Capítulo 3

Sistema de Arquivos

A parte mais visível de um sistema operacional é o seu sistema de arquivos. Programas aplicativos utilizam o sistema de arquivos (via chamadas de sistema) para criar, ler, gravar e remover arquivos. Usuários utilizam interativamente o sistema de arquivos (via *shell*) para listar, alterar propriedades e remover arquivos. A conveniência e facilidade de uso de um sistema operacional é fortemente determinada pela interface, estrutura e confiabilidade de seu sistema de arquivos [1].

3.1 Interface do Sistema de Arquivos

Do ponto de vista do usuário, o aspecto mais importante do sistema de arquivos é como este se apresenta, isto é, o que constitui um arquivo, como os arquivos são identificados e protegidos, que operações são permitidas sobre os arquivos, e assim por diante.

Fundamentos Básicos

A maior parte dos sistemas operacionais trazem a seguinte proposta para armazenamento de informação: permitir aos usuários definir objetos chamados *arquivos*, que podem armazenar programas, dados, ou qualquer outra informação. Estes arquivos não são parte endereçável de nenhum processo e o sistema operacional provê chamadas de sistema para criar, destruir, ler, atualizar e proteger arquivos.

Todos os sistemas operacionais visam uma independência dos dispositivos de armazenamento, permitindo acessar um arquivo sem especificar em qual dispositivo o mesmo se encontra fisicamente armazenado. Um programa que lê um arquivo de entrada e escreve um arquivo saída deve ser capaz de operar com arquivos armazenados em quaisquer dispositivos, sem necessidade de um código especial para explicitar o tipo de periférico.

Alguns sistemas operacionais provêem maior independência dos dispositivos de armazenamento que outros. No UNIX, por exemplo, um sistema de arquivos pode ser montado em qualquer dispositivo de armazenamento, permitindo que qualquer arquivo seja acessado pelo seu nome (*path name*), sem considerar o dispositivo físico. No MS-DOS, por outro lado, o usuário deve especificar em qual dispositivo cada arquivo se encontra (exceto quando um dispositivo é *default* e for omitido). Assim, se o dispositivo *default* for o *drive* C, para executar um programa localizado no *drive* A com arquivos de entrada e saída no *drive* B, cada um deles deverá ser especificado juntamente com o nome do arquivo:

```
A:programa < B:entrada > B:saida
```

A maior parte dos sistemas operacionais suportam vários tipos de arquivos. O UNIX, por exemplo, mantém arquivos regulares, diretórios e arquivos especiais. Arquivos regulares contém dados e programas do usuário. Diretórios permitem identificar arquivos através de nomes simbólicos (i.e. sequência de caracteres ASCII). Arquivos especiais são usados para especificar periféricos tais como terminais, impressoras, unidades de fita, etc. Assim podemos para copiar um arquivo `abc` para o terminal (arquivo especial `/dev/tty`) através do comando:

```
cp abc /dev/tty
```

Em muitos sistemas, arquivos regulares são subdivididos em diferentes tipos em função de sua utilização. Os tipos são identificados pelos nomes com que os arquivos regulares terminam. Por exemplo,

`arquivo.c` - Arquivo fonte em C
`arquivo.obj` - Arquivo objeto
`arquivo.bin` - Programa binário executável
`arquivo.lib` - Biblioteca de arquivos .OBJ usados pelo *linker*

Em certos sistemas, as extensões são simples convenções: o sistema operacional não faz uso delas. Em outros, o sistema operacional tem regras rígidas em relação aos nomes. Por exemplo, o sistema não executará um arquivo a menos que sua extensão seja .BIN.

Diretórios

Para organizar os arquivos, o sistema de arquivos provê *diretórios*, os quais em muitos casos são também arquivos. Um diretório contém tipicamente um número de registros, um por arquivo. Sistemas primitivos admitiam um único diretório compartilhado por todos os usuários, ou um único diretório por usuário. Os sistemas operacionais modernos permitem um número arbitrário de diretórios por usuário (via de regra, formando uma hierarquia). A Fig. 3.1 ilustra estas três situações.

Quando o sistema de arquivos é organizado como uma árvore de diretórios, algum meio se faz necessário para especificar nomes de arquivos. Dois métodos são comumente empregados. No primeiro método, cada arquivo é identificado pela sequência de diretórios desde o diretório raiz até o arquivo (caminho absoluto). Como um exemplo, o caminho `/usr/mfm/mailbox` significa que o diretório raiz (/) contém o subdiretório `usr`, o qual contém o subdiretório `mfm`, que por sua vez contém o arquivo `mailbox`. Nomes absolutos para caminhos sempre começam na raiz e são únicos.

Uma outra forma de especificar nomes de arquivos é através de seu caminho relativo. É usado em conjunto com o conceito de diretório de trabalho (ou diretório corrente). Um usuário pode designar um diretório como diretório corrente. Neste caso, todos os caminhos são referenciados a partir do diretório corrente. Se o diretório corrente for é `/usr/mfm`, então o arquivo cujo caminho absoluto é `/usr/mfm/mailbox` pode ser referenciado simplesmente como `mailbox`. Em UNIX o diretório corrente é denotado por `.` (ponto).

3.2 Projeto do Sistema de Arquivos

Examinaremos agora o sistema de arquivos do ponto de vista do projetista de sistemas operacionais. Aos usuários interessa como os arquivos são identificados, quais operações são

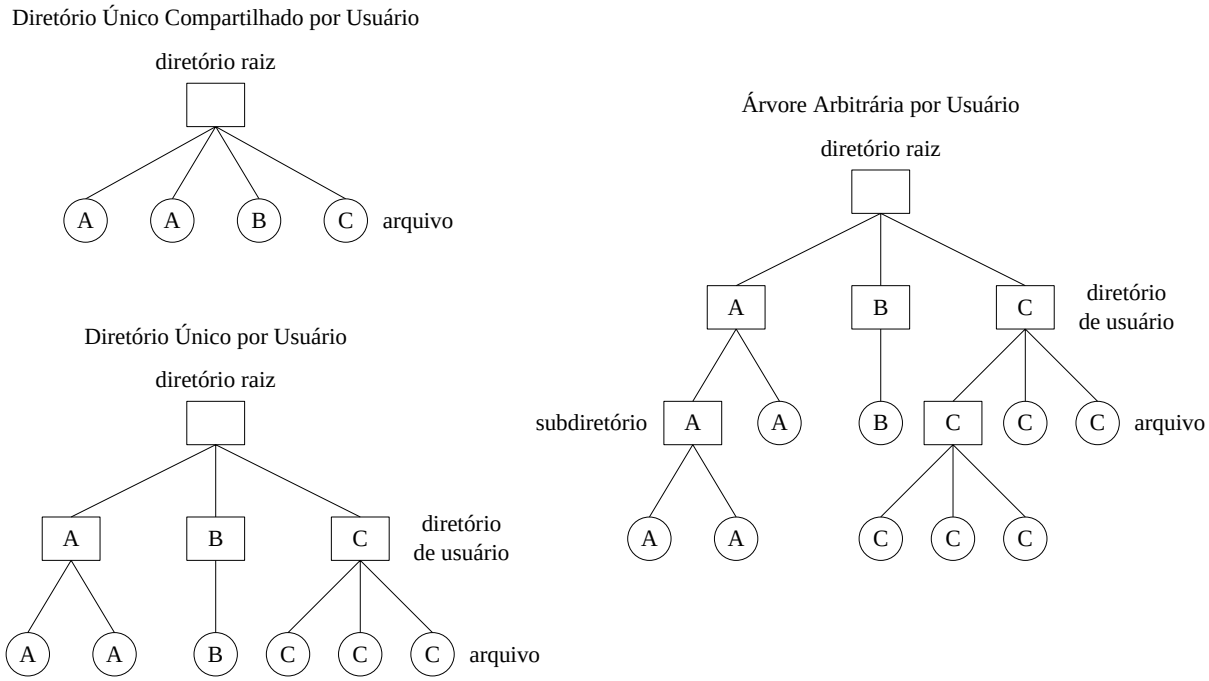

Fig. 3.1: Três projetos de sistemas de arquivos: (a) diretório único compartilhado pelos usuários; (b) um diretório por usuário; (c) árvore arbitrária por usuário

permitidas, como os diretórios são organizados, etc. Projetistas estão interessados em como o espaço de disco é gerenciado, como os arquivos são armazenados, e como manipular arquivos de forma eficientemente e confiável.

Gerenciamento de Espaço em Disco

Arquivos são normalmente armazenados em disco, sendo portanto o gerenciamento do espaço em disco de maior interesse do projetista. Duas estratégias são possíveis para armazenamento em um arquivo com n bytes: n bytes consecutivos do disco são alocados; ou o arquivo é dividido em um número de blocos não necessariamente contíguos. A mesma política está presente no sistema de gerenciamento de memória entre a segmentação pura e a paginação.

Armazenar um arquivo como uma sequência contígua de bytes apresenta um problema óbvio que é o crescimento do arquivo, uma ocorrência muito comum. O arquivo provavelmente terá que ser movido no disco. O mesmo problema é apresentado para segmentação na memória, exceto que mover um segmento na memória é uma operação relativamente mais rápida. Por esta razão, normalmente todos os sistemas de arquivos armazenam os arquivos em blocos de tamanho fixo, que não precisam ser adjacentes[1].

Uma vez decidido armazenar arquivos em blocos de tamanho fixo, a questão é definir qual o tamanho do bloco a ser usado. Dado a forma como os discos são organizados, os setores, as trilhas e os cilindros são candidatos óbvios para a unidade de alocação.

