→ Sincronização de Threads

A sincronização de Threads em Java é baseada no conceito do Monitor (de Hoare). Cada objeto Java tem associado um monitor (ou "lock") que pode ser ativado se a palavra chave **synchronized** for usada na definição de pelo menos um método de instância:

public **synchronized** void metodoX();

O monitor da classe será ativado se um método de classe for declarado como **synchronized:**

public static **synchronized** void metodoY();

- Se várias Threads invocarem um método declarado como **synchronized** em simultâneo o monitor associado ao objeto garantirá a execução desse método em exclusão mútua.

1 - Suponha dois processos **p1** e **p2** que partilham uma variável comum, **variavelPart**. Pretende-se construir um exemplo que ilustre a violação de uma secção crítica, sem usar qualquer tipo de mecanismo de sincronização,
- Considere que o processo **p1** possui duas variáveis locais, x e y, inicializadas com valores simétricos, e que dentro de um ciclo infinito transfere a quantidade armazenada em **variavelPart** de x para y. O processo **p2** vai, em cada iteração, incrementar a variável partilhada.

Pretende-se que no processo **p1** a condição x + y = 0 seja verdadeira durante toda a execução do programa. Quando, no processo p1, se deteta que a secção crítica foi violada (porque x + y **!=** 0) o processo deve terminar e acabar o programa.

a) Supondo a estrutura que se segue para os processos p1 e p2, comece por criar duas classes que permitam criar os processos (Threads) p1 e p2. Para construir o objeto partilhado, pode usar uma classe com um atributo do tipo int, ou simplesmente usar um array de inteiros com um elemento.

Processo 1

```
x = M; y = - M;
While (true){
  //secção crítica  1
  x = x – variavelPart;
  y = y + variavelPart;
  <parte restante 1>
  if (x+y != 0 ){
    print "Secção crítica violada"
          break;
  }//fim do if
  …
}// fim do While
```

Processo 2

```
...
While (true){
  //secção crítica  2
  variavelPart =
        variavelPart +1;
  <parte restante 2>
}
...
```

b) Construa uma classe de teste que, instanciando os processos p1 e p2, permita simular a violação da secção crítica.

c) Para que o processo p2 termine, após a violação da secção crítica, transforme-o numa Thread daemon.

d) Modifique o programa de maneira a garantir a execução de cada secção crítica em exclusão mútua. (ver T03b, página 30 e seguintes …)

d1) Usando a instrução synchronized.

d2) Usando a classe Semáforo.

2 – Suponha que uma **sala de cinema** pretendia um pequeno programa que lhe permitisse gerir a venda de bilhetes em diferentes postos de venda. A sala tem uma lotação fixa, e a cada bilhete vendido é atribuído um número sequencial, por ordem de aquisição. Pretende-se poder:

- consultar o nome do filme em exibição;
- consultar o número de bilhetes disponíveis para venda;
- vender um bilhete (indicando ao utilizador o número correspondente ao bilhete em venda).

a) Construa uma classe, SalaCinema, que lhe permita realizar as operações descritas.

b) Para testar o programa começou-se por simular a sua execução concorrente, sendo cada posto de venda uma Thread que acede à classe anterior.

Supondo que o código do método run da thread que implementa o posto de venda é o seguinte:

```
public void run(){
      int pausa;
      while (true){
         try {
            pausa = (int)(Math.random()* 2000);
            sleep(pausa);
            System.out.println( posto + " vendeu o bilhete " + SC.venderBilhete() +
                                                  " para o filme " + SC.filme());
            if (SC.disponiveis() == 0) {
               System.out.println(posto + " fim");break;
            }
         } catch (InterruptedException ex) {
            System.out.println(ex.getMessage());
         }
      }
   }
```

Onde: **SC** é o objeto SalaCinema partilhado pelos vários postos de venda;
e **posto** é uma String com o nome do posto de venda.

- Complete e implemente a classe que simula o posto de venda, PostoVenda.

c) Construa uma classe de teste onde é criada a sala de cinema, e os 3 postos de venda. Execute o seu código e veja se se comporta corretamente.

d) Na classe PostoVenda, substitua a linha " pausa = (int)(Math.random()* 2000);" por "pausa = 2;" Nota alguma diferença no output?

Para explorar:

3 - Implemente e explore o exemplo do "Readers-Writers Problem" apresentado em T03b (página 21).

4 - O exemplo abaixo ilustra a sincronização de um método de classe e de um método de objeto ou instância.

Exercício - Depois de o estudar, implemente-o e analise os resultados.

```
public class CriticaUm {
  public synchronized void method_A() {
    System.out.println ( Thread.currentThread().getName() + " Método A");
    try {
      Thread.sleep( (int) Math.round(Math.random()*5000));
    }
    catch (InterruptedException e)
    { … }
    System.out.println ( Thread.currentThread().getName() + " Saindo do Método A");
   }
```

```
   public   static   synchronized void method_B() {
        System.out.println ( Thread.currentThread().getName() + " Método B");
        try {
         Thread.sleep( (int) Math.round(Math.random() *5000));
        }
        catch (InterruptedException e)
       { … }
       System.out.println ( Thread.currentThread().getName() + " Saindo do Método B");
    }
 }

public class Monitors extends Thread{
  CriticaUm C;
   public Monitors(String nomeObjecto) {
    Thread Thread_a, Thread_b;
    C= new CriticaUm();
    Thread_a = new Thread (this, nomeObjecto + ":Thread a");
    Thread_b = new Thread (this, nomeObjecto + ":Thread b");
    Thread_a.start();
    Thread_b.start();
  }
  public void run (){
    C.method_A();
    C.method_B();
  }
}

public class Teste {
   public static void main (String args[]){
   Monitors M1, M2;
   M1=new Monitors ("Objecto 1");
   M2=new Monitors ("Objecto 2");
   }}
```

→ Invocação recursiva de métodos sincronizados (código reentrante)

Uma Thread que adquiriu um lock num objeto (ao executar um método sincronizado) pode invocar recursivamente o mesmo método. O lock é readquirido pela Thread.

5 - Estude o exemplo abaixo.

```
public class Reentrant {
    static int times =1;
  public synchronized void myMethod() {
     int i = times++;
     System.out.println("myMethod has started " + i + " time(s)" );
     while (times <4)
        myMethod();
     System.out.println("myMethod has exited " + i + " time(s)");
   }
}

public class Teste extends Thread {
 Reentrant R;
  public Teste (){
    R= new Reentrant();
    Thread T = new Thread(this);
    T.start();
  }
  public void run(){
     R.myMethod();
  }
  public static void main(String args[]) {
      Teste T1 = new Teste();
  }}
```