VOLTAR

Problemas de Concorrência e Compartilhamento

Apresentar os problemas de concorrência, compartilhamento de recursos e uma proposta em Java para solução de parte dos problemas

NESTE TÓPICO

Como vimos no tópico **Conceitos e Metas dos Sistemas Distribuídos**, a concorrência é um sintoma natural quando decidimos implementar um projeto de distribuição do sistema. Isto é, a concorrência se dá quando compartilhamos algum recurso onde diversos usuários passam a acessar o mesmo simultaneamente ou concorrentemente. A concorrência não gerenciada pela infraestrutura e pelo software ao recurso compartilhado pode gerar diversos problemas no sistema, a saber:

- Queda no desempenho do sistema (tempo de resposta)
- Indisponibilidade do sistema
- Indisponibilidade de um ou mais recursos compartilhados
- *Deadlock*

Assim, a concorrência não pode ser ignorada e deve ser considerada nos requisitos de qualquer projeto de um sistema distribuído. Desta forma, o projeto deve analisar as possibilidades de arquitetura de hardware, rede, software e modelo de desenvolvimento de software para lidar com o problema.

No tópico **Tipo de Arquiteturas**, vimos algumas soluções de arquiteturas para a distribuição do sistema e para lidar com o problema da concorrência. Neste sentido, podemos lembrar das arquiteturas voltadas para melhorar o

desempenho do sistema com objetivo de reduzir a concorrência, tais como as arquiteturas em cluster e grade. Mas a arquitetura de hardware não resolve todos os problemas, então vimos no tópico **Threads em JAVA e o Ambiente Multithreading** como desenvolver aplicações em Java utilizando os threads para o processamento de parte do código paralelamente, principalmente quando temos mais de um processador disponível.

Um tema interessante que vemos na disciplina de **Sistemas Operacionais** é a sincronização de processos. Assim, os diversos processos residentes na memória principal somente podem executar cada um no seu tempo. Mas como organizar a execução dos processos? A resposta está no sincronismo dos processos que acontece a partir de um escalonador (*scheduler*) que reside no kernel do sistema operacional. O escalonador adota alguns critérios ou algoritmos de escalonamento, tais como o FIFO (First In First Out, LIFO (Last In First Out), Prioridade (Importância), Menor Ciclo (Menor tempo de CPU) e o Round Robin (Circular/Revezamento da CPU).

Os threads podem executar paralelamente em diversos processadores que compartilham um espaço de memória comum. O problema está na utilização desde espaço de memória, pois se não existe um controle para acesso, por exemplo, quem lê e quem grava ou quem faz uma coisa e depois outra.

O seguinte exemplo demonstrado pela Figura 1, vemos um problema crônico denominado por Deadlock, ou seja, quando dois threads desejam acessar os recursos que estão presos por outro thread.

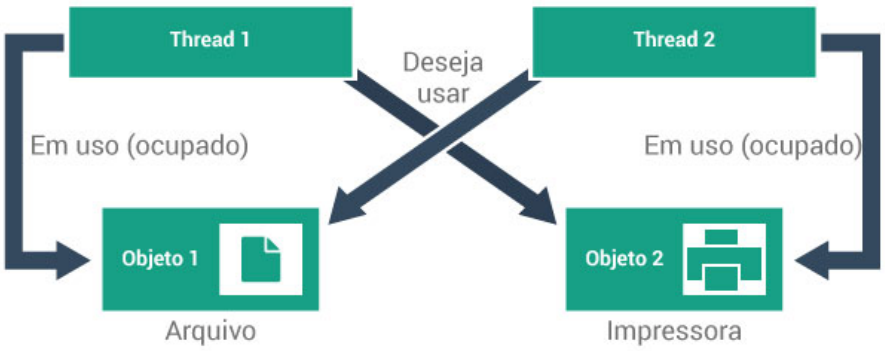

Figura 1 - Threads concorrentes e o problema do Deadlock

O outro exemplo demonstrado na Figura 2 mostra os três threads concorrendo à mesma área de memória compartilhada. Assim, considerando que a execução dos threads não é sincronizada, eles concorrem ao mesmo recurso e os valores apresentados vai depender de qual thread executar primeiro e assim sucessivamente.

