CAPÍTULO 4

SISTEMAS DE ARQUIVOS

Todas as aplicações de computadores precisam armazenar e recuperar informações. Enquanto um processo está sendo executado, ele pode armazenar uma quantidade limitada de informações dentro do seu próprio espaço de endereçamento. No entanto, a capacidade de armazenamento está restrita ao tamanho do espaço do endereçamento virtual. Para algumas aplicações esse tamanho é adequado, mas, para outras, como reservas de passagens aéreas, bancos ou sistemas corporativos, ele é pequeno demais.

Um segundo problema em manter informações dentro do espaço de endereçamento de um processo é que, quando o processo é concluído, as informações são perdidas. Para muitas aplicações (por exemplo, bancos de dados), as informações precisam ser retidas por semanas, meses, ou mesmo para sempre. Perdê-las quando o processo que as está utilizando é concluído é algo inaceitável. Além disso, elas não devem desaparecer quando uma falha no computador mata um processo.

Um terceiro problema é que frequentemente é necessário que múltiplos processos acessem (partes de) uma informação ao mesmo tempo. Se temos um diretório telefônico on-line armazenado dentro do espaço de um único processo, apenas aquele processo pode acessá-lo. A maneira para solucionar esse problema é tornar a informação em si independente de qualquer processo.

Assim, temos três requisitos essenciais para o armazenamento de informações por um longo prazo:

1. Deve ser possível armazenar uma quantidade muito grande de informações.
2. As informações devem sobreviver ao término do processo que as está utilizando.
3. Múltiplos processos têm de ser capazes de acessá-las ao mesmo tempo.

Discos magnéticos foram usados por anos para esse armazenamento de longo prazo. Em anos recentes, unidades de estado sólido tornaram-se cada vez mais populares, à medida que elas não têm partes móveis que possam quebrar. Elas também oferecem um rápido acesso aleatório. Fitas e discos óticos também foram amplamente usados, mas são dispositivos com um desempenho muito pior e costumam ser usados como backups. Estudaremos mais sobre discos no Capítulo 5, mas por ora, basta pensar em um disco como uma sequência linear de blocos de tamanho fixo e que dão suporte a duas operações:

1. Leia o bloco *k*.
2. Escreva no bloco *k*.

Na realidade, existem mais operações, mas com essas duas, em princípio, você pode solucionar o problema do armazenamento de longo prazo.

No entanto, essas são operações muito inconvenientes, mais ainda em sistemas grandes usados por muitas aplicações e possivelmente múltiplos usuários (por exemplo, em um servidor). Apenas algumas das questões que rapidamente surgem são:

1. Como você encontra informações?
2. Como impedir que um usuário leia os dados de outro?
3. Como saber quais blocos estão livres?

e há muitas mais.

Da mesma maneira que vimos como o sistema operacional abstraía o conceito do processador para criar a abstração de um processo e como ele abstraía o conceito da memória física para oferecer aos processos espaços de endereçamento (virtuais), podemos solucionar esse problema com uma nova abstração: o arquivo. Juntas, as abstrações de processos (e threads), espaços de endereçamento e arquivos são os conceitos mais importantes relacionados com os sistemas operacionais. Se você realmente compreender esses três conceitos do início ao fim, estará bem encaminhado para se tornar um especialista em sistemas operacionais.

Arquivos são unidades lógicas de informação criadas por processos. Um disco normalmente conterá milhares ou mesmo milhões deles, cada um independente dos outros. Na realidade, se pensar em cada arquivo como uma espécie de espaço de endereçamento, você não estará muito longe da verdade, exceto que eles são usados para modelar o disco em vez de modelar a RAM.

Processos podem ler arquivos existentes e criar novos se necessário. Informações armazenadas em arquivos devem ser **persistentes**, isto é, não devem ser afetadas pela criação e término de um processo. Um arquivo deve desaparecer apenas quando o seu proprietário o remove explicitamente. Embora as operações para leitura e escrita de arquivos sejam as mais comuns, existem muitas outras, algumas das quais examinaremos a seguir.

Arquivos são gerenciados pelo sistema operacional. Como são estruturados, nomeados, acessados, usados, protegidos, implementados e gerenciados são tópicos importantes no projeto de um sistema operacional. Como um todo, aquela parte do sistema operacional lidando com arquivos é conhecida como **sistema de arquivos** e é o assunto deste capítulo.

Do ponto de vista do usuário, o aspecto mais importante de um sistema de arquivos é como ele aparece, em outras palavras, o que constitui um arquivo, como os arquivos são nomeados e protegidos, quais operações são permitidas e assim por diante. Os detalhes sobre se listas encadeadas ou mapas de bits são usados para o armazenamento disponível e quantos setores existem em um bloco de disco lógico não lhes interessam, embora sejam de grande importância para os projetistas do sistema de arquivos. Por essa razão, estruturamos o capítulo como várias seções. As duas primeiras dizem respeito à interface do usuário para os arquivos e para os diretórios, respectivamente. Então segue uma discussão detalhada de como o sistema de arquivos é implementado e gerenciado. Por fim, damos alguns exemplos de sistemas de arquivos reais.

4.1 Arquivos

Nas páginas a seguir examinaremos os arquivos do ponto de vista do usuário, isto é, como eles são usados e quais propriedades têm.

4.1.1 Nomeação de arquivos

Um arquivo é um mecanismo de abstração. Ele fornece uma maneira para armazenar informações sobre o disco e lê-las depois. Isso deve ser feito de tal modo que isole o usuário dos detalhes de como e onde as informações estão armazenadas, e como os discos realmente funcionam.

É provável que a característica mais importante de qualquer mecanismo de abstração seja a maneira como os objetos que estão sendo gerenciados são nomeados; portanto, começaremos nosso exame dos sistemas de arquivos com o assunto da nomeação de arquivos. Quando um processo cria um arquivo, ele lhe dá um nome. Quando o processo é concluído, o arquivo continua a existir e pode ser acessado por outros processos usando o seu nome.

As regras exatas para a nomeação de arquivos variam de certa maneira de sistema para sistema, mas todos os sistemas operacionais atuais permitem cadeias de uma a oito letras como nomes de arquivos legais. Desse modo, *andrea*, *bruce* e *cathy* são nomes de arquivos possíveis. Não raro, dígitos e caracteres especiais também são permitidos, assim nomes como *2, urgente!* e *Fig.2-14* são muitas vezes válidos também. Muitos sistemas de arquivos aceitam nomes com até 255 caracteres.

Alguns sistemas de arquivos distinguem entre letras maiúsculas e minúsculas, enquanto outros, não. O UNIX pertence à primeira categoria; o velho MS-DOS cai na segunda. (Como nota, embora antigo, o MS-DOS ainda é amplamente usado em sistemas embarcados, portanto ele não é obsoleto de maneira alguma.) Assim, um sistema UNIX pode ter todos os arquivos a seguir como três arquivos distintos: *maria*, *Maria* e *MARIA*. No MS-DOS, todos esses nomes referem-se ao mesmo arquivo.

Talvez seja um bom momento para fazer um comentário aqui sobre os sistemas operacionais. O Windows 95 e o Windows 98 usavam o mesmo sistema de arquivos do MS-DOS, chamado **FAT-16**, e portanto herdaram muitas de suas propriedades, como a maneira de se formarem os nomes dos arquivos. O Windows 98 introduziu algumas extensões ao FAT-16, levando ao **FAT-32**, mas esses dois são bastante parecidos. Além disso, o Windows NT, Windows 2000, Windows XP, Windows

Vista, Windows 7 e Windows 8 ainda dão suporte a ambos os sistemas de arquivos FAT, que estão realmente obsoletos agora. No entanto, esses sistemas operacionais novos também têm um sistema de arquivos nativo muito mais avançado (**NTFS — native file system**) que tem propriedades diferentes (como nomes de arquivos em Unicode). Na realidade, há um segundo sistema de arquivos para o Windows 8, conhecido como **ReFS** (ou **Resilient File System — sistema de arquivos resiliente**), mas ele é voltado para a versão de servidor do Windows 8. Neste capítulo, quando nos referimos ao MS-DOS ou sistemas de arquivos FAT, estaremos falando do FAT-16 e FAT-32 como usados no Windows, a não ser que especificado de outra forma. Discutiremos o sistema de arquivos FAT mais tarde neste capítulo e NTFS no Capítulo 12, onde examinaremos o Windows 8 com detalhes. Incidentalmente, existe também um novo sistema de arquivos semelhante ao FAT, conhecido como sistema de arquivos **exFAT**, uma extensão da Microsoft para o FAT-32 que é otimizado para flash drives e sistemas de arquivos grandes. ExFAT é o único sistema de arquivos moderno da Microsoft que o OS X pode ler e escrever.

Muitos sistemas operacionais aceitam nomes de arquivos de duas partes, com as partes separadas por um ponto, como em *prog.c*. A parte que vem em seguida ao ponto é chamada de **extensão do arquivo** e costuma indicar algo seu a respeito. No MS-DOS, por exemplo, nomes de arquivos têm de 1 a 8 caracteres, mais uma extensão opcional de 1 a 3 caracteres. No UNIX, o tamanho da extensão, se houver, cabe ao usuário decidir, e um arquivo pode ter até duas ou mais extensões, como em *homepage.html.zip*, onde *.html* indica uma página da web em HTML e *.zip* indica que o arquivo (*homepage.html*) foi compactado usando o programa *zip*. Algumas das extensões de arquivos mais comuns e seus significados são mostradas na Figura 4.1.

Em alguns sistemas (por exemplo, todas as variações do UNIX), as extensões de arquivos são apenas convenções e não são impostas pelo sistema operacional. Um arquivo chamado *file.txt* pode ser algum tipo de arquivo de texto, mas aquele nome tem a função mais de lembrar o proprietário do que transmitir qualquer informação real para o computador. Por outro lado, um compilador C pode realmente insistir em que os arquivos que ele tem de compilar terminem em *.c*, e se isso não acontecer, pode recusar-se a compilá-los. O sistema operacional, no entanto, não se importa.

Convenções como essa são especialmente úteis quando o mesmo programa pode lidar com vários tipos diferentes de arquivos. O compilador C, por exemplo, pode receber uma lista de vários arquivos a serem compilados e ligados, alguns deles arquivos C e outros arquivos de linguagem de montagem. A extensão então se torna essencial para o compilador dizer quais são os

FIGURA 4.1 Algumas extensões comuns de arquivos.

Extensão	Significado
.bak	Cópia de segurança
.c	Código-fonte de programa em C
.gif	Imagem no formato Graphical Interchange Format
.hlp	Arquivo de ajuda
.html	Documento em HTML
.jpg	Imagem codificada segundo padrões JPEG
.mp3	Música codificada no formato MPEG (camada 3)
.mpg	Filme codificado no padrão MPEG
.o	Arquivo objeto (gerado por compilador, ainda não ligado)
.pdf	Arquivo no formato PDF (Portable Document File)
.ps	Arquivo PostScript
.tex	Entrada para o programa de formatação TEX
.txt	Arquivo de texto
.zip	Arquivo compactado

arquivos C, os arquivos de linguagem de montagem e outros arquivos.

Em contrapartida, o Windows é consciente das extensões e designa significados a elas. Usuários (ou processos) podem registrar extensões com o sistema operacional e especificar para cada uma qual é seu "proprietário". Quando um usuário clica duas vezes sobre o nome de um arquivo, o programa designado para essa extensão de arquivo é lançado com o arquivo como parâmetro. Por exemplo, clicar duas vezes sobre *file.docx* inicializará o Microsoft Word, tendo *file.docx* como seu arquivo inicial para edição.

4.1.2 Estrutura de arquivos

Arquivos podem ser estruturados de várias maneiras. Três possibilidades comuns estão descritas na Figura 4.2. O arquivo na Figura 4.2(a) é uma sequência desestruturada de bytes. Na realidade, o sistema operacional não sabe ou não se importa sobre o que há no arquivo. Tudo o que ele vê são bytes. Qualquer significado deve ser imposto por programas em nível de usuário. Tanto UNIX quanto Windows usam essa abordagem.

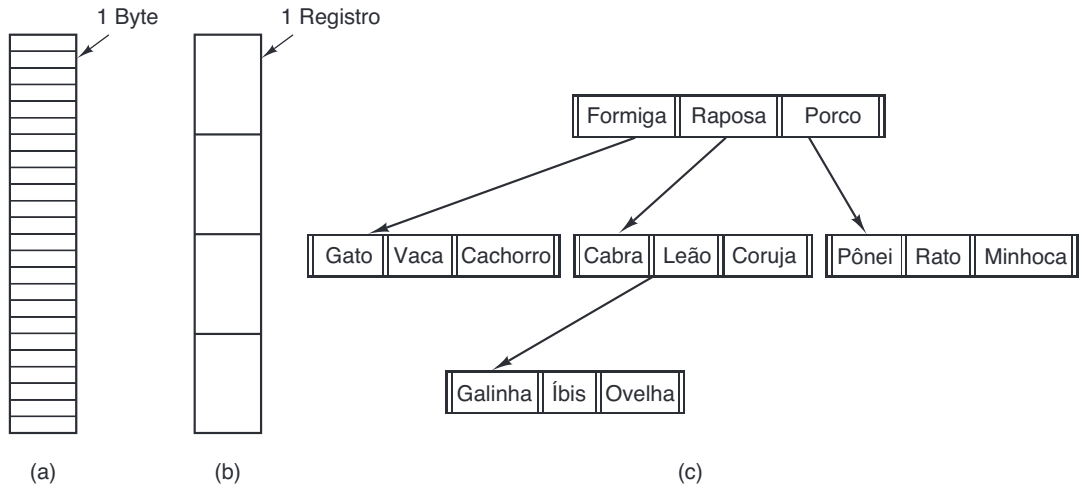
Ter o sistema operacional tratando arquivos como nada mais que sequências de bytes oferece a máxima flexibilidade. Programas de usuários podem colocar qualquer coisa que eles quiserem em seus arquivos e nomeá-los do jeito que acharem conveniente. O sistema operacional não ajuda, mas também não interfere. Para usuários que querem realizar coisas incomuns, o segundo ponto pode ser muito importante. Todas as versões do UNIX (incluindo Linux e OS X) e o Windows usam esse modelo de arquivos.

O primeiro passo na estruturação está ilustrado na Figura 4.2(b). Nesse modelo, um arquivo é uma sequência de registros de tamanho fixo, cada um com alguma estrutura interna. O fundamental para que um arquivo seja uma sequência de registros é a ideia de que a operação de leitura retorna um registro e a operação de escrita sobrepõe ou anexa um registro. Como nota histórica, décadas atrás, quando o cartão de 80 colunas perfurado era o astro, muitos sistemas operacionais de computadores de grande porte baseavam seus sistemas de arquivos em arquivos consistindo em registros de 80 caracteres, na realidade, imagens de cartões. Esses sistemas também aceitavam arquivos com registros de 132 caracteres, destinados às impressoras de linha (que naquela época eram grandes impressoras de corrente com 132 colunas). Os programas liam a entrada em unidades de 80 caracteres e a escreviam em unidades de 132 caracteres, embora os últimos 52 pudessem ser espaços, é claro. Nenhum sistema de propósito geral atual usa mais esse modelo como seu sistema primário de arquivos, mas na época dos cartões perfurados de 80 colunas e impressoras de 132 caracteres por linha era um modelo comum em computadores de grande porte.

O terceiro tipo de estrutura de arquivo é mostrado na Figura 4.2(c). Nessa organização, um arquivo consiste em uma árvore de registros, não necessariamente todos do mesmo tamanho, cada um contendo um campo **chave** em uma posição fixa no registro. A árvore é ordenada no campo chave, a fim de permitir uma busca rápida por uma chave específica.

A operação básica aqui não é obter o "próximo" registro, embora isso também seja possível, mas aquele com a chave específica. Para o arquivo zoológico da Figura 4.2(c), você poderia pedir ao sistema para obter

FIGURA 4.2 Três tipos de arquivos. (a) Sequência de bytes. (b) Sequência de registros. (c) Árvore.



o registro cuja chave fosse *pônei*, por exemplo, sem se preocupar com sua posição exata no arquivo. Além disso, novos registros podem ser adicionados, com o sistema o operacional, e não o usuário, decidindo onde colocá-los. Esse tipo de arquivo é claramente bastante diferente das sequências de bytes desestruturadas usadas no UNIX e Windows, e é usado em alguns computadores de grande porte para o processamento de dados comerciais.

4.1.3 Tipos de arquivos

Muitos sistemas operacionais aceitam vários tipos de arquivos. O UNIX (novamente, incluindo OS X) e o Windows, por exemplo, apresentam arquivos regulares e diretórios. O UNIX também tem arquivos especiais de caracteres e blocos. **Arquivos regulares** são aqueles que contêm informações do usuário. Todos os arquivos da Figura 4.2 são arquivos regulares. **Diretórios** são arquivos do sistema para manter a estrutura do sistema de arquivos. Estudaremos diretórios a seguir. **Arquivos especiais de caracteres** são relacionados com entrada/saída e usados para modelar dispositivos de E/S seriais como terminais, impressoras e redes. **Arquivos especiais de blocos** são usados para modelar discos. Neste capítulo, estaremos interessados fundamentalmente em arquivos regulares.

Arquivos regulares geralmente são arquivos ASCII ou arquivos binários. Arquivos ASCII consistem de linhas de texto. Em alguns sistemas, cada linha termina com um caractere de retorno de carro (carriage return). Em outros, o caractere de próxima linha (line feed) é usado. Alguns sistemas (por exemplo, Windows) usam ambos. As linhas não precisam ser todas do mesmo tamanho.

A grande vantagem dos arquivos ASCII é que eles podem ser exibidos e impressos como são e editados com qualquer editor de texto. Além disso, se grandes números de programas usam arquivos ASCII para entrada e saída, é fácil conectar a saída de um programa com a entrada de outro, como em pipelines do interpretador de comandos (*shell*). (O uso de pipelines entre processos não é nem um pouco mais fácil, mas a interpretação da informação certamente torna-se mais fácil se uma convenção padrão, como a ASCII, for usada para expressá-la.)

Outros arquivos são binários, o que apenas significa que eles não são arquivos ASCII. Listá-los em uma impressora resultaria em algo completamente incompreensível. Em geral, eles têm alguma estrutura interna conhecida pelos programas que os usam.

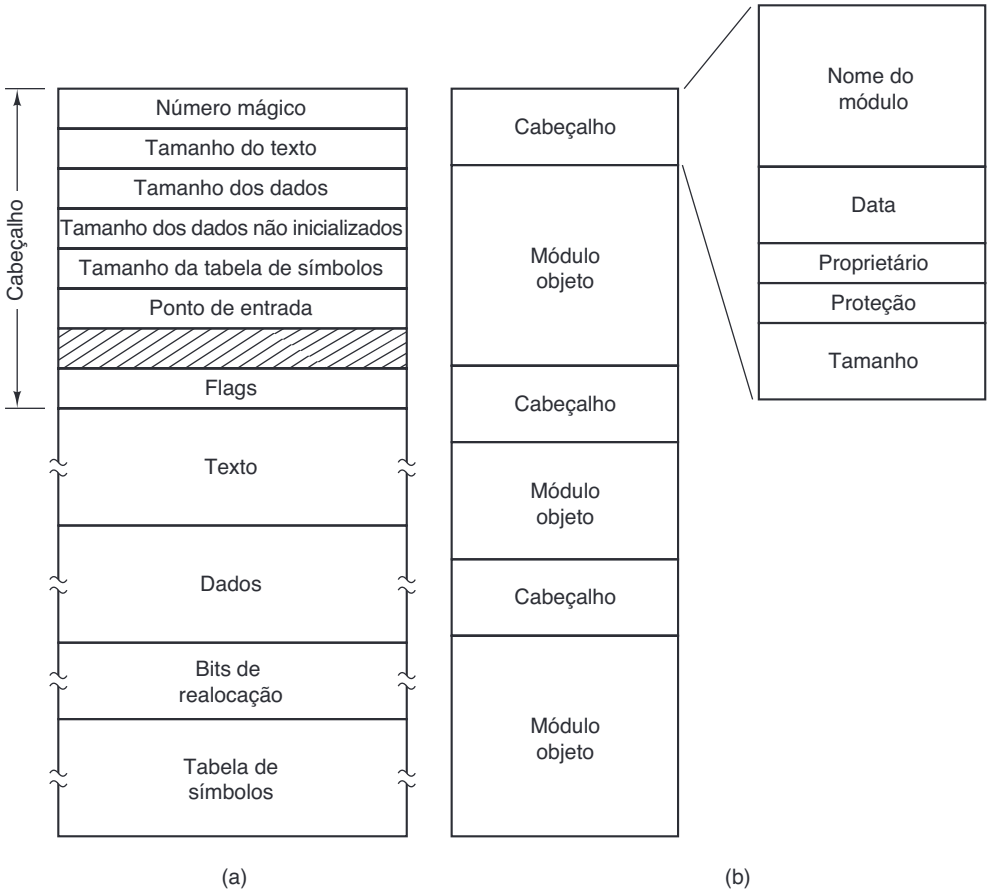
Por exemplo, na Figura 4.3(a) vemos um arquivo binário executável simples tirado de uma versão inicial do UNIX. Embora tecnicamente o arquivo seja apenas uma sequência de bytes, o sistema operacional o executará somente se ele tiver o formato apropriado. Ele tem cinco seções: cabeçalho, texto, dados, bits de realocação e tabela de símbolos. O cabeçalho começa com o chamado **número mágico**, identificando o arquivo como executável (para evitar a execução acidental de um arquivo que não esteja em seu formato). Então vêm os tamanhos das várias partes do arquivo, o endereço no qual a execução começa e alguns bits de sinalização. Após o cabeçalho, estão o texto e os dados do próprio programa, que são carregados para a memória e realocados usando os bits de realocação. A tabela de símbolos é usada para correção de erros.

Nosso segundo exemplo de um arquivo binário é um repositório (archive), também do UNIX. Ele consiste em uma série de rotinas de biblioteca (módulos) compiladas, mas não ligadas. Cada uma é prefaciada por um cabeçalho dizendo seu nome, data de criação, proprietário, código de proteção e tamanho. Da mesma forma que o arquivo executável, os cabeçalhos de módulos estão cheios de números binários. Copiá-los para a impressora produziria puro lixo.

Todo sistema operacional deve reconhecer pelo menos um tipo de arquivo: o seu próprio arquivo executável; alguns reconhecem mais. O velho sistema TOPS-20 (para o DECSystem 20) chegou ao ponto de examinar data e horário de criação de qualquer arquivo a ser executado. Então ele localizava o arquivo-fonte e via se a fonte havia sido modificada desde a criação do binário. Em caso positivo, ele automaticamente recompilava a fonte. Em termos de UNIX, o programa *make* havia sido embutido no shell. As extensões de arquivos eram obrigatórias, então ele poderia dizer qual programa binário era derivado de qual fonte.

Ter arquivos fortemente tipificados como esse causa problemas sempre que o usuário fizer algo que os projetistas do sistema não esperavam. Considere, como um exemplo, um sistema no qual os arquivos de saída do programa têm a extensão *.dat* (arquivos de dados). Se um usuário escrever um formatador de programa que lê um arquivo *.c* (programa C), o transformar (por exemplo, convertendo-o em um layout padrão de indentação), e então escrever o arquivo transformado como um arquivo de saída, ele será do tipo *.dat*. Se o usuário tentar oferecer isso ao compilador C para compilá-lo, o sistema se recusará porque ele tem a extensão errada. Tentativas de copiar *file.dat* para *file.c* serão rejeitadas

FIGURA 4.3 (a) Um arquivo executável. (b) Um repositório (archive).



pelo sistema como inválidas (a fim de proteger o usuário contra erros).

Embora esse tipo de "facilidade para o usuário" possa ajudar os novatos, é um estorvo para os usuários experientes, pois eles têm de devotar um esforço considerável para driblar a ideia do sistema operacional do que seja razoável ou não.

4.1.4 Acesso aos arquivos

Os primeiros sistemas operacionais forneciam apenas um tipo de acesso aos arquivos: **acesso sequencial**. Nesses sistemas, um processo podia ler todos os bytes ou registros em um arquivo em ordem, começando do princípio, mas não podia pular nenhum ou lê-los fora de ordem. No entanto, arquivos sequenciais podiam ser trazidos de volta para o ponto de partida, então eles podiam ser lidos tantas vezes quanto necessário. Arquivos sequenciais eram convenientes quando o meio de armazenamento era uma fita magnética, em vez de um disco.

Quando os discos passaram a ser usados para armazenar arquivos, tornou-se possível ler os bytes ou registros de um arquivo fora de ordem, ou acessar os registros pela chave em vez de pela posição. Arquivos ou registros que podem ser lidos em qualquer ordem são chamados de **arquivos de acesso aleatório**. Eles são necessários para muitas aplicações.

Arquivos de acesso aleatório são essenciais para muitas aplicações, por exemplo, sistemas de bancos de dados. Se um cliente de uma companhia aérea liga e quer reservar um assento em um determinado voo, o programa de reservas deve ser capaz de acessar o registro para aquele voo sem ter de ler primeiro os registros para milhares de outros voos.

Dois métodos podem ser usados para especificar onde começar a leitura. No primeiro, cada operação read fornece a posição no arquivo onde começar a leitura. No segundo, uma operação simples, seek, é fornecida para estabelecer a posição atual. Após um seek, o arquivo pode ser lido sequencialmente da posição agora atual. O segundo método é usado no UNIX e no Windows.

4.1.5 Atributos de arquivos

Todo arquivo possui um nome e sua data. Além disso, todos os sistemas operacionais associam outras

informações com cada arquivo, por exemplo, a data e o horário em que foi modificado pela última vez, assim como o tamanho do arquivo. Chamaremos esses itens extras de **atributos** do arquivo. Algumas pessoas os chamam de **metadados**. A lista de atributos varia bastante de um sistema para outro. A tabela da Figura 4.4 mostra algumas das possibilidades, mas existem outras. Nenhum sistema existente tem todos esses atributos, mas cada um está presente em algum sistema.

Os primeiros quatro atributos concernem à proteção do arquivo e dizem quem pode acessá-lo e quem não pode. Todos os tipos de esquemas são possíveis, alguns dos quais estudaremos mais tarde. Em alguns sistemas o usuário deve apresentar uma senha para acessar um arquivo, caso em que a senha deve ser um dos atributos.

As sinalizações (flags) são bits ou campos curtos que controlam ou habilitam alguma propriedade específica. Arquivos ocultos, por exemplo, não aparecem nas listagens de todos os arquivos. A sinalização de arquivamento é um bit que controla se foi feito um backup do arquivo recentemente. O programa de backup remove esse bit e o sistema operacional o recoloca sempre que um arquivo for modificado. Dessa maneira, o programa consegue dizer quais arquivos precisam de backup. A sinalização temporária permite que um arquivo seja marcado para ser deletado automaticamente quando o processo que o criou for concluído.

O tamanho do registro, posição da chave e tamanho dos campos-chave estão presentes apenas em arquivos cujos registros podem ser lidos usando uma chave. Eles proporcionam a informação necessária para encontrar as chaves.

Os vários registros de tempo controlam quando o arquivo foi criado, acessado e modificado pela última vez, os quais são úteis para uma série de finalidades. Por exemplo, um arquivo-fonte que foi modificado após a criação do arquivo-objeto correspondente precisa ser recompilado. Esses campos fornecem as informações necessárias.

O tamanho atual nos informa o tamanho que o arquivo tem no momento. Alguns sistemas operacionais

FIGURA 4.4 Alguns possíveis atributos de arquivos.

Atributo	Significado
Proteção	Quem tem acesso ao arquivo e de que modo
Senha	Necessidade de senha para acesso ao arquivo
Criador	ID do criador do arquivo
Proprietário	Proprietário atual
Flag de somente leitura	0 para leitura/escrita; 1 para somente leitura
Flag de oculto	0 para normal; 1 para não exibir o arquivo
Flag de sistema	0 para arquivos normais; 1 para arquivos de sistema
Flag de arquivamento	0 para arquivos com backup; 1 para arquivos sem backup
Flag de ASCII/binário	0 para arquivos ASCII; 1 para arquivos binários
Flag de acesso aleatório	0 para acesso somente sequencial; 1 para acesso aleatório
Flag de temporário	0 para normal; 1 para apagar o arquivo ao sair do processo
Flag de travamento	0 para destravados; diferente de 0 para travados
Tamanho do registro	Número de bytes em um registro
Posição da chave	Posição da chave em cada registro
Tamanho da chave	Número de bytes na chave
Momento de criação	Data e hora de criação do arquivo
Momento do último acesso	Data e hora do último acesso do arquivo
Momento da última alteração	Data e hora da última modificação do arquivo
Tamanho atual	Número de bytes no arquivo
Tamanho máximo	Número máximo de bytes no arquivo

de antigos computadores de grande porte exigiam que o tamanho máximo fosse especificado quando o arquivo fosse criado, a fim de deixar que o sistema operacional reservasse a quantidade máxima de memória antecipadamente. Sistemas operacionais de computadores pessoais e de estações de trabalho são inteligentes o suficiente para não precisarem desse atributo.

4.1.6 Operações com arquivos

Arquivos existem para armazenar informações e permitir que elas sejam recuperadas depois. Sistemas diferentes proporcionam operações diferentes para permitir armazenamento e recuperação. A seguir uma discussão das chamadas de sistema mais comuns relativas a arquivos.

