Paradigmas de Linguagens de Programação
Lista 5
Prof. Sergio d Zorzo

---

1. Considere o código Java abaixo:

```
public class Main {
    double a, b, c, discriminante, dividendo, divisor, x;
    class Bhaskara1 extends Thread {
        public void run() {
            double result = b * b - 4 * a * c;
            discriminante = result;
        }
    }
    class Bhaskara2 extends Thread {
        public void run() {
            double result = -b + Math.sqrt(discriminante);
            dividendo = result;
        }
    }
    class Bhaskara3 extends Thread {
        public void run() {
            double result = 2 * a;
            divisor = result;
        }
    }
    class Bhaskara4 extends Thread {
        public void run() {
            double result = dividendo / divisor;
            x = result;
        }
    }

    void executar() {
        a = 2; b = -6; c = -20;
        Bhaskara1 b1 = new Bhaskara1();   Bhaskara2 b2 = new Bhaskara2();
        Bhaskara3 b3 = new Bhaskara3();   Bhaskara4 b4 = new Bhaskara4();
        b1.run();  b2.run();  b3.run();   b4.run();
        System.out.println("Valor de X = " + x);
    }
    public static void main(String[] args) {
        new Main().executar();
    }
}
```

a. Descreva, para os seguintes pares de tarefas, se é necessária sincronização de cooperação ou competição ou nenhuma:

i) Bhaskara1 e Bhaskara2

Cooperação

ii) Bhaskara1 e Bhaskara3

Nenhuma

iii) Bhaskara1 e Bhaskara4

Nenhuma (apesar de indiretamente haver cooperação)

iv) Bhaskara2 e Bhaskara3

Nenhuma

v) Bhaskara2 e Bhaskara4

Cooperação

vi) Bhaskara3 e Bhaskara4

Cooperação

b. Apesar de haverem problemas de cooperação e competição, o programa acima, se executado, funciona corretamente (Experimente!). Porque? O que está errado?

Porque está sendo feita a chamada a run(), ao invés de start(). Isso causa a execução sequencial normal, e portanto os problemas de cooperação e competição não se manifestam.

c. Uma vez corrigido o erro citado no item b, sincronize o programa acima, para realizar o cálculo correto do valor de x e imprimi-lo na tela ao final. Obs: faça todas as sincronizações de cooperação e competição necessárias.

```
public class Main {

    final Object semaforoDiscriminante = new Object();
    boolean discriminanteCalculado = false;
    final Object semaforoDividendo = new Object();
    boolean dividendoCalculado = false;
    final Object semaforoDivisor = new Object();
    boolean divisorCalculado = false;
    final Object semaforoX = new Object();
    boolean xCalculado = false;
    double a, b, c;
    double discriminante;
    double dividendo;
    double divisor;
    double x;

    class Bhaskara1 extends Thread {

        public void run() {
            double result = b * b - 4 * a * c;
            synchronized (semaforoDiscriminante) {
                discriminante = result;
                discriminanteCalculado = true;
                semaforoDiscriminante.notifyAll();
            }
        }
    }

    class Bhaskara2 extends Thread {

        public void run() {
            double result = 0;
            synchronized (semaforoDiscriminante) {
                while (!discriminanteCalculado) {
                    try {
                        semaforoDiscriminante.wait();
                    } catch (InterruptedException ex) {
                    }
                }
                result = -b + Math.sqrt(discriminante);
            }
            synchronized (semaforoDividendo) {
                dividendo = result;
                dividendoCalculado = true;
                semaforoDividendo.notifyAll();
            }
        }
    }

    class Bhaskara3 extends Thread {

        public void run() {
            double result = 2 * a;
            synchronized (semaforoDivisor) {
                divisor = result;
                divisorCalculado = true;
                semaforoDivisor.notifyAll();
            }
        }
    }

    class Bhaskara4 extends Thread {

        public void run() {
            double result = 0;
            synchronized (semaforoDividendo) {
                while (!dividendoCalculado) {
                    try {
                        semaforoDividendo.wait();
                    } catch (InterruptedException ex) {
                    }
```

```
                }
                synchronized (semaforoDivisor) {
                    while (!divisorCalculado) {
                        try {
                            semaforoDivisor.wait();
                        } catch (InterruptedException ex) {
                        }
                    }
                    result = dividendo / divisor;
                }
            }
            synchronized (semaforoX) {
                x = result;
                xCalculado = true;
                semaforoX.notifyAll();
            }
        }
    }

    void executar() throws InterruptedException {
        a = 2;
        b = -6;
        c = -20;
        Bhaskara1 b1 = new Bhaskara1();
        Bhaskara2 b2 = new Bhaskara2();
        Bhaskara3 b3 = new Bhaskara3();
        Bhaskara4 b4 = new Bhaskara4();
        b1.start();
        b2.start();
        b3.start();
        b4.start();
        synchronized (semaforoX) {
            while (!xCalculado) {
                semaforoX.wait();
            }
            System.out.println("Valor de X = " + x);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        new Main().executar();
    }
}
```

