Capítulo 23

Uso de arquivos

23.1 Introdução

Arquivos são usados por processos para ler e escrever dados em dispositivos de armazenamento, como discos. Para usar arquivos, um processo tem à sua disposição uma *interface de acesso*, que depende da linguagem utilizada e do sistema operacional subjacente. Na sequência desta seção serão discutidos aspectos relativos ao uso de arquivos, como a interface de acesso, as formas de acesso aos dados contidos nos arquivos e problemas relacionados ao compartilhamento de arquivos entre vários processos.

23.2 Interface de acesso

A interface de acesso a um arquivo normalmente é composta por uma representação lógica do arquivo, denominada **descritor de arquivo** (*file descriptor* ou *file handle*), e um conjunto de funções para manipular o arquivo. Através dessa interface, um processo pode localizar o arquivo no dispositivo físico, ler e modificar seu conteúdo, entre outras operações.

Na verdade, existem dois níveis de interface de acesso: uma **interface de baixo nível**, oferecida pelo sistema operacional aos processos através de chamadas de sistema, e uma **interface de alto nível**, composta de funções na linguagem de programação usada para implementar cada aplicação.

A interface de baixo nível é definida por chamadas de sistema, pois os arquivos são abstrações criadas e mantidas pelo núcleo do sistema operacional. Por essa razão, essa interface é dependente do sistema operacional subjacente. A Tabela 23.1 traz alguns exemplos de chamadas de sistema oferecidas pelos sistemas operacionais Linux e Windows para o acesso a arquivos:

Por outro lado, a interface de alto nível é específica para cada linguagem de programação e normalmente não depende do sistema operacional subjacente. Esta independência auxilia a portabilidade de programas entre sistemas operacionais distintos. A interface de alto nível é implementada sobre a interface de baixo nível, geralmente na forma de uma biblioteca de funções e/ou um suporte de execução (*runtime*). A Tabela 23.2 apresenta algumas funções/métodos para acesso a arquivos das linguagens C e Java.

Tabela 23.1: Chamadas de sistema para arquivos

Operação	**Linux**	**Windows**
Abrir arquivo	`OPEN`	`NtOpenFile`
Ler dados	`READ`	`NtReadRequestData`
Escrever dados	`WRITE`	`NtWriteRequestData`
Fechar arquivo	`CLOSE`	`NtClose`
Remover arquivo	`UNLINK`	`NtDeleteFile`
Criar diretório	`MKDIR`	`NtCreateDirectoryObject`

Tabela 23.2: Funções de biblioteca para arquivos (`fd`: *file descriptor*, `obj`: objeto)

Operação	**C (padrão C99)**	**Java (classe File)**
Abrir arquivo	`fd = fopen(...)`	`obj = File(...)`
Ler dados	`fread(fd, ...)`	`obj.read()`
Escrever dados	`fwrite(fd, ...)`	`obj.write()`
Fechar arquivo	`fclose(fd)`	`obj.close()`
Remover arquivo	`remove(...)`	`obj.delete()`
Criar diretório	`mkdir(...)`	`obj.mkdir()`

A Figura 23.1 ilustra a interface de acesso em dois níveis de abstração. Nela, pode-se observar que os processos são estruturados em dois níveis: o código da aplicação propriamente dita, que efetua chamadas de funções/métodos na linguagem em que foi programada, e o suporte de execução, que traduz as chamadas de função recebidas da aplicação nas chamadas de sistema aceitas pelo sistema operacional subjacente. As chamadas de sistema são recebidas pela interface de sistema de arquivos do núcleo, que as encaminha para as rotinas que implementam as ações correspondentes.

