Modelos para Computação Concorrente ou Sistemas Operacionais

Memória Compartilhada - O Problema da Seção Crítica – Semáforos

(com slides de Ben-Ari)

Fernando Luís Dotti

PUCRS | ESCOLA POLITÉCNICA

Bibliografia Base

[disponível na biblioteca]

M. Ben-Ari

Principles of Concurrent and Distributed Programming

Second Edition

Addison-Wesley, 2006

- Semáforos
  - Construção de sincronização com
    - V: valor inteiro não negativo
    - L: lista de processos
    - Operações atômicas Wait [ou P] e signal [ou V], onde

```
S(val,list) { v:=val; l:=list}   /* construtor */
wait(s): if (s.v > 0 )
            then s.c := s.v -1
            else {
                    s.l := s.l U p  /* adicionar o processo na lista s.l */
                    block(p) ;     /*bloqueia o processo p no SO*/
             }
 signal(s): if (s.l = {} )
             then s.v := s.v+1
               else {
               tome um elemento q de l
               s.l := s.l – q  /*remover o processo "q" da lista s.l */
               wakeup(q)  /* acorda o processo p no SO */
                }
```

Scenario for Starvation

n	Process p	Process q	Process r	S
1	**p1: wait(S)**	q1: wait(S)	r1: wait(S)	$(1, \emptyset)$
2	p2: signal(S)	**q1: wait(S)**	r1: wait(S)	$(0, \emptyset)$
3	p2: signal(S)	q1: blocked	**r1: wait(S)**	$(0, \{q\})$
4	**p1: signal(S)**	q1: blocked	r1: blocked	$(0, \{q, r\})$
5	**p1: wait(S)**	q1: blocked	r2: signal(S)	$(0, \{q\})$
6	p1: blocked	q1: blocked	**r2: signal(S)**	$(0, \{p, q\})$
7	p2: signal(S)	q1: blocked	**r1: wait(S)**	$(0, \{q\})$

```
S(val,list) { v:=val; l:=list }          /* construtor : s = S(_,{}) – lista sempre é vazia no início */
wait(s): if (s.v > 0 )  then s.c := s.v -1   /* se v > 0, decrementa; senao ...}
         else  s.l := s.l  U  p          /* adicionar o processo na lista s.l – necessidade de tratar como FIFO */
               block(p) ;                /*bloqueia o processo p no SO*/
signal(s): if (s.l = {} ) then s.v := s.v+1   /* se ninguem bloqueado, incrementa v, senao ... */
           else  wakeup(head(s.l))       /* acorda primeiro processo (no SO) – necessidade de tratar como FIFO */
                 s.l := tail(s.l)        /*remove o processo da lista s.l */
```

- Semáforos fortes
  - Construção de sincronização com
    - V: valor inteiro não negativo
    - L: lista de processos administrada como fila
    - Operações atômicas Wait [ou P] e signal [ou V], onde

```
S(val) { v:=val; l:=emptyList}   /* construtor */
s.wait(): atomic {
                if (s.v > 0 )
                then s.v := s.v -1
                else {
                     s.l := append(s.l,p)  /* adicionar o processo no final da fila s.l */
                     block(p);     /*bloqueia o processo p no SO*/
                }
            }
 s.signal(): atomic {
                  if (s.l = emptyList )
                   then s.v := s.v+1
                  else {
                       q := head(s.l)  /* tome o primeiro elemento */
                       s.l := tail(s.l)  /*remover o processo "q" da lista s.l */
                       wakeup(q)  /* acorda o processo p no SO */
                   }
                }
```

```
S(val,list) { v:=val; l:=list }                  /* construtor  :  tipicamente s = S(_,{}) */
wait(s): if (s.v > 0 )   then s.c := s.v -1      /* se v > 0, decrementa; senao ...}
            else   s.l := s.l  U  p              /* adicionar o processo na lista s.l */
                   block(p) ;                    /*bloqueia o processo p no SO*/
signal(s):  if (s.l = {} ) then s.v := s.v+1     /* se ninguem bloqueado, incrementa v, senao ... */
              else  wakeup(head(s.l))            /* acorda primeiro processo (no SO) */
                    s.l := tail(s.l)             /*remove o processo da lista s.l */
```

```
S(val) { v:=val; l:= emptyList}          /* construtor  :  s = S(_,{}) – lista sempre é vazia no início */
wait(s): if (s.v > 0 )   then s.v := s.v -1   /* se v > 0, decrementa; senao ...}
          else   s.l := s.l  U  p            /* adicionar o processo na lista s.l */
                 block(p) ;                  /*bloqueia o processo p no SO*/
signal(s):  if (s.l = {} ) then s.v := s.v+1  /* se ninguem bloqueado, incrementa v, senao ... */
            else  wakeup(head(s.l))           /* acorda primeiro processo (no SO) */
                  s.l := tail(s.l)            /*remove o processo da lista s.l */
```

