INE5645 – Programação Paralela e Distribuída - Prova 1 – 22/10/2012

Aluno: ____

Parte 1 – Controle de Concorrência – Conceitos Básicos (2,5)

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

1.2 (Verdade/Falso) Itens próprios de **threads** são: Espaço de endereçamento, Variáveis globais, Contador de programa lógico, Registradores, Pilha, Estado.

O espaço de endereçamento de uma thread é o espaço de endereçamento do processo.

1.3 (Verdade/Falso) Thread é uma unidade de gerenciamento de seus recursos.

Quem gerencia recursos é o processo.

1.4 (Verdade/Falso) Preempção é o ato de forçar uma thread a parar de executar no processador.

1.5 (Verdade/Falso) Escalonamento preemptivo é aquele que usa preempção. Time-slicing é preemptivo, mas preempção não implica em time-slicing.

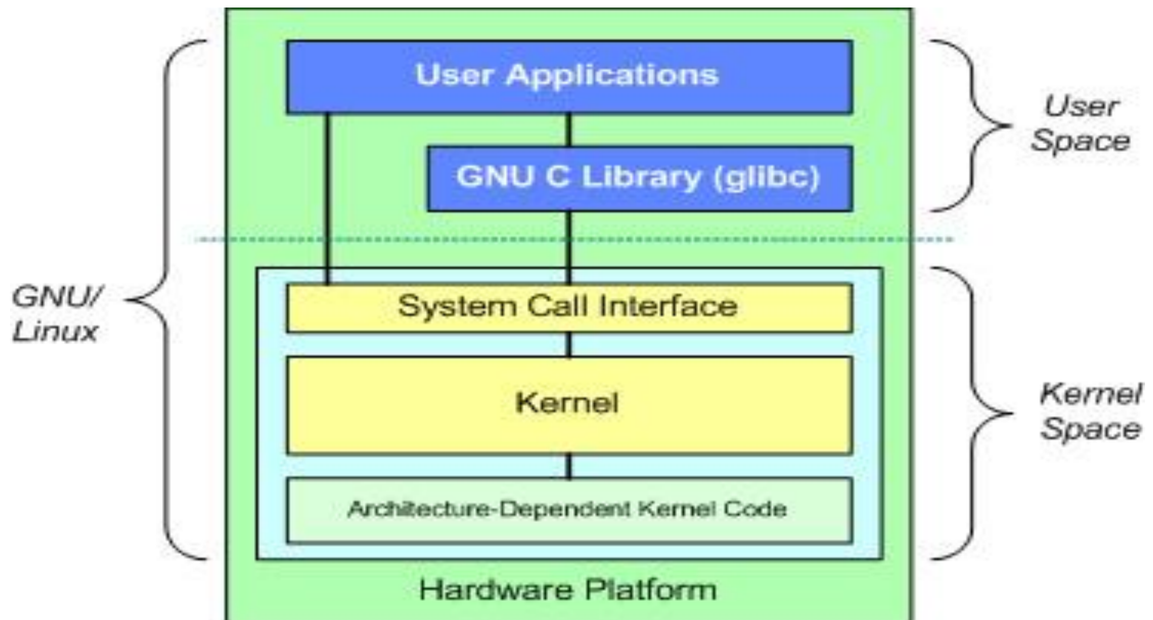
1.6 (Verdade/Falso) No código Java abaixo, o escalonamento é por prioridade de threads:

```
ScheduledThreadPoolExecutor step = new ScheduledThreadPoolExecutor(PoolSize);
    stpe.scheduleAtFixedRate(new WorkerThread("WorkerThread-Executando-At-Fixed-Rate"), 0, 5, TimeUnit.MILISECONDS);
```

Neste caso, é visível que de 5 em 5 MILISECONDS o processador alterna entre threads: scheduleAtFixedRate, ou seja escalona uma thread de 5 em 5 MILISECONDS.

1.7 (Verdade/Falso) As threads que você tem programado são executadas no espaço de endereçamento do usuário, ao invés do espaço do kernel do SO.

Como mostra a figura seguinte, para o exemplo do Linux, mas para o Windows é similar. A JVM do Java aparece como uma aplicação do usuário, sob a execução do IDE que você usa (Eclipse ou Netbeans) e que acessa, diretamente, através das **chamadas de sistema** para se comunicar com o kernel do SO, ou acessa uma GNU C Library, a qual acessa, então, as chamadas de sistema, para chegar ao kernel. Veja a figura abaixo. Note que você (usuário) tem controle total sobre as threads que você programa. E processos ou threads no espaço do kernel, você não tem controle. Neste espaço, quem controla é o SO.



1.8 (Verdade/Falso) A propriedade de Exclusão Mútua e a ausência de deadlock são requisitos críticos de correção que devem sempre ser satisfeitos num sistema concorrente real.

Isto é o que se espera de um programa concorrente.

