Overview of Concurrency

Concurrency

- Concurrency is about **multiple things happening at same time** in random order.

- Go provides a **built-in support for concurrency.**

Why we need to think about concurrency?

```go
// Add - sequential code to add numbers
func Add(numbers []int) int64 {
    var sum int64
    for _, n := range numbers {
        sum += int64(n)
    }
    return sum
}
```

How can we make this function run faster?

Computing Environment

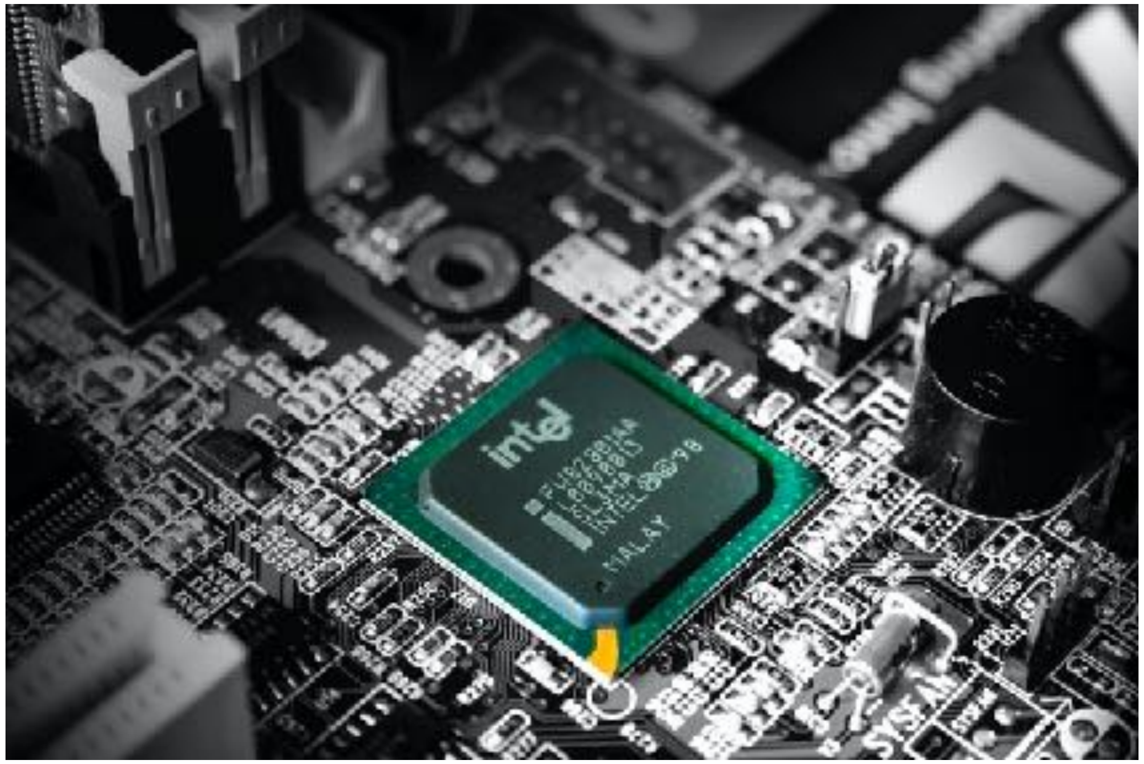
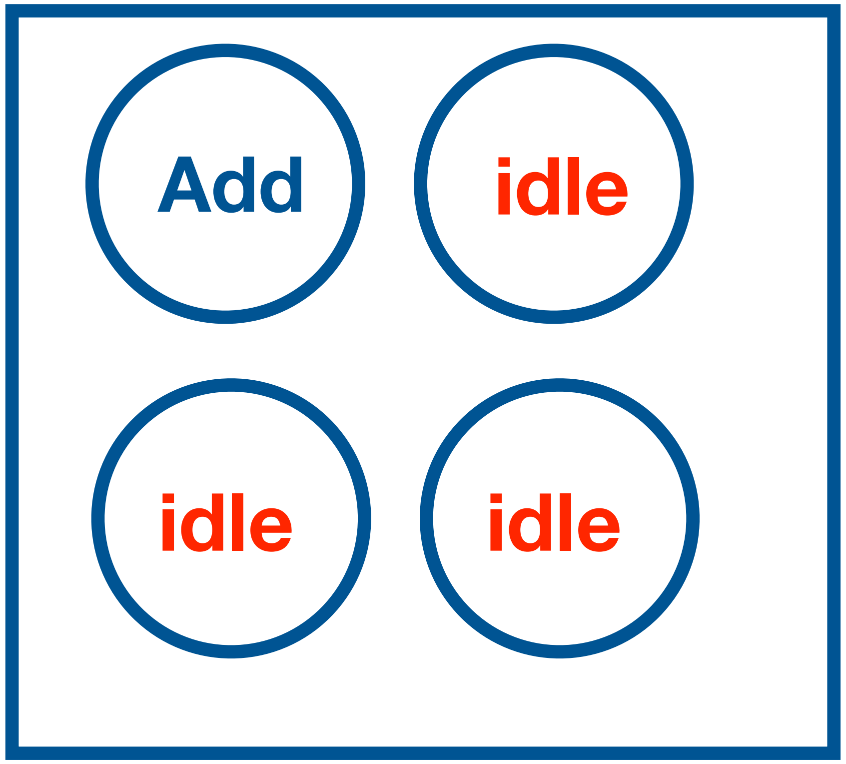


Photo by Slejven Djurakovic on Unsplash

Multi-Core Processor



- When Add() is executed it runs on single core.

Computing Environment

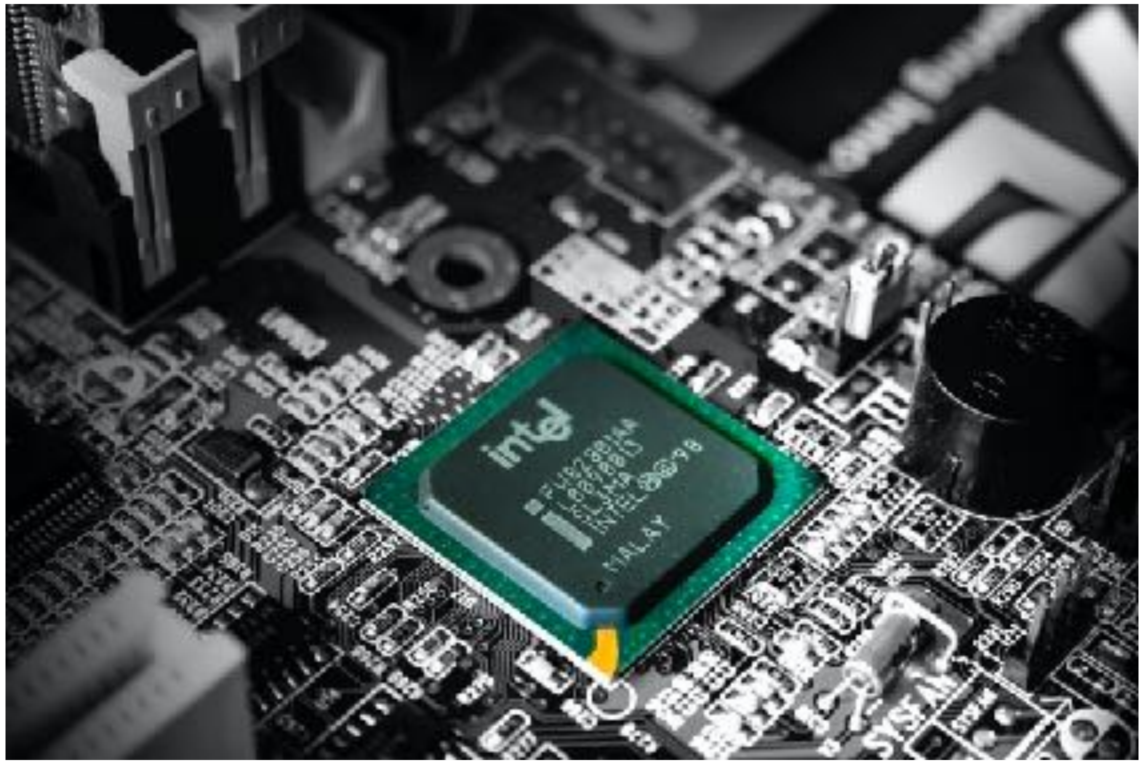
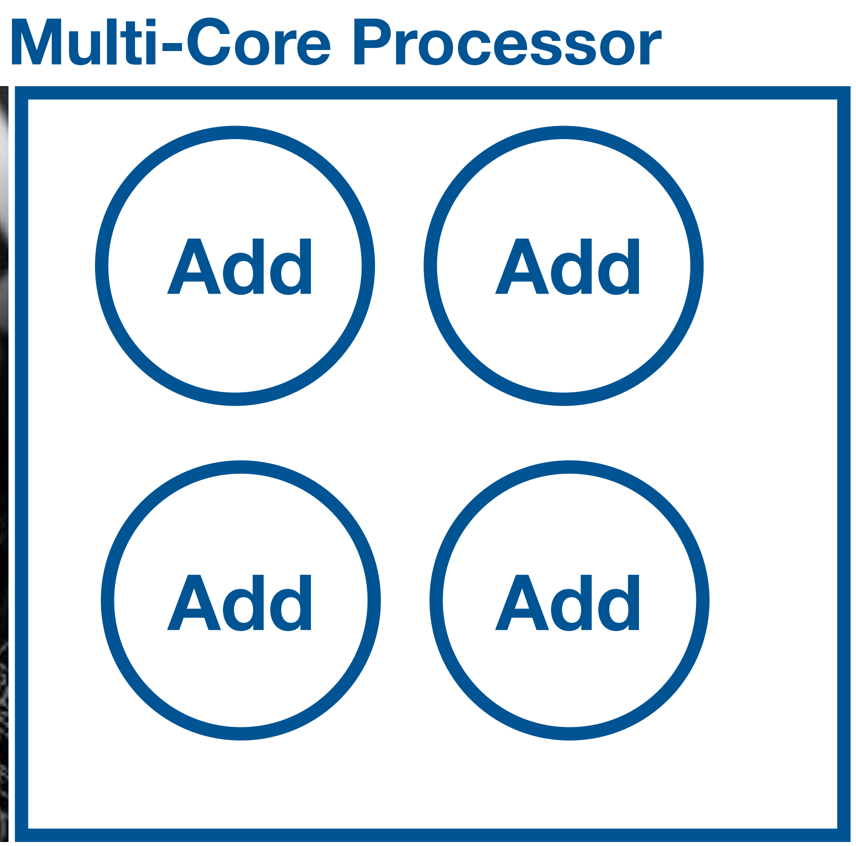


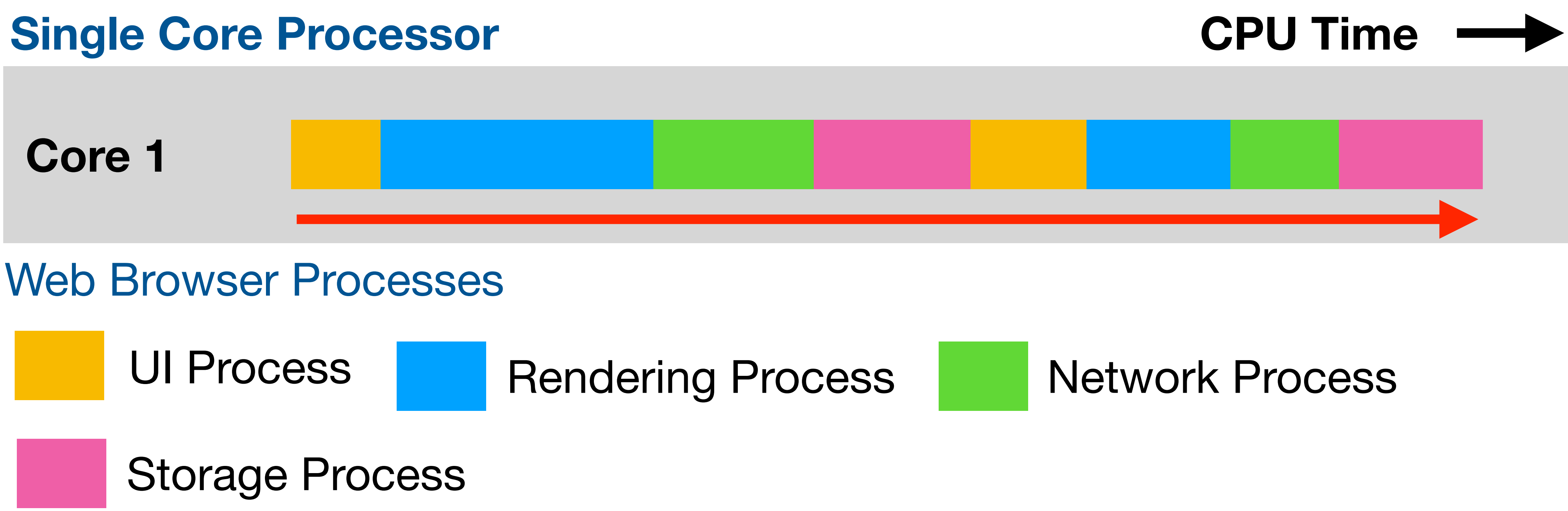
Photo by Slejven Djurakovic on Unsplash



- Divide the input and run multiple instances of Add() function on each part in parallel on different core.

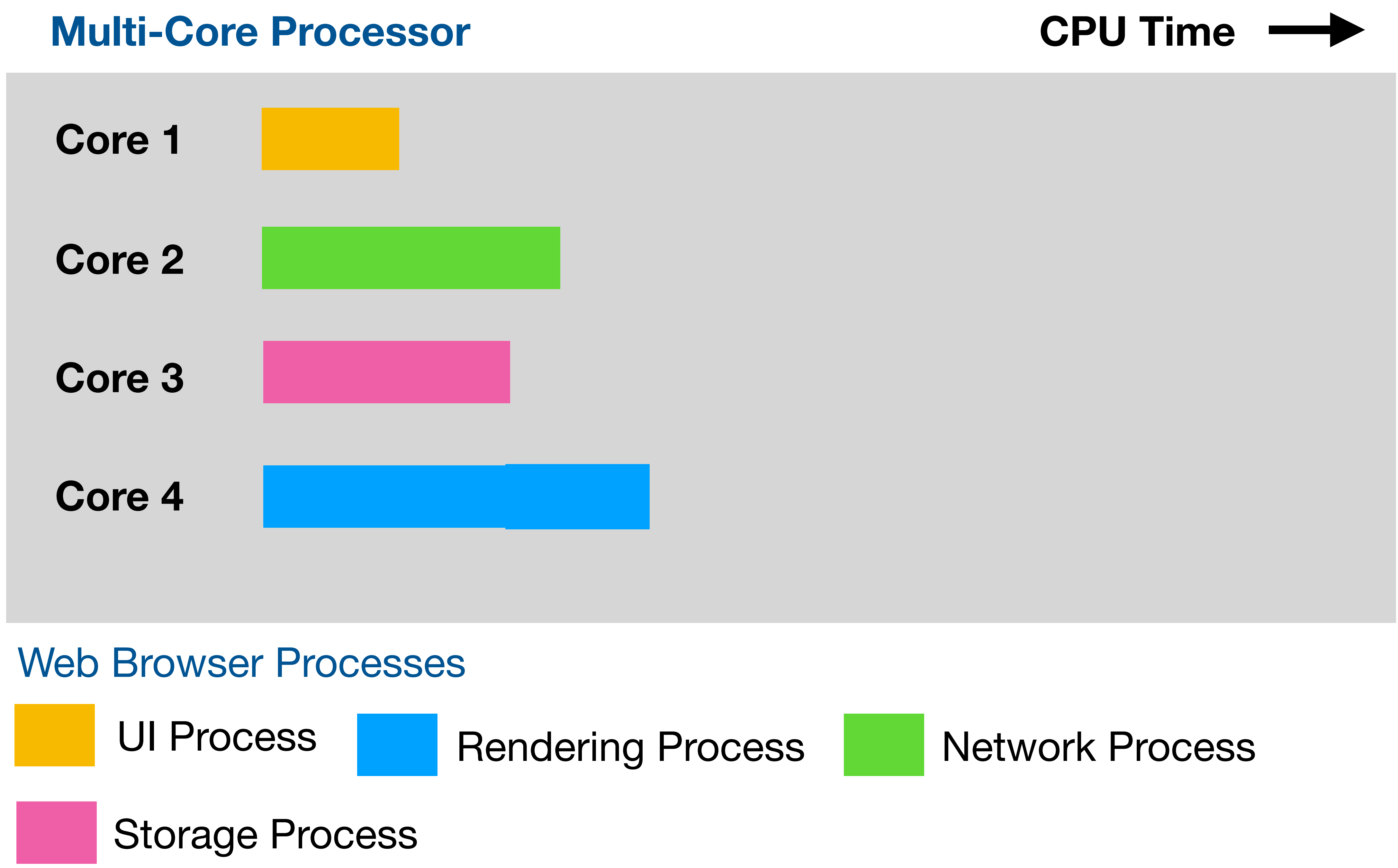
Concurrency

- Concurrency is composition of **independent execution computations**, which may or may not run in parallel.



Parallelism

- Parallelism is ability to **execute multiple computations simultaneously.**



Summary

Why we need to think about Concurrency?

- In order to run faster, application needs to be **divided into multiple independent units** and run them in parallel.

Summary

What is concurrency?

- Concurrency is composition of independent execution of computations.

Summary

What is Parallelism?

- Parallelism is ability to execute multiple computations simultaneously.

- Concurrency enables Parallelism.

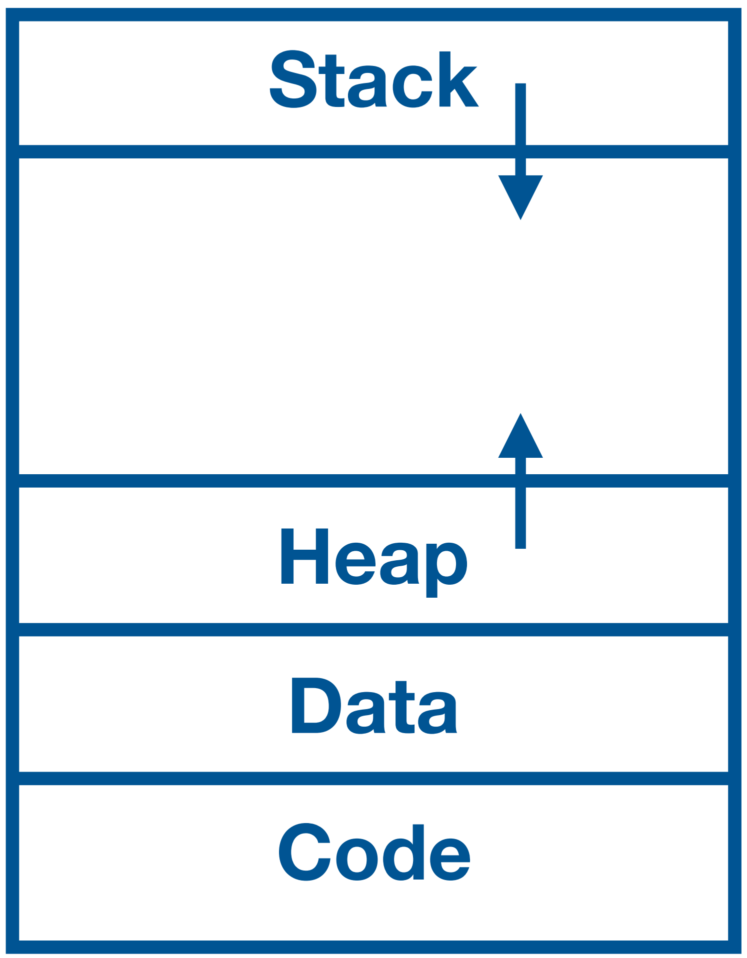
Why there was a need to build concurrency primitives in Go?

Operating System

- The job of operating system is to give fair chance for all processes access to CPU, memory and other resources.

What is a Process

- An instance of a running program is called a process.
- Process provides environment for program to execute.

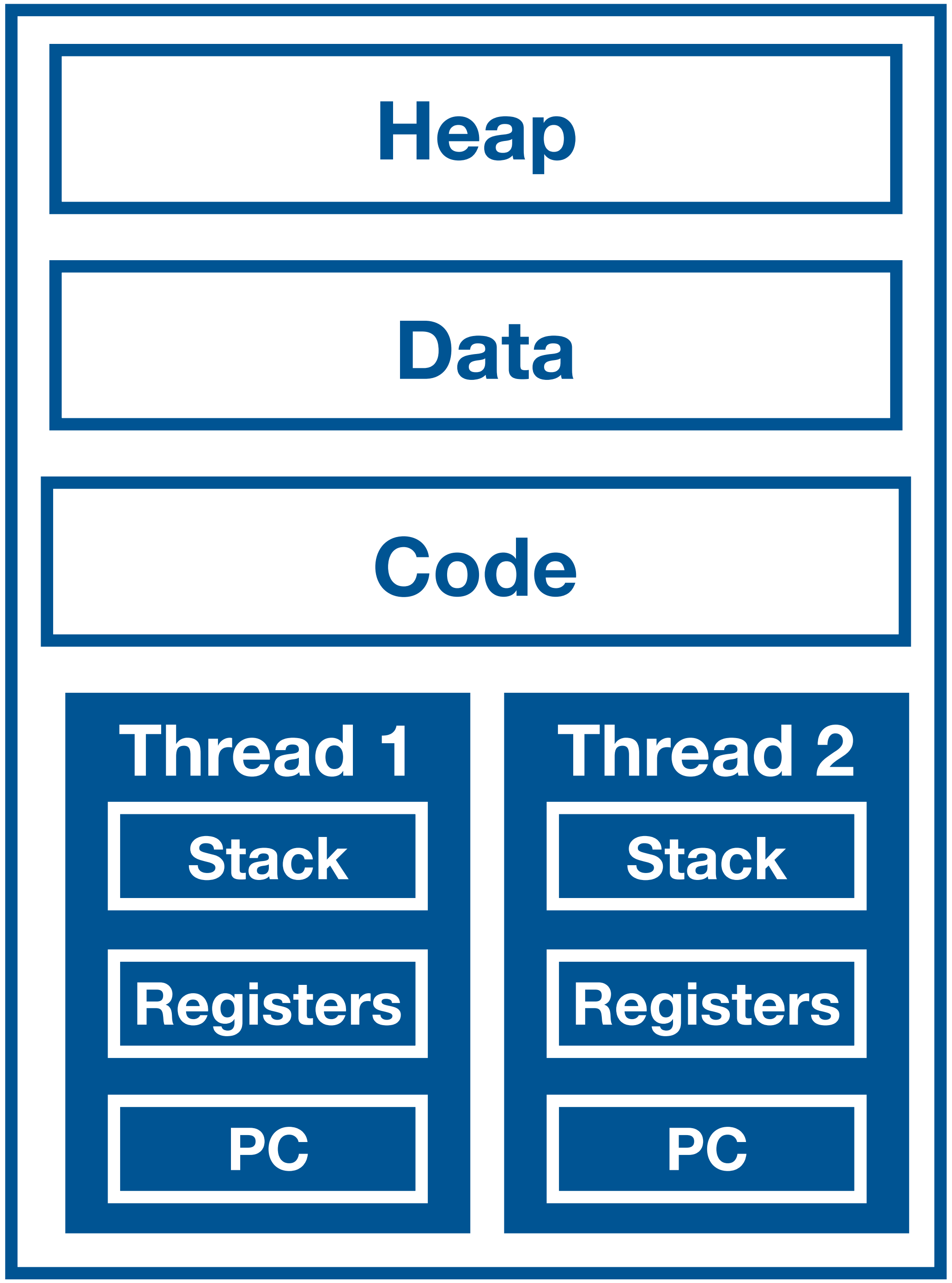


- OS allocates memory.

- Code - machine instructions
- Data - Global data
- Heap - Dynamic memory allocation
- Stack - Local variables of function

Threads

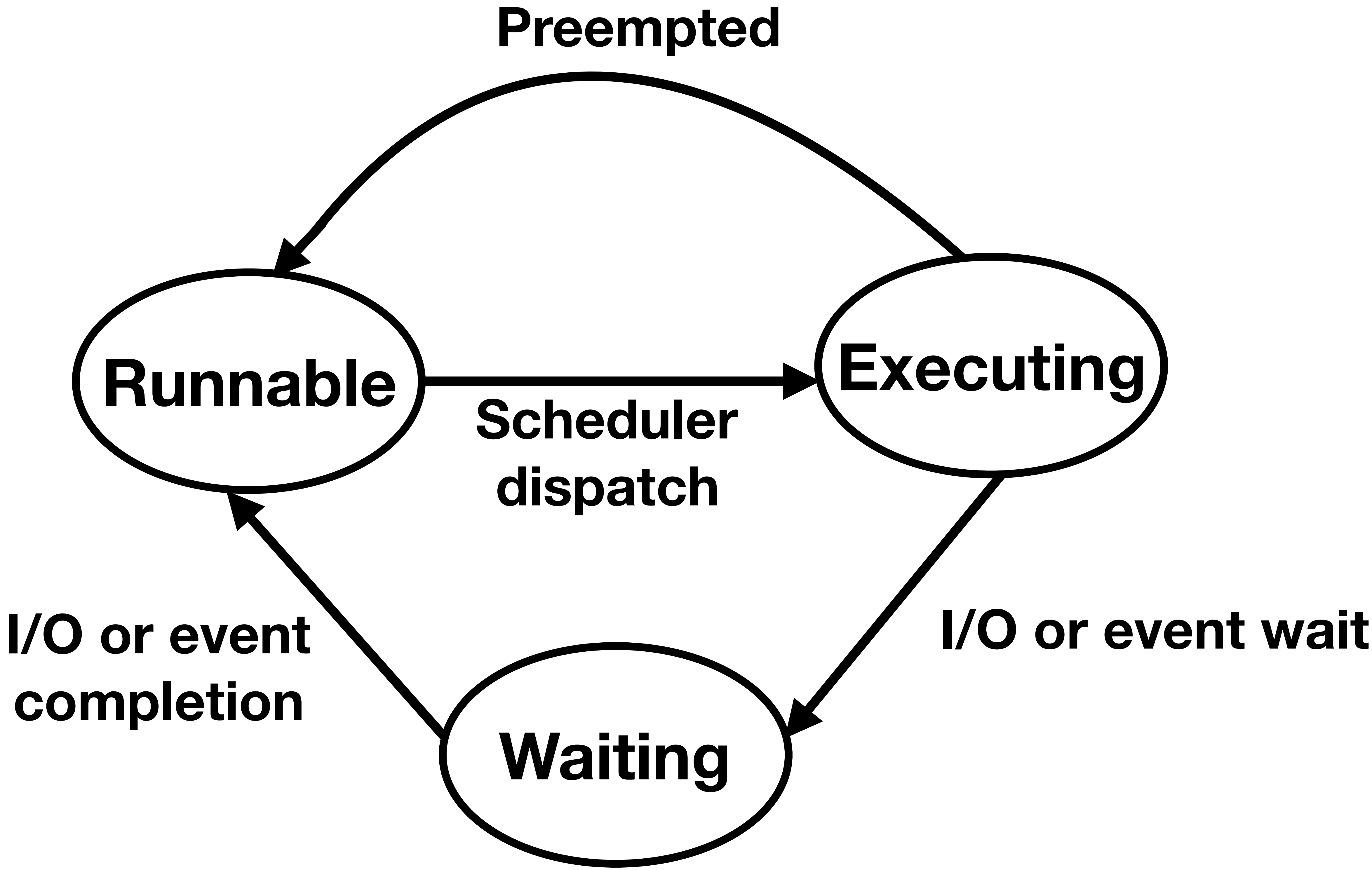
- Threads are **smallest unit of execution** that CPU accepts.
- Process has atleast one thread - main thread.
- Process can have multiple threads.
- Threads share same address space.



Threads

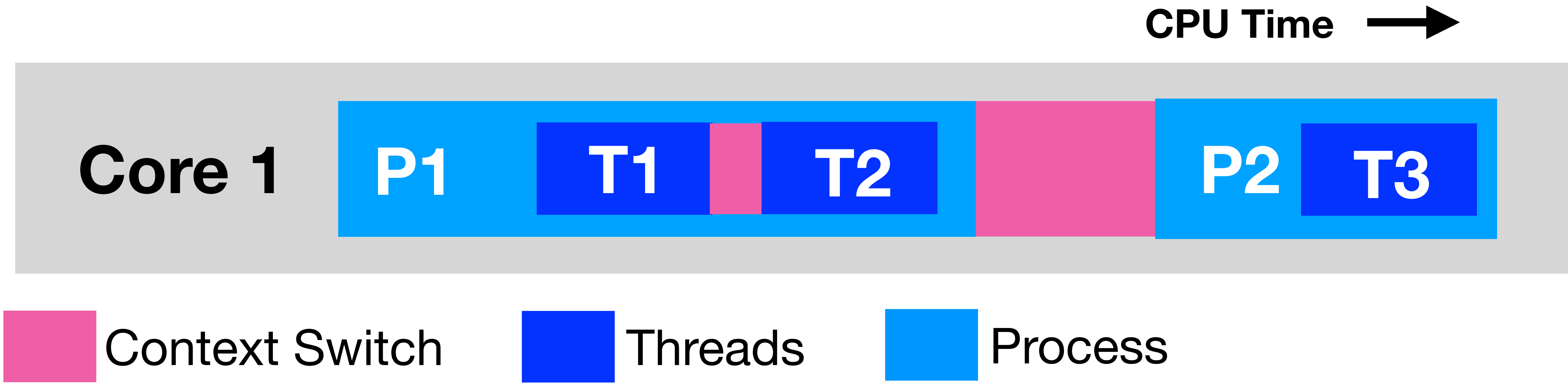
- Threads run independent of each other.
- OS scheduler makes scheduling decisions at thread level, not process level.
- Threads can run concurrently or in parallel.

Thread States



Can we divide our application into Processes and Threads and achieve Concurrency?

Context Switches are expensive



Process Context

- Process state
- CPU scheduling Information
- Memory management information
- Accounting information
- I/O Status information

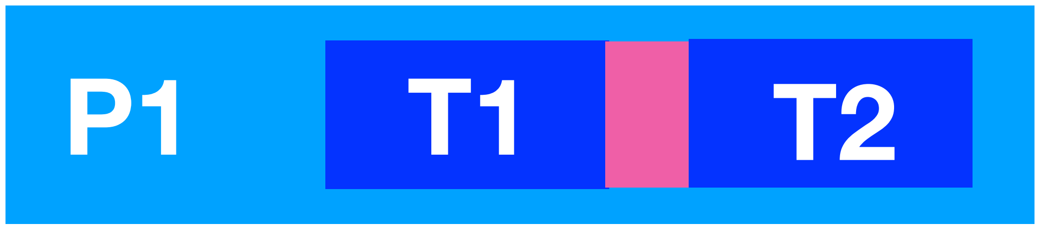
Thread Context

- Program counter
- CPU registers
- Stack

C10k Problem

- Scheduler allocates a process a time slice for execution on CPU core.

- This CPU time slice is divided equally among threads.



C10k Problem

Scheduler Period	Number of threads	Thread time slice
10ms	2	5ms
10ms	5	2ms
10ms	1000	10us ?

C10k Problem

- If minimum time for thread is slice is 2ms.

Scheduler Period	Number of threads	Thread time slice
2s	1000	2ms
20s	10,000	2ms

- It can take 20s scheduler cycle for 10,000 threads.

Fixed Stack Size

- Threads are allocated fixed stack size (on my machine it is 8MB)

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) unlimited
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 256
pipe size            (512 bytes, -p) 1
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 709
virtual memory          (kbytes, -v) unlimited
```

Summary

What is Process?

- An instance of a running program is called a process.
- Process provides environment for program to execute.

Summary

What is a Thread?

- Threads are smallest unit of execution that CPU accepts.
- Process has atleast one thread - main thread.
- Process can have multiple threads.
- Threads share same address space.

Summary

What are the limitations of thread?

- Fixed stack size.

- C10K problem, as we scale up number of threads, scheduler cycle increases and application can become less responsive..

Why Concurrency is hard?

Shared Memory

- Threads communicate between each other by sharing memory.
- sharing of memory between threads creates lot of complexity
- Concurrent access to shared memory by two or more threads can lead to **Data Race** and outcome can be **Un-deterministic**.

Concurrent Access and Atomicity

i = 0

Thread 1 **Thread 2**

i++

- Increment operation is not atomic. It involves,
    - Retrieve the value of *i*
    - Increment the value of *i*
    - Store the value of *i*

Thread 1 Thread 2

i++

		Retrieve	Increment	Store
Scenario 1 Thread1 and Thread2 execute sequentially	Thread 1	0	1	1
	Thread 2	1	2	2
Scenario 2 Thread2 preempts Thread1 before store	Thread 1	0	1	1
	Thread 2	0	1	1

- **On concurrent access to memory leads to un-deterministic outcome.**

Memory Access Synchronization

- We need to guard the access to shared memory so that a thread gets exclusive access at a time.

Thread 1

```
Lock()
    i++
Unlock()
```

Thread 2

```
Lock()
    i++
Unlock()
```

- It is a Developer's convention to lock() and unlock()

- If Developers don't follow this convention, we have no guarantee of exclusive access!

Memory Access Synchronization

- Locking **reduces parallelism**. Locks force to execute sequentially.

- Inappropriate use of locks can lead to **Deadlocks.**

Deadlock

Thread 1

```
v1.Lock()   1
    v2.Lock()   3   Waiting
        WriteToSharedMemory()
    v2.Unlock()
v1.Lock()
```

Thread 2

```
v2.Lock()   2
    v1.Lock()   4   Waiting
        WriteToSharedMemory()
    v1.Unlock()
v2.Lock()
```

- Circular wait leads to **Deadlocks.**

Summary

Why concurrency is hard?

- Sharing of memory between threads creates complexity.

- Concurrent access to shared memory can lead to race conditions and outcomes can be un-deterministic.

- Memory access synchronisation tools reduces parallelism and comes with limitation.

Goroutines

Communicating Sequential Processes (CSP)

- Tony Hoare (1978)

  - Each process is built for sequential execution.

  - Data is communicated between processes.
    No shared memory.

  - Scale by adding more of the same.

Go's Concurrency Tool Set

- goroutines
- channels
- select
- sync package

Goroutines

- We can think Goroutines as **user space threads managed by go runtime.**
- Goroutines extremely lightweight. Goroutines starts with **2KB of stack**, which grows and shrinks as required.
- **Low CPU overhead -** three instructions per function call.
- Can **create hundreds of thousands of goroutines** in the same address space.
- **Channels are used for communication of data** between goroutines. Sharing of memory can be avoided.

Goroutines

- Context switching between goroutines is much **cheaper** than thread context switching.

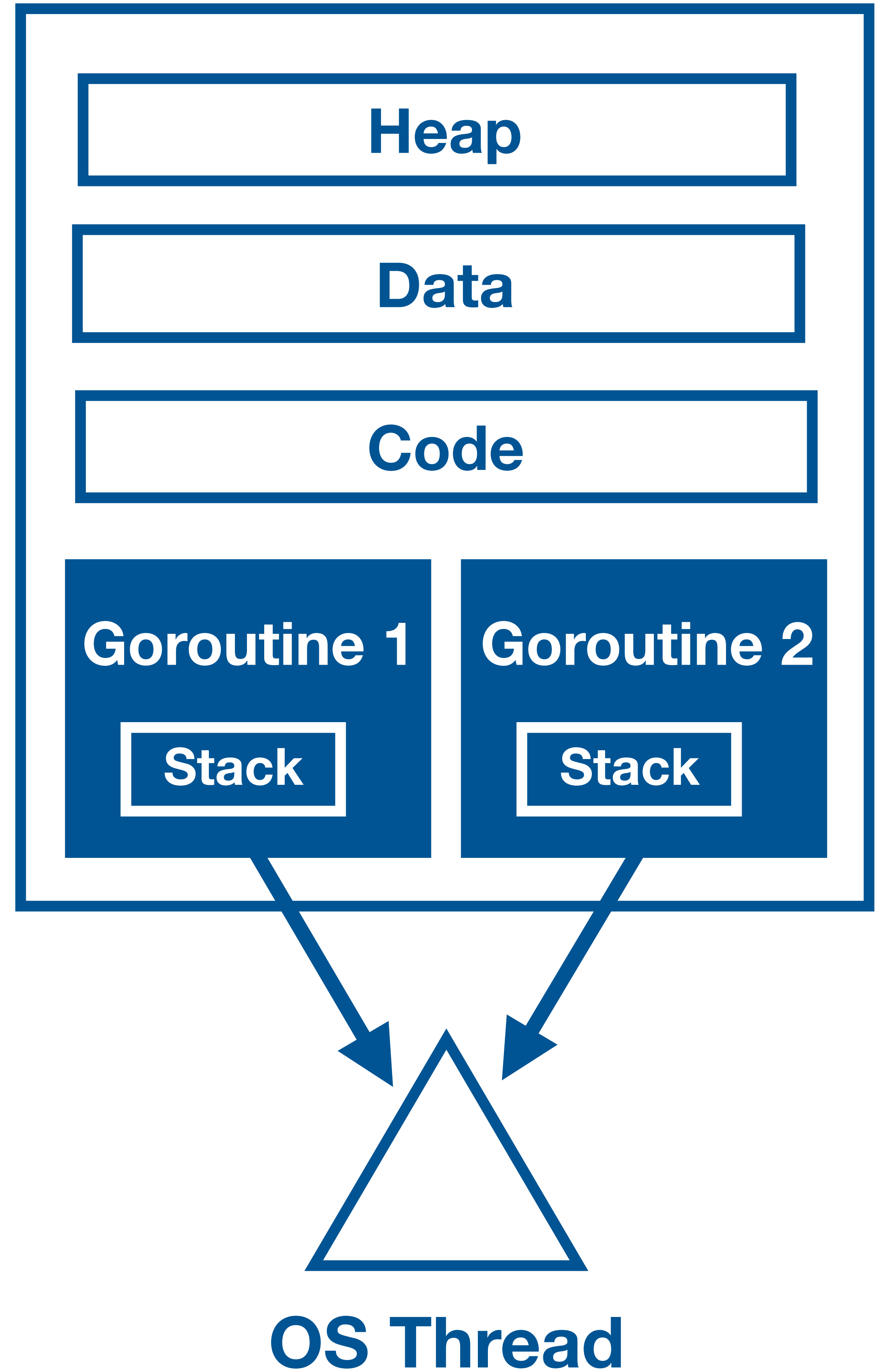
- Go runtime can be more **selective** in **what is persisted** for retrieval, how it is persisted, and when the persisting needs to occur.

Goroutines

- Go runtime creates worker OS threads.

- Goroutines runs in the context of OS thread.

- Many goroutines execute in the context of single OS thread.



Summary

What are Goroutines?

- Goroutines are user space threads managed by go runtime.

Summary

What are advantages of goroutines over OS threads?

- Goroutine are extremely lightweight compared to OS threads.
- Stack size is very small of 2kb as opposed to 8MB of stack of OS threads.
- Context switching is very cheap as it happens in user space, goroutines have very less state to be stored.
- Houndreds of thousands of goroutines can be created on single machine.

sync.WaitGroup

What could be possible outputs of this program?

```go
func main() {
	var data int

	go func() {
		data++
	}()

	if data == 0 {
		fmt.Printf("the value is %v\n", data)
	}
}
```

Race Condition

- Race Condition occurs when **order of execution** is **NOT guaranteed**.

- Concurrent Programs does not execute in the order they are coded.

```go
func main() {
	var data int

	go func() {
		data++
	}()

	if data == 0 {
		fmt.Printf("the value is %v\n", data)
	}
}
```

Race Condition

- Compiler does lot of **optimisation** that changes the order of execution.

What could be possible outputs of this program?

```
5  func main() {
6      var data int
7
8      go func() {
9          data++
10     }()
11
12     if data == 0 {
13         fmt.Printf("the value is %v\n", data)
14     }
15 }
```

Output	Execution sequence
Nothing is printed	Line 9, 12
the value is 0	Line 12, 13
the value is 1	Line 12, 9, 13

What could be possible outputs of this program?

```go
5    func main() {
6        var data int
7
8        go func() {
9            data++
10       }()
11
12       if data == 0 {
13           fmt.Printf("the value is %v\n", data)
14       }
15   }
```

Output	Execution sequence
Nothing is printed	Line 9, 12
the value is 0	Line 12, 13
the value is 1	Line 12, 9, 13

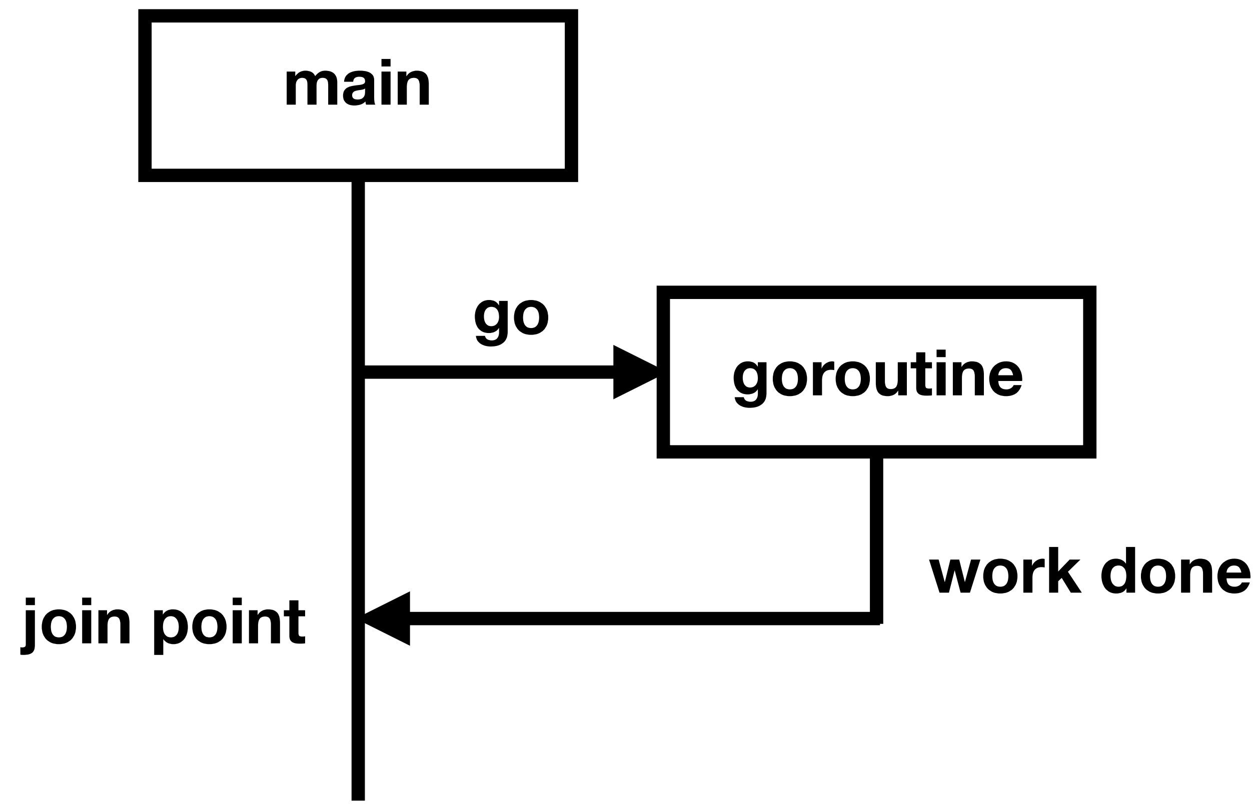
What could be possible outputs of this program?

