CAPÍTULO 20

Apêndice - Problemas com concorrência

"Quem pouco pensa, engana-se muito."
— Leonardo da Vinci

20.1 - THREADS ACESSANDO DADOS COMPARTILHADOS

O uso de Threads começa a ficar interessante e complicado quando precisamos compartilhar objetos entre várias Threads.

Imagine a seguinte situação: temos um Banco com milhões de Contas Bancárias. Clientes sacam e depositam dinheiro continuamente, 24 horas por dia. No primeiro dia de cada mês, o Banco precisa atualizar o saldo de todas as Contas de acordo com uma taxa específica. Para isso, ele utiliza o `AtualizadorDeContas` que vimos anteriormente.

O `AtualizadorDeContas`, basicamente, pega uma a uma cada uma das milhões de contas e chama seu método `atualiza`. A atualização de milhões de contas é um processo demorado, que dura horas; é inviável parar o banco por tanto tempo até que as atualizações tenham completado. É preciso executar as atualizações paralelamente às atividades, de depósitos e saques, normais do banco.

Ou seja, teremos várias threads rodando paralelamente. Em uma thread, pegamos todas as contas e vamos chamando o método `atualiza` de cada uma. Em outra, podemos estar sacando ou depositando dinheiro. Estamos compartilhando objetos entre múltiplas threads (as contas, no nosso caso).

Imagine a seguinte possibilidade (mesmo que muito remota): no exato instante em que o atualizador está atualizando uma Conta X, o cliente dono desta Conta resolve efetuar um saque. Como sabemos, ao trabalhar com Threads, o

escalonador pode parar uma certa Thread a qualquer instante para executar outra, e você não tem controle sobre isso.

Veja essa classe Conta:

```
1  public class Conta {
2
3    private double saldo;
4
5      // outros métodos e atributos...
6
7      public void atualiza(double taxa) {
8        double saldoAtualizado = this.saldo * (1 + taxa);
9        this.saldo = saldoAtualizado;
10     }
11
12     public void deposita(double valor) {
13       double novoSaldo = this.saldo + valor;
14       this.saldo = novoSaldo;
15     }
16 }
```

Imagine uma Conta com saldo de 100 reais. Um cliente entra na agência e faz um depósito de 1000 reais. Isso dispara uma Thread no banco que chama o método `deposita()`; ele começa calculando o `novoSaldo` que passa a ser 1100 (linha 13). Só que por algum motivo que desconhecemos, o escalonador pára essa thread.

Neste exato instante, ele começa a executar uma outra Thread que chama o método `atualiza` da mesma `Conta`, por exemplo, com taxa de 1%. Isso quer dizer que o `novoSaldo` passa a valer 101 reais (linha 8). E, nesse instante o escalonador troca de Threads novamente. Ele executa a linha 14 na Thread que fazia o depósito; o saldo passa a valer 1100. Acabando o deposita, o escalonador volta pra Thread do atualiza e executa a linha 9, fazendo o saldo valer 101 reais.

Resultado: o depósito de mil reais foi totalmente ignorado e seu Cliente ficará pouco feliz com isso. Perceba que não é possível detectar esse erro, já que todo o código foi executado perfeitamente, sem problemas. **O problema, aqui, foi o acesso simultâneo de duas Threads ao mesmo objeto**.

E o erro só ocorreu porque o escalonador parou nossas Threads naqueles exatos lugares. Pode ser que nosso código fique rodando 1 ano sem dar problema algum e em um belo dia o escalonador resolve alternar nossas Threads daquela forma. Não sabemos como o escalonador se comporta! Temos que proteger nosso código contra esse tipo de problema. Dizemos que essa classe não é *thread safe*, isso é, não está pronta para ter uma instância utilizada entre várias threads concorrentemente.

O que queríamos era que não fosse possível alguém atualizar a `Conta` enquanto outra pessoa está depositando um dinheiro. Queríamos que uma Thread não pudesse mexer em uma `Conta` enquanto outra Thread está mexendo nela. Não há como impedir o escalonador de fazer tal escolha. Então, o que fazer?

