**INE5645 - Programação Paralela e Distribuída - Prova 1 – 23/09/2015**

**Prof. Bosco** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Parte 1 –Gerenciamento de threads – (2,0)**

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 ? (1,0)

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.

1. Para esta intercalação acima, o que acontece, sobre o resultado da variável compartilhada b, se você definir o tamanho do pool PoolSize = 1 ? (1,0)

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.

**Parte 2 – Controle de Concorrência – Semáforo (2,0)**

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, segue que, um semáforo ***S*** satisfaz as seguintes **invariantes de estado** do semáforo (Isto é, vale para todos os valores que o semáforo assume):   
  
***S>=0***e***S = S0 + #Signals - #Waits***, onde ***S0*** é o valor inicial do semáforo, ***#Signals*** é o número de *signals* executado sobre S, e ***#Waits*** é o número de *waits* completados executados sobre S.

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);***

1. 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. (1,0)

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

**Considerando-se, agora, 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 *signal(S)* fazendo o semáforo S = 1. Mas, como o escalonamento é preemptivo (baseado em prioridades das threads, que permitem threads de mais alta prioridade executarem **tanto tempo elas necessitem**), o escalonador passa para de menor prioridade. Veja a figura sobre prioridades de threads em Java, no livro Deitel, Cap23.

1. *Deadlock* descreve uma situação onde nenhuma thread é capaz de ganhar acesso regular a recursos compartilhados e por isso são incapazes de progredir. Supondo o quadro acima, em que situação haverá *deadlock* ? (1,0)

Não haverá deadlock. Para provar, suponha que existe. Se é assim, ambas as threads serão suspensas sob Wait(S). Então S=0, porque ambas estão suspensas. Donde, o números de threads em sua seção crítica será zero, ou seja #CS = 0, porque nenhuma está em sua seção crítica. Existe o invariante #CS + S = 1 (que pode ser visto no livro Ben-Ari, Concurrent Programming, que é sempre verdade (isto pode ser provado, pode ser visto de forma intuitiva), teremos #CS + S = 1, donde 0 + 0 = 1. O que é uma contradição. Então, a hipótese de existir deadlock é falsa. Logo, não haver deadlock será verdade.  
  
**Você pose ter respondido certo com outras palavras.**

**Parte 3 – Controle de Concorrência com Monitor (2,0)**

Usando uma pseudo-linguagem, construir um monitor chamado **Emulação-Semaforo**, que emule um semáforo representado por uma variável ***S***. O semáforo ***S*** é inicializado a um valor inteiro não-negativo ***S0***. Deve existir uma variável de condição do monitor chamada **not\_zero**, a qual mantém a fila de processos ou threads esperando para o semáforo ser não-zero. Denomine os procedimentos do monitor como **semaforo-wait** e **semaforo-signal**.

**Monitor** 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

**Parte 4 – Controle de Concorrência com Locks (2,0)**

**4.1** Um servidor gerencia ítens de dados ***a1, a2, ..., an*** e ele acessa as variáveis ***i*** e ***j***, conforme o quadro abaixo. O servidor oferece duas operações para seus clientes:   
  
 ***read(i)*** *retorna o valor de ai*; ***write(i, v)*** *atribui o valor v à ai*.

As transações ***T*** e ***U*** são definidas como segue:

-------------------------------------------------------------------------------------------------------------

***T:*** | ***U:***---------------------------------------------------- | -------------------------------------------------------  
 **|  
*ai = read ( i );*** |

|  
 | ***ai = write ( i, 55 );***

|

|   
 | ***aj = write ( j, 66 );***

|  
***aj = write ( j, 44 );*** |  
 |

-------------------------------------------------------------------------------------------------------------

1. Explique porque **equivalência serial** requer que uma vez uma transação tenha liberado um lock sobre um item de dado, não é permitido obter quaisquer mais locks.

Porque a ordem de diferentes pares de operações conflitantes de duas transações devem ser a mesma. **Para ser equivalente serialmente** é requerido que **uma** das duas seguintes condições ocorra como uma das regras seguintes na respostada questão b.

1. No quadro acima, em termos de operações conflitantes, o que está acontecendo entre as transações T e U ?  
     
   A intercalação é **não-equivalente serialmente**. Note que **o acesso de cada transação às variáveis** i e j **é serializado**, pois T faz o seu acesso a i antes de U fazer, e U faz o seu acesso a j, antes de T fazer. **Mas, a ordem dos pares das operações conflitantes não é serialmente equivalente.** **Para ser equivalente serialmente** é requerido que **uma** das duas seguintes condições ocorra.
   * + 1. **T acessa i antes de U e T acessa j antes de U.  
           OU**
       2. **U acessa i antes de T e U acessa j antes de T.**

Ou seja, uma das duas intercalações devem existir:

-------------------------------------------------------------------------------------------------------------

***T:*** | ***U:***---------------------------------------------------- | -------------------------------------------------------  
 **|  
*ai = read ( i );*** |

**| *aj = write ( j, 44 );*** | |  
 |   
 |  
 | ***ai = write ( i, 55 );***

|  
 | ***aj = write ( j, 66 );***

|

-------------------------------------------------------------------------------------------------------------

OU

-------------------------------------------------------------------------------------------------------------

***T:*** | ***U:***---------------------------------------------------- | -------------------------------------------------------  
 **|** ***ai = write ( i, 55 );***

|  
 | ***aj = write ( j, 66 );***

**|  
*ai = read ( i );*** |

**| *aj = write ( j, 44 );*** | |  
 |   
 |

-------------------------------------------------------------------------------------------------------------

(c) Descreva, **usando o quadro acima**, uma intercalação das transações   
 T e U, nas quais locks sejam liberados com o efeito que a   
 intercalação seja serialmente equivalente.

