**INE5645 – PROGRAMAÇÃO PARALELA E DISTRIBUIDA - LISTA DE EXERCÍCIOS 1**1. Na abstração em programação concorrente, cada processo ou thread é considerada estar operando sobre seu próprio processador, executando seu próprio programa. Somente temos que considerar a possível interação em dois casos:  
  
Contenção: (explicar o que é) Dois processos/threads competem para o mesmo recurso: recursos de computação em geral, ou acesso a uma particular célula de memória ou canal em particular.  
  
Comunicação: (explique o que é) Dois processos/threads podem necessitar se comunicar causando a informação ser passada de um para o outro. Processo/threads podem se sincronizar (concordam que certo evento ocorreu) para se comunicarem.

2. Descreva o que é o problema da exclusão mútua ?

N threads estão executando sobre um loop infinito uma sequência de instruções que podem ser divididas em subseqüências: a seção crítica e a seção não-crítica. O programa deve satisfazer a propriedade de exclusão mútua: instruções de uma seção critica não devem ser intercaladas.

A forma de solução para exclusão mútua:

Loop  
 Seção Não-Crítica;  
 Pre-Protocolo; instruções adicionais que são executadas por threads desejando entrar sua   
 seção crítica.  
 Seção Crítica;  
 Pos-Protocolo; instruções adicionais que são executadas por threads desejando deixar sua  
 seção crítica.  
 Seção Não-Crítica;  
End loop;

Uma thread pode parar em sua seção não-crítica, mas não pode parar durante a execuação de seus protocolos e seção crítica. Se uma thread parar em sua seção não-crítica, ela não deve interferir com outras threads.

O programa concorrente não deve ter *deadlock*. Se alguma thread está tentando entrar em sua seção crítica, então uma delas deve eventualmente (num tempo finito, mas não especificado) ser bem sucedida.

Não deve haver nenhum *starvation* de uma das threads. Se uma thread indica sua intenção de entrar em sua seção crítica por começar a execução do pré-protocolo, eventualmente, ela se sucederá.

Na ausência de contenção para a seção crítica, uma única thread desejando entrar em sua seção crítica, será bem sucedida.

3. Caracterize o que é um ***monitor***; o que são ***locks***; e o que é um ***semáforo****.*

Monitor - É um mecanismo de sincronização de processos/threads que provê programação concorrente estruturada que concentra a responsabilidade para correção dentro de poucos módulos. Seções críticas tais como alocação de dispositivos de I/O e memória, requisições de enfileiramento para I/O, e assim por diante, são centralizados em um programa privilegiado. Porgramas ordinários requisitam serviços que são realizados pelo monitor central. A sintaxe de monitores é baseada no encapsulamento de itens de dados e as procedures que operam sobre eles em um único módulo. A interface de um monitor consiste num conjunto de procedures. Essas procedures operam sobre dados que são ocultos dentro do módulo monitor.

