
# Sistemas Operacionais

## Aula 8 – Threads – Parte 01

Professor: Wellington Franco

# Introdução

> *Em sistemas operacionais tradicionais, cada processo tem um espaço de endereçamento único e um único thread de controle. Contudo, frequentemente há situações em que é desejável possuir múltiplos threads de controle no mesmo espaço de endereçamento executando quase-paralelo, como se fossem processos separados.*
>
> **TANEMBAUM (2010)**

# Recapitulando

- Um processo é um programa em execução **com uma única thread de controle**

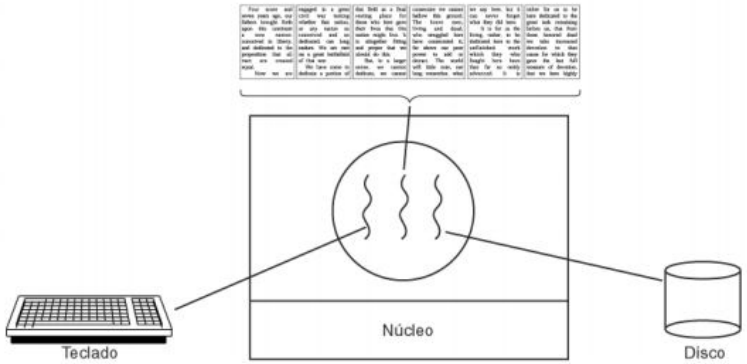
- Nos sistema operacionais modernos existe o conceito de multithread

# Um exemplo: programa editor de texto

- Imagine que você está escrevendo um livro
- A tela deve mostrar em tempo real o que você digita
- O processo de formatação do texto pode ser demorado:
- Pode ser necessário ajustar um grande número de páginas à medida que correções pontuais são efetuadas
- Isso pode comprometer a interação com o usuário
- Como resolver esse problema? Utilizando threads!
- OBS: Processos possuem um elevado custo computacional

# Um exemplo: programa editor de texto

- Um só processo com duas *threads*:
  - Uma está sempre disponível para interagir com o usuário
  - Uma responsável somente para atualizar o texto na tela
- Possível adicionar uma terceira para salvamento automático



# Definição de threads

> ***Uma thread é uma unidade básica de uso de CPU**. Cada thread é composta por um identificador, um contador de programa, um conjunto de registradores e uma pilha. Eles compartilham com as demais threads o código-fonte, os dados e outros recursos de alto nível atribuídos ao processo dono das threads em questão.*
>
> **SILBERSCHARTZ, P. B. GALVIN, G. GAGNE (2009)**

# Visão geral

- Uma thread é uma unidade básica de utilização da CPU;