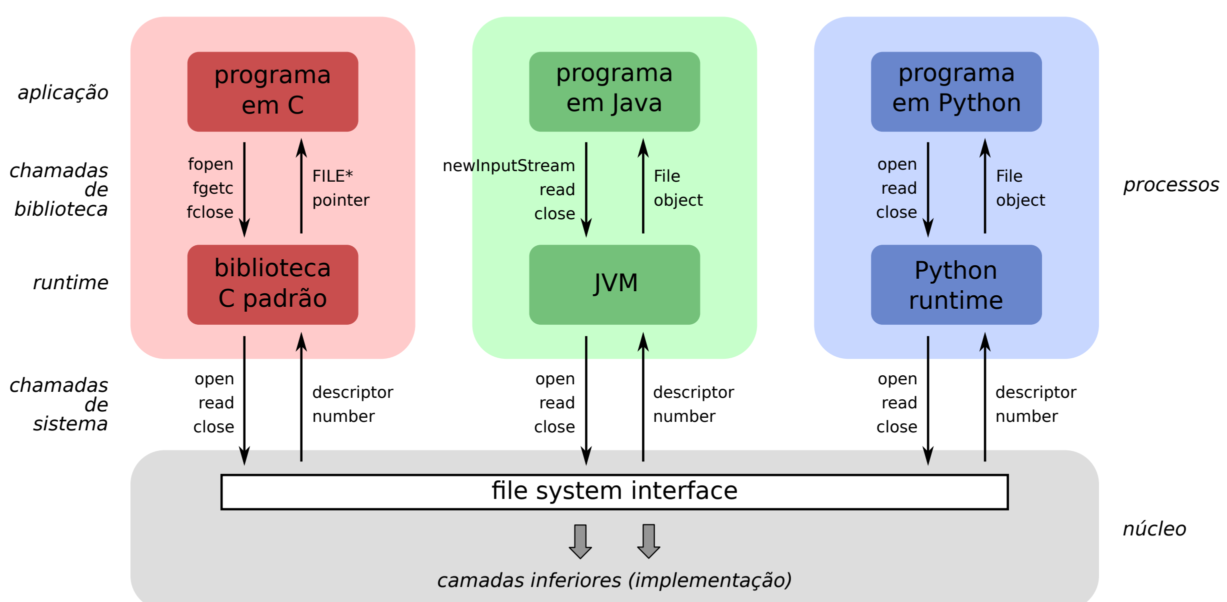

Figura 23.1: Relação entre funções de linguagem e chamadas de sistema.

23.2.1 Descritores de arquivos

Um descritor de arquivo é uma representação lógica de um arquivo em uso por um processo. O descritor é criado no momento da abertura do arquivo e serve como uma referência ao mesmo nas operações de acesso subsequentes. É importante observar que os descritores de arquivo usados nos dois níveis de interface geralmente são distintos.

Descritores de alto nível representam arquivos dentro de uma aplicação. Eles dependem da linguagem de programação escolhida, pois são implementados pelas bibliotecas que dão suporte à linguagem. Por outro lado, independem do sistema operacional subjacente, facilitando a portabilidade da aplicação entre SOs distintos. Por exemplo, descritores de arquivos usados na linguagem C são ponteiros para estruturas do tipo `FILE` (ou seja, `FILE*`), mantidas pela biblioteca C. Da mesma forma, descritores de arquivos usados em Java e Python são objetos de classes específicas. Em Java, por exemplo, os descritores de arquivos abertos são objetos instanciados a partir da classe `File`.

Por outro lado, os **descritores de baixo nível** não são ligados a uma linguagem específica e variam conforme o sistema operacional. Em sistemas POSIX/UNIX, o descritor de arquivo de baixo nível é um número inteiro positivo ($n \geq 0$), que indica simplesmente a posição do arquivo correspondente em uma tabela de arquivos abertos mantida pelo núcleo. Por outro lado, em sistemas Windows, os arquivos abertos por um processo são representados pelo núcleo por **referências de arquivos** (*file handles*), que são estruturas de dados criadas pelo núcleo para representar cada arquivo aberto.

Dessa forma, cabe às bibliotecas e ao suporte de execução de cada linguagem de programação mapear a representação de arquivo aberto fornecida pelo núcleo do sistema operacional subjacente na referência de arquivo aberto usada por aquela linguagem. Esse mapeamento é necessário para garantir que as aplicações que usam arquivos (ou seja, quase todas elas) sejam portáveis entre sistemas operacionais distintos.

