Modelos para Computação Concorrente ou Sistemas Operacionais

Memória Compartilhada e O Problema da Seção Crítica

(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

Síntese

- Compunicação entre processos
  - Canais
  - Memória Compartilhada

- Memória Compartilhada ←Foco desta apresentação
  - onde ocorre
  - necessidade de coordenação

Memória Compartilhada

- Comunicação entre processos concorrentes ocorre através de estruturas de memória compartilhadas
- Exemplos:
  - todos os ambientes que oferecem threads, go rotinas, ou análogos
  - sistemas que permitem processos do sistema operacional compartilharrem memória

Memória Compartilhada

- Exemplos:
  - threads concorrentes inserem e retiram de: filas, arvores (balanceadas, binarias, etc.), pilhas, tabelas, etc.
  - processamento de frames de imagens com sobreposições
  - cálculos matriciais onde as matrizes são compartilhadas e os processos operam diferentes linhas/colunas
  - vários usuários editando um mesmo texto
  - etc.

Memória Compartilhada

- Exemplo

```
package main

import "fmt"

var sharedTest int = 0 // variavel compartilhada
var ch_fim chan struct{} = make(chan struct{})

func MyFunc() {
    for k := 0; k < 100; k++ {
        sharedTest = sharedTest + 1
    }
    ch_fim <- struct{}{}
}

func main() {

    for i := 0; i < 100; i++ {
        go MyFunc()
    }
    fmt.Println("Criei 100 processos")

    for i := 0; i < 100; i++ {
        <-ch_fim
    }
    fmt.Println("Processos acabaram. Resultado  ", sharedTest)
}
```

Memória Compartilhada

- Exemplo

```
/* Copyright (C) 2006 M. Ben-Ari. See copyright.txt */
class Count extends Thread {
    static volatile int n = 0;

    public void run() {
      for (int i = 0; i < 10000; i++) {
        n++;
      }
    }
    public static void main(String[] args) {
      Count p = new Count();
      Count q = new Count();
      p.start();
      q.start();
      try { p.join(); q.join(); }
      catch (InterruptedException e) { }
      System.out.println("The value of n is " + n);
    }
}
```

Memória Compartilhada

- Exemplo

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int global = 0 ;          // shared

void *accessToShared()
 {        for(int i=1;i<10001;i++)   global++;    }

main()
{
     pthread_t thread1, thread2;

     /* cria duas threads independentes.
        ambas executam o mesmo codigo de funcao */
     pthread_create( &thread1, NULL, accessToShared, NULL);
     pthread_create( &thread2, NULL, accessToShared, NULL);

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);
     printf("%i global\n", global);
     exit(EXIT_SUCCESS);
}
```

Memória Compartilhada

- Exemplo

execute este programa várias vezes

observe os resultados

```
package main

import "fmt"

var sharedTest int = 0 // variavel compartilhada
var ch_fim chan struct{} = make(chan struct{})

func MyFunc() {
    for k := 0; k < 100; k++ {
        sharedTest = sharedTest + 1
    }
    ch_fim <- struct{}{}
}

func main() {

    for i := 0; i < 100; i++ {
        go MyFunc()
    }
    fmt.Println("Criei 100 processos")

    for i := 0; i < 100; i++ {
        <-ch_fim
    }
    fmt.Println("Processos acabaram. Resultado  ", sharedTest)
}
```


como o entrelaçamento das operações concorrentes pode gerar este resultado

?

Algorithm 2.3: Atomic assignment statements	
integer n ← 0	
p	**q**
p1: n ← n + 1	q1: n ← n + 1

Scenario for Atomic Assignment Statements

Process p	Process q	n
p1: n←n+1	q1: n←n+1	0
(end)	**q1: n←n+1**	1
(end)	(end)	2

Process p	Process q	n
p1: n←n+1	**q1: n←n+1**	0
p1: n←n+1	(end)	1
(end)	(end)	2

Algorithm 2.4: Assignment statements with one global reference

integer n ← 0

p	**q**
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

Correct Scenario for Assignment Statements

Process p	Process q	n	p.temp	q.temp
p1: temp←n	q1: temp←n	0	?	?
p2: n←temp+1	q1: temp←n	0	0	?
(end)	**q1: temp←n**	1	0	?
(end)	**q2: n←temp+1**	1	0	1
(end)	(end)	2	0	1

Incorrect Scenario for Assignment Statements

Process p	Process q	n	p.temp	q.temp
p1: temp←n	q1: temp←n	0	?	?
p2: n←temp+1	**q1: temp←n**	0	0	?
p2: n←temp+1	q2: n←temp+1	0	0	0
(end)	**q2: n←temp+1**	1	0	0
(end)	(end)	1	0	0

o nível de atomicidade das operações sobre os dados compartilhados deve ser claro/conhecido

como podemos fazer com que
os programas concorrentes anteriores
implementem o nível desejado de
atomicidade no acesso aos dados
compartilhados
?

Memória Compartilhada

- Exemplo

Como criar este nível de atomicidade?