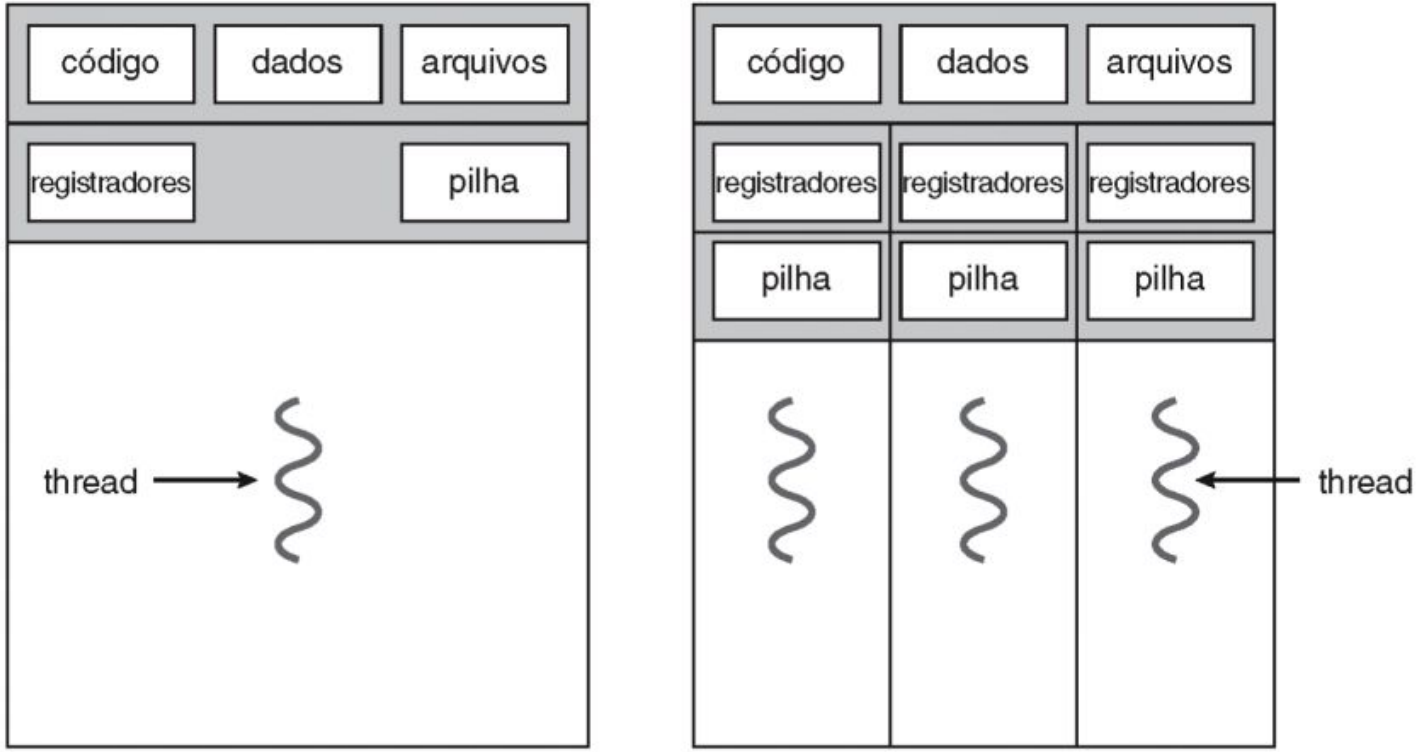
- É composto por
  - ID de Thread,
  - Contador de programas,
  - Conjunto de registradores
  - Pilha

# Visão geral

- Compartilha com outra threads do mesmo processo
  - Seção de código
  - Seção de dados
  - Recursos do SO

- Um processo tradicional possui uma única thread de controle
- Um processo com vários threads de controle, poderá executar mais de uma tarefa ao mesmo tempo.

# Visão geral


código
dados
arquivos
registradores
pilha
thread
código
dados
arquivos
registradores
registradores
registradores
pilha
pilha
pilha
thread

