Sistemas Distribuídos Aplicados à Linguagem Java

Rogério Santos Pozza
pozza@cp.cefetpr.br

Roteiro

- Introdução
- Sistemas Distribuídos x Paralelos
- Características Sistemas Distribuídos x Paralelos
- Projetos de Sistemas Distribuídos
- Especificação de Processos e Tarefas
- Exercícios

Introdução

- Sistemas computacionais Paralelos e Distribuídos são largamente utilizados (redes bancárias, jogos, clusters, etc.)
- **Sistema Paralelo** - consiste de múltiplos processadores que se comunicam usando memória compartilhada
- **Sistemas Distribuídos** - mantém múltiplos processadores conectados via uma rede de comunicação (troca de mensagens). Ou ainda: "Coleção de computadores independentes que aparecem para o usuário como um único e simples sistema coerente" - *TANENBAUM*
- **Redes de Computadores** - "Coleção de computadores autônomos com o objetivo de compartilhar dados" - *TANENBAUM*

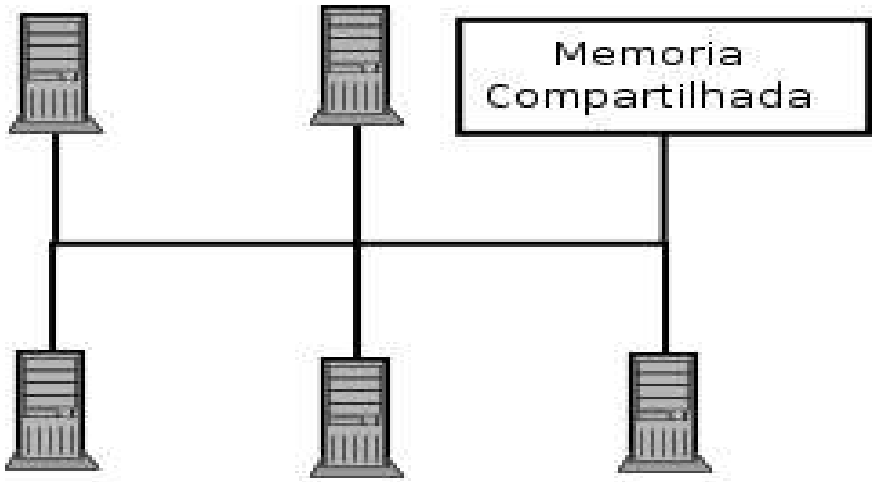
Introdução

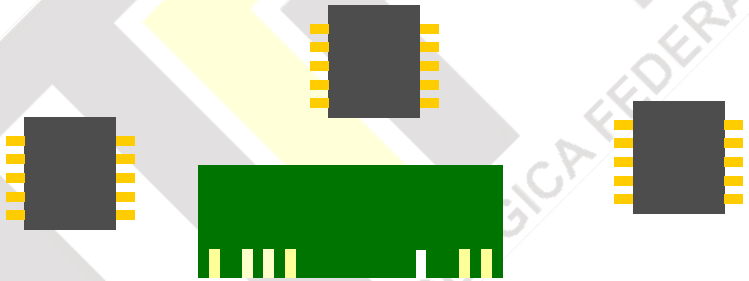
WatchPad

Introdução

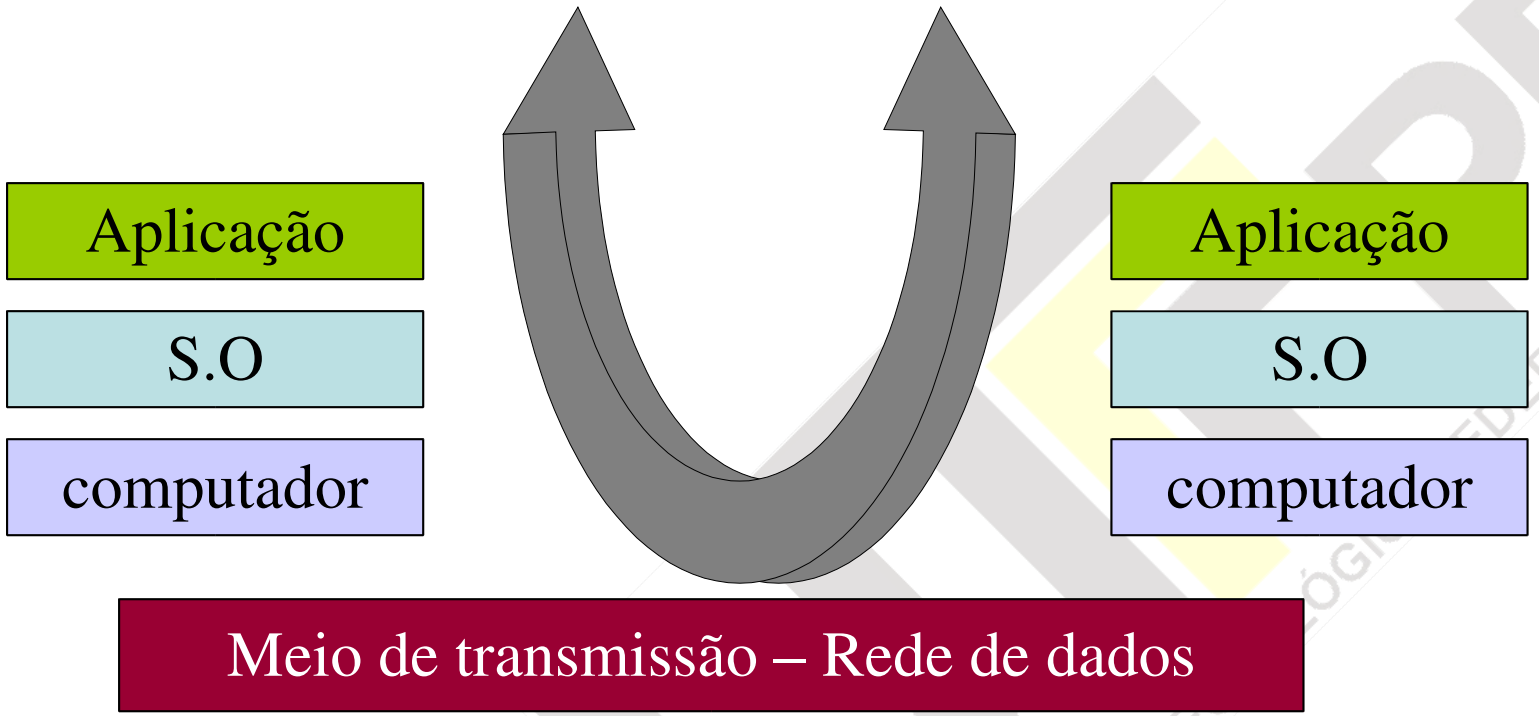
- Programação paralela e distribuída requer um conjunto de técnicas diferentes das utilizadas na programação seqüencial