1. Create. O arquivo é criado sem dados. A finalidade dessa chamada é anunciar que o arquivo está vindo e estabelecer alguns dos atributos.
2. Delete. Quando o arquivo não é mais necessário, ele tem de ser removido para liberar espaço para o disco. Há sempre uma chamada de sistema para essa finalidade.
3. Open. Antes de usar um arquivo, um processo precisa abri-lo. A finalidade da chamada open é permitir que o sistema busque os atributos e lista de endereços do disco para a memória principal a fim de tornar mais rápido o acesso em chamadas posteriores.
4. Close. Quando todos os acessos são concluídos, os atributos e endereços de disco não são mais necessários, então o arquivo deve ser fechado para liberar espaço da tabela interna. Muitos sistemas encorajam isso impondo um número máximo de arquivos abertos em processos. Um disco é escrito em blocos, e o fechamento de um arquivo força a escrita do último bloco dele, mesmo que não esteja inteiramente cheio ainda.
5. Read. Dados são lidos do arquivo. Em geral, os bytes vêm da posição atual. Quem fez a chamada deve especificar a quantidade de dados necessária e também fornecer um buffer para colocá-los.
6. Write. Dados são escritos para o arquivo de novo, normalmente na posição atual. Se a posição atual for o final do arquivo, seu tamanho aumentará. Se estiver no meio do arquivo, os dados existentes serão sobrescritos e perdidos para sempre.
7. Append. Essa chamada é uma forma restrita de write. Ela pode acrescentar dados somente para o final do arquivo. Sistemas que fornecem um conjunto mínimo de chamadas do sistema raramente têm append, mas muitos sistemas fornecem múltiplas maneiras de fazer a mesma coisa, e esses às vezes têm append.
8. Seek. Para arquivos de acesso aleatório, é necessário um método para especificar de onde tirar os dados. Uma abordagem comum é uma chamada de sistema, seek, que reposiciona o ponteiro de arquivo para um local específico dele. Após essa chamada ter sido completa, os dados podem ser lidos da, ou escritos para, aquela posição.
9. Get attributes. Processos muitas vezes precisam ler atributos de arquivos para realizar seu trabalho. Por exemplo, o programa *make* da UNIX costuma ser usado para gerenciar projetos de desenvolvimento de software consistindo de muitos arquivos-fonte. Quando *make* é chamado, ele examina os momentos de alteração de todos os arquivos-fonte e objetos e organiza o número mínimo de compilações necessárias para atualizar tudo. Para realizar o trabalho, o *make* deve examinar os atributos, a saber, os momentos de alteração.
10. Set attributes. Alguns dos atributos podem ser alterados pelo usuário e modificados após o arquivo ter sido criado. Essa chamada de sistema torna isso possível. A informação sobre o modo de proteção é um exemplo óbvio. A maioria das sinalizações também cai nessa categoria.
11. Rename. Acontece com frequência de um usuário precisar mudar o nome de um arquivo. Essa chamada de sistema torna isso possível. Ela nem sempre é estritamente necessária, porque o arquivo em geral pode ser copiado para um outro com um nome novo, e o arquivo antigo é então deletado.

4.1.7 Exemplo de um programa usando chamadas de sistema para arquivos

Nesta seção examinaremos um programa UNIX simples que copia um arquivo do seu arquivo-fonte para um de destino. Ele está listado na Figura 4.5. O programa tem uma funcionalidade mínima e um mecanismo para reportar erros ainda pior, mas proporciona uma ideia razoável de como algumas das chamadas de sistema relacionadas a arquivos funcionam.

FIGURA 4.5 Um programa simples para copiar um arquivo.

```
/* Programa que copia arquivos. Verificacao e relato de erros e minimo.*/

#include <sys/types.h>                  /* inclui os arquivos de cabecalho necessarios*/
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* prototipo ANS */

#define BUF_SIZE 4096                   /* usa um tamanho de buffer de 4096 bytes*/
#define OUTPUT_MODE 0700                /* bits de protecao para o arquivo de saida*/

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);             /* erro de sintaxe se argc nao for 3*/

    /* Abre o arquivo de entrada e cria o arquivo de saida*/
    in_fd = open(argv[1], O_RDONLY);    /* abre o arquivo de origem*/
    if (in_fd < 0) exit(2);             /* se nao puder ser aberto, saia*/
    out_fd = creat(argv[2], OUTPUT_MODE); /* cria o arquivo de destino */
    if (out_fd < 0) exit(3);            /* se nao puder ser criado, saia*/

    /* Laco de copia*/
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* le um bloco de dados*/
        if (rd_count <= 0) break        /* se fim de arquivo ou erro, sai do laco*/
        wt_count = write(out_fd, buffer, rd_count); /* escreve dados*/
        if (wt_count <= 0) exit(4);     /* wt_count <= 0 e um erro*/
    }

    /* Fecha os arquivos*/
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)                  /* nenhum erro na ultima leitura*/
        exit(0);
    else
        exit(5);                        /* erro na ultima leitura*/
}
```

O programa, *copyfile*, pode ser chamado, por exemplo, pela linha de comando

```
copyfile abc xyz
```

para copiar o arquivo *abc* para *xyz*. Se *xyz* já existir, ele será sobrescrito. De outra maneira, ele será criado. O programa precisa ser chamado com exatamente dois argumentos, ambos nomes legais de arquivos. O primeiro é o fonte; o segundo é o arquivo de saída.

Os quatro comandos *#include* próximos do início do programa fazem que um grande número de definições e protótipos de funções sejam incluídos no programa. Essas inclusões são necessárias para deixá-lo em conformidade com os padrões internacionais relevantes, mas não nos ocuparemos mais com elas. A linha seguinte é um protótipo da função para *main*, algo exigido pelo ANSI C, mas também não relevante para nossas finalidades.

O primeiro comando *#define* é uma definição macro, que estabelece a sequência de caracteres *BUF_SIZE* como uma macro que se expande no número 4096. O programa lerá e escreverá em pedaços de 4096 bytes. É considerada uma boa prática de programação dar nomes a constantes como essa e usá-los em vez das constantes. Não apenas essa convenção torna os programas mais fáceis de ler, mas também de manter. O segundo comando *#define* determina quem pode acessar o arquivo de saída.

O programa principal é chamado *main* e tem dois argumentos: *argc* e *argv*. Estes são oferecidos pelo sistema operacional quando o programa é chamado. O primeiro diz quantas sequências estão presentes na linha de comando que invocou o programa, incluindo o nome dele. Deveriam ser 3. O segundo é um arranjo de ponteiros para os argumentos. Na chamada de exemplo dada, os elementos desse arranjo conteriam ponteiros para os seguintes valores:

argv[0] = “copyfile”

argv[1] = “abc”

argv[2] = “xyz”

É por meio desse arranjo que o programa acessa os seus argumentos.

Cinco variáveis são declaradas. As duas primeiras, *in_fd* e *out_fd*, conterão os **descritores de arquivos**, valores inteiros pequenos retornados quando um arquivo é aberto. As outras duas, *rd_count* e *wt_count*, são as contagens de bytes retornadas pelas chamadas de sistema read e write, respectivamente. A última, *buffer*, é um buffer usado para conter os dados lidos e fornecer os dados para serem escritos.

O primeiro comando real confere *argc* para ver se ele é 3. Se não for, o programa terminará com um código de estado 1. Qualquer código de estado diferente de 0 significa que ocorreu um erro. O código de estado é o único meio de reportar erros presente nesse programa. Uma versão comercial normalmente imprimiria mensagens de erros também.

Então tentamos abrir o arquivo-fonte e criar o arquivo-destino. Se o arquivo-fonte for aberto de maneira bem-sucedida, o sistema designa um pequeno inteiro para *in_fd*, a fim de identificá-lo. Chamadas subsequentes devem incluir esse inteiro de maneira que o sistema saiba qual arquivo ele quer. Similarmente, se o destino for criado de maneira bem-sucedida, *out_fd* recebe um valor para identificá-lo. O segundo argumento para *creat* estabelece o modo de proteção. Se a abertura ou a criação falhar, o descritor do arquivo correspondente será definido como –1, e o programa terminará com um código de erro.

Agora entra em cena o laço da cópia. Esse laço começa tentando ler 4 KB de dados para o *buffer*. Ele faz isso chamando a rotina de biblioteca *read*, que na realidade invoca a chamada de sistema read. O primeiro parâmetro identifica o arquivo, o segundo dá o buffer e o terceiro diz quantos bytes devem ser lidos. O valor designado para rd_count dá o número de bytes que foram realmente lidos. Em geral, esse valor será de 4096, exceto se menos bytes estiverem restando no arquivo. Quando o final do arquivo tiver sido alcançado, ele será 0. Se o *rd_count* chegar a 0 ou um valor negativo, a cópia não poderá continuar, de maneira que o comando *break* é executado para terminar o laço (de outra maneira interminável).

A chamada *write* descarrega o buffer para o arquivo de destino. O primeiro parâmetro identifica o arquivo, o segundo dá o buffer e o terceiro diz quantos bytes escrever, análogo a *read*. Observe que a contagem de bytes é o número de bytes realmente lidos, não *BUF_SIZE*. Esse ponto é importante porque o último *read* não retornará 4096 a não ser que o arquivo coincidentemente seja um múltiplo de 4 KB.

Quando o arquivo inteiro tiver sido processado, a primeira chamada além do fim do arquivo retornará a 0 para *rd_count*, que a fará deixar o laço. Nesse ponto, os dois arquivos estão próximos e o programa sai com um estado indicando uma conclusão normal.

Embora chamadas do sistema Windows sejam diferentes daquelas do UNIX, a estrutura geral de um programa ativado pela linha de comando no Windows para copiar um arquivo é moderadamente similar àquele da Figura 4.5. Examinaremos as chamadas do Windows 8 no Capítulo 11.

4.2 Diretórios

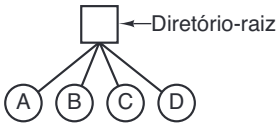
Para controlar os arquivos, sistemas de arquivos normalmente têm **diretórios** ou **pastas**, que são em si arquivos. Nesta seção discutiremos diretórios, sua organização, suas propriedades e as operações que podem ser realizadas por eles.

4.2.1 Sistemas de diretório em nível único

A forma mais simples de um sistema de diretório é ter um diretório contendo todos os arquivos. Às vezes ele é chamado de **diretório-raiz**, mas como ele é o único, o nome não importa muito. Nos primeiros computadores pessoais, esse sistema era comum, em parte porque havia apenas um usuário. Curiosamente, o primeiro supercomputador do mundo, o CDC 6600, também tinha apenas um único diretório para todos os arquivos, embora fosse usado por muitos usuários ao mesmo tempo. Essa decisão foi tomada sem dúvida para manter simples o design do software.

Um exemplo de um sistema com um diretório é dado na Figura 4.6. Aqui o diretório contém quatro arquivos. As vantagens desse esquema são a sua simplicidade e a capacidade de localizar arquivos rapidamente — há apenas um lugar para se procurar, afinal. Às vezes ele ainda

FIGURA 4.6 Um sistema de diretório em nível único contendo quatro arquivos.



é usado em dispositivos embarcados simples como câmeras digitais e alguns players portáteis de música.

4.2.2 Sistemas de diretórios hierárquicos

O nível único é adequado para aplicações dedicadas muito simples (e chegou a ser usado nos primeiros computadores pessoais), mas para os usuários modernos com milhares de arquivos seria impossível encontrar qualquer coisa se todos os arquivos estivessem em um único diretório.

Em consequência, é necessária uma maneira para agrupar arquivos relacionados em um mesmo local. Um professor, por exemplo, pode ter uma coleção de arquivos que juntos formam um livro que ele está escrevendo, uma segunda coleção contendo programas apresentados por estudantes para outro curso, um terceiro grupo contendo o código de um sistema de escrita de compiladores avançado que ele está desenvolvendo, um quarto grupo contendo propostas de doações, assim como outros arquivos para correio eletrônico, minutas de reuniões, estudos que ele está escrevendo, jogos e assim por diante.

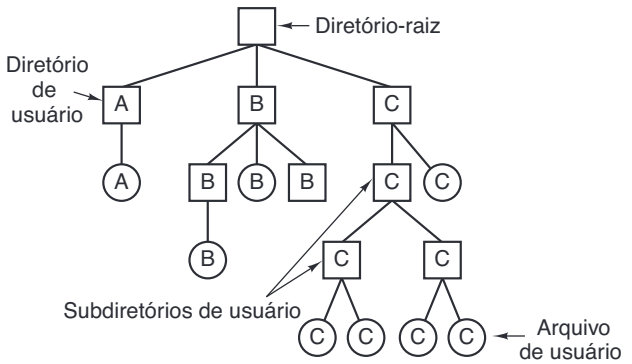
Faz-se necessária uma hierarquia (isto é, uma árvore de diretórios). Com essa abordagem, o usuário pode ter tantos diretórios quantos forem necessários para agrupar seus arquivos de maneira natural. Além disso, se múltiplos usuários compartilham um servidor de arquivos comum, como é o caso em muitas redes de empresas, cada usuário pode ter um diretório-raiz privado para sua própria hierarquia. Essa abordagem é mostrada na Figura 4.7. Aqui, cada diretório *A, B* e *C* contido no diretório-raiz pertence a um usuário diferente, e dois deles criaram subdiretórios para projetos nos quais estão trabalhando.

A capacidade dos usuários de criarem um número arbitrário de subdiretórios proporciona uma ferramenta de estruturação poderosa para eles organizarem o seu trabalho. Por essa razão, quase todos os sistemas de arquivos modernos são organizados dessa maneira.

4.2.3 Nomes de caminhos

Quando o sistema de arquivos é organizado com uma árvore de diretórios, alguma maneira é

FIGURA 4.7 Um sistema hierárquico de diretórios.



necessária para especificar os nomes dos arquivos. Dois métodos diferentes são os mais usados. No primeiro, cada arquivo recebe um **nome de caminho absoluto** consistindo no caminho do diretório-raiz para o arquivo. Como exemplo, o caminho */usr/ast/caixapostal* significa que o diretório-raiz contém um subdiretório *usr*, que por sua vez contém um subdiretório *ast*, que contém o arquivo *caixapostal*. Nomes de caminhos absolutos sempre começam no diretório-raiz e são únicos. No UNIX, os componentes do caminho são separados por /. No Windows o separador é \. No MULTICS era >. Desse modo, o mesmo nome de caminho seria escrito como a seguir nesses três sistemas:

Windows	\usr\ast\caixapostal
UNIX	/usr/ast/caixapostal
MULTICS	>usr>ast>caixapostal

Não importa qual caractere é usado, se o primeiro caractere do nome do caminho for o separador, então o caminho será absoluto.

O outro tipo é o **nome de caminho relativo**. Esse é usado em conjunção com o conceito do **diretório de trabalho** (também chamado de **diretório atual**). Um usuário pode designar um diretório como o de trabalho atual, caso em que todos os nomes de caminho não começando no diretório-raiz são presumidos como relativos ao diretório de trabalho. Por exemplo, se o diretório de trabalho atual é */usr/ast*, então o arquivo cujo caminho absoluto é */usr/ast/caixapostal* pode ser referenciado somente como *caixa postal.* Em outras palavras, o comando UNIX

```
cp /usr/ast/caixapostal /usr/ast/caixapostal.bak
```

e o comando

```
cp caixapostal caixapostal.bak
```

realizam exatamente a mesma coisa se o diretório de trabalho for */usr/ast*. A forma relativa é muitas vezes mais conveniente, mas ela faz o mesmo que a forma absoluta.

Alguns programas precisam acessar um arquivo específico sem se preocupar em saber qual é o diretório de trabalho. Nesse caso, eles devem usar sempre os nomes de caminhos absolutos. Por exemplo, um verificador ortográfico talvez precise ler */usr/lib/dictionary* para realizar esse trabalho. Nesse caso ele deve usar o nome de caminho absoluto completo, pois não sabe em qual diretório de trabalho estará quando for chamado. O nome de caminho absoluto sempre funcionará, não importa qual seja o diretório de trabalho.

É claro, se o verificador ortográfico precisar de um número grande de arquivos de */usr/lib*, uma abordagem alternativa é ele emitir uma chamada de sistema para mudar o seu diretório de trabalho para */usr/lib* e então usar apenas *dictionary* como o primeiro parâmetro para open. Ao mudar explicitamente o diretório de trabalho, o verificador sabe com certeza onde ele se situa na árvore de diretórios, assim pode então usar caminhos relativos.

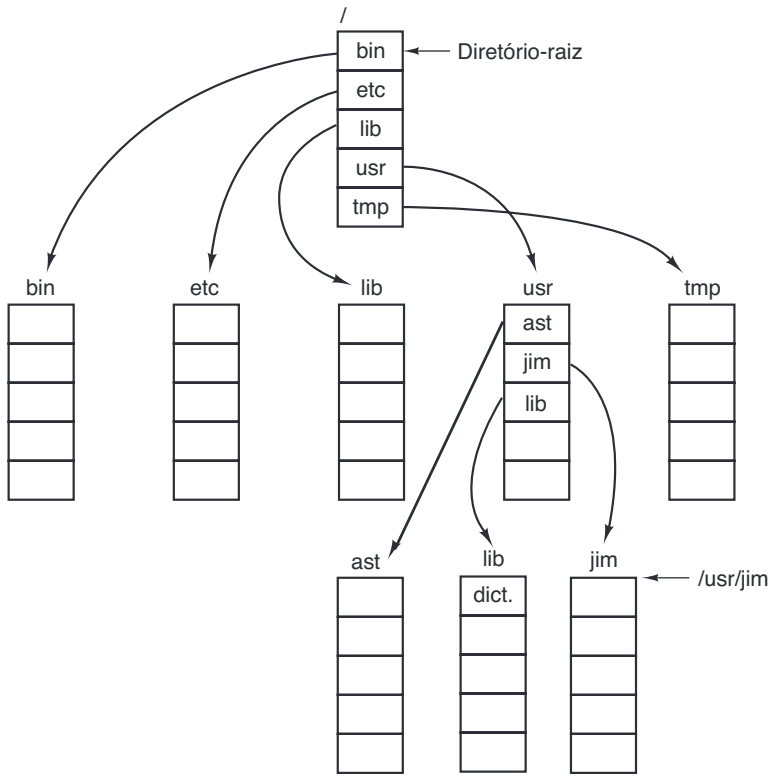
Cada processo tem seu próprio diretório de trabalho, então quando ele o muda e mais tarde sai, nenhum outro processo é afetado e nenhum traço da mudança é deixado para trás no sistema de arquivos. Dessa maneira, é sempre perfeitamente seguro para um processo mudar seu diretório de trabalho sempre que ele achar conveniente. Por outro lado, se uma *rotina de biblioteca* muda o diretório de trabalho e não volta para onde estava quando termina, o resto do programa pode não funcionar, pois sua suposição sobre onde está pode tornar-se subitamente inválida. Por essa razão, rotinas de biblioteca raramente alteram o diretório de trabalho e, quando precisam fazê-lo, elas sempre o alteram de volta antes de retornar.

A maioria dos sistemas operacionais que aceita um sistema de diretório hierárquico tem duas entradas especiais em cada diretório, "." e "..", geralmente pronunciadas como "ponto" e "pontoponto". Ponto refere-se ao diretório atual; pontoponto refere-se ao pai (exceto no diretório-raiz, onde ele refere-se a si mesmo). Para ver como essas entradas são usadas, considere a árvore de diretórios UNIX da Figura 4.8. Um determinado processo tem */usr/ast* como seu diretório de trabalho. Ele pode usar .. para subir na árvore. Por exemplo, pode copiar o arquivo */usr/lib/dictionary* para o seu próprio diretório usando o comando

```
cp ../lib/dictionary .
```

O primeiro caminho instrui o sistema a subir (para o diretório *usr*), então a descer para o diretório *lib* para encontrar o arquivo dictionary.

FIGURA 4.8 Uma árvore de diretórios UNIX.



O segundo argumento (ponto) refere-se ao diretório atual. Quando o comando *cp* recebe um nome de diretório (incluindo ponto) como seu último argumento, ele copia todos os arquivos para aquele diretório. É claro, uma maneira mais natural de realizar a cópia seria usar o nome de caminho absoluto completo do arquivo-fonte:

```
cp /usr/lib/dictionary .
```

Aqui o uso do ponto poupa o usuário do desperdício de tempo de digitar *dictionary* uma segunda vez. Mesmo assim, digitar

```
cp /usr/lib/dictionary dictionary
```

também funciona bem, assim como

```
cp /usr/lib/dictionary /usr/ast/dictionary
```

Todos esses comandos realizam exatamente a mesma coisa.

4.2.4 Operações com diretórios

As chamadas de sistema que podem gerenciar diretórios exibem mais variação de sistema para sistema do que as chamadas para gerenciar arquivos. Para dar uma impressão do que elas são e como funcionam, daremos uma amostra (tirada do UNIX).

1. Create. Um diretório é criado. Ele está vazio exceto por ponto e pontoponto, que são colocados ali automaticamente pelo sistema (ou em alguns poucos casos, pelo programa *mkdir*).
2. Delete. Um diretório é removido. Apenas um diretório vazio pode ser removido. Um diretório contendo apenas ponto e pontoponto é considerado vazio à medida que eles não podem ser removidos.
3. Opendir. Diretórios podem ser lidos. Por exemplo, para listar todos os arquivos em um diretório, um programa de listagem abre o diretório para ler os nomes de todos os arquivos que ele contém. Antes que um diretório possa ser lido, ele deve ser aberto, de maneira análoga a abrir e ler um arquivo.
4. Closedir. Quando um diretório tiver sido lido, ele será fechado para liberar espaço de tabela interno.
5. Readdir. Essa chamada retorna a próxima entrada em um diretório aberto. Antes, era possível ler diretórios usando a chamada de sistema read usual, mas essa abordagem tem a desvantagem de forçar o programador a saber e lidar com a estrutura interna de diretórios. Por outro lado, readdir sempre retorna uma entrada em um formato padrão, não importa qual das estruturas de diretório possíveis está sendo usada.
6. Rename. Em muitos aspectos, diretórios são como arquivos e podem ser renomeados da mesma maneira que eles.
7. Link. A ligação (*linking*) é uma técnica que permite que um arquivo apareça em mais de um diretório. Essa chamada de sistema especifica um arquivo existente e um nome de caminho, e cria uma ligação do arquivo existente para o nome especificado pelo caminho. Dessa maneira, o mesmo arquivo pode aparecer em múltiplos diretórios. Uma ligação desse tipo, que incrementa o contador no i-node do arquivo (para monitorar o número de entradas de diretório contendo o arquivo), às vezes é chamada de **ligação estrita** (hard link).
8. Unlink. Uma entrada de diretório é removida. Se o arquivo sendo removido estiver presente somente em um diretório (o caso normal), ele é removido do sistema de arquivos. Se ele estiver presente em múltiplos diretórios, apenas o nome do caminho especificado é removido. Os outros continuam. Em UNIX, a chamada de sistema para remover arquivos (discutida anteriormente) é, na realidade, unlink.

A lista anterior mostra as chamadas mais importantes, mas há algumas outras também, por exemplo, para gerenciar a informação de proteção associada com um diretório.

Uma variação da ideia da ligação de arquivos é a **ligação simbólica**. Em vez de ter dois nomes apontando para a mesma estrutura de dados interna representando um arquivo, um nome pode ser criado que aponte para um arquivo minúsculo que nomeia outro arquivo. Quando o primeiro é usado — aberto, por exemplo — o sistema de arquivos segue o caminho e encontra o nome no fim. Então ele começa todo o processo de localização usando o novo nome. Ligações simbólicas têm a vantagem de conseguirem atravessar as fronteiras de discos e mesmo nomear arquivos em computadores remotos. No entanto, sua implementação é de certa maneira menos eficiente do que as ligações estritas.

4.3 Implementação do sistema de arquivos

Agora chegou o momento de passar da visão do usuário do sistema de arquivos para a do implementador. Usuários estão preocupados em como os arquivos são

nomeados, quais operações são permitidas neles, como é a árvore de diretórios e questões de interface similares. Implementadores estão interessados em como os arquivos e os diretórios estão armazenados, como o espaço de disco é gerenciado e como fazer tudo funcionar de maneira eficiente e confiável. Nas seções a seguir examinaremos uma série dessas áreas para ver quais são as questões e compromissos envolvidos.

4.3.1 Esquema do sistema de arquivos

Sistemas de arquivos são armazenados em discos. A maioria dos discos pode ser dividida em uma ou mais partições, com sistemas de arquivos independentes em cada partição. O Setor 0 do disco é chamado de **MBR** (**Master Boot Record** — registro mestre de inicialização) e é usado para inicializar o computador. O fim do MBR contém a tabela de partição. Ela dá os endereços de início e fim de cada partição. Uma das partições da tabela é marcada como ativa. Quando o computador é inicializado, a BIOS lê e executa o MBR. A primeira coisa que o programa MBR faz é localizar a partição ativa, ler seu primeiro bloco, que é chamado de **bloco de inicialização**, e executá-lo. O programa no bloco de inicialização carrega o sistema operacional contido naquela partição. Por uniformidade, cada partição começa com um bloco de inicialização, mesmo que ela não contenha um sistema operacional que possa ser inicializado. Além disso, a partição poderá conter um no futuro.

Fora iniciar com um bloco de inicialização, o esquema de uma partição de disco varia bastante entre sistemas de arquivos. Muitas vezes o sistema de arquivos vai conter alguns dos itens mostrados na Figura 4.9. O primeiro é o **superbloco**. Ele contém todos os parâmetros-chave a respeito do sistema de arquivos e é lido para a memória quando o computador é inicializado ou o sistema de arquivos é tocado pela primeira vez. Informações típicas no superbloco incluem um número mágico para identificar o tipo de sistema de arquivos, seu número de blocos e outras informações administrativas fundamentais.

Em seguida podem vir informações a respeito de blocos disponíveis no sistema de arquivos, na forma de um mapa de bits ou de uma lista de ponteiros, por exemplo. Isso pode ser seguido pelos i-nodes, um arranjo de estruturas de dados, um por arquivo, dizendo tudo sobre ele. Depois pode vir o diretório-raiz, que contém o topo da árvore do sistema de arquivos. Por fim, o restante do disco contém todos os outros diretórios e arquivos.

4.3.2 Implementando arquivos

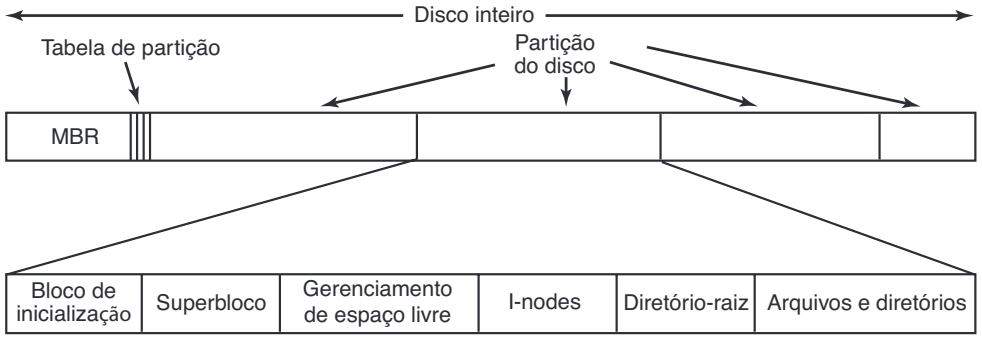
É provável que a questão mais importante na implementação do armazenamento de arquivos seja controlar quais blocos de disco vão com quais arquivos. Vários métodos são usados em diferentes sistemas operacionais. Nesta seção, examinaremos alguns deles.

Alocação contígua

O esquema de alocação mais simples é armazenar cada arquivo como uma execução contígua de blocos de disco. Assim, em um disco com blocos de 1 KB, um arquivo de 50 KB seria alocado em 50 blocos consecutivos. Com blocos de 2 KB, ele seria alocado em 25 blocos consecutivos.

Vemos um exemplo de alocação em armazenamento contíguo na Figura 4.10(a). Aqui os primeiros 40 blocos de disco são mostrados, começando com o bloco 0 à esquerda. De início, o disco estava vazio. Então um arquivo *A*, de quatro blocos de comprimento, foi escrito a partir do início (bloco 0). Após isso, um arquivo de seis blocos, *B*, foi escrito começando logo depois do fim do arquivo *A*.

FIGURA 4.9 Um esquema possível para um sistema de arquivos.



Observe que cada arquivo começa no início de um bloco novo; portanto, se o arquivo *A* realmente ocupar 3½ blocos, algum espaço será desperdiçado ao fim de cada último bloco. Na figura, um total de sete arquivos é mostrado, cada um começando no bloco seguinte ao final do anterior. O sombreamento é usado apenas para tornar mais fácil a distinção entre os blocos. Não tem significado real em termos de armazenamento.

A alocação de espaço de disco contíguo tem duas vantagens significativas. Primeiro, ela é simples de implementar porque basta se lembrar de dois números para monitorar onde estão os blocos de um arquivo: o endereço em disco do primeiro bloco e o número de blocos no arquivo. Dado o número do primeiro bloco, o número de qualquer outro bloco pode ser encontrado mediante uma simples adição.

Segundo, o desempenho da leitura é excelente, pois o arquivo inteiro pode ser lido do disco em uma única operação. Apenas uma busca é necessária (para o primeiro bloco). Depois, não são mais necessárias buscas ou atrasos rotacionais, então os dados são lidos com a capacidade total do disco. Portanto, a alocação contígua é simples de implementar e tem um alto desempenho.