```go
package main

import "fmt"

var sharedTest int = 0 // variavel compartilhada
var ch_fim chan struct{} = make(chan struct{})

func MyFunc() {
    for k := 0; k < 100; k++ {
        sharedTest = sharedTest + 1
    }
    ch_fim <- struct{}{}
}

func main() {

    for i := 0; i < 100; i++ {
        go MyFunc()
    }
    fmt.Println("Criei 100 processos")

    for i := 0; i < 100; i++ {
        <-ch_fim
    }
    fmt.Println("Processos acabaram. Resultado  ", sharedTest)
}
```

Memória Compartilhada

```go
package main

import "fmt"

var sharedTest int = 0 // variavel compartilhada
var ch_fim chan struct{} = make(chan struct{})

func MyFunc() {
    for k := 0; k < 100; k++ {


        sharedTest = sharedTest + 1  // deve ser atômico


    }
    ch_fim <- struct{}{}
}

func main() {

    for i := 0; i < 100; i++ {
        go MyFunc()
    }
    fmt.Println("Criei 100 processos")

    for i := 0; i < 100; i++ {
        <-ch_fim
    }
    fmt.Println("Processos acabaram. Resultado  ", sharedTest)
}
```

O problema da Seção Crítica

- sistema com N processos, N > 1
- cada processo pode ter um código próprio
- os processos compartilham variáveis, de qualquer tipo
- cada processo possui SC's de código, onde atualizam os dados compartilhados
- a execução de 1 SC deve:
  - ser de forma mutuamente exclusiva
  - ter espera limitada para poder executar
  - não bloquear

O problema da Seção Crítica

- Seção Crítica deve prover
  - exclusão mútua
  - espera limitada (não postergação)
    - um processo espera um tempo limitado na *entry-section*
  - não bloqueio
    - processos fora da SC não devem bloquear outros processos
    - somente os processos querendo entrar na SC devem participar da seleção do próximo a entrar
- velocidades indeterminadas
  - não se faz suposições sobre a velocidade relativa dos processos

Seção Crítica

```go
package main

import "fmt"

var sharedTest int = 0 // variavel compartilhada
var ch_fim chan struct{} = make(chan struct{})

func MyFunc() {
    for k := 0; k < 100; k++ {

        // PROTOCOLO DE ENTRADA NA SEÇÃO CRÍTICA

        sharedTest = sharedTest + 1     // SEÇÃO CRÍTICA

        // PROTOCOLO DE SAÍDA DA SEÇÃO CRÍTICA

    }
    ch_fim <- struct{}{}
}

func main() {

    for i := 0; i < 100; i++ {
        go MyFunc()
    }
    fmt.Println("Criei 100 processos")

    for i := 0; i < 100; i++ {
        <-ch_fim
    }
    fmt.Println("Processos acabaram. Resultado  ", sharedTest)
}
```

Seção Crítica

```go
package main

import "fmt"

var sharedTest int = 0 // variavel compartilhada
var ch_fim chan struct{} = make(chan struct{})


func MyFunc() {
    for k := 0; k < 100; k++ {

        // PROTOCOLO DE ENTRADA NA SEÇÃO CRÍTICA

        sharedTest = sharedTest + 1     // SEÇÃO CRÍTICA

        // PROTOCOLO DE SAÍDA DA SEÇÃO CRÍTICA
    }
    ch_fim <- struct{}{}
}

func main() {

    for i := 0; i < 100; i++ {
        go MyFunc()
    }
    fmt.Println("Criei 100 processos")

    for i := 0; i < 100; i++ {
        <-ch_fim
    }
    fmt.Println("Processos acabaram. Resultado  ", sharedTest)
}
```

COMO VOCÊ IMPLEMENTARIA ESTES PROTOCOLOS COM CANAIS ? (DEIXANDO O RESTO INTACTO)

Seção Crítica

```go
package main

import "fmt"

var sharedTest int = 0 // variavel compartilhada
var ch_fim chan struct{} = make(chan struct{})
var mx chan struct{} = make(chan struct{}, 1) // CANAL PARA CONTROLE DA SEÇÃO CRÍTICA

func MyFunc() {
    for k := 0; k < 100; k++ {

        <-mx                              // PROTOCOLO DE ENTRADA NA SEÇÃO CRÍTICA

        sharedTest = sharedTest + 1       // SEÇÃO CRÍTICA

        mx <- struct{}{}                  // PROTOCOLO DE SAÍDA DA SEÇÃO CRÍTICA
    }
    ch_fim <- struct{}{}
}

func main() {
    mx <- struct{}{}                      // UM PROCESSO PODE ACESSAR S.C.
    for i := 0; i < 100; i++ {
        go MyFunc()
    }
    fmt.Println("Criei 100 processos")

    for i := 0; i < 100; i++ {
        <-ch_fim
    }
    fmt.Println("Processos acabaram. Resultado  ", sharedTest)
}
```

Abordagens para solucionar o problema da SC

- Soluções de SW para seção crítica
  - 2 processos
  - N processos
- Suporte de HW
- Semáforos
- Monitores