# concurrency in go

Gilvan Ritter
github.com/gritt

Software Developer at ThoughtWorks

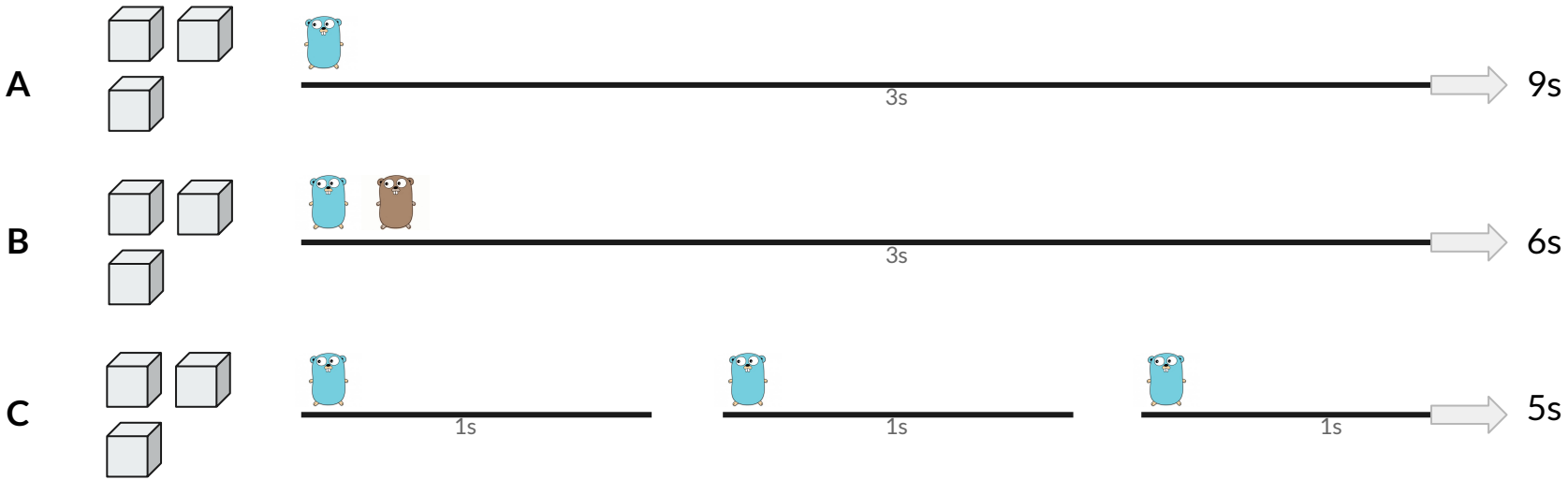
# paralelismo vs concorrência

- paralelismo é a execução **simultânea** de processos, indepententes ou dependentes
- paralelismo é possível em CPUs multi-core
- **paralelismo é sobre execução**
- **paralelismo é executar várias coisas ao mesmo tempo**

