Laboratório de Programação Concorrente

Lab7 - Go Routines and Channels - 25.2

Objetivo

O objetivo deste laboratório é que você se familiarize com a utilização de canais na linguagem Go, explorando diferentes cenários de comunicação concorrente entre produtores e consumidores. O desenvolvimento será incremental, em etapas que aumentam gradativamente a complexidade e introduzem novos conceitos: pipelines, múltiplos produtores/consumidores, canais unidirecionais e canais bufferizados.

Descrição

Considere que você está desenvolvendo um sistema de monitoramento de sensores em uma rede IoT. Cada sensor gera dados numéricos de forma assíncrona, e um servidor central precisa processar essas leituras. A comunicação entre sensores e servidor será feita por canais em Go.

Etapas Incrementais

Etapa 1: Produtor e Consumidor Simples

- Implemente um programa com uma goroutine produtora que gera valores aleatórios (simulando leituras de um sensor) entre 0 e 100.
- Uma goroutine consumidora lê do canal e imprime apenas os valores acima de um limite pré-definido (exemplo: maior que 50).
- Tanto o produtor quanto o consumidor rodam em loop infinito.

Etapa 2: Produtor Finito

- Modifique a versão anterior para que o produtor gere apenas 10.000 valores aleatórios.
- Quando terminar, feche o canal e encerre o programa.

Etapa 3: Múltiplos Sensores (Produtores)

- Agora considere que existem dois sensores (duas goroutines produtoras).
- Cada produtor gera uma quantidade aleatória de valores.
- O consumidor central (único) recebe valores de ambos e continua imprimindo apenas os que estão acima do limite.

Etapa 4: Canal Unidirecional e Bufferizado

- Refaça o programa anterior:

    - Defina o canal como unidirecional (chan<- para produtores e <-chan para consumidores).
    - Use um canal bufferizado (ex.: buffer de 100 elementos) para reduzir bloqueios entre produtores e consumidor.

Etapa 5: Vários Consumidores

- Evolua para ter múltiplas goroutines consumidoras, cada uma processando valores recebidos do canal.
- Cada consumidor imprime valores com uma identificação própria (ex.: Consumidor 1 recebeu 87).

Observe e discuta com sua dupla o comportamento de divisão de trabalho entre os consumidores.

Para gerar números aleatórios, por exemplo entre 0 e 100, use a biblioteca abaixo:

```
"math/rand"

rand.Seed(42)
for {
	v := rand.Intn(100)
}
```

Visão geral do código base

Aqui temos o pdf deste laboratório.

https://github.com/giovannifs/fpc/blob/master/2025.2/Lab7/

Todas as soluções devem ser desenvolvidas no diretório src dentro do Lab7, um arquivo para cada etapa do lab. Seguem os nomes:

- lab7-e1.go
- lab7-e2.go
- lab7-e3.go
- lab7-e4.go
- lab7-e5.go

Prazo

03/feb/26 16:00

Entrega

Você deve criar e manter um repositório privado no GitHub com a sua solução. No entanto, a entrega do laboratório deverá ser realizada por meio de submissão online utilizando o script submit-answer.sh, disponibilizado na estrutura de arquivos do próprio laboratório. Uma vez que você tenha concluído sua resposta, seguem as instruções:

1) Crie um arquivo lab7_matr1_matr2.tar.gz somente com o "src" do repositório que vocês trabalharam. Para isso, supondo que o diretório raiz de seu repositório privado chama-se lab7_pc, você deve executar:

```
tar -cvzf lab7_matr1_matr2.tar.gz lab7_pc/src
```

2) Submeta o arquivo lab7_matr1_matr2.tar.gz usando o script submit-answer.sh, disponibilizado no mesmo repositório do laboratório:

```
bash submit-answer.sh lab7 path/lab7_matr1_matr2.tar.gz
```

Lembre-se que você deve manter o seu repositório privado no GitHub para fins de comprovação em caso de problema no empacotamento ou transmissão online. Alterações no código realizadas após o prazo de entrega não serão analisadas.

Canais Bufferizados

Há dois tipos de canais: bufferizados e não-bufferizados

Já vimos a criação de canais não-bufferizados

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

Para criar canais bufferizados, simplesmente especificamos o tamanho

do buffer

```
ch := make(chan int, 3)
```

Uma operação send em um canal não-bufferizado bloqueia a goroutine sender, até que outra goroutine execute a operação receive. De modo análogo, se a goroutine receiver executar primeiro, bloqueará até que sender faça sua parte.

Canais Unidirecionais

Canais unidirecionais impõem restrições que tornam o código ainda mais robustos

in <-chan string passa a ser um canal receive only

out chan<- string passa a ser um canal send only

o uso do canal é verificado em tempo de compilação

```
func filterContent( in <-chan string, out chan<- string) {
    for word := range in {
        if isLetter(word) {
            out <- word
        }
    }
    close(out)
}
```

Fechamento de canais

Go implementa uma terceira operação básica, também como função implícita. para canais: **close**

A operação close envia um valor especial para o canal. Uma operação send posterior à **close** implica em **panic**

Operação receive feitas após **close** geram os valores restantes no canal até que todos sejam consumidos; após isso, outras chamadas retornam imediatamente como valor nulo do tipo em questão