Capítulo 8

Sistemas de Arquivos

Todo aplicativo de computador precisa armazenar e recuperar informações. Enquanto um processo está executando, ele pode armazenar uma quantidade limitada de informações dentro de seu próprio espaço de endereços. Entretanto, a capacidade de armazenamento é limitada ao tamanho do espaço de endereço virtual. Para alguns aplicativos, esse tamanho é adequado, mas para outros, como reservas de linha aérea, sistemas bancários ou registro geral de grandes corporações, é, de longe, muito pequeno.

Um segundo problema com manter as informações dentro do espaço de endereços de um processo, é que quando o processo termina, as informações são perdidas. Para muitos aplicativos (por exemplo, para bancos de dados), as informações devem ser mantidas durante semanas, meses ou mesmo eternamente. É inaceitável que elas desapareçam quando o processo que as utiliza termina. Além disso, elas não devem desaparecer quando uma falha do computador "mata" o processo.

Um terceiro problema: frequentemente é necessário que múltiplos processos acessem as informações ou parte das informações ao mesmo tempo. Se tivermos uma lista de telefones on-line armazenada no espaço de endereços de um único processo, somente aquele processo pode acessá-la. A maneira de resolver esse problema é tornar as próprias informações independentes de qualquer processo.

Assim, temos três exigências essenciais para armazenamento de informações de longo prazo:

- Deve ser possível armazenar uma grande quantidade de informações.
- A informação deve sobreviver à finalização do processo que a utiliza.
- Múltiplos processos devem ser capazes de acessar as informações concorrentes.

A solução normal para todos esses problemas é armazenar as informações em discos e em outras mídias externas, em unidades chamadas arquivos. Os processos, então, podem lê-los e gravar novas informações se necessário. As informações armazenadas em arquivos devem ser persistentes, isto é, não devem ser afetadas pela criação e pela finalização de

processos. Um arquivo deve desaparecer apenas quando seu proprietário explicitamente removê-lo.

Os arquivos são gerenciados pelo sistema operacional. O modo como eles são estruturados, nomeados, acessados, utilizados, protegidos e implementados constitui temas importantes no projeto de um sistema operacional. Como um todo, a parte do sistema operacional que lida com arquivos é conhecida como o sistema de arquivos.

Do ponto de vista do usuário, o aspecto mais importante de um sistema de arquivos é como aparece para ele, isto é, o que constitui um arquivo, como os arquivos são nomeados e protegidos, que operações são permitidas em arquivos e assim por diante. Os detalhes como listas encadeadas ou mapas de bits são utilizados para registrar o espaço livre e quantos setores estão em um bloco lógico são de interesse menor, embora sejam de grande importância para os projetistas do sistema de arquivos.

Assim como o DOS estaremos trabalhando com a FAT 16, ou seja, os arquivos usando o sistema de 16 bits, ao contrário do Windows que usa a FAT 32 ou ainda NTFS.

Todos os arquivos do sistema podem ser acessados por outros sistemas operacionais, como o Windows, Linux, DOS, entre outros.

8.1 Arquivos

Nos itens seguintes, veremos arquivos do ponto de vista do usuário, isto é, como eles são utilizados e quais são suas propriedades.

8.1.1 Nomes de Arquivos

Os arquivos são um mecanismo de abstração. Oferecem uma maneira de armazenar as informações no disco e de lê-las de volta mais tarde. Isso deve ser feito de tal maneira que esconda do usuário os detalhes de como e onde as informações são armazenadas e de como os discos realmente trabalham.

Provavelmente, a mais importante característica de qualquer mecanismo de abstração é a maneira como são nomeados os objetos que estão sendo gerenciados. Assim, iniciaremos nosso exame dos sistemas de arquivos com o tema atribuição de nomes de arquivos. Quando um processo termina, a arquivo continua a existir e a poder ser acessado por outros processos, utilizando seu nome.

As regras exatas para nomes de arquivos variam um pouco de um sistema para outro, mas todos os sistemas operacionais permitem cadeias de uma a oito letras como nomes de arquivos válidos. Assim andrea, bruce e cathy são nomes possíveis de arquivo. Freqüentemente, algarismos e caracteres especiais também são permitidos, então, nomes como 2, urgente! e Figura 2-14 freqüentemente são válidos também. Muitos sistemas de arquivos suportam nomes com até 255 caracteres.

Alguns sistemas de arquivos distinguem entre nomes escritos com letras maiúsculas e minúsculas, enquanto outros não o fazem. O Unix entra na primeira categoria; o MS-DOS

entra na segunda. Assim um sistema Unix pode ter todos os seguintes nomes como arquivos distintos: barbara, Barbara, BARBARA, BARbara e BarBaRa. No MS-DOS, todos eles designam o mesmo arquivo.

Muitos sistemas operacionais suportam nomes de arquivos de duas partes, ambas separadas por um ponto, como em prog.c. A parte que se segue ao ponto é chamada extensão de arquivo e normalmente indica algo sobre o arquivo. No MS-DOS, por exemplo, nomes de arquivo têm de 1 a 8 caracteres, com mais uma extensão opcional de 1 a 3 caracteres. No Unix, o tamanho da extensão, se houver uma, é o usuário quem determina, e um arquivo pode ter até duas ou mais extensões, como em prog.c.Z, onde Z é comumente utilizado para indicar que o arquivo (o prog.c) foi compactado, utilizando o algoritmo de compactação de Ziv-Lempel.