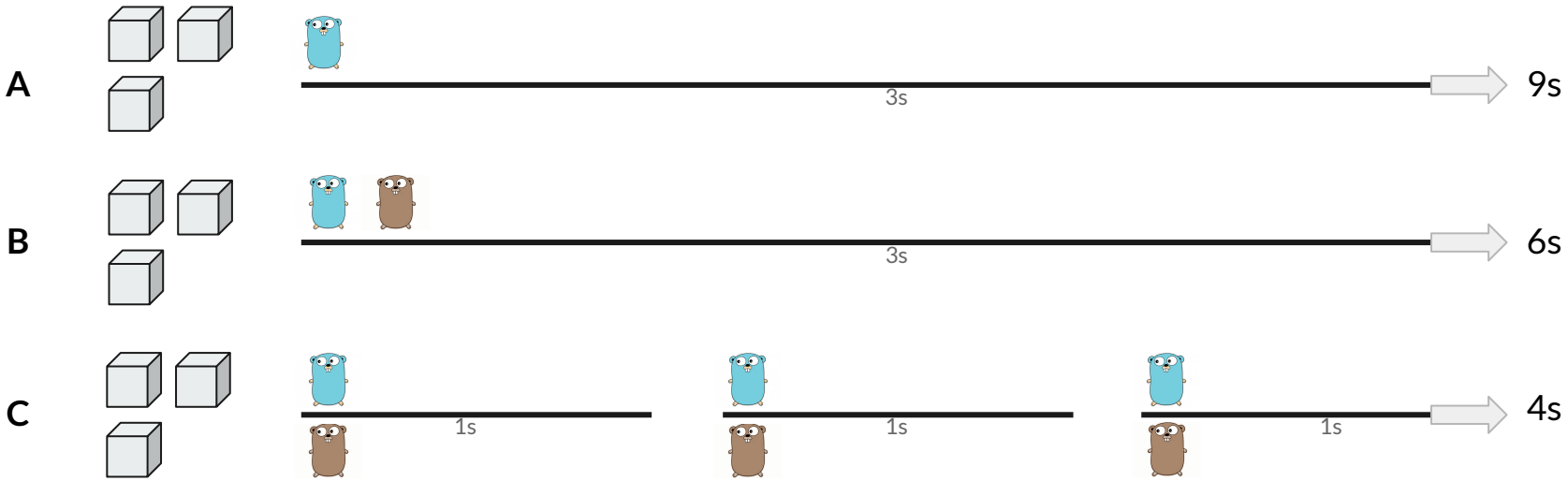
# paralelismo vs concorrência

- concorrência é a composição de processos executando de maneira independente
- concorrência é como estruturar um programa em partes independentes e coordenar a sua execução com algum tipo de comunicação
- **concorrência é sobre estrutura**
- **concorrência é lidar com várias coisas ao mesmo tempo**

# paralelismo vs concorrência



# paralelismo vs concorrência



# goroutines

- implementação de concorrência em Go, built in
  - `go func(){...}`
- executam no mesmo endereço de memória e são mais baratas que threads
- são gerenciadas pelo Go em conjunto com o OS
- abstrai o agendamento / execução  etc

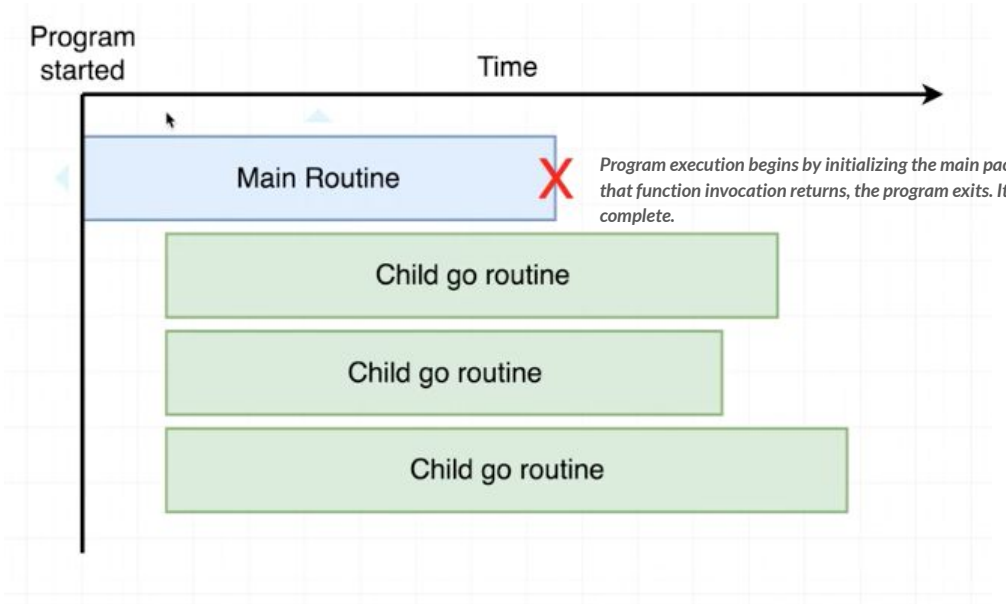
# goroutines

```
fmt.Print("hello")

go func() {
    fmt.Print("i am a goroutine")
}

fmt.Print("world")
```

# goroutines




*Program execution begins by initializing the main package and then invoking the function main. When that function invocation returns, the program exits. It does not wait for other (non-main) goroutines to complete.*

# channels

- "channels" são canais que permitem trocar informações entre goroutines
- são valores tipados, carregam informações do tipo atribuído
- são usados para comunicação e sincronização de goroutines
- podem possuir um tamanho (buffer)

# channels

```
jobs := make(chan int)

//envia 5 para o canal jobs

jobs <- 5

// lê canal jobs e atribui à variável jobNumber

jobNumber := <- jobs  // block

fmt.Println(<- jobs)  // block
```

# wait groups

- para esperar múltiplas goroutines finalizarem
- funciona como um contador
- ao lançar goroutines, incremente o WaitGroup
- ao finalizar uma goroutine decremente  o WaitGroup

# wait groups

```
var wg sync.WaitGroup

for i := 1; i <= 5; i++ {

    wg.Add(1)

    go func(i, &wg) {... defer wg.Done()}

}

wg.Wait()
```

# jobs / workers pattern

- é uma padrão comum na construção de pipelines de consumidores
- ETL: extract, transform, load
- consumidores executam essas pipelines de tempo em tempo indefinidamente

# referências


- https://blog.golang.org/waza-talk
- https://github.com/gritt/go-data-intensive
- https://gobyexample.com
- http://marcio.io/2015/07/handling-1-million-requests-per-minute-with-golang