Sistemas de Arquivos

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

Um segundo problema em manter informações dentro do espaço de endereçamento de um processo é que, quando o processo é concluído, as informações são perdidas.

Muitas vezes, é necessário que múltiplos processos acessem uma informação ao mesmo tempo. Se temos um diretório telefônico on-line armazenado dentro do espaço de um processo, só aquele processo pode acessá-lo. É preciso tornar a informação independente de qualquer processo.

• Três requisitos para o armazenamento de informações por um longo prazo:

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

Discos magnéticos foram usados por anos para esse armazenamento de longo prazo. Recentemente, unidades de estado sólido tornaram-se cada vez mais populares, já que não têm partes móveis que possam quebrar.

Basta pensar em um disco como uma sequência linear de blocos de tamanho fixo e que dão suporte a duas operações:
1. Leia o bloco k.
2. Escreva no bloco k.

Existem mais operações, mas com essas duas é possível solucionar o problema do armazenamento de longo prazo.

Entretanto, essas operações são muito inconvenientes, mais ainda em sistemas grandes usados por muitas aplicações e múltiplos usuários. Questões que rapidamente surgem podem ser:

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

As abstrações de processos (e threads), espaços de endereçamento e arquivos são os conceitos mais importantes relacionados com os sistemas operacionais.

Arquivos são unidades lógicas de informação criadas por processos. Um disco normalmente conterá milhares ou mesmo milhões deles, cada um independente dos outros.

Processos podem ler arquivos existentes e criar novos. Informações armazenadas em arquivos devem ser persistentes: não devem ser afetadas pela criação e término de um processo. Um arquivo deve desaparecer apenas quando o seu proprietário o remove explicitamente.

Arquivos são gerenciados pelo sistema operacional. Como um todo, aquela parte do sistema operacional lidando com arquivos é conhecida como sistema de arquivos.

Um arquivo fornece uma maneira para armazenar informações sobre o disco e lê-las depois. Isso deve ser feito de tal modo que isole o usuário dos detalhes de como e onde as informações estão armazenadas.

Nomeação de Arquivos

Quando um processo cria um arquivo, ele lhe dá um nome. Quando o processo é concluído, o arquivo continua a existir e pode ser acessado por outros processos usando o seu nome.

As regras para a nomeação de arquivos variam de sistema para sistema, mas todos os sistemas operacionais atuais permitem cadeias de uma a oito letras como nomes de arquivos legais. Dígitos e caracteres especiais também são frequentemente permitidos.

Alguns sistemas de arquivos distinguem entre letras maiúsculas e minúsculas. O UNIX faz essa distinção; o MS-DOS, não.

Um sistema UNIX pode ter todos os arquivos a seguir como três arquivos distintos: maria, Maria e MARIA. No MS-DOS, todos esses nomes referem-se ao mesmo arquivo.

Muitos sistemas operacionais aceitam nomes de arquivos de duas partes, com as partes separadas por um ponto, como em prog.c. A parte que vem depois do ponto é a extensão do arquivo.

Estrutura de Arquivos

Arquivos podem ser estruturados de várias maneiras. Três possibilidades comuns abaixo:

O arquivo em (a) é uma sequência desestruturada de bytes. O sistema operacional não sabe ou não se importa com o que há no arquivo; o que ele vê são bytes. Qualquer significado deve ser imposto por programas em nível de usuário. Ter o sistema operacional tratando arquivos como nada mais que sequências de bytes oferece a máxima flexibilidade. Todas as versões do UNIX (incluindo Linux e OS X) e o Windows usam esse modelo de arquivos.

O primeiro passo na estruturação está ilustrado em (b). Nesse modelo, um arquivo é uma sequência de registros de tamanho fixo, cada um com alguma estrutura interna. O fundamental para que um arquivo seja uma sequência de registros é a ideia de que a operação de leitura retorna um registro e a operação de escrita sobrepõe ou anexa um registro.

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

Tipos de arquivos

Muitos sistemas operacionais aceitam vários tipos de arquivos. O UNIX (incluindo OS X) e o Windows, por exemplo, apresentam arquivos regulares e diretórios. O UNIX também tem arquivos especiais de caracteres e blocos.

