

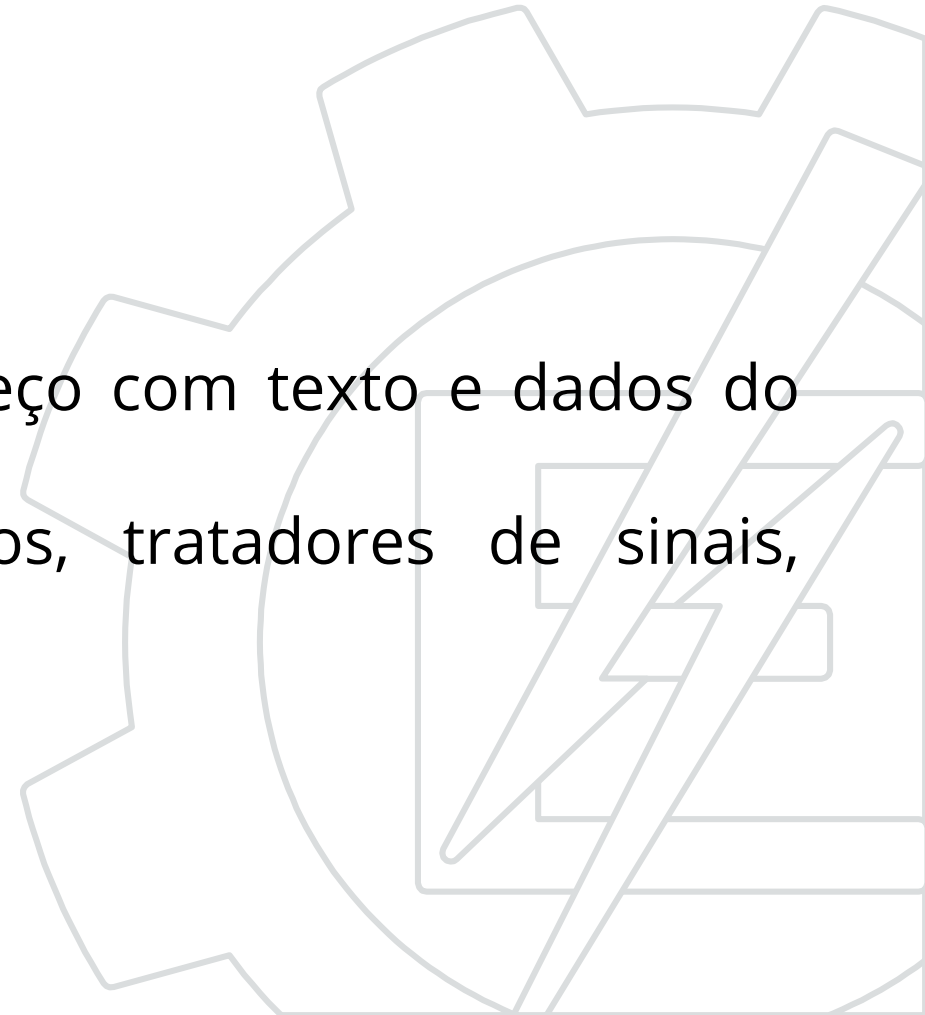
Sistemas Operacionais

Threads



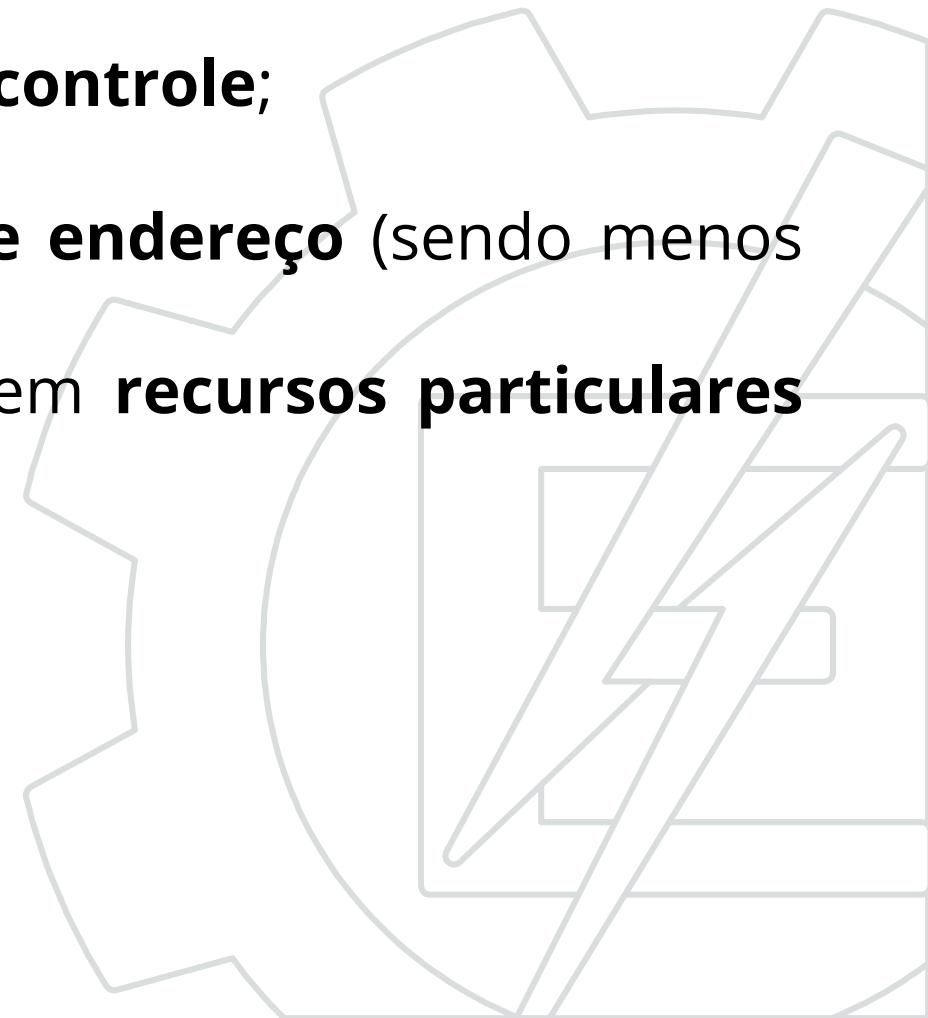
O modelo de Processo

- Possui um **espaço de endereço exclusivo** (0 até algum endereço máximo);
- Possui uma **única linha de execução** (*thread*);
- **Agrupamento de recursos** (espaço de endereço com texto e dados do programa, arquivos abertos, processos filhos, tratadores de sinais, alarmes pendentes, etc.)



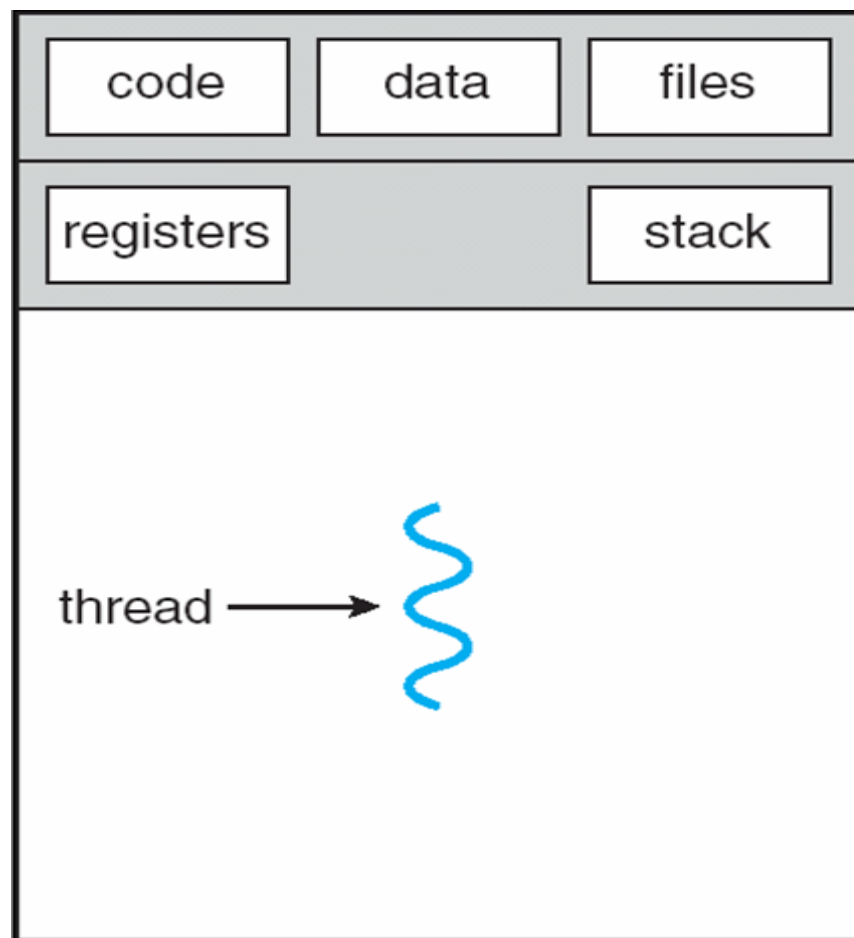
O modelo de *Thread*

- Conjunto de *threads* compõe as **linhas de execuções de um processo**;
- Um espaço de endereço e **múltiplas linhas de controle**;
- *Threads* **compartilham um mesmo espaço de endereço** (sendo menos independentes que os processos), mas possuem **recursos particulares** (PC, registradores, pilha).

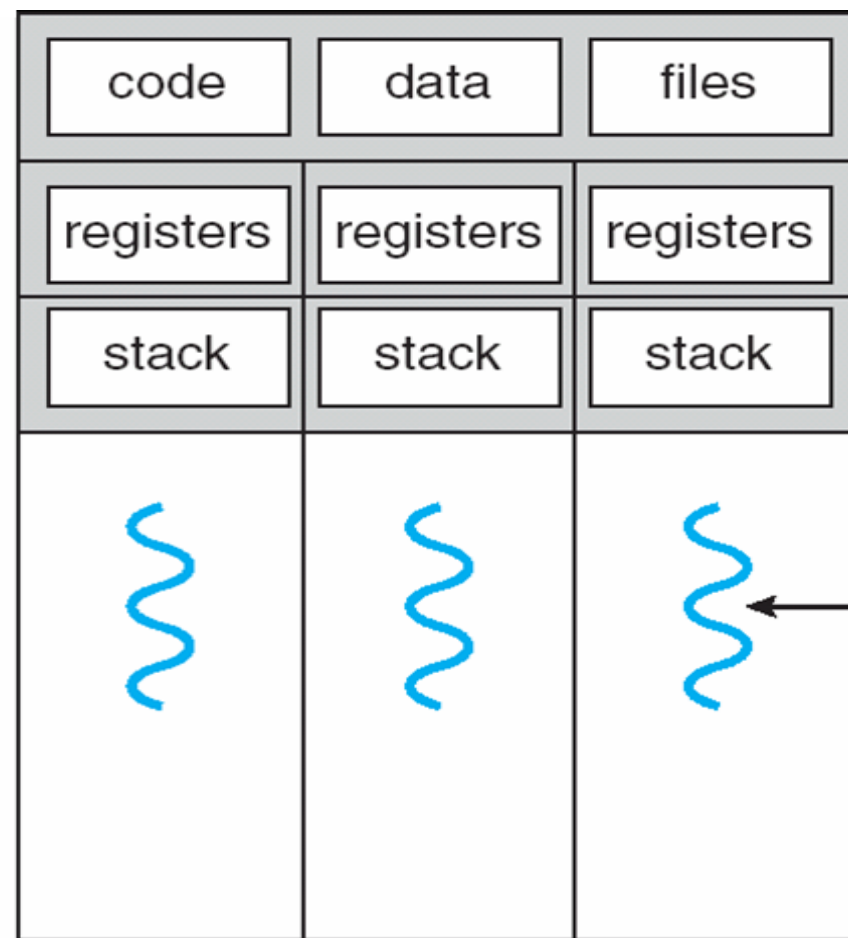


Threads

Single-thread e Multithread



single-threaded process

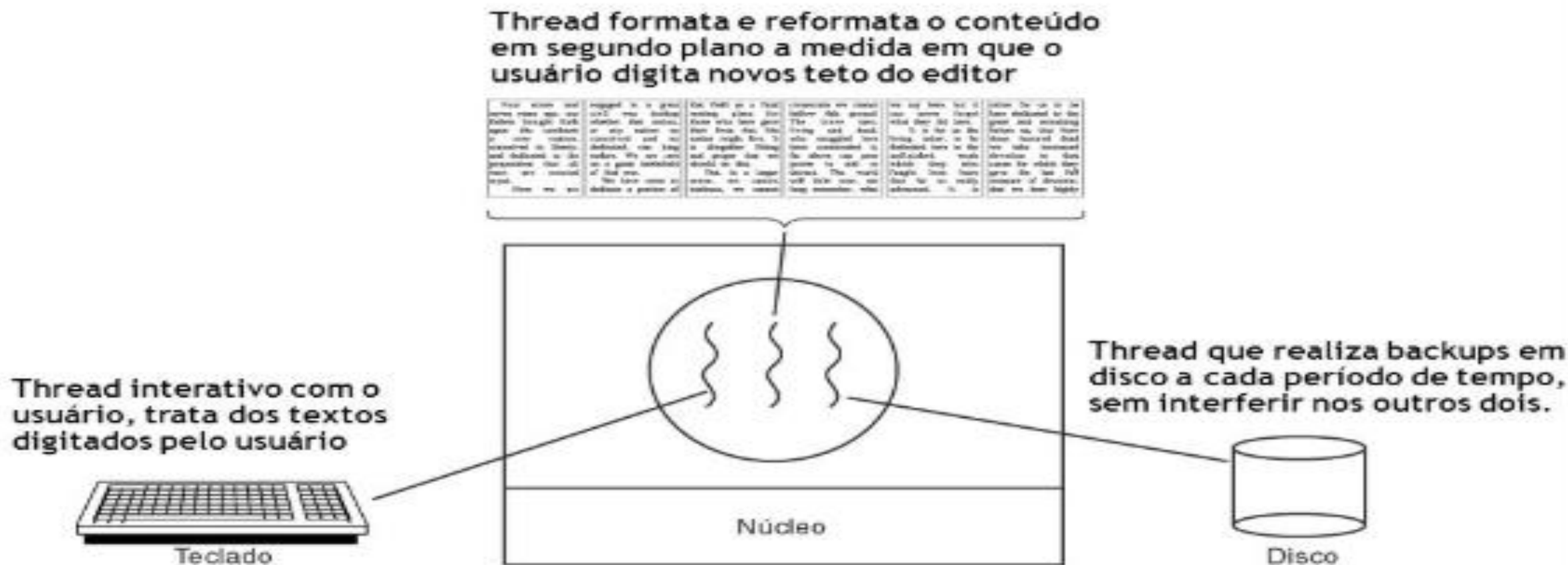


multithreaded process

Threads

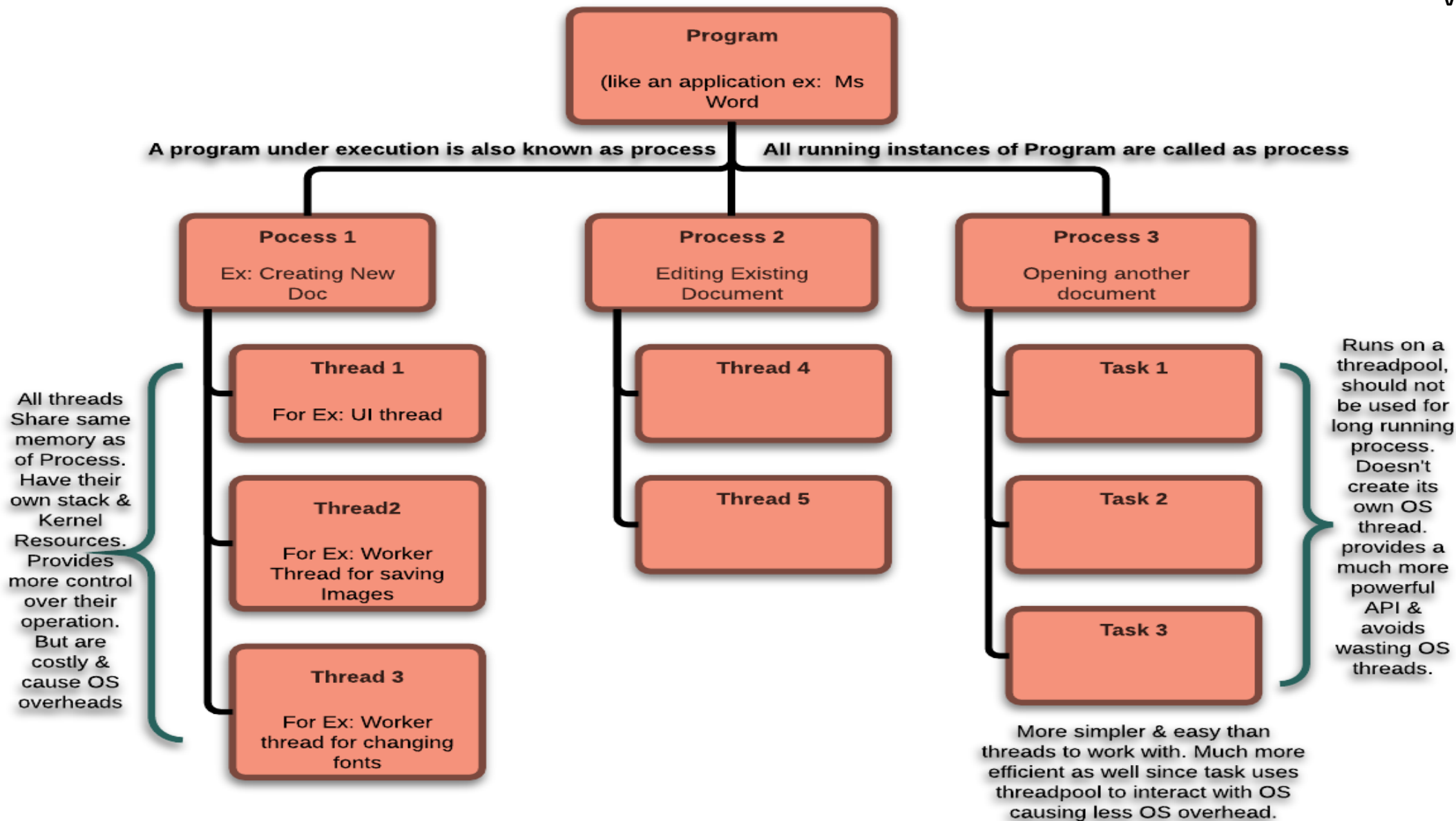
Vantagens

- Em muitas aplicações há **múltiplas atividades** ao mesmo tempo;
- *CPU-bound* e *I/O-bound* podem se sobrepor, acelerando a aplicação.

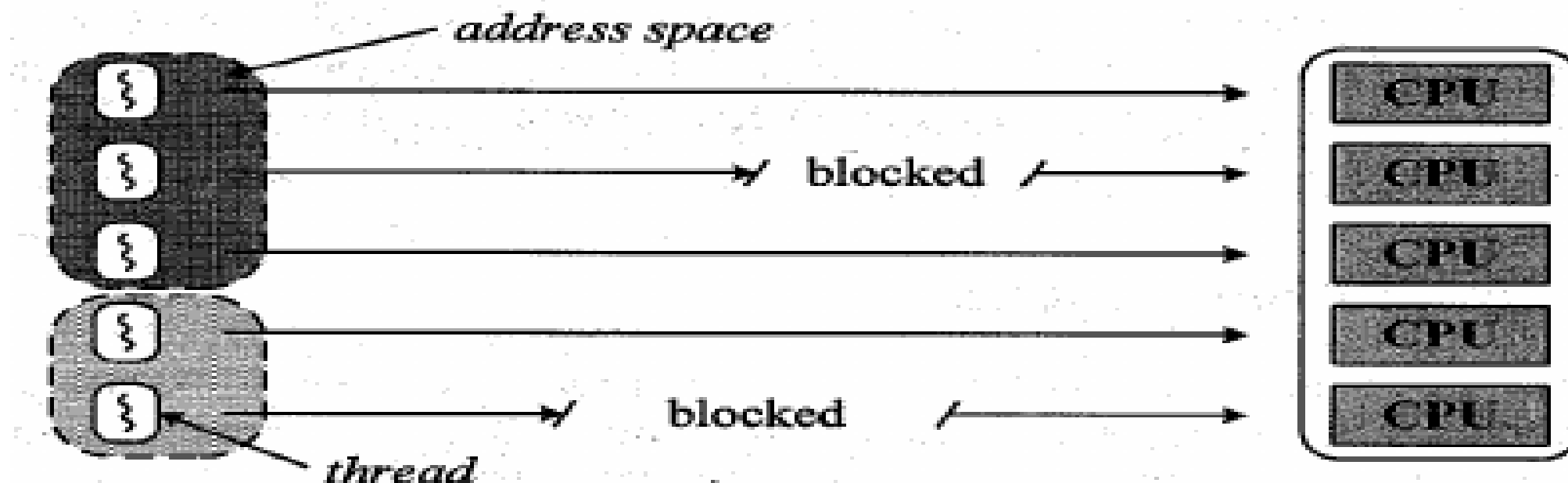


Threads

Vantagens



- Podemos decompô-las em **atividades paralelas**;
- Algumas tarefas precisam de **compartilhamento do espaço** de endereçamento.



