

Introdução a Threads Java

Prof. Gerson Geraldo Homrich Cavalheiro

Universidade Federal de Pelotas
Departamento de Informática
Instituto de Física e Matemática
Pelotas – RS – Brasil
<http://gersonc.anahy.org>

1 A Programação Concorrente

O termo concorrência é comumente aplicado para definir sistemas onde é possível compartilhar o uso dos recursos de processamento de um computador – tipicamente o processador – para suportar o fluxo de execução de dois ou mais programas independentes [4] [3]. Nestes casos, a concorrência visa obter ganho de desempenho no sistema e/ou prover um ambiente *time-sharing*. Um ambiente *time-sharing* fornece um mecanismo de compartilhamento dos recursos de processamento de um computador entre processos submetidos por diferentes usuários, desta forma, cada usuário tem o sentimento de que possui um computador próprio para execução de seu programa. Já o uso da programação concorrente para ganho de desempenho, vem de uma constatação de que enquanto um programa realiza operações de E/S, o processador permanece inativo. Considerando que os programas são independentes entre si, a concorrência é empregada para não desperdiçar o tempo de uso da unidade de processamento: assim, quando um programa necessita realizar uma operação de E/S, um novo programa é selecionado para ocupar o processador e avançar na execução de suas instruções.

Seguindo o modelo de von Neumann, a arquitetura dos computadores consiste em uma máquina de cálculo que possui um processador, uma área de memória para dados e código executável e de dispositivos de E/S. Neste modelo, as instruções de um programa são executadas sequencialmente, uma à uma, na mesma ordem em que se encontram no código programa fonte. Programas em execução concorrente, neste caso, não possuem nenhuma restrição temporal de execução e competem pelo tempo de acesso ao processador, dividindo entre si o espaço de memória disponível e o acesso aos dispositivos de E/S.

Para garantir a correta execução dos programas em ambientes que exploram a concorrência a este nível, o aspecto mais importante é o de prover mecanismos que impossibilitem que a execução de um programa interfira na execução de outro. De uma forma mais clara, não permitir que uma instrução de um programa venha alterar alguma posição de memória pertencente a outro programa (proteção de memória) e garantir que um programa correto tenha respeitada a ordem de execução de suas

instruções e que, após ter sido iniciado, tem sua execução completada em um tempo finito (proteção do processador).

O princípio destas considerações é de que os programas são inteiramente dissociados. Cada programa executa uma tarefa específica, não havendo, entre dois programas, qualquer forma de colaboração dinâmica. Portanto, entre os fluxos de execução de cada programa não existe nenhuma dependência temporal, visto que não há troca de informações entre eles. Em outras palavras, o conjunto de fluxos em execução não reflete uma aplicação, e sim aplicações autônomas processando atividades não relacionadas entre si.

Assim temos a primeira definição para concorrência: um conjunto de fluxos de execução independentes que disputam o uso dos recursos de arquitetura para terem suas instruções executadas. Esta disputa reflete o compartilhamento do tempo de uso do processador e de espaço de armazenamento na memória. A abordagem adotada pela programação concorrente [5] possui uma diferença importante: neste caso, busca-se a implementação de uma única aplicação, utilizando-se do recurso de decompô-la em diversos fluxos de execução. Cada um destes fluxos é responsável pela execução de uma parte da solução do problema, o que implica que eles não sejam verdadeiramente independentes. Sendo cada fluxo de execução responsável por uma parte da solução do problema, de alguma forma é necessário que resultados obtidos por um destes fluxos seja comunicado a outro de forma à compor a solução final, ou seja, a solução a ser apresentada pela aplicação. Em [2], este nível de concorrência é identificado como concorrência a nível de unidade.

Desta forma, programar de forma concorrente implica detectar na aplicação as atividades que não possuem restrições temporais e os pontos onde restrições temporais existem. Ou seja, para cada conjunto de instruções de um programa, definir quais produzem resultados necessários ao início da execução de outro conjunto e quais conjuntos são independentes entre si.

