

SISTEMAS OPERACIONAIS

Threads

Definição de thread

- Thread

- É às vezes chamado de processo leve (LWP).

Porque compartilham muitos dos recursos dos Processos aos quais estão associados. Processos tradicionais, portanto, podem ser denominados de processos pesados.

- Existem threads de instrução ou threads de controle.

- Os threads compartilham espaço de endereço e outras informações globais com seu próprio processo.

- Registradores, pilha, máscaras de sinal e outros dados específicos de thread são nativos a cada thread.

- Os threads devem ser gerenciados pelo sistema operacional ou pela aplicação de usuário.

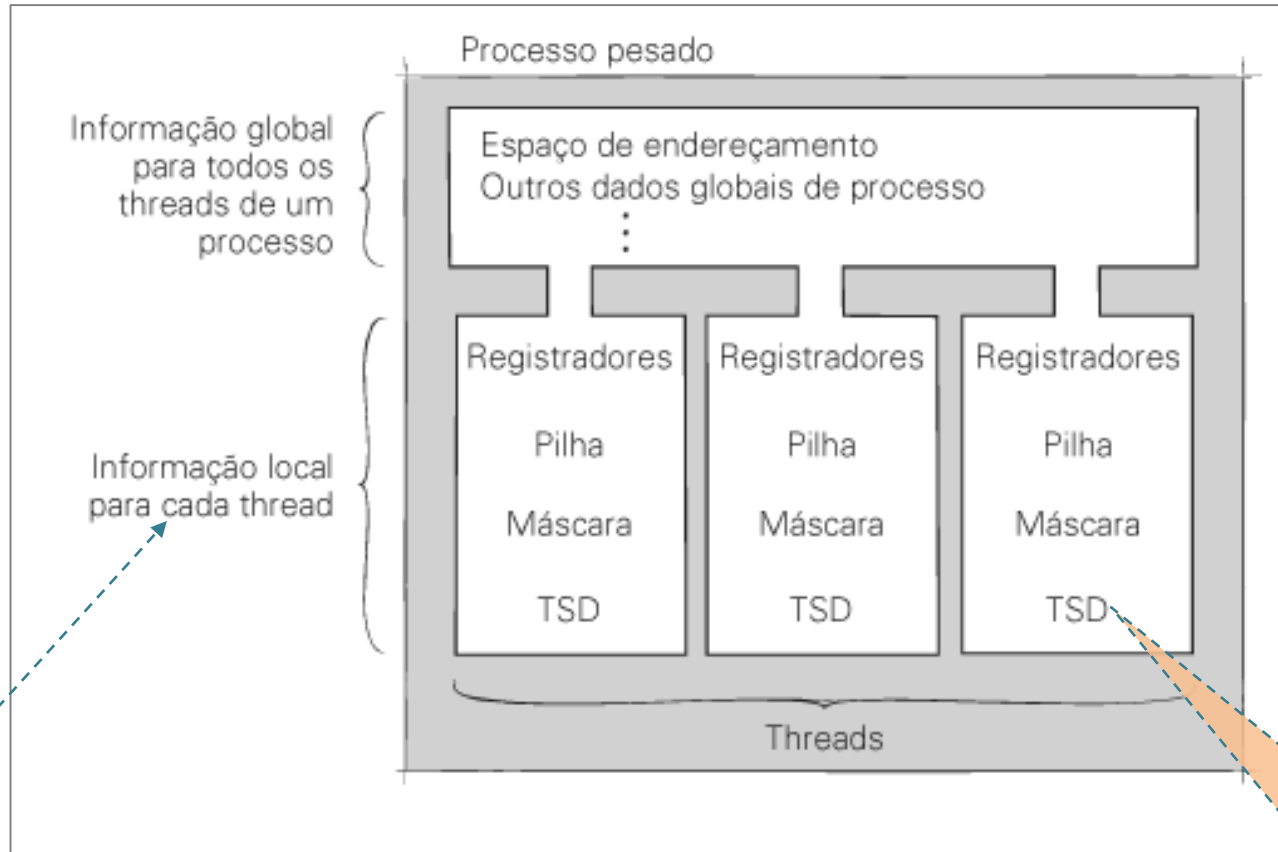
- Muitos S.O. suportam threads, mas as implementações variam consideravelmente:

- Exemplos: threads Win32, C-threads (Macintosh), Pthreads (Linux, Win XP).

Bibliotecas de suporte a THREADS com APIs diferentes.

Definição de thread

Relação entre thread e processo:



Sub-conjunto de recursos contidos em um processo que a THREAD utiliza.

