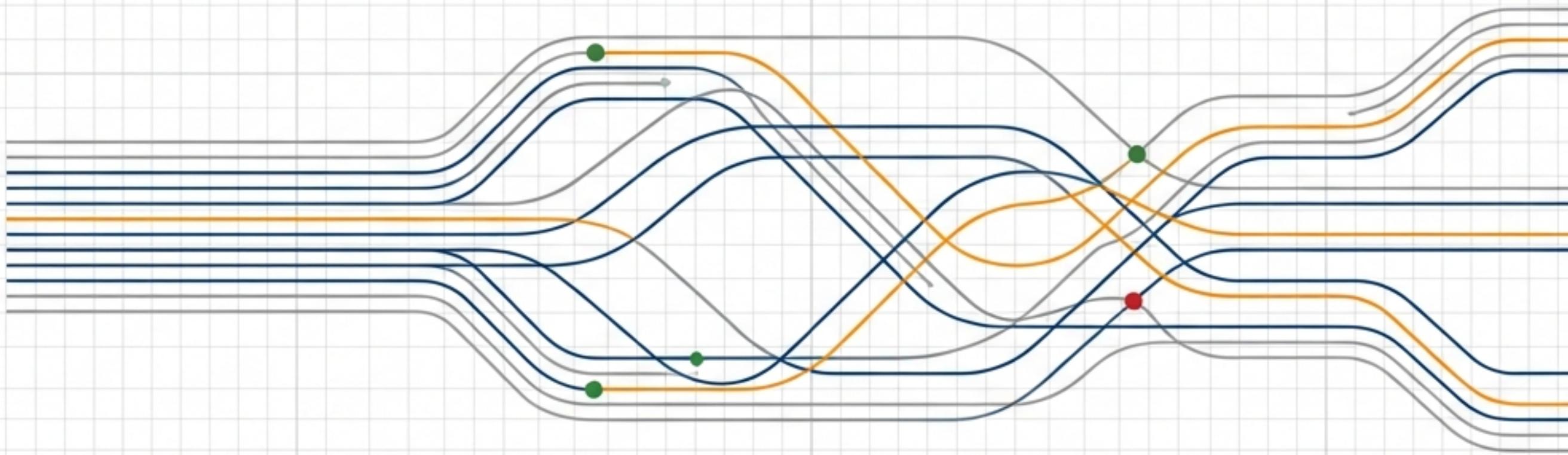


Programação Concorrente em Java: Threads

Ciclo de vida, Sincronização e Multithreading na Prática



Baseado no material de Rafael Vargas Mesquita



MULTITASKING VISUALIZED

O Mundo é Concorrente

(e o seu Software também deveria ser)

A Analogia Biológica:

Assim como o corpo humano respira, bombeia sangue e processa pensamentos simultaneamente, aplicações modernas precisam realizar múltiplas tarefas ao mesmo tempo.

A Necessidade:

Enquanto um software processa dados, ele precisa responder ao usuário, acessar arquivos e comunicar-se pela rede.

O Mecanismo:

Em Java, a **Thread** é o agente central que permite dividir o trabalho em tarefas independentes para manter a aplicação responsiva.