Locks – É um mecanismo de sincronização de processos/threads (transações podem implementar processos/threads), em que processos/threads (transações, por exemplo) devem ser programados de modo que seus efeitos sobre os dados compartilhados sejam **equivalentes serialmente**. Se for sabido que cada uma de várias threads (transações) tem o mesmo efeito correto quando executada sozinha, então podemos inferir que, se essas threads (transações) forem executadas uma por vez, em alguma ordem, o efeito combinado também será correto. Uma intercalação das operações das threads (transações) em que o efeito combinado é igual ao que seria se as threads (transações) tivessem sido executadas uma por vez, em alguma ordem, é uma intercalação equivalente serialmente. Quando dizemos que duas threads (transações) distintas tem o mesmo efeito, queremos dizer que as operações de leitura sobre variáveis de instâncias de objetos (por exemplo, contas bancárias) retornam os mesmos valores e que essas variáveis de instância de objetos tem os mesmos valores finais. O uso de equivalência serial como critério para execução concorrente correta evita a ocorrência de atualizações perdidas ou recuperações inconsistentes. Um exemplo simples de mecanismos para disposição em série é o caso de locks (travas) exclusivos. Nesse esquema, um servidor tenta impedir o acesso (travar) a qualquer objeto que esteja para ser usado por qualquer operação de uma thread (ou transação) de um cliente do servidor. Se um cliente solicitar acesso a um objeto que já está bloqueado (travado) devido a uma thread de outro cliente, o pedido será suspenso e o cliente deverá esperar até que o objeto seja destravado. A implementação de locks pode ser feita para que o funcionamento de locks (travas) e de bloqueio sejam implementadas por um objeto em separado num servidor, que chamamos de gerenciador de locks (travas). **O gerenciador de (locks) travas mantém um conjunto de travas** (por exemplo numa tabela hashing). **Cada trava é uma instância da classe Lock e é associado a um objeto (uma conta bancária) em particular.** A classe Lock aparece nos slides. Os métodos de Lock são sincronizados para que as threads que estão tentando adquirir (método aquire, que usa wait) ou liberar (método release, que usa notify) uma trava não interfiram umas com as outras. Tentativas de adquirir a trava usam o método wait quando precisam esperar que outra thread a libere. Todos os pedidos para obter travas (locks) e liberá-las em nomes de threads (transações) são enviadas para uma instância do gerenciador de travas (*class LockManager*).

Semáforos – Algoritmos concorrentes podem ser executados somente usando-se instruções de linguagem de máquina que computadores provêem. Essas instruções podem ser usadas para implementar soluções corretas para exclusão mútua e outros problemas de programação concorrente, mas são de baixo nível em demasia para ser eficiente e confiável. Semáforos provêem primitivas de programação concorrente de mais alto nível do que instruções de linguagem de máquina. Um semáforo é uma variável S, inteira, que pode tomar valores não-negativos e exatamente duas operações (primitivas do semáforo) são definidas sobre o semáforo S. Wait(S) – se S>0 então S := S-1, senão suspende a execução da thread e esta é dita estar suspensa sobre S. Signal(S) – se existem threads suspensas sobre S, então acorde uma delas, senão S := S+1.

4. Uso de mecanismos de sincronização:Programação Concorrente serve para resolver o problema da exclusão mútua entre processos/threads, para o qual você estudou três mecanismos importantes: **semáforos**, **monitores** e **locks**. Qual o seu sentimento com relação ao uso destes mecanismos, ao realizar as três tarefas práticas, no que tange a vantagem/desvantagem de um mecanismo sobre o outro, ou características de cada mecanismo que mostre as vantagens do uso de um em relação ao outro.

Mecanismos de sincronização são ferramentas importantes para o alto

desempenho, quando se possui mais de um processador, ou mesmo com um

processador, pois pode-se ocupar de forma mais eficiente os recursos

disponíveis (principalmente o processador). Assim, uma comparação é importante entre os mecanismos de sincronização já estudados.

O tamanho da região crítica (muito pequena ou grande), é um fator que irá influenciar

no mecanismo de sincronização a ser utilizado. Num cenário onde temos uma região crítica muito grande, com dezenas ou centenas de instruções, é mais aconselhado usar os monitores ou locks. O ganho de desempenho dos monitores com relação aos locks nestes casos é muito pequeno, mas devido sua facilidade de uso, é mais aconselhado.

Seja com baixa ou alta contenção (números pequeno ou grande de processos ou threads que concorrem pelo mesmo recurso: recursos de computação em geral, ou acesso a um recurso como memória ou um canal de comunicação), quando se possui região crítica muito pequena alta contenção (500 threads ou mais), aconselha-se o uso de locks ao invés de monitores, devido o ganho no desempenho.

Semáforos – Provêem um mecanismo de sincronização, mas que é não estruturado. O uso de semáforos na construção de grandes sistemas é difícil de garantir o uso correto, principalmente sendo utilizado por diversos implementadores do sistema. Se um implementador esquecer uma chamada *Signal(S)* após um seção crítica, o programa pode entrar em *deadlock* e a causa será difícil de isolar.

