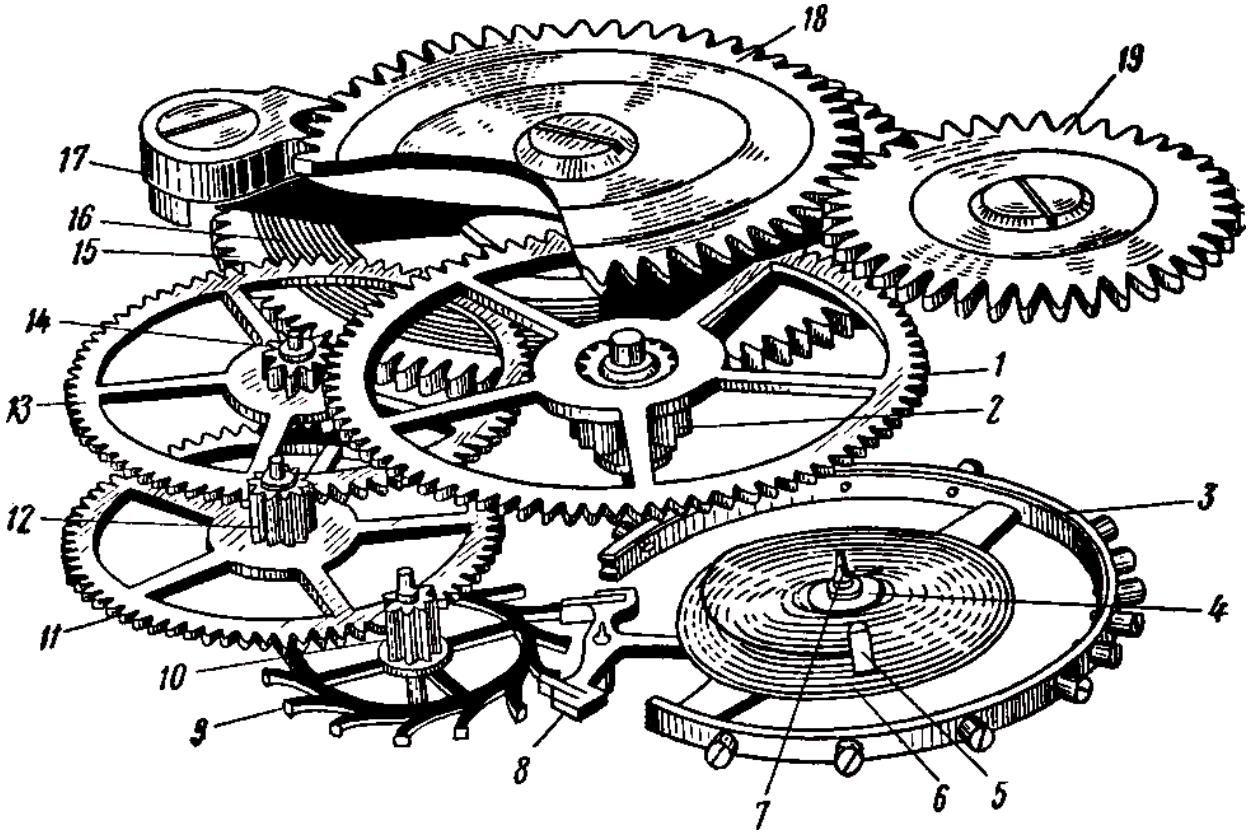
Sistemas Operacionais: Conceitos e Mecanismos

Prof. Carlos A. Maziero

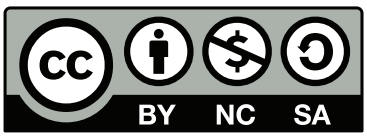
DINF - UFPR



Sistemas Operacionais: Conceitos e Mecanismos

ISSN: a definir

Sobre o autor: Carlos A. Maziero é professor do Depto de Informática da Universidade Federal do Paraná (UFPR) desde 2015. Anteriormente, foi professor da Universidade Tecnológica Federal do Paraná (UTFPR), entre 2011 e 2015, professor da Pontifícia Universidade Católica do Paraná (PUCPR), entre 1998 e 2011, e professor da Universidade Federal de Santa Catarina (UFSC), de 1996 a 1998. Formado em Engenharia Elétrica (UFSC, 1988), tem Mestrado em Engenharia Elétrica (UFSC, 1990), Doutorado em Informática (Université de Rennes I - França, 1994) e Pós-Doutorados na Università degli Studi di Milano – Italia (2009) e no IRISA/INRIA Rennes – França (2018). Atua em ensino e pesquisa nas áreas de sistemas operacionais, sistemas distribuídos e segurança de sistemas.



Este texto foi produzido usando exclusivamente software livre: Sistema Operacional *GNU/Linux* (distribuições *Fedora* e *Ubuntu*), compilador de texto LaTeX 2$_\varepsilon$, gerenciador de referências *BibTeX*, editor gráfico *Inkscape*, criadores de gráficos *GNUPlot* e *GraphViz* e processador PS/PDF *GhostScript*, entre outros. Algumas figuras do texto usam ícones de `https://www.creativetail.com` e de outras fontes, sob licença *Creative Commons*.

Versão compilada em 5 de abril de 2019.

Prefácio

Os sistemas operacionais são elementos fundamentais para o funcionamento de praticamente qualquer sistema de computação, dos minúsculos sistemas embarcados e telefones celulares aos gigantescos centros de processamento de dados das grandes empresas. Apesar da imensa diversidade de sistemas operacionais existentes, eles tentam resolvem problemas de mesma natureza e seguem basicamente os mesmos princípios.

Conhecer sistemas operacionais a fundo não é algo reservado a *hackers*, mas importante para todo profissional de computação, pois os mecanismos implementados pelo sistema operacional afetam diretamente o comportamento e o desempenho das aplicações. Além disso, o sistema operacional é uma peça chave na configuração de serviços de rede e na segurança do sistema.

Existem muitos livros de sistemas operacionais disponíveis no mercado, quase todos muito bons, escritos por profissionais reconhecidos mundialmente. Entretanto, bons livros de Sistemas Operacionais podem custar centenas de reais, o que os torna inacessíveis a uma parcela significativa da população. Este livro seria apenas mais uma opção de compra nas livrarias, não fosse por um pequeno detalhe: foi concebido como um Livro Aberto, desde seu início. Um Livro Aberto (do inglês *Open Book*) é um livro amplamente disponível na Internet em formato digital, sem custo. No exterior, muitos *open books* estão também disponíveis nas livrarias, para aquisição em formato impresso.

Eu acredito que "inclusão digital" não significa somente permitir o acesso a computadores à parcela mais pobre da população, mas também desmistificar o funcionamento dessa tecnologia e incentivar seu estudo, para fomentar as próximas gerações de técnicos, engenheiros e cientistas, vindas de todas as classes sociais. Nosso país não pode mais se dar ao luxo de desperdiçar pessoas inteligentes somente porque são pobres.

Prof. Carlos Maziero, Dr.

Agradecimentos

Este texto é fruto de alguns anos de trabalho. Embora eu o tenha redigido sozinho, ele nunca teria se tornado uma realidade sem a ajuda e o apoio de muitas pessoas, de várias formas. Em primeiro lugar, agradeço à minha família, pelas incontáveis horas em que me subtraí de seu convívio para me dedicar a este trabalho.

Agradeço também a todos os docentes e estudantes que utilizaram este material, pelas inúmeras correções e sugestões de melhoria. Em particular, meus agradecimentos a Alexandre Koutton, Altair Santin, Antônio Barros, Antônio Gonçalves, Carlos Roland, Carlos Silla, Diogo Olsen, Douglas da Costa, Fabiano Beraldo, Fred Maranhão, Jeferson Amend, Marcos Laureano, Paulo Resende, Rafael Hamasaki, Richard Reichardt, Tadeu Ribeiro Reis, Thayse Solis, Thiago Ferreira, Thiago Vieira, Urlan de Barros e Vagner Sacramento.

Desejo expressar meu mais profundo respeito pelos autores dos grandes clássicos de Sistemas Operacionais, como Andrew Tanenbaum e Abraham Silberschatz, que iluminaram meus passos nesta área e que seguem como referências inequívocas e incontornáveis.

Agradeço à Pontifícia Universidade Católica do Paraná, onde fui professor de Sistemas Operacionais por 13 anos, pelas condições de trabalho que me permitiram dedicar-me a esta empreitada. Também à Universidade Tecnológica Federal do Paraná, onde trabalhei de 2011 a 2015, e à UFPR, onde trabalho desde 2015, pelas mesmas razões.

Dedico os capítulos sobre segurança computacional deste livro aos colegas docentes e pesquisadores do Departamento de Tecnologias da Informação da Universidade de Milão em Crema, onde estive em um pós-doutorado no ano de 2009, com uma bolsa CAPES/MEC. Os capítulos sobre virtualização são dedicados à equipe ADEPT IRISA/INRIA, Université de Rennes 1 - França, na qual pude passar três meses agradáveis e produtivos durante o inverno 2007-08, como professor/pesquisador convidado.

Carlos Maziero

Curitiba PR, Abril de 2019

Sumário

Parte I

Introdução

Capítulo 1

Conceitos básicos

Um sistema de computação é constituído basicamente por hardware e software. O hardware é composto por circuitos eletrônicos (processador, memória, portas de entrada/saída, etc.) e periféricos eletro-óptico-mecânicos (teclados, mouses, discos rígidos, unidades de disquete, CD ou DVD, dispositivos USB, etc.). Por sua vez, o software de aplicação é representado por programas destinados ao usuário do sistema, que constituem a razão final de seu uso, como editores de texto, navegadores Internet ou jogos. Entre os aplicativos e o hardware reside uma camada de software multifacetada e complexa, denominada genericamente de *Sistema Operacional* (SO). Neste capítulo veremos quais os objetivos básicos do sistema operacional, quais desafios ele deve resolver e como ele é estruturado para alcançar seus objetivos.

1.1 Objetivos de um SO

Existe uma grande distância entre os circuitos eletrônicos e dispositivos de hardware e os programas aplicativos em software. Os circuitos são complexos, acessados através de interfaces de baixo nível (geralmente usando as portas de entrada/saída do processador) e muitas vezes suas características e seu comportamento dependem da tecnologia usada em sua construção. Por exemplo, a forma de acesso de baixo nível a discos rígidos IDE difere da forma de acesso a discos SCSI ou leitores de CD. Essa grande diversidade pode ser uma fonte de dor de cabeça para o desenvolvedor de aplicativos. Portanto, é desejável oferecer aos programas aplicativos uma forma de acesso homogênea aos dispositivos físicos, que permita abstrair sua complexidade e as diferenças tecnológicas entre eles.

O sistema operacional é uma camada de software que opera entre o hardware e os programas aplicativos voltados ao usuário final. Trata-se de uma estrutura de software ampla, muitas vezes complexa, que incorpora aspectos de baixo nível (como *drivers* de dispositivos e gerência de memória física) e de alto nível (como programas utilitários e a própria interface gráfica).

A Figura 1.1 ilustra a estrutura geral de um sistema de computação típico. Nela, podemos observar elementos de hardware, o sistema operacional e alguns programas aplicativos.

Os objetivos básicos de um sistema operacional podem ser sintetizados em duas palavras-chave: “abstração” e “gerência”, cujos principais aspectos são detalhados a seguir.

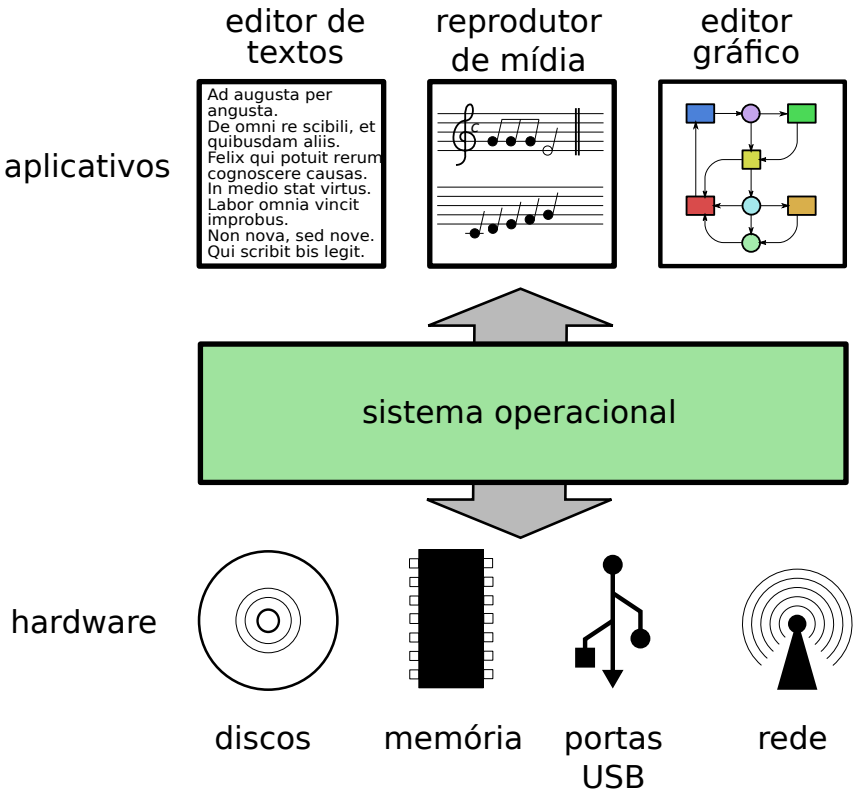


Figura 1.1: Estrutura de um sistema de computação típico

1.1.1 Abstração de recursos

Acessar os recursos de hardware de um sistema de computação pode ser uma tarefa complexa, devido às características específicas de cada dispositivo físico e a complexidade de suas interfaces. Por exemplo, a sequência a seguir apresenta os principais passos envolvidos na abertura de um arquivo (operação `open`) em um disco:

1. verificar se os parâmetros informados estão corretos (nome do arquivo, identificador do disco, buffer de leitura, etc.);
2. verificar se o disco está disponível;
3. ligar o motor do disco e aguardar atingir a velocidade de rotação correta;
4. posicionar a cabeça de leitura sobre a trilha onde está a tabela de diretório;
5. ler a tabela de diretório e localizar o arquivo ou subdiretório desejado;
6. mover a cabeça de leitura para a posição do bloco inicial do arquivo;
7. ler o bloco inicial do arquivo e depositá-lo em um buffer de memória.

Assim, o sistema operacional deve definir interfaces abstratas para os recursos do hardware, visando atender os seguintes objetivos:

- *Prover interfaces de acesso aos dispositivos, mais simples de usar que as interfaces de baixo nível*, para simplificar a construção de programas aplicativos. Por exemplo: para ler dados de um disco rígido, um programador de aplicação usa o conceito

de *arquivo*, que implementa uma visão abstrata do disco rígido, acessível através de operações como `open`, `read` e `close`. Caso tivesse de acessar o disco diretamente, seria necessário manipular portas de entrada/saída e registradores com comandos para o controlador de disco (sem falar na dificuldade de localizar os dados desejados dentro do disco).

- *Tornar os aplicativos independentes do hardware*. Ao definir uma interface abstrata de acesso a um dispositivo de hardware, o sistema operacional desacopla o hardware dos aplicativos e permite que ambos evoluam de forma mais autônoma. Por exemplo, o código de um editor de textos não deve ser dependente da tecnologia de discos utilizada no sistema.

- *Definir interfaces de acesso homogêneas para dispositivos com tecnologias distintas*. Através de suas abstrações, o sistema operacional permite aos aplicativos usar a mesma interface para dispositivos diversos. Por exemplo, um aplicativo acessa dados em disco através de arquivos e diretórios, sem precisar se preocupar com a estrutura real de armazenamento dos dados, que podem estar em um disquete, um disco SATA, uma máquina fotográfica digital conectada à porta USB, um CD ou mesmo um disco remoto, compartilhado através da rede.

1.1.2 Gerência de recursos

Os programas aplicativos usam o hardware para atingir seus objetivos: ler e armazenar dados, editar e imprimir documentos, navegar na Internet, tocar música, etc. Em um sistema com várias atividades simultâneas, podem surgir conflitos no uso do hardware, quando dois ou mais aplicativos precisam dos mesmos recursos para poder executar. Cabe ao sistema operacional definir *políticas* para gerenciar o uso dos recursos de hardware pelos aplicativos, e resolver eventuais disputas e conflitos. Vejamos algumas situações onde a gerência de recursos do hardware se faz necessária:

- Cada computador normalmente possui menos processadores que o número de tarefas em execução. Por isso, o uso desses processadores deve ser distribuído entre os aplicativos presentes no sistema, de forma que cada um deles possa executar na velocidade adequada para cumprir suas funções sem prejudicar os demais. O mesmo ocorre com a memória RAM, que deve ser distribuída de forma justa entre as aplicações.

- A impressora é um recurso cujo acesso deve ser efetuado de forma mutuamente exclusiva (apenas um aplicativo por vez), para não ocorrer mistura de conteúdo nos documentos impressos. O sistema operacional resolve essa questão definindo uma fila de trabalhos a imprimir (*print jobs*) normalmente atendidos de forma sequencial (FIFO).

- Ataques de negação de serviço (*DoS – Denial of Service*) são comuns na Internet. Eles consistem em usar diversas técnicas para forçar um servidor de rede a dedicar seus recursos para atender um determinado usuário, em detrimento dos demais. Por exemplo, ao abrir milhares de conexões simultâneas em um servidor de e-mail, um atacante pode reservar para si todos os recursos do servidor (processos, conexões de rede, memória e processador), fazendo com que os demais usuários não sejam mais atendidos. É responsabilidade do

sistema operacional do servidor detectar tais situações e impedir que todos os recursos do sistema sejam monopolizados por um só usuário (ou um pequeno grupo).

Assim, um sistema operacional visa abstrair o acesso e gerenciar os recursos de hardware, provendo aos aplicativos um ambiente de execução abstrato, no qual o acesso aos recursos se faz através de interfaces simples, independentes das características e detalhes de baixo nível, e no qual os conflitos no uso do hardware são minimizados.

1.2 Funcionalidades

Para cumprir seus objetivos de abstração e gerência, o sistema operacional deve atuar em várias frentes. Cada um dos recursos do sistema possui suas particularidades, o que impõe exigências específicas para gerenciar e abstrair os mesmos. Sob esta perspectiva, as principais funcionalidades implementadas por um sistema operacional típico são:

Gerência do processador: esta funcionalidade, também conhecida como gerência de processos, de tarefas ou de atividades, visa distribuir a capacidade de processamento de forma justa[1] entre as aplicações, evitando que uma aplicação monopolize esse recurso e respeitando as prioridades definidas pelos usuários.

O sistema operacional provê a ilusão de que existe um processador independente para cada tarefa, o que facilita o trabalho dos programadores de aplicações e permite a construção de sistemas mais interativos. Também faz parte da gerência de atividades fornecer abstrações para sincronizar atividades interdependentes e prover formas de comunicação entre elas.

Gerência de memória: tem como objetivo fornecer a cada aplicação uma área de memória própria, independente e isolada das demais aplicações e inclusive do sistema operacional. O isolamento das áreas de memória das aplicações melhora a estabilidade e segurança do sistema como um todo, pois impede aplicações com erros (ou aplicações maliciosas) de interferir no funcionamento das demais aplicações. Além disso, caso a memória RAM existente seja insuficiente para as aplicações, o sistema operacional pode aumentá-la de forma transparente às aplicações, usando o espaço disponível em um meio de armazenamento secundário (como um disco rígido).

Uma importante abstração construída pela gerência de memória, com o auxílio do hardware, é a noção de *memória virtual*, que desvincula os endereços de memória vistos por cada aplicação dos endereços acessados pelo processador na memória RAM. Com isso, uma aplicação pode ser carregada em qualquer posição livre da memória, sem que seu programador tenha de se preocupar com os endereços de memória onde ela irá executar.

Gerência de dispositivos: cada periférico do computador possui suas particularidades; assim, o procedimento de interação com uma placa de rede é completamente

[1] Distribuir de forma justa, mas não necessariamente igual, pois as aplicações têm demandas distintas de processamento. Por exemplo, um navegador de Internet demanda menos o processador que um aplicativo de edição de vídeo, e por isso o navegador pode receber menos tempo de processador.

diferente da interação com um disco rígido SATA. Todavia, existem muitos problemas e abordagens em comum para o acesso aos periféricos. Por exemplo, é possível criar uma abstração única para a maioria dos dispositivos de armazenamento como *pendrives*, discos SATA ou IDE, CDROMs, etc., na forma de um vetor de blocos de dados. A função da gerência de dispositivos (também conhecida como *gerência de entrada/saída*) é implementar a interação com cada dispositivo por meio de *drivers* e criar modelos abstratos que permitam agrupar vários dispositivos similares sob a mesma interface de acesso.

Gerência de arquivos: esta funcionalidade é construída sobre a gerência de dispositivos e visa criar arquivos e diretórios, definindo sua interface de acesso e as regras para seu uso. É importante observar que os conceitos abstratos de arquivo e diretório são tão importantes e difundidos que muitos sistemas operacionais os usam para permitir o acesso a recursos que nada tem a ver com armazenamento. Exemplos disso são as conexões de rede (nos sistemas UNIX e Windows, cada socket TCP é visto como um descritor de arquivo no qual pode-se ler ou escrever dados) e as informações internas do sistema operacional (como o diretório `/proc` do UNIX). No sistema experimental *Plan 9* [Pike et al., 1993], por exemplo, todos os recursos do sistema operacional são vistos como arquivos.

Gerência de proteção: com computadores conectados em rede e compartilhados por vários usuários, é importante definir claramente os recursos que cada usuário pode acessar, as formas de acesso permitidas (leitura, escrita, etc.) e garantir que essas definições sejam cumpridas. Para proteger os recursos do sistema contra acessos indevidos, é necessário: a) definir usuários e grupos de usuários; b) identificar os usuários que se conectam ao sistema, através de procedimentos de autenticação; c) definir e aplicar regras de controle de acesso aos recursos, relacionando todos os usuários, recursos e formas de acesso e aplicando essas regras através de procedimentos de autorização; e finalmente d) registrar o uso dos recursos pelos usuários, para fins de auditoria e contabilização.

Além dessas funcionalidades básicas oferecidas pela maioria dos sistemas operacionais, várias outras vêm se agregar aos sistemas modernos, para cobrir aspectos complementares, como a interface gráfica, suporte de rede, fluxos multimídia, fontes de energia, etc. As funcionalidades do sistema operacional geralmente são interdependentes: por exemplo, a gerência do processador depende de aspectos da gerência de memória, assim como a gerência de memória depende da gerência de dispositivos e da gerência de proteção.

Uma regra importante a ser observada na construção de um sistema operacional é a separação entre os conceitos de **política** e **mecanismo**[2] [Levin et al., 1975]. Como *política* consideram-se os aspectos de decisão mais abstratos, que podem ser resolvidos por algoritmos de nível mais alto, como por exemplo decidir a quantidade de memória que cada aplicação ativa deve receber, ou qual o próximo pacote de rede a enviar para satisfazer determinadas especificações de qualidade de serviço.

Por outro lado, como *mecanismo* consideram-se os procedimentos de baixo nível usados para implementar as políticas, ou seja, para atribuir ou retirar memória de uma aplicação, enviar ou receber um pacote de rede, etc. Os mecanismos devem

[2]Na verdade essa regra é tão importante que deveria ser levada em conta na construção de qualquer sistema computacional complexo.

ser suficientemente genéricos para suportar mudanças de política sem necessidade de modificações. Essa separação entre os conceitos de política e mecanismo traz uma grande flexibilidade aos sistemas operacionais, permitindo alterar sua personalidade (sistemas mais interativos ou mais eficientes) sem ter de alterar o código que interage diretamente com o hardware. Alguns sistemas, como o InfoKernel [Arpaci-Dusseau et al., 2003], permitem que as aplicações escolham as políticas do sistema mais adequadas às suas necessidades específicas.

1.3 Categorias

Os sistemas operacionais podem ser classificados segundo diversos parâmetros e aspectos, como tamanho de código, velocidade, suporte a recursos específicos, acesso à rede, etc. A seguir são apresentados alguns tipos de sistemas operacionais usuais (muitos sistemas operacionais se encaixam bem em mais de uma das categorias apresentadas):

Batch (de lote): os sistemas operacionais mais antigos trabalhavam "por lote", ou seja, todos os programas a executar eram colocados em uma fila, com seus dados e demais informações para a execução. O processador recebia os programas e os processava sem interagir com os usuários, o que permitia um alto grau de utilização do sistema. Atualmente, este conceito se aplica a sistemas que processam tarefas sem interação direta com os usuários, como os sistemas de processamento de transações bancárias. Além disso, o termo "em lote" também é usado para designar um conjunto de comandos que deve ser executado em sequência, sem interferência do usuário. Exemplos clássicos desses sistemas incluem o IBM OS/360 e o VAX/VMS, entre outros.

De rede: um sistema operacional de rede deve possuir suporte à operação em rede, ou seja, a capacidade de oferecer às aplicações locais recursos que estejam localizados em outros computadores da rede, como arquivos e impressoras. Ele também deve disponibilizar seus recursos locais aos demais computadores, de forma controlada. A maioria dos sistemas operacionais atuais oferece esse tipo de funcionalidade.

Distribuído: em um sistema operacional distribuído, os recursos de cada computador estão disponíveis a todos na rede, de forma transparente aos usuários. Ao lançar uma aplicação, o usuário interage com sua interface, mas não sabe onde ela está executando ou armazenando seus arquivos: o sistema é quem decide, de forma transparente ao usuário. Sistemas operacionais distribuídos já existem há muito tempo (por exemplo, o Amoeba [Tanenbaum et al., 1991]); recentemente, os ambientes de computação em nuvem têm implementado esse conceito. Em uma aplicação na nuvem, o usuário interage com a interface da aplicação em um computador ou telefone, mas não tem uma visão clara das máquinas onde seus dados estão sendo processados e armazenados.

Multiusuário: um sistema operacional multiusuário deve suportar a identificação do "dono" de cada recurso dentro do sistema (arquivos, processos, áreas de memória, conexões de rede) e impor regras de controle de acesso para impedir o uso desses recursos por usuários não autorizados. Essa funcionalidade é

fundamental para a segurança dos sistemas operacionais de rede e distribuídos. Grande parte dos sistemas atuais são multiusuários.

Servidor: um sistema operacional servidor deve permitir a gestão eficiente de grandes quantidades de recursos (disco, memória, processadores), impondo prioridades e limites sobre o uso dos recursos pelos usuários e seus aplicativos. Normalmente um sistema operacional servidor também tem suporte a rede e multiusuários.

Desktop: um sistema operacional "de mesa" é voltado ao atendimento do usuário doméstico e corporativo para a realização de atividades corriqueiras, como edição de textos e gráficos, navegação na Internet e reprodução de mídia. Suas principais características são a interface gráfica, o suporte à interatividade e a operação em rede. Exemplos de sistemas *desktop* são os vários sistemas Windows (XP, Vista, 7, 10, etc.), MacOS e Linux.

Móvel: um sistema operacional móvel é usado em equipamentos de uso pessoal compactos, como *smartphones* e *tablets*. Nesse contexto, as principais prioridades são a gestão eficiente da energia (bateria), a conectividade nos diversos tipos de rede (*wifi*, GSM, Bluetooth, NFC, etc) e a interação com uma grande variedade de sensores (GPS, giroscópio, luminosidade, tela de toque, leitor de digitais, etc). Android e iOS são bons exemplos desta categoria.

Embarcado: um sistema operacional é dito embarcado (embutido ou *embedded*) quando é construído para operar sobre um hardware com poucos recursos de processamento, armazenamento e energia. Aplicações típicas desse tipo de sistema aparecem em sistemas de automação e controladores automotivos, equipamentos eletrônicos de uso doméstico (leitores de DVD, TVs, fornos de microondas, centrais de alarme, etc.). Muitas vezes um sistema operacional embarcado se apresenta na forma de uma biblioteca a ser ligada ao programa da aplicação durante sua compilação. LynxOS, TinyOS, Contiki e VxWorks são exemplos de sistemas operacionais embarcados.

Tempo real: são sistemas nos quais o tempo é essencial. Ao contrário da ideia usual, um sistema operacional de tempo real não precisa ser necessariamente ultrarrápido; sua característica essencial é ter um comportamento temporal previsível, ou seja, seu tempo de resposta deve ser previsível no melhor e no pior caso de operação. A estrutura interna de um sistema operacional de tempo real deve ser construída de forma a minimizar esperas e latências imprevisíveis, como tempos de acesso a disco e sincronizações excessivas. Exemplos de sistemas operacionais de tempo real incluem o QNX, RT-Linux e VxWorks. Muitos sistemas embarcados têm características de tempo real, e vice-versa.

Existem sistemas de tempo real *críticos* (*hard real-time systems*), nos quais a perda de um prazo pelo sistema pode perturbar seriamente o sistema físico sob seu controle, com graves consequências humanas, econômicas ou ambientais. Exemplos desse tipo de sistema seriam o controle de funcionamento de uma turbina de avião ou de um freio ABS. Por outro lado, nos sistemas de tempo-real *não-críticos* (*soft real-time systems*), a perda de um prazo é perceptível e degrada o serviço prestado, sem maiores consequências. Exemplos desse tipo de sistema são os softwares de reprodução de mídia: em caso de atrasos, podem ocorrer falhas na música que está sendo tocada.

1.4 Um breve histórico dos SOs

Os primeiros sistemas de computação, no final dos anos 1940, não possuíam sistema operacional: as aplicações eram executadas diretamente sobre o hardware. Por outro lado, os sistemas de computação atuais possuem sistemas operacionais grandes, complexos e em constante evolução. A seguir são apresentados alguns dos marcos mais relevantes na história dos sistemas operacionais [Wikipedia, 2018]:

Anos 40: cada programa executava sozinho e tinha total controle do computador. A carga do programa em memória, a varredura dos periféricos de entrada para busca de dados, a computação propriamente dita e o envio dos resultados para os periférico de saída, byte a byte, tudo devia ser programado detalhadamente pelo desenvolvedor da aplicação.

Anos 50: os sistemas de computação fornecem "bibliotecas de sistema" (*system libraries*) que encapsulam o acesso aos periféricos, para facilitar a programação de aplicações. Algumas vezes um programa "monitor" (*system monitor*) auxilia a carga e descarga de aplicações e/ou dados entre a memória e periféricos (geralmente leitoras de cartão perfurado, fitas magnéticas e impressoras de caracteres).

1961: o grupo do pesquisador Fernando Corbató, do MIT, anuncia o desenvolvimento do CTSS – *Compatible Time-Sharing System [Corbató et al., 1962]*, o primeiro sistema operacional com compartilhamento de tempo.

1965: a IBM lança o OS/360, um sistema operacional avançado, com compartilhamento de tempo e excelente suporte a discos.

1965: um projeto conjunto entre MIT, GE e Bell Labs define o sistema operacional *Multics*, cujas ideias inovadoras irão influenciar novos sistemas durante décadas.

1969: Ken Thompson e Dennis Ritchie, pesquisadores dos Bell Labs, criam a primeira versão do UNIX.

1981: a Microsoft lança o MS-DOS, um sistema operacional comprado da empresa *Seattle Computer Products* em 1980.

1984: a Apple lança o sistema operacional Mac OS 1.0 para os computadores da linha Macintosh, o primeiro a ter uma interface gráfica totalmente incorporada ao sistema.

1985: primeira tentativa da Microsoft no campo dos sistemas operacionais com interface gráfica, através do MS-Windows 1.0.

1987: Andrew Tanenbaum, um professor de computação holandês, desenvolve um sistema operacional didático simplificado, mas respeitando a API do UNIX, que foi batizado como *Minix*.

1987: IBM e Microsoft apresentam a primeira versão do OS/2, um sistema multitarefa destinado a substituir o MS-DOS e o Windows. Mais tarde, as duas empresas rompem a parceria; a IBM continua no OS/2 e a Microsoft investe no ambiente Windows.

1991: Linus Torvalds, um estudante de graduação finlandês, inicia o desenvolvimento do Linux, lançando na rede Usenet o núcleo 0.01, logo abraçado por centenas de programadores ao redor do mundo.

1993: a Microsoft lança o Windows NT, o primeiro sistema 32 bits da empresa, que contava com uma arquitetura interna inovadora.

1993: lançamento dos UNIX de código aberto FreeBSD e NetBSD.

1993: a Apple lança o Newton OS, considerado o primeiro sistema operacional móvel, com gestão de energia e suporte para tela de toque.

1995: a AT&T lança o Plan 9, um sistema operacional distribuído.

1999: a empresa VMWare lança um ambiente de virtualização para sistemas operacionais de mercado.

2001: a Apple lança o MacOS X, um sistema operacional com arquitetura distinta de suas versões anteriores, derivada da família UNIX BSD.

2005: lançado o Minix 3, um sistema operacional micro-núcleo para aplicações embarcadas. O Minix 3 faz parte do *firmware* dos processadores Intel mais recentes.

2006: lançamento do Windows Vista.

2007: lançamento do *iPhone* e seu sistema operacional iOS, derivado do sistema operacional *Darwin*.

2007: lançamento do Android, um sistema operacional baseado no núcleo Linux para dispositivos móveis.

2010: Windows Phone, SO para celulares pela Microsoft.

2015: Microsoft lança o Windows 10.

Esse histórico reflete apenas o surgimento de alguns sistemas operacionais relativamente populares; diversos sistemas acadêmicos ou industriais de grande importância pelas contribuições inovadoras, como *Mach* [Rashid et al., 1989], *Chorus* [Rozier and Martins, 1987], *QNX* e outros, não estão representados.

Referências

A. Arpaci-Dusseau, R. Arpaci-Dusseau, N. Burnett, T. Denehy, T. Engle, H. Gunawi, J. Nugent, and F. Popovici. Transforming policies into mechanisms with InfoKernel. In *19th ACM Symposium on Operating Systems Principles*, October 2003.

F. Corbató, M. Daggett, and R. Daley. An experimental time-sharing system. In *Proceedings of the Spring Joint Computer Conference*, 1962.

R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. *SIGOPS Operating Systems Review*, 9(5):132–140, Nov. 1975.

R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in Plan 9. *Operating Systems Review*, 27(2):72–76, April 1993.

R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. B. Jones. Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON*, pages 176–178, San Francisco, CA, USA, 1989. IEEE Comput. Soc. Press.

M. Rozier and J. L. Martins. The Chorus distributed operating system: Some design issues. In Y. Paker, J.-P. Banatre, and M. Bozyiğit, editors, *Distributed Operating Systems*, pages 261–287, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

A. Tanenbaum, M. Kaashoek, R. van Renesse, and H. Bal. The Amoeba distributed operating system – a status report. *Computer Communications*, 14:324–335, July 1991.

Wikipedia. Wikipedia online enciclopedia. `http://www.wikipedia.org`, 2018.

Capítulo 2

Estrutura de um SO

Este capítulo apresenta os principais componentes de uma sistema operacional e os mecanismos de hardware necessários para sua implementação.

2.1 Elementos do sistema operacional

Um sistema operacional não é um bloco único e fechado de software executando sobre o hardware. Na verdade, ele é composto de diversos componentes com objetivos e funcionalidades complementares. Alguns dos componentes mais relevantes de um sistema operacional típico são:

Núcleo: é o coração do sistema operacional, responsável pela gerência dos recursos do hardware usados pelas aplicações. Ele também implementa as principais abstrações utilizadas pelos aplicativos e programas utilitários.

Código de inicialização: (*boot code*) a inicialização do hardware requer uma série de tarefas complexas, como reconhecer os dispositivos instalados, testá-los e configurá-los adequadamente para seu uso posterior. Outra tarefa importante é carregar o núcleo do sistema operacional em memória e iniciar sua execução.

Drivers: módulos de código específicos para acessar os dispositivos físicos. Existe um driver para cada tipo de dispositivo, como discos rígidos SATA, portas USB, placas gráfica, etc. Muitas vezes o *driver* é construído pelo próprio fabricante do hardware e fornecido em forma compilada (em linguagem de máquina) para ser acoplado ao restante do sistema operacional.

Programas utilitários: são programas que facilitam o uso do sistema computacional, fornecendo funcionalidades complementares ao núcleo, como formatação de discos e mídias, configuração de dispositivos, manipulação de arquivos (mover, copiar, apagar), interpretador de comandos, terminal, interface gráfica, gerência de janelas, etc.

As diversas partes do sistema operacional estão relacionadas entre si conforme apresentado na Figura 2.1. Na figura, a região cinza indica o sistema operacional propriamente dito. A forma como os diversos componentes do SO são interligados e se relacionam varia de sistema para sistema; algumas possibilidades de organização são discutidas no Capítulo 3.