```go
5    func main() {
6        var data int
7
8        go func() {
9            data++
10       }()
11
12       if data == 0 {
13           fmt.Printf("the value is %v\n", data)
14       }
15   }
```

Output	Execution sequence
Nothing is printed	Line 9, 12
the value is 0	Line 12, 13
the value is 1	Line 12, 9, 13

Can you make main() wait for goroutine to execute before checking value of data?

WaitGroup




```
var wg sync.WaitGroup
wg.Add(1)

    go func() {
        defer wg.Done()
        ....
    }()

    wg.Wait()
```

- Deterministically block main goroutine.

Summary

- How do we ensure that all goroutines have ended?

wg.Add(n) - indicates the number of goroutines started.

wg.Done() - indicates a goroutine is exiting.

wg.Wait() - block, till all goroutines exit.

Goroutines & Closures

Goroutines & Closures

- Goroutines execute within the **same address space** they are created in.
- They can directly modify variables in the enclosing lexical block.

```
func inc() {
    var i int
    go func() {
        i++
        fmt.Println(i)
    }()
    return
}
```

Deep Dive - Go Scheduler

M:N Scheduler

- The Go scheduler is part of the Go runtime. It is known as M:N scheduler

- Go scheduler runs in user space.

- Go scheduler uses OS threads to schedule goroutines for execution.

- Goroutines runs in the context of os threads.

M:N Scheduler

- Go runtime create number of worker OS threads, equal to GOMAXPROCS.

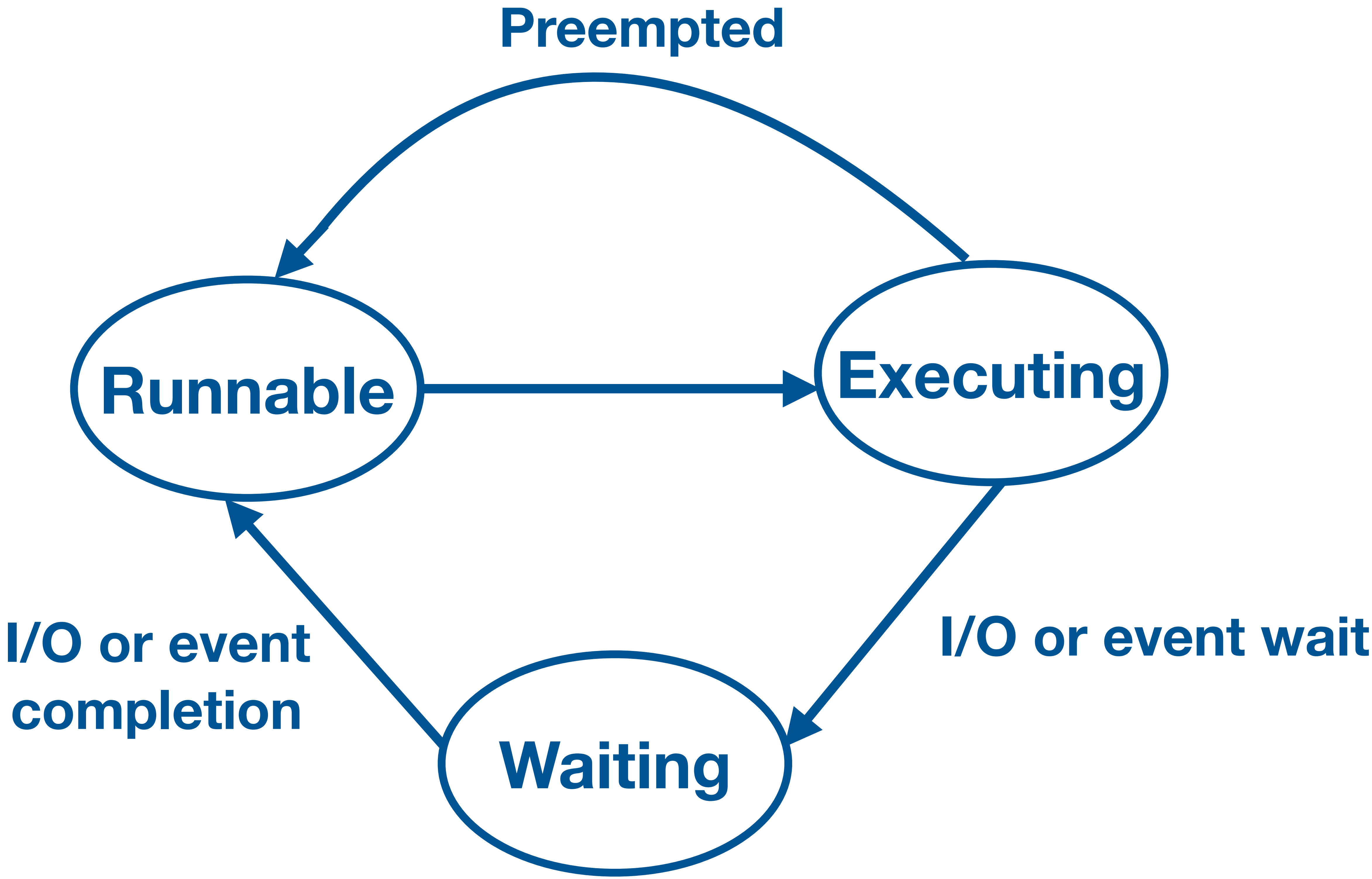
- GOMAXPROCS - default value is number of processors on machine.

- Go scheduler distributes runnable goroutines over multiple worker OS threads.

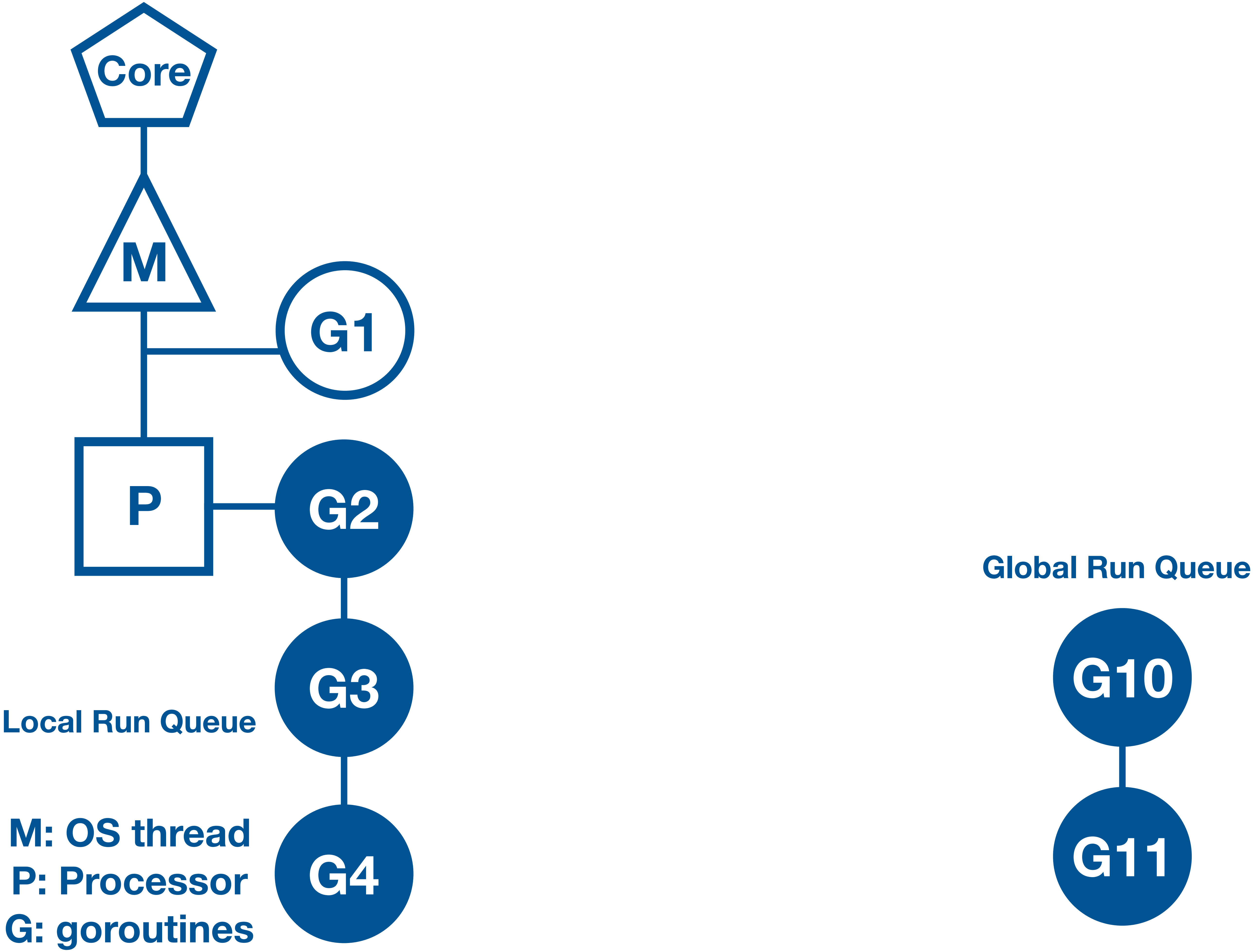
- At any time, N goroutines could be scheduled on M OS threads that runs on at most GOMAXPROCS numbers of processors.

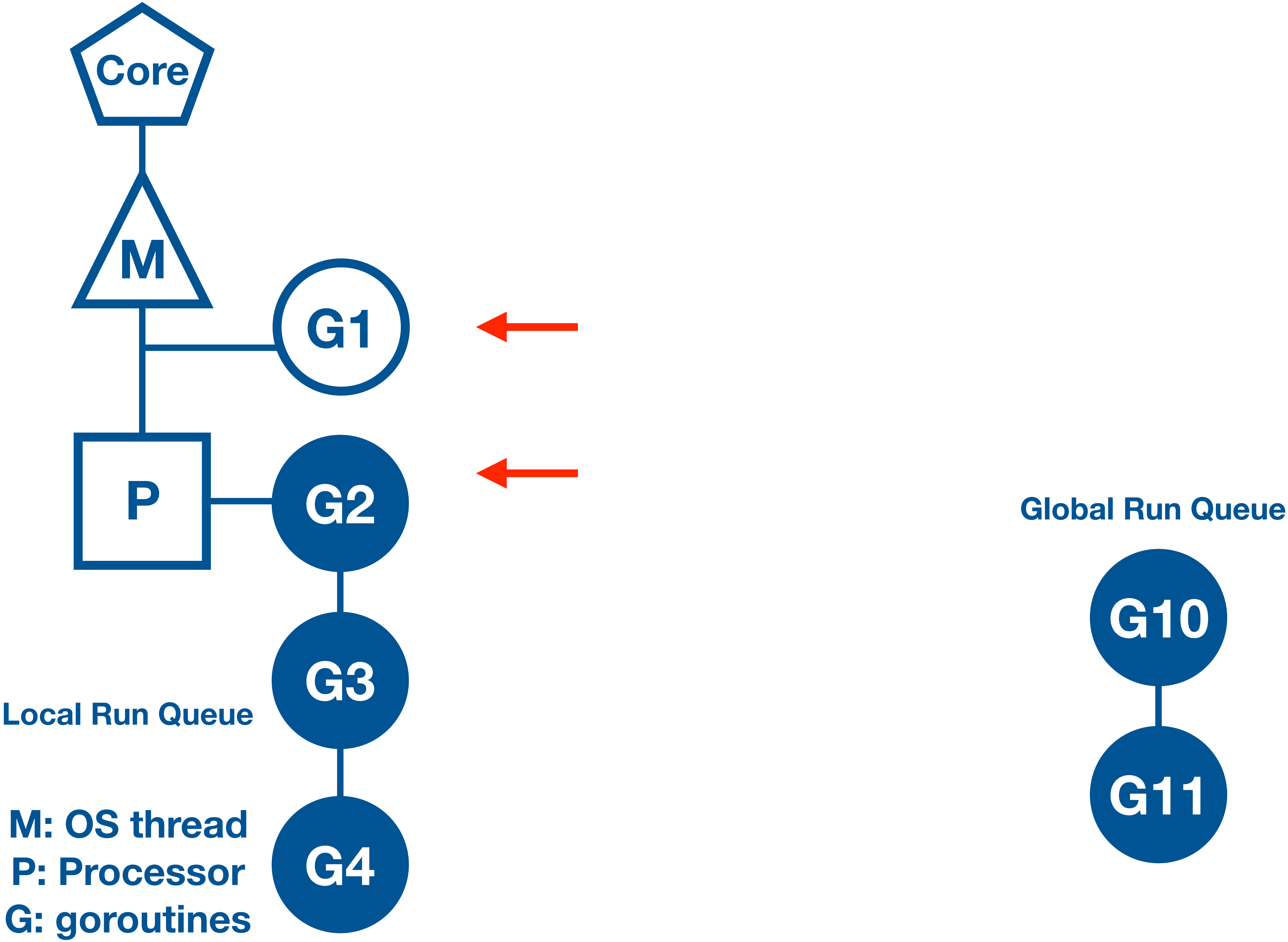
Asynchronous Preemption

- As of Go 1.14, Go scheduler implements asynchronous preemption.
- This prevents long running Goroutines from hogging onto CPU, that could block other Goroutines.
- The asynchronous preemption is triggered based on a time condition. When a goroutine is running for more than 10ms, Go will try to preempt it.

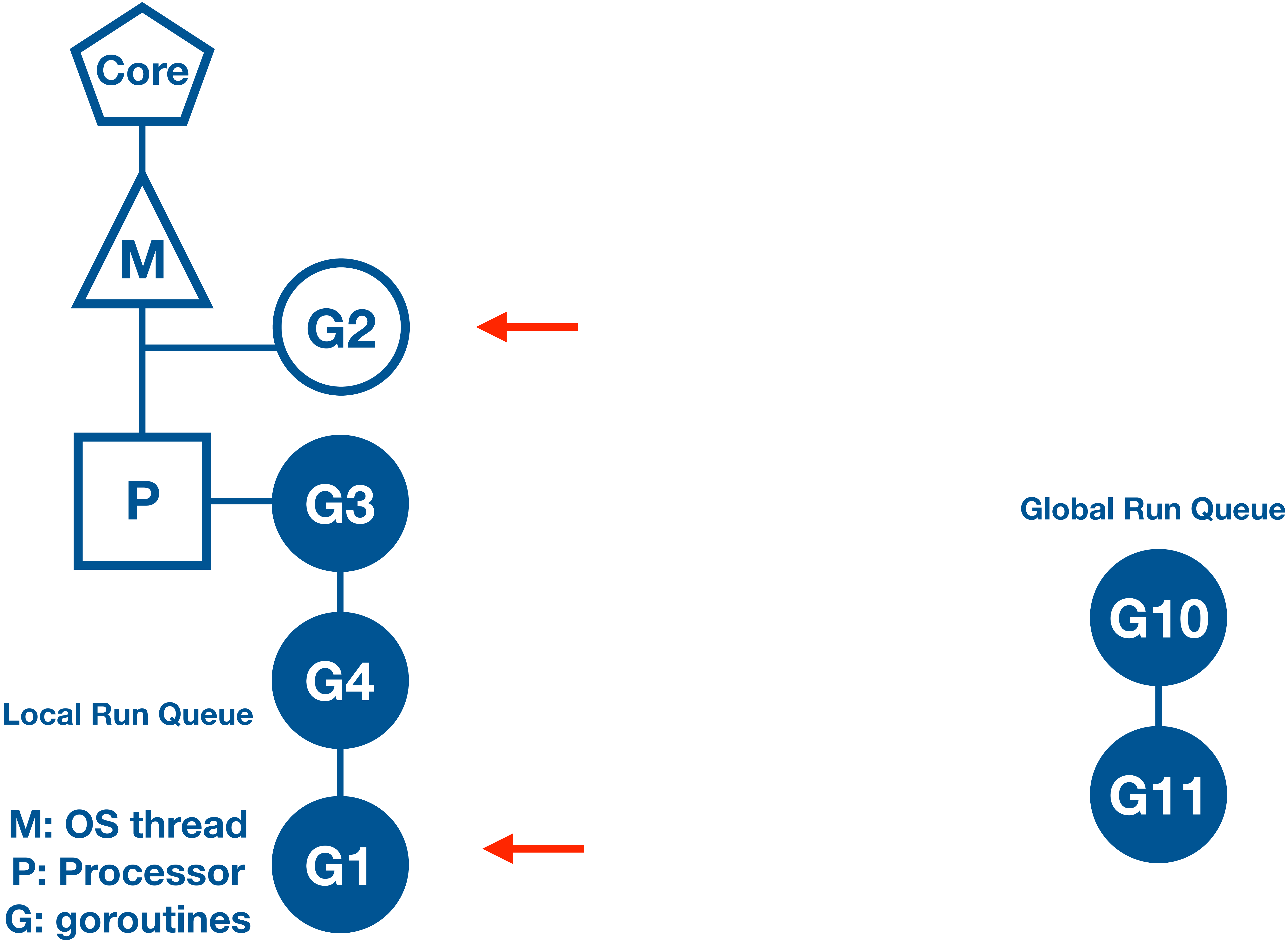


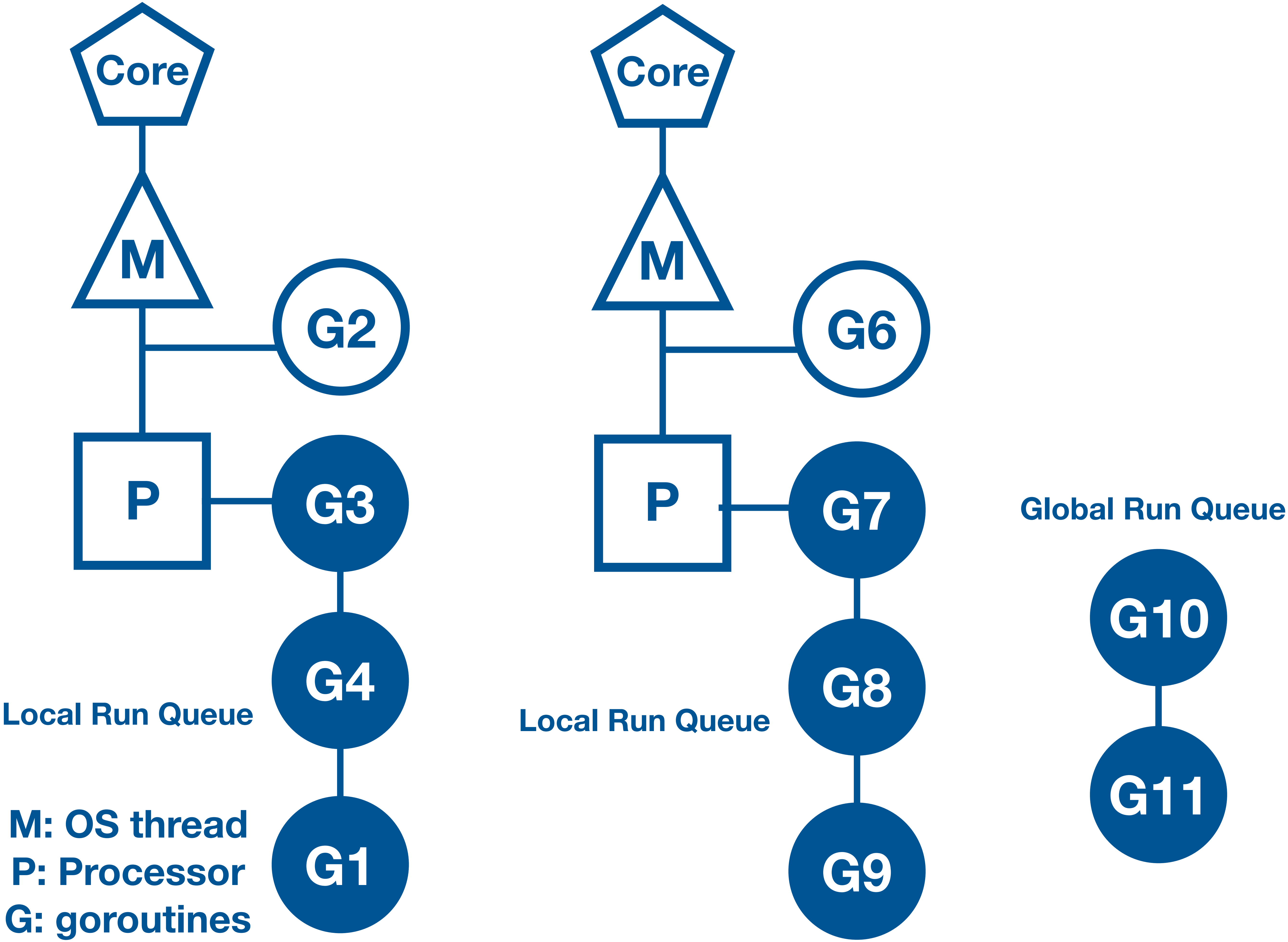
Goroutine States


Core
M
G1
P
G2
G3
Local Run Queue
G4
M: OS thread
P: Processor
G: goroutines
Global Run Queue
G10
G11



Core
M
G1
P
G2
G3
G4
Local Run Queue
M: OS thread
P: Processor
G: goroutines
Global Run Queue
G10
G11


Core
M
G2
P
G3
G4
Local Run Queue
G1
M: OS thread
P: Processor
G: goroutines
Global Run Queue
G10
G11


Core
M
G2
P
G3
G4
Local Run Queue
G1
M: OS thread
P: Processor
G: goroutines
Core
M
G6
P
G7
G8
Local Run Queue
G9
Global Run Queue
G10
G11

Summary

How does Go scheduler work?

- Go run time has mechanism known as MN Scheduler.
- N goroutines could be scheduled on M OS threads that runs on at most GOMAXPROCS numbers of processors.
- As of Go 1.14 Go scheduler implements asynchronous preemption, each Goroutine is given a time slice of 10ms.

Summary

What are components of Go scheduler?

- M - represents OS thread.
- P - is the logical processor, which manages scheduling of goroutines.
- G - is the goroutine, which also includes scheduling information like stack and instruction pointer.
- Local run queue - where runnable goroutines are arranged.
- Global run queue - when a goroutine is created, they are placed into global run queue.

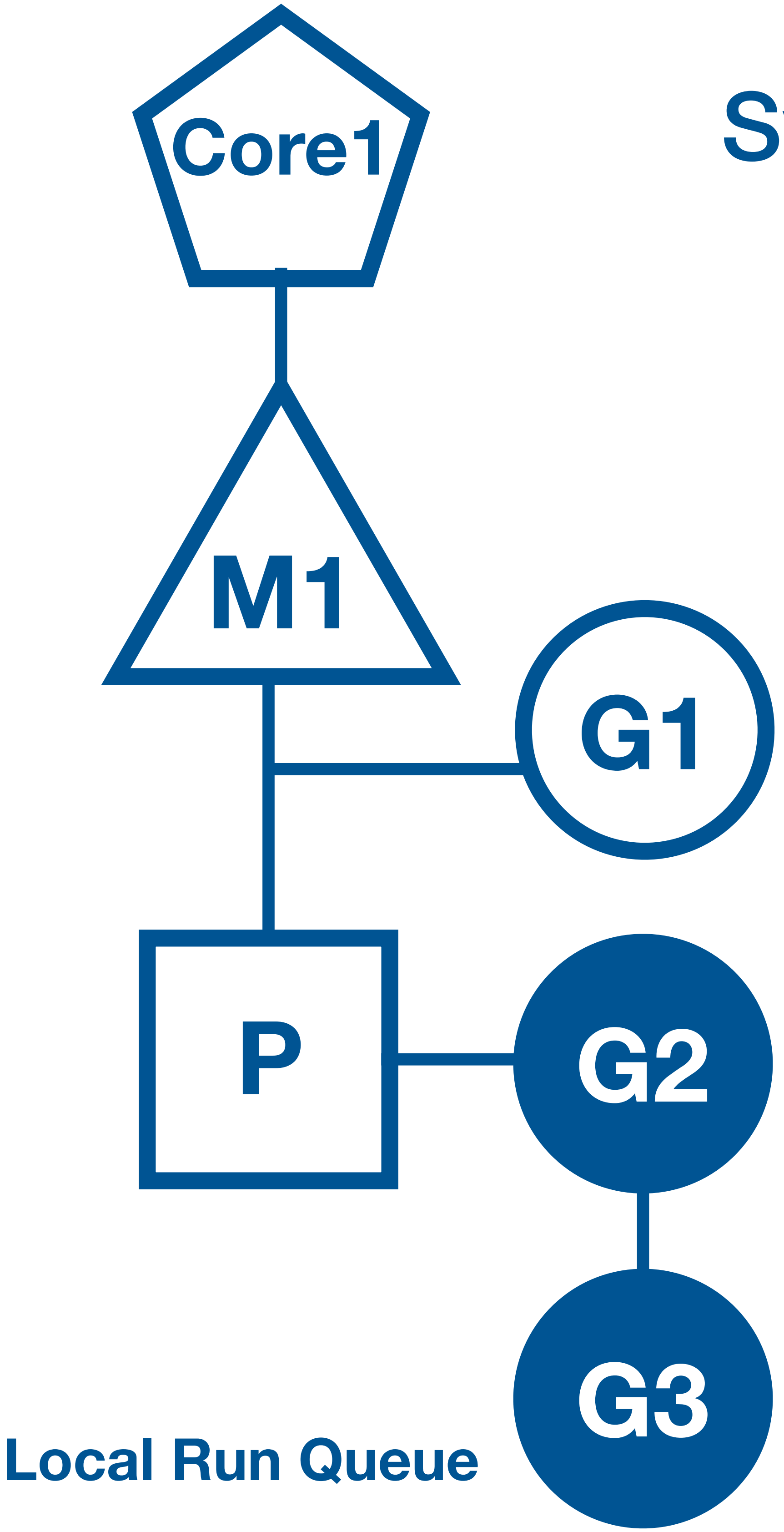
Context Switch due to Synchronous System Call

Scenario

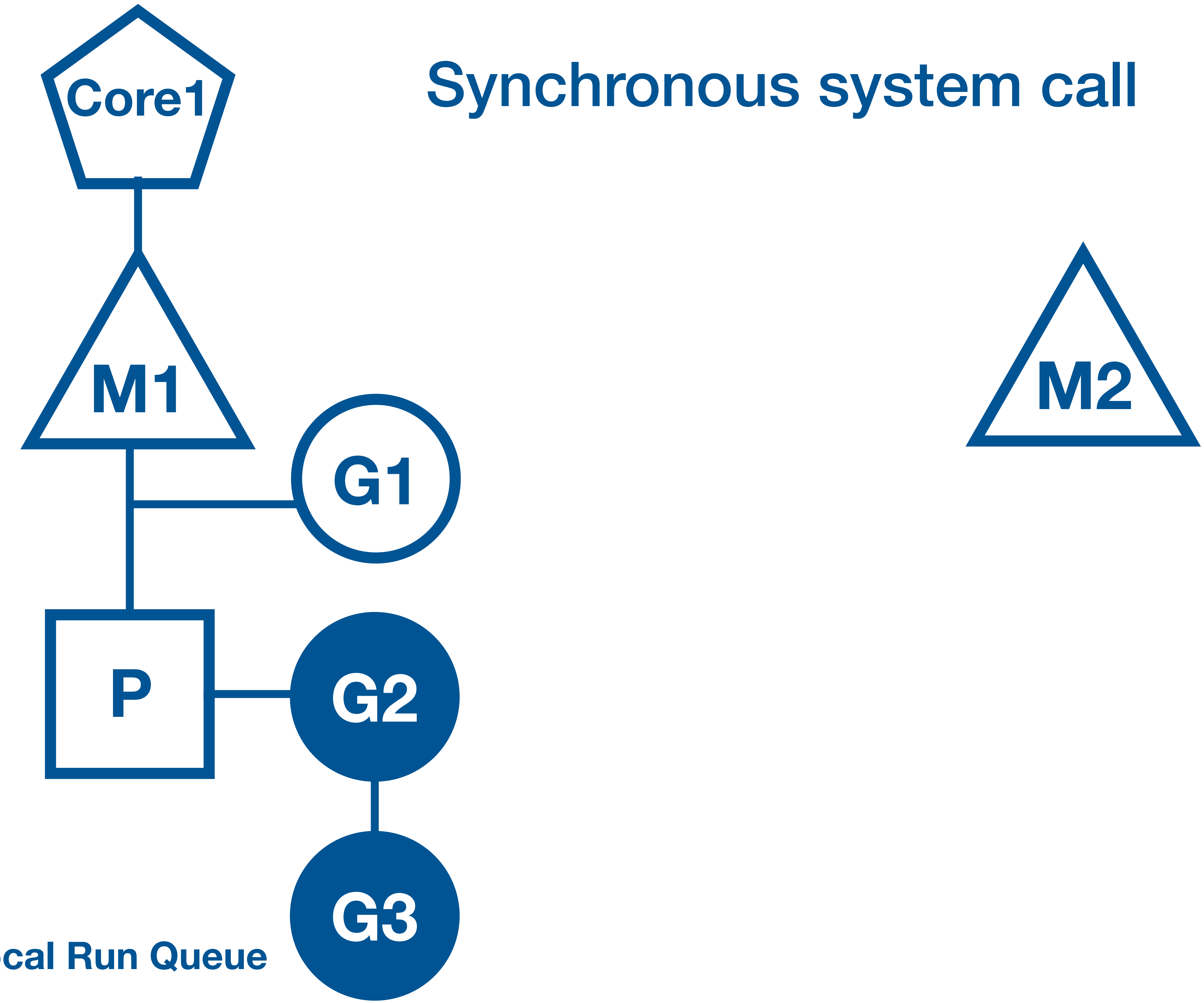
What happens in general when synchronous system call are made?

- synchronous system calls wait for I/O operation to be completed.
- OS thread is moved out of the CPU to waiting queue for I/O to complete.
- Synchronous system call reduces parallelism.

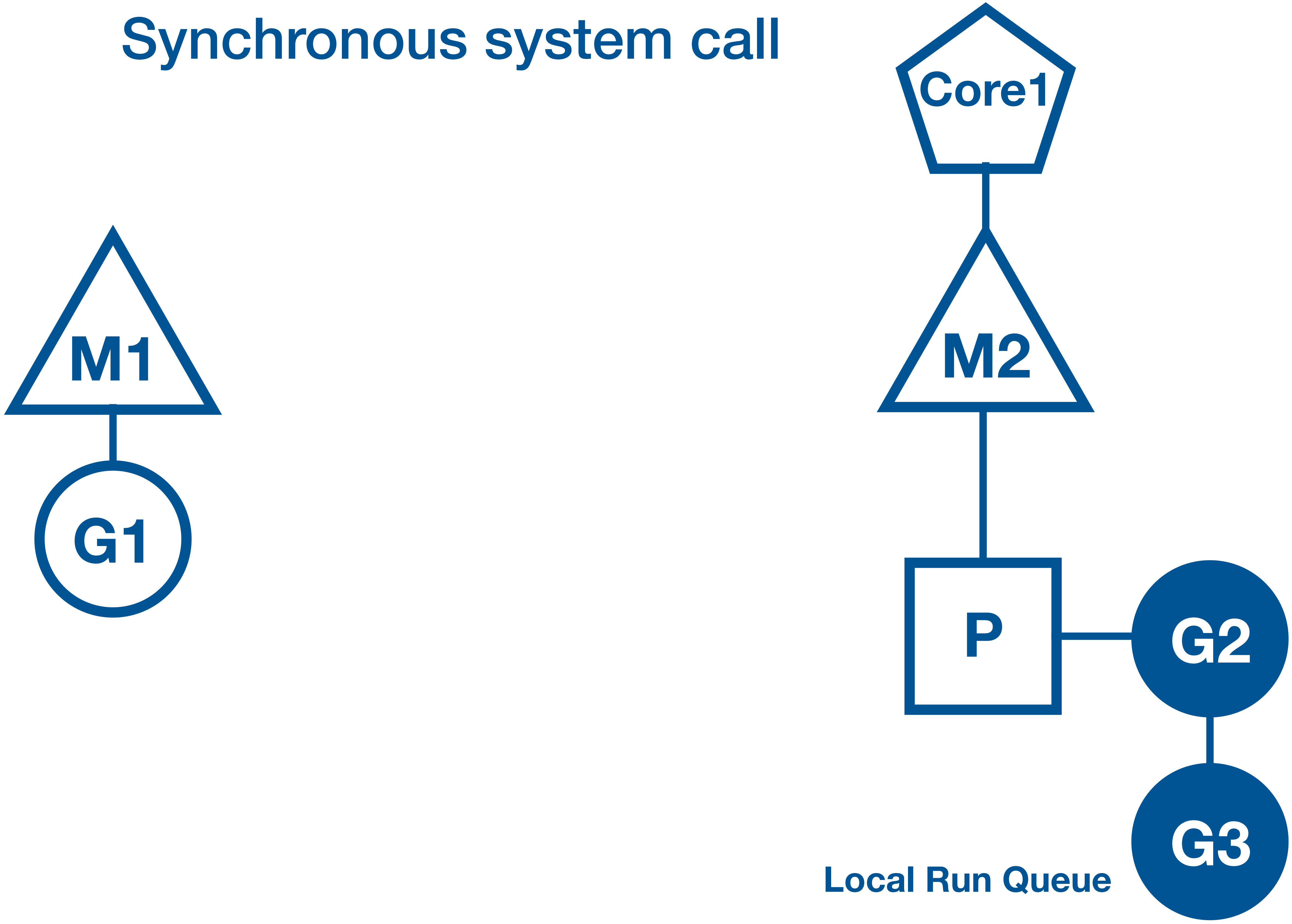
Synchronous system call



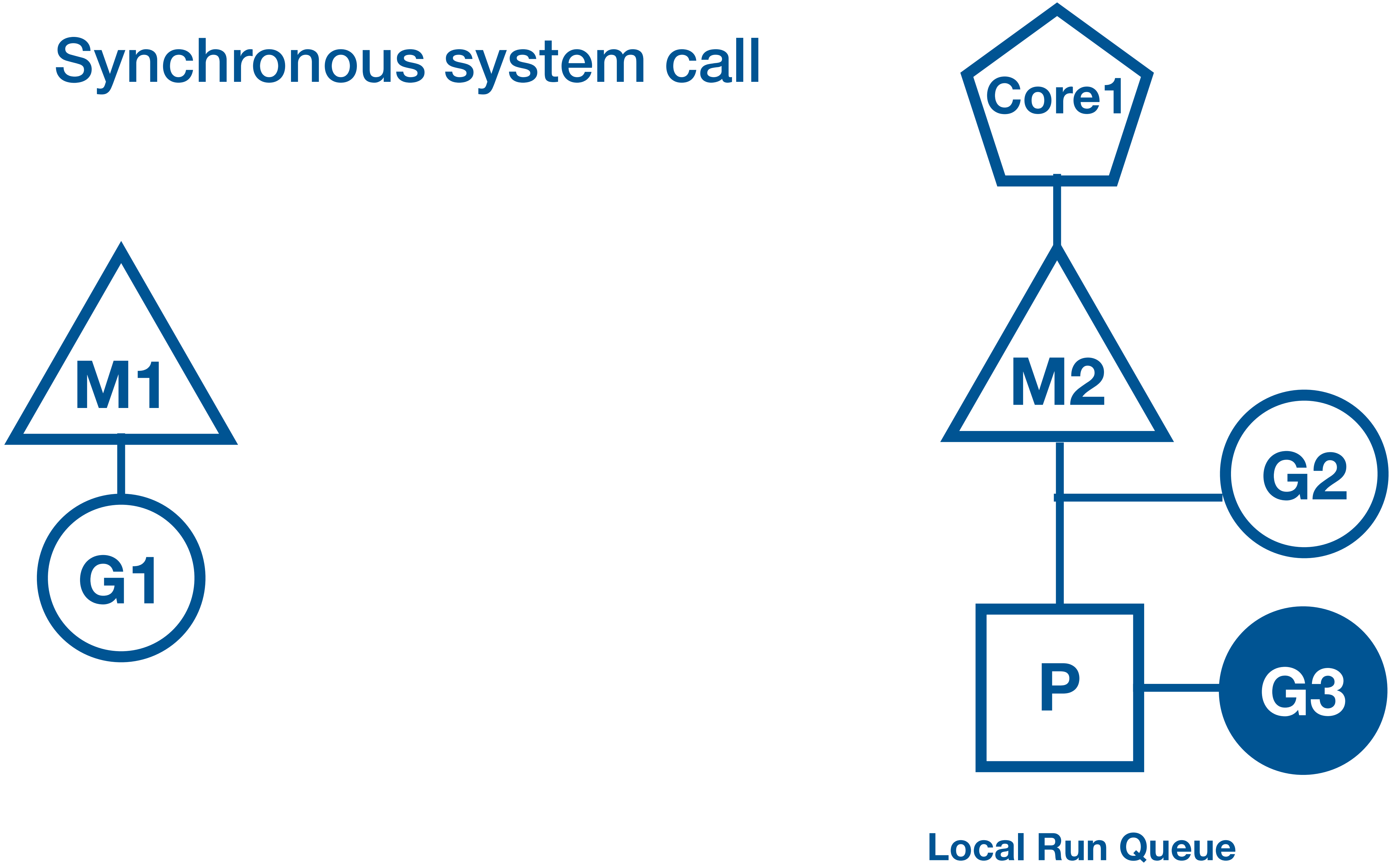

