Sistemas Operacionais: Conceitos e Mecanismos

VI - Gerência de Arquivos

Prof. Carlos Alberto Maziero
DInf UFPR
http://www.inf.ufpr.br/maziero

4 de agosto de 2017

Este texto foi produzido usando exclusivamente software livre: Sistema Operacional *GNU/Linux* (distribuições *Fedora* e *Ubuntu*), compilador de texto LaTeX 2$_\varepsilon$, gerenciador de referências *BibTeX*, editor gráfico *Inkscape*, criadores de gráficos *GNUPlot* e *GraphViz* e processador PS/PDF *GhostScript*, entre outros.

Sumário

Resumo

Um sistema operacional tem por finalidade permitir que os usuários do computador executem aplicações, como editores de texto, jogos, reprodutores de áudio e vídeo, etc. Essas aplicações processam informações como textos, músicas e filmes, armazenados sob a forma de arquivos em um disco rígido ou outro meio. Este módulo apresenta a noção de arquivo, suas principais características e formas de acesso, a organização de arquivos em diretórios e as técnicas usadas para criar e gerenciar arquivos nos dispositivos de armazenamento.

1 Arquivos

Desde os primórdios da computação, percebeu-se a necessidade de armazenar informações para uso posterior, como programas e dados. Hoje, parte importante do uso de um computador consiste em recuperar e apresentar informações previamente armazenadas, como documentos, fotografias, músicas e vídeos. O próprio sistema operacional também precisa manter informações armazenadas para uso posterior, como programas, bibliotecas e configurações. Geralmente essas informações devem ser armazenadas em um dispositivo não volátil, que preserve seu conteúdo mesmo quando o computador estiver desligado. Para simplificar o armazenamento e busca de informações, surgiu o conceito de *arquivo*, que será discutido a seguir.

1.1 O conceito de arquivo

Um arquivo é basicamente um conjunto de dados armazenados em um dispositivo físico não volátil, com um nome ou outra referência que permita sua localização posterior. Do ponto de vista do usuário e das aplicações, o arquivo é a unidade básica de armazenamento de informação em um dispositivo não volátil, pois para eles não há forma mais simples de armazenamento persistente de dados. Arquivos são extremamente versáteis em conteúdo e capacidade: podem conter desde um texto ASCII com alguns bytes até sequências de vídeo com dezenas de gigabytes, ou mesmo mais.

Como um dispositivo de armazenamento pode conter milhões de arquivos, estes são organizados em estruturas hierárquicas denominadas *diretórios* (conforme ilustrado na Figura 1 e discutido mais detalhadamente na Seção 3.1). A organização física e lógica dos arquivos e diretórios dentro de um dispositivo é denominada *sistema de arquivos*. Um sistema de arquivos pode ser visto como uma imensa estrutura de dados armazenada de forma persistente em um dispositivo físico. Existe um grande número de sistemas de arquivos, dentre os quais podem ser citados o NTFS (nos sistemas Windows), Ext2/Ext3/Ext4 (Linux), HPFS (MacOS), FFS (Solaris) e FAT (usado em *pendrives* USB, máquinas fotográficas digitais e leitores MP3). A organização dos sistemas de arquivos será discutida na Seção 4.

1.2 Atributos

Conforme apresentado, um arquivo é uma unidade de armazenamento de informações que podem ser dados, código executável, etc. Cada arquivo é caracterizado por um

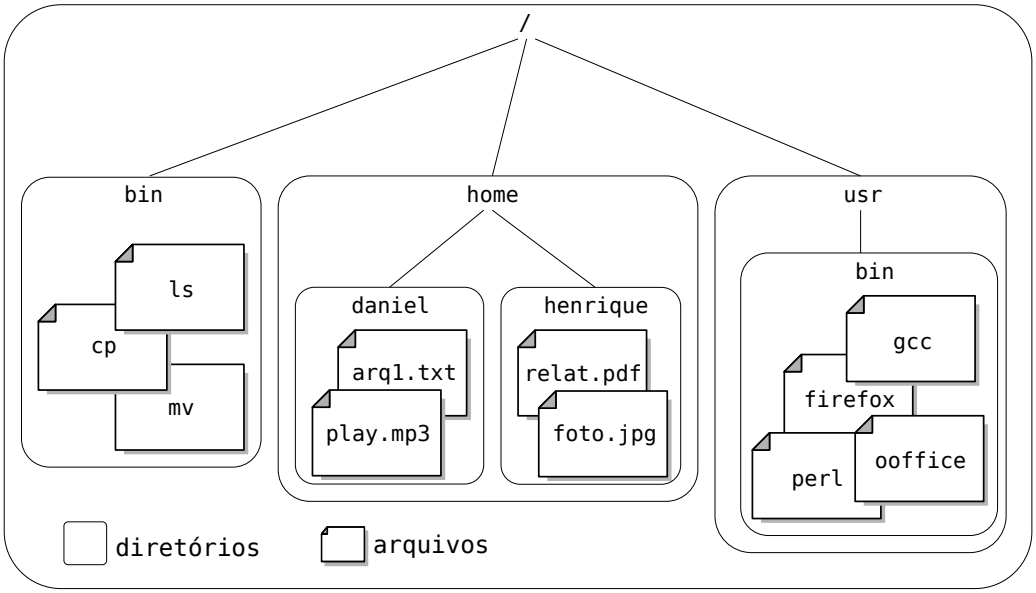

Figura 1: Arquivos organizados em diretórios dentro de um dispositivo.

conjunto de *atributos*, que podem variar de acordo com o sistema de arquivos utilizado. Os atributos mais usuais são:

Nome: string de caracteres que identifica o arquivo para o usuário, como "`foto1.jpg`", "`relatório.pdf`", "`hello.c`", etc.;

Tipo: indicação do formato dos dados contidos no arquivo, como áudio, vídeo, imagem, texto, etc. Muitos sistemas operacionais usam parte do nome do arquivo para identificar o tipo de seu conteúdo, na forma de uma extensão: "`.doc`", "`.jpg`", "`.mp3`", etc.;

Tamanho: indicação do tamanho do conteúdo do arquivo, em bytes ou registros;

Datas: para fins de gerência, é importante manter as datas mais importantes relacionadas ao arquivo, como suas datas de criação, de último acesso e de última modificação do conteúdo;

Proprietário: em sistemas multiusuários, cada arquivo tem um proprietário, que deve estar corretamente identificado;

Permissões de acesso: indicam que usuários têm acesso àquele arquivo e que formas de acesso são permitidas (leitura, escrita, remoção, etc.);

Localização: indicação do dispositivo físico onde o arquivo se encontra e da posição do arquivo dentro do mesmo;

Outros atributos: vários outros atributos podem ser associados a um arquivo, por exemplo para indicar se é um arquivo de sistema, se está visível aos usuários, se tem conteúdo binário ou textual, etc. Cada sistema de arquivos normalmente define seus próprios atributos específicos, além dos atributos usuais.

Nem sempre os atributos oferecidos por um sistema de arquivos são suficientes para exprimir todas as informações a respeito de um arquivo. Nesse caso, a "solução" encontrada pelos usuários é usar o nome do arquivo para exprimir a informação desejada. Por exemplo, em muitos sistemas a parte final do nome do arquivo (sua extensão) é usada para identificar o formato de seu conteúdo. Outra situação frequente é usar parte do nome do arquivo para identificar diferentes versões do mesmo conteúdo[1]: `relat-v1.txt`, `relat-v2.txt`, etc.

1.3 Operações

As aplicações e o sistema operacional usam arquivos para armazenar e recuperar dados. O uso dos arquivos é feito através de um conjunto de operações, geralmente implementadas sob a forma de chamadas de sistema e funções de bibliotecas. As operações básicas envolvendo arquivos são:

Criar: a criação de um novo arquivo implica em alocar espaço para ele no dispositivo de armazenamento e definir seus atributos (nome, localização, proprietário, permissões de acesso, etc.);

Abrir: antes que uma aplicação possa ler ou escrever dados em um arquivo, ela deve solicitar ao sistema operacional a "abertura" desse arquivo. O sistema irá então verificar se o arquivo existe, verificar se as permissões associadas ao arquivo permitem aquele acesso, localizar seu conteúdo no dispositivo de armazenamento e criar uma referência para ele na memória da aplicação;

Ler: permite transferir dados presentes no arquivo para uma área de memória da aplicação;

Escrever: permite transferir dados na memória da aplicação para o arquivo no dispositivo físico; os novos dados podem ser adicionados no final do arquivo ou sobrescrever dados já existentes;

Mudar atributos: para modificar outras características do arquivo, como nome, localização, proprietário, permissões, etc.

Fechar: ao concluir o uso do arquivo, a aplicação deve informar ao sistema operacional que o mesmo não é mais necessário, a fim de liberar as estruturas de gerência do arquivo na memória do núcleo;

Remover: para eliminar o arquivo do dispositivo, descartando seus dados e liberando o espaço ocupado por ele.

Além dessas operações básicas, outras operações podem ser definidas, como truncar, copiar, mover ou renomear arquivos. Todavia, essas operações geralmente podem ser construídas usando as operações básicas.

[1] Alguns sistemas operacionais, como o *TOPS-20* e o *OpenVMS*, possuem sistemas de arquivos com suporte automático a múltiplas versões do mesmo arquivo.

1.4 Formatos

Em sua forma mais simples, um arquivo contém basicamente uma sequência de bytes, que pode estar estruturada de diversas formas para representar diferentes tipos de informação. O formato ou estrutura interna de um arquivo pode ser definido – e reconhecido – pelo núcleo do sistema operacional ou somente pelas aplicações. O núcleo do sistema geralmente reconhece apenas alguns poucos formatos de arquivos, como binários executáveis e bibliotecas. Os demais formatos de arquivos são vistos pelo núcleo apenas como sequências de bytes sem um significado específico, cabendo às aplicações interpretá-los.

Os arquivos de dados convencionais são estruturados pelas aplicações para armazenar os mais diversos tipos de informações, como imagens, sons e documentos. Uma aplicação pode definir um formato próprio de armazenamento ou seguir formatos padronizados. Por exemplo, há um grande número de formatos públicos padronizados para o armazenamento de imagens, como JPEG, GIF, PNG e TIFF, mas também existem formatos de arquivos proprietários, definidos por algumas aplicações específicas, como o formato PSD (do editor *Adobe Photoshop*) e o formato XCF (do editor gráfico *GIMP*). A adoção de um formato proprietário ou exclusivo dificulta a ampla utilização das informações armazenadas, pois somente aplicações que reconheçam aquele formato conseguem ler corretamente as informações contidas no arquivo.

1.4.1 Arquivos de registros

Alguns núcleos de sistemas operacionais oferecem arquivos com estruturas internas que vão além da simples sequência de bytes. Por exemplo, o sistema *OpenVMS* [Rice, 2000] proporciona *arquivos baseados em registros*, cujo conteúdo é visto pelas aplicações como uma sequência linear de registros de tamanho fixo ou variável, e também *arquivos indexados*, nos quais podem ser armazenados pares {*chave/valor*}, de forma similar a um banco de dados relacional. A Figura 2 ilustra a estrutura interna desses dois tipos de arquivos.

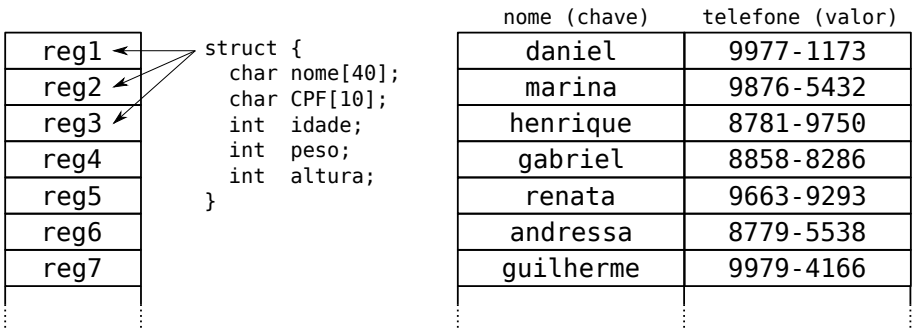

Figura 2: Arquivos estruturados: registros em sequência e registros indexados.

Nos sistemas operacionais cujo núcleo não suporta arquivos estruturados como registros, essa funcionalidade pode ser facilmente obtida através de bibliotecas específicas ou do suporte de execução de algumas linguagens de programação. Por exemplo, a

biblioteca *Berkeley DB* disponível em plataformas UNIX oferece suporte à indexação de registros sobre arquivos UNIX convencionais.

1.4.2 Arquivos de texto

Um tipo de arquivo de uso muito frequente é o arquivo de *texto puro* (ou *plain text*). Esse tipo de arquivo é muito usado para armazenar informações textuais simples, como códigos fontes de programas, arquivos de configuração, páginas HTML, dados em XML, etc. Um arquivo de texto é formado por linhas de caracteres ASCII de tamanho variável, separadas por caracteres de controle. Nos sistemas UNIX, as linhas são separadas por um caractere *New Line* (ASCII 10 ou "\n"). Já nos sistemas DOS/Windows, as linhas de um arquivo de texto são separadas por dois caracteres: o caractere *Carriage Return* (ASCII 13 ou "\r") seguido do caractere *New Line*. Por exemplo, considere o seguinte programa em C armazenado em um arquivo `hello.c` (os caracteres "␣" indicam espaços em branco):

```
1 int␣main()
2 {
3 ␣␣printf("Hello,␣world\n");
4 ␣␣exit(0);
5 }
```

O arquivo de texto `hello.c` seria armazenado da seguinte forma[2] em um ambiente UNIX:

```
1 0000  69 6e 74 20 6d 61 69 6e 28 29 0a 7b 0a 20 20 70
2        i  n  t  ␣  m  a  i  n  (  ) \n  {  \n ␣  ␣  p
3 0010  72 69 6e 74 66 28 22 48 65 6c 6c 6f 2c 20 77 6f
4        r  i  n  t  f  (  "  H  e  l  l  o  ,  ␣  w  o
5 0020  72 6c 64 5c 6e 22 29 3b 0a 20 20 65 78 69 74 28
6        r  l  d  \  n  "  )  ;  \n ␣  ␣  e  x  i  t  (
7 0030  30 29 3b 0a 7d 0a
8        0  )  ;  \n  }  \n
```

Por outro lado, o mesmo arquivo `hello.c` seria armazenado da seguinte forma em um sistema DOS/Windows:

```
1 0000  69 6e 74 20 6d 61 69 6e 28 29 0d 0a 7b 0d 0a 20
2        i  n  t  ␣  m  a  i  n  (  ) \r \n  {  \r \n ␣
3 0010  20 70 72 69 6e 74 66 28 22 48 65 6c 6c 6f 2c 20
4        ␣  p  r  i  n  t  f  (  "  H  e  l  l  o  ,  ␣
5 0020  77 6f 72 6c 64 5c 6e 22 29 3b 0d 0a 20 20 65 78
6        w  o  r  l  d  \  n  "  )  ;  \r \n ␣  ␣  e  x
7 0030  69 74 28 30 29 3b 0d 0a 7d 0d 0a
8        i  t  (  0  )  ;  \r \n  }  \r \n
```

[2] Listagem obtida através do comando `hd` do Linux, que apresenta o conteúdo de um arquivo em hexadecimal e seus caracteres ASCII correspondentes, byte por byte.

Essa diferença na forma de representação da separação entre linhas pode provocar problemas em arquivos de texto transferidos entre sistemas Windows e UNIX, caso não seja feita a devida conversão.

1.4.3 Arquivos executáveis

Um arquivo executável é dividido internamente em várias seções, para conter código, tabelas de símbolos (variáveis e funções), listas de dependências (bibliotecas necessárias) e outras informações de configuração. A organização interna de um arquivo executável ou biblioteca depende do sistema operacional para o qual foi definido. Os formatos de executáveis mais populares atualmente são [Levine, 2000]:

- **ELF** (*Executable and Linking Format*): formato de arquivo usado para programas executáveis e bibliotecas na maior parte das plataformas UNIX modernas. É composto por um cabeçalho e várias seções de dados, contendo código executável, tabelas de símbolos e informações de relocação de código.
- **PE** (*Portable Executable*): é o formato usado para executáveis e bibliotecas na plataforma Windows. Consiste basicamente em uma adaptação do antigo formato COFF usado em plataformas UNIX.