Algorithm 6.1: Critical section with semaphores (two processes)	
binary semaphore S $\leftarrow (1, \emptyset)$ p1, q1 :(1, { })	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wait(S)	q2: wait(S)
p3: critical section	q3: critical section
p4: signal(S)	q4: signal(S)

Algorithm 6.1: Critical section with semaphores (two processes)	
binary semaphore S $\leftarrow (1, \emptyset)$	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wait(S)	q2: wait(S)
p3: critical section	q3: critical section
p4: signal(S)	q4: signal(S)

S=(1,{})

p1: ...

q1: ...

p2: wait(S)

q2: wait(S) // um entra antes ... ex: p

p2: wait(S)

S=(0,{})

p3: c.s. ....

q2: wait(S)

S=(0,{q})

p3: c.s. ....

q2: blocked

p3: c.s. ....

q2: blocked

p4: signal(S)

S=(0,{}) q is unblocked and completes wait(S)

p1: ...

q3: c.s. ....

p1: ...

q3: c.s. ....

p1: ...

q3: signal(S)

S=(1,{})

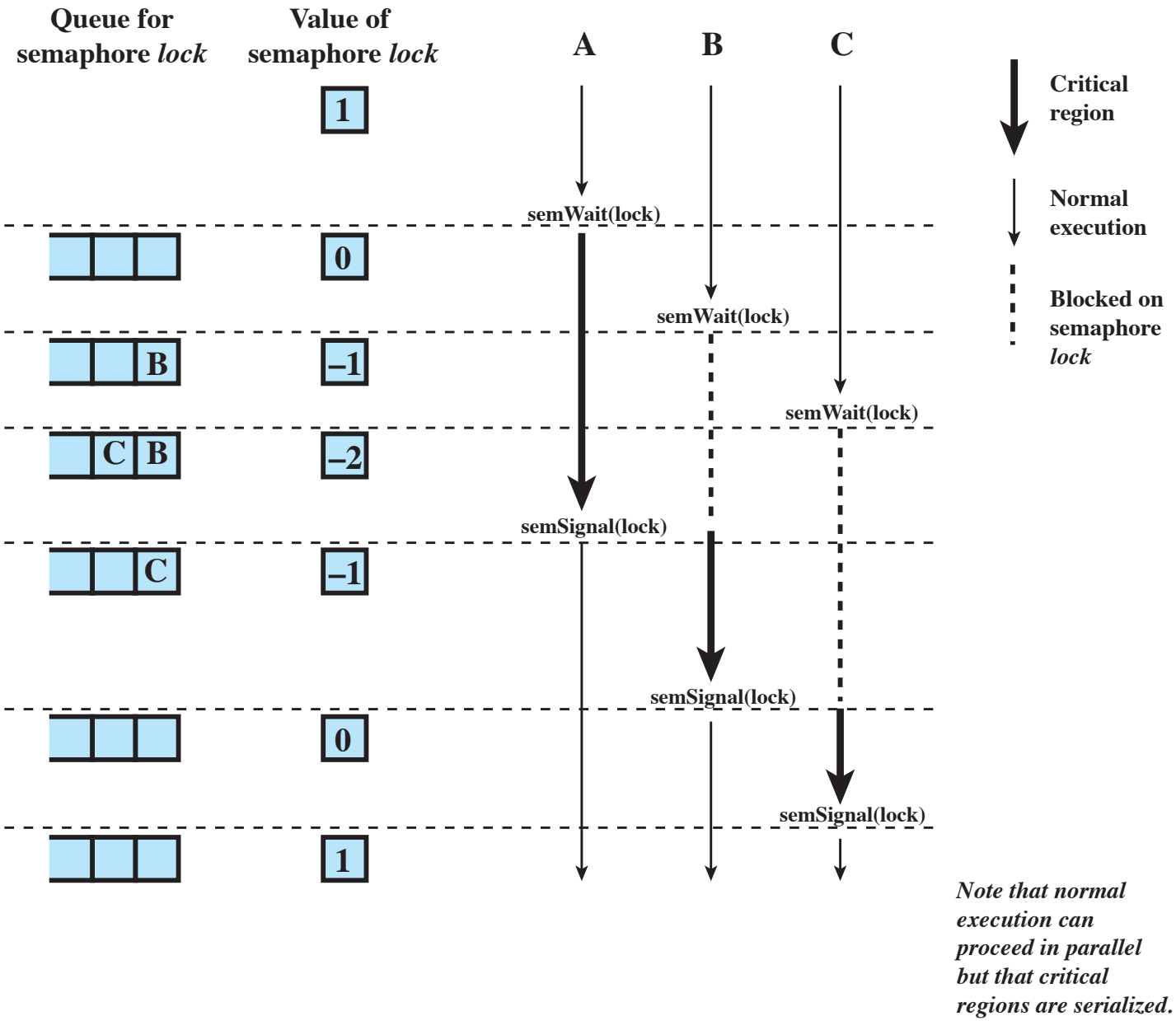
```
S(val,list) { v:=val; l:=list }                 /* construtor  :  s = S(_,{}) – lista sempre é vazia no início */
 wait(s): if (s.v > 0 )   then s.c := s.v -1    /* se v > 0, decrementa; senao ...} 
          else   s.l := s.l  U  p               /* adicionar o processo na lista s.l */
                 block(p) ;                     /*bloqueia o processo p no SO*/
 signal(s):  if (s.l = {} ) then s.v := s.v+1   /* se ninguem bloqueado, incrementa v, senao ... */
             else  wakeup(head(s.l))            /* acorda primeiro processo (no SO) */
                   s.l := tail(s.l)             /*remove o processo da lista s.l */
```

Algorithm 6.4: Critical section with semaphores (N proc., abbrev.)

binary semaphore S $\leftarrow (1, \emptyset)$

```
      loop forever
p1:     wait(S)
p2:     signal(S)
```

Exemplo de variante: semáforo permite valor negativo (= nro processos bloqueados)



Semáforos - atomicidade

Como operações Wait e Signal do semáforo são garantidas como atômicas?

- sistema de único processador
  - opção de desabilitar interrupções no início da operação
- um ou mais processadores
  - uso de soluções de SW (spin-lock, busy-wait)
  - uso de operações de HW
  - observação: o uso de operações com espera ocupada representa overhead menor pois a região atômica é curta e rápida (operação wait e signal) e não uma operação de maior complexidade de uma aplicação genérica.