Thread-Specific Data

Motivação na criação de threads

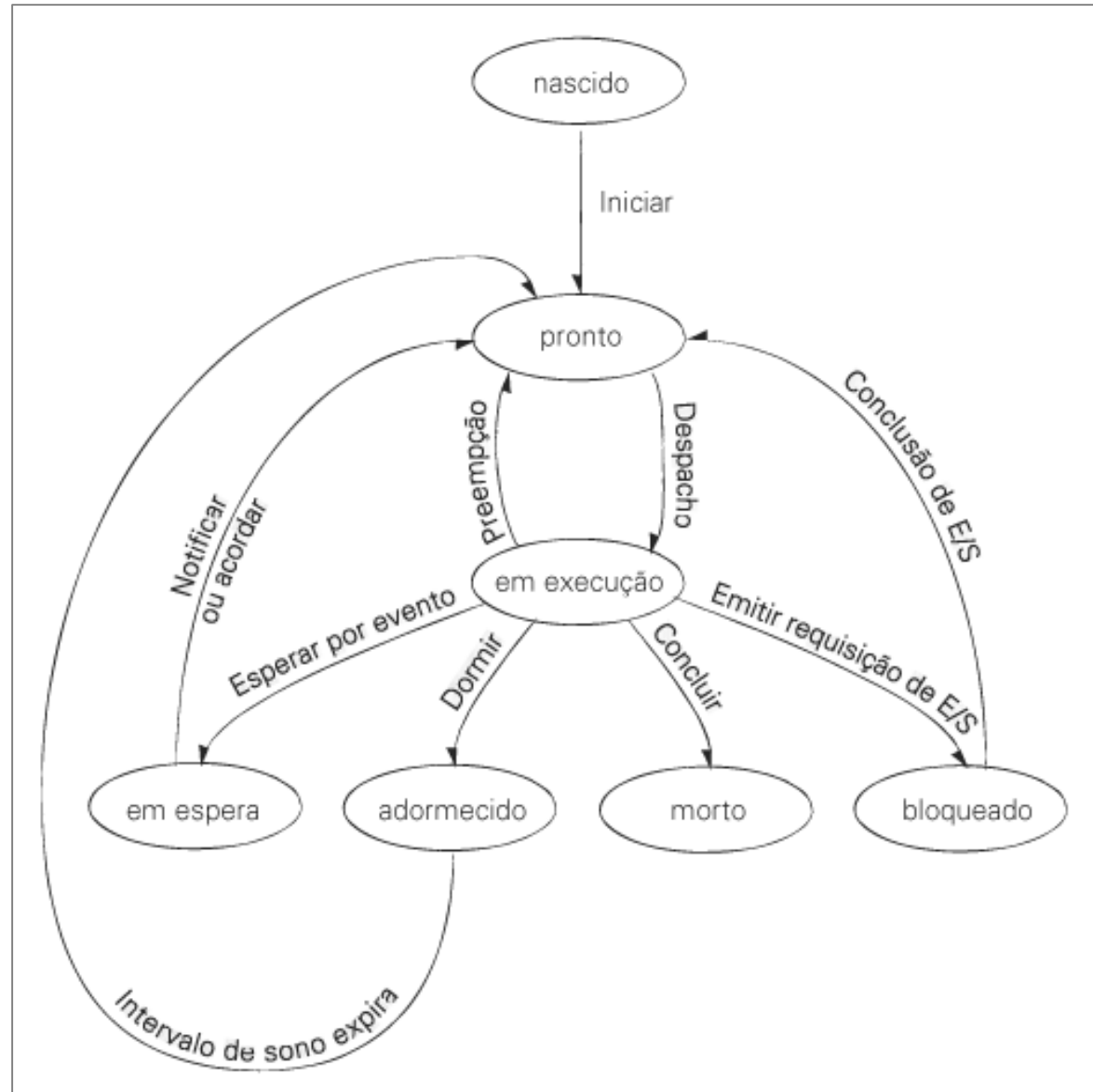
- Os threads tornaram-se proeminentes por causa de tendências subsequentes em relação:
 - Ao projeto de software
 - Maior simplicidade para exprimir tarefas inerentemente paralelas.
 - Ao desempenho
 - Maior escalonamento para sistemas com múltiplos processadores.
 - À cooperação
 - O custo operacional do espaço de endereço compartilhado é menor que o da IPC (*Interprocess Communication*).

Motivação na criação de threads

- Todo thread transita entre uma série de estados de thread distintos.
- Os threads e os processos têm muitas operações em comum (por exemplo, criar, sair, retomar e suspender).
- A criação de thread não requer que o sistema operacional inicialize recursos compartilhados entre os processos-pai e os respectivos threads.
 - Isso reduz o esforço de criação e término de threads, em comparação à criação e ao término de processo.

Estados de thread: ciclo de vida de um thread

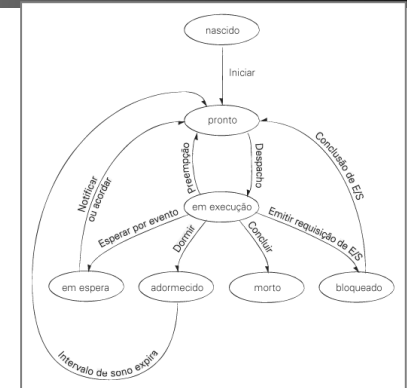
Ciclos de vida do thread.



Estados de thread: ciclo de vida de um thread

- **Estados de thread**

- Estado nascido
- Estado pronto (estado executável)
- Estado em execução
- Estado morto
- Estado bloqueado



Um thread entra no estado **bloqueado** quando deve esperar pela conclusão de uma requisição de E/S (ler dados de um disco). Um thread **bloqueado** não é despachado para um processador até que sua requisição de E/S tenha sido concluída. Nesse ponto, o thread retorna ao estado **pronto**, para que possa retomar a execução quando um processador estiver disponível.

- Estado de espera

Quando um thread deve esperar por um evento (por exemplo, movimento do mouse ou um sinal de outro thread) pode entrar no estado de **espera**. Uma vez nesse estado, ele volta ao estado **pronto** quando um outro thread o **notificar** (o termo **acordar** também é usado). Quando um thread em **espera** recebe um evento de notificação, ele transita do estado **em espera** para o estado **pronto**.

- Estado adormecido : O período de sono especifica por quanto tempo um thread ficará adormecido.

Por exemplo, um processador de texto pode conter um thread que grave periodicamente uma cópia do documento corrente em disco para recuperação posterior. Se o thread não dormisse entre backups sucessivos, exigiria um laço no qual ele testasse continuamente se deve ou não gravar uma cópia do documento no disco. Esse laço consumiria tempo de processador sem realizar trabalho produtivo, reduzindo o desempenho do sistema. Nesse caso, é mais eficiente que o thread especifique um intervalo de sono (igual ao período entre backups sucessivos) e entre no estado **adormecido**. O thread **adormecido** volta ao estado **pronto** quando seu intervalo de sono expira, momento em que grava uma cópia do documento em disco e retorna ao estado **adormecido**.

- Os threads e os processos têm operações em comum:
 - Criar
 - Sair (terminar)
 - Suspende
 - Retomar
 - Dormir
 - Acordar

Operações de thread

- Algumas das operações de thread não correspondem precisamente às operações de processo.
 - Cancelar
 - Indica que um thread deve ser terminado, mas não garante que o thread será terminado.
 - Os threads podem mascarar o sinal de cancelamento.
 - Associar
 - Para que um thread primário aguarde até que todos os outros threads terminem, ele se associa a esses threads.
 - O thread que se associa é bloqueado até que o thread ao qual ele se associou termine.