- O foco do curso é nessas técnicas

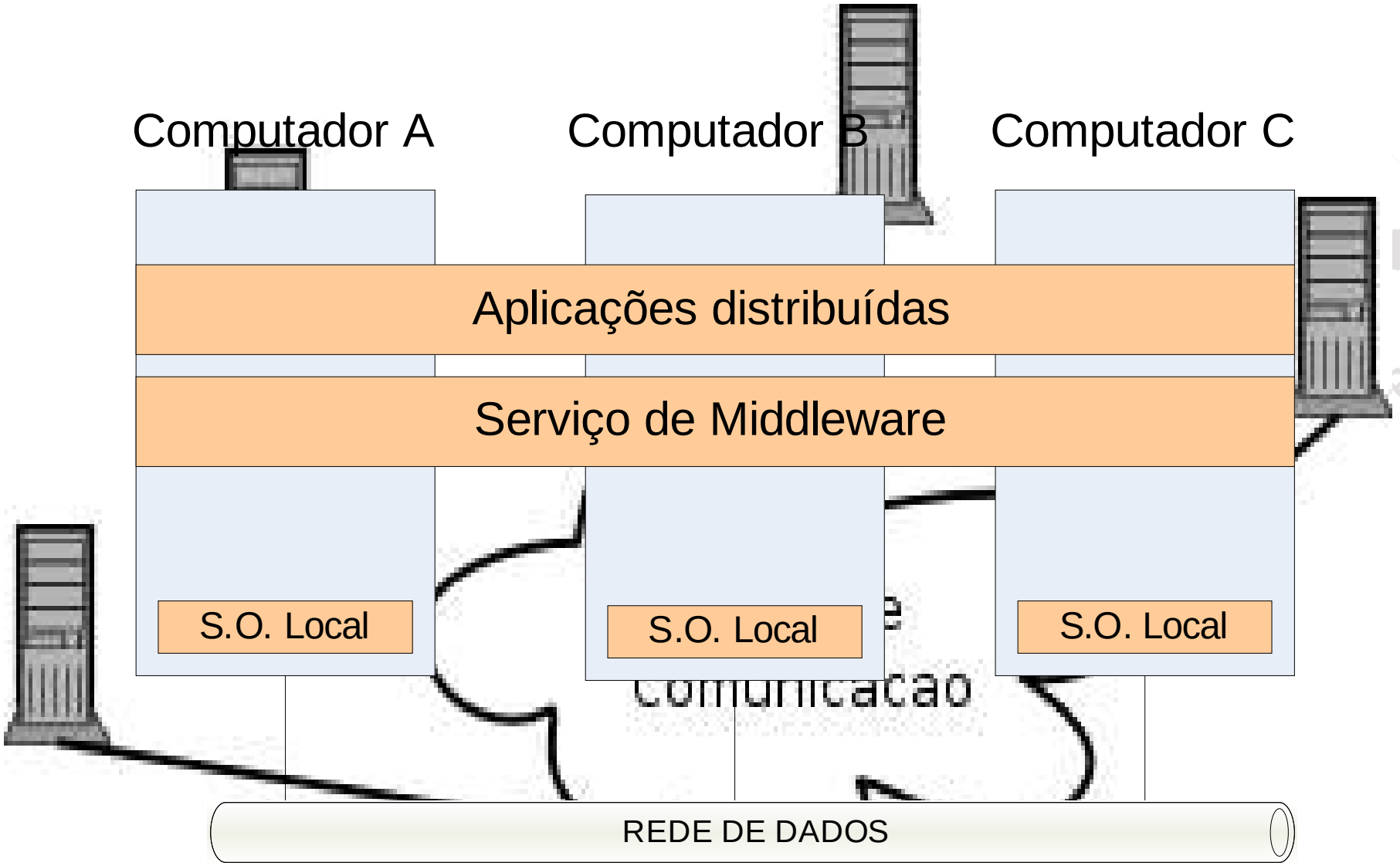




Sistema Paralelo



Redes de Computadores



Sistema Distribuído

Introdução – Sistemas Distribuídos x Paralelos

- A distinção entre sistemas distribuídos e paralelos, nesse curso, será apenas **lógica**
- Dado um sistema físico em que processadores têm memória compartilhada, torna-se fácil simular mensagens
- Do mesmo modo, dado um sistema em que processadores são conectados por uma rede, é possível simular memória compartilhada
- **Logo um hardware paralelo pode executar software distribuído e vice-versa**