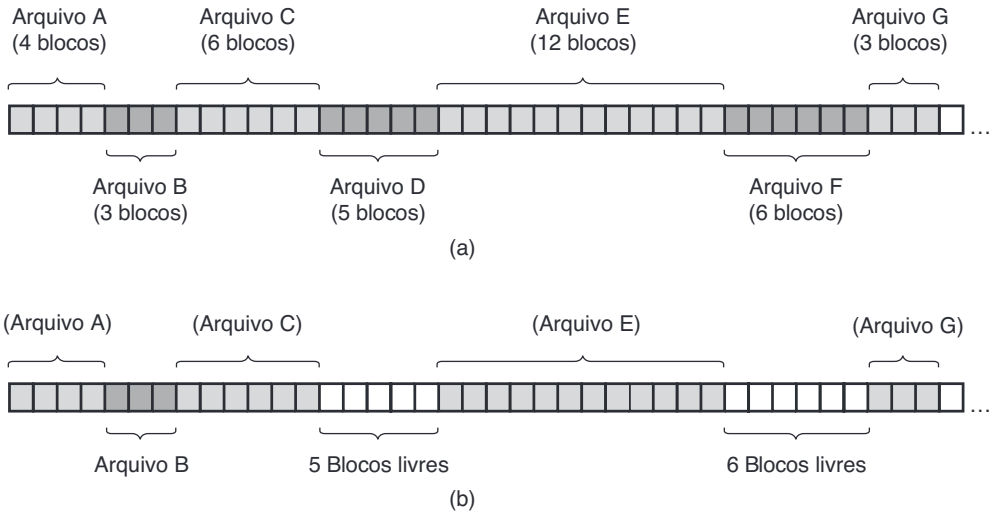
Infelizmente, a alocação contígua tem um ponto fraco importante: com o tempo, o disco torna-se fragmentado. Para ver como isso acontece, examine a Figura 4.10(b). Aqui dois arquivos, *D* e *F*, foram removidos. Quando um arquivo é removido, seus blocos são naturalmente liberados, deixando uma lacuna de blocos livres no disco. O disco não é compactado imediatamente para eliminá-la, já que isso envolveria copiar todos os blocos seguindo essa lacuna, potencialmente milhões de blocos, o que levaria horas ou mesmo dias em discos grandes. Como resultado, em última análise o disco consiste em arquivos e lacunas, como ilustrado na figura.

De início, essa fragmentação não é problema, já que cada novo arquivo pode ser escrito ao final do disco, seguindo o anterior. No entanto, finalmente o disco estará cheio e será necessário compactá-lo, o que tem custo proibitivo, ou reutilizar os espaços livres nas lacunas. Reutilizar o espaço exige manter uma lista de lacunas, o que é possível. No entanto, quando um arquivo novo vai ser criado, é necessário saber o seu tamanho final a fim de escolher uma lacuna do tamanho correto para alocá-lo.

Imagine as consequências de um projeto desses. O usuário inicializa um processador de texto a fim de criar um documento. A primeira coisa que o programa pergunta é quantos bytes o documento final terá. A pergunta deve ser respondida ou o programa não continuará. Se o número em última análise provar-se pequeno demais, o programa precisará ser terminado prematuramente, pois a lacuna do disco estará cheia e não haverá lugar para colocar o resto do arquivo. Se o usuário tentar evitar esse problema dando um número irrealisticamente grande como o tamanho final, digamos, 1 GB, o editor talvez não consiga encontrar uma lacuna tão grande e anunciará que o arquivo não pode ser criado. É claro, o usuário estaria livre para inicializar o programa novamente e dizer 500 MB dessa vez, e assim por diante até que uma lacuna adequada fosse localizada. Ainda assim, é pouco provável que esse esquema deixe os usuários felizes.

No entanto, há uma situação na qual a alocação contígua é possível e, na realidade, ainda usada: em CD-ROMs. Aqui todos os tamanhos de arquivos são

FIGURA 4.10 (a) Alocação contígua de espaço de disco para sete arquivos. (b) O estado do disco após os arquivos D e F terem sido removidos.



conhecidos antecipadamente e jamais mudarão durante o uso subsequente do sistema de arquivos do CD-ROM.

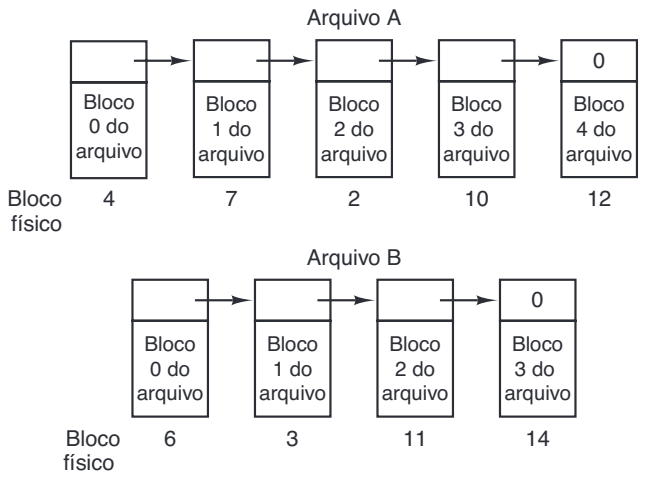
A situação com DVDs é um pouco mais complicada. Em princípio, um filme de 90 minutos poderia ser codificado como um único arquivo de comprimento de cerca de 4,5 GB, mas o sistema de arquivos utilizado, **UDF** (**Universal Disk Format** — formato universal de disco), usa um número de 30 bits para representar o tamanho do arquivo, o que limita os arquivos a 1 GB. Em consequência, filmes em DVD são em geral armazenados contiguamente como três ou quatro arquivos de 1 GB. Esses pedaços físicos do único arquivo lógico (o filme) são chamados de **extensões**.

Como mencionamos no Capítulo 1, a história muitas vezes se repete na ciência de computadores à medida que surgem novas gerações de tecnologia. A alocação contígua na realidade foi usada nos sistemas de arquivos de discos magnéticos anos atrás pela simplicidade e alto desempenho (a facilidade de uso para o usuário não contava muito à época). Então a ideia foi abandonada por causa do incômodo de ter de especificar o tamanho final do arquivo no momento de sua criação. Mas com o advento dos CD-ROMs, DVDs, Blu-rays e outras mídias óticas para escrita única, subitamente arquivos contíguos eram uma boa ideia de novo. Desse modo, é importante estudar sistemas e ideias antigas que eram conceitualmente limpas e simples, pois elas podem ser aplicáveis a sistemas futuros de maneiras surpreendentes.

Alocação por lista encadeada

O segundo método para armazenar arquivos é manter cada um como uma lista encadeada de blocos de disco, como mostrado na Figura 4.11. A primeira palavra de cada bloco é usada como um ponteiro para a próxima. O resto do bloco é reservado para dados.

FIGURA 4.11 Armazenando um arquivo como uma lista encadeada de blocos de disco.



Diferentemente da alocação contígua, todos os blocos do disco podem ser usados nesse método. Nenhum espaço é perdido para a fragmentação de disco (exceto para a fragmentação interna no último bloco). Também, para a entrada de diretório é suficiente armazenar meramente o endereço em disco do primeiro bloco. O resto pode ser encontrado a partir daí.

Por outro lado, embora a leitura de um arquivo sequencialmente seja algo direto, o acesso aleatório é de extrema lentidão. Para chegar ao bloco n, o sistema operacional precisa começar do início e ler os blocos $n - 1$ antes dele, um de cada vez. É claro que realizar tantas leituras será algo dolorosamente lento.

Também, a quantidade de dados que um bloco pode armazenar não é mais uma potência de dois, pois os ponteiros ocupam alguns bytes do bloco. Embora não seja fatal, ter um tamanho peculiar é menos eficiente, pois muitos programas leem e escrevem em blocos cujo tamanho é uma potência de dois. Com os primeiros bytes de cada bloco ocupados por um ponteiro para o próximo bloco, a leitura de todo o bloco exige que se adquira e concatene a informação de dois blocos de disco, o que gera uma sobrecarga extra por causa da cópia.

Alocação por lista encadeada usando uma tabela na memória

Ambas as desvantagens da alocação por lista encadeada podem ser eliminadas colocando-se as palavras do ponteiro de cada bloco de disco em uma tabela na memória. A Figura 4.12 mostra como são as tabelas para o exemplo da Figura 4.11. Em ambas, temos dois arquivos. O arquivo *A* usa os blocos de disco 4, 7, 2, 10 e 12, nessa ordem, e o arquivo *B* usa os blocos de disco 6, 3, 11 e 14, nessa ordem. Usando a tabela da Figura 4.12, podemos começar com o bloco 4 e seguir a cadeia até o fim. O mesmo pode ser feito começando com o bloco 6. Ambos os encadeamentos são concluídos com uma marca especial (por exemplo, −1) que corresponde a um número de bloco inválido. Essa tabela na memória principal é chamada de **FAT** (**File Allocation Table** — tabela de alocação de arquivos).

Usando essa organização, o bloco inteiro fica disponível para dados. Além disso, o acesso aleatório é muito mais fácil. Embora ainda seja necessário seguir o encadeamento para encontrar um determinado deslocamento dentro do arquivo, o encadeamento está inteiramente na memória, portanto ele pode ser seguido sem fazer quaisquer referências ao disco. Da mesma maneira que no

FIGURA 4.12 Alocação por lista encadeada usando uma tabela de alocação de arquivos na memória principal.

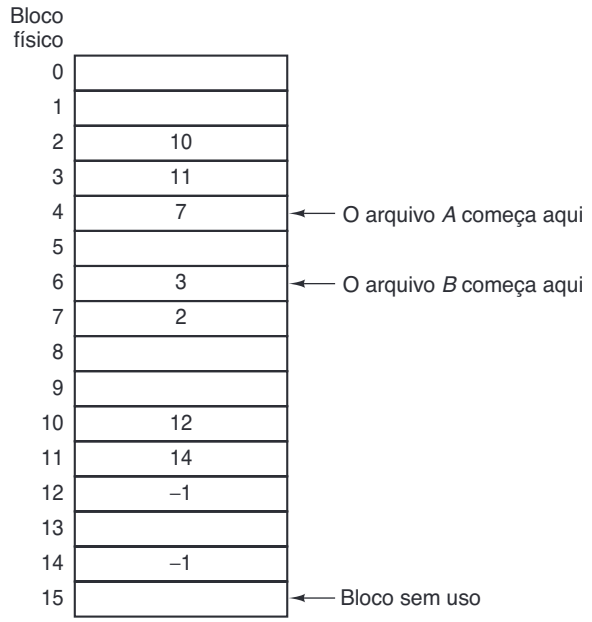
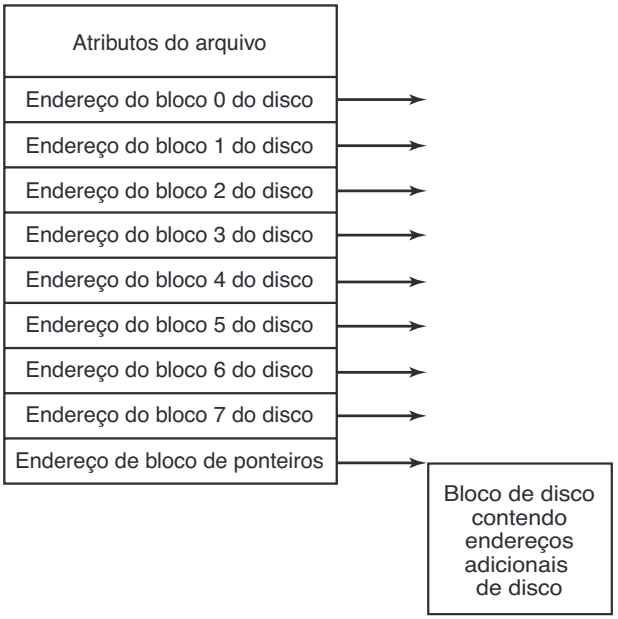


FIGURA 4.13 Um exemplo de i-node.



método anterior, é suficiente para a entrada de diretório manter um único inteiro (o número do bloco inicial) e ainda assim ser capaz de localizar todos os blocos, não importa o tamanho do arquivo.

A principal desvantagem desse método é que a tabela inteira precisa estar na memória o todo o tempo para fazê-la funcionar. Com um disco de 1 TB e um tamanho de bloco de 1 KB, a tabela precisa de 1 bilhão de entradas, uma para cada um dos 1 bilhão de blocos de disco. Cada entrada precisa ter no mínimo 3 bytes. Para aumentar a velocidade de consulta, elas deveriam ter 4 bytes. Desse modo, a tabela ocupará 3 GB ou 2,4 GB da memória principal o tempo inteiro, dependendo de o sistema estar otimizado para espaço ou tempo. Não é algo muito prático. Claro, a ideia da FAT não se adapta bem para discos grandes. Era o sistema de arquivos MS-DOS original e ainda é aceito completamente por todas as versões do Windows.

I-nodes

Nosso último método para monitorar quais blocos pertencem a quais arquivos é associar cada arquivo a uma estrutura de dados chamada de **i-node** (**index-node** — nó-índice), que lista os atributos e os endereços de disco dos blocos do disco. Um exemplo simples é descrito na Figura 4.13. Dado o i-node, é então possível encontrar todos os blocos do arquivo. A grande vantagem desse esquema sobre os arquivos encadeados usando uma tabela na memória é que o i-node precisa estar na memória apenas quando o arquivo correspondente estiver aberto. Se cada i-node ocupa n bytes e um máximo de k arquivos puderem estar abertos simultaneamente, a memória total ocupada pelo arranjo contendo os i-nodes para os arquivos abertos é de apenas kn bytes. Apenas essa quantidade de espaço precisa ser reservada antecipadamente.

Esse arranjo é em geral muito menor do que o espaço ocupado pela tabela de arquivos descrita na seção anterior. A razão é simples. A tabela para conter a lista encadeada de todos os blocos de disco é proporcional em tamanho ao disco em si. Se o disco tem n blocos, a tabela precisa de n entradas. À medida que os discos ficam maiores, essa tabela cresce linearmente com eles. Por outro lado, o esquema i-node exige um conjunto na memória cujo tamanho seja proporcional ao número máximo de arquivos que podem ser abertos ao mesmo tempo. Não importa que o disco tenha 100 GB, 1.000 GB ou 10.000 GB.

Um problema com i-nodes é que se cada um tem espaço para um número fixo de endereços de disco, o que acontece quando um arquivo cresce além de seu limite? Uma solução é reservar o último endereço de disco não para um bloco de dados, mas, em vez disso, para o endereço de um bloco contendo mais endereços de blocos de disco, como mostrado na Figura 4.13. Mais avançado ainda seria ter dois ou mais desses blocos contendo endereços de disco ou até blocos de disco apontando para outros blocos cheios de endereços. Voltaremos aos i-nodes quando estudarmos o UNIX no Capítulo 10.

De modo similar, o sistema de arquivos NTFS do Windows usa uma ideia semelhante, apenas com i-nodes maiores, que também podem conter arquivos pequenos.

4.3.3 Implementando diretórios

Antes que um arquivo possa ser lido, ele precisa ser aberto. Quando um arquivo é aberto, o sistema operacional usa o nome do caminho fornecido pelo usuário para localizar a entrada de diretório no disco. A entrada de diretório fornece a informação necessária para encontrar os blocos de disco. Dependendo do sistema, essa informação pode ser o endereço de disco do arquivo inteiro (com alocação contígua), o número do primeiro bloco (para ambos os esquemas de listas encadeadas), ou o número do i-node. Em todos os casos, a principal função do sistema de diretórios é mapear o nome do arquivo em ASCII na informação necessária para localizar os dados.

Uma questão relacionada de perto refere-se a onde os atributos devem ser armazenados. Todo sistema de arquivos mantém vários atributos do arquivo, como o proprietário de cada um e seu momento de criação, e eles devem ser armazenados em algum lugar. Uma possibilidade óbvia é fazê-lo diretamente na entrada do diretório. Alguns sistemas fazem precisamente isso. Essa opção é mostrada na Figura 4.14(a). Nesse design simples, um diretório consiste em uma lista de entradas de tamanho fixo, um por arquivo, contendo um nome de arquivo (de tamanho fixo), uma estrutura dos atributos do arquivo e um ou mais endereços de disco (até algum máximo) dizendo onde estão os blocos de disco.

Para sistemas que usam i-nodes, outra possibilidade para armazenar os atributos é nos próprios i-nodes, em vez de nas entradas do diretório. Nesse caso, a entrada do diretório pode ser mais curta: apenas um nome de arquivo e um número de i-node. Essa abordagem está ilustrada na Figura 4.14(b). Como veremos mais tarde, esse método tem algumas vantagens sobre colocá-los na entrada do diretório.

Até o momento presumimos que os arquivos têm nomes curtos de tamanho fixo. No MS-DOS, os arquivos têm um nome base de 1-8 caracteres e uma extensão opcional de 1-3 caracteres. Na Versão 7 do UNIX, os nomes dos arquivos tinham 1-14 caracteres, incluindo quaisquer extensões. No entanto, quase todos os sistemas operacionais modernos aceitam nomes de arquivos maiores e de tamanho variável. Como eles podem ser implementados?

A abordagem mais simples é estabelecer um limite para o tamanho do nome dos arquivos e então usar um dos designs da Figura 4.14 com 255 caracteres reservados para cada nome de arquivo. Essa abordagem é simples, mas desperdiça muito espaço de diretório, já que poucos arquivos têm nomes tão longos. Por razões de eficiência, uma estrutura diferente é desejável.

Uma alternativa é abrir mão da ideia de que todas as entradas de diretório sejam do mesmo tamanho. Com esse método, cada entrada de diretório contém uma porção fixa, começando com o tamanho da entrada e, então, seguido por dados com um formato fixo, normalmente incluindo o proprietário, momento de criação, informações de proteção e outros atributos. Esse cabeçalho de comprimento fixo é seguido pelo nome do arquivo real, não importa seu tamanho, como mostrado na Figura 4.15(a) em um formato em que o byte mais significativo aparece primeiro (*big-endian*) — SPARC, por exemplo. Nesse exemplo, temos três arquivos, *project-budget, personnel* e *foo*. Cada nome de arquivo é concluído com um caractere especial (em geral 0), que é representado na figura por um quadrado com um "X" dentro. Para permitir que cada entrada de diretório comece junto ao limite de uma palavra, cada nome de arquivo é preenchido de modo a completar um número inteiro de palavras, indicado pelas caixas sombreadas na figura.

FIGURA 4.14 (a) Um diretório simples contendo entradas de tamanho fixo com endereços de disco e atributos na entrada do diretório. (b) Um diretório no qual cada entrada refere-se a apenas um i-node.

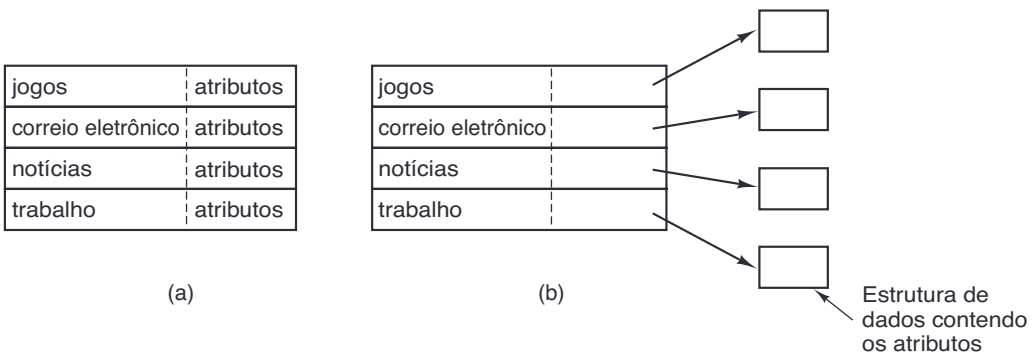
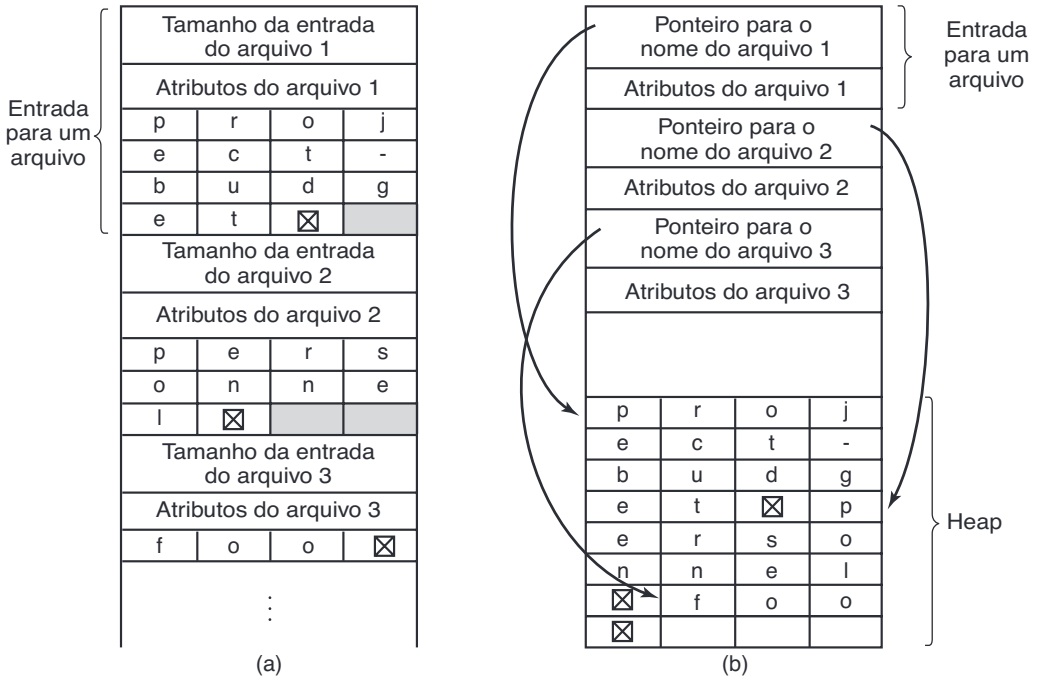


FIGURA 4.15 Duas maneiras de gerenciar nomes de arquivos longos em um diretório. (a) Sequencialmente. (b) No heap.



Uma desvantagem desse método é que, quando um arquivo é removido, uma lacuna de tamanho variável é introduzida no diretório e o próximo arquivo a entrar poderá não caber nela. Esse problema é na essência o mesmo que vimos com arquivos de disco contíguos, apenas agora é possível compactar o diretório, pois ele está inteiramente na memória. Outro problema é que uma única entrada de diretório pode se estender por múltiplas páginas, de maneira que uma falta de página pode ocorrer durante a leitura de um nome de arquivo.

Outra maneira de lidar com nomes de tamanhos variáveis é tornar fixos os tamanhos das próprias entradas de diretório e manter os nomes dos arquivos em um heap (monte) no fim de cada diretório, como mostrado na Figura 4.15(b). Esse método tem a vantagem de que, quando uma entrada for removida, o arquivo seguinte inserido sempre caberá ali. É claro, o heap deve ser gerenciado e faltas de páginas ainda podem ocorrer enquanto processando nomes de arquivos. Um ganho menor aqui é que não há mais nenhuma necessidade real para que os nomes dos arquivos comecem junto aos limites das palavras, de maneira que não é mais necessário completar os nomes dos arquivos com caracteres na Figura 4.15(b) como eles são na Figura 4.15(a).

Em todos os projetos apresentados até o momento, os diretórios são pesquisados linearmente do início ao fim quando o nome de um arquivo precisa ser procurado. Para diretórios extremamente longos, a busca linear pode ser lenta. Uma maneira de acelerar a busca é usar uma tabela de espalhamento em cada diretório. Defina o tamanho da tabela n. Ao entrar com um nome de arquivo, o nome é mapeado em um valor entre 0 e $n - 1$, por exemplo, dividindo-o por n e tomando-se o resto. Alternativamente, as palavras compreendendo o nome do arquivo podem ser somadas e essa quantidade dividida por n, ou algo similar.

De qualquer maneira, a entrada da tabela correspondendo ao código de espalhamento é verificada. Entradas de arquivo seguem a tabela de espalhamento. Se aquela vaga já estiver em uso, uma lista encadeada é construída, inicializada naquela entrada da tabela e unindo todas as entradas com o mesmo valor de espalhamento.

A procura por um arquivo segue o mesmo procedimento. O nome do arquivo é submetido a uma função de espalhamento para selecionar uma entrada da tabela de espalhamento. Todas as entradas da lista encadeada inicializada naquela vaga são verificadas para ver se o nome do arquivo está presente. Se o nome não estiver na lista, o arquivo não está presente no diretório.

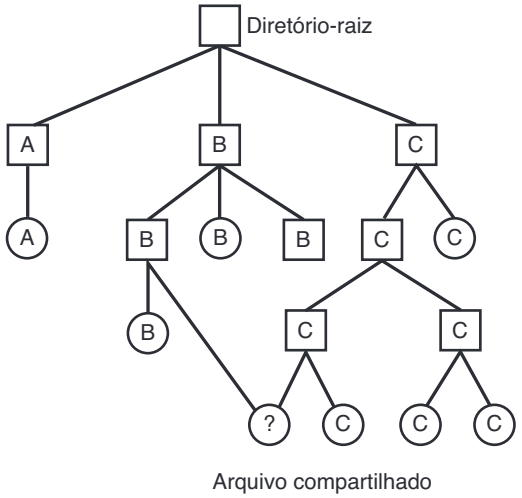
Usar uma tabela de espalhamento tem a vantagem de uma busca muito mais rápida, mas a desvantagem de uma administração mais complexa. Ela é uma alternativa realmente séria apenas em sistemas em que é esperado que os diretórios contenham de modo rotineiro centenas ou milhares de arquivos.

Uma maneira diferente de acelerar a busca em grandes diretórios é colocar os resultados em uma cache de buscas. Antes de começar uma busca, é feita primeiro uma verificação para ver se o nome do arquivo está na cache. Se estiver, ele pode ser localizado de imediato. É claro, a cache só funciona se um número relativamente pequeno de arquivos compreender a maioria das buscas.

4.3.4 Arquivos compartilhados

Quando vários usuários estão trabalhando juntos em um projeto, eles muitas vezes precisam compartilhar arquivos. Em consequência, muitas vezes é conveniente que um arquivo compartilhado apareça simultaneamente em diretórios diferentes pertencendo a usuários distintos. A Figura 4.16 mostra o sistema de arquivos da Figura 4.7 novamente, apenas com um dos arquivos do usuário *C* agora presente também em um dos diretórios do usuário *B*. A conexão entre o diretório do usuário *B* e o arquivo compartilhado é chamada de **ligação**. O sistema de arquivos em si é agora um **Gráfico Acíclico Orientado** (**Directed Acyclic Graph — DAG**), em vez de uma árvore. Ter o sistema de arquivos como um DAG complica a manutenção, mas a vida é assim.

FIGURA 4.16 Sistema de arquivos contendo um arquivo compartilhado.



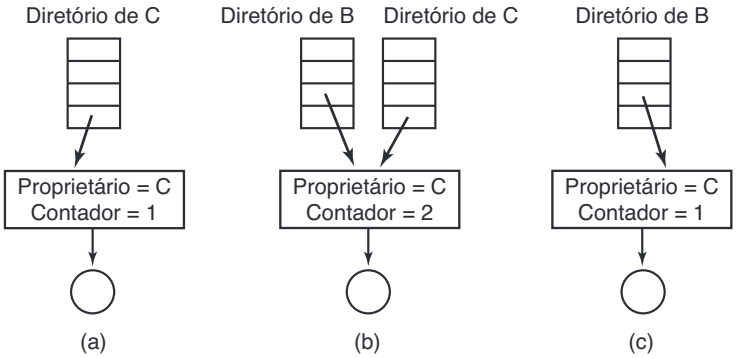
Compartilhar arquivos é conveniente, mas também apresenta alguns problemas. Para começo de conversa, se os diretórios realmente contiverem endereços de disco, então uma cópia desses endereços terá de ser feita no diretório do usuário *B* quando o arquivo for ligado. Se *B* ou *C* subsequentemente adicionarem blocos ao arquivo, os novos blocos serão listados somente no diretório do usuário que estiver realizando a adição. As mudanças não serão visíveis ao outro usuário, derrotando então o propósito do compartilhamento.

Esse problema pode ser solucionado de duas maneiras. Na primeira solução, os blocos de disco não são listados em diretórios, mas em uma pequena estrutura de dados associada com o arquivo em si. Os diretórios apontariam então apenas para a pequena estrutura de dados. Essa é a abordagem usada em UNIX (em que a pequena estrutura de dados é o i-node).

Na segunda solução, *B* se liga a um dos arquivos de *C*, obrigando o sistema a criar um novo arquivo do tipo LINK e a inseri-lo no diretório de *B*. O novo arquivo contém apenas o nome do caminho do arquivo para o qual ele está ligado. Quando *B* lê do arquivo ligado, o sistema operacional vê que o arquivo sendo lido é do tipo LINK, verifica seu nome e o lê. Essa abordagem é chamada de **ligação simbólica**, para contrastar com a ligação (estrita) tradicional.

Cada um desses métodos tem seus problemas. No primeiro, no momento em que *B* se liga com o arquivo compartilhado, o i-node grava o proprietário do arquivo como *C*. Criar uma ligação não muda a propriedade (ver Figura 4.17), mas aumenta o contador de ligações no i-node, então o sistema sabe quantas entradas de diretório apontam no momento para o arquivo.

FIGURA 4.17 (a) Situação antes da ligação. (b) Depois da criação da ligação. (c) Depois que o proprietário original remove o arquivo.



Se *C* subsequentemente tentar remover o arquivo, o sistema se vê diante de um dilema. Se remover o arquivo e limpar o i-node, *B* terá uma entrada de diretório apontando para um i-node inválido. Se o i-node for transferido mais tarde para outro arquivo, a ligação de *B* apontará para o arquivo errado. O sistema pode avaliar, a partir do contador no i-node, que o arquivo ainda está em uso, mas não há uma maneira fácil de encontrar todas as entradas de diretório para o arquivo a fim de removê-las. Ponteiros para os diretórios não podem ser armazenados no i-node, pois pode haver um número ilimitado de diretórios.

