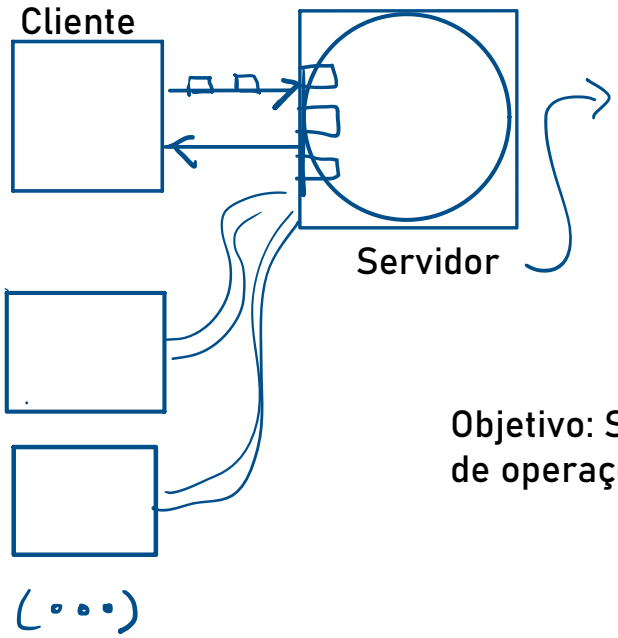
Aula PL #02

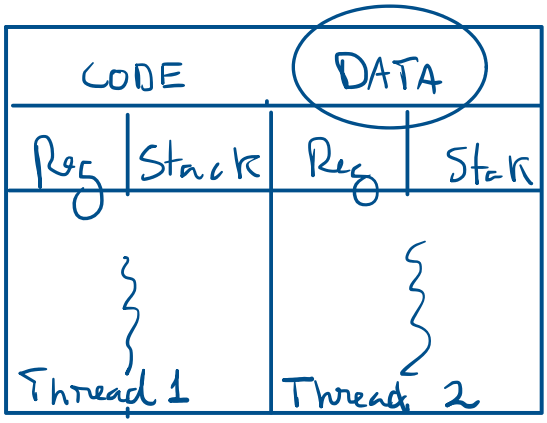
26 de setembro de 2023 14:11

(Troca de mensagens)



Objetivo: Simular invocação concorrente de operações.

Processo - execução num determinado programa, espaço próprio de endereçamento.
Não têm memória partilhada.

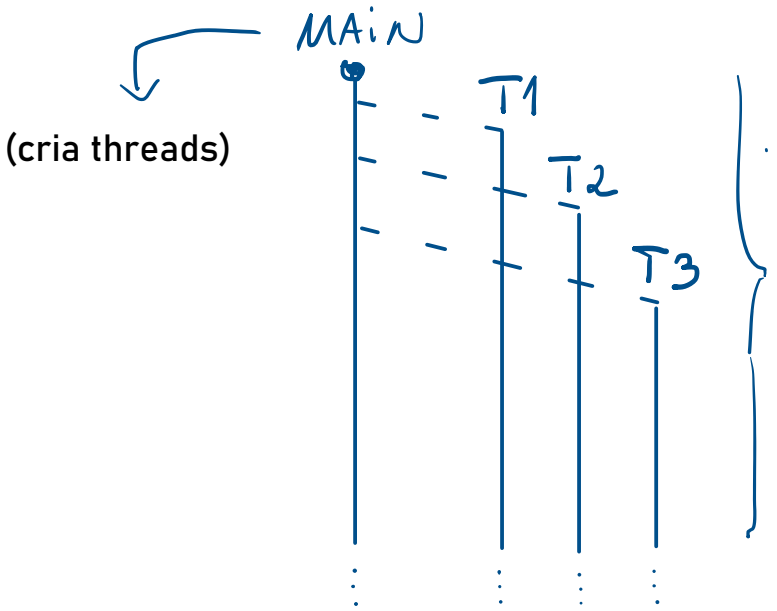


THREADS
↳ Memória partilhada
Mais leves do que processos.

1. N Threads (N=10)
↳ escreve os números de 1 a I

Executam sem sincronização, de forma independente.

Por isso, o resultado pode ser não determinístico.
"Não há uma ordem no output".



(cria threads)

execução concorrente (fios de execução independentes)

Exercício 2

```
Bank bank = new Bank();
1 usage  new *
class DepositRunnable implements Runnable {
    new *
    public void run() {
        for (int i = 0; i < 1000; i++) {
            bank.deposit( value: 1000);
        }
    }
}

int N = 10;
Thread[] threads = new Thread[N];

for (int i = 0; i < N; i++) {
    // threads[i] = new Thread(new Increment());
    threads[i] = new Thread(new DepositRunnable());
}
```

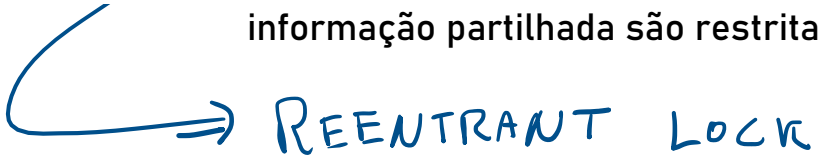
Output:

```
Balance: 5453000
```

≠ 1 000 000

Quando uma thread lê um valor, esse pode já ter sido alterado por outra thread, mas não atualizado. Por esse motivo, as zonas de código de alteração de informação partilhada são restritas.

informação partilhada são restritas.