Concorrência vs. Paralelismo

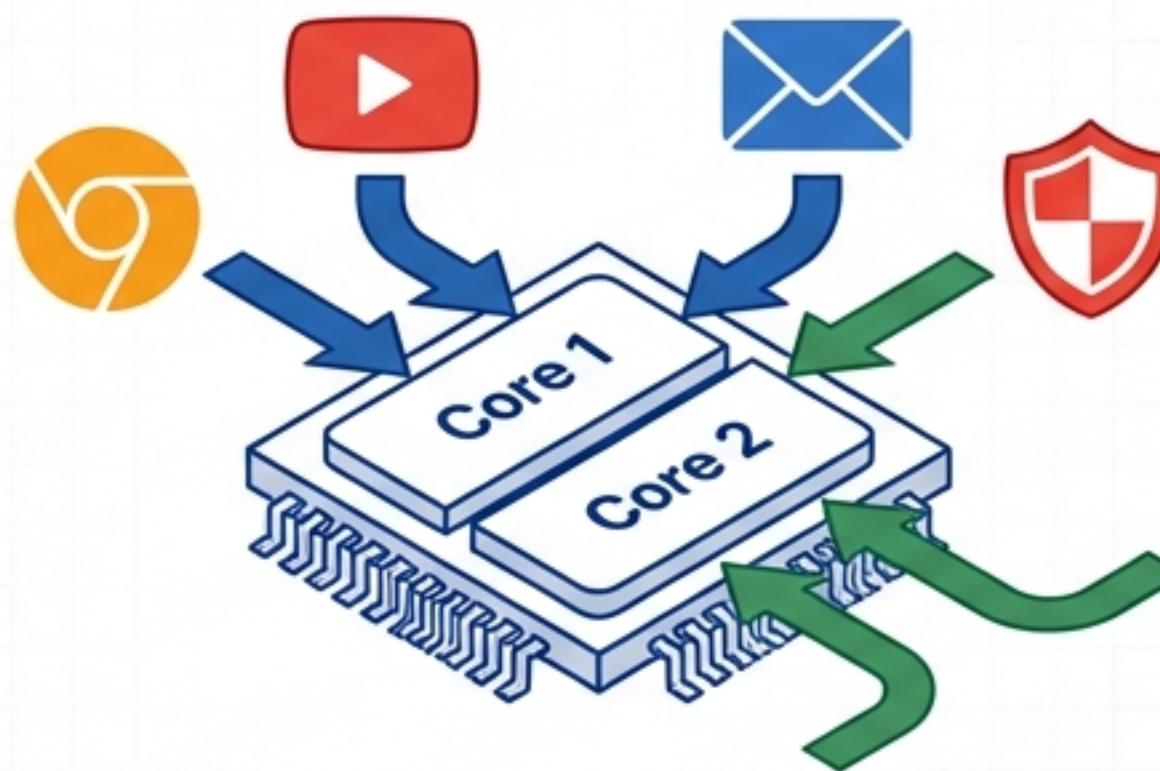
Concorrência é sobre estrutura. Paralelismo é sobre execução.

Concorrência (Single Core)



- **Cenário:** Um único núcleo (Single Core).
- **Mecânica:** Alternância rápida (Context Switching).
- **Resultado:** Ilusão de simultaneidade.

Paralelismo Real (Multi-Core)