- **Tempo de resposta**

programa dividido em várias linhas de execução

Ex. *Writer: user interface, keystrokes, spellchecker, file I/O*

- **Compartilhamento de recursos**

threads compartilham código e dados (ex.: var. globais) do processo pai

processos precisam de mecanismos de IPC (*shared* mem., msg.)

- **Economia**

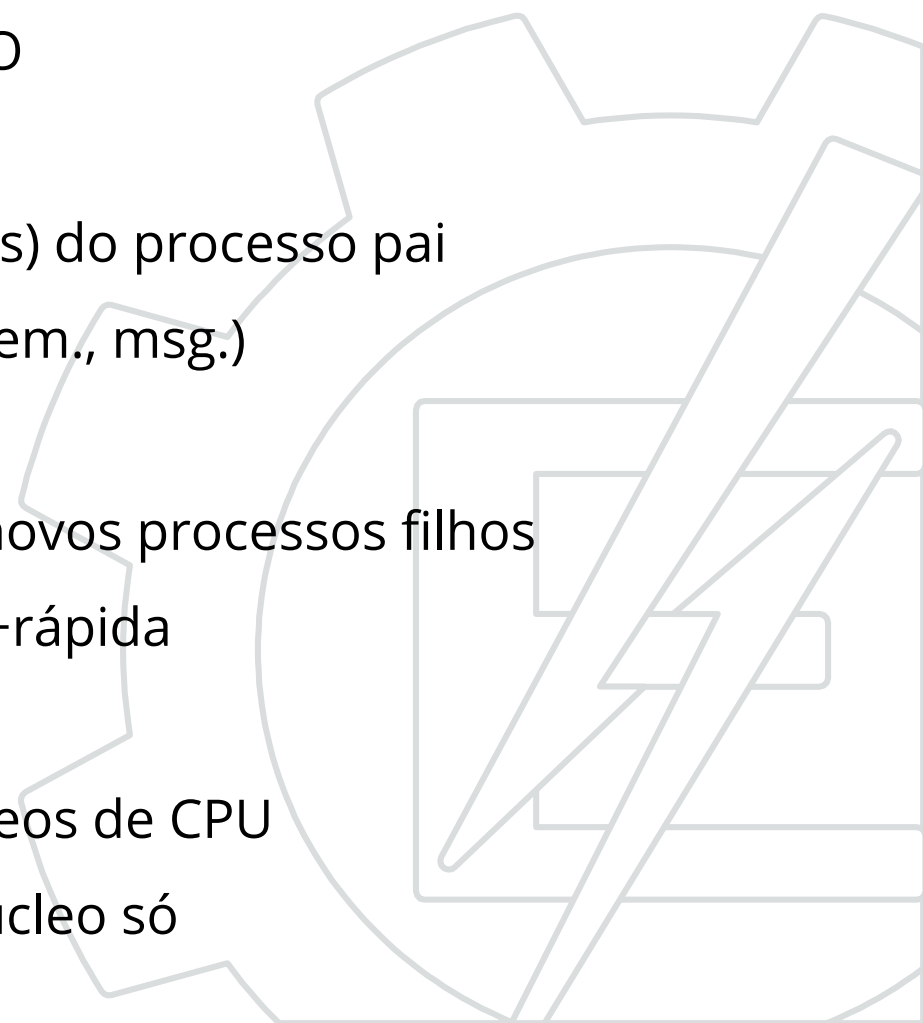
É mais barato criar *threads* no processo do que criar novos processos filhos

Ex. Solaris: criação 30x +rápida, troca de contexto 5x +rápida

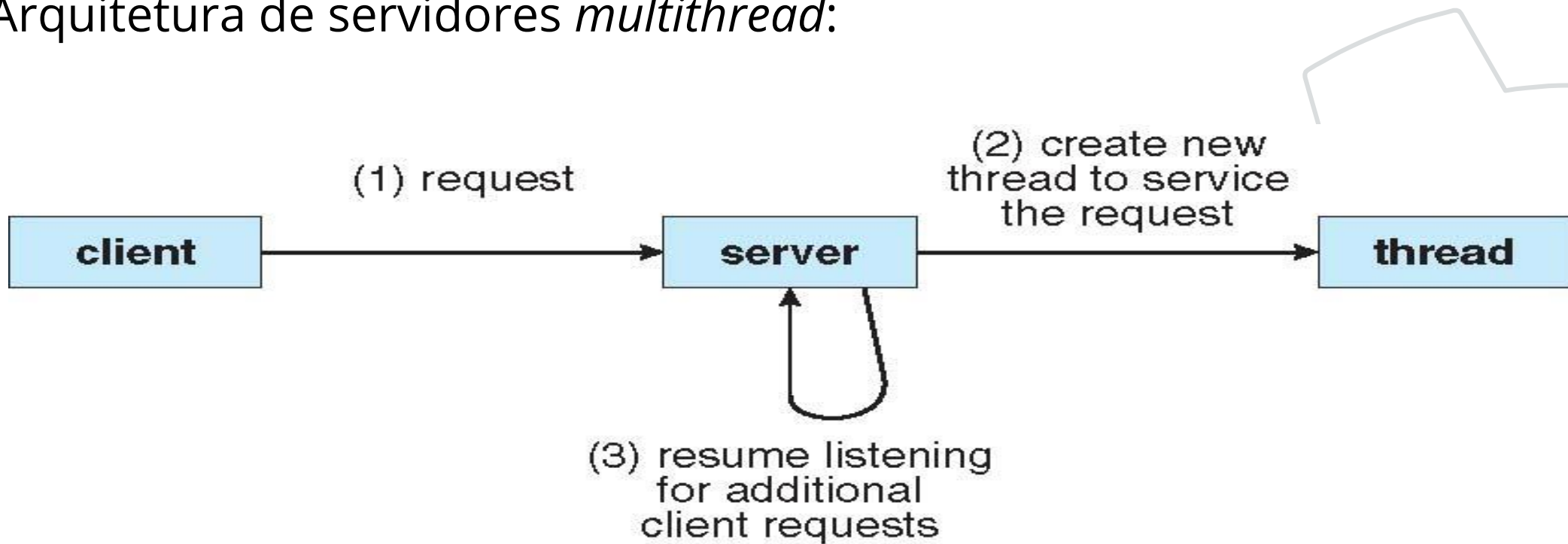
- **Escalabilidade**

threads podem rodar em paralelo em diferentes núcleos de CPU

um processo de uma thread só pode rodar em um núcleo só



- Úteis em sistemas com múltiplas CPUs >> **Paralelismo real.**
- Arquitetura de servidores *multithread*:



- **Paralelismo de dados** tem como foco a **distribuição de partes** (*subsets*) do mesmo dado em múltiplos núcleos de processamento (*cores*) e a realização da **mesma operação em cada um dos núcleos**. Por exemplo, a realização da soma dos elementos de um vetor de tamanho N.
- **Paralelismo de tarefa** envolve a **distribuição** não dos dados, mas **das tarefas em múltiplos núcleos** de processamento (*cores*). Cada *thread* executa uma operação específica. Diferentes *threads* podem operar sobre o mesmo dado, ou sobre dados diferentes. Em contraste com o paralelismo de dados, o paralelismo de tarefas deve utilizar pelo menos duas *threads* cada uma realizando uma operação diferente nos elementos do vetor.

- **Preemptiva:**

- Controle do sistema operacional;
- Compartilhamento da CPU é garantido.

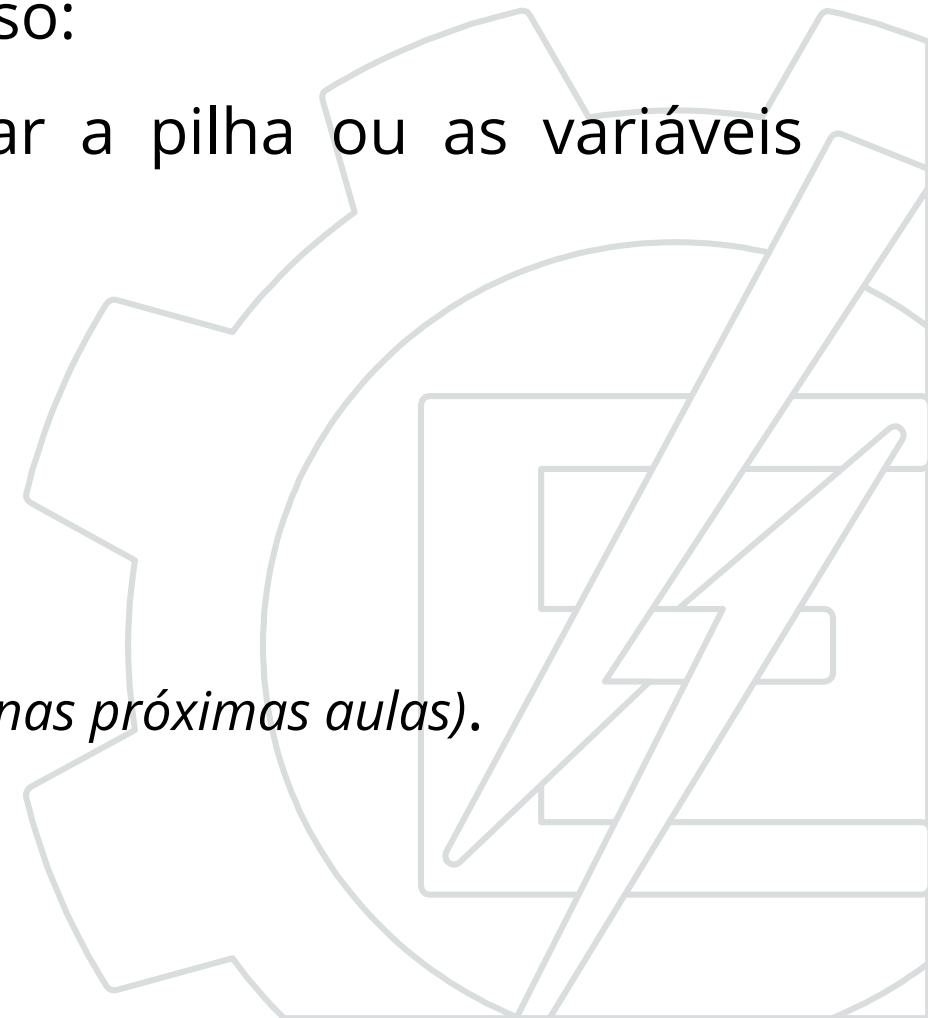
- **Cooperativa:**

- Controle da *thread*;
- Compartilhamento da CPU não é garantido.

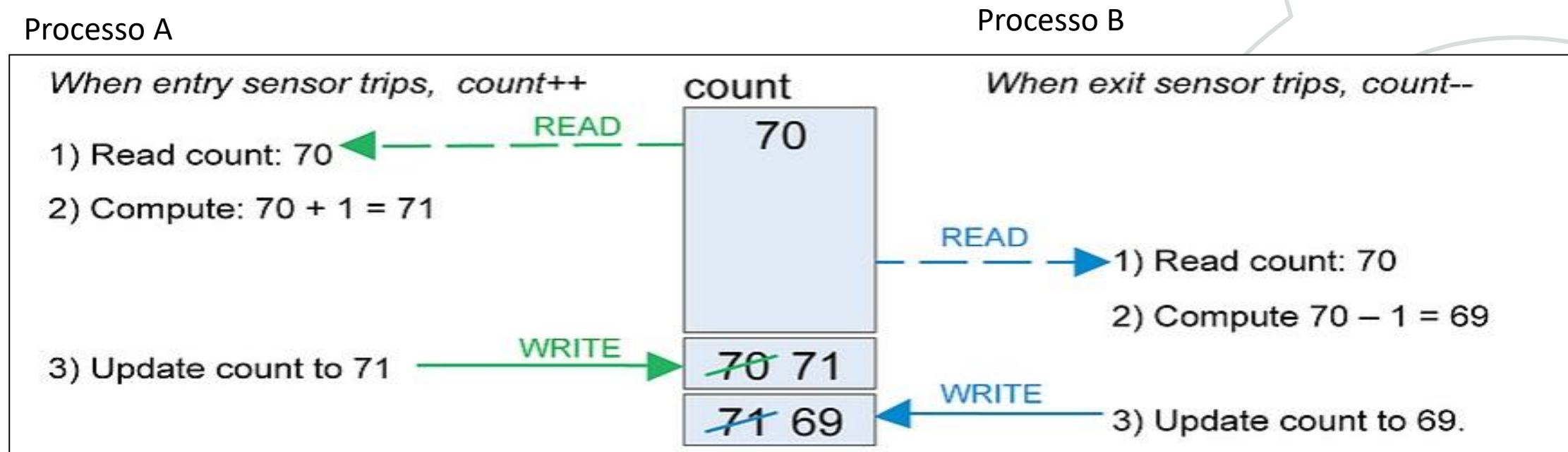


- Problemas (*issues*) incluem:
 - Cancelamento de *thread*;
 - Controle/manuseio de sinais (*signal handling - synchronous/asynchronous*);
 - Manuseio (*handling*) de dados em uma thread específica;
 - Escalonamento (*scheduler*) de ações.
- Cancelamento:
 - Cancelamento assíncrono finaliza a *thread-alvo* imediatamente;
 - Cancelamento aprovado (*deferred cancellation*) permite que a *thread-alvo* verifique periodicamente se deve ser cancelada.

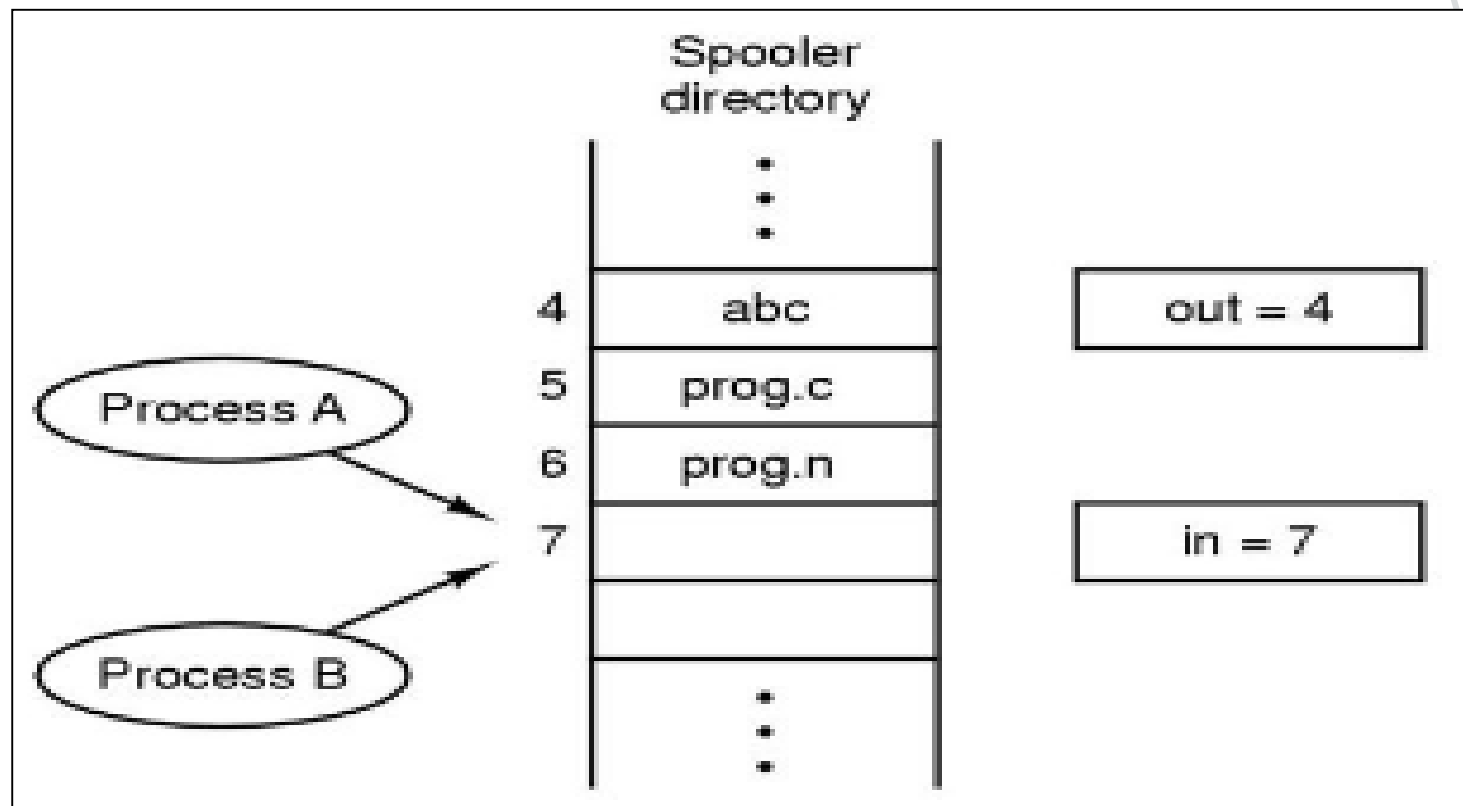
- Como cada *thread* pode ter acesso a qualquer endereço de memória dentro do espaço de endereçamento do processo:
 - Uma *thread* pode ler, escrever ou apagar a pilha ou as variáveis globais de outra *thread*. Exemplo:
$$a = b + c;$$
$$x = a + y;$$
- Necessidade de sincronizar a execução (*será visto nas próximas aulas*).



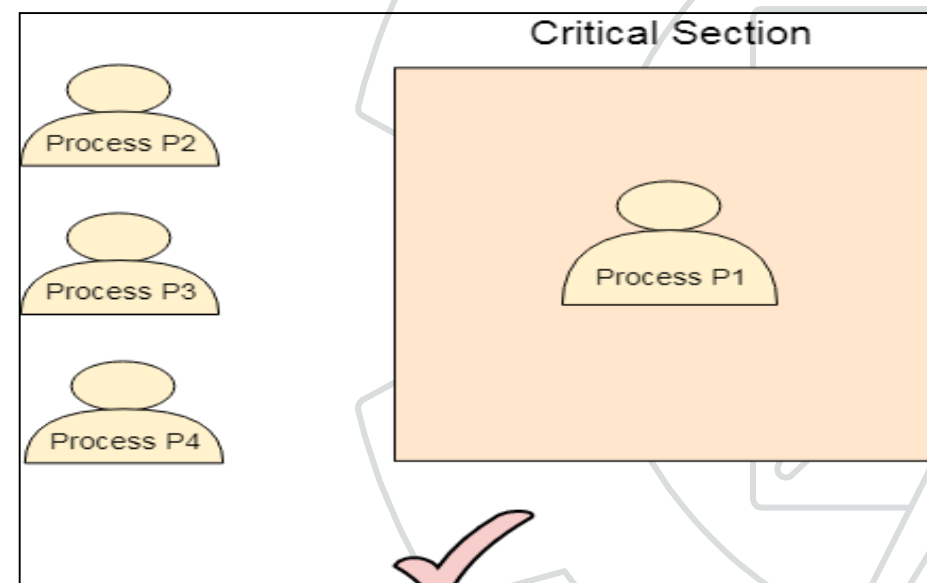
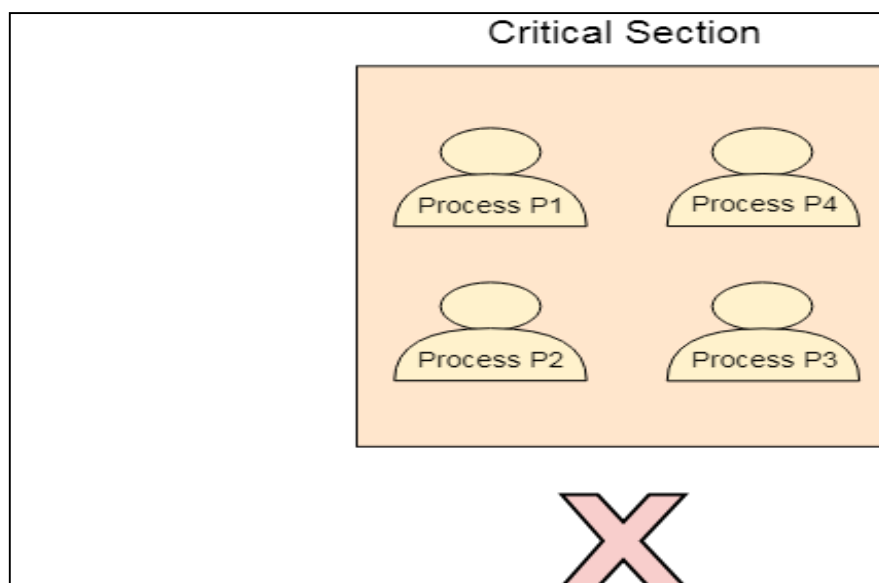
- Acesso concorrente a dados compartilhados pode resultar em **inconsistências**;
- **Condições de corrida** (*race conditions*):
 - Situação onde dois ou mais processos acessam e manipulam recursos compartilhados simultaneamente.



