65DSD - TRABALHO 2

Simulador de Tráfego com Threads

Uma análise de concorrência com **Semáforos** e **Monitores** em Java

Autores:

Mario Alves dos Santos Júnior

Carlos Vinicius Boehme

SISTEMAS PARALELOS E DISTRIBUÍDOS

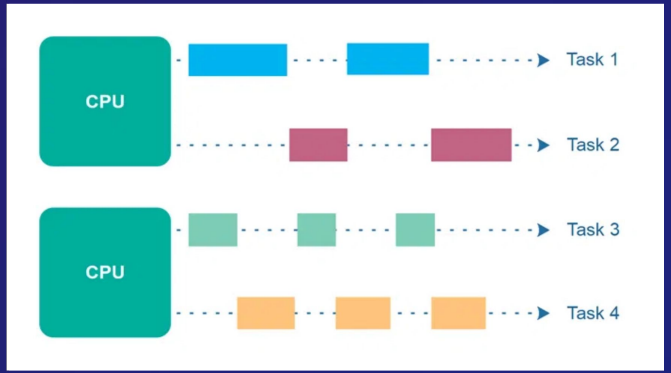
O Desafio da Concorrência no Tráfego Urbano

O Problema

Simular o movimento de múltiplos veículos (representados por **threads)** em uma uma malha viária urbana, onde cruzamentos representam **seções críticas** que exigem controle de acesso para evitar colisões e garantir o fluxo contínuo.

Objetivos do Projeto

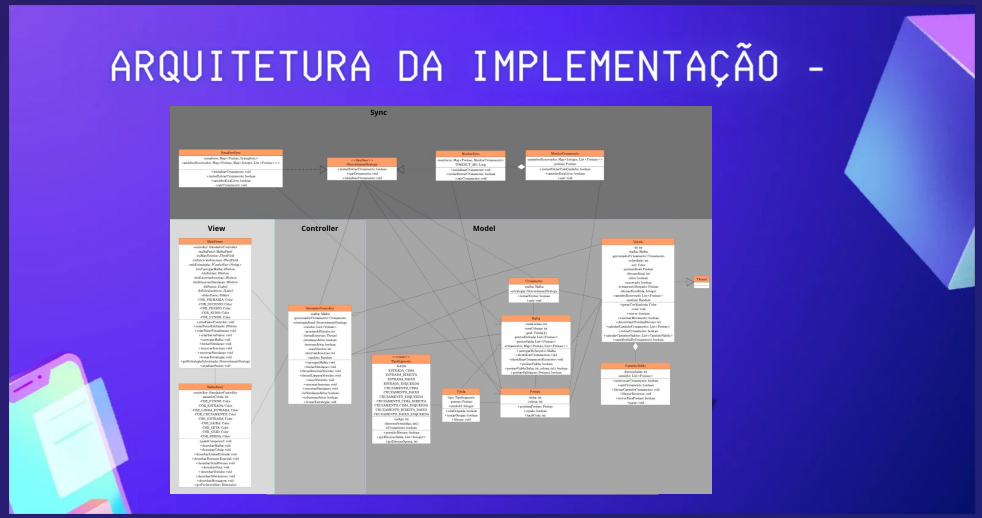
- Simular veículos autônomos em malha configurável
- Implementar e comparar **Semáforos** e **Monitores**
- Visualizar comportamento em tempo real
- Demonstrar conceitos de programação concorrente



Contexto: Este problema é análogo a sistemas distribuídos reais, como controle de tráfego aéreo, gerenciamento de recursos em bancos de dados e coordenação de processos em sistemas operacionais.

Estrutura Modular Baseada no Padrão MVC

O projeto foi desenvolvido seguindo o padrão **Model-View-Controller (MVC)** , garantindo separação de responsabilidades e facilitando a manutenção e extensibilidade do código.



Estrutura de Pacotes

Pacote	Responsabilidade
model	Lógica de negócio e estruturas de dados (Veiculo, Malha, Cruzamento, Celula)
view	Interface gráfica com Java Swing (MainFrame, MalhaPanel)
controller	Coordenação da simulação (SimuladorController)
sync	Estratégias de sincronização (SemaforoSync, MonitorSync)

Padrão Strategy: Permite troca dinâmica entre diferentes mecanismos de sincronização, demonstrando design orientado a objetos robusto e flexível.