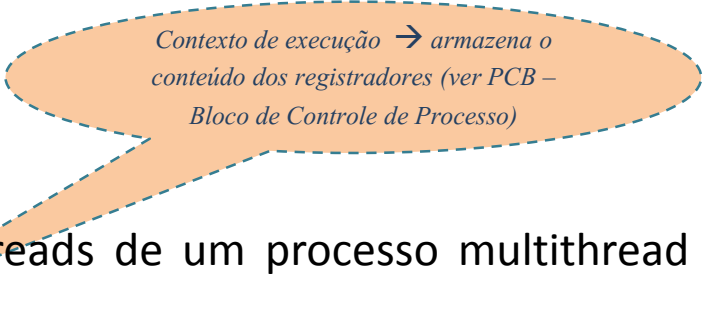
cancelar — Um thread ou processo pode fazer com que um thread termine prematuramente, cancelando-o. Diferentemente do término de um processo, o cancelamento de um thread não garante que ele termine. Isso porque threads podem desativar ou mascarar sinais; se um thread mascarar o sinal de cancelamento, ele não receberá o sinal até que o sinal de cancelamento seja reabilitado pelo thread.⁹ Entretanto, um thread não pode mascarar um sinal de abortar.

associar — Em algumas implementações de thread (por exemplo, Windows XP), quando um processo é iniciado, ele cria um **thread primário**. O thread primário age como qualquer outro thread, exceto que, se ele voltar, o processo termina. Para evitar que um processo termine antes que todos os seus threads concluam a execução, o thread primário tipicamente dorme até que cada thread que ele criar tenha concluído a execução. Nesse caso, diz-se que o thread primário se associa a cada um dos threads que cria. Quando um thread se associa a outro thread, o primeiro não executa até que o último termine.¹⁰

- Três são os modelos de thread mais conhecidos:
 - Threads de usuário
 - Threads de núcleo
 - Uma combinação de ambos

Threads de usuário

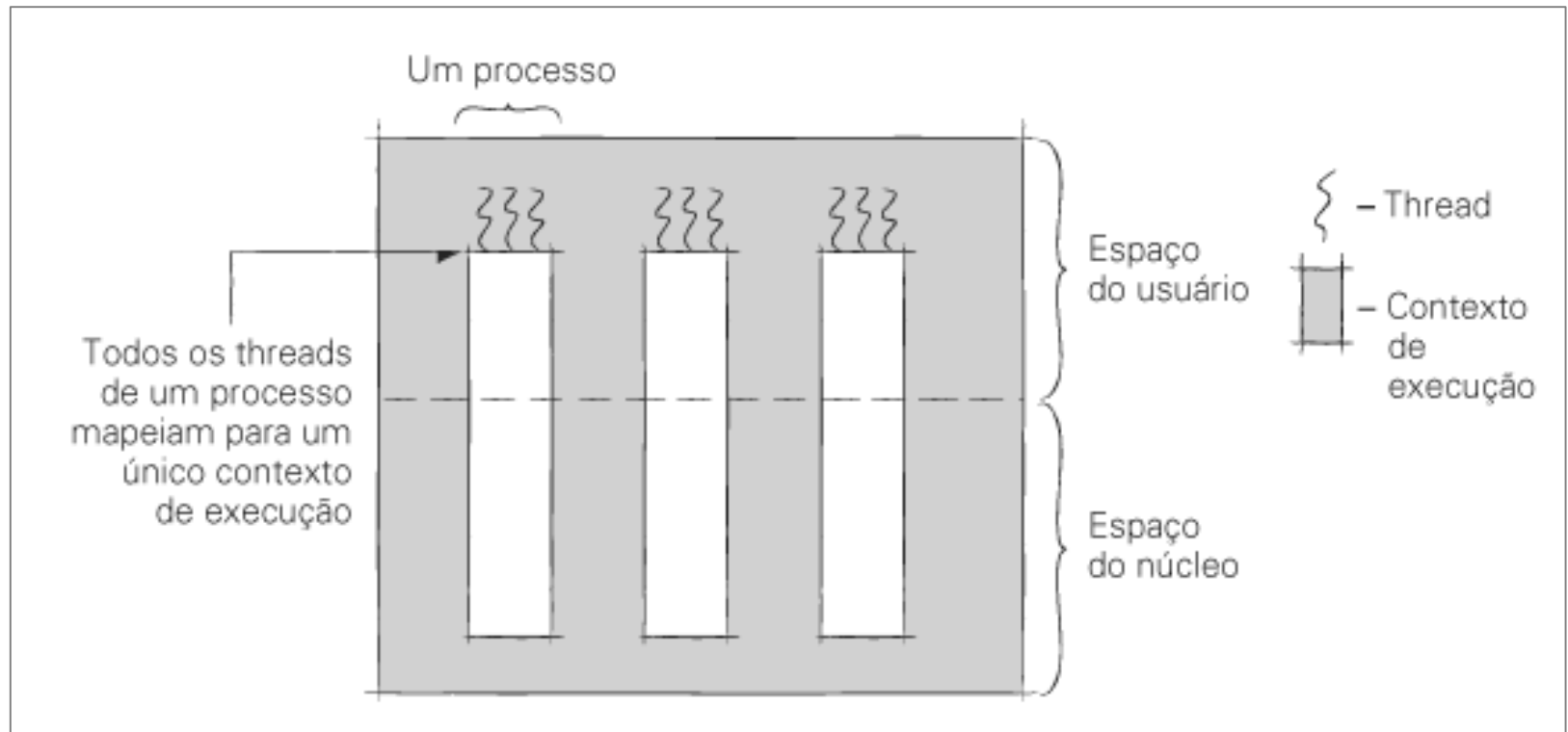
- Os threads de usuário executam operações de suporte a threads no espaço do usuário.
 - Isso significa que os threads são criados por bibliotecas em tempo de execução que não podem realizar instruções privilegiadas nem acessar as primitivas do núcleo diretamente.
- Implementação de thread de usuário
 - Mapeamentos de thread muitos-para-um
 - O sistema operacional mapeia todos os threads de um processo multithread para um único contexto de execução.
 - Vantagens
 - As bibliotecas de usuário podem escalonar seus threads para otimizar o desempenho.
 - A sincronização é realizada fora do núcleo, e isso evita chaveamento de contexto.
 - É mais portátil.
 - Desvantagens
 - O núcleo considera o processo multithread como um único thread de controle.
 - » Isso pode fazer com que o desempenho fique abaixo do ideal se um thread requisitar uma operação E/S.
 - » Não pode ser escalonado para executar em múltiplos processadores ao mesmo tempo.



Contexto de execução → armazena o conteúdo dos registradores (ver PCB – Bloco de Controle de Processo)

Threads de usuário

Threads de usuário.

