INE 5645 Programação Paralela e Distribuída – 2012.1 – Turma 0238B – Prof. Bosco
Nome: Bosco-Soluções Matrícula: _____

Parte 1 – Processos e Threads (20%)

1.1 **(Verdade/Falso)** Diferença entre programa e processo. (2%)

Um **programa** é um algoritmo expresso por uma linguagem adequada ao computador que contém **atividades** que devem ser executadas e essas são chamadas de **processos**. Um **programa** (um software) corresponde a um conjunto de processos. A idéia principal é que um **processo** é um código seqüencial e constitui uma atividade (tarefa) dentro de um espaço de endereçamento. O modelo de processo é baseado em dois conceitos independentes: **fluxo de execução** e **agrupamento de recursos**. Por recursos entende-se arquivos abertos, processos filhos, alarmes pendentes, tratadores de eventos, sinalização no *kernel*, informação sobre contabilidade de execução. Pô-los todos juntos, no contexto de processos, facilita o gerenciamento desses recursos.

1.2 **(Verdade/Falso)** Um processo de Sistemas Operacionais. (2%)

Todos os softwares que podem executar em um computador, inclusive o SO (os mais tradicionais são assim, como o UNIX), são organizados para serem executados num processador ou numa máquina com vários processadores em paralelo , com vários **processos sequenciais** (também chamados processos). Um **processo** é uma atividade (ou tarefa) de um programa, que contém o código e dados dessa atividade. Essas são: leitura de dados, escrita de dados, cálculos no processador, comunicação com o usuário, comunicação com um BD, comunicação com a rede interna ou externa, entre outras. Um processo define **a unidade de processamento concorrente**, que é executada num dado instante num processador, utilizando um contador de programa lógico, usando o único contador de programa físico (registro no processador), valores em registradores, variáveis do programa e uma pilha de execução. Processos são escalonados para o processador, que faz uma troca a todo momento do processo sendo executado, através do mecanismo chamado **multiprogramação**.

1.3 Complete o texto: O que as threads acrescentam ao modelo de processo ? (2%)

Thread e **processo** são conceitos diferentes. Em SO tradicionais (os mais antigos), cada processo tem **um único fluxo de execução**, que define a unidade de processamento

concorrente destinada para ser executada sob as condições de desempenho de um processador da época. Com o surgimento de processadores de mais alto desempenho, e que a noção de processo era diferente dos requisitos para os sistemas distribuídos, uma nova unidade de processamento concorrente chamada **thread** pôde ser definida dentro de um próprio processo, materializando novas unidades de fluxo de execução e assim pode-se ter múltiplos fluxos de execução num mesmo processo.

O que as **threads** acrescentam ao modelo de processo é permitir que múltiplos fluxos de execução ocorram no mesmo ambiente do processo, com um certo grau de independência uma das outras. Assim, **múltiplas threads** podem executar concorrentemente em um processo, e é análogo a múltiplos processos executando concorrentemente em um único processador.

Neste caso, **threads** compartilham o mesmo espaço de endereçamento e recursos do **processo** onde são executadas e o termo **multithreading** é usado para descrever a situação em que múltiplas threads sendo executadas no mesmo processo.

1.4 Complete o texto: Quando um **processo** com múltiplas threads é executado em um SO com um único processador, as threads são escalonadas para execução, alternando rapidamente, dando a ilusão que são executadas em paralelo num processador mais lento que o processador real. Programação concorrente é uma **abstração** (dica - o que ignora detalhes) que é projetada para tornar possível se raciocinar sobre o comportamento dinâmico de processos ou threads concorrentes. (2%)

1.5 (**Verdade**/**Falso**) Itens propriedade de processos são: Espaço de endereçamento, Variáveis globais, Contador de programa lógico, Registradores, Pilha, Estado e Recursos. (2%)

1.6 (**Verdade**/**Falso**) Itens próprios de threads são: Variáveis globais, Contador de programa lógico, Registradores, Pilha, Estado. (Verdade/**Falso**) Thread é uma unidade de gerenciamento de seus recursos. (2%)

1.7 (**Verdade**/**Falso**) Threads distintas em um processo não são tão independentes quanto processos distintos. (2%)

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).

1.8 (**Verdade**/**Falso**) Não há proteção entre threads porque é impossível e desnecessário. (2%)

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.

1.9 Complete o texto: 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. Sublinhe: (**Verdade** ou **Falso**) Escalonamento *time-slicing* (fatias de tempo de processador) é preemptivo, mas preempção não implica em time-slicing. (2%)

1.10 Complete o texto: O conceito de **contenção** corresponde ao conceito de processos/threads competirem para o mesmo recurso, por exemplo, para acessar uma área de memória ou um canal de comunicação, em particular. (2%)

Parte 2 – Correção de Programas Concorrentes (15%)

2.a **(Verdade/Falso)** Intercalações em um programa concorrente são sequências de instruções atômicas (a execução no processador não pode sofrer parada) de processos/threads sequenciais. (5%)

2.b **(Verdade/Falso)** Intercalações arbitrárias são permitidas. Um processo/thread pode completar centenas de instruções antes que qualquer outro processo/thread comece a executar uma instrução. Ou pode existir uma intercalação de uma instrução em um tempo de cada processo/thread. (5%)

2.c **(Verdade/Falso)** Um programa concorrente é requerido ser correto sob todas as intercalações possíveis. (5%)

Parte 3 – Semáforos e Monitores (30%)

3.1 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;


