Sistemas Operacionais

Aula 04 - Threads

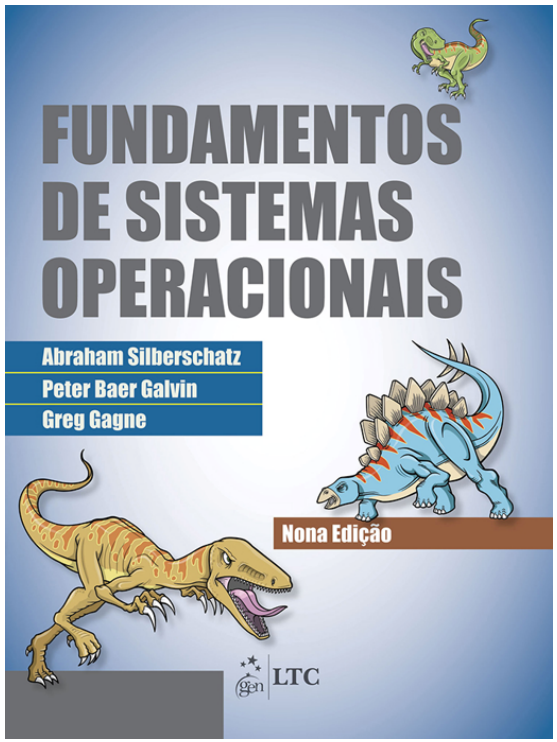
Prof. Samuel Souza Brito

Universidade Federal de Ouro Preto – UFOP
Instituto de Ciências Exatas e Aplicadas – ICEA
Departamento de Computação e Sistemas – DECSI

João Monlevade-MG
2024/2

Referencial Teórico

- O material aqui apresentado é baseado no Capítulo 4 do livro:
  - Silberschatz, A.; Galvin, P. B.; Gagne, G., Fundamentos de Sistemas Operacionais, Editora LTC, 9ª edição, 2015.



Introdução

- O modelo de processo apresentado nas aulas anteriores considera que o processo é um programa em execução com uma única *thread* de controle.
  - SOs modernos fornecem recursos para que um processo tenha diversas *threads* de controle.
  - ***Multithreading***.
- *Multithreading* pode ser suportada por um computador com uma única CPU?

Introdução

- O modelo de processo apresentado nas aulas anteriores considera que o processo é um programa em execução com uma única *thread* de controle.
  - SOs modernos fornecem recursos para que um processo tenha diversas *threads* de controle.
  - ***Multithreading***.
- *Multithreading* pode ser suportada por um computador com uma única CPU?
  - Sim!
  - *Threads* executam de forma concorrente!

Visão Geral

Visão Geral

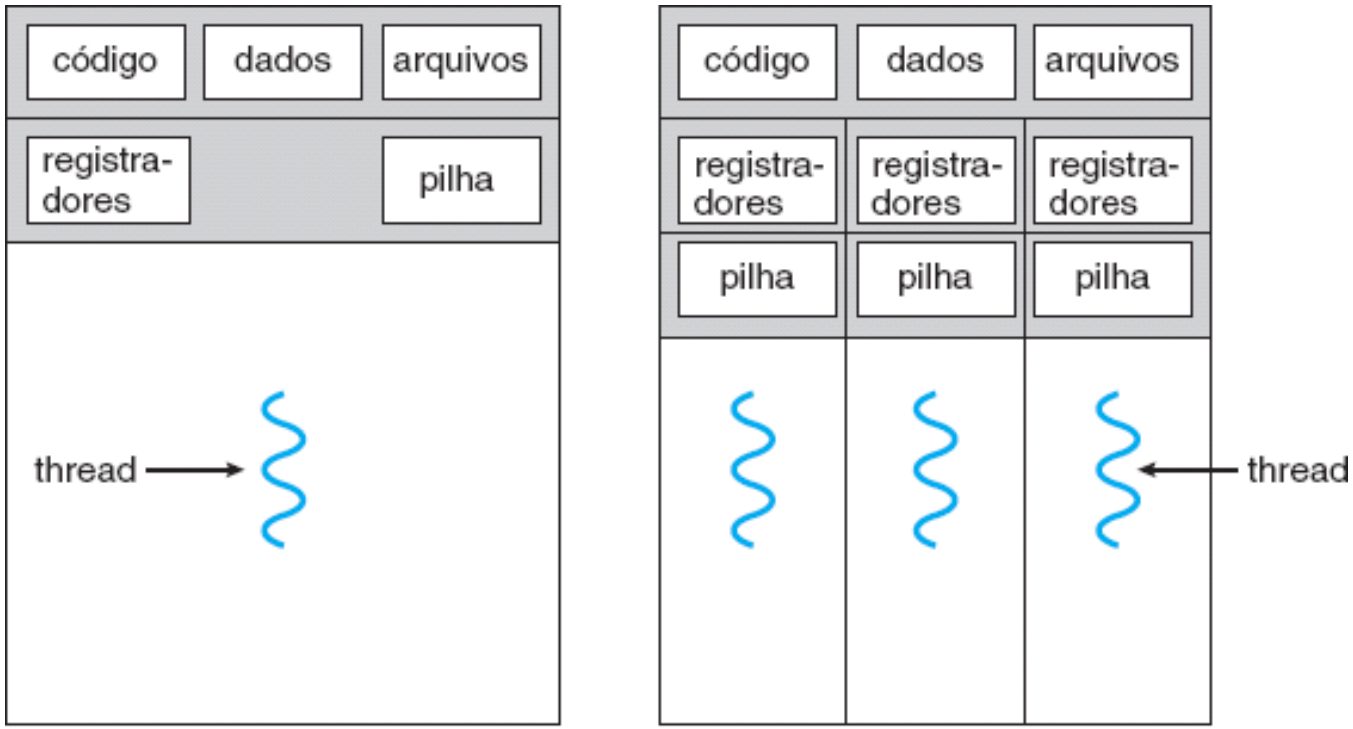
- **Processo**:
  - Programa em execução.
  - Espaço de endereçamento.
  - Recursos de sincronização e comunicação.
  - Recursos de mais alto nível.
- **Thread**:
  - Linha ou Encadeamento de execução.
  - Fluxo de controle de instruções.
  - Forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrentemente.
  - Compartilham recursos do processo.
  - Objetivo:
    - Maximizar o grau de execução concorrente.
    - Sobreposição de E/S e processamento.

