Artigo

Utilizando Threads - parte 1

Veja neste artigo de Paulo César M. Jeveaux: Utilizando Threads - parte 1

computadores com apenas um processador que executam várias tarefas simultâneas. Ou seja, vários processos que compartilham do uso da CPU tomando certas fatias de tempo para execução. A esta capacidade é denominado o termo multiprocessamento. Teoricamente existe uma grande proteção para que um processo não afete a execução de outro, modificando, por exemplo, a área de dados do outro processo, a menos que haja um mecanismo de comunicação entre os processos (IPC – Inter Process Communication). Este alto grau de isolamento reduz os desagradáveis GPFs (General Protection Fault), pois o sistema se torna mais robusto.

Em contrapartida, o início de cada processo é bastante custoso, em termos de uso de memória e desempenho, e o mecanismo de troca de mensagens entre os processos é mais complexo e mais lento, se comparado a um único programa acessando a própria base de dados. Uma solução encontrada foi o uso de threads, (também conhecidas por linhas de execução). A thread pode ser vista como um subprocesso de um processo, que permite compartilhar a sua área de dados com o programa ou outras threads.

O início de execução de uma thread é muito mais rápido do que um processo, e o acesso a sua área de dados funciona como um único programa. Existem basicamente duas abordagens para a implementação das threads na JVM: utilização de mecanismos nativos de operação do S.O., e a implementação completa da operação thread na JVM. A diferença básica é que as threads com mecanismos nativos do S.O. são mais rápidas. Em contrapartida a implementada pela JVM tem independência completa de plataforma. Basicamente, em ambos os casos, a operação das mesmas é obtida através de uma fatia de tempo fornecida pelo S.O. ou pela JVM. Isto cria um paralelismo virtual, como pode ser observado

Estado de uma thread

A execução de uma thread pode passar por quatro estados: novo, executável, bloqueado e encerrado.

A thread está no estado de novo, quando é criada. Ou seja, quando é alocada área de memória para ela através do operador new.Ao ser criada, a thread passa a ser registrada dentro da JVM, para que a mesma posso ser executada.

A thread está no estado de executável, quando for ativada. O processo de ativação é originado pelo método start(). É importante frisar que uma thread executável não está necessariamente sendo executada, pois quem determina o tempo de sua execução é a JVM ou o S.O.

A thread está no estado de bloqueado, quando for desativada. Para desativar uma thread é necessário que ocorra uma das quatro operações a seguir:

1. Foi chamado o método `sleep` (long tempo) da thread;
2. Foi chamado o método `suspend()` da thread (método deprecado)
3. A trhead chamou o método `wait()`;
4. A thread chamou uma operação de I/O que bloqueia a CPU;

Para a thread sair do estado de bloqueado e voltar para o estado de executável, uma das seguintes operações deve ocorrer, em oposição as ações acima:

- Retornar com o método `notify()` (ou `notifyAll()`), caso a thread estiver em espera;
- Retornar após a conclusão da operação de I/O.

A thread está no estado de encerrado, quando encerrar a sua execução. Isto pode acorrer pelo término do método `run()`, ou pela chamada explícita do método `stop()`.

Começando a trabalhar com threads

Para entender o uso de uma thread, está apresentado a seguir, um programa que fica indefinidamente imprimindo um contador na saída padrão (`SemThread.java`).

```
1 public class SemThread {
2   public static void main(String[] args) {
3   int i = 0;
4   while(true)
5   System.out.println("Número: "+ i++);
6   }
7  }
```

Aparentemente este programa ocupa completamente a CPU, e é o que realmente ocorre em S.O.s corporativos (ex.:Windows 3x). Porém em S.O. preemptivos (ex.: Windows NT, Solaris, Windows 98, OS/2, etc), o próprio S.O. se encarrega de gerenciar a ocupação da CPU, o que permite rodar outros processos, mesmo que um processo não retorne o controle para o S.O.. Como implementar o programa `SemThread`, permitindo que outros processos compartilhem a CPU? A solução é

1. Implementar a interface Runnable;
2. Ser derivada da classe Thread;

Neste exemplo a classe `Escrita` é derivada da classe `Thread`. No método `run()` da classe Escrita está contido o código necessário para implementar adequadamente o programa acima.

```
1  class Escrita extends Thread {
2   private int i;
3   public void run() {
4   while(true)
5   System.out.println("Número :"+ i++);
6   }
7  }
8
9
10
11 public class SimplesThread1 {
12  public static void main(String[] args) {
13  Escrita e = new Escrita(); //Cria o contexto de execução
14  e.start(); //Ativa a thread
15  }
16 }
```

No exemplo `SimplesThread2` a classe `Escrita` implementa a interface `Runnable`. Qualquer classe que implementar a interface `Runnable` deve ter a descrição do método `run()`.

```
 }
}



public class SimplesThread2 {
 public static void main(String[] args) {
 Escrita e = new Escrita(); //Cria o contexto de execução
 Thread t = new Thread(e); //Cria a linha de execução
 t.start(); //Ativa a thread
 }
}

```

A classe `SimplesThread2` cria o contexto de execução da thread no momento que cria uma instância de um objeto `Runnable`, que no caso é o objeto Escrita.

```
Escrita e = new Escrita(); //Poderia ser Runnable e = new Escr
```

Para criar uma linha de execução, basta criar a thread, fornecendo o contexto (o local onde há o método `run` da thread).

```
Thread t = new Thread(e);
```

O início da thread propriamente dito ocorrerá com o método `start()`.

