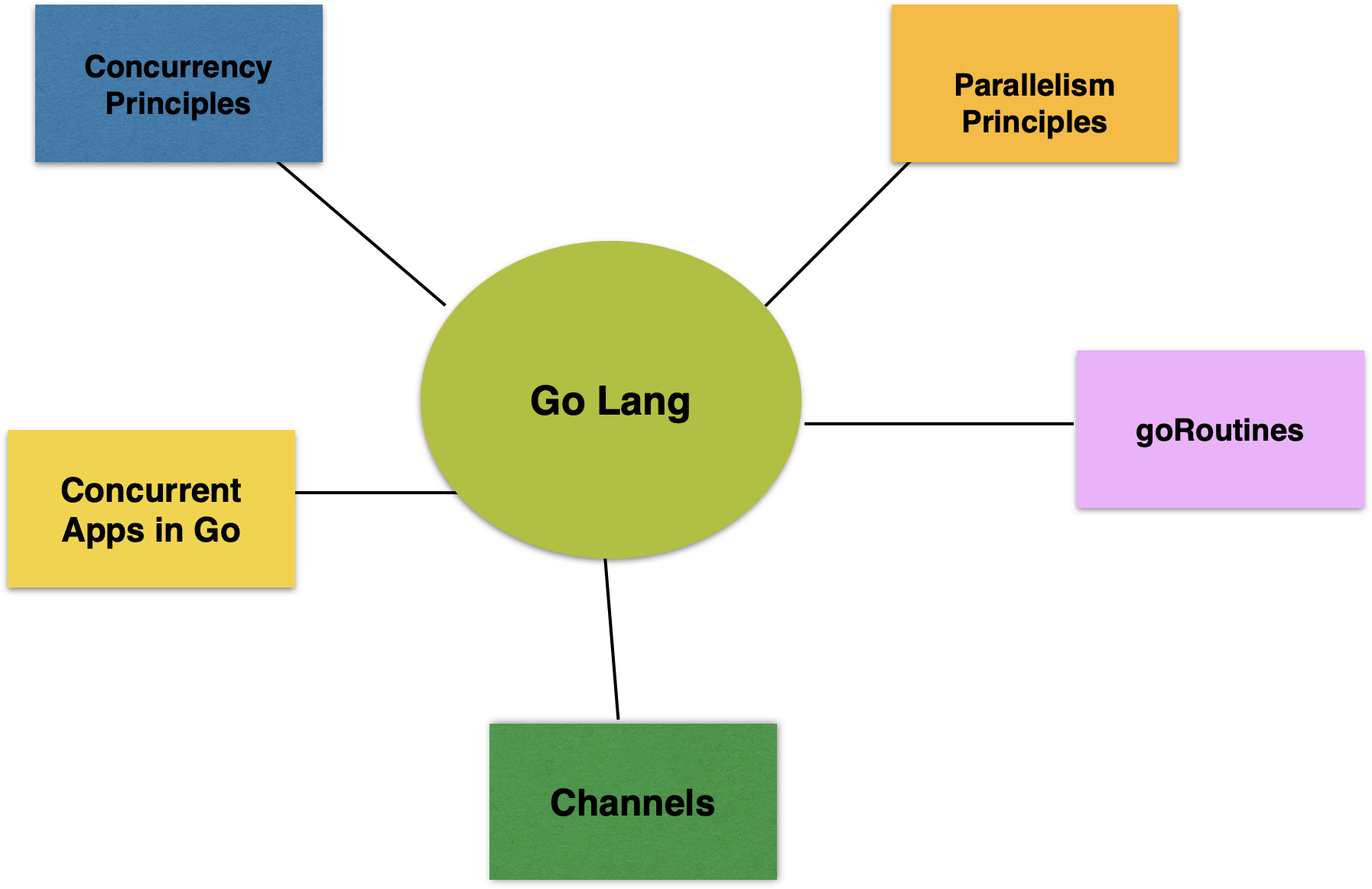
**C H A P T E R 8**

CHAPTER 8

# *Go Concurrent Apps* Introduction

In this chapter, readers will learn how to build Concurrent applications using Go. Readers will know how to use go Routines, Channels, and concurrency using Go. Readers will understand concurrency and parallelism principles.



## Structure

The chapter covers the following topics:

* Concurrency & Parallelism Principles
* goRoutines
* Channels
* Concurrent Apps in Go

## Objectives

You can build concurrent applications in Go Language. Reader will be presented with various examples showing the Go Routines and how to implement Concurrency.

