INE5645 – PROGRAMAÇÃO PARALELA E DISTRIBUÍDA - Prova 1 – 12/05/2014

Aluno: _____Bosco_____

1. Um servidor acessa as variáveis *i* e *j*. O servidor oferece duas operações para seus clientes:

   read(i) retorna o valor de *i*

   write(i, v) atribui o *valor* v à *i*

   As transações T e U são definidas como segue:

   T: x=read(i); write(j,44);

   U: write(i,55); write(j,66);

   a) Considerando o uso de l*ocks* e *unlocks* no quadro abaixo:

T	*Locks de T*	*U*	*Locks de U*
	lock i		
x := read(i);			
	unlock(i)		
			lock i
		write(i,55);	
			lock j
		write(j,66);	
			unlock(i,j)
	lock j		
write(j,44);			
	unlock j		

(a) Explique o que está ocorrendo entre as transações *T* e *U*, quanto aos acessos a *i*. Diga em que ordem de acesso existe entre *T* e *U*. (0,5)

T conflita com *U* no acesso a *i*. Ordem de acesso é *T*, e depois, *U*.

(b) Explique o que está ocorrendo entre as transações *T* e *U*, quanto aos acessos a *j*. Diga em que ordem de acesso existe entre *U* e *T*. (0,5)

T conflita com U no acesso a *j* . Ordem de acesso é *U,* e depois, *T*.

(c) O que se pode afirmar quanto a intercalação entre as transações T e U sobre o efeito dos locks ? (0,5)

Existe uma intercalação das transações T e U, na qual o efeito dos *locks* (travas), sendo liberados mais cedo, gera o efeito de que as transações não sejam serialmente equivalentes.

2. Considerando o quadro montado na questão 1, monte um outro quadro, criando uma intercalação das transações T e U,na qual o efeito dos *locks* (travas) torne as transações serialmente equivalentes. Sugestão: Lembre que a ordem de diferentes pares de operações conflitantes de duas transações deve ser a mesma. (1,5)

T	*Locks de T*	*U*	*Locks de U*
	lock i		
x := read(i);			
	lock j		
write(j,44);			
	unlock (i,j)		
			lock i
		write(i,55);	
			lock j
		write(j,66);	
			unlock(i,j)

3. Considere o uso de *locks* usados na descrição abaixo em que duas transações T e U executam as operações atômicas de *deposit* e *withdraw*.

(a) O que se pode afirmar sobre as execuções das transações T e U ? (1,0)

T		*U*	
Operações	*Locks*	*Operações*	*Locks*
	lock a		
a.deposit(100);			
			lock b
		b.deposit(200);	
b.withdraw(100);			
…	*espera por unlock de U sobre b*	*a.withdraw(200);*	
		...	*espera por unlock de T sobre a*
…			
…			

Resposta: *Impasse com locks de escrita(operação deposit sobre a e b), provocando deadlock entre as transações T e U.*

(b) Usando somente *locks* e *unlocks* remonte o quadro acima para que as transações T e U sejam equivalentes serialmente. (1,0)

T	*U*

Operações	*Locks*	*Operações*	*Locks*
	lock a		
a.deposit(100);			
	lock b		
b.withdraw(100);			
	unlock (a,b)		
			lock b
		b.deposit(200);	
			lock a
		a.withdraw(200);	
			unlock(a,b)