23.2.2 A abertura de um arquivo

Para poder ler ou escrever dados em um arquivo, cada aplicação precisa "abri-lo" antes. A **abertura de um arquivo** consiste basicamente em preparar as estruturas de memória necessárias para acessar os dados do arquivo. Assim, na abertura de um arquivo, os seguintes passos são realizados:

1. No processo:
   (a) a aplicação solicita a abertura do arquivo (`fopen()`, se for um programa C);
   (b) o suporte de execução da aplicação recebe a chamada de função, trata os parâmetros recebidos e invoca uma chamada de sistema para abrir o arquivo.
2. No núcleo:
   (a) o núcleo recebe a chamada de sistema;
   (b) localiza o arquivo no dispositivo físico, usando seu nome e caminho de acesso;

(c) verifica se o processo tem as permissões necessárias para usar aquele arquivo da forma desejada (vide Seção 23.5);

(d) cria uma estrutura de dados na memória do núcleo para representar o arquivo aberto;

(e) insere uma referência a essa estrutura na relação de arquivos abertos mantida pelo núcleo, para fins de gerência;

(f) devolve à aplicação uma referência a essa estrutura (o descritor de baixo nível), para ser usada nos acessos subsequentes ao arquivo.

3. No processo:

(a) o suporte de execução recebe do núcleo o descritor de baixo nível do arquivo;

(b) o suporte de execução cria um descritor de alto nível e o devolve ao código da aplicação;

(c) o código da aplicação recebe o descritor de alto nível do arquivo aberto, para usar em suas operações subsequentes envolvendo aquele arquivo.

Assim que o processo tiver terminado de usar um arquivo, ele deve solicitar ao núcleo o **fechamento do arquivo**, que implica em concluir as operações de escrita eventualmente pendentes e remover da memória do núcleo as estruturas de dados criadas durante sua abertura. Normalmente, os arquivos abertos são automaticamente fechados quando o processo é encerrado, mas pode ser necessário fechá-los antes disso, caso seja um processo com vida longa, como um *daemon* servidor de páginas Web, ou que abra muitos arquivos, como um compilador.

23.3 Formas de acesso

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

23.3.1 Acesso sequencial

No **acesso sequencial**, os dados são sempre lidos e/ou escritos em sequência, do início ao final do arquivo. Para cada arquivo aberto por uma aplicação é definido um *ponteiro de acesso*, que inicialmente aponta para a primeira posição do arquivo. A cada leitura ou escrita, esse ponteiro é incrementado e passa a indicar a posição da próxima leitura ou escrita. Quando esse ponteiro atinge o final do arquivo, as leituras não são mais permitidas, mas as escritas podem sê-lo, permitindo acrescentar dados ao final do mesmo. A chegada do ponteiro ao final do arquivo é normalmente sinalizada ao processo através de um *flag* de fim de arquivo (*EoF – End-of-File*).

A Figura 23.2 traz um exemplo de acesso sequencial em leitura a um arquivo, mostrando a evolução do ponteiro do arquivo durante uma sequência de leituras. Uma primeira leitura de 15 bytes do arquivo traz os caracteres "`Qui␣scribit␣bis`"; uma segunda leitura de mais 9 bytes traz "`␣legit.␣`", e assim sucessivamente. O acesso

sequencial é implementado em praticamente todos os sistemas operacionais de mercado e constitui a forma mais usual de acesso a arquivos, usada pela maioria das aplicações.

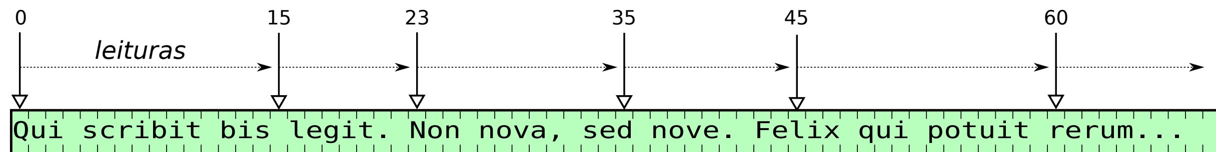

Figura 23.2: Leituras sequenciais em um arquivo de texto.

