Universidade Estadual de Campinas - UNICAMP
Faculdade de Eng. Elétrica e de Computação

Introdução aos Sistemas Operacionais

Eleri Cardozo
FEEC - UNICAMP

Maurício F. Magalhães
FEEC - UNICAMP

Luís F. Faina
Faculdade de Computação
Universidade Federal de Uberlândia

Janeiro de 2002
Campinas, SP - Brasil

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Programas computacionais (ou software) constituem o elo entre o aparato eletrônico (ou hardware) e o ser humano. Tal elo se faz necessário dada a discrepância entre o tipo de informação manipulada pelo homem e pela máquina. A máquina opera com cadeias de códigos binários enquanto o homem opera com estruturas mais abstratas como conjuntos, arquivos, algoritmos, etc [1].

Programas computacionais podem ser classificados em dois grandes grupos:

- software de sistema, que manipulam a operação do computador;
- programas aplicativos, que resolvem problemas para o usuário.

O mais importante dos softwares de sistema é o sistema operacional, que controla todos os recursos do computador e proporciona a base de sustentação para a execução de programas aplicativos.

1.1 O que é um Sistema Operacional ?

A maioria de usuários de computador têm alguma experiência com sistemas operacionais, mas é difícil definir precisamente o que é um sistema operacional. Parte do problema decorre do fato do sistema operacional realizar duas funções básicas que, dependendo do ponto de vista abordado, uma se destaca sobre a outra. Estas funções são descritas a seguir.

O Sistema Operacional como uma Máquina Virtual

A arquitetura[1] da maioria dos computadores no nível da linguagem de máquina é primitiva e difícil de programar, especificamente para operações de entrada e saída. É preferível para um programador trabalhar com abstrações de mais alto nível onde detalhes de implementação das abstrações não são visíveis. No caso de discos, por exemplo, uma abstração típica é que estes armazenam uma coleção de arquivos identificados por nomes simbólicos.

O programa que esconde os detalhes de implementação das abstrações é o sistema operacional. A abstração apresentada ao usuário pelo sistema operacional é simples e mais fácil de usar que o hardware original.

Nesta visão, a função do sistema operacional é apresentada ao usuário como uma *máquina estendida* ou *máquina virtual* que é mais fácil de programar que o hardware que a suporta.

[1] Conjunto de instruções, organização de memória, entrada/saída (E/S), estrutura de barramento, etc.

O Sistema Operacional como um Gerenciador de Recursos

Um computador moderno é composto de vários subsistemas tais como processadores, memorias, discos, terminais, fitas magnéticas, interfaces de rede, impressoras, e outros dispositivos de E/S. Neste ponto de vista, o sistema operacional tem a função de gerenciar de forma adequada estes recursos de sorte que as tarefas impostas pelos usuários sejam atendidas da forma mais rápida e confiável possível. Um exemplo típico é o compartilhamento da unidade central de processamento (CPU) entre as várias tarefas (programas) em sistemas multiprogramados. O sistema operacional é o responsável pela distribuição de forma otimizada da CPU entre as tarefas em execução.

1.2 História dos Sistemas Operacionais

Os sistemas operacionais têm evoluído com o passar dos anos. Nas próximas seções vamos apresentar de forma sucinta este desenvolvimento.

A Primeira Geração (1945-1955): Válvulas e Plugs

Após muitos esforços mal sucedidos de se construir computadores digitais antes da 2ª guerra mundial, em torno da metade da década de 1940 alguns sucessos foram obtidos na construção de máquinas de cálculo empregando-se válvulas e relés. Estas máquinas eram enormes, ocupando salas com *racks* que abrigavam dezenas de milhares de válvulas (e consumiam quantidades imensas de energia).

Naquela época, um pequeno grupo de pessoas projetava, construía, programava, operava e mantinha cada máquina. Toda programação era feita absolutamente em linguagem de máquina, muitas vezes interligando *plugs* para controlar funções básicas da máquina. Linguagens de programação eram desconhecidas; sistemas operacionais idem. Por volta de 1950 foram introduzidos os cartões perfurados aumentando a facilidade de programação.

A Segunda Geração (1955-1965): Transistores e Processamento em Batch

A introdução do transistor mudou radicalmente o quadro. Computadores tornaram-se confiáveis e difundidos (com a fabricação em série), sendo empregados em atividades múltiplas. Pela primeira vez, houve uma separação clara entre projetistas, construtores, operadores, programadores e pessoal de manutenção. Entretanto, dado seu custo ainda elevado, somente corporações e universidades de porte detinham recursos e infra-estrutura para empregar os computadores desta geração.

Estas máquinas eram acondicionadas em salas especiais com pessoal especializado para sua operação. Para executar um *job* (programa), o programador produzia um conjunto de cartões perfurados (um cartão por comando do programa), e o entregava ao operador que dava entrada do programa no computador. Quando o computador completava o trabalho, o operador devolvia os cartões com a impressão dos resultados ao programador.

A maioria dos computadores de 2ª geração foram utilizados para cálculos científicos e de engenharia. Estes sistemas eram largamente programados em *FORTRAN* e *ASSEMBLY*. Sistemas operacionais típicos[2] eram o FMS *(Fortran Monitor Systems)* e o IBSYS *(IBM's Operating Systems)*.

[2]Que eram capazes de gerenciar apenas um *job* por vez

A Terceira Geração (1965-1980): Circuitos Integrados e Multiprogramação

No início dos anos 60, a maioria dos fabricantes de computadores mantinham duas linhas distintas e incompatíveis de produtos. De um lado, havia os computadores científicos que eram usados para cálculos numéricos nas ciências e na engenharia. Do outro, haviam os computadores comerciais que executavam tarefas como ordenação de dados e impressão de relatórios, sendo utilizados principalmente por instituições financeiras.

A IBM tentou resolver este problema introduzindo a série *System/360*. Esta série consistia de máquinas com mesma arquitetura e conjunto de instruções. Desta maneira, programas escritos para uma máquina da série executavam em todas as demais. A série 360 foi projetada para atender tanto aplicações científicas quanto comerciais.

Não foi possível para a IBM escrever um sistema operacional que atendesse a todos os conflitos de requisitos dos usuários. O resultado foi um sistema operacional (OS/360) enorme e complexo comparado com o FMS.

A despeito do tamanho e problemas, o OS/360 atendia relativamente bem às necessidades dos usuários. Ele também popularizou muitas técnicas ausentes nos sistemas operacionais de 2ª geração, como por exemplo a multiprogramação. Outra característica apresentada foi a capacidade de ler *jobs* dos cartões perfurados para os discos, assim que o programador os entregasse. Dessa maneira, assim que um *job* terminasse, o computador iniciava a execução do seguinte, que já fôra lido e armazenado em disco. Esta técnica foi chamada *spool (simultaneous peripherical operation on line)*, sendo também utilizada para a saída de dados.

O tempo de espera dos resultados dos programas reduziu-se drasticamente com a 3ª geração de sistemas. O desejo por respostas rápidas abriu caminho para o *time-sharing*, uma variação da multiprogramação onde cada usuário tem um terminal *on-line* e todos compartilham uma única CPU.

Após o sucesso do primeiro sistema operacional com capacidade de *time-sharing* (o CTSS) desenvolvido no MIT, um consórcio envolvendo o MIT, a GE e o Laboratório Bell foi formado com o intuito de desenvolver um projeto ambicioso para a época: um sistema operacional que suportasse centenas de usuários *on-line*. O MULTICS *(MULTiplexed Information and Computing Service)* introduziu muitas idéias inovadoras, mas sua implementação mostrou-se impraticável para a década de sessenta. O projeto MULTICS influenciou os pesquisadores da Bell que viriam a desenvolver o UNIX uma década depois.

A Quarta Geração (1980-): Computadores Pessoais e Estações de Trabalho

Com o desenvolvimento de circuitos integrados em larga escala (LSI), *chips* contendo milhares de transistores em um centímetro quadrado de silício, surgiu a era dos computadores pessoais e estações de trabalho. Em termos de arquitetura, estes não diferem dos minicomputadores da classe do PDP-11, exceto no quesito mais importante: preço. Enquanto os minicomputadores atendiam companhias e universidades, os computadores pessoais e estações de trabalho passaram a atender usuários individualmente.

O aumento do potencial destas máquinas criou um vastíssimo mercado de software a elas dirigido. Como requisito básico, estes produtos (tanto aplicativos quanto o próprio sistema operacional) necessitavam ser "amigáveis", visando usuários sem conhecimento aprofundado de computadores e sem intenção de estudar muito para utilizá-los. Esta foi certamente a maior mudança em relação ao OS/360 que era tão obscuro que diversos livros foram escritos sobre ele. Dois sistemas operacionais tem dominado o mercado: MS-DOS (seguido do MS-Windows) para os computadores pessoais e UNIX (com suas várias vertentes) para as estações de trabalho.

O próximo desenvolvimento no campo dos sistemas operacionais surgiu com a tecnologia de redes de computadores: os sistemas operacionais de rede e distribuídos.

Sistemas operacionais de rede diferem dos sistemas operacionais para um simples processador no tocante à capacidade de manipular recursos distribuídos pelos processadores da rede. Por exemplo, um arquivo pode ser acessado por um usuário em um processador, mesmo que fisicamente o arquivo se encontre em outro processador. Sistemas operacionais de rede provêem ao usuário uma interface transparente de acesso a recursos compartilhados (aplicativos, arquivos, impressoras, etc), sejam estes recursos locais ou remotos.

Sistemas operacionais distribuídos são muito mais complexos. Estes sistemas permitem que os processadores cooperem em serviços intrínsecos de sistemas operacionais tais como escalonamento de tarefas e paginação. Por exemplo, em um sistema operacional distribuído uma tarefa pode "migrar" durante sua execução de um computador sobrecarregado para outro que apresente carga mais leve. Contrário aos sistemas operacionais de rede que são largamente disponíveis comercialmente, sistemas operacionais distribuídos têm sua utilização ainda restrita.

1.3 Conceitos Básicos em Sistemas Operacionais

Processos

Um conceito fundamental em sistemas operacionais é o de *processo* ou *tarefa*. Um processo (às vezes chamado de processo sequencial) é basicamente um programa em execução, sendo uma entidade ativa que compete por recursos (principalmente CPU) e interage com outros processos.

Em um instante qualquer, um processo está em um determinado estado. Estes estados podem ser:

- executando (usando a CPU para executar as instruções do programa);
- bloqueado (aguardando recursos, que não CPU, indisponíveis no momento);
- ativo (aguardando apenas CPU para executar).

Um processo em execução passa por um sequência de estados ordenados no tempo. Um processo possui duas importantes propriedades:

- o resultado da execução de um processo independe da velocidade com que é executado;
- se um processo for executado novamente com os mesmos dados, ele passará precisamente pela mesma sequência de instruções e fornecerá o mesmo resultado.

Estas propriedades enfatizam a natureza sequencial e determinística de um processo. O processo sequencial é definido pelo resultado de suas instruções, não pela velocidade com que as instruções são executadas.

Sistemas Multitarefas e Multiusuários

Como já mencionado, um programa em execução é chamado de processo ou tarefa. Um sistema operacional multitarefa se distingue pela sua habilidade de suportar a execução concorrente de processos sobre um processador único, sem necessariamente prover elaborada forma de gerenciamento de recursos (CPU, memória, etc).

Sistemas operacionais multiusuários permitem acessos simultâneos ao computador através de dois ou mais terminais de entrada. Embora frequentemente associada com multiprogramação, multitarefa não implica necessariamente em uma operação multiusuário. Operação multiprocessos sem suporte de multiusuários pode ser encontrado em sistemas operacionais de alguns computadores pessoais (por exemplo, Windows'98) avançados e em sistemas de tempo-real.

Multiprogramação

Multiprogramação é um conceito mais geral que multitarefa e denota um sistema operacional que provê gerenciamento da totalidade de recursos tais como CPU, memória, sistema de arquivos, em adição ao suporte da execução concorrente dos processos.

Em uma máquina podemos ter o conjunto de processos sendo executados de forma serial ou de forma concorrente, ou seja, os recursos presentes na máquina podem ser alocados a um único programa até a conclusão de sua execução ou esses recursos podem ser alocados de modo dinâmico entre um número de programas ativos de acordo com o nível de prioridade ou o estágio de execução de cada um dos programas.

No caso de um computador no qual o sistema operacional utilizado permite apenas a monoprogramação, os programas serão executados instrução-a-instrução, até que seu processamento seja concluído. Durante a sua execução, o programa passará por diversas fases, alterando momentos em que se encontra executando ou bloqueado aguardando, por exemplo, a conclusão de uma operação de entrada/saída de dados (normalmente lenta, se comparada à velocidade de execução das instruções por parte do processador).

Através do uso da multiprogramação é possível reduzir os períodos de inatividade da CPU e consequentemente aumentar a eficiência do uso do sistema como um todo. O termo multiprogramação denota um sistema operacional o qual em adição ao suporte de múltiplos processos concorrentes, permite que instruções e dados de dois ou mais processos disjuntos estejam residentes na memória principal simultaneamente.

O nível de multiprogramação presente em um sistema pode ser classificado como integral ou serial. A multiprogramação é denominada integral caso mais de um processo possa se encontrar em execução em um dado instante, enquanto que no caso da serial apenas um processo se encontra em execução a cada instante, sendo a CPU alocada aos processos de forma intercalada ao longo do tempo. Uma vez que a maioria dos computadores apresenta apenas uma única CPU, a multiprogramação serial é encontrada com mais frequência.

Multiprocessamento

Embora a maioria dos computadores disponha de uma única CPU que executa instruções uma a uma, certos projetos mais avançados incrementaram a velocidade efetiva de computação permitindo que várias instruções sejam executadas ao mesmo tempo. Um computador com múltiplos processadores que compartilhem uma memória principal comum é chamado um multiprocessador. O sistema que suporta tal configuração é um sistema que suporta o multiprocessamento.

Interpretador de Comandos (Shell)

O interpretador de comando é um processo que perfaz a interface do usuário com o sistema operacional. Este processo lê o teclado a espera de comandos, interpreta-os e passa seus parâ-

metros ao sistema operacional. Serviços como login/logout, manipulação de arquivos, execução de programas, etc, são solicitados através do interpretador de comandos.

Chamadas de Sistema (System Calls)

Assim como o interpretador de comandos é a interface usuário/sistema operacional, as chamadas do sistema constituem a interface programas aplicativos/sistema operacional. As chamadas do sistema são funções que podem ser ligadas com os aplicativos provendo serviços como: leitura do relógio interno, operações de entrada/saída, comunicação inter-processos, etc.

1.4 O Sistema Operacional UNIX

Dada sua larga aceitação, o sistema operacional UNIX será utilizado como referência neste curso. O sistema UNIX (Fig. 1.1) é dividido em duas partes:

- programas e serviços: shell, mail, vi, date, etc;
- núcleo *(kernel)*: provê suporte aos programas e serviços.

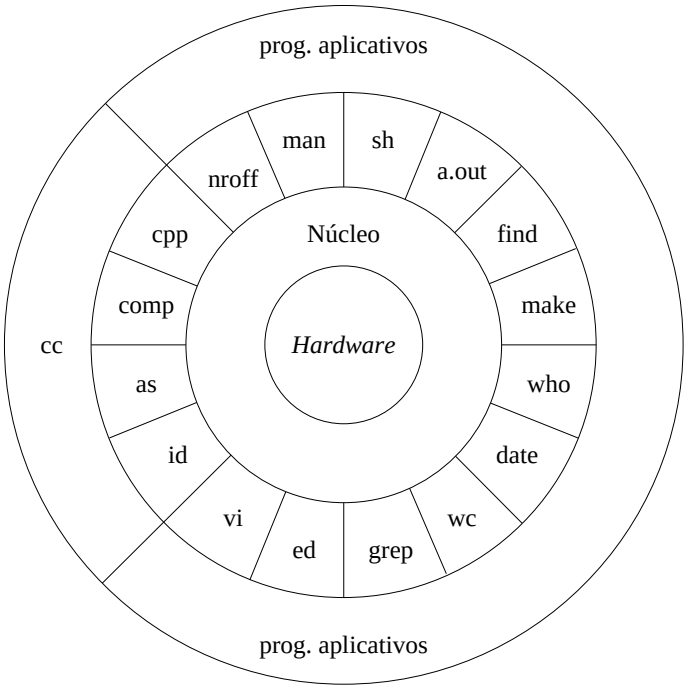

Fig. 1.1: Arquitetura do sistema operacional UNIX

Desenvolvido nos Laboratórios da Bell em meados da década de setenta, o sistema UNIX inicialmente atendia as necessidades específicas do grupo de Ciência da Computação da Bell. A razão da aceitação do UNIX é explicada pelos atributos abaixo:

- escrito em linguagem de alto nível, o que facilita sua portabilidade;
- interface simples para com o usuário *(shell)*;

- fornece primitivas que permitem o desenvolvimento de programas complexos a partir de programas mais simples;
- estrutura hierárquica de arquivos;
- formatação de arquivos baseada no conceito de *stream* (cadeia) de bytes;
- interfaceamento simples e consistente com os dispositivos periféricos;
- multiusuário/multiprogramado;
- esconde a arquitetura do hardware, permitindo que um programa execute em múltiplas plataformas.

Programas interagem com o núcleo do sistema operacional através da evocação de um conjunto bem definido de chamadas de sistema. O conjunto de chamadas do sistema e os algoritmos internos que implementam estas chamadas formam o corpo do núcleo. A organização do ambiente UNIX pode ser vista na forma de camadas conforme mostrado na Fig. 1.1.

O Sistema de Arquivos

O sistema de arquivos do UNIX é caracterizado por [2, 3]:

- estrutura hierárquica;
- tratamento consistente dos dados de arquivo;
- facilidade na criação/eliminação de arquivos;
- crescimento dinâmico de arquivos;
- proteção aos dados dos arquivos;
- tratamento dos dispositivos periféricos como arquivos de dados.

O sistema de arquivo é organizado na forma de árvore conforme pode ser visto no exemplo da Fig. 1.2.

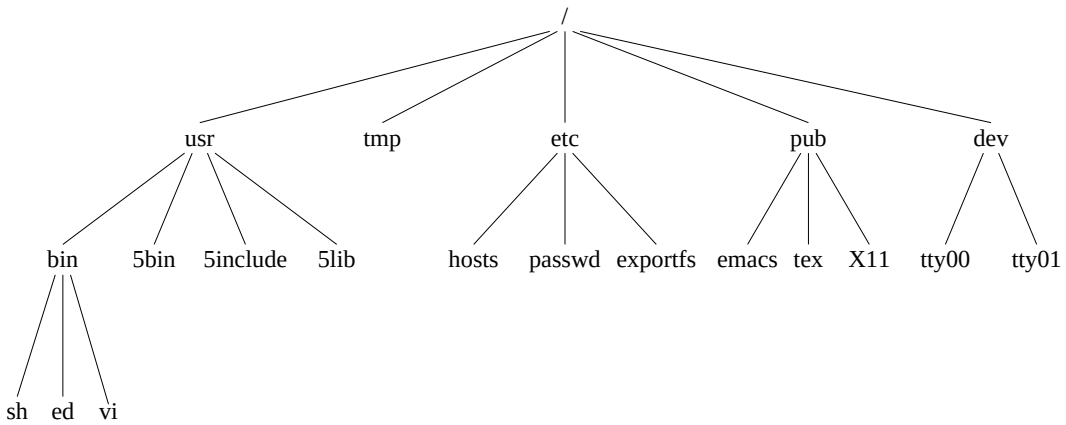

Fig. 1.2: Organização hierárquica do sistema de arquivos

Ainda com relação a esta Fig. 1.2 temos:

- /: diretório raiz;
- não-folhas: diretórios de arquivos;
- folhas: diretórios ou arquivos regulares ou arquivos especiais de dispositivos.

A localização de um arquivo na hierarquia pode ser na forma absoluta ou relativa. Na forma absoluta utiliza-se o caracter / no início do endereço para indicar a raiz, enquanto no caso relativo inicia-se o caminho com o nome do arquivo que tem o diretório atual como o ponto de partida do endereço.

Os programas no ambiente UNIX não possuem nenhum conhecimento sobre o formato interno no qual o núcleo armazena os dados de arquivo. Os dados são fornecidos pelo UNIX como um *stream* (cadeia) de bytes, cabendo aos programas interpretarem o seu conteúdo. Este tratamento estende-se também aos diretórios, ou seja, estes são vistos pelo sistema operacional como arquivos regulares.

O acesso aos arquivos é controlado pelas permissões de acesso associadas a cada arquivo. No caso, temos os seguintes tipos de permissões: leitura, escrita e execução, para os seguintes tipos de usuários: proprietário do arquivo, grupo de usuários ou qualquer outro usuário.

Uma característica importante do UNIX é o fato de que os programas acessam os dispositivos periféricos com a mesma sintaxe utilizada para o acesso aos arquivos regulares. Os dispositivos também são protegidos da mesma forma que os arquivos regulares.

O código abaixo ilustra o programa `copy` que copia o conteúdo de um arquivo para outro.

```
/* ------------------------------------------------------------------ */

#include <stdio.h>
#include <sys/fcntl.h>
#include <sys/stat.h>

char buffer[512];

void  copy(int old, int new)
{
  int count;
  while( (count = read(old, buffer, sizeof(buffer))) > 0 )
    write(new, buffer, count);
}


main(int argc, char *argv[])
{
  int fdold, fdnew;

  if(argc != 3)  {
    printf("Uso: copy f1 f2\n");
    exit(1);
  }

  /* abre arquivo para leitura */
  fdold = open(argv[1], O_RDONLY);
  if(fdold == -1)  {   /* erro no open */
    printf("Impossivel abrir %s\n", argv[1]);
    exit(1);
  }
```

```
  /* cria arquivo novo */
  fdnew = creat(argv[2], 0666);
  if(fdnew == -1)  {    /* erro no creat */
    printf("Impossivel criar %s\n", argv[2]);
    exit(1);
  }

  /* chama copy */
  copy(fdold, fdnew);
  exit(0);
}
```

/* ——— */

O Ambiente de Processamento

Um programa é um arquivo executável, e um processo é uma instância do programa em execução. No UNIX vários processos podem executar simultâneamente, sendo que várias instâncias de um mesmo programa podem existir ao mesmo tempo no sistema.

O programa abaixo ilustra os comandos `fork`, `execl`, `wait` e `exit` (implícito) utilizados na criação e sincronização de processos.

/* ——— */

```
#include <stdio.h>
#include <sys/wait.h>
#include <sys/time.h>

main(int argc, char *argv[])
{
  int pid;
  struct timeval tv1, tv2;
  double t1, t2;

  pid = fork();                                /* fork */
  if(pid == 0) execl(argv[1], NULL);   /* processo filho */
  gettimeofday(&tv1, NULL);   /* processo pai continua ... */
  t1 = (double)(tv1.tv_sec) + (double)(tv1.tv_usec)/ 1000000.00;
  wait(NULL);           /* sincroniza com o termino do filho */
  gettimeofday(&tv2, NULL);
  t2 = (double)(tv2.tv_sec) + (double)(tv2.tv_usec)/ 1000000.00;

  printf("\n0 tempo de execucao de %s eh: %lf\n", argv[1], (t2 - t1));
}
```

/* ——— */

Uma das características marcantes do UNIX é que este não suporta, no nível do núcleo, muitas das funções que fazem parte dos núcleos de outros sistemas operacionais. No caso do UNIX, estas funções são, em geral, programas situados no nível do usuário. O exemplo de programa mais destacado neste caso é o programa *shell* que é o responsável pela interpretação dos comandos do usuário.

Na maior parte das vezes o *shell* executa o comando `fork` e o processo filho executa o comando solicitado através da chamada `exec`. As palavras restantes na linha de comando são tratadas como parâmetros do comando. O *shell* aceita três tipos de comandos:

- arquivo executável produzido por compilação;
- arquivo executável contendo uma sequência de linhas de comando do *shell*;
- comando interno do *shell*.

O *shell* normalmente executa um comando sincronamente, ou seja, espera o término do processo associado ao comando antes da leitura de uma nova linha de comando. Também é possível a execução assíncrona do comando. Neste caso, o *shell* lê e executa a próxima linha de comando sem esperar o término do processo associado ao comando anterior, o qual executa em *background*.

Como o *shell* é um programa que se situa no nível do usuário, não fazendo parte do núcleo, é fácil modificá-lo para um ambiente particular. De fato, o UNIX disponibiliza vários *shells* para o usuário, cada qual com determinados recursos (por exemplo, capacidade de edição das linhas de comando e capacidade de completar nomes de arquivos).

Modularidade no Ambiente UNIX

O ambiente tem como filosofia permitir aos usuários o desenvolvimento de programas pequenos e modulares que possam ser usados como blocos primitivos na construção de programas mais complexos. Existem duas formas de compor programas no UNIX:

- redirecionamento de entrada/saída (E/S): os processos possuem, convencionalmente, acesso a três tipos de arquivos padrão: entrada, saída e erro.

  Processos que são executados a partir de um terminal possuem, tipicamente, o terminal como arquivo de entrada, saída e erro. Estes arquivos podem ser redirecionados independentemente. Exemplo:

  - `ls`: lista todos os arquivos do diretório corrente na saída padrão;
  - `ls > output`: redireciona a saída padrão para o arquivo chamado "output" no diretório atual;
  - `mail mjb < carta`: faz com que o programa *mail* (correio eletrônico) leia o conteúdo da mensagem do arquivo "carta", e não do terminal.

- pipe: permite que um fluxo de dados seja estabelecido entre um processo produtor e um processo consumidor.

  Processos podem redirecionar a sua saída padrão para um *pipe* a ser lido por outro processo que tenha redirecionado a sua entrada padrão para o mesmo *pipe*. Exemplo:

  - `grep` main a.c b.c c.c
    Lista as ocorrências da palavra "main" nos arquivos a.c, b.c e c.c.
  - `grep` main a.c b.c c.c | `wc` -l
    Submete a saída do comando anterior a um utilitário que conta o número de linhas de um arquivo (`wc`, opção -l).

Serviços do Sistema Operacional

Dentre os serviços suportados pelo núcleo do UNIX temos:

- controle de execução dos processos: criação, terminação, suspensão, comunicação entre processos;
- escalonamento (ordem de acesso à CPU) de processos;
- alocação de memória principal para execução dos processos. Caso a memória esteja escassa, o núcleo move temporariamente processos da memória primária para a secundária[3];
- alocação de memória secundária para armazenamento/recuperação eficiente dos dados do usuário (este serviço constitui o sistema de arquivos);
- acesso controlado aos dispositivos periféricos tais como terminais, fitas, discos, redes, etc.

O núcleo fornece estes serviços de forma transparente. Exemplo:

- o núcleo deteta que um dado arquivo é um arquivo regular ou um dispositivo, mas esconde esta distinção dos processos do usuário;
- o núcleo formata os dados em um arquivo para fins de armazenamento interno, entretanto, este formato é escondido do usuário, sendo retornado para este um "stream" não formatado de bytes.

Aspectos do Hardware

A execução dos processos do usuário no ambiente UNIX é dividida em 2 níveis: usuário e núcleo. Quando um processo executa uma chamada do sistema, o modo de execução do processo muda do modo usuário para o modo núcleo. As diferenças entre estes 2 modos são:

- processos no modo usuário podem acessar as suas instruções e dados, mas não as instruções e dados do núcleo ou de qualquer outro processo. Processos no modo núcleo podem acessar endereços do núcleo ou do usuário;
- algumas instruções são privilegiadas e resultam em erro quando executadas no modo usuário.

Interrupções e Exceções

O UNIX permite que dispositivos tais como periféricos de E/S ou o relógio do sistema interrompam a CPU assincronamente. Geralmente, o hardware define prioridades para os dispositivos de acordo com a ordem na qual as interrupções deverão ser atendidas caso ocorram simultaneamente.

Uma condição de exceção refere-se à ocorrência de um evento não esperado provocado pelo processo. Alguns destes eventos podem ser: endereçamento ilegal da memória, execução de instrução privilegiada, divisão por zero, etc.

[3]Esta transferência pode ser do processo completo *(swapping)*, ou de segmentos do processo (paginação).

As exceções podem ser caracterizadas como algo que ocorre no curso da execução de uma instrução, onde o sistema tenta reiniciar a instrução após tratar a exceção. No caso das interrupções, estas podem ser consideradas como se ocorressem entre a execução de duas instruções, sendo que o sistema continua a executar a partir da próxima instrução após tratar a interrupção.

O UNIX utiliza um mesmo mecanismo para manipular as condições de interrupção e exceção.

Níveis de Execução do Processador

O núcleo necessita muitas vezes impedir a ocorrência de interrupções durante a execução de atividades críticas. Exemplo: o núcleo não deve aceitar interrupção do disco enquanto estiver operando sobre estruturas de dados internas, isto porque o tratamento da interrupção pode interferir na atualização das estruturas provocando inconsistências.

Normalmente, os computadores possuem instruções privilegiadas que permitem definir o nível de execução do processador. A atribuição do nível de execução do processador em um determinado valor mascara a interrupção daquele nível e dos níveis inferiores (tornando habilitadas as de níveis superiores).

Na Fig. 1.3, caso o núcleo mascare a interrupção do disco, todas as interrupções, exceto a do relógio e dos erros da máquina, são enfileiradas para tratamento a *posteriori*.

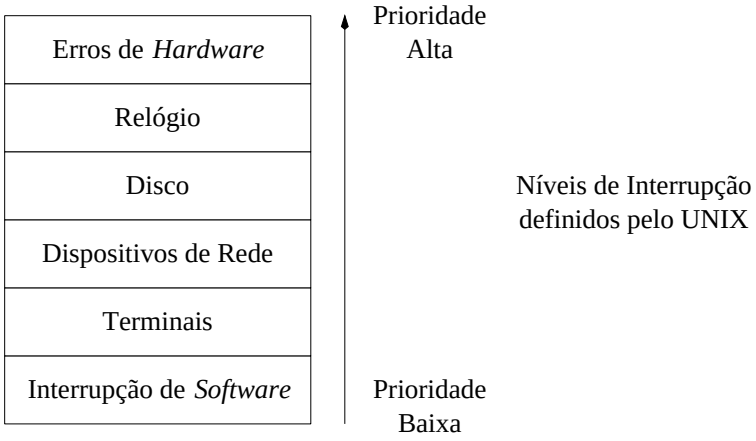

Fig. 1.3: Níveis de interrupção definidas pelo UNIX

Gerenciamento de Memória

O núcleo reside permanentemente na memória principal, assim como, os processos em execução (ou pelo menos parte deles).

Quando da compilação, são gerados endereços no programa que representam variáveis e instruções. O compilador gera estes endereços para uma máquina virtual como se nenhum outro programa fosse executar simultaneamente na máquina real. Quando da execução do programa, o núcleo aloca espaço na memória principal através do mapeamento do endereço virtual no endereço físico da máquina. Este mapeamento depende das características do hardware da máquina.

1.5 Arquitetura do Sistema Operacional UNIX

Uma visão mais detalhada da arquitetura do núcleo do UNIX é mostrada na Fig. 1.4.

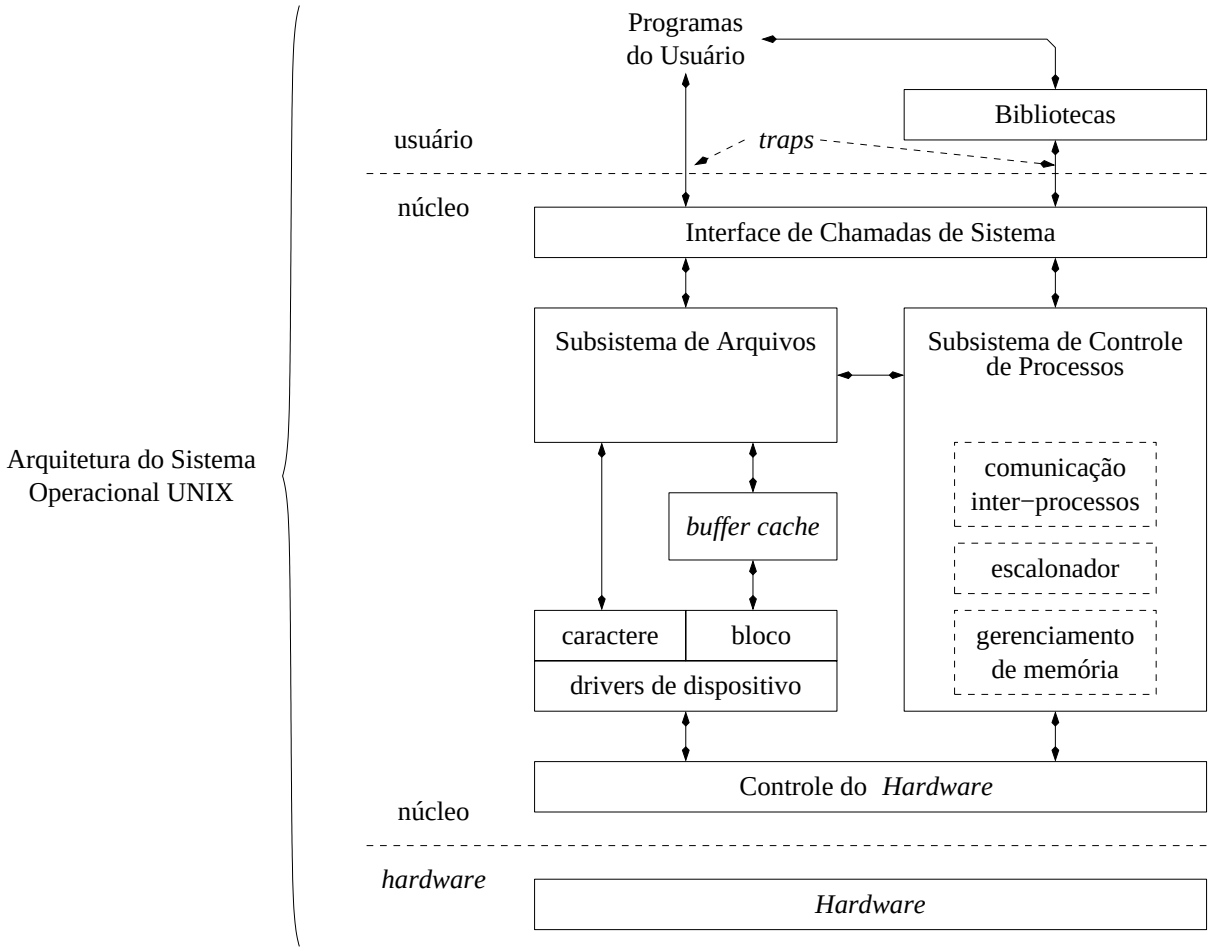

Fig. 1.4: Arquitetura do núcleo do sistema operacional UNIX

Os programas em linguagem de máquina podem evocar chamadas ao sistema diretamente, isto é, sem o uso da biblioteca. Por sua vez, os programas em linguagem de alto nível realizam as chamadas como se estivessem evocando funções ordinárias, cabendo à biblioteca mapear estas chamadas de funções nas primitivas necessárias para acessar o sistema operacional. Outras bibliotecas permitem um uso mais sofisticado das chamadas ao sistema (exemplo: biblioteca de E/S).

A Fig. 1.4 divide as chamadas ao sistema em chamadas ao sub-sistema de arquivos e ao sub-sistema de controle dos processos. O sub-sistema de arquivos acessa os dados nos arquivos através de um mecanismo de "buferização" que, através da interação com os *drivers* de dispositivos de E/S orientados a bloco, regula o fluxo de dado entre o núcleo e os dispositivos de armazenamento secundário. Os dispositivos de E/S orientados a bloco são dispositivos de armazenamento de acesso randômico.

O sub-sistema de arquivo interage diretamente com dispositivos que não são do tipo bloco (terminais, por exemplo). Neste caso, não há a intervenção do mecanismo de "buferização".

Os sub-sistemas de arquivo e de controle dos processos interagem quando do carregamento de um arquivo na memória para execução. O módulo de gerenciamento da memória controla a alocação de memória, ou seja, caso em um determinado instante o sistema não possua memória

física suficiente para todos os processos, o núcleo move-os entre a memória física e a memória secundária de modo a que todos os processos tenham as mesmas chances de execução. Duas políticas são normalmente utilizadas: permuta *(swapping)* e paginação.

O módulo de escalonamento aloca a CPU aos processos, os quais executam até o instante em que liberam a CPU para aguardar um recurso, ou então, são "preemptados" porque a execução excedeu o "quantum" de tempo disponível para o processo. Neste caso, o escalonador escolhe o processo pronto de maior prioridade.

Ainda com relação aos processos, existem várias formas de comunicação entre estes, variando desde a sinalização assíncrona de eventos até a transmissão síncrona de mensagens.

Uma Visão do Sistema de Arquivos

A representação interna de um arquivo é dado por um *inode*. Este contém uma descrição do *layout* no disco do arquivo de dado, assim como outras informações tais como: proprietário do arquivo, permissões de acesso e instantes de acesso.

Todo arquivo possui um *inode*, o qual é alocado quando da sua criação, podendo possuir, entretanto, vários nomes, todos mapeados no mesmo *inode*. Cada um destes nomes denomina-se *link*. Os *inodes* são armazenados no sistema de arquivos e são lidos em uma *tabela de inodes* (em memória) quando da manipulação dos respectivos arquivos.

Duas outras estruturas de dados são importantes: *tabela de arquivo* (TA) e *tabela descritora de arquivo do usuário* (TDAU), sendo que TA é uma estrutura global ao núcleo enquanto uma TDAU é criada para cada processo. Quando da criação/abertura de um arquivo, o núcleo associa uma entrada de cada uma das tabelas ao *inode* correspondente ao arquivo, permitindo que as entradas destas três estruturas -TA, TDAU e *inode*- mantenham o estado do arquivo, assim como, os direitos de acesso ao arquivo.

TA mantém o *offset*, no arquivo correspondente, do próximo byte a ser lido/escrito, assim como, os direitos de acesso do processo;

TDAU identifica todos os arquivos abertos para o processo.

A Fig. 1.5 ilustra o uso das tabelas TDAU, TA e de *inodes*. Note um link onde dois campos na TDAU apontam para o mesmo campo na TA.

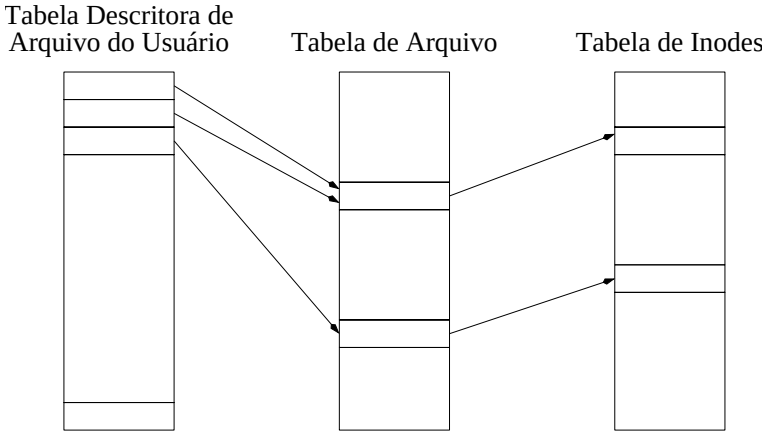

Fig. 1.5: Estruturas de dado do sistema de arquivos

O núcleo retorna um descritor de arquivo quando das chamadas `open` e `create`, o qual corresponde a um índice na TDAU. Quando da execução de um `write` ou um `read`, o núcleo utiliza o descritor de arquivo para acessar a TDAU e através desta alcançar a TA e o *inode* do arquivo onde, através deste último, o núcleo encontra o dado no arquivo. Esta arquitetura dos dados permite vários níveis de acesso compartilhado ao arquivo.