A Figura 3 ilustra a estrutura interna de um arquivo executável no formato ELF, usado tipicamente em sistemas UNIX (Linux, Solaris, etc.). Esse arquivo é dividido em seções, que representam trechos de código e dados sujeitos a ligação dinâmica e relocação; as seções são agrupadas em segmentos, de forma a facilitar a carga em memória do código e o lançamento do processo.

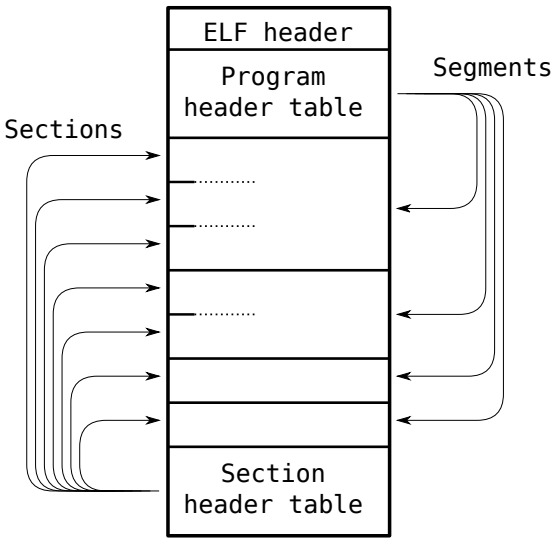

Figura 3: Estrutura interna de um arquivo executável em formato ELF [Levine, 2000].

Além de executáveis e bibliotecas, o núcleo de um sistema operacional costuma reconhecer alguns tipos de arquivos não convencionais, como diretórios, atalhos (*links*), dispositivos físicos e estruturas de comunicação do núcleo, como *sockets*, *pipes* e filas de mensagens (vide Seção 1.5).

1.4.4 Identificação de conteúdo

Um problema importante relacionado aos formatos de arquivos é a correta identificação de seu conteúdo pelos usuários e aplicações. Já que um arquivo de dados pode ser visto como uma simples sequência de bytes, como é possível saber que tipo de informação essa sequência representa? Uma solução simples para esse problema consiste em indicar o tipo do conteúdo como parte do nome do arquivo: um arquivo "`praia.jpg`" provavelmente contém uma imagem em formato JPEG, enquanto um arquivo "`entrevista.mp3`" contém áudio em formato MP3. Essa estratégia, amplamente utilizada em muitos sistemas operacionais, foi introduzida nos anos 1980 pelo sistema operacional DOS. Naquele sistema, os arquivos eram nomeados segundo uma abordagem denominada "8.3", ou seja, 8 caracteres seguidos de um ponto (".") e mais 3 caracteres de extensão, para definir o tipo do arquivo.

Outra abordagem, frequentemente usada em sistemas UNIX, é o uso de alguns bytes no início de cada arquivo para a definição de seu tipo. Esses bytes iniciais são denominados "números mágicos" (*magic numbers*), e são usados em muitos tipos de arquivos, como exemplificado na Tabela 1:

Tabela 1: Números mágicos de alguns tipos de arquivos

Tipo de arquivo	bytes iniciais	Tipo de arquivo	bytes iniciais
Documento PostScript	`%!`	Documento PDF	`%PDF`
Imagem GIF	`GIF89a`	Imagem JPEG	`0xFFD8`
Música MIDI	`MThd`	Classes Java (JAR)	`0xCAFEBABE`

Nos sistemas UNIX, o utilitário `file` permite identificar o tipo de arquivo através da análise de seus bytes iniciais e do restante de sua estrutura interna, sem levar em conta o nome do arquivo. Por isso, constitui uma ferramenta importante para identificar arquivos desconhecidos ou com extensão errada.

Além do uso de extensões no nome do arquivo e de números mágicos, alguns sistemas operacionais definem atributos adicionais no sistema de arquivos para indicar o conteúdo de cada arquivo. Por exemplo, o sistema operacional MacOS 9 definia um atributo com 4 bytes para identificar o tipo de cada arquivo (*file type*), e outro atributo com 4 bytes para indicar a aplicação que o criou (*creator application*). Os tipos de arquivos e aplicações são definidos em uma tabela mantida pelo fabricante do sistema. Assim, quando o usuário solicitar a abertura de um determinado arquivo, o sistema irá escolher a aplicação que o criou, se ela estiver presente. Caso contrário, pode indicar ao usuário uma relação de aplicações aptas a abrir aquele tipo de arquivo.

Recentemente, a necessidade de transferir arquivos através de e-mail e de páginas Web levou à definição de um padrão de tipagem de arquivos conhecido como *Tipos MIME* (da sigla *Multipurpose Internet Mail Extensions*) [Freed and Borenstein, 1996]. O padrão MIME define tipos de arquivos através de uma notação uniformizada na forma "`tipo/subtipo`". Alguns exemplos de tipos de arquivos definidos segundo o padrão MIME são apresentados na Tabela 2.

O padrão MIME é usado para identificar arquivos transferidos como anexos de e-mail e conteúdos recuperados de páginas Web. Alguns sistemas operacionais, como

Tabela 2: Tipos MIME correspondentes a alguns formatos de arquivos

Tipo MIME	Significado
`application/java-archive`	Arquivo de classes Java (JAR)
`application/msword`	Documento do Microsoft Word
`application/vnd.oasis.opendocument.text`	Documento do OpenOffice
`audio/midi`	Áudio em formato MIDI
`audio/mpeg`	Áudio em formato MP3
`image/jpeg`	Imagem em formato JPEG
`image/png`	Imagem em formato PNG
`text/csv`	Texto em formato CSV (*Comma-separated Values*)
`text/html`	Texto HTML
`text/plain`	Texto puro
`text/rtf`	Texto em formato RTF (*Rich Text Format*)
`text/x-csrc`	Código-fonte em C
`video/quicktime`	Vídeo no formato *Quicktime*

o BeOS e o MacOS X, definem atributos de acordo com esse padrão para identificar o conteúdo de cada arquivo dentro do sistema de arquivos.

1.5 Arquivos especiais

O conceito de arquivo é ao mesmo tempo simples e poderoso, o que motivou sua utilização de forma quase universal. Além do armazenamento de código e dados, arquivos também podem ser usados como:

Abstração de dispositivos de baixo nível: os sistemas UNIX costumam mapear as interfaces de acesso de vários dispositivos físicos em arquivos dentro do diretório `/dev` (de *devices*), como por exemplo:

- `/dev/ttyS0`: porta de comunicação serial COM1;
- `/dev/audio`: placa de som;
- `/dev/sda1`: primeira partição do primeiro disco SCSI (ou SATA).

Abstração de interfaces do núcleo: em sistemas UNIX, os diretórios `/proc` e `/sys` permitem consultar e/ou modificar informações internas do núcleo do sistema operacional, dos processos em execução e dos *drivers* de dispositivos. Por exemplo, alguns arquivos oferecidos pelo Linux:

- `/proc/cpuinfo`: informações sobre os processadores disponíveis no sistema;
- `/proc/3754/maps`: disposição das áreas de memória alocadas para o processo cujo identificador (PID) é 3754;
- `/sys/block/sda/queue/scheduler`: definição da política de escalonamento de disco (vide Seção **??**) a ser usada no acesso ao disco `/dev/sda`.

Canais de comunicação: na família de protocolos de rede TCP/IP, a metáfora de arquivo é usada como interface para os canais de comunicação: uma conexão TCP é

apresentada aos dois processos envolvidos como um arquivo, sobre o qual eles podem escrever (enviar) e ler (receber) dados entre si. Vários mecanismos de comunicação local entre processos de um sistema também usam a metáfora do arquivo, como é o caso dos *pipes* em UNIX.

Em alguns sistemas operacionais experimentais, como o *Plan 9* [Pike et al., 1993, Pike et al., 1995] e o *Inferno* [Dorward et al., 1997], todos os recursos e entidades físicas e lógicas do sistema são mapeadas sob a forma de arquivos: processos, *threads*, conexões de rede, usuários, sessões de usuários, janelas gráficas, áreas de memória alocadas, etc. Assim, para finalizar um determinado processo, encerrar uma conexão de rede ou desconectar um usuário, basta remover o arquivo correspondente.

Embora o foco deste texto esteja concentrado em arquivos convencionais, que visam o armazenamento de informações (bytes ou registros), muitos dos conceitos aqui expostos são igualmente aplicáveis aos arquivos não convencionais descritos nesta seção.

2 Uso de arquivos

Arquivos são usados por processos para ler e escrever dados de forma não volátil. Para usar arquivos, um processo tem à sua disposição uma *interface de acesso*, que depende da linguagem utilizada e do sistema operacional subjacente. Essa interface normalmente é composta por uma representação lógica de cada arquivo usado pelo processo (uma *referência* ao arquivo) e por um conjunto de funções (ou métodos) para realizar operações sobre esses arquivos. Através dessa interface, os processos podem localizar arquivos no disco, ler e modificar seu conteúdo, entre outras operações.

Na sequência desta seção serão discutidos aspectos relativos ao uso de arquivos, como a abertura do arquivo, as formas de acesso aos seus dados, o controle de acesso e problemas associados ao compartilhamento de arquivos entre vários processos.

2.1 Abertura de um arquivo

Para poder ler ou escrever dados em um arquivo, cada aplicação precisa antes "abri-lo". A **abertura de um arquivo** consiste basicamente em preparar as estruturas de memória necessárias para acessar os dados do arquivo em questão. Assim, para abrir um arquivo, o núcleo do sistema operacional deve realizar as seguintes operações:

1. Localizar o arquivo no dispositivo físico, usando seu nome e caminho de acesso (vide Seção 3.2);
2. Verificar se a aplicação tem permissão para usar aquele arquivo da forma desejada (leitura e/ou escrita);
3. Criar uma estrutura na memória do núcleo para representar o arquivo aberto;
4. Inserir uma referência a essa estrutura na lista de arquivos abertos mantida pelo sistema, para fins de gerência;

5. Devolver à aplicação uma referência a essa estrutura, para ser usada nos acessos subsequentes ao arquivo recém aberto.

Concluída a abertura do arquivo, o processo solicitante recebe do núcleo uma referência para o arquivo recém aberto, que deve ser informada pelo processo em suas operações subsequentes envolvendo aquele arquivo. Assim que o processo tiver terminado de usar um arquivo, ele deve solicitar ao núcleo o **fechamento do arquivo**, que implica em concluir as operações de escrita eventualmente pendentes e remover da memória do núcleo as estruturas de gerência criadas durante sua abertura. Normalmente, os arquivos abertos são automaticamente fechados quando do encerramento do processo, mas pode ser necessário fechá-los antes disso, caso seja um processo com vida longa, como um *daemon* servidor de páginas Web, ou que abra muitos arquivos, como um compilador.

As referências a arquivos abertos usadas pelas aplicações dependem da linguagem de programação utilizada para construí-las. Por exemplo, em um programa escrito na linguagem C, cada arquivo aberto é representado por uma variável dinâmica do tipo `FILE*`, que é denominada um **ponteiro de arquivo** (*file pointer*). Essa variável dinâmica é alocada no momento da abertura do arquivo e serve como uma referência ao mesmo nas operações de acesso subsequentes. Já em Java, as referências a arquivos abertos são objetos instanciados a partir da classe `File`. Na linguagem Python existem os *file objects*, criados a partir da chamada `open`.

Por outro lado, cada sistema operacional tem sua própria convenção para a representação de arquivos abertos. Por exemplo, em sistemas Windows os arquivos abertos por um processo são representados pelo núcleo por **referências de arquivos** (*file handles*), que são estruturas de dados criadas pelo núcleo para representar cada arquivo aberto. Por outro lado, em sistemas UNIX os arquivos abertos por um processo são representados por **descritores de arquivos** (*file descriptors*). Um descritor de arquivo aberto é um número inteiro não negativo, usado como índice em uma tabela que relaciona os arquivos abertos por aquele processo, mantida pelo núcleo. Dessa forma, cabe às bibliotecas e ao suporte de execução de cada linguagem de programação mapear a representação de arquivo aberto fornecida pelo núcleo do sistema operacional subjacente na referência de arquivo aberto usada por aquela linguagem. Esse mapeamento é necessário para garantir que as aplicações que usam arquivos (ou seja, quase todas elas) sejam portáveis entre sistemas operacionais distintos.

2.2 Formas de acesso

Uma vez aberto um arquivo, a aplicação pode ler os dados contidos nele, modificá-los ou escrever novos dados. Há várias formas de se ler ou escrever dados em um arquivo, que dependem da estrutura interna do mesmo. Considerando apenas arquivos simples, vistos como uma sequência de bytes, duas formas de acesso são usuais: o *acesso sequencial* e o *acesso direto* (ou acesso aleatório).

No **acesso sequencial**, os dados são sempre lidos e/ou escritos em sequência, do início ao final do arquivo. Para cada arquivo aberto por uma aplicação é definido um *ponteiro de acesso*, que inicialmente aponta para a primeira posição do arquivo. A cada leitura ou escrita, esse ponteiro é incrementado e passa a indicar a posição da próxima

leitura ou escrita. Quando esse ponteiro atinge o final do arquivo, as leituras não são mais permitidas, mas as escritas ainda o são, permitindo acrescentar dados ao final do mesmo. A chegada do ponteiro ao final do arquivo é normalmente sinalizada ao processo através de um *flag* de fim de arquivo (*EoF - End-of-File*).

A Figura 4 traz um exemplo de acesso sequencial em leitura a um arquivo, mostrando a evolução do ponteiro do arquivo durante uma sequência de leituras. A primeira leitura no arquivo traz a *string* "`Qui␣scribit␣bis`", a segunda leitura traz "`␣legit.␣`", e assim sucessivamente. O acesso sequencial é implementado em praticamente todos os sistemas operacionais de mercado e constitui a forma mais usual de acesso a arquivos, usada pela maioria das aplicações.

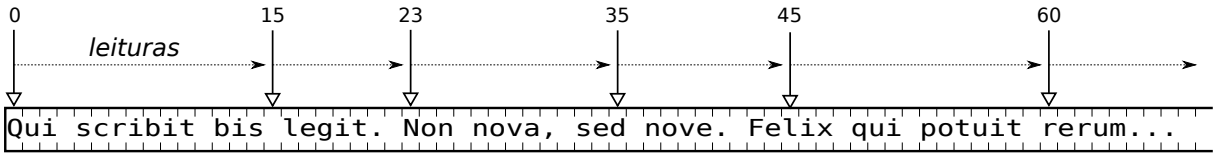

Figura 4: Leituras sequenciais em um arquivo de texto.

Por outro lado, no método de **acesso direto** (ou aleatório), pode-se indicar a posição no arquivo onde cada leitura ou escrita deve ocorrer, sem a necessidade de um ponteiro. Assim, caso se conheça previamente a posição de um determinado dado no arquivo, não há necessidade de percorrê-lo sequencialmente até encontrar o dado desejado. Essa forma de acesso é muito importante em gerenciadores de bancos de dados e aplicações congêneres, que precisam acessar rapidamente as posições do arquivo correspondentes ao registros desejados em uma operação.

Na prática, a maioria dos sistemas operacionais usa o acesso sequencial como modo básico de operação, mas oferece operações para mudar a posição do ponteiro do arquivo caso necessário, o que permite então o acesso direto a qualquer registro do arquivo. Nos sistemas POSIX, o reposicionamento do ponteiro do arquivo é efetuado através das chamadas `lseek` e `fseek`.

Uma forma particular de acesso direto ao conteúdo de um arquivo é o **mapeamento em memória** do mesmo, que faz uso dos mecanismos de memória virtual (paginação). Nessa modalidade de acesso, um arquivo é associado a um vetor de bytes (ou de registros) de mesmo tamanho na memória principal, de forma que cada posição do vetor corresponda à sua posição equivalente no arquivo. Quando uma posição específica do vetor ainda não acessada é lida, é gerada uma falta de página. Nesse momento, o mecanismo de paginação da memória virtual intercepta o acesso à memória, lê o conteúdo correspondente no arquivo e o deposita no vetor, de forma transparente à aplicação. Escritas no vetor são transferidas para o arquivo por um procedimento similar. Caso o arquivo seja muito grande, pode-se mapear em memória apenas partes dele. A Figura 5 ilustra essa forma de acesso.

