

Capítulo 24

Sistemas de arquivos

24.1 Introdução

Vários problemas importantes devem ser resolvidos para a implementação eficiente de arquivos e diretórios, que vão dos aspectos de baixo nível, como o acesso aos dispositivos físicos, a aspectos mais abstratos, como a implementação da interface de acesso a arquivos para os programadores.

Neste capítulo são discutidos os principais elementos dos sistemas de arquivos, que são os subsistemas do sistema operacional que implementam o conceito de arquivo sobre o armazenamento bruto proporcionado pelos dispositivos físicos.

24.2 Arquitetura geral

Os principais elementos que realizam a implementação de arquivos no sistema operacional estão organizados em camadas, sendo apresentados na Figura 24.1 e detalhados a seguir:

Dispositivos: como discos rígidos ou bancos de memória *flash*, são os responsáveis pelo armazenamento de dados.

Controladores: são os circuitos eletrônicos dedicados ao controle dos dispositivos físicos. Eles são acessados através de portas de entrada/saída, interrupções e canais de acesso direto à memória (DMA).

Drivers: interagem com os controladores de dispositivos para configurá-los e realizar as transferências de dados entre o sistema operacional e os dispositivos. Como cada controlador define sua própria interface, também possui um *driver* específico. Os *drivers* ocultam as diferenças entre controladores e fornecem às camadas superiores do núcleo uma interface padronizada para acesso aos dispositivos de armazenamento.

Gerência de blocos: gerencia o fluxo de blocos de dados entre as camadas superiores e os dispositivos de armazenamento. Como os discos são dispositivos orientados a blocos, as operações de leitura e escrita de dados são sempre feitas com blocos de dados, e nunca com bytes individuais. As funções mais importantes desta camada são efetuar o mapeamento de blocos lógicos nos blocos físicos do dispositivo (Seção 24.4.1) e o *caching/buffering* de blocos (Seção 24.4.2).

Alocação de arquivos: realiza a alocação dos arquivos sobre os blocos lógicos oferecidos pela camada de gerência de blocos. Cada arquivo é visto como uma sequência de blocos lógicos que deve ser armazenada nos blocos dos dispositivos de forma eficiente, robusta e flexível. As principais técnicas de alocação de arquivos são discutidas na Seção 24.5.

Sistema de arquivos virtual: o VFS (*Virtual File System*) constrói as abstrações de diretórios e atalhos, além de gerenciar as permissões associadas aos arquivos e as travas de acesso compartilhado. Outra responsabilidade importante desta camada é manter o registro de cada arquivo aberto pelos processos, como a posição da última operação no arquivo, o modo de abertura usado e o número de processos que estão usando o arquivo.

Interface do sistema de arquivos: conjunto de chamadas de sistema oferecidas aos processos do espaço de usuários para a criação e manipulação de arquivos.

Bibliotecas de entrada/saída: usam as chamadas de sistema da interface do núcleo para construir funções padronizadas de acesso a arquivos para cada linguagem de programação (como aquelas apresentadas na Seção 23.6 para a linguagem C ANSI).

Na Figura 24.1, a maior parte da implementação do sistema de arquivos está localizada dentro do núcleo, mas isso obviamente varia de acordo com a arquitetura do sistema operacional. Em sistemas micronúcleo (Seção 3.2), por exemplo, as camadas de gerência de blocos e as superiores provavelmente estariam no espaço de usuário.

Na implementação de um sistema de arquivos, considera-se que cada arquivo possui **dados** e **metadados**. Os dados de um arquivo são o seu conteúdo em si (uma música, uma fotografia, um documento ou uma planilha); por outro lado, os metadados do arquivo são seus atributos (nome, datas, permissões de acesso, etc.) e todas as informações de controle necessárias para localizar e manter seu conteúdo no disco. Também são considerados metadados as informações necessárias à gestão do sistema de arquivos, como os mapas de blocos livres, etc.

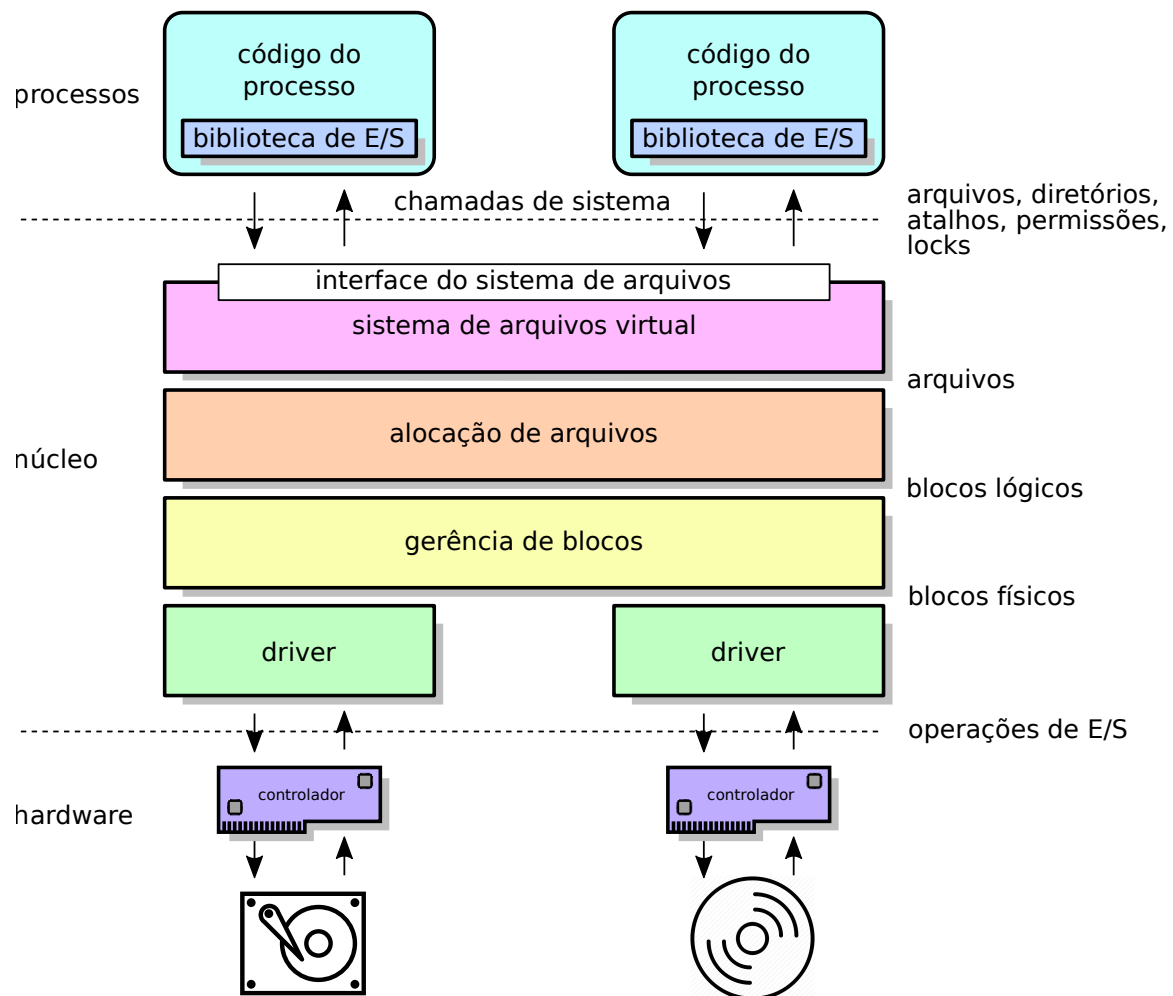


Figura 24.1: Camadas da implementação da gerência de arquivos.

24.3 Espaços de armazenamento

Um computador normalmente possui um ou mais dispositivos para armazenar arquivos, que podem ser discos rígidos, discos óticos (CD-ROM, DVD-ROM), discos de estado sólido (baseados em memória *flash*, como *pendrives* USB), etc. A estrutura física dos discos rígidos e demais dispositivos é discutida em detalhes no Capítulo 20.

24.3.1 Discos e partições

Em linhas gerais, um disco é visto pelo sistema operacional como um grande vetor de blocos de dados de tamanho fixo, numerados sequencialmente. As operações de leitura e escrita de dados nesses dispositivos são feitas bloco a bloco, por essa razão esses dispositivos são chamados *dispositivos de blocos* (*block devices* ou *block-oriented devices*).

O espaço de armazenamento de cada dispositivo é dividido em uma pequena área de configuração reservada, no início do disco, e uma ou mais *partições*, que podem ser vistas como espaços independentes. A área de configuração contém uma *tabela de partições* com informações sobre o particionamento do dispositivo (número do bloco inicial, quantidade de blocos e outras informações sobre cada partição). Além disso, essa

área contém também um pequeno código executável usado no processo de inicialização do sistema operacional (*boot*), por isso ela é usualmente chamada de *boot sector* ou MBR (*Master Boot Record*, nos PCs).

No início de cada partição do disco há também um ou mais blocos reservados, usados para a descrição do conteúdo daquela partição e para armazenar o código de lançamento do sistema operacional, se aquela for uma partição inicializável (*bootable partition*). Essa área reservada é denominada *bloco de inicialização* ou VBR - *Volume Boot Record*. O restante dos blocos da partição está disponível para o armazenamento de arquivos. A Figura 24.2 ilustra a organização básica do espaço de armazenamento em um disco rígido típico, com três partições.

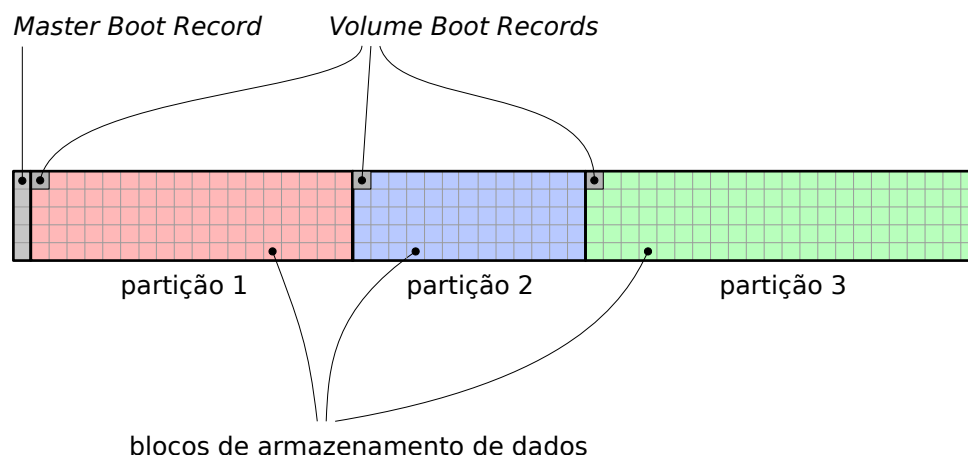


Figura 24.2: Organização em partições de um disco rígido.

É importante lembrar que existem diversos formatos possíveis para os blocos de inicialização de disco e de partição, além da estrutura da própria tabela de partição. Esses formatos devem ser reconhecidos pelo código de inicialização do computador (BIOS) e pelo sistema operacional instalado, para que os dados do disco possam ser acessados.