Uma instalação pode possuir várias unidades físicas de disco, cada uma delas contendo um ou mais sistemas de arquivo. O núcleo relaciona-se com os sistemas de arquivo de um ponto de vista lógico ao invés de tratar com discos. Cada dispositivo lógico é identificado por um número do dispositivo lógico. A conversão entre os endereços do dispositivo lógico (sistema de arquivo) e os endereços, no dispositivo físico (disco) é realizada pelo *driver* do disco.

Um sistema de arquivos consiste de uma sequência de blocos lógicos, cada um contendo qualquer múltiplo de 512 bytes. O tamanho de um bloco lógico é homogêneo dentro do sistema de arquivos podendo, entretanto, variar para diferentes sistemas de arquivo em uma dada configuração. Blocos maiores representam um aumento na taxa de transferência dos dados entre a memória e o disco. Entretanto, blocos maiores demandam mais espaço em memória para manipulá-los. Um sistema de arquivos possui a seguinte estrutura (ver Fig. 1.6):

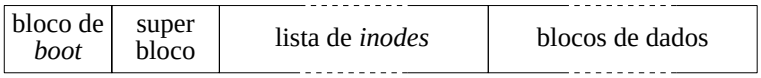

Fig. 1.6: Estrutura do sistema de arquivos

- bloco de *boot*: contém o código do *bootstrap* que é lido na máquina quando da partida do sistema operacional;
- super-bloco: descreve o estado de um sistema de arquivo;
- lista de *inodes*: tem o seu tamanho definido quando da configuração do sistema de arquivos. Um dos *inodes* corresponde à raiz do sistema de arquivo. É através deste *inode* que a estrutura de diretórios do sistema é acessada;
- bloco de dados: contém arquivos e dados administrativos. Um bloco de dados só pode pertencer a um único arquivo do sistema.

Processos

Um processo corresponde à execução de um programa e consiste de um conjunto de bytes que a CPU interpreta como instrução de máquina, dado e pilha.

O processo executa uma sequência de instruções que é auto-contida e que não salta para um outro processo. Ele lê e escreve seus dados nas suas áreas de dado e pilha, mas não pode ler ou escrever nas áreas de dado e pilha de outro processo. Os processos comunicam com outros processos e com o resto do sistema através de chamadas de sistema.

Do ponto de vista prático, um processo em UNIX é uma entidade criada pela chamada `fork`. Exceto o primeiro, qualquer outro processo é criado através da chamada `fork`. O processo que chamou o `fork` é identificado como "processo pai" e o que acabou de ser criado é identificado como "processo filho". Todo processo tem um único pai mas pode ter vários filhos. O núcleo identifica cada processo através de um número denominado *process identifier* (PID). No caso do processo filho, ele recebe como retorno após a execução do `fork` o valor 0, enquanto o processo

pai recebe um valor diferente de 0 que corresponde ao PID do filho. Através do teste do valor retornado pelo `fork`, um processo pode distinguir se ele é o processo pai ou o processo filho e, em consequência, tomar a ação correspondente.

O processo 0 é um processo especial criado quando da iniciação do sistema *(boot)*. Após criar o processo 1, conhecido como *init*, o processo 0 torna-se o processo *swapper*. O processo 1 é ancestral de qualquer outro processo no sistema, possuindo uma relação especial com estes, relação esta que será discutida nos capítulos subsequentes.

Um arquivo executável gerado quando da compilação de um programa consiste das seguintes partes:

- um conjunto de cabeçalhos que descrevem os atributos dos arquivos;
- o texto do programa;
- representação em linguagem de máquina dos dados que possuem valor inicial e uma indicação de quanto espaço o núcleo necessita alocar para os dados sem valor inicial, denominado *bss*;
- outras seções tais como informações sobre a tabela de símbolos.

O núcleo carrega um arquivo executável, gerado pelo compilador, durante a execução de uma chamada `exec`, consistindo o processo carregado de três partes: texto, dado e pilha.

As regiões de texto e dado correspondem às seções do texto e dados iniciados e não-iniciados (bss). A pilha é criada automaticamente e o seu tamanho é ajustado dinamicamente pelo núcleo em tempo de execução. Um quadro *(frame)* da pilha contém os parâmetros para a função chamada, suas variáveis locais e os dados necessários (apontador de pilha e contador de programa) para recuperar o quadro anterior na pilha.

Como um processo no UNIX pode executar em 2 modos, núcleo ou usuário, utiliza-se uma pilha separada para cada modo. O lado esquerdo da Fig. 1.7 mostra a pilha do usuário para o programa *copy* quando da chamada de sistema `write`.

Cada chamada de sistema possui uma entrada na biblioteca de chamadas de sistema, a qual é codificada em assembler, contendo instruções especiais *(traps)* que, quando executadas, provocam uma interrupção resultando em um chaveamento no hardware para o modo núcleo passando a utilizar a pilha do núcleo. A construção da pilha do núcleo ocorre nos mesmos moldes da construção da pilha no modo usuário.

Todo processo possui uma entrada na *tabela de processos* (TP) do núcleo e a cada um é alocada uma área U que contém dados privados manipulados somente pelo núcleo. A TP aponta para uma *tabela de regiões do processo (pregion)*, cujas entradas apontam para entradas na *tabela de região*. Uma região é uma área contígua de um espaço de endereçamento do processo, tal como: texto, dado e pilha.

As entradas na tabela de região descrevem os atributos da região, ou seja, se a região contém texto ou dado, se é uma região compartilhada ou privada e se o conteúdo da região encontra-se em memória.

O nível extra de encadeamento, ou seja, da *pregion* para a tabela de região, permite que processos independentes compartilhem regiões de memória.

Quando um processo evoca a chamada `exec` o núcleo aloca regiões para o texto, dado e pilha do processo que está sendo criado, após liberar as regiões antigas do processo que estava executando. Quando um processo evoca `fork` o núcleo duplica o espaço de endereçamento do processo antigo permitindo, quando possível, que processos compartilhem regiões ou, caso

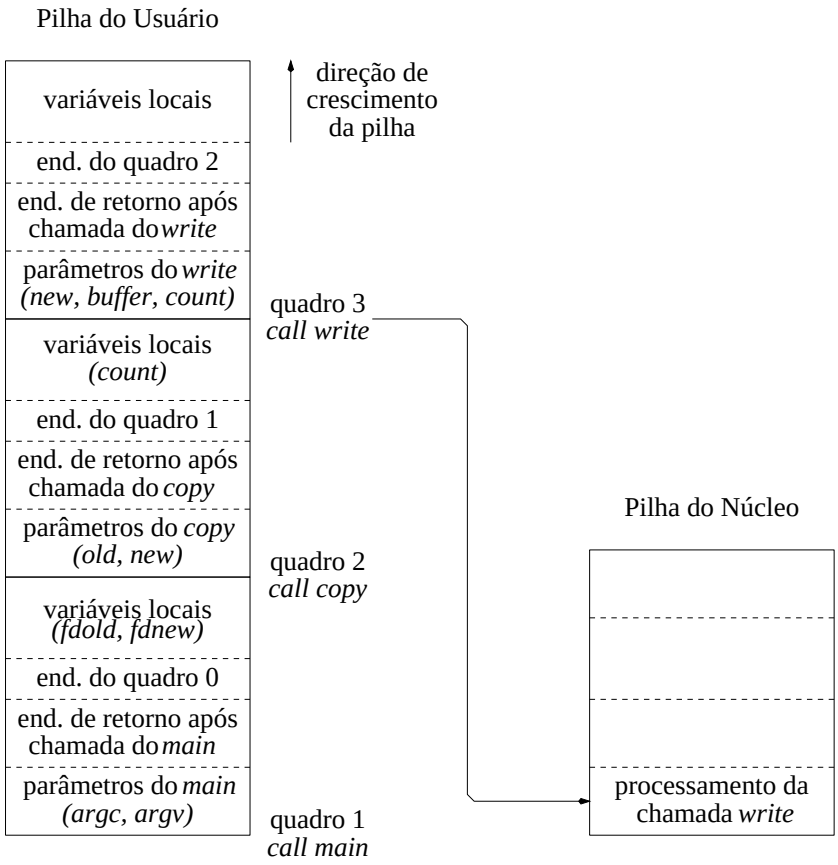

Fig. 1.7: Estado das pilhas para o programa *copy*

contrário, fazendo uma cópia da região. Quando um processo evoca `exit` o núcleo libera as regiões que o processo estava usando. A Fig. 1.8 ilustra as estruturas de dados associadas ao controle dos processos.

A entrada na tabela de processos e a área U contém informações de controle e status sobre o processo. A área U pode ser vista como uma extensão da entrada do processo na tabela de processos.

Campos importantes da tabela de processos:

- campo de estado;
- identificadores dos usuários que possuem o processo;
- um conjunto descritor de evento quando o processo está bloqueado.

A área U contém informações que descrevem o processo e que são acessadas somente durante a execução do processo. Os campos mais importantes são:

- apontador para o campo na TP do processo em execução;
- descritores de arquivo para todos os arquivos abertos;
- parâmetros internos de E/S;
- limites de tamanho do processo e arquivo.

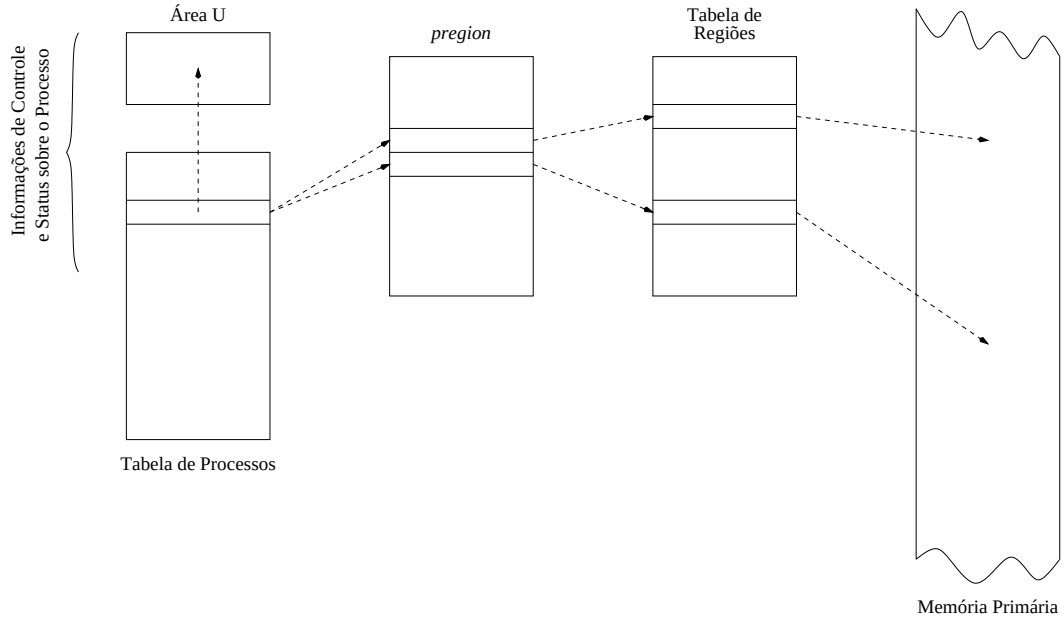

Fig. 1.8: Estruturas de dados associadas ao controle dos processos.

Contexto de um Processo

O contexto de um processo é o estado definido pelo seu texto correspondendo aos valores das suas variáveis globais e estruturas de dados, os valores dos registros de máquina usados, os valores armazenados no seu "slot" na tabela de processos e na área U e o conteúdo das suas pilhas de usuário e núcleo. Quando o núcleo decide executar um novo processo realiza-se uma mudança de contexto.

Quando da realização de uma mudança de contexto o núcleo salva informações suficientes de modo que posteriormente ele possa recuperar o contexto do processo e continuar a sua execução. Da mesma forma, quando da mudança do modo usuário para o modo núcleo, o núcleo salva as informações necessárias para que o processo possa retornar ao modo usuário e continuar a execução. Neste último caso, temos uma mudança de modo e não de um chaveamento de contexto.

Estados do Processo

O ciclo de vida de um processo pode ser representada por um conjunto de estados (Fig. 1.9):

- executando no modo usuário;
- executando no modo núcleo;
- pronto;
- bloqueado (dormindo).

O núcleo protege a sua consistência permitindo chaveamento de contexto apenas quando o processo transita do estado "executando no modo núcleo" para o modo "bloqueado". O núcleo também eleva o nível de execução do processador quando da execução de regiões críticas de modo a impedir interrupções que possam provocar inconsistências em suas estruturas de dados.

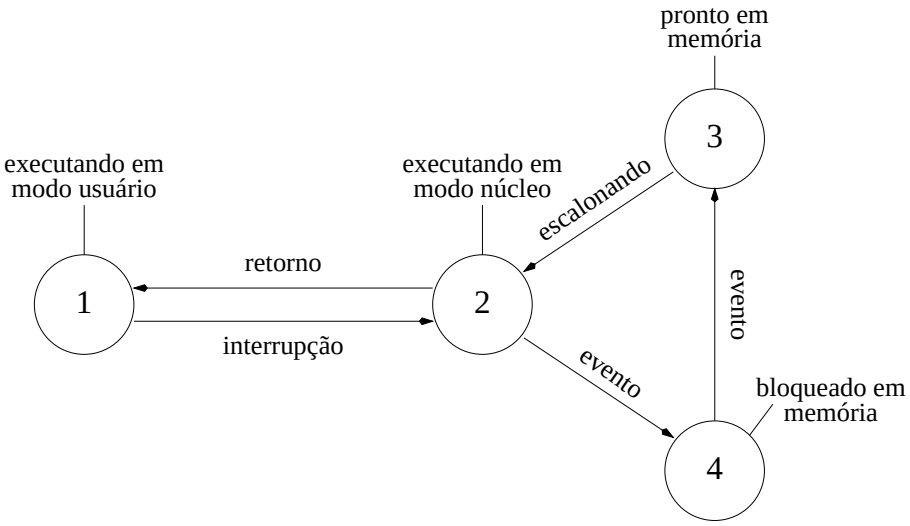

Fig. 1.9: Estados de um processo

O escalonador de processo realiza, periodicamente, a "preempção" de processos executando no modo usuário de forma a que os processos não monopolizem a CPU.

Capítulo 2

Processos

2.1 Introdução

No capítulo anterior definimos o conceito de processo, bem como algumas generalidades sobre como o sistema operacional UNIX gerencia processos. Neste capítulo, avançaremos no estudo de processos, analisando problemas de concorrência, escalonamento e comunicação inter-processos [1].

2.1.1 Modelo de Processos

A maioria dos computadores modernos são capazes de realizar diversas atividades em paralelo. Enquanto executa um programa do usuário, o computador pode ler um disco ou utilizar a impressora. Em sistemas multiprogramados, a CPU é comutada de programa a programa em períodos da ordem de milisegundos, dando ao usuário a impressão de paralelismo.

O gerenciamento de atividades paralelas é difícil de ser implementado com eficiência. Entretanto, projetistas de sistemas operacionais ao longo dos anos vêm desenvolvendo modelos objetivando tornar esta tarefa mais simples.

No modelo mais empregado atualmente, todos os programas executáveis no computador, muitas vezes incluindo subsistemas do sistema operacional, estão organizados na forma de processos. Conceitualmente, cada processo tem uma própria CPU virtual (tabela armazenando conteúdo de registradores, contador de programa, etc). A posse da CPU real é passada periodicamente de processo a processo. O sistema pode ser visto como uma coleção de processos sendo executados em *pseudo* paralelismo[1]. Conforme definido anteriormente, a habilidade de executar múltiplos programas em uma única CPU denomina-se multiprogramação.

Em sistemas multiprogramados, a velocidade de execução de um processo é função da quantidade de processos competindo pela CPU. Isto implica que o tempo de execução de um processo varia a cada nova execução, dependendo da "carga" da máquina. Assim sendo, processos não devem ser programados com considerações intrínsecas de tempo. Quando são requeridas considerações de tempo real, medidas especiais devem ser tomadas para assegurar que estas irão ocorrer.

[1] Paralelismo *real* é obtido apenas com a utilização de múltiplas CPUs.

2.1.2 Concorrência e Regiões Críticas

Em muitos sistemas operacionais, processos frequentemente compartilham outros recursos além da CPU. Por exemplo, durante uma chamada de sistema um processo pode ter acesso a uma tabela mantida pelo núcleo. Neste caso, o núcleo deve inibir a comutação da CPU para outro processo, até que todas as operações na tabela sejam efetuadas. Caso contrário, a tabela fatalmente assumiria um estado inconsistente onde apenas algumas alterações foram processadas.

Em situações como a exemplificada acima, a tabela é definida como um *recurso compartilhado*, e a parte do código que o manipula como uma *região crítica*. A execução de uma região crítica deve ser um procedimento controlado a fim de evitar que os recursos compartilhados atinjam estados inconsistentes.

A chave para prevenir problemas em áreas compartilhadas é proibir que mais de um processo leia ou escreva dados compartilhados ao mesmo tempo, ou seja, deve-se garantir a *mútua exclusão*. Se for garantido que nenhum par de processos estejam executando ao mesmo tempo uma região crítica delimitando um mesmo recurso compartilhado, inconsistências são evitadas[2].

Embora este quesito evite inconsistências, o mesmo não garante eficiência na utilização dos recursos compartilhados. Para assegurarmos uma boa solução, é necessário que:

- dois processos não estejam simultaneamente dentro de suas regiões críticas referentes ao mesmo recurso compartilhado (garantia de mútua exclusão);
- a garantia de mútua exclusão se dê independente da velocidade relativa dos processos ou número de CPUs;
- nenhum processo executando fora de regiões críticas bloqueie outro processo;
- nenhum processo espere um tempo arbitrariamente longo para executar uma região crítica (ou sofra "estarvação").

Vários algoritmos de controle visando garantir as propriedades acima foram propostos. Estes algoritmos são classificados segundo o modo com que esperam pela autorização de entrada em uma região crítica: espera ocupada (competindo pela CPU durante a espera) ou bloqueada (não competindo pela CPU).

Todo algoritmo de mútua exclusão possui, invariavelmente, duas partes (implementadas como duas funções distintas). A primeira função é evocada quando o processo deseja iniciar a execução de uma região crítica. Quando esta função retorna, o processo está apto a executar a região crítica. No final da execução, o processo evoca a segunda função, anunciando que a região crítica já foi executada. Esta segunda função, via de regra, provoca o retorno da primeira função em outro processo.

Para permitir que regiões críticas que acessam recursos compartilhados distintos possam ser executadas ao mesmo tempo, cada recurso compartilhado possui um identificador (via de regra um número inteiro). Assim sendo, as duas funções que compõem o algoritmo de garantia de mútua exclusão possuem este identificador como parâmetro.

[2]Note que regiões críticas delimitando diferentes recursos podem ser executadas por diferentes processos ao mesmo tempo.

2.1.3 Mútua Exclusão Com Espera Ocupada

Nesta seção analisaremos algumas propostas para garantir exclusão mútua nas regiões críticas, não permitindo que mais de um processo possa manipular um recurso compartilhado ao mesmo tempo. Em todas, o processo que está tentando acessar uma região crítica em execução por outro processo permanece em espera ocupada, isto é, competindo pela CPU mas sem avançar no seu processamento. Por esta razão, os métodos que empregam espera bloqueada são os mais utilizados na prática.

Desabilitar Interrupções

A solução mais simples é o método de desabilitar todas as interrupções quando se está entrando na região crítica, e reabilitá-las ao sair. Esta proposta não é muito atrativa, pois dá ao processo usuário poder de desabilitar todas as interrupções, inclusive aquelas que permitem o núcleo reassumir a CPU. Caso o processo não as reabilite por algum motivo, o sistema operacional jamais reassume o controle do hardware.

Em outro aspecto, é conveniente para o sistema operacional desabilitar interrupções durante algumas instruções, enquanto está atualizando variáveis internas. Assim, desabilitar interrupções é uma solução útil para o sistema operacional, não para processos de aplicação.

Variáveis LOCK

Uma segunda tentativa leva-nos a uma solução por software. Considere uma variável simples e compartilhada[3] LOCK, inicialmente igual a 0. Quando um processo deseja entrar em sua região crítica ele primeiro testa o LOCK. Se for 0, o processo altera para 1 e executa a região crítica. Se for 1 ele espera até que seja 0. Embora pareça uma boa solução, o que irá ocorrer se ambos testam uma variável de valor 0 ao mesmo tempo ?

Alternância Estrita

Esta proposta define uma variável TURN, inicialmente 0. Ela indica quem deve esperar e quem pode entrar na seção crítica. Se TURN for 0, o processo 0 pode entrar na região crítica. Ao sair, deve passar o valor de TURN para 1. Quando TURN é 1 o processo 1 pode entrar na seção crítica. Ao sair passa o valor de TURN para 0[4].

Este algoritmo garante a mútua exclusão. Entretanto, os processos estritamente se alternam na posse do recurso compartilhado. Isto faz com que um processo necessite aguardar o acesso a um recurso compartilhado por todos os demais até que chegue novamente a sua vez. O que ocorre quando o número de acessos for diferente entre os processos ?

Solução de Peterson

Obtida pela combinação das idéias de variáveis LOCK e TURN, criando-se também uma solução por software para o problema. Esta solução evita os problemas individuais das soluções anteriores, mas é pouco utilizada na prática por utilizar espera ocupada.

[3] Uma para cada recurso compartilhado.

[4] Este algoritmo é facilmente generalizado para N processos, $N > 2$.

Instrução TSL

Esta proposta requer uma pequena ajuda do hardware. Ela utiliza a instrução TSL *(Test and Set Lock)* presente em muitos processadores. Esta instrução permite a implementação de variáveis *LOCK* cujo teste e atualização são atômicos (em outras palavras, a instrução TSL é indivisível mesmo frente a interrupções de hardware).

2.1.4 Mútua Exclusão com Espera Bloqueada

Serão apresentados a seguir alguns mecanismos de garantia de mútua exclusão que bloqueiam os processos quando tentam executar uma região crítica "ocupada". São mais eficientes que os anteriores, posto que processos bloqueados não competem pela CPU.

Sleep e Wakeup

Um dos métodos mais simples consiste do par *sleep* e *wakeup*. *sleep* é uma chamada de sistema que muda o estado de um processo em execução para bloqueado. Um processo bloqueado volta a tornar-se ativo quando outro o desbloqueia através da chamada *wakeup*. O método é o mesmo que emprega variáveis LOCK operadas por instruções TSL, exceto que quando a variável apresenta valor 1, o processo executa *sleep*. O processo que altera o valor de LOCK para 0 ao sair da região crítica é o responsável por ativar um processo bloqueado (via *wakeup*).

Infelizmente, com o emprego de apenas *sleep* e *wakeup* é fácil demonstrar e existência de um estado onde todos os processos encontram-se bloqueados. Esta situação é denominada *deadlock*.

Semáforos

São variáveis inteiras que contam o número de vezes que a operação *wakeup* tenha sido realizada. Duas operações, DOWN e UP (generalizações de *sleep* e *wakeup*) são definidas. A operação DOWN é executada no início da região crítica, enquanto UP é executada no final. O semáforo consiste de um contador iniciado em 1 e uma lista de processos aguardando liberação para executar a região crítica protegida pelo semáforo.

A operação DOWN decrementa o contador do semáforo de uma unidade e verifica seu valor. Se for igual a 0, retorna (fazendo com que o processo entre na região crítica). Se o valor for menor que 0, o processo é bloqueado e adicionado à lista de processos aguardando liberação. A operação UP incrementa o valor do semáforo. Se um ou mais processos estiverem bloqueados sobre aquele semáforo, um deles é escolhido da lista pelo sistema para completar a operação DOWN. Neste caso o sistema remove-o da lista e emite-lhe um sinal de *wakeup*.

As operações com semáforos são atômicas e implementadas com instruções TSL.

Contadores de Evento

Um outro tipo de variável de sincronização entre processos. Três operações são definidas para um contador de evento (E):

- READ(E): retorna o valor corrente de E;
- ADVANCE(E): incrementa atomicamente E;
- AWAIT(E,v): bloqueia até que E $\geq$ v.

Note que os contadores de eventos nunca decrescem e partem sempre de 0. Contadores de evento são mais convenientes que semáforos para problemas do tipo produtor-consumidor com *buffer* limitado.

Monitores

Semáforos e contadores de evento tornam simples a proteção de recursos compartilhados. Entretanto, uma simples troca na ordem da chamada das primitivas pode gerar uma situação de *deadlock*. Em suma, a utilização de semáforos e contadores de eventos deve se processar com extrema cautela.

Monitores são uma proposta de mecanismo de sincronização de alto nível. Um monitor é uma coleção de procedimentos, variáveis e estruturas de dados agrupados em um bloco. Os processos podem acessar os procedimentos do monitor mas não suas estruturas internas. Monitores têm uma característica importante: somente um processo pode estar ativo[5] no monitor em um dado instante (garantindo portanto a mútua exclusão).

Monitores constituem-se em um conceito de linguagem de programação, ou seja, o compilador reconhece que os monitores são especiais e pode manusear as chamadas do monitor diferentemente de outras chamadas. Esta é uma das desvantagens de monitores: precisam de linguagens de programação que os incorpore (por exemplo, Java implementa monitores através da palavra-chave `synchronized`).

2.1.5 Comunicação Inter-processos

Muitos autores consideram os mecanismos de mútua exclusão apresentados acima como formas de processos se comunicarem. Entretanto, preferimos considerar tais mecanismos como de sincronização inter-processos, usando o termo *comunicação* apenas quando ocorrer intercâmbio de informação entre processos. Sincronização são procedimentos de controle, normalmente usados para garantir mútua exclusão, e utilizados para gerenciar a *competição* entre os processos. Comunicação, por sua vez, visa promover a *cooperação* entre os processos.

Passagem de Mensagem

Este método de comunicação entre processos usa duas chamadas de sistema: `send` e `receive`.

- `send`(destino, mensagem): envia mensagem a um processo destino.
- `receive`(fonte, mensagem): recebe mensagem de um processo fonte.

Destino e fonte de mensagens são buffers alocados pelos processos para fins de envio e recepção de mensagens. Mensagens são estruturas tipadas ou não cujo conteúdo é interpretado unicamente pelos processos emissor e receptor da mensagem.

Compartilhamento de Dados

Processos podem se comunicar através do compartilhamento de uma área comum onde dados podem ser escritos por um e lidos por outro processo. O acesso a esta área comum deve ser disciplinado por um mecanismo de mútua exclusão (tipicamente semáforos) ou tornando as instruções de leitura e gravação atômicas. Duas primitivas são necessárias:

[5] Executando qualquer um de seus procedimentos.

- `store`(posição, dado): grava dados em uma certa posição;
- `fetch`(posição, dado): acessa dados de uma certa posição.

Chamada de Procedimentos Remotos

Chamada de procedimentos remotos (ou RPC) é uma forma mais estruturada de troca de mensagens entre processos servidores e clientes. Um processo servidor dispõe de um conjunto de serviços que um processo cliente evoca como se evocasse um procedimento local. O cliente indica o serviço desejado ao servidor, passando parâmetros para sua execução, se for o caso. Recebida a requisição, esta é processada pelo servidor[6] que retorna os resultados ao cliente. O envio e recepção de parâmetros e retornos se dá por troca de mensagens. Uma biblioteca de RPC possui duas primitivas básicas:

- `register_rpc`(serviço): utilizada por servidores para anunciar que serviços estão aptos a processar;
- `call_rpc`(serviço, parâmetros, resultados): utilizada por clientes para evocar serviços.

2.2 Escalonamento de Processos

Quando mais de um processo estiver ativo (pronto para executar), cabe ao sistema operacional decidir qual terá a posse da CPU. A parte do sistema operacional que toma esta decisão é chamada *escalonador* e o algoritmo utilizado é o *algoritmo de escalonamento*.

Vários critérios devem ser observados por um algoritmo de escalonamento:

1. *progresso*: garantir que cada processo tenha acesso à CPU;
2. *eficiência*: manter a CPU ocupada praticamente 100% do tempo;
3. *tempo de resposta*: minimizar o tempo de resposta na execução dos processos, principalmente os interativos (editores, planilhas, etc);
4. *tempo de espera*: minimizar o tempo de espera de serviços não interativos (compilação, impressão, etc);
5. *vazão*: maximizar o número de processos executados por unidade de tempo.

É importante observar que alguns desses objetivos são contraditórios. Se um algoritmo favorece o escalonamento de processos interativos certamente estará comprometendo os não interativos. Vejamos alguns algoritmos de escalonamento.

Escalonamento Round Robin

Este é o mais antigo e simples algoritmo de escalonamento. Cada processo é executado por um intervalo de tempo *(quantum)*. Se o processo ainda estiver executando ao final do *quantum*, ele é suspenso e a CPU é alocada a outro processo. Se o processo acabar ou for bloqueado antes do final do *quantum*, a CPU também é passada a outro processo. A única questão a ser analisada é o tamanho do *quantum*. Se for muito pequeno, diminui a eficiência da CPU, pois a alocação da CPU para outro processo implica um certo *overhead*. Se for muito grande, degrada a resposta para os processos interativos.

[6] Ou colocada em uma fila de espera.

Algoritmos com Prioridades

O algoritmo Round Robin faz a consideração que todos os processos são de igual importância. Certas aplicações, como controle de processos industriais, demandam um algoritmo de escalonamento com prioridades. A idéia básica é que cada processo tem uma prioridade e processos com prioridades superiores devem ser executados primeiro. Para prevenir que processos de alta prioridade executem indefinidamente, o escalonador, via de regra, diminui a prioridade dos processos com o aumento de seu respectivo tempo de execução.

Múltiplas Filas

Este é um algoritmo que define classes com prioridades. Processos na classe de menor prioridade são executados por um *quantum*. Processos na classe seguinte, por dois *quanta*. Na próxima classe por 4 *quanta*, e assim por diante. Quando um processo utiliza todos os *quanta* a ele alocados, o mesmo é interrompido e sua classe tem a prioridade diminuída. Este algoritmo diminui o número de comutações da CPU entre os processos ativos.

Tarefas Pequenas Primeiro

Este algoritmo é indicado para aplicações não interativas, onde o tempo médio de execução é conhecido a priori. O algoritmo define que as tarefas menores devem ser executadas primeiro. Prova-se que esta política minimiza o tempo médio de espera das tarefas.

Algoritmo "Policy-Driven"

Este algoritmo particiona a CPU de forma equânime entre os usuários (não entre os processos). O algoritmo define que se existirem n usuários ligados ao sistema, e cada usuário deverá receber $1/n$ do poder da CPU. Para isto, o sistema deve manter informações do tempo de CPU que cada usuário já dispôs desde que entrou no sistema, e do instante de tempo que cada usuário ligou-se ao sistema.

Escalonamento em Dois Níveis

Até agora foi considerado que todos os processos residem em memória primária. Entretanto se esta memória for insuficiente, processos ativos podem ser armazenados temporariamente em memória secundária (tipicamente disco). O meio mais prático para controlar a comutação de processos é definir dois níveis de escalonamento. Um escalonador de baixo nível se restringe a troca de processos que estão na memória primária no momento. Um escalonador de alto nível decide sobre a troca dos processos entre as memórias primária e secundária.

2.3 Gerenciamento de Processos no UNIX

No capítulo 1 introduzimos o conceito de processo, como são criados e como o sistema operacional os mantém. Nesta seção detalharemos mais a estruturação de processos no UNIX e apresentaremos como processos são controlados, escalonados e se comunicam [3].

Transições de Estado

Na Fig. 1.9 apresentamos um diagrama de estados simplificado onde um processo possuía 4 estados: executando em modo usuário, executando em modo núcleo, pronto e bloqueado (dormindo). A rigor, um processo no UNIX apresenta 9 estados (Fig. 2.1):

1. Executando em modo usuário;
2. Executando em modo núcleo;
3. Pronto para execução (aguardando apenas CPU) e residindo em memória primária;
4. Bloqueado (dormindo) e residindo em memória primária;
5. Pronto para execução, mas residindo em memória secundária (aguardando *swapping*);
6. Bloqueado (dormindo) e residindo em memória secundária;
7. O processo está saindo do modo núcleo e retornando ao modo usuário, quando ocorre uma mudança de contexto e o processo perde a CPU;
8. O processo acabou de ser criado e está em transição para "pronto";
9. O processo executou um `exit`, não mais existe, mas seu registro é mantido até que seja enviado ao processo pai seu código de retorno e outras estatísticas.

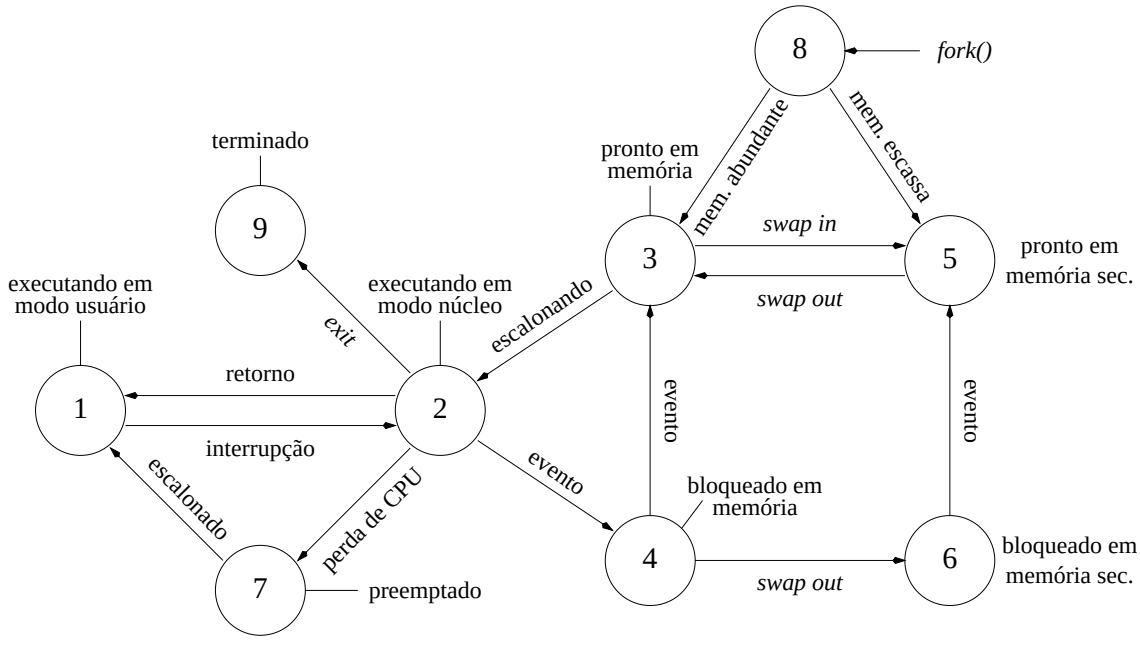

Fig. 2.1: Diagrama completo de transição de estados para processos

Vejamos o que tipicamente ocorre a partir da criação de um processo. Após a execução de uma chamada `fork`, são criados para o novo processo um campo na tabela de processos, sua área U e apontadores para sua *pregion*. O processo encontra-se no estado 8. Dependendo da disponibilidade ou não de memória primária, o processo move-se, respectivamente, para os estados 3 ou 5. Assuma que o processo deslocou-se para o estado 3 (pronto em memória

primária). O escalonador seleciona então o processo para executar, movendo-o para o estado 2 (executando em modo núcleo), onde a chamada `fork` será completada para este processo, retornando 0. A partir daí, o processo entra em execução no modo usuário, processando suas instruções uma a uma. Expirado seu *quantum* de CPU, uma interrupção de relógio faz com que o processo retorne ao modo núcleo novamente. Terminado o tratamento da interrupção, o escalonador pode decidir alocar a CPU a um outro processo, movendo-o para o estado 7. Este estado é similar ao estado 3, sendo a distinção feita para enfatizar que o processo tem a CPU tomada somente quando está apto a retornar para o modo usuário, diferente do estado 3 onde deve voltar ao modo núcleo para completar uma chamada de sistema.

A eventual execução de uma chamada de sistema faz com que o processo abandone o modo usuário (estado 1) e continue sua execução no modo núcleo (estado 2). Suponha que o processo requeira uma operação de E/S do disco. O núcleo coloca o processo no estado 4 (dormindo em memória) até que o processo seja notificado que a operação de E/S se completou (mais precisamente, quando a operação se completa, o hardware interrompe a CPU, cujo tratamento da interrupção resulta no "acordar" do processo). Se ao ser desbloqueado o processo ainda estiver residindo em memória primária, o mesmo é movido para o estado 3, aguardando CPU.

Entretanto, se durante sua permanência no estado 4 o núcleo necessitar de espaço na memória primária, o processo sofre *swapping*, sendo removido da memória primária e armazenado em memória secundária (tipicamente disco). Neste caso, o processo atinge o estado 6 (dormindo em memória secundária). Uma vez completada a operação de E/S com o processo no estado 6, este transita para o estado 5 (pronto em memória secundária). Quando o *swapper* escolhe o processo para alocá-lo novamente em memória primária, este volta para o estado 3.

No estado 3, quando o escalonador volta a atribuir a CPU ao processo, o mesmo atinge o estado 2 onde completa a chamada de sistema e volta ao estado 1, executando no modo usuário.

Quando um `exit` é executado pelo processo, o mesmo transita, via estado 2, para seu estado terminal (9), permanecendo neste estado até que o processo pai seja notificado.

Algumas observações sobre o diagrama de transição de estados apresentado na Fig. 2.1:

- uma vez criado, as transições de estado de um processo dependem exclusivamente do sistema operacional;
- um processo pode forçar a entrada no modo núcleo através da execução de uma chamada de sistema, mas a saída deste estado foge ao seu controle;
- um processo pode atingir o estado 9 sem explicitamente evocar um `exit`: traps aritméticos como divisão por zero e *overflows*, ou de segmentação como referência a posições inválidas de memória, podem forçar compulsoriamente o término do processo.

Descritores de Processos

A tabela de processos e a área U descrevem o estado dos processos. A primeira é uma estrutura mantida pelo núcleo com os seguintes campos:

- o estado do processo;
- a localização da área U e do próprio processo, seja na memória primária ou secundária, bem como o tamanho do processo;
- o identificador do usuário (UID), isto é, o "dono" do processo;

- o identificador do processo (PID), único durante toda a vida do processo;
- eventos aguardados pelo processo quando no estado bloqueado (dormindo);
- parâmetros de escalonamento, utilizados para decidir quais processos transitarão entre estados de execução nos modos usuário e núcleo;
- sinais enviados ao processo, mas ainda não tratados;
- marcas de tempo como tempo total CPU, "despertadores" armados pelo processo, etc, além de recursos consumidos do núcleo (estes parâmetros são utilizados no cômputo da prioridade de escalonamento do processo).

A área U de um processo armazena as seguintes informações:

- um ponteiro de volta ao respectivo índice na tabela de processos;
- privilégios que o processo dispõe, de acordo com o seu UID;
- tempos de execução nos modos usuário e núcleo;
- ponteiros para os gerenciadores de sinais definidos pelo processo;
- o terminal de *login* associado ao processo, caso exista;
- um campo de erro, armazenando os erros gerados por chamadas de sistema;
- parâmetros de entrada/saída, tais como endereço de dados a serem transferidos, tamanho destes dados, destino, etc;
- o diretório corrente e o diretório raiz;
- descritores de arquivos abertos pelo processo;
- limites (tamanho máximo de pilha, arquivos, etc);
- permissões (modos de acesso atribuídos durante a criação de arquivos).

2.4 Escalonamento de Processos no Unix

O escalonamento de processos no UNIX segue um algoritmo que combina prioridades e Round Robin. É necessário enfatizar que tal algoritmo tem o objetivo de compartilhar a CPU de forma equânime entre múltiplos usuários (sendo portanto orientada ao *time-sharing*). Tal algoritmo é inadequado para aplicações que requeiram o cumprimento estrito de restrições temporais, como impostas por aquelas denominadas de tempo real estritas *(hard real-time)*.