23.3.2 Acesso aleatório

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

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

23.3.3 Acesso mapeado em memória

Uma forma particular de acesso aleatório ao conteúdo de um arquivo é o **mapeamento em memória** do mesmo, que faz uso dos mecanismos de paginação em disco (Capítulo 17). Nessa modalidade de acesso, o arquivo é associado a um vetor de bytes (ou de registros) de mesmo tamanho na memória principal, de forma que cada posição do vetor corresponda à sua posição equivalente no arquivo. Quando uma posição específica do vetor na memória é lida pela primeira vez, é gerada uma falta de página. Nesse momento, o mecanismo de memória virtual intercepta o acesso à memória, lê o conteúdo correspondente no arquivo e o deposita no vetor, de forma transparente à aplicação, que em seguida pode acessá-lo. Escritas no vetor são transferidas para o arquivo por um procedimento similar. Caso o arquivo seja muito grande, pode-se mapear em memória apenas partes dele. A Figura 23.3 ilustra essa forma de acesso.

O acesso mapeado em memória é extensivamente usado pelo núcleo para carregar código executável (programas e bibliotecas) na memória. Como somente as partes efetivamente acessadas do código serão carregadas em RAM, esse procedimento é usualmente conhecido como *paginação sob demanda* (*demand paging*), pois os dados são lidos do arquivo para a memória em páginas.

23.3.4 Acesso indexado

Alguns sistemas operacionais oferecem também a possibilidade de **acesso indexado** aos dados de um arquivo, como é o caso do *OpenVMS* [Rice, 2000]. Esse

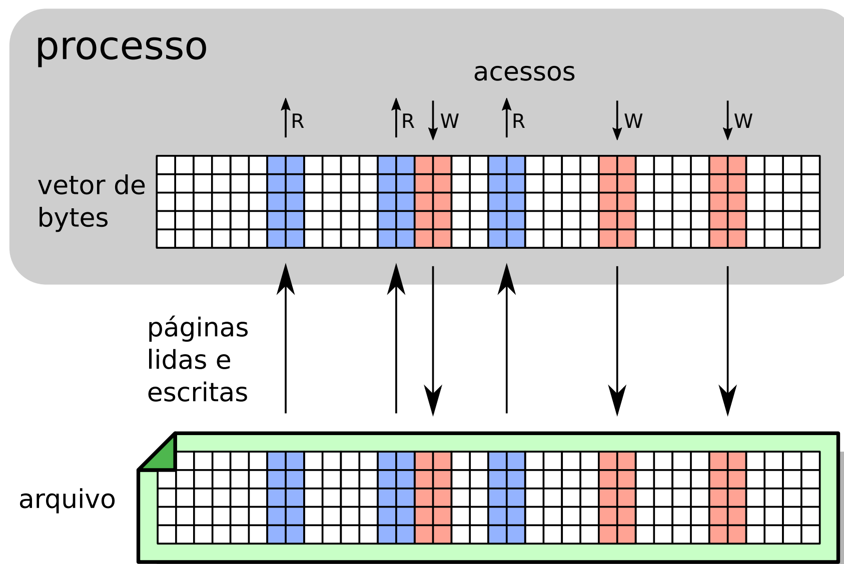

Figura 23.3: Arquivo mapeado em memória.

sistema implementa arquivos cuja estrutura interna pode ser vista como uma tabela de pares *chave/valor*. Os dados do arquivo são armazenados em registros com chaves (índices) associados a eles, e podem ser recuperados usando essas chaves, como em um banco de dados relacional.

Como o próprio núcleo desse sistema implementa os mecanismos de acesso e indexação do arquivo, o armazenamento e busca de dados nesse tipo de arquivo costuma ser muito rápido, dispensando bancos de dados para a construção de aplicações mais simples. A maioria dos sistemas operacionais de mercado não implementa essa funcionalidade diretamente no núcleo, mas ela pode ser facilmente obtida através de bibliotecas populares como BerkeleyDB ou SQLite.