- Acesso concorrente a dados compartilhados pode resultar em **inconsistências**;
- **Condições de corrida** (*race conditions*):
 - Situação onde dois ou mais processos acessam e manipulam recursos compartilhados simultaneamente.

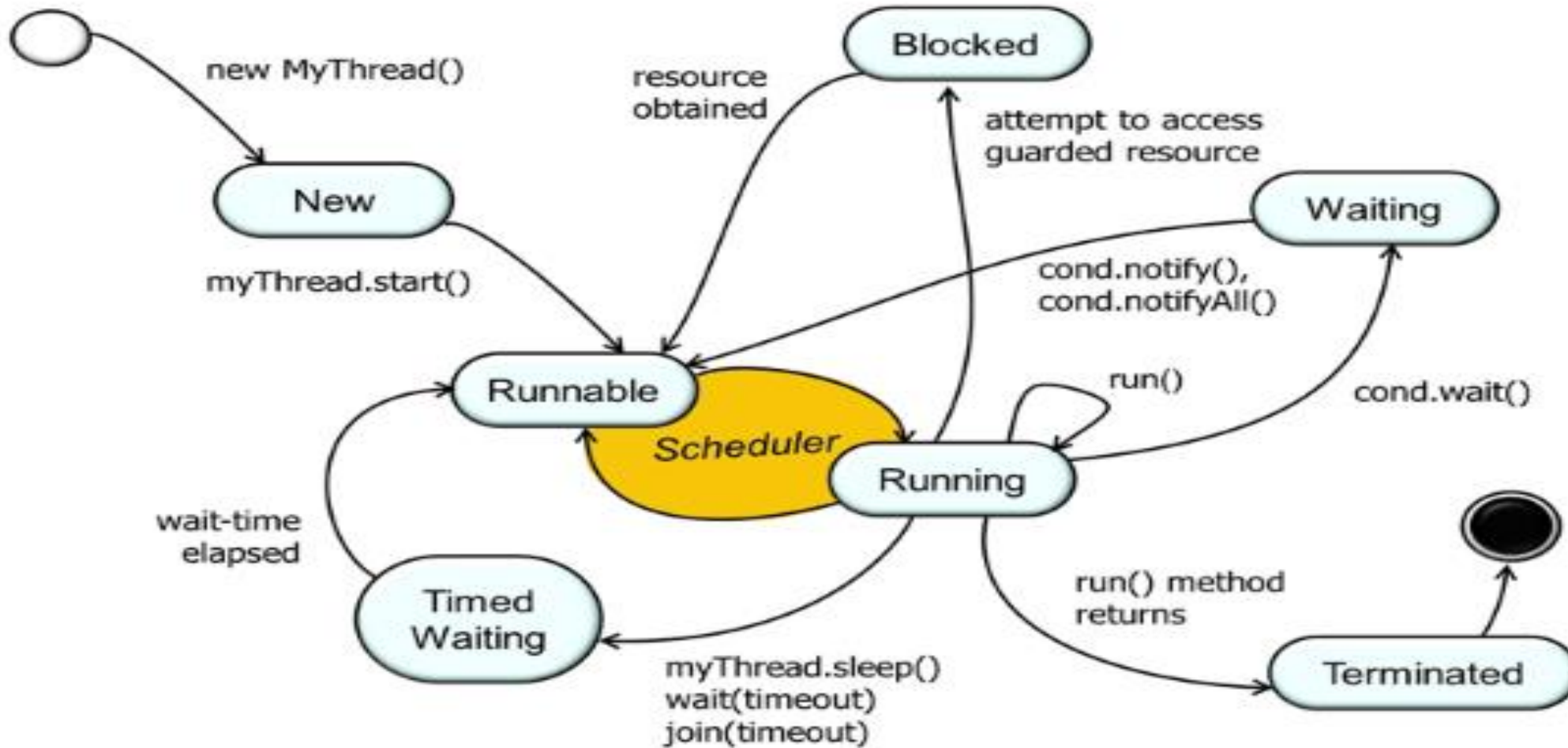


- Acesso concorrente a dados compartilhados pode resultar em **inconsistências**;
- **Condições de corrida** (*race conditions*):
 - Situação onde dois ou mais processos acessam e manipulam recursos compartilhados simultaneamente.
 - **Seção crítica:** N processos competem para usar alguma estrutura de dados compartilhada (*será visto nas próximas aulas*).



Threads

Java Threads State Diagram



Threads

Windows Threads State Diagram

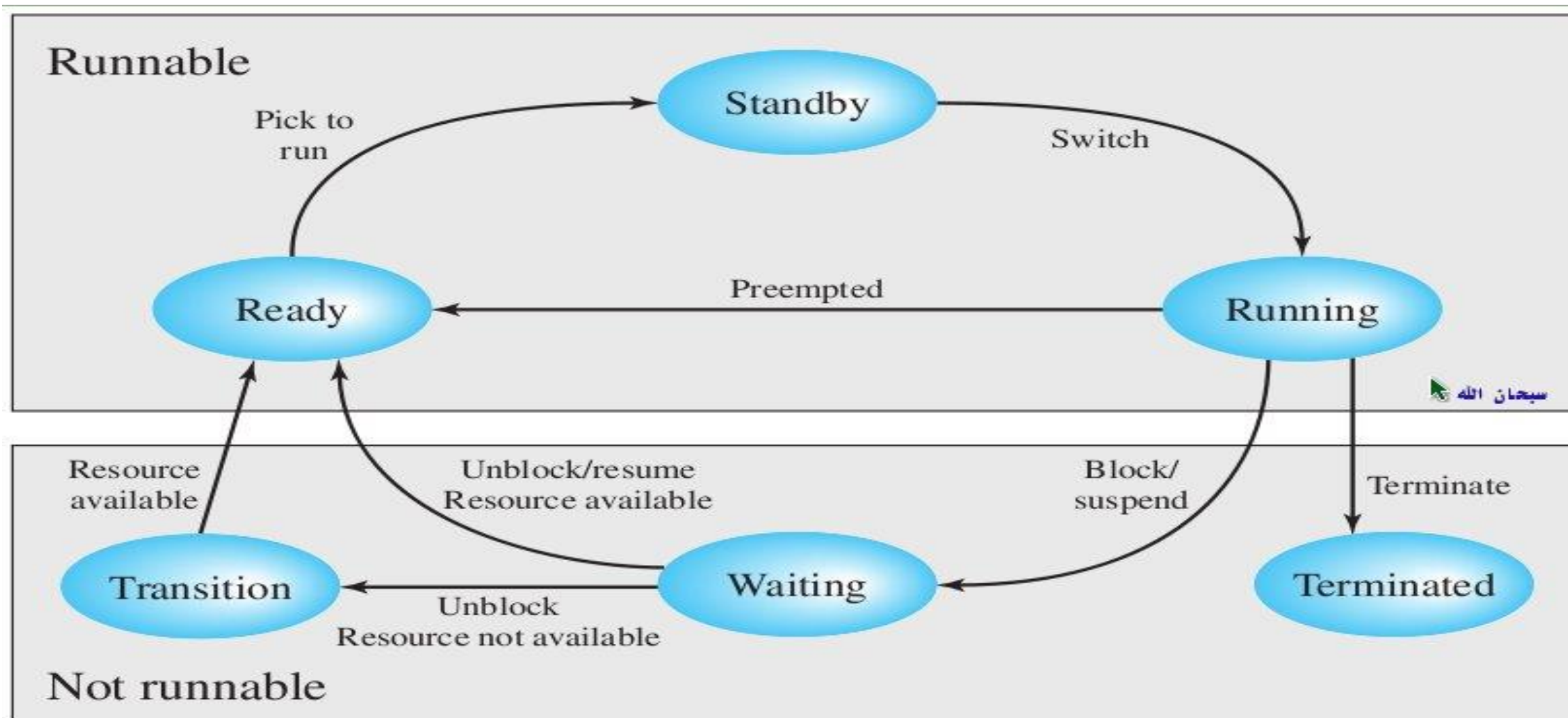
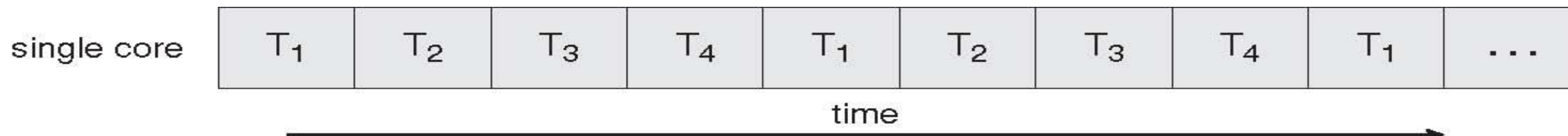


Figure 4.14 Windows Thread States

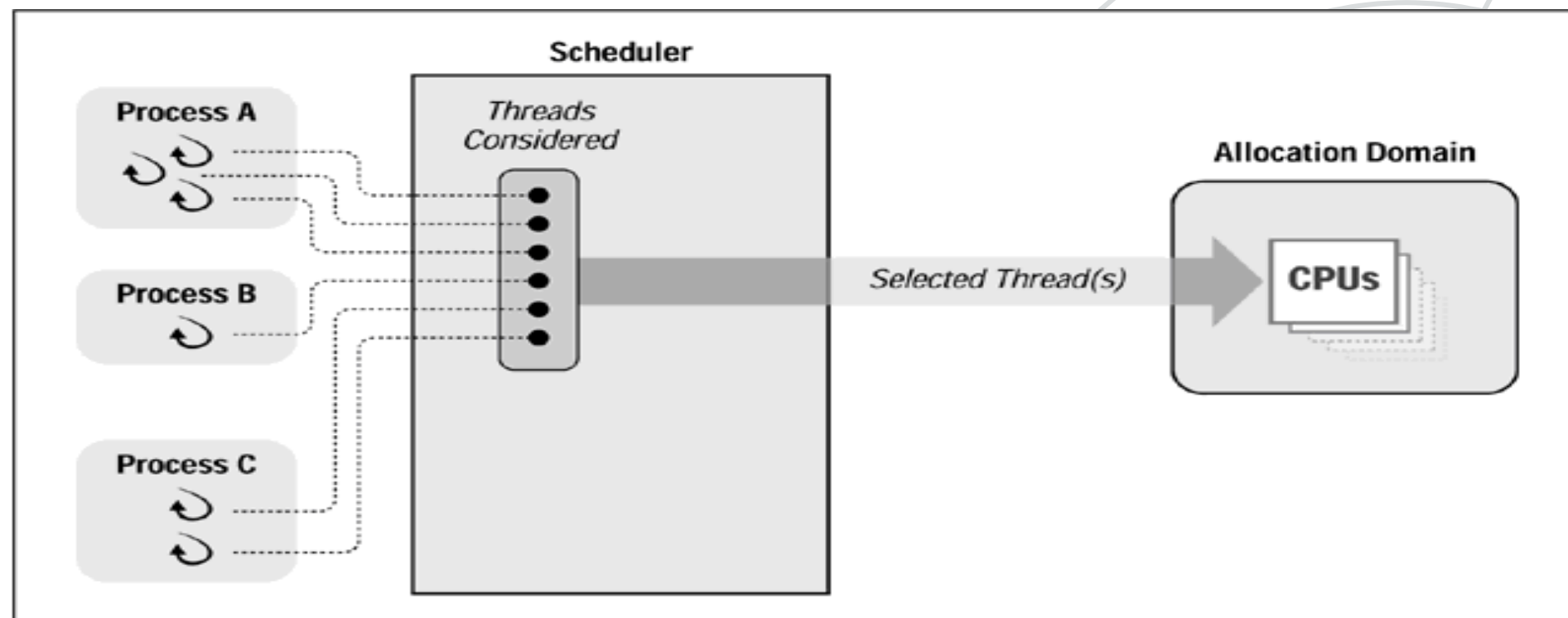
Threads

Execução concorrente - Mononúcleo



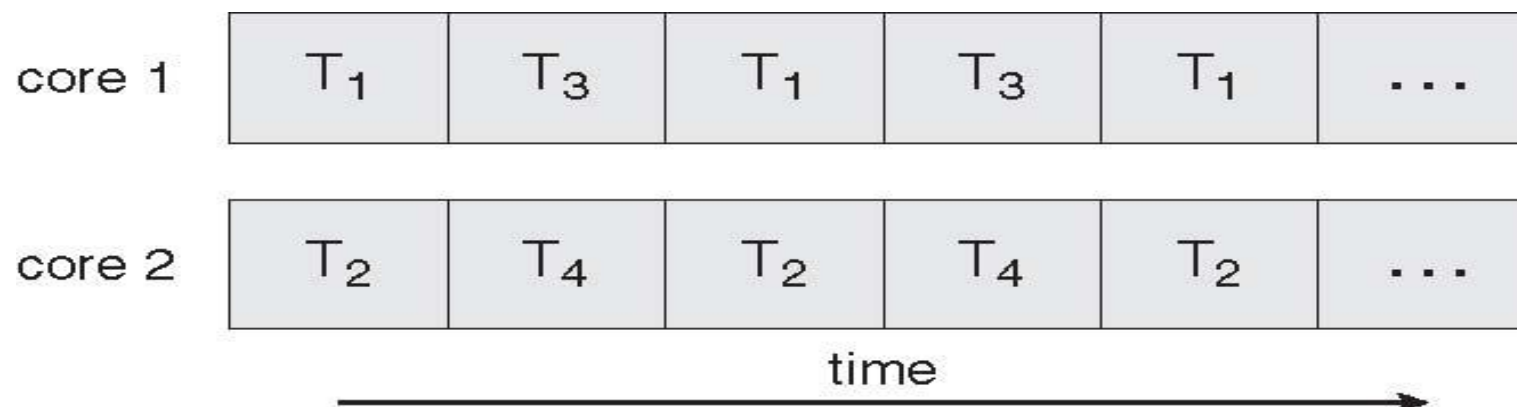
1 CPU com 1 núcleo executa cada *thread* concorrentemente

Neste exemplo dual-core, cada um das 4 *threads* é executada novamente após 4 ciclos de clock



Threads

Execução paralela - Multinúcleo



2 CPUs com 1 núcleo (ou 1 CPU com 2 núcleos) executa 2 *threads* paralelamente

Neste exemplo dual-core, cada um das 4 *threads* é executada novamente após 2 ciclos de *clock*

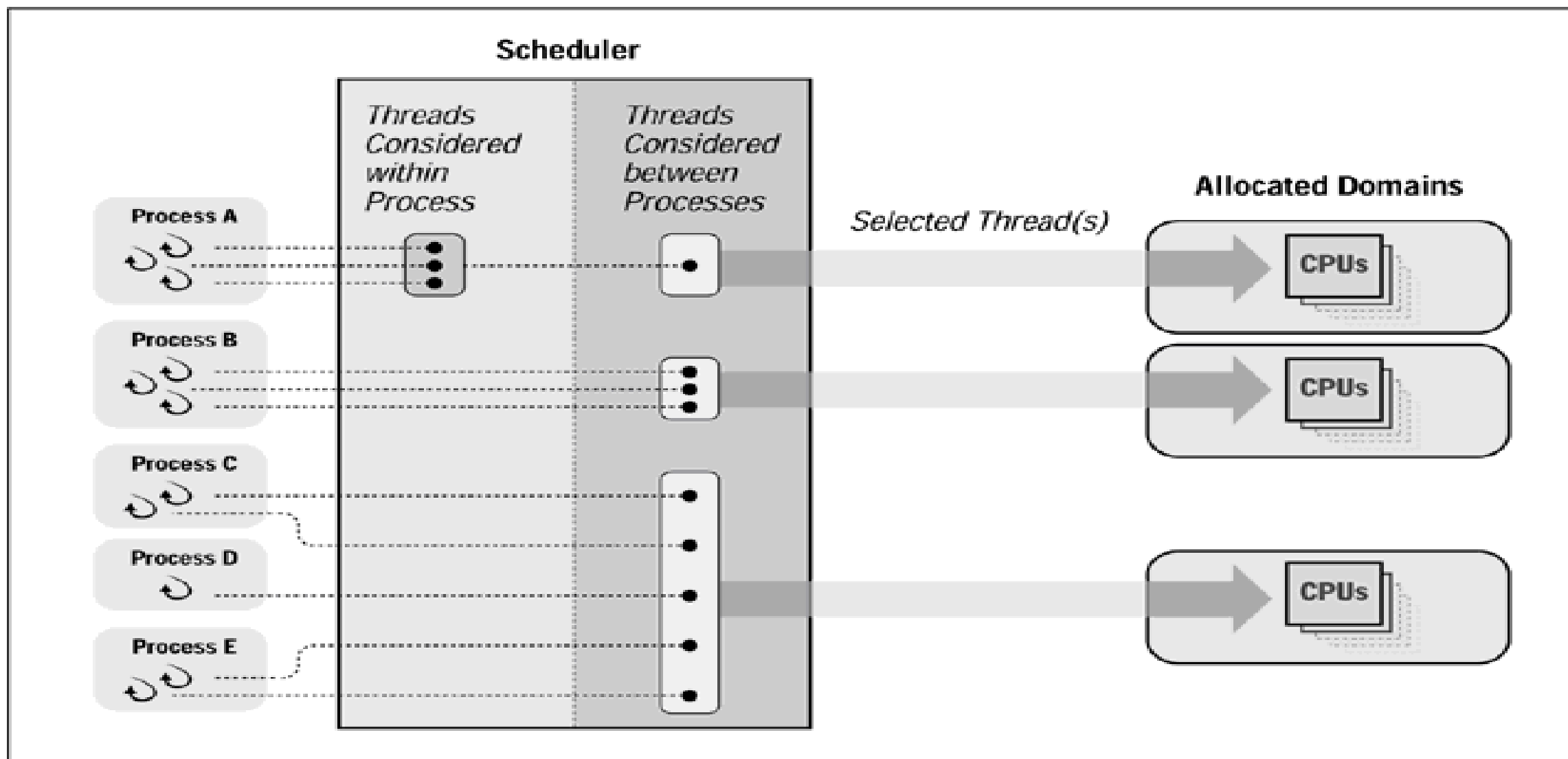
Sistemas atuais oferecem melhor desempenho com HW que melhora o desempenho de *threads*

Ex.: há CPUs que suportam 1, **2, 4 ou até 8 *threads* por núcleo**

OBS.: **cada núcleo executa 1 thread por vez**, porém as outras *threads* já estão nele carregadas, acelerando a troca de contexto entre elas

Threads

Execução paralela - Multinúcleo



- Sistemas *multicore* apresentam **novos desafios para os programadores** utilizarem melhor os múltiplos núcleos de computação:
 - **Identificar tarefas:** seções do programa que podem ser divididas entre *threads*
 - **Balanceamento:** equalizar o trabalho a ser realizado por cada *thread*
 - **Divisão dos dados:** para que possam ser acessados/manipulados em *multicore*
 - **Dependência de dados:** entre duas ou mais *threads*, cujo acesso deve ser sincronizado
 - **Teste e depuração:** muitos caminhos diferentes de execução são possíveis ao rodar o programa