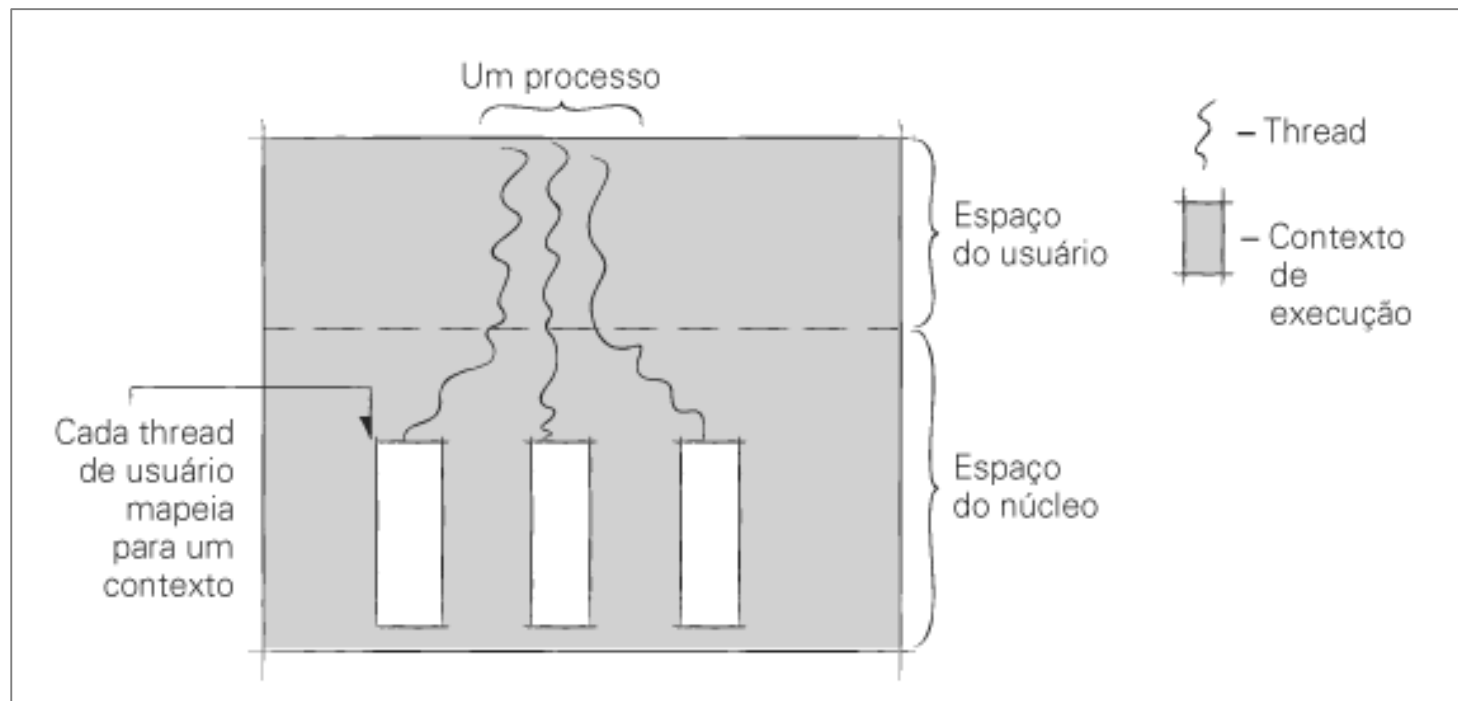


O desempenho do thread de usuário varia dependendo do sistema e do comportamento do processo. Muitas das deficiências de threads de usuário estão relacionadas com o fato de que, para o núcleo, um processo multithread é um único thread de controle. Por exemplo, threads de usuário não escalam bem para sistemas multiprocessadores, porque o núcleo não pode despachar os threads de um processo para múltiplos processadores simultaneamente, portanto, threads de usuário podem resultar em desempenho abaixo do ótimo em sistemas multiprocessadores.

- Os threads de núcleo tentam resolver as limitações dos threads de usuário mapeando cada thread para o seu próprio contexto de execução.
 - O thread de núcleo oferece mapeamento de thread um-para-um.
 - Vantagens: maior escalabilidade, interatividade e rendimento.
 - Desvantagens: sobrecarga decorrente do chaveamento de contexto e menor portabilidade em virtude de as APIs serem específicas ao sistema operacional.
- Os threads de núcleo nem sempre são a solução ideal para as aplicações.

Threads de núcleo

Threads de núcleo.