Uma unidade de alocação grande, tal como um cilindro, implica que muitos arquivos, até mesmo arquivos de 1 byte, deverão ocupar o cilindro inteiro. Por outro lado, usar uma unidade de alocação pequena, significa que cada arquivo terá muitos blocos. A leitura de cada

[1]Salvo alguns sistemas operacionais, notadamente os voltados à computação de tempo-real, onde o armazenamento contínuo é adotado por razões de desempenho.

bloco normalmente requer uma busca e uma latência rotacional. Assim, a leitura de arquivos consistindo de muitos blocos pequenos será lenta.

É compromisso usual escolher um bloco de tamanho 512, 1K ou 2K bytes. Se um bloco de tamanho 1K for escolhido em um disco com setor de 512 bytes, então o sistema de arquivo sempre irá ler ou escrever em dois setores consecutivos, e tratá-los como uma unidade indivisível.

Uma vez escolhido o tamanho do bloco, a próxima questão é como manter o rastreamento de blocos livres no disco. Dois métodos são largamente usados (Fig. 3.2). O primeiro consiste no uso de uma lista ligada de blocos, com cada elemento da lista armazenando tantos blocos livres quanto possível. Com elementos de 1K e o número do bloco de 16 bits, cada elemento na lista de blocos livre armazena 511 blocos livres. Um disco com 20 Gigabytes necessita de uma lista ocupando aproximadamente 40K blocos para apontar para todos os 20G blocos do disco (ou seja, a lista ocupa 0,2% do disco).

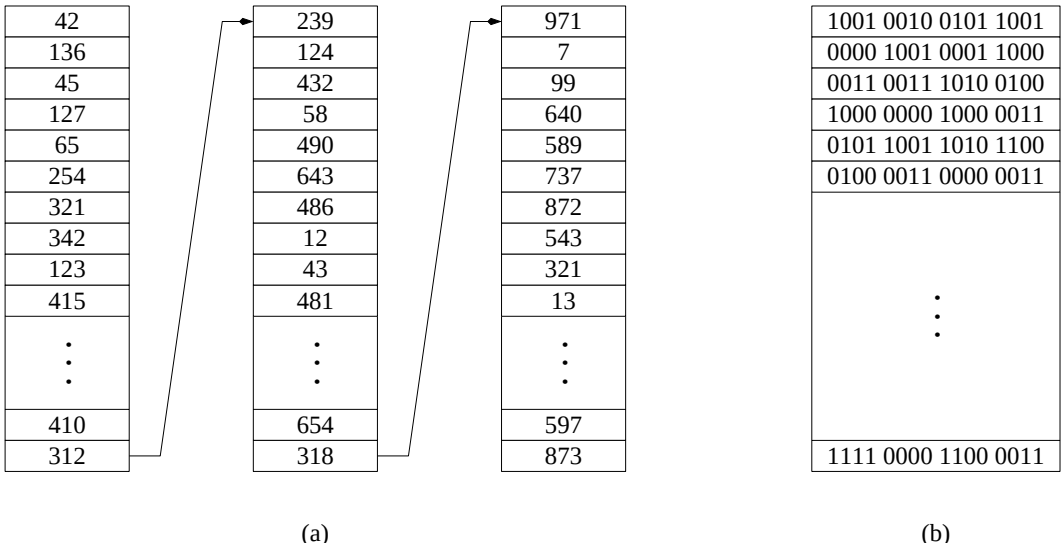

Fig. 3.2: (a) blocos livres armazenados em lista ligada; (b) um mapa de bits.

Uma outra técnica de gerenciamento de espaço livre é o mapa de bits. Um disco com n blocos necessita de um mapa de bits com n bits. Blocos livres são representados por 1s no mapa de bits; blocos alocados por 0s (ou vice-versa). Um disco com 20 Gigabytes necessita de 20M bits para o mapa, volume equivalente a 2500 blocos (ou seja, o mapa ocupa apenas 0,013% do disco). Não é surpresa que um mapa de bit necessite de menos espaço, desde que usa um bit por bloco, versus 16 bits da lista ligada. Entretanto, para um disco cheio (com poucos blocos livres) a lista ligada necessita de menos espaço que o mapa de bits.

Armazenamento de Arquivos

Se um arquivo consistir de uma sequência de blocos, o sistema de arquivos deve ter uma maneira de acessar todos os blocos do arquivo. Como visto acima, um método possível consiste em armazenar os blocos de um arquivo em uma lista ligada. Cada bloco de disco de 1024 bytes, contém 1022 bytes de dados e um ponteiro de 2 bytes para o próximo elemento da lista. Esse método tem duas desvantagens, entretanto. Primeiro, o número de bytes de dados em um elemento da lista não é uma potência de 2, o que frequentemente é uma desvantagem para sua manipulação. Segundo, e mais sério, o acesso aleatório é de difícil implementação. Se um programa busca o byte 32768 de um arquivo, o sistema operacional tem que percorrer 32768/1022

ou 33 blocos para encontrar o dado. Ler 33 blocos para buscar um dado, é inaceitável.

Entretanto, a idéia de representar um arquivo como uma lista encadeada, pode ser ainda explorada se mantermos os ponteiros em memória. A Fig. 3.3 mostra o esquema de alocação usado pelo MS-DOS. Neste exemplo, temos três arquivos, A, com os blocos 6,8,4 e 2; B, com os blocos 5, 9 e 12; e C, com os blocos 10, 3 e 13.

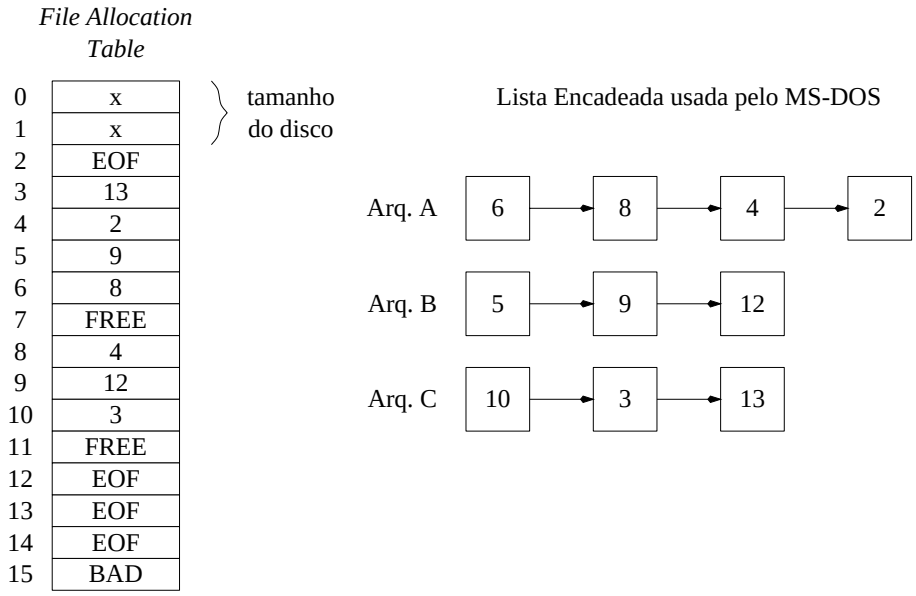

Fig. 3.3: Esquema de lista encadeada usado pelo MS-DOS. Os registros 0 e 1 são usadas para especificação do tipo do disco. Os códigos EOF e FREE são usados para *End Of File* e registros livres, respectivamente.

Associada a cada disco, existe uma tabela chamada Tabela de Alocação de Arquivos *(File Allocation Table–FAT)* que contém um registro para cada bloco do disco. O registro no diretório para cada arquivo fornece o endereço inicial do arquivo na FAT. Cada unidade da FAT contém o número do próximo bloco do arquivo. O arquivo A começa no bloco 6, então o registro 6 da FAT contém o endereço do próximo bloco do arquivo A, que é 8. O registro 8 da FAT contém o número do próximo bloco que é o 4. O registro 4 aponta para o registro 2, que está marcado como fim do arquivo.

Este esquema vai se tornando ineficiente a medida que a capacidade do disco aumenta. Para limitar o tamanho da tabela, deve-se aumentar o tamanho do bloco. Suponha um disco de 2 Gigabytes que contém 16K blocos de 32K, resultando em uma FAT com 16K entradas de 2 bytes cada. Dois problemas são intrínsecos deste esquema:

1. dado que mais que um arquivo não pode ocupar o mesmo bloco, um arquivo de 1 byte é armazenado em um bloco de 32 Kbytes;

2. por razões de eficiência, toda a FAT deve estar presente integralmente em memória, independentemente do número de arquivos abertos.

Um método mais eficaz, seria manter listas dos blocos para diferentes arquivos em lugares diferentes. Isto é exatamente o que o UNIX faz.

Associado a cada arquivo no UNIX, tem-se uma pequena tabela (no disco), chamada *inode*, como mostrado na Fig. 3.4. Ela contém informações sobre o arquivo tais como tamanho e

proteção. Os itens chaves são os 10 números de blocos do disco e os 3 números de blocos indiretos. Para arquivos com 10 blocos ou menos, todos os endereços dos blocos de dados no disco são mantidos no próprio *inode*, sendo sua localização imediata.

