DCC205 – PROGRAMAÇÃO ORIENTADA A OBJETOS

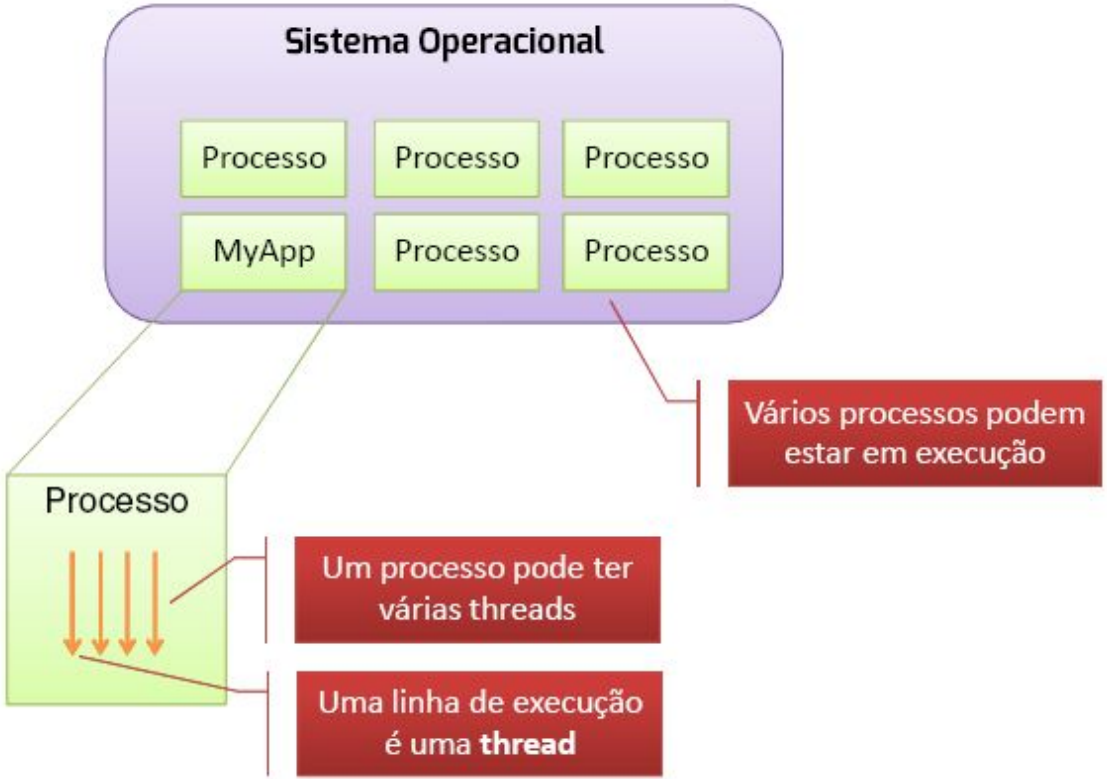
Aula 11 – Programação MultiThread e Sincronismo