Um termo frequentemente utilizado em sistemas de arquivos é o **volume**, que significa um espaço de armazenamento de dados, do ponto de vista do sistema operacional. Em sua forma mais simples, cada volume corresponde a uma partição, mas configurações mais complexas são possíveis. Por exemplo, o subsistema LVM (*Logical Volume Manager*) do Linux permite construir volumes lógicos combinando vários discos físicos, como nos sistemas RAID (Seção 21.2).

Antes de ser usado, cada volume ou partição deve ser *formatado*, ou seja, preenchido com as estruturas de dados necessárias para armazenar arquivos, diretórios, atalhos e outras entradas. Cada volume pode ser formatado de forma independente e receber um sistema de arquivos distinto dos demais volumes.

24.3.2 Montagem de volumes

Para que o sistema operacional possa acessar os arquivos armazenados em um volume, ele deve ler os dados presentes em seu bloco de inicialização, que descrevem o tipo de sistema de arquivos do volume, e criar as estruturas em memória que representam esse volume dentro do núcleo do SO. Além disso, ele deve definir um identificador para o volume, de forma que os processos possam acessar seus arquivos. Esse procedimento

é denominado *montagem* do volume, e seu nome vem do tempo em que era necessário montar fisicamente os discos rígidos ou fitas magnéticas nos leitores, antes de poder acessar os dados. O procedimento oposto, a *desmontagem*, consiste em fechar todos os arquivos abertos no volume e remover as estruturas de memória usadas para gerenciá-lo.

Apesar de ser realizada para todos os volumes, a montagem é um procedimento particularmente frequente no caso de mídias removíveis, como CD-ROMs, DVD-ROMs e *pendrives* USB. Neste caso, a desmontagem do volume inclui também ejetar a mídia (CD, DVD) ou avisar o usuário que ela pode ser removida (discos USB).

Ao montar um volume, deve-se fornecer aos processos e usuários uma referência para seu acesso, denominada *ponto de montagem* (*mounting point*). Sistemas UNIX normalmente definem os pontos de montagem de volumes como posições dentro da árvore principal do sistema de arquivos. Dessa forma, há um volume principal, montado durante a inicialização do sistema operacional, onde normalmente reside o próprio sistema operacional e que define a estrutura básica da árvore de diretórios. Os volumes secundários são montados como subdiretórios na árvore do volume principal, através do comando `mount`.

A Figura 24.3 apresenta um exemplo de montagem de volumes em plataformas UNIX. Nessa figura, o disco SSD contém o sistema operacional e foi montado como raiz da árvore de diretórios, durante a inicialização do sistema. O disco rígido contém os diretórios de usuários e seu ponto de montagem é o diretório `/home`. Já o diretório `/media/cdrom` é o ponto de montagem de um CD-ROM, com sua árvore de diretórios própria, e o diretório `/media/backup` é o ponto de montagem de um *pendrive*.

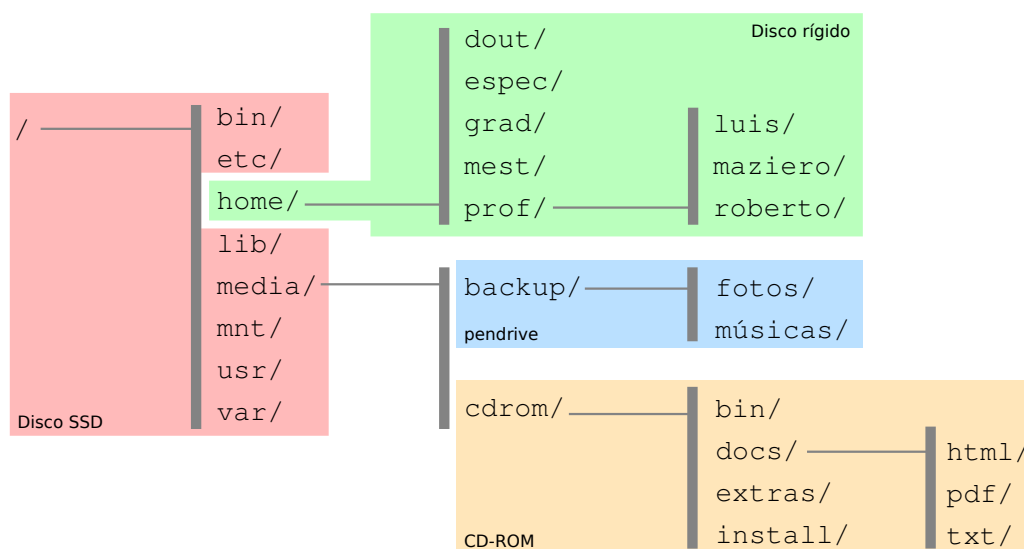


Figura 24.3: Montagem de volumes em UNIX.

Em sistemas de arquivos de outras plataformas, como DOS e Windows, é comum definir cada volume montado como um disco lógico distinto, chamado simplesmente de disco ou *drive* e identificado por uma letra (“A:”, “C:”, “D:”, etc.). Todavia, o sistema de arquivos NTFS do Windows também permite a montagem de volumes como subdiretórios da árvore principal, da mesma forma que o UNIX.

24.4 Gestão de blocos

A função primordial da camada de gestão de blocos é interagir com os *drivers* de dispositivos para realizar as operações de leitura e escrita de blocos de dados. Neste nível do subsistema ainda não existe a noção de arquivo, que só será implementada nas camadas superiores. Portanto, todas as operações nesta camada dizem respeito a blocos de dados.

Além da interação com os *drivers*, esta camada também é responsável pelo mapeamento entre blocos físicos e blocos lógicos e pelo mecanismo de *caching* de blocos, abordados a seguir.

24.4.1 Blocos físicos e lógicos

Conforme visto na Seção 21.1.1, um disco rígido pode ser visto como um conjunto de blocos de tamanho fixo (geralmente de 512 ou 4.096 bytes). Os blocos do disco rígido são normalmente denominados *blocos físicos*. Como esses blocos são pequenos, seu número em um disco rígido recente pode ser imenso: um disco rígido de 500 GBytes contém mais de um bilhão de blocos físicos!

Para simplificar a gerência da imensa quantidade de blocos físicos e melhorar o desempenho das operações de leitura/escrita, os sistemas operacionais costumam agrupar os blocos físicos em *blocos lógicos* ou *clusters*, que são grupos de 2^n blocos físicos consecutivos. A maior parte das operações e estruturas de dados definidas nos discos pelos sistemas operacionais são baseadas em blocos lógicos, que também definem a unidade mínima de alocação de arquivos e diretórios: cada arquivo ou diretório ocupa um ou mais blocos lógicos para seu armazenamento.

O número de blocos físicos em cada bloco lógico é fixo e definido pelo sistema operacional ao formatar a partição, em função de vários parâmetros, como o tamanho da partição, o sistema de arquivos usado e o tamanho das páginas de memória RAM. Blocos lógicos com tamanhos de 4 KB a 64 KBytes são frequentemente usados.

Blocos lógicos maiores (32 KB ou 64 KB) levam a uma menor quantidade de blocos lógicos a gerenciar pelo SO em cada disco e implicam em mais eficiência de entrada/saída, pois mais dados são transferidos em cada operação. Entretanto, blocos grandes podem gerar muita *fragmentação interna*. Por exemplo, um arquivo com 200 bytes de dados ocupará um bloco lógico inteiro. Se esse bloco lógico tiver 32 KBytes (32.768 bytes), serão desperdiçados 32.568 bytes, que ficarão alocados ao arquivo sem ser usados.

Pode-se concluir que blocos lógicos menores diminuiriam a perda de espaço útil por fragmentação interna. Todavia, usar blocos menores implica em ter mais blocos a gerenciar por disco e menos bytes transferidos em cada operação de leitura/escrita, o que tem impacto negativo sobre o desempenho do sistema.

A fragmentação interna diminui o espaço útil do disco, por isso deve ser evitada. Uma forma de tratar esse problema é escolher um tamanho de bloco lógico adequado ao tamanho médio dos arquivos a armazenar no disco, no momento de sua formatação. Alguns sistemas de arquivos (como o UFS do Solaris e o ReiserFS do Linux) permitem a alocação de partes de blocos lógicos, através de técnicas denominadas *fragmentos de blocos* ou *alocação de sub-blocos* [Vahalia, 1996].

24.4.2 *Caching* de blocos

Discos são dispositivos lentos, portanto as operações de leitura e escrita de blocos podem ter latências elevadas. O desempenho nos acessos ao disco pode ser melhorado através de um *cache*, ou seja, uma área de memória RAM na camada de gerência de blocos, onde o conteúdo dos blocos lidos/escritos pode ser mantido para acessos posteriores.

É possível fazer *caching* de leitura e de escrita. No *caching* de leitura (*read caching*), blocos lidos anteriormente do disco são mantidos em memória, para acelerar leituras subsequentes desses mesmos blocos. No *caching* de escrita (*write caching*, também chamado *buffering*), blocos a escrever no disco são mantidos em memória, para agrupar várias escritas pequenas em poucas escritas maiores (e mais eficientes) e para agilizar leituras posteriores desses dados.

Quatro estratégias básicas de *caching* são usuais (ilustradas na Figura 24.4):

Read-through: quando um processo solicita uma leitura, o cache é consultado; caso o dado esteja no cache ele é entregue ao processo; caso contrário, o bloco é lido do disco, copiado no cache e entregue ao processo. Leituras subsequentes do mesmo bloco (pelo mesmo ou outro processo) poderão acessar o conteúdo diretamente do cache, sem precisar acessar o disco.

Read-ahead: ao atender uma requisição de leitura, são trazidos para o cache mais dados que os solicitados pela requisição; além disso, leituras de dados ainda não solicitados podem ser agendadas em momentos de ociosidade dos discos. Dessa forma, futuras requisições podem ser beneficiadas pela leitura antecipada dos dados. Essa política pode melhorar muito o desempenho de acesso sequencial a arquivos e em outras situações com boa localidade de referências (Seção 14.9).

Write-through: quando um processo solicita uma escrita, o conteúdo é escrito diretamente no disco, enquanto o processo solicitante aguarda a conclusão da operação. Uma cópia do conteúdo é mantida no cache, para beneficiar leituras futuras. Não há ganho de desempenho na operação de escrita. Esta estratégia também é denominada *escrita síncrona*.

Write-back: (ou *write-behind*) quando um processo solicita uma escrita, os dados são copiados para o cache e o processo solicitante é liberado. A escrita efetiva dos dados no disco é efetuada posteriormente. Esta estratégia melhora o desempenho de escrita, pois libera mais cedo os processos que solicitam escritas (eles não precisam esperar pela escrita real dos dados no disco) e permite agrupar as operações de escrita, gerando menos acessos a disco. Todavia, existe o risco de perda de dados, caso ocorra um erro no sistema ou falta de energia antes que os dados sejam efetivamente transferidos do cache para o disco.

A estratégia de *caching* utilizada em cada operação pode ser definida pelas camadas superiores (alocação de arquivos, etc) e depende do tipo de informação a ser lida ou escrita. Usualmente, dados de arquivos são escritos usando a estratégia *write-back*, para obter um bom desempenho, enquanto metadados (estruturas de diretórios e outras informações de controle do sistema de arquivos) são escritas usando *write-through* para garantir mais confiabilidade (a perda de metadados tem muito mais impacto na confiabilidade do sistema que a perda de dados dentro de um arquivo específico).

