Threads em Java

por: Antônio Theóphilo
theocjr@jspbrasil.com.br

Introdução

Com o advento da programação multi-thread, ficou difícil para qualquer programador desenvolver soluções sem precisar lançar mão dos conceitos de programação concorrente. Mesmo que você não saiba nada de threads, você com certeza está utilizando seus recursos no momento que está escrevendo uma simples aplicação com interface gráfica que imprime "Hello World" na tela, já que a Máquina Virtual Java cria uma thread só para tomar conta do gerenciamento da interface gráfica. No entanto nem sempre o programador pode ter a mão toda essa transparência e por diversas vezes é preciso "botar a mão na massa" para criar e controlar ele mesmo as threads em seu programa. É este o objetivo deste texto: mostrar como criar e controlar threads em uma aplicação Java.
Este texto visa dar uma explicação introdutória sobre o conceito de threads e como esses conceitos podem ser aplicados à linguagem Java. Este texto está organizado da seguinte forma: primeiramente será explicado o que são threads e para que elas servem; logo após é mostrado, através de dois exemplos, como podemos criar threads em Java; depois apresenta-se as formas de sincronização entre threads que acesam dados compartilhados através de mais dois exemplos; por fim é dada uma pequena conclusão e segue-se com a bibliografia recomendada e utilizada neste texto.

Threads, o que são e para que servem?

Antes de falar em threads e explicar o que são e para que servem, vamos mostrar como um programa é organizado: todo programa possui uma área de dados e uma área de texto (que é a área que contém as instruções do programa). Quando uma programa é posto para "rodar", é criado pelo sistema operacional um **processo** que nada mais é do que o programa em funcionamento, juntamente com alguns dados adicionais para controlar a sua execução (contador de programa, apontador de pilha, ...). No momento da criação do processo existe uma única **linha de execução** nele, e thread nada mais é do que isso: uma linha de execução dentro de um processo (veja figura 1.a).
Então como se dá a programação multi-thread? Ora programação multi-thread como o próprio nome diz é o fato de se ter várias threads (isto é, várias linhas de execução) dentro da mesma aplicação (processo), e como isso é possível? Todas as threads compartilham a mesma área de processo, portanto elas "rodam" sobre a mesma área de texto (a área que contém as instruções do programa). Cada thread possui um conjunto de dados que controlam a sua própria execução, exemplos de alguns deles são: o contador de programa; o apontador de pilha; o valor dos registradores, ... No momento que uma thread é posta para rodar ela verifica seus dados para saber aonde ela parou, para aonde ela deve ir e quais foram os dados que ela estava trabalhando antes de ter sido interrompida. Assim várias linhas de execução podem coexistir em uma mesma área de processo sem que uma atrapalhe a execução da outra (veja figura 1.b). Em um ambiente multi-processado (plataforma com vários processadores) as várias threads podem ser escalonadas para diferentes processadores e neste momento se tem uma aplicação verdadeiramente paralela, com as várias threads atuando ao mesmo tempo. Em um ambiente mono-processado esse paralelismo é então simulado, alguma entidade (no caso de Java essa entidade é a Máquina Virtual Java) fica responsável por escalonar o procesador para as várias threads do processo. O sistema operacional de qualquer plataforma multi-tarefa (exemplo: Windows, Linux, ...) faz esse mesmo escalonamento só que não com threads, mas com processos. No momento que uma thread tem a sua cota de tempo de execução terminada, ela salva os seus dados de controle de execução (que são poucos) e uma outra thread "carrega" os seus dados de controle para começar a executar. Mesmo com a explícita queda de rendimento em relação a ambientes multi-processados, esse cenário por muitas vezes ainda é bem melhor do que se fosse utilizada uma única thread com um único processador.