A única coisa a fazer é remover a entrada de diretório de *C*, mas deixar o i-node intacto, com o contador em 1, como mostrado na Figura 4.17(c). Agora temos uma situação na qual *B* é o único usuário com uma entrada de diretório para um arquivo cujo proprietário é *C*. Se o sistema fizer contabilidade ou tiver cotas, *C* continuará pagando a conta pelo arquivo até que *B* decida removê-lo. Se *B* o fizer, nesse momento o contador vai para 0 e o arquivo é removido.

Com ligações simbólicas esse problema não surge, pois somente o verdadeiro proprietário tem um ponteiro para o i-node. Os usuários que têm ligações para o arquivo possuem apenas nomes de caminhos, não ponteiros de i-node. Quando o *proprietário* remove o arquivo, ele é destruído. Tentativas subsequentes de usar o arquivo por uma ligação simbólica fracassarão quando o sistema for incapaz de localizá-lo. Remover uma ligação simbólica não afeta o arquivo de maneira alguma.

O problema com ligações simbólicas é a sobrecarga extra necessária. O arquivo contendo o caminho deve ser lido, então ele deve ser analisado e seguido, componente a componente, até que o i-node seja alcançado. Toda essa atividade pode exigir um número considerável de acessos adicionais ao disco. Além disso, um i-node extra é necessário para cada ligação simbólica, assim como um bloco de disco extra para armazenar o caminho, embora se o nome do caminho for curto, o sistema poderá armazená-lo no próprio i-node, como um tipo de otimização. Ligações simbólicas têm a vantagem de poderem ser usadas para ligar os arquivos em máquinas em qualquer parte no mundo, simplesmente fornecendo o endereço de rede da máquina onde o arquivo reside, além de seu caminho naquela máquina.

Há também outro problema introduzido pelas ligações, simbólicas ou não. Quando as ligações são permitidas, os arquivos podem ter dois ou mais caminhos. Programas que inicializam em um determinado diretório e encontram todos os arquivos naquele diretório e seus subdiretórios, localizarão um arquivo ligado múltiplas vezes. Por exemplo, um programa que salva todos os arquivos de um diretório e seus subdiretórios em uma fita poderá fazer múltiplas cópias de um arquivo ligado. Além disso, se a fita for lida então em outra máquina, a não ser que o programa que salva para a fita seja inteligente, o arquivo ligado será copiado duas vezes para o disco, em vez de ser ligado.

4.3.5 Sistemas de arquivos estruturados em diário (log)

Mudanças na tecnologia estão pressionando os sistemas de arquivos atuais. Em particular, CPUs estão ficando mais rápidas, discos tornam-se muito maiores e baratos (mas não muito mais rápidos), e as memórias crescem exponencialmente em tamanho. O único parâmetro que não está se desenvolvendo de maneira tão acelerada é o tempo de busca dos discos (exceto para discos em estado sólido, que não têm tempo de busca).

A combinação desses fatores significa que um gargalo de desempenho está surgindo em muitos sistemas de arquivos. Pesquisas realizadas em Berkeley tentaram minimizar esse problema projetando um tipo completamente novo de sistema de arquivos, o **LFS (Log-structured File System** — sistema de arquivos estruturado em diário). Nesta seção, descreveremos brevemente como o LFS funciona. Para uma abordagem mais completa, ver o estudo original sobre LFS (ROSENBLUM e OUSTERHOUT, 1991).

A ideia que impeliu o design do LFS é de que à medida que as CPUs ficam mais rápidas e as memórias RAM maiores, caches em disco também estão aumentando rapidamente. Em consequência, agora é possível satisfazer uma fração muito substancial de todas as solicitações de leitura diretamente da cache do sistema de arquivos, sem a necessidade de um acesso de disco. Segue dessa observação que, no futuro, a maior parte dos acessos ao disco será para escrita, então o mecanismo de leitura antecipada usado em alguns sistemas de arquivos para buscar blocos antes que eles sejam necessários não proporciona mais um desempenho significativo.

Para piorar as coisas, na maioria dos sistemas de arquivos, as operações de escrita são feitas em pedaços muito pequenos. Escritas pequenas são altamente ineficientes, dado que uma escrita em disco de 50 μs muitas vezes é precedida por uma busca de 10 ms e um atraso rotacional de 4 ms. Com esses parâmetros, a eficiência dos discos cai para uma fração de 1%.

A fim de entender de onde vêm todas essas pequenas operações de escrita, considere criar um arquivo

novo em um sistema UNIX. Para escrever esse arquivo, o i-node para o diretório, o bloco do diretório, o i-node para o arquivo e o próprio arquivo devem ser todos escritos. Embora essas operações possam ser postergadas, fazê-lo expõe o sistema de arquivos a sérios problemas de consistência se uma queda no sistema ocorrer antes que as escritas tenham sido concluídas. Por essa razão, as escritas de i-node são, em geral, feitas imediatamente.

A partir desse raciocínio, os projetistas do LFS decidiram reimplementar o sistema de arquivos UNIX de maneira que fosse possível utilizar a largura total da banda do disco, mesmo diante de uma carga de trabalho consistindo em grande parte de pequenas operações de escrita aleatórias. A ideia básica é estruturar o disco inteiro como um grande diário (log).

De modo periódico, e quando há uma necessidade especial para isso, todas as operações de escrita pendentes armazenadas na memória são agrupadas em um único segmento e escritas para o disco como um único segmento contíguo ao fim do diário. Desse modo, um único segmento pode conter i-nodes, blocos de diretório e blocos de dados, todos misturados. No começo de cada segmento há um resumo do segmento, dizendo o que pode ser encontrado nele. Se o segmento médio puder ser feito com o tamanho de cerca de 1 MB, quase toda a largura de banda de disco poderá ser utilizada.

Neste projeto, i-nodes ainda existem e têm até a mesma estrutura que no UNIX, mas estão dispersos agora por todo o diário, em vez de ter uma posição fixa no disco. Mesmo assim, quando um i-node é localizado, a localização dos blocos acontece da maneira usual. É claro, encontrar um i-node é muito mais difícil agora, já que seu endereço não pode ser simplesmente calculado a partir do seu i-número, como no UNIX. Para tornar possível encontrar i-nodes, é mantido um mapa do i-node, indexado pelo i-número. O registro *i* nesse mapa aponta para o i-node *i* no disco. O mapa fica armazenado no disco, e também é mantido em cache, de maneira que as partes mais intensamente usadas estarão na memória a maior parte do tempo.

Para resumir o que dissemos até o momento, todas as operações de escrita são inicialmente armazenadas na memória, e periodicamente todas as operações de escrita armazenadas são escritas para o disco em um único segmento, ao final do diário. Abrir um arquivo agora consiste em usar o mapa para localizar o i-node para o arquivo. Uma vez que o i-node tenha sido localizado, os endereços dos blocos podem ser encontrados a partir dele. Todos os blocos em si estarão em segmentos, em alguma parte no diário.

Se discos fossem infinitamente grandes, a descrição anterior daria conta de toda a história. No entanto, discos reais são finitos, então finalmente o diário ocupará o disco inteiro, momento em que nenhum segmento novo poderá ser escrito para o diário. Felizmente, muitos segmentos existentes podem ter blocos que não são mais necessários. Por exemplo, se um arquivo for sobrescrito, seu i-node apontará então para os blocos novos, mas os antigos ainda estarão ocupando espaço em segmentos escritos anteriormente.

Para lidar com esse problema, o LFS tem um thread **limpador** que passa o seu tempo escaneando o diário circularmente para compactá-lo. Ele começa lendo o resumo do primeiro segmento no diário para ver quais i-nodes e arquivos estão ali. Então confere o mapa do i-node atual para ver se os i-nodes ainda são atuais e se os blocos de arquivos ainda estão sendo usados. Em caso negativo, essa informação é descartada. Os i-nodes e blocos que ainda estão sendo usados vão para a memória para serem escritos no próximo segmento. O segmento original é então marcado como disponível, de maneira que o arquivo pode usá-lo para novos dados. Dessa maneira, o limpador se movimenta ao longo do diário, removendo velhos segmentos do final e colocando quaisquer dados ativos na memória para serem reescritos no segmento seguinte. Em consequência, o disco é um grande buffer circular, com o thread de escrita adicionando novos segmentos ao início e o thread limpador removendo os antigos do final.

Aqui o sistema de registro não é trivial, visto que, quando um bloco de arquivo é escrito de volta para um novo segmento, o i-node do arquivo (em alguma parte no diário) deve ser localizado, atualizado e colocado na memória para ser escrito no segmento seguinte. O mapa do i-node deve então ser atualizado para apontar para a cópia nova. Mesmo assim, é possível fazer a administração, e os resultados do desempenho mostram que toda essa complexidade vale a pena. As medidas apresentadas nos estudos citados mostram que o LFS supera o UNIX em desempenho por uma ordem de magnitude em escritas pequenas, enquanto tem um desempenho que é tão bom quanto, ou melhor que o UNIX para leituras e escritas grandes.

4.3.6 Sistemas de arquivos journaling

Embora os sistemas de arquivos estruturados em diário sejam uma ideia interessante, eles não são tão usados, em parte por serem altamente incompatíveis com os sistemas de arquivos existentes. Mesmo assim, uma das ideias inerentes a eles, a robustez diante de

falhas, pode ser facilmente aplicada a sistemas de arquivos mais convencionais. A ideia básica aqui é manter um diário do que o sistema de arquivos vai fazer antes que ele o faça; então, se o sistema falhar antes que ele possa fazer seu trabalho planejado, ao ser reinicializado, ele pode procurar no diário para ver o que acontecia no momento da falha e concluir o trabalho. Esse tipo de sistema de arquivos, chamado de **sistemas de arquivos journaling**, já está em uso na realidade. O sistema de arquivos NTFS da Microsoft e os sistemas de arquivos Linux ext3 e ReiserFS todos usam journaling. O OS X oferece sistemas de arquivos journaling como uma opção. A seguir faremos uma breve introdução a esse tópico.

Para ver a natureza do problema, considere uma operação corriqueira simples que acontece todo o tempo: remover um arquivo. Essa operação (no UNIX) exige três passos:

1. Remover o arquivo do seu diretório.
2. Liberar o i-node para o conjunto de i-nodes livres.
3. Retornar todos os blocos de disco para o conjunto de blocos de disco livres.

No Windows, são exigidos passos análogos. Na ausência de falhas do sistema, a ordem na qual esses passos são dados não importa; na presença de falhas, ela importa. Suponha que o primeiro passo tenha sido concluído e então haja uma falha no sistema. O i-node e os blocos de arquivos não serão acessíveis a partir de arquivo algum, mas também não serão acessíveis para realocação; eles apenas estarão em algum limbo, diminuindo os recursos disponíveis. Se a falha ocorrer após o segundo passo, apenas os blocos serão perdidos.

Se a ordem das operações for mudada e o i-node for liberado primeiro, então após a reinicialização, o i-node poderá ser realocado, mas a antiga entrada de diretório continuará apontando para ele, portanto para o arquivo errado. Se os blocos forem liberados primeiro, então uma falha antes de o i-node ser removido significará que uma entrada de diretório válida aponta para um i-node listando blocos que pertencem agora ao conjunto de armazenamento livre e que provavelmente serão reutilizados em breve, levando dois ou mais arquivos a compartilhar ao acaso os mesmos blocos. Nenhum desses resultados é bom.

O que o sistema de arquivos journaling faz é primeiro escrever uma entrada no diário listando as três ações a serem concluídas. A entrada no diário é então escrita para o disco (e de maneira previdente, quem sabe lendo de novo do disco para verificar que ela foi, de fato, escrita corretamente). Apenas após a entrada no diário ter sido escrita é que começam as várias operações. Após as operações terem sido concluídas de maneira bem-sucedida, a entrada no diário é apagada. Se o sistema falhar agora, ao se recuperar, o sistema de arquivos poderá conferir o diário para ver se havia alguma operação pendente. Se afirmativo, todas elas podem ser reexecutadas (múltiplas vezes no caso de falhas repetidas) até que o arquivo seja corretamente removido.

Para que o journaling funcione, as operações registradas no diário devem ser **idempotentes**, isto é, elas podem ser repetidas quantas vezes forem necessárias sem prejuízo algum. Operações como "Atualize o mapa de bits para marcar i-node *k* ou bloco *n* como livres" podem ser repetidas sem nenhum problema até o objetivo ser consumado. Do mesmo modo, buscar um diretório e remover qualquer entrada chamada *foobar* também é uma operação idempotente. Por outro lado, adicionar os blocos recentemente liberados do i-node *K* para o final da lista livre não é uma operação idempotente, pois eles talvez já estejam ali. A operação mais cara "Pesquise a lista de blocos livres e inclua o bloco *n* se ele ainda não estiver lá" é idempotente. Sistemas de arquivos journaling têm de arranjar suas estruturas de dados e operações ligadas ao diário de maneira que todos sejam idempotentes. Nessas condições, a recuperação de falhas pode ser rápida e segura.

Para aumentar a confiabilidade, um sistema de arquivos pode introduzir o conceito do banco de dados de uma **transação atômica**. Quando esse conceito é usado, um grupo de ações pode ser formado pelas operações begin transaction e end transaction. O sistema de arquivos sabe então que ele precisa completar todas as operações do grupo ou nenhuma delas, mas não qualquer outra combinação.

O NTFS possui um amplo sistema de journaling e sua estrutura raramente é corrompida por falhas no sistema. Ela está em desenvolvimento desde seu primeiro lançamento com o Windows NT em 1993. O primeiro sistema de arquivos Linux a fazer journaling foi o ReiserFS, mas sua popularidade foi impedida porque ele era incompatível com o então sistema de arquivos ext2 padrão. Em comparação, o ext3, que é um projeto menos ambicioso do que o ReiserFS, também faz journaling enquanto mantém a compatibilidade com o sistema ext2 anterior.

4.3.7 Sistemas de arquivos virtuais

Muitos sistemas de arquivos diferentes estão em uso — muitas vezes no mesmo computador — mesmo para o mesmo sistema operacional. Um sistema Windows

pode ter um sistema de arquivos NTFS principal, mas também uma antiga unidade ou partição FAT-16 ou FAT-32, que contenha dados antigos, porém ainda necessários, e de tempos em tempos um flash drive, um antigo CD-ROM ou um DVD (cada um com seu sistema de arquivos único) podem ser necessários também. O Windows lida com esses sistemas de arquivos díspares identificando cada um com uma letra de unidade diferente, como em *C:*, *D:* etc. Quando um processo abre um arquivo, a letra da unidade está implícita ou explicitamente presente, então o Windows sabe para qual sistema de arquivos passar a solicitação. Não há uma tentativa de integrar sistemas de arquivos heterogêneos em um todo unificado.

Em comparação, todos os sistemas UNIX fazem uma tentativa muito séria de integrar múltiplos sistemas de arquivos em uma única estrutura. Um sistema Linux pode ter o ext2 como o diretório-raiz, com a partição ext3 montada em */usr* e um segundo disco rígido com o sistema de arquivos ReiserFS montado em */home*, assim como um CD-ROM ISO 9660 temporariamente montado em */mnt*. Do ponto de vista do usuário, existe uma hierarquia de sistema de arquivos única. O fato de ela lidar com múltiplos sistemas de arquivos (incompatíveis) não é visível para os usuários ou processos.

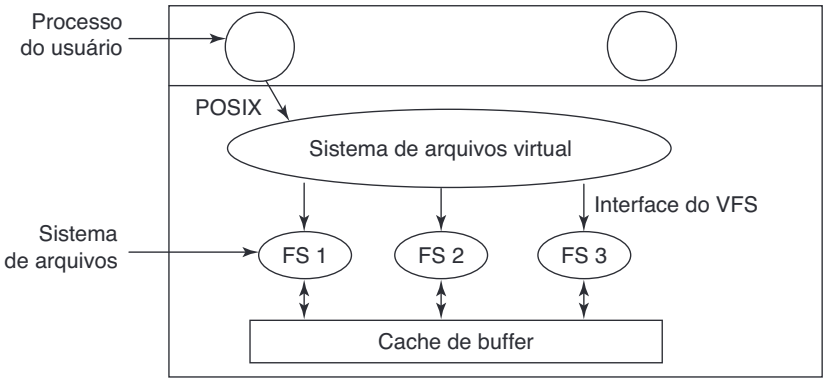
No entanto, a presença de múltiplos sistemas de arquivos é definitivamente visível à implementação, e desde o trabalho pioneiro da Sun Microsystems (KLEIMAN, 1986), a maioria dos sistemas UNIX usou o conceito de um **VFS (Virtual File System** — sistema de arquivos virtuais) para tentar integrar múltiplos sistemas de arquivos em uma estrutura ordeira. A ideia fundamental é abstrair a parte do sistema de arquivos que é comum a todos os sistemas de arquivos e colocar aquele código em uma camada separada que chama os sistemas de arquivos subjacentes para realmente gerenciar os dados. A estrutura como um todo está ilustrada na Figura 4.18. A discussão a seguir não é específica ao Linux, FreeBSD ou qualquer outra versão do UNIX, mas dá uma ideia geral de como os sistemas de arquivos virtuais funcionam nos sistemas UNIX.

Todas as chamadas de sistemas relativas a arquivos são direcionadas ao sistema de arquivos virtual para processamento inicial. Essas chamadas, vindas de outros processos de usuários, são as chamadas POSIX padrão, como open, read, write, lseek e assim por diante. Desse modo, o VFS tem uma interface "superior" para os processos do usuário, e é a já conhecida interface POSIX.

O VFS também tem uma interface "inferior" para os sistemas de arquivos reais, que são rotulados de **interface do VFS** na Figura 4.18. Essa interface consiste em várias dúzias de chamadas de funções que os VFS podem fazer para cada sistema de arquivos para realizar o trabalho. Assim, para criar um novo sistema de arquivos que funcione com o VFS, os projetistas do novo sistema de arquivos devem certificar-se de que ele proporcione as chamadas de funções que o VFS exige. Um exemplo óbvio desse tipo de função é aquela que lê um bloco específico do disco, coloca-o na cache de buffer do sistema de arquivos e retorna um ponteiro para ele. Desse modo, o VFS tem duas interfaces distintas: a superior para os processos do usuário e a inferior para os sistemas de arquivos reais.

Embora a maioria dos sistemas de arquivos sob o VFS represente partições em um disco local, este nem sempre é o caso. Na realidade, a motivação original para a Sun produzir o VFS era dar suporte a sistemas de arquivos remotos usando o protocolo **NFS (Network File System** — sistema de arquivos de rede). O projeto VFS foi feito de tal forma que enquanto o sistema de arquivos real fornecer as funções que o VFS exigir, o VFS não sabe ou se preocupa onde estão armazenados os dados ou como é o sistema de arquivos subjacente.

FIGURA 4.18 Posição do sistema de arquivos virtual.



Internamente, a maioria das implementações de VFS é na essência orientada para objetos, mesmo que todos sejam escritos em C em vez de C++. Há vários tipos de objetos fundamentais que são em geral aceitos. Esses incluem o superbloco (que descreve um sistema de arquivos), o v-node (que descreve um arquivo) e o diretório (que descreve um diretório de sistemas de arquivos). Cada um desses tem operações associadas (métodos) a que os sistemas de arquivos reais têm de dar suporte. Além disso, o VFS tem algumas estruturas internas de dados para seu próprio uso, incluindo a tabela de montagem e um conjunto de descritores de arquivos para monitorar todos os arquivos abertos nos processos do usuário.

Para compreender como o VFS funciona, vamos repassar um exemplo cronologicamente. Quando o sistema é inicializado, o sistema de arquivos raiz é registrado com o VFS. Além disso, quando outros sistemas de arquivos são montados, seja no momento da inicialização ou durante a operação, também devem registrar-se com o VFS. Quando um sistema de arquivos se registra, o que ele basicamente faz é fornecer uma lista de endereços das funções que o VFS exige, seja como um longo vetor de chamada (tabela) ou como vários deles, um por objeto de VFS, como demanda o VFS. Então, assim que um sistema de arquivos tenha se registrado com o VFS, este sabe como, digamos, ler um bloco a partir dele — ele simplesmente chama a quarta (ou qualquer que seja) função no vetor fornecido pelo sistema de arquivos. De modo similar, o VFS então também sabe como realizar todas as funções que o sistema de arquivos real deve fornecer: ele apenas chama a função cujo endereço foi fornecido quando o sistema de arquivos registrou.

Após um sistema de arquivos ter sido montado, ele pode ser usado. Por exemplo, se um sistema de arquivos foi montado em */usr* e um processo fizer a chamada

```
open("/usr/include/unistd.h", O_RDONLY)
```

durante a análise do caminho, o VFS vê que um novo sistema de arquivos foi montado em */usr* e localiza seu superbloco pesquisando a lista de superblocos de sistemas de arquivos montados. Tendo feito isso, ele pode encontrar o diretório-raiz do sistema de arquivos montado e examinar o caminho *include/unistd.h* ali. O VFS então cria um v-node e faz uma chamada para o sistema de arquivos real para retornar todas as informações no i-node do arquivo. Essa informação é copiada para o v-node (em RAM), junto com outras informações, e, o mais importante, cria o ponteiro para a tabela de funções para chamar operações em v-nodes, como read, write, close e assim por diante.

Após o v-node ter sido criado, o VFS registra uma entrada na tabela de descritores de arquivo para o processo que fez a chamada e faz que ele aponte para o novo v-node. (Para os puristas, o descritor de arquivos na realidade aponta para outras estruturas de dados que contêm a posição atual do arquivo e um ponteiro para o v-node, mas esse detalhe não é importante para nossas finalidades aqui.) Por fim, o VFS retorna o descritor de arquivos para o processo que chamou, assim ele pode usá-lo para ler, escrever e fechar o arquivo.

Mais tarde, quando o processo realiza um read usando o descritor de arquivos, o VFS localiza o v-node do processo e das tabelas de descritores de arquivos e segue o ponteiro até a tabela de funções, na qual estão os endereços dentro do sistema de arquivos real, no qual reside o arquivo solicitado. A função responsável pelo read é chamada agora e o código dentro do sistema de arquivos real vai e busca o bloco solicitado. O VFS não faz ideia se os dados estão vindo do disco local, um sistema de arquivos remoto através da rede, um pen-drive ou algo diferente. As estruturas de dados envolvidas são mostradas na Figura 4.19. Começando com o número do processo chamador e o descritor do arquivo, então o v-node, o ponteiro da função de leitura e a função de acesso dentro do sistema de arquivos real são localizados.

Dessa maneira, adicionar novos sistemas de arquivos torna-se algo relativamente direto. Para realizar a operação, os projetistas primeiro tomam uma lista de chamadas de funções esperadas pelo VFS e então escrevem seu sistema de arquivos para prover todas elas. Como alternativa, se o sistema de arquivos já existe, então eles têm de prover funções adaptadoras que façam o que o VFS precisa, normalmente realizando uma ou mais chamadas nativas ao sistema de arquivos real.

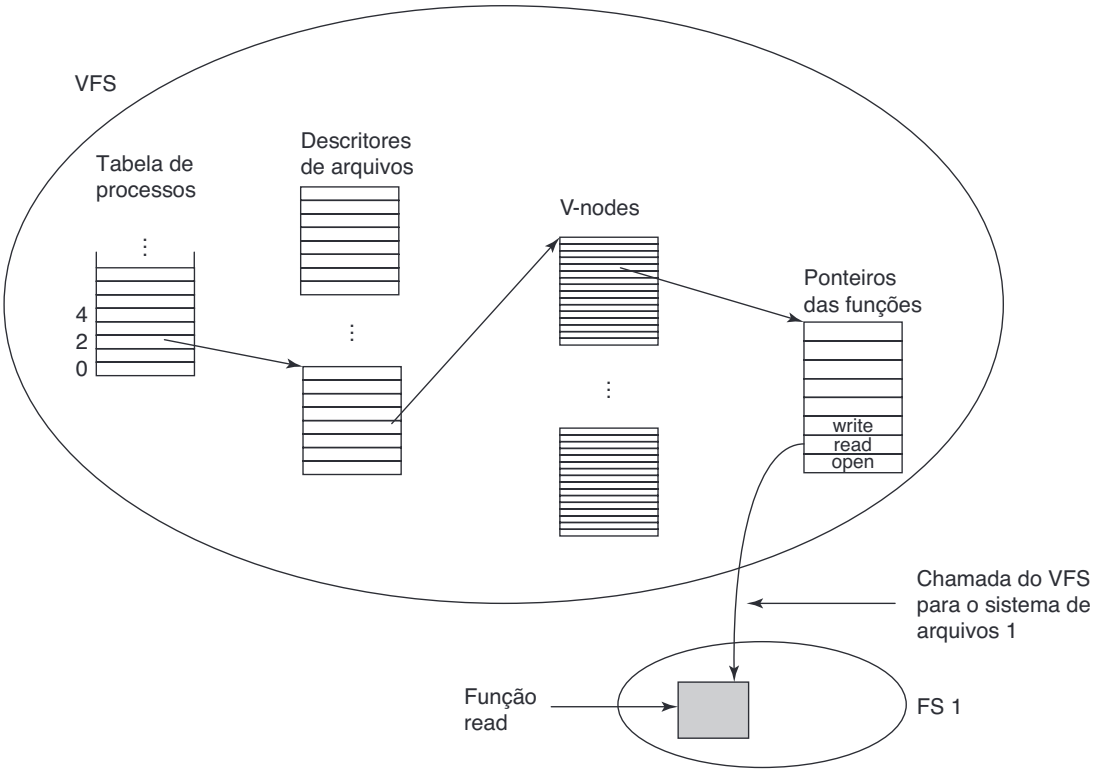
4.4 Gerenciamento e otimização de sistemas de arquivos

Fazer o sistema de arquivos funcionar é uma coisa: fazê-lo funcionar de forma eficiente e robustamente na vida real é algo bastante diferente. Nas seções a seguir examinaremos algumas das questões envolvidas no gerenciamento de discos.

4.4.1 Gerenciamento de espaço em disco

Arquivos costumam ser armazenados em disco, portanto o gerenciamento de espaço de disco é uma preocupação importante para os projetistas de sistemas de

FIGURA 4.19 Uma visão simplificada das estruturas de dados e código usados pelo VFS e pelo sistema de arquivos real para realizar uma operação read.



arquivos. Duas estratégias gerais são possíveis para armazenar um arquivo de *n* bytes: ou são alocados *n* bytes consecutivos de espaço, ou o arquivo é dividido em uma série de blocos (não necessariamente) contíguos. A mesma escolha está presente em sistemas de gerenciamento de memória entre a segmentação pura e a paginação.

Como vimos, armazenar um arquivo como uma sequência contígua de bytes tem o problema óbvio de que se um arquivo crescer, ele talvez tenha de ser movido dentro do disco. O mesmo problema ocorre para segmentos na memória, exceto que mover um segmento na memória é uma operação relativamente rápida em comparação com mover um arquivo de uma posição no disco para outra. Por essa razão, quase todos os sistemas de arquivos os dividem em blocos de tamanho fixo que não precisam ser adjacentes.

Tamanho do bloco

Uma vez que tenha sido feita a opção de armazenar arquivos em blocos de tamanho fixo, a questão que surge é qual tamanho o bloco deve ter. Dado o modo como os discos são organizados, o setor, a trilha e o cilindro são candidatos óbvios para a unidade de alocação (embora sejam todos dependentes do dispositivo, o que é um ponto negativo). Em um sistema de paginação, o tamanho da página também é um argumento importante.

Ter um tamanho de bloco grande significa que todos os arquivos, mesmo um de 1 byte, ocuparão um cilindro inteiro. Também significa que arquivos pequenos desperdiçam uma grande quantidade de espaço de disco. Por outro lado, um tamanho de bloco pequeno significa que a maioria dos arquivos ocupará múltiplos blocos e, desse modo, precisará de múltiplas buscas e atrasos rotacionais para lê-los, reduzindo o desempenho. Então, se a unidade de alocação for grande demais, desperdiçamos espaço; se ela for pequena demais, desperdiçamos tempo.

Fazer uma boa escolha exige ter algumas informações sobre a distribuição do tamanho do arquivo. Tanenbaum et al. (2006) estudaram a distribuição do tamanho do arquivo no Departamento de Ciências de Computação de uma grande universidade de pesquisa (a Universidade Vrije) em 1984 e então novamente em 2005, assim como em um servidor da web comercial hospedando um site de política (<*www.electoral-vote.com*>). Os resultados são mostrados na Figura 4.20, na qual é listada, para cada grupo, a porcentagem de todos os arquivos menores ou iguais ao tamanho (representado por potência de base dois).

FIGURA 4.20 Porcentagem de arquivos menores do que um determinado tamanho (em bytes).

Tamanho	UV 1984	UV 2005	Web
1	1,79	1,38	6,67
2	1,88	1,53	7,67
4	2,01	1,65	8,33
8	2,31	1,80	11,30
16	3,32	2,15	11,46
32	5,13	3,15	12,33
64	8,71	4,98	26,10
128	14,73	8,03	28,49
256	23,09	13,29	32,10
512	34,44	20,62	39,94
1 KB	48,05	30,91	47,82
2 KB	60,87	46,09	59,44
4 KB	75,31	59,13	70,64
8 KB	84,97	69,96	79,69
16 KB	92,53	78,92	86,79
32 KB	97,21	85,87	91,65
64 KB	99,18	90,84	94,80
128 KB	99,84	93,73	96,93
256 KB	99,96	96,12	98,48
512 KB	100,00	97,73	98,99
1 MB	100,00	98,87	99,62
2 MB	100,00	99,44	99,80
4 MB	100,00	99,71	99,87
8 MB	100,00	99,86	99,94
16 MB	100,00	99,94	99,97
32 MB	100,00	99,97	99,99
64 MB	100,00	99,99	99,99
128 MB	100,00	99,99	100,00

Por exemplo, em 2005, 59,13% de todos os arquivos na Universidade de Vrije tinham 4 KB ou menos e 90,84% de todos eles, 64 KB ou menos. O tamanho de arquivo médio era de 2.475 bytes. Algumas pessoas podem achar esse tamanho pequeno surpreendente.