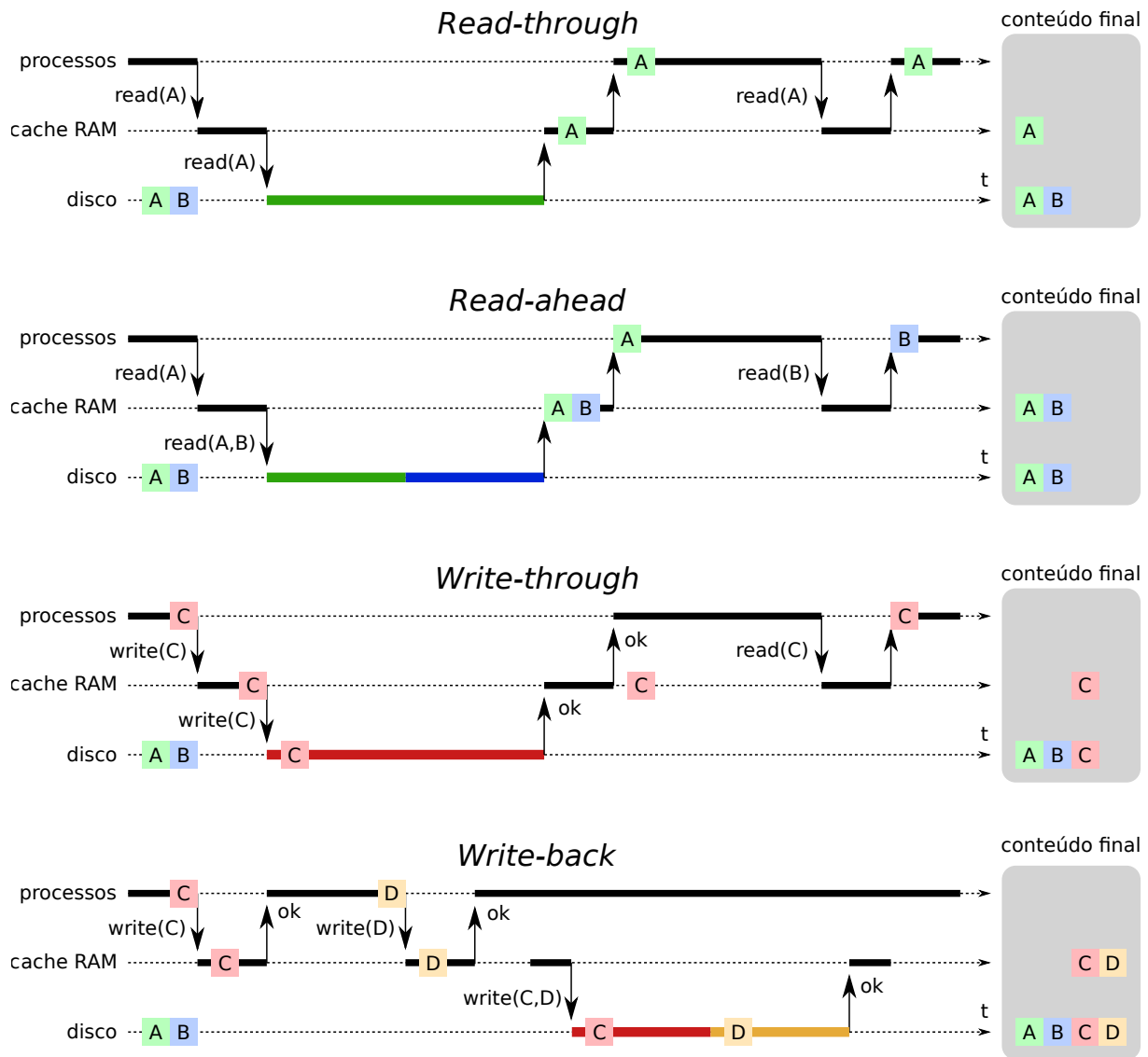


Figura 24.4: Estratégias de *caching* de blocos (A ... D indicam blocos de disco).

Outro aspecto importante do *cache* é a gestão de seu tamanho e conteúdo. Obviamente, o *cache* de blocos reside em RAM e é menor que os discos subjacentes, portanto ele pode ficar cheio. Nesse caso, é necessário definir uma política para selecionar que conteúdo pode ser removido do *cache*. Políticas de substituição clássicas, como FIFO, LRU (*Least Recently Used*) e de segunda chance são frequentemente utilizadas.

24.5 Alocação de arquivos

Como visto nas seções anteriores, um espaço de armazenamento é visto pelas camadas superiores do sistema operacional como um grande vetor de blocos lógicos de tamanho fixo. O problema da alocação de arquivos consiste em dispor (alocar) o conteúdo e os metadados dos arquivos dentro desses blocos (Figura 24.5). Como os blocos são pequenos, um arquivo pode precisar de muitos blocos para ser armazenado no disco: por exemplo, um arquivo de filme em formato MP4 com 1 GB de tamanho ocuparia 262.144 blocos de 4 KBytes. O conteúdo do arquivo deve estar disposto nesses

blocos de forma a permitir um acesso rápido, flexível e confiável. Por isso, a forma de alocação dos arquivos nos blocos do disco tem um impacto importante sobre o desempenho e a robustez do sistema de arquivos.

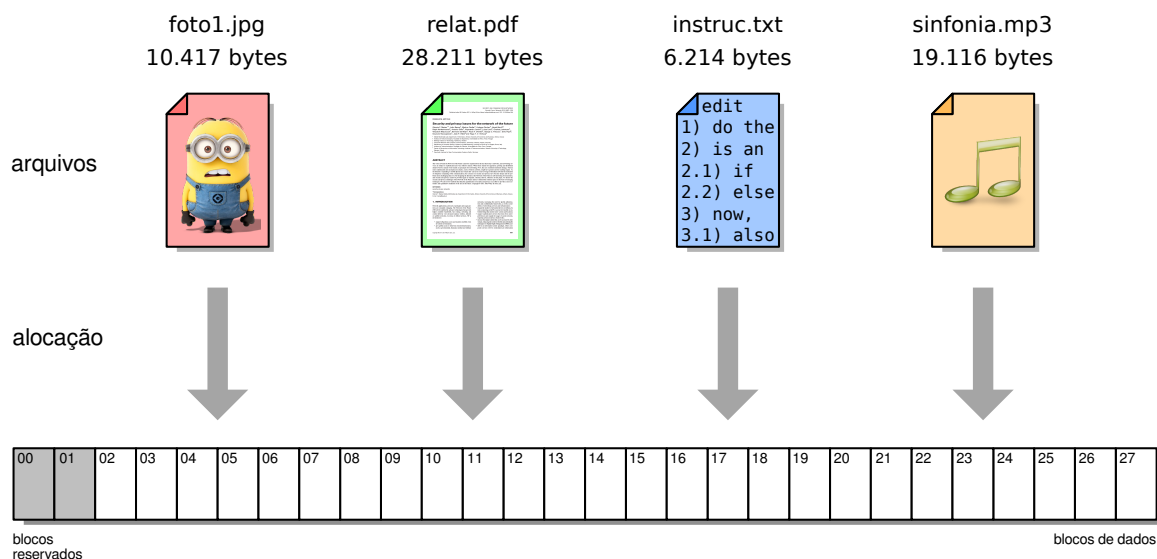


Figura 24.5: O problema da alocação de arquivos.

Há três estratégias básicas de alocação de arquivos nos blocos lógicos do disco, que serão apresentadas a seguir: as alocações **contígua**, **encadeada** e **indexada**. Essas estratégias serão descritas e analisadas à luz de três critérios: a **rapidez** oferecida por cada estratégia no acesso aos dados do arquivo, tanto para acessos sequenciais quanto para acessos aleatórios; a **robustez** de cada estratégia frente a erros, como blocos de disco defeituosos (*bad blocks*) e dados corrompidos; e a **flexibilidade** oferecida por cada estratégia para a criação, modificação e exclusão de arquivos.

Um conceito importante na alocação de arquivos é o **bloco de controle de arquivo** (FCB - *File Control Block*), que nada mais é que uma estrutura contendo os metadados do arquivo e uma referência para a localização de seu conteúdo no disco. A implementação do FCB depende do sistema de arquivos: em alguns pode ser uma simples entrada na tabela de diretório, enquanto em outros é uma estrutura de dados separada, como a *Master File Table* do sistema NTFS e os *i-nodes* dos sistemas UNIX.

24.5.1 Alocação contígua

Na alocação contígua, os dados do arquivo são dispostos de forma sequencial sobre um conjunto de blocos consecutivos no disco, sem “buracos” entre os blocos. Assim, a localização do conteúdo do arquivo no disco é definida pelo endereço de seu primeiro bloco. A Figura 24.6 apresenta um exemplo dessa estratégia de alocação (para simplificar o exemplo, considera-se que a tabela de diretórios contém os metadados de cada arquivo, como nome, tamanho em bytes e número do bloco inicial).

Como os blocos de cada arquivo se encontram em sequência no disco, o acesso sequencial aos dados do arquivo é rápido, por exigir pouca movimentação da cabeça de leitura do disco. O acesso aleatório também é rápido, pois a posição de cada byte do arquivo pode ser facilmente calculada a partir da posição do bloco inicial, conforme indica o algoritmo 3.

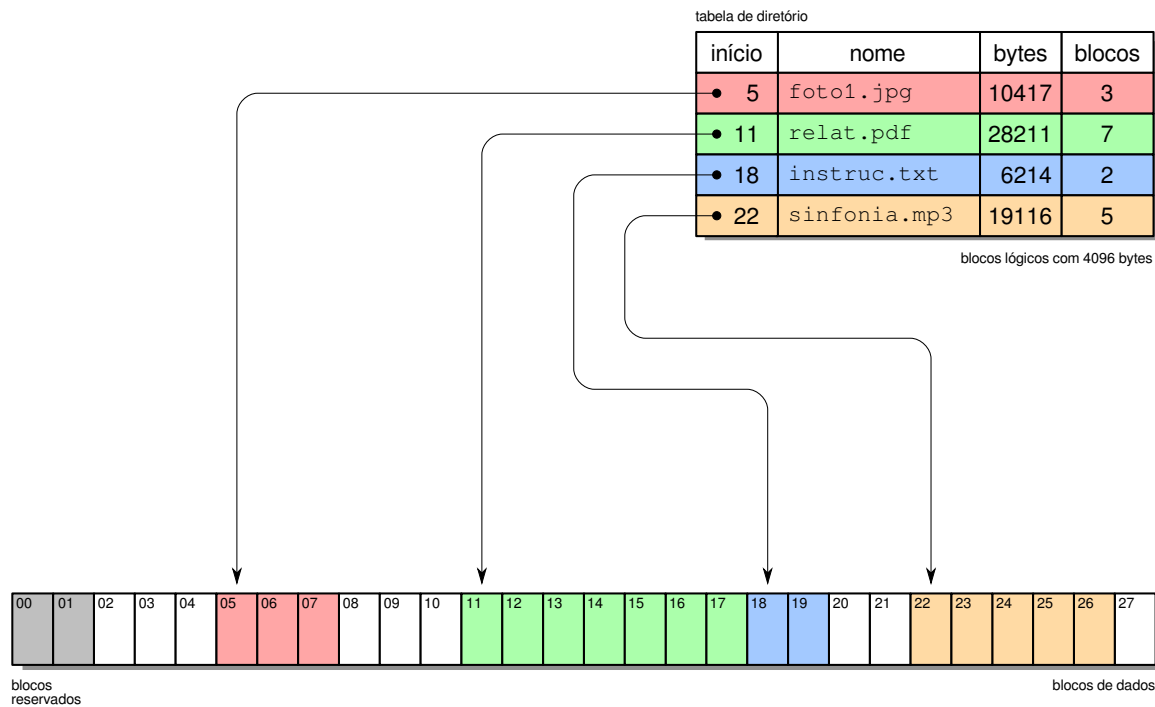


Figura 24.6: Estratégia de alocação contígua.