O termo concorrência, portanto, aplicando-se também a fluxos de execução onde determinados trechos de instruções não possuem restrições temporais de execução – independentemente do fato de compartilharem ou competirem por recursos de execução –, intercalados por trocas de dados entre estes fluxos. Esta forma de programação permite que diferentes fluxos de execução colaborem entre si, empregando mecanismos de sincronização [2], para atingir um objetivo comum: o resultado esperado.

Um programa concorrente é, desta forma, composto por um conjunto de fluxos de execução que, de alguma forma ordenada, trocam informações. Os fluxos de execução consistem no suporte à execução das tarefas definidas na aplicação e as sincronizações oferecem os mecanismos sobre os quais são implementadas as trocas de dados entre estas tarefas. Um programa concorrente será corretamente executado se todas as tarefas definidas forem executadas respeitando as sincronizações definidas entre elas.

Em um ambiente multithread, a concorrência é definida por um conjunto de atividades concorrentes sendo executadas em uma arquitetura dotada de $n \geq 1$ processadores e um espaço de endereçamento (memória) único. A concorrência pode ser dita real ou temporal, dependendo do número de fluxos de execução ativos em um determinado momento e do número de recursos de processamento – processadores – disponíveis. Caso haja um número de processadores pelo menos igual ao de fluxos de execução ativos, existe a concorrência real, também denominado paralelismo. Caso contrário, um número de fluxos de execução ativos em um determinado instante de tempo maior que o número de processadores disponíveis, existe a concorrência clássica, onde há o compartilhamento de recursos.

A forma de interação mais natural, e menos onerosa, para o compartilhamento de informações em um programa multithread é através do espaço de armazenamento provido pela memória do próprio computador. O mecanismo de comunicação explora o fato de que os dados manipulados pelas instruções contidas em diferentes fluxos de execução são variáveis armazenadas em memória. O acesso a estas variáveis se dá através de triviais operações de leitura e escrita (tipo *load* e *store*). Sendo a memória um recurso pertencente ao nó e compartilhado pelos fluxos em execução, um dado escrito por uma instrução em um fluxo de execução pode ser lido por uma instrução em outro fluxo.

Mesmo que a comunicação entre fluxos de execução se apoie em instruções de leitura e escrita, portanto idênticas às utilizadas em uma execução seqüencial, um cuidado adicional deve ser tomado na sincronização ao acesso a dados compartilhados. Em uma execução seqüencial, o resultado de uma instrução é comunicado a outra instrução através de uma escrita em memória, que reflete o efeito colateral da execução da instrução: a própria alteração do estado do dado na memória, que no futuro servira de entrada para uma outra instrução. A comunicação entre duas instruções se dá através do compartilhamento de certas posições de memória e a sincronização entre duas instruções é realizado automaticamente, executado uma instrução somente a partir do momento em que a instrução que lhe antecede foi completada (aqui cabe salientar que outros níveis de concorrência, como o provido por arquiteturas super-escalares, não estão sendo considerados). Com isto garante-se que as alterações de estado da memória foram realizados e que todo dado lido pela instrução em execução está atualizado.

Em uma execução concorrente, o sincronismo implícito é inexistente e toda instrução em um fluxo de execução que acesse um dado compartilhado com outro fluxo consiste em uma instrução crítica. Mesmo que uma instrução crítica possa acessar um dado compartilhado da mesma forma que uma instrução executada em um fluxo de execução de execução seqüencial, um mecanismo externo deve prover a sincronização entre a instrução que produz o valor para o dado compartilhado e a instrução que necessita deste dado como entrada para sua própria execução. O

correto emprego da sincronização é a única garantia de que a comunicação entre as tarefas vai ser realizada como esperado.

Java é uma linguagem orientada a objetos que oferece na sua interface de programação a classe `Thread`, através da qual o programador pode dotar objetos de fluxos de execução independentes. Desta classe são herdados três métodos cujo uso é associado à manipulação de *threads*: `run`, `start` e `join`.

