memória de forma a verificar sua integridade). Somente após o registro no log é que as diferentes operações têm início. Depois que as operações são concluídas com sucesso, a entrada é excluída do log. Se houver alguma parada, é possível, após sua recuperação, que o sistema verifique o log para saber se há alguma operação pendente. Caso haja, todas elas podem ser novamente executadas (várias vezes, no caso de repetidas paradas), até que o arquivo seja removido corretamente.

Para que o *journaling* funcione, as operações registradas no log devem ser **idempotentes**, ou seja, devem poder ser repetidas sempre que necessário sem causarem nenhum dano. Operações como "atualize o mapa e assinale o i-node *k* ou o bloco *n* como livre" podem ser repetidas sem nenhum problema até que se alcance o objetivo final. Analogamente, a busca em um diretório e a remoção de uma entrada denominada *foobar* também são uma operação idempotente. Por outro lado, a inclusão dos novos blocos livres no i-node *K* no final da lista correspondente não é uma operação idempotente, visto que eles podem já estar lá. Uma operação mais cara, "pesquise na lista de blocos livres e inclua o bloco *n* somente se ele ainda não estiver lá", é idempotente. Os sistemas de arquivos *journaling* precisam organizar suas estruturas de dados e operações ligadas ao log de forma que todos sejam idempotentes. Sob essas circunstâncias, a recuperação de travamentos pode ser feita de forma rápida e segura.

Para aumentar a confiança, um sistema de arquivos pode introduzir o conceito de banco de dados denominado **transação atômica**. Quando utilizado, um grupo de ações pode ser formado pelas operações begin transaction e end transaction. Assim, o sistema reconhece que as únicas duas alternativas são concluir todas as operações do grupo ou não concluir nenhuma delas.

O NTFS possui um extenso sistema de *journaling* e sua estrutura dificilmente é corrompida por paradas do sistema. Ele vem sendo desenvolvido desde a primeira versão do Windows NT, em 1993. A primeira versão do Linux a implementar *journaling* foi a ReiserFS, mas sua popularidade foi frustrada por conta de uma incompatibilidade com os sistemas ext2. Em contrapartida, o ext3, que é um projeto menos ambicioso que o do ReiserFS, implementa *journaling* e mantém a compatibilidade com os sistemas ext2 anteriores.

4.3.7 Sistemas de arquivos virtuais

Muitos sistemas de arquivos diferentes estão em uso — em geral, no mesmo computador — e até para o mesmo sistema operacional. Um sistema Windows pode ter um sistema NTFS principal, mas também uma antiga unidade ou partição FAT-16 ou FAT-32, que contenha dados antigos, mas ainda necessários. Também pode ser que, eventualmente, seja necessário utilizar um CD-ROM ou DVD (cada um com seu próprio sistema de arquivos). O Windows gerencia esses sistemas distintos por meio da identificação de cada um com uma letra de unidade diferente — por exemplo, *C:*, *D:* etc. Quando um processo abre um arquivo, a letra da unidade está implícita ou explicitamente presente, de modo que o Windows saiba para qual sistema de arquivos passar a requisição. Não existem tentativas de unificação dos sistemas de arquivos heterogêneos.

Em contrapartida, todos os sistemas UNIX modernos tentam integrar os diferentes sistemas de arquivos em uma única estrutura. Um sistema Linux pode ter o ext2 como diretório-raiz, com uma partição ext3 montada em */usr*, um segundo disco rígido com um sistema de arquivos ReiserFS montado em */home* e um CD-ROM ISO 9660 temporariamente montado em */mnt*. Da perspectiva do usuário, existe somente uma hierarquia de sistema de arquivos, e o fato de o sistema lidar com diferentes tipos (incompatíveis) não fica visível nem ao usuário nem aos processos.