De acordo com esse algoritmo, o byte de número 14.372 do arquivo `relat.pdf` da Figura 24.6 estará na posição 2.084 do bloco 14 do disco rígido:

$$b_i = b_0 + i \div B = 11 + 14.372 \div 4.096 = 11 + 3 = 14$$

$$o_i = i \bmod B = 14.372 \bmod 4.096 = 2.084$$

A estratégia de alocação contígua apresenta uma boa robustez a falhas de disco: caso um bloco do disco apresente defeito e não permita a leitura dos dados contidos nele, apenas o conteúdo daquele bloco é perdido: o conteúdo do arquivo nos blocos anteriores e posteriores ao bloco defeituoso ainda poderão ser acessados normalmente. Além disso, seu desempenho é muito bom, pois os blocos de cada arquivo estão próximos entre si no disco, minimizando a movimentação da cabeça de leitura do disco.

O ponto fraco desta estratégia é sua baixa flexibilidade, pois o tamanho máximo de cada arquivo precisa ser conhecido no momento de sua criação. No exemplo da Figura 24.6, o arquivo `relat.pdf` não pode aumentar de tamanho, pois não há blocos livres imediatamente após ele (o bloco 18 está ocupado). Para poder aumentar o tamanho desse arquivo, ele teria de ser movido (ou o arquivo `instruc.txt`) para liberar os blocos necessários.

Outro problema desta estratégia é a fragmentação externa, de forma similar à que ocorre nos mecanismos de alocação de memória (Capítulo 16): à medida em que arquivos são criados e destruídos, as áreas livres do disco vão sendo divididas em pequenas áreas isoladas (os fragmentos) que diminuem a capacidade de alocação de arquivos maiores. Por exemplo, na situação da Figura 24.6 há 9 blocos livres no disco, mas somente podem ser criados arquivos com até 3 blocos. As técnicas de alocação *first/best/worst-fit* utilizadas em gerência de memória também podem ser aplicadas para

Algoritmo 3 Localizar a posição do i -ésimo byte do arquivo no disco

Entrada: i : número do byte a localizar no arquivo B : tamanho dos blocos lógicos, em bytes P : tamanho dos ponteiros de blocos, em bytes b_0 : número do bloco do disco onde o arquivo inicia**Saída:** b_i : número do bloco do disco onde se encontra o byte i o_i : posição do byte i dentro do bloco b_i (*offset*) \div : divisão inteira

mod: módulo (resto da divisão inteira)

$$b_i = b_0 + i \div B$$

$$o_i = i \bmod B$$

return(b_i, o_i)

atenuar este problema. Contudo, a desfragmentação de um disco é problemática, pois pode ser uma operação muito lenta e os arquivos não devem ser usados durante sua realização.

A baixa flexibilidade desta estratégia e a possibilidade de fragmentação externa limitam muito seu uso em sistemas operacionais de propósito geral, nos quais arquivos são constantemente criados, modificados e destruídos. Todavia, ela pode encontrar uso em situações específicas, nas quais os arquivos não sejam modificados constantemente e seja necessário rapidez nos acessos sequenciais e aleatórios aos dados. Um exemplo dessa situação são sistemas dedicados para reprodução de dados multimídia, como áudio e vídeo. O sistema ISO 9660, usado em CD-ROMs, é um exemplo de sistema de arquivos que usa a alocação contígua.

24.5.2 Alocação encadeada simples

Esta forma de alocação foi proposta para resolver os principais problemas da alocação contígua: sua baixa flexibilidade e a fragmentação externa. Na alocação encadeada, cada bloco do arquivo no disco contém dados do arquivo e também um ponteiro para o próximo bloco, ou seja, um campo indicando a posição no disco do próximo bloco do arquivo. Desta forma é construída uma lista encadeada de blocos para cada arquivo, não sendo mais necessário manter os blocos do arquivo lado a lado no disco. Esta estratégia elimina a fragmentação externa, pois todos os blocos livres do disco podem ser utilizados sem restrições, e permite que arquivos sejam criados sem a necessidade de definir seu tamanho final. A Figura 24.7 ilustra um exemplo dessa abordagem.

Nesta abordagem, o acesso sequencial aos dados do arquivo é simples e rápido, pois cada bloco do arquivo contém um “ponteiro” para o próximo bloco. Todavia, caso os blocos estejam muito espalhados no disco, a cabeça de leitura terá de fazer muitos deslocamentos, diminuindo o desempenho de acesso ao disco. Já o acesso aleatório ao arquivo fica muito prejudicado com esta abordagem: caso se deseje acessar o n -ésimo

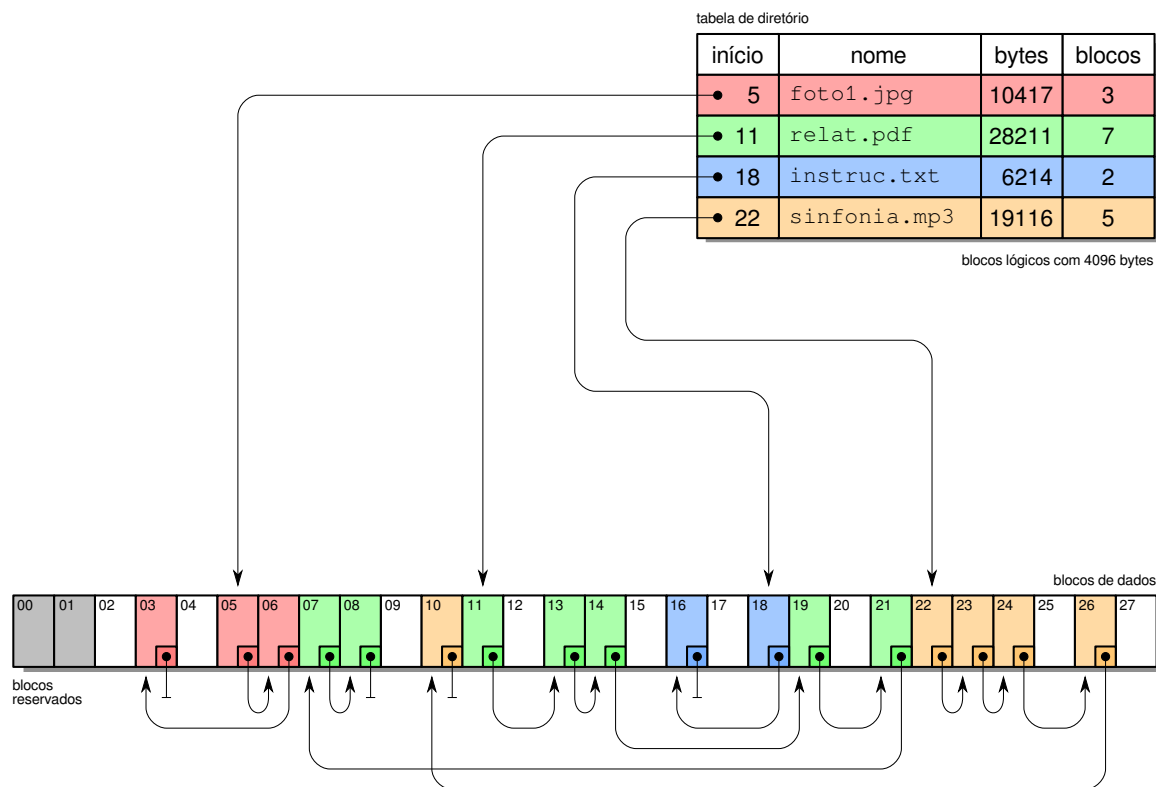


Figura 24.7: Estratégia de alocação encadeada simples.

bloco do arquivo, os $n - 1$ blocos anteriores terão de ser lidos em sequência, para poder encontrar os ponteiros que levam ao bloco desejado. O algoritmo 4 mostra claramente esse problema, indicado através do laço *while*.

A dependência dos ponteiros de blocos também acarreta problemas de robustez: caso um bloco do arquivo seja corrompido ou se torne defeituoso, todos os blocos posteriores a este também ficarão inacessíveis. Por outro lado, esta abordagem é muito flexível, pois não há necessidade de se definir o tamanho máximo do arquivo durante sua criação, e arquivos podem ser expandidos ou reduzidos sem maiores dificuldades. Além disso, qualquer bloco livre do disco pode ser usados por qualquer arquivo, eliminando a fragmentação externa.

24.5.3 Alocação encadeada FAT

Os principais problemas da alocação encadeada simples são o baixo desempenho nos acessos aleatórios e a relativa fragilidade em relação a erros nos blocos do disco. Ambos os problemas provêm do fato de que os ponteiros dos blocos são armazenados nos próprios blocos, junto dos dados do arquivo. Para resolver esse problema, os ponteiros podem ser retirados dos blocos de dados e armazenados em uma tabela separada. Essa tabela é denominada **Tabela de Alocação de Arquivos** (FAT - *File Allocation Table*), sendo a base dos sistemas de arquivos FAT12, FAT16 e FAT32 usados nos sistemas operacionais MS-DOS, Windows e em muitos dispositivos de armazenamento portáteis, como *pendrives*, reprodutores MP3 e câmeras fotográficas digitais.

Na abordagem FAT, os ponteiros dos blocos de cada arquivo são mantidos em uma tabela única, armazenada em blocos reservados no início da partição. Cada entrada dessa tabela corresponde a um bloco lógico do disco e contém um ponteiro indicando o

Algoritmo 4 Localizar a posição do i -ésimo byte do arquivo no disco**Entrada:**

i : número do byte a localizar no arquivo
 B : tamanho dos blocos lógicos, em bytes
 P : tamanho dos ponteiros de blocos, em bytes
 b_0 : número do primeiro bloco do arquivo no disco

Saída:

b_i : número do bloco do disco onde se encontra o byte i
 o_i : posição do byte i dentro do bloco b_i (*offset*)

$$b_{aux} = b_0$$

▷ define bloco inicial do percurso

$$b = i \div (B - P)$$

▷ calcula número de blocos a percorrer

while $b > 0$ **do**

$$block = read_block(b_{aux})$$

▷ lê um bloco do disco

$$b_{aux} = \text{ponteiro extraído de } block$$

$$b = b - 1$$
end while

$$b_i = b_{aux}$$

$$o_i = i \bmod (B - P)$$

$$\text{return}(b_i, o_i)$$

próximo bloco do mesmo arquivo. As entradas da tabela também podem conter valores especiais para indicar o último bloco de cada arquivo, blocos livres, blocos defeituosos e blocos reservados. Uma cópia dessa tabela é mantida em cache na memória durante o uso do sistema, para melhorar o desempenho na localização dos blocos dos arquivos. A Figura 24.8 apresenta o conteúdo da tabela de alocação de arquivos para o exemplo apresentado anteriormente na Figura 24.7.