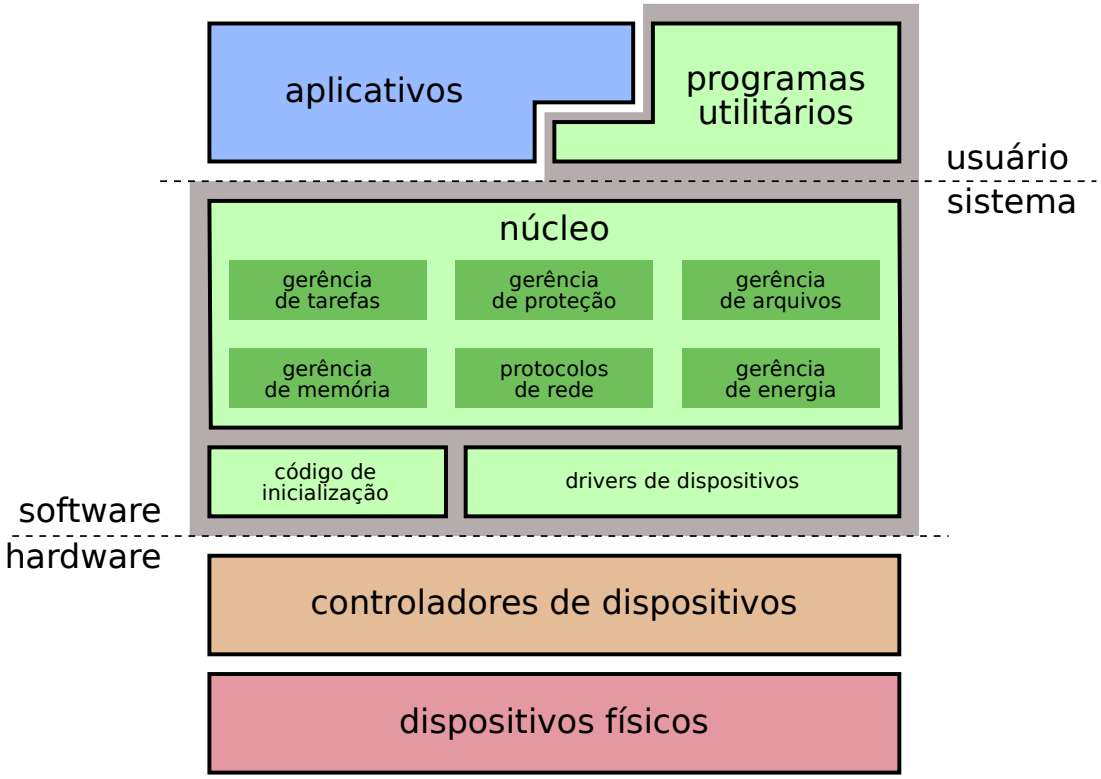


Figura 2.1: Estrutura de um sistema operacional típico

A camada mais baixa do sistema operacional, que constitui o chamado "núcleo" do sistema (ou (*kernel*), usualmente executa em um modo especial de operação do processador, denominado *modo privilegiado* ou modo sistema (vide Seção 2.2.3). Os demais programas e aplicações executam em um modo denominado *modo usuário*.

Sistemas operacionais reais têm estruturas que seguem aquela apresentada na Figura 2.1, embora sejam geralmente muito mais complexas. Por exemplo, o sistema operacional Android, usado em telefones celulares, é baseado no Linux, mas tem uma rica estrutura de bibliotecas e serviços no nível de usuário, que são usados para a construção de aplicações. A Figura 2.2 representa a arquitetura de um sistema Android 8.0.

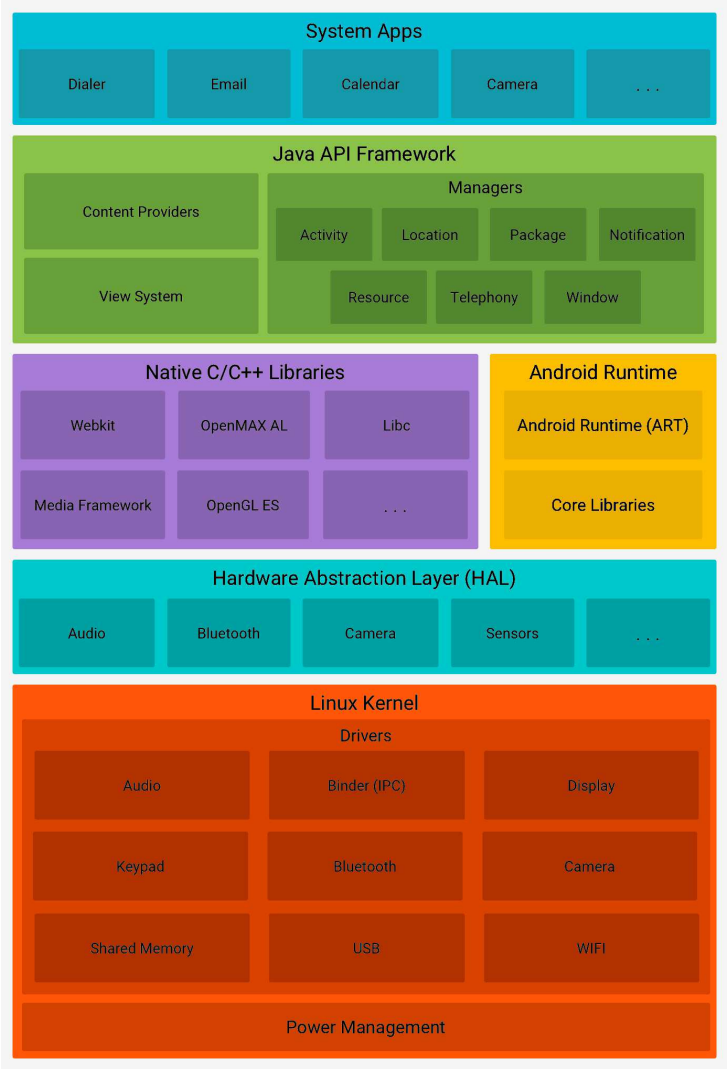


Figura 2.2: Estrutura de um sistema operacional Android [Google, 2018].

2.2 Elementos de hardware

O sistema operacional executa diretamente sobre o hardware, abstraindo e gerenciando recursos para as aplicações. Para que o SO possa cumprir suas funções com eficiência e confiabilidade, o hardware deve prover algumas funcionalidades básicas que são discutidas nesta seção.

2.2.1 Arquitetura do computador

Um computador típico é constituído de um ou mais processadores, responsáveis pela execução das instruções das aplicações, uma ou mais áreas de memórias que armazenam as aplicações em execução (seus códigos e dados) e dispositivos periféricos que permitem o armazenamento de dados e a comunicação com o mundo exterior, como discos, terminais e teclados.

A maioria dos computadores monoprocessados atuais segue uma arquitetura básica definida nos anos 40 por János (John) Von Neumann, conhecida por "arquitetura Von Neumann". A principal característica desse modelo é a ideia de "programa

armazenado", ou seja, o programa a ser executado reside na memória junto com os dados. Os principais elementos constituintes do computador estão interligados por um ou mais barramentos (para a transferência de dados, endereços e sinais de controle). A Figura 2.3 ilustra a arquitetura de um computador típico.

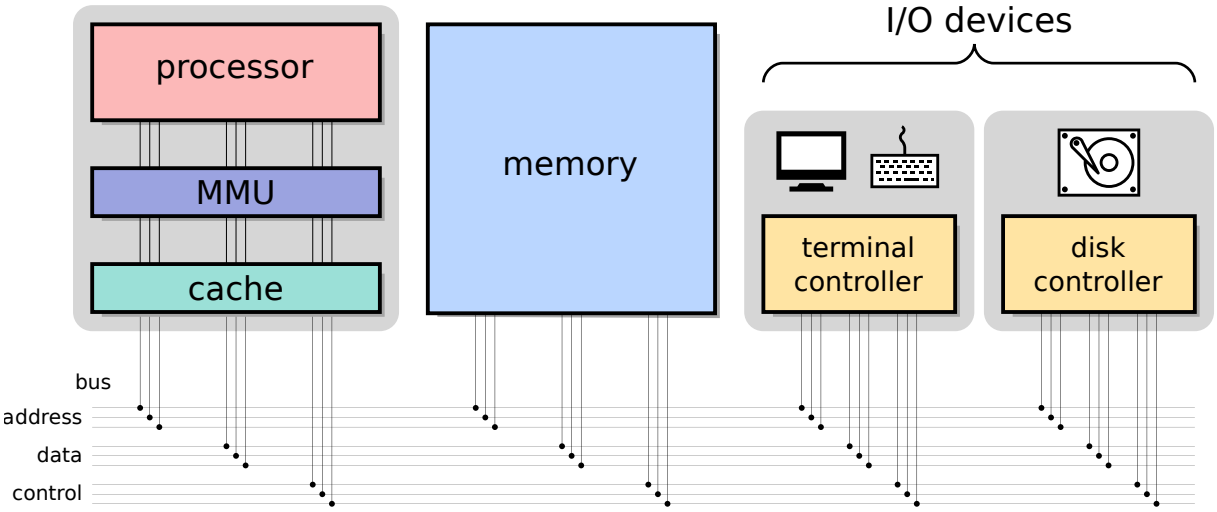


Figura 2.3: Arquitetura de um computador típico

O centro do sistema de computação é o processador. Ele é responsável por continuamente ler instruções e dados da memória ou dos periféricos, processá-los e enviar os resultados de volta à memória ou a outros periféricos. Em sua forma mais simples, um processador convencional é constituído de uma unidade lógica e aritmética (ULA), que realiza os cálculos e operações lógicas, um conjunto de registradores para armazenar dados de trabalho e alguns registradores para funções especiais (contador de programa, ponteiro de pilha, flags de status, etc.).

Processadores modernos são incrivelmente mais complexos, podendo possuir diversos núcleos de processamento (os chamados *cores*), cada um contendo vários processadores lógicos internos (*hyperthreading*). Além disso, um computador pode conter vários processadores trabalhando em paralelo. Outro aspecto diz respeito à memória, que pode comportar vários níveis de cache, dentro do processador e na placa-mãe. Uma descrição mais detalhada da arquitetura de computadores modernos pode ser encontrada em [Stallings, 2010].

Todas as transferências de dados entre processador, memória e periféricos são feitas através dos barramentos: o **barramento de endereços** indica a posição de memória (ou o dispositivo) a acessar, o **barramento de controle** indica a operação a efetuar (leitura ou escrita) e o **barramento de dados** transporta a informação a ser transferida entre o processador e a memória ou um controlador de dispositivo.

O acesso do processador à memória é mediado por um controlador específico (que pode estar fisicamente dentro do próprio chip do processador): a **Unidade de Gerência de Memória** (MMU - *Memory Management Unit*). Ela é responsável por analisar cada endereço de memória acessado pelo processador, validá-lo, efetuar conversões de endereçamento porventura necessárias e executar a operação solicitada pelo processador (leitura ou escrita de uma posição de memória). Uma memória *cache* permite armazenar os dados mais recentemente lidos da memória, para melhorar o desempenho.

Os periféricos do computador (discos, teclado, monitor, etc.) são acessados através de circuitos eletrônicos específicos, denominados **controladores**: a placa de

vídeo permite o acesso ao monitor, a placa *ethernet* dá acesso à rede, o controlador USB permite acesso ao mouse, teclado e outros dispositivos USB externos. Dentro do computador, cada dispositivo é representado por seu respectivo controlador. Os controladores podem ser acessados através de *portas de entrada/saída* endereçáveis: a cada controlador é atribuída uma faixa de endereços de portas de entrada/saída. A Tabela 2.1 a seguir apresenta alguns endereços portas de entrada/saída para acessar controladores em um PC típico (podem variar):

dispositivo	**endereços de acesso**
temporizador	0040-0043
teclado	0060-006F
porta serial COM1	02F8-02FF
controlador SATA	30BC-30BF
controlador Ethernet	3080-309F
controlador	3000-303F

Tabela 2.1: Endereços de acesso a dispositivos (em hexadecimal).

2.2.2 Interrupções e exceções

A comunicação entre o processador e os dispositivos se dá através do acesso às portas de entrada/saída, que podem ser lidas e/ou escritas pelo processsador. Esse acesso é feito por iniciativa do processador, quando este precisa ler ou escrever dados no dispositivo. Entretanto, muitas vezes um dispositivo precisa informar o processador rapidamente sobre um evento interno, como a chegada de um pacote de rede, um clique de mouse ou a conclusão de uma operação de disco. Neste caso, o controlador tem duas alternativas:

- aguardar até que o processador o consulte, o que poderá ser demorado caso o processador esteja ocupado com outras tarefas (o que geralmente ocorre);
- notificar o processador, enviando a ele uma *requisição de interrupção* (IRQ – *Interrupt ReQuest*) através do barramento de controle.

Ao receber a requisição de interrupção, os circuitos do processador suspendem seu fluxo de execução corrente e desviam para um endereço pré-definido, onde se encontra uma *rotina de tratamento de interrupção* (*interrupt handler*). Essa rotina é responsável por tratar a interrupção, ou seja, executar as ações necessárias para atender o dispositivo que a gerou. Ao final da rotina de tratamento da interrupção, o processador retoma o código que estava executando quando recebeu a requisição.

A Figura 2.4 representa os principais passos associados ao tratamento de uma interrupção envolvendo a placa de rede Ethernet, enumerados a seguir:

1. o processador está executando um programa qualquer (em outras palavras, um fluxo de execução);

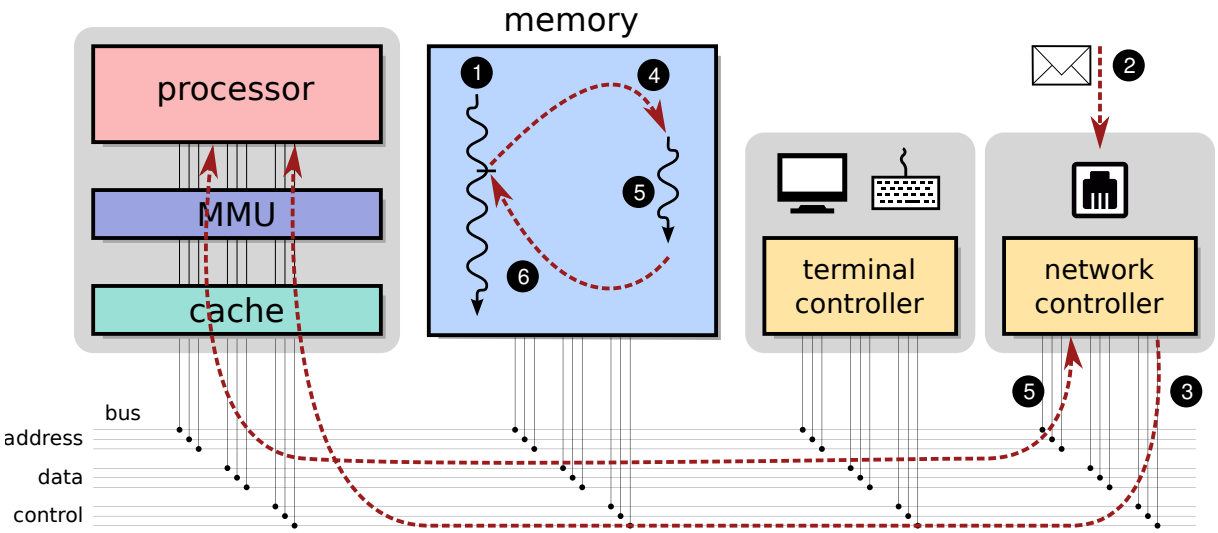


Figura 2.4: Roteiro típico de um tratamento de interrupção

2. um pacote vindo da rede é recebido pela placa Ethernet;
3. o controlador Ethernet envia uma solicitação de interrupção (IRQ) ao processador;
4. o processamento é desviado do programa em execução para a rotina de tratamento da interrupção;
5. a rotina de tratamento é executada para interagir com o controlador de rede (via barramentos de dados e de endereços) para transferir os dados do pacote de rede do controlador para a memória;
6. a rotina de tratamento da interrupção é finalizada e o processador retorna à execução do programa que havia sido interrompido.

Essa sequência de ações ocorre a cada requisição de interrupção recebida pelo processador, que geralmente corresponde a um evento ocorrido em um dispositivo: a chegada de um pacote de rede, um click no mouse, uma operação concluída pelo disco, etc. Interrupções não são eventos raros, pelo contrário: centenas ou mesmo milhares de interrupções são recebidas pelo processador por segundo, dependendo da carga e da configuração do sistema (número e tipo dos periféricos). Assim, as rotinas de tratamento de interrupção devem ser curtas e realizar suas tarefas rapidamente, para não prejudicar o desempenho do sistema.

Para distinguir interrupções geradas por dispositivos distintos, cada interrupção é identificada pelo hardware por um número inteiro. Como cada interrupção pode exigir um tipo de tratamento diferente (pois os dispositivos são diferentes), cada IRQ deve disparar sua própria rotina de tratamento de interrupção. A maioria das arquiteturas atuais define uma tabela de endereços de funções denominada *Tabela de Interrupções* (IVT - *Interrupt Vector Table*); cada entrada dessa tabela aponta para a rotina de tratamento da interrupção correspondente. Por exemplo, se a entrada 5 da tabela contém o valor `3C20h`, então a rotina de tratamento da IRQ 5 iniciará na posição `3C20h` da memória RAM. A tabela de interrupções reside em uma posição fixa da memória RAM, definida

pelo fabricante do processador, ou tem sua posição indicada pelo conteúdo de um registrador da CPU específico para esse fim.

A maioria das interrupções recebidas pelo processador têm como origem eventos externos a ele, ocorridos nos dispositivos periféricos e reportados por seus controladores. Entretanto, alguns eventos gerados pelo próprio processador podem ocasionar o desvio da execução usando o mesmo mecanismo das interrupções: são as *exceções*. Ações como instruções ilegais (inexistentes ou com operandos inválidos), tentativas de divisão por zero ou outros erros de software disparam exceções no processador, que resultam na ativação de uma rotina de tratamento de exceção, usando o mesmo mecanismo das interrupções (e a mesma tabela de endereços de funções). A Tabela 2.2 representa parte da tabela de interrupções do processador Intel Pentium.

Tabela 2.2: Tabela de Interrupções do processador Pentium [Patterson and Henessy, 2005]

IRQ	**Descrição**
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	Intel reserved
16	floating point error
17	alignment check
18	machine check
19-31	Intel reserved
32-255	maskable interrupts (devices & exceptions)

O mecanismo de interrupção torna eficiente a interação do processador com os dispositivos periféricos. Se não existissem interrupções, o processador perderia muito tempo "varrendo" todos os dispositivos do sistema para verificar se há eventos a serem tratados. Além disso, as interrupções permitem construir funções de entrada/saída

assíncronas, ou seja, o processador não precisa esperar a conclusão de cada operação solicitada a um dispositivo, pois o dispositivo gera uma interrupção para "avisar" o processador quando a operação for concluída. O Capítulo 19 traz informações mais detalhadas sobre o tratamento de interrupções pelo sistema operacional.

2.2.3 Níveis de privilégio

Núcleo, *drivers*, utilitários e aplicações são constituídos basicamente de código de máquina. Todavia, devem ser diferenciados em sua capacidade de interagir com o hardware: enquanto o núcleo e os drivers devem ter pleno acesso ao hardware, para poder configurá-lo e gerenciá-lo, os aplicativos e utilitários devem ter acesso mais restrito a ele, para não interferir nas configurações e na gerência, o que poderia desestabilizar o sistema inteiro. Além disso, aplicações com acesso pleno ao hardware seriam um risco à segurança, pois poderiam contornar facilmente os mecanismos de controle de acesso aos recursos (tais como arquivos e áreas de memória).

Para permitir diferenciar os privilégios de execução dos diferentes tipos de software, os processadores modernos implementam *níveis de privilégio de execução*. Esses níveis são controlados por flags especiais na CPU, que podem ser ajustados em função tipo de código em execução. Os processadores no padrão Intel x86, por exemplo, dispõem de 4 níveis de privilégio (0 . . . 3, sendo 0 o nível mais privilegiado), embora a maioria dos sistemas operacionais construídos para esses processadores só use os níveis extremos (0 para o núcleo do sistema operacional e 3 para utilitários e aplicações)[1].

Figura 2.5: Níveis de privilégio de processadores Intel x86

Na forma mais simples desse esquema, podemos considerar dois níveis básicos de privilégio:

Nível núcleo: também denominado nível *supervisor*, *sistema*, *monitor* ou ainda *kernel space*. Para um código executando nesse nível, todas as funcionalidades do processador estão disponíveis: todas as instruções disponíveis podem ser executadas e todos os recursos internos (registradores e portas de entrada/saída) e áreas de memória podem ser acessados. Ao ser ligado durante a inicialização do computador, o processador entra em operação neste nível.

[1] A ideia de definir níveis de privilégio para o código foi introduzida pelo sistema operacional Multics [Corbató and Vyssotsky, 1965], nos anos 1960. No Multics, os níveis eram organizados em camadas ao redor do núcleo, como em uma cebola, chamadas "anéis de proteção" (*protection rings*).

Nível usuário: neste nível, também chamado *userspace*, somente um subconjunto das instruções do processador e registradores estão disponíveis. Por exemplo, instruções consideradas "perigosas", como `RESET` (reiniciar o processador) e `IN`/`OUT` (acessar portas de entrada/saída), são proibidas. Caso o código em execução tente executar uma instrução proibida, o hardware irá gerar uma exceção, desviando a execução para uma rotina de tratamento dentro do núcleo. Essa rotina irá tratar o erro, provavelmente abortando o programa em execução[2].

Dessa forma, o núcleo do sistema operacional, bem como *drivers* e o código de inicialização, executam em modo núcleo, enquanto os programas utilitários e os aplicativos executam em modo usuário. Os flags que definem o nível de privilégio só podem ser modificados por código executando no nível núcleo, o que impede usuários maliciosos de contornar essa barreira de proteção.

Além da proteção oferecida pelos níveis de privilégio, o núcleo do sistema operacional pode configurar a unidade de gerência de memória (MMU) para criar uma área de memória exclusiva para cada aplicação, isolada das demais aplicações e do núcleo. As aplicações não podem contornar essa barreira de memória, pois a configuração da MMU só pode ser alterada em modo supervisor, ou seja, pelo próprio núcleo.

Com a proteção provida pelos níveis de privilégio do processador e pelas áreas de memória exclusivas configuradas pela MMU, obtém-se um modelo de execução confinada, no qual as aplicações executam isoladas umas das outras e do núcleo, conforme ilustrado na Figura 2.6. Esse modelo de execução é bastante robusto, pois isola os erros, falhas e comportamentos indevidos de uma aplicação do restante do sistema, proporcionando mais estabilidade e segurança.

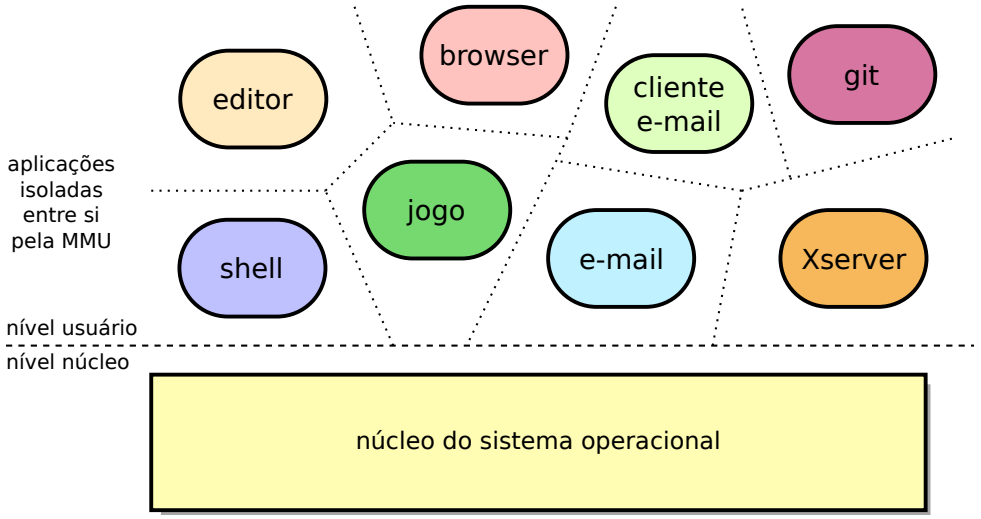


Figura 2.6: Separação entre o núcleo e as aplicações

[2]E também irá gerar a famosa frase "este programa executou uma instrução ilegal e será finalizado", no caso do Windows.

2.3 Chamadas de sistema

O confinamento de cada aplicação em sua área de memória, imposto pela MMU aos acessos em memória em nível usuário, provê robustez e confiabilidade ao sistema, pois garante que uma aplicação não poderá interferir nas áreas de memória de outras aplicações ou do núcleo. Entretanto, essa proteção introduz um novo problema: como invocar, a partir da aplicação, as rotinas oferecidas pelo núcleo para o acesso ao hardware e demais serviços do SO? Deve-se lembrar que o código do núcleo reside em uma área de memória inacessível à aplicação, então operações como `jump` e `call` não funcionariam.

A resposta a esse problema está no mecanismo de interrupção, apresentado na Seção 2.2.2. Os processadores implementam instruções específicas para invocar serviços do núcleo, funcionando de forma similar às interrupções. Ao ser executada, essa instrução comuta o processador para o nível privilegiado e executa o código contido em uma rotina predefinida, como em uma interrupção. Por essa razão, esse mecanismo é denominado *interrupção de software*, ou *trap*.

A ativação de uma rotina do núcleo usando esse mecanismo é denominada **chamada de sistema** (*system call* ou *syscall*). Os sistemas operacionais definem chamadas de sistema para todas as operações envolvendo o acesso a recursos de baixo nível (periféricos, arquivos, alocação de memória, etc.) ou abstrações lógicas (criação e encerramento de tarefas, operadores de sincronização, etc.). Geralmente as chamadas de sistema são oferecidas para as aplicações em modo usuário através de uma *biblioteca do sistema* (*system library*), que prepara os parâmetros, invoca a chamada e deporta devolve à aplicação os resultados obtidos.

Nos processadores modernos a chamada de sistema e seu retorno são feitos usando instruções específicas como `sysenter/sysexit` (x86 32 bits), ou `syscall/sysret` (x86 64 bits). Antes de invocar a chamada de sistema, alguns registradores do processador são preparados com valores específicos, como o número da operação desejada (*opcode*, geralmente no registrador AX), o endereço dos parâmetros da chamada, etc. As regras para o preenchimento dos registradores são específicas para cada chamada de sistema, em cada sistema operacional.

Um exemplo simples mas ilustrativo do uso de chamadas de sistema em ambiente Linux é apresentado a seguir. O programa em linguagem C imprime uma *string* na saída padrão (chamada `write`) e em seguida encerra a execução (chamada `exit`):

```
1 #include <unistd.h>
2
3 int main (int argc, char *argv[])
4 {
5   write (1, "Hello World!\n", 13) ; /* write string to stdout */
6   _exit (0) ;                      /* exit with no error */
7 }
```

Esse código em C, ao ser reescrito em Assembly x86, mostra claramente a preparação e invocação das chamadas de sistema:

```
; to assembly and link (in Linux 64 bit):
; nasm -f elf64 -o hello.o hello.asm; ld -o hello hello.o

section .data
msg db 'Hello World!', 0xA  ; output string (0xA = "\n")
len equ 13                  ; string size

section .text

global _start

_start:
   mov rax, 1               ; syscall opcode (1: write)
   mov rdi, 1               ; file descriptor (1: stdout)
   mov rsi, msg             ; data to write
   mov rdx, len             ; number of bytes to write
   syscall                  ; make syscall

   mov rax, 60              ; syscall opcode (60: _exit)
   mov rdi, 0               ; exit status (0: no error)
   syscall                  ; make syscall
```

A Figura 2.7 detalha o funcionamento básico da chamada de sistema `write`, que escreve dados em um arquivo previamente aberto). A seguinte sequência de passos é realizada:

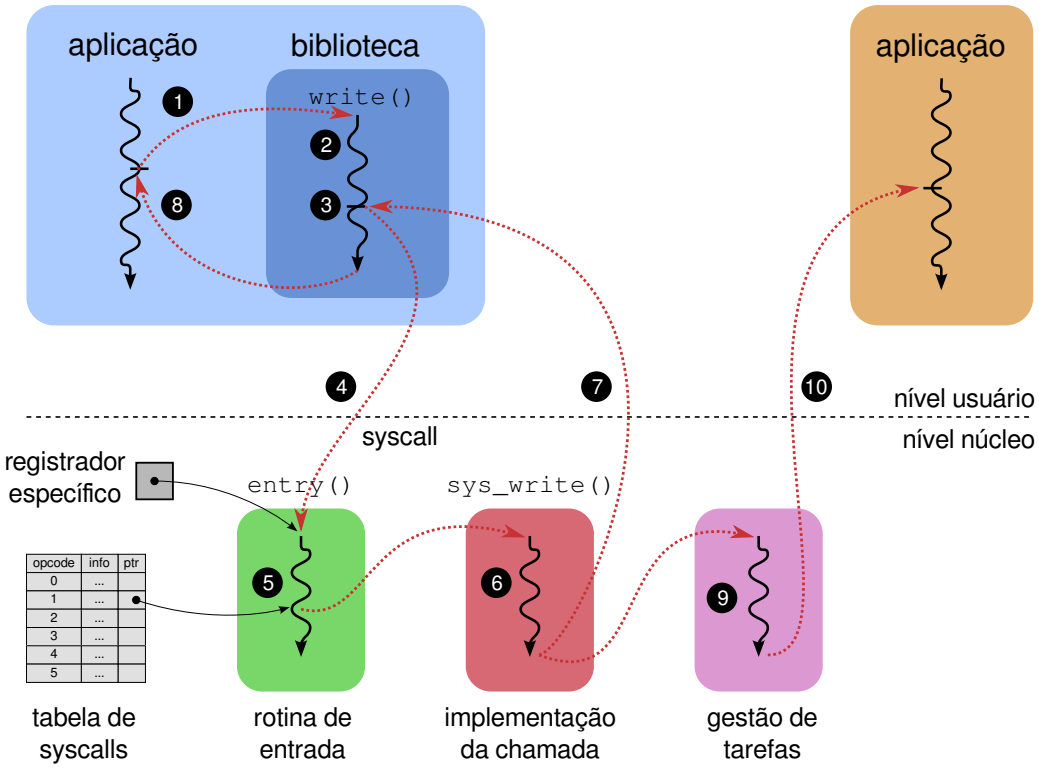


Figura 2.7: Roteiro típico de uma chamada de sistema

1. No nível usuário, a aplicação invoca a função `write(fd, &buffer, bytes)` da biblioteca de sistema (geralmente a biblioteca padrão da linguagem C).

2. A função `write` preenche os registradores da CPU com os parâmetros recebidos e escreve o *opcode* da chamada de sistema `write` no registrador AX.
3. A função `write` invoca uma chamada de sistema, através da instrução `syscall`.
4. O processador comuta para o nível privilegiado (*kernel level*) e transfere o controle para a rotina de entrada (`entry`), apontada por um registrador específico.
5. A rotina de entrada recebe em AX o *opcode* da operação desejada (1, `write`), consulta a tabela de chamadas de sistema mantida pelo núcleo e invoca a função que implementa essa operação dentro do núcleo (`sys_write`).
6. A função `sys_write` obtém o endereço dos parâmetros nos demais registradores, verifica a validade dos mesmos e efetua a operação desejada pela aplicação.
7. Ao final da execução da função, eventuais valores de retorno são escritos nos registradores (ou na área de memória da aplicação) e o processamento retorna à função `write`, em modo usuário.
8. A função `write` finaliza sua execução e retorna o controle ao código principal da aplicação.
9. Caso a operação solicitada não possa ser concluida imediatamente, a função `sys_write` passa o controle para a gerência de atividades, ao invés de retornar para a aplicação solicitante. Isto ocorre, por exemplo, quando é solicitada a leitura de uma entrada do teclado.
10. Na sequência, a gerência de atividades suspende a execução da aplicação solicitante e devolve o controle do processador a alguma outra aplicação que também esteja aguardando o retorno de uma chamada de sistema e cuja operação solicitada já tenha sido concluída.

A maioria dos sistemas operacionais implementa centenas de chamadas de sistema distintas, para as mais diversas finalidades. O conjunto de chamadas de sistema oferecidas por um núcleo define a API (*Application Programming Interface*) do sistema operacional. Exemplos de APIs bem conhecidas são a *Win32*, oferecida pelos sistemas Microsoft derivados do Windows NT, e a API *POSIX* [Gallmeister, 1994], que define um padrão de interface de núcleo para sistemas UNIX.

O conjunto de chamadas de sistema de um SO pode ser dividido nas seguintes grandes áreas:

- Gestão de processos: criar, carregar código, terminar, esperar, ler/mudar atributos.
- Gestão da memória: alocar/liberar/modificar áreas de memória.
- Gestão de arquivos: criar, remover, abrir, fechar, ler, escrever, ler/mudar atributos.
- Comunicação: criar/destruir canais de comunicação, receber/enviar dados.
- Gestão de dispositivos: ler/mudar configurações, ler/escrever dados.
- Gestão do sistema: ler/mudar data e hora, desligar/suspender/reiniciar o sistema.

Referências

F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the Multics system. In *AFIPS Conference Proceedings*, pages 185–196, 1965.

B. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly Media, Inc, 1994.

Google. *Android Platform Architecture*, April 2018. https://developer.android.com/guide/platform.

D. Patterson and J. Henessy. *Organização e Projeto de Computadores*. Campus, 2005.

W. Stallings. *Arquitetura e organização de computadores. 8ª ed*. Pearson, 2010.

Capítulo 3

Arquiteturas de SOs

Embora a definição de níveis de privilégio (Seção 2.3) imponha uma estruturação mínima a um sistema operacional, forçando a separação entre o núcleo e o espaço de usuário, os vários elementos que compõem o sistema podem ser organizados de diversas formas, separando suas funcionalidades e modularizando seu projeto. Neste capítulo serão apresentadas as arquiteturas mais populares para a organização de sistemas operacionais.

3.1 Sistemas monolíticos

Em um sistema monolítico[1], o sistema operacional é um "bloco maciço" de código que opera em modo núcleo, com acesso a todos os recursos do hardware e sem restrições de acesso à memória. Por isso, os componentes internos do sistema operacional podem se relacionar entre si conforme suas necessidades. A Figura 3.1 ilustra essa arquitetura.

A grande vantagem da arquitetura monolítica é seu desempenho: qualquer componente do núcleo pode acessar os demais componentes, áreas de memória ou mesmo dispositivos periféricos diretamente, pois não há barreiras impedindo esses acessos. A interação direta entre componentes leva a sistemas mais rápidos e compactos, pois não há necessidade de utilizar mecanismos específicos de comunicação entre os componentes do núcleo.

Todavia, a arquitetura monolítica pode levar a problemas de robustez do sistema. Como todos os componentes do SO têm acesso privilegiado ao hardware, caso um componente perca o controle devido a algum erro (acessando um ponteiro inválido ou uma posição inexistente em um vetor, por exemplo), esse erro pode se propagar rapidamente por todo o núcleo, levando o sistema ao colapso (travamento, reinicialização ou funcionamento errático).

Outro problema relacionado às arquiteturas monolíticas diz respeito ao processo de desenvolvimento. Como os componentes do sistema têm acesso direto uns aos outros, podem ser fortemente interdependentes, tornando a manutenção e evolução do núcleo mais complexas. Como as dependências e pontos de interação entre os componentes podem ser pouco evidentes, pequenas alterações na estrutura de dados de um componente podem ter um impacto inesperado em outros componentes, caso estes acessem aquela estrutura diretamente.

[1] A palavra "monólito" vem do grego *monos* (único ou unitário) e *lithos* (pedra).

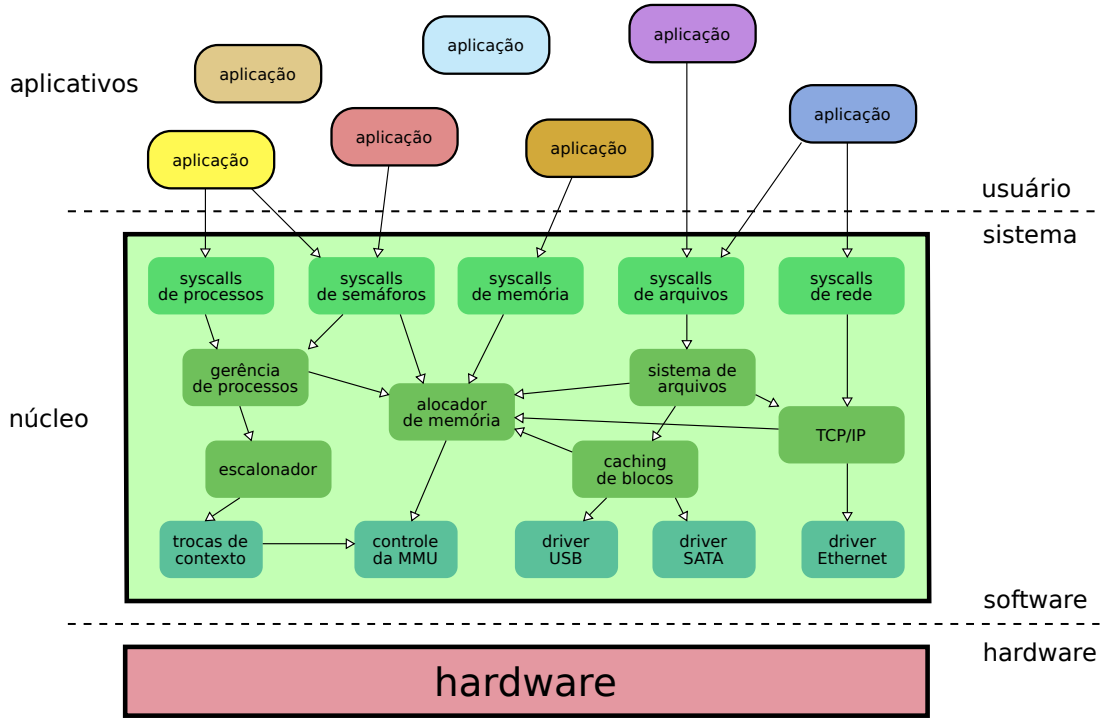


Figura 3.1: Núcleo de sistema operacional monolítico.

A arquitetura monolítica foi a primeira forma de organizar os sistemas operacionais; sistemas UNIX antigos e o MS-DOS seguiam esse modelo. O núcleo do Linux é monolítico, mas seu código vem sendo gradativamente estruturado e modularizado desde a versão 2.0 (embora boa parte do código ainda permaneça no nível de núcleo). O sistema FreeBSD [McKusick and Neville-Neil, 2005] também usa um núcleo monolítico.

