

TDE3 Sistemas





Introdução

Este trabalho aborda o problema clássico do *Jantar dos Filósofos*, ilustrando a ocorrência de deadlocks em sistemas concorrentes. Vamos explorar também condições de corrida em threads e apresentar soluções práticas usando hierarquia de recursos e semáforos para evitar impasses e garantir sincronização adequada.



Jantar dos Filósofos

```
Constantes: N = 5
Recursos: Garfos[0..4]

Para cada Filósofo(i):
    // Define a ordem de pegar os garfos (menor primeiro)
    primeiro_garfo = min(i, (i + 1) % N)
    segundo_garfo = max(i, (i + 1) % N)

    Loop Infinito:
        pensar()
        estado[i] = "COM FOME"

        adquirir(Garfos[primeiro_garfo]) // Pega o menor índice
        adquirir(Garfos[segundo_garfo])  // Pega o maior índice

        estado[i] = "COMENDO"
        comer()

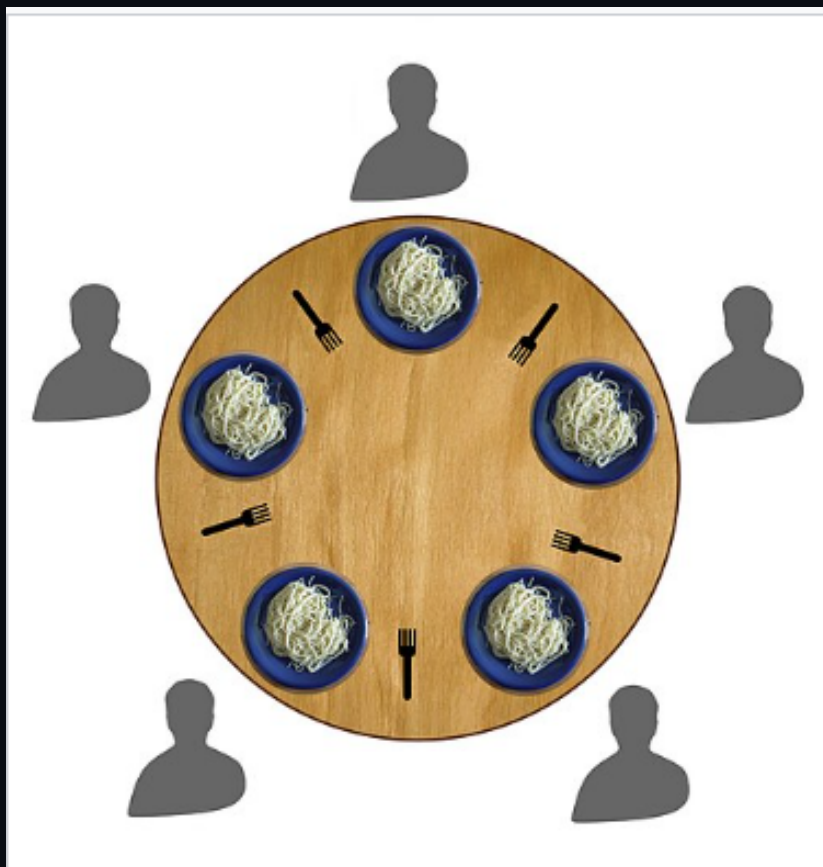
        liberar(Garfos[segundo_garfo])
        liberar(Garfos[primeiro_garfo])

        estado[i] = "PENSANDO"
```

