15

UNIX

15.1 Histórico

Na década de 1960, inúmeros esforços foram direcionados para o desenvolvimento de um verdadeiro sistema operacional de tempo compartilhado que viesse a substituir os sistemas batch da época. Em 1965, o MIT (Massachusetts Institute of Technology), a Bell Labs e a General Electric se uniram para desenvolver o MULTICS (MULTiplexed Information and Computing Service). Em 1969, a Bell Labs retirou-se do projeto, porém um de seus pesquisadores nele envolvido, Ken Thompson, desenvolveu sua própria versão do sistema operacional, que veio a se chamar UNICS (UNiplexed Information and Computing Service) e, posteriormente, Unix.

O Unix foi inicialmente desenvolvido em Assembly para um minicomputador PDP-7 da Digital. Para torná-lo mais fácil de ser portado para outras plataformas, Thompson desenvolveu uma linguagem de alto nível chamada B e reescreveu o código do sistema nessa nova linguagem. Em função das limitações da linguagem B, Thompson e Dennis Ritchie, também da Bell Labs, desenvolveram a linguagem C, na qual o Unix seria reescrito e, posteriormente, portado para um minicomputador PDP-11 em 1973.

No ano seguinte, Ritchie e Thompson publicaram o artigo *The Unix Timesharing System* (Ritchie e Thompson, 1974), que motivou a comunidade acadêmica a solicitar uma cópia do sistema. Na época, a Bell Labs era uma subsidiária da AT&T e, mesmo tendo criado e desenvolvido o Unix, não podia comercializá-lo devido às leis americanas antimonopólio, que impediam seu envolvimento no mercado de computadores. Apesar dessa limitação, as universidades poderiam licenciar o Unix, recebendo inclusive o código-fonte do sistema. Como a grande maioria das universidades utilizava computadores da linha PDP-11, não existiam dificuldades para se adotar o sistema como plataforma-padrão no meio acadêmico.

Uma das primeiras instituições de ensino a licenciar o Unix foi a universidade de Berkeley, na Califórnia. A universidade desenvolveu sua própria versão do sistema, batizada de 1BSD (First Berkeley Software Distribution), seguida por outras versões, chegando até a 4.4BSD, quando o projeto acadêmico foi encerrado. O Unix de Berkeley introduziu inúmeros melhoramentos no sistema, merecendo destaque o mecanismo de

memória virtual, C shell, Fast File System, sockets e o protocolo TCP/IP. Em função dessas facilidades, vários fabricantes passaram a utilizar o BSD como base para seus próprios sistemas, como a Sun Microsystems e a Digital. Para dar continuidade ao desenvolvimento do Unix de Berkeley foi criada a Berkeley Software Design que, posteriormente, lançaria o sistema FreeBSD.

Em 1982, a AT&T foi autorizada a comercializar o sistema que tinha desenvolvido. A primeira versão lançada foi a System III, logo seguida pela System V. Diversas versões foram posteriormente desenvolvidas, sendo a versão System V Release 4 (SVR4) a que teve a maior importância. Vários fabricantes basearam seus sistemas no Unix da AT&T, como a IBM, a HP e a SCO. Em 1993, o Unix da AT&T é comercializado para a Novell, que, posteriormente, lança o UnixWare, com base nesse sistema. No mesmo ano, a Novell transfere os direitos sobre a marca Unix para o consórcio X/Open e, posteriormente, vende o UnixWare para a SCO.

Em 1991, o finlandês Linus Torvalds começou o desenvolvimento do Linux, com base em suas experiências com o sistema Minix. O Minix foi desenvolvido pelo professor Andrew Tanenbaum, da Universidade Vrije, na Holanda, com fins apenas educacionais, e pode ser obtido livremente, incluindo o código-fonte (Minix, 2002). O Linux evoluiu a partir da colaboração de vários programadores que ajudaram no desenvolvimento do kernel, utilitários e vários aplicativos. Atualmente, o Linux é utilizado para fins tanto acadêmicos como comerciais e pode ser obtido sem custos, acompanhado do seu código-fonte (Linux, 2002). No Brasil existe o sistema Tropix, desenvolvido pelos pesquisadores Oswaldo Vernet e Pedro Salenbauch, do Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro. O código-fonte do Tropix está disponível na Internet gratuitamente (Tropix, 2002).

Várias tentativas foram feitas visando unificar as versões do Unix de Berkeley (BSD) e da AT&T (System V), além das inúmeras outras implementações oferecidas pelo mercado. A AT&T publicou um conjunto de especificações, conhecidas como System V Interface Definition. Fabricantes ligados à vertente do Unix da AT&T fundaram a Unix International, enquanto os ligados ao Unix de Berkeley criaram a Open Software Foundation (OSF). Nenhuma dessas iniciativas resultou na padronização de um único Unix.

A mais importante tentativa de unificação do Unix nesse sentido foi dada pelo IEEE (Institute of Electrical and Electronics Engineers) através do seu comitê POSIX (Portable Operating System Unix). Como resultado desse trabalho, surgiu o padrão IEEE 1003.1, publicado em 1990, estabelecendo uma biblioteca-padrão de chamadas e um conjunto de utilitários que todo sistema Unix deveria oferecer. Em 1995, o X/Open cria o UNIX95, um programa para garantir uma especificação única do Unix. Posteriormente, o consórcio passa a chamar-se The Open Group e lança a terceira versão de sua especificação, incluindo o padrão POSIX e representantes da indústria.

A Fig. 15.1 apresenta a evolução das duas principais vertentes do Unix e alguns dos sistemas operacionais derivados das versões da AT&T e de Berkeley (Lévénez, 2002). Atualmente, diversas versões do Unix são encontradas não só no meio acadêmico, mas também sendo comercializadas por inúmeros fabricantes, como a Sun Microsystems (SunOS e Solaris), HP (HP-UX), IBM (AIX) e Compaq (Compaq Unix).

As implementações do sistema Unix variam conforme suas versões, plataformas de hardware e fabricantes, principalmente se comparadas às versões originais de Berkeley e da AT&T. No decorrer deste capítulo será apresentada uma visão geral do sistema, sem entrar em detalhes das diferentes implementações.