```
count, err := f.Read(data)
```


Synchronous system call
Core1
M1
G1
P
G2
G3
Local Run Queue
M2

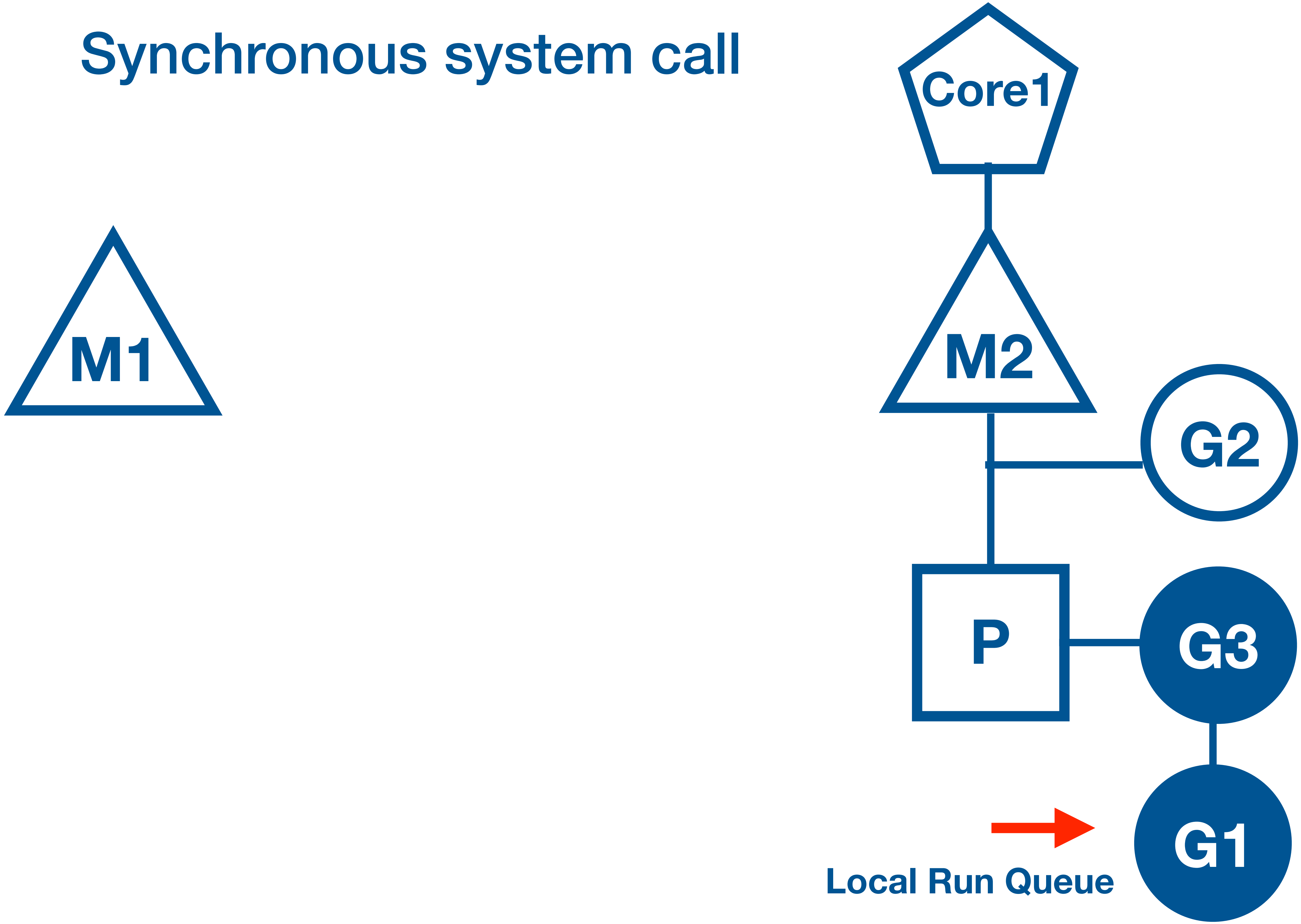
Synchronous system call



Synchronous system call



Synchronous system call



Summary

How does context switching works when a goroutine calls synchronous system call?

- When Goroutine makes synchronous system call, Go scheduler bring a new OS thread from thread pool.
- Moves the logical processor P to new thread.
- Goroutine which made the system call will still be attached to old thread.

Summary

- Other Goroutines in LRQ are scheduled for execution on new OS thread.

- Once system call returns, Goroutine is moved back to run queue on logical processor P and old thread is put to sleep.

Context Switching due to Asynchronous System Calls

Scenario

What happens in general when asynchronous system call are made?

- File descriptor is set to non-blocking mode
- If file descriptor is not ready, for I/O operation, system call does not block, but returns an error.
- Asynchronous IO increases the application complexity.
- Setup event loops using callbacks functions.

netpoller

- Netpoller to convert asynchronous system call to blocking system call.

- When a goroutine makes a asynchronous system call, and file descriptor is not ready, goroutine is parked at netpoller os thread.

- netpoller uses interface provided by OS to do polling on file descriptors

    - kqueue (MacOS)

    - epoll (Linux)

    - iocp(Windows)

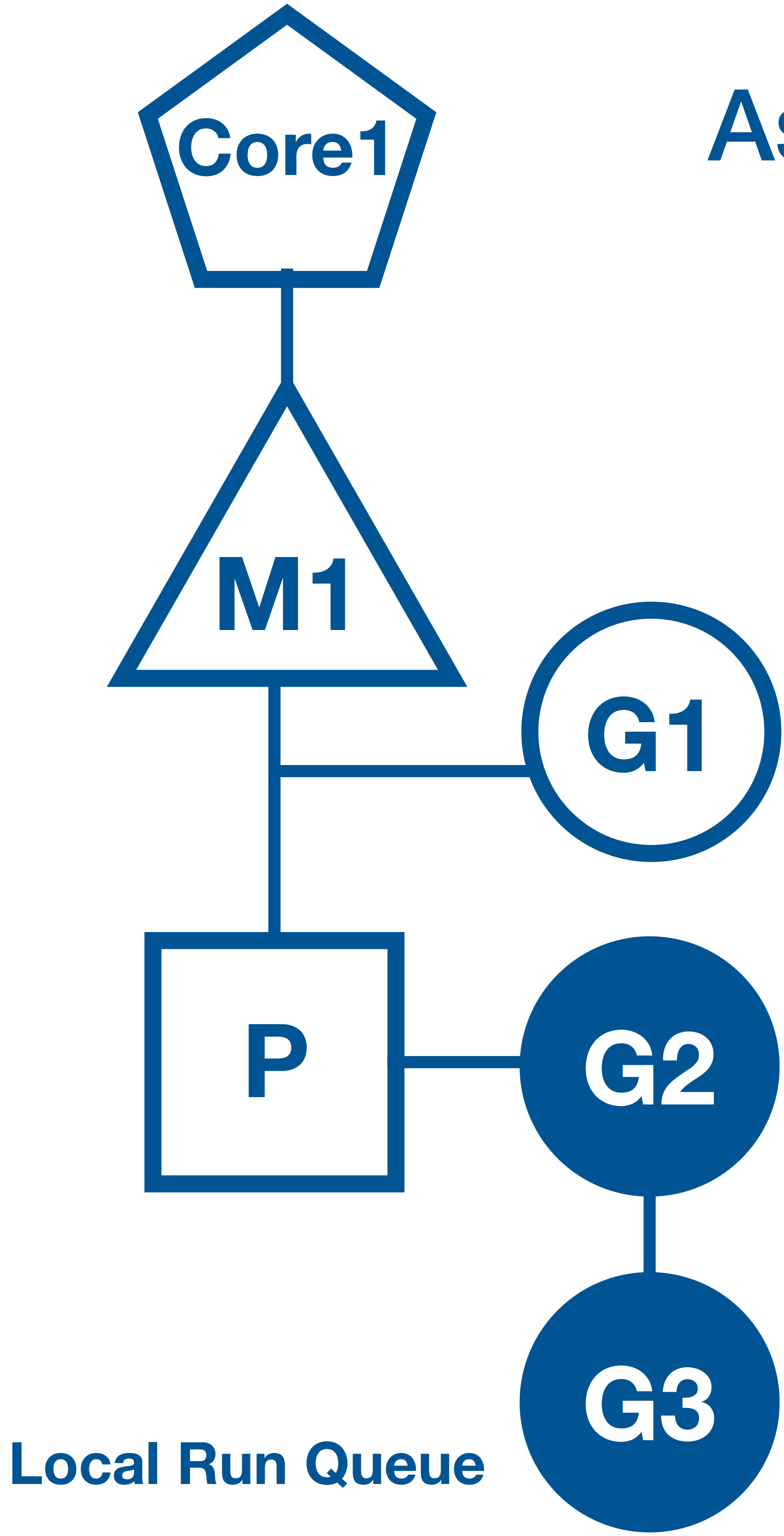
netpoller

- Netpoller gets notification from OS, when file descriptor is ready for I/O operation.

- Netpoller notifies goroutine to retry I/O operation.

- Complexity of managing asynchronous system call is moved from Application to Go runtime, which manages it efficiently.

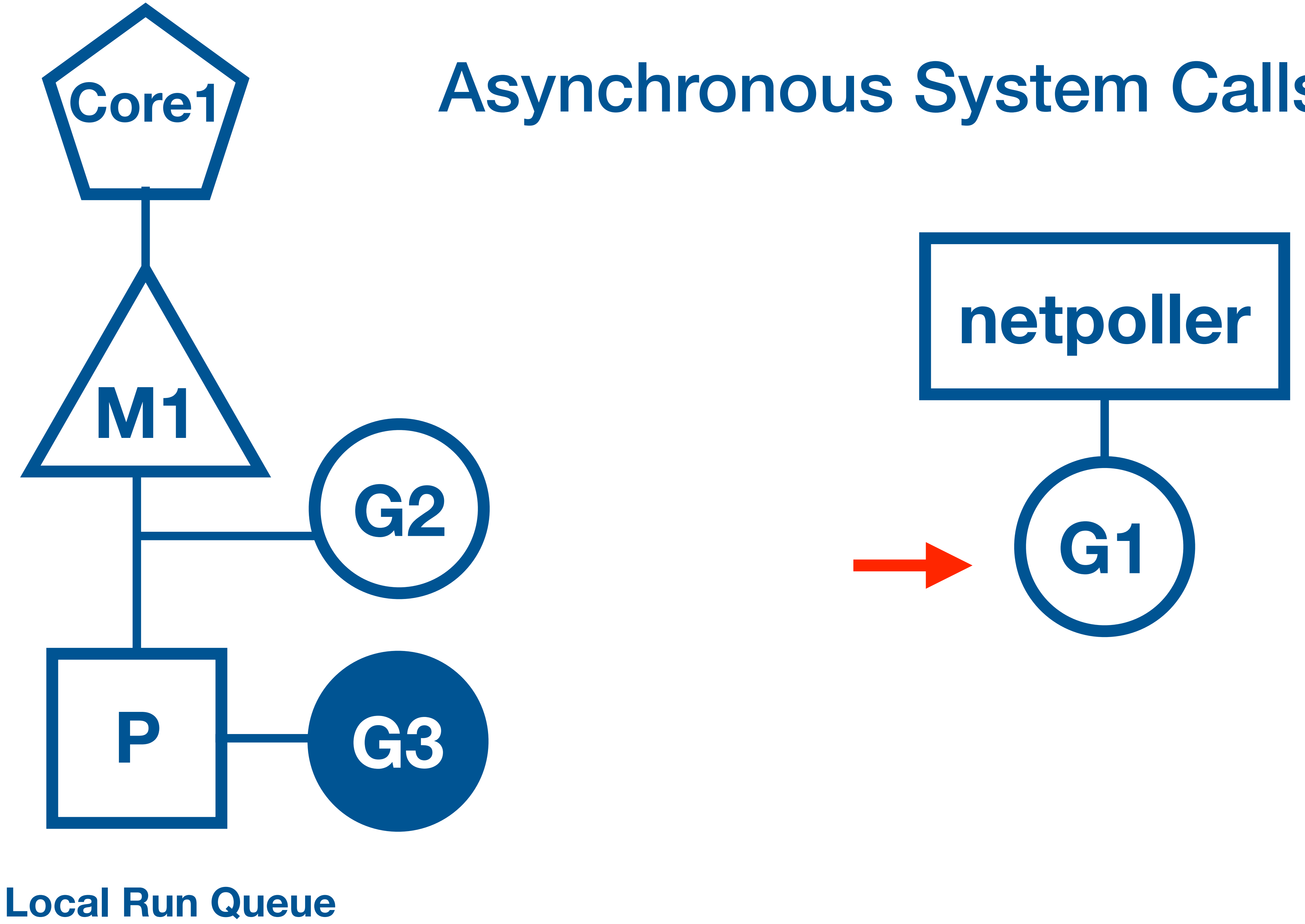
Asynchronous System Calls



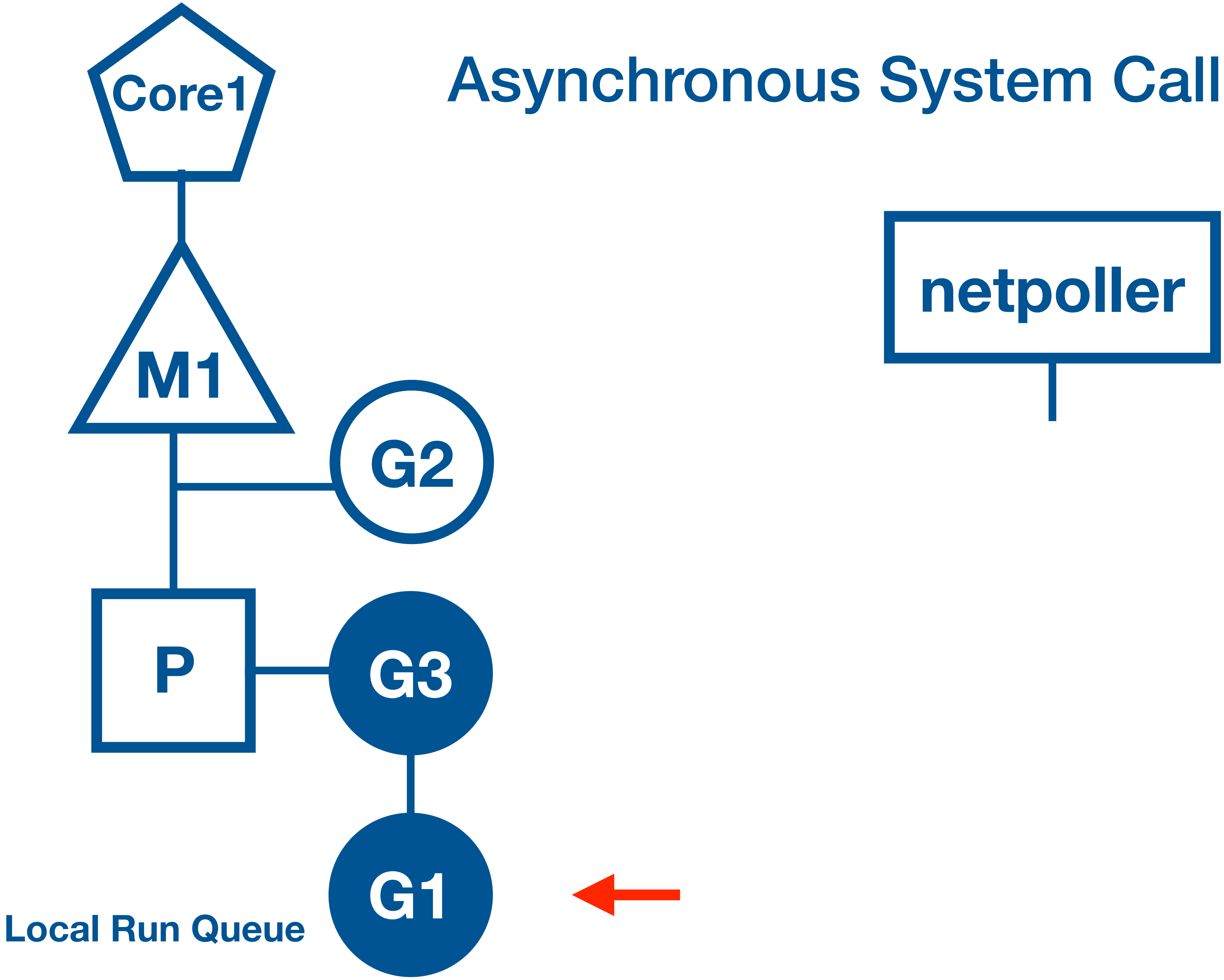

```go
conn, err := net.Dial("tcp", "localhost:8000")

msg, _ := bufio.NewReader(conn).ReadString('\n')
```

```go
n, err := syscall.Read(fd.Sysfd, p)
if err != nil {
    n = 0
    if err == syscall.EAGAIN && fd.pd.pollable() {
        if err = fd.pd.waitRead(fd.isFile); err == nil {
            continue
        }
    }
```

Asynchronous System Calls



Asynchronous System Calls



Summary

What happens when goroutine makes asynchronous system call?

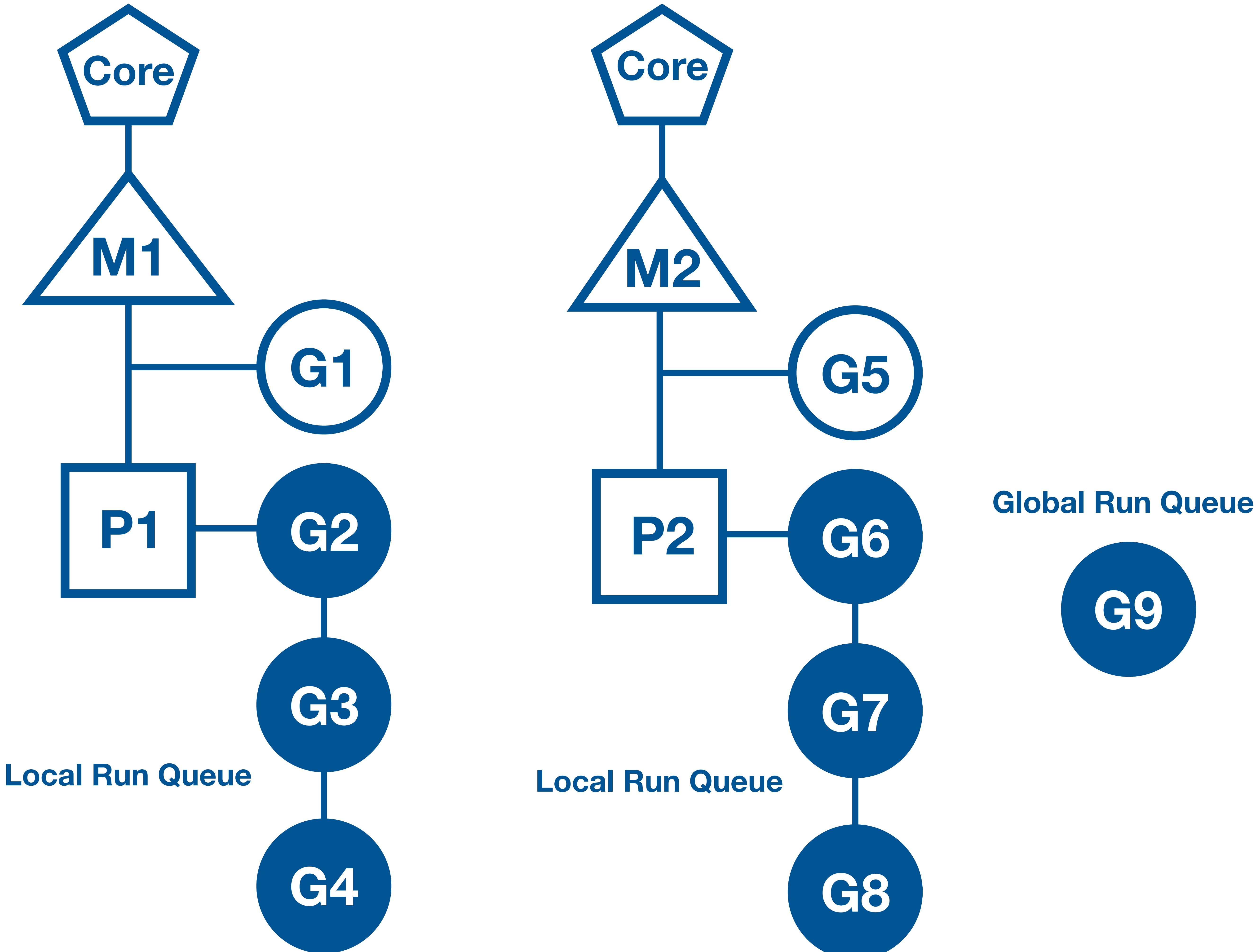
- Go uses netpoller to handle asynchronous system call.
- netpoller uses interface provided by OS to do polling on file descriptors and notifies the goroutine to try I/O operation when it ready.
- Application complexity of managing asynchronous system call is moved to Go runtime, which manages it efficiently.

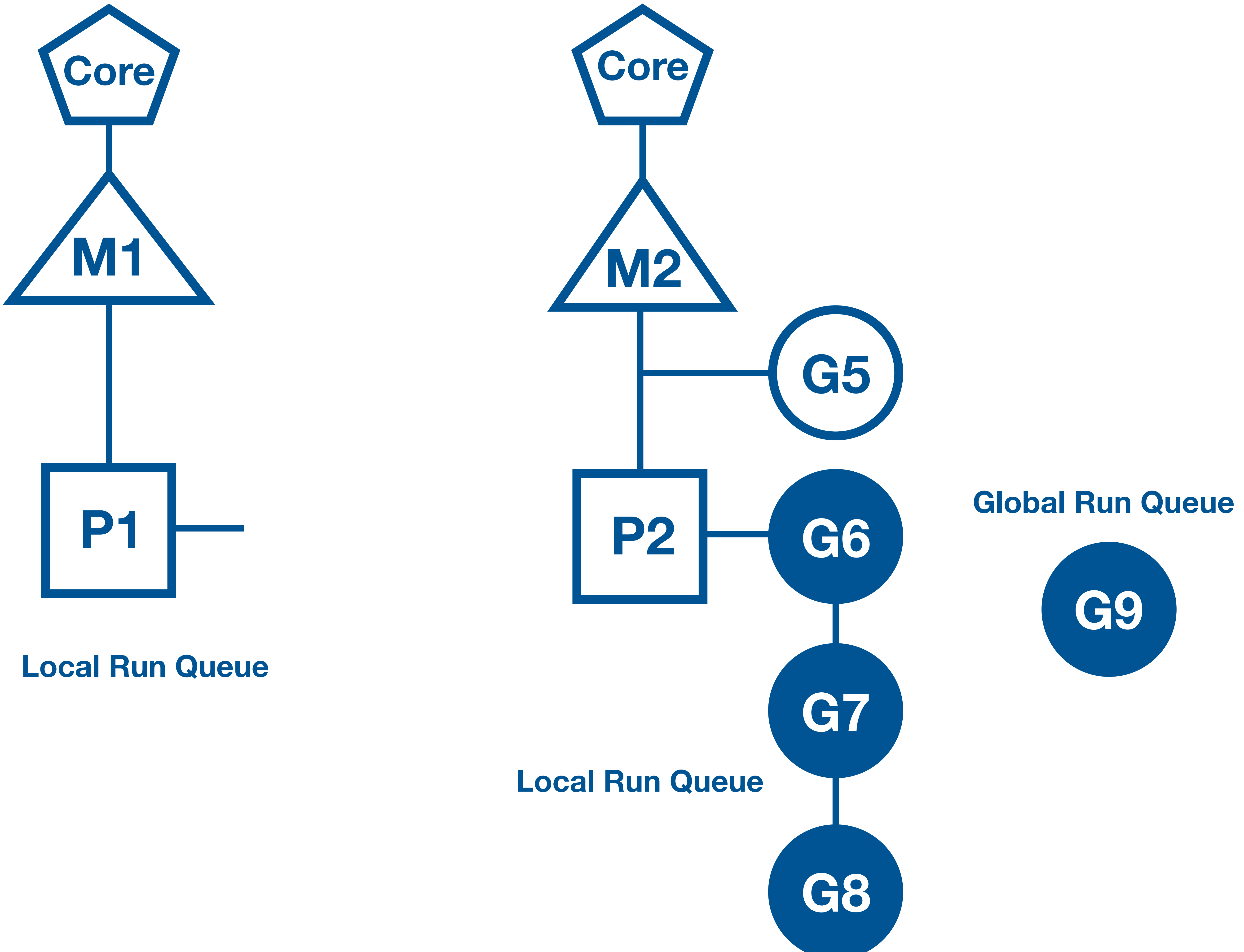
Work Stealing

Work Stealing

- Work stealing helps to balance the goroutines across all logical processors.

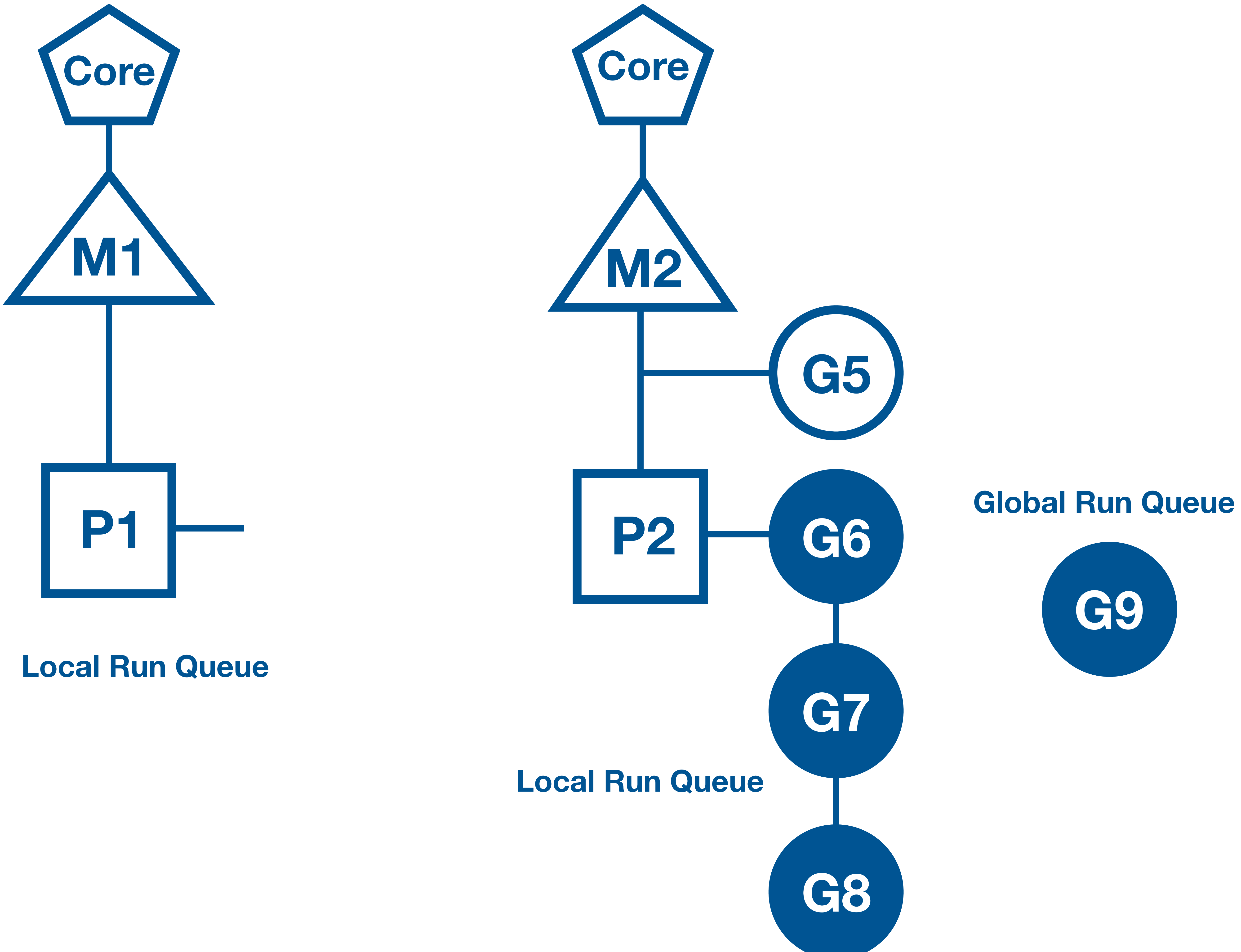
- Work gets better distributed and gets done more efficiently.

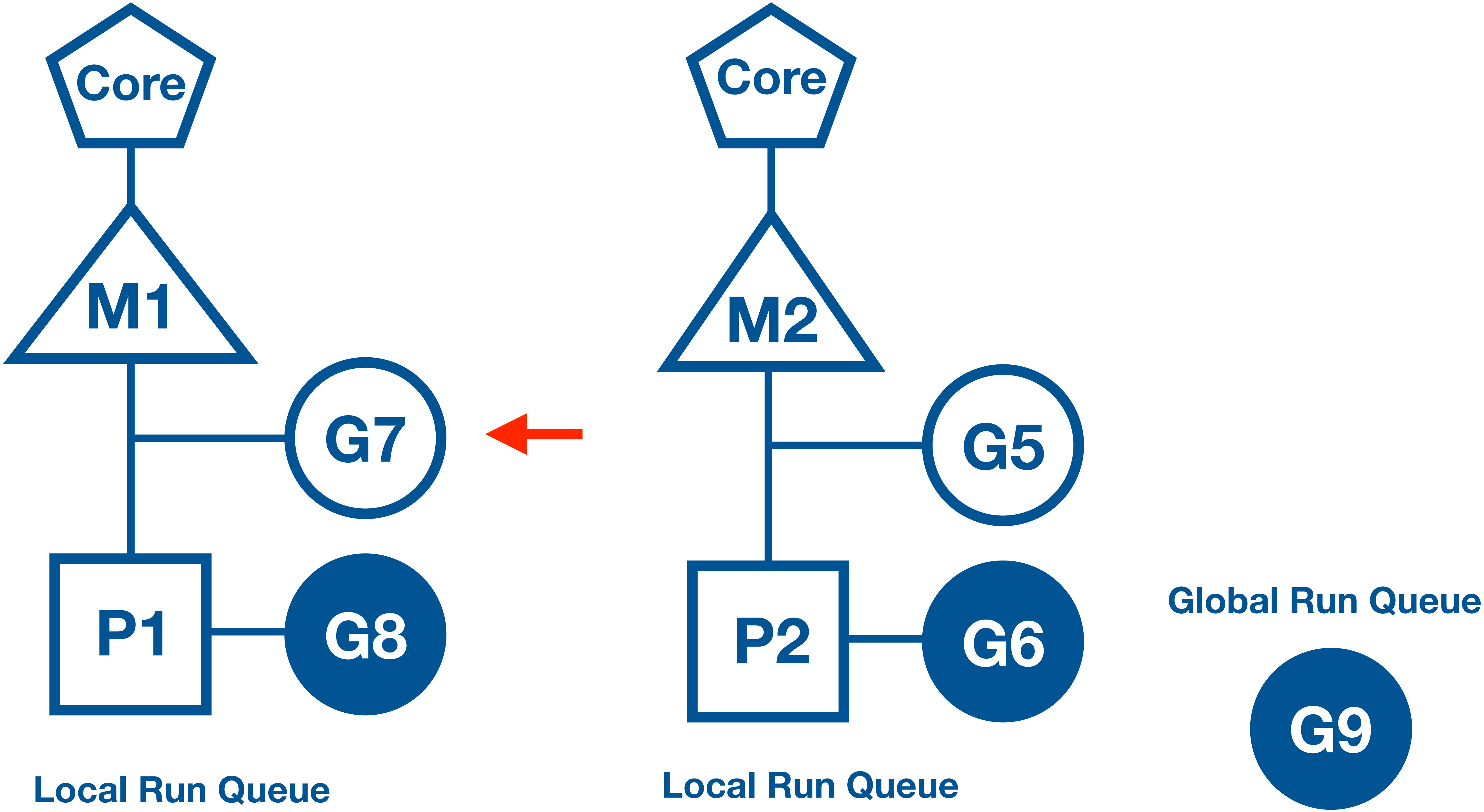

Core
M1
G1
P1
G2
G3
G4
Local Run Queue
Core
M2
G5
P2
G6
G7
G8
Local Run Queue
Global Run Queue
G9

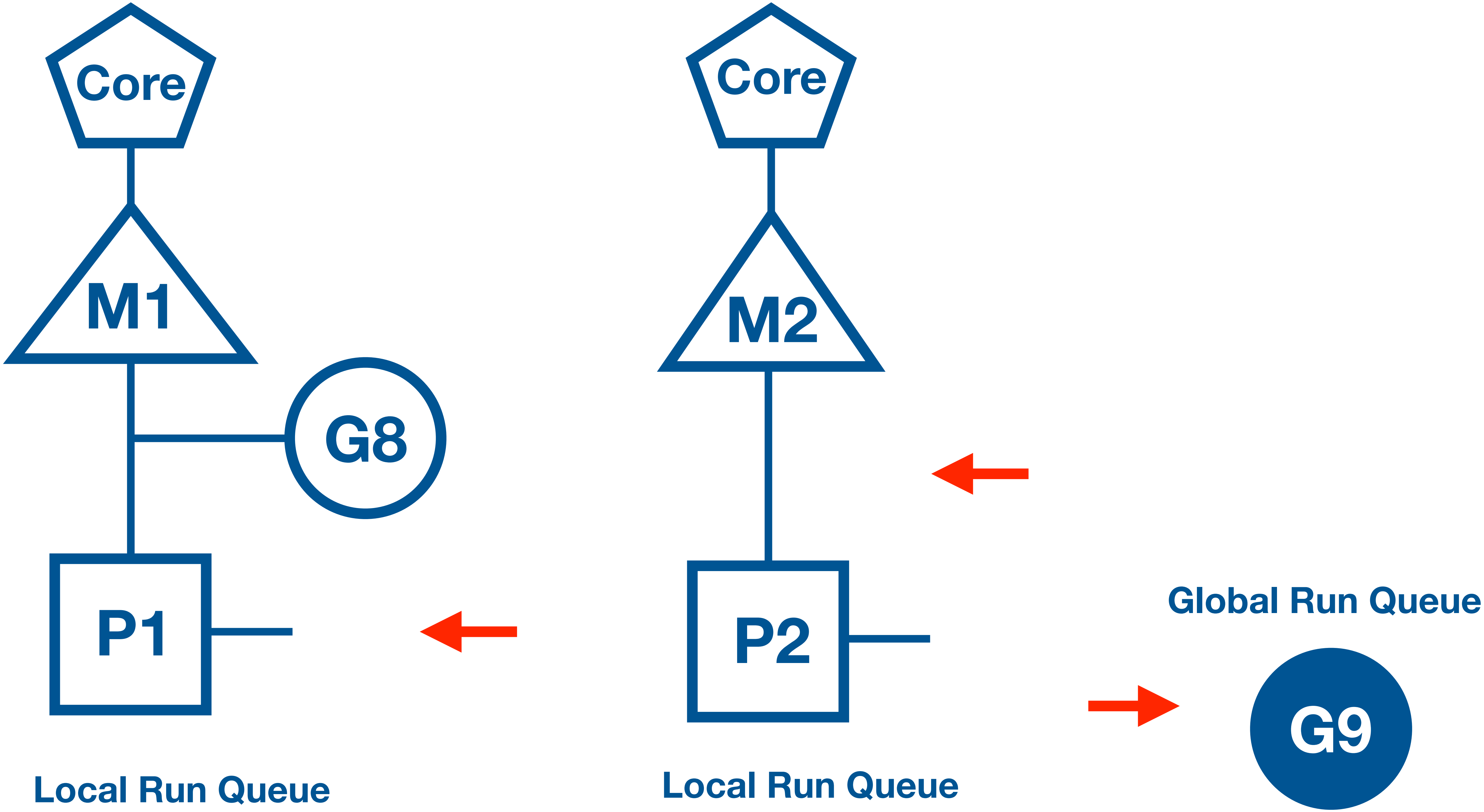

Core
M1
P1
Local Run Queue
Core
M2
G5
P2
G6
G7
G8
Local Run Queue
Global Run Queue
G9

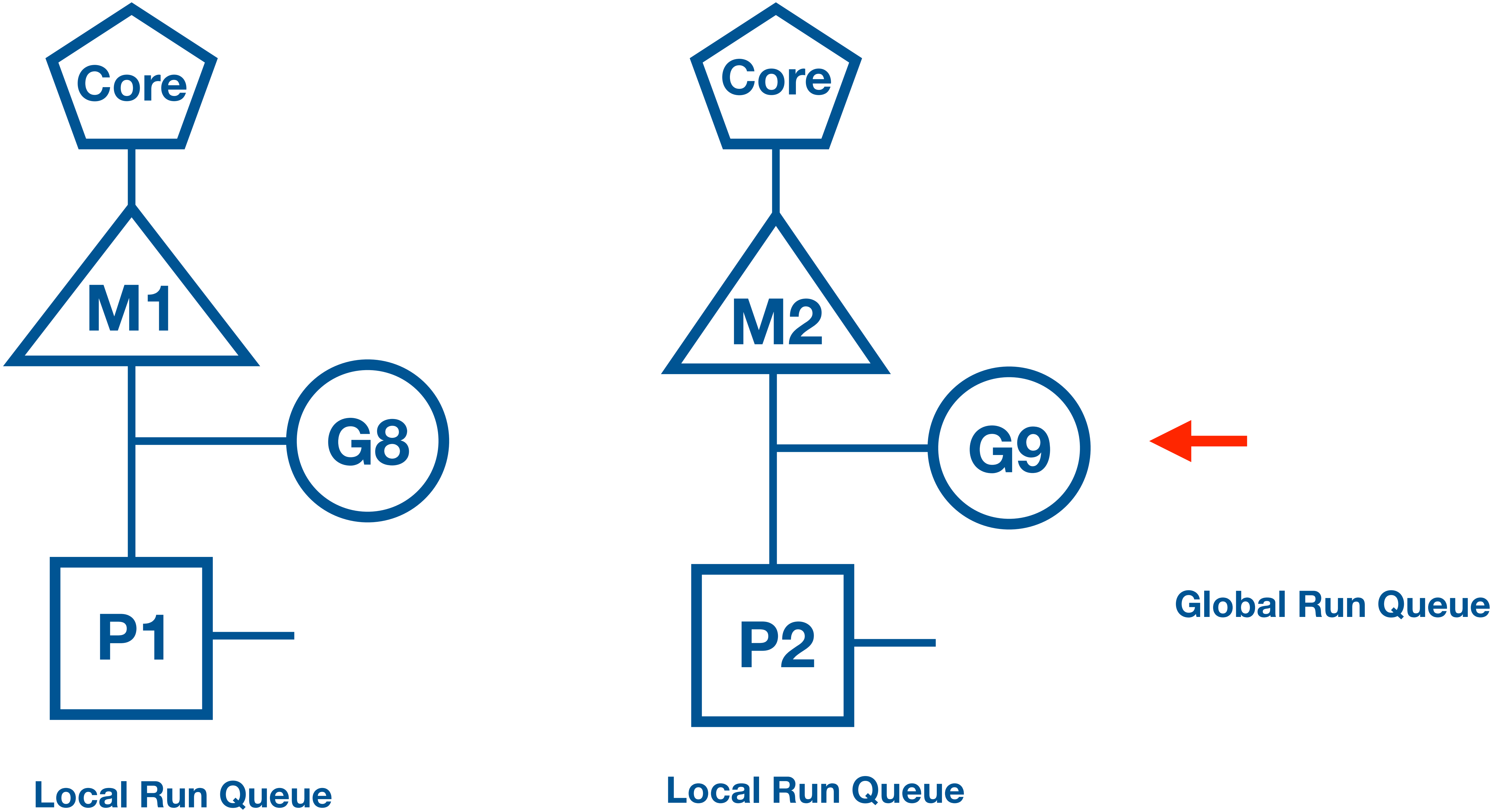
Work Stealing Rule

- If there is no goroutines in local run queue.
    - Try to steal from other logical processors.
    - If not found, check the global runnable queue for a G
    - If not found, check netpoller.


Core
M1
P1
Local Run Queue
Core
M2
G5
P2
G6
G7
G8
Local Run Queue
Global Run Queue
G9


Core
M1
G7
P1
G8
Local Run Queue
Core
M2
G5
P2
G6
Local Run Queue
Global Run Queue
G9


Core
Core
M1
M2
G8
P1
P2
Global Run Queue
G9
Local Run Queue
Local Run Queue


Core
M1
G8
P1
Local Run Queue
Core
M2
G9
P2
Local Run Queue
Global Run Queue

Summary

How work stealing scheduler works?

- If logical processor run out of goroutines in its local run queue, it will steal goroutines from other logical processor's or global run queue.

- Work stealing helps in better distribution of goroutines across all logical processors.

Channels

How to get value computed by goroutine into main routine?

```go
func main() {
  go func(a, b int) {
    c := a + b
  }(1, 2)
  // TODO: get the value computed from goroutine
  // fmt.Printf("computed value %v\n", c)
}
```

Channels

- Communicate data between Goroutines
- Synchronise Goroutines
- Typed
- Thread-safe

Photo by Hassan OUAJBIR on Unsplash   Photo by Jim Reardan on Unsplash

Declare and Initialize

```
var ch chan T

ch = make(chan T)
```

OR

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

<- operator

- Pointer operator is used for sending and receiving the value from channel.

- The "arrow" indicates the direction of data flow.

Send

ch <- v

Receive

v = <-ch

Channels are blocking

ch <- value

Goroutine wait for a receiver to be ready.

<- ch

Goroutine wait for a value to be sent.

- It is responsibility of channel to make the goroutine runnable again once it has data.

close(ch)

- No more values to be sent.

value, ok = <- ch

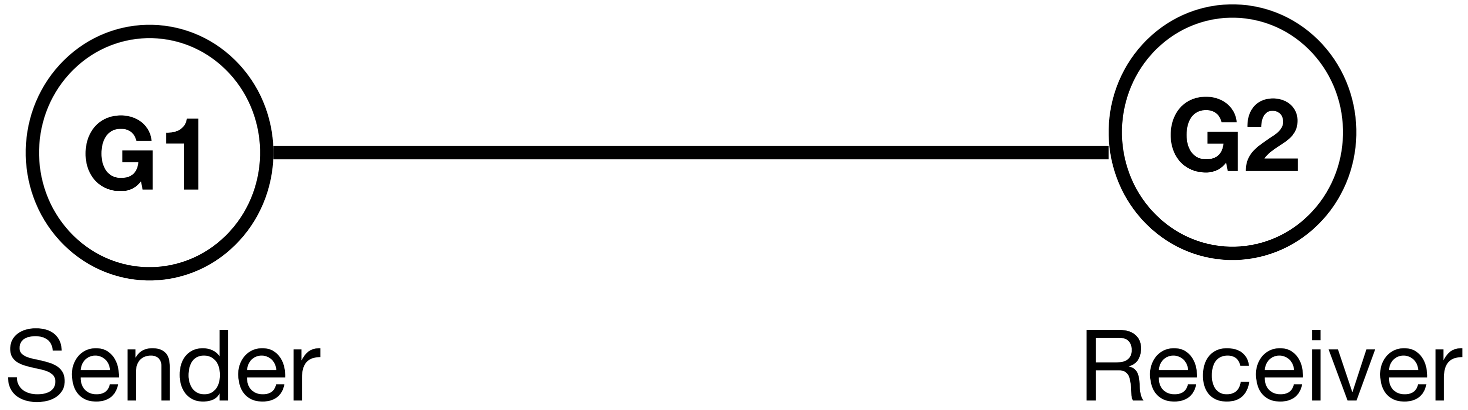
- ok = true, value generated by a write.
- ok = false, value generated by a close.

```go
for value := range ch {
...
}
```

- Iterate over values received from a channel
- Loop automatically breaks, when a channel is closed.
- range does not return the second boolean value.

Unbuffered Channels

- Synchronous

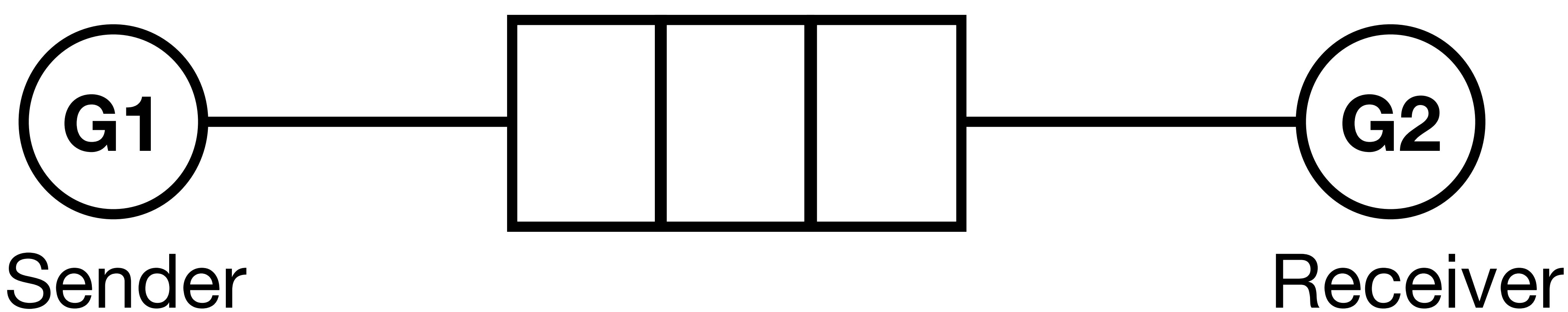


ch := make(chan *Type*)

Buffered Channels

- channels are given capacity
- in-memory FIFO queue
- Asynchronous

ch := make(chan *Type*, *capacity*)



Channel Direction

- When using channels as function parameters, you can specify if a channel is meant to only send or receive values.
- This specificity increases the type-safety of the program.