Em alguns casos, as extensões de arquivos são apenas convenções e não são necessariamente impostas. Um arquivo chamado arquivo.txt é provavelmente algum tipo de arquivo de texto, mas esse nome é mais para lembrar ao proprietário do que é para carregar quaisquer informações específicas para o computador. Por outro lado, m compilador C, ao compilar, pode realmente insistir em que os arquivos sejam terminados em .c e pode recusar-se a compilá-los se essa exigência não for atendida.

Convenções como essa são especialmente úteis quando o mesmo programa pode tratar vários tipos diferentes de arquivos. O compilador C, por exemplo, pode receber uma lista de vários arquivos a compilar e a vincular juntos, sendo alguns deles arquivos em C e, outros, arquivos em linguagem assembly. A extensão, então, torna-se essencial para informar ao compilador quais são arquivos em C, quais são em assembly e quais são outros arquivos.

8.1.2 Estruturas de Arquivos

Os arquivos podem ser estruturados de várias maneiras. Três possibilidades comuns são representadas na Figura A-1. O arquivo na Figura A-1(a) é uma seqüência de bytes não estruturada. Com efeito, o sistema operacional não sabe nem se importa com o que está no arquivo. Tudo que ele vê são bytes. Qualquer significado deve ser imposto por programas no nível do usuário. Tanto o Unix como o MS-DOS utilizam essa abordagem. A propósito, o Windows 95 utiliza basicamente o sistema de arquivos do MS-DOS, com uma pequena adição sintática (p. ex., nomes de arquivos longos). Então, quase tudo dito neste capítulo sobre o MS-DOS também se aplica ao Windows 95. O Windows NT, porém, é completamente diferente.

Ter o sistema operacional considerando arquivos como nada mais do que seqüência de bytes oferece um máximo de flexibilidade. Os programas de usuários podem colocar qualquer coisa que quiserem em arquivos e nomeá-los de qualquer maneira que lhe seja conveniente. O sistema operacional não ajuda, mas também não atrapalha. Para usuários que querem fazer coisas incomuns, este último aspecto pode ser muito importante.

O primeiro passo na estrutura é mostrado na Figura A-1(b). Neste modelo, um arquivo é uma seqüência de registros de comprimento fixo, cada um com alguma estrutura interna. O cerne da idéia de que um arquivo é uma seqüência de registros está no fato de que a

operação de leitura retorna um registro, e as operações de gravação sobrescrevem ou anexam um registro. Antigamente, quando reinavam os cartões perfurados de 80 colunas, muitos sistemas operacionais baseavam seus sistemas de arquivos em arquivos que consistiam em registros de 80 caracteres, que, de fato, representavam imagens de cartões. Esses sistemas também suportavam arquivos de registro com 132 caracteres, que foram projetados para impressoras de linha (que nesse tempo eram grandes impressoras de cadeia com 132 colunas). Os programas liam a entrada em unidades de 80 caracteres e gravavam em unidades de 132 caracteres, embora os 52 finais pudessem ser espaços, naturalmente.

Um (antigo) sistema que via arquivos como seqüências de registros de comprimento fixo era o CP/M. Ele utilizava um registro de 128 caracteres. Hoje em dia, a idéia de um arquivo como uma seqüência de registros de comprimento fixo foi completamente abandonada, embora um dia tenha sido a norma.

O terceiro tipo de estrutura de arquivos é mostrado na Figura A-1(c). Nessa organização, um arquivo consiste em uma árvore de registros, não necessariamente todos do mesmo comprimento, cada um contendo um campo-chave em uma posição fixa no registro. A árvore é classificada pelo campo-chave, permitindo localizar rapidamente uma chave particular.

A operação básica aqui não é obter o "próximo" registro, embora isso também seja possível, mas obter o registro com uma chave específica. Para o arquivo do zoológico da Figura A-1(c), poderia ser solicitado que o sistema obtivesse o registro cuja chave é potro, por exemplo, sem se preocupar com sua posição exata no arquivo. Além disso, novos registros podem ser adicionados ao arquivo, com o sistema operacional e não com o usuário, decidindo onde colocá-los. Esse tipo de arquivo é claramente bem diferente dos fluxos de byte não estruturados utilizados no Unix e no MS-DOS, mas é amplamente utilizado nos mainframes de grande porte ainda usados em algum processamento comercial de dados.

entra fig_08_01.jpg

Figura 8.1 – Entra legenda da figura[RP1].

8.1.3 Tipos de Arquivos

Muitos sistemas operacionais suportam vários tipos de arquivos. O Unix e o MS-DOS, por exemplo, têm arquivos e diretórios comuns. O Unix também tem arquivos de caractere e de bloco especiais. Arquivos comuns são os que contêm informações do usuário. Todos os arquivos da Figura A-1 são comuns. Diretórios são arquivos de sistema para manter a estrutura do sistema de arquivos. Estudaremos diretórios a seguir. Arquivos especiais de caractere relacionam-se com a entrada/saída e são utilizados para modelar dispositivos de E/S seriais como terminais, impressoras e redes. Arquivos especiais de bloco são utilizados para modelar discos. Estaremos interessados principalmente em arquivos comuns.