Visão Geral

- *Thread* é uma unidade básica de utilização de CPU.
- É composta por:
  - ID de *thread*
  - Contador de programa (PC)
  - Conjunto de registradores
  - Pilha
- Compartilha com outras *threads* pertencentes ao mesmo processo:
  - Seção de código
  - Seção de dados
  - Outros recursos do SO (arquivos abertos, sinais, etc.)

Visão Geral

- Um processo tradicional (ou processo pesado - *heavyweight process*) possui uma única *thread* de controle.
- Múltiplas *threads* de controle permitem o processo realizar mais de uma tarefa ao mesmo tempo.



processo com um único thread

processo com múltiplos threads

Motivação

- Muitos pacotes de software executados nos computadores modernos são dotados de múltiplas *threads*.
  - Um processo, várias *threads* de controle.
- Exemplos:
  - Navegador Web:
    - Uma *thread* exibindo imagens ou texto.
    - Outra *thread* recebe dados da rede.
  - Processador de textos:
    - Uma *thread* para exibir gráficos.
    - Outra *thread* para ler os toques de tecla do usuário.
    - Uma terceira *thread* para verificação ortográfica e gramatical.
- Aplicações também podem ser projetadas para alavancar capacidades de processamento em sistemas *multicore*.
  - Podem executar diversas tarefas CPU-intensivas em paralelo, em múltiplos núcleos de computação.

Motivação

- Criação de processos é demorada e exige muitos recursos.
- Exemplo:
  - Servidor Web:
    - Vários clientes fazendo requisições concorrentemente.
    - Um processo tradicional atenderia uma única requisição por vez.
    - Solução inicial: criar um processo para atender cada requisição.

Motivação

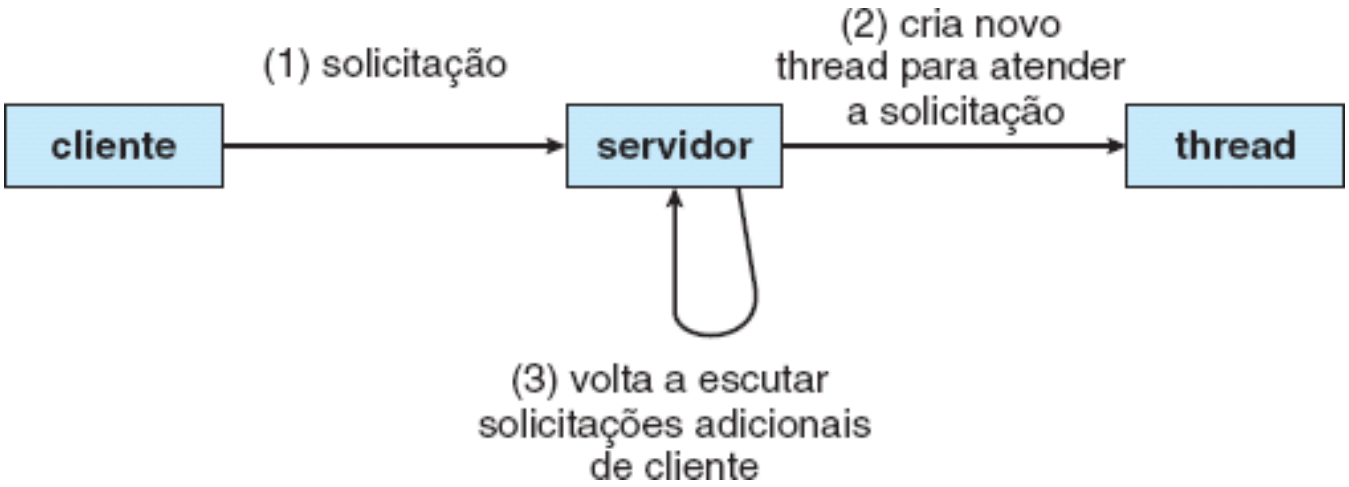
- Criação de processos é demorada e exige muitos recursos.
- Exemplo:
  - Servidor Web:
    - Vários clientes fazendo requisições concorrentemente.
    - Um processo tradicional atenderia uma única requisição por vez.
    - Solução inicial: criar um processo para atender cada requisição.
  - Se o novo processo tiver de realizar as mesmas tarefas do processo existente, por que incorrer em todo esse *overhead*?