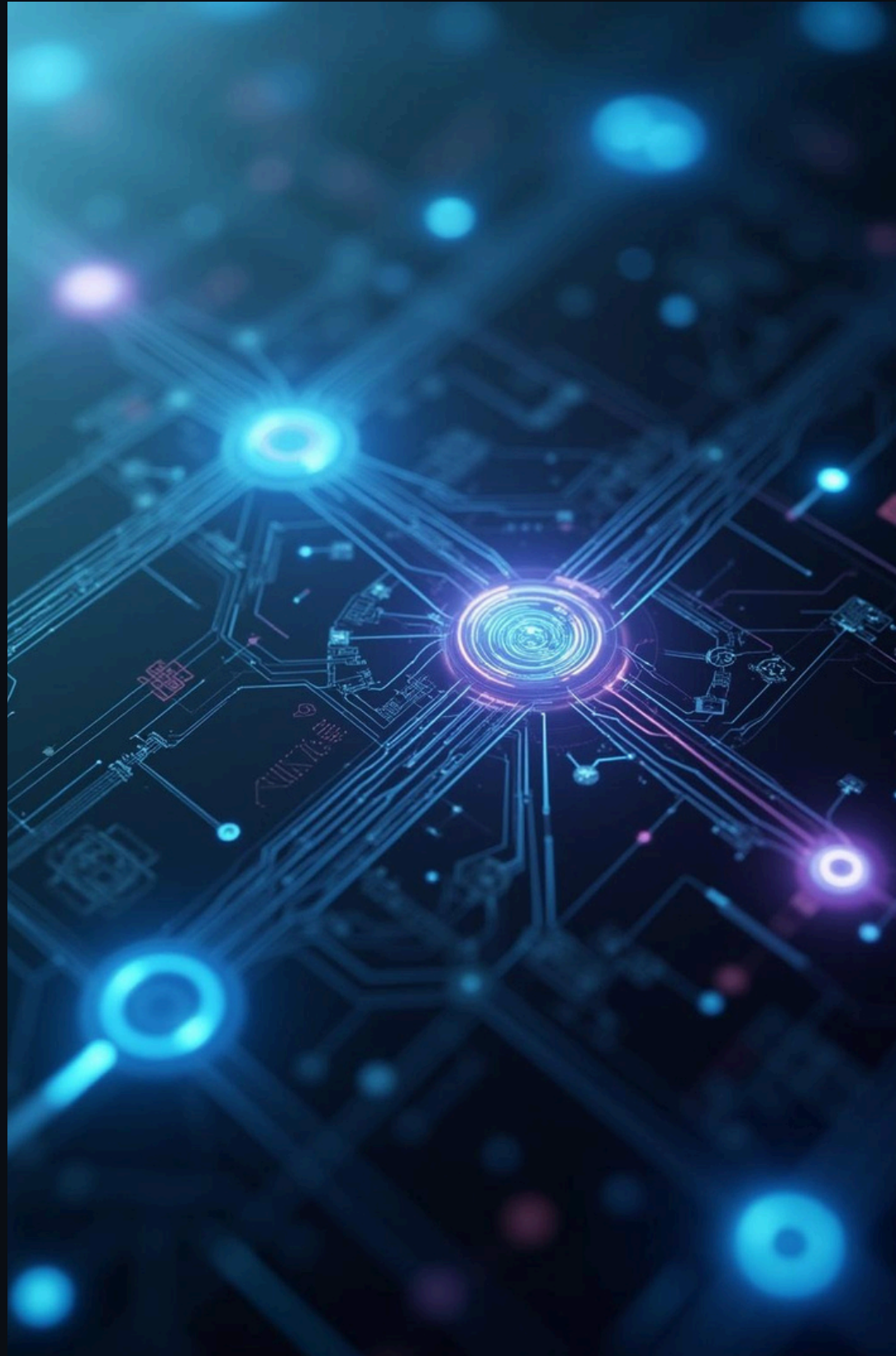
O que é Deadlock?

Definição Simples: Deadlock (ou Impasse) é uma situação em que dois ou mais processos (ou threads) ficam bloqueados indefinidamente, pois cada um está segurando um recurso e esperando por outro recurso que está sendo segurado por outro processo do grupo. Ninguém progride porque ninguém solta o que tem.

