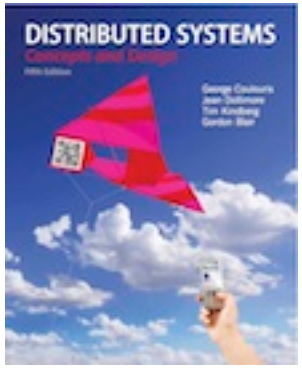
Slides for Chapter 7: Operating System support



From **Coulouris, Dollimore, Kindberg and Blair**

Distributed Systems: Concepts and Design

Edition 5, © Addison-Wesley 2012

7.1 Introdução

- Este capítulo descreve como o middleware é suportado pelos recursos do sistema operacional nos nós de um sistema distribuído.
- O sistema operacional facilita o encapsulamento e a proteção dos recursos dentro dos servidores e disponibiliza mecanismos necessários para acessar esses recursos, incluindo a comunicação e o escalonamento.
- Um tema importante do capítulo é a função do núcleo ou kernel do sistema operacional.

7.1 Introdução

- Abaixo da camada de middleware está a camada do sistema operacional (SO)
- A tarefa de qualquer sistema operacional é fornecer abstrações dos recursos físicos subjacentes – processadores, memória, comunicação e mídias de armazenamento.
- O sistema operacional gerencia os recursos físicos de um computador apresentando-os através de suas abstrações via uma interface denominada de chamada de sistema.
- O UNIX e o Windows são SO de Rede. Eles têm recurso de rede incorporado, sendo capazes de acessar recursos remotos.

7.1 Introdução

- A característica marcante é que os nós que estão sendo executados em um sistema operacional de rede mantém a autonomia no gerenciamento de seus próprios recursos de processamento. Em outras palavras, existem várias imagens do sistema, uma por nó: a conexão de nós em rede deve ser explícita e não transparente.
- SO Distribuído: SO no qual o usuário nunca se preocupasse com o local onde seus programas são executados, ou com a localização de quaisquer recursos. Haveria uma imagem única do sistema. Este tipo de SO não existe.

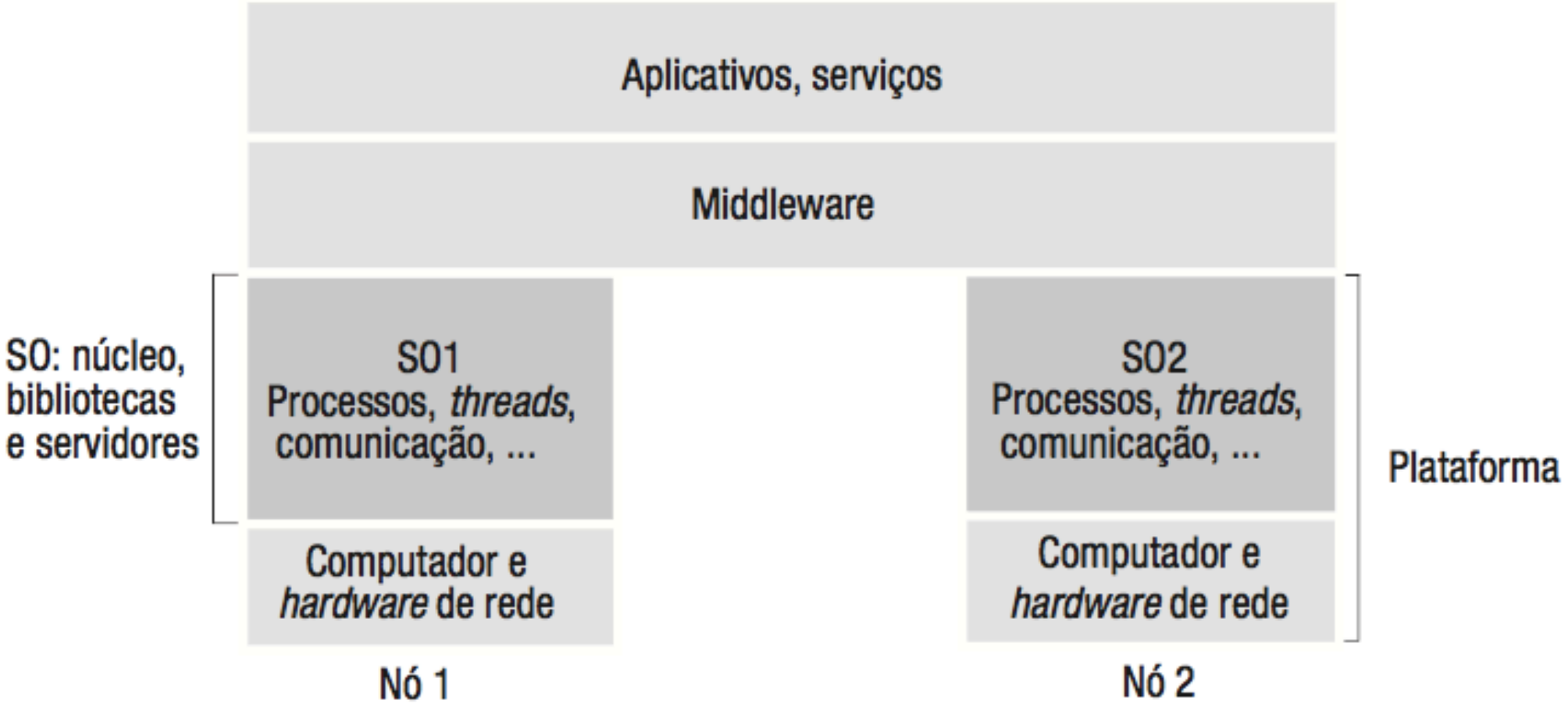
7.1 Introdução

Conclusão:
SO em rede + middleware = Sistema Distribuído

7.2 A camada do sistema operacional

- O par SO-hardware é genericamente denominado *plataforma*.
- O SO executado em um nó fornece seu próprio conjunto de abstrações (processamento, armazenamento e comunicação).
- O middleware utiliza essas abstrações para implementar seus mecanismos de invocações remotas entre objetos ou processos nos nós.

