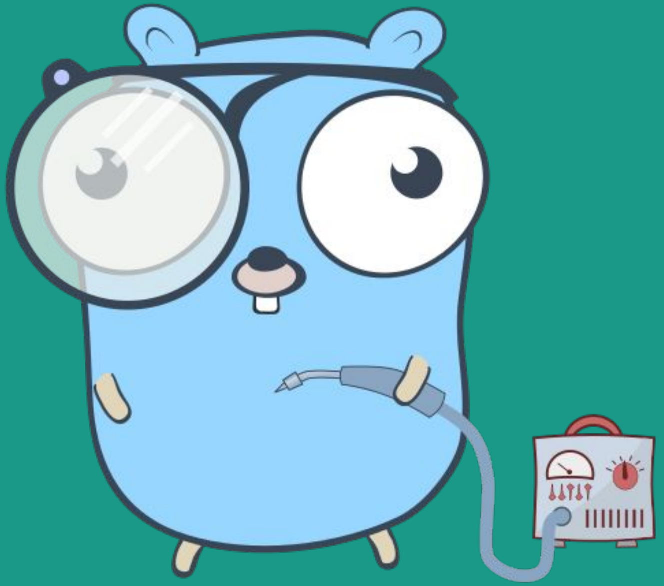
# Concurrency patterns

Ilija Matoski
ilijamt@gmail.com

2022-09-28

https://matoski.com
https://linkedin.com/ilijamt
https://github.com/ilijamt

- Concurrency is the composition of independently executing things (functions or processes in the abstract).
- Parallelism is the simultaneous execution of multiple things (possibly related, possibly not)
- Concurrency is dealing with a lot of things at once.
- Parallelism is doing a lot of things at once.

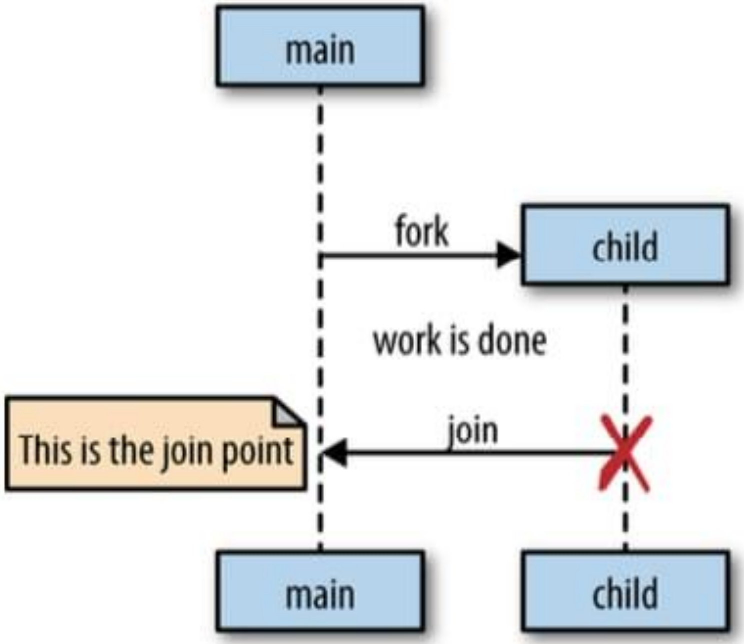


# Building blocks of Concurrency in Go (goroutines)

- Channels
- Buffered Channels
- Select statement
- Wait groups
- Mutex
- sync - Package
- runtime - Go Package

## goroutines

- Fork/Join model
- Not necessarily running in parallel
- Have the same address space
- goroutines as light-weight threads (few kb of memory)
- Have their own call stack which can grow and shrink dynamically.



