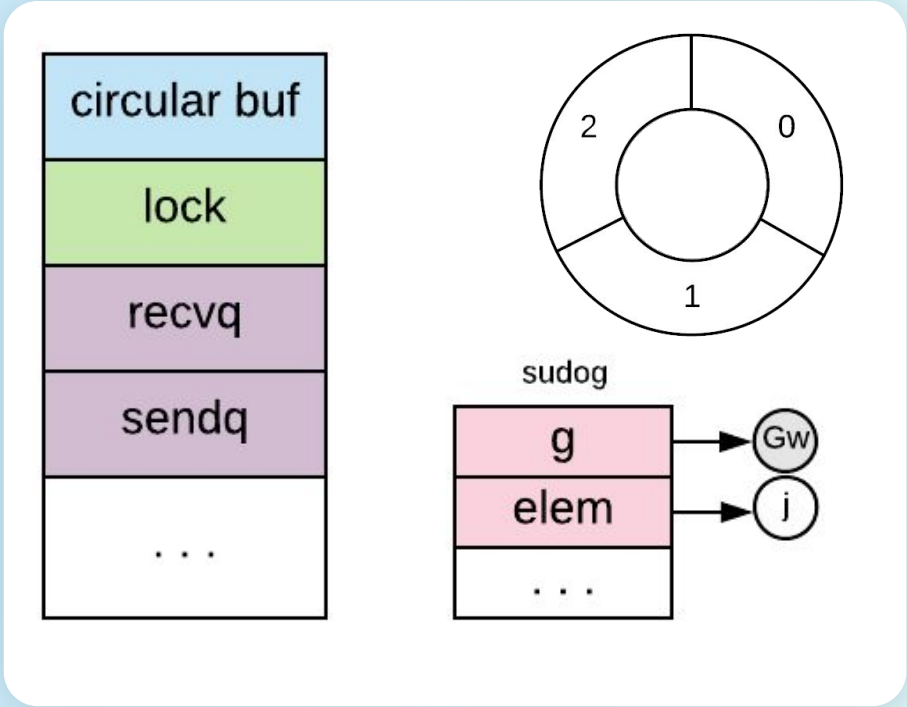
Gmain **Gwork**

```
job := <-ch
```

Gw
- creates a *sudog* struct
- pushes it to the *recvq*
- calls *gopark(Gw)* - gets paused

```
ch <- job1
```

Gm
- pops a *sudog* from *recvq*



Send & Receive blocking nature

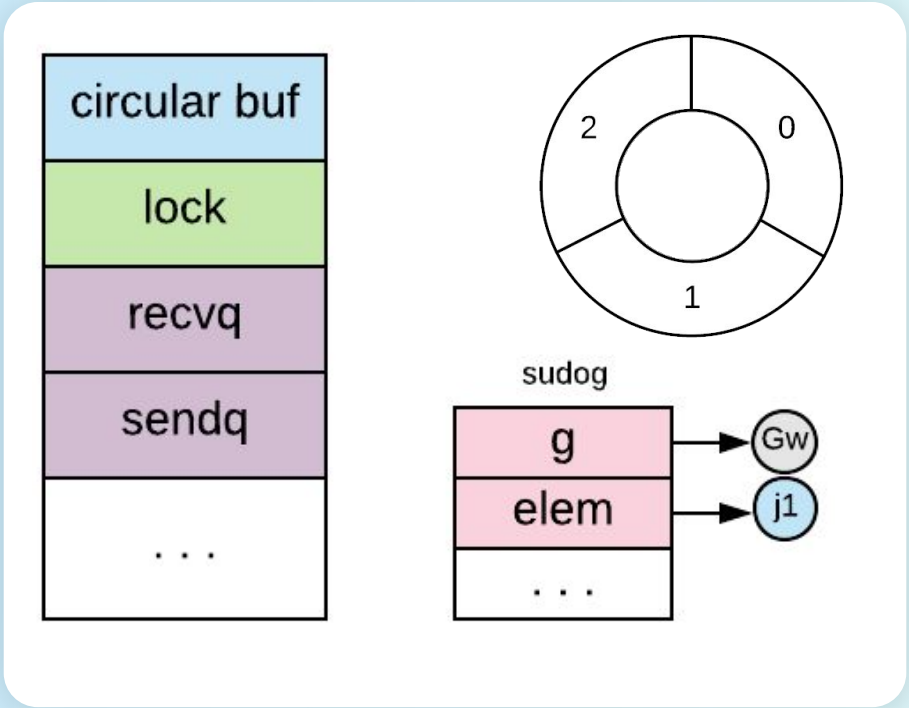
Gmain **Gwork**

```
job := <-ch
```

Gw
- creates a *sudog* struct
- pushes it to the *recvq*
- calls *gopark(Gw)* - gets paused

```
ch <- job1
```