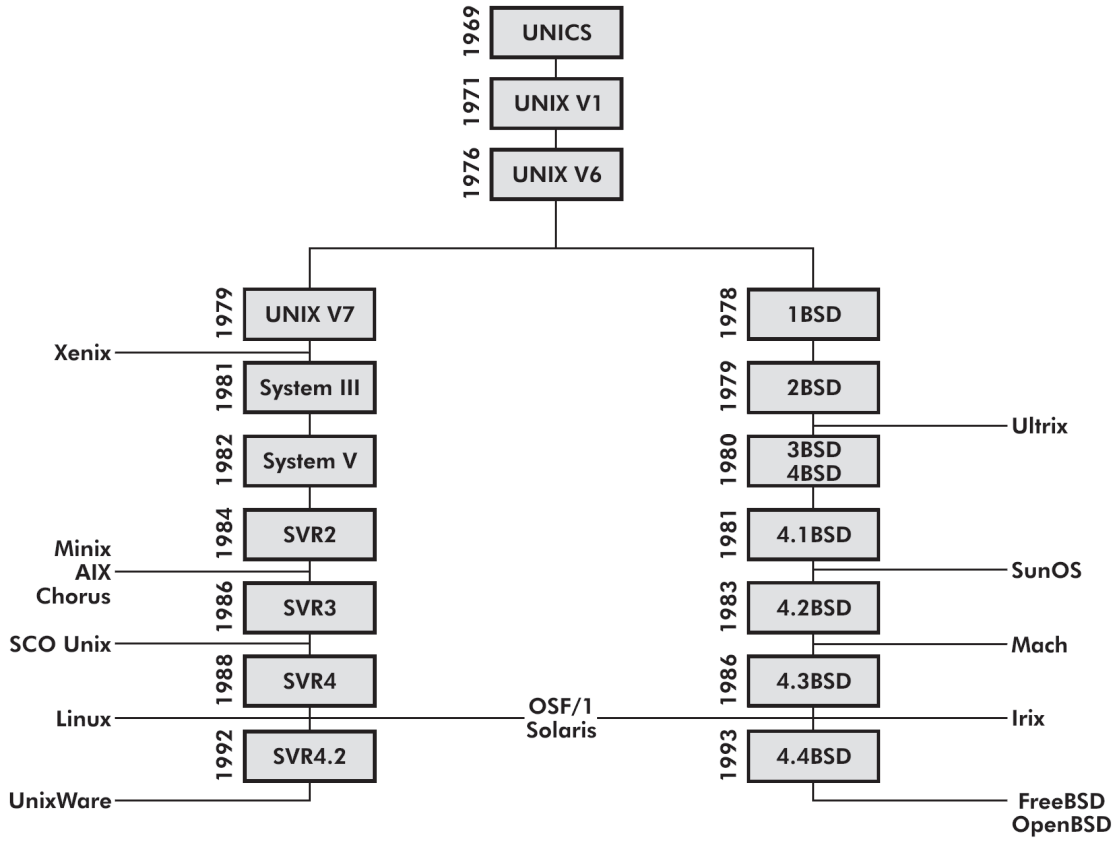


Fig. 15.1 Evolução do Unix.

15.2 Características

O sistema operacional Unix é um sistema multiprogramável, multiusuário, que suporta múltiplos processadores e implementa memória virtual. Entre as muitas razões para explicar o sucesso alcançado pelo Unix incluem-se as características a seguir:

- Escrito em uma linguagem de alto nível, o que torna fácil a compreensão e alteração do seu código e a portabilidade para outras plataformas de hardware.
- Oferece um conjunto de system calls que permite que programas complexos sejam desenvolvidos a partir de uma interface simples.
- Flexibilidade, podendo ser utilizado como sistema operacional de computadores pessoais, estações de trabalho e servidores de todos os portes voltados para banco de dados, Web, correio eletrônico e aplicação.
- Implementação de threads, em algumas versões, e diversos mecanismos de comunicação e sincronização, como memória compartilhada, pipes e semáforos.
- Suporte a um grande número de aplicativos disponíveis no mercado, sendo muitos gratuitos.
- Suporte a diversos protocolos de rede, como o TCP/IP, e interfaces de programação, como sockets, podendo ser utilizado como servidor de comunicação, roteador, firewall e proxy.
- Implementação de sistema de arquivos com uma estrutura bastante simples, em que os arquivos são representados apenas como uma seqüência de bytes. Além disso, existem diversas opções para sistemas de arquivos distribuídos, como NFS (Network File System), AFS (Andrew File System) e DFS (Distributed File System).
- Oferece uma interface simples e uniforme com os dispositivos de E/S.

15.3 Estrutura do Sistema

A maior parte do código que compõe o núcleo do Unix é escrita em Linguagem C, e o restante, como os device drivers, em Assembly, o que confere ao sistema uma grande portabilidade para diferentes plataformas de hardware.

O Unix utiliza o modelo de camadas para a estruturação do sistema, implementando dois níveis de modo de acesso: usuário e kernel. A Fig. 15.2 apresenta as camadas do sistema de forma simplificada, sem a preocupação de representar a estrutura interna real e abranger a maioria das implementações.

- Kernel

  O kernel é responsável por controlar o hardware e fornecer as system calls para que os programas tenham acesso às rotinas do sistema, como criação e gerência de processos, gerência de memória virtual, sistema de arquivos e gerência de E/S.

  O kernel pode ser dividido em duas partes: a parte dependente do hardware e a parte independente do hardware. A parte dependente do hardware consiste nas rotinas de tratamento de interrupções e exceções, device drivers, tratamento de sinais, ou seja, todo o código que deve ser reescrito quando se está portando um sistema Unix para uma nova plataforma. A parte independente do hardware não deve ter a princípio nenhum vínculo com a plataforma onde está sendo executada, e é responsável pelo tratamento das system calls, gerência de processos, gerência de memória, escalonamento, pipes, paginação, swapping e sistema de arquivos.

  O kernel do Unix, comparado com o de outros sistemas operacionais, oferece um conjunto relativamente pequeno de system calls, a partir das quais podem ser construídas rotinas de maior complexidade. A estratégia de criar um sistema modular e simples foi muito importante no desenvolvimento do Unix, pois novos utilitários podem ser facilmente integrados ao sistema sem que o kernel tenha que sofrer qualquer tipo de alteração.

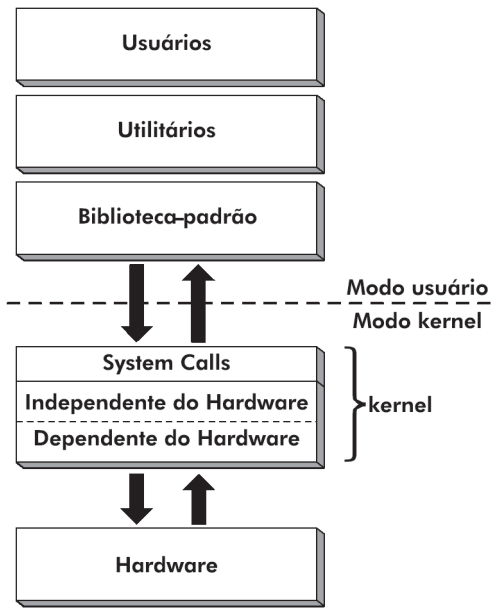


Fig. 15.2 Estrutura do Unix.