O núcleo divide os processos segundo duas classes de prioridades:

- prioridades em modo núcleo (altas), referentes a processos bloqueados no estado 4 ou 5;
- prioridades em modo usuário (baixas), referentes a processos prontos no estado 7.

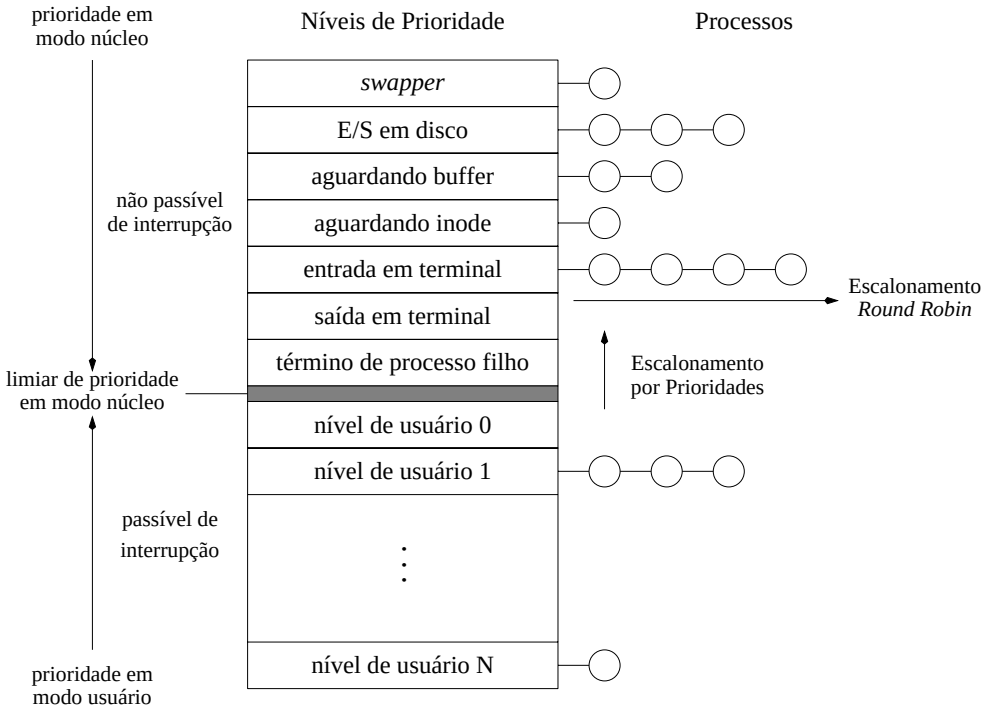

Fig. 2.2: Classes de prioridades para fins de escalonamento de processos

A Fig. 2.2 mostra as classes de prioridades adotadas. Prioridades em modo núcleo são subdivididas em dois grupos. O primeiro grupo, de elevada prioridade, é constituído de processos bloqueados a espera de *swapping*, E/S em disco, buffers de cache e *inodes*. Quando acordados, estes processos completam suas respectivas chamadas de sistema ininterruptamente (visando a rápida liberação de recursos do núcleo, tais como buffers).

O segundo grupo, de prioridade mais baixa que o primeiro, constitui-se de processos bloqueados a espera de entrada de terminal, saída em terminal e terminação de processo filho. Tais processos podem ser interrompidos, pois estes estados de bloqueio não demandam grandes recursos do núcleo.

Finalmente, processos aguardando apenas CPU (estado 7) são dispostos segundo um certo número de níveis de prioridade (este número é dependente da particular implementação).

Processos em uma mesma classe de prioridade são dispostos em uma fila, como representado na Fig. 2.2 pelos círculos interligados.

O algoritmo de escalonamento do UNIX é processado segundo o seguinte esquema. Quando ocorre uma interrupção do hardware:

- caso a interrupção habilite um ou mais processos com prioridade de modo núcleo, aloque a CPU àquele de mais alta prioridade;
- caso contrário, se uma mudança de contexto se fizer necessária, aloque a CPU ao processo de mais alta prioridade dentre as de modo usuário.

Em existindo mais de um processo apto a executar no mesmo nível, a seleção se dá segundo a política Round Robin.

Pode-se observar que o escalonamento de processos no UNIX se dá em duas direções: por prioridades na vertical, e por Round Robin na horizontal (para processos de mesma prioridade). Outras características importantes do algoritmo de escalonamento do UNIX:

1. As prioridades no modo núcleo dependem apenas do evento aguardado pelo processo.

2. Processos transitando do modo núcleo para o modo usuário, tem seu nível de prioridade rebaixado em relação à sua posição inicial. Esta penalidade, por utilizar recursos do núcleo, visa evitar o monopólio da CPU pelos processos que utilizam chamadas de sistema de forma frequente.

3. A cada intervalo de tempo (tipicamente um segundo), as prioridades em modo usuário são recomputadas. Processos que se utilizaram recentemente da CPU são penalizados em benefício dos processos que não a utilizaram.

2.5 Controle de Processos no UNIX

O controle de processos se divide em duas atividades: instanciação (criação) e interrupção de processos.

Instanciação de Processos

O mecanismo básico de criação de processos no UNIX é a chamada de sistema `fork`. A Fig. 2.3 ilustra o que ocorre quando um processo executa esta chamada de sistema. Um novo elemento na tabela de processos mantida pelo núcleo é criado; a área U é criada à imagem da área U do processo pai; uma tabela de região é criada com áreas de pilha e dados copiadas do processo pai, e área de texto compartilhada com o processo pai. A pilha mantida pelo núcleo é copiado também.

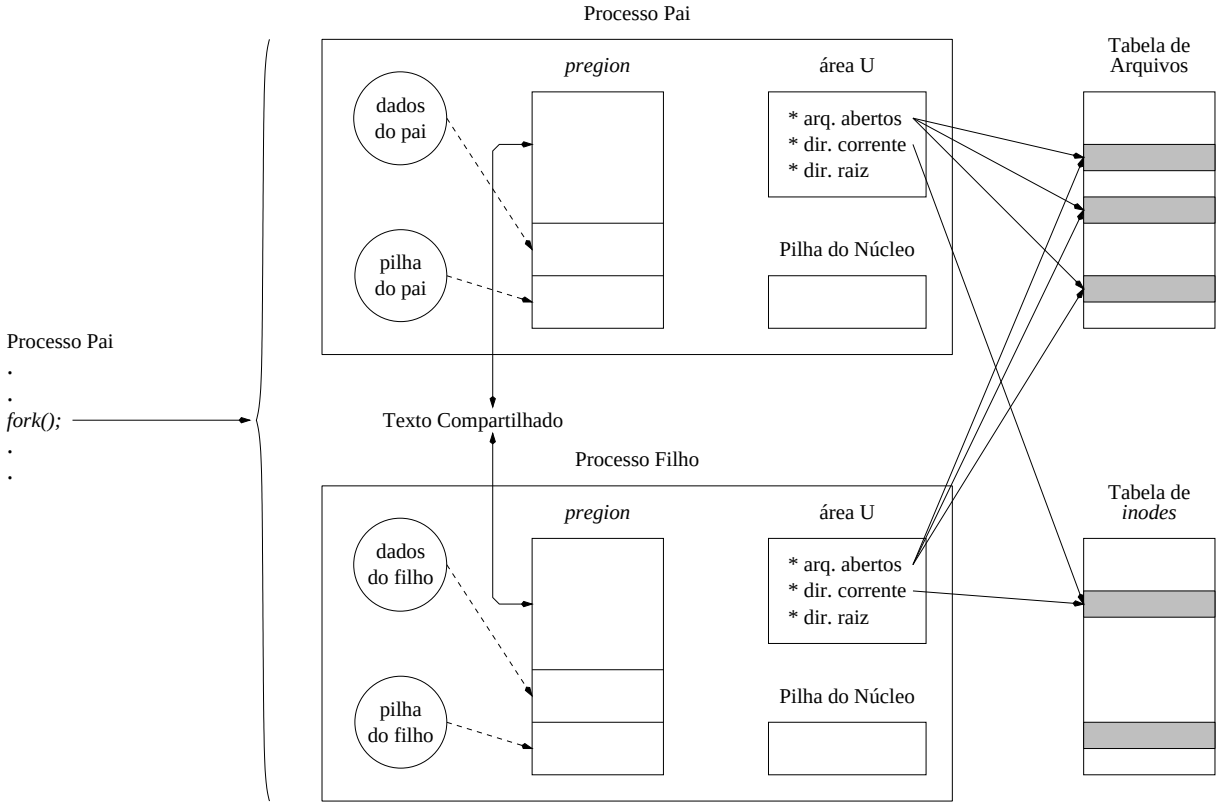

Fig. 2.3: A execução de uma chamada de sistema `fork`.

Como pode ser observado, a cópia da área U faz com que todos os descritores de arquivo permaneçam ativos para o processo filho. A cópia das regiões de dados e pilhas faz com que toda a história de execução do processo pai (valores de variáveis, retorno de funções, etc) seja herdada pelo processo filho.

Uma alternativa de instanciação de processos é a família `exec` de chamadas de sistema. Uma chamada `exec` utiliza os recursos do processo que a executou para instalar um novo processo. Diferente do `fork`, o processo que executa uma chamada `exec` deixa de existir.

A chamada `exec` tem como parâmetros o arquivo executável e dados a serem passados ao novo processo, sendo por este obtidos na função *main* através dos argumentos *argc* e *argv*.

Basicamente, uma chamada `exec` libera as regiões de memória relativas a texto, dados e pilha alocadas pelo processo executor da chamada, instalando novas regiões para o novo processo. O campo na tabela de processos mantida pelo núcleo e a região U permanecem inalterados. A pilha mantida pelo núcleo é refeita para o novo processo. Isto significa que a história do processo antigo é perdida, mas o PID e descritores de arquivo por ele abertos permanecem válidos[7].

Fica claro agora uma das utilidades da chamada `fork`: criar um novo processo sem que o processo que o criou tenha sua execução terminada. Exemplo:

```
/* ------------------------------------------------------------------ */

  int pid, fd1, fd2;
  char arg1[16], arg2[16];
  ...
  fd1 = open("arq1", O_RDONLY, 0);
  fd2 = open("arq2", O_RDONLY, 0);
  sprintf(arg1, "%d", fd1);
  sprintf(arg2, "%d", fd2);
  pid = fork();

  if(pid == 0)  {  /* processo filho */
    execlv("prog2", arg1, arg2, NULL);
    /* se passar por aqui, execv falhou */
    printf("execlv falhou !");
    exit(0);
  }

  /* processo pai continua */
  ...

/* ------------------------------------------------------------------ */
```

No exemplo acima, o programa executa um `fork`, deixando a cargo do filho a execução do novo processo. Os descritores de arquivo *fd1* e *fd2* são passados como argumento para o novo programa. Capturando estes argumentos, o novo programa é capaz de executar operações nos respectivos arquivos sem reabrí-los.

```
/* ------------------------------------------------------------------ */

/* prog1 */
main(int argc, char *argv[])
{
  int fd1, fd2;
```

[7] Entretanto, a única maneira do novo processo conhecer o valor destes descritores é através da passagem de parâmetros via *argc* e *argv*.

```
  /* acessa descritores abertos pelo executor do exec */
  sscanf(argv[1], "%d", &fd1);
  sscanf(argv[2], "%d", &fd2);
  ...
}

/* ____________________________________________________________ */
```

Interrupção de Processos

Processos no UNIX podem ter sua execução alterada assincronamente por ação de um outro processo (ou do usuário, através do *shell*). Esta ação é referida como o envio de um sinal. Sinais podem ser enviados a um processo para notificá-lo:

- de uma requisição de mudança de estado (ex: morte de um processo via chamada `kill`);
- do término do processo filho;
- da ocorrência de exceções (ex: trap aritmético);
- de situações irrecuperáveis (ex: recursos exauridos durante o processamento de um `exec`);
- da ocorrência de erros inesperados (ex: utilização de um *pipe* "quebrado");
- do disparo de "despertadores" de tempo (ex: retorno da chamada `sleep`);
- de interrupções oriundas de terminais (ex: Ctrl-C);
- de interrupções para fins de depuração (ex: ocorrência de um *breakpoint*).

Sinais são explicitamente enviados a processos através da chamada de sistema `kill`. `kill` tem como parâmetros o PID do processo ao qual o sinal se destina; e o tipo de sinal. Existem em torno de 30 tipos de sinais no UNIX System V, sendo os mais importantes mostrados na Tab. 2.1.

sinal	**significado**
SIGHUP	hang-up
SIGINT	interrupção
SIGILL	instrução ilegal (trap)
SIGFPE	exceção aritmética (trap)
SIGKILL	término forçado
SIGSEGV	violação de segmentação (trap)
SIGSYS	argumento inválido em chamada de sistema
SIGALRM	alarme de relógio
SIGSTOP	suspensão da execução
SIGCONT	continuação da execução
SIGCHLD	mudança de status de processo filho

Tab. 2.1: Exemplos de sinais no UNIX System V

Processos podem apresentar as seguintes reações a sinais:

- o processo é compulsoriamente terminado;
- o processo é bloqueado até a ocorrência de um sinal de desbloqueio;
- o processo ignora o sinal;
- o processo "captura" o sinal.

Sinais são armazenados na tabela de processos mantida pelo núcleo, sendo a ação correspondente ao sinal tomada quando o processo passa da execução de modo núcleo para modo usuário.

Para cada sinal é definido um comportamento *default* para o processo que o recebe. Em geral, este comportamento é simplesmente o término do processo. Um processo pode alterar o comportamento *default* frente a ocorrência de um dado sinal através da chamada de sistema `signal`. `signal` possui dois parâmetros: o número do sinal (definido em *signal.h*); e a ação a ser executada quando do recebimento deste sinal. A ação pode possuir três valores:

- 0: o processo é terminado quando receber o sinal;
- 1: o sinal é ignorado;
- endereço válido de função: a função é chamada assincronamente quando da ocorrência do sinal. Esta função é denominada *gerenciador do sinal* para o processo.

Exemplo: capturar interrupções de teclado. Quando o usuário executa um Ctrl-C, um sinal do tipo SIGINT é enviado ao processo que está executando em *foreground*. O comportamento *default* para este sinal é o término do processo. Suponha um programador precavido que deseje a confirmação que o Ctrl-C não foi acidental. O código abaixo ilustra esta situação:

/* ——————————————————————————————— */

```
/* gerenciador para SIGINT */
int ger()
{
  int c;
  printf("Tem certeza que quer terminar [s/n] ? ");
  c = getchar();
  if(c == 's') exit(0);
}


main()
{
  signal(SIGINT, ger);
  ...
}
```

/* ——————————————————————————————— */

Todas as vezes que um SIGINT for enviado ao processo, a função "ger" é chamada assincronamente, solicitando confirmação do término da execução. Caso o usuário responda "n", a função retorna e o programa continua normalmente.

A maneira como o núcleo processa sinais é descrita sucintamente a seguir. Quando um sinal é enviado a um processo (pelo núcleo ou por outro processo), o núcleo simplesmente ativa o

campo correspondente ao sinal na tabela de processos. Neste campo está localizado também o gerenciador do sinal (definido pelo processo ou *default*).

No momento que um processo passa do modo núcleo para o modo usuário, o núcleo verifica se existe sinais enviados ao processo e ainda não tratados. Caso exista, o núcleo executa os seguintes passos:

1. Salva o contador de programa e o *stack pointer* do processo.
2. Caso o processo não defina um gerenciador para o sinal, executa a ação *default*. Caso contrário, prossiga.
3. Associa temporariamente a ação *default* para o sinal.
4. Cria um novo quadro na pilha como se o processo estivesse evocando o gerenciador neste momento.
5. Direciona o contador de programa para o endereço da rotina gerenciadora do sinal e atualiza o *stack pointer* para levar em conta o aumento da pilha causado pela chamada do gerenciador.

O passo 3 merece um comentário adicional. Ele existe para evitar que uma "rajada" de sinais ocasione um *stack overflow* pelo empilhamento de múltiplas chamadas do gerenciador (caso o intervalo de ocorrência dos sinais seja menor que o tempo de execução do gerenciador). Entretanto, durante a execução da rotina gerenciadora, o processo fica em uma condição vulnerável, pois a ação *default* é que será executada face a ocorrência de um novo sinal de mesmo tipo.

2.6 Comunicação e Sincronização Inter-processos no UNIX

Nesta seção descreveremos três mecanismos de comunicação inter-processos (*pipes*, mensagens e memória compartilhada), e um de sincronização inter-processos (semáforos). *Pipes* são disponíveis em qualquer versão de UNIX, enquanto os demais estão presentes apenas nas versões compatíveis com o System V.

Pipes

O mecanismos originais de comunicação inter-processos são os chamados *pipes*. Em geral, *pipes* são empregados para estabelecer comunicação entre processos pai e filho. Um *pipe* é um canal unidirecional de comunicação, isto é, a informação flui em uma única direção. Para estabelecer-se comunicação bidirecional, são necessários dois *pipes*.

Pipes podem ser vistos como um buffer conectando dois processos. Um processo escreve dados em um dos lados do buffer, enquanto o outro lê estes dados no lado oposto. Esta forma de comunicação pode ser obtida com o emprego de arquivos em disco. A diferença básica é que *pipes* são bloqueantes, isto é, a gravação em um *pipe* cheio ou a leitura em um *pipe* vazio causa o bloqueio do processo.

Pipes são implementados pelo núcleo como um sistema de arquivos. Cada *pipe* tem um *inode* associado, sendo que o núcleo aloca blocos de dados à medida que dados vão sendo escritos no *pipe* (desalocando-os à medida que são lidos).

Pipes são criados com a chamada de sistema `pipe`. A chamada retorna dois descritores de arquivos, sendo o primeiro para leitura e o segundo para gravação. Esta chamada, em geral, se processa antes de ocorrer um `fork`. Se a comunicação for no sentido pai → filho, o processo pai fecha o primeiro descritor (com a chamada `close`), e o filho o segundo. A partir daí, o pai executa chamadas `write` no segundo descritor e o filho `read` no primeiro. Um segundo *pipe* pode ser empregado para a comunicação no sentido inverso, atentando-se para o fechamento correto dos descritores que não serão empregados pelo processo.

Após o final da sessão de comunicação, os lados abertos do *pipe* também são fechados a fim de liberar os recursos a ele associados pelo núcleo.

Exemplo: enviar um string do processo pai para o processo filho.

```
/* ------------------------------------------------------------------ */

main()
{
  int fd[2];
  char buff[32];

  if(pipe(fd) == -1) {perror("pipe"); exit(0);}

  if(fork() != 0)  {   /* PAI */
    close(fd[0]);
    strcpy(buff, "oi filho !");
    write(fd[1], buff, strlen(buff) + 1);
    close(fd[1]);
    exit(0);
  }
  lse  {  /* FILHO */
    close(fd[1]);
    read(fd[0], buff, 32);
    printf("%s", buff);
    close(fd[0]);
    exit(0);
  }

/* ------------------------------------------------------------------ */
```

Mensagens

Troca de mensagens é um mecanismo de comunicação mais flexível que *pipes*. Enquanto um *pipe* é um canal síncrono e unidirecional, mensagens são canais tanto síncronos quanto assíncronos e bidirecionais.

Para o envio e recepção de mensagens, cria-se um *port* de comunicação. *Ports* são identificados por um número inteiro. A chamada de sistema `msgget` cria um *port*, retornando seu identificador local. O primeiro parâmetro é uma chave atribuída ao *port*, seu identificador global. O segundo parâmetro são opções relativas a criação, acesso, etc. Via de regra, `msgget` retorna um *port* dado seu identificador global, criando-o caso tal *port* inexista.

O núcleo mantém uma tabela de *ports*, e mensagens enviadas a *ports* são enfileiradas, sendo recebidas na ordem que foram enviadas. A comunicação é bidirecional, posto que, de posse de um identificador local de *port*, um processo pode tanto enviar quanto receber mensagens neste port.

Mensagens são enviadas com a chamada de sistema `msgsnd`. A chamada possui quatro parâmetros: o identificador do *port* para o qual a mensagem deve ser enviada; a estrutura que

contém a mensagem; seu tamanho em bytes; e a opção de envio assíncrono (retornando um código de erro caso não exista espaço para o núcleo armazenar a mensagem no *port*). A opção *default* é o envio síncrono, onde o processo bloqueia ante a impossibilidade de armazenamento da mensagem no *port*.

Estruturas contendo mensagens devem ser definidas como *structs* contendo dois campos: um inteiro (tipo da mensagem); e uma cadeia de bytes (o conteúdo da mensagem). Exemplo:

```
struct mensagem {
  int tipo;    /* tipo da mensagem */
  char conteudo[1024];  /* conteudo da mensagem (1K max.) */
};
```

A recepção de mensagens se dá através da chamada de sistema `msgrcv`. Cinco parâmetros são necessários: o identificador local do *port*; a estrutura onde a mensagem será copiada; o tamanho máximo em bytes alocados pela estrutura; o tipo de mensagem desejada; e opção de recebimento assíncrono (retornando um código de erro caso o *port* não contenha mensagem do tipo especificado).

Uma quarta chamada de sistema, `msgctl`, é empregada para obter e alterar o status de *ports*. Possui três parâmetros: o identificador local do *port*; o tipo de operação e o endereço da estrutura com as informações de entrada ou retorno, de acordo com a ação explicitada no segundo parâmetro.

Memória Compartilhada

Uma outra forma de comunicação disponível no UNIX é o compartilhamento de um espaço virtual. Este espaço é criado via chamada de sistema `shmget`. `shmget` possui três parâmetros: um identificador global da região compartilhada, o tamanho da região em bytes; e opções de controle (criação e acesso). Como `msgget`, esta chamada retorna um identificador local da região, criando-a caso inexista.

Uma vez acessada, um processo associa a região compartilhada em seu próprio espaço de endereçamento. Para tal, utiliza-se a chamada de sistema `shmat` que possui três parâmetros: o identificador local da região compartilhada, o endereço local que apontará para a região; e parâmetros de controle (se a região deve ser considerada de leitura apenas, por exemplo).

O procedimento inverso, isto é, a desassociação de uma região compartilhada de um endereçamento local se dá com a chamada de sistema `shmdt`. Esta chamada possui um único parâmetro: o endereço local previamente associado a uma região compartilhada.

Finalmente, a chamada `shmctl` é empregada para obter e alterar o status de uma região compartilhada (permissões, desativação, etc). É similar a `msgctl`.

Deve-se observar que o mecanismo de memória compartilhada não necessita operações especiais para a manipulação de dados. Como a região é mapeada em um endereço local, leituras e escritas neste endereço se processam com os comandos de associação e cópia binária providos pelas linguagem C.

Deve-se observar ainda, que este mecanismo de comunicação é sempre assíncrono, e pode suportar comunicação do tipo "de-um-para-muitos" (um processo escreve na memória compartilhada e vários outros lêem).

Memória compartilhada é implementada pelo núcleo através de uma tabela de regiões alocadas por `shmget`. Esta tabela aponta para a tabela de regiões de memória mantida pelo núcleo. Após a chamada `shmat`, a tabela de regiões do processo aponta para a respectiva região mantida pelo núcleo (Fig. 2.4).

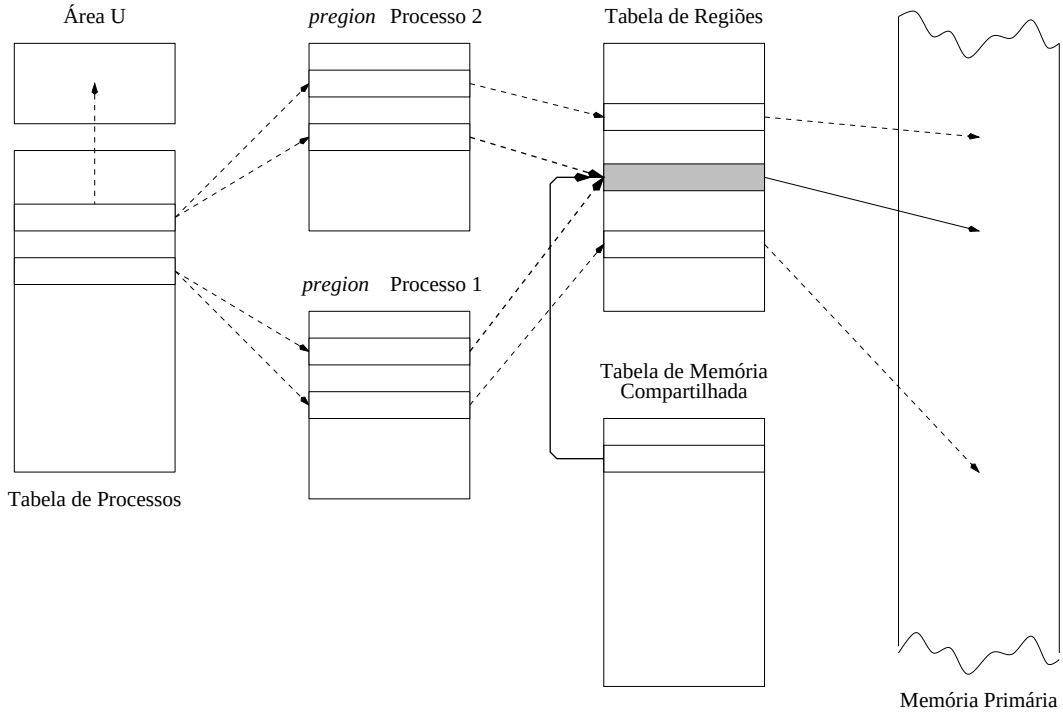

Fig. 2.4: Esquema de memória compartilhada

O código abaixo aloca um texto em uma região de memória compartilhada.

```
/* ---------------------------------------------------------------------- */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define KEY 67
extern char *shmat();

main()
{
  char *buff;
  char *poema[16];
  int i, sh;

  /* cria area compartilhada */
  sh = shmget(KEY, 1024, 0777 | IPC_CREAT);

  /* assoacia area compartilhada a um endereco local */
  buff = shmat(sh, 0, 0);

  poema[0] = "As armas e os baroes assinalados";
  poema[1] = "Que da ocidental praia Lusitana";
  poema[2] = "Por mares nunca dantes navegados";
  poema[3] = "Passaram ainda alem de Tapobrana";
  poema[4] = "E, em perigos e guerras esforcados";
  poema[5] = "Mais do que prometia a forca humana";
  poema[6] = "Por gente remota edificaram";
 poema[7] = "Novo reino, que tanto sublimaram";
```

```
  /* armazena o texto na area compartilhada */
  for(i = 0; i < 8; i++) strcpy((buff + i * 100), poema[i]);
}
```

Agora, de um outro processo, podemos acessar o texto e imprimí-lo.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define KEY 67
extern char *shmat();

main()
{
  char *buff;
  char *poema[16];
  int i, sh;

  /* acessa area compartilhada */
  sh = shmget(KEY, 1024, 0777);
  if(sh < 0)  {
    printf("\nArea compartilhada nao criada\n");
    exit(0);
  }

  /* assoacia area compartilhada a um endereco local */
  buff = shmat(sh, 0, 0);

  /* acessa texto da area compartilhada */
  for(i = 0; i < 8; i++) poema[i] = (buff + i * 100);

  for(i = 0; i < 8; i++)  printf("\n%s", poema[i]);
  printf("\n\n");
}
```

/* —— */

Semáforos

Semáforo é um mecanismo de sincronização inter-processos composto das operações DOWN e UP. Via de regra, define-se um semáforo para cada recurso compartilhado. A execução da parte do código que acessa tais recursos (região crítica) é abraçada pelas operações DOWN e UP.

Processos definem/acessam semáforos com a chamada de sistema `semget`. Esta chamada requer três parâmetros: um identificador global para um *array* de semáforos; o número de semáforos contido no array; e um flag estipulando ações relativas a permissões e criação.

Operações em semáforos se processam através da chamada de sistema `semop`. Esta chamada requer três parâmetros: o identificador local do semáforo (obtido via `semget`); um array de estruturas *sembuf*; e o número de estruturas no array. Cada estrutura efetua uma operação no semáforo de índice estipulado na estrutura.

```
struct sembuf {
  short  sem_num;   /* indice do semaforo */
  short  sem_op;    /* operacao requisitada */
  short  sem_flag;  /* controle da operacao */
};
```

O segundo campo da estrutura *sembuf* opera no respectivo semáforo de acordo com a seguinte lógica. Se *sem_ op* for:

- negativo: caso a soma de *sem_ op* com o valor do semáforo seja não negativa, some *sem_ op* ao valor do semáforo e retorne. Caso seja negativa, bloqueie.
- positivo: some *sem_ op* ao valor do semáforo e retorne.
- nulo: retorne se o valor do semáforo for nulo; bloqueie, caso contrário.

É fácil notar que fazendo *sem_ op* -1 ou $+1$ implementa-se, respectivamente, as operações básicas DOWN e UP.

Similar a `msgctl` e `shmctl`, a chamada de sistema `semctl` é empregada para obter e alterar o status de semáforos (permissões, desativação, etc).

O código abaixo ilustra o uso de semáforo para acesso a uma região crítica (no caso, o vídeo).

```
/* ____________________________________________________________ */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

/* cria um semaforo */
int DefSem(key_t key)
{
  int semid;
  union semun arg;

  semid = semget(key, 1, 0777 | IPC_CREAT);
  arg.val = 1;
  semctl(semid, 0, SETVAL, arg);
  return(semid);
}


/* acessa um semaforo ja criado */
int GetSem(key_t key)
{
  int semid;
  semid = semget(key, 1, 0777);
  return(semid);
}


/* define operacao DOWN */
void DOWN(int semid)
{
  struct sembuf psem[2];
  psem[0].sem_num = 0;
```

```
  psem[0].sem_op = -1;
  psem[0].sem_flg = SEM_UNDO;
  semop(semid, psem, 1);
}


/* define operacao UP */
void UP(int semid)
{
  struct sembuf vsem[2];
  vsem[0].sem_num = 0;
  vsem[0].sem_op = 1;
  vsem[0].sem_flg = SEM_UNDO;
  semop(semid, vsem, 1);
}

main()
{
  char *poema[16];
  int i;
  int semid;

  semid = GetSem(13);

  if(semid < 0)  {
    printf("\nSemaforo nao criado !\n");
    exit(0);
  }

  poema[0] = "As armas e os baroes assinalados";
  poema[1] = "Que da ocidental praia Lusitana";
  poema[2] = "Por mares nunca dantes navegados";
  poema[3] = "Passaram ainda alem de Tapobrana";
  poema[4] = "E, em perigos e guerras esforcados";
  poema[5] = "Mais do que prometia a forca humana";
  poema[6] = "Por gente remota edificaram";
  poema[7] = "Novo reino, que tanto sublimaram";

  while(1)  {
    DOWN(semid);    /* entrada na Regiao Critica */
    for(i = 0; i < 8; i++)  {
      printf("\n%s", poema[i]);
      sleep(1);
    }
    printf("\n\n");
    UP(semid);   /* saida da Regiao Critica */
  }
}
```

/* ——— */

Capítulo 3

Sistema de Arquivos

A parte mais visível de um sistema operacional é o seu sistema de arquivos. Programas aplicativos utilizam o sistema de arquivos (via chamadas de sistema) para criar, ler, gravar e remover arquivos. Usuários utilizam interativamente o sistema de arquivos (via *shell*) para listar, alterar propriedades e remover arquivos. A conveniência e facilidade de uso de um sistema operacional é fortemente determinada pela interface, estrutura e confiabilidade de seu sistema de arquivos [1].

3.1 Interface do Sistema de Arquivos

Do ponto de vista do usuário, o aspecto mais importante do sistema de arquivos é como este se apresenta, isto é, o que constitui um arquivo, como os arquivos são identificados e protegidos, que operações são permitidas sobre os arquivos, e assim por diante.

Fundamentos Básicos

A maior parte dos sistemas operacionais trazem a seguinte proposta para armazenamento de informação: permitir aos usuários definir objetos chamados *arquivos*, que podem armazenar programas, dados, ou qualquer outra informação. Estes arquivos não são parte endereçável de nenhum processo e o sistema operacional provê chamadas de sistema para criar, destruir, ler, atualizar e proteger arquivos.

Todos os sistemas operacionais visam uma independência dos dispositivos de armazenamento, permitindo acessar um arquivo sem especificar em qual dispositivo o mesmo se encontra fisicamente armazenado. Um programa que lê um arquivo de entrada e escreve um arquivo saída deve ser capaz de operar com arquivos armazenados em quaisquer dispositivos, sem necessidade de um código especial para explicitar o tipo de periférico.

Alguns sistemas operacionais provêem maior independência dos dispositivos de armazenamento que outros. No UNIX, por exemplo, um sistema de arquivos pode ser montado em qualquer dispositivo de armazenamento, permitindo que qualquer arquivo seja acessado pelo seu nome (*path name*), sem considerar o dispositivo físico. No MS-DOS, por outro lado, o usuário deve especificar em qual dispositivo cada arquivo se encontra (exceto quando um dispositivo é *default* e for omitido). Assim, se o dispositivo *default* for o *drive* C, para executar um programa localizado no *drive* A com arquivos de entrada e saída no *drive* B, cada um deles deverá ser especificado juntamente com o nome do arquivo:

```
A:programa < B:entrada > B:saida
```

A maior parte dos sistemas operacionais suportam vários tipos de arquivos. O UNIX, por exemplo, mantém arquivos regulares, diretórios e arquivos especiais. Arquivos regulares contém dados e programas do usuário. Diretórios permitem identificar arquivos através de nomes simbólicos (i.e. sequência de caracteres ASCII). Arquivos especiais são usados para especificar periféricos tais como terminais, impressoras, unidades de fita, etc. Assim podemos para copiar um arquivo `abc` para o terminal (arquivo especial `/dev/tty`) através do comando:

```
cp abc /dev/tty
```

Em muitos sistemas, arquivos regulares são subdivididos em diferentes tipos em função de sua utilização. Os tipos são identificados pelos nomes com que os arquivos regulares terminam. Por exemplo,

`arquivo.c` - Arquivo fonte em C
`arquivo.obj` - Arquivo objeto
`arquivo.bin` - Programa binário executável
`arquivo.lib` - Biblioteca de arquivos .OBJ usados pelo *linker*

Em certos sistemas, as extensões são simples convenções: o sistema operacional não faz uso delas. Em outros, o sistema operacional tem regras rígidas em relação aos nomes. Por exemplo, o sistema não executará um arquivo a menos que sua extensão seja .BIN.

Diretórios

Para organizar os arquivos, o sistema de arquivos provê *diretórios*, os quais em muitos casos são também arquivos. Um diretório contém tipicamente um número de registros, um por arquivo. Sistemas primitivos admitiam um único diretório compartilhado por todos os usuários, ou um único diretório por usuário. Os sistemas operacionais modernos permitem um número arbitrário de diretórios por usuário (via de regra, formando uma hierarquia). A Fig. 3.1 ilustra estas três situações.

Quando o sistema de arquivos é organizado como uma árvore de diretórios, algum meio se faz necessário para especificar nomes de arquivos. Dois métodos são comumente empregados. No primeiro método, cada arquivo é identificado pela sequência de diretórios desde o diretório raiz até o arquivo (caminho absoluto). Como um exemplo, o caminho `/usr/mfm/mailbox` significa que o diretório raiz (/) contém o subdiretório `usr`, o qual contém o subdiretório `mfm`, que por sua vez contém o arquivo `mailbox`. Nomes absolutos para caminhos sempre começam na raiz e são únicos.

Uma outra forma de especificar nomes de arquivos é através de seu caminho relativo. É usado em conjunto com o conceito de diretório de trabalho (ou diretório corrente). Um usuário pode designar um diretório como diretório corrente. Neste caso, todos os caminhos são referenciados a partir do diretório corrente. Se o diretório corrente for é `/usr/mfm`, então o arquivo cujo caminho absoluto é `/usr/mfm/mailbox` pode ser referenciado simplesmente como `mailbox`. Em UNIX o diretório corrente é denotado por `.` (ponto).

3.2 Projeto do Sistema de Arquivos

Examinaremos agora o sistema de arquivos do ponto de vista do projetista de sistemas operacionais. Aos usuários interessa como os arquivos são identificados, quais operações são

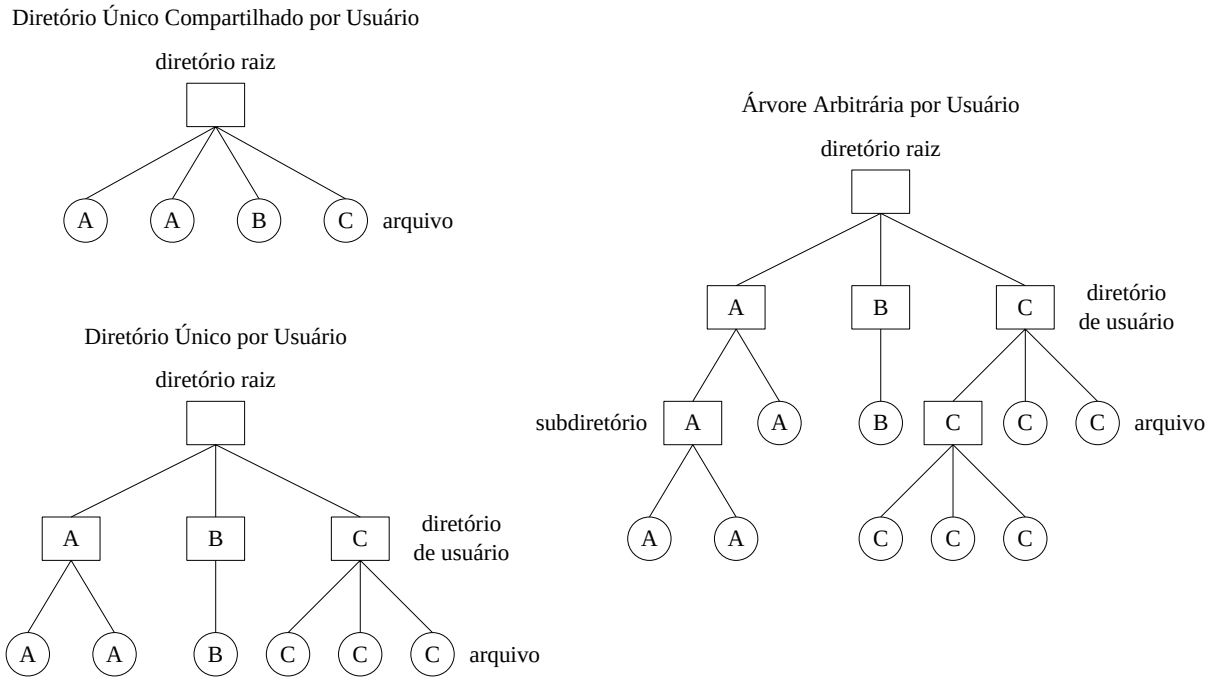

Fig. 3.1: Três projetos de sistemas de arquivos: (a) diretório único compartilhado pelos usuários; (b) um diretório por usuário; (c) árvore arbitrária por usuário

permitidas, como os diretórios são organizados, etc. Projetistas estão interessados em como o espaço de disco é gerenciado, como os arquivos são armazenados, e como manipular arquivos de forma eficientemente e confiável.

Gerenciamento de Espaço em Disco

Arquivos são normalmente armazenados em disco, sendo portanto o gerenciamento do espaço em disco de maior interesse do projetista. Duas estratégias são possíveis para armazenamento em um arquivo com n bytes: n bytes consecutivos do disco são alocados; ou o arquivo é dividido em um número de blocos não necessariamente contíguos. A mesma política está presente no sistema de gerenciamento de memória entre a segmentação pura e a paginação.

Armazenar um arquivo como uma sequência contígua de bytes apresenta um problema óbvio que é o crescimento do arquivo, uma ocorrência muito comum. O arquivo provavelmente terá que ser movido no disco. O mesmo problema é apresentado para segmentação na memória, exceto que mover um segmento na memória é uma operação relativamente mais rápida. Por esta razão, normalmente todos os sistemas de arquivos armazenam os arquivos em blocos de tamanho fixo, que não precisam ser adjacentes[1].

Uma vez decidido armazenar arquivos em blocos de tamanho fixo, a questão é definir qual o tamanho do bloco a ser usado. Dado a forma como os discos são organizados, os setores, as trilhas e os cilindros são candidatos óbvios para a unidade de alocação.

Uma unidade de alocação grande, tal como um cilindro, implica que muitos arquivos, até mesmo arquivos de 1 byte, deverão ocupar o cilindro inteiro. Por outro lado, usar uma unidade de alocação pequena, significa que cada arquivo terá muitos blocos. A leitura de cada

[1]Salvo alguns sistemas operacionais, notadamente os voltados à computação de tempo-real, onde o armazenamento contínuo é adotado por razões de desempenho.

bloco normalmente requer uma busca e uma latência rotacional. Assim, a leitura de arquivos consistindo de muitos blocos pequenos será lenta.

É compromisso usual escolher um bloco de tamanho 512, 1K ou 2K bytes. Se um bloco de tamanho 1K for escolhido em um disco com setor de 512 bytes, então o sistema de arquivo sempre irá ler ou escrever em dois setores consecutivos, e tratá-los como uma unidade indivisível.

Uma vez escolhido o tamanho do bloco, a próxima questão é como manter o rastreamento de blocos livres no disco. Dois métodos são largamente usados (Fig. 3.2). O primeiro consiste no uso de uma lista ligada de blocos, com cada elemento da lista armazenando tantos blocos livres quanto possível. Com elementos de 1K e o número do bloco de 16 bits, cada elemento na lista de blocos livre armazena 511 blocos livres. Um disco com 20 Gigabytes necessita de uma lista ocupando aproximadamente 40K blocos para apontar para todos os 20G blocos do disco (ou seja, a lista ocupa 0,2% do disco).

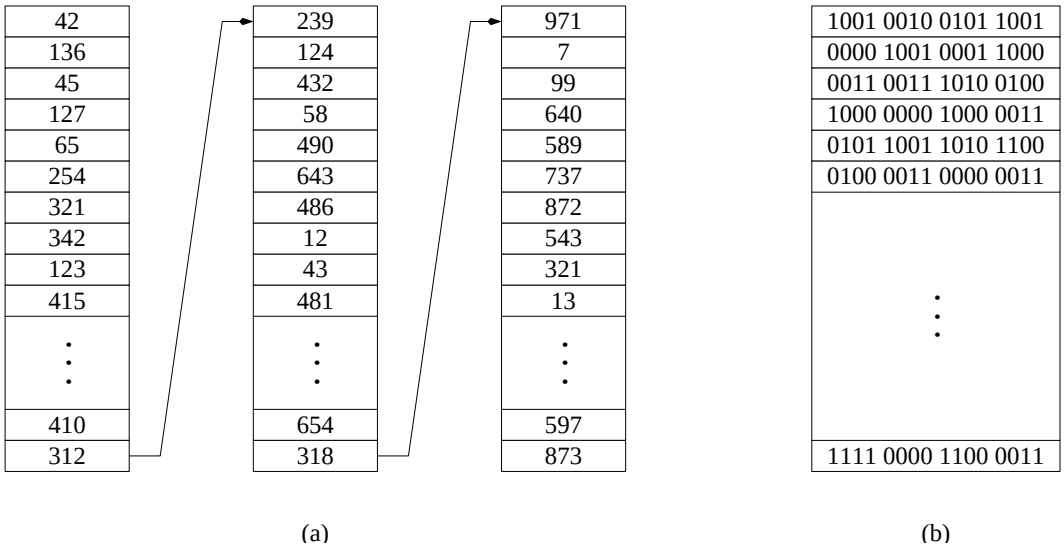

Fig. 3.2: (a) blocos livres armazenados em lista ligada; (b) um mapa de bits.

Uma outra técnica de gerenciamento de espaço livre é o mapa de bits. Um disco com n blocos necessita de um mapa de bits com n bits. Blocos livres são representados por 1s no mapa de bits; blocos alocados por 0s (ou vice-versa). Um disco com 20 Gigabytes necessita de 20M bits para o mapa, volume equivalente a 2500 blocos (ou seja, o mapa ocupa apenas 0,013% do disco). Não é surpresa que um mapa de bit necessite de menos espaço, desde que usa um bit por bloco, versus 16 bits da lista ligada. Entretanto, para um disco cheio (com poucos blocos livres) a lista ligada necessita de menos espaço que o mapa de bits.

Armazenamento de Arquivos

Se um arquivo consistir de uma sequência de blocos, o sistema de arquivos deve ter uma maneira de acessar todos os blocos do arquivo. Como visto acima, um método possível consiste em armazenar os blocos de um arquivo em uma lista ligada. Cada bloco de disco de 1024 bytes, contém 1022 bytes de dados e um ponteiro de 2 bytes para o próximo elemento da lista. Esse método tem duas desvantagens, entretanto. Primeiro, o número de bytes de dados em um elemento da lista não é uma potência de 2, o que frequentemente é uma desvantagem para sua manipulação. Segundo, e mais sério, o acesso aleatório é de difícil implementação. Se um programa busca o byte 32768 de um arquivo, o sistema operacional tem que percorrer 32768/1022

ou 33 blocos para encontrar o dado. Ler 33 blocos para buscar um dado, é inaceitável.

Entretanto, a idéia de representar um arquivo como uma lista encadeada, pode ser ainda explorada se mantermos os ponteiros em memória. A Fig. 3.3 mostra o esquema de alocação usado pelo MS-DOS. Neste exemplo, temos três arquivos, A, com os blocos 6,8,4 e 2; B, com os blocos 5, 9 e 12; e C, com os blocos 10, 3 e 13.

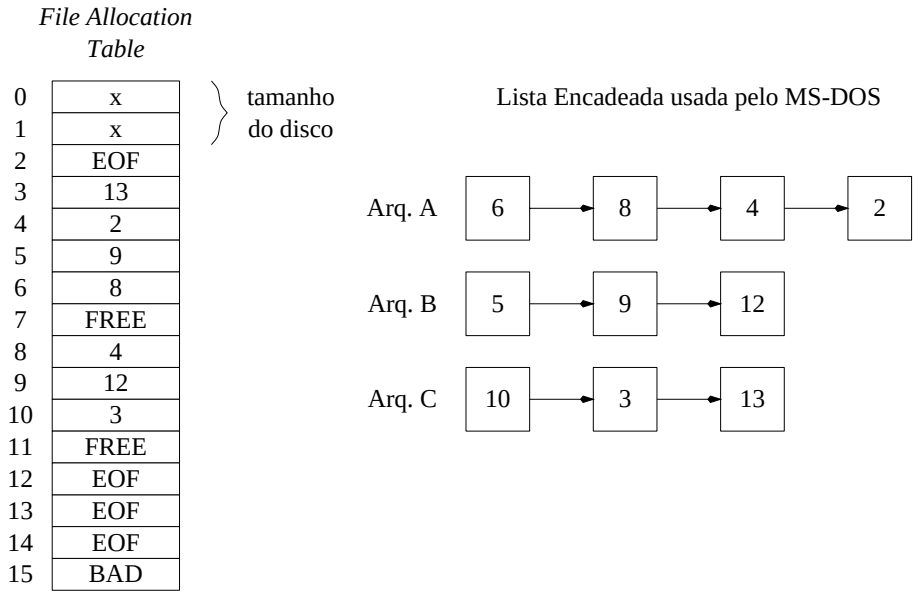

Fig. 3.3: Esquema de lista encadeada usado pelo MS-DOS. Os registros 0 e 1 são usadas para especificação do tipo do disco. Os códigos EOF e FREE são usados para *End Of File* e registros livres, respectivamente.

Associada a cada disco, existe uma tabela chamada Tabela de Alocação de Arquivos *(File Allocation Table–FAT)* que contém um registro para cada bloco do disco. O registro no diretório para cada arquivo fornece o endereço inicial do arquivo na FAT. Cada unidade da FAT contém o número do próximo bloco do arquivo. O arquivo A começa no bloco 6, então o registro 6 da FAT contém o endereço do próximo bloco do arquivo A, que é 8. O registro 8 da FAT contém o número do próximo bloco que é o 4. O registro 4 aponta para o registro 2, que está marcado como fim do arquivo.

Este esquema vai se tornando ineficiente a medida que a capacidade do disco aumenta. Para limitar o tamanho da tabela, deve-se aumentar o tamanho do bloco. Suponha um disco de 2 Gigabytes que contém 16K blocos de 32K, resultando em uma FAT com 16K entradas de 2 bytes cada. Dois problemas são intrínsecos deste esquema:

1. dado que mais que um arquivo não pode ocupar o mesmo bloco, um arquivo de 1 byte é armazenado em um bloco de 32 Kbytes;

2. por razões de eficiência, toda a FAT deve estar presente integralmente em memória, independentemente do número de arquivos abertos.

Um método mais eficaz, seria manter listas dos blocos para diferentes arquivos em lugares diferentes. Isto é exatamente o que o UNIX faz.

Associado a cada arquivo no UNIX, tem-se uma pequena tabela (no disco), chamada *inode*, como mostrado na Fig. 3.4. Ela contém informações sobre o arquivo tais como tamanho e

proteção. Os itens chaves são os 10 números de blocos do disco e os 3 números de blocos indiretos. Para arquivos com 10 blocos ou menos, todos os endereços dos blocos de dados no disco são mantidos no próprio *inode*, sendo sua localização imediata.

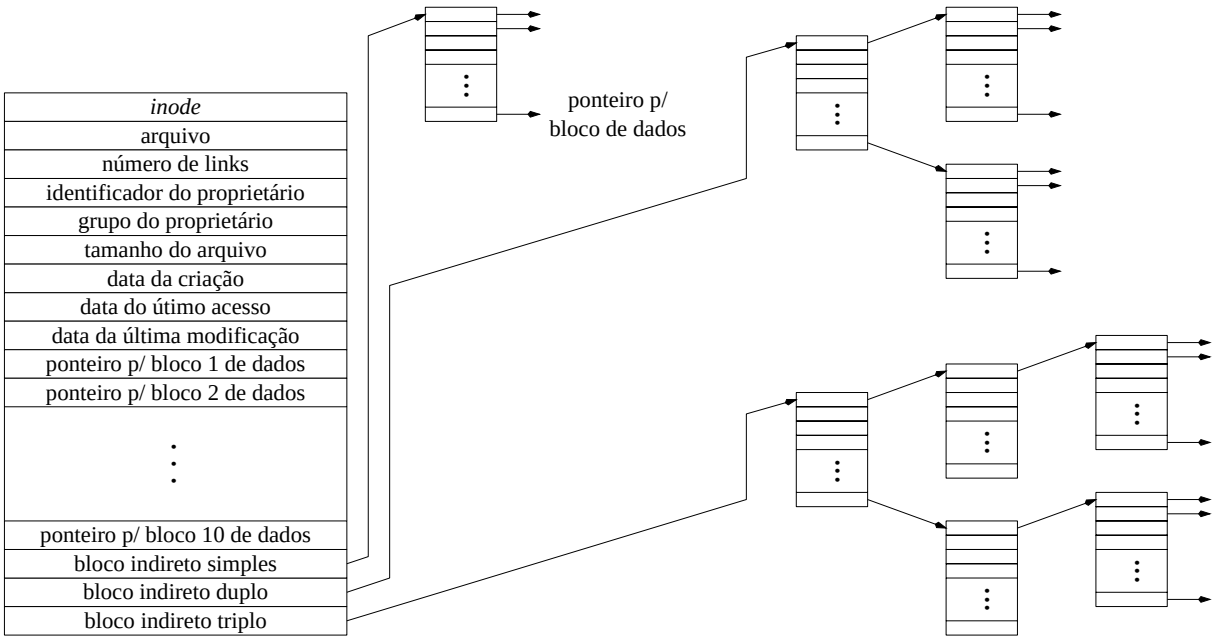

Fig. 3.4: Estrutura do *inode*

Quando o tamanho de um arquivo supera 10 blocos, um bloco livre é adquirido e o ponteiro indireto simples passa a apontar para ele. Este bloco é usado para armazenar os ponteiros dos blocos de disco. Com um bloco de disco de 1K e endereços de disco de 32 bits, o bloco indireto simples pode acessar 256 endereços de disco. Esse esquema é suficiente para arquivos de até 266 blocos (10 no *inode*, 256 no bloco indireto simples).

Acima de 266 blocos, o ponteiro indireto duplo é usado para apontar para um bloco de disco de até 256 ponteiros, que não apontam para blocos de dados, mas sim para 256 blocos indiretos simples. O bloco indireto duplo é suficiente para arquivos de até $266 + 256^2 = 65802$ blocos. Para arquivos maiores que 64K bytes, o ponteiro indireto triplo é usado para apontar para um bloco que contém ponteiros para 256 blocos indiretos duplos, permitindo arquivos de até 16 Gigabytes.

Arquivos maiores que 16 Gigabytes não podem ser manuseados por este esquema. Entretanto, aumentando o tamanho do bloco para 2K, cada bloco de ponteiro acessa 512 ponteiros ao invés de 256, e o tamanho máximo de um arquivo se torna 128 Gigabytes. O ponto forte do esquema do UNIX é que os blocos indiretos são usados somente quando for necessário. Para blocos de 1K bytes, arquivos com menos de 10K bytes não necessitam blocos indiretos. Note que mesmo para os arquivos mais longos são necessários, no máximo, 3 acessos a disco para localizar o endereço de um arquivo (descontado o acesso ao *inode*).

Estrutura de Diretório

Antes de um arquivo ser manipulado, ele deve ser aberto. Quando um arquivo é aberto, o sistema operacional usa o nome de caminho fornecido pelo usuário para localizar os blocos no disco. Mapeando nomes de caminhos em *inodes* (ou equivalentes), introduz-se ao tópico de como

sistemas de diretórios são organizados. Estes variam desde soluções simples até razoavelmente sofisticadas.

Vamos começar com um sistema de diretório particularmente simples, aquele do CP/M, ilustrado na Fig. 3.5(a). Neste sistema, existe apenas um diretório. Assim, a localização de um arquivo reduz-se a procura em um único diretório. Encontrado o registro do arquivo, tem-se o número de blocos do disco, posto que estes são armazenados no próprio registro. Se o arquivo utiliza mais blocos de disco que o permitido no registro, o arquivo terá registros adicionais no diretório.

Os campos na Fig. 3.5(a) são resumidos a seguir. O campo de *usuário* informa a quem pertence o arquivo. Durante a pesquisa, apenas os registros pertencentes ao usuário corrente são considerados. Os próximos campos dão o nome e tipo do arquivo. O campo *tamanho* é necessário porque um arquivo grande que ocupa mais de 16 blocos, utiliza múltiplos registros no diretório. Estes campos são usados para especificar a ordem dos registros. O campo *contador de bloco* informa quais dos 16 blocos de disco estão em uso. Os 16 campos finais contém os números dos blocos de disco. O tamanho dos arquivos é medido em blocos, não em bytes.

Vamos considerar agora exemplos de sistemas de diretório em árvore (hierárquicos). A Fig. 3.5(b) ilustra um registro de diretório do MS-DOS com 32 bytes de comprimento e armazenando o nome do arquivo e o número do primeiro bloco, dentre outros itens. O número do primeiro bloco pode ser usado como um índice dentro da FAT, para achar o segundo bloco, e assim sucessivamente. Deste modo, todos os blocos podem ser encontrados a partir do primeiro bloco armazenado na FAT. Exceto para um diretório raiz, o qual é de tamanho fixo (448 registros para um disquete de 1.44M), os diretórios do MS-DOS são arquivos e podem conter um número arbitrário de registros.

A estrutura de diretório usada no UNIX é extremamente simples, como mostra a Fig. 3.5(c). Cada registro contém exatamente o nome do arquivo e seu número de *inode*. Todas as informações sobre tipo, tamanho, marcas de tempo, propriedade, e blocos do disco estão contidas no *inode* (veja Fig. 3.4). Todos os diretórios do UNIX são arquivos e podem conter um número arbitrário destes registros.

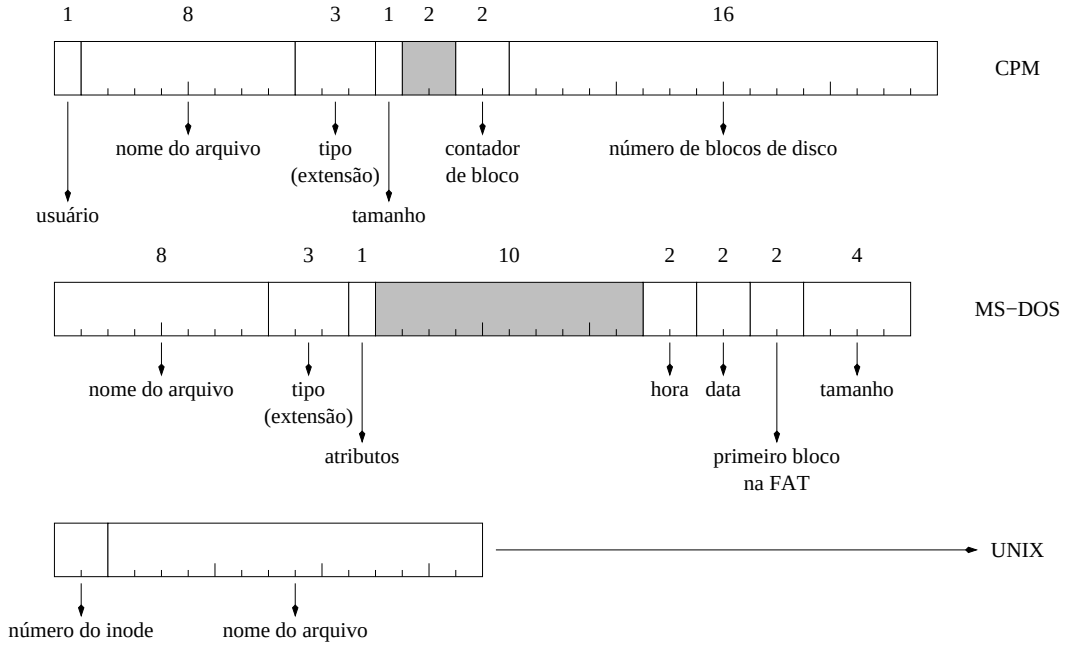

Fig. 3.5: Registros de diretórios: (a) CPM; (b) MS-DOS; (c) UNIX

Quando um arquivo é aberto, o sistema de arquivos recebe o nome de arquivo fornecido e localiza seus blocos no disco. Vamos considerar como o nome de caminho `/usr/mfm/mailbox` é localizado. Usaremos o UNIX como um exemplo, mas o algoritmo é basicamente o mesmo para todo sistema hierárquico de diretórios. Primeiro, o sistema de arquivo localiza o diretório raiz. No UNIX, o *inode* da raiz é posicionado num lugar fixo no disco.

Então, procura-se pelo primeiro componente do caminho, `usr`, no diretório raiz para achar o *inode* do arquivo `/usr`. Deste *inode*, o sistema localiza o diretório para `/usr` e procura pelo próximo componente, `mfm`, neste caso. Quando o registro para `mfm` é encontrado, tem-se o *inode* para o diretório `/usr/mfm`. A partir deste *inode*, pode-se achar o próprio diretório e procurar pela entrada do arquivo `mailbox`. O *inode* para este arquivo é então lido para a memória e lá será mantido até que o arquivo seja fechado. Este processo é ilustrado na Fig. 3.6.

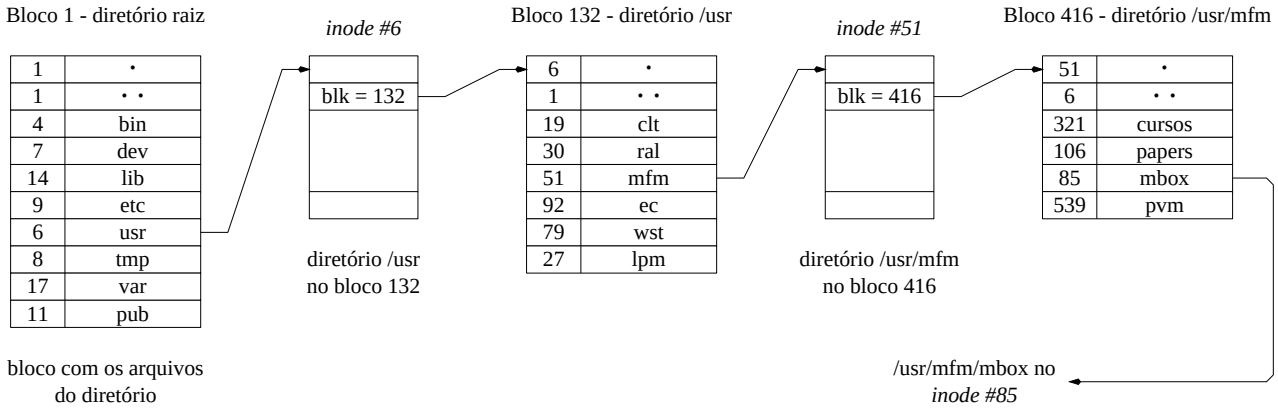

Fig. 3.6: Os passos para achar `/usr/mfm/mailbox`

Nomes de caminhos relativos são pesquisados de forma idêntica, apenas partindo do diretório de trabalho em vez de partir-se do diretório raiz. Todos os diretórios têm registros para `.` e `..`, criados juntamente com o diretório. O registro `.` armazena o número do *inode* do diretório corrente, e o registro `..` o número do *inode* do diretório pai. Assim, o procedimento de procurar por `../src/prog.c` simplesmente localiza `..` no diretório de trabalho, acha o número do *inode* para o diretório pai, e pesquisa pelo diretório `src`. Nenhum mecanismo especial é necessário para manipular estes nomes.

Arquivos Compartilhados

Não raro, usuários desenvolvendo trabalhos em equipe necessitam compartilhar arquivos. Como resultado, é conveniente que um mesmo arquivo pertença simultaneamente a diferentes diretórios. A Fig. 3.7 mostra o sistema de arquivos da Fig. 3.1(c), com um dos arquivos do usuário C presente em um dos diretórios do usuário B. A associação entre um diretório e um arquivo pertencente a outro diretório é chamada de *conexão* ou *link* (linha pontilhada da Fig. 3.7). O sistema de arquivos é agora um grafo acíclico dirigido, ou DAG *(directed acyclic graph)*.

Compartilhar arquivos é conveniente, mas também fonte de alguns problemas. Por exemplo, se diretórios armazenam endereços de disco, como no CP/M, a conexão se dá pela cópia dos endereços dos blocos do diretório do qual o arquivo já faz parte para o diretório sendo conectado. Se um usuário aumentar o tamanho do arquivo, os novos blocos serão listados somente no diretório deste usuário: as mudanças não serão visíveis para os outros usuários, anulando desta forma o propósito do compartilhamento.

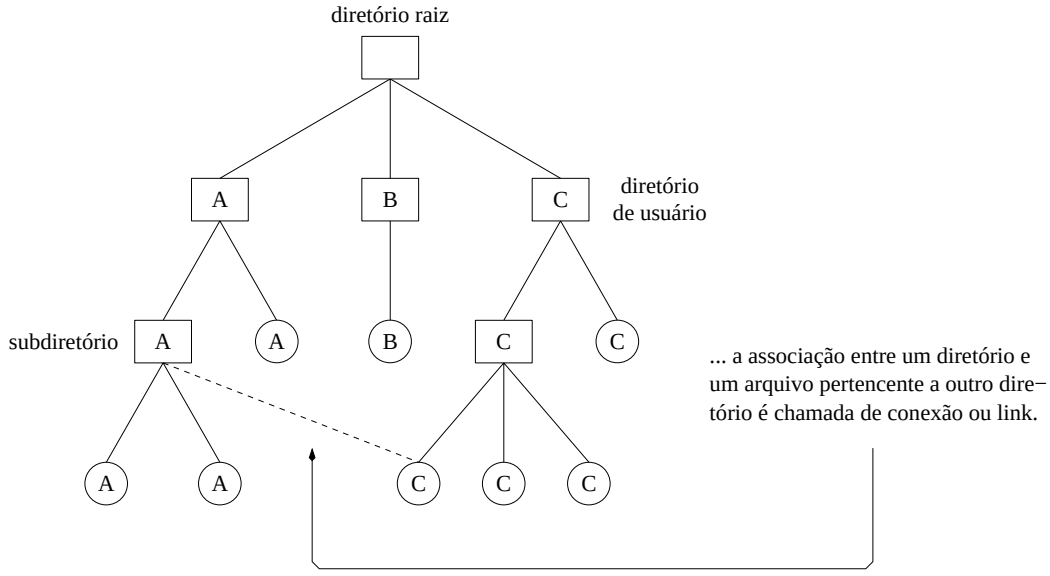

Fig. 3.7: Um sistema de arquivos contendo um arquivo compartilhado

Este problema pode ser solucionado de duas maneiras. Na primeira, os blocos do disco não são listados nos diretórios, mas em uma pequena estrutura de dados associada com o arquivo em questão. Os diretórios então apontariam justamente para a pequena estrutura de dados. Esta é a estratégia usada no UNIX (onde a pequena estrutura de dados é o *inode*).

Na segunda solução, B conecta-se a um dos arquivos de C através da criação (em B) de um arquivo especial. Este arquivo contém justamente o caminho do arquivo conectado. Quando B referencia o arquivo conectado, o sistema operacional identifica-o como do tipo *link*, lendo deste apenas o caminho para o arquivo compartilhado. De posse do caminho, o arquivo compartilhado é acessado. Este método é chamado de *conexão simbólica*.

Cada um destes métodos têm suas desvantagens. No primeiro método, no momento que B conecta-se ao arquivo compartilhado, o *inode* registra C como proprietário do arquivo. A criação de uma conexão não muda o proprietário do arquivo (ver Fig. 3.8), mas incrementa um contador no *inode* que diz quantos diretórios apontam para o arquivo.

No caso de C subsequentemente tentar remover o arquivo, o sistema encontra um problema. Se o sistema remover o arquivo e seu *inode*, B terá um registro de diretório apontando para um *inode* inválido. Se o *inode* for mais tarde reutilizado para um outro arquivo, a conexão de B apontará para o arquivo incorreto. O sistema pode ver pelo contador do *inode* que o arquivo está ainda em uso, mas não há maneira de encontrar todos os registros do diretório para o arquivo, a fim de apagá-los. Ponteiros para os diretórios não podem ser armazenados no *inode*, uma vez que podem haver um número ilimitado de diretórios.

A única solução é remover os registros do diretório C, mas abandonar o *inode* intacto, com o contador em 1, como mostra a Fig. 3.8(c). Temos agora uma situação na qual B tem um registro de diretório para um arquivo de C. Se o sistema faz controle de cotas, C continuará sendo contabilizado pelo arquivo até B decidir removê-lo, momento em que o contador irá para 0 e o arquivo será removido.

Empregando-se conexões simbólicas este problema não ocorre pois somente um diretório proprietário guarda o ponteiro para o *inode*. Diretórios que acabaram de conectar-se ao arquivo armazenam o caminho para o arquivo, não ponteiros para o seu *inode*. Quando o arquivo for removido do diretório proprietário, o *inode* é liberado pelo sistema, e subsequentes tentativas

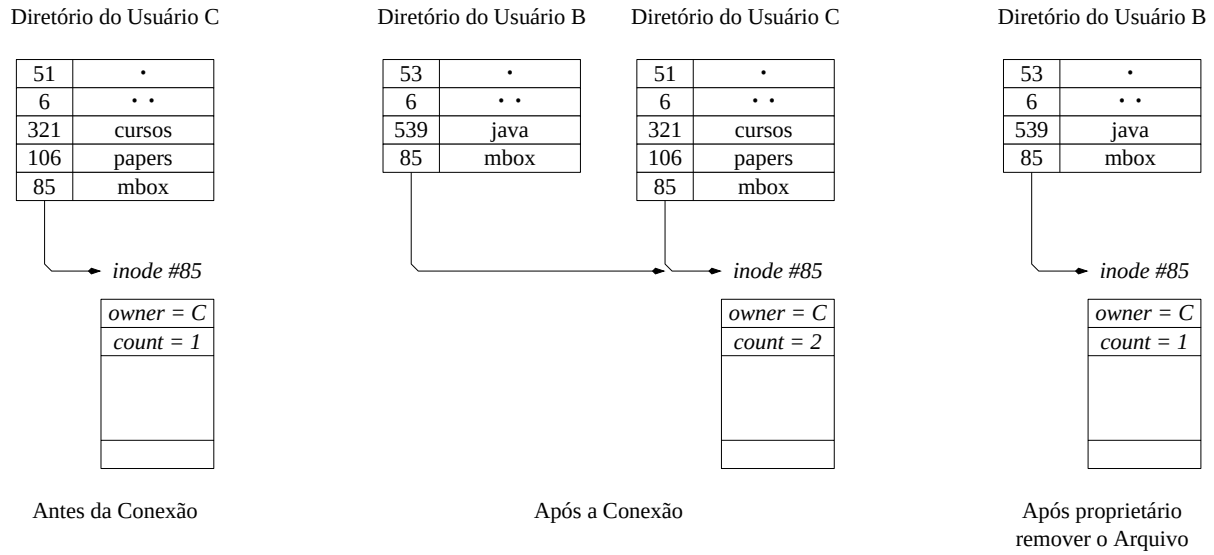

Fig. 3.8: (a) situação anterior à conexão; (b) após a conexão ter sido feita; (c) após o proprietário remover o arquivo

para usar o arquivo via conexão simbólica falhará, dada a incapacidade do sistema em localizar o arquivo. Remover uma conexão simbólica não afeta o arquivo, causando apenas o decréscimo do contador do *inode*.

Conexões simbólicas introduzem um *overhead* extra na manipulação de arquivos. Num acesso via conexão simbólica, vários *inodes* devem ser lidos do disco: o primeiro para acessar o caminho e os subsequentes para percorrer todo o caminho até a localização do arquivo (ver Fig. 3.6). Além de múltiplos acessos a disco, um *inode* extra é necessário para cada conexão simbólica, juntamente com um bloco extra para armazenar o caminho.

Existe ainda outro problema introduzido pelas conexões, simbólicas ou não. Quando conexões são permitidas, uma pesquisa numa árvore de diretórios pode encontrar o mesmo arquivo várias vezes. Isto é um problema a se considerar, por exemplo, em aplicativos que efetuam *backups*.

3.3 Confiabilidade do Sistema de Arquivos

Blocos Defeituosos

Discos frequentemente apresentam blocos defeituosos *(bad blocks)*, isto é, blocos onde a escrita e/ou leitura é impossibilitada. Duas soluções para o problema de blocos defeituosos são empregadas, uma em *hardware* e outra em *software*. A solução em *hardware* consiste em dedicar um setor no disco para a lista de blocos defeituosos. Quando o controlador do disco é iniciado, este lê a lista de blocos defeituosos e escolhe blocos sobressalentes para substituí-los. São feitas então indireções dos blocos defeituosos para os blocos sobressalentes. Daí por diante, qualquer operação envolvendo um bloco defeituoso terá efeito em seu respectivo bloco sobressalente.

A solução em *software* requer que o usuário informe (ou que o sistema de arquivos detecte) os blocos defeituosos. Estes blocos são armazenados num arquivo, acessado quando da construção da lista de blocos livres. Conhecendo os blocos defeituosos, o sistema operacional não os incorpora na lista de blocos livres, eliminando assim sua ocorrência futura em arquivos de dados.

Backups

Mesmo com uma estratégia engenhosa para tratar os blocos defeituosos, é importante se proceder *backups* frequentes. Sistemas de arquivos em discos de pequena capacidade podem ser salvos em fita magnética ou disquetes de alta densidade.

Para discos de grande capacidade (dezenas de gigabytes), salvar o conteúdo inteiro em fitas é inconveniente e consome muito tempo. Uma estratégia de fácil implementação, mas que diminui pela metade a capacidade de armazenamento, é prover cada computador com um segundo disco de igual capacidade. Ambos os discos são divididos em duas metades: dados e *backup*. Diariamente, a porção de dados de um disco é copiada para a porção de *backup* do outro disco, e vice-versa. Deste modo, se um disco for completamente destruído, nenhuma informação é perdida.

Uma outra alternativa é o *backup* incremental. Em sua forma mais simples, copia-se para fita todos os arquivos a cada semana ou mês, e, diariamente, apenas daqueles arquivos que foram modificados deste o último *backup* completo. Um outro esquema, mais eficiente, copia-se apenas aqueles arquivos que foram alterados desde o último *backup*. Para implementar este método, o horário da última duplicação para cada arquivo deve ser mantida no disco.

Consistência do Sistema de Arquivos

Outro tópico envolvendo confiabilidade é a consistência do sistema de arquivos. Muitos sistemas de arquivos lêem blocos, modifica-os, e os regrava mais tarde. Se o sistema falha antes que todos os blocos modificados forem escritos no disco, o sistema de arquivos assume um estado inconsistente. Este problema é especialmente crítico se alguns dos blocos que não foram escritos, são blocos de *inodes*, blocos de diretório, ou blocos contendo a lista de blocos livres.

Para verificar a consistência do sistema de arquivos, muitos sistemas operacionais utilizam programas utilitários desenvolvidos para este fim. Este programa é executado sempre que o sistema é iniciado, particularmente depois de um desligamento abrupto. A seguir é descrito como tal utilitário opera no UNIX[2].

O controle de consistência se dá em dois níveis: blocos e arquivos. Para controle de consistência no nível de bloco, o utilitário constrói uma tabela com dois contadores por bloco, ambos iniciados em 0. O primeiro contador rastreia quantas vezes o bloco aparece no arquivo; o segundo registra com que frequência ele aparece na lista de blocos livres.

O utilitário lê todos os *inodes*. Começando de um *inode*, é possível construir uma lista de todos os números de blocos usados no correspondente arquivo. Assim que cada número de bloco é lido, seu respectivo contador na primeira tabela é incrementado. A seguir, é examinada a lista de blocos livres rastreando todos os blocos que não estão em uso. Cada ocorrência de um bloco na lista de blocos livres resulta no incremento do respectivo contador na segunda tabela.

Se o sistema de arquivo for consistente, cada bloco terá o valor 1 na primeira tabela ou na segunda tabela. Contudo, em caso de falha, pode detectar-se blocos que não ocorrem em nenhuma das tabelas (blocos perdidos). Embora blocos perdidos não causem um dano real, eles desperdiçam espaço, reduzindo assim a capacidade do disco. A solução para blocos perdidos é direta: o verificador do sistema de arquivos acrescenta-os na lista de blocos livres.

Outra situação possível de ocorrer é a repetição de blocos na lista de blocos livres. A solução neste caso também é simples: reconstruir a lista de blocos livres, eliminando-se as duplicações.

[2]Este utilitário denomina-se *fsck (file system checker)*.

O mais grave é a ocorrência do mesmo bloco de dados em dois ou mais arquivos. Se cada um desses arquivos for removido, o bloco duplicado será posto na lista de blocos livres, chegando-se em uma situação em que o mesmo bloco está, ambiguamente, em uso e livre ao mesmo tempo. Se ambos os arquivos forem removidos, o bloco será adicionado na lista de blocos livres duas vezes.

A ação apropriada do utilitário é alocar um bloco livre, copiar o conteúdo do bloco duplicado para o mesmo, e inserir a cópia em um dos arquivos. Desde modo, a informação dos arquivos não é alterada (embora certamente incorreta para um dos arquivos), mas a estrutura do sistema de arquivos é, pelo menos, consistente. O erro será informado para permitir ao usuário examinar a falha.

Ainda para verificar se cada bloco é contado corretamente, o utilitário também examina o sistema de diretórios (consistência no nível de arquivos). Neste caso, é usada uma tabela de contadores por arquivos (não por blocos, como anteriormente). A verificação começa no diretório raiz e, recursivamente, desce a árvore inspecionando cada diretório no sistema de arquivos. Para cada arquivo encontrado, incrementa-se o contador para o seu respectivo *inode*.

Quando toda a árvore de diretórios é percorrida, tem-se uma lista, indexada pelo número do *inode*, descrevendo quantos diretórios apontam para aquele *inode*. O utilitário então compara esses valores com os contadores dos *inodes*. Em um sistema de arquivos consistente, ambos contadores coincidirão. Contudo, dois tipos de erros podem ocorrer: o contador do *inode* pode ser maior ou menor que o da lista do utilitário.

Se a contagem no *inode* for maior que o número de registros do diretório, então mesmo se todos os arquivos forem removidos dos diretórios, o contador ainda será diferente de 0 e o *inode* não será liberado. Este erro não é catastrófico, mas consome espaço no disco com arquivos que não estão em nenhum dos diretórios. O contador de conexões do *inode* deve ser corrigido através da atribuição do valor obtido pelo utilitário.

Outro erro (potencialmente catastrófico) ocorre quando o contador do *inode* é menor que o encontrado pelo utilitário. A medida que os arquivos que apontam para o *inode* vão sendo removidos, o contador do *inode* pode chegar a zero, momento que o *inode* e seus respectivos blocos são liberados. Esta ação resultará em um dos diretórios apontando para um *inode* não mais em uso, cujos blocos podem rapidamente ser atribuídos a outros arquivos. Novamente, a solução é forçar o contador do *inode* para o número real de registros do diretório (obtidas pelo utilitário).

Estas duas operações, verificar blocos e verificar diretórios, são frequentemente integradas por razões de eficiência (i.e., uma única passagem sobre os *inodes* é requerida). Outros controles heurísticos são também possíveis. Por exemplo, diretórios têm um formato definido, com um número *inodes* e nomes ASCII. Se um número *inode* for maior que o número de *inodes* no disco, o diretório encontra-se num estado inconsistente.

3.4 Desempenho do Sistema de Arquivos

Um acesso a disco é muito mais lento que um acesso a memória. Ler uma palavra da memória leva tipicamente algumas centenas de nanosegundos. Ler um bloco do disco requer alguns milisegundos, um fator 100.000 vezes mais lento. Como resultado, muitos sistemas de arquivos têm sido projetados para reduzir o número necessário de acessos a disco.

A técnica mais comum para reduzir o acesso a disco é a *block cache* ou *buffer cache*. Neste contexto, uma *cache* é uma coleção de blocos que pertencem logicamente ao disco, mas são mantidos na memória por razões de desempenho.

Vários algoritmos podem ser usados para gerenciar o *cache*, mas o mais comum é o que verifica todas as requisições de leitura para ver se o bloco referido está na *cache*. Se estiver, a requisição de leitura pode ser satisfeita sem acesso a disco. Se o bloco não estiver na *cache*, ele é inicialmente lido para a *cache*, e então copiado para a área do processo que requisitou o acesso. Requisições subsequentes do mesmo bloco podem ser satisfeitas através da *cache*.

Quando um bloco tem que ser carregado para uma *cache* cheia, algum bloco terá que ser removido e reescrito no disco, caso tenha sido modificado desde o instante em que foi instalado na *cache*. Esta situação é muito parecida com a paginação, e todos os algoritmos usuais de paginação, tal como o "menos recentemente usado" (LRU) podem ser aplicados neste contexto.

Se um bloco for essencial para a consistência do sistema de arquivos (basicamente tudo, exceto blocos de dados), e foi modificado na *cache*, é necessário que o mesmo seja escrito no disco imediatamente. Escrevendo blocos críticos rapidamente no disco, reduzimos a probabilidade que falhas danifiquem o sistema de arquivos.

Até mesmo com estas medidas para manter a integridade do sistema de arquivos, é indesejável manter blocos de dados na *cache* durante muito tempo antes que sejam descarregados em disco. Os sistemas de arquivos adotam duas estratégias para tal. No UNIX, a chamada de sistema `sync`, força com que todos os blocos modificados sejam gravados em disco imediatamente. Quando o sistema é iniciado, um programa, usualmente chamado *update*, é ativado. De 30 em 30 segundos, a atualização da *cache* é estabelecida. Como resultado, na pior hipótese, perde-se os blocos gravados nos últimos 30 segundos em caso de uma pane.

A solução do MS-DOS é gravar todo bloco modificado para o disco tão logo tenha sido escrito. *Caches* nas quais blocos modificados são reescritos imediatamente no disco são chamadas *caches de escrita direta*. Elas requerem muito mais E/S de disco que *caches* de escrita não direta. A diferença entre estas duas técnicas pode ser vista quando um programa escreve num buffer de 1K, caracter por caracter. O UNIX coleta todos os caracteres da *cache*, e escreve o bloco de uma vez em 30 segundos, ou quando o bloco for removido da *cache*.

O MS-DOS faz acesso a disco para cada um dos caracteres escritos. Naturalmente, muitos programas fazem "buferização" interna, procedendo gravações em disco apenas quando existir uma determinada quantidade de bytes pendentes. A estratégia adotada pelo MS-DOS foi influenciada pela garantia que a remoção de um disco flexível de sua unidade não causa perda de dados. No UNIX, é necessária uma chamada `sync` antes da remoção de qualquer meio de armazenamento (ou da parada programada do sistema).

Cache não é a única maneira de aumentar o desempenho do sistema de arquivos. Uma outra maneira é reduzir a quantidade de movimentos do braço do disco, colocando blocos que estão sendo acessados em sequência, preferencialmente em um mesmo cilindro. Quando um arquivo é escrito, o sistema de arquivos aloca os blocos um por vez, a medida do necessário. Se os blocos livres estiverem gravados em um mapa de bits, e o mapa de bits inteiro está na memória principal, é fácil escolher um bloco que está mais perto do bloco anterior. Com uma lista de blocos livres, parte da qual está no disco, é mais difícil alocar blocos próximos.

