Introdução ao uso de Threads em Java

Daniel de Angelis Cordeiro
danielc@ime.usp.br

26 de março de 2004

Sumário

1 Introdução

Este seção fornece uma breve introdução sobre *threads* adaptada do artigo *Introduction to Java Threads* [4].

1.1 O que são *threads*?

Todos os sistemas operacionais modernos possuem o conceito de processos que, de forma simplificada, são programas diferentes e independentes executados pelo sistema operacional.

Threading é um artifício que permite a coexistência de múltiplas atividades dentro de um único processo. Java é a primeira linguagem de programação a incluir explicitamente o conceito de *Threads* na própria linguagem. *Threads* são também chamados de "processos leves" (*lightweight processes*) pois, da mesma forma que processos, *threads* são independentes, possuem sua própria pilha de execução, seu próprio *program counter* e suas próprias variáveis locais. Porém, *threads* de um mesmo processo compartilham memória, descritores de arquivos (*file handles*) e outros atributos que são específicos daquele processo.

Um processo pode conter múltiplas *threads* que parecem executar ao mesmo tempo e de forma assíncrona em relação as outras *threads*. Todas as *threads* de um mesmo processo compartilham o mesmo espaço de endereçamento de memória, o que significa que elas têm acesso as mesmas variáveis e objetos, e que eles podem alocar novos objetos de um mesmo *heap*. Ao mesmo tempo que isso torna possível que *threads* compartilhem informações entre si, é necessário que o desenvolvedor se assegure que uma *thread* não atrapalhe a execução de outra.

A API de *threads* em Java é incrivelmente simples. Mas escrever um programa complexo que as use de forma eficiente e correta não é tão simples assim (esta é uma das razões da existência de MAC 438). É responsabilidade do programador impedir que uma *thread* interfira de forma indesejável no funcionamento das outras *threads* do mesmo processo.

1.2 Todo programa em Java é *multithreaded*

Acredite, se você escreveu algum programa em Java então já fez um programa *multithreaded*.

Todo programa em Java possui pelo menos uma *thread*: a *thread* `main`. Além dessa, a máquina virtual mantém algumas outras que realizam tarefas como coleta de lixo ou finalização de objetos. Algumas classes disponíveis na API de Java também utilizam *threads* em suas implementações. Como exemplo, podemos citar as classes de Java Swing ou as classes da implementação de RMI.

1.3 Por que utilizar *threads*?

Há muitos motivos para se usar *threads*. Entre eles, podemos citar:

- Responsividade em Interfaces Gráficas: imagine se o seu navegador web parasse de carregar uma página só porque você clicou no menu "arquivo";
- Sistemas multiprocessados: o uso de *threads* permite que o SO consiga dividir as tarefas entre todos os processadores disponíveis aumentando, assim, a eficiência do processo;
- Simplificação na Modelagem de Aplicações: suponha que você precise fazer um programa que simule a interação entre diferentes entidades. Carros em uma estrada, por exemplo. É mais fácil fazer um *loop* que atualiza todos os carros da simulação ou criar um objeto *carro* que anda sempre que tiver espaço a frente dele?

- Processamento assíncrono ou em segundo plano: com *threads* um servidor de e-mail pode, por exemplo, atender a mais de um usuário simultaneamente.

2 Utilizando *threads*

2.1 Criando *threads*

Há duas formas equivalentes de criarmos uma thread em Java. Ou criamos um objeto que estende a classe `Thread` e sobrescrevemos o seu método `public void run()` ou implementamos a interface `Runnable` que é definida como:

```
package java.lang;

public interface Runnable {
    public abstract void run();
}
```

O exemplo abaixo ilustra a utilização de *threads* utilizando as duas formas.

```
public class Exemplo extends Thread {

    // classe interna que estende thread
    // (a classe é definida como static apenas para podermos
    //  alocar uma instância no main)
    static class Tarefa1 extends Thread {
        // método que sobrescreve o método run() herdado
        public void run() {
            for(int i=0; i<1000; i++) {
                System.out.println("Usando Herança");
            }
        }
    }

    // classe interna que implementa a interface Runnable
    static class Tarefa2 implements Runnable {
        public void run() {
            for(int i=0; i<1000; i++) {
                System.out.println("Usando Runnable");
            }
        }
    }

    public static void main(String[] args) {
```

```
        // basta criarmos uma intância da classe que extende Thread
        Thread threadComHeranca = new Tarefa1();

        // primeiro devemos alocar uma instância de tarefa
        Tarefa2 tarefa = new Tarefa2();
        // e depois criamos a nova Thread, passando tarefa como argumento
        Thread threadComRunnable = new Thread(tarefa);

        // agora iniciamos as duas Threads
        threadComHeranca.start();
        threadComRunnable.start();
    }

}
```

A saída do texto, como esperado, são as duas mensagens intercaladas aleatoriamente na saída padrão. Dependendo do sistema operacional e dos recursos computacionais do computador a saída pode parecer seqüencial. Esse comportamento será explicado na seção 2.3.