Figura 7.1
Camadas do Sistema



A Figura 7.1 mostra como a camada de sistema operacional, em cada um de dois nós, oferece suporte para uma camada de *middleware* comum para fornecer uma infraestrutura distribuída para aplicativos e serviços.

7.2 A camada do sistema operacional

- Os núcleos e os processos servidores gerenciam os recursos os apresentam aos clientes por meio de uma interface.
- Encapsulamento: devem fornecer uma interface de serviço útil para seus recursos – os detalhes devem ser ocultados dos clientes.
- Proteção: contra acessos ilegítimos - permissões de leitura e contra acessos de processos.
- Processamento concorrente: os clientes podem compartilhar recursos e acessá-los concorrentemente. Os gerenciadores de recurso são responsáveis pela transparência da concorrência.

7.2 A camada do sistema operacional

mecanismo de invocação: Os processos clientes acessam recursos fazendo invocações a métodos remotos em um objeto servidor ou chamadas de sistema ao núcleo. Tarefas da invocação:

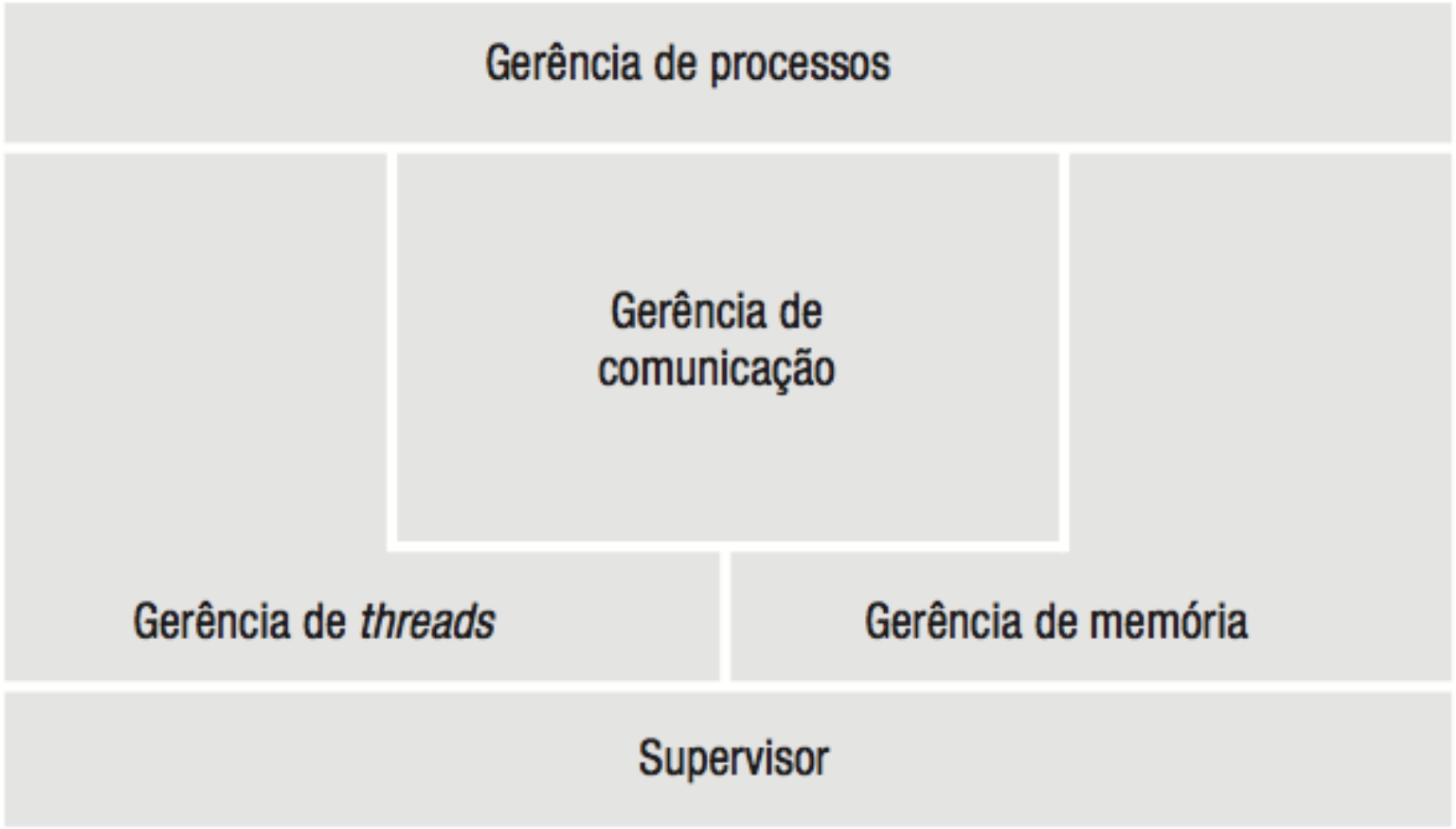
- Comunicação: parâmetros e resultados de operação precisam ser passados por gerenciadores de recursos, via rede ou internamente ao próprio computador.
- Escalonamento: quando uma operação é invocada, seu processamento deve ser agendado dentro do núcleo ou do servidor.

7.2 A camada do sistema operacional

Os componentes básicos do SO:

- Gerência de processos: trata da criação de processos e das operações neles executadas.
- Gerência de threads: criação de threads, sincronização e escalonamento. As threads são fluxos de execução associados aos processos.
- Gerência de comunicação: trata da comunicação entre threads associadas a diferentes processos.
- Gerência de memória: trata do gerenciamento da memória física e virtual.
- Supervisor: interrupções, chamadas de sistema e exceções; do controle da unidade de g. de memória; do processador e da unidade em ponto flutuante. No Windows, ele é conhecido como HAL (Hardware Abstraction Layer).