# channels

- Based on Communicating Sequential Processes put down by Hoare in 1978
- Golang makes use of channels to achieve communication between concurrent processes
- Don’t communicate by sharing memory; share memory by communicating

## channels

close(channelName)

channelName := make(chan datatype)

channelName := make(chan datatype, capacity)

channelName <- data    <- Sending on a channel

data := <- channelName    <- Receiving on a channel

# channels - Directions

```
func ping(pings chan<- string, msg string) {
    pings <- msg
}

func pong(pings <-chan string, pongs chan<- string) {
    msg := <-pings
    pongs <- msg
}
```

```
pings := make(chan string, 1)
pongs := make(chan string, 1)
ping(pings, "passed message")
pong(pings, pongs)
fmt.Println(<-pongs)
```

# select

```
select {
  case channel operation:
    statement(s);
       . //more cases
       .
       .
  default : //Optional
    statement(s);
}
```

# select

The select statement blocks the code and waits for multiple channel operations simultaneously.

- Default Case
- Nil Channel
- Empty Select

```
select {
  case channel operation:
    statement(s);
       . //more cases
       .
       .
  default : //Optional
    statement(s);
}
```

# select - Default Case

- Executes when every other send/receive operation is blocked.
- If we have more than one send/receive operation ready at the same time? The select statement chooses one at random

```
select {
case msg1 := <-channel1:
 fmt.Println(msg1)
case msg2 := <-channel2:
 fmt.Println(msg2)
default:
 fmt.Println("No channel is ready")
}
```

# select - Nil channel

- Nil channel will block the select statement forever, giving an error stating a deadlock
- To disable the send/receive operations of a channel in select statement, we can use nil channels
- To avoid deadlock we need to add a default case or a case that is not blocked

```
ch1 := make(chan string)
ch1 = nil
ch2 := make(chan string)
```

```
select {
case msg1 := <-ch1:
  fmt.Println(msg1)
case msg2 := <-ch2:
  fmt.Println(msg2)
default:
  fmt.Println("No channel is ready")
}
```

# select - Empty Select

- Blocks execution of code on the current goroutine
- **for { }** will cause the CPU% to max, and the process's STATE will be running
- **select { },** will not cause the CPU% to max, and the process's STATE will be sleeping

```
package main

func main() {
  select {}
}
```

```
$ go run empty-select.go
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [select (no cases)]:
main.main()
  /tmp/run.go:4 +0x17
exit status 2
```

# sync

- Cond
- Mutex
- Once
- Pool
- RWMutex
- WaitGroup
- Semaphore

# sync.Cond

- Implements a condition variable which indicates the goroutines which are waiting for an event or want to announce an event
- The thing to note is that if cond.Wait() before cond.Broadcast() then the listeners will keep on waiting forever.

# sync.Cond

```go
var wg = new(sync.WaitGroup)
var vals = make(map[string]int)
wg.Add(5)

m := sync.Mutex{}
cond := sync.NewCond(&m)

go listen("lis1", vals, cond, wg)
go listen("lis2", vals, cond, wg)

go broadcast("b1", vals, cond, wg)

wg.Wait()
```

```go
func listen(name string, a map[string]int, c *sync.Cond, wg *sync.WaitGroup) {
	defer wg.Done()
	c.L.Lock()
	c.Wait()
	fmt.Println(name, " val:", a["T"])
	c.L.Unlock()
}

func broadcast(name string, a map[string]int, c *sync.Cond, wg *sync.WaitGroup) {
	defer wg.Done()
	time.Sleep(time.Second)
	c.L.Lock()
	a["T"] = 25
	c.Broadcast()
	c.L.Unlock()
}
```

## sync.Mutex

- A mutex, or a mutual exclusion, prevents other processes from entering a critical section of data while a process occupies it.

```
mu := sync.Mutex{}
mu.Lock()
go func() {
	mu.Lock() // <- This blocks until mu.Unlock() is called
	fmt.Println("I am in the goroutine")
	mu.Unlock()
}()
fmt.Println("I am in main routine")
mu.Unlock()
time.Sleep(time.Second)
```

## sync.RWMutex

- Stands for Reader/Writer mutual exclusion and is essentially the same as Mutex, but it gives the lock to more than one reading process or just a writing process

# sync.Once

- Is an object that performs an action only once

```
var once sync.Once
onceBody := func() {
	fmt.Println("Only once")
}
done := make(chan bool)
for i := 0; i < 10; i++ {
	go func() {
		once.Do(onceBody)
		done <- true
	}()
}
for i := 0; i < 10; i++ {
	<-done
}
```

