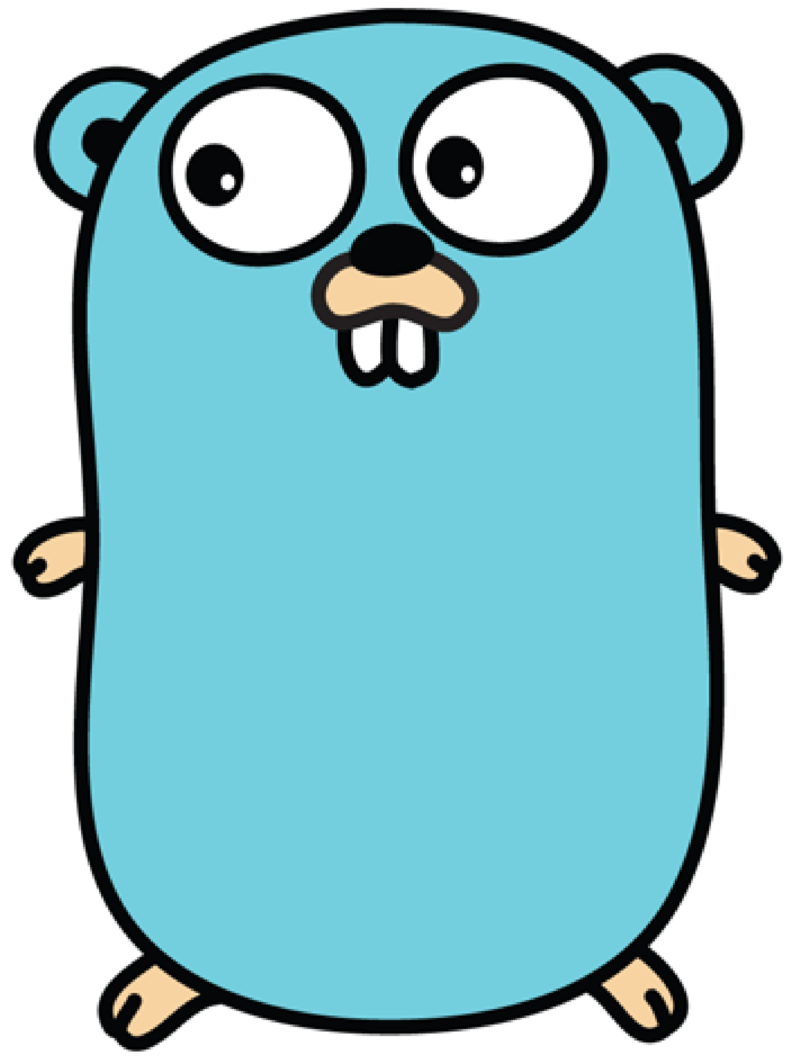
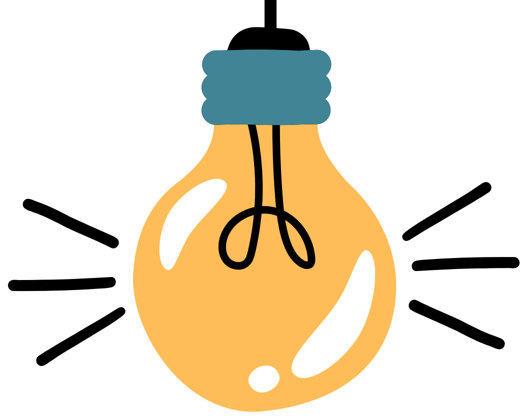
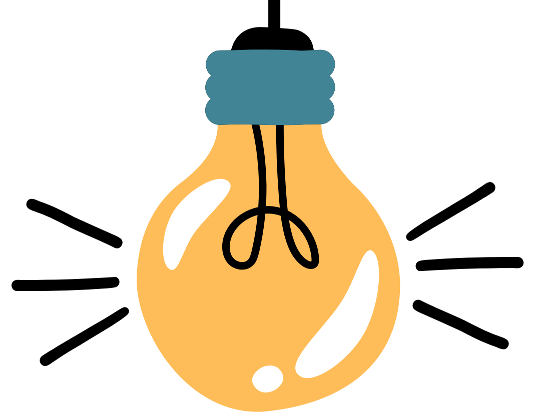
CONCURRENCY in GOLANG