Os arquivos regulares são aqueles que contêm informações do usuário. Todos os arquivos da figura anterior são arquivos regulares.

Os diretórios são arquivos do sistema para manter a estrutura do sistema de arquivos.

Os arquivos especiais de caracteres são relacionados com entrada/saída e usados para modelar dispositivos de E/S seriais como terminais, impressoras e redes.

Os arquivos especiais de blocos são usados para modelar discos.

Arquivos regulares geralmente são arquivos ASCII ou arquivos binários.

Arquivos ASCII consistem de linhas de texto. Em alguns sistemas, cada linha termina com um caractere de retorno de carro (carriage return). Em outros, o caractere de próxima linha (line feed) é usado.

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

Um arquivo binário executável simples, tirado de uma versão inicial do UNIX. Embora o arquivo seja apenas uma sequência de bytes, o sistema operacional o executará somente se ele tiver o formato apropriado.

Ele tem cinco seções: cabeçalho, texto, dados, bits de realocação e tabela de símbolos. O cabeçalho começa com o chamado número mágico, identificando o arquivo como executável.

Acesso aos arquivos

Os primeiros sistemas operacionais forneciam apenas um tipo de acesso aos arquivos: sequencial. Nesses sistemas, um processo lia todos os bytes ou registros em um arquivo em ordem, começando do princípio, mas não podia pular nenhum ou lê-los fora de ordem.

Arquivos sequenciais podiam ser trazidos de volta para o ponto de partida, então podiam ser lidos quando necessário. Arquivos sequenciais eram convenientes quando o meio de armazenamento era uma fita magnética, em vez de um disco.

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

Atributos de Arquivos

Todo arquivo possui um nome e sua data. Além disso, os sistemas operacionais associam outras informações com os arquivos, como data e horário em que foi modificado pela última vez, e tamanho do arquivo. Esses itens extras são atributos do arquivo ou metadados.

Operações com arquivos

1. Create. O arquivo é criado sem dados. A finalidade dessa chamada é anunciar que o arquivo está vindo e estabelecer alguns dos atributos.

2. Delete. Quando o arquivo não é mais necessário, ele tem de ser removido para liberar espaço para o disco. Há sempre uma chamada de sistema para essa finalidade.

3. Open. Antes de usar um arquivo, um processo precisa abri-lo. A finalidade da chamada open é permitir que o sistema busque os atributos e lista de endereços do disco para a memória principal a fim de tornar mais rápido o acesso em chamadas posteriores.

4. Close. Quando todos os acessos são concluídos, os atributos e endereços de disco não são mais necessários, então o arquivo deve ser fechado para liberar espaço da tabela interna. Muitos sistemas encorajam isso impondo um número máximo de arquivos abertos em processos. Um disco é escrito em blocos, e o fechamento de um arquivo força a escrita do último bloco dele, mesmo que não esteja inteiramente cheio ainda.

5. Read. Dados são lidos do arquivo. Em geral, os bytes vêm da posição atual. Quem fez a chamada deve especificar a quantidade de dados necessária e também fornecer um buffer para colocá-los.

6. Write. Dados são escritos para o arquivo de novo, normalmente na posição atual. Se a posição atual for o final do arquivo, seu tamanho aumentará. Se estiver no meio do arquivo, os dados existentes serão sobrescritos e perdidos para sempre.

7. Append. Essa chamada é uma forma restrita de write. Ela pode acrescentar dados somente para o final do arquivo. Sistemas que fornecem um conjunto mínimo de chamadas do sistema raramente têm append, mas muitos sistemas fornecem múltiplas maneiras de fazer a mesma coisa, e esses às vezes têm append.

8. Seek. Para arquivos de acesso aleatório, é necessário um método para especificar de onde tirar os dados. Uma abordagem comum é uma chamada de sistema, seek, que reposiciona o ponteiro de arquivo para um local específico dele. Após essa chamada ter sido completa, os dados podem ser lidos da, ou escritos para, aquela posição.

9. Get attributes. Processos muitas vezes precisam ler atributos de arquivos para realizar seu trabalho. Por exemplo, o programa make da UNIX costuma ser usado para gerenciar projetos de desenvolvimento de software consistindo de muitos arquivos-fonte. Quando make é chamado, ele examina os momentos de alteração de todos os arquivos-fonte e objetos e organiza o número mínimo de compilações necessárias para atualizar tudo. Para realizar o trabalho, o make deve examinar os atributos, a saber, os momentos de alteração.