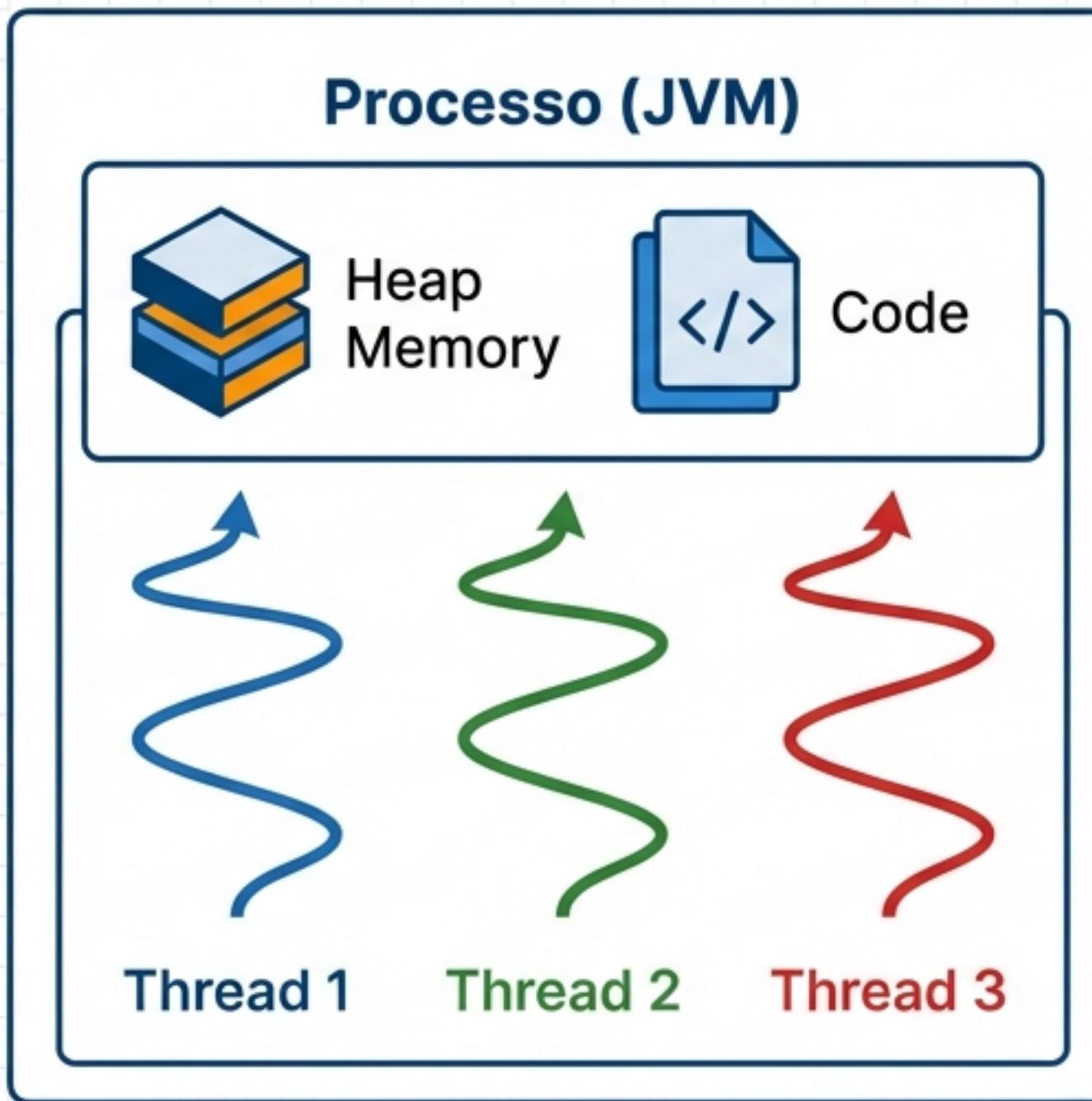


- **Cenário:** Múltiplos núcleos.
- **Mecânica:** Execução física simultânea.
- **Resultado:** Tarefas rodam verdadeiramente ao mesmo tempo.

A Menor Unidade de Processamento

Processo: O programa em execução. Contém os recursos macro.

Thread: Um fluxo de execução *dentro* do processo. Compartilha a memória do processo pai.



Single-Thread
(Sequencial)



Multi-Thread
(Concorrente)



A JVM utiliza threads até para tarefas internas, como o *Garbage Collection*.

Criando Threads: A Prática

1. extends Thread

Herança direta. Menos flexível (Java não permite herança múltipla).

```
Runnable tarefa = () -> {
    System.out.println(Thread.currentThread().getName() + " rodando");
};

Thread t1 = new Thread(tarefa);
t1.start(); // O jeito certo!
```

2. implements Runnable (Recomendado)

Separa a tarefa (`run`) da execução. Mais flexível e orientado a objetos.

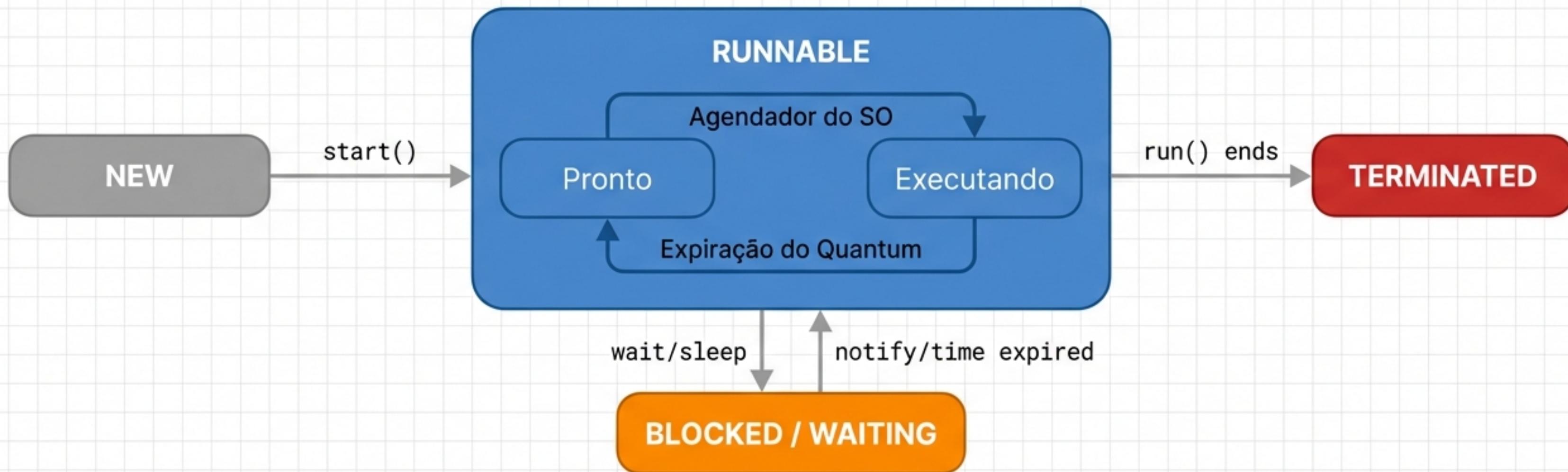


ATENÇÃO: start() vs run()

Jamais chame `t.run()` diretamente!