Arquivos comuns são geralmente arquivos ASCII ou arquivos binários. Os arquivos ASCII consistem em linhas de texto. Em alguns sistemas, cada linha é terminada por um caractere de retorno de carro. Em outro, o caractere de quebra de linha é utilizado.

Ocasionalmente, ambos são exigidos. As linhas não necessitam ser todas do mesmo comprimento.

A grande vantagem de arquivos ASCII é que podem ser exibidos e impressos como são e podem ser editados com um editor de texto comum. Além disso, se um grande número de programas utiliza arquivos ASCII para entrar e saída, é fácil conectar a saída de um programa à entrada de outro, como em canalizações (pipelines) do shell. (O "encanamento" interprocesso não é nada fácil, mas interpretar a informação certamente é, se uma convenção-padrão, como ASCII, for utilizada para expressá-la.)

Outro tipo são arquivos binários, o que significa simplesmente que eles não são arquivos ASCII. Imprimi-los resulta em uma lista incompreensível cheia de, aparentemente, lixo aleatório. Normalmente, eles têm alguma estrutura interna.

Por exemplo, na Figura A-2(a) vemos um arquivo binário executável simples, obtido de uma versão anterior do Unix. Embora tecnicamente o arquivo seja somente uma seqüência de bytes, o sistema operacional somente executará um arquivo se tiver o formato adequado. Ele tem cinco seções: cabeçalho, texto, dados, bits de realocação e tabela de símbolos. O cabeçalho inicia com o chamado número mágico, identificando o arquivo como um arquivo executável (para evitar a execução acidental de um arquivo que não tenha esse formato). Então, vêm inteiros de 16 bits fornecendo os tamanhos das várias partes do arquivo, o endereço em que a execução inicia e alguns bits de sinalização. Seguindo-se ao cabeçalho estão o texto e dados do próprio programa. Estes últimos são carregados na memória e realocados utilizando os bits de realocação. A tabela de símbolos é utilizada para depuração.

Nosso segundo exemplo de arquivo binário também é um arquivo do Unix. Consiste em uma coleção de procedimentos de biblioteca (módulos) compilados, mas não linkeditados. Cada um deles é prefaciado por um cabeçalho que informa seu nome, sua data de criação, seu proprietário, seu código de proteção e seu tamanho. Assim como com o arquivo executável, os cabeçalhos de módulo estão cheios de números binários. Copiá-los para a impressora produziria simplesmente lixo.

Todo sistema operacional deve reconhecer um tipo de arquivo, seu próprio arquivo executável, mas alguns reconhecem mais. O antigo sistema TOPS-20 ia tão longe a ponto de examinar a data/hora de criação de qualquer arquivo a ser executado. Então, ele localizava o arquivo fonte e via se o fonte tinha sido modificado desde que o binário foi criado. Se tivesse, automaticamente recompilava o fonte. Em termos do Unix, o programa make foi construído dentro do shell. Como as extensões de arquivos eram obrigatórias, o sistema operacional poderia dizer qual programa binário derivava de qual fonte.

Em uma trilha semelhante, quando um usuário do Windows dá um clique duplo em um arquivo, um programa apropriado é carregado com o arquivo como parâmetro. O sistema operacional determina qual programa deve executar com base na extensão do arquivo.

Implementar rigidamente esses tipos de arquivos causa problemas sempre que o usuário faz qualquer coisa que os projetistas de sistema não previram. Considere, por exemplo, um sistema em que os arquivos de saída do programa têm tipo dat (arquivos de dados). Se um usuário escreve um formatador de programa que lê um arquivo .pas, transforma-o (p. ex.,

convertendo-o para um leiaute de alinhamento) e, então, grava o arquivo transformado como saída, o arquivo de saída será do tipo dat. Se o usuário tentar oferecer isso para o compilador Pascal compilá-lo, o sistema recusará porque tem a extensão errada. As tentativas de copiar fle.dat para file.pas serão rejeitadas pelo sistema como inválidas (para proteger o usuário contra erros).

Embora esse tipo de "interface amigável" possa ajudar novatos, ele coloca os usuários experientes contra a parede porque eles precisam dedicar um esforço considerável para contornar a idéia que o sistema operacional tem sobre o que é razoável e o que não é.

entra fig_08_02.jpg

Figura 8.2 – (a) Um arquivo executável. (b) Um arquivo.

8.1.4 Acesso a Arquivos

Os sistemas operacionais antigos ofereciam somente um tipo de acesso a arquivos: acesso seqüencial. Nesses sistemas, um processo poderia ler todos os bytes ou registros de um arquivo em ordem, iniciando no começo, mas não poderia pular e lê-los fora de ordem. Arquivos seqüenciais podem ser retrocedidos, entretanto, podendo, então, ser lidos conforme for necessário. Arquivos seqüenciais são convenientes quando a mídia de armazenamento é fita magnética, em vez de disco.