Introdução – Sistemas Distribuídos x Paralelos

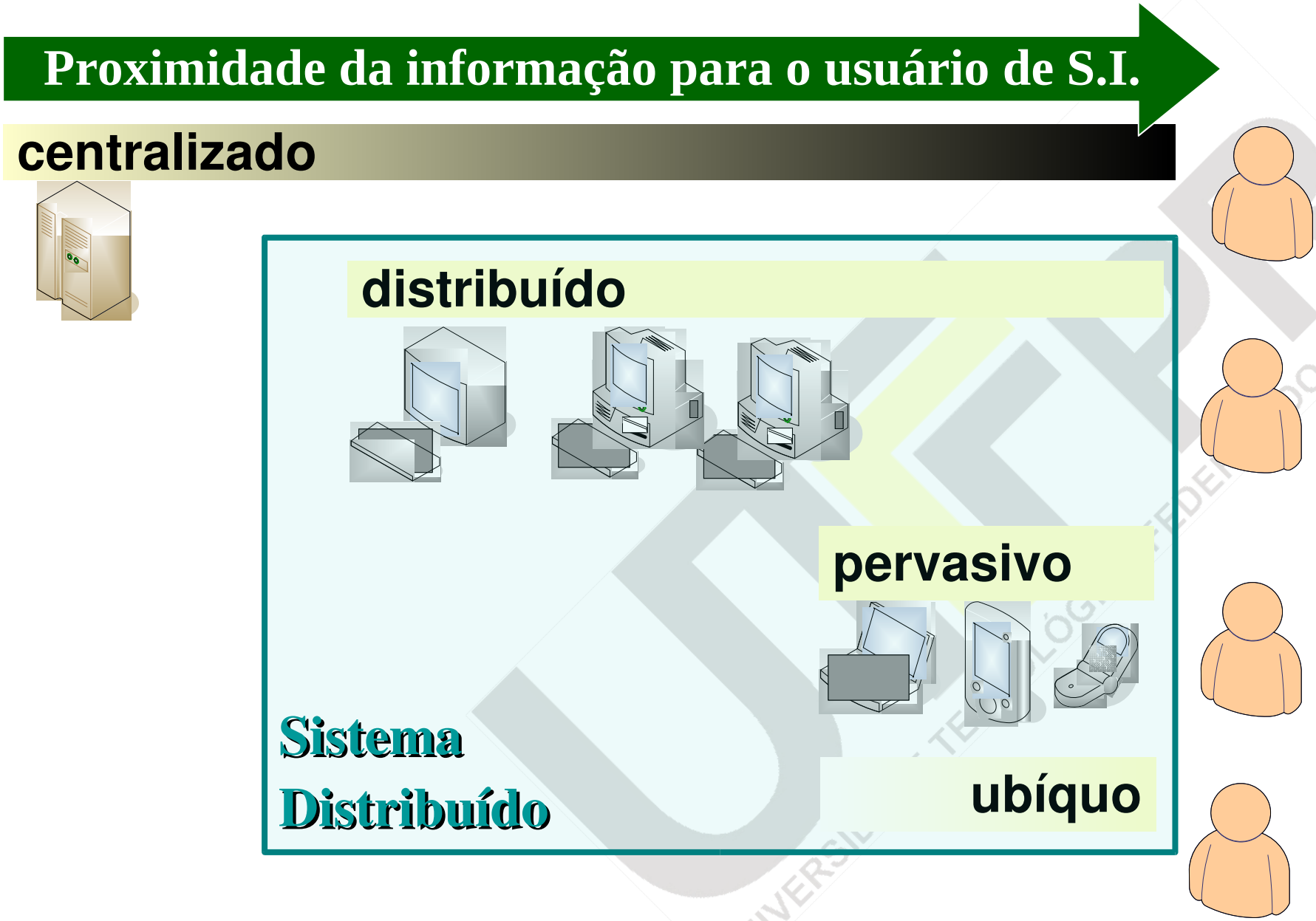
- Será construída aplicações em hardware paralelo ou distribuídos?

Será escrita aplicações assumindo memória compartilhada ou troca de mensagens?

- No nível de hardware, será **considerado** máquinas multiprocessadas conectadas por uma rede, sendo tanto **paralelo** quanto **distribuído**

- **Por que esses sistemas não são completamente paralelos?**

Introdução – Sistemas Distribuídos x Paralelos



Introdução – Sistemas Distribuídos x Paralelos

- **Escalabilidade** – sistemas distribuídos são inerentemente mais escaláveis que sistemas paralelos (mais processadores, mesma quantidade de memória)
- **Modularidade e Integridade** – um sistema distribuído é mais flexível por um simples processador poder ser adicionado ou retirado facilmente, de modelo diferente ou não
- **Compartilhamento de Dados** – sistemas distribuídos provêem compartilhamento de dados como um banco de dados distribuídos

Introdução – Sistemas Distribuídos x Paralelos

- **Compartilhamento de Recursos** – sistemas distribuídos compartilham recursos
- **Estrutura Geográfica** – a estrutura geográfica de uma aplicação deve ser inerentemente distribuída. A baixa taxa de comunicação deve forçar um processamento local
- **Confiança** – sistemas distribuídos são mais confiáveis que sistemas paralelos porque a falha de um processador não afeta a disponibilidade de outros
- **Baixo custo** – disponibilidade de largura de rede e baixo custo de estações de trabalho são fatores favoráveis

Desvantagens

Software	Pequenos softwares, complexidade alta
Rede	Rede pode saturar e causar problemas
Segurança	Acesso fácil pode comprometer sigilo e integridade dos dados

Por que um sistema não é puramente distribuído?