10.Set attributes. Alguns dos atributos podem ser alterados pelo usuário e modificados após o arquivo ter sido criado. Essa chamada de sistema torna isso possível. A informação sobre o modo de proteção é um exemplo óbvio. A maioria das sinalizações também cai nessa categoria.

11.Rename. Acontece com frequência de um usuário precisar mudar o nome de um arquivo. Essa chamada de sistema torna isso possível. Ela nem sempre é estritamente necessária, porque o arquivo em geral pode ser copiado para um outro com um nome novo, e o arquivo antigo é então deletado.

Diretórios

Para controlar os arquivos, sistemas de arquivos normalmente têm diretórios ou pastas, que são em si arquivos. A forma mais simples de um sistema de diretório é ter um diretório contendo todos os arquivos, um diretório-raiz.

Curiosamente, o primeiro supercomputador do mundo, o CDC 6600, também tinha apenas um único diretório para todos os arquivos, embora fosse usado por muitos usuários ao mesmo tempo. Essa decisão foi tomada sem dúvida para manter simples o design do software.

As vantagens desse esquema são a sua simplicidade e a capacidade de localizar arquivos rapidamente — há apenas um lugar para se procurar, afinal.

O nível único é adequado para aplicações dedicadas muito simples. Para usuários modernos, com milhares de arquivos, seria impossível encontrar qualquer coisa se os arquivos estivessem em um único diretório. É necessária uma maneira para agrupar arquivos relacionados em um mesmo local.

Com hierarquia, o usuário pode ter tantos diretórios quantos forem necessários. Além disso, se múltiplos usuários compartilham um servidor de arquivos comum, cada usuário pode ter um diretório-raiz privado para sua própria hierarquia.

Ex: cada diretório A, B e C contido no diretório-raiz pertence a um usuário diferente, e dois deles criaram subdiretórios para projetos nos quais estão trabalhando.

Nomes de caminhos

Quando o sistema de arquivos é organizado com uma árvore de diretórios, alguma maneira é necessária para especificar os nomes dos arquivos. Dois métodos diferentes são os mais usados. No primeiro, cada arquivo recebe um nome de caminho absoluto consistindo no caminho do diretório-raiz para o arquivo.

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

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

O outro tipo é o nome de caminho relativo. Esse é usado em conjunção com o conceito do diretório de trabalho (também chamado de diretório atual). Um usuário pode designar um diretório como o de trabalho atual, caso em que todos os nomes de caminho não começando no diretório-raiz são presumidos como relativos ao diretório de trabalho.

cp /home/pvb/texto1.txt /home/pvb/texto2.txt
cp texto1.txt texto2.txt

Produziriam o mesmo resultado se o diretório corrente for /home/pvb

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

A maioria dos sistemas operacionais que aceita um sistema de diretório hierárquico tem duas entradas especiais em cada diretório, “.” e “..”, geralmente pronunciadas como “ponto” e “pontoponto”. Ponto refere-se ao diretório atual; pontoponto refere-se ao pai (exceto no diretório-raiz, onde ele refere-se a si mesmo).

Operações com Diretórios

As chamadas de sistema que podem gerenciar diretórios exibem mais variação de sistema para sistema do que as chamadas para gerenciar arquivos.

1. Create. Um diretório é criado. Ele está vazio exceto por ponto e pontoponto, que são colocados ali automaticamente pelo sistema (ou em alguns poucos casos, pelo programa mkdir).

2. Delete. Um diretório é removido. Apenas um diretório vazio pode ser removido. Um diretório contendo apenas ponto e pontoponto é considerado vazio à medida que eles não podem ser removidos.

3. Opendir. Diretórios podem ser lidos. Por exemplo, para listar todos os arquivos em um diretório, um programa de listagem abre o diretório para ler os nomes de todos os arquivos que ele contém. Antes que um diretório possa ser lido, ele deve ser aberto, de maneira análoga a abrir e ler um arquivo.