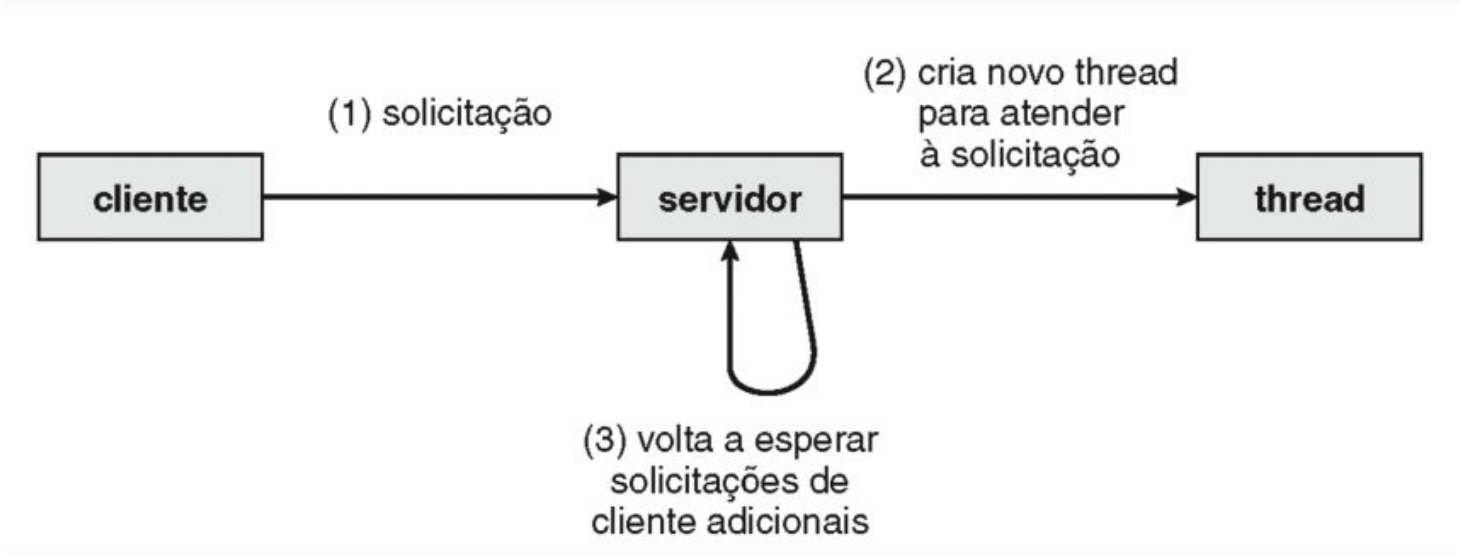
# Motivação

- Um navegador web pode ter uma thread para exibir imagens e outra thread para recuperar dados da rede

- Um processador de texto pode ter um thread para exibir os elementos gráficos, outro para responder ao teclado e outra para verificar a ortográfia.

- Servidor web atendendo a múltiplos clientes

# Motivação


(1) solicitação
(2) cria novo thread
para atender
à solicitação
cliente
servidor
thread
(3) volta a esperar
solicitações de
cliente adicionais

# Threads: Informações gerais

- *Threads* são chamadas de **processos leves** ou **miniprocessos**
- Processos são utilizados para agrupar recursos, *threads* são as entidades escalonadas para a execução sobre a CPU
- Permitem múltiplas execuções paralelas de tarefas:
  - Desde que elas tenham um certo grau de independência
- A grande (des)vantagem é o **compartilhamento de recursos**:
  - Simplifica o trabalho cooperativo na solução do problema
  - Porém, **não existe proteção contra operações indevidas**

# Threads: Informações gerais

- Uma *thread* pode ler, escrever ou até mesmo apagar os itens do processo e comprometer o funcionamento das outras
- O que realmente é compartilhado e exclusivo por *thread*?

| Itens por processo | Itens por thread |
|---|---|
| Espaço de endereçamento | Contador de programa |
| Variáveis globais | Registradores |
| Arquivos abertos | Pilha |
| Processos filhos | Estado |
| Alarmes pendentes | |
| Sinais e manipuladores de sinais | |
| Informação de contabilidade | |

# Benefícios

- **Capacidade de resposta:** permite que uma aplicação interativa continue a executar outras partes, mesmo se bloqueada ou com uma operação demorada

- **Compartilhamento de recursos:** os processos só podem compartilhar recursos pela memória compartilhada ou transmissão de mensagem. As threads compartilham memória e recursos do processo. Assim, várias threads em uma mesma aplicação compartilham o mesmo espaço de endereçamento

# Benefícios

- **Economia:** a alocação de memória e recursos para a criação de processos é dispendiosa. É mais econômico criar e alternar entre threads.

- **Escalabilidade:** a utilização de múltiplas threads em multiprocessadores

# Programação multicore

- **Multicore: muitos núcleos no mesmo chip**

- A programação com várias threads fornece um mecanismo para o uso mais eficiente de muitos núcleos e o aumento da concorrência.