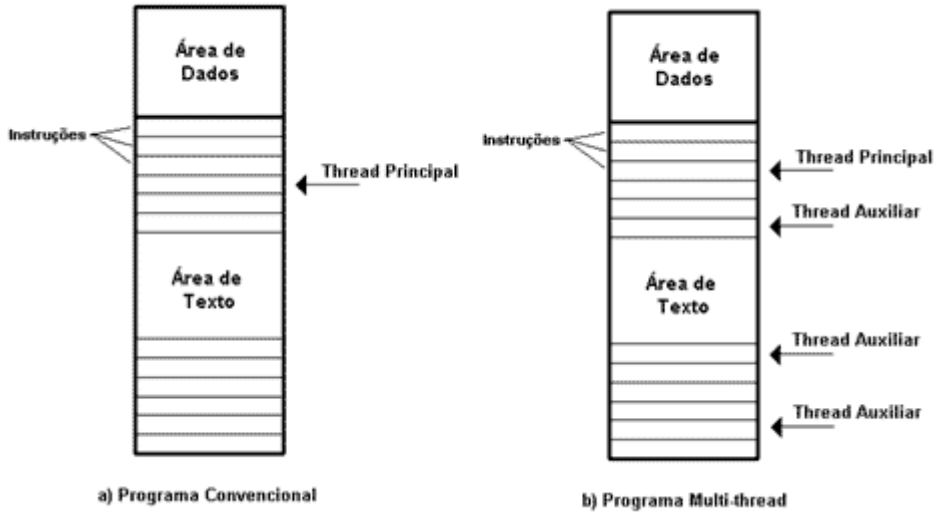

Figura 1

Imagine uma aplicação que funciona em rede e que também precisa ficar atualizando uma interface gráfica com dados que são processados localmente. Caso exista uma única thread de execução o programa pode estar a espera de uma resposta da rede que caiu e ficar congelado por muito tempo dando ao usuário uma impressão de que o programa está travado. Caso existam threads separadas para a comunicação em rede e a atualização da interface gráfica, enquanto uma thread aguarda por uma resposta da rede, a outra thread pode assumir o controle do processador, atualizar a interface gráfica com novos dados e tornar a utilização do mesmo mais eficiente além de dar uma resposta ao usuário muito mais agradável. Essa é uma das grandes vantagens da programação multi-thread, além de permitir o paralelismo em máquinas multi-processadas, ela permite em máquinas mono-processadas uma utilização mais efetiva do processador.
Esse tipo de programação possui um nome: **programação concorrente**. Por muitos anos para usar esses conceitos, o programador não dispunha das facilidades de desenvolvimento de aplicações multi-thread e por isso para desenvolver seus programas concorrentes ele precisava criar processos-filhos a partir de um processo-pai (veja a figura 2.a). Ambas as abordagens geralmente conseguem o mesmo resultado, a diferença é que na hora de se criar um novo processo-filho e sistema operacional tem um trabalho muito maior do que a criação de uma nova thread. É necessário alocar um novo espaço de endereçamento para o novo processo, além do que a troca de contexto (ou seja, o escalonamento) entre processos é muito mais custosa do entre threads. Por esse motivo a programação multi-thread (veja figura 2.b) tem ganhado um grande espaço entre os programadores de aplicações concorrentes.

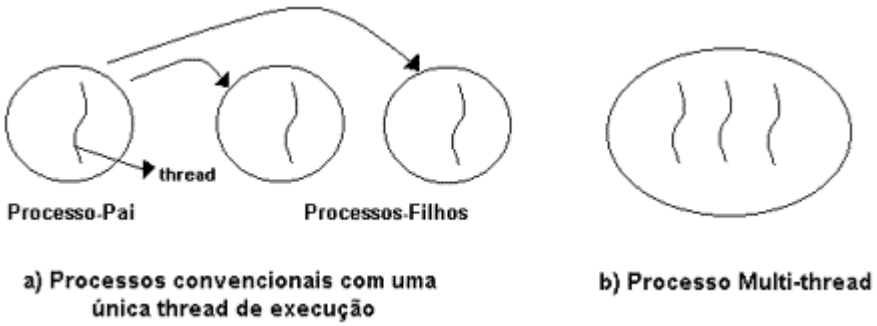

Figura 2

O último assunto a ser exposto nesta seção diz respeito aos problemas advindos de se ter várias threads (linhas de execução) dentro de uma mesma área de processo compartilhando acesso a dados. Não é difícil perceber que se não houver uma correta **sincronização** de acesso, dados podem se tornar inconsistentes devido ao aceso múltiplo e paralelo das threads. Imagine que uma thread pode estar lendo o valor de uma variável enquanto outra a está atualizando. Ou pior, que duas threads ao mesmo tempo estão tentando incrementar a mesma variável, qual será o

resultado? A variável será incremetada uma ou duas vezes? Como controlar isso? Esse assunto é totalmente endereçado pela linguagem Java e é mostrado logo após a explicação de como se criar threads em Java.