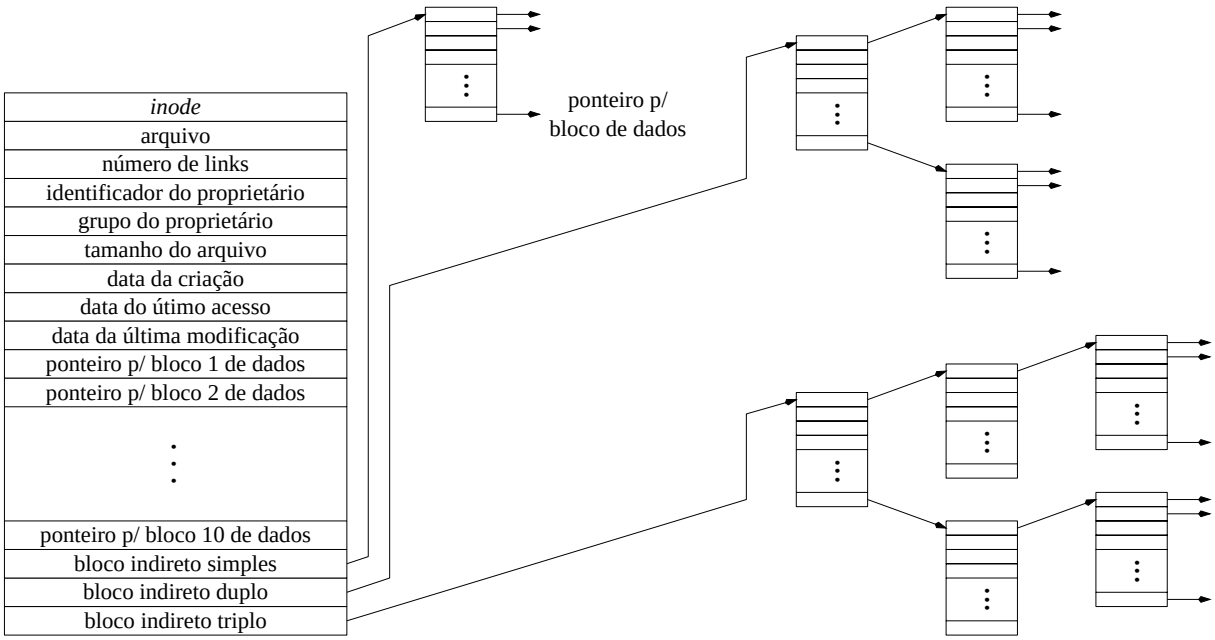

Fig. 3.4: Estrutura do *inode*

Quando o tamanho de um arquivo supera 10 blocos, um bloco livre é adquirido e o ponteiro indireto simples passa a apontar para ele. Este bloco é usado para armazenar os ponteiros dos blocos de disco. Com um bloco de disco de 1K e endereços de disco de 32 bits, o bloco indireto simples pode acessar 256 endereços de disco. Esse esquema é suficiente para arquivos de até 266 blocos (10 no *inode*, 256 no bloco indireto simples).

Acima de 266 blocos, o ponteiro indireto duplo é usado para apontar para um bloco de disco de até 256 ponteiros, que não apontam para blocos de dados, mas sim para 256 blocos indiretos simples. O bloco indireto duplo é suficiente para arquivos de até $266 + 256^2 = 65802$ blocos. Para arquivos maiores que 64K bytes, o ponteiro indireto triplo é usado para apontar para um bloco que contém ponteiros para 256 blocos indiretos duplos, permitindo arquivos de até 16 Gigabytes.

Arquivos maiores que 16 Gigabytes não podem ser manuseados por este esquema. Entretanto, aumentando o tamanho do bloco para 2K, cada bloco de ponteiro acessa 512 ponteiros ao invés de 256, e o tamanho máximo de um arquivo se torna 128 Gigabytes. O ponto forte do esquema do UNIX é que os blocos indiretos são usados somente quando for necessário. Para blocos de 1K bytes, arquivos com menos de 10K bytes não necessitam blocos indiretos. Note que mesmo para os arquivos mais longos são necessários, no máximo, 3 acessos a disco para localizar o endereço de um arquivo (descontado o acesso ao *inode*).

Estrutura de Diretório

Antes de um arquivo ser manipulado, ele deve ser aberto. Quando um arquivo é aberto, o sistema operacional usa o nome de caminho fornecido pelo usuário para localizar os blocos no disco. Mapeando nomes de caminhos em *inodes* (ou equivalentes), introduz-se ao tópico de como

sistemas de diretórios são organizados. Estes variam desde soluções simples até razoavelmente sofisticadas.

Vamos começar com um sistema de diretório particularmente simples, aquele do CP/M, ilustrado na Fig. 3.5(a). Neste sistema, existe apenas um diretório. Assim, a localização de um arquivo reduz-se a procura em um único diretório. Encontrado o registro do arquivo, tem-se o número de blocos do disco, posto que estes são armazenados no próprio registro. Se o arquivo utiliza mais blocos de disco que o permitido no registro, o arquivo terá registros adicionais no diretório.

Os campos na Fig. 3.5(a) são resumidos a seguir. O campo de *usuário* informa a quem pertence o arquivo. Durante a pesquisa, apenas os registros pertencentes ao usuário corrente são considerados. Os próximos campos dão o nome e tipo do arquivo. O campo *tamanho* é necessário porque um arquivo grande que ocupa mais de 16 blocos, utiliza múltiplos registros no diretório. Estes campos são usados para especificar a ordem dos registros. O campo *contador de bloco* informa quais dos 16 blocos de disco estão em uso. Os 16 campos finais contém os números dos blocos de disco. O tamanho dos arquivos é medido em blocos, não em bytes.

Vamos considerar agora exemplos de sistemas de diretório em árvore (hierárquicos). A Fig. 3.5(b) ilustra um registro de diretório do MS-DOS com 32 bytes de comprimento e armazenando o nome do arquivo e o número do primeiro bloco, dentre outros itens. O número do primeiro bloco pode ser usado como um índice dentro da FAT, para achar o segundo bloco, e assim sucessivamente. Deste modo, todos os blocos podem ser encontrados a partir do primeiro bloco armazenado na FAT. Exceto para um diretório raiz, o qual é de tamanho fixo (448 registros para um disquete de 1.44M), os diretórios do MS-DOS são arquivos e podem conter um número arbitrário de registros.

A estrutura de diretório usada no UNIX é extremamente simples, como mostra a Fig. 3.5(c). Cada registro contém exatamente o nome do arquivo e seu número de *inode*. Todas as informações sobre tipo, tamanho, marcas de tempo, propriedade, e blocos do disco estão contidas no *inode* (veja Fig. 3.4). Todos os diretórios do UNIX são arquivos e podem conter um número arbitrário destes registros.

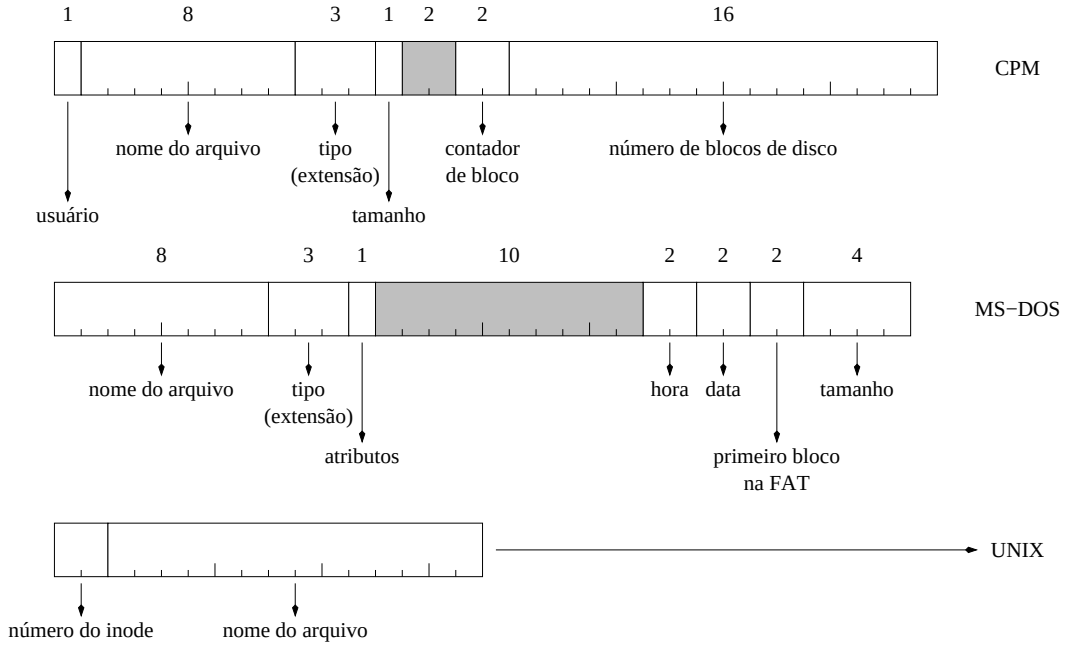

Fig. 3.5: Registros de diretórios: (a) CPM; (b) MS-DOS; (c) UNIX

Quando um arquivo é aberto, o sistema de arquivos recebe o nome de arquivo fornecido e localiza seus blocos no disco. Vamos considerar como o nome de caminho `/usr/mfm/mailbox` é localizado. Usaremos o UNIX como um exemplo, mas o algoritmo é basicamente o mesmo para todo sistema hierárquico de diretórios. Primeiro, o sistema de arquivo localiza o diretório raiz. No UNIX, o *inode* da raiz é posicionado num lugar fixo no disco.

Então, procura-se pelo primeiro componente do caminho, `usr`, no diretório raiz para achar o *inode* do arquivo `/usr`. Deste *inode*, o sistema localiza o diretório para `/usr` e procura pelo próximo componente, `mfm`, neste caso. Quando o registro para `mfm` é encontrado, tem-se o *inode* para o diretório `/usr/mfm`. A partir deste *inode*, pode-se achar o próprio diretório e procurar pela entrada do arquivo `mailbox`. O *inode* para este arquivo é então lido para a memória e lá será mantido até que o arquivo seja fechado. Este processo é ilustrado na Fig. 3.6.