# sync.Pool

- Is a collection of temporary objects which can be accessed and saved by many goroutines simultaneously
- Any item stored in the Pool may be removed automatically at any time without notification. If the Pool holds the only reference when this happens, the item might be deallocated.
- A Pool is safe for use by multiple goroutines simultaneously.
- Pool's purpose is to cache allocated but unused items for later reuse, relieving pressure on the garbage collector.
- An example of good use of a Pool is in the fmt package, which maintains a dynamically-sized store of temporary output buffers

# sync.WaitGroup

- Blocks a program and waits for a set of goroutines to finish before moving to the next steps of execution

```
var wg sync.WaitGroup

wg.Add(2)

go func() {
	defer wg.Done()
	// Do work.
	r := time.Duration(rand.Intn(100)) * time.Microsecond
	time.Sleep(r)
	fmt.Printf("Worked for %s\n", r)
}()
go func() {
	defer wg.Done()
	// Do work.
	r := time.Duration(rand.Intn(100)) * time.Microsecond
	time.Sleep(r)
	fmt.Printf("Worked for %s\n", r)
}()

wg.Wait()
fmt.Println("Finished")
```

# sync.Semaphore

- Use them as a Worker Pool
- Binary Semaphore
- Counting Semaphore

# runtime.GOMAXPROCS(n int) int

- GOMAXPROCS allows us to set the maximum number of CPUs that can be executed simultaneously.
- Calling the function will set the number of CPUs to be **n** but will return the previous value set for the number of the CPUs.
- Calling the function with **n < 1** then we return the previous setting

```
fmt.Printf("GOMAXPROCS is %d\n", runtime.GOMAXPROCS(3)) // previous setting
fmt.Printf("GOMAXPROCS is %d\n", runtime.GOMAXPROCS(0)) // will return 3
```

# runtime.NumCPU() int

- Returns the number of logical CPUs that can be used by the current process.

# Benchmarks - synchronization of values

- No synchronization
- Using atomics
- Mutex
- Buffered channels

# Benchmarks - No synchronization

```
var j int64

func Benchmark_Int64(b *testing.B) {
	for i := 0; i < b.N; i++ {
		j++
	}
}
```

| | i7-8550U | 5950X |
|---|---|---|
| **ops** | 870 million | 1 billion |
| **ns/op** | 1.409 | 0.2247 |

## Benchmarks - Using atomics

```
var j int64

func Benchmark_Int64(b *testing.B) {
	for i := 0; i < b.N; i++ {
		atomic.AddInt64(&j, 1)
	}
}
```

| | i7-8550U | 5950X |
|---|---|---|
| **ops** | 183 million | 737 million |
| **ns/op** | 6.638 | 1.638 |

# Benchmarks - Using Mutex

```
var j int64
var m sync.Mutex

func Benchmark_Int64(b *testing.B) {
	for i := 0; i < b.N; i++ {
		m.Lock()
		j++
		m.Unlock()
	}
}
```

| | i7-8550U | 5950X |
|---|---|---|
| **ops** | 92 million | 306 million |
| **ns/op** | 13.07 | 3.888 |