23.4 Compartilhamento de arquivos

Em um sistema multitarefas, é frequente ter arquivos acessados por mais de um processo, ou mesmo mais de um usuário. Conforme estudado no Capítulo 10, o acesso simultâneo a recursos compartilhados pode gerar condições de disputa (*race conditions*), que levam à inconsistência de dados e outros problemas. O acesso concorrente em leitura a um arquivo não acarreta problemas, mas a possibilidade de escritas e leituras concorrentes pode levar a condições de disputa, precisando ser prevista e tratada de forma adequada.

23.4.1 Travas em arquivos

A solução mais simples e mais frequentemente utilizada para gerenciar o acesso concorrente a arquivos é o uso de travas de exclusão mútua (*mutex locks*), como as estudadas no Capítulo 11. A maioria dos sistemas operacionais oferece algum mecanismo de sincronização para o acesso a arquivos, na forma de uma ou mais travas (*locks*) associadas a cada arquivo aberto. A sincronização pode ser feita sobre o arquivo inteiro ou sobre algum trecho específico dele, permitindo que dois ou mais processos possam trabalhar em partes distintas de um arquivo sem conflitos.

Vários tipos de travas podem ser oferecidas pelos sistemas operacionais. Do ponto de vista da rigidez das travas, elas podem ser:

Travas obrigatórias (*mandatory locks*): são impostas pelo núcleo de forma incontornável: se um processo obtiver a trava de um arquivo, outros processos que solicitarem acesso ao mesmo arquivo serão suspensos até que aquela trava seja liberada. É o tipo de trava mais usual em sistemas Windows.

Travas recomendadas (*advisory locks*): não são impostas pelo núcleo do sistema operacional, mas gerenciadas pelo suporte de execução (bibliotecas). Os processos envolvidos no acesso aos mesmos arquivos devem travá-los explicitamente quando forem acessá-los. Contudo, um processo pode ignorar essa regra e acessar um arquivo ignorando a trava, caso necessário. Travas recomendadas são úteis para gerenciar concorrência entre processos de uma mesma aplicação. Neste caso, cabe ao programador implementar os controles necessários nos processos para impedir acessos conflitantes aos arquivos compartilhados.

Em relação ao compartilhamento das travas de arquivos, elas podem ser:

Travas exclusivas (ou *travas de escrita*): garantem acesso exclusivo ao arquivo: enquanto uma trava exclusiva estiver ativa, nenhum outro processo poderá obter outra trava sobre o mesmo arquivo.

Travas compartilhadas (ou *travas de leitura*): impedem outros processos de criar travas exclusivas sobre aquele arquivo, mas permitem a existência de outras travas compartilhadas.

É fácil observar que as travas exclusivas e compartilhadas implementam o modelo de sincronização *leitores/escritores* (descrito na Seção 12.2), no qual os leitores acessam o arquivo usando travas compartilhadas e os escritores o fazem usando travas exclusivas.

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

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

23.4.2 Semântica de acesso

Quando um arquivo é aberto e usado por um único processo, o funcionamento das operações de leitura e escrita é simples e inequívoco: quando um dado é escrito no arquivo, ele está prontamente disponível para leitura se o processo desejar lê-lo novamente. No entanto, arquivos podem ser abertos por vários processos simultaneamente, e os dados escritos por um processo podem não estar prontamente disponíveis aos demais processos que leem aquele arquivo. Isso ocorre porque os discos rígidos são

normalmente lentos, o que leva os sistemas operacionais a usar *buffers* intermediários para acumular os dados a escrever e assim otimizar o acesso aos discos.