Finalmente, alguns sistemas operacionais oferecem também a possibilidade de **acesso indexado** aos dados de um arquivo, como é o caso do *OpenVMS* [Rice, 2000]. Esse sistema implementa arquivos cuja estrutura interna pode ser vista como um conjunto de pares *chave/valor*. Os dados do arquivo são armazenados e recuperados de acordo com suas chaves correspondentes, como em um banco de dados relacional. Como o

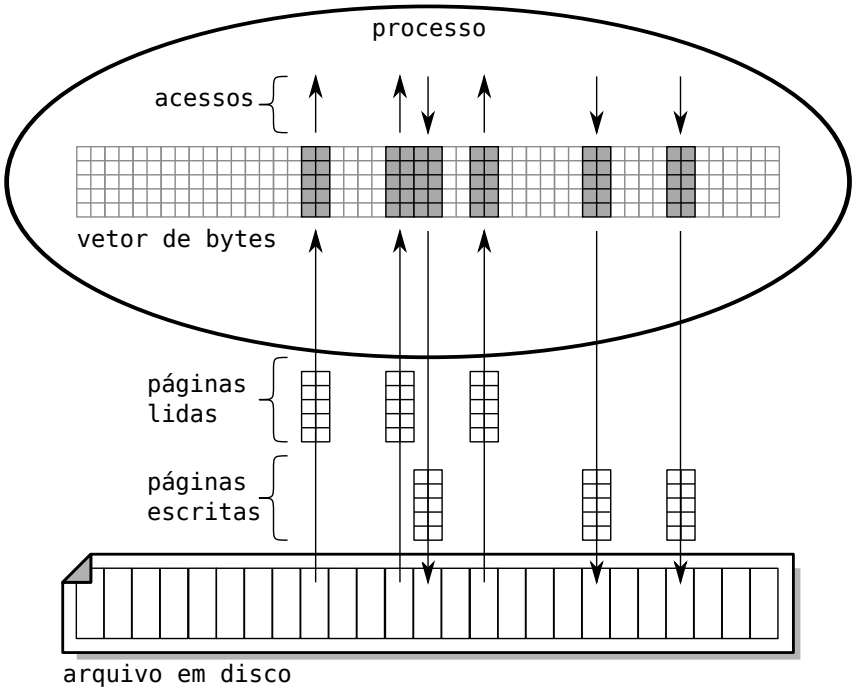

Figura 5: Arquivo mapeado em memória.

próprio núcleo do sistema implementa os mecanismos de acesso e indexação do arquivo, o armazenamento e busca de dados nesse tipo de arquivo costuma ser muito rápido, dispensando bancos de dados para a construção de aplicações mais simples.

2.3 Controle de acesso

Como arquivos são entidades que sobrevivem à existência do processo que as criou, é importante definir claramente o proprietário de cada arquivo e que operações ele e outros usuários do sistema podem efetuar sobre o mesmo. A forma mais usual de controle de acesso a arquivos consiste em associar os seguintes atributos a cada arquivo e diretório do sistema de arquivos:

- *Proprietário*: identifica o usuário dono do arquivo, geralmente aquele que o criou; muitos sistemas permitem definir também um *grupo proprietário* do arquivo, ou seja, um grupo de usuários com acesso diferenciado sobre o mesmo;
- *Permissões de acesso*: define que operações cada usuário do sistema pode efetuar sobre o arquivo.

Existem muitas formas de se definir permissões de acesso a recursos em um sistema computacional; no caso de arquivos, a mais difundida emprega listas de controle de acesso (ACL - *Access Control Lists*) associadas a cada arquivo. Uma lista de controle de acesso é basicamente uma lista indicando que usuários estão autorizados a acessar o arquivo, e como cada um pode acessá-lo. Um exemplo conceitual de listas de controle de acesso a arquivos seria:

```
1 arq1.txt  : (João: ler), (José: ler, escrever), (Maria: ler, remover)
2 video.avi : (José: ler), (Maria: ler)
3 musica.mp3: (Daniel: ler, escrever, apagar)
```

No entanto, essa abordagem se mostra pouco prática caso o sistema tenha muitos usuários e/ou arquivos, pois as listas podem ficar muito extensas e difíceis de gerenciar. O UNIX usa uma abordagem bem mais simplificada para controle de acesso, que considera basicamente três tipos de usuários e três tipos de permissões:

- Usuários: o proprietário do arquivo (*User*), um grupo de usuários associado ao arquivo (*Group*) e os demais usuários (*Others*).
- Permissões: ler (*Read*), escrever (*Write*) e executar (*eXecute*).

Dessa forma, no UNIX são necessários apenas 9 bits para definir as permissões de acesso a cada arquivo ou diretório. Por exemplo, considerando a seguinte listagem de diretório em um sistema UNIX (editada para facilitar sua leitura):

```
1 host:~> ls -l
2 d rwx --- --- 2 maziero prof    4096 2008-09-27 08:43 figuras
3 - rwx r-x --- 1 maziero prof    7248 2008-08-23 09:54 hello-unix
4 - rw- r-- r-- 1 maziero prof      54 2008-08-23 09:54 hello-unix.c
5 - rw- --- --- 1 maziero prof      59 2008-08-23 09:49 hello-windows.c
6 - rw- r-- r-- 1 maziero prof  195780 2008-09-26 22:08 main.pdf
7 - rw- --- --- 1 maziero prof   40494 2008-09-27 08:44 main.tex
```

Nessa listagem, o arquivo `hello-unix.c` (linha 4) pode ser acessado em leitura e escrita por seu proprietário (o usuário `maziero`, com permissões `rw-`), em leitura pelos usuários do grupo `prof` (permissões `r--`) e em leitura pelos demais usuários do sistema (permissões `r--`). Já o arquivo `hello-unix` (linha 3) pode ser acessado em leitura, escrita e execução por seu proprietário (permissões `rwx`), em leitura e execução pelos usuários do grupo `prof` (permissões `r-x`) e não pode ser acessado pelos demais usuários (permissões `---`). No caso de diretórios, a permissão de leitura autoriza a listagem do diretório, a permissão de escrita autoriza sua modificação (criação, remoção ou renomeação de arquivos ou subdiretórios) e a permissão de execução autoriza usar aquele diretório como diretório de trabalho ou parte de um caminho.

No mundo Windows, o sistema de arquivos NTFS implementa um controle de acesso bem mais flexível que o do UNIX, que define permissões aos proprietários de forma similar, mas no qual permissões complementares a usuários individuais podem ser associadas a qualquer arquivo. Mais detalhes sobre os modelos de controle de acesso usados nos sistemas UNIX e Windows podem ser encontrados no Capítulo **??**.

É importante destacar que o controle de acesso é normalmente realizado somente durante a abertura do arquivo, para a criação de sua referência em memória. Isso significa que, uma vez aberto um arquivo por um processo, este terá acesso ao arquivo enquanto o mantiver aberto, mesmo que as permissões do arquivo sejam alteradas para

impedir esse acesso. O controle contínuo de acesso aos arquivos é pouco frequentemente implementado em sistemas operacionais, porque verificar as permissões de acesso a cada operação de leitura ou escrita em um arquivo teria um impacto negativo significativo sobre o desempenho do sistema.

2.4 Compartilhamento de arquivos

Em um sistema multitarefas, é frequente ter arquivos acessados por mais de um processo, ou mesmo mais de um usuário, caso as permissões de acesso ao mesmo o permitam. Conforme estudado no Capítulo **??**, o acesso simultâneo a recursos compartilhados pode gerar condições de disputa (*race conditions*), que levam à inconsistência de dados e outros problemas. O acesso concorrente em leitura a um arquivo não acarreta problemas, mas a possibilidade de escritas e leituras simultâneas tem de ser prevista e tratada de forma adequada.

2.4.1 Travas em arquivos

A solução mais simples e mais frequentemente utilizada para gerenciar o acesso compartilhado a arquivos é o uso de travas de exclusão mútua (*mutex locks*), estudadas no Capítulo **??**. A maioria dos sistemas operacionais oferece algum mecanismo de sincronização para o acesso a arquivos, na forma de uma ou mais travas (*locks*) associadas a cada arquivo aberto. A sincronização pode ser feita sobre o arquivo inteiro ou sobre algum trecho específico dele, permitindo que dois ou mais processos possam trabalhar em partes distintas de um arquivo sem necessidade de sincronização entre eles.

As travas oferecidas pelo sistema operacional podem ser **obrigatórias** (*mandatory locks*) ou **recomendadas** (*advisory locks*). As travas obrigatórias são impostas pelo núcleo de forma incontornável: se um processo obtiver a trava do arquivo para si, outros processos que solicitarem acesso ao arquivo serão suspensos até que a respectiva trava seja liberada. Por outro lado, as travas recomendadas não são impostas pelo núcleo do sistema operacional. Neste caso, um processo pode acessar um arquivo mesmo sem ter sua trava. Caso sejam usadas travas recomendadas, cabe ao programador implementar os controles de trava necessários em suas aplicações, para impedir acessos conflitantes aos arquivos.

As travas sobre arquivos também podem ser **exclusivas** ou **compartilhadas**. Uma trava exclusiva, também chamada *trava de escrita*, garante acesso exclusivo ao arquivo: enquanto uma trava exclusiva estiver ativa, nenhum outro processo poderá obter uma trava sobre aquele arquivo. Já uma trava compartilhada (ou *trava de leitura*) impede outros processos de criar travas exclusivas sobre aquele arquivo, mas permite a existência de outras travas compartilhadas. Em conjunto, as travas exclusivas e compartilhadas implementam um modelo de sincronização *leitores/escritores* (descrito na Seção **??**), no qual os leitores acessam o arquivo usando travas compartilhadas e os escritores o fazem usando travas exclusivas.

É importante observar que normalmente as travas de arquivos são atribuídas a processos, portanto um processo só pode ter um tipo de trava sobre um mesmo arquivo. Além disso, todas as suas travas são liberadas quando o processo fecha o arquivo

ou encerra sua execução. No UNIX, a manipulação de travas em arquivos é feita através das chamadas de sistema `flock` e `fcntl`. Esse sistema oferece por default travas recomendadas exclusivas ou compartilhadas sobre arquivos ou trechos de arquivos. Sistemas Windows oferecem por default travas obrigatórias sobre arquivos, que podem ser exclusivas ou compartilhadas, ou travas recomendadas sobre trechos de arquivos.

2.4.2 Semântica de acesso

Quando um arquivo é aberto e usado por um único processo, o funcionamento das operações de leitura e escrita é simples e inequívoco: quando um dado é escrito no arquivo, ele está prontamente disponível para leitura se o processo desejar lê-lo novamente. No entanto, arquivos podem ser abertos por vários processos simultaneamente, e os dados escritos por um processo podem não estar prontamente disponíveis aos demais processos que leem aquele arquivo. Isso ocorre porque os discos rígidos são normalmente lentos, o que leva os sistemas operacionais a usar *buffers* intermediários para acumular os dados a escrever e assim otimizar o acesso aos discos. A forma como os dados escritos por um processo são percebidos pelos demais processos que abriram aquele arquivo é chamada de *semântica de compartilhamento*. Existem várias semânticas possíveis, mas as mais usuais são [Silberschatz et al., 2001]:

Semântica UNIX: toda modificação em um arquivo é imediatamente visível a todos os processos que mantêm aquele arquivo aberto; existe também a possibilidade de vários processos compartilharem o mesmo ponteiro de posicionamento do arquivo. Essa semântica é a mais comum em sistemas de arquivos locais, ou seja, para acesso a arquivos nos dispositivos locais;

Semântica de sessão: considera que cada processo usa um arquivo em uma sessão, que inicia com a abertura do arquivo e que termina com seu fechamento. Modificações em um arquivo feitas em uma sessão somente são visíveis na mesma seção e pelas sessões que iniciarem depois do encerramento da mesma, ou seja, depois que o processo fechar o arquivo; assim, sessões concorrentes de acesso a um arquivo compartilhado podem ver conteúdos distintos para o mesmo arquivo. Esta semântica é normalmente aplicada a sistemas de arquivos de rede, usados para acesso a arquivos em outros computadores;

Semântica imutável: de acordo com esta semântica, se um arquivo pode ser compartilhado por vários processos, ele é marcado como imutável, ou seja, seu conteúdo não pode ser modificado. É a forma mais simples de garantir a consistência do conteúdo do arquivo entre os processos que compartilham seu acesso, sendo por isso usada em alguns sistemas de arquivos distribuídos.

A Figura 6 traz um exemplo de funcionamento da semântica de sessão: os processos p_1 a p_4 compartilham o acesso ao mesmo arquivo, que contém apenas um número inteiro, com valor inicial 23. Pode-se perceber que o valor 39 escrito por p_1 é visto por ele na mesma sessão, mas não é visto por p_2, que abriu o arquivo antes do fim da sessão de p_1. O processo p_3 vê o valor 39, pois abriu o arquivo depois que p_1 o fechou, mas não vê o valor escrito por p_2. Da mesma forma, o valor 71 escrito por p_2 não é percebido por p_3, mas somente por p_4.

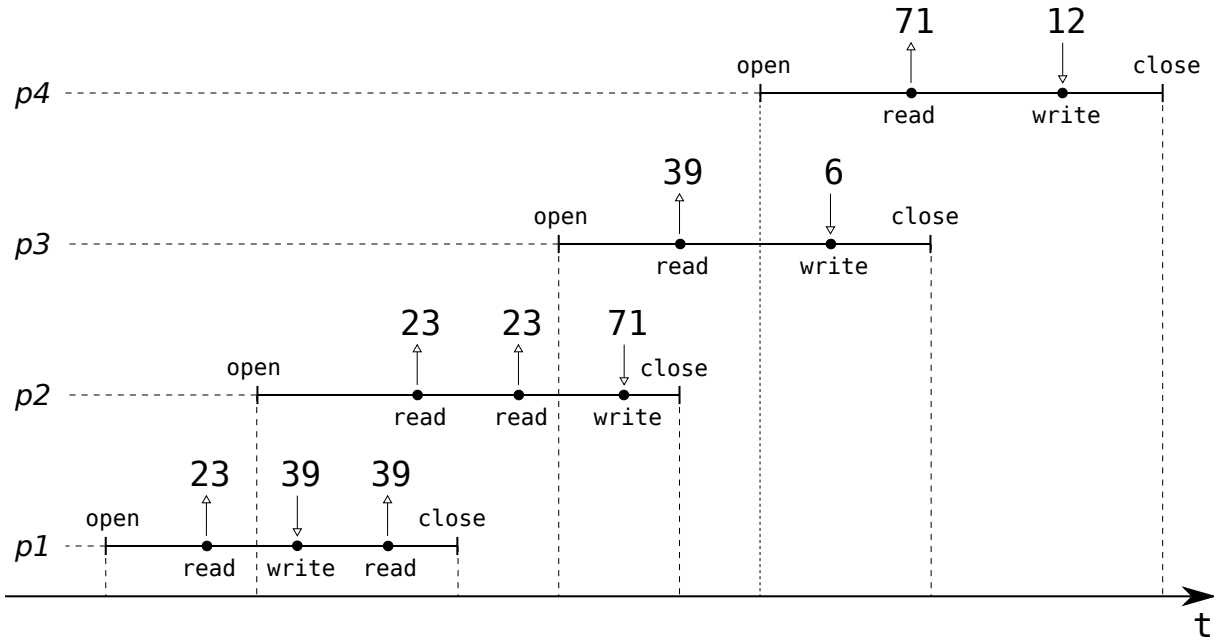

Figura 6: Compartilhamento de arquivo usando a semântica de sessão.

2.5 Exemplo de interface

Como visto na Seção 2.1, cada linguagem de programação define sua própria forma de representar arquivos abertos e as funções ou métodos usados para manipulá-los. A título de exemplo, será apresentada uma visão geral da interface para arquivos oferecida pela linguagem C no padrão ANSI [Kernighan and Ritchie, 1989]. Em C, cada arquivo aberto é representado por uma variável dinâmica do tipo `FILE*`, criada pela função `fopen`. As funções de acesso a arquivos são definidas na **Biblioteca Padrão de Entrada/Saída** (*Standard I/O Library*, definida no arquivo de cabeçalho `stdio.h`). As funções mais usuais dessa biblioteca são apresentadas a seguir:

- Abertura e fechamento de arquivos:
  - `FILE * fopen (const char *filename, const char *opentype)`: abre o arquivo cujo nome é indicado por `filename`; a forma de abertura (leitura, escrita, etc.) é indicada pelo parâmetro `opentype`; em caso de sucesso, devolve uma referência ao arquivo;
  - `int fclose (FILE *f)`: fecha o arquivo referenciado por `f`;
- Leitura e escrita de caracteres e *strings*:
  - `int fputc (int c, FILE *f)`: escreve um caractere no arquivo;
  - `int fgetc (FILE *f)`: lê um caractere do arquivo ;
- Reposicionamento do ponteiro do arquivo:

  - `long int ftell (FILE *f)`: indica a posição corrente do ponteiro do arquivo referenciado por `f`;
  - `int fseek (FILE *f, long int offset, int whence)`: move o ponteiro do arquivo para a posição indicada por `offset`;
  - `void rewind (FILE *f)`: retorna o ponteiro do arquivo à sua posição inicial;
  - `int feof (FILE *f)`: indica se o ponteiro chegou ao final do arquivo;

- Tratamento de travas:
  - `void flockfile (FILE *f)`: solicita acesso exclusivo ao arquivo, podendo bloquear o processo solicitante caso o arquivo já tenha sido reservado por outro processo;
  - `void funlockfile (FILE *f)`: libera o acesso ao arquivo.