Threads em Java

Ao contrário de outras linguagens de programação que utilizam o conceito de threads através de bibliotecas específicas (como a biblioteca Pthreads para C[6] ou a biblioteca JThreads/C++[5]), Java incorpora este conceito dentro da própria linguagem e por isso não é necessário então linkar seu programa multi-thread com nenhuma outra biblioteca externa. Existem basicamente duas formas de se criar uma thread em Java: herdando da classe *java.lang.Thread* ou implementando a interface *java.lang.Runnable* (a classe *java.lang.Thread* implementa esta interface). Iremos mostrar como criar threads simples que imprimem mensagens em um buffer (usaremos um objeto da classe *java.lang.StringBuffer* para simular este buffer) das duas formas. O primeiro exemplo consiste de duas classes que são descritas a seguir:

- **ThreadHeranca**: Classe que representa uma thread criada a partir da herança da classe *java.lang.Thread*;
- **Principal**: Contem o método *main* que cria o buffer e as threads que irão acrescentar caracteres ao buffer;

Compile este código e execute a classe Principal várias vezes. Verifique como as saídas de cada execução são diferentes uma das outras devido à concorrência da aplicação. Abaixo está mostrado o código da classe ThreadHeranca e uma explicação de cada parte dele;

```
1. public class ThreadHeranca extends java.lang.Thread{
2.      private StringBuffer buf;
3.      private String texto;

4.      public ThreadHeranca(StringBuffer buf, String texto){
5.              this.buf = buf;
6.              this.texto = texto;
        }

7.      public void run(){
8.              for(int i=0; i < 20; i++){
9.                  buf.append(texto);
10.                 try {
11.                         sleep((long)(Math.random() * 100));
12.                 } catch (InterruptedException ie) {
                    }
                }
        }
   }
```

1: Uma das formas de se criar uma thread é herdando da classe *java.lang.Thread*;
2: Referência ao buffer compartilhado que irá ser alterado pela thread;
3: Referência ao objeto String que contém o texto que será inserido no buffer;
4-6: Construtor que irá inicializar as variáveis do objeto;
7: Esta é uma das partes mais importantes da criação de uma thread. Este é o método que será chamado quando a thread começar a ser executada. Caso o programador não implemente este método, uma implementação default que não faz nada é fornecida pela classe *java.lang.Thread*, mas normalmente o programador terá a necessidade que sobrecarregar este método colocando nele o código que deverá ser executado pela thread;
8-9: Esta thread irá acrescentar 20 vezes o conteúdo do objeto String *texto* no final do buffer;
10-12: A thread é colocada para dormir por um tempo arbitrário para dar a aplicação um aspecto de imprevisibilidade (e também para dar tempo à máquina virtual de escalonar as outras threads para executar já que este código é muito rápido).

Abaixo está mostrado o código da classe Principal e uma explicação de cada parte dele;

```
1. public class Principal{
2.      public static void main(String []args){
3.              StringBuffer buf = new StringBuffer();

4.              Th  dH  h 1  Th  dH  (h f
```

```
1. public class Principal{
2.      public static void main(String []args){
3.              StringBuffer buf = new StringBuffer();

4.              ThreadHeranca her1 = new ThreadHeranca(buf,
"thread 1\n");
5.              ThreadHeranca her2 = new ThreadHeranca(buf,
"thread 2\n");
6.              ThreadHeranca her3 = new ThreadHeranca(buf,
"thread 3\n");

7.              her1.start();
8.              her2.start();
9.              her3.start();

10.             for(int i=0; i < Integer.MAX_VALUE/10; i++);
11.             System.out.println(buf.toString());
        }
   }
```

1-2: Declaração da classe e do método *main*;
3: Criação do buffer que terá referências em cada thread;
4-6: Criacão das threads (elas apenas foram criadas, a execução de cada uma delas ainda não foi disparada);
7-9: Agora cada uma das threads tem a sua execução iniciada. Note que chamamos o método *start()* que não foi implementado pelo programador. A explicação para isso é que este método é implementado pela classe *java.lang.Thread* e nada mais é do que uma chamada à Máquina Virtual Java para ela criar uma thread que executará concorrentemente o método *run()* que implementamos anteriormente. O leitor pode estar se perguntando por que não chamamos diretamente o método *run()* que contém o código que queremos executar. A resposta é que se fizéssemos isso o método *main()* iria esperar o retorno do método *run()* e não haveria assim concorrência nenhuma, seria uma chamada a uma função normal como outra qualquer;
10: Este código é apenas para fazer este método "esperar" que as threads terminem a sua execução (não usamos o método *sleep()* porque estamos em um contexto estático);
11: Finalmente imprimimos o resultado do buffer na tela para comprovarmos que as três threads executaram seus respectivos códigos concorrentemente.

