um processo ser criado, o criador e criatura são iguais. WaitForSingleObject é usado para esperar por um evento. É possível se esperar por muitos eventos com essa chamada. Se o parâmetro especifica um processo, então quem chamou espera pelo processo especificado terminar, o que é feito usando ExitProcess.

As próximas seis chamadas operam em arquivos e são funcionalmente similares a suas correspondentes do UNIX, embora difiram nos parâmetros e detalhes. Ainda assim, os arquivos podem ser abertos, fechados, lidos e escritos de uma maneira bastante semelhante ao UNIX. As chamadas SetFilePointer e GetFileAttributesEx estabelecem a posição do arquivo e obtêm alguns de seus atributos.

O Windows tem diretórios e eles são criados com chamadas API CreateDirectory e RemoveDirectory, respectivamente. Há também uma noção de diretório atual, determinada por SetCurrentDirectory. A hora atual do dia é conseguida usando GetLocalTime.

A interface Win32 não tem links para os arquivos, tampouco sistemas de arquivos montados, segurança ou sinais, de maneira que não existem as chamadas correspondentes ao UNIX. É claro, Win32 tem um número enorme de outras chamadas que o UNIX não tem, em especial para gerenciar a interface gráfica GUI. O Windows Vista tem um sistema de segurança elaborado e também dá suporte a links de arquivos. Os Windows 7 e 8 acrescentam ainda mais aspectos e chamadas de sistema.

Uma última nota a respeito do Win32 talvez valha a pena ser feita. O Win32 não é uma interface realmente uniforme ou consistente. A principal culpada aqui foi a necessidade de ser retroativamente compatível com a interface anterior de 16 bits usada no Windows 3.x.

1.7 Estrutura de sistemas operacionais

Agora que vimos como os sistemas operacionais parecem por fora (isto é, a interface do programador), é hora de darmos uma olhada por dentro. Nas seções a seguir, examinaremos seis estruturas diferentes que foram tentadas, a fim de termos alguma ideia do espectro de possibilidades. Isso não quer dizer que esgotaremos o assunto, mas elas dão uma ideia de alguns projetos que foram tentados na prática. Os seis projetos que discutiremos aqui são sistemas monolíticos, sistemas de camadas, micronúcleos, sistemas cliente-servidor, máquinas virtuais e exonúcleos.

FIGURA 1.23 As chamadas da API Win32 que correspondem aproximadamente às chamadas UNIX da Figura 1.18. Vale a pena enfatizar que o Windows tem um número muito grande de outras chamadas de sistema, a maioria das quais não corresponde a nada no UNIX.

UNIX	Win32	Descrição
fork	CreateProcess	Cria um novo processo
waitpid	WaitForSingleObject	Pode esperar que um processo termine
execve	(nenhuma)	CreateProcess = fork + execve
exit	ExitProcess	Conclui a execução
open	CreateFile	Cria um arquivo ou abre um arquivo existente
close	CloseHandle	Fecha um arquivo
read	ReadFile	Lê dados a partir de um arquivo
write	WriteFile	Escreve dados em um arquivo
lseek	SetFilePointer	Move o ponteiro do arquivo
stat	GetFileAttributesEx	Obtém vários atributos do arquivo
mkdir	CreateDirectory	Cria um novo diretório
rmdir	RemoveDirectory	Remove um diretório vazio
link	(nenhuma)	Win32 não dá suporte a ligações
unlink	DeleteFile	Destrói um arquivo existente
mount	(nenhuma)	Win32 não dá suporte a mount
umount	(nenhuma)	Win32 não dá suporte a mount
chdir	SetCurrentDirectory	Altera o diretório de trabalho atual
chmod	(nenhuma)	Win32 não dá suporte a segurança (embora o NT suporte)
kill	(nenhuma)	Win32 não dá suporte a sinais
time	GetLocalTime	Obtém o tempo atual

1.7.1 Sistemas monolíticos

De longe a organização mais comum, na abordagem monolítica todo o sistema operacional é executado como um único programa em modo núcleo. O sistema operacional é escrito como uma coleção de rotinas, ligadas a um único grande programa binário executável. Quando a técnica é usada, cada procedimento no sistema é livre para chamar qualquer outro, se este oferecer alguma computação útil de que o primeiro precisa. Ser capaz de chamar qualquer procedimento que você quer é muito eficiente, mas ter milhares de procedimentos que podem chamar um ao outro sem restrições pode também levar a um sistema difícil de lidar e compreender. Também, uma quebra em qualquer uma dessas rotinas derrubará todo o sistema operacional.

