Roteiro - ExclusãoMútua

- Introdução
- Algoritmo de Peterson
- Algoritmo de Lamport
- Solução de Hardware
- Exercícios

Introdução

- Quando processos compartilham dados, é importante **sincronizar** o acesso a eles
- Dados podem ser perdidos como resultado da **concorrência** entre processos ou *threads*
- Por exemplo, em *x=x+1*, e assumindo que o valor inicial da variável compartilha *x* seja *0*
- Tendo dois processos **P0** e **P1** que incrementem concorrentemente essa variável
- O valor final poderá ser comprometido

Introdução

- É natural que o programador assuma que o valor de *x* será *1* depois de executado ambos os processos
- Porém, não se pode prever que a instrução *x=x+1* não ocorra **atomicamente**
- A instrução *x=x+1* é decomposta em várias instruções para ser executada pelo processador, como:

```
LD R,x; lê o valor de x para o registrador R
INC R ; incrementa o registrador R
ST R, x; armazena o valor do registrador R em x
```

Introdução

- Agora a execução dos processos **P0** e **P1** pode ser intercaladas:

```
P0: LD R,x ; lê o valor de x para R
P0: INC R ; incrementa o registrador R
     P1: LD R,x ; lê o valor de x para R
     P1: INC R ; incrementa o registrador R
P0: ST R,x ; armazena o valor de R em x
     P1: ST R,x ; armazena o valor de R em x
```

- Ambos processos leram o valor *0* e armazeram *1* em *x*, resultando em perda de atualização ou "*lost update*"
- O problema é assegurar que *x=x+1* seja executado **atomicamente**
- A seção que deve ser executada atomicamente é chamada de **região crítica** ou **seção crítica**

Introdução

- O problema de assegurar que uma região crítica seja executada atomicamente é chamada de **exclusão mútua**
- Este é um problemas fundamentais na computação concorrente
- Pode-se abstrair o problema da exclusão mútua implementando a interface mostrada abaixo:

```
public interface Lock{
public void requestCS(int pid); //deve bloquear
public void releaseCS(int pid);
}
```

Introdução

- É dados dois protocolos, um de entrada (*requestCS*) e outro de saída (*releaseCS*)
- O processo que quer **entrar** na região crítica deve chamar *requestCS* com seu identificar como argumento
- Quando o processo ou *thread* **terminar** o acesso a região crítica deve chamar o método *releaseCS*
- **A chamada dos métodos fica a cargo do programador**

Introdução – Exemplo Lock

```
class myThread extends Thread{
    int myPID;     Lock lock;
    void processa(int tempo){try{Thread.sleep(1000);}catch(Exception e){}}
    public myThread(int pid, Lock lock){
        myPID=pid;   this.lock=lock;}
    void secaoNaoCritica(){
        System.out.println(myPID+" não está na região crítica");
        processa(r.nextInt(1000));}
    void secaoCritica(){
        System.out.println(myPID+" está na região crítica");
        processa(r.nextInt(250));  }
    public void run(){
        while (true){
            lock.requestCS(myPID);
            secaoCritica();
            lock.releaseCS(myPID);
            secaoNaoCritica();}}}
```

Introdução – Exemplo Lock

```
public class testaMyThread{
    public static void main(String args[]){
        myThread t[];
        int N=Integer.parseInt(args[0]);
        t=new myThread[N];
        Lock lock=new Attempt();//algoritmo de exclusão mútua
        for (int i=0; i<N;i++){
            t[i]=new myThread(i,lock);
            t[i].start();
        }
    }
}
```

Algoritmo de Peterson

- Esse algoritmo tem como preocupação a **exclusão mútua** entre processos ou *threads*
- Utiliza uma variável *boolean abre* inicializada como verdadeira
- O protocolo de entrada deve esperar que seu valor seja *true*
- Se for *true*, o processo entra na região crítica e seta o valor de *abre* para *false*
- Quando sair, o processo atribui o valor de *true* para a variável *abre*

