**INE5645 - Programação Paralela e Distribuída - Prova 1 – 23/05/2016**

**Nome :** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**1 – Escalonamento em Java – (2,5)**

Duas transações T e U permitem o problema da **atualização perdida** conforme o quadro:

Suponha que você implemente seu programa em Java, considerando um gerenciamento de pool de threads, para as duas transações T e U, como:

**ExecuteService ES = Executors.newFixedThreadPool(PoolSize);**

1. Para a intercalação acima, o que acontece, sobre o resultado da variável compartilhada b, se você definir o tamanho do pool PoolSize = 2 ?
2. Se o tamanho do pool for iguala 2, as duas transações irão competir concorrentemente, mas diante da intercalação acima, não haverá equivalência serial entre as duas transações e o problema da atualização perdida ocorrerá sobre a variável compartilhada b.
3. Para esta intercalação acima, o que acontece, sobre o resultado da variável compartilhada b, se você definir o tamanho do pool PoolSize = 1 ?

Se o tamanho do pool for igual a 1, a execução será uma transação de cada vez, e neste caso não haverá intercalação entre as transações, pois as execuções das duas será serial, proporcionando o resultado correto sobre a variável compartilhada b.

**2 –Sincronização de Threads – Semáforo (2,5)**

Considere a seguinte definição de semáforo. Um semáforo **S** é uma variável de valor inteiro, a qual pode tomar somente valores não negativos (**S>=0**), e duas operações são definidas sobre **S**: ***wait(S)***, se **S>0** então ***S = S-1*** e executa o processo/thread senão suspende a execução do processo/thread sobre **S**. O processo/thread é dito estar suspenso sobre **S,** aguardando numa fila em **S**; ***signal(S)***, se existem processos/threads que tenham sido suspensas sobre ***S***, então acorde um deles, senão ***S = S+1***. Da definição de semáforo acima, seja o pseudo-código seguinte:

**S: Semaphore := 1;**

**Thread T1 is**

**Begin**

**loop**

**Non\_Critical\_Section;**

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

**end loop;**

**End T1;**

**Non\_Critical\_Section\_2;*wait(S);***

**Thread T2 is**

**Begin**

**loop**

**Non\_Critical\_Section;**

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

**end loop;**

**End T2;**

**Non\_Critical\_Section\_2;*wait(S);***

Considerando níveis de prioridade de execução de 1 a 10 (1 é de menor prioridade e 10 a maior prioridade) para as threads T1 e T2. Em que cenário haverá *starvation* na concorrência entre as threads ? Explique sua resposta. (2,5)

Não haverá starvation. No programa concorrente acima, parte-se do princípio que as duas threads tendo a mesma prioridade (5 como em Java) o esquema de escalonamento será *Time-slicing* (não-premptivo), ou seja, fatias de tempo iguais para cada thread no processador.

Mas, considerando-se, o esquema de escalonamento preemptivo, com níveis de prioridade de execução de 1 a 10 (1 é de menor prioridade e 10 a maior prioridade) para as threads T1 e T2, respectivamente, uma vez que a thread de maior prioridade ganhe o processador, após executar a sua seção crítica, ela executará tanto tempo ela necessite e, depois executará  *signal(S)* fazendo o semáforo S = 1. O escalonador passa, então, para a thread de menor prioridade. O semáforo éfundamental para a sincronização entre as threads, com ou com prioridades. Veja a figura sobre prioridades de threads em Java, no livro Deitel, Cap23.

**3 –intercalação de Transações – (2,5)**

Suponha que as transações **T** e **U** num servidor, sejam definidas como   
segue:  
 ***T:*** *read ( i );* ***U:*** *write (i, 55);   
 write ( j, 44); write ( j, 66);*

**Monte duas figuras dando dois exemplos de intercalações que são serialmente equivalentes.** Quando dizemos que um par de operações conflitam, significa dizer que seu efeito combinado depende da ordem na qual elas são executadas. O efeito de uma operação refere-se ao valor de um objeto estabelecido por um *write* e o resultado retornado por um *read.*Para qualquer par de transações, é possível determinar a ordem de pares de operações conflitantes sobre objetos acessados por ambas as transações. **Serialmente equivalente** pode ser definido em termos de operações conflitantes como segue: “**para duas transações serem serialmente equivalentes, é necessário e suficiente que todos os pares de operações conflitantes das duas transações sejam executados na mesma ordem em todos os objetos que ambas acessam**”. **Pares de operações conflitantes** das duas transações **devem ser executados na mesma ordem em todos os objetos i e j** que elas acessam. **Ordem de acesso serialmente equivalente** requer uma das duas condições: (1) **T acessa i antes de U e T acessa j antes de U.** (2) **U acessa i antes de T e U acessa j antes de T.** Mostre uma intercalação das transações T e U, com o efeito que tal intercalação seja **serialmente equivalente**, escolhendo uma das condições anteriores.

Qualquer intercalação entre T e U que satisfaça uma das condições mencionadas.

**4.** **Indique Verdade/Falso (2,5)**

(a) (**Verdade**/**Falso**) Itens propriedade de processos são: Espaço de endereçamento, Variáveis globais, Contador de programa lógico, Registradores, Pilha, Estado e Recursos.   
  
Exatamente a definição da construção do que é um processo, o qual gerencia os seus recursos para servir as suas threads.

(b) (**Verdade**/**Falso**) Itens próprios de threads são: Variáveis globais, Contador de programa lógico, Registradores, Pilha, Estado. Thread é uma unidade de gerenciamento de seus recursos.

Quem gerencia os recursos de um thread é o processo que a contém.

(c) (**Verdade**/**Falso**) Threads distintas em um processo não são tão independentes quanto processos distintos.

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).

(d) **(Verdade/Falso)** Não há proteção entre threads porque é impossível e desnecessário.

Falso. No caso de processos diversos, que podem ser de usuários diferentes e até mutuamente hostis, um processo sendo 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 se esses processos forem de usuários diferentes, proteção entre threads é importante.

(e) **(Verdade/Falso):** Um esquema de escalonamento (scheduling) é o ato de **selecionar** processos /threads prontas **para o estado executável**. Um tal esquema pode ser **preemptivo**, ou seja, o ato de forçar uma thread parar sua execução baseado em prioridades das threads, que permitem threads de mais alta prioridade executarem tanto tempo quanto elas necessitem. Ou pode ser **não-preemptivo**, o qual força thread parar sua execução num certo tempo de processamento, ou seja, é baseado em **fatias de tempo** (*time slicing*) de processamento. O escalonamento em Java pode ser feito usando estas duas formas, com prioridades definidas de 1 (menor) a 10 (maior) prioridade. Para threads com prioridades iguais, o escalomanmento é *time-slicing*.  
  
Exatamente o que define estes conceitos.