Bem falta agora mostrar a segunda forma de se criar uma thread em Java: através da interface *java.lang.Runnable*. Para isso basta fazermos pouquíssimas modificações no exemplo anterior. Abaixo está o código da classe ThreadInterface que substitui a classe ThreadHeranca com uma explicação das mudanças em relação à anterior:

```
1:  public class ThreadInterface implements Runnable{
```

```
    }
```

1: Esta é uma das poucas mudanças a se fazer na classe. Agora não mais herdamos da classe *java.lang.Thread* mas ao invés disso implementamos a interface *java.lang.Runnable*. Esta interface apenas requer que implementemos o método *public void run()* que conterá, assim como no exemplo anterior, o código que será executado pela thread;
2-9: Este código é idêntico ao exemplo anterior;
10: Esta linha de código tenta simular o método *sleep()* que é um método da classe *Thread* já que agora não mais herdamos desta classe.

Agora mostramos o código Principal2 com as modificações feitas:

```
1. public class Principal2{
2.      public static void main(String []args){
3.              StringBuffer buf = new StringBuffer();

4.              Thread inter1 = new Thread(new
ThreadInterface(buf, "thread 1\n"));
5.              Thread inter2 = new Thread(new
ThreadInterface(buf, "thread 2\n"));
6.              Thread inter3 = new Thread(new
Threadinterface(buf, "thread 3\n"));

7.              inter1.start();
8.              inter2.start();
9.              inter3.start();

10.             for(int i=0; i < Integer.MAX_VALUE/10; i++);
11.             System.out.println(buf.toString());
        }
   }
```

1-3: Este código é idêntico ao exemplo anterior;
4-6: Esta é a única mudança neste código. Agora nós criamos um objeto *java.lang.Thread* passando como parâmetro um objeto que implementa a interface *Runnable* (que é o nosso objeto *ThreadInterface*). Após isso podemos usar este objeto da mesma forma que fizemos com o exemplo anterior;
7-11: Este código é idêntico ao exemplo anterior.

A partir de agora todos os nossos exemplos de threads serão com threads que herdam da classe *java.lang.Thread* devido às facilidades que a mesma oferece (por exemplo o método *sleep(long)*).

Sincronização

Uma vez que duas linhas de execução (threads) são disparadas dentro de uma aplicação, muitas vezes é necessário **sincronizar** essas linhas para evitar que assim dados compartilhados se tornem inconsistentes ou então que essas linhas atuem em momentos errados. Algumas bibliotecas que suportam threads oferecem este recurso através de **semáforos** (exemplo: Pthreads[6]) que é uma construção de baixo nível que permite a sincronização mas tem o incoveniente de ser mais propensa a erros por parte do programador. Uma outra forma de prover sincronização é através do uso de uma construção de mais alto nível chamada **monitor**. Algumas bibliotecas oferecem ambas as construções (exemplo JThreads/C++[5]), no caso de Java a própria linguagem incorpora o conceito de monitores através da palavra-chave **synchronized** e de alguns métodos disponíveis para qualquer objeto que iremos expor no decorrer do texto.
Antes de começar a mostrar exemplos de sincronização é necessário falar de alguns aspectos. O primeiro é que todos os objetos em Java possuem implicitamente um monitor associado a cada um deles. É através de operações sobre esses monitores que a sincronização em Java ocorre. Por falar em sincronização, existem duas formas delas: sincronização de **competição** e sincronização de **cooperação**. Os dois exemplos que se seguem mostram exatamente como implementar em Java

threads que aplicam essas duas formas de sincronização. Sincronização de competição ocorre quando duas ou mais threads competem pelo mesmo recurso compartilhado e por isso precisam se comunicar de alguma forma para que os dados não se tornem inconsistentes devido à concorrência das threads no acesso aos mesmos. Já a sincronização de cooperação ocorre quando o aspecto mais importante de duas ou mais threads não é a competição por um recurso mas sim a comunicação entre elas para que uma atue num momento específico que depende de uma ação ou estado da outra.

