departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Concorrência e Paralelismo — 1º teste — 2013-11-13

Número: _____ Nome: _____

NOTA IMPORTANTE: Os primeiros dois exercícios (1A e 1B) são exclusivos entre si. Resolva apenas um deles. Se resolver ambos apenas será considerada a resolução do exercício 1A.

1	2	3	4	5	6(a)	6(b)	6(c)	7	8(a)	8(b)	TOTAL
1,5 val	*2 val*	*2 val*	*2 val*	*2 val*	*1 val*	*1,5 val*	*1,5 val*	*2,5 val*	*2 val*	*2 val*	*20 val*

1A. [1,5 val] Considere que inicialmente 'x=0', 'y=0' e 'z=0', e que os métodos M1, M2 e M3, são executados em paralelo.
M1: x=x+1; y=y+1; M2: x=y+1; y=2; M3: z=x+y;
Liste todos os valores possíveis para 'x', para 'y' e para 'z' no final da execução.

X:
Y:
Z:

1B. [1,5 val] Considere que inicialmente 'x=0' e 'y=0' e que os métodos M1, M2 e M3, são executados em paralelo.
M1: x=x+1; M2: y=y+1; M3: x=5; y=x+1
Liste todos os valores possíveis para 'x', para 'y', e para os pares '(x,y)' no final da execução.

X:
Y:
(X,Y):

2. [2 val] Indique com um [X] se a seguinte frase é verdadeira ou falsa. Se for verdadeira, explique porquê! Se for falsa, dê um contraexemplo. *"Um programa Java concorrente em que todas as variáveis partilhadas contêm o modificador 'volatile' está necessariamente isento de (low-level) data races."*

Verdadeira [ ] Falsa [ ]
Justificação:

3. [2 val] Considere o exemplo "Too-Much-Milk-Problem" apresentado nas aulas teóricas de 24-09-2013. Classifique cada uma das seguintes frases relativas àquele exemplo com "[S]" ou "[L]", conforme representem uma propriedade de "*Safety*" ou de "*Liveness*" respectivamente. *(Se errar não desconta!)*

[ ]	O Bob e a Alice não vão às compras no mesmo dia.
[ ]	Se o Bob chegar a casa antes da Alice, é ele quem vai às compras.
[ ]	Nem o Bob nem a Alice foram às compras.
[ ]	No final do dia, há sempre pelo menos uma garrafa de leite no frigorífico.
[ ]	No final do dia, nunca haverá duas garrafas de leite no frigorífico.

4. [2 val] Faça corresponder a cada uma das propriedades (identificadas com números) um exemplo/definição (identificados com letras) que se lhe aplique:

1:
2:
3:
4:
5:
6:
7:
8:

1	Lock-freedom	A	Não há dois processos a entrar na secção crítica ao mesmo tempo.
2	Mutual Exclusion	B	Nunca acontece nada de mal.
3	Liveness	C	Se existe um processo a tentar entrar na região crítica, então um processo, não necessariamente o mesmo, irá entrar na região crítica.
4	Safety	D	Alguma coisa boa acontecerá.
5	Deadlock-freedom	E	Se um processo tentar entrar na região crítica, ele acabará por entrar.
6	Wait-freedom	F	Todas as operações do algoritmo executarão num número finito de passos.
7	Obstruction-freedom	G	Quando os threads do programa executarem por um tempo suficientemente grande, pelo menos um dos threads progredirá (para uma definição sensata de "progresso").
8	Starvation-freedom	H	Em qualquer ponto, um único thread que execute um número necessário de passos (limitado) em isolamento (i.e., com todos os demais threads suspensos) completará a sua operação.