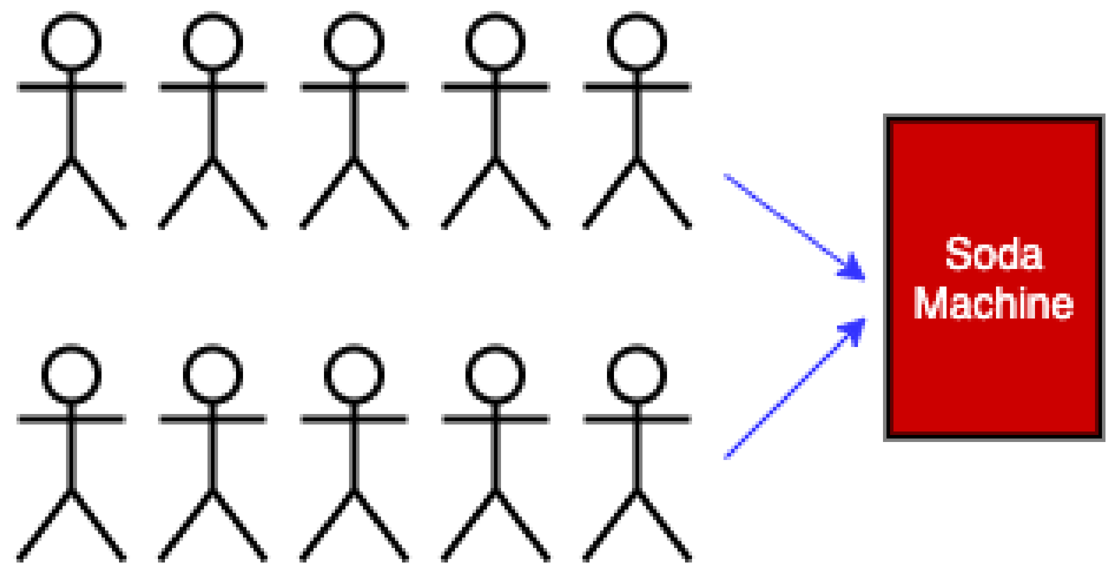


Aditi Garg
23MA10002

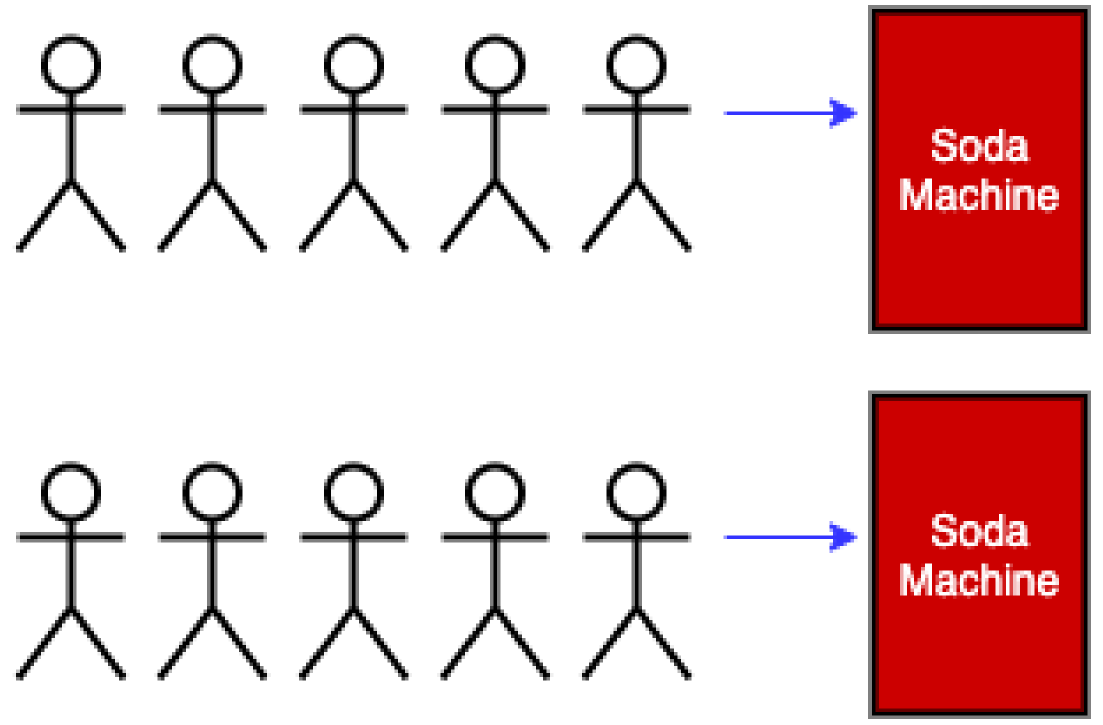



CONTENTS

CONCURRENCY V/S PARALLELISM



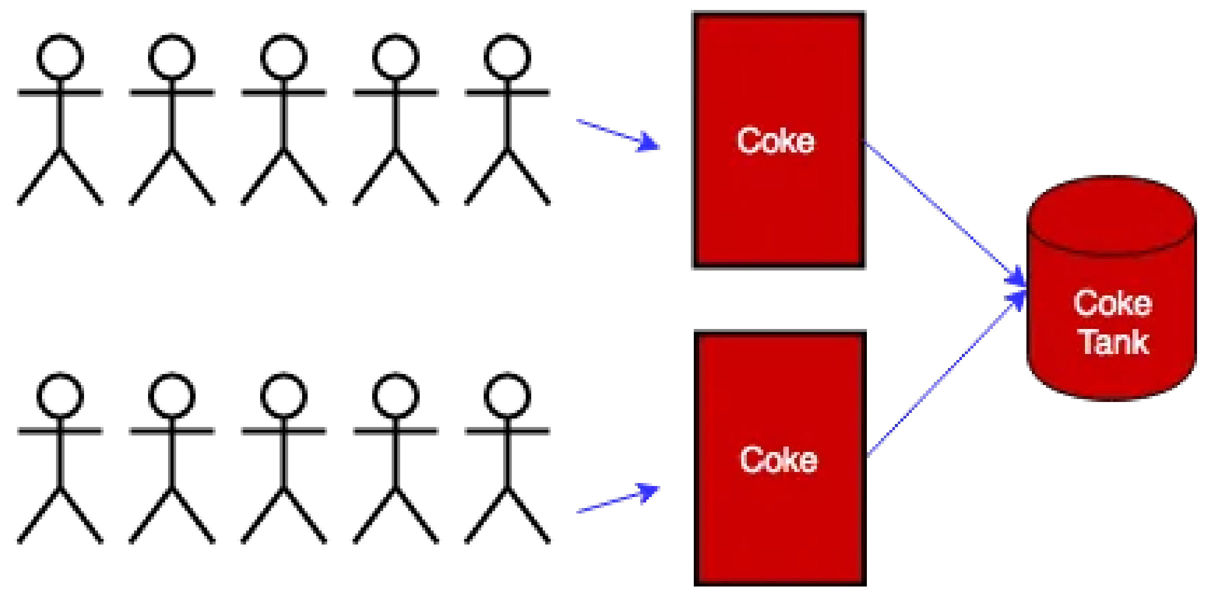
Concurrent: 2 queue, 1 dispensing machine



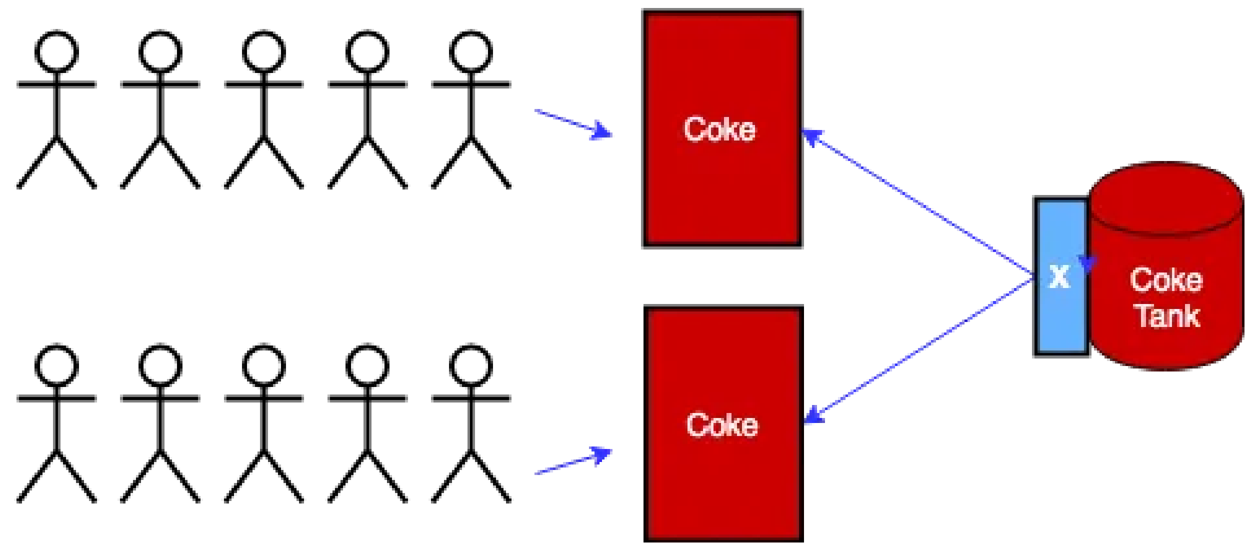
Parallel: 2 queue, 2 dispensing machines

Concurrency	Parallelism
Concurrency is the task of running and managing the multiple computations at the same time.	While parallelism is the task of running multiple computations simultaneously.
Concurrency is achieved through the interleaving operation of processes on the central processing unit(CPU) or in other words by the context switching.	While it is achieved by through multiple central processing units(CPUs).
Concurrency can be done by using a single processing unit.	While this can't be done by using a single processing unit. it needs multiple processing units.
Concurrency increases the amount of work finished at a time.	While it improves the throughput and computational speed of the system.
Concurrency deals lot of things simultaneously.	While it do lot of things simultaneously.
Concurrency is the non-deterministic control flow approach.	While it is deterministic control flow approach.
In concurrency debugging is very hard.	While in this debugging is also hard but simple than concurrency.

PROBLEM : INDEPENDENTLY RUNNING THREADS



To save on installation costs, a company used a shared tank filled with 500 L daily for vending machines dispensing 500ml soda only if sufficient liquid was present. At day end, with 500ml left, multiple machines operating simultaneously led to Serena and Venus receiving less than 500ml due to shared variable issues from independently running threads



he application X holds a boolean variable (tankInUse) whether the tank is in use by some vending machine or not. If this variable is true the vending machine would hold till the variable is again false. This would very well impact the efficiency of the dispense.
So they came up with an optimization that they would only consider this flag if the edge condition is reached which is tank has only 500ml soda left. Thus now they need to have an extra variable on the application X, something like volumeLeft.