Semáforos - atomicidade

- observação: o uso de operações com espera ocupada representa overhead menor pois a região atômica é curta e rápida (operação wait e signal) e não uma operação de maior complexidade de uma aplicação genérica.

```
S(val,list) { v:=val; l:=list }
```

```
wait(s):
        entradaSC
        if (s.v > 0 )
        then s.c := s.v -1
        else   s.l := s.l  U  p
               block(p) ;
        saidaSC
```

```
signal(s):
        entradaSC
        if (s.l = {} )
        then s.v := s.v+1
        else  wakeup(head(s.l))
              s.l := tail(s.l)
        saidaSC
```

entradaSC e *saidaSC* podem utilizar TAS, CAS, ou mesmo solução de SW

Semáforos em Java
uso da abstração

```java
/*
   Exemplo de exclusao mutua com semaforo, em Java.
   Um contador compartilhado.
   PUCRS - Escola Politecnica
   Prof: Fernando Dotti

   as operacoes        acquire e  release
   sao equivalentes a  wait    e  signal   da literatura
*/
import java.util.concurrent.Semaphore;

// a classe abaixo especifica um contador onde cada incremento
// faz a protecao para exclusao mutua.   Ela usa um semaforo
// iniciado em 1 para isso.
class CounterSema {
  private  int n;
  private Semaphore s;

  public CounterSema(){
        n = 0;
        s = new Semaphore(1);
  }

  public void incr(int id){
    try {
       s.acquire();    //   WAIT
    } catch (InterruptedException ie) {}
       n++;            //   codigo da SC
       s.release();    //   SIGNAL
  }

  public int value() { return n; }
}

// abaixo apenas implementamos threads que invocam o
// incremento em um objeto CounterSema
class CounterThread extends Thread {
  private int id;
  private CounterSema c_s;
  private int limit;

    public CounterThread(int _id, CounterSema _c_s, int _limit){
       id = _id;  c_s = _c_s;  limit = _limit;
    }

    public void run() {
       for (int i = 0; i < limit; i++) {  c_s.incr(id);   }
    }
}
```

```java
// o Teste cria um unico objeto CounterSema e passa para 5 threads
// que incrementam o mesmo nrIncr vezes (100000 como exemplo)
class TesteSemaphore {
  public static void main(String[] args) {

    int nrIncr = 100000;

    CounterSema c = new CounterSema();

    // criamos os objetos - ainda nao executam
      CounterThread p = new CounterThread(0,c,nrIncr);
      CounterThread q = new CounterThread(1,c,nrIncr);
      CounterThread r = new CounterThread(2,c,nrIncr);
      CounterThread s = new CounterThread(3,c,nrIncr);
      CounterThread t = new CounterThread(4,c,nrIncr);

    // aqui dispara as threads!!
    // o start dispara o metodo run de cada thread
    // como um processo concorrente aos demais
      p.start();
      q.start();
      r.start();
      s.start();
      t.start();
    // aqui temos main + 5 threads ativas
      try { p.join(); q.join(); r.join(); s.join(); t.join(); }

    // aqui temos somente main ativa
      catch (InterruptedException e) { }
      System.out.println("The value of n is " + c.value());
    }
}
```

Semáforos em Go

Concorrência em Go – sincronização semáforos

A razão de Go não oferecer o semáforo básico é que estes tem uma analogia direta com canais

Veja a seguir duas implementação de semáforos em Go

- A primeira usa canais sincronizantes
- A segunda usa um canal buferizado

Concorrência em Go – semáforos


```
package main
import (
        "fmt"
)
type Semaphore struct {
        wai, sig chan struct{}
        val     int
}
func NewSemaphore() *Semaphore {
        s := &Semaphore{
                wai: make(chan struct{}),
                sig: make(chan struct{}),
                val: 0}
        go func() {   // comportamento do semáforo
                for {
                        if s.val == 0 {    // se val == 0
                                <-s.sig     // permite signal
                                s.val++    // processo fazendo wait bloqueia
                         }
                         if s.val > 0 {
                                 select {
                                     case <-s.sig:    //permite signal
                                                s.val++
                                     case <-s.wai:   //permite wait
                                                s.val--
                                     }
                           }
                }
        }()
        return s
}
```

```
func (s *Semaphore) Wait() {
        s.wai <- struct{}{}
}

func (s *Semaphore) Signal() {
        s.sig <- struct{}{}
}
// um exemplo de uso de semaforo
func useSC(shared *int, s *Semaphore, st string) {
        for {
                s.Wait()
                *shared = ((*shared) + 1) % 100000
                s.Signal()
                fmt.Print(st + " ")
                fmt.Println(*shared)
        }
}
func main() {
        s := NewSemaphore()
        sh := 0
        go useSC(&sh, s, "a")
        go useSC(&sh, s, "b")
        go useSC(&sh, s, "c")
        go useSC(&sh, s, "d")
        s.Signal() // incrementa semaforo para 1
        var blq chan int = make(chan int)
        <-blq
}
```