- As razões para manter um sistema paralelo em cada nó de uma rede são **tecnológicos**
  - Com a atual tecnologia é mais **rápido** atualizar a memória que enviar uma mensagem para outro processador (por exemplo, o novo valor de uma variável deve se comunicar com múltiplos processadores)
  - Conseqüentemente é mais eficiente obter **paralelismo fino** em um sistema paralelo que em um sistema distribuído

Introdução – Sistemas Distribuídos x Paralelos

- Atualmente, as interfaces providas aos programadores podem abstrair o hardware (por exemplo, SO's)
- No nível de programação, o curso será baseado em objetos distribuídos ***multithreads***
  - múltiplos processos que se comunicam via troca de mensagens
  - cada processo é composto de várias *threads*
  - orientado à objeto (reusabilidade, simplicidade, eficiente, compartilhamento)

- Uma breve classificação

Item	de rede	distribuído	multiprocessado
Parece um único processador virtual	Não	Sim	Sim
Todos tem que executar o mesmo S.O.	Não	Sim	Sim
Quantas cópias do S.O. existem	N	N	1
Elemento de comunicação	Arquivos compartilhados	Mensagens	Memória compartilhada
Precisa de protocolo de rede	Sim	Sim	Não
Tem uma simples fila de execução	Não	Não	Sim

Introdução – Características Sistemas Distribuídos e Paralelos

- Um sistema distribuído é caracterizado pela **ausência** de memória compartilhada
- Logo, em um sistema distribuído é **impossível** para qualquer processador conhecer o estado global do sistema
- Como resultado, é difícil observar qualquer propriedade global do sistema (por exemplo, hora)
- Existem algoritmos que propõem maneiras de conseguir tais propriedades

Introdução – Características Sistemas Distribuídos e Paralelos

- Sistema paralelo ou distribuído pode ser **fracamente** ou **fortemente** acoplado
  - ausência de *clock* compartilhado resulta em um sistema fracamente acoplado. Em um sistema distribuído geograficamente é impossível sincronizar os *clocks* de diferentes processadores
  - sistema paralelo um *clock* compartilhado pode ser simulado, projetando um sistema fortemente acoplado
- No corrente curso assumimos que os sistemas são **fracamente acoplados**

- Sistemas distribuídos podem ser classificados em **síncronos** ou **assíncronos**
  - SD é assíncrono se não há espera no tempo de comunicação das mensagens (*email*)
  - SD é síncrono quando as mensagens necessitam de sincronismo entre elas (*gaim*)

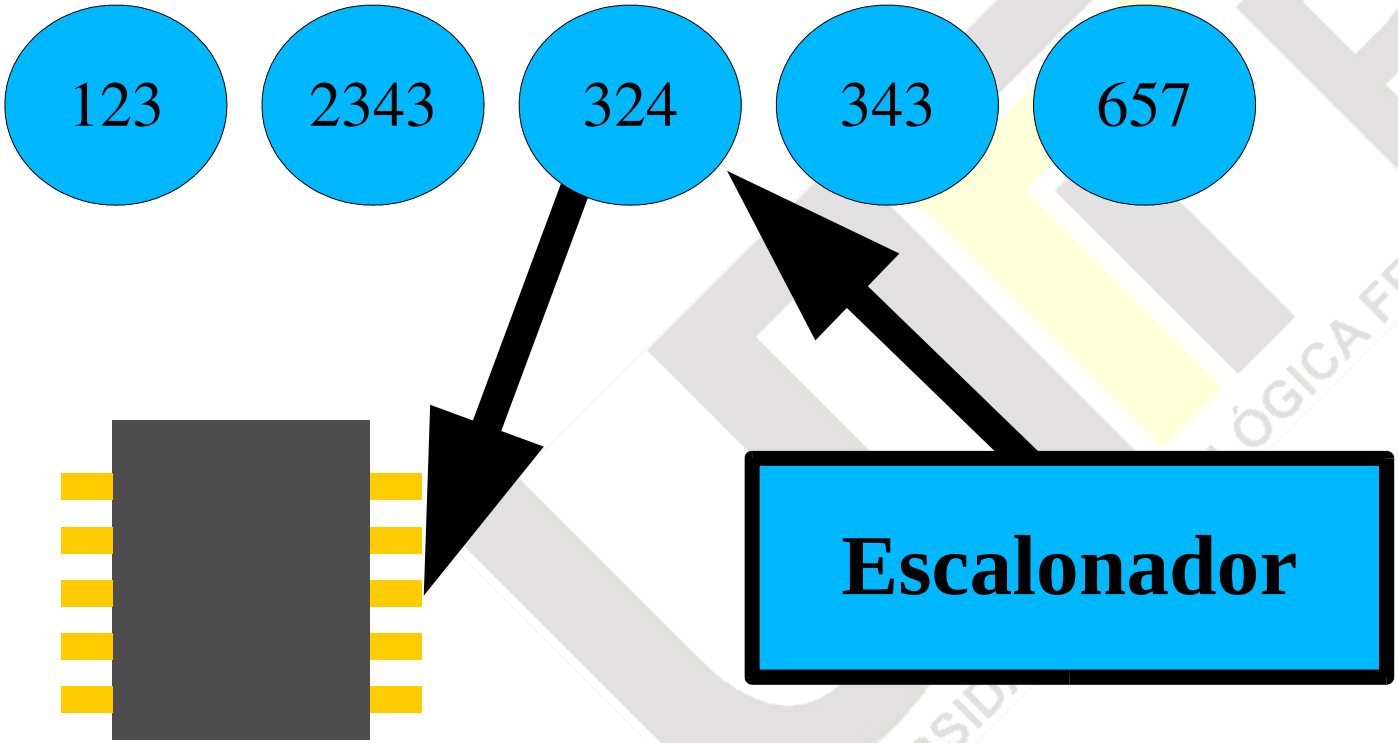
- A pesquisa em sistemas distribuídos mostra que alguns conceitos devem ser levados em consideração nos seus projetos:
  - **Tolerância à falhas** – o sistema deve ser capaz de esconder ou "mascarar" falhas de um ou mais componentes do sistema, incluindo processadores, memória e comunicação da rede. Geralmente requer redundância o que pode ser tornar muito caro dependendo do grau de tolerância à falhas

- A pesquisa em sistemas distribuídos mostra que alguns conceitos devem ser levados em consideração nos seus projetos:
  - **Transparência** – Um sistema deve *user-friendly* sempre que possível. Isso requer que o usuário não tenha preocupação com detalhes desnecessários. Por exemplo, em um sistema heterogêneo a diferença entre a representação de datas (indiano, inglês, etc.) deve ser escondido do usuário. Do mesmo modo, o uso de recursos por usuários não requerem o conhecimento de onde tais recursos se encontram.

