Concurrency in Golang

Version 1.0  
  
github: <https://github.com/oprincipe/go-course>

**Table of Contents**

[Definitions 2](#__RefHeading___Toc108_791421740)

[Concurrency in detail 2](#__RefHeading___Toc106_791421740)

[Go Concurrency Model – CSP 3](#__RefHeading___Toc233_791421740)

[Go’s Concurrency Toolset 4](#__RefHeading___Toc235_791421740)

[Channels 4](#__RefHeading___Toc237_791421740)

[Why Blocking is important 4](#__RefHeading___Toc239_791421740)

[Closing channels 5](#__RefHeading___Toc241_791421740)

[Select 6](#__RefHeading___Toc243_791421740)

[Shape your data flow 6](#__RefHeading___Toc267_791421740)

[Where channels fail: 7](#__RefHeading___Toc269_791421740)

[Lock-Free or Weight free code 8](#__RefHeading___Toc281_791421740)

# Definitions

Differences between:

* Parallelism: just parallel execution (all of us wrote this before)
* Concurrency:
  + **Design**: your program as a collection of independent processes
  + **Design**: your those processes to **eventually** run in parallel
  + **Design:** your code so that the outcome is always the same

## Concurrency in detail

* group code (and data) by identifying independent tasks
* no race conditions → this produce unpredictable outcomes
* no deadlocks → no outcomes at all
* more workers → faster execution   
   (slowness could happen if you have bad concurrency model)

# Go Concurrency Model – CSP

Go use the Communicating Sequential Processes (CSP)

* Tony Hoare, 1978
* Each process is built for sequential execution
* Data is communicated between processes via channels  
  **No shared state! Only LOCAL state**
* Scale by adding more of the same

Where parallelism is complicated and you should consider communications, locks and states, CSP simply says: *just write sequential code and forget about that 😃*

No shared state results in:

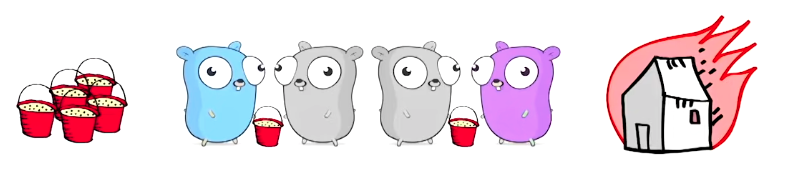
1. No deadlocks
2. No run conditions
3. To send data into another process just simply copy that data to it. This will make you sure that the process will run independently

# Go**’s Concurrency Toolset**

1. go routines
2. channels
3. select
4. sync packages

## Channels

* Think about of a bucket chain
* 3 components: **sender**, ***buffer***, **receiver** (buffer is optional)

The channel is blocked when the first guy pull the bucket but there’s no ones to take it (*no receiver*).

The channel is also blocked when there’s no data. In this case sender is blocked

**Code reference**: concurrency/buffered\_unbuffered.go

### Why Blocking is important

*Remember:*

* *no deadlocks*
* *more workers = faster execution*

Blocking breaks concurrency; can lead to deadlocks; can prevent scaling

### Closing channels

Code reference: concurrency/closing\_channels.go

* Close send a special “closed” message
* The receiver will at some point see “closed” and it will stop trying to get data
* *if you send more “closed” messages it generate* ***panic*** *ends!*

*Example:*

*c := make(chan int)*

close(c)

fmt.Println(<-c)

*What will be printed?*

*# 0, false*

a receiver always returns two values  
- 0 → as it is the zero value of int  
- false → because “no more data” or “returned value is not valid”

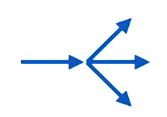
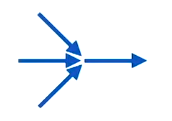
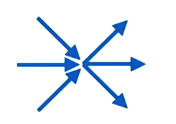
**Always** close channels from the sender and **never** from the receiver

### Select

Code reference: concurrency/select\_channels.go

* Like a switch statement on channel operations
* The order of cases doesn’t matter at all
* There is a default case too
* The first non-blocking case is chosen (send and/or receive)

### Shape your data flow

* Channels are streams of data
* Dealing with multiple streams is the true power of select
* 3 different steps
  + **Fanout** → one stream of data is coming in and multiple come out (like a little load balancer)   
    
  + **Funnel** → join multiple streams in out (like  
    
  + **Turnout** → mix and match between streams  
    

### Where channels fail

* You can create deadlocks with channels  
  *we can make them self unlockable but it’s not desirable. Sometimes we want channels to wait, or operations to wait, and if we do then we have the same problem as before*
* Channels pass around copies, which can impact performance  
  *it’s not really convenient with big data (i.e. photoshop file)*
* Passing pointers to channels can create race conditions  
  *this is a race condition to a backdoor because pointers prevents the sender goroutines to holding the same pointer and modifying afterwards – so, never do this*
* What about “naturally shared” structure like caches or registries?  
  *You can build a cache with a channel in front but (please) don’t do that (it’s really ugly and doesn’t make sense).  
  Caches are about sharing… so far you have some structures where need sharing*

**Can you solve the above problems with Mutex?** [**https://en.wikipedia.org/wiki/Mutual\_exclusion**](https://en.wikipedia.org/wiki/Mutual_exclusion)

Mutex are not an optimal solution because of the following reasons:

* Mutex are like toiles  
  *The longer you occupy them, the longer the queue gets*
* Read/write locks mutexes can only reduce the problem  
  *it only mitigates the problem and you can reach the point where you have more read locks than writes*
* Using multiple mutexes *will* cause deadlocks sooner or later (Resin server? 😂)  
  *early or later the order will be compromised (by human or engine) and you’ll land to deadlocks sometimes later*
* All-in-all not the solution we’re looking for …

**Three shades of code:**

* **Blocking =** Your program may get locked up (for undefined time)
* **Lock free =** At leas one part of your program is always making progress
* **Wait free =** All parts of your programs are always making progress

# Lock-Free or Weight free code

***Easy to learn, hard to master***

There is a packages in the sync package which is named: “Sync Atomic” with a little set of functions in there: store, loads, sets, gets, add, subtract, swap and there’s a fancy function called “**compare and swap**” which swap but only if it reach certain conditions.  
The cool thing of the above functions is that they are **thread safe by design,** actually they are CPU instructions, so your CPU cares about that this works and you don’t need mutexes for that.

## Common patterns

### Spinning Compare And Swap (CAS)

Code reference: concurrency/spinning\_cas.go

(the very basic and easy pattern)  
you need a variable and a constant (which is called “free” in this case) and now you can start spinning and you use these CAS operations.  
So you see if your state equals the free value and if it does you immediately change it to something else.  
As an **atomic** operation, the moment you change this value, you’re the only one doing this.  
Everybody else is still spinning and you’re the only one who’s not, and everybody’s still spinning until you set it to free again.

### Ticket storage

Doc. reference: docs/Atomic Operations Debug.pptx  
 Code reference: concurrency/ticket\_storage.go

* You need an **indexed data structure** (i.e. a slice), a **ticket** and a **done** variable
* A function draws a new ticket by adding 1 to the ticket  
  This function is **atomic** you can be sure that you are the only one who got that number.  
  Nobody else will … *(64 overflow? eheh yes, but this will takes sometimes)*
* Every ticket number is unique as we never decrement
* Treat the ticket as an index to store your data  
  Now that you have you index, you can use it to store data and nobody else could store data with that index (no race condition)
* Increase done to extend the “ready to read” range

when you’ve done, you increment the done counter so that we never know where  
the data is written because the write itself is not atomic