d. No programa acima, a tarefa Bhaskara2 calcula somente o valor positivo da raiz do discriminante. No entanto, a fórmula de Bhaskara envolve o cálculo de dois valores para a raiz do discriminante, um positivo e um negativo. Veja:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Ou seja, para uma mesma tripla <a,b,c>, dois valores de x são possíveis. Modifique o programa acima, para calcular dois valores de x em paralelo, mantendo as sincronizações necessárias.

```
public class Main {

    final Object semaforoDiscriminante = new Object();
    boolean discriminanteCalculado = false;
    final Object semaforoDividendoPositivo = new Object();
    boolean dividendoPositivoCalculado = false;
    final Object semaforoDividendoNegativo = new Object();
    boolean dividendoNegativoCalculado = false;
    final Object semaforoDivisor = new Object();
    boolean divisorCalculado = false;
    final Object semaforoXPositivo = new Object();
    boolean xPositivoCalculado = false;
    final Object semaforoXNegativo = new Object();
    boolean xNegativoCalculado = false;
    double a, b, c;
    double discriminante;
    double dividendoPositivo;
    double dividendoNegativo;
```

```
    double divisor;
    double xPositivo;
    double xNegativo;

    class Bhaskara1 extends Thread {

        public void run() {
            double result = b * b - 4 * a * c;
            synchronized (semaforoDiscriminante) {
                discriminante = result;
                discriminanteCalculado = true;
                semaforoDiscriminante.notifyAll();
            }
        }
    }

    class Bhaskara2Positivo extends Thread {

        public void run() {
            double result = 0;
            synchronized (semaforoDiscriminante) {
                while (!discriminanteCalculado) {
                    try {
                        semaforoDiscriminante.wait();
                    } catch (InterruptedException ex) {
                    }
                }
                result = -b + Math.sqrt(discriminante);
            }
            synchronized (semaforoDividendoPositivo) {
                dividendoPositivo = result;
                dividendoPositivoCalculado = true;
                semaforoDividendoPositivo.notifyAll();
            }
        }
    }

    class Bhaskara2Negativo extends Thread {

        public void run() {
            double result = 0;
            synchronized (semaforoDiscriminante) {
                while (!discriminanteCalculado) {
                    try {
                        semaforoDiscriminante.wait();
                    } catch (InterruptedException ex) {
                    }
                }
                result = -b - Math.sqrt(discriminante);
            }
            synchronized (semaforoDividendoNegativo) {
                dividendoNegativo = result;
                dividendoNegativoCalculado = true;
                semaforoDividendoNegativo.notifyAll();
            }
        }
    }

    class Bhaskara3 extends Thread {

        public void run() {
            double result = 2 * a;
            synchronized (semaforoDivisor) {
                divisor = result;
                divisorCalculado = true;
                semaforoDivisor.notifyAll();
            }
        }
    }

    class Bhaskara4Positivo extends Thread {

        public void run() {
            double result = 0;
            synchronized (semaforoDividendoPositivo) {
                while (!dividendoPositivoCalculado) {
                    try {
```

```
                    semaforoDividendoPositivo.wait();
                } catch (InterruptedException ex) {
                }
            }
            synchronized (semaforoDivisor) {
                while (!divisorCalculado) {
                    try {
                        semaforoDivisor.wait();
                    } catch (InterruptedException ex) {
                    }
                }
                result = dividendoPositivo / divisor;
            }
        }
        synchronized (semaforoXPositivo) {
            xPositivo = result;
            xPositivoCalculado = true;
            semaforoXPositivo.notifyAll();
        }
    }
}

class Bhaskara4Negativo extends Thread {

    public void run() {
        double result = 0;
        synchronized (semaforoDividendoNegativo) {
            while (!dividendoNegativoCalculado) {
                try {
                    semaforoDividendoNegativo.wait();
                } catch (InterruptedException ex) {
                }
            }
            synchronized (semaforoDivisor) {
                while (!divisorCalculado) {
                    try {
                        semaforoDivisor.wait();
                    } catch (InterruptedException ex) {
                    }
                }
                result = dividendoNegativo / divisor;
            }
        }
        synchronized (semaforoXNegativo) {
            xNegativo = result;
            xNegativoCalculado = true;
            semaforoXNegativo.notifyAll();
        }
    }
}

void executar() throws InterruptedException {
    a = 2;
    b = -6;
    c = -20;
    Bhaskara1 b1 = new Bhaskara1();
    Bhaskara2Positivo b2p = new Bhaskara2Positivo();
    Bhaskara2Negativo b2n = new Bhaskara2Negativo();
    Bhaskara3 b3 = new Bhaskara3();
    Bhaskara4Positivo b4p = new Bhaskara4Positivo();
    Bhaskara4Negativo b4n = new Bhaskara4Negativo();
    b1.start();
    b2p.start();
    b2n.start();
    b3.start();
    b4p.start();
    b4n.start();
    synchronized (semaforoXPositivo) {
        while (!xPositivoCalculado) {
            semaforoXPositivo.wait();
        }
        System.out.println("Valor de X = " + xPositivo);
    }
    synchronized (semaforoXNegativo) {
        while (!xNegativoCalculado) {
            semaforoXNegativo.wait();
        }
```

```
            System.out.println("Valor de X = " + xNegativo);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        new Main().executar();
    }
}
```