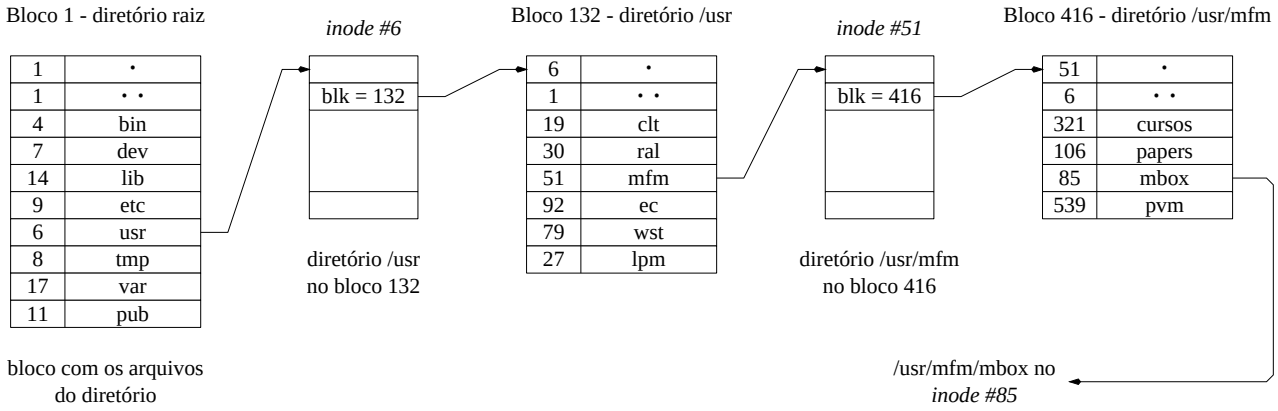

Fig. 3.6: Os passos para achar `/usr/mfm/mailbox`

Nomes de caminhos relativos são pesquisados de forma idêntica, apenas partindo do diretório de trabalho em vez de partir-se do diretório raiz. Todos os diretórios têm registros para `.` e `..`, criados juntamente com o diretório. O registro `.` armazena o número do *inode* do diretório corrente, e o registro `..` o número do *inode* do diretório pai. Assim, o procedimento de procurar por `../src/prog.c` simplesmente localiza `..` no diretório de trabalho, acha o número do *inode* para o diretório pai, e pesquisa pelo diretório `src`. Nenhum mecanismo especial é necessário para manipular estes nomes.

Arquivos Compartilhados

Não raro, usuários desenvolvendo trabalhos em equipe necessitam compartilhar arquivos. Como resultado, é conveniente que um mesmo arquivo pertença simultaneamente a diferentes diretórios. A Fig. 3.7 mostra o sistema de arquivos da Fig. 3.1(c), com um dos arquivos do usuário C presente em um dos diretórios do usuário B. A associação entre um diretório e um arquivo pertencente a outro diretório é chamada de *conexão* ou *link* (linha pontilhada da Fig. 3.7). O sistema de arquivos é agora um grafo acíclico dirigido, ou DAG *(directed acyclic graph)*.

Compartilhar arquivos é conveniente, mas também fonte de alguns problemas. Por exemplo, se diretórios armazenam endereços de disco, como no CP/M, a conexão se dá pela cópia dos endereços dos blocos do diretório do qual o arquivo já faz parte para o diretório sendo conectado. Se um usuário aumentar o tamanho do arquivo, os novos blocos serão listados somente no diretório deste usuário: as mudanças não serão visíveis para os outros usuários, anulando desta forma o propósito do compartilhamento.

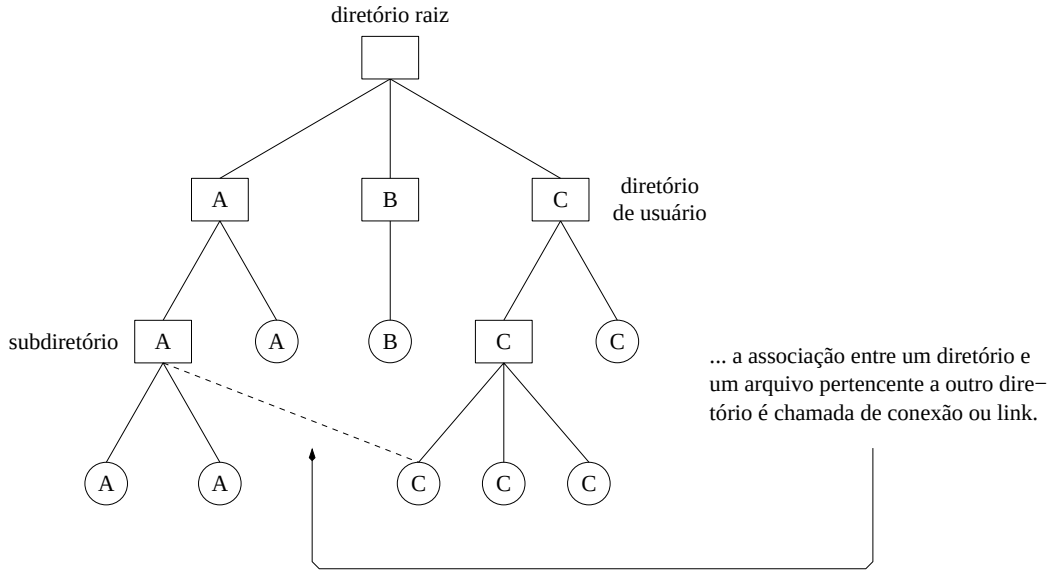

Fig. 3.7: Um sistema de arquivos contendo um arquivo compartilhado

Este problema pode ser solucionado de duas maneiras. Na primeira, os blocos do disco não são listados nos diretórios, mas em uma pequena estrutura de dados associada com o arquivo em questão. Os diretórios então apontariam justamente para a pequena estrutura de dados. Esta é a estratégia usada no UNIX (onde a pequena estrutura de dados é o *inode*).

Na segunda solução, B conecta-se a um dos arquivos de C através da criação (em B) de um arquivo especial. Este arquivo contém justamente o caminho do arquivo conectado. Quando B referencia o arquivo conectado, o sistema operacional identifica-o como do tipo *link*, lendo deste apenas o caminho para o arquivo compartilhado. De posse do caminho, o arquivo compartilhado é acessado. Este método é chamado de *conexão simbólica*.

Cada um destes métodos têm suas desvantagens. No primeiro método, no momento que B conecta-se ao arquivo compartilhado, o *inode* registra C como proprietário do arquivo. A criação de uma conexão não muda o proprietário do arquivo (ver Fig. 3.8), mas incrementa um contador no *inode* que diz quantos diretórios apontam para o arquivo.

No caso de C subsequentemente tentar remover o arquivo, o sistema encontra um problema. Se o sistema remover o arquivo e seu *inode*, B terá um registro de diretório apontando para um *inode* inválido. Se o *inode* for mais tarde reutilizado para um outro arquivo, a conexão de B apontará para o arquivo incorreto. O sistema pode ver pelo contador do *inode* que o arquivo está ainda em uso, mas não há maneira de encontrar todos os registros do diretório para o arquivo, a fim de apagá-los. Ponteiros para os diretórios não podem ser armazenados no *inode*, uma vez que podem haver um número ilimitado de diretórios.

A única solução é remover os registros do diretório C, mas abandonar o *inode* intacto, com o contador em 1, como mostra a Fig. 3.8(c). Temos agora uma situação na qual B tem um registro de diretório para um arquivo de C. Se o sistema faz controle de cotas, C continuará sendo contabilizado pelo arquivo até B decidir removê-lo, momento em que o contador irá para 0 e o arquivo será removido.

Empregando-se conexões simbólicas este problema não ocorre pois somente um diretório proprietário guarda o ponteiro para o *inode*. Diretórios que acabaram de conectar-se ao arquivo armazenam o caminho para o arquivo, não ponteiros para o seu *inode*. Quando o arquivo for removido do diretório proprietário, o *inode* é liberado pelo sistema, e subsequentes tentativas

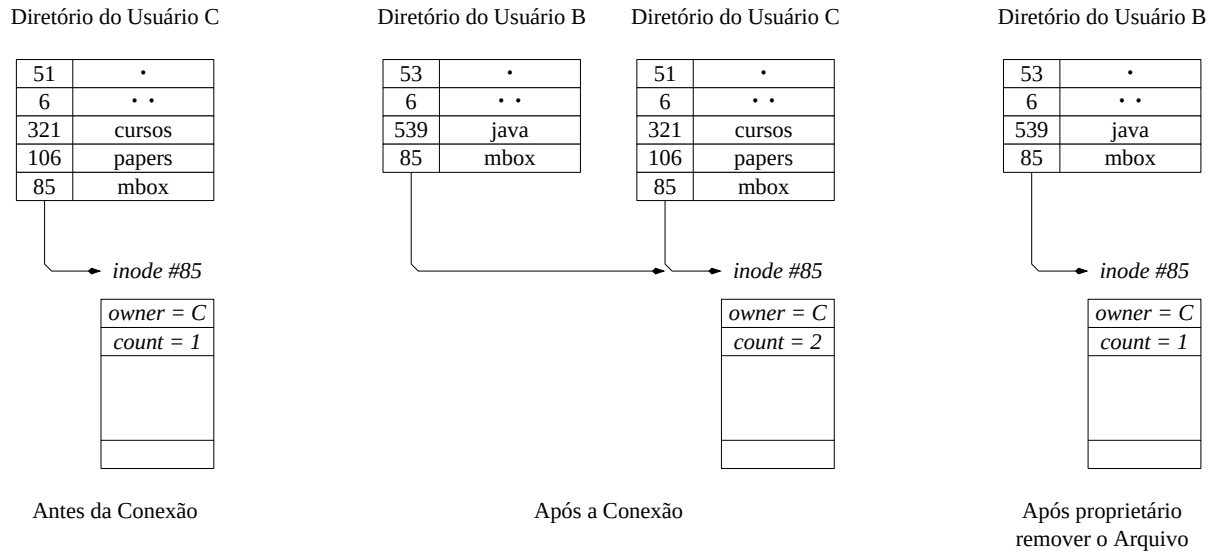

Fig. 3.8: (a) situação anterior à conexão; (b) após a conexão ter sido feita; (c) após o proprietário remover o arquivo

para usar o arquivo via conexão simbólica falhará, dada a incapacidade do sistema em localizar o arquivo. Remover uma conexão simbólica não afeta o arquivo, causando apenas o decréscimo do contador do *inode*.