Thread  T2  is
Begin
    loop
        Non_Critical_Section;
        wait(S);
        Critical_Section_2;
        signal(S);
        Non_Critical_Section;
    end loop;
End T2;
```

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, 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, como em Java) para as threads T1 e T2, o que acontecerá se supormos T1 com prioridade 1 e T2 com prioridade 10. (10%)

Obs: A ideia de que uma thread tenha prioridade máxima e outra prioridade mínima, subentende-se que pode existir *starvation* para a thread de menor prioridade. 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, o escalonador escolhe a de maior prioridade (neste caso, só existem duas threads) sempre e ....

3.2 Para o programa concorrente do item 2.1, complete a demonstração do teorema de que não há deadlock. Dem: Suponha que exista **deadlock** . Então, ambos as threads devem estar suspensas sobre wait(S). Logo **S = 0** para ambos as threads. E **#CS = 0**, porque nenhuma thread entrou em sua seção crítica. Mas, pelo invariante #CS + S = 1 (mostrado em aula), logo **0** + **0** = 1. Portanto, foi encontrada uma contradição na hipótese, e assim, o Teorema (Não há deadlock.) é verdadeiro. (5%)

3.3 Usando a definição de semáforo acima, e 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**. Sugestão: use a definição de semáforo dentro de um monitor. Use o verso da folha, se precisar. (15%)

```
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 (35%)

4.1 O problema de atualização perdida (lost update, constante nos slides de aula) está ilustrado abaixo com as transações T e U, sobre as contas bancárias A, B e C, cujos valores iniciais são $100, $200 e $300, respectivamente. A transação T transfere um valor da conta A para a conta B e a transação U transfere um valor da conta C para B. Considere que em ambos os casos, o valor transferido é calculado para acrescentar ao saldo de B, o valor de 10% ao seu valor inicial. Considerando os efeitos das duas transações T e U executarem concorrentemente: (15%)

(a) explicar, em poucas linhas, o problema que está ocorrendo na descrição que segue: (5%)

- O efeito sobre a conta B de executar as transações T e U, deve ser para aumentar o *balance* (saldo) de B em 10%, duas vezes. Assim, o valor final deveria ser $242.

O problema da atualização perdida (The lost update problem)

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

- Os efeitos de permitir as transações T e U executarem concorrentemente como na figura "lost update", ambas as transações obtém o *balance* de B como $200 e então *deposit* $20.
- O resultado é incorreto, aumentando o *balance* de B em $20 ao invés de $42.

(b) Em seguida, montar um quadro descrevendo a correção com equivalência serial, usando as mesmas transações T e U com *locks* exclusivos. (10%)

Transações *T* and *U* com Locks - Figura 4

Transaction : *T*		**Transaction : *U***	
balance = b.getBalance() *b.setBalance(bal*1.1)* *a.withdraw(bal/10)*		*balance = b.getBalance()* *b.setBalance(bal*1.1)* *c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T's* unlock on *B*
closeTransaction	unlock *A, B*	•••	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		closeTransaction	unlock *B C*

4.2 Em um par de operações conflitantes *read* e *write*, seu efeito combinado depende da ordem que as operações são executadas. O efeito de uma operação se refere ao valor de uma variável estabelecido por uma operação *write* e o resultado retornado por uma operação *read*. (10%)

Observe o quadro abaixo e explique:

Uma intercalação não equivalente serialmente de operações de transações *T* e *U*

Transaction *T*:	**Transaction *U*:**
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

(a) Por que as transações T e U realizando as operações conflitantes *write* e *read*, não são serialmente equivalentes ??? Antes de responder leia o ítem (b). (5%)

Porque a ordem de intercalação não é serialmente equivalente, porque os pares de operações conflitantes:

Read and *write* operation conflict rules

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* opions does not depend on the order in which they are executed.
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution.
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution.

não são feitos na mesma ordem em ambosas variáveis i e j..

(b) Escreva uma condição para que a equivalência serial das transações T e U seja alcançada, levando-se em consideração que em **todos os pares de operações conflitantes sejam executados na mesma ordem em todas as variáveis, i e j**, que ambas acessam. (5%)

Intercalações ordenadas serialmente equivalentes requerem **uma** das seguintes condições:

- T acessa *i* antes de U e T acessa *j* antes de U.
- U acessa *i* antes de T e U acessa *j* antes de T.

4.3 Considere o uso de *locks* usados na descrição abaixo em que duas transações T e U executam as operações atômicas de *deposit* e *withdraw*. Responda as seguintes questões: (10%)

Transaction *T*		Transaction *U*	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
• • •	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for *T*'s lock on *A*
• • •		• • •	
• • •		• • •	

(a)Do ponto de vista da execução concorrente das transações T e U, dentro do que você aprendeu sobre o uso de *locks*, a descrição apresentada está correta? Se sim, explique em poucas linhas, o porquê está correta. (5%)

Não está correta.

(b) Se não está, explique em poucas linhas o porquê não está correta, dizendo o nome da propriedade de incorreção que você identifica. Monte um outro quadro descritivo mostrando a correção de execução das transações. (5%)

As duas transações estão em deadlock.

Figure 16.23
Resolution of the deadlock in Figure 15.19

Transaction T		Transaction U	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
• • •	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for T's lock on *A*
	(timeout elapses)	• • •	
	T's lock on *A* becomes vulnerable, unlock *A*, abort T	• • •	
		a.withdraw(200);	write locks *A* unlock *A*, *B*