Terceiro Estudo Dirigido

Programação Concorrente (ICP-361) - 2025-2 - Profa. Silvana Rossetto

[1]IC/CCMN/UFRJ – 4 de novembro de 2025

Questão 1 A classe **Java** abaixo implementa um vetor com três métodos de acesso: inserção, remoção e consulta. Seu atributo principal é um *vetor de Object*. Se o vetor estiver cheio, o método de inserção deve bloquear o fluxo de execução até que um espaço seja liberado. O mesmo vale para a remoção, se o vetor estiver vazio, o método de remoção deverá bloquear o fluxo de execução até que um elemento seja inserido no vetor. Os elementos devem ser retirados na mesma ordem em que foram inseridos. Um mesmo elemento não pode ser retirado mais de uma vez e não pode ser sobreescrito antes de ser removido. Considerando o exposto acima e o código apresentado, responda justificando todas as respostas:

(a) Se criarmos um objeto (instância) dessa classe, ele poderá ser compartilhado por várias threads de uma mesma aplicação, garantindo que os métodos executarão corretamente e que todos os requisitos serão contemplados?

(b) Por que a palavra `synchronized` aparece na assinatura dos métodos? Ela poderia ser excluída de algum deles?

(c) As chamadas do método `notifyAll` poderiam ser substituídas por `notify`?

```
class Vetor {
  static final int N = 10; //qtde de elementos no vetor
  private Object[] vetor = new Object[N]; //area de dados compartilhada
  //variaveis de estado
  private int count=0; //qtde de posicoes ocupadas
  private int in=0;  //proxima posicao de insercao
  ptivate int  out=0;  //proxima posicao de retirada

  // Construtor
  Vetor() {... // inicializa o vetor }

  // Insere um item
  public synchronized void Insere (Object item) {
      if (count==N) { //se o vetor estiver cheio, bloqueia o fluxo
           wait();
      }
      vetor[in%N] = item; //insere o elemento e ajusta o estado
      in++;
      count++;
      notifyAll(); //sinaliza possiveis fluxos esperando para remocao
  }
  // Remove um item
  public synchronized Object Remove (int id) {
     int aux;
     if (count==0) { //se nao ha elementos, bloqueia o fluxo
          wait();
     }
     aux = vetor[out%N];  //retira o elemento e ajusta o estado
     out++;
     count--;
     notifyAll();  //sinaliza possiveis fluxos esperando para inserção
     return aux;
  }
  // Informa a quantidade de elementos no vetor
  public synchronized Quantidade () {
     return count;
  }
}
```

Questão 2 A classe **Java** abaixo implementa um `monitor` para resolver o problema dos leitores e escritores com a escrita sendo feita diretamente dentro do monitor. Os requisitos do problema continuam sendo os seguintes: mais de um leitor pode ler ao mesmo tempo uma área de dados compartilhada, mas apenas um escritor pode escrever de cada vez nessa mesma área; e enquanto o escritor está escrevendo os leitores não podem ler. Antes de ler a área de dados compartilhada, as threads leitoras deverão chamar o método `EntraLeitor()` e após terminar a leitura deverão chamar o método `SaiLeitor()`. As threads escritoras deverão chamar o método `Escrita()` passando o dado que deve ser escrito. **Tarefa:** Responda as questões abaixo, justificando suas respostas:

(a) Essa implementação do problema está correta? Atende corretamente a todos os requisitos do problema?
(b) Todas as possibilidades de execução concorrente das threads são garantidas nessa solução?
(c) O que acontecerá se a chamada ao método `notify()` da linha 16 for excluída?
(d) Há possibilidade de ocorrência de *starvation/inanição* neste código?

```
1: class LEMonitor {
2:   private int leit;
3:   LEMonitor() { this.leit = 0; }

4:   //Entrada para leitores
5:   public synchronized void EntraLeitor() {
        this.leit++; }

6:   //Saida para leitores
7:   public synchronized void SaiLeitor() {
8:     this.leit--;
9:     if (this.leit == 0)
10:        this.notify();
11:  }

12:  // Metodo para escritores
13:  public synchronized void Escrita(String str) {
14:     while (this.leit>0) { this.wait(); }
15:     //realiza a escrita de 'str'
16:     this.notify();
17:  } }
```

