Unidade 3

Controle de Concorrência

Primitivas de Programação Concorrente Clássica

Programação Concorrente

- A abstração de programação concorrente é o estudo de **sequências de execução intercaladas**, de instruções atômicas de processos sequenciais.

- A abstração de programação concorrente trata com **sequências intercaladas de instruções atômicas**.

Abstração

- Assim, em nossa abstração, cada processo é considerado como operando sobre seu próprio processador.

Possíveis interações

- Somente temos que considerar possíveis interações em dois casos:
  - **Contenção:** Dois processos competem pelo mesmo recurso: acessando uma célula de memória ou canal em particular ou computando recursos em geral.
  - **Comunicação:** Dois processos podem necessitar se comunicar causando informação ser passada de um ao outro. Precisam se sincronizar: concordar que um certo evento tem tomado lugar entre eles.

Problema

N := 0

Processo P1:

```
   begin  N := N + 1  end
```

Processo P2:

```
   begin  N := N + 1  end
```

Variável incrementada por dois processos P1 e P2

Processo	Instrução	Valor de N
Inicialmente		0
P1	INC N	1
P2	INC N	2

Variável incrementada por dois processos P1 e P2

Processo	Instrução	Valor de N
Inicialmente		0
P2	INC N	1
P1	INC N	2

- Se o compilador traduz a instrução de mais alto nível em uma única instrução INC, qualquer intercalação das sequências de instruções dos dois processos, dará o memso valor.

- Por outro lado se toda a computação é feita em registradores, o código compilado parecerá como o código:

Processo	Instrução	N	R1	R2
Inicial		0	-	-
P1	Load R1	0	0	-
P2	Load R2	0	0	0
P1	Add R1, 1	0	1	0
P2	Add R2, 1	0	1	1
P1	Store R1,N	1	1	1
P2	Store R2,N	1	1	1

- Assim, a correção de um programa concorrente depende das instruções atômicas usadas pelo processador.

- Instruções atômicas assumem a existência de memória comum acessível a todos os processos.

- Memória comum pode ser usada em dois modos, os quais diferem somente em **o que** é acessado pelo processos:
  - Dados globais que podem ser lidos e escritos por mais do que um processo.
  - Código de rotinas de SO, que podem ser chamadas por mais do que um processo.

- Instruções em memória comum são eficientes para implementar sobre um único processador /computador sendo compartilhado por vários processos.

Problema da Exclusão Mútua para N Processos

- N processos estão executando em um loop infinito, uma sequência de instruções que podem ser divididas dentro de subsequências: uma **seção crítica** e uma **seção não crítica**.

  P programa deve satisfazer a **propriedade de exclusão mútua**: **instruções em seções críticas de dois ou mais processos não devem ser intercaladas**.

```
Loop
    Seção Não-Crítica;
    Pre-Protocolo;
    Seção Crítica;
    Pos-Protocolo;
End Loop;
```

- A solução será descrita por **inserir dentro do *Loop* instruções adicionais** que serão executadas por um processo desejando **entrar** (Pre-Protocolo) e **deixar** (Pos-Protocolo) a sua seção crítica.

- **Um processo pode parar em sua seção Não Crítica.**

- **Não pode parar a execução em seus protocolos e Seção Crítica.**

- **Se um processo pára em sua Seção Não-Crítica, ele não deve interferir sobre outros processos.**

Deadlock

- O programa não pode entrar em ***deadlock***.

  Se alguns processos estão tentando entrar em sua Seção-Crítica então um deles deve ser, eventualmente, bem sucedido.

Starvation

- Nenhum ***starvation*** deve existir de um dos processos .

  Se um processo indica sua intenção para entrar em sua Seção-Crítica, por começar a execução do seu Pre-Protocol, eventualmente, ele será bem sucedido.

Contenção

- Na ausência de contenção (processos **não** competem por um mesmo recurso) para a Seção-Crítica, um único processo desejando entrar em sua Seção-Crítica será bem sucedido.

Semáforos

- Um semáforo S é uma variável inteira que pode tomar somente valores não negativos.

- Duas operações são definidas sobre um semáforo S.

Semáforos

- Wait(S) - Se S>0 então S:=S-1, senão suspende a execução do processo. O processo é dito estar suspenso sobre o Semáforo S.

- Signal (S) – Se existem processos que tem sido suspensos sobre o semáforo S, acorde um deles. Senão S:=S+1.

Propriedades dos Semáforos

- **Wait(S)** e **Signal(S)** são instruções atômicas. Nenhuma instrução pode ser intercalada entre o teste S>0 e o decremento de S, ou nenhuma instrução pode ser intercalada entre a verificação da suspensão de processos e o incremento de S.

Propriedades dos Semáforos

- A um semáforo S deve ser dado um valor inicial não negativo qualquer.

- A operação Signal (S) deve acordar um dos processos suspensos. A definição não especifica qual processo deverá ser acordado.