2. Suponha que duas tarefas A e B devem usar uma variável compartilhada BUF_SIZE. A tarefa A adiciona 2 a BUF_SIZE e a tarefa B subtrai 1 de BUF_SIZE. Assuma que essas operações aritméticas são feitas pelo processo de três passos:
   - buscar o valor atual
   - efetuar a operação
   - armazenar de volta o resultado

   Não havendo sincronização de competição, quais as sequências de eventos que são possíveis e quais são os valores resultantes dessas operações? Assuma que o valor inicial de BUF_SIZE é 6.

```
Seja:
 A1 = Ler BUF_SIZE
 A2 = Somar 2
 A3 = Atribuir resultado a BUF_SIZE
 B1 = Ler BUF_SIZE
 B2 = Subtrair 1
 B3 = Atribuir resultado a BUF_SIZE

Porém, só são relevantes os eventos A1 e A3, já que A2 não
  interfere em BUF_SIZE.

Resultados possíveis:

A1 -> A3 -> B1 -> B3 -> BUF_SIZE = 7
A1 -> B1 -> A3 -> B3 -> BUF_SIZE = 5
A1 -> B1 -> B3 -> A3 -> BUF_SIZE = 8
B1 -> A1 -> A3 -> B3 -> BUF_SIZE = 5
B1 -> A1 -> B3 -> A3 -> BUF_SIZE = 8
B1 -> B3 -> A1 -> A3 -> BUF_SIZE = 7

Todas as outras combinações envolvendo A2 e B2 levam a um
  desses resultados.
```

3. Considere o programa Java abaixo:

```
public class Main2 {
    final static int CAPACIDADE_BUFFER = 10;
    int[] buffer = new int[CAPACIDADE_BUFFER];
    int fim = 0;
    int ini = 0;
    volatile int n = 0;
    class Produtor extends Thread {
        public void run() {
            for (int i = 0; i < 1000; i++) {
                while (n == CAPACIDADE_BUFFER) { }
                buffer[fim] = produzir(i);
                fim = (fim + 1) % CAPACIDADE_BUFFER;
                n++;
            }
        }
        private int produzir(int valor) { return valor; }
    }

    class Consumidor extends Thread {
        public void run() {
            for (int i = 0; i < 1000; i++) {
                while (n == 0) { }
                consumir(buffer[ini]);
                ini = (ini + 1) % CAPACIDADE_BUFFER;
                n--;
            }
        }
        private void consumir(int valor) {
            for(int i:buffer) { System.out.print(i+" ");  }
            System.out.println(" - Consumindo " + valor);
            System.out.flush();
        }
    }

    void executar() {
        Produtor p = new Produtor();
        Consumidor c = new Consumidor();
        p.start();    c.start();
    }
    public static void main(String args[]) {
        new Main2().executar();
    }
}
```

Note que o compartilhamento da variável n pode gerar inconsistências, pois produtor e consumidor estão acessando-a ao mesmo tempo, um para incrementar e outro para decrementar. Utilize um monitor para evitar esse problema, garantindo acesso mutuamente exclusivo à região crítica.