Quando se começou a utilizar discos para armazenar arquivos, tornou-se possível ler os bytes, ou registros de um arquivo, fora de ordem ou acessar registros por chave, em vez de por posição. Os arquivos cujos bytes ou registros podem ser lidos em qualquer ordem são chamados arquivos de acesso aleatório.

Arquivos de acesso aleatório são essenciais para muitos aplicativos como, por exemplo, sistemas de banco de dados. Se um cliente de linha aérea telefonar e quiser reservar um assento em um determinado vôo, o programa de reserva deve ser capaz de acessar o registro desse vôo sem primeiro ler os registros de milhares de outros vôos.

Dois métodos são utilizados para especificar onde inicia a leitura. No primeiro, cada operação READ dá a posição no arquivo em que deve iniciar a leitura. No segundo, uma operação essencial, SEEK, é oferecida para configurar a posição atual. Depois de um SEEK, o arquivo pode ser lido seqüencialmente a partir da posição atual.

Em alguns antigos sistemas operacionais de mainframes, os arquivos são classificados como tendo acesso seqüencial ou aleatório no momento em que eles são criados. Isso permite que o sistema utilize diferentes técnicas de armazenamento para as duas classes. Sistemas operacionais modernos não fazem essa distinção. Todos os seus arquivos são automaticamente de acesso aleatório.

8.1.5 Atributos de Arquivos

Cada arquivo tem um nome e dados. Além disso, todos os sistemas operacionais associam outras informações com cada arquivo, por exemplo, a data e a hora em que o arquivo foi criado e o tamanho do arquivo. Chamaremos esses itens extras de atributos do arquivo. A

lista de atributos varia consideravelmente de sistema para sistema. A Figura A-3 mostra algumas possibilidades, mas também existem outras. Nenhum sistema existente tem todas essas possibilidades, mas cada uma delas está presente em algum sistema.

Os primeiros quatro atributos relacionam-se com a proteção do arquivo e informam quem pode e quem não pode acessá-lo. Todos os tipos de esquemas são possíveis, alguns estudados mais adiante. Em alguns sistemas, o usuário deve apresentar uma senha para acessar um arquivo, caso em que a senha deve ser um dos atributos.

Os sinalizadores são bits ou campos curtos que controlam ou ativam alguma propriedade específica. Arquivos ocultos, por exemplo, não aparecem em listagens de todos os arquivos. O sinalizador de arquivo é um bit que monitora se o arquivo foi salvo em backup. O programa de backup limpa-o, e o sistema operacional configura-o sempre que um arquivo é modificado. Dessa maneira, o programa de backup pode dizer quais arquivos precisam ser salvos em backup. O sinalizador de temporário permite que um arquivo seja marcado para exclusão automática quando o processo que o criou terminar.

Os campos de comprimento do registro, de posição e de comprimento da chave somente estão presentes em arquivos cujos registros podem ser pesquisados, utilizando uma chave. Elas oferecem as informações exigidas para localizar as chaves.

Os vários campos de tempo monitoram a data e a hora em que o arquivo foi criado, mais recentemente acessado e modificado. São úteis para vários propósitos. Por exemplo, um arquivo fonte que foi modificado após a criação do arquivo objeto correspondente precisa ser recompilado. Esses campos oferecem as informações necessárias.

O tamanho atual informa o tamanho atual do arquivo. Alguns sistemas operacionais de mainframe exigem que o tamanho máximo seja especificado quando o arquivo é criado, deixando o sistema operacional reservar a quantidade máxima de armazenamento de antemão. Sistemas operacionais de estações de trabalho e computadores pessoais são suficientemente inteligentes para prescindir desse recurso.

Campo	Significado
Proteção	Quem pode acessar o arquivo e de que maneira.
Senha	Senha necessária para acessar o arquivo.
Criador	Id da pessoa que criou o arquivo.
Proprietário	Proprietário atual.
Sinalizador de somente leitura	0 para leitura/gravação; 1 para somente leitura.
Sinalizador de oculto	0 para normal; 1 para não exibir em listagens.
Sinalizador de sistema	0 para arquivos normais; 1 para arquivos de sistema.
Sinalizador de arquivo	0 para salvo em backup; 1 para ser salvo em backup.
Sinalizador de ASCII/binário	0 para arquivo ASCII; 1 para arquivo binário.
Sinalizador de acesso aleatório	0 para acesso seqüencial somente; 1 para acesso aleatório.
Sinalizador de temporário	0 para normal; 1 para excluir o arquivo na saída do processo.
Sinalizador de bloqueio	0 para desativado; 1 não-zero para bloqueado.
Comprimento do registro	Número de bytes em um registro.
Posição da chave	Deslocamento da chave dentro de cada registro.

Comprimento da chave	Número de bytes no campo-chave.
Tempo de criação	Data e hora em que o arquivo foi criado.
Tempo do último acesso	Data e hora em que o arquivo foi acessado pela última vez.
Tempo da última alteração	Data e hora em que o arquivo foi alterado pela última vez.
Tamanho atual	Número de bytes no arquivo.
Tamanho máximo	Número de bytes até o qual o arquivo pode crescer.

entra fig_08_03.jpg

Figura 8.3 – Alguns possíveis atributos de arquivo.