Concorrência em Go – semáforos

```
package main

import (
        "fmt"
)
const MaxInt = 32767
type Semaphore struct {
        sChan chan struct{}
}
func NewSemaphore() *Semaphore {
        s := &Semaphore{
                sChan: make(chan struct{}, MaxInt),
        }
        return s
}
// semaforo é um canal bufferizado!
// valor do semaforo é numero de itens no canal
```

Wait e signal em semáforo tem mesmo comportamento de leitura e escrita em um canal bufferizado, com buffer hipoteticamente ilimitado

canal deve ser incializado com tantos itens quantos créditos do semáforo.

```
func (s *Semaphore) Wait() {
        <- s.sChan
}       //  wait é uma leitura do canal

func (s *Semaphore) Signal() {
        s.sChan <- struct{}{}
}       // signal é uma escrita
// mesmo exemplo de uso de semaforo
func useSC(shared *int, s *Semaphore, st string) {
        for {
                s.Wait()
                *shared = ((*shared) + 1) % 100000
                s.Signal()
                fmt.Print(st + " ")
                fmt.Println(*shared)
        }
}
func main() {
        var blq chan int = make(chan int)
        s := NewSemaphore()
        sh := 0
        go useSC(&sh, s, "a")
        go useSC(&sh, s, "b")
        go useSC(&sh, s, "c")
        go useSC(&sh, s, "d")
        s.Signal() // incrementa semaforo para 1
        <-blq
}
```

Usando Semáforos em Go

- Nossa definiçao: Package MCCSemaforo

```
package MCCSemaforo

type Semaphore struct {
        wai, sig chan struct{} // canais para wait e signal
        val     int         // valor do semaforo
}

func NewSemaphore(v int) *Semaphore {
        s := &Semaphore{
                wai: make(chan struct{}),
                sig: make(chan struct{}),
                val: v} // inicia semaforo com um valor, deve ser >= 0
        go func() {
                for {
                        if s.val == 0 { // se val == 0
                                <-s.sig // pode permitir apenas signal, processos fazendo wait bloqueiam
                                s.val++ // se acontecer signal, entao incrementa val
                        }
                        if s.val > 0 { // senao pode permitir tanto wait como signal, alterando val
                                select {
                                case <-s.sig:
                                        s.val++
                                case <-s.wai:
                                        s.val--
                                }
                        }
                }
        }()
        return s
}
```

```
func (s *Semaphore) Wait() { // fazer wait ee ter sucesso na sincronizacao em s.wai
        s.wai <- struct{}{} // se nao sincroniza, fica em espera, implementando o Wait()
}

func (s *Semaphore) Signal() { // fazer signal ee ter sicesso na sincronizacao em s.sig
        s.sig <- struct{}{} // esta sincronizacao sempre ee possivel como visto acima
}
```