A classe `Thread` dispõe de vários métodos. Abaixo segue uma descrição resumida de alguns destes:

1. `Thread(...)` – construtor da classe. Permite que seja instanciado um objeto do tipo Thread;
2. `void run()` – Deve conter o código que se deseja executar, quando a thread estiver ativa;
3. `void start()` – Inicia a thread. Ou seja, efetiva a chamada do método `run()`;
4. `void stop()` – encerra a thread;
5. `static void sleep` (long tempo) – deixa thread corrente inativa por no mínimo tempo milisegundos e promove outra thread. Note que este método é de classe e, consequentemente, uma thread não pode fazer outra thread dormir por um tempo;
6. `static void yield()` – Deixa a thread em execução temporariamente inativa e, quando possível, promove outra thread de mesma prioridade ou maior;
7. `void suspend()` – Coloca a thread no final da fila de sua prioridade e a deixa inativa (método deprecado);
8. `void resume()` – Habilita novamente a execução da thread. Este método deve ser executado por outra thread, já que a thread suspensa não está sendo executada (método deprecado);
9. `void interrupt()` – envia o pedido de interrupção de execução de uma thread;
10. `static boolena interrupted()` – Verifica se a thread atual está interrompida;
11. `void join()` – Aguarda outra thread para encerrar;
12. `boolean isAlive()` – retorna true caso uma thread estiver no estado executável ou bloqueado. Nos demais retorna false;
13. `void setPriority` (`int` prioridade) – Define a prioridade de execução de

para que as mesmas possam compartilhar a mesma base de dados sem causar conflitos;

16. `void wait()` – Interrompe a thread corrente e coloca a mesma na fila de espera (do objeto compartilhado) e aguarda que a mesma seja notificada. Este método somente pode ser chamado dentro de um método de sincronizado;
17. `void notify()` – Notifica a próxima thread, aguardando na fila;
18. `void notifyAll()` – Notifica todas as threads.

Há também, vários métodos para trabalhar com agrupamentos de threads. A documentação necessária pode ser encontrada no JDK, no pacote `Java.lang.ThreadGroup`.

Entendendo melhor o uso de threads

O que acontece com a thread quando termina o método main? Porque o Garbage Collection não elimina a thread da memória, já que não há nenhuma referência para a mesma? O que ocorre é que o programa pode não ter uma referência explícita para a thread, mas implicitamente a thread está cadastrada na JV e continuará cadastrada enquanto não for encerrada. Desta forma, mesmo após executar o último comando do main, o programa permanece sendo executado. Para encerrá-lo, todas as referências implícitas do programa devem ser eliminadas. Este mesmo princípio ocorre para os componentes de uma interface gráfica, onde por exemplo, mesmo ao final do main um frame pode ficar ativo. Para visualizar melhor o uso de threads, o arquivo `VariasThreads.java`, apresenta um incremento da classe `SimplesThread2`. A classe Escrita passou a ter uma variável de instância que identifca a thread que está sendo executada.

```
3   private static int cont = 0;
4   private int identificacao;
5   public void run() {
6   while(true)
7   System.out.println("Número (" + identificacao + "): " + i++);
8   }
9   public Escrita() {
10  cont++;
11  identificacao = cont;
12  }
13 }
14
15 public class VariasThreads {
16  public static void main(String[] args) {
17  Runnable r1 = new Escrita();
18  Runnable r2 = new Escrita();
19  New Thread(r1).start();
20  New Thread(r2).start();
21  }
22 }
```

Listagem 1. NOME

A execução de uma thread nativa depende do S.O. Embora a linguagem Java seja totalmente portável, certos cuidados tem que ser tomados para que as threads cooperam adequadamente, independente da JVM. Na verdade, o que se espera é que uma thread, após ser executada, passe a promover outras threads, mantendo a ordem de prioridade entre as mesmas. Se isto não ocorrer, alguns S.O. poderão ter as demais threads paradas durante a execução da thread " `xxxxx` ". O programa `VariasThreads2.java` refaz a classe Escrita para, após exibir a mensagem na saída padrão, a thread ficar inativa por pelo menos 500 milisegundos. O método `sleep` (long tempo) faz com que a thread adormeça por tempo milisegundos e

```
3   private static int cont = 0;
4   private int identificacao;
5   public void run() {
6   while(true)
7   System.out.println("Número (" + identificacao + "): " + i++);
8   try {
9   Thread.sleep(500);
10  }
11  catch(InterruptedException e) {}
12  }
13  public Escrita() {
14  cont++;
15  identificacao = cont;
16  }
17 }
18 public class VariasThreads {
19  public static void main(String[] args) {
20  New Thread(new Escrita()).start();
21  New Thread(new Escrita()).start();
22  }
23 }
```

Para analisar a segunda abordagem de implementação de threads, o programa `MultiThread.java` cria três threads com tempos de espera e nomes distintos. Para gerar um tempo de espera randômico foi utilizado o método `Math.random()`.

```
1  class UmaThread extends Thread {
2   private int delay;
3   public UmaThread(String identifacacao, int delay) {
4   super(identificacao);
5   this.delay = delay;
6   }
```

```
12   catch(InterruptedException e) {
13   System.out.println("Thread: " + identificacao + " foi interrc
14   }
15   System.out.prinln(">>" + identificacao + " " + delay);
16   }
17  }
18
19  public class MultiThread {
20   public static void main(String[] args) {
21   UmaThread t1,t2,t3;
22   t1 = new UmaThread("Primeira", (int)(Math.random()*8000));
23   t2 = new UmaThread("Segunda", (int)(Math.random()*8000));
24   t3 = new UmaThread("Terceira", (int)(Math.random()*8000));
25
26   t1.start();
27  t2.start();
28  t3.start();
29   }
30  }
```

Tecnologias:

Java