A forma como os dados escritos por um processo são percebidos pelos demais processos que abriram aquele mesmo arquivo é chamada de *semântica de compartilhamento*. Existem várias semânticas possíveis, mas as mais usuais são [Silberschatz et al., 2001]:

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

Semântica UNIX: toda modificação em um arquivo é imediatamente visível a todos os processos que mantêm aquele arquivo aberto; Essa semântica é a mais comum em sistemas de arquivos locais, ou seja, para acesso a arquivos nos dispositivos locais;

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

Semântica de transação: uma *transação* é uma sequência de operações de leitura e escrita em um ou mais arquivos emitidas por um processo e delimitadas por comandos de início e fim de transação (`begin ... end`), como em um sistema de bancos de dados. Todas as modificações parciais do arquivo durante a execução de uma transação não são visíveis às demais transações, somente após sua conclusão. Pode-se afirmar que a semântica de transação é similar à semântica de sessão, mas aplicada a cada transação (sequência de operações) e não ao período completo de uso do arquivo (da abertura ao fechamento).

A Figura 23.4 traz um exemplo de funcionamento da semântica de sessão: os processos p_1 a p_4 compartilham o acesso ao mesmo arquivo, que contém apenas um número inteiro, com valor inicial 23 (cada escrita substitui o valor contido no arquivo). Pode-se perceber que o valor 39 escrito por p_1 é visto por ele na mesma sessão, mas não é visto por p_2, que abriu o arquivo antes do fim da sessão de p_1. O processo p_3 vê o valor 39, pois abriu o arquivo depois que p_1 o fechou, mas não vê o valor 71 escrito por p_2. Da mesma forma, p_4 lê o valor 71 escrito por p_2, mas não percebe o valor 6 escrito por p_3.

Em termos práticos, pode-se imaginar o comportamento da semântica de sessão desta forma: cada processo faz uma cópia do arquivo ao abri-lo, trabalha sobre essa cópia e atualiza o conteúdo do arquivo original ao fechar sua cópia. Essa ideia de "cópia local" torna a semântica de sessão adequada para a implementação de sistemas de arquivos distribuídos.

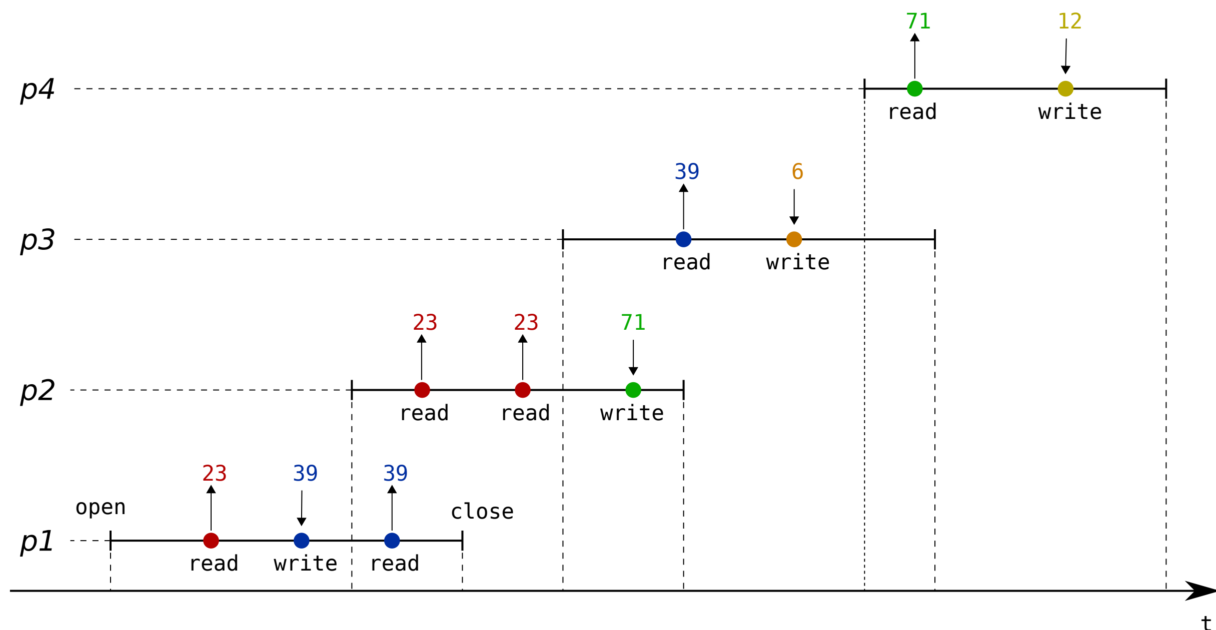

Figura 23.4: Compartilhamento de arquivo usando a semântica de sessão.

