

### Pergunta 1



Considere os seguintes excertos de um programa cliente-servidor. Considere ainda a existência de dois clientes que se conectam simultaneamente ao servidor e indique as afirmações verdadeiras.

Cliente	Servidor
<pre>BufferedReader in = new BufferedReader(     new InputStreamReader(         socket.getInputStream()));  boolean connected = true;  while (connected) {     String line = in.readLine();      if (line.equals("zero")) {         connected = false;     } else System.out.println(line); }  socket.close();</pre>	<pre>while (true) {     Socket socket = ss.accept();      PrintWriter out = new PrintWriter(socket.getOutputStream());      String lines[] = {"m1", "m2", "m3"};      for (String line : lines) {         out.println(line);         out.flush();     } }</pre>

Respostas Seleccionadas: É possível que um cliente produza o seguinte output:

m1

m2

☒ m3

☒ Os clientes nunca chegam a fazer socket.close();

Respostas: É possível que um cliente produza o seguinte output:

m1

m2

☒ m3

☒ Os clientes nunca chegam a fazer socket.close();

O segundo cliente a ligar pode produzir o output:

m3

m1

m2

O servidor só atende o primeiro cliente.

### Pergunta 2



Considere os seguintes fragmentos de código. Selecione as afirmações verdadeiras, supondo threads concorrentes a executar partindo do estado inicial envolvendo contas c1, c2 e c3, com saldos iniciais de 100.

```
class Conta {
    int saldo;
    Lock l = new ReentrantLock();
    Condition c = l.newCondition();
}

void transfer(Conta a, Conta b, int quantia) throws InterruptedException {
    a.l.lock();
    b.l.lock();
```

← OK

```

while (a.saldo < quantia)
    a.c.await();
a.saldo -= quantia;
b.saldo += quantia;
b.l.unlock();
a.l.unlock();
}

void deposito(Conta a, int quantia) {
    a.l.lock();
    a.saldo += quantia;
    a.c.signalAll();
    a.l.unlock();
}

```

Respostas Seleccionadas: Podem ficar thread(s) permanentemente bloqueadas com a execução concorrente (denotada por ||) das três operações:  
transfer(c1, c2, 200) || transfer(c2, c3, 200) || { deposito(c1, 100); deposito(c2, 100) }



Podem ficar thread(s) permanentemente bloqueadas com a execução concorrente (denotada por ||) das quatro operações:  
transfer(c1, c2, 200) || transfer(c2, c3, 150) || transfer(c2, c3, 150) || deposito(c1, 100);



Respostas: Podem ficar thread(s) permanentemente bloqueadas com a execução concorrente (denotada por ||) das três operações:  
transfer(c1, c2, 200) || transfer(c2, c3, 200) || { deposito(c1, 100); deposito(c2, 100) }

Podem ficar thread(s) permanentemente bloqueadas com a execução concorrente (denotada por ||) das quatro operações:  
transfer(c1, c2, 200) || transfer(c2, c3, 150) || transfer(c2, c3, 150) || deposito(c1, 100);



Podem ficar thread(s) permanentemente bloqueadas com a execução concorrente (denotada por ||) das três operações:  
transfer(c1, c2, 50) || transfer(c1, c2, 50) || transfer(c1, c3, 50)

Podem ficar thread(s) permanentemente bloqueadas com a execução concorrente (denotada por ||) das três operações:  
transfer(c1, c2, 50) || transfer(c2, c3, 50) || transfer(c3, c1, 50)



### Pergunta 3



Considere o seguinte código, em que os métodos serão usados por múltiplos threads concorrentes:

```

List<Integer> lista = new ArrayList<>();
(1) _____

```

```

int get (int index) {
    (2) _____
    try {
        return lista.get(index);
    } finally {
        (3) _____
    }
}

```

```

void put (int value) {
    (4) _____
    try {
        lista.add(value);
    } finally {
        (5) _____
    }
}

```

Indique quais as opções corretas que preenchem os blocos de código em falta e permitem um correto funcionamento deste programa.

Respostas Seleccionadas: (1) Lock **wl** = new ReentrantLock(); Lock **rl** = new ReentrantLock();  
(2) **rl**.lock();  
(3) **rl**.unlock();  
(4) **wl**.lock();  
(5) **wl**.unlock();



(1) ReadWriteLock **lock** = new ReentrantReadWriteLock();  
(2) **lock**.readLock().lock();  
(3) **lock**.readLock().unlock();  
(4) **lock**.writeLock().lock();  
(5) **lock**.writeLock().unlock();



Respostas: (1) ReadWriteLock **lock** = new ReentrantReadWriteLock(); Lock **wl** = lock.writeLock(); Lock **rl** = lock.readLock();  
(2) **wl**.lock();  
(3) **wl**.unlock();  
(4) **rl**.lock();  
(5) **rl**.unlock();



(1) Lock **wl** = new ReentrantLock(); Lock **rl** = new ReentrantLock();  
(2) **rl**.lock();  
(3) **rl**.unlock();  
(4) **wl**.lock();  
(5) **wl**.unlock();



(1) Lock **wl** = new ReentrantLock(); Lock **rl** = **wl**;  
(2) **wl**.lock();  
(3) **wl**.unlock();  
(4) **rl**.lock();  
(5) **rl**.unlock();