Sincronização de Competição

Iremos mostrar agora como sincronizar várias threads que competem por um recurso compartilhado. A idéia é parecida com o exemplo anterior aonde várias threads adicionam um String num buffer compartilhado. Ao invés de usarmos um objeto *StringBuffer* como buffer, vamos criar o nosso próprio buffer através da classe Buffer que está descrita abaixo:

```
1.public class Buffer{

2.    private StringBuffer buf;
```

1: Declaração da classe;
2: O buffer é implementado através de um objeto privado *StringBuffer* que vai ter seu acesso controlado por métodos da classe;
3-4: Construtor da classe;
5-7: Este método insere uma String no buffer. A palavra-chave **synchronized** será explicada adiante. Ao contrário do exemplo anterior, este método insere um caracter de cada vez no buffer através do método *append(char)* ao invés de inserir o String de uma vez só com o método *append(String)*. Esta mudança é proposital para mostrar o problema da sincronização que explicaremos a seguir; 8-9: Método que imprime o buffer. Mais uma vez deixamos a explicação da palavra-chave **synchronized** para depois.

Agora vamos mostrar a classe ThreadHeranca2 que implementa as threads que irão adicionar caracteres no buffer:

```
8.          for(int i=0; i < 4000; i++)
9.              buf.add(texto);
        }
    }
```

1: Declaração da classe;
2: Referência ao Buffer compartilhado que a thread irá usar para acrescentar Strings;
3: Referência ao String que será inserido no buffer;
4-6: Construtor da classe que inicializa as referências;
7-9: Ação da thread que consiste em adicionar 4000 vezes no buffer a String *texto*.

Finalmente iremos mostrar a última classe que é a Principal3 que contém o método *main* da aplicação:

```
1. public class Principal3{
2.     public static void main(String []args){
3.         Buffer buf = new Buffer();

4.         ThreadHeranca2 her1 = new ThreadHeranca2(buf, "thread
1\n");
5.         ThreadHeranca2 her2 = new ThreadHeranca2(buf, "thread
2\n");
6.         ThreadHeranca2 her3 = new ThreadHeranca2(buf, "thread
3\n");
7.         ThreadHeranca2 her4 = new ThreadHeranca2(buf, "thread
4\n");
8.         ThreadHeranca2 her5 = new ThreadHeranca2(buf, "thread
5\n");
9.         ThreadHeranca2 her6 = new ThreadHeranca2(buf, "thread
6\n");

10.        her1.start();
11.        her2.start();
12.        her3.start();
13.        her4.start();
14.        her5.start();
15.        her6.start();

16.        for(int i=0; i < Integer.MAX_VALUE/10; i++);
17.        buf.print();

        }
    }
```

1-2: Declaração da classe e do método *main*;
3: Criação do buffer que será compartilhado pelas threads;
4-9: Criação das threads passando para cada uma uma String diferente e o mesmo buffer;
10-15: Disparo das threads;
16: Espera forçada para aguardar que as threads tenham tempo de executar todas as inserções antes de imprimir o buffer;
17: Impressão do buffer para verificação dos resultados.

Vamos agora explicar a sincronização na aplicação. Antes disso experimente retirar a palavra-chave *synchronized* do método *add* da classe Buffer, execute algumas vezes e analise a impressão do buffer. Como a saída é um pouco longa, aconselhamos a você redirecionar a saída padrão do programa para um arquivo (ex. *java Principal3 > arquivoDeSaida*) e assim analisar o arquivo depois de finalizada a aplicação. Você deve ter verificado que algumas linhas são impressas de uma maneira esquisita. Por exemplo, em minha máquina eu notei uma linha assim: *thrthread 3ead 4*. O