A alocação com FAT resolve o problema de desempenho da alocação sequencial simples, mantendo sua flexibilidade de uso. A tabela FAT é um dado crítico para a robustez do sistema, pois o acesso aos arquivos ficará comprometido caso ela seja corrompida. Para minimizar esse problema, cópias da FAT são geralmente mantidas na área reservada.

24.5.4 Alocação indexada simples

Nesta abordagem, a estrutura em lista encadeada da estratégia anterior é substituída por um vetor contendo um *índice de blocos* do arquivo. Cada entrada desse índice corresponde a um bloco do arquivo e aponta para a posição desse bloco no disco. O índice de blocos de cada arquivo é mantido no disco em uma estrutura denominada *nó de índice* (*index node*) ou simplesmente *nó- i* (*i -node*). O *i -node* de cada arquivo contém, além de seu índice de blocos, os principais atributos do mesmo, como tamanho, permissões, datas de acesso, etc. Os *i -nodes* de todos os arquivos são agrupados em uma tabela de *i -nodes*, mantida em uma área reservada do disco, separada dos blocos de dados dos arquivos. A Figura 24.9 apresenta um exemplo de alocação indexada.

máximo de arquivos ou diretórios que podem ser criados na partição (pois cada arquivo ou diretório consome um *i-node*).

24.5.5 Alocação indexada multinível

Para aumentar o tamanho máximo dos arquivos armazenados, algumas das entradas do índice de blocos podem ser transformadas em ponteiros indiretos. Essas entradas apontam para blocos do disco que contêm outros ponteiros, criando assim uma estrutura em árvore. Considerando um sistema com blocos lógicos de 4 Kbytes e ponteiros de 32 bits (4 bytes), cada bloco lógico pode conter 1.024 ponteiros para outros blocos, o que aumenta muito a capacidade do índice de blocos. Além de ponteiros indiretos, podem ser usados ponteiros dupla e triplamente indiretos. Os sistemas de arquivos Ext2/3/4 do Linux, por exemplo, usam *i-nodes* com 12 ponteiros diretos (que apontam para blocos de dados), um ponteiro indireto, um ponteiro duplamente indireto e um ponteiro triplamente indireto. A estrutura do *inode* do sistema de arquivos Ext4 do Linux é apresentada na Tabela 24.1, e a estrutura de ponteiros é apresentada na Figura 24.10.

Offset	Size	Name	Description
0x00	2	<code>i_mode</code>	entry type and permissions
0x02	2	<code>i_uid</code>	user ID
0x04	4	<code>i_size_lo</code>	size (bytes)
0x08	4	<code>i_atime</code>	data access time
0x0C	4	<code>i_ctime</code>	inode change time
0x10	4	<code>i_mtime</code>	data modif time
0x14	4	<code>i_dtime</code>	deletion time
0x18	2	<code>i_gid</code>	group ID
0x1A	2	<code>i_links_count</code>	hard links counter
0x1C	2	<code>i_blocks_lo</code>	number of blocks
0x20	4	<code>i_flags</code>	several flag bits
0x24	4	<code>l_i_version</code>	inode version
0x28	60	<code>i_block[15]</code>	block map (pointers)
...

Tabela 24.1: Estrutura (parcial) do *inode* do sistema de arquivos Ext4 do Linux; o vetor “`i_block`” contém os 15 ponteiros usados na alocação indexada multinível.

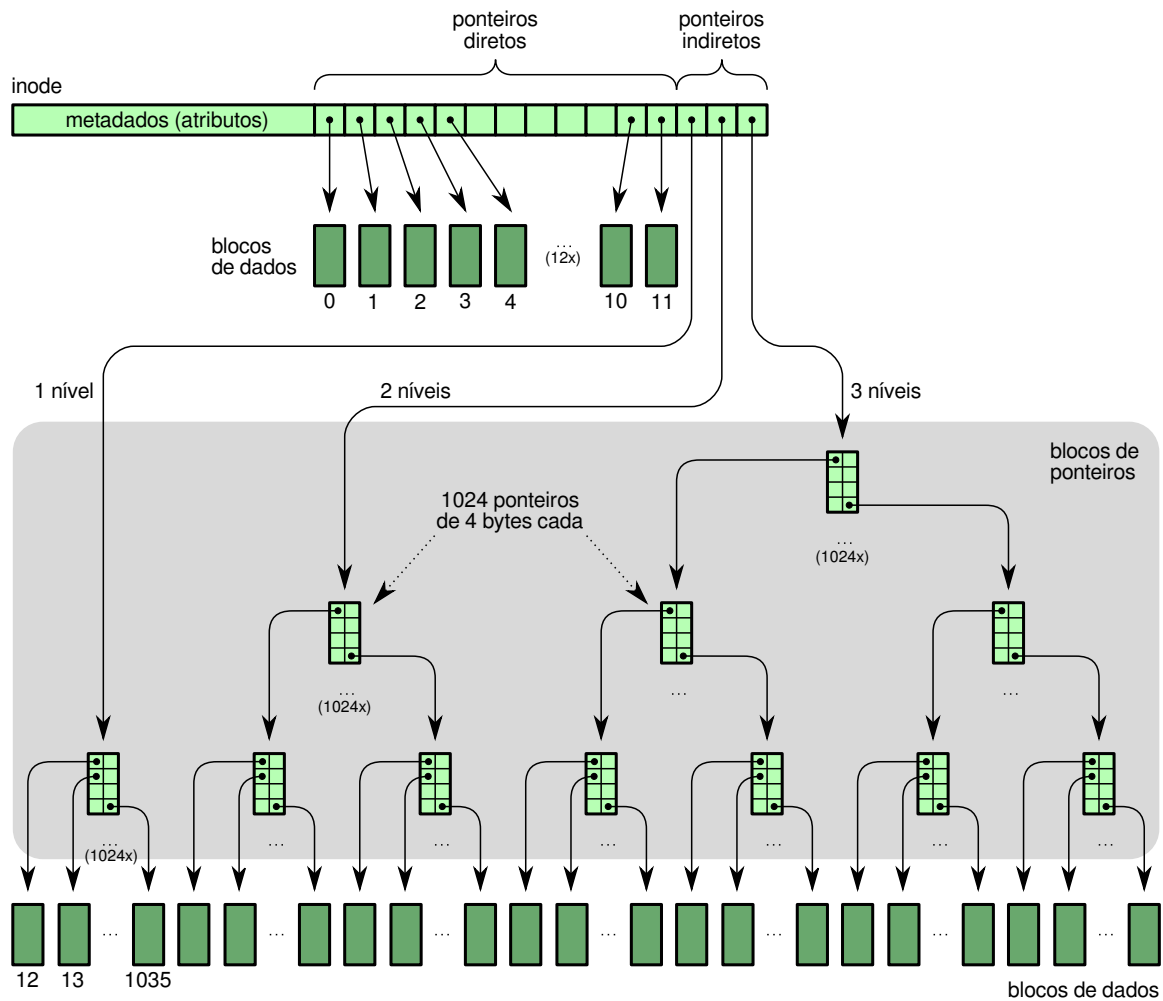


Figura 24.10: Estratégia de alocação indexada multi-nível.

Considerando blocos lógicos de 4 Kbytes e ponteiros de 4 bytes, cada bloco de ponteiros contém 1.024 ponteiros. Dessa forma, o cálculo do tamanho máximo de um arquivo nesse sistema é simples:

$$\begin{aligned}
 max &= 4.096 \times 12 && \text{(ponteiros diretos)} \\
 &+ 4.096 \times 1.024 && \text{(ponteiro 1-indireto)} \\
 &+ 4.096 \times 1.024 \times 1.024 && \text{(ponteiro 2-indireto)} \\
 &+ 4.096 \times 1.024 \times 1.024 \times 1.024 && \text{(ponteiro 3-indireto)} \\
 &= 4.402.345.721.856 \text{ bytes} \\
 max &\approx 4 \text{ Tbytes}
 \end{aligned}$$

Apesar dessa estrutura aparentemente complexa, a localização e acesso de um bloco do arquivo no disco permanece relativamente simples, pois a estrutura homogênea de ponteiros permite calcular rapidamente a localização dos blocos. A localização do bloco lógico de disco correspondente ao *i*-ésimo bloco lógico de um arquivo segue o algoritmo 5.

Em relação ao desempenho, pode-se afirmar que esta estratégia é bastante rápida, tanto para acessos sequenciais quanto para acessos aleatórios a blocos, devido aos índices de ponteiros dos blocos presentes nos *i-nodes*. Contudo, no caso de blocos

Algoritmo 5 Localizar a posição do i -ésimo byte do arquivo no disco

```

1: Entrada:
2:  $i$ : número do byte a localizar no arquivo
3:  $B$ : tamanho dos blocos lógicos, em bytes (4.096 neste exemplo)
4:  $ptr[0...14]$ : vetor de ponteiros contido no  $i$ -node
5:  $block[0...1023]$ : bloco de ponteiros para outros blocos (1.024 ponteiros de 4 bytes)

6: Saída:
7:  $b_i$ : número do bloco do disco onde se encontra o byte  $i$ 
8:  $o_i$ : posição do byte  $i$  dentro do bloco  $b_i$  (offset)

9:  $o_i = i \bmod B$ 
10:  $pos = i \div B$ 
11: if  $pos < 12$  then
12:    $b_i = ptr[pos]$ 
13: else
14:    $pos = pos - 12$ 
15:   if  $pos < 1024$  then
16:      $block_1 = read\_block(ptr[12])$ 
17:      $b_i = block_1[pos]$ 
18:   else
19:      $pos = pos - 1024$ 
20:     if  $pos < 1024^2$  then
21:        $block_1 = read\_block(ptr[13])$ 
22:        $block_2 = read\_block(block_1[pos \div 1024])$ 
23:        $b_i = block_2[pos \bmod 1024]$ 
24:     else
25:        $pos = pos - 1024^2$ 
26:        $block_1 = read\_block(ptr[14])$ 
27:        $block_2 = read\_block(block_1[pos \div (1024^2)])$ 
28:        $block_3 = read\_block(block_2[(pos \div 1024) \bmod 1024])$ 
29:        $b_i = block_3[pos \bmod 1024]$ 
30:     end if
31:   end if
32: end if
33: return( $b_i, o_i$ )

```

▶ usar ponteiros diretos
 ▶ o ponteiro é o número do bloco b_i

▶ usar ponteiro 1-indireto
 ▶ ler 1º bloco de ponteiros
 ▶ achar endereço do bloco b_i

▶ usar ponteiro 2-indireto
 ▶ ler 1º bloco de ponteiros
 ▶ ler 2º bloco de ponteiros
 ▶ achar endereço do bloco b_i
 ▶ usar ponteiro 3-indireto

▶ ler 1º bloco de ponteiros
 ▶ ler 2º bloco
 ▶ ler 3º bloco
 ▶ achar endereço do bloco b_i

situados no final de arquivos muito grandes, podem ser necessários três ou quatro acessos a disco adicionais para localizar o bloco desejado, devido à necessidade de ler os blocos com ponteiros indiretos.