Solução por Hierarquia de Recursos

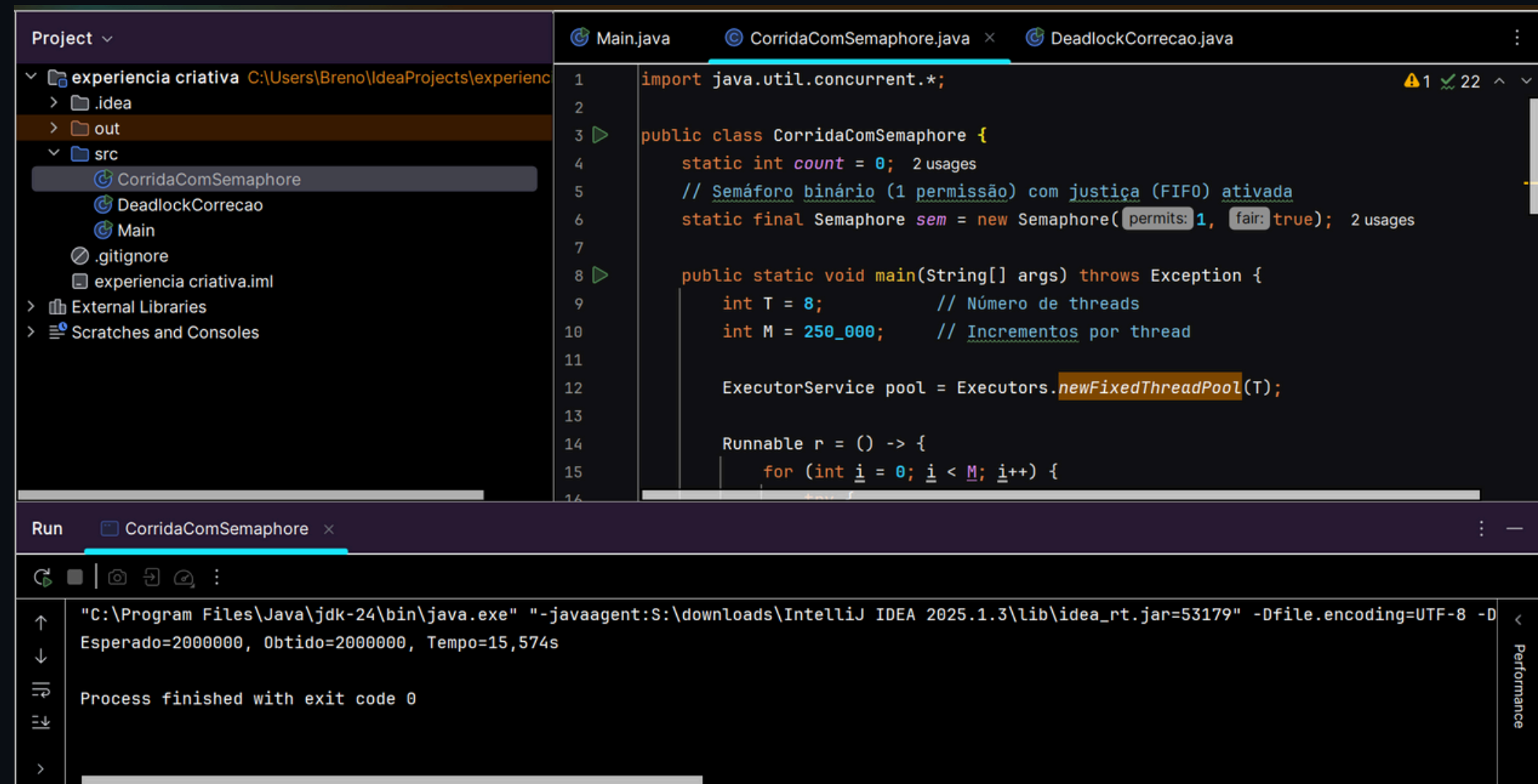


Para resolver isso, é necessário utilizar estratégia de Hierarquia de Recursos. Nós numeramos os garfos de 0 a 4. A lógica é a seguinte: todo filósofo deve pegar obrigatoriamente o garfo de menor número primeiro e só depois o de maior número. Isso quebra a condição de espera circular de Coffman, pois o último filósofo não conseguirá fechar o ciclo de dependência."



Threads e Deadlock

Uso de Semáforos para Sincronização



The screenshot displays the IntelliJ IDEA IDE interface. On the left, the Project Explorer shows a project named 'experiencia criativa' with a source folder 'src' containing files 'CorridaComSemaphore', 'DeadlockCorrecao', and 'Main'. The main editor window shows the 'CorridaComSemaphore.java' file with the following code:

```
1 import java.util.concurrent.*;
2
3 public class CorridaComSemaphore {
4     static int count = 0; 2 usages
5     // Semáforo binário (1 permissão) com justiça (FIFO) ativada
6     static final Semaphore sem = new Semaphore(permits: 1, fair: true); 2 usages
7
8     public static void main(String[] args) throws Exception {
9         int T = 8; // Número de threads
10        int M = 250_000; // Incrementos por thread
11
12        ExecutorService pool = Executors.newFixedThreadPool(T);
13
14        Runnable r = () -> {
15            for (int i = 0; i < M; i++) {
16                // ...
17            }
18        }
19    }
20 }
```

Below the editor, the Run tab shows the execution of 'CorridaComSemaphore'. The console output indicates that the process finished successfully with exit code 0. The performance metrics show 'Esperado=2000000, Obtido=2000000, Tempo=15,574s'.

Semáforos são mecanismos que garantem **exclusão mútua**, permitindo que apenas uma thread acesse um recurso crítico por vez. O uso do modo *fair* cria uma fila FIFO, prevenindo a inanição das threads. Essa sincronização evita condições de corrida, embora possa impactar o desempenho em termos de throughput.