Entretanto, com uma lista de blocos livres alguns agrupamentos de blocos podem ser feitos. O artifício é manter a trilha do disco armazenada não em blocos, mais em grupos consecutivos de blocos. Se uma trilha consistir de 64 setores de 512 bytes, o sistema pode usar blocos de 1K bytes (2 setores), porém alocando espaço no disco em unidades de 2 blocos (4 setores). Isto não é o mesmo que ter um bloco de 2K, posto que na *cache* ainda se usa blocos de 1K com transferência para disco também de 1K. Entretanto, a leitura sequencial reduz o número de busca de um fator de 2, melhorando consideravelmente o desempenho.

Outra variante é fazer uso do posicionamento rotacional. Quando se alocam blocos, o siste-

ma atenta para colocar blocos consecutivos de um arquivo no mesmo cilindro, mas intercalados. Deste modo, se um disco tiver um tempo de rotação de 16.67 mseg e 4 mseg são necessários para o processo do usuário requerer e acessar um bloco do disco, cada bloco deve ser colocado em ao menos um quarto da distância do seu antecessor.

Um outro agravante no desempenho dos sistemas que usam *inodes*, ou algo similar, é que para ler até mesmo um pequeno arquivo, seja necessário 2 acessos no disco: um para o *inode* e outro para o bloco. Caso todos os *inodes* estejam próximos do início do disco, distância média entre o *inode* estará em torno da metade do número de cilindros, o que exige longas buscas.

Melhor alternativa é instalar os *inodes* no meio do disco, reduzindo a média de busca entre o *inode* e o primeiro bloco de um fator de 2. Uma outra idéia consiste em dividir o disco em grupos de cilindros, cada qual com os seus próprios *inodes*, blocos, e lista de blocos livres. Quando se cria um novo arquivo, qualquer *inode* pode ser escolhido, mas tendo-se o cuidado de achar um bloco no mesmo grupo de cilindros onde o *inode* está. Caso nenhum bloco esteja disponível, escolhe-se um bloco do cilindro mais próximo.

3.5 O Sistema de Arquivos do UNIX (System V)

O função do núcleo do ponto de vista do sistema de arquivos consiste em permitir que os processos armazenem novas informações ou que recuperem informações previamente armazenadas [3, 2]. Para a realização destas atividades o núcleo necessita trazer informações auxiliares para a memória. Como exemplo, podemos destacar o super-bloco, o qual descreve, dentre outras informações, a quantidade de espaço livre disponível no sistema de arquivos e os *inodes* disponíveis. Estas informações são manipuladas de forma transparente para os processos.

Com o objetivo de minimizar a frequência de acesso ao disco o núcleo gerencia um *pool* interno de buffers denominado de *cache de buffers*. Neste caso, trata-se de uma estrutura de *software* que não deve ser confundida com a memória *cache* em *hardware* que é utilizada para acelerar as referências à memória.

3.5.1 O Cache de Buffers

O número de buffers que fazem parte do *cache de buffers* depende do tamanho da memória e das restrições de desempenho impostas, sendo o espaço necessário para armazená-los alocado pelo núcleo durante a iniciação do sistema. Cada buffer é formado de duas partes: um segmento de memória utilizado para abrigar os dados que são lidos/escritos da/na memória e um cabeçalho que identifica o buffer.

A visão que o núcleo possui do sistema de arquivos no disco é uma visão lógica. O mapeamento desta visão lógica na estrutura física do disco é realizada pelos *drivers* dos dispositivos. Dentro desta visão, o núcleo enxerga o sistema de arquivos como sendo formado por vários blocos de disco; no caso, um buffer corresponde a um bloco, sendo o seu conteúdo identificado pelo núcleo através de um exame dos campos presentes no cabeçalho associado ao buffer. Podemos concluir que o buffer é a cópia na memória de um bloco de disco, cópia esta que é mantida até o instante em que o núcleo decide mapear um outro bloco no buffer. Uma restrição importante é a de que um bloco do disco não pode ser mapeado em mais de um buffer ao mesmo tempo.

O cabeçalho de um buffer contém 4 classes de informação (ver Fig. 3.9): identificação, posicionamento, estado e posição dentro do segmento de memória. A identificação permite reconhecer a qual bloco do disco o buffer está associado. O cabeçalho do buffer contém um campo com o número do dispositivo e um campo com o número do bloco. Estes campos

especificam o sistema de arquivo e o número do bloco de dado no disco identificando de forma única o buffer. Com relação ao posicionamento, o cabeçalho possui apontadores para posicionar o buffer em duas estruturas: lista de buffers livres e a fila *hash*. O cabeçalho possui um apontador para a área de dado (segmento) correspondente ao buffer. Deve ser observado que o tamanho desta área deve ser igual ao tamanho do bloco do disco. O quarto campo é o campo de status que indica a situação do buffer, podendo ter um dos seguintes valores:

- buffer trancado (*locked*);
- buffer contendo dados válidos;
- buffer com escrita retardada - significa a situação em que o núcleo deve escrever o conteúdo do buffer no disco antes de reatribuir o buffer a um outro bloco;
- o núcleo encontra-se lendo ou escrevendo o conteúdo do buffer no disco;
- um processo encontra-se aguardando que o buffer torne-se livre.

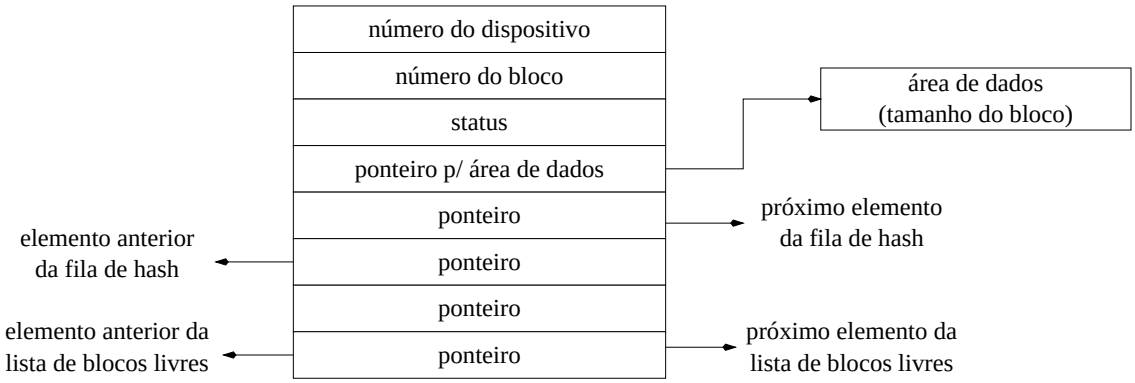

Fig. 3.9: Cabeçalho do buffer

Estrutura do Pool de Buffers

Os blocos de dado no disco são organizados no *cache de buffers* através de uma política denominada de "uso menos recente". Através desta política, quando um buffer é alocado a um determinado bloco, este buffer não poderá ser realocado a um outro bloco, a menos que todos os outros blocos tenham sido usados mais recentemente. Para implementação desta política o núcleo mantém uma lista de buffers livres que é organizada segundo a ordem do uso menos recente.

O núcleo retira um buffer da cabeça da lista quando da necessidade de um buffer livre. É possível que um buffer seja retirado do meio da lista caso o núcleo identifique um bloco específico no *cache de buffers*. Em ambos os casos o buffer é removido da lista. Quando o núcleo retorna um buffer ao pool, ele normalmente coloca o buffer no final da lista sendo que, ocasionalmente (em situações de erro), o buffer é colocado na cabeça da lista. O buffer nunca é recolocado no meio da lista de buffers livres.

Quando o núcleo necessita acessar um bloco do disco, ele procura um buffer que possua a combinação adequada *número de dispositivo-número de bloco*. Para evitar a necessidade do núcleo pesquisar o *cache de buffers* por completo, estes são organizados em filas separadas,

espalhadas por uma função de *hash* que tem como parâmetros os números do dispositivo e do bloco. Os buffers são colocados em uma fila *hash* circular, duplamente ligada, em uma forma equivalente à estrutura da lista de buffers livres.

Todo buffer está na fila *hash*, não existindo, entretanto, significado para a sua posição na fila. Um buffer pode encontrar-se, simultaneamente, na lista de buffers livres, caso o seu estado seja livre, e na fila *hash* (por exemplo, o buffer 64 da Fig. 3.10). Desta forma, o núcleo pode procurar um buffer na lista *hash* caso ele esteja procurando um buffer específico, ou ele pode remover um buffer da lista de buffers livres caso ele esteja procurando por um buffer livre qualquer. Resumindo, um buffer que se encontra na lista *hash* pode ou não encontrar-se na lista de buffers livres.

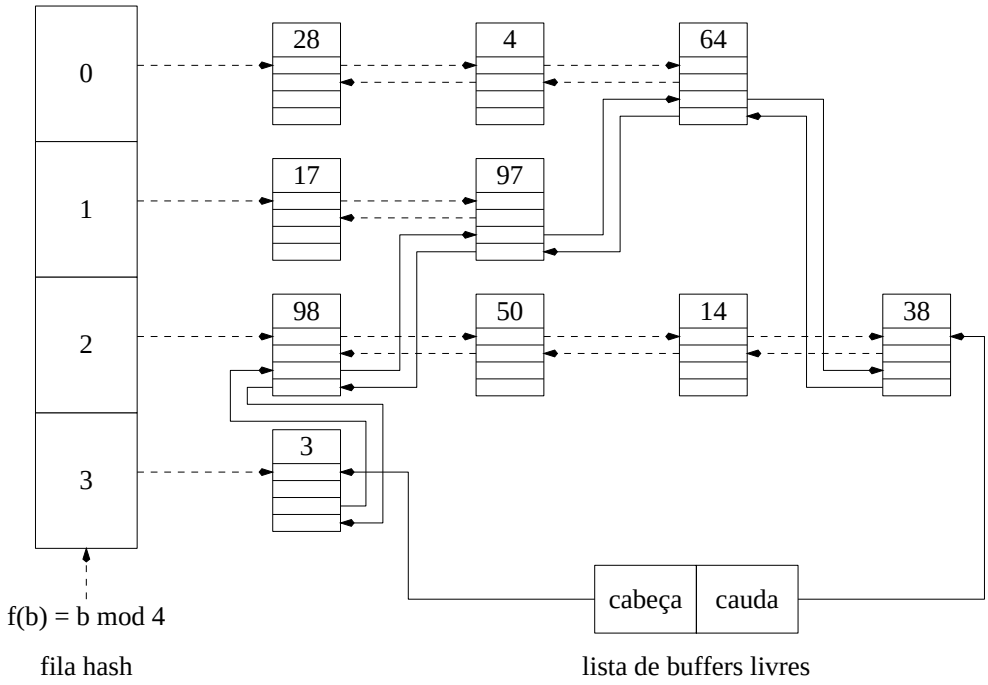

Fig. 3.10: Estrutura do *cache de buffers*: fila de *hash* e lista de buffers livres

Cenários para Recuperação de um Buffer

O *cache de buffers* é manipulado por um conjunto de algoritmos que são invocados pelo sistema de arquivos. O sistema de arquivos determina o número do dispositivo lógico e o número do bloco a ser acessado, por exemplo, quando um processo necessita ler dados de um arquivo.

Quando da leitura ou escrita de dados de um determinado bloco do disco, o núcleo determina se o bloco encontra-se no *cache de buffers* e, caso ele não se encontre, um buffer livre é atribuído ao bloco. Os algoritmos para leitura/escrita em blocos do disco usam um algoritmo `(getblk)` para alocação de buffers a partir do pool.

Existem cinco cenários típicos associados ao algoritmo `getblk`:

1. O núcleo encontra o buffer correspondente ao bloco na fila *hash* e o buffer está livre. Neste caso, o algoritmo marca o buffer como "ocupado", remove-o da lista de buffers livres e retorna um ponteiro para o buffer.

2. O núcleo não encontra o buffer correspondente ao bloco na fila *hash* e aloca um buffer da lista de buffers livres, reposicionando-o na fila de *hash*.

3. O núcleo não encontra o buffer correspondente ao bloco na fila *hash* e, na tentativa de alocar um buffer da lista de buffer livre (como no cenário 2), encontra um buffer na lista de buffers livres marcado com escrita adiada *(delayed write)*. Neste caso, o núcleo deve escrever (assincronamente) o buffer no disco e continuar a procura de um buffer livre para o bloco.

4. O núcleo não encontra o buffer correspondente ao bloco na fila *hash*, e a lista de buffers livres encontra-se vazia. Neste caso, o processo é bloqueado até que outro processo devolva um buffer à lista de buffers livres.

5. O núcleo encontra o buffer correspondente ao bloco na fila *hash*, mas o seu buffer está ocupado (outro processo está acessando exatamente este bloco). Neste caso, o processo é bloqueado até que o outro processo conclua a operação sob este bloco.

Note que em qualquer cenário acima, ou o algoritmo `getblk` volta um buffer já posicionado na fila de *hash* para que a operação de E/S possa continuar; ou bloqueia o processo a espera de buffer ou liberação do bloco.

Além do algoritmo `getblk`, existem outros quatro algoritmos que operam o *cache de buffers*:

1. `brelse`: libera um bloco, retornando-o à lista de blocos livres.

2. `bread`: lê sincronamente (com espera) o bloco do disco para o buffer.

3. `breada`: lê assincronamente (sem espera) o bloco do disco para o buffer.

4. `bwrite`: escreve um bloco do buffer para o disco.

3.5.2 Representação Interna dos Arquivos

Todo arquivo no UNIX *System* V contém um único *inode*. O *inode* possui as informações necessárias para um processo acessar um arquivo, tais como: proprietário do arquivo, direito de acesso, tamanho do arquivo e localização dos dados (blocos) do arquivo. A referência a um arquivo é feita pelo seu nome e, através deste, o núcleo determina o *inode* do arquivo.

Os algoritmos envolvidos na manipulação dos *inodes* resumidos a seguir utilizam os algoritmos que operam sobre o *cache de buffers* discutidos anteriormente.

- `iget`: retorna um *inode* previamente identificado, possivelmente através da sua leitura do disco via o *cache de buffers*;

- `iput`: libera o *inode*;

- `bmap`: define os parâmetros do núcleo para o acesso a um arquivo;

- `namei`: converte um nome (*path*) de arquivo no *inode* correspondente;

- `alloc`: aloca blocos do disco para os arquivos;

- `free`: libera blocos de disco;

- `ialloc`: aloca *inodes* para os arquivos;
- `ifree`: libera *inodes*.

Estes algoritmos utilizam outros que manipulam o *cache de buffers* (`getblk, brelse, bread, breada, bwrite`).

Estrutura do Inode

Um *inode* existe estaticamente no disco e o núcleo realiza a sua leitura para a memória quando necessita manipulá-lo. O *inode* no disco contém os seguintes campos (ver Fig. 3.4):

- identificador do dono do arquivo: divido em dono individual e grupo;
- tipo do arquivo: regular, diretório, especial ou FIFO (pipes);
- permissão de acesso;
- instantes de acesso ao arquivo: última modificação, último acesso e última modificação ocorrida no *inode*;
- número de conexões *(links)* associados ao arquivo;
- endereços no disco dos blocos de dados do arquivo;
- tamanho do arquivo.

A cópia do *inode* em memória contém os seguintes campos em adição aos campos do *inode* em disco:

- status do *inode* na memória indicando:

  - o *inode* está trancado;

  - processo encontra-se esperando que o *inode* seja liberado;

  - a representação do *inode* na memória difere da cópia do disco devido a mudanças nos dados do *inode*;

  - a representação do arquivo na memória difere da cópia do disco devido a mudanças nos dados do arquivo;

  - o arquivo é um *mount point*.

- o número do dispositivo lógico do sistema de arquivos que contém o arquivo;
- o número do *inode*. O *inode* no disco não necessita deste número pois os *inodes* são armazenados em um arranjo linear no disco;
- apontadores para outros *inodes* na memória. O núcleo liga os *inodes* em filas *hash* e em uma lista de *inodes* livres da mesma forma que os buffers são ligados no *cache de buffers*;
- um contador de referência, indicando o número de instâncias do arquivo que estão ativas.

A diferença mais significativa entre o *inode* na memória e o cabeçalho do buffer no *cache de buffers* é o contador de referência presente no *inode*, o qual indica o número de instâncias (acessos) ativas do arquivo. Um *inode* torna-se ativo quando um processo realiza a sua alocação, por exemplo, através da abertura do arquivo. Um *inode* é colocado na lista de *inodes* livres se, e somente se, o seu contador de referência é 0, significando que o núcleo pode realocar sua estrutura na memória a um outro *inode* do disco. A lista de *inodes* livres serve como um *cache de inodes* (exatamente como o *cache de buffers*).

Acesso aos Inodes

O núcleo identifica um *inode* específico através do número do seu sistema de arquivo e do seu número dentro deste sistema. O algoritmo `iget` cria uma cópia do *inode* na memória física, realizando um papel equivalente ao algoritmo `getblk` utilizado para encontrar um bloco do disco no *cache de buffers*.

Caso o algoritmo não encontre o *inode* na fila *hash* associada ao número do dispositivo e ao número do *inode* procurado, um *inode* é alocado a partir da lista de *inodes* livres sendo consequentemente trancado *(locked)*. A partir deste ponto o núcleo encontra-se em condição de ler o *inode* do disco correspondente ao arquivo que esta sendo referenciado e copiá-lo para a memória física. Para determinar o bloco do disco que contém o *inode* referenciado e para determinar o *offset* do *inode* dentro do bloco, o núcleo utiliza as seguintes expressões:

- *nº* do bloco: ((*nº* do *inode*-1) / número de *inodes* por bloco) + número do bloco inicial da lista de *inodes*
- *offset*: ((*nº* do *inode*-1) MOD (número de *inodes* por bloco) * tamanho do *inode* no disco

Quando o núcleo libera um *inode* (algoritmo `iput`) ele decrementa seu contador de referência. Caso o valor do contador se torne 0, o núcleo escreve o *inode* no disco caso a cópia na memória seja diferente da cópia no disco. O *inode* é colocado na lista de *inodes* livres na hipótese de que este *inode* possa ser necessário posteriormente.

Estrutura de um Arquivo Regular

A indicação dos blocos do disco que constituem um determinado arquivo encontra-se no *inode* associado ao arquivo. Esta indicação traduz-se na utilização de 13 números para blocos. Os 10 primeiros números (blocos diretos) contém números para blocos de disco; o 11º número é um indireto simples, o 12º um indireto duplo e o 13º um indireto triplo. Esta solução, conforme discutido anteriormente, permite que os 10 primeiros números componham um arquivo de 10 Kbytes (supondo blocos de 1 Kbytes). Elevando-se este número para 11, 12 e 13, tem-se, respectivamente, arquivos de até 256 Kilobytes, 64 Megabytes e 16 Gigabytes (máximo para o UNIX).

Os processos enxergam o arquivo como um conjunto de bytes começando com o endereço 0 e terminando com o endereço equivalente ao tamanho do arquivo menos 1. O núcleo converte esta visão dos processos em termos de bytes em uma visão em termos de blocos: o arquivo começa com o bloco 0 (apontado pelo primeiro número de bloco no *inode*) e continua até o número de bloco lógico correspondente ao tamanho do arquivo.

Um exame mais detalhado das entradas dos blocos no *inode* pode mostrar que algumas entradas possuem o valor 0, indicando que o bloco lógico não contém dados. Isto acontece caso nenhum processo tenha escrito dados no arquivo em qualquer *offset* correspondente a estes

blocos. Deve ser observado que nenhum espaço em disco é desperdiçado para estes blocos. Este *layout* pode ser criado pelo uso das chamadas `lseek` e `write`.

Estrutura do Diretório

Quando da abertura de um arquivo, o sistema operacional utiliza o nome *(path)* do arquivo fornecido pelo usuário para localizar os blocos do disco associados ao arquivo. O mapeamento do nome nos *inodes* está associado à forma como o sistema de diretório encontra-se organizado.

A estrutura de diretório usada no UNIX é extremamente simples. Cada entrada contém o nome do arquivo e o número do seu *inode*, representados através de 16 bytes, ou seja, 2 bytes para o número do *inode* e 14 bytes para o nome[3]. Todos os diretórios do UNIX são arquivos e podem conter um número arbitrário destas entradas de 16 bytes.

Quando um arquivo é aberto, o sistema deve, através do nome do arquivo, localizar os seus blocos no disco. Caso o nome do arquivo seja relativo, isto é, associado ao diretório corrente, o núcleo acessa o *inode* associado ao diretório corrente na área U do processo e realiza um procedimento semelhante ao exemplo da subseção 3.2.

3.5.3 Atribuição de *inodes* e Blocos

Até o momento consideramos a hipótese de que um *inode* havia sido previamente associado a um arquivo e que os blocos do disco já continham os dados. Será discutido na sequência como o núcleo atribui *inodes* e blocos do disco. Para isto é importante conhecer a estrutura do super-bloco.

O Super-Bloco

O super-bloco é um segmento contínuo de disco formado pelos seguintes campos:

- tamanho do sistema de arquivos;
- número de blocos livres no sistema;
- lista de blocos livres disponíveis no sistema de arquivos;
- índice do próximo bloco livre na lista de blocos livres;
- tamanho da lista de *inodes*;
- número de *inodes* livres no sistema de arquivos;
- lista de *inodes* livres no sistema de arquivos;
- índice do próximo *inode* livre na lista de *inodes*;
- campos trancados *(locked)* para as listas de blocos livres e *inodes* livres;
- *flag* indicando que o super-bloco foi modificado.

O super-bloco é mantido na memória física de modo a agilizar as operações sobre o sistema de arquivos, sendo periodicamente copiado no disco para manter a consistência do sistema de arquivos.

[3]Versões correntes do UNIX permitem nomes bem mais extensos.

Atribuição de um Inode a um Novo Arquivo

O algoritmo `ialloc` atribui um *inode* do disco para um arquivo recém criado. O sistema de arquivos contém uma lista linear de *inodes*. Um *inode* nesta lista encontra-se livre quando o seu campo de tipo é zero. Para melhorar o desempenho, o super-bloco do sistema de arquivos contém um arranjo que atua como um *cache* no qual são armazenados os *inodes* livres do sistema de arquivos Não confundir esta lista com aquela para abrigar os *inodes* livres no pool de *inodes* em memória. Aquela está relacionada à manipulação de *inodes* associados a arquivos já criados e que serão ativos para manipulação por parte dos processos. Esta lista de *inodes* livres associada ao super-bloco abriga os *inodes* no disco que não estão alocados a nenhum arquivo e que poderão ser associados aos arquivos que serão criados no sistema.

Alocação de Blocos no Disco

Quando um processo necessita escrever dados em um arquivo, o núcleo aloca blocos do disco para a expansão do arquivo, o que significa associar novos blocos ao *inode*. Para agilizar a determinação de quais blocos do disco estão disponíveis para serem alocados ao arquivo, o super-bloco contém um arranjo que relaciona o número de blocos do disco que estão livres. Este arranjo utiliza blocos de disco cujo conteúdo são números de blocos livres. Uma das entradas destes blocos constitui-se de um apontador para outro bloco contendo números de blocos livres. Estabelece-se, desta forma, uma lista ligada (em disco) de blocos que contém números de blocos livres. O super-bloco possui parte desta lista ligada, de tamanho máximo correspondendo a um bloco, sendo gerenciada quando da liberação de blocos e quando da requisição de blocos de modo a manter sempre uma indicação de parte dos blocos livres no disco. O algoritmo `alloc` realiza a alocação de um bloco do sistema de arquivos alocando o próximo bloco disponível na lista do super-bloco.

3.5.4 Chamadas de Sistema Referentes ao Sistema de Arquivos

As chamadas de sistema que compõem o sistema de arquivos utilizam, dentre outros, os algoritmos descritos acima (`getblk, ialloc`, etc.). Cada arquivo no UNIX é identificado por um *descritor de arquivos* (um número inteiro). Descritores de arquivos identificam univocamente arquivos no nível de processo, isto é, dois arquivos abertos pelo mesmo processo possuem descritores necessariamente diferentes.

As principais chamadas de sistema referentes ao sistema de arquivos serão sucintamente descritas a seguir.

1. `open`: abre um arquivo para leitura/gravação. Parâmetros da chamada indicam, por exemplo, se o arquivo deve ser criado (caso inexista) e permissões.

2. `dup`: duplica um descritor de arquivo, retornando um novo descritor associado ao mesmo arquivo.

3. `pipe`: cria no sistema de arquivos um *pipe*. Pipes são arquivos de uso exclusivo por parte de dois processos: um produtor e um consumidor de dados.

4. `close`: encerra a operação sobre um arquivo aberto.

5. `link`: cria uma conexão simbólica para um arquivo já existente.

6. `unlink`: remove uma conexão simbólica.
7. `read`: lê dados de um arquivo (para um buffer em memória).
8. `write`: escreve dados num arquivo (de um buffer em memória).
9. `lseek`: altera a posição corrente de leitura/gravação no arquivo.
10. `mknod`: cria um arquivo especial (tipo bloco ou caractere), associando-o a periférico (unidade de disco, fita, terminal, etc).
11. `mount`: adiciona uma nova unidade lógica ao sistema de arquivos.
12. `umount`: remove uma unidade lógica do sistema de arquivos.
13. `chdir`: altera o diretório corrente na área U.
14. `chown`: altera o usuário que têm a posse do arquivo.
15. `chmod`: altera permissões de um arquivo.
16. `stat`: fornece informações sobre um arquivo (tipicamente as constantes no *inode* do arquivo).

Capítulo 4

Gerenciamento de Memória

Memória é um recurso importante que deve ser cuidadosamente gerenciado. Enquanto a capacidade de armazenamento dos computadores vem crescendo continuamente, a complexidade do software cresce talvez à taxas maiores [1]. A parte do sistema operacional que gerencia a memória é chamada de *gerenciador de memória*, sendo o objeto deste capítulo.

Dentre outras tarefas, o gerenciador de memória monitora quais partes da memória estão em uso e quais estão disponíveis; aloca e libera memória para os processos; e gerencia a permuta de processos entre memória principal e secundária (quando a memória principal não é capaz de abrigar todos os processos).

4.1 Gerenciamento Sem Permuta ou Paginação

Sistemas de gerenciamento de memória podem ser divididos em duas grandes classes: aqueles que movem processos entre a memória principal e secundária (tipicamente disco) durante a execução, e aqueles que mantém os processos fixos em memória primária. Na primeira classe, o gerenciamento é baseado em técnicas de *swapping* (permuta) ou de *paginação*.

4.1.1 Monoprogramação

O esquema mais simples possível de gerenciamento de memória consiste em ter-se somente um processo na memória durante toda a sua execução. O usuário carrega um programa do disco para a memória, podendo este fazer uso de toda a máquina. Se a memória for insuficiente, o programa simplesmente tem sua execução rejeitada. Embora essa técnica ter sido comum em meados da década de sessenta, ela não é mais utilizada.

A técnica usada em microcomputadores é mostrada na Fig. 4.1. A memória é dividida entre o sistema operacional e um processo do usuário. O sistema operacional pode estar no final da memória RAM *(Random Access Memory)* como mostrado na Fig. 4.1(a), ou em ROM *(Read Only Memory)*, como mostrado na Fig. 4.1(b), ou ainda tendo os *device drivers* em ROM e o resto do sistema operacional em RAM ocupando a parte baixa da memória, como mostrado na Fig. 4.1(c).

A arquitetura IBM PC original (processadores Intel x86) utilizava o modelo da Fig. 4.1(c), com os *device drivers* localizados no bloco de 8K mais alto dentro do espaço de 1M de endereçamento. O programa na ROM é chamado de *BIOS (Basic Input Output System)*.

Quando o sistema é organizado dessa maneira, somente um processo pode estar em execução por vez. O usuário entra com um comando no terminal, e o sistema operacional carrega o

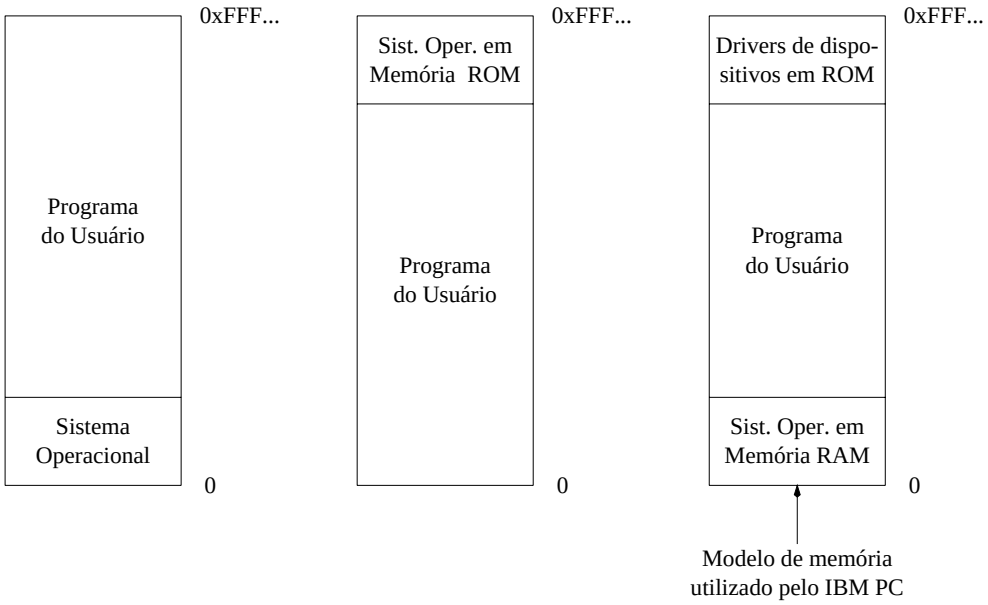

Fig. 4.1: Três formas de organizar a memória para o sistema operacional e um processo do usuário

programa requerido do disco para a memória e o executa. Quando o processo termina, o sistema operacional reassume a CPU e espera por um novo comando para carregar um outro processo na memória já liberada pelo primeiro.

4.1.2 Multiprogramação e Uso da Memória

Embora a monoprogramação seja usada em pequenos computadores, em computadores com múltiplos usuários ela é proibitiva. Grandes computadores frequentemente provêem serviços interativos para vários usuários simultaneamente. Para tal, a habilidade de ter-se mais de um processo na memória em um mesmo instante de tempo é imprescindível por razões de desempenho.

Uma outra razão para ter-se a multiprogramação, é que muitos processos gastam uma substancial fração do seu tempo para completar E/S em disco. É comum para um processo permanecer em um *loop* lendo um bloco de dados de um arquivo em disco e então realizando alguma computação sobre o conteúdo dos blocos lidos. Se for gasto 40 mseg para ler um bloco e a computação demanda apenas 10 mseg, sem a multiprogramação a CPU estará desocupada esperando pelo acesso ao disco durante 80% do tempo.

Modelagem da Multiprogramação

Quando a multiprogramação é usada, o percentual de utilização da CPU aumenta. Via de regra, se a média dos processos utilizam CPU somente 20% do tempo que permanecem na memória, com 5 processos em memória, a CPU deverá estar ocupada o tempo todo. Este modelo é otimista, entretanto, pois assume que os 5 processos nunca estejam esperando por E/S ao mesmo tempo, bem como despreza o esforço de gerenciamento dos 5 processos por parte do sistema operacional.

O melhor modelo é ver o uso da CPU do ponto de vista probabilístico. Suponha que os processo gastem em média uma fração p do tempo à espera de E/S. Com n processos na

memória por vez, a probabilidade que todos os n processos estejam esperando por E/S é p^n. A utilização da CPU é então $1 - p^n$. A Fig. 4.2 mostra a utilização da CPU em função de n, chamado *grau de multiprogramação*.

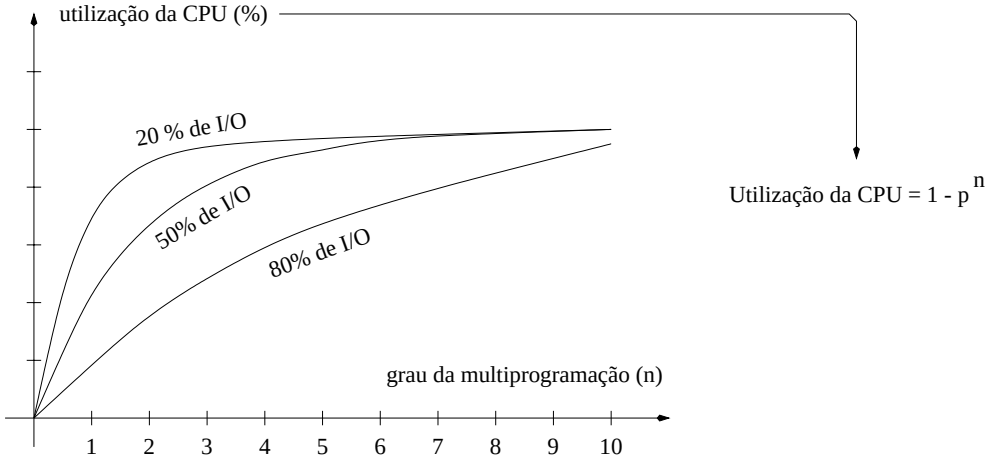

Fig. 4.2: Utilização da CPU como uma função do número de processos na memória

Da Fig. está claro que caso os processos gastem 80% do seu tempo esperando por E/S, ao menos 10 processos devem estar na memória por vez para obter um desperdício de CPU em torno de 10%. Quando se considera que um processo interativo aguardando comandos do terminal está em um estado de espera de E/S, deve ficar claro que tempos de espera para E/S superiores a 80% são usuais. Processos utilizando unidades de armazenamento com elevada frequência, também contribuem para o aumento deste percentual.

4.1.3 Multiprogramação com Partições Fixas

Se adotarmos a estratégia de admitir mais de um processo na memória por vez, devemos estabelecer uma estratégia de organização da memória. A estratégia mais simples consiste em dividir a memória em n partições (possivelmente diferentes). Estas partições podem, por exemplo, ser estabelecidas na configuração do sistema operacional.

Quando um processo inicia, este pode ser colocado em uma fila de entrada para ocupar a menor partição de tamanho suficiente para acomodá-lo. Desde que as partições são fixas, qualquer espaço em uma partição não usado pelo processo é perdido. A Fig. 4.3(a) apresenta este esquema de partição.

A desvantagem de se ordenar os processos que chegam em filas separadas torna-se aparente quando a fila para a maior partição está vazia, mas a fila para a menor partição está cheia, como no caso das partições 1 e 4 na Fig. 4.3(a). Uma organização alternativa é manter uma fila única como na Fig. 4.3(b). Toda vez que uma partição é liberada, a mesma é alocada ao primeiro processo da fila. Uma vez que é indesejável gastar uma partição grande com um processo pequeno, uma estratégia mais eficaz é procurar em toda fila de entrada a maior tarefa para a partição liberada. Note que o último algoritmo discrimina os processos pequenos, quando é usualmente desejável dar a eles o melhor tratamento, não o pior.

Este sistema, com partições fixas definidas pelo operador, foi usado pelo sistema operacional OS/360 nos grandes mainframes da IBM por muitos anos. Ele era chamado de MFT *(Multiprograming with a Fixed number of Task)*. Ele é simples de se entender e igualmente simples

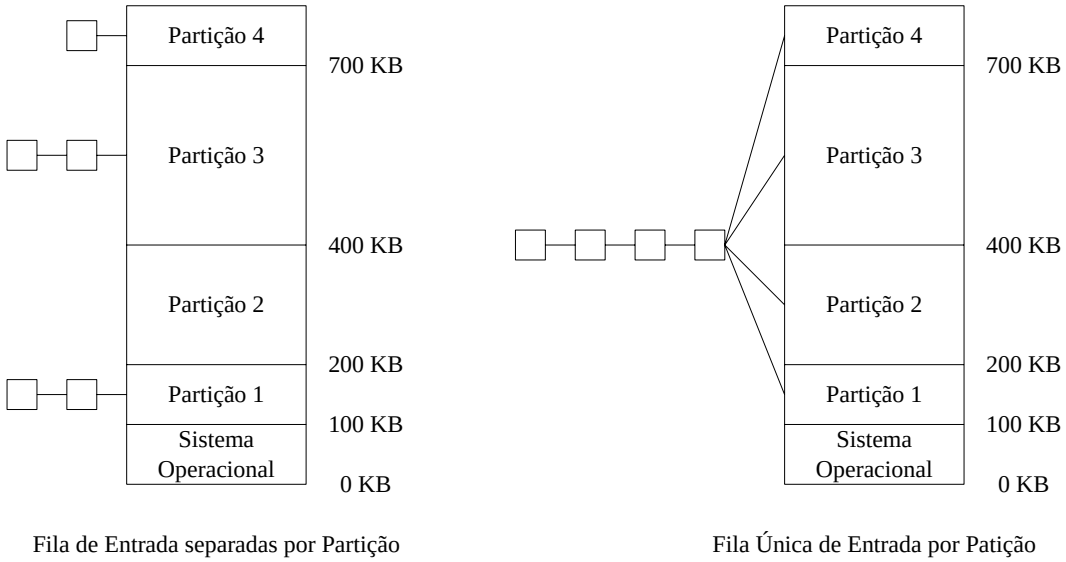

Fig. 4.3: (a) Partições de memória fixa com filas de entrada separadas para cada partição; (b) partição de memória fixa com uma fila simples de entrada

de implementar: os processos que chegam são colocados em uma fila até que uma partição adequada seja liberada, quando então são carregados e executados.

Realocação e Proteção

Multiprogramação introduz dois problemas essenciais que devem ser resolvidos: realocação e proteção. Da Fig. 4.3 está claro que diferentes processos serão executados em endereços diferentes. Quando um programa é ligado *(linked)*[1], o *linker* deve saber em qual endereço na memória o programa começará.

Por exemplo, suponha que a primeira instrução de um programa é uma chamada para um procedimento de endereço relativo 100 dentro do arquivo binário produzido pelo *linker*. Se o programa for carregado na partição 1 da Fig. 4.3(a), esta instrução saltará para o endereço absoluto 100, em plena área do sistema operacional. O que é necessário é uma chamada para 100K + 100. Se o programa for carregado na da partição 2, ele deve ser executado como uma chamada para 200K + 100, e assim por diante. Este problema é conhecido como o problema da *realocação*.

Uma solução possível é modificar realmente as instruções quando o programa é carregado para a memória (técnica denominada carregamento dinâmico). Programas carregados na partição 1 têm 100K adicionados para cada endereço, programas carregados na partição 2 têm 200K adicionados ao endereçamento, e assim sucessivamente. Para realizar a realocação durante o carregamento, o *linker* deve incluir no programa binário uma lista contando que segmentos do programa são endereços para ser realocados.

Realocação durante o carregamento não resolve o problema da *proteção*. Pelo fato de programas operarem endereços absolutos de memória, não existe maneira de proibir um programa de ler ou gravar em qualquer posição de memória. Em sistemas multi-usuários é indesejável permitir que processos leiam e escrevam em posições de memória alocadas a outros processos.

[1]Isto é, o programa principal, subrotinas escritas pelo usuário, e bibliotecas são combinados dentro de um espaço de endereçamento único.

A solução de proteção adotada pela IBM na família 360 foi dividir a memória em blocos de 2K bytes e atribuir um código de proteção de 4 bits para cada bloco. A cada processo é atribuído um código único de 4 bits, gravado também nos blocos de memória que ele ocupa. Este código é parte do registro PSW *(program status word)* quando o processo tem a posse da CPU. O *hardware* protege qualquer tentativa de programa em execução de acessar a memória cujo código de proteção difere daquele presente na PSW. Desde que somente o sistema operacional pode mudar os códigos de proteção dos blocos de memória e dos processos, processos do usuário estão protegidos de interferências entre si e com o sistema operacional.

