



Goroutines and Channels

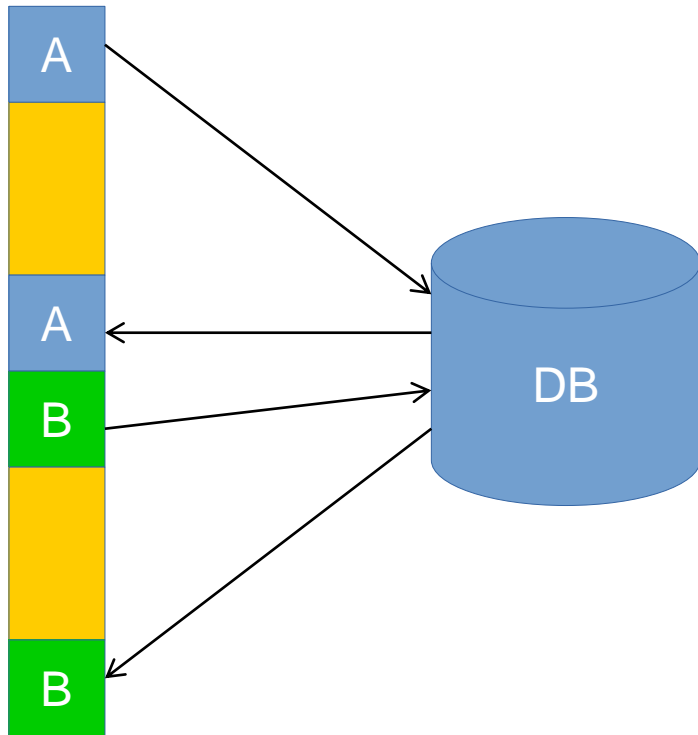
Goroutines, channels, parallel loops, select, goroutines cancelation

Where to Find The Code and Materials?

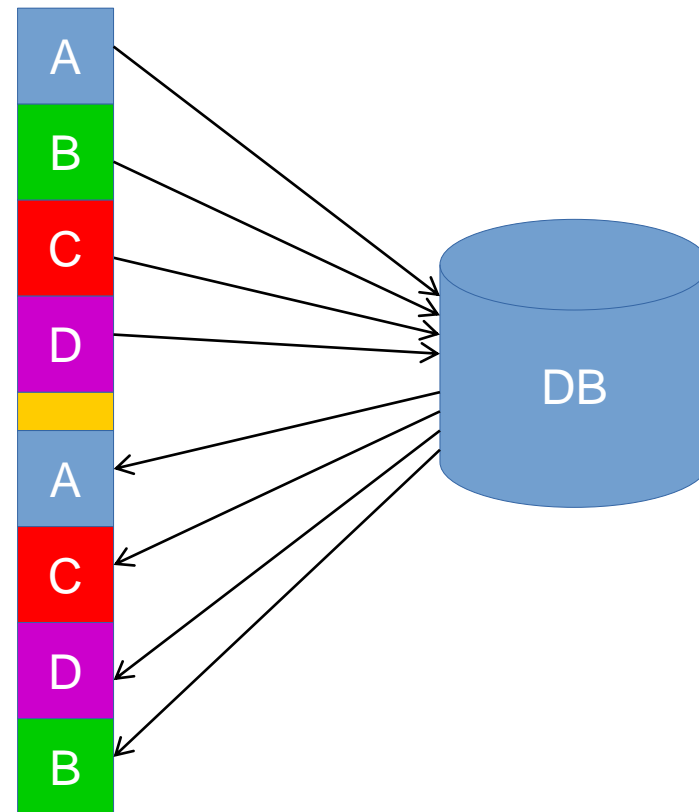
<https://github.com/iproduct/coursegopro>

Synchronous vs. Asynchronous IO

Synchronous



Asynchronous



Blocking vs. Non-blocking

- **Blocking** concurrency – uses **Mutual Exclusion** primitives (aka **Locks**) to prevent threads from simultaneously accessing/modifying the same resource
- **Non-blocking** concurrency does not make use of locks.
- One of the most advantageous feature of **non-blocking** vs. **blocking** is that, **threads does not have to be suspended/waken up by the OS**. Such overhead can amount to **1ms** to a **few 10ms**, so removing this can be a big performance gain. In java for example, it also means that you can choose to use **non-fair locking**, which can have much more system throughput than **fair-locking**.

Non-blocking Concurrency

- In [computer science](#), an [algorithm](#) is called **non-blocking** if failure or [suspension](#) of any [thread](#) cannot cause failure or suspension of another thread;^[1] for some operations, these algorithms provide a useful alternative to traditional [blocking implementations](#). A non-blocking algorithm is **lock-free** if there is guaranteed system-wide [progress](#), and **wait-free** if there is also guaranteed per-thread progress. "Non-blocking" was used as a synonym for "lock-free" in the literature until the introduction of obstruction-freedom in 2003.
- It has been shown that widely available atomic *conditional* primitives, [CAS](#) and [LL/SC](#), cannot provide starvation-free implementations of many common data structures without memory costs growing linearly in the number of threads.

[Wikipedia]

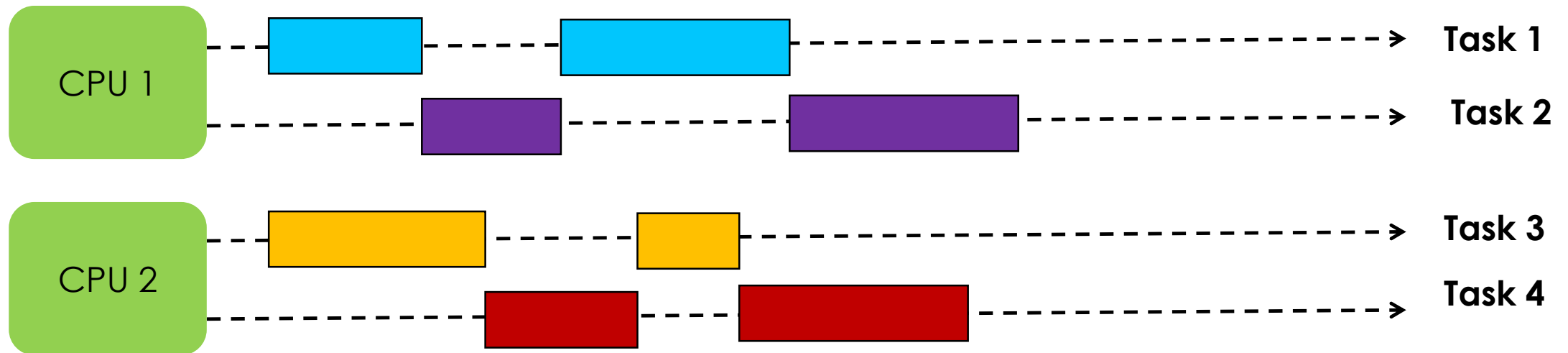
Concurrency vs. Parallelism

Question:

*What is the difference between
concurrency and **parallelism**?*

Concurrency vs. Parallelism

- Concurrency refers to how a single CPU can make progress on multiple tasks seemingly at the same time (AKA concurrently).
- Parallelism allows an application to parallelize the execution of a single task - typically by splitting the task up into subtasks which can be completed in parallel.



Scalability Problem

- **Scalability** is the ability of a program to handle growing workloads.
- One way in which programs can scale is **parallelism**: if we want to process a large chunk of data, we describe its **processing as a sequence of transforms** on a **stream**, and by setting it to parallel we ask **multiple processing cores** to process the parts of the task simultaneously.
- The problem is that the **processes** and **threads**, the **OS supported units of concurrency**, cannot match the scale of the application domain's **natural units of concurrency** — a **session**, an **HTTP request**, or a **database transactional operation**.
- A server can handle upward of a **million concurrent open sockets**, yet the operating system cannot efficiently handle more than a **few thousand active threads**. So it becomes a **mapping problem - M:N**

Why are OS Threads Heavy?

- **Universal** – represent all languages and types of workloads
- Can be **suspended and resumed** - this requires preserving its **state**, which includes the **instruction pointer**, as well as all of the **local computation data**, stored on the **stack**.
- The **stack should be quite large**, because we can not assume constraints in advance.
- Because the OS kernel must schedule **all types of threads** that **behave very differently** in terms of processing and blocking — some serving HTTP requests, others playing videos => its **scheduler** must be all-encompassing, and not optimized.

Solutions To Thread Scalability Problem

- Because threads are costly to create, we **pool** them -> but we must pay the price: **leaking thread-local data** and a **complex cancellation protocol**.
- Thread pooling is **coarse grained** – not enough threads for all tasks.
- So instead **of blocking the thread**, the task should **return the thread to the pool** while it is waiting for some external event, such as a response from a database or a service, or any other activity that would block it.
- The task is **no longer bound to a single thread** for its entire execution.
- Proliferation of **asynchronous APIs**, from **Noide.js** to **NIO in Java**, to the many **“reactive” libraries** (**Reactive Extensions - Rx**, etc.) => intrusive, all-encompassing frameworks, even basic control flow, like loops and try/catch, need to be reconstructed in **“reactive” DSLs**, supporting classes with hundreds of methods.

What Color is Your Function?

<http://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/>

- Synchronous functions return values, async ones do not and instead invoke callbacks.
- Synchronous functions give their result as a return value, async functions give it by invoking a callback you pass to it.
- You can't call an async function from a synchronous one because you won't be able to determine the result until the async one completes later.
- Async functions don't compose in expressions because of the callbacks, have different error-handling.
- Node's whole idea is that the core libs are all asynchronous. (Though they did dial that back and start adding __Sync() versions of a lot functions.)

Async/Await in JS (and some other languages)

- Cooperative scheduling points are marked explicitly with `await` => `scalable synchronous code` – but we mark it as `async` – a bit of confusing!
- Solves the context issue by introducing a new kind of context that is `like thread` but is `incompatible with threads` – one `blocks` and the other returns some sort of `Promise` - you can not easily mix sync and async code.

Right-Sized Concurrency

- If we could make threads lighter, we could have more of them, and can use them as intended:
 1. to directly represent domain units of concurrency;
 2. by virtualizing scarce computational resources;
 3. hiding the complexity of managing those resources.
- Example: Goroutines, Erlang
- creating and blocking goroutines is cheap
- Managed by the Golang runtime, and scheduled cooperatively unlike the existing threads of OS.

How Are Goroutines Better?

- The goroutines make use of a **stack**, so it can **represent execution state more compactly**.
- Control over execution by optimized **scheduler**;
- **Millions of goroutines** => **every unit of concurrency** in the application domain can be represented by its own **goroutine**
- Just spawn a new goroutine, one per task.
- **Example: HTTP request** - a new **goroutine is already spawned** to handle it, but now, in the course of handling the request, you want to simultaneously **query a database**, and issue **outgoing requests** to three other services? No problem: **spawn more goroutines**.

How Are Goroutines Better?

- You need to **wait for something to happen** without wasting precious resources – forget about callbacks or reactive stream chaining – **just block!**
- Write straightforward, boring code.
- Goroutines preserve all the benefits threads give us are preserved by : **control flow, exception context**; only the **runtime cost in footprint and performance is gone!**

Welcome Goroutines!

Goroutines and channels



A Long Computation ...

Let's have an example computation:

|

```
func main() {  
    compute("compute!")  
}  
  
func compute(msg string) {  
    for i := 0; ; i++ {  
        fmt.Println(msg, i)  
        time.Sleep(1 * time.Second)  
    }  
}
```

Let's make it less boring ...