3.2 Sistemas micronúcleo

Outra possibilidade de estruturar o SO consiste em retirar do núcleo todo o código de alto nível, normalmente associado às abstrações de recursos, deixando no núcleo somente o código de baixo nível necessário para interagir com o hardware e criar algumas abstrações básicas. O restante do código seria transferido para programas separados no espaço de usuário, denominados *serviços*. Por fazer o núcleo de sistema ficar menor, essa abordagem foi denominada *micronúcleo* (ou *µ-kernel*).

A abordagem micronúcleo oferece maior modularidade, pois cada serviço pode ser desenvolvido de forma independente dos demais; mais flexibilidade, pois os serviços podem ser carregados e desativados conforme a necessidade; e mais robustez, pois caso um serviço falhe, somente ele será afetado, devido ao confinamento de memória entre os serviços.

Um micronúcleo normalmente implementa somente a noção de tarefa, os espaços de memória protegidos para cada aplicação, a comunicação entre tarefas e as operações de acesso às portas de entrada/saída (para acessar os dispositivos). Todos os aspectos de alto nível, como políticas de uso do processador e da memória, o sistema de arquivos, o controle de acesso aos recursos e até mesmos os *drivers* são implementados fora do núcleo, em processos que se comunicam usando o mecanismo de comunicação provido pelo núcleo.

Um bom exemplo de sistema micronúcleo é o Minix 3 [Herder et al., 2006]. Neste sistema, o núcleo oferece funcionalidades básicas de gestão de interrupções, configuração da CPU e da MMU, acesso às portas de entrada/saída e primitivas de troca de mensagens entre aplicações. Todas as demais funcionalidades, como a gestão de arquivos e de memória, protocolos de rede, políticas de escalonamento, etc., são providas por processos fora do núcleo, denominados *servidores*. A Figura 3.2 apresenta a arquitetura do Minix 3:

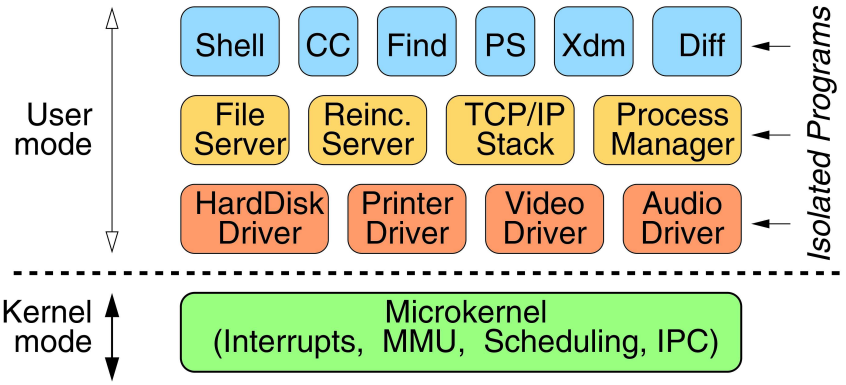


Figura 3.2: Visão geral da arquitetura do MINIX 3 [Herder et al., 2006].

Por exemplo, no sistema Minix 3 a seguinte sequência de ações ocorre quando uma aplicação deseja ler dados de um arquivo no disco:

1. usando as primitivas de mensagens do núcleo, a aplicação envia uma mensagem (m_1) ao servidor de arquivos, solicitando a leitura de dados de um arquivo;
2. o servidor de arquivos verifica se possui os dados em seu cache local; se não os possuir, envia uma mensagem (m_2) ao *driver* de disco solicitando a leitura dos dados do disco em seu espaço de memória;
3. o *driver* de disco envia uma mensagem (m_3) ao núcleo solicitando operações nas portas de entrada/saída do controlador de disco, para ler dados do mesmo;
4. o núcleo verifica se o *driver* de disco tem permissão para usar as portas de entrada/saída e agenda a operação solicitada;
5. quando o disco concluir a operação, o núcleo transfere os dados lidos para a memória do *driver* de disco e responde (m_4) a este;
6. o *driver* de disco solicita (m_5) então ao núcleo a cópia dos dados recebidos para a memória do servidor de arquivos e responde (m_6) à mensagem anterior deste, informando a conclusão da operação;
7. o servidor de arquivos solicita (m_7) ao núcleo a cópia dos dados recebidos para a memória da aplicação e responde (m_8) à mensagem anterior desta;
8. a aplicação recebe a resposta de sua solicitação e usa os dados lidos.

Da sequência acima pode-se perceber que foram necessárias 8 mensagens (m_i) para realizar uma leitura de dados do disco, cada uma correspondendo a uma chamada

de sistema. Cada chamada de sistema tem um custo elevado, pois implica na mudança do fluxo de execução e reconfiguração da memória acessível pela MMU. Além disso, foram necessárias várias cópias de dados entre as áreas de memória de cada entidade. Esses fatores levam a um desempenho bem inferior ao da abordagem monolítica, razão que dificulta a adoção plena dessa abordagem.

O micronúcleos vem sendo investigados desde os anos 1980. Dois exemplos clássicos dessa abordagem são os sistemas Mach [Rashid et al., 1989] e Chorus [Rozier and Martins, 1987]. Os melhores exemplos de micronúcleos bem sucedidos são provavelmente o Minix 3 [Herder et al., 2006] e o QNX. Contudo, vários sistemas operacionais atuais adotam parcialmente essa estruturação, adotando núcleos híbridos, como por exemplo o MacOS X da Apple, o Digital UNIX e o Windows NT.

3.3 Sistemas em camadas

Uma forma mais elegante de estruturar um sistema operacional faz uso da noção de camadas: a camada mais baixa realiza a interface com o hardware, enquanto as camadas intermediárias proveem níveis de abstração e gerência cada vez mais sofisticados. Por fim, a camada superior define a interface do núcleo para as aplicações (as chamadas de sistema). As camadas têm níveis de privilégio decrescentes: a camada inferior tem acesso total ao hardware, enquanto a superior tem acesso bem mais restrito (vide Seção 2.2.3).

A abordagem de estruturação de software em camadas teve sucesso no domínio das redes de computadores, através do modelo de referência OSI (*Open Systems Interconnection*) [Day, 1983], e também seria de se esperar sua adoção no domínio dos sistemas operacionais. No entanto, alguns inconvenientes limitam a aplicação do modelo em camadas de forma intensiva nos sistemas operacionais:

- O empilhamento de várias camadas de software faz com que cada pedido de uma aplicação demore mais tempo para chegar até o dispositivo periférico ou recurso a ser acessado, prejudicando o desempenho.

- Nem sempre a divisão de funcionalidades do sistema em camadas é óbvia, pois muitas dessas funcionalidades são interdependentes e não teriam como ser organizadas em camadas. Por exemplo, a gestão de entrada/saída necessita de serviços de memória para alocar/liberar *buffers* para leitura e escrita de dados, mas a gestão de memória precisa da gestão de entrada/saída para implementar a paginação em disco (*paging*). Qual dessas duas camadas viria antes?

Em decorrência desses inconvenientes, a estruturação em camadas é apenas parcialmente adotada hoje em dia. Como exemplo de sistema fortemente estruturado em camadas pode ser citado o MULTICS [Corbató and Vyssotsky, 1965].

Muitos sistemas mais recentes implementam uma camada inferior de abstração do hardware para interagir com os dispositivos (a camada *HAL – Hardware Abstraction Layer*, implementada no Windows NT e seus sucessores), e também organizam em camadas alguns subsistemas, como a gerência de arquivos e o suporte de rede (seguindo o modelo OSI). Essa organização parcial em camadas pode ser facilmente observada nas arquiteturas do Minix 3 (Figura 3.2) Windows 2000 (Figura 3.3) e Android (Figura 2.2).

3.4 Sistemas híbridos

Apesar das boas propriedades de modularidade, flexibilidade e robustez proporcionadas pelos micronúcleos, sua adoção não teve o sucesso esperado devido ao baixo desempenho. Uma solução encontrada para esse problema consiste em trazer de volta ao núcleo os componentes mais críticos, para obter melhor desempenho. Essa abordagem intermediária entre o núcleo monolítico e micronúcleo é denominada *núcleo híbrido*. É comum observar também nos núcleos híbridos uma influência da arquitetura em camadas.

Vários sistemas operacionais atuais empregam núcleos híbridos, entre eles o Microsoft Windows, a partir do Windows NT. As primeiras versões do Windows NT podiam ser consideradas micronúcleo, mas a partir da versão 4 vários subsistemas foram movidos para dentro do núcleo para melhorar seu desempenho, transformando-o em um núcleo híbrido. Os sistemas MacOS e iOS da Apple também adotam um núcleo híbrido chamado XNU (de *X is Not Unix*), construído a partir dos núcleos Mach (micronúcleo) [Rashid et al., 1989] e FreeBSD (monolítico) [McKusick and Neville-Neil, 2005]. A figura 3.3 mostra a arquitetura interna do núcleo usado no SO Windows 2000.

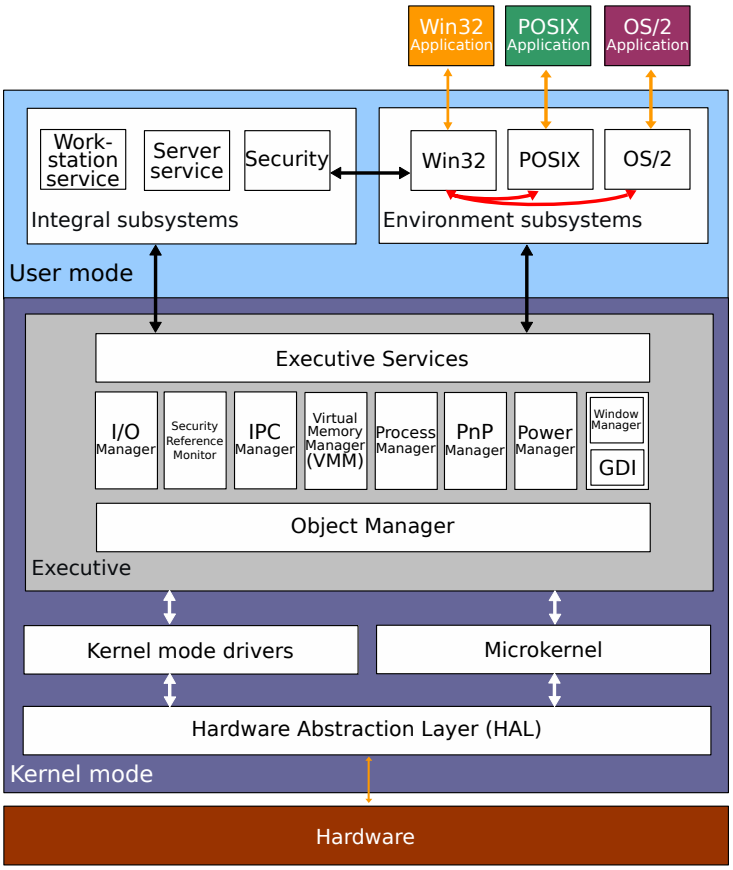


Figura 3.3: Arquitetura do sistema Windows 2000 [Wikipedia, 2018].

3.5 Arquiteturas avançadas

Além das arquiteturas clássicas (monolítica, em camadas, micronúcleo), recentemente surgiram várias propostas organizar os componentes do sistema operacional,

com vistas a contextos de aplicação específicos, como nuvens computacionais. Esta seção apresenta algumas dessas novas arquiteturas.

3.5.1 Máquinas virtuais

Apesar de ser um conceito dos anos 1960 [Goldberg, 1973], a virtualização de sistemas e aplicações ganhou um forte impulso a partir dos anos 2000, com a linguagem Java, a consolidação de servidores (juntar vários servidores com seus sistemas operacionais em um único computador) e, mais recentemente, a computação em nuvem.

Uma máquina virtual é uma camada de software que "transforma" um sistema em outro, ou seja, que usa os serviços fornecidos por um sistema operacional (ou pelo hardware) para construir a interface de outro sistema. Por exemplo, o ambiente (JVM - *Java Virtual Machine*) usa os serviços de um sistema operacional convencional (Windows ou Linux) para construir um computador virtual que processa *bytecode*, o código binário dos programas Java. Da mesma forma, o ambiente *VMWare* permite executar o sistema Windows sobre um sistema Linux subjacente, ou vice-versa.

Um ambiente de máquinas virtuais consiste de três partes básicas, que podem ser observadas nos exemplos da Figura 3.4:

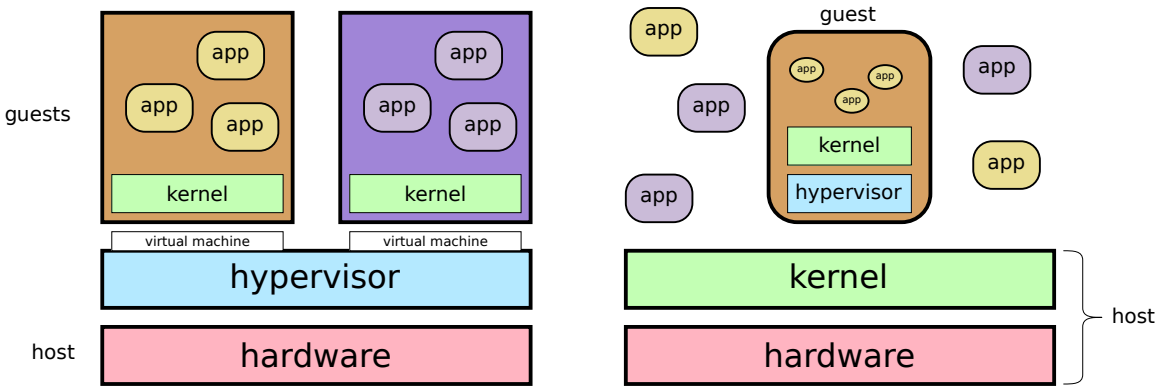


Figura 3.4: Ambientes de virtualização de sistema (esq.) e de aplicação (dir.).

- O sistema real, ou sistema hospedeiro (*host*), que contém os recursos reais de hardware e software do sistema;
- a camada de virtualização, denominada *hipervisor* ou *monitor de virtualização* (*VMM - Virtual Machine Monitor*), que constrói a máquina virtual a partir dos recursos do sistema real;
- o sistema virtual, ou sistema convidado (*guest*); em alguns casos, vários sistemas virtuais podem coexistir, executando sobre o mesmo sistema real.

Existem diversas possibilidades de uso da virtualização, levando a vários tipos de hipervisor, por exemplo:

Hipervisor nativo: executa diretamente sobre o hardware, virtualizando os recursos deste para construir máquinas virtuais, onde executam vários sistemas operacionais com suas respectivas aplicações. O exemplo da esquerda na Figura

3.4 ilustra esse conceito. Um exemplo deste tipo de hipervisor é o ambiente *VMware ESXi Server* [Newman et al., 2005].

Hipervisor convidado: executa sobre um sistema operacional hospedeiro, como no exemplo da direita na Figura 3.4. Os hipervisores *VMWare Workstation* e *VirtualBox* implementam essa abordagem.

Hipervisor de sistema: suporta a execução de um sistema operacional convidado com suas aplicações, como nos dois exemplos da Figura 3.4. *Xen*, *KVM* e *VMWare* são exemplos desta categoria.

Hipervisor de aplicação: suporta a execução de uma única aplicação em uma linguagem específica. As máquinas virtuais JVM (*Java Virtual Machine*) e CLR (*Common Language Runtime*, para C#) são exemplos típicos dessa abordagem.

Por permitir a execução de vários sistemas independentes e isolados entre si sobre o mesmo hardware, hipervisores de sistema são muito usados em ambientes corporativos de larga escala, como as nuvens computacionais. Hipervisores de aplicação também são muito usados, por permitir a execução de código independente de plataforma, como em Java. Avanços recentes no suporte de hardware para a virtualização e na estrutura interna dos hipervisores permitem a execução de máquinas virtuais com baixo custo adicional em relação à execução nativa.

O capítulo 31.4 deste livro contém uma apresentação mais detalhada do conceito de virtualização e de suas possibilidades de uso.

3.5.2 Contêineres

Além da virtualização, outra forma de isolar aplicações ou subsistemas em um sistema operacional consiste na virtualização do espaço de usuário (*userspace*). Nesta abordagem, denominada *servidores virtuais* ou *contêineres*, o espaço de usuário do sistema operacional é dividido em áreas isoladas denominadas *domínios* ou *contêineres*. A cada domínio é alocada uma parcela dos recursos do sistema operacional, como memória, tempo de processador e espaço em disco.

Para garantir o isolamento entre os domínios, cada domínio tem sua própria interface de rede virtual e, portanto, seu próprio endereço de rede. Além disso, na maioria das implementações cada domínio tem seu próprio espaço de nomes para os identificadores de usuários, processos e primitivas de comunicação. Assim, é possível encontrar um usuário `pedro` no domínio d_3 e outro usuário `pedro` no domínio d_7, sem relação entre eles nem conflitos. Essa noção de espaços de nomes distintos se estende aos demais recursos do sistema: identificadores de processos, semáforos, árvores de diretórios, etc. Entretanto, o núcleo do sistema operacional é o mesmo para todos os domínios.

Processos em um mesmo domínio podem interagir entre si normalmente, mas processos em domínios distintos não podem ver ou interagir entre si. Além disso, processos não podem trocar de domínio nem acessar recursos de outros domínios. Dessa forma, os domínios são vistos como sistemas distintos, acessíveis somente através de seus endereços de rede. Para fins de gerência, normalmente é definido um domínio d_0, chamado de *domínio inicial*, *privilegiado* ou *de gerência*, cujos processos têm visibilidade e acesso aos processos e recursos dos demais domínios.

A Figura 3.5 mostra a estrutura típica de um ambiente de contêineres. Nela, pode-se observar que um processo pode migrar de d_0 para d_1, mas que os processos em d_1 não podem migrar para outros domínios. A comunicação entre processos confinados em domínios distintos (d_2 e d_3) também é proibida.

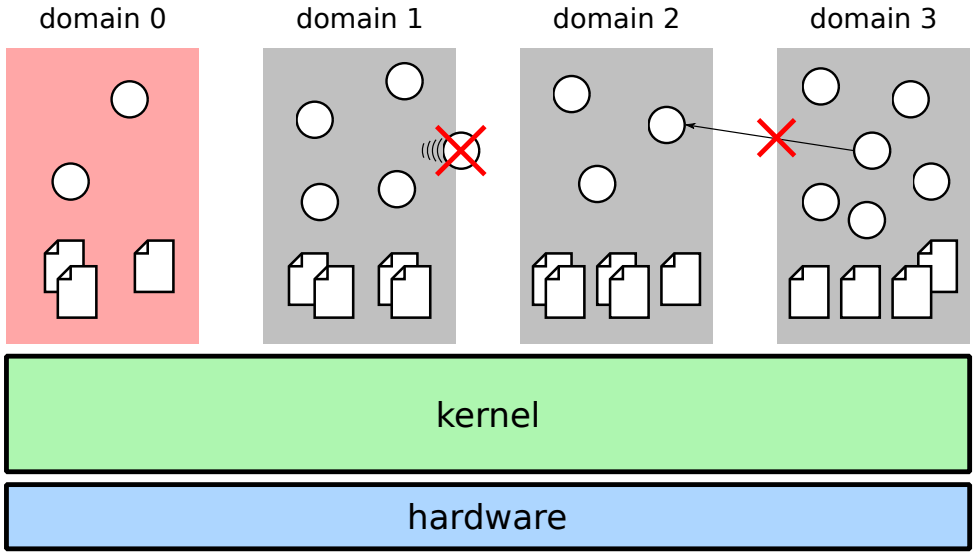


Figura 3.5: Sistema com contêineres (domínios).

Existem implementações de contêineres, como as *Jails* do sistema FreeBSD [McKusick and Neville-Neil, 2005] ou as *zonas* do sistema Solaris [Price and Tucker, 2004], que oferecem o isolamento entre domínios e o controle de uso dos recursos pelos mesmos. O núcleo Linux oferece diversos mecanismos para o isolamento de espaços de recursos, que são usados por plataformas de gerenciamento de contêineres como *Linux Containers* (LXC), *Docker* e *Kubernetes*.

3.5.3 Sistemas exonúcleo

Em um sistema convencional, as aplicações executam sobre uma pilha de serviços de abstração e gerência de recursos providos pelo sistema operacional. Esses serviços facilitam a construção de aplicações, mas podem constituir uma sobrecarga significativa para o desempenho do sistema. Para cada abstração ou serviço utilizado é necessário interagir com o núcleo através de chamadas de sistema, que impactam negativamente o desempenho. Além disso, os serviços providos pelo núcleo devem ser genéricos o suficiente para atender a uma vasta variedade de aplicações. Aplicações com demandas muito específicas, como um servidor Web de alto desempenho, nem sempre são atendidas da melhor forma possível.

Os exonúcleos (*exokernels*) [Engler, 1998] tentam trazer uma resposta a essas questões, reduzindo ainda mais os serviços oferecidos pelo núcleo. Em um sistema exonúcleo, o núcleo do sistema apenas proporciona acesso controlado aos recursos do hardware, mas não implementa nenhuma abstração. Por exemplo, o núcleo provê acesso compartilhado à interface de rede, mas não implementa nenhum protocolo. Todas as abstrações e funcionalidades de gerência que uma aplicação precisa terão de ser implementadas pela própria aplicação, em seu espaço de memória.

Para simplificar a construção de aplicações sobre exonúcleos, são usados sistemas operacionais "de biblioteca" ou *LibOS* (*Library Operating System*), que implementam as

abstrações usuais, como memória virtual, sistemas de arquivos, semáforos e protocolos de rede. Um LibOS nada mais é que um conjunto de bibliotecas compartilhadas usadas pela aplicação para prover os serviços de que esta necessita. Diferentes aplicações sobre o mesmo exonúcleo podem usar LibOS diferentes, ou implementar suas próprias abstrações. A Figura 3.6 ilustra a arquitetura de um exonúcleo típico.

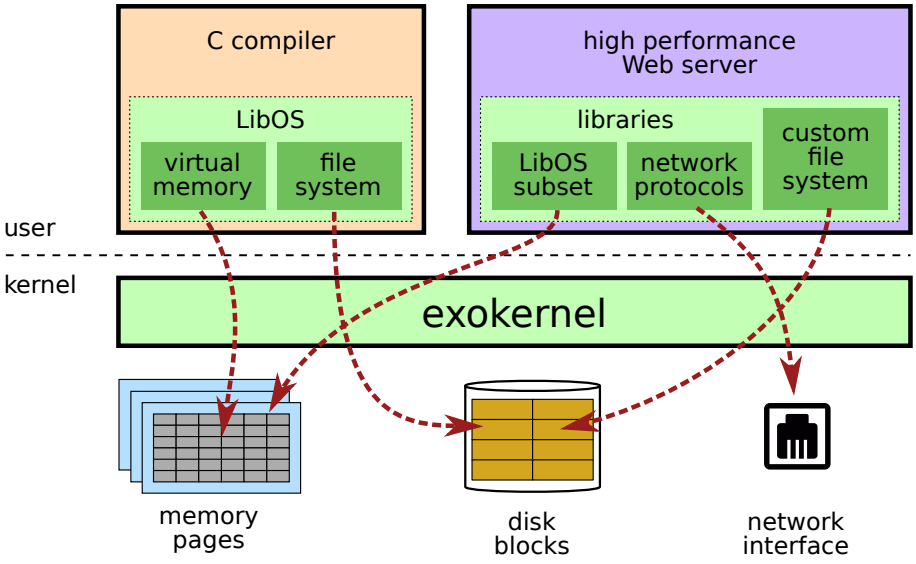


Figura 3.6: Sistema exonúcleo (adaptado de [Engler, 1998]).

É importante observar que um exonúcleo é diferente de um micronúcleo, pois neste último as abstrações são implementada por um conjunto de processos servidores independentes e isolados entre si, enquanto no exonúcleo cada aplicação implementa suas próprias abstrações (ou as incorpora de bibliotecas compartilhadas).

Até o momento não existem exonúcleos em produtos comerciais, as implementações mais conhecidas são esforços de projetos de pesquisa, como Aegis/ExOS [Engler, 1998] e Nemesis [Hand, 1999].

3.5.4 Sistemas uninúcleo

Nos sistemas uninúcleo (*unikernel*) [Madhavapeddy and Scott, 2013], um núcleo de sistema operacional, as bibliotecas e uma aplicação são compilados e ligados entre si, formando um bloco monolítico de código, que executa em um único espaço de endereçamento, em modo privilegiado. Dessa forma, o custo da transição aplicação/núcleo nas chamadas de sistema diminui muito, gerando um forte ganho de desempenho.

Outro ponto positivo da abordagem uninúcleo é o fato de incluir no código final somente os componentes necessários para suportar a aplicação-alvo e os *drivers* necessários para acessar o hardware-alvo, levando a um sistema compacto, que pode ser lançado rapidamente.

A estrutura de uninúcleo se mostra adequada para computadores que executam uma única aplicação, como servidores de rede (HTTP, DNS, DHCP) e serviços em ambientes de nuvem computacional (bancos de dados, Web Services). De fato, o maior interesse em usar uninúcleos vem do ambiente de nuvem, pois os "computadores" da nuvem são na verdade máquinas virtuais com hardware virtual simples (com *drivers* padronizados) e que executam serviços isolados (vide Seção 3.5.1).

Os sistemas OSv [Kivity et al., 2014] e MirageOS [Madhavapeddy and Scott, 2013] são bons exemplos de uninúcleos. Como eles foram especialmente concebidos para as nuvens, são também conhecidos como "CloudOS". A Figura 3.7 mostra a arquitetura desses sistemas.

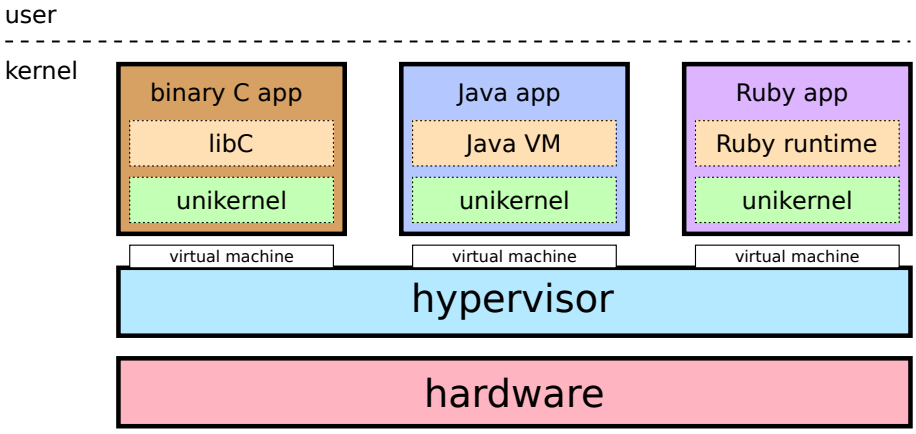


Figura 3.7: Sistema uninúcleo.

É interessante observar que, em sistemas embarcados e nas chamadas *appliances* de rede (roteadores, modems, etc), é usual compilar o núcleo, bibliotecas e aplicação em um único arquivo binário que é em seguida executado diretamente sobre o hardware, em modo privilegiado. Este é o caso, por exemplo, do sistema TinyOS [Levis et al., 2005], voltado para aplicações de Internet das Coisas (IoT - *Internet of Things*). Apesar desse sistema não ser oficialmente denominado um uninúcleo, a abordagem basicamente é a mesma.

Referências

F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the Multics system. In *AFIPS Conference Proceedings*, pages 185–196, 1965.

J. Day. The OSI reference model. *Proceedings of the IEEE*, December 1983.

D. Engler. *The Exokernel Operating System Architecture*. PhD thesis, Massachusetts Institute of Technology, Cambridge - MA, October 1998.

R. Goldberg. Architecture of virtual machines. In *AFIPS National Computer Conference*, 1973.

S. Hand. Self-paging in the Nemesis operating system. In *3rd Symposium on Operating Systems Design and Implementation (OSDI)*, volume 99, pages 73–86, 1999.

J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40 (3), Sept. 2006.

A. Kivity, D. L. G. Costa, and P. Enberg. OSv – optimizing the operating system for virtual machines. In *USENIX Annual Technical Conference*, page 61, 2014.

P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. TinyOS: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

A. Madhavapeddy and D. J. Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30:30–30:44, Dec. 2013. ISSN 1542-7730.

M. McKusick and G. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2005.

M. Newman, C.-M. Wiberg, and B. Braswell. *Server Consolidation with VMware ESX Server*. IBM RedBooks, 2005. http://www.redbooks.ibm.com.

D. Price and A. Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *18th USENIX conference on System administration*, pages 241–254, 2004.

R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. B. Jones. Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON*, pages 176–178, San Francisco, CA, USA, 1989. IEEE Comput. Soc. Press.

M. Rozier and J. L. Martins. The Chorus distributed operating system: Some design issues. In Y. Paker, J.-P. Banatre, and M. Bozyiğit, editors, *Distributed Operating Systems*, pages 261–287, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

Wikipedia. Wikipedia online enciclopedia. http://www.wikipedia.org, 2018.

Parte II

Gestão de tarefas

Capítulo 4

O conceito de tarefa

Um sistema de computação quase sempre tem mais atividades a executar que o número de processadores disponíveis. Assim, é necessário criar métodos para multiplexar o(s) processador(es) da máquina entre as atividades presentes. Além disso, como as diferentes tarefas têm necessidades distintas de processamento e nem sempre a capacidade de processamento existente é suficiente para atender a todos, estratégias precisam ser definidas para que cada tarefa receba uma quantidade de processamento que atenda suas necessidades. Este capítulo apresenta os principais conceitos, estratégias e mecanismos empregados na gestão do processador e das atividades em execução em um sistema de computação.

4.1 Objetivos

Em um sistema de computação, é frequente a necessidade de executar várias tarefas distintas simultaneamente. Por exemplo:

- O usuário de um computador pessoal pode estar editando uma imagem, imprimindo um relatório, ouvindo música e trazendo da Internet um novo software, tudo ao mesmo tempo.
- Em um grande servidor de e-mails, milhares de usuários conectados remotamente enviam e recebem e-mails através da rede.
- Um navegador Web precisa buscar os elementos da página a exibir, analisar e renderizar o código HTML e os gráficos recebidos, animar os elementos da interface e responder aos comandos do usuário.

Por outro lado, um processador convencional somente trata um fluxo de instruções de cada vez. Até mesmo computadores com vários processadores, vários *cores* ou com tecnologia *hyper-threading*, por exemplo, têm mais atividades a executar que o número de processadores disponíveis. Como fazer para atender simultaneamente as múltiplas necessidades de processamento dos usuários?

Uma solução ingênua para esse problema seria equipar o sistema com um processador para cada tarefa, mas essa solução ainda é inviável econômica e tecnicamente. Outra solução seria *multiplexar o processador* entre as várias tarefas que requerem processamento, ou seja, compartilhar o uso do processador entre as várias tarefas, de forma a atendê-las da melhor maneira possível.

Para uma gestão eficaz do processamento, é fundamental compreender o conceito de *tarefa*. O restante deste capítulo aborda o conceito de tarefa, como estas são definidas, quais os seus estados possíveis e como/quando elas mudam de estado.

4.2 O conceito de tarefa

Uma tarefa é definida como sendo a execução de um fluxo sequencial de instruções, construído para atender uma finalidade específica: realizar um cálculo complexo, a edição de um gráfico, a formatação de um disco, etc. Assim, a execução de uma sequência de instruções em linguagem de máquina, normalmente gerada pela compilação de um programa escrito em uma linguagem qualquer, é denominada "tarefa" ou "atividade" (do inglês *task*).

É importante ressaltar as diferenças entre os conceitos de *tarefa* e de *programa*:

Programa: é um conjunto de uma ou mais sequências de instruções escritas para resolver um problema específico, constituindo assim uma aplicação ou utilitário. O programa representa um conceito *estático*, sem um estado interno definido (que represente uma situação específica da execução) e sem interações com outras entidades (o usuário ou outros programas). Os arquivos `C:\Windows\notepad.exe` e `/usr/bin/vi` são exemplos de programas (no caso, para edição de texto).

Tarefa: é a execução sequencial, por um processador, da sequência de instruções definidas em um programa para realizar seu objetivo. Trata-se de um conceito *dinâmico*, que possui um estado interno bem definido a cada instante (os valores das variáveis internas e a posição atual da execução evoluem com o tempo) e interage com outras entidades: o usuário, os dispositivos periféricos e/ou outras tarefas. Tarefas podem ser implementadas de várias formas, como processos (Seção 5.3) ou *threads* (Seção 5.4).

Fazendo uma analogia simples, pode-se dizer que um programa é o equivalente de uma "receita de torta" dentro de um livro de receitas (um diretório) guardado em uma estante (um disco) na cozinha (o computador). Essa receita de torta define os ingredientes necessários (entradas) e o modo de preparo (programa) da torta (saída). Por sua vez, a ação de "executar" a receita, providenciando os ingredientes e seguindo os passos definidos na mesma, é a tarefa propriamente dita. A cada momento, a cozinheira (o processador) está seguindo um passo da receita (posição da execução) e tem uma certa disposição dos ingredientes e utensílios em uso (as entradas e variáveis internas da tarefa).

Assim como uma receita de torta pode definir várias atividades interdependentes para elaborar a torta (preparar a massa, fazer o recheio, assar, decorar, etc.), um programa também pode definir várias sequências de execução interdependentes para atingir seus objetivos. Por exemplo, o programa do navegador Web ilustrado na Figura 4.1 define várias tarefas que uma janela de navegador deve executar simultaneamente, para que o usuário possa navegar na Internet:

1. buscar via rede os vários elementos que compõem a página Web;
2. receber, analisar e renderizar o código HTML e os gráficos recebidos;

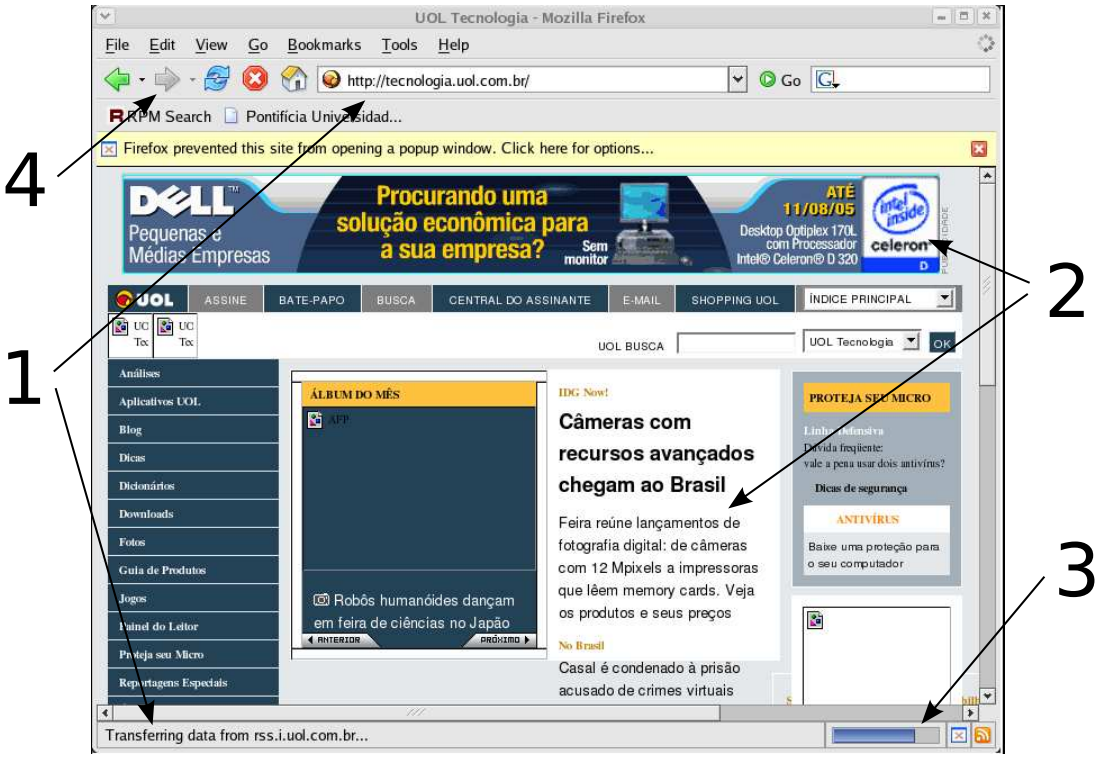


Figura 4.1: Tarefas de um navegador Internet

3. animar os diferentes elementos que compõem a interface do navegador;

4. receber e tratar os eventos do usuário (*clicks*) nos botões do navegador.

Dessa forma, as tarefas definem as atividades a serem realizadas dentro do sistema de computação. Como geralmente há muito mais tarefas a realizar que processadores disponíveis, e as tarefas não têm todas a mesma importância, a gerência de tarefas tem uma grande importância dentro de um sistema operacional.

4.3 A gerência de tarefas

Em um computador, o processador tem de executar todas as tarefas submetidas pelos usuários. Essas tarefas geralmente têm comportamento, duração e importância distintas. Cabe ao sistema operacional organizar as tarefas para executá-las e decidir em que ordem fazê-lo. Nesta seção será estudada a organização básica do sistema de gerência de tarefas e sua evolução histórica.