23.5 Controle de acesso

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

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

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

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

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

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

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

```
host:~> ls -l
- rwx r-x --- 1 maziero prof    7248 2008-08-23 09:54 hello-unix
- rw- r-- r-- 1 maziero prof      54 2008-08-23 09:54 hello-unix.c
- rw- r-- r-- 1 maziero prof  195780 2008-09-26 22:08 main.pdf
- rw- --- --- 1 maziero prof   40494 2008-09-27 08:44 main.tex
```

Nessa listagem, o arquivo `hello-unix.c` (linha 3) pode ser acessado em leitura e escrita (`rw-`) por seu proprietário (`maziero`), em leitura (`r--`) pelos usuários do grupo `prof` e em leitura (`r--`) pelos demais usuários do sistema. Já o arquivo `hello-unix` (linha 2) pode ser acessado em leitura, escrita e execução (`rwx`) por seu proprietário, em leitura e execução (`r-x`) pelos usuários do grupo `prof` e não pode ser acessado (`---`) pelos demais usuários.

No mundo Windows, o sistema de arquivos NTFS implementa um controle de acesso bem mais flexível que o controle básico do UNIX[1], que define permissões aos proprietários de forma similar, mas no qual permissões complementares a usuários individuais podem ser associadas a qualquer arquivo. Mais detalhes sobre os modelos de controle de acesso usados nos sistemas UNIX e Windows podem ser encontrados no Capítulo 29.

É importante destacar que o controle de acesso é geralmente realizado somente durante a abertura do arquivo, para a criação de sua referência em memória. Isso significa que, uma vez aberto um arquivo por um processo, este terá acesso ao conteúdo do mesmo enquanto o mantiver aberto, mesmo que as permissões do arquivo sejam alteradas para impedir esse acesso. O controle contínuo de acesso aos arquivos é raramente implementado em sistemas operacionais, porque verificar as permissões de acesso a cada operação de leitura ou escrita em um arquivo teria um impacto negativo significativo sobre o desempenho do sistema.

23.6 Interface de acesso

Como visto na Seção 23.2, cada linguagem de programação define sua própria forma de representar arquivos abertos e as funções ou métodos usados para manipulá-los. A título de exemplo, será apresentada uma visão geral da interface para arquivos oferecida pela linguagem C no padrão ANSI [Kernighan and Ritchie, 1989].

Em C, cada arquivo aberto é representado por uma variável dinâmica do tipo `FILE`, criada pela função `fopen`. As funções de acesso a arquivos são definidas na **Biblioteca Padrão de Entrada/Saída** (*Standard I/O Library*, cuja interface é definida no arquivo de cabeçalho `stdio.h`). Algumas das funções mais usuais dessa biblioteca são apresentadas a seguir:

[1]Sistemas UNIX oferecem mecanismos de controle de acesso mais elaborados, como as ACLs estendidas, mas que fogem ao escopo deste texto.

- Abertura e fechamento de arquivos:
  - `FILE * fopen (const char *filename, const char *opentype)`: abre o arquivo cujo nome é indicado por `filename`; a forma de abertura (leitura, escrita, etc.) é indicada pelo parâmetro `opentype`; em caso de sucesso, devolve uma referência ao arquivo;
  - `int fclose (FILE *f)`: fecha o arquivo referenciado pelo descritor `f`;
- Leitura e escrita de caracteres e *strings*:
  - `int fputc (int c, FILE *f)`: escreve um caractere no arquivo;
  - `int fgetc (FILE *f)`: lê um caractere do arquivo;
- Reposicionamento do ponteiro do arquivo:
  - `long int ftell (FILE *f)`: indica a posição corrente do ponteiro de acesso do arquivo referenciado por `f`;
  - `int fseek (FILE *f, long int offset, int whence)`: move o ponteiro do arquivo para a posição indicada por `offset`;
  - `void rewind (FILE *f)`: retorna o ponteiro de acesso ao início do arquivo;
  - `int feof (FILE *f)`: indica se o ponteiro chegou ao final do arquivo;
- Tratamento de travas:
  - `void flockfile (FILE *f)`: solicita acesso exclusivo ao arquivo, podendo bloquear o processo solicitante caso o arquivo já tenha sido travado por outro processo;
  - `void funlockfile (FILE *f)`: libera a trava de acesso ao arquivo.

