

# Afinal, que tem de bom na concorrência em Go?

Tirar proveito do hardware nunca foi tão fácil

# \$ id Icaparelli

- Lucas Caparelli
- Engenheiro de Software  
@Gympass
- Trabalho estendendo uma  
plataforma de produtos  
internos baseada em  
Kubernetes




# Concorrência

- Eventos concorrentes são aqueles sobre os quais não se pode afirmar nada em relação a sua ordenação
- Podem ou não ocorrer simultaneamente (*em paralelo*)
- Dois eventos: **A** e **B**
- Relação “acontece-antes-de”:  $\rightarrow$
- Se **A**  $\rightarrow$  **B** ou **B**  $\rightarrow$  **A**: não são concorrentes
- Caso contrário: são concorrentes

# Exemplo Depósito Bancário

- Se eu fizer um depósito na minha própria conta:
  - **evento A:** lê o saldo atual da conta
  - **evento B:** atualiza o saldo levando em conta o depósito
- Se uma outra pessoa fizer um depósito na minha conta:
  - **evento C:** lê o saldo atual da conta
  - **evento D:** atualiza o saldo levando em conta o depósito
- Concluimos que: **A -> B; C -> D;** e ***nada mais***



# Concorrência != Paralelismo

# Calculando Pi

```
import "io"

func calcularPi(w io.Writer) {
    var piAtual string
    for {
        piAtual = computarProximoDigitoPi(piAtual)
        w.Write([]byte(piAtual))
    }
}
```

# Calculando Pi e gravando concorrentemente

```
import "io"

func calcularPi(w io.Writer, capacidade int) {
    pisComputados := make(chan string, capacidade)
    go func(){
        pi := <- pisComputados
        w.Write([]byte(pi))
    }()

    var piAtual string
    for {
        piAtual = computarProximoDigitoPi(piAtual)
        pisComputados <- piAtual
    }
}
```

# Sincronização



# Sincronização entre processos

- Processos concorrentes precisam se comunicar para saber como proceder e *quando* proceder
- Classicamente através do compartilhamento de memória (locks, semáforos, etc.)
- Alta complexidade, carga maior no programador

# Communicating Sequential Processes (CSP)

- Proposta por Tony Hoare, 1984
- Não existe compartilhamento de estado
- Processos sincronizam entre si através de mensagens
- Base para o modelo de concorrência em Go
- “Não se comunique compartilhando memória; compartilhe memória se comunicando” - provérbio Go

# Problemas em Concorrência

# Condições de Corrida

```
package main

type usuario struct {
    nome  string
    saldo float64
}

func depositar(quant float64, usuario *user) {
    novoSaldo := usuario.saldo + quant
    usuario.saldo = novoSaldo
}

func main() {
    alice := &usuario{nome: "alice"}
    go depositar(100.0, alice)
    depositar(200.0, alice)
}
```

# Exemplo de Execução

routine A - depositar(100.0, alice)

1 novoSaldo := usuario.saldo + quant  
  (novoSaldo == 100.0)

2 .....

3 .....

4 usuario.saldo = novoSaldo  
  (alice.saldo == 100.0)

routine B - depositar(200.0, alice)

novoSaldo := usuario.saldo + quant  
(novoSaldo == 200.0)

usuario.saldo = novoSaldo  
(alice.saldo == 200.0)

# Condições de Corrida - ~~Arrumado!~~ Sincronizado!

```
package main

import "sync"

type usuario struct {
    nome  string
    saldo float64
    lock  sync.Mutex
}

func depositar(quant float64, usuario *user) {
    user.lock.Lock()
    novoSaldo := usuario.saldo + quant
    usuario.saldo = novoSaldo
    user.lock.Unlock()
}

func main() {
    alice := &usuario{nome: "alice"}
    go depositar(100.0, alice)
    depositar(200.0, alice)
}
```

Locks solucionam todos nossos  
problemas! ... só que não.

# Deadlocks

```
package main

import "sync"

type numero struct {
    valor float64
    lock  sync.Mutex
}

func trocarValores(a,b *numero) {
    a.lock.Lock()
    b.lock.Lock()
    a.valor, b.valor = b.valor, a.valor
    b.lock.Unlock()
    a.lock.Unlock()
}

func main() {
    um := &numero{valor: 1.0}
    dois := &numero{valor: 2.0}

    go trocarValores(um, dois)
    trocarValores(dois, um)
}
```



# Exemplo de Execução

routine A - trocarValores(um, dois)

```
1 a.lock.Lock()  
  //(A adquire lock de 'um')  
2 .....  
3 b.lock.Lock()  
  //(A bloqueia esperando lock 'dois')  
4 .....  
5 .....  
6 .....  
7 .....  
8 .....  
9 .....  
10 .....
```

routine B - trocarValores(dois, um)

```
a.lock.Lock()  
//(B adquire lock de 'dois')  
  
b.lock.Lock()  
//(B bloqueia esperando lock 'um')
```

# Entre outros problemas...

- Inanição (starvation), não-determinismo, contenção...
- Alguns inerentes à concorrência
- Outros podem ser menores de acordo com o nível de abstração
- CSP introduz uma camada de abstração acima de mecanismos de sincronização clássicos