1.9 (Verdade/Falso) Programar com semáforos é sempre vantajoso em relação a programar com monitor.

Não, monitor, também tem suas vantagens.

A sintaxe de monitores é baseada no encapsulamento de itens de dados e as operações que manipulam esses dados, dentro de um módulo. É um mecanismo de sincronização que concentra a responsabilidade de correção dentro de um módulo (ou de poucos módulos, se mais monitores forem usados). Sua principal vantagem em relação aos outros mecanismos de sincronização é a simplicidade de programação, pois não é necessário bloquear/desbloquear locks ou controlar um grande número de semáforos, num sistema complexo.

1.10 (Verdade/Falso) O tamanho da região crítica (muito pequena ou grande), é um fator que irá influenciar no mecanismo de sincronização a ser utilizado. Num cenário onde temos uma região crítica muito grande, com dezenas ou centenas de instruções, é mais aconselhado usar os monitores ou locks. O ganho de desempenho dos monitores com relação aos locks nestes casos é muito pequeno, mas devido sua facilidade de uso, é mais aconselhado. Semáforos provêem um mecanismo de sincronização, mas que é não estruturado. O uso de semáforos na construção de grandes sistemas é difícil de garantir o uso correto, principalmente sendo utilizado por diversos implementadores do sistema. Monitor é um mecanismo de sincronização que concentra a responsabilidade de correção dentro de um módulo (ou de poucos módulos, se mais monitores forem usados). Sua principal vantagem em relação aos outros mecanismos de sincronização é a simplicidade de programação, pois não é necessário bloquear/desbloquear locks ou controlar um grande número de semáforos.

Parte 2 – Controle de Concorrência – Semáforo (2,5)

2.1 Seja o pseudo-código seguinte:

```
S:  Semaphore := 1;

Thread  T1  is
Begin
     loop
          Non_Critical_Section_1;
          wait(S);
          Critical_Section_1;
          signal(S);
          Non_Critical_Section_1;
     end loop
End T1;

Thread  T2  is
Begin
     loop
          Non_Critical_Section_2;
          wait(S);
          Critical_Section_2;
          signal(S);
          Non_Critical_Section_2;
     end loop
 End T2;
```

Para o programa concorrente do item 2.1, foi demonstrado dois teoremas: Teo1: **O programa não apresenta deadlock.** Teo2: **O programa não apresenta starvation.** Indique, se você acha que existe, uma situação ou circunstância de programação, não considerada no código acima, que pode fazer com que uma das threads entre em ***starvation*** e Teo2 falhe. Sugestão: examine as possibilidades de escalonamento *time-slicing* e escalonamento por prioridades.

No caso *time-slicing* as duas threads são consideradas ter a mesma prioridade. O tempo de processamento de cada uma, na sua seção-crítica é o mesmo do time-slicing e foi provado que não existe *starvation*. No caso, de escalonamento por prioridade, por mais desbalanceadas que tais prioridades sejam, não haverá *starvation*. O programa acima tem apenas duas threads e, portanto, só haverá, no máximo, uma thread em espera (a de menor prioridade), e que terá sua seção-crítica executada, quando um signal(S) da thread de maior prioridade for executado. Você pode ver a figura sobre escalonamento em prioridades, mostrada no Cap 23-*Multithreading* (Livro Deitel), como Java funciona. Nesta figura, threads com mais alta prioridades são executadas entre elas, até se esgotarem, possivelmente por *time-slicing*. Passando, após, a executar um thread de menor prioridade.

Parte 3 – Controle de Concorrência com Monitor (2,5)

3. 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 **nao_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**. (3.0)

// O processo ou thread que chamou o monitor é suspenso na fila FIFO associada com a variável condition não-zero.

// Se a fila para a variável condition não-zero é não vazia, então acorde o processo na cabeça da fila.

Solução: Algo como poderia ser escrito:

```
Monitor  Emulação_Semáforo

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.
```

Parte 4 – Controle de Concorrência com Locks (2,5)

4. 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 o nome da incorreção que você identifica. (0.5)

. Existem duas variáveis que são manipuladas por T e U. Vejam que T bloqueia “a” e U bloqueia “b”. Depois, T espera por U desbloquear “b” e U espera por T desbloquear “a”. As duas transações T e U estão em deadlock, uam espera pela outra.

(b) Corrija o programa concorrente acima. Monte um outro quadro descritivo mostrando a correção de execução das transações ***T*** e ***U***.

Transação T		Transação U	
Operações	**Travas**	**Operações**	**Travas**
a.deposit (100);	trava de escrita de “a”		
		b.deposit(200)	trava de escrita de “b”
b.withdraw(100);			
....	espera pela trava de U sobre “b”	a.withdraw(200);	espera pela trava de T sobre “a”
	(decorre o tempo)	....	
Desbloqueia “a”	....		
		a.withdraw(200);	travas de escrita de “a”
			desbloqueia “a”, “b”