- Concorrência: a execução de cada thread de cada vez / intercalando threads

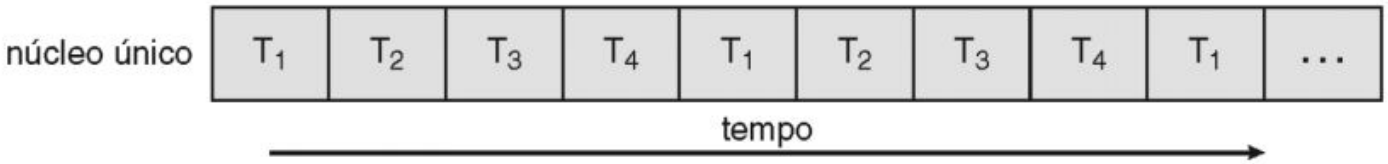
# Processos X Threads

- Tempo de criação/destruição de *threads* é de 10 a 20 vezes menor do que o tempo de criação/destruição de um processo
- Tempo de troca de contexto entre *threads* é de 5 a 50 vezes menor do que o tempo de troca de contexto entre processos
- *Threads* compartilham recursos do processo que as criaram

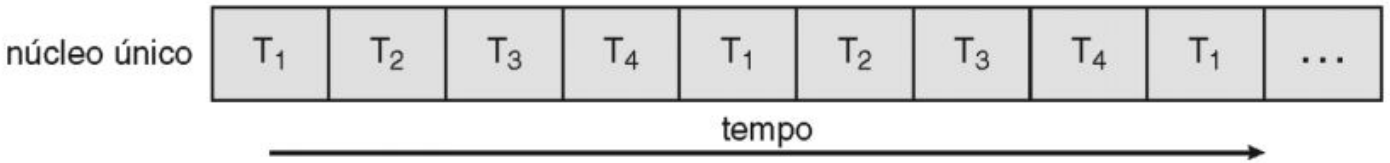
| Platform | fork() | | | pthread_create() | | |
|---|---|---|---|---|---|---|
| | real | user | sys | real | user | sys |
| AMD 2.4 GHz Opteron (8cpus/node) | 41.07 | 60.08 | 9.01 | 0.66 | 0.19 | 0.43 |
| IBM 1.9 GHz POWER5 p5-575 (8cpus/node) | 64.24 | 30.78 | 27.68 | 1.75 | 0.69 | 1.10 |
| IBM 1.5 GHz POWER4 (8cpus/node) | 104.05 | 48.64 | 47.21 | 2.01 | 1.00 | 1.52 |
| INTEL 2.4 GHz Xeon (2 cpus/node) | 54.95 | 1.54 | 20.78 | 1.64 | 0.67 | 0.90 |
| INTEL 1.4 GHz Itanium2 (4 cpus/node) | 54.54 | 1.07 | 22.22 | 2.03 | 1.26 | 0.67 |

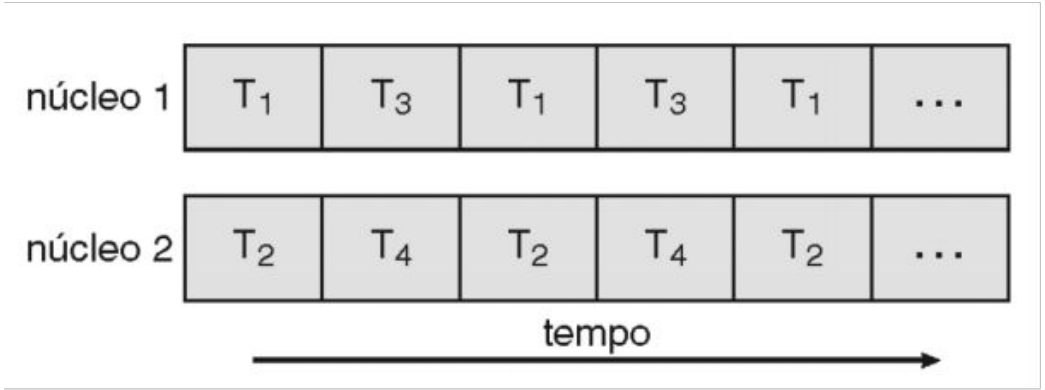
Fonte: http://www.llnl.gov/computing/tutorials/pthreads (tempo em segundos para criação 50000 processos /threads)

# Programação multicore



# Programação multicore



# Programação multicore

- A tendência ao uso de sistemas multicore tem pressionado os projetistas de sistemas a fazer o melhor uso dos diversos núcleos de computação

- Áreas de desafios:
  - **Divisão de atividades**: separar a aplicação em tarefas concorrentes para execução em paralelo.