O exemplo a seguir ilustra o uso de algumas dessas funções. Esse programa abre um arquivo chamado `numeros.dat` para operações de leitura (linha 9), verifica se a abertura do arquivo foi realizada corretamente (linhas 11 a 15), lê seus caracteres e os imprime na tela até encontrar o fim do arquivo (linhas 17 a 23) e finalmente o fecha (linha 25).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main (int argc, char *argv[], char* envp[])
5  {
6     FILE *arq ;
7     char c ;
8
9     arq = fopen ("infos.dat", "r") ; /* abertura do arquivo em leitura */
10
11    if (! arq) /* referencia de arquivo invalida */
12    {
13       perror ("Erro ao abrir arquivo") ;
14       exit (1) ;
15    }
16
17    while (1)
18    {
19       c = getc (arq) ; /* le um caractere do arquivo */
20       if (feof (arq))  /* chegou ao final do arquivo? */
21          break ;
22       putchar(c) ;     /* imprime o caractere na tela */
23    }
24
25    fclose (arq) ;      /* fecha o arquivo */
26    exit (0) ;
27 }
```

3 Organização de volumes

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 será discutida em detalhes no Capítulo **??**; por enquanto, um disco rígido pode ser visto basicamente como um grande vetor de blocos de bytes. Esses blocos de dados, também denominados *setores*, têm tamanho fixo geralmente entre 512 e 4.096 bytes e são 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*).

Em um computador no padrão PC, o espaço de armazenamento de cada dispositivo é dividido em uma pequena área inicial de configuração e uma ou mais *partições*, que podem ser vistas como espaços independentes. A área de configuração é denominada MBR - *Master Boot Record*, e contém uma *tabela de partições* com informações sobre o particionamento do dispositivo. Além disso, contém também um pequeno código executável, usado no processo de inicialização do sistema operacional. No início de cada partição geralmente há um bloco reservado, utilizado para a descrição do conteúdo daquela partição e para armazenar o código de lançamento do sistema operacional, se for uma partição inicializável (*bootable partition*). Esse bloco reservado é denominado *bloco de inicialização* ou VBR - *Volume Boot Record*. O restante dos blocos da partição está disponível para o armazenamento de arquivos. A Figura 7 ilustra a organização básica do espaço de armazenamento em um dispositivo de blocos típico: um disco rígido.

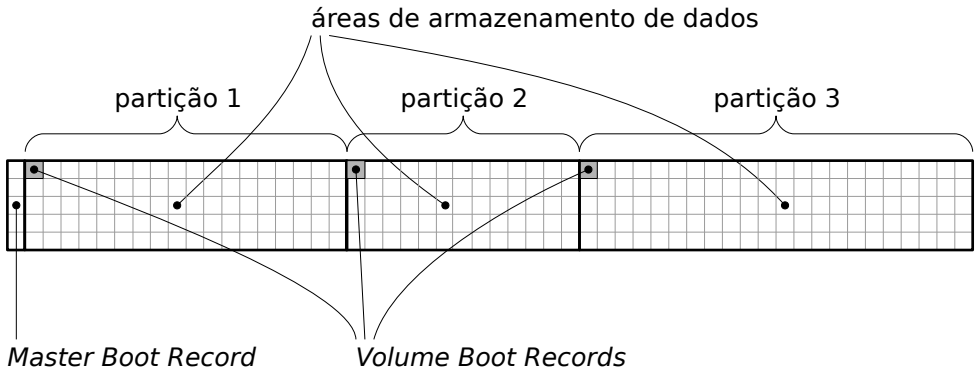

Figura 7: Organização em partições de um disco rígido típico.

Cada partição deve ser *formatada*, ou seja, estruturada para conter um sistema de arquivos, que pode conter arquivos, diretório, atalhos e outras entradas. Cada dispositivo ou partição devidamente preparado e formatado para receber um sistema de arquivos é designado como um *volume*.

3.1 Diretórios

A quantidade de arquivos em um sistema atual pode ser muito grande, chegando facilmente a milhões deles em um computador *desktop* típico, e muito mais em servidores.

Embora o sistema operacional possa tratar facilmente essa imensa quantidade de arquivos, essa tarefa não é tão simples para os usuários: identificar e localizar de forma inequívoca um arquivo específico em meio a milhões de outros arquivos pode ser impraticável.

Para permitir a organização de arquivos dentro de uma partição, são usados *diretórios*. Um diretório, também chamado de *pasta* (*folder*), representa um contêiner de informações, que pode conter arquivos ou mesmo outros diretórios. Da mesma forma que os arquivos, diretórios têm nome e atributos, que são usados na localização e acesso aos arquivos neles contidos.

Cada espaço de armazenamento possui ao menos um diretório principal, denominado *diretório raiz* (*root directory*). Em sistemas de arquivos mais antigos e simples, o diretório raiz de um volume estava definido em seus blocos de inicialização, normalmente reservados para informações de gerência. Todavia, como o número de blocos reservados era pequeno e fixo, o número de entradas no diretório raiz era limitado. Nos sistemas mais recentes, um registro específico dentro dos blocos de inicialização aponta para a posição do diretório raiz dentro do sistema de arquivos, permitindo que este tenha um número muito maior de entradas.

O uso de diretórios permite construir uma estrutura hierárquica (em árvore) de armazenamento dentro de um volume, sobre a qual os arquivos são distribuídos. A Figura 8 representa uma pequena parte da árvore de diretórios típica de um sistema Linux, cuja estrutura é definida nas normas *Filesystem Hierarchy Standard* [Russell et al., 2004].

Os primeiros sistemas de arquivos implementavam apenas o diretório raiz, que continha todos os arquivos do volume. Posteriormente, ofereceram subdiretórios, ou seja, um nível de diretórios abaixo do diretório raiz. Os sistemas atuais oferecem uma estrutura muito mais flexível, com um número de níveis de diretórios muito mais elevado, ou mesmo ilimitado (como no NTFS e no Ext3).

A implementação de diretórios é relativamente simples: um diretório é implementado como um arquivo estruturado, cujo conteúdo é uma relação de entradas. Os tipos de entradas normalmente considerados nessa relação são arquivos normais, diretórios, atalhos (vide Seção 3.3) e entradas associadas a arquivos especiais, como os discutidos na Seção 1.1. Cada entrada contém ao menos o nome do arquivo (ou do diretório), seu tipo e a localização física do mesmo no volume. Deve ficar claro que um diretório não contém fisicamente os arquivos e subdiretórios, ele apenas os relaciona.

Duas entradas padronizadas são usualmente definidas em cada diretório: a entrada "`.`" (ponto), que representa o próprio diretório, e a entrada "`..`" (ponto-ponto), que representa seu diretório pai (o diretório imediatamente acima dele na hierarquia de diretórios). No caso do diretório raiz, ambas as entradas apontam para ele próprio.

A Figura 9 apresenta uma possibilidade de implementação de parte da estrutura de diretórios apresentada na Figura 8. Os tipos das entradas em cada diretório são: "`A`" para arquivos normais e "`D`" para diretórios.

A relação de entradas em um diretório, também chamada de *índice do diretório*, pode ser implementada como uma lista linear, como no caso do MS-DOS e do Ext2 (Linux) ou como algum tipo de tabela *hash* ou árvore, o que é feito no NTFS e no Ext3, entre outros. A implementação em lista linear é mais simples, mas tem baixo desempenho. A implementação em tabela *hash* ou árvore provê um melhor desempenho quando

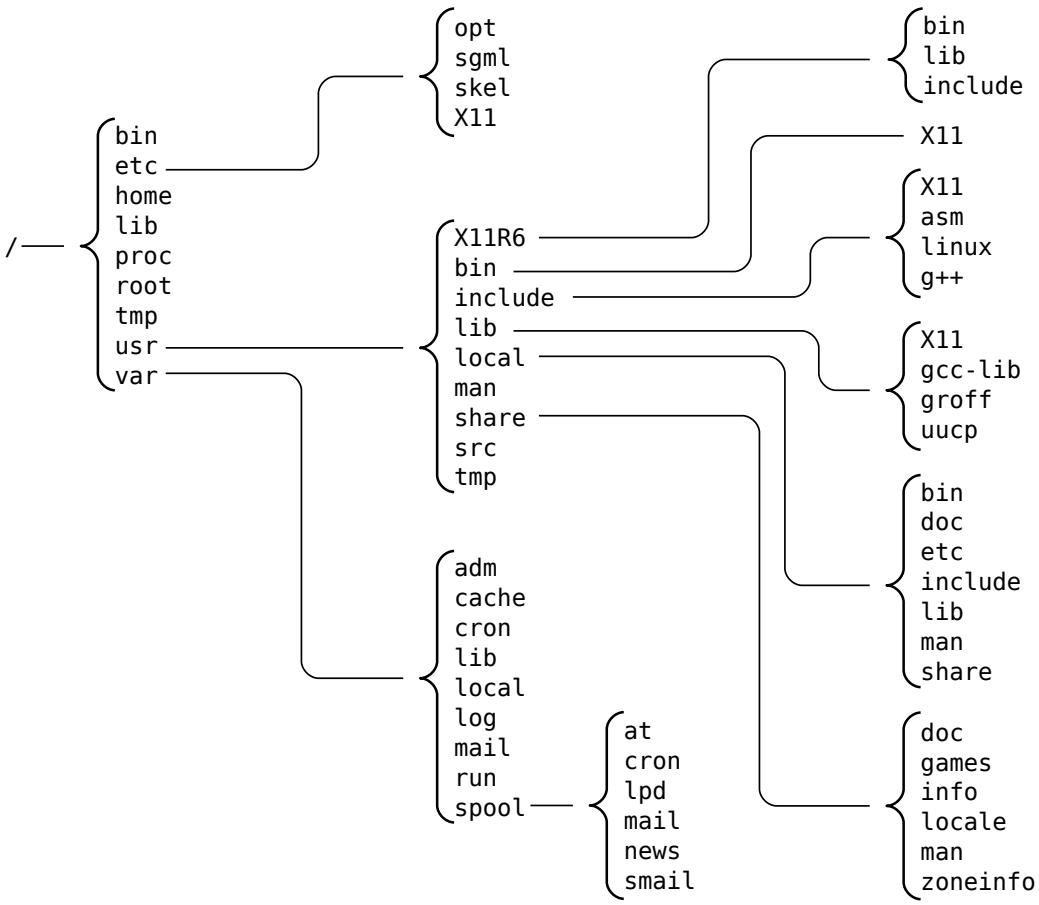

Figura 8: Estrutura de diretórios típica de um sistema Linux.

é necessário percorrer a estrutura de diretórios em busca de arquivos, o que ocorre frequentemente.

3.2 Caminhos de acesso

Em um sistema de arquivos, os arquivos estão dispersos ao longo da hierarquia de diretórios. Para poder abrir e acessar um arquivo, torna-se então necessário conhecer sua localização completa, ao invés de somente seu nome. A posição de um arquivo dentro do sistema de arquivos é chamada de *caminho de acesso* ao arquivo. Normalmente, o caminho de acesso a um arquivo é composto pela sequência de nomes de diretórios que levam até ele, separadas por um caractere específico. Por exemplo, o sistema Windows usa como separador o caractere "\", enquanto sistemas UNIX usam o caractere "/"; outros sistemas podem usar caracteres como "`:`" e "`!`". Exemplos de caminhos de acesso a arquivos seriam `\Windows\system32\ole32.dll` (no Windows) e `/usr/bin/bash` (em sistemas UNIX).

A maioria dos sistemas implementa o conceito de *diretório de trabalho* ou diretório corrente de um processo (*working directory*). Ao ser criado, cada novo processo recebe um diretório de trabalho, que será usado por ele como local default para criar novos arquivos ou abrir arquivos existentes, quando não informar os respectivos caminhos de

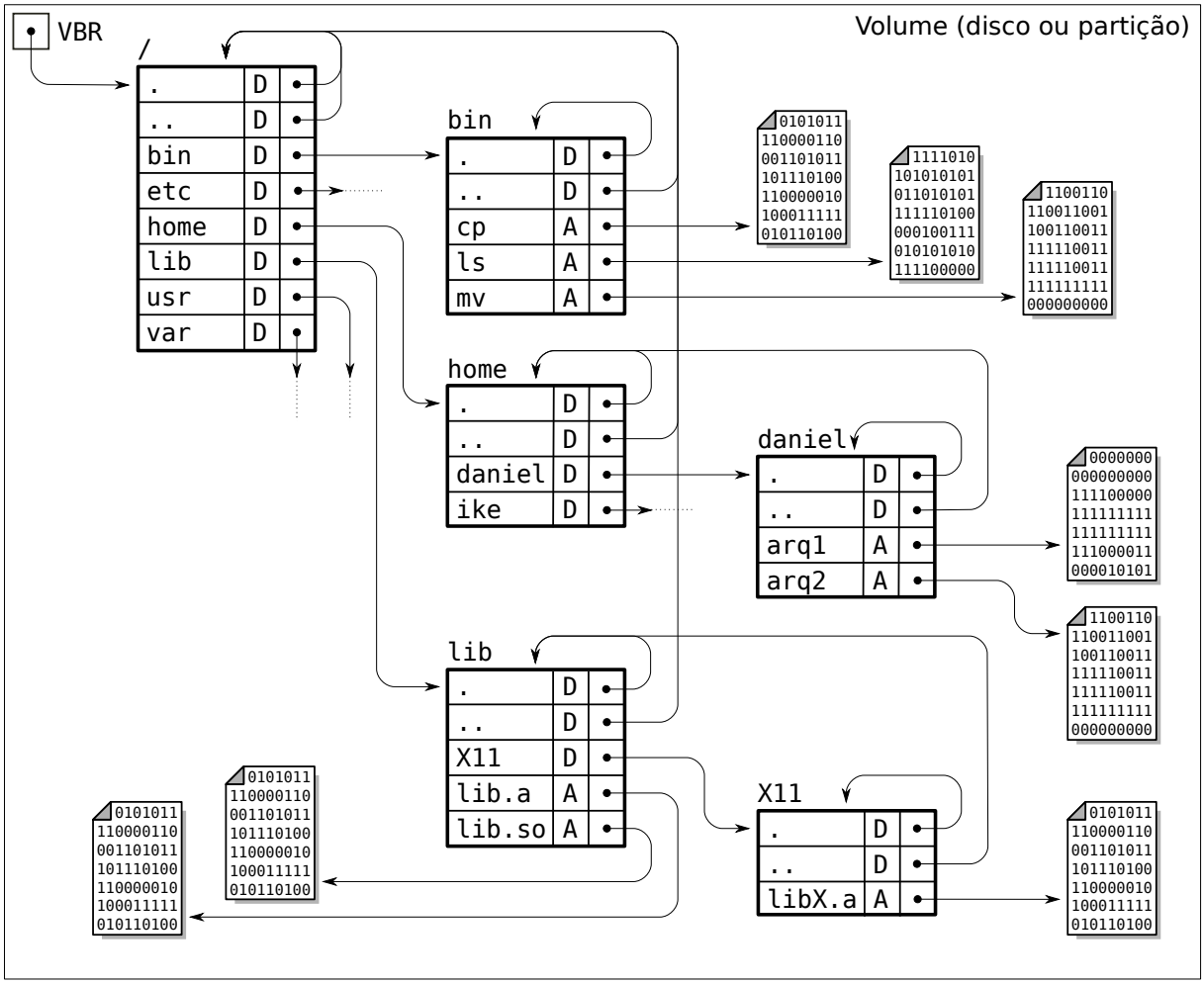

Figura 9: Implementação de uma estrutura de diretórios.

acesso. Cada processo geralmente herda o diretório de trabalho de seu pai, mas pode mudar de diretório através de chamadas de sistema (como `chdir` nos sistemas UNIX).

Existem basicamente três formas de se referenciar arquivos em um sistema de arquivos:

Referência direta: somente o nome do arquivo é informado; neste caso, considera-se que o arquivo está (ou será criado) no diretório de trabalho do processo. Exemplos:

```
1 prova1.doc
2 materiais.pdf
3 uma-bela-foto.jpg
```

Referência absoluta: o caminho de acesso ao arquivo é indicado a partir do diretório raiz do sistema de arquivos, e não depende do diretório de trabalho do processo; uma referência absoluta a um arquivo sempre inicia com o caractere separador, indicando que o nome do arquivo está referenciado a partir do diretório raiz do sistema de arquivos. O caminho de acesso mais curto a um arquivo a partir

do diretório raiz é denominado *caminho canônico* do arquivo. Nos exemplos de referências absolutas a seguir, os dois primeiros são caminhos canônicos, enquanto os dois últimos não o são:

```
\Windows\system32\drivers\etc\hosts.lm
/usr/local/share/fortunes/brasil.dat
\Documents and Settings\Carlos Maziero\..\All Users\notas.xls
/home/maziero/bin/scripts/../../docs/proj1.pdf
```

Referência relativa: o caminho de acesso ao arquivo tem como início o diretório de trabalho do processo, e indica subdiretórios ou diretórios anteriores, através de referências "`..`"; eis alguns exemplos:

```
imagens\satelite\brasil\geral.jpg
..\users\maziero\documentos\prova-2.doc
public_html/static/fotografias/rennes.jpg
../../../share/icons/128x128/calculator.svg
```

Durante a abertura de um arquivo, o sistema operacional deve encontrar a localização do mesmo no dispositivo de armazenamento, a partir do nome e caminho informados pelo processo. Para isso, é necessário percorrer as estruturas definidas pelo caminho do arquivo até encontrar sua localização, em um procedimento denominado *localização de arquivo* (*file lookup*). Por exemplo, para abrir o arquivo `/usr/lib/X11/libX.a` da Figura 9 seria necessário executar os seguintes passos[3]:

1. Acessar o disco para ler o VBR (*Volume Boot Record*) do volume;
2. Nos dados lidos, descobrir onde se encontra o diretório raiz (`/`) daquele sistema de arquivos;
3. Acessar o disco para ler o diretório raiz;
4. Nos dados lidos, descobrir onde se encontra o diretório `usr`;
5. Acessar o disco para ler o diretório `usr`;
6. Nos dados lidos, descobrir onde se encontra o diretório `lib`;
7. Acessar o disco para ler o diretório `lib`;
8. Nos dados lidos, descobrir onde se encontra o diretório `X11`;
9. Acessar o disco para ler o diretório `X11`;
10. Nos dados lidos, descobrir onde se encontra o arquivo `libX11.a`;

[3]Para simplificar, foram omitidas as verificações de existência de entradas, de permissões de acesso e os tratamentos de erro.

11. Acessar o disco para ler o bloco de controle do arquivo `libX11.a`, que contém seus atributos;
12. Criar as estruturas em memória que representam o arquivo aberto;
13. Retornar uma referência ao arquivo para o processo solicitante.

Pode-se perceber que a localização de arquivo é um procedimento trabalhoso. Neste exemplo, foram necessárias 5 leituras no disco (passos 1, 3, 5, 7 e 9) apenas para localizar a posição do bloco de controle do arquivo desejado no disco. Assim, o tempo necessário para localizar um arquivo pode ser muito elevado, pois discos rígidos são dispositivos lentos. Para evitar esse custo e melhorar o desempenho do mecanismo de localização de arquivos, é mantido em memória um cache de entradas de diretório localizadas recentemente, gerenciado de acordo com uma política LRU (*Least Recently Used*). Cada entrada desse cache contém um nome de arquivo ou diretório e sua localização no dispositivo físico. Esse cache geralmente é organizado na forma de uma tabela *hash*, o que permite localizar rapidamente os arquivos ou diretórios recentemente utilizados.

3.3 Atalhos

Em algumas ocasiões, pode ser necessário ter um mesmo arquivo ou diretório replicado em várias posições dentro do sistema de arquivos. Isso ocorre frequentemente com arquivos de configuração de programas e arquivos de bibliotecas, por exemplo. Nestes casos, seria mais econômico armazenar apenas uma instância dos dados do arquivo no sistema de arquivos e criar referências indiretas (ponteiros) para essa instância, para representar as demais cópias do arquivo. O mesmo raciocínio pode ser aplicado a diretórios duplicados. Essas referências indiretas a arquivos ou diretórios são denominadas *atalhos* (*links*).

Existem basicamente duas abordagens para a construção de atalhos:

Atalhos simbólicos (*soft links*): cada "cópia" do arquivo original é, na verdade, um pequeno arquivo de texto contendo uma *string* que indica o caminho até o arquivo original (pode ser usado um caminho simples, absoluto ou relativo à posição do atalho). Como o caminho ao arquivo original é indicado de forma simbólica, este pode estar localizado em outro dispositivo físico (outro disco ou uma unidade de rede). O arquivo original e seus atalhos simbólicos são totalmente independentes: caso o arquivo original seja movido, renomeado ou removido, os atalhos simbólicos apontarão para um arquivo inexistente; neste caso, diz-se que aqueles atalhos estão "quebrados" (*broken links*).

Atalhos físicos (*hard links*): várias referências do arquivo no sistema de arquivos apontam para a mesma localização do dispositivo físico onde o conteúdo do arquivo está de fato armazenado. Normalmente é mantido um contador de referências a esse conteúdo, indicando quantos atalhos físicos apontam para o mesmo: somente quando o número de referências ao arquivo for zero, aquele conteúdo poderá ser removido do dispositivo. Como são usadas referências à posição do arquivo no dispositivo, atalhos físicos só podem ser feitos para arquivos dentro do mesmo sistema de arquivos (o mesmo volume).

A Figura 10 traz exemplos de implementação de atalhos simbólicos e físicos a arquivos em um sistema de arquivos UNIX. As entradas de diretórios indicadas como "L" correspondem a atalhos simbólicos (de *links*). Nessa figura, pode-se constatar que as entradas `/bin/ls` e `/usr/bin/dir` são atalhos físicos para o mesmo conteúdo no disco, enquanto a entrada `/bin/shell` é um atalho simbólico para o arquivo `/usr/bin/sh` e `/lib` é um atalho simbólico para o diretório `/usr/lib`.

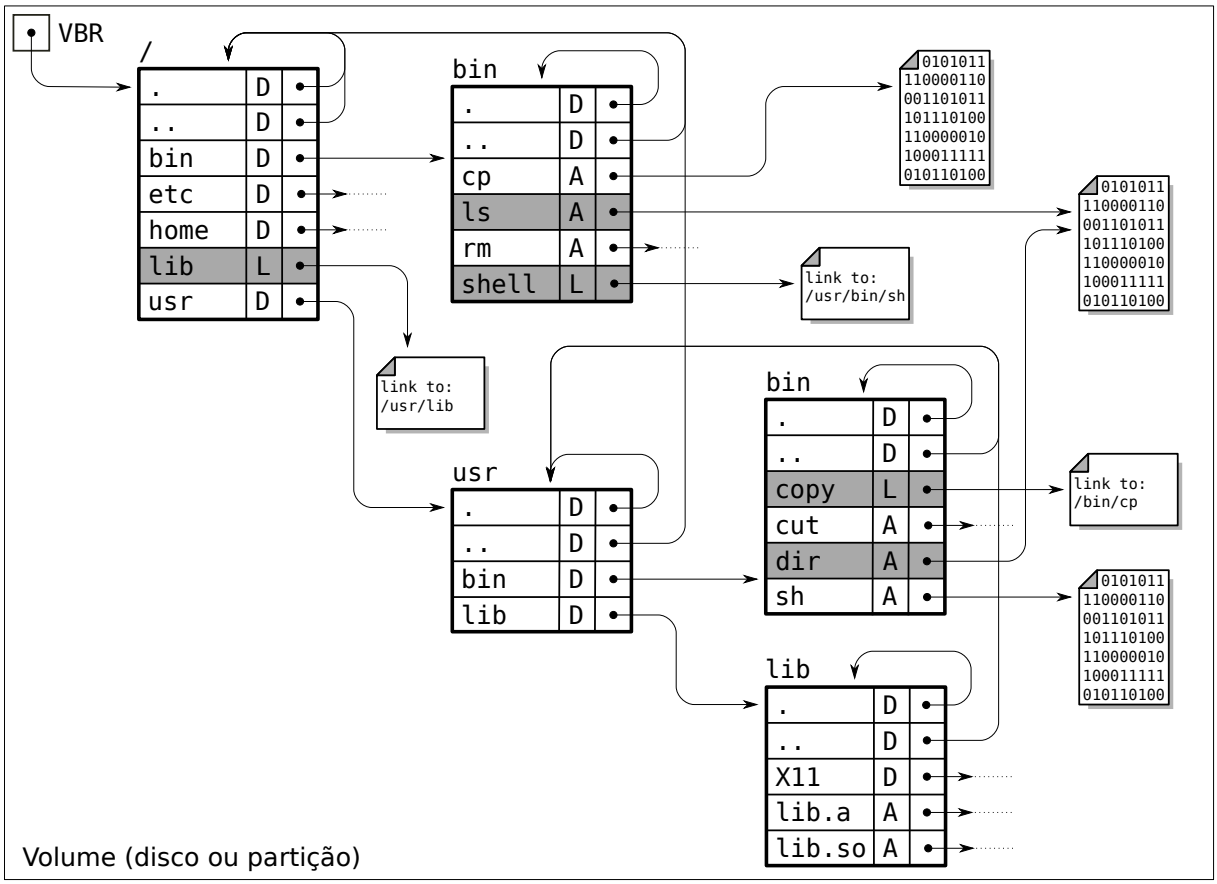

Figura 10: Atalhos simbólicos e físicos a arquivos em UNIX.

Sistemas UNIX suportam atalhos físicos e simbólicos, com algumas limitações: atalhos físicos geralmente só podem ser feitos para arquivos dentro do mesmo sistema de arquivos (mesmo volume) e não são permitidos atalhos físicos para diretórios[4]. Em ambientes Windows, o sistema de arquivos NTFS suporta ambos os tipos de atalhos (embora atalhos simbólicos só tenham sido introduzidos no Windows Vista), com limitações similares.

3.4 Montagem de volumes

Para que o sistema operacional possa acessar o sistema de arquivos presente em um determinado volume, ele deve ler os dados presentes em seu bloco de inicialização, que

[4] Atalhos físicos para diretórios geralmente são proibidos porque permitiriam diretórios recursivos, tornando muito complexa a implementação de rotinas de verificação e gerência do sistema de arquivos.

descrevem o tipo de sistema de arquivos presente, e criar as estruturas em memória que representam esse volume dentro do núcleo. 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 seus 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.

A montagem é um procedimento frequente no caso de mídias móveis, 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 11 apresenta um exemplo de montagem de volumes em plataformas UNIX. Nessa figura, o disco rígido 1 contém o sistema operacional e foi montado como raiz da árvore de diretórios durante a inicialização do sistema. O disco rígido 2 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 uma mídia removível (CD-ROM), com sua árvore de diretórios própria.

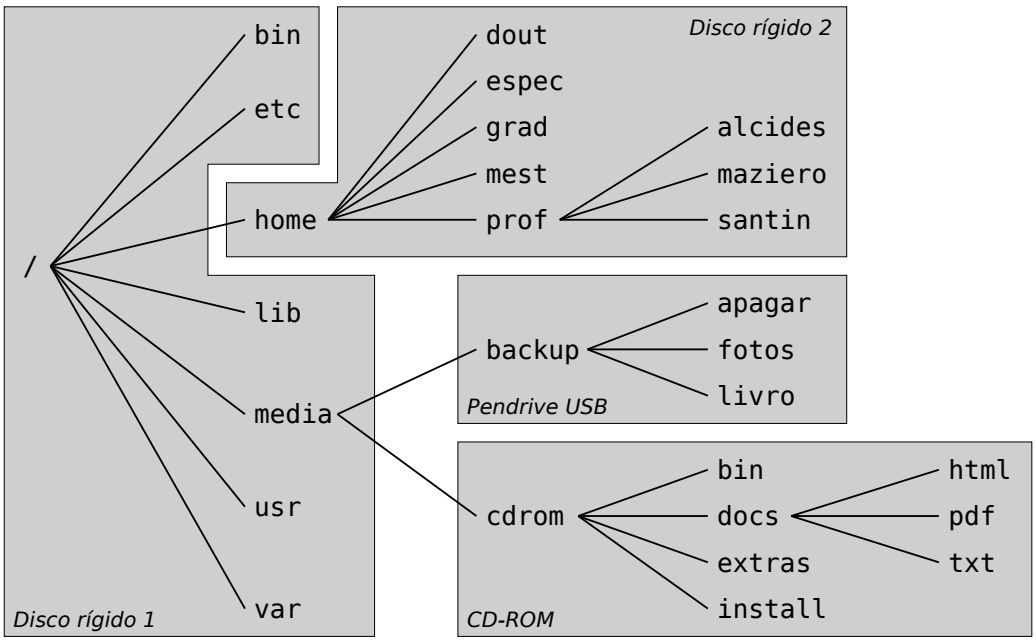

Figura 11: Montagem de volumes em UNIX.

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 mesma forma que o UNIX.

4 Sistemas de arquivos

Vários problemas importantes devem ser resolvidos na construção de um sistema de arquivos, que vão do acesso de baixo nível aos dispositivos físicos de armazenamento à implementação da interface de acesso a arquivos para os programadores. 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.

