CSP com golang 24.1

Universidade Federal de Campina Grande – UFCG
CEEI - UASC
Programação Concorrente
Professores: Thiago Emmanuel Pereira

A abordagem de concorrência de Go é baseada em CSP (communicating sequential processes).

Segundo esse conceito, programas são compostos por processos (no sentido de um fluxo de execução) que não compartilham estado; ao invés disso, processos se comunicam e sincronizam usando canais. Dito de outra forma, seguindo o lema de Go, “Share memory by communicating, don’t communicate by sharing memory.”

p.s1 - Read the classics: C. A. R. Hoare. 1978. Communicating sequential processes. Commun. ACM 21, 8 (August 1978), 666-677. DOI=http://dx.doi.org/10.1145/359576.359585

p.s2 - Além do modelo CSP, go também implementa primitivas de concorrência por memória compartilhada, como vimos em threads.

Aviso: Embora seja uma ótima linguagem, esse não é um curso de Go.

para se aprofundar

https://golang.org/

https://tour.golang.org/list

https://golang.org/doc/effective_go.html

https://gobyexample.com/

http://www.golangbootcamp.com/

https://github.com/golang/go/wiki/LearnConcurrency

https://www.golang-book.com/books/intro

roteiro

- goroutines
- channels
- select/cancelamento
- padrões concorrentes

goroutines

Em Go, todo fluxo de execução é uma goroutine. Em cada programa, há pelo menos uma goroutine.

Por enquanto, pense em goroutines como sendo uma *thread* (embora bem mais leve)

```go
package main

import (
    "fmt"
)

func main() {
    n := 42 //de maneira menos concisa var n int = 42
    result := fib(n)
    fmt.Printf("Fib(%d) = %d\n", n, result)
}

func fib(x int) int {
    if x < 2 {
        return x
    }
    return fib(x-1) + fib(x-2)
}
```

goroutines

Em Go, todo fluxo de execução é uma goroutine. Em cada programa, há pelo menos uma goroutine.

Por enquanto, pense em goroutines como sendo uma *thread* (embora bem mais leve).

Novas goroutines podem ser criadas com **go**. Seguem um modelo **fork-join**, tal como a system-call **fork** do UNIX (há um split entre pai-filho)

```go
package main

import (
    "fmt"
)

func main() {
    n := 42 //de maneira menos concisa var n int = 42
    result := fib(n)
    fmt.Printf("Fib(%d) = %d\n", n, result)
}

func fib(x int) int {
    if x < 2 {
        return x
    }
    return fib(x-1) + fib(x-2)
}
```

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    n := 42 //de maneira menos concisa var n int = 42

    go alert()
    result := fib(n)
    fmt.Printf("Fib(%d) = %d\n", n, result)
}

func fib(x int) int {
    if x < 2 {
        return x
    }
    return fib(x-1) + fib(x-2)
}

func alert() {
    for {
        fmt.Printf("demora. Ai!\n")
        time.Sleep(1 * time.Second)
    }
}
```

Este código exemplo estabelece um servidor TCP que responde com o horário atual

Clientes pode estabelecer conexão através de *netcat* ou telnet (p.ex *`nc localhost 8000`)*

O código só atende um cliente por vez. Como deixá-lo concorrente?

```go
// Adaptado de Alan A. A. Donovan & Brian W. Kernighan.
// a TCP server that periodically writes the time.
package main

import (
    "io"
    "log"
    "net"
    "time"
)

func main() {

    //escuta na porta 8000 (pode ser monitorado com lsof -Pn -i4 | grep 8000)
    listener, err := net.Listen("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    for {
        //aceita uma conexão criada por um cliente
        conn, err := listener.Accept()
        if err != nil {
            // falhas na conexão. p.ex abortamento
            log.Print(err)
            continue
        }
        // serve a conexão estabelecida
        handleConn(conn)
    }
}

func handleConn(c net.Conn) {
    defer c.Close()
    for {
        // envia o conteúdo servido na conexão
        _, err := io.WriteString(c, time.Now().Format("02:05:00\n"))
        if err != nil {
            // p.ex erro ao enviar os dados para um cliente que desconectou
            return
        }
        time.Sleep(1 * time.Second)
    }
}
```

Este código exemplo estabelece um servidor TCP que responde com o horário atual

Clientes pode estabelecer conexão através de *netcat* ou telnet (p.ex *`nc localhost 8000`)*

O código só atende um cliente por vez. Como deixá-lo concorrente?

Basta criar uma nova goroutine
E se fosse em java? E em C?