- `t.run()`: Executa na thread atual (**sequencial**).
- `t.start()`: Cria uma nova pilha de execução (concorrente).

O Ciclo de Vida da Thread



- **NEW:** Instanciada, mas não iniciada.
- **RUNNABLE:** Viva. Pode estar na CPU ou na fila de espera.
- **BLOCKED:** Parada aguardando recurso ou sinal.
- **TERMINATED:** Execução finalizada.

Por que uma Thread para?

Diferença entre **Sleep** e **Wait**

TIMED_WAITING (O Sono)



Exemplo: Editor salvando backup automático.

Código: Thread.sleep(30000)

Mecânica: Pausa voluntária por tempo fixo.
Retorna automaticamente.

WAITING / BLOCKED (O Bloqueio)

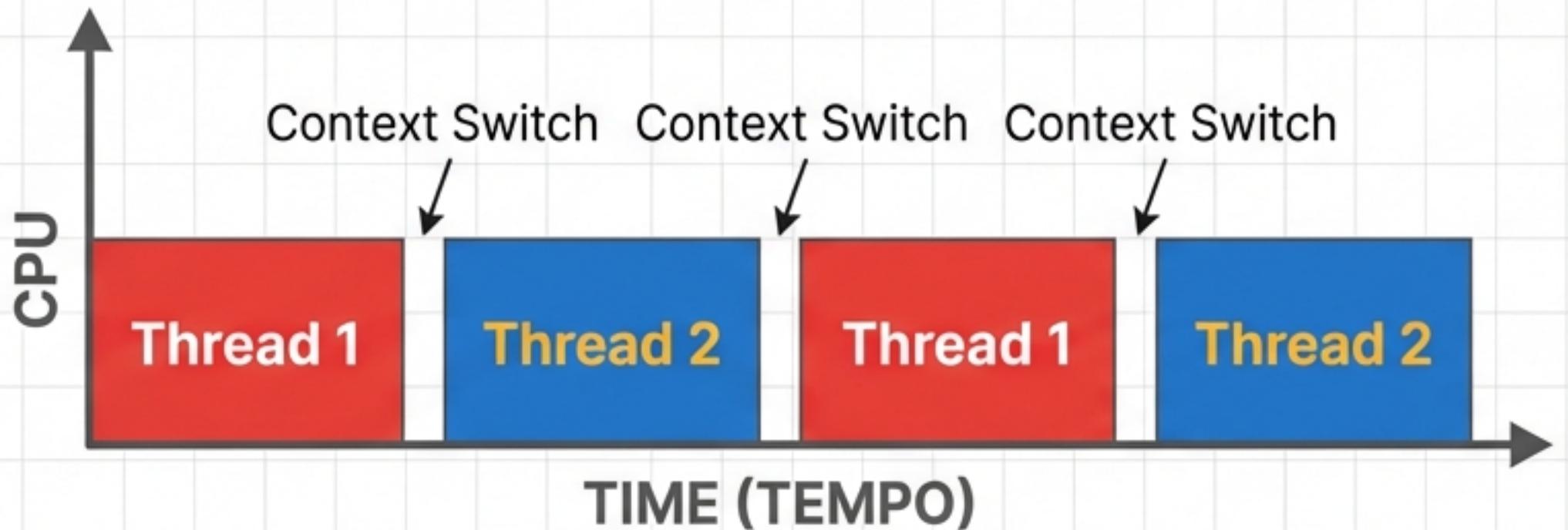


Exemplo: Tentando acessar um arquivo já aberto.

Código: wait() ou synchronized

Mecânica: Espera indefinida. Precisa de um
notify() ou liberação de recurso para voltar.