Nesta seção serão discutidos os principais elementos que compõem a gerência de arquivos em um sistema operacional típico.

4.1 Arquitetura geral

Os principais elementos que constituem a gerência de arquivos estão organizados em camadas, conforme apresentado na Figura 12. No nível mais baixo dessa arquitetura estão os **dispositivos de armazenamento**, como discos rígidos ou bancos de memória *flash*, responsáveis pelo armazenamento dos dados e metadados dos arquivos. Esses dispositivos são acessados através de **controladores**, que são circuitos eletrônicos dedicados ao controle e interface dos dispositivos. A interface entre controladores e dispositivos de armazenamento segue padrões como SATA, ATAPI, SCSI, USB e outros.

Os controladores de dispositivos são configurados e acessados pelo núcleo do sistema operacional através de **drivers de dispositivos**, que são componentes de software capazes de interagir com os controladores. Os *drivers* usam portas de entrada/saída, interrupções e canais de acesso direto à memória (DMA) para interagir com os controladores e realizar as operações de controle e de entrada/saída de dados. Como cada controlador define sua própria interface, também possui um *driver* específico. Os *drivers* ocultam essas interfaces e fornecem às camadas superiores do núcleo uma interface padronizada para acesso aos dispositivos de armazenamento. Desta forma, os detalhes tecnológicos e particularidades de cada dispositivo são isolados, tornando o restante do sistema operacional independente da tecnologia subjacente.

Acima dos *drivers* está a camada de **gerência de blocos**, que gerencia o fluxo de blocos de dados entre a memória e os dispositivos de armazenamento. É importante lembrar que os discos são dispositivos orientados a blocos, ou seja, 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, oferecer às camadas superiores a

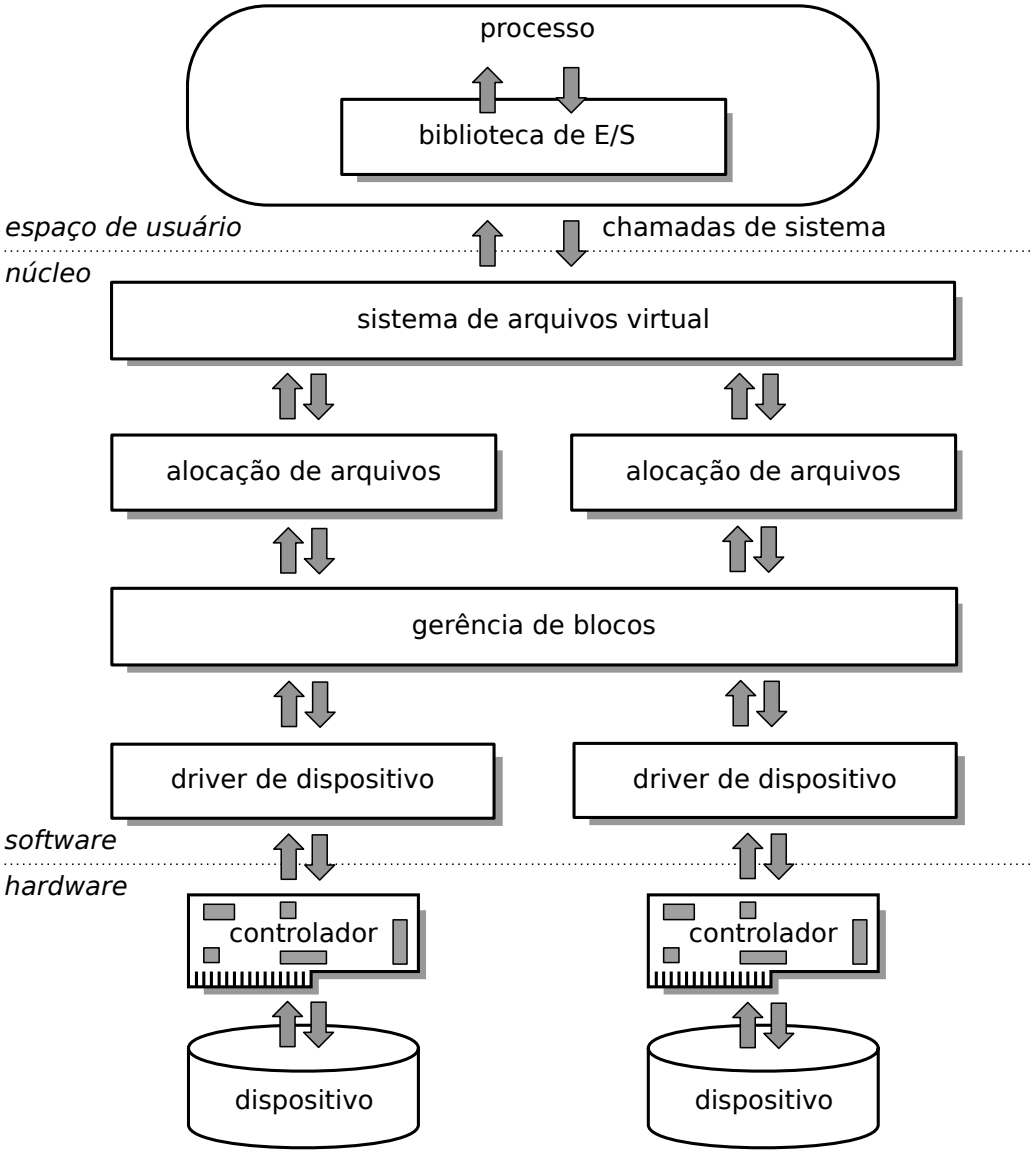

Figura 12: Camadas da implementação da gerência de arquivos.

abstração de cada dispositivo físico como sendo um imenso vetor de blocos lógicos, independente de sua configuração real, e também efetuar o *caching/buffering* de blocos (Seção **??**).

A seguir está a camada de **alocação de arquivos**, que tem como função principal alocar os arquivos sobre os blocos lógicos oferecidos pela 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 4.3.

Acima da alocação de arquivos está o **sistema de arquivos virtual** (VFS - *Virtual File System*), que provê uma interface de acesso a arquivos independente dos dispositivos físicos e das estratégias de alocação de arquivos empregadas pelas camadas inferiores. O sistema de arquivos virtual normalmente gerencia as permissões associadas aos arquivos

e as travas de acesso compartilhado, além de construir as abstrações de diretórios e atalhos. Outra responsabilidade importante desta camada é manter informações sobre 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. A interface de acesso ao sistema de arquivos virtual é oferecida aos processos através de um conjunto de **chamadas de sistema**.

Finalmente, as **bibliotecas de entrada/saída** usam as chamadas de sistema oferecidas pelo sistema operacional para construir funções padronizadas de acesso a arquivos para cada linguagem de programação, como aquelas apresentadas na Seção 2.5 para a linguagem C ANSI.

4.2 Blocos físicos e lógicos

Um dos aspectos mais importantes dos sistemas de arquivos é a forma como o conteúdo dos arquivos é disposto dentro do disco rígido ou outro dispositivo de armazenamento secundário. Conforme visto na Seção 3, um disco rígido pode ser visto como um conjunto de blocos de tamanho fixo (geralmente de 512 bytes). Os blocos do disco rígido são normalmente denominados *blocos físicos*. Como esses blocos são pequenos, o número de blocos físicos em um disco rígido recente pode ser imenso: um disco rígido de 250 GBytes contém mais de 500 milhões de blocos físicos! Para simplificar a gerência dessa quantidade de blocos físicos e melhorar o desempenho das operações de leitura/escrita, os sistemas operacionais costumam trabalhar com *blocos lógicos* ou *clusters*, que são grupos de 2^n blocos físicos consecutivos. Blocos lógicos com 4K, 8K, 16K e 32K bytes são frequentemente usados. 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 de uma partição é 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 muito pequenos implicam em ter mais blocos a gerenciar e menos bytes transferidos em cada operação de leitura/escrita, o que tem impacto negativo sobre o desempenho do sistema. Por outro lado, blocos lógicos muito grandes podem levar à *fragmentação interna*: um arquivo com 200 bytes armazenado em um sistema de arquivos com blocos lógicos de 32.768 bytes (32K) ocupará um bloco lógico, do qual 32.568 bytes serão desperdiçados, pois ficarão alocados ao arquivo sem serem usados. A fragmentação interna diminui o espaço útil do disco rígido, por isso deve ser evitada. Uma forma de evitá-la é escolher um tamanho de bloco lógico adequado ao tamanho médio dos arquivos a armazenar no disco, ao formatá-lo. Além disso, 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].

4.3 Alocação física de arquivos

Um dispositivo de armazenamento é visto pelas camadas superiores 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 lógicos. Como os blocos lógicos são pequenos, cada arquivo poderá precisar de muitos blocos lógicos para ser armazenado no disco (Figura 13). Os dados e metadados de um arquivo devem estar dispostos nesses blocos de forma a permitir um acesso rápido e confiável. Como um arquivo pode ocupar milhares ou mesmo milhões de blocos, a forma de alocação dos arquivos nos blocos do disco tem um impacto importante sobre o desempenho e a robustez do sistema de arquivos.

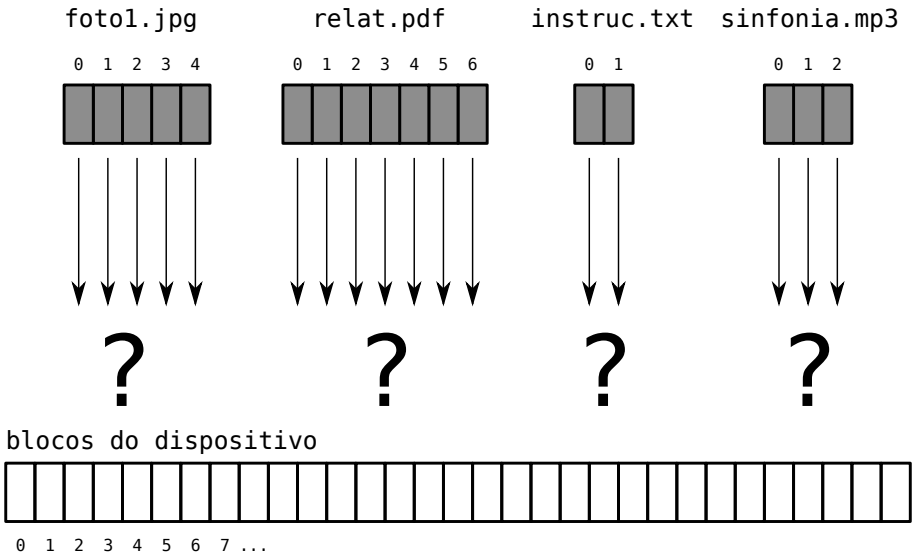

Figura 13: O problema da alocação de arquivos.

A alocação de um arquivo no disco tem como ponto de partida a definição de um **bloco de controle de arquivo** (FCB - *File Control Block*), que nada mais é que uma estrutura contendo os metadados do arquivo e a localização de seu conteúdo no disco. Em alguns sistemas de arquivos mais simples, como o sistema FAT (*File Alocation Table*) usado em plataformas MS-DOS, o FCB é bastante pequeno e cabe na entrada correspondente ao arquivo, na tabela de diretório onde ele se encontra definido. Em sistemas de arquivos mais complexos, os blocos de controle de arquivos são definidos em estruturas separadas, como a *Master File Table* do sistema NTFS e os *i-nodes* dos sistemas UNIX.

Há três estratégias usuais de alocação de arquivos nos blocos lógicos do disco, que serão apresentadas a seguir: as alocações **contígua**, **encadeada** e **indexada**. Como diretórios são usualmente implementados na forma de arquivos, as estratégias de alocação discutidas aqui são válidas também para a alocação de diretórios. 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 diretos; 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 e diretórios.

4.3.1 Alocação contígua

Na alocação contígua, os dados do arquivo são dispostos de forma ordenada 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 14 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).

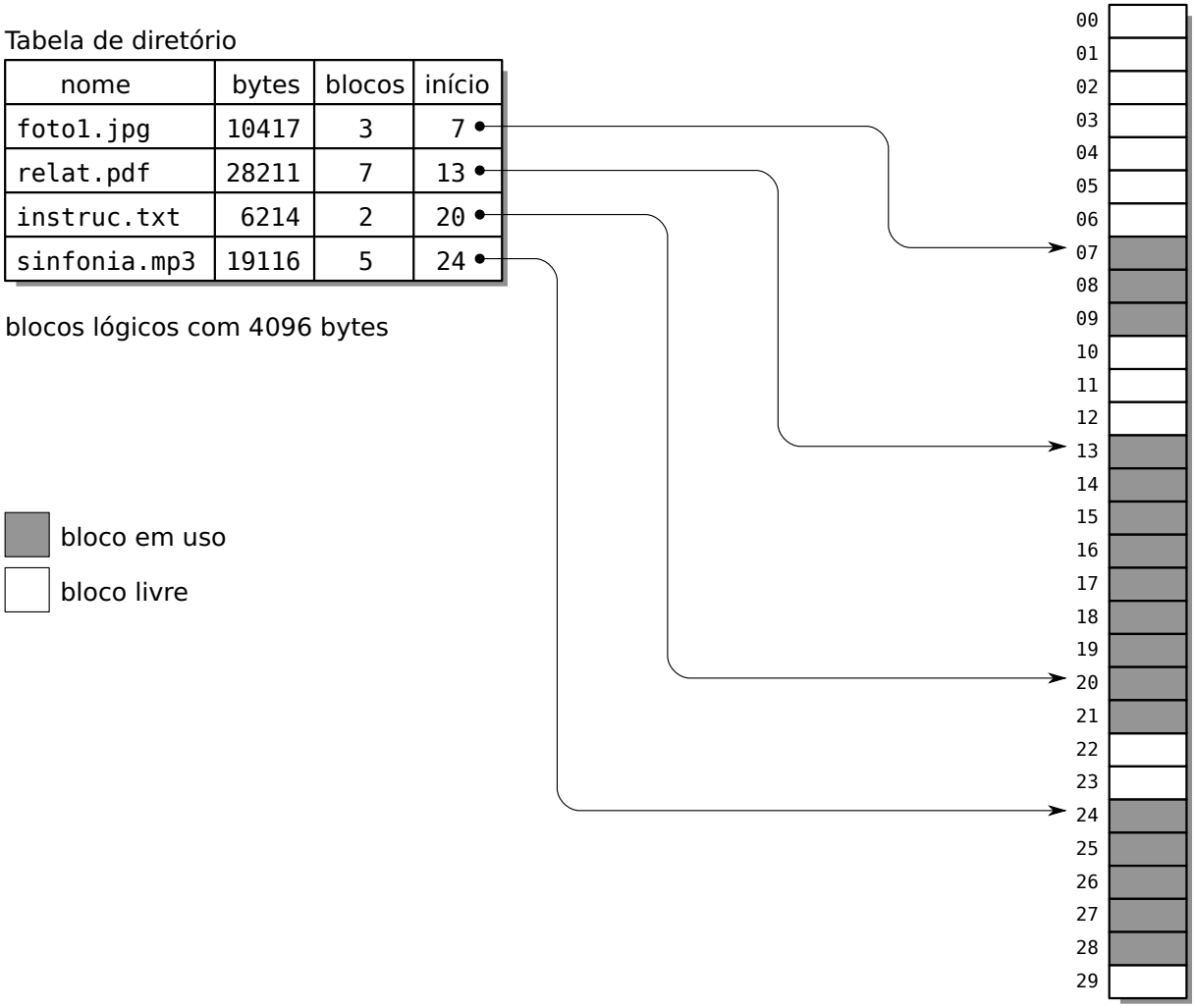

Figura 14: Estratégia de alocação contígua.

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 direto a posições específicas do arquivo 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 1. De acordo com esse algoritmo, o byte de número 14.372 do arquivo `relat.pdf` da Figura 14 estará na posição 2.084 do bloco 16 do disco rígido.

Algoritmo 1 Localizar a posição do i-ésimo byte do arquivo no disco

i: número do byte a localizar
B: tamanho dos blocos lógicos, em bytes
b_0: número do bloco do disco onde o arquivo inicia
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)