Defeitos em blocos de dados não afetam os demais blocos de dados, o que torna esta estratégia robusta. Todavia, defeitos nos metadados (o *i-node* ou os blocos de ponteiros) podem danificar grandes extensões do arquivo; por isso, muitos sistemas que usam esta estratégia implementam técnicas de redundância de *i-nodes* e metadados para melhorar a robustez.

Em relação à flexibilidade, pode-se afirmar que esta forma de alocação é tão flexível quanto a alocação encadeada, não apresentando fragmentação externa e permitindo o uso de todas as áreas livres do disco para armazenar dados. Todavia, são impostos limites para o tamanho máximo dos arquivos criados e para o número máximo de arquivos no volume.

24.5.6 Alocação por extensões

Embora seja muito flexível, a alocação indexada é ineficiente para o armazenamento de arquivos muito grandes. A necessidade de manter um ponteiro para cada bloco do arquivo alocado exige uma grande quantidade de metadados, que precisa ser armazenada e gerenciada. Por outro lado, observa-se que a alocação contígua é excessivamente rígida mas é muito econômica e eficiente em relação aos metadados: nessa forma de alocação, para cada arquivo basta manter um ponteiro para seu início e conhecer seu tamanho em blocos. Dessas observações surgiu a alocação por extensões, que mistura características de ambas as abordagens.

Na alocação por extensões, cada arquivo é alocado no disco como um conjunto de grupos de blocos contíguos, denominados *extensões* (do inglês *extents*). Os metadados necessários para gerir cada extensão são poucos: o número do bloco lógico inicial da extensão, do bloco físico inicial correspondente e a quantidade de blocos naquela extensão. A figura 24.11 ilustra o funcionamento da alocação por extensões. Nela, o arquivo `relat.pdf` tem 52.748 bytes (13 blocos de 4.096 bytes cada) e está alocado no disco em 4 extensões:

1. 3 blocos, blocos lógicos 0 a 2, blocos físicos 6 a 8;
2. 5 blocos, blocos lógicos 3 a 7, blocos físicos 11 a 15;
3. 4 blocos, blocos lógicos 8 a 11, blocos físicos 21 a 24;
4. 1 bloco, blocos lógicos 12 a 12, blocos físicos 27 a 27.

Os metadados de cada extensão do arquivo são mantidos em *descritores de extensão*. O *i-node* de cada arquivo tem espaço para registrar um número pequeno de descritores de extensão, pois *i-nodes* têm tamanho fixo. Caso sejam necessárias mais extensões, o sistema de arquivos constrói uma árvore de descritores de extensão usando blocos de disco adicionais, de forma similar à usada na alocação indexada multinível (Seção 24.5.5). Extensões são implementadas na maioria dos sistemas de arquivos atuais, como NTFS (Windows), HFS (MacOS), Ext4 e ZFS (Linux).

O uso de extensões contribui para diminuir o grau de espalhamento dos blocos dos arquivos no disco, melhorando o desempenho do sistema. Deve-se observar que o número de extensões usadas por um arquivo não reflete necessariamente seu tamanho:

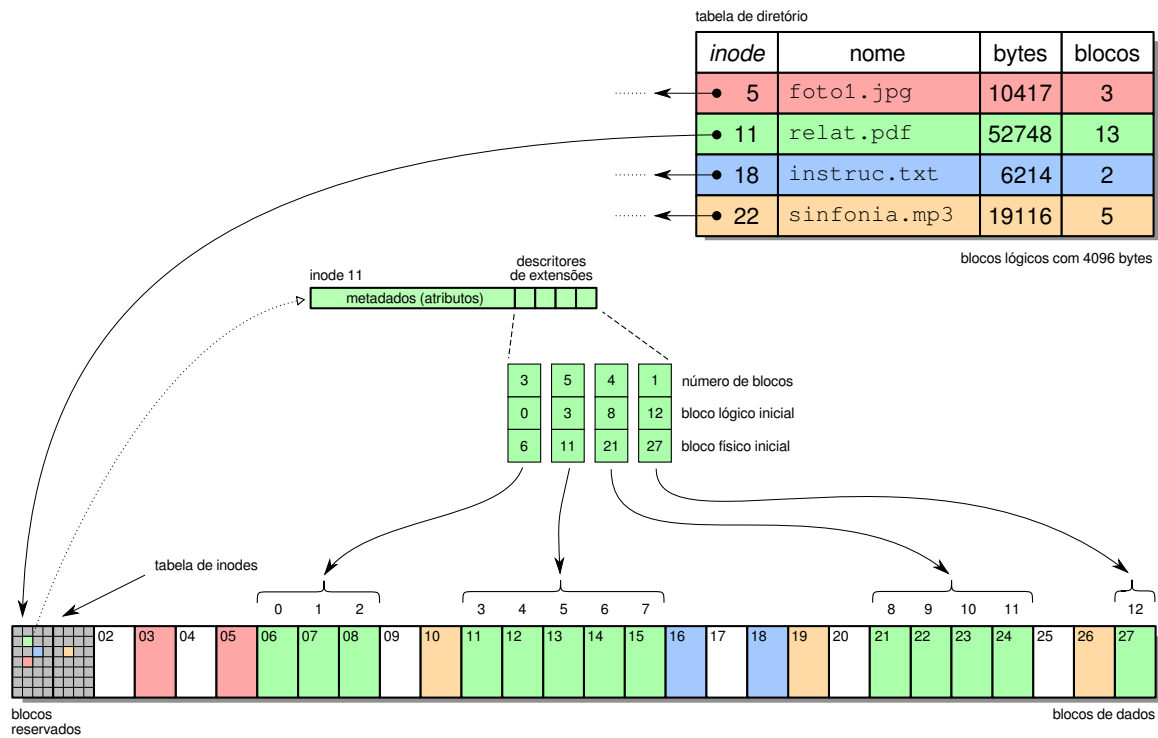


Figura 24.11: Estratégia de alocação por extensões.

um arquivo de 10 GBytes pode ter seus blocos totalmente contíguos no disco, usando portanto somente uma extensão, enquanto um arquivo de 100 KBytes muito espalhado pode estar usando dezenas de extensões. Neste caso, o SO pode prover ferramentas para reorganizar arquivos muito fragmentados, agrupando suas extensões.

24.5.7 Análise comparativa

A Tabela 24.2 traz um comparativo entre as principais formas de alocação estudadas neste texto, sob a ótica de suas características de rapidez, robustez e flexibilidade de uso.

Estratégia	Contígua	Encadeada	FAT	Indexada	Extensões
Rapidez (acesso sequencial)	Alta, os blocos do arquivo estão sempre em sequência no disco.	Alta, se os blocos do arquivo estiverem próximos no disco.	Alta, se os blocos do arquivo estiverem próximos no disco.	Alta, se os blocos do arquivo estiverem próximos no disco.	Alta, se houver poucas extensões.
Rapidez (acesso aleatório)	Alta, as posições dos blocos podem ser calculadas sem acessar o disco.	Baixa, é necessário ler todos os blocos a partir do início do arquivo até encontrar o bloco desejado.	Alta, se os blocos do arquivo estiverem próximos no disco.	Alta, se os blocos do arquivo estiverem próximos no disco.	Alta, se houver poucas extensões.
Robustez	Alta, blocos com erro não impedem o acesso aos demais blocos do arquivo.	Baixa: erro em um bloco leva à perda dos dados daquele bloco e de todos os blocos subsequentes do arquivo.	Alta, desde que não ocorram erros na tabela de alocação.	Alta, desde que não ocorram erros no <i>i-node</i> nem nos blocos de ponteiros.	Alta, blocos com erro não impedem o acesso aos demais blocos do arquivo.
Flexibilidade	Baixa, o tamanho máximo dos arquivos deve ser conhecido a priori; nem sempre é possível aumentar o tamanho de um arquivo existente.	Alta, arquivos podem ser criados em qualquer local do disco, sem risco de fragmentação externa.	Alta, arquivos podem ser criados em qualquer local do disco, sem risco de fragmentação externa.	Alta, arquivos podem ser criados em qualquer local do disco, sem risco de fragmentação externa.	Alta, mas há risco de fragmentação excessiva e queda de desempenho.
Limites	O tamanho de um arquivo é limitado ao tamanho do disco.	O número de bits do ponteiro limita o número de blocos endereçáveis e o tamanho do arquivo.	O número de bits do ponteiro limita o número de blocos endereçáveis e o tamanho do arquivo.	Número de ponteiros no <i>i-node</i> limita o tamanho do arquivo; tamanho da tabela de <i>i-nodes</i> limita número de arquivos.	O tamanho de um arquivo é limitado ao tamanho do disco.

Tabela 24.2: Quadro comparativo das estratégias de alocação de arquivos.

24.6 Gestão do espaço livre

Além de manter informações sobre que blocos são usados por cada arquivo no disco, a camada de alocação de arquivos deve manter um registro atualizado de quais blocos estão livres, ou seja não estão ocupados por nenhum arquivo ou metadado. Isto é importante para obter rapidamente blocos no momento de criar um novo arquivo ou aumentar um arquivo existente. Algumas técnicas de gerência de blocos são sugeridas na literatura [Silberschatz et al., 2001; Tanenbaum, 2003]: o mapa de bits, a lista de blocos livres e a tabela de extensões livres.

Na abordagem de **mapa de bits**, um pequeno conjunto de blocos na área reservada do volume é usado para manter um mapa de bits. Nesse mapa, cada bit representa um bloco lógico da partição, que pode estar livre ou ocupado. Para o exemplo da Figura 24.9, o mapa de bits de blocos livres seria: 1101 0111 1011 0110 1011 0111 1010 (considerando 0 para bloco livre e incluindo os blocos reservados no início da partição). O mapa de bits tem como vantagens ser simples de implementar e ser bem compacto: para um disco de 500 GBytes com blocos lógicos de 4.096 bytes, seriam necessários

131.072.000 bits no mapa, o que representa 16.384.000 bytes (16 Mbytes), ocupando 4.000 blocos (ou seja, 0,003% do total de blocos lógicos do disco).

Na abordagem de **lista de blocos livres**, cada bloco livre contém um ponteiro para o próximo bloco livre do disco, de forma similar à alocação encadeada de arquivos vista na Seção 24.5.2. Essa abordagem é ineficiente, por exigir um acesso a disco para cada bloco livre requisitado. Uma melhoria simples consiste em armazenar em cada bloco livre um vetor de ponteiros para outros blocos livres; o último ponteiro desse vetor apontaria para um novo bloco livre contendo mais um vetor de ponteiros, e assim sucessivamente. Essa abordagem permite obter um grande número de blocos livres a cada acesso a disco.

Outra melhoria similar consiste em manter uma **tabela de extensões livres**, contendo a localização e o tamanho de todos os grupos de blocos contíguos livres no disco. Cada entrada dessa tabela contém o número do bloco inicial e o número de blocos no grupo, de forma similar à usada na alocação por extensões (Seção 24.5.6). Para o exemplo da figura 24.6, a tabela de extensões livres teria o seguinte conteúdo: $\{[2, 3], [8, 3], [20, 2], [27, 1]\}$, com entradas na forma $[bloco\ inicial, tamanho]$.

