

Threads e Sockets

Sistemas Distribuídos e Mobile

Prof. Me. Gustavo Torres Custódio

gustavo.custodio@anhembi.br

Introdução

Threads

Sockets

Exercícios



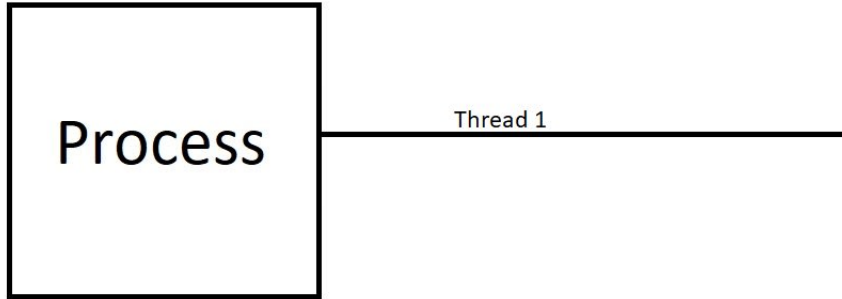
Threads e Sockets

Threads

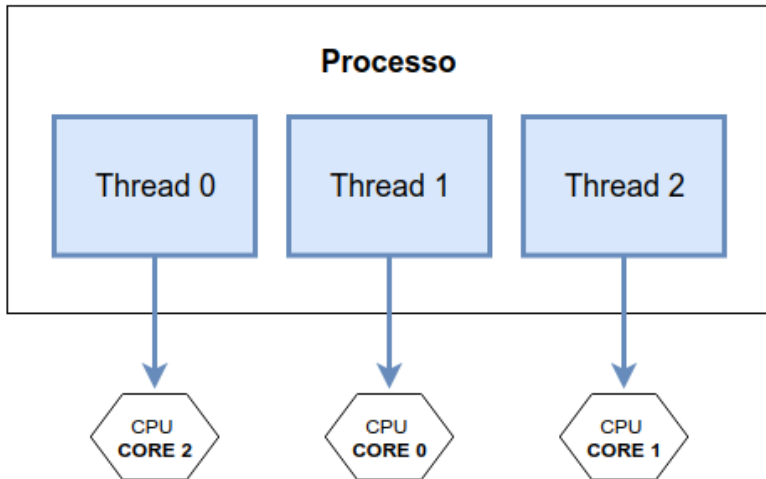
Threads

- Threads **são divisões de um processo** do sistema operacional.
 - Um programa comum possui uma única thread.
 - Criar threads é menos custoso do que a criação de processos.
 - Utilizar threads permite separar a execução de um programa em múltiplos núcleos de processador.

Threads



Threads



Threads

- `run()`: é o método que executa as atividades de uma thread. Quando este método finaliza, a thread também termina.
- `start()`: método que dispara a execução de uma thread. Este método chama o método `run()` antes de terminar.
- `sleep(int x)`: método que coloca a thread para dormir por x milisegundos.

Threads

- `join()`: método que espera o término da thread para qual foi enviada a mensagem para ser liberada.
- `interrupt()`: método que interrompe a execução de uma thread.
- `interrupted()`: método que testa se uma thread está ou não interrompida.

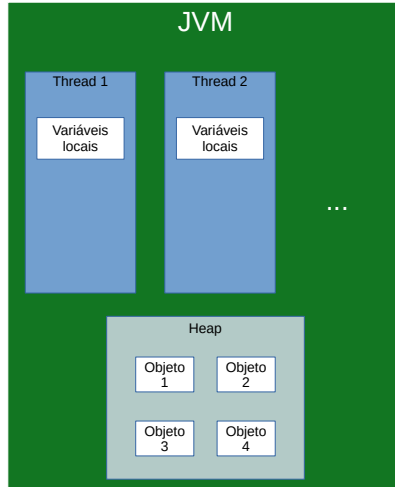
Thread

- Estados de uma Thread:
 - Nova - a thread ainda não foi iniciada.
 - Executando - a thread sendo executada na JVM está nesse estado.
 - Bloqueada - a thread está bloqueada esperando outra thread sair de uma região crítica.
 - Esperando - a thread está esperando outra thread executar uma ação.
 - Dormindo - a thread fica inativa por um intervalo de tempo.
 - Morta - a thread foi executada e encerrada.

Threads em Java

- Threads dividem o mesmo *heap*, onde os objetos instanciados são armazenados.
 - Threads diferentes podem acessar o mesmo objeto.
- Cada thread possui seu próprio conjunto de variáveis e pilha de execução.