Apesar das duas formas de uso de *threads* serem equivalentes, classes que necessitem estender outra classe que não `Thread` são obrigadas a usar a interface `Runnable`, já que Java não possui herança múltipla. Além disso, os "puristas" em orientação a objetos dizem que normalmente estamos interessados em criar classes que *usem threads* e não classes que *sejam threads* e que, portanto, deveríamos implementar toda a lógica em uma classe que implementa `Runnable` e depois criar a *thread* só quando for necessário.

2.2 Possíveis estados de uma *thread*

Uma *thread* pode estar em um dos seguintes estados:

- criada,
- em execução,
- suspensa ou
- morta.

Uma *thread* se encontra no estado "criada" logo após a instanciação de um objeto `Thread`. Neste ponto nenhum recurso foi alocado para a *thread*. A única transição válida neste estado é a transição para o estado "em execução".

A *thread* passa para o estado "em execução" quando o método `start()` do objeto é chamado. Neste ponto a *thread* pode ficar "em execução", se tornar "suspensa" ou se tornar "morta".

Na verdade, uma *thread* pode estar "em execução" mas, ainda assim, não estar ativa. Em computadores que possuem um único processador é impossível existirem duas *threads*

executando ao mesmo tempo. Dessa forma uma *thread* que está esperando para ser executada pode estar no estado "em execução" e ainda assim estar parada.

A *thread* se torna "suspensa" quando um destes eventos ocorrer:

- execução do método `sleep()`;
- a *thread* chama o método `wait()` para esperar que uma condição seja satisfeita;
- a *thread* está bloqueada em uma operação de entrada/saída (I/O).

A chamada ao comando `sleep(int seconds)` faz com que a *thread* espere um tempo determinado para executar novamente. A *thread* não é executada durante este intervalo de tempo mesmo que o processador se torne disponível novamente. Após o intervalo dado, a *thread* volta ao estado "em execução" novamente.

Se a *thread* chamar o comando `wait()` então ela deve esperar uma outra *thread* avisar que a condição foi satisfeita através dos comandos `notify()` ou `notifyAll()`. Esses métodos serão explicados na seção 3.

Se a *thread* estiver esperando uma operação de entrada/saída ela retornará ao estado "em execução" assim que a operação for concluída.

Por fim, a *thread* se torna "morta" se o método `run()` chegar ao fim de sua execução ou se uma exceção não for lançada e não for tratada por um bloco *try/catch*.

2.3 Prioridades de *threads*

Anteriormente foi dito que diversas *threads* podem executar concorrentemente em um processo. Porém em sistemas monoprocessados isso é parcialmente verdade. Conceitualmente é isso que ocorre. Mas na prática apenas uma *thread* pode ocupar o processador por vez. A máquina virtual de Java define uma ordem para que as *threads* ocupem o processador de forma que o usuário tem a ilusão de que elas realmente executam concorrentemente. A isso é dado o nome de *escalonamento de tarefas*.

As prioridades de *threads* em Java podem variar entre os inteiros `Thread.MIN_PRIORITY` e `Thread.MAX_PRIORITY`. Quanto maior o inteiro, maior a prioridade. Para definir a prioridade de uma *thread* é necessário chamar o método `setPriority(int priority)` da classe `Thread` após a instanciação do objeto. A máquina virtual utilizará sua prioridade relativa para realizar o escalonamento.

A máquina virtual sempre escolhe a *thread* com maior prioridade relativa e que esteja no estado de "execução" para ser executada. Uma *thread* de menor prioridade só será executada se a *thread* atual for para o estado "suspenso", se o método `run()` terminar ou se o método `yield()`, cuja finalidade é dar chance para que outras *threads* executem, for chamado. Se existirem duas *threads* de mesma prioridade esperando para serem executadas, a máquina virtual escolherá uma delas, como em um escalonamento do tipo *round-robin*.