Motivação

- Criação de processos é demorada e exige muitos recursos.
- Exemplo:
  - Servidor Web:
    - Vários clientes fazendo requisições concorrentemente.
    - Um processo tradicional atenderia uma única requisição por vez.
    - Solução inicial: criar um processo para atender cada requisição.
  - Se o novo processo tiver de realizar as mesmas tarefas do processo existente, por que incorrer em todo esse *overhead*?
    - Solução: utilização de múltiplas *threads*.

Motivação

Arquitetura de servidor com múltiplas *threads*:



Motivação

- A maioria dos kernels dos SOs modernos são *multithreaded*.
- Múltiplas *threads* operam no kernel e cada *thread* executa uma tarefa específica.
  - Gerenciamento de dispositivos, gerenciamento da memória, manipulação de interrupções, etc.
- Solaris tem um conjunto de *threads* no kernel especificamente para a manipulação de interrupções.
- Linux usa uma *thread* do kernel para gerenciar o montante de memória livre no sistema.

Benefícios

Benefícios

1. **Capacidade de resposta**:
   - Tornar uma aplicação interativa *multithreaded* pode permitir que um programa continue a ser executado, mesmo que parte dele esteja bloqueada ou executando uma operação demorada.
   - Aumenta a capacidade de resposta para o usuário.
   - Particularmente útil no projeto de interfaces de usuário.
     - Considere o que ocorre quando um usuário clica em um botão que causa a execução de uma operação demorada.
     - Uma aplicação com uma única *thread* deixaria de responder ao usuário até que a operação fosse concluída.
     - Por outro lado, se a operação demorada for executada em uma *thread* separada, a aplicação continuará respondendo ao usuário.

Benefícios

2. **Compartilhamento de recursos**:
   - Processos só podem compartilhar recursos por meio de técnicas como memória compartilhada e transmissão de mensagens.
   - *Threads* compartilham a memória e os recursos do processo ao qual pertencem.
     - Isso permite que uma aplicação tenha múltiplas *threads* de atividade diferentes dentro do mesmo espaço de endereçamento.

Benefícios

3. **Economia**:
   - A alocação de memória e recursos para a criação de processos é dispendiosa.
   - Já que as *threads* compartilham os recursos do processo ao qual pertencem, é mais econômico criar *threads* e trocar seus contextos.
   - Em geral, é significantemente mais demorado criar e gerenciar processos do que *threads*.
     - No Solaris: criação de um processo é cerca de trinta vezes mais lenta do que a criação de uma *thread*; troca de contexto é cerca de cinco vezes mais lenta.

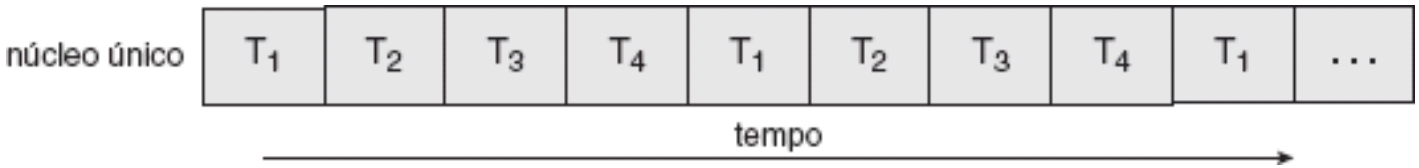
Benefícios

4. **Escalabilidade**:
   - Os benefícios da criação de múltiplas *threads* podem ser ainda maiores em uma arquitetura multiprocessada.
   - *Threads* podem ser executadas em paralelo em diferentes núcleos de processamento.
   - Um processo com uma única *thread* só pode ser executado em um processador, independentemente de quantos estiverem disponíveis.