que significa isso? O que aconteceu foi exatamente um problema de sinchronização de competição. Enquanto a thread 4 estava imprimindo caracter a caracter a sua String, a Máquina Virtual Java "chaveou" a execução para a thread 3 que por sua vez imprimiu sua String no meio da impressão da String da thread 4. Isso obviamente não é o que queremos. Como controlar isso? É exatamente através da palavra-chave *synchronized* no método add. Quando um método com esta palavra-chave é chamado ele tenta obter o monitor do objeto que contém este método para travá-lo. Caso alguma outra thread já tenha adquirido este monitor antes, a thread corrente é posta para esperar numa fila até que o monitor esteja disponível para ele. Assim que o monitor estiver disponível a thread trava-o e inicia a execução. Até o fim do método nenhum outro método synchronized chamado por outra thread poderá começar a execução até que a thread corrente libere o monitor, que vai acontecer quando o método terminar. **Atenção:** a Máquina Vitual Java apenas impede que outras threads chamando métodos *synchronized* não continuem a execução a espera da liberação do monitor. Se uma outra thread chamar neste objeto um método sem a palavra-chave *synchronized*, ele será executado normalmente e não esperará por liberação nenhuma, portanto monitores apenas controlam códigos que contenham a palavra-chave *synchronized*. Falamos códigos porque esta palavra-chave pode ser aplicada a blocos de código como por exemplo:

```
1.    public void add(String novo){
2.        synchronized(this) {
3.            for(int i=0; i < novo.length(); i++)
4.                buf.append(novo.charAt(i));
          }
      }
```

O código acima tem a mesma função do que se fosse aplicado na assinatura do método, a diferença é que nele você especifica de que objeto você quer o monitor (no caso acima é o monitor do próprio objeto).

Mas vamos voltar ao nosso código: como agora estamos controlando o acesso das threads na escrita no buffer, é impossível acontecer o que foi mostrado acima. Quando uma thread estiver imprimindo no buffer, mesmo que a Máquina Virtual Java dê a vez para outra thread executar, ela não conseguirá obter o monitor do buffer pois este está com a primeira thread que só liberará quando terminar a sua impressão.
A palavra-chave *synchronized* é poderosa, no entanto deve ser utilizada de maneira criteriosa. Isso porque se ela for usada desplicentemente, ela irá atrasar a execução da aplicação de maneira desnecessária pois todo os outros blocos com esta palavra-chave ficarão a espera para executar. Apenas acesso a código compartilhado deve ser posto em blocos *synchronized*. Uma boa referência sobre dicas e padrões de programação com threads está em [2].

Sincronização de Cooperação

Agora vamos mostrar como utilizar a linguagem Java para implementar uma aplicação que precisa implantar a sincronização de cooperação.
Primeiramente antes de mostrar as contruções da linguagem vamos apresentar o problema. Imagine que a thread principal da aplicação (aquela que contém o método *main*) vai imprimir o ~~conteúdo de um buffer, contudo ela apenas deve imprimir quando o~~ buffer estiver cheio. Outras threads ficarão concorrentemente adicionando caracteres ao buffer até preenchê-lo. Aonde está a sincronização de cooperação? Está exatamente na necessidade de as threads que preenchem o buffer avisar a thread principal quando o buffer encheu para que ela possa imprimi-lo e assim esvazie o buffer. Vamos examinar o código e depois vamos analisar cada elemento dele. Primeiro vamos ver a classe Buffer2 que implementa o buffer que vamos utilizar:

```
7.          tam = 0;
        }

8.      public synchronized void inserir(char c){
9.          if(tam==MAX)
10.             notify();
11.         else
12.             buf[tam++] = c;
        }

13.     public synchronized String esvaziar(){
14.         if(tam < MAX)
15.             try{
16.                 wait();
17.             }catch(InterruptedException ie){
18.             }
19.         tam = 0;
20.         return new String(buf);
        }
    }
```

1: Declaração da classe;
2: O buffer será implementado através de um array de caracteres visto que seu tamanho agora é fixo, por isso não precisamos mais usar as facilidades da classe *StringBuffer* que auto-incrementa o buffer interno;
3: Variável que guarda o tamanho do buffer;
4: Constante que guarda o tamanho máximo do buffer;
5-7: Construtor da classe;
8: Método que irá acrescentar caracteres no buffer. Como várias threads irão concorrentemente querer acrescentar caracteres, é imprescindível o uso da palavara-chave *synchronized* (sincronização de competição);
9-12: Primeiramente este código testa se o buffer está cheio, caso não esteja ele insere o caractere e atualiza o valor da variável *tam*. Caso o buffer esteja cheio ele avisa à thread principal que é hora dela "agir" (explicaremos a função *notify()* em mais detalhas adiante);
13: Método que irá esvaziar o buffer e devolver seu contéudo ao chamador do método. Deve estar claro para o leitor que é imprescindível o uso da palavra-chave *synchronized* devido ao acesso simultâneo das variáveis internas;
14-18: Testa para ver se o buffer está cheio. Caso não esteja a thread que chamou este método irá esperar (detalhes de como funciona o método *wait()* serão dados em breve);
19: A esta altura o buffer está cheio e é hora de esvaziá-lo. Isso é feito atualizando a variável *tam*;
20: Retorno do conteúdo do buffer na forma de um objeto *String*.

