Artigo

Utilizando Threads - parte 2

Veja neste artigo de Paulo César M. Jeveaux: Utilizando Threads - parte 2

por Paulo César M. Jeveaux

O exemplo é semelhante a classe VariasThread2.java, porém, a classe UmaThread já é derivada da classe Thread. Assim, o método run() pode existir sem implementar a interface Runnable e o método sleep(long tempo) não precisa ser precedido da palavra Thread. Para armazenar a identificação da Thread foi utilizada as variáveis da própria classe Thread. Para obter o nome armazenado foi utilizado o método getName(); O leitor pode se perguntar qual o melhor método de realização de thread: utilizar a interface Runnable ou derivar da classe Thread. Na verdade ambas implementações levam a resultados semelhantes. Todavia, devido a restrição de Java não ter herança múltipla, a abordagem de interfaces permite que a classe, além de implementar uma thread, seja derivada de outra classe. **Prioridades de Threads**

As threads sempre iniciam sua execução com a prioridade herdada da superclasse, que por default é igual a 5 (Thread.NORM_PRIORITY). Porém, o programador pode alterar a prioridade da thread como convier, dentro do range de prioridades de 1 (Thread.MIN_PRIORITY) e 10 (Thread.MAX_PRIORITY).
A JVM tem um dispositivo que escala as threads. Este dispositivo sempre tenta escalonar a thread de maior prioridade que esteja no estado executável. A thread escalonada passa a promover outras threads nos seguintes momentos:

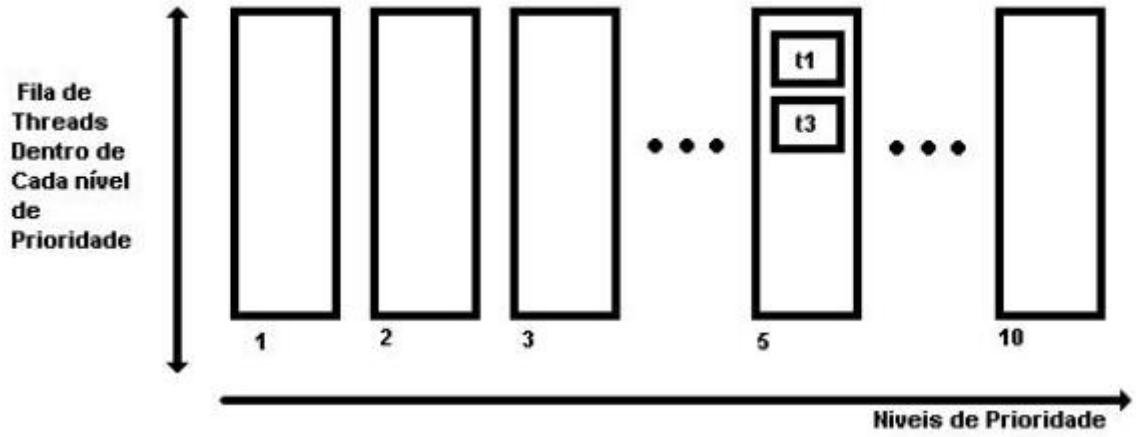
25.Quando é chamado o método yield();

26.Quando passa para o estado de encerrada ou bloqueada;

27.Quando passa a estar executável outra thread de prioridade mais alta.

Em alguns S.O., tais como o Solaris, threads nativas de mesma prioridade devem ser promovidas para poderem ser executadas. Isto pode ser efetuado pelo método yield().

A figura que segue mostra os dez níveis de prioridade que podem ser inseridas as threads. Dentro de cada nível



Para exemplificar o uso de prioridades, o arquivo TestePrioridade.java mostra a execução de três threads de prioridade distinta.

```
class Escrita extends Thread {
    private int i;
    Escrita(String identificacao) {
        Super(identificacao);
    }
    Escrita(String identificação, int prioridade) {
        super(identificacao);
        setPriority(prioridade);
    }
    public void run() {
```

```
                System.out.println(getName() + " " + i);
                yield();
            }
        }
}

public class TestePrioridade {
    publis static void main(String[] args) {
        int i = 0;
        new Escrita("Menor",4).start();
        new Escrita("Maior",6).start();
        new Escrita("Default",5).start();
    }
}
```

Neste programa pode ser observado que a thread de maior prioridade é executada muitas vezes mais que a thread de prioridade default, e a thread de prioridade baixa raramente executada.

Sincronismo de Threads

A solução para este problema é utilizar um mecanismo de sincronização que permita a thread acessar a base de dados compartilhada apenas quando os dados estiverem estáveis (outras threads não estejam manipulando estes dados).

Java adotou a palavra chave synchronized para informar que um determinado bloco deve estar síncrono com as demais threads. O sincronismo gera um bloco atômico (indivisível). Assim, este bloco passa a ser protegido, evitando que a thread atual seja interrompida por outra thread, independente da prioridade.

O arquivo BancoSemSincronismo.java apresenta a simulação de operações de transferências bancárias. Este programa, extraído parcialmente de [COR 99], não se preocupa com detalhes de sincronismo entre as operações.

```
public class BancoSemSincronismo {

    public static void main(String[] args) {

        Banco b = new Banco();

        for(int i=0; i < Banco.NUN_CONTAS; i++) {

            new Transferencias(b, i).start();

        }

    }

}
```

A classe BancoSemSincronismo cria um banco hipotético b e Banco.NUM_CONTAS contas bancárias para efetuar transferências randômicas. Ao criar as contas bancárias, está sendo criada as threads que efetuarão as transações bancárias.

não há depósitos e/ou retiradas, a quantia total de dinheiro no banco deve, pelo menos teoricamente, permanecer inalterada.