# Concurrency& Parallelism Principles

Let us recap from the chapter 2,

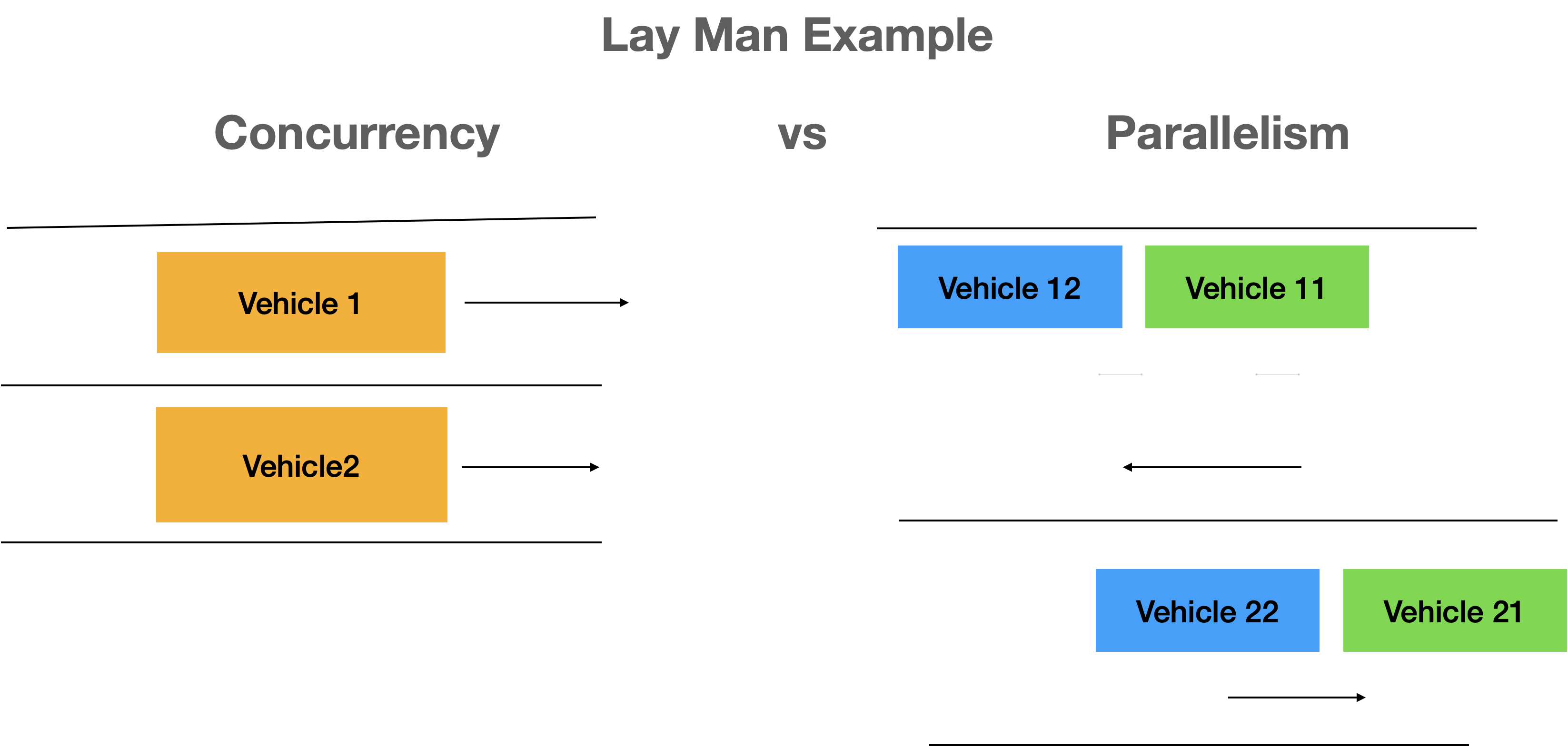
*Concurrency differs from parallelism in that concurrent tasks need to end at the same time and start at the same time. On the other hand, single threaded tasks get executed one by one like code statements.*

Graphical user interface

Description automatically generated

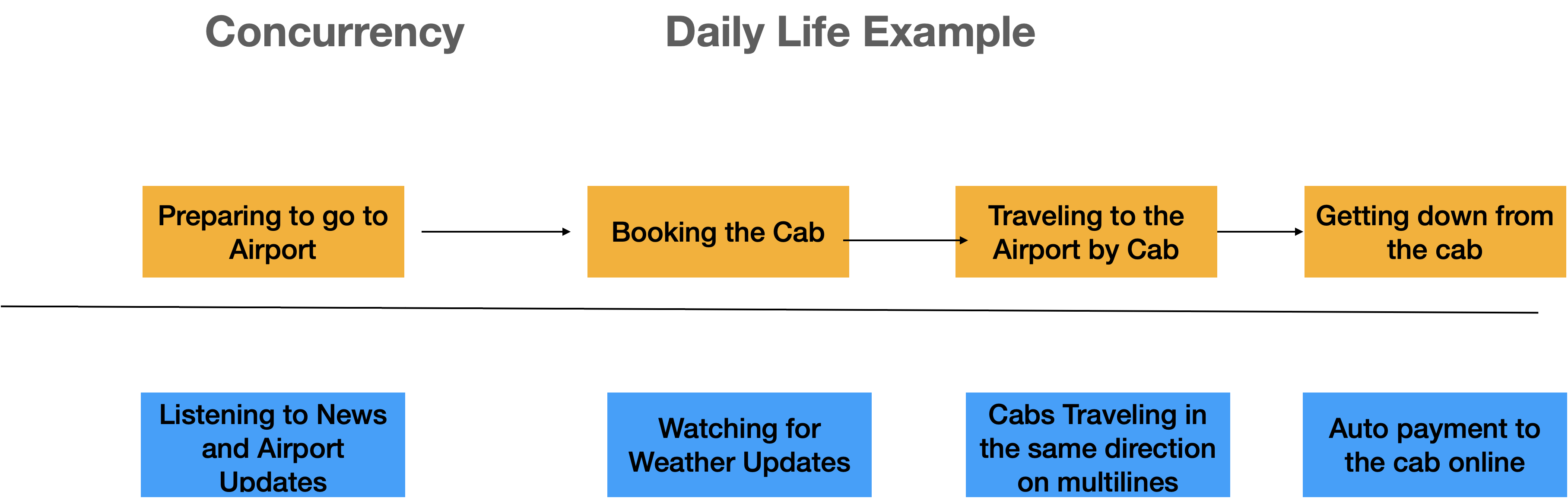
Concurrent Tasks

Let us first look at a layman example. On the road, there will be vehicles going in parallel roads in different directions. If there are two lanes on the road, you can have vehicles concurrently move in the same direction on multiple lanes.



