Capítulo 3

Arquiteturas de SOs

Embora a definição de níveis de privilégio (Seção 2.3) imponha uma estruturação mínima a um sistema operacional, forçando a separação entre o núcleo e o espaço de usuário, os vários elementos que compõem o sistema podem ser organizados de diversas formas, separando suas funcionalidades e modularizando seu projeto. Neste capítulo serão apresentadas as arquiteturas mais populares para a organização de sistemas operacionais.

3.1 Sistemas monolíticos

Em um sistema monolítico[1], o sistema operacional é um "bloco maciço" de código que opera em modo núcleo, com acesso a todos os recursos do hardware e sem restrições de acesso à memória. Por isso, os componentes internos do sistema operacional podem se relacionar entre si conforme suas necessidades. A Figura 3.1 ilustra essa arquitetura.

A grande vantagem da arquitetura monolítica é seu desempenho: qualquer componente do núcleo pode acessar os demais componentes, áreas de memória ou mesmo dispositivos periféricos diretamente, pois não há barreiras impedindo esses acessos. A interação direta entre componentes leva a sistemas mais rápidos e compactos, pois não há necessidade de utilizar mecanismos específicos de comunicação entre os componentes do núcleo.

Todavia, a arquitetura monolítica pode levar a problemas de robustez do sistema. Como todos os componentes do SO têm acesso privilegiado ao hardware, caso um componente perca o controle devido a algum erro (acessando um ponteiro inválido ou uma posição inexistente em um vetor, por exemplo), esse erro pode se propagar rapidamente por todo o núcleo, levando o sistema ao colapso (travamento, reinicialização ou funcionamento errático).

Outro problema relacionado às arquiteturas monolíticas diz respeito ao processo de desenvolvimento. Como os componentes do sistema têm acesso direto uns aos outros, podem ser fortemente interdependentes, tornando a manutenção e evolução do núcleo mais complexas. Como as dependências e pontos de interação entre os componentes podem ser pouco evidentes, pequenas alterações na estrutura de dados de um componente podem ter um impacto inesperado em outros componentes, caso estes acessem aquela estrutura diretamente.

[1] A palavra "monólito" vem do grego *monos* (único ou unitário) e *lithos* (pedra).

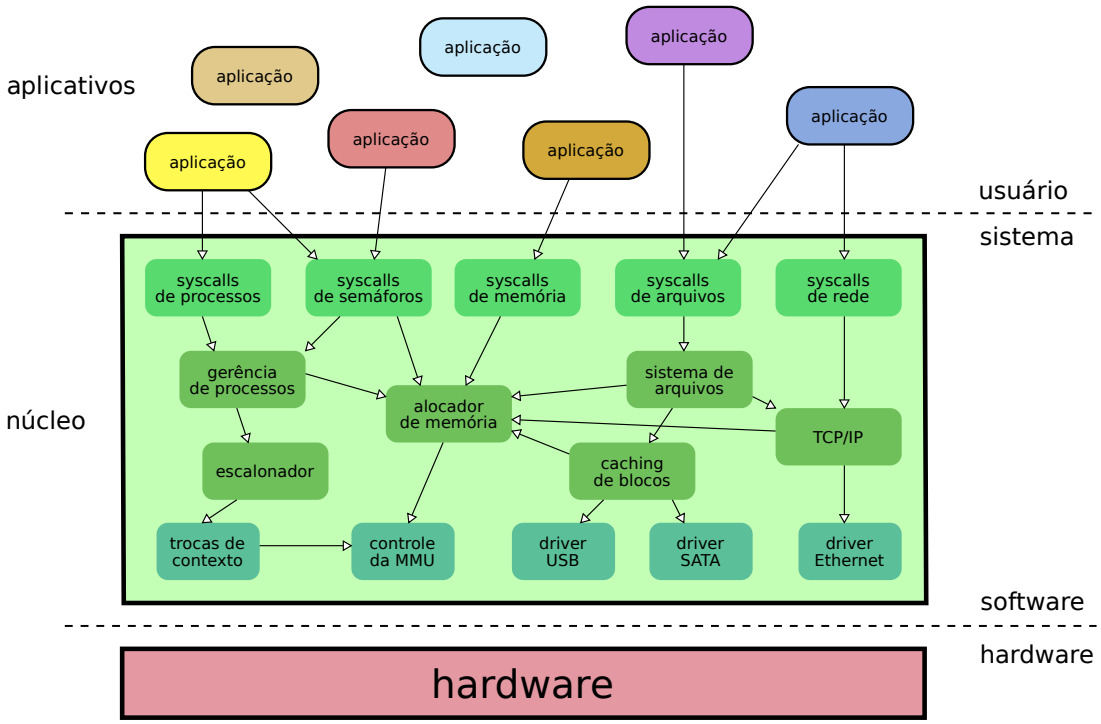

Figura 3.1: Núcleo de sistema operacional monolítico.