Let's introduce some randomness:

```
func main() {  
    compute2("compute!")  
}  
func compute2(msg string) {  
    for i := 0; ; i++ {  
        fmt.Println(msg, i)  
        time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond)  
    }  
}
```

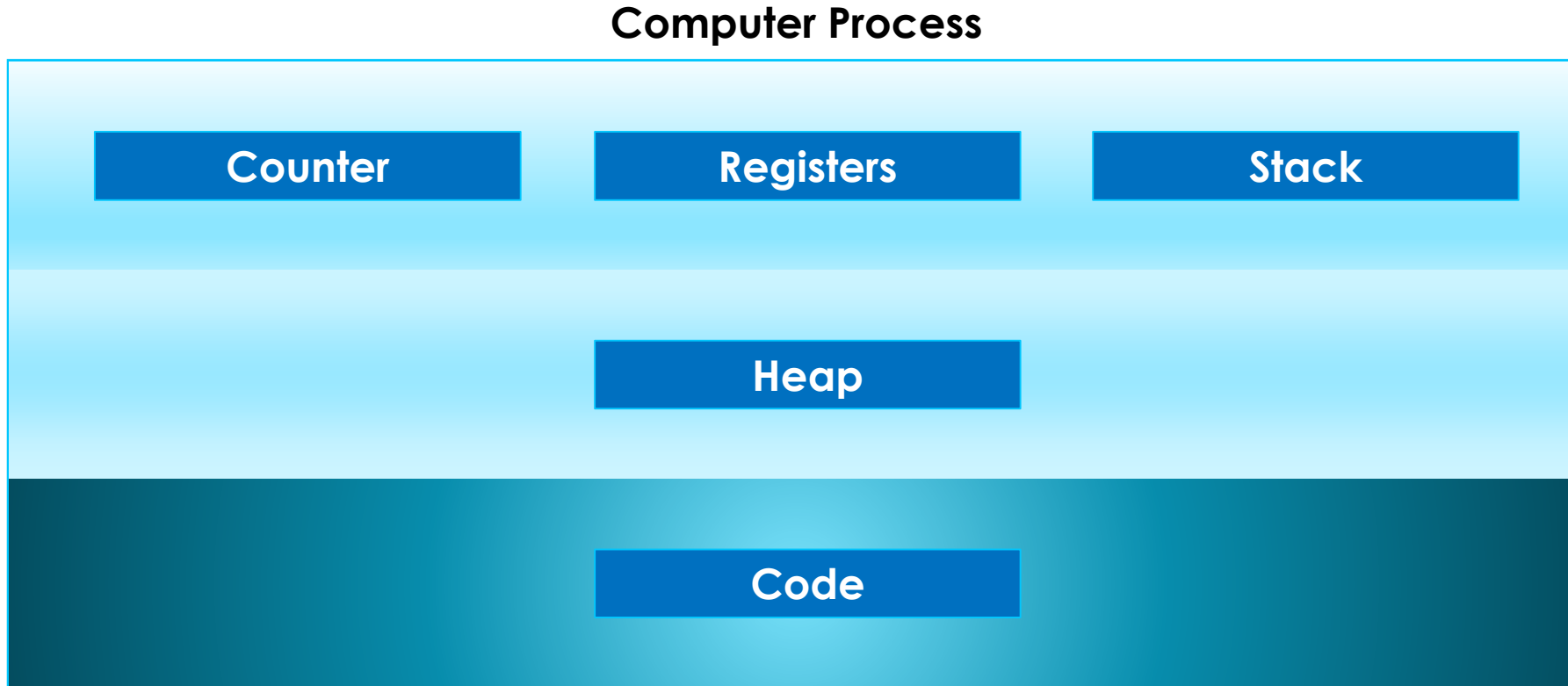
But why should we wait it?

Let's compute in separate goroutine:

```
func main() {  
    go compute3("compute!")  
    time.Sleep(5 * time.Second)  
}  
func compute3(msg string) {  
    rand.Seed(time.Now().UnixNano())  
    for i := 0; ; i++ {  
        fmt.Println(msg, i)  
        time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond)  
    }  
}
```

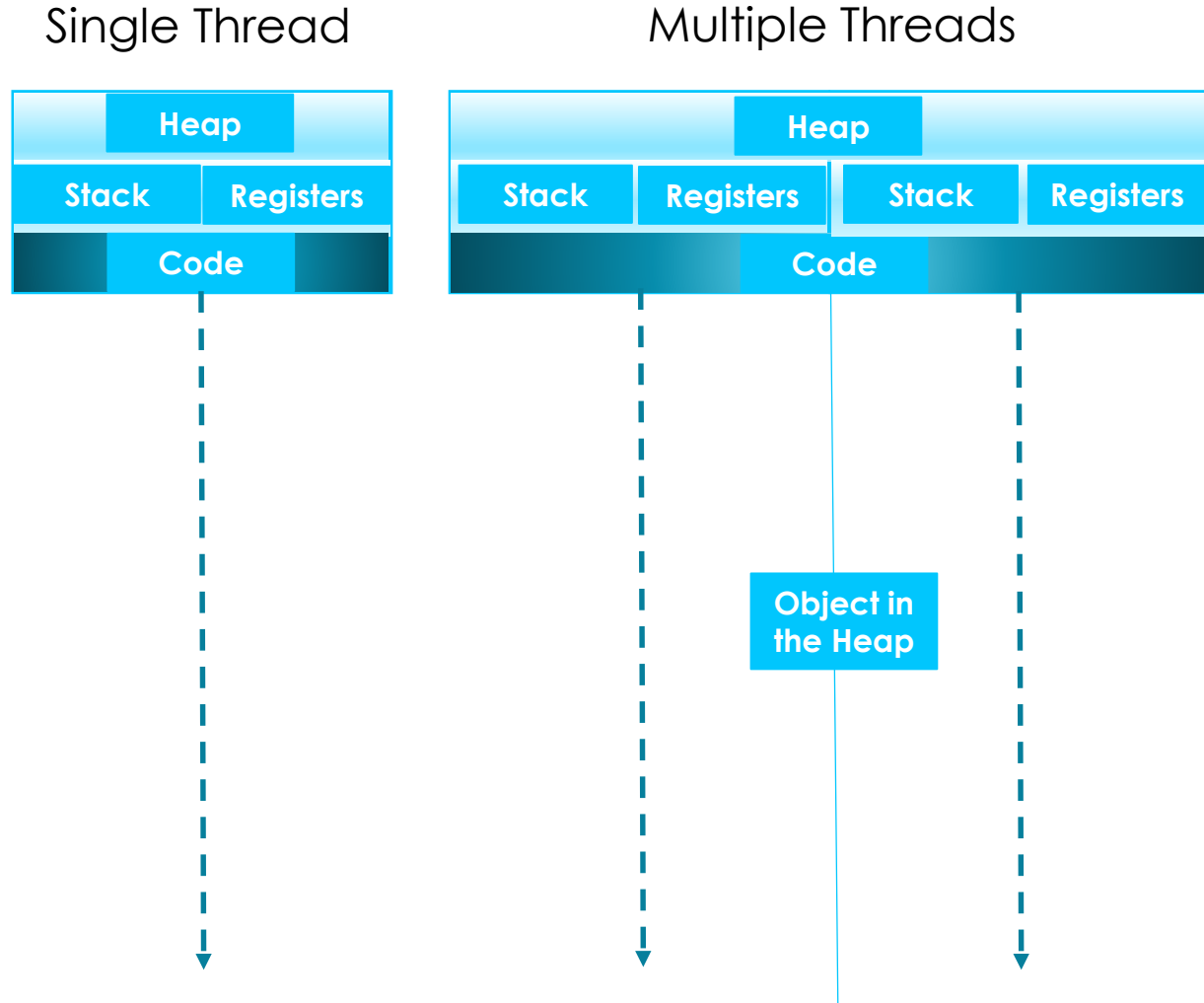
When main goroutine completes all other goroutines stop.