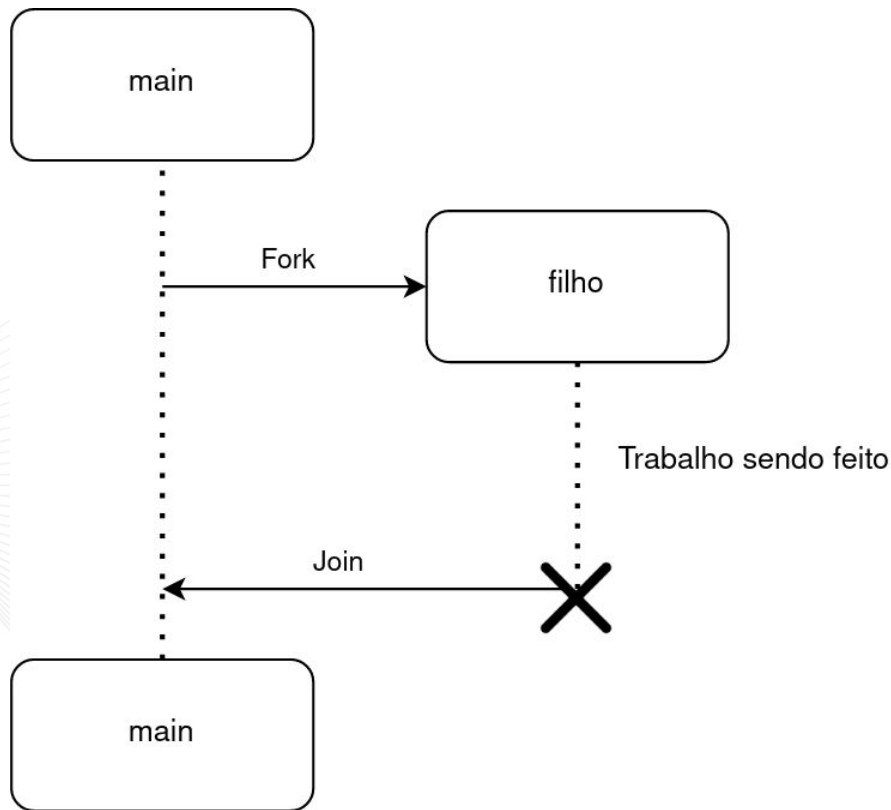
# Channels & Goroutines

# Channels

- Seguro para uso concorrente
- Permite sincronização através da troca de mensagens
- Facilmente compostos entre si
- Similar à ideia de mensageria e eventos em arquitetura de software
- escrita: **channel <-**
- leitura: **<- channel**
- Usa locks por baixo dos panos, mas você não precisa gerenciá-las no seu código ;-)

# Goroutines

- “threads” super leves, começando com 4 KB
- Gerenciadas pelo runtime
- Iniciadas pela keyword **go** a frente de uma chamada de função
- Modelo fork-join



# Revisitando Pi

```
import "io"

func calcularPi(w io.Writer, capacidade int) {
    pisComputados := make(chan string, capacidade)
    go func(){
        pi := <- pisComputados
        w.Write([]byte(pi))
    }()

    var piAtual string
    for {
        piAtual = computarProximoDigitoPi(piAtual)
        pisComputados <- piAtual
    }
}
```

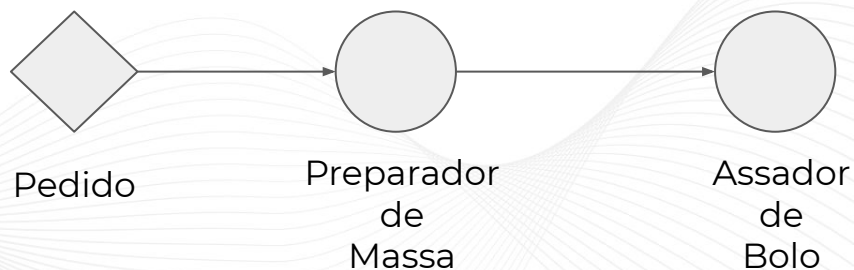
# CSP

Exemplos em:

<https://github.com/LCaparelli/presentations/tree/main/presentations/concorrenca-go>