Esta estratégia apresenta uma boa robustez a falhas de disco: caso um bloco do disco apresente defeito e não permita a leitura de seus dados, apenas o conteúdo daquele bloco é perdido: o conteúdo do arquivo nos blocos anteriores e posteriores ao bloco defeituoso ainda poderão ser acessados sem dificuldades. Por outro lado, o ponto fraco desta estratégia é sua baixa flexibilidade, pois o tamanho final de cada arquivo precisa ser conhecido no momento de sua criação. Além disso, esta estratégia está sujeita à fragmentação externa, de forma similar à técnica de alocação contígua estudada nos mecanismos de alocação de memória (vide Seção **??**): à medida em que arquivos são criados e destruídos, as áreas livres do disco vão sendo fracionadas em pequenas áreas isoladas (os fragmentos) que diminuem a capacidade de alocação de arquivos maiores. Por exemplo, na situação da Figura 14 há 13 blocos livres no disco, mas somente podem ser criados arquivos com até 7 blocos de tamanho. As técnicas de alocação *first/best/worst-fit* utilizadas em gerência de memória também podem ser aplicadas para 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 os 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 diretos aos dados. Um exemplo dessa situação são sistemas dedicados para reprodução de dados multimídia, como áudio e vídeo.

4.3.2 Alocação encadeada

Esta forma de alocação foi proposta para contornar a pouca flexibilidade da alocação contígua e eliminar a fragmentação externa. Nela, cada bloco do arquivo no disco

contém dados do arquivo e também um ponteiro para o próximo bloco, ou seja, um campo indicando o número do próximo bloco do arquivo no disco. 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 são utilizáveis sem restrições, e permite que arquivos sejam criados sem a necessidade de definir seu tamanho final. A Figura 15 ilustra um exemplo dessa abordagem.

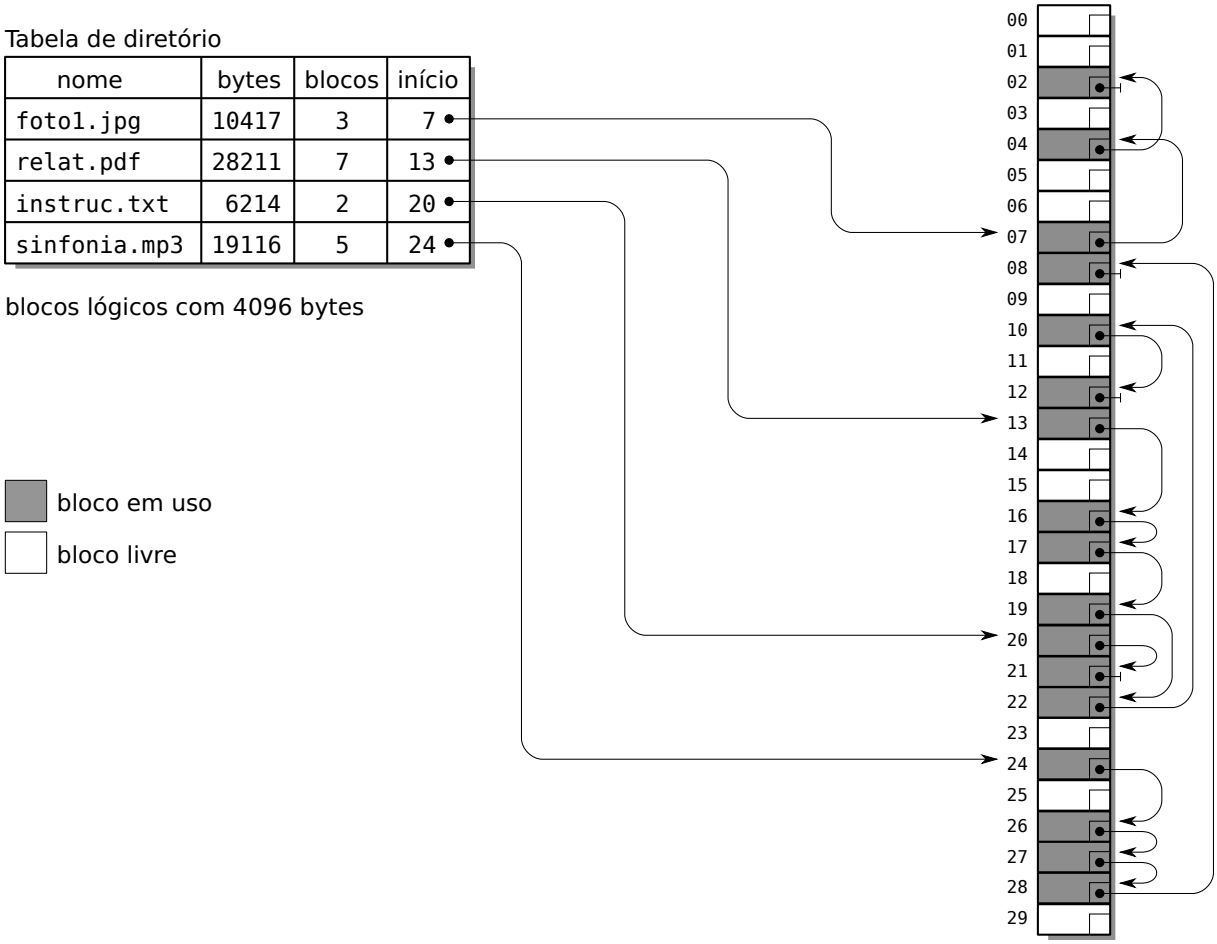

Figura 15: Estratégia de alocação encadeada.

Nesta abordagem, o acesso sequencial aos dados do arquivo é simples e rápido, pois cada bloco contém o ponteiro do próximo bloco do arquivo. 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 direto a posições específicas do arquivo fica muito prejudicado com esta abordagem: caso se deseje acessar um bloco no meio do arquivo, todos os blocos anteriores terão de ser lidos em sequência, para poder seguir os ponteiros que levam ao bloco desejado. O algoritmo 2 mostra claramente esse problema, indicado através do laço *while*. Essa dependência dos blocos anteriores 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

Algoritmo 2 Localizar a posição do i-ésimo byte do arquivo no disco

i: número do byte a localizar
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
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*)

// define bloco inicial do percurso
$b_{aux} = b_0$
// calcula número de blocos a percorrer
$b = i \div (B - P)$
while $b > 0$ **do**
  $block$ = *read_block* (b_{aux})
  b_{aux} = ponteiro extraído de $block$
  $b = b - 1$
end while
$b_i = b_{aux}$
$o_i = i \bmod (B - P)$
return (b_i, o_i)

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.

Os principais problemas da alocação encadeada são o baixo desempenho nos acessos diretos 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 da 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 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 16 apresenta o conteúdo da tabela de alocação de arquivos para o exemplo apresentado anteriormente na Figura 15.

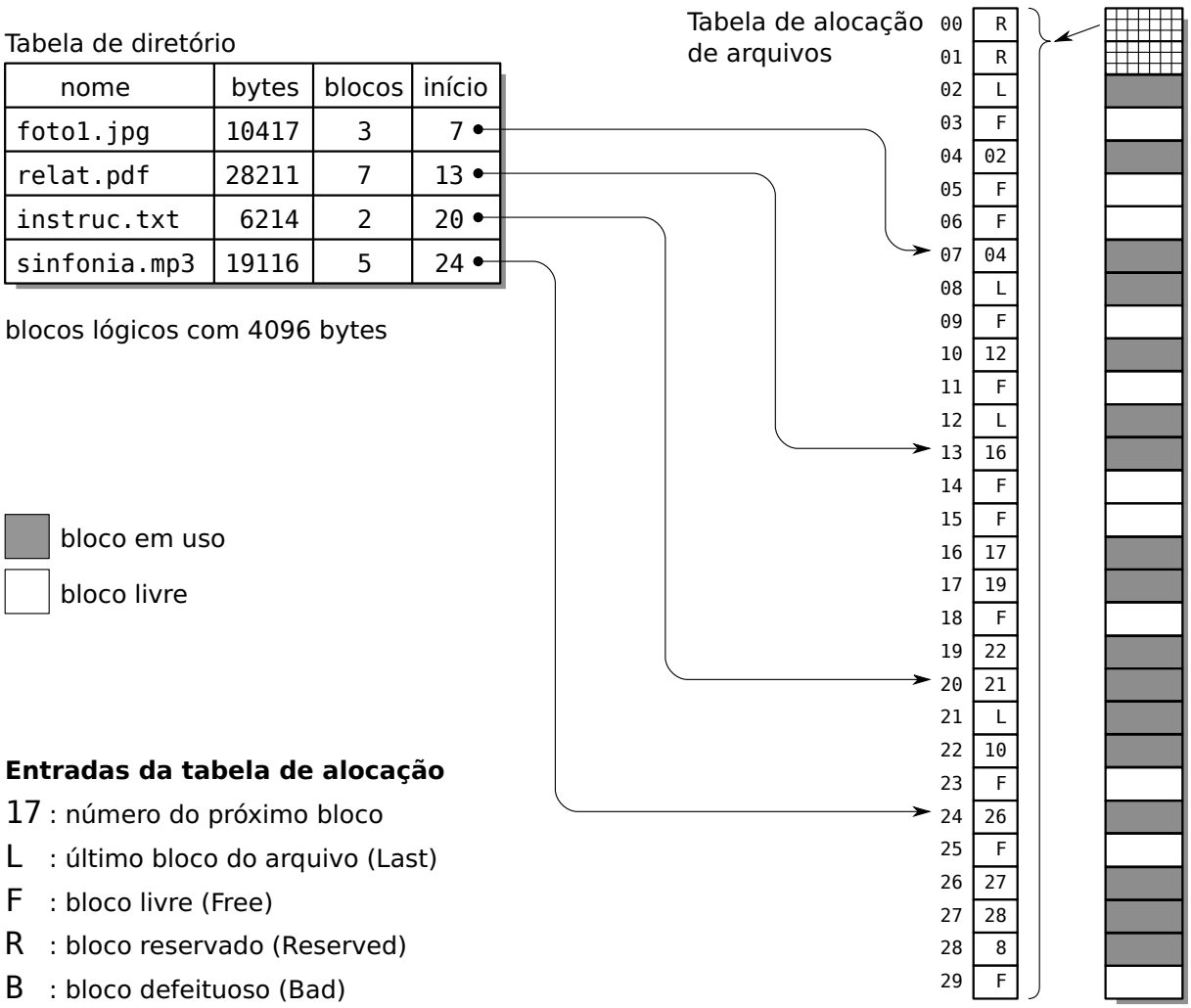

Figura 16: Uma tabela de alocação de arquivos.

4.3.3 Alocação indexada

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 17 apresenta um exemplo de alocação indexada.

Como os *i-nodes* também têm tamanho fixo, o número de entradas no índice de blocos de um arquivo é limitado. Por isso, esta estratégia de alocação impõe um tamanho máximo para os arquivos. Por exemplo, se o sistema usar blocos de 4 KBytes e o índice de blocos suportar 64 entradas, só poderão ser armazenados arquivos com até 256 KBytes. Além disso, a tabela de *i-nodes* também tem um tamanho fixo, determinado durante

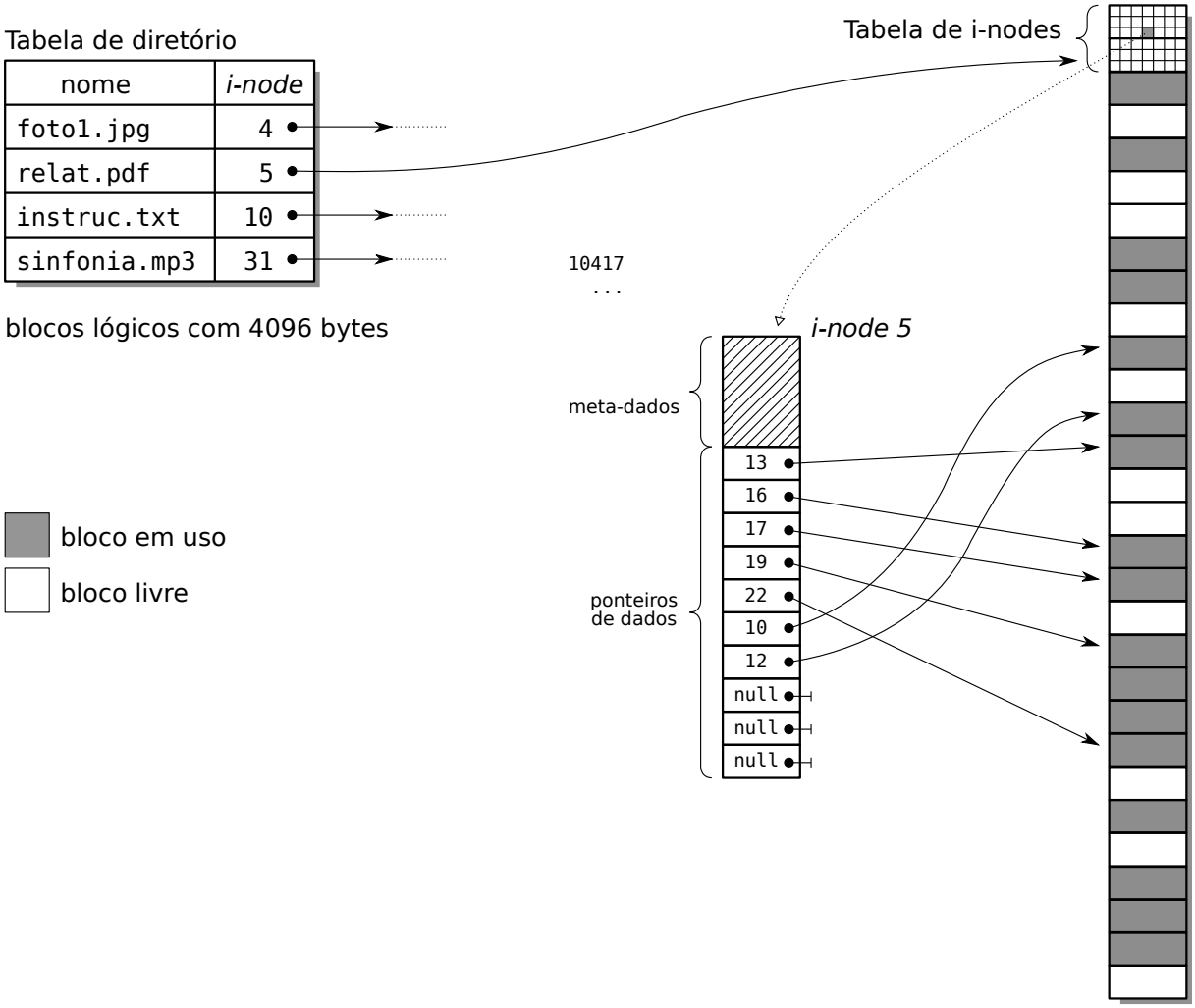

Figura 17: Estratégia de alocação indexada simples.

a formatação do sistema de arquivos, o que limita o número máximo de arquivos ou diretórios que podem ser criados na partição.

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 contém outros ponteiros, criando assim uma estrutura em árvore. Considerando um sistema com blocos lógicos de 4K bytes e ponteiros de 32 bits (4 bytes), cada bloco lógico pode conter 1024 ponteiros, o que aumenta muito a capacidade do índice de blocos. Além de ponteiros indiretos, podem ser usados ponteiros dupla e triplamente indiretos. Por exemplo, os sistemas de arquivos Ext2/Ext3 do Linux (apresentado na Figura 18) 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. Considerando blocos lógicos de 4K bytes e ponteiros de 4 bytes, cada bloco de ponteiros contém 1024 ponteiros. Dessa forma, o cálculo do tamanho máximo de um arquivo nesse sistema é simples:

$$
\begin{array}{rcll}
max & = & 4096 \times 12 & \text{(ponteiros diretos)} \\
 & + & 4096 \times 1024 & \text{(ponteiro indireto)} \\
 & + & 4096 \times 1024 \times 1024 & \text{(ponteiro indireto duplo)} \\
 & + & 4096 \times 1024 \times 1024 \times 1024 & \text{(ponteiro indireto triplo)} \\
 & = & 4.402.345.721.856 \textit{ bytes} & \\
max & \approx & 4T \textit{ bytes} &
\end{array}
$$

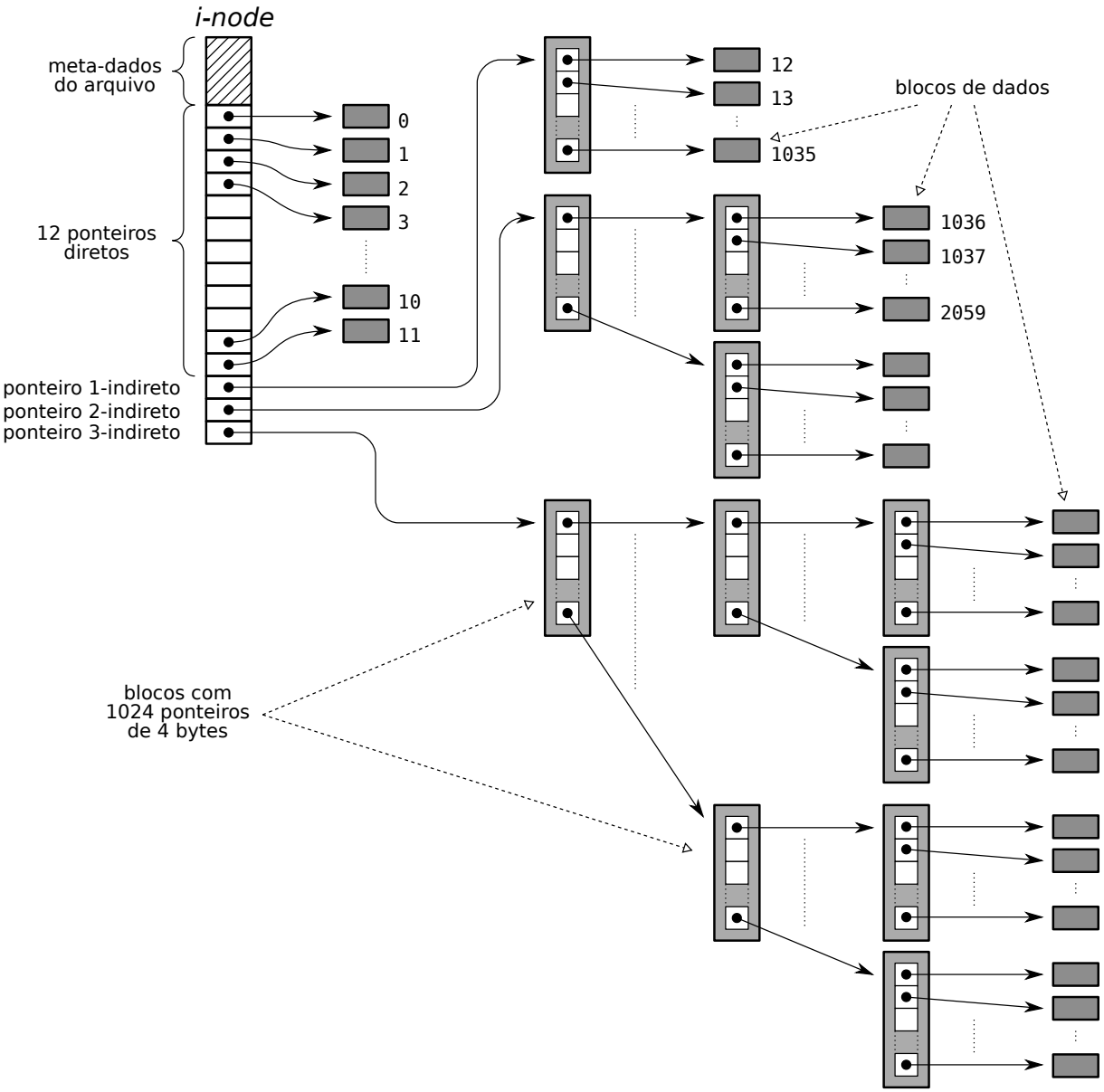

Figura 18: Estratégia de alocação indexada multi-nível.

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 a localização dos blocos com exatidão. A localização do bloco lógico de disco correspondente ao *i*-ésimo bloco lógico de um arquivo segue o algoritmo 3.

Em relação ao desempenho, pode-se afirmar que esta estratégia é bastante rápida, tanto para acessos sequenciais quanto para acessos diretos a blocos, devido aos índices de ponteiros dos blocos presentes nos *i-nodes*. Contudo, no caso de blocos no final de arquivos muito grandes, podem ser necessários três ou quatro acessos a disco adicionais para localizar o bloco desejado, devido aos ponteiros indiretos. Defeitos em blocos de dados não afetam os demais blocos de dados, o que torna esta estratégia robusta.

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

1. B: tamanho dos blocos lógicos, em bytes
2. b_i: número do bloco do disco onde se encontra o byte i
3. o_i: posição do byte i dentro do bloco b_i (*offset*)
4. ptr[0...14]: vetor de ponteiros do *i-node*
5. $block$[0...1023]: bloco de ponteiros para outros blocos
6.
7. $o_i = i \bmod B$
8. $b_{aux} = i \div B$
9. **if** $b_{aux} < 12$ **then** // ponteiros diretos
10. // o endereço do bloco b_i é o próprio valor do ponteiro
11. $b_i = ptr[b_{aux}]$
12. **else**
13. $b_{aux} = b_{aux} - 12$
14. **if** $b_{aux} < 1024$ **then** // ponteiro indireto simples
15. // ler bloco de ponteiros de nível 1
16. $block_1 = read\_block\ (ptr[12])$
17. // encontrar o endereço do bloco b_i
18. $b_i = block_1[b_{aux}]$
19. **else**
20. $b_{aux} = b_{aux} - 1024$
21. **if** $b_{aux} < 1024 \times 1024$ **then** // ponteiro indireto duplo
22. // ler bloco de ponteiros de nível 1
23. $block_1 = read\_block\ (ptr[13])$
24. // ler bloco de ponteiros de nível 2
25. $block_2 = read\_block\ (block_1[b_{aux} \div 1024])$
26. // encontrar o endereço do bloco b_i
27. $b_i = block_2[b_{aux} \bmod 1024]$
28. **else** // ponteiro indireto triplo
29. $b_{aux} = b_{aux} - (1024 \times 1024)$
30. // ler bloco de ponteiros de nível 1
31. $block_1 = read\_block\ (ptr[14])$
32. // ler bloco de ponteiros de nível 2
33. $block_2 = read\_block\ (block_1[b_{aux} \div (1024 \times 1024)])$
34. // ler bloco de ponteiros de nível 3
35. $block_3 = read\_block\ (block_2[(b_{aux} \div 1024) \bmod 1024])$
36. // encontrar o endereço do bloco b_i
37. $b_i = block_3[b_{aux} \bmod 1024]$
38. **end if**
39. **end if**
40. **end if**
41. **return** (b_i, o_i)

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 do disco para armazenar dados. Todavia, o tamanho máximo dos arquivos criados é limitado, bem como o número máximo de arquivos na partição.

