Paradigmas de Linguagens de Programação
Lista 5 – Paradigma Concorrente
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
   ii) Bhaskara1 e Bhaskara3
   iii) Bhaskara1 e Bhaskara4
   iv) Bhaskara2 e Bhaskara3
   v) Bhaskara2 e Bhaskara4
   vi) Bhaskara3 e Bhaskara4

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

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.

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.

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.

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!

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.

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

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?
b. O que pode acontecer se removermos todas as linhas indicadas com `/* a */` e `/* c */` (e somente elas), do programa acima?
c. O que pode acontecer se removermos todas as linhas indicadas com `/* c */` (e somente elas), do programa acima?
d. O que pode acontecer se trocarmos os comandos “while” das linhas indicadas com `/* a */` por comandos “if” com as mesmas condições?

(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.

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.

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