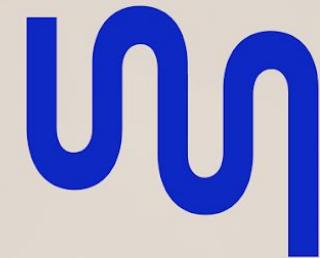




iscte

INSTITUTO
UNIVERSITÁRIO
DE LISBOA



emprego
digital

Módulo 4: Laboratório de Programação

Aula 4

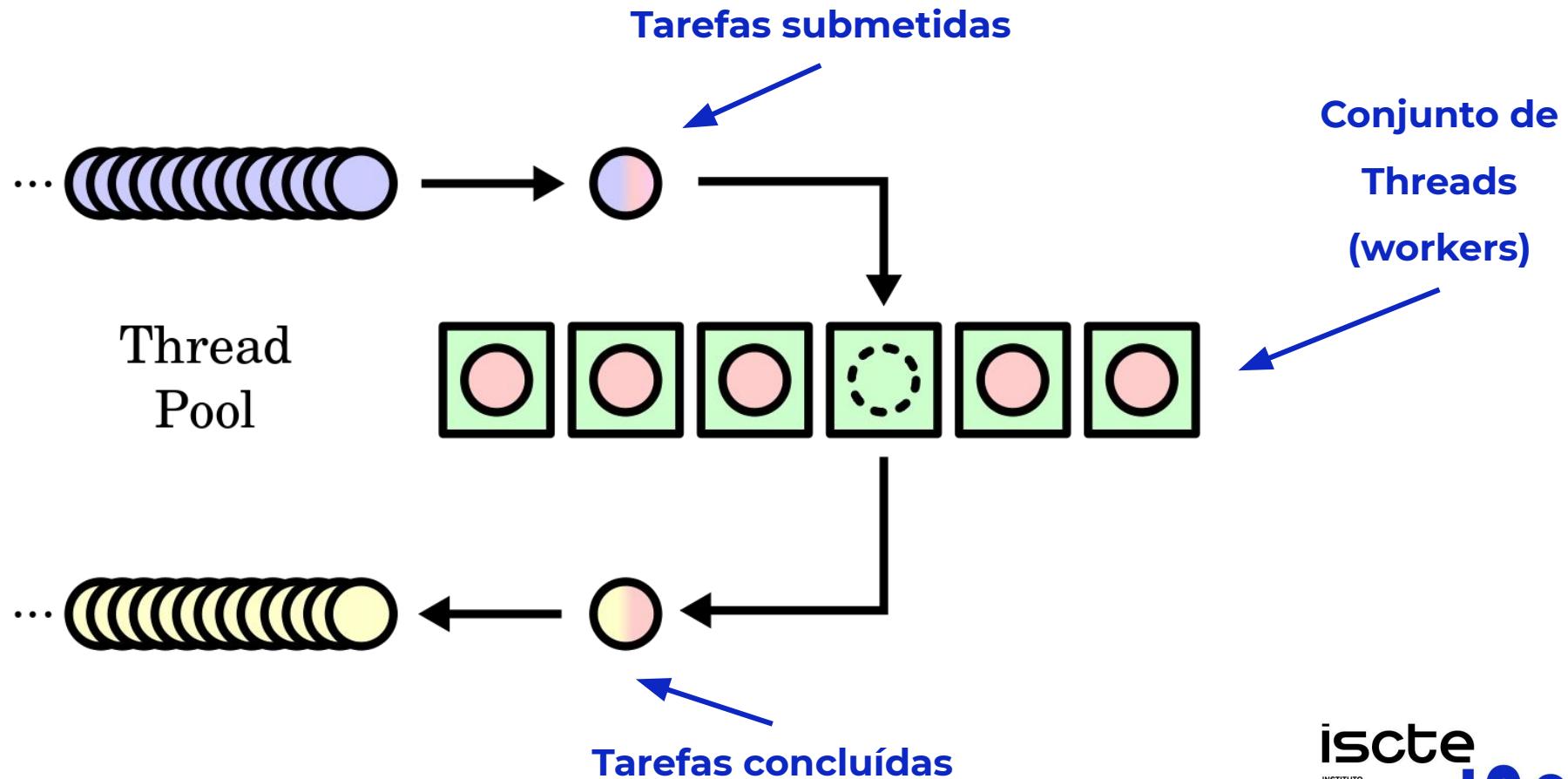
Thread Pool; Resolução de Exercícios



O que são Thread Pools

- Uma thread pool é simplesmente uma **estrutura constituída por um conjunto de threads (workers), que executam várias tarefas diferentes** (tantas quanto quisermos, basta submeter a tarefa à pool).
- É constituída por uma fila de tarefas e um conjunto de threads.
- As threads vão à fila de tarefas retirar a próxima tarefa para a executar.
 - Caso não existam tarefas na fila, as threads devem ficar à espera (em wait) até que sejam submetidas novas tarefas.
 - As tarefas devem implementar a interface Runnable. O propósito da tarefa deve ser **implementado no método run()**.

Thread Pool



Ao usar Thread Pool, as nossas tarefas deixam de ser **Threads** e passam a ser **Runnables**.

Isto acontece porque as threads que vão executar a nossa tarefa estão dentro da *thread pool*. Assim, não precisamos de criar uma *Thread*, mas apenas de definir um *Runnable*.

Thread Pool - Vantagens

- Esta estrutura apresenta algumas vantagens em relação à criação normal de threads:
 - As threads dentro da *thread pool* (*workers*) são recicladas e executam várias tarefas, **evitando a sobrecarga da criação e destruição de threads** específicas com um tempo de vida curto.
 - O número de threads trabalhadoras pode ser **limitado à capacidade real de processamento paralelo do nosso processador**, evitando a sobrecarga de ter mais *threads* do que o nosso sistema comporta correr em paralelo.

Thread Pool - Exemplo

```
public class AminhaTarefa implements Runnable {  
    @Override  
    public void run() {  