- A pesquisa em sistemas distribuídos mostra que alguns conceitos devem ser levados em consideração nos seus projetos:
  - **Flexibilidade** – Um sistema deve ser capaz de interagir com largo número de outros sistemas e serviços. Isso requer que o sistema "conheça" um conjunto de regras de sintáxe e semântica para interação. Geralmente tais facilidades são obtidas por linguagens de definição de interfaces. Outra forma de flexibilidade é dar ao usuário a separação ente política e mecanismo.

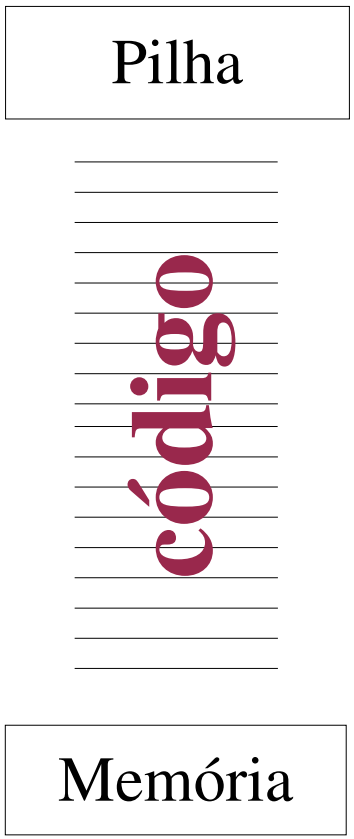
- A pesquisa em sistemas distribuídos mostra que alguns conceitos devem ser levados em consideração nos seus projetos:
  - **Escalabilidade** – Se um sistema for projetado para não ser escalável, provavelmente ele terá resultados insatisfatórios quando o número de usuários ou recursos aumentar. Por exemplo, um servidor pode saturar quando o número de requisições de clientes para determinado serviço aumentar.

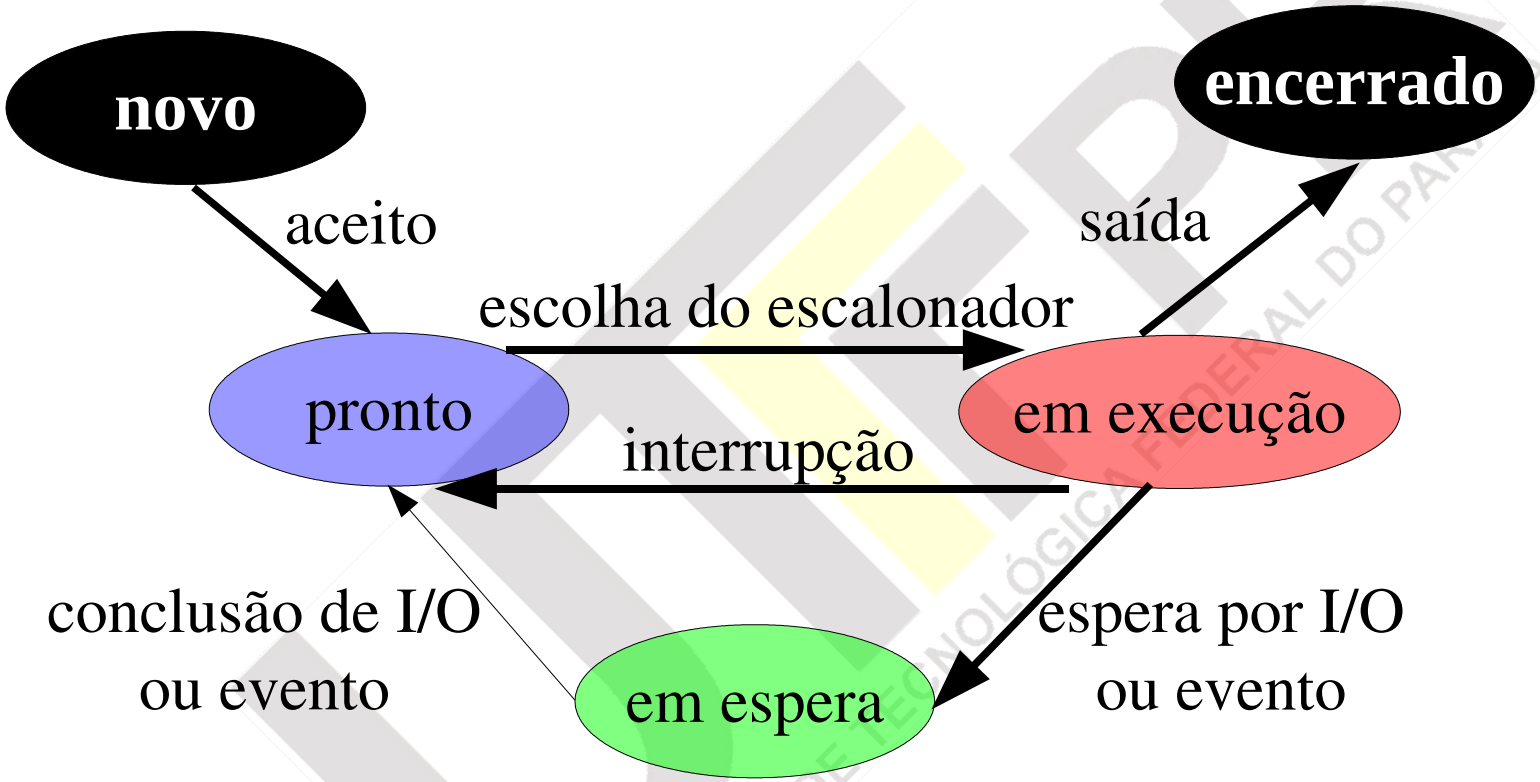
- Aspectos de concorrência surgem quando um simples **processador** é organizado como um conjunto de **processos cooperando**