  - **Equilíbrio:** separação de maneira equilibrada, de tal forma que os núcleos executem tarefas de mesmo esforço

# Programação multicore

- Áreas de desafios:
  - **Divisão de dados:** separar os dados para acesso e manipulação
  - **Dependência de dados:** sincronia entre as tarefas separadas e os dados separados para cada uma delas
  - **Teste e depuração:** diferente caminhos para diferentes threads

# Modelos de geração de multithreads

- Modo dual:
  - **Nível usuário: threads de usuário**
    - São suportados acima do kernel e sem gerenciamento do kernel
  - **Nível kernel: threads de kernel**
    - São suportados e gerenciados pelo kernel

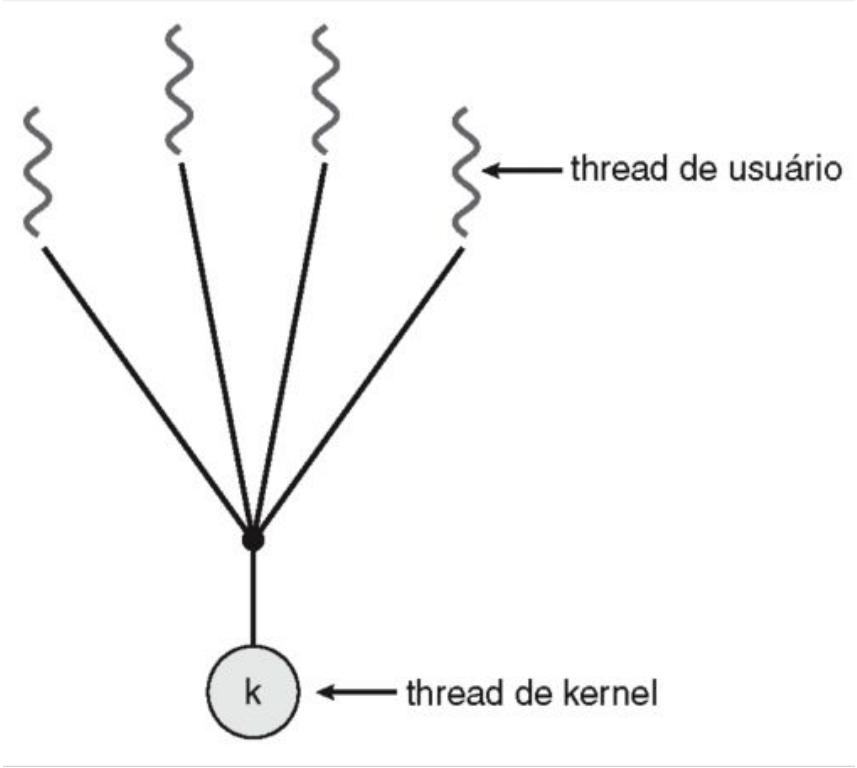
# Modelos de geração de multithreads

- Relacionamento entre elas:
  - **Modelo Muitos-para-um**
  - **Modelo Um-para-um**
  - **Modelo Muitos-para-muitos**

# Modelo muitos-para-um

- Mapeia muitas threads do usuário para uma do kernel
- Gerenciamento é feito pela biblioteca de threads no espaço do usuário, sendo eficiente;
- O processo é bloqueado se uma thread fizer um system call bloqueadora
- Apenas um thread pode acessar o kernel por vez, assim não ocorre paralelismo.