Concurrency Approaches: Processes



Threads

- There can be many threads in the same process
- The threads can access the shared memory
- This means that the global objects can be accessed by all threads
- Provided by the OS
- Cheaper to create than processes
- Some languages expose them directly other hide them behind a **level of abstraction**



Goroutines

- They are executed independently from the main function
- Can be hundreds of thousands of them – the initial stack can be as low as 2KB
- The goroutines stack can grow dynamically as needed
- In Golang there is a smart scheduler that can map goroutines to OS threads
- Goroutines follow the idea of [Communicating sequential processes](#) of Hoare

But why should we bother: concurrency problems

If we access the same memory from two threads/ goroutines, than we have a race! Lets see this pseudo-code:

```
int i = 0
```

```
thread1 { i++ }
```

```
thread2 { i++ }
```

```
wait { thread1 } { thread2 }
```

```
print i
```

What will be the result of the computation?

Critical Sections

- We provide **Mutual Exclusion** between different threads accessing the same resource concurrently
- There are many ways to implement **Mutual Exclusion**
- In Golang there are **sync.Mutex**, **sync.RWMutex**, **atomic operations**, **semaphores**, **error groups** (structured concurrency), **concurrent hash maps**, etc.
- And the message passing using **Channels** of course :)

Share by communicating vs. Communicate by sharing

We can use `Mutex / Atomic` primitives for mutual exclusion between `goroutines` as in other languages, but in most cases it happens to be simpler and more obvious to handle data to other goroutines using `channels`.