SYNCHRONIZATION

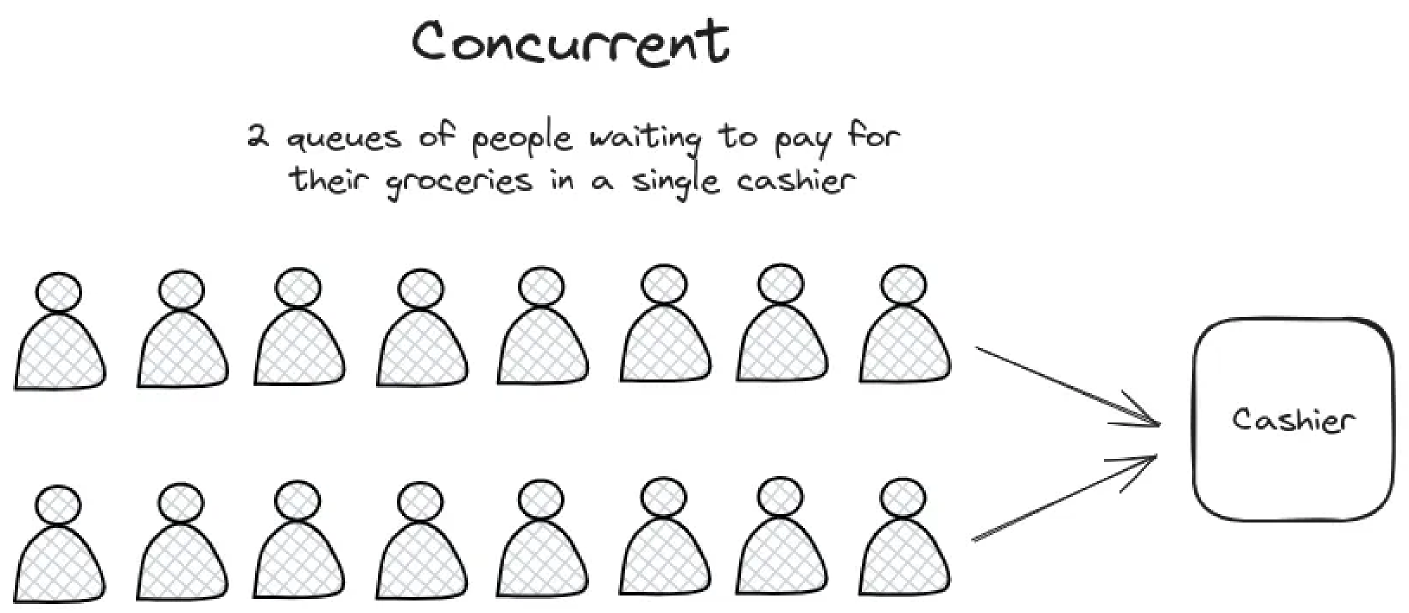
Let’s consider this scenario you are just checking in to this hotel and the last room is getting vacant at the same moment, the other person is completing his payment and the manager is not so optimistic thus asks you to wait until it completes. You would be okay with it, right? as it would be just a couple of minutes. That much locking you can adhere to.

Concurrency can be achieved by using multiple processors, cores, or threads, depending on the level of parallelism that you want to achieve. Synchronization is the coordination of your concurrent tasks, to ensure that they do not interfere with each other, or access shared resources in an inconsistent or unsafe way.
In Go to solve this problem we have Mutex.