O método `run` é um método virtual¹ na classe `Thread`, devendo ser implementado pela classe do usuário com o serviço que deve ser executado pela *thread* a ser criada. Os métodos `start` e `join` são implementados pela própria classe `Thread`. Eles permitem, respectivamente, iniciar a execução de uma nova *thread* e sincronizar com seu término.

É importante observar que a função a ser executada pela *thread* é definido por um método de uma classe. Assim sendo, para executar uma *thread*, é necessário a criação de um objeto da classe que especializa a classe `Thread`. Os métodos `start` e `join` são invocados neste objeto, e o método `run`, invocado implicitamente por `start`, é executado no escopo deste objeto.

Abaixo um exemplo simplificado² de manipulação de *threads* em Java. Também é apresentado um diagrama mostrando como poderiam estar distribuídos no tempo a execução concorrente das *threads*.

```
class ThExemplo extends Thread {
    public ThExemplo() { ... }
    public void run() { Implementação do serviço }
}
...
public void static main( String args[] ) {
    ThExemplo a = new ThExemplo(),
                b = new ThExemplo();
    a.start();
    b.start();
    ...
    try
        a.join();
        catch( Exception e ) ...
    try
        b.join();
        catch( Exception e ) ...
}
```

O controle de execução de seções críticas em Java é garantido por um mecanismo de monitores. Este mecanismo garante que determinada seção de código será executada por apenas uma *thread* em um determinado instante de tempo. Caso uma *thread* deseje executar algum serviço do monitor, o primeiro passo é tentar obter a permissão

¹ Muitos dos termos utilizados nesta seção referem-se a terminologia adotada em orientação a objetos. Maiores detalhes em [2] ou mesmo em [1].

² Todo tratamento de exceções foram suprimidos nos exemplos de programas em Java apresentados.

de execução: caso obtenha a permissão, é garantido que o monitor não está sendo utilizado por nenhuma outra *thread*. Caso a permissão seja negada, é sinal que uma outra *thread* está sendo servida pelo monitor; neste caso a *thread* requisitante permanece bloqueada aguardando que o monitor seja liberado. Caso duas ou mais *threads* estejam bloqueadas aguardando o acesso ao monitor, no momento em que o monitor for liberado, uma das *threads* será escalonada a executar, enquanto a outra continuará aguardando. A regra mais comum é de permitir acesso ao monitor à *thread* que está aguardando a mais tempo, contudo esta regra não deve nunca ser considerada para provar a correção de um programa. A sincronização pode ser aplicada a diferentes níveis de granularidade: objeto, classe e bloco.

É de se notar que um monitor tem uma estrutura bastante próxima a um objeto: uma entidade que encapsula dados e fornece métodos para acessar estes dados. O mecanismo de monitores em Java permite garantir que trechos de código internos aos objetos sejam executados por apenas uma *thread*. Os mecanismos de sincronização entre *threads* são discutidos a seguir.

Sincronização em nível de objeto

A sincronização a nível de objeto garante que apenas uma *thread* possa executar métodos dentro do escopo de um objeto.

Esta forma de sincronização faz uso (implícito) do ponteiro `this`, permitindo que o objeto sirva uma única *thread* por vez. Seu uso é bastante intuitivo, como mostra o exemplo abaixo:

```
class Buffer {
    private Pilha p;
    public Buffer() { p = new Pilha(); }
    synchronized public void Insere( Object _o ) {
        p.Push( _o );
    }
    synchronized public Object Retira() {
        return p.Pop();
    }
}
```

A classe `Buffer` representa uma área de memória compartilhada entre duas *threads*; como no caso de um *buffer* para um algoritmo do tipo produtor/consumidor. `Insere` e `Retira` consistem em métodos que atuam sobre uma memória compartilhada, portanto seções críticas. A garantia de que somente uma *thread* vai executar o código correspondente a um destes métodos de um determinado objeto `Buffer` que vir a ser criado é dada pelo modificador `synchronized`. Método que possuam este modificador garantem a interface de acesso aos serviços do objeto, executados no escopo do objeto – em outras palavras, se existirem dois objetos `Buffer` criados em um programa, ambos poderão executar ao mesmo tempo seus respectivos métodos `Insere` e `Retira`, mas um mesmo objeto que esteja sendo utilizado ao mesmo tempo por duas *threads* não poderá ter os métodos `Insere` e `Retira` executando mesmo tempo.