Threads

Usuário *versus* Kernel

Threads do Usuário:

seu gerenciamento é feito por uma biblioteca de *threads* no nível do usuário

Três principais bibliotecas para *threads*:

POSIX *Pthreads*

Win32 threads

Java threads

Threads do Kernel

suportadas pelo Kernel

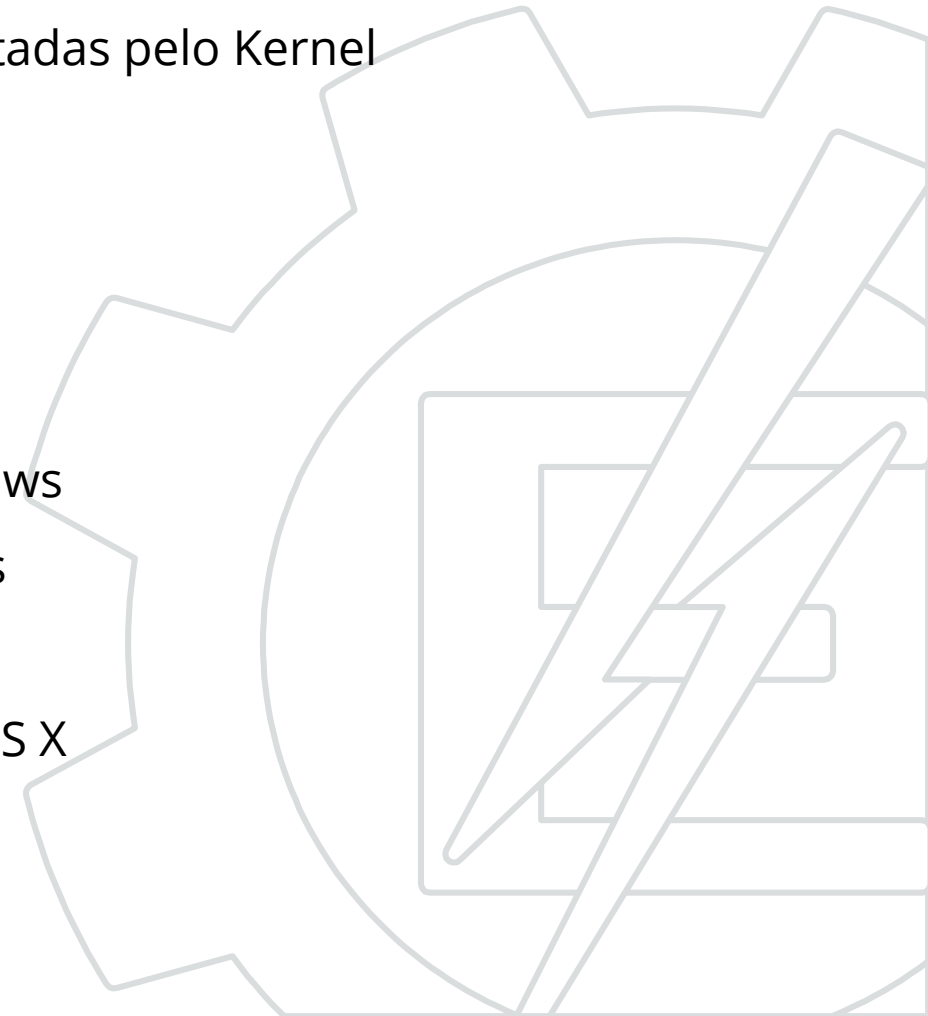
Exemplos

Windows

Solaris

Linux

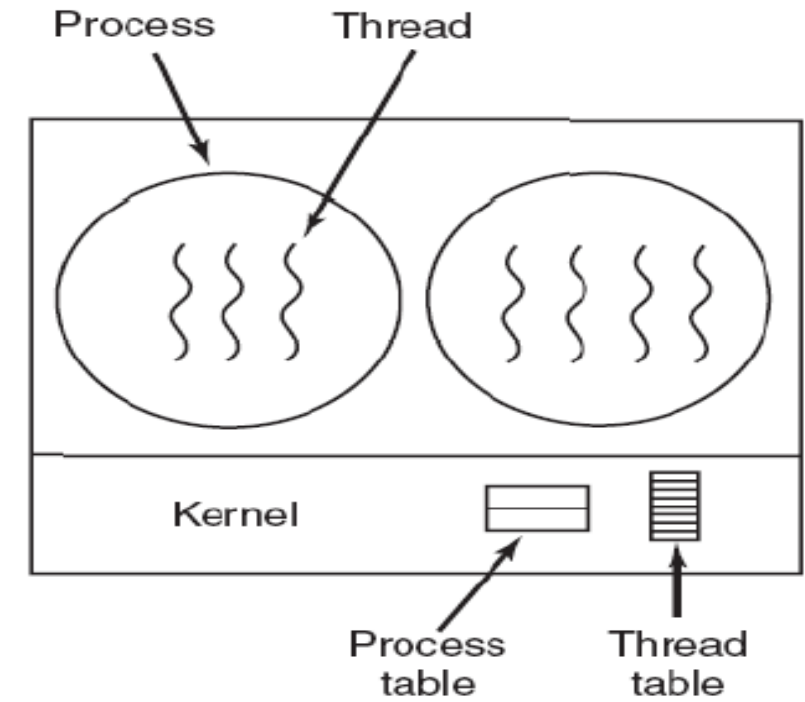
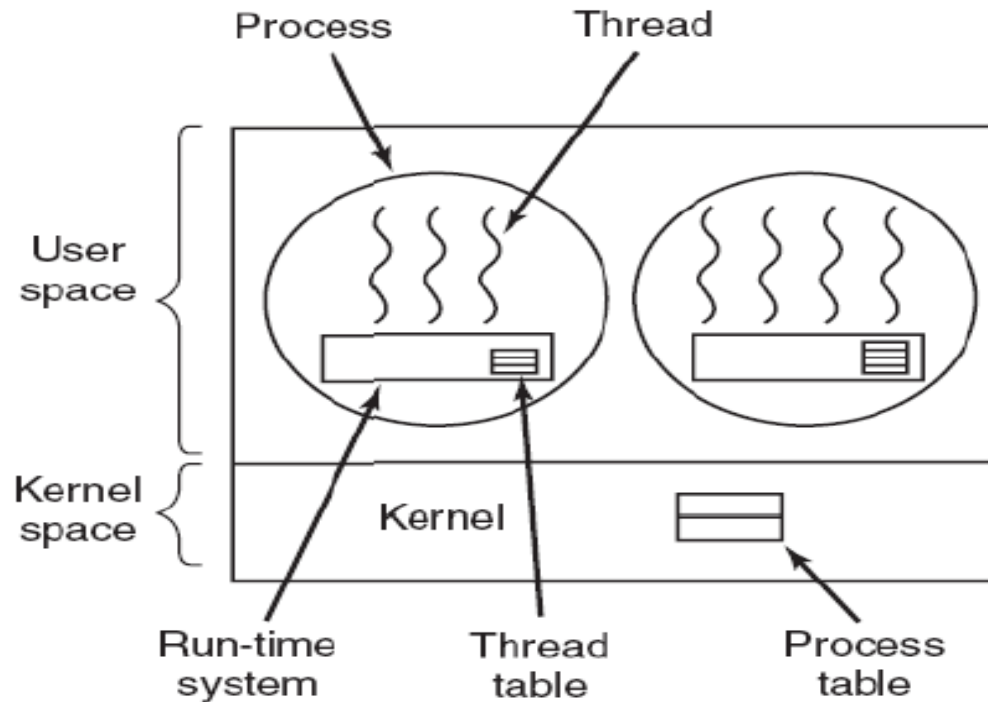
Mac OS X

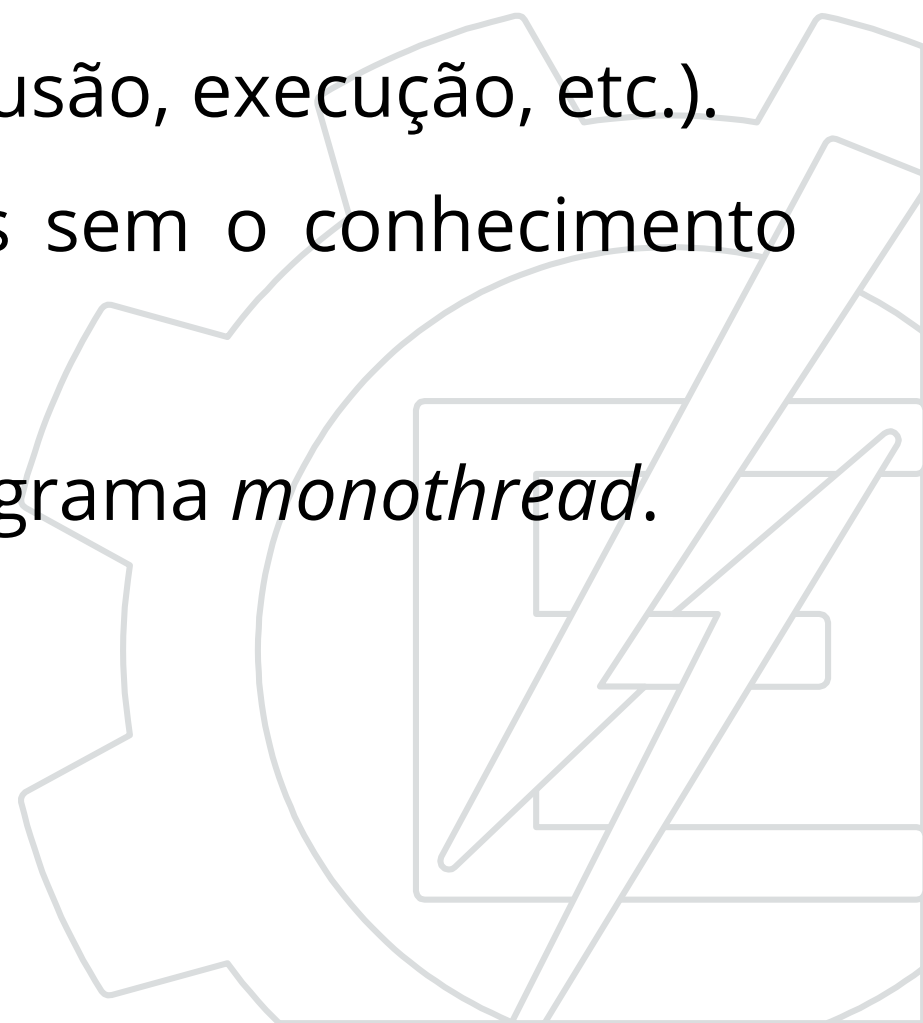


Threads

Tipos

- No modo usuário
- No modo núcleo (*kernel*)
- Híbrido

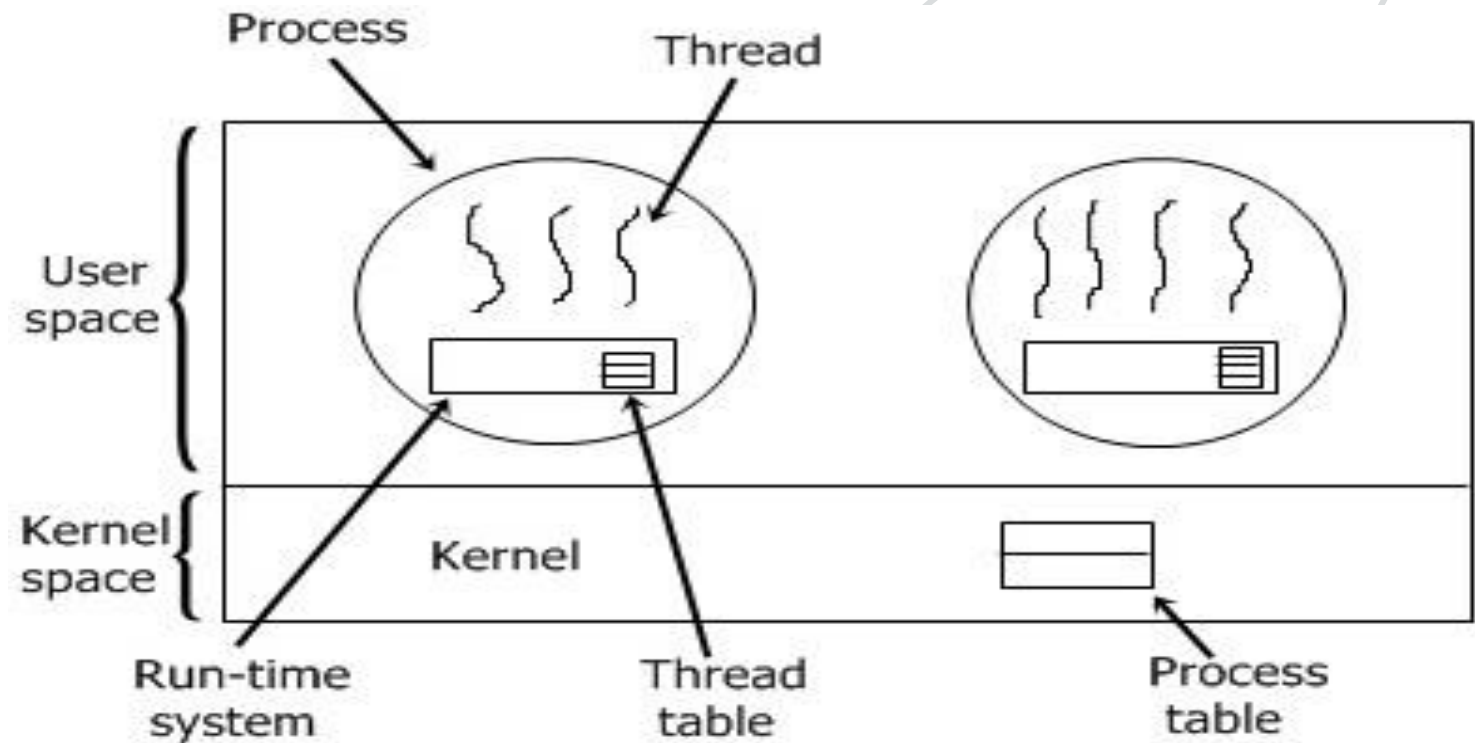


- Implementada totalmente no **espaço do usuário**.
 - Por meio de uma biblioteca (criação, exclusão, execução, etc.).
 - Criação e escalonamento são realizados sem o conhecimento do kernel.
 - Para o kernel, é como se rodasse um programa *monothread*.
 - Gerenciadas como processos no kernel.
- 

Threads

no Modo usuário

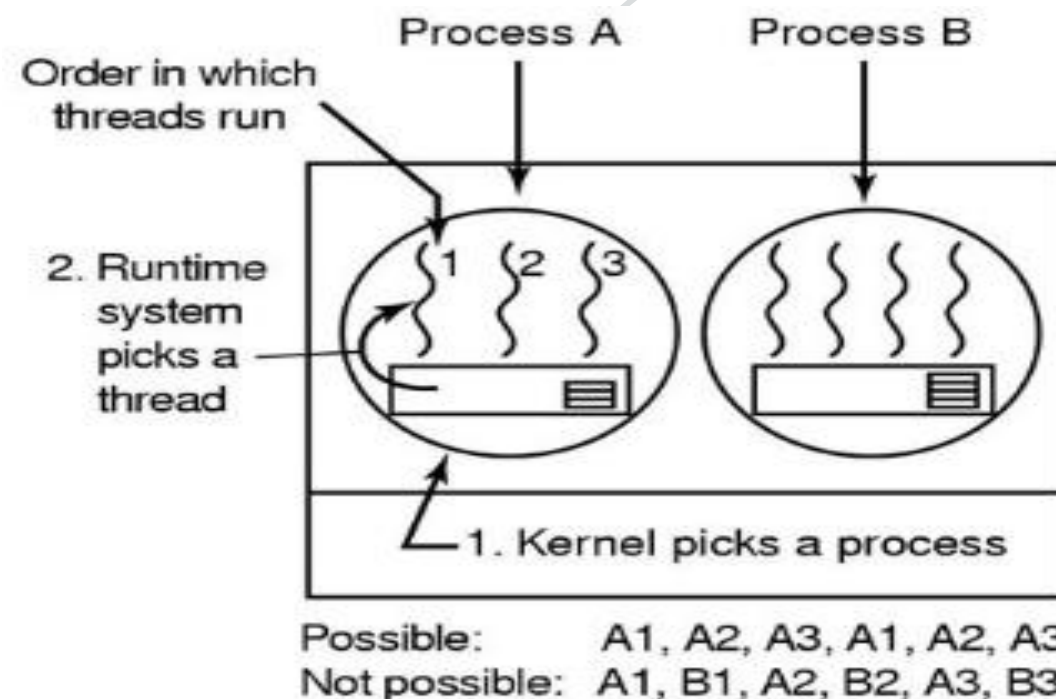
- Cada processo possui sua própria tabela de *threads*.
- Funciona como uma tabela de processos, gerenciada pelo *runtime*.
- Controle apenas as propriedades da *thread* (PC, ponteiro da pilha, registradores, estado, etc.).



Threads

no Modo usuário - Escalonamento

- O núcleo escolhe um processo e passa o controle a ele, que escolhe uma *thread*.
- A gerência da *thread* fica no espaço do usuário e o núcleo só escala em nível de processo.
- Não pode ser visualizada pelo núcleo.



Threads

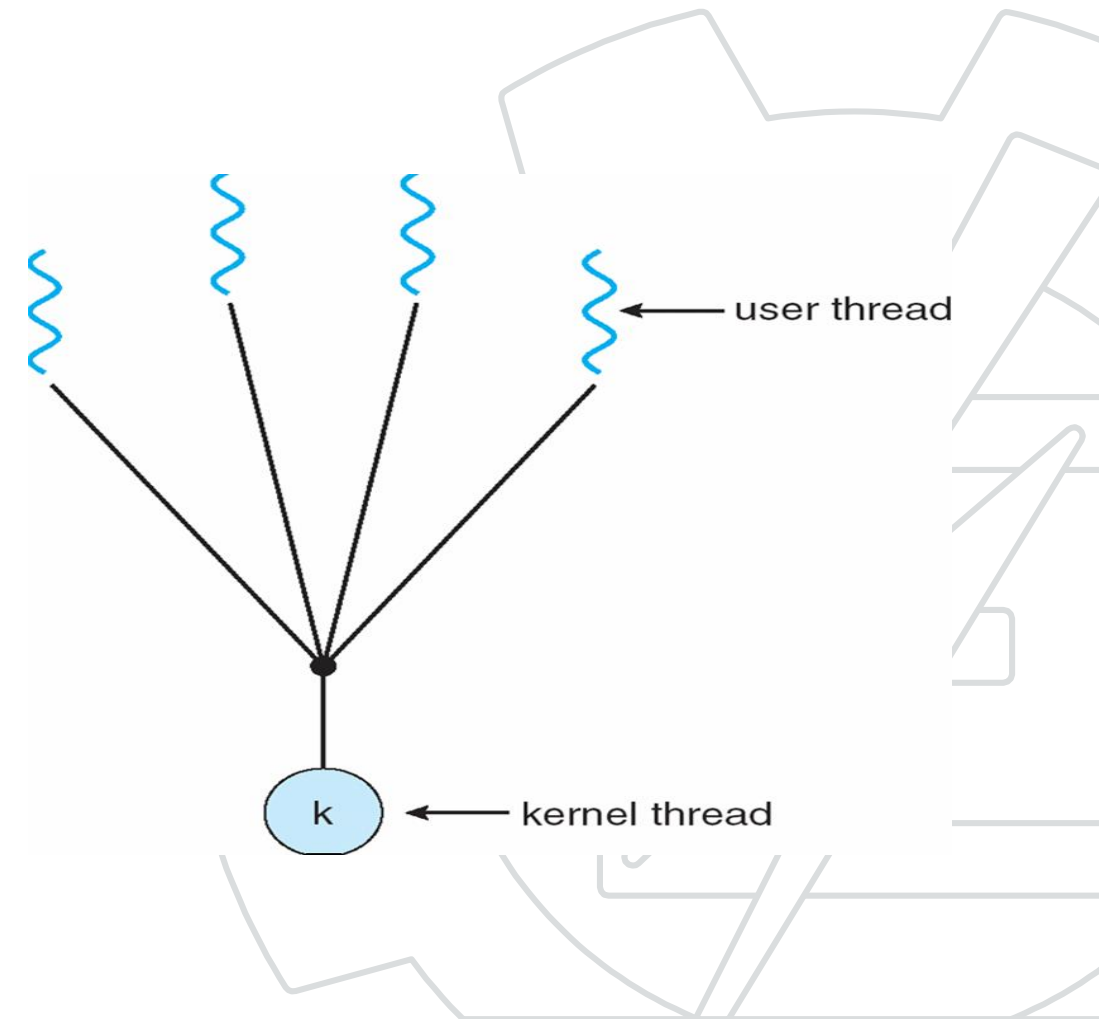
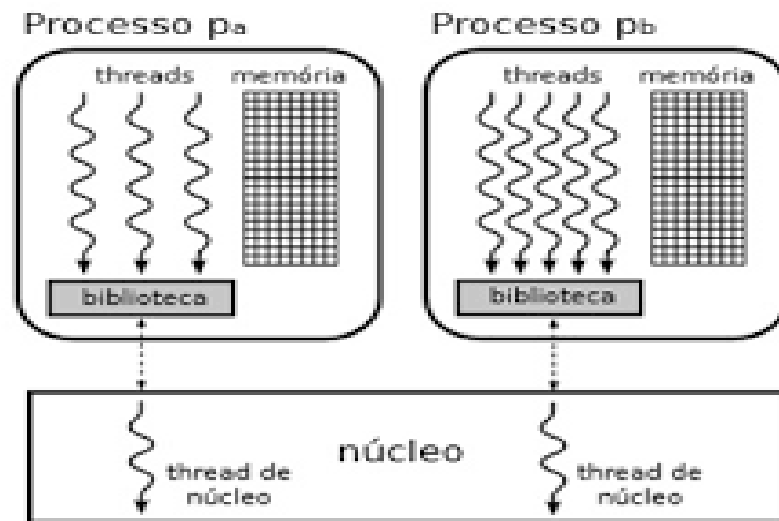
no Modo usuário – Escalonamento
Modelo N:1 (Muitos-para-um)

