PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL – ESCOLA POLITÉCNICA
Fundamentos de Processamento Paralelo e Distribuído – Prof. Fernando Dotti
Lista de Exercícios

1. Explique a razao do código abaixo gerar resultado diferente de 0 (zero). Exemplifique e explique uma situação em que o valor de sharedTest torna-se inconsistente.

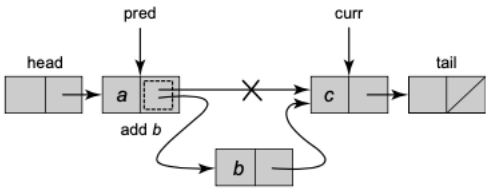
```
var sharedTest int = 0 // variavel compartilhada
var ch_fim chan struct{} = make(chan struct{})
func MyFunc(inc int) {
    for k := 0; k < 1000; k++ {
        sharedTest = sharedTest + inc
    }
    ch_fim <- struct{}{}
}

func main() {
    for i := 0; i < 10; i++ {
        go MyFunc(1)
        go MyFunc(-1)
    }
    fmt.Println("Criei 20 processos")
    for i := 0; i < 20; i++ {
        <-ch_fim
    }
    fmt.Println("Processos acabaram. Resultado  ", sharedTest)
}
```

2. Considere uma lista de elementos, em que os mesmos são inseridos em posições conforme sua ordem. A lista é mantida do menor para o maior elemento.

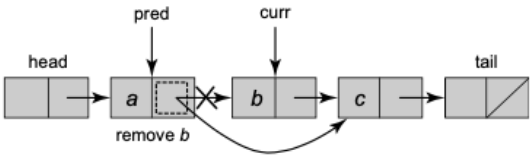
O método de inserção (a): procura o local a inserir, cria um nodo adicional, e o liga à estrutura existente, como na figura (a). (figuras do livro *The Art of Multiprocessor Programming, M.Herlihy e Nir Shavit*)

O método de remoção (b): procura o elemento na lista e, se ele estiver, remove-o, como na figura (b).

(a)



(b)



Suponha que duas threads invocam concorrentemente os métodos (a) e (b) sobre uma lista. Aponte uma situação em que a lista resultante ficará inconsistente devido à concorrência das operações.

3. Considere o seguinte algoritmo como tentativa de resolução da seção crítica por software. Assuma que cada linha é atômica,

boolean wantp ← false, wantq ← false	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wantp ← true	q2: wantq ← true
p3: while wantq	q3: while wantp
p4: wantp ← false	q4: wantq ← false
p5: wantp ← true	q5: wantq ← true
p6: critical section	q6: critical section
p7: wantp ← false	q7: wantq ← false

Discuta se as seguintes propriedades são satisfeitas pelo mesmo, justificando:

a. Exclusão Mútua
b. Progresso
c. Não postergação

4. Considere o seguinte algoritmo para sessão crítica utilizando a operação atômica (em HW) *exchange*, que troca os valores das variáveis parâmetro.

integer common ← 1	
p	**q**
integer local1 ← 0	integer local2 ← 0
loop forever	loop forever
p1: non-critical section	q1: non-critical section
repeat	repeat
p2: exchange(common, local1)	q2: exchange(common, local2)
p3: until local1 = 1	q3: until local2 = 1
p4: critical section	q4: critical section
p5: exchange(common, local1)	q5: exchange(common, local2)

Discuta se as seguintes propriedades são satisfeitas pelo mesmo, justificando:

a. Exclusão Mútua
b. Progresso
c. Não postergação

5. Monte os diagramas de estados e transições para cada programa abaixo, onde os estados são dados pelos diferentes valores das tuplas [x,y] ou [x,y,z] conforme o caso.

```
var x, y int = 0, 0

func px() {
    x = 1
    x = 2
 }

func py() {
    y = 1
    y = 2
 }

func main() { // QUESTAO A
   go pX()
   go py()
   <-make(chan struct{}, 0)
}
```

```
var x, y int = 0, 0
    semaforo s = novoSem(0)

func px() {
    x = 1
    x = 2
    s.signal()
 }

func py() {
    s.wait()
    y = 1
    y = 2
 }

func main() { // QUESTAO B
   go pX()
   go py()
   <-make(chan struct{}, 0)
}
```

```
var x, y int = 0, 0
    semaforo s = novoSem(0)

func px() {
    x = 1
    s.signal()
    x = 2
 }

func py() {
    y = 1
    s.wait()
    y = 2
 }

func main() { // QUESTAO C
   go pX()
   go py()
   <-make(chan struct{}, 0)
}
```

```
var x, y int = 0, 0
    semaforo s1 = novoSem(0)
    semaforo s2 = novoSem(0)

func px() {
    x = 1
    s1.signal()
    s2.wait()
    x = 2
 }

func py() {
    y = 1
    s1.wait()
    s2.signal()
    y = 2
 }

func main() { // QUESTAO D
   go pX()
   go py()
   <-make(chan struct{}, 0)
}
```

```
var x, y, z int = 0, 0, 0
    semaforo s = novoSem(0)

func px() {
    x = 1
    x = 2
    s.signal()
 }

func py() {
    y = 1
    y = 2
    s.signal()
 }

func pz() {
    s.wait()
    s.wait()
    z = 1
 }

func main() { // QUESTAO E
   go pX()
   go py()
   go pz()
   <-make(chan struct{}, 0)
}
```

```
var x, y, z int = 0, 0, 0
    semaforo s = novoSem(0)

func px() {
    x = 1
    x = 2
    s.signal()
 }

func py() {
    y = 1
    y = 2
    s.signal()
 }

func pz() {
    s.wait()
    z = 1
    s.wait()
    z = 2
 }

func main() { // QUESTAO F
   go pX()
   go py()
   go pz()
   <-make(chan struct{}, 0)
}
```