Gm
- pops a *sudog* from *recvq*
- copies *job1* to the *job* variable itself



Send & Receive blocking nature

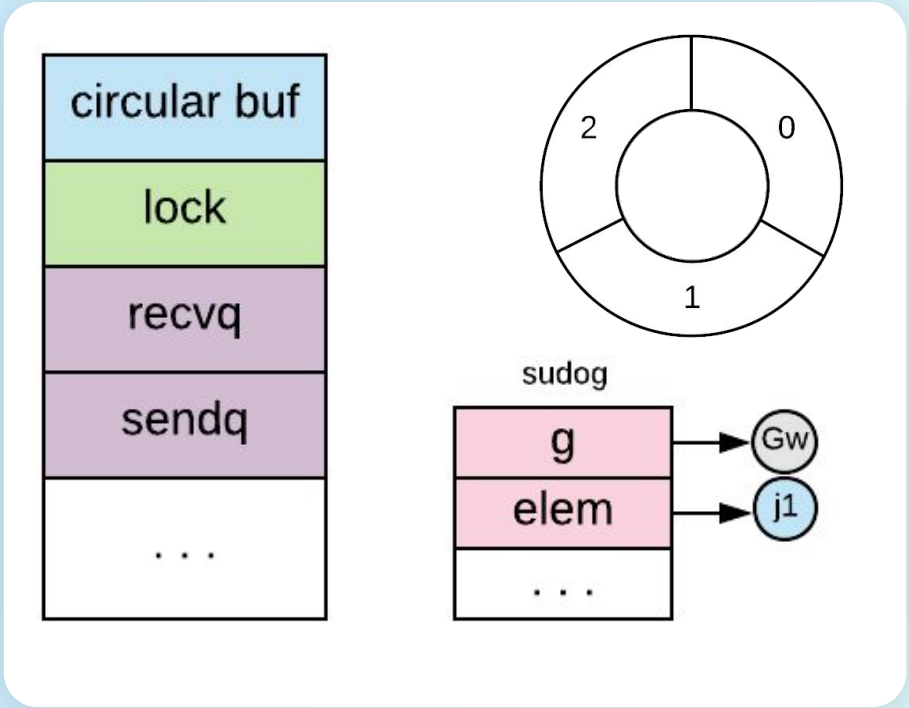
Gmain **Gwork**

```
job := <-ch
```

Gw
- creates a *sudog* struct
- pushes it to the *recvq*
- calls *gopark(Gw)* - gets paused

```
ch <- job1
```

Gm
- pops a *sudog* from *recvq*
- copies *job1* to the *job* variable itself
- calls *goready(Gw)* - gets runnable



Send & Receive blocking nature

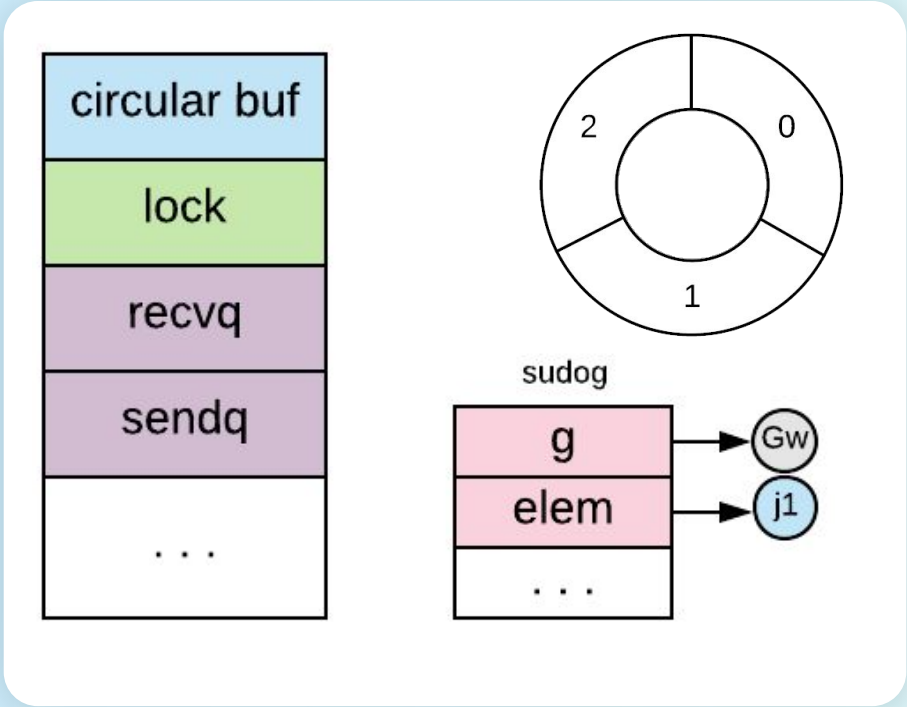
Gmain **Gwork**

```
job := <-ch
```

Gw
- creates a *sudog* struct
- pushes it to the *recvq*
- calls *gopark(Gw)* - gets paused

```
ch <- job1
```