Algoritmo de Peterson

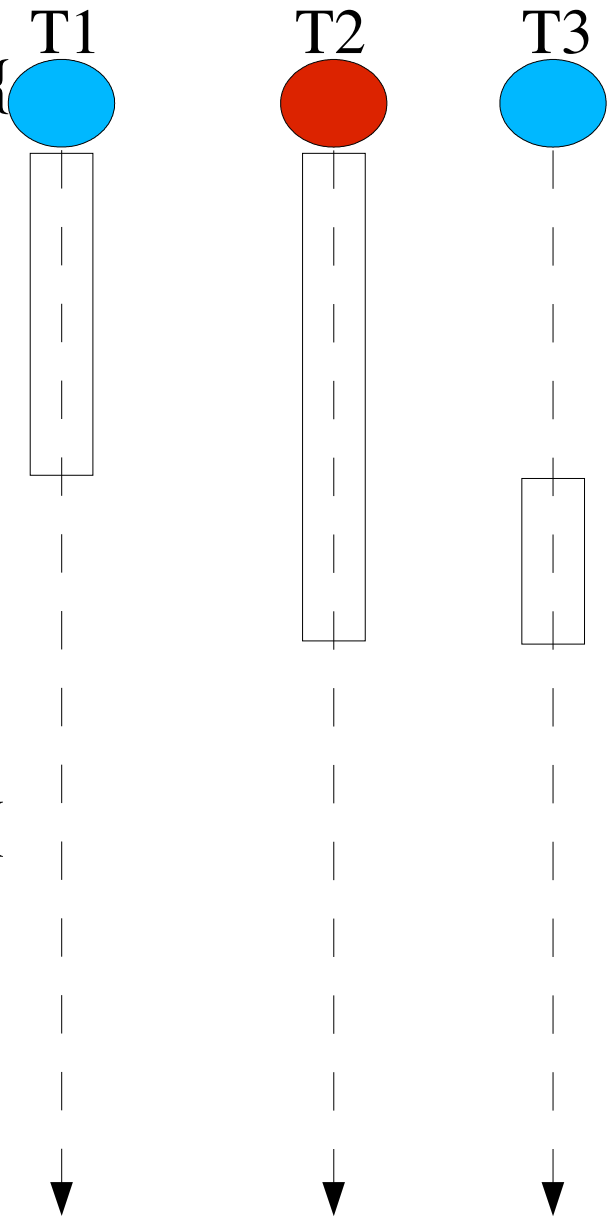
```
class Attempt implements Lock{
    boolean abre=true;
    public void requestCS(int i){//verifica o valor de abre
        while(!abre); // espera ocupada
        abre=false; //atribui falso para entrar na região crítica
    }
    public void releaseCS(int i){
        abre=true; //atribui verdadeiro ao sair
    }
}
```

Algoritmo de Peterson

- O algoritmo mostrado não funciona corretamente porque o teste de *abre* e sua atribuição não é feita **atomicamente**
- Conceitualmente, um processo pode checar o valor de *abre* e passar pelo condicional do *while*
- Entretanto, antes do processo setar *abre* como *false*, um outro processo pode testar o *while* e setar *abre* como *false*
- Assim, o primeiro processo seta também *abre* como *false* e, consequentemente, ambos estão na região crítica
- Isso ocorre quando processos têm tempo de execução diferentes

Algoritmo de Peterson

```
class Attempt implements Lock{
    boolean abre=true;
    public void requestCS(int i)
        while(!abre);
        abre=false;
    }
    public void releaseCS(int i){
        abre=true;
    }
}
```



T4

Ambos na região crítica

Exclusão mútua violada

Algoritmo de Peterson

- No próximo exemplo é proposto duas variáveis compartilhadas: *querCS[0]* e *querCS[1]*
- Cada processo primeiro seta seu próprio *querCS* como *true* e espera que o valor do *querCS* de outro processo seja *false*
- É utilizado *1-i* para identificar o processo quando são dois processos (**P0** e **P1**)
- Para deixar a região crítica, simplesmente seta como *false querCS*

Algoritmo de Peterson

```
class Attempt1 implements Lock{
    boolean querCS[]={false,false};
    public void requestCS(int i){ // protocolo de entrada
        querCS[i]=true;  // intenção de entrar
        while(querCS[1-i]); //Espera ocupada
    }
    public void releaseCS(int i){
        querCS[i]=false;  // protocolo de saída
    }
}
```

Algoritmo de Peterson

- Infelizmente, esse algoritmo também não funciona corretamente porque ambos processos podem setar *querCS* como *true* e executar **indefinidamente**
- Outra maneira de fixar o problema é baseado no valor de uma variável *vez*
- Um processo espera por sua vez para entrar na região crítica
- Ao sair da região crítica é setado o valor para *1-i*

Algoritmo de Peterson

```
class Attempt2 implements Lock{
    int vez=0;
    public void requestCS(int i){
        while(vez==1-i);
    }
    public void releaseCS(int i){
        vez=1-i;
    }
}
```

- Esse protocolo garante a exclusão mútua
- Entretanto, se um dos processos for mais rápido que o outro, acarretará em problemas
- Assim, se for garantido que os processos serão executados de modo alternado, não ocorrerá problema