Conexões simbólicas introduzem um *overhead* extra na manipulação de arquivos. Num acesso via conexão simbólica, vários *inodes* devem ser lidos do disco: o primeiro para acessar o caminho e os subsequentes para percorrer todo o caminho até a localização do arquivo (ver Fig. 3.6). Além de múltiplos acessos a disco, um *inode* extra é necessário para cada conexão simbólica, juntamente com um bloco extra para armazenar o caminho.

Existe ainda outro problema introduzido pelas conexões, simbólicas ou não. Quando conexões são permitidas, uma pesquisa numa árvore de diretórios pode encontrar o mesmo arquivo várias vezes. Isto é um problema a se considerar, por exemplo, em aplicativos que efetuam *backups*.

3.3 Confiabilidade do Sistema de Arquivos

Blocos Defeituosos

Discos frequentemente apresentam blocos defeituosos *(bad blocks)*, isto é, blocos onde a escrita e/ou leitura é impossibilitada. Duas soluções para o problema de blocos defeituosos são empregadas, uma em *hardware* e outra em *software*. A solução em *hardware* consiste em dedicar um setor no disco para a lista de blocos defeituosos. Quando o controlador do disco é iniciado, este lê a lista de blocos defeituosos e escolhe blocos sobressalentes para substituí-los. São feitas então indireções dos blocos defeituosos para os blocos sobressalentes. Daí por diante, qualquer operação envolvendo um bloco defeituoso terá efeito em seu respectivo bloco sobressalente.

A solução em *software* requer que o usuário informe (ou que o sistema de arquivos detecte) os blocos defeituosos. Estes blocos são armazenados num arquivo, acessado quando da construção da lista de blocos livres. Conhecendo os blocos defeituosos, o sistema operacional não os incorpora na lista de blocos livres, eliminando assim sua ocorrência futura em arquivos de dados.

Backups

Mesmo com uma estratégia engenhosa para tratar os blocos defeituosos, é importante se proceder *backups* frequentes. Sistemas de arquivos em discos de pequena capacidade podem ser salvos em fita magnética ou disquetes de alta densidade.

Para discos de grande capacidade (dezenas de gigabytes), salvar o conteúdo inteiro em fitas é inconveniente e consome muito tempo. Uma estratégia de fácil implementação, mas que diminui pela metade a capacidade de armazenamento, é prover cada computador com um segundo disco de igual capacidade. Ambos os discos são divididos em duas metades: dados e *backup*. Diariamente, a porção de dados de um disco é copiada para a porção de *backup* do outro disco, e vice-versa. Deste modo, se um disco for completamente destruído, nenhuma informação é perdida.

Uma outra alternativa é o *backup* incremental. Em sua forma mais simples, copia-se para fita todos os arquivos a cada semana ou mês, e, diariamente, apenas daqueles arquivos que foram modificados deste o último *backup* completo. Um outro esquema, mais eficiente, copia-se apenas aqueles arquivos que foram alterados desde o último *backup*. Para implementar este método, o horário da última duplicação para cada arquivo deve ser mantida no disco.

Consistência do Sistema de Arquivos

Outro tópico envolvendo confiabilidade é a consistência do sistema de arquivos. Muitos sistemas de arquivos lêem blocos, modifica-os, e os regrava mais tarde. Se o sistema falha antes que todos os blocos modificados forem escritos no disco, o sistema de arquivos assume um estado inconsistente. Este problema é especialmente crítico se alguns dos blocos que não foram escritos, são blocos de *inodes*, blocos de diretório, ou blocos contendo a lista de blocos livres.

Para verificar a consistência do sistema de arquivos, muitos sistemas operacionais utilizam programas utilitários desenvolvidos para este fim. Este programa é executado sempre que o sistema é iniciado, particularmente depois de um desligamento abrupto. A seguir é descrito como tal utilitário opera no UNIX[2].

O controle de consistência se dá em dois níveis: blocos e arquivos. Para controle de consistência no nível de bloco, o utilitário constrói uma tabela com dois contadores por bloco, ambos iniciados em 0. O primeiro contador rastreia quantas vezes o bloco aparece no arquivo; o segundo registra com que frequência ele aparece na lista de blocos livres.

O utilitário lê todos os *inodes*. Começando de um *inode*, é possível construir uma lista de todos os números de blocos usados no correspondente arquivo. Assim que cada número de bloco é lido, seu respectivo contador na primeira tabela é incrementado. A seguir, é examinada a lista de blocos livres rastreando todos os blocos que não estão em uso. Cada ocorrência de um bloco na lista de blocos livres resulta no incremento do respectivo contador na segunda tabela.

Se o sistema de arquivo for consistente, cada bloco terá o valor 1 na primeira tabela ou na segunda tabela. Contudo, em caso de falha, pode detectar-se blocos que não ocorrem em nenhuma das tabelas (blocos perdidos). Embora blocos perdidos não causem um dano real, eles desperdiçam espaço, reduzindo assim a capacidade do disco. A solução para blocos perdidos é direta: o verificador do sistema de arquivos acrescenta-os na lista de blocos livres.

Outra situação possível de ocorrer é a repetição de blocos na lista de blocos livres. A solução neste caso também é simples: reconstruir a lista de blocos livres, eliminando-se as duplicações.

[2]Este utilitário denomina-se *fsck (file system checker)*.

O mais grave é a ocorrência do mesmo bloco de dados em dois ou mais arquivos. Se cada um desses arquivos for removido, o bloco duplicado será posto na lista de blocos livres, chegando-se em uma situação em que o mesmo bloco está, ambiguamente, em uso e livre ao mesmo tempo. Se ambos os arquivos forem removidos, o bloco será adicionado na lista de blocos livres duas vezes.

A ação apropriada do utilitário é alocar um bloco livre, copiar o conteúdo do bloco duplicado para o mesmo, e inserir a cópia em um dos arquivos. Desde modo, a informação dos arquivos não é alterada (embora certamente incorreta para um dos arquivos), mas a estrutura do sistema de arquivos é, pelo menos, consistente. O erro será informado para permitir ao usuário examinar a falha.

Ainda para verificar se cada bloco é contado corretamente, o utilitário também examina o sistema de diretórios (consistência no nível de arquivos). Neste caso, é usada uma tabela de contadores por arquivos (não por blocos, como anteriormente). A verificação começa no diretório raiz e, recursivamente, desce a árvore inspecionando cada diretório no sistema de arquivos. Para cada arquivo encontrado, incrementa-se o contador para o seu respectivo *inode*.

Quando toda a árvore de diretórios é percorrida, tem-se uma lista, indexada pelo número do *inode*, descrevendo quantos diretórios apontam para aquele *inode*. O utilitário então compara esses valores com os contadores dos *inodes*. Em um sistema de arquivos consistente, ambos contadores coincidirão. Contudo, dois tipos de erros podem ocorrer: o contador do *inode* pode ser maior ou menor que o da lista do utilitário.

Se a contagem no *inode* for maior que o número de registros do diretório, então mesmo se todos os arquivos forem removidos dos diretórios, o contador ainda será diferente de 0 e o *inode* não será liberado. Este erro não é catastrófico, mas consome espaço no disco com arquivos que não estão em nenhum dos diretórios. O contador de conexões do *inode* deve ser corrigido através da atribuição do valor obtido pelo utilitário.

Outro erro (potencialmente catastrófico) ocorre quando o contador do *inode* é menor que o encontrado pelo utilitário. A medida que os arquivos que apontam para o *inode* vão sendo removidos, o contador do *inode* pode chegar a zero, momento que o *inode* e seus respectivos blocos são liberados. Esta ação resultará em um dos diretórios apontando para um *inode* não mais em uso, cujos blocos podem rapidamente ser atribuídos a outros arquivos. Novamente, a solução é forçar o contador do *inode* para o número real de registros do diretório (obtidas pelo utilitário).

Estas duas operações, verificar blocos e verificar diretórios, são frequentemente integradas por razões de eficiência (i.e., uma única passagem sobre os *inodes* é requerida). Outros controles heurísticos são também possíveis. Por exemplo, diretórios têm um formato definido, com um número *inodes* e nomes ASCII. Se um número *inode* for maior que o número de *inodes* no disco, o diretório encontra-se num estado inconsistente.

3.4 Desempenho do Sistema de Arquivos

Um acesso a disco é muito mais lento que um acesso a memória. Ler uma palavra da memória leva tipicamente algumas centenas de nanosegundos. Ler um bloco do disco requer alguns milisegundos, um fator 100.000 vezes mais lento. Como resultado, muitos sistemas de arquivos têm sido projetados para reduzir o número necessário de acessos a disco.

A técnica mais comum para reduzir o acesso a disco é a *block cache* ou *buffer cache*. Neste contexto, uma *cache* é uma coleção de blocos que pertencem logicamente ao disco, mas são mantidos na memória por razões de desempenho.