8.2 Implementação do Sistema de Arquivos

Provavelmente a questão mais importante ao implementar armazenamento de arquivos é monitora quais blocos de disco acompanham quais arquivos. Vários métodos são utilizados em diferentes sistemas operacionais. Nesta seção, examinaremos alguns deles.

8.2.1 Alocação Contígua

O esquema mais simples de alocação é armazenar cada arquivo como um bloco contíguo de dados no disco. Assim, em um disco com blocos de 1K, um arquivo de 50K alocaria 50 blocos consecutivos. Esse esquema tem duas vantagens significativas. Em primeiro lugar, é simples de implementar porque monitorar onde os blocos de um arquivo estão reduz-se a lembrar um número, o endereço de disco do primeiro bloco. Em segundo lugar, o desempenho é excelente porque o arquivo inteiro pode ser lido do disco em uma única operação.

A alocação contígua, infelizmente, também tem duas desvantagens igualmente significativas. Primeiro: não é praticável a menos que o tamanho máximo do arquivo seja conhecido no momento que em o arquivo é criado. Sem essa informação, o sistema operacional não sabe quanto espaço em disco reservar. Entretanto, em sistemas em que os arquivos devem ser gravados em uma única tacada, ela pode ser utilizada com grande vantagem.

A segunda desvantagem é a fragmentação do disco que resulta dessa política de alocação. É desperdiçado espaço que, de outra maneira, talvez pudesse ser utilizado. A compactação de disco normalmente tem um custo proibitivo, embora concebivelmente possa ser feita à noite, quando o sistema está desocupado.

8.2.2 Alocação por Lista Encadeada

O segundo método para armazenar arquivos é manter cada um como uma lista encadeada de blocos de disco, como mostrado na Figura A-4. A primeira palavra de cada bloco é utilizada como um ponteiro para o seguinte. O resto do bloco é para dados.

Diferentemente da alocação contígua, todos os blocos do disco podem ser utilizados nesse método. Nenhum espaço é desperdiçado em fragmentação de disco (exceto em fragmentação interna do último bloco). Além disso, é suficiente para a entrada de diretório

meramente armazenar o endereço de disco do primeiro bloco. O restante pode ser encontrado iniciando aí.

Por outro lado, embora a leitura seqüencial de um arquivo seja simples e direta, o acesso aleatório é extremamente lento. Além disso, o espaço de dados em um bloco não é mais uma potência de dois porque o ponteiro ocupa alguns bytes. Embora não fatal, ter um tamanho peculiar é menos eficiente porque muitos programas lêem e gravam em blocos cujo tamanho é uma potência de dois.

entra figura fig_08_04.jpg

Figura 8.4 – Armazenando um arquivo como uma lista encadeada de blocos de disco.

8.2.3 Alocação por Lista Encadeada Utilizando um Índice

As duas vantagens de alocação por lista encadeada podem ser eliminadas pegando a palavra de ponteiro de cada bloco de disco e colocando-a em uma tabela ou em um índice na memória. A Figura A-5 mostra a aparência que tem a tabela para o exemplo da Figura A-4. Em ambas as figuras, temos dois arquivos. O arquivo A utiliza os blocos de disco 4, 7, 2, 10 e 12, nessa ordem, e o arquivo B utiliza os blocos de disco 6, 3, 11 e 14, nessa ordem. Utilizando a tabela da Figura A-5, podemos iniciar com o bloco 4 e seguir a cadeia completamente até o fim. O mesmo pode ser feito iniciando com o bloco 6.

Utilizando essa organização, o bloco inteiro está disponível para dados. Além disso, o acesso aleatório é muito mais fácil. Embora a cadeia ainda deva ser seguida para localizar-se em dado deslocamento dentro do arquivo, a cadeia está inteiramente na memória, portanto, ela pode ser seguida sem fazer qualquer referência de disco. Como no método anterior, é suficiente que a entrada de diretório mantenha um inteiro simples (o número do bloco inicial) e ainda seja capaz de localizar todos os blocos, independente do tamanho do arquivo. O MS-DOS utiliza esse método para alocação de disco.

A desvantagem principal desse método é que a tabela inteira deve estar na memória todo o tempo para fazê-lo funcionar. Com um disco grande, digamos, 500.000 blocos de 1K (500 M), tabela terá 500.000 entradas, cada uma das quais com o mínimo 3 bytes. Para acelerar as consultas, elas devem ser de 4 bytes. Assim, a tabela ocupará 1,5 ou 2 megabytes todo o tempo dependendo de o sistema ser otimizado para espaço ou tempo. Embora o MS-DOS utilize esse mecanismo, ele evita tabelas enormes, utilizando blocos grandes (até 32K) em discos grandes.

Tabela 8.1 – Alocação por lista encadeada, utilizando uma tabela na memória principal

entra planilha 8.1 do arquivo “Tabelas.xls”

8.2.3 [KMH2]Nós-I

Nosso último método para monitorar quais blocos pertencem a quais arquivos é associar com cada arquivo uma pequena tabela chamada nó-i (nó de índice), que lista os atributos e os endereços de disco dos blocos do arquivo, como mostrado na Figura A-6.