Uma característica interessante da alocação indexada é a possibilidade de criar *arquivos esparsos*. Um arquivo esparso contém áreas mapeadas no disco (contendo dados) e áreas não mapeadas (sem dados). Somente as áreas mapeadas estão fisicamente alocadas no disco rígido, pois os ponteiros correspondentes a essas áreas no *i-node* apontam para blocos do disco contendo dados do arquivo. Os ponteiros relativos às áreas não mapeadas têm valor nulo, servindo apenas para indicar que aquela área do arquivo ainda não está mapeada no disco (conforme indicado na Figura 19). Caso um processo leia uma área não mapeada, receberá somente zeros. As áreas não mapeadas serão alocadas em disco somente quando algum processo escrever nelas. Arquivos esparsos são muito usados por gerenciadores de bancos de dados e outras aplicações que precisem manter arquivos com índices ou tabelas *hash* que possam conter grandes intervalos sem uso.

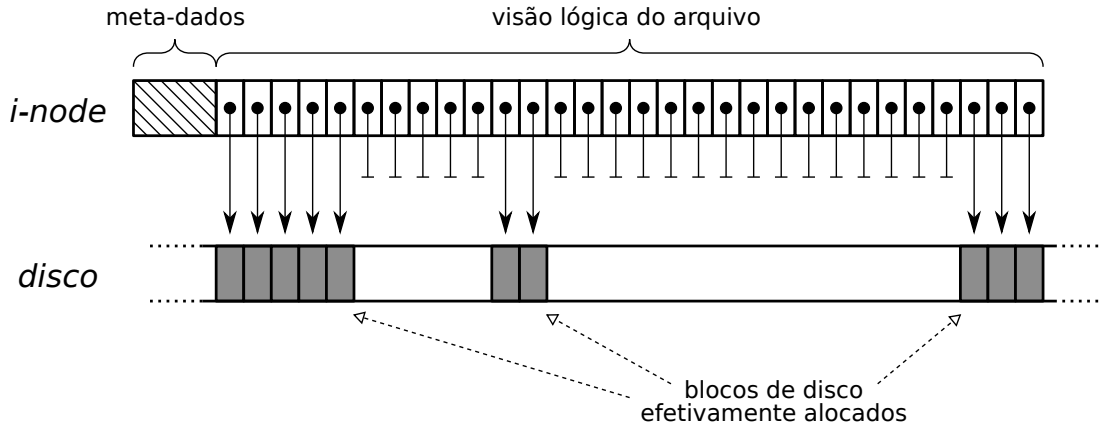

Figura 19: Alocação de um arquivo esparso.

4.3.4 Análise comparativa

A Tabela 3 traz um comparativo entre as principais formas de alocação estudadas aqui, sob a ótica de suas características de rapidez, robustez e flexibilidade de uso.

4.3.5 Gerência de 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. Duas

Estratégia	Rapidez	Robustez	Flexibilidade
Contígua	Alta, pois acessos sequencial e direto rápidos, pois os blocos do arquivo estão próximos no disco.	Alta, pois blocos defeituosos não impedem o acesso aos demais blocos do arquivo.	Baixa, pois o tamanho máximo dos arquivos deve ser conhecido a priori; nem sempre é possível aumentar o tamanho de um arquivo existente.
Encadeada	Acesso sequencial é rápido, se os blocos estiverem próximos; o acesso direto é lento, pois é necessário ler todos os blocos a partir do início do arquivo até encontrar o bloco desejado.	Baixa, pois um bloco defeituoso leva à perda dos dados daquele bloco e de todos os blocos subsequentes, até o fim do arquivo.	Alta, pois arquivos podem ser criados em qualquer local do disco, sem risco de fragmentação externa.
FAT	Alta, pois acessos sequencial e direto são rápidos, se os blocos do arquivo estiverem próximos no disco.	Mais robusta que a alocação encadeada, desde que não ocorram erros na tabela de alocação.	Alta, pois arquivos podem ser criados em qualquer local do disco, sem risco de fragmentação externa.
Indexada	Alta, pois os acessos sequencial e direto são rápidos, se os blocos do arquivo estiverem próximos no disco.	Alta, desde que não ocorram erros no *i-node* nem nos blocos de ponteiros.	Alta, pois arquivos podem ser criados em qualquer local do disco, sem risco de fragmentação externa. No entanto, o tamanho máximo dos arquivos é limitado pelo número de ponteiros definidos nos *i-nodes*.

Tabela 3: Quadro comparativo das estratégias de alocação de arquivos

técnicas de gerência de blocos livres são frequentemente utilizadas: o mapa de bits e a lista de blocos livres [Silberschatz et al., 2001, Tanenbaum, 2003].

Na abordagem de **mapa de bits**, um pequeno conjunto de blocos no início da partição é reservado para manter um mapa de bits. Cada bit nesse mapa de bits representa um bloco lógico da partição, que pode estar livre (o bit vale 1) ou ocupado (o bit vale 0). Essa abordagem como vantagem ser bastante compacta e simples de implementar: em um disco de 80 GBytes com blocos lógicos de 4.096 bytes, seriam necessários 20.971.520 bits no mapa de bits, o que representa 2.621.440 bytes ou 640 blocos (ou seja, 0,003% do total de blocos lógicos do disco).

A abordagem de **lista de blocos livres** pode ser implementada de várias formas. Na forma mais simples, cada bloco livre contém um ponteiro para o próximo bloco livre do disco, de forma similar à alocação encadeada de arquivos vista na Seção 4.3.2. Apesar de simples, essa abordagem é pouco eficiente, por exigir um acesso a disco para cada bloco livre requisitado. A abordagem FAT (Seção 4.3.2) é uma melhoria desta técnica, na qual os blocos livres são indicados por flags específicos na tabela de alocação de

arquivos. Outra 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 livre a cada acesso a disco. Outra melhoria similar consiste em armazenar uma tabela de *extensões* de blocos livres, ou seja, a localização e o tamanho de um conjunto de blocos livres consecutivos no disco, de forma similar à alocação contígua (Seção 4.3.1).

4.4 O sistema de arquivos virtual

O sistema de arquivos virtual gerencia os aspectos do sistema de arquivos mais próximos do usuário, como a verificação de permissões de acesso, o controle de concorrência (atribuição e liberação travas) e a manutenção de informações sobre os arquivos abertos pelos processos.

Conforme apresentado na Seção 2.1, quando um processo abre um arquivo, ele recebe do núcleo uma referência ao arquivo aberto, a ser usada nas operações subsequentes envolvendo aquele arquivo. Em sistemas UNIX, as referências a arquivos abertos são denominadas *descritores de arquivos*, e correspondem a índices de entradas em uma tabela de arquivos abertos pelo processo (*process file table*), mantida pelo núcleo. Cada entrada dessa tabela contém informações relativas ao uso do arquivo por aquele processo, como o ponteiro de posição corrente e o modo de acesso ao arquivo solicitado pelo processo (leitura, escrita, etc.).

Adicionalmente, cada entrada da tabela de arquivos do processo contém uma referência para uma entrada correspondente na tabela global de arquivos abertos (*system file table*) do sistema. Nessa tabela global, cada entrada contém um contador de processos que mantém aquele arquivo aberto, uma trava para controle de compartilhamento e uma referência às estruturas de dados que representam o arquivo no sistema de arquivos onde ele se encontra, além de outras informações [Bach, 1986, Vahalia, 1996, Love, 2004].

A Figura 20 apresenta a organização geral das estruturas de controle de arquivos abertos presentes no sistema de arquivos virtual de um núcleo UNIX típico. Essa estrutura é similar em outros sistemas, mas pode ser simplificada em sistemas mais antigos e simples, como no caso do DOS. Deve-se observar que toda essa estrutura é independente do dispositivo físico onde os dados estão armazenados e da estratégia de alocação de arquivos utilizada; por essa razão, esta camada é denominada *sistema de arquivos virtual*. Essa transparência permite que os processos acessem de maneira uniforme, usando a mesma interface, arquivos em qualquer meio de armazenamento e armazenados sob qualquer estratégia de alocação.

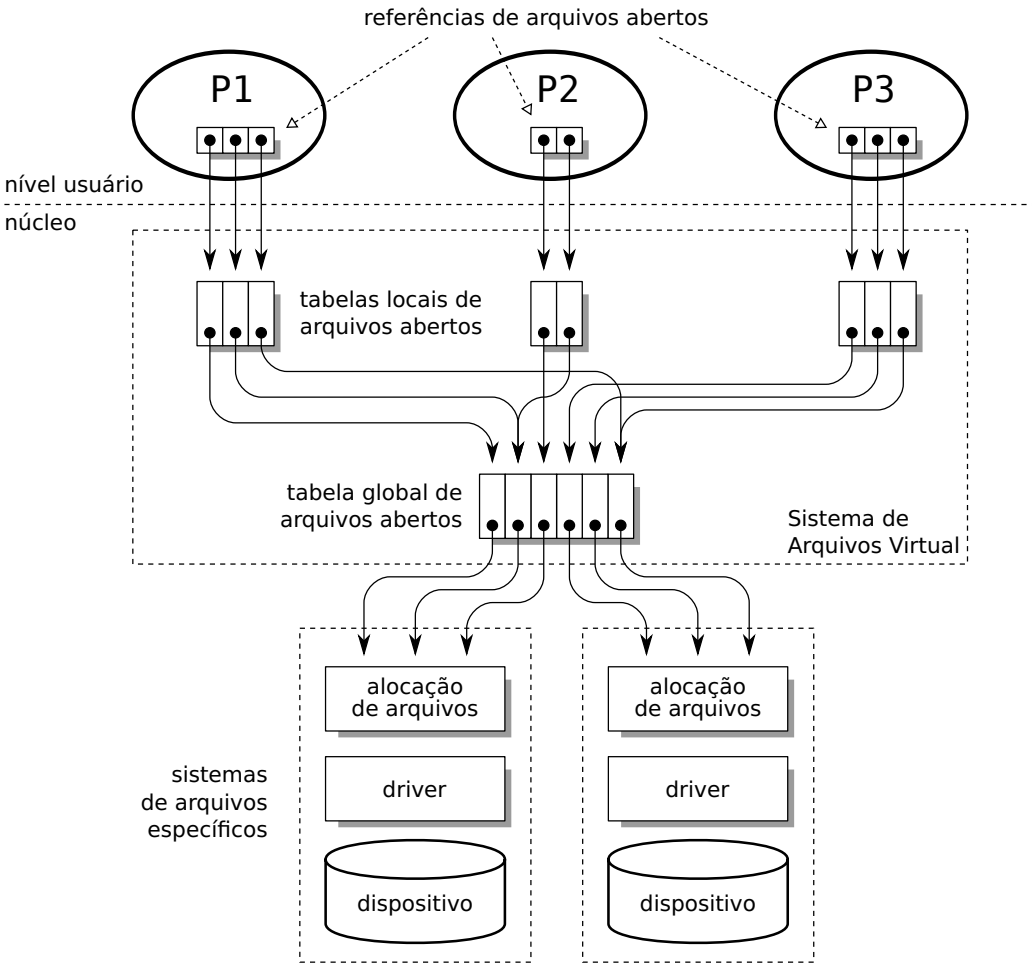

Figura 20: Estruturas de controle de arquivos abertos em um núcleo UNIX.

5 Tópicos avançados

- Journaling FS
- Extents
- Log-structured Fyle Systems

Referências

[Bach, 1986] Bach, M. J. (1986). *The design of the UNIX operating System*. Prentice-Hall.

[Dorward et al., 1997] Dorward, S., Pike, R., Presotto, D., Ritchie, D., Trickey, H., and Winterbottom, P. (1997). The Inferno operating system. *Bell Labs Technical Journal*, 2(1):5–18.

[Freed and Borenstein, 1996] Freed, N. and Borenstein, N. (1996). RFC 2046: Multipurpose Internet Mail Extensions (MIME) part two: Media types.

[Kernighan and Ritchie, 1989] Kernighan, B. and Ritchie, D. (1989). *C: a Linguagem de Programação - Padrão ANSI*. Campus/Elsevier.

[Levine, 2000] Levine, J. (2000). *Linkers and Loaders*. Morgan Kaufmann.

[Love, 2004] Love, R. (2004). *Linux Kernel Development*. Sams Publishing Developer's Library.

[Pike et al., 1995] Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. (1995). Plan 9 from Bell Labs. *Journal of Computing Systems*, 8(3):221–254.

[Pike et al., 1993] Pike, R., Presotto, D., Thompson, K., Trickey, H., and Winterbottom, P. (1993). The use of name spaces in Plan 9. *Operating Systems Review*, 27(2):72–76.

[Rice, 2000] Rice, L. (2000). *Introduction to OpenVMS*. Elsevier Science & Technology Books.

[Russell et al., 2004] Russell, R., Quinlan, D., and Yeoh, C. (2004). Filesystem Hierarchy Standard.

[Silberschatz et al., 2001] Silberschatz, A., Galvin, P., and Gagne, G. (2001). *Sistemas Operacionais – Conceitos e Aplicações*. Campus.

[Tanenbaum, 2003] Tanenbaum, A. (2003). *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall.

[Vahalia, 1996] Vahalia, U. (1996). *UNIX Internals – The New Frontiers*. Prentice-Hall.