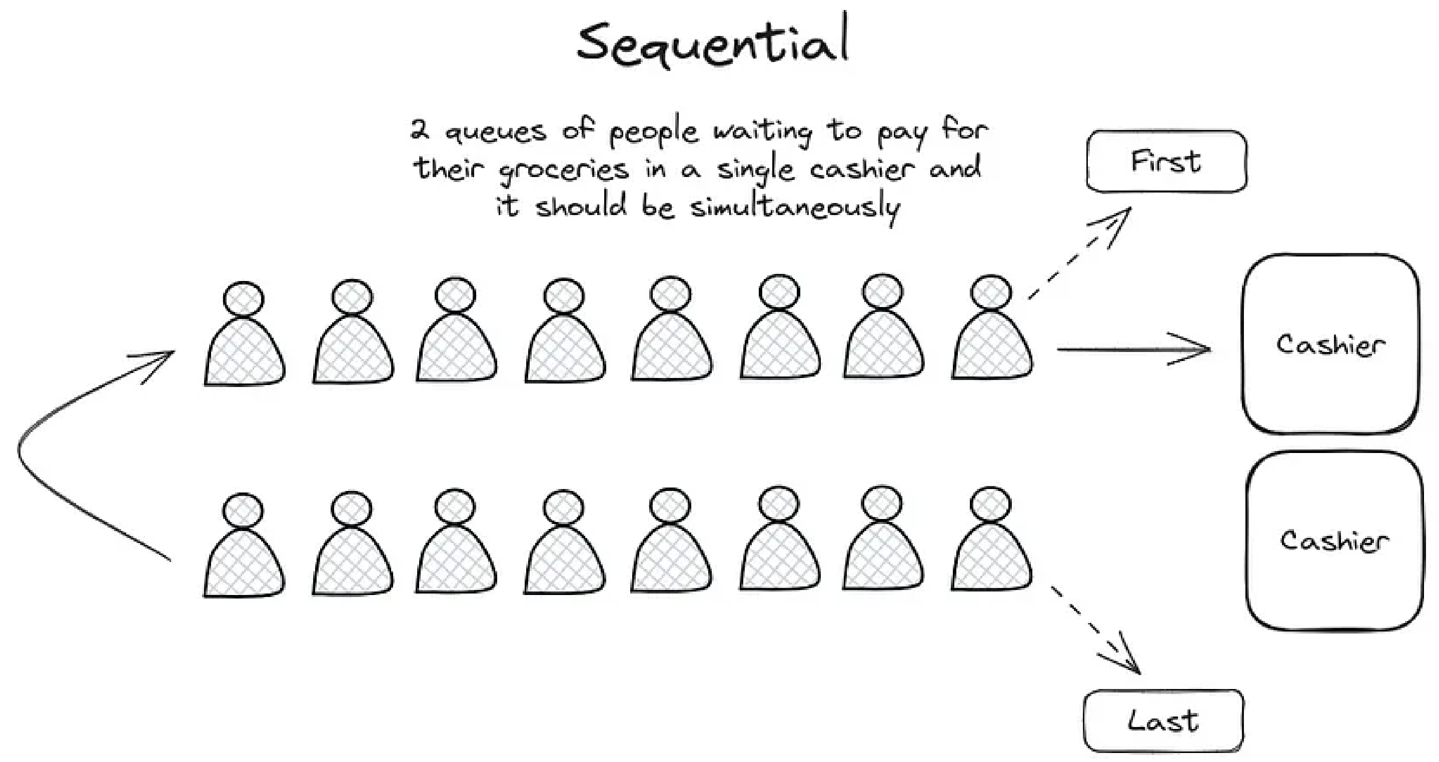
CONCURRENT VS SEQUENTIAL PROGRAMMING

- Concurrency is the composition of independently executing computations.
- Concurrency is a way to structure software, particularly as a way to write clean code that interacts well with the real world.
- It is not parallelism.

- Sequential Computing is a traditional method where instructions are executed one after the other, causing delays as each instruction waits for the previous one to finish.
- This approach uses a single processor, leading to slower performance due to high workload.



Concurrent
2 queues of people waiting to pay for their groceries in a single cashier
Cashier



Sequential
2 queues of people waiting to pay for their groceries in a single cashier and it should be simultaneously
First
Cashier
Cashier
Last

WHY CONCURRENT PROGRAMMING AND WHY NOT?

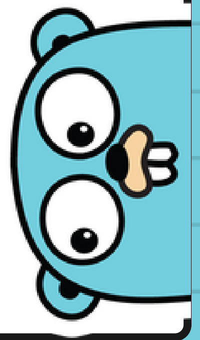
Advantages of Concurrency:

1. Multitasking: Run multiple applications simultaneously, enhancing productivity and user experience.
2. Efficient Resource Allocation: Optimize resource usage by allowing unused resources from one application to be utilized by others.
3. Improved Responsiveness: Concurrent execution prevents one application from monopolizing resources, leading to faster response times for all.
4. Enhanced Performance: By efficiently utilizing hardware resources, concurrency boosts overall system performance and throughput.

Drawbacks of Concurrency:

1. Interference Management: Requires mechanisms to prevent interference between concurrent applications, ensuring they operate smoothly together.
2. Coordination Overhead: Additional coordination mechanisms are needed to manage concurrent tasks effectively.
3. Performance Overheads: Introducing concurrency adds complexity and overhead to the operating system, potentially impacting performance.
4. Risk of Degraded Performance: Excessive concurrent tasks can overwhelm system resources, leading to performance degradation.

HOW CONCURRENT PROGRAMMING IS DONE IN GO?



Go Routines

- A goroutine is a lightweight units of Go Concurrency managed by the Go runtime.
- In simple words, every concurrently executing activity in Go language is known as a Goroutines.