O Agendador e as Prioridades



Prioridades Java:

- MIN_PRIORITY (1)
- NORM_PRIORITY (5) - Padrão
- MAX_PRIORITY (10)

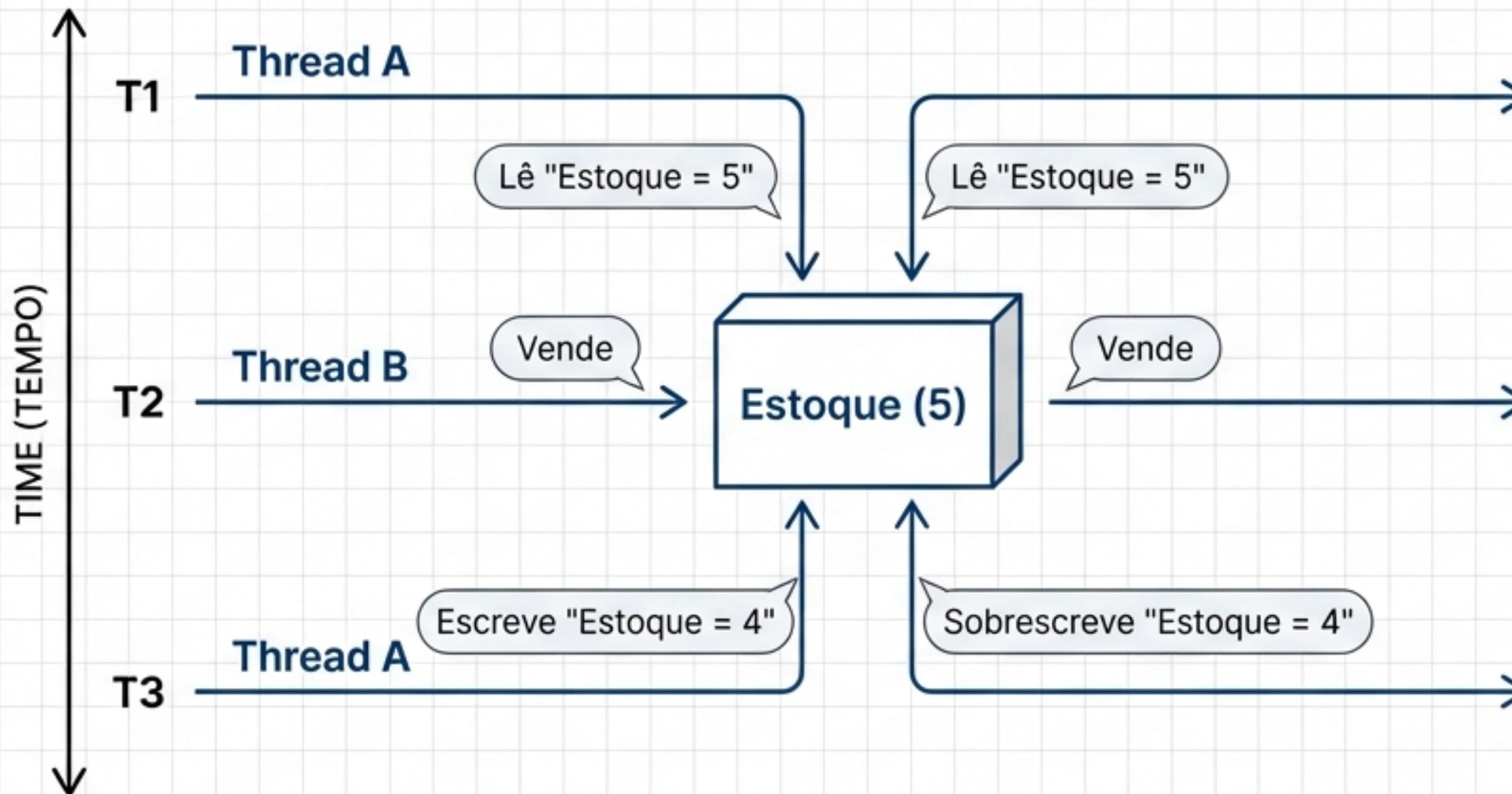
- **O Agendador (Scheduler):** O componente do SO que decide quem roda.
- **Quantum (Fatia de Tempo):** Período curto de posse da CPU antes da preempção.

*Note: Maior prioridade = Maior chance de execução (sem garantia absoluta).



O Perigo: Condição de Corrida

O Caso da Loja de iPhones



Cenário:

15 clientes tentam comprar 5 iPhones simultaneamente.