Monitor – A sintaxe de monitores é baseada no encapsulamneto de items de dados e as operações que manipulam esses dados, dentro de um módulo. É um mecanismo de sincronização que concentra a responsabilidade de correção dentro de um módulo (ou de poucos módulos, se mais monitores forem usados). Sua principal vantagem em relação aos outros mecanismos de sincronização é a simplicidade de programação, pois não é necessário bloquear/desbloquear locks ou controlar um grande número de semáforos, num sistema complexo. Desta forma, torna-se mais difícil gerar deadlocks. Sua principal desvantagem é a falta de flexibilidade.

Locks -

Suas principais vantagens quanto aos monitores, além da flexibilidade:

– vários locks podem ser adquiridos e liberados em ordem diferente, em

diferentes escopos;

– é permitido mudar a estratégia da fila de espera (FIFO, aleatória),

com os locks “justos”.

Como era de se esperar, também possuem desvantagens:

– seu uso é mais complicado que o dos monitores;

– deadlocks podem facilmente ser gerados.

5. Esquematize usando pseudo-código a forma de solução para exclusão mútua.

Loop  
 Seção Não-Crítica;  
 Pre-Protocolo; instruções adicionais que são executadas por threads desejando entrar sua   
 seção crítica.  
 Seção Crítica;  
 Pos-Protocolo; instruções adicionais que são executadas por threads desejando deixar sua  
 seção crítica.  
 Seção Não-Crítica;  
End loop;

6. O que é *starvation* ?

Uma thread indica sua intenção de entrar em sua seção crítica por começar a execução do seu pré-protocolo, e **num tempo finito**, ela não se sucederá.

7. Exclusão mútua com semáforos:

S: Semaphore := 1;

**Task P1 is**

**Begin**

**Loop**

**Non\_Critical\_Section\_1;   
 wait(S);  
 Critical\_Section\_1;  
 signal(S);  
   
 end loop;**

**End P1;**

**Task P2 is**

**Begin**

**Loop**

**Non\_Critical\_Section\_2;   
 wait(S);  
 Critical\_Section\_2;  
 signal(S);  
   
 end loop;**

**End P2;**

Para o pseudo-código ilustrando o problema da exclusão mútua com semáforos. Estude a seguindo as páginas 48 e 49 do capítulo de semáforos postado na página (Cap. 4 Semaphoree, M. Bem-Ari, Principles of Concurrent and Distributed Programming) as provas que:

1. A propriedade da **exclusão mútua é satisfeita**.
2. O programa não tem ***deadlock***.
3. O programa não apresenta nenhum ***starvation***.

Sugestão: inicie pela página 48, seção 4.2 sobre propriedades invariantes de semáforos. Examine o código. Estude os três teoremas da página 49.

8. Emule um semáforo S usando a construção de um monitor, considerando as operações Semaforo-Wait e Semaforo\_Signal. Use pseudo-linguagem, definindo uma variável C, tipo *condition.*

**Monitor** Emulação\_Semaforo

S: integer := S0;

Not\_Zero: Condition;

**Operation** Semaforo\_Wait

**If** S = 0 **then** Wait (Not\_Zero); **end if**;

S := S - 1;

**end** Semaforo\_Wait;

**Operation** Semaforo\_Signal

S := S + 1;

Signal (Not\_Zero);

**end** Semaforo\_Signal;

**end** Monitor.

Este monitor emula um semáforo. A variável S retém o valor do semáforo e é inicializado a algum valor não negativo S0. A variável de condição (Condition) Not\_Zero mantém a fila de processo ou threads esperando para o semáforo ser não-zero.

Wait(Not\_Zero) O processo/thread que chamou a operação do monitor contendo esta instrução é suspenso sob uma fila associada com a condição Not\_Zero.

Signal(Not\_Zero) Se a fila para a condição Not\_Zero é não vazia, então acorde o processo/thread na cabeça da fila.