Syntax

```
func name(){
// statements
}

// using go keyword as the
// prefix of your function call
go name()
```

Example

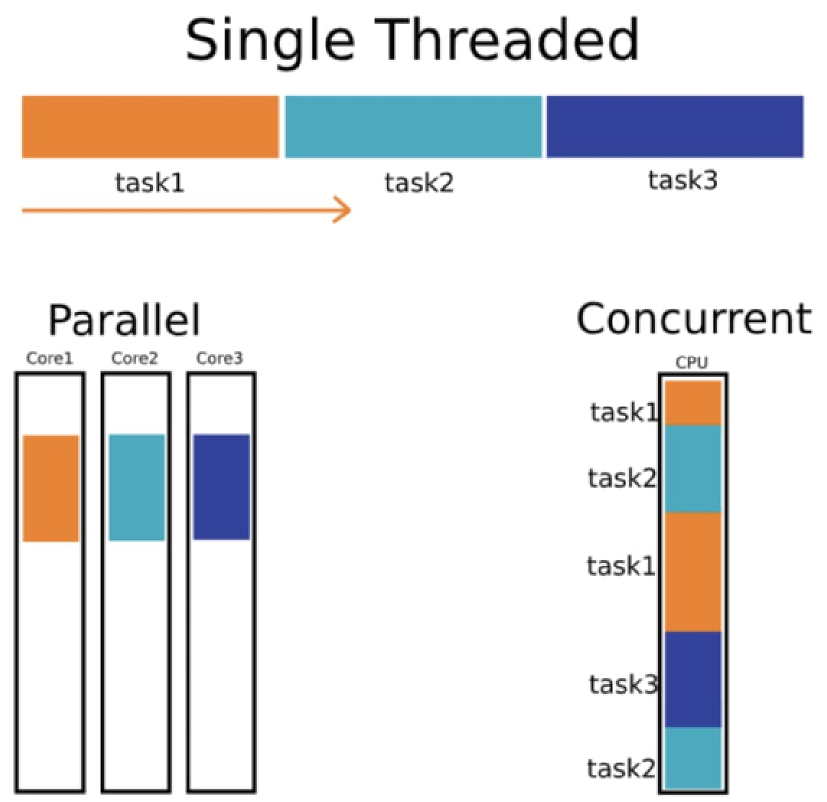
```
package main

import "fmt"

func display(str string) {
  for w := 0; w < 6; w++ {
    fmt.Println(str)
  }
}

func main() {

  // Calling Goroutine
  go display("Go Routine")

  // Calling normal function
  display("Concurrency")
}
```

Example Modified

```
package main

import (
  "fmt"
  "time"
)

func display(str string) {
  for w := 0; w < 6; w++ {
    time.Sleep(1 * time.Second)
    fmt.Println(str)
  }
}

func main() {

  // Calling Goroutine
  go display("Go Routine")

  // Calling normal function
  display("Concurrency")
}
```

ADVANTAGES OF GO ROUTINE

- Goroutines are more cost-effective than threads.
- Goroutines' stack size can dynamically adjust, unlike threads with fixed stack sizes.
- Goroutines communicate through channels to prevent race conditions.
- If a Goroutine blocks a thread, the remaining Goroutines are assigned to a new OS thread transparently to the programmer.



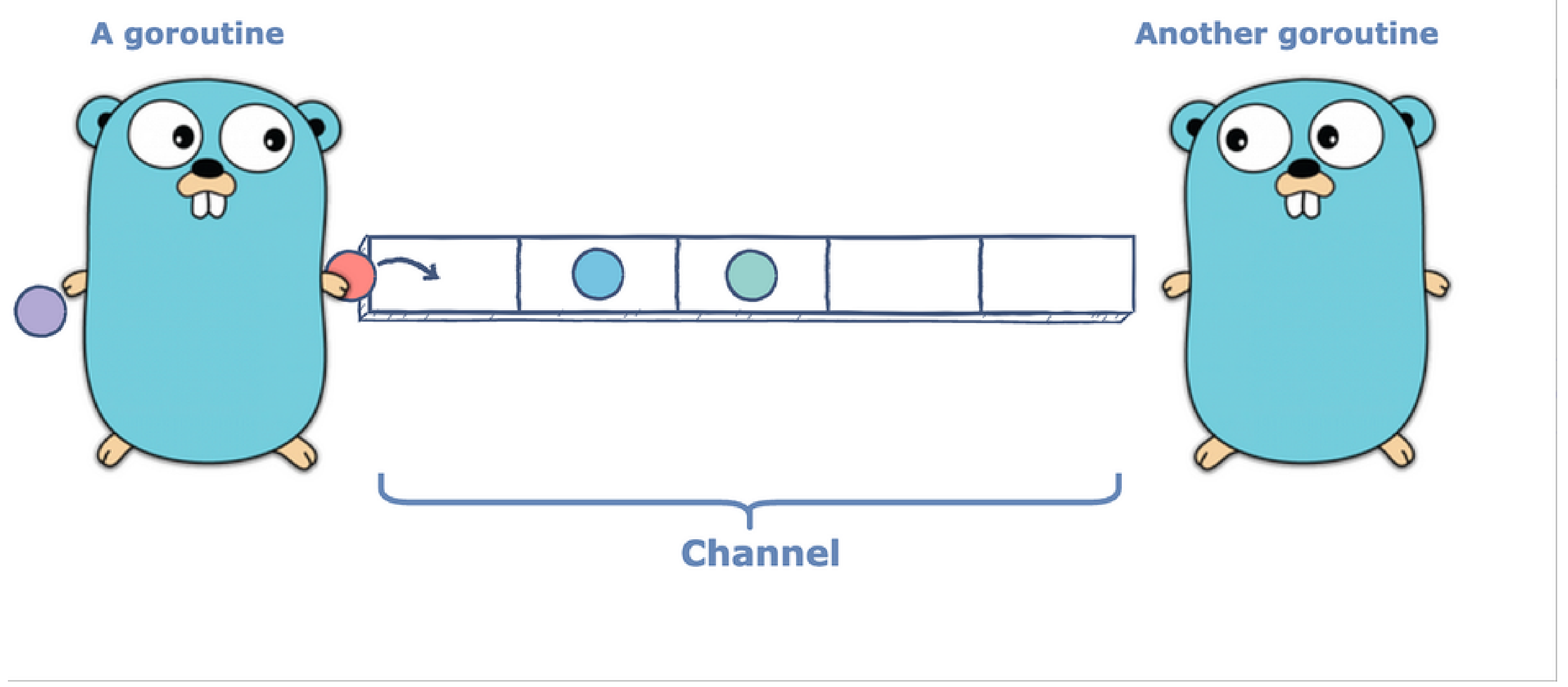
HOW CONCURRENT PROGRAMMING IS DONE IN GO?