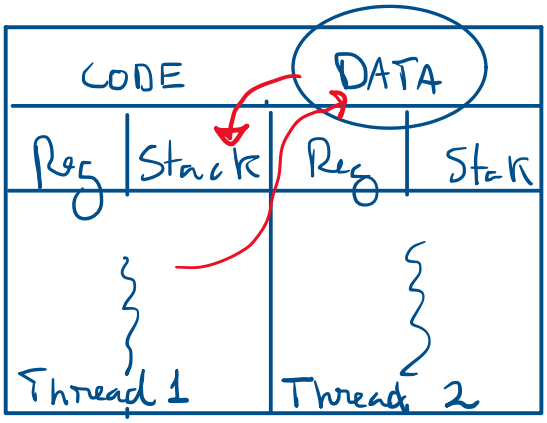


Singleon pattern adicionado: (não era necessário, a instância banco poderia ser passada no construtor)

```java
public class Bank {

    3 usages
    private static Bank instance;

    2 usages  new *
    public static Bank getInstance() {
        if (instance == null) {
            instance = new Bank();
        }
        return instance;
    }
```

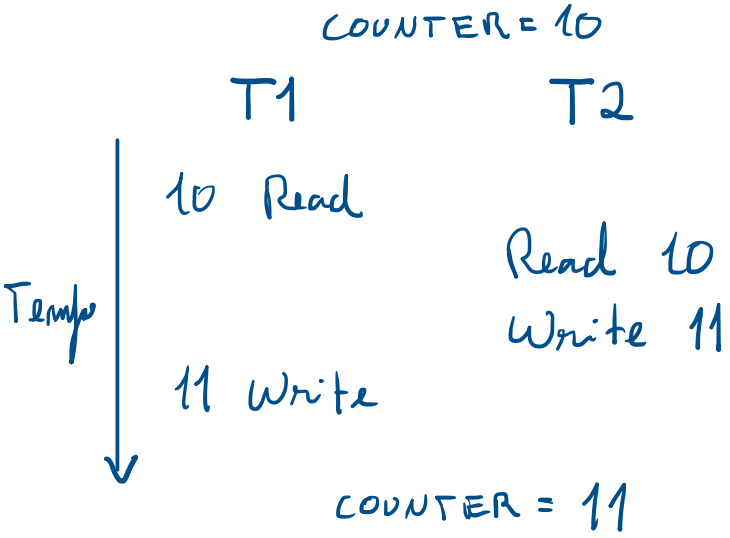
```java
public class Deposit implements Runnable {
    3 usages
    private final ReentrantLock lock;

    1 usage  new *
    public Deposit(ReentrantLock lock) {
        this.lock = lock;
    }

    new *
    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            lock.lock();
            try {
                Bank.getInstance().deposit( value: 100);
            } finally {
                lock.unlock();
            }
        }
    }
}
```

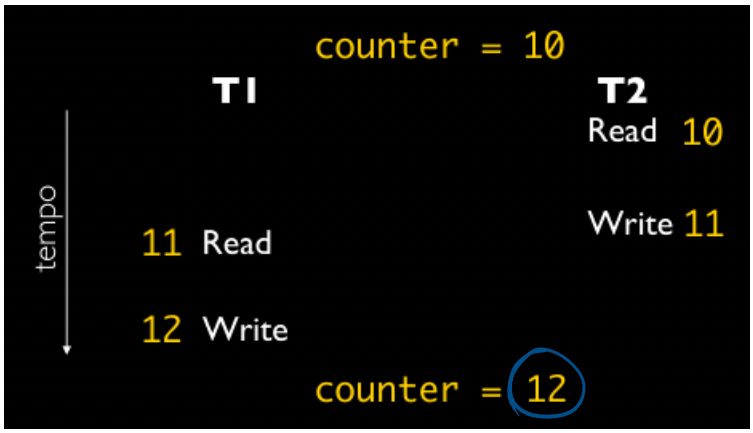
(Mas não me parece eficiente.)



t = this.counter;
t = t + 1;
this.counter = t

Passos podem acontecer; Threads concorrentes podem interferir.





⇒ Necessidade de haver secções críticas.

Exclusão mútua é a propriedade que garante que dois processos ou threads não acedem simultaneamente a um recurso partilhado.

Uma secção crítica é uma parte do programa onde os recursos partilhados são acedidos.