4. Emule um semáforo S usando a construção de um monitor, considerando neste monitor, as operações de sincronização *Semaforo-Wait* e *Semaforo_Signal*. Para na ter que escrever uma linguiagem de programção, use uma pseudo-linguagem, definindo uma variável Not_Zero do tipo *Condition.* (2,0)

```
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 inicial 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.

5 . Indique Verdade ou Falso: (2,0)

a) (Verdade / Falso) Threads distintas em um processo não são tão independentes. (0,5)

Verdade. Todas as threads tem exatamente o mesmo espaço de endereçamento, o que significa que elas compartilham as mesmas variáveis globais do processo. Além de compartilharem o mesmo espaço de endereçamento, todas as threads compartilham o mesmo conjunto de recursos do processo (arquivos abertos, processos filhos, alarmes pendentes, tratadores de eventos, informação sobre contabilidade de execução, entre outros).

b) (Verdade / Falso) Não há proteção entre threads que executam num mesmo processo, porque é impossível e desnecessário. (0,5)

Falso. Num processo criado por um usuário, presume-se que este tenha criado múltiplas threads no processo para que essas possam cooperar e não competir. Do ponto de vista de threads em processos diferentes, e ainda mais de esses processos forem de usuários diferentes, a proteção entre threads (visando segurança) é importante.

c) (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. (0,5)

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

d) (Verdade / Falso) No problema dos Leitores x Escritores, existem três tipos de soluções de sincronização: (0,5).

- Na priorização dos leitores: sempre que um leitor quiser ler e não houver escritor escrevendo (pode haver escritor esperando), ele tem acesso à variável compartilhada.

Nesta solução, um escritor pode ter de esperar indefinidamente (havendo *starvation*), pois novos leitores sempre chegam.

- Na priorização dos escritores: quando um escritor desejar escrever, mais nenhum leitor pode fazer leituras enquanto o escritor não for atendido. Nesta solução, um leitor pode ter de esperar indefinidamente (havendo *starvation*), pois novos escritores sempre chegam.
- Havendo prioridades iguais: não há risco de *starvation*, pois leitores e escritores têm as mesmas chances de acesso à variável compartilhada; pode haver uma queda de desempenho em relação às soluções anteriores.

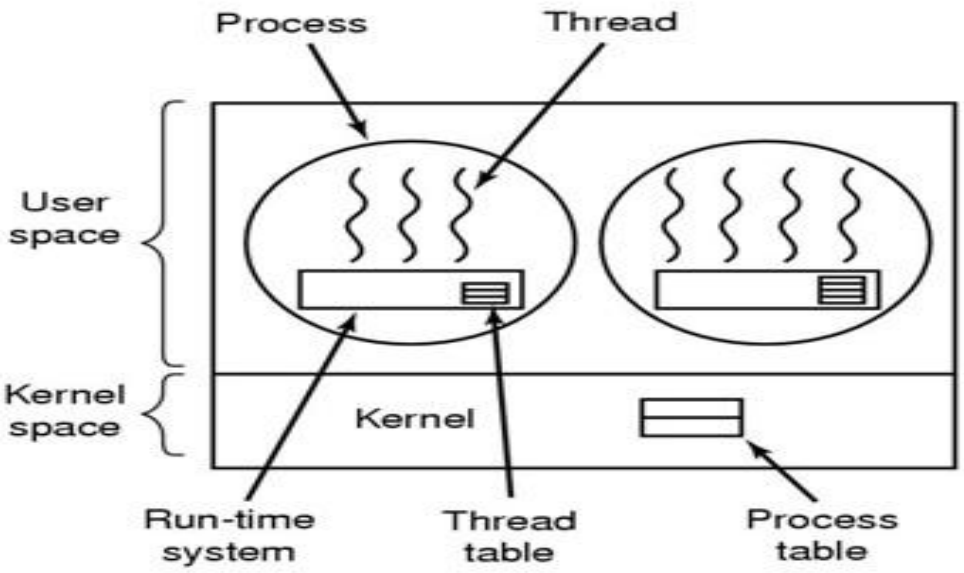
6. Indique Verdade ou Falso: (1,0)

a) (**Verdade** ou **Falso**) Itens de propriedade de processos são: Espaço de endereçamento, Variáveis globais, Contador de programa lógico, Registradores, Pilha, Estado e Recursos. (0,50)

Verdade. Os itens refletem o que é um processo.

b) (**Verdade** ou **Falso**) Espaço de endereçamento e recursos pertencem aos processos. Threads pertencentes a um processo compartilham o mesmo espaço de endereçamento do processo, e usam os mesmos recursos do processo hospedeiro. (0,50)

7. Considere a figura abaixo. Procure lembrar do ambiente de programação que você usou para programar seus programas das tarefas práticas executadas durante seu aprendizado. Explique, brevemente, as afirmações seguintes: (1,0)



a) Usualmente, o conjunto de threads em um SO, é dividido em duas categorias: **thread ao nível do usuário e thread ao nível de Kernel**. (0,5) Correto, como mostrado, claramente, na figura.

b) **As threads ao nível do usuário, em um pool de threads, são escalonadas pelo programador**, tendo a grande vantagem de cada processo usar um algoritmo de escalonamento que melhor se adapte a situação: escalonamento *time-sliced* ou escalonamento preemptivo. O sistema operacional neste tipo de thread não faz o escalonamento, em geral ele não sabe que elas existem. Neste modo, o programador é responsável por criar, executar, escalonar e destruir a thread. (0,5)

Correto, quando você programa, você pode se utilizar das formas de escalonamento, por exemplo, como usadas na linguagem Java.