- Obs1: O problema existe, apesar de raramente acontecer. Lembre-se de que a instrução `x++` na verdade envolve uma leitura, modificação e escrita, em sequência. Entre essas três instruções pode acontecer de outra Thread acessar o valor de x depois da leitura, mas antes da escrita, apesar de isso ser raro. Para poder ver isso mais facilmente, substitua as instruções `n++` e `n--` pelo seguinte trecho:

```
// n++; comente essa linha, substituindo-a pelas próximas (idem para n--)
int temp = n;
try {
  Thread.sleep(13); // no n--, coloque sleep(11), para aumentar a chance de conflito
} catch (InterruptedException ex) { }
temp ++; // ou temp--
try {
  Thread.sleep(13);
} catch (InterruptedException ex) { }
n = temp;
```

- **Item bônus – não cai na prova, é apenas para seu aprendizado:** Observe, no programa, que a variável `n` tem um modificador "volatile". Ele não tem importância para a resolução do exercício (nem irá cair na prova). Mas seja curioso: pesquise! Experimente tirá-lo para ver o que acontece. Busque explicações. Não precisa entregar para o professor, e não vale nota, mas aumentará seu conhecimento!

```
// Obs: a solução abaixo apenas funciona para quando temos uma única thread
   produtora e uma consumidora. Caso existam vários produtores e vários
   consumidores, é necessário uma seção crítica maior.

public class Main2 {

    final static int CAPACIDADE_BUFFER = 10;
    int[] buffer = new int[CAPACIDADE_BUFFER];
    int fim = 0;
    int ini = 0;
    volatile int n = 0;
    final Object monitorN = new Object();

    class Produtor extends Thread {

        public void run() {
            for (int i = 0; i < 1000; i++) {
                while (n == CAPACIDADE_BUFFER) {
                }
                buffer[fim] = produzir(i);
                fim = (fim + 1) % CAPACIDADE_BUFFER;
                synchronized (monitorN) {
                    n++;
                }
            }
        }

        private int produzir(int valor) {
            return valor;
        }
    }

    class Consumidor extends Thread {

        public void run() {
            for (int i = 0; i < 1000; i++) {
                while (n == 0) {
                }
                consumir(buffer[ini]);
                ini = (ini + 1) % CAPACIDADE_BUFFER;
                synchronized (monitorN) {
                    n--;
                }
            }
        }

        private void consumir(int valor) {
            for (int i : buffer) {
                System.out.print(i + " ");
            }
            System.out.println(" - Consumindo " + valor);
            System.out.flush();
        }
    }

    void executar() {
        Produtor p = new Produtor();
        Consumidor c = new Consumidor();
        p.start();
        c.start();
    }

    public static void main(String args[]) {
        new Main2().executar();
    }
}
```