Os primeiros endereços de disco são armazenados no próprio nó-i, então, para arquivos pequenos, todas as informações necessárias estão diretamente no nó-i, que é carregado do disco para a memória principal quando o arquivo é aberto. Para arquivos maiores, um dos endereços no nó-i é o endereço de um bloco de disco chamado bloco indireto simples. Esse bloco contém endereços adicionais de disco. Se isso ainda não for suficiente, outro endereço do nó-i, chamado bloco indireto duplo, contém o endereço de um bloco que, por sua vez, contém uma lista de blocos indiretos simples. Cada um desses blocos indiretos simples aponta para algumas centenas de blocos de dados. Se isso ainda não for suficiente, um bloco indireto triplo também poderá ser utilizado. O ~~UNIX~~ Unix utiliza esse esquema.

entra fig_08_05.jpg

Figura 8.5 – Um nó-i.

8.2.4 Gerenciamento de Espaço em Disco

Os arquivos normalmente são armazenados em disco, portanto, o gerenciamento de espaço em disco é uma questão importante para projetistas de sistema de arquivos. Duas estratégias gerais são possíveis para armazenar um arquivo de n bytes: os n bytes consecutivos de espaço em disco são alocado ou o arquivo é dividido em um número de blocos (não necessariamente) contíguos. As mesmas escolhas devem ser feitas em sistemas de gerenciamento de memória entre segmentação pura e paginação.

Armazenar um arquivo como uma seqüência contígua de bytes tem o problema óbvio de que, se um arquivo crescer, ele provavelmente precisará ser movido no disco. O mesmo problema aplica-se a segmentos de memória, exceto que mover um segmento na memória é uma operação relativamente rápida se comparada com mover um arquivo de uma posição no disco para outra. Por essa razão, quase todos os sistemas dividem os arquivos em blocos de tamanho fixo que não precisam ser adjacentes.

8.2.4 Tamanho de Bloco

Uma vez que foi decidido armazenar arquivos em blocos de tamanho fixo, surge a pergunta de qual tamanho o bloco deve ter. Dada a maneira como os discos são organizados, o setor, a trilha e o cilindro são candidatos óbvios para a unidade de alocação. Em um sistema com paginação, o tamanho da página é também um competidor importante.

Ter uma unidade grande de alocação, como um cilindro, significa que cada arquivo, mesmo um arquivo de 1 byte, ocupa um cilindro inteiro. Estudos (Mullender e Tanenbaum, 1984) demonstraram que o tamanho médio de arquivo em ambientes UNIX está por volta de 1K, então, alocar um cilindro e 32K para cada arquivo desperdiçaria 31/32 ou 97% de espaço total em disco. Por outro lado, utilizar uma unidade de alocação pequena significa que cada arquivo consistirá em muitos blocos. A leitura de cada bloco normalmente exige uma busca e um atraso rotacional, portanto, ler um arquivo consistindo em muitos blocos pequenos é lento.

Como um exemplo, considere um disco de 32.768 bytes por trilha, um tempo de rotação de 16,67ms e um tempo de busca médio e 30ms. O tempo em milissegundos para ler um bloco de k bytes é, então, a soma dos tempos de busca, de atraso rotacional e transferência:

30 + 8,3 + (k / 32768) x 16,67

A curva sólida da Figura A-10 mostra a taxa de dados para tal disco como uma função do tamanho de bloco. Se fizermos a suposição grosseira de que todos os arquivos têm 1K (o tamanho médio medido), a curva tracejada dessa figura dá a eficiência do espaço em disco. A má notícia é que a boa utilização de espaço (tamanho de bloco < 2K) significa taxas de dados baixas e vice-versa. A eficiência de tempo e de espaço está inerentemente em conflito.

O ajuste é escolhe o tamanho de bloco de 512, 1K ou 2K bytes. Se um tamanho de bloco de 1K for escolhido em um disco com um tamanho de setor de 512 bytes, então, o sistema de arquivos sempre lerá ou gravará dois setores consecutivos e irá tratá-los como uma única unidade indivisível. Qualquer que seja a decisão tomada, provavelmente ela deve ser reavaliada periodicamente, uma vez que, como todos os aspectos da tecnologia de computador, os usuários tiram proveito dos recursos mais abundantes, exigindo cada vez mais. Um gerenciador de sistema informa que o tamanho médio de arquivos no sistema de uma universidade que ele gerencia aumentou lentamente com os anos e que em 1997, o tamanho médio de arquivo cresceu para 12K, no caso dos alunos, e para 15K, no caso dos profissionais e dos professores da faculdade.

entra fig_08_06.jpg

Figura 8.6 – A curva sólida (escala da esquerda) fornece a taxa de transferência de dados de um disco. A curva tracejada (escala da direita) fornece a eficiência do espaço em disco. Todos os arquivos são de 1K.[KMH3]

8.2.5 Monitorando Blocos Livres

Uma vez que um tamanho de bloco foi escolhido, a propósito questão é como monitorar blocos livres. Dois métodos são amplamente utilizados, como mostrado na Figura 8.7. O primeiro consiste em utilizar uma lista encadeada de blocos de disco, com cada bloco armazenado tantos quantos números livres de bloco de disco couberem. Com um bloco de 1K e um número de bloco de disco de 32 bits, cada bloco na lista de livres armazena os números de 255 blocos livres. (Uma entrada é necessária para o ponteiro para o próximo bloco). Um disco de 200MB necessita de uma lista de livres de no máximo 804 blocos para armazenar todos os 200K números de bloco de disco. Blocos livres freqüentemente são utilizados para armazenar a lista de livres.