Lay Man Example

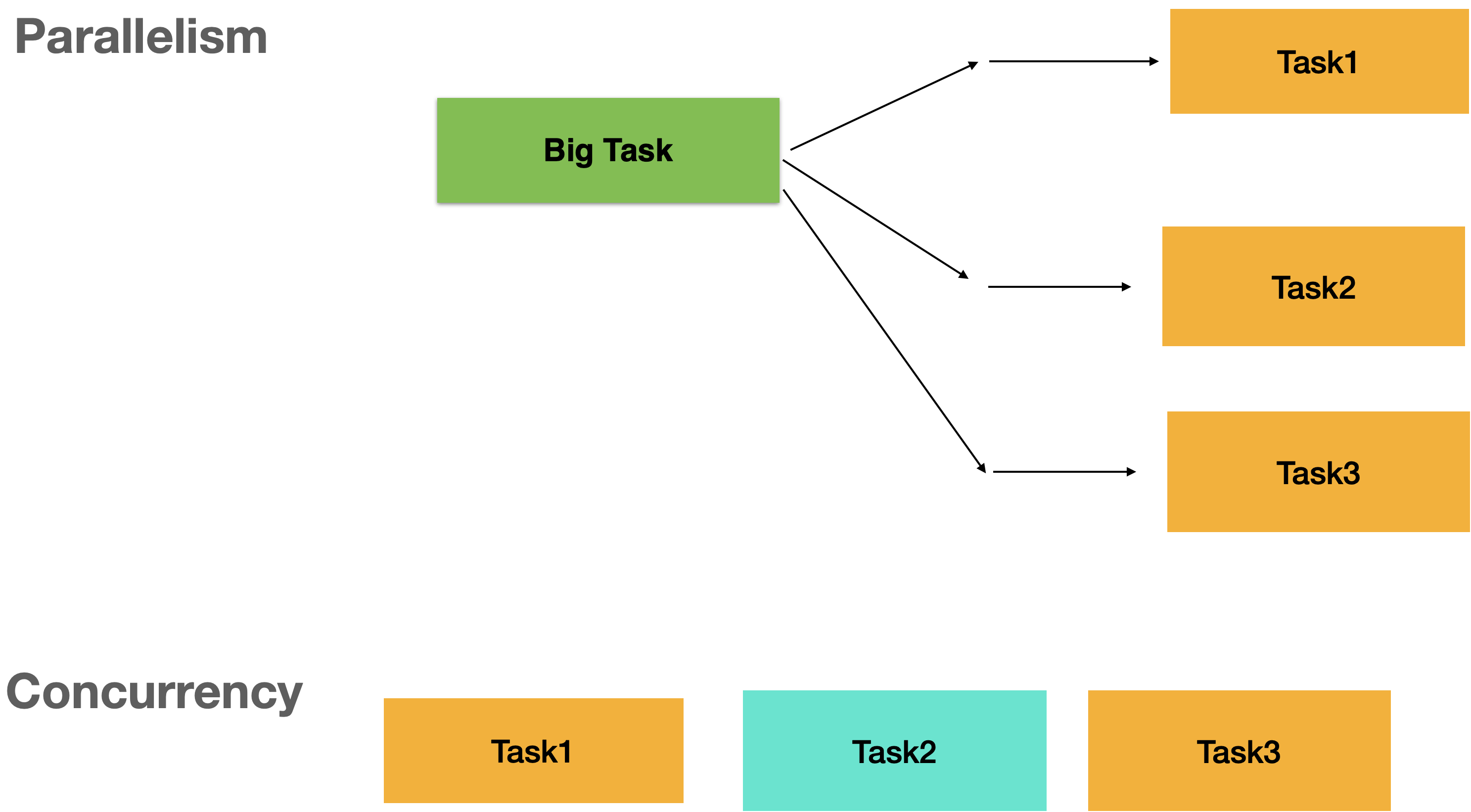
In daily life, you can see the above happening on the road when you are travelling. It is same applied to multi lane airports or train stations with multiple platforms.



Daily life example - Concurrency

Like the road and lanes example, there are instances where concurrency and parallelism principles are applied. Concurrent apps have tasks executed during the same time. These tasks can be related to app loading, app rendering, and media rendering & loading.