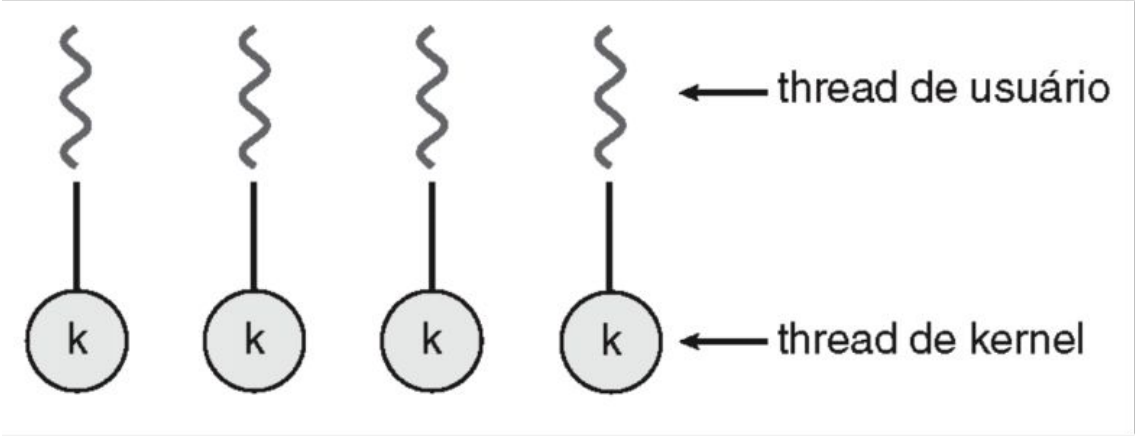
# Modelo muitos-para-um



# Modelo um-para-um

- Mapeia cada thread do usuário para um do kernel
- Fornece mais concorrência, uma vez que cada thread usuário interage com uma de kernel, assim pode ter system call bloqueadora.
- Permite que várias threads sejam executadas em paralelo.
- **Desvantagem:** cada thread de usuário necessita da criação de uma thread de kernel, assim é custoso. Logo, é restrito a quantidade de threads suportadas.

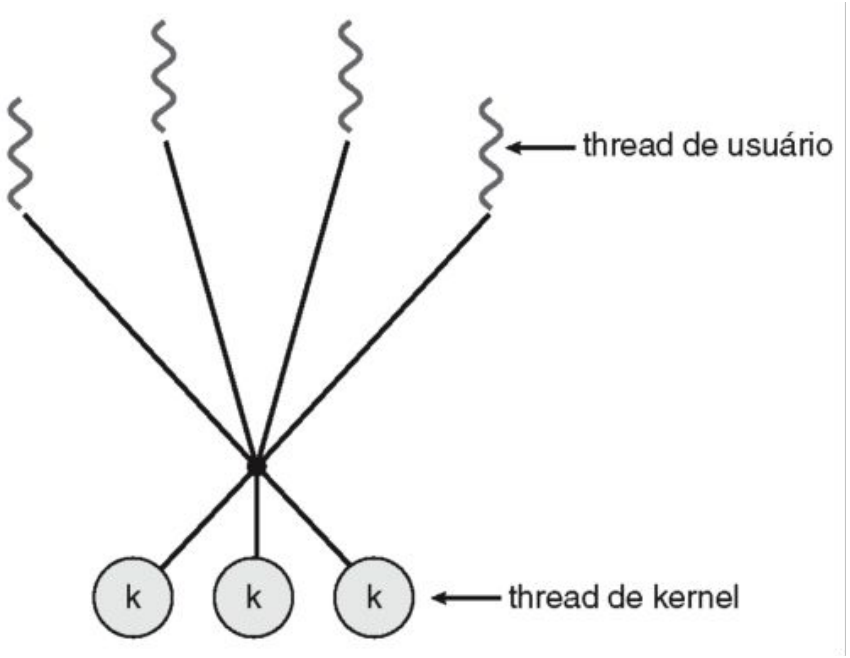
# Modelo um-para-um



# Modelo muitos-para-muitos

- Conecta muitas threads de usuário com uma quantidade menor ou igual das de kernel, conforme a necessidade
- Não sofre de ação bloqueante de uma thread que necessita de system call bloqueadora
- Não possui o problema do não paralelismo
- Cria quantas threads forem necessárias
- **Híbrido**