Figura 7.2
Funcionalidades básicas de um SO

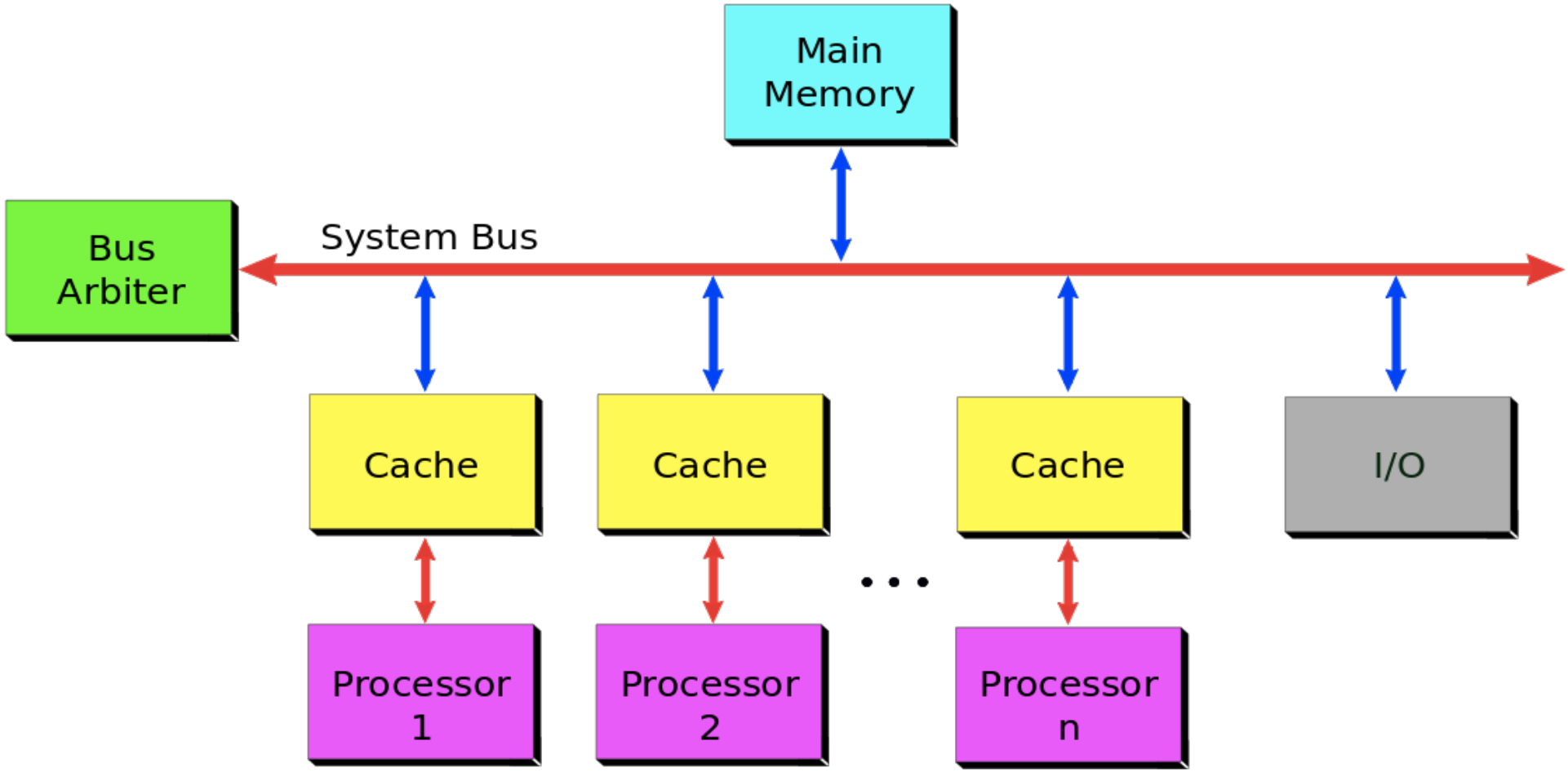


Multiprocessadores de memória compartilhada • Os multiprocessadores de memória compartilhada vêm equipados com vários processadores que compartilham um ou mais módulos de memória (RAM). Os processadores também podem ter sua própria memória privativa. Os multiprocessadores podem ser construídos de diversas formas [Stone 1993]. Os mais simples e mais baratos são construídos por meio da incorporação de uma placa contendo alguns processadores (2–8) em um computador pessoal.

Na arquitetura de processamento simétrico, cada processador executa o mesmo núcleo e os núcleos desempenham papéis equivalentes no gerenciamento dos recursos de hardware. Os núcleos compartilham suas principais estruturas de dados, como a fila de threads prontas para execução, mas alguns de seus dados de trabalho são privativos. Cada processador pode, simultaneamente, executar uma thread, acessando dados na memória compartilhada, que pode ser privativa (protegida pelo hardware) ou compartilhada com as demais threads.

Os multiprocessadores podem ser usados para muitas tarefas de computação de alto desempenho. Nos sistemas distribuídos, eles são particularmente úteis para a implementação de servidores, pois o servidor pode executar um único programa com várias threads, tratando simultaneamente requisições de diversos clientes – por exemplo, fornecendo acesso a um banco de dados compartilhado (veja a Seção 7.4)

SMP - Symmetric Multiprocessor System



By Ferruccio Zulian - Milan.Italy

7.3 Proteção

Proteção contra acessos ilegítimos: um código benigno, contendo um erro ou um comportamento imprevisto, pode fazer com que parte do sistema aja incorretamente.

- Núcleo e proteção: O núcleo é um programa diferenciado pelo fato de que permanece carregado a partir da inicialização do sistema e seu código é executado com privilégios de acesso completos aos recursos físicos presentes em seu computador.
  - é executado com o processador no modo supervisor (privilegiado), e o núcleo providencia para que os outros processos sejam executados no modo usuário (não privilegiado).