Que conclusões podemos tirar desses dados? Por um lado, com um tamanho de bloco de 1 KB, apenas em torno de 30-50% de todos os arquivos cabem em um único bloco, enquanto com um bloco de 4 KB, a percentagem de arquivos que cabem em um bloco sobe para a faixa de 60-70%. Outros dados no estudo mostram que com um bloco de 4 KB, 93% dos blocos do disco são usados por 10% dos maiores arquivos. Isso significa que o desperdício de espaço ao fim de cada pequeno arquivo é insignificante, pois o disco está cheio por uma pequena quantidade de arquivos grandes (vídeos) e o montante total de espaço tomado pelos arquivos pequenos pouco importa. Mesmo dobrando o espaço requerido por 90% dos menores arquivos, isso mal seria notado.

Por outro lado, utilizar um pequeno bloco significa que cada arquivo consistirá em muitos blocos. Ler cada bloco exige uma busca e um atraso rotacional (exceto em um disco em estado sólido), então a leitura de um arquivo consistindo em muitos blocos pequenos será lenta.

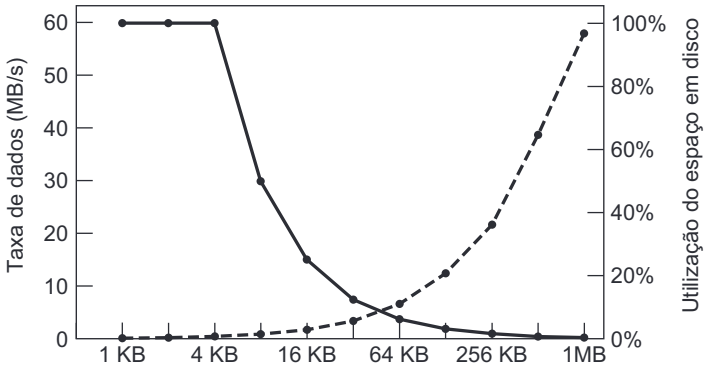
Como exemplo, considere um disco com 1 MB por trilha, um tempo de rotação de 8,33 ms e um tempo de busca de 5 ms. O tempo em milissegundos para ler um bloco de k bytes é então a soma dos tempos de busca, atraso rotacional e transferência:

$$5 + 4{,}165 + (k/1000000) \times 8{,}33$$

A curva tracejada da Figura 4.21 mostra a taxa de dados para um disco desses como uma função do tamanho do bloco. Para calcular a eficiência de espaço, precisamos fazer uma suposição a respeito do tamanho médio do arquivo. Para simplificar, vamos presumir que todos os arquivos tenham 4 KB. Embora esse número seja ligeiramente maior do que os dados medidos na Universidade de Vrije, os estudantes provavelmente têm mais arquivos pequenos do que os existentes em um centro de dados corporativo, então, como um todo, talvez seja um palpite melhor. A curva sólida da Figura 4.21 mostra a eficiência de espaço como uma função do tamanho do bloco.

As duas curvas podem ser compreendidas como a seguir. O tempo de acesso para um bloco é completamente dominado pelo tempo de busca e atraso rotacional, então levando-se em consideração que serão necessários 9 ms para acessar um bloco, quanto mais dados forem buscados, melhor. Assim, a taxa de dados cresce quase linearmente com o tamanho do bloco (até as transferências demorarem tanto que o tempo de transferência começa a importar).

FIGURA 4.21 A curva tracejada (escala da esquerda) mostra a taxa de dados de um disco. A curva contínua (escala da direita) mostra a eficiência do espaço em disco. Todos os arquivos têm 4 KB.



Agora considere a eficiência de espaço. Com arquivos de 4 KB e blocos de 1 KB, 2 KB ou 4 KB, os arquivos usam 4, 2 e 1 bloco, respectivamente, sem desperdício. Com um bloco de 8 KB e arquivos de 4 KB, a eficiência de espaço cai para 50% e com um bloco de 16 KB ela chega a 25%. Na realidade, poucos arquivos são um múltiplo exato do tamanho do bloco do disco, então algum espaço sempre é desperdiçado no último bloco de um arquivo.

O que as curvas mostram, no entanto, é que o desempenho e a utilização de espaço estão inerentemente em conflito. Pequenos blocos são ruins para o desempenho, mas bons para a utilização do espaço do disco. Para esses dados, não há equilíbrio que seja razoável. O tamanho mais próximo de onde as duas curvas se cruzam é 64 KB, mas a taxa de dados é de apenas 6,6 MB/s e a eficiência de espaço é de cerca de 7%, nenhum dos dois valores é muito bom. Historicamente, sistemas de arquivos escolheram tamanhos na faixa de 1 KB a 4 KB, mas com discos agora excedendo 1 TB, pode ser melhor aumentar o tamanho do bloco para 64 KB e aceitar o espaço de disco desperdiçado. Hoje é muito pouco provável que falte espaço de disco.

Em um experimento para ver se o uso de arquivos do Windows NT era apreciavelmente diferente do uso de arquivos do UNIX, Vogels tomou medidas nos arquivos na Universidade de Cornell (VOGELS, 1999). Ele observou que o uso de arquivos no NT é mais complicado que no UNIX. Ele escreveu:

> *Quando digitamos alguns caracteres no editor de texto do Notepad, o salvamento dessa digitação em um arquivo desencadeará 26 chamadas de sistema, incluindo 3 tentativas de abertura que falharam, 1 arquivo sobrescrito e 4 sequências adicionais de abertura e fechamento.*

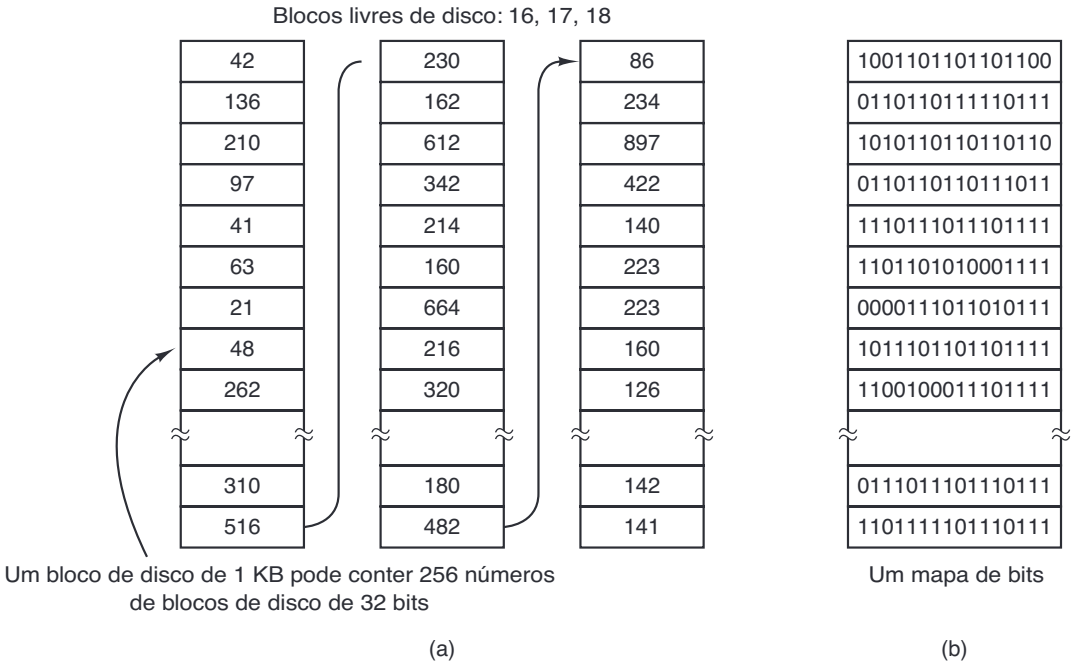
Não obstante isso, Vogels observou um tamanho médio (ponderado pelo uso) de arquivos apenas lidos como 1 KB, arquivos apenas escritos como 2,3 KB e arquivos lidos e escritos como 4,2 KB. Considerando as diferentes técnicas de mensuração de conjuntos de dados, e o ano, esses resultados são certamente compatíveis com os da Universidade de Vrije.

Monitoramento dos blocos livres

Uma vez que um tamanho de bloco tenha sido escolhido, a próxima questão é como monitorar os blocos livres. Dois métodos são amplamente usados, como mostrado na Figura 4.22. O primeiro consiste em usar uma lista encadeada de blocos de disco, com cada bloco contendo tantos números de blocos livres de disco quantos couberem nele. Com um bloco de 1 KB e um número de bloco de disco de 32 bits, cada bloco na lista livre contém os números de 255 blocos livres. (Uma entrada é reservada para o ponteiro para o bloco seguinte.) Considere um disco de 1 TB, que tem em torno de 1 bilhão de blocos de disco. Armazenar todos esses endereços em blocos de 255 exige cerca de 4 milhões de blocos. Em geral, blocos livres são usados para conter a lista livre, de maneira que o armazenamento seja essencialmente gratuito.

A outra técnica de gerenciamento de espaço livre é o mapa de bits. Um disco com n blocos exige um mapa de bits com n bits. Blocos livres são representados por 1s no mapa, blocos alocados por 0s (ou vice-versa). Para nosso disco de 1 TB de exemplo, precisamos de 1 bilhão de bits para o mapa, o que exige em torno de 130.000 blocos de 1 KB para armazenar. Não surpreende que o mapa de bits exija menos espaço, tendo em vista que ele usa 1 bit por bloco, *versus* 32 bits no modelo de lista encadeada. Apenas se o disco estiver praticamente cheio (isto é, tiver poucos blocos livres) o esquema da lista encadeada exigirá menos blocos do que o mapa de bits.

FIGURA 4.22 (a) Armazenamento da lista de blocos livres em uma lista encadeada. (b) Um mapa de bits.



Se os blocos livres tenderem a vir em longos conjuntos de blocos consecutivos, o sistema da lista de blocos livres pode ser modificado para controlar conjuntos de blocos em vez de blocos individuais. Um contador de 8, 16 ou 32 bits poderia ser associado com cada bloco dando o número de blocos livres consecutivos. No melhor caso, um disco basicamente vazio seria representado por dois números: o endereço do primeiro bloco livre seguido pelo contador de blocos livres. Por outro lado, se o disco se tornar severamente fragmentado, o controle de conjuntos de blocos será menos eficiente do que o controle de blocos individuais, pois não apenas o endereço deverá ser armazenado, mas também o contador.

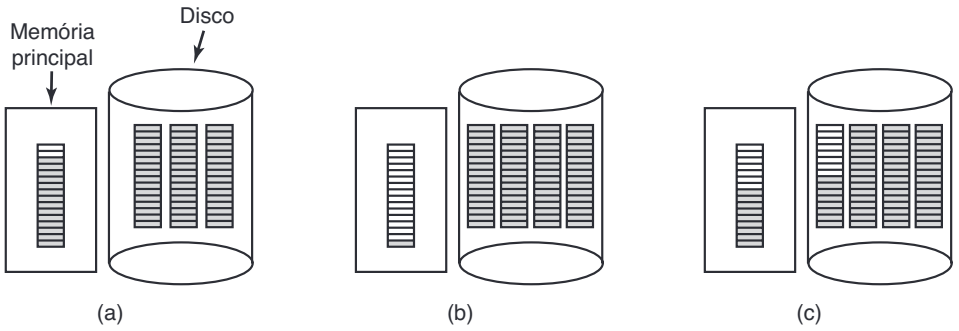
Essa questão ilustra um problema que os projetistas de sistemas operacionais muitas vezes enfrentam. Existem múltiplas estruturas de dados e algoritmos que podem ser usados para solucionar um problema, mas a escolha do melhor exige dados que os projetistas não têm e não terão até que o sistema seja distribuído e amplamente utilizado. E, mesmo assim, os dados podem não estar disponíveis. Por exemplo, nossas próprias medidas de tamanhos de arquivos na Universidade de Vrije em 1984 e 1995, os dados do site e os dados de Cornell são apenas quatro amostras. Embora muito melhor do que nada, temos pouca certeza se eles são também representativos de computadores pessoais, computadores corporativos, computadores do governo e outros. Com algum esforço poderíamos ser capazes de conseguir algumas amostras de outros tipos de computadores, mas mesmo assim seria uma bobagem extrapolar para todos os computadores dos tipos mensurados.

Voltando para o método da lista de blocos livres por um momento, apenas um bloco de ponteiros precisa ser mantido na memória principal. Quando um arquivo é criado, os blocos necessários são tomados do bloco de ponteiros. Quando ele se esgota, um novo bloco de ponteiros é lido do disco. De modo similar, quando um arquivo é removido, seus blocos são liberados e adicionados ao bloco de ponteiros na memória principal. Quando esse bloco completa, ele é escrito no disco.

Em determinadas circunstâncias, esse método leva a operações desnecessárias de E/S em disco. Considere a situação da Figura 4.23(a), na qual o bloco de ponteiros na memória tem espaço para somente duas entradas. Se um arquivo de três blocos for liberado, o bloco de ponteiros transbordará e ele deverá ser escrito para o disco, levando à situação da Figura 4.23(b). Se um arquivo de três blocos for escrito agora, o bloco de ponteiros cheio deverá ser lido novamente, trazendo-nos de volta para a Figura 4.23(a). Se o arquivo de três blocos recém-escrito constituir um arquivo temporário, quando ele for liberado, será necessária outra operação de escrita para escrever novamente o bloco de ponteiros cheio no disco. Resumindo, quando o bloco de ponteiros estiver quase vazio, uma série de arquivos temporários de vida curta pode causar muitas operações de E/S em disco.

Uma abordagem alternativa que evita a maior parte dessas operações de E/S em disco é dividir o bloco

FIGURA 4.23 (a) Um bloco na memória quase cheio de ponteiros para blocos de disco livres e três blocos de ponteiros em disco. (b) Resultado da liberação de um arquivo de três blocos. (c) Uma estratégia alternativa para lidar com os três blocos livres. As entradas sombreadas representam ponteiros para blocos de discos livres.



cheio de ponteiros. Desse modo, em vez de ir da Figura 4.23(a) para a Figura 4.23(b), vamos da Figura 4.23(a) para a Figura 4.23(c) quando três blocos são liberados. Agora o sistema pode lidar com uma série de arquivos temporários sem realizar qualquer operação de E/S em disco. Se o bloco na memória encher, ele será escrito para o disco e o bloco meio cheio será lido do disco. A ideia aqui é manter a maior parte dos blocos de ponteiros cheios em disco (para minimizar o uso deste), mas manter o bloco na memória cheio pela metade, de maneira que ele possa lidar tanto com a criação quanto com a remoção de arquivos, sem uma operação de E/S em disco para a lista de livres.

Com um mapa de bits, também é possível manter apenas um bloco na memória, usando o disco para outro bloco apenas quando ele ficar completamente cheio ou vazio. Um benefício adicional dessa abordagem é que ao realizar toda a alocação de um único bloco do mapa de bits, os blocos de disco estarão mais próximos, minimizando assim os movimentos do braço do disco. Já que o mapa de bits é uma estrutura de dados de tamanho fixo, se o núcleo estiver (parcialmente) paginado, o mapa de bits pode ser colocado na memória virtual e ter suas páginas paginadas conforme a necessidade.

Cotas de disco

Para evitar que as pessoas exagerem no uso do espaço de disco, sistemas operacionais de múltiplos usuários muitas vezes fornecem um mecanismo para impor cotas de disco. A ideia é que o administrador do sistema designe a cada usuário uma cota máxima de arquivos e blocos, e o sistema operacional se certifique de que os usuários não excedam essa cota. Um mecanismo típico é descrito a seguir.

Quando um usuário abre um arquivo, os atributos e endereços de disco são localizados e colocados em uma tabela de arquivos aberta na memória principal. Entre os atributos há uma entrada dizendo quem é o proprietário. Quaisquer aumentos no tamanho do arquivo serão cobrados da cota do proprietário.

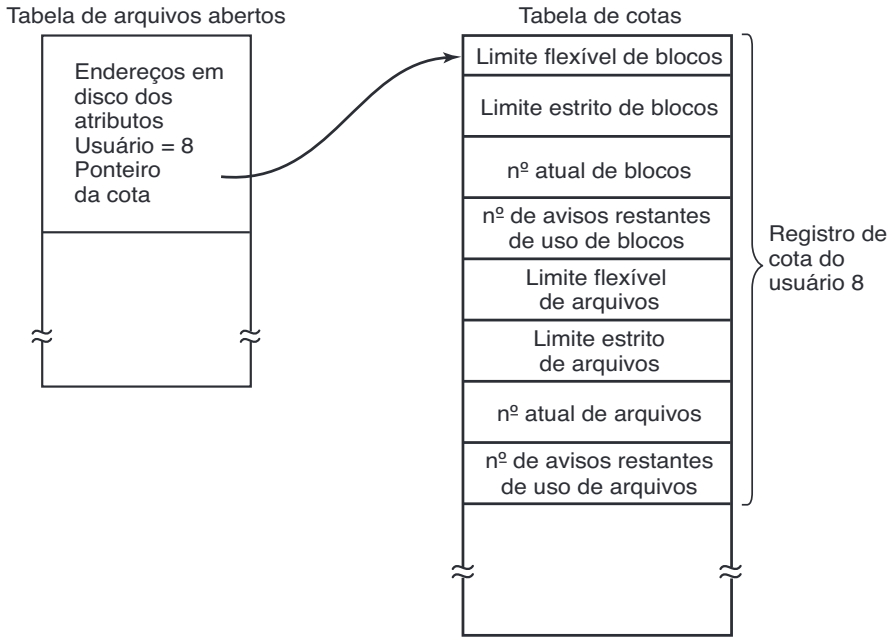
Uma segunda tabela contém os registros de cotas de todos os usuários com um arquivo aberto, mesmo que esse arquivo tenha sido aberto por outra pessoa. Essa tabela está mostrada na Figura 4.24. Ela foi extraída de um arquivo de cotas no disco para os usuários cujos arquivos estão atualmente abertos. Quando todos os arquivos são fechados, o registro é escrito de volta para o arquivo de cotas.

Quando uma nova entrada é feita na tabela de arquivos abertos, um ponteiro para o registro de cota do proprietário é atribuído a ela, a fim de facilitar encontrar os vários limites. Toda vez que um bloco é adicionado a um arquivo, o número total de blocos cobrados do proprietário é incrementado, e os limites flexíveis e estritos são verificados. O limite flexível pode ser excedido, mas o limite estrito não. Uma tentativa de adicionar blocos a um arquivo quando o limite de blocos estrito tiver sido alcançado resultará em um erro. Verificações análogas também existem para o número de arquivos a fim de evitar que algum usuário sobrecarregue todos os i-nodes.

Quando um usuário tenta entrar no sistema, este examina o arquivo de cotas para ver se ele excedeu o limite flexível para o número de arquivos ou o número de blocos de disco. Se qualquer um dos limites foi violado, um aviso é exibido, e o contador de avisos restantes é reduzido para um. Se o contador chegar a zero, o usuário ignorou o aviso vezes demais, e não tem permissão para entrar. Conseguir a autorização para entrar novamente exigirá alguma conversa com o administrador do sistema.

Esse método tem a propriedade de que os usuários podem ir além de seus limites flexíveis durante uma sessão de uso, desde que removam o excesso antes de se desconectarem. Os limites estritos jamais podem ser excedidos.

FIGURA 4.24 As cotas são relacionadas aos usuários e monitoradas em uma tabela de cotas.



4.4.2 Backups (cópias de segurança) do sistema de arquivos

A destruição de um sistema de arquivos é quase sempre um desastre muito maior do que a destruição de um computador. Se um computador for destruído pelo fogo, por uma descarga elétrica ou uma xícara de café derrubada no teclado, isso é irritante e custará dinheiro, mas geralmente uma máquina nova pode ser comprada com um mínimo de incômodo. Computadores pessoais baratos podem ser substituídos na mesma hora, bastando uma ida à loja (menos nas universidades, onde emitir uma ordem de compra exige três comitês, cinco assinaturas e 90 dias).

Se o sistema de arquivos de um computador estiver irrevogavelmente perdido, seja pelo hardware ou pelo software, restaurar todas as informações será difícil, exigirá tempo e, em muitos casos, será impossível. Para as pessoas cujos programas, documentos, registros tributários, arquivos de clientes, bancos de dados, planos de marketing, ou outros dados estiverem perdidos para sempre as consequências podem ser catastróficas. Apesar de o sistema de arquivos não conseguir oferecer qualquer proteção contra a destruição física dos equipamentos e da mídia, ele pode ajudar a proteger as informações. A solução é bastante clara: fazer cópias de segurança (backups). Mas isso pode não ser tão simples quanto parece. Vamos examinar a questão.

A maioria das pessoas não acredita que fazer backups dos seus arquivos valha o tempo e o esforço — até que um belo dia seu disco morre abruptamente, momento que a maioria delas jamais esquecerá. As empresas, no entanto, compreendem (normalmente) bem o valor dos seus dados e costumam realizar um backup ao menos uma vez ao dia, muitas vezes em fita. As fitas modernas armazenam centenas de gigabytes e custam centavos por gigabyte. Não obstante isso, realizar backups não é algo tão trivial quanto parece, então examinaremos algumas das questões relacionadas a seguir.

Backups para fita são geralmente feitos para lidar com um de dois problemas potenciais:

1. Recuperação em caso de um desastre.
2. Recuperação de uma bobagem feita.

O primeiro problema diz respeito a fazer o computador funcionar novamente após uma quebra de disco, fogo, enchente ou outra catástrofe natural. Na prática, essas coisas não acontecem com muita frequência, razão pela qual muitas pessoas não se preocupam em fazer backups. Essas pessoas também tendem a não ter seguro contra incêndio em suas casas pela mesma razão.

A segunda razão é que os usuários muitas vezes removem acidentalmente arquivos de que precisam mais tarde outra vez. Esse problema ocorre com tanta frequência que, quando um arquivo é "removido" no Windows, ele não é apagado de maneira alguma, mas simplesmente movido para um diretório especial, a **cesta de reciclagem**, de maneira que ele possa buscado e restaurado facilmente mais tarde. Backups levam esse princípio mais longe ainda e permitem que arquivos que

foram removidos há dias, mesmo semanas, sejam restaurados de velhas fitas de backup.

Fazer backup leva um longo tempo e ocupa um espaço significativo, portanto é importante fazê-lo de maneira eficiente e conveniente. Essas considerações levantam as questões a seguir. Primeiro, será que todo o sistema de arquivos deve ser copiado ou apenas parte dele? Em muitas instalações, os programas executáveis (binários) são mantidos em uma parte limitada da árvore do sistema de arquivos. Não é necessário realizar backup de todos esses arquivos se todos eles podem ser reinstalados a partir do site do fabricante ou de um DVD de instalação. Também, a maioria dos sistemas tem um diretório para arquivos temporários. Em geral não há uma razão para fazer um backup dele também. No UNIX, todos os arquivos especiais (dispositivos de E/S) são mantidos em um diretório */dev*. Fazer um backup desse diretório não só é desnecessário, como é realmente perigoso, pois o programa de backup poderia ficar pendurado para sempre se ele tentasse ler cada um desses arquivos até terminar. Resumindo, normalmente é desejável fazer o backup apenas de diretórios específicos e tudo neles em vez de todo o sistema de arquivos.

Segundo, é um desperdício fazer o backup de arquivos que não mudaram desde o último backup, o que leva à ideia de **cópias incrementais**. A forma mais simples de cópia incremental é realizar uma cópia (backup) completa periodicamente, digamos por semana ou por mês, e realizar uma cópia diária somente daqueles arquivos que foram modificados desde a última cópia completa. Melhor ainda é copiar apenas aqueles arquivos que foram modificados desde a última vez em que foram copiados. Embora esse esquema minimize o tempo de cópia, ele torna a recuperação mais complicada, pois primeiro a cópia mais recente deve ser restaurada e depois todas as cópias incrementais têm de ser restauradas na ordem inversa. Para facilitar a recuperação, muitas vezes são usados esquemas de cópias incrementais mais sofisticados.

Terceiro, visto que quantidades imensas de dados geralmente são copiadas, pode ser desejável comprimir os dados antes de escrevê-los na fita. No entanto, com muitos algoritmos de compressão, um único defeito na fita de backup pode estragar o algoritmo e tornar um arquivo inteiro ou mesmo uma fita inteira ilegível. Desse modo, a decisão de comprimir os dados de backup deve ser cuidadosamente considerada.

Quarto, é difícil realizar um backup em um sistema de arquivos ativo. Se os arquivos e diretórios estão sendo adicionados, removidos e modificados durante o processo de cópia, a cópia resultante pode ficar inconsistente. No entanto, como realizar uma cópia pode levar horas, talvez seja necessário deixar o sistema off-line por grande parte da noite para realizar o backup, algo que nem sempre é aceitável. Por essa razão, algoritmos foram projetados para gerar fotografias (snapshots) rápidas do estado do sistema de arquivos copiando estruturas críticas de dados e então exigindo que nas mudanças futuras em arquivos e diretórios sejam realizadas cópias dos blocos em vez de atualizá-los diretamente (HUTCHINSON et al., 1999). Dessa maneira, o sistema de arquivos é efetivamente congelado no momento do snapshot; portanto, pode ser copiado depois quando o usuário quiser.

Quinto e último, fazer backups introduz muitos problemas não técnicos na organização. O melhor sistema de segurança on-line no mundo pode ser inútil se o administrador do sistema mantiver todos os discos ou fitas de backup em seu gabinete e deixá-lo aberto e desguarnecido sempre que for buscar um café no fim do corredor. Tudo o que um espião precisa fazer é aparecer por um segundo, colocar um disco ou fita minúsculos em seu bolso e cair fora lepidamente. Adeus, segurança. Também, realizar um backup diário tem pouco uso se o fogo que queimar os computadores também queimar todos os discos de backup. Por essa razão, discos de backup devem ser mantidos longe dos computadores, mas isso introduz mais riscos (pois agora dois locais precisam contar com segurança). Para uma discussão aprofundada dessas e de outras questões administrativas práticas, ver Nemeth et al. (2013). A seguir discutiremos apenas as questões técnicas envolvidas em realizar backups de sistemas de arquivos.

Duas estratégias podem ser usadas para copiar um disco para um disco de backup: uma cópia física ou uma cópia lógica. Uma **cópia física** começa no bloco 0 do disco, escreve em ordem todos os blocos de disco no disco de saída, e para quando ele tiver copiado o último. Esse programa é tão simples que provavelmente pode ser feito 100% livre de erros, algo que em geral não pode ser dito a respeito de qualquer outro programa útil.

Mesmo assim, vale a pena fazer vários comentários a respeito da cópia física. Por um lado, não faz sentido fazer backup de blocos de disco que não estejam sendo usados. Se o programa de cópia puder obter acesso à estrutura de dados dos blocos livres, ele pode evitar copiar blocos que não estejam sendo usados. No entanto, pular blocos que não estejam sendo usados exige escrever o número de cada bloco na frente dele (ou o equivalente), já que não é mais verdade que o bloco *k* no backup era o bloco *k* no disco.

Uma segunda preocupação é copiar blocos defeituosos. É quase impossível manufaturar discos grandes sem quaisquer defeitos. Alguns blocos defeituosos estão sempre presentes. Às vezes, quando é feita uma formatação de baixo nível, os blocos defeituosos são detectados, marcados como tal e substituídos por blocos de reserva guardados ao final de cada trilha para precisamente esse tipo de emergência. Em muitos casos, o controlador de disco gerencia a substituição de blocos defeituosos de forma transparente sem que o sistema operacional nem fique sabendo a respeito.

No entanto, às vezes os blocos passam a apresentar defeitos após a formatação, caso em que o sistema operacional eventualmente vai detectá-los. Em geral, ele soluciona o problema criando um "arquivo" consistindo em todos os blocos defeituosos — somente para certificar-se de que eles jamais apareçam como livres e sejam ocupados. Desnecessário dizer que esse arquivo é completamente ilegível.

Se todos os blocos defeituosos forem remapeados pelo controlador do disco e escondidos do sistema operacional como descrito há pouco, a cópia física funcionará bem. Por outro lado, se eles forem visíveis para o sistema operacional e mantidos em um ou mais arquivos de blocos defeituosos ou mapas de bits, é absolutamente essencial que o programa de cópia física tenha acesso a essa informação e evite copiá-los para evitar erros de leitura de disco intermináveis enquanto tenta fazer o backup do arquivo de bloco defeituoso.

Sistemas Windows têm arquivos de paginação e hibernação que não são necessários no caso de uma restauração e não devem ser copiados em primeiro lugar. Sistemas específicos talvez também tenham outros arquivos internos que não devem ser copiados, então o programa de backup precisa ter consciência deles.

As principais vantagens da cópia física são a simplicidade e a grande velocidade (basicamente, ela pode ser executada na velocidade do disco). As principais desvantagens são a incapacidade de pular diretórios selecionados, realizar cópias incrementais e restaurar arquivos individuais mediante pedido. Por essas razões, a maioria das instalações faz cópias lógicas.

Uma **cópia lógica** começa em um ou mais diretórios especificados e recursivamente copia todos os arquivos e diretórios encontrados ali que foram modificados desde uma determinada data de base (por exemplo, o último backup para uma cópia incremental ou instalação de sistema para uma cópia completa). Assim, em uma cópia lógica, o disco da cópia recebe uma série de diretórios e arquivos cuidadosamente identificados, o que torna fácil restaurar um arquivo ou diretório específico mediante pedido.

Tendo em vista que a cópia lógica é a forma mais usual, vamos examinar um algoritmo comum em detalhe usando o exemplo da Figura 4.25 para nos orientar. A maioria dos sistemas UNIX usa esse algoritmo. Na figura vemos uma árvore com diretórios (quadrados) e arquivos (círculos). Os itens sombreados foram modificados desde a data de base e desse modo precisam ser copiados. Os arquivos não sombreados não precisam ser copiados.