- (1) `ReadWriteLock lock = new ReentrantReadWriteLock();`  
(2) `lock.readLock().lock();`  
(3) `lock.readLock().unlock();`  
(4) `lock.writeLock().lock();`  
✔ (5) `lock.writeLock().unlock();`

#### Pergunta 4



Considere um sistema distribuído em que há migração de código do servidor, que guarda e manipula estado persistente, para um cliente interativo, com uma interface do utilizador. Esta migração:

- Respostas Seleccionadas: ✔ contribui principalmente para a redução da latência percebida pelo utilizador  
✘ contribui principalmente para a redução da quantidade de dados transferidos na rede
- Respostas: ✔ resulta num compromisso entre a eficiência de execução do código migrado e a segurança do cliente  
✔ contribui principalmente para a redução da latência percebida pelo utilizador  
contribui principalmente para a redução da quantidade de dados transferidos na rede  
dispensa a utilização de mecanismos de serialização de dados

#### Pergunta 5



Considere o seguinte código-fonte que simula o comportamento de uma máquina de café. A máquina tem uma capacidade máxima de 100 unidades, e disponibiliza dois métodos:

- **tirarCafe()**: remove da máquina uma unidade de café, bloqueando se esta estiver vazia;
- **reabasterMaquina(N)**: abastece a máquina com N unidades de café.

Assuma que a máquina nunca deve ultrapassar o limite da sua capacidade. Considere também que múltiplas threads podem invocar o método **tirarCafe** mas apenas uma thread pode invocar o método **reabastecerMaquina**.

```
int quantidade = 0;
int capacidade = 100;
Lock lock = new ReentrantLock();
Condition cond1 = lock.newCondition();
Condition cond2 = lock.newCondition();

void tirarCafe () { void reabastecerMaquina (int unidades) {
    lock.lock();      lock.lock();
    (1) _____ (3) _____
    quantidade--;      quantidade += unidades;
    (2) _____ (4) _____
    lock.unlock();     lock.unlock();
}                    }
```

Selecione as opções que completam as funções de modo a garantir o funcionamento correto da máquina.

- Respostas Seleccionadas: (1) `while (quantidade > 0) { cond1.await(); }`  
(2) `if (quantidade==0) { cond2.signalAll(); }`  
(3) `while (quantidade + unidades > capacidade) { cond2.await(); }`  
✘ (4) `cond1.signalAll();`
- Respostas: (1) `while (quantidade==0) { cond2.await(); }`  
(2) `cond2.signalAll();`  
(3) `while (quantidade + unidades > capacidade) { cond1.await(); }`  
(4) `cond1.signalAll();`  
(1) `while (quantidade > 0) { cond1.await(); }`  
(2) `if (quantidade==0) { cond2.signalAll(); }`  
(3) `while (quantidade + unidades > capacidade) { cond2.await(); }`  
(4) `cond1.signalAll();`  
(1) `while (quantidade==0) { cond1.await(); }`  
(2) `cond2.signalAll();`  
(3) `while (quantidade + unidades > capacidade) { cond2.await(); }`  
✔ (4) `cond1.signalAll();`  
(1) `while (quantidade==0) { cond1.await(); }`  
(2) `if (quantidade==0) { cond2.signalAll(); }`  
(3) `cond2.await();`  
(4) `cond1.signalAll();`

### Pergunta 6



Considere o desempenho (*performance*) de programas concorrentes que usam primitivas de sincronização para delimitar e controlar o acesso a secções críticas.

Respostas  
Selecionadas:  
Respostas:

- ✔ O desempenho depende da probabilidade de diferentes threads tentarem usar a mesma secção crítica ao mesmo tempo
  - ✔ A utilização de um R/W lock é vantajosa quando o número de acessos à secção crítica de leitura é muito maior que o número de acessos para escrita
  - ✔ O desempenho depende da probabilidade de diferentes threads tentarem usar a mesma secção crítica ao mesmo tempo
- A utilização de um R/W lock em vez de um mutex (e.g., ReentrantLock) é mais vantajosa quanto mais pequena for a duração da secção crítica para os leitores
- O desempenho depende apenas do número de threads que são iniciados ao mesmo tempo

### Pergunta 7



O protocolo *2-phase commit* (2PC) obriga a escritas em disco (memória persistente):

- Respostas Selecionadas: ✔ no coordenador antes de enviar a decisão no início da segunda fase
- Respostas:
- ✔ no máximo uma vez no decurso de uma transação com sucesso
  - ✔ para garantir que todos os participantes decidem o mesmo resultado
  - ✔ no participante a seguir a enviar qualquer resposta na primeira fase
  - ✔ no coordenador antes de enviar a decisão no início da segunda fase

### Pergunta 8



A utilização de um serviço de nomes num sistema distribuído contribui para a *transparência de localização* num sistema cliente-servidor...

- Respostas Selecionadas: ✔ porque o programa cliente não precisa de conhecer o endereço físico do servidor remoto
- Respostas:
- ✔ porque o programa servidor não precisa de conhecer de antemão o endereço físico do cliente
  - ✔ ao encapsular a tradução de endereços lógicos para endereços físicos
  - ✔ porque o programa cliente não precisa de conhecer o endereço físico do servidor remoto
  - ✔ porque o programa cliente não precisa de conhecer o nome lógico do servidor remoto