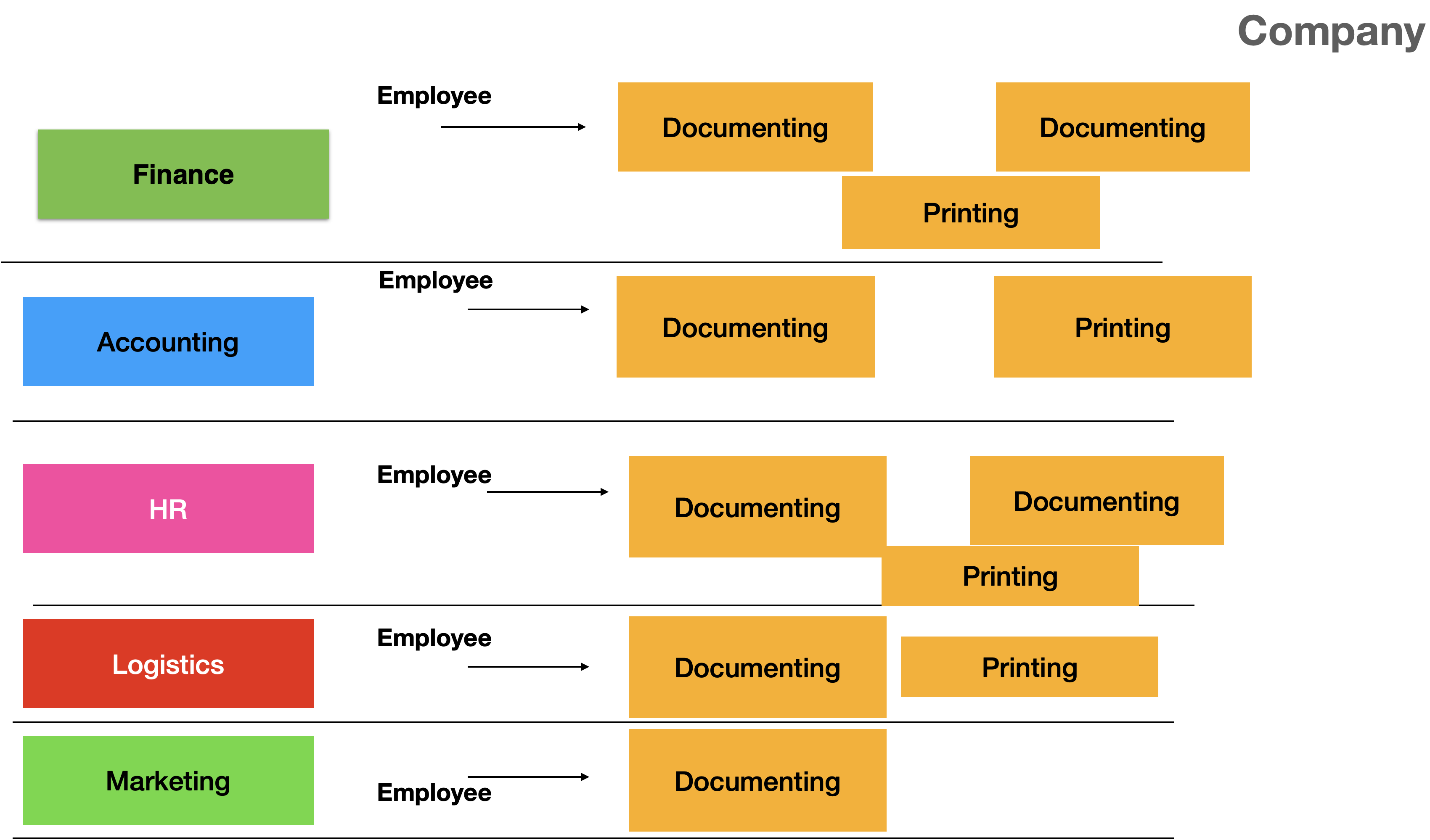
In Go language, concurrency features are available out of box. First let us look at concurrency vs Parallelism.



Parallelism vs Concurrency

You can see this happening in the browser when you click on the website and download an executable at the same time. Rendering, downloading, asynchronous processing of requests, rendering asynchronous responses, and loading of the browser panels happen at the same time. This is a good example for Concurrent applications. Parallelism can be observed in bigdata problems where batches of 1 k are analyzed in a dataset of 1Gigabyte. Multiple threads are spanned to work on 1k each and the overall computation time comes down as they are parallelized.

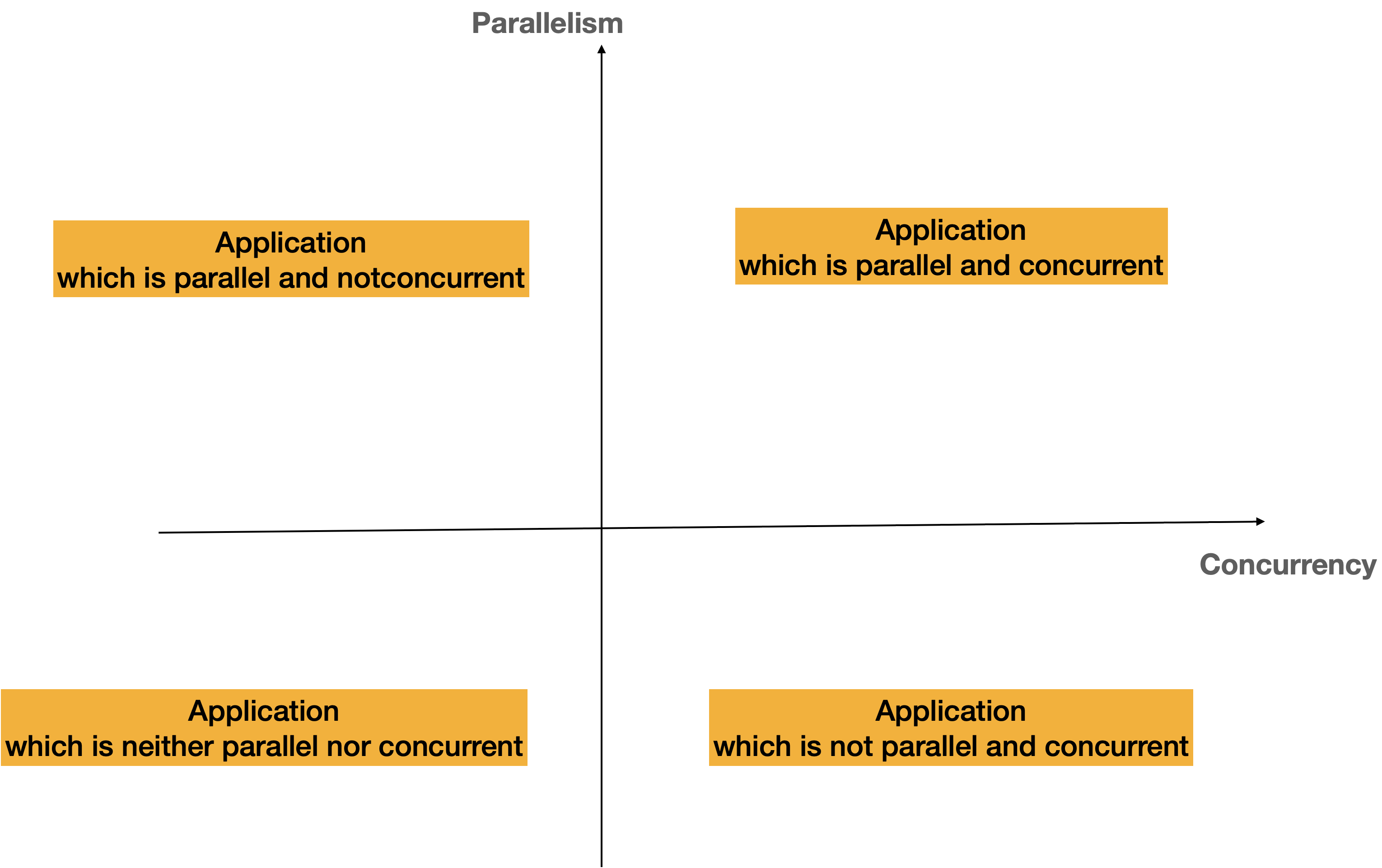
Now let us look at a company where employees are working on their specific departmental tasks.