4.3.1 Sistemas monotarefa

Os primeiros sistemas de computação, nos anos 40, executavam apenas uma tarefa de cada vez. Nestes sistemas, cada programa binário era carregado do disco para a memória e executado até sua conclusão. Os dados de entrada da tarefa eram carregados na memória junto à mesma e os resultados obtidos no processamento eram descarregados de volta no disco após a conclusão da tarefa. Todas as operações de transferência de código e dados entre o disco e a memória eram coordenados por um

operador humano. Esses sistemas primitivos eram usados sobretudo para aplicações de cálculo numérico, muitas vezes com fins militares (problemas de trigonometria, balística, mecânica dos fluidos, etc.).

A Figura 4.2 ilustra um sistema desse tipo. Nessa figura, a etapa 1 corresponde à carga do código na memória, a etapa 2 à carga dos dados de entrada na memória, a etapa 3 ao processamento (execução propriamente dita), consumindo dados e produzindo resultados, e a etapa 4 ao término da execução, com a descarga no disco dos resultados de saída produzidos.

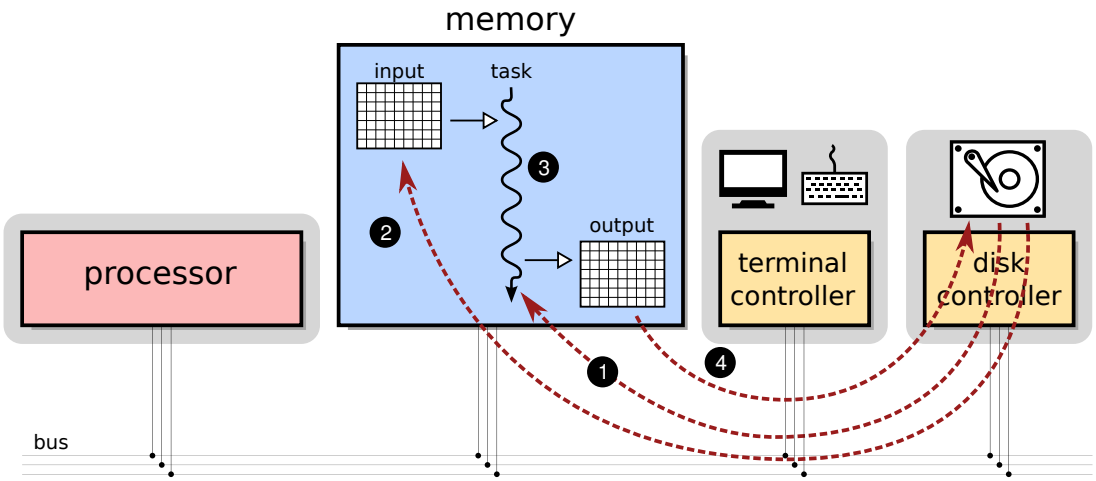


Figura 4.2: Execução de tarefa em um sistema monotarefa.

Nesse método de processamento de tarefas é possível delinear um diagrama de estados para cada tarefa executada pelo sistema, que está representado na Figura 4.3.

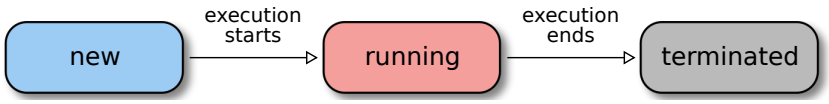


Figura 4.3: Estados de uma tarefa em um sistema monotarefa.

4.3.2 O monitor de sistema

Com a evolução do hardware, as tarefas de carga e descarga de código entre memória e disco, coordenadas por um operador humano, passaram a se tornar críticas: mais tempo era perdido nesses procedimentos manuais que no processamento da tarefa em si. Para resolver esse problema foi construído um *programa monitor*, que era carregado na memória no início da operação do sistema, com a função de gerenciar a execução dos demais programas. O programa monitor executava continuamente os seguintes passos sobre uma fila de programas a executar, armazenada no disco:

1. carregar um programa do disco para a memória;
2. carregar os dados de entrada do disco para a memória;
3. transferir a execução para o programa recém carregado;
4. aguardar o término da execução do programa;

5. escrever os resultados gerados pelo programa no disco.

Percebe-se claramente que a função do monitor é gerenciar uma fila de programas a executar, mantida no disco. Na medida em que os programas são executados pelo processador, novos programas podem ser inseridos na fila pelo operador do sistema. Além de coordenar a execução dos demais programas, o monitor também colocava à disposição destes uma biblioteca de funções para simplificar o acesso aos dispositivos de hardware (teclado, leitora de cartões, disco, etc.). Assim, o monitor de sistema constitui o precursor dos sistemas operacionais.

4.3.3 Sistemas multitarefas

O uso do monitor de sistema agilizou o uso do processador, mas outros problemas persistiam. Como a velocidade de processamento era muito maior que a velocidade de comunicação com os dispositivos de entrada e saída, o processador ficava ocioso durante os períodos de transferência de informação entre disco e memória[1]. Se a operação de entrada/saída envolvesse fitas magnéticas, o processador podia ficar parado vários minutos, aguardando a tranferência de dados. O custo dos computadores e seu consumo de energia eram elevados demais para deixá-los ociosos por tanto tempo.

A solução encontrada para resolver esse problema foi permitir ao monitor suspender a execução da tarefa que espera dados externos e passar a executar outra tarefa. Mais tarde, quando os dados de que a tarefa suspensa necessita estiverem disponíveis, ela pode ser retomada no ponto onde parou. Para tal, é necessário ter mais memória (para poder carregar mais de um programa ao mesmo tempo) e criar mecanismos no monitor para suspender uma tarefa e retomá-la mais tarde.

Uma forma simples de implementar a suspensão e retomada de tarefas de forma transparente consiste no monitor fornecer um conjunto de rotinas padronizadas de entrada/saída à tarefas; essas rotinas implementadas pelo monitor recebem as solicitações de entrada/saída de dados das tarefas e podem suspender uma execução quando for necessário, devolvendo o controle ao monitor.

A Figura 4.4 ilustra o funcionamento de um sistema multitarefa. Os passos numerados representam a execução (*start*), suspensão (*read*), retomada (*resume*) e conclusão (*exit*) de tarefas pelo monitor. Na figura, a tarefa *A* é suspensa ao solicitar uma leitura de dados, sendo retomada mais tarde, após a execução da tarefa *B*.

Essa evolução levou a sistemas mais produtivos (e complexos), nos quais várias tarefas podiam estar em andamento simultaneamente: uma estava ativa (executando) e as demais prontas (esperando pelo processador) ou suspensas (esperando dados ou eventos externos). O diagrama de estados da Figura 4.5 ilustra o comportamento de uma tarefa em um sistema desse tipo:

4.3.4 Sistemas de tempo compartilhado

Solucionado o problema de evitar a ociosidade do processador, restavam no entanto vários outros problemas a resolver. Por exemplo, o programa a seguir contém

[1] Essa diferença de velocidades permanece imensa nos sistemas atuais. Por exemplo, em um computador atual a velocidade de acesso à memória é de cerca de 5 nanossegundos ($5 \times 10^{-9}s$), enquanto a velocidade de acesso a dados em um disco rígido SATA é de cerca de 5 milissegundos ($5 \times 10^{-3}s$), ou seja, um milhão de vezes mais lento!

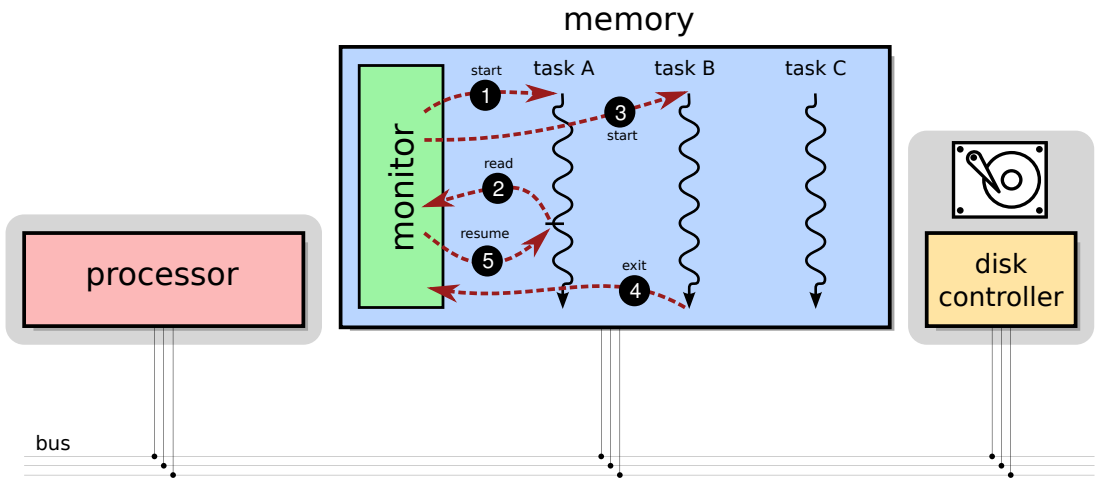


Figura 4.4: Execução de tarefas em um sistema multitarefas.

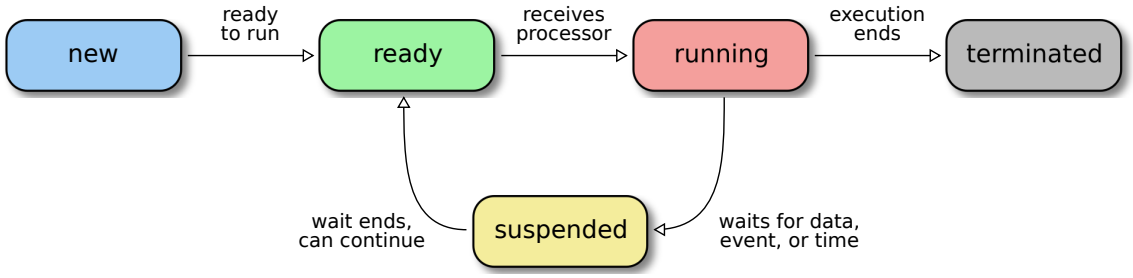


Figura 4.5: Diagrama de estados de uma tarefa em um sistema multitarefas.

um laço infinito; quando uma tarefa executar esse código, ela nunca encerrará nem será suspensa aguardando operações de entrada/saída de dados:

```
1  // calcula a soma dos primeiros 1000 inteiros
2
3  #include <stdio.h>
4
5  int main ()
6  {
7    int i = 0, soma = 0 ;
8
9    while (i <= 1000)
10     soma += i ;      // erro: o contador i não foi incrementado
11
12   printf ("A soma vale %d\n", soma);
13   exit(0) ;
14 }
```

Esse tipo de programa podia inviabilizar o funcionamento do sistema, pois a tarefa em execução nunca termina nem solicita operações de entrada/saída, monopolizando o processador e impedindo a execução das demais tarefas (pois o controle nunca volta ao monitor). Além disso, essa solução não era adequada para a criação de aplicações interativas. Por exemplo, a tarefa de um terminal de comandos pode ser suspensa a cada leitura de teclado, perdendo o processador. Se ela tiver de esperar muito para voltar ao processador, a interatividade com o usuário fica prejudicada.

Para resolver essa questão, foi introduzido no início dos anos 60 um novo conceito: o *compartilhamento de tempo*, ou *time-sharing*, através do sistema CTSS – *Compatible Time-Sharing System* [Corbató, 1963]. Nessa solução, para cada atividade que recebe o processador é definido um prazo de processamento, denominado *fatia de tempo* ou *quantum*[2]. Esgotado seu *quantum*, a tarefa em execução perde o processador e volta para uma fila de tarefas "prontas", que estão na memória aguardando sua oportunidade de executar, e outra tarefa é ativada.

O ato de retirar um recurso "à força" de uma tarefa (neste caso, o processador) é denominado *preempção*. Sistemas que implementam esse conceito são chamados *sistemas preemptivos*. Em um sistema operacional típico, a implementação da preempção por tempo usa as interrupções geradas por um temporizador programável disponível no hardware. Esse temporizador é programado para gerar interrupções em intervalos regulares (a cada milissegundo, por exemplo) que são recebidas por um tratador de interrupção (*interrupt handler*) e encaminhadas ao núcleo; essas ativações periódicas do tratador de interrupção são normalmente chamadas de *ticks*.

Quando uma tarefa recebe o processador, o *núcleo* ajusta um contador de *ticks* que essa tarefa pode usar, ou seja, seu quantum é definido em número de *ticks*. A cada *tick*, esse contador é decrementado; quando ele chegar a zero, a tarefa perde o processador e volta à fila de tarefas prontas. Essa dinâmica de execução está ilustrada na Figura 4.6.

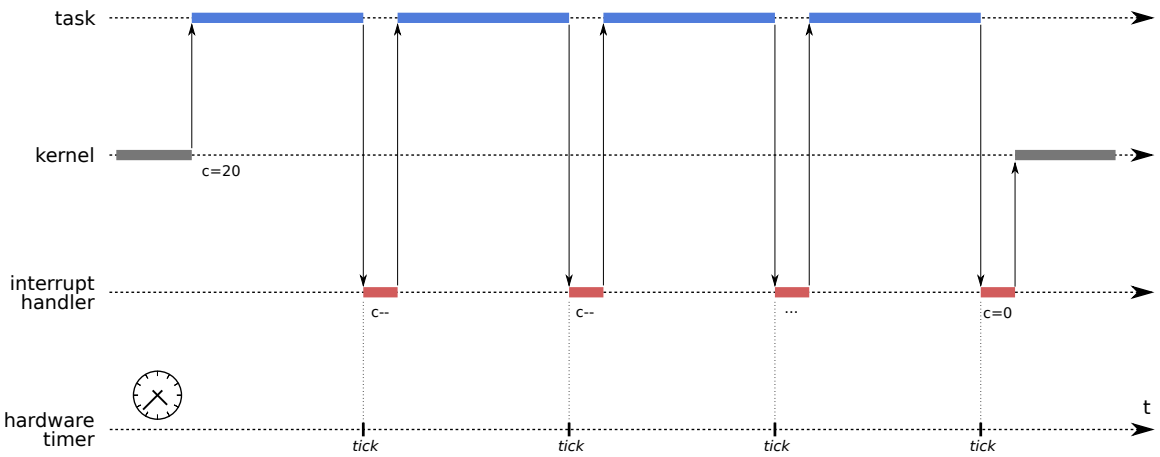


Figura 4.6: Dinâmica da preempção por tempo.

O diagrama de estados das tarefas da Figura 4.5 deve ser portanto reformulado para incluir a preempção por tempo que implementa a estratégia de tempo compartilhado. A Figura 4.7 apresenta esse novo diagrama.

[2] A duração do *quantum* depende muito do tipo de sistema operacional; no Linux ela varia de 10 a 200 milissegundos, dependendo do tipo e prioridade da tarefa [Love, 2010].

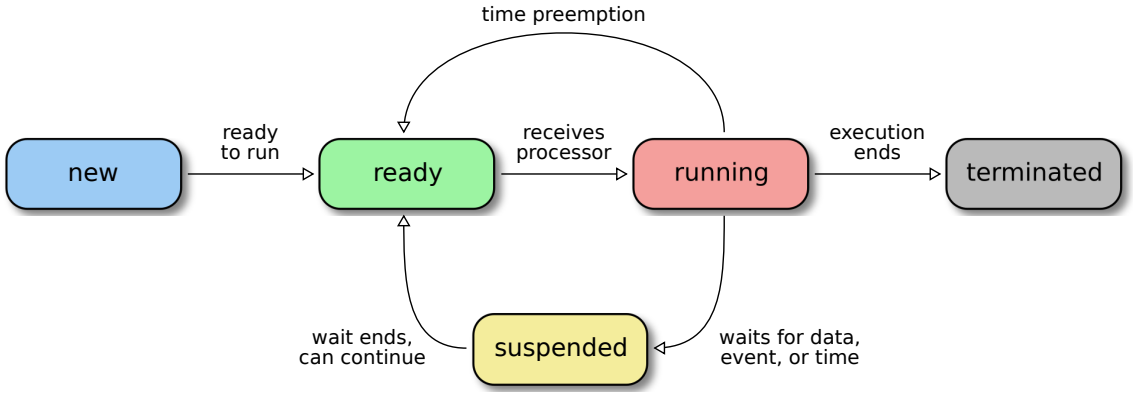


Figura 4.7: Diagrama de estados de uma tarefa em um sistema de tempo compartilhado.

4.4 Ciclo de vida das tarefas

O diagrama apresentado na Figura 4.7 é conhecido na literatura da área como *diagrama de ciclo de vida das tarefas*. Os estados e transições do ciclo de vida têm o seguinte significado:

Nova: A tarefa está sendo criada, i.e. seu código está sendo carregado em memória, junto com as bibliotecas necessárias, e as estruturas de dados do núcleo estão sendo atualizadas para permitir sua execução.

Pronta: A tarefa está em memória, pronta para iniciar ou retomar sua execução, apenas aguardando a disponibilidade do processador. Todas as tarefas prontas são organizadas em uma fila (*fila de prontas*, *ready queue* ou *run queue*), cuja ordem é determinada por algoritmos de escalonamento, que são estudados na Seção 6.

Executando: O processador está dedicado à tarefa, executando suas instruções e fazendo avançar seu estado.

Suspensa: A tarefa não pode executar porque depende de dados externos ainda não disponíveis (do disco ou da rede, por exemplo), aguarda algum tipo de sincronização (o fim de outra tarefa ou a liberação de algum recurso compartilhado) ou simplesmente espera o tempo passar (em uma operação *sleeping*, por exemplo).

Terminada: O processamento da tarefa foi encerrado e ela pode ser removida da memória do sistema.

Tão importantes quanto os estados das tarefas apresentados na Figura 4.7 são as *transições* entre esses estados, que são explicadas a seguir:

$\cdots \rightarrow$ **Nova:** Esta transição ocorre quando uma nova tarefa é admitida no sistema e começa a ser preparada para executar.

Nova $\rightarrow$ **Pronta:** ocorre quando a nova tarefa termina de ser carregada em memória, juntamente com suas bibliotecas e dados, estando pronta para executar.

Pronta $\rightarrow$ **Executando:** esta transição ocorre quando a tarefa é escolhida pelo escalonador para ser executada (ou para continuar sua execução), dentre as demais tarefas prontas.

Executando → Pronta: esta transição ocorre quando se esgota a fatia de tempo destinada à tarefa (ou seja, o fim do *quantum*); como nesse momento a tarefa não precisa de outros recursos além do processador, ela volta à fila de tarefas prontas até recebê-lo novamente.

Executando → Suspensa: caso a tarefa em execução solicite acesso a um recurso não disponível, como dados externos ou alguma sincronização, ela abandona o processador e fica suspensa até o recurso ficar disponível.

Suspensa → Pronta: quando o recurso solicitado pela tarefa se torna disponível, ela pode voltar a executar, portanto volta ao estado de pronta para aguardar o processador (que pode estar ocupado com outra tarefa).

Executando → Terminada: ocorre quando a tarefa encerra sua execução ou é abortada em consequência de algum erro (acesso inválido à memória, instrução ilegal, divisão por zero, etc.). Na maioria dos sistemas a tarefa que deseja encerrar avisa o sistema operacional através de uma chamada de sistema (no Linux é usada a chamada `exit`).

Terminada → ···: Uma tarefa terminada é removida da memória e seus registros e estruturas de controle no núcleo são liberados.

A estrutura do diagrama de ciclo de vida das tarefas pode variar de acordo com a interpretação dos autores. Por exemplo, a forma apresentada neste texto condiz com a apresentada em [Silberschatz et al., 2001] e outros autores. Por outro lado, o diagrama apresentado em [Tanenbaum, 2003] divide o estado "suspenso" em dois subestados separados: "bloqueado", quando a tarefa aguarda a ocorrência de algum evento (tempo, entrada/saída, etc.) e "suspenso", para tarefas bloqueadas que foram movidas da memória RAM para a área de troca pelo mecanismo de paginação em disco (vide Capítulo 17). Todavia, tal distinção de estados não faz mais sentido nos sistemas operacionais atuais baseados em memória paginada, pois neles os processos podem executar mesmo estando somente parcialmente carregados na memória.

Nos sistemas operacionais de mercado é possível consultar o estado das tarefas em execução no sistema. Isso pode ser feito no Windows, por exemplo, através do utilitário "Gerenciador de Tarefas". No Linux, diversos utilitários permitem gerar essa informação, como o comando `top`, cuja saída é mostrada no exemplo a seguir:

```
top - 16:58:06 up  8:26,  1 user,  load average: 6,04, 2,36, 1,08
Tarefas: 218 total,   7 executando, 211 dormindo,   0 parado,   0 zumbi
%Cpu(s): 49,7 us, 47,0 sy,  0,0 ni,  3,2 id,  0,0 wa,  0,0 hi,  0,1 si,  0,0 st
KiB Mem : 16095364 total,  9856576 free,  3134380 used,  3104408 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 11858380 avail Mem

  PID USUÁRIO   PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
32703 maziero   20   0 2132220 432628 139312 S  44,8  2,7   0:53.64 Web Content
 2192 maziero   20   0 9617080 686444 248996 S  29,8  4,3  20:01.81 firefox
11650 maziero   20   0 2003888 327036 129164 R  24,0  2,0   1:16.70 Web Content
 9844 maziero   20   0 2130164 442520 149508 R  17,9  2,7   1:29.18 Web Content
11884 maziero   20   0   25276   7692   3300 S  15,5  0,0   0:37.18 bash
20425 maziero   20   0   24808   7144   3212 S  14,4  0,0   0:08.39 bash
 1782 maziero   20   0 1788328 235200  77268 S   8,7  1,5  24:12.75 gnome-shell
...
```

Nesse exemplo, existem 218 tarefas no sistema, das quais 7 estão executando (tarefas cuja coluna de Status indica "R" – *running*) e as demais estão suspensas (com Status "S" – *sleeping*). O Linux, como a maioria dos sistemas operacionais, considera que uma tarefa está executando se estiver usando ou esperando por um processador.

Referências

F. Corbató. *The Compatible Time-Sharing System: A Programmer's Guide*. MIT Press, 1963.

R. Love. *Linux Kernel Development, Third Edition*. Addison-Wesley, 2010.

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

A. Tanenbaum. *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall, 2003.

Capítulo 5

Implementação de tarefas

Como visto no capítulo anterior, uma tarefa é a unidade básica de atividade dentro de um sistema operacional. Tarefas podem ser implementadas de várias formas, como processos, *threads*, transações e *jobs*. Neste capítulo são descritos os problemas relacionados à implementação do conceito de tarefa em um sistema operacional típico. São descritas as estruturas de dados necessárias para representar uma tarefa e as operações necessárias para que o processador possa comutar de uma tarefa para outra de forma transparente e eficiente.

5.1 Contextos

Na Seção 4.2 vimos que uma tarefa possui um estado interno bem definido, que representa sua situação atual: a posição de código que ela está executando, os valores de suas variáveis e os recursos que ela utiliza, por exemplo. Esse estado se modifica conforme a execução da tarefa evolui. O estado de uma tarefa em um determinado instante é denominado **contexto**. Uma parte importante do contexto de uma tarefa diz respeito ao estado interno do processador durante sua execução, como o valor do contador de programa (PC - *Program Counter*), do apontador de pilha (SP - *Stack Pointer*) e demais registradores. Além do estado interno do processador, o contexto de uma tarefa também inclui informações sobre os recursos usados por ela, como arquivos abertos, conexões de rede e semáforos.

A cada tarefa presente no sistema é associado um *descritor*, ou seja, uma estrutura de dados no núcleo que representa essa tarefa. Nessa estrutura de dados são armazenadas as informações relativas ao seu contexto e os demais dados necessários à sua gerência, como prioridades, estado, etc. Essa estrutura de dados é geralmente chamada de TCB (do inglês *Task Control Block*) ou PCB (*Process Control Block*). Um TCB tipicamente contém as seguintes informações:

- identificador da tarefa (pode ser um número inteiro, um apontador, uma referência de objeto ou um algum outro identificador);
- estado da tarefa (nova, pronta, executando, suspensa, terminada, etc.);
- informações de contexto do processador (valores contidos nos registradores);
- lista de áreas de memória usadas pela tarefa;

- listas de arquivos abertos, conexões de rede e outros recursos usados pela tarefa (exclusivos ou compartilhados com outras tarefas);
- informações de gerência e contabilização (prioridade, usuário proprietário, data de início, tempo de processamento já decorrido, volume de dados lidos/escritos, etc.).

Dentro do núcleo, os descritores das tarefas são geralmente organizados em listas ou vetores de TCBs. Por exemplo, normalmente há uma lista de tarefas prontas para executar, uma lista de tarefas aguardando acesso ao disco rígido, etc. Para ilustrar concretamente o conceito de TCB, o Apêndice A apresenta o TCB do núcleo Linux em sua versão 1.0.

5.2 Trocas de contexto

Para que o sistema operacional possa suspender e retomar a execução de tarefas de forma transparente (sem que as tarefas o percebam), é necessário definir operações para salvar o contexto atual de uma tarefa em seu TCB e restaurá-lo mais tarde no processador. Por essa razão, o ato de suspender uma tarefa e reativar outra é denominado uma **troca de contexto**.

A implementação da troca de contexto é uma operação delicada, envolvendo a manipulação de registradores e *flags* específicos de cada processador, sendo por essa razão geralmente codificada em linguagem de máquina. No Linux, as operações de troca de contexto para a plataforma Intel x86 estão definidas através de diretivas em Assembly no arquivo `arch/i386/kernel/process.c` dos fontes do núcleo.

Durante uma troca de contexto, existem questões de ordem mecânica e de ordem estratégica a serem resolvidas, o que traz à tona a separação entre mecanismos e políticas já discutida anteriormente (vide Seção 1.2). Por exemplo, o armazenamento e recuperação do contexto e a atualização das informações contidas no TCB de cada tarefa são aspectos mecânicos, providos por um conjunto de rotinas denominado **despachante** ou **executivo** (do inglês *dispatcher*). Por outro lado, a escolha da próxima tarefa a receber o processador a cada troca de contexto é estratégica, podendo sofrer influências de diversos fatores, como as prioridades, os tempos de vida e os tempos de processamento restante de cada tarefa. Essa decisão fica a cargo de um componente de código denominado **escalonador** (*scheduler*, vide Seção 6). Assim, o despachante implementa os mecanismos da gerência de tarefas, enquanto o escalonador implementa suas políticas.

A Figura 5.1 apresenta um diagrama temporal com os principais passos envolvidos em uma troca de contexto:

1. uma tarefa *A* está executando;
2. ocorre uma interrupção do temporizador do hardware e a execução desvia para a rotina de tratamento, no núcleo;
3. a rotina de tratamento ativa o despachante;
4. o despachante salva o estado da tarefa *A* em seu TCB e atualiza suas informações de gerência;

5. opcionalmente, o despachante consulta o escalonador para escolher a próxima tarefa a ativar (B);

6. o despachante resgata o estado da tarefa B de seu TCB e a reativa.

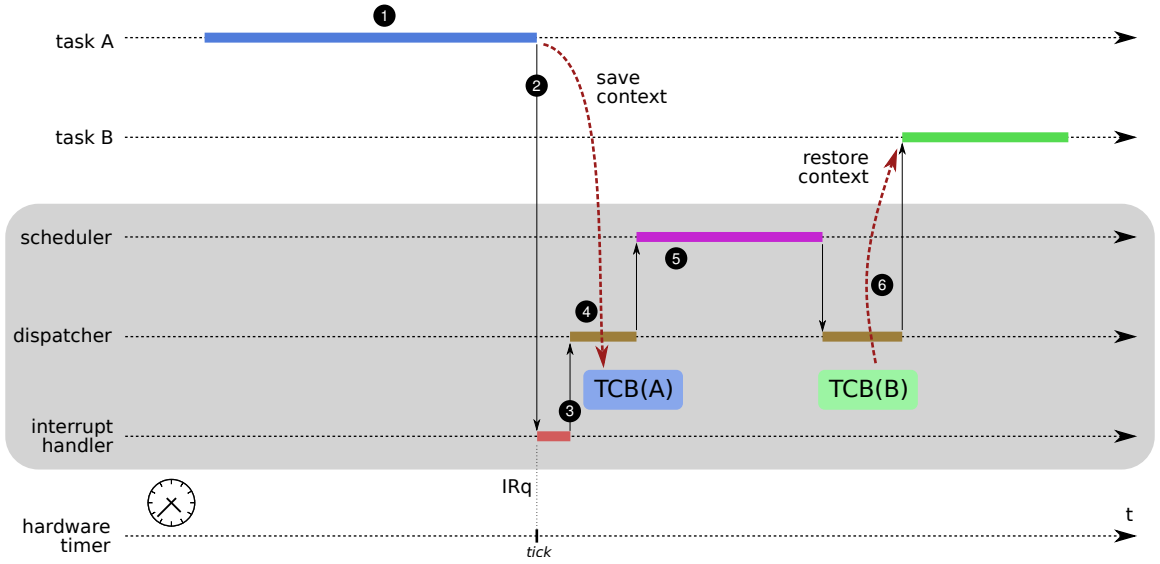


Figura 5.1: Passos de uma troca de contexto.

É importante observar que uma troca de contexto pode ser provocada pelo fim do *quantum* atual (através de uma interrupção de tempo), por um evento ocorrido em um periférico (uma interrupção do respectivo controlador), por uma chamada de sistema emitida pela tarefa corrente (uma interrupção de software) ou até mesmo por algum erro de execução que possa provocar uma exceção no processador.

A frequência de trocas de contexto tem impacto na eficiência do sistema operacional: quanto menor o número de trocas de contexto e menor a duração de cada troca, mais tempo sobrará para a execução das tarefas em si. Assim, é possível definir uma **medida de eficiência** $\mathcal{E}$ do uso do processador, em função das durações médias do *quantum* de tempo t_q e da troca de contexto t_{tc}:

$$\mathcal{E} = \frac{t_q}{t_q + t_{tc}}$$

Por exemplo, um sistema no qual as trocas de contexto duram $100\mu s$ e cujo *quantum* médio é de $10ms$ terá uma eficiência $\mathcal{E} = \frac{10ms}{10ms+100\mu s} = 99\%$. Caso a duração do quantum seja reduzida para $1ms$, a eficiência cairá para $\mathcal{E} = \frac{1ms}{1ms+100\mu s} = 91\%$. A eficiência final da gerência de tarefas é influenciada por vários fatores, como a carga do sistema (mais tarefas ativas implicam em mais tempo gasto pelo escalonador, aumentando t_{tc}) e o perfil das aplicações (aplicações que fazem muita entrada/saída saem do processador antes do final de seu *quantum*, diminuindo o valor médio de t_q).

Nos sistemas atuais, a realização de uma troca de contexto, envolvendo a interrupção da tarefa atual, o salvamento de seu contexto e a reativação da próxima tarefa, é uma operação relativamente rápida (alguns microssegundos, dependendo do hardware e do sistema operacional). A execução do escalonador, entretanto, pode ser bem mais demorada, sobretudo se houverem muitas tarefas prontas para executar. Por

esta razão, muitos sistemas operacionais não executam o escalonador a cada troca de contexto, mas apenas periodicamente, quando há necessidade de reordenar a fila de tarefas prontas. Nesse caso, o despachante sempre ativa a primeira tarefa dessa fila.

5.3 Processos

Há diversas formas de implementar o conceito de tarefa. Uma forma muito empregada pelos sistemas operacionais é o uso de **processos**, apresentado nesta seção.

5.3.1 O conceito de processo

Historicamente, um processo era definido como sendo uma tarefa com seus respectivos recursos, como arquivos abertos e canais de comunicação, em uma área de memória delimitada e isolada das demais. Ou seja, um processo seria uma espécie de "cápsula" isolada de execução, contendo uma tarefa e seus recursos. Essa visão é mantida por muitos autores, como [Silberschatz et al., 2001] e [Tanenbaum, 2003], que apresentam processos como equivalentes a tarefas.

De fato, os sistemas operacionais mais antigos, até meados dos anos 80, suportavam somente um fluxo de execução em cada processo. Assim, as unidades de execução (tarefa) e de recursos (processo) se confundiam. No entanto, quase todos os sistemas operacionais atuais suportam e existência de mais de uma tarefa em cada processo, como é o caso do Linux, Windows, iOS e os sistemas UNIX mais recentes.

Hoje em dia o processo deve ser visto como uma *unidade de contexto*, ou seja, um contêiner de recursos utilizados por uma ou mais tarefas para sua execução: áreas de memória (código, dados, pilha), informações de contexto e descritores de recursos do núcleo (arquivos abertos, conexões de rede, etc). Um processo pode então conter várias tarefas, que compartilham esses recursos. Os processos são isolados entre si pelos mecanismos de proteção providos pelo hardware (isolamento de áreas de memória, níveis de operação e chamadas de sistema), impedindo que uma tarefa do processo p_a acesse um recurso atribuído ao processo p_b. A Figura 5.2 ilustra o conceito de processo, visto como um contêiner de recursos.

Os sistemas operacionais atuais geralmente associam por *default* uma única tarefa a cada processo, o que corresponde à execução de um programa sequencial (iniciado pela função `main()` de um programa em C, por exemplo). Caso se deseje associar mais tarefas ao mesmo processo (para construir o navegador Internet da Figura 4.1, por exemplo), cabe ao desenvolvedor escrever o código necessário para solicitar ao núcleo a criação dessas tarefas adicionais, usualmente sob a forma de *threads* (apresentadas na Seção 5.4).

O núcleo do sistema operacional mantém descritores de processos, denominados PCBs (*Process Control Blocks*), para armazenar as informações referentes aos processos ativos. Um PCB contém informações como o identificador do processo (PID – *Process IDentifier*), seu usuário, prioridade, data de início, caminho do arquivo contendo o código executado pelo processo, áreas de memória em uso, arquivos abertos, etc. A listagem a seguir mostra um conjunto de processos no sistema Linux.

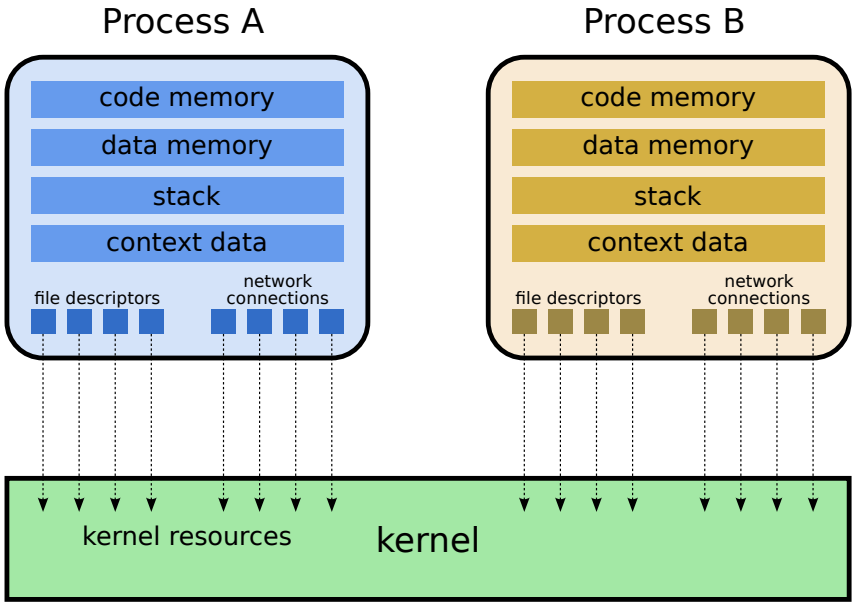


Figura 5.2: O processo visto como um contêiner de recursos.

```
top - 16:58:06 up  8:26,  1 user,  load average: 6,04, 2,36, 1,08
Tarefas: 218 total,   7 executando, 211 dormindo,   0 parado,   0 zumbi
%Cpu(s): 49,7 us, 47,0 sy,  0,0 ni,  3,2 id,  0,0 wa,  0,0 hi,  0,1 si,  0,0 st
KiB Mem : 16095364 total,  9856576 free,  3134380 used,  3104408 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 11858380 avail Mem

  PID USUÁRIO   PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
32703 maziero   20   0 2132220 432628 139312 S  44,8  2,7   0:53.64 Web Content
 2192 maziero   20   0 9617080 686444 248996 S  29,8  4,3  20:01.81 firefox
11650 maziero   20   0 2003888 327036 129164 R  24,0  2,0   1:16.70 Web Content
 9844 maziero   20   0 2130164 442520 149508 R  17,9  2,7   1:29.18 Web Content
11884 maziero   20   0   25276   7692   3300 S  15,5  0,0   0:37.18 bash
20425 maziero   20   0   24808   7144   3212 S  14,4  0,0   0:08.39 bash
 1782 maziero   20   0 1788328 235200  77268 S   8,7  1,5  24:12.75 gnome-shell
...
```

Associando-se tarefas a processos, o descritor (TCB) de cada tarefa pode ser bastante simplificado: para cada tarefa, basta armazenar seu identificador, os registradores do processador e uma referência ao processo onde a tarefa executa. Observa-se também que a troca de contexto entre duas tarefas dentro do mesmo processo é muito mais simples e rápida que entre tarefas em processos distintos, pois somente os registradores do processador precisam ser salvos/restaurados (pois as áreas de memória e demais recursos são comuns a ambas). Essas questões são aprofundadas na Seção 5.4.

5.3.2 Gestão de processos

Durante a vida do sistema, processos são criados e destruídos. Essas operações são disponibilizadas às aplicações através de chamadas de sistema; cada sistema operacional tem suas próprias chamadas para a gestão de processos. O quadro 5.1 traz exemplos de algumas chamadas de sistema usadas na gestão de processos.

Ação	Windows	Linux
Criar um novo processo	`CreateProcess()`	`fork(), execve()`
Encerrar o processo corrente	`ExitProcess()`	`exit()`
Encerrar outro processo	`TerminateProcess()`	`kill()`
Obter o ID do processo corrente	`GetCurrentProcessId()`	`getpid()`

Tabela 5.1: Chamadas de sistema para a gestão de processos.