6. Suponha que você dispõe de canais e precisa de uma implementação de semáforos, usando as operações s=NewSemaphore(_); s.Wait() e s.Signal() conforme visto em aula. A seguinte implementação é sugerida em algum lugar da internet e nem tudo funciona como deveria. Qual a razão ?

```
// --------------------------------
type Semaphore struct {
    sChan chan struct{}
}
func NewSemaphore(init int) *Semaphore {
    s := &Semaphore{
        sChan: make(chan struct{}, init),
    }
    return s
}
func (s *Semaphore) Wait() {
    s.sChan <- struct{}{}
}
func (s *Semaphore) Signal() {
    <- s.sChan
}
// --------------------------------
```

7.Dado o seguinte código utilizando semáforos, substitua semáforos pelo uso de canais.

```
func semaSC() {
    var sharedTest int = 0
    var ch_fim chan struct{} =
                    make(chan struct{})
    var sem *MCCSemaforo.Semaphore =
              MCCSemaforo.NewSemaphore(1)

    for i := 0; i < 100; i++ {
        go func() {
            for k := 0; k < 100; k++ {
                sem.Wait()
                sharedTest = sharedTest + 1
                sem.Signal()
            }
            ch_fim <- struct{}{}
        }()
    }
    for i := 0; i < 100; i++ {
        <-ch_fim
    }
    fmt.Println("Resultado ", sharedTest)
}
```

8. O problema dos leitores e escritores apresenta uma situação em que um recurso pode ser lido por processos leitores e escrito por processos escritores. *Processos leitores podem acessar o recurso concorrentemente com outros leitores. Processos escritores acessam somente em exclusão mútua com relação a qualquer outro processo, seja leitor ou escritor.* Abaixo, está uma estrutura do algoritmo concorrente para os leitores e escritores. Somente o algoritmo do leitor está incompleto. O restante não necessita modificações.

a. Complete o algoritmo do processo leitor onde indicado em comentários, usando as declarações existentes. (Se quiser, voce pode declarar novas variáveis, mas não seria necessário).
b. Indique se sua solução tem postergaçao indefinida ou não, que tipo de processo pode ser postergado, e a razão.

```
package main

var contLeitores int = 0
var mx := NewSemaphore(1)
var recurso := NewSemaphore(1)

func escritor() {
    for {
        recurso.Wait()
        fmt.Println("Escreve")
        recurso.Signal()
    }
}

func main() {
    for i := 0; i < 10; i++ {
        go escritor()
        go leitor()
    }
    <-make(chan struct{})
}
```

```
func leitor() {
 for {
   // codigo para um leitor acessar recurso concorrentemente
   // com outros leitores.   COMPLETE









   fmt.Println("aqui acontece cesso ao recurso")
   // codigo para um leitor sair do recurso.   COMPLETE.









  }
}
```

9. Um programa tem um conjunto de N threads ou go-rotinas. Todas estas threads tem o mesmo código. Neste código elas tem vários pontos críticos (PCR), que são trechos de código com requisitos de sincronização, conforme a seguir.

Toda thread i, i de 1 a N, tem a seguinte estrutura {
  Loop {
    Processamento local

    PCR1: trecho de código em que no máximo 10 processos executam concorrentemente

    PCR2: trecho de código em que no máximo 5 processos executam concorrentemente
  }
}

Apresente uma solução de sincronização com (a) semáforos e outra com (b) canais para que os requisitos dos pontos críticos sejam garantidos.

10. Abaixo segue uma modelagem do Jantar dos Filósofos utilizando o conceito de monitor (da literatura).

```
monitor DP
{
    status state[5];
    condition self[5];

    Pickup(int i) {
        state[i] = hungry;
        test(i);
        if (state[i] != eating)
            self[i].wait;
    }

    Putdown(int i) {
        state[i] = thinking;
        test((i + 1) % 5);
        test((i + 4) % 5);
    }
```

```
    test(int i) {
     if (state[(i + 1) % 5] != eating
            && state[(i + 4) % 5] != eating
            && state[i] == hungry) {
              state[i] = eating;
              self[i].signal();
        }
    }

    init()
    {
      for i = 0 to 4
          state[i] = thinking;
    }
}
```

Com este monitor, cada thread filósofo(i,DP), i em 0..4, é como segue:

```
Filosofo(int i, DP dp){
    loop {
            ...
            dp.Pickup(i);
            // eat
            do.Putdown(i);
          }
 }
```

```
filosofos main {
   var dp =  cria monitor DP
   para i em 0 até 4
        inicia thread Filósofo(i, dp)
   aguarda final de threads
}
```

A) Esta modelagem tem deadlock ? Argumente mostrando que sim ou que não, conforme sua resposta.

B) Nesta modelagem, algum filósofo pode ser indefinidamente postergado ? Argumente mostrando que sim ou que não, conforme sua resposta.