Sistemas Distribuídos
Programação de um Semáforo

Thiago O. da Silva[1] **Cláudio Matheus da S. Sousa**[3]

[1]*Departamento de Ciência da Computação – Universidade Federal do Piauí (UFPI)*
Teresina – PI – Brazil

[2]*Departamento de Ciência da Computação – Universidade Federal do Piauí (UFPI)*
Teresina – PI – Brazil

{*thiago_silva,c.matheus*}@ufpi.edu.br

Abstract. *This report aims to plan an application of a code implementation for a traffic light in a distributed application.*

Resumo. *Este relatório tem como objetivo planejar uma aplicação de um a implementação de código para um semáforo em uma aplicação de distribuída.*

1. Ambiente distribuído simulado com vários nós.

1.1. Servidor

```
// Dependências
const express = require('express')
const {Sequelize, Model, DataTypes} = require('sequelize')

// Inicialização
const app = express()
const sequelize = new Sequelize(/* config bd */)

// Models
const User = sequelize.define('User', {/* atributos */})

// APIs
app.get('/users', authMiddleware, async (req, res) => {
  // obtém usuários
  const users = await User.findAll()
  res.json(users)
})

app.listen(3000)
```

2. Cliente

```
// Requisições HTTP
const axios = require('axios')

// Buscar usuários
const getUsers = async () => {
  const res = await axios.get('/users', {
    headers: {
      Authorization: `Bearer ${jwtToken}`
    }
  })
```

```
  console.log(res.data)
}

getUsers()
```

3. Implementação de semáforo

Um semáforo (mutex) é um mecanismo de controle de acesso concorrente que garante que apenas uma thread ou processo possa acessar uma seção crítica de código por vez. Resumidamente, um semáforo coordena o acesso a recursos compartilhados, garantindo exclusividade e prevenindo erros de concorrência.

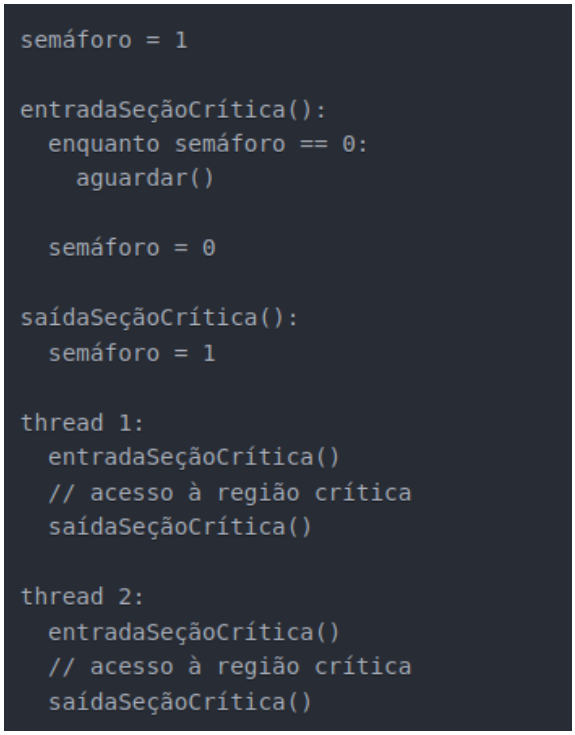

Figure 1. Pseudocódigo de um Semáforo

4. Recurso compartilhado fictício

```
// Histórico local de operações
const operacoesPendentes = [];

function propagarOperacoes() {

  // Envia operações locais para outros servidores
  enviarOperacoes(operacoesPendentes);

  // Limpa operações já propagadas
  operacoesPendentes = [];
```

```
}

function aplicarOperacoes(operacoesRecebidas) {

  // Aplica cada operação no banco de dados local
  operacoesRecebidas.forEach(op => aplicarOperacao(op))

}

// Chamada periódica para propagação
setInterval(propagarOperacoes, 5000);

// Ao receber operações externas
receiveOperacoes(ops => aplicarOperacoes(ops));
```

5. Simulação

5.1. Implementando Semáforo corretamente

- Nesse caso, o lock impede o acesso simultaneous ao recurso dadosCompartilhados, fazendo com que as threads aguardem sua vez para modificar o dado de maneira segura.
- Dessa forma, evitamos condições de corrida e inconsistências que ocorreriam se as threads modificassem os dados ao mesmo tempo sem coordenação.
- O uso correto do semáforo garante que cada thread tenha acesso exclusivo na sua vez, preservando a integridade do recurso compartilhado em ambientes concorrentes.

```
// recurso compartilhado
let dadosCompartilhados = 0;

// semáforo
let lock = false;

// entrada na seção crítica
function entrarSecaoCritica() {

  while(lock) {
    // aguarda liberação
  }

  lock = true; // obtém lock

}

// saída da seção crítica
function sairSecaoCritica() {
```

```
  lock = false; // libera lock

}

// thread 1
entrarSecaoCritica();
dadosCompartilhados++;
sairSecaoCritica();

// thread 2
entrarSecaoCritica();
dadosCompartilhados--;
sairSecaoCritica();
```

5.2. Implementando Semáforo incorretamente

- Neste caso, as threads estão incrementando o contador compartilhado sem antes verificar o lock do semáforo.

```
// recurso compartilhado
let contador = 0;

// semáforo
let lock = false;

function incrementar() {

  // não verifica o lock
  contador++;

}

// thread 1
incrementar();

// thread 2
incrementar();
```

5.3. Conclusão

Implementação errada do Semáforo pode levar as seguintes condições:

- Incrementos simultâneos: como não há exclusão mútua, as threads podem executar ao mesmo tempo, potencialmente incrementando o contador de forma incorreta.
- Dados inconsistentes: o valor final do contador pode não refletir o número real de incrementos feitos pelas threads, pois houve sobreposição.
- Condições de corrida: a ordem e tempo de execução das threads impacta o resultado final de forma imprevisível.

Uma implementação correta deveria sempre verificar o lock do semáforo antes de acessar a seção crítica, e liberá-lo após o uso, para garantir o acesso exclusivo de cada thread.

Ao ignorar o semáforo, temos uma situação propensa a falhas e inconsistências típicas de acesso concorrente sem sincronização.