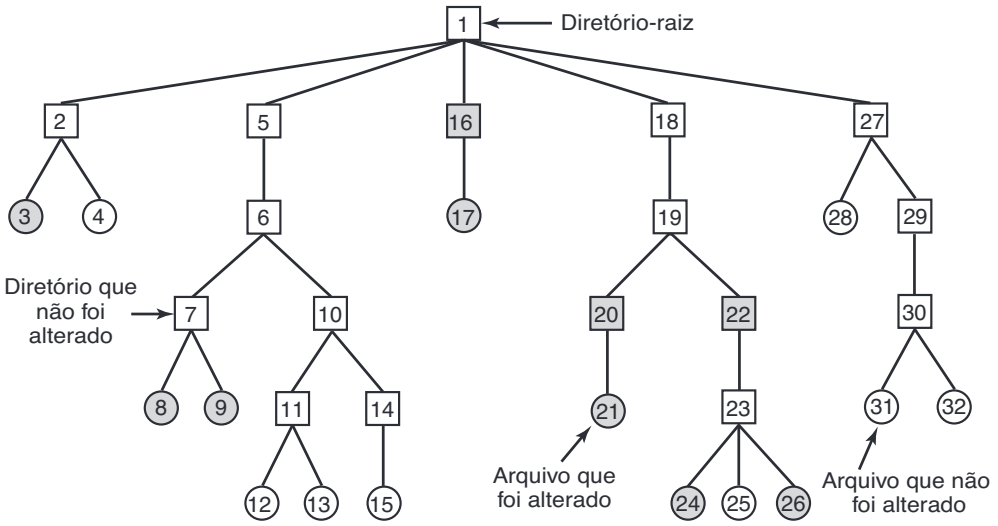
Esse algoritmo também copia todos os diretórios (mesmo os inalterados) que ficam no caminho de um arquivo ou diretório modificado por duas razões. A primeira é tornar possível restaurar os arquivos e diretórios copiados para um sistema de arquivos novos em um computador diferente. Dessa maneira, os programas de cópia e restauração podem ser usados para transportar sistemas de arquivos inteiros entre computadores.

A segunda razão para copiar diretórios inalterados que estejam acima de arquivos modificados é tornar possível restaurar de maneira incremental um único arquivo (possivelmente para recuperar alguma bobagem cometida). Suponha que uma cópia completa do sistema de arquivos seja feita no domingo à noite e uma cópia incremental seja feita segunda-feira à noite. Na terça-feira o diretório */usr/jhs/proj/nr3* é removido, junto com todos os diretórios e arquivos sob ele. Na manhã ensolarada de quarta-feira suponha que o usuário queira restaurar o arquivo */usr/jhs/proj/nr3/plans/summary.* No entanto, não é possível apenas restaurar o arquivo *summary* porque não há lugar para colocá-lo. Os diretórios *nr3* e *plans* devem ser restaurados primeiro. Para obter seus proprietários, modos, horários etc. corretos, esses diretórios precisam estar presentes no disco de cópia mesmo que eles mesmos não tenham sido modificados antes da cópia completa anterior.

O algoritmo de cópia mantém um mapa de bits indexado pelo número do i-node com vários bits por i-node. Bits serão definidos como 1 ou 0 nesse mapa conforme o algoritmo é executado. O algoritmo opera em quatro fases. A fase 1 começa do diretório inicial (a raiz neste exemplo) e examina todas as entradas nele. Para cada arquivo modificado, seu i-node é marcado no mapa de bits. Cada diretório também é marcado (modificado ou não) e então inspecionado recursivamente.

Ao fim da fase 1, todos os arquivos modificados e todos os diretórios foram marcados no mapa de bits, como mostrado (pelo sombreamento) na Figura 4.26(a). A fase 2 conceitualmente percorre a árvore de novo de maneira recursiva, desmarcando quaisquer diretórios

FIGURA 4.25 Um sistema de arquivos a ser copiado. Os quadrados são diretórios e os círculos, arquivos. Os itens sombreados foram modificados desde a última cópia. Cada diretório e arquivo estão identificados por seu número de i-node.



que não tenham arquivos ou diretórios modificados neles ou sob eles. Essa fase deixa o mapa de bits como mostrado na Figura 4.26(b). Observe que os diretórios 10, 11, 14, 27, 29 e 30 estão agora desmarcados, pois não contêm nada modificado sob eles. Eles não serão copiados. Por outro lado, os diretórios 5 e 6 serão copiados mesmo que não tenham sido modificados, pois serão necessários para restaurar as mudanças de hoje para uma máquina nova. Para fins de eficiência, as fases 1 e 2 podem ser combinadas para percorrer a árvore uma única vez.

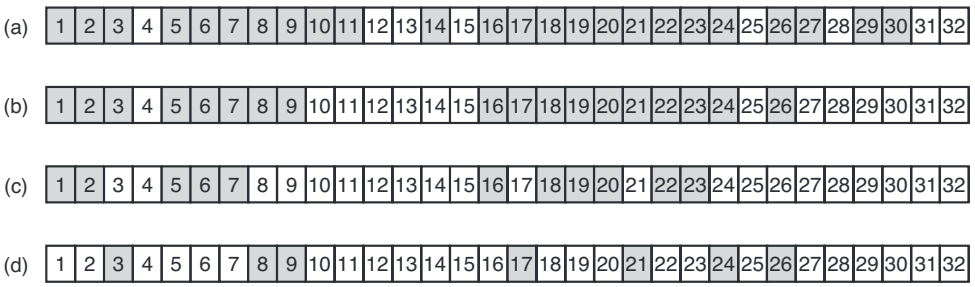
Nesse ponto, sabe-se quais diretórios e arquivos precisam ser copiados. Esses são os arquivos que estão marcados na Figura 4.26(b). A fase 3 consiste em escanear os i-nodes em ordem numérica e copiar todos os diretórios que estão marcados para serem copiados. Esses são mostrados na Figura 4.26(c). Cada diretório é prefixado pelos atributos do diretório (proprietário, horários etc.), de maneira que eles possam ser restaurados. Por fim, na fase 4, os arquivos marcados na Figura 4.26(d) também são copiados, mais uma vez prefixados por seus atributos. Isso completa a cópia.

Restaurar um sistema de arquivos a partir do disco de cópia é algo simples. Para começar, um sistema de arquivos vazio é criado no disco. Então a cópia completa mais recente é restaurada. Já que os diretórios aparecem primeiro no disco de cópia, eles são todos restaurados antes, fornecendo um esqueleto ao sistema de arquivos. Então os arquivos em si são restaurados. Esse processo é repetido com a primeira cópia incremental feita após a cópia completa, depois a seguinte e assim por diante.

Embora a cópia lógica seja simples, há algumas questões complicadas. Por exemplo, já que a lista de blocos livres não é um arquivo, ele não é copiado e assim deve ser reconstruído desde o ponto de partida depois de todas as cópias terem sido restauradas. Realizá-lo sempre é possível já que o conjunto de blocos livres é apenas o complemento do conjunto dos blocos contidos em todos os arquivos combinados.

Outra questão são as ligações. Se um arquivo está ligado a dois ou mais diretórios, é importante que seja restaurado apenas uma vez e que todos os diretórios que supostamente estejam apontando para ele assim o façam.

FIGURA 4.26 Mapas de bits usados pelo algoritmo da cópia lógica.



Ainda outra questão é o fato de que os arquivos UNIX possam conter lacunas. É permitido abrir um arquivo, escrever alguns bytes, então deslocar para uma posição mais distante e escrever mais alguns bytes. Os blocos entre eles não fazem parte do arquivo e não devem ser copiados e restaurados. Arquivos contendo a imagem de processos terminados de modo anormal (core files) apresentam muitas vezes uma lacuna de centenas de megabytes entre os segmentos de dados e a pilha. Se não for tratado adequadamente, cada core file restaurado preencherá essa área com zeros e desse modo terá o mesmo tamanho do espaço de endereço virtual (por exemplo, 2^{32} bytes, ou pior ainda, 2^{64} bytes).

Por fim, arquivos especiais, chamados pipes, e outros similares (qualquer coisa que não seja um arquivo real) jamais devem ser copiados, não importa em qual diretório eles possam ocorrer (eles não precisam estar confinados em */dev*). Para mais informações sobre backups de sistemas de arquivos, ver Chervenak et al. (1998) e Zwicky (1991).

4.4.3 Consistência do sistema de arquivos

Outra área na qual a confiabilidade é um problema é a consistência do sistema de arquivos. Muitos sistemas de arquivos leem blocos, modificam-nos e só depois os escrevem. Se o sistema cair antes de todos os blocos modificados terem sido escritos, o sistema de arquivos pode ser deixado em um estado inconsistente. O problema é especialmente crítico se alguns dos blocos que não foram escritos forem blocos de i-nodes, de diretórios ou blocos contendo a lista de blocos livres.

Para lidar com sistemas de arquivos inconsistentes, a maioria dos programas tem um programa utilitário que confere a consistência do sistema de arquivos. Por exemplo, UNIX tem *fsck*; Windows tem *sfc* (e outros). Esse utilitário pode ser executado sempre que o sistema é iniciado, especialmente após uma queda. A descrição a seguir explica como o *fsck* funciona. *Sfc* é de certa maneira diferente, pois ele funciona em um sistema de arquivos distinto, mas o princípio geral de usar a redundância inerente do sistema de arquivos para repará-lo ainda é válido. Todos os verificadores conferem cada sistema de arquivos (partição do disco) independentemente dos outros.

Dois tipos de verificações de consistência podem ser feitos: blocos e arquivos. Para conferir a consistência do bloco, o programa constrói duas tabelas, cada uma contendo um contador para cada bloco, inicialmente contendo 0. Os contadores na primeira tabela monitoram quantas vezes cada bloco está presente em um arquivo; os contadores na segunda tabela registram quantas vezes cada bloco está presente na lista de livres (ou o mapa de bits de blocos livres).

O programa então lê todos os i-nodes usando um dispositivo cru, que ignora a estrutura de arquivos e apenas retorna todos os blocos de disco começando em 0. A partir de um i-node, é possível construir uma lista de todos os números de blocos usados no arquivo correspondente. À medida que cada número de bloco é lido, seu contador na primeira tabela é incrementado. O programa então examina a lista de livres ou mapa de bits para encontrar todos os blocos que não estão sendo usados. Cada ocorrência de um bloco na lista de livres resulta em seu contador na segunda tabela sendo incrementado.

Se o sistema de arquivos for consistente, cada bloco terá um 1 na primeira ou na segunda tabela, como ilustrado na Figura 4.27(a). No entanto, como consequência de uma queda no sistema, as tabelas podem ser parecidas com a Figura 4.27(b), na qual o bloco 2 não ocorre em nenhuma tabela. Ele será reportado como um **bloco desaparecido**. Embora blocos desaparecidos não causem nenhum prejuízo real, eles desperdiçam espaço e reduzem assim a capacidade do disco. A solução para os blocos desaparecidos é simples: o verificador do sistema de arquivos apenas os adiciona à lista de blocos livres.

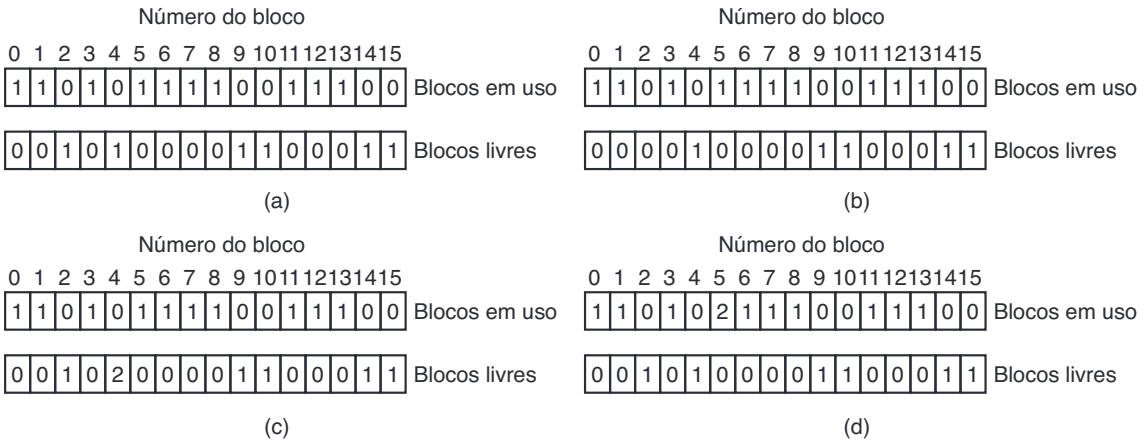
Outra situação que pode ocorrer é aquela da Figura 4.27(c). Aqui vemos um bloco, número 4, que ocorre duas vezes na lista de livres. (Duplicatas podem ocorrer apenas se a lista de livres for realmente uma lista; com um mapa de bits isso é impossível.) A solução aqui também é simples: reconstruir a lista de livres.

A pior coisa que pode ocorrer é o mesmo bloco de dados estar presente em dois ou mais arquivos, como mostrado na Figura 4.27(d) com o bloco 5. Se qualquer um desses arquivos for removido, o bloco 5 será colocado na lista de livres, levando a uma situação na qual o mesmo bloco estará ao mesmo tempo em uso e livre. Se ambos os arquivos forem removidos, o bloco será colocado na lista de livres duas vezes.

A ação apropriada para o verificador de sistema de arquivos é alocar um bloco livre, copiar os conteúdos do bloco 5 nele e inserir a cópia em um dos arquivos. Dessa maneira, o conteúdo de informação dos arquivos ficará inalterado (embora quase certamente adulterado), mas a estrutura do sistema de arquivos ao menos ficará consistente. O erro deve ser reportado, a fim de permitir que o usuário inspecione o dano.

Além de conferir para ver se cada bloco está contabilizado corretamente, o verificador do sistema de arquivos também confere o sistema de diretórios. Ele,

FIGURA 4.27 Estados do sistema de arquivos. (a) Consistente. (b) Bloco desaparecido. (c) Bloco duplicado na lista de livres. (d) Bloco de dados duplicados.



também, usa uma tabela de contadores, mas esses são por arquivo, em vez de por bloco. Ele começa no diretório-raiz e recursivamente percorre a árvore, inspecionando cada diretório no sistema de arquivos. Para cada i-node em cada diretório, ele incrementa um contador para contar o uso do arquivo. Lembre-se de que por causa de ligações estritas, um arquivo pode aparecer em dois ou mais diretórios. Ligações simbólicas não contam e não fazem que o contador incremente para o arquivo-alvo.

Quando o verificador tiver concluído, ele terá uma lista, indexada pelo número do i-node, dizendo quantos diretórios contém cada arquivo. Ele então compara esses números com as contagens de ligações armazenadas nos próprios i-nodes. Essas contagens começam em 1 quando um arquivo é criado e são incrementadas cada vez que uma ligação (estrita) é feita para o arquivo. Em um sistema de arquivos consistente, ambas as contagens concordarão. No entanto, dois tipos de erros podem ocorrer: a contagem de ligações no i-node pode ser alta demais ou baixa demais.

Se a contagem de ligações for mais alta do que o número de entradas de diretório, então mesmo que todos os arquivos sejam removidos dos diretórios, a contagem ainda será diferente de zero e o i-node não será removido. Esse erro não é sério, mas desperdiça espaço no disco com arquivos que não estão em diretório algum. Ele deve ser reparado atribuindo-se o valor correto à contagem de ligações no i-node.

O outro erro é potencialmente catastrófico. Se duas entradas de diretório estão ligadas a um arquivo, mas os i-nodes dizem que há apenas uma, quando qualquer uma das entradas de diretório for removida, a contagem do i-node irá para zero. Quando uma contagem de i-node vai para zero, o sistema de arquivos a marca como inutilizada e libera todos os seus blocos. Essa ação resultará em um dos diretórios agora apontando para um i-node não usado, cujos blocos logo podem ser atribuídos a outros arquivos. Outra vez, a solução é apenas forçar a contagem de ligações no i-node a assumir o número real de entradas de diretório.

Essas duas operações, conferir os blocos e conferir os diretórios, muitas vezes são integradas por razões de eficiência (por exemplo, apenas uma verificação nos i-nodes é necessária). Outras verificações também são possíveis. Por exemplo, diretórios têm um formato definido, com números de i-nodes e nomes em ASCII. Se um número de i-node é maior do que o número de i-nodes no disco, o diretório foi danificado.

Além disso, cada i-node tem um modo, alguns dos quais são legais, mas estranhos, como o 0007, que possibilita ao proprietário e ao seu grupo não terem acesso a nada, mas permite que pessoas de fora leiam, escrevam e executem o arquivo. Pode ser útil ao menos reportar arquivos que dão aos usuários de fora mais direitos do que ao proprietário. Diretórios com mais de, digamos, 1.000 entradas também são suspeitos. Arquivos localizados nos diretórios de usuários, mas que são de propriedade do superusuário e que tenham o bit SETUID em 1, são problemas de segurança potenciais porque tais arquivos adquirem os poderes do superusuário quando executados por qualquer usuário. Com um pouco de esforço, é possível montar uma lista bastante longa de situações tecnicamente legais, mas peculiares, que vale a pena relatar.

Os parágrafos anteriores discutiram o problema de proteger o usuário contra quedas no sistema. Alguns sistemas de arquivos também se preocupam em proteger o usuário contra si mesmo. Se o usuário quiser digitar

```
rm *.o
```

para remover todos os arquivos terminando com *.o* (arquivos-objeto gerados pelo compilador), mas acidentalmente digita

```
rm * .o
```

(observe o espaço após o asterisco), *rm* removerá todos os arquivos no diretório atual e então reclamará que não pode encontrar *.o*. No Windows, os arquivos que são removidos são colocados na cesta de reciclagem (um diretório especial), do qual eles podem ser recuperados mais tarde se necessário. É claro, nenhum espaço é liberado até que eles sejam realmente removidos desse diretório.

4.4.4 Desempenho do sistema de arquivos

O acesso ao disco é muito mais lento do que o acesso à memória. Ler uma palavra de 32 bits de memória pode levar 10 ns. A leitura de um disco rígido pode chegar a 100 MB/s, o que é quatro vezes mais lento por palavra de 32 bits, mas a isso têm de ser acrescentados 5-10 ms para buscar a trilha e então esperar pelo setor desejado para chegar sob a cabeça de leitura. Se apenas uma única palavra for necessária, o acesso à memória será da ordem de um milhão de vezes mais rápido que o acesso ao disco. Como consequência dessa diferença em tempo de acesso, muitos sistemas de arquivos foram projetados com várias otimizações para melhorar o desempenho. Nesta seção cobriremos três delas.

Cache de blocos

A técnica mais comum usada para reduzir os acessos ao disco é a **cache de blocos** ou **cache de buffer**. (A palavra "cache" é pronunciada como se escreve e é derivada do verbo francês *cacher*, que significa "esconder".) Nesse contexto, uma cache é uma coleção de blocos que logicamente pertencem ao disco, mas estão sendo mantidas na memória por razões de segurança.

Vários algoritmos podem ser usados para gerenciar a cache, mas um comum é conferir todas as solicitações para ver se o bloco necessário está na cache. Se estiver, o pedido de leitura pode ser satisfeito sem acesso ao disco. Se o bloco não estiver, primeiro ele é lido na cache e então copiado para onde quer que seja necessário. Solicitações subsequentes para o mesmo bloco podem ser satisfeitas a partir da cache.

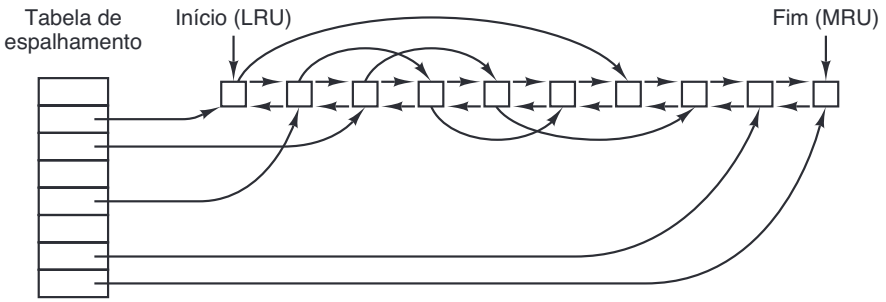
A operação da cache está ilustrada na Figura 4.28. Como há muitos (seguidamente milhares) blocos na cache, alguma maneira é necessária para determinar rapidamente se um dado bloco está presente. A maneira usual é mapear o dispositivo e endereço de disco e olhar o resultado em uma tabela de espalhamento. Todos os blocos com o mesmo valor de espalhamento são encadeados em uma lista de maneira que a cadeia de colisão possa ser seguida.

Quando um bloco tem de ser carregado em uma cache cheia, alguns blocos têm de ser removidos (e reescritos para o disco se eles foram modificados depois de trazidos para o disco). Essa situação é muito parecida com a paginação, e todos os algoritmos de substituição de páginas usuais descritos no Capítulo 3, como FIFO, segunda chance e LRU, são aplicáveis. Uma diferença bem-vinda entre a paginação e a cache de blocos é que as referências de cache são relativamente raras, de maneira que é viável manter todos os blocos na ordem exata do LRU com listas encadeadas.

Na Figura 4.28, vemos que além das colisões encadeadas da tabela de espalhamento, há também uma lista bidirecional ligando todos os blocos na ordem de uso, com o menos recentemente usado na frente dessa lista e o mais recentemente usado no fim. Quando um bloco é referenciado, ele pode ser removido da sua posição na lista bidirecional e colocado no fim. Dessa maneira, a ordem do LRU exata pode ser mantida.

Infelizmente, há um problema. Agora que temos uma situação na qual o LRU exato é possível, ele passa a ser indesejável. O problema tem a ver com as quedas no sistema e consistência do sistema de arquivos

FIGURA 4.28 As estruturas de dados da cache de buffer.



discutidas na seção anterior. Se um bloco crítico, como um bloco do i-node, é lido na cache e modificado, mas não reescrito para o disco, uma queda deixará o sistema de arquivos em estado inconsistente. Se o bloco do i--node for colocado no fim da cadeia do LRU, pode levar algum tempo até que ele chegue à frente e seja reescrito para o disco.

Além disso, alguns blocos, como blocos de i-nodes, raramente são referenciados duas vezes dentro de um intervalo curto de tempo. Essas considerações levam a um esquema de LRU modificado, tomando dois fatores em consideração:

1. É provável que o bloco seja necessário logo novamente?
2. O bloco é essencial para a consistência do sistema de arquivos?

Para ambas as questões, os blocos podem ser divididos em categorias como blocos de i-nodes, indiretos, de diretórios, de dados totalmente preenchidos e de dados parcialmente preenchidos. Blocos que provavelmente não serão necessários logo de novo irão para a frente, em vez de para o fim da lista do LRU, de maneira que seus buffers serão reutilizados rapidamente. Blocos que talvez sejam necessários logo outra vez, como o bloco parcialmente preenchido que está sendo escrito, irão para o fim da lista, de maneira que permanecerão por ali um longo tempo.

A segunda questão é independente da primeira. Se o bloco for essencial para a consistência do sistema de arquivos (basicamente, tudo exceto blocos de dados) e foi modificado, ele deve ser escrito para o disco imediatamente, não importando em qual extremidade da lista LRU será inserido. Ao escrever blocos críticos rapidamente, reduzimos muito a probabilidade de que uma queda arruíne o sistema de arquivos. Embora um usuário possa ficar descontente se um de seus arquivos for arruinado em uma queda, é provável que ele fique muito mais descontente se todo o sistema de arquivos for perdido.

Mesmo com essa medida para manter intacta a integridade do sistema de arquivos, é indesejável manter blocos de dados na cache por tempo demais antes de serem escritos. Considere o drama de alguém que está usando um computador pessoal para escrever um livro. Mesmo que o nosso escritor periodicamente diga ao editor para escrever para o disco o arquivo que está sendo editado, há uma boa chance de que tudo ainda esteja na cache e nada no disco. Se o sistema cair, a estrutura do sistema de arquivos não será corrompida, mas um dia inteiro de trabalho será perdido.

Essa situação não precisa acontecer com muita frequência para que tenhamos um usuário descontente. Sistemas adotam duas abordagens para lidar com isso. A maneira UNIX é ter uma chamada de sistema, sync, que força todos os blocos modificados para o disco imediatamente. Quando o sistema é inicializado, um programa, normalmente chamado *update*, é inicializado no segundo plano para adentrar um laço infinito que emite chamadas sync, dormindo por 30 s entre chamadas. Como consequência, não mais do que 30 s de trabalho são perdidos pela quebra.

Embora o Windows tenha agora uma chamada de sistema equivalente a sync, chamada FlushFileBuffers, no passado ele não tinha. Em vez disso, ele tinha uma estratégia diferente que era, em alguns aspectos, melhor do que a abordagem do UNIX (e outros, pior). O que ele fazia era escrever cada bloco modificado para o disco tão logo ele fosse escrito para a cache. Caches nas quais todos os blocos modificados são escritos de volta para o disco imediatamente são chamadas de **caches de escrita direta (write-through caches)**. Elas exigem mais E/S de disco do que caches que não são de escrita direta.

A diferença entre essas duas abordagens pode ser vista quando um programa escreve um bloco totalmente preenchido de 1 KB, um caractere por vez. O UNIX coletará todos os caracteres na cache e escreverá o bloco uma vez a cada 30 s, ou sempre que o bloco for removido da cache. Com uma cache de escrita direta, há um acesso de disco para cada caractere escrito. É claro, a maioria dos programas trabalha com buffer interno, então eles normalmente não escrevem um caractere, mas uma linha ou unidade maior em cada chamada de sistema write.

Uma consequência dessa diferença na estratégia de cache é que apenas remover um disco de um sistema UNIX sem realizar sync quase sempre resultará em dados perdidos, e frequentemente um sistema de arquivos corrompido também. Com as caches de escrita direta não há problema algum. Essas estratégias diferentes foram escolhidas porque o UNIX foi desenvolvido em um ambiente no qual todos os discos eram rígidos e não removíveis, enquanto o primeiro sistema de arquivos Windows foi herdado do MS-DOS, que teve seu início no mundo dos discos flexíveis. Como os discos rígidos tornaram-se a norma, a abordagem UNIX, com sua eficiência melhor (mas pior confiabilidade), tornou-se a norma, e também é usada agora no Windows para discos rígidos. No entanto, o NTFS toma outras medidas (por exemplo, journaling) para incrementar a confiabilidade, como discutido anteriormente.

Alguns sistemas operacionais integram a cache de buffer com a cache de páginas. Isso é especialmente atraente quando arquivos mapeados na memória são aceitos. Se um arquivo é mapeado na memória, então algumas das suas páginas podem estar na memória por causa de uma paginação por demanda. Tais páginas dificilmente são diferentes dos blocos de arquivos na cache do buffer. Nesse caso, podem ser tratadas da mesma maneira, com uma cache única para ambos os blocos de arquivos e páginas.

Leitura antecipada de blocos

Uma segunda técnica para melhorar o desempenho percebido do sistema de arquivos é tentar transferir blocos para a cache antes que eles sejam necessários para aumentar a taxa de acertos. Em particular, muitos arquivos são lidos sequencialmente. Quando se pede a um sistema de arquivos para obter o bloco k em um arquivo, ele o faz, mas quando termina, faz uma breve verificação na cache para ver se o bloco $k + 1$ já está ali. Se não estiver, ele programa uma leitura para o bloco $k + 1$ na esperança de que, quando ele for necessário, já terá chegado na cache. No mínimo, ele já estará a caminho.

É claro, essa estratégia de leitura antecipada funciona apenas para arquivos que estão de fato sendo lidos sequencialmente. Se um arquivo estiver sendo acessado aleatoriamente, a leitura antecipada não ajuda. Na realidade, ela piora a situação, pois emperra a largura de banda do disco, fazendo leituras em blocos inúteis e removendo blocos potencialmente úteis da cache (e talvez emperrando mais ainda a largura de banda escrevendo os blocos de volta para o disco se eles estiverem sujos). Para ver se a leitura antecipada vale a pena ser feita, o sistema de arquivos pode monitorar os padrões de acesso para cada arquivo aberto. Por exemplo, um bit associado com cada arquivo pode monitorar se o arquivo está em "modo de acesso sequencial" ou "modo de acesso aleatório". De início, é dado o benefício da dúvida para o arquivo e ele é colocado no modo de acesso sequencial. No entanto, sempre que uma busca é feita, o bit é removido. Se as leituras sequenciais começarem de novo, o bit é colocado em 1 novamente. Dessa maneira, o sistema de arquivos pode formular um palpite razoável sobre se ele deve ler antecipadamente ou não. Se ele cometer algum erro de vez em quando, não é um desastre, apenas um pequeno desperdício de largura de banda de disco.