A arquitetura monolítica foi a primeira forma de organizar os sistemas operacionais; sistemas UNIX antigos e o MS-DOS seguiam esse modelo. O núcleo do Linux é monolítico, mas seu código vem sendo gradativamente estruturado e modularizado desde a versão 2.0 (embora boa parte do código ainda permaneça no nível de núcleo). O sistema FreeBSD [McKusick and Neville-Neil, 2005] também usa um núcleo monolítico.

3.2 Sistemas micronúcleo

Outra possibilidade de estruturar o SO consiste em retirar do núcleo todo o código de alto nível, normalmente associado às abstrações de recursos, deixando no núcleo somente o código de baixo nível necessário para interagir com o hardware e criar algumas abstrações básicas. O restante do código seria transferido para programas separados no espaço de usuário, denominados *serviços*. Por fazer o núcleo de sistema ficar menor, essa abordagem foi denominada *micronúcleo* (ou *μ-kernel*).

A abordagem micronúcleo oferece maior modularidade, pois cada serviço pode ser desenvolvido de forma independente dos demais; mais flexibilidade, pois os serviços podem ser carregados e desativados conforme a necessidade; e mais robustez, pois caso um serviço falhe, somente ele será afetado, devido ao confinamento de memória entre os serviços.

Um micronúcleo normalmente implementa somente a noção de tarefa, os espaços de memória protegidos para cada aplicação, a comunicação entre tarefas e as operações de acesso às portas de entrada/saída (para acessar os dispositivos). Todos os aspectos de alto nível, como políticas de uso do processador e da memória, o sistema de arquivos, o controle de acesso aos recursos e até mesmos os *drivers* são implementados fora do núcleo, em processos que se comunicam usando o mecanismo de comunicação provido pelo núcleo.

Um bom exemplo de sistema micronúcleo é o Minix 3 [Herder et al., 2006]. Neste sistema, o núcleo oferece funcionalidades básicas de gestão de interrupções, configuração da CPU e da MMU, acesso às portas de entrada/saída e primitivas de troca de mensagens entre aplicações. Todas as demais funcionalidades, como a gestão de arquivos e de memória, protocolos de rede, políticas de escalonamento, etc., são providas por processos fora do núcleo, denominados *servidores*. A Figura 3.2 apresenta a arquitetura do Minix 3:

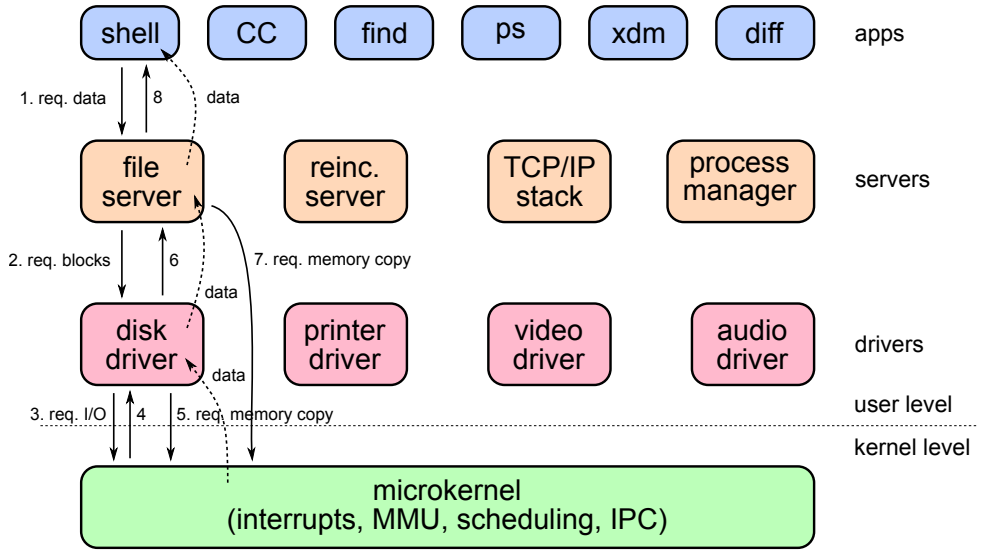

Figura 3.2: Visão geral da arquitetura do MINIX 3 (adaptado de [Herder et al., 2006]).