A outra técnica de gerenciamento de espaço livre é o mapa de bits. Um disco com n blocos requer um mapa de bits com n bits. Blocos livres são representados por 1s no mapa, e blocos alocados por 0s (ou vice-versa). Um disco de 200MB requer 200K bits para o mapa, o que requer somente 25 blocos. Não é de surpreender que o mapa de bits exija menos espaço, uma vez que utiliza 1 bit por bloco, versus 32 bits no modelo de lista encadeada. Somente se o disco estiver quase cheio é que o esquema de lista encadeada irá requerer menos blocos que o mapa de bits.

Se houver memória principal suficiente para armazenar o mapa de bits, esse método é geralmente preferível. Se, entretanto, somente 1 bloco de memória puder ser dispensado para monitorar blocos livres no disco e o disco estiver quase cheio, então, a lista encadeada pode ser melhor. Com somente 1 bloco de mapa de bits na memória, é possível que nenhum bloco livre possa ser encontrado, fazendo com que acessos de disco adicionais sejam necessários para ler o restante do mapa de bits. Quando um bloco novo da lista encadeada é carregado na memória, 255 blocos de disco podem ser alocados antes de ir-se para o disco a fim de buscar o próximo bloco da lista.

entra fig_08_07.jpg

Figura 8.7 – (a) Armazenando a lista de livres em uma lista encadeada. (b) Um mapa de bits.

8.3 – Exemplo de Sistema de Arquivos

Para entender melhor os conceitos de sistemas de arquivos, será apresentado um exemplo escrito em linguagem C. Neste exemplo, existe um arquivo de nome USER.FAT, que representa a FAT escrita em disco em um sistemas real.

Com esse exemplo de FAT, uma FAT deve ser montada antes de qualquer operação através do comando “M”, que busca os dados da FAT (arquivo USER.FAT) e coloca-os a disposição das operações de “L” para lista de arquivos, “A” para abrir um arquivo e exibir seu conteúdo, “C” para criar um novo arquivo de texto e “D” para apagar um arquivo existente.

O programa é explícito quando as operações, e representam o que realmente acontece em um sistemas de arquivos de um sistema operacional, nesse caso de forma mais simplificada, ou seja, não são salvos os clusters aonde estão armazenados os arquivos, se estão vinculados a algum diretório, seus tamanhos, proprietário e demais informações úteis, de qualquer forma, nada impede de serem implementada tais melhorias no código.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

void ler_FAT();
void nomes_arq();
void abrir_arq(char nome[]);
void grava_arq();
void apaga_arq();
int ver_nome(char vnome[]);
int streql (char *str1, char *str2);

/* Tamanho de cada entrada na FAT = 26 */
/* Permite atualmente ateh 10 entradas */
char entrada[100][26];
/* Tamanho temporario de nome de arquivo = 11 */
char lista[100][11];
```

```
char nome[11];
/* String adicionada ao final do nome do arquivo nas entradas da FAT */
char *padrao = "%A%rwxr__r__%/&";
/* Arquivos criados com 200 linhas por 80 colunas */
char conteudo[200][80];
/* posfat contem a posicao (linha) da entrada na FAT (user.fat) */
int posfat=0;
/* Quantidade de entradas na fat */
int qtent=0;
/* Flag marcador de FAT montada */
int ffat=0;

void main()
{
  int i,vn;
  char str[1];
  char file[11];
  char ch;

  for(;;)
  {
    do
    {
      printf("\n(S)air, (M)ontar FAT, L(I)star, (A)brir, (C)riar, (D)eletar: ");
      gets(str);
      ch = toupper(*str);
    } while(ch!='S' && ch!='M' && ch!='I' && ch!='A' && ch!='C' && ch!='D');


    switch(ch)
    {
      case 'S':
        exit(0);

      case 'M':
        if (ffat == 0)
        {
          ler_FAT();
          printf("Quantidade de entradadas na fat: %i\n",qtent);
          for (i=0; i<qtent; i++)
          {
            printf("%s \n",entrada[i]);
          }
        }
        else
              printf("FAT ja montada!");
        break;

      case 'I':
```

```
  if (ffat == 1)
  {
    for (i=0; i<qtent; i++)
      printf("%s \n",lista[i]);
  }
  else
    printf("FAT nao montada!");
  break;

case 'A':
  if (ffat == 1)
  {
    printf("Nome: ");
    gets(nome);
    vn = ver_nome(nome);
    if (vn == 1)
      abrir_arq(nome);
    else
      printf("Nome de arquivo nao existe na entrada da FAT!\n");
  }
  else
    printf("FAT nao montada!");
  break;

case 'C':
  if (ffat == 1)
  {
    printf("Nome: ");
    gets(nome);
    grava_arq();
  }
  else
    printf("FAT nao montada!");
  break;

case 'D':
  if (ffat == 1)
  {
    printf("Nome: ");
    gets(nome);
    vn = ver_nome(nome);
    if (vn == 1)
      apaga_arq(nome);
    else
      printf("Nome de arquivo nao existe na entrada da FAT!\n");
  }
  else
    printf("FAT nao montada!");
  break;
```