Redução do movimento do braço do disco

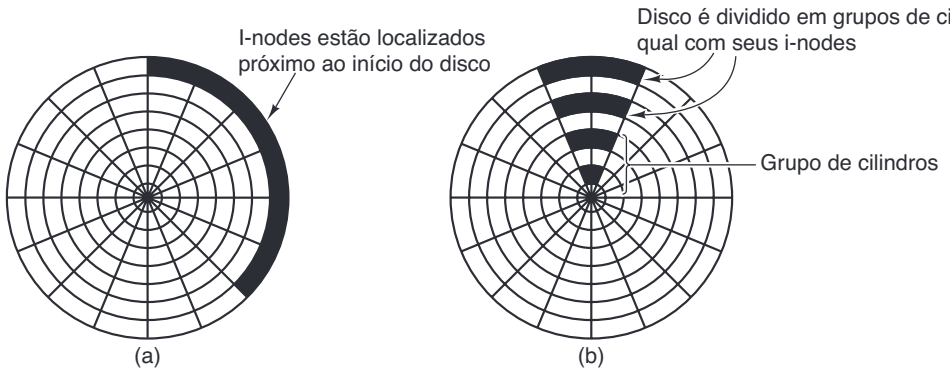
A cache e a leitura antecipada não são as únicas maneiras de incrementar o desempenho do sistema de arquivos. Outra técnica importante é reduzir o montante de movimento do braço do disco colocando blocos que têm mais chance de serem acessados em sequência próximos uns dos outros, de preferência no mesmo cilindro. Quando um arquivo de saída é escrito, o sistema de arquivos tem de alocar os blocos um de cada vez, conforme a demanda. Se os blocos livres forem registrados em um mapa de bits, e todo o mapa de bits estiver na memória principal, será bastante fácil escolher um bloco livre o mais próximo possível do bloco anterior. Com uma lista de blocos livres, na qual uma parte está no disco, é muito mais difícil alocar blocos próximos juntos.

No entanto, mesmo com uma lista de blocos livres, algum agrupamento de blocos pode ser conseguido. O truque é monitorar o armazenamento do disco, não em blocos, mas em grupos de blocos consecutivos. Se todos os setores consistirem em 512 bytes, o sistema poderia usar blocos de 1 KB (2 setores), mas alocar o armazenamento de disco em unidades de 2 blocos (4 setores). Isso não é o mesmo que ter blocos de disco de 2 KB, já que a cache ainda usaria blocos de 1 KB e as transferências de disco ainda seriam de 1 KB, mas a leitura de um arquivo sequencialmente em um sistema de outra maneira ocioso reduziria o número de buscas por um fator de dois, melhorando consideravelmente o desempenho. Uma variação sobre o mesmo tema é levar em consideração o posicionamento rotacional. Quando aloca blocos, o sistema faz uma tentativa de colocar blocos consecutivos em um arquivo no mesmo cilindro.

Outro gargalo de desempenho em sistemas que usam i-nodes (ou qualquer equivalente a eles) é que a leitura mesmo de um arquivo curto exige dois acessos de disco: um para o i-node e outro para o bloco. A localização usual do i-node é mostrada da Figura 4.29(a). Aqui todos os i-nodes estão próximos do início do disco, então a distância média entre um i-node e seus blocos será metade do número de cilindros, exigindo longas buscas.

Uma melhora simples de desempenho é colocar os i-nodes no meio do disco, em vez de no início, reduzindo assim a busca média entre o i-node e o primeiro bloco por um fator de dois. Outra ideia, mostrada na Figura 4.29(b), é dividir o disco em grupos de cilindros, cada um com seus próprios i-nodes, blocos e lista de livres (MCKUSICK et al., 1984). Ao criar um arquivo novo, qualquer i-node pode ser escolhido, mas uma tentativa é feita para encontrar um bloco no

FIGURA 4.29 (a) I-nodes posicionados no início do disco. (b) Disco dividido em grupos de cilindros, cada um com seus próprios blocos e i-nodes.



mesmo grupo de cilindros que o i-node. Se nenhum estiver disponível, então um bloco em um grupo de cilindros próximo é usado.

É claro, o movimento do braço do disco e o tempo de rotação são relevantes somente se o disco os tem. Mais e mais computadores vêm equipados com **discos de estado sólido (SSDs — Solid State Disks)** que não têm parte móvel alguma. Para esses discos, construídos com a mesma tecnologia dos flash cards, acessos aleatórios são tão rápidos quanto os sequenciais e muitos dos problemas dos discos tradicionais deixam de existir. Infelizmente, surgem novos problemas. Por exemplo, SSDs têm propriedades peculiares em suas operações de leitura, escrita e remoção. Em particular, cada bloco pode ser escrito apenas um número limitado de vezes, portanto um grande cuidado é tomado para dispersar uniformemente o desgaste sobre o disco.

4.4.5 Desfragmentação de disco

Quando o sistema operacional é inicialmente instalado, os programas e os arquivos que ele precisa são instalados de modo consecutivo começando no início do disco, cada um seguindo diretamente o anterior. Todo o espaço de disco livre está em uma única unidade contígua seguindo os arquivos instalados. No entanto, à medida que o tempo passa, arquivos são criados e removidos, e o disco prejudica-se com a fragmentação, com arquivos e espaços vazios por toda parte. Em consequência, quando um novo arquivo é criado, os blocos usados para isso podem estar espalhados por todo o disco, resultando em um desempenho ruim.

O desempenho pode ser restaurado movendo os arquivos a fim de deixá-los contíguos e colocando todo (ou pelo menos a maior parte) o espaço livre em uma ou mais regiões contíguas no disco. O Windows tem um programa, *defrag*, que faz precisamente isso. Os usuários do Windows devem executá-lo com regularidade, exceto em SSDs.

A desfragmentação funciona melhor em sistemas de arquivos que têm bastante espaço livre em uma região contígua ao fim da partição. Esse espaço permite que o programa de desfragmentação selecione arquivos fragmentados próximos do início da partição e copie todos os seus blocos para o espaço livre. Fazê-lo libera um bloco contíguo de espaço próximo do início da partição na qual o original ou outros arquivos podem ser colocados contiguamente. O processo pode então ser repetido com o próximo pedaço de espaço de disco etc.

Alguns arquivos não podem ser movidos, incluindo o arquivo de paginação, o arquivo de hibernação e o log de journaling, pois a administração que seria necessária para fazê-lo daria mais trabalho do que seu valor. Em alguns sistemas, essas áreas são contíguas e de tamanho fixo de qualquer maneira, então elas não precisam ser desfragmentadas. O único momento em que sua falta de mobilidade é um problema é quando elas estão localizadas próximas do fim da partição e o usuário quer reduzir o tamanho da partição. A única maneira de solucionar esse problema é removê-las completamente, redimensionar a partição e então recriá-las depois.

Os sistemas de arquivos Linux (especialmente ext2 e ext3) geralmente sofrem menos com a desfragmentação do que os sistemas Windows pela maneira que os blocos de discos são selecionados, então a desfragmentação manual raramente é exigida. Também, os SSDs não sofrem de maneira alguma com a fragmentação. Na realidade, desfragmentar um SSD é contraprodutivo. Não apenas não há ganho em desempenho, como os SSDs se desgastam; portanto, desfragmentá-los apenas encurta sua vida.

4.5 Exemplos de sistemas de arquivos

Nas próximas seções discutiremos vários exemplos de sistemas de arquivos, desde os bastante simples aos mais sofisticados. Como os sistemas de arquivos UNIX modernos e o sistema de arquivos nativo do Windows 8 são cobertos no capítulo sobre o UNIX (Capítulo 10) e no capítulo sobre o Windows 8 (Capítulo 11), não os cobriremos aqui. Examinaremos, no entanto, seus predecessores a seguir.

4.5.1 O sistema de arquivos do MS-DOS

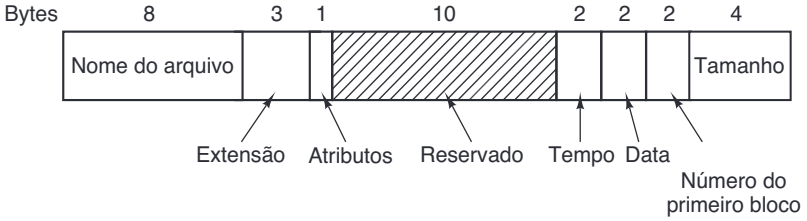
O sistema de arquivos do MS-DOS é o sistema com o qual os primeiros PCs da IBM vinham instalados. Foi o principal sistema de arquivos até o Windows 98 e o Windows ME. Ainda é aceito no Windows 2000, Windows XP e Windows Vista, embora não seja mais padrão nos novos PCs exceto para discos flexíveis. No entanto, ele e uma extensão dele (FAT-32) tornaram-se amplamente usados para muitos sistemas embarcados. Muitos players de MP3 o usam exclusivamente, além de câmeras digitais. O popular iPod da Apple o usa como o sistema de arquivos padrão, embora hackers que conhecem do assunto conseguem reformatar o iPod e instalar um sistema de arquivos diferente. Desse modo, o número de dispositivos eletrônicos usando o sistema de arquivos MS-DOS é muito maior agora do que em qualquer momento no passado e, decerto, muito maior do que o número usando o sistema de arquivos NTFS mais moderno. Por essa razão somente, vale a pena examiná-lo em detalhe.

Para ler um arquivo, um programa de MS-DOS deve primeiro fazer uma chamada de sistema open para abri-lo. A chamada de sistema open especifica um caminho, o qual pode ser absoluto ou relativo ao diretório de trabalho atual. O caminho é analisado componente a componente até que o diretório final seja localizado e lido na memória. Ele então é vasculhado para encontrar o arquivo a ser aberto.

Embora os diretórios de MS-DOS tenham tamanhos variáveis, eles usam uma entrada de diretório de 32 bytes de tamanho fixo. O formato de uma entrada de diretório de MS-DOS é mostrado na Figura 4.30. Ele contém o nome do arquivo, atributos, data e horário de criação, bloco de partida e tamanho exato do arquivo. Nomes de arquivos mais curtos do que 8 + 3 caracteres são ajustados à esquerda e preenchidos com espaços à direita, separadamente em cada campo. O campo *Atributos* é novo e contém bits para indicar que um arquivo é somente de leitura, precisa ser arquivado, está escondido ou é um arquivo de sistema. Arquivos somente de leitura não podem ser escritos. Isso é para protegê-los de danos acidentais. O bit arquivado não tem uma função de sistema operacional real (isto é, o MS-DOS não o examina ou o altera). A intenção é permitir que programas de arquivos em nível de usuário o desliguem quando efetuarem o backup de um arquivo e que os outros programas o liguem quando modificarem um arquivo. Dessa maneira, um programa de backup pode apenas examinar o bit desse atributo em cada arquivo para ver quais arquivos devem ser copiados. O bit oculto pode ser alterado para evitar que um arquivo apareça nas listagens de diretório. Seu uso principal é evitar confundir usuários novatos com arquivos que eles possam não compreender. Por fim, o bit sistema também oculta arquivos. Além disso, arquivos de sistema não podem ser removidos acidentalmente usando o comando *del*. Os principais componentes do MS-DOS têm esse bit ligado.

A entrada de diretório também contém a data e o horário em que o arquivo foi criado ou modificado pela última vez. O tempo é preciso apenas até ±2 segundos, pois ele está armazenado em um campo de 2 bytes, que pode armazenar somente 65.536 valores únicos (um dia contém 86.400 segundos). O campo do tempo é subdividido em segundos (5 bits), minutos (6 bits) e horas (5 bits). A data conta em dias usando três subcampos: dia (5 bits), mês (4 bits) e ano — 1980 (7 bits). Com um número de 7 bits para o ano e o tempo começando em 1980, o maior valor que pode ser representado é 2107. Então, o MS-DOS tem um problema Y2108 em si. Para evitar a catástrofe, seus usuários devem atentar para esse problema o mais cedo possível. Se o MS-DOS tivesse

FIGURA 4.30 A entrada de diretório do MS-DOS.



usado os campos data e horário combinados como um contador de 32 bits, ele teria representado cada segundo exatamente e atrasado a catástrofe até 2116.

O MS-DOS armazena o tamanho do arquivo como um número de 32 bits, portanto na teoria os arquivos podem ser de até 4 GB. No entanto, outros limites (descritos a seguir) restringem o tamanho máximo do arquivo a 2 GB ou menos. Uma parte surpreendentemente grande da entrada (10 bytes) não é usada.

O MS-DOS monitora os blocos de arquivos mediante uma tabela de alocação de arquivos na memória principal. A entrada do diretório contém o número do primeiro bloco de arquivos. Esse número é usado como um índice em uma FAT de 64 K entradas na memória principal. Seguindo o encadeamento, todos os blocos podem ser encontrados. A operação da FAT está ilustrada na Figura 4.12.

O sistema de arquivos FAT vem em três versões: FAT-12, FAT-16 e FAT-32, dependendo de quantos bits um endereço de disco contém. Na realidade, FAT-32 não é um nome adequado, já que apenas os 28 bits menos significativos dos endereços de disco são usados. Ele deveria chamar-se FAT-28, mas as potências de dois soam bem melhor.

Outra variante do sistema de arquivos FAT é o exFAT, que a Microsoft introduziu para dispositivos removíveis grandes. A Apple licenciou o exFAT, de maneira que há um sistema de arquivos moderno que pode ser usado para transferir arquivos entre computadores Windows e OS X. Como o exFAT é de propriedade da Microsoft e a empresa não liberou a especificação, não o discutiremos mais aqui.

Para todas as FATs, o bloco de disco pode ser alterado para algum múltiplo de 512 bytes (possivelmente diferente para cada partição), com o conjunto de tamanhos de blocos permitidos (chamado **cluster sizes** — tamanhos de aglomerado — pela Microsoft) sendo diferente para cada variante. A primeira versão do MS-DOS usava a FAT-12 com blocos de 512 bytes, dando um tamanho de partição máximo de $2^{12} \times 512$ bytes (na realidade somente 4086 × 512 bytes, pois 10 dos endereços de disco foram usados como marcadores especiais, como fim de arquivo, bloco defeituoso etc.). Com esses parâmetros, o tamanho de partição de disco máximo era em torno de 2 MB e o tamanho da tabela FAT na memória era de 4096 entradas de 2 bytes cada. Usar uma entrada de tabela de 12 bits teria sido lento demais.

Esse sistema funcionava bem para discos flexíveis, mas quando os discos rígidos foram lançados, ele tornou-se um problema. A Microsoft solucionou o problema permitindo tamanhos de blocos adicionais de 1 KB, 2 KB e 4 KB. Essa mudança preservou a estrutura e o tamanho da tabela FAT-12, mas permitiu partições de disco de até 16 MB.

Como o MS-DOS dava suporte para quatro partições de disco por unidade de disco, o novo sistema de arquivos FAT-12 funcionava para discos de até 64 MB. Além disso, algo tinha de ceder. O que aconteceu foi a introdução do FAT-16, com ponteiros de disco de 16 bits. Adicionalmente, tamanhos de blocos de 8 KB, 16 KB e 32 KB foram permitidos. (32.768 é a maior potência de dois que pode ser representada em 16 bits.) A tabela FAT-16 ocupava 128 KB de memória principal o tempo inteiro, mas com as memórias maiores então disponíveis, ela era amplamente usada e logo substituiu o sistema de arquivos FAT-12. A maior partição de disco a que o FAT-16 pode dar suporte é 2 GB (64 K entradas de 32 KB cada) e o maior disco, 8 GB, a saber quatro partições de 2 GB cada. Por um bom tempo, isso foi o suficiente.

Mas não para sempre. Para cartas comerciais, esse limite não é um problema, mas para armazenar vídeos digitais usando o padrão DV, um arquivo de 2 GB contém apenas um pouco mais de 9 minutos de vídeo. Como um disco de PC suporta apenas quatro partições, o maior vídeo que pode ser armazenado em disco é de mais ou menos 38 minutos, não importa o tamanho do disco. Esse limite também significa que o maior vídeo que pode ser editado on-line é de menos de 19 minutos, pois ambos os arquivos de entrada e saída são necessários.

A partir da segunda versão do Windows 95, foi introduzido o sistema de arquivos FAT-32 com seus endereços de disco de 28 bits e a versão do MS-DOS subjacente ao Windows 95 foi adaptada para dar suporte à FAT-32. Nesse sistema, as partições poderiam ser teoricamente $2^{28} \times 2^{15}$ bytes, mas na realidade elas eram limitadas a 2 TB (2048 GB), pois internamente o sistema monitora os tamanhos de partições em setores de 512 bytes usando um número de 32 bits, e $2^9 \times 2^{32}$ é 2 TB. O tamanho máximo da partição para vários tamanhos de blocos e todos os três tipos FAT é mostrado na Figura 4.31.

Além de dar suporte a discos maiores, o sistema de arquivos FAT-32 tem duas outras vantagens sobre o FAT-16. Primeiro, um disco de 8 GB usando FAT-32 pode ter uma única partição. Usando o FAT-16 ele tem quatro partições, o que aparece para o usuário do Windows como *C:*, *D:*, *E:* e *F:* unidades de disco lógicas. Cabe ao usuário decidir qual arquivo colocar em qual unidade e monitorar o que está onde.

A outra vantagem do FAT-32 sobre o FAT-16 é que para um determinado tamanho de partição de disco, um

FIGURA 4.31 Tamanho máximo da partição para diferentes tamanhos de blocos. As caixas vazias representam combinações proibidas.

Tamanho do bloco	FAT-12	FAT-16	FAT-32
0,5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

tamanho de bloco menor pode ser usado. Por exemplo, para uma partição de disco de 2 GB, o FAT-16 deve usar blocos de 32 KB; de outra maneira, com apenas 64K endereços de disco disponíveis, ele não pode cobrir toda a partição. Em comparação, o FAT-32 pode usar, por exemplo, blocos de 4 KB para uma partição de disco de 2 GB. A vantagem de um tamanho de bloco menor é que a maioria dos arquivos é muito mais curta do que 32 KB. Se o tamanho do bloco for 32 KB, um arquivo de 10 bytes imobiliza 32 KB de espaço de disco. Se o arquivo médio for, digamos, 8 KB, então com um bloco de 32 KB, três quartos do disco serão desperdiçados, o que não é uma maneira muito eficiente de se usar o disco. Com um arquivo de 8 KB e um bloco de 4 KB, não há desperdício de disco, mas o preço pago é mais RAM consumida pelo FAT. Com um bloco de 4 KB e uma partição de disco de 2 GB, há 512 K blocos, de maneira que o FAT precisa ter 512K entradas na memória (ocupando 2 MB de RAM).

O MS-DOS usa a FAT para monitorar blocos de disco livres. Qualquer bloco que no momento não esteja alocado é marcado com um código especial. Quando o MS-DOS precisa de um novo bloco de disco, ele pesquisa a FAT para uma entrada contendo esse código. Desse modo, não são necessários nenhum mapa de bits ou lista de livres.

4.5.2 O sistema de arquivos do UNIX V7

Mesmo as primeiras versões do UNIX tinham um sistema de arquivos multiusuário bastante sofisticado, já que era derivado do MULTICS. A seguir discutiremos o sistema de arquivos V7, aquele do PDP-11 que tornou o UNIX famoso. Examinaremos um sistema de arquivos UNIX moderno no contexto do Linux no Capítulo 10.

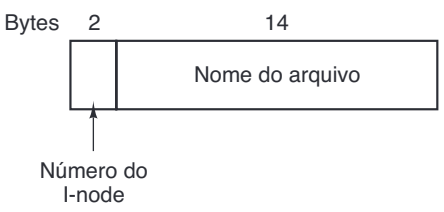
O sistema de arquivos tem forma de uma árvore começando no diretório-raiz, com a adição de ligações, formando um gráfico orientado acíclico. Nomes de arquivos podem ter até 14 caracteres e contêm quaisquer caracteres ASCII exceto / (porque se trata de um separador entre componentes em um caminho) e NUL (pois é usado para preencher os espaços que sobram nos nomes mais curtos que 14 caracteres). NUL tem o valor numérico de 0.

Uma entrada de diretório UNIX contém uma entrada para cada arquivo naquele diretório. Cada entrada é extremamente simples, pois o UNIX usa o esquema i-node ilustrado na Figura 4.13. Uma entrada de diretório contém apenas dois campos: o nome do arquivo (14 bytes) e o número do i-node para aquele arquivo (2 bytes), como mostrado na Figura 4.32. Esses parâmetros limitam o número de arquivos por sistema a 64 K.

Assim como o i-node da Figura 4.13, o i-node do UNIX contém alguns atributos. Os atributos contêm o tamanho do arquivo, três horários (criação, último acesso e última modificação), proprietário, grupo, informação de proteção e uma contagem do número de entradas de diretórios que apontam para o i-node. Este último campo é necessário para as ligações. Sempre que uma ligação nova é feita para um i-node, o contador no i-node é incrementado. Quando uma ligação é removida, o contador é decrementado. Quando ele chega a 0, o i-node é reivindicado e os blocos de disco são colocados de volta na lista de livres.

O monitoramento dos blocos de disco é feito usando uma generalização da Figura 4.13 a fim de lidar com arquivos muito grandes. Os primeiros 10 endereços de disco são armazenados no próprio i-node, então para pequenos arquivos, todas as informações necessárias estão diretamente no i-node, que é buscado do disco para a memória principal quando o arquivo é aberto. Para arquivos um pouco maiores, um dos endereços no i-node é o de um bloco de disco chamado **bloco indireto simples**. Esse bloco contém endereços de disco adicionais. Se isso ainda não for suficiente, outro endereço no i-node, chamado **bloco indireto duplo**, contém o endereço de um bloco com uma lista de blocos indiretos

FIGURA 4.32 Uma entrada do diretório do UNIX V7.



simples. Cada um desses blocos indiretos simples aponta para algumas centenas de blocos de dados. Se mesmo isso não for suficiente, um **bloco indireto triplo** também pode ser usado. O quadro completo é dado na Figura 4.33.

Quando um arquivo é aberto, o sistema deve tomar o nome do arquivo fornecido e localizar seus blocos de disco. Vamos considerar como o nome do caminho */usr/ast/mbox* é procurado. Usaremos o UNIX como exemplo, mas o algoritmo é basicamente o mesmo para todos os sistemas de diretórios hierárquicos. Primeiro, o sistema de arquivos localiza o diretório-raiz. No UNIX o seu i-node está localizado em um local fixo no disco. A partir desse i-node, ele localiza o diretório-raiz, que pode estar em qualquer lugar, mas digamos bloco 1.

Em seguida, ele lê o diretório-raiz e procura o primeiro componente do caminho, *usr*, no diretório-raiz para encontrar o número do i-node do arquivo */usr*. Localizar um i-node a partir desse número é algo direto, já que cada i-node tem uma localização fixa no disco. A partir desse i-node, o sistema localiza o diretório para */usr* e pesquisa o componente seguinte, *ast*, nele. Quando encontrar a entrada para *ast*, ele terá o i-node para o diretório */usr/ast*. A partir desse i-node ele pode fazer uma busca no próprio diretório e localizar *mbox*. O i-node para esse arquivo é então lido na memória e mantido ali até o arquivo ser fechado. O processo de busca está ilustrado na Figura 4.34.

Nomes de caminhos relativos são procurados da mesma maneira que os absolutos, apenas partindo do diretório de trabalho em vez de do diretório-raiz. Todo diretório tem entradas para . e .. que são colocadas ali quando o diretório é criado. A entrada . tem o número de i-node para o diretório atual, e a entrada para .. tem o número de i-node para o diretório pai. Desse modo, uma rotina procurando *../dick/prog.c* apenas procura .. no diretório de trabalho, encontra o número de i-node do diretório pai e busca pelo diretório *dick*. Nenhum mecanismo especial é necessário para lidar com esses nomes. No que diz respeito ao sistema de diretórios, eles são apenas cadeias ASCII comuns, como qualquer outro nome. A única questão a ser observada aqui é que .. no diretório-raiz aponta para si mesmo.

4.5.3 Sistemas de arquivos para CD-ROM

Como nosso último exemplo de um sistema de arquivos, vamos considerar aqueles usados nos CD-ROMs. Eles são particularmente simples, pois foram projetados para meios de escrita única. Entre outras coisas, por exemplo, eles não têm provisão para monitorar blocos livres, pois em um arquivo de CD-ROM arquivos não podem ser liberados ou adicionados após o disco ter sido fabricado. A seguir examinaremos o principal tipo de sistema de arquivos para CD-ROMs e duas de suas extensões. Embora os CD-ROMs estejam ultrapassados, eles são também simples, e os sistemas de arquivos usados em DVDs e Blu-ray são baseados nos usados para CD-ROMs.

FIGURA 4.33 Um i-node do UNIX.

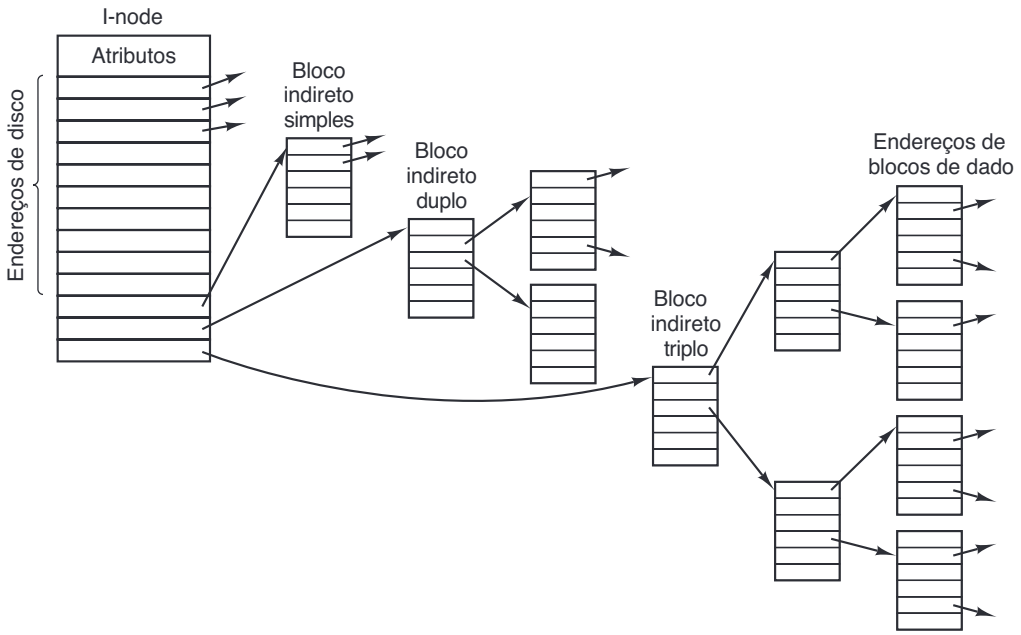
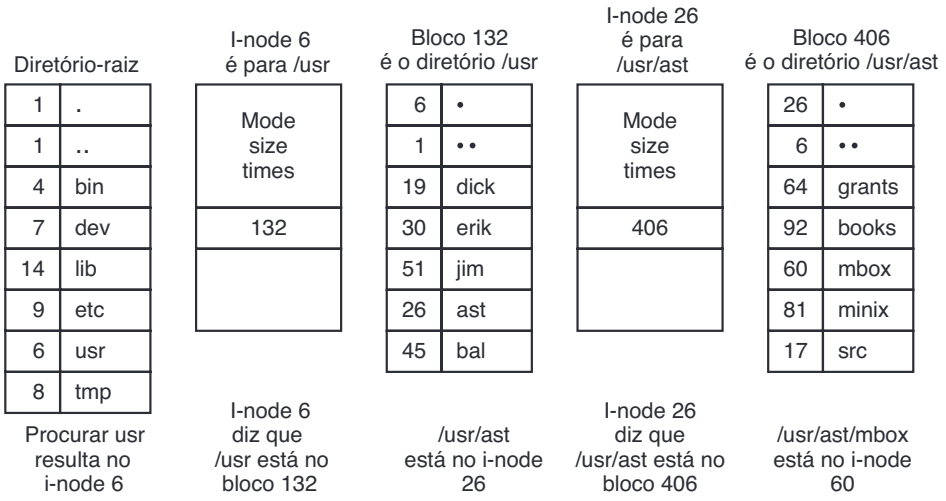


FIGURA 4.34 Os passos para pesquisar em */usr/ast/mbox*.



Alguns anos após o CD-ROM ter feito sua estreia, foi introduzido o CD-R (**CD Recordable** — CD gravável). Diferentemente do CD-ROM, ele permite adicionar arquivos após a primeira gravação, que são apenas adicionados ao final do CD-R. Arquivos nunca são removidos (embora o diretório possa ser atualizado para esconder arquivos existentes). Como consequência desse sistema de arquivos "somente adicionar", as propriedades fundamentais não são alteradas. Em particular, todo o espaço livre encontra-se em uma única parte contígua no fim do CD.

O sistema de arquivos ISO 9660

O padrão mais comum para sistemas de arquivos de CD-ROM foi adotado como um Padrão Internacional em 1988 sob o nome **ISO 9660**. Virtualmente, todo CD-ROM no mercado é compatível com esse padrão, às vezes com extensões a serem discutidas a seguir. Uma meta desse padrão era tornar todo CD-ROM legível em todos os computadores, independente do ordenamento de bytes e do sistema operacional usado. Em consequência, algumas limitações foram aplicadas ao sistema de arquivos para possibilitar que os sistemas operacionais mais fracos (como MS-DOS) pudessem lê-los.

Os CD-ROMs não têm cilindros concêntricos como os discos magnéticos. Em vez disso, há uma única espiral contínua contendo os bits em uma sequência linear (embora seja possível buscar transversalmente às espirais). Os bits ao longo da espiral são divididos em blocos lógicos (também chamados setores lógicos) de 2352 bytes. Alguns desses bytes são para preâmbulos, correção de erros e outros destinos. A porção líquida (payload) de cada bloco lógico é 2048 bytes. Quando usados para música, os CDs têm as posições iniciais, finais e espaços entre as trilhas, mas eles não são usados para CD-ROMs de dados. Muitas vezes a posição de um bloco ao longo da espiral é representada em minutos e segundos. Ela pode ser convertida para um número de bloco linear usando o fator de conversão de 1 s = 75 blocos.

O ISO 9660 dá suporte a conjuntos de CD-ROM com até $2^{16} - 1$ CDs no conjunto. Os CD-ROMs individuais também podem ser divididos em volumes lógicos (partições). No entanto, a seguir, nos concentraremos no ISO 9660 para um único CD-ROM não particionado.