```
func pong(in <-chan string, out chan<- string) {}
```

Default value - Channels

- Default value for channels: **nil**

```
var ch chan interface{}
```

- reading/writing to a nil channel will block forever.

```
var ch chan interface{}

<-ch

ch <- struct{}{}
```

Default value - Channels

- closing nil channel will panic

```
var ch chan interface{}
close(ch)
```

- Ensure the channels are initialized first.

Ownership - Channels

- Owner of channel is a **goroutine that instantiates, writes, and closes a channel.**

- Channel utilizers only have a read-only view into the channel

Ownership of channels avoids

- Deadlocking by writing to a nil channel
- closing a nil channel
- writing to a closed channel
- closing a channel more than once

Panic

Summary

- Channels are used to communicate data between Goroutines.
- In unbuffered channels send and receive are synchronous.
- Buffered channels we can specify the capacity of buffer.
- Channel direction used in function paremeter increases type safety.
- Establishing the ownership of channel avoids - deadlocks and panics.

Deep Dive - Channels

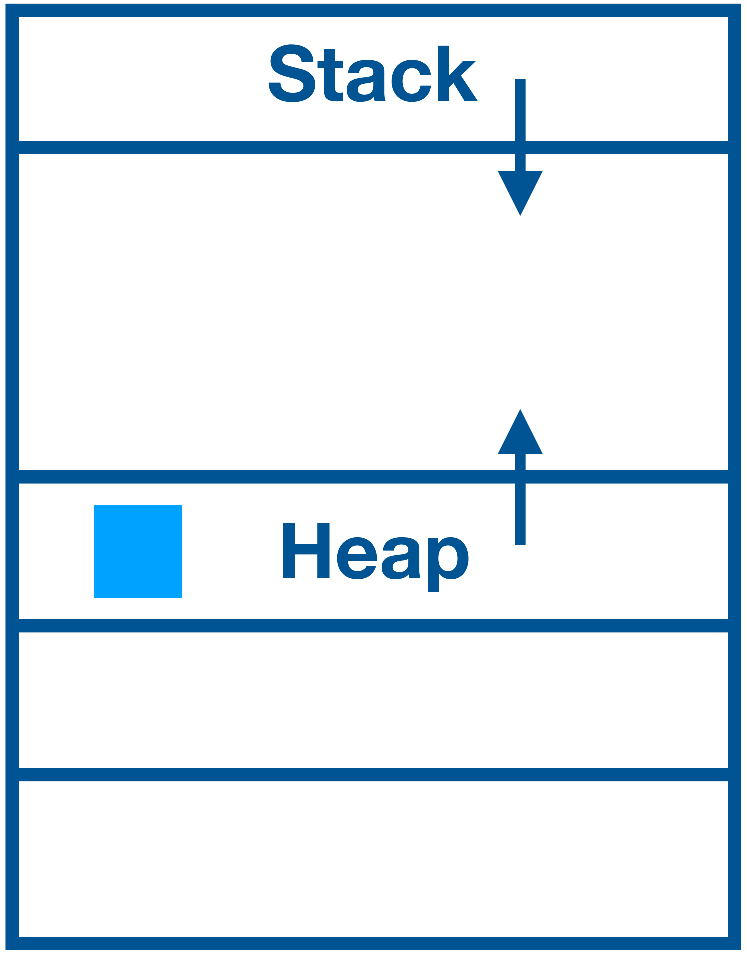
```go
ch := make(chan int, 3)
```

```go
type hchan struct {
  qcount   uint           // total data in the queue
  dataqsiz uint           // size of the circular queue
  buf      unsafe.Pointer // points to an array of dataqsiz elements
  elemsize uint16
  closed   uint32
  elemtype *_type // element type
  sendx    uint   // send index
  recvx    uint   // receive index
  recvq    waitq  // list of recv waiters
  sendq    waitq  // list of send waiters

  // lock protects all fields in hchan,
  lock mutex
}

type waitq struct {
  first *sudog
  last  *sudog
}
```

```go
// sudog represents a g in a wait list, such as for sending/receiving
// on a channel.

type sudog struct {

    g *g

    next *sudog
    prev *sudog
    elem unsafe.Pointer // data element (may point to stack)

     ...

     c            *hchan // channel

}
```

```go
ch := make(chan int, 3)
```

- hchan struct is allocated in heap.
- make() returns a pointer to it.
- Since 'ch' is pointer it can be between functions for send and receive.



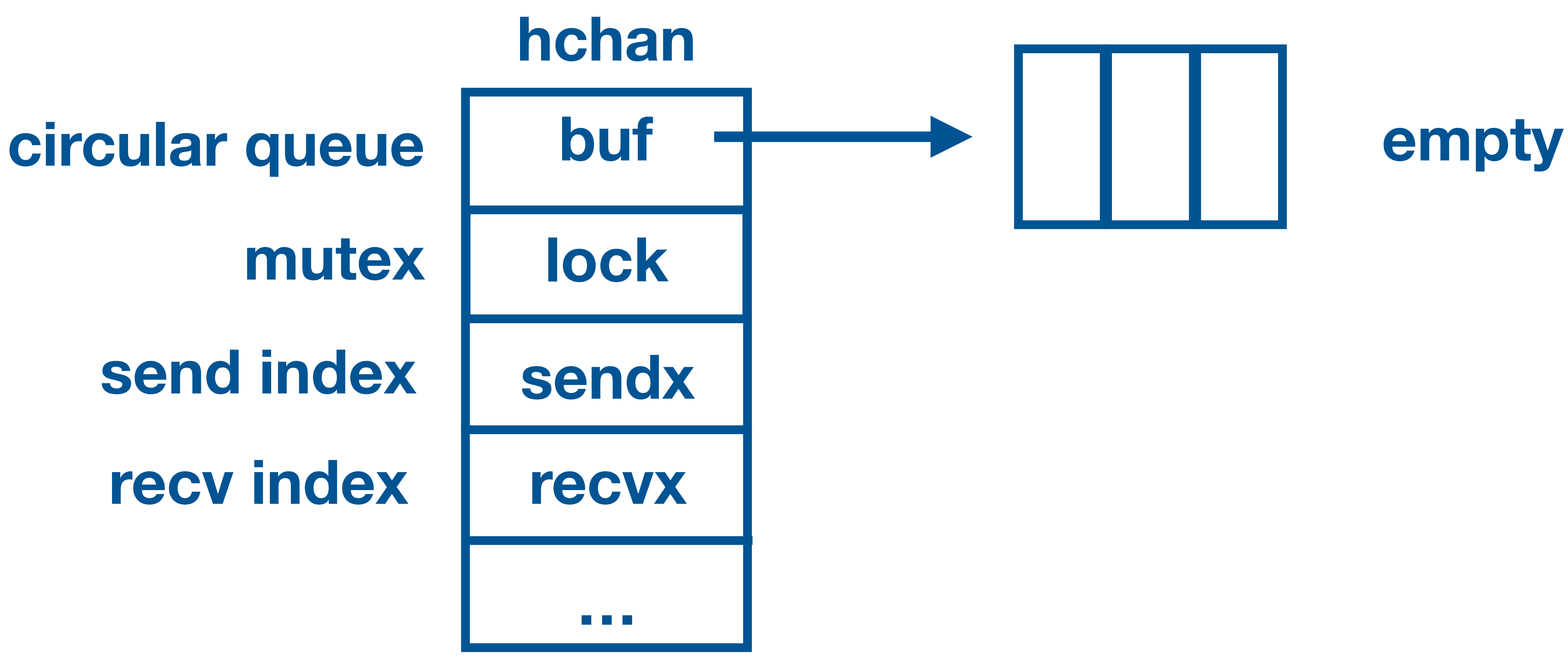
```go
ch := make(chan int, 3)
```

```
ch: <chan int>
    qcount: 0
    dataqsiz: 3
  > buf: <*[3]int>(0xc00013a060)
    elemsize: 8
    closed: 0
  > elemtype: <*runtime._type>(0x10d0660)
    sendx: 0
    recvx: 0
  > recvq: <waitq<int>>
  > sendq: <waitq<int>>
  > lock: <runtime.mutex>
```

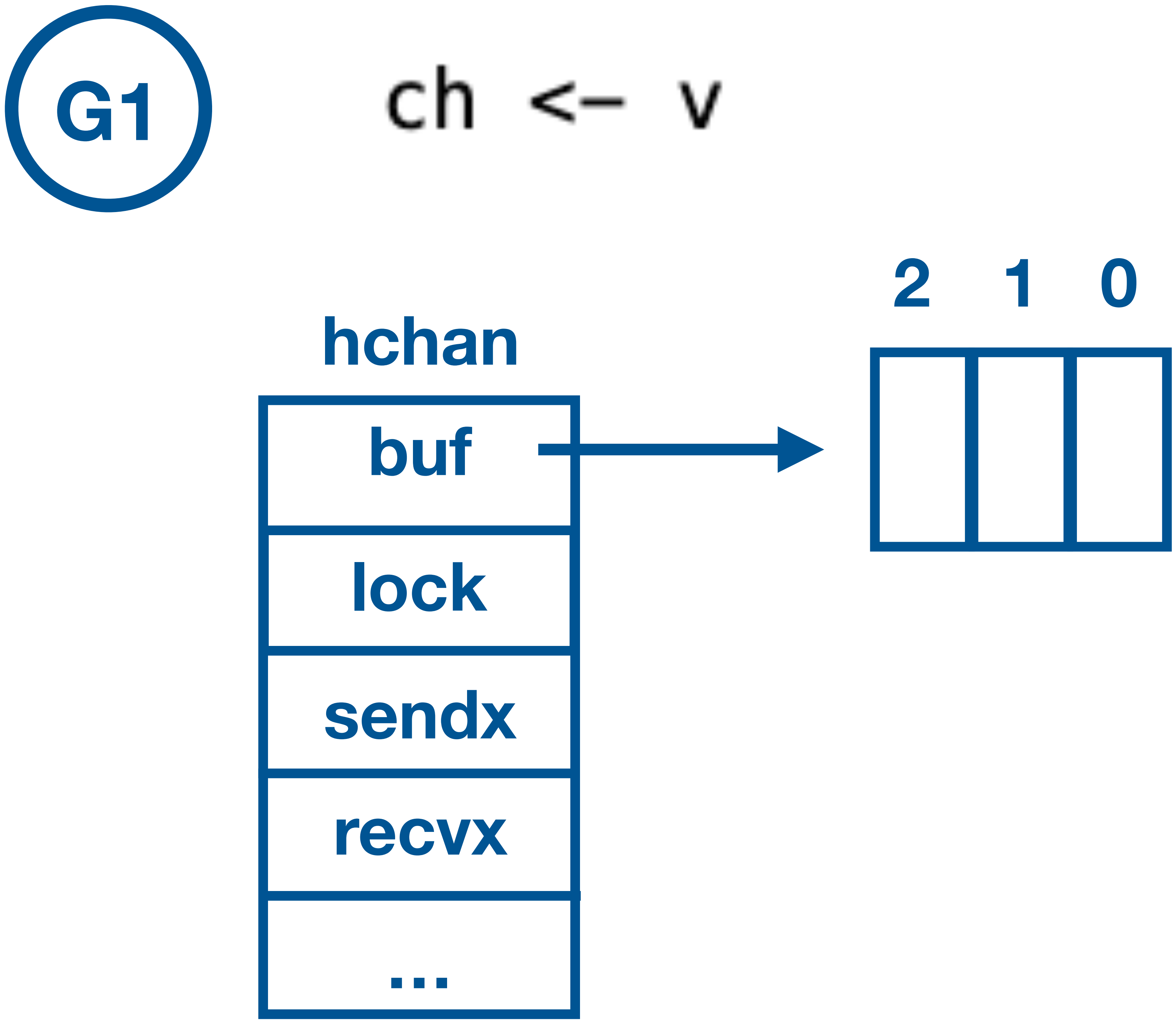
Send and Receive buffered channels

```go
ch := make(chan int, 3)

// G1 - goroutine
func G1(ch chan<- int) {
  for _, v := range []int{1, 2, 3, 4} {
    ch <- v
  }
}

// G2 - goroutine
func G2(ch <-chan int) {
  for v := range ch {
    fmt.Println(v)
  }
```

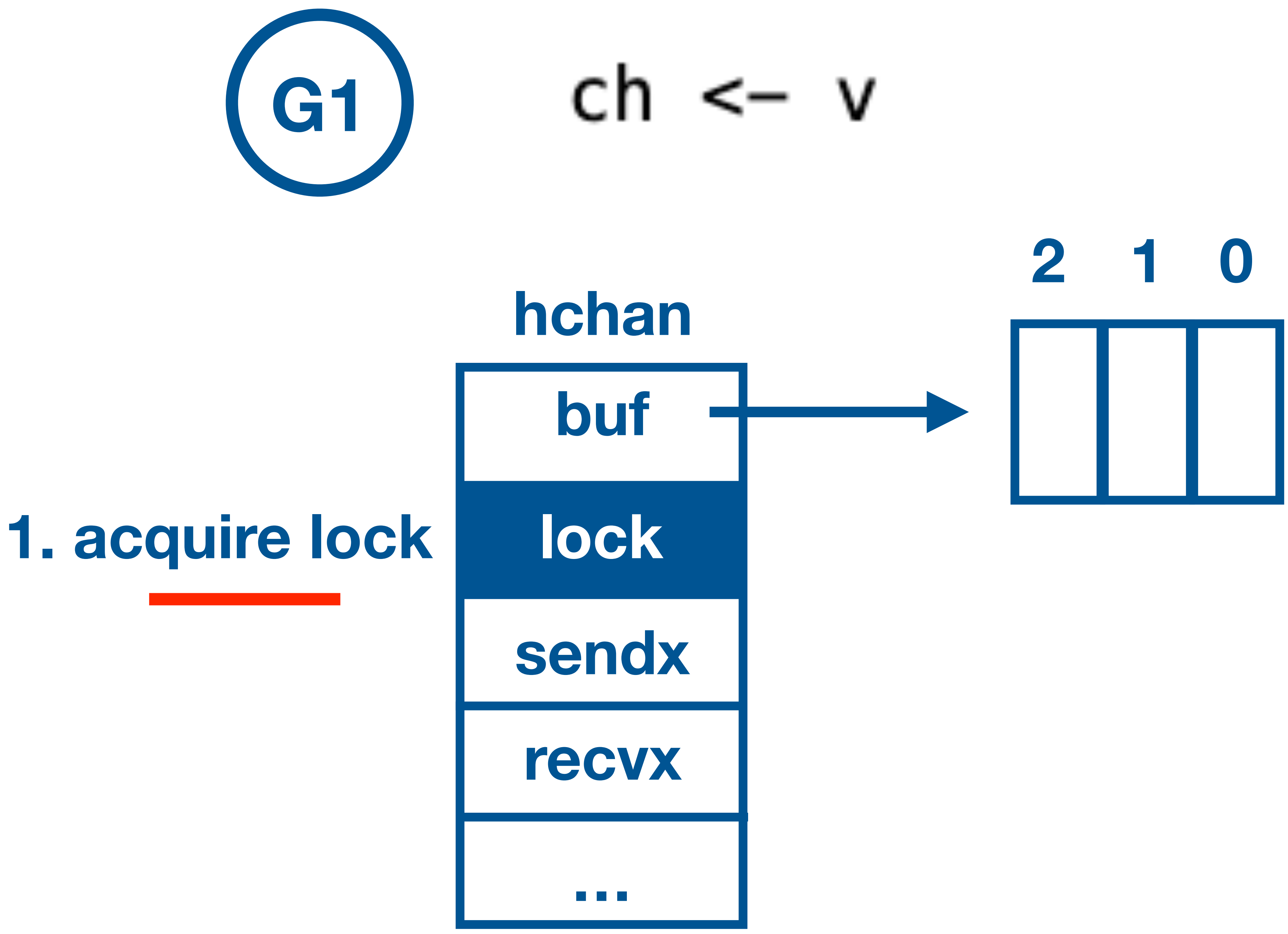
```go
ch := make(chan int, 3)
```



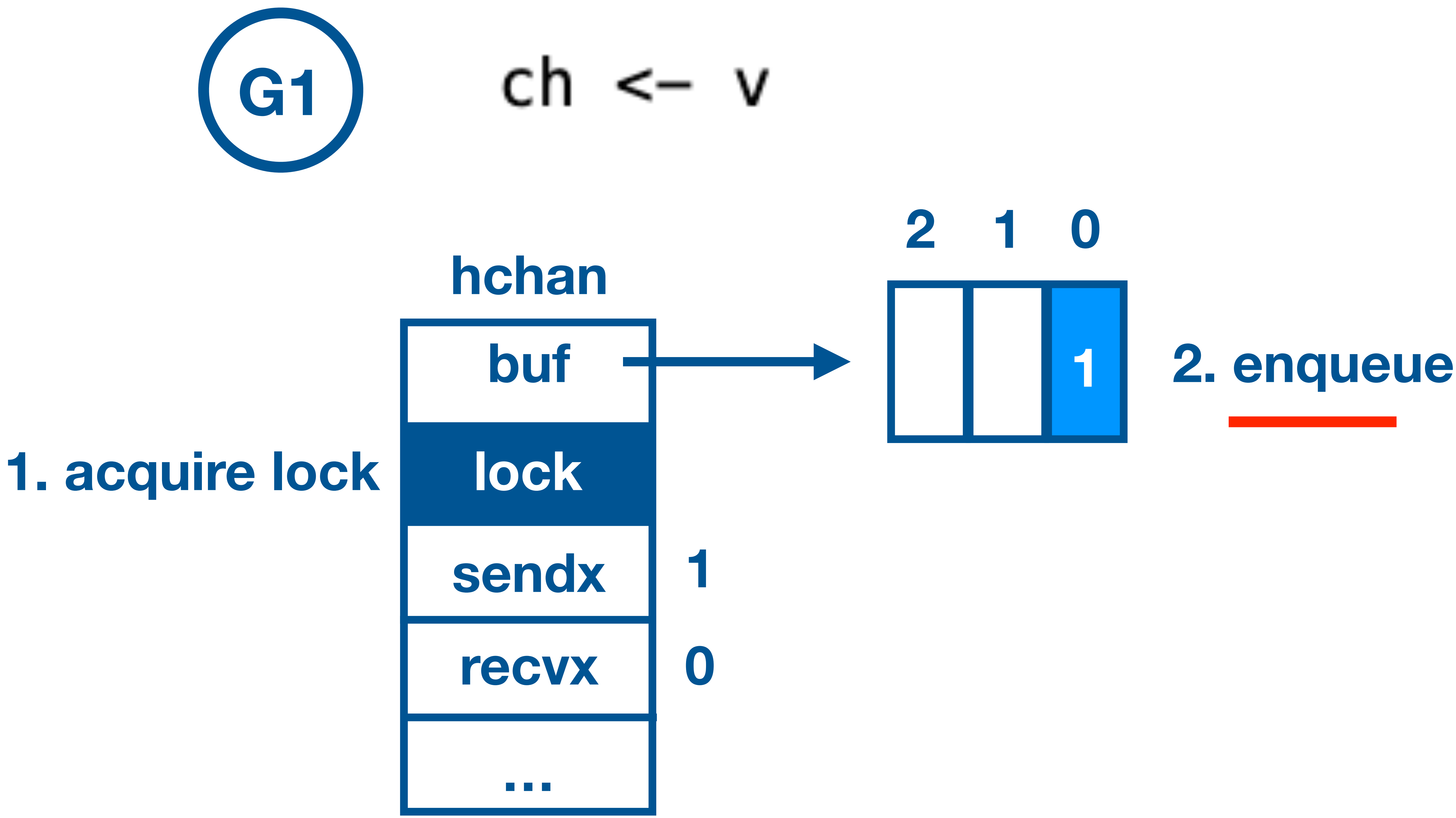
Scenario-1 : G1 executes first.



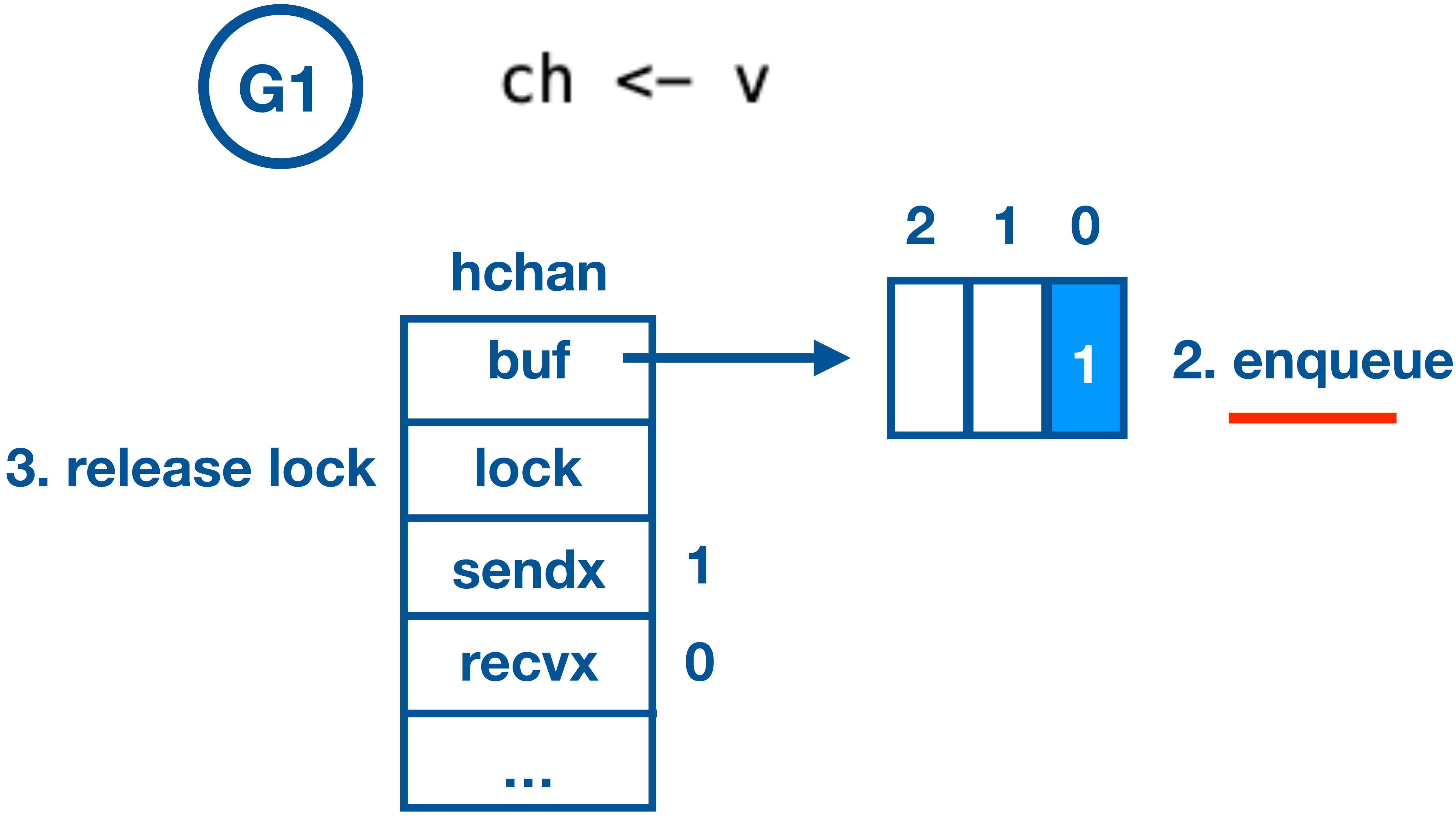
Scenario-1 : G1 executes first.

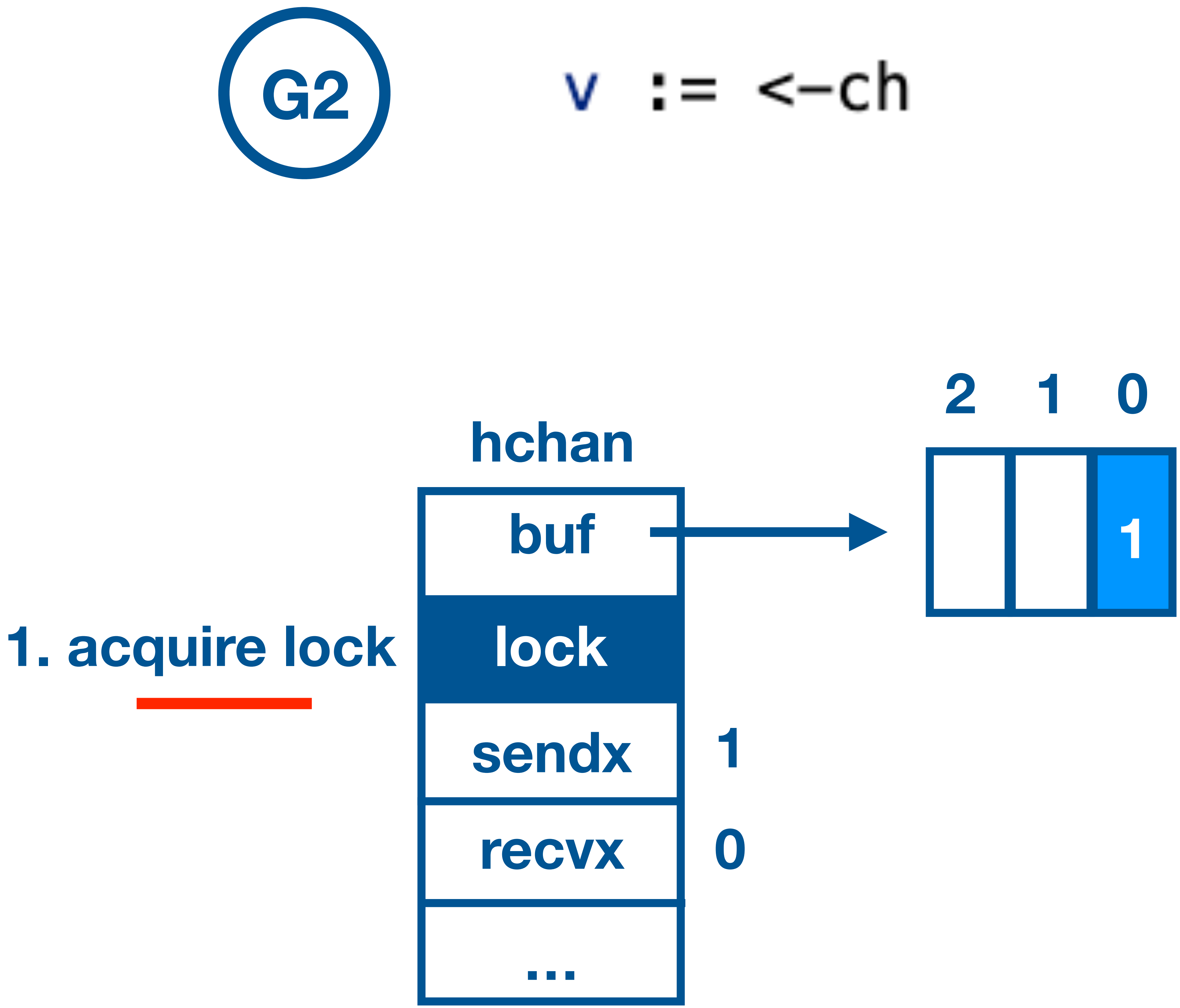


Scenario-1 : G1 executes first.



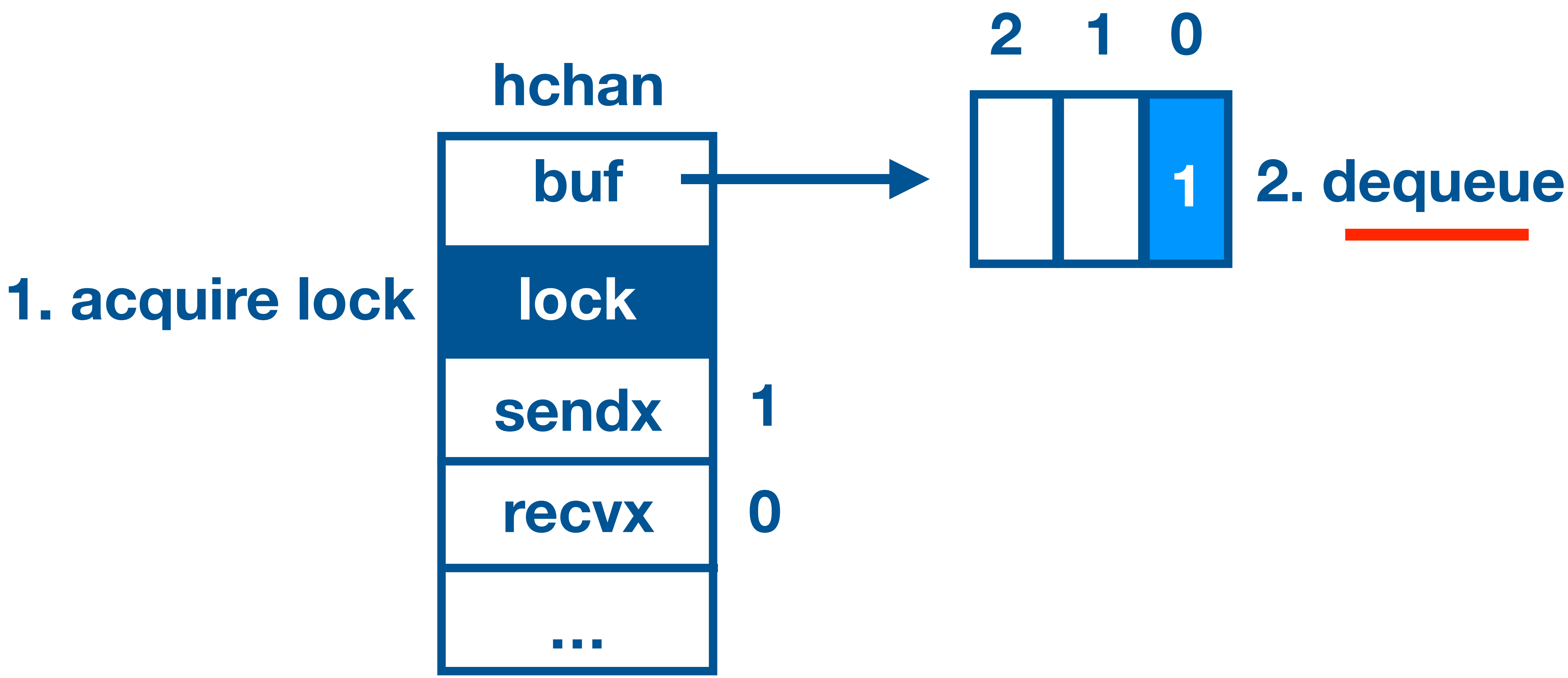
Scenario-1 : G1 executes first.

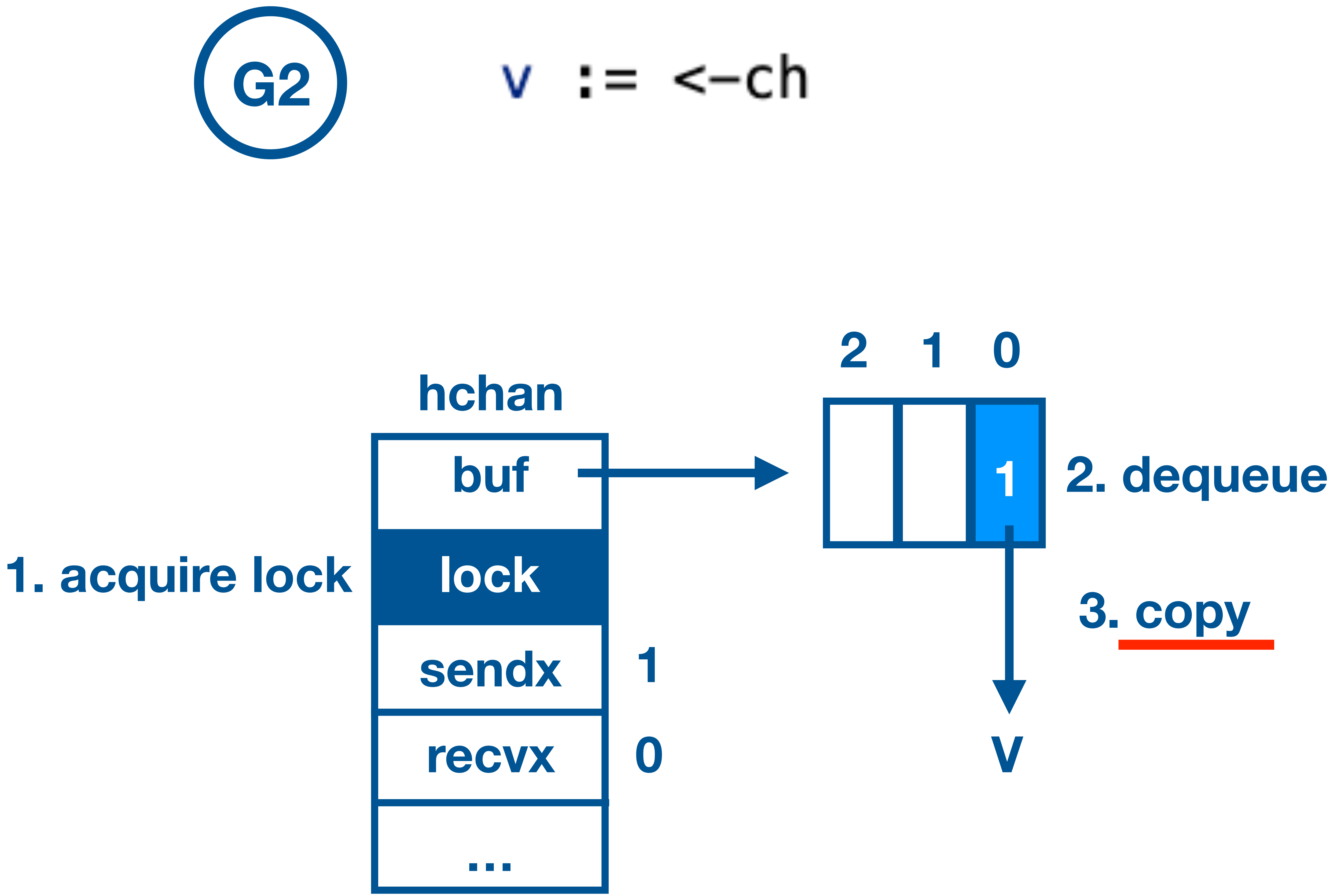



G2
v := <-ch
hchan
buf
1. acquire lock
lock
sendx
1
recvx
0
...
2
1
0
1

G2

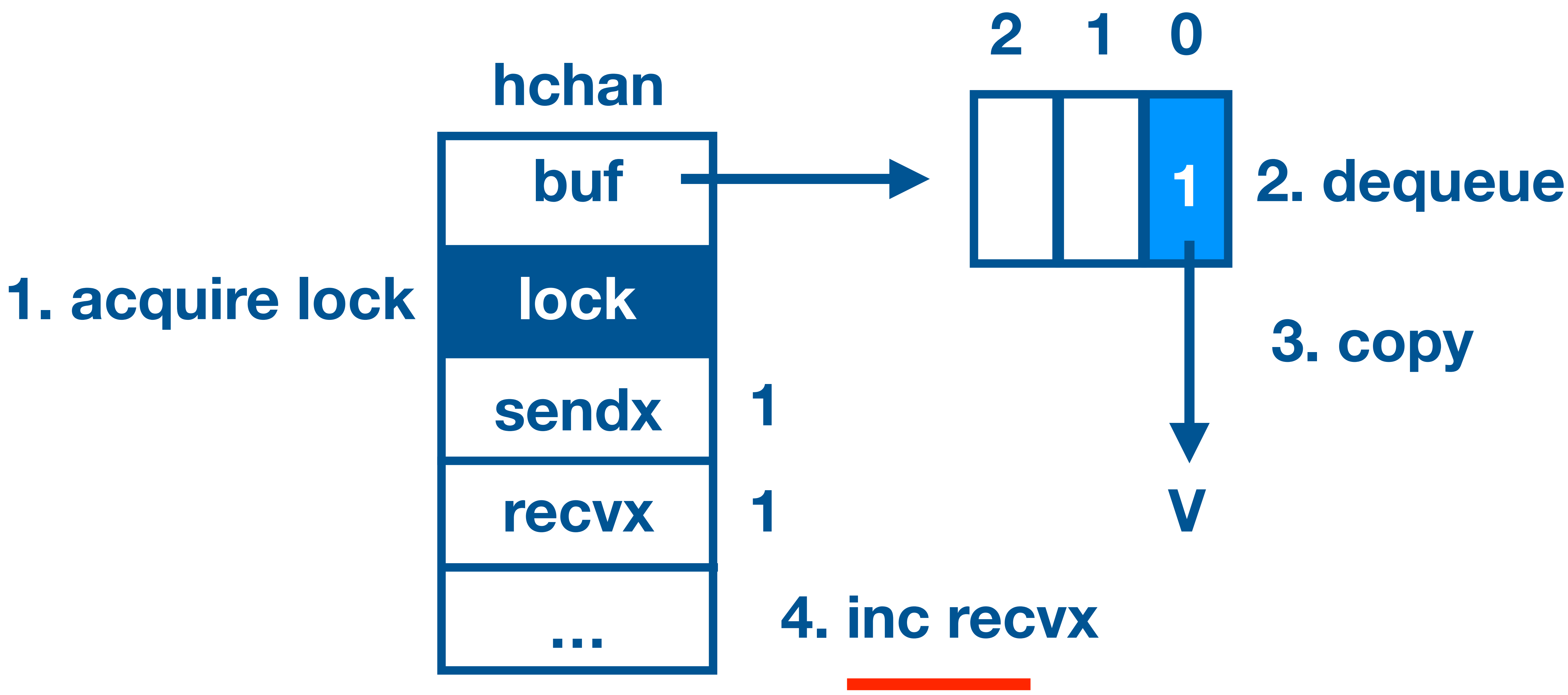
```
v := <-ch
```

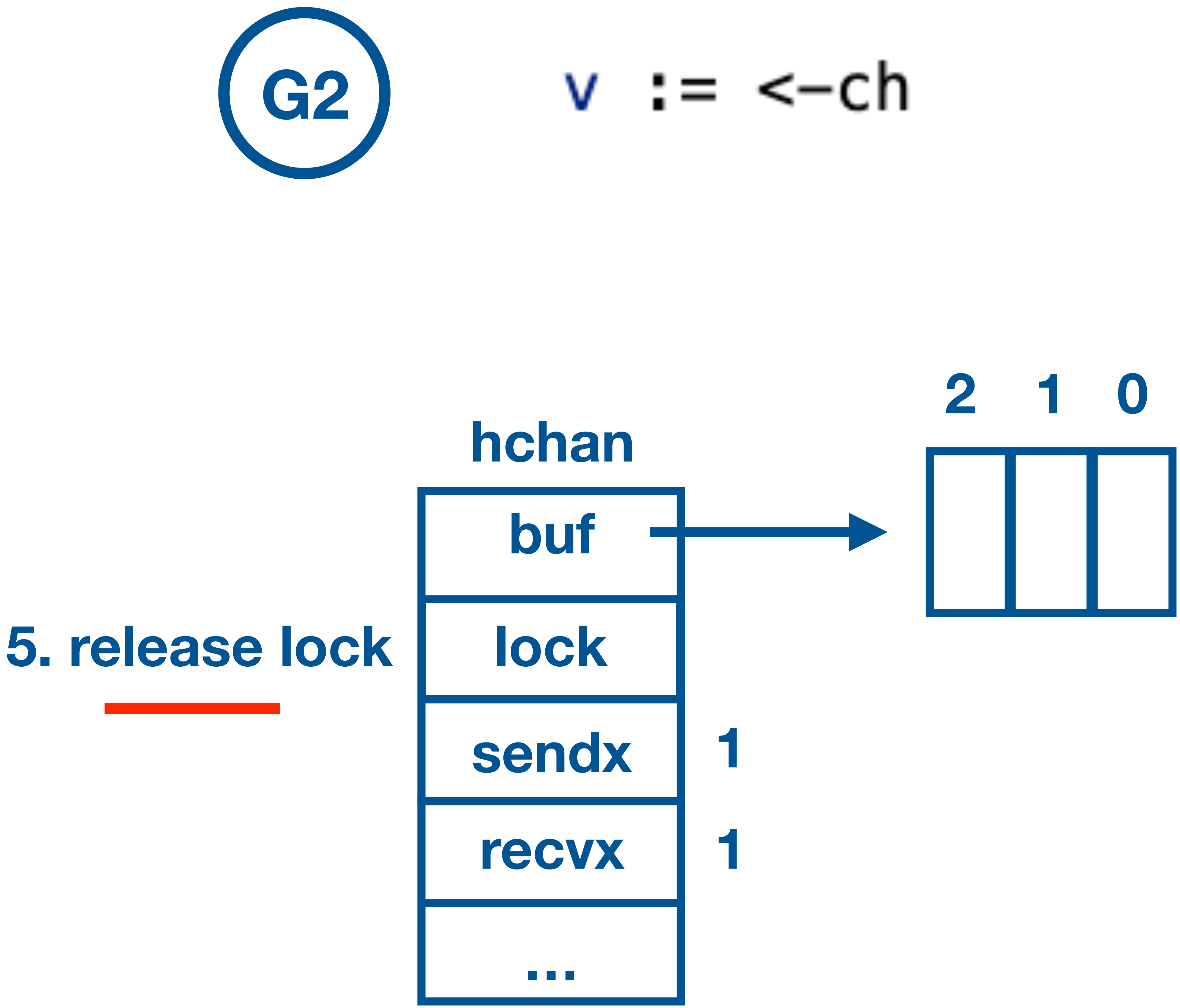



G2
v := <-ch
hchan
buf
lock
sendx
recvx
...
1. acquire lock
1
0
2 1 0
1
2. dequeue
3. copy
v

G2

