Aula 28 - Threads em Java

Programas e Processos

Um programa é um conceito estático, isto é, um programa é um arquivo em disco que contém um código executável por uma CPU. Quando este programa é executado dizemos que ele é um processo. Portanto um processo é um programa em execução, ou um conceito dinâmico. Note que um programa em execução aloca recursos, como memória, disco, impressora, isto é tudo o que precisa para ser executado. Podemos até considerar a CPU como um recurso alocado por um processo, uma vez que podem haver vários processos em execução ao mesmo tempo e só um deles está com o recurso CPU num determinado instante.
Um mesmo programa pode ser executado várias vezes simultaneamente. Assim, podemos Ter um só programa e vários processos (deste programa) em execução simultaneamente. Uma outra forma de dizer é que temos várias linhas de execução deste programa.
O sistema operacional controla a execução dos vários processos:

a) dando uma fatia de tempo para cada um de acordo com algum esquema de prioridade.
b) garantindo o sincronismo entre os processos quando os mesmo precisam trocar informações.

Threads em Java

Em Java é possível lançar várias linhas de execução do mesmo programa. Chamamos a isso de Threads ou MultiThreading. A diferença com os processos e programas acima é que o Java é interpretado. Quem cuida dos vários Threads de um programa é o próprio interpretador Java. Algumas vantagens em relação aos processos:

a) O chaveamento entre os threads é mais rápido que o chaveamento entre processos
b) A troca de mensagens entre os threads também é mais eficiente.

Claro que essa maior eficiência ocorre porque o interpretador tem o controle maior sobre os threads. No entanto existe a ineficiência do interpretador que é grande.

Vejamos alguns exemplos do livro:
“Aprendendo Java 2”
Mello, Chiara e Villela
Novatec Editora Ltda. – www.novateceditora.com.br

Vejamos primeiramente um exemplo de execução seqüencial:

```
// ThreadTest1.java
public class ThreadTest1 extends Object
{
 public static void main(String args[])
  {
   new ThreadTest1();
  }
 ThreadTest1()
  {
   Contador cc = new Contador();
   for (int i=0; i<5; i++)
```

```
     {
      System.out.println("(ThreadTest1) Contador = " +
cc.proximoValor());
     }
   ClasseNormal CN = new ClasseNormal();
   CN.func(cc);
   for (int i=0; i<5; i++)
     {
      System.out.println("(ThreadTest1) Contador = " +
cc.proximoValor());
     }
  }
}

class Contador
{
 int ct = 0;
 public Contador() { }
 int proximoValor()
  {
   ct++;
   try
     {
      Thread.sleep(500);
     }
   catch (Exception e) { }
   return ct;
  }
}

class ClasseNormal
{
 void func(Contador cx)
  {
   for (int i=0; i<5; i++)
     {
      System.out.println("(ClasseNormal) Contador = " +
cx.proximoValor());
     }
  }
}
```

Veja a saida:

```
(ThreadTest1) Contador = 1
(ThreadTest1) Contador = 2
(ThreadTest1) Contador = 3
(ThreadTest1) Contador = 4
(ThreadTest1) Contador = 5
(ClasseNormal) Contador = 6
(ClasseNormal) Contador = 7
(ClasseNormal) Contador = 8
(ClasseNormal) Contador = 9
(ClasseNormal) Contador = 10
```

```
(ThreadTest1) Contador = 11
(ThreadTest1) Contador = 12
(ThreadTest1) Contador = 13
(ThreadTest1) Contador = 14
(ThreadTest1) Contador = 15
```

Vejamos agora um exemplo contendo duas linhas de execução:

```
// ThreadTest2.java
import java.lang.Thread;
public class ThreadTest2 extends Object
{
 public static void main(String args[])
  {
   new ThreadTest2();
  }
 ThreadTest2()
  {
   Contador cc = new Contador();
   for (int i=0; i<5; i++)
     {
      System.out.println("(ThreadTest2) Contador = " +
cc.proximoValor());
     }
   ThreadClass1 objetoThread = new ThreadClass1(cc);
   Thread thread = new Thread(objetoThread);
   thread.start();
   for (int i=0; i<5; i++)
     {
      System.out.println("(ThreadTest2) Contador = " +
cc.proximoValor());
     }
   }
}

class Contador
{
 int ct = 0;
 public Contador() {}
 int proximoValor()
  {
   ct++;
   try
    {
     Thread.sleep(500);
    }
   catch (Exception e) { }
   return ct;
  }
}

class ThreadClass1 implements Runnable
{
 Contador cx = null;
 ThreadClass1(Contador cz)
```

```
   {
    cx = cz;
   }
 public void run()
   {
    func();
   }
 void func()
   {
    for (int i=0; i<5; i++)
      {
       System.out.println("(ThreadClass1) Contador = " +
cx.proximoValor());
      }
   }
}
```