Threads em Java



Threads em Java

- Em Java, threads são implementadas usando a classe **Thread** ou a interface **Runnable**
 - `class` NomeClasse `extends` Thread;
 - `class` NomeClasse `implements` Runnable.
 - Recomenda-se utilizar o Runnable porque, dessa forma, a classe fica livre para herdar de outra classe.

Threads em Java

- Esqueleto de uma classe com a interface Runnable:

```
public class Classe implements Runnable {  
    // restante do código  
  
    public void run() {  
        // Código executado depois do start ser usado  
    }  
}  
  
Classe c = new Classe();  
Thread t = new Thread(c);  
// Iniciando uma thread  
t.start();
```

Exemplo

```
public class ParImpar implements Runnable {  
    int numeroThread;  
  
    public static void main(String[] args) {  
        Thread pi1 = new Thread(new ParImpar(1));  
        Thread pi2 = new Thread(new ParImpar(2));  
        Thread pi3 = new Thread(new ParImpar(3));  
        Thread pi4 = new Thread(new ParImpar(4));  
        pi1.start();  
        pi2.start();  
        pi3.start();  
        pi4.start();  
    }  
}
```

Exemplo

```
public ParImpar(int numeroThread) {  
    this.numeroThread = numeroThread;  
}  
  
public void run() {  
    for (int i=0; i < 100; i++) {  
        int valor = this.numeroThread % 2;  
        if (valor == 1) {  
            System.out.println("Cara");  
        }else {  
            System.out.println("Coroa");  
        }  
    }  
}
```

Condições de Corrida

- É necessário tomar cuidado com threads quando trabalhamos com múltiplas threads simultaneamente em um mesmo recurso.
 - Mesmo arquivo, variável etc.
- Múltiplas Threads podem acessar o mesmo recurso simultaneamente, modificá-los e causar inconsistências.
 - Chamamos isso de condição de corrida.

Exemplo

- Suponha o seguinte código que trabalha com múltiplas threads escrevendo em um arquivo ao mesmo tempo:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class Gerenciador {
    int contagem = 0;

    public void salvarNumeroSorteado(int numero) throws IOException {
        FileWriter fw = new FileWriter("resultado.txt", true);
        BufferedWriter bw = new BufferedWriter(fw);
        PrintWriter out = new PrintWriter(bw);
        out.println("\nSorteio número " + contagem + ":");
        out.println(numero);
        out.close();
        // Contagem de números sorteados
        contagem++;
    }
}
```

Exemplo

```
public class Sorteio implements Runnable{
    Gerenciador gerenciador;

    Sorteio(Gerenciador gerenciador) {
        this.gerenciador = gerenciador;
    }

    public static void main(String[] args) throws IOException{
        Gerenciador gerenciador = new Gerenciador();
        Thread s1 = new Thread(new Sorteio(gerenciador));
        Thread s2 = new Thread(new Sorteio(gerenciador));
        Thread s3 = new Thread(new Sorteio(gerenciador));
        Thread s4 = new Thread(new Sorteio(gerenciador));

        s1.start();
        s2.start();
        s3.start();
        s4.start();
    }
}
```

Exemplo

```
public void run(){
    int numeroSorteado;
    Random gerador = new Random();
    try {
        for (int i = 0; i < 100; i++) {
            System.out.println("Sorteando número");
            numeroSorteado = gerador.nextInt(60) + 1;
            gerenciador.salvarNumeroSorteado(numeroSorteado);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

- Cada uma das threads sorteia 100 números de 1 a 60 e salva em um arquivo.

Exemplo

- A contagem de números sorteados se torna inconsistente.

Sorteio número 0: 57

Sorteio número 0: 11

Sorteio número 2: 3

Sorteio número 3: 4

Sorteio número 3: 4

- Como resolver esse problema?

Synchronized

- A palavra chave **synchronized** do Java é responsável por garantir que múltiplas threads diferentes não acessem o mesmo recurso simultaneamente.
 - A palavra synchronized define **regiões críticas**,
 - ou seja, regiões do código que só podem ser acessadas por uma thread de cada vez.

Synchronized

- A palavra **synchronized** deve ser utilizada sempre no mesmo objeto.
 - O synchronized não vai afetar objetos diferentes.
 - No exemplo anterior, a palavra pode ser usada na classe Gerenciador.