4. Modifique o programa acima para usar semáforos e evitar que os laços “while” de produtor-consumidor fiquem constantemente testando o valor de n. Isso, além de desnecessário, piora o desempenho do programa.

```
// Obs: a solução abaixo utiliza vários semáforos e monitores. Veja a questão
   6 para uma versão com apenas um semáforo. É possível utilizar um único
   semáforo pois, coincidentemente, no problema do produtor-consumidor, quando
   o semáforo está “verde” para um lado, está necessariamente “vermelho” para
   o outro lado.

public class Main2 {

    final static int CAPACIDADE_BUFFER = 10;
    int[] buffer = new int[CAPACIDADE_BUFFER];
    int fim = 0;
    int ini = 0;
    volatile int n = 0;
    final Object monitorN = new Object();
    final Object semaforoBufferCheio = new Object();
    final Object semaforoBufferVazio = new Object();

    class Produtor extends Thread {

        public void run() {
            for (int i = 0; i < 1000; i++) {
                synchronized (semaforoBufferCheio) {
                    while (n == CAPACIDADE_BUFFER) {
                        try {
                            semaforoBufferCheio.wait();
                        } catch (InterruptedException ex) {
                        }
                    }
                    buffer[fim] = produzir(i);
                    fim = (fim + 1) % CAPACIDADE_BUFFER;
                    synchronized (monitorN) {
                        n++;
                    }
                }
                synchronized (semaforoBufferVazio) {
                    semaforoBufferVazio.notifyAll();
                }

            }
        }

        private int produzir(int valor) {
            return valor;
        }
    }

    class Consumidor extends Thread {

        public void run() {
            for (int i = 0; i < 1000; i++) {
                synchronized (semaforoBufferVazio) {
                    while (n == 0) {
                        try {
                            semaforoBufferVazio.wait();
                        } catch (InterruptedException ex) {
                        }
                    }
                    consumir(buffer[ini]);
                    ini = (ini + 1) % CAPACIDADE_BUFFER;
                    synchronized (monitorN) {
                        n--;
                    }
                }
                synchronized (semaforoBufferCheio) {
                    semaforoBufferCheio.notifyAll();
                }

            }
        }
```

```
        private void consumir(int valor) {
            for (int i : buffer) {
                System.out.print(i + " ");
            }
            System.out.println(" - Consumindo " + valor);
            System.out.flush();
        }
    }

    void executar() {
        Produtor p = new Produtor();
        Consumidor c = new Consumidor();
        p.start();
        c.start();
    }

    public static void main(String args[]) {
        new Main2().executar();
    }
}
```

5. Vamos agora refazer os exercícios 3 e 4 utilizando uma abordagem mais "OO". Considere a classe Buffer abaixo:

```
public class Buffer {
    private final static int CAPACIDADE_BUFFER = 10;
    private List<Integer> buffer = new LinkedList<Integer>();
    public boolean estaVazio() {
        return buffer.isEmpty();
    }
    public boolean estaCheio() {
        return buffer.size() == CAPACIDADE_BUFFER;
    }
    public void armazenar(int x) {
        if (buffer.size() == CAPACIDADE_BUFFER) {
            throw new RuntimeException("Buffer overflow");
        }
        buffer.add(x);
    }
    public int remover() {
        if (buffer.isEmpty()) {
            throw new RuntimeException("Buffer underflow");
        }
        return buffer.remove(0);
    }
}
```

Note que os métodos dessa classe acessam a lista buffer tanto para escrita como para leitura. Imagine que você, desenvolvedor da classe, não sabe nada a respeito de como esses métodos serão chamados, em que sequência, ou frequência. Mas sabe que um buffer deve ser capaz de ser acessado por múltiplas Threads ao mesmo tempo.

a. Quais situações podem acontecer durante o acesso simultâneo a essa classe?

```
Situação 1: duas threads solicitam armazenamento exatamente no mesmo
   momento. Caso elas façam o teste sobre a capacidade do buffer
   exatamente o mesmo tempo, poderá ocorrer um estouro de capacidade
   (overflow) que não é detectado pela exceção do código acima.
Situação 2: duas threads solicitam remoção exatamente no mesmo
   momento. Caso elas façam o teste sobre a lista vaze exatamente ao
   mesmo tempo, e a lista tiver somente um elemento, poderá ocorrer
   um acesso indevido (underflow) que não é detectado pela exceção do
   código acima.
Situação 3: uma thread solicita leitura enquanto outra solicita
   escrita. Caso as chamadas para buffer.add e buffer.remove (ou
   buffer.isEmpty ou buffer.size()) sejam simultâneas, isso poderá
   acarretar em um erro de acesso concorrente.
```

b. O que acontece se, em cada método, for adicionado um modificador "synchronized" (ex: `public synchronized estaVazio()`, `public synchronized armazenar()`).

Não teremos mais acesso simultâneo aos métodos da classe, ou seja, enquanto uma thread estiver executando um método, nenhuma outra thread conseguirá executar outro método. Isso porque todas as threads acessam o mesmo objeto (this), e os métodos estão sincronizados em torno do mesmo (this).
Como resultado 1: não irão ocorrer mais problemas de overflow ou underflow não detectados pela exceção acima, isto é: todo overflow ou underflow irá ser detectado pelo código acima.
Como resultado 2: não teremos mais problemas de acesso concorrente à lista, já que a mesma é privada, e todos os métodos são sincronizados em torno do objeto corrente (this), que é um objeto diretamente associado à lista.

c. Considere a seguinte afirmação:

Fazendo com que todos os métodos de uma classe sejam "synchronized", automaticamente todo código que a utiliza não terá problemas de sincronização de cooperação ou competição.