7.4 Processos e threads

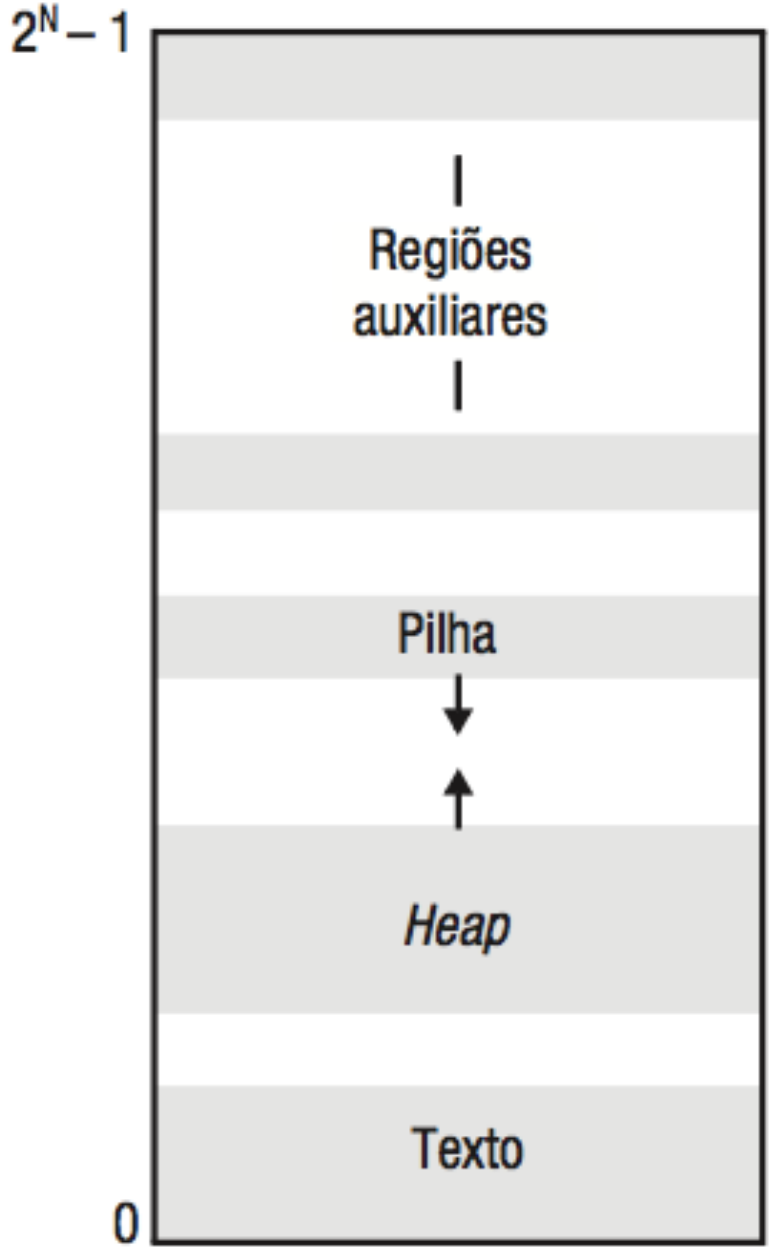
Processo: consiste em um ambiente de execução, junto a uma ou mais threads.

Thread: é uma atividade (fluxo de instruções).

Ambiente de execução (processo): é a unidade de gerenciamento de recursos, gerenciado pelo núcleo, aos quais suas threads têm acesso. Consiste em:

- um espaço de endereçamento;
- recursos de sincronização e Comunicação entre threads, como semáforos e interfaces de comunicação (por exemplo, soquetes);
- recursos de nível mais alto, como arquivos e janelas abertas.

7.4.1 Espaços de endereçamento



7.4.1 Espaços de endereçamento

- Os processos acessam as mesmas posições de memória nas regiões compartilhadas, enquanto suas regiões não compartilhadas permanecem protegidas. Os usos de regiões compartilhadas incluem:
  - Bibliotecas
  - Núcleo
  - Compartilhamento de dados e comunicação

7.4.2 Criação de um novo processo

- Tradicionalmente: operação indivisível fornecida pelo sistema operacional (fork do UNIX seguido de exec para carregar um novo código e executá-lo).
- Para um sistema distribuído: precisa levar em conta a utilização de vários computadores (distintos serviços de sistema) dois aspectos:
    - A escolha de um computador (host) de destino.
    - A criação de um ambiente de execução (e de uma thread inicial dentro dele).

7.4.3 Threads

Arquiteturas de servidores multithreadeds

a) Arquitetura do conjunto de trabalhadores: O servidor cria um conjunto fixo de threads "trabalhadores" para processar os pedidos.