# Fazendo bolo

- Preparar a massa
- Assar bolo

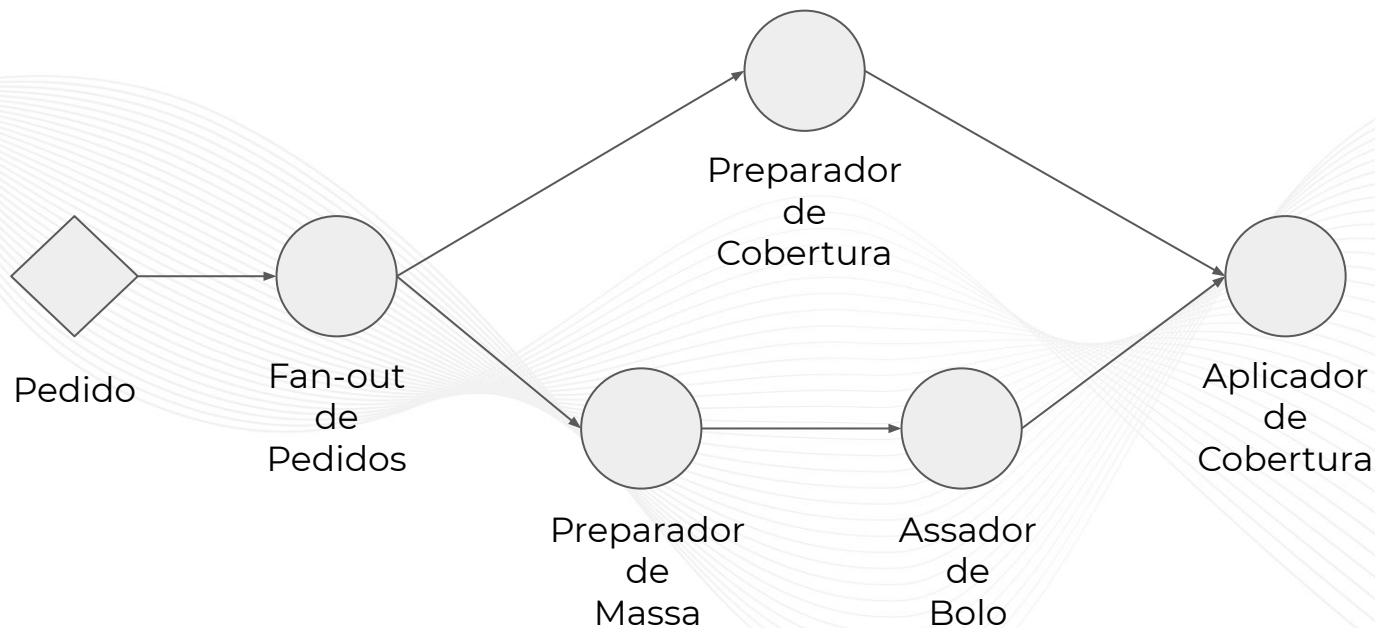




# Fazendo bolo

- Preparar a massa
- Assar bolo
- Preparar cobertura
- Aplicar cobertura

# Fazendo bolo

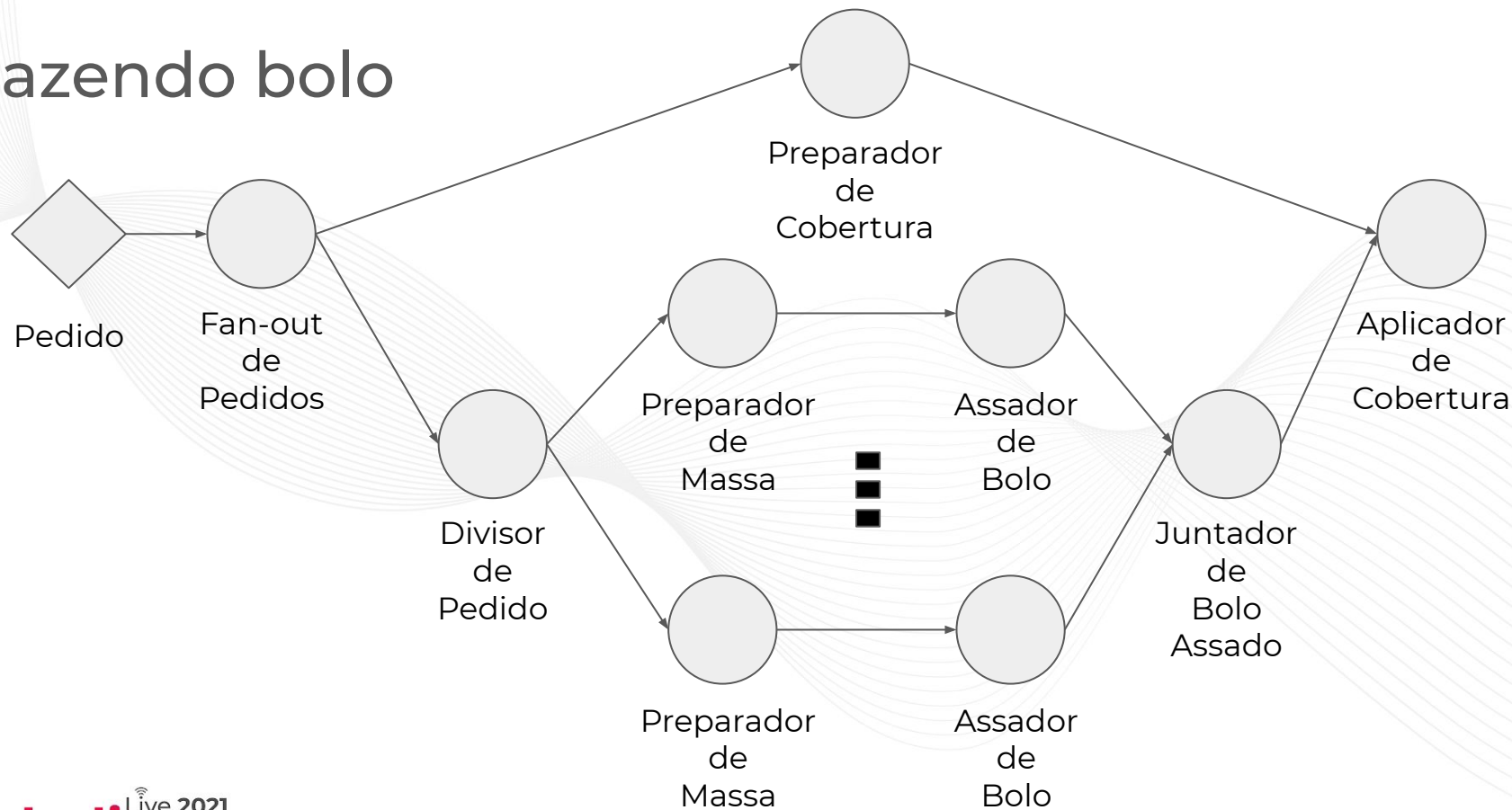




Dá pra melhorar?

Dá!\*

# Fazendo bolo



# Runtime

# “Threads” leves...

- Goroutines não são threads!
- Goroutines são **multiplexadas** nos processadores pelo runtime
- Programador não precisa se preocupar com gerenciamento de threads do SO
- Programador não precisa saber quantas threads terá disponível em execução
- Programador *raramente* precisa se preocupar com número de goroutines executando

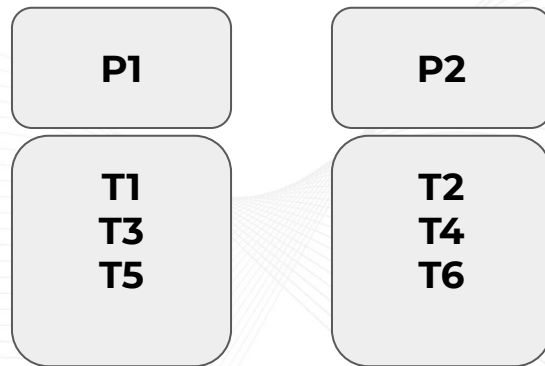
# Multiplexadas?

- O runtime decide qual goroutine vai executar e quando
- Cada goroutine tem seu próprio contexto
- Quando uma goroutine bloqueia ou é preemptada, o runtime realiza sua troca de contexto inserindo outra goroutine na thread de SO em execução



