INE 5645 Programação Paralela e Distribuída – 2011.1 – Turma 0238B – Prof. Bosco
Nome: _____BOSCO_____ Matrícula: _____

Parte 1 – Processos e Threads (20%)

1.1 Explique, resumidamente, o que é um processo em Sistema Operacional. (0,20)

Todos os softwares que podem executar em um computador, inclusive o SO (os mais tradicionais são assim), são organizados para serem executados num processador, como vários processos sequenciais (também chamados processos). Um **processo** é uma atividade (ou tarefa) de um programa, que contém o código e dados de uma 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**.

1.2 Explique a diferença entre programa e processo. (0.20)

A diferença entre um processo e um programa é sutíl, mas crucial. **Um programa é um algoritmo expresso por uma linguagem adequada ao computador que contém atividades que devem ser executadas** e são chamadas de processos. Um programa (um software) corresponde a um conjunto de processos. A idéia principal é que um processo constitui uma atividade (tarefa) e que tem um espaço de endereçamento. O modelo de processo é baseado em dois conceitos independentes: agrupamento de recursos e fluxo de execução. Por recursos entende-se arquivos abertos, processos filhos, alarmes pendentes, tratadores de eventos, sinais no kernel, informação sobre contabilidade de execução. O processo deve facilitar o gerenciamento desses recursos.

1.3 Thread e **processo** são conceitos diferentes. Explique sucintamente, a diferença entre o conceito de processo e o conceito de thread. O que as threads acrescentam ao modelo de processo ? (0,20)

Em SO tradicionais, cada processo tem um único **fluxo de execução** (o que define uma **thread**), 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, uma nova unidade de processamento concorrente pôde ser definida dentro do próprio processo, materializando novas unidades de fluxo de execução e assim pode-se ter múltiplos fluxos de execução (múltiplas threads) 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 grau de independência uma das outras. Assim, **múltiplas threads executam concorrentemente em um processo**, e é análogo a **múltiplos processos executando concorrentemente em um único computador**.

No primeiro caso, threads compartilham o mesmo espaço de endereçamento e recursos do processo onde são executadas e o termo *multhreading* é usado para descrever a situação em que múltiplas threads são executadas no mesmo processo. Quando um processo com múltiplas threads é executado em um SO com um único processador, as threads são escalonadas para execução, alternando rapidamente entre as threads, dando a ilusão que são executadas em paralelo num processador mais lento que o processador real.

1.4 (**Verdade** ou **Falso**) **Threads distintas em um processo não são tão independentes quanto processos distintos.** Explique, resumidamente. (0,20)

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).

1.5 (**Verdade** ou **Falso**) **Não há proteção entre threads porque é impossível e desnecessário.** (0,20)
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 de esses processos forem de usuários diferentes, proteção entre threads é importante.

1.6 (**Verdade** ou **Falso**) Itens propriedade de processos são: Espaço de endereçamento, Variáveis globais, Contador de programa lógico, Registradores, Pilha, Estado e Recursos. (0,20)

Verdade. Os itens refletem o que é um processo.

1.7 (**Verdade** ou **Falso**) Itens propriedade de threads são: Variáveis globais, Contador de programa lógico, Registradores, Pilha, Estado. (0,20)

Verdade. Compare com 1.6 e veja que espaço de endereçamento e recursos pertencem aos processos.

1.8 (**Verdade** ou **Falso**) Thread é uma unidade de gerenciamento de seus recursos. (0,20)
Falso. Quem gerencia recursos é o processo.

1.9 (**Verdade** ou **Falso**) O esquema de escalonamento (*scheduling*) fundamental para threads é preemptivo (força uma thread parar sua execução) e baseado em prioridade. O *scheduler* (escalonador), neste caso, é um algoritmo não baseado em fracionamento de tempo (*time-slicing* é preemptivo, mas preempção não implica em *time-slicing*), que permite threads de mais alta prioridade executarem tanto tempo quanto elas necessitem. (0,20)

Verdade. Ou o escalonamento é preemptivo por fracionamento de tempo para threads de igual prioridade, ou escalonamento é preemptivo baseado em prioridade estabelecida para threads.

1.10 Dê exemplo de uma aplicação (pode ser que processem uma quantidade grande de dados) que é melhor construída considerando-se múltiplas threads compartilhando o mesmo espaço de endereçamento ou espaços distintos. Descreva a estrutura dessa aplicação e dizendo quais as finalidades de suas threads. (0,20) Todos deram um exemplo.