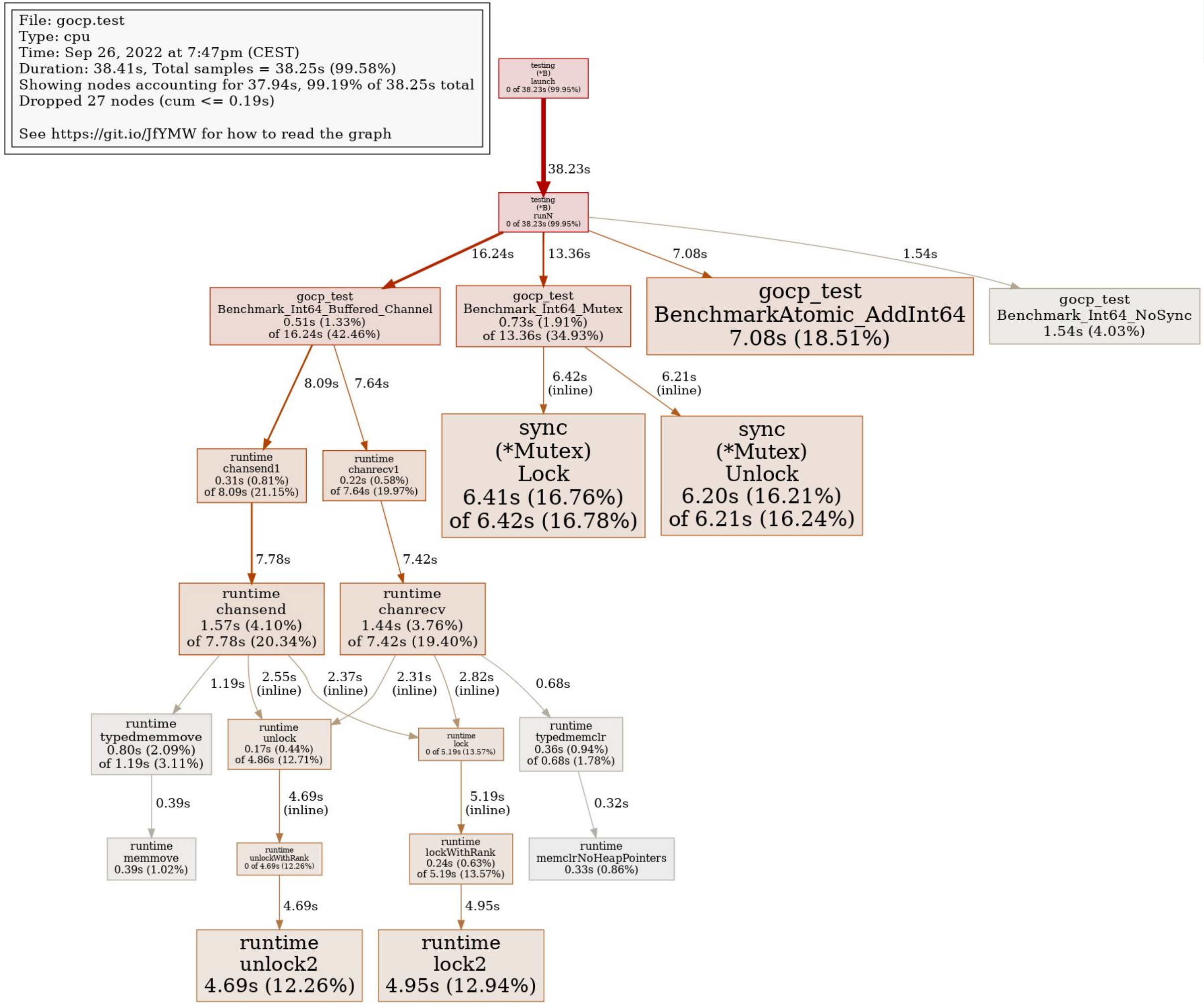
# Benchmarks - Channels

```
var j int64
var c make(chan interface{}, 1)

func Benchmark_Int64(b *testing.B) {
	for i := 0; i < b.N; i++ {
		c <- nil
		j++
		<- c
	}
}
```

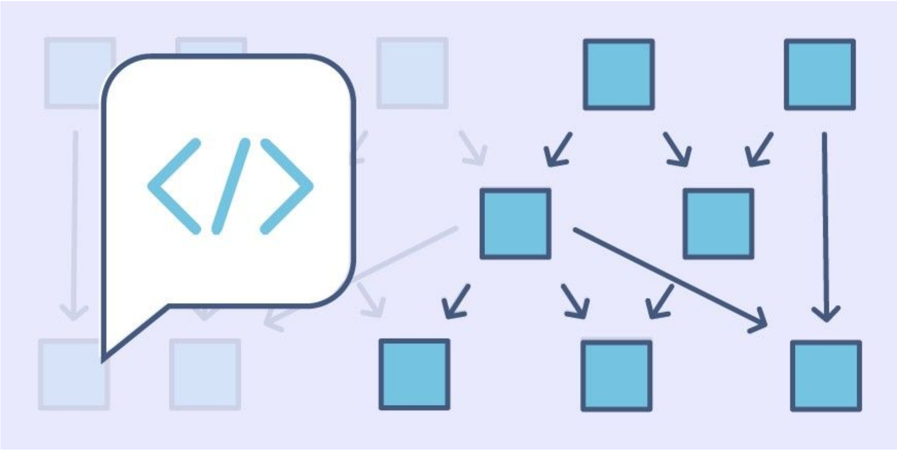
| | i7-8550U | 5950X |
|---|---|---|
| **ops** | 30 million | 57 million |
| **ns/op** | 42.70 | 20.18 |