A presença de múltiplos sistemas de arquivos, contudo, é totalmente visível à implementação e, desde o trabalho pioneiro da Sun Microsystems (Kleiman, 1986), a maior parte dos sistemas UNIX passou a usar o conceito de **VFS** (*virtual file system* — sistema de arquivos virtual) para tentar integrar diferentes sistemas de arquivos em uma estrutura ordenada. A ideia principal é abstrair a parte comum aos diferentes sistemas e colocar o código em uma camada separada que chama o sistema de arquivos subjacente para fazer o gerenciamento do dado. A estrutura geral é ilustrada na Figura 4.16. A discussão a seguir não é exclusiva do Linux, do FreeBSD nem de nenhuma outra versão do UNIX, mas dá uma ideia geral de como os sistemas de arquivos virtuais funcionam nos sistemas UNIX.

Todas as chamadas de sistema relacionadas a arquivos são direcionadas ao VFS para o processamento inicial. Essas chamadas, oriundas de processos do usuário, são as chamadas POSIX padrão, como open, read, write, lseek etc. Portanto, o VFS possui uma interface 'superior' com os processos do usuário: a já conhecida interface POSIX.

O VFS também possui uma interface 'inferior' com os arquivos do sistema, denominada na Figura 4.16 como **interface VFS**. Ela consiste de algumas chamadas de funções que podem ser realizadas pelo VFS de forma a fazer com que o sistema correspondente realize suas tarefas. Para criar um novo sistema de arquivos que trabalhe com VFS, portanto, os projetistas devem se certificar de que ele oferece as chamadas requeridas pelo VFS. Um exemplo óbvio desse tipo de chamada é aquela que lê um bloco específico do disco, armazena o conteúdo lido na cache de buffer do sistema de arquivos e retorna um ponteiro para o local. Assim sendo, o VFS possui duas interfaces: a superior — com os processos do usuário — e a inferior, com os arquivos do sistema.

Não é sempre que os sistemas gerenciados pelo VFS representam partições em um disco local. Na verdade, a motivação original da Sun para construir o VFS foi dar suporte a sistemas de arquivos remotos utilizando o protocolo **NFS** (*network file system* — sistema de arquivos de rede). O projeto do NFS é tão ambicioso que, desde que o sistema

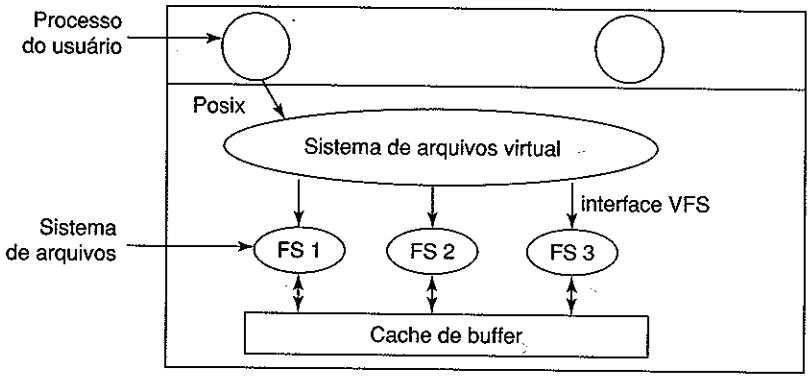

Figura 4.16 Posição do sistema de arquivos virtual.

de arquivos real forneça ao VFS as funções requeridas, o sistema virtual não precisará saber ou se preocupar com o local de armazenamento dos dados ou com o tipo de sistema de arquivos em uso.

Internamente, a maior parte das implementações VFS é essencialmente orientada a objetos, mesmo que sejam escritas em C e não em C++. Existem alguns tipos principais de objetos normalmente suportados que incluem o superbloco (que descreve um sistema de arquivos), o v-node (que descreve um arquivo) e o diretório (que descreve um diretório do sistema de arquivos). Cada um deles tem um grupo de operações associadas (métodos) que o sistema de arquivos real deve dar suporte. Além disso, o VFS possui algumas estruturas de dados internas para seu próprio uso, incluindo a tabela de montagem e um arranjo de descritores de arquivos para controlar todos os arquivos abertos nos processos do usuário.