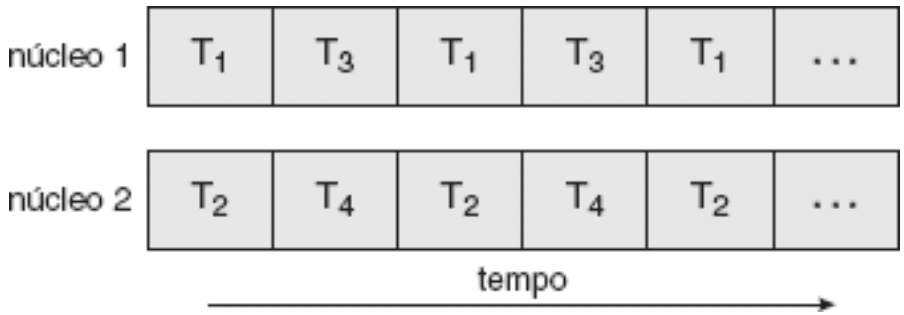
Programação Multicore

Programação Multicore

Execução concorrente em um sistema com um único núcleo:



Execução paralela em um sistema multicore:



Tipos de Paralelismo

Paralelismo de dados:

- Distribuição de subconjuntos dos dados em múltiplos núcleos de computação.
  - Mesma operação em cada núcleo.
- Exemplo: somar os conteúdos de um vetor de tamanho N.
  - Aplicação de *thread* única:
    - Somar todos os elementos de 0 a $N-1$
  - Aplicação com duas *threads*:
    - *Thread1* soma os elementos das posições 0 a $N/2-1$
    - *Thread2* soma os elementos das posições $N/2$ a $N-1$

Tipos de Paralelismo

Paralelismo de tarefas:

- Envolve a distribuição não de dados, mas de tarefas em vários núcleos de computação separados.
  - Cada *thread* executa uma única operação.
- Diferentes *threads* podem estar operando sobre os mesmos dados ou sobre dados diferentes.
- Exemplo:
  - Calcular a média aritmética e a mediana de um vetor de números inteiros de tamanho N.
    - Uma *thread* para calcular a média e outra para calcular a mediana.

Tipos de Paralelismo

- Paralelismo de dados envolve a distribuição de dados por vários núcleos.
- Paralelismo de tarefas envolve a distribuição de tarefas em vários núcleos.
- Em geral, as aplicações usam um híbrido dessas duas estratégias.

Modelos de Geração de *Multithreads*

Modelos de Geração de *Multithreads*

- O suporte a *threads* pode ser fornecido no nível de usuário ou kernel.
- **Threads de usuário** *vs* **threads de kernel**.
  - **Threads de usuário** são suportadas acima do kernel e gerenciadas sem o suporte do kernel, enquanto as **threads de kernel** são suportadas e gerenciadas diretamente pelo SO.
- Praticamente todos os SOs modernos dão suporte às *threads* de kernel.

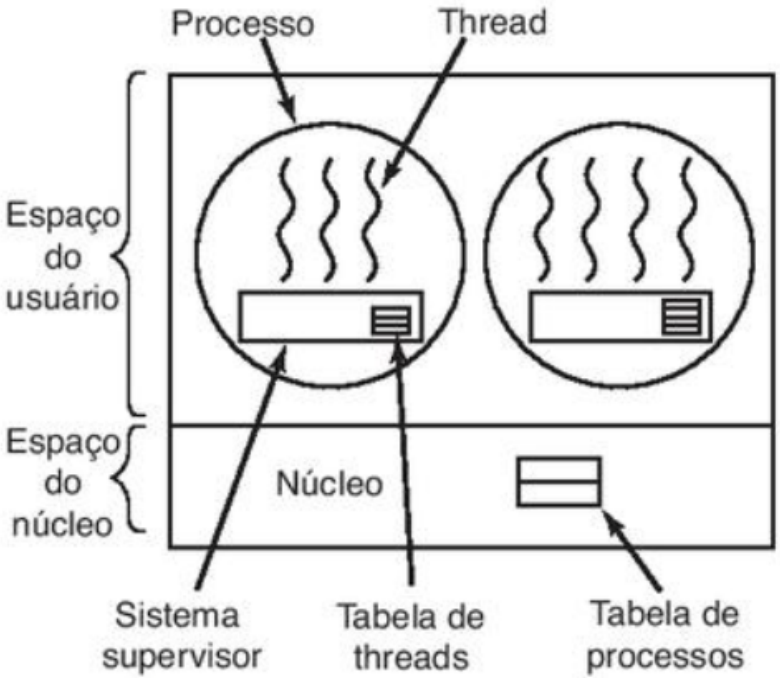
Threads em Modo Usuário

- Executam em modo usuário.
  - Implementadas pela aplicação e não pelo SO.
  - O núcleo do SO não interfere nas *threads*.
- Uma biblioteca é responsável por gerenciar essas *threads*.
  - Criar, remover, escalonar, etc.
- **Vantagens**:
  - Permitem a implementação de *threads* mesmo em SOs que não suportam *threads*.
  - Rápidas e eficientes.
- **Desvantagens**:
  - SO gerencia o processo como se houvesse uma única *thread*.
  - Tratamento individual de sinais.
  - Redução do grau de paralelismo.