Parte 2 – Semáforos e Monitores (40%)

2. 1 Dos mecanismos de sincronização de threads que você exercitou (monitor, locks, semáforos), explique seu sentimento quanto as vantagens de cada mecanismo relativo ao outro. (1.0)

Vide lista 1.

2.2. 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;
```

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. (0.5)

Suponha que exista deadlock. Então, para o programa concorrente (as duas threads sendo executadas) entrar em deadlock, ambos os processo/threads devem ser suspensos em WAIT(S). Neste caso, S = 0, porque, por hipótese, estão suspensos e o #CS, número de processos/threads em sua região crítica deve ser 0. Mas, pelo invariante #CS + S = 1 (estamos considerando S binário), temos 0 + 0 = 1. O que é uma contradição. Portanto, não existindo, starvation. Se tem prioridades definidas, supõe-se que o enunciado de 1.9 se aplica.

2.3 ***Starvation*** descreve uma situação onde uma thread é incapaz de ganhar acesso regular a recursos compartilhados (pode ser o processador) e é incapaz de progredir. Isso acontece quando os recursos compartilhados são indisponíveis por longos períodos por "ganância" de threads. Por exemplo, em Java, suponha que um objeto fornece um método sincronizado e que muitas vezes leva muito tempo para retornar. Se um thread chama este método freqüentemente, outras threads que também precisam ter acesso sincronizado freqüentes para o mesmo objeto, muitas vezes, serão bloqueadas.

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 *liverlock*. Tal como acontece com *deadlock*, threads "livelocked" são incapazes de ter progressos em suas execuções. No entanto, as threads não são bloqueadas – ficam simplesmente demasiadamente ocupadas respondendo uma a outra, do que poder retomar ao trabalho para que foram programadas. Isto é comparável a duas pessoas tentando passar uma a outra numa corrida de atletismo: Afonso move-se para à esquerda para deixar Gastão passar, enquanto Gastão se move para à direita para deixar Afonso passar. Vendo que eles ainda estão se bloqueando a todos os outros movimentos, Afonso move-se para a sua direita, enquanto, Gastão move-se para sua esquerda. Eles ainda estão bloqueando uns aos outros, então ... Tente dar um exemplo computacional, nos moldes do exemplo de *starvation* acima em que *livelock* pode ocorrer. (1.5)

Imagine um exemplo em que duas ou mais threads precisam adquirir todas as Locks de um objeto, se a thread não conseguir obter todas as locks então ela tenta de novo, isso cria um

livelock com todas as threads tentando obter todos os locks mas nenhuma consegue porque uma atrapalha a outra.

LiveLock pode ocorrer numa situação onde há um mecanismo de travas em um dado programa que tenham dos recursos compartilhados, A e B. Se uma thread T1 requer lock sobre A e um T2 requer lock sobre B simultaneamente. Após isso T1 tenta acessar B e T2 tenta acessar A concomitantemente, T1 e T2 vão para um estado *sleep*, quando acordam, voltam a buscar pelo recurso alvo, que continuará com lock em ambas situações. Dada situação está constituído um livelock .

2.4 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**. (1.0)

```
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 3 – Controle de Concorrência com Locks (40%)

3.1 O problema de atualização perdida (*lost update*) 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:

(a) explicar, em poucas linhas, o problema que está ocorrendo na descrição que segue: (0,5)

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

(b) Seguindo os mesmos nomes e notações acima, montar um quadro similar descrevendo a correção do problema, mostrando as intercalações **serialmente equivalentes** das transações ***T*** e ***U***. (0.5)

(c) Em seguida, montar um quadro descrevendo a correção com equivalência serial, usando as mesmas transações ***T*** e ***U*** com *locks* exclusivos. Aproveite tudo o que fez no quadro precedente. Acrescente apenas os *locks*. (0.5)

3.2 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*. Observe o quadro abaixo e explique: (a) Porque as transações T e U realizando as operações conflitantes *write* e *read*, não são serialmente equivalentes; (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. (1.0)

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)

3.3 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 execução concorrente tem uma incorreção. Responda as seguintes questões:

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*' lock on *B*	*a.withdraw(200);*	waits for *T*' lock on *A*
•••		•••	
•••		•••	

(a) Explique em poucas linhas o porquê não está correta, dizendo qual a incorreção que você identifica. (0.5)

Existe a situação de *deadlock* entre as duas transações T e U. As transações estão esperando e cada uma depende da outra liberar um lock para que possam ser retomadas.

(b) Monte um outro quadro (figura como acima) mostrando a correção de execução das transações ***T*** e ***U***. (1.0)

Figure 13.23
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*