iscte
INSTITUTO UNIVERSITÁRIO DE LISBOA

emprego digital

Módulo 4: Laboratório de Programação

Aula 2
Sincronização e Coordenação; Problemas de Produtor Consumidor

UPskill

Sincronização de Threads

As *Threads* partilham memória para que a comunicação seja baseada em referências partilhadas.

Isto é uma forma muito eficaz de comunicar mas está sujeita a dois tipos de erros:

- Interferência entre Threads;
- Erros de inconsistência de memória.

Comunicação entre Threads

Pode ser problemático devido ao fenómeno de "***condição de corrida***" entre *threads* que tentam aceder aos **mesmos dados** ao **mesmo tempo**.

Uma *condição de corrida* é um problema quando a ordem de execução de duas *threads* podem afectar o valor de uma determinada variável ou do resultado de execução do vosso programa.

Exemplo com uma *Stack*

```
public class Stack {

    private int pos = 0;

    private int stack[] = new int[100];

    public void push(int i) {
        pos++;
        stack[pos] = i;
    }
    public int pop() {
        if (pos > 0) {
            return stack[pos--];
        } else return 0;
    }
    public int peek() {
        return stack[pos];
    }
}
```

Operação não-atómica (3 passos)

1. Ler *pos* da memória
2. Incrementar *pos*
3. Guardar *pos* em memória

Exemplo com uma *Stack* (cont.)

Se duas *Threads* (A e B) ambas fizerem um *push* na **mesma Stack**, ao **mesmo tempo**, com valores diferentes (7 e 4), teríamos dois resultados possíveis:

Resultado Coerente:

A) pos++ (pos = 1)

A) pilha[pos] = 7

B) pos++ (pos = 2)

B) pilha[pos] = 4

Resultado Incoerente:

A) pos++ (pos = 1)

B) pos++ (pos = 2)

A) pilha[pos] = 7

B) pilha[pos] = 4

- pos[1] sem valor
- pos[2] = 4
- valor 7 da thread A perdido

Interferência entre Threads

Interferência pode acontecer quando as operações de duas ou mais *threads* não são atómicas.

Operações **não-atómicas** podem ser subdivididas num **conjunto de operações atómicas**. Isto significa que operações atómicas podem estar a ser intercaladas ou paralelizadas entre threads diferentes, o que faz com que haja acesso/escrita em estados intermediários ou incompletos.

Erros de Inconsistência de dados

Acontece quando duas ou mais *threads* têm versões diferentes dos mesmos dados:

Thread A:	**Thread B**:
pos++;	int a = pos;

O resultado depende da ordem de acesso aos dados (pos).

É importante estabelecer ordem na execução de diferentes operações de *threads* distintas.

Sincronização e Operações Atómicas

Operações Atómicas: Instruções que podem ser executadas por apenas uma única *thread* a qualquer altura;

Secção crítica: Sequência de instruções que têm de ser executadas atomicamente;

Synchronized: Um método ou parte do código que deve ser tratado como crítico e deve ser executado atomicamente.

Sincronização

O mecanismo de sincronização funciona ao **suspender** todas as outras *threads* durante a execução de código sincronizado.
Isto não é eficiente se um método for complexo e demorar tempo a executar, e também suspende *threads* que poderiam não interferir.

Uma solução que só bloqueia *threads* que possam interferir com um determinado método é demasiado complexa porque não é possível prever que *threads* irão fazer o quê no futuro.

Cadeados

Solução para o mecanismo de sincronização no Java.

Cadeados permitem bloquear secções críticas. Antes de tentar executar uma secção crítica de código, a *thread* verifica se o **cadeado está aberto**. Se sim, então a *thread* **fecha o cadeado** dessa secção e executa.

Se o cadeado está fechado, tem de esperar que a *thread* que está a executar essa secção crítica acabe.

Mutex: Exclusão Mútua

Mecanismo de cadeados básico em programação multi-*thread*.