O exemplo a seguir ilustra o uso de algumas dessas funções. Esse programa abre um arquivo chamado `numeros.dat` para operações de leitura (linha 9), verifica se a abertura do arquivo foi realizada corretamente (linhas 11 a 15), lê seus caracteres e os imprime na tela até encontrar o fim do arquivo (linhas 17 a 22) e finalmente o fecha (linha 24) e encerra a execução.

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

int main (int argc, char *argv[])
{
  FILE *arq ;              // descritor de arquivo
  char c ;

  arq = fopen ("infos.dat", "r") ; // abre arquivo para leitura

  if (! arq)               // descritor inválido
  {
    perror ("Erro ao abrir arquivo") ;
    exit (1) ;
  }

  c = getc (arq) ;         // le um caractere do arquivo
  while (! feof (arq))     // enquanto não chegar ao fim do arquivo
  {
    putchar (c) ;          // imprime o caractere na tela
    c = getc (arq) ;       // le um caractere do arquivo
  }

  fclose (arq) ;           // fecha o arquivo
  exit (0) ;
}
```

Exercícios

1. Descreva o que são, onde/como são usados e quais as diferenças entre descritores de arquivo de alto nível e de baixo nível.

2. O que é um ponteiro de arquivo? Para que ele serve?

3. Comente as principais formas de acesso a arquivos. Qual o uso mais apropriado para cada uma delas?

4. Apresente e explique os quatro principais tipos de travas sobre arquivos compartilhados disponíveis no sistema operacional.

5. Apresente e explique as quatro principais semânticas de acesso a arquivos compartilhados em um sistema operacional.

6. Sobre as afirmações a seguir, relativas ao uso de arquivos, indique quais são incorretas, justificando sua resposta:

   (a) No acesso sequencial, o ponteiro de posição corrente do arquivo é reiniciado a cada operação.

   (b) O acesso direto pode ser implementado usando o acesso sequencial e operações de posicionamento do ponteiro do arquivo.

   (c) No acesso mapeado em memória, o conteúdo do arquivo é copiado para a memória RAM durante a sua abertura.

(d) O acesso indexado é raramente implementado pelo núcleo em sistemas operacionais *desktop*, sendo mais frequente em ambientes *mainframe*.

(e) Travas de uso exclusivo e compartilhado implementam um modelo de sincronização de tipo *produtor/consumidor* no acesso ao arquivo.

(f) Segundo a semântica de compartilhamento UNIX, o conteúdo de um arquivo é considerado imutável durante um compartilhamento.

7. Um conjunto de processos p_1, p_2, p_3 e p_4 abrem em leitura/escrita um arquivo compartilhado contendo um número inteiro, cujo valor inicial é 34. As operações realizadas pelos processos são indicadas na tabela a seguir no formato $[t, op]$, onde t é o instante da operação e op é a operação realizada:

p_1	p_2	p_3	p_4
$[0, open]$	$[3, open]$	$[7, open]$	$[9, open]$
$[2, write\ 41]$	$[6, write\ 27]$	$[8, read\ X]$	$[11, read\ Y]$
$[6, close]$	$[8, close]$	$[9, write\ 4]$	$[12, close]$
		$[10, close]$	

Considerando a semântica de sessão para o compartilhamento de arquivos, determine os valores de X e Y, **explicando seu raciocínio**. Cada operação de escrita no arquivo substitui o valor anterior.

Referências

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

L. Rice. *Introduction to OpenVMS*. Elsevier Science & Technology Books, 2000.

A. Silberschatz, P. Galvin, and G. Gagne. *Sistemas Operacionais – Conceitos e Aplicações*. Campus, 2001.