Carlos Bruno Oliveira Lopes carlosbrunocb@gmail.com

Threads

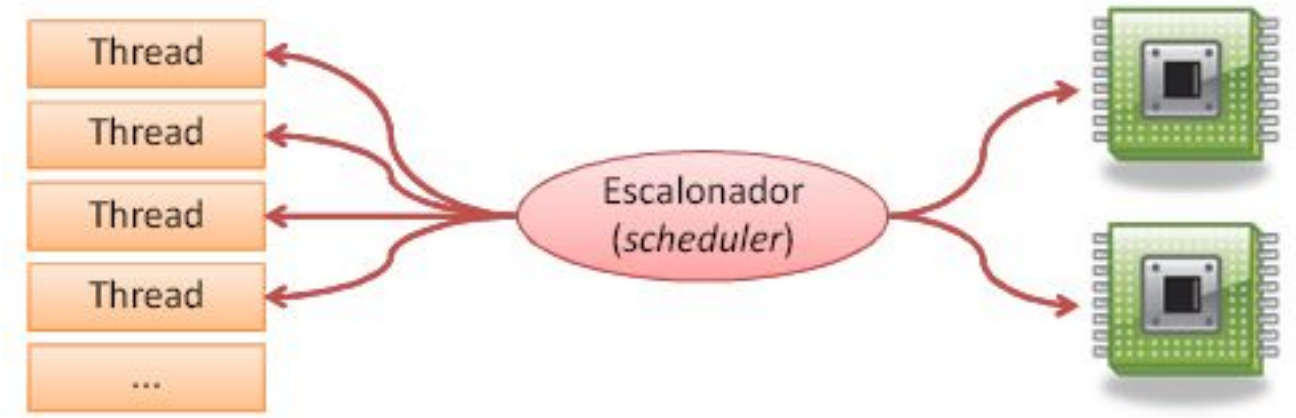
- Em várias situações, precisamos "rodar duas coisas ao mesmo tempo".
- Imagine um programa que gera um relatório muito grande em PDF. É um processo demorado e, para dar alguma satisfação para o usuário, queremos mostrar uma barra de progresso.
- Queremos então gerar o PDF e ao mesmo tempo atualizar a barrinha.
- No mesmo programa (um processo), se queremos executar coisas em paralelo, normalmente usamos Threads.
- As threads de um processo compartilham o heap do processo
    - Área de memória onde ficam armazenados os objetos

Threads


Sistema Operacional
Processo
Processo
Processo
MyApp
Processo
Processo
Vários processos podem estar em execução
Processo
Um processo pode ter várias threads
Uma linha de execução é uma **thread**

Escalonamento de Threads

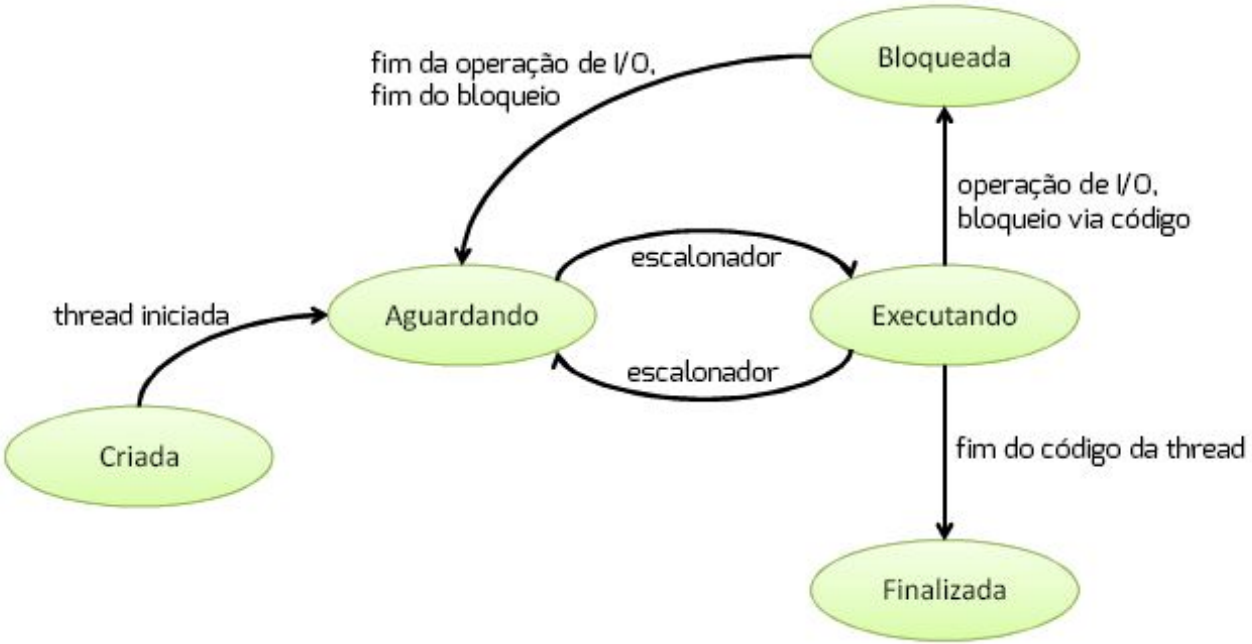
Um núcleo de um processador só pode executar uma tarefa por vez.