# Benchmarks

| | No synchronization | | atomics | | Mutex | | Channels | |
|---|---|---|---|---|---|---|---|---|
| | i7-8550U | 5950X | i7-8550U | 5950X | i7-8550U | 5950X | i7-8550U | 5950X |
| ops/million | 870 | 1000 | 183 | 737 | 92 | 306 | 30 | 57 |
| ns/op | 1.409 | 0.2247 | 6.638 | 1.638 | 13.07 | 3.888 | 42.7 | 20.18 |


File: gocp.test
Type: cpu
Time: Sep 26, 2022 at 7:47pm (CEST)
Duration: 38.41s, Total samples = 38.25s (99.58%)
Showing nodes accounting for 37.94s, 99.19% of 38.25s total
Dropped 27 nodes (cum <= 0.19s)
See https://git.io/JfYMW for how to read the graph
testing (*B) launch 0 of 38.23s (99.95%)
38.23s
testing (*B) runN 0 of 38.23s (99.95%)
16.24s
13.36s
7.08s
1.54s
gocp_test Benchmark_Int64_Buffered_Channel 0.51s (1.33%) of 16.24s (42.46%)
gocp_test Benchmark_Int64_Mutex 0.73s (1.91%) of 13.36s (34.93%)
gocp_test BenchmarkAtomic_AddInt64 7.08s (18.51%)
gocp_test Benchmark_Int64_NoSync 1.54s (4.03%)
8.09s
7.64s
6.42s (inline)
6.21s (inline)
runtime chansend1 0.31s (0.81%) of 8.09s (21.15%)
runtime chanrecv1 0.22s (0.58%) of 7.64s (19.97%)
sync (*Mutex) Lock 6.41s (16.76%) of 6.42s (16.78%)
sync (*Mutex) Unlock 6.20s (16.21%) of 6.21s (16.24%)
7.78s
7.42s
runtime chansend 1.57s (4.10%) of 7.78s (20.34%)
runtime chanrecv 1.44s (3.76%) of 7.42s (19.40%)
1.19s
2.55s (inline)
2.37s (inline)
2.31s (inline)
2.82s (inline)
0.68s
runtime typedmemmove 0.80s (2.09%) of 1.19s (3.11%)
runtime unlock 0.17s (0.44%) of 4.86s (12.71%)
runtime lock 0 of 5.19s (13.57%)
runtime typedmemclr 0.36s (0.94%) of 0.68s (1.78%)
0.39s
4.69s (inline)
5.19s (inline)
0.32s
runtime memmove 0.39s (1.02%)
runtime unlockWithRank 0 of 4.69s (12.26%)
runtime lockWithRank 0.24s (0.63%) of 5.19s (13.57%)
runtime memclrNoHeapPointers 0.33s (0.86%)
4.69s
4.95s
runtime unlock2 4.69s (12.26%)
runtime lock2 4.95s (12.94%)

# Patterns in Go



- Generator
- Fan-In, Fan-Out
- Sequencing
- Range and Close
- For-Select
- Quit Channel
- Timeout using Select
- Channel Ring Buffer
- Barriers

## Generator

- Generators return the next value in a sequence each time they are called.
- Each value is available as an output before the generator computes the next value.

```
func fibonacci(n int) chan int {
   ch := make(chan int)
   go func() {
      k := 0
      for i, j := 0, 1; k < n ; k++ {
         ch <- i
         i, j = i+j,i
      }
      close(ch)
   }()
   return ch
}
```