- Biblioteca-padrão

  Para cada rotina do sistema existe um procedimento na biblioteca-padrão do Unix que permite esconder os detalhes da mudança de modo de acesso usuário-kernel-usuário. A biblioteca implementa uma interface entre os programas e o sistema operacional, fazendo com que as system calls sejam chamadas. O POSIX define a biblioteca-padrão e não as system calls, ou seja, quais procedimentos a biblioteca deve oferecer, as funções de cada procedimento e os parâmetros de entrada e saída que devem ser utilizados.

- Utilitários

  A camada mais externa do sistema é a interface com o usuário, formada por diversos programas utilitários, como editores de textos, compiladores e o shell. O shell é o interpretador de comandos, responsável por ler os comandos do usuário, verificar se a sintaxe está correta e passar o controle para outros programas que realizam a tarefa solicitada. A Tabela 15.1 apresenta alguns exemplos de comandos e utilitários encontrados na maioria dos sistemas Unix.

Tabela 15.1 Exemplos de comandos e utilitários

Utilitário	Descrição
ls	Lista o conteúdo de diretórios.
cp	Copia arquivos.
mkdir	Cria um diretório.
pwd	Exibe o diretório corrente.
grep	Pesquisa por um determinado padrão dentro do arquivo.
sort	Ordena um arquivo.
cc	Executa o compilador C.
vi	Permite a criação e a edição de arquivos-textos, como programas e scripts.

Devido às várias implementações do Unix, três interpretadores de comandos se tornaram populares. O Bourne Shell (sh) foi o primeiro shell disponível, e pode ser encontrado em todas as versões do Unix. O C Shell (csh) é o padrão para o BSD, e o Korn Shell (ksh) é o padrão para o System V. Além das interfaces orientadas a caractere, existem também interfaces gráficas disponíveis, como o X Windows System.

15.4 Processos e Threads

O Unix, como um sistema multiprogramável, suporta inúmeros processos, que podem ser executados concorrentemente ou simultaneamente. O modelo de processo implementado pelo sistema é muito semelhante ao apresentado no Cap. 5 — Processo. As primeiras versões do Unix não implementavam o conceito de threads, porém as versões mais recentes já oferecem algum tipo de suporte a aplicações multithread.

Um processo é criado através da system call fork. O processo que executa o fork é chamado de processo-pai, enquanto o novo processo é chamado de processo-filho ou subprocesso. Cada processo-filho tem seu próprio espaço de endereçamento individual, independente do processo-pai. Apesar de o espaço de endereçamento dos subprocessos ser independente, todos os arquivos abertos pelo pai são compartilhados com seus filhos. Sempre que um processo é criado, o sistema associa identificadores, que fazem parte do contexto de software, permitindo implementar mecanismos de segurança (Tabela 15.2).

Tabela 15.2 Identificadores

Identificador	Descrição
PID	O Process Identification identifica unicamente um processo para o sistema.
PPID	O Parent Process Identification identifica o processo-pai.
UID	O User Identification identifica o usuário que criou o processo.
GID	O Group Identification identifica o grupo do usuário que criou o processo.

A system call fork, além de criar o subprocesso, copia o espaço de endereçamento do processo-pai para o filho, incluindo o código executável e suas variáveis. Por ser apenas uma cópia, uma alteração no espaço de endereçamento do processo-filho não implica modificação das posições de memória do processo-pai. As versões mais recentes do Unix utilizam a técnica conhecida como copy-on-write para evitar a duplicação de todo o espaço de endereçamento do processo-pai e o tempo gasto na tarefa. Nesse esquema, o espaço de endereçamento do processo-pai é compartilhado com o filho, sendo copiadas apenas as páginas que foram alteradas pelo subprocesso. Uma outra solução, implementada no Unix BSD, é a utilização da system call vfork, que não copia o espaço de endereçamento do processo-pai.

Quando o sistema é ativado o processo 0 é criado, o qual, por sua vez, cria o processo 1. Esse processo, conhecido como init, é o pai de todos os outros processos que venham a ser criados no sistema. Sempre que um usuário inicia uma sessão, o processo init cria um novo processo para a execução do shell. Quando o usuário executa um comando, dois eventos podem ocorrer: o shell pode criar um subprocesso para a execução do comando ou o próprio shell pode executar o comando no seu próprio contexto.

No Unix é possível criar processos foreground e background. No primeiro caso, existe uma comunicação direta do usuário com o processo durante a sua execução. Processos background não podem ter interação com o usuário e são criados com o uso do símbolo &. O comando a seguir mostra a criação de um processo background para a execução de prog.

```
# pgm &
```

Processos do sistema operacional no Unix são chamados de daemons. Os daemons são responsáveis por tarefas administrativas no sistema, como, por exemplo, escalonamento de tarefas (cron), gerência de filas de impressão, suporte a serviços de rede, suporte à gerência de memória (swapper) e gerência de logs. Os daemons são criados automaticamente durante a inicialização do sistema.

Processos no Unix podem se comunicar através de um mecanismo de troca de mensagens, conhecido como pipe. O comando a seguir mostra o mecanismo de pipe entre dois processos. O primeiro processo é criado a partir da execução do comando ls, que lista os arquivos do diretório corrente. A saída desse processo é redirecionada para a entrada do segundo processo, criado para a execução do comando grep. O comando grep seleciona dentre a lista de arquivos as linhas que possuem o string “pgm”.

```
# ls | grep pgm
```

Outro mecanismo de comunicação entre processos muito importante no Unix é conhecido como sinal. Um sinal permite que um processo seja avisado da ocorrência de eventos síncronos ou assíncronos. Por exemplo, quando um programa executa uma divisão

por zero, o sistema avisa ao processo sobre o problema através de um sinal. O processo, por sua vez, pode aceitar o sinal ou simplesmente ignorá-lo. Caso o processo aceite o sinal, é possível especificar uma rotina de tratamento.

Sinais são definidos de diferentes maneiras em cada versão do Unix. O POSIX define um conjunto de sinais padrões que devem ser suportados pelo Unix, a fim de compatibilizar a utilização de sinais. A Tabela 15.3 apresenta alguns dos sinais definidos pelo POSIX.