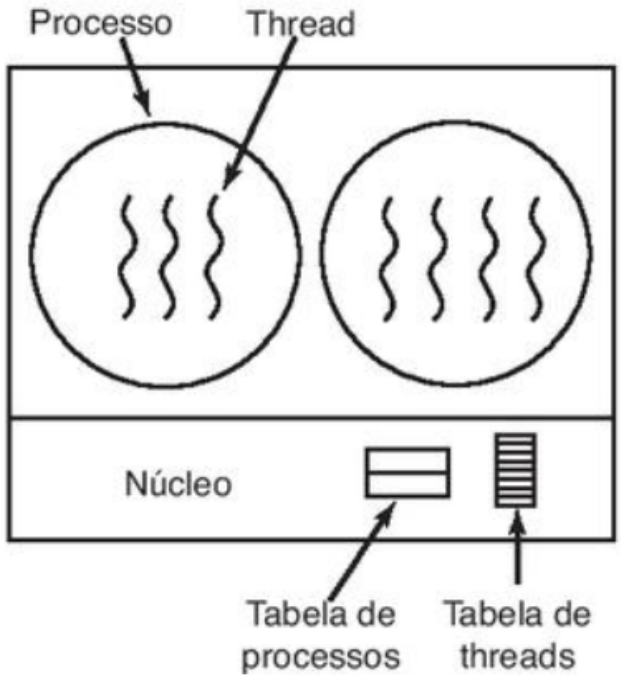
Threads em Modo Kernel

- São implementadas diretamente pelo SO.
- **Vantagens**:
  - SO sabe da existência de cada *thread* e pode escaloná-la individualmente.
  - *Threads* de um mesmo processo podem ser executadas simultaneamente.
- **Desvantagem**:
  - Muitas chamadas de sistema.
- SOs podem "reciclar" *threads* para evitar a criação e destruição em excesso.

Modelos de Geração de *Multithreads*



(a) Um pacote de threads no espaço do usuário.



(b) Um pacote de threads gerenciado pelo núcleo.

Modo Híbrido

- Combina as vantagens das *threads* de usuário e de kernel.
- Um processo pode ter várias *threads* de kernel.
  - Cada *thread* de kernel pode ter várias *threads* de usuário.
- Necessidade de realizar mapeamento.
- **Vantagem**:
  - Maior flexibilidade.
- **Desvantagens**:
  - Problemas herdados de ambas implementações.
  - *Threads* em modo usuário que precisam executar em diferentes processadores precisam utilizar diferentes *threads* em modo de kernel.

Modo Híbrido

- Deve existir um relacionamento entre as *threads* de usuário e as *threads* de kernel.
  - Examinaremos três maneiras comuns de estabelecer esse relacionamento.