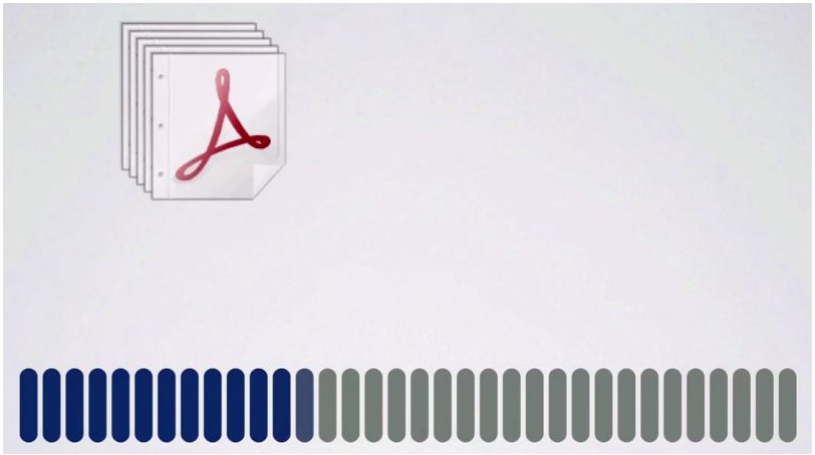
Na presença de múltiplos processadores ou processadores multi-core, é possível a execução verdadeiramente simultânea.