Uma solução alternativa, adotada por praticamente todos os microprocessadores atuais, para realocação e proteção é equipar a máquina com dois registradores especiais no *hardware*, chamados de registradores de *base* e *limite*. Quando um processo é escalonado, o registrador de base é carregado com o endereço do começo da sua partição, e o registrador limite é carregado com o tamanho da partição. Cada endereço de memória referenciado tem o conteúdo do registrador de base a ele adicionado antes de ser enviado para o barramento de acesso à memória.

Por exemplo, se o registrador de base for 100K, uma instrução CALL 100 é efetivamente modificada para CALL (100K + 100). Endereços são comparados com o registrador de limite para prevenir endereçamento fora do espaço alocado ao processo. O *hardware* também protege os registradores de base e limite para evitar que programas dos usuários os modifiquem.

Uma vantagem adicional do uso de registrador de base para realocação é que um programa pode ser movido na memória após ter sua execução iniciada. Depois de ter sido movido, tudo que se precisa para torná-lo pronto para execução em outra posição da memória é mudar o valor do registrador de base. Quando a realocação é feita por alteração dos endereços do programa quando o mesmo é carregado, sua execução em outra posição de memória demanda que todos os endereços sejam novamente recomputados.

4.2 Permuta (Swapping)

Em um sistema operando em *batch*, a organização da memória em partições fixas é simples e efetiva. Quanto mais *jobs* estiverem na memória, mais a CPU estará ocupada e não há razão para usar algo mais complicado. Com *time-sharing*, a situação é diferente: há normalmente mais usuários que memória para armazenar os seus processos, sendo então necessário mover temporariamente processos para disco. Obviamente, para continuar sua execução, um processo movido para disco deve ser trazido novamente para a memória.

4.2.1 Multiprogramação com Partições Variáveis

Em princípio, um sistema que utiliza *swapping* pode ser baseado em partições fixas. Sempre que um processo é bloqueado, ele pode ser movido para o disco e um outro processo trazido do disco para a sua partição em memória. Na prática, partições fixas são pouco atrativas quando a área de memória é escassa pois muita memória é perdida por programas muito menores que o tamanho da partição. Assim sendo, um novo sistema de gerenciamento de memória foi desenvolvido, chamado gerenciamento com *partições variáveis*.

Quando partições variáveis são usadas, o número e tamanho de processos na memória varia dinamicamente. A Fig. 4.4 mostra como partições variáveis são implementadas. Inicialmente, somente o processo A está na memória. Então o processo B e C são criados ou trazidos do disco. Na Fig. 4.4(d) o processo A termina ou é movido para o disco. Então o processo D inicia e B termina. Finalmente, o processo E inicia.

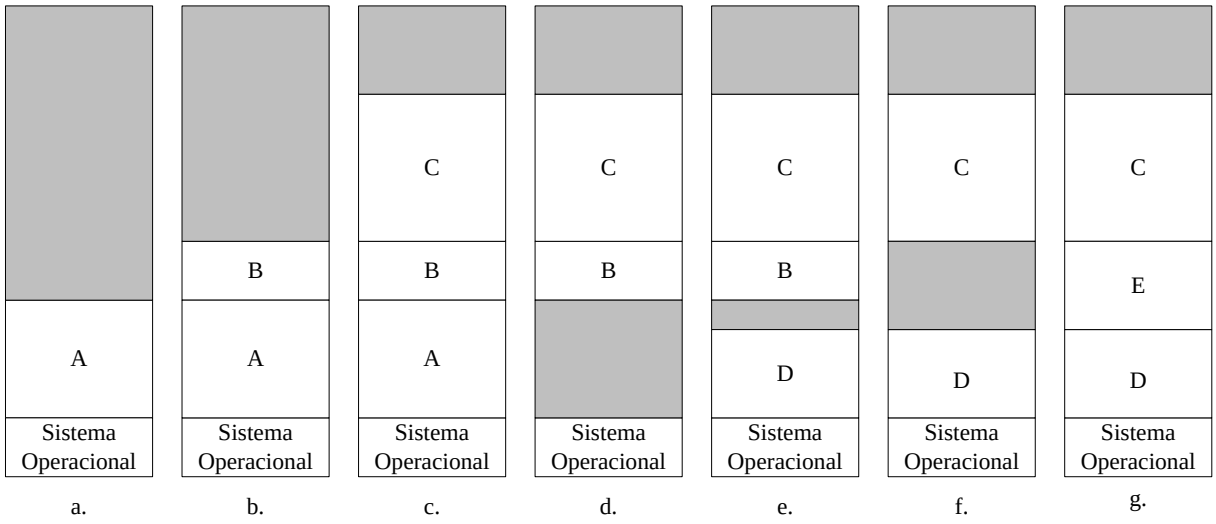

Fig. 4.4: Mudanças na alocação de memória com processos chegando e deixando a memória (regiões sombreadas representam espaço livre)

A principal diferença entre partições fixas da Fig. 4.3 e partições variáveis da Fig. 4.4 é que o número, a localização e o tamanho das partições variam dinamicamente ao longo do tempo. A flexibilidade de não se ter um número fixo de partições aumenta a utilização da memória, mas também complica a tarefa de alocar e liberar a memória, bem como gerenciá-la.

É possível combinar todos os espaços livres disjuntos em um único espaço livre movendo todos processos para um lado da memória. Está técnica é conhecida como *compactação da memória*. Ela não é empregada com frequência pelo fato de requerer muito tempo de CPU. Por exemplo, um microcomputador com 64M bytes de memória e que pode copiar 32 bytes por μs (32 megabyte/seg), gasta 2 seg para compactar toda a memória. Certos *mainframes* utilizam *hardware* especial para a compactação da memória.

Um ponto negativo neste método é saber o quanto de memória alocar para um processo. Se os processos são criados com um tamanho fixo que permanece constante ao longo de sua execução, então a alocação é simples: aloca-se exatamente o necessário ao tamanho do processo.

Na prática, os segmentos de dados e pilha de um processo tendem a crescer durante a sua execução. Alocação dinâmica de memória e recursão (presentes em praticamente em todas as linguagens modernas de programação) são exemplos típicos de crescimento destes segmentos. Se o processo necessitar expandir sua memória e existir um espaço livre adjacente, simplesmente o espaço livre pode vir a ser incorporado ao espaço de endereçamento do processo. De outra forma, se o processo está adjacente a outro processo, o primeiro deverá ser movido para um espaço livre grande o suficiente para armazena-lo, ou um ou mais processos terão que ser movidos para disco com o intuito de criar espaço na memória. Se o processo não puder crescer na memória e a área do disco reservada para abrigar processos permutados estiver cheia, o processo deve ser terminado.

Se for esperado que muitos processos crescerão na memória quando executados, uma boa política seria alocar uma pequena área extra toda vez que o processo é permutado ou movido. Contudo, quando os processos são permutados para o disco, somente a área de memória atualmente em uso deve ser copiada, sendo desnecessário permutar a área extra de memória. A Fig. 4.5(a) mostra a configuração da memória na qual a área para crescimento foi alocada para os dois processos.

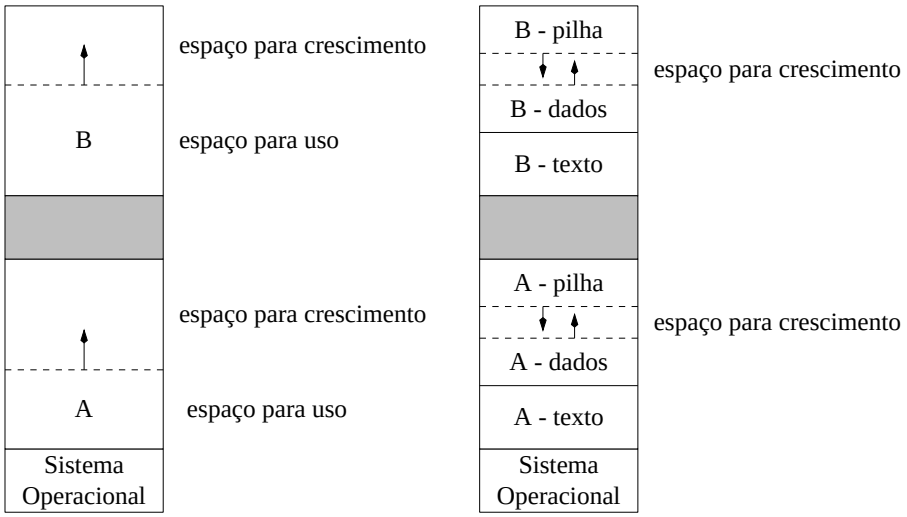

Fig. 4.5: (a) Espaço para crescimento do segmento de dados. (b) espaço para crescimento da pilha e do segmento de dados.

4.2.2 Gerenciamento de Memória com Mapa de Bits

Com um mapa de bits, a memória é dividida em unidades de alocação, desde um pequeno número de palavras até muitos Kbytes. Para cada unidade de alocação existe um bit no mapa de bits, que é 0 se a unidade estiver livre e 1 caso esteja ocupada (ou vice-versa). A Fig. 4.6 mostra parte da memória e o correspondente mapa de bits.

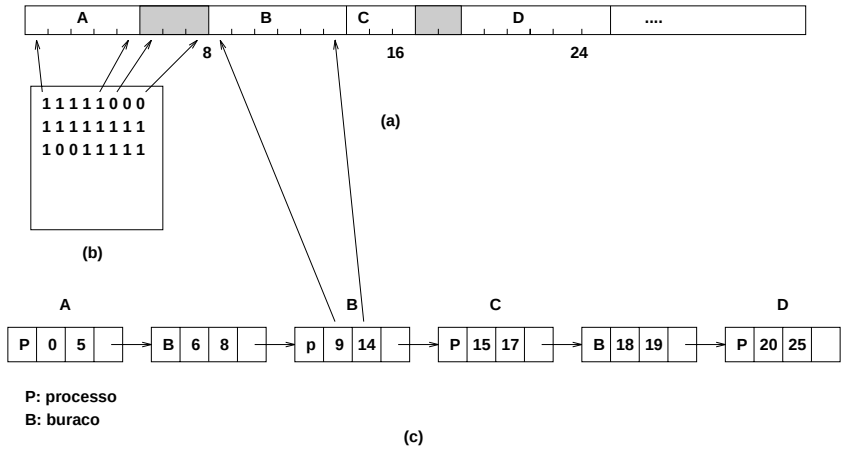

Fig. 4.6: (a) Parte da memória com 5 processos e 3 espaços livres (as marcas mostram as unidades de alocação da memória e as regiões sombreadas estão livres); (b) Mapa de bits correspondente. (c) A mesma informação como uma lista ligada

O tamanho de cada unidade de alocação é uma importante característica de projeto. Para pequenas unidades de alocação tem-se um mapa de bits maior. Entretanto, mesmo com uma unidade de alocação tão pequena como com 4 bytes, 32 bits de memória irão requerer somente 1 bit no mapa (3% da memória). Se a unidade de alocação for grande, o mapa de bits será pequeno, mas memória considerável pode ser desperdiçada se o tamanho do processo não for um múltiplo exato da unidade de alocação.

Um mapa de bits (ocupando uma porção fixa da memória) provê uma maneira simples de gerenciar memória, uma vez que o tamanho do mapa de bits depende somente do tamanho da

memória e do tamanho da unidade de alocação. O problema principal é que quando se decide trazer um processo de k unidades de alocação para a memória, o gerenciador de memória deve pesquisar no mapa de bits uma sequência de k consecutivos bits 0 no mapa. Esta operação é lenta, razão pela qual os mapas de bits são evitados na prática.

4.2.3 Gerenciamento de Memória com Listas Encadeadas

Outra maneira de gerenciar a memória é manter uma lista de alocações e segmentos de memória livre, onde um segmento é um processo ou um espaço livre entre dois processos. A memória da Fig. 4.6(a) é representada na mesma Fig. (c) como uma lista encadeada de segmentos. Cada entrada da lista especifica um espaço livre (L) ou um processo (P), contendo o endereço onde começa, o tamanho, e um ponteiro para a próxima entrada da lista.

Neste exemplo, a lista de segmentos é mantida ordenada por endereços. A vantagem desse método é que quando o processo termina ou é movido para o disco, a atualização da lista é direta. Na finalização de um processo, ao seu lado teremos processos ou espaços livres. Quatro situações podem ocorrer como mostradas na Fig. 4.7.

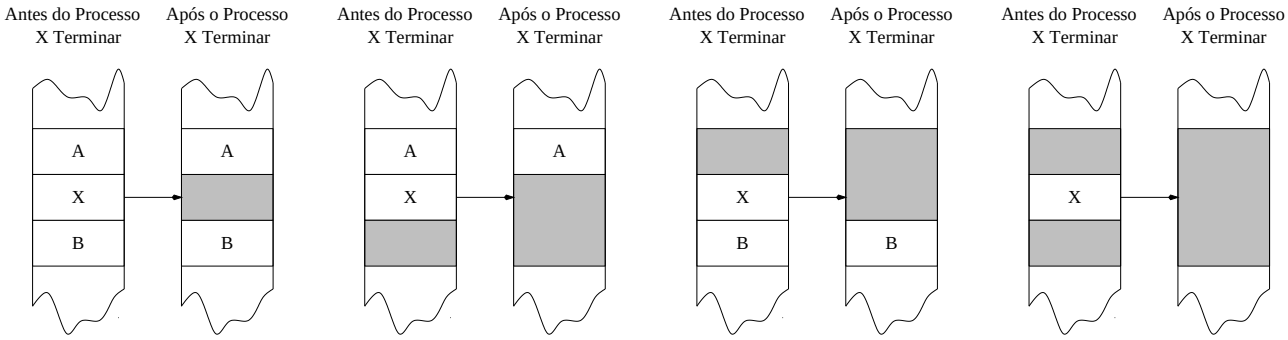

Fig. 4.7: Quatro combinações de memória quando um processo terminar

Quando processos e espaços livres são mantidos na lista ordenada por endereços, vários algoritmos podem ser usados para alocar memória, a fim de criar ou permutar processos. Tais algoritmos são evocados quando o gerenciador de memória necessita um segmento de memória de M bytes.

Algoritmo First-fit

É o algoritmo mais simples. O algoritmo procura ao longo da lista de segmentos até encontrar um espaço livre de tamanho maior ou igual a M. Caso o espaço livre tenha tamanho superior a M *(N)*, o espaço livre é quebrado em dois segmentos: um para o processo (de tamanho M) e o outro para a memória não usada (de tamanho N - M). *First-fit* é um algoritmo rápido pois finaliza a busca o mais cedo possível.

Algoritmo Next-fit

Este algoritmo opera da mesma forma que o *first-fit*, exceto que guarda a posição da lista onde o último espaço livre foi alocado. Da próxima vez que é chamado, o algoritmo começa a procurar a partir deste ponto.

Algoritmo Best-fit

Este algoritmo procura pela lista inteira e toma o espaço livre de tamanho mais próximo de M. É um algoritmo lento e cria na memória espaços livres pequenos que dificilmente serão alocados. Entretanto, para M grande, *best-fit* aumenta as chances de se encontrar na lista um espaço livre de tamanho adequado, posto que minimiza o uso espaços livres grandes para atender requisições pequenas.

Como um exemplo, considere a Fig. 4.6. Se um bloco de tamanho 2 for solicitado, o algoritmo *first fit* alocará o espaço livre 5, e o *best fit* o espaço livre 18.

Algoritmo Quick-fit

Este algoritmo mantém listas separadas para tamanhos comumente requeridos. Por exemplo, seja uma tabela com n entradas, na qual a primeira é um ponteiro para a cabeça da lista de espaços livres de tamanho 4K, a segunda é um ponteiro para a cabeça da lista de espaços livres de tamanho 8K, a terceira de tamanho 12K, e assim sucessivamente. Com o *quick-fit*, acha-se um espaço livre de tamanho requerido muito rapidamente, mas com a desvantagem de todos os esquemas de classificar os espaços livres por tamanho, a saber, quando um processo termina ou é permutado para disco, determinar seus vizinhos para uma possível fusão é uma operação custosa. Se fusões não forem feitas, a memória rapidamente se fragmentará em um grande número de pequenos espaços livres não utilizáveis.

Todos os quatro algoritmos podem aumentar seus respectivos desempenhos mantendo-se em separado listas para processos e espaços livres. Neste caso, todos devotam suas energias para inspeção de espaços livres, não de processos. O preço pago por esse aumento de velocidade na alocação é uma complexidade adicional e diminuição de velocidade quando se trata de liberar memória, uma vez que um segmento livre tem de ser removido da lista de processos e inserido na lista de espaços livres. Novamente, a ineficiência está em se determinar possíveis fusões.

4.2.4 Alocação de Espaço Para Permuta

Os algoritmos apresentados acima mantém o rastreamento da memória principal. Assim, quando processos são permutados do disco para a memória, o sistema pode alocar espaço em memória para eles. Em alguns sistemas, quando um processo está na memória, nenhum espaço em disco lhe é reservado. Quando for movido da memória, espaço deve ser alocado na área de disco para abrigá-lo (portanto, em cada troca o processo pode residir em lugares diferentes no disco). Neste caso, os algoritmos para gerenciamento de espaço para permuta são os mesmos usados para gerenciamento da memória principal.

Em outros sistemas, quando um processo é criado, um espaço para permuta é alocado em disco (usando um dos algoritmos descritos acima). Sempre que um processo em memória dá lugar a outro processo, o mesmo é posicionado no espaço em disco a ele previamente alocado. Quando um processo termina, o seu espaço para permuta em disco é desalocado. Esta técnica é mais eficiente que a anterior pois uma única alocação de espaço em disco por processo é necessária (lembre-se que um processo pode ser permutado várias vezes durante a sua execução). Entretanto, uma área maior de disco deve ser reservada para *swapping*.

4.3 Memória Virtual

Desde o início da informática, o tamanho dos programas vem superando a quantidade de memória disponível para abrigá-los. A solução usualmente adotada era dividir o programa em partes, denominados *overlays*. O *overlay* 0 começa a ser executado primeiro. Quando termina, o *overlay* seguinte é executado, e assim por diante. Alguns sistemas com *overlay* eram relativamente complexos, permitindo múltiplos *overlays* na memória por vez. *Overlays* eram mantidos em disco e permutados para a memória pelo sistema operacional.

Overlays apresentavam um problema: a partição do programa era deixada a cargo do programador. Um método que foi desenvolvido para deixar o particionamento do programa a cargo do sistema operacional veio a ser conhecido como *memória virtual*. A idéia básica é que a combinação do tamanho do programa, dados e pilha, pode exceder a quantia de memória física disponível. O sistema operacional mantém aquelas partes do programa correntemente em uso na memória principal e o resto no disco. Por exemplo, um programa de tamanho 10M bytes pode ser executado em uma máquina que aloca 1M bytes de memória por processo, escolhendo-se cuidadosamente quais dos 1M será mantido na memória a cada instante, com segmentos sendo permutados entre disco e memória assim que forem necessários.

Memória virtual está intimamente relacionada com multiprogramação. Por exemplo, oito programas de 10M cada podem ser alocados em partições de 2M em uma memória de 16M, com cada programa operando como se tivesse sua própria máquina virtual de 2K. Enquanto um programa está esperando que parte de seu espaço de endereçamento seja trazido à memória, o programa é bloqueado, aguardando E/S. Até que a operação de E/S seja completada, a CPU pode ser direcionada para outro processo.

4.3.1 Paginação

A maioria dos sistemas com memória virtual usa uma técnica chamada *paginação*. Em qualquer computador existe certo conjunto de endereços de memória que programas podem referenciar. Quando um programa usa uma instrução como *MOVE REG,1000*, ele está movendo o conteúdo do endereço de memória 1000 para o registrador REG (ou vice versa, dependendo do computador). Endereços podem ser gerados usando indexação, registradores base, registradores de segmento, dentre outras maneiras.

Estes endereços gerados pelos programas são chamados *endereços virtuais* e formam o *espaço virtual de endereçamento* do processo. Em computadores sem memória virtual, o endereço virtual é colocado diretamente no barramento de memória e causa uma palavra da memória física com mesmo endereço ser lida ou escrita. Quando memória virtual é usada, os endereços de memória não vão diretamente para o barramento de memória. Ao invés disso, eles vão à unidade de gerenciamento de memória *(Memory Management Unit, MMU)*, onde um *hardware* específico mapeia os endereços virtuais nos endereços da memória física como ilustrado na Fig. 4.8.

Um exemplo de como este mapeamento se processa é mostrado na Fig. 4.9. Neste exemplo, temos um computador que pode gerar endereços de 16 bits, de 0 até 64K. Estes são os endereços virtuais. Este computador, entretanto, tem somente 32K de memória física, assim, embora programas de 64K possam ser escritos, eles não podem ser carregados para a memória na sua totalidade para serem executados. Uma cópia completa do programa deve estar presente no disco e segmentos do programa podem ser trazidos para a memória pelo sistema a medida que se tornem necessários.

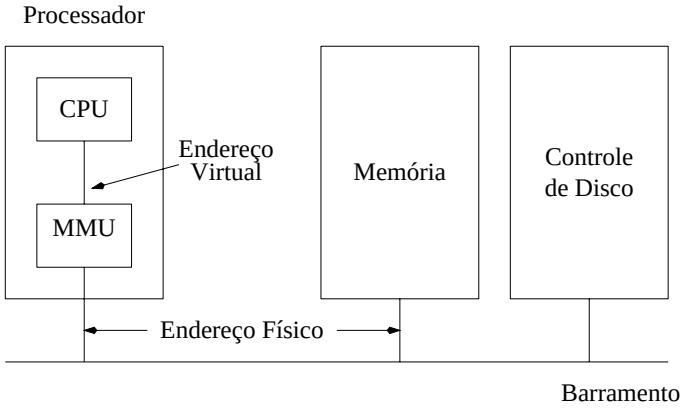

Fig. 4.8: A posição e função da MMU

O espaço de endereçamento virtual é dividido em unidades chamadas *páginas*. As unidades correspondentes na memória física são chamadas *page frames*. As páginas e *page frames* são sempre do mesmo tamanho. Neste exemplo elas são de 4K, mas tamanhos de páginas de 512 bytes, 1K, e 2K são comumente usados. Com 64K de espaço de endereço virtual e 32K de memória física, temos 16 páginas e 8 *page frames*. Transferências entre memória e disco são sempre feitas em unidades de páginas.

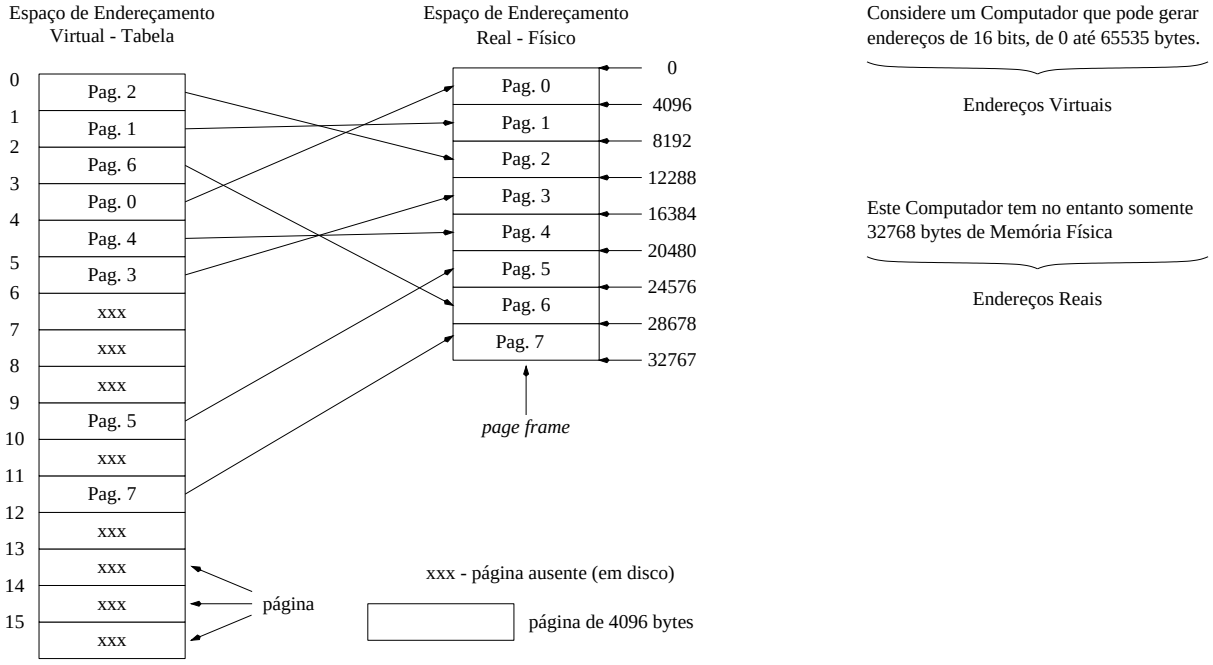

Fig. 4.9: Relação entre endereço virtual e endereço físico de memória, dada pela *tabela de páginas*

Quando o programa tenta acessar o endereço 0, por exemplo, usando a instrução MOV REG,0 o endereço virtual 0 é enviado para a MMU. Ela reconhece que este endereço cai na página 0 (0 a 4095), o qual, de acordo com seu mapeamento é a *page frame* número 2 (8192 até 12287). Ele então transforma o endereço para 8192 e coloca o endereço 8192 no barramento. A tabela de memória nada sabe a respeito da MMU, e apenas vê uma requisição para leitura ou escrita no endereço 8192, a qual é respeitada. Assim, a MMU mapeou todo endereço virtual

entre 0 e 4095 em endereço físico de 8192 a 12287.

O que acontece se o programa tenta usar um página não mapeada? Como exemplo, seja a instrução MOV REG,32780, o qual referencia o byte número 12 dentro da página virtual 8 (começando em 32768). A MMU observa que a página está sem mapeamento (indicada por um *X* na Fig. 4.9), e força a CPU a causar uma interrupção (*trap*) no sistema operacional. Este *trap* é chamado uma falta de página (*page fault*). O sistema operacional remove o *page frame* menos usado e escreve seu conteúdo de volta no disco. Ele então busca a página referenciada para o *page frame* liberado, atualiza o mapa, e retorna à instrução interrompida.

Agora vejamos como a MMU trabalha e porque temos de escolher um tamanho de página como uma potência de 2. Na Fig. 4.10 temos um exemplo de um endereço virtual, 8196 (0010000000000100 em binário), sendo mapeado usando o mapa da MMU da Fig. 4.9. O endereço virtual de 16 bits é divido em um número de página de 4 bits e offset de 12 bits dentro da página. Com 4 bits para o número da página, podemos representar 16 páginas, e com 12 bits para o deslocamento *(offset)*, podemos endereçar todos os 4096 bytes dentro da uma página.

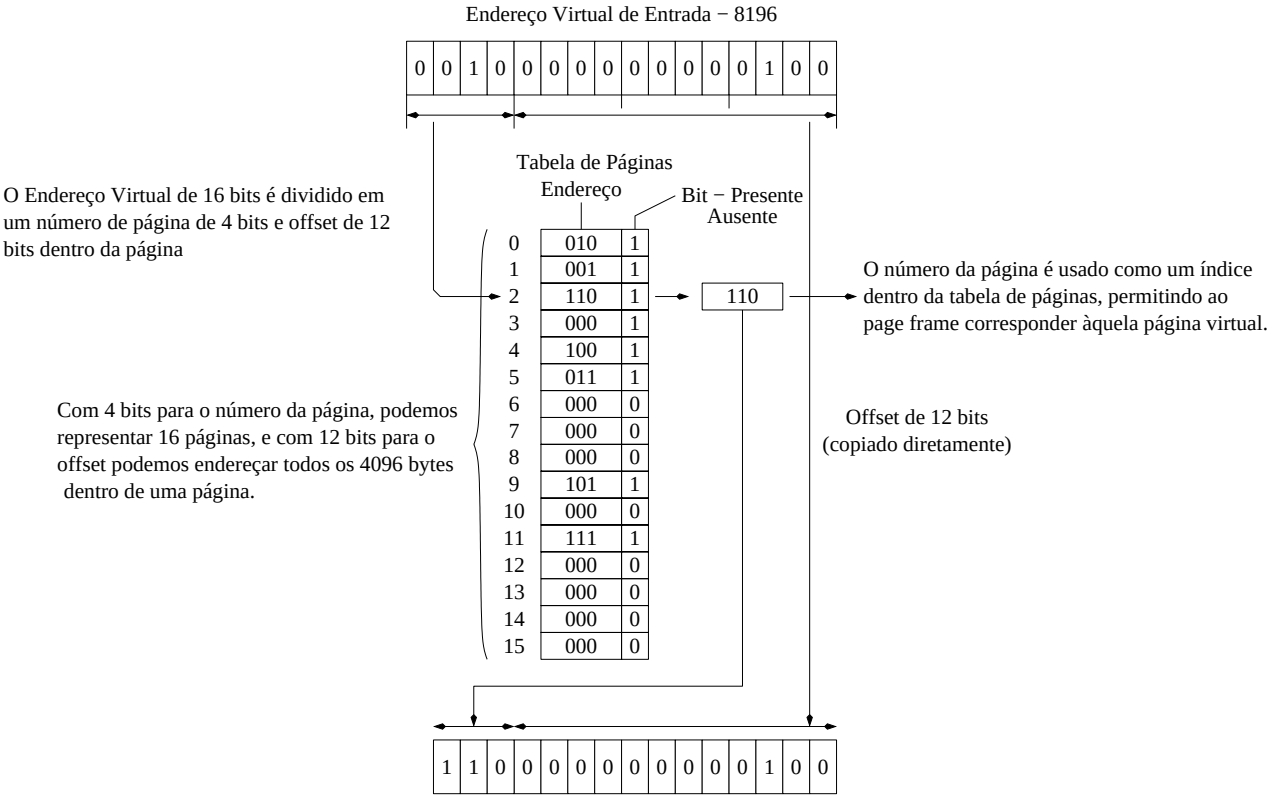

Fig. 4.10: Operação interna da MMU com 16 páginas de 4K

O número da página é usado como um índice dentro da tabela de páginas, permitindo ao *page frame* corresponder àquela página virtual. Se o bit ausente/presente for 0, uma interrupção é causada. Se for 1, o número do *page frame* encontrado na tabela de páginas é copiado para os 3 bits[2] de mais alta ordem do registrador de saída, juntamente com os 12 bits do *offset*, o qual é copiado sem ser modificado dentro do endereço virtual de entrada. O registrador de saída é então colocado no barramento de memória como endereço de memória física. Note que no procedimento descrito acima, entra em endereço virtual de 16 bits (0 - 64K) e sai um endereço físico de 15 bits (0 - 32K).

[2]Apenas 3 bits porque a memória física tem apenas 8 *page frames*.

4.3.2 Segmentação

Iniciada com o projeto MULTICS, a idéia de memória segmentada sobrevive até hoje. Uma implementação típica provê suporte de *hardware* para até 16 processos, cada um com espaço de endereçamento virtual de 1K páginas de 2K ou 4K. Se introduzirmos páginas de 4K para este exemplo, cada processo terá um espaço de endereçamento virtual de 4M, consistindo de 1024 páginas de 4K cada.

Este esquema poderia ser implementado dando a cada processo sua própria tabela com 1024 números de *page frames*. Entretanto, esta técnica raramente é empregada. Em vez disto, o *hardware* da MMU contém uma tabela com 16 seções, uma para cada um dos 16 processos. Cada seção tem 64 descritores de segmento, então o espaço de endereçamento para cada processo de 4M é dividido em 64 segmentos, cada um contendo 16 páginas de 4K. As tabelas de segmento e página são descritas na Fig. 4.11(a).

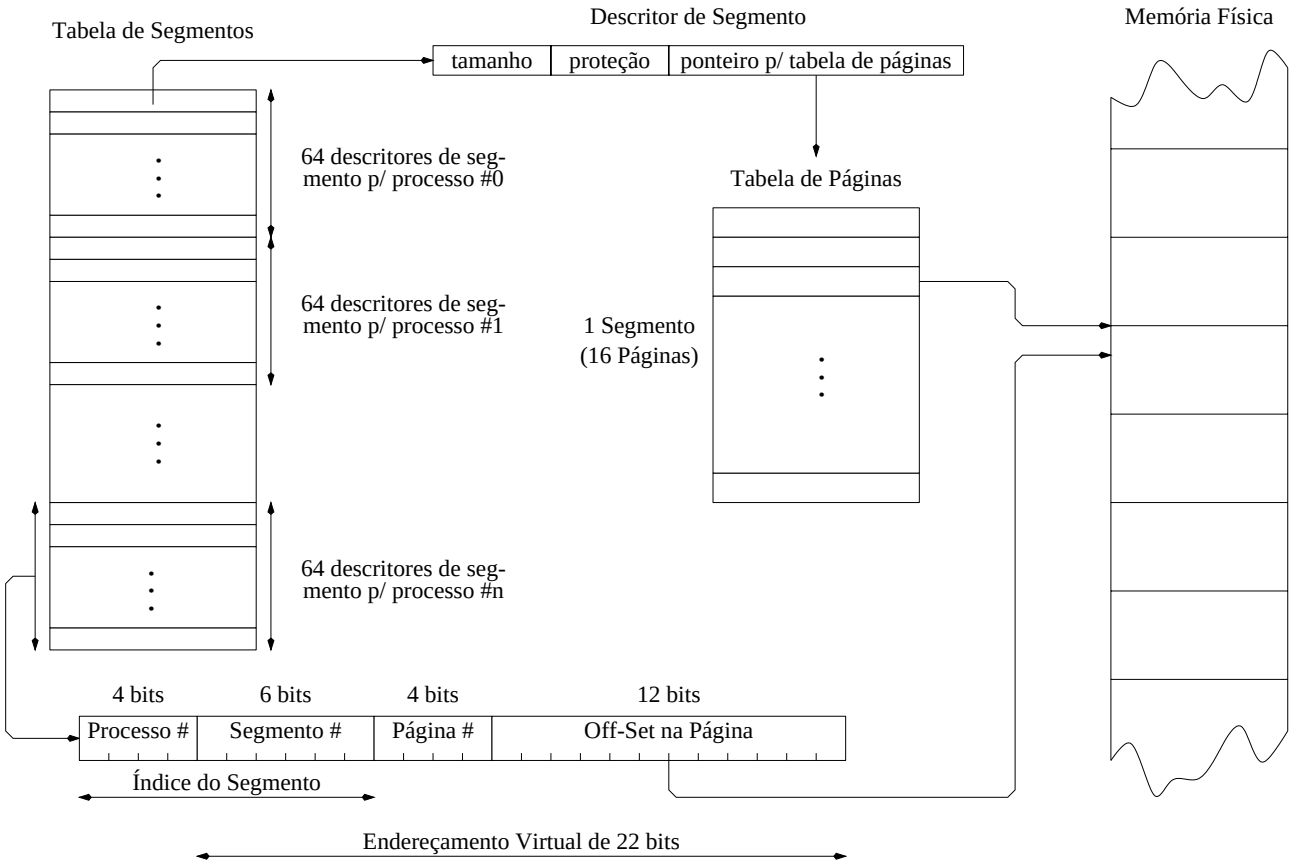

Fig. 4.11: (a) MMU usada em muitos computadores baseados no 68000; (b) endereçamento virtual para um sistema de 4M

Cada um dos descritores de segmento contém um tamanho de segmento (0 a 16 páginas), bits de proteção informando se um segmento pode ser lido ou escrito, e um ponteiro para a própria tabela de páginas. Cada uma das tabelas de páginas contém 16 entradas, cada entrada apontando para uma *page frame* na memória (guardando o número da *page frame*).

Quando o sistema operacional começa um processo, ele carrega um número de 4 bits do processo em um registrador especial do *hardware*. Sempre que o processo referencia a memória, a MMU traduz o endereço virtual como se segue. Toma-se o número de 4 bits do processo e os 6 bits de mais alta ordem dos 22 que compõem o endereço virtual (necessários para o endereço

de 4M), combinando-os em um número de 10 bits usados para indexar a tabela de segmentos e localizar o descritor de segmento relevante. Note que existem exatos $2^{10} = 1024$ segmentos na tabela: 64 segmentos/processo multiplicados por 16 processos.

Verifica-se então os bits de proteção no descritor de segmentos para ver se o acesso é permitido. Se o for, a MMU então verifica o número da página extraída do endereço virtual com o tamanho do segmento no descritor de segmentos, para verificar se o segmento é grande o bastante. Caso seja, o número da página é usado como um índice para a tabela de páginas, cujo endereço é fornecido no descritor de segmento (todas tabelas de páginas são mantidas em uma memória rápida especial dentro da MMU). Uma vez que o número da estrutura de página é encontrado, ele é combinado com o deslocamento do endereço virtual para formar o endereço físico da memória, o qual é então enviado ao barramento.

Uma das características chave deste modelo é que quando o sistema operacional escolhe um processo, apena o registrador do número de 4 bits do processo é alterado. Isto faz com que não tenha que recarregar todas as tabelas de segmento ou de página. Dois ou mais processos podem dividir um segmento se seus descritores de segmentos apontam para a mesma tabela de páginas. Qualquer mudança feita por um processo em uma página deste segmento é automaticamente visível para os outros processos.

Enquanto o modelo da Fig. 4.11 provê cada um dos 16 processos com somente 64 segmentos de 64K cada, a idéia pode facilmente ser estendida para espaços de endereço maiores ao preço de uma tabela maior dentro da MMU. A maior parte do espaço é para as tabelas de páginas. Se tivéssemos quatro vezes mais memória para as tabelas de páginas dentro da MMU e seus conteúdos com 4 processos em vez de 16, o MMU pode-riamos suportar 64 segmentos de 1M por processo, por exemplo.

4.4 Algoritmos de Troca de Página

Quando uma falta de página ocorre, o sistema operacional tem que escolher uma página para remover da memória a fim de dar lugar à que será trazida do disco. Se a página a ser removida foi modificada enquanto estava na memória, ela deve ser reescrita no disco para manter a cópia em disco atualizada. Se, todavia, a página não tiver sido alterada, a cópia do disco já está atualizada, não sendo necessária sua reescrita. A página a ser lida é simplesmente superposta à página retirada.

Apesar de ser possível escolher uma página aleatória para dar lugar à página em demanda, o desempenho do sistema é melhorado se for escolhida uma página pouco usada (referenciada). Se uma página muito usada é removida, ela provavelmente terá de ser trazida de volta em breve, resultando um esforço adicional. Algoritmos eficientes de troca de página visam minimizar este esforço.

4.4.1 Troca Ótima de Página

O melhor algoritmo de troca de páginas é fácil de descrever mas impossível de implementar. O algoritmo opera da seguinte maneira. No momento que ocorre uma falta de página, um certo conjunto de páginas está na memória. Uma dessas páginas será referenciada em muitas das próxima instruções (a página contendo a instrução). Outras páginas não serão referenciadas antes de 10, 100, ou talvez 1000 instruções. Cada página pode ser rotulada com o número de instruções que serão executadas antes que a página seja inicialmente referenciada.

O algoritmo da página ótima simplesmente diz que a página com o maior rótulo deve ser removida, adiando-se o máximo possível a próxima falta de página.

O único problema com este algoritmo é que ele não é realizável. No momento da falta de página, o sistema operacional não tem como saber quando cada página será referenciada. (Observamos facilmente uma situação similar com o algoritmo *menor job primeiro* - como pode o sistema dizer qual *job* é o menor?). No máximo podemos executar um programa em um simulador e, mantendo uma lista de todas as páginas referenciadas, implementar o algoritmo na segunda execução (usando as informações coletadas na primeira execução).

Nesta linha, é possível comparar o desempenho de algoritmos realizáveis como o melhor possível. Se um algoritmo apresenta um desempenho de, digamos, somente 1% pior que o ótimo, o esforço gasto no aprimoramento do algoritmo produzirá, no máximo, um aumento de 1% no desempenho deste (para os casos estudados, obviamente).

4.4.2 Troca da Página Não Recentemente Usada (NRU)

Para permitir que o sistema operacional colete estatísticas sobre quais páginas estão sendo usadas e quais não estão, muitos computadores com memória virtual têm 2 bits associados à cada página. Um bit, R ou *bit de referência*, é ativado pelo *hardware* em qualquer leitura ou escrita de página. O outro bit, M ou *bit de modificação*, é ativado pelo *hardware* quando uma página é escrita (i.e., um byte é alterado/adicionado na página). É importante que estes bits sejam atualizados em qualquer referência de memória, assim, é essencial que eles sejam ativados pelo *hardware*. Uma vez que um bit foi ativado, ele permanece ativado até que o sistema operacional o desative (por software).

Se o *hardware* não dispõe dos bits R e M, eles podem ser simulados como se segue. Quando um processo é iniciado, todas as suas entradas de tabela de páginas são marcadas como ausentes na memória. Tão logo uma página seja referenciada, uma falta de página ocorrerá. O sistema operacional então ativa o bit R (em sua tabela interna), muda a entrada de tabela de páginas para apontar para a página correta, com modo *READ ONLY*, e reinicia a instrução. Se a página é subsequentemente escrita, uma outra falta de página ocorrerá, permitindo ao sistema operacional ativar o bit M e mudar o modo da página para *READ/WRITE*.

Os bits R e M podem ser usados para construir um algoritmo de paginação simples como se segue. Quando um processo é iniciado, ambos os bits de página para todas estas páginas são declarados 0 pelo sistema operacional. Periodicamente (i.e., a cada interrupção de relógio), o bit R é zerado, para distinguir páginas que não foram referenciadas recentemente daquelas que tenham sido.

Quando uma falta de página ocorre, o sistema operacional examina todas as páginas e as classifica em 4 categorias baseado nos valores correntes de seus bits R e M:

- Classe 0 : não referenciada, não modificada.
- Classe 1 : não referenciada, modificada.
- Classe 2 : referenciada, não modificada.
- Classe 3 : referenciada, modificada.

Ainda que as páginas na classe 1 pareçam, à primeira vista, impossíveis de existir, elas ocorrem quando as páginas da classe 3 têm seu bit R zerado pela interrupção do relógio.

Interrupções de relógio não zeram o bit M porque esta informação é necessária para determinar se uma página terá que ser reescrita no disco ou não.

O algoritmo *Não Recentemente Usada (Not Recently Used, NRU)*, remove uma página aleatória da classe não vazia de numeração mais baixa. Implícito neste algoritmo é que é melhor remover uma página modificada que não foi referenciada pelo menos no último *tick* de relógio (tipicamente 20 mseg), que uma página não modificada mas muito usada. As características principais do NRU é que ele é fácil de entender, eficiente de se implementar, e gera um desempenho que, enquanto certamente não ótimo, é geralmente tido como adequado.

4.4.3 Troca da Página FIFO

O algoritmo de paginação *First-In-First-Out (FIFO)* é similar ao NRU. O sistema operacional mantém uma lista de todas as páginas correntes na memória, sendo a página da cabeça da lista a mais velha e a página do fim a instalada mais recentemente. Em uma falta de página, a página da cabeça é removida e a nova página acrescentada no fim da lista.

Uma simples modificação no FIFO para evitar o problema da retirada de uma página muito usada é examinar os bits R e M da página mais velha. Se a página pertencer a classe 0 (não referenciada, não modificada), ela é retirada, senão a próxima página mais velha é examinada, e assim por diante. Se páginas da classe 0 não estão presentes na memória, então o algoritmo é repetido procurando por páginas da classe 1, 2 e 3.

Outra variação do FIFO é o algoritmo *Segunda Chance*. A idéia é primeiro examinar a página mais velha como uma vítima potencial. Se seu bit R é 0, a página é trocada imediatamente. Se o bit R é 1, o bit é zerado e a página é colocada no fim da lista de páginas, como se estivesse acabado de chegar à memória. Então a pesquisa continua. O que o *Segunda Chance* faz é procurar por uma página velha que não tem sido referenciada no *tick* de relógio anterior. Se todas as páginas tiverem sido referenciadas, o algoritmo degenera-se e torna-se simplesmente um FIFO.

Uma pequena variação do *Segunda Chance* é guardar todas as páginas em uma lista circular. Em vez de colocar as páginas no fim da lista para dar a elas uma segunda chance, o ponteiro da cabeça da lista é simplesmente avançado uma página, produzindo o mesmo efeito.

4.4.4 Troca da Página Menos Recentemente Usada (LRU)

Uma boa aproximação para o algoritmo ótimo é baseada em uma observação comum que as páginas muito usadas nas últimas instruções, provavelmente o serão nas próximas instruções. Da mesma forma, páginas que não têm sido usadas por um longo tempo provavelmente continuarão sem uso. Esta observação sugere um algoritmo realizável: quando ocorre uma falta de página, retira-se a página que não tem sido usada por um tempo longo. Esta estratégia é chamada de *Menos Recentemente Usada (Least Recently Used - LRU)*.

Embora o algoritmo LRU seja teoricamente realizável, seu custo é alto. Para implementação completa do LRU, é necessário manter uma lista ligada de todas as páginas em memória, com a página mais recentemente usada no início e a menos recentemente usada no final. A dificuldade é que a lista deve ser atualizada em toda referência de memória. Encontrar a página na lista, removê-la de sua posição corrente, e movê-la para o início representa um esfôrço não desprezível.

Manipular uma lista ligada a toda instrução é proibitivo, até mesmo em *hardware*. Entretanto, há outras maneiras de implementar LRU com um *hardware* especial. Vamos considerar o caminho mais simples primeiro. Este método requer equipar o *hardware* com um contador

de 64 bits, C, que é automaticamente incrementado após cada instrução. Além disso, cada entrada na tabela de páginas deve também ter um campo grande o bastante para conter o contador. Após cada referência de memória, o corrente valor de C é armazenado na entrada da tabela de páginas para a página referenciada. Quando ocorre uma falta de página, o sistema operacional examina todos os contadores na tabela de páginas para achar o menor deles. A página correspondente é a menos recentemente usada.

Agora vejamos um segundo algoritmo LRU, também em *hardware*. Para uma máquina com N *page frames*, o LRU deve manter uma matriz de NxN bits, inicialmente todos zero. Quando o *page frame* k é referenciado, o *hardware* primeiro ativa todos os bits da linha k para 1, atribuindo a todos os bits da coluna k o valor 0. Em algum instante, a linha na qual o valor binário é menor, é a menos recentemente usada, a linha na qual o valor é o próximo superior é a segunda menos recentemente usada, e assim por diante.