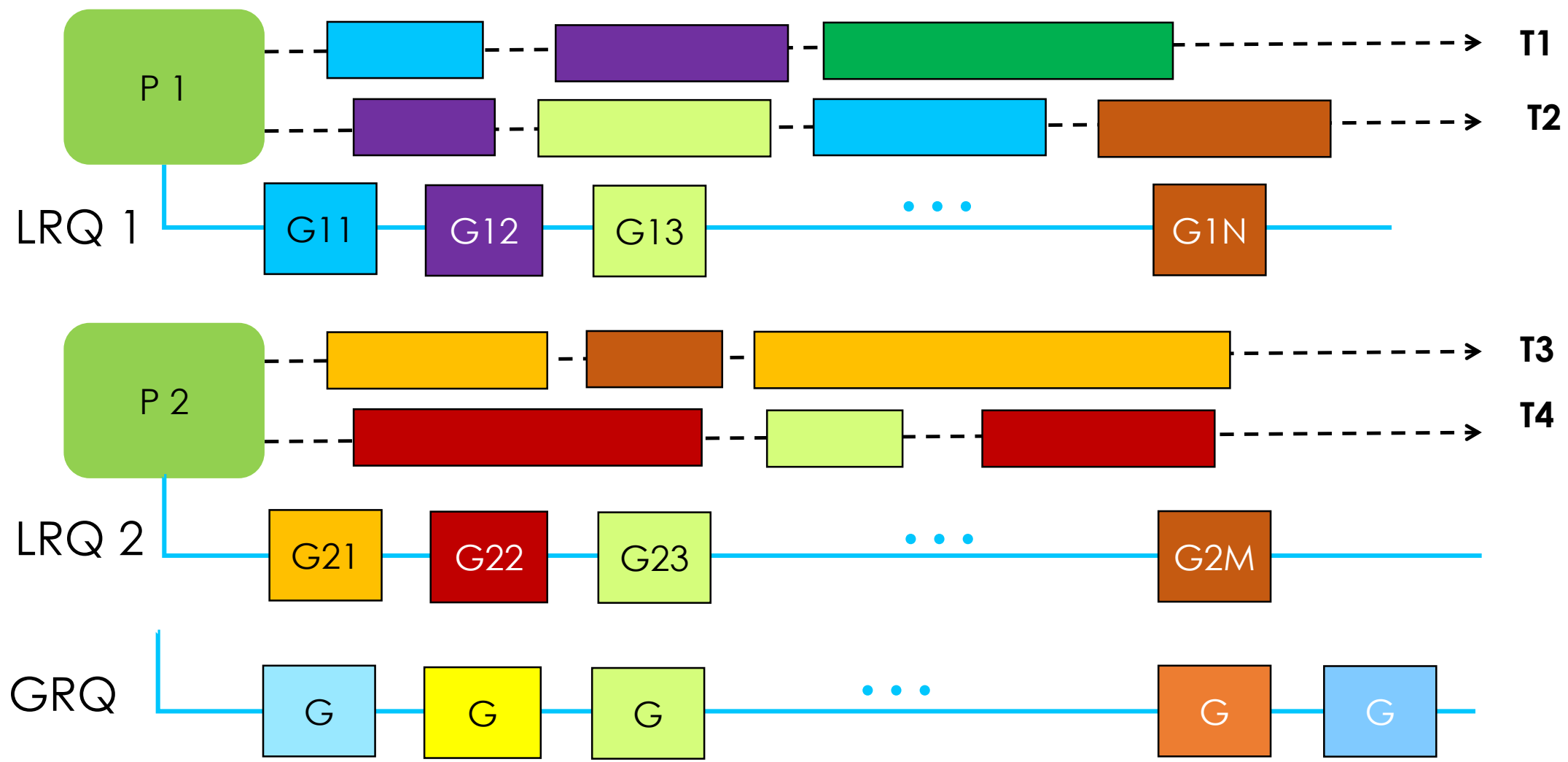
Goroutines Scheduling

GRQ – Global Run Queue LRQ – Local Run Queue

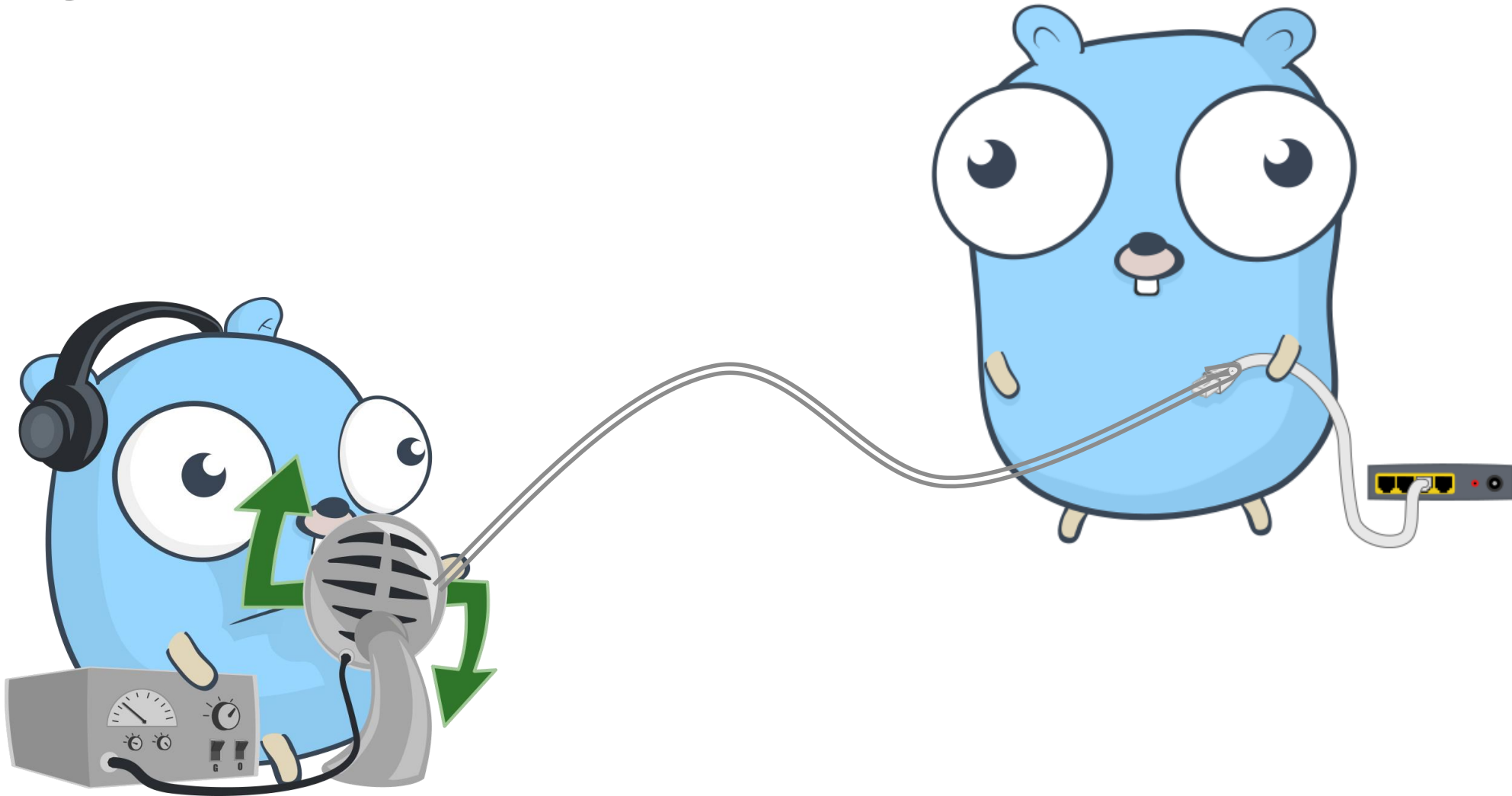
P – Processor

T – Thread

G – Goroutine



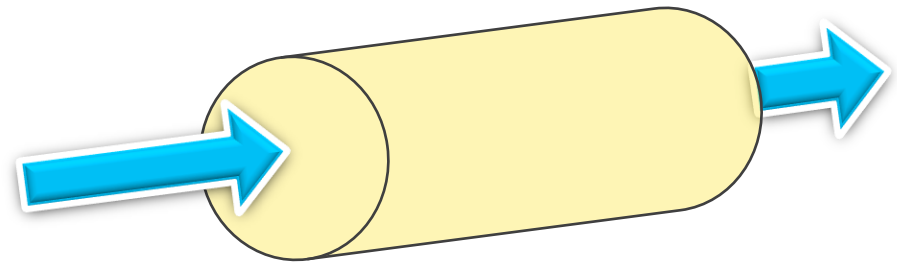
Channels



Channels

- Channels are Golang provided type used for communication between goroutines
- Like a pipeline in which the water can flow – but instead of water there are messages that are sent and received from different ends of the channel
- There is a special language support for channels in Go
- The channels in Go are typed – you should provide the type of messages that will be sent and received using this channel
- Can be created using **make**

```
stringChannel := make(chan string)
```



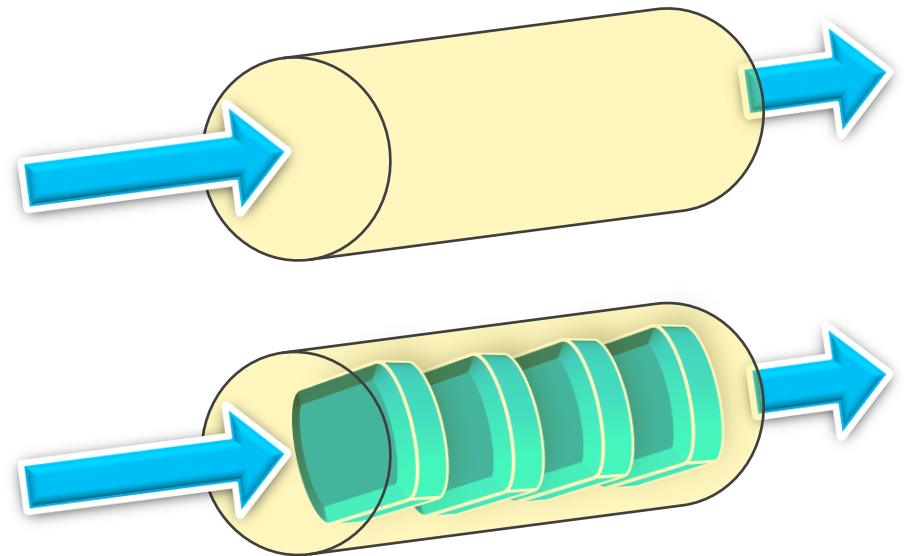
Types of Channels

- Channels can be buffered and non-buffered. By default they are non-buffered:
- Non - buffered channel

```
stringChannel := make(chan string)
```