Por exemplo, no sistema Minix 3 a seguinte sequência de ações ocorre quando uma aplicação deseja ler dados de um arquivo no disco:

1. usando as primitivas de mensagens do núcleo, a aplicação envia uma mensagem (m_1) ao servidor de arquivos, solicitando a leitura de dados de um arquivo;

2. o servidor de arquivos verifica se possui os dados em seu cache local; se não os possuir, envia uma mensagem (m_2) ao *driver* de disco solicitando a leitura dos dados do disco em seu espaço de memória;

3. o *driver* de disco envia uma mensagem (m_3) ao núcleo solicitando operações nas portas de entrada/saída do controlador de disco, para ler dados do mesmo;

4. o núcleo verifica se o *driver* de disco tem permissão para usar as portas de entrada/saída e agenda a operação solicitada;

5. quando o disco concluir a operação, o núcleo transfere os dados lidos para a memória do *driver* de disco e responde (m_4) a este;

6. o *driver* de disco solicita (m_5) então ao núcleo a cópia dos dados recebidos para a memória do servidor de arquivos e responde (m_6) à mensagem anterior deste, informando a conclusão da operação;

7. o servidor de arquivos solicita (m_7) ao núcleo a cópia dos dados recebidos para a memória da aplicação e responde (m_8) à mensagem anterior desta;

8. a aplicação recebe a resposta de sua solicitação e usa os dados lidos.

Da sequência acima pode-se perceber que foram necessárias 8 mensagens (m_i) para realizar uma leitura de dados do disco, cada uma correspondendo a uma chamada de sistema. Cada chamada de sistema tem um custo elevado, pois implica na mudança do fluxo de execução e reconfiguração da memória acessível pela MMU. Além disso, foram necessárias várias cópias de dados entre as áreas de memória de cada entidade. Esses fatores levam a um desempenho bem inferior ao da abordagem monolítica, razão que dificulta a adoção plena dessa abordagem.

O micronúcleos vem sendo investigados desde os anos 1980. Dois exemplos clássicos dessa abordagem são os sistemas Mach [Rashid et al., 1989] e Chorus [Rozier and Martins, 1987]. Os melhores exemplos de micronúcleos bem sucedidos são provavelmente o Minix 3 [Herder et al., 2006] e o QNX. Contudo, vários sistemas operacionais atuais adotam parcialmente essa estruturação, adotando núcleos híbridos, como por exemplo o MacOS X da Apple, o Digital UNIX e o Windows NT.

3.3 Sistemas em camadas

Uma forma mais elegante de estruturar um sistema operacional faz uso da noção de camadas: a camada mais baixa realiza a interface com o hardware, enquanto as camadas intermediárias proveem níveis de abstração e gerência cada vez mais sofisticados. Por fim, a camada superior define a interface do núcleo para as aplicações (as chamadas de sistema). As camadas têm níveis de privilégio decrescentes: a camada inferior tem acesso total ao hardware, enquanto a superior tem acesso bem mais restrito (vide Seção 2.2.3).

A abordagem de estruturação de software em camadas teve sucesso no domínio das redes de computadores, através do modelo de referência OSI (*Open Systems Interconnection*) [Day, 1983], e também seria de se esperar sua adoção no domínio dos sistemas operacionais. No entanto, alguns inconvenientes limitam a aplicação do modelo em camadas de forma intensiva nos sistemas operacionais:

- O empilhamento de várias camadas de software faz com que cada pedido de uma aplicação demore mais tempo para chegar até o dispositivo periférico ou recurso a ser acessado, prejudicando o desempenho.

- Nem sempre a divisão de funcionalidades do sistema em camadas é óbvia, pois muitas dessas funcionalidades são interdependentes e não teriam como ser organizadas em camadas. Por exemplo, a gestão de entrada/saída necessita de serviços de memória para alocar/liberar *buffers* para leitura e escrita de dados, mas a gestão de memória precisa da gestão de entrada/saída para implementar a paginação em disco (*paging*). Qual dessas duas camadas viria antes?

Em decorrência desses inconvenientes, a estruturação em camadas é apenas parcialmente adotada hoje em dia. Como exemplo de sistema fortemente estruturado em camadas pode ser citado o MULTICS [Corbató and Vyssotsky, 1965].

Muitos sistemas mais recentes implementam uma camada inferior de abstração do hardware para interagir com os dispositivos (a camada *HAL – Hardware Abstraction Layer*, implementada no Windows NT e seus sucessores), e também organizam em camadas alguns subsistemas, como a gerência de arquivos e o suporte de rede (seguindo

o modelo OSI). Essa organização parcial em camadas pode ser facilmente observada nas arquiteturas do Minix 3 (Figura 3.2) Windows 2000 (Figura 3.3) e Android (Figura 2.2).