Você concorda ou discorda? Por que? (Dica: pense um pouco, responda com sua intuição, depois faça o exercício 6, e confira sua resposta)

Discordo. Ainda podem haver problemas de cooperação ou competição, uma vez que, dependendo da sequência de chamadas, podem ser gerados estados inconsistentes.
Neste exemplo, ainda pode acontecer overflow ou underflow. A diferença é que, com synchronized nos métodos, todo estado inconsistente será DETECTADO pelo código (pelos testes nos métodos "armazenar" e "remover"). Mas o problema persiste, sendo necessário um mecanismo de semáforo.

6. Ainda sobre o problema produtor-consumidor, considere o código Java abaixo, que utiliza a classe Buffer (versão já modificada, com modificadores synchronized) do exercício 5:

```
public class Main2 {
  Buffer buffer = new Buffer();
  final Object semaforoBuffer = new Object();
  class Produtor extends Thread {
    public void run() {
      for (int i = 0; i < 1000; i++) {
        int x = produzir(i);
/* c */ synchronized (semaforoBuffer) {
/* a */   while (buffer.estaCheio()) {
/* b */     try { semaforoBuffer.wait(); } catch (InterruptedException ex) {   }
/* a */   }
          buffer.armazenar(x);
/* b */   semaforoBuffer.notifyAll();
/* c */ }
      }
    }
    private int produzir(int valor) { return valor; }
  }

  class Consumidor extends Thread {
    public void run() {
      for (int i = 0; i < 1000; i++) {
        int x = 0;
/* c */ synchronized (semaforoBuffer) {
/* a */   while (buffer.estaVazio()) {
/* b */     try { semaforoBuffer.wait(); } catch (InterruptedException ex) { }
/* a */   }
          x = buffer.remover();
```

```
/* b */   semaforoBuffer.notifyAll();
/* c */ }
        consumir(x);
      }
    }
    private void consumir(int valor) {
      System.out.println("Consumindo: "+valor);
      // Fazer alguma coisa com o valor
    }
  }

  void executar() {
    new Produtor().start();  new Produtor().start();  new Produtor().start();
    new Produtor().start();  new Produtor().start();
    new Consumidor().start();  new Consumidor().start();  new Consumidor().start();
    new Consumidor().start();  new Consumidor().start();
  }
  public static void main(String args[]) {
    new Main2().executar();
  }
}
```

a. O que pode acontecer se removermos todas as linhas indicadas com `/* b */` (e somente elas), do programa acima?

Teremos potencial situação de loop infinito:
Em um caso, um consumidor ficará infinitamente aguardando um produtor, que nunca irá executar, pois o consumidor está com a chave de acesso única.
Em outro caso, um produtor ficará infinitamente aguardando um consumidor, que nunca irá executar, pois o produtor está com a chave de acesso única.

b. O que pode acontecer se removermos todas as linhas indicadas com `/* a */` e `/* c */` (e somente elas), do programa acima?

Teremos um erro durante a execução, causado pela chamada a notify (ou wait), fora de um contexto sincronizado.

c. O que pode acontecer se removermos todas as linhas indicadas com `/* c */` (e somente elas), do programa acima?

Teremos um erro durante a execução, causado pela chamada a notify (ou wait), fora de um contexto sincronizado.

d. O que pode acontecer se trocarmos os comandos “while” das linhas indicadas com `/* a */` por comandos “if” com as mesmas condições?

Pode acontecer um overflow ou underflow do buffer, já que as chamadas “notifyAll” acordam todas as threads aguardando. Essas threads, por sua vez, ao serem acordadas, não irão testar novamente sua condição de espera.
Por exemplo, podem haver três consumidores aguardando o buffer ser modificado por um produtor. Um produtor, ao produzir um único valor, notifica todos os três consumidores. Estes entram em execução e não testam novamente se o buffer está vazio. Um deles consome o valor, deixando o buffer vazio, e os outros dois irão tentar consumir um valor de um buffer vazio, causando underflow. Uma situação semelhante pode acontecer com produtores em estado de espera, causando overflow.