A *thread* escolhida continuará ocupando o processador até que uma das seguintes condições se tornem verdadeiras:

- uma *thread* de maior prioridade fica pronta para ser executada;

- a *thread* atual chama `yield()` ou o método `run()` termina;
- o tempo reservado para a *thread* termina (só em SOs onde os processos possuem um tempo máximo para permanecer no processador para serem executados. Discutiremos isso na seção 2.3.1).

Como regra geral, podemos pensar que o escalonador escolherá sempre a *thread* com maior prioridade para ser executada em determinado momento. Porém, a máquina virtual é livre para escolher uma *thread* de menor prioridade para evitar espera indefinida (*starvation*).

2.3.1 Fatias de tempo

Alguns sistemas operacionais como o **GNU/Linux** ou o Microsoft®Windows™[1] fornecem para cada *thread* "fatias de tempo" (*time-slices*), que nada mais são do que o intervalo de tempo máximo que a *thread* pode ocupar o processador antes de ser obrigada a cedê-lo a outra *thread*.

O exemplo da seção 2.1 produziria o seguinte resultado em sistemas que não possuem fatias de tempo para suas *threads*:

```
Usando Runnable
(...) ("Usando Runnable" repetido 998 vezes)
Usando Runnable
Usando Herança
(...) ("Usando Herança" repetido 998 vezes)
Usando Herança
```

Ou seja, a primeira *thread* escalonada seria executada até o fim.

Já em um ambiente com fatias de tempo, as *threads* seriam interrompidas e o resultado da execução seria mais ou menos assim:

```
Usando Runnable
Usando Runnable
Usando Herança
Usando Runnable
Usando Herança
Usando Herança
(...)
```

A especificação de Java [2] **não** garante que o ambiente terá fatias de tempo. Fatias de tempo só ocorrerão se o programa rodar em um sistema operacional que possua essa característica. Seus programas não devem partir desse pressuposto.

[1] O autor não sabe ser imparcial quando o assunto é sistema operacional. :-)

3 Sincronismo entre *threads*

A maior parte dos programas *multithreadeds* necessitam que as *threads* se comuniquem de forma a sincronizar suas ações. Para isso, a linguagem Java provê *locks* (também chamados de *monitores*).

Para impedir que múltiplas *threads* acessem um determinado recurso, cada *thread* precisa adquirir um *lock* antes de usar o recurso e liberá-lo depois. Imagine um *lock* como uma permissão para usar o recurso. Apenas aquele que possuir a permissão poderá utilizar o recurso. Os outros deverão esperar. A isso damos o nome de *exclusão mútua*.

Threads usam *locks* em variáveis compartilhadas para realizar sincronizações e comunicações entre *threads*. A *thread* que tiver o *lock* de um objeto sabe que nenhuma outra *thread* conseguirá obter o *lock* deste objeto. Mesmo se a *thread* que possuir o *lock* for interrompida, nenhuma outra *thread* poderá adquirir o *lock* até que a primeira volte a utilizar o processador, termine a tarefa e libere o *lock*. A *thread* que tentar obter um *lock* que já estiver com outra *thread* será suspensa e só voltará ao estado "em execução" quando obtiver novamente o *lock*.

Como exemplo, suponha uma classe que encapsula o serviço de impressão. Várias *threads* de seu programa tentarão imprimir, mas apenas uma *thread* por vez deve conseguir. Sua classe, então, seria mais ou menos assim:

```
import java.io.File;

class Impressora {

    synchronized public void imprimir(File arquivo) {
        (executa o código para impressão do arquivo)
    }

    public bolean estaImprimindo() {
        (devolve true se a impressora está imprimindo)
    }

}
```

Note que no exemplo acima apenas uma *thread* por vez pode executar o método imprimir. Mas várias *threads* podem verificar simultaneamente se a impressora está imprimindo algo ou não.

3.1 Utilizando *locks*

Em Java, cada objeto possui um *lock*. Uma *thread* pode adquirir um *lock* de um objeto usando a palavra-chave `synchronized`. Esta palavra-chave pode ser usada na declaração de métodos (neste caso o *lock* do objeto `this` é utilizado) ou em blocos de código (neste caso, o objeto que contém o *lock* que será usado deve ser especificado). Os seguintes usos de `synchronized` são equivalentes:

```
synchronized public void teste() {
    façaAlgo();
}

public void teste() {
    synchronized(this) {
        façaAlgo();
    }
}
```

O *lock* é liberado automaticamente assim que o bloco `synchronized` termina de ser executado.