Simulação do LRU em Software

Embora os algoritmos apresentados sejam realizáveis, eles são dependentes de *hardware* especial, e são de pouco uso para o projetista de sistema operacional construindo um sistema para uma máquina que não dispõe deste *hardware*. Uma solução que pode ser implementada em software faz-se necessária. Uma possibilidade é o algoritmo chamado de *Não Frequentemente Usada (Not Frequently Used - NFU)*. O algoritmo NFU requer um contador em software associado a cada página, inicialmente zero. Em cada *tick* de relógio, o sistema operacional pesquisa todas as páginas na memória. Para cada página, o bit R, que é 0 ou 1, é adicionado ao contador. Em suma, os contadores são uma tentativa de guardar a frequência com que cada página tem sido referenciada. Quando uma falta de página ocorre, a página com o menor contador é escolhida para substituição.

O principal problema com o NFU é que ele nunca esquece referências anteriores. Páginas muito (e não mais) referenciadas no começo da execução de um programa permanecem com um contador alto até o final da execução. Felizmente, uma pequena modificação no NFU faz com que este seja capaz de simular LRU muito bem (o algoritmo modificado é denominado *Aging*). A modificação tem duas partes. Primeiro, os contadores são cada um deslocados 1 bit para a direita antes do bit R ser incrementado. Segundo, o bit R é incrementado no bit mais a esquerda.

Quando ocorre uma falta de página, a página de menor contador é removida. É óbvio que a página que não tenha sido referenciada por, digamos, quatro *ticks* de relógio terá quatro zeros significativos em seu contador, tendo assim um valor mais baixo que o contador de uma página que tenha sido referenciada nos quatro últimos *ticks* de relógio.

Uma diferença entre LRU e *Aging* é que, no último os contadores têm um número finito de bits (tipicamente 8). Portanto, não podemos classificar as páginas segundo referências anteriores à capacidade do contador.

4.5 Gerenciamento de Memória no UNIX

Versões anteriores ao *System V* e ao 4.2 BSD empregavam um esquema de permuta *(swapping)* de processos como política de gerenciamento de memória [3, 2]. Versões atuais empregam o esquema de paginação por demanda, que descreveremos a seguir.

4.5.1 Paginação por Demanda

Este esquema necessita da habilidade do *hardware* de reiniciar uma instrução interrompida pela ausência de página cujo endereço foi referenciado na instrução. Assim que a página é trazida para a memória, a instrução interrompida é reiniciada.

No esquema de paginação por demanda, o espaço virtual de endereçamento é muito superior à quantidade de memória física disponível, sendo limitado apenas pelo capacidade de endereçamento virtual da MMU.

O núcleo mantém 4 estruturas principais para fins de gerenciamento de memória: tabela de páginas, tabela de frames (pfdata), descritores de blocos e tabela de uso de *swap*.

A Tabela de Páginas

A tabela de páginas tem como entrada o número da página. Deve-se notar que esta tabela tem dimensão fixa pois a quantidade de páginas é igual à quantidade física de memória dividida pelo tamanho da página. Cada entrada na tabela possui os seguintes campos:

- endereço físico de memória que contém os dados referentes à esta página;
- idade da página: por quantos ciclos esta página está ativa (na memória);
- COPY-ON-WRITE: flag que indica que esta página está sendo compartilhada para fins de leitura, devendo ser desmembrada caso alguns dos processos que a compartilham altere seu conteúdo;
- modificação: flag que indica se o processo modificou o conteúdo da página recentemente;
- referência: flag que indica se o processo referenciou o conteúdo da página recentemente;
- validade: flag que indica se o conteúdo da página é válido (isto é, o endereço físico guarda o conteúdo da página);
- proteção: indica se o conteúdo da página é do tipo READ ONLY ou READ/WRITE.
- descritor de bloco com os seguintes campos:
  - dispositivo de *swap* com área disponível para ler/gravar o conteúdo da página;
  - número do bloco alocado à página;
  - o tipo da página: *swap*, arquivo executável, *demand fill* e *demand zero* (definidos mais adiante).

A tabela de frames armazena dados adicionais à página:

- endereço físico de memória que contém os dados referentes à esta página;
- um contador de referência indicando quantos processos compartilham esta página em memória;
- o dispositivo de *swap* associado à página;
- número do bloco alocado à página.

Finalmente, a tabela de uso de *swap* é acessada pelo dispositivo de *swap* e número do bloco neste dispositivo. Esta tabela armazena apenas um contador de referência indicando quantas páginas se utilizam deste bloco em disco.

Deve-se notar que algumas informações são replicadas em tabelas distintas. Esta replicação visa beneficiar a eficiência do esquema de paginação, diminuindo o número de consultas às tabelas.

A Fig. 4.12 ilustra uma referência ao endereço virtual 1493K. O *hardware* mapeia este endereço na página número 794. O conteúdo desta página pode estar em memória ou no dispositivo de *swap* #1, bloco 2743. As tabelas de frames e de uso de *swap* mostram seus respectivos contadores de referência em 1, informando que o processo é o único a utilizar esta página tanto em memória como em disco. As tabelas de páginas e a de frames apontam para o endereço físico de memória onde os dados referentes a esta página estão armazenados.

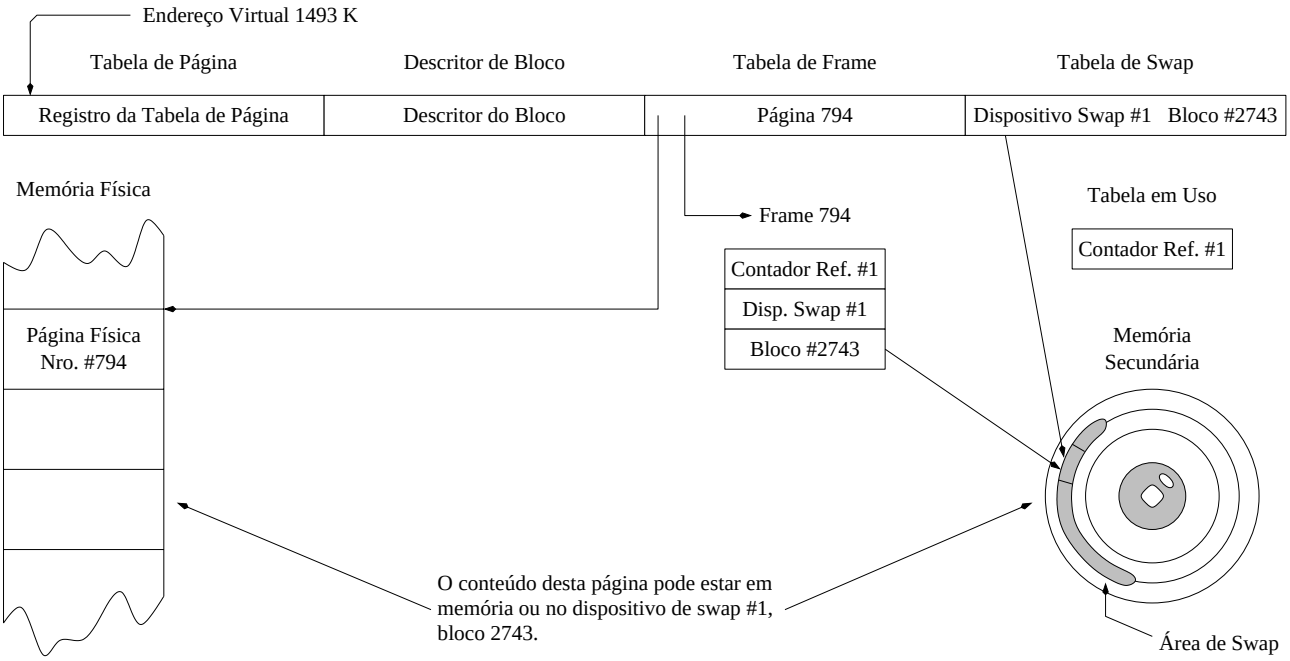

Fig. 4.12: As várias estruturas de dados empregadas para gerenciamento de memória

A Chamada "fork" em um Sistema Paginado

A Fig. 4.13 ilustra a situação imediatamente após uma chamada de sistema `fork` em um sistema paginado. A área de texto não é duplicada tendo tanto o processo pai quanto o filho as mesmas entradas para a tabela de páginas (estas páginas são do tipo READ ONLY). O núcleo duplica as tabelas de páginas que contém as áreas de dados e de pilha. As entradas destas tabelas compartilham as mesmas entradas na tabela de frames (com o contador de referência agora em 2). Todas as entradas na tabela de páginas são marcadas como COPY-ON-WRITE, significando que quando qualquer um dos dois processos alterar o conteúdo da página, a mesma deve ser desmembrada, desvinculando-se os endereços físicos de memória e as entradas na tabela de frames.

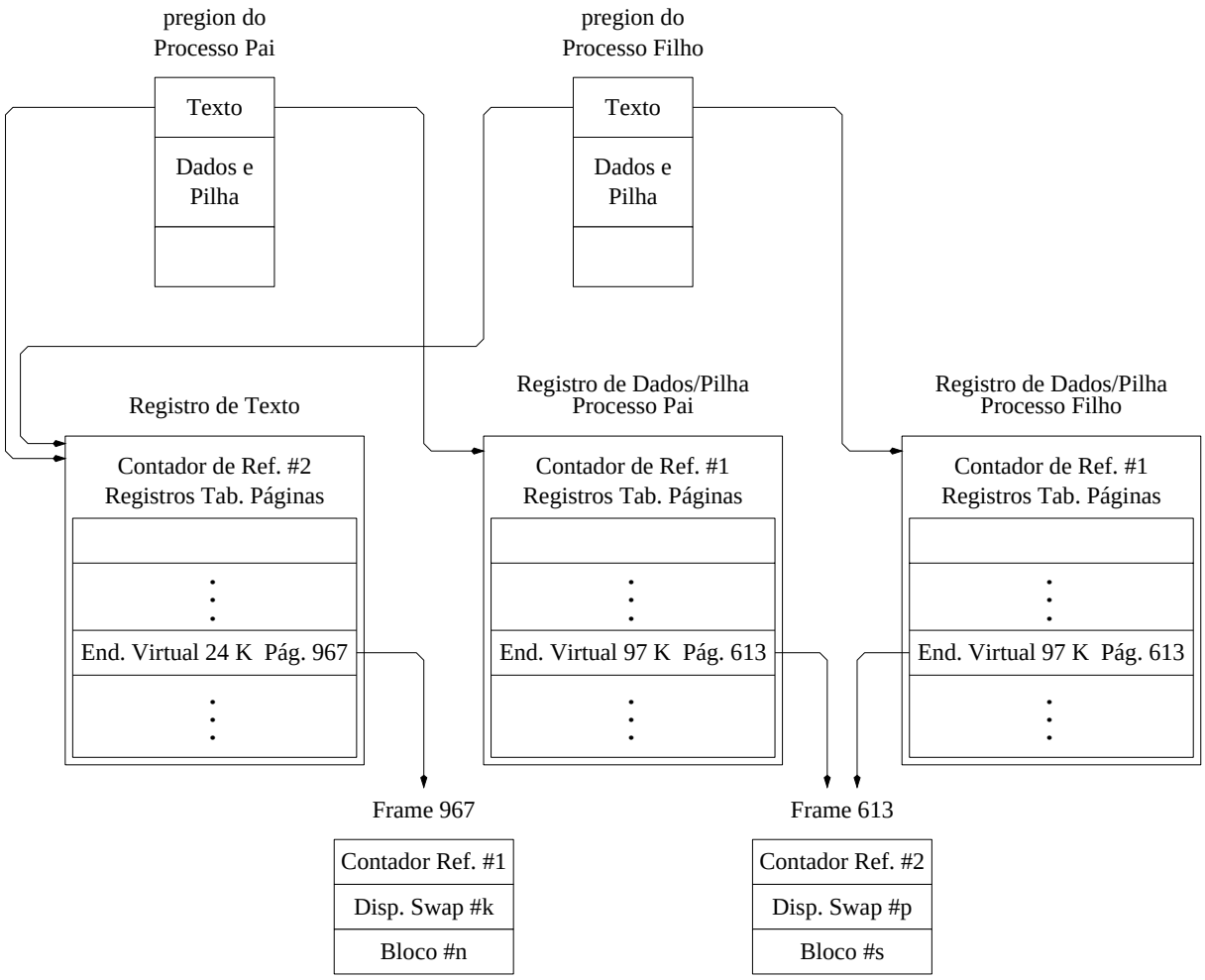

Fig. 4.13: Situação após um `fork` em um sistema paginado

A Chamada "exec" em um Sistema Paginado

Durante uma chamada `exec`, o núcleo carrega o programa executável do disco para a memória. Em um sistema paginado, pode ocorrer que o tamanho do executável supere o tamanho físico da memória. Neste caso, após todo o executável ter sido carregado, parte dele já se encontra no dispositivo de *swap*. Inicialmente, é montado a tabela de páginas (com os respectivos descritores de blocos) para o processo. O sistema conhece o tamanho do executável a priori, informação esta presente no cabeçalho do próprio executável. As páginas são marcadas como *demand zero* para áreas de pilha ou *demand fill* para áreas de texto e dados.

A partir daí, o núcleo começa a cópia das regiões de texto e dados para as páginas em memória. O núcleo aloca uma página para cada registro da tabela de página. Páginas *demand zero* (para pilha) não estão presentes no código executável, sendo simplesmente alocadas e zeradas. Páginas *demand fill* (texto e dados) são copiadas do disco das respectivas porções do executável.

Para copiar diretamente do executável para uma página em memória, o núcleo adiciona ao *inode* (em memória) um vetor de blocos que compõem o executável. O descritor de bloco da tabela de página, nesta fase, armazena o índice do bloco no vetor que contém a porção do executável a ser carregado nesta página. Ao copiar a porção do executável para a página, o núcleo localiza o bloco acessando seu índice no descritor de bloco e seu número na posição

correspondente no vetor (ver Fig. 4.14).

Uma vez copiado a parte do arquivo executável para a memória física alocada à página, o descritor de bloco é atualizado, contendo agora um bloco físico de *swap* associado à página.

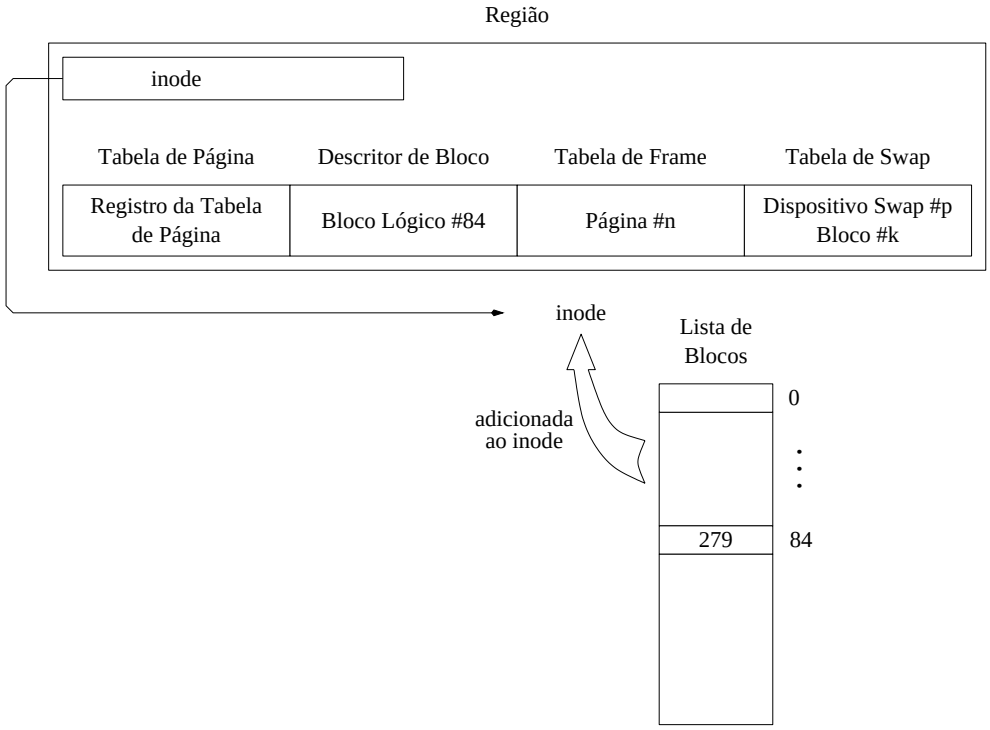

Fig. 4.14: Lista de blocos adicionada ao *inode* durante a carga de um executável

4.5.2 O Processo Paginador

O processo paginador remove da memória páginas não referenciadas pelos processos por um longo tempo. Quando executado, o processo vai montando uma lista de páginas candidatas à permuta (ver Fig. 4.15). O processo paginador zera o bit de referência da página (veja subseção 4.4.2). Uma página é candidata à permuta quando seu contador de referência foi zerado há um determinado número de passadas do processo paginador. Se uma página candidata à permuta é novamente referenciada, a mesma é removida da lista.

O processo paginador é ativado quando a quantidade de memória disponível atinge um limite mínimo. Páginas são então removidas da memória e gravadas em disco até que um limiar de memória livre seja atingido.

Ao gravar uma página em disco, o processo paginador apaga o bit de validade da página e decrementa seu contador de referência na tabela de frames. Se o contador de referência vai a zero (indicando que um único processo estava utilizando a página), o núcleo adiciona o campo da tabela de frames referente à página em uma lista de páginas livres. O conteúdo de uma página na lista de páginas livres continua válido até que o sistema associe a página à outro processo. Mesmo na lista de páginas livres, uma página pode ser revalidada (sem necessidade de trazê-la do disco) caso um processo torne a referenciá-la. Neste caso, a página é removida da lista de páginas livres, tendo seu bit de validade reativado.

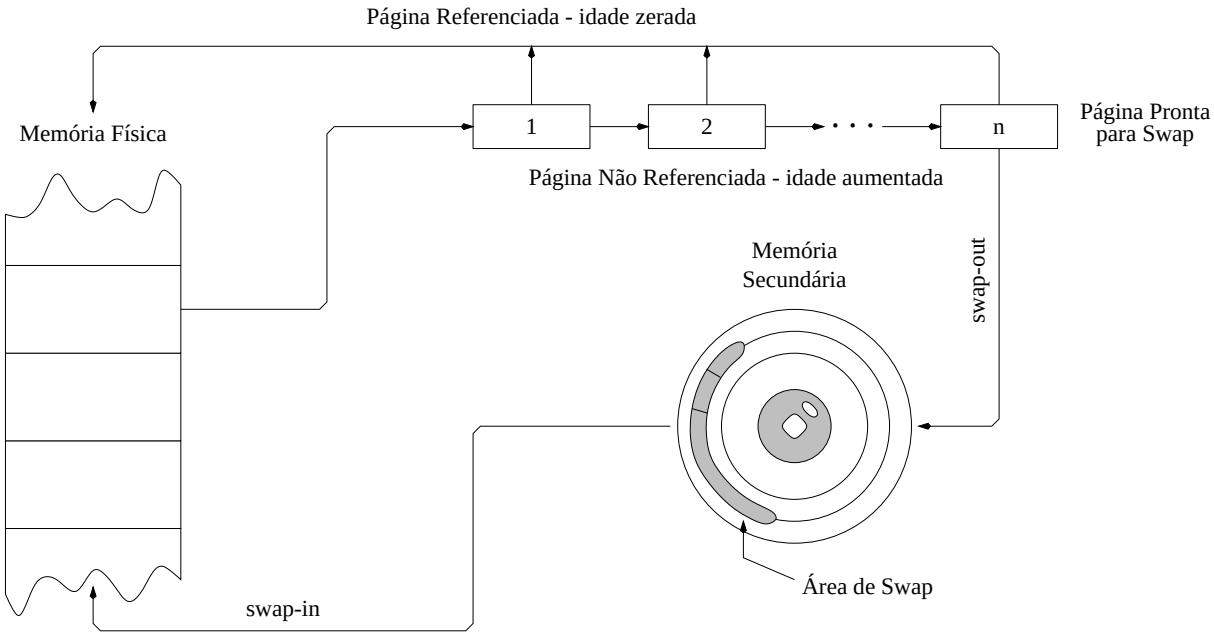

Fig. 4.15: Fila de páginas candidatas a permuta

4.5.3 Falta de Paginação

Quando um processo referencia uma página cujo bit de validade esteja apagado, o processo incorre em uma falta de paginação. O núcleo deve então providenciar a validação da página para que o processo possa continuar sua execução. A página referenciada pode estar em um dos cinco estados abaixo:

1. no dispositivo de swap e não em memória;
2. em memória, na lista de páginas livres;
3. em um arquivo executável sendo carregado;
4. marcada como *demand fill*;
5. marcada como *demand zero*.

No primeiro caso, o sistema deve alocar uma página livre e atualizar o conteúdo da memória com aquele presente no dispositivo de *swap*. O processo é bloqueado até que a operação de E/S se complete.

No segundo caso, se a página desde quando adicionada à lista livre não for associada a nenhum outro processo, a página continua válida, sendo removida da lista de páginas livres sem que nenhuma operação de E/S se faça necessária.

No terceiro caso, o núcleo através do descritor de bloco encontra no vetor de blocos do *inode* aquele que contém a parte do executável sendo requisitado. Uma operação de E/S se faz necessária para trazer o conteúdo do disco para a memória física associada à página. Neste caso, o núcleo também associa à pagina um bloco de *swap*.

No quarto caso, uma página é alocada, sendo seu conteúdo atualizado de forma similar ao caso anterior.

Finalmente, no último caso, uma página é alocada e seu conteúdo de memória simplesmente zerado.

4.5.4 Falta de Proteção

Um segundo tipo de *trap* de memória, além da falta de paginação, corre quando um processo acessa uma página válida, mas os bits de proteção da página impedem o acesso. Dois exemplos desta situação são típicos:

1. O processo tenta escrever em uma página marcada como READ ONLY, por exemplo, na sua própria área de texto.

2. O processo tenta escrever em uma página marcada como COPY-ON-WRITE, por exemplo, após uma chamada `fork`.

No primeiro caso, a falta é ilegal, sendo um sinal (tipicamente SIGBUS) enviado ao processo. O processo é então terminado, caso não tenha provido gerenciador para o sinal.

No segundo caso, a falta é legal, mas processo é suspenso até o núcleo desmembrar a página referenciada. O núcleo providencia a alocação de uma nova página, marca-a como READ/WRITE, copia o conteúdo da página em falta para a nova página, e decrementa seu contador de referência. Se o contador vai a zero, a página marcada como COPY-ON-WRITE pode ser reusada. A tabela de página do processo aponta para a nova página, agora com permissão de escrita. A partir deste ponto, o processo retoma sua execução.

Capítulo 5

Entrada/Saída

Uma das principais funções do sistema operacional é controlar todos os dispositivos de entrada/saída (E/S) do computador, emitindo comandos para os dispositivos, atendendo interrupções e manipulando erros. O sistema operacional também prover uma interface entre os dispositivos e o resto do sistema, que seja simples e fácil de usar (se possível, a interface deve ser a mesma para todos os dispositivos) [1]. O código de entrada/saída representa uma fração significativa do total do sistema operacional. A forma como o sistema operacional gerencia E/S é o objeto deste capítulo.

5.1 Princípios do Hardware

Diferentes profissionais vêem o hardware de E/S de diferentes maneiras. Engenheiros eletrônicos vêem em termos de *chips*, fios, fontes de potência, motores e todos os outros componentes físicos que constituem em conjunto o hardware. Programadores vêem como a interface apresentada ao software, os comandos que o hardware aceita, as funções suportadas, e os erros que são reportados. O nosso interesse aqui é restringir em como o hardware é programado, e não como ele trabalha internamente.

Dispositivos de E/S

Dispositivos de E/S podem ser divididos em duas grandes categorias: *dispositivos de bloco* e *dispositivos de caracteres*. Um dispositivo de bloco armazena informações em blocos de tamanho fixo, cada um com seu próprio endereço. Tamanhos comuns de blocos estão na faixa de 128 bytes a 1024 bytes. A propriedade essencial dos dispositivos de bloco é a possibilidade de ler ou escrever cada bloco independentemente de todos os demais. Em outras palavras, em qualquer instante, o programa pode ler ou escrever qualquer um dos blocos. Discos são dispositivos de bloco.

O outro tipo de dispositivo de E/S, o de caracteres, libera ou aceita uma fila de caracteres sem definir nenhuma estrutura de bloco. O dispositivo não é endereçável e não aceita operações de busca. Terminais, impressoras e leitoras óticas são exemplos de dispositivos de caracteres.

Este esquema de classificação apresenta exceções. Relógios, por exemplo, geram interrupções em intervalos regulares, ou seja, não são endereçáveis por bloco nem aceitam filas de caracteres. Contudo, este modelo é geral o suficiente para ser usado como base na construção de um sistema operacional com bom nível de independência dos dispositivos de E/S. O sistema de arquivo, por exemplo, negocia apenas com dispositivos de blocos, e deixa a parte dependente

do dispositivo para o software de mais baixo nível, chamado *acionadores de dispositivos (device drivers)*.

Controladores de Dispositivos

Unidades de E/S consistem tipicamente de componentes mecânicos e eletrônicos. É frequente a separação das duas porções para se obter um projeto mais geral e modular. O componente eletrônico é chamado de *controlador do dispositivo (device controller ou adapter)*. Em mini e microcomputadores, ele normalmente toma forma de um circuito impresso que pode ser inserido no computador. O componente mecânico é o dispositivo propriamente dito.

A distinção entre dispositivo e controlador deve ser ressaltada, já que o sistema operacional vê o controlador, não o dispositivo. Normalmente, mini e microcomputadores usam um barramento único (Fig. 5.1) para comunicação entre CPU e os controladores. *Mainframes* frequentemente usam um modelo diferente, no qual múltiplos barramentos e processadores especializados de E/S aliviam parte da carga da CPU.

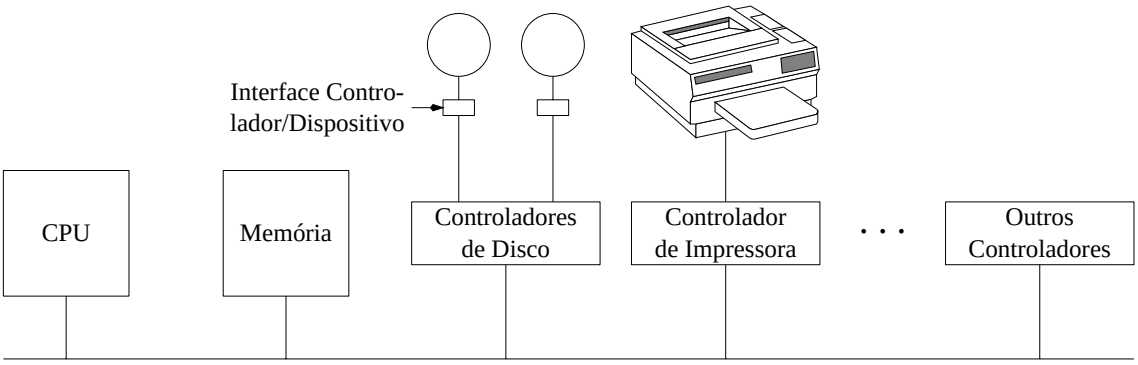

Fig. 5.1: Um modelo para conexão da CPU, memória, controladores e dispositivos de E/S

A interface entre o controlador e o dispositivo é, via de regra, uma interface de baixo nível. O disco, por exemplo, pode ser formatado com 8 setores de 512 bytes por trilha. O que realmente sai do *driver*, entretanto, é uma lista serial de bits, partindo com um preâmbulo, depois os 4096 bits no setor, e finalmente o *checksum* ou o código de correção de erro. O preâmbulo é escrito quando o disco é formatado, e contém o número de cilindros e de setores, o tamanho do setor, e outros dados.

A tarefa do controlador é converter a lista serial de bits em um bloco de bytes e realizar alguma correção de erro necessária. O bloco de bytes é tipicamente montado, bit por bit, em um *buffer* mantido no controlador. Após o *checksum* ter sido verificado e o bloco declarado livre de erro, o mesmo pode então ser copiado para a memória principal.

O controlador para o terminal CRT *(catode ray tube)* também trabalha como um dispositivo serial de bits e em baixo nível. Ele lê da memória o byte contendo o símbolo a ser exibido, e gera os sinais usados na modulação do feixe do CRT para causar a escrita na tela. O controlador também gera os sinais para o *retrace* horizontal após ter terminado de esquadrinhar a linha, como também, sinais para fazer o *retrace* vertical após a tela toda ter sido esquadrinhada. Se não tivéssemos um controlador CRT, o sistema operacional teria que gerar estes sinais diretamente no tubo. Com o controlador, o sistema operacional inicia-o com poucos parâmetros, tais como o número de caracteres por linha e o número de linhas por tela, deixando o controlador tomar conta do direcionador do feixe de raios catódicos.

Cada controlador tem alguns poucos registradores que são usados para comunicação com a CPU. Em alguns computadores estes registradores são parte do espaço de endereçamento regular. A Tab. 5.1 mostra os endereços de E/S e os vetores de interrupção alocados para alguns dos controladores do IBM PC. A atribuição de endereços de E/S para dispositivos é feita por um decodificador lógico associado ao controlador. Alguns IBM PC-compatíveis usam diferentes endereços de E/S.

dispositivo	endereço de E/S	vetor interrupção
relógio	040 - 043	8
teclado	060 - 063	9
porta serial secundária	2F8 - 2FF	11
disco rígido	320 - 32F	13
impressora	378 - 37F	15
vídeo monocromático	380 - 3BF	-
vídeo colorido	3D0 - 3DF	-
disco flexível	3F0 - 3F7	14
porta serial primária	3F8 - 3FF	12

Tab. 5.1: Exemplos de controladores no IBM PC com seus endereços e vetores de interrupção.

O sistema operacional realiza E/S escrevendo comandos nos registradores dos controladores. O controlador de disquete do IBM PC, por exemplo, aceita 15 diferentes comandos, tais como `read`, `write`, `seek`, `format`, e `recalibrate`. Muitos dos comandos têm parâmetros, os quais são também carregados nos registradores do controlador. Quando um comando é aceito, a CPU pode abandonar o controlador e atender a outra tarefa. Quando completado, o controlador causa uma interrupção com o objetivo de permitir que o sistema operacional tome o controle da CPU e teste o resultado da operação. A CPU obtém o resultado e o *status* do dispositivo pela leitura de um ou mais bytes de informação nos registradores do controlador.

Acesso Direto à Memória (DMA)

Muitos controladores, especialmente os que operam em blocos, suportam DMA. Vejamos primeiro como discos operam sem DMA. Primeiro, o controlador lê serialmente o bloco (um ou mais setores) do dispositivo, bit a bit, até que este seja transferido para o buffer interno do controlador. Depois, o controlador executa a operação de *checksum* para atestar que o bloco está livre de erros. Esta operação é executada somente após o bloco todo ter sido transferido. Então, o controlador causa uma interrupção.

Quando o sistema operacional reassume a CPU, ele pode ler o bloco do buffer do controlador (byte a byte ou palavra a palavra) em uma operação cíclica, onde em cada ciclo um byte ou palavra é transferido do controlador para a memória.

Obviamente, o ciclo de transferência de bytes dos controladores para a memória consome um tempo apreciável da CPU. DMA foi criado para livrar a CPU desta tarefa. Quando utilizado, a CPU fornece duas informações ao controlador (além do endereço do bloco a ser lido): o endereço de memória para onde o bloco deve ser copiado e o número de bytes a serem transferidos A Fig. 5.2 ilustra todo este processo.

Após o controlador ter lido o bloco, efetuado o *checksum* e não ter constatado erros de leitura, o próprio controlador copia (via barramento) o primeiro byte (ou palavra) para o

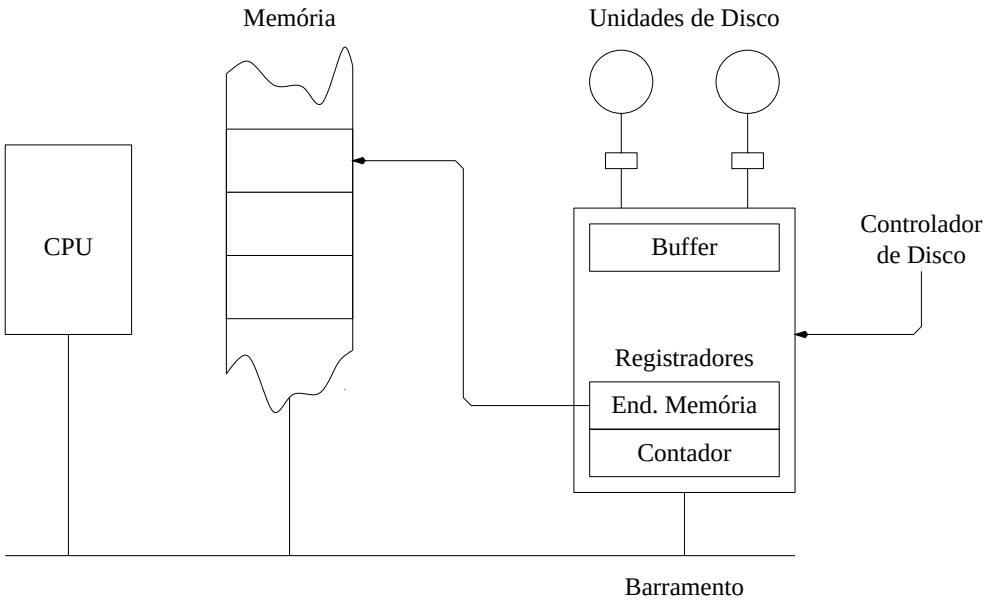

Fig. 5.2: A transferência via DMA é processada sem intervenção da CPU

endereço de memória suprido pela CPU quando da requisição da operação. A partir daí, a cópia prossegue incrementado-se o endereço da memória e decrementado-se o contador do DMA (bytes transferidos), até que este se torne zero. Neste instante, o controlador causa uma interrupção. Quando o sistema operacional reassume a CPU, nenhuma transferência necessita ser efetuada pois o buffer requisitado já se encontra em memória.

5.2 Princípios do Software

Os objetivos gerais do software de E/S são fáceis de serem estabelecidos. A idéia básica é organizar o software como uma série de camadas, com as mais baixas escondendo peculiaridades do hardware e as mais altas apresentando-se de forma simples para o usuário.

Objetivos do Software de E/S

O conceito chave no projeto do software de E/S é a independência do dispositivo. Programas que utilizam arquivos devem operar independentemente do local onde o arquivo encontra-se armazenado. Relacionado com a independência do dispositivo está a uniformidade de nome. O nome de um dispositivo ou arquivo deve ser simplesmente uma cadeia de caracteres *(string)* ou um inteiro não dependente do dispositivo em nenhum caso.

Outra característica importante é a manipulação de erros. Em geral os erros devem ser manipulados o mais próximo possível do hardware. Se o controlador encontra um erro, o mesmo deve tentar corrigí-lo, se possível. Se não, o *driver* do dispositivo deve fazê-lo, talvez apenas tentando ler novamente. Muitos erros são transientes e desaparecem se a operação for repetida. Somente se as camadas mais baixas não conseguirem resolver o problema é que este deve ser apresentado às camadas superiores.

Outra característica chave são as transferências síncronas (bloqueantes) e assíncronas (manipuladas por interrupção). Muitos dispositivos de E/S são assíncronos: a CPU inicia a transferência e se ocupa de outras atividades até que chegue uma interrupção. O sistema operacional

realiza as operações de forma assíncrona, mas para o usuário ela se apresenta como transferência síncrona pois o processo permanecerá bloqueado até a operação ser finalizada (o que torna muito mais simples a programação).

O conceito final que deve ser observado é dispositivos compartilhados e dedicados. Alguns dispositivos de E/S, como discos, podem ser utilizados por muitos usuários ao mesmo tempo. Outros dispositivos, como impressoras, devem ser dedicados a um único usuário até que este finalize a operação. A inclusão de dispositivos dedicados introduz uma variedade de problemas, como o *deadlock*[1]. Sistemas operacionais devem manipular ambos os dispositivos de maneira a evitar estes problemas.

Estes objetivos podem ser organizados de maneira clara e eficiente pela estruturação do software em quatro camadas:

1. Manipulação de interrupções.
2. *Drivers* de dispositivos.
3. Software do sistema operacional independente do dispositivo.
4. Software do nível do usuário.

Manipuladores de Interrupções

Interrupções são eventos complicados de se tratar. Elas devem ser isoladas de modo que apenas uma pequena parte do sistema operacional as manipule. Um meio para isolá-las é bloquear os processos aguardando operações de E/S até que uma interrupção anuncie que a operação se completou.

Quando a interrupção acontece, a rotina de tratamento daquela interrupção libera o processo bloqueado. Em alguns sistemas isto é conseguido fazendo-se um UP sobre um semáforo. Em outros, ele fará um SIGNAL sobre a variável de condição no monitor. E ainda em outros, uma mensagem é enviada ao processo bloqueado. Em todos os casos, o efeito da interrupção é que o processo que estava previamente bloqueado deverá agora estar habilitado para execução.

Drivers de Dispositivos

Todo o código dependente do dispositivo se concentra no *driver* do dispositivo. Cada *driver* manipula um dispositivo ou uma classe de dispositivos intimamente relacionados.

Observamos que cada controlador de dispositivos tem registradores para receber comandos. O *driver* do dispositivo envia estes comandos e testa se foram carregados propriamente. Desta maneira, o *driver* é a parte do sistema operacional que conhece estes registradores e para que são utilizados. O *driver* reconhece setores, trilhas, cilindros, cabeças de leitura/escrita, motor, fator de entrelaçamento e todos os mecanismos que fazem um disco operar propriamente.

Em termos gerais, a função de um *driver* é aceitar requisições abstratas de um software de mais alto nível, e providenciar para que o pedido seja atendido. Uma típica requisição é ler um bloco. Se o *driver* está desocupado no momento, a requisição é aceita, sendo processada imediatamente. Caso o *driver* esteja processando uma requisição, esta normalmente entra em uma fila de requisições pendentes.

[1] Um *deadlock* ocorre quando existe um conjunto de processos bloqueados onde cada processo aguarda um evento para continuar sua execução, evento este que para ocorrer necessita da ação de outro processo pertencente a este mesmo conjunto.

O primeiro passo é transcrever os termos abstratos da requisição para ações concretas. Para um *disk driver*, por exemplo, isto significa informar onde o bloco se encontra no disco, verificar se o motor do *drive* está girando, determinar se o braço está posicionado no cilindro apropriado, e assim por diante. Em poucas palavras, o *driver* deve decidir quais operações do controlador são requeridas e em que sequência.

Uma vez determinado quais comandos emitir ao controlador, o *driver* inicia a emissão escrevendo nos registradores do controlador do dispositivo. Alguns controladores podem manusear somente um comando por vez. Outros controladores aceitam uma lista de comandos, os quais são processados sem a ajuda do sistema operacional.

Após o comando ou comandos terem sido emitidos, podem ocorrer duas situações. Em muitos casos o *device driver* deve esperar até que o controlador execute as operações requisitadas. Se estas operações forem lentas (envolvendo movimentos mecânicos, por exemplo), o *driver* bloqueia até que as operações se completem. Em outros casos, entretanto, as operações são rápidas, situação esta em que o *driver* não precisa ser bloqueado. Como um exemplo dessa situação, o deslocamento da tela em terminais (incluindo o IBM PC) requer apenas escrita de uns poucos bytes nos registradores do controlador. Nenhum movimento mecânico é necessário e a operação toda pode se completar em alguns poucos microsegundos.

Neste ponto, após a operação ter sido completada, o *driver* deve verificar a ocorrência de erros. Se tudo estiver correto, ele passa os dados (o bloco lido, por exemplo) para a próxima camada do software de E/S. Finalmente, ele retorna alguma informação de *status* de erros. Se alguma requisição está na fila, uma delas pode agora ser selecionada e iniciada. Caso contrário, o *driver* fica aguardando a próxima requisição.

Software de E/S Independente do Dispositivo

Embora alguns dos softwares de E/S sejam específicos do dispositivo, uma grande fração deles é independente do dispositivo. O limite exato entre os *drivers* e o software independente dos dispositivos é função do sistema, uma vez que algumas funções que são independentes do dispositivo, podem se concentrar nos *drivers*, por eficiência e por outras razões. As funções listadas abaixo são tipicamente implementadas no software independente do dispositivo:

- interface uniforme para com os *drivers* de dispositivos;
- identificação simbólica dos dispositivos;
- proteção dos dispositivos;
- manipulação de blocos independente dos dispositivos;
- "buferização";
- alocação de espaço nos dispositivos do tipo bloco;
- alocação e liberação de dispositivos dedicados;
- gerenciamento de erros.

A função básica do software independente do dispositivo é realizar as funções de E/S que são comuns a todos os dispositivos, além de prover uma interface uniforme para o software do usuário.

A questão principal em um sistema operacional é como objetos tais como os arquivos e dispositivos de E/S são identificados. O software independente do dispositivo se encarrega do mapeamento simbólico dos nomes dos dispositivos para os seus *drivers* apropriados. No UNIX, por exemplo, cada *device driver* tem a ele associado um *inode*, o qual armazena informações necessárias à localização do *driver* e do dispositivo associados ao *inode*. Estes dispositivos se localizam no diretório `/dev` e o usuário tem permissão limitada para operá-los.

Relacionado ao nome está a proteção. Como o sistema previne usuários de acessar dispositivos que não estão autorizados a acessar? Em muitos microcomputadores, não há nenhuma proteção. Em outros sistemas, acessos a dispositivos de E/S pelos usuários é completamente proibido.

Diferentes discos podem ter diferentes tamanhos de setor. O software independente do dispositivo deve encobrir este fato e prover um tamanho de bloco uniforme para camadas superiores, por exemplo, pelo tratamento de vários setores como um único bloco lógico. Deste modo, os níveis superiores somente negociam com dispositivos abstratos que usam o mesmo tamanho de bloco lógico, independente do tamanho físico do setor.

"Buferização" é uma outra questão, tanto para dispositivos de blocos como para de caracteres. Para dispositivos de bloco, o hardware executa escrita e leitura de blocos inteiros, mas o processo do usuário está livre para ler ou escrever unidades arbitrárias. Para dispositivos de caracteres, pode ser interessante coletar caracteres em um *buffer* e submete-los em conjunto para o dispositivo (requerendo, portanto, "buferização").

Quando um arquivo é criado e preenchido com dados, novos blocos de disco têm que ser alocados para o arquivo. Para realizar esta alocação, o sistema operacional precisa de uma lista ou mapa de bits dos blocos livres no disco, mas o algoritmo para localizar um bloco livre é independente do dispositivo e pode ser implementado acima do *driver*.

Alguns dispositivos, tais como as fitas magnéticas, devem ser usadas somente por um único processo em um dado momento. É o sistema operacional que examina a requisição para usar o dispositivo e aceita ou não, dependendo da disponibilidade do dispositivo requisitado.

A manipulação de erros também é feita nesta camada. Um erro típico é causado por um bloco do disco ruim e que não pode mais ser lido. Após o *driver* tentar ler o bloco várias vezes, ele informa ao software independente do dispositivo o erro. Se ocorreu em um arquivo do usuário, é suficiente informar o erro para o mesmo. Entretanto, se o erro ocorreu em uma área crítica, o sistema operacional deve apresentar uma mensagem e, eventualmente, solicitar intervenção do administrador.

Software no Nível do Usuário

Embora muito do software de E/S esteja embutido no sistema operacional, uma pequena porção deste consiste de bibliotecas ligadas juntamente com programas do usuário, e até mesmo com programas inteiros executando fora do núcleo. Chamadas de sistema, incluindo chamadas do subsistema de E/S, são normalmente feitas por procedimentos da biblioteca. Quando um programa em C contém a chamada