3.4 Sistemas híbridos

Apesar das boas propriedades de modularidade, flexibilidade e robustez proporcionadas pelos micronúcleos, sua adoção não teve o sucesso esperado devido ao baixo desempenho. Uma solução encontrada para esse problema consiste em trazer de volta ao núcleo os componentes mais críticos, para obter melhor desempenho. Essa abordagem intermediária entre o núcleo monolítico e micronúcleo é denominada *núcleo híbrido*. É comum observar também nos núcleos híbridos uma influência da arquitetura em camadas.

Vários sistemas operacionais atuais empregam núcleos híbridos, entre eles o Microsoft Windows, a partir do Windows NT. As primeiras versões do Windows NT podiam ser consideradas micronúcleo, mas a partir da versão 4 vários subsistemas foram movidos para dentro do núcleo para melhorar seu desempenho, transformando-o em um núcleo híbrido. Os sistemas MacOS e iOS da Apple também adotam um núcleo híbrido chamado XNU (de *X is Not Unix*), construído a partir dos núcleos Mach (micronúcleo) [Rashid et al., 1989] e FreeBSD (monolítico) [McKusick and Neville-Neil, 2005]. A figura 3.3 mostra a arquitetura interna do núcleo usado no SO Windows 2000.

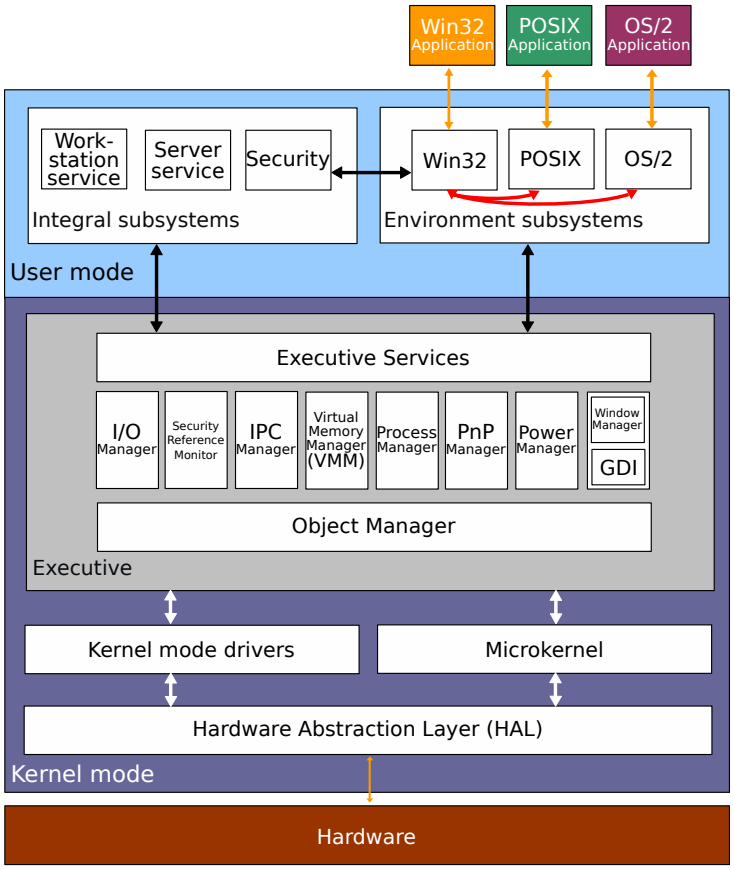

Figura 3.3: Arquitetura do sistema Windows 2000 [Wikipedia, 2018].

3.5 Arquiteturas avançadas

Além das arquiteturas clássicas (monolítica, em camadas, micronúcleo), recentemente surgiram várias propostas organizar os componentes do sistema operacional, com vistas a contextos de aplicação específicos, como nuvens computacionais. Esta seção apresenta algumas dessas novas arquiteturas.

3.5.1 Máquinas virtuais

Apesar de ser um conceito dos anos 1960 [Goldberg, 1973], a virtualização de sistemas e de aplicações ganhou um forte impulso a partir dos anos 2000, com a linguagem Java, a consolidação de servidores (juntar vários servidores com seus sistemas operacionais em um único computador físico) e, mais recentemente, a computação em nuvem.

Uma máquina virtual é uma camada de software que "transforma" um sistema em outro, ou seja, que usa os serviços fornecidos por um sistema operacional (ou pelo hardware) para construir a interface de outro sistema. Por exemplo, o ambiente (JVM - *Java Virtual Machine*) usa os serviços de um sistema operacional convencional (Windows ou Linux) para construir um computador virtual que processa *bytecode*, o código binário dos programas Java. Da mesma forma, o ambiente *VMWare* permite executar o sistema Windows sobre um sistema Linux subjacente, ou vice-versa.