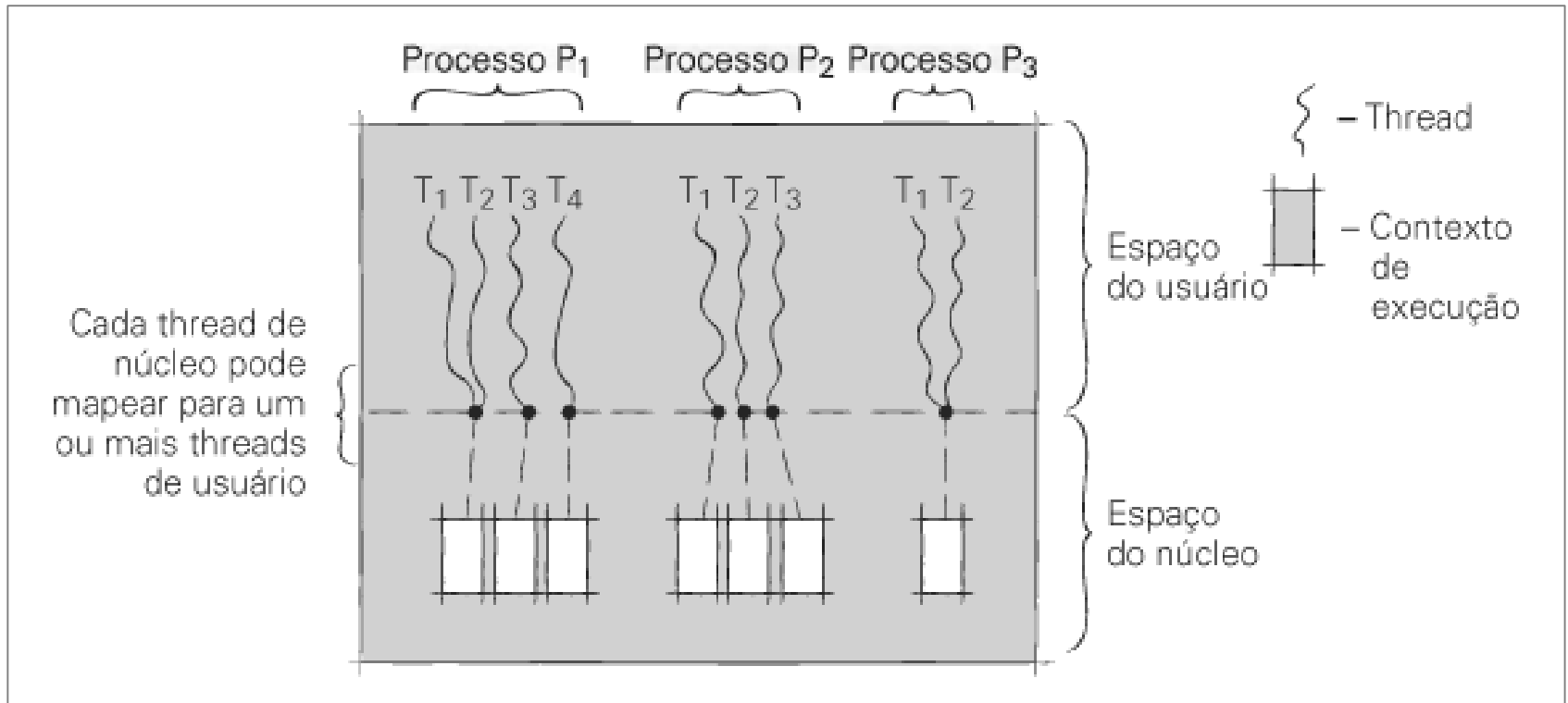


Combinação de threads de usuário e de núcleo

- Implementação da combinação de threads de usuário e de núcleo
 - Mapeamento de threads muitos-para-muitos (mapeamento de threads m-to-n)
 - O número de threads de usuário e de núcleo tem de ser o mesmo.
 - Em comparação com os mapeamentos de threads um-para-um, esse mapeamento consegue reduzir a sobrecarga implementando o reservatório de threads.
- Threads operários
 - Threads de núcleo persistentes que ocupam o reservatório de threads.
 - Os threads operários melhoram o desempenho em ambientes em que os threads são criados e destruídos com frequência.
 - Cada novo thread é executado por um thread operário.
- Ativação de escalonador
 - Técnica que permite que uma biblioteca de usuário escalone seus threads.
 - Ocorre quando o sistema operacional chama uma biblioteca de threads de usuário para determinar se algum de seus threads precisam ser reescalonados.

Combinação de threads de usuário e de núcleo

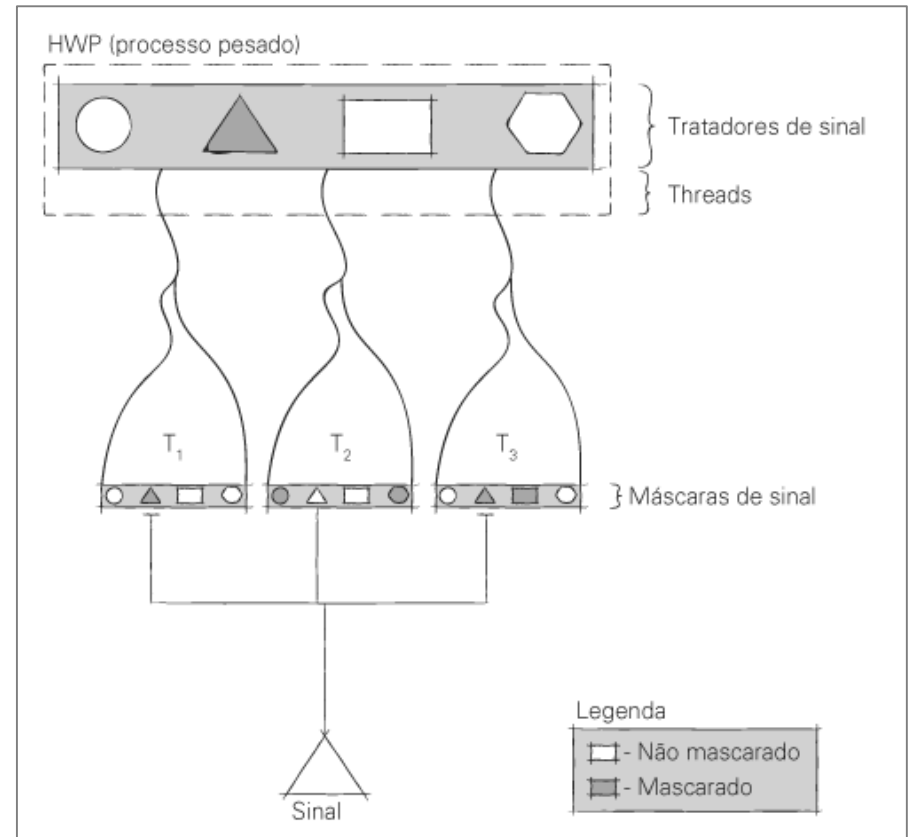
Modelo de operação de thread híbrido.



- Dois tipos de sinal
 - Síncrono:
 - Resulta diretamente da execução de um programa.
 - Pode ser emitido (entregue) para um thread que esteja sendo executado no momento.
 - Assíncrono
 - Resulta de um evento em geral não relacionado com a instrução corrente.
 - A biblioteca de threads precisa identificar todo receptor de sinal para que os sinais assíncronos sejam emitidos (entregues) devidamente.
- Todo thread normalmente está associado a um conjunto de sinais pendentes que são emitidos (entregues) quando ele é executado.
- O thread pode mascarar todos os sinais, exceto aqueles que deseja receber.

Entrega de sinal de thread

Mascaramento de sinal.