Tabela 15.3 Sinais POSIX

Sinal	Descrição
SIGALRM	Sinaliza o término de um temporizador.
SIGFPE	Sinaliza um erro em uma operação de ponto flutuante.
SIGILL	Sinaliza que a tecla DEL ou CTRL-C foi digitada para interromper o processo em execução.
SIGKILL	Sinaliza que o processo deve ser eliminado.
SIGTERM	Sinaliza que o processo deve terminar.
SIGSEGV	Sinaliza que o processo fez acesso a um endereço inválido de memória.
SIGUSR1	Pode ser definido pela própria aplicação.
SIGUSR2	Pode ser definido pela própria aplicação.

Um processo no Unix é formado por duas estruturas de dados: a estrutura do processo (proc structure) e a área do usuário (user area ou u area). A estrutura do processo, que contém o seu contexto de software, deve ficar sempre residente na memória principal, enquanto a área do usuário pode ser retirada da memória, sendo necessária apenas quando o processo é executado. A Tabela 15.4 apresenta um resumo das informações contidas nessas duas estruturas.

Tabela 15.4 Processo no Unix

Estrutura do processo	Área do usuário
Identificação: PID, PPID, GID e UID	Registradores
Estado do processo e prioridade	Tabela de descritores de arquivos
Endereço da área do usuário	Contabilidade de recursos do sistema, como UCP, memória e disco
Máscara de sinais	Informações sobre a system call corrente
Ponteiros para as tabelas de páginas	Tratadores de sinais

Os processos existentes no sistema são organizados em um vetor, chamado tabela de processos, onde cada elemento representa uma estrutura do processo. O tamanho desse vetor é predefinido e limita o número máximo de processos no sistema. A estrutura do processo, por sua vez, possui um ponteiro para a área do usuário. Quando um processo executa um fork, o sistema procura por um elemento livre na tabela de processos, onde é criada a estrutura do processo-filho, a partir das informações copiadas da estrutura do processo-pai.

A Tabela 15.5 apresenta algumas system calls voltadas para a gerência de processos disponíveis na maioria dos sistemas Unix.

Tabela 15.5 Gerência de processos

System call	Descrição
fork	Cria um processo-filho idêntico ao processo-pai.
execve execv execl execle	Permite substituir o código executável do processo-filho depois da sua criação.
waitpid	Aguarda até o término do processo-filho.
exit	Termina o processo corrente.
kill	Envia um sinal para um processo.
sigaction	Define qual ação deve ser tomada ao receber um determinado sinal.
alarm	Permite definir um temporizador.

Em função do overhead gerado no mecanismo de criação e eliminação de processos, vários sistemas Unix implementaram o conceito de threads, porém sem qualquer preocupação com compatibilidade (Tabela 15.6). Em 1995, o padrão POSIX P1003.1c, também conhecido como Pthreads, foi aprovado, permitindo que aplicações multithread pudessem ser desenvolvidas de forma padronizada.

Tabela 15.6 Arquitetura de threads

Ambiente	Arquitetura
Compaq Unix 5	Modo híbrido
IBM-AIX 4.2	Modo kernel
HP-UX 10.1	Modo usuário
Linux 2	Modo kernel
Sun Solaris 8	Modo híbrido
SunOS 4	Modo usuário

O POSIX não define como os threads devem ser implementados no sistema, ou seja, o padrão pode ser implementado utilizando pacotes apenas em modo usuário, modo kernel ou uma combinação de ambos (modo híbrido). As vantagens e desvantagens de cada tipo de implementação podem ser consultadas no Cap. 6 — Thread. O padrão POSIX também define mecanismos de sincronização entre threads, como semáforos, mutexes e variáveis condicionais. A Tabela 15.7 apresenta as principais system calls definidas pelos Pthreads.

Tabela 15.7 Gerência de POSIX threads

System call	Descrição
pthread_create	Cria um novo thread.
pthread_exit	Finaliza o thread corrente.
pthread_join	Aguarda pelo término de um thread.
pthread_mutex_init	Cria um mutex.
pthread_mutex_lock	Verifica o estado do mutex.
pthread_mutex_unlock	Libera o mutex.
pthread_mutex_destroy	Elimina o mutex.
pthread_cond_init	Cria uma variável condicional.
pthread_cond_wait	Aguarda por uma variável condicional.
pthread_cond_signal	Libera um thread.
pthread_cond_destroy	Elimina uma variável condicional.

15.5 Gerência do Processador

A gerência do processador no Unix utiliza dois tipos de política de escalonamento: escalonamento circular com prioridades e escalonamento por prioridades. A política de escalonamento tem o objetivo de permitir o compatilhamento da UCP por vários processos interativos e batch, além de oferecer baixos tempos de respostas para os usuários interativos.

Os processos no Unix podem ter prioridades entre 0 e 127, e quanto menor o valor, maior a prioridade. Processos executados no modo usuário têm valor de prioridade entre 50 e 127 (menor prioridade), enquanto processos no modo kernel têm valores de prioridade entre 0 e 49 (maior prioridade). Os processos no estado de pronto ficam aguardando para serem escalonados em diversas filas, cada fila associada a uma prioridade. O algoritmo de escalonamento seleciona para execução o processo de maior prioridade, ou seja, o primeiro processo da fila de menor valor (Fig. 15.3). O processo, depois de escalonado, poderá permanecer no processador no máximo uma fatia de tempo, que varia entre 10 e 100 milissegundos. Depois de terminado seu quantum, o processo retorna para o final da fila associada à sua prioridade.

O escalonador recalcula a prioridade de todos os processos no estado de pronto periodicamente. Para realizar o cálculo da nova prioridade é utilizada a fórmula a seguir, com base em três variáveis: p_cpu, p_nice e base.

$$\text{Prioridade} = \text{p\_cpu} + \text{p\_nice} + \text{base}$$

A variável p_cpu pertence ao contexto de software do processo e permite contabilizar o tempo durante o qual o processo utilizou o processador. Quando o processo é criado, a variável é inicializada com zero. Sempre que o processo é executado, o valor de p_cpu é incrementado de uma unidade a cada tick do clock, até o valor máximo de 127. Quanto maior o valor da variável, menor será sua prioridade. A variável p_cpu permite penalizar os processos CPU-bound e, dessa maneira, distribuir de forma mais igualitária o processador.