- Antes da construção de sistemas concorrentes, é necessário entender a diferença entre programa e processo
  - **Programa -** um conjunto de instruções em uma linguagem de alto ou baixo nível
  - **Processo -** pode ser visto como consistindo de três segmentos na memória: dados, código e pilha de execução

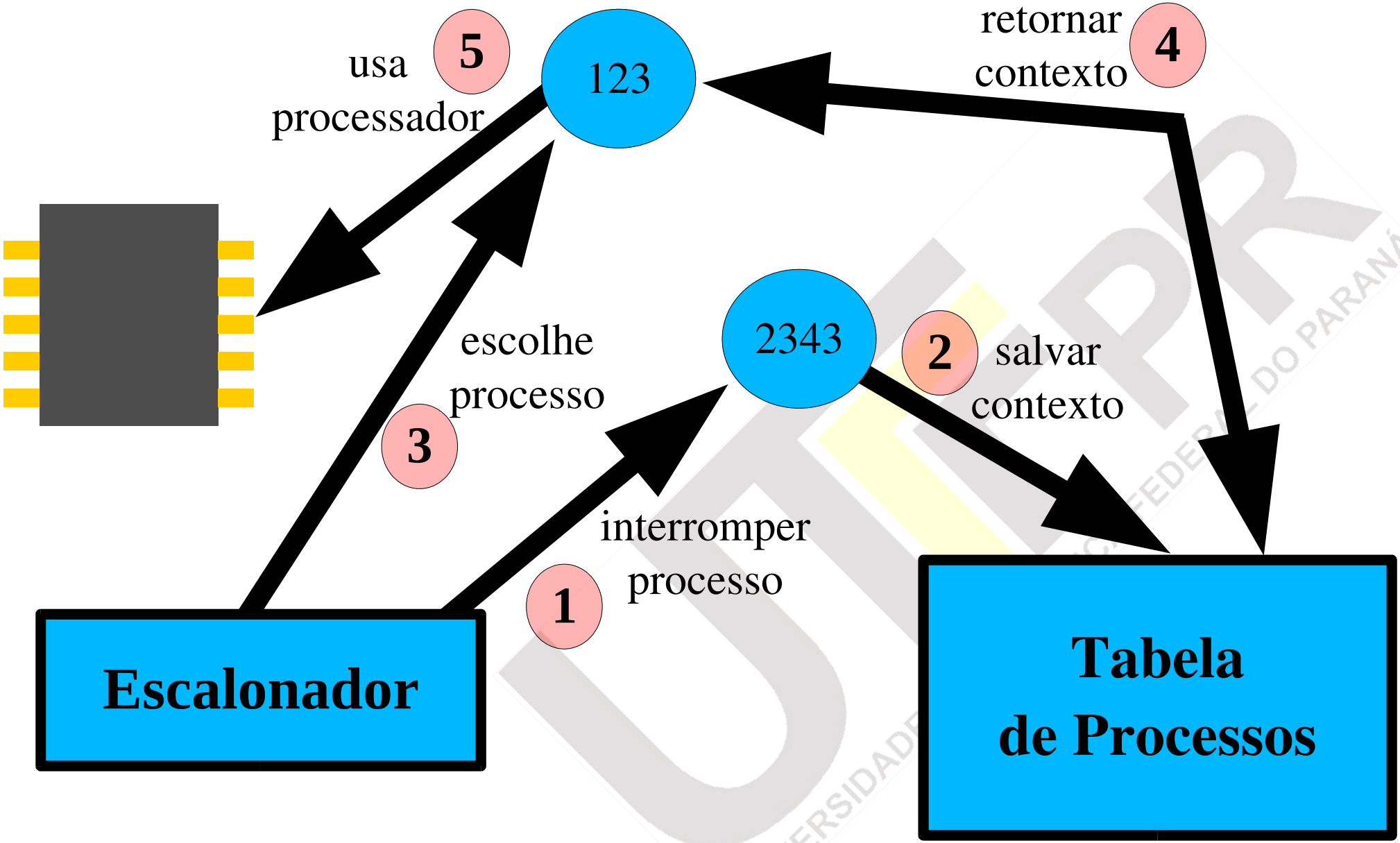
Processo

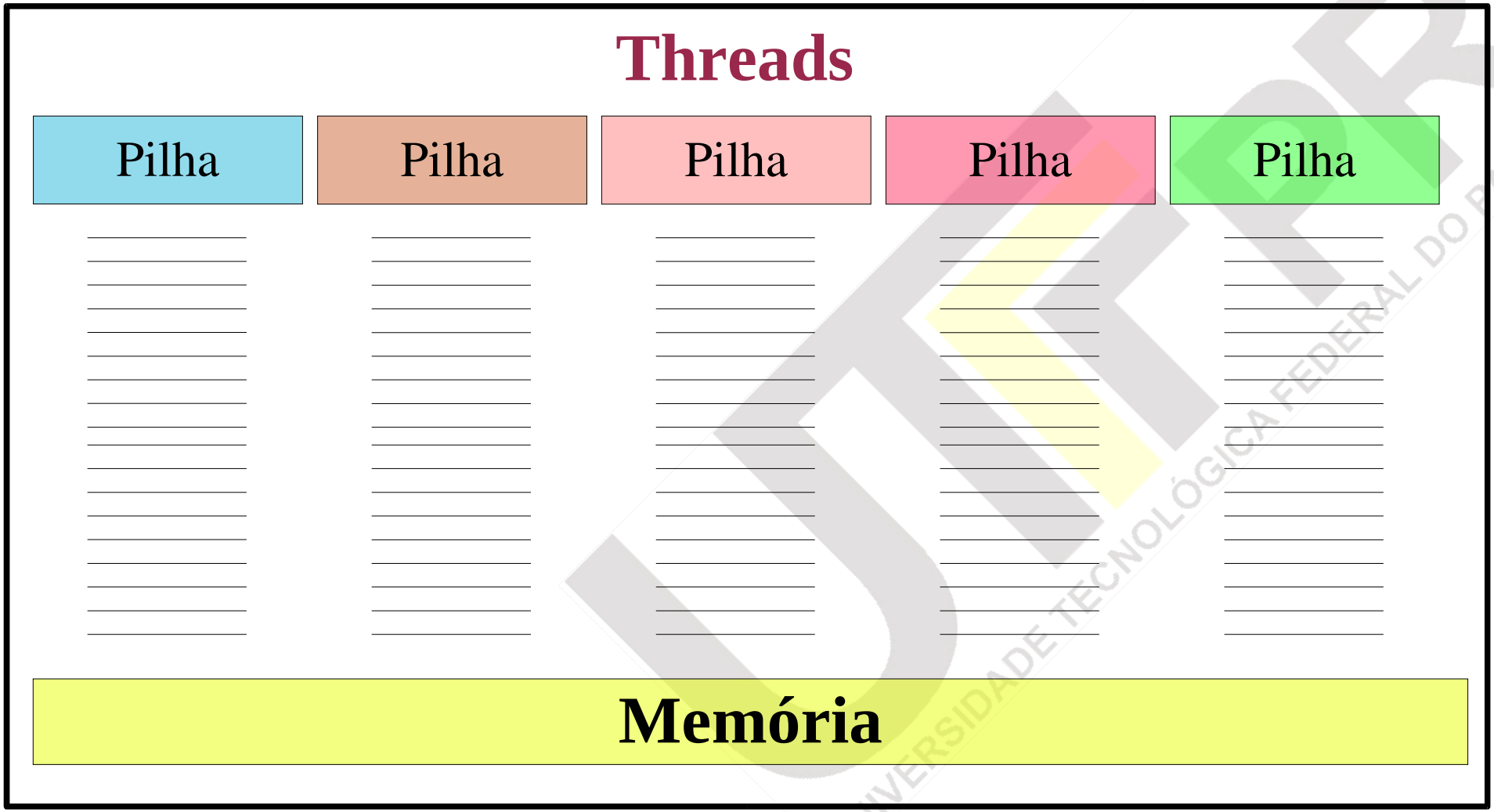

Pilha
código
Memória


novo
encerrado
aceito
saída
escolha do escalonador
pronto
em execução
interrupção
conclusão de I/O
ou evento
em espera
espera por I/O
ou evento

- **Código** é o conjunto de instruções na memória que o processo executa
- **Dados** consiste na memória necessária para alocação de variáveis estáticas e em tempo de execução
- **Pilha** ativa chamadas de funções por meio de variáveis locais
- **Quando processos compartilham o espaço de endereçamento, geralmente, dados e código, são chamados de *threads* ou processos leves**

Introdução – Especificação de Processos e Tarefas




PROCESSO
Threads
Pilha
Pilha
Pilha
Pilha
Pilha
Memória

- Qualquer linguagem de programação que **suporte concorrência** deve ter um meio de especificar a estrutura de processos e como vários processos comunicam-se e sincronizam-se
  - No **UNIX** processos são organizados como uma árvores de processos, sendo identificado por um número (PID – *process identifier*)
  - **UNIX** provê chamadas de sistema ***fork*** e ***wait*** para criar e sincronizar processos

- Quando um processo executa uma chamada ***fork***, um processo filho é criado com uma cópia do **espaço de endereçamento** do processo pai
- A única diferença entre eles é o valor do **código de retorno** do ***fork***

```
pid=fork() //cria um novo processo
if (pid==0){ //se pid for igual a um, é o processo filho