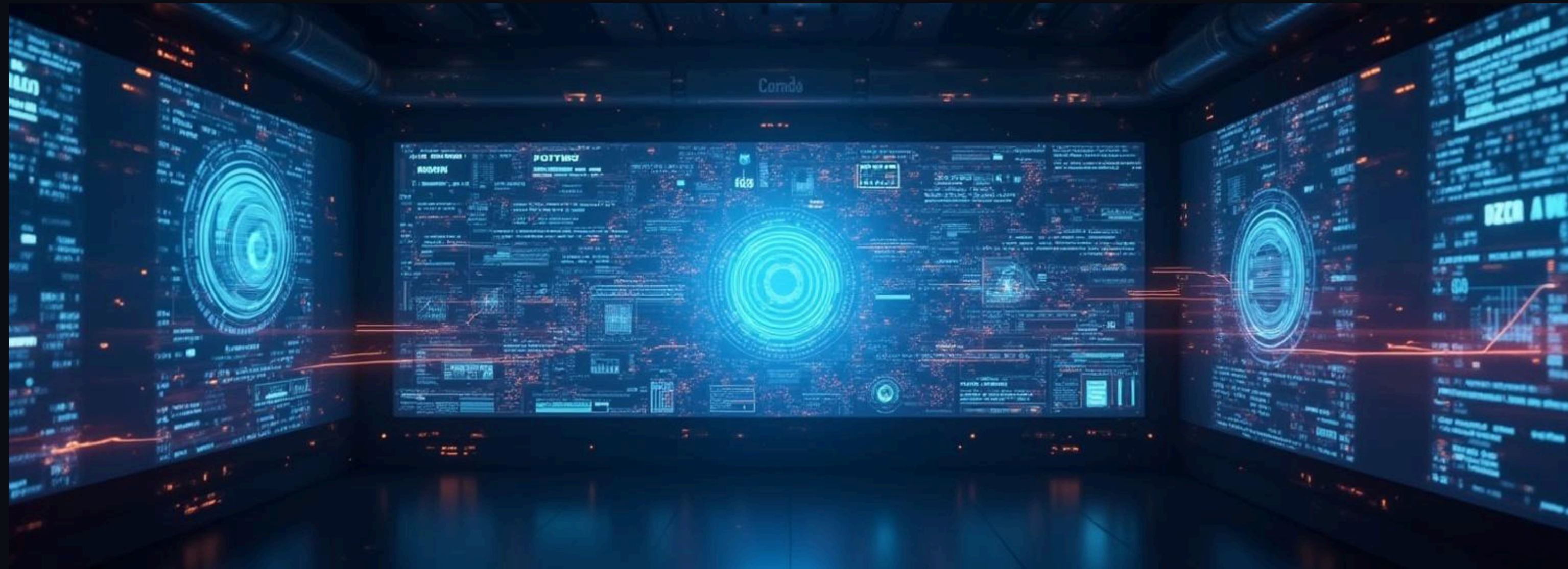
```
Project
└─ experiencia criativa C:\Users\Breno\IdeaProjects\experiencia criativa
   └─ src
      ├── CorridaComSemaphore
      ├── DeadlockCorrecao
      ├── Main
      ├── .gitignore
      └── experiencia criativa.iml
   └─ External Libraries
   └─ Scratches and Consoles

Main.java CorridaComSemaphore.java DeadlockCorrecao.java
1 import java.util.concurrent.*;
2
3 public class DeadlockCorrecao {
4     static final Object LOCK_A = new Object(); 2 usages
5     static final Object LOCK_B = new Object(); 2 usages
6
7     public static void main(String[] args) {
8         Thread t1 = new Thread(() -> {
9             // Ordem: A -> B
10            synchronized (LOCK_A) {
11                System.out.println("T1 pegou A");
12                dormir(ms: 50);
13                synchronized (LOCK_B) {
14                    System.out.println("T1 pegou B -> T1 concluiu");
15                }
16            }
17        });
18    }
19 }
```

```
Run DeadlockCorrecao
"C:\Program Files\Java\jdk-24\bin\java.exe" "-javaagent:S:\downloads\IntelliJ IDEA 2025.1.3\lib\idea_rt.jar=53222" -Dfile.encoding=UTF-8 -D
T1 pegou A
T1 pegou B -> T1 concluiu
T2 pegou A
T2 pegou B -> T2 concluiu
```

Análise e Correção do Deadlock

O deadlock ocorre quando coexistem quatro condições: exclusão mútua, hold and wait, não preempção e espera circular. A correção requer impor uma ordem fixa na aquisição dos recursos, eliminando a espera circular e prevenindo que as threads travem indefinidamente, garantindo a continuidade e a estabilidade do sistema.



Conclusões

A aplicação da hierarquia de recursos e o uso de semáforos são fundamentais para evitar deadlocks e condições de corrida em sistemas operacionais. Compreender as condições de Coffman permite diagnosticar e corrigir esses impasses, assegurando a eficiência e confiabilidade dos processos concorrentes. A prática e o controle rigoroso são essenciais para o sucesso em ambientes multithread.



Obrigado!

CREDITS: Breno Augusto Rocha