O Erro:

Sem sincronização, múltiplas threads leem o mesmo valor antes da atualização.

Consequência:

Estoque negativo ou vendas fantasma.

Bad Code

```
if (estoque > 0) {  
    Thread.sleep(100); // Latência  
    estoque--; // CRÍTICO!   
}
```

****Inconsistência! Uma venda foi perdida.****

A Solução: Sincronização



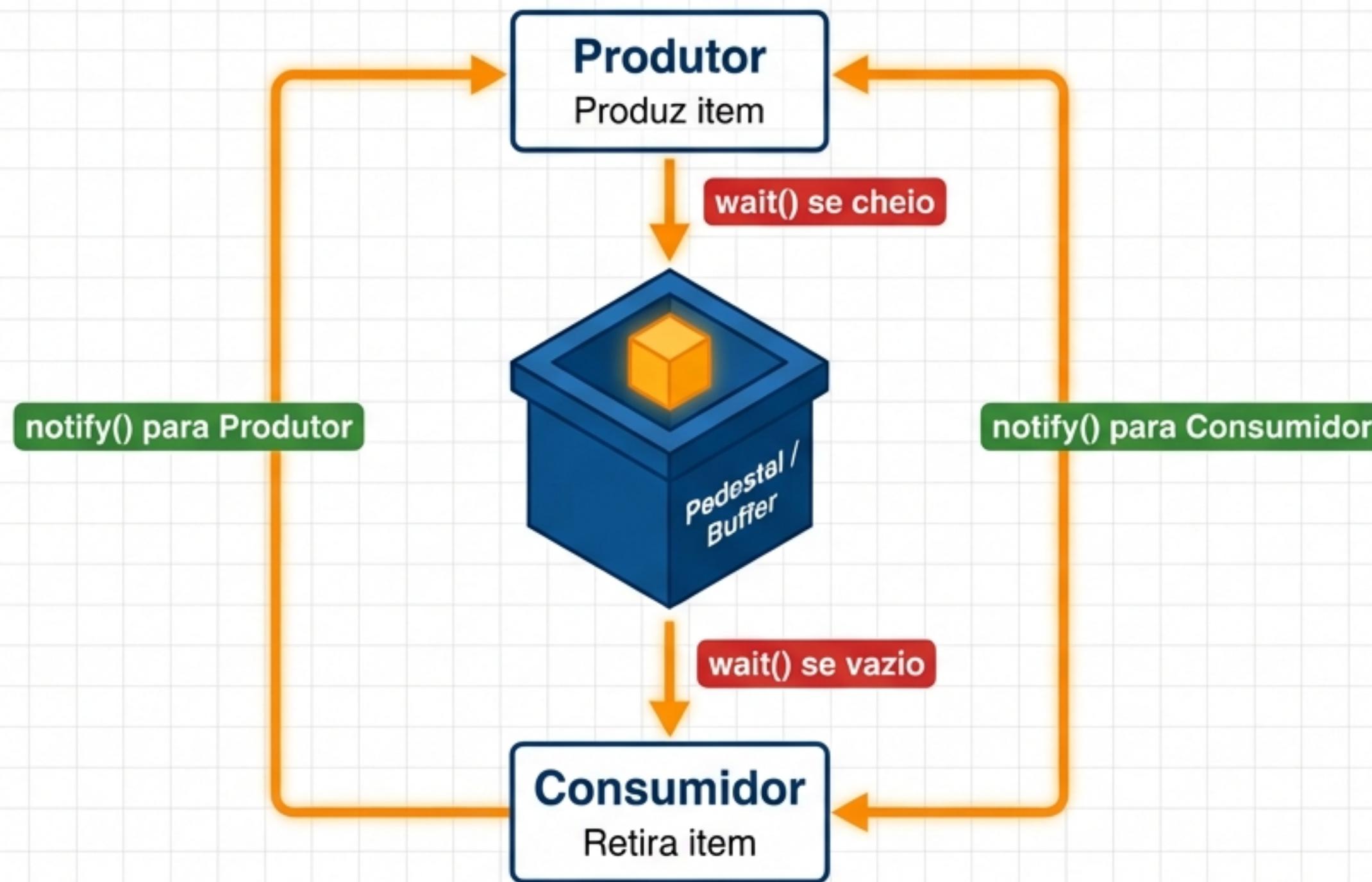
- Exclusão Mútua: Apenas uma thread pode entrar na Seção Crítica por vez.
- Monitor (Lock): O mecanismo interno que tranca o objeto.