# Escalonamento de tarefa - abordagem ingênua

- **P** processadores
- **T** tarefas
- Dividir as **T** tarefas entre os **P** processadores igualmente

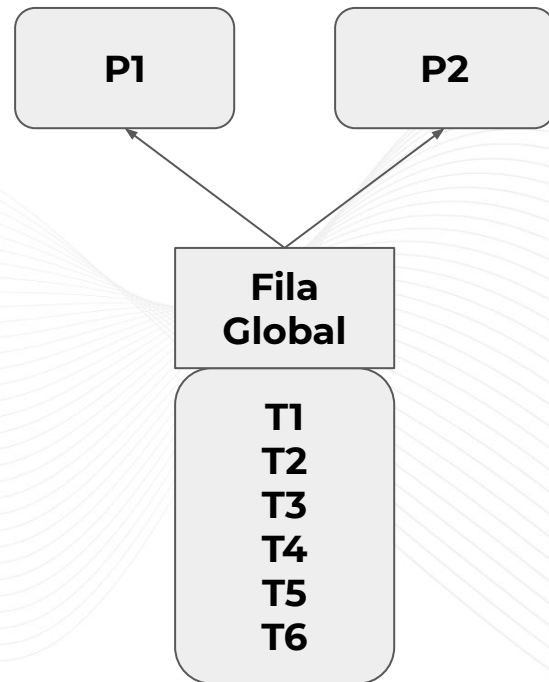


# Escalonamento de tarefa - abordagem ingênua

- E se **T1**, **T3** e **T5** terminarem após 1 segundo enquanto as outras vão levar o dobro? **P1** ficaria sem trabalho para fazer mesmo com tarefas pendentes
- E se **T1**, **T3** e **T5** todas dependerem das outras tarefas? **P1** ficará sem trabalhar até as demais tarefas serem cumpridas
- Localidade de cache potencialmente prejudicada, já que tarefas relacionadas podem acabar em processadores diferentes

# Escalonamento de tarefa - fila global

- **P** processadores
- **T** tarefas
- Enfileirar tarefas numa fila global
- Processadores buscam tarefas na fila
- Seção crítica para desenfileirar