-------------------------------------------------------------------------------------------------------------

***T:*** | ***U:***---------------------------------------------------- | -------------------------------------------------------  
**openTransaction** lock i **|  
*ai = read ( i );*** |  
 lock j |

***aj = write ( j, 44 );*** | **openTransaction** |  
 |   
 | Wait para T fazer unlock i, j  
 |  
 **close Transaction** unlock i, j | …. ….   
 | lock i  
 | ***ai = write ( i, 55 );***

| lock j  
 | ***aj = write ( j, 66 );***

|

-------------------------------------------------------------------------------------------------------------

(d) Suponha que as transações T e U no servidor, sejam definidas como   
 segue:  
 *T: ai = read( i ); aj = write( j, 44);  
 U: ai = write (i, 55); aj = write( j, 66);*

Os valores iniciais de *ai* e *aj*são 10 e 20. Dê dois exemplos de   
 intercalações que são serialmente equivalentes e dois exemplos que não são.   
 Desenhar à mão, os quadros dos seus exemplos.

Obs: Estes valores *ai* e *aj*, 10 e 20, respectivamente, não tem influência influência.   
 **Serialmente equivalentes:**

1. **2)**

**Para ser equivalente serialmente** é requerido que **uma** das duas seguintes condições ocorra:

**T acessa i antes de U e T acessa j antes de U, ou U acessa i antes de T e U acessa j antes de T.**

**Serialmente não-equivalentes:  
 1) 2)**

**Para não ser equivalente serialmente** é requerido que não valha **nenhuma** das duas seguintes condições ocorra:

**T acessa i antes de U e T acessa j antes de U, ou U acessa i antes de T e U acessa j antes de T**

**Parte 5 – Conceitos Básicos (Verdade/Falso) (2,0)  
 Explique, resumidamente cada questão** (0,25)

**5.1**  (**Verdade** ou **Falso**) Threads distintas em um processo não são tão independentes quanto processos distintos. (0,25)

Verdade. Todas as threads no processo 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).

**5.2** (**Verdade** ou **Falso**) Não há proteção entre threads porque é impossível e desnecessário. (0,25)

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.

**5.3** (**Verdade** ou **Falso**) Itens propriedade de **threads** são: (a) **Espaço de endereçamento,** (b) Variáveis globais, (c) Contador de programa lógico, (d) Registradores, (e) Pilha, (f) Estado, (g) **Recursos**. (0,25)

Espaço de endereçamento e recursos são propriedades de processos.

**5.4**  (**Verdade** ou **Falso**) Itens propriedade de **processos** são: (a) Variáveis globais, (b) Contador de programa lógico, (c) Registradores, (d) Pilha, (e) Estado. (0,25)

Deve ser incluído:Espaço de endereçamento e recursos como propriedades de processos.

**5.5**  (**Verdade** ou **Falso**) Thread é uma unidade de processamento concorrente. (0,25)

Antigamente, a única unidade de processamento concorrente era o processo (antes de haver thread), mas do jeito que programamos atualmente, com threads sendo cidadãos de primeira classe, essas passam a ser, no caso de estarmos no espaço do usuário, as unidades de processamento concorrente. Um SO pode ter no nível de kernel, processos com as unidades de processamento concorrente.

**5.6**  (**Verdade** ou **Falso**) O esquema de escalonamento (scheduling) fundamental para threads é preemptivo (o ato de forçar uma thread parar sua execução), baseado em prioridade. Um algoritmo **não** baseado em fatias de tempo (time-slicing é preemptivo, mas preempção não implica em time-slicing), que permitem threads de mais alta prioridade executarem tanto tempo elas necessitem. (0,25)

**O que é verdade**: 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 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 que é verdade é: **escalonamento *time-slicing* é preemptivo, mas preempção não implica em *time-slicing*.**

**5.7** (**Verdade** ou **Falso**) Uma aplicação que é melhor construída considerando-se múltiplas threads compartilhando o mesmo espaço de endereçamento, representa a construção de um processo que implementa essa aplicação. (0,25)

Neste caso, a aplicação é construída como um processo contendo diversas threads internas a ele.

**5.8** (**Verdade** ou **Falso**) **Thread** e **processo** são conceitos diferentes. Threads acrescentam ao modelo de processo um ambiente de programação *multithreading*. (0,25)

Veja as respostas das questões 5.3 e 5.4. Também a resposta da questão 5.7.