Good Code

```
public synchronized void efetuarCompra() {  
    // Agora é seguro!  
    if (estoque > 0) {  
        estoque--;  
        System.out.println("Venda Sucesso.");  
    }  
}
```

Comunicação entre Threads

Padrão Produtor e Consumidor (Wait & Notify)



A Coreografia

1. **wait()**: “O pedestal está cheio/vazio? Libero o lock e durmo.”
2. **notify()**: “Realizei minha tarefa! Acordo quem estava esperando.”

Text Note: **wait()** e **notify()** devem ser sempre chamados dentro de um bloco **synchronized**.

Multithreading em Interfaces Gráficas (GUI)

O Problema:



Rodar tarefas pesadas no clique do botão congela a tela.

A Regra:

A Thread da GUI (JavaFX Application Thread) serve apenas para desenho.

A Solução:

1. Processamento pesado em Worker Thread.
2. Atualização visual via agendamento.

JavaFX Example

```
Platform.runLater(() -> {  
    label.setText("Processo Finalizado!");  
});
```

Or reference
'javafx.concurrent.Task'.

Execução em
Background





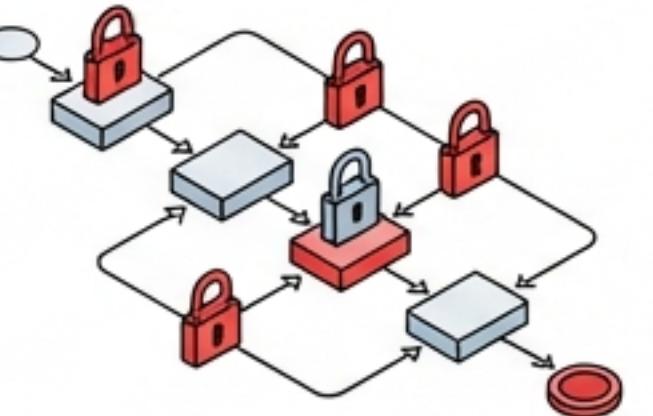
Checklist de Segurança e Riscos

Deadlock (Abraço Mortal)



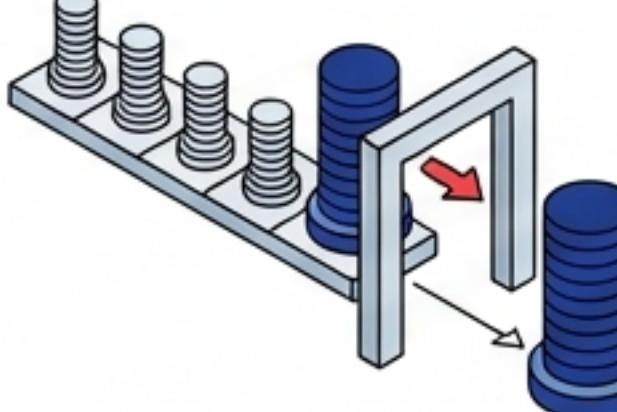
Duas threads travadas,
uma esperando o lock da
outra eternamente.

Sincronização Excessiva



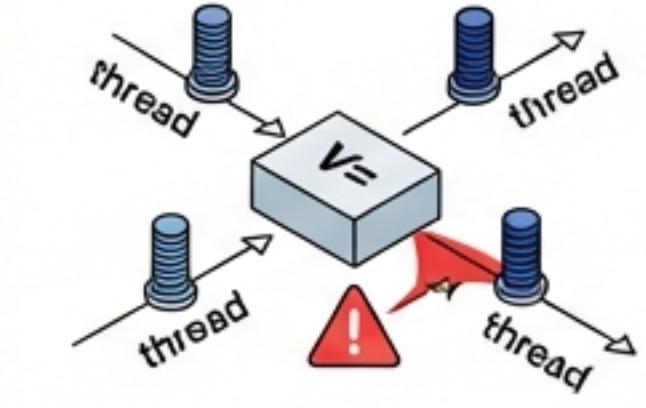
Torna o programa
sequencial e lento.
Sincronize apenas o
necessário.