```
v := <-ch
```


hchan
buf
lock
sendx
recvx
...
1
1
2
1
0
1
1. acquire lock
2. dequeue
3. copy
v
4. inc recvx


G2
v := <-ch
hchan
buf
lock
sendx
recvx
...
1
1
2
1
0
5. release lock

- There is no memory share between goroutines
- Goroutines copy elements into and from hchan
- hchan is protected by mutex lock.

“Do not communicate by sharing memory;

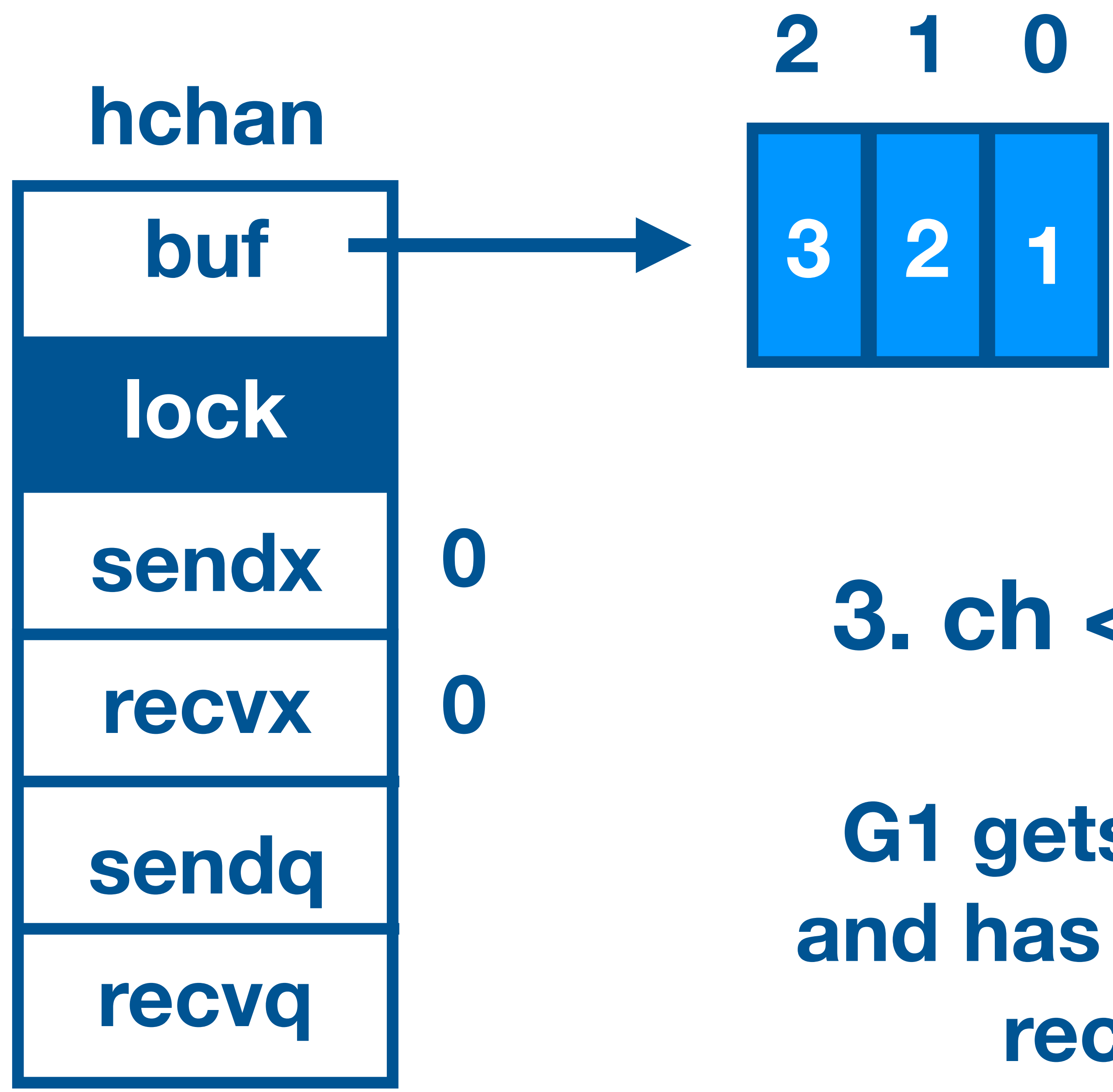
instead, share memory by communicating”

Buffer Full Scenario

G1

```go
for _, v := range []int{1, 2, 3, 4} {
    ch <- v
}
```

1. enqueue 1, 2, 3

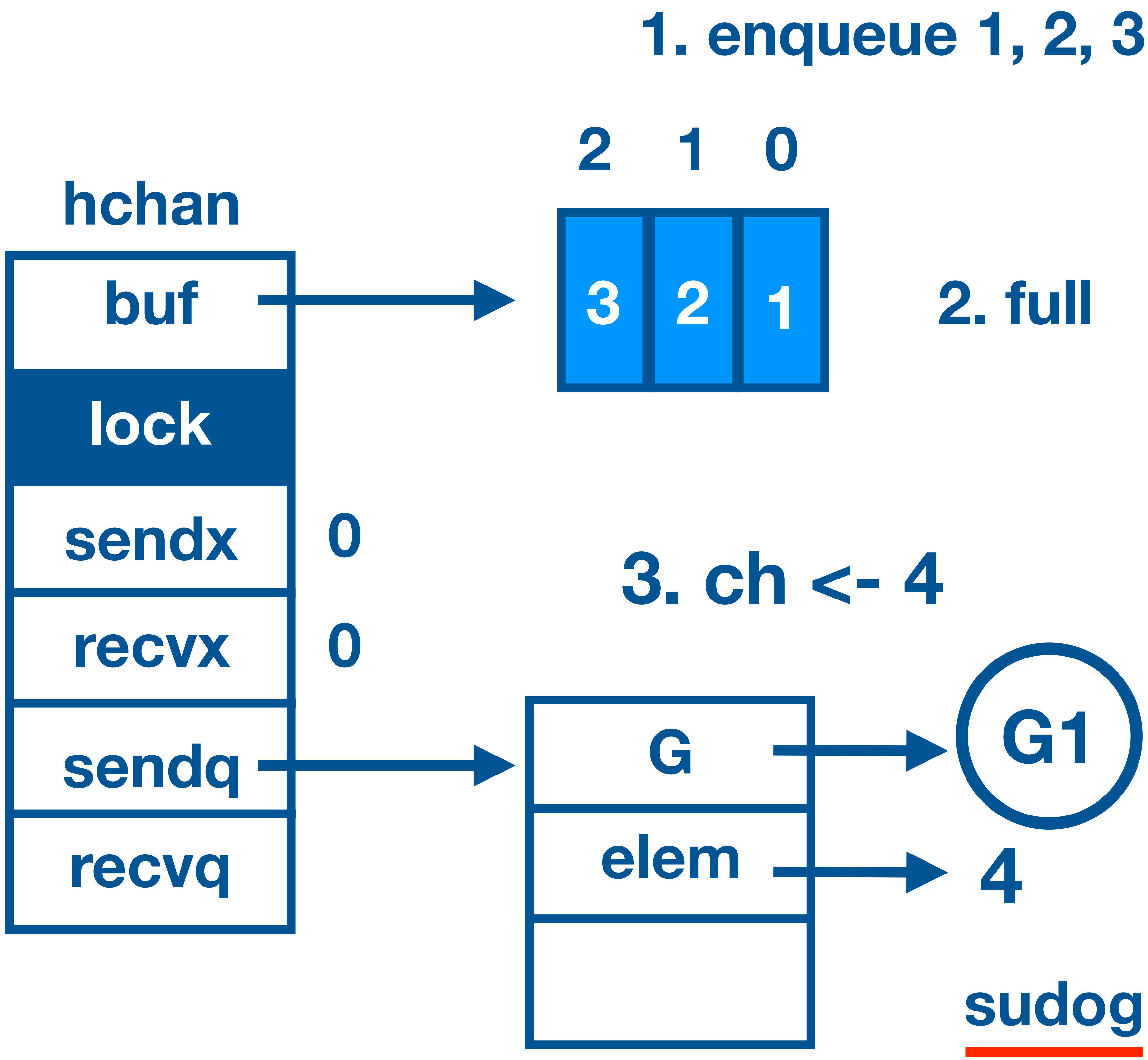


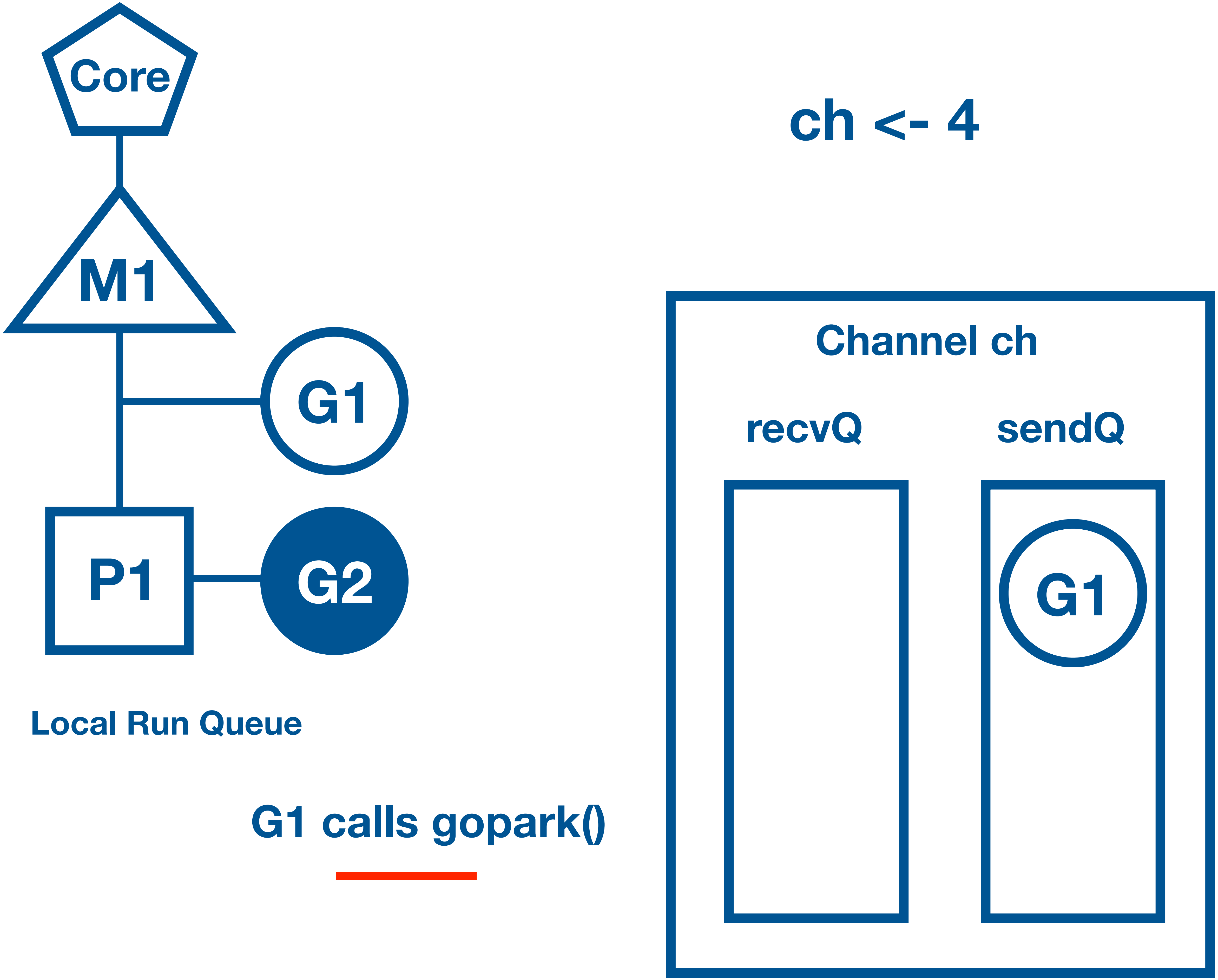
2. full

3. ch <- 4

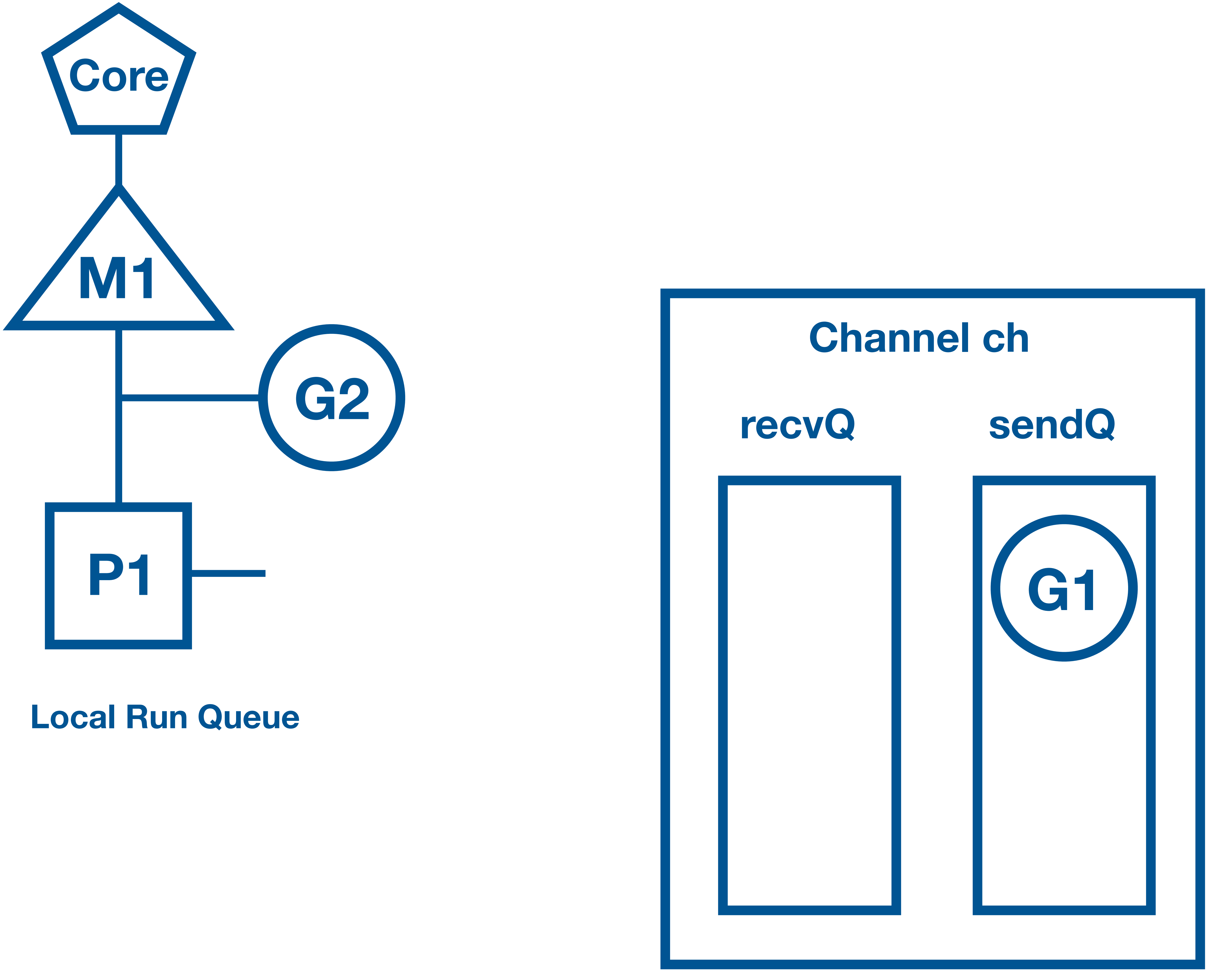
G1 gets blocked and has to wait for receiver

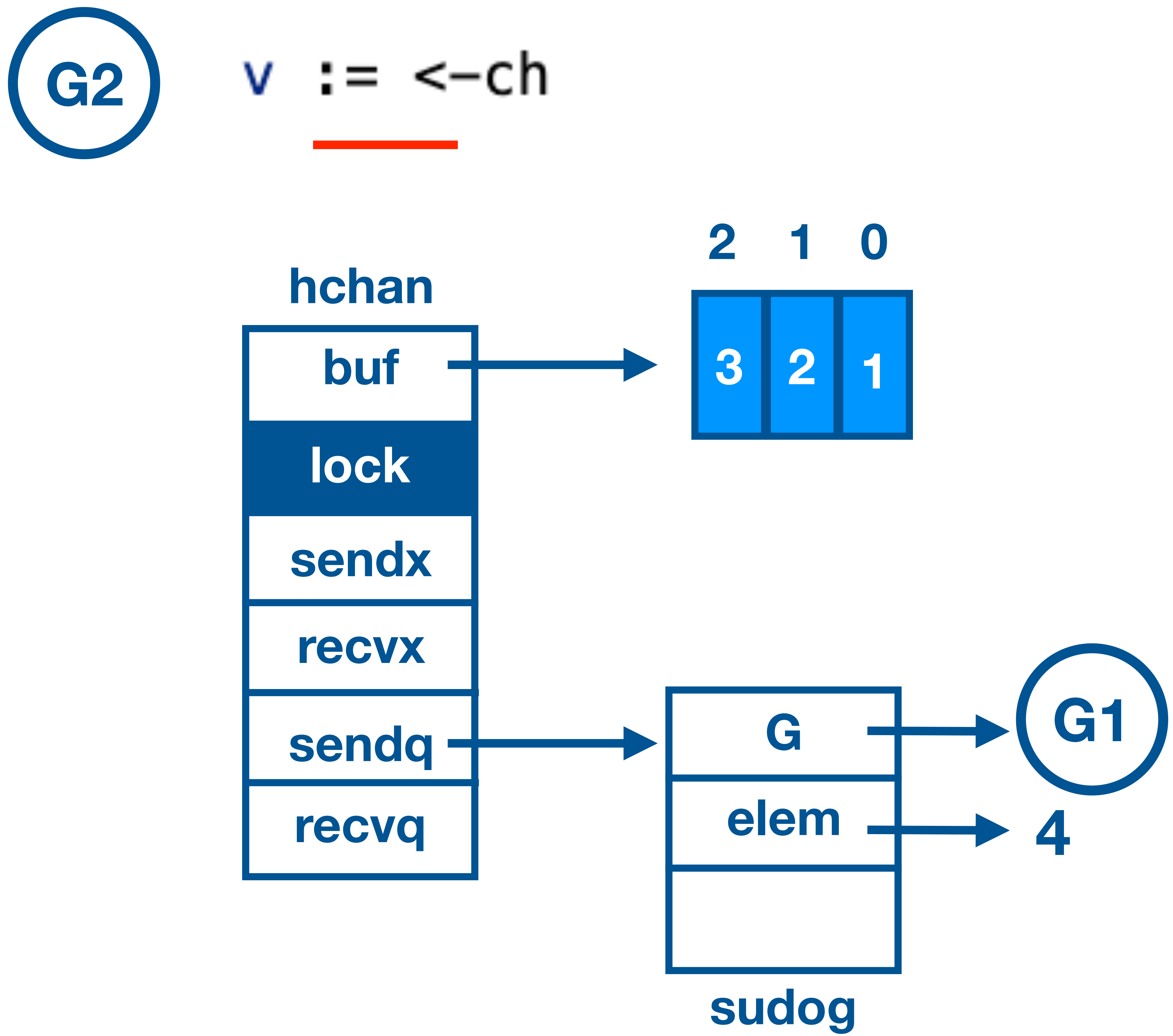

```go
for _, v := range []int{1, 2, 3, 4} {
	ch <- v
}
```

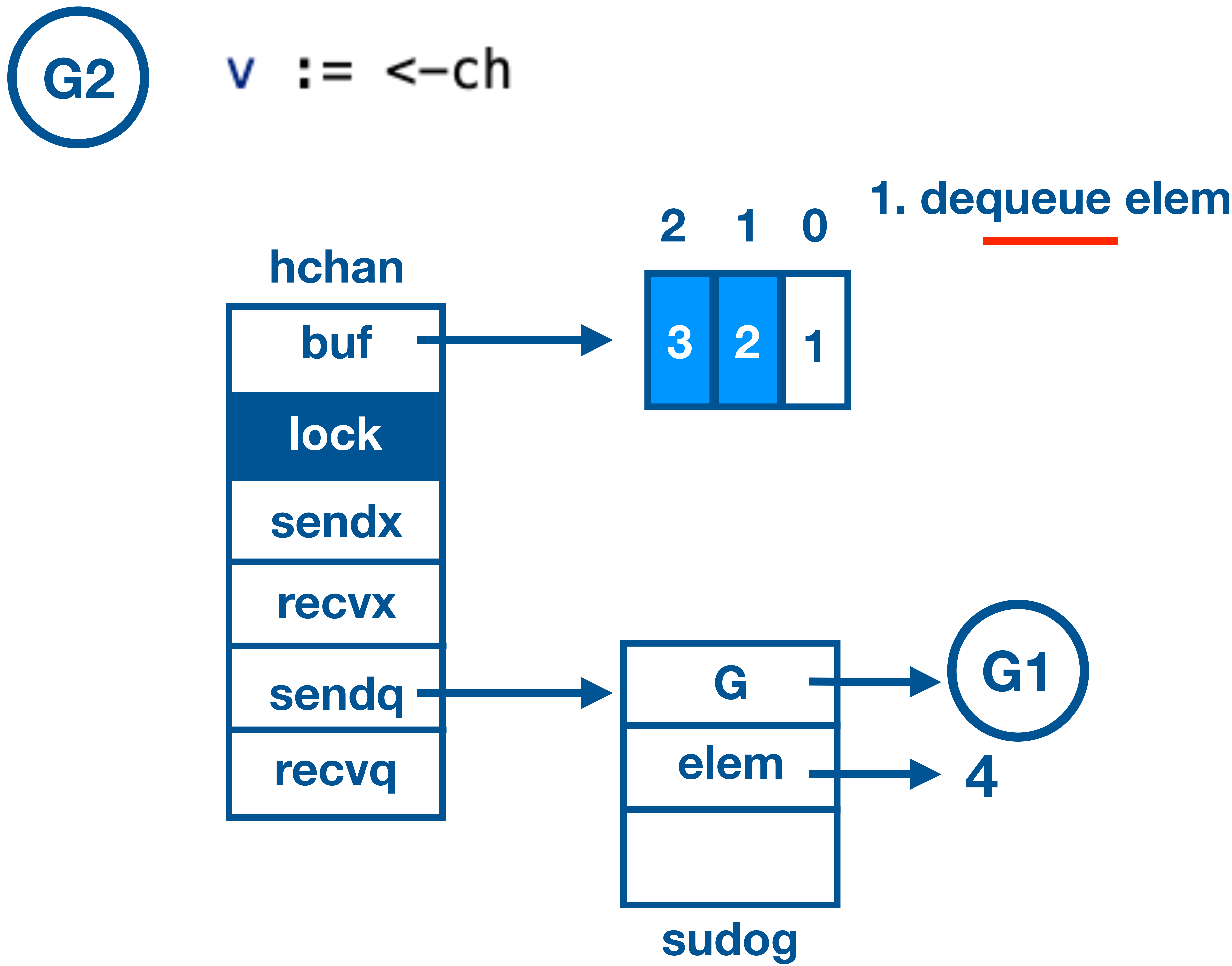




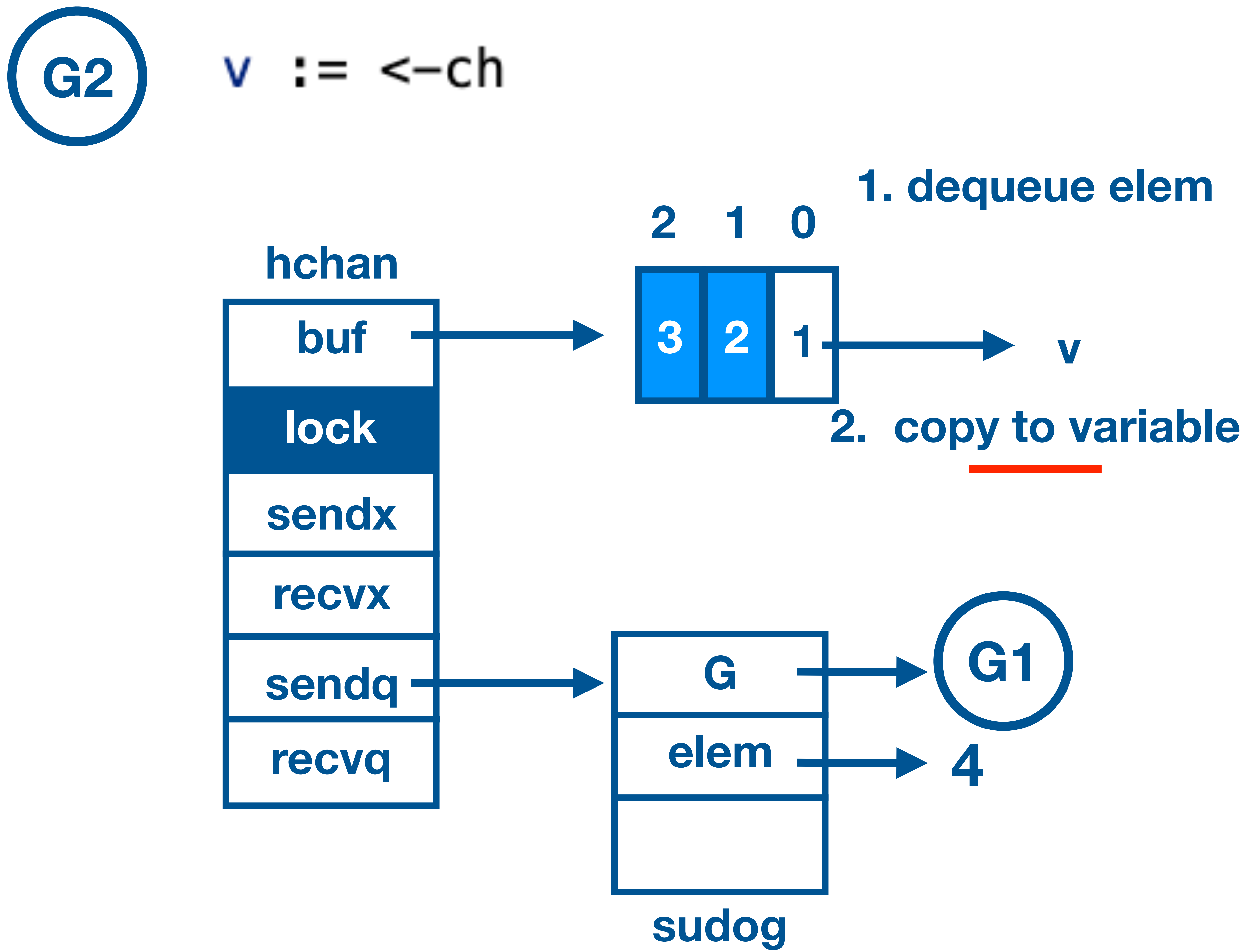
Core
M1
G1
P1
G2
Local Run Queue
G1 calls gopark()
ch <- 4
Channel ch
recvQ
sendQ
G1

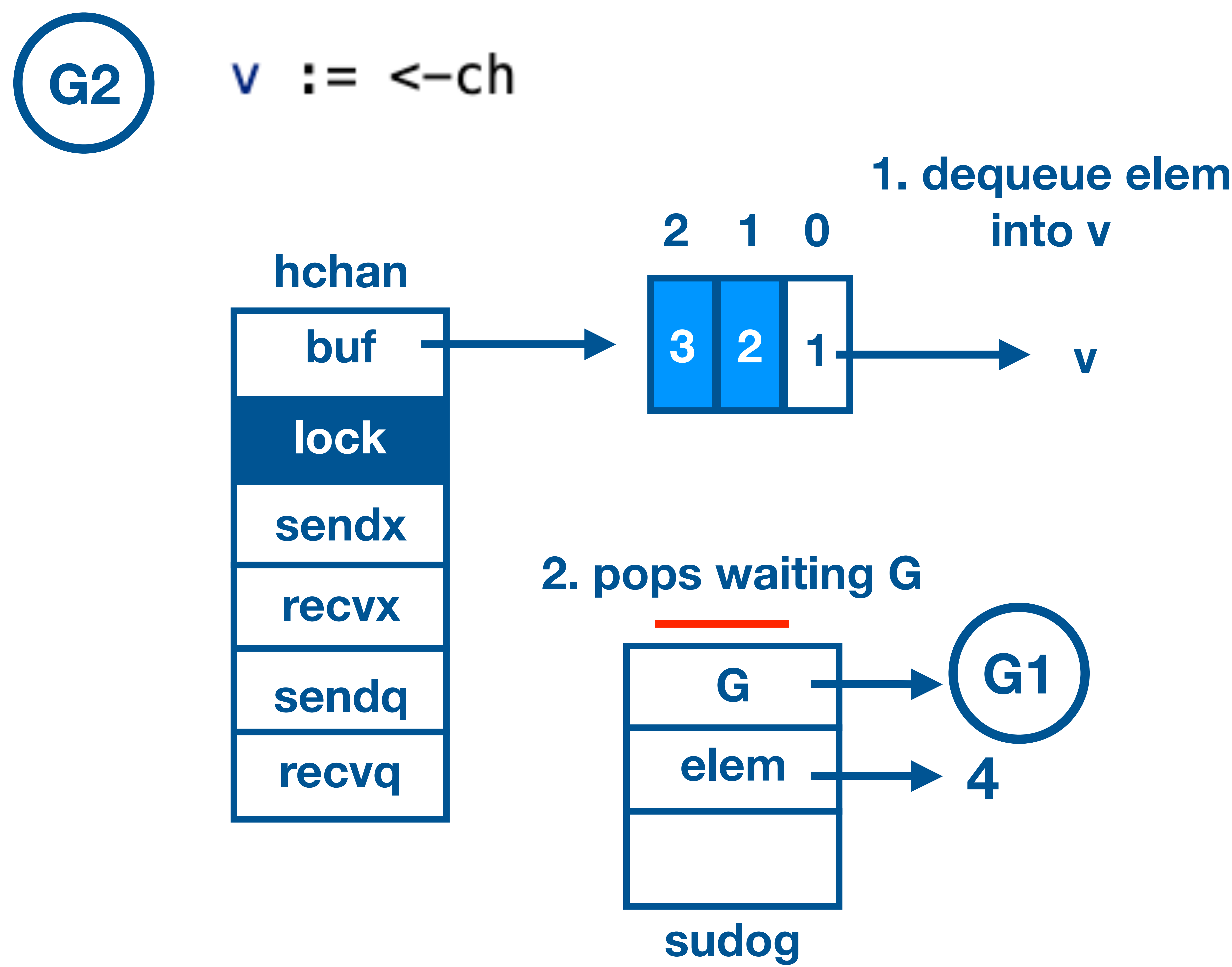

Core
M1
G2
P1
Local Run Queue
Channel ch
recvQ
sendQ
G1

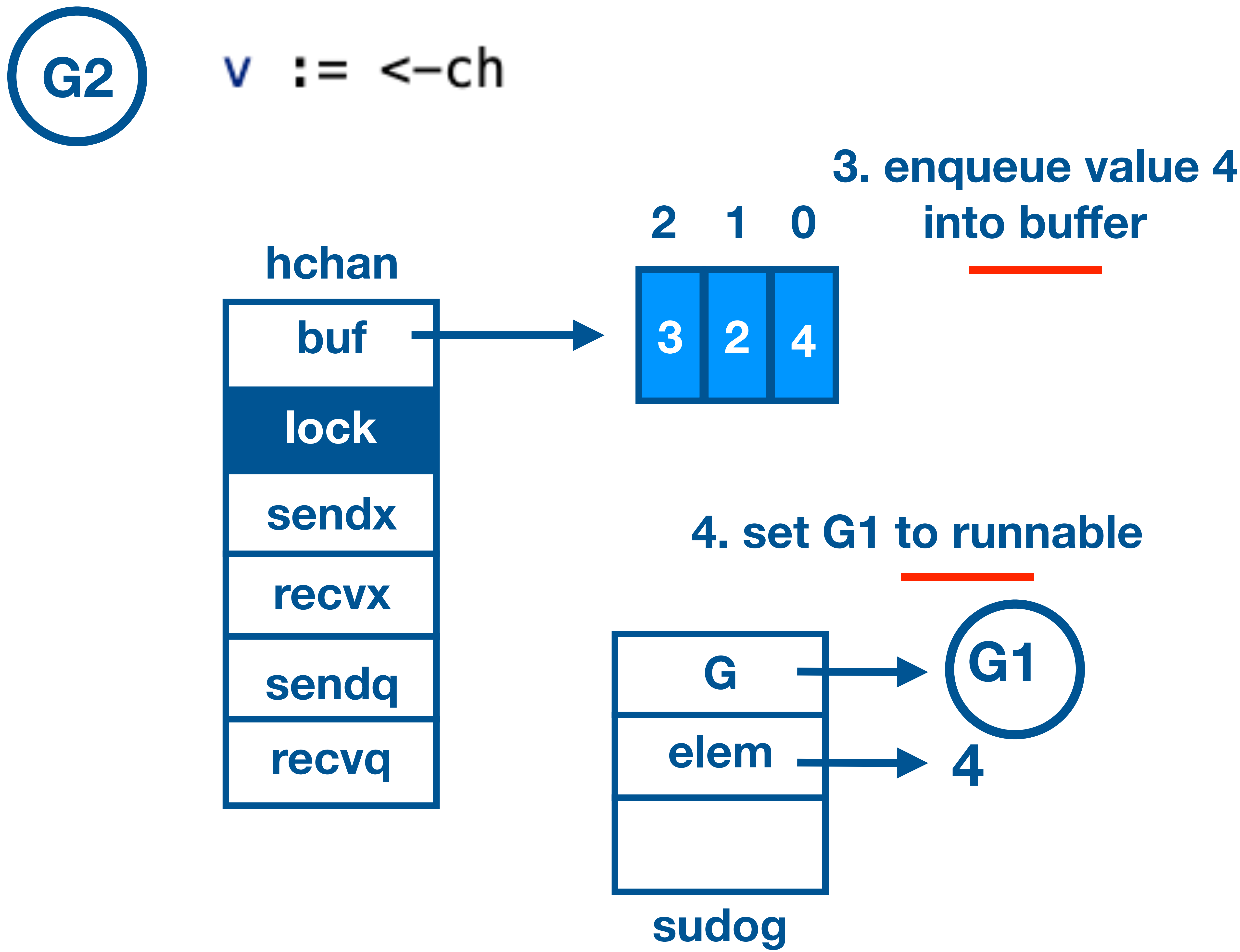

G2
v := <-ch
hchan
buf
lock
sendx
recvx
sendq
recvq
2 1 0
3 2 1
G
elem
sudog
G1
4

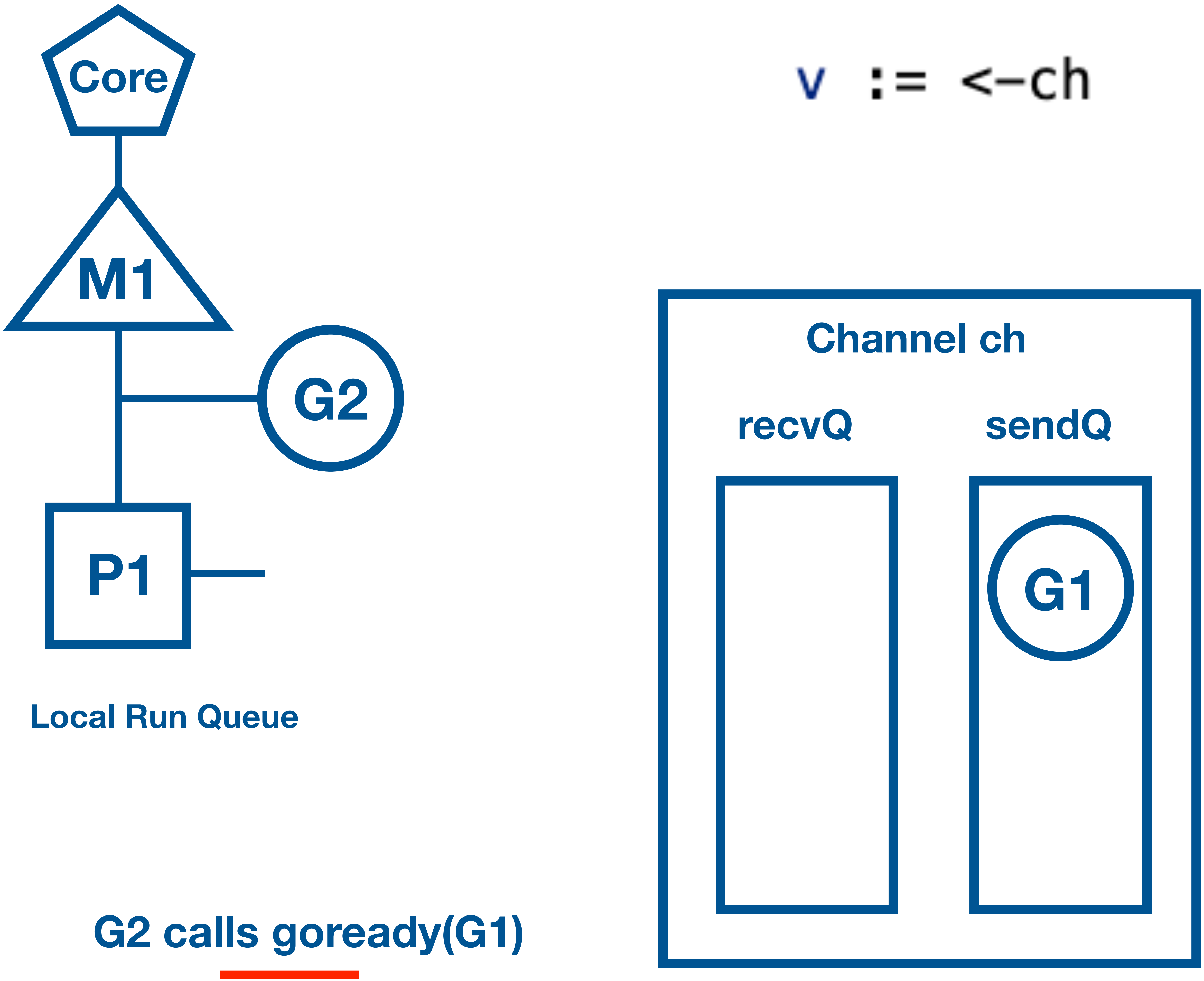


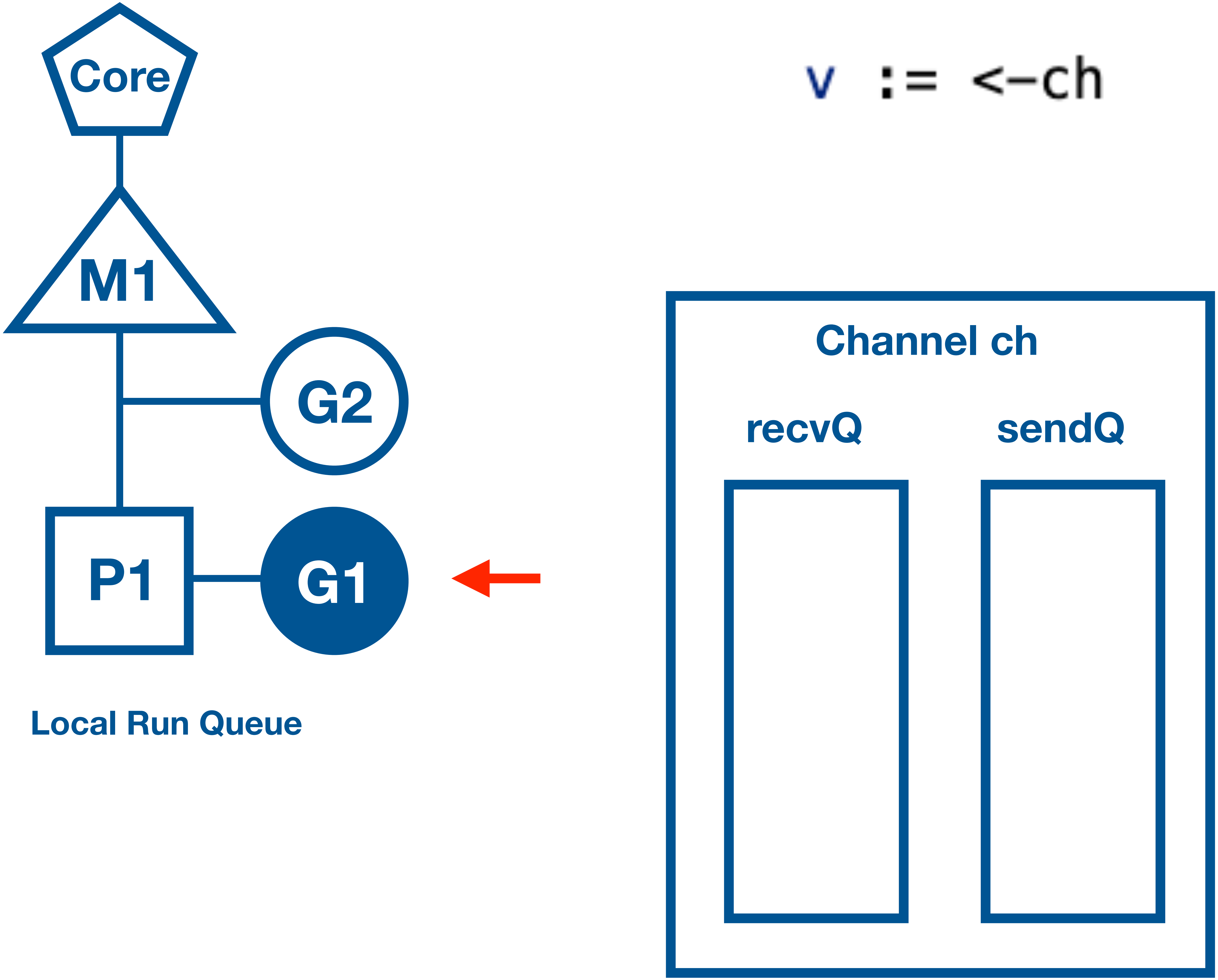
G2
v := <-ch
1. dequeue elem
2 1 0
hchan
buf
lock
sendx
recvx
sendq
recvq
3
2
1
G
elem
sudog
G1
4


G2
v := <-ch
1. dequeue elem
2 1 0
hchan
buf
3
2
1
v
lock
2. copy to variable
sendx
recvx
sendq
G
G1
recvq
elem
4
sudog


G2
v := <-ch
1. dequeue elem
into v
2 1 0
hchan
buf
3
2
1
v
lock
sendx
recvx
sendq
recvq
2. pops waiting G
G
G1
elem
4
sudog


G2
v := <-ch
3. enqueue value 4
into buffer
2 1 0
hchan
buf
lock
sendx
recvx
sendq
recvq
3 2 4
4. set G1 to runnable
G
elem
G1
4
sudog


Core
M1
G2
P1
Local Run Queue
v := <-ch
Channel ch
recvQ
sendQ
G1
G2 calls goready(G1)



Core
M1
G2
P1
G1
Local Run Queue
v := <-ch
Channel ch
recvQ
sendQ

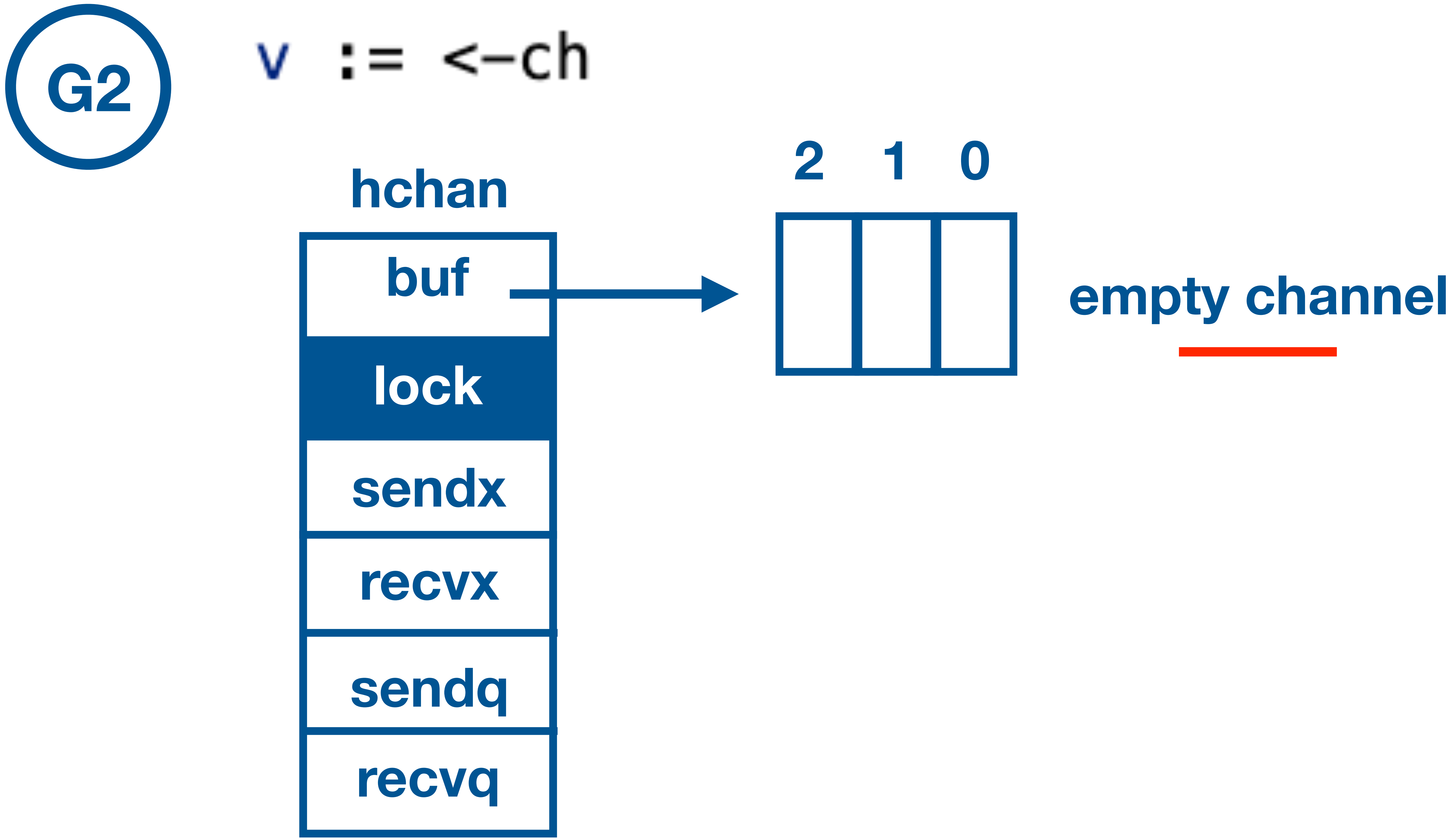
Summary

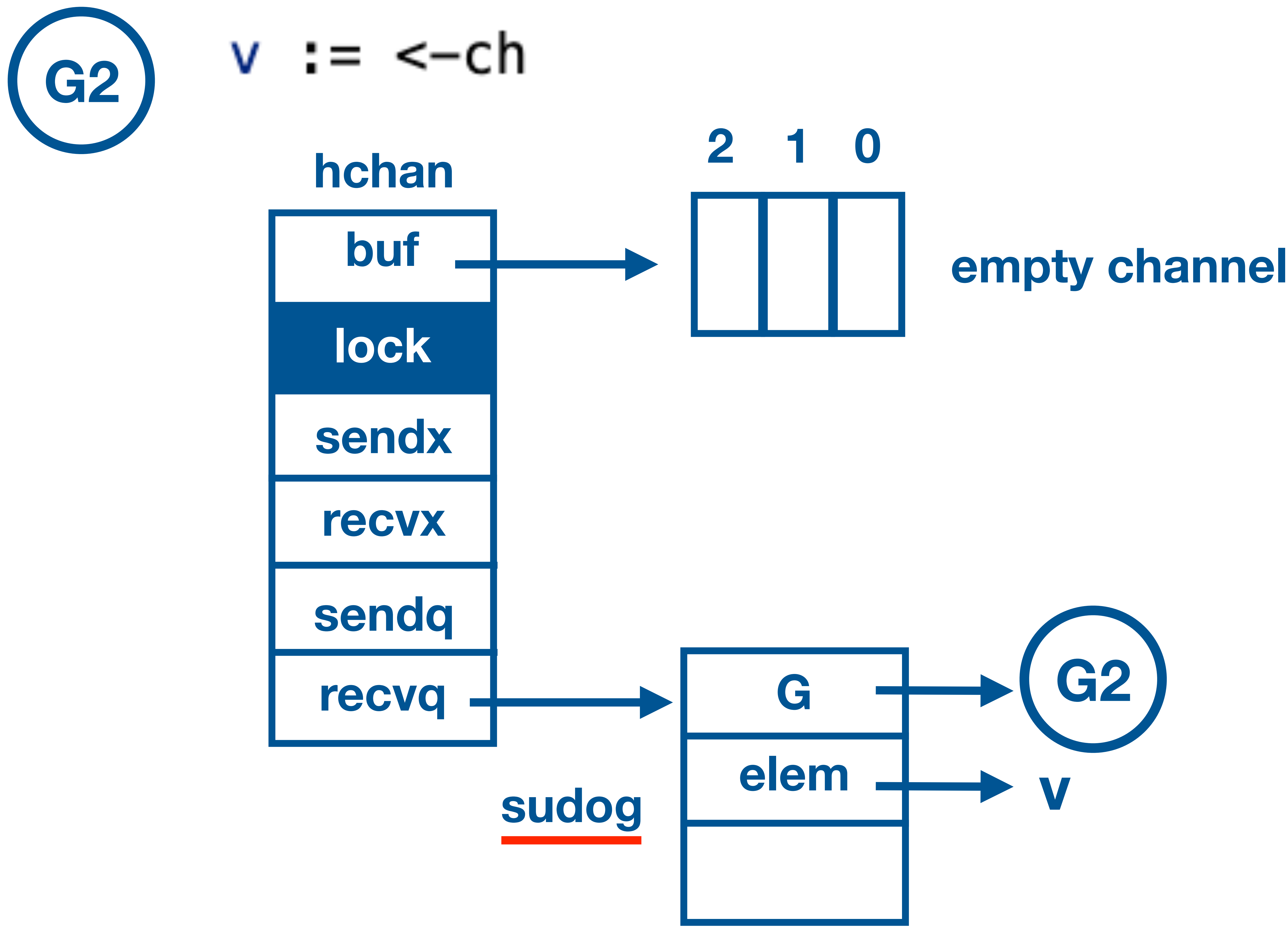
- when channel buffer is full and a goroutine tries to send value.
- Sender Goroutine gets blocked, it is parked on sendQ.
- Data will be saved in the elem field of the sudog structure.
- When Receiver comes along, it dequeues the value from buffer.
- Enqueues the data from elem field to the buffer.
- Pops the goroutine in sendq, and puts it into runnable state.

Buffer Empty Scenario

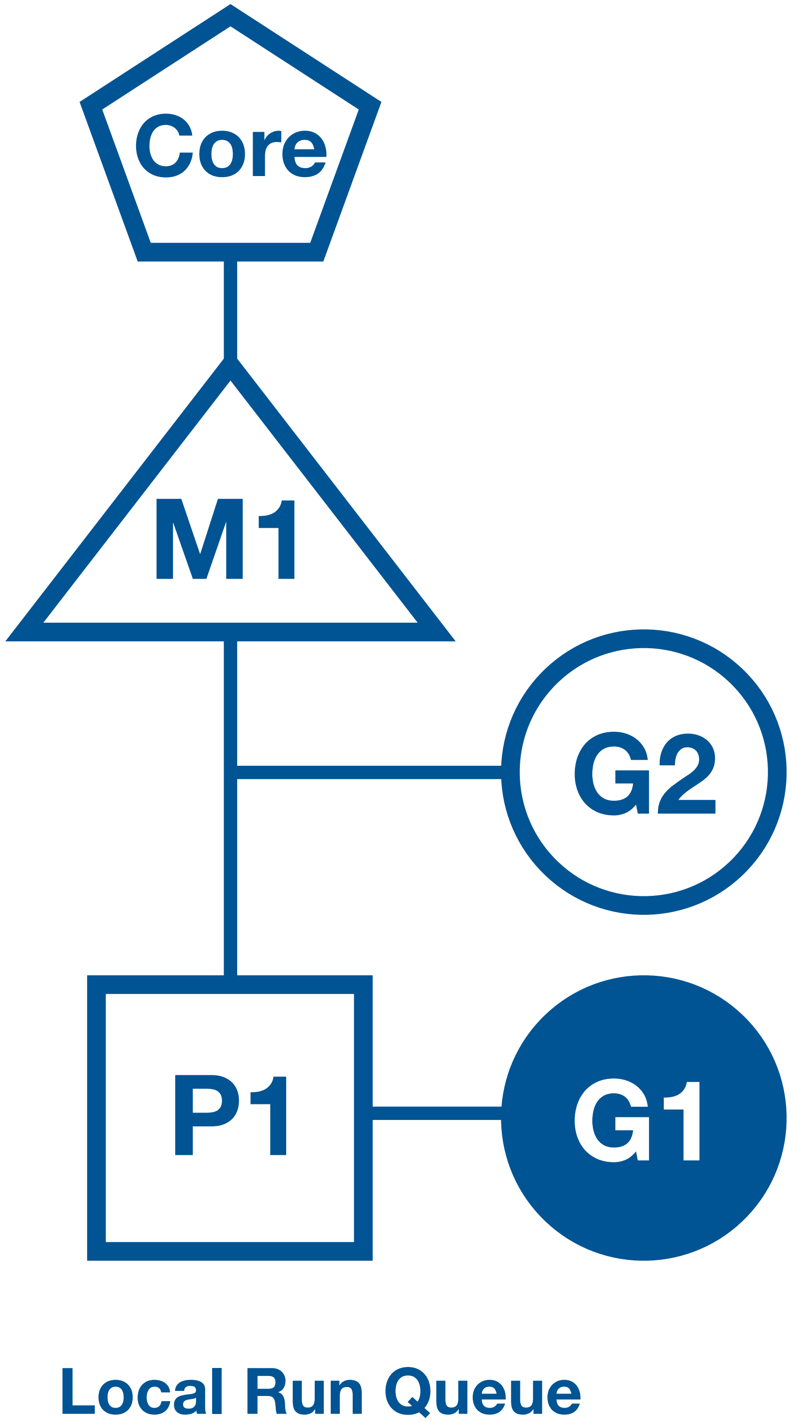
Scenario: G2 tries to recv on empty channel