        //codigo da tarefa...  
    }  
  
    public static void main(String[] args) {  
        AminhaTarefa t1 = new AminhaTarefa();  
        AminhaTarefa t2 = new AminhaTarefa();  
  
        //Criação de uma thread pool com 2 threads  
        ExecutorService pool = Executors.newFixedThreadPool(4);  
  
        pool.submit(t1); //Submissão da tarefa t1 à thread pool  
        pool.submit(t2); //Submissão da tarefa t2 à thread pool  
    }  
}
```

Usamos um Runnable em vez de uma Thread, pois as Threads vão estar dentro da Thread Pool, aqui apenas queremos definir a tarefa

Número de Threads dentro da Thread Pool.

Thread Pool - Métodos

A Thread Pool tem alguns métodos muito úteis para a gestão das Threads:

- **submit(Runnable r)**: submete uma tarefa (Runnable) para ser executada por uma das threads da Thread Pool.
- **awaitTermination(long timeout, TimeUnit unit)**: fica em espera até todas as tarefas terminarem a sua execução, ou o timeout passar.
- **isTerminated()**: indica se todas as tarefas submetidas já foram terminadas - não fica em espera.
- **shutdown()**: termina as tarefas atuais, mas não aceita tarefas novas, encerrando a *thread pool* de forma elegante.

Exercício 1

O Professor Emanuel de portugues pediu aos formandos do programa Upskills que desenvolvessem um programa que permitisse capitalizar a primeira letra de cada palavra de um texto. O texto deverá ser lido de um ficheiro .txt.

Para paralelizar o processamento no caso de textos muito longos, o tratamento das palavras do texto deverá ser feito por *threads* (*cada uma* deverá processar *no máximo 200 palavras*), com recurso a uma *thread pool* de 6 *threads*, a capacidade máxima de processamento paralelo do computador (six-core), onde o programa irá correr.

Implemente o programa descrito.