Por outro lado, a abordagem de alocação FAT (Seção 24.5.3) usa a própria tabela de alocação de arquivos para gerenciar os blocos livres, que são indicados por flags específicos; no exemplo da Figura 24.8, os blocos livres estão indicados com o flag “F” na tabela. Para encontrar blocos livres ou liberar blocos usados, basta consultar ou modificar as entradas da tabela.

É importante lembrar que, além de manter o registro dos blocos livres, na alocação indexada também é necessário gerenciar o uso dos *inodes*, ou seja, manter o registros de quais *inodes* estão livres ou ocupados. Isso geralmente também é feito através de um mapa de bits.

24.7 Falhas e recuperação

Um sistema de arquivos armazena dados e metadados. Os **dados** são o conteúdo dos arquivos em si, ou seja, a informação contida nos próprios arquivos. Os **metadados** são todas as demais informações mantidas no disco, que descrevem a organização do sistema de arquivos, a estrutura dos diretórios e dos próprios arquivos, como os atributos de cada arquivo e a localização de seus blocos de dados no disco. Dessa forma, os metadados de um sistema de arquivos definem uma imensa estrutura de dados armazenada no disco e atualizada pelo núcleo do sistema operacional através da leitura e escrita de blocos do disco.

Uma alteração no sistema de arquivos pode exigir várias operações individuais no disco, envolvendo dados e/ou metadados. Por exemplo, as seguintes operações com metadados são necessárias para criar o arquivo “/home/user/novo.txt”, inicialmente vazio:

1. Encontrar um *inode* livre e marcá-lo como ocupado no mapa de *inodes*;
2. Preencher esse *inode* com os atributos do arquivo “novo.txt”;
3. Localizar o *inode* do diretório “/home/user/”;
4. Incluir uma nova entrada na tabela do diretório “/home/user/”, contendo o nome e o *inode* do arquivo “novo.txt”.

Sistemas operacionais estão sujeitos a paradas abruptas e inesperadas, denominadas falhas por *crash*, geralmente ocasionadas por quedas de energia ou problemas no hardware ou software. Então, caso ocorra uma falha por *crash* durante a sequência de operações em disco acima descritas, o sistema de arquivos pode ficar inconsistente:

- Caso ocorra um *crash* entre as operações 1 e 2, somente a escrita 1 será completada (atualizar mapa de *inodes*). Em consequência, um *inode* será marcado como ocupado no mapa de *inodes* livres/ocupados, mas não será efetivamente usado.
- Caso ocorra um *crash* entre as operações 2 e 4, somente as escritas 1 e 2 serão realizadas (atualizar mapa de *inodes* e atualizar *inode*). Em consequência, um *inode* será marcado como ocupado e será preenchido na tabela de *inodes*, mas não estará referenciado em nenhum diretório e portanto não poderá ser aberto ou localizado no futuro.

As operações em disco são feitas de forma sequencial e podem ser relativamente demoradas, sobretudo em discos rígidos. Para ganhar desempenho, muitas operações em disco são realizadas usando cache assíncrono, ou seja, as escritas em disco solicitadas pelo SO são mantidas em um cache *write-back* na memória RAM e só são efetuadas no disco mais tarde (vide Seção 24.4.2).

O uso de cache em RAM melhora muito o desempenho dos acessos a disco, mas aumenta o risco de perda de dados em caso de *crash*. Falhas na atualização de dados dos arquivos podem levar à perda de conteúdo dos mesmos, mas falhas envolvendo os metadados podem levar à perda de subárvores de diretórios ou do sistema de arquivos inteiro. Por isso, geralmente os metadados são atualizados de forma síncrona (estratégia *write-through*).

24.7.1 Verificação do sistema de arquivos

Quando um sistema de arquivos é desligado de forma correta, seja através do desligamento normal do computador ou da desmontagem do sistema de arquivos, essa informação é registrada no VBR (*Volume Boot Record*) das partições de disco em uso. Ao reiniciar o sistema, o SO verifica o status de cada partição: caso ela não tenha sido desligada ou removida corretamente, presume-se que tenha ocorrido um *crash* e possam existir inconsistências; a partição deve então passar por um procedimento de verificação e recuperação de sua consistência.

Um mecanismo básico de verificação de consistência do sistema de arquivos consiste em percorrer todas as estruturas de dados internas do mesmo para verificar se estão corretas, corrigindo os problemas encontrados quando for possível. Nos sistemas UNIX, esse procedimento é feito pela ferramenta “*fsck*” (do inglês *File System Check*). Por exemplo, a ferramenta “*fsck.ext4*”, usada na verificação de sistemas de arquivos Ext4, realiza diversas análises, entre as quais:

- O conteúdo da tabela de *inodes* está correto?
- Cada bloco de dados em uso está sendo referenciado por um único *inode*?
- Para todos os *inodes* contendo diretórios, as entradas do diretório apontam para *inodes* em uso?
- Todos os *inodes* em uso estão referenciados por ao menos uma tabela de diretório?

- Os mapas de *inodes* e de blocos livres estão corretos?

Para auxiliar a verificação de consistência, vários sistemas de arquivos atuais implementam somas de verificação (*checksums*) que permitem detectar corrupção nos metadados e, em alguns casos, também nos dados dos arquivos. Essas somas de verificação são armazenadas junto dos metadados do sistema de arquivos, são verificadas a cada acesso aos mesmos e atualizadas quando os metadados são alterados [Arpaci-Dusseau and Arpaci-Dusseau, 2014].

O processo de verificação de inconsistências na inicialização pode levar várias dezenas de minutos ou mesmo horas se o sistema de arquivos for grande. Além disso, ele deve ser feito antes que o sistema de arquivos seja disponibilizado às aplicações, o que pode tornar a inicialização do sistema operacional muito demorada.

24.7.2 Sistemas de arquivos com registro (*journal*)

Uma abordagem mais eficiente para assegurar a consistência do sistema de arquivos é registrar previamente as alterações que serão efetuadas em um registro, denominado diário ou *journal*. Esse registro fica no disco, em uma área separada do sistema de arquivos, e armazena uma descrição das próximas alterações que serão realizadas no mesmo, sendo portanto escrito **antes de realizá-las**. Essa técnica é bastante usada na gerência de transações em bancos de dados, onde é conhecida como *write-ahead logging* [Arpaci-Dusseau and Arpaci-Dusseau, 2014].

O funcionamento do registro é relativamente simples:

1. Antes de realizar uma alteração no sistema de arquivos que exija várias operações, o SO registra no *journal* quais escritas serão realizadas, com informações suficientes para poder repeti-las se for necessário. Por exemplo, a criação de um arquivo `"/home/user/novo.txt"` demanda as seguintes operações de alteração de metadados (assumindo que será usado o *inode* livre 317):
 - (a) Marcar o *inode* 317 como ocupado no mapa de *inodes*;
 - (b) Preencher o *inode* 317 com os atributos do arquivo `"novo.txt"`;
 - (c) Incluir no diretório `"/home/user/"` uma entrada com o nome `"novo.txt"` apontando para o *inode* 317.
2. Após registrá-las no *journal*, o SO realiza as operações necessárias no sistema de arquivos, uma de cada vez.
3. Caso todas as operações sejam concluídas com sucesso, o registro das mesmas no *journal* pode ser descartado.
4. Caso ocorra uma falha de *crash* durante as operações, ao reiniciar o SO analisa o *journal* da partição e completa as operações pendentes, levando o sistema de arquivos de volta a um estado consistente; após isto, as entradas correspondentes no *journal* podem ser descartadas.

O registro das operações pendentes é implementado como um buffer circular de tamanho fixo, no qual as escritas são geralmente sequenciais, contribuindo para um bom desempenho. A escrita de informações no registro é feita de forma síncrona, para

diminuir o risco de perda de informações no próprio *journal*. Além disso, são usadas somas de verificação (*checksums*) para assegurar que as entradas do registro estejam consistentes.

A técnica de *journaling* é usada na maioria dos sistemas de arquivos atuais, como NTFS (Windows), HFS (MacOS), EXT4 (Linux) e JFS (IBM). Usualmente o *journaling* se aplica somente aos metadados do sistema de arquivos, por uma questão de desempenho. Ele pode ser estendido também aos dados dos arquivos, para evitar a perda de dados no caso de *crash*, mas o custo é elevado: com *journaling*, cada bloco de dados precisa ser escrito no disco duas vezes: uma no registro e outra no seu destino final.

24.7.3 Sistemas de arquivos *copy-on-write* (CoW)

Outra abordagem usada para recuperar falhas em sistemas de arquivos de forma mais rápida e eficiente que a verificação completa são os sistemas de arquivos com cópia ao escrever (CoW, do inglês *Copy-on-Write*), à vezes também chamados sistemas de arquivos com versionamento (*Versioning File Systems*). Os sistemas de arquivos ZFS e BTRFS, cuja popularidade vem crescendo em ambientes Linux, usam a abordagem *copy-on-write* [Rodeh et al., 2013].

Em um sistema de arquivos convencional, quando o SO atualiza o conteúdo de um arquivo, ele reescreve o conteúdo atualizado do bloco em sua posição atual no disco, sobrescrevendo o conteúdo anterior. Em um sistema de arquivos *copy-on-write*, o novo conteúdo é escrito em um bloco livre do disco e os metadados são ajustados para apontar para essa versão atualizada do bloco. Como os metadados também estão em blocos do disco, eles também são atualizados usando blocos livres de forma similar, preservando suas versões anteriores. Os blocos contendo versões anteriores dos dados e metadados são mantidos no disco enquanto houver espaço livre. À medida em que espaço livre for necessário, os blocos com versões mais antigas são automaticamente descartados.

Esta estratégia permite manter no disco várias “fotografias” (*snapshots*) do sistema de arquivos em estados anteriores. Esses *snapshots* permitem a um usuário resgatar versões anteriores de um arquivo ou diretório, o que é útil no caso de apagamentos ou sobrescritas acidentais. Além disso, após uma falha por *crash* o sistema operacional pode rapidamente encontrar o último *snapshot* consistente antes da falha e retornar a ele, descartando versões corrompidas dos dados ou metadados.

24.8 Exemplo: o sistema de arquivos Ext4

O sistema de arquivos Ext4 (*Fourth Extended filesystem*) é o sistema de arquivos mais usado em sistemas Linux atualmente [Mathur et al., 2007; Kernel Development Community, 2025]. Como seu nome indica, ele deriva dos sistemas de arquivos anteriores Ext, Ext2 e Ext3. Suas principais características são:

- Na configuração default, suporta partições com até 64 ZBytes e arquivos com até 16 TBytes.
- Usa *journaling* e *checksum* de metadados para melhorar a tolerância a falhas.
- Suporta quotas, controle de acesso avançado e criptografia.

- Suporta alocação indexada multinível e alocação por extensões; cada *inode* contém 4 descritores de extensões e pode apontar para uma árvore de descritores, se mais extensões forem necessárias;
- A gestão de espaço livre é feita por bitmaps de blocos livres e bitmaps de *inodes* livres;

A Figura 24.12 traz uma visão simplificada da estrutura do sistema de arquivos Ext4; uma descrição mais precisa e detalhada pode ser consultada em [Kernel Development Community, 2025].