Para entender como funciona o VFS, vamos executar cronologicamente um exemplo. Quando o sistema é inicializado, o sistema de arquivos raiz é registrado com o VFS. Além disso, quando outros sistemas de arquivos são montados, seja no momento da inicialização seja durante a operação, eles também devem ser registrados com o VFS. Quando se registra, o que o sistema de arquivos faz é fornecer uma lista dos endereços das funções exigidas pelo VFS, seja no formato de um único arranjo de chamadas (tabela) ou como vários deles, um para cada objeto, conforme solicita o VFS. Uma vez que um sistema de arquivos tenha se registrado, o VFS sabe como ler um de seus blocos — simplesmente chamando a função equivalente no arranjo fornecido pelo sistema de arquivos. O VFS também sabe como executar cada uma das outras funções que os sistemas de arquivos reais devem fornecer: ele simplesmente chama a função cujo endereço foi dado durante o registro do sistema de arquivos.

Depois que um sistema de arquivos foi montado, ele pode ser usado. Se um sistema de arquivos foi montado em */usr*, por exemplo, e um processo faz a chamada

```
open("/usr/include/unistd.h",O_RDONLY)
```

durante a análise do caminho, o VFS verifica que um novo sistema foi montado em */usr* e localiza seu superbloco por meio de uma busca na lista de superblocos de sistemas de arquivos montados. Feito isso, é possível encontrar o diretório-raiz do sistema montado e pesquisar pelo caminho *include/unistd.h*. Então, o VFS cria um v-node (na RAM), junto com outras informações, e, o mais importante, cria um ponteiro apontando para a tabela de funções para chamada de operações no v-node, como read, write, close etc.

Após a criação do v-node, o VFS registra uma entrada para o processo na tabela de descritores de arquivos e faz com que ela aponte para o novo v-node. (Para os puristas, o que o descritor do arquivo faz, na verdade, é apontar para outra estrutura de dados que contém a posição atual do arquivo e um ponteiro para o v-node, mas esse detalhe não é relevante para os nossos propósitos.) Finalmente, o VFS retorna o descritor do arquivo para o processo, de modo que a informação possa ser usada em operações de leitura, escrita e fechamento do arquivo.

Mais tarde, quando o processo realizar um read com o descritor do arquivo, o VFS localiza o v-node a partir do processo e da tabela de descritores e segue o ponteiro até a tabela de funções, na qual estão os endereços do sistema de arquivos real, nos quais reside o arquivo solicitado. A função responsável pelo read é, então, chamada, e o código do sistema real visita e recupera o bloco selecionado. O VFS não faz ideia se os dados vêm do disco local de um sistema remoto na rede, de um CD-ROM, de um cartão de memória ou de algum outro meio de armazenamento. As estruturas de dados envolvidas estão representadas na Figura 4.17. A localização começa pelo número do processo chamador e pelo descritor do arquivo, segue para o v-node, o ponteiro da função de leitura e a função de acesso dentro do sistema real.

Dessa maneira, torna-se relativamente simples incluir um novo sistema de arquivos. Para fazer um, os projetistas primeiro tomam uma lista de chamadas de funções esperadas pelo VFS e, em seguida, escrevem seu próprio sistema de arquivos, de modo a oferecer todas elas. No caso de o