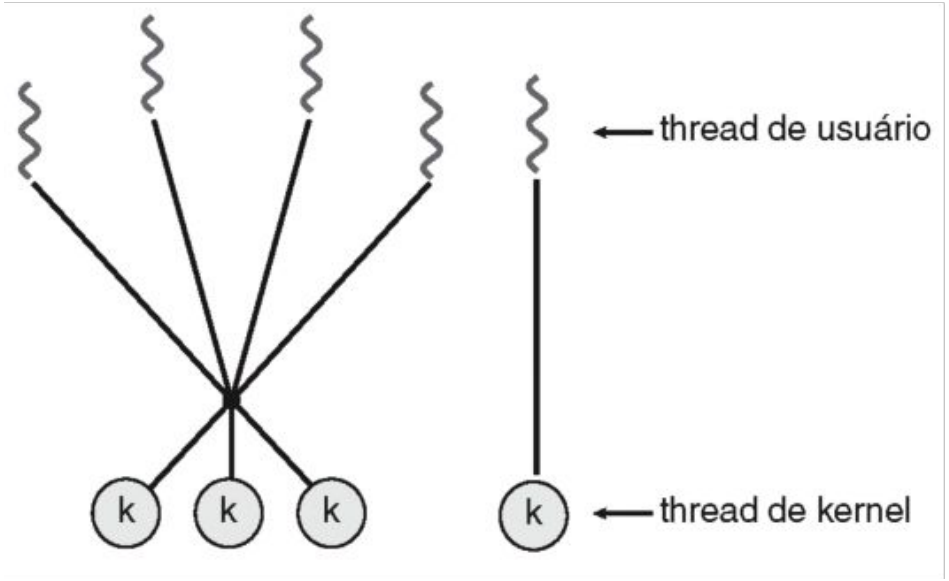
# Modelo muitos-para-muitos



# Modelo Two-Level

> *Uma variação popular do modelo Many-to-Many é responsável por multiplexar muitas threads de usuário em uma quantidade menor de threads de núcleo, mas também permite uma thread de usuário para cada thread de núcleo. Tal variação é chamada de* ***modelo two-level****. É suportado por SOs IRIX, HP-UX e Solaris mais antigos.*
>
> **SILBERSCHARTZ, P. B. GALVIN, G. GAGNE (2009)**

# Modelo de dois níveis



# Bibliotecas de threads

- Fornece ao programador uma API para a criação e gerenciamento de threads

- Duas abordagens:
  - Biblioteca inteiramente no espaço do usuário sem suporte do kernel
    - Não existe system call

  - Biblioteca localizada no nível de kernel com suporte direto ao SO
    - Chamadas na API geralmente são associadas a system call

# Pthreads

- Pode ser fornecida tanto no nível do usuário quanto no núcleo
- API padrão POSIX (IEEE 1003.1c) para criação e sincronização de threads
- A API especifica unicamente o comportamento da biblioteca, a implementação fica a cargo dos desenvolvedores
- Comum em sistemas operacionais UNIX (Solaris, Linux, Mac OS X)

# Threads em java

- Threads em Java são gerenciadas pela JVM

- Tipicamente implementados de acordo com o modelo de threads do sistema operacional “hospedeiro”

- Threads em Java podem ser criadas
  - Estendendo a classe **Thread** e sobrepor o método **run()**

  - Implementando a interface **Runnable**

# Questões relacionadas à criação de threads

- **As chamadas de sistema fork() e exec()**
- **Cancelamento**
- **Manipulação de sinais**
- **Pools de threads**
- **Dados específicos de threads**
- **Ativações de Scheduler**

# As chamadas de sistema fork() e exec()

- O **fork**() duplica apenas a thread que invocou a chamada ou todos as threads do programa?

- O que acontece com o **exec**() ?