4. Closedir. Quando um diretório tiver sido lido, ele será fechado para liberar espaço de tabela interno.

5. Readdir. Essa chamada retorna a próxima entrada em um diretório aberto. Antes, era possível ler diretórios usando a chamada de sistema read usual, mas essa abordagem tem a desvantagem de forçar o programador a saber e lidar com a estrutura interna de diretórios. Por outro lado, readdir sempre retorna uma entrada em um formato padrão, não importa qual das estruturas de diretório possíveis está sendo usada.

6. Rename. Em muitos aspectos, diretórios são como arquivos e podem ser renomeados da mesma maneira que eles.

7. Link. A ligação (linking) é uma técnica que permite que um arquivo apareça em mais de um diretório. Essa chamada de sistema especifica um arquivo existente e um nome de caminho, e cria uma ligação do arquivo existente para o nome especificado pelo caminho. Dessa maneira, o mesmo arquivo pode aparecer em múltiplos diretórios. Uma ligação desse tipo, que incrementa o contador no i-node do arquivo (para monitorar o número de entradas de diretório contendo o arquivo), às vezes é chamada de ligação estrita (hard link).

8. Unlink. Uma entrada de diretório é removida. Se o arquivo sendo removido estiver presente somente em um diretório (o caso normal), ele é removido do sistema de arquivos. Se ele estiver presente em múltiplos diretórios, apenas o nome do caminho especificado é removido. Os outros continuam. Em UNIX, a chamada de sistema para remover arquivos (discutida anteriormente) é, na realidade, unlink.

Uma variação da ideia da ligação de arquivos é a ligação simbólica. Em vez de ter dois nomes apontando para a mesma estrutura de dados interna representando um arquivo, um nome pode ser criado que aponte para um arquivo minúsculo que nomeia outro arquivo. Quando o primeiro é usado — aberto, por exemplo — o sistema de arquivos segue o caminho e encontra o nome no fim. Então ele começa todo o processo de localização

usando o novo nome. Ligações simbólicas têm a vantagem de conseguirem atravessar as fronteiras de discos e mesmo nomear arquivos em computadores remotos. No entanto, sua implementação é de certa maneira menos eficiente do que as ligações estritas.

Implementação do sistema de arquivos

Usuários estão preocupados em como os arquivos são nomeados, quais operações são permitidas neles, como é a árvore de diretórios e questões de interface similares.

Implementadores estão interessados em como os arquivos e os diretórios estão armazenados, como o espaço de disco é gerenciado e como fazer tudo funcionar de maneira eficiente e confiável.

Sistemas de arquivos são armazenados em discos. A maioria dos discos pode ser dividida em uma ou mais partições, com sistemas de arquivos independentes em cada partição. O Setor 0 do disco é chamado de MBR (Master Boot Record — registro mestre de inicialização) e é usado para inicializar o computador.

Muitas vezes o sistema de arquivos vai conter alguns dos itens. O primeiro é o superbloco. Ele contém todos os parâmetros-chave a respeito do sistema de arquivos e é lido para a memória quando o computador é inicializado ou o sistema de arquivos é tocado pela primeira vez.

Em seguida podem vir informações a respeito de blocos disponíveis no sistema de arquivos, na forma de um mapa de bits ou de uma lista de ponteiros, por exemplo.

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

Os I-nodes correspondem a um arranjo de estruturas de dados para implementar os descritores de arquivos/diretórios.

O próximo campo corresponde ao diretório raiz, que contám o topo da árvore de diretórios.

O restante do disco contém todos os outros diretórios e arquivos.

Alocação contígua

O esquema de alocação mais simples é armazenar cada arquivo como uma execução contígua de blocos de disco. Assim, em um disco com blocos de 1 KB, um arquivo de 50 KB seria alocado em 50 blocos consecutivos. Com blocos de 2 KB, ele seria alocado em 25 blocos consecutivos. Observe que cada arquivo começa no início de um bloco novo; portanto, se o arquivo A realmente ocupar 3½ blocos, algum espaço será desperdiçado ao fim de cada último bloco.

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

Segundo, o desempenho da leitura é excelente, pois o arquivo inteiro pode ser lido do disco em uma única operação. Apenas uma busca é necessária (para o primeiro bloco).