No caso dos sistemas UNIX, a criação de novos processos é feita em duas etapas: na primeira etapa, um processo cria uma réplica de si mesmo, usando a chamada de sistema `fork()`. Todo o espaço de memória do processo inicial (pai) é copiado para o novo processo (filho), incluindo o código da(s) tarefa(s) associada(s) e os descritores dos arquivos e demais recursos associados ao mesmo. A Figura 5.3 ilustra o funcionamento dessa chamada.

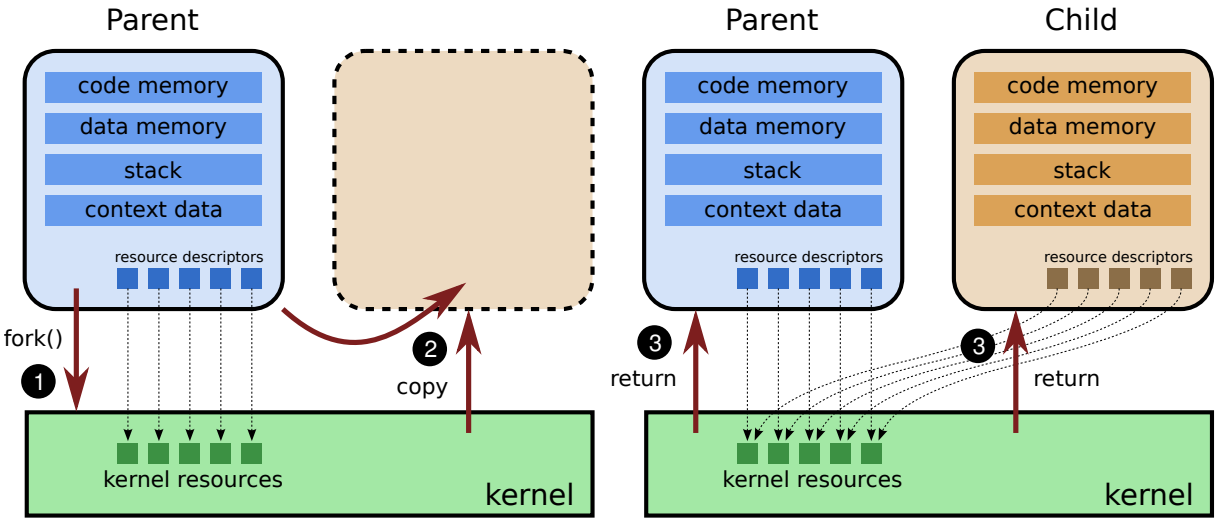


Figura 5.3: Execução da chamada de sistema `fork()`.

A chamada de sistema `fork()` é invocada por um processo, mas dois processos recebem seu retorno: o *processo pai*, que a invocou, e o *processo filho*, que acabou de ser criado e que possui o mesmo estado interno que o pai (ele também está aguardando o retorno da chamada de sistema). Ambos os processos acessam os mesmos recursos do núcleo, embora executem em áreas de memória distintas.

Na segunda etapa, o processo filho usa a chamada de sistema `execve()` para carregar um novo código binário em sua memória. Essa chamada substitui o código do processo que a invoca pelo código executável contido em um arquivo informado como parâmetro. A listagem a seguir apresenta um exemplo de uso conjunto dessas duas chamadas de sistema:

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[], char *envp[])
{
   int pid ;                        // identificador de processo

   pid = fork () ;                  // replicação do processo

   if ( pid < 0 )                   // fork funcionou?
   {
      perror ("Erro: ") ;           // não, encerra este processo
      exit (-1) ;
   }
   else                             // sim, fork funcionou
      if ( pid > 0 )                // sou o processo pai?
         wait (0) ;                 // sim, vou esperar meu filho concluir
      else                          // não, sou o processo filho
      {
         // carrega outro código binário para executar
         execve ("/bin/date", argv, envp) ;
         perror ("Erro: ") ;        // execve não funcionou
      }
   printf ("Tchau !\n") ;
   exit(0) ;                        // encerra este processo
}
```

A chamada de sistema `exit()` usada no exemplo acima serve para informar ao núcleo do sistema operacional que o processo em questão não é mais necessário e pode ser destruído, liberando todos os recursos a ele associados (arquivos abertos, conexões de rede, áreas de memória, etc.). Processos podem solicitar ao núcleo o encerramento de outros processos, mas essa operação só é aplicável a processos do mesmo usuário, ou se o processo solicitante pertencer ao administrador do sistema.

Outro aspecto importante a ser considerado em relação a processos diz respeito à comunicação. Tarefas associadas ao mesmo processo podem trocar informações facilmente, pois compartilham as mesmas áreas de memória. Todavia, isso não é possível entre tarefas associadas a processos distintos. Para resolver esse problema, o núcleo deve prover às aplicações chamadas de sistema que permitam a comunicação interprocessos (IPC – *Inter-Process Communication*). Esse tema será estudado em profundidade no Capítulo 8.

5.3.3 Hierarquia de processos

Na operação de criação de processos do UNIX aparece de maneira bastante clara a noção de **hierarquia** entre processos: processos são filhos de outros processos. À medida em que processos são criados, forma-se uma *árvore de processos* no sistema, que pode ser usada para gerenciar de forma coletiva os processos ligados à mesma aplicação ou à mesma sessão de trabalho de um usuário, pois irão constituir uma sub-árvore de processos. Por exemplo, quando um processo encerra, seus filhos são informados sobre isso, e podem decidir se também encerram ou se continuam a executar.

Por outro, nos sistemas Windows, todos os processos têm o mesmo nível hierárquico, não havendo distinção entre pais e filhos. O comando `pstree` do sistema Linux permite visualizar a árvore de processos do sistema, como mostra a listagem de exemplo a seguir[1].

```
init-+-cron
     |-dhcpcd
     |-getty
     |-getty
     |-ifplugd
     |-ifplugd
     |-lighttpd---php-cgi-+-php-cgi
     |                    `-php-cgi
     |-logsave
     |-logsave
     |-ntpd
     |-openvpn
     |-p910nd
     |-rsyslogd-+-{rsyslogd}
     |          `-{rsyslogd}
     |-sshd---sshd---sshd---pstree
     |-thd
     `-udevd-+-udevd
             `-udevd
```

5.4 Threads

Conforme visto na Seção 5.3, os primeiros sistemas operacionais suportavam uma única tarefa por processo. À medida em que as aplicações se tornavam mais complexas, essa limitação se tornou bem inconveniente. Por exemplo, um editor de textos geralmente executa tarefas simultâneas de edição, formatação, paginação e verificação ortográfica sobre os mesmos dados (o texto em edição). Da mesma forma, processos que implementam servidores de rede (de arquivos, bancos de dados, etc.) devem gerenciar as conexões de vários usuários simultaneamente, que muitas vezes requisitam as mesmas informações. Essas demandas evidenciaram a necessidade de suportar mais de uma tarefa operando sobre os mesmos recursos, ou seja, dentro do mesmo processo.

5.4.1 Definição de *thread*

Uma *thread* é definida como sendo um fluxo de execução independente. Um processo pode conter uma mais *threads*, cada uma executando seu próprio código e compartilhando recursos com as demais *threads* localizadas no mesmo processo[2]. Cada *thread* é caracterizada por um código em execução e um pequeno contexto local, o chamado *Thread Local Storage* (TLS), composto pelos registradores do processador e uma área de pilha em memória, para que a *thread* possa armazenar variáveis locais efetuar chamadas de funções.

[1] Listagem obtida de um sistema *Raspberry Pi* com SO Linux *Raspbian* 8.

[2] Alguns autores usam também o termo *processo leve* (*lightweigth process*) como sinônimo de *thread*.

Threads são também utilizadas para implementar fluxos de execução dentro do núcleo do SO, neste caso recebendo o nome de *threads de núcleo* (em oposição às *threads* dos processos, denominadas *user threads*). Além de representar as *threads* de usuário dentro do núcleo, as *threads* de núcleo também incluem atividades internas do núcleo, como rotinas de *drivers* de dispositivos ou tarefas de gerência. A Figura 5.4 ilustra o conceito de *threads* de usuário e de núcleo. Na figura, o processo A tem várias *threads*, enquanto o processo B é sequencial (tem uma única *thread*).

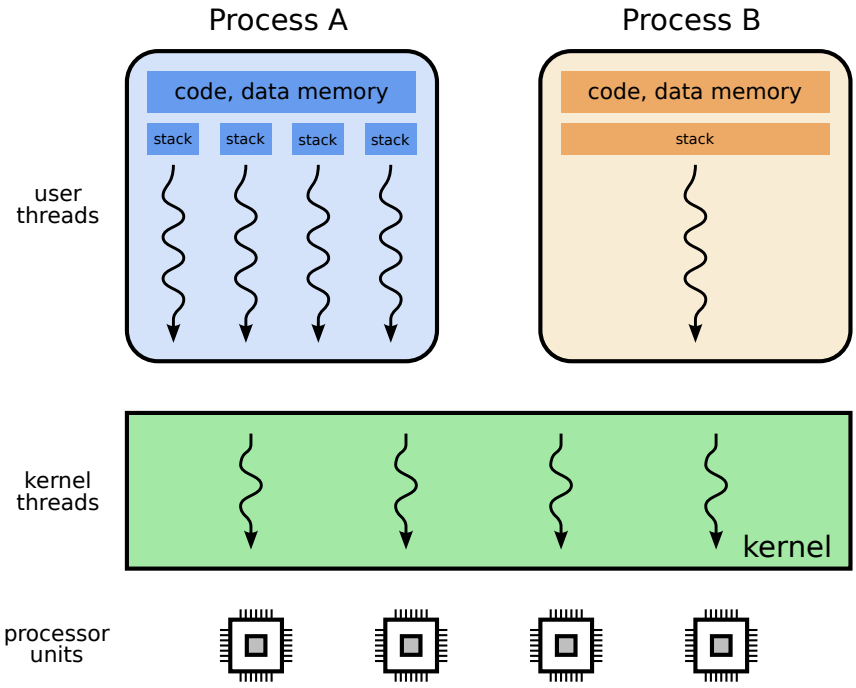


Figura 5.4: *Threads* de usuário e de núcleo.

5.4.2 Modelos de *threads*

As *threads* contidas nos processos, definidas no espaço de usuário, devem ser gerenciadas pelo núcleo do sistema operacional. Essa gerência pode ser feita de diversas formas, conforme os modelos de implementação de *threads* apresentados nesta seção.

O modelo N:1

Os sistemas operacionais mais antigos suportavam apenas processos sequenciais, com um único fluxo de execução em cada um. Os desenvolvedores de aplicações contornaram esse problema construindo bibliotecas para salvar, modificar e restaurar os registradores da CPU dentro do processo, permitindo assim criar e gerenciar vários fluxos de execução (*threads*) dentro de cada processo, sem a participação do núcleo.

Com essas bibliotecas, uma aplicação pode lançar várias *threads* conforme sua necessidade, mas o núcleo do sistema irá sempre perceber (e gerenciar) apenas um fluxo de execução dentro de cada processo (ou seja, o núcleo irá manter apenas uma *thread* de núcleo por processo). Esta forma de implementação de *threads* é denominada **Modelo de *Threads* N:1**, pois N *threads* dentro de um processo são mapeadas em uma única *thread* no núcleo. Esse modelo também é denominado *fibers* ou ainda *green threads*. A Figura 5.5 ilustra o modelo N:1.

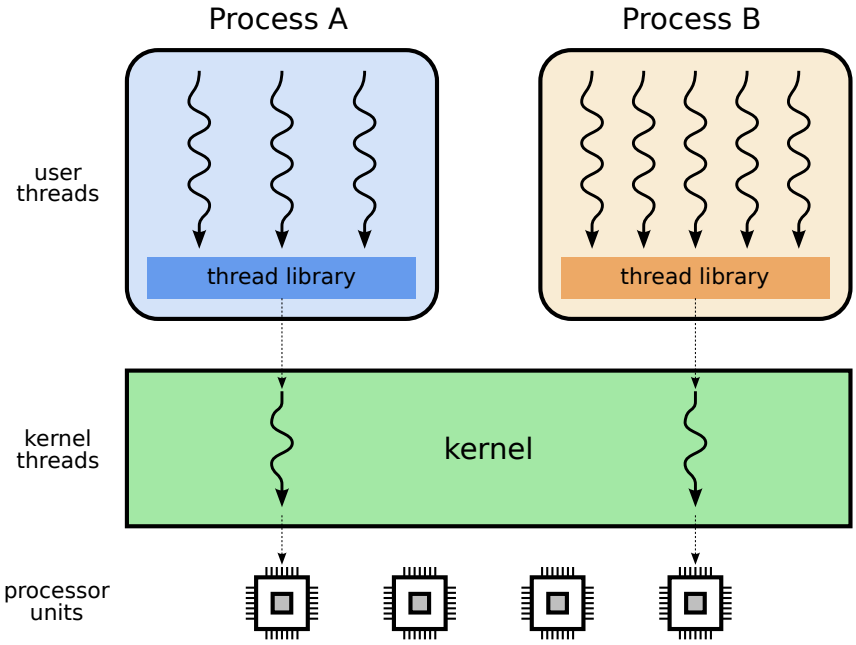


Figura 5.5: O modelo de *threads* N:1.

O modelo de *threads* N:1 é muito utilizado, por ser leve e de fácil implementação. Como o núcleo somente vê uma *thread*, a carga de gerência imposta ao núcleo é pequena e não depende do número de *threads* dentro da aplicação. Essa característica torna este modelo útil na construção de aplicações que exijam muitos *threads*, como jogos ou simulações de grandes sistemas[3]. Exemplos de implementação desse modelo são as bibliotecas *GNU Portable Threads* [Engeschall, 2005], *User-Mode Scheduling* (UMS, Microsoft) e *Green Threads* (Java).

Por outro lado, o modelo de *threads* N:1 apresenta alguns problemas:

- as operações de entrada/saída são realizadas pelo núcleo; se uma *thread* de usuário solicitar uma leitura do teclado, por exemplo, a *thread* de núcleo correspondente será suspensa até a conclusão da operação, bloqueando todas as *threads* daquele processo;
- o núcleo do sistema divide o tempo de processamento entre as *threads* de núcleo. Assim, um processo com 100 *threads* irá receber o mesmo tempo de processador que outro com apenas uma *thread*, ou seja, cada *thread* do primeiro processo irá receber 1/100 do tempo que recebe a *thread* única do outro processo;
- *threads* do mesmo processo não podem executar em paralelo, mesmo se o computador dispuser de vários processadores ou *cores*, porque o núcleo somente escalona as *threads* de núcleo nos processadores.

O modelo 1:1

A necessidade de suportar aplicações *multithread* levou os desenvolvedores de sistemas operacionais a incorporar a gerência dos *threads* dos processo no núcleo

[3] A simulação detalhada do tráfego viário de uma cidade grande, por exemplo, pode exigir uma *thread* para cada veículo, podendo portanto chegar a milhões de *threads*.

do sistema. Para cada *thread* de usuário foi então associado um *thread* correspondente dentro do núcleo, suprimindo com isso a necessidade de bibliotecas de *threads*. Essa forma de implementação é denominada **Modelo de *Threads* 1:1**, sendo apresentada na Figura 5.6. Este é o modelo mais frequente nos sistemas operacionais atuais, como o Windows NT e seus descendentes, além da maioria dos UNIXes.

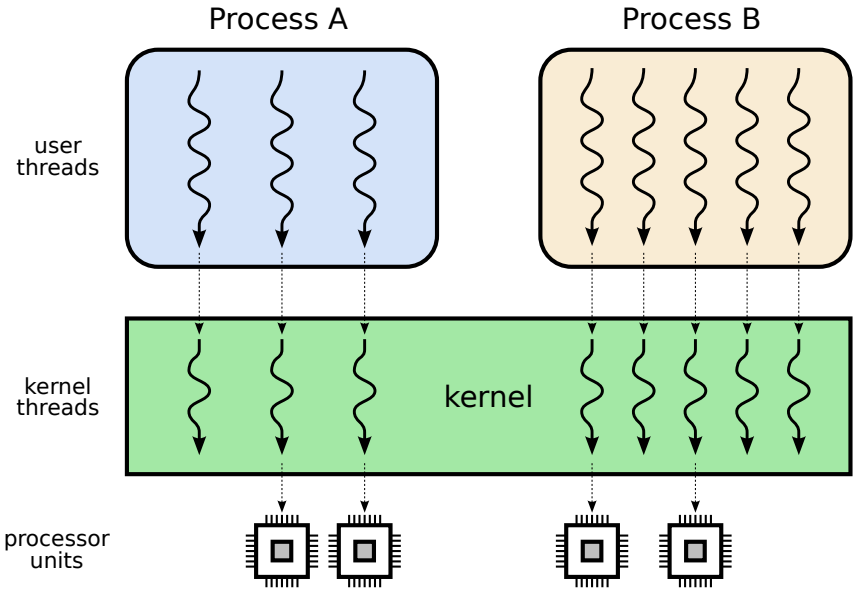


Figura 5.6: O modelo de *threads* 1:1.

O modelo 1:1 resolve vários problemas: caso uma *thread* de usuário solicite uma operação bloqueante, somente sua *thread* de núcleo correspondente será suspensa, sem afetar as demais *threads* do processo. Além disso, a distribuição de processamento entre as *threads* é mais justa e, caso o hardware tenha mais de um processador, mais *threads* do mesmo processo podem executar ao mesmo tempo, o que não era possível no modelo N:1.

O modelo de *threads* 1:1 é adequado para a maioria das situações usuais e atende bem às necessidades das aplicações interativas e servidores de rede. No entanto, é pouco escalável: a criação de um número muito grande de *threads* impõe uma carga elevada ao núcleo do sistema, inviabilizando aplicações com muitas tarefas (como grandes servidores Web e simulações de grande porte).

O modelo N:M

Para resolver o problema de escalabilidade da abordagem 1:1, alguns sistemas operacionais implementam um modelo híbrido, que agrega características dos modelos anteriores. Nesse novo modelo, uma biblioteca gerencia um conjunto de N *threads* de usuário (dentro do processo), que é mapeado em $M < N$ *threads* no núcleo. Essa abordagem, denominada **Modelo de *Threads* N:M**, é apresentada na Figura 5.7.

O conjunto de *threads* de núcleo associadas a um processo, denominado *thread pool*, geralmente contém uma *thread* para cada *thread* de usuário bloqueada, mais uma *thread* para cada processador disponível; esse conjunto pode ser ajustado dinamicamente, conforme a necessidade da aplicação.

O modelo N:M é implementado pelo Solaris e também pelo projeto KSE (*Kernel-Scheduled Entities*) do FreeBSD [Evans and Elischer, 2003] baseado nas ideias apresentadas

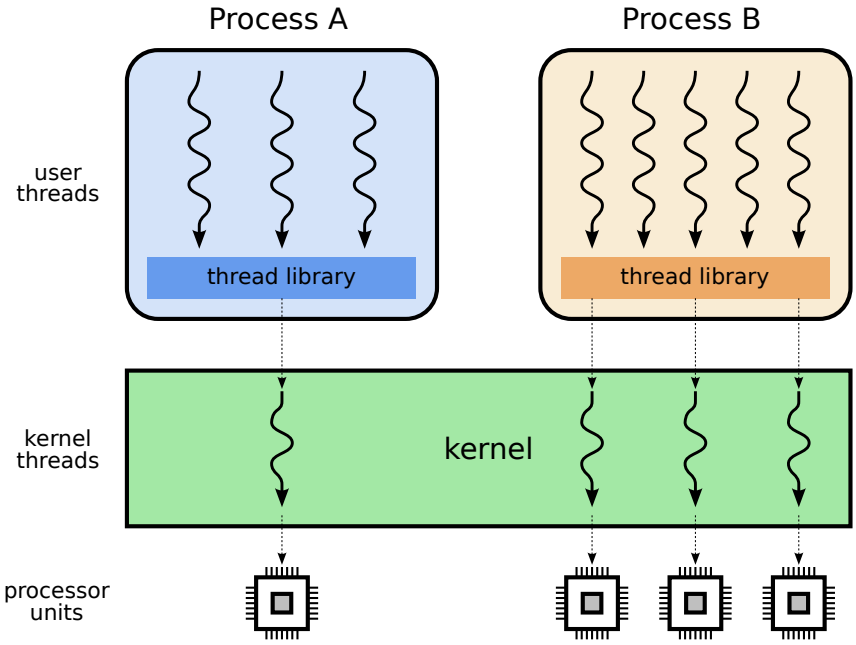


Figura 5.7: O modelo de *threads* N:M.

em [Anderson et al., 1992]. Ele alia as vantagens de maior interatividade do modelo 1:1 à maior escalabilidade do modelo N:1. Como desvantagens desta abordagem podem ser citadas sua complexidade de implementação e maior custo de gerência dos *threads* de núcleo, quando comparados ao modelo 1:1.

Comparação entre os modelos

A Tabela 5.2 resume os principais aspectos dos três modelos de implementação de *threads* e faz um comparativo entre eles.

5.4.3 Programando com *threads*

No passado, cada sistema operacional definia sua própria interface para a criação de *threads*, o que levou a problemas de portabilidade das aplicações. Em 1995 foi definido o padrão *IEEE POSIX 1003.1c*, mais conhecido como *POSIX Threads* ou simplesmente *PThreads* [Nichols et al., 1996], que define uma interface padronizada para o uso de *threads* na linguagem C. Esse padrão é amplamente difundido e suportado hoje em dia. A listagem a seguir, extraída de [Barney, 2005], exemplifica o uso do padrão *PThreads*:

Modelo	**N:1**	**1:1**	**N:M**
Resumo	N *threads* do processo mapeados em uma *thread* de núcleo	Cada *thread* do processo mapeado em uma *thread* de núcleo	N *threads* do processo mapeados em M< N *threads* de núcleo
Implementação	no processo (biblioteca)	no núcleo	em ambos
Complexidade	baixa	média	alta
Custo de gerência	baixo	médio	alto
Escalabilidade	alta	baixa	alta
Paralelismo entre *threads* do mesmo processo	não	sim	sim
Troca de contexto entre *threads* do mesmo processo	rápida	lenta	rápida
Divisão de recursos entre tarefas	injusta	justa	variável, pois o mapeamento *thread*→ processador é dinâmico
Exemplos	GNU Portable Threads, Microsoft UMS	Windows, Linux	Solaris, FreeBSD KSE

Tabela 5.2: Comparativo dos modelos de *threads*.

```c
// Exemplo de uso de threads Posix em C no Linux
// Compilar com gcc exemplo.c -o exemplo -lpthread

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define NUM_THREADS 5

// cada thread vai executar esta função
void *print_hello (void *threadid)
{
   printf ("%ld: Hello World!\n", (long) threadid);
   sleep (5) ;
   printf ("%ld: Bye bye World!\n", (long) threadid);
   pthread_exit (NULL);     // encerra esta thread
}
```

```
19 // thread "main" (vai criar as demais threads)
20 int main (int argc, char *argv[])
21 {
22    pthread_t thread[NUM_THREADS];
23    long status, i;
24
25    // cria as demais threads
26    for(i = 0; i < NUM_THREADS; i++)
27    {
28       printf ("Creating thread %ld\n", i);
29       status = pthread_create (&thread[i], NULL, print_hello, (void *) i);
30
31       if (status) // ocorreu um erro
32       {
33          perror ("pthread_create");
34          exit (-1);
35       }
36    }
37
38    // encerra a thread "main"
39    pthread_exit (NULL);
40 }
```

Nessa listagem pode-se perceber que o programa inicia sua execução com apenas uma *thread* (representada pela função `main()`, linha 21); esta, por sua vez, cria as demais *threads* (linha 30) e indica o código a ser executado pelas mesmas (no exemplo, todas executam a mesma função `print_hello()`, linha 12).

Threads podem ser utilizadas em diversas outras linguagens de programação, como Java, Python, Perl, etc. O código a seguir traz um exemplo simples de criação de *threads* em Java (extraído da documentação oficial da linguagem):

```
1  // save as MyThread.java; javac MyThread.java; java MyThread
2
3  public class MyThread extends Thread {
4     int threadID;
5
6     private static final int NUM_THREADS = 5 ;
7
8     MyThread (int ID) {
9        threadID = ID;
10    }
11
12    // corpo de cada thread
13    public void run () {
14       System.out.println (threadID + ": Hello World!") ;
15       try {
16         Thread.sleep(5000);
17       }
18       catch (InterruptedException e) {
19         e.printStackTrace();
20       }
21       System.out.println (threadID + ": Bye bye World!") ;
22    }
```