O escalonador, além de recalcular a prioridade dos processos, também reduz o valor de p_cpu periodicamente, com base no valor de decay. O cálculo de decay varia con-

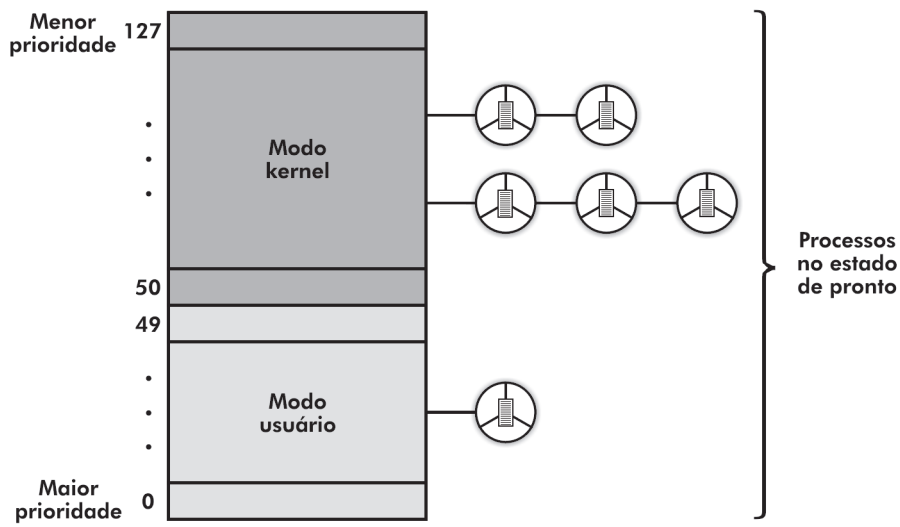


Fig. 15.3 Níveis de prioridade.

forme a versão do Unix e tem a função de evitar o problema de starvation de processos de baixa prioridade. Quanto menos um processo utilizar o processador, menor será o valor de p_cpu e, logo, maior sua prioridade. Esse esquema também privilegia processos I/O-bound, pois mantém suas prioridades elevadas, permitindo que tenham maiores chances de serem executados quando saírem do estado de espera.

No BSD, a variável p_nice pode assumir valores entre 0 e 39, sendo o default 20. Quanto maior o valor atribuído à variável, menor a prioridade do processo. A variável permite alterar a prioridade de um processo de diferentes maneiras. A própria aplicação pode reduzir voluntariamente a sua prioridade a fim de não prejudicar os demais processos utilizando a system call nice. O administrador do sistema pode reduzir o valor da variável utilizando o comando nice, aumentando assim a prioridade de um processo. Processos em background recebem automaticamente um valor maior para p_nice, a fim de não prejudicar os processos interativos.

A variável-base geralmente está associada ao tipo de evento que colocou o processo no estado de espera. Quando a espera termina, é atribuído um baixo valor à variável, fazendo com que o processo receba um aumento de prioridade. Com isso, processos I/O-bound têm suas prioridades aumentadas dependendo do tipo de operação realizada, fazendo com que processos interativos não sejam prejudicados. Por outro lado, processos CPU-bound podem ser executados enquanto os processos I/O-bound estão no estado de espera.

15.6 Gerência de Memória

As primeiras versões do Unix utilizavam basicamente a técnica de swapping para a gerência de memória. Apenas a partir da versão 3BSD o Unix passou a utilizar paginação por demanda. Atualmente, a grande maioria das versões do Unix, tanto BSD como System V, implementa gerência de memória virtual por paginação com swapping. Neste item será apresentada uma abordagem com base na versão 4BSD, mas grande parte dos conceitos e mecanismos apresentados também pode ser encontrada no System V.

No Unix, os conceitos de espaço de endereçamento virtual e mapeamento seguem as mesmas definições apresentadas no Cap. 10 — Gerência de Memória Virtual. Muitos detalhes de implementação, como tamanho de página, níveis de tabelas de páginas e TLBs, são dependentes da arquitetura de hardware, podendo variar conforme a versão de cada sistema.

O espaço de endereçamento dos processos no Unix é dividido em três segmentos: texto, dados e pilha (Fig. 15.4). O segmento de texto corresponde à área onde está o código executável dos programas, sendo uma área protegida contra gravação. O segmento de texto é estático e pode ser compartilhado por vários processos, utilizando o esquema de memória compartilhada. O segmento de dados corresponde às variáveis do programa, como tipos numéricos, vetores e strings. A área de dados é dinâmica, podendo aumentar ou diminuir durante a execução do programa. A pilha armazena informações de controle do ambiente do processo, como parâmetros passados a um procedimento ou system call. A área de pilha cresce dinamicamente do endereço virtual mais alto para o mais baixo.

O Unix implementa o esquema de paginação por demanda como política de busca de páginas. Nesse esquema, páginas do processo são trazidas do disco para a memória principal apenas quando são referenciadas. O sistema mantém uma lista de páginas livres com todos os frames disponíveis na memória, e gerencia os frames de todos os pro-

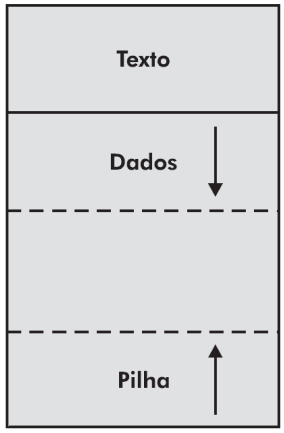


Fig. 15.4 Espaço de endereçamento.

cessos em uma lista de páginas em uso. Quando um processo faz referência a uma página que não se encontra na lista de páginas em uso, ocorre um page fault. O sistema identifica se a página está na memória através do bit de validade. Nesse caso, a gerência de memória retira uma página da lista de páginas livres e transfere para a lista de páginas em uso. Como pode ser observado, o Unix utiliza uma política global de substituição de páginas.

O mecanismo de paginação é implementado parte pelo kernel e parte pelo daemon page (processo 2). Periodicamente, o daemon é ativado para verificar o número de páginas livres na memória. Se o número de páginas for insuficiente, o daemon page inicia o trabalho de liberação de páginas dos processos para recompor a lista de páginas livres. As páginas de texto podem ser transferidas sem problemas para a lista de páginas livres, pois podem ser recuperadas no arquivo executável. Por outro lado, para as páginas de dados o sistema utiliza o bit de modificação para verificar se a página foi modificada. Nesse caso, antes de ser liberada para a lista de páginas livres a página é gravada em disco.

O sistema implementa uma variação do algoritmo FIFO circular como política de substituição de páginas. Esse algoritmo, conhecido como two-handed clock, utiliza dois ponteiros, ao contrário do FIFO circular, que implementa apenas um. O primeiro ponteiro fica à frente do segundo um certo número de frames na lista de páginas em uso. Enquanto o primeiro ponteiro desliga o bit de referência das páginas, o segundo verifica o seu estado. Se o bit de referência continuar desligado, significa que a página não foi referenciada desde o momento em que o primeiro ponteiro desligou o bit, fazendo com que essa página seja selecionada. Apesar de estarem liberadas para outros processos, as páginas selecionadas permanecem um certo tempo intactas na lista de páginas livres. Dessa forma, é possível que as páginas retornem aos processos de onde foram retiradas, eliminando-se a necessidade de acesso a disco. O daemon page também é responsável por implementar a política de substituição de páginas.