Para construir o programa objeto real do sistema operacional quando essa abordagem é usada, é preciso primeiro compilar todas as rotinas individuais (ou os arquivos contendo as rotinas) e então juntá-las em um único arquivo executável usando o ligador (*linker*) do sistema. Em termos de ocultação de informações, essencialmente não há nenhuma — toda rotina é visível para toda outra rotina (em oposição a uma estrutura contendo módulos ou pacotes, na qual grande parte da informação é escondida dentro de módulos, e apenas os pontos de entrada oficialmente designados podem ser chamados de fora do módulo).

Mesmo em sistemas monolíticos, no entanto, é possível se ter alguma estrutura. Os serviços (chamadas de sistema) providos pelo sistema operacional são requisitados colocando-se os parâmetros em um local bem definido (por exemplo, em uma pilha) e então executando uma instrução de desvio de controle (*trap*). Essa instrução chaveia a máquina do modo usuário para o modo núcleo e transfere o controle para o sistema operacional, mostrado no passo 6 na Figura 1.17. O sistema operacional então busca os parâmetros e determina qual chamada de sistema será executada. Depois disso, ele indexa uma tabela que contém na linha *k* um ponteiro para a rotina que executa a chamada de sistema *k* (passo 7 na Figura 1.17).

Essa organização sugere uma estrutura básica para o sistema operacional:

1. Um programa principal que invoca a rotina de serviço requisitada.
2. Um conjunto de rotinas de serviço que executam as chamadas de sistema.
3. Um conjunto de rotinas utilitárias que ajudam as rotinas de serviço.

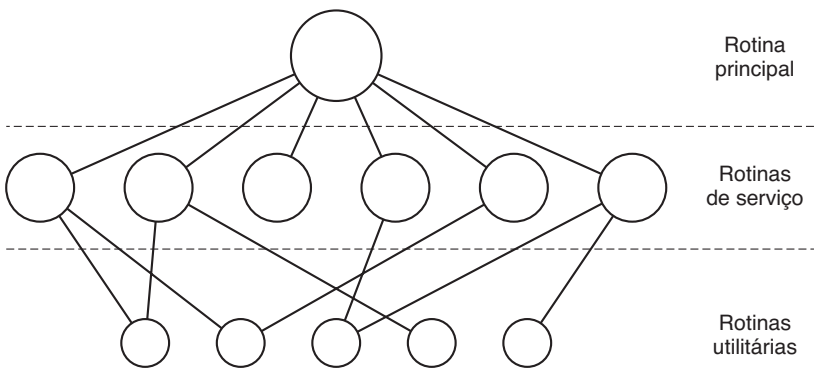
Nesse modelo, para cada chamada de sistema há uma rotina de serviço que se encarrega dela e a executa. As rotinas utilitárias fazem coisas que são necessárias para várias rotinas de serviços, como buscar dados de programas dos usuários. Essa divisão em três camadas é mostrada na Figura 1.24.

Além do sistema operacional principal que é carregado quando o computador é inicializado, muitos sistemas operacionais dão suporte a extensões carregáveis, como drivers de dispositivos de E/S e sistemas de arquivos. Esses componentes são carregados conforme a demanda. No UNIX eles são chamados de **bibliotecas compartilhadas**. No Windows são chamados de **DLLs (Dynamic Link Libraries** — bibliotecas de ligação dinâmica). Eles têm a extensão de arquivo *.dll* e o diretório *C:\Windows\system32* nos sistemas Windows tem mais de 1.000 deles.

1.7.2 Sistemas de camadas

Uma generalização da abordagem da Figura 1.24 é organizar o sistema operacional como uma hierarquia de camadas, cada uma construída sobre a camada abaixo dela.

FIGURA 1.24 Um modelo de estruturação simples para um sistema monolítico.



O primeiro sistema construído dessa maneira foi o sistema THE desenvolvido na Technische Hogeschool Eindhoven na Holanda por E. W. Dijkstra (1968) e seus estudantes. O sistema THE era um sistema em lote simples para um computador holandês, o Electrologica X8, que tinha 32 K de palavras de 27 bits (bits eram caros na época).