```
bytes_lidos = fread(buffer, tam_item, n_itens, arquivo);
```

o procedimento da biblioteca `fread` será ligado com o programa. A coleção de todos estes procedimentos da biblioteca é parte do sistema de E/S.

Enquanto estes procedimentos fazem pouco mais que colocar seus parâmetros em lugares apropriados para a chamada do sistema, há outros procedimentos de E/S que fazem o trabalho

real. Em particular, a formatação de uma entrada e saída é feita por um procedimento da biblioteca.

Nem todo o software de E/S utilizado pelo usuário consiste de procedimentos da biblioteca. Outra importante categoria é o sistema *spooling*. *Spooling* é o modo de negociação com os dispositivos dedicados de E/S em um sistema com multiprogramação. Considere um dispositivo típico: impressora. Embora seja fácil permitir que algum processo do usuário abra o arquivo especial associado à impressora, suponha que o arquivo permaneça com o processo aberto por várias horas. Nenhum outro processo poderá imprimir. Para estes casos, cria-se um processo especial, chamado *daemon*, e um diretório especial, chamado *spooling directory*. Para imprimir um arquivo, o aplicativo recebe o arquivo inteiro para ser impresso e o armazena no *spooling directory*. Então o *daemon*, o único processo que tem permissão de usar o arquivo especial associado à impressora, transfere um arquivo do *spooling directory* para a impressora por vez. Protegendo-se os arquivos especiais contra o uso direto por usuários, o problema de monopolização é eliminado.

Spooling não é somente usado para impressoras, sendo igualmente usado em outras situações. Por exemplo, transferência de mensagens de correio eletrônico se processa através de um *network daemon*. Para enviar uma mensagem, o aplicativo armazena-a em um diretório de *spooling*. Periodicamente, o *network daemon* acessa o diretório e transmite todas as mensagens armazenadas.

A Fig. 5.3 resume o sistema de E/S, mostrando todas os níveis bem como as funções principais de cada nível.

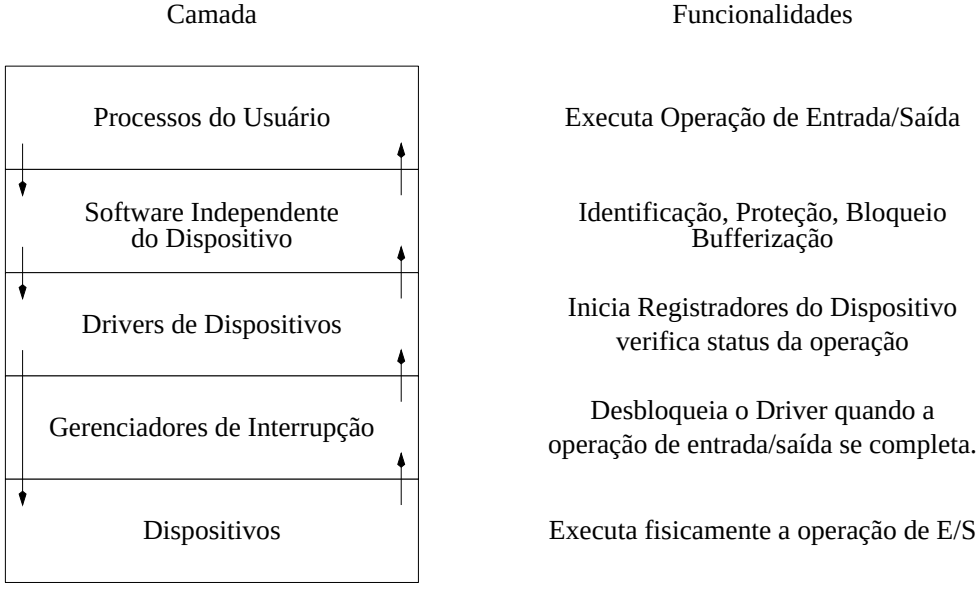

Fig. 5.3: Níveis do sistema de E/S e funções principais de cada nível

5.3 Discos Rotativos

Nas seções seguintes, descreveremos brevemente o hardware do disco, passando para os *disk drivers* a seguir.

5.3.1 Hardware do Disco

Todos os discos rotativos são organizados em cilindros, cada qual contendo tantas trilhas quanto cabeças empilhadas verticalmente. As trilhas são divididas em setores, com um número de setores na circunferência, tipicamente entre 32 a 128. Todos os setores contém o mesmo número de bytes, embora setores no anel externo do disco sejam fisicamente maiores que aqueles no anel interno. O espaço extra não é aproveitado.

Um aspecto que tem importante implicações no *disk driver* é a possibilidade do controlador fazer buscas *(seek)* em dois ou mais dispositivos ao mesmo tempo. Estas são conhecidas como busca entrelaçada *(overlapped seek)*. Enquanto o controlador e o software estão esperando uma busca se completar em um dispositivo, o controlador pode iniciar uma busca em outro. Muitos controladores podem também ler ou escrever em um dispositivo enquanto executam uma busca em um ou mais de um dispositivos, mas nenhum pode ler ou escrever em dois dispositivos no mesmo tempo. (Ler ou escrever requer que o controlador mova bits na faixa de microsegundos, assim uma transferência usa muito de sua capacidade computacional). A habilidade de realizar duas ou mais buscas ao mesmo tempo pode reduzir consideravelmente o tempo médio de acesso.

5.3.2 Software do Disco

Nesta seção veremos algumas características genéricas relacionadas com os *disk drivers*. O tempo de leitura ou escrita de um bloco do disco é determinado por três fatores: o tempo de *seek* (tempo para mover o braço para o cilindro desejado), o atraso rotacional (tempo para o setor desejado ficar sob a cabeça de leitura/escrita), e o tempo de transferência. Para muitos discos, o tempo de *seek* domina. Assim, reduzindo-se o tempo de *seek*, podemos melhorar substancialmente o desempenho do dispositivo

Algoritmos de Escalonamento do Braço do Disco

Se o *disk driver* aceita uma requisição por vez e a executa nesta ordem, isto é, *First-Come, First-Served (FCFS)*, pouco pode ser feito para otimizar o tempo de *seek*. Entretanto, outra estratégia é possível: é provável que enquanto o braço está executando um *seek* na metade de uma requisição, uma outra requisição de disco pode ter sido gerada por outro processo. Muitos *disk drivers* mantém uma tabela, indexada pelo número do cilindro, com todas as requisições pendentes para cada cilindro, encadeadas juntas em uma lista.

Para este tipo de estrutura de dados, podemos melhorar o algoritmo de escalonamento *First-Come, First-Served*. Considere um disco com 40 cilindros. Uma requisição chega para ler um bloco no cilindro 11. Enquanto a busca para o cilindro 11 está em progresso, novas requisições chegam para os cilindros 1, 36, 16, 34, 9, e 12, nesta ordem. Elas são inseridas na tabela de requisições pendentes, tendo cada cilindro um lista separada. As requisições são apresentadas na Fig. 5.4.

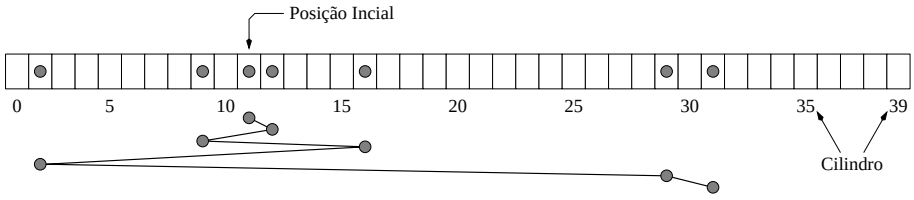

Fig. 5.4: Algoritmo de escalonamento *menor seek primeiro* (SSF)

Quando a requisição corrente termina (cilindro 11), o *disk driver* tem que escolher qual será a próxima requisição. Usando FCFS, ele irá para o cilindro 1, então para o 36, e assim por diante. Este algoritmo requer movimentos do braço percorrendo 10, 35, 20, 18, 25, e 3 cilindros, totalizando 111 cilindros.

Alternativamente, a próxima requisição pode ser manuseada a fim de minimizar o tempo de *seek*. Dadas as requisições da figura 5.4, a sequência 12, 9, 16, 1, 34, e 36, como mostrado na linha segmentada da Fig. 5.4. Com esta sequência, os movimentos do braço percorrem 1, 3, 7, 15, 33, e 2 cilindros, totalizando 61 cilindros. Este algoritmo, *menor seek primeiro* (SSF), diminuiu o total de movimentos do braço pela metade, comparado com o FCFS.

Infelizmente, SSF apresenta um problema. Suponha mais requisições chegando enquanto as requisições da Fig. 5.4 está sendo processada. Por exemplo, se, após chegar ao cilindro 16, uma nova requisição para o cilindro 8 está presente. Esta requisição terá prioridade sobre o cilindro 1. Se a requisição for para o cilindro 13, o braço irá para o 13, ao invés de ir para o cilindro 1. Com discos muito carregados, o braço tende a permanecer no meio do disco a maior parte do tempo, prejudicando assim as requisições das extremidades. Requisições distantes do meio são em média mais demoradas, colocando o objetivo de mínima resposta no tempo e equitatividade em conflito.

Um algoritmo para reconciliar os objetivos conflitantes entre a eficiência e equitatividade constitui-se em manter o movimento do braço na mesma direção até não haver mais requisições pendentes naquela direção, e então o movimento do braço é mudado. Este algoritmo, conhecido como *algoritmo do elevador*, requer o software mantenha 1 bit: o bit da direção corrente, *UP* ou *DOWN*. Quando a requisição termina, o *disk driver* testa o bit. Se for *UP*, o braço é movido para a próxima requisição pendente de posições mais altas, se houver. Se não houver requisições pendentes para posições mais altas, o bit de direção é revertido. Quando o bit é mudado para *DOWN*, o movimento será para a próxima requisição de posição mais baixa, se houver.

A Fig. 5.5 ilustra o algoritmo do elevador usando as mesmas sete requisições da Fig. 5.4, assumindo que o bit de direção esteja inicialmente em *UP*. A ordem na qual os cilindros são servidos é 12, 16, 34, 36, 9, e 1, gerando movimento do braço de 1, 4, 18, 2, 27, e 8, totalizando 60 cilindros. Neste caso, o algoritmo do elevador é sensivelmente melhor que SSF, embora seja usualmente pior. Uma propriedade interessante do algoritmo do elevador é que dada uma coleção de requisições, o limite superior para o total de movimentos é fixado: ele é apenas duas vezes o número de cilindros.

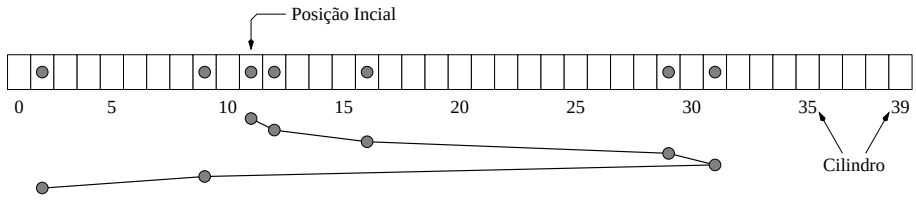

Fig. 5.5: Escalonamento de requisições no disco através do algoritmo do elevador

Alguns controladores de disco provêem um modo do software para inspecionar o número de setores correntes sob a cabeça. Com estes controladores, uma outra otimização é possível. Se duas ou mais requisições para o mesmo cilindro estão pendentes, o *driver* pode emitir a requisição para o setor que passará sob a cabeça do próximo cilindro. Note que quando trilhas múltiplas estão presentes em um cilindro, requisições consecutivas podem ser conduzidas para diferentes trilhas sem qualquer penalidade. O controlador pode selecionar alguma das cabeças instantâneamente, uma vez que seleção de cabeça não involve movimento dos braços nem atraso

rotacional.

Manipulação de Erros

Discos rotativos estão submetidos a uma larga variedade de erros. Alguns dos mais comuns são:

- erros de programação (i.e. requisição para setor não existente);
- erro de *checksum* transiente (i.e. causado por sujeira na cabeça);
- erro de *checksum* permanente (i.e. bloco do disco fisicamente danificado);
- erro de *seek* (i.e. enviar o braço para o cilindro 6, mas ele vai para o 7);
- erro de controlador (i.e. recusa do controlador em aceitar comandos).

É função do *disk driver* manipular cada um desses erros da melhor maneira possível. Erros de programação ocorrem quando o *driver* diz ao controlador para executar uma operação de *seek* em um cilindro não existente, ler de um setor não existente, usar uma cabeça não existente, ou transferir de ou para uma posição de memória inexistente.

Erros de *checksum* transientes são causados por poeira no ar entre a cabeça e a superfície do disco. Em muitos casos eles podem ser eliminados pela repetição da operação algumas vezes. Se o erro persiste, o bloco deve ser marcado como defeituoso.

Um modo de evitar blocos defeituosos é escrever um programa especial que toma a relação destes blocos como entrada, e cria um arquivo contendo todos os blocos defeituosos. Uma vez que este arquivo tenha sido criado, para o alocador do disco parecerá que estes blocos estão ocupados, não os alocando para outros arquivos. Como o arquivo de blocos defeituosos nunca é lido, os blocos defeituosos permanecerão inertes no disco.

Evitar a utilização de blocos defeituosos constitui em uma tarefa árdua. Alguns controladores "inteligentes" reservam algumas poucas trilhas, não disponíveis para programas do usuário. Quando um disco é formatado, o controlador determina quais blocos são defeituosos e automaticamente substitui por uma trilha de reserva. A tabela de mapas de trilhas defeituosos para trilhas de reserva é mantida na memória interna do controlador e no disco. Esta substituição é transparente para o *driver*.

Erros de *seek* são causados por problemas mecânicos no braço. O controlador mantém o rastreamento da posição do braço internamente. Para realizar um *seek*, ele emite uma série de pulsos para o motor do braço, um pulso por cilindro, para mover o braço para o novo cilindro. Quando o braço chega no destino, o controlador lê o número do cilindro (escrito quando o *drive* foi formatado). Se o braço está no lugar errado, um erro de *seek* ocorreu.

Alguns computadores corrigem o erro de *seek* automaticamente, enquanto outros simplesmente atribuem um bit de erro e deixam o resto para o *driver*. O *driver* manipula este erro pela emissão de um comando `recalibrate`, que ajusta os movimentos do braço aos cilindros do disco. Caso esta operação não solucione o problema, o dispositivo deve ser reparado.

Como temos visto, o controlador é um pequeno computador especializado, contendo software, variáveis, *buffers*, e ocasionalmente *bugs*. Algumas vezes uma sequência não usual de eventos, tal como uma interrupção sobre um dispositivo ocorrendo simultaneamente com um comando `recalibrate` para outro, expõe o *bug* e causa o controlador entrar em *loop* ou perder-se do estava fazendo. Projetistas de controladores usualmente consideram a pior situação e

provêem um pino no *chip*, o qual, quando em nível alto, força o controlador esquecer sua tarefa corrente, reiniciando-o. Se tudo isso falhar, o *disk driver* pode reiniciar o controlador. Se isto também for em vão, o *driver* imprime uma mensagem e termina sua operação.

Cache de Rastreamento

O tempo requerido para uma operação de *seek* para um novo cilindro é usualmente muito maior que o tempo de transferência ou rotação. Em outras palavras, uma vez que o *driver* tenha posicionado o braço em algum lugar, pouco importa o tempo gasto para ler um setor ou uma trilha inteira.

Alguns *disk drivers* tiram proveito desta propriedade mantendo internamente um cache do rastreamento *(track-at-a-time cache)*, o que não é conhecido pelo software independente do dispositivo. Se um setor é demandado e o mesmo encontra-se no cache, nenhuma transferência do disco é requerida. A desvantagem da cache de rastreamento (em adição à complexidade do software e espaço de buffer necessário), é que a transferência da cache para o programa que requisitou a operação deve ser feita pela CPU, usando um *loop* programado, ao invés de DMA.

Alguns controladores aprimoram este processo, e fazem o *track-at-a-time caching* na sua memória interna, transparente para o *driver*. Assim a transferência entre o controlador e a memória pode usar DMA. Note que ambos controlador e *driver* podem ler ou escrever trilhas inteiras em um único comando, mas que o software independente do dispositivo não pode, uma vez que ele considera o disco como uma sequência linear de blocos, sem considerar como eles são divididos em trilhas e cilindros.

5.4 Entrada/Saída no UNIX

Drivers Convencionais

Dispositivos no UNIX podem ser do tipo bloco ou caracteres. O interfaceamento com os dispositivos se dá via subsistema de arquivos. Cada dispositivo tem um nome idêntico a nomes de arquivos (`/dev/tty`, `/dev/console`, etc) e um respectivo *inode*. *inodes* associados à dispositivos têm como tipo de arquivo "block" ou "character special", o que os distingue dos arquivos regulares [3, 2].

Cada dispositivo tem seu *device driver* associado. Conforme visto anteriormente, *drivers* são programas que manipulam o controlador do dispositivo. No UNIX, um *driver* implementa as chamadas de sistema `open`, `close`, `read`, `write` e `ioctl` para o seu dispositivo, bem como provê tratamento para as interrupções oriundas deste.

Em dispositivos que operam em modo bloco, a transferência de dados entre o espaço de endereçamento de um processo e o dispositivo de dá via *buffer cache* (ver Cap. 3). A razão para tal é que tais dispositivos (discos, por exemplo) são lentos, sendo portanto a "buferização" um meio de aumentar a taxa de transferência de dados. Dispositivos do tipo caracteres (terminais, por exemplo) são inerentemente rápidos e não necessitam deste recurso.

A Fig. 5.6 apresenta o esquema básico de entrada/saída no UNIX. Quando um processo executa uma chamada de sistema `open`, por exemplo, o núcleo acessa o *inode* do arquivo passado à chamada e descobre que é um arquivo associado a um dispositivo de E/S. Através de uma *tabela de chaveamento de dispositivo*, o núcleo obtém o endereço da rotina `open` para este dispositivo. Uma vez iniciado o dispositivo, um campo na tabela de arquivos (ver Cap. 3) é adicionado, sendo o índice deste campo (descritor) retornado ao processo.

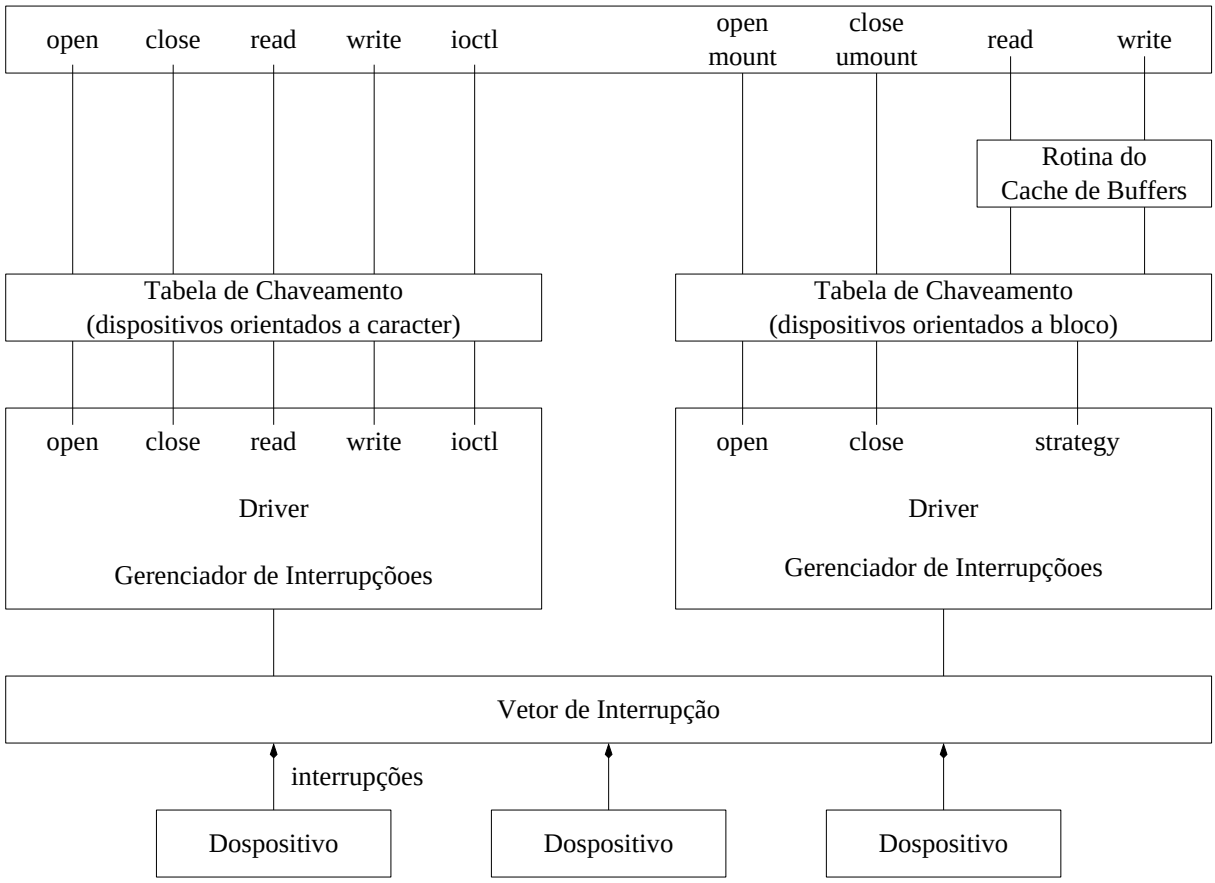

Fig. 5.6: Esquema básico de E/S no UNIX

Uma chamada `close` faz com que o núcleo acesse o respectivo procedimento para o dispositivo, cuja identificação é obtida na tabela de arquivos.

Chamadas `ioctl` permitem o usuário operar tanto arquivos regulares quanto dispositivos do tipo caracteres. Operações típicas são:

- bloquear um arquivo;
- desligar o eco do terminal;
- ajustar taxa de transferência de modens;
- reenrolar uma fita.

Para dispositivos do tipo bloco, chamadas `read` e `write` seguem o mesmo procedimento. Para tais dispositivos, na qual o *driver* tem que interagir com as rotinas de *buffer cache*, o procedimento é outro. Uma rotina denominada `strategy` perfaz as operações de leitura e escrita em tais dispositivos.

Quando uma operação de leitura ou escrita é requisitada, o núcleo identifica a rotina `strategy` definida para o dispositivo, passando à mesma o endereço do cabeçalho do buffer para onde os dados devem ser copiados, caso leitura, ou contendo os dados para escrita.

Streams

Stream é um conceito que provê maior modularidade na implementação de *drivers* para dispositivos do tipo caracteres (principalmente drivers de rede que são estruturados em múltiplas camadas).

Um *stream* é um conjunto de pares de listas ligadas. Uma lista armazena requisições de escrita e na outra estão as leituras que o dispositivo já efetuou (Fig. 5.7). Cada par de listas implementa um determinado nível de abstração, desde os mais altos (como as chamadas de sistema) até os mais baixos (como controle direto do dispositivo), passando por níveis intermediários (como protocolos). A Fig. 5.7 ilustra um *driver* de terminal empregando o conceito de *stream*.

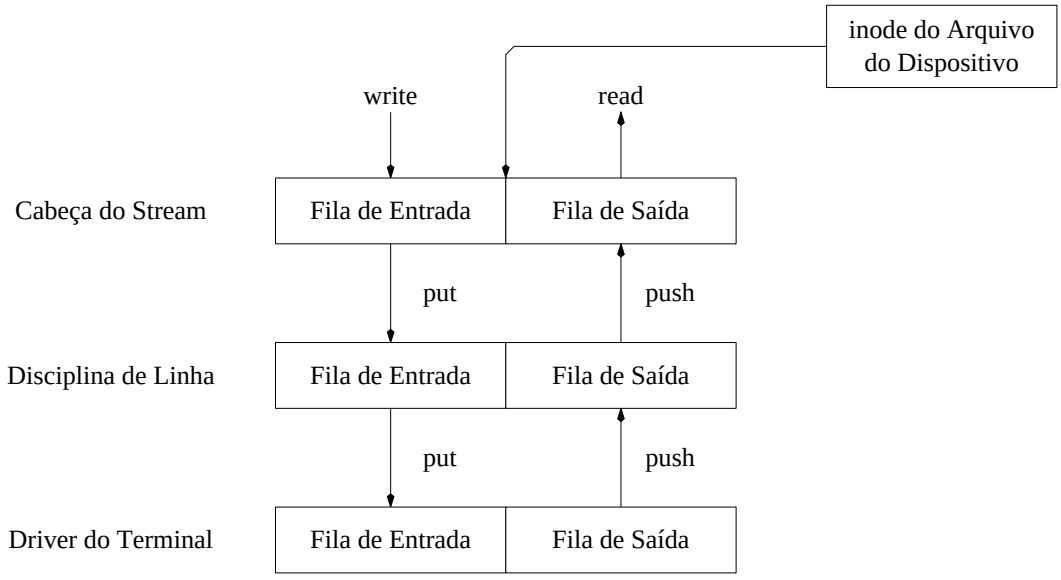

Fig. 5.7: *Driver* de terminal composto de um *stream* com três pares de listas

Quando um processo escreve dados em um *stream*, o núcleo envia os dados para a próxima lista, até chegar ao dispositivo. Quando o *driver* recebe dados do dispositivo, ocorre o processo inverso: os dados são enviados até chegar à lista de nível mais alto, permanecendo a disposição do processo que requisitou a operação.

Cada lista do *stream* consiste em uma estrutura de dados contendo:

- um procedimento de abertura invocado durante uma chamada `open`;
- um procedimento de fechamento invocado durante uma chamada `close`;
- um procedimento de `put` para adicionar dados à lista;
- um procedimento de padrão (serviço) chamado quando a lista é processada;
- um ponteiro para a próxima lista no *stream*;
- um ponteiro para a fila de itens aguardando passagem para a para a próxima lista no *stream*;
- campos utilizados para o controle de fluxo, escalonamento de serviços. etc.

O núcleo descobre se o *driver* é comum ou baseado em *streams* quando da consulta à tabela de chaveamento de dispositivo.

O fluxo de dados entre listas se dá por estruturas denominadas *mensagens*. Uma mensagem consiste de uma lista de blocos contendo dados ou controle. Mensagens de controle são originadas durante o processamento de uma chamada `ioctl`. Uma chamada `open` causa a criação do *stream*, com um conjunto de blocos para o fluxo de mensagens.

Quando um processo escreve em um *stream*, o núcleo copia os dados do espaço de endereçamento do processo para os blocos alocados ao *stream*. O *stream* transfere estes blocos para a próxima lista invocando sua rotina de *put*. Caso tal operação falhe (por não haver blocos disponíveis nesta lista, por exemplo), estes blocos são marcados para processamento futuro. A leitura de um *stream* se dá com a chamada `ioctl` com a opção PUSH. Streams são destruídos por ocasião da chamada `close` para seu dispositivo.

Streams permitem a implementação de *drivers* onde as chamadas `open`, `close`, `read` e `write` possuem código dependente do dispositivo físico apenas nos níveis mais baixos (próximos ao dispositivo).

Bibliografia

[1] Andrew Tanenbaum. *"Modern Operating Systems"*. Massachusetts Addison-Wesley, 1984. ISBN 0-13-595752-4.

[2] Uresh Vahalia. *"UNIX Internals: The New Frontiers"*. Prentice Hall, Upper Saddle River, New Jersey 07458, 1996. ISBN 0-13-101908-2.

[3] Maurice J. Bach. *"The Design of The UNIX Operating System"*. Prentice Hall Software Series, Englewood Cliffs, New Jersey 07632, 1990. ISBN 0-13-201799-7.