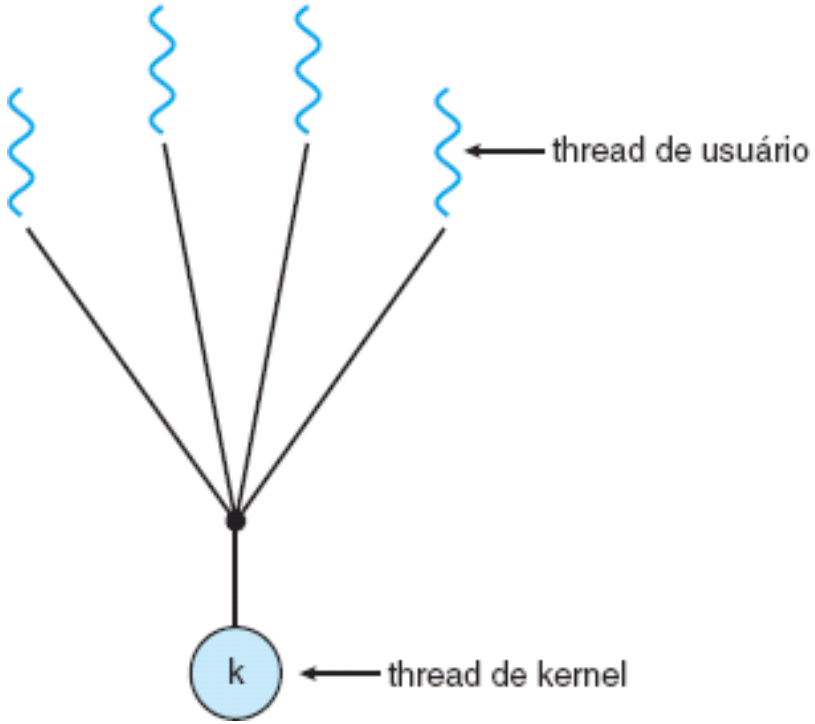
Modelo Muitos-Para-Um

- Mapeia **muitas threads de usuário** para **uma thread de kernel**.
- O gerenciamento das *threads* é feito pela biblioteca de *threads* no espaço do usuário.
- O processo inteiro será bloqueado se uma *thread* fizer uma chamada de sistema bloqueadora.
- Exemplo:
  - **Green threads**: uma biblioteca de *threads* disponível para sistemas Solaris e adotada em versões iniciais de Java.
- **Desvantagem**:

Modelo Muitos-Para-Um

- Mapeia **muitas threads de usuário** para **uma thread de kernel**.
- O gerenciamento das *threads* é feito pela biblioteca de *threads* no espaço do usuário.
- O processo inteiro será bloqueado se uma *thread* fizer uma chamada de sistema bloqueadora.
- Exemplo:
  - **Green threads**: uma biblioteca de *threads* disponível para sistemas Solaris e adotada em versões iniciais de Java.
- **Desvantagem**:
  - Já que apenas uma *thread* por vez pode acessar o kernel, muitas *threads* ficam incapazes de executar em paralelo em sistemas *multicore*.

Modelo Muitos-Para-Um



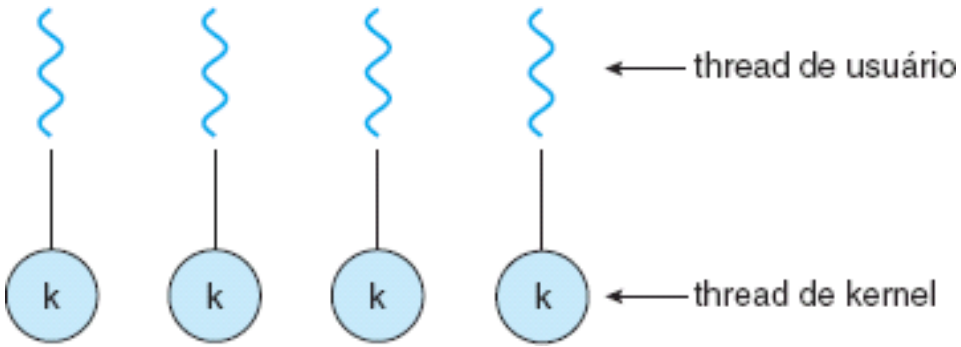
Modelo Um-Para-Um

- Mapeia **cada thread de usuário** para **uma thread de kernel**.
- Fornece mais concorrência do que o modelo muitos-para-um.
  - Permite que outra *thread* seja executada quando uma *thread* faz uma chamada de sistema bloqueadora.
- Permite que múltiplas *threads* sejam executadas em paralelo em multiprocessadores.
- **Desvantagem**:

Modelo Um-Para-Um

- Mapeia **cada thread de usuário** para **uma thread de kernel**.
- Fornece mais concorrência do que o modelo muitos-para-um.
  - Permite que outra *thread* seja executada quando uma *thread* faz uma chamada de sistema bloqueadora.
- Permite que múltiplas *threads* sejam executadas em paralelo em multiprocessadores.
- **Desvantagem**:
  - A criação de uma *thread* de usuário requer a criação da *thread* de kernel correspondente.
  - *Overhead* de criação de *threads* de kernel pode sobrecarregar o desempenho de uma aplicação.
  - A maioria das implementações desse modelo restringe o número de *threads* suportadas pelo sistema.
- Linux e Windows implementam esse modelo.