Escalonamento de Threads

Estados de Uma Thread



Gerando o PDF



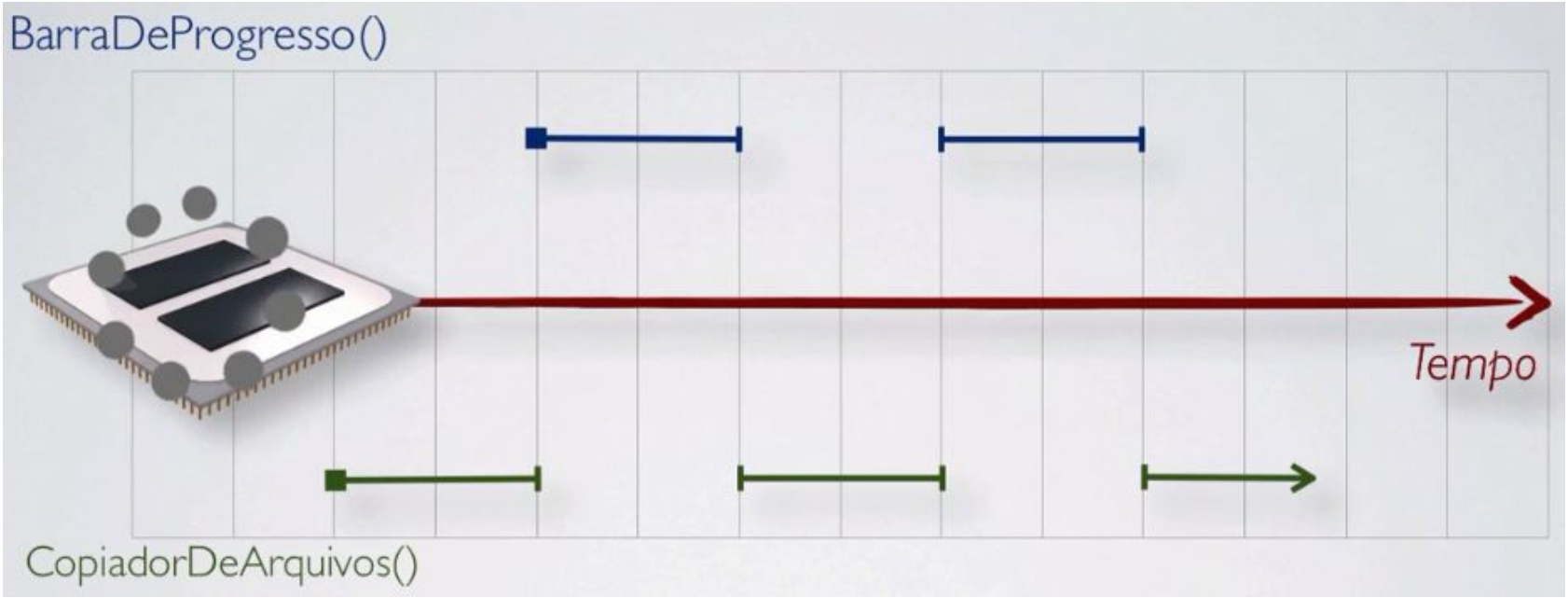
Threads em Java

- Em Java, usamos a classe Thread do pacote java.lang para criarmos linhas de execução paralelas;
- A classe **Thread** recebe como argumento um **objeto com o código que desejamos rodar**. Por exemplo, no programa de PDF e barra de progresso:

```
public class GeraPDF {
    public void rodar () {
        // lógica para gerar o pdf...
    }
}
```

```
public class BarraDeProgresso {
    public void rodar () {
        // mostra barra de progresso e vai atualizando ela...
    }
}
```

Threads em Java



No método main

- O código abaixo não compilará. Por que? O que esta faltando?
- Como a classe Thread sabe que deve chamar o método roda?
- Como ela sabe que nome de método daremos e que ela deve chamar esse método especial?
- Falta na verdade um **contrato** entre as nossas classes a serem executadas e a classe Thread.

```
GeraPDF gerapdf = new GeraPDF();
Thread threadDoPdf = new Thread(gerapdf);
threadDoPdf.start();

BarraDeProgresso barraDeProgresso = new BarraDeProgresso();
Thread threadDaBarra = new Thread(barraDeProgresso);
threadDaBarra.start();
```

Interface Runnable

- Na interface **Runnable**, há apenas um método chamado **run()**.
- Basta implementá-lo, "assinar" o contrato e a classe Thread já saberá executar nossa classe.

```
public class GeraPDF implements Runnable {
    public void run () {
        // lógica para gerar o pdf...
    }
}
```

```
public class BarraDeProgresso implements Runnable {
    public void run () {
        // mostra barra de progresso e vai atualizando ela...
    }
}
```

Threads em Java

- A classe Thread recebe no construtor um objeto que é um **Runnable**, e seu método **start** chama o método **run()** da nossa classe;
- A classe Thread não sabe qual é o tipo específico da nossa classe; para ela, basta saber que a classe segue o contrato estabelecido e possui o método **run()**;
- Quando o método **run()** termina, a thread também termina;

Estendendo a classe Thread

- A classe Thread implementa **Runnable**;
- Então, você pode criar uma subclasse dela e reescrever o run que, na classe Thread, não faz nada:

```
public class GeraPDF extends Thread {
    public void run () {
        // ...
    }
}
```

- E, como nossa classe é uma Thread, podemos usar o start diretamente:

```
GeraPDF gera = new GeraPDF();
gera.start();
```

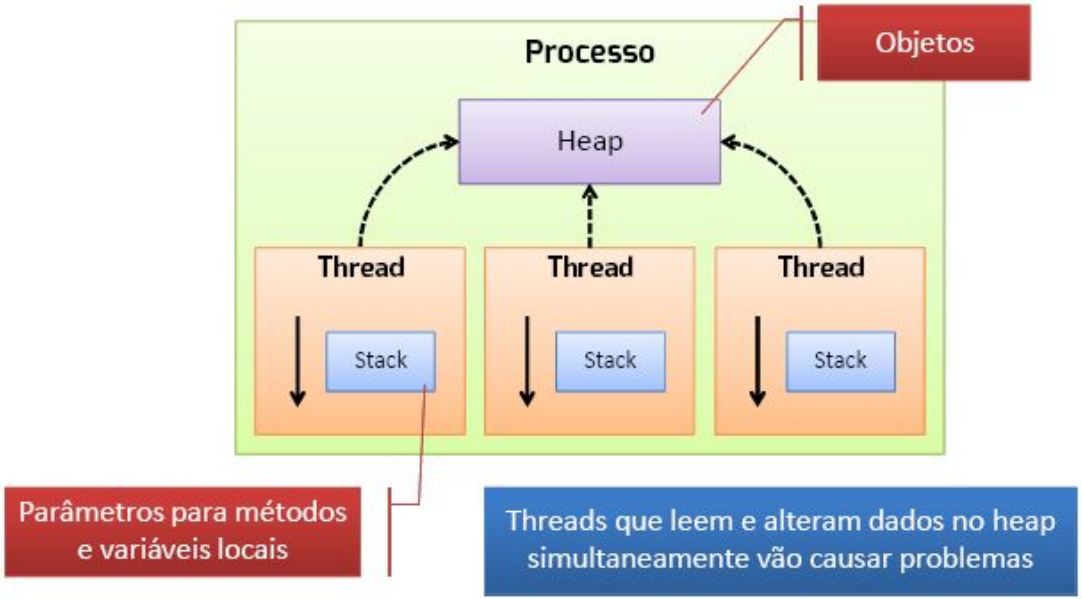
Apesar de ser um código mais simples, estamos usando herança apenas por "preguiça" (herdamos um monte de métodos mas usamos apenas o run), e não por polimorfismo, que seria a grande vantagem. Prefira implementar Runnable a herdar de Thread.