20.2 - Controlando o acesso concorrente

Uma ideia seria criar uma **trava** e, no momento em que uma Thread entrasse em um desses métodos, ela trancaria a entrada com uma chave. Dessa maneira, mesmo que sendo colocada de lado, nenhuma outra Thread poderia entrar nesses métodos, pois a chave estaria com a outra Thread.

Essa ideia é chamada de **região crítica**. É um pedaço de código que definimos como crítico e que não pode ser executado por duas threads ao mesmo tempo. Apenas uma thread por vez consegue entrar em alguma região crítica.

Podemos fazer isso em Java. Podemos usar qualquer objeto como um **lock** (trava, chave), para poder **sincronizar** em cima desse objeto, isto é, se uma Thread entrar em um bloco que foi definido como sincronizado por esse lock, apenas uma Thread poderá estar lá dentro ao mesmo tempo, pois a chave estará com ela.

A palavra chave `synchronized` dá essa característica a um bloco de código e recebe qual é o objeto que será usado como chave. A chave só é devolvida no momento em que a Thread que tinha essa chave sair do bloco, seja por `return` ou disparo de uma exceção (ou ainda na utilização do método `wait()`)..

Queremos, então, bloquear o acesso simultâneo a uma mesma `Conta`:

```
public class Conta {

  private double saldo;

  // outros métodos e atributos...

  public void atualiza(double taxa) {
    synchronized (this) {
      double saldoAtualizado = this.saldo * (1 + taxa);
      this.saldo = saldoAtualizado;
    }
  }

  public void deposita(double valor) {
    synchronized (this) {
      double novoSaldo = this.saldo + valor;
      this.saldo = novoSaldo;
    }
```

```
  }
}
```

Observe o uso dos blocos `synchronized` dentro dos dois métodos. Eles bloqueiam uma Thread utilizando o mesmo objeto `Conta`, o `this`.

Esses métodos são mutuamente exclusivos e só executam de maneira atômica. Threads que tentam pegar um lock que já está pego, ficarão em um conjunto especial esperando pela liberação do lock (não necessariamente numa fila).

Sincronizando o bloco inteiro

É comum sempre sincronizarmos um método inteiro, normalmente utilizando o `this`.

```
public void metodo() {
  synchronized (this) {
    // conteúdo do metodo
  }
}
```

Para este mesmo efeito, existe uma sintaxe mais simples, onde o `synchronized` pode ser usado como modificador do método:

```
public synchronized void metodo() {
  // conteúdo do metodo
}
```

Mais sobre locks, monitores e concorrência

Se o método for estático, será sincronizado usando o lock do objeto que representa a classe (`NomeDaClasse.class`).

Além disso, o pacote `java.util.concurrent`, conhecido como **JUC**, entrou no Java 5.0 para facilitar uma série de trabalhos comuns que costumam aparecer em uma aplicação concorrente.

Esse pacote ajuda até mesmo criar threads e pool de threads, através dos Executors.

20.3 - Vector e Hashtable

Duas collections muito famosas são `Vector` e `Hashtable`, a diferença delas com suas irmãs `ArrayList` e `HashMap` é que as primeiras são thread safe.

Você pode se perguntar porque não usamos sempre essas classes thread safe. Adquirir um lock tem um custo, e caso um objeto não vá ser usado entre diferentes threads, não há porque usar essas classes que consomem mais recursos. Mas nem sempre é fácil enxergar se devemos sincronizar um bloco, ou se devemos utilizar blocos sincronizados.

Antigamente o custo de se usar locks era altíssimo, hoje em dia isso custa pouco para a JVM, mas não é motivo para você sincronizar tudo sem necessidade.

20.4 - Um pouco mais...