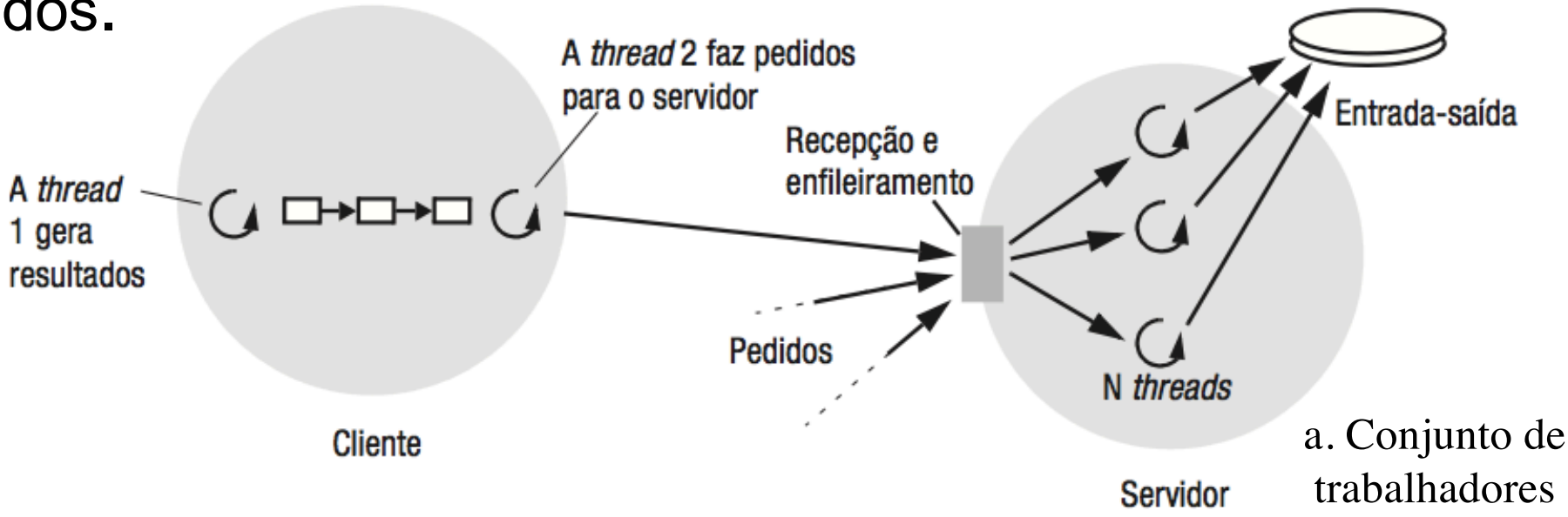


Figura 7.5 Cliente e servidor baseados em *threads*.

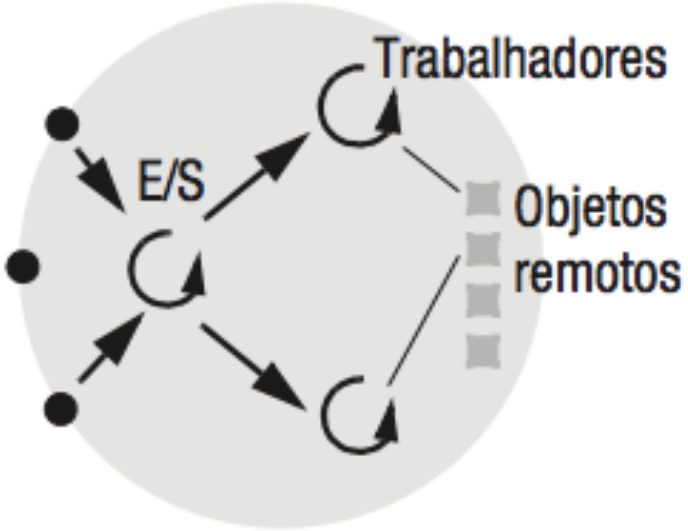
Desvantagens: (1) falta de flexibilidade (número de *threads* trabalhadores pode ser pequeno) (2) grande quantidade de chaveamentos de contexto entre as *threads* de E/S e as trabalhadoras, pois elas manipulam uma fila compartilhada.

7.4.3 Threads

Arquiteturas de servidores multithreadeds

b) arquitetura thread por pedido: a thread de E/S gera uma nova thread trabalhadora para cada pedido, e esse trabalhador se auto terminará quando tiver processado o pedido.

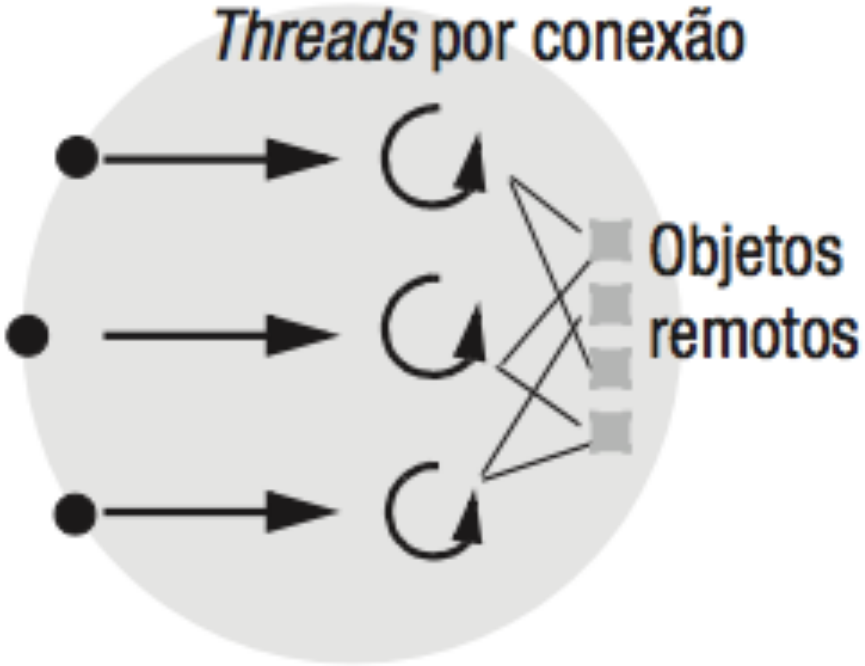
- Vantagem: as threads não disputam uma fila compartilhada e a taxa de rendimento é potencialmente maximizada, pois a thread de E/S pode criar tantos trabalhadoras quantos forem os pedidos pendentes.
- Desvantagem: a sobrecarga das operações de criação e destruição de threads.



7.4.3 Threads

Arquiteturas de servidores multithreadeds

c) A arquitetura de thread por conexão: O servidor cria uma nova thread trabalhador quando um cliente estabelece uma conexão e destrói a thread quando o cliente fecha a conexão. Nesse meio-tempo, o cliente pode fazer vários pedidos pela conexão, destinados a um ou mais objetos remotos.



7.4.3 Threads

Arquiteturas de servidores multithreadeds

d) A arquitetura de thread por objeto: associa uma thread a cada objeto remoto. Uma thread de E/S recebe pedidos e os enfileira para os trabalhadores, uma fila por objeto.