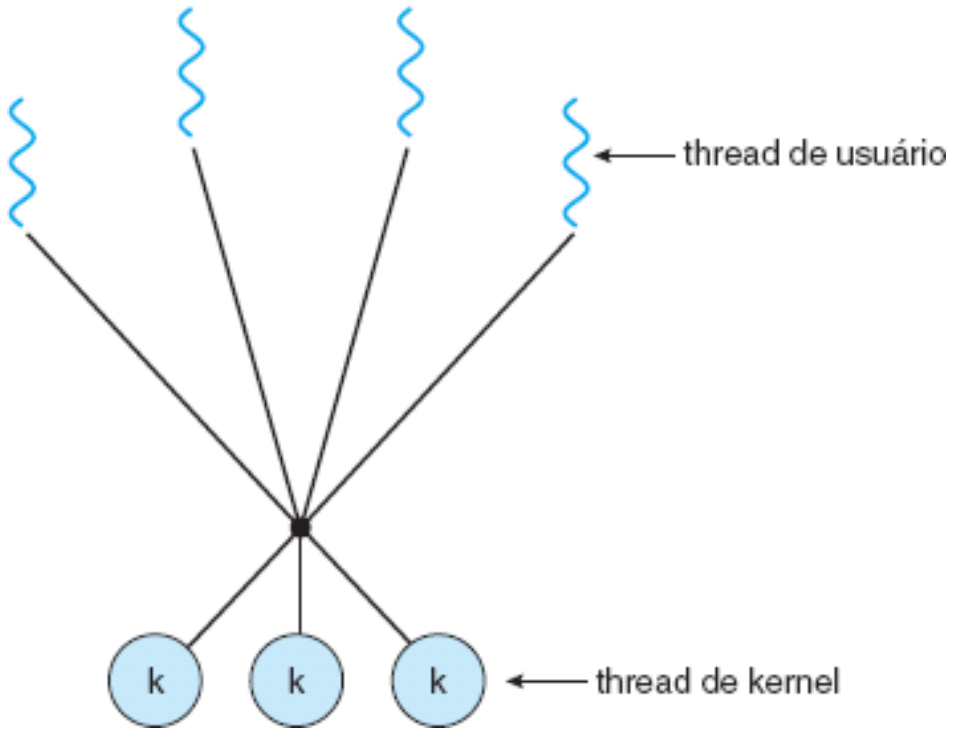
Modelo Um-Para-Um



Modelo Muitos-Para-Muitos

- Multiplexa **muitas threads de nível de usuário** para um **número menor ou igual de threads de kernel**.
- O número de *threads* de kernel pode ser específico para determinada aplicação ou máquina.
- Desenvolvedores podem criar quantas *threads* de usuário forem necessárias e as *threads* de kernel correspondentes podem ser executadas em paralelo em um multiprocessador.
- Quando uma *thread* executa uma chamada de sistema bloqueadora, o kernel pode designar outra *thread* para execução.

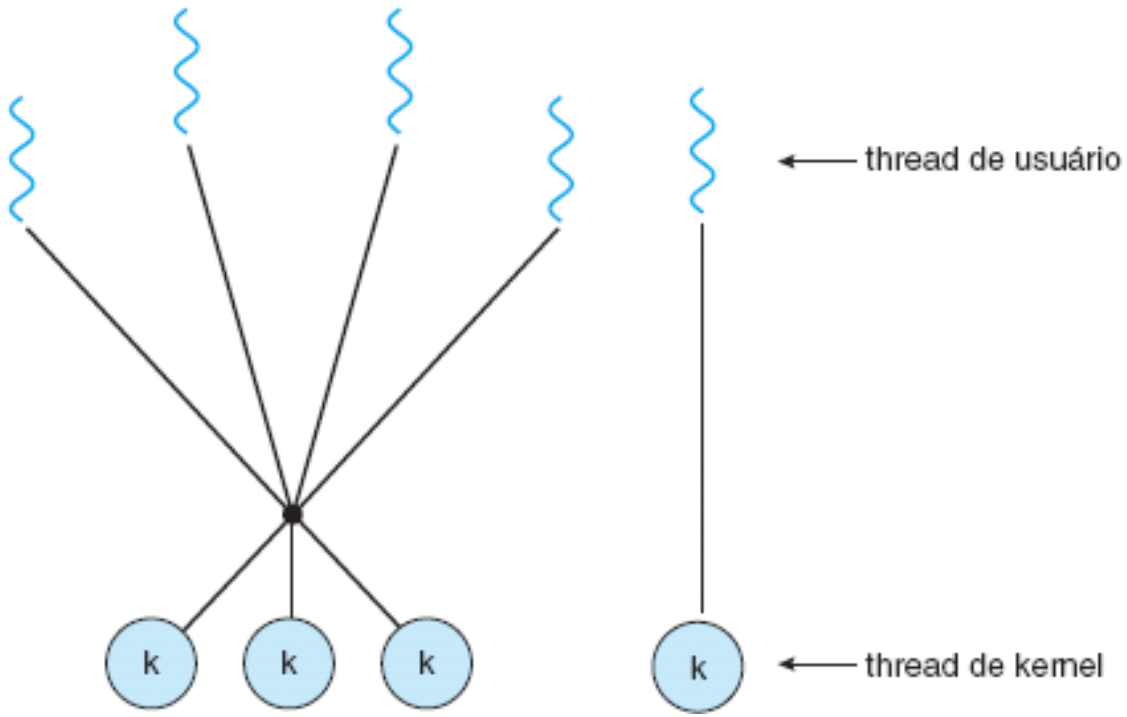
Modelo Muitos-Para-Muitos



Modelo de Dois Níveis

- Variação do modelo muitos-para-muitos.
- Multiplexa **muitas threads de nível de usuário** para um **número menor ou igual de threads de kernel**, mas também permite que **uma thread de nível de usuário** seja limitada a **uma thread de kernel**.
- Sistema operacional Solaris suportava o modelo de dois níveis em versões anteriores ao Solaris 9.
  - A partir do Solaris 9, esse sistema passou a usar o modelo um-para-um.