Resolução exercício 1 - WordCapitalizeTask

```
public class WordCapitalizeTask implements Runnable {  
    private List<String> words;  
  
    public WordCapitalizeTask(List<String> words) {  
        this.words = words;  
    }  
  
    @Override  
    public void run() {  
        for (String word : words) {  
            word = word.substring(0, 1).toUpperCase() + word.substring(1);  
            System.out.println(word + " foi capitalizada");  
        }  
    }  
}
```

Resolução exercício 1 - Main

```
public static void main(String[] args) {
    LinkedList<String> words = new LinkedList<String>();
    LinkedList<WordCapitalizeTask> tasks = new LinkedList<WordCapitalizeTask>();

    try {
        Scanner s = new Scanner(new File("..."));
        s.useDelimiter(" ");
        while (s.hasNext()) {
            String word = s.next();
            words.add(word);
        }
    } catch (FileNotFoundException e) {
        System.out.println("Ficheiro não encontrado");
    }

    ExecutorService pool = Executors.newFixedThreadPool(6);

    for (int i = 0; i < words.size(); i = i + 200) {
        List<String> subl = null;
        if (i + 200 < words.size()) {
            subl = words.subList(i, i + 200);
        } else {
            subl = words.subList(i, words.size());
        }
        WordCapitalizeTask w = new WordCapitalizeTask(subl);
        pool.submit(w);
    }
}
```

Exercício 2

Pretende desenvolver-se um jogo chamado “Lotaria UPskill” onde um conjunto de bolas são lançadas. Deve haver uma bola por cada aluno da turma. Assim, 20 bolas no total. Cada bola começa com o valor 0 e a cada segundo soma-lhe um valor aleatório entre 0 e 100. As bolas correm por 35 iterações (35 segundos).

O sistema aguarda que todas as bolas terminem e verifica qual o maior valor. Esse será o valor premiado!

Para não sobrecarregar o sistema com o processamento simultâneo de 20 *threads*, implemente o programa descrito com recurso a uma *thread pool*.

Resolução exercício 2 - Ball

```
public class Ball implements Runnable {  
  
    private int value;  
    private int id;  
  
    public Ball(int id){  
        this.id=id;  
    }  
  
    @Override  
    public void run() {  
        for(int i =0; i<35; i++) {  
            try {  
                sleep(1000);  
            } catch (InterruptedException e) {}  
            value += new Random().nextInt(101);  
            System.out.println(" A bola " + id + " passou a ter o valor " + value);  
        }  
    }  
    public int getValue() {  
        return value;  
    }  
  
    public int getId() {  
        return id;  
    }  
}
```

Resolução exercício 2 - Main

```
public static void main(String[] args) {
    List<Ball> balls = new LinkedList<Ball>();

    ExecutorService pool = Executors.newFixedThreadPool(6);
    for (int i = 0; i < 20; i++) {
        Ball b = new Ball(i);
        balls.add(b);
        pool.submit(b);
    }
    try {
        pool.shutdown();
        pool.awaitTermination(70, TimeUnit.SECONDS);
    } catch (InterruptedException e) {}
    Ball vencedor = balls.get(0);
    for (Ball b : balls) {
        if (vencedor.getValue() < b.getValue()) {
            vencedor = b;
        }
    }
    System.out.println("O vencedor é " + vencedor.getId() + " com " + vencedor.getValue());
}
```

Exercício 3

Considere um programa com cinco produtores, cinco consumidores e um recurso partilhado. Neste programa os produtores produzem números e os consumidores consomem números de acordo com determinadas regras. O recurso partilhado tem a capacidade de guardar dois números de cada vez.

Os produtores produzem 10 números aleatórios entre 1 e 30 que vão sendo colocados no recurso partilhado. Os consumidores imprimem 5 números, cada um deles correspondente à soma de dois números consumidos no recurso partilhado.

Assim, o programa deve garantir as seguintes regras:

- O recurso partilhado só guarda dois números de cada vez.
- Os produtores só conseguem colocar um número caso o recurso tenha menos de dois números. Se o recurso tiver dois números terão que esperar por uma notificação dos consumidores a informar que o recurso ficou vazio.
- Os consumidores só conseguem retirar um par de números. Os dois números são somados e o resultado da soma é devolvido. Se o recurso não tiver os dois números, os consumidores devem esperar por uma notificação dos produtores.

Resolução Exercício 3

Produtor

```
public class Produtor extends Thread {  
  
    private Mesa m;  
    private int id;  
  
    public Produtor(Mesa m, int id){  
        this.m=m;  
        this.id=id;  
    }  
  
    @Override  
    public void run() {  
        for(int i = 0; i<10; i++){  
            int number= new Random().nextInt(30) +1;  
            try {  
                m.putNumero(number);  
                System.out.println("Produtor: " + id + " acabou de criar um novo número: " + number);  
            } catch (InterruptedException e) {}  
        }  
    }  
}
```

Consumidor

```
public class Consumidor extends Thread {  
  
    private Mesa m;  
    private int id;  
  
    public Consumidor(Mesa m, int id){  
        this.m=m;  
        this.id=id;  
    }  
  
    @Override  
    public void run() {  
        for (int i =0; i<5; i++){  
            try {  
                int result = m.getSoma();  
                System.out.println("Consumidor: " + id + " obteve a soma " + result);  
            } catch (InterruptedException e) {}  
        }  
    }  
}
```

Resolução Exercício 3

Recurso Partilhado: Mesa

```
public class Mesa {  
  
    public static final int MAX_NUMEROS = 2;  
    private LinkedList<Integer> listaNumeros = new LinkedList<Integer>();  
  
    public Mesa() {  
    }  
  
    public synchronized void putNumero(int number) throws InterruptedException {  
        while (listaNumeros.size() >= MAX_NUMEROS) {  
            wait();  
        }  
        listaNumeros.add(number);  
        notifyAll();  
    }  
  
    public synchronized int getSoma() throws InterruptedException {  
        while (listaNumeros.size() < MAX_NUMEROS) {  
            wait();  
        }  
        int p = listaNumeros.removeFirst();  
        int u = listaNumeros.removeFirst();  
        notifyAll();  
        return p + u;  
    }  
}
```

Main

```
public class Main {  
  
    public static void main(String[] args) {  
  
        Mesa m = new Mesa();  
        Produtor[] pr = new Produtor[5];  
        Consumidor[] cs = new Consumidor[5];  
  
        for (int i = 0; i < 5; i++) {  
            pr[i] = new Produtor(m, i);  
            cs[i] = new Consumidor(m, i);  
        }  
  
        for (int i = 0; i < 5; i++) {  
            pr[i].start();  
        }  
        for (int i = 0; i < 5; i++) {  
            cs[i].start();  
        }  
    }  
}
```

Exercício 4

Considere que uma empresa de electricistas pretende desenvolver uma aplicação que permite gerir o processamento das suas empreitadas e dos seus funcionários. A empresa efectua trabalhos de reparação de instalações eléctricas de pequena dimensão. Tem actualmente 10 electricistas e uma telefonista. A telefonista tem de inserir 15 tarefas num reservatório (recurso partilhado) que apenas permite guardar 5 tarefas. Caso o reservatório esteja cheio a telefonista espera por uma vaga para colocar a tarefa. Os funcionários retiram uma tarefa qualquer do reservatório. Caso não haja tarefas devem esperar. Após terem retirado a tarefa com sucesso procedem à reparação e voltam à empresa para pedir uma nova tarefa (15 segundos).

Implemente as classes necessárias para resolver este problema.

Resolução Exercício 4

Produtor: Telefonista

```
public class Telefonista extends Thread{  
  
    private Reservatorio r;  
  
    public Telefonista (Reservatorio r){  
        this.r=r;  
    }  
  
    @Override  
    public void run() {  
        for(int i =0; i<15; i++){  
            Tarefa t = new Tarefa("Nova Tarefa: "+ i);  
            try {  
                r.putTarefa(t);  
                System.out.println("Telefonista acabou de inserir "  
+ t.getNome());  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

Consumidor: Electricista

```
public class Electricista extends Thread{  
    private Reservatorio r;  
    private int id;  
  
    public Electricista(Reservatorio r, int id) {  
        this.r = r;  
        this.id=id;  
    }  
  
    @Override  
    public void run() {  
        while(true){  
            try {  
                Tarefa t = r.getTarefa();  
                System.out.println("Electricista " +id + " removeu " +  
t.getNome());  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            try {  
                sleep(15000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

Resolução Exercício 4

Recurso Partilhado: Reservatorio

```
public class Reservatorio {  
  
    public static final int MAX_TAREFAS=5;  
    private LinkedList<Tarefa> listaTarefas = new LinkedList<Tarefa>();  
  
    public Reservatorio(){  
    }  
  
    public synchronized void putTarefa (Tarefa t) throws InterruptedException  
{  
        while(listaTarefas.size() >= MAX_TAREFAS){  
            wait();  
        }  
        listaTarefas.add(t);  
        notifyAll();  
    }  
  
    public synchronized Tarefa getTarefa () throws InterruptedException {  
        while(listaTarefas.size()==0){  
            wait();  
        }  
        Tarefa tarefa = listaTarefas.removeFirst();  
        notifyAll();  
        return tarefa;  
    }  
}
```

Main

```
public class Main {  
    public static void main(String[] args) {  
        Reservatorio r = new Reservatorio();  
        Electricista[] ele = new Electricista[10];  
  
        Telefonista t = new Telefonista(r);  
  
        for(int i =0; i<10; i++){  
            ele[i] = new Electricista(r, i);  
        }  
        t.start();  
        for(int i =0; i<10; i++){  
            ele[i].start();  
        }  
    }  
}
```

Exercício 5

Uma barbearia consiste numa sala de espera com 3 cadeiras e numa sala do barbeiro apenas com uma cadeira. Caso não existam clientes para serem atendidos o barbeiro deve ficar à espera.

Se um cliente entra na barbearia e todas as cadeiras estão ocupadas, então o cliente sai da loja, e regressa após um tempo entre 3 e 10 segundos. Se o barbeiro está ocupado, mas existem cadeiras disponíveis na sala de espera, o cliente senta-se numa das cadeiras livres. Se o barbeiro está inactivo, à espera de clientes, o cliente deve notificá-lo da sua presença.

Escreva um programa para coordenar o barbeiro e os clientes.

Exercício 6

Numa fábrica de automóveis existe um conjunto de robots que fabricam e montam os diferentes componentes que compõem um carro. Existem dois fabricantes de chassis (*FabricanteChassis*), quatro fabricantes de motores (*FabricanteMotor*) e dois *FabricanteRodas*. Os carros são montados numa sala que apenas tem capacidade para ter um carro de cada vez. Como a sala é muito pequena quando um *FabricanteChassis* pretende colocar o chassis do novo carro a sala tem de estar vazia. Os *FabricanteRoda* podem entrar na sala desde que já exista um chassis colocado e que ainda não tenha as quatro rodas montadas. Os *FabricanteMotor* podem entrar na sala desde que o chassis não tenha um motor já instalado e efetuar a sua instalação. Assim que o chassis tiver o motor, é retirado da sala e a sala fica pronta para receber um novo chassis. O objectivo deste exercício é fazer uma aplicação multithreaded que simula o processo de montagem do carro. Deve focar o seu esforço na coordenação das threads.

A classe *Main* é responsável por criar os objetos e iniciar as threads envolvidas. O *Main* deve deixar o processo de fabricação a correr durante 10 segundos e depois terminar todas as threads. Após todas as threads terem terminado o *Main* deve escrever para a consola o número de carros fabricados.



O futuro profissional começa aqui