```
23    public static void main (String args[]) {
24
25       MyThread[] t = new MyThread[NUM_THREADS] ;
26
27       // cria as threads
28       for (int i = 0; i < NUM_THREADS; i++) {
29          t[i] = new MyThread (i);
30       }
31
32       // inicia a execução das threads
33       for (int i = 0; i < NUM_THREADS; i++) {
34          t[i].start () ;
35       }
36    }
37 }
```

5.5 Uso de processos *versus threads*

Neste capítulo foram apresentadas duas formas de implementar tarefas em um sistema operacional: os processos e as *threads*. No caso do desenvolvimento de sistemas complexos, compostos por várias tarefas interdependentes executando em paralelo, qual dessas formas de implementação deve ser escolhida? Um navegador de Internet (*browser*), um servidor Web ou um servidor de bancos de dados atendendo vários usuários simultâneos são exemplos desse tipo de sistema.

A implementação de sistemas usando **um processo para cada tarefa** é uma possibilidade. Essa abordagem tem como grande vantagem a robustez, pois um erro ocorrido em um processo ficará restrito ao seu espaço de memória pelos mecanismos de isolamento do hardware. Além disso, os processos podem ser configurados para executar com usuários (e permissões) distintas, aumentando a segurança do sistema. Por outro lado, nessa abordagem o compartilhamento de dados entre os diversos processos que compõem a aplicação irá necessitar de mecanismos especiais, como áreas de memória compartilhada, pois um processo não pode acessar as áreas de memória dos demais processos. Como exemplos de sistemas que usam a abordagem baseada em processos, podem ser citados o servidor web Apache (nas versões 1.*) e o servidor de bancos de dados PostgreSQL.

Também é possível implementar todas as tarefas em um único processo, usando **uma *thread* para cada tarefa**. Neste caso, todas as *threads* compartilham o mesmo espaço de endereçamento e os mesmos recursos (arquivos abertos, conexões de rede, etc), tornando mais simples implementar a interação entre as tarefas. Além disso, a execução é mais ágil, pois é mais rápido criar uma *thread* que um processo. Todavia, um erro em uma *thread* pode se alastrar às demais, pondo em risco a robustez da aplicação inteira. Muitos sistemas usam uma abordagem multi-*thread*, como o servidor Web IIS (Microsoft *Internet Information Server*) e o editor de textos LibreOffice.

Sistemas mais recentes e sofisticados usam uma abordagem híbrida, com o **uso conjunto de processos e *threads***. A aplicação é composta por vários processos, cada um contendo diversas *threads*. Busca-se com isso aliar a robustez proporcionada pelo isolamento entre processos e o desempenho, menor consumo de recursos e facilidade de programação proporcionados pelas *threads*. Esse modelo híbrido é usado, por exemplo, nos navegadores Chrome e Firefox: cada aba de navegação é atribuída a um novo

processo, que cria as *threads* necessárias para buscar e renderizar aquele conteúdo e interagir com o usuário. O servidor Web Apache 2.* e o servidor de bases de dados Oracle também usam essa abordagem.

A Figura 5.8 ilustra as três abordagens apresentadas nesta seção; a Tabela 5.3 sintetiza a comparação entre elas.

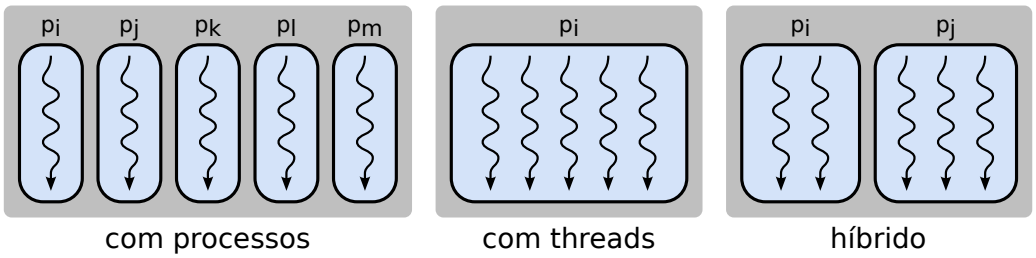


Figura 5.8: Implementação de sistemas usando processos, *threads* ou híbrido.

Característica	**Com processos**	**Com *threads* (1:1)**	**Híbrido**
Custo de criação de tarefas	alto	baixo	médio
Troca de contexto	lenta	rápida	variável
Uso de memória	alto	baixo	médio
Compartilhamento de dados entre tarefas	canais de comunicação e áreas de memoria compartilhada.	variáveis globais e dinâmicas.	ambos.
Robustez	alta, um erro fica contido no processo.	baixa, um erro pode afetar todas as *threads*.	média, um erro pode afetar as *threads* no mesmo processo.
Segurança	alta, cada processo pode executar com usuários e permissões distintas.	baixa, todas as *threads* herdam as permissões do processo onde executam.	alta, *threads* que necessitam as mesmas permissões podem ser agrupadas em um mesmo processo.
Exemplos	Servidor Apache (versões 1.*), SGBD PostGres	Servidor Apache (versões 2.*), SGBD MySQL	Navegadores Chrome e Firefox, SGBD Oracle

Tabela 5.3: Comparativo entre uso de processos e de *threads*.

Referências

T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

B. Barney. POSIX threads programming. `http://www.llnl.gov/computing/tutorials/pthreads`, 2005.

R. Engeschall. The GNU Portable Threads. `http://www.gnu.org/software/pth`, 2005.

J. Evans and J. Elischer. Kernel-scheduled entities for FreeBSD. http://www.aims.net.au/chris/kse, 2003.

B. Nichols, D. Buttlar, and J. Farrell. *PThreads Programming*. O'Reilly Media, Inc, 1996.

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

A. Tanenbaum. *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall, 2003.

Capítulo 6

Escalonamento de tarefas

Um dos componentes mais importantes da gerência de tarefas é o **escalonador de tarefas** (*task scheduler*), que decide a ordem de execução das tarefas prontas. O algoritmo utilizado no escalonador define o comportamento do sistema operacional, permitindo obter sistemas que tratem de forma mais eficiente e rápida as tarefas a executar, que podem ter características diversas: aplicações interativas, processamento de grandes volumes de dados, programas de cálculo numérico, etc.

6.1 Tipos de tarefas

Antes de se definir o algoritmo usado por um escalonador de tarefas, é necessário ter em mente a natureza das tarefas que o sistema irá executar. Existem vários critérios que definem o comportamento de uma tarefa; uma primeira classificação possível diz respeito ao seu comportamento temporal:

Tarefas de tempo real: exigem previsibilidade em seus tempos de resposta aos eventos externos, pois geralmente estão associadas ao controle de sistemas críticos, como processos industriais, tratamento de fluxos multimídia, etc. O escalonamento de tarefas de tempo real é um problema complexo, fora do escopo deste livro e discutido mais profundamente em [Burns and Wellings, 1997; Farines et al., 2000].

Tarefas interativas: são tarefas que recebem eventos externos (do usuário ou através da rede) e devem respondê-los rapidamente, embora sem os requisitos de previsibilidade das tarefas de tempo real. Esta classe de tarefas inclui a maior parte das aplicações dos sistemas *desktop* (editores de texto, navegadores Internet, jogos) e dos servidores de rede (e-mail, web, bancos de dados).

Tarefas em lote (*batch*): são tarefas sem requisitos temporais explícitos, que normalmente executam sem intervenção do usuário, como procedimentos de *backup*, varreduras de antivírus, cálculos numéricos longos, tratamentos de grandes massas de dados em lote, renderização de animações, etc.

Além dessa classificação, as tarefas também podem ser classificadas de acordo com seu comportamento no uso do processador:

Tarefas orientadas a processamento (*CPU-bound tasks*): são tarefas que usam intensivamente o processador na maior parte de sua existência. Essas tarefas passam a maior parte do tempo nos estados *pronta* ou *executando*. A conversão de arquivos de vídeo e outros processamentos longos (como os feitos pelo projeto *SETI@home* [Anderson et al., 2002]) são bons exemplos desta classe de tarefas.

Tarefas orientadas a entrada/saída (*IO-bound tasks*): são tarefas que dependem muito mais dos dispositivos de entrada/saída que do processador. Essas tarefas ficam boa parte de suas existências no estado *suspenso*, aguardando respostas às suas solicitações de leitura e/ou escrita de dados nos dispositivos de entrada/saída. Exemplos desta classe de tarefas incluem editores, compiladores e servidores de rede.

É importante observar que uma tarefa pode mudar de comportamento ao longo de sua execução. Por exemplo, um conversor de arquivos de áudio WAV→MP3 alterna constantemente entre fases de processamento e de entrada/saída, até concluir a conversão dos arquivos desejados.

6.2 Objetivos e métricas

Ao se definir um algoritmo de escalonamento, deve-se ter em mente seu objetivo. Todavia, os objetivos do escalonador são muitas vezes contraditórios; o desenvolvedor do sistema tem de escolher o que priorizar, em função do perfil das aplicações a suportar. Por exemplo, um sistema interativo voltado à execução de jogos exige valores de quantum baixos, para que cada tarefa pronta receba rapidamente o processador (provendo maior interatividade). Todavia, valores pequenos de quantum implicam em uma menor eficiência $\mathcal{E}$ no uso do processador, conforme visto na Seção 5.2. Vários critérios podem ser definidos para a avaliação de escalonadores; os mais frequentemente utilizados são:

Tempo de execução (ou de vida) (*turnaround time*, t_t): diz respeito ao tempo total da execução de uma tarefa, ou seja, o tempo decorrido entre a criação da tarefa e seu encerramento, computando todos os tempos de processamento e de espera. É uma medida típica de sistemas em lote, nos quais não há interação direta com os usuários do sistema. Não deve ser confundido com o **tempo de processamento** (t_p), que é o tempo total de uso de processador demandado pela tarefa.

Tempo de espera (*waiting time*, t_w): é o tempo total perdido pela tarefa na fila de tarefas prontas, aguardando o processador. Deve-se observar que esse tempo não inclui os tempos de espera em operações de entrada/saída (que são inerentes à aplicação e aos dispositivos).

Tempo de resposta (*response time*, t_r): é o tempo decorrido entre a chegada de um evento ao sistema e o resultado imediato de seu processamento. Por exemplo, em um editor de textos seria o tempo decorrido entre apertar uma tecla e o caractere correspondente aparecer na tela. Essa medida de desempenho é típica de sistemas interativos, como sistemas *desktop* e de tempo real; ela depende sobretudo da rapidez no tratamento das interrupções de hardware pelo núcleo e do valor do *quantum* de tempo, para permitir que as tarefas interativas cheguem mais rápido ao processador quando saem do estado suspenso.

Justiça: este critério diz respeito à distribuição do processador entre as tarefas prontas: duas tarefas de comportamento e prioridade similares devem ter durações de execução similares.

Eficiência: a eficiência $\mathcal{E}$, conforme definido na Seção 5.2, indica o grau de utilização do processador na execução das tarefas do usuário. Ela depende sobretudo da rapidez da troca de contexto e da quantidade de tarefas orientadas a entrada/saída no sistema (tarefas desse tipo geralmente abandonam o processador antes do fim do *quantum*, gerando assim mais trocas de contexto que as tarefas orientadas a processamento).

6.3 Escalonamento preemptivo e cooperativo

O escalonador de um sistema operacional pode ser preemptivo ou cooperativo (não-cooperativo):

Sistemas preemptivos: nestes sistemas uma tarefa pode perder o processador caso termine seu *quantum* de tempo, caso execute uma chamada de sistema ou caso ocorra uma interrupção que acorde uma tarefa mais prioritária (que estava suspensa aguardando um evento). A cada interrupção, exceção ou chamada de sistema, o escalonador reavalia todas as tarefas da fila de prontas e decide se mantém ou substitui a tarefa atualmente em execução.

Sistemas cooperativos: a tarefa em execução permanece no processador tanto quanto possível, só liberando o mesmo caso termine de executar, solicite uma operação de entrada/saída ou libere explicitamente o processador[1], voltando à fila de tarefas prontas. Esses sistemas são chamados de *cooperativos* por exigir a cooperação das tarefas entre si na gestão do processador, para que todas possam executar.

Atualmente a maioria dos sistemas operacionais de uso geral atuais é preemptiva. Sistemas mais antigos, como o Windows 3.*, PalmOS 3 e MacOS 8 e 9 operavam de forma cooperativa.

Em um sistema preemptivo simples, normalmente as tarefas só são interrompidas quando o processador está no modo usuário; a *thread* de núcleo correspondente a cada tarefa não sofre interrupções. Entretanto, os sistemas mais sofisticados implementam a preempção de tarefas também no modo núcleo. Essa funcionalidade é importante para sistemas de tempo real, pois permite que uma tarefa de alta prioridade chegue mais rapidamente ao processador quando for reativada. Núcleos de sistema que oferecem essa possibilidade são denominados **núcleos preemptivos**; Solaris, Linux 2.6 e Windows NT são exemplos de núcleos preemptivos.

6.4 Algoritmos de escalonamento de tarefas

Esta seção descreve os algoritmos mais simples de escalonamento de tarefas encontrados na literatura. Esses algoritmos raramente são usados *ipsis litteris*, mas

[1] Uma tarefa pode liberar explicitamente o processador e voltar à fila de prontas através de uma chamada de sistema específica, como `sched_yield()` em Linux ou `SwitchToThread()` em Windows.

servem de base conceitual para a construção dos escalonadores mais complexos que são usados em sistemas operacionais reais. Para a descrição do funcionamento de cada algoritmo será considerado um sistema monoprocessado e um conjunto hipotético de 4 tarefas ($t_1 \cdots t_5$) na fila de prontas do sistema operacional, descritas na Tabela 6.1 a seguir.

Tarefa	t_1	t_2	t_3	t_4	t_5
Ingresso	0	0	1	3	5
Duração	5	2	4	1	2
Prioridade	2	3	1	4	5

Tabela 6.1: Tarefas na fila de prontas.

Para simplificar a análise dos algoritmos, as tarefas $t_1 \cdots t_5$ são orientadas a processamento, ou seja, não param para realizar operações de entrada/saída. Cada tarefa tem uma data de ingresso (instante em que entrou no sistema), uma duração (tempo de processamento que necessita para realizar sua execução) e uma prioridade (usada nos algoritmos PRIOc e PRIOp, seção 6.4.5).

6.4.1 First-Come, First Served (FCFS)

A forma de escalonamento mais elementar consiste em simplesmente atender as tarefas em sequência, à medida em que elas se tornam prontas (ou seja, conforme sua ordem de ingresso na fila de tarefas prontas). Esse algoritmo é conhecido como FCFS – *First Come - First Served* – e tem como principal vantagem sua simplicidade.

O diagrama da Figura 6.1 mostra o escalonamento das tarefas de Tabela 6.1 usando o algoritmo FCFS cooperativo (ou seja, sem *quantum* ou outras interrupções). Os quadros sombreados representam o uso do processador (observe que em cada instante apenas uma tarefa ocupa o processador). Os quadros brancos representam as tarefas que já ingressaram no sistema e estão aguardando o processador (tarefas prontas).

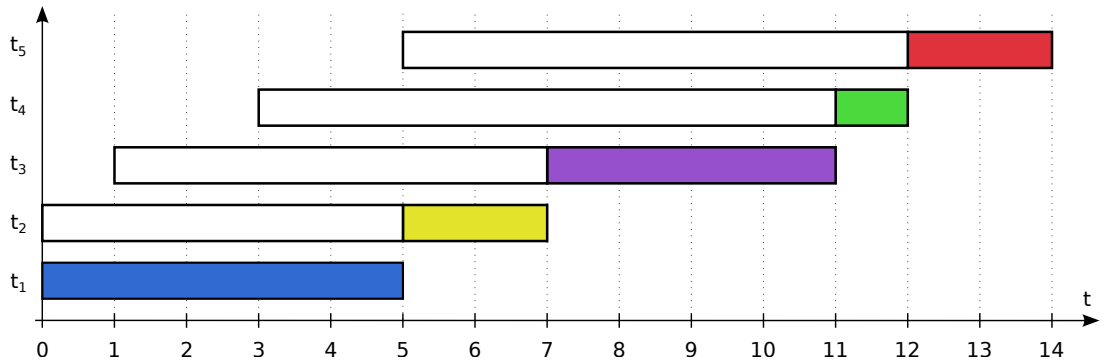


Figura 6.1: Escalonamento FCFS.

Calculando o tempo médio de execução (T_t, a média de $t_t(t_i)$) e o tempo médio de espera (T_w, a média de $t_w(t_i)$) para a execução da Figura 6.1, temos:

$$T_t = \frac{t_t(t_1) + \cdots + t_t(t_5)}{5} = \frac{(5-0) + (7-0) + (11-1) + (12-3) + (14-5)}{5}$$

$$= \frac{5+7+10+9+9}{5} = \frac{40}{5} = 8,0s$$

$$T_w = \frac{t_w(t_1)+\cdots+t_w(t_5)}{5} = \frac{(0-0)+(5-0)+(7-1)+(11-3)+(12-5)}{5}$$

$$= \frac{0+5+6+8+7}{5} = \frac{26}{5} = 5,2s$$

6.4.2 Round-Robin (RR)

A adição da preempção por tempo ao escalonamento FCFS dá origem a outro algoritmo de escalonamento bastante popular, conhecido como **escalonamento por revezamento**, ou *Round-Robin*. Considerando as tarefas definidas na tabela anterior e um quantum $t_q = 2s$, seria obtida a sequência de escalonamento apresentada na Figura 6.2.

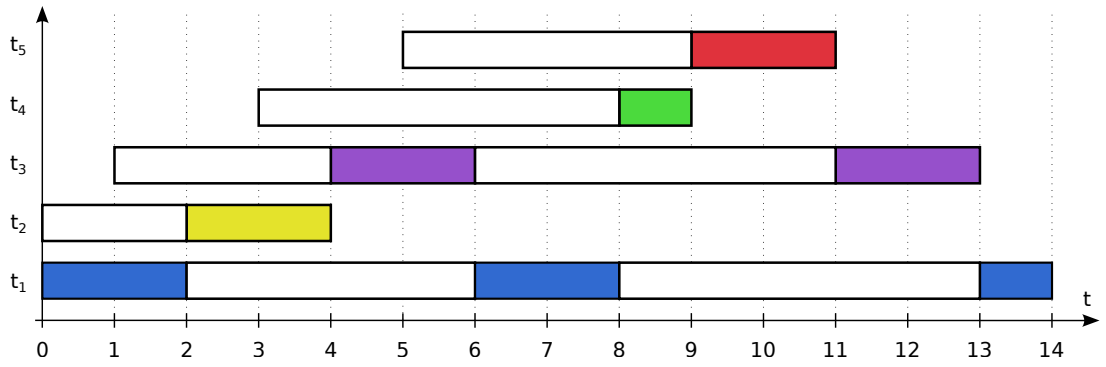


Figura 6.2: Escalonamento *Round-Robin*.

Na Figura 6.2, é importante observar que a execução das tarefas não obedece uma sequência óbvia como $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \rightarrow t_5 \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots$, mas uma sequência bem mais complexa: $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_1 \rightarrow t_4 \rightarrow t_5 \rightarrow \ldots$. Isso ocorre por causa da ordem das tarefas na fila de tarefas prontas. Por exemplo, a tarefa t_1 para de executar e volta à fila de tarefas prontas no instante $t = 2$, antes de t_4 ter entrado no sistema (em $t = 3$). Por isso, t_1 retorna ao processador antes de t_4 (em $t = 6$). A Figura 6.3 detalha a evolução da fila de tarefas prontas ao longo do tempo, para esse exemplo.

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para a execução da Figura 6.2, temos:

$$T_t = \frac{t_t(t_1)+\cdots+t_t(t_5)}{5} = \frac{14+4+12+6+6}{5} = \frac{42}{5} = 8,4s$$

$$T_w = \frac{t_w(t_1)+\cdots+t_w(t_5)}{5} = \frac{9+2+8+5+4}{5} = \frac{28}{5} = 5,6s$$

Observa-se o aumento nos tempos T_t e T_w em relação ao algoritmo FCFS simples, o que mostra que o algoritmo *Round-Robin* é menos eficiente para a execução de tarefas em lote. Entretanto, por distribuir melhor o uso do processador entre as tarefas ao longo do tempo, ele proporciona tempos de resposta bem melhores às aplicações interativas. Outro problema deste escalonador é o aumento no número de trocas de contexto, que,

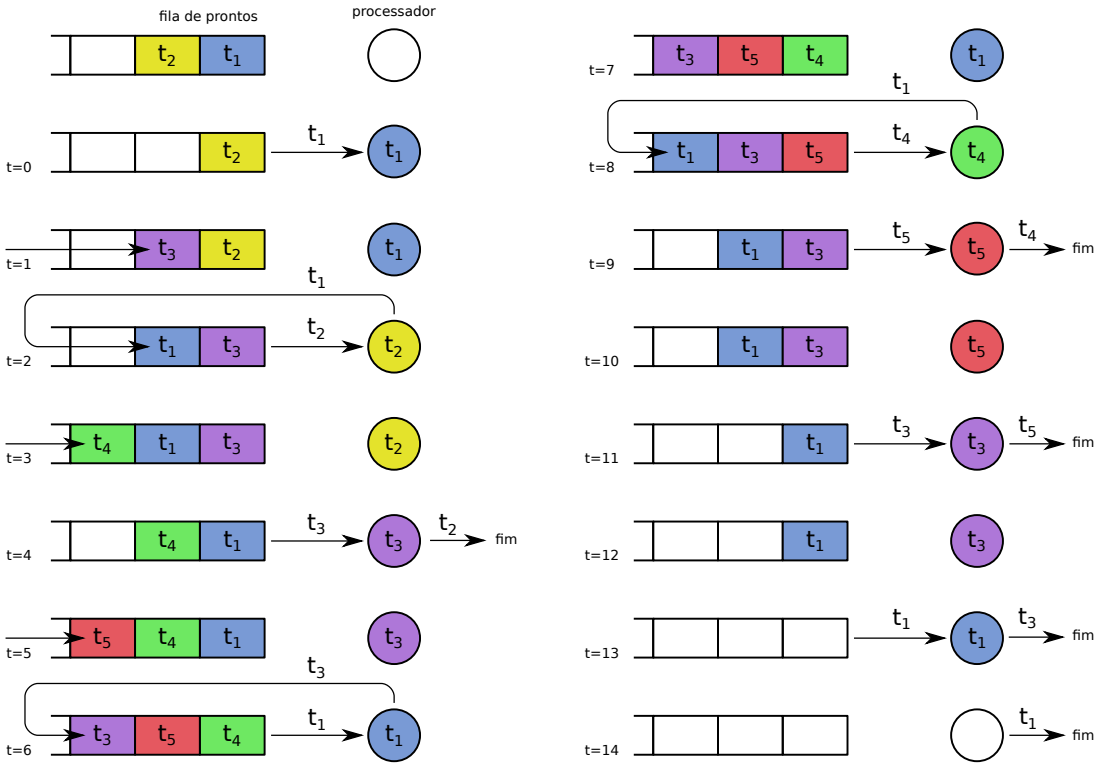


Figura 6.3: Evolução da fila de tarefas prontas no escalonamento *Round-Robin*.

se for significativo, pode levar a uma degradação de desempenho, conforme discutido na Seção 5.2.

Deve-se observar que os algoritmos de escalonamento FCFS e RR não levam em conta a importância das tarefas nem seu comportamento em relação ao uso dos recursos. Por exemplo, nesses algoritmos as tarefas orientadas a entrada/saída irão receber menos tempo de processador que as tarefas orientadas a processamento (pois as primeiras geralmente não usam integralmente seus *quanta* de tempo), o que pode ser prejudicial para aplicações interativas.

6.4.3 Shortest Job First (SJF)

O algoritmo de escalonamento conhecido como *menor tarefa primeiro* (SJF - *Shortest Job First*) consiste em atribuir o processador à menor (mais curta) tarefa da fila de tarefas prontas. Esse algoritmo (e sua versão preemptiva, SRTF) proporciona os menores tempos médios de espera das tarefas. Aplicando-se este algoritmo às tarefas da Tabela 6.1, obtém-se o escalonamento apresentado na Figura 6.4.

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para a execução da Figura 6.4, temos:

$$T_t = \frac{t_t(t_1) + \cdots + t_t(t_5)}{5} = \frac{14+2+5+4+4}{5} = \frac{29}{5} = 5,8s$$

$$T_w = \frac{t_w(t_1) + \cdots + t_w(t_5)}{5} = \frac{9+0+1+3+2}{5} = \frac{15}{5} = 3,0s$$

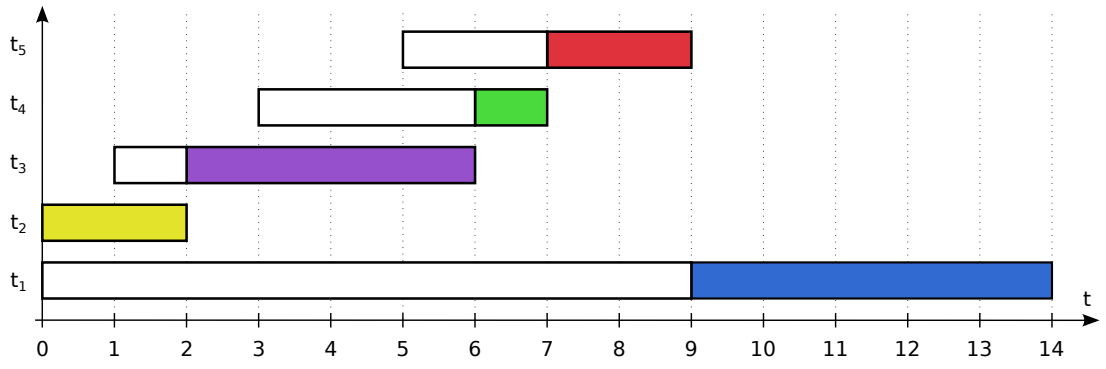


Figura 6.4: Escalonamento SJF.

A maior dificuldade no uso do algoritmo SJF consiste em estimar a priori a duração de cada tarefa, antes de sua execução. Com exceção de algumas tarefas em lote ou de tempo real, essa estimativa é inviável; por exemplo, como estimar por quanto tempo um editor de textos irá ser utilizado? Por causa desse problema, o algoritmo SJF puro é pouco utilizado. No entanto, ao associarmos o algoritmo SJF ao algoritmo RR, esse algoritmo pode ser de grande valia, sobretudo para tarefas orientadas a entrada/saída.

Supondo uma tarefa orientada a entrada/saída em um sistema preemptivo com $t_q = 10ms$. Nas últimas 3 vezes em que recebeu o processador, essa tarefa utilizou $3ms$, $4ms$ e $4,5ms$ de cada quantum recebido. Com base nesses dados históricos, é possível estimar qual a utilização de *quantum* provável da tarefa na próxima vez em que receber o processador. Essa estimativa pode ser feita por média simples (cálculo mais rápido) ou por extrapolação (cálculo mais complexo, podendo influenciar o tempo de troca de contexto t_{tc}).

A estimativa de uso do próximo quantum assim obtida pode ser usada como base para a aplicação do algoritmo SJF, o que irá beneficiar as tarefas orientadas a entrada/saída, que usam menos o processador. Obviamente, uma tarefa pode mudar de comportamento repentinamente, passando de uma fase de entrada/saída para uma fase de processamento, ou vice-versa. Nesse caso, a estimativa de uso do próximo *quantum* será incorreta durante alguns ciclos, mas logo voltará a refletir o comportamento atual da tarefa. Por essa razão, apenas a história recente da tarefa deve ser considerada (3 a 5 últimas ativações).

Outro problema associado ao escalonamento SJF é a possibilidade de *inanição* (*starvation*) das tarefas mais longas. Caso o fluxo de tarefas curtas chegando ao sistema seja elevado, as tarefas mais longas nunca serão escolhidas para receber o processador e vão literalmente "morrer de fome", esperando na fila sem poder executar. Esse problema pode ser resolvido através de técnicas de envelhecimento de tarefas, como a apresentada na Seção 6.4.6.

6.4.4 Shortest Remaining Time First (SRTF)

O algoritmo SJF é cooperativo, ou seja, uma vez que uma tarefa recebe o processador, ela executa até encerrar (ou liberá-lo explicitamente). Em uma variante preemptiva, o escalonador deve comparar a duração prevista de cada nova tarefa que ingressa no sistema com o tempo de processamento restante das demais tarefas presentes, inclusive aquela que está executando no momento. Caso a nova tarefa tenha um tempo restante menor, ela recebe o processador. Essa abordagem é denominada

por alguns autores de *menor tempo restante primeiro* (SRTF – *Short Remaining Time First*) [Tanenbaum, 2003]. O algoritmo SRTF está exemplificado na Figura 6.5.

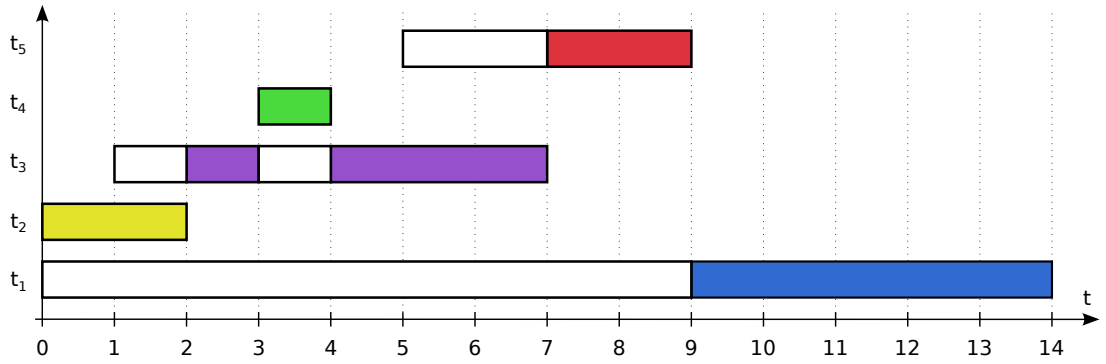


Figura 6.5: Escalonamento SRTF.

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para a execução da Figura 6.5, temos:

$$T_t = \frac{t_t(t_1) + \cdots + t_t(t_5)}{5} = \frac{14+2+6+1+4}{5} = \frac{27}{5} = 5,4s$$

$$T_w = \frac{t_w(t_1) + \cdots + t_w(t_5)}{5} = \frac{9+0+2+0+2}{5} = \frac{13}{5} = 2,6s$$

Pode-se observar que os tempos médios de execução T_t e de espera T_w são os menores observados até aqui. De fato, SRTF é o algoritmo de escalonamento que permite obter os mínimos valores de T_t e T_w. Ele torna o processamento de tarefas curtas muito eficiente, todavia com o risco de inanição das tarefas mais longas.

6.4.5 Escalonamento por prioridades fixas (PRIOc, PRIOp)

Vários critérios podem ser usados para ordenar a fila de tarefas prontas e escolher a próxima tarefa a executar; a data de ingresso da tarefa (usada no FCFS) e sua duração prevista (usada no SJF) são apenas dois deles. Inúmeros outros critérios podem ser especificados, como o comportamento da tarefa (em lote, interativa ou de tempo real), seu proprietário (administrador, gerente, estagiário), seu grau de interatividade, etc.

No escalonamento por prioridade, a cada tarefa é associada uma prioridade, geralmente na forma de um número inteiro, que representa sua importância no sistema. Os valores de prioridade são então usados para definir a ordem de execução das tarefas. O algoritmo de escalonamento por prioridade define um modelo mais genérico de escalonamento, que permite modelar várias abordagens, entre as quais o FCFS e o SJF.

O escalonamento por prioridade pode ser cooperativo ou preemptivo. O diagrama da Figura 6.6 mostra o escalonamento das tarefas da Tabela 6.1 usando o **algoritmo por prioridade cooperativo**, ou PRIOc. Neste exemplo, os valores de prioridade são considerados em uma escala de prioridade positiva, ou seja, valores numéricos maiores indicam maior prioridade.

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para a execução da Figura 6.6, temos:

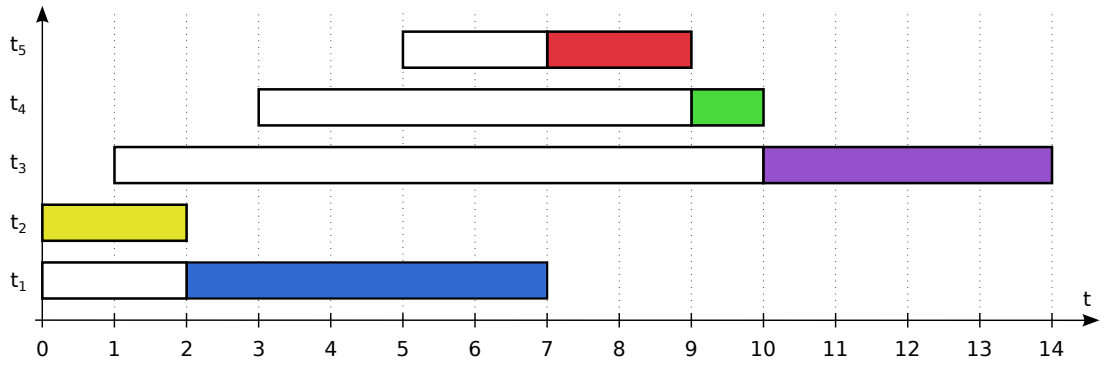


Figura 6.6: Escalonamento por prioridade cooperativo (PRIOc).

$$T_t = \frac{t_t(t_1) + \cdots + t_t(t_5)}{5} = \frac{7+2+13+7+4}{5} = \frac{33}{5} = 6,6s$$

$$T_w = \frac{t_w(t_1) + \cdots + t_w(t_5)}{5} = \frac{2+0+9+6+2}{5} = \frac{19}{5} = 3,8s$$

No **escalonamento por prioridade preemptivo** (PRIOp), quando uma tarefa de maior prioridade se torna disponível para execução, o escalonador entrega o processador a ela, trazendo a tarefa atualmente em execução de volta para a fila de prontas. Em outras palavras, a tarefa em execução pode ser "preemptada" por uma nova tarefa mais prioritária. Esse comportamento é apresentado na Figura 6.7 (observe que, quando t_4 ingressa no sistema, ela recebe o processador e t_1 volta a esperar na fila de prontas).

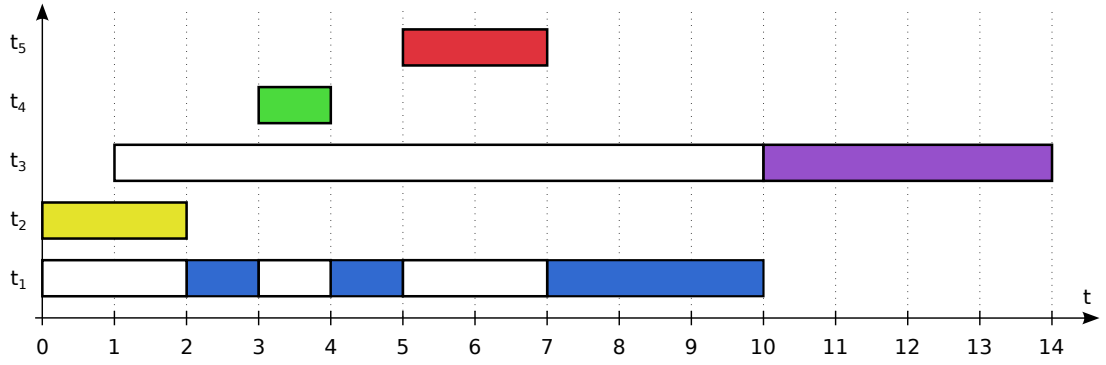


Figura 6.7: Escalonamento por prioridade preemptivo (PRIOp).

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para a execução da Figura 6.7, temos:

$$T_t = \frac{t_t(t_1) + \cdots + t_t(t_5)}{5} = \frac{10+2+13+1+2}{5} = \frac{28}{5} = 5,6s$$

$$T_w = \frac{t_w(t_1) + \cdots + t_w(t_5)}{5} = \frac{5+0+9+0+0}{5} = \frac{14}{5} = 2,8s$$

6.4.6 Escalonamento por prioridades dinâmicas (PRIOd)

No escalonamento por prioridades fixas apresentado na seção anterior, as tarefas de menor prioridade só recebem o processador na ausência de tarefas de maior prioridade. Caso existam tarefas de maior prioridade frequentemente ativas, as de menor prioridade podem "morrer de fome" por nunca conseguir chegar ao processador. Esse fenômeno se denomina **inanição** (do inglês *starvation*).

Para evitar a inanição das tarefas de menor prioridade, um fator interno denominado **envelhecimento** (*aging*) deve ser definido. O envelhecimento aumenta a prioridade da tarefa proporcionalmente ao tempo que ela está aguardando o processador. Dessa forma, o envelhecimento define um esquema de **prioridades dinâmicas**, que permite a elas executar periodicamente e assim evitar a inanição.

Uma forma de implementar o envelhecimento de tarefas está indicada no algoritmo a seguir, que associa duas prioridades a cada tarefa t_i: a prioridade estática ou fixa pe_i, definida por fatores externos, e a prioridade dinâmica pd_i, que evolui ao longo da execução (considera-se uma escala de prioridades positiva).

Definições:
- N : número de tarefas no sistema
- t_i : tarefa i, $1 \leq i \leq N$
- pe_i : prioridade estática da tarefa t_i
- pd_i : prioridade dinâmica da tarefa t_i

Quando uma tarefa nova t_{nova} ingressa no sistema:
- $pe_{nova} \leftarrow$ *prioridade fixa*
- $pd_{nova} \leftarrow pe_{nova}$

Para escolher t_{prox}, a próxima tarefa a executar:
- escolher $t_{prox} \mid pd_{prox} = max_{i=1}^{N}(pd_i)$
- $\forall t_i \neq t_{prox} : pd_i \leftarrow pd_i + \alpha$
- $pd_{prox} \leftarrow pe_{prox}$

O funcionamento é simples: a cada turno o escalonador escolhe como próxima tarefa (t_{prox}) aquela que tiver a maior prioridade dinâmica (pd). A prioridade dinâmica das demais tarefas é aumentada de uma constante α, conhecida como *fator de envelhecimento* (ou seja, essas tarefas "envelhecem" um pouco e no próximo turno terão mais chances de ser escolhidas). Por sua vez, a tarefa escolhida (t_{prox}) "rejuvenesce", tendo sua prioridade dinâmica ajustada de volta para o valor de sua prioridade estática ($pd_{prox} \leftarrow pe_{prox}$).

Aplicando o escalonamento com prioridades dinâmicas com envelhecimento, a execução das tarefas da Tabela 6.1 pode ser observada na Figura 6.8.

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para a execução da Figura 6.8, temos:

$$T_t = \frac{t_t(t_1) + \cdots + t_t(t_5)}{5} = \frac{13 + 2 + 13 + 1 + 2}{5} = \frac{31}{5} = 6,2s$$

$$T_w = \frac{t_w(t_1) + \cdots + t_w(t_5)}{5} = \frac{8 + 0 + 9 + 0 + 0}{5} = \frac{17}{5} = 3,4s$$

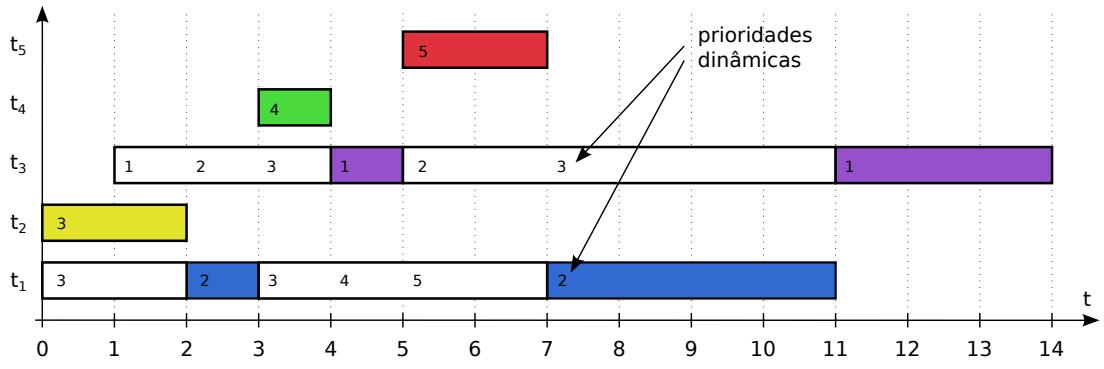


Figura 6.8: Escalonamento por prioridade preemptivo dinâmico (PRIOd).

As prioridades dinâmicas resolvem um problema importante em sistemas de tempo compartilhado. Nesses sistemas, as prioridades definidas pelo usuário estão intuitivamente relacionadas à **proporcionalidade** desejada na divisão do tempo de processamento. Por exemplo, caso um sistema de tempo compartilhado receba três tarefas: t_1 com prioridade 1, t_2 com prioridade 2 e t_3 com prioridade 3, espera-se que t_3 receba mais o processador que t_2, e esta mais que t_1 (assumindo uma escala de prioridades positiva).

Contudo, se associarmos apenas prioridades fixas a um escalonador *Round-Robin*, as tarefas irão executar de forma sequencial, sem a distribuição proporcional do processador. Esse resultado indesejável ocorre porque, a cada fim de *quantum*, sempre a tarefa mais prioritária é escolhida. Essa situação está ilustrada na Figura 6.9.

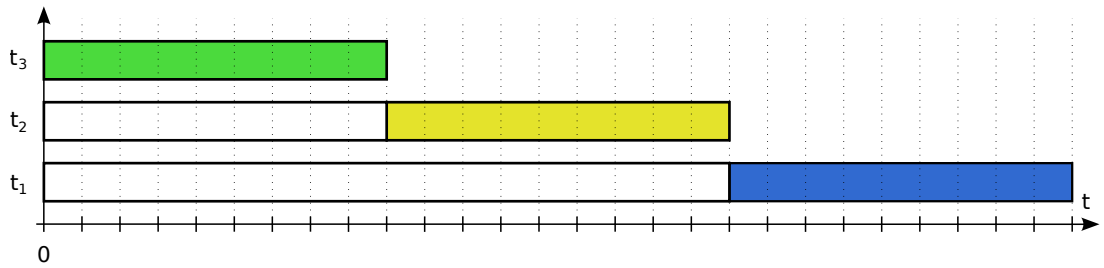


Figura 6.9: Escalonamento *Round-Robin* com prioridades fixas.

Usando o algoritmo de envelhecimento, a divisão do processador entre as tarefas se torna proporcional às suas prioridades estáticas. A Figura 6.10 ilustra a proporcionalidade obtida: percebe-se que todas as três tarefas recebem o processador periodicamente, mas que t_3 recebe mais tempo de processador que t_2, e que t_2 recebe mais que t_1.

6.4.7 Definição de prioridades

A definição da prioridade de uma tarefa é influenciada por diversos fatores, que podem ser classificados em dois grandes grupos:

Fatores externos: informações providas pelo usuário ou o administrador do sistema, que o escalonador não conseguiria estimar sozinho. Os fatores externos mais comuns são a classe do usuário (administrador, diretor, estagiário) o valor pago

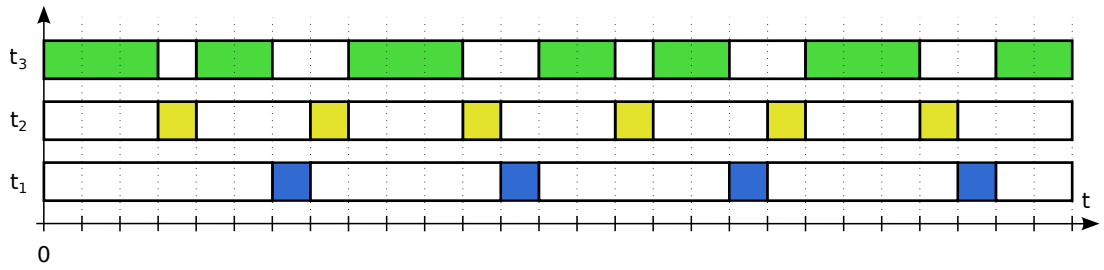


Figura 6.10: Escalonamento *Round-Robin* com prioridades dinâmicas.

pelo uso do sistema (serviço básico, serviço *premium*) e a importância da tarefa em si (um detector de intrusão, um *script* de reconfiguração emergencial, etc.).

Fatores internos: informações que podem ser obtidas ou estimadas pelo próprio escalonador, com base em dados disponíveis no sistema operacional. Os fatores internos mais utilizados são a idade da tarefa, sua duração estimada, sua interatividade, seu uso de memória ou de outros recursos, seu envelhecimento, etc. Os fatores internos mudam continuamente e devem ser recalculados periodicamente pelo escalonador.

Os fatores externos são usualmente sintetizados em um valor inteiro denominado **prioridade estática** (ou *prioridade de base*), que resume a "opinião" do usuário ou administrador sobre a importância daquela tarefa. Em geral, cada família de sistemas operacionais define sua própria escala de prioridades estáticas. Alguns exemplos de escalas comuns são:

Windows (2000 em diante): processos e *threads* são associados a *classes de prioridade* (6 classes para processos e 7 classes para *threads*); a prioridade final de uma *thread* depende de sua prioridade de sua própria classe de prioridade e da classe de prioridade do processo ao qual está associada, assumindo valores entre 0 e 31. As prioridades do processos, apresentadas aos usuários no *Gerenciador de Tarefas*, apresentam os seguintes valores *default*:

- 24: *tempo real*
- 13: *alta*
- 10: *acima do normal*
- 8: *normal*
- 6: *abaixo do normal*
- 4: *baixa* ou *ociosa*

Além disso, geralmente a prioridade da tarefa responsável pela janela ativa recebe um incremento de prioridade: +1 ou +2, conforme a configuração do sistema (essa informação é considerada um fator interno de prioridade).

Linux (núcleo 2.4 em diante): considera duas escalas de prioridade separadas:

Tarefas de tempo real: usam uma escala de 0 a 99 positiva (valores maiores indicar maior prioridade). Somente o núcleo ou o administrador (*root*) podem lançar tarefas de tempo real.

Demais tarefas: usam uma escala que vai de −20 a +19, negativa (valores maiores indicam **menor** prioridade). Esta escala, denominada *nice level*, é padronizada em todos os sistemas UNIX-*like*. A prioridade das tarefas de usuário pode ser ajustada através dos comandos `nice` e `renice`.

6.4.8 Comparação entre os algoritmos apresentados

A Tabela 6.2 traz uma comparação sucinta entres os algoritmos apresentados nesta seção. Pode-se observar que os algoritmos preemptivos (RR, SRTF e PRIOp) possuem um número de trocas de contexto maior que seus correspondentes cooperativos, o que era de se esperar. Também pode-se constatar que o algoritmo SRTF proporciona os melhores tempos médios de execução T_t e de espera T_w, enquanto os piores tempos são providos pelo algoritmo RR (que, no entanto, oferece um melhor tempo de resposta a aplicações interativas).

Observa-se também que o tempo total de processamento é constante, pois ele só depende da carga de processamento de cada tarefa e não da ordem em que são executadas. Contudo, esse tempo pode ser influenciado pelo número de trocas de contexto, caso seja muito elevado.

Algoritmo de escalonamento	FCFS	RR	SJF	SRTF	PRIOc	PRIOp	PRIOd
Tempo médio de execução T_t	8,0	8,4	5,8	5,4	6,6	5,6	6,2
Tempo médio de espera T_w	5,2	5,6	3,0	2,6	3,8	2,8	3,4
Número de trocas de contexto	4	7	4	5	4	6	8
Tempo total de processamento	14	14	14	14	14	14	14

Tabela 6.2: Comparação entre os algoritmos apresentados.

6.4.9 Outros algoritmos de escalonamento

Além dos algoritmos vistos neste capítulo, diversos outros algoritmos foram propostos na literatura, alguns dos quais servem de base conceituais para escalonadores usados em sistemas operacionais reais. Dentre eles, podem ser citados os escalonadores de múltiplas filas, com ou sem *feedback* [Arpaci-Dusseau and Arpaci-Dusseau, 2014], os escalonadores justos [Kay and Lauder, 1988; Ford and Susarla, 1996], os escalonadores multiprocessador [Black, 1990] e multicore [Boyd-Wickizer et al., 2009], os escalonadores de tempo real [Farines et al., 2000] e os escalonadores multimídia [Nieh and Lam, 1997].

6.5 Escalonadores reais

Sistemas operacionais de mercado como Windows, Linux e MacOS são feitos para executar tarefas de diversos tipos, como jogos, editores de texto, conversores de vídeo, backups, etc. Para lidar com essa grande diversidade de tarefas, os escalonadores desses sistemas implementam algoritmos complexos, combinando mais de uma política de escalonamento.

No Linux, as tarefas são divididas em diversas classes de escalonamento, de acordo com suas demandas de processamento. Cada classe possui sua própria fila de

tarefas, em um esquema conhecido como *Multiple Feedback Queues* [Arpaci-Dusseau and Arpaci-Dusseau, 2014]. As classes definidas no escalonador atual (núcleo 4.16, em 2018) são:

Classe `SCHED_DEADLINE`: classe específica para tarefas de tempo real que devem executar periodicamente e respeitar prazos *deadlines* predefinidos. A fila de tarefas desta classe é organizada por um algoritmo chamado *Earliest Deadline First* (EDF), usual em sistemas de tempo real [Farines et al., 2000].

Classe `SCHED_FIFO`: tarefas nesta classe são escalonadas usando uma política com prioridade fixa preemptiva, sem envelhecimento nem *quantum*. Portanto, uma tarefa nesta classe executa até bloquear por recursos, liberar explicitamente o processador (através da chamada de sistema `sched_yield()`) ou ser interrompida por outra tarefa de maior prioridade nesta mesma classe.

Classe `SCHED_RR`: esta classe implementa uma política similar à `SCHED_FIFO`, com a inclusão da preempção por tempo (*Round-Robin*). O valor do *quantum* é proporcional à prioridade de cada tarefa, variando de 10*ms* a 200*ms*.

Classe `SCHED_NORMAL` ou `SCHED_OTHER`: é a classe padrão, que suporta as tarefas interativas dos usuários. As tarefas são escalonadas por uma política baseada em prioridades dinâmicas (envelhecimento) e *Round-Robin* com *quantum* variável.

Classe `SCHED_BATCH`: similar à classe `SCHED_NORMAL`, mas pressupõe que as tarefas são orientadas a processamento (*CPU-bound*) e portanto são ativadas menos frequentemente que as tarefas em `SCHED_NORMAL`.

Classe `SCHED_IDLE`: classe com a menor prioridade de escalonamento; tarefas nesta classe só recebem processador caso não exista nenhuma outra tarefa ativa no sistema.

As tarefas são ativadas de acordo com sua classe, sendo `SCHED_DEADLINE` > `SCHED_FIFO` > `SCHED_RR` > `SCHED_NORMAL/BATCH/IDLE`. Assim, tarefas nas chamadas classes interativas (`SCHED_NORMAL/BATCH/IDLE`) só executam se não houverem tarefas ativas nas classes denominadas de tempo real (`SCHED_DEADLINE/FIFO/RR`).

O núcleo Linux usa algoritmos distintos para as várias classes de escalonamento. Para as tarefas interativas (`SCHED_NORMAL`) é usado o *Completely Fair Scheduler* (CFS), baseado no conceito de escalonadores justos [Kay and Lauder, 1988]. O CFS mantém a lista de tarefas prontas em uma árvore rubro-negra ordenada por tempos de processamento realizado, o que confere uma boa escalabilidade (os custos de busca, inserção e remoção nessa árvore são da ordem de $O(\log n)$). Por outro lado, as tarefas nas classes de tempo real são escalonadas por algoritmos específicos, como EDF (*Earliest Deadline First*) para `SCHED_DEADLINE`, o PRIOp para `SCHED_FIFO` e o PRIOp+RR para `SCHED_RR`.

Referências

D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.

R. Arpaci-Dusseau and A. Arpaci-Dusseau. *Multi-level Feedback Queue*, chapter 8. Arpaci-Dusseau Books, 2014. http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf.

D. L. Black. Scheduling and resource management techniques for multiprocessors. Technical Report CMU-CS-90-152, Carnegie-Mellon University, Computer Science Dept, 1990.

S. Boyd-Wickizer, R. Morris, and M. Kaashoek. Reinventing scheduling for multicore systems. In *12th conference on Hot topics in operating systems*, page 21. USENIX Association, 2009.

A. Burns and A. Wellings. *Real-Time Systems and Programming Languages, 2nd edition*. Addison-Wesley, 1997.

J.-M. Farines, J. da Silva Fraga, and R. S. de Oliveira. *Sistemas de Tempo Real – 12ª Escola de Computação da SBC*. Sociedade Brasileira de Computação, 2000. http://www.das.ufsc.br/gtr/livro.

B. Ford and S. Susarla. CPU Inheritance Scheduling. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105, 1996.

J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, January 1988.

J. Nieh and M. Lam. The design, implementation and evaluation of SMART: a scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 184–197, 1997.

A. Tanenbaum. *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall, 2003.

Capítulo 7

Tópicos em gestão de tarefas

Este capítulo traz tópicos de estudo específicos, que aprofundam ou complementam os temas apresentados nesta parte do livro, mas cuja leitura não é essencial para a compreensão do conteúdo principal da disciplina. Algumas seções deste capítulo podem estar incompletas ou não ter sido revisadas.

7.1 Inversão e herança de prioridades

Um problema importante que pode ocorrer em sistemas com escalonamento baseado em prioridade é a *inversão de prioridades* [Sha et al., 1990]. A inversão de prioridades ocorre quando uma tarefa de alta prioridade é impedida de executar por causa de uma tarefa de baixa prioridade.

Este tipo de problema envolve o conceito de *exclusão mútua*: alguns recursos do sistema devem ser usados por uma tarefa de cada vez, para evitar problemas de consistência de seu estado interno. Isso pode ocorrer com arquivos, portas de entrada/saída, áreas de memória compartilhada e conexões de rede, por exemplo. Quando uma tarefa obtém acesso a um recurso com exclusão mútua, as demais tarefas que desejam usá-lo ficam esperando no estado suspenso, até que o recurso esteja novamente livre. As técnicas usadas para implementar a exclusão mútua são descritas no Capítulo 10.

Para ilustrar esse problema, pode ser considerada a seguinte situação: um determinado sistema possui uma tarefa de alta prioridade t_a, uma tarefa de baixa prioridade t_b e uma tarefa de prioridade média t_m. Além disso, há um recurso R que deve ser acessado em exclusão mútua; para simplificar, somente t_a e t_b estão interessadas em usar esse recurso. A seguinte sequência de eventos, ilustrada na Figura 7.1, é um exemplo de como pode ocorrer uma inversão de prioridades:

1. Em um dado momento, o processador está livre e é alocado a uma tarefa de baixa prioridade t_b que é a única tarefa na fila de prontas;
2. durante seu processamento, t_b solicita acesso exclusivo ao recurso R e começa a usá-lo;
3. a tarefa t_m com prioridade maior que t_b acorda e volta à fila de prontas;
4. t_b volta ao final da fila de tarefas prontas, aguardando o processador; enquanto t_b não voltar a executar, o recurso R permanecerá alocado a ela e ninguém poderá usá-lo;

5. a tarefa de alta prioridade t_a acorda, volta à fila de prontas, recebe o processador e solicita acesso ao recurso R; como o recurso está alocado a t_b, a tarefa t_a é suspensa até que t_b libere o recurso.

Assim, a tarefa de alta prioridade t_a não pode executar, porque o recurso de que necessita está nas mãos da tarefa de baixa prioridade t_b. Por sua vez, t_b não pode executar, pois o processador está ocupado com t_m.

Dessa forma, t_a deve esperar que t_b execute e libere R, o que configura a inversão de prioridades. A espera de t_a pode ser longa, pois t_b tem baixa prioridade e pode demorar a executar novamente, caso existam outras tarefas em execução no sistema (como t_m). Como tarefas de alta prioridade são geralmente críticas para o funcionamento de um sistema, a inversão de prioridades pode ter efeitos graves.

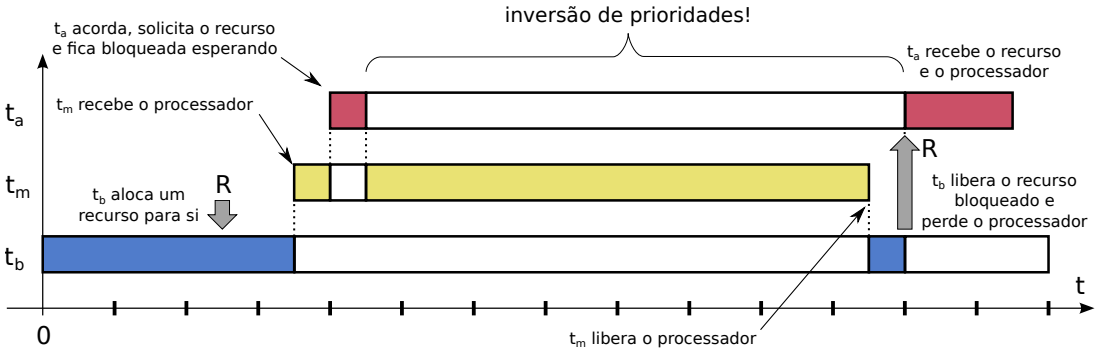


Figura 7.1: Cenário de uma inversão de prioridades.

Uma solução elegante para o problema da inversão de prioridades é obtida através de um *protocolo de herança de prioridade* [Sha et al., 1990]. O protocolo de herança de prioridade mais simples consiste em aumentar temporariamente a prioridade da tarefa t_b que detém o recurso de uso exclusivo R. Caso esse recurso seja requisitado por uma tarefa de maior prioridade t_a, a tarefa t_b "herda" temporariamente a prioridade de t_a, para que possa executar e liberar o recurso R mais rapidamente. Assim que liberar o recurso, t_b retorna à sua prioridade anterior. Essa estratégia está ilustrada na Figura 7.2.

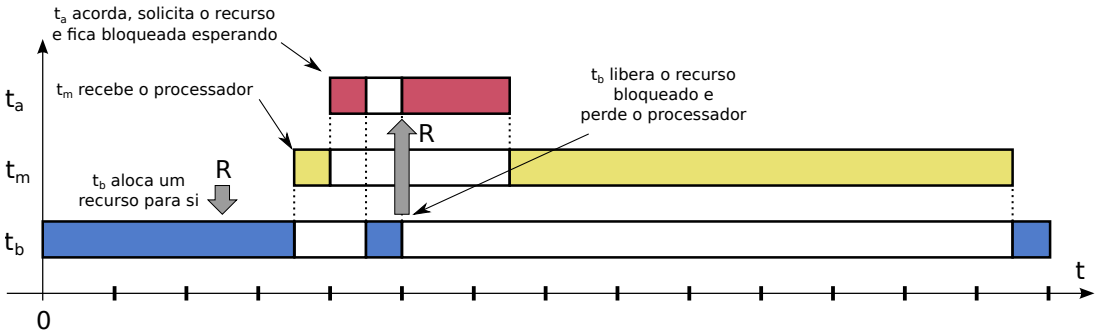


Figura 7.2: Um protocolo de herança de prioridade.

Provavelmente o mais célebre exemplo real de inversão de prioridades tenha ocorrido na sonda espacial *Mars Pathfinder*, enviada pela NASA em 1996 para explorar o solo do planeta Marte (Figura 7.3) [Jones, 1997]. O software da sonda executava sobre o sistema operacional de tempo real *VxWorks* e consistia de 97 *threads* com vários níveis de prioridades fixas. Essas tarefas se comunicavam através de uma área de transferência em

memória compartilhada (na verdade, um *pipe* UNIX), com acesso mutuamente exclusivo controlado por semáforos (semáforos são estruturas de sincronização discutidas na Seção 11.1).

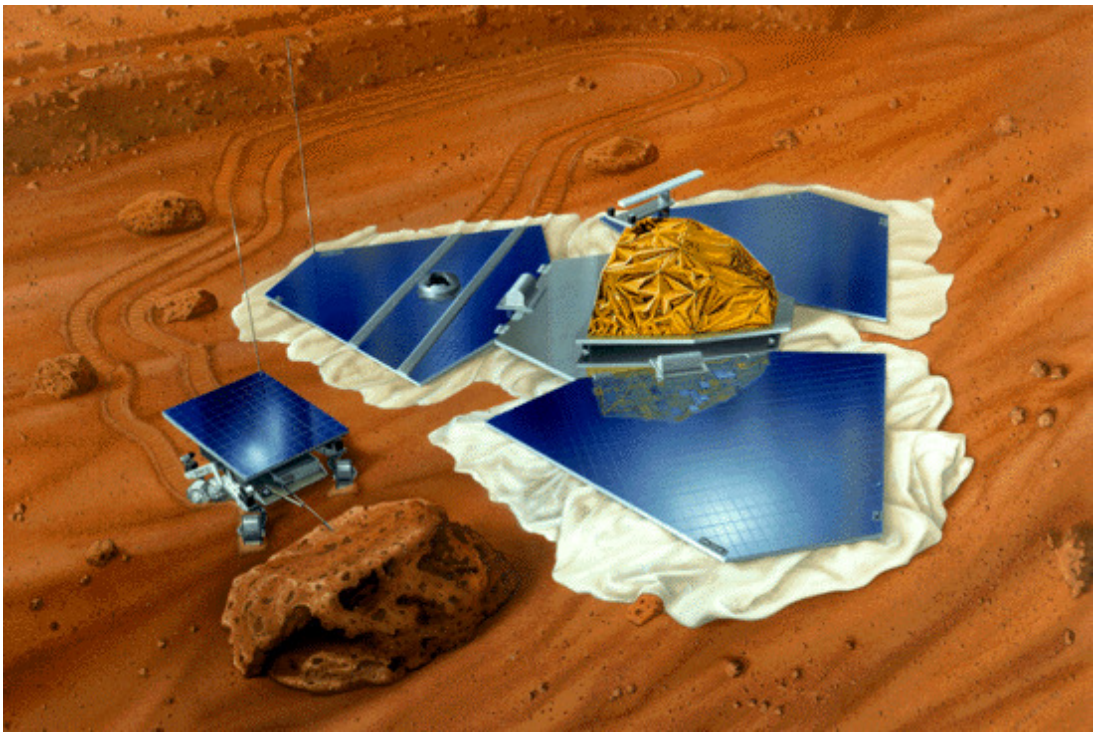


Figura 7.3: Sonda *Mars Pathfinder* com o robô *Sojourner* (NASA).

A gerência da área de transferência estava a cargo de uma tarefa t_{ger}, rápida e de alta prioridade, que era ativada frequentemente para mover blocos de informação para dentro e fora dessa área. A coleta de dados meteorológicos era feita por uma tarefa t_{met} de baixa prioridade, que executava esporadicamente e escrevia seus dados na área de transferência, para uso por outras tarefas. Por fim, a comunicação com a Terra estava sob a responsabilidade de uma tarefa t_{com} de prioridade média e potencialmente demorada (Tabela 7.1 e Figura 7.4).

tarefa	**função**	**prioridade**	**duração**
t_{ger}	gerência da área de transferência	alta	curta
t_{met}	coleta de dados meteorológicos	baixa	curta
t_{com}	comunicação com a Terra	média	longa

Tabela 7.1: Algumas tarefas do software da sonda *Mars Pathfinder*.

Como o sistema *VxWorks* usa um escalonador preemptivo com prioridades fixas, as tarefas eram atendidas conforme suas necessidades na maior parte do tempo. Todavia, a exclusão mútua no acesso à área de transferência escondia uma inversão de prioridades: caso a tarefa de coleta de dados meteorológicos t_{met} perdesse o processador sem liberar a área de transferência, a tarefa de gerência t_{ger} teria de ficar esperando até que t_{met} voltasse a executar para liberar a área. Isso poderia demorar se, por azar, a tarefa de comunicação t_{com} estivesse executando, pois ela tinha mais prioridade que t_{met}.

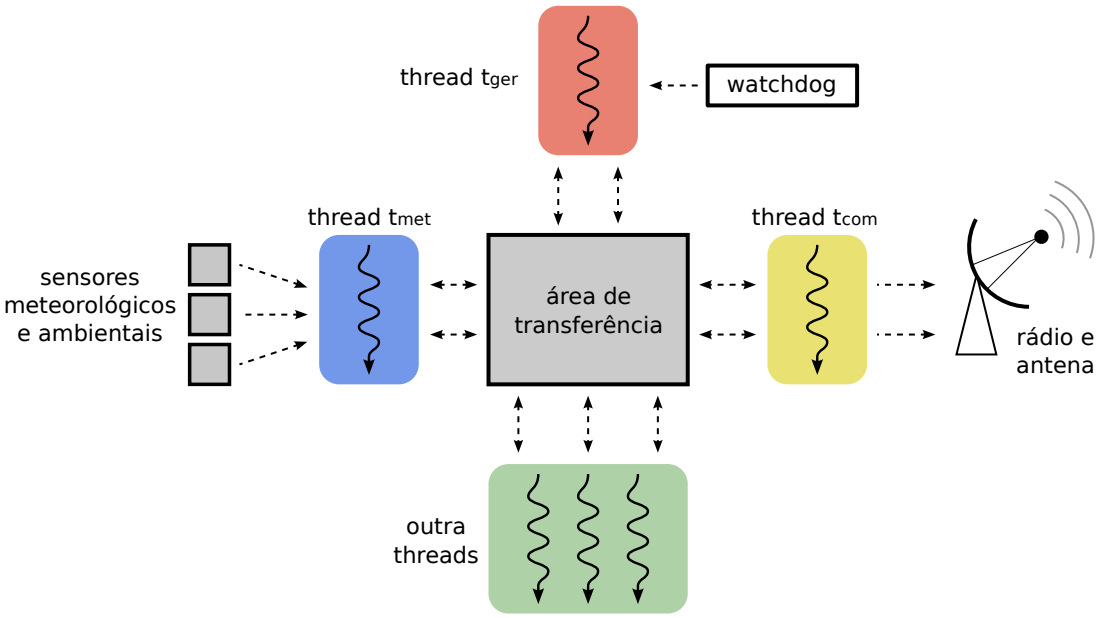


Figura 7.4: Principais tarefas do software embarcado da sonda *Mars Pathfinder*.

Como todos os sistemas críticos, a sonda *Mars Pathfinder* possui um sistema de proteção contra erros, ativado por um temporizador (*watchdog*). Caso a gerência da área de transferência ficasse parada por muito tempo, um procedimento de reinício geral do sistema (*reboot*) era automaticamente ativado pelo temporizador. Dessa forma, a inversão de prioridades provocava reinícios esporádicos e imprevisíveis no software da sonda, interrompendo suas atividades e prejudicando seu funcionamento. A solução foi obtida através de um *patch*[1] que ativou a herança de prioridades: caso a tarefa de gerência t_{ger} fosse bloqueada pela tarefa de coleta de dados t_{met}, esta última herdava a alta prioridade de t_{ger} para poder liberar rapidamente a área de transferência, mesmo se a tarefa de comunicação t_{com} estivesse em execução.

Referências

M. Jones. What really happened on Mars Rover Pathfinder. *ACM Risks-Forum Digest*, 19 (49), 1997.

L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[1] Fica ao leitor imaginar como pode ser depurado e corrigido um bug de software em uma sonda a 100 milhões de Km da Terra...

Parte III

Interação entre tarefas

Capítulo 8

Comunicação entre tarefas

Muitas implementações de sistemas complexos são estruturadas como várias tarefas interdependentes, que cooperam entre si para atingir os objetivos da aplicação, como por exemplo em um navegador Web. Para que as várias tarefas que compõem uma aplicação possam cooperar, elas precisam comunicar informações umas às outras e coordenar suas atividades, para garantir que os resultados obtidos sejam coerentes. Este módulo apresenta os principais conceitos, problemas e soluções referentes à comunicação entre tarefas.

8.1 Objetivos

Nem sempre um programa sequencial é a melhor solução para um determinado problema. Muitas vezes, as implementações são estruturadas na forma de várias tarefas interdependentes que cooperam entre si para atingir os objetivos da aplicação, como por exemplo em um navegador Web. Existem várias razões para justificar a construção de sistemas baseados em tarefas cooperantes, entre as quais podem ser citadas:

Atender vários usuários simultâneos: um servidor de banco de dados ou de e-mail completamente sequencial atenderia um único cliente por vez, gerando atrasos intoleráveis para os demais clientes. Por isso, servidores de rede são implementados com vários processos ou threads, para atender simultaneamente todos os usuários conectados.

Uso de computadores multiprocessador: um programa sequencial executa um único fluxo de instruções por vez, não importando o número de processadores presentes no hardware. Para aumentar a velocidade de execução de uma aplicação, esta deve ser "quebrada" em várias tarefas cooperantes, que poderão ser escalonadas simultaneamente nos processadores disponíveis.

Modularidade: um sistema muito grande e complexo pode ser melhor organizado dividindo suas atribuições em módulos sob a responsabilidade de tarefas interdependentes. Cada módulo tem suas próprias responsabilidades e coopera com os demais módulos quando necessário. Sistemas de interface gráfica, como os projetos *GNOME* [Gnome, 2005] e *KDE* [KDE, 2005], são geralmente construídos dessa forma.

Construção de aplicações interativas: navegadores Web, editores de texto e jogos são exemplos de aplicações com alta interatividade; nelas, tarefas associadas à interface reagem a comandos do usuário, enquanto outras tarefas comunicam através da rede, fazem a revisão ortográfica do texto, renderizam imagens na janela, etc. Construir esse tipo de aplicação de forma totalmente sequencial seria simplesmente inviável.

Para que as tarefas presentes em um sistema possam cooperar, elas precisam **comunicar**, compartilhando as informações necessárias à execução de cada tarefa, e **coordenar** suas atividades, para que os resultados obtidos sejam consistentes (sem erros). Este módulo visa estudar os principais conceitos, problemas e soluções empregados para permitir a comunicação entre tarefas executando em um sistema. A coordenação entre tarefas será estudada a partir do Capítulo 10.

8.2 Escopo da comunicação

Tarefas cooperantes precisam trocar informações entre si. Por exemplo, a tarefa que gerencia os botões e menus de um navegador Web precisa informar rapidamente as demais tarefas caso o usuário clique nos botões *stop* ou *reload*. Outra situação de comunicação frequente ocorre quando o usuário seleciona um texto em uma página da Internet e o arrasta para um editor de textos. Em ambos os casos ocorre a transferência de informação entre duas tarefas distintas.

Implementar a comunicação entre tarefas pode ser simples ou complexo, dependendo da situação. Se as tarefas estão no mesmo processo, elas compartilham a mesma área de memória e a comunicação pode então ser implementada facilmente, usando variáveis globais comuns. Entretanto, caso as tarefas pertençam a processos distintos, não existem variáveis compartilhadas; neste caso, a comunicação tem de ser feita por intermédio do núcleo do sistema operacional, usando chamadas de sistema. Caso as tarefas estejam em computadores distintos, o núcleo deve implementar mecanismos de comunicação específicos, fazendo uso de mecanismos de comunicação em rede. A Figura 8.1 ilustra essas três situações.

Apesar da comunicação poder ocorrer entre *threads*, processos locais ou computadores distintos, com ou sem o envolvimento do núcleo do sistema, os mecanismos de comunicação são habitualmente denominados de forma genérica como "mecanismos IPC" (*Inter-Process Communication*).

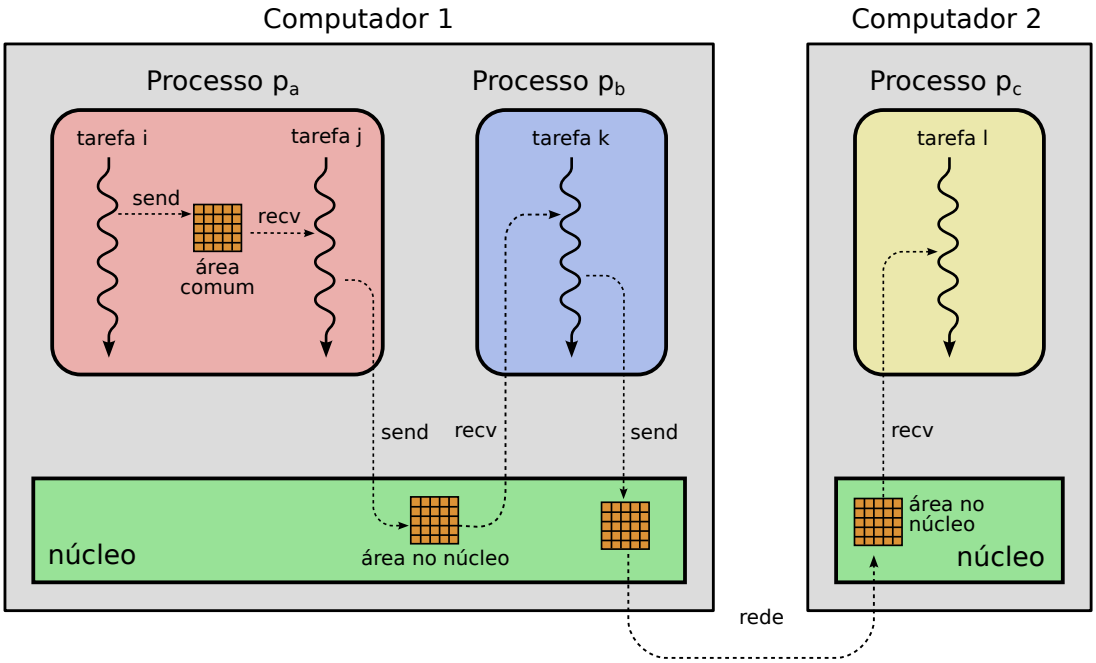


Figura 8.1: Comunicação intraprocesso ($t_i \rightarrow t_j$), interprocessos ($t_j \rightarrow t_k$) e intersistemas ($t_k \rightarrow t_l$).

8.3 Aspectos da comunicação

A implementação da comunicação entre tarefas pode ocorrer de várias formas. Ao definir os mecanismos de comunicação oferecidos por um sistema operacional, seus projetistas devem considerar muitos aspectos, como o formato dos dados a transferir, o sincronismo exigido nas comunicações, a necessidade de *buffers* e o número de emissores/receptores envolvidos em cada ação de comunicação. As próximas seções analisam alguns dos principais aspectos que caracterizam e distinguem entre si os vários mecanismos de comunicação.

8.3.1 Comunicação direta ou indireta

De forma mais abstrata, a comunicação entre tarefas pode ser implementada por duas primitivas básicas: *enviar (dados, destino)*, que envia os dados relacionados ao destino indicado, e *receber (dados, origem)*, que recebe os dados previamente enviados pela origem indicada. Essa abordagem, na qual o emissor identifica claramente o receptor e vice-versa, é denominada **comunicação direta**.

Poucos sistemas empregam a comunicação direta; na prática são utilizados mecanismos de **comunicação indireta**, por serem mais flexíveis. Na comunicação indireta, emissor e receptor não precisam se conhecer, pois não interagem diretamente entre si. Eles se relacionam através de um **canal de comunicação**, que é criado pelo sistema operacional, geralmente a pedido de uma das partes. Neste caso, as primitivas de comunicação não designam diretamente tarefas, mas canais de comunicação aos quais as tarefas estão associadas: *enviar (dados, canal)* e *receber (dados, canal)*. A Figura 8.2 ilustra essas duas formas de comunicação.

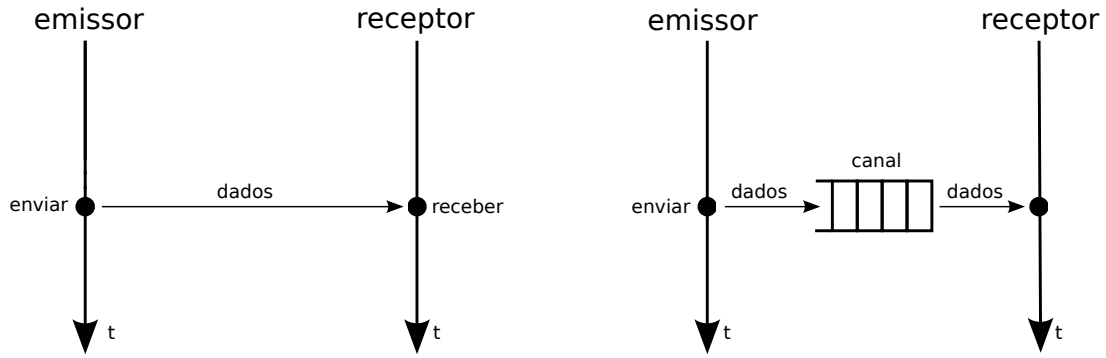


Figura 8.2: Comunicação direta (esquerda) e indireta (direita).

8.3.2 Sincronismo

Em relação aos aspectos de sincronismo do canal de comunicação, a comunicação entre tarefas pode ser:

Síncrona (ou bloqueante): quando as operações de envio e recepção de dados bloqueiam (suspendem) as tarefas envolvidas até a conclusão da comunicação: o emissor será bloqueado até que a informação seja recebida pelo receptor, e vice-versa. A Figura 8.3 apresenta os diagramas de tempo de duas situações frequentes em sistemas com comunicação síncrona.

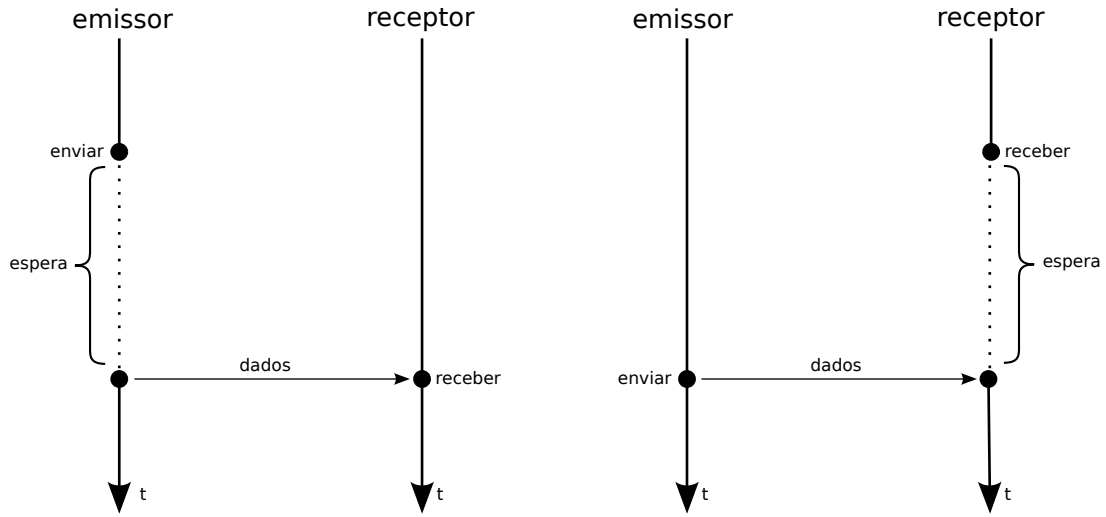


Figura 8.3: Comunicação síncrona.

Assíncrona (ou não-bloqueante): em um sistema com comunicação assíncrona, as primitivas de envio e recepção não são bloqueantes: caso a comunicação não seja possível no momento em que cada operação é invocada, esta retorna imediatamente com uma indicação de erro. Deve-se observar que, caso o emissor e o receptor operem ambos de forma assíncrona, torna-se necessário criar um canal ou *buffer* para armazenar os dados da comunicação entre eles. Sem esse canal, a comunicação se tornará inviável, pois raramente ambos estarão prontos para comunicar ao mesmo tempo. Esta forma de comunicação está representada no diagrama de tempo da Figura 8.4.

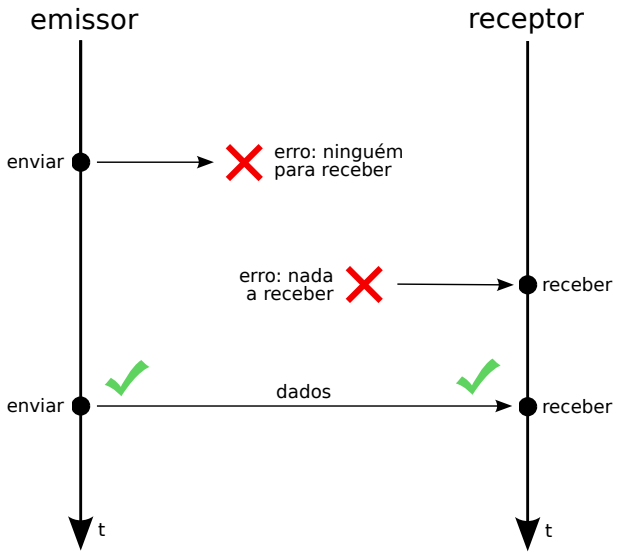


Figura 8.4: Comunicação assíncrona.

Semissíncrona (ou semibloqueante): primitivas de comunicação semissíncronas têm um comportamento síncrono (bloqueante) durante um prazo pré-definido. Caso esse prazo se esgote sem que a comunicação tenha ocorrido, a primitiva se encerra com uma indicação de erro. Para refletir esse comportamento, as primitivas de comunicação recebem um parâmetro adicional, o *prazo*: *enviar (dados, destino, prazo)* e *receber (dados, origem, prazo)*. A Figura 8.5 ilustra duas situações em que ocorre esse comportamento.

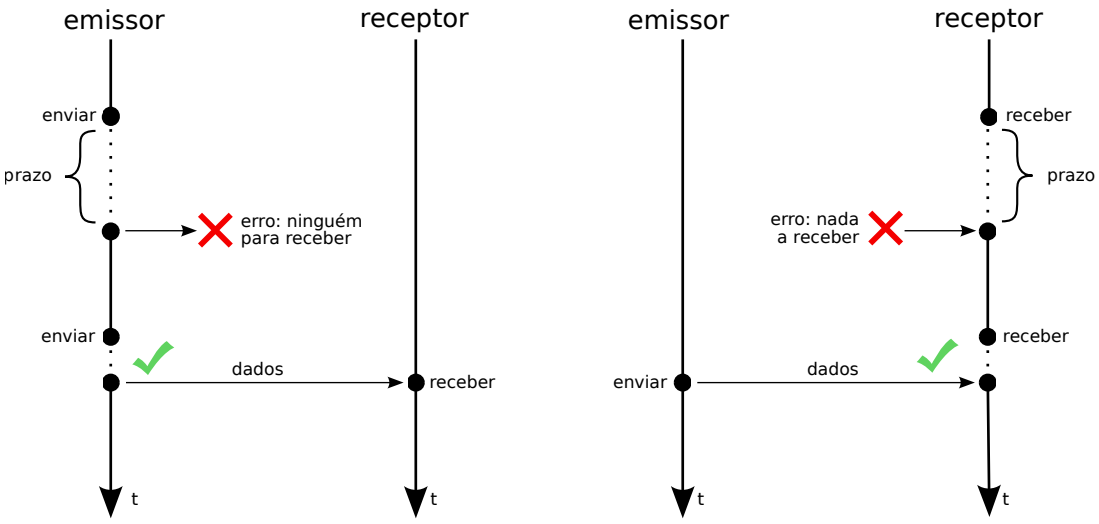


Figura 8.5: Comunicação semissíncrona.

8.3.3 Formato de envio

A informação enviada pelo emissor ao receptor pode ser vista basicamente de duas formas: como uma **sequência de mensagens** independentes, cada uma com seu próprio conteúdo, ou como um **fluxo sequencial** e contínuo de dados, imitando o comportamento de um arquivo com acesso sequencial.

Na abordagem baseada em mensagens, cada mensagem consiste de um pacote de dados que pode ser tipado ou não. Esse pacote é recebido ou descartado pelo receptor em sua íntegra; não existe a possibilidade de receber "meia mensagem" (Figura 8.6). Exemplos de sistema de comunicação orientados a mensagens incluem as *message queues* do UNIX e os protocolos de rede IP e UDP, apresentados na Seção 9.

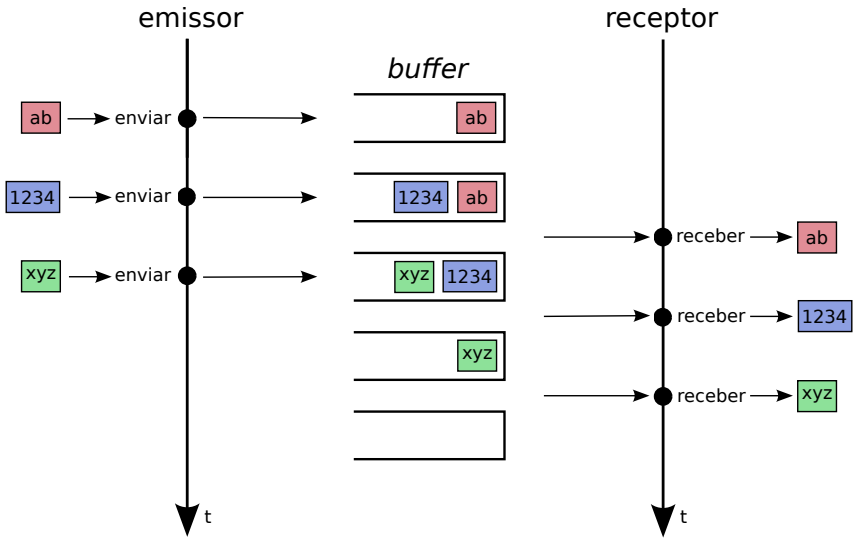


Figura 8.6: Comunicação baseada em mensagens.

Caso a comunicação seja definida como um fluxo contínuo de dados, o canal de comunicação é visto como o equivalente a um arquivo: o emissor "escreve" dados nesse canal, que serão "lidos" pelo receptor respeitando a ordem de envio dos dados. Não há separação lógica entre os dados enviados em operações separadas: eles podem ser lidos byte a byte ou em grandes blocos a cada operação de recepção, a critério do receptor. A Figura 8.7 apresenta o comportamento dessa forma de comunicação.

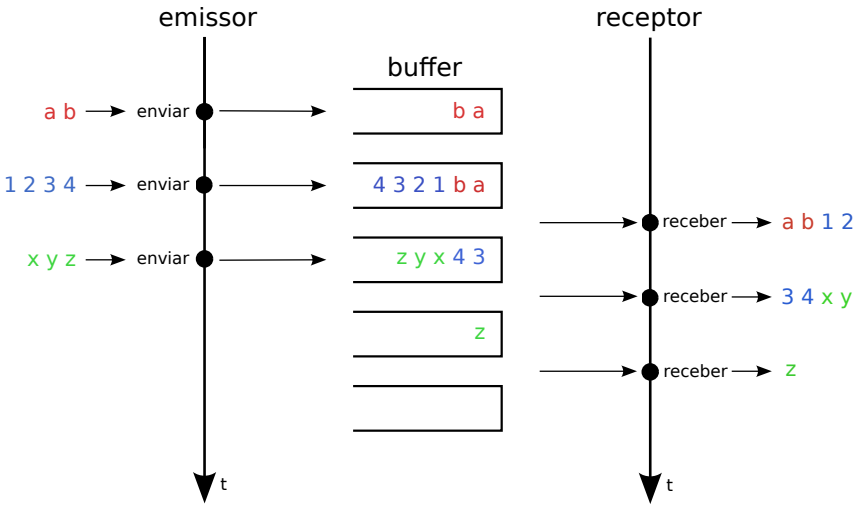


Figura 8.7: Comunicação baseada em fluxo de dados.

Exemplos de sistemas de comunicação orientados a fluxo de dados incluem os *pipes* do UNIX e o protocolo de rede TCP/IP (este último é normalmente classificado como *orientado a conexão*, com o mesmo significado). Nestes dois exemplos a analogia

com o conceito de arquivos é tão forte que os canais de comunicação são identificados por descritores de arquivos e as chamadas de sistema `read` e `write` (normalmente usadas com arquivos) são usadas para enviar e receber os dados. Esses exemplos são apresentados em detalhes na Seção 9.

8.3.4 Capacidade dos canais

O comportamento síncrono ou assíncrono de um canal de comunicação pode ser afetado pela presença de *buffers* que permitam armazenar temporariamente os dados em trânsito, ou seja, as informações enviadas pelo emissor e que ainda não foram recebidas pelo receptor. Em relação à capacidade de *buffering* do canal de comunicação, três situações devem ser analisadas:

Capacidade nula ($n = 0$): neste caso, o canal não pode armazenar dados; a comunicação é feita por transferência direta dos dados do emissor para o receptor, sem cópias intermediárias. Caso a comunicação seja síncrona, o emissor permanece bloqueado até que o destinatário receba os dados, e vice-versa. Essa situação específica (comunicação síncrona com canais de capacidade nula) implica em uma forte sincronização entre as partes, sendo por isso denominada *Rendez-Vous* (termo francês para "encontro"). A Figura 8.3 ilustra dois casos de *Rendez-Vous*. Por outro lado, a comunicação assíncrona torna-se inviável usando canais de capacidade nula (conforme discutido na Seção 8.3.2).

Capacidade infinita ($n = \infty$): o emissor sempre pode enviar dados, que serão armazenados no *buffer* do canal enquanto o receptor não os consumir. Obviamente essa situação não existe na prática, pois todos os sistemas de computação têm capacidade de memória e de armazenamento finitas. No entanto, essa simplificação é útil no estudo dos algoritmos de comunicação e sincronização, pois torna menos complexas a modelagem e análise dos mesmos.

Capacidade finita ($0 < n < \infty$): neste caso, uma quantidade finita (n) de dados pode ser enviada pelo emissor sem que o receptor os consuma. Todavia, ao tentar enviar dados em um canal já saturado, o emissor poderá ficar bloqueado até surgir espaço no *buffer* do canal e conseguir enviar (comportamento síncrono) ou receber um retorno indicando o erro (comportamento assíncrono). A maioria dos sistemas reais opera com canais de capacidade finita.

Para exemplificar esse conceito, a Figura 8.8 apresenta o comportamento de duas tarefas trocando dados através de um canal de comunicação com capacidade para duas mensagens e comportamento bloqueante.

8.3.5 Confiabilidade dos canais

Quando um canal de comunicação transporta todos os dados enviados através dele para seus receptores, respeitando seus valores e a ordem em que foram enviados, ele é chamado de **canal confiável**. Caso contrário, trata-se de um **canal não-confiável**. Há várias possibilidades de erros envolvendo o canal de comunicação, ilustradas na Figura 8.9:

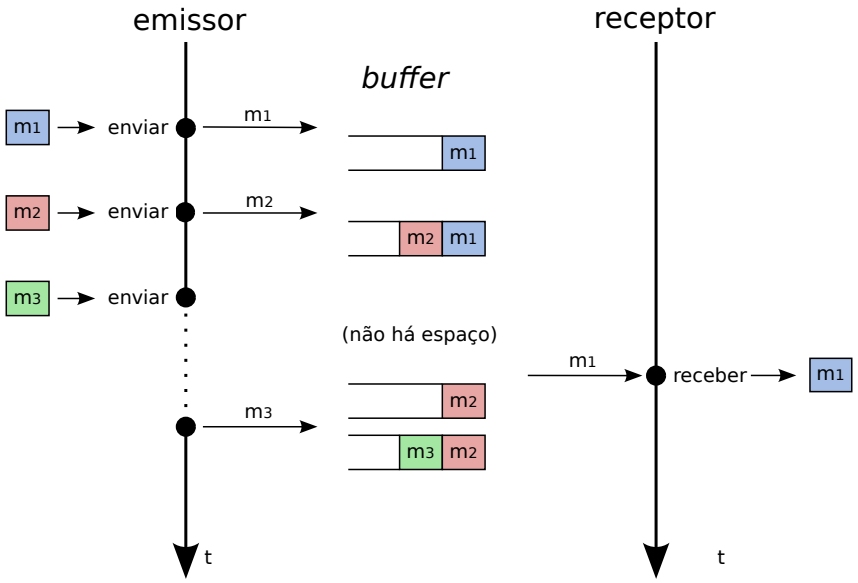


Figura 8.8: Comunicação bloqueante usando um canal com capacidade 2.

Perda de dados: nem todos os dados enviados através do canal chegam ao seu destino; podem ocorrer perdas de mensagens (no caso de comunicação orientada a mensagens) ou de sequências de bytes, no caso de comunicação orientada a fluxo de dados.

Perda de integridade: os dados enviados pelo canal chegam ao seu destino, mas podem ocorrer modificações em seus valores devido a interferências externas.

Perda da ordem: todos os dados enviados chegam íntegros ao seu destino, mas o canal não garante que eles serão entregues na ordem em que foram enviados. Um canal em que a ordem dos dados é garantida é denominado **canal FIFO** ou **canal ordenado**.

Os canais de comunicação usados no interior de um sistema operacional para a comunicação entre processos ou *threads* locais são geralmente confiáveis, ao menos em relação à perda ou corrupção de dados. Isso ocorre porque a comunicação local é feita através da cópia de áreas de memória, operação em que não há risco de erros. Por outro lado, os canais de comunicação entre computadores distintos envolvem o uso de tecnologias de rede, cujos protocolos básicos de comunicação são não-confiáveis (como os protocolos *Ethernet*, IP e UDP). Mesmo assim, protocolos de rede de nível mais elevado, como o TCP, permitem construir canais de comunicação confiáveis.

8.3.6 Número de participantes

Nas situações de comunicação apresentadas até agora, cada canal de comunicação envolve apenas um emissor e um receptor. No entanto, existem situações em que uma tarefa necessita comunicar com várias outras, como por exemplo em sistemas de *chat* ou mensagens instantâneas (IM – *Instant Messaging*). Dessa forma, os mecanismos de comunicação também podem ser classificados de acordo com o número de tarefas participantes:

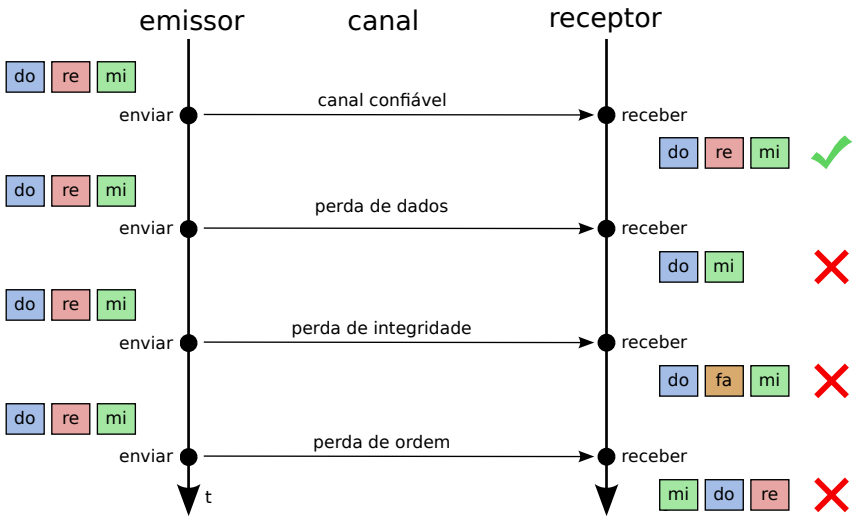


Figura 8.9: Comunicação com canais não confiáveis.

1:1: quando exatamente um emissor e um receptor interagem através do canal de comunicação; é a situação mais frequente, implementada por exemplo nos *pipes* UNIX e no protocolo TCP.

M:N: quando um ou mais emissores enviam mensagens para um ou mais receptores. Duas situações distintas podem se apresentar neste caso:

- Cada mensagem é recebida por **apenas um receptor** (em geral aquele que pedir primeiro); neste caso a comunicação continua sendo ponto-a-ponto, através de um canal compartilhado. Essa abordagem é conhecida como *mailbox* (Figura 8.10), sendo implementada nas *message queues* do UNIX e Windows e também nos *sockets* do protocolo UDP. Na prática, o *mailbox* funciona como um *buffer* de dados, no qual os emissores depositam mensagens e os receptores as consomem.
- Cada mensagem é recebida por **vários receptores** (cada receptor recebe uma cópia da mensagem). Essa abordagem, ilustrada na Figura 8.11, é conhecida como *barramento de mensagens* (*message bus*), *canal de eventos* ou ainda *canal publish-subscribe*. Na área de redes, essa forma de comunicação é chamada de *difusão de mensagens* (*multicast*). Exemplos dessa abordagem podem ser encontrados no D-Bus [Free Desktop, 2018], o barramento de mensagens usado nos ambientes de *desktop* Gnome e KDE, e no COM, a infraestrutura de comunicação interna entre componentes nos sistemas Windows [Microsoft, 2018].

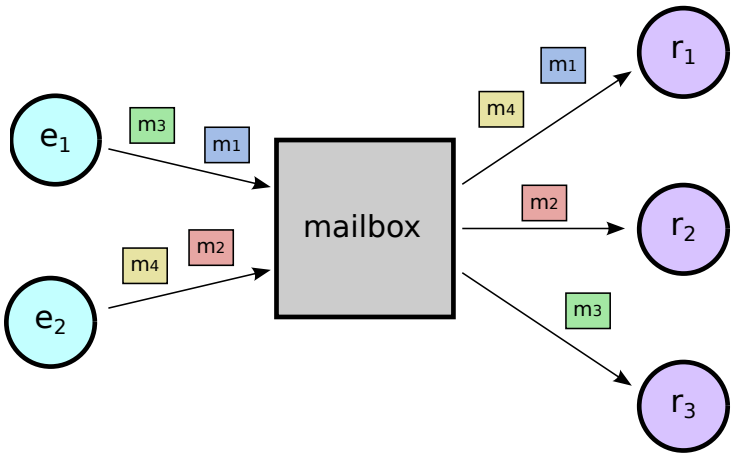


Figura 8.10: Comunicação M:N através de um *mailbox*.

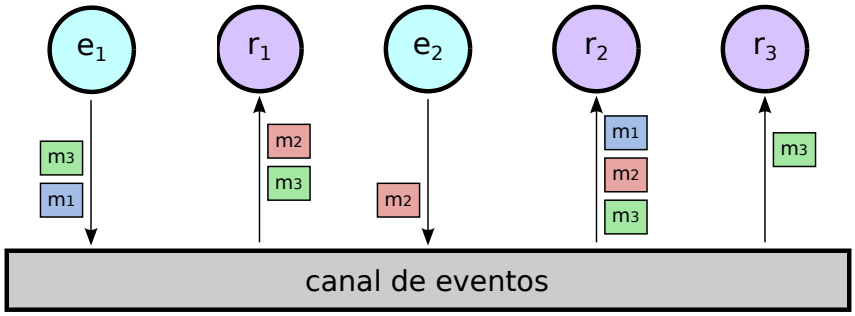


Figura 8.11: Comunicação M:N através de um barramento de mensagens.

Referências

Free Desktop. D-Bus - a message bus system. https://www.freedesktop.org/wiki/Software/dbus/, 2018.

Gnome. Gnome: the free software desktop project. http://www.gnome.org, 2005.

KDE. KDE desktop project. http://www.kde.org, 2005.

Microsoft. Component object model (COM). https://docs.microsoft.com/en-us/windows/desktop/com/component-object-model--com--portal, 2018.

Capítulo 9

Mecanismos de comunicação

Neste capítulo são apresentados alguns mecanismos de comunicação usados com frequência em sistemas operacionais, com ênfase em sistemas UNIX. Mais detalhes sobre estes e outros mecanismos podem ser obtidos em [Stevens, 1998; Robbins and Robbins, 2003]. Mecanismos de comunicação implementados nos sistemas Windows são apresentados em [Petzold, 1998; Hart, 2004].

9.1 Pipes

Um dos mecanismos de comunicação entre processos mais simples de usar no ambiente UNIX é o *pipe*, ou "cano". Um *pipe* é um canal de comunicação unidirecional entre dois processos. Na interface de linha de comandos UNIX, o *pipe* é frequentemente usado para conectar a saída padrão (*stdout*) de um processo à entrada padrão (*stdin*) de outro processo, permitindo assim a comunicação entre eles. A linha de comando a seguir traz um exemplo do uso de *pipes*:

```
1 $ who | grep marcos | sort
```

Esse comando lança simultaneamente os processos `who`, `grep` e `sort`, conectados por dois *pipes*. O comando `who` gera uma listagem de usuários conectados ao computador em sua saída padrão. O comando `grep marcos` é um filtro que lê as linhas de sua entrada padrão e envia para sua saída padrão somente as linhas contendo a *string* "`marcos`". O comando `sort` ordena a linhas recebidas em sua entrada padrão e as envia para sua saída padrão.

Ao associar esses comandos com *pipes*, é produzida uma lista ordenada das linhas de saída do comando `who` que contêm a *string* `marcos`, como mostra a figura 9.1. Deve-se observar que todos os processos envolvidos são lançados simultaneamente; suas ações são coordenadas pelo comportamento síncrono dos *pipes*.

O *pipe* pode ser classificado como um canal de comunicação local entre dois processos (1:1), unidirecional, síncrono, orientado a fluxo, confiável e com capacidade finita (os *pipes* do Linux armazenam 4 KBytes por default). O *pipe* é visto pelos processos como um arquivo, ou seja, o envio e a recepção de dados são feitos pelas chamadas de sistema `write` e `read`, como em arquivos normais[1].

[1] As funções `scanf`, `printf`, `fprintf` e congêneres normalmente usam as chamadas de sistema `read` e `write` em suas implementações.

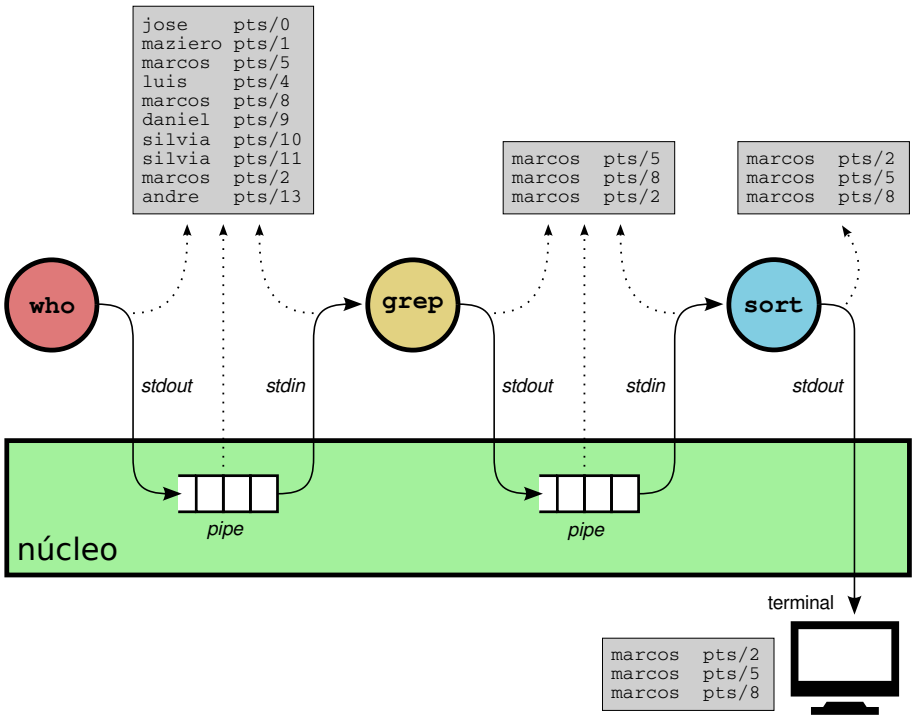


Figura 9.1: Comunicação através de *pipes*.

O uso de pipes na linha de comando UNIX é trivial, mas seu uso na construção de programas é um pouco mais complexo. Vários exemplos do uso de pipes UNIX na construção de programas são apresentados em [Robbins and Robbins, 2003].

Os *pipes* padrão têm vida curta: eles só existem durante a execução da linha de comando ou do processo que os criou, sendo destruídos logo em seguida. Por outro lado, os *pipes nomeados* (*named pipes*, ou *FIFOs*) permanecem desde sua criação até serem explicitamente destruídos ou o sistema ser encerrado. Um *pipe* nomeado é basicamente um *pipe* independente de processos e que tem um nome próprio, para que os processos interessados possam encontrá-lo. Esse nome é baseado na árvore de diretórios do sistema de arquivos, como se fosse um arquivo (mas ele não usa o disco).

Pipes nomeados podem ser criados na linha de comandos em Linux. No Windows, eles podem ser criados dentro de programas. A listagem a seguir apresenta um exemplo de criação, uso e remoção de um *pipe* nomeado usando comandos em Linux:

```
# cria um pipe nomeado, cujo nome é/tmp/pipe
$ mkfifo /tmp/pipe

# mostra o nome do pipe no diretório
$ ls -l /tmp/pipe
prw-rw-r-- 1 maziero maziero 0 sept. 6 18:14 pipe|

# envia dados (saída do comando date) para o pipe nomeado
$ date > /tmp/pipe

# EM OUTRO TERMINAL, recebe dados do pipe nomeado
$ cat < /tmp/pipe
Thu Sep 6 2018, 18:01:50 (UTC+0200)

# remove o pipe nomeado
$ rm /tmp/pipe
```

9.2 Filas de mensagens

As filas de mensagens são um bom exemplo de implementação do conceito de *mailbox* (vide Seção 8.3.6), permitindo o envio e recepção ordenada de mensagens tipadas entre processos em um sistema operacional. As filas de mensagens foram definidas inicialmente na implementação UNIX *System V*, sendo ainda suportadas pela maioria dos sistemas. O padrão *POSIX* também define uma interface para manipulação de filas de mensagens, sendo mais recente e de uso recomendado. Nos sistemas Windows, filas de mensagens podem ser criadas usando o mecanismo de *MailSlots* [Russinovich et al., 2008].

As filas de mensagens são mecanismos de comunicação entre vários processos (N:M ou N:1, dependendo da implementação), confiáveis, orientadas a mensagens e com capacidade finita. As operações de envio e recepção podem ser síncronas ou assíncronas, dependendo da implementação e a critério do programador.

As principais chamadas para usar filas de mensagens POSIX na linguagem C são:

- `mq_open`: abre uma fila já existente ou cria uma nova fila;
- `mq_setattr` e `mq_getattr`: permitem ajustar ou obter atributos (parâmetros) da fila, que definem seu comportamento, como o tamanho máximo da fila, o tamanho de cada mensagem, etc.;
- `mq_send`: envia uma mensagem para a fila; caso a fila esteja cheia, o emissor fica bloqueado até que alguma mensagem seja retirada da fila, abrindo espaço para o envio; a variante `mq_timedsend` permite definir um prazo máximo de espera: caso o envio não ocorra nesse prazo, a chamada retorna com erro;
- `mq_receive`: recebe uma mensagem da fila; caso a fila esteja vazia, o receptor é bloqueado até que surja uma mensagem para ser recebida; a variante `mq_timedreceive` permite definir um prazo máximo de espera;
- `mq_close`: fecha o descritor da fila criado por `mq_open`;

- `mq_unlink`: remove a fila do sistema, destruindo seu conteúdo.

A listagem a seguir implementa um "consumidor de mensagens", ou seja, um programa que cria uma fila para receber mensagens. O código apresentado segue o padrão *POSIX* (exemplos de uso de filas de mensagens no padrão *System V* estão disponíveis em [Robbins and Robbins, 2003]). Para compilá-lo em Linux é necessário efetuar a ligação com a biblioteca de tempo real *POSIX* (usando a opção `-lrt`).

```
1  // Arquivo mq-recv.c: recebe mensagens de uma fila de mensagens POSIX.
2  // Em Linux, compile usando: cc -o mq-recv -lrt mq-recv.c
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <mqueue.h>
7  #include <sys/stat.h>
8
9  #define QUEUE "/my_queue"
10
11 int main (int argc, char *argv[])
12 {
13    mqd_t queue;               // descritor da fila de mensagens
14    struct mq_attr attr;       // atributos da fila de mensagens
15    int  msg ;                 // as mensagens são números inteiros
16
17    // define os atributos da fila de mensagens
18    attr.mq_maxmsg = 10 ;      // capacidade para 10 mensagens
19    attr.mq_msgsize = sizeof(msg) ; // tamanho de cada mensagem
20    attr.mq_flags = 0 ;
21
22    // abre ou cria a fila com permissoes 0666
23    if ((queue = mq_open (QUEUE, O_RDWR|O_CREAT, 0666, &attr)) < 0)
24    {
25       perror ("mq_open");
26       exit (1);
27    }
28
29    // recebe cada mensagem e imprime seu conteudo
30    for (;;)
31    {
32       if ((mq_receive (queue, (void*) &msg, sizeof(msg), 0)) < 0)
33       {
34          perror("mq_receive:") ;
35          exit (1) ;
36       }
37       printf ("Received msg value %d\n", msg);
38    }
39 }
```

A listagem a seguir implementa o programa produtor das mensagens consumidas pelo programa anterior. Vários produtores e consumidores de mensagens podem operar sobre uma mesma fila, mas os produtores de mensagens devem ser lançados após um consumidor, pois é este último quem cria a fila (neste código de exemplo).

```c
// Arquivo mq-send.c: envia mensagens para uma fila de mensagens POSIX.
// Em Linux, compile usando: cc -o mq-send -lrt mq-send.c

#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <unistd.h>

#define QUEUE "/my_queue"

int main (int argc, char *argv[])
{
   mqd_t queue;       // descritor da fila
   int   msg;         // mensagem a enviar

   // abre a fila de mensagens, se existir
   if((queue = mq_open (QUEUE, O_RDWR)) < 0)
   {
      perror ("mq_open");
      exit (1);
   }

   for (;;)
   {
      msg = random() % 100 ; // valor entre 0 e 99

      // envia a mensagem
      if (mq_send (queue, (void*) &msg, sizeof(msg), 0) < 0)
      {
         perror ("mq_send");
         exit (1);
      }
      printf ("Sent message with value %d\n", msg);
      sleep (1) ;
   }
}
```

Deve-se observar que o arquivo /my_queue referenciado em ambas as listagens serve unicamente como identificador comum para a fila de mensagens; nenhum arquivo de dados com esse nome será criado pelo sistema. As mensagens não transitam por arquivos, apenas pela memória do núcleo. Referências de recursos através de nomes de arquivos são frequentemente usadas para identificar vários mecanismos de comunicação e coordenação em UNIX, como filas de mensagens, semáforos e áreas de memória compartilhadas (vide Seção 9.3).

9.3 Memória compartilhada

A comunicação entre tarefas situadas em processos distintos deve ser feita através do núcleo, usando chamadas de sistema. Não existe a possibilidade de acesso a variáveis comuns a ambos, pois suas áreas de memória são distintas e isoladas. A comunicação através do núcleo pode ser ineficiente caso seja frequente e o volume de dados a transferir seja elevado, por causa das trocas de contexto envolvidas nas chamadas de sistema. Para essas situações, seria conveniente ter uma área de memória

comum que possa ser acessada direta e rapidamente pelos processos interessados, sem o custo da intermediação do núcleo.

A maioria dos sistemas operacionais atuais oferece mecanismos para o compartilhamento de áreas de memória entre processos (*shared memory areas*). As áreas de memória compartilhadas e os processos que as acessam são gerenciados pelo núcleo, mas o acesso ao conteúdo de cada área é feito diretamente pelos processos, sem intermediação do núcleo.

A criação e uso de uma área de memória compartilhada entre dois processos p_a e p_b em um sistema UNIX pode ser resumida na seguinte sequência de passos, ilustrada na Figura 9.2:

1. O processo p_a solicita ao núcleo a criação de uma área de memória compartilhada;
2. o núcleo aloca uma nova área de memória e a registra em uma lista de áreas compartilháveis;
3. o núcleo devolve ao processo p_a o identificador (*id*) da área alocada;
4. o processo p_a solicita ao núcleo que a área identificada por *id* seja anexada ao seu espaço de endereçamento;
5. o núcleo modifica a configuração de memória do processo p_a para incluir a área indicada por *id* em seu espaço de endereçamento;
6. o núcleo devolve a p_a um ponteiro para a área alocada;
7. O processo p_b executa os passos 4-6 e também recebe um ponteiro para a área alocada;
8. Os processos p_a e p_b comunicam através de escritas e leituras de valores na área de memória compartilhada.

Deve-se observar que, ao solicitar a criação da área de memória compartilhada, p_a define as permissões de acesso à mesma; por isso, o pedido de anexação da área de memória feito por p_b pode ser recusado pelo núcleo, se violar as permissões definidas por p_a.

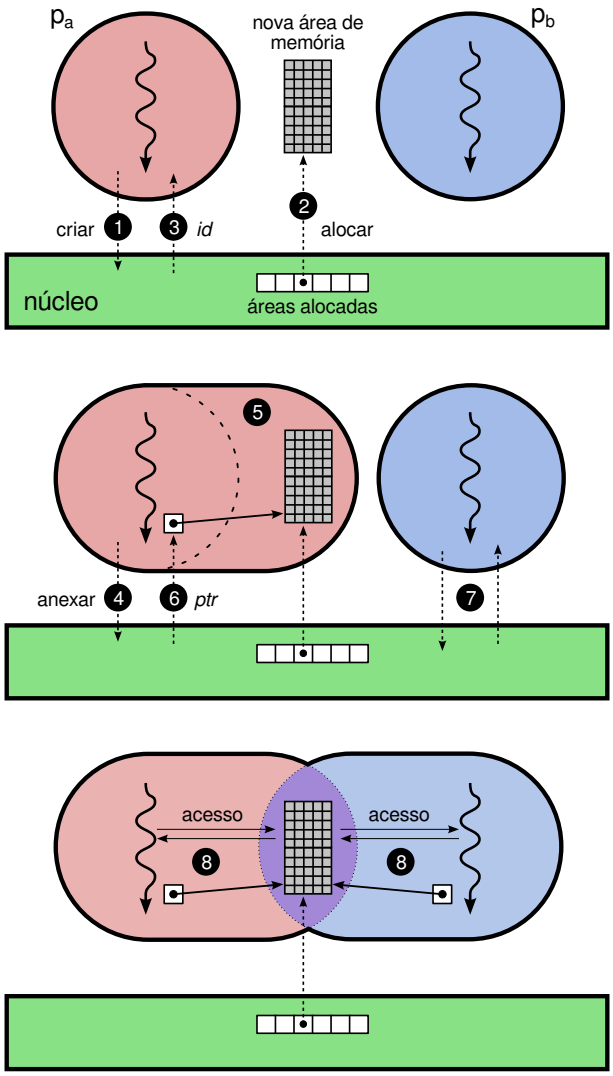


Figura 9.2: Criação e uso de uma área de memória compartilhada.

A Listagem 9.1 exemplifica a criação e uso de uma área de memória compartilhada, usando o padrão POSIX (exemplos de implementação no padrão *System V* podem ser encontrados em [Robbins and Robbins, 2003]). Para compilá-lo em Linux é necessário efetuar a ligação com a biblioteca de tempo real *POSIX*, usando a opção `-lrt`. Para melhor observar seu funcionamento, devem ser lançados dois ou mais processos executando esse código simultaneamente.

Deve-se observar que não existe nenhuma forma de coordenação ou sincronização implícita no acesso à área de memória compartilhada. Assim, dois processos podem escrever sobre os mesmos dados simultaneamente, levando a possíveis inconsistências. Por essa razão, mecanismos de coordenação adicionais (como os apresentados no Capítulo 10) podem ser necessários para garantir a consistência dos dados armazenados em áreas compartilhadas.

Listing 9.1: Memória Compartilhada

```
// Arquivo shmem.c: cria e usa uma área de memória compartilhada POSIX.
// Em Linux, compile usando: cc -o shmem -lrt shmem.c

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

int main (int argc, char *argv[])
{
   int fd, value, *ptr;

   // Passos 1 a 3: abre/cria uma area de memoria compartilhada
   fd = shm_open ("/sharedmem", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
   if (fd == -1) {
      perror ("shm_open");
      exit (1) ;
   }

   // ajusta o tamanho da area compartilhada para sizeof (value)
   if (ftruncate (fd, sizeof (value)) == -1) {
      perror ("ftruncate");
      exit (1) ;
   }

   // Passos 4 a 6: mapeia a area no espaco de enderecamento deste processo
   ptr = mmap (NULL, sizeof(value), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
   if (ptr == MAP_FAILED) {
      perror ("mmap");
      exit (1);
   }

   for (;;) {
      // Passo 8: escreve um valor aleatorio na area compartilhada
      value = random () % 1000 ;
      (*ptr) = value ; // escreve na area
      printf ("Wrote value %i\n", value) ;
      sleep (1);

      // Passo 8: le e imprime o conteudo da area compartilhada
      value = (*ptr) ; // le da area
      printf ("Read value %i\n", value);
      sleep (1) ;
   }
}
```

Referências

J. Hart. *Windows System Programming, 3rd edition*. Addison-Wesley Professional, 2004.

C. Petzold. *Programming Windows, 5th edition*. Microsoft Press, 1998.

K. Robbins and S. Robbins. *UNIX Systems Programming*. Prentice-Hall, 2003.

M. Russinovich, D. Solomon, and A. Ionescu. *Microsoft Windows Internals, Fifth Edition*. Microsoft Press, 2008.

R. Stevens. *UNIX Network Programming*. Prentice-Hall, 1998.

Capítulo 10

Coordenação entre tarefas

Muitas implementações de sistemas complexos são estruturadas como várias tarefas interdependentes, que cooperam entre si para atingir os objetivos da aplicação, como por exemplo em um navegador Web. Para que as várias tarefas que compõem a aplicação possam cooperar, elas precisam comunicar informações umas às outras e coordenar suas atividades, para garantir que os resultados obtidos sejam coerentes. Neste capítulo serão estudados os problemas que podem ocorrer quando duas ou mais tarefas acessam os mesmos recursos de forma concorrente; também serão apresentadas algumas técnicas usadas para coordenar os acessos das tarefas aos recursos compartilhados.

10.1 O problema da concorrência

Quando duas ou mais tarefas acessam simultaneamente um recurso compartilhado, podem ocorrer problemas de consistência dos dados ou do estado do recurso acessado. Esta seção descreve detalhadamente a origem dessas inconsistências, através de um exemplo simples, mas que permite ilustrar claramente o problema.

10.1.1 Uma aplicação concorrente

O código apresentado a seguir implementa de forma simplificada a operação de depósito de um valor em um saldo de conta bancária informado como parâmetro. Para facilitar a compreensão do código de máquina apresentado na sequência, todos os valores manipulados são inteiros.

```
1 void depositar (long * saldo, long valor)
2 {
3    (*saldo) += valor ;
4 }
```

Após a compilação em uma plataforma *Intel* 64 bits, a função `depositar` assume a seguinte forma em *Assembly*:

```
0000000000000000 <depositar>:
       ; inicializa a função
       push  %rbp
       mov   %rsp,%rbp
       mov   %rdi,-0x8(%rbp)
       mov   %esi,-0xc(%rbp)

       ; carrega o conteúdo da memória apontada por "saldo" em EDX
       mov   -0x8(%rbp),%rax       ; saldo → rax (endereço do saldo)
       mov   (%rax),%edx           ; mem[rax] → edx

       ; carrega o conteúdo de "valor" no registrador EAX
       mov   -0xc(%rbp),%eax       ; valor → eax

       ; soma EAX ao valor em EDX
       add   %eax,%edx             ; eax + edx → edx

       ; escreve o resultado em EDX na memória apontada por "saldo"
       mov   -0x8(%rbp),%rax       ; saldo → rax
       mov   %edx,(%rax)           ; edx → mem[rax]

       ; finaliza a função
       nop
       pop   %rbp
       retq
```

Consideremos que a função `depositar` faz parte de um sistema mais amplo de gestão de contas em um banco, que pode ser acessado simultaneamente por centenas ou milhares de usuários em agências e terminais distintos. Caso dois clientes em terminais diferentes tentem depositar valores na mesma conta ao mesmo tempo, existirão duas tarefas t_1 e t_2 acessando os dados da conta de forma concorrente. A Figura 10.1 ilustra esse cenário.

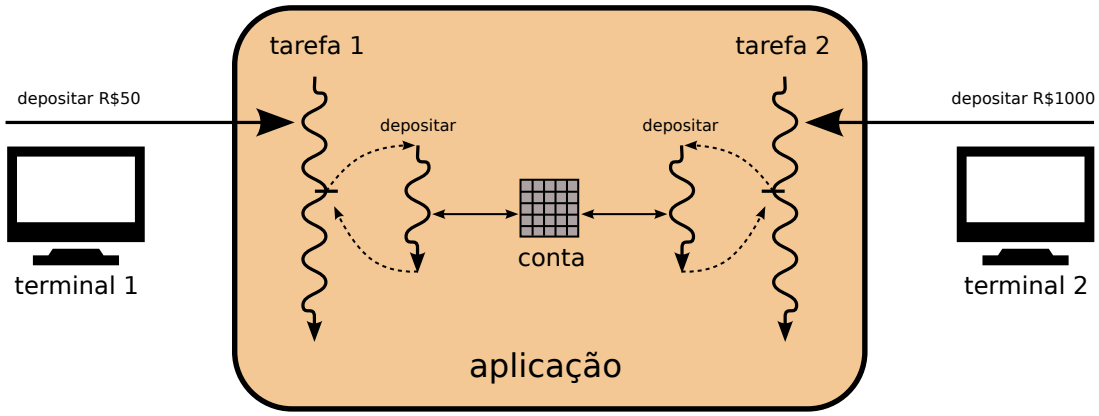


Figura 10.1: Acessos concorrentes a variáveis compartilhadas.

10.1.2 Condições de disputa

O comportamento dinâmico da aplicação da Figura 10.1 pode ser modelado através de diagramas de tempo. Caso o depósito da tarefa t_1 execute integralmente **antes** ou **depois** do depósito efetuado por t_2, teremos os diagramas de tempo da Figura

10.2. Em ambas as execuções o saldo inicial da conta passou de R$ 0,00 para R$ 1.050,00, conforme o esperado.

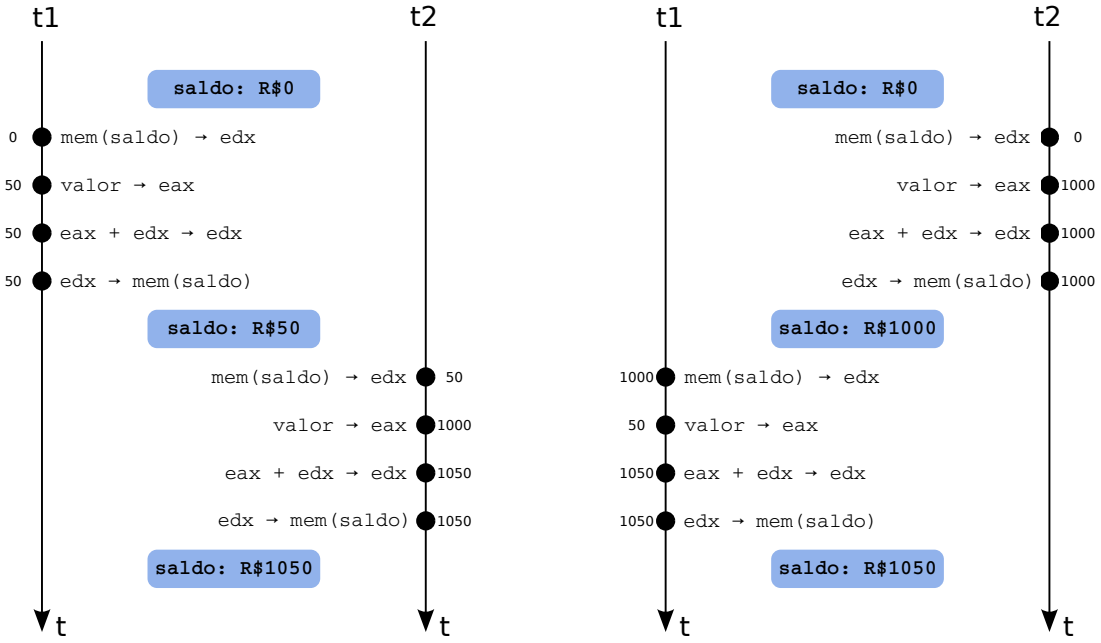


Figura 10.2: Operações de depósitos não-concorrentes.

No entanto, caso as operações de depósito de t_1 e de t_2 se entrelacem, podem ocorrer interferências entre ambas, levando a resultados incorretos. Em sistemas monoprocessados, a sobreposição pode acontecer caso ocorram trocas de contexto durante a execução da função `depositar`. Em sistemas multiprocessados a situação é mais complexa, pois cada tarefa poderá estar executando em um processador distinto.

Os diagramas de tempo apresentados na Figura 10.3 mostram execuções onde houve entrelaçamento das operações de depósito de t_1 e de t_2. Em ambas as execuções o saldo final **não corresponde** ao resultado esperado, pois um dos depósitos é perdido. Pode-se observar que apenas é concretizado o depósito da tarefa que realizou a escrita do resultado na memória por último (operação `edx` → `mem(saldo)`)[1].

Os erros e inconsistências gerados por acessos concorrentes a dados compartilhados, como os ilustrados na Figura 10.3, são denominados **condições de disputa**, ou condições de corrida (do inglês *race conditions*). Condições de disputa podem ocorrer em sistemas onde várias tarefas acessam de forma concorrente recursos compartilhados (variáveis, áreas de memória, arquivos abertos, etc.), sob certas condições.

É importante observar que condições de disputa são erros *dinâmicos*, ou seja, erros que não aparecem no código fonte e que só se manifestam durante a execução. Assim, são dificilmente detectáveis através da simples análise do código fonte. Além disso, erros dessa natureza não se manifestam a cada execução, mas apenas quando certos entrelaçamentos ocorrerem. Assim, uma condição de disputa poderá permanecer latente no código durante anos, ou mesmo nunca se manifestar. A depuração de programas contendo condições de disputa pode ser muito complexa, pois o problema só se manifesta com acessos simultâneos aos mesmos dados, o que pode ocorrer raramente

[1] Não há problema em ambas as tarefas usarem os mesmos registradores, pois os valores de todos os registradores são salvos/restaurados a cada troca de contexto entre tarefas.