O sistema tinha seis camadas, com mostrado na Figura 1.25. A camada 0 lidava com a alocação do processador, realizando o chaveamento de processos quando ocorriam interrupções ou quando os temporizadores expiravam. Acima da camada 0, o sistema consistia em processos sequenciais e cada um deles podia ser programado sem precisar preocupar-se com o fato de que múltiplos processos estavam sendo executados em um único processador. Em outras palavras, a camada 0 fornecia a multiprogramação básica da CPU.

A camada 1 realizava o gerenciamento de memória. Ela alocava espaço para processos na memória principal e em um tambor magnético de 512 K palavras usado para armazenar partes de processos (páginas) para as quais não havia espaço na memória principal. Acima da camada 1, os processos não precisavam se preocupar se eles estavam na memória ou no tambor magnético; o software da camada 1 certificava-se de que as páginas fossem trazidas à memória no momento em que eram necessárias e removidas quando não eram mais.

A camada 2 encarregava-se da comunicação entre cada processo e o console de operação (isto é, o usuário). Acima dessa camada cada processo efetivamente tinha o seu próprio console de operação. A camada 3 encarregava-se do gerenciamento dos dispositivos de E/S e armazenava temporariamente os fluxos de informação que iam ou vinham desses dispositivos. Acima da camada 3, cada processo podia lidar com dispositivos de E/S abstratos mais acessíveis, em vez de dispositivos reais com muitas peculiaridades. A camada 4 era onde os programas dos usuários eram encontrados. Eles não precisavam se preocupar com o gerenciamento de processo, memória, console ou E/S. O processo operador do sistema estava localizado na camada 5.

FIGURA 1.25 Estrutura do sistema operacional THE.

Camada	Função
5	O operador
4	Programas de usuário
3	Gerenciamento de entrada/saída
2	Comunicação operador–processo
1	Memória e gerenciamento de tambor
0	Alocação do processador e multiprogramação

Outra generalização do conceito de camadas estava presente no sistema MULTICS. Em vez de camadas, MULTICS foi descrito como tendo uma série de anéis concêntricos, com os anéis internos sendo mais privilegiados do que os externos (o que é efetivamente a mesma coisa). Quando um procedimento em um anel exterior queria chamar um procedimento em um anel interior, ele tinha de fazer o equivalente de uma chamada de sistema, isto é, uma instrução de desvio, TRAP, cujos parâmetros eram cuidadosamente conferidos por sua validade antes de a chamada ter permissão para prosseguir. Embora todo o sistema operacional fosse parte do espaço de endereço de cada processo de usuário em MULTICS, o hardware tornou possível que se designassem rotinas individuais (segmentos de memória, na realidade) como protegidos contra leitura, escrita ou execução.

Enquanto o esquema de camadas THE era na realidade somente um suporte para o projeto, pois em última análise todas as partes do sistema estavam unidas em um único programa executável, em MULTICS, o mecanismo de anéis estava bastante presente no momento de execução e imposto pelo hardware. A vantagem do mecanismo de anéis é que ele pode ser facilmente estendido para estruturar subsistemas de usuário. Por exemplo, um professor poderia escrever um programa para testar e atribuir notas a programas de estudantes executando-o no anel n, com os programas dos estudantes seriam executados no anel $n + 1$, de maneira que eles não pudessem mudar suas notas.

1.7.3 Micronúcleos

Com a abordagem de camadas, os projetistas têm uma escolha de onde traçar o limite núcleo-usuário. Tradicionalmente, todas as camadas entram no núcleo, mas isso não é necessário. Na realidade, um forte argumento pode ser defendido para a colocação do mínimo possível no modo núcleo, pois erros no código do núcleo podem derrubar o sistema instantaneamente. Em comparação, processos de usuário podem ser configurados para ter menos poder, de maneira que um erro possa não ser fatal.

Vários pesquisadores estudaram repetidamente o número de erros por 1.000 linhas de código (por exemplo, BASILLI e PERRICONE, 1984; OSTRAND e WEYUKER, 2002). A densidade de erros depende do tamanho do módulo, idade do módulo etc., mas um número aproximado para sistemas industriais sérios fica entre dois e dez erros por mil linhas de código. Isso significa que em um sistema operacional monolítico de cinco milhões de linhas de código é provável que

contenha entre 10.000 e 50.000 erros no núcleo. Nem todos são fatais, é claro, tendo em vista que alguns erros podem ser coisas como a emissão de uma mensagem de erro incorreta em uma situação que raramente ocorre. Mesmo assim, sistemas operacionais são a tal ponto sujeitos a erros, que os fabricantes de computadores colocam botões de reinicialização neles (muitas vezes no painel da frente), algo que os fabricantes de TVs, aparelhos de som e carros não o fazem, apesar da grande quantidade de software nesses dispositivos.

