INE 5645 – PROGRAMAÇÃO PARALELA E DISTRIBUÍDA – PROVA 1 – TURMA B - 2013.1
Prof. Bosco - 13/05/2013

Aluno: _____**Gabarito Prova 1**_____

1. Na abstração em programação concorrente, cada processo ou thread é considerada como se estivesse operando sobre seu próprio processador. Somente temos que considerar a possível interação entre processos/threads em dois casos:

(a) **(Verdade/ Falso)** - Contenção é quando dois processos/threads competem para o mesmo recurso: recursos de computação em geral, ou acesso a uma particular célula de memória ou canal de comunicação em particular. (Vale 1,0)

(b) **(Verdade/Falso)** - Comunicação: Dois processos/threads podem necessitar se comunicar causando informação ser passada de um para o outro. Processos/threads podem se sincronizar (concordam que certo evento ocorreu) para se comunicarem. E a comunicação pode ser síncrona ou assíncrona. (Vale 1,0)

2. N threads estão executando sobre um *loop* infinito, uma sequência de instruções em cada thread que pode ser dividida em subseqüências: a seção crítica e a seção não-crítica. Um programa concorrente constituído de N threads deve satisfazer a propriedade de exclusão mútua, ou seja, instruções de uma seção critica não devem ser intercaladas. A forma de solução para exclusão mútua segue o modelo abaixo:

Thread_N:

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

(Vale 0,25 cada)

(a) Cite algumas instruções Java, que são executadas por threads desejando entrar sua seção crítica.

wait(), acquire(), lock() ... estas correspondem ao pré-protocolo

(b) Cite algumas instruções Java que são executadas por threads desejando deixar sua seção crítica.

notify(), release(), unlock() ... estas correspondem ao pós-protocolo

(c) **(Verdade/Falso)** Uma thread pode parar em sua seção não-crítica, mas não pode parar durante a execução de seus protocolos e sua seção crítica. Se uma thread parar em sua seção não-crítica, ela não deve interferir com outras threads.

(d) **(Verdade/Falso)** Um programa concorrente não deve ter ***deadlock***. Se alguma thread está tentando sair (entrar) de sua seção crítica, então uma delas deve eventualmente ser bem sucedida.

(e) **(Verdade/Falso)** 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.

(f) **(Verdade/Falso)** O conceito de ***livelock***: Uma thread, frequentemente, age em resposta a uma outra thread. Se a ação da outra thread também é uma resposta à ação de uma outra thread, então esta situação pode resultar em *livelock*. Tal como acontece com *deadlock*, threads "livelocked" são incapazes de ter progressos em suas execuções. No entanto, em **livelock**, as threads não são bloqueadas – ficam simplesmente demasiadamente ocupadas respondendo uma a outra, podendo retomar ao trabalho para que foram programadas.

(g) **(Verdade/Falso)** A palavra "eventualmente" em programação concorrente, tem um significado importante. "Eventualmente" significa que: num tempo finito, mas não especificado, algum evento ocorrerá.

(h) **(Verdade/Falso)** Na ausência de contenção para a seção crítica, uma única thread desejando entrar em sua seção crítica, será bem sucedida.

3. (Verdade/Falso) O esquema de escalonamento (*scheduling*) fundamental para threads é preemptivo (o ato de forçar uma thread parar sua execução), baseado em prioridade, quando um algoritmo não baseado em fatias de tempo é executado. Ou ocorre através de *time-slicing*, quando threads são paradas de executar após transcorrido um intervalo de tempo, *default* ou fixado executando no processador. *Time-slicing* é preemptivo, mas preempção não implica em *time-slicing*, pois permitem threads de mais alta prioridade executarem tanto tempo elas necessitem. Descreva brevemente, o que acontece no escalonamento em Java. (Vale 2,0)

Em Java existem os dois esquemas de escalonamento. Se existem threads com mesma prioridade, essas são escalonadas em time-slicing. Quando threads tem prioridades diferentes, ou seja o processador tem uma de menor prioridade, mas existe uma de maior prioridade para

executar, a de menor é forçada a parar (preempção) e a maior prioridade passa a ser executada. Lembrem da figura do Deitel, no capítulo 23 da sexta edição.