Usando Classes Anônimas

```
public static void main(String[] args) {
    Thread t1 = new Thread(new Runnable() {
        //Thread Processando
        @Override
        public void run() {
            for (int i=0; i<10;i++) {
                System.out.println("Processando");
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    });
    Thread t2 = new Thread(new Runnable() {
        //Thread Copiando
        @Override
        public void run() {
            for (int i=0; i<10;i++) {
                System.out.println("Copiando");
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    });
    t1.start();
    t2.start();
}
```

Prioridades em Threads

- Threads podem ter prioridades
  - Não existe garantia de que as prioridades serão seguidas;
- Constantes
  - MIN_PRIORITY (1)
  - NORM_PRIORITY (5)
  - MAX_PRIORITY (10)
- O método yield() pode ajudar outras threads de mesma prioridade a executar



Compartilhamento de Dados

Threads compartilham o heap do processo

Sincronização de Threads

- Muitas vezes é necessário que várias threads acessem o mesmo objeto
  - Race Conditions
- Sincronizar as threads é necessário para evitar acesso simultâneo às regiões críticas (critical sections)

Sincronização de Threads

- A sincronização de threads em Java é feita através do uso de monitores
- Monitor é um objeto Java qualquer que cuida de uma região crítica
- Só é permitida a execução de uma thread por vez
- As outras threads ficam aguardando

Usando o synchronized

- É possível sincronizar o método todo (método **synchronized**)

```
public synchronized void metodo() {
    //código sincronizado
}
```

- É possível sincronizar apenas um bloco (bloco **synchronized**)

```
synchronized(monitor) {
    //código sincronizado
}
```

Problemas com concorrência

- 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. No primeiro dia de cada mês, o Banco precisa atualizar o saldo de todas as Contas de acordo com uma **taxa específica**.
  - A atualização de milhões de contas é um processo demorado
  - É preciso executar as atualizações paralelamente às atividades, de depósitos e saques, normais do banco.
  - Estamos compartilhando objetos entre múltiplas threads (as contas, no nosso caso).

Problemas com concorrência

Imagine a seguinte possibilidade: no exato instante em que o atualizador está atualizando uma Conta X, o cliente dono desta Conta resolve efetuar um depósito.

```
public class Conta {

    private double saldo;

        // outros métodos e atributos...
        public void atualiza(double taxa) {
            double saldoAtualizado = this.saldo * (1 + taxa);
            this.saldo = saldoAtualizado;
        }

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

Dizemos que essa classe não é ::thread safe::,

Solução

- 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;
        }
    }
}
```

Thread utilizando o mesmo objeto Conta, o **this**. Esses métodos são mutuamente exclusivos e só executam de maneira atômica.

Exercício em Sala

- Crie uma classe FazDeposito que recebe uma Conta no construtor. Essa Conta será um atributo da classe FazDeposito. Além disso, a classe deve implementar a interface Runnable. Sobreescreva o método run e dentro dele faça um depósito na conta recebida pelo construtor de 100 reais.
- Crie uma classe App com o método main.
- Dentro de App crie uma instâncias de FazDeposito e passe para ela uma conta com 500 de saldo;
- Crie três Threads que rodem o mesmo objeto de FazDeposito na classe App;
- Imprima o saldo da conta passada para a classe FazDeposito.