Um ambiente de máquinas virtuais consiste de três partes básicas, que podem ser observadas nos exemplos da Figura 3.4:

- O sistema real, ou sistema hospedeiro (*host*), que contém os recursos reais de hardware e software do sistema;
- a camada de virtualização, denominada *hipervisor* ou *monitor de virtualização* (*VMM - Virtual Machine Monitor*), que constrói a máquina virtual a partir dos recursos do sistema real;
- o sistema virtual, ou sistema convidado (*guest*); em alguns casos, vários sistemas virtuais podem coexistir, executando sobre o mesmo sistema real.

Existem diversas possibilidades de uso da virtualização, levando a vários tipos de hipervisor, por exemplo:

Hipervisor de aplicação: suporta a execução de uma única aplicação em uma linguagem específica. As máquinas virtuais JVM (*Java Virtual Machine*) e CLR (*Common Language Runtime*, para C#) são exemplos típicos dessa abordagem.

Hipervisor de sistema: suporta a execução de um sistema operacional convidado com suas aplicações, como nos dois exemplos da Figura 3.4. Esta categoria se subdivide em:

Hipervisor nativo: executa diretamente sobre o hardware, virtualizando os recursos deste para construir máquinas virtuais, onde executam vários sistemas operacionais com suas respectivas aplicações. O exemplo da esquerda na Figura 3.4 ilustra esse conceito. Um exemplo deste tipo de hipervisor é o ambiente *VMware ESXi Server* [Newman et al., 2005].

Hipervisor convidado: executa sobre um sistema operacional hospedeiro, como no exemplo da direita na Figura 3.4. Os hipervisores *VMWare Workstation* e *VirtualBox* implementam essa abordagem.

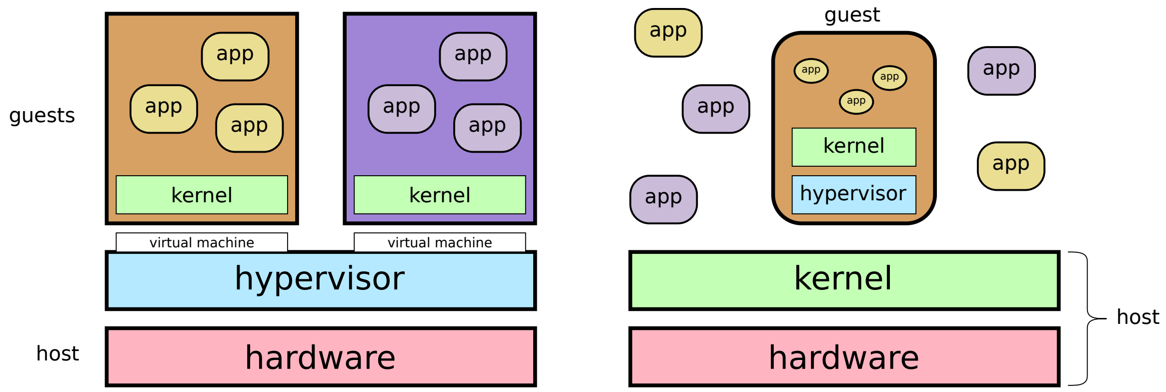

Figura 3.4: Ambientes de virtualização nativo (esq.) e convidado (dir.).

Por permitir a execução de vários sistemas independentes e isolados entre si sobre o mesmo hardware, hipervisores de sistema são muito usados em ambientes corporativos de larga escala, como as nuvens computacionais. Hipervisores de aplicação também são muito usados, por permitir a execução de código independente de plataforma, como em Java e C#. Avanços recentes no suporte de hardware para a virtualização e na estrutura interna dos hipervisores permitem a execução de máquinas virtuais com baixo custo adicional em relação à execução nativa.

O capítulo 31.4 deste livro contém uma apresentação mais detalhada do conceito de virtualização e de suas possibilidades de uso.

3.5.2 Contêineres

Além da virtualização, outra forma de isolar aplicações ou subsistemas em um sistema operacional consiste na virtualização do espaço de usuário (*userspace*). Nesta abordagem, denominada *servidores virtuais* ou *contêineres*, o espaço de usuário do sistema operacional é dividido em áreas isoladas denominadas *domínios* ou *contêineres*. A cada domínio é alocada uma parcela dos recursos do sistema operacional, como memória, tempo de processador e espaço em disco.

Para garantir o isolamento entre os domínios, cada domínio tem sua própria interface de rede virtual e, portanto, seu próprio endereço de rede. Além disso, na maioria das implementações cada domínio tem seu próprio espaço de nomes para os identificadores de usuários, processos e primitivas de comunicação. Assim, é possível encontrar um usuário `pedro` no domínio d_3 e outro usuário `pedro` no domínio d_7, sem relação entre eles nem conflitos. Essa noção de espaços de nomes distintos se estende aos demais recursos do sistema: identificadores de processos, semáforos, árvores de diretórios, etc. Entretanto, o núcleo do sistema operacional é o mesmo para todos os domínios.