**Concurrency in a Company**

In an organization, you can concurrent operations happening in various departments like marketing, sales, logistics, accounting, finance, and HR. Many documents are created by the employees and print outs are taken from the printers available. Concurrent tasks of creating documents and parallelly the print outs being generated happens. Each employee focuses on documenting task and stop before taking a printout. The employee can work on another document while printing the document. All these activities happen at the same time.

After looking at different concurrency and parallelism principles, you can understand how we can simulate the same using Golang go routines. We will talk about go routines in detail in the next section. While building an application, you need to start looking at the concurrent and parallel tasks which can be performed by the user. Application can be concurrent and not parallel. It can be parallel and not concurrent. Big data application is a good example. Application can be neither parallel nor concurrent in nature. Workflow based application where tasks are executed in sequence and there is a wait time for completion of the task. There can be applications which can be both concurrent and parallel.



**Assessment Matrix: Parallelization vs Concurrency**

**Note: Concurrent tasks can be in progress whereas parallelized tasks can be in execution during the same time. Concurrency can utilize parallelism principles to ensure multiple tasks progress during the same time. Parallelism is not the only motive for concurrency. Overall processing time comes down as multi cores are used for processing during concurrent tasks execution. Go Lang has built in features for concurrency.**

# goRoutines

Let us recap from Chapter 2 about goRoutines:

*In Go Lang, go routine gets executed with a thread owned by it. Go routines are functions.*

Now let us look at how Go Routines gets executed in a computer. A computer will have CPUs and processors associated with the CPU. An operating system like Unix, Linux, MacOS, and Windows will have capability to spawn processes and execute threads within the process.

A picture containing text, screenshot, font, rectangle

Description automatically generated

GoRoutines

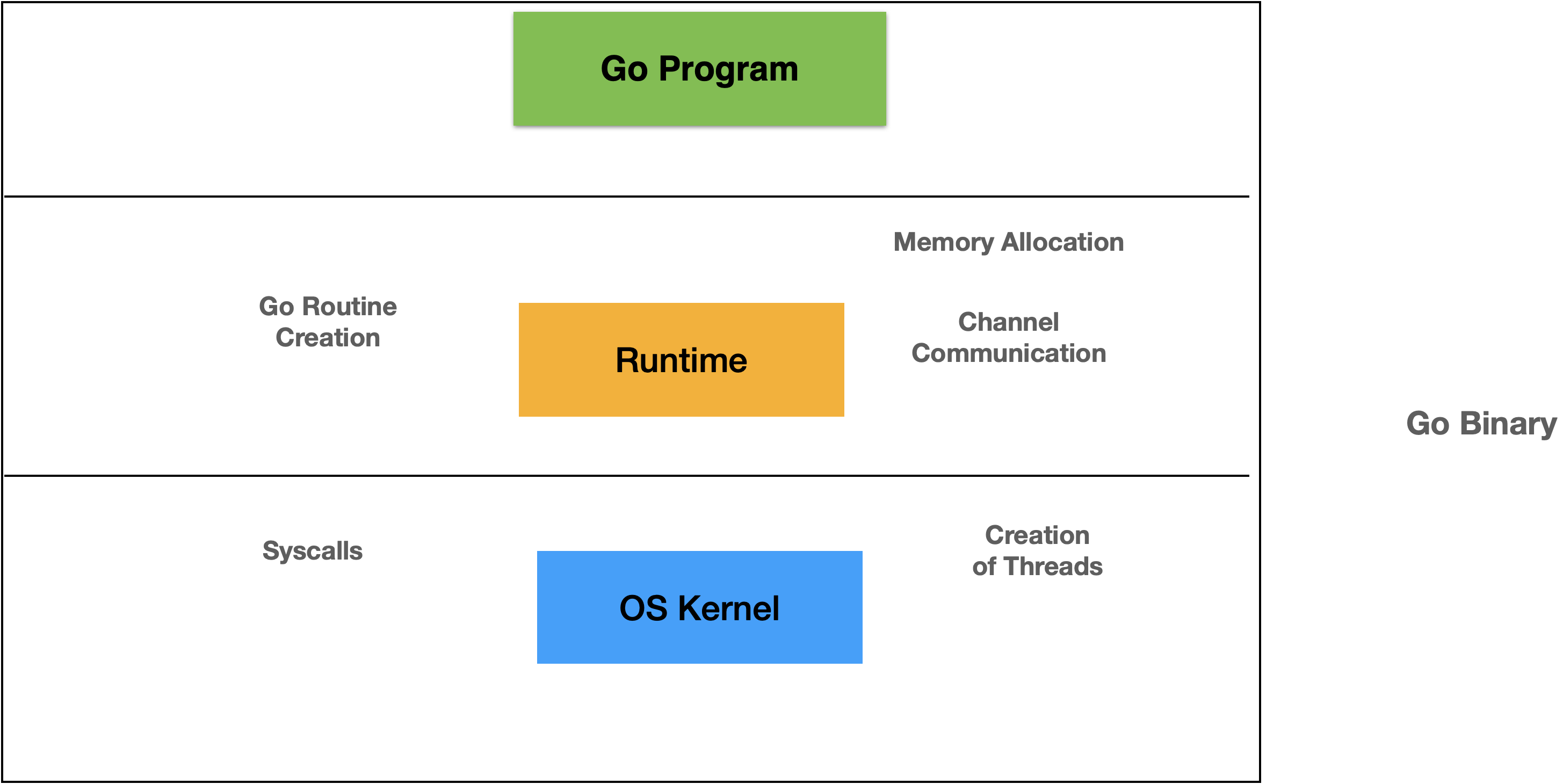
A processor manages different threads in a process allocated to the core. Go routines are associated with threads in an operation system. Now let us look how the queuing of Go Routines when we have multiple cores and CPUs in a computer.