A ideia básica por trás do projeto de micronúcleo é atingir uma alta confiabilidade através da divisão do sistema operacional em módulos pequenos e bem definidos, apenas um dos quais — o micronúcleo — é executado em modo núcleo e o resto é executado como processos de usuário comuns relativamente sem poder. Em particular, ao se executar cada driver de dispositivo e sistema de arquivos como um processo de usuário em separado, um erro em um deles pode derrubar esse componente, mas não consegue derrubar o sistema inteiro. Desse modo, um erro no driver de áudio fará que o som fique truncado ou pare, mas não derrubará o computador. Em comparação, em um sistema monolítico, com todos os drivers no núcleo, um driver de áudio com problemas pode facilmente referenciar um endereço de memória inválido e provocar uma parada dolorosa no sistema instantaneamente.

Muitos micronúcleos foram implementados e empregados por décadas (HAERTIG et al., 1997; HEISER et al., 2006; HERDER et al., 2006; HILDEBRAND, 1992; KIRSCH et al., 2005; LIEDTKE, 1993, 1995, 1996; PIKE et al., 1992; e ZUBERI et al., 1999). Com a exceção do OS X, que é baseado no micronúcleo Mach (ACETTA et al., 1986), sistemas operacionais de computadores de mesa comuns não usam micronúcleos. No entanto, eles são dominantes em aplicações de tempo real, industriais, de aviônica e militares, que são cruciais para missões e têm exigências de confiabilidade muito altas. Alguns dos micronúcleos mais conhecidos incluem Integrity, K42, L4, PikeOS, QNX, Symbian e MINIX 3. Daremos agora uma breve visão geral do MINIX 3, que levou a ideia da modularidade até o limite, decompondo a maior parte do sistema operacional em uma série de processos de modo usuário independentes. MINIX 3 é um sistema em conformidade com o POSIX, de código aberto e gratuitamente disponível em <*www.minix3.org*> (GIUFFRIDA et al., 2012; GIUFFRIDA et al., 2013; HERDER et al., 2006; HERDER et al., 2009; e HRUBY et al., 2013).

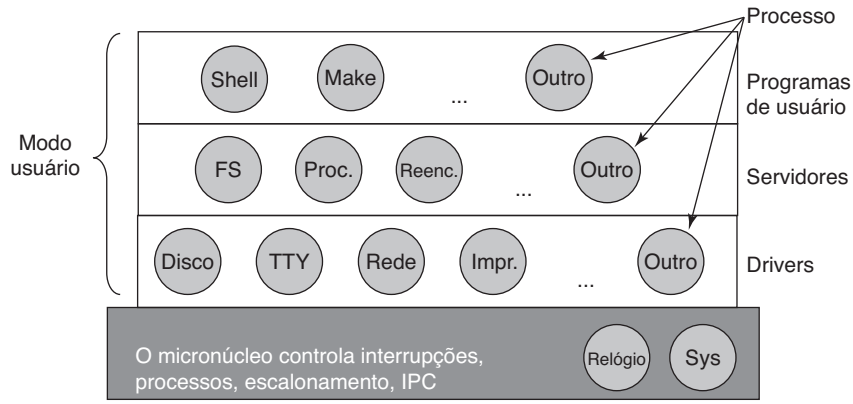
O micronúcleo MINIX 3 tem apenas em torno de 12.000 linhas de C e cerca de 1.400 linhas de assembler para funções de nível muito baixo como capturar interrupções e chavear processos. O código C gerencia e escalona processos, lida com a comunicação entre eles (passando mensagens entre processos) e oferece um conjunto de mais ou menos 40 chamadas de núcleo que permitem que o resto do sistema operacional faça o seu trabalho. Essas chamadas realizam funções como associar os tratadores às interrupções, transferir dados entre espaços de endereços e instalar mapas de memória para processos novos. A estrutura de processo de MINIX 3 é mostrada na Figura 1.26, com os tratadores de chamada de núcleo rotulados *Sys*. O driver de dispositivo para o relógio também está no núcleo, pois o escalonador interage de perto com ele. Os outros drivers de dispositivos operam como processos de usuário em separado.