- Buffered channel

```
ch := make(chan string, 4)
```



IO using channels

- You can send to a channel or receive from channel:

```
ch <- "hello"  
read := <-ch
```

- Sending and receiving operations are done using the `<-` operator
`chan <- someValue` - sends value to channel
`someVar = <-chan` - receives a value from channel
- If the channel is non-buffered or if the buffer is full, the **sending side can block** until a value is read from the reading side
- If the channel is empty **the receiving side can block** until there is a value sent to the channel.
- The goroutine can block, but it is **NOT blocking the OS thread** behind it – the thread just continues to execute the next goroutine in LRQ/GRQ

Channels are first class objects in Go

- You can send channels as payload by using a channel:

```
c := make(chan chan int)
```

- Channels can be given as function arguments

```
func doSomething(input chan int) {  
    // do something  
}
```

- Channels can be returned as function results

```
func doSomethingElse() chan int {  
    result := make(chan int)  
    return result  
}
```

Closing channels

- A channel can be closed using the builtin function **close**:

`close(ch)`

- If closed, the channel can not be opened again
- Writing to a closed channel brings panic
- Reading from a closed channel never blocks
- You can continue reading from the reading side all the values if the channel is buffered.
- After that, the reading returns the **zero value** for the channel data type, and **false** as second result, if read as follows:

`someVar, ok = <-chan`

Example – closing a channel

```
func main() {  
    ch := make(chan string)  
    go func() {  
        for i := 0; i < 10; i++ {  
            ch <- fmt.Sprintf("Sending message number %d", i)  
        }  
        close(ch)  
    }()  
    var val string  
    for ok := true; ok; {  
        val, ok = <-ch  
        fmt.Printf("Receiving: %#v, %#v\n", val, ok)  
    }  
    ch <- fmt.Sprintf("... but I have a question ...") // results in panic  
}
```

Restricted channels

- The channels can be declared as read-only (**<-chan**) or write-only type (**chan<-**):

```
func randomFeed(count, maxVal int) <-chan int {  
    ch := make(chan int)  
  
    go func() {  
        defer close(ch)  
        for i := 0; i < count; i++ {  
            ch <- rand.Intn(maxVal)  
        }  
    }()  
    return ch  
}
```

```
func main() {  
    intVals := randomFeed(5, 1000)  
    for value := range intVals {  
        fmt.Println(value)  
    }  
}
```

Range

Reading from channel until closed can be done most conveniently using **for - range**:

- The **range** blocks until new value is available or the channel is closed
- When the channel is closed the **for-range** loop exits:

```
intVals := randomFeed(5, 1000)
for value := range intVals {
    fmt.Println(value)
}
```

Deadlock

Deadlock happens if a group of goroutines can not continue to progress, because they are mutually waiting each other for something (e.g. to free a resource, or to write/read from channel). For example:

```
func main() {  
    ch := make(chan int)  
    ch <- 1  
    fmt.Println(<-ch)  
}
```

fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:

main.main()

D:/CourseGO/git/coursegopro/08-goroutines-channels/deadlock-simple/deadlock-simple.go:7 +0x37

Deadlock

- A goroutine can get stuck
 - either because it's waiting for a channel or
 - because it is waiting for one of the locks in the sync package
- Common reasons are that
 - no other goroutine has access to the channel or the lock,
 - a group of goroutines are waiting for each other and none of them is able to proceed
- Currently Go only detects when the program as a whole freezes, NOT when a subset of goroutines get stuck.
- With channels it's often easy to figure out what caused a deadlock. Programs that make heavy use of mutexes can, on the other hand, be notoriously difficult to debug.

Signaling to other goroutines and simple synchronization

- Use `struct{} {}` as signalling value – does not consume memory

```
func main() {  
    orderReady := make(chan struct{})  
    go func() {  
        fmt.Println("Chef is preparing the pizza ...")  
        time.Sleep(3 * time.Second)  
        fmt.Println("Pizza is ready!")  
        orderReady <- struct{} {}  
    }()  
    fmt.Println("Taking orders from clients")  
    <- orderReady  
    fmt.Println("Pizza is served ...")  
    fmt.Println("Simulation complete.")  
}
```

Select – multiplexing channel operations

- Similar to switch but for channels, not for types and values
- Selects non-deterministically the first channel that is ready to send or receive a value and executes the corresponding case operations
- Blocks if there is no channel that is ready to send or receive, and no default case is provided
- Example:

```
select {  
case out <- url:  
    fmt.Printf("Generating URL: %s\n", url)  
case <-ctx.Done():  
    return  
}
```

Select – multiplexing channel operations

```
func fibonacci(quit <-chan struct{}) <-chan int {
    fibChannel := make(chan int)
    go func() {
        defer close(fibChannel)
        fmt.Println("Generating fibonacci numbers ...")
        a, b := 0, 1
        for {
            select {
            case fibChannel <- a: // out channel
                fmt.Printf("a = %d\n", a)
                a, b = b, a+b
            case <-quit:          // in channel
                fmt.Println("Canceling generation.")
                return
            }
        }
    }()
    return fibChannel
}
```

```
func main() {

    quitChannel := make(chan struct{})
    fibChannel := fibonacci(quitChannel)
    fmt.Println("Fibonacci consumer goroutine started ...")
    for i := 0; i < 10; i++ {
        value := <-fibChannel
        fmt.Printf("Fibonacci [%d] = %d\n", i, value)
    }
    quitChannel <- struct{}{}
    close(quitChannel)
    fmt.Println("Starting fibonacci generator ...")
    fmt.Printf("Final number of goroutines: %d\n",
        runtime.NumGoroutine())
}
```