Infelizmente, a alocação contígua tem um ponto fraco importante: com o tempo, o disco torna-se fragmentado.

Reutilizar o espaço exige manter uma lista de lacunas, o que é possível. No entanto, quando um arquivo novo vai ser criado, é necessário saber o seu tamanho final a fim de escolher uma lacuna do tamanho correto para alocá-lo.

No entanto, há uma situação na qual a alocação contígua é possível e, na realidade, ainda usada: em CD-ROMs. Aqui todos os tamanhos de arquivos são conhecidos antecipadamente e jamais mudarão durante o uso subsequente do sistema de arquivos do CD-ROM.

Alocação por lista encadeada

O segundo método para armazenar arquivos é manter cada um como uma lista encadeada de blocos de disco.

A primeira palavra de cada bloco é usada como um ponteiro para a próxima. O resto do bloco é reservado para dados. Diferentemente da alocação contígua, todos os blocos do disco podem ser usados nesse método. Nenhum espaço é perdido para a fragmentação de disco.

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

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

Alocação por lista encadeada usando uma tabela na memória

Ambas as desvantagens da alocação por lista encadeada - acesso aleatório de extrema lentidão e quantidade de dados que um bloco pode armazenar não mais uma potência de

dois - podem ser eliminadas colocando-se as palavras do ponteiro de cada bloco de disco em uma tabela na memória.

Usando essa organização, o bloco inteiro fica disponível para dados. Além disso, o acesso aleatório é muito mais fácil. Embora ainda seja necessário seguir o encadeamento para encontrar um determinado deslocamento dentro do arquivo, o encadeamento está inteiramente na memória, portanto ele pode ser seguido sem fazer quaisquer referências ao disco.

O arquivo A usa os blocos de disco 4, 7, 2, 10 e 12, nessa ordem, e o arquivo B usa os blocos de disco 6, 3, 11 e 14, nessa ordem. Usando a tabela da figura, podemos começar com o bloco 4 e seguir a cadeia até o fim. Essa tabela na memória principal é chamada de FAT (File Allocation Table — tabela de alocação de arquivos). A principal desvantagem desse método é que a tabela inteira precisa estar na memória o todo o tempo para fazê-la funcionar.

Claro, a ideia da FAT não se adapta bem para discos grandes. Era o sistema de arquivos MS-DOS original e ainda é aceito completamente por todas as versões do Windows.

I-nodes

Nosso último método para monitorar quais blocos pertencem a quais arquivos é associar cada arquivo a uma estrutura de dados chamada de i-node (index-node — nó-índice), que lista os atributos e os endereços de disco dos blocos do disco.

Dado o i-node, é então possível encontrar todos os blocos do arquivo. A grande vantagem desse esquema sobre os arquivos encadeados usando uma tabela na memória é que o i-node precisa estar na memória apenas quando o arquivo correspondente estiver aberto. Se cada i-node ocupa n bytes e um máximo de k arquivos puderem estar abertos simultaneamente, a memória total ocupada pelo arranjo contendo os i- -nodes para os arquivos abertos é de apenas kn bytes. Apenas essa quantidade de espaço precisa ser reservada antecipadamente.

O esquema i-node exige um conjunto na memória cujo tamanho seja proporcional ao número máximo de arquivos que podem ser abertos ao mesmo tempo.

Um problema com i-nodes é que se cada um tem espaço para um número fixo de endereços de disco, o que acontece quando um arquivo cresce além de seu limite? Uma solução é reservar o último endereço de disco não para um bloco de dados, mas, em vez disso, para o endereço de um bloco contendo mais endereços de blocos de disco.

Implementando diretórios

Antes que um arquivo possa ser lido, ele precisa ser aberto. Quando um arquivo é aberto, o sistema operacional usa o nome do caminho fornecido pelo usuário para localizar a entrada de diretório no disco. A entrada de diretório fornece a informação necessária para encontrar

os blocos de disco. Dependendo do sistema, essa informação pode ser o endereço de disco do arquivo inteiro (com alocação contígua), o número do primeiro bloco (para ambos os esquemas de listas encadeadas), ou o número do i-node. Em todos os casos, a principal função do sistema de diretórios é mapear o nome do arquivo em ASCII na informação necessária para localizar os dados.