Muitas *threads* de nível de usuário mapeadas para uma única thread do kernel

Exemplos:

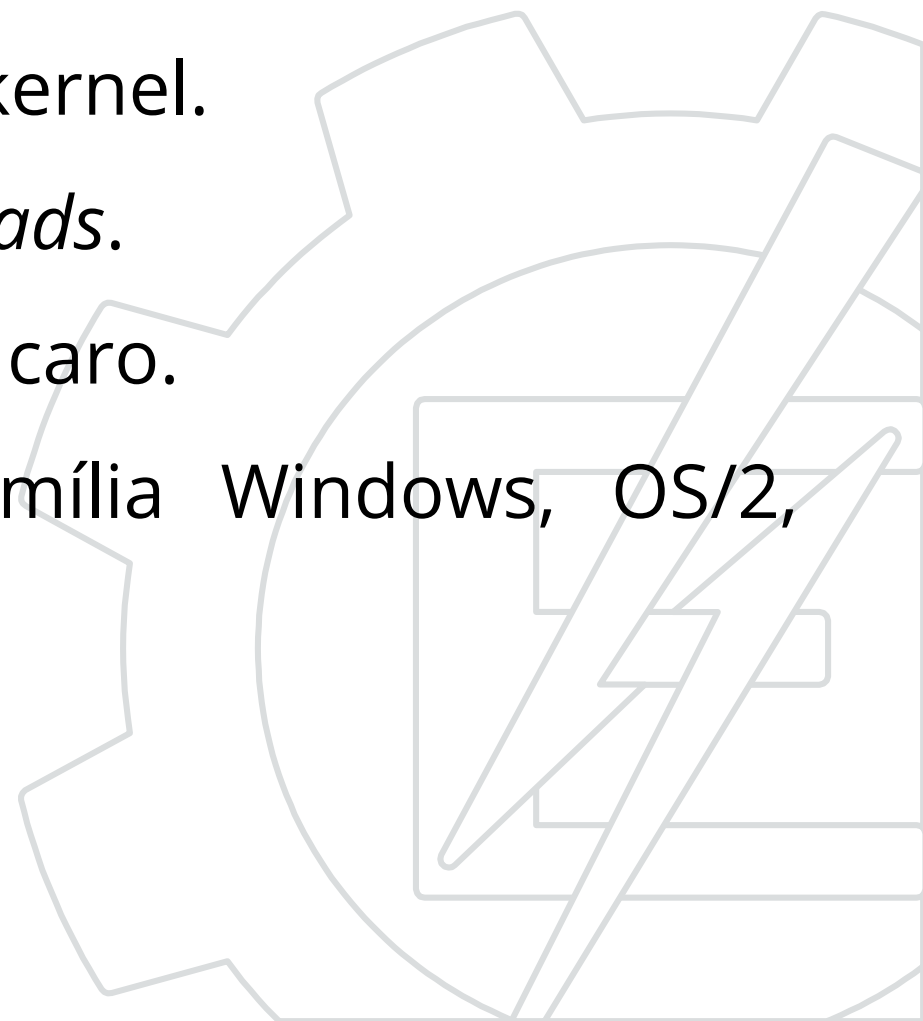
Green Threads no Solaris

Portable Threads no GNU



- Suportada diretamente pelo S.O.
- Criação, escalonamento e gerenciamento são feitos pelo kernel
- O núcleo possui tabela de *threads* (com todas as *threads* do sistema) e tabela de processos separadas.
- As tabelas de *threads* agora estão no kernel.
- Os algoritmos de escalonamento mais utilizados são *Round-Robin* e Prioridade.

- Gerenciar *threads* em modo núcleo é mais caro devido à alternância entre modo usuário e modo kernel.
- Mudança de contexto pode envolver *threads*.
- Criar e destruir *threads* no núcleo é mais caro.
- Exemplo (mapeamento 1:1): Linux, família Windows, OS/2, Solaris 9.



Threads

no Modo *kernel* – Escalonamento
Modelo 1:1 (Um-para-um)

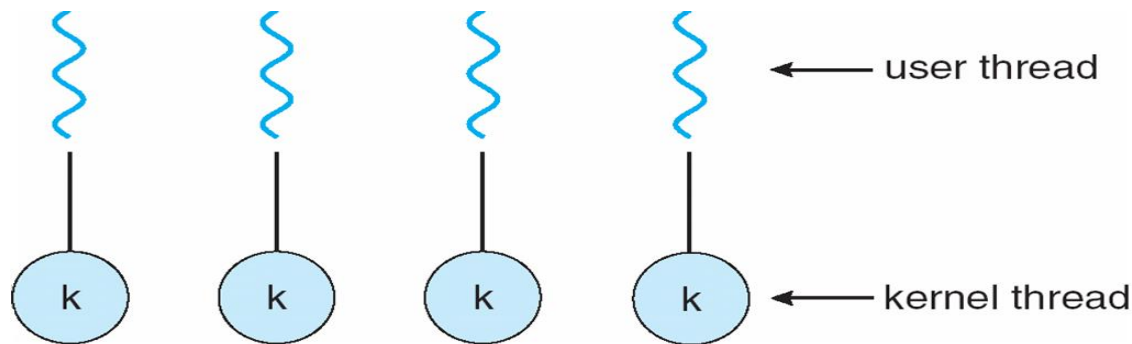
Cada *thread* de nível do usuário mapeada para uma *thread* do kernel

Exemplos

Windows NT/XP/2000

Linux

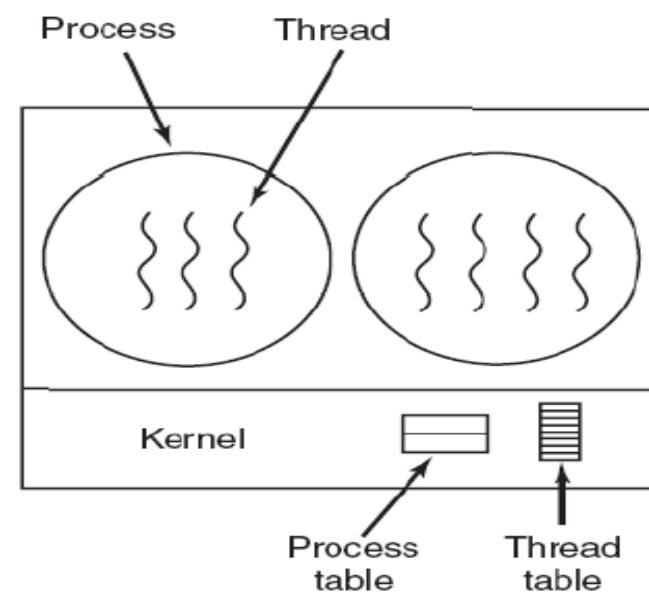
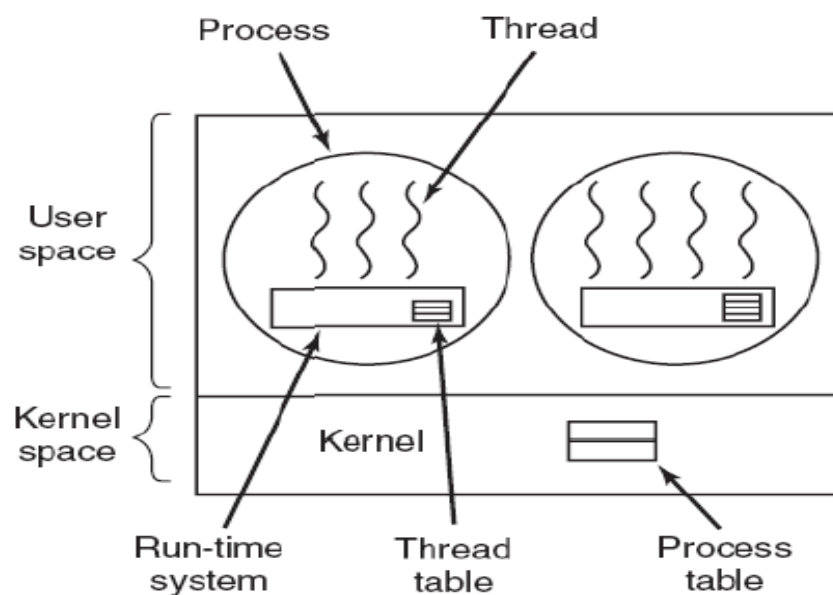
Solaris a partir da versão 9



Threads

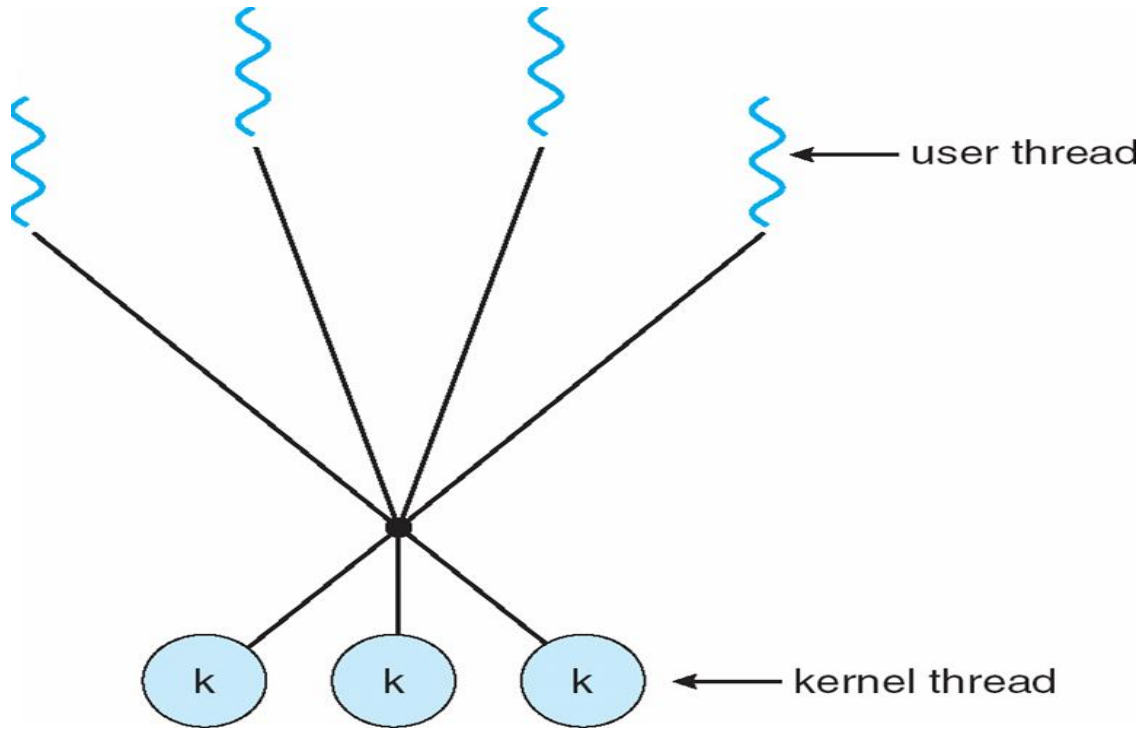
Modo *kernel* - Escalonamento

- O núcleo escolhe a *thread* diretamente.
- A *thread* é quem recebe o *quantum*, sendo suspensa se excedê-lo.
- *Thread* bloqueada por E/S não bloqueia o processo.
- Permite múltiplas *threads* em paralelo.



Multithreads

Modelo Muitos-para-muitos



Permite muitas threads do nível do usuário serem mapeados para muitas threads do kernel

Permite que o sistema operacional crie um número suficiente de threads do kernel

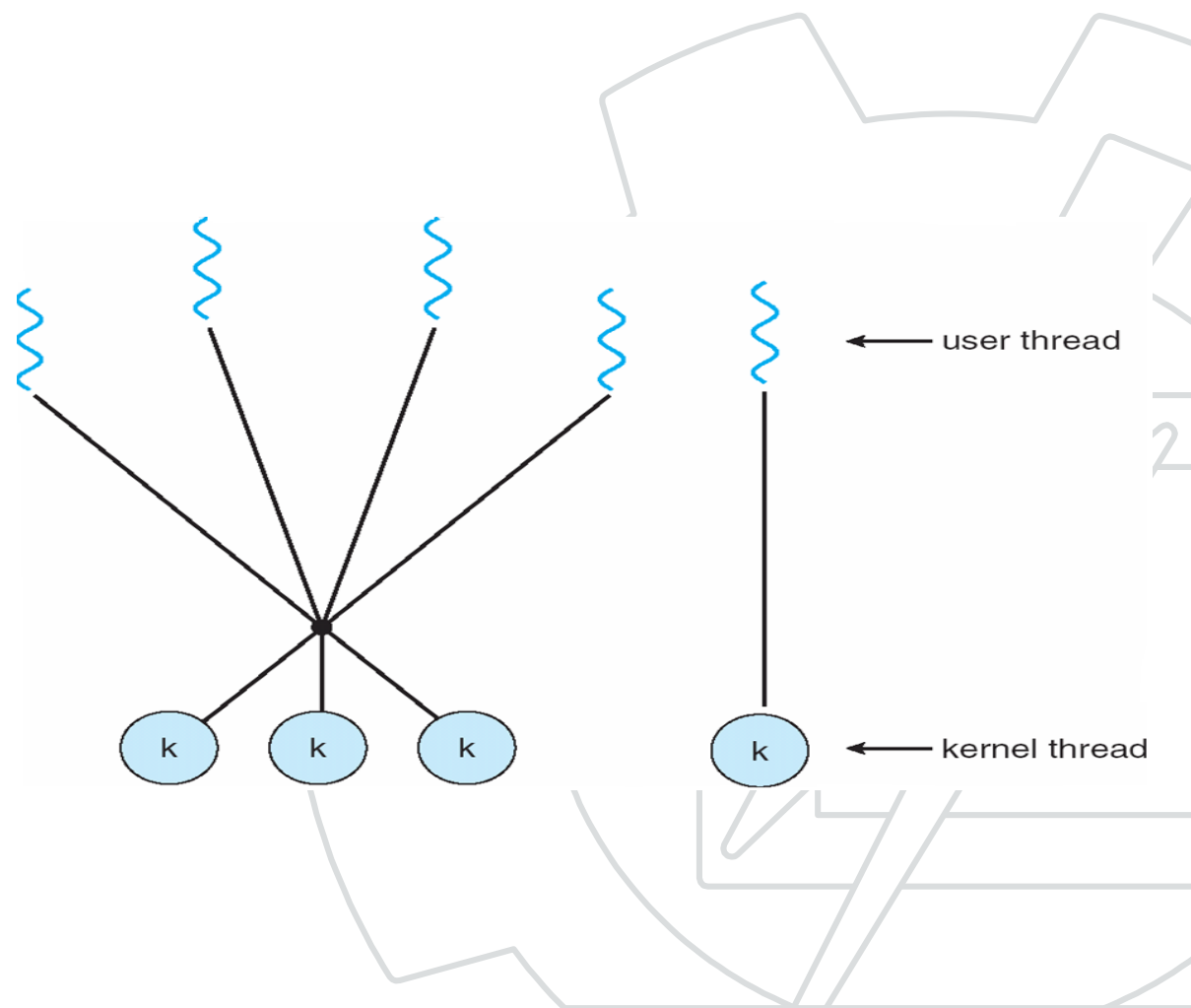
Exemplos:

Solaris, versões anteriores a 9

Windows NT/2000 com o pacote

ThreadFiber

- Similar ao modelo N:M, exceto pela permissão de uma thread do usuário ser ligada (*bound*) à thread do kernel
- Exemplos
 - IRIX
 - HP-UX
 - Tru64 UNIX
 - Solaris 8 e versões anteriores

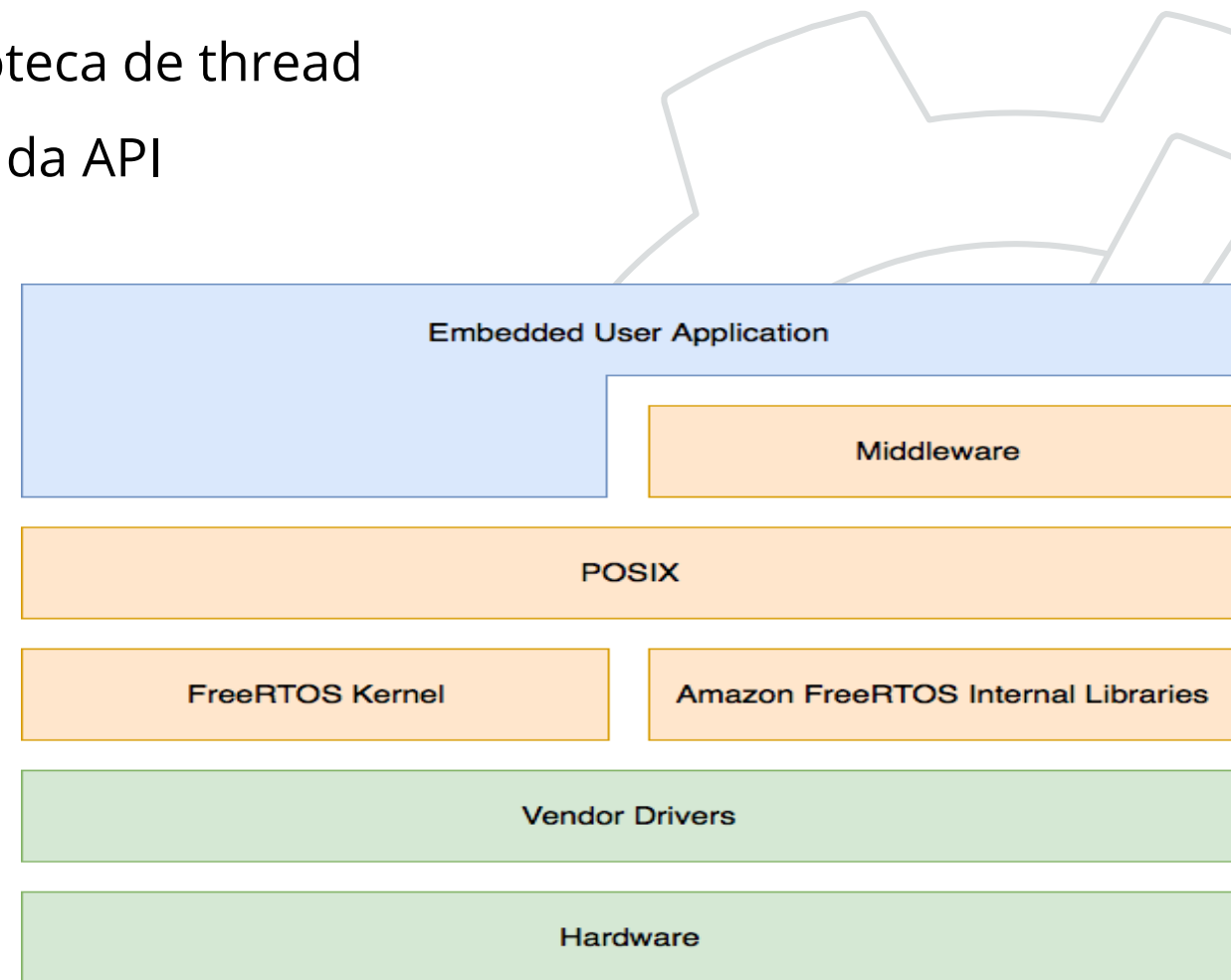


USER LEVEL THREAD	KERNEL LEVEL THREAD
User thread are implemented by users.	kernel threads are implemented by OS.
OS doesn't recognized user level threads.	Kernel threads are recognized by OS.
Implementation of User threads is easy.	Implementation of Kernel thread is complicated.
Context switch time is less.	Context switch time is more.
Context switch requires no hardware support.	Hardware support is needed.
If one user level thread perform blocking operation then entire process will be blocked.	If one kernel thread perform blocking operation then another thread can continue execution.
Example : Java thread, POSIX threads.	Example : Window Solaris.