Vamos analisar os seguintes cenários representados pelas Tabelas 1, 2 e 3.

Tabela 1 - Ordem de Execução dos Threads - Cenário 1

Ordem de execução	Método	Resultado
Thread 1	Ler()	José
Thread 2	Atualizar(Nome)	De José para Maria

Thread 3	Ler()	José

Nesta ordem de execução dos threads, teremos um usuário visualizando o nome José e outro Maria, pois entre uma leitura e outra, ocorreu uma atualização do nome. O primeiro usuário terá lido um nome desatualizado, pois o nome já foi atualizado por outro thread e eles executaram simultaneamente.

Tabela 2 - Ordem de Execução dos Threads - Cenário 2

Ordem de execução	Método	Resultado
Thread 2	Atualizar(Nome)	De José para Maria
Thread 3	Ler()	José
Thread 1	Ler()	Maria

Nesta ordem de execução dos threads, um usuário irá visualizar o nome José e o outro Maria, pois o thread 3 executou paralelamente ao thread 2. O problema é que para um usuário aparecerá o nome José e para outro Maria. O problema é que na percepção dos usuários o sistema trará informações diferentes mesmo executando as consultas simultaneamente.

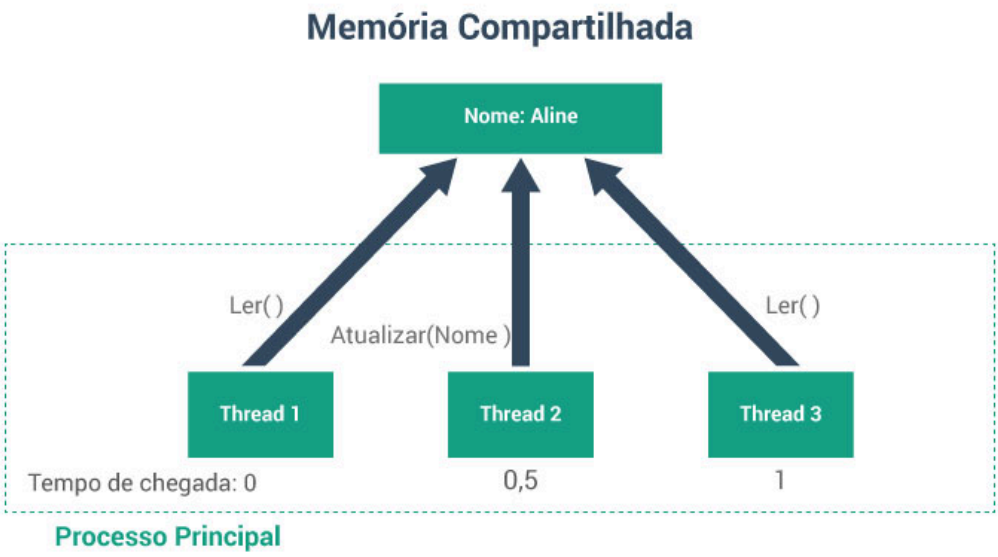

Figura 2 - Concorrência da memória principal e a falta de sincronismo entre os threads

Para evitar os resultados diferentes dependendo da ordem de execução dos threads é que se faz necessária a sincronização. Diante dos cenários apresentados, veremos o problema de concorrência e seus impactos com um exemplo simples em Java. Logo em seguida, com outro exemplo em Java, veremos como fazer a sincronização dos threads.

O Conceito das classes Produtor e Consumidor

O seguinte código Java mostra na saída os threads em execução e a variação dos resultados dependendo de qual executa primeiro, pois neste exemplo os threads não estão sincronizados. Em outras palavras, a execução dos threads deve ser coordenada de maneira que, enquanto um thread está em execução o outro aguarda e vice-versa.