# For-Select

- Select statement lets a goroutine wait on multiple operations
- Select blocks until one of the cases can run
- A random one is chosen if multiple are ready
- If there is a default, it will execute on every loop without blocking the loop and waiting on a value from a channel

```
for {
  select {
  case i := <-c1:
    // use i
  case a:= <- c2
    // use a
  default:
    // executes on every turn
  }
}
```

# Range and Close

- Only the sender should close a channel, never the receiver
- Sending on a closed channel will cause a panic
- Closing is only necessary when the receiver must be told there are no more values coming, such as to terminate a range loop.

```
func fibonacci(n int, c chan int) {
	x, y := 0, 1
	for i := 0; i < n; i++ {
		c <- x
		x, y = y, x+y
	}
	close(c)
}

c := make(chan int, 10)
go fibonacci(cap(c), c)
for i := range c {
	fmt.Println(i)
}
```

# Timeout using Select

```
for {
    select {
    case s := <-ch:
        fmt.Println(s)
        return
    case <-time.After(time.Second):
        fmt.Println("bail!")
        return
    }
}
```

# Quit Channel

```go
channel := make(chan string)
quit := make(chan string)

for i := 0; i < 6; i++ {
	go Race(channel, quit, i)
}

for {
	select {
	case raceUpdates := <-channel:
		fmt.Println(raceUpdates)
	case winnerAnnoucement := <-quit:
		fmt.Println(winnerAnnoucement)
		return
	}
}
```

```go
func Race(channel, quit chan string, i int) {
	channel <- fmt.Sprintf("Mouse %d started!", i)
	for {
		rand.Seed(time.Now().UnixNano())
		time.Sleep(time.Duration(rand.Intn(500)+500) * time.Millisecond)
		quit <- fmt.Sprintf("Mouse %d reached the finishing line!", i)
	}
}
```

```
Mouse 5 started!
Mouse 1 started!
Mouse 0 started!
Mouse 3 started!
Mouse 2 started!
Mouse 4 started!
Mouse 5 reached the finishing line!
```

## Channel Ring Buffer

- Connect two buffered channels that forward messages from the incoming channel to the outgoing channel
- If a message cannot be placed on the outgoing channel, drop the oldest message in the outgoing channel
- Slow consumers might lose messages (their oldest messages), but it will never block

# Channel Ring Buffer

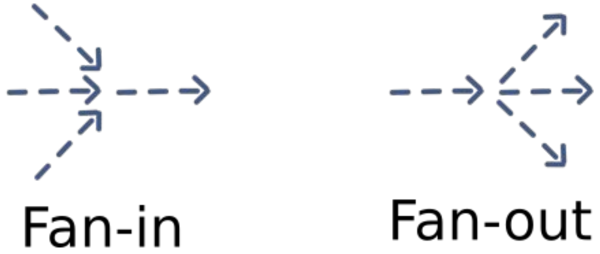
```go
type RingBuffer struct {
  inputCh  <-chan int
  outputCh chan int
}

func (r *RingBuffer) Run() {
  for v := range r.inputCh {
    select {
    case r.outputCh <- v:
    default:
      <-r.outputCh
      r.outputCh <- v
    }
  }
  close(r.outputCh)
}
```

```go
func main() {
  in := make(chan int)
  out := make(chan int, 5)
  rb := &RingBuffer{in, out}
  go rb.Run()

  for i := 0; i < 10; i++ {
    in <- i
  }

  close(in)

  for res := range out {
    fmt.Println(res)
  }
}
```

```
go run ring-buffer.go
4
5
6
7
8
9
```

# Fan-In, Fan-Out

- It’s a way to converge and diverge data into a single data stream from multiple streams or from one stream to multiple streams or pipelines
- Fan-In refers to a technique in which you join data from multiple inputs into a single entity.
- Fan-Out means to divide the data from a single source into multiple smaller chunks. In this lesson, we’ll learn how to make use of both these techniques.



# Questions