A picture containing text, screenshot, post-it note, rectangle

Description automatically generated

Operating system has a scheduler for assigning the process and threads to a core. CPU1 will manage threads in a process. Scheduler in Go Lang manages the global Execution queue of Go routines. Go Lang scheduler picks the go Routine to a Local Run queue of an operating system thread. This scheduler is based on cooperative model, and it is not preemptive. In the cooperative model, threads can yield for explicit execution. This helps in allocating the other go routine for execution. During the allocation, the context gets switched across the go routines. This is where case scheduler plays an important role. Go Lang Runtime decides the priority of go Routine execution. Runtime takes care of methods invocation, garbage collection, network communication, channel operations, blocking and other events. Context switching happens only for the event of change of go Routine to be executed.

The below diagram shows how Go Lang Program interacts with Runtime and OS kernel.



Go Binary – Runtime and Kernel Interaction

OS Kernel manages the sys calls and creation of the threads. Runtime manages the creation of the go Routines, channel communication and memory allocation. In the groups scenario, go routines group can wait till the other group of go Routines are executed. Sync Package is used to do a sleep before the shift of allocation of goRoutines group.

Now let us look at the sample code for an anonymous goroutine.

**anonymous\_go\_routine.go**

package main

import (

"fmt"

"time"

)

func main() {

fmt.Println("Starting the main function")

go func() {

fmt.Println("Calling the anonymous function")

}()

time.Sleep(1 \* time.Second)

fmt.Println("Leaving the main Function")

}

After verifying the installation of Go compiler, you can compile and run the using the below commands. The commands are shown as below:

go run anonymous\_go\_routine.go

The output will be as shown below:

(base) apples-MacBook-Air:src bhagvan.kommadi$ go run anonymous\_go\_routine.go

Starting the main function

Calling the anonymous function

Leaving the main Function

(base) apples-MacBook-Air:src bhagvan.kommadi$

As shown in the code above, anonymous function does not have the name. In the program above, you can see the start of the main function, anonymous function invocation, and exiting of the main function. Now let us look at an example of a Go Routine.

**go\_routine\_exampe.go**

package main

import (

"fmt"

"time"

)

func show(message string) {

for i := 0; i < 7; i++ {

time.Sleep(1 \* time.Second)

fmt.Println(message)

}

}

func main() {

go show("go routine")

show("running a method")

}

You can see in the code above, a go Routine named show is defined which shows the message. In the main method, example code shows the difference between the execution of goroutine and just the method.

You can compile and run the using the below commands. The commands are shown as below:

go run go\_routine\_example.go

The output will be as shown below:

(base) apples-MacBook-Air:src bhagvan.kommadi$ go run go\_routine\_example.go

0:running a method

0:go routine

1:go routine

1:running a method

2:running a method

2:go routine

3:go routine

3:running a method

4:running a method

4:go routine

5:go routine

5:running a method

6:running a method

6:go routine

(base) apples-MacBook-Air:src bhagvan.kommadi$

In the above code, you can see the difference and sequence of execution of goroutine and just the method in go Lang.

# Channels

Let us recap from chapter 2 about Channels:

*Channels provide the capability or a feature for go routines to share information to another go routine. Channel can send the data to another channel. Channels are bidirectional. Channel can be created by using chan key word*.

A black background with white text

Description automatically generated with low confidence

**Channels**

In the diagram above, you can how channel is used for go Routine communication. Concurent go Routines can communicate using channel for exchange of messages and data. Channel Operator <- is used in the code for different channel operations. In a web application, go Routines can help in improvement of performance by designing and developing concurrent tasks as go Routines.

A black background with white text

Description automatically generated with low confidence

**Channel – Send and Receive Messages**

Go Routines exchange data and information using channels. Channel blocks and waits to receive and transmit the data to the destination go Routine. Channel data is typically shared by iterating in a loop over the data.