```
    }
  }
}

/* Extrai da matriz entrada os nomes do arquivos e salva na matriz lista
com Nx30, aonde N eh o numero de entradas e 30 o tamanho maximo de nome de
um arquivo */
void nomes_arq()
{
        int j,c,sai,pro;

        for (j=0; j<qtent; j++)
        {
                sai = 0;
                c = 0;
                while (sai == 0)
                {
                        if (entrada[j][c] == '%')
                                sai = 1;
                        else
                                lista[j][c] = entrada[j][c];

                        c++;
                }
        }
}

/* Abre o arquivo fisicamente, assimila as linhas como "entradas" da FAT e
as armazena em uma matriz chamada entrada com Nx26, aonde N eh uma
numero de entradas da FAT e 26 o tamanho maximo de cada uma delas */
void ler_FAT()
{
        FILE *fp;
        char ch;
  int c=0, i=0;

  if ((fp = fopen("user.fat","r")) != NULL)
        {
                ch = getc(fp);
                while(ch != EOF)
                {
                        if (ch == '&')
                        {
        entrada[i][c] = ch;
                                i++;
                                c=-2;
                        }
                        else
```

```
                                        entrada[i][c] = ch;

                                c++;
                                ch = getc(fp);
                        }
    fclose(fp);
                        qtent = i;

    nomes_arq();

    ffat = 1;
            }
            else
            {
    if ((fp = fopen("user.fat","a")) != NULL)
    {
      ffat = 1;
      fclose(fp);
    }
    else
      printf("\nErro! Nao foi possivel criar arquivo da FAT.\n");
            }

}

/* Abre o arquivo intitulado "nome", e coleta todo seu conteudo jogando-o para a
matriz conteudo com 200x80, ou seja, 200 linhas por 80 colunas */
void abrir_arq(char nome[])
{

            FILE *fp;
            char ch;
            int c=0, i=0;

            if ((fp = fopen(nome,"r+")) != NULL)
            {
                        ch = getc(fp);
                        while(ch != EOF)
                        {

        if (ch!='&')
        {
          if (c == 80)
          {
            i++;
            c = 0;
          }
          putchar(ch);
          conteudo[i][c] = ch;
```

```
            ch = getc(fp);
        }
        else
            break;
                    }
    printf("\n");
  }
            else
            {
                    printf("\nArquivo nao existe!\n");
            }

}

/* Verifica se o nome passado em vnome esta na matriz lista, retornando 0 para
falso e 1 para verdadeiro */
int ver_nome(char vnome[])
{
            int flag=0,j=0;

            for (j=0; j < qtent; j++)
            {
                    if (strlen(lista[j]) == strlen(vnome))
                    {
                            flag = streql(lista[j], vnome);
                            if (flag == 1)
                                    break;
                    }
            }

            posfat = j;
            return flag;
}

void grava_arq()
{
  FILE *fp;
  char ch=' ';
  int c;

  printf("\nGrava arq nome= %s", nome);
  if((fp=fopen(nome, "w"))==NULL)
  {
    printf("Arquivo nao pode ser aberto.\n");
  }
  else
  {
    printf("\nConteudo - & e ENTER para encerrar:\n");
```

```
    while(ch!='&')
    {
      ch = getchar();
      putc(ch,fp);
    }

    fclose(fp);

    if ((fp = fopen("user.fat","a")) != NULL)
    {
      for (c=0; c<strlen(nome); c++)
        putc(nome[c],fp);
      for (c=0; c<15; c++)
        putc(padrao[c],fp);
      putc('\n',fp);

      fclose(fp);
    }

    ffat = 0;
    ler_FAT();
  }

}

void apaga_arq()
{
  FILE *fp2;
  int c=0,j=-2,lala;

  printf("\nApagando arquivo: %s", nome);

  remove(nome);

  if ((fp2 = fopen("user.fat","w")) != NULL)
            {
    if (posfat > 0)
    {
      for (c=0; c<posfat; c++)
      {
        if (c == 0)
          j = j + 2;

        while(entrada[c][j] != '&')
        {
            putc(entrada[c][j],fp2);
          j++;
        }
        putc('&',fp2);
```

```
            putc('\n',fp2);

            j = 0;
          }
        }

        j = 0;
        for (c=posfat+1; c<qtent; c++)
        {

          while(entrada[c][j] != '&')
          {
            if (entrada[c][j] != ' ')
              putc(entrada[c][j],fp2);
            j++;
          }
          putc('&',fp2);
          putc('\n',fp2);

          j = 0;
        }

        fclose(fp2);
                }

    /* Zerar entradas atuais da fat */
    for (j=0; j<qtent; j++)
    {
      for (c=0; c<26; c++)
        entrada[j][c] = ' ';
    }

    ffat = 0;
    ler_FAT();
}

/* Comparar strings */
int streql (char *str1, char *str2)
{
        while ((*str1 == *str2) && (*str1))
        {
                str1++;
                str2++;
        }
        return ((*str1 == NULL) && (*str2 == NULL));
}
```