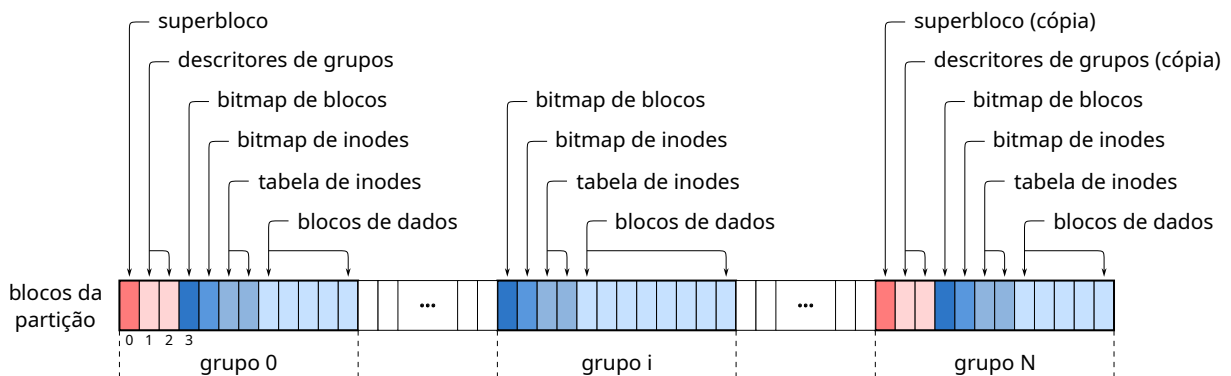


Figura 24.12: Visão geral do sistema de arquivos Ext4.

Uma partição formatada com Ext4 é dividida em blocos de 4 KBytes, que são organizados em grupos de blocos contíguos de mesmo tamanho (geralmente 32.768 blocos ou 128 Mbytes por grupo). O primeiro grupo de blocos da partição (grupo 0) tem sua estrutura interna constituída pelas seguintes áreas:

- Superbloco (1 bloco): contém uma descrição detalhada da partição: tipo e versão do sistema de arquivos, tamanho dos blocos e dos grupos, número total de blocos, de grupos e de *inodes*, número de blocos e de *inodes* livres, características suportadas, localização do journal, status, etc. Os primeiros 1.024 bytes do superbloco são reservados para o código de inicialização (boot) usado pela BIOS, caso seja necessário.
- Tabela de descritores de grupos (alguns blocos): cada entrada desta tabela descreve um grupo de blocos contíguos da partição, contendo informações sobre a localização do grupo e de seus campos internos, além de *checksums* dos dados do grupo para detectar inconsistências.
- Bitmap de blocos (1 bloco): mapa de bits dos blocos do grupo, no qual cada bit indica se o bloco correspondente está livre ou ocupado;
- Bitmap de *inodes* (1 bloco): mapa de bits dos *inodes* do grupo, no qual cada bit indica se o *inode* correspondente está livre ou ocupado;
- Tabela de *inodes* (alguns blocos): contém os *inodes* dos arquivos e diretórios do grupo; cada *inode* pode armazenar arquivos usando uma árvore de ponteiros para blocos (alocação indexada multinível, Seção 24.5.5) ou uma tabela de extensões (alocação por extensões, Seção 24.5.6);

- Blocos de dados (todos os demais blocos do grupo): contêm os dados dos arquivos, blocos de ponteiros indiretos (para a alocação indexada) e blocos com árvores de extensões (para a alocação por extensões).

Os demais grupos de blocos de uma partição Ext4 contêm os bitmaps de blocos e de *inodes*, a tabela de *inodes* e os blocos de dados. Além disso, alguns grupos espalhados ao longo da partição contêm cópias do superbloco e da tabela de descritores de grupos, para uso caso o grupo 0 seja perdido ou corrompido. Por fim, alguns *inodes* do grupo 0 são reservados para uso específico, conforme indicado na Tabela 24.3.

Inode	Uso
0	não usado
1	lista de blocos defeituosos na partição
2	diretório raiz
3	controle de quota de uso por usuários
4	controle de quota de uso por grupos de usuários
5	inicializador (<i>Boot loader</i>)
6, 7, 9, 10	reservado
8	arquivo de registro (<i>journal</i>)
11	primeiro <i>inode</i> disponível para arquivos

Tabela 24.3: *Inodes* especiais no grupo 0 do sistema Ext4.

Exercícios

1. Apresente a arquitetura de gerência de arquivos presente em um sistema operacional típico, explicando seus principais elementos constituintes.
2. Enumere principais problemas a resolver na implementação de um sistema de arquivos.
3. Explique o que é alocação contígua de arquivos, apresentando suas vantagens e desvantagens.
4. No contexto de alocação de arquivos, o que significa o termo *best-fit*?
5. Explique a alocação de arquivos em listas encadeadas, apresentando suas principais vantagens e desvantagens.
6. Explique a estrutura do sistema de arquivos conhecido como FAT, comentando sobre suas qualidades e deficiências.
7. Por que a alocação de arquivos em listas encadeadas é considerada pouco robusta? O que pode ser feito para melhorar essa característica?
8. Explique o esquema de alocação indexada de arquivos usando índices multi-níveis.

9. O que é fragmentação interna e fragmentação externa? Por que elas ocorrem?
10. Analise o impacto das fragmentações interna e externa nos sistemas de alocação contígua, indexada e por lista encadeadas.
11. Considere um sistema operacional hipotético que suporte simultaneamente as estratégias de alocação contígua, encadeada e indexada para armazenamento de arquivos em disco. Que critérios devem ser considerados para decidir a estratégia a usar para cada arquivo em particular?
12. Sobre as afirmações a seguir, relativas às técnicas de alocação de arquivos, indique quais são incorretas, justificando sua resposta:
 - (a) A alocação contígua é muito utilizada em sistemas desktop, por sua flexibilidade.
 - (b) A alocação FAT é uma alocação encadeada na qual os ponteiros de blocos foram transferidos para um vetor de ponteiros.
 - (c) Na alocação indexada os custos de acesso sequencial e aleatório a blocos são similares.
 - (d) Na alocação contígua, blocos defeituosos podem impedir o acesso aos demais blocos do arquivo.
 - (e) Na alocação contígua, o custo de acesso a blocos aleatórios é alto.
 - (f) Apesar de complexa, a alocação indexada é muito usada em *desktops* e servidores.
13. Considerando um arquivo com 500 blocos em disco, calcule quantas leituras e quantas escritas em disco são necessárias para (a) inserir um novo bloco no início do arquivo ou (b) inserir um novo bloco no final do arquivo, usando as formas de alocação de blocos contígua, encadeada e indexada.

Alocação	Contígua		Encadeada		Indexada	
	leituras	escritas	leituras	escritas	leituras	escritas
Inserir um novo bloco no início do arquivo						
Inserir um novo bloco no final do arquivo						

Observações:

- (a) Considere somente as operações de leitura e escrita nos blocos do próprio arquivo e no *i-node*; a tabela de diretório sempre está em memória;
- (b) Para a alocação contígua, assuma que não há espaço livre depois do arquivo, somente antes dele;

- (c) Para a alocação encadeada, assuma que a tabela de diretório contém apenas um ponteiro para o início do arquivo no disco. Os ponteiros dos blocos estão contidos nos próprios blocos;
- (d) Para a alocação indexada, considere *i-nodes* com somente um nível, contendo somente os ponteiros para os blocos de dados. O *i-node* está no disco.
14. Considere um disco rígido com capacidade total de 1 Mbyte, dividido em 1.024 blocos de 1.024 bytes cada. Os dez primeiros blocos do disco são reservados para a tabela de partições, o código de inicialização (*boot*) e o diretório raiz do sistema de arquivos. Calcule o tamanho máximo de arquivo (em bytes) que pode ser criado nesse disco para cada uma das formas de alocação a seguir, explicando seu raciocínio:
- (a) Alocação contígua.
- (b) Alocação encadeada, com ponteiros de 64 bits contidos nos próprios blocos.
- (c) Alocação indexada, com *i-nodes* contendo somente ponteiros diretos de 64 bits; considere que o *i-node* não contém meta-dados, somente ponteiros, e que ele ocupa exatamente um bloco do disco.
15. Considerando a tabela FAT (*File Allocation Table*) a seguir, indique:
- (a) o número de blocos ocupados pelo arquivo *relat.pdf*;
- (b) o tamanho (em blocos) do maior arquivo que ainda pode ser criado nesse disco;
- (c) quais arquivos estão íntegros e quais estão corrompidos por blocos defeituosos (*bad blocks*);
- (d) quantos blocos do disco estão perdidos, ou seja, não são usados por arquivos nem estão marcados como livres ou defeituosos.

Na tabela, a letra R indica bloco reservado (*Reserved*), F indica bloco livre (*Free*), L indica o último bloco de um arquivo (*Last*) e B indica bloco defeituoso (*Bad*).

arquivo	início	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
readme.txt	76	R	R	R	R	R	F	17	F	15	68	13	53	F	L	63	L	F	26	F	F
icone.gif	14	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
retrato.jpg	29	33	L	F	38	L	F	11	55	F	36	F	35	43	B	F	B	20	F	8	F
relat.pdf	6	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
format.exe	31	21	32	F	50	B	L	F	F	40	F	L	45	F	58	F	B	F	F	72	F
carta.doc	67	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
programa.c	73	44	F	F	51	F	F	F	60	24	F	F	F	10	27	F	F	41	F	L	F

16. O sistema de arquivos indexado do sistema *Minix* possui os seguintes campos em cada *i-node*:
- meta-dados (tipo, dono, grupo, permissões, datas e tamanho)
 - 7 ponteiros diretos
 - 1 ponteiro indireto

- 1 ponteiro duplamente indireto

A implementação básica desse sistema de arquivos considera blocos de 1.024 bytes e ponteiros de 32 bits. Desenhe o diagrama do sistema de arquivos e calcule o tamanho máximo de arquivo que ele suporta, indicando seu raciocínio.

17. O sistema de arquivos indexado `ext2fs`, usado no *Linux*, possui os seguintes campos em cada *i-node*:

- meta-dados (tipo, dono, grupo, permissões, datas e tamanho)
- 12 ponteiros diretos
- 1 ponteiro indireto
- 1 ponteiro duplamente indireto
- 1 ponteiro triplamente indireto

A implementação básica do `ext2fs` considera blocos de 1.024 bytes e ponteiros de 64 bits. Desenhe o diagrama do sistema de arquivos e determine o tamanho máximo de arquivo que ele suporta, indicando seu raciocínio.

18. Explique como é efetuada a gerência de espaço livre através de *bitmaps*.

Referências

- R. Arpaci-Dusseau and A. Arpaci-Dusseau. *Multi-level Feedback Queue*, chapter 8. Arpaci-Dusseau Books, 2014. <http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf>.
- Kernel Development Community. The Linux kernel documentation – ext4 data structures and algorithms. <https://docs.kernel.org/filesystems/ext4>, 2025. Accessed: 2025-02-27.
- A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33. Citeseer, 2007.
- O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-Tree filesystem. *ACM Transactions on Storage*, 9(3), Aug. 2013. ISSN 1553-3077.
- A. Silberschatz, P. Galvin, and G. Gagne. *Sistemas Operacionais – Conceitos e Aplicações*. Campus, 2001.
- A. Tanenbaum. *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall, 2003.
- U. Vahalia. *UNIX Internals – The New Frontiers*. Prentice-Hall, 1996.