```go
// a TCP server that periodically writes the time.
package main

import (
    "io"
    "log"
    "net"
    "time"
)

func main() {

    //escuta na porta 8000 (pode ser monitorado com lsof -Pn -i4 | grep 8000)
    listener, err := net.Listen("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    for {
        //aceita uma conexão criada por um cliente
        conn, err := listener.Accept()
        if err != nil {
            // falhas na conexão. p.ex abortamento
            log.Print(err)
            continue
        }
        // serve a conexão estabelecida
        go handleConn(conn)
    }
}

func handleConn(c net.Conn) {
    defer c.Close()
    for {
        // envia o conteúdo servido na conexão
        _, err := io.WriteString(c, time.Now().Format("02:05:00\n"))
        if err != nil {
            // p.ex erro ao enviar os dados para um cliente que desconectou
            return
        }
        time.Sleep(1 * time.Second)
    }
}
```

Exercício 1

Quão "leves" são as goroutines? Quanto de memória uma goroutine ocupa? E 100, 1000, …? Em java/C, quanto "pesam" as *threads?*

Exercício 2

Implemente um programa em que uma **goroutine** gere valores aleatórios, enquanto uma segunda verifique se os valores gerados são pares ou ímpares, e os imprima na saída padrão

"math/rand"

rand.Int()

Como coordenar esse trabalho no estilo que estudamos anteriormente?

Canais

Vimos que goroutines implementam os sequential processes de C**SP**

Canal é a primitiva que permite que uma **goroutine** envie dados para outra (ou seja, se comuniquem, ao invés de usar memória compartilhada, como vimos com *threads)*

Um canal é um meio de comunicação de um determinado tipo (*element type)*

Canais são criados através da função ***make*** definida pela própria linguagem. Por exemplo, para **criar** um canal **ch** de **inteiros**:

ch := make(chan int)

Canais

A comunicação no canal se dá através de duas operações básicas: ***send*** e ***receive***

A operação **send** transmite um valor a partir de um **goroutine** através do canal, para uma outra **goroutine** que executa a operação **receive**

Em ambos os casos, usamos o operador **<-** para realizar a comunicação

```
ch := make(chan int)

//realiza a operação send do valor x para o canel ch
ch <- x
//realiza a operação receive e executa uma atribuição
//do valor recebido para a variável y
y = <- ch
//realiza a operação receive e descarta o valor recebido
<-ch
```

Exercício 2

Implemente um programa em que uma **goroutine** gere valores aleatórios, enquanto uma segunda verifique se os valores gerados são pares ou ímpares, e os imprima na saída padrão

"math/rand"

rand.Int()

Como usar canais para resolver esse problema?

Um padrão concorrente: pipeline

Canais são parte fundamental da estrutura de programas concorrentes

Por exemplo, são usados na construção de **pipelines,** um padrão recorrente na construção de programas concorrentes (vide o exercício anterior)

Em um pipeline, a saída de uma goroutine é usada como entrada para a próxima goroutine

Exercício 3

Construa um pipeline em que uma goroutine gere strings aleatórias, enquanto uma segunda filtre as strings que contém somente valores alpha, e uma terceira escreva os valores filtrados na saída padrão

```
package main

import (
    "fmt"
    "math/rand"
    "time"
    "unicode"
)

//below random string functions are based on Jon Calhoun code
const charset = "abcdefghijklmnopqrstuvwxyz1234567890"

var seededRand *rand.Rand = rand.New(
    rand.NewSource(time.Now().UnixNano()))

func StringWithCharset(length int, charset string) string {
    b := make([]byte, length)
    for i := range b {
        b[i] = charset[seededRand.Intn(len(charset))]
    }
    return string(b)
}

func RandString(length int) string {
    return StringWithCharset(length, charset)
}

func isLetter(s string) bool {
    for _, r := range s {
        if !unicode.IsLetter(r) {
            return false
        }
    }
    return true
}
```

Exercício 3

```
func generateContent(out chan string) {
    for {
        out <- RandString(5)
    }
}

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

func main() {
    rawContent := make(chan string)
    filteredContent := make(chan string)

    go generateContent(rawContent)
    go filterContent(rawContent, filteredContent)

    for {
        fmt.Printf("alpha: <%s>\n", <-filteredContent)
    }
}
```

```
package main

import (
    "fmt"
    "math/rand"
    "time"
    "unicode"
)