Vários algoritmos podem ser usados para gerenciar o *cache*, mas o mais comum é o que verifica todas as requisições de leitura para ver se o bloco referido está na *cache*. Se estiver, a requisição de leitura pode ser satisfeita sem acesso a disco. Se o bloco não estiver na *cache*, ele é inicialmente lido para a *cache*, e então copiado para a área do processo que requisitou o acesso. Requisições subsequentes do mesmo bloco podem ser satisfeitas através da *cache*.

Quando um bloco tem que ser carregado para uma *cache* cheia, algum bloco terá que ser removido e reescrito no disco, caso tenha sido modificado desde o instante em que foi instalado na *cache*. Esta situação é muito parecida com a paginação, e todos os algoritmos usuais de paginação, tal como o "menos recentemente usado" (LRU) podem ser aplicados neste contexto.

Se um bloco for essencial para a consistência do sistema de arquivos (basicamente tudo, exceto blocos de dados), e foi modificado na *cache*, é necessário que o mesmo seja escrito no disco imediatamente. Escrevendo blocos críticos rapidamente no disco, reduzimos a probabilidade que falhas danifiquem o sistema de arquivos.

Até mesmo com estas medidas para manter a integridade do sistema de arquivos, é indesejável manter blocos de dados na *cache* durante muito tempo antes que sejam descarregados em disco. Os sistemas de arquivos adotam duas estratégias para tal. No UNIX, a chamada de sistema `sync`, força com que todos os blocos modificados sejam gravados em disco imediatamente. Quando o sistema é iniciado, um programa, usualmente chamado *update*, é ativado. De 30 em 30 segundos, a atualização da *cache* é estabelecida. Como resultado, na pior hipótese, perde-se os blocos gravados nos últimos 30 segundos em caso de uma pane.

A solução do MS-DOS é gravar todo bloco modificado para o disco tão logo tenha sido escrito. *Caches* nas quais blocos modificados são reescritos imediatamente no disco são chamadas *caches de escrita direta*. Elas requerem muito mais E/S de disco que *caches* de escrita não direta. A diferença entre estas duas técnicas pode ser vista quando um programa escreve num buffer de 1K, caracter por caracter. O UNIX coleta todos os caracteres da *cache*, e escreve o bloco de uma vez em 30 segundos, ou quando o bloco for removido da *cache*.

O MS-DOS faz acesso a disco para cada um dos caracteres escritos. Naturalmente, muitos programas fazem "buferização" interna, procedendo gravações em disco apenas quando existir uma determinada quantidade de bytes pendentes. A estratégia adotada pelo MS-DOS foi influenciada pela garantia que a remoção de um disco flexível de sua unidade não causa perda de dados. No UNIX, é necessária uma chamada `sync` antes da remoção de qualquer meio de armazenamento (ou da parada programada do sistema).

Cache não é a única maneira de aumentar o desempenho do sistema de arquivos. Uma outra maneira é reduzir a quantidade de movimentos do braço do disco, colocando blocos que estão sendo acessados em sequência, preferencialmente em um mesmo cilindro. Quando um arquivo é escrito, o sistema de arquivos aloca os blocos um por vez, a medida do necessário. Se os blocos livres estiverem gravados em um mapa de bits, e o mapa de bits inteiro está na memória principal, é fácil escolher um bloco que está mais perto do bloco anterior. Com uma lista de blocos livres, parte da qual está no disco, é mais difícil alocar blocos próximos.

Entretanto, com uma lista de blocos livres alguns agrupamentos de blocos podem ser feitos. O artifício é manter a trilha do disco armazenada não em blocos, mais em grupos consecutivos de blocos. Se uma trilha consistir de 64 setores de 512 bytes, o sistema pode usar blocos de 1K bytes (2 setores), porém alocando espaço no disco em unidades de 2 blocos (4 setores). Isto não é o mesmo que ter um bloco de 2K, posto que na *cache* ainda se usa blocos de 1K com transferência para disco também de 1K. Entretanto, a leitura sequencial reduz o número de busca de um fator de 2, melhorando consideravelmente o desempenho.

Outra variante é fazer uso do posicionamento rotacional. Quando se alocam blocos, o siste-

ma atenta para colocar blocos consecutivos de um arquivo no mesmo cilindro, mas intercalados. Deste modo, se um disco tiver um tempo de rotação de 16.67 mseg e 4 mseg são necessários para o processo do usuário requerer e acessar um bloco do disco, cada bloco deve ser colocado em ao menos um quarto da distância do seu antecessor.

Um outro agravante no desempenho dos sistemas que usam *inodes*, ou algo similar, é que para ler até mesmo um pequeno arquivo, seja necessário 2 acessos no disco: um para o *inode* e outro para o bloco. Caso todos os *inodes* estejam próximos do início do disco, distância média entre o *inode* estará em torno da metade do número de cilindros, o que exige longas buscas.

Melhor alternativa é instalar os *inodes* no meio do disco, reduzindo a média de busca entre o *inode* e o primeiro bloco de um fator de 2. Uma outra idéia consiste em dividir o disco em grupos de cilindros, cada qual com os seus próprios *inodes*, blocos, e lista de blocos livres. Quando se cria um novo arquivo, qualquer *inode* pode ser escolhido, mas tendo-se o cuidado de achar um bloco no mesmo grupo de cilindros onde o *inode* está. Caso nenhum bloco esteja disponível, escolhe-se um bloco do cilindro mais próximo.

3.5 O Sistema de Arquivos do UNIX (System V)

O função do núcleo do ponto de vista do sistema de arquivos consiste em permitir que os processos armazenem novas informações ou que recuperem informações previamente armazenadas [3, 2]. Para a realização destas atividades o núcleo necessita trazer informações auxiliares para a memória. Como exemplo, podemos destacar o super-bloco, o qual descreve, dentre outras informações, a quantidade de espaço livre disponível no sistema de arquivos e os *inodes* disponíveis. Estas informações são manipuladas de forma transparente para os processos.

Com o objetivo de minimizar a frequência de acesso ao disco o núcleo gerencia um *pool* interno de buffers denominado de *cache de buffers*. Neste caso, trata-se de uma estrutura de *software* que não deve ser confundida com a memória *cache* em *hardware* que é utilizada para acelerar as referências à memória.

3.5.1 O Cache de Buffers

O número de buffers que fazem parte do *cache de buffers* depende do tamanho da memória e das restrições de desempenho impostas, sendo o espaço necessário para armazená-los alocado pelo núcleo durante a iniciação do sistema. Cada buffer é formado de duas partes: um segmento de memória utilizado para abrigar os dados que são lidos/escritos da/na memória e um cabeçalho que identifica o buffer.

A visão que o núcleo possui do sistema de arquivos no disco é uma visão lógica. O mapeamento desta visão lógica na estrutura física do disco é realizada pelos *drivers* dos dispositivos. Dentro desta visão, o núcleo enxerga o sistema de arquivos como sendo formado por vários blocos de disco; no caso, um buffer corresponde a um bloco, sendo o seu conteúdo identificado pelo núcleo através de um exame dos campos presentes no cabeçalho associado ao buffer. Podemos concluir que o buffer é a cópia na memória de um bloco de disco, cópia esta que é mantida até o instante em que o núcleo decide mapear um outro bloco no buffer. Uma restrição importante é a de que um bloco do disco não pode ser mapeado em mais de um buffer ao mesmo tempo.

O cabeçalho de um buffer contém 4 classes de informação (ver Fig. 3.9): identificação, posicionamento, estado e posição dentro do segmento de memória. A identificação permite reconhecer a qual bloco do disco o buffer está associado. O cabeçalho do buffer contém um campo com o número do dispositivo e um campo com o número do bloco. Estes campos

especificam o sistema de arquivo e o número do bloco de dado no disco identificando de forma única o buffer. Com relação ao posicionamento, o cabeçalho possui apontadores para posicionar o buffer em duas estruturas: lista de buffers livres e a fila *hash*. O cabeçalho possui um apontador para a área de dado (segmento) correspondente ao buffer. Deve ser observado que o tamanho desta área deve ser igual ao tamanho do bloco do disco. O quarto campo é o campo de status que indica a situação do buffer, podendo ter um dos seguintes valores:

- buffer trancado (*locked*);
- buffer contendo dados válidos;
- buffer com escrita retardada - significa a situação em que o núcleo deve escrever o conteúdo do buffer no disco antes de reatribuir o buffer a um outro bloco;
- o núcleo encontra-se lendo ou escrevendo o conteúdo do buffer no disco;
- um processo encontra-se aguardando que o buffer torne-se livre.

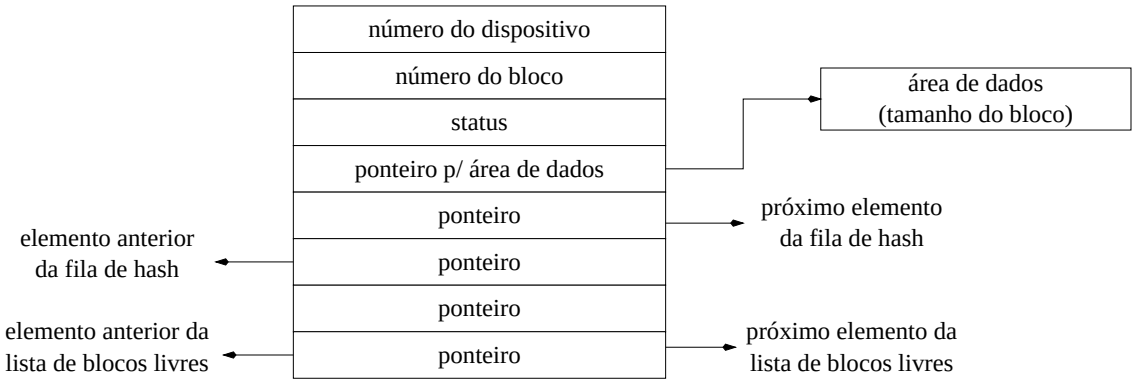

Fig. 3.9: Cabeçalho do buffer

Estrutura do Pool de Buffers

Os blocos de dado no disco são organizados no *cache de buffers* através de uma política denominada de "uso menos recente". Através desta política, quando um buffer é alocado a um determinado bloco, este buffer não poderá ser realocado a um outro bloco, a menos que todos os outros blocos tenham sido usados mais recentemente. Para implementação desta política o núcleo mantém uma lista de buffers livres que é organizada segundo a ordem do uso menos recente.