Em casos em que o sistema não consegue manter um número suficiente de páginas livres, o mecanismo de swapping é ativado. Nesse caso, o daemon swapper seleciona, inicialmente, os processos que estão há mais tempo inativos. Em seguida, o swapper seleciona dentre os quatro processos que mais consomem memória principal aquele que estiver mais tempo inativo. Esse mecanismo é repetido até que a lista de páginas livres retorne ao seu tamanho normal. Para cada processo transferido para disco é atribuída uma prioridade, calculada em função de vários parâmetros. Em geral, o processo que está mais tempo em disco é selecionado para retornar à memória principal.

Para controlar todas as páginas e listas na memória principal, a gerência de memória mantém uma estrutura de mapeamento dos frames na memória (core map), com informações sobre todas as páginas livres e em uso. O core map fica residente na parte não-paginável da memória principal, juntamente com o kernel do sistema.

15.7 Sistema de Arquivos

O sistema de arquivos foi o primeiro componente a ser desenvolvido no Unix, e as primeiras versões comerciais utilizavam o System V File System (S5FS). Devido às suas limitações, Berkeley desenvolveu o Fast File System (FFS) introduzido no 4.2BSD. Posteriormente, o SVR4 passou também a suportar o sistema de arquivos de Berkeley.

Um arquivo no Unix é simplesmente uma seqüência de bytes sem significado para o sistema operacional, que desconhece se o conteúdo do arquivo representa um texto ou um programa executável. O sistema tem apenas a função de prover o acesso seqüencial ou aleatório ao arquivo, ficando a cargo da aplicação a organização e outros métodos de acesso. O tamanho do nome de arquivos era, inicialmente, limitado a 14 caracteres, mas as versões mais recentes ampliam esse nome para 255 caracteres.

O sistema de arquivos do Unix tem como base uma estrutura de diretórios hierárquica, sendo o diretório raiz (root) representado pela barra (/). Os diretórios são implementados através de arquivos comuns, responsáveis pela manutenção da estrutura hierárquica do sistema de arquivos. Todo diretório contém os nomes de arquivos ponto (.) e dois pontos (..), que correspondem, respectivamente, ao próprio diretório e ao seu diretório-pai (Fig. 15.5).

Alguns nomes de diretório do sistema de arquivos são padronizados, como o diretório de programas executáveis do sistema (/bin), o diretório de arquivos especiais ligados aos dispositivos de E/S (/dev), o diretório de bibliotecas (/lib) e o diretório que agrupa os subdiretórios dos usuários (/usr). Geralmente, cada usuário possui seu diretório default de login, denominado diretório home ou de trabalho (/maia e /machado).

A localização de um arquivo dentro da estrutura de diretórios é indicada utilizando-se um pathname, que representa uma seqüência de diretórios separados por barra. Existem dois tipos de pathname: absoluto e relativo. Um pathname absoluto indica a localização de um arquivo através de uma seqüência completa de diretórios e subdiretórios, a partir da raiz do sistema de arquivos. Um pathname relativo indica a localização de um

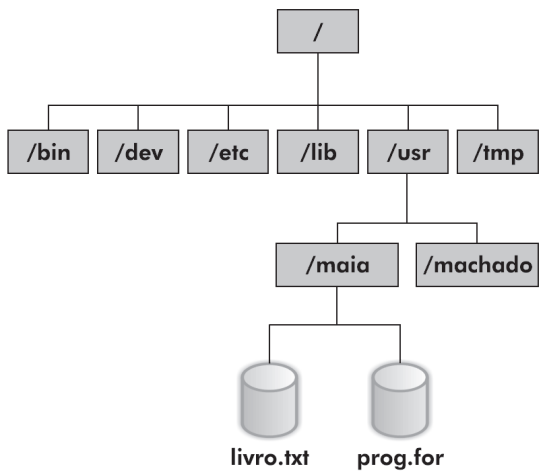


Fig. 15.5 Estrutura de diretórios.

arquivo a partir do diretório corrente. Por exemplo, na Fig. 15.5 o pathname /usr/maia/livro.txt representa o caminho absoluto para o arquivo livro.txt. Caso um usuário esteja posicionado no diretório /usr/machado, basta especificar o caminho relativo ../maia/livro.txt para ter acesso ao mesmo arquivo.

O sistema de arquivos proporciona um mecanismo para compartilhamento de arquivos conhecido como link simbólico. Um link é uma entrada em um diretório que faz referência a um arquivo em um outro diretório. Um arquivo pode possuir múltiplos links, de forma que diferentes nomes de arquivos podem ser utilizados por diversos usuários no acesso às informações de um único arquivo. O número de links de um arquivo é o número de diferentes nomes que ele possui. Existem diversas vantagens na utilização de links, como redução de utilização de espaço em disco, uma vez que existe uma única cópia dos dados, compartilhamento entre diversos usuários da última versão do arquivo e facilidade de acessso a arquivos em outros diretórios.

Cada arquivo no Unix pertence a uma ou mais dentre três categorias de usuários. Todo arquivo ou diretório tem um dono (user) e pertence a um grupo (group). Qualquer usuário que não seja o dono do arquivo e não pertença ao grupo enquadra-se na categoria outros (others). O administrador do sistema, utilizando a conta root, não pertence a nenhuma dessas categorias, tendo acesso irrestrito a todos os arquivos. Para cada categoria de usuário podem ser concedidos três tipos de acesso: leitura (r), gravação (w) e execução (x). A Tabela 15.8 apresenta as permissões que podem ser aplicadas a arquivos e diretórios.

Tabela 15.8 Permissões para arquivos e diretórios

Arquivo	**Descrição**
r	Permissão para ler e copiar o arquivo.
w	Permissão para alterar e eliminar o arquivo.
x	Permissão para executar o arquivo.
Diretório	**Descrição**
r	Permissão para listar o conteúdo do diretório.
w	Permissão para criar, eliminar e renomear arquivos no diretório.
x	Permissão para que o usuário possa se posicionar no diretório e acessar os arquivos abaixo desse diretório.

A Tabela 15.9 apresenta algumas system calls relacionadas à gerência do sistema de arquivos, envolvendo operações com arquivos e diretórios.

Tabela 15.9 System calls do sistema de arquivos

System call	**Descrição**
creat	Cria um arquivo.
open	Abre um arquivo.
read	Lê um dado do arquivo para o buffer.
write	Grava um dado do buffer no arquivo.
position	Posiciona o ponteiro do arquivo.
close	Fecha um arquivo.
mkdir	Cria um diretório.
chdir	Altera o diretório default.
rmdir	Elimina um diretório.
link	Cria um link simbólico para um arquivo.
unlink	Elimina um link simbólico para um arquivo.