Processos em um mesmo domínio podem interagir entre si normalmente, mas processos em domínios distintos não podem ver ou interagir entre si. Além disso, processos não podem trocar de domínio nem acessar recursos de outros domínios. Dessa forma, os domínios são vistos como sistemas distintos, acessíveis somente através de

seus endereços de rede. Para fins de gerência, normalmente é definido um domínio d_0, chamado de *domínio inicial*, *privilegiado* ou *de gerência*, cujos processos têm visibilidade e acesso aos processos e recursos dos demais domínios.

A Figura 3.5 mostra a estrutura típica de um ambiente de contêineres. Nela, pode-se observar que um processo pode migrar de d_0 para d_1, mas que os processos em d_1 não podem migrar para outros domínios. A comunicação entre processos confinados em domínios distintos (d_2 e d_3) também é proibida.

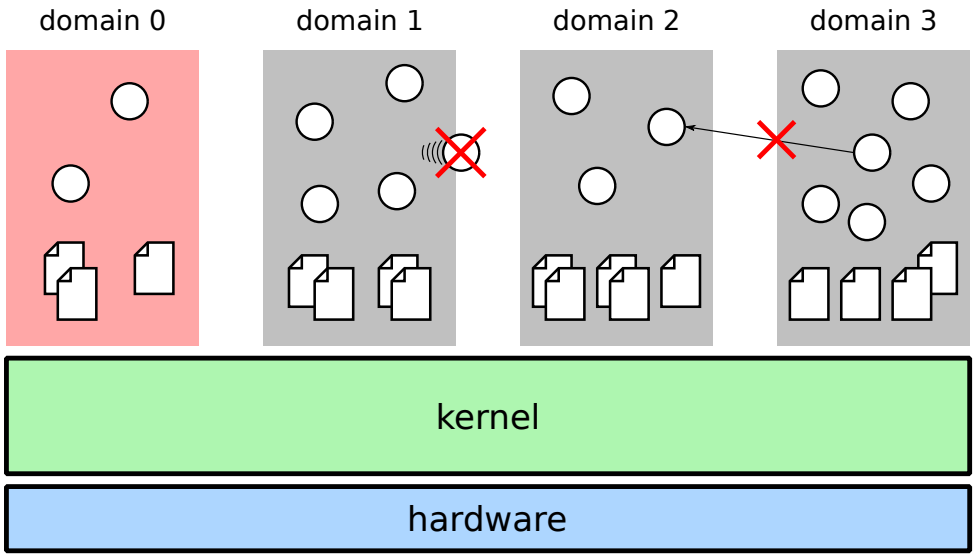

Figura 3.5: Sistema com contêineres (domínios).

Existem implementações de contêineres, como as *Jails* do sistema FreeBSD [McKusick and Neville-Neil, 2005] ou as *zonas* do sistema Solaris [Price and Tucker, 2004], que oferecem o isolamento entre domínios e o controle de uso dos recursos pelos mesmos. O núcleo Linux oferece diversos mecanismos para o isolamento de espaços de recursos, que são usados por plataformas de gerenciamento de contêineres como *Linux Containers* (LXC), *Docker* e *Kubernetes*.

3.5.3 Sistemas exonúcleo

Em um sistema convencional, as aplicações executam sobre uma pilha de serviços de abstração e gerência de recursos providos pelo sistema operacional. Esses serviços facilitam a construção de aplicações, mas podem constituir uma sobrecarga significativa para o desempenho do sistema. Para cada abstração ou serviço utilizado é necessário interagir com o núcleo através de chamadas de sistema, que impactam negativamente o desempenho. Além disso, os serviços providos pelo núcleo devem ser genéricos o suficiente para atender a uma vasta variedade de aplicações. Aplicações com demandas muito específicas, como um servidor Web de alto desempenho, nem sempre são atendidas da melhor forma possível.

Os exonúcleos (*exokernels*) [Engler, 1998] tentam trazer uma resposta a essas questões, reduzindo ainda mais os serviços oferecidos pelo núcleo. Em um sistema exonúcleo, o núcleo do sistema apenas proporciona acesso controlado aos recursos do hardware, mas não implementa nenhuma abstração. Por exemplo, o núcleo provê acesso compartilhado à interface de rede, mas não implementa nenhum protocolo.

Todas as abstrações e funcionalidades de gerência que uma aplicação precisa terão de ser implementadas pela própria aplicação, em seu espaço de memória.