Agora vamos ver a implementação da classe ThreadHeranca3 que implementa as threads que irão acrescentar caracteres ao buffer:

```
1. public class ThreadHeranca3 extends Thread{

2.      private Buffer2 buf;
3.      private char c;

4.      public ThreadHeranca3(Buffer2 buf, char c){
5.          this.buf = buf;
```

```
10.             try {
11.                 sleep((long)5);
12.             } catch (InterruptedException ie) {
                }
            }
        }
    }
```

1: Declaração da classe;
2: Referência ao buffer compartilhado;
3: Caracter que será inserido pela thread no buffer;
4-6: Construtor da thread que inicializa o buffer compartilhado e o caracter a ser impresso;
7-12: Método principal da thread. Ele tenta acrescentar 1000 caracteres no buffer. Após cada inserção a thread espera 5 milisegundos antes de executar a próxima inserção.

Finalmente é hora de ver o último código desta seção (e deste texto, ufa!!) que é o código que contém o método *main*. Vamos ver a classe Principal4:

```
1. public class Principal4{
2.     public static void main(String []args){
3.         Buffer2 buf = new Buffer2();

4.         ThreadHeranca3 her1 = new ThreadHeranca3(buf,'1');
5.         ThreadHeranca3 her2 = new ThreadHeranca3(buf,'2');
6.         ThreadHeranca3 her3 = new ThreadHeranca3(buf,'3');

7.         her1.start();
8.         her2.start();
9.         her3.start();

10.        String result;
11.        for(int i=0; i < 10; i++){
12.            result = buf.esvaziar();
13.            System.out.println("\n\nImpressao # " + (i+1) +
":\n Buffer = " +
                                    result + " -> Tamanho: " +
result.length());
        }
    }
}
```

1: Declaração da classe e do método *main*;
3: Criação do buffer compartilhado;
4-6: Criação das threads que irão acrescentar caracteres distintos no mesmo buffer;
7-9: Disparo das threads;
10: String que guardará o retorno da operação de esvaziamento do buffer;
11-13: Esta thread tenta esvaziar o buffer 10 vezes e imprime o seu resultado bem como o seu tamanho para verificarmos o sucesso da sincronização.

Rode esta aplicação e verifique os resultados. Depois da última impressão pode ser que em sua máquina a aplicação demore alguns segundos para finalizar, isso se deve a espera para as threads que ainda querem imprimir no buffer terminem.

Agora vamos ao que interessa: como a Máquina Virtual Java permitiu essa sincronização? Tudo se deve aos métodos *wait()* e *notify()* que estão presentes na classe *java.lang.Object*, portanto disponíveis para qualquer objeto. Vamos tentar organizar as idéias - para cada objeto em Java existem duas filas: uma é a fila de threads que acessam métodos *synchronized* deste objeto e estão prontas a espera da liberação do monitor para continuarem a executar; a segunda fila (que na verdade não é uma fila mas um conjunto de threads visto que nesta estrutura não existe a noção

de ordem de chegada) é um espaço de threads que estão "dormindo", esperando até que alguma outra thread em execução as acordem e coloquem elas na primeira fila a espera para adquirir o monitor do objeto. Vamos chamar a primeira de *fila de threads prontas* e a segunda de *conjunto de threads a espera*. É exatamente este o papel dos métodos *wait()* e *notify()*. O primeiro método coloca a thread que possui o monitor do objeto para esperar no conjunto de threads a espera e libera o monitor para outra thread que está na fila de threads prontas. O segundo método ao contrário "acorda uma thread qualquer do conjunto de threads a espera (por isso é um conjunto e não uma fila, você não pode garantir que thread será acordada, essa escolha é aleatória da Máquina Virtual Java) e a coloca na fila de threads prontas. Ela agora só será executada quando o monitor estiver disponível para ela (veja a figura 3).