Um modo de implementar sinais pendentes para um modelo de thread muitos-para-muitos é criar um thread de núcleo para cada processo multithread que monitora e entrega seus sinais assíncronos. O sistema operacional Solaris 7 empregava um thread denominado Asynchronous Signal Lightweight Process (ASLWP) que monitorava sinais e gerenciava sinais pendentes para que fossem entregues ao thread apropriado, mesmo que o thread não estivesse *executando* no instante da emissão do sinal.³⁴

Se um processo multithread empregar uma biblioteca de nível de usuário, o sistema operacional simplesmente entregará todos os sinais ao processo porque não pode distinguir threads individuais. A biblioteca de nível de usuário registrará tratadores de sinais no sistema operacional que serão executados mediante o recebimento de um sinal. A biblioteca de threads de usuário do processo poderá, desse modo, entregar o sinal a qualquer um de seus threads que não o mascararam.^{35, 36}

- Término de threads (cancelamento)
 - É diferente de implementação de threads.
 - Se for terminado prematuramente, o thread pode provocar erros sutis nos processos, porque vários threads compartilham o mesmo espaço de endereço.
 - Determinadas implementações de thread permitem que um thread determine quando ele pode ser terminado, a fim de evitar que o processo entre em um estado inconsistente.

- Os threads que usam a API de thread POSIX são chamados de Pthreads.
 - A especificação POSIX determina que os registradores do processador, a pilha e a máscara de sinal sejam mantidos individualmente para cada thread.
 - A especificação POSIX especifica como os sistemas operacionais devem emitir sinais a Pthreads, além de especificar diversos modos de cancelamento de thread.

- O Linux aloca o mesmo tipo de descritor para processos e threads (tarefas).
- Para criar tarefas-filha, o Linux usa a chamada fork, baseada no Unix.
- Para habilitar os threads, o Linux oferece uma versão modificada, denominada clone.
 - Clone aceita argumentos que determinam os recursos que devem ser compartilhados com a tarefa-filha.

4.8 *POSIX e Pthreads*

POSIX (Portable Operating Systems Interface for Computing Environments) é um conjunto de padrões para interfaces de sistemas operacionais publicado pelo Portable Application Standards Committee (PASC) do IEEE baseados, em grande parte, no UNIX System V.³⁸ A especificação POSIX define uma interface-padrão entre threads e sua biblioteca de suporte a threads (veja no site deste livro: “Curiosidades, Padrões e conformidade: compatibilidade plugue-a-plugue”. Threads que usam a API de thread POSIX são denominados **Pthreads** (às vezes também denominados threads POSIX ou threads POSIX 1003.1c).³⁹ A especificação POSIX não se preocupa com os detalhes da implementação da interface de suporte a threads — Pthreads podem ser implementados no núcleo ou por bibliotecas de nível de usuário.

POSIX determina que os registradores do processador, a pilha e a máscara de sinal sejam mantidos individualmente para cada thread, e que qualquer outro recurso deva ser acessível globalmente a todos os threads no processo.⁴⁰ Também define um modelo de sinal que aborda muitas das preocupações discutidas na Seção 4.7, “Considerações sobre implementação de threads. De acordo com o POSIX, quando um thread gerar um sinal síncrono devido a uma exceção, tal como uma operação ilegal de memória, o sinal será entregue somente àquele thread. Se o sinal não for específico para um thread, tal como um sinal para matar um processo, a biblioteca de suporte a threads entregará o sinal a um thread que não o mascare. Se houver múltiplos threads que não mascaram o sinal de matar, ele será entregue a um desses threads. Mais importante, não se pode usar o sinal de matar para terminar determinado thread – quando um thread age obedecendo a um sinal de matar, o processo inteiro, incluindo todos os seus threads, terminarão. Esse exemplo demonstra uma outra propriedade importante do modelo de sinal POSIX: embora as máscaras de sinais sejam armazenadas individualmente em cada thread, as rotinas de tratamento de sinais são globais para todos os threads de um processo.^{41, 42}

Para terminar um thread particular, o POSIX fornece uma operação de cancelamento que especifica um thread visado e cujo resultado depende do modo de cancelamento desse thread. Se o thread visado preferir **cancelamento assíncrono**, ele poderá ser terminado a qualquer momento durante sua execução. Se o thread **adiar o cancelamento**, ele não será cancelado até verificar, explicitamente, se há uma requisição de cancelamento. O cancelamento adiado permite que um thread conclua uma série de operações antes de ser abruptamente terminado. Um thread também pode **desabilitar o cancelamento**, o que significa que ele não é notificado sobre uma operação de cancelamento ter sido requisitada.⁴³

Além das operações comuns discutidas na Seção 4.5, “Operações de thread”, a especificação POSIX fornece funções que suportam operações mais avançadas. Ela permite que programas especifiquem vários níveis de paralelismo e implementem uma variedade de políticas de escalonamento, entre elas algoritmos definidos por usuário e escalonamento em tempo real. A especificação também aborda sincronização usando travas, semáforos e variáveis de condição (veja o Capítulo 5, “Execução assíncrona concorrente”).^{44, 45, 46}

Hoje, poucos dos sistemas operacionais mais populares fornecem implementações Pthreads nativas completas, ou seja, no núcleo. Todavia, as bibliotecas de suporte a threads do POSIX existem para fornecer uma ampla faixa de suporte para vários sistemas operacionais. Por exemplo, embora o Linux não esteja em conformidade com o padrão POSIX por definição, o projeto Native POSIX Thread Library (NPTL) visa a fornecer uma biblioteca de suporte a threads conforme o padrão POSIX que emprega threads de núcleo no Linux.⁴⁷ Similarmente, a interface para a linha de sistemas operacionais Microsoft Windows (API Win 32) não segue o padrão POSIX, mas os usuários podem instalar um subsistema POSIX. [Nota: O sistema operacional Solaris 9 da Sun Microsystems fornece duas bibliotecas de suporte a threads: uma biblioteca de Pthreads em conformidade com o padrão POSIX e uma biblioteca de threads herdada do Solaris (denominada threads de interface do usuário (UI)). Há pouca diferença entre Pthreads e threads Solaris — esse último foi projetado para que chamadas a funções de thread Pthreads e Solaris vindas de dentro da mesma aplicação fossem válidas.⁴⁸

1. Qual a razão primária para criar interfaces de thread padronizadas, como Pthreads?
2. Quais modelos de funcionamento de threads o padrão POSIX exige?