Fora do núcleo, o sistema é estruturado como três camadas de processos, todos sendo executados em modo usuário. A camada mais baixa contém os drivers de dispositivos. Como são executados em modo usuário, eles não têm acesso físico ao espaço da porta de E/S e não podem emitir comandos de E/S diretamente. Em vez disso, para programar um dispositivo de E/S, o driver constrói uma estrutura dizendo quais valores escrever para quais portas de E/S e faz uma chamada de núcleo dizendo para o núcleo fazer a escrita. Essa abordagem significa que o núcleo pode conferir para ver que o driver está escrevendo (ou lendo) a partir da E/S que ele está autorizado a usar. Em consequência (e diferentemente de um projeto monolítico), um driver de áudio com erro não consegue escrever por acidente no disco.

Acima dos drivers há outra camada no modo usuário contendo os servidores, que fazem a maior parte do trabalho do sistema operacional. Um ou mais servidores de arquivos gerenciam o(s) sistema(s) de arquivos, o gerente de processos cria, destrói e gerencia processos, e assim por diante. Programas de usuários obtêm serviços de sistemas operacionais enviando mensagens curtas para os servidores solicitando as chamadas de sistema POSIX. Por exemplo, um processo precisando fazer uma `read`, envia uma mensagem para um dos servidores de arquivos dizendo a ele o que ler.

Um servidor interessante é o **servidor de reencarnação**, cujo trabalho é conferir se os outros servidores e drivers estão funcionando corretamente. No caso da detecção de um servidor ou driver defeituoso, ele é automaticamente substituído sem qualquer intervenção do usuário. Dessa maneira, o sistema está regenerando a si mesmo e pode atingir uma alta confiabilidade.

FIGURA 1.26 Estrutura simplificada do sistema MINIX.



O sistema tem muitas restrições limitando o poder de cada processo. Como mencionado, os drivers podem tocar apenas portas de E/S autorizadas, mas o acesso às chamadas de núcleo também é controlado processo a processo, assim como a capacidade de enviar mensagens para outros processos. Processos também podem conceder uma permissão limitada para outros processos para que o núcleo acesse seus espaços de endereçamento. Como exemplo, um sistema de arquivos pode conceder uma permissão para que a unidade de disco deixe o núcleo colocar uma leitura recente de um bloco do disco em um endereço específico dentro do espaço de endereço do sistema de arquivos. A soma de todas essas restrições é que cada driver e servidor têm exatamente o poder de fazer o seu trabalho e nada mais, dessa maneira limitando muito o dano que um componente com erro pode provocar.

Uma ideia de certa maneira relacionada a ter um núcleo mínimo é colocar o **mecanismo** para fazer algo no núcleo, mas não a **política**. Para esclarecer esse ponto, considere o escalonamento de processos. Um algoritmo de escalonamento relativamente simples é designar uma prioridade numérica para todo processo e então fazer que o núcleo execute o processo mais prioritário e que seja executável. O mecanismo — no núcleo — é procurar pelo processo mais prioritário e executá-lo. A política — designar prioridades para processos — pode ser implementada por processos de modo usuário. Dessa maneira, política e mecanismo podem ser desacoplados e o núcleo tornado menor.

1.7.4 O modelo cliente-servidor

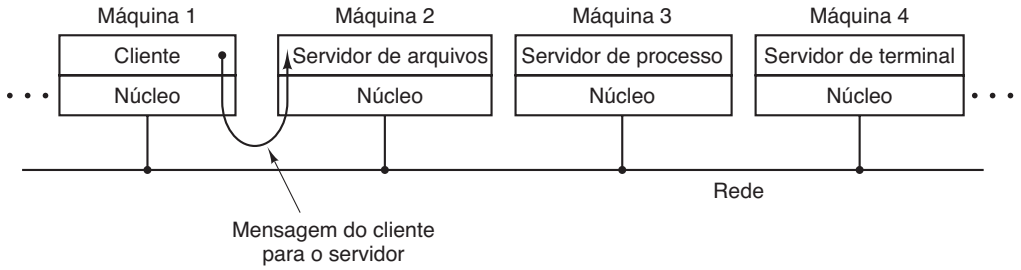
Uma ligeira variação da ideia do micronúcleo é distinguir duas classes de processos, os **servidores**, que prestam algum serviço, e os **clientes**, que usam esses serviços. Esse modelo é conhecido como o modelo **cliente-servidor**. Muitas vezes, a camada mais baixa é a do micronúcleo, mas isso não é necessário. A essência encontra-se na presença de processos clientes e processos servidores.