Modelo de Dois Níveis



Bibliotecas de *Threads*

Bibliotecas de *Threads*

- Uma **biblioteca de threads** fornece ao programador uma API para criação e gerenciamento de *threads*.
- Duas maneiras principais de implementar:
  1. Inteiramente no espaço do usuário, sem suporte do kernel.
     - Código e estruturas de dados da biblioteca existem no espaço do usuário.
     - A invocação de uma função da biblioteca resulta em uma chamada de função local no espaço do usuário e não em uma chamada de sistema.
  2. No nível do kernel com suporte direto do SO.
     - Código e estruturas de dados da biblioteca existem no espaço do kernel.
     - Invocar uma função da biblioteca resulta em uma chamada de sistema para o kernel.

Bibliotecas de *Threads*

Exemplos:

1. **Pthreads**:
   - Extensão de *threads* do padrão POSIX.
   - Pode ser fornecida como uma biblioteca de nível de usuário ou de kernel.
2. **Threads do Windows**:
   - Biblioteca de nível de kernel disponível em sistemas Windows.
3. **Threads do Java**:
   - Permite que *threads* sejam criadas e gerenciadas diretamente por programas em Java.
   - Implementada com o uso de uma biblioteca de *threads* disponível no sistema hospedeiro.
     - Em sistemas Windows, são implementadas com o uso da API Windows.
     - Em sistemas UNIX e Linux, costumam usar a biblioteca Pthreads.

Questões Relacionadas com a Criação de *Threads*

Manipulação de Sinais

- Um sinal é usado em sistemas UNIX para notificar um processo de que determinado evento ocorreu.
- Todos os sinais seguem o mesmo padrão:
  1. Um sinal é gerado pela ocorrência de um evento específico.
  2. O sinal é liberado para um processo.
  3. Uma vez liberado, o sinal deve ser manipulado.
- Opções de manipulação de sinais:
  - Liberar o sinal para a *thread* ao qual ele é aplicável.
  - Liberar o sinal para cada *thread* do processo.
  - Liberar o sinal para certas *threads* do processo.
  - Designar uma *thread* específica para receber todos os sinais do processo.

Cancelamento de *Threads*

- O **cancelamento de threads** envolve o encerramento de uma *thread* antes que ela seja concluída.
- Exemplos:
  - Se múltiplas *threads* estiverem pesquisando concorrentemente em um banco de dados e uma delas retornar o resultado, as *threads* restantes podem ser canceladas.
  - Usuário pressiona um botão em um navegador web que impede que uma página web continue a ser carregada.
- Uma *thread* que está para ser cancelada costuma ser chamada de **thread-alvo**.

Cancelamento de *Threads*

- O cancelamento de uma *thread-alvo* pode ocorrer em dois cenários diferentes:
  - **Cancelamento assíncrono**: uma *thread* encerra imediatamente a *thread-alvo*.
  - **Cancelamento adiado**: a *thread-alvo* verifica, periodicamente, se deve ser encerrado, dando a si próprio a oportunidade de terminar de maneira ordenada.
- Problemas do cancelamento assíncrono?

Cancelamento de *Threads*

- O cancelamento de uma *thread-alvo* pode ocorrer em dois cenários diferentes:
  - **Cancelamento assíncrono**: uma *thread* encerra imediatamente a *thread-alvo*.
  - **Cancelamento adiado**: a *thread-alvo* verifica, periodicamente, se deve ser encerrado, dando a si próprio a oportunidade de terminar de maneira ordenada.
- Problemas do cancelamento assíncrono?
  - Uma *thread* pode não liberar recursos utilizados por ela.

Pools de *Threads*

- Bancos de *threads*.
- Criar uma série de *threads* e colocá-las em um banco, no qual fiquem esperando para executar.
  - Quantidade de *threads* pode ser definida heuristicamente.
- **Vantagens:**
  - Geralmente é um pouco mais rápido requisitar uma *thread* já existente do que criar uma nova *thread*.
  - Permite que o número de *threads* na aplicação seja limitado ao tamanho do banco.

Leitura Complementar

- Capítulo 4 do livro "Fundamentos de Sistemas Operacionais" (Editora LTC, 9ª edição, 2015):
  - Seções **4.4.1** à **4.4.3** (bibliotecas de *threads*)
  - Seção **4.5** (*threading* implícito)
  - Seção **4.7** (*threads* em sistemas Windows e Linux)

Dúvidas?