Now let us look at an example where go Routines help in improving the performance and efficiency. In the case of big data solutions, jobs or tasks are created over data creation, processing, querying, persistence, and storing of the information. These different tasks communicate through channels as batches of data are processed through this pipeline.



Big data solutions involve handling data tasks like receiving data and sending data from source to the destination. The challenge is handling terabytes of data like transactions, logs, metrics, and batch job updates. Most of the database driven solutions have steps for staging, cleansing, processing, and analysis of the data. Communication is the key for these solutions between the source and the destination. Where the data is getting processed and which step in the process is a very important solution in an enterprise. Business activity monitoring and the analytics solutions focus on identifying the activities related to business data and presenting the status of the pipeline execution.

In enterprises, the challenge is managing the cost vs efficiency for big data solutions. Most of the time developers chose small programs and scripts to handle different steps in the process. Go lang is used not just for the big data solutions but also for small programs. Data pipelines can be written in go lang programs. Each pipeline has input and output data and different steps in the process can be data pipelines. Concurrency model helps in handling breaking the big data in manageable chunks and handling data using data pipelines.

An example for using go routines and channels is a go lang based program for processing customers transactions to update loyalty points for different customers. Loyalty program has gold, silver, and bronze levels. Each program has number of points to be achieved. Go routines and channels are used to process the data in batches.

Error handling and exceptions management is easy in go lang as the transactions which are problematic can moved into separate file. At the end of the execution, you can identify the transactions which are analyzed and others which are causing errors.

Now let us start coding a channel. You can look at the channel example below.

**channel\_example.go**

package main

import "fmt"

func main() {

var channel chan int

fmt.Println(" channel's Value: ", channel)

fmt.Printf("channel's Type: %T ", channel)

channel1 := make(chan int)

fmt.Println("\n channel1's Value: ", channel1)

fmt.Printf("channel1's Type: %T ", channel1)

}

Sample code above shows creation of a channel and its value and type is printed.

You can now compile and run the channel\_example.go using the command below:

go run channel\_example.go

The output will be as shown below:

(base) apples-MacBook-Air:src bhagvan.kommadi$ go run channel\_example.go

channel's Value: <nil>

channel's Type: chan int

channel1's Value: 0xc00005a060

channel1's Type: chan int (base) apples-MacBook-Air:src bhagvan.kommadi$

Now, moving to the next example. We will look at the code for channel messaging- send and receive methods.

**channel\_example.go**

package main

import "fmt"

func exfunc(channel chan int) {

fmt.Println(123 + <-channel)

}

func main() {

fmt.Println("Main method has started")

channel := make(chan int)

go exfunc(channel)

channel <- 23

fmt.Println("Main method has ended")

}

In the above example, channel messaging and sending /receiving messages from the channel to go routine is shown.

You can now compile and run the channel\_messaging.go using the command below:

go run channel\_messaging.go

The output will be as shown below:

(base) apples-MacBook-Air:src bhagvan.kommadi$ go run channel\_messaging.go

Main method has started

146

Main method has ended

(base) apples-MacBook-Air:src bhagvan.kommadi

Now, Let us look at the code for channel messaging- send and receive methods.

**channel\_closing.go**

package main

import "fmt"

func newfunc(channel chan string) {

for v := 0; v < 4; v++ {

channel <- "running the channel"

}

close(channel)

}

func main() {

channel := make(chan string)

go newfunc(channel)

for {

result, message := <-channel

if message == false {

fmt.Println("Closing the channel ", message)

break

}

fmt.Println("Opening the channel ", result, message)

}

}

In the example above, you can see the channel closing operation.

You can now compile and run the channel\_closing.go using the command below:

go run channel\_closing.go

The output will be as shown below:

(base) apples-MacBook-Air:src bhagvan.kommadi$ go run channel\_closing.go

Opening the channel running the channel true

Opening the channel running the channel true

Opening the channel running the channel true

Opening the channel running the channel true

Closing the channel false

(base) apples-MacBook-Air:src bhagvan.kommadi$

## Channel Types

Channel Types can be of multiple directions or only on a single direction. chan T stands for multiple directions or bidirectional channel type. In two directions, channels can transmit and receive messages. chan <-T stands for send direction channel type. Only sending the messages is possible. <-chan T stands for receive direction channel type.

The other types of channels are shown below in the table below:

| **Method** | **Nil Channel Type** | **Non-Nil Channel Type** | **Closed Channel Type** | **Not-Closed Channel Type** |
| --- | --- | --- | --- | --- |
| Close | panic | succeed to close | panic | succeed to close |
| Send Value | block for ever | block or succeed to send | panic | block or succeed to send |
| Receive Value | block for ever | block or succeed to receive | never block | block or succeed to receive |

The different channel types shown above are :

* Nil Channel Type
* Non-Nil Channel Type
* Closed Channel Type
* Not-Closed Channel Type

## Channel Patterns

Now let us look at different patterns in implementing channels.



**BroadCast Pattern – Channel Implementation**

Channels are used to send a single message to different recipients using broadcast pattern.

A single input channel receives the messages from a sender to broadcast to more than one output channels. More than one recipients receive a single and same message from the output channels. Broadcast pattern helps in ensuring concurrent sending of messages to different recipients.

Now let us look at Fan-In Pattern in Go Lang.



Fan-In Pattern in Go Lang

A Fan in pattern is used to combine different channels into one output channel. The idea is to gather the concurrent tasks data into one channel. This helps one consumer to process messages in a combined form.

Now let us look at Fan Out pattern related to channels. The below diagram shows the Fan-Out pattern where data coming from a single input channels from various senders is routed to different recipients on different output channels.