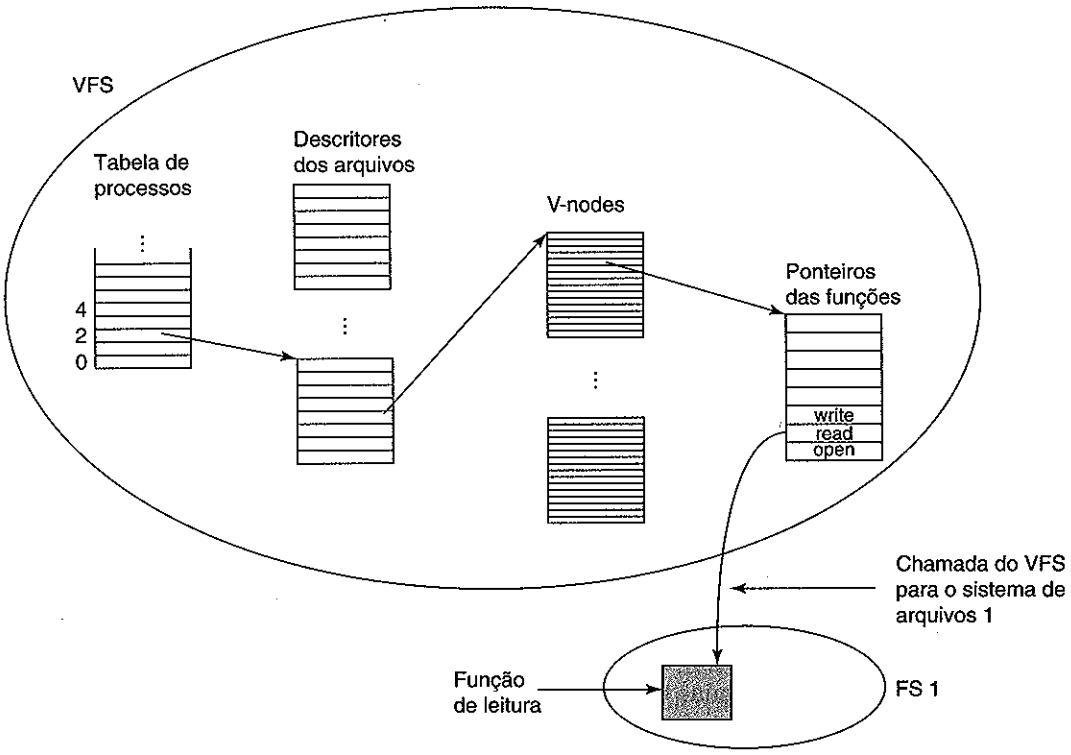

Figura 4.17 Uma visão simplificada das estruturas de dados e do código utilizado pelo VFS e pelo sistema de arquivos real para uma operação read.

sistema de arquivos já existir, será necessário providenciar funções adaptadoras que façam o que o VFS precisa, em geral realizando uma ou mais chamadas nativas ao sistema de arquivos real.

4.4 Gerenciamento e otimização dos sistemas de arquivos

Fazer o sistema de arquivos funcionar é uma coisa; fazê-lo funcionar de forma eficiente e robusta na vida real é algo bastante diferente. Nas seções a seguir, discutiremos algumas questões relacionadas ao gerenciamento de discos.

4.4.1 Gerenciamento de espaço em disco

Os arquivos normalmente são armazenados em disco; portanto, o gerenciamento do espaço em disco é uma das principais preocupações dos projetistas de sistemas. Existem duas estratégias gerais para armazenar um arquivo de n bytes: ou são alocados n bytes consecutivos de espaço em disco, ou o arquivo é dividido em vários blocos (não necessariamente) contíguos. O mesmo compromisso existe para os sistemas de gerenciamento de memória entre segmentação pura e paginação.

Conforme vimos, o armazenamento de um arquivo como uma sequência contígua de bytes apresenta o problema óbvio de que, se o arquivo cresce, provavelmente ele deverá ser movido dentro do disco. O mesmo problema ocorre para segmentos na memória, exceto que o movimento de um segmento na memória é uma operação relativamente rápida se comparada ao movimento de um arquivo de uma posição do disco para outra. Por isso, quase todos os sistemas de arquivos quebram os arquivos em blocos de tamanho fixo, que não precisam ser adjacentes.

Tamanho do bloco

Uma vez que se opta pelo armazenamento em blocos de tamanho fixo, a questão que surge é qual deve ser o tamanho do bloco. Dado o modo como os discos são organizados, o setor, a trilha e o cilindro são candidatos naturais à unidade de alocação (embora sejam todos dependentes do dispositivo, o que é um ponto negativo). Em um sistema de paginação, o tamanho da página é também um argumento importante.

Uma grande unidade de alocação, como um cilindro, significa que cada arquivo, mesmo um arquivo de 1 byte, ocupará um cilindro inteiro. Também significa que arquivos pequenos desperdiçam um espaço significativo do disco. Por outro lado, um tamanho pequeno de bloco significa que a maioria dos arquivos ocupará mais de um bloco e, portanto, demandaram múltiplas buscas e atrasos de rotação para serem lidos, reduzindo o desempenho. Se a unidade de alocação for muito grande, ocorre desperdício de espaço; se for muito pequena, desperdício de tempo.