- Fornecem ao programador uma **API** (*Application Programming Interface*) para a criação e gerência de threads
- Duas formas principais de implementação
 - Biblioteca **totalmente no espaço do usuário**
 - Biblioteca **no nível do kernel, suportada pelo S.O.**



- Uma API padrão POSIX (**IEEE 1003.1c**) para a criação e sincronização de *threads*
 - pode ser fornecida tanto no nível do usuário quanto no nível do kernel
- **API especifica o comportamento** da biblioteca de thread
- Implementação é feita no desenvolvimento da API
- Comum nos sistemas operacionais *Unix-like*
 - ex.: Linux, Mac OS X, Solaris



- *Threads* Java são gerenciadas pela JVM implementada no modelo de threads fornecido pelo S.O. subjacente
- *Threads* Java podem ser criadas:
 - Estendendo a classe *Thread*
 - Implementando a interface *Runnable*



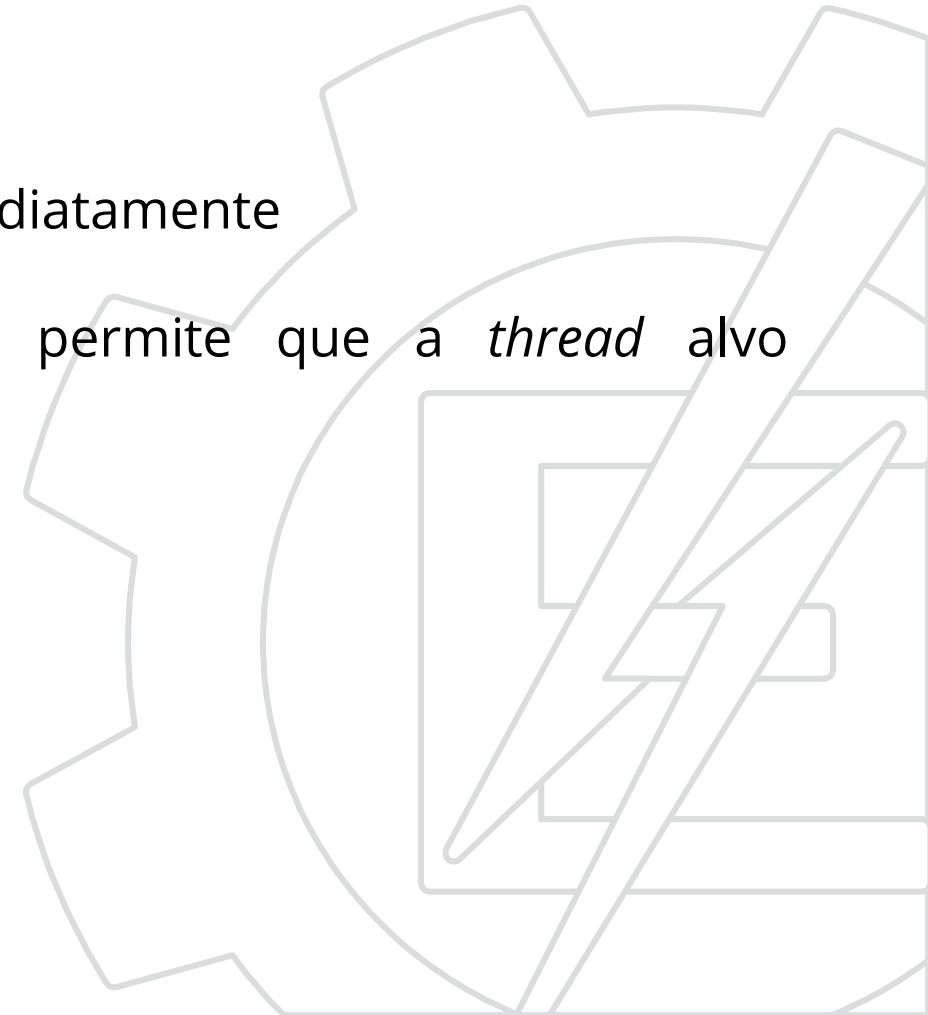
- Semântica das chamadas de sistema **fork()** e **exec()**
- Cancelamento de uma *Thread*
 - Assíncrono ou postergado (deferred)?
- Tratamento de Sinais
- *Thread pools*
- Dados específicos da *Thread*
- Ativações do Escalonador - *Scheduler activations*



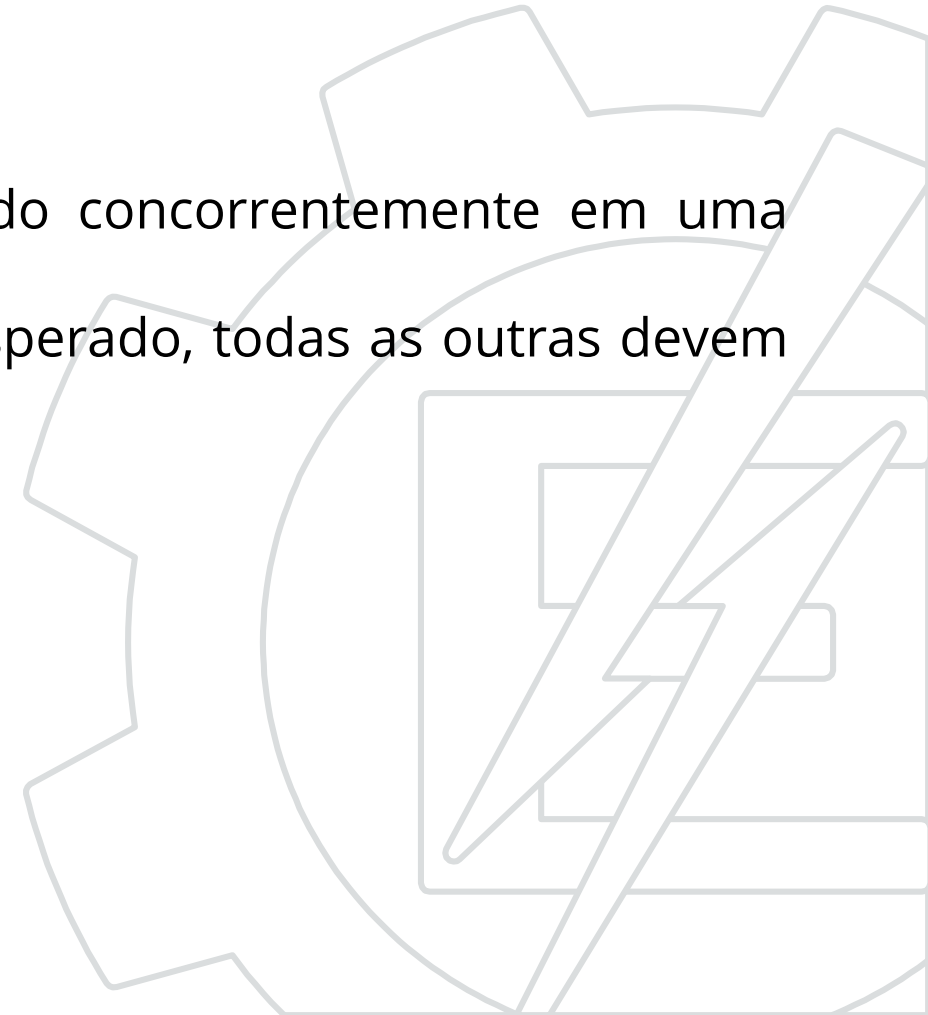
fork() duplica somente a *thread* que a chamou ou todas as *threads*?

- Se uma *thread* invoca a chamada de sistema **exec()**, o programa especificado no parâmetro irá substituir todo o processo - incluindo todas as *threads*.
- O tipo de chamada **fork()** a ser utilizada depende da aplicação. Se a chamada *exec()* é chamada imediatamente após o *fork()*, a duplicação de todas as *threads* é desnecessária, pois o programa será substituído. Neste caso, duplicar somente a *thread* que realizou a chamada é apropriado.

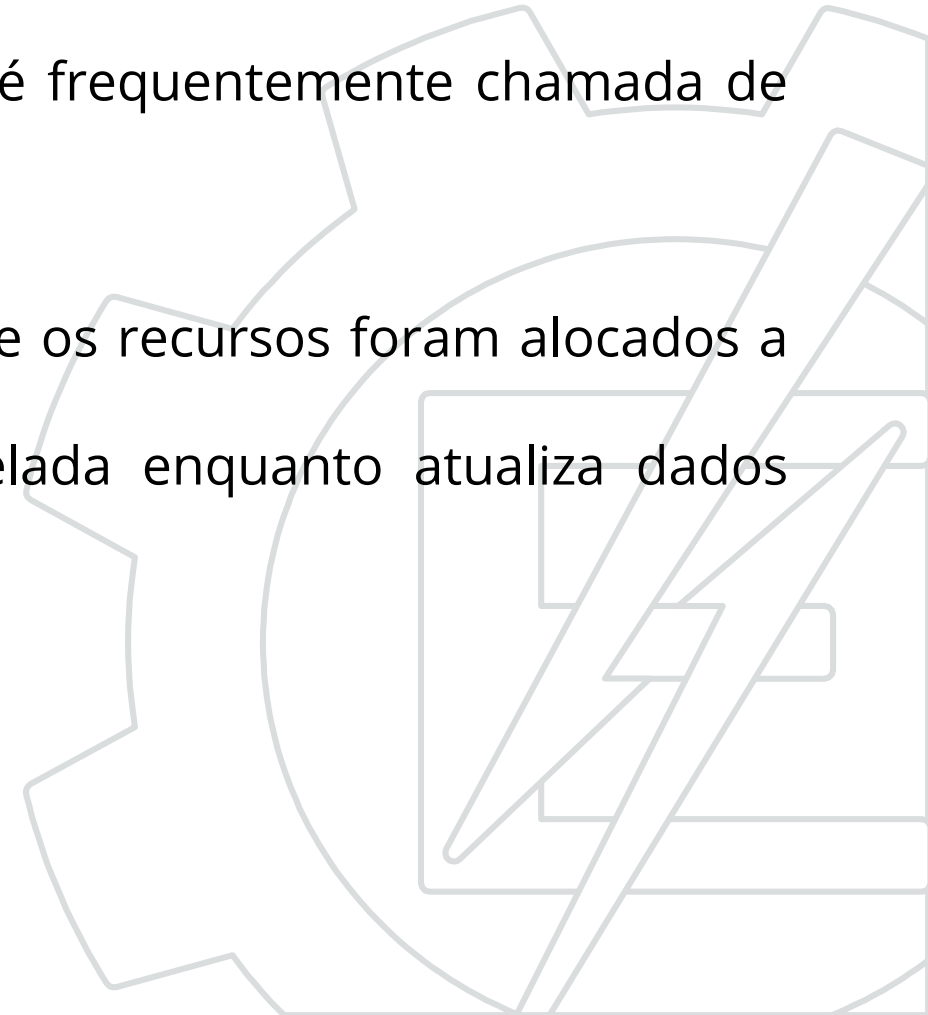
- Terminação de um *thread* antes de sua finalização
- Duas abordagens no geral:
 - **Cancelamento Assíncrono** termina a *thread*-alvo imediatamente
 - **Cancelamento Postergado** (*Deferred cancellation*) permite que a *thread* alvo periodicamente verifique se ela deve ser cancelada

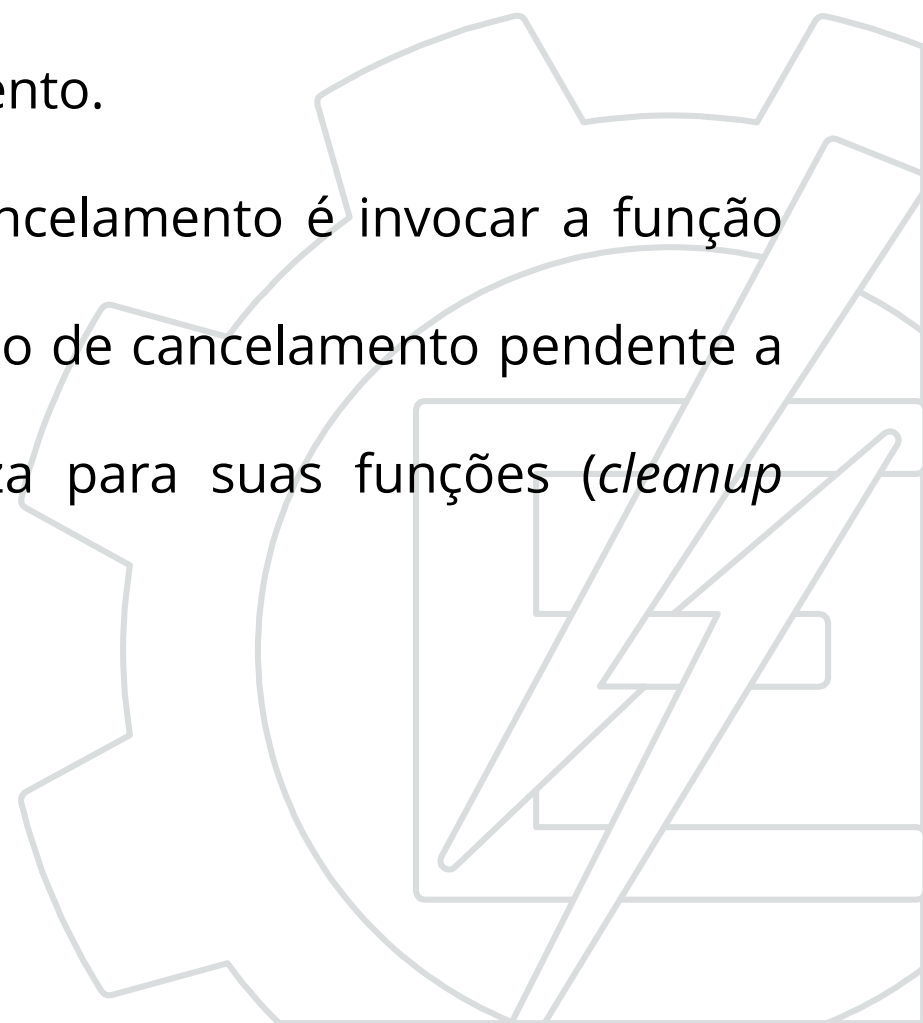


- Cancelamento de uma *thread* significa encerrá-la antes do término de sua execução, antes de completar sua tarefa.
 - Por exemplo, se múltiplas *threads* estão buscando concorrentemente em uma base de dados e uma delas retorna o resultado esperado, todas as outras devem ser canceladas.



- Outra ocorrência é quando o usuário pressiona o botão para parar o carregamento de uma página *web* no *browser*. A *thread* que é cancelada é frequentemente chamada de *thread-alvo*.
- A dificuldade de cancelamento ocorre em situações onde os recursos foram alocados a uma *thread* cancelada ou quando a *thread* é cancelada enquanto atualiza dados compartilhados com outras *threads*.

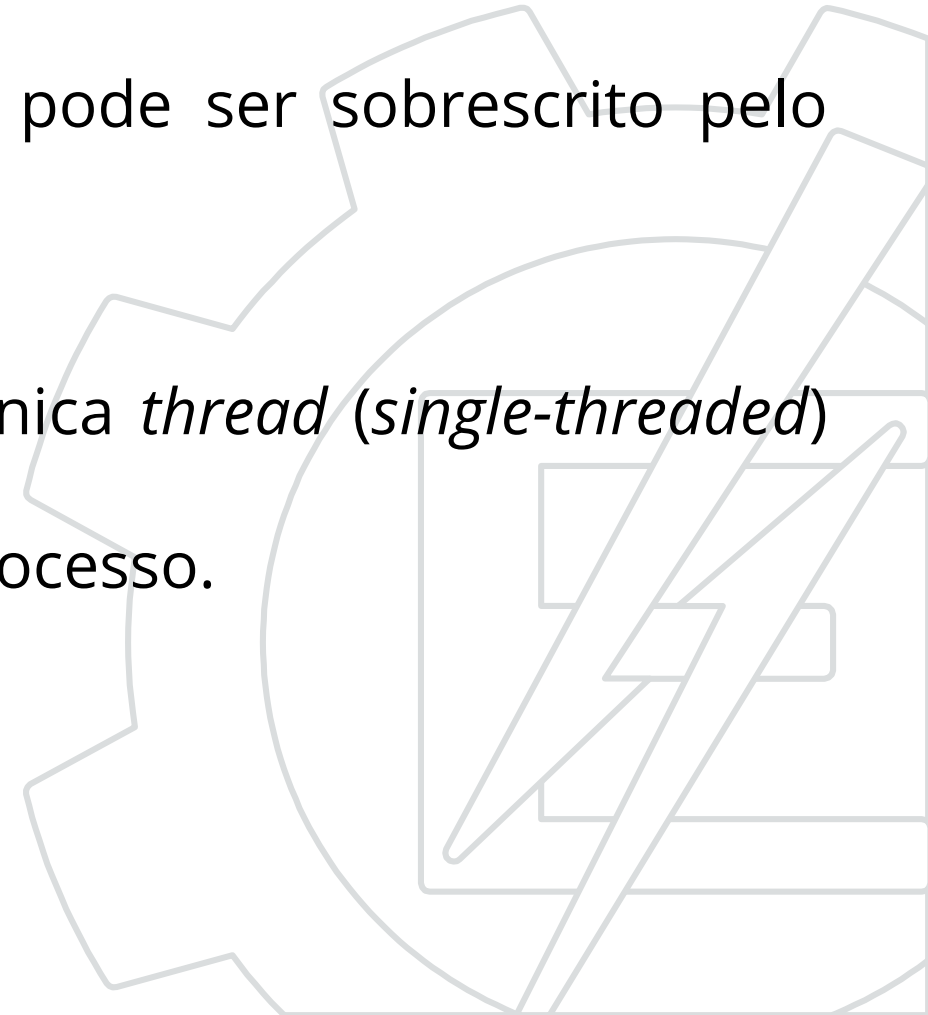


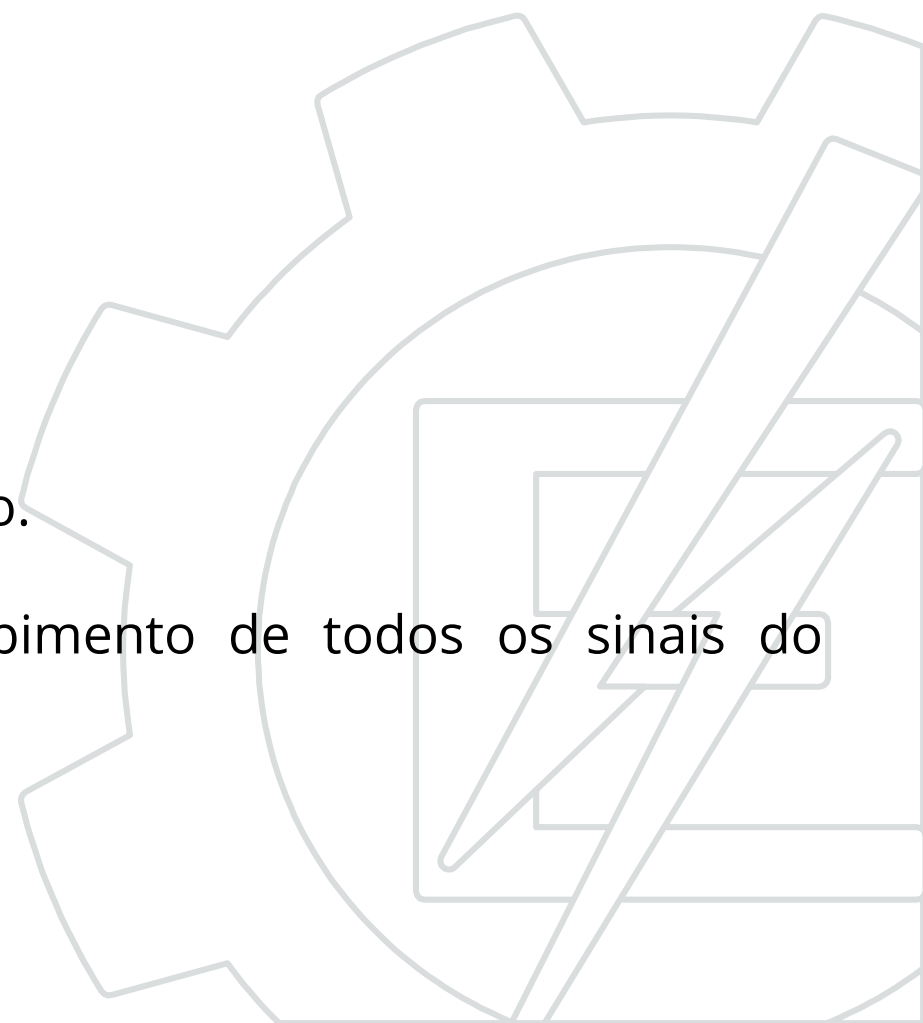
- O cancelamento-padrão é o autorizado (*deferred*). Neste caso, o cancelamento ocorre quando a *thread* alcança ou chega ao ponto de cancelamento.
 - Uma estratégia para estabelecer um ponto de cancelamento é invocar a função *pthread testcancel()*. Se é encontrada uma requisição de cancelamento pendente a função sabe como invocar um modo de limpeza para suas funções (*cleanup handler*).
- 