# Cancelamento

- Encerrar uma thread antes de ser concluída

- Por exemplo, uma interrupção no carregamento de uma página (várias imagens, textos, ...)

- Um thread que está para ser cancelado costuma ser chamado de **thread-alvo.**

# Cancelamento

- Duas abordagens
  - **Cancelamento assíncrono**: uma thread encerra imediatamente
  - **Cancelamento adiado**: permite que a thread-alvo verifique periodicamente se deve ser cancelada, assim, permite que ela possa ser encerrada de maneira ordenada

# Manipulação de sinais

- Sinais são utilizados no UNIX para notificar um processo de que um determinado evento ocorreu

- Todos os sinais seguem o mesmo padrão
  - São gerados por um evento particular
  - São entregues a um processo
  - São manipulados pelo processo

- Exemplo: Acesso ilegal à memória, divisão por zero

# Manipulação de sinais

- **Opções:**
  - Entregar o sinal para a thread a qual ele se aplica
  - Entregar o sinal para todas as threads do processo
  - Entregar o sinal para certas threads do processo
  - Escolher uma thread específica para receber todos os sinais do processo

# Pools de threads

- Cria um número de threads em um pool onde eles aguardam por trabalho
- Evitar a criação infinita de threads (pode acabar os recursos da máquina)
- Uma vez que a thread finalize o trabalho, volta para o pool. Não é finalizada.

# Pools de threads

- **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(s) aplicação(ões) seja limitado ao tamanho do pool

# Dados específicos de threads

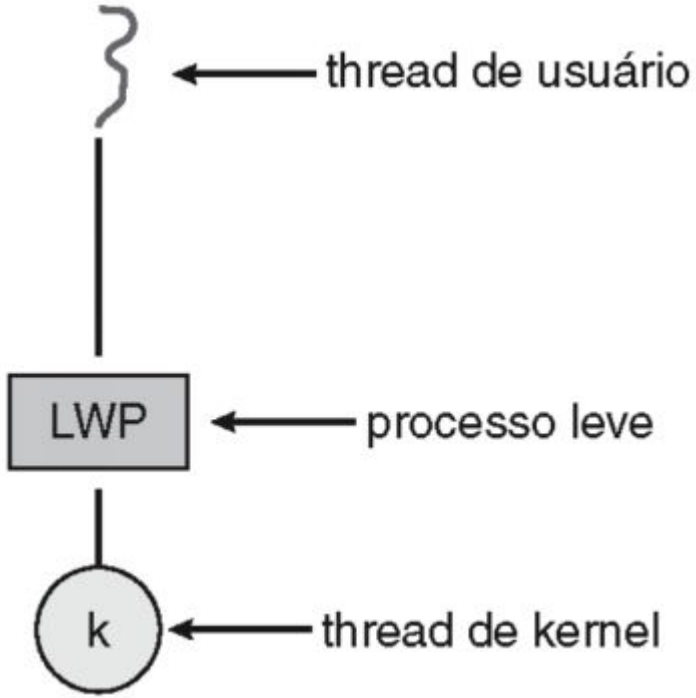
- Permite que cada thread tenha sua própria cópia dos dados

- Útil quando não se tem controle sobre o processo de criação de threads (ex., quando se usa um thread pool)

# Ativações de Scheduler

- Tanto o modelo muitos-para-muitos quanto o de dois níveis utilizam uma estrutura de dados intermediária (LWP) entre as threads do núcleo e do usuário

- O LWP funciona como um processador virtual para a aplicação no qual a aplicação pode agendar a execução de uma thread



# Ativações de Scheduler

- O núcleo, em seguida, atribui uma de suas threads para executar a thread do usuário associada a um LWP

- Para esse esquema funciona, o núcleo precisa notificar a aplicação sobre certos eventos de escalonamento de threads

- Essa notificação é feita através de um procedimento chamado **upcall** e a esse esquema de comunicação é conhecido com ativação do escalonador

# Dúvidas??

E-mail: wellington@crateus.ufc.br