**Fan-Out Pattern in Go Lang**

Fanout pattern helps in sharing the messages across more than one recipient. A single function in go lang can have an array of output channels as parameters. This function can return an input channel which is the multiplexer. Multiplexer helps in sharing data and distributing the messages to more than one output channels.

# Concurrent Apps in Go

Concurrency principles and mechanisms are applied in the console, desktop, and web applications. Goroutines and channels are used to ensure the computation happens concurrently. The performance of the application increases when the goroutines and channels are used. Performance tools and profilers are used to improve the application response. Browser applications are analyzed using the developer tools for capturing the performance times.

Now let us look at building a concurrency application. Web Links can be loaded to analyze the metadata of the website. We can create the RSS feed for the website looking at various topics and content covered. RSS feeds can be updated frequently based on the website updates. This application can be built using Go Lang concurrency principles.

You can have web links configured based on your preferences and topics selected. Web links can be scanned to generate RSS feeds topic wise. The application will have capability to add or remove the web links from the topics. New topics can be added for the application with web links.

RSS feeds can be loaded into the application based on the topic. Feeds can be stored in a No SQL database. This application can be on the cloud serving as a software as a service.

Users can register using social authentication – google/msn/yahoo/twitter. Users can create a profile and preferences. Feeds can be stored in the No SQL database and can be categorized with topics. They can be tagged with keywords and made searchable for quick access. Cache mechanisms can be used to improve the retrieval time of the master data or user preferences and interests.

There is another way to speed up the performance by having in memory user specific feeds for a category selected loaded. User can browse them quickly. Scalability is an issue for this pattern to be implemented. Concurrent users can browse the feeds quickly based on the category selection. In memory, multiple categories and feeds specific to that category can be loaded.

User profiles can be modified, and demographics of the users are stored in the NoSQL database. This helps in recommending the feeds for the users. Users can access correlated feeds read by other users. Tags can be selected for feeds and users can add new tags to the existing tagged feeds.

The below diagram shows the relationship between user profile and demographics of the users in the system. Typical demographics of the user gathered are:

* Age
* Gender
* Interests
* Preferences
* Job/Profession
* Designation
* …

**User Profile**

Recommendation system can be built as an add on to the system to provide new feeds and tags for user. User will be eager to come back to the application. This is because new content will be ready processed as recommendations. This can be similar to adsense where context of the user can be used based on the reader’s browsing history of the feeds.

User satisfaction is dependent on the user profile and other factors mentioned below:

* User Profile
* Content
* Context
* Interests
* ..



**Reader Satisfaction Triangle**

The above diagram shows the triangular forces driving the user satisfaction based on user profile, context, and content. The recommended feeds are evaluated for performance based on the following factors:

* Click through rate
* Stay Time
* Upvotes
* Comments
* Reposts



**Recommendation System Metrics**

The above diagram shows the important metrics to measure the performance and efficiency of the recommendation system.

Content provided to the user can be of the following :

* Textual information
* Images/Posters
* Video
* A web Link.
* Document

This content can have multiple labels. Different labels for the user content are shown in the diagram below.



**User Label Analysis**

As shown in the diagram user Interest specific labels can be:

* Category
* Topic
* Keyword
* Source
* Community
* Vertical

User Profile specific labels can be:

* Gender
* Age
* Location
* Preferences
* Interests
* Favorites
* Bookmarks

User behavior specific labels can be:

* Browsing Time
* Reading Time
* Staying Time
* Click through analysis

Label generation can be done by using the following methods:

* Noise Cancelling by filtering the content
* Penalizing the trends on feeds
* Using Time Decaying and browsing history
* Using global bias and click rate
* Likes/DisLikes/Unsubscribes

Now let us start building this application. First step is to look at accessing the links concurrently.

**concurrency\_app.go**

package main

import (

"fmt"

"net/http"

)

func main() {

channel := make(chan string)

links := []string{

"https://www.golangbot.com",

"https://www.golang.org",

"https://www.changelog.com/gotime",

"http://qvault.io/",

"https://golang.ch/",

"https://gosamples.dev/",

"https://golangcode.com",

"https://appliedgo.net",

}

for \_, link := range links {

go accessLink(link, channel)

}

for message := range channel {

fmt.Println(message)

}

}

func accessLink(link string, channel chan string) {

if \_, error := http.Get(link); error != nil {

channel <- link + " is not accessible"

} else {

channel <- link + " is accessible."

}

}

You can now compile and run the concurrency\_app.go using the command below

go run concurrency\_app.go

The output will be as shown below:

(base) apples-MacBook-Air:app bhagvan.kommadi$ go run concurrency\_app.go

https://www.changelog.com/gotime is accessible

https://gosamples.dev/ is accessible

https://appliedgo.net is accessible

https://golang.ch/ is accessible

https://www.golang.org is accessible

https://golangcode.com is accessible

http://qvault.io/ is accessible

https://www.golangbot.com is accessible

^Csignal: interrupt

(base) apples-MacBook-Air:app bhagvan.kommadi$

# Conclusion

In this chapter we have covered topics related to Concurrency and parallelism. Examples were presented to demonstrate how channels, and go Routines can be used in real life. Summary of the chapter is mentioned below.

* Concurrency is the superset of parallelism. In real life, concurrency is observed in various scenarios.
* Go routines can be used for processing data by having a function executed by the thread.
* Channels are used for communication between multiple go routines.
* Channel Types can be single direction and multiple direction types.
* Channel patterns are broadcast, Fan-in and Fan-Out patterns.