4.9 Threads Linux

O suporte para threads no sistema operacional Linux foi introduzido como threads de usuário na versão 1.0.9 e como threads de núcleo na versão 1.3.56.⁴⁹ Embora o Linux suporte threads, é importante observar que muitos dos seus subsistemas de núcleo não distinguem entre threads e processos. De fato, o Linux aloca o mesmo tipo de descritor de processo a processos e threads, ambos denominados **tarefas**. O Linux usa a chamada ao sistema baseada no UNIX, denominada *fork*, para criar tarefas-filha. O Linux responde à chamada ao sistema *fork* criando uma tarefa que contém uma cópia de todos os recursos de seu pai (por exemplo, espaço de endereçamento, conteúdos de registradores, pilha).

Para habilitar o funcionamento de threads, o Linux fornece uma versão modificada da chamada ao sistema *fork* denominada *clone*. Similarmente à *fork*, *clone* cria uma cópia da tarefa que está chamando — na hierarquia do processo a cópia se torna uma filha da tarefa que emitiu a chamada ao sistema *clone*. Diferentemente de *fork*, *clone* aceita argumentos que especificam quais recursos compartilhar com o processo-filho. No nível mais alto de compartilhamento de recursos, as tarefas criadas por *clone* correspondem aos threads discutidos na Seção 4.2 “Definição de thread”.

A partir da versão 2.6 do núcleo, o Linux fornece um mapeamento de thread um-para-um que suporta um número arbitrário de threads no sistema. Todas as tarefas são gerenciadas pelo mesmo escalonador, o que significa que processos e threads de igual prioridade recebem o mesmo nível de serviço. O escalonador foi projetado para escalar bem até um grande número de processos e threads. A combinação de um mapeamento um-para-um com um algoritmo de escalonamento eficiente oferece ao Linux uma implementação de thread de alta escalabilidade (veja o quadro “Reflexões sobre sistemas operacionais, Escalabilidade”). Embora não suporte threads POSIX por definição, o Linux é distribuído com uma biblioteca de suporte a threads POSIX. No núcleo 2.4 uma biblioteca de suporte a threads, denominada *LinuxThreads*, fornecia funcionalidade POSIX, mas não estava totalmente em conformidade com a especificação POSIX. Um projeto mais recente, *Native POSIX Thread Library (NPTL)*, atingiu uma conformidade quase completa com o POSIX e provavelmente se tornará a biblioteca de suporte a threads padrão para o núcleo 2.6.⁵⁰

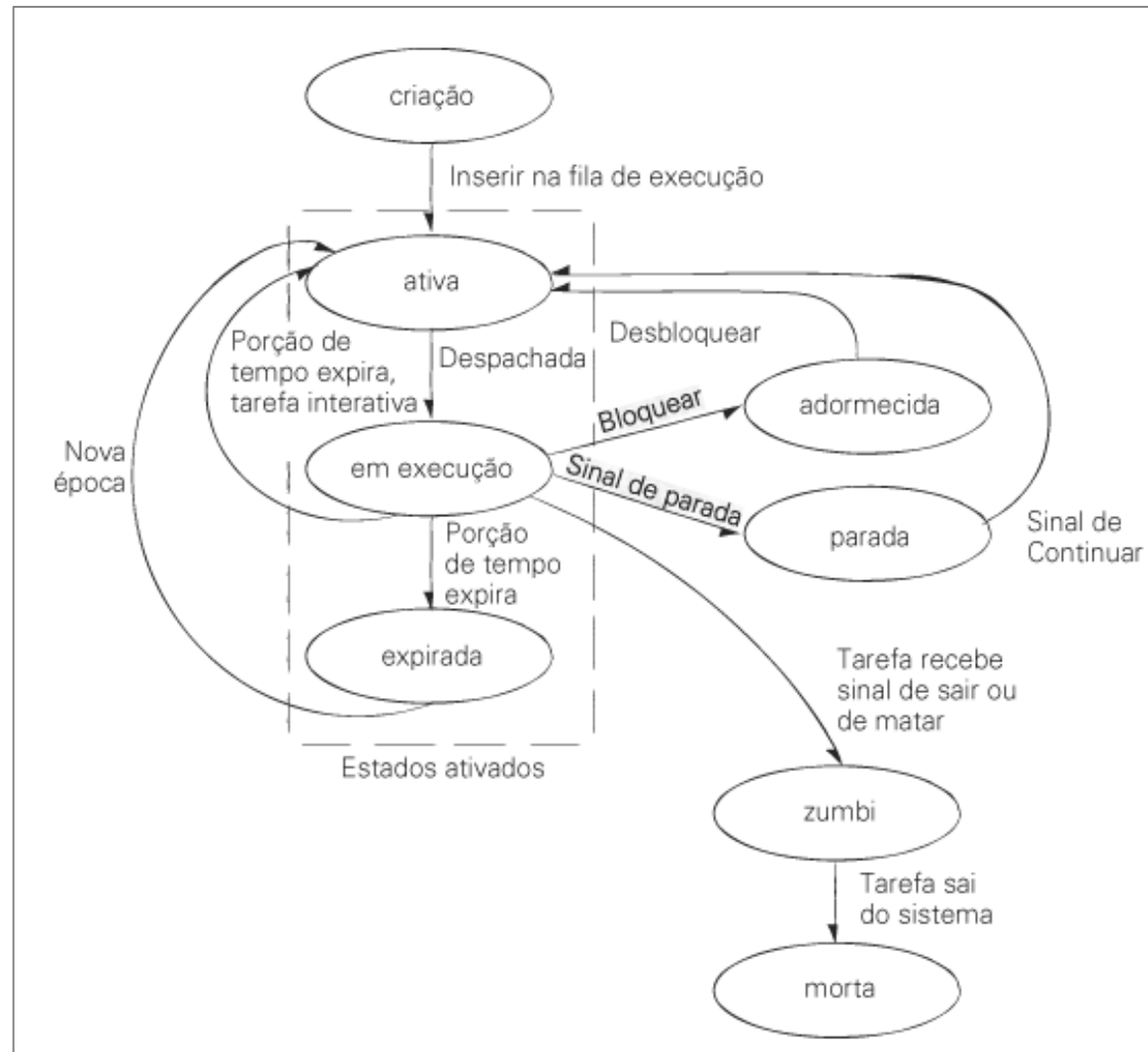
Cada tarefa da tabela de processo armazena informações sobre seu estado corrente (por exemplo, *em execução*, *parada*, *morta*). Uma tarefa que está no estado *de execução* pode ser despachada para um processador (Figura 4.7). Uma tarefa entra no estado *adormecido* quando está dormindo, bloqueada ou não pode executar em um processador por qualquer outra razão.