Estruturas Fundamentais do Sistema de Simulação

As classes base que representam a infraestrutura viária e controlam a ocupação do espaço

1

Malha.java

Representa a infraestrutura urbana como uma matriz de células

- Carregada a partir de arquivos de texto configuráveis
- Identifica automaticamente pontos de entrada e saída
- Gerencia diferentes tipos de segmentos
- Fornece métodos para navegação na grade

Formato: Matriz bidimensional onde cada célula contém um código numérico representando o tipo de segmento viário

2

Celula.java

Unidade básica da malha com controle de ocupação thread-safe

- Controla ocupação por veículos usando mecanismos thread-safe
- Armazena o tipo de segmento e ID do veículo ocupante
- Implementa métodos sincronizados

Métodos principais:

```
tentarOcupar(veiculoId)
liberar()
estaOcupada()
```

3

TipoSegmento.java

Define as regras de navegação para cada tipo de via

- Define direções permitidas para cada tipo de via
- Implementa lógica de navegação (retas, curvas, cruzamentos)
- Garante que veículos sigam regras válidas

Tipos suportados:

Retas (horizontal/vertical)
Curvas (4 direções)
Cruzamentos (múltiplas saídas)

Veículos Autônomos: Threads Independentes

Classe Veiculo.java (extends Thread)

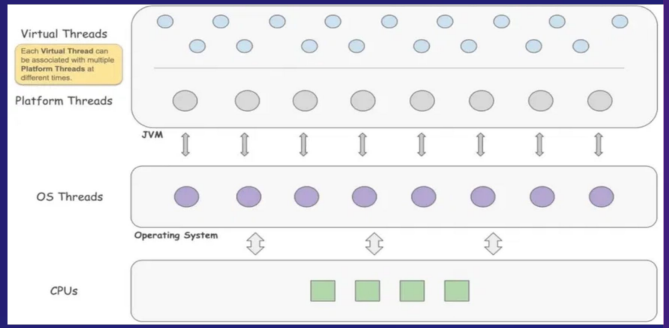
Cada veículo é uma **thread independente** com ciclo de vida próprio, possuindo atributos únicos: ID, cor, velocidade, posição e direção.

Ciclo de Vida (método run())

1. **Inicialização:** Ocupa a célula inicial na malha
2. **Movimento Contínuo:** Loop principal executado enquanto ativo
3. **Tomada de Decisão:** Determina direção baseada no tipo de segmento
4. **Gestão de Cruzamentos:** Solicita permissão, atravessa e sai
5. **Finalização:** Libera recursos ao chegar à saída

Estratégias de Movimento

- **Estradas:** Movimento simples, segue direção permitida
- **Entrada em cruzamento:** Calcula caminho válido e solicita reserva
- **Travessia:** Segue caminho previamente reservado
- **Saída:** Libera recursos e retorna ao movimento normal



Implementação Thread-Safe:

Cada veículo gerencia seu próprio estado e interage com recursos compartilhados (células da malha) através de métodos sincronizados.

Cruzamentos como Seções Críticas

O Problema

Cruzamentos representam **seções críticas** onde múltiplas threads (veículos) competem por acesso a recursos compartilhados (células do cruzamento). Sem sincronização adequada, ocorrem:

Condições de Corrida

Múltiplos veículos tentam ocupar a mesma célula simultaneamente

Colisões

Violação da integridade dos dados (dois veículos na mesma posição)

Deadlocks

Veículos bloqueiam uns aos outros indefinidamente

Solução Implementada: Padrão Strategy

```
public interface SincronizacaoStrategy { void inicializarCruzamento (Posicao cruzamento); boolean tentarEntrarCruzamento (Posicao cruzamento, List<Posicao > caminhoDesejado, int veiculoId, int direcaoSaida, Malha malha); void sairCruzamento (Posicao cruzamento, int veiculoId); String getNome (); }
```

SemaforoSync

Implementação usando **java.util.concurrent.Semaphore** com controle explícito de acquire/release

MonitorSync

Implementação usando **synchronized** com wait/notifyAll para espera condicional

Semáforos: Controle de Acesso Explícito

Classe SemaforoSync.java

Utiliza **java.util.concurrent.Semaphore** com política FIFO (fair = true). Cada cruzamento é protegido por um **semáforo binário** com 1 permissão.

Protocolo de Entrada

1. Tentativa de Aquisição

tryAcquire(100, TimeUnit.MILLISECONDS) com timeout

2. Verificação de Disponibilidade

Checa se todas as células do caminho estão livres

3. Reserva Atômica

Ocupa todas as células do caminho desejado

4. Liberação do Semáforo

release() para permitir outras tentativas

5. Rollback em Falha

Se a reserva falhar, libera tudo e retorna false

Protocolo de Saída

Remoção do Caminho Reservado

Remove o caminho do mapa de controle

Liberação de Células

As células são liberadas pelo próprio veículo durante o movimento

Vantagens

- **Controle explícito e previsível** do acesso aos recursos compartilhados
- **Baixo overhead computacional** comparado a outras primitivas
- **Timeout evita espera indefinida** e possíveis deadlocks

Característica Principal: Abordagem de "baixo nível" onde o controle de espera e notificação é gerenciado pelo próprio semáforo do Java, oferecendo controle fino sobre a sincronização.