//below random string functions are based on Jon Calhoun code
const charset = "abcdefghijklmnopqrstuvwxyz1234567890"

var seededRand *rand.Rand = rand.New(
    rand.NewSource(time.Now().UnixNano()))

func StringWithCharset(length int, charset string) string {
    b := make([]byte, length)
    for i := range b {
        b[i] = charset[seededRand.Intn(len(charset))]
    }
    return string(b)
}

func RandString(length int) string {
    return StringWithCharset(length, charset)
}

func isLetter(s string) bool {
    for _, r := range s {
        if !unicode.IsLetter(r) {
            return false
        }
    }
    return true
}
```

Exercício 3

```go
func generateContent(out chan string) {
    for {
        out <- RandString(5)
    }
}

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

func main() {
    rawContent := make(chan string)
    filteredContent := make(chan string)

    go generateContent(rawContent)
    go filterContent(rawContent, filteredContent)

    for {
        fmt.Printf("alpha: <%s>\n", <-filteredContent)
    }
}
```

Neste exercício, a goroutine generateContent executa indefinidamente.

Como limitar essa execução à, digamos, 100 palavras randômicas geradas?

Exercício 3

```
func generateContent(out chan string) {
    for i := 0; i < 100; i++ {
        out <- RandString(5)
    }
}

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

func main() {
    rawContent := make(chan string)
    filteredContent := make(chan string)

    go generateContent(rawContent)
    go filterContent(rawContent, filteredContent)

    for {
        fmt.Printf("alpha: <%s>\n", <-filteredContent)
    }
}
```

Neste exercício, a goroutine generateContent executa indefinidamente.

Como limitar essa execução à, digamos, 100 palavras randômicas geradas?

OK? Qual o problema?

Exercício 3

```go
func generateContent(out chan string) {
    for i := 0; i < 100; i++ {
        out <- RandString(5)
    }
}

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

func main() {
    rawContent := make(chan string)
    filteredContent := make(chan string)

    go generateContent(rawContent)
    go filterContent(rawContent, filteredContent)

    for {
        fmt.Printf("alpha: <%s>\n", <-filteredContent)
    }
}
```

Neste exercício, a goroutine generateContent executa indefinidamente.

Como limitar essa execução à, digamos, 100 palavras randômicas geradas?

OK? Qual o problema?

```
fatal error: all goroutines are asleep - deadlock!

        goroutine 1 [chan receive]:
        main.main()
                pipeline.go:60 +0xeb

        goroutine 6 [chan receive]:
        main.filterContent(0xc42001c060, 0xc42001c0c0)
                pipeline.go:45 +0x57
        created by main.main
                pipeline_done.go:57 +0xb5
```

Canais

Como uma **goroutine** consumidora (receiver) pode ser notificada que a **goroutine** produtora (sender) não enviará outros dados, e portanto, não precisa mais esperar para receber no canal?

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

Exercício 3

```go
func generateContent(out chan string) {
    for i := 0; i < 100; i++ {
        out <- RandString(5)
    }
    close(out)
}

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

func main() {
    rawContent := make(chan string)
    filteredContent := make(chan string)

    go generateContent(rawContent)
    go filterContent(rawContent, filteredContent)

    for {
        fmt.Printf("alpha: <%s>\n", <-filteredContent)
    }
}
```

Neste exercício, a goroutine generateContent executa indefinidamente.

Como limitar essa execução à, digamos, 100 palavras randômicas geradas?

OK? Qual o problema?

E agora?

Exercício 3

```go
func generateContent(out chan string) {
    for i := 0; i < 100; i++ {
        out <- RandString(5)
    }
    close(out)
}

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

func main() {
    rawContent := make(chan string)
    filteredContent := make(chan string)

    go generateContent(rawContent)
    go filterContent(rawContent, filteredContent)

    for {
        fmt.Printf("alpha: <%s>\n", <-filteredContent)
    }
}
```

Neste exercício, a goroutine generateContent executa indefinidamente.

Como limitar essa execução à, digamos, 100 palavras randômicas geradas?

OK? Qual o problema?

E agora?

Não temos mais um deadlock. Temos um livelock :)

```go
func generateContent(out chan string) {
    for i := 0; i < 100; i++ {
        out <- RandString(5)
    }
    close(out)
}