Outra característica importante sobre *locks* em Java é que eles são reentrantes. Isso quer dizer que uma *thread* pode readquirir um *lock* que ela já possuir. Veja o exemplo abaixo:

```
public class Reentrante {

    public synchronized void entrar() {
        reentrar();
    }

    private synchronized void reentrar() {
        System.out.println("Não foi necessário adquirir outro lock");
    }

}
```

Ao executar o método `entrar()` a *thread* deve adquir o *lock* do objeto `this`. Mas graças ao fato dos *locks* serem reentrantes, não foi necessário readquirir o *lock* para executar o método `reentrar()`. O *lock* só será liberado quando o método `entrar()` terminar se ser executado.

Também é tarefa do `synchronized` garantir a visibilidade das variáveis compartilhadas. A máquina virtual de Java mantém uma memória *cache* para cada uma de suas *threads* e uma memória principal que é acessada por todas as *threads*. As atualizações de memória ocorrem primeiro no *cache* e só depois na memória principal. Para impedir que uma *thread* atualize dados que são compartilhados apenas em seu *cache* local, a palavra-chave `synchronized` obriga que os dados sejam gravados na memória principal. Dessa forma outras *threads* sempre acessam o valor mais atual dessas variáveis compartilhadas.

3.2 Variáveis voláteis

A palavra-chave `volatile` em Java serve para indicar que uma variável é volátil. Isso significa três coisas: primeiro que as operações de leitura e escrita em variáveis voláteis são atômicas, isto é, duas *threads* não podem escrever em uma variável volátil ao mesmo

tempo. Segundo, as mudanças feitas em variáveis voláteis são automaticamente visíveis para todas as *threads*. E, por último, indica para o compilador que a variável pode ser modificada por outras *threads*.

Veja o seguinte código:

```
class VolatileTeste {
   boolean flag;

   public void foo() {
      flag = false;
      if(flag) {
         (...)
      }
   }

}
```

Um compilador poderia otimizar o código removendo todo o bloco dentro do `if`, pois sabe que a condição do `if` nunca será verdadeira. Porém isso não é verdade em um ambiente *multithreaded*. Para impedir esse tipo de otimização basta declarar a variável `flag` como sendo uma `volatile boolean flag`.

Para uma explicação detalhada sobre o verdadeiro significado das palavras-chave `volatile` e `synchronized` em termos do modelo de memória do Java, leia o excerto [7] do livro "Concurrent Programming in Java", de Doug Lea.

3.3 Quase todas as classes não são sincronizadas

Como sincronismo implica em uma pequena diminuição no desempenho do código (devido a obtenção e liberação do *lock*), a maior parte das classes de uso geral, como as classes que estendem `java.util.Collection`, por exemplo, não são sincronizadas. Isso significa que classes como `HashMap` não podem ser usadas por múltiplas *threads* sem o uso de `synchronized`.

Você pode usar as classes que são `Collection` em uma aplicação *multithreaded* se você utilizar um *lock* toda vez que você acessar um método em uma coleção compartilhada entre as *threads*. Para cada coleção, você deve usar o mesmo *lock* todas as vezes. Normalmente é utilizado o *lock* do próprio objeto que representa a coleção.

A classe `Collections` provê um conjunto de *wrappers* para as interfaces `List`, `Map` e `Map`. Você pode sincronizar um `Map` com o método `Collections.synchronizedMap`. Veja a API [1] da classe `Collections` para mais informações.

Se a documentação da classe não diz nada sobre ser *thread-safety*, então você deve assumir que a sincronização é responsabilidade sua.

4 Os métodos `wait()`, `notify()` e `notifyAll()`

Códigos como o do exemplo abaixo utilizam o que chamamos de *espera ativa*. Ou seja, a *thread* fica esperando uma determinada condição ser satisfeita dentro de um laço. Essa solução é ruim porque a *thread* fica ocupando o processador apenas para verificar se a condição já foi satisfeita ou não.

```
while(true) {
  if(condiçãoEsperada)
    break;
}
```

A classe `Object` fornece os métodos `wait()`, `notify()` e `notifyAll()`. Com esses métodos Java fornece uma implementação do conceito de monitores [10] que não utiliza espera ativa. Ao invés disso, esses métodos enviam eventos para as *threads* indicando se elas devem ser suspensas ou se devem voltar ao estado "em execução".

Para utilizar quaisquer dos três métodos de um objeto qualquer a *thread* deve possuir o *lock* do mesmo objeto.

