**Goroutines**

(<https://www.golang-book.com/books/intro/10>)

### **Goroutines**

Rather than relying on the kernel to manage their activity, goroutines are cooperatively scheduled by a runtime artifact the Go runtime scheduler. The scheduler manages:

* 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.

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

Demonstrate the issues with goroutines

package main

var a string

var done bool

func setup() {

a = "hello, world"

done = true

}

func main() {

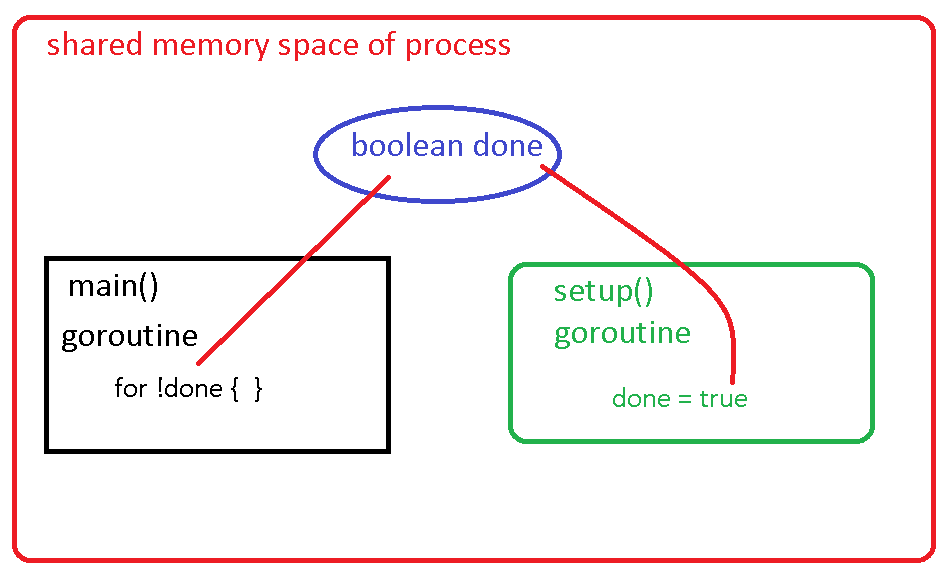
go setup() // kicks off a thread for setup()

for !done {

}

print(a)

}



**Traditional Explicit Synchronisation (Locks)**

**Using explicit synchronization this code exits**

<https://en.wikipedia.org/wiki/Mutual_exclusion>

https://golang.org/pkg/sync/

Types of Go Locks

sync.Mutex

sync.RWMutex

package main

import "sync"

var l sync.Mutex

var a string

var done bool

func setup() {

a = "hello, world"

done = true

l.Unlock()

}

func main() {

l.Lock()

go setup()

l.Lock()

for !done {

}

print(a)

}

**Synchronised Set**

http://blog.golang.org/defer-panic-and-recover

**package main**

**import (**

**"fmt"**

**"sync"**

**)**

**type Set struct {**

**m map[string]bool**

**sync.RWMutex**

**}**

**func New() \*Set {**

**return &Set{**

**m: make(map[string]bool),**

**}**

**}**

**func main() {**

**// Initialize our Set**

**s := New()**

**// Add example items**

**s.Add("item1")**

**s.Add("item1") // duplicate item**

**s.Add("item2")**

**fmt.Printf("%d items\n", s.Len())**

**// Clear all items**

**s.Clear()**

**if s.IsEmpty() {**

**fmt.Printf("0 items\n")**

**}**

**s.Add("item2")**

**s.Add("item3")**

**s.Add("item4")**

**// Check for existence**

**if s.Has("item2") {**

**fmt.Println("item2 does exist")**

**}**

**// Remove some of our items**

**s.Remove("item2")**

**s.Remove("item4")**

**fmt.Println("list of all items:", s.List())**

**}**

**// Add add**

**func (s \*Set) Add(item string) {**

**s.Lock()**

**defer s.Unlock()**

**s.m[item] = true**

**}**

**// Remove deletes the specified item from the map**

**func (s \*Set) Remove(item string) {**

**s.Lock()**

**defer s.Unlock()**

**delete(s.m, item)**

**}**

**// Has looks for the existence of an item**

**func (s \*Set) Has(item string) bool {**

**s.RLock()**

**defer s.RUnlock()**

**\_, ok := s.m[item]**

**return ok**

**}**

**// Len returns the number of items in a set.**

**func (s \*Set) Len() int {**

**return len(s.List())**

**}**

**// Clear removes all items from the set**

**func (s \*Set) Clear() {**

**s.Lock()**

**defer s.Unlock()**

**s.m = make(map[string]bool)**

**}**

**// IsEmpty checks for emptiness**

**func (s \*Set) IsEmpty() bool {**

**if s.Len() == 0 {**

**return true**

**}**

**return false**

**}**

**// Set returns a slice of all items**

**func (s \*Set) List() []string {**

**s.RLock()**

**defer s.RUnlock()**

**list := make([]string, 0)**

**for item := range s.m {**

**list = append(list, item)**

**}**

**return list**

**}**