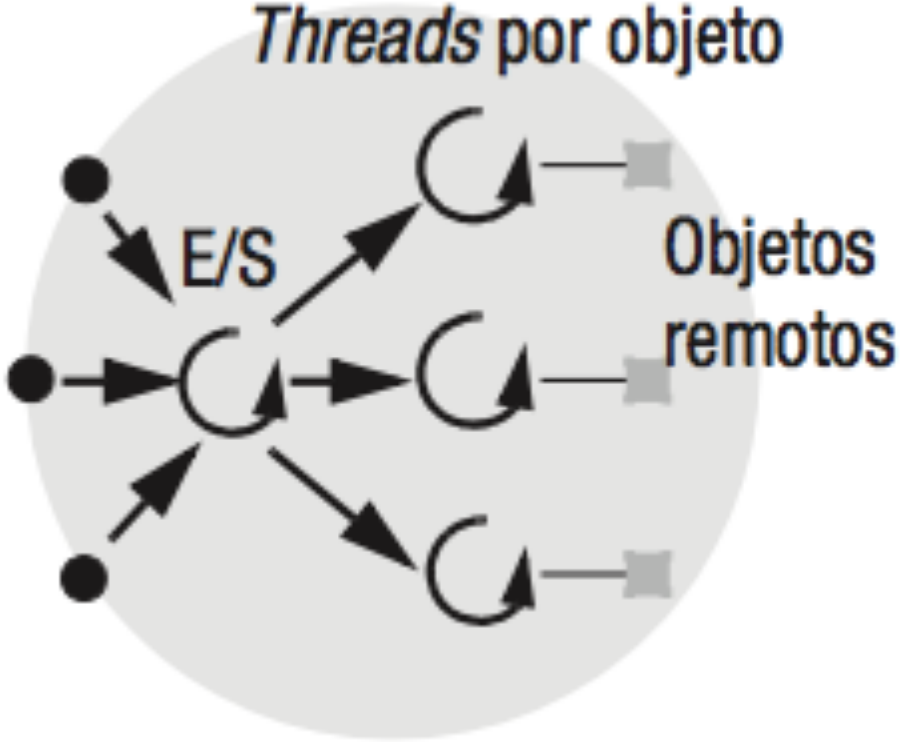
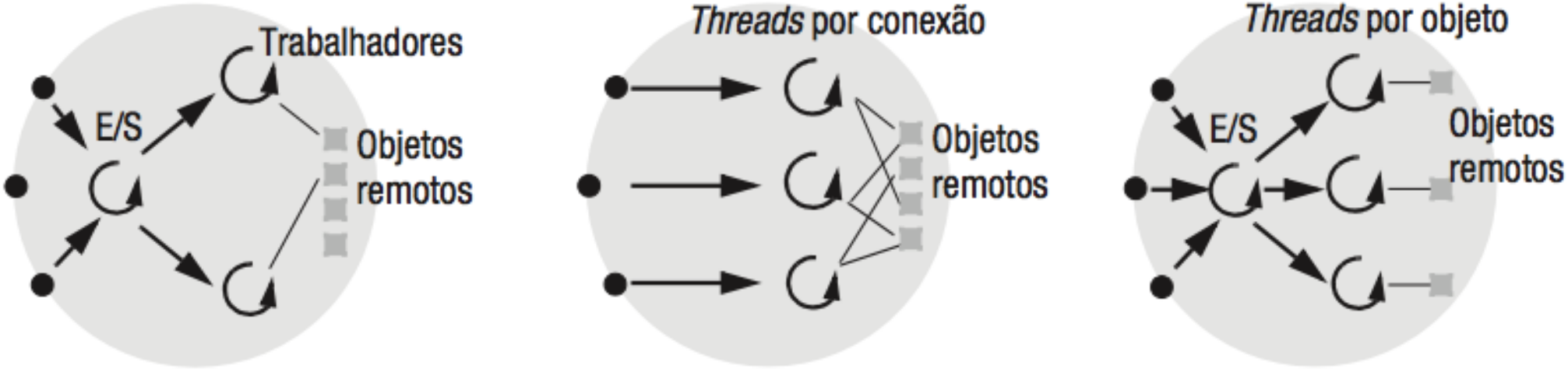


Figure 7.6
Arquiteruras baseadas em threads



b. Thread por pedido

c. Thread por conexão

d. Thread por objeto

Nas duas últimas arquiteturas, o servidor se beneficia das menores sobrecargas de gerenciamento de thread, se comparadas à arquitetura de thread por pedido. A desvantagem é que os clientes podem sofrer atrasos quando uma thread trabalhador tem vários pedidos pendentes, mas outra thread não tem trabalho a fazer.

7.4.3 Threads

Threads em clientes: As threads podem ser úteis para os clientes, assim como para os servidores.

- As invocações a métodos remotos normalmente bloqueiam o chamador, mesmo quando não há rigorosamente nenhuma necessidade de esperar.
- Esse processo cliente pode incorporar uma segunda thread, a qual realiza as invocações a métodos remotos e bloqueia, enquanto a primeira thread é capaz de continuar calculando mais resultados.
- O caso dos clientes multithreaded fica evidente no exemplo de navegadores Web: Os usuários sentem demoras substanciais enquanto páginas são procuradas; portanto, é fundamental que os navegadores manipulem vários pedidos de páginas Web paralelamente.

7.4.3 Threads
Threads versus múltiplos processos

Ambiente de execução	*Thread*
Tabelas de espaço de endereçamento	Registradores internos do processador salvos
Interfaces de comunicação, arquivos abertos	Prioridade e estado da execução (como *BLOCKED*)
Semáforos, outros objetos de sincronização	Informações do tratamento da interrupção de *software*
Lista de identificadores de *thread*	Identificador do ambiente de execução
Páginas do espaço de endereçamento residentes na memória; entradas de cache em *hardware*	

Figura 7.7 Estados associados aos ambientes de execução e às *threads*.

Comparação entre processos e *threads*:

•Criar uma nova *thread* dentro de um processo existente é menos oneroso do que criar um processo.

•O chaveamento para uma *thread* diferente dentro de um mesmo processo é menos oneroso do que chavear entre *threads* pertencentes a processos diferentes.

•As *threads* dentro de um processo podem compartilhar dados e outros recursos conveniente e eficientemente, em comparação a processos distintos.

•Porém, as *threads* dentro de um processo não são protegidas umas das outras.

7.4.3 Threads

Programação com threads Java

Classe Thread: Construtor e métodos de gerenciamento de *threads* Java.

