**The Go memory model**  (<https://golang.org/ref/mem>)

**The Go Playground https://play.golang.org/**

**Intro Example**

package main

import "fmt"

type API struct { a int }

func (api API ) passByReference() { api.a = 5 }

func (api \*API ) passByValue() { api.a = 5 }

func main() {

var api= API{0}

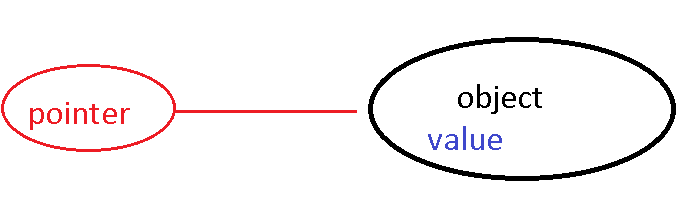
api.passByReference()

fmt.Println(api)

api.passByValue()

fmt.Println(api)

}



The memory model defines an interface between a program and any hardware or software that may transform that program . It answers questions such as: Is there enough synchronization to ensure a thread's write will occur before another's read? Can two threads write to adjacent fields in a memory location at the same time? Must the final value of a location always be one of those written to it?

Programming languages generally already provide *synchronization* mechanisms, such as locks, or possibly transactional memory, for limiting simultaneous access to variables by different threads

**Terminology**

pointers, new, make, arrays, slice, map, channels, locks, synchronization, threads, goroutines, data race

Go is a garbage collected language, so you don't have explicit control over the allocation and deallocation of memory. You don't need to explicitly deallocate memory however you do need to understand how memory is allocated and how to optimise the possibility for deallocation of memory

Go's arrays are values. This means that when you assign or pass around an array value you will make a copy of its contents.

To avoid the copy you could pass a pointer to the array, but then that's a pointer to an array, not an array.

A pointer to an array is a *slice*.

[**http://blog.golang.org/go-slices-usage-and-internals**](http://blog.golang.org/go-slices-usage-and-internals)

Go has two allocation key words new and make.

They follow simple rules .

* new a built-in function that allocates memory, but unlike its namesakes in some other languages it does not initialize the memory, it only zeros it. That is, new(T) allocates zeroed storage for a new item of type T and returns its address, a value of type \*T. In Go terminology, it returns a pointer to a newly allocated zero value of type T.
* make make(T, args) serves a purpose different from new(T). It creates slices, maps, and channels only, and it returns an initialized (not zeroed) value of type T (not \*T). These types represent references to data structures that must be initialized before use.
* A slice is a three-item descriptor containing a pointer to the data (inside an array), the length, and the capacity, and until those items are initialized, the slice is nil.

**http://www.goinggo.net/2013/07/understanding-pointers-and-memory.html**

**Concurrency is not Parallelism.**

Parallelism is when two or more threads are executing code simultaneously against different processors.

Memory operations conflict if they simultaneously access the same memory location and at least one is a write

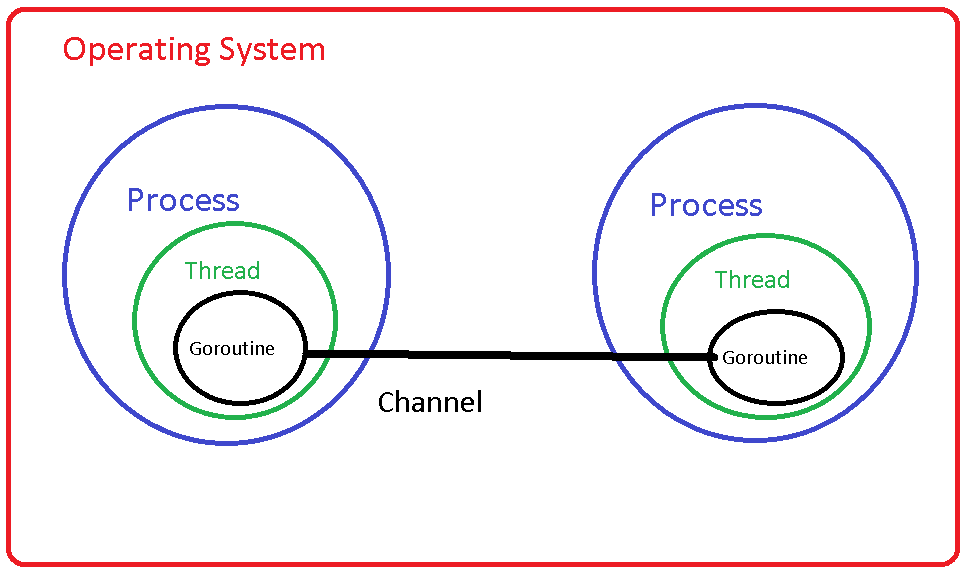
A data race occurs when 2 or more threads perform conflicting operations

We perform synchronisation to ensure data-race-free programs

Downside synchronisation is expensive and can impact on performance

The extent to which synchronisation is supported in a memory model we speak of the “strength” of the model

Go implements the “happens before rule” <https://golang.org/ref/mem> only in a single thread of execution, thus Go supports a “**weak memory model**” this means we must us explicit synchronisation to achieve data-race-free programs



### **Processes**

In the beginning, computers ran one job at a time in a batch processing model. In the 60’s a desire for more interactive forms of computing lead to the development of multiprocessing, or time sharing, operating systems. By the 70’s this idea was well established for network servers, ftp, telnet, rlogin, and later Tim Burners-Lee’s [CERN httpd](http://www.w3.org/Daemon/), handled each incoming network connections by forking a child process.

In a time-sharing system, the operating systems maintains the illusion of concurrency by rapidly switching the attention of the CPU between active processes by recording the state of the current process, then restoring the state of another. This is called context switching.

### **Threads**

This lead to the development of threads, which are conceptually the same as processes, but share the same memory space. As threads share address space, they are lighter to schedule than processes, so are faster to create and faster to switch between.

Threads still have an expensive context switch cost; a lot of state must be retained. Goroutines take the idea of threads a step further.

### **Goroutines**

Rather than relying on the kernel to manage their time sharing, goroutines are cooperatively scheduled. The switch between goroutines only happens at well defined points, when an explicit call is made to the Go runtime scheduler. The major points where a goroutine will yield to the scheduler include:

* Channel send and receive operations, if those operations would block.
* The go statement, although there is no guarantee that new goroutine will be scheduled immediately.
* Blocking syscalls like file and network operations.
* After being stopped for a garbage collection cycle.

In other words, places where the goroutine cannot continue until it has more data, or more space to put data.

Many goroutines are multiplexed onto a single operating system thread by the Go runtime. This makes goroutines cheap to create and cheap to switch between. Tens of thousands of goroutines in a single process are the norm, hundreds of thousands are not unexpected.