- Sinais são usados em sistemas UNIX para notificar um processo que um evento em particular ocorreu.
- Um tratamento de sinal é usado para processar sinais:
 1. Um sinal é gerado por um evento em particular
 2. Um sinal é entregue a um processo
 3. O sinal é tratado

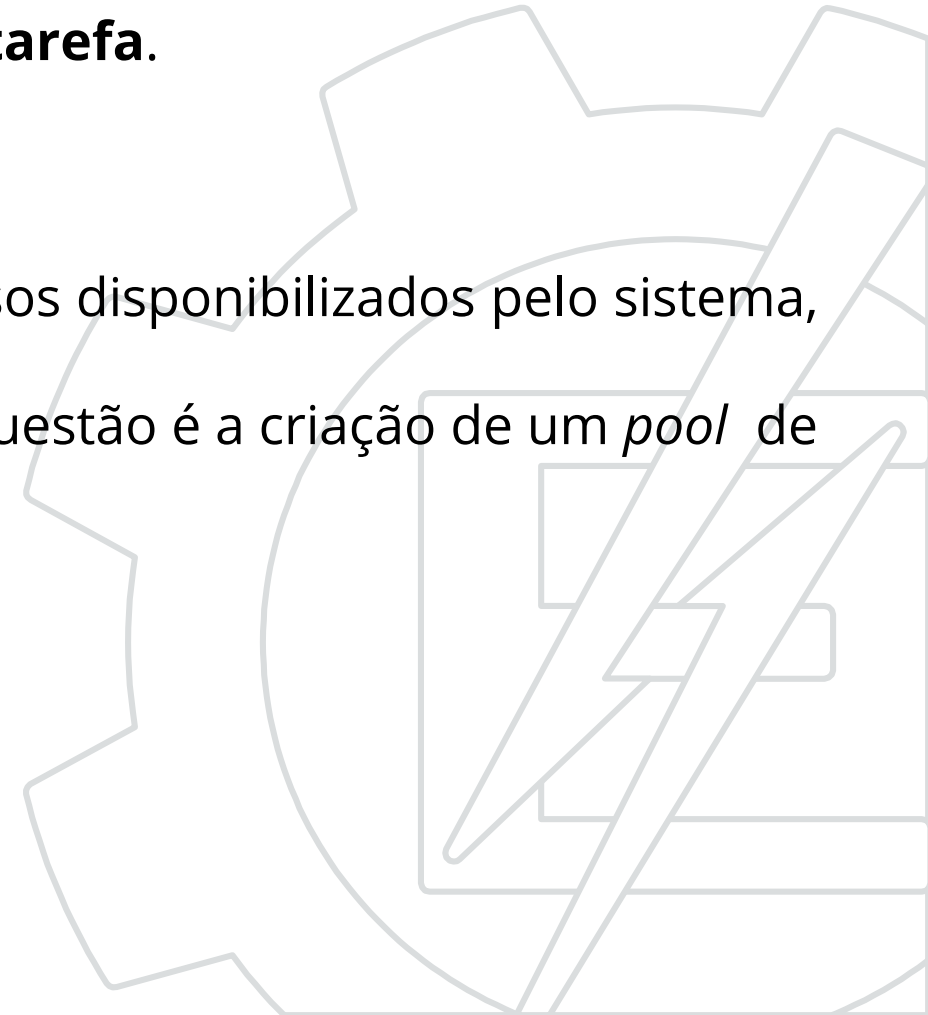


- Todo sinal possui um tratamento-padrão que o *kernel* executa quando o mesmo é recebido. O tratamento padronizado pode ser sobrescrito pelo usuário.
- Tratamentos de sinais em programas de uma única *thread* (*single-threaded*) são diretos: os sinais são sempre entregues ao processo.

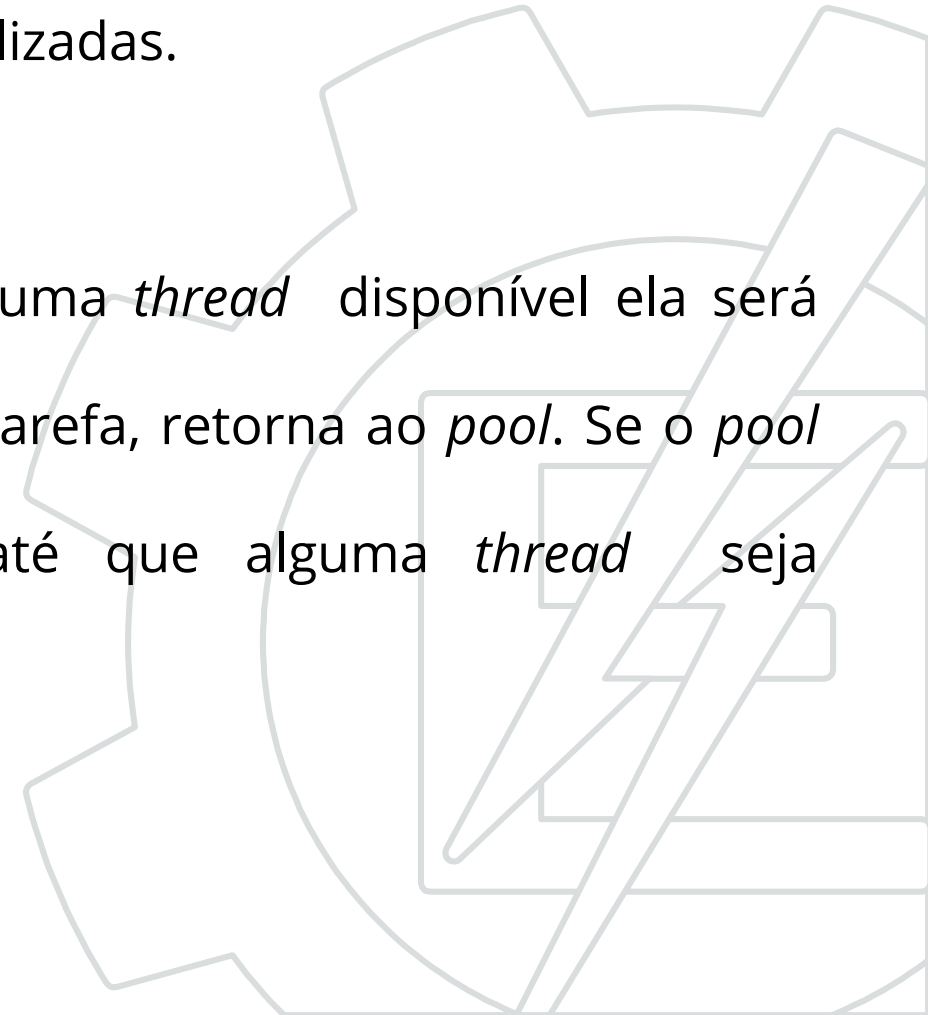


- O tratamento de sinais em programas *multithread* podem seguir as seguintes alternativas:
 - Entrega do sinal à *thread* ao qual o sinal se aplica.
 - Entrega do sinal a todas as *threads* do processo.
 - Entrega do sinal a certas/algumas *threads* do processo.
 - Definição de uma *thread* específica para o recebimento de todos os sinais do processo.
- 

- A primeira questão acerca do tempo de criação de uma *thread* está relacionada ao fato de que a mesma **será descartada após a finalização de sua tarefa**.
- A criação de inúmeras *threads* pode utilizar todos os recursos disponibilizados pelo sistema, como tempo de CPU ou memória. Uma solução para esta questão é a criação de um *pool* de *threads*.



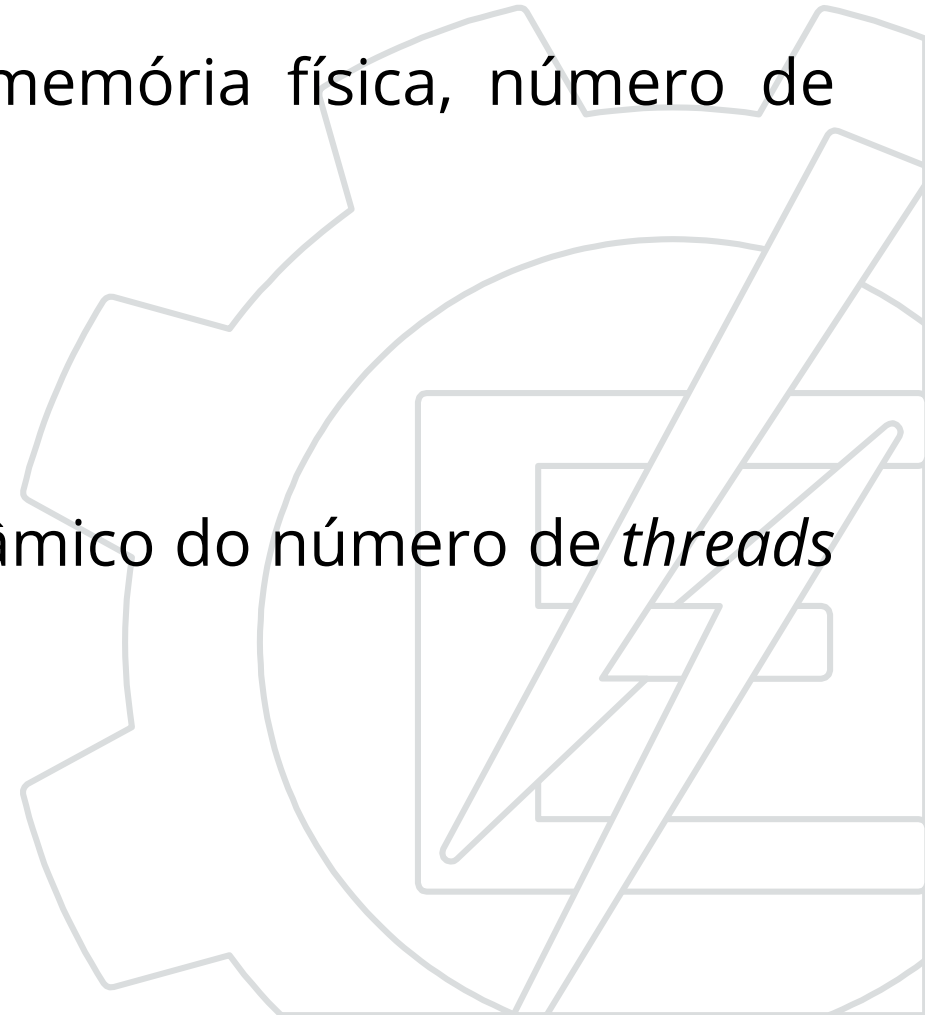
- A idéia geral é a criação de um certo número de *threads* no processo de inicialização e alocá-las neste *pool*, onde elas irão aguardar para serem utilizadas.
- Quando o servidor receber uma requisição, caso existe uma *thread* disponível ela será acordada e executará a requisição. Assim que encerrar a tarefa, retorna ao *pool*. Se o *pool* não possuir *threads* disponíveis o servidor aguarda até que alguma *thread* seja disponibilizada.

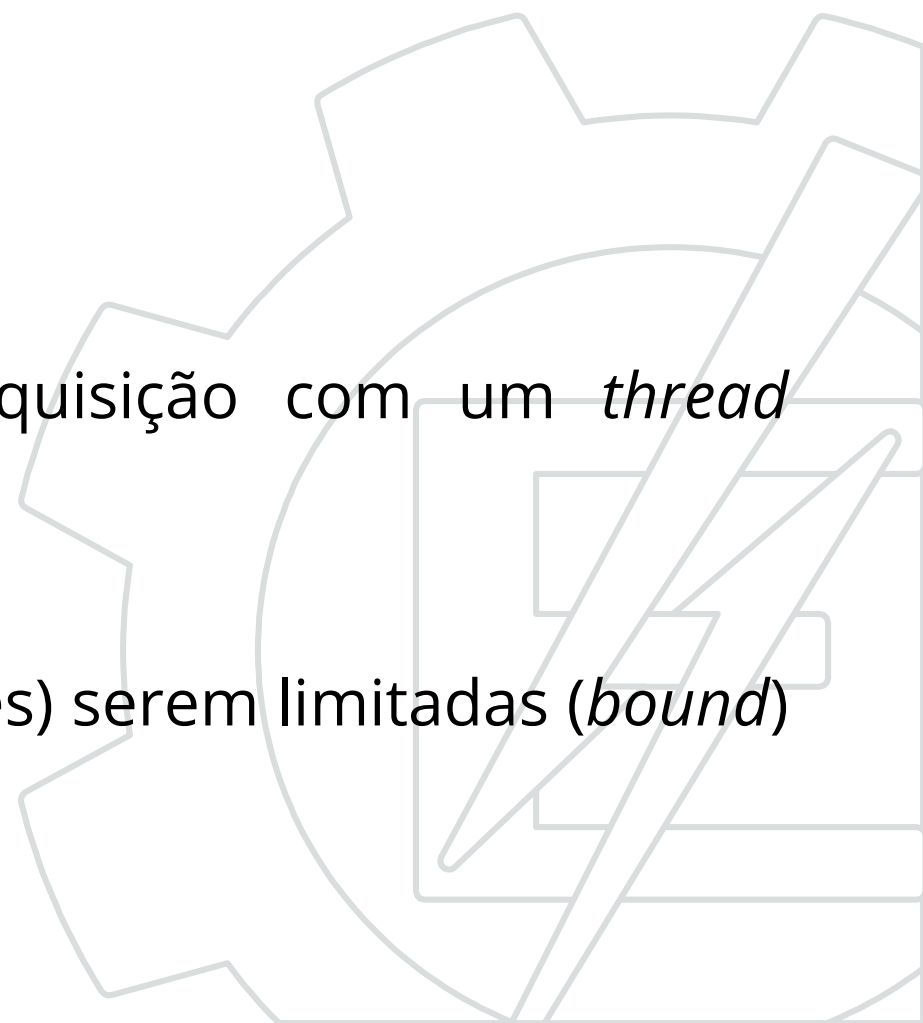


◦ *Pools de Threads* oferecem os seguintes benefícios:

- **Responder** a uma requisição com uma *thread* existente é mais rápido do que aguardar pela **criação** de uma *thread*.
- O *pool de threads* **limita o número de requisições** a serem atendidas simultaneamente, respeitando a capacidade de processamento do sistema.
- Separar a tarefa a ser executada do mecanismo de criação de tarefas permite o uso de diferentes estratégias para a resolução das requisições.
 - Por exemplo, uma tarefa pode ser agendada para executar após um tempo de espera (*delay*) ou pode ser executada periodicamente.

- O número de *threads* pode ser determinado de modo heurístico baseado no número de CPUs do sistema, quantidade de memória física, número de clientes e requisições esperado, etc.
- *Pools* mais sofisticados podem sofrer ajuste dinâmico do número de *threads* de acordo com algumas regras e padrões.

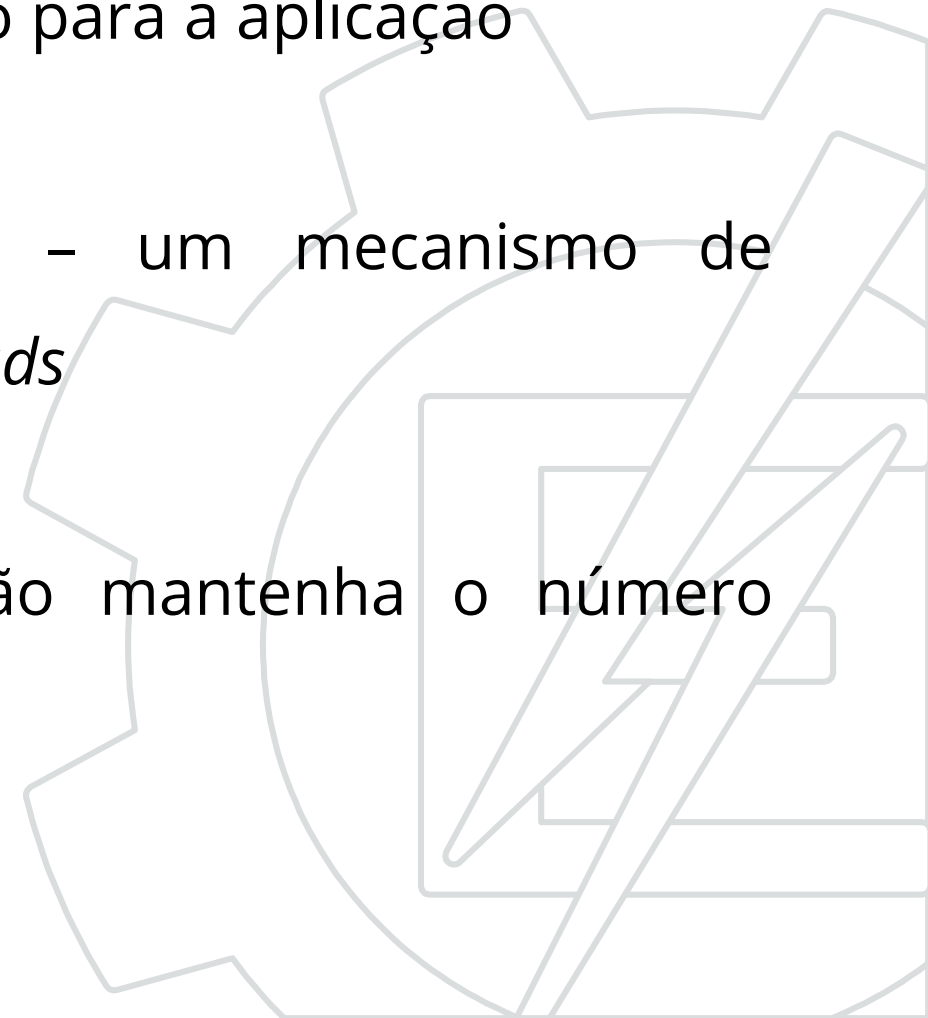


- Cria um número de *threads* em um pool onde aguardam por trabalho
 - Vantagens:
 - Normalmente mais rápido servir uma requisição com um *thread* existente do que criar uma nova *thread*
 - Permite o número de *threads* da aplicação(ões) serem limitadas (*bound*) ao tamanho do *pool*
- 

- **Grand Central Dispatch (GCD)** - é uma tecnologia para sistemas operacionais da Apple - Mac OS X e iOS. Ela permite que desenvolvedores de aplicações identifiquem seções do código que podem ser executadas em paralelo.
- **OpenMP** é um conjunto de diretivas de compiladores que fornecem suporte à programação paralela em ambientes de memória compartilhada.