    //processo filho
    cout << "Processo Filho"
}
else{
    //processo pai
    cout << "Processo Pai"
}
```

PID - 3432

```
pid=fork()
if (pid==0){
        //processo filho
     cout << “Processo Filho”
}
else{
     //processo pai
     cout << “Processo Pai”
}
```

PID - 3434

```
pid=fork()
if (pid==0){
     //processo filho
     cout << “Processo Filho”
}
else{
     //processo pai
     cout << “Processo Pai”
}
```

Mesmo código

PID - 3432

```
pid=fork()
if (pid==0){
        //processo filho
     cout << “Processo Filho”
}
else{
     //processo pai
     cout << “Processo Pai”
}
```

Dados, Pilha e Memória diferentes e independentes

PID - 3434

```
pid=fork()
while(n>m){
     //instruções
     ....
}
```

Códigos diferentes

- A chamada ***wait*** é usada pelo processo pai para esperar o término do processo filho
- Um processo **termina** quando é executada a última instrução no código ou faz-se uma **chamada explícita** para o sistema acabar

- Outra construção de programação paralela é *cobegin – coend : cobegin **S1** || **S2** coend*
  - A instrução salienta que **S1** e **S2** devem ser executados em paralelo
  - Caso um termine antes, deverá aguardar o encerramento do outro processo

- Outra método de especificação de concorrência é explicitamente criar objetos ***threads***
  - Por exemplo, em Java há pré-definida uma classe chamada ***Thread***
  - É possível extender a classe ***Thread***, usando o método ***run()*** e chamando ***start()*** para lançar uma ***thread***

```
public class HelloWorldThread extends Thread{

    public void run(){
        System.out.println("Olá mundo!!!!!");
    }

    public static void main(String args[]){
        HelloWorldThread t=new HelloWorldThread();
        t.start();
    }
}
```