```go
ch := make(chan int, 3)

// G1 - goroutine
func G1(ch chan<- int) {
    for _, v := range []int{1, 2, 3, 4} {
        ch <- v
    }
}

// G2 - goroutine
func G2(ch <-chan int) {
    for v := range ch {
        fmt.Println(v)
    }
}
```

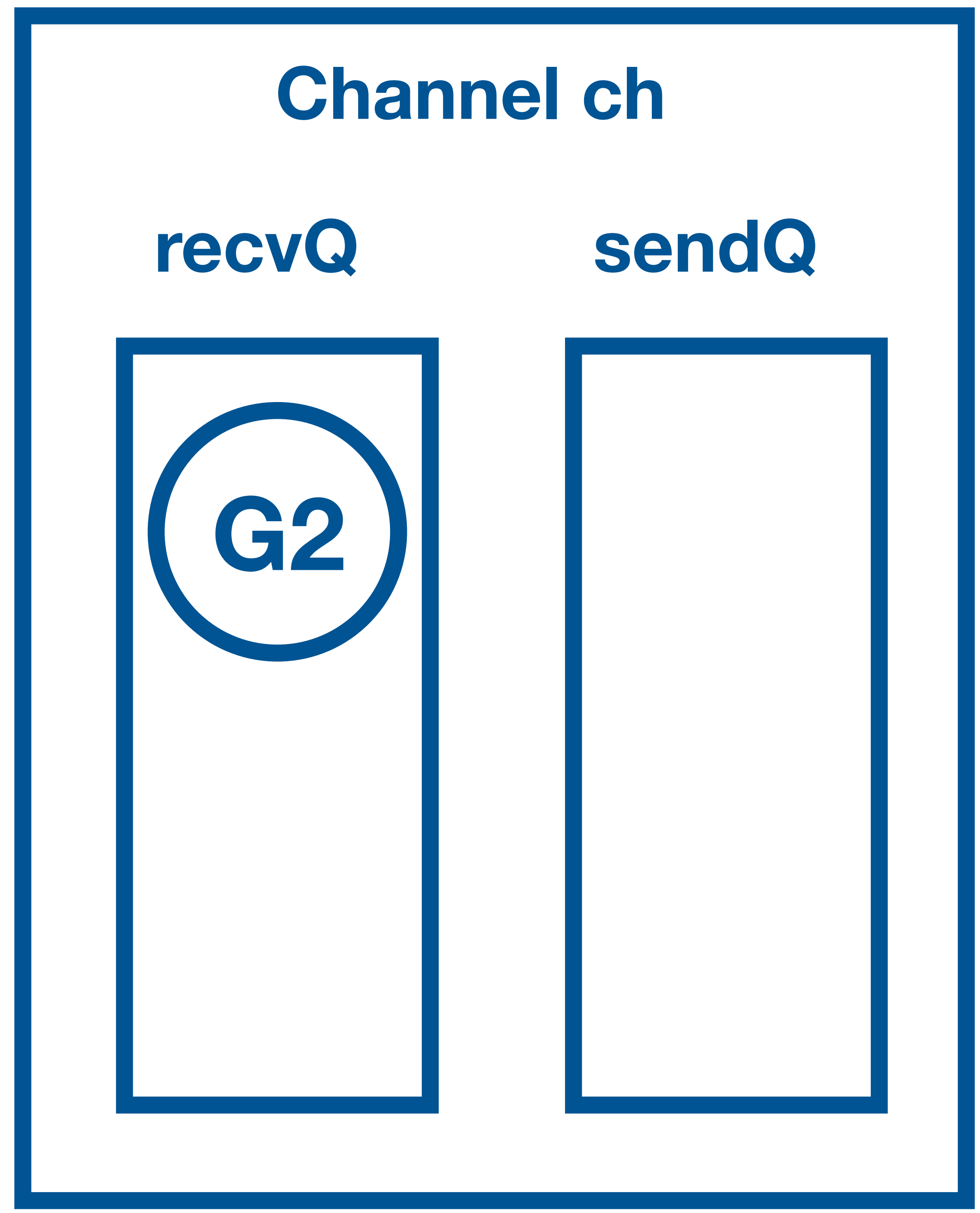

G2
v := <-ch
hchan
buf
lock
sendx
recvx
sendq
recvq
2
1
0
empty channel



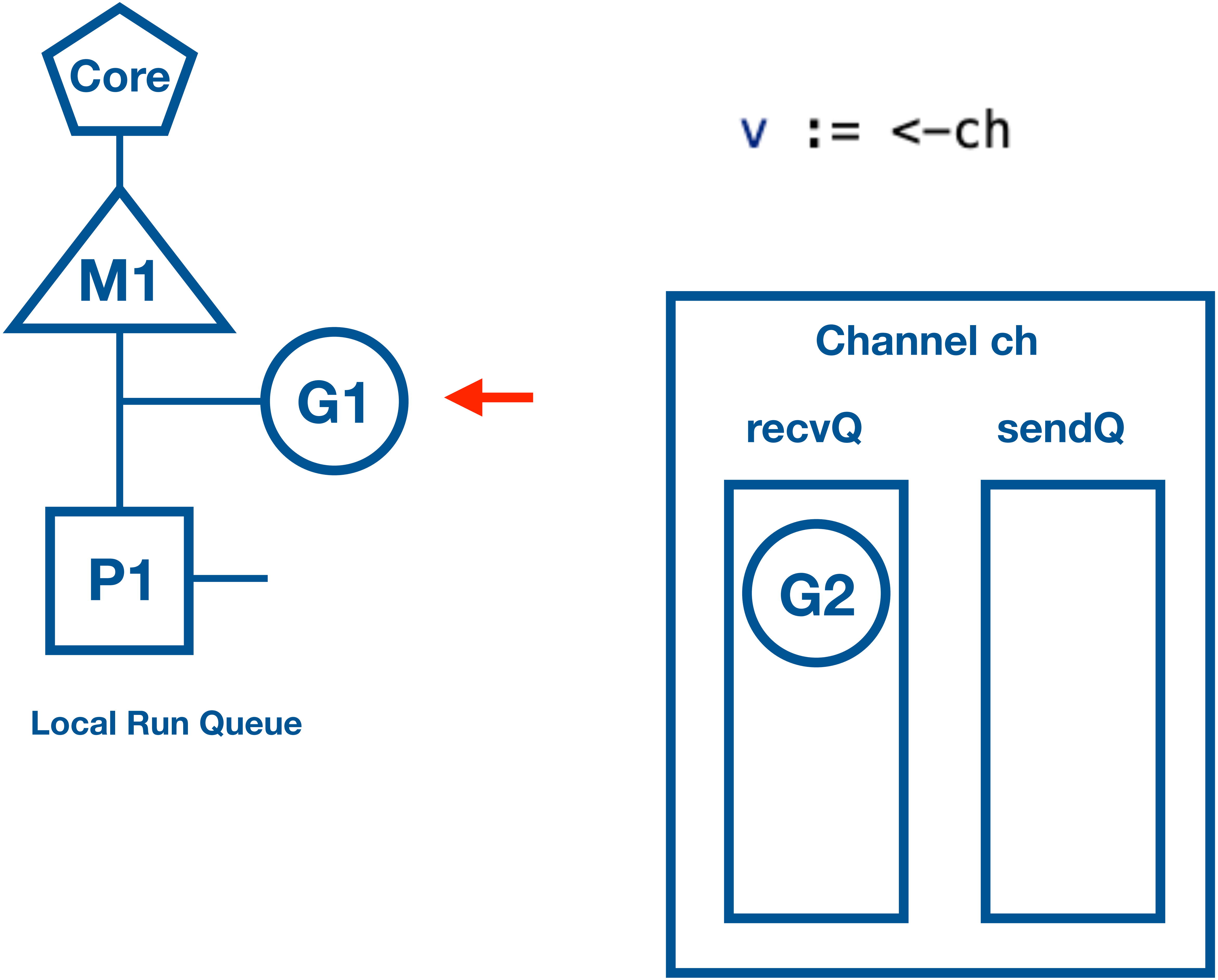
G2
v := <-ch
hchan
buf
lock
sendx
recvx
sendq
recvq
2 1 0
empty channel
sudog
G
elem
G2
v

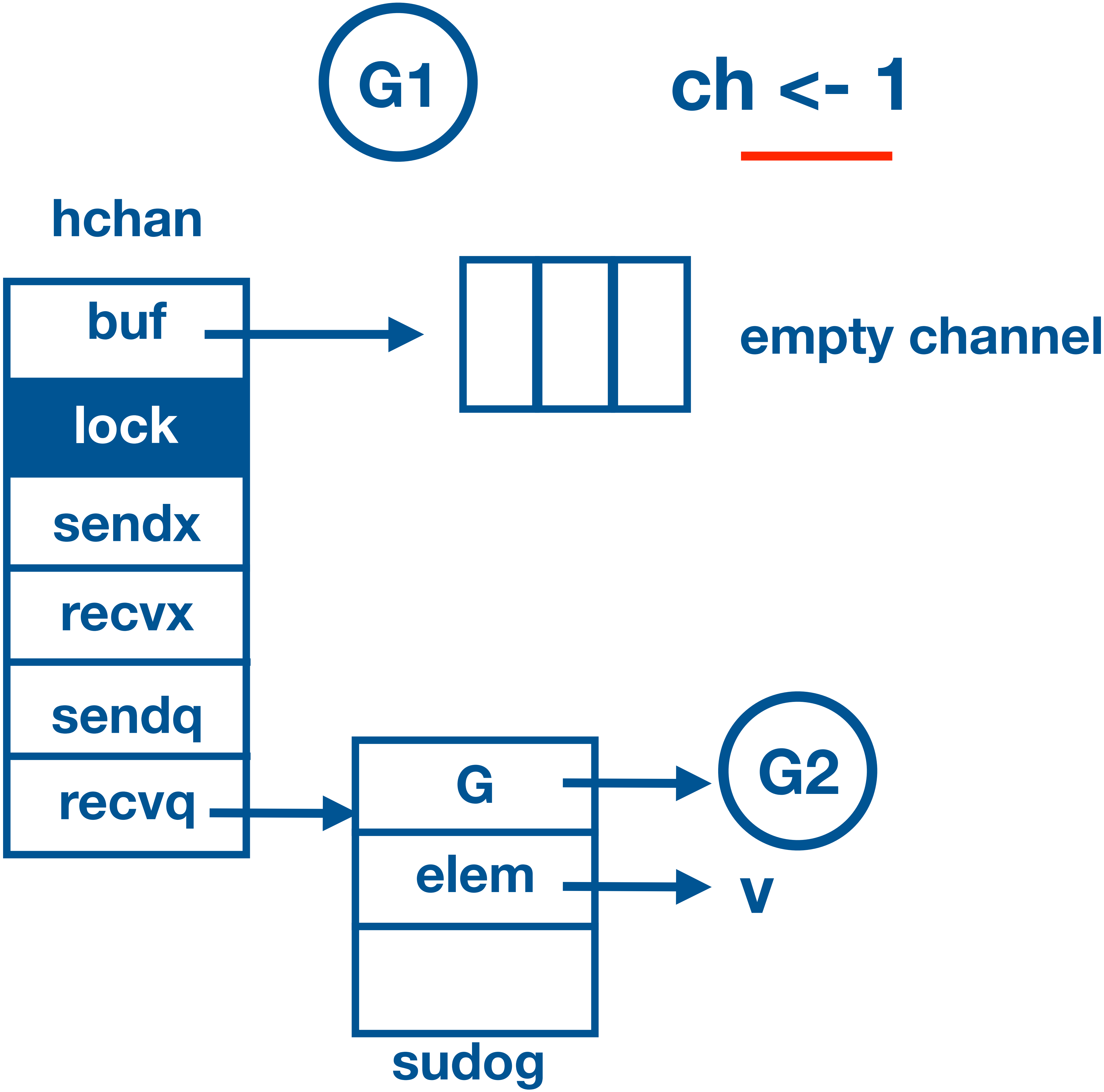



```
v := <-ch
```

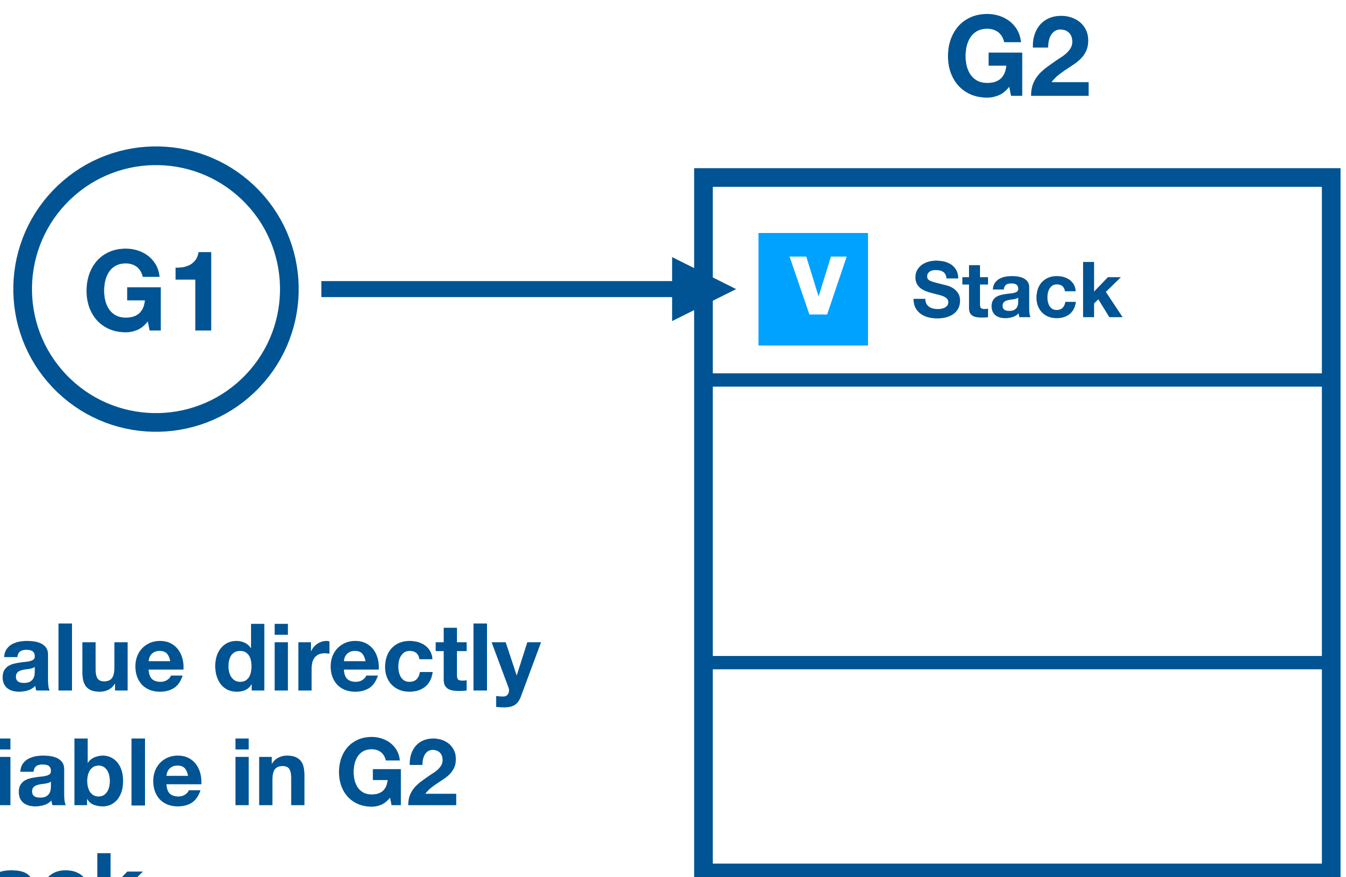


G2 calls gopark()

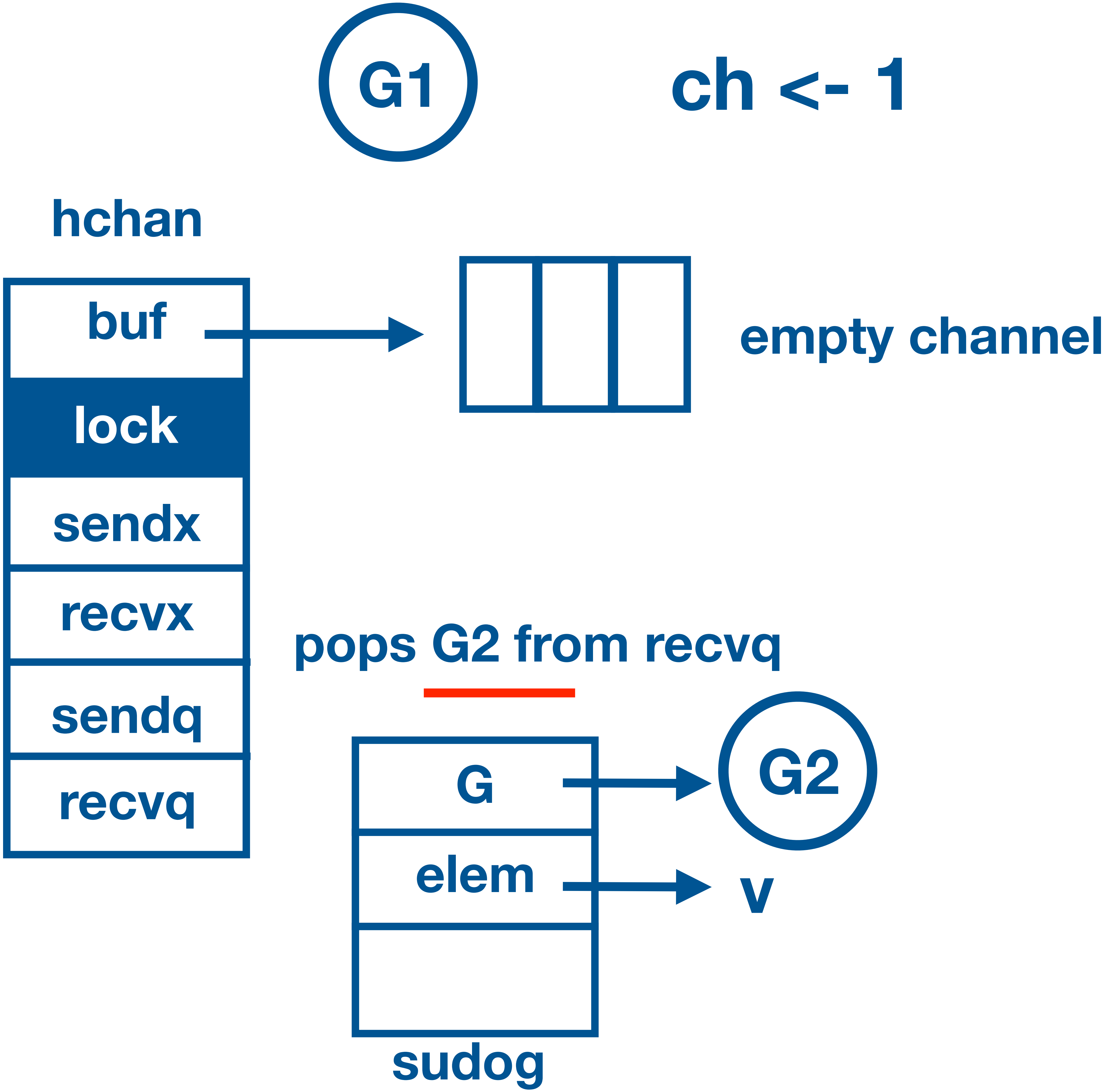

Core
M1
G1
P1
Local Run Queue
v := <-ch
Channel ch
recvQ
sendQ
G2

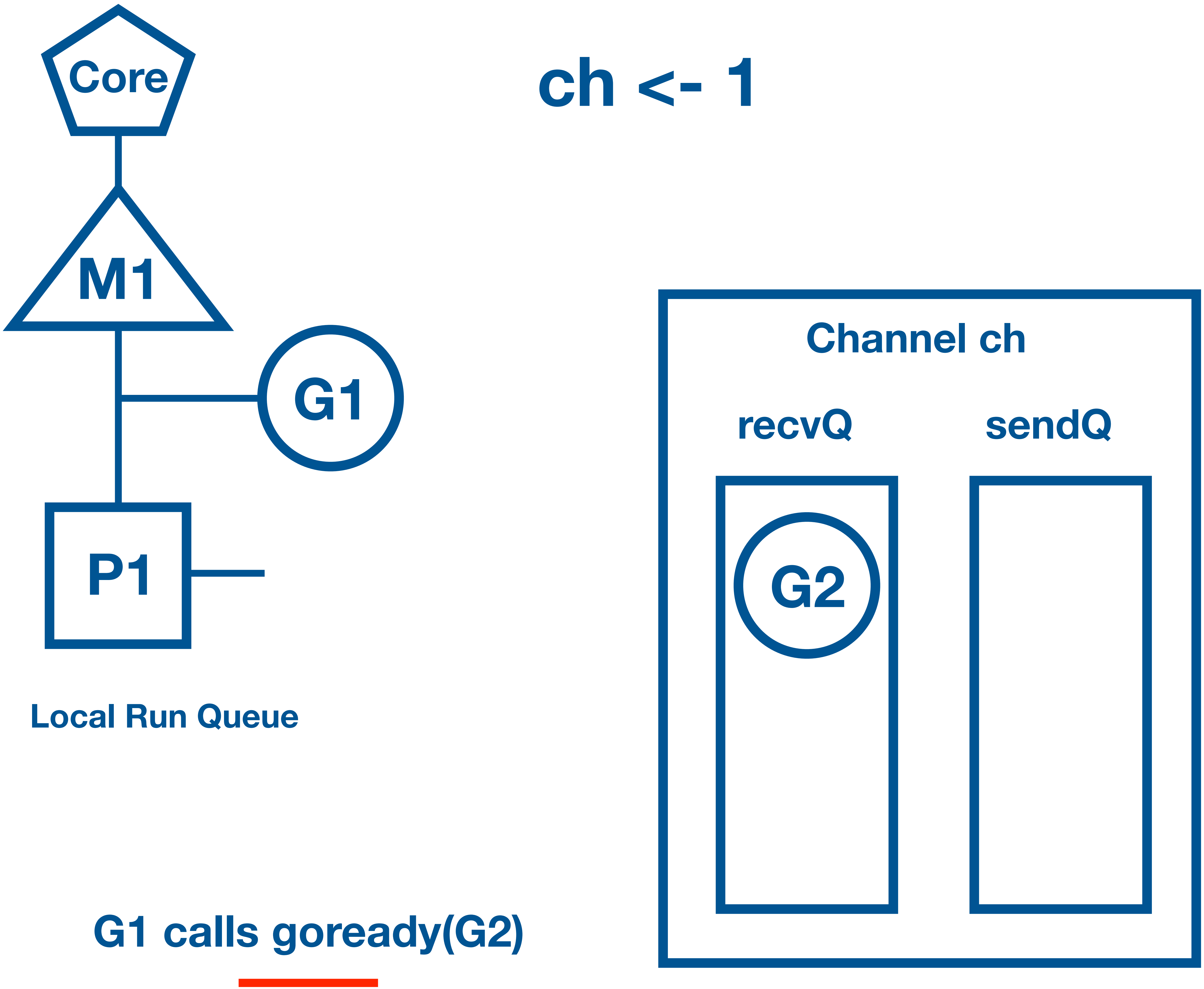

G1
ch <- 1
hchan
buf
lock
sendx
recvx
sendq
recvq
empty channel
G
elem
sudog
G2
v

G1 ch <- 1

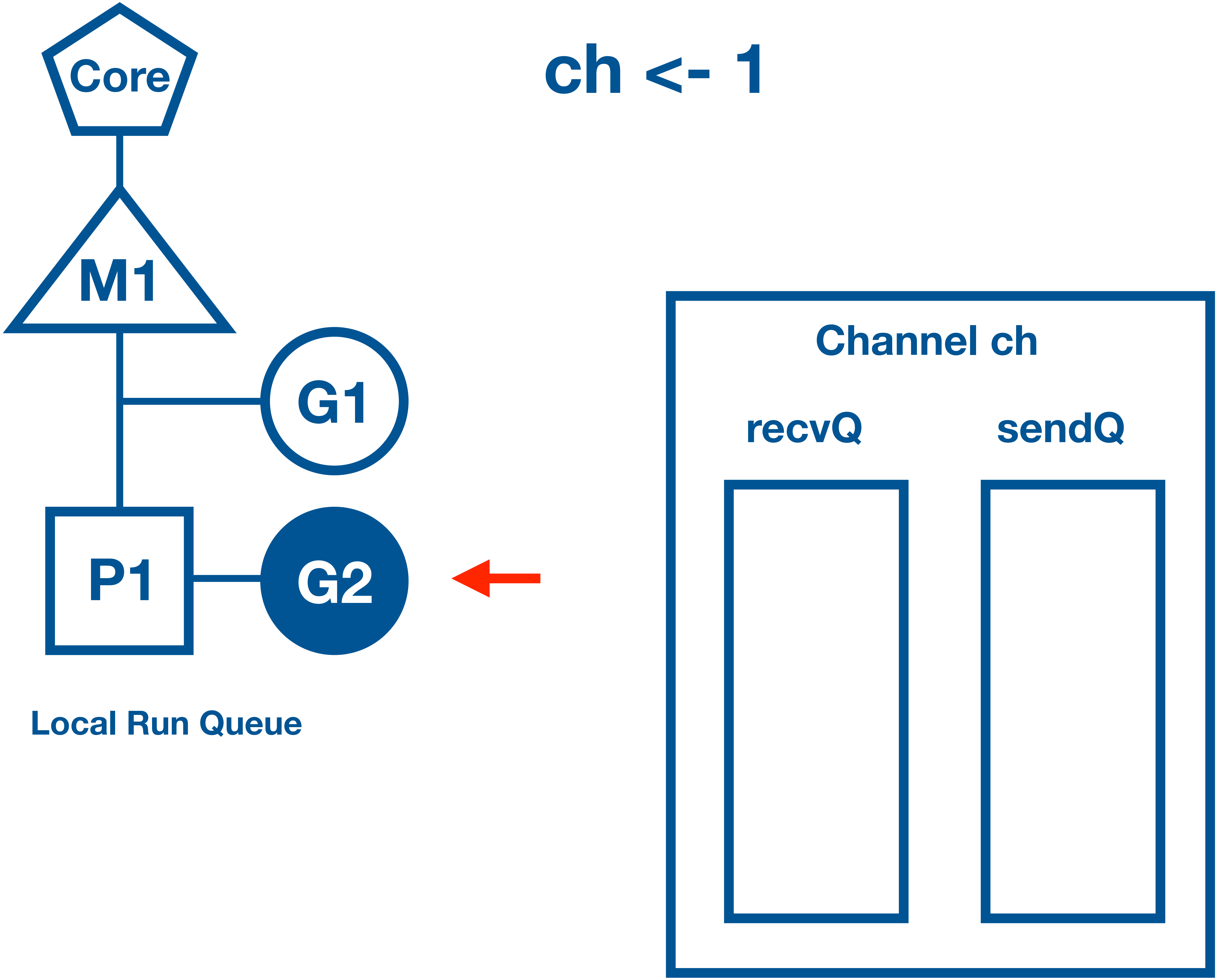


G1 writes value directly into v variable in G2 stack


G1
ch <- 1
hchan
buf
lock
sendx
recvx
sendq
recvq
empty channel
pops G2 from recvq
G
elem
G2
v
sudog


Core
M1
G1
P1
Local Run Queue
ch <- 1
Channel ch
recvQ
sendQ
G2
G1 calls goready(G2)

ch <- 1



Summary

- When goroutine calls receive on empty buffer.
- Goroutine is blocked, it is parked into recvq.
- elem field of the sudog structure holds the reference to the stack variable of receiver goroutine.
- When sender comes along, Sender finds the goroutine in recvq.
- Sender copies the data, into the stack variable, on the receiver goroutine directly..
- Pops the goroutine in recvq, and puts it into runnable state.

Send and Receive
Unbuffered channels

Send on unbuffered channel

- When sender goroutine wants to send values.
- if there is corresponding receiver waiting in recvq.
- Sender will write the value directly into receiver goroutine stack variable.
- Sender goroutine puts the receiver goroutine back to runnable state.

- If there is no receiver goroutine in recvq.
- Sender gets parked into sendq
- Data is saved in elem field in sudog struct.
- Receiver comes and copies the data.
- Puts the sender to runnable state again.

Receive on unbuffered channel

- Receiver goroutine wants to receive value.
- If it find a goroutine in waiting in sendq
- Receiver copies the value in elem field to its variable.
- Puts the sender goroutine to runnable state.

- If there was no sender goroutine in sendq.
- Receiver gets parked into recvq
- Reference to variable is saved in elem field in sudog struct.
- Sender comes and copies the data directly to receiver stack variable.
- Puts the receiver to runnable state.

Summary

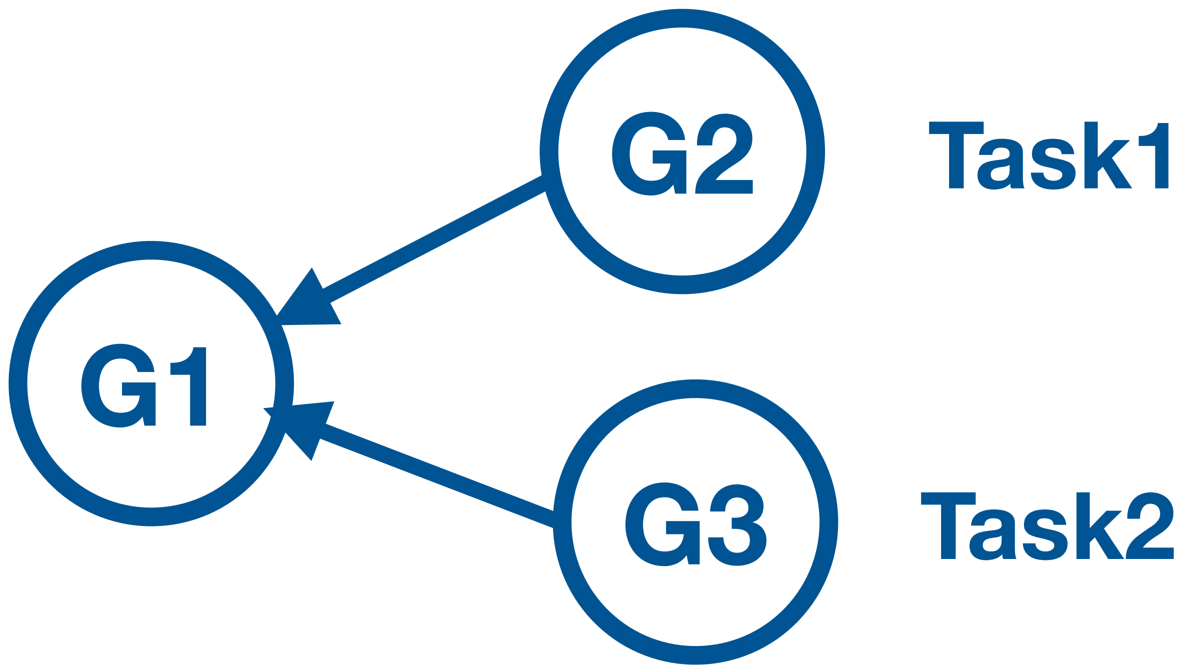
How channels work?

- hchan struct represents channel.
- It contains circular ring buffer and mutex lock.
- Goroutines that gets blocked on send or recv are parked in sendq or recvq.
- Go scheduler moves the blocked goroutines, out of OS thread.
- Once channel operation is complete, goroutine is moved back to local run queue.

select

Scenario

- **G1 wants to receive result of computation from G2 and G3**



- In what order are we going to receive results?

g1 <- g2		**g1 <- g3**
	OR	
g1 <- g3		**g1 <- g2**

- What if G3 was much faster than G2 in one instance, and G2 is faster than G3 in another?

Can we do operation on channel which ever is ready and don't worry about the order?

Select

- select statement is like a switch
- Each cases specifies communication
- All channel operation are considered simultaneously.