Ela entra no estado *parado* quando recebe um sinal de parada (isto é, de suspensão). O estado *zumbi* indica que uma tarefa foi terminada, mas ainda não foi removida do sistema. Por exemplo, se uma tarefa contiver diversos threads, ela entrará em estado *zumbi* enquanto notifica seus threads que recebeu um sinal de término. Uma tarefa no estado *morto* pode ser removida do sistema. Esses estados são discutidos mais detalhadamente na Seção 20.5.1, “Organização de processos e threads”.

1. Explique a diferença entre as chamadas ao sistema *fork* e *clone* no Linux.
2. Qual a diferença entre o estado *zumbi* e o estado *morta*?

Threads Linux

Diagrama de transição de estado de tarefa do Linux.





Reflexões sobre sistemas operacionais

Paralelismo

Um modo de implementar paralelismo é fazê-lo na máquina local com técnicas como multiprogramação, multithread, multiprocessamento e paralelismo maciço. O hardware de computador é construído para poder realizar processamento em paralelo com entrada/saída. Multiprocessadores são construídos para ter diversos processadores trabalhando em paralelo — paralelismo maciço leva

isso ao extremo com centenas, milhares ou até mais processadores trabalhando em paralelo.

Hoje, um outro tipo de paralelismo está se tornando proeminente, a saber, computação distribuída em redes de computadores. Estudaremos computação distribuída nos capítulos 16 a 18, nos quais examinamos redes de computadores e as questões da construção de sistemas operacionais distribuídos. Um

sistema operacional é, primariamente, um administrador de recursos. Durante anos esses recursos foram o hardware, o software e os dados de um sistema de computador local. Hoje, um sistema operacional distribuído deve gerenciar recursos onde quer que eles residam, seja no sistema do computador local, seja em sistemas de computadores distribuídos em redes como a Internet.



Reflexões sobre sistemas operacionais

Conformidade com padrões

Imagine tentar acessar a Internet sem os protocolos padronizados de rede. Ou imagine comprar uma lâmpada se os bocais não fossem padronizados. E até mais fundamentalmente, imagine o que seria dirigir se cada cidade resolvesse atribuir às luzes verde e vermelha dos semáforos significados diferentes dos padrões correntes de 'pare' e 'siga'. Há diversas organizações importantes, nacionais e internacionais, que promovem padrões para a indústria de computadores, tais como POSIX, ANSI (American National Standards Institute), ISO (International Orga-

nization for Standardization), OMG (Object Management Group), W3C (World Wide Web Consortium) e muitas mais.

Um projetista de sistemas operacionais deve estar a par dos muitos padrões aos quais um sistema deve obedecer. Muitas vezes esses padrões não são estáticos; pelo contrário, evoluem conforme mudam as necessidades da comunidade mundial dos usuários de computadores e de comunicação.

Padrões também têm desvantagens — podem retardar ou até mesmo sufocar a inovação. Eles forçam

as organizações de desenvolvimento de sistemas operacionais (e outras) a gastar significativas quantidades de tempo e dinheiro para obedecer a um vasto conjunto de padrões, muitos dos quais obscuros e até desatualizados. Referimo-nos a padrões e a organizações de padrões por todo o livro e consideramos cuidadosamente o significado e o impacto da conformidade aos padrões.



Reflexões sobre sistemas operacionais

Escalabilidade

As necessidades de capacidade de computação dos usuários tendem a aumentar com o tempo. Sistemas operacionais precisam ser escaláveis, ou seja, devem poder se ajustar dinamicamente à medida que mais capacidades de software e hardware são adicionadas a um sistema. Um sistema operacional

multiprocessador, por exemplo, deve escalar suavemente do gerenciamento de uma configuração de dois processadores para o gerenciamento de uma configuração de quatro processadores. Veremos que, hoje, a maioria dos sistemas emprega uma arquitetura de drivers de dispositivos que facilita a

adição de novos tipos de dispositivos, mesmo os que não existiam quando o sistema operacional foi implementado. Por todo este livro discutiremos técnicas para tornar sistemas operacionais escaláveis. Concluiremos com discussões de escalabilidade no Linux e no Windows XP.

EXERCÍCIOS

1. O texto menciona que recursos de multithread não estão disponíveis diretamente em linguagens como C e C++. Como, mesmo assim, programadores conseguem escrever códigos multithread nessas linguagens?

2. Qual a vantagem fundamental que você obteria executando uma aplicação multithread em um sistema multiprocessador em vez de um sistema uniprocessador?

1. Por que os processos tradicionais são chamados de processos pesados?

2. Porque é difícil escrever aplicações multithread portáteis?

1. Como um projeto de software melhorado ajuda a fazer que aplicações multithread executem mais rapidamente?

2. Por que threads do mesmo processo em geral se comunicam mais eficientemente do que em processos separados?

1. Como um thread entra no estado *morto*?

2. Quais as semelhanças entre os estados de espera, bloqueado e adormecido? Quais as diferenças?

1. Qual a diferença entre um sinal de cancelamento e um sinal de abortar, discutidos na Seção 3.5.1, “Sinais”?

2. Por que a criação de threads requer, tipicamente, um número menor de ciclos de processo do que a criação de processos?

1. Explique por que implementações de thread de usuário promovem a portabilidade.

2. Por que, em mapeamentos de thread muitos-para-um, o sistema operacional bloqueia o processo multithread inteiro quando um único thread fica bloqueado?

1. Em quais cenários threads de núcleo são mais eficientes do que threads de usuário?

2. Por que uma aplicação de software escrita para threads de núcleo é menos portátil do que um software escrito para threads de usuário?

EXERCÍCIOS

1. Por que é ineficiente uma aplicação especificar um tamanho de reservatório de threads maior do que o número máximo de threads de usuário *prontos* em qualquer instante durante a execução da aplicação?
2. Como as ativações de escalonador melhoram o desempenho em um mapeamento de thread muitos-para-muitos?

1. Por que a entrega de um sinal síncrono é mais simples do que a entrega de um sinal assíncrono?
2. Explique como o ASLWP resolve o problema do tratamento de sinais em um modelo de thread muitos-para-muitos.

1. Cite três modos pelos quais um thread pode terminar.
2. Por que se deve permitir que um thread desabilite seu sinal de cancelamento?