(Obs: dê as respostas acima considerando as modificações de cada item isoladamente, isto é, as modificações no item a não são repetidas nos itens b, c e d, e assim por diante)

7. Construa uma classe “Reservas” em Java que realiza a reserva e liberação de um número fixo de salas para reunião (não precisa fazer a classe oferecer suporte a qualquer número de salas. Faça-a, por exemplo, considerando que existem somente 5 salas). Supondo que as salas de reunião podem ser reservadas e liberadas de vários pontos de acesso distintos, construa um código que simula diferentes situações de acesso simultâneo. Neste código, inicie várias Threads que são executadas ao mesmo tempo e ficam constantemente reservando e liberando salas.

a. **Item bônus – não cai na prova, é apenas para seu aprendizado:** tente criar casos de teste que demonstram as falhas no tratamento da concorrência em sua classe. Experimente, por exemplo, iniciar várias threads simultaneamente, adicionar esperas em pontos estratégicos do código, imprimir o estado das reservas, etc.

```
class Reservas {

    private String sala1, sala2, sala3;

    public Reservas() {
        sala1 = "livre";
        sala2 = "livre";
        sala3 = "livre";
    }

    public int solicita() {
        int sala = 0;
        while (sala == 0) {
            if (sala1.equals("livre")) {
                simularDemora();
                ocuparSala(1);
                sala = 1;
            } else if (sala2.equals("livre")) {
                simularDemora();
                ocuparSala(2);
                sala = 2;
            } else if (sala3.equals("livre")) {
                simularDemora();
                ocuparSala(3);
                sala = 3;
            }
        }
        return sala;
    }

    public void libera(int sala) {
        if (sala == 1) {
            sala1 = "livre";
        } else if (sala == 2) {
            sala2 = "livre";
        } else if (sala == 3) {
            sala3 = "livre";
        }
    }

    private synchronized void ocuparSala(int sala) {
        if (sala == 1) {
            if (sala1.equals("ocupada")) {
                throw new RuntimeException("Erro de competição! Duas threads
   tentando ocupar a sala 1 ao mesmo tempo!");
            } else {
                sala1 = "ocupada";
            }
        }
        if (sala == 2) {
            if (sala2.equals("ocupada")) {
                throw new RuntimeException("Erro de competição! Duas threads
   tentando ocupar a sala 2 ao mesmo tempo!");
            } else {
                sala2 = "ocupada";
            }
        }
        if (sala == 3) {
            if (sala3.equals("ocupada")) {
```

```
                throw new RuntimeException("Erro de competição! Duas threads
   tentando ocupar a sala 3 ao mesmo tempo!");
            } else {
                sala3 = "ocupada";
            }
        }
    }

    private void simularDemora() {
        try {
            Thread.sleep(500);
        } catch (InterruptedException ex) {
        }
    }
}

// Classe de teste

import java.util.Random;

public class Main3 {

    class TestaReservas extends Thread {

        int num;

        public TestaReservas(int num) {
            this.num = num;
        }

        public void run() {
            while (true) {
                try {
                    int sala = reserva.solicita();
                    System.out.println(num + ": Reservada sala " + sala);
                    try {
                        Thread.sleep(random.nextInt(1000));
                    } catch (InterruptedException ex) {
                    }
                    reserva.libera(sala);
                    System.out.println(num + ": Liberada sala " + sala);
                } catch (RuntimeException re) {
                    System.out.println(num + ": " + re.getMessage());
                }
            }
        }
    }
    Reservas reserva;
    Random random;

    public Main3() {
        reserva = new Reservas();
        random = new Random();
    }

    void executa() {
        for (int i = 0; i < 10; i++) {
            TestaReservas tr = new TestaReservas(i);
            tr.start();
        }
    }

    public static void main(String args[]) {
        new Main3().executa();
    }
}
```