Para simplificar a construção de aplicações sobre exonúcleos, são usados sistemas operacionais "de biblioteca" ou *LibOS* (*Library Operating System*), que implementam as abstrações usuais, como memória virtual, sistemas de arquivos, semáforos e protocolos de rede. Um LibOS nada mais é que um conjunto de bibliotecas compartilhadas usadas pela aplicação para prover os serviços de que esta necessita. Diferentes aplicações sobre o mesmo exonúcleo podem usar LibOS diferentes, ou implementar suas próprias abstrações. A Figura 3.6 ilustra a arquitetura de um exonúcleo típico.

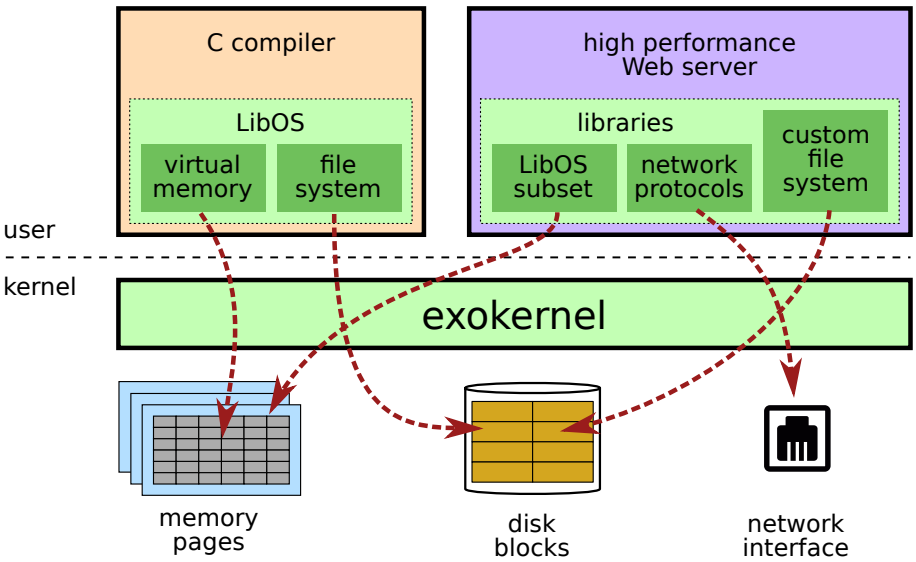

Figura 3.6: Sistema exonúcleo (adaptado de [Engler, 1998]).

É importante observar que um exonúcleo é diferente de um micronúcleo, pois neste último as abstrações são implementada por um conjunto de processos servidores independentes e isolados entre si, enquanto no exonúcleo cada aplicação implementa suas próprias abstrações (ou as incorpora de bibliotecas compartilhadas).

Até o momento não existem exonúcleos em produtos comerciais, as implementações mais conhecidas são esforços de projetos de pesquisa, como Aegis/ExOS [Engler, 1998] e Nemesis [Hand, 1999].

3.5.4 Sistemas uninúcleo

Nos sistemas uninúcleo (*unikernel*) [Madhavapeddy and Scott, 2013], um núcleo de sistema operacional, as bibliotecas e uma aplicação são compilados e ligados entre si, formando um bloco monolítico de código, que executa em um único espaço de endereçamento, em modo privilegiado. Dessa forma, o custo da transição aplicação/núcleo nas chamadas de sistema diminui muito, gerando um forte ganho de desempenho.

Outro ponto positivo da abordagem uninúcleo é o fato de incluir no código final somente os componentes necessários para suportar a aplicação-alvo e os *drivers* necessários para acessar o hardware-alvo, levando a um sistema compacto, que pode ser lançado rapidamente.

A estrutura de uninúcleo se mostra adequada para computadores que executam uma única aplicação, como servidores de rede (HTTP, DNS, DHCP) e serviços em

ambientes de nuvem computacional (bancos de dados, Web Services). De fato, o maior interesse em usar uninúcleos vem do ambiente de nuvem, pois os "computadores" da nuvem são na verdade máquinas virtuais com hardware virtual simples (com *drivers* padronizados) e que executam serviços isolados (vide Seção 3.5.1).

Os sistemas OSv [Kivity et al., 2014] e MirageOS [Madhavapeddy and Scott, 2013] são bons exemplos de uninúcleos. Como eles foram especialmente concebidos para as nuvens, são também conhecidos como "CloudOS". A Figura 3.7 mostra a arquitetura desses sistemas.