Questão 3 O programa **Java** mostrado abaixo implementa a classe `Conta` que armazena no atributo `saldo` o saldo de uma conta. A classe oferece métodos para: obter o saldo (`get`), alterá-lo (`set`), incrementar (`incrementa`) ou decrementar o saldo (`decrementa`). No programa principal, são disparadas duas threads (foo e bar) distintas que compartilham uma única instância de `Conta`. Uma dessas threads é uma instância da classe `Bar` que irá verificar se o saldo da conta é superior a uma quantia dada e, caso seja, decrementará esta quantia da conta. Essa verificação existe para evitar que a conta fique com o saldo negativo, que seria um estado errôneo. A outra thread é uma instância da classe `Foo`, e tem como função configurar o saldo da conta para um novo valor. Ao final da execução do programa, a thread principal imprime o valor final do saldo na saída padrão.

```
class Conta {
  private double saldo;
  Conta() {  this.saldo = 0;  }
  public synchronized double get ()                  { return this.saldo;   }
  public synchronized void set (double saldo)        { this.saldo = saldo;  }
  public synchronized void incrementa (double valor) { this.saldo += valor; }
  public synchronized void decrementa (double valor) { this.saldo -= valor; }
}
class Foo extends Thread {
  private Conta conta;
  public Foo (Conta conta) { this.conta = conta; }
  public void run() {  conta.set(100); }
}
class Bar extends Thread {
  private Conta conta;
  private double vaquinha;
  public Bar (Conta conta, double vaquinha) {
    this.conta = conta;
    this.vaquinha = vaquinha;
  }
  public void run () {
    if (conta.get() >= vaquinha) {
      conta.decrementa (vaquinha);
    }
  }
}
public class Exemplo {
  public static void main (...) { //argumentos omitidos
    Conta conta = new Conta ();
    conta.set(500);
    Thread bar = new Bar (conta, 500);
    Thread foo = new Foo (conta);
    bar.start(); foo.start();
    bar.join(); foo.join();
    System.out.println (conta.get ());
  }
}
```

Avalie o programa apresentado e responda as questões abaixo:

(a) Em uma execução do programa, o saldo final da conta ficou negativo (situação indesejada). Por que isso aconteceu?

(b) Proponha uma correção para o programa para garantir que o saldo da conta não fique negativo.

Questão 4 Carros vindos do norte e do sul chegam em uma ponte de uma única pista, apenas carros na mesma direção podem compartilhar a ponte, carros em direção oposta devem esperar até a ponte ficar liberada. O último carro a cruzar a ponte libera a travessia para os carros aguardando na direção oposta. O código Java abaixo implementa uma solução para esse problema (o tratamento de exceções foi omitido). Responda as questões abaixo:

(a) Verifique se a lógica dessa solução atende às condições colocadas e garante ausência de condição de corrida, *deadlock* e *starvation* (inanição).

(b) Apresente argumentos detalhados para embasar sua resposta para o item (a).

```
class Ponte {
 private int sulE=0, norteE=0, norteP=0, sulP=0; Ponte() {//construtor}
 public synchronized void EntraSul() { //entrada dos carros do sul
   while((norteP>0) || (norteE>0)) {
      sulE++; wait(); sulE--;
   }
   sulP++;
 }

 public synchronized void SaiSul() { //saida dos carros do sul
   sulP--;
   if((sulP == 0) && (norteE > 0)) notifyAll();
 }

 public synchronized void EntraNorte() { //entrada dos carros do norte
   while((sulP>0) || (sulE>0)) {
      norteE++; wait(); norteE--;
   }
   norteP++;
 }

 public synchronized void SaiNorte() { //saida dos carros do norte
   norteP--;
   if((norteP==0) && (sulE>0)) notifyAll();
 }
```