```
select {
case <-ch1:
  // block of statements
case <-ch2:
  // block of statements
case ch3 <- struct{}{}:
  // block of statements
}
```

Select

- select waits until some case is ready to proceed.
- when one the channels is ready, that operation will proceed.

```go
select {
case <-ch1:
  // block of statements
case <-ch2:
  // block of statements
case ch3 <- struct{}{}:
  // block of statements
}
```

Select

- Select is also very helpful in implementing,
    - Timeouts
    - Non-blocking communication

Timeout waiting on channel

```
select {
case v := <-ch:
    fmt.Println(v)
case <-time.After(3 * time.Second):
    fmt.Println("timeout")
}
```

- select waits until there is event on channel ch or until timeout is reached.

- The time.After function takes in a time.Duration argument and returns a channel that will send the current time after the duration you provide it.

Non-blocking communication

```go
select {
case m := <-ch:
    fmt.Println("received message", m)
default:
    fmt.Println("no message received")
}
```

- send or receive on a channel, but avoid blocking if the channel is not ready.

- Default allows you to exit a select block without blocking.

Empty Select

- Empty select statement will **block forever.**

Select {}

- Select on nil channel will block forever.

```
var ch chan string
select {
	case v := <-ch:
	case  ch <- v:
}
```

Summary

- Select is like switch statement with each case statement specifing channel operation.

- Select will block until any of the case statement is ready.

- With select we can implement timeout and non-blocking communicaton.

- Select on nil channel will block forever.

sync Package

sync.Mutex

When to use channels and when to use mutex

Channels:

- Passing copy of data.
- Distributing units of work.
- Communicating asynchronous results

Mutex:

- Caches
- State

Mutex

- Used for protect shared resources.

Photo by Elaine Casap on Unsplash

- sync.Mutex - Provide exclusive access to a shared resource.

```
mu.Lock()

balance += amount

mu.Unlock()
```

```
mu.Lock()

defer mu.Unlock()

balance -= amount
```

- sync.RWMutex - Allows multiple readers. Writers get exclusive lock.

```
mu.Lock()

balance += amount

mu.Unlock()
```

```
mu.RLock()

defer mu.RUnlock()

return balance
```

Summary

- Mutex is used guards access to shared resources.

- It is developers convention to call Lock() to access shared memory and call Unlock() when done.

- The critical section represents the bottleneck between the goroutines.

Summary

- **When do you use mutex, channels and waitgroup?**

Channels:

- Passing copy of data.
- Distributing units of work.
- Communicating asynchronous results

Mutex:

- Caches
- State

WaitGroup

- Cleanup: wait until all goroutines terminate cleanly.
- Barrier point where can collect computational result from multiple goroutines.

sync.Atomic

sync.Atomic

- Low level atomic operations on memory.
- Lockless operation.
- Used for atomic operations on counters.

```
atomic.AddUint64(&ops, 1)

value := atomic.LoadUint64(&ops)
```

sync.Cond

sync.Cond

- Condition Variable is one of the synchronization mechanisms.

- A condition variable is basically a container of Goroutines that are waiting for a certain condition.

How to make a goroutine wait till some event(condition) occur?

One Way - Wait in a loop for the condition

```go
var sharedRsc = make(map[string]string)
go func() {
	defer wg.Done()
	mu.Lock()
	for len(sharedRsc) == 0 {
		mu.Unlock()
		time.Sleep(100 * time.Millisecond)
		mu.Lock()
	}
	// Do processing..
	fmt.Println(sharedRsc["rsc"])
	mu.Unlock()
}()
```

- We need some way to make goroutine suspend while waiting.
- We need some way to signal the suspended goroutine when particular event has occured.

Channels?

- We can use channels to block a goroutine on receive.

- Sender goroutine to indicate occurence of event.

- What if there are multiple goroutines waiting on multiple conditions/event?

sync.Cond

- Conditional Variable are type

```go
var c *sync.Cond
```

- We use constructor method sync.NewCond() to create a conditional variable, it takes sync.Locker interface as input, which is usually sync.Mutex.

```go
m := sync.Mutex{}
c := sync.NewCond(&m)
```

sync.Cond

- It has 3 methods.

```
c.Wait()
c.Signal()
c.Broadcast()
```

c.Wait()

```
c.L.Lock()
for !condition() {
    c.Wait()
}
... make use of condition ...
c.L.Unlock()
```

- suspends execution of the calling goroutine.
- automatically unlocks c.L
- Wait cannot return unless awoken by Broadcast or Signal.
- Wait locks c.L before returning.
- Because c.L is not locked when Wait first resumes, the caller typically cannot assume that the condition is true when Wait returns. Instead, the caller should Wait in a loop

c.Signal()

```
func (c *Cond) Signal()
```

- Signal wakes one goroutine waiting on c, if there is any.
- Signal finds goroutine that has been waiting the longest and notifies that.
- It is allowed but not required for the caller to hold c.L during the call.

c.Broadcast()

```go
func (c *Cond) Broadcast()
```

- Broadcast wakes all goroutines waiting on c.
- It is allowed but not required for the caller to hold c.L during the call.

G2

```go
mu := sync.Mutex{}
c := sync.NewCond(&mu)

var sharedRsc = make(map[string]string)

go func() {
    defer wg.Done()
    c.L.Lock()
    for len(sharedRsc) == 0 {
        c.Wait()
    }
    // Do processing..
    fmt.Println(sharedRsc["rsc"])
    c.L.Unlock()
}()
```

G1

```go
go func() {
    defer wg.Done()
    c.L.Lock()
    sharedRsc["rsc"] = "foo"
    c.Signal()
    c.L.Unlock()
}()
```

G2

```go
var sharedRsc = make(map[string]string)
go func() {
  defer wg.Done()
  c.L.Lock()
  for len(sharedRsc) < 1 {
    c.Wait()
  }
  // Do processing
  fmt.Println(sharedRsc["rsc1"])
  c.L.Unlock()
}()
```

G3

```go
go func() {
  defer wg.Done()
  c.L.Lock()
  for len(sharedRsc) < 2 {
    c.Wait()
  }
  // Do processing
  fmt.Println(sharedRsc["rsc2"])
  c.L.Unlock()
}()
```

G1

```go
go func() {
  defer wg.Done()
  c.L.Lock()
  sharedRsc["rsc1"] = "foo"
  sharedRsc["rsc2"] = "bar"
  c.Broadcast()
  c.L.Unlock()
}()
```

Summary

- Conditional Variable is used to synchronise execution of goroutines.
- Wait suspends the execution of goroutine.
- Signal wakes one goroutine waiting on c.
- Broadcast wakes all goroutines waiting on c.

sync.Once

sync.Once

- Run one-time initialization functions

once.Do(*funcValue*)

- sync.Once ensure that only one call to Do ever calls the function passed in—even on different goroutines.

Photo by Annie Spratt on Unsplash

sync.Pool

sync.Pool

- create and make available pool of things for use.

Photo by Debby Hudson on Unsplash

```
b := bufPool.Get().(*bytes.Buffer)

bufPool.Put(b)
```

Go Race Detector

Go Race Detector

- Go provides race detector tool for finding race conditions in Go code.

```
$ go test -race mypkg    // test the package
$ go run -race mysrc.go  // compile and run the program
$ go build -race mycmd   // build the command
$ go install -race mypkg // install the package
```

Go Race Detector

- Binary needs to be race enabled.
- When racy behaviour is detected a warning is printed.
- Race enabled binary will 10 times slower and consume 10 times more memory.
- Integration tests and load tests are good candidates to test with binary with race enabled.

Web Crawler

- Build web crawler using Go’s concurrency feature.
- Fetch URLs in parallel to speed up the web crawl.

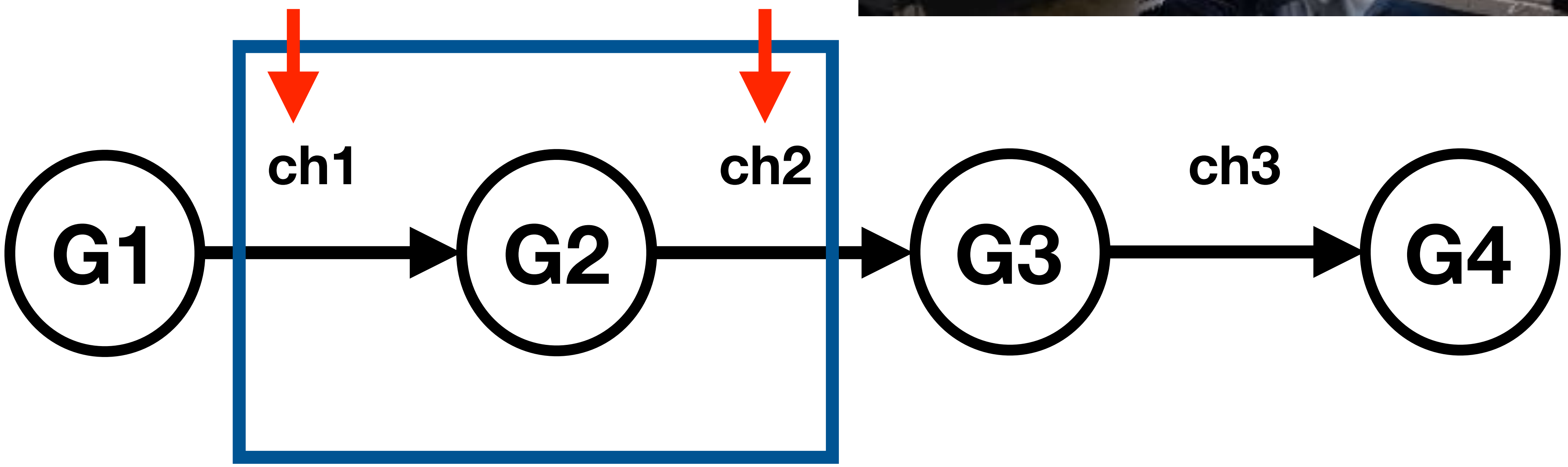
Concurrency Patterns

Pipelines

Pipeline

- Process streams, or batches of data.

Unsplash: Remy Gieling



- **Stage - take data in, perform an operation on it, and send the data out.**

Stages

- Separate the concerns of each stage.
- Process individual stage concurrently.
- A stage could consume and return the same type.

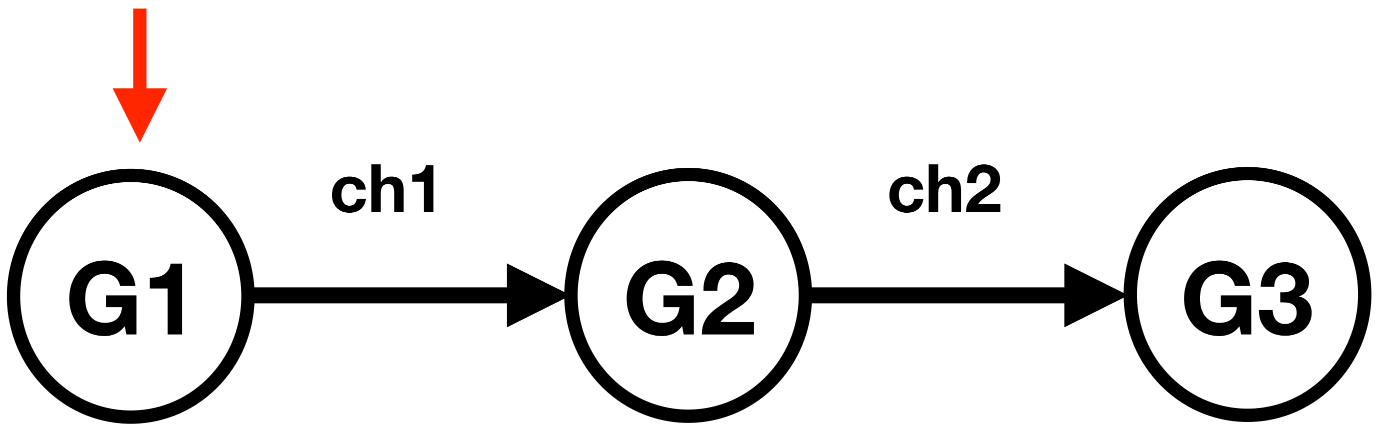
```
func square(in <-chan int) <-chan int {
```

- This enables composability of pipeline.

```
square(square(generator(2, 3)))
```

Image Processing Pipeline

- Input: List of images.
- Output: Thumbnail images



- **G1**: Generate a list of images to process.
- **G2**: Generate thumbnail images.
- **G3**: Store thumbnail image to disk or Transfer to storage bucket in cloud.

Summary

What are pipelines used for ?

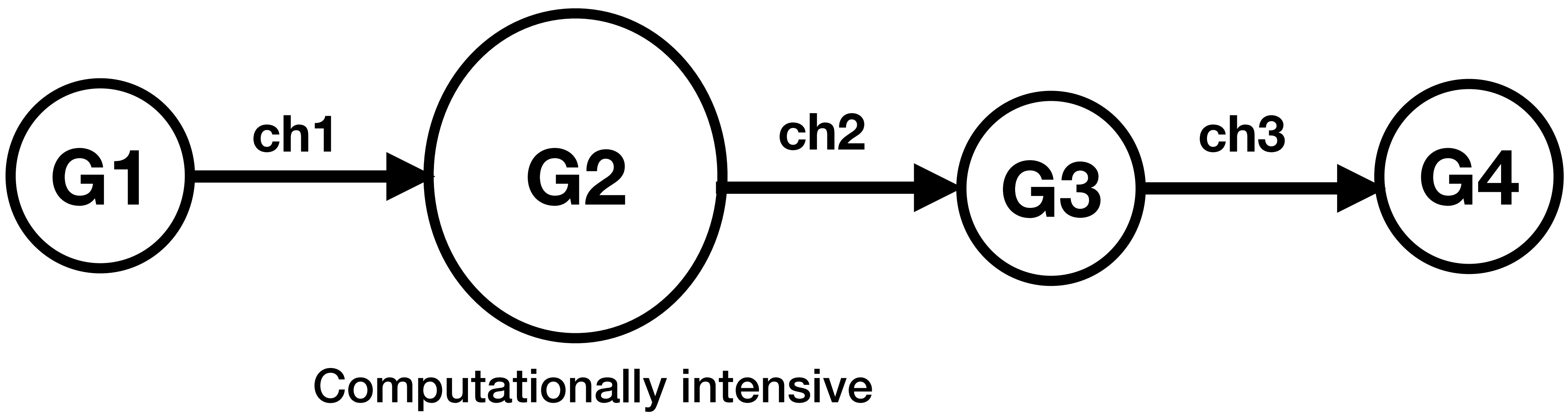
- Pipelines are used to process Streams or Batches of data.

- Pipelines enables us to make an efficient use of I/O and multiple CPUs cores.

- Pipeline is a series of stages, connected by channels.

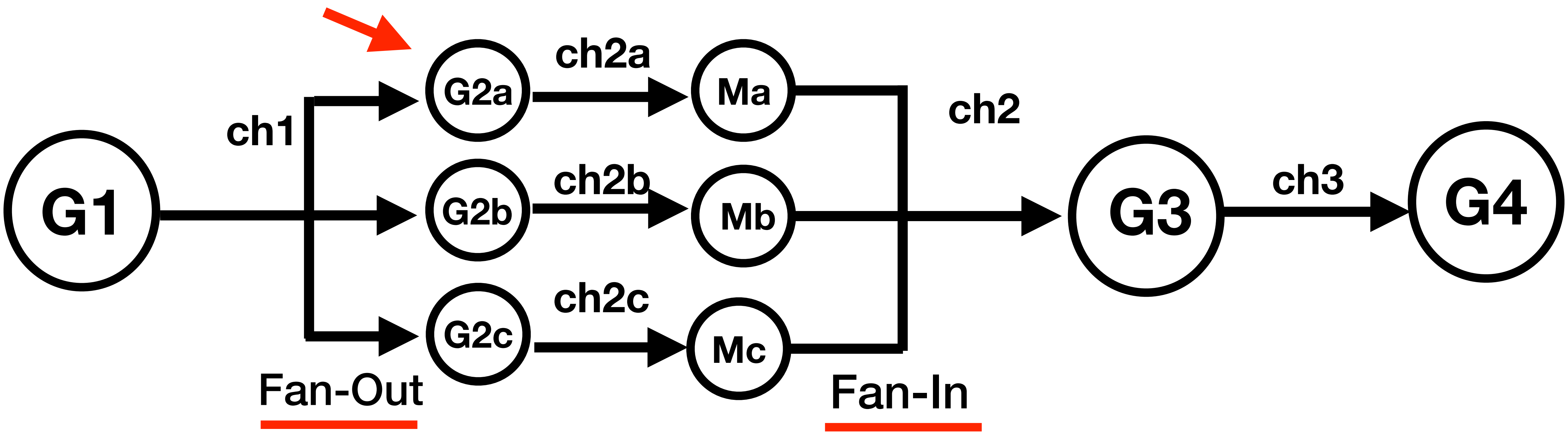
- Each stage is a represented by a goroutine.

Fan-out, Fan-in


G1
ch1
G2
ch2
G3
ch3
G4
Computationally intensive

Can we break computationally intensive stage into multiple goroutines and run them in parallel to speed it up?

Fan-out, Fan-in



Summary

What is fan-out?

- Multiple goroutines are started to read data from the single channel.

- Distribute work amongst a group of workers goroutines to parallelize the CPU usage and the I/O usage.

- Helps computational intensive stage to run faster.

Summary

What is Fan-in?

- Process of combining multiple results into one channel.

- We create Merge goroutines, to read data from multiple input channels and send the data to a single output channel.

Pattern in our Pipelines

- Upstream stages close their outbound channels when all the send operations are done.

```go
func generator(nums ...int) <-chan int {
  out := make(chan int)
  go func() {
    for _, n := range nums {
      out <- n
    }
    close(out)
  }()
  return out
}
```

Pattern in our Pipelines

- Downstream stages keep receiving values from inbound channel until the channel is closed.

```
func square(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        for n := range in {
            out <- n * n
        }
        close(out)
    }()
    return out
}
```

- All goroutines exit once all values have been successfully sent downstream.

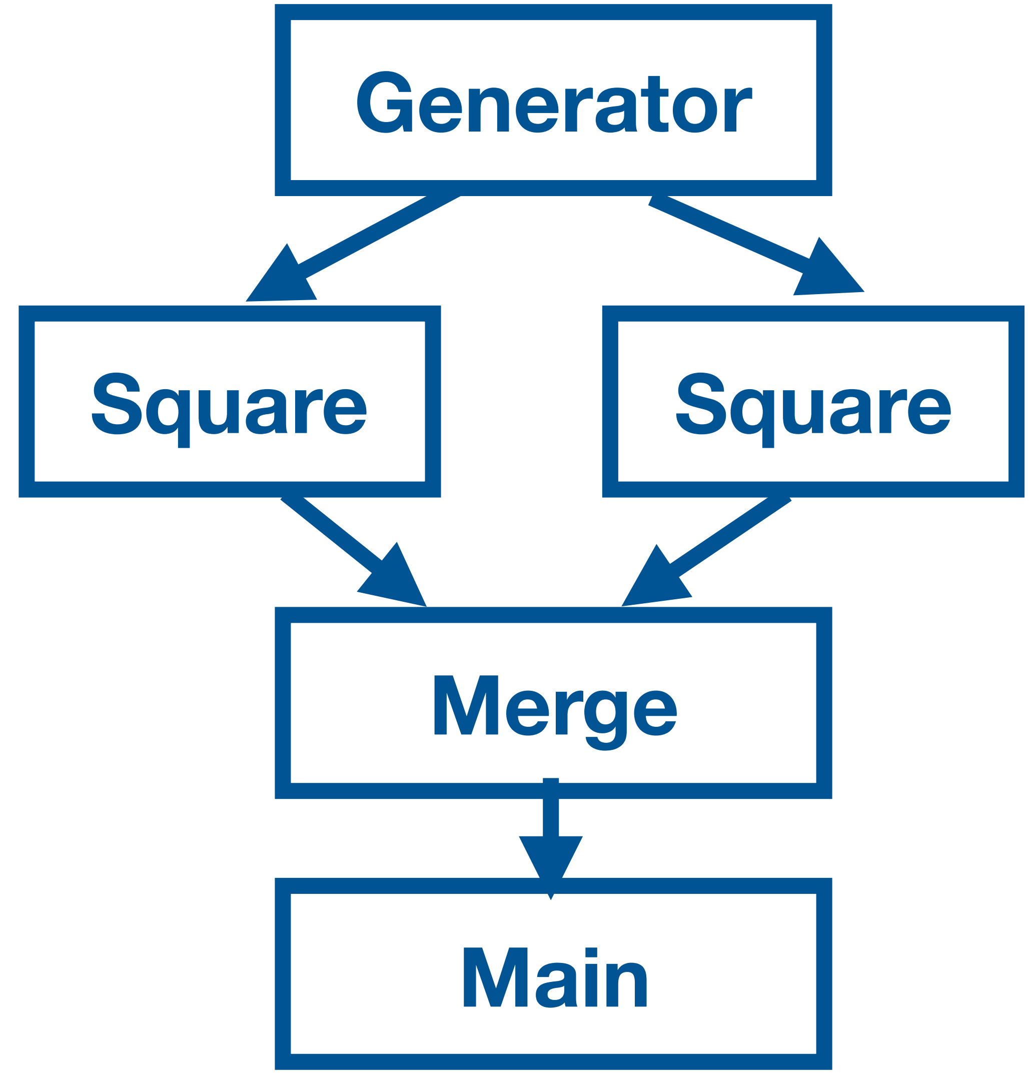
```go
func merge(cs ...<-chan int) <-chan int {

  output := func(c <-chan int) {
    for n := range c {
      out <- n
    }
    wg.Done()
  }

  wg.Add(len(cs))
  for _, c := range cs {
    go output(c)
  }

  go func() {
    wg.Wait()
    close(out)
  }()
```

```go
func main() {
  in := generator(2, 3)

  c1 := square(in)
  c2 := square(in)

  for n := range merge(c1, c2) {
    fmt.Println(n)
  }
}
```

Real Pipelines

- Real pipelines - Receiver Stages may only need a subset of values to make progress.
- A stage can exit early because an inbound value represents an error in an earlier stage.
- Receiver should not have to wait for the remaining values to arrive
- we want earlier stages to stop producing values that later stages don't need.

```go
func main() {
    in := generator(2, 3)

    c1 := square(in)
    c2 := square(in)

    out := merge(c1, c2)

    fmt.Println(<-out)
}
```



- Main grouting just receives one value.
- Abandones the inbound channel from merge.
- merge goroutines will be blocked on channel send operation.
- Square and generator groutings will also be blocked on send.
- **This leads to GOROUTINE LEAK.**

How can we signal to goroutine to abandon what they are doing and terminate?

Cancellation of goroutines

- Pass a read-only ‘done’ channel to goroutine

- Close the channel, to send broadcast signal to all goroutine.

- On receiving the signal on done channel, Goroutines needs to abandon their work and terminate.

- We use ‘select’ to make send/ receive operation on channel pre-emptible.

```
select {
case out <- n:
case <-done:
    return
}
```

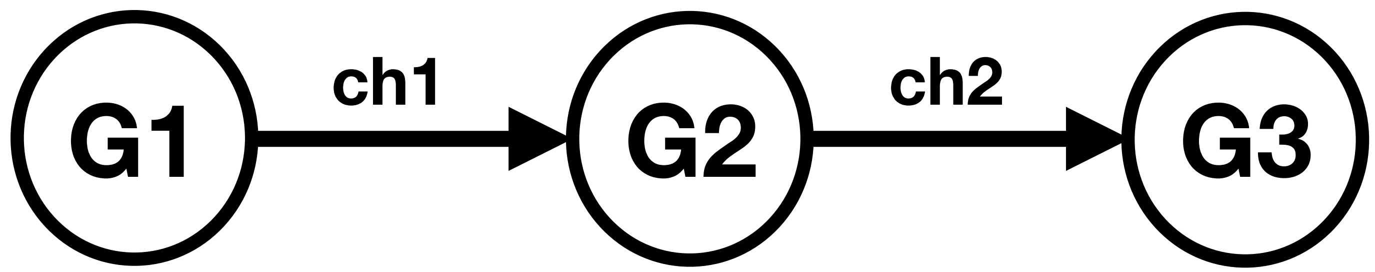
Guidelines for Pipeline Construction

- stages close their outbound channels when all the send operations are done.

- stages keep receiving values from inbound channels until those channels are closed **or the senders are unblocked.**

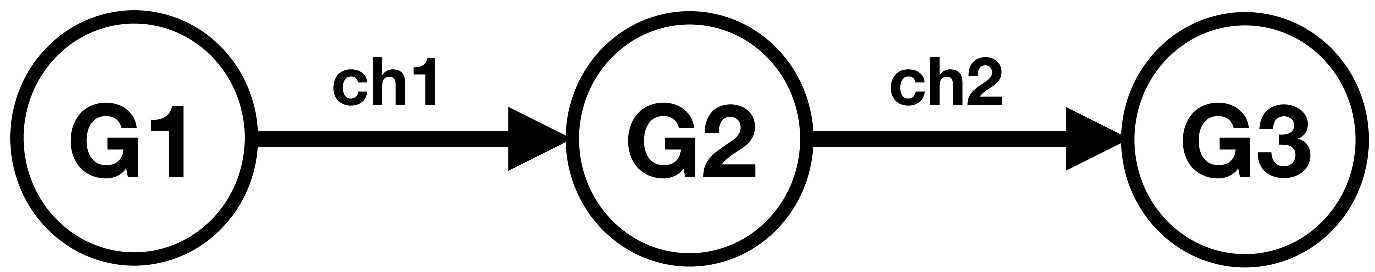
- Pipelines unblock senders by explicitly signalling senders when the receiver may abandon the channel.

Cancellation



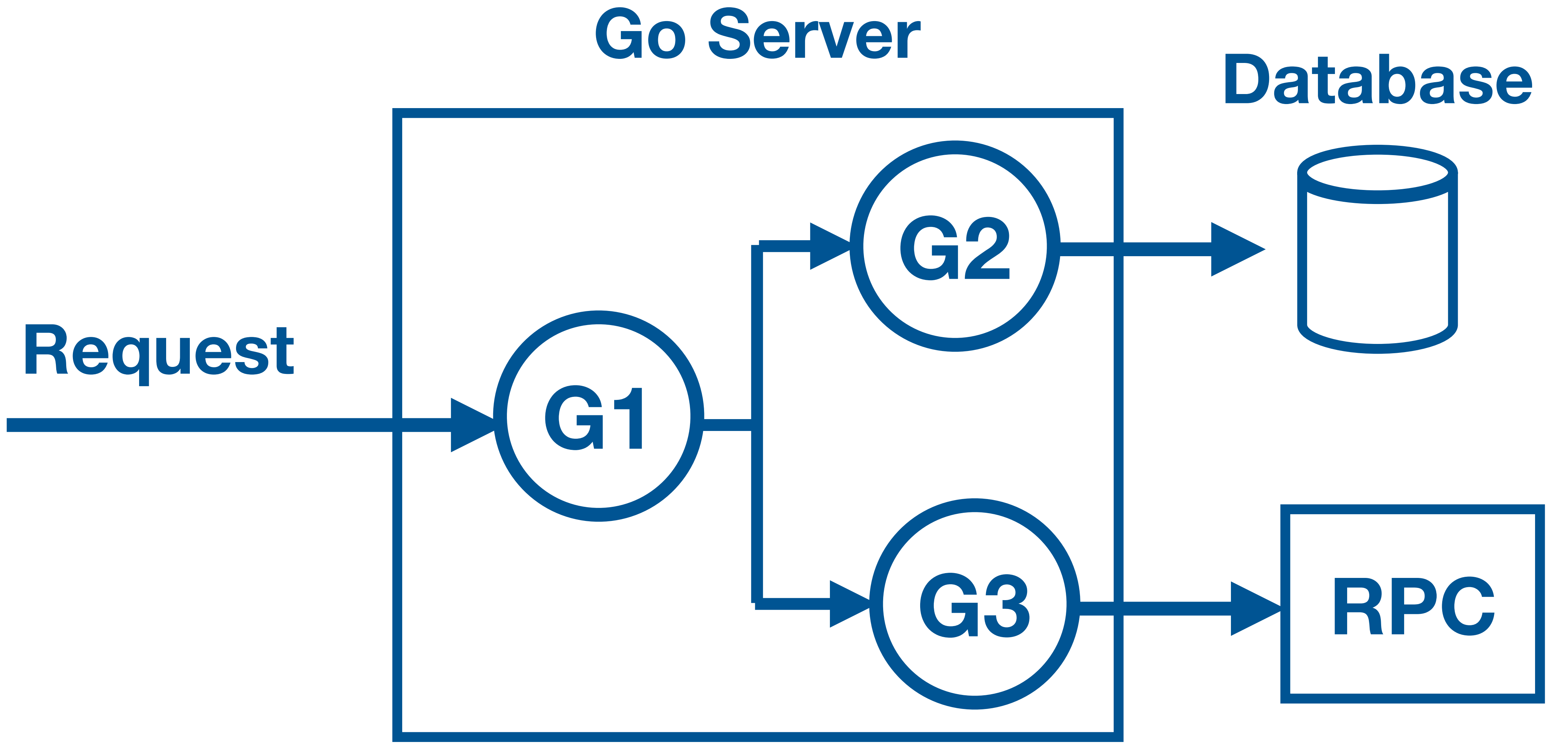
- **Channels are used to receive and emit values.**
- G1 generate batch of data into channel ch1.
- A stage consumes and returns the same type.
- Each stage takes
  - Common done channel
  - Input channel
  - Returns output channel

Properties of pipeline stage



- **Channels are used to receive and emit values.**
- G1 generate batch of data into channel ch1.
- A stage consumes and returns the same type.
- Each stage takes
    - Common done channel
    - Input channel
    - Returns output channel

Context Package


Go Server
Database
Request
G1
G2
G3
RPC

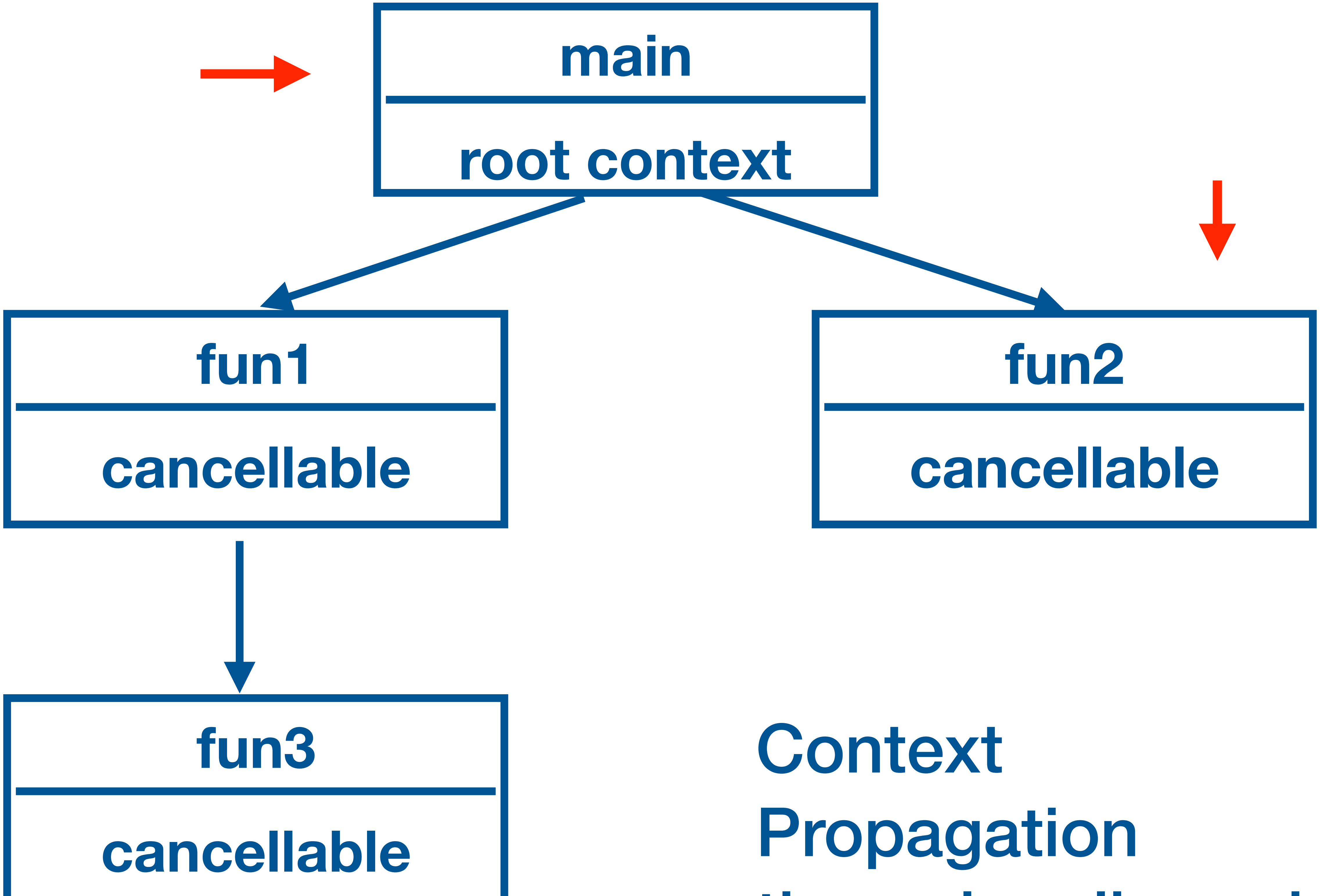
How can we propagate

- request-scoped data?
- cancellation signal?

Context Package

Context Package serves two primary purpose:

- Provides API’s for cancelling branches of call-graph.
- Provides a data-bag for transporting request-scoped data through call-graph.



Context
Propagation
through call graph

```go
// A Context carries a deadline, cancelation signal, and request-scoped values
// across API boundaries. Its methods are safe for simultaneous use by multiple
// goroutines.
type Context interface {
    // Done returns a channel that is closed when this Context is canceled
    // or times out.
    Done() <-chan struct{}

    // Err indicates why this context was canceled, after the Done channel
    // is closed.
    Err() error

    // Deadline returns the time when this Context will be canceled, if any.
    Deadline() (deadline time.Time, ok bool)

    // Value returns the value associated with key or nil if none.
    Value(key interface{}) interface{}
}
```

- A Context is safe for simultaneous use by multiple goroutines.
- Single Context can be passed to any number of goroutines.
- Cancelling the context signals all the goroutines to abandon their work and terminate.

Context Package Functions

Context package provides functions to create new context

- context.Background()
- context.TODO()

Background()

```go
func main() {
    ctx := context.Background()
```

- Background returns a empty context.
- Root of any context tree.
- It is never canceled, has no value and has no deadline.
- Typically used by main function.
- Acts as a top level context for incoming request.

TODO()

```
func fun() {
    ctx := context.TODO()
```

- TODO() returns an empty Context.
- TODO's intended purpose is to serve as a placeholder.

Summary

What is Context Package used for?

- Context package can be used to send,
    - Request-scoped values
    - Cancellation signals

- Across API boundaries to all goroutines involved in handling a request.

Summary

- Context package provides functions to create context.
- **context.Background()** - returns an empty context, it is the root of any Context tree.
- **context.TODO()** - returns an empty Context, intended purpose is to serve as a placeholder.

Context Package for cancellation

Context Package… cancellation

- Context is immutable.
- Context package provides function to add new behaviour.
- To add cancellation behaviour we have function like,
  - context.WithCancel()
  - context.WithTimeout()
  - context.WithDeadline()
- The derived context is passed to child goroutines to facilitate their cancellation.

WithCancel()

```
// Create a context that is cancellable.
ctx, cancel := context.WithCancel(context.Background())
defer cancel()
```

- WithCancel returns a copy of parent with a new Done channel.
- cancel() can be used to close context's done channel.
- Closing the done channel indicates to an operation to abandon its work and return.
- Canceling the context releases the resources associated with it.

cancel()

- cancel() does not wait for the work to stop.
- cancel() may be called by multiple goroutines simultaneously.
- After the first call, subsequent calls to a cancel() do nothing.

Workflow

```go
// Create a context that is cancellable.
ctx, cancel := context.WithCancel(context.Background())
ch := generator(ctx)
...
if n == 5 {
    cancel()
}
```

Parent Goroutine

```go
for {
    select {
    case <-ctx.Done():
        return ctx.Err()
    case dst <- n:
        n++
    }
}
```

Child Goroutine

WithDeadline()

```
deadline := time.Now().Add(5 * time.Millisecond)
ctx, cancel := context.WithDeadline(context.Background(), deadline)
defer cancel()
```

- WithDeadline() takes parent context and clock time as input.
- WithDeadline returns a new Context that closes its done channel when the machine's clock advances past the given deadline

go/src/context/context.go

```go
func WithDeadline(parent Context, d time.Time) (Context, CancelFunc) {
    ...
   c := &timerCtx{
     cancelCtx: newCancelCtx(parent),
     deadline:  d,
   }

   ...

   c.timer = time.AfterFunc(dur, func() {
     c.cancel(true, DeadlineExceeded)
   })

   ...
```

```go
deadline, ok := ctx.Deadline()
if ok {
    if deadline.Sub(time.Now().Add(10*time.Millisecond)) <= 0 {
        return context.DeadlineExceeded
    }
}
```

...