A comunicação entre clientes e servidores é realizada muitas vezes pela troca de mensagens. Para obter um serviço, um processo cliente constrói uma mensagem dizendo o que ele quer e a envia ao serviço apropriado. O serviço então realiza o trabalho e envia de volta a resposta. Se acontecer de o cliente e o servidor serem executados na mesma máquina, determinadas otimizações são possíveis, mas conceitualmente, ainda estamos falando da troca de mensagens aqui.

Uma generalização óbvia dessa ideia é ter os clientes e servidores sendo executados em computadores diferentes, conectados por uma rede local ou de grande área, como descrito na Figura 1.27. Tendo em vista que os clientes comunicam-se com os servidores enviando mensagens, os clientes não precisam saber se as mensagens são entregues localmente em suas próprias máquinas, ou se são enviadas através de uma rede para servidores em uma máquina remota. No que diz respeito ao cliente, a mesma coisa acontece em ambos os casos: pedidos são enviados e as respostas retornadas. Desse modo, o modelo cliente-servidor é uma abstração que pode ser usada para uma única máquina ou para uma rede de máquinas.

Cada vez mais, muitos sistemas envolvem usuários em seus PCs em casa como clientes e grandes máquinas em outra parte operando como servidores. Na realidade, grande parte da web opera dessa maneira. Um PC pede uma página na web para um servidor e ele a entrega. Esse é o uso típico do modelo cliente-servidor em uma rede.

FIGURA 1.27 O modelo cliente-servidor em uma rede.



1.7.5 Máquinas virtuais

Os lançamentos iniciais do OS/360 foram estritamente sistemas em lote. Não obstante isso, muitos usuários do 360 queriam poder trabalhar interativamente em um terminal, de maneira que vários grupos, tanto dentro quanto fora da IBM, decidiram escrever sistemas de compartilhamento de tempo para ele. O sistema de compartilhamento de tempo oficial da IBM, TSS/360, foi lançado tarde, e quando enfim chegou, era tão grande e lento que poucos converteram-se a ele. Ele foi finalmente abandonado após o desenvolvimento ter consumido algo em torno de US$ 50 milhões (GRAHAM, 1970). Mas um grupo no Centro Científico da IBM em Cambridge, Massachusetts, produziu um sistema radicalmente diferente que a IBM por fim aceitou como produto. Um descendente linear, chamado **z/VM**, é hoje amplamente usado nos computadores de grande porte da IBM, os zSeries, que são intensamente usados em grandes centros de processamento de dados corporativos, por exemplo, como servidores de comércio eletrônico que lidam com centenas ou milhares de transações por segundo e usam bancos de dados cujos tamanhos chegam a milhões de gigabytes.

VM/370

Esse sistema, na origem chamado CP/CMS e mais tarde renomeado VM/370 (SEAWRIGHT e MacKINNON, 1979), foi baseado em uma observação astuta: um sistema de compartilhamento de tempo fornece (1) multiprogramação e (2) uma máquina estendida com uma interface mais conveniente do que apenas o hardware. A essência do VM/370 é separar completamente essas duas funções.

O cerne do sistema, conhecido como o **monitor de máquina virtual**, opera direto no hardware e realiza a multiprogramação, fornecendo não uma, mas várias máquinas virtuais para a camada seguinte, como mostrado na Figura 1.28. No entanto, diferentemente de todos os outros sistemas operacionais, essas máquinas virtuais não são máquinas estendidas, com arquivos e outros aspectos interessantes. Em vez disso, elas são cópias *exatas* do hardware exposto, incluindo modos núcleo/usuário, E/S, interrupções e tudo mais que a máquina tem.

Como cada máquina virtual é idêntica ao hardware original, cada uma delas pode executar qualquer sistema operacional capaz de ser executado diretamente sobre o hardware. Máquinas virtuais diferentes podem — e frequentemente o fazem — executar diferentes sistemas operacionais. No sistema VM/370 original da IBM, em algumas é executado o sistema operacional OS/360 ou um dos outros sistemas operacionais de processamento de transações ou em lote grande, enquanto em outras é executado um sistema operacional monousuário interativo chamado **CMS (Conversational Monitor System** — sistema monitor conversacional), para usuários interativos em tempo compartilhado. Esse sistema era popular entre os programadores.

FIGURA 1.28 A estrutura do VM/370 com CMS.