Todo CD-ROM começa com 16 blocos cuja função não é definida pelo padrão ISO 9660. Um fabricante de CD-ROMs poderia usar essa área para oferecer um programa de inicialização que permitisse que o computador fosse inicializado pelo CD-ROM, ou para algum outro propósito nefando. Em seguida vem um bloco contendo o **descritor de volume primário**, que contém algumas informações gerais sobre o CD-ROM. Entre essas informações estão o identificador do sistema (32 bytes), o identificador do volume (32 bytes), o identificador do editor (128 bytes) e o identificador do preparador dos dados (128 bytes). O fabricante pode preencher esses campos da maneira que quiser, exceto que somente letras maiúsculas, dígitos e um número muito pequeno de caracteres de pontuação podem ser usados para assegurar a compatibilidade entre as plataformas.

O descritor do volume primário também contém os nomes de três arquivos, que podem conter um resumo, uma notificação de direitos autorais e informações bibliográficas, respectivamente. Além disso, determinados números-chave também estão presentes, incluindo o tamanho do bloco lógico (normalmente 2048, mas

4096, 8192 e valores maiores de potências de 2 são permitidos em alguns casos), o número de blocos no CD-ROM e as datas de criação e expiração do CD-ROM. Por fim, o descritor do volume primário também contém uma entrada de diretório para o diretório-raiz, dizendo onde encontrá-lo no CD-ROM (isto é, em qual bloco ele começa). A partir desse diretório, o resto do sistema de arquivos pode ser localizado.

Além do descritor de volume primário, um CD-ROM pode conter um descritor de volume complementar. Ele contém informações similares ao descritor primário, mas não abordaremos essa questão aqui.

O diretório-raiz, e todos os outros diretórios, quanto a isso, consistem em um número variável de entradas, a última das quais contém um bit marcando-a como a entrada final. As entradas de diretório em si também são de tamanhos variáveis. Cada entrada de diretório consiste em 10 a 12 campos, dos quais alguns são em ASCII e outros são numéricos binários. Os campos binários são codificados duas vezes, uma com os bits menos significativos nos primeiros bytes — little-endian (usados nos Pentiums, por exemplo) e outra com os bits mais significativos nos primeiros bytes — big endian (usados nas SPARCs, por exemplo). Desse modo, um número de 16 bits usa 4 bytes e um número de 32 bits usa 8 bytes.

O uso dessa codificação redundante era necessário para evitar ferir os sentimentos alheios quando o padrão foi desenvolvido. Se o padrão tivesse estabelecido little-endian, então as pessoas de empresas cujos produtos eram big-endian se sentiriam desvalorizadas e não teriam aceitado o padrão. O conteúdo emocional de um CD-ROM pode, portanto, ser quantificado e mensurado exatamente em quilobytes/hora de espaço desperdiçado.

O formato de uma entrada de diretório ISO 9660 está ilustrado na Figura 4.35. Como entradas de diretório têm comprimentos variáveis, o primeiro campo é um byte indicando o tamanho da entrada. Esse byte é definido com o bit de ordem mais alta à esquerda para evitar qualquer ambiguidade.

Entradas de diretório podem opcionalmente ter atributos estendidos. Se essa prioridade for usada, o segundo byte indicará o tamanho dos atributos estendidos.

Em seguida vem o bloco inicial do próprio arquivo. Arquivos são armazenados como sequências contíguas de blocos, assim a localização de um arquivo é completamente especificada pelo bloco inicial e o tamanho, que está contido no próximo campo.

A data e o horário em que o CD-ROM foi gravado estão armazenados no próximo campo, com bytes separados para o ano, mês, dia, hora, minuto, segundo e zona do fuso horário. Os anos começam a contar em 1900, o que significa que os CD-ROMs sofrerão de um problema Y2156, pois o ano seguinte a 2155 será 1900. Esse problema poderia ter sido postergado com a definição da origem do tempo em 1988 (o ano que o padrão foi adotado). Se isso tivesse ocorrido, o problema teria sido postergado até 2244. Cada 88 anos extras ajuda.

O campo *Flags* contém alguns bits diversos, incluindo um para ocultar a entrada nas listagens (um atributo copiado do MS-DOS), um para distinguir uma entrada que é um arquivo de uma entrada que é um diretório, um para capacitar o uso dos atributos estendidos e um para marcar a última entrada em um diretório. Alguns outros bits também estão presentes nesse campo, mas não os abordaremos aqui. O próximo campo lida com a intercalação de partes de arquivos de uma maneira que não é usada na versão mais simples do ISO 9660, portanto não nos aprofundaremos nela.

O campo a seguir diz em qual CD-ROM o arquivo está localizado. É permitido que uma entrada de diretório em um CD-ROM refira-se a um arquivo localizado em outro CD-ROM no conjunto. Dessa maneira, é possível construir um diretório-mestre no primeiro CD-ROM que liste todos os arquivos que estejam em todos os CD-ROMs no conjunto completo.

O campo marcado *L* na Figura 4.35 mostra o tamanho do nome do arquivo em bytes. Ele é seguido pelo nome do próprio arquivo. Um nome de arquivo consiste em um nome base, um ponto, uma extensão, um ponto e vírgula e um número binário de versão (1 ou 2 bytes). O nome base e a extensão podem usar letras maiúsculas, os dígitos 0-9 e o caractere sublinhado. Todos os outros caracteres são proibidos para certificar-se de que todos os computadores possam lidar com todos os nomes de

FIGURA 4.35 A entrada de diretório do ISO 9660.

arquivos. O nome base pode ter até oito caracteres; a extensão até três caracteres. Essas escolhas foram ditadas pela necessidade de tornar o padrão compatível com o MS-DOS. Um nome de arquivo pode estar presente em um diretório múltiplas vezes, desde que cada um tenha um número de versão diferente.

Os últimos dois campos nem sempre estão presentes. O campo *Preenchimento* é usado para forçar que toda entrada de diretório seja um número par de bytes a fim de alinhar os campos numéricos de entradas subsequentes em limites de 2 bytes. Se o preenchimento for necessário, um byte 0 é usado. Por fim, temos o campo *Uso do sistema*. Sua função e tamanho são indefinidos, exceto que ele deve conter um número par de bytes. Sistemas diferentes o utilizam de maneiras diferentes. O Macintosh, por exemplo, mantém os flags do Finder nele.

Entradas dentro de um diretório são listadas em ordem alfabética, exceto para as duas primeiras entradas. A primeira entrada é para o próprio diretório. A segunda é para o pai. Nesse sentido, elas são similares para as entradas de diretório . e .. do UNIX. Os arquivos em si não precisam estar na ordem do diretório.

Não há um limite explícito para o número de entradas em um diretório. No entanto, há um limite para a profundidade de aninhamento. A profundidade máxima de aninhamento de um diretório é oito. Esse limite foi estabelecido arbitrariamente para simplificar algumas implementações.

O ISO 9660 define o que são chamados de três níveis. O nível 1 é o mais restritivo e especifica que os nomes de arquivos sejam limitados a 8 + 3 caracteres como descrevemos, e também exige que todos os arquivos sejam contíguos como descrevemos. Além disso, ele especifica que os nomes dos diretórios sejam limitados a oito caracteres sem extensões. O uso desse nível maximiza as chances de que um CD-ROM seja lido em todos os computadores.

O nível 2 relaxa a restrição de tamanho. Ele permite que arquivos e diretórios tenham nomes de até 31 caracteres, mas ainda do mesmo conjunto de caracteres.

O nível 3 usa os mesmos limites de nomes do nível 2, mas relaxa parcialmente o pressuposto de que os arquivos tenham de ser contíguos. Com esse nível, um arquivo pode consistir em várias seções (extensões), cada uma como uma sequência contígua de blocos. A mesma sequência pode ocorrer múltiplas vezes em um arquivo e pode ocorrer em dois ou mais arquivos. Se grandes porções de dados são repetidas em vários arquivos, o nível 3 oferecerá alguma otimização de espaço ao não exigir que os dados estejam presentes múltiplas vezes.

Extensões Rock Ridge

Como vimos, o ISO 9660 é altamente restritivo em várias maneiras. Logo depois do seu lançamento, as pessoas na comunidade UNIX começaram a trabalhar em uma extensão a fim de tornar possível representar os sistemas de arquivos UNIX em um CD-ROM. Essas extensões foram chamadas de **Rock Ridge**, em homenagem à cidade no filme de Mel Brooks, *Banzé no Oeste* (*Blazing Saddles*), provavelmente porque um dos membros do comitê gostou do filme.

As extensões usam o campo *Uso do sistema* para possibilitar a leitura dos CD-ROMs Rock Ridge em qualquer computador. Todos os outros campos mantêm seu significado ISO 9660 normal. Qualquer sistema que não conheça as extensões Rock Ridge apenas as ignora e vê um CD-ROM normal.

As extensões são divididas nos campos a seguir:

1. PX — atributos POSIX.
2. PN — Números de dispositivo principal e secundário.
3. SL — Ligação simbólica.
4. NM — Nome alternativo.
5. CL — Localização do filho.
6. PL — Localização do pai.
7. RE — Realocação.
8. TF — Estampas de tempo (Time stamps).

O campo *PX* contém o padrão UNIX para bits de permissão *rwxrwxrwx* para o proprietário, grupo e outros. Ele também contém os outros bits contidos na palavra de modo, como os bits SETUID e SETGID, e assim por diante.

Para permitir que dispositivos sejam representados em um CD-ROM, o campo *PN* está presente. Ele contém os números de dispositivos principais e secundários associados com o arquivo. Dessa maneira, os conteúdos do diretório */dev* podem ser escritos para um CD-ROM e mais tarde reconstruídos corretamente no sistema de destino.

O campo *SL* é para ligações simbólicas. Ele permite que um arquivo em um sistema refira-se a um arquivo em um sistema diferente.

O campo mais importante é *NM*. Ele permite que um segundo nome seja associado com o arquivo. Esse nome não está sujeito às restrições de tamanho e conjunto de caracteres do ISO 9660, tornando possível expressar nomes de arquivos UNIX arbitrários em um CD-ROM.

Os três campos seguintes são usados juntos para contornar o limite de oito diretórios que podem ser aninhados no ISO 9660. Usando-os é possível especificar que um diretório seja realocado, e dizer onde ele vai

na hierarquia. Trata-se efetivamente de uma maneira de contornar o limite de profundidade artificial.

Por fim, o campo *TF* contém as três estampas de tempo incluídas em cada i-node UNIX, a saber o horário que o arquivo foi criado, modificado e acessado pela última vez. Juntas, essas extensões tornam possível copiar um sistema de arquivos UNIX para um CD-ROM e então restaurá-lo corretamente para um sistema diferente.

Extensões Joliet

A comunidade UNIX não foi o único grupo que não gostou do ISO 9660 e queria uma maneira de estendê-lo. A Microsoft também o achou restritivo demais (embora tenha sido o próprio MS-DOS da Microsoft que causou a maior parte das restrições em primeiro lugar). Portanto, a Microsoft inventou algumas extensões chamadas **Joliet**. Elas foram projetadas para permitir que os sistemas de arquivos Windows fossem copiados para um CD-ROM e então restaurados, precisamente da mesma maneira que o Rock Ridge foi projetado para o UNIX. Virtualmente todos os programas que executam sob o Windows e usam CD-ROMs aceitam Joliet, incluindo programas que gravam em CDs regraváveis. Em geral, esses programas oferecem uma escolha entre os vários níveis de ISO 9660 e Joliet.

As principais extensões oferecidas pelo Joliet são:

1. Nomes de arquivos longos.
2. Conjunto de caracteres Unicode.
3. Aninhamento de diretórios mais profundo que oito níveis.
4. Nomes de diretórios com extensões.

A primeira extensão permite nomes de arquivos de até 64 caracteres. A segunda extensão capacita o uso do conjunto de caracteres Unicode para os nomes de arquivos. Essa extensão é importante para que o software possa ser empregado em países que não usam o alfabeto latino, como Japão, Israel e Grécia. Como os caracteres Unicode ocupam 2 bytes, o nome de arquivo máximo em Joliet ocupa 128 bytes.

Assim como no Rock Ridge, a limitação sobre aninhamentos de diretórios foi removida no Joliet. Os diretórios podem ser aninhados o mais profundamente quanto necessário. Por fim, nomes de diretórios podem ter extensões. Não fica claro por que essa extensão foi incluída, já que os diretórios do Windows virtualmente nunca usam extensões, mas talvez um dia venham a usar.

4.6 Pesquisas em sistemas de arquivos

Os sistemas de arquivos sempre atraíram mais pesquisas do que outras partes do sistema operacional e até hoje é assim. Conferências inteiras como FAST, MSST e NAS são devotadas em grande parte a sistemas de arquivos e armazenamento. Embora os sistemas de arquivos-padrão sejam bem compreendidos, ainda há bastante pesquisa sendo feita sobre backups (SMALDONE et al., 2013; e WALLACE et al., 2012), cache (KOLLER et al.; Oh, 2012; e ZHANG et al., 2013a), exclusão de dados com segurança (WEI et al., 2011), compressão de arquivos (HARNIK et al., 2013), sistemas de arquivos para dispositivos flash (NO, 2012; PARK e SHEN, 2012; e NARAYANAN, 2009), desempenho (LEVENTHAL, 2013; e SCHINDLER et al., 2011), RAID (MOON e REDDY, 2013), confiabilidade e recuperação de erros (CHIDAMBARAM et al., 2013; MA et al., 2013; MCKUSICK, 2012; e VAN MOOLENBROEK et al., 2012), sistemas de arquivos em nível do usuário (RAJGARHIA e GEHANI, 2010), verificações de consistência (FRYER et al., 2012) e sistemas de arquivos com controle de versões (MASHTIZADEH et al., 2013). Apenas mensurar o que está realmente acontecendo em um sistema de arquivos também é um tópico de pesquisa (HARTER et al., 2012).

A segurança é um tópico sempre presente (BOTELHO et al., 2013; LI et al., 2013c; e LORCH et al., 2013). Por outro lado, um novo tópico refere-se aos sistemas de arquivos na nuvem (MAZUREK et al., 2012; e VRABLE et al., 2012). Outra área que tem ganhado atenção recentemente é a procedência — o monitoramento da história dos dados, incluindo de onde vêm, quem é o proprietário e como eles foram transformados (GHOSHAL e PLALE, 2013; e SULTANA e BERTINO, 2013). Manter os dados seguros e úteis por décadas também interessa às empresas que têm um compromisso legal de fazê-lo (BAKER et al., 2006). Por fim, outros pesquisadores estão repensando a pilha do sistema de arquivos (APPUSWAMY et al., 2011).

4.7 Resumo

Quando visto de fora, um sistema operacional é uma coleção de arquivos e diretórios, mais as operações sobre eles. Arquivos podem ser lidos e escritos, diretórios criados e destruídos e arquivos podem ser movidos

de diretório para diretório. A maioria dos sistemas de arquivos modernos dá suporte a um sistema de diretórios hierárquico no qual os diretórios podem ter subdiretórios, e estes podem ter "subsubdiretórios" *ad infinitum*.

Quando visto de dentro, um sistema de arquivos parece bastante diferente. Os projetistas do sistema de arquivos precisam estar preocupados com como o armazenamento é alocado e como o sistema monitora qual bloco vai com qual arquivo. As possibilidades incluem arquivos contíguos, listas encadeadas, tabelas de alocação de arquivos e i-nodes. Sistemas diferentes têm estruturas de diretórios diferentes. Os atributos podem ficar nos diretórios ou em algum outro lugar (por exemplo, um i-node). O espaço de disco pode ser gerenciado usando listas de espaços livres ou mapas de bits. A confiabilidade do sistema de arquivos é reforçada a partir da realização de cópias incrementais e um programa que possa reparar sistemas de arquivos danificados. O desempenho do sistema de arquivos é importante e pode ser incrementado de diversas maneiras, incluindo cache de blocos, leitura antecipada e a colocação cuidadosa de um arquivo próximo do outro. Sistemas de arquivos estruturados em diário (log) também melhoram o desempenho fazendo escritas em grandes unidades.

Exemplos de sistemas de arquivos incluem ISO 9660, MS-DOS e UNIX. Eles diferem de muitas maneiras, incluindo pelo modo de monitorar quais blocos vão para quais arquivos, estrutura de diretórios e gerenciamento de espaço livre em disco.

PROBLEMAS

1. Dê cinco nomes de caminhos diferentes para o arquivo */etc./passwd*. (*Dica*: lembre-se das entradas de diretório "." e "..").
2. No Windows, quando um usuário clica duas vezes sobre um arquivo listado pelo Windows Explorer, um programa é executado e dado aquele arquivo como parâmetro. Liste duas maneiras diferentes através das quais o sistema operacional poderia saber qual programa executar.
3. Nos primeiros sistemas UNIX, os arquivos executáveis (arquivos *a.out*) começavam com um número mágico, bem específico, não um número escolhido ao acaso. Esses arquivos começavam com um cabeçalho, seguido por segmentos de texto e dados. Por que você acha que um número bem específico era escolhido para os arquivos executáveis, enquanto os outros tipos de arquivos tinham um número mágico mais ou menos aleatório como primeiro caractere?
4. A chamada de sistema open no UNIX é absolutamente essencial? Quais seriam as consequências de não a ter?
5. Sistemas que dão suporte a arquivos sequenciais sempre têm uma operação para voltar arquivos para trás (rewind). Os sistemas que dão suporte a arquivos de acesso aleatório precisam disso, também?
6. Alguns sistemas operacionais fornecem uma chamada de sistema rename para dar um nome novo para um arquivo. Existe alguma diferença entre usar essa chamada para renomear um arquivo e apenas copiar esse arquivo para um novo com o nome novo, seguido pela remoção do antigo?
7. Em alguns sistemas é possível mapear parte de um arquivo na memória. Quais restrições esses sistemas precisam impor? Como é implementado esse mapeamento parcial?
8. Um sistema operacional simples dá suporte a apenas um único diretório, mas permite que ele tenha nomes arbitrariamente longos de arquivos. Seria possível simular algo próximo de um sistema de arquivos hierárquico? Como?
9. No UNIX e no Windows, o acesso aleatório é realizado por uma chamada de sistema especial que move o ponteiro "posição atual" associado com um arquivo para um determinado byte nele. Proponha uma forma alternativa para o acesso aleatório sem ter essa chamada de sistema.
10. Considere a árvore de diretório da Figura 4.8. Se */usr/jim* é o diretório de trabalho, qual é o nome de caminho absoluto para o arquivo cujo nome de caminho relativo é *../ast/x*?
11. A alocação contígua de arquivos leva à fragmentação de disco, como mencionado no texto, pois algum espaço no último bloco de disco será desperdiçado em arquivos cujo tamanho não é um número inteiro de blocos. Estamos falando de uma fragmentação interna, ou externa? Faça uma analogia com algo discutido no capítulo anterior.
12. Descreva os efeitos de um bloco de dados corrompido para um determinado arquivo: (a) contíguo, (b) encadeado e (c) indexado (ou baseado em tabela).
13. Uma maneira de usar a alocação contígua do disco e não sofrer com espaços livres é compactar o disco toda vez que um arquivo for removido. Já que todos os arquivos são contíguos, copiar um arquivo exige uma busca e atraso rotacional para lê-lo, seguido pela transferência em velocidade máxima. Escrever um arquivo de volta exige o mesmo trabalho. Presumindo um tempo de busca de 5 ms, um atraso rotacional de 4 ms, uma taxa de

transferência de 80 MB/s e o tamanho de arquivo médio de 8 KB, quanto tempo leva para ler um arquivo para a memória principal e então escrevê-lo de volta para o disco na nova localização? Usando esses números, quanto tempo levaria para compactar metade de um disco de 16 GB?

14. Levando em conta a resposta da pergunta anterior, a compactação do disco faz algum sentido?

15. Alguns dispositivos de consumo digitais precisam armazenar dados, por exemplo, como arquivos. Cite um dispositivo moderno que exija o armazenamento de arquivos e para o qual a alocação contígua seria uma boa ideia.

16. Considere o i-node mostrado na Figura 4.13. Se ele contém 10 endereços diretos e esses tinham 8 bytes cada e todos os blocos do disco eram de 1024 KB, qual seria o tamanho do maior arquivo possível?

17. Para uma determinada turma, os históricos dos estudantes são armazenados em um arquivo. Os registros são acessados aleatoriamente e atualizados. Presuma que o histórico de cada estudante seja de um tamanho fixo. Qual dos três esquemas de alocação (contíguo, encadeado e indexado por tabela) será o mais apropriado?

18. Considere um arquivo cujo tamanho varia entre 4 KB e 4 MB durante seu tempo de vida. Qual dos três esquemas de alocação (contíguo, encadeado e indexado por tabela) será o mais apropriado?

19. Foi sugerido que a eficiência poderia ser incrementada e o espaço de disco poupado armazenando os dados de um arquivo curto dentro do i-node. Para o i-node da Figura 4.13, quantos bytes de dados poderiam ser armazenados dentro dele?

20. Duas estudantes de computação, Carolyn e Elinor, estão tendo uma discussão sobre i-nodes. Carolyn sustenta que as memórias ficaram tão grandes e baratas que, quando um arquivo é aberto, é mais simples e mais rápido buscar uma cópia nova do i-node na tabela de i-nodes, em vez de procurar na tabela inteira para ver se ela já está ali. Elinor discorda. Quem está certa?

21. Nomeie uma vantagem de ligações estritas sobre ligações simbólicas e uma vantagem de ligações simbólicas sobre ligações estritas.

22. Explique como as ligações estritas e as ligações flexíveis diferem em relação às alocações de i-nodes.

23. Considere um disco de 4 TB que usa blocos de 4 KB e o método da lista de livres. Quantos endereços de blocos podem ser armazenados em um bloco?

24. O espaço de disco livre pode ser monitorado usando-se uma lista de livres e um mapa de bips. Endereços de disco exigem D bits. Para um disco com B blocos, F dos quais estão disponíveis, estabeleça a condição na qual a lista de livres usa menos espaço do que o mapa de bits. Para D tendo um valor de 16 bits, expresse a resposta como uma percentagem do espaço de disco que precisa estar livre.

25. O começo de um mapa de bits de espaço livre fica assim após a partição de disco ter sido formatada pela primeira vez: 1000 0000 0000 0000 (o primeiro bloco é usado pelo diretório-raiz). O sistema sempre busca por blocos livres começando no bloco de número mais baixo, então após escrever o arquivo *A*, que usa seis blocos, o mapa de bits fica assim: 1111 1110 0000 0000. Mostre o mapa de bits após cada uma das ações a seguir:

(a) O arquivo *B* é escrito usando cinco blocos.

(b) O arquivo *A* é removido.

(c) O arquivo *C* é escrito usando oito blocos.

(d) O arquivo *B* é removido.

26. O que aconteceria se o mapa de bits ou a lista de livres contendo as informações sobre blocos de disco livres fossem perdidos por uma queda no computador? Existe alguma maneira de recuperar-se desse desastre ou é "adeus, disco"? Discuta suas respostas para os sistemas de arquivos UNIX e FAT-16 separadamente.

27. O trabalho noturno de Oliver Owl no centro de computadores da universidade é mudar as fitas usadas para os backups de dados durante a noite. Enquanto espera que cada fita termine, ele trabalha em sua tese que prova que as peças de Shakespeare foram escritas por visitantes extraterrestres. Seu processador de texto executa no sistema sendo copiado, pois esse é o único que eles têm. Há algum problema com esse arranjo?

28. Discutimos como realizar cópias incrementais detalhadamente no texto. No Windows é fácil dizer quando copiar um arquivo, pois todo arquivo tem um bit de arquivamento. Esse bit não existe no UNIX. Como os programas de backup do UNIX sabem quais arquivos copiar?

29. Suponha que o arquivo 21 na Figura 4.25 não foi modificado desde a última cópia. De qual maneira os quatro mapas de bits da Figura 4.26 seriam diferentes?

30. Foi sugerido que a primeira parte de cada arquivo UNIX fosse mantida no mesmo bloco de disco que o seu i-node. Qual a vantagem que isso traria?

31. Considere a Figura 4.27. Seria possível que, para algum número de bloco em particular, os contadores em *ambas* as listas tivessem o valor 2? Como esse problema poderia ser corrigido?

32. O desempenho de um sistema de arquivos depende da taxa de acertos da cache (fração de blocos encontrados na cache). Se for necessário 1 ms para satisfazer uma solicitação da cache, mas 40 ms para satisfazer uma solicitação se uma leitura de disco for necessária, dê uma fórmula para o tempo médio necessário para satisfazer uma solicitação se a taxa de acerto é h. Represente graficamente essa função para os valores de h variando de 0 a 1,0.

33. Para um disco rígido USB externo ligado a um computador, o que é mais adequado: uma cache de escrita direta ou uma cache de bloco?

34. Considere uma aplicação em que os históricos dos estudantes são armazenados em um arquivo. A aplicação pega a identidade de um estudante como entrada e subsequentemente lê, atualiza e escreve o histórico correspondente; isso é repetido até a aplicação desistir. A técnica de "leitura antecipada de bloco" seria útil aqui?

35. Considere um disco que tem 10 blocos de dados começando do bloco 14 até o 23. Deixe 2 arquivos no disco: f1 e f2. A estrutura do diretório lista que os primeiros blocos de dados de f1 e f2 são respectivamente 22 e 16. Levando-se em consideração as entradas de tabela FAT a seguir, quais são os blocos de dados designados para f1 e f2?

(14,18); (15,17); (16,23); (17,21); (18,20); (19,15); (20, −1); (21, −1); (22,19); (23,14).

Nessa notação, (*x, y*) indicam que o valor armazenado na entrada de tabela *x* aponta para o bloco de dados *y*.

36. Considere a ideia por trás da Figura 4.21, mas agora para um disco com um tempo de busca médio de 6 ms, uma taxa rotacional de 15.000 rpm e 1.048.576 bytes por trilha. Quais são as taxas de dados para os tamanhos de blocos de 1 KB, 2 KB e 4 KB, respectivamente?

37. Um determinado sistema de arquivos usa blocos de disco de 4 KB. O tamanho de arquivo médio é 1 KB. Se todos os arquivos fossem exatamente de 1 KB, qual fração do espaço do disco seria desperdiçada? Você acredita que o desperdício para um sistema de arquivos real será mais alto do que esse número ou mais baixo do que ele? Explique sua resposta.

38. Levando-se em conta um tamanho de bloco de 4 KB e um valor de endereço de ponteiro de disco de 4 bytes, qual é o maior tamanho de arquivo (em bytes) que pode ser acessado usando 10 endereços diretos e um bloco indireto?

39. Arquivos no MS-DOS têm de competir por espaço na tabela FAT-16 na memória. Se um arquivo usa *k* entradas, isto é, *k* entradas que não estão disponíveis para qualquer outro arquivo, qual restrição isso ocasiona sobre o tamanho total de todos os arquivos combinados?

40. Um sistema de arquivos UNIX tem blocos de 4 KB e endereços de disco de 4 bytes. Qual é o tamanho de arquivo máximo se os i-nodes contêm 10 entradas diretas, e uma entrada indireta única, dupla e tripla cada?

41. Quantas operações de disco são necessárias para buscar o i-node para um arquivo com o nome de caminho */usr/ast/courses/os/handout.t*? Presuma que o i-node para o diretório-raiz está na memória, mas nada mais ao longo do caminho está na memória. Também presuma que todo diretório caiba em um bloco de disco.

42. Em muitos sistemas UNIX, os i-nodes são mantidos no início do disco. Um projeto alternativo é alocar um i-node quando um arquivo é criado e colocar o i-node no começo do primeiro bloco do arquivo. Discuta os prós e contras dessa alternativa.

43. Escreva um programa que inverta os bytes de um arquivo, para que o último byte seja agora o primeiro e o primeiro, o último. Ele deve funcionar com um arquivo arbitrariamente longo, mas tente torná-lo razoavelmente eficiente.

44. Escreva um programa que comece em um determinado diretório e percorra a árvore de arquivos a partir daquele ponto registrando os tamanhos de todos os arquivos que encontrar. Quando houver concluído, ele deve imprimir um histograma dos tamanhos dos arquivos usando uma largura de célula especificada como parâmetro (por exemplo, com 1024, tamanhos de arquivos de 0 a 1023 são colocados em uma célula, 1024 a 2047 na seguinte etc.).

45. Escreva um programa que escaneie todos os diretórios em um sistema de arquivos UNIX e encontre e localize todos os i-nodes com uma contagem de ligações estritas de duas ou mais. Para cada arquivo desses, ele lista juntos todos os nomes que apontam para o arquivo.

46. Escreva uma nova versão do programa *ls* do UNIX. Essa versão recebe como argumentos um ou mais nomes de diretórios e para cada diretório lista todos os arquivos nele, uma linha por arquivo. Cada campo deve ser formatado de uma maneira razoável considerando o seu tipo. Liste apenas o primeiro endereço de disco, se houver.

47. Implemente um programa para mensurar o impacto de tamanhos de buffer no nível de aplicação nos tempos de leitura. Isso consiste em ler para e escrever a partir de um grande arquivo (digamos, 2 GB). Varie o tamanho do buffer de aplicação (digamos, de 64 bytes para 4 KB). Use rotinas de mensuração de tempo (como *gettimeofday* e *getitimer* no UNIX) para mensurar o tempo levado por diferentes tamanhos de buffers. Analise os resultados e relate seus achados: o tamanho do buffer faz uma diferença para o tempo de escrita total e tempo por escrita?

48. Implemente um sistema de arquivos simulado que será completamente contido em um único arquivo regular armazenado no disco. Esse arquivo de disco conterá diretórios, i-nodes, informações de blocos livres, blocos de dados de arquivos etc. Escolha algoritmos adequados para manter informações sobre blocos livres e para alocar blocos de dados (contíguos, indexados, encadeados). Seu programa aceitará comandos de sistema do usuário para criar/remover diretórios, criar/remover/abrir arquivos, ler/escrever de/para um arquivo selecionado e listar conteúdos de diretórios.