Gm
- pops a *sudog* from *recvq*
- copies *job1* to the *job* variable itself
- calls *goready(Gw)* - gets runnable
- resumes



Send & Receive blocking nature

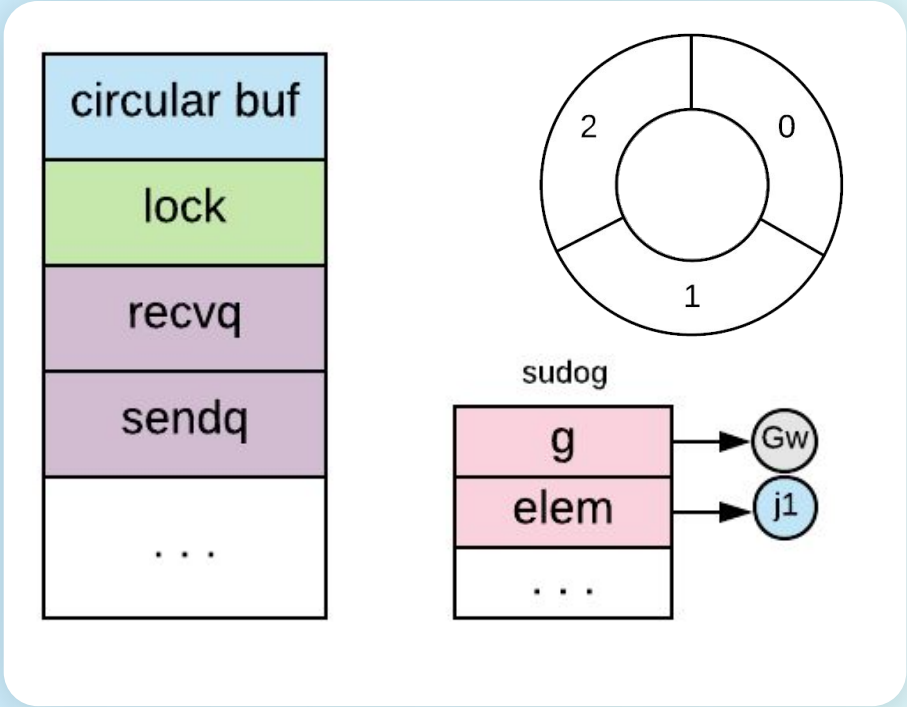
Gmain **Gwork**

```
job := <-ch
```

Gw
- creates a *sudog* struct
- pushes it to the *recvq*
- calls *gopark(Gw)* - gets paused

```
ch <- job1
```

Gm
- pops a *sudog* from *recvq*
- copies *job1* to the *job* variable itself
- calls *goready(Gw)* - gets runnable
- resumes



Send & Receive blocking nature

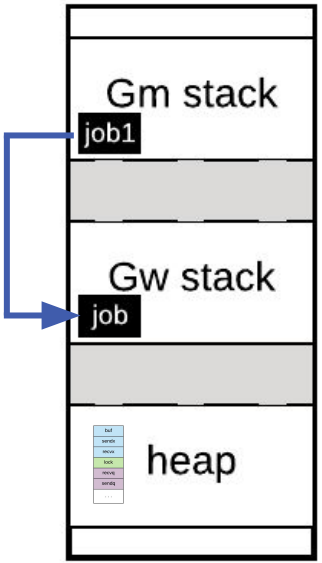
Gmain **Gwork**

```
job := <-ch
```

Gw
- creates a *sudog* struct
- pushes it to the *recvq*
- calls *gopark(Gw)* - gets paused

```
ch <- job1
```

Gm
- pops a *sudog* from *recvq*
- copies *job1* to the *job* variable itself
- calls *goready(Gw)* - gets runnable
- resumes



Wrapping it up...

- Go is an outcome of a need
- Go provides built-in concurrency mechanisms
- Channels are complicated machinery with simple API
- Delicate balance between simplicity and performance

References

- Concurrency is not Parallelism (Rob Pike)
- gobyexample - goroutines
- gobyexample - channels
- Circular Buffer
- chan.go (search: “type hchan struct”)
- Share memory by communicating
- runtime2.go (search: “type sudog struct”)
- Go scheduler
- Advanced concurrency patterns

Thank you

Eran Avidor, TL @ Yotpo

eranavidor.com

eran@yotpo.com

@_eranavidor