Select with default case == non-blocking

```
func fibonacci(quit <-chan struct{}) <-chan int {
    fibChannel := make(chan int)
    go func() {
        defer close(fibChannel)
        fmt.Println("Generating fibonacci numbers ...")
        a, b := 0, 1
        for {
            select {
            case fibChannel <- a: // out channel
                fmt.Printf("a = %d\n", a)
                a, b = b, a+b
            case <-quit:          // in channel
                fmt.Println("Canceling generation.")
                return
            default:
                fmt.Println("No activity – sleeping ...")
                time.Sleep(100 * time.Millisecond)
            }
        }
    }()
    return fibChannel
}
```

```
func main() {

    quitChannel := make(chan struct{})
    fibChannel := fibonacci(quitChannel)
    fmt.Println("Fibonacci consumer goroutine started ...")
    for i := 0; i < 10; i++ {
        value := <-fibChannel
        fmt.Printf("Fibonacci [%d] = %d\n", i, value)
    }
    quitChannel <- struct{}{}
    close(quitChannel)
    fmt.Println("Starting fibonacci generator ...")
    fmt.Printf("Final number of goroutines: %d\n",
        runtime.NumGoroutine())
}
```

Nil channels

- Generally NOT very useful!
- Nil channels have a few interesting behaviors:
 - Sends to them **block forever**
 - Receives from them **block forever**
 - **Closing** them leads to **panic**

```
func main() {  
    var c chan string  
    c <- "message" // Deadlock  
}  
  
func main() {  
    var c chan string  
    fmt.Println(<-c) // Deadlock  
}
```

But sometimes NIL channels can be useful:

```
func sendTo(c chan<- int, iter int) {  
    for i := 0; i <= iter; i++ {  
        c <- i  
    }  
  
    close(c)  
}
```

```
func main() {  
    ch1 := make(chan int)  
    ch2 := make(chan int)  
    go sendTo(ch1, 5)  
    go sendTo(ch2, 10)  
    for {  
        select {  
        case x, ok := <-ch1:  
            if ok {  
                fmt.Println("Channel 1 sent", x)  
            } else { ch1 = nil }  
        case y, ok := <-ch2:  
            if ok {  
                fmt.Println("Channel 2 sent", y)  
            } else { ch2 = nil }  
        }  
        if ch1 == nil && ch2 == nil { break }  
    }  
    fmt.Println("Program finished normally.")  
}
```

Semaphors: limiting the number of concurrent goroutines

```
func DoWork(n int, jobs <-chan struct{}) {  
    fmt.Println("doing", n)  
    time.Sleep(500 * time.Millisecond)  
    fmt.Println("finished", n)  
    <-jobs // release the token  
}  
  
func main() {  
    // concurrentJobs is a buffered channel implementing semaphore that blocks  
    // if more than 20 goroutines are started at once  
    var concurrentJobs = make(chan struct{}, 5)  
    for i := 0; i < 30; i++ {  
        concurrentJobs <- struct{}{} // acquire a token  
        go DoWork(i, concurrentJobs)  
        fmt.Printf("Current number of goroutines: %d\n", runtime.NumGoroutine())  
    }  
}
```

sync.WaitGroup

```
func main() {  
    var wg sync.WaitGroup  
    var urls = []string{  
        "http://www.golang.org/",  
        "http://www.google.com/",  
        "http://www.somestupidname.com/",  
    }  
    for _, url := range urls {  
        wg.Add(1) // Increment the WaitGroup counter.  
        go func(url string) { // Launch a goroutine to fetch the URL.  
            defer wg.Done() // Decrement the counter when the goroutine completes.  
            http.Get(url) // Fetch the URL  
        }(url)  
    }  
    // Wait for all HTTP fetches to complete.  
    wg.Wait()  
}
```

sync.Once

```
type Worker struct {  
    once sync.Once  
}  
  
func (w *Worker) Run() {  
    w.once.Do(func() {  
        fmt.Println("this will run only once")    // printed only once!  
    })  
}  
  
func main() {  
    w := Worker{}  
    w.Run()  
    w.Run()  
    w.Run()  
}
```

Mutexes: sync.Mutex

```
func main() {  
    var wg sync.WaitGroup  
    var mu sync.Mutex  
    wg.Add(2)  
    go func() {  
        mu.Lock()  
        time.Sleep(3 * time.Second)  
        fmt.Println("go routine 1 releasing lock after 3s:", time.Now())  
        mu.Unlock()  
        wg.Done()  
    }()  
    go func() {  
        fmt.Println("go routine 2 trying to acquire lock:", time.Now())  
        mu.Lock()  
        fmt.Println("go routine 2 acquired lock after 3s:", time.Now())  
        mu.Unlock()  
        wg.Done()  
    }()  
    wg.Wait()  
}
```

Mutexes: sync.RWMutex

```
type RWMutex struct {  
    mu sync.RWMutex  
    value int  
}  
  
func (m *RWMutex) Store(value int) {  
    m.mu.Lock()  
    defer m.mu.Unlock()  
    m.value = value  
}  
  
func (m *RWMutex) Load() int {  
    m.mu.RLock()  
    time.Sleep(100 * time.Nanosecond)  
    defer m.mu.RUnlock()  
    return m.value  
}
```


Mutexes: sync.RWMutex – benchmarking I

```
func BenchmarkMain(b *testing.B) {  
    b.Run("BasicMutex-6-1", func(b *testing.B) {  
        BasicMutex_Load(b, 6, 1)  
    })  
    b.Run("BasicMutex-1-6", func(b *testing.B) {  
        BasicMutex_Load(b, 1, 6)  
    })  
    b.Run("BasicMutex-0-6", func(b *testing.B) {  
        BasicMutex_Load(b, 0, 6)  
    })  
    b.Run("RWMutex-6-1", func(b *testing.B) {  
        RWMutex_Load(b, 6, 1)  
    })  
    b.Run("RWMutex-1-6", func(b *testing.B) {  
        RWMutex_Load(b, 1, 6)  
    })  
    b.Run("RWMutex-0-6", func(b *testing.B) {  
        RWMutex_Load(b, 0, 6)  
    })  
}
```