- Em algumas situações as *threads* podem necessitar de suas próprias cópias dos dados. Este fato é chamado de **Thread-local storage (TLS)**.
- É fácil confundir TLS com variáveis locais:
 - Enquanto variáveis locais são visíveis apenas durante a invocação simples da função, TLS são visíveis além da invocação da função.
 - Em algumas situações, TLS são similares a dados estáticos (*static data*).
 - A diferença é que TLS é único para cada *thread*. A maioria das bibliotecas de *threads* - incluindo Windows e *Pthreads* - fornecem suporte a TLS, assim como Java fornece.

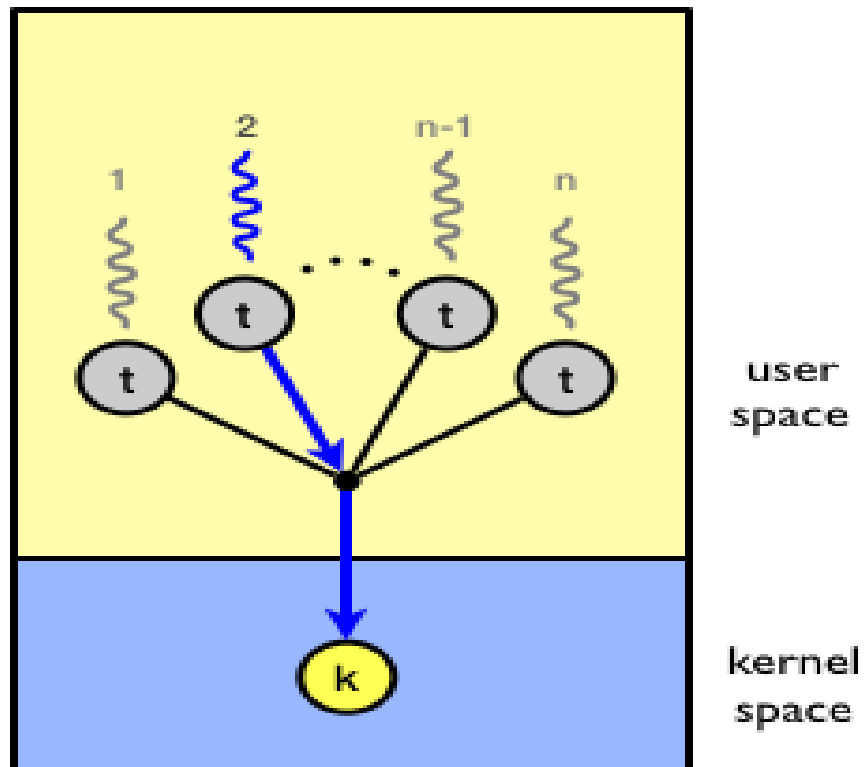
- Ambos modelos N:M e Dois-Níveis, requerem comunicação para manter o número apropriado de *threads* do kernel alocado para a aplicação
- Ativação de escalonadores fornece ***upcalls*** – um mecanismo de comunicação do *kernel* para a biblioteca de *threads*
- Esta comunicação permite que uma aplicação mantenha o número correto de *threads* do kernel



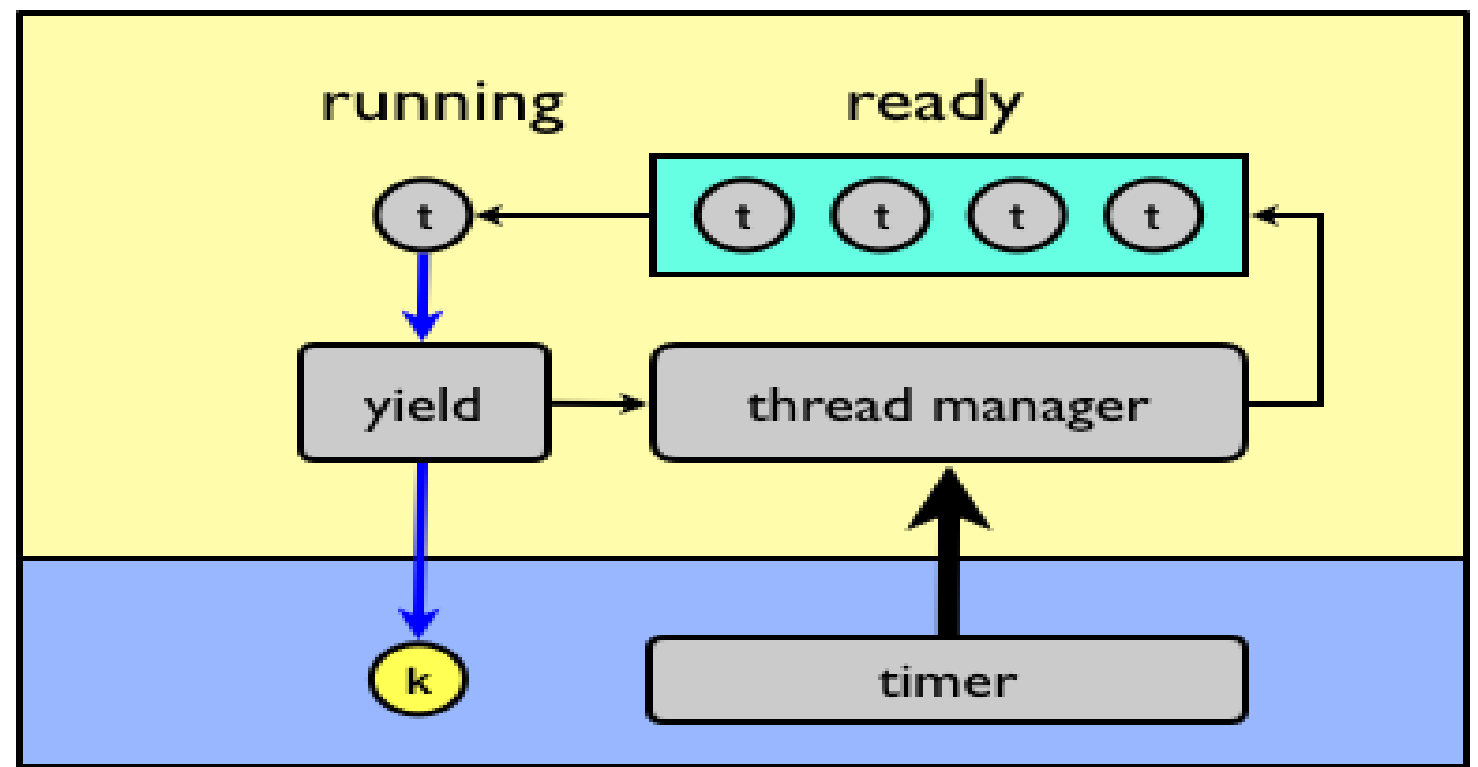
Thread

Ativação de escalonadores

many-to-one
user-level threads



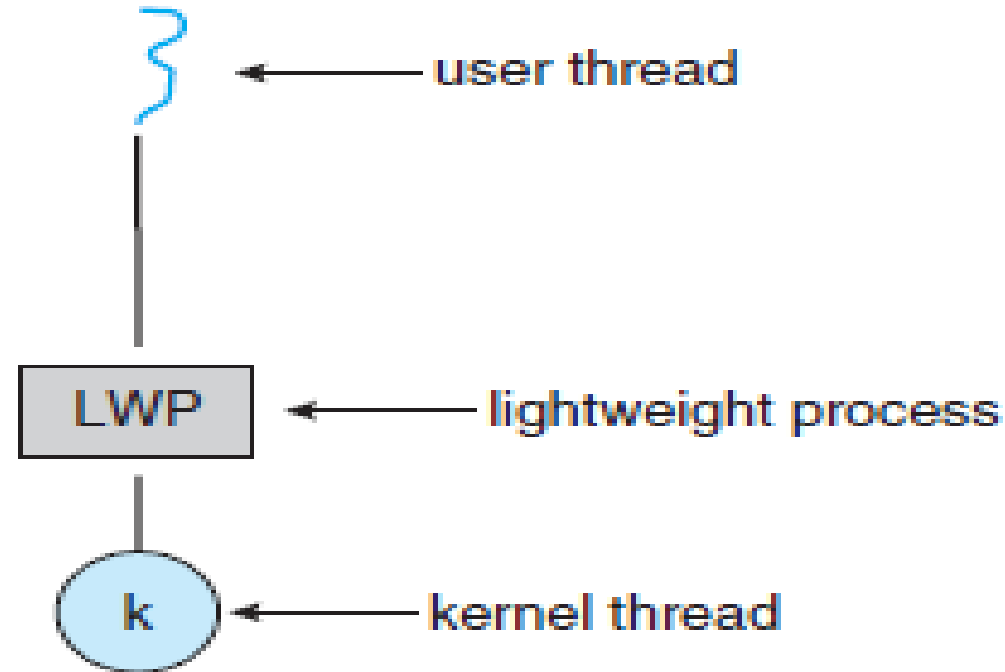
cooperative and preemptive scheduling
of many-to-one user level threads



Thread

Ativação de escalonadores

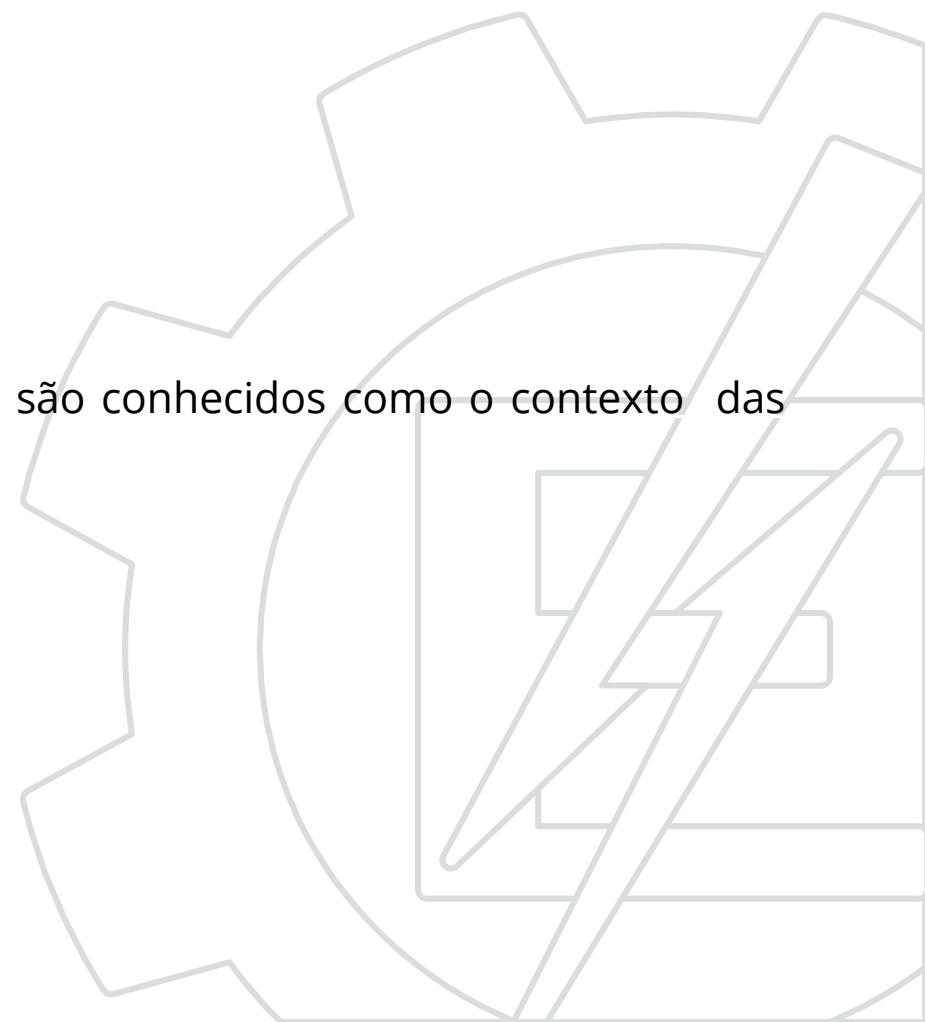
- Esta estrutura de dados é tipicamente conhecida como **lightweight process (LWP)**. Para a biblioteca de *threads*, o LWP funciona como um processador virtual onde a aplicação pode escalonar a execução de uma thread



- *Threads* no Windows XP
- *Threads* no Linux

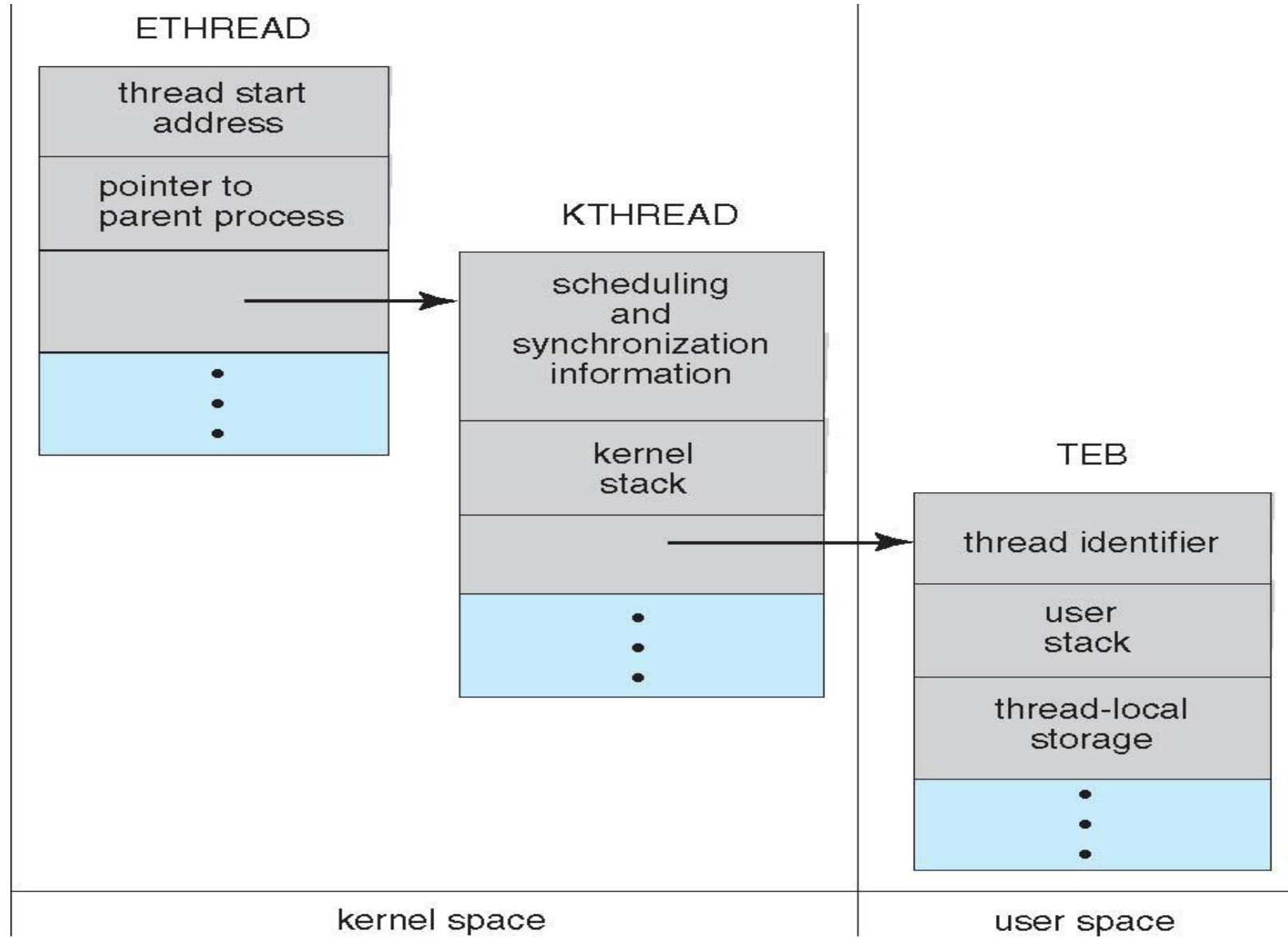


- Implementa o mapeamento um-para-um (1:1), nível do kernel
- Cada *thread* contém
 - Um identificador da *thread*
 - Um conjunto de registradores
 - Pilhas separadas para o usuário e o *kernel*
 - Área privada de armazenamento de dados
- O conjunto de registradores, pilhas e área privada de armazenamento são conhecidos como o contexto das *threads*
- As principais estruturas de dados de uma thread são:
 - ETHREAD (*executive thread block*)
 - KTHREAD (*kernel thread block*)
 - TEB (*thread environment block*)

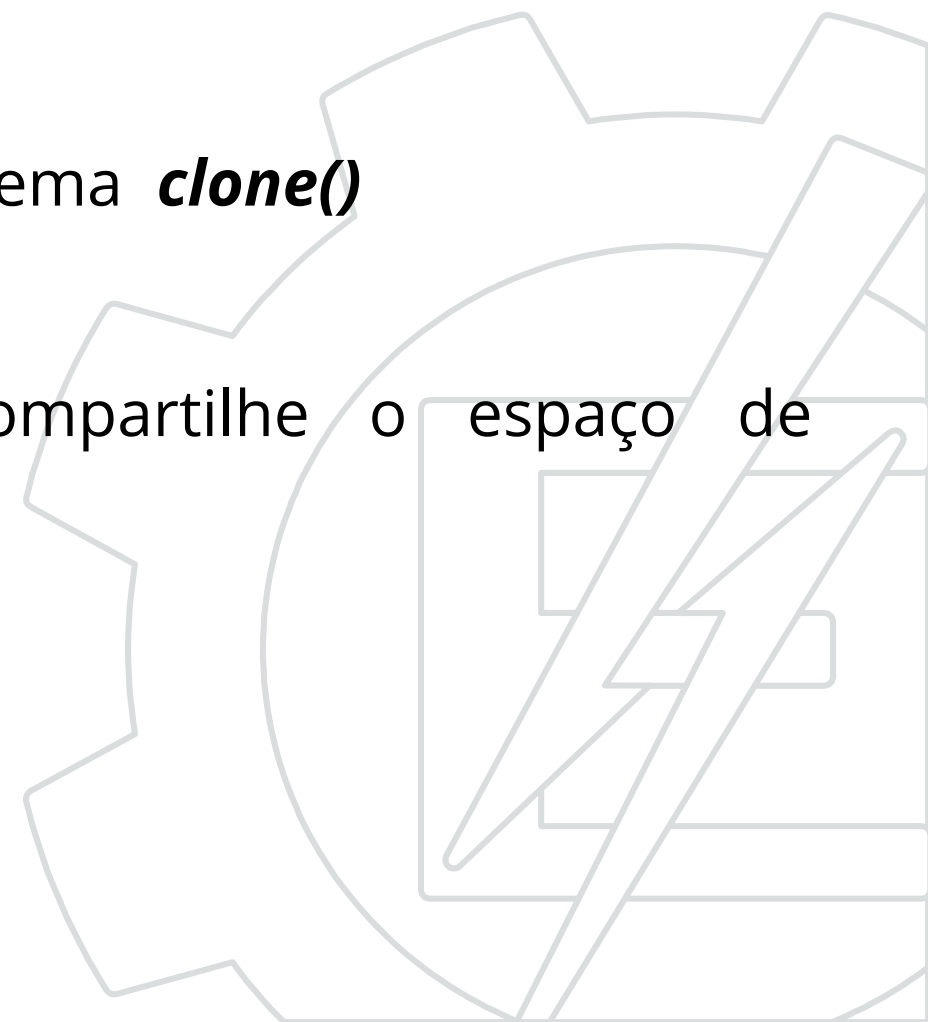


Threads

Windows XP



- Linux refere-se à elas como **tarefas** (*tasks*) em vez de *threads*
- A criação de *threads* é feita via chamada de sistema **clone()**
- **clone()** permite que uma tarefa filha compartilhe o espaço de endereçamento da tarefa pai (processo)



flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Bibliografia

- TANENBAUM, Andrew S; BOS, Herbert. Sistemas operacionais modernos. 4a ed. São Paulo: Pearson Education do Brasil, 2016.

Capítulo 2.

<https://plataforma.bvirtual.com.br/Acervo/Publicacao/1233>

- DEITEL, H.M; DEITEL, P.J; CHOFFNES,D.R. Sistemas Operacionais. 3a ed. São Paulo: Pearson Prentice Hall, 2005. **Capítulo 4.**

<https://plataforma.bvirtual.com.br/Acervo/Publicacao/315>



Sistemas Operacionais

Prof. Otávio Gomes

otavio.gomes@unifei.edu.br