4. Use pseudo-linguagem para emular um semáforo binário S (assume 0 ou 1), usando a construção de um monitor, considerando as operações **Semaforo-Wait** (um acquire) e **Semaforo_Signal** (um release). Você pode definir uma variável "Não-Zero", tipo *condition,* para indicar o status do semáforo S, que é testado "Não Zero" nas operações que sincronizam duas threads T1 e T2. (Vale 2,0). Use o verso da folha.

Emulacao-semaforo

not_zero : Condition

S: integer :=0

procedure semaforo-wait

if S=0 then

wait(not_zero)

end if

S= S-1

end semaforo-wait

procedure semaforo-signal

S = S-1

Signal(not_Zero)

end semaforo-signal

End emulacao-semaforo

5. Dado a figura seguinte, verifique a correção do funcionamento das transações V e W, explicando o porquê a questão da concorrência não está resolvida.

(a) Dizer qual o problema existente entre V e W. (Vale 1,0)

The Inconsistent Retrievals Problem

Transaction : *V*		**Transaction : *W***	
a.withdraw(100) *b.deposit(100)*		*aAgency.agencyTotal()*	
OpenTransaction			
a.withdraw(100);	$100		
		OpenTransaction	
		total = a.getBalance()	$100
		total =total+b.getBalance()	$300
		total = total+c.getBalance	
		CloseTransaction	
b.deposit(100)	$300		
CloseTransaction			

Supondo que os saldos de ambas as contas A e B são, inicialmente, iguais a $200.

Sejam as transações bancárias *V* e *W*. Pelas instruções resumidas, em destaque, em cada transação, a transação *V* deve transferir uma quantia ($100) da conta *A* para a conta *B*. e a transação *W* deve executar o método *agencyTotal* para obter a soma dos saldos de todas as contas da agência *Agency*.

Na execução concorrente em questão, a transação V é aberta, executa o saque em A e o resultado deste saque é $100. A transação B é aberta. O saldo de A é obtido e armazenado em *total* ($100). Em seguida, o saldo de B é obtido. Como B tinha $200, total acaba ficando com ($200 + $100) = $300. A conta C não é compartilhada entre V e W. A transação W é fechada. Em seguida, T faz o depósito sacado de A em B. A conta B fica com $300. A transação T é fechada. Mas, o depósito em B foi feito depois da soma dos saldos das contas ter sido obtido. Isto dá um resultado errado.

É sabido que cada uma das transações V e W tem o efeito correto quando executadas sozinhas. Então, podemos inferir que, se essas transações forem executadas uma por vez, em alguma ordem, o efeito combinado também será correto. Uma intercalação das instruções das transações em que o efeito combinado é igual ao que seria se as transações tivessem sido executadas uma por vez, em alguma ordem, é uma intercalação equivalente serialmente.

O problema das recuperações inconsistentes pode ocorrer quando uma transação de recuperação (por exemplo, W) é executada concorrentemente com uma transação de atualização (por exemplo, T). Isso não pode ocorrer se a transação de recuperação for executada antes ou depois da transação de atualização.

Note que o acesso de cada transação à B, não se dá em série, com relação uma a outra, pois T não faz todos os seus acessos à B, antes de W.

Uma intercalação equivalente serialmente, de uma transação de recuperação e de uma transação de atualização como na figura seguinte, seria o resultado correto.

Transação : T	Transação : W
a.withdraw(100);	
b.deposit(100);	*aAgency.agencyTotal()*
OpenTransation	
a.withdraw(100); $100	
b.deposit(100); $300	
	OpenTransation
	total = a.getBalance() $100
	total = total + b.getBalance() $400
	total = total + c.getBalance()
	CloseTransation
CloseTransation	

(b) Mostre, usando a figura acima ou refazendo a mesma, como pode a intercalação das transações V e W ser resolvida usando-se ***locks***. Use o verso da folha, se precisar.

(Vale 1,0)

Transação : T		Transação : W
a.withdraw(100);		
b.deposit(100);		*aAgency.agencyTotal()*
OpenTransation		
a.withdraw(100);	lock A	
b.deposit(100);	lock B	
		OpenTransation
		total = a.getBalance() Espera pelo lock de V em A.
		total = total + b.getBalance() Espera pelo lock de V em B.
CloseTransation	unlock A, B	
		total = a.getBalance() lock A

| *total = total + b.getBalance()* lock B
| *total = total + c.getBalance()* lock C
| *CloseTransation* unlock A, B, C