Algoritmo de Peterson

- Combinando o que foi visto dos algoritmos tem-se um outra proposta
- Neste caso mantem-se *querCS[0]* e *querCS[1]* e uma variável *vez*
- A chamada para os processos funciona do mesmo jeito que os anteriores

```
class PetersonAlgorithm implements Lock{
    boolean querCS[]={false, false};
    int turn=1;
    public void requestCS(int i){
        int j=1-i;
        querCS[i]=true;
        turn=j;
        while(querCS[j] && (turn==j));}
    public void releaseCS(int i){
        querCS[i]=false;
}}
```

Algoritmo de Peterson

- Mostrou-se que o Algoritmo de Peterson tenta resolver as seguintes propriedades:
  - **Exclusão Mútua** – dois processos não podem executar a mesma seção ao mesmo tempo
  - **Progresso** – se um ou mais processos tentar entrar na região crítica e não há processo algum dentro dela, pelo menos um processo deve ser capaz de entrar
  - **Liberdade de *Starvation*** – se um processo está tentando entrar na região crítica, eventualmente ele terá sucesso

Algoritmo de Peterson - Exercício

- O algoritmo de Peterson realmente satisfaz todas as propriedades citas? Prove.

Algoritmo de Lamport's Bakery

- O algoritmo de Peterson trabalha bem com dois processos, porém com *N* o resultado é mais complexo
- O algoritmo de Lamport propõe uma solução para *N* processos ou *threads*, levando em consideração que todos possuem números de identificação distintos
- Porém, em sistemas concorrentes, é difícil assegurar que todo processo tenha um número único de identificação
- No caso de empate, usa-se a menor identificação do processo

Algoritmo de Lamport's Bakery

```
class Bakery implements Lock {
    int N;
    boolean[] choosing;
    int[] number;
    public Bakery(int numProc) {
        N = numProc;
        choosing = new boolean[N];
        number = new int[N];
        for (int j = 0; j < N; j++) {
            choosing[j] = false;
            number[j] = 0;
        }
    }
```

Algoritmo de Lamport's Bakery

```
 public void requestCS(int i) {
      choosing[i] = true;   //escolhendo um número
      for (int j = 0; j < N; j++)
         if (number[j] > number[i])
            number[i] = number[j];
      number[i]++;
      choosing[i] = false;  // verificando se o número é o menor
      for (int j = 0; j < N; j++) {
         while (choosing[j]) ; // processo j está na entrada
         while ((number[j] != 0) && ((number[j] < number[i]) ||
((number[j] == number[i]) && j < i)))
      }
   }
   public void releaseCS(int i) { // protocolo de saída
      number[i] = 0;
   }
}
```

Solução de Hardware

- Soluções via software são complexas e exigem muito processamento
- Exclusão mútua pode ser conseguida com uma pequena ajuda do hardware
  - Em um sistema monoprocessado, um processo pode **desabilitar** todas as interrupções antes de entrar na região crítica. Isso garante que o processador não o interrompa enquanto o processo não terminar o processamento
  - Muitas máquinas têm instruções com alto nível de **atomicidade** para leitura ou escrita

Solução de Hardware

- Instrução *TestAndSet* fornecida por alguma máquina, faz ambas leituras e escritas de maneira **atômica**
- Por exemplo, uma classe para ler o valor antigo da memória e colocar um valor novo

```
public class TestAndSet {
    int myValue = -1;
    public synchronized int testAndSet(int newValue) {
        int oldValue = myValue;
        myValue = newValue;
        return oldValue;
    }
}
```

Solução de Hardware

- Se a instrução *testAndSet* está disponível, pode-se desenvolver um protocolo muito simples de exclusão mútua
- Porém, apesar de realizar exclusão mútua, ainda existe espera ocupada

```
class HWMutex implements Lock {
    TestAndSet lockFlag=new TestAndSet();
    public void requestCS(int i) { // protocolo de entrada
        while (lockFlag.testAndSet(1) == 1) ;
    }
    public void releaseCS(int i) { // protocolo de saída
        lockFlag.testAndSet(0);
    }
}
```

Exercícios

- Mostre que qualquer das modificações abaixo no algoritmo de Peterson acarretará problema:
  - Um processo no algoritmo de Peterson seta a sua variável *vez* ao invés de setar a do outro processo
  - Um processo seta a variável *vez* antes de setar *querCS*
- Mostre que o algoritmo de Peterson garante liberdade de *startvation*

Exercícios

- Considere o protocolo de Dekker para exclusão mútua entre dois processos. Este protocolo satisfaz (a) **exclusão mútua** e (b) **liberdade de *livelock*** (ambos processos tentarem entrar na região crítica e nenhum deles conseguirem)? Ele satisfaz a liberdade de *starvation*?
- Modifique o algoritmo de Bakery para resolver *K* exclusões mútuas, em que *K* processos podem estar em regiões críticas concorrentemente.
- Faça um algoritmo de exclusão mútua que use instrução atômica de *swap*.