A saida do exemplo acima é:

```
(ThreadTest2) Contador = 1
(ThreadTest2) Contador = 2
(ThreadTest2) Contador = 3
(ThreadTest2) Contador = 4
(ThreadTest2) Contador = 5
(ThreadTest2) Contador = 7
(ThreadClass1) Contador = 8
(ThreadClass1) Contador = 9
(ThreadTest2) Contador = 10
(ThreadClass1) Contador = 11
(ThreadTest2) Contador = 12
(ThreadClass1) Contador = 13
(ThreadTest2) Contador = 14
(ThreadClass1) Contador = 15
(ThreadTest2) Contador = 15
```

Note que o 6 não é impresso e o 15 é impresso 2 vezes. Isso ocorre porque há 2 linhas de execução compartilhando as mesmas variáveis.

A interface Runnable

O exemplo acima mostra a primeira forma de lançar um thread, implementando a interface Runnable. Toda classe que implementa a interface Runnable deve especificar um método cuja
assinatura é public void run(), executado no momento em que a linha de execução é inicializada.

Os comandos abaixo cria uma nova linha de execução:

```
ThreadClass1 objetoThread = new ThreadClass1(contador);
Thread thread = new Thread(objetoThread);
thread.start();
```

Estendendo a classe Thread

A segunda forma de utilizar threads em Java é estender a própria classe
Thread, presente no pacote java.lang.Thread.
Deve-se implementar um método run que é chamado sempre que uma linha de execução é criada para o objeto..

O exemplo abaixo usa esta forma:

```
// ThreadTest3.java
import java.lang.Thread;

public class ThreadTest3 extends Object
{
 public static void main(String args[])
  {
   new ThreadTest3();
  }
 ThreadTest3() {
    Contador cc = new Contador();
    for (int i=0; i<5; i++)
      {
       System.out.println("(ThreadTest3) Contador = " +
cc.proximoValor());
      }
    ThreadClass2 objetoThread = new ThreadClass2(cc);
    objetoThread.start();
    for (int i=0; i<5; i++)
      {
       System.out.println("(ThreadTest3) Contador = " +
cc.proximoValor());
      }
    }
}

class Contador
{
 int ct = 0;
 public Contador() { }
 int proximoValor()
   {
    ct++;
    try
      {
       Thread.sleep(500);
      } catch (Exception e) { }
    return ct;
   }
}

class ThreadClass2 extends Thread
{
 Contador cx = null;
 ThreadClass2(Contador cz)
  {
```

```
   cx = cz;
  }

 public void run()
  {
   func();
  }
 void func()
  {
   for (int i=0; i<5; i++)
     {
      System.out.println("(ThreadClass2) Contador = " +
cx.proximoValor());
     }
  }
}
```

Veja a saída:

```
(ThreadTest3) Contador = 1
(ThreadTest3) Contador = 2
(ThreadTest3) Contador = 3
(ThreadTest3) Contador = 4
(ThreadTest3) Contador = 5
(ThreadTest3) Contador = 7
(ThreadClass2) Contador = 8
(ThreadTest3) Contador = 9
(ThreadClass2) Contador = 10
(ThreadTest3) Contador = 11
(ThreadClass2) Contador = 12
(ThreadTest3) Contador = 13
(ThreadClass2) Contador = 14
(ThreadTest3) Contador = 15
(ThreadClass2) Contador = 15
```

Ainda está pulando o 6 e colocando o 15 duas vezes. Conforme dito anteriormente, isso ocorre porque o método próximo Valor() é compartilhado sem proteção pelas 2 linhas de execução. A maneira de resolver isso é protegê-lo, colocando-se:

```
synchronized int proximoValor()
```

E a saída ficaria:

```
(ThreadTest3) Contador = 1
(ThreadTest3) Contador = 2
(ThreadTest3) Contador = 3
(ThreadTest3) Contador = 4
(ThreadTest3) Contador = 5
(ThreadTest3) Contador = 6
(ThreadClass2) Contador = 7
```

```
(ThreadTest3) Contador = 8
(ThreadClass2) Contador = 9
(ThreadTest3) Contador = 10
(ThreadClass2) Contador = 11
(ThreadTest3) Contador = 12
(ThreadClass2) Contador = 13
(ThreadTest3) Contador = 14
(ThreadClass2) Contador = 15
```