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

func main() {
    rawContent := make(chan string)
    filteredContent := make(chan string)

    go generateContent(rawContent)
    go filterContent(rawContent, filteredContent)

    for {
        alpha, ok := <-filteredContent
        if !ok {
            break
        }
        fmt.Printf("alpha: <%s>\n", alpha)
    }
}
```

```go
func generateContent(out chan string) {
    for i := 0; i < 100; i++ {
        out <- RandString(5)
    }
    close(out)
}

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

func main() {
    rawContent := make(chan string)
    filteredContent := make(chan string)

    go generateContent(rawContent)
    go filterContent(rawContent, filteredContent)

    for alpha := range filteredContent {
        fmt.Printf("alpha: <%s>\n", alpha)
    }
}
```

range encapsula a verificação de fechamento do canal. Código muito mais simples (note que nem sempre os canais precisam ser fechados, somente quando queremos sinalizar para seus consumidores)

Canais

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

COMUNICAÇÃO COM CANAIS NÃO-BUFFERIZADOS SINCRONIZAM GOROUTINES (por isso, esses canais também são chamados de canais síncronos)

No jargão de programação concorrente, dizemos que o envio do valor para o canal ***happens-before*** o despertar da **goroutine** que estiver bloqueada esperando para receber os valores do canal

Quando a ação **A** *happens-before* a ação **B**, temos não somente que **A** acontece antes temporalmente mas também que todas as operações feitas antes disso, por exemplo, atualização de variáveis, foram completadas e se pode confiar no valor atual delas. Como vimos antes, a **visibilidade é garantida.**

canais unidirecionais

```go
func generateContent(out chan string) {
    for i := 0; i < 100; i++ {
        out <- RandString(5)
    }
    close(out)
}

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

func main() {
    rawContent := make(chan string)
    filteredContent := make(chan string)

    go generateContent(rawContent)
    go filterContent(rawContent, filteredContent)

    for alpha := range filteredContent {
        fmt.Printf("alpha: <%s>\n", alpha)
    }
}
```

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

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

select

Considere, em um sistema grande, que módulos desse sistema se integram através de canais. Por exemplo, um *gateway* que se conecta à dois serviços HTTP (além de tratar requisições de seus próprios clientes).

Nesse caso, digamos que há 3 canais. O *gateway* deve reagir, quando receber quando evento em um desses canais. Como ficaria o *"main loop"* do *gateway?*

forma geral

```
select {
    case <- ch1:
        // executa o receive em ch1 e descarta o valor
    case x := <-ch2:
        // executa o receive em ch2 e atribui em x
    case ch3 <- y:
        // executa send em ch3
    default:
        // claúsula opcional
}
```

Cada ***case*** descreve uma comunicação possível e um bloco de declarações associado com cada uma das comunicações especificadas.

O ***select*** esperará que uma das comunicações esteja pronta para ocorrer e em seguida, executa o bloco de declarações associado. As demais comunicações **não** acontecerão.

Caso múltiplas comunicações estiverem prontas para ocorrer, uma delas é escolhida aleatoriamente

polling

```
select {
    case <- ch1:
        // executa o receive em ch1 e descarta o valor
    case x := <-ch2:
        // executa o receive em ch2 e atribui em x
    case ch3 <- y:
        // executa send em ch3
    default:
        // claúsula opcional
}
```

O que fazer caso demore muito para que um ***send*** ou ***receive*** aconteça (ou mesmo, não aconteça)?

polling

```
select {
    case <- ch1:
        // executa o receive em ch1 e descarta o valor
    case x := <-ch2:
        // executa o receive em ch2 e atribui em x
    case ch3 <- y:
        // executa send em ch3
    default:
        // claúsula opcional
}
```

O que fazer caso demore muito para que um ***send*** ou ***receive*** aconteça (ou mesmo, não aconteça)?

Lembram do *tryLock()*?

Em um ***select***, caso não exista nenhuma comunicação pronta, será executa a cláusula ***default*** (caso tenha sido escrita).

timed polling

É possível também executar um *Timed Polling*, tal como fazíamos com `tryLock(long timeout, TimeUnit unit)`

Em Go, podemos usar a função **`time.Tick`**. Essa função retorna um canal e, implicitamente, envia um valor para o canal após **`n`** unidades de tempo. P.ex, poder ter **`time.Tick(1 * time.Second)`**

Como ficaria o código para um **select** em que se espera por no máximo segundos até que uma mensagem num canal **ch** seja recebida?

timed polling