`wait()` faz a *thread* que chamou o método dormir até que ela seja interrompida pelo método `Thread.interrupt()`, até que o tempo especificado no argumento de `wait()` tenha passado ou até que outra *thread* a notifique usando o método `notify()` ou `notifyAll()`. Antes de ser suspensa, a *thread* libera o *lock* do objeto. Note, entretanto, que a *thread* continuará a manter todos os outros *locks* que ela possuir enquanto estiver suspensa.

O método `notify()` acorda, se existir, alguma[2] *thread* que esteja esperando um evento deste objeto (ou seja, que tenha executado o comando `wait()` neste objeto). E o comando `notifyAll()` acorda **todas** as *threads* que estejam esperando neste objeto. Antes que a *thread* acordada possa prosseguir, ela deve esperar a *thread* que a acordou liberar o *lock* e obtê-lo novamente.

O exemplo abaixo deve tornar esses conceitos mais claros. Ele implementa uma *caixa* onde um inteiro pode ser armazenado. Se uma *thread* tentar armazenar um inteiro em uma caixa cheia ela será suspensa até que alguma *thread* retire o inteiro da caixa. Do mesmo modo, se uma *thread* tentar retirar um inteiro de uma caixa vazia ela será suspensa até que uma outra *thread* coloque um inteiro na caixa. Esse é um exemplo do problema do *produtor/consumidor*.

```
public class Caixa {

    private int conteudo;
    private boolean cheia = false;

    public synchronized int get() {
        while (cheia == false) {
```

[2] Uma *thread* **qualquer** é acordada. Não há nenhuma garantia de ordem.

```
            try {
                // espera até que um produtor coloque um valor.
                wait();
            } catch (InterruptedException e) {
            }
        }
        cheia = false;
        // informa à todos que a caixa foi esvaziada.
        notifyAll();
        return conteudo;
    }

    public synchronized void put(int valor) {
        while (cheia == true) {
            try {
                // espera até que um consumidor esvazie a caixa.
                wait();
            } catch (InterruptedException e) {}
        }
        conteudo = valor;
        cheia = true;
        // informa à todos que um novo valor foi inserido.
        notifyAll();
    }

}
```

5 Mais Informações

Informações mais detalhadas sobre programação *multithreaded* podem ser encontradas nas referências deste texto.

A API [1] da classe *Thread* de Java traz, além de seus métodos e atributos, uma pequena explicação sobre o seu funcionamento. A API da classe *Object* traz explicações sobre os métodos `wait()`, `notify()` e `notifyAll()` necessários para a manipulação de *locks*.

O capítulo 17 (*Threads and Locks*) da especificação da linguagem Java [2] possui informações detalhadíssimas sobre o funcionamento de *threads*, *locks* e sobre a atomicidade de operações em Java que, apesar de irem muito além do escopo deste texto, valem a leitura.

Recomendo também a leitura dos artigos [6] e [8], que discutem os problemas da técnica de "double check", muito utilizada em programação concorrente mas que não funciona em Java devido a especificação do modelo de memória de Java.

Referências

[1] Java™2 Plataform API Specification. http://java.sun.com/j2se/1.4.2/docs/api/.

[2] James Gosling, Bill Joy, Gilad Bracha e Guy Steele. The Java Language Specification, 2a edição. Addison-Wesley, 2000. Disponível em http://java.sun.com/docs/books/jls/.

[3] Mary Campione e Kathy Walrath. The Java Tutorial, terceira edição. Addison-Wesley, 1998. Disponível em http://java.sun.com/docs/books/tutorial/.

[4] Brian Goetz. Introduction to Java threads. IBM DeveloperWorks, setembro 2002. Disponível em http://www-106.ibm.com/developerworks/edu/j-dw-javathread-i.html.

[5] Alex Roetter. Writing multithreaded Java applications. IBM DeveloperWorks, fevereiro 2001. Disponível em http://www-106.ibm.com/developerworks/library/j-thread.html.

[6] Brian Goetz. Double-checked locking: Clever, but broken. JavaWorld, fevereiro 2001. Disponível em http://www.javaworld.com/jw-02-2001/jw-0209-double.html.

[7] Doug Lea. Synchronization and the Java Memory Model. Disponível em http://gee.cs.oswego.edu/dl/cpj/jmm.html.

[8] David Bacon et al. The "Double-Checked Locking is Broken"Declaration. Disponível em http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

[9] Neel Kumar. IBM DeveloperWorks, abril 2000. Disponível em http://www-106.ibm.com/developerworks/java/library/j-threadsafe/.

[10] Gregory Andrews. Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley, 1999.