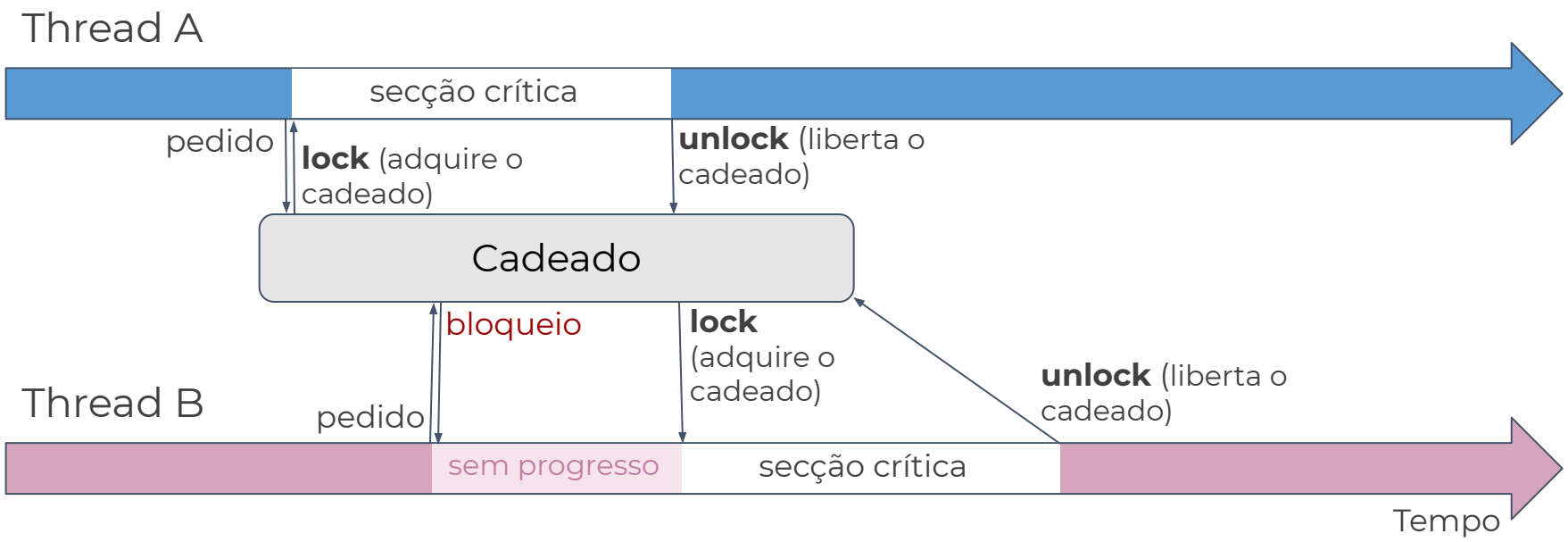
Um cadeado *mutex* tem duas operações básicas: ***Lock*** (fechar) e ***Unlock*** (abrir).

Só uma *thread* é que pode ter o cadeado a qualquer momento.

Se uma *thread* tentar fechar um cadeado *mutex* que já está fechado, fica suspenso à espera que o cadeado abra.

Se múltiplas *threads* estão à espera que um cadeado abra, só um (habitualmente por ordem, mas nem sempre!) é que adquire o cadeado.

Mutex: Exclusão Mútua



Cadeados Intrínsecos

No Java, todos os objetos têm um cadeado intrínseco, isto é, inerente ao próprio objeto.

Por convenção, uma *thread* que necessite de acesso exclusivo e consistente aos dados de um objeto, tem de adquirir o cadeado intrínseco desse objeto antes de aceder aos dados, e libertar quando acabar.

É da responsabilidade do programador definir os métodos ou secções de código sujeitas a este cadeado. No Java usamos a chave "**synchronized**".

Exemplo Synchronized (*Stack*)

```
public class Stack {

    private int pos = 0;

    private int stack[] = new int[100];

    public synchronized void push(int i) {
        pos++;
        stack[pos] = i;
    }
    public synchronized int pop() {
        if (pos > 0) {
            return stack[pos--];
        } else return 0;
    }
    public synchronized int peek() {
        return stack[pos];
    }
}
```

Adquire Cadeado (Lock)

Liberta Cadeado (Unlock)

Exemplo Bloco Synchronized

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}

public void addEmployee(String name){
    synchronized(company) {
        employeeCount++;
    }
    nameList.add(name);
}
```

Adquire cadeado do **próprio objeto**

Liberta **próprio** cadeado

Adquire cadeado do **objeto "company"**

Liberta cadeado da **"company"**

Cadeados Externos - Interface *Lock*

`void lock()`	Adquire o cadeado
`boolean tryLock()`	Adquire o cadeado, só se estiver livre no momento da invocação
`void unlock()`	Liberta o cadeado

```
Lock l = ...;
l.lock();
try {
   //acesso a dados protegidos pelo cadeado "l"
} finally {
   l.unlock();
}
```

Exercício 1

Incremento e decremento concorrente

Crie uma classe “Contador” que contenha os seguintes métodos:

- `void incrementar();`
- `void decrementar();`
- `int consulta();`

De seguida programe uma classe (*thread*) onde cada instância incrementa o valor do contador e mostra na consola o nome do processo e o valor após o incremento. Repetir esta acção **20** vezes.

Teste o programa com **4** threads e **1** contador.

Coordenação

Já vimos como sincronizar *threads* para evitar interferência e como a ordem de acesso a recursos tem um papel importante no resultado do programa: Se não tivermos cuidado, podemos encontrar inconsistências de dados/na memória.

Acesso seguro a recursos partilhados podem **necessitar** de meios mais sofisticados de coordenação:

- Problemas **produtor-consumidor**.

Produtor de Números