Synchronized

```
public synchronized void salvarNumeroSorteado(int numero) throws IOException {  
    FileWriter fw = new FileWriter("resultado.txt", true);  
    BufferedWriter bw = new BufferedWriter(fw);  
    PrintWriter out = new PrintWriter(bw);  
    out.println("\nSorteio número " + contagem + ":");  
    out.println(numero);  
    out.close();  
    // Contagem de números sorteados  
    contagem++;  
}
```

Synchronized

- Adicionando o **synchronized**, garantimos que apenas uma thread possa acessar a região crítica por vez.
- Observe o resultado da contagem do arquivo depois dessa mudança.
- Quando temos um servidor em Java, é possível que múltiplas threads disputam um elemento compartilhado.
 - A região crítica deve ser destacada para evitar inconsistências.



Threads e Sockets

Sockets

Sockets

- Threads podem ser utilizadas em Sockets.
 - Um servidor pode, por exemplo, alocar uma thread para cada conexão de um cliente.
 - Isso permite múltiplos acessos simultaneamente e melhora o tempo de resposta do servidor.
 - Os problemas de recursos compartilhados entre múltiplas threads ainda são válidos nessa situação.
- Suponha um sistema cliente servidor de envio e recebimento de mensagens.

Sockets

- Código do Servidor:

```
import java.io.IOException; import java.net.ServerSocket;
import java.net.Socket; import java.util.Scanner;

public class Server implements Runnable{
    public Socket cliente;

    public Server(Socket cliente){
        this.cliente = cliente;
    }

    public static void main(String[] args) throws IOException{
        //Cria um socket na porta 12345
        ServerSocket servidor = new ServerSocket (12345);
        System.out.println("Porta 12345 aberta!");

        // Aguarda alguém se conectar. A execução do servidor
        // fica bloqueada na chamada do método accept da classe
        System.out.println("Aguardando conexão do cliente...");

        while (true) {
            Socket cliente = servidor.accept();
            // Cria uma thread do servidor para tratar a conexão
            Server tratamento = new Server(cliente);
            Thread t = new Thread(tratamento);
            // Inicia a thread para o cliente conectado
            t.start();
        }
    }
}
```

Sockets

- Código do Servidor:

```
public void run(){
    System.out.println("Nova conexao com o cliente " +
        this.cliente.getInetAddress().getHostAddress());

    try {
        Scanner s = null;
        s = new Scanner(this.cliente.getInputStream());

        //Exibe mensagem no console
        while(s.hasNextLine()){
            System.out.println(s.nextLine());
        }

        //Finaliza objetos
        s.close();
        this.cliente.close();
    }catch (IOException e) {
        e.printStackTrace();
    }
}
```

Sockets

- Código do Cliente:

```
import java.io.IOException;
import java.io.PrintStream;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Scanner;

//Prefira implementar a interface Runnable do que estender a classe Thread, pois
//neste caso utilizaremos apenas o método run.
public class Client implements Runnable {
    private Socket cliente;

    public Client(Socket cliente) {
        this.cliente = cliente;
    }

    public static void main(String args[]) throws UnknownHostException, IOException {
        Socket socket = new Socket("127.0.0.1", 12345);
        /*Cria um novo objeto Cliente com a conexão socket para que seja executado em
        um novo processo.
        Permitindo assim a conexão de vários clientes com o servidor.*/
        Client c = new Client(socket);
        Thread t = new Thread(c);
        t.start();
    }
}
```

Sockets

- Código do Cliente:

```
public void run() {  
    try {  
        PrintStream saida;  
        System.out.println("O cliente conectou ao servidor");  
        //Prepara para leitura do teclado  
        Scanner teclado = new Scanner(System.in);  
  
        //Cria objeto para enviar a mensagem ao servidor  
        saida = new PrintStream(this.cliente.getOutputStream());  
  
        //Envia mensagem ao servidor  
        while (teclado.hasNextLine()) {  
            saida.println(teclado.nextLine());  
        }  
        saida.close();  
        teclado.close();  
        this.cliente.close();  
        System.out.println("Fim do cliente!");  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```



Threads e Sockets

Exercícios

Exercício 1

- Modifique o código do Clinte Servidor com Threads e faça com que o número de acessos seja registrado em um arquivo.
 - use uma variável compartilhada para isso.
 - evite múltiplos clientes acessando o arquivo ao mesmo tempo.

Exercício 2

- Faça um Servidor com múltiplas Threads, onde o Cliente pode comprar e vender ingressos para um show.
 - Se o usuário digitar a opção 1, ele recebe um ingresso.
 - Se o usuário digitar a opção 2, ele vende um ingresso.
- O servidor começa com 100 ingressos e deve manter a contagem do total.
 - Ele não pode permitir que o usuário receba um ingresso se o total está esgotado.

Conteúdo



<https://gustavotcustodio.github.io/sdmobile.html>

Obrigado

gustavo.custodio@anhembi.br