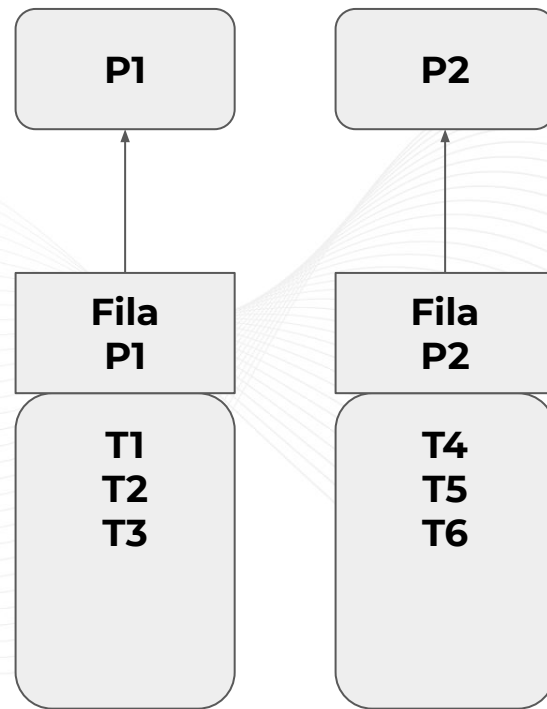


# Escalonamento de tarefa - fila global

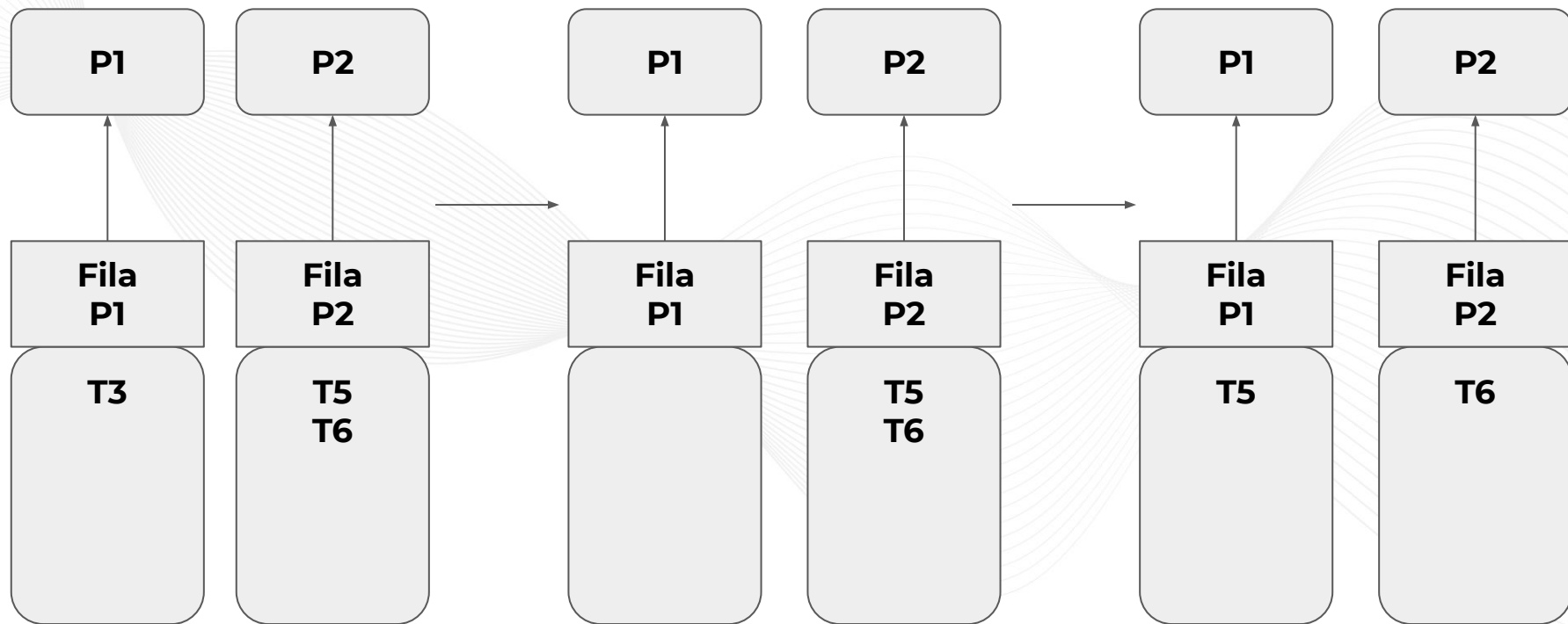
- Processadores não ficam mais sem trabalhar enquanto houver tarefas na fila
- Problema de contenção de lock, overhead de gerenciamento
- Problema de localidade de cache exacerbado: cada processador vai precisar carregar a fila em cache toda vez que buscar uma tarefa para executar