```go
for {
    select {
    case <-ctx.Done():
        return ctx.Err()
    case dst <- n:
        n++
    }
}
```

Child Goroutine

WithTimeout()

```
duration := 5 * time.Millisecond
ctx, cancel := context.WithTimeout(context.Background(), duration)
defer cancel()
```

- WithTimeout() takes parent context and time duration as input.

- WithTimeout() returns a new Context that closes its done channel after the given timeout duration.

- WithTimout() is useful for setting a deadline on the requests to backend servers.

go/src/context/context.go

```go
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc) {
	return WithDeadline(parent, time.Now().Add(timeout))
}
```

- WithTimout() is a wrapper over WithDeadline().

Difference in using WithTimeout and WithDeadline

- WithTimout() - timer countdown begins from the moment the context is created

- WithDeadline() - Set explicit time when timer will expire.

Summary

How context package can be used for cancellation of an operation?

- Context package provides functions to derive new context values from existing ones to add cancellation behaviour.

- **context.WithCancel()** - is used to create a cancellation context.

- **cancel()** is used to close the done channel.

- On receiving the close signal, goroutine is suppose to abandon its operation and return.

Summary

- **context.WithDeadline()** - is used to set deadline to an operation.

- context.WithDeadline() - creates a new context, whose done channel gets closed when machine's clock advances past the given deadline.

- **ctx.Deadline()** can used to know if a deadline is associated with the context.

Summary

- **context.WithTimeout()** - used to set timeout to an operation.

- context.WithTimeout() - creates a new context, whose done channel is closed after the given timeout duration.

Context Package as Data bag

Context Package as Data bag

- Context Package can used to transport request-scoped data down the call graph.

- context.WithValue() provides a way to associate request-scoped values with a Context,

WithValue()

```go
type userIDType string
ctx := context.WithValue(context.Background(),
                         userIDType("userIDKey"), "jane")
```

Parent Goroutine

```go
userid := ctx.Value(userIDType("userIDKey")).(string)
```

Child Goroutine

Summary

- Context package can be used as data bag to carry request-scoped data.

- **context.WithValue()** - used to associate request-scoped data with a context.

- **ctx.Value()** - is used to extract the value given a key from the context.

Go’s Idioms for Context Package

Incoming requests to a server should create a Context

- Create context early in processing task or request.
- Create a top level context

```
func main() {
    ctx := context.Background()
```

- http.Request value already contains a Context.

```
func handleFunc(w http.ResponseWriter, req *http.Request) {
        ctx, cancel = context.WithCancel(req.Context())
```

Outgoing calls to servers should accept a Context

- Higher level calls need to tell lower level calls how long they are willing to wait.

```
// Create a context with a timeout of 100 milliseconds.
ctx, cancel := context.WithTimeout(req.Context(), 100*time.
Millisecond)
defer cancel()

// Bind the new context into the request.
req = req.WithContext(ctx)

// Do will handle the context level timeout.
resp, err := http.DefaultClient.Do(req)
```

- http.DefaultClient.Do() method to respect cancellation signal on timer expiry and return with error message.

Pass a Context to function performing I/O

- Any function that is performing I/O should accept a Context value as it's first parameter and respect any timeout or deadline configured by the caller.

- Any API's that takes a Context, the idiom is to have the first parameter accept the Context value.

```
tcpsock_posix.go
dialTCP(ctx context.Context, laddr, raddr *TCPAddr) (*TCPConn, error)
listenTCP(ctx context.Context, laddr *TCPAddr) (*TCPListener, error)
dialUDP(ctx context.Context, laddr, raddr *UDPAddr) (*UDPConn, error)
listenUDP(ctx context.Context, laddr *UDPAddr) (*UDPConn, error)

sql.go
QueryContext(ctx context.Context, query string, args ...interface{})
PrepareContext(ctx context.Context, query string)
```

Any change to a Context value creates a new Context value that is then propagated forward.

```
func main() {
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()
    ....
        go func(ctx context.Context) {
            ctx, cancel := context.WithTimeout(ctx, 1*time.Second)
            defer cancel()
            ....
                func(ctx context.Context) (string, error) {
                    select {
                        case <-ctx.Done():
                            return "", ctx.Err()
                        ....
                    }
                }()
        } (ctx)
}
```

When a Context is canceled, all Contexts derived from it are also canceled

- If a parent Context is cancelled, all children derived by that parent Context are cancelled as well.

Use TODO context if you are unsure about which Context to use

- If a function is not responsible for creating top level context.
- We need a temporary top-level Context until we figured out where the actual Context will come from.

Use context values only for request-scoped data

- Do not use the Context value to pass data into a function which becomes essential for its successful execution.

- A function should be able to execute its logic with an empty Context value.

Summary

- Incoming requests to a server should create a Context.
- Outgoing calls to servers should accept a Context
- Any function that is performing I/O should accept a Context value.
- Any change to a Context value creates a new Context value that is then propagated forward.
- If a parent Context is cancelled, all children derived from it are also cancelled.

Summary

- Use TODO context if you are unsure about which Context to use.

- Use context values only for request-scoped data, not for passing optional parameters to functions.

HTTP Server Timeouts with Context Package

HTTP Server Timeouts

- Setting timeouts in server is important to conserve system resources and to protect from DDOS attack.

- File descriptors are limited.

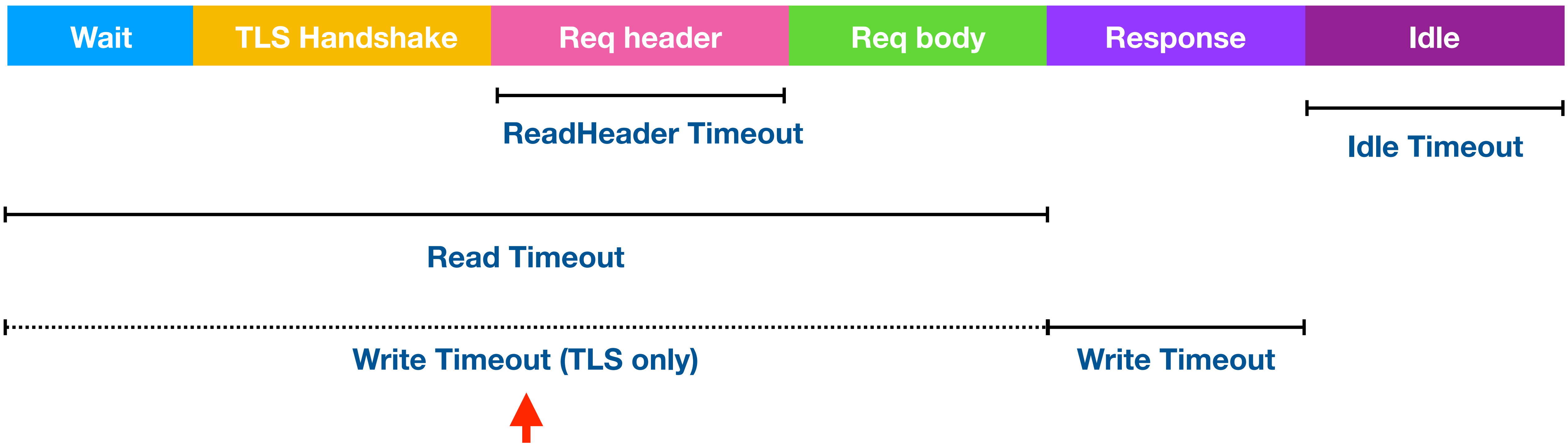
- Malicious user can open many client connections, consuming all file descriptors.

- Server will not able to accept any new connection.

http: Accept error: accept tcp [::]:80: accept: too many open files; retrying in 1s.

net/http Timeouts

- There are four main timeouts exposed in http.server
  - Read Timeout
  - Read Header Timeout
  - Write Timeout
  - Idle Timeout



- **Read Timeout:** covers the time from when the connection is accepted, to when the request body is fully read.

- **ReadHeader Timeout:** amount of time allowed to read request headers

- **Write Timeout:** covers the time from the end of the request header read to the end of the response write.

- **Idle Timeout:** maximum amount of time to wait for the next request when keep-alive is enabled.

Set Timeouts by explicitly using a Server

```go
srv := &http.Server{
    ReadTimeout:       1 * time.Second,
    ReadHeaderTimeout: 1 * time.Second,
    WriteTimeout:      1 * time.Second,
    IdleTimeout:       30 * time.Second,
    Handler:           serveMux,
}
```

- Set Connection timeouts when dealing with untrusted clients and networks.

- Protect Server from clients which are slow to read and write.

HTTP Handler Functions

- Connection timeouts apply at network connection level.
- HTTP Handler Functions are **unaware of these timeouts**, they run to completion, consuming resources.

How to efficiently timeout http handler function?

http.TimeoutHandler()

- net/http package provides TimeoutHandler()

```
srv := http.Server{
    Addr:         "localhost:8000",
    WriteTimeout: 2 * time.Second,
    Handler:      http.TimeoutHandler(http.HandlerFunc(slowHandler),
                                      1*time.Second,
                                      "Timeout!\n"),
}
```

- TimeoutHandler returns a Handler that runs input handler with the given time limit.
- If input handler runs for longer than its time limit, the handler sends the client a **503 Service Unavailable error** and HTML error message.

We need to propagate the timeout awareness down the call graph

Context Timeouts and Cancellation

- Use Context timeouts and cancellation to propagate the cancellation signal down the call graph.
- The Request type already has a context attached to it.

```
ctx := req.Context()
```

- Server cancels this context when,
    - Client closes the connection.
    - Timeout
    - ServeHTTP method returns.

go/src/net/http/server.go

```go
func (h *timeoutHandler) ServeHTTP(w ResponseWriter, r *Request) {
	ctx := h.testContext
	if ctx == nil {
		var cancelCtx context.CancelFunc
		ctx, cancelCtx = context.WithTimeout(r.Context(), h.dt)
		defer cancelCtx()
	}
	r = r.WithContext(ctx)
```

```go
srv := http.Server{
	Addr:         "localhost:8000",
	WriteTimeout: 2 * time.Second,
	Handler:      http.TimeoutHandler(http.HandlerFunc(slowHandler),
	                                  1*time.Second,
	                                  "Timeout!\n"),
}
```

```go
ctx := req.Context()
```

```go
select {
case <-ctx.Done():
	return ctx.Err()
// some work
case <-time.After(5 * time.Second):
	fmt.Println("work done!")
	return nil
}
```

```go
rows, err := db.QueryContext(ctx, "SELECT product, price FROM catalog")
if err != nil {
	return nil, err
}
```

Summary

- Setting HTTP Server Timeouts is important to conserve resources and to protect from DDOS attack.
- http.TimeoutHandler() can be used to set timeout for our handler functions.
- Request Context can be used to propagate the cancellation signal down the call graph.

Interfaces

“if something can do this, then it can be used here”

```go
func (m *Metal) Density() float64 {
    return m.mass / m.volume
}
```

```go
func (g *Gas) Density() float64 {
    var density float64
    density = (g.molecularMass * g.pressure) / (0.0821 * (g.temperature + 273))
    return density
}
```

Interfaces

- **Abstract Type**

```go
type Dense interface {
  Density() float64
}
```

- Defines Behaviours - **set of method signatures.**

```go
func IsDenser(a, b *Metal) bool {
    return a.Density() > b.Density()
}
```

```go
func IsDenser(a, b Dense) bool {
	return a.Density() > b.Density()
}
```

```go
type Dense interface {
	Density() float64
}
```

```go
result := IsDenser(&gold, &silver)
```

```go
result = IsDenser(&oxygen, &hydrogen)
```

How is interface able to **dynamically dispatch** to correct method and receiver value?

Type is compile time property

```
var a *Metal

a = &gold

a.Density()
```

```
(*Metal)Density()
```

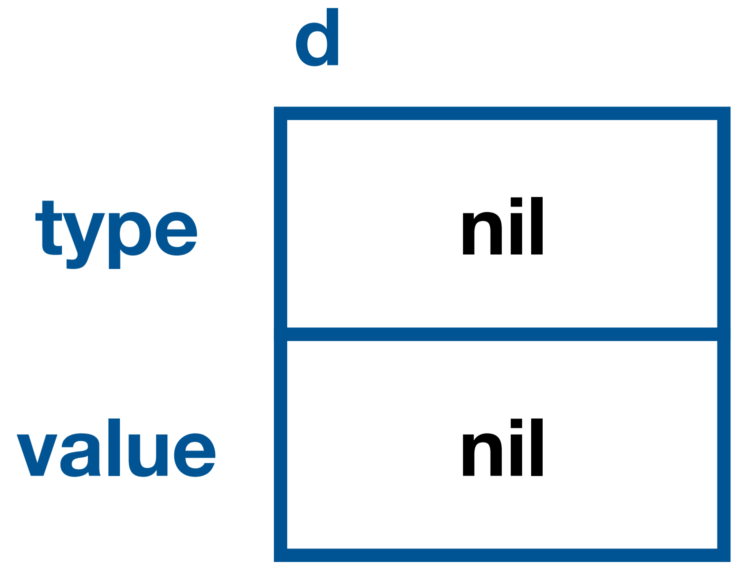
```
var a Dense

a = &gold

a.Density()
```

?

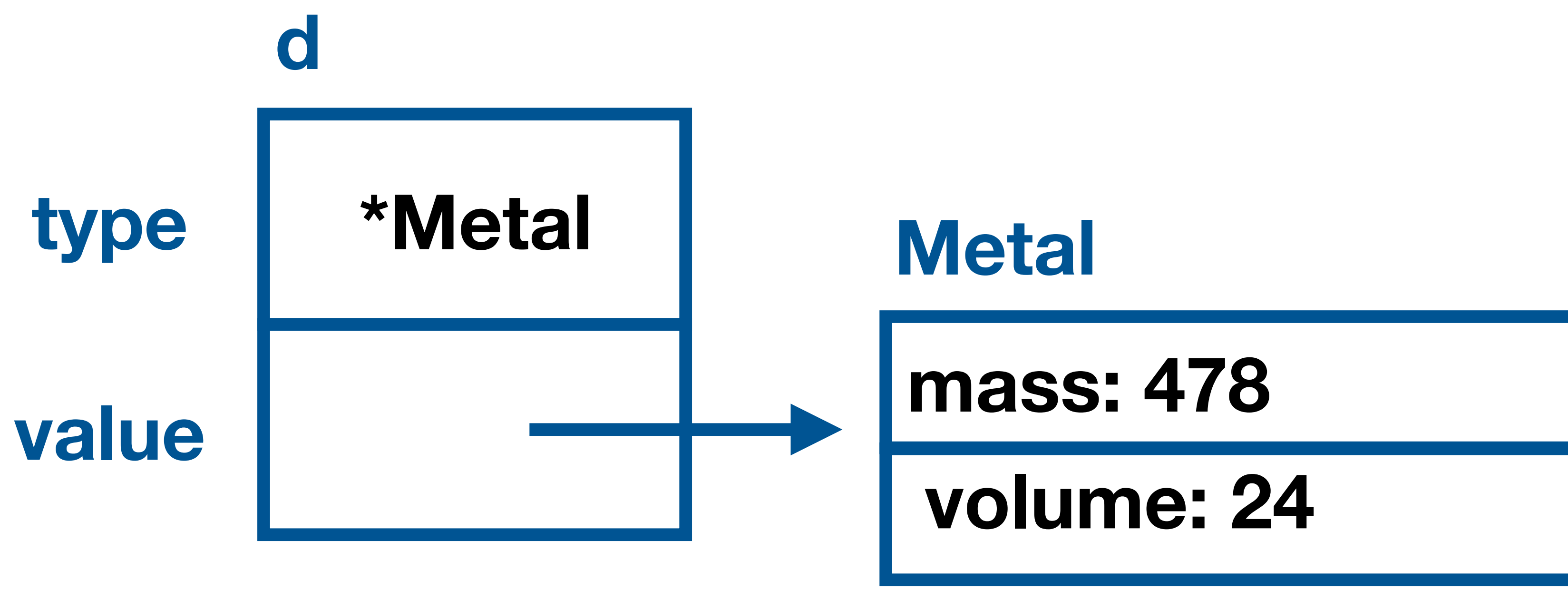
Interface values

- Dynamic type
- Dynamic value

```
var d Dense
```



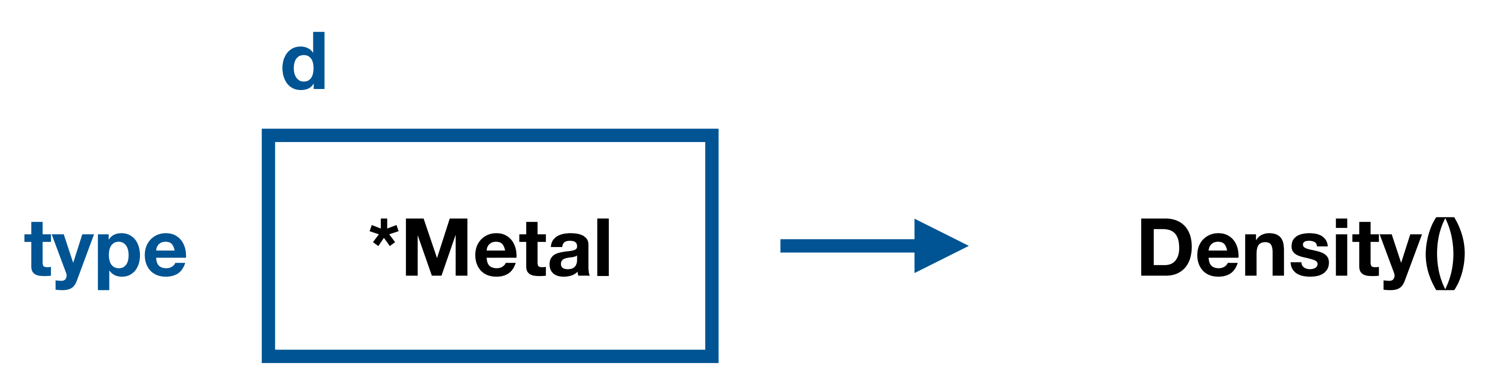

```
d = &gold
```



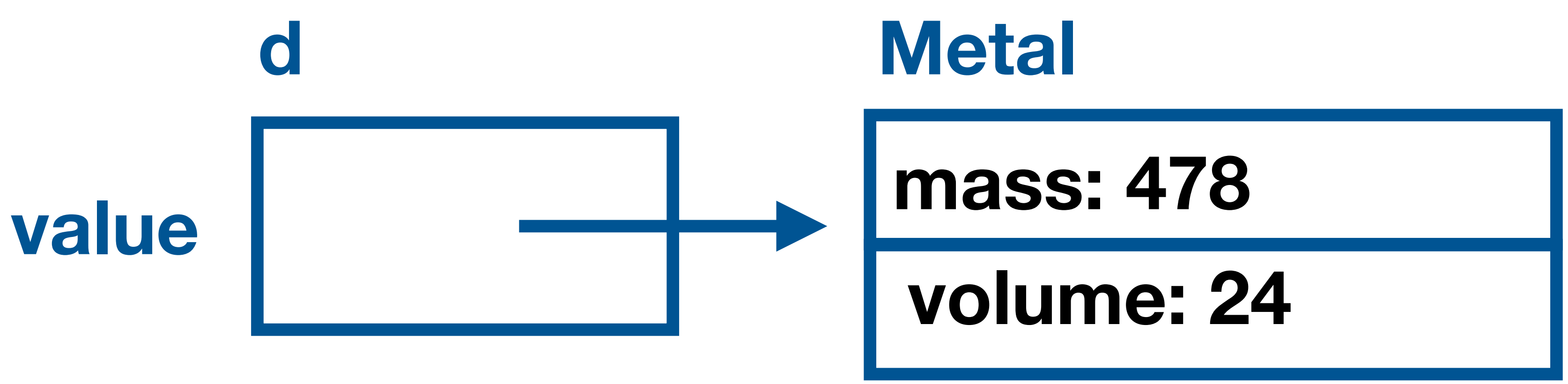
```
d.Density()
```

- Method call through an interface must use dynamic dispatch.

- Compiler would have generated code to obtain the address of the method from the type descriptor, then make an indirect call to that address.



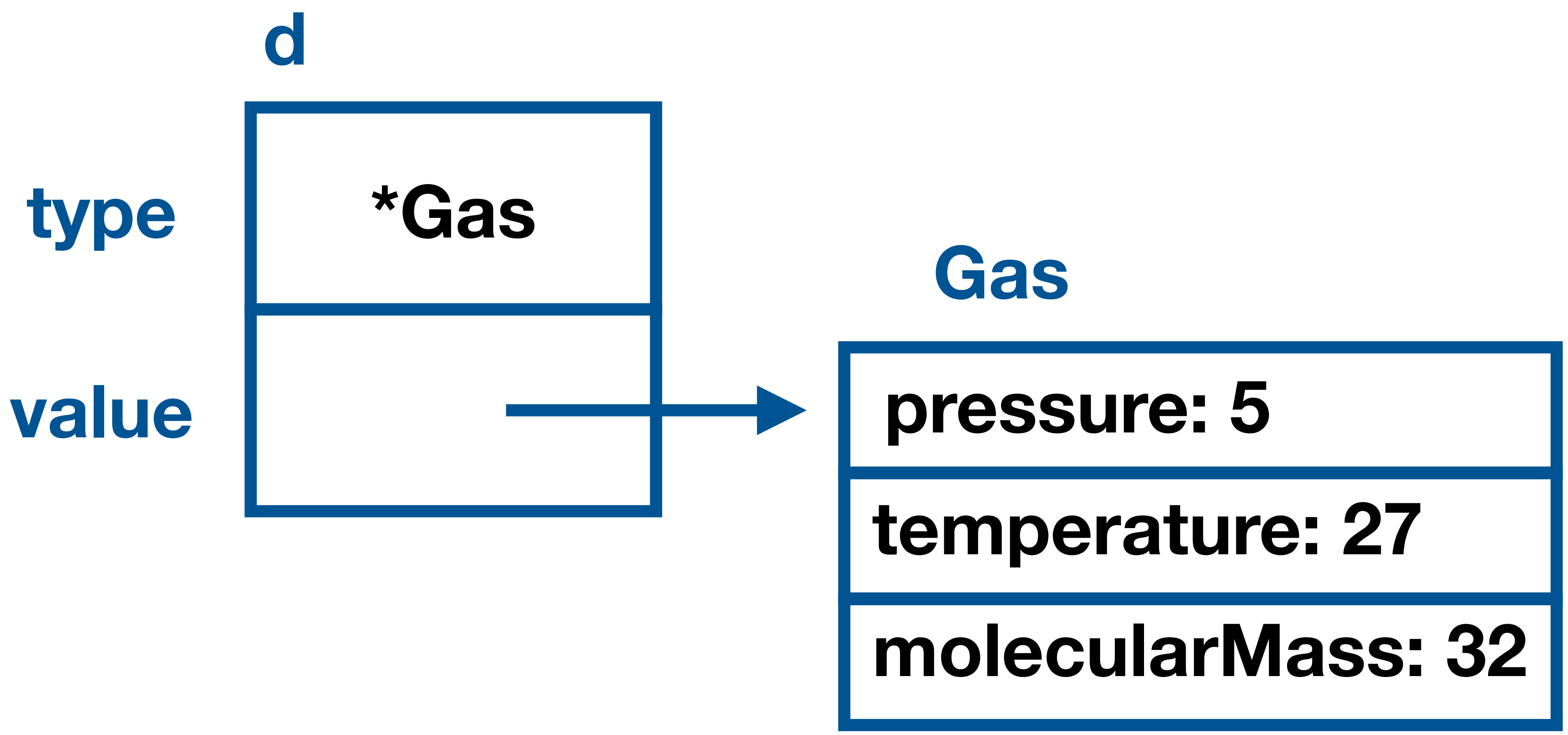
- The receiver argument for the call is a copy of the interface's dynamic value.



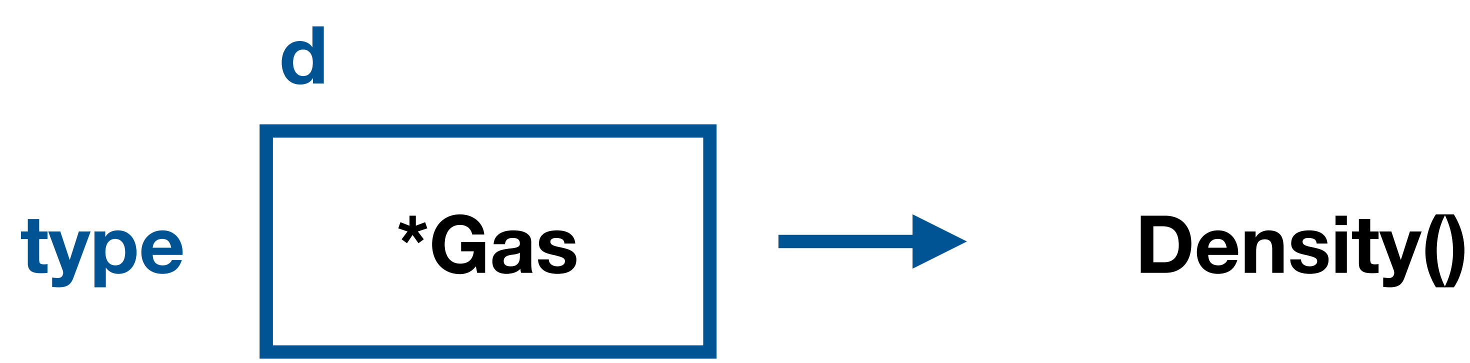

```
d.Density()
```

```
gold.Density()
```

```
d = &oxygen
```




```
d.Density()
```



Purpose of Interface

Encapsulate

- Interface enables us to encapsulate the logic within user defined data type.

```go
func (m *Metal) Density() float64 {
    return m.mass / m.volume
}
```

```go
func (g *Gas) Density() float64 {
  var density float64
  density = (g.molecularMass * g.pressure) / (0.0821 * (g.temperature + 273))
  return density
}
```

Abstraction

- Interface provides **abstraction** for higher level functions with guarantee on behaviour of the underlying concrete type.

```go
func IsDenser(a, b Dense) bool {
  return a.Density() > b.Density()
}
```

```go
result := IsDenser(&gold, &silver)
```

```go
result = IsDenser(&oxygen, &hydrogen)
```

Implicit Interfaces Lead To Good Design

Interfaces

- Interfaces are satisfied **implicitly**.

```
class Bicycle implements Vehicle{
```

Interfaces

- User defined data types just need to possess the methods defined in interface to be considered an instance of interface.

Definition of Interface is decoupled from implementation

Interfaces

- We are not locking us with abstraction at the start of a project.
- we can define interfaces as and when abstractions become apparent.
- This design lets us create new interfaces that are satisfied by existing concrete types, without changing the existing types.

Interfaces

- Interface definition and concrete type definition could appear in any package without prearrangement.

- It makes it easier to abstract dependencies.

Convention

- Keep Interfaces simple and short.
- Define interface when there are two or more concrete types that must be dealt with in a uniform way.
- Create smaller interfaces with fewer, simpler methods.

“ask only for what is needed”

Interfaces from Standard Library

```go
package main

import (
  "bytes"
  "fmt"
  "os"
)

func main() {
  var buf bytes.Buffer

  fmt.Fprintf(os.Stdout, "hello ")
  fmt.Fprintf(&buf, "world")
}
```

Package fmt

```go
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error) {
	p := newPrinter()
	p.doPrintf(format, a)
	n, err = w.Write(p.buf)
	p.free()
	return
}
```

Package io

```go
type Writer interface {
	Write(p []byte) (n int, err error)
}
```

Package os

```go
func (f *File) Write(b []byte) (n int, err error) {
    if err := f.checkValid("write"); err != nil {
        return 0, err
    }
    n, e := f.write(b)
    if n < 0 {
        n = 0
    }
    ...
```

Package bytes

```go
func (b *Buffer) Write(p []byte) (n int, err error) {
    b.lastRead = opInvalid
    m, ok := b.tryGrowByReslice(len(p))
    if !ok {
        m = b.grow(len(p))
    }
    return copy(b.buf[m:], p), nil
}
```

Package fmt

```go
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error) {
	p := newPrinter()
	p.doPrintf(format, a)
	n, err = w.Write(p.buf)
	p.free()
	return
}
```

Package io

```go
type Writer interface {
	Write(p []byte) (n int, err error)
}
```

```go
package main

import (
  "bytes"
  "fmt"
  "os"
)

func main() {
  var buf bytes.Buffer

  fmt.Fprintf(os.Stdout, "hello ")
  fmt.Fprintf(&buf, "world")
}
```

io.Writer interface

- The io.Writer interface type is one of the most widely used interfaces.
- It provides an abstraction of all the types to which bytes can be written, which includes
    - Files
    - Memory buffers
    - Network connections
    - HTTP clients

Other interface types in package io

```go
type Reader interface {
  Read(p []byte) (n int, err error)
}
```

```go
type Closer interface {
  Close() error
}
```

```go
type ReadWriter interface {
  Reader
  Writer
}
```

```go
type ReadWriteCloser interface {
  Reader
  Writer
  Closer
}
```

Stringer Interface

```go
type Stringer interface {
  String() string
}
```

- Stringer interface provides a way for types to control how their values are printed.

- The fmt package functions (Println, Fprintln,..) checks if concrete type has string method, if it does, then they call string method of the type to format values.

Summary

- io.Writer interface provides an abstraction of all the types to which bytes can be written.

- Stringer interface provides a way for types to format their values for print.

Interface Satisfaction

Interface Satisfaction

- A type satisfies an interface if it implements all the methods the interface requires.

*os.File

- *os.File satisfies io.Reader, Writer, Closer, and ReadWriter.

```
func (f *File) Read(b []byte) (n int, err error)
```

```
func (f *File) Write(b []byte) (n int, err error)
```

```
func (file *file) close() error
```

*bytes.Buffer

- A bytes.Buffer satisfies io.Reader, Writer, and ReadWriter.

```go
func (b *Buffer) Read(p []byte) (n int, err error)

func (b *Buffer) Write(p []byte) (n int, err error)
```

- Does not satisfy Closer as it does not have a Close method.

Assignability Rule

- An expression may be assigned to an interface only if its type satisfies the interface.

```go
package main

import (
	"bytes"
	"fmt"
	"io"
	"os"
	"time"
)

func main() {
	var w io.Writer
	w = os.Stdout
	w = new(bytes.Buffer)
	w = time.Second
	fmt.Println(w)
}
```

```
cannot use time.Second (constant 1000000000 of type
time.Duration) as io.Writer value in assignment: missing
method Write compiler
```

Concealing the concrete type and value

- Interface wraps concrete type, **only methods defined by interface are revealed** even if concrete type implements others.

```go
os.Stdout.Write([]byte("hello"))
os.Stdout.Close()
```

```go
package main

import (
	"os"
)

func main() {
	printer(os.Stdout, "hello")
}

func printer(f *os.File, str string) {
	f.Write([]byte(str))
}
```

```go
os.Stdout.Write([]byte("hello"))
os.Stdout.Close()
```

```go
package main

import (
	"os"
)

func main() {
	printer(os.Stdout, "hello")
}

func printer(f *os.File, str string) {
	f.Write([]byte(str))
	f.Close()
}
```

```go
package main

import (
	"os"
)

func main() {
	printer(os.Stdout, "hello")
}

func printer(f *os.File, str string) {
	f.Write([]byte(str))
}
```

```go
package main

import (
  "io"
  "os"
)

func main() {
  printer(os.Stdout, "hello")
}

func printer(w io.Writer, str string) {
  w.Write([]byte(str))
}
```

```go
package main

import (
	"io"
	"os"
)

func main() {
	printer(os.Stdout, "hello")
}

func printer(w io.Writer, str string) {
	w.Write([]byte(str))
	w.Close()
}
```

```
w.Close undefined (type io.Writer has no field or method
Close) compiler
```

Summary

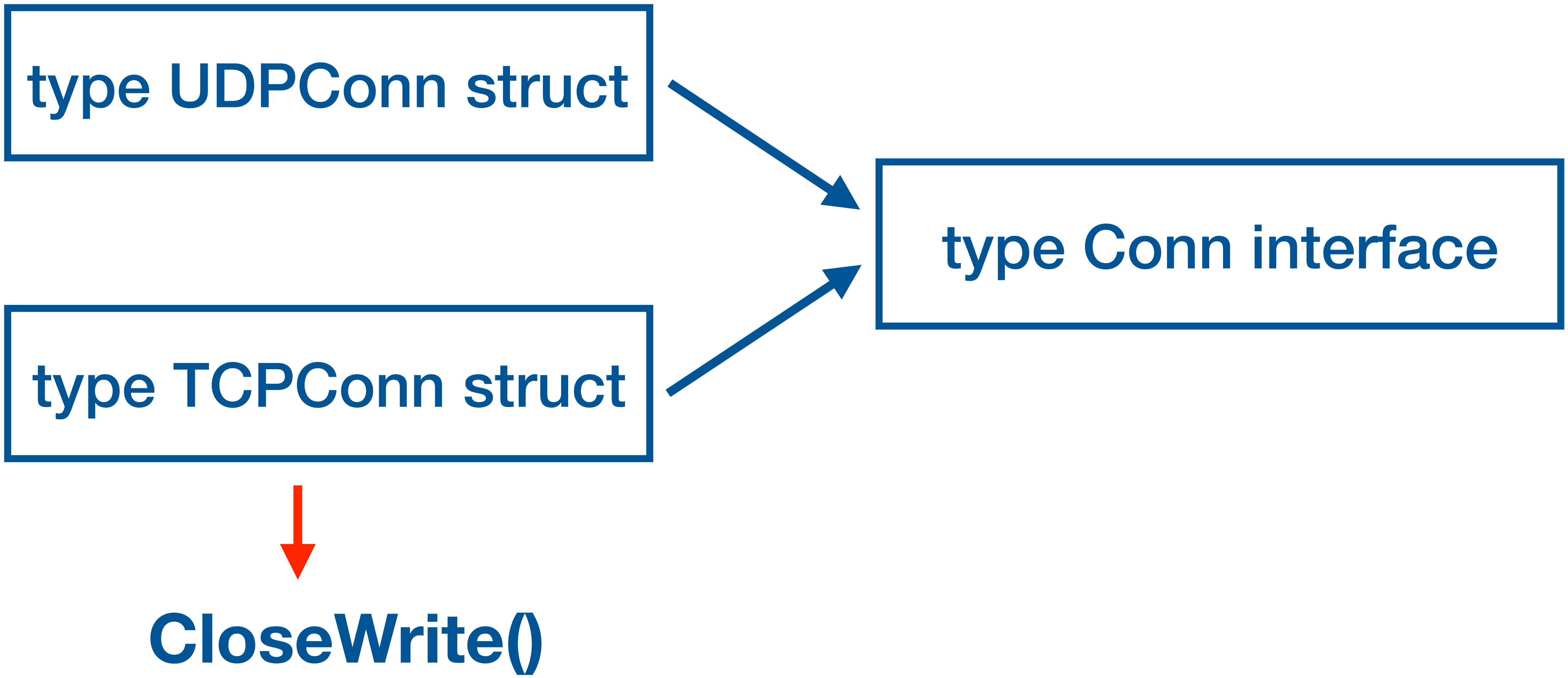
- A type satisfies an interface if it implements all the methods the interface defines.

- Interface wraps concrete type.

- Only methods defined by interface are revealed even if concrete type implements other methods.

Type assertion

Package net


type UDPConn struct
type TCPConn struct
type Conn interface
CloseWrite()

```go
func shutdownWrite(conn net.Conn) {
  // Call .CloseWrite to shuts down the writing side
  // of the TCP connection.
}
```

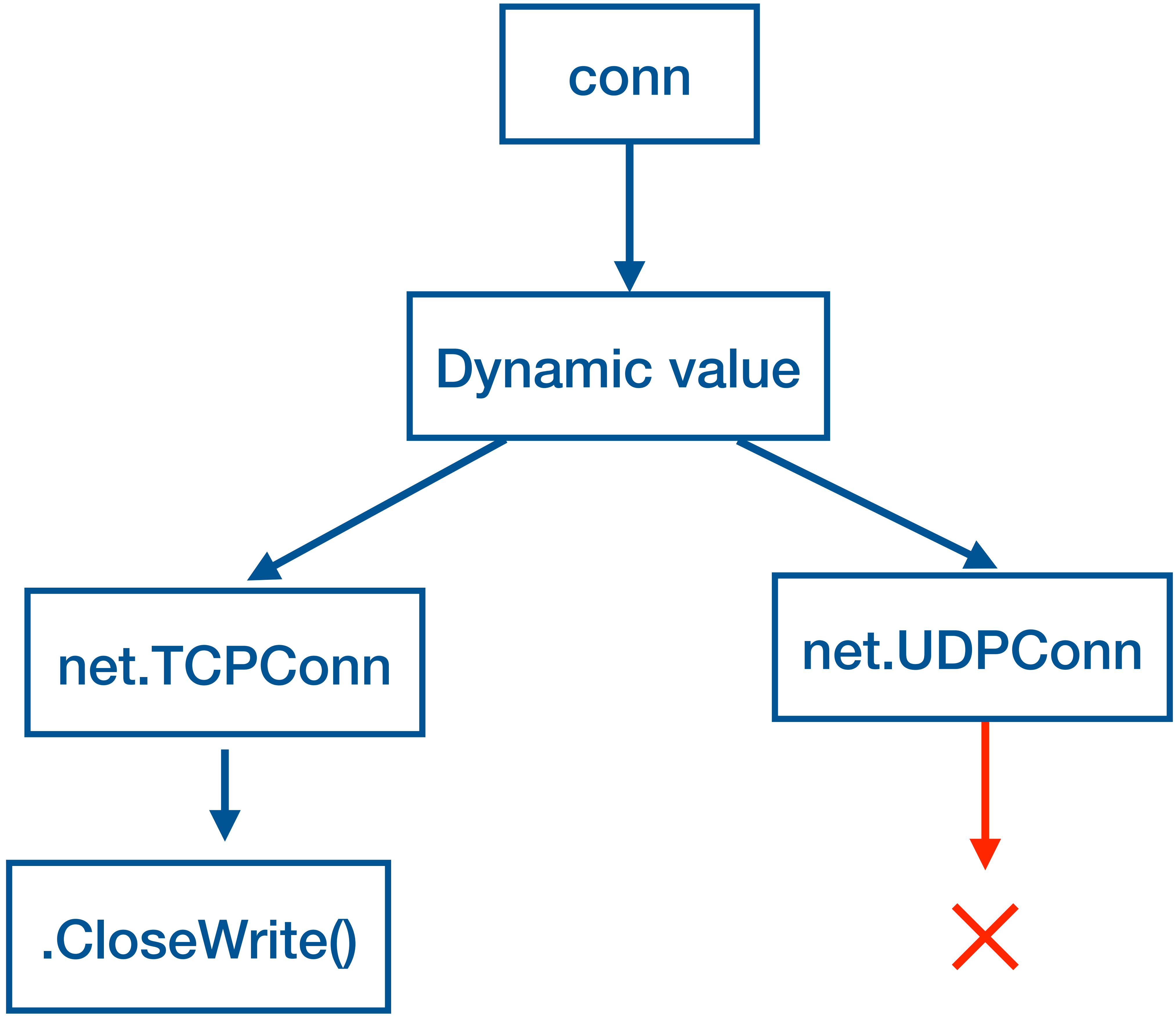
UDPConn

TCPConn

```go
func shutdownWrite(conn net.Conn) {
	// Call .CloseWrite to shuts down the writing side
	// of the TCP connection.
}
```

```go
func shutdownWrite(conn net.Conn) {
  // Call .CloseWrite to shuts down the writing side
  // of the TCP connection.
  conn.CloseWrite()
}
```

net.TCPConn.CloseWrite()


conn
Dynamic value
net.TCPConn
net.UDPConn
.CloseWrite()

Type assertion

- A type assertion is an operation applied to an interface value.

- Extract dynamic value from interface value.

Type assertion

```
v := x.(T)
```

- Checks interface values's dynamic type is identical to concrete type typename, returns dynamic value.

- If check fails, then the operation panics.

Type assertion

v, ok := x.(T)

- The second return value is boolean.

- If type assertion is successful,
  v == dynamic value, ok == true

- If type assertion is fails,
  v == zero value of type, ok == false

- No run-time panic occurs in this case.

```go
func shutdownWrite(conn net.Conn) {
  v, ok := conn.(*net.TCPConn)
  if ok {
    v.CloseWrite()
  }
}
```

```go
func shutdownWrite(conn net.Conn) {
  v, ok := conn.(*net.TCPConn)
  if ok {
    v.CloseWrite()
  }
}
```

Summary

- Type assertion is used to get concrete value from interface value by specifying the explicit type.

- Type assertion is useful to apply distinguished operation of the type.

Empty Interface

Empty Interface

```go
func fmt.Println(a ...interface{}) (n int, err error)
```

```go
func fmt.Errorf(format string, a ...interface{}) error
```

- Empty interface specifies no methods.
- We can assign any value to the empty interface.

```go
package main

import "fmt"

func main() {
  describe(42)
  describe("hello")
}

func describe(value interface{}) {
  switch v := value.(type) {
  case int:
    fmt.Printf("v is integer with value %d\n", v)
  case string:
    fmt.Printf("v is a string, whose length is %d\n", len(v))
  default:
    fmt.Println("we dont know what 'v' is!")
  }
}
```

Type Switch

```go
switch v := value.(type) {
case int:
    fmt.Printf("v is integer with value %d\n", v)
case string:
    fmt.Printf("v is a string, whose length is %d\n", len(v))
default:
    fmt.Println("we dont know what 'v' is!")
}
```

- Used to discover the dynamic type of an interface variable.

```go
package main

import "fmt"

func main() {
	describe(42)
	describe("hello")
}

func describe(value interface{}) {
	switch v := value.(type) {
	case int:
		fmt.Printf("v is integer with value %d\n", v)
	case string:
		fmt.Printf("v is a string, whose length is %d\n", len(v))
	default:
		fmt.Println("we dont know what 'v' is!")
	}
}
```

```
$ go run .
v is integer with value 42
v is a string, whose length is 5
```

```go
package main

import "fmt"

func main() {
	describe(42)
	describe("hello")
}

func describe(value interface{}) {
	switch v := value.(type) {
	case int:
		fmt.Printf("v is integer with value %d\n", v)
	case string:
		fmt.Printf("v is a string, whose length is %d\n", len(v))
	default:
		fmt.Println("we dont know what 'v' is!")
	}
}
```

```
$ go run .
v is integer with value 42
v is a string, whose length is 5
```

Caution while using Empty Interfaces

- Empty interface gives no knowledge of about data coming in.
- Benefits of static typed language is nullified.
- Need to use reflect library to turn arbitrary structs into specific type.

- Prefer to use specific data type.

- Create an interface with some specific methods that we need.