O método transfere(int de, int para, int quantidade), efetua a transferência de quantidade de conta de para a conta para. Antes de efetuar a transferência, o método tem uma proteção para a conta nunca ficar com um valor negativo (devendo dinheiro). Caso isto ocorra, a thread adormece por 5 milisegundos. Observe o código:

```
while(conta[de] < quantidade) {
        try {
            Thread.sleep(5);
        }
        catch(InterruptedException e) {}
}
```

O que ocorre na prática é que outra thread, em algum instante aleatório colocará a quantia necessária para que a transação possa ser efetivada. E quando isto ocorrer, o método decrementa a quantia da conta inicial e coloca a mesma quantia na conta do favorecido pela transferência.

```
conta[de] - = quantidade;
conta[para] + = quantidade;
```

A cada 5000 transferências o programa exibe o valor total de dinheiro no banco. A classe completa pode ser

```
public static final int NUN_CONTAS = 10;        //Número de contas no banco

private long conta[];                           //Array que armazena o valor das contas

private int transfere;                  //indica o número de transferências bancárias



public Banco() {
        conta = new long[NUN_CONTAS];
        for(int i = 0; i < NUN_CONTAS; i++) {
            conta[i] = VALOR_TOTAL;
        }
        tranfere = 0;
        teste();
    }
```

```
        while(conta[de] < quantidade) {
            try {
                Thread.sleep(5);
            }
            catch(Interruptedexception e) {}
        }
        conta[de] - = quantidade;
        conta[para] + = quantidade;
        transfere++;
        if(transfere % 5000 == 0)
            teste();
    }
    public void teste() {
        long soma = 0;
        for(int i = 0; i < NUM_CONTAS; i++)
            soma + = conta[i];
        System.out.println("Transações: " + transfere + "Soma: " + soma);
    }
```

transferência bancária. Este classe é responsável pela trhead das transações bancárias.

O método run() calcula uma conta aleatória para a qual irá ser transferida uma quantia igualmente aleatória. A thread permanece adormecida por 1 milisegundo após ter sido efetuada a operação.

```
class Transferência extends Thread {
    private Banco db;
    private int de;
    public Transferência(Banco b, int i) {
        de = i;
        this.b = b;
    }
    public void run() {
        while(true) {
            int para = (int)(Banco.NUM_CONTAS * Math.random());
            if(para = = de)
                para = para % Banco.NUN_CONTAS;
            int quantidade=1+(int)(Banco.VALOR_CONTAS * Math.random()) /2
            b.transfere(de, para, quantidade);
            try {
```

```
            }
        }
}
```

Observe um resultado obtido, a partir da execução do programa:

Transações: 0 Soma: 10000

Transações: 5000 Soma: 97051

Transações: 10000 Soma: 97051

Transações: 15000 Soma: 96356

Transações: 20000 Soma: 96356

Transações: 25000 Soma: 97885

Transações: 30000 Soma: 99077

Através do resultado, pode-se que a soma total do dinheiro das contas passou a ser alterada, mesmo não ocorrendo nenhum depósito ou retirada. O que acontece é que a JVM, pode promover outra thread a qualquer instante, desde que tenha terminado a execução de uma bytcode (nunca durante a sua execução). Cada instrução pode ser implementada por vários bytecodes; sendo assim, algumas instruções que deveriam ser atômicas podem ser divididas com uma razoável probabilidade. Este efeito ocorre no método transfere(int de, int para, int quantidade), mais especificamente nas instruções abaixo:

```
conta[de] - = quantidade;
```

conta é comum.

O método teste() consiste em outro trecho de código que também pode dar problema, mas apenas temporário, pois enquanto está sendo calculado o valor total de uma conta, a thread pode ser interrompida e a quantidade ser alterada por outra thread. Obviamente como esta operação ocorre a cada 5000 transações a probabilidade de erros é bem menor.

```
if(transfere % 5000 = = 0) Teste();
```

Para solucionar o problema de sincronismo, Java dispõe de um mecanismo realizado através da palavra chave sinchronyzed e dos métodos wait(), notify() ou notifyAll(). O arquivo BancoComSincronismo.java apresenta a nova versão sincronizada, como pode ser observado nos trechos de código que foram alterados.

```
public sinchronyzed void Transfere(int de, int para, int quantidade) {
    while(conta[de] < quantidade) {
        try {
            wait():
        }catch(InterruptedException e) {}
    }
    conta[de] - = quantidade;
    conta[para] + = quantidade;
    transfere++;
    if(transfere % 5000 = = 0)
```

```
        notify();
}
```

Devido a palavra chave sinchronyzed, o método transfere(int de, int para, int quantidade) passa a ser atômico. Assim, nenhuma thread pode interromper a sua execução.

Quando a thread chega ao final do bloco protegido é evocado o método notify() para promover outra thread que será escalonada. Porém, para evitar que a proteção para valores negativos pare o processamento do programa, a thread chama o método wait(), que bloqueia a thread corrente (quebra a atomicidade) e promove outras threads. Estas novas threads colocarão valores suficientes para a thread atual sair do laço de proteção. É claro que este sistema deve ser bem projetado para não surgir situações de deadlocks; onde todas as threads ficam bloqueadas porque alguma não foi satisfeita e nenhuma outra thread está ativa para poder satisfazer as condições.

Bibliografia

28.Deitel H.M., Deitel P.J., Java Como Programar, 3ª Ed.

29.Programação em Java – Universidade do Vale do Rio dos Sinos

Tecnologias:

Java