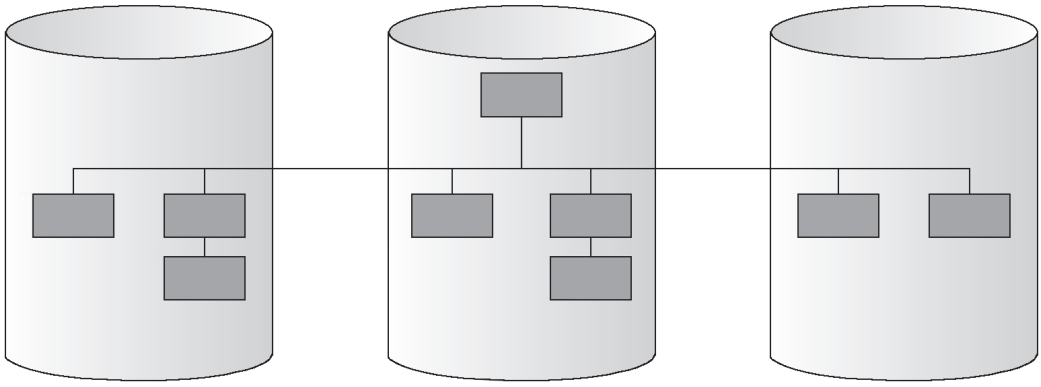
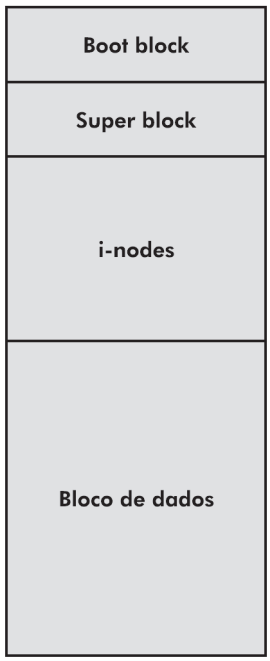


Fig. 15.6 Sistema de arquivos.

No Unix não existe uma dependência entre a estrutura lógica do sistema de arquivos e o local onde os arquivos estão fisicamente armazenados (Fig. 15.6). Dessa forma, é possível criar um sistema de arquivos onde os diretórios e arquivos estão fisicamente distribuídos em vários discos, porém para o usuário é como se existisse uma única estrutura lógica de diretórios. Esse modelo permite adicionar novos discos ao sistema de arquivos sempre que necessário, sem alterar sua estrutura lógica. Além disso, os diversos discos podem estar residentes em estações remotas. Nesse caso, existem padrões para a implementação de sistemas de arquivos remotos, como Network File System (NFS), Remote File System (RFS) e Andrew File System (AFS).

A estrutura do sistema de arquivos do Unix varia conforme a implementação. Em geral, qualquer disco deve ter a estrutura semelhante à descrita na Fig. 15.7. O boot block, quando utilizado, serve para realizar a carga do sistema. O super block possui informações sobre a estrutura do sistema de arquivos, incluindo o número de i-nodes, o número de blocos do disco e o início da lista de blocos livres. Qualquer problema com o super block tornará inacessível o sistema de arquivos.

Fig. 15.7 Estrutura do sistema de arquivos.



Os i-nodes (index-nodes) permitem identificar e mapear os arquivos no disco. Cada i-node possui 64 bytes e descreve um único arquivo contendo seus atributos, como tamanho, datas de criação e modificação, seu dono, grupo, proteção, permissões de acesso, tipo do arquivo, além da localização dos blocos de dados no disco. Os blocos de dados contêm os dados propriamente ditos, ou seja, arquivos e diretórios. Um arquivo pode ser formado por um ou mais blocos, contíguos ou não no disco.

No caso de arquivos pequenos, todos os blocos que compõem o arquivo podem ser mapeados diretamente pelo i-node. No caso de arquivos grandes, o número de entradas para endereçamento no i-node não é suficiente para mapear todos os blocos do arquivo. Nesse caso, utilizam-se os redirecionamentos único, duplo e triplo. No redirecionamento único uma das entradas no i-node aponta para uma outra estrutura, que por sua vez endereça os blocos do arquivo no disco. Os redirecionamentos duplo e triplo são apenas uma extensão do conceito apresentado (Fig. 15.8).

15.8 Gerência de Entrada/Saída

A gerência de entrada/saída no Unix foi desenvolvida de forma integrada ao sistema de arquivos. O acesso aos dispositivos de E/S, como terminais, discos, impressoras e à própria rede, é feito através de arquivos especiais. Cada dispositivo está associado a um ou mais arquivos especiais, localizados no diretório /dev. Por exemplo, uma impressora pode ser o arquivo /dev/1p; um terminal, /dev/tty1; e uma interface de rede, /dev/net.