```
public class Producer extends Thread {
    private NumberContainer numberContainer;
    private int id;

    public Producer(NumberContainer nc, int id) {
        numberContainer = nc;
        this.id = id;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            numberContainer.put(i);
            System.out.println("Producer #" +
                    this.id + " put: " + i);
        }
    }
}
```

Consumidor

```
public class Consumer extends Thread {
    private NumberContainer numberContainer;
    private int id;

    public Consumer(NumberContainer nc, int id) {
        numberContainer = nc;
        this.id = id;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = numberContainer.get();
            System.out.println("Consumer #" +
                    this.id + " got: " + i);
        }
    }
}
```

Recurso Partilhado

```
public class NumberContainer {
    private int contents;

    public synchronized void put(int value) {
        contents = value;
    }

    public synchronized int get() {
        return contents;
    }
}
```

Sem Coordenação

O produtor e o consumidor partilham um recurso (NumberContainer).

Se assumirmos que o produtor é um pouco mais rápido do que o consumidor:

```
...
Consumer #1 got: 3
Producer #1 put: 4
Producer #1 put: 5
Consumer #1 got: 5
...
```

O valor "**4**" não foi consumido.

Coordenação entre *Threads*

Num problema produtor-consumidor as *threads* partilham dados; o produtor produz dados e o consumidor faz algo com esses dados.

As duas (ou mais!) threads têm de coordenar, **não só sincronizar**, o acesso a esse recurso partilhado.

O consumidor não deveria tentar consumir dados antes do produtor produzir dados, e o produtor não deverá produzir mais dados enquanto o consumidor não consumir o anterior.

Coordenação entre *Threads*

A forma mais comum de coordenar *threads* é testar a condição **dentro** de uma secção sincronizada.

Se a condição não for verdadeira, a *thread* liberta o cadeado e espera. Outras *threads* podem então adquirir o cadeado de forma a aceder à secção sincronizada relacionada.

No Java este mecanismo é implementado nos métodos ***wait***, ***notify*** e ***notifyAll***.

Recurso Partilhado (Nova Versão)

```
public class NumberContainer {
   private int contents;
   private boolean available = false;
   public synchronized void put(int value) {
       while(available) {
           try {
             wait();
           } catch (InterruptedException e) {}
       }
       contents = value;
       available = true;
       notifyAll();
   }

   ...
```

Coloca a *thread* em espera e liberta o cadeado

Thread saiu da espera e adquiriu o cadeado

Notifica *threads* em espera

Recurso Partilhado (Nova Versão)

```
    ...

    public synchronized int get() {
        while(!available) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        available = false;
        notifyAll();
        return contents;
    }

}
```

Coloca a *thread* em espera e liberta o cadeado

Thread saiu da espera e adquiriu o cadeado

Notifica *threads* em espera

Método *wait();*

wait - Causa a *thread* atual ficar em espera até que outra *thread* invoque o método **notify()** ou **notifyAll()** para este objeto.

A *thread* atual tem de possuir o monitor do próprio objeto (cadeado intrínseco).
A *thread* liberta a posse deste monitor e fica à espera até que outra *thread* notifique uma ou todas as *threads* à espera do monitor deste objeto.

Depois de ser notificado e antes de retornar, a *thread* tem de voltar a obter posse do monitor.

Método *wait();*

Interrupções e “despertares espúrios” são possíveis e este método deve ser sempre utilizado dentro de um ciclo:

```
synchronized(obj) {
   while(<condição não é verdadeira>) {
       obj.wait();
       ... //executar acção apropriada para a condição
   }
}
```

Este método só pode ser chamado por uma *thread* que possui o monitor deste objeto (dentro de um método ou bloco sincronizado)

Método *wait();*

wait(long timeout) - Espera para ser notificada. Se a notificação não acontecer dentro dos milissegundos definidos, o método retorna.

wait(0) - Equivalente ao ***wait()***;

Quando o método retorna não há informação sobre se foi por *timeout* ou por uma notificação, pelo que o programador deve ter em conta o tempo passado para poder distinguir,

Exemplo