5. [2 val] Considere a seguinte implementação de uma barreira simplificada em Java:

```
public class SimpleBarrier {
   public synchronized void block()      {  wait();  }
   public synchronized void release()    {  notify(); }
   public synchronized void releaseAll() {  notifyAll();  }
} // Barrier.
```

Esta classe poderá ser utilizada conforme o exemplo em baixo à esquerda onde, assumindo que 'B.block()' foi executado primeiro, o *thread1* esperará até que o *thread2* invoque 'B.release()':

init: SimpleBarrier B=new SimpleBarrier();

thread1:	**thread2:**
…	…
B.wait();	…
(…)	B.release();
…	…

thread1:	**thread2:**
…	…
this.block();	…
(…)	this.release();
…	…

Explique porque é que é necessário uma classe 'Barrier' e um objecto 'B' para o programa funcionar como pretendido! Ou seja, porque é que o programa não pode ser simplesmente como apresentado acima à direita (na caixa com borda a preto)?

6. Considere um programa com três *threads* que são executados em paralelo em três processadores:

init: int a=0, b=0, c= 0;

T1	**T2**	**T3**
(1) a=1;	(3) b=1;	(5) c=1;
(2) output(b, c);	(4) output(a, c);	(6) output(a, b);

A função 'output()' lê o valor corrente das variáveis recebidas como argumento e escreve esses valores atomicamente no ecrã (i.e., o output das instruções (2), (4) e (6) não se mistura). Por exemplo, se as instruções forem executas pela ordem (1), (2), (3), (4), (5), (6), no terminal será apresentado 001011.

(a) [1 val] Considere apenas as instruções de afectação (1), (3) e (5). Quantos entrelaçamentos diferentes destas instruções poderá o programa gerar? *(Pergunta de escolha múltipla… se errar não desconta!)*

[ ] 1 [ ] 3 [ ] 6 [ ] 8 [ ] 9 [ ] 12

(b) [1,5 val] Existe algum entrelaçamento que, preservando a ordem de execução canónica, produza o resultado 111111? Se sim, apresente o entrelaçamento em causa. Se não, explique porquê.

(c) [1,5 val] Existe algum entrelaçamento, que preserve ou não a ordem de execução canónica, que produza o resultado 011001? Se sim, apresente o entrelaçamento em causa. Se não, explique porquê.

7. [2,5 val] Complete o código da classe "BarrierN" que implementa uma barreira convencional. Esta barreira é inicializada com um valor N e tem um único método 'arrive()'. Todos os threads que invocam o método ficam bloqueados excepto o N-ésimo. Quando este thread invocar o método todos prosseguem. Exemplo de utilização:

init: BarrierN BN=new BarrierN(3);

thread1:	**thread2:**	**thread3:**
…	…	…
BN.arrive();	…	…
(…)	BN.arrive();	…
(…)	(…)	BN.arrive();
…	…	…

```
public class BarrierN
{
   private int a, b;

   public BarrierN (int howmany)  { _____ = howmany;    _____ = _____;  }
   public synchronized void arrive()   {
      _____;
      _____ (a < b) {   _____();   return;  }
      _____();
   }
} // BarrierN
```

8. O programa seguinte simula um banco onde cada titular tem 3 contas. O método 'transferTo()' permite transferir dinheiro entre contas do mesmo titular e o método 'interest()' calcula o valor dos juros a creditar pelo total das contas desse titular. Considere um programa que tem dois threads, que executam concorrentemente os seguintes métodos M1 e M2:

```
1.    M1: transferTo(acc[0], acc[1], 5);
2.    M2: acc[0].credit( interest(0.01) );

3.    Account acc[3];  // array the 3 contas do mesmo titular

4.    void transferTo(Account a, Account b, int v) {
5.       a.debit(v);     // 'debit()' is a synchronized/atomic operation
6.       b.credit(v);    // 'credit()' is a synchronized/atomic operation
7.    }

8.    float interest(float rate) {
9.       float x=0;
10.      synchronized (acc[0]) {  synchronized (acc[1]) {  synchronized (acc[2]) {   // get lock over all the accounts
11.         for (int i=0; i < 3; i++) {  x += acc[i] * rate;   }
12.      } } }
13.      return x;
14.   }
```

a) [2 val] Este programa tem um *High-Level data race*. Explique onde e porquê (use os números de linha para referenciar o programa).

b) [2 val] Este programa pode dar origem a *deadlocks*. Explique como (use os números de linha para referenciar o programa).