Questão 5 O código **Java** abaixo implementa uma solução para o problema de gerenciar o acesso a uma impressora.O tratamento de erro foi omitido por simplicidade. A fila de impressão tem tamanho máximo igaul a 5 e os *jobs* são atendidos em ordem de chegada. Em uma aplicação foram implementadas várias threads que simulam os *jobs* e uma thread que simula a impressora. A thread que simula o *job* chama repetidamente o método *WaitPrint()* até que ele retorne `true`. A thread que simula a impressora chama os métodos *WaitJob* (para esperar o próximo *job*) e *EndJob* (depois que termina a impressão do *job*) indefinidamente. Em uma sequência de testes com 10 threads *job* e uma thread impressora, a aplicação terminou corretamente em parte das execuções e não terminou nas demais. **Identifique o(s) erro(s) no código e descreva porque não ocorreu em todas as execuções**.

```
01 class Manager { //...declaração de variaveis
02  Manager() {
03     this.esperando = 0;  //numero de jobs esperando (no maximo 5)
04     this.ocupada = 0;    //estado da impressora (0: livre; 1: ocupada)
05     this.proximoJob = 0; //senha do job no inicio da fila de espera
06     this.ultimoJob = 0;  //proxima senha do job que entrar na fila
07  }

08  public synchronized boolean WaitPrint () {
09    if(this.esperando == 5)
10       return false;
11    int minhaSenha = this.ultimoJob;
12    this.esperando++;
13    this.ultimoJob++;
14    while(this.proximoJob != minhaSenha)
15       wait();
16    this.ocupada = 1;
17    notify();
18    return true;
19  }

20  public synchronized void WaitJob () {
21    while((this.esperando == 0) || (this.ocupada == 0))
22       wait();
23    this.esperando--;
24  }

25  public synchronized void EndJob () {
26    this.ocupada = 0;
27    this.proximoJob++;
28    if(this.esperando > 0)
29       notifyAll();
30  }
}
```

Questão 6 O código **Java** abaixo implementa um *pool de threads* com dois métodos públicos: `execute` (escalona uma tarefa para execução pelo pool) e `shutdown` (encerra o pool depois que todas as tarefas já escalonadas foram finalizadas). O tratamento de exceções foi omitido por simplicidade. Em uma aplicação foi criado um pool de threads com 10 threads e foram disparadas 100 tarefas para execução pelo pool. Em seguida o pool de threads foi encerrado. A aplicação foi executada várias vezes. Ocorreu que em uma das execuções a aplicação não finalizou. Identifique o erro no código e mostre como corrigi-lo.

```
01 class FilaTarefas {
02  private final int nThreads;
03  private final MyPoolThreads[] threads;
04  private final LinkedList<Runnable> queue;
05  private boolean shutdown;

06  public FilaTarefas(int nThreads) {
07    this.shutdown=false;
08    this.nThreads=nThreads;
09    queue=new LinkedList<Runnable>();
10    threads = new MyPoolThreads[nThreads];
11    for (int i=0; i<nThreads; i++) {
12      threads[i] = new MyPoolThreads();
13      threads[i].start();
14    }
15  }

16  public void execute(Runnable r) {
17     synchronized(queue) {
18       if (this.shutdown) return;
19       queue.addLast(r);
20       queue.notify();
21     }
22  }

23  public void shutdown() {
24     synchronized(queue)
25        { this.shutdown=true; }
26     for (int i=0; i<nThreads; i++)
27       threads[i].join();
28  }

29  private class MyPoolThreads extends Thread {
30     public void run() {
31       Runnable r;
32       while (true) {
33         synchronized(queue) {
34           while (queue.isEmpty() && (!shutdown))
35              queue.wait();
36           if(queue.isEmpty() && shutdown)  return;
37           r=(Runnable) queue.removeFirst();
38         }
39         r.run();
40       }
41     }
42  }
}
```