Figura 3

Vale a pena agora enfatizar algo que já foi dito: a fila de threads prontas e o conjunto de threads a espera são únicas por objeto Java, ou seja, cada objeto Java possui um monitor implicitamente associado a ele e esses dois conjuntos de threads. É a Máquina Virtual Java que gerencia e controla ambos os conjuntos de cada objeto.

Existe ainda um outro método chamado *notifyAll()* que ao invés de acordar uma thread, acorda todas que estão a espera e as coloca na fila de threads prontas. Mais uma vez você não tem como garantir a ordem que estas thredas ocuparão na fila de threads prontas. Essa ordem é aleatória e depende da Máquina Virtual Java.

Todos esses métodos (*wait(), notify()* e *notifyAll()*) são métodos disponíveis a qualquer objeto Java, no entanto para uma thread poder chamá-los ela precisa adquirir o monitor deste objeto (através da palavra-chave *synchronized*), caso contrário uma exceção de Runtime chamada *IllegalMonitorStateException* será levantada. Isso é altamente compreensível, não tem lógica uma thread chamar o método *wait()* de um objeto, liberar um monitor que ela nem mesmo possui e entregá-lo a outra thread. Ou então várias threads simultaneamente chamarem *notify()* em um mesmo objeto e acordarem não uma mais duas threads. Todas essas ações precisam ser sincronizadas, este é outro motivo porque existe a palavra-chave *synchronized* nos métodos *inserir* e *esvaziar()* da classe Buffer2.

Conclusão

Este texto está muito longe de esgotar o assunto sobre threads em Java e de cobrir todas as possibilidades oferecidas por esta incrível linguagem para programação concorrente. Ele vai apenas um pouco além do básico tentando sempre que possível explicar como as coisas funcionam "por baixo dos panos (ou do teclado)". Uma vasta bibliografia sobre threads em Java está disponível, algumas na Web, algumas impressas e a leitura delas é altamente recomendável. A próxima seção lista as principais referências sobre o assunto e em sua maioria estão na língua inglesa. As referências [1] e [3] são básicas para qualquer programador Java. A referência [2], como já foi dito, é ótima para aqueles que querem aprender dicas e padrões de como programar correta e eficientemente com threads em Java. Para aquele mais interessado no aspecto teórico e acadêmico deste assunto, um bom texto está em [4]. Vale também a pena verificar como outras linguagens implementam estes conceitos. A biblioteca JThreads/C++[5] por exemplo estende a biblioteca Pthreads[6] para C (de mais baixo nível) e faz a interface de programação para C++ ser muito parecida com a interface de Java. O manual desta biblioteca faz um comparativo de programas multi-thread Java com programas multi-thread C++ e inclusive explica as construções de Java para threads de uma maneira muito didática.
A programação concorrente é muito poderosa porém muito traiçoeira. Você pode escrever um programa concorrente que funciona um milhão de vezes corretamente e só na próxima vez aparece um erro trágico. Isso se deve ao carater imprevisível e aleatório da ordem de execução das threads. Tudo pode influir na execução de um programa concorrente: a carga de processos no processador, a quantidade de memória disponível no sistema, entre tantas outras variáveis que podem influenciar a sua execução. Não é fácil depurar um programa concorrente errado pois cada execução será provavelmente diferente da anterior. Toda esta dificuldade é ao mesmo tempo amedrontadora e desafiante. Um bom domínio do assunto e das construções da linguagem é imprescindível para um bom programador de aplicações concorrentes.

Qualquer sugestão, crítica ou dúvida sobre este texto será muito bem-vinda e pode ser feita pelo endereço theocjr@jspbrasil.com.br.

Bibliografia

- Sun Microsystems. *The Java Tutorial: A Pratical Guide for Programmers.* http://java.sun.com/docs/books/tutorial/
- Lea, Doug. *Concurrent Programming in Java: Design Principles and Patterns* Addison-Wesley. 1996.
- Sun Microsystems. *Java API Documentation.* http://java.sun.com/j2se/1.4/docs/api/
- Tanenbaum, Andrew S., et al. *Sistemas Operacionais: Projeto e Implementação.* 2. ed. Porto Alegre. Bookman. 2000.
- IONA Technologies inc. *JThreads/C++: Java-like threads for C++.* Versão 2.0.0. 2001. www.ooc.com/jtc/
- Nichols, Bradford., et al. *Pthreads Programming: A POSIX Standard for Better Multiprocessing.* O'Reilly Nutshell.