Thread(ThreadGroup group, Runnable target, String name)
Cria uma nova thread no estado SUSPENDED, a qual pertencerá a group e será identificada como name; a thread executará o método run( ) de target.

setPriority(int newPriority), getPriority( ) Configura e retorna a prioridade da thread.

run( )
A thread executa o método run( ) de seu objeto de destino, caso ele tenha um; caso contrário, ela executa seu próprio método run( ) (Thread implementa Runnable).

start( )
Muda o estado da thread de SUSPENDED para RUNNABLE.

sleep(long millisecs)
Passa a thread para o estado SUSPENDED pelo tempo especificado.

yield( )
Passa para o estado READY e ativa o escalonamento.

destroy( )
Termina (destrói) a thread.

7.4.3 Threads

Programação com threads Java

Métodos de sincronização Thread e Object.

thread.join(long millisecs)
Bloqueia até a thread terminar, mas não mais que o tempo especificado.

thread.interrupt( )
Interrompe a thread: a faz retornar de uma invocação a método que causa bloqueio, como sleep( ).

object.wait(long millisecs, int nanosecs)
Bloqueia a thread até que uma chamada feita para notify( ), ou notifyAll( ), em object, ative a thread, ou que a thread seja interrompida ou, ainda, que o tempo especificado tenha decorrido.

object.notify( ), object.notifyAll( )
Ativa, respectivamente, uma ou todas as threads que tenham chamado wait( ) em object.

7.4.3 Threads

Ciclo de Vida de uma Thread Java

- Uma nova thread é criada na mesma máquina virtual Java (JVM) que da sua criadora, no estado SUSPENDED.
- Após se tornar RUNNABLE com o método start(), ela executa o método run() de um objeto fornecido em seu construtor.
- A JVM e as threads são todas executadas em um único processo no sistema operacional.
- As threads podem receber uma prioridade.
- Uma thread termina quando retorna do método run() ou quando seu método destroy() é chamado.
- Os programas podem gerenciar as threads em grupos.

7.4.3 Threads

Sincronização de Threads Java (Monitores)

- Principais problemas: o compartilhamento dos objetos e as técnicas usadas para coordenação e cooperação entre threads.
- As variáveis locais de cada thread, presentes nos métodos, são privativas – assim como suas pilhas. Entretanto, as threads não possuem cópias privativas de variáveis estáticas (classe), nem variáveis de instância de objeto.
- Em princípio, condições de corrida *(race conditions*) podem surgir quando as threads manipulam estruturas de dados, como as filas, concorrentemente.

7.4.3 Threads

Sincronização de Threads Java (Monitores)

- Java fornece a palavra-chave synchronized para designar a construção conhecida como monitor para a coordenação de threads.
- Os programadores designam métodos inteiros, ou blocos de código arbitrários, como pertencentes a um monitor associado a um objeto individual.
- A garantia do monitor é que no máximo uma thread pode ser executada dentro dele em certo momento.
  - método wait(): bloqueia a thread até que ocorra uma certa condição.
  - notify() / notifyAll(): desbloqueia uma/todas threads que estão esperando nesse objeto.
  - join(): bloqueia o chamador até o término de uma outra thread.
  - interrupt(): ativa prematuramente uma thread que esteja bloqueada.

7.4.3 Threads

Sincronização de Threads Java (Monitores)

- É necessário cuidado:
  - as garantias do monitor de Java se aplicam somente ao código synchronized de um objeto;
  - uma classe pode ter uma mistura de métodos synchronized e não-synchronized;
  - o monitor implementado por um objeto Java tem apenas uma variável de condição implícita, enquanto, em geral, um monitor pode ter diversas variáveis de condição explícitas.

7.6 Arquiteturas de Sistemas Operacionais

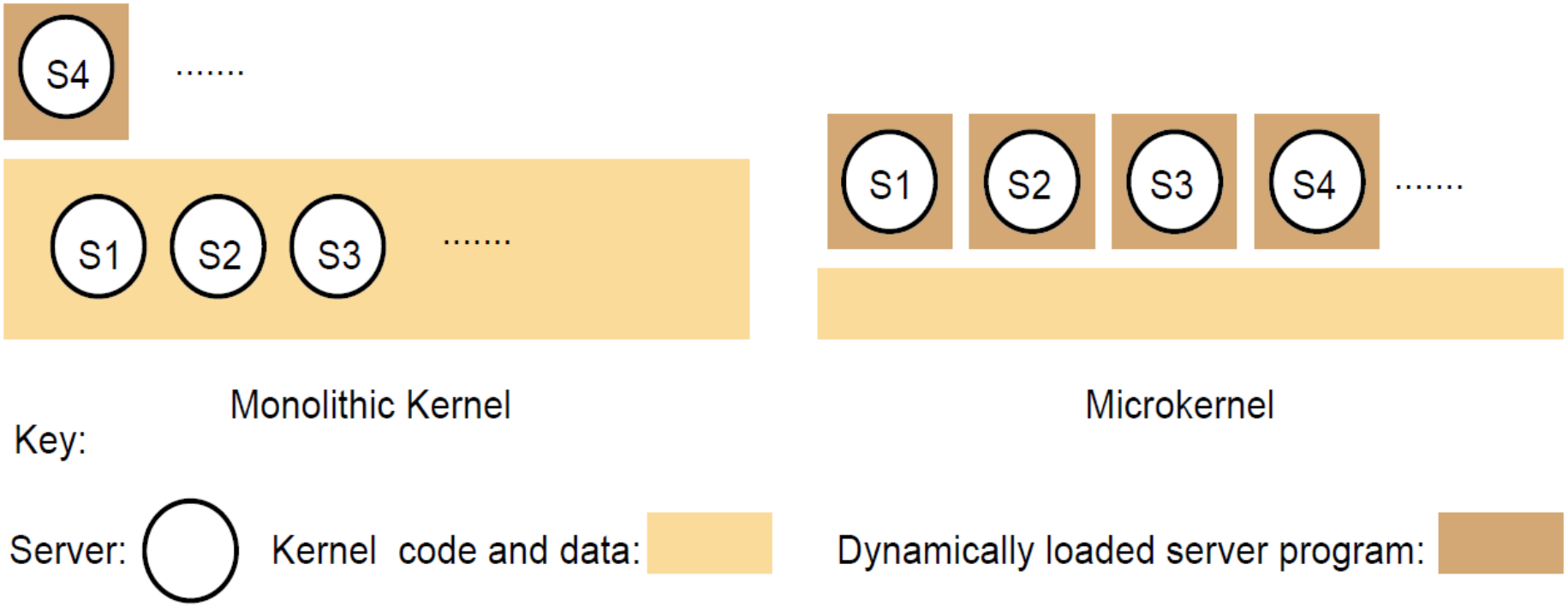
Núcleos Monolíticos X micronúcleos (microkernel)

Esses projetos diferem, principalmente, na decisão sobre qual funcionalidade pertence ao núcleo e qual é deixada para os processos servidores que podem ser carregados dinamicamente para execução.