Sincronização em nível de classe

A sincronização a nível de classe garante que apenas uma *thread* possa executar métodos dentro do escopo de uma classe.

Uma classe pode possuir atributos de classe, ou seja, membros cujo escopo seja a classe, necessitando portanto de um controle de sincronização a nível de classe. Para garantir a exclusividade de execução de *threads* no escopo de uma classe, faz-se uso de métodos `static`. Seu uso remete ao exemplo anterior, como mostra o exemplo abaixo:

```
class Buffer {
    private static Pilha p;
    initialize { p = new Pilha(); }
    public Buffer() {}
    synchronized static public void Insere( Object _o ) {
        p.Push( _o );
    }
    synchronized static public Object Retira() {
        return p.Pop();
    }
}
```

Neste exemplo, a classe `Buffer` define que a área de dados a ser compartilhada entre os produtores e os consumidores é um atributo de classe, o que significa que todos produtores e consumidores compartilharão a mesma área de dados. Desta forma o escopo deste atributo é a classe, e, portanto, não pode ter seu acesso protegido no escopo de um objeto, mas deve ser protegido considerando a execução de todos os produtores e consumidores que venham a ser construídos.

Sincronização em nível de bloco

A sincronização a nível de bloco garante que apenas uma *thread* possa executar o conjunto de instruções definidas dentro do escopo de um bloco. Para controlar a execução de uma seção crítica definida por um bloco, é utilizado um mecanismo de sincronização apoiado em um objeto auxiliar. Este objeto auxiliar é necessário pois, ao contrário dos mecanismos descritos anteriormente, não existe um ponto de apoio à sincronização. Este objeto auxiliar pode ser um objeto qualquer, mas a garantia de correção semântica é de responsabilidade do usuário que deve utilizar sempre o mesmo objeto para blocos que compartilhem um mesmo dado. O único cuidado é que este objeto não seja `null` e deve ser evitado seu uso dentro da seção crítica. Seu pode ser exemplificado da seguinte forma:

```

class Buffer {
    private Object mon;
    private Pilha p;
    public Buffer() {
        p = new Pilha();
        mon = new Object();
    }
    public void Insere( Object _o ) {
        synchronized( mon ) { p.Push( _o ); }
    }
    synchronized public Object Retira() {
        Object aux;
        synchronized( mon ) { aux = p.Pop(); }
        return aux;
    }
}

```

Neste exemplo, retornamos à primeira implementação dada à classe **Buffer**; observe que a presente implementação oferece um resultado semântico idêntico ao proposto anteriormente. O objeto que está sendo utilizado para garantir o acesso à seção crítica é um objeto privado (**mon**), especialmente criado para controlar acesso os blocos sincronizados. É bastante comum encontrar implementações deste mecanismo de sincronização que fazem uso da referência **this**. No entanto, por medida de segurança é indicada a utilização de um atributo privado do objeto. Observe que caso duas *threads* tentem executar os serviços **Insere** ou **Retira**, será permitido: o monitor é aplicado somente no momento em que iniciar a execução da seção crítica propriamente dita: o acesso ao *buffer*. Este mecanismo é especialmente interessante no caso em que os trechos de seções críticas são pequenos, não justificando a sincronização efetuada no método.

Bibliografia

1. H. M. Deitel and P. J. Deitel. *Java: como programar*. Bookman, Porto Alegre, 2000.
2. Robert W. Sebesta. *Conceitos de linguagens de programação*. Bookman, Porto Alegre, 4. ed., 2000.
3. A. Silberschatz and P. Galvin. *Operating systems concepts*. John Wiley & Sons, New York, 5 edition, 1997.
4. Andrew S. Tanenbaum. *Modern operating systems*. Prentice Hall, New Jersey, 1992.
5. Barry Wilkinson and Michael Allen. *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice-Hall, Upper Saddle River, 1999.