Go Channels

Channels are the pipes that connect concurrent goroutines. You can send values into channels from one goroutine and receive those values into another goroutine.
Send a value into a channel using the channel <- syntax.
v := <-ch Receive from ch, and assign value to v.

```
package main

import "fmt"

func main() {

    messages := make(chan string)

    go func() { messages <- "ping" }()

    msg := <-messages
    fmt.Println(msg)
}
```

```
$ go run channels.go
ping
```



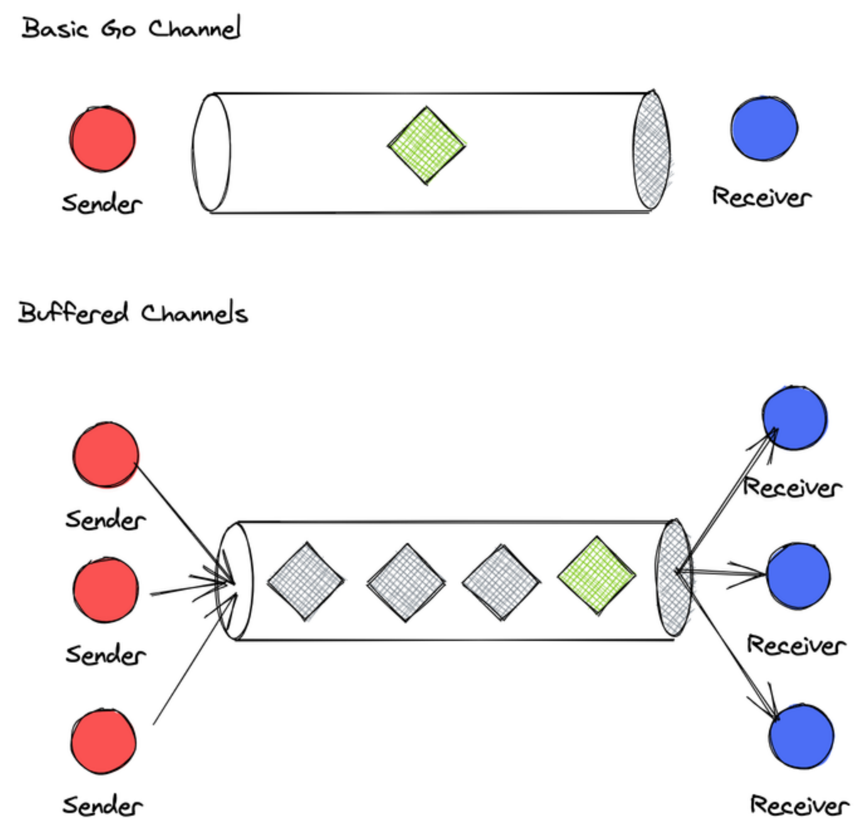
HOW CONCURRENT PROGRAMMING IS DONE IN GO?

Buffered Channels

By default channels are unbuffered, which states that they will only accept sends (chan <-) if there is a corresponding receive (<- chan) which are ready to receive the sent value. Buffered channels allows to accept a limited number of values without a corresponding receiver for those values. It is possible to create a channel with a buffer. Buffered channel are blocked only when the buffer is full. Similarly receiving from a buffered channel are blocked only when the buffer will be empty.

```go
package main
 import (
    "fmt"
)
 func main() {
    // create a buffered channel
    // with a capacity of 2.
    ch := make(chan string, 2)
    ch <- "1"
    ch <- "2"
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```

```
$ go run channels.go
1
2
```



HOW CONCURRENT PROGRAMMING IS DONE IN GO?

Deadlocks in Golang Channel

A deadlock occurs when two or more goroutines are waiting forever for one another to free a shared resource. This can happen if two or more goroutines are vying for the same lock on a resource, but one is already locked out of the one the other requires.