Exclusão Mútua com Semáforos

```
S: Semaphore :=1
P1:    loop
           Seção-Não-Crítica-1;
           wait(s);
           Seção-Crítica-1;
           Signal(S);
       end loop;
```

Exclusão Mútua com Semáforos

```
P2:    loop
          Seção-Não-Crítica-2;
          wait(s);
          Seção-Crítica-2;
          Signal(S);
       end loop;
```

Propriedades de Semáforos

- Teorema 1: **A propriedade de exclusão mútua é satisfeita.**
- Teorema 2: **O programa não tem *deadlock*.**
- Teorema 3: **Não existe *starvation*.**

Monitores

- Semáforos : Solução para problemas comuns.
- Semáforos : Primitiva de baixo nível.
- Semáforos : Em grandes sistemas, usar semáforo somente, a responsabilidade para o correto uso de semáforos é difundida entre todos os implementadores do sistema.

Deadlock com Semáforos

- Se um implementador esquece de chamar um **Signal(S) após uma seção crítica**, o programa pode entrar em deadlock e a causa da falha será difícil de isolar.

Monitores

- Proporcionam uma primitiva de programação concorrente estruturada, que concentra a responsabilidade de correção em poucos módulos.

- Tipo de Dados baseado em estado.

Monitores

- **Seções críticas** para alocação de dispositivos de I/O ou alocação de memória são centralizados em **um programa privilegiado**.

Monitor

- Programas ordinários requerem serviços que são realizados pelo monitor central.

- Temos um programa que manipula todos as requisições de serviços envolvendo dispositivos compartilhados ou estruturas de dados.

Monitor

- Podemos definir um monitor separado para cada objeto ou grupo de objetos relacionados.

- Processos podem requisitar serviços de vários monitores.

Monitor

- Se um mesmo monitor é chamado por dois processos, a implementação do monitor garante que esses processos são executados serialmente para preservar exclusão mutua.

- Se monitores diferentes são chamados, os processos podem ser intercalados.

Monitor

- A sintaxe de monitores é baseada no **encapsulamento** de itens de dados e os procedimentos que operam sobre esses itens, colocados dentro de **um único módulo**.

- A **interface** para um **monitor** consistirá de um **conjunto de procedimentos**.

Monitor

- Esses procedimentos operam sobre dados que são ocultos dentro do módulo.

- Um monitor não somente protege os dados internos de acessos inrestristos, mas também sincroniza as chamadas aos procedimentos da interface.

Monitor

- A implementação garante que os procedimentos são executados sob exclusão mútua sob variáveis globais.

- Na **semântica de um monitor**, **somente um processo é permitido executar uma operação no monitor em qualquer tempo**.

Monitor

- Monitor define uma primitiva de sincronização que permitirá um processo suspender ele próprio.

- O monitor não é um processo, mas um módulo estático de dados e declarações de procedimentos (procedures).

Monitor Produtor-Consumidor

```
Monitor  Produtor_Consumidor

    B: array(0..N-1) of integer;
    In_Ptr,  Out_Ptr:  Integer  := 0;
    Count:Integer  := 0;
    Not_Full, Not_Empty:  Condition;
```

Produtor coloca ítem no Buffer B

```
Procedure  Append (I:  in Integer)
Begin
    If  Count = N then  WAIT (Not_Full) ;
    B(In-Ptr) := I;
    In_Ptr := (In_Ptr) mod N;
    SIGNAL (Not_Empty) ;
End  Append;
```

Consumidor retira ítem do Buffer B

```
Procedure Take (I: out  Integer)
   Begin
       If  Count = 0  then  WAIT (Not-Empty) ;
       I := B(Out_Ptr);
       Out_Ptr  := (Out_Ptr  + 1) mod N;
       SIGNAL (Not_Full) ;
   End  Take;
```

Processo Producer

```
Process   Producer
    I: Integer;
   begin
     loop
        Produce (I);
        Append (I);
     end loop;
   end Producer
```

Processo Consumidor

```
Process  Consumer
    I:  Integer;
begin
    loop
        Take (I) ;
        Consumer (I);
    end loop;
end Consumer;
```

Operações do Monitor

- **WAIT** e **SIGNAL** aqui, **não tem nenhuma relação** com as duas primitivas usadas em operações de semáforo.

- Para **sincronização** são definidas **Variáveis de Condição**: Not_Empty e Not_Full.

Variáveis de Condição

- **Not_Empty** : Usada pelo consumidor para suspender ele próprio, até que o buffer seja não vazio.

- **Not_Full** : Usada pelo produtor para suspender ele próprio, quando o buffer estiver cheio.

Três Operações sobre Variáveis de Condição

- **WAIT(C)** pode ser lido: **"Estou esperando para C ocorrer"**.
  O processo que chamou a procedure do monitor contendo esta declaração, é suspenso sob uma fila FIFO associada com C.

- **SIGNAL(C)** pode ser lido: **"Estou sinalizando que C ocorreu"**.
  Se a fila para C é não vazia, então acorde o processo na cabeça da fila.

- **NON_EMPTY(C)** : Uma função booleana que retorna true se a fila para C é não vazia.

Tarefa Avulsa 2 – Entregar na aula do dia 23/03

- Implementar em Java, a solução Monitor apresentada para o Problema do **Produtor x Consumidor com buffer limitado**.

- Consultar Deitel, “Java : Como Programar”,
  Páginas 783-787.

- Ver o **modificador** de métodos: **synchronized** para implementar o monitor.

- Os processos Produtor e o Consumidor deverão ser implementados como Threads.