

Terceiro Estudo Dirigido - Gabarito

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

¹IC/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 sobrescrito 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?

Resp.: Não, os trechos de código que verificam quando as threads deverão se bloquear para aguardar até que as condições lógicas da aplicação sejam atendidas deveriam usar while e não if. Isso é necessário para contemplar o caso de existir várias threads que podem retirar e inserir elementos no vetor.

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

Resp.: Trata-se de uma implementação de sincronização por exclusão mútua. Da forma como está, apenas um dos métodos de uma instância da classe poderá ser executado de cada vez. Todos os métodos precisam ser executados com exclusão mútua pois acessam variáveis compartilhadas e tomam decisões em função do valor dessas variáveis.

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

Resp.: Não, pois as threads que acessarão os métodos da classe poderão se bloquear por razões distintas (condições lógicas) na mesma variável de condição, que será o objeto `this`. Se substituirmos o `notifyAll` por `notify`, poderá ocorrer de um tipo de thread ser desbloqueada indevidamente, podendo levar a aplicação a um bloqueio geral das threads.

```
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
    private 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?

Resp.: Sim, está correta. A solução garante acesso com exclusão mútua à variável compartilhada. Mais de um leitor pode ler ao mesmo tempo (invocando os métodos `EntraLeitor` e `SaiLeitor` antes e após o término da leitura, respectivamente). Apenas um escritor pode escrever de cada vez (invocando o método `Escrita`), pois o método garante exclusão mútua entre escritores. Não permite leitura e escrita ao mesmo tempo, pois a escrita está sendo feita dentro do monitor e só é realizada se não houver leitores lendo. Quando o último leitor termina, ele desbloqueia um escritor. Outros escritores são desbloqueados um a um pelo último escritor a escrever. Leitores ficam bloqueados apenas na entrada do monitor, quando a escrita termina, os leitores são liberados.

(b) Todas as possibilidades de execução concorrente das threads são garantidas nessa solução?

Resp.: Sim, pelo exposto acima, mais de um leitor pode passar pelo método `EntraLeitor` e executar a leitura ao mesmo tempo. Não há outras possibilidades de execução concorrente no problema leitores e escritores.

(c) O que acontecerá se a chamada ao método `notify()` da linha 16 for excluída?

Resp.: Se mais de um escritor se bloquear ao encontrar leitores lendo, apenas um deles será desbloqueado quando os leitores terminarem (resultado da linha 10). Os demais permanecerão bloqueados até que nova leitura seja realizada e finalizada, o que é um condição indesejada.

(d) Há possibilidade de ocorrência de *starvation/inanição* neste código? *Resp.: Sim, se leitores estiverem constantemente entrando, mantendo o número de leitores lendo diferente de zero, os escritores poderão ficar em espera por longo tempo.*

```

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?

Resp.: É possível observar uma violação de atomicidade na operação da instância da thread Bar, pois o estado do saldo pode ser alterado pela thread Foo entre a chamada de `get()`, feita na verificação de saldo, e a chamada de decremento, podendo, assim, resultar em um saldo negativo.

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

Resp.: A operação apropriada do programa seria a thread efetuar a checagem de saldo e decremento de forma atômica, e as outras threads só enxergarem o valor final do saldo após estas operações. Para corrigir isso, é necessário garantir a atomicidade na execução do método principal da thread Bar, como mostrado abaixo.

```
public void run () {
    synchronized (conta) {
        //garante execução atômica usando objeto 'conta' (compartilhado pelas threads) como lock
        //(o mesmo objeto que garante exclusão mútua no acesso aos métodos de Conta)
    }
}
```

```

        if (conta.get() >= vaquinha) {
            conta.decrementa (vaquinha);
        }
    }
}

```

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:

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

```

Resp.: O código atende às condições colocadas, impedindo que carros do norte e do sul atravessem a ponte ao mesmo tempo.

O código garante ausência de condição de corrida pois todas as variáveis compartilhadas são acessadas dentro de blocos sincronized do mesmo objeto de lock, garantindo exclusão mútua no acesso a essas variáveis e nas ações decorrentes das mudanças de estado refletidas nessas variáveis.

O código não garante ausência de deadlock. Se, por exemplo, carros do sul chegam primeiro, eles começam a atravessar a ponte. Quando o primeiro carro do norte chegar, ele ficará bloqueado. Em seguida, todos os carros do sul que chegarem também ficarão bloqueados. Quando o último carro do sul sair da ponte, ele desbloqueará todos os carros bloqueados. Até aqui tudo certo. Entretanto, como as variáveis norteE e sulE estarão diferentes de zero, nenhum carro conseguirá mais atravessar a ponte pois todos ficarão bloqueados.

O código tenta evitar ausência de inanição pois impede carros em um mesmo sentido de atravessarem se houver carros na direção oposta já esperando para atravessar a ponte.

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 igual 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 variáveis
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     }
}
```

Resp.: A chamada notify() na última linha do método WaitPrint (linha 17) deveria ser substituída por uma chamada notifyAll(). Como o objeto de sincronização é o mesmo para a thread Print e para as threads Jobs, quando o notify() é chamado ele pode desbloquear uma thread Job, quando o esperado seria (nesse ponto do código) que ele desbloqueasse a thread Print. O motivo do erro não ocorrer em todas as execuções é que ora pode ocorrer do notify() desbloquear a thread Print, ora uma thread Job.

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 }

```

Resp.: O erro está no método shutdown. Depois de alterar a variável this.shutdown para true, o método deveria executar queue.notifyAll() para garantir que todas as threads do pool avaliem o novo estado da aplicação e saiam do método run quando a fila de tarefas ficar vazia. O erro na execução (a aplicação não finaliza) ocorre nos casos em que o método shutdown é executado e pelo menos uma thread do poll já encontra-se bloqueada.