O núcleo retira um buffer da cabeça da lista quando da necessidade de um buffer livre. É possível que um buffer seja retirado do meio da lista caso o núcleo identifique um bloco específico no *cache de buffers*. Em ambos os casos o buffer é removido da lista. Quando o núcleo retorna um buffer ao pool, ele normalmente coloca o buffer no final da lista sendo que, ocasionalmente (em situações de erro), o buffer é colocado na cabeça da lista. O buffer nunca é recolocado no meio da lista de buffers livres.

Quando o núcleo necessita acessar um bloco do disco, ele procura um buffer que possua a combinação adequada *número de dispositivo-número de bloco*. Para evitar a necessidade do núcleo pesquisar o *cache de buffers* por completo, estes são organizados em filas separadas,

espalhadas por uma função de *hash* que tem como parâmetros os números do dispositivo e do bloco. Os buffers são colocados em uma fila *hash* circular, duplamente ligada, em uma forma equivalente à estrutura da lista de buffers livres.

Todo buffer está na fila *hash*, não existindo, entretanto, significado para a sua posição na fila. Um buffer pode encontrar-se, simultaneamente, na lista de buffers livres, caso o seu estado seja livre, e na fila *hash* (por exemplo, o buffer 64 da Fig. 3.10). Desta forma, o núcleo pode procurar um buffer na lista *hash* caso ele esteja procurando um buffer específico, ou ele pode remover um buffer da lista de buffers livres caso ele esteja procurando por um buffer livre qualquer. Resumindo, um buffer que se encontra na lista *hash* pode ou não encontrar-se na lista de buffers livres.

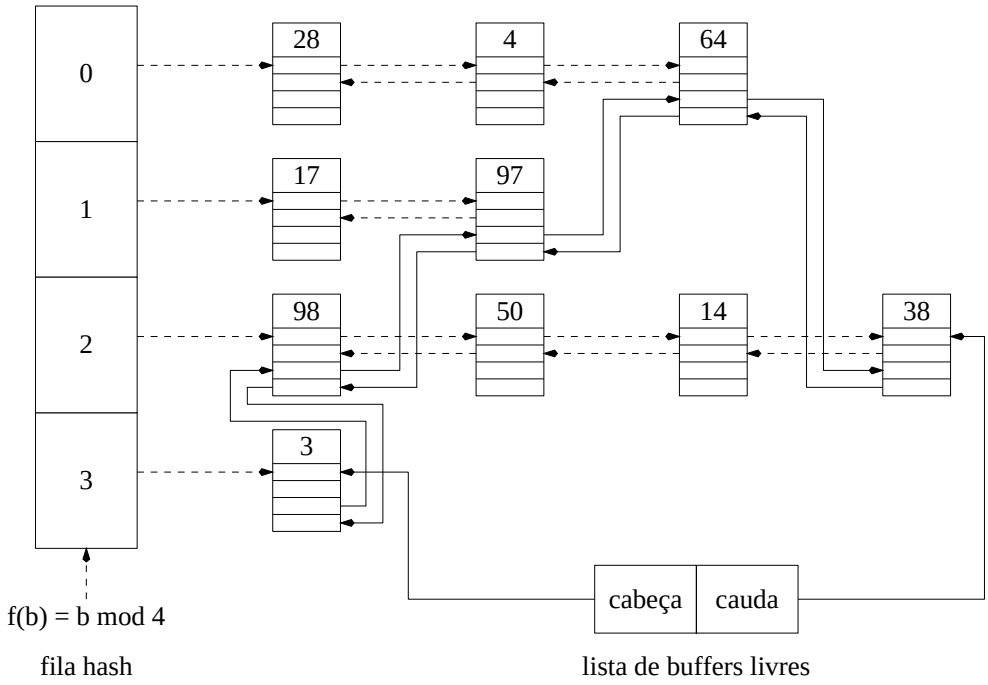

Fig. 3.10: Estrutura do *cache de buffers*: fila de *hash* e lista de buffers livres

Cenários para Recuperação de um Buffer

O *cache de buffers* é manipulado por um conjunto de algoritmos que são invocados pelo sistema de arquivos. O sistema de arquivos determina o número do dispositivo lógico e o número do bloco a ser acessado, por exemplo, quando um processo necessita ler dados de um arquivo.

Quando da leitura ou escrita de dados de um determinado bloco do disco, o núcleo determina se o bloco encontra-se no *cache de buffers* e, caso ele não se encontre, um buffer livre é atribuído ao bloco. Os algoritmos para leitura/escrita em blocos do disco usam um algoritmo `(getblk)` para alocação de buffers a partir do pool.

Existem cinco cenários típicos associados ao algoritmo `getblk`:

1. O núcleo encontra o buffer correspondente ao bloco na fila *hash* e o buffer está livre. Neste caso, o algoritmo marca o buffer como "ocupado", remove-o da lista de buffers livres e retorna um ponteiro para o buffer.

2. O núcleo não encontra o buffer correspondente ao bloco na fila *hash* e aloca um buffer da lista de buffers livres, reposicionando-o na fila de *hash*.

3. O núcleo não encontra o buffer correspondente ao bloco na fila *hash* e, na tentativa de alocar um buffer da lista de buffer livre (como no cenário 2), encontra um buffer na lista de buffers livres marcado com escrita adiada *(delayed write)*. Neste caso, o núcleo deve escrever (assincronamente) o buffer no disco e continuar a procura de um buffer livre para o bloco.

4. O núcleo não encontra o buffer correspondente ao bloco na fila *hash*, e a lista de buffers livres encontra-se vazia. Neste caso, o processo é bloqueado até que outro processo devolva um buffer à lista de buffers livres.

5. O núcleo encontra o buffer correspondente ao bloco na fila *hash*, mas o seu buffer está ocupado (outro processo está acessando exatamente este bloco). Neste caso, o processo é bloqueado até que o outro processo conclua a operação sob este bloco.

Note que em qualquer cenário acima, ou o algoritmo `getblk` volta um buffer já posicionado na fila de *hash* para que a operação de E/S possa continuar; ou bloqueia o processo a espera de buffer ou liberação do bloco.

Além do algoritmo `getblk`, existem outros quatro algoritmos que operam o *cache de buffers*:

1. `brelse`: libera um bloco, retornando-o à lista de blocos livres.

2. `bread`: lê sincronamente (com espera) o bloco do disco para o buffer.

3. `breada`: lê assincronamente (sem espera) o bloco do disco para o buffer.

4. `bwrite`: escreve um bloco do buffer para o disco.

3.5.2 Representação Interna dos Arquivos

Todo arquivo no UNIX *System* V contém um único *inode*. O *inode* possui as informações necessárias para um processo acessar um arquivo, tais como: proprietário do arquivo, direito de acesso, tamanho do arquivo e localização dos dados (blocos) do arquivo. A referência a um arquivo é feita pelo seu nome e, através deste, o núcleo determina o *inode* do arquivo.

Os algoritmos envolvidos na manipulação dos *inodes* resumidos a seguir utilizam os algoritmos que operam sobre o *cache de buffers* discutidos anteriormente.

- `iget`: retorna um *inode* previamente identificado, possivelmente através da sua leitura do disco via o *cache de buffers*;

- `iput`: libera o *inode*;

- `bmap`: define os parâmetros do núcleo para o acesso a um arquivo;

- `namei`: converte um nome (*path*) de arquivo no *inode* correspondente;

- `alloc`: aloca blocos do disco para os arquivos;

- `free`: libera blocos de disco;

- `ialloc`: aloca *inodes* para os arquivos;
- `ifree`: libera *inodes*.

Estes algoritmos utilizam outros que manipulam o *cache de buffers* (`getblk, brelse, bread, breada, bwrite`).

Estrutura do Inode

Um *inode* existe estaticamente no disco e o núcleo realiza a sua leitura para a memória quando necessita manipulá-lo. O *inode* no disco contém os seguintes campos (ver Fig. 3.4):

- identificador do dono do arquivo: divido em dono individual e grupo;
- tipo do arquivo: regular, diretório, especial ou FIFO (pipes);
- permissão de acesso;
- instantes de acesso ao arquivo: última modificação, último acesso e última modificação ocorrida no *inode*;
- número de conexões *(links)* associados ao arquivo;
- endereços no disco dos blocos de dados do arquivo;
- tamanho do arquivo.

A cópia do *inode* em memória contém os seguintes campos em adição aos campos do *inode* em disco:

- status do *inode* na memória indicando:

  - o *inode* está trancado;

  - processo encontra-se esperando que o *inode* seja liberado;

  - a representação do *inode* na memória difere da cópia do disco devido a mudanças nos dados do *inode*;

  - a representação do arquivo na memória difere da cópia do disco devido a mudanças nos dados do arquivo;

  - o arquivo é um *mount point*.

- o número do dispositivo lógico do sistema de arquivos que contém o arquivo;
- o número do *inode*. O *inode* no disco não necessita deste número pois os *inodes* são armazenados em um arranjo linear no disco;
- apontadores para outros *inodes* na memória. O núcleo liga os *inodes* em filas *hash* e em uma lista de *inodes* livres da mesma forma que os buffers são ligados no *cache de buffers*;
- um contador de referência, indicando o número de instâncias do arquivo que estão ativas.