8. Modifique a classe acima para tornar os métodos de “Reservas” sincronizados para cooperar e para competir.

a. **Item bônus – não cai na prova, é apenas para seu aprendizado:** execute seus casos de teste e verifique se os problemas foram solucionados.

```
class Reservas {

    private String sala1, sala2, sala3;
    private int salasLivres;

    public Reservas() {
        sala1 = "livre";
        sala2 = "livre";
        sala3 = "livre";
        salasLivres = 3;
    }

    public synchronized int solicita() {
        int sala = 0;
        while (salasLivres == 0) {
            try {
                wait();
            } catch (InterruptedException ex) {
            }
        }
        if (sala1.equals("livre")) {
            simularDemora();
            ocuparSala(1);
            sala = 1;
        } else if (sala2.equals("livre")) {
            simularDemora();
            ocuparSala(2);
            sala = 2;
        } else if (sala3.equals("livre")) {
            simularDemora();
            ocuparSala(3);
            sala = 3;
        }
        salasLivres --;
        return sala;
    }

    public synchronized void libera(int sala) {
        if (sala == 1) {
            sala1 = "livre";
        } else if (sala == 2) {
            sala2 = "livre";
        } else if (sala == 3) {
            sala3 = "livre";
        }
        salasLivres ++;
        notifyAll();
    }

    private synchronized void ocuparSala(int sala) {
        if (sala == 1) {
            if (sala1.equals("ocupada")) {
                throw new RuntimeException("Erro de competição! Duas threads
tentando ocupar a sala 1 ao mesmo tempo!");
            } else {
                sala1 = "ocupada";
            }
        }
        if (sala == 2) {
            if (sala2.equals("ocupada")) {
                throw new RuntimeException("Erro de competição! Duas threads
tentando ocupar a sala 2 ao mesmo tempo!");
            } else {
                sala2 = "ocupada";
            }
        }
        if (sala == 3) {
            if (sala3.equals("ocupada")) {
```

```
                throw new RuntimeException("Erro de competição! Duas threads
    tentando ocupar a sala 3 ao mesmo tempo!");
            } else {
                sala3 = "ocupada";
            }
        }
    }

    private void simularDemora() {
        try {
            Thread.sleep(500);
        } catch (InterruptedException ex) {
        }
    }
}
```

9. Considere o código Java abaixo:

```
class A {
    void proc1(B b) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entrou em A.proc1()");
        try { Thread.sleep(1000); } catch (InterruptedException e) { }
        System.out.println(name + " tentando chamar B.proc4()");
        b.proc4();
    }
    void proc2() { System.out.println(" dentro de A.proc2()"); }
}
class B {
    void proc3(A a) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entrou em B.proc3()");
        try { Thread.sleep(1000); } catch (InterruptedException e) { }
        System.out.println(name + " tentando chamar A.proc2()");
        a.proc2();
    }
    void proc4() { System.out.println(" dentro de B.proc4()"); }
}
public class Main3 {
    A a = new A(); B b = new B();
    void executar() {
        Thread t1 = new Thread() {
            public void run() {
                System.out.println(getName() + " começou!");
                a.proc1(b);
                System.out.println(getName() + " terminou!");
            }};
        t1.setName("t1");
        Thread t2 = new Thread() {
            public void run() {
                System.out.println(getName() + " começou!");
                b.proc3(a);
                System.out.println(getName() + " terminou!");
            }};
        t2.setName("t2");
        t1.start(); t2.start();
    }

    public static void main(String args[]) {
        new Main3().executar();
    }
}
```

a. Descreva o que o código acima faz. Qual uma possível saída de sua execução?

São criadas duas Threads, t1 e t2. T1 executa uma chamada para proc1 da classe A, que por sua vez chama proc4 da classe B. T2 executa uma chamada para proc3 da classe B, que por sua vez chama proc 2 da classe A.

Uma possível saída seria:

```
t1 começou!
t1 entrou em A.proc1()
t2 começou!
t2 entrou em B.proc3()
t1 tentando chamar B.proc4()
 dentro de B.proc4()
t1 terminou!
t2 tentando chamar A.proc2()
 dentro de A.proc2()
t2 terminou!
```

b. O que acontece se os métodos proc1, proc2, proc3 e proc4 forem sincronizados (adicionando o modificador synchronized)?

Ocorre uma situação de deadlock, pois: t1 detém a chave "a" (da classe A), pois entrou em proc1. t2 detém a chave "b" (da classe B), pois entrou em proc3. Porém, t1 tenta executar proc4, e para isso precisa obter a chave "b", que está com t2. Enquanto isso t2 tenta executar proc2, e para isso precisa obter a chave "a", que está com t1. Ambas ficarão aguardando eternamente.

Uma possível saída seria:

```
t1 começou!
t1 entrou em A.proc1()
t2 começou!
t2 entrou em B.proc3()
t2 tentando chamar A.proc2()
t1 tentando chamar B.proc4()
```