# Escalonamento de tarefa - fila descentralizada

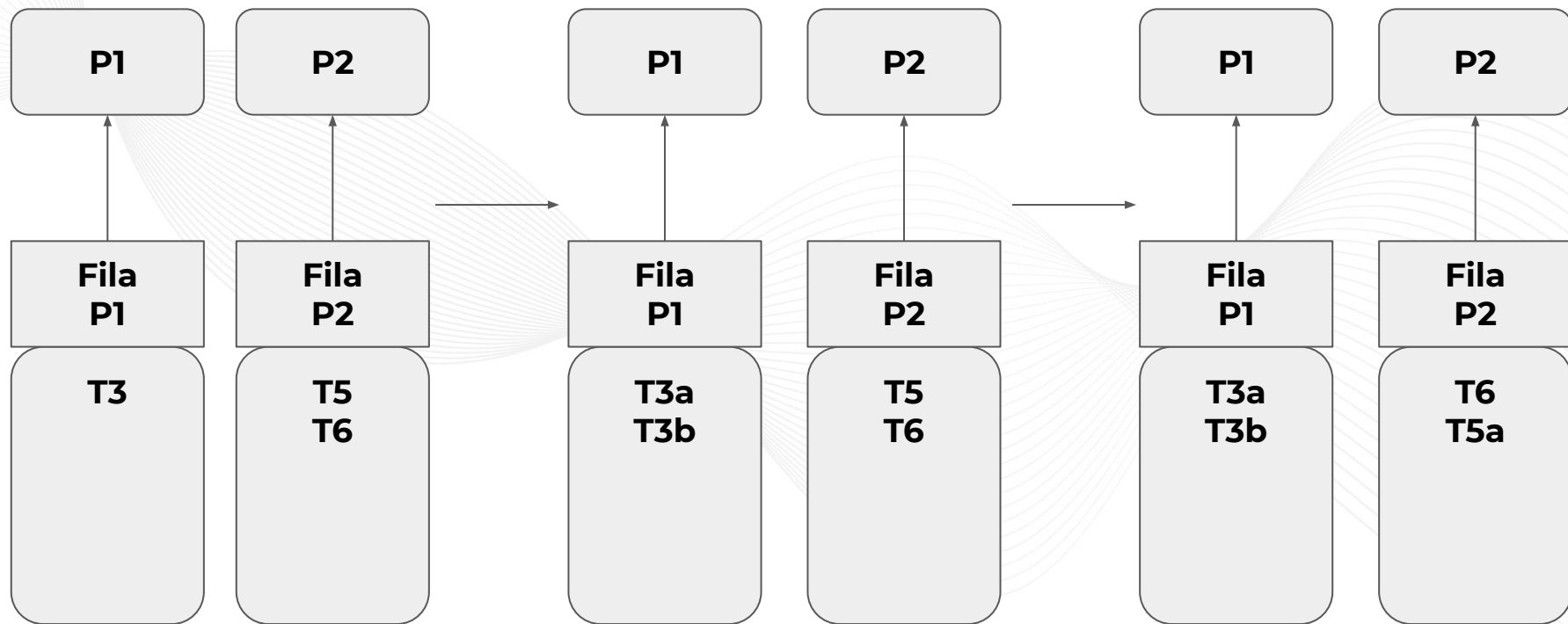
- **P** processadores
- **T** tarefas
- Cada processador tem sua própria fila
- Processadores roubam tarefas de outras filas se a sua estiver vazia
- Tarefas filhas enviadas para fila do mesmo processador



# Escalonamento de tarefa - fila descentralizada



# Escalonamento de tarefa - fila descentralizada



# Roubando tarefas ou continuações?

- Tarefas em Go são as Goroutines
- Continuações são todas instruções abaixo da criação de uma goroutine

```
func algumaFunc() {  
    go fazerTrabalhoConcorrente() // tarefa  
    continuarTrabalho()           // continuação  
}
```