Mutexes: sync.RWMutex – benchmarking II

```
func RWMutex_Load(b *testing.B, nLoads int, nStores int) {  
    mu := RWMutex{}  
    mu.Store(10)  
    for i := 0; i < b.N; i++ {  
        for j := 0; j < nLoads; j++ {  
            go mu.Load()  
        }  
        for j := 0; j < nStores; j++ {  
            go mu.Store(i)  
        }  
    }  
}
```

Mutexes: sync.RWMutex – benchmarking III

goarch: amd64

cpu: Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz

BenchmarkMain

BenchmarkMain/BasicMutex-3-0

BenchmarkMain/BasicMutex-3-0-12	109976	11434 ns/op
---------------------------------	--------	-------------

BenchmarkMain/BasicMutex-6-1

BenchmarkMain/BasicMutex-6-1-12	46353	25159 ns/op
---------------------------------	-------	-------------

BenchmarkMain/BasicMutex-3-3

BenchmarkMain/BasicMutex-3-3-12	58431	21415 ns/op
---------------------------------	-------	-------------

BenchmarkMain/BasicMutex-1-6

BenchmarkMain/BasicMutex-1-6-12	51291	24679 ns/op
---------------------------------	-------	-------------

BenchmarkMain/BasicMutex-0-6

BenchmarkMain/BasicMutex-0-6-12	1000000	1164 ns/op
---------------------------------	---------	------------

BenchmarkMain/RWMutex-3-0

BenchmarkMain/RWMutex-3-0-12	1220876	1002 ns/op
------------------------------	---------	------------

BenchmarkMain/RWMutex-6-1

BenchmarkMain/RWMutex-6-1-12	397364	4849 ns/op
------------------------------	--------	------------

BenchmarkMain/RWMutex-3-3

BenchmarkMain/RWMutex-3-3-12	394406	18694 ns/op
------------------------------	--------	-------------

BenchmarkMain/RWMutex-1-6

BenchmarkMain/RWMutex-1-6-12	295240	31376 ns/op
------------------------------	--------	-------------

BenchmarkMain/RWMutex-0-6

BenchmarkMain/RWMutex-0-6-12	851794	1462 ns/op
------------------------------	--------	------------

PASS

Patterns: Worker Pool - I

```
jobs := make(chan job)
results := make(chan result)
ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
runtime.GOMAXPROCS(runtime.NumCPU())
for i := 0; i < runtime.NumCPU(); i++ {
    go func(ctx context.Context, workerId int, jobs <-chan job, results chan<- result) {
        defer close(results)
        for job := range jobs {
            var r result
            if job.input > 0 {
                r = result{job.id, workerId, process(job.input), nil}
            } else {
                r = result{job.id, workerId, 0, fmt.Errorf("invalid argument error: %f", job.input)}
            }
            select {
            case results <- r:
            case <-ctx.Done():
                return
            }
        }
    }(ctx, i, jobs, results)
}
```

Patterns: Worker Pool - II

```
// send out jobs
go func() {
    for i := 0; i < numJobs; i++ {
        jobs <- job{i, math.Abs(float64(i-14) * math.Pi)}
    }
    close(jobs)
}()

// do something with results
for r := range results {
    if r.err != nil {
        // do something with error
        log.Printf("Error executing Job %d: %v\n", r.jobId, r.err.Error())
        cancel()
    } else {
        fmt.Printf("Job %d executed by worker %d -> Result: %v\n", r.jobId, r.workerId, r.output)
    }
}
```

Quiz 1

- Can you tell, what will be printed by the following main function?

```
func main() {  
    select {}  
    fmt.Println("Demo finished")  
}
```

■



Quiz 2

- Can you tell, what will be printed by the following main function?

```
func main() {  
    ch := make(chan string)  
    go func() { ch <- "hi" }()  
    select {  
    case case1 := <-ch:  
        fmt.Printf("case1: %s\n", case1)  
    case case2 := <-ch:  
        fmt.Printf("case2: %s\n", case2)  
    case <-time.After(time.Nanosecond): // Timeout !!!  
        fmt.Printf("Timeout after 1 ns\n")  
    default:  
        fmt.Printf("No activity ...\n")  
    }  
}
```



Golang Concurrent Programming Advices - I

<https://yourbasic.org/golang/concurrent-programming/> [[CC BY 3.0 license](#).]

- **Goroutines are lightweight threads** - A goroutine is a lightweight thread of execution. All goroutines in a single program share the same address space.
- **Channels offer synchronized communication** - A channel is a mechanism for two goroutines to synchronize execution and communicate by passing values.
- **Select waits on a group of channels** - A select statement allows you to wait for multiple send or receive operations simultaneously.
- **Data races explained** - A data race is easily introduced by mistake and can lead to situations that are very hard to debug. This article explains how to avoid this headache.
- **How to detect data races** - By starting your application with the '-race' option, the Go runtime might be able to detect and inform you about data races.
- **How to debug deadlocks** - The Go runtime can often detect when a program freezes because of a deadlock. This article explains how to debug and solve such issues.

Golang Concurrent Programming Advices - II

<https://yourbasic.org/golang/concurrent-programming/> [[CC BY 3.0 license](#).]

Waiting for goroutines - A `sync.WaitGroup` waits for a group of goroutines to finish.

Broadcast a signal on a channel - When you read from a closed channel, you receive a zero value. This can be used to broadcast a signal to several goroutines on a single channel.

How to kill a goroutine - One goroutine can't forcibly stop another. To make a goroutine stoppable, let it listen for a stop signal on a channel.

Timer and Ticker: events in the future - Timers and Tickers are used to wait for, repeat, and cancel events in the future.

Mutual exclusion lock (mutex) - A `sync.Mutex` is used to synchronize data by explicit locking in Go.

3 rules for efficient parallel computation - To efficiently schedule parallel computation on separate CPUs is more of an art than a science. This article gives some rules of thumb.

Thank's for Your Attention!



Trayan Iliev

IPT – Intellectual Products & Technologies

<http://iproduct.org/>

<http://robolearn.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

<https://www.facebook.com/IPT.EACAD>