Exercício em Sala

Fica assim:

```java
public class App {

    public static void main(String[] args) {
        Conta conta = new Conta (500);
        FazDeposito acao = new FazDeposito(conta);

        Thread t1 = new Thread(acao);
        Thread t2 = new Thread(acao);
        Thread t3 = new Thread(acao);

        t1.start();
        t2.start();
        t3.start();

        System.out.println(conta.getSaldo());
    }

}
```

```java
public class FazDeposito implements Runnable{

    private Conta conta;

    public FazDeposito(Conta conta) {
        this.conta = conta;
    }

    @Override
    public void run() {
        conta.deposita(200);
    }

}
```

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.
- 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.

Exemplo - ArrayList

- Imagine que temos um objeto que guarda todas as mensagens que uma aplicação de chat recebeu.
- Vamos usar uma ArrayList 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 classe **ProduzMensagens** para adicionar as queries

```java
public class ProduzMensagens implements Runnable{

    private int inicio;
    private int fim;
    Collection<String> mensagens;

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

    public void run () {
        for (int i = inicio; 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.

```java
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 Finalizadas");

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

        if (mensagens.contains(null))
            throw new IllegalStateException("Não devia ter null aqui dentro");

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

}
```

Solução

Teste o código anterior usando synchronized ao adicionar na coleção:

```
public class ProduzMensagens implements Runnable{

    private int inicio;
    private int fim;
    Collection<String> mensagens;

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

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

}
```

Usando o Vector

- 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?

Comunicação Entre Threads

- Às vezes a atividade de uma thread depende da atividade de outra
  - Comunicação é necessária
- Métodos

Método	Descrição
wait()	Faz a thread esperar até que outra thread a notifique ou que determinado tempo tenha passado
notify()	Notifica uma thread que está aguardando
notifyAll()	Notifica todas as threads que estão aguardando (uma delas acessa a região crítica e as outras voltam a esperar)

Problemas de Sincronização

- *Starvation* (Inanição)
  - *Starvation* descreve uma situação em que um thread não consegue obter acesso regular a recursos compartilhados e não consegue fazer progresso. Isso acontece quando recursos compartilhados ficam indisponíveis por longos períodos por threads "gananciosos". Por exemplo, suponha que um objeto forneça um método sincronizado que geralmente leva muito tempo para retornar. Se um thread invocar este método frequentemente, outros threads que também precisam de acesso sincronizado frequente ao mesmo objeto serão frequentemente bloqueados.
- *Deadlock* (Impasse)
  - Duas ou mais threads estão paradas aguardando por algo que nunca vai acontecer;
  - Travamento do sistema;

Problemas Clássicos

- “Produtor e Consumidor”
- “Leitor e Escritor”
- “Jantar dos Filósofos”
- “Barbeiro Adormecido”

Considerações Finais

- Quando o assunto é thread, muito pouco é garantido
- Cuidado com o sincronismo dos dados
- Encontrar problemas de sincronismo é bastante difícil
- Quando for programar multithread, tome bastante cuidado e saiba o que você está fazendo
- Se você quer algo executado em sequência, então não use threads

Sugestão de Leitura

- http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
- https://www.w3schools.com/java/java_threads.asp

Referências Bibliográficas

- DEITEL, Harvey M. e DEITEL, Paul J. Java - Como Programar, 8ª edição. Pearson. 2010.
- BLOCH, Joshua. Effective Java, 2ª edição. Addison-Wesley, 2008.
- CAELUM. Java e Orientação a Objetos. Disponível em: https://www.caelum.com.br/apostila-java-orientacao-objetos/
- SOFTBLUE. Professor Carlos Eduardo Gusso Tosin. Fundamentos de Java. http://www.softblue.com.br/.
- K19. Java e Orientação a Objetos. Disponível em: http://www.k19.com.br/cursos/orientacao-a-objetos-em-java.
- HORSTMANN, CORNELL. Core Java Volume I – Fundamentos, 8º Edição. São Paulo, Pearson Education, 2010.
- BRAUDE, E. J. Projeto de software - da programação à arquitetura: uma abordagem baseada em Java. Porto Alegre: Bookman, 2005.
- SANTOS, R. Introdução à Programação Orientada a Objetos usando Java. São Paulo: Campus, 2003.
- Slides do Professor Doutor Horácio Fernandes da UFAM.