Starvation (Inanição)



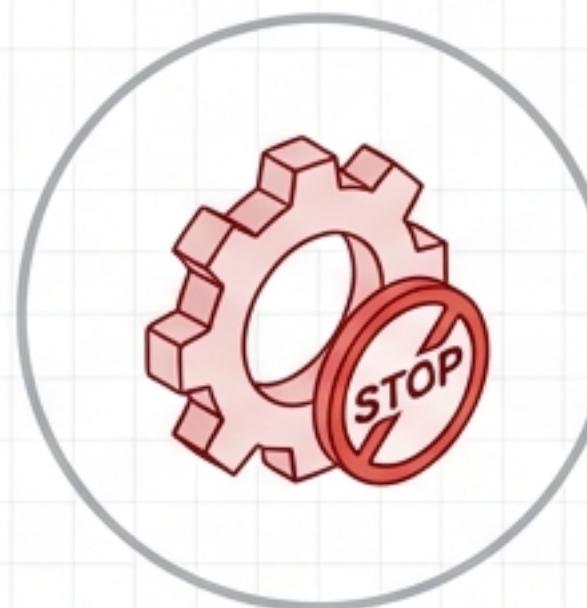
Threads de baixa
prioridade nunca
executam devido ao
excesso de carga.

Atomicidade



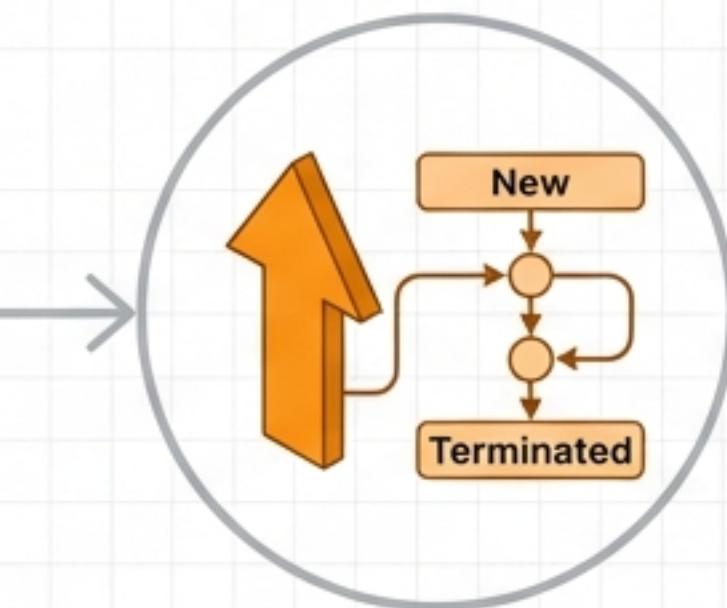
Operações como
'count++' não são
atômicas. Use
'AtomicInteger'.

Resumo da Jornada



O Problema

Processamento sequencial ineficiente.



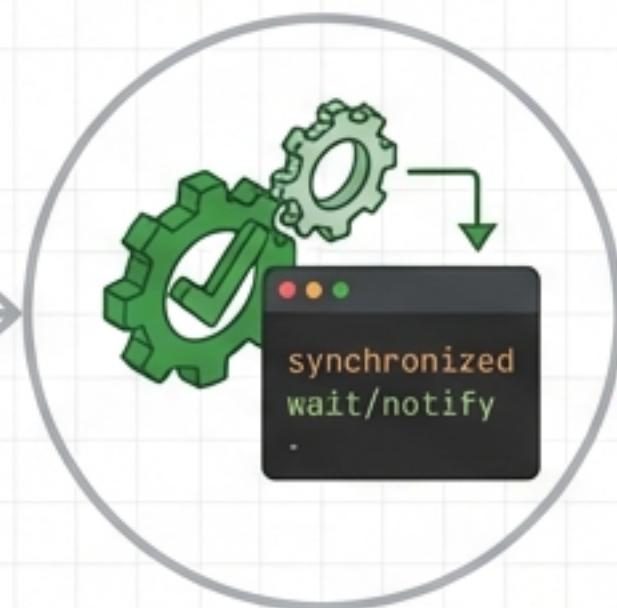
A Ferramenta

Threads e Ciclo de Vida
(New -> Terminated).



O Risco

Race Conditions em dados compartilhados.



A Solução

Sincronização
(`synchronized`,
'wait/notify').

Conclusão e Próximos Passos

A programação concorrente adiciona complexidade, mas é o pilar de performance e responsividade em sistemas modernos.

Dominar Threads, Locks e Sincronização distingue desenvolvedores comuns de engenheiros de software.



Próximo Nível: Programação Distribuída

Conectando aplicações via Sockets e Rede.