Monitores: Encapsulamento com Sincronização Implícita

Classe MonitorSync.java

Mecanismo

• Utiliza blocos **synchronized** e métodos **wait()** / **notifyAll()**

• Cada cruzamento possui um objeto **MonitorCruzamento** que encapsula o estado

Protocolo de Entrada

1. Método `synchronized` garante exclusão mútua

2. Se caminho não está livre, chama `wait(timeout)`

3. Ao ser acordado, verifica novamente a condição

4. Se caminho livre, reserva todas as células

5. Armazena caminho reservado e retorna **true**

Protocolo de Saída

• Remove o caminho reservado do mapa de controle

• Chama `notifyAll()` para acordar threads em espera

Vantagens

✓ Paradigma OO: estado e sincronização encapsulados

✓ Espera eficiente: threads não consomem CPU

✓ Notificação broadcast: todos reavaliam condições

⚡ Diferença Chave

Monitores implementam espera **passiva** (thread dorme), enquanto semáforos podem implementar espera **ativa** (polling).

Interface Interativa para Experimentação

Interface gráfica desenvolvida em **Java Swing** para visualização e controle da simulação em tempo real

1. Painel de Visualização

MalhaPanel: Renderiza a malha e veículos em tempo real

- Cada veículo tem cor distinta para identificação
- Atualização contínua para animação fluida
- Representação visual dos tipos de segmento

2. Controles de Configuração

Seleção de Malha: ComboBox com arquivos .txt disponíveis

Estratégia: Toggle entre Semáforos e Monitores

Quantidade de Veículos: Spinner (1-100)

Intervalo de Inserção: Slider (delay em ms)

3. Botões de Ação

Iniciar/Pausar: Controla o estado da simulação

Encerrar Inserção: Para de criar novos veículos

Encerrar Tudo: Finaliza todos os veículos

Feedback: Mostra número de threads ativas

Componentes da Interface

A interface permite experimentação interativa com diferentes configurações de malha, estratégias de sincronização e parâmetros de simulação, facilitando a análise comparativa do comportamento dos veículos sob diferentes condições de concorrência.

MainFrame.java + **MalhaPanel.java**

Execução e Observação do Comportamento Concorrente

Roteiro de Demonstração

1. **Inicialização**
   Executar Main.java e carregar malha de exemplo (malha_exemplo.txt)
2. **Seleção de Estratégia**
   Escolher "Semáforos" no ComboBox de estratégias
3. **Configuração**
   Definir 20 veículos máximos e intervalo de inserção de 500ms
4. **Execução**
   Clicar em "Iniciar" e observar veículos sendo criados nos pontos de entrada
5. **Comparação**
   Encerrar simulação, trocar para "Monitores" e repetir execução

Resultado Esperado

Ambas as estratégias previnem colisões com sucesso, demonstrando a eficácia dos mecanismos de sincronização implementados.

Observações Chave

Nos Cruzamentos

Veículos aguardam sua vez de forma ordenada, respeitando a estratégia de sincronização selecionada. Não há colisões ou sobreposições.

Nas Vias

Movimento fluido sem colisões. Cada célula é ocupada por no máximo um veículo por vez, garantindo integridade dos dados.

Nas Saídas

Veículos são removidos automaticamente ao atingir pontos de saída, e suas threads são finalizadas corretamente.

Visualização em Tempo Real

A interface gráfica atualiza continuamente, permitindo observar o comportamento das threads e a eficácia da sincronização. Cada veículo possui cor única para fácil rastreamento.

Análise Comparativa e Lições Aprendidas

Comparativo entre Estratégias

Aspecto	Semáforos	Monitores
Paradigma	Primitiva de sincronização explícita	Encapsulamento OO com sincronização implícita
Complexidade	Controle manual de acquire/release	Gerenciamento automático de locks
Espera	Timeout com tryAcquire	Espera condicional com wait/notify
Deadlock	Médio (se mal implementado)	Baixo (notifyAll acorda todos)
Performance	Ligeiramente mais rápido (menos overhead)	Overhead de notificações broadcast
Legibilidade	Menos intuitivo	Mais alinhado com OO

Resultados Observados

- Ambas as estratégias **preveniram colisões** com sucesso
- Nenhum deadlock observado em testes com até 100 veículos
- Monitores apresentaram comportamento mais previsível em cenários de alta contenção
- Semáforos demonstraram performance ligeiramente superior em baixa contenção

Conclusões

1. A escolha entre Semáforos e Monitores depende do contexto e preferências de design
2. O padrão Strategy permitiu comparação direta sem modificar o simulador
3. O projeto demonstrou conceitos fundamentais de programação concorrente
4. A visualização em tempo real facilitou a compreensão do comportamento das threads

Lição Principal: Ambas as abordagens são eficazes para resolver problemas de concorrência. A arquitetura modular e o uso de padrões de design (MVC, Strategy) foram fundamentais para o sucesso do projeto, permitindo extensibilidade e facilitando a comparação entre diferentes estratégias de sincronização.

Obrigado!

Perguntas?

Autores:

Mario Alves dos Santos Júnior

Carlos Vinicius Boehme

65DSD_Threads

Simulador de Tráfego em Malha Viária

SISTEMAS PARALELOS E DISTRIBUÍDOS