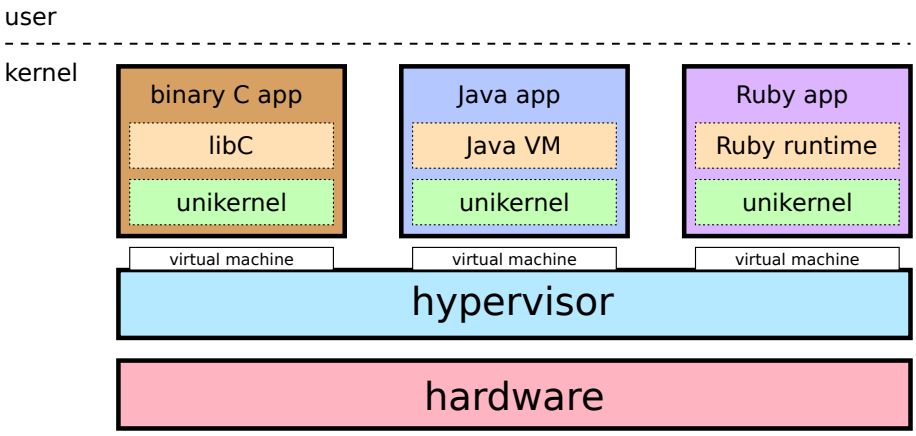

Figura 3.7: Sistema uninúcleo.

É interessante observar que, em sistemas embarcados e nas chamadas *appliances* de rede (roteadores, modems, etc), é usual compilar o núcleo, bibliotecas e aplicação em um único arquivo binário que é em seguida executado diretamente sobre o hardware, em modo privilegiado. Este é o caso, por exemplo, do sistema TinyOS [Levis et al., 2005], voltado para aplicações de Internet das Coisas (IoT - *Internet of Things*). Apesar desse sistema não ser oficialmente denominado um uninúcleo, a abordagem basicamente é a mesma.

Exercícios

1. Monte uma tabela com os benefícios e deficiências mais relevantes das principais arquiteturas de sistemas operacionais.

2. O Linux possui um núcleo similar com o da figura 3.1, mas também possui "tarefas de núcleo" que executam como os gerentes da figura 3.2. Seu núcleo é monolítico ou micronúcleo? Por quê?

3. Sobre as afirmações a seguir, relativas às diversas arquiteturas de sistemas operacionais, indique quais são incorretas, justificando sua resposta:

   (a) Uma máquina virtual de sistema é construída para suportar uma aplicação escrita em uma linguagem de programação específica, como Java.

   (b) Um hipervisor convidado executa sobre um sistema operacional hospedeiro.

   (c) Em um sistema operacional micronúcleo, os diversos componentes do sistema são construídos como módulos interconectados executando dentro do núcleo.

(d) Núcleos monolíticos são muito utilizados devido à sua robustez e facilidade de manutenção.

(e) Em um sistema operacional micronúcleo, as chamadas de sistema são implementadas através de trocas de mensagens.

Atividades

1. O sistema operacional Minix 3 usa uma arquitetura micronúcleo. Ele pode ser obtido gratuitamente no site `http://www.minix3.org`. Instale-o em uma máquina virtual e explore seus processos, usando os comandos `top` e `ps`. Identifique os principais processos em execução, usando a documentação do site.

Referências

F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the Multics system. In *AFIPS Conference Proceedings*, pages 185–196, 1965.

J. Day. The OSI reference model. *Proceedings of the IEEE*, December 1983.

D. Engler. *The Exokernel Operating System Architecture*. PhD thesis, Massachusetts Institute of Technology, Cambridge - MA, October 1998.

R. Goldberg. Architecture of virtual machines. In *AFIPS National Computer Conference*, 1973.

S. Hand. Self-paging in the Nemesis operating system. In *3rd Symposium on Operating Systems Design and Implementation (OSDI)*, volume 99, pages 73–86, 1999.

J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40 (3), Sept. 2006.

A. Kivity, D. L. G. Costa, and P. Enberg. OSv – optimizing the operating system for virtual machines. In *USENIX Annual Technical Conference*, page 61, 2014.

P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. TinyOS: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

A. Madhavapeddy and D. J. Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30:30–30:44, Dec. 2013. ISSN 1542-7730.

M. McKusick and G. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2005.

M. Newman, C.-M. Wiberg, and B. Braswell. *Server Consolidation with VMware ESX Server*. IBM RedBooks, 2005. `http://www.redbooks.ibm.com`.

D. Price and A. Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *18th USENIX conference on System administration*, pages 241–254, 2004.

R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. B. Jones. Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON*, pages 176–178, San Francisco, CA, USA, 1989. IEEE Comput. Soc. Press.

M. Rozier and J. L. Martins. The Chorus distributed operating system: Some design issues. In Y. Paker, J.-P. Banatre, and M. Bozyiğit, editors, *Distributed Operating Systems*, pages 261–287, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

Wikipedia. Wikipedia online enciclopedia. http://www.wikipedia.org, 2018.