1. Você pode mudar a prioridade de cada uma de suas Threads, mas isto também é apenas uma sugestão ao escalonador.

2. Existe um método `stop` nas Threads, porque não é boa prática chamá-lo?

3. Um tópico mais avançado é a utilização de `wait`, `notifiy` e `notifyAll` para que as Threads comuniquem-se de eventos ocorridos, indicando que podem ou não podem avançar de acordo com condições

4. O pacote `java.util.concurrent` foi adicionado no Java 5 para facilitar o trabalho na programação concorrente. Ele possui uma série de primitivas para que você não tenha de trabalhar diretamente com `wait` e `notify`, além de ter diversas coleções thread safe.

20.5 - Exercícios avançados de programação concorrente e locks

Exercícios só recomendados se você já tinha algum conhecimento prévio de programação concorrente, locks, etc.

1. Vamos enxergar o problema ao se usar uma classe que não é *thread-safe*: a `ArrayList` por exemplo.

Imagine que temos um objeto que guarda todas as mensagens que uma aplicação de chat recebeu. Vamos usar uma `ArrayList<String>` para armazená-las. Nossa aplicação é *multi-thread*, então diferentes threads vão inserir diferentes mensagens para serem registradas. Não importa a ordem que elas sejam guardadas, desde que elas um dia sejam!

Vamos usar a seguinte classe para adicionar as queries:

```
public class ProduzMensagens implements Runnable {
  private int comeco;
  private int fim;
  private Collection<String> mensagens;

  public ProduzMensagens(int comeco, int fim, Collection<String> mensagens) {
    this.comeco = comeco;
    this.fim = fim;
    this.mensagens = mensagens;
  }

  public void run() {
    for (int i = comeco; i < fim; i++) {
      mensagens.add("Mensagem " + i);
    }
  }
}
```

Vamos criar três threads que rodem esse código, todas adicionando as mensagens na **mesma** `ArrayList`. Em outras palavras, teremos threads compartilhando e acessando um mesmo objeto: é aqui que mora o perigo.

```
public class RegistroDeMensagens {

  public static void main(String[] args) throws InterruptedException {
    Collection<String> mensagens = new ArrayList<String>();

    Thread t1 = new Thread(new ProduzMensagens(0, 10000, mensagens));
    Thread t2 = new Thread(new ProduzMensagens(10000, 20000, mensagens));
    Thread t3 = new Thread(new ProduzMensagens(20000, 30000, mensagens));

    t1.start();
    t2.start();
```

```
    t3.start();

    // faz com que a thread que roda o main aguarde o fim dessas
    t1.join();
    t2.join();
    t3.join();

    System.out.println("Threads produtoras de mensagens finalizadas!");

    // verifica se todas as mensagens foram guardadas
    for (int i = 0; i < 15000; i++) {
      if (!mensagens.contains("Mensagem " + i)) {
        throw new IllegalStateException("não encontrei a mensagem: " + i);
      }
    }

    // verifica se alguma mensagem ficou nula
    if (mensagens.contains(null)) {
      throw new IllegalStateException("não devia ter null aqui dentro!");
    }

    System.out.println("Fim da execucao com sucesso");
  }
}
```

Rode algumas vezes. O que acontece?

2. Teste o código anterior, mas usando `synchronized` ao adicionar na coleção:

```
public void run() {
  for (int i = comeco; i < fim; i++) {
    synchronized (mensagens) {
      mensagens.add("Mensagem " + i);
    }
  }
}
```

3. Sem usar o `synchronized` teste com a classe `Vector`, que **é uma** `Collection` e é *thread-safe*.

O que mudou? Olhe o código do método `add` na classe `Vector`. O que tem de diferente nele?

4. Novamente sem usar o `synchronized`, teste usar `HashSet` e `LinkedList`, em vez de `Vector`. Faça vários testes, pois as threads vão se entrelaçar cada vez de uma maneira diferente, podendo ou não ter um efeito inesperado.

No capítulo de Sockets usaremos threads para solucionar um problema real de execuções paralelas.

CAPÍTULO ANTERIOR:

Apêndice - Sockets

PRÓXIMO CAPÍTULO:

Apêndice - Instalação do Java