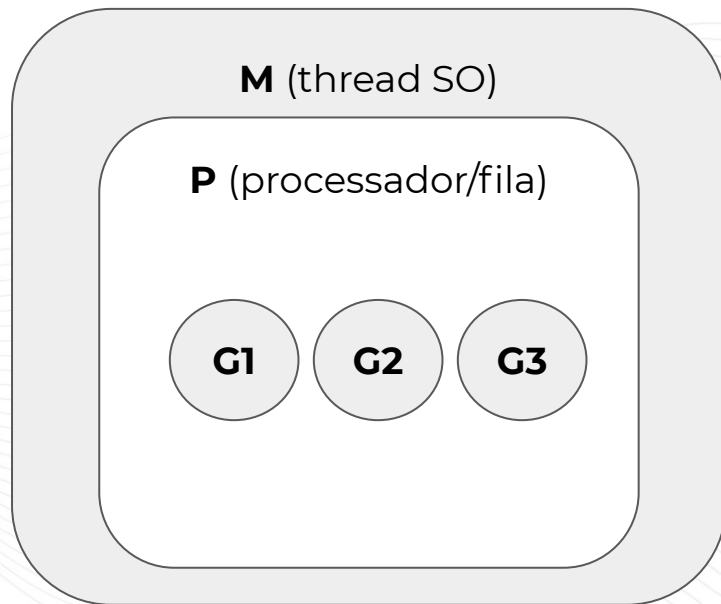


# Roubando tarefas ou continuações?

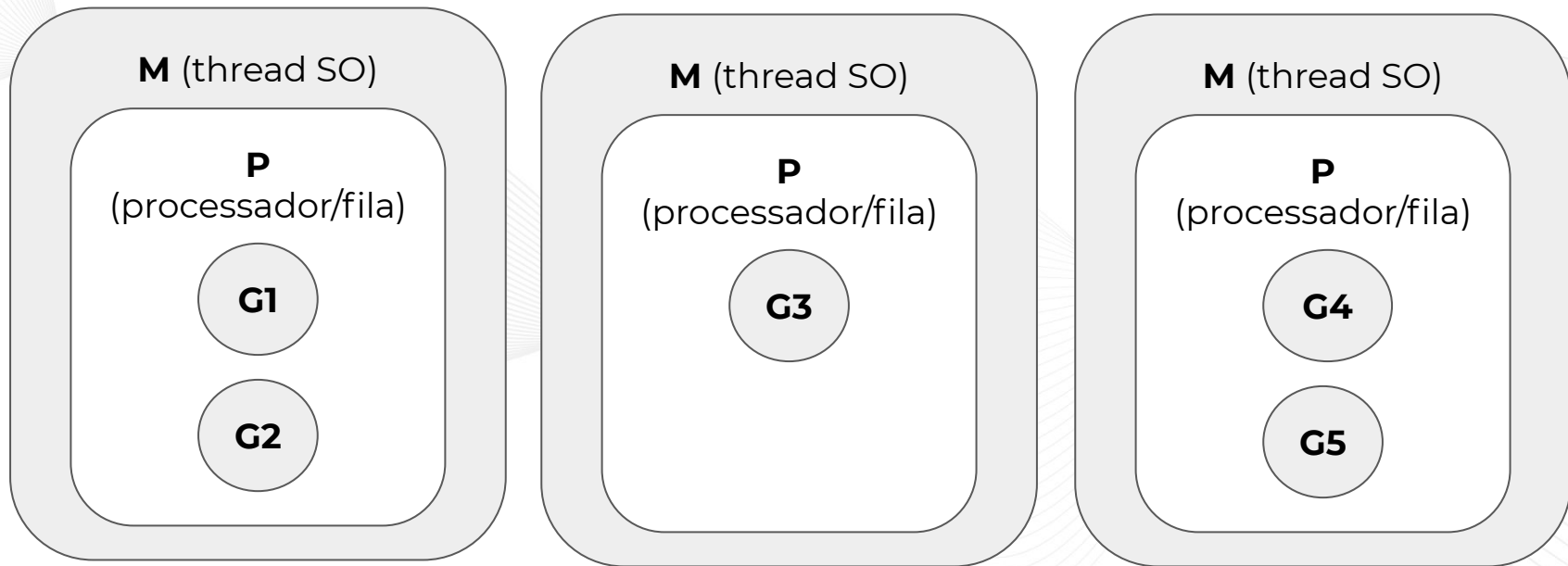
- Runtime enfileira continuações
- Tarefas são executadas imediatamente
- Caso comum é de que o resultado do trabalho concorrente será usado na continuação
- Faz sentido partir para esse trabalho imediatamente
- Faz com que um processador bloqueie menos vezes por chegar num ponto de Join que não pode ser cumprido

# Escalonamento de tarefa - jeito Go

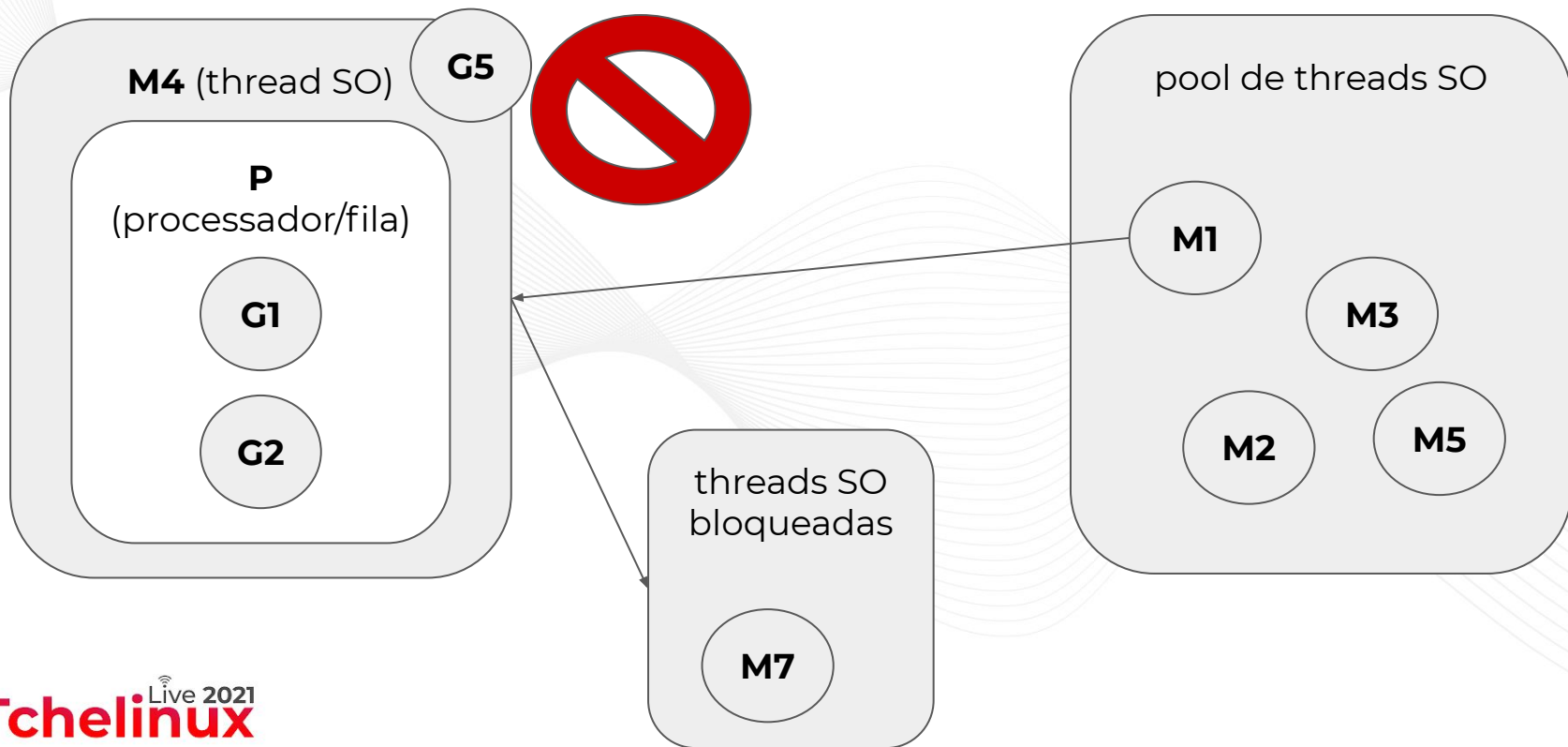
- **P** processadores/filas
- **G** goroutines (estado -- PC)
- **M** thread do SO/máquina
- Threads do SO são iniciadas pelo runtime e hospedam processadores/filas
- Processadores escalonam e hospedam goroutines