```
package main
import "fmt"
func main() {
    c := make(chan int)
    select {
    case <-c: // this case will never be
selected because of deadlock
        fmt.Println("received") // this will never
be printed
    }
}
```

```
$go run bill.go

fatal error: all goroutines are
asleep - deadlock!

goroutine 1 [chan receive]:
main.main()
 $:7 +0x25
exit status 2
```

HOW CONCURRENT PROGRAMMING IS DONE IN GO?

Waitgroups

WaitGroup is actually a type of counter which blocks the execution of function (or might say A goroutine) until its internal counter become 0.

How It Works ?

WaitGroup exports 3 methods.

- Add(int): It increases WaitGroup counter by given integer value.
- Done(): It decreases WaitGroup counter by 1, we will use it to indicate termination of a goroutine.
- Wait():It Blocks the execution until it’s internal counter becomes 0.

Note: WaitGroup is concurrency safe, so its safe to pass pointer to it as argument for Groutines.

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

func worker(id int) {
    fmt.Printf("Worker %d starting\n", id)

    time.Sleep(time.Second)
    fmt.Printf("Worker %d done\n", id)
}
```

```go
func main() {

 var wg sync.WaitGroup

 for i := 1; i <= 5; i++ {
 wg.Add(1)

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

 wg.Wait()

}
```

```
$ go run waitgroups.go
Worker 5 starting
Worker 3 starting
Worker 4 starting
Worker 1 starting
Worker 2 starting
Worker 4 done
Worker 1 done
Worker 2 done
Worker 5 done
Worker 3 done
```

HOW CONCURRENT PROGRAMMING IS DONE IN GO?

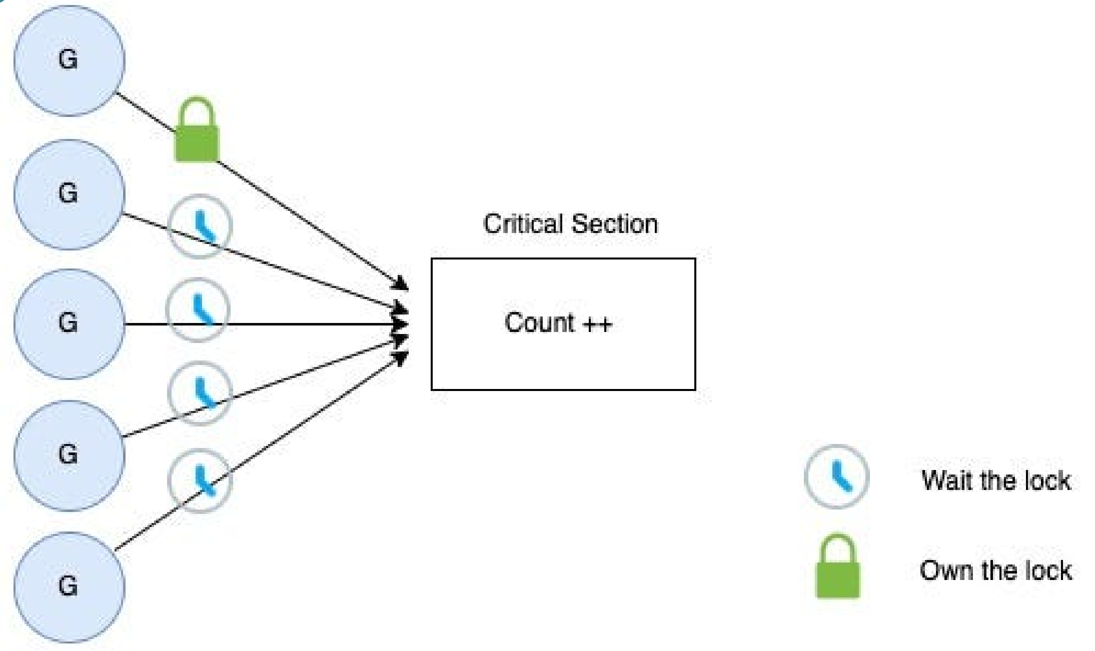
MUTEXES

CRITICAL SECTION: When a program runs concurrently, the parts of code which modify shared resources should not be accessed by multiple Goroutines at the same time. This section of code that modifies shared resources is called critical section.
undesirable situation where the output of the program depends on the sequence of execution of Goroutines is called **race condition.**
A Mutex is used to provide a locking mechanism to ensure that only one Goroutine is running the critical section of code at any point in time to prevent race conditions from happening.
Mutex is available in the sync package. There are two methods defined on Mutex namely Lock and Unlock. Any code that is present between a call to Lock and Unlock will be executed by only one Goroutine, thus avoiding race condition.

```go
package main
import (
 "fmt"
 "sync"
)
var x = 0
func increment(wg *sync.WaitGroup, m
*sync.Mutex) {
 m.Lock()
 x = x + 1
 m.Unlock()
 wg.Done()
}
```

```go
func main() {
 var w sync.WaitGroup
 var m sync.Mutex
 for i := 0; i < 1000; i++ {
 w.Add(1)
 go increment(&w, &m)
 }
 w.Wait()
 fmt.Println("final value of x", x)
}
```

```
final value of x 1000

Program exited.
```



THANK YOU

www.reallygreatsite.com