```
tick := time.Tick(10 * time.Second)

select {
    case <-tick:
        //timeout
    case <- ch:
        //receive no canal ch
}
```

É possível também executar um *Timed Polling*, tal como fazíamos com `tryLock(long timeout, TimeUnit unit)`

Em Go, podemos usar a função **`time.Tick`**. Essa função retorna um canal e, implicitamente, envia um valor para o canal após **n** unidades de tempo. P.ex, poder ter **`time.Tick(1 * time.Second)`**

Como ficaria o código para um **select** em que se espera por no máximo segundos até que uma mensagem num canal **ch** seja recebida?

timed polling + max_tries

Um padrão comum é esperar uma ação acontecer, por no máximo **max_tries** tentativas.

Como modificar o exemplo anterior para que tenhamos no máximo 5 tentativas, onde esperamos 30 segundos em cada tentativa?

timed polling + max_tries

```
tick := time.Tick(30 * time.Second)

for try := 5; try > 0; try-- {
    select {
        case <-tick:
            //timeout
        case <- ch:
            //receive no canal ch
    }
}
```

Um padrão comum é esperar uma ação acontecer, por no máximo **max_tries** tentativas.

Como modificar o exemplo anterior para que tenhamos no máximo 5 tentativas, onde esperamos 30 segundos em cada tentativa?

Cuidado com goroutine leaks!

cancelamento

tl;dr Como um goroutine pode cancelar a execução de outra(s)?

cancelamento

tl;dr Como um goroutine pode cancelar a execução de outra(s)?

- Seguido o modelo de comunicação entre goroutines, podemos enviar uma mensagem através de um canal.
- Mas, se precisarmos cancelar um número arbitrário de goroutines, como podemos saber quantas mensagens de cancelamento devemos mandar no canal?

cancelamento

tl;dr Como um goroutine pode cancelar a execução de outra(s)?

- Seguido o modelo de comunicação entre goroutines, podemos enviar uma mensagem através de um canal.
- Mas, se precisarmos cancelar um número arbitrário de goroutines, como podemos saber quantas mensagens de cancelamento devemos mandar no canal?

cancelamento

```go
package main

import (
    "os"
    "time"
    "fmt"
    "math/rand"
)

func msg(id int) {
    for {
        fmt.Printf("Ui <%d>\n", id)
        time.Sleep(1 * time.Second)
    }
}

func killer() {
    //how to kill the msg goroutines?

    //read from input stream
    os.Stdin.Read(make([]byte, 1))
}

func main() {
    done := make(chan interface{})

    go killer()

    r := rand.New(rand.NewSource(time.Now().UnixNano()))
    var ranN = r.Intn(35)
    fmt.Printf("%d gourotines\n", ranN)

    for i := 0; i < ranN; i++ {
        go msg(i)
    }

    time.Sleep(1 * time.Minute)
}
```

```go
package main

import (
    "os"
    "time"
    "fmt"
    "math/rand"
)

func msg(id int) {
    for {
        fmt.Printf("Ui <%d>\n", id)
        time.Sleep(1 * time.Second)
    }
}

func killer() {
    //how to kill the msg goroutines?

    //read from input stream
    os.Stdin.Read(make([]byte, 1))
}

func main() {
    done := make(chan interface{})

    go killer()

    r := rand.New(rand.NewSource(time.Now().UnixNano()))
    var ranN = r.Intn(35)
    fmt.Printf("%d gourotines\n", ranN)

    for i := 0; i < ranN; i++ {
        go msg(i)
    }

    time.Sleep(1 * time.Minute)
}
```

```go
package main

import (
    "fmt"
    "math/rand"
    "os"
    "time"
)

func msg(id int, done chan interface{}) {
    for {
        select {
        case <-done:
            fmt.Println("I'm going to die")
            return
        default:
            fmt.Printf("Ui <%d>\n", id)
            time.Sleep(1 * time.Second)
        }
    }
}

func killer(done chan interface{}) {
    //how to kill the msg goroutines?

    //read from input stream
    os.Stdin.Read(make([]byte, 1))
    close(done)
}

func main() {
    done := make(chan interface{})

    go killer(done)

    r := rand.New(rand.NewSource(time.Now().UnixNano()))
    var ranN = r.Intn(35)
    fmt.Printf("%d gourotines\n", ranN)

    for i := 0; i < ranN; i++ {
        go msg(i, done)
    }

    time.Sleep(1 * time.Minute)
}
```