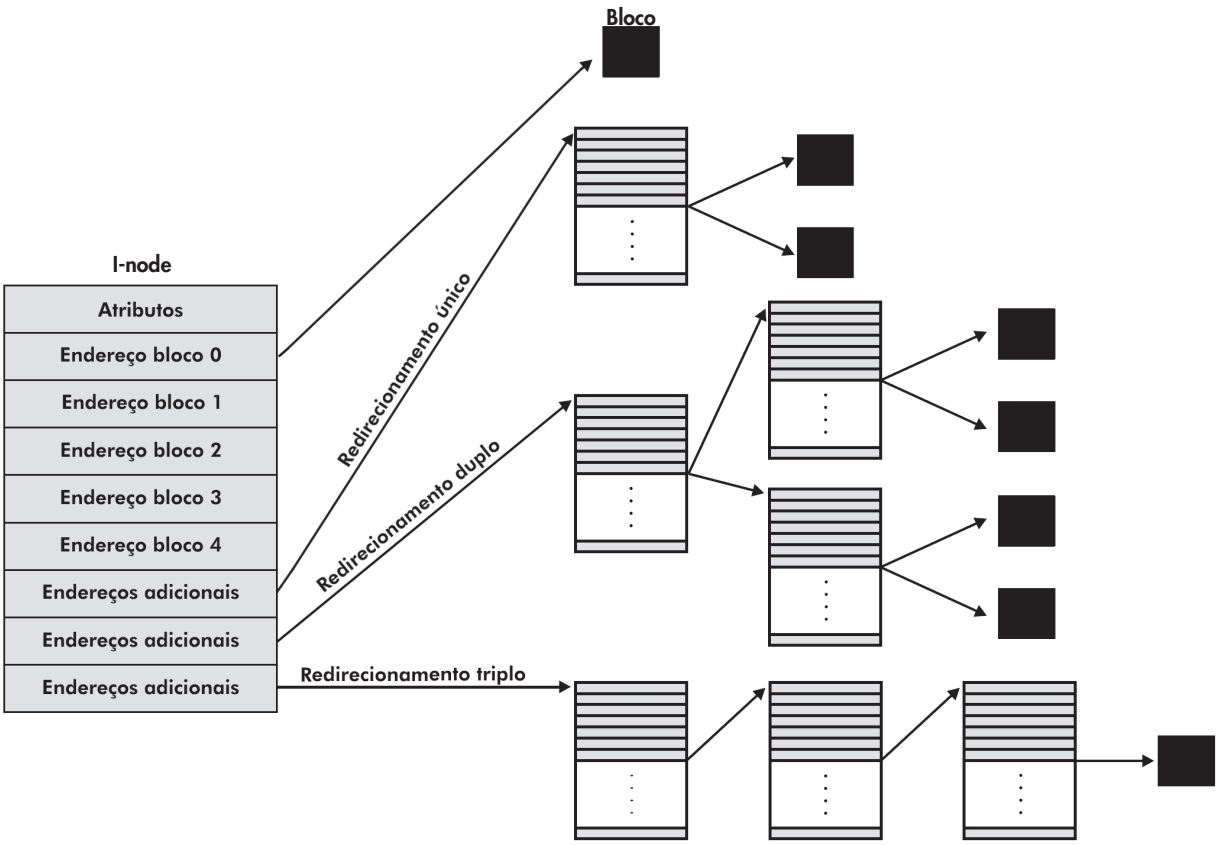
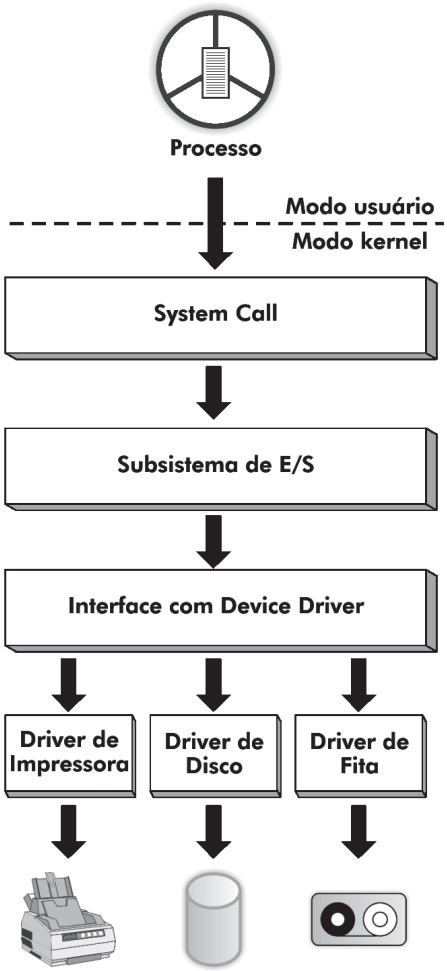


Fig. 15.8 Estrutura do i-node.

Fig. 15.9 Gerência de E/S.



Os arquivos especiais podem ser acessados da mesma forma que qualquer outro arquivo, utilizando simplesmente as system calls de leitura e gravação. No Unix, todas as operações de E/S são realizadas como uma seqüência de bytes, não existindo o conceito de registro ou método de acesso. Isso permite enviar o mesmo dado para diferentes dispositivos de saída, como um arquivo em disco, terminal, impressora ou linha de comunicação. Dessa forma, as system calls de E/S podem manipular qualquer tipo de dispositivo de maneira uniforme.

A Fig. 15.9 apresenta as camadas que compõem a gerência de E/S no Unix. Os processos se comunicam com o subsistema de E/S através das system calls de E/S. O subsistema de E/S é a parte do kernel responsável por lidar com as funções entrada e saída independentes do dispositivo, como buffering e controle de acesso. Para permitir a comunicação entre o subsistema de E/S e os diferentes drivers de maneira uniforme, o sistema implementa uma interface com os device drivers, padronizada com base nas especificações Device Driver Interface (DDI) e Driver Kernel Interface (DKI).

Os device drivers têm a função de isolar os dispositivos de E/S do restante do kernel, tornando-o independente da arquitetura de hardware, e para cada dispositivo existe um device driver associado. Os device drivers são acoplados ao sistema operacional quando o kernel é gerado, e sempre que um novo dispositivo é acrescentado ao sistema o driver correspondente deve ser acoplado ao núcleo. A tarefa de geração do kernel não

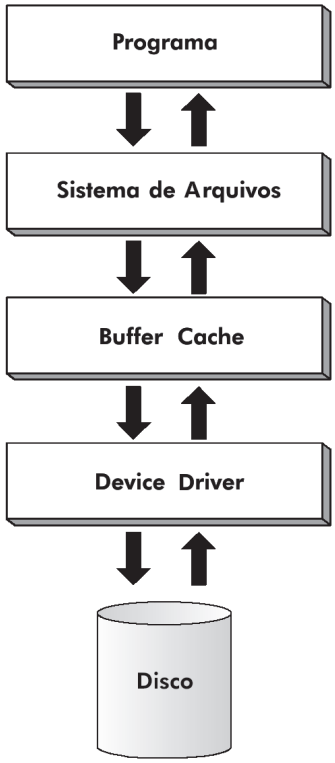


Fig. 15.10 Operação orientada a bloco.

é simples, e exige que o sistema seja reinicializado. As versões mais recentes do Unix, como o Linux, permitem que os device drivers possam ser acoplados ao núcleo com o sistema em funcionamento, sem a necessidade de uma nova geração do kernel e reinicialização do sistema.

Os device drivers podem ser divididos em dois tipos: orientados a bloco e orientados a caractere. Os device drivers orientados a bloco estão ligados a dispositivos como discos e CD-ROMs, que permitem a transferência de blocos de informações do mesmo tamanho. Os drivers orientados a caractere são voltados para atender dispositivos como terminais e impressoras que transferem informação de tamanho variável, geralmente caractere a caractere ou uma seqüência de caracteres.

No caso das operações orientadas a bloco, deve existir a preocupação em minimizar o número de transferências entre o dispositivo e a memória, utilizando o buffer cache (Fig. 15.10). O buffer cache é uma área na memória principal onde ficam armazenados temporariamente os blocos recentemente referenciados. Por exemplo, quando uma operação de leitura a disco é realizada o subsistema de E/S verifica se o bloco está no buffer cache. Se o bloco se encontra no cache, é possível passá-lo diretamente para o sistema de arquivos, sem acesso ao disco, melhorando assim o desempenho do sistema.