Pergunta 1

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 Selecionadas:

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

(5) **lock**.writeLock().unlock();

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

(5) **wl**.unlock();

Respostas:

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

(5) **lock**.writeLock().unlock();

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

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

(5) **wl**.unlock();

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

Pergunta 2

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

Respostas Selecionadas:
- ✗ 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

Respostas:
- O desempenho depende da apenas do número de threads que são iniciados 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
- ✓ 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

---

Pergunta 3

O *middleware* de invocação remota (RPC/RMI) contribui para a *transparência de acesso* num sistema cliente-servidor...

Respostas Selecionadas:
- ✗ porque o programa cliente não precisa de conhecer o endereço físico do servidor remoto
- ✗ porque o programa servidor não precisa de conhecer o endereço físico do cliente

Respostas:
- porque o programa cliente não precisa de conhecer o endereço físico do servidor remoto
- ✓ ao encapsular/esconder a serialização de dados e os detalhes da comunicação entre cliente e servidor
- ✓ porque o programa cliente usa um procedimento/função/método remoto no servidor da mesma forma que usa um local
- porque o programa servidor não precisa de conhecer o endereço físico do cliente

---

Pergunta 4

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

Respostas Selecionadas:
- ✗ no participante a seguir a enviar qualquer resposta na primeira fase
- ✓ no coordenador antes de enviar a decisão no início da segunda fase

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

Pergunta 5

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

```
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();
```

Servidor

```
while (true) {
    Socket socket = ss.accept();

    PrintWriter out = new PrintWriter(
            socket.getOutputStream());

    String lines[] = {"m1", "m2", "zero", "m3"};

    for (String line : lines) {
        out.println(line);
        out.flush();
    }
}
```

Respostas Selecionadas:
- ✓ Remover out.flush(); afeta o output produzido pelos clientes.
  Ambos os clientes produzem o seguinte output:

  m1
  m2
- ✗ m3

Respostas:
- ✓ Nenhum cliente chega a ler a mensagem "m3".
- O servidor atende concorrentemente os dois clientes
- ✓ Remover out.flush(); afeta o output produzido pelos clientes.
  Ambos os clientes produzem o seguinte output:

  m1
  m2
  m3

Pergunta 6

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 Selecionadas:
- ✓ 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

Respostas:
- dispensa a utilização de mecanismos de serialização de dados
- contribui principalmente para a redução da quantidade de dados transferidos na rede
- ✓ 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

Pergunta 7

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();
    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 Selecionadas:
- ✗ 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)

Respostas:
- 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 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(c2, c3, 50) || transfer(c3, c1, 50)
- ✓ 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) }

Pergunta 8

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()**: abastece a máquina com 20 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 () {
  lock.lock();
  (1)_____
  quantidade--;
  (2)_____
  lock.unlock();
}
```

```
void reabastecerMaquina () {
  lock.lock();
  (3)_____
  quantidade += 20;
  (4)_____
  lock.unlock();
}
```

Respostas Selecionadas:

(1) **while** (**quantidade**<=0) { **cond1**.await(); }
(2) **cond2**.signal();
(3) **while** (**quantidade** > **capacidade** - 20) { **cond2**.await(); }
(4) **cond1**.signalAll();

Respostas:

(1) **while** (**quantidade**>0) { **cond2**.await(); }
(2) **cond2**.signalAll();
(3) **while** (**quantidade** + 20 > **capacidade**) { **cond1**.await(); }
(4) **cond1**.signalAll();

(1) **while** (**quantidade** > 0) { **cond1**.await(); }
(2) **if** (**quantidade**==0) { **cond2**.signalAll(); }
(3) **while** (**quantidade** + 20 > **capacidade**) { **cond2**.await(); }
(4) **cond1**.signalAll();

(1) **while** (**quantidade**<=0) { **cond1**.await(); }
(2) **cond2**.signal();
(3) **while** (**quantidade** > **capacidade** - 20) { **cond2**.await(); }
(4) **cond1**.signalAll();

(1) **while** (**quantidade**==0) { **cond1**.await();}
(2) **if** (**quantidade** <= 20) { **cond2**.signalAll(); }
(3) **cond2**.await();
(4) **cond1**.signalAll();