```
public synchronized boolean get(int timeout) {
    long startTime = System.currentTimeMillis();
    try {
        while (!condition) {
            long timeLeft = timeout - (System.currentTimeMillis() - startTime);
            wait(timeLeft);
            if(System.currentTimeMillis() - startTime > timeout){
                return false;
            }
        }
        condition = false;
    } catch (InterruptedException e) {}
    return true;
}
```

Métodos *notify()* e *notifyAll()*

notify - Acorda uma *thread* que está atualmente em espera (se existir);

notifyAll - Acorda todas as *threads* que estão atualmente em espera (se existirem);

A *thread* atual tem de possuir o monitor do próprio objeto (cadeado intrínseco). A *thread* liberta a posse deste monitor e fica à espera até que outra *thread* notifique uma ou todas as *threads* à espera do monitor deste objeto.

Depois de ser notificado e antes de retornar, a *thread* tem de voltar a obter posse do monitor.

Exercício 2

Considerem um restaurante que tem um chefe de cozinha e um empregado de mesa.
O empregado tem de esperar que o chefe prepare a refeição. Quando o chefe tem a refeição pronta, avisa o empregado, que entrega a mesma e volta a ficar à espera.

Desenhar um programa que represente as duas classes (coordenadas). No teste assume-se que o chefe produz uma refeição por segundo e devem ser realizadas 10 no total.

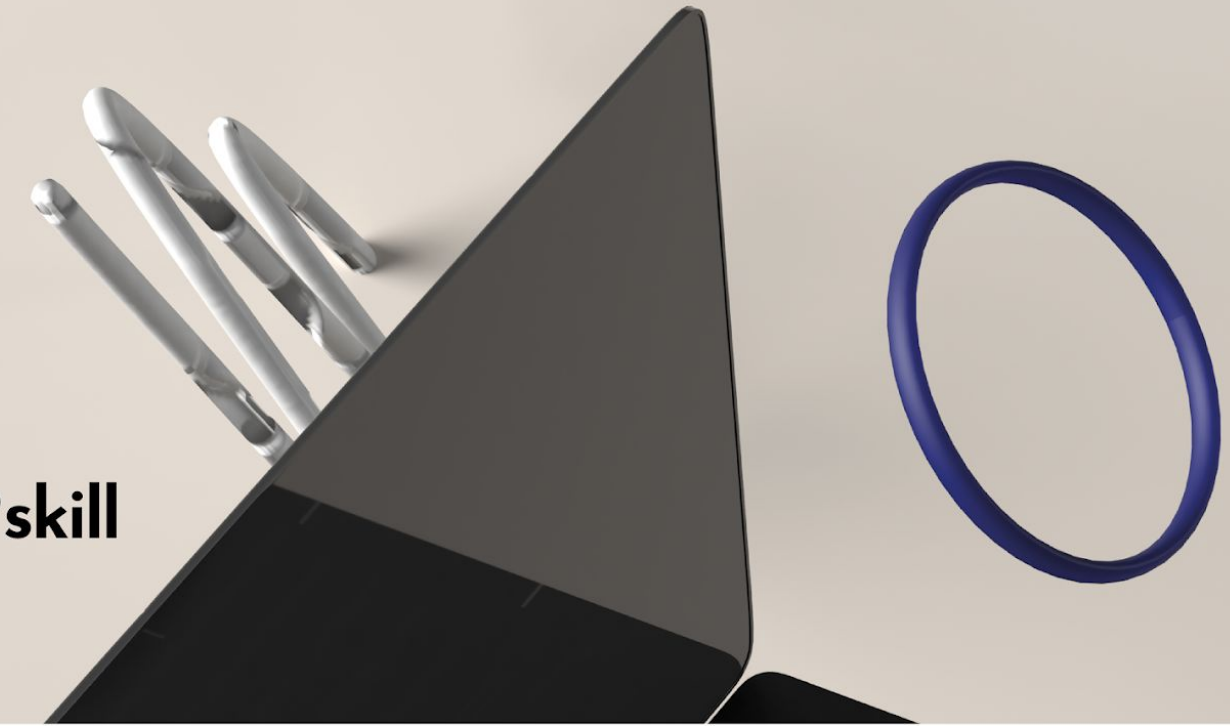
Exercício 3

Criar duas *threads*. A primeira *thread* deve imprimir os números de 1 a 52. A segunda *thread* deve imprimir o alfabeto A a Z.
A ordem de impressão na consola deverá ser:

```
1 2 A 3 4 B 5 6 ... Y 51 52 Z
```

Ou seja, a cada dois números, uma letra.

O futuro profissional começa aqui



iscte INSTITUTO UNIVERSITÁRIO DE LISBOA

emprego digital

UPskill