A diferença mais significativa entre o *inode* na memória e o cabeçalho do buffer no *cache de buffers* é o contador de referência presente no *inode*, o qual indica o número de instâncias (acessos) ativas do arquivo. Um *inode* torna-se ativo quando um processo realiza a sua alocação, por exemplo, através da abertura do arquivo. Um *inode* é colocado na lista de *inodes* livres se, e somente se, o seu contador de referência é 0, significando que o núcleo pode realocar sua estrutura na memória a um outro *inode* do disco. A lista de *inodes* livres serve como um *cache de inodes* (exatamente como o *cache de buffers*).

Acesso aos Inodes

O núcleo identifica um *inode* específico através do número do seu sistema de arquivo e do seu número dentro deste sistema. O algoritmo `iget` cria uma cópia do *inode* na memória física, realizando um papel equivalente ao algoritmo `getblk` utilizado para encontrar um bloco do disco no *cache de buffers*.

Caso o algoritmo não encontre o *inode* na fila *hash* associada ao número do dispositivo e ao número do *inode* procurado, um *inode* é alocado a partir da lista de *inodes* livres sendo consequentemente trancado *(locked)*. A partir deste ponto o núcleo encontra-se em condição de ler o *inode* do disco correspondente ao arquivo que esta sendo referenciado e copiá-lo para a memória física. Para determinar o bloco do disco que contém o *inode* referenciado e para determinar o *offset* do *inode* dentro do bloco, o núcleo utiliza as seguintes expressões:

- *nº* do bloco: ((*nº* do *inode*-1) / número de *inodes* por bloco) + número do bloco inicial da lista de *inodes*
- *offset*: ((*nº* do *inode*-1) MOD (número de *inodes* por bloco) * tamanho do *inode* no disco

Quando o núcleo libera um *inode* (algoritmo `iput`) ele decrementa seu contador de referência. Caso o valor do contador se torne 0, o núcleo escreve o *inode* no disco caso a cópia na memória seja diferente da cópia no disco. O *inode* é colocado na lista de *inodes* livres na hipótese de que este *inode* possa ser necessário posteriormente.

Estrutura de um Arquivo Regular

A indicação dos blocos do disco que constituem um determinado arquivo encontra-se no *inode* associado ao arquivo. Esta indicação traduz-se na utilização de 13 números para blocos. Os 10 primeiros números (blocos diretos) contém números para blocos de disco; o 11º número é um indireto simples, o 12º um indireto duplo e o 13º um indireto triplo. Esta solução, conforme discutido anteriormente, permite que os 10 primeiros números componham um arquivo de 10 Kbytes (supondo blocos de 1 Kbytes). Elevando-se este número para 11, 12 e 13, tem-se, respectivamente, arquivos de até 256 Kilobytes, 64 Megabytes e 16 Gigabytes (máximo para o UNIX).

Os processos enxergam o arquivo como um conjunto de bytes começando com o endereço 0 e terminando com o endereço equivalente ao tamanho do arquivo menos 1. O núcleo converte esta visão dos processos em termos de bytes em uma visão em termos de blocos: o arquivo começa com o bloco 0 (apontado pelo primeiro número de bloco no *inode*) e continua até o número de bloco lógico correspondente ao tamanho do arquivo.

Um exame mais detalhado das entradas dos blocos no *inode* pode mostrar que algumas entradas possuem o valor 0, indicando que o bloco lógico não contém dados. Isto acontece caso nenhum processo tenha escrito dados no arquivo em qualquer *offset* correspondente a estes

blocos. Deve ser observado que nenhum espaço em disco é desperdiçado para estes blocos. Este *layout* pode ser criado pelo uso das chamadas `lseek` e `write`.

Estrutura do Diretório

Quando da abertura de um arquivo, o sistema operacional utiliza o nome *(path)* do arquivo fornecido pelo usuário para localizar os blocos do disco associados ao arquivo. O mapeamento do nome nos *inodes* está associado à forma como o sistema de diretório encontra-se organizado.

A estrutura de diretório usada no UNIX é extremamente simples. Cada entrada contém o nome do arquivo e o número do seu *inode*, representados através de 16 bytes, ou seja, 2 bytes para o número do *inode* e 14 bytes para o nome[3]. Todos os diretórios do UNIX são arquivos e podem conter um número arbitrário destas entradas de 16 bytes.

Quando um arquivo é aberto, o sistema deve, através do nome do arquivo, localizar os seus blocos no disco. Caso o nome do arquivo seja relativo, isto é, associado ao diretório corrente, o núcleo acessa o *inode* associado ao diretório corrente na área U do processo e realiza um procedimento semelhante ao exemplo da subseção 3.2.

3.5.3 Atribuição de *inodes* e Blocos

Até o momento consideramos a hipótese de que um *inode* havia sido previamente associado a um arquivo e que os blocos do disco já continham os dados. Será discutido na sequência como o núcleo atribui *inodes* e blocos do disco. Para isto é importante conhecer a estrutura do super-bloco.

O Super-Bloco

O super-bloco é um segmento contínuo de disco formado pelos seguintes campos:

- tamanho do sistema de arquivos;
- número de blocos livres no sistema;
- lista de blocos livres disponíveis no sistema de arquivos;
- índice do próximo bloco livre na lista de blocos livres;
- tamanho da lista de *inodes*;
- número de *inodes* livres no sistema de arquivos;
- lista de *inodes* livres no sistema de arquivos;
- índice do próximo *inode* livre na lista de *inodes*;
- campos trancados *(locked)* para as listas de blocos livres e *inodes* livres;
- *flag* indicando que o super-bloco foi modificado.

O super-bloco é mantido na memória física de modo a agilizar as operações sobre o sistema de arquivos, sendo periodicamente copiado no disco para manter a consistência do sistema de arquivos.

[3]Versões correntes do UNIX permitem nomes bem mais extensos.

Atribuição de um Inode a um Novo Arquivo

O algoritmo `ialloc` atribui um *inode* do disco para um arquivo recém criado. O sistema de arquivos contém uma lista linear de *inodes*. Um *inode* nesta lista encontra-se livre quando o seu campo de tipo é zero. Para melhorar o desempenho, o super-bloco do sistema de arquivos contém um arranjo que atua como um *cache* no qual são armazenados os *inodes* livres do sistema de arquivos Não confundir esta lista com aquela para abrigar os *inodes* livres no pool de *inodes* em memória. Aquela está relacionada à manipulação de *inodes* associados a arquivos já criados e que serão ativos para manipulação por parte dos processos. Esta lista de *inodes* livres associada ao super-bloco abriga os *inodes* no disco que não estão alocados a nenhum arquivo e que poderão ser associados aos arquivos que serão criados no sistema.

Alocação de Blocos no Disco

Quando um processo necessita escrever dados em um arquivo, o núcleo aloca blocos do disco para a expansão do arquivo, o que significa associar novos blocos ao *inode*. Para agilizar a determinação de quais blocos do disco estão disponíveis para serem alocados ao arquivo, o super-bloco contém um arranjo que relaciona o número de blocos do disco que estão livres. Este arranjo utiliza blocos de disco cujo conteúdo são números de blocos livres. Uma das entradas destes blocos constitui-se de um apontador para outro bloco contendo números de blocos livres. Estabelece-se, desta forma, uma lista ligada (em disco) de blocos que contém números de blocos livres. O super-bloco possui parte desta lista ligada, de tamanho máximo correspondendo a um bloco, sendo gerenciada quando da liberação de blocos e quando da requisição de blocos de modo a manter sempre uma indicação de parte dos blocos livres no disco. O algoritmo `alloc` realiza a alocação de um bloco do sistema de arquivos alocando o próximo bloco disponível na lista do super-bloco.

3.5.4 Chamadas de Sistema Referentes ao Sistema de Arquivos

As chamadas de sistema que compõem o sistema de arquivos utilizam, dentre outros, os algoritmos descritos acima (`getblk, ialloc`, etc.). Cada arquivo no UNIX é identificado por um *descritor de arquivos* (um número inteiro). Descritores de arquivos identificam univocamente arquivos no nível de processo, isto é, dois arquivos abertos pelo mesmo processo possuem descritores necessariamente diferentes.

As principais chamadas de sistema referentes ao sistema de arquivos serão sucintamente descritas a seguir.

1. `open`: abre um arquivo para leitura/gravação. Parâmetros da chamada indicam, por exemplo, se o arquivo deve ser criado (caso inexista) e permissões.

2. `dup`: duplica um descritor de arquivo, retornando um novo descritor associado ao mesmo arquivo.

3. `pipe`: cria no sistema de arquivos um *pipe*. Pipes são arquivos de uso exclusivo por parte de dois processos: um produtor e um consumidor de dados.

4. `close`: encerra a operação sobre um arquivo aberto.

5. `link`: cria uma conexão simbólica para um arquivo já existente.

6. `unlink`: remove uma conexão simbólica.
7. `read`: lê dados de um arquivo (para um buffer em memória).
8. `write`: escreve dados num arquivo (de um buffer em memória).
9. `lseek`: altera a posição corrente de leitura/gravação no arquivo.
10. `mknod`: cria um arquivo especial (tipo bloco ou caractere), associando-o a periférico (unidade de disco, fita, terminal, etc).
11. `mount`: adiciona uma nova unidade lógica ao sistema de arquivos.
12. `umount`: remove uma unidade lógica do sistema de arquivos.
13. `chdir`: altera o diretório corrente na área U.
14. `chown`: altera o usuário que têm a posse do arquivo.
15. `chmod`: altera permissões de um arquivo.
16. `stat`: fornece informações sobre um arquivo (tipicamente as constantes no *inode* do arquivo).