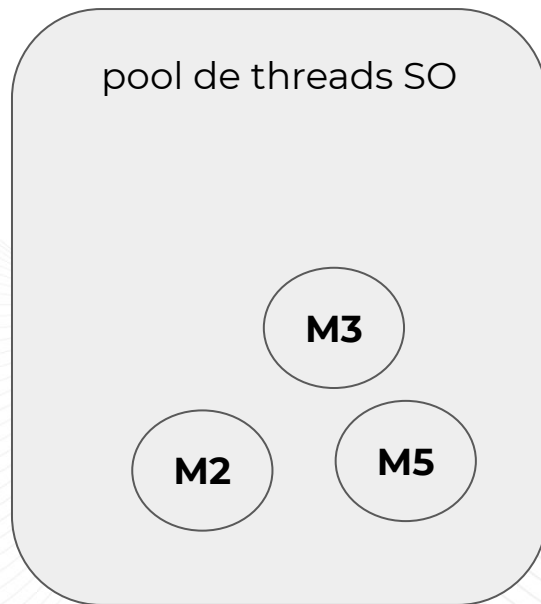
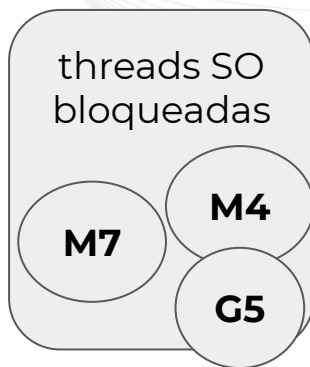
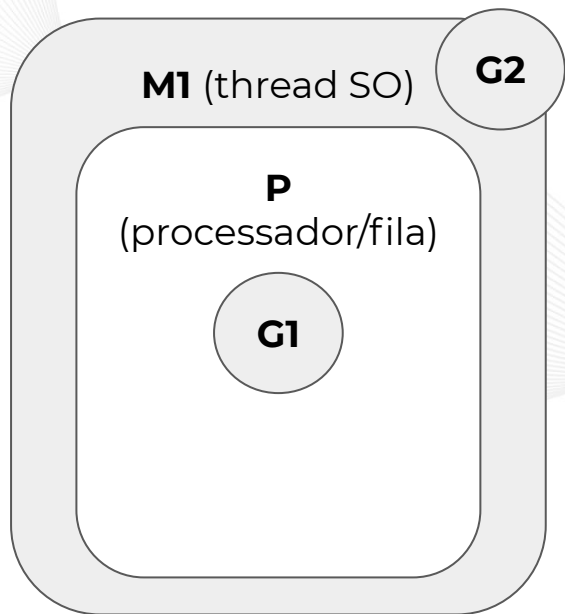
# Escalonamento de tarefa - jeito Go



# Threads bloqueadas?



# Threads bloqueadas?



# Conclusão

- Runtime faz de tudo para ser eficiente e escalável, e toda essa complexidade fica debaixo dos panos: para usar tudo isso precisamos apenas da keyword **go**
- Channels trazem uma camada de abstração mais simples para sincronização de processos concorrentes
- Goroutines são construídas em cima de funções: primitivas familiares a qualquer programador, não precisamos saber nada sobre escalonamento, estruturas de dados complicadas, etc
- Channels promovem design em torno de estruturas que sejam facilmente compostas com outras