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

Para sistemas que usam i-nodes, outra possibilidade para armazenar os atributos é nos próprios i-nodes, em vez de nas entradas do diretório. Nesse caso, a entrada do diretório pode ser mais curta: apenas um nome de arquivo e um número de i-node.

Gerenciando nomes longos

Uma alternativa é abrir mão da ideia de que todas as entradas de diretório sejam do mesmo tamanho. Com esse método, cada entrada de diretório contém uma porção fixa, começando com o tamanho da entrada e, então, seguido por dados com um formato fixo, normalmente incluindo o proprietário, momento de criação, informações de proteção e outros atributos. Esse cabeçalho de comprimento fixo é seguido pelo nome do arquivo real, não importa seu tamanho.

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

Outra maneira de lidar com nomes de tamanhos variáveis é tornar fixos os tamanhos das próprias entradas de diretório e manter os nomes dos arquivos em um heap (monte) no fim de cada diretório.

Esse método tem a vantagem de que, quando uma entrada for removida, o arquivo seguinte inserido sempre caberá ali. É claro, o heap deve ser gerenciado e faltas de páginas ainda podem ocorrer enquanto processando nomes de arquivos. Um ganho menor aqui é que não há mais nenhuma necessidade real para que os nomes dos arquivos comecem junto aos limites das palavras, de maneira que não é mais necessário completar os nomes dos arquivos com caracteres.

Para diretórios extremamente longos, a busca linear pode ser lenta. Uma maneira de acelerar a busca é usar uma tabela de espalhamento em cada diretório. Defina o tamanho da tabela n. Ao entrar com um nome de arquivo, o nome é mapeado em um valor entre 0 e n – 1.

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

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

Uma maneira diferente de acelerar a busca em grandes diretórios é colocar os resultados em uma cache de buscas.

Arquivos compartilhados

Quando vários usuários estão trabalhando juntos em um projeto, eles muitas vezes precisam compartilhar arquivos. Em consequência, muitas vezes é conveniente que um arquivo compartilhado apareça simultaneamente em diretórios diferentes pertencendo a usuários distintos.

A conexão entre o diretório do usuário B e o arquivo compartilhado é chamada de ligação. O sistema de arquivos em si é agora um Gráfico Acíclico Orientado (Directed Acyclic Graph — DAG), em vez de uma árvore. Ter o sistema de arquivos como um DAG complica a manutenção, mas a vida é assim.

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

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

Na segunda solução, B se liga a um dos arquivos de C, obrigando o sistema a criar um novo arquivo do tipo LINK e a inseri-lo no diretório de B. O novo arquivo contém apenas o nome do caminho do arquivo para o qual ele está ligado. Quando B lê do arquivo ligado, o

sistema operacional vê que o arquivo sendo lido é do tipo LINK, verifica seu nome e o lê. Essa abordagem é chamada de ligação simbólica.

Sistemas de arquivos journaling

A ideia básica aqui é manter um diário do que o sistema de arquivos vai fazer antes que ele o faça; então, se o sistema falhar antes que ele possa fazer seu trabalho planejado, ao ser reinicializado, ele pode procurar no diário para ver o que acontecia no momento da falha e concluir o trabalho. Esse tipo de sistema de arquivos, chamado de sistemas de arquivos journaling, já está em uso na realidade.

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

Para aumentar a confiabilidade, um sistema de arquivos pode introduzir o conceito do banco de dados de uma transação atômica.

Gerenciamento e otimização de sistemas de arquivos

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

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

Feita a opção de armazenar arquivos em blocos de tamanho fixo, devemos decidir qual tamanho o bloco terá. Dado o modo como os discos são organizados, o setor, a trilha e o cilindro são candidatos óbvios para a unidade de alocação. Em um sistema de paginação, o tamanho da página também é um argumento importante.

Se a unidade de alocação for grande demais, desperdiçamos espaço; se ela for pequena demais, desperdiçamos tempo.

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

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

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

Uma vez que um tamanho de bloco tenha sido escolhido, a próxima questão é como monitorar os blocos livres. Dois métodos são amplamente usados. O primeiro consiste em usar uma lista encadeada de blocos de disco, com cada bloco contendo tantos números de blocos livres de disco quantos couberem nele. A outra técnica de gerenciamento de espaço livre é o mapa de bits. Um disco com n blocos exige um mapa de bits com n bits.