Usando Semáforos em Go

- Nossa definiçao: Package MCCSemaforo
  - Encontre na página moodle da disciplina

```
// ATENCAO: codigo parcialmente encontrado na internet livremente
// usado aqui com objetivo de exemplificacao de sincronizacao.
// Note que a linguagem Go conta com sua propria biblioteca de
// sincronizacao.   Aqui estamos exemplificando como construir a
// semantica de semaforos a partir do uso de canais como forma de
// prover atomicidade.    Funcoes s.wait() e s.signal() tem o
// mesmo significado da literatura.
// este pacote oferece a abstracao de semaforo da literatura
// atraves da estrutura MCCSemaforo.Semaphore
// semaphore.Wait() e .Signal() sao as operacoes de semaforos
// tipicas
// Instrucoes rapidas:
// coloque o pacote MCCSemaforo (este arquivo) dentro de um diretorio
// chamado MCCSemaforo, no diretorio corrente (onde esta seu codigo).
//
// No seu codigo que usa semaforo, faca:
// import (
//          "./MCCSemaforo"
// )
// exemplo de declaracao e uso de um semaforo:
//      MCCSemaforo.Semaphore s = MCCSemaforo.NewSemaphore(1)
//      s.Wait()
//      s.Signal()
```

```
package main
import (
        "./MCCSemaforo"
        "fmt"
)
func useSC(shared *int, s *MCCSemaforo.Semaphore, fim chan int) {
        for i := 0; i < 100; i++ {
                s.Wait()                // entrada na SC
                *shared = ((*shared) + 1) % 100000 // SC
                s.Signal()              // saida da SC
        }
        fim <- 1
}
func main() {
        var ch_fim chan int = make(chan int)
        s := MCCSemaforo.NewSemaphore(1) // semaforo iniciado em 1, para SC
        sh := 0                 // A VARIAVEL COMPARTILHADA!!
        // -----------------------
        for i := 0; i < 100; i++ { // inicia 100 processos que usam SC
                go useSC(&sh, s, ch_fim)
                    // todos usam sh, o mesmo semaroro e sinalizam fim em ch_fim
        }
        for i := 0; i < 100; i++ { // espera os 100 processos acabarem
                <-ch_fim
        }
        fmt.Println("Resultado COM semaforos para SC  ", sh)
}
```

Resolvendo nível de atomicidade para SC

```go
package main

import (
    "fmt"
    "sync"
    "./MCCSemaforo"
)

func noSC() {
    var sharedTest int = 0
    var ch_fim chan struct{}=
                    make(chan struct{})

    // sem definicao

    for i := 0; i < 100; i++ {
        go func() {
            for k := 0; k < 100; k++ {
                // ENTRA NA SC
                sharedTest=sharedTest+ 1
                // SAI DA SC
            }
            ch_fim <- struct{}{}
        }()
    }
    for i := 0; i < 100; i++ {
        <-ch_fim
    }
    fmt.Println("Resultado ", sharedTest)
}
```

```go
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() // ENTRA NA SC
                sharedTest = sharedTest + 1
                sem.Signal() // SAI DA SC
            }
            ch_fim <- struct{}{}
        }()
    }
    for i := 0; i < 100; i++ {
        <-ch_fim
    }
    fmt.Println("Resultado ", sharedTest)
}
```

```go
func mxSC() {
    var sharedTest int = 0
    var ch_fim chan struct{} =
                    make(chan struct{})

    m := new(sync.Mutex)

    for i := 0; i < 100; i++ {
        go func() {
            for k := 0; k < 100; k++ {
                m.Lock() // ENTRA NA SC
                sharedTest = sharedTest + 1
                m.Unlock() // SAI DA SC
            }
            ch_fim <- struct{}{}
        }()
    }
    for i := 0; i < 100; i++ {
        <-ch_fim
    }
    fmt.Println("Resultado ", sharedTest)
}
```

```go
func chSC() {
    var sharedTest int = 0
    var ch_fim chan struct{} = make(chan struct{})
    var sc chan struct{} = make(chan struct{}, 1)

    for i := 0; i < 100; i++ {
        go func() {
            for k := 0; k < 100; k++ {
                <-sc // ENTRA NA SC - le item do canal
                sharedTest = sharedTest + 1
                sc <- struct{}{} // SAI DA SC - devolve item
            }
            ch_fim <- struct{}{}
        }()
    }
    sc <- struct{}{}
    for i := 0; i < 100; i++ {
        <-ch_fim
    }
    fmt.Println("Resultado ", sharedTest)
}
```

```go
func main() {
    noSC()
    chSC()
    mxSC()
    semaSC()
}
```