Monolítico: núcleo é um bloco maciço – ele executa todas as funções básicas do sistema operacional e ocupa alguns megabytes de código e dados – e que não é diferenciado – ele é codificado de maneira não modular. É difícil alterar qualquer componente de software individual para adaptá-lo aos requisitos variáveis. Ex: Sistemas baseados em UNIX

Micronúcleo: fornece apenas as abstrações mais básicas (espaços de endereçamento, threads e comunicação local entre processos); todos os demais serviços são fornecidos por servidores carregados dinamicamente, apenas nos computadores do sistema distribuído que necessitam deles.

Figure 7.15
Monolithic kernel and microkernel

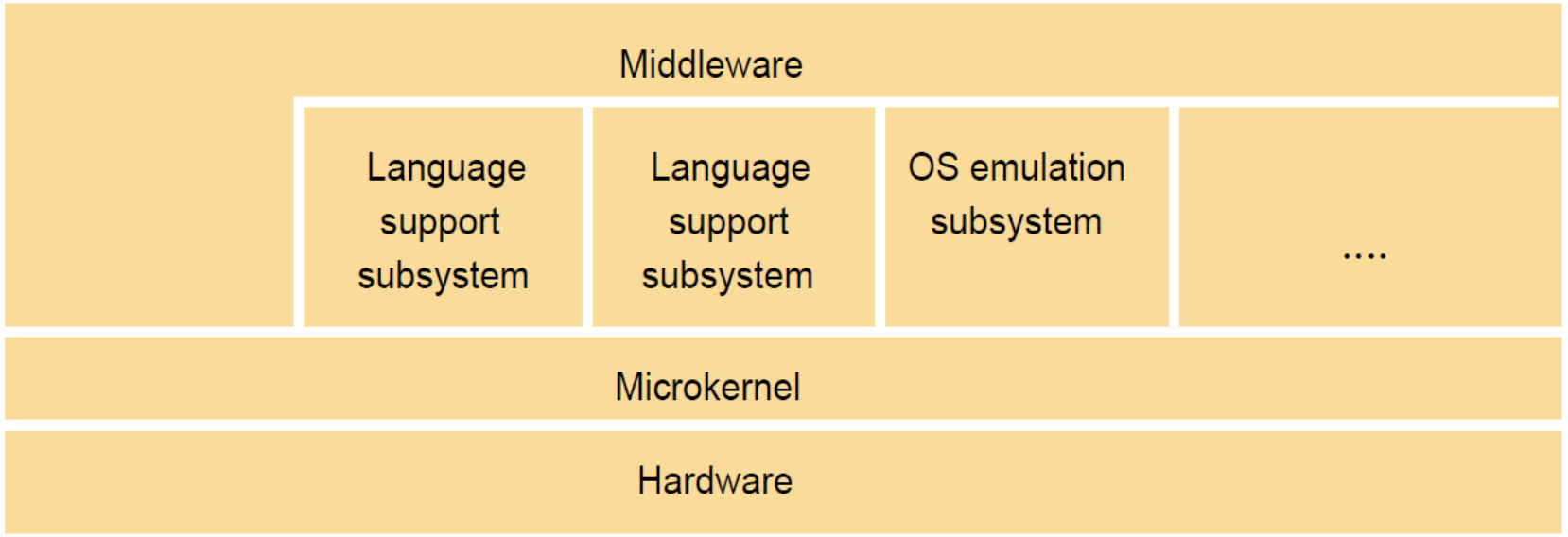


7.6 Arquiteturas de Sistemas Operacionais
micronúcleos (microkernel)

O micronúcleo aparece como uma camada entre a camada de hardware e a camada composta pelos principais componentes de sistema, chamados de subsistemas.

- Se o desempenho for o principal objetivo, em vez da portabilidade, o middleware pode usar diretamente os recursos do micronúcleo.
- Caso contrário, ele utilizará um subsistema de suporte para uma linguagem em tempo de execução, ou uma interface de sistema operacional de nível mais alto, fornecida por um subsistema de emulação do sistema operacional.
- Pode haver mais de uma interface de chamada de sistema – mais de um “sistema operacional” – apresentada ao programador em uma mesma plataforma.
- emulação do sistema operacional é diferente de máquinas virtuais.

Figure 7.16
The role of the microkernel



The microkernel supports middleware via subsystems

7.7 Virtualização em nível de S.Operacional

O objetivo da virtualização é fornecer várias máquinas virtuais (imagens virtuais do hardware) sobre a arquitetura de máquina física subjacente, com cada máquina virtual executando uma instância separada do sistema operacional.

- A virtualização é implementada por uma camada fina de software sobre a arquitetura de máquina física subjacente; essa camada é chamada de monitor de máquina virtual ou hipervisor.
- Esse monitor de máquina virtual fornece uma interface rigorosamente baseada na arquitetura física subjacente. Mais precisamente, na virtualização total, ou completa, o monitor de máquina virtual oferece uma interface idêntica à arquitetura fí- sica subjacente. Isso traz a vantagem de os sistemas operacionais existentes poderem ser executados de forma transparente e sem modificação no monitor de máquina virtual.

Figure 7.17
The architecture of Xen