Não surpreende que o mapa de bits exija menos espaço, tendo em vista que ele usa 1 bit por bloco, versus 32 bits no modelo de lista encadeada. Apenas se o disco estiver praticamente cheio (isto é, tiver poucos blocos livres) o esquema da lista encadeada exigirá menos blocos do que o mapa de bits.

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

Apenas um bloco de ponteiros precisa ser mantido na memória principal. Quando um arquivo é criado, os blocos necessários são tomados do bloco de ponteiros. Quando ele se esgota, um novo bloco de ponteiros é lido do disco. De modo similar, quando um arquivo é removido, seus blocos são liberados e adicionados ao bloco de ponteiros na memória principal. Quando esse bloco completa, ele é escrito no disco.

Manter a maior parte dos blocos de ponteiros cheios em disco (para minimizar o uso deste), mas manter o bloco na memória cheio pela metade, de maneira que ele possa lidar tanto com a criação quanto com a remoção de arquivos, sem uma operação de E/S em disco para a lista de livres.

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

Desempenho do sistema de arquivos

O acesso ao disco é muito mais lento do que o acesso à memória. Se apenas uma única palavra for necessária, o acesso à memória será da ordem de um milhão de vezes mais rápido que o acesso ao disco. Como consequência dessa diferença em tempo de acesso, muitos sistemas de arquivos foram projetados com várias otimizações para melhorar o desempenho.

Cache de blocos

Técnica mais comum, usada para reduzir os acessos ao disco. Nesse contexto, uma cache é uma coleção de blocos que logicamente pertencem ao disco, mas estão sendo mantidas na memória por razões de segurança.

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

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

Uma diferença bem-vinda entre a paginação e a cache de blocos é que as referências de cache são relativamente raras, de maneira que é viável manter todos os blocos na ordem exata do LRU com listas encadeadas.

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

Caches nas quais todos os blocos modificados são escritos de volta para o disco imediatamente são chamadas de caches de escrita direta (write-through caches). Elas exigem mais E/S de disco do que caches que não são de escrita direta.

Leitura antecipada de blocos

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

É claro, essa estratégia de leitura antecipada funciona apenas para arquivos que estão de fato sendo lidos sequencialmente. Se um arquivo estiver sendo acessado aleatoriamente, a leitura antecipada não ajuda. Na realidade, ela piora a situação, pois emperra a largura de banda do disco, fazendo leituras em blocos inúteis e removendo blocos potencialmente úteis da cache (e talvez emperrando mais ainda a largura de banda escrevendo os blocos de volta para o disco se eles estiverem sujos). Para ver se a leitura antecipada vale a pena ser feita, o sistema de arquivos pode monitorar os padrões de acesso para cada arquivo aberto. Por exemplo, um bit associado com cada arquivo pode monitorar se o arquivo está em “modo de acesso sequencial” ou “modo de acesso aleatório”.

Redução do movimento do braço do disco

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

Uma variação sobre o mesmo tema é levar em consideração o posicionamento rotacional. Quando aloca blocos, o sistema faz uma tentativa de colocar blocos consecutivos em um arquivo no mesmo cilindro.

Outro gargalo de desempenho em sistemas que usam i-nodes (ou qualquer equivalente a eles) é que a leitura mesmo de um arquivo curto exige dois acessos de disco: um para o i-node e outro para o bloco. Duas soluções possíveis:

1: i-nodes estão próximos do início do disco, então a distância média entre um i-node e seus blocos será metade do número de cilindros, exigindo longas buscas. Uma melhora simples de desempenho é colocar os i-nodes no meio do disco, em vez de no início, reduzindo assim a busca média entre o i-node e o primeiro bloco por um fator de dois.

2: Dividir o disco em grupos de cilindros, cada um com seus próprios i-nodes, blocos e lista de livres (MCKUSICK et al., 1984). Ao criar um arquivo novo, qualquer i-node pode ser escolhido, mas uma tentativa é feita para encontrar um bloco no mesmo grupo de cilindros que o i-node. Se nenhum estiver disponível, então um bloco em um grupo de cilindros próximo é usado.

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