- *HelloWorldThread* precisa herdar métodos da classe ***Thread***
  - **E se for necessário extender uma classe, por exemplo *Foo*, mas também tornar os objetos dessa nova classe como *threads* separadas?**
- Java não tem suporte para herança múltipla, porém deve-se extender a classe ***Foo*** e a classe ***Thread***
- Para resolver esse problema, Java provê uma interface chamada ***Runnable*** com o método ***public void run()***

```
class Foo{
    String name;
    public Foo(String s){
        name=s;
    }
    public void setName(String s){
        name=s;
    }
    public String getName(){
        return name;
    }
}
```

```
class FooBar extends Foo implements Runnable{
    public FooBar(String s){
        super(s);
    }
    public void run(){
        for(int i=0;i<10;i++)
            System.out.println(getName()+" : Olá mundo!!");
    }
    public static void main(String args[]){
        FooBar f1=new FooBar("Romeu");
        Thread t1=new Thread(f1);
        t1.start();
        FooBar f2=new FooBar("Julieta");
        Thread t2=new Thread(f2);
        t2.start();
    }
}
```

- *Threads* podem **esperar** que outras *threads* terminem sua execução via mecanismo ***join()***

- Como exemplo tem-se o seguinte programa:

Introdução – Especificação de Processos e Tarefas

```
class Produto implements Runnable{
	private int estoque = 1000;
	public void run(){
		for(int i=0;i<4;i++)
			efetuarPedido();
	}
	public void efetuarPedido(){
		try{
			if(this.estoque>0){
				System.out.println("Pedido faturado para o cliente "+Thread.currentThread().
getName());
				Thread.sleep(250);
				this.estoque--;
			} else{
				System.out.println("Nao tem estoque para o cliente "+Thread.currentThread().
getName());
			}
		}catch(Exception e){
			System.out.println(e);}}
	public Produto(int valor){this.estoque=valor;}}
```

```
public class pedidoCompra{
    public static void main(String args[]){
        Produto p=new Produto(15);
        Thread[] t=new Thread[15];
        for (int i=0;i<t.length;i++){
            try{
                t[i]=new Thread(p);
                t[i].setName("Cliente"+i);
                t[i].start();
            }catch(Exception e){}
        }
    }
}
```

- O resultado da execução desse programa não é **consistente**
- Para resolver esse efeito podemos usar o método ***join*** no seguinte trecho:

```
...
                try{
                    t[i]=new Thread(p);
                    t[i].setName("Cliente"+i);
                    t[i].start();
                    t[i].join();
                }catch(Exception e){}
            }
        }
}
```

```
public class Fibonacci extends Thread {
   ...
    public void run() {
        if ((n == 0)||(n == 1 )) result = 1;
        else {
            Fibonacci f1 = new Fibonacci(n-1);
            Fibonacci f2 = new Fibonacci(n-2);
            f1.start();
            f2.start();
            try {
                f1.join();
                f2.join();
            } catch (InterruptedException e){};
            result = f1.getResult() + f2.getResult();}
    }
    ...
}
```

Introdução – Escalonando Threads

- No exemplo da classe *FooBar* tem-se duas *threads*
- Se o programa for executado tanto numa máquina monoprocessada quanto em uma multiprocessada a ordem de execução das ***threads*** dependerá de sua **prioridade**
  - O programador pode verificar a prioridade de uma *thread* com *setPriority* e determinar a mesma com *getPriority*
  - **MIN_PRIORITY**(1) e **MAX_PRIORITY**(10) são constantes inteiras que definem uma classe *Thread*
  - Por default, uma *thread* tem a prioridade **NORM_PRIORITY**(5)

Introdução – Escalonando Threads

```
class FooBar extends Foo implements Runnable{
    public FooBar(String s){  super(s);}
    public void run(){
        for(int i=0;i<10;i++)  System.out.println(getName()+" : Olá mundo!!");}
    public static void main(String args[]){
        FooBar f1=new FooBar2("Romeu - t1");
        Thread t1=new Thread(f1);
        FooBar f2=new FooBar2("Julieta - t2");
        Thread t2=new Thread(f2);
        t2.setPriority(3);
        System.out.println("Prioridade t1="+t1.getPriority()+" e
t2="+t2.getPriority());
        if (t1.getPriority()>t2.getPriority())
            t1.start();
        else t2.start();}}
```

Introdução – Exercícios

- Defina a classe Contador como uma subclasse de Thread, que imprima números de 0 a 10. Crie a classe TesteContador que deve definir o método main que cria e inicia a execução do thread Contador. Teste o resultado executando a classe TesteContador.
- Altere as classes Contador e TesteContador de modo que a classe Contador seja definida como uma implementação da interface Runnable. Teste o resultado.
- Altere o método main da classe TesteContador para criar dois ou mais threads Contador e inicialize a execução dos mesmos.

Introdução – Exercícios

- Dê vantagens e desvantagens do modelo de programação paralela sobre o modelo de sistema distribuído
- Escreva uma classe em Java que permita paralelizar uma pesquisa em um *array* de inteiros. Isso deve ser feito com o seguinte método: *public static int parallelSearch(int x, int[] A, int numThreads)*. Este método cria tantas *threads* quanto especificadas em *numThreads*, divide o *array* A em muitas partes e da a cada *thread* parte do array para procurar sequencialmente pelo valor x. Se uma *thread* encontrar o valor *x*, então é retornado o índice *i* (*A[i]=x*), ao contrário *-1*.

- Considere o seguinte código abaixo:

```
Class Schedule{
   static int x=0;
   static int y=0;
   public static int op1(){x=1; return y;}
   public static int op2(){y=2; return 3*x;}
}
```

Se uma *thread* chamar *op1* a outra *op2*, qual valor deverá ser retornado para *op1* e *op2*?

- Escreva um programa *multithread* em java que ordene um *array* usando *merge sort* recursivo. O programa principal cria duas *threads* para ordenar as duas partes do *array* quer serão ordenados.