Observe as classes Produtor e Consumidor. Estas classes foram implementadas para que possam instanciar objetos colaborativos. Isto é, a classe Produtor produz dados e a classe Consumidor faz a leitura dos dados de tal maneira que operam em conjunto na geração e leitura de dados.

Vamos entender os exemplos a seguir.

A classe Valores

A classe Valores foi criada com a estrutura de atributos e métodos para que os objetos instanciados (threads produtoras e consumidoras) possam armazenar (método guardar()) e apresentar (método exibir()) os valores em tempo de execução.

```
1.  public class Valores {
2.      int valor;
3.
4.      public int exibir() {
5.          return this.valor;
6.      }
7.      public void guardar(int valor) {
8.          this.valor = valor;
9.      }
10. }
```

A classe Produtor

A classe Produtor foi criada para executar a inserção e atualização de dados no atributo valor e implementa a interface *Runnable*. A variável tempo receberá um valor randômico passado pelo método *random*(). Esse tempo é utilizado na linha 16 para colocar o thread para dormir enquanto aguarda o objeto Consumidor executar.

O método run() executará a tarefa designada ao thread Produtor. Assim, como estamos trabalhando com valores de 0 a 10 que são exibidos na saída, a instrução *for* na linha 11 está controlando o *loop* (laço). O método guardar(i) na linha 13, atribuirá um inteiro de 0 a 10 em cada passagem do laço ao atributo valor da classe Valores.

```
1.  public class Produtor implements java.lang.Runnable{
2.      Valores valor;
3.
4.      public Produtor(Valores valor) {
5.          this.valor = valor;
6.      }
7.
8.      public void run() {
9.          int tempo;
10.         int i;
11.         for(i = 0;i <= 10;i++) {
12.             tempo  = (int)(Math.random() * 3000);
13.             valor.guardar(i);
14.             System.out.println ("O Produtor está guardando o valor: \t" + i);
15.             try {
16.                 Thread.sleep(tempo);
17.             } catch (InterruptedException e) {
18.             }
19.         }
20.     }
21. }
```

A classe Consumidor

A classe Consumidor foi criada para os dados passados ao atributo valor e implementa a interface Runnable. A variável tempo receberá um valor randômico passado pelo método random() na linha 10. Esse tempo é utilizado na linha 15 para colocar o thread para dormir enquanto aguarda o objeto Produtor executar. O método run() executará a tarefa designada ao thread Consumidor. Assim, como estamos trabalhando com valores de 0 a 10 que são exibidos na saída, a instrução ***for*** na linha 9 está controlando o loop (laço). O método exibir(i) na linha 12, fará a leitura do inteiro de 0 a 10 em cada passagem do laço ao atributo valor da classe Valores.

```
1.  public Consumidor(Valores valor) {
2.
3.          this.valor = valor;
4.  }
5.
6.
7.      public void run() {
8.          int tempo;
9.          for(int i = 0;i <= 10;i++) {
10.             tempo  = (int)(Math.random() * 3000);
11.             System.out.println("O Consumidor está lendo o valor: \t"
12.                     + valor.exibir());
13.
14.             try {
15.                 Thread.sleep(tempo);
16.             }catch (InterruptedException e) {
17.             }
18.         }
19.     }
20. }
```

Classe Processo

A classe Processo possui o método main (principal) responsável pela execução do projeto. Na linha 3, o objeto valor é instanciado e passado como parâmetro nos construtores das classes Produtor e Consumidor (linhas 7 e

8). Nas linhas 7 e 8, os threads Produtor e Consumidor são instanciados e despachados para execução.

```
1.  public class Processo {
2.      public static void main(String args[]) {
3.          Valores valor = new Valores();
4.
5.          System.out.println("Iniciando os Threads Produtor e Consumidor");
6.          System.out.println("----------------------------------------");
7.          new Thread(new Produtor(valor)).start();
8.          new Thread(new Consumidor(valor)).start();
9.      }
10. }
```

Saída do Programa: Produtor e Consumidor sem Sicronismo

Observe os casos em negrito que, devido a falta de sincronismo dos threads, o Produtor e o Consumidor executam duas vezes seguintes nestes casos. O impacto deste comportamento é que o thread Consumidor deixa de ler um ou mais valores atualizados (valores 2 e 10), pois fez a leitura do mesmo valor duas vezes (valores 5 e 7).

Iniciando as Threads Produtor e Consumidor

O Produtor está guardando o valor: 0
O Consumidor está lendo o valor: 0
O Produtor está guardando o valor: 1
O Consumidor está lendo o valor: 1
O Produtor está guardando o valor: 2
O Produtor está guardando o valor: 3
O Consumidor está lendo o valor: 3
O Produtor está guardando o valor: 4
O Consumidor está lendo o valor: 4
O Produtor está guardando o valor: 5
O Consumidor está lendo o valor: 5
O Consumidor está lendo o valor: 5
O Produtor está guardando o valor: 6
O Consumidor está lendo o valor: 6
O Produtor está guardando o valor: 7
O Consumidor está lendo o valor: 7
O Consumidor está lendo o valor: 7
O Produtor está guardando o valor: 8
O Consumidor está lendo o valor: 8
O Produtor está guardando o valor: 9
O Consumidor está lendo o valor: 9
O Produtor está guardando o valor: 10

Sincronizando os Threads Produtor e Consumidor

A proposta de solução de sincronismo apresentada a seguir mantém as três classes (Valores, Produtor e Consumidor) que vimos no exemplo não sincronizado acima.

Classe Valores

A classe Valores sofreu algumas alterações em relação a versão anterior, pois foi considerada a questão do sincronismo entre os threads Produtor e Consumidor. Assim, foi criado um atributo booleano denominado por bloqueado (linha 3) e, dependendo da ocasião, ele receberá os valores verdadeiro ou falso/*true* ou *false* (linhas 6, 15 e 26). A proposta do booleano é controlar o acesso ao *While* que foi programado para colocar os threads Produtor e Consumidor em espera (linhas 11 e 23).

Outro ponto crucial é o uso de monitores ou inibidores que utilizam tokens e permitem que uma thread venha a adquirir como controle na execução. Neste caso, foram implementados os métodos guardar e exibir com o segmento de código definido pela declaração *synchronized* (linhas 8 e 19). Desta forma, declaramos que os métodos tratam-se de uma região crítica e que não pode ser acessado concorrentemente por dois ou mais threads.

A sincronização que foi implementada fez o uso de eventos denominados pelos métodos *wait*() e *notify*(). O método *wait*() poduz eventos para aguardar e o *notify*() eventos para a notificação para parar de aguardar. Uma referência destes métodos é a utilização de Semáforos na linguagem C, onde controla o acesso ao recurso pelos bits zero e um. Desta forma, temos:

- No método **guardar**: *wait*() na linha 11 e *notify*() na linha 16.
- No método **exibir**: *wait*() na linha 23 e *notify*() na linha 27.

```
1.  public class Valores {
2.      int valor;
3.      private boolean bloqueado;
4.
5.      public Valores(){
6.          bloqueado = false;
7.      }
8.      public synchronized void guardar(int valores) {
9.          while (bloqueado)
10.             try{
11.                 wait();
12.             }catch (InterruptedException e){
13.              }
14.         this.valor = valores;
15.         bloqueado = true;
16.         notify();
17.     }
18.
19.     public synchronized int exibir() {
20.
21.         while (!bloqueado)
22.             try{
23.                 wait();
24.             }catch (InterruptedException e){
25.              }
26.             bloqueado = false;
27.             notify();
28.             return this.valor;
29.      }
30. }
```

Classe Produtor

A classe Produtor não sofreu alterações e considera as mesmas características apresentadas no código anterior.

```
public class Produtor implements java.lang.Runnable{
    Valores valor;

    public Produtor(Valores valor) {
        this.valor = valor;
    }

    public void run() {
        int tempo;
        int i;
        for(i = 0;i <= 10;i++) {
            tempo  = (int)(Math.random() * 3000);
            valor.guardar(i);
            System.out.println ("O Produtor está guardando o valor: \t" + i);
            try {
                Thread.sleep(tempo);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

Classe Consumidor

A classe Consumidor não sofreu alterações e considera as mesmas características apresentadas no código anterior.

```
public class Consumidor  implements java.lang.Runnable {

    Valores valor;

    public Consumidor(Valores valor) {

        this.valor = valor;
}


    public void run() {
        int tempo;
        for(int i = 0;i <= 10;i++) {
            tempo  = (int)(Math.random() * 3000);
            System.out.println("O Consumidor está lendo o valor: \t"
                    + valor.exibir());

            try {
                Thread.sleep(tempo);

            }catch (InterruptedException e) {
            }
        }
    }
}
```

Classe Processo

A classe Processo não sofreu alterações e considera as mesmas características apresentadas no código anterior.

```
1.  public class Processo {
2.      public static void main(String args[]) {
3.          Valores valor = new Valores();
4.
5.          System.out.println("Iniciando as Threads Produtor e Consumidor");
6.          System.out.println("----------------------------------------");
7.          new Thread(new Produtor(valor)).start();
8.          new Thread(new Consumidor(valor)).start();
9.      }
10. }
```

Saída do Programa: Produtor e Consumidor com Sicronismo

Observe que a saída do programa exibe o sincronismo dos threads Produtor e Consumidor. Não há casos onde o Produtor executa duas ou mais vezes seguidas, assim como acontece com o Consumidor.

Iniciando as Threads Produtor e Consumidor

O Produtor está guardando o valor: 0
O Consumidor está lendo o valor: 0
O Produtor está guardando o valor: 1
O Consumidor está lendo o valor: 1
O Produtor está guardando o valor: 2
O Consumidor está lendo o valor: 2
O Produtor está guardando o valor: 3
O Consumidor está lendo o valor: 3
O Produtor está guardando o valor: 4
O Consumidor está lendo o valor: 4
O Produtor está guardando o valor: 5
O Consumidor está lendo o valor: 5
O Produtor está guardando o valor: 6
O Consumidor está lendo o valor: 6
O Produtor está guardando o valor: 7
O Consumidor está lendo o valor: 7
O Produtor está guardando o valor: 8
O Consumidor está lendo o valor: 8
O Produtor está guardando o valor: 9
O Consumidor está lendo o valor: 9
O Produtor está guardando o valor: 10
O Consumidor está lendo o valor: 10

Quiz

Exercício

Problemas de Concorrência e Compartilhamento

INICIAR

Referências

TANENBAUM, A. S.; STEEN, M. V.; Sistemas Distribuídos; Princípios e Paradigmas; Pearson Prentice Hall; 2ª edição; 2007.

TANENBAUM, A.S, Sistemas Operacionais Modernos, 3ª edição, Pearson Education do Brasil, 2010.

DA COSTA, Celso Maciel; STRINGHINI, Denise; CAVALHEIRO, Gerson Geraldo Homrich. Programação Concorrente: Threads, MPI e PVM. Escola Regional de Alto Desempenho, II ERAD, captulo, v. 2, 2002.

SILBERSCHATZ, Abraham. Sistemas operacionais com Java. Elsevier Brasil, 2008.

Avalie este tópico

ANTERIOR
Threads em JAVA e o Ambiente Multithreading

Índice

PRÓXIMO
Modelo Cliente/Servidor e Sockets em Java

Biblioteca (https://www.uninove.br/conheca-a-uninove/biblioteca/sobre-a-biblioteca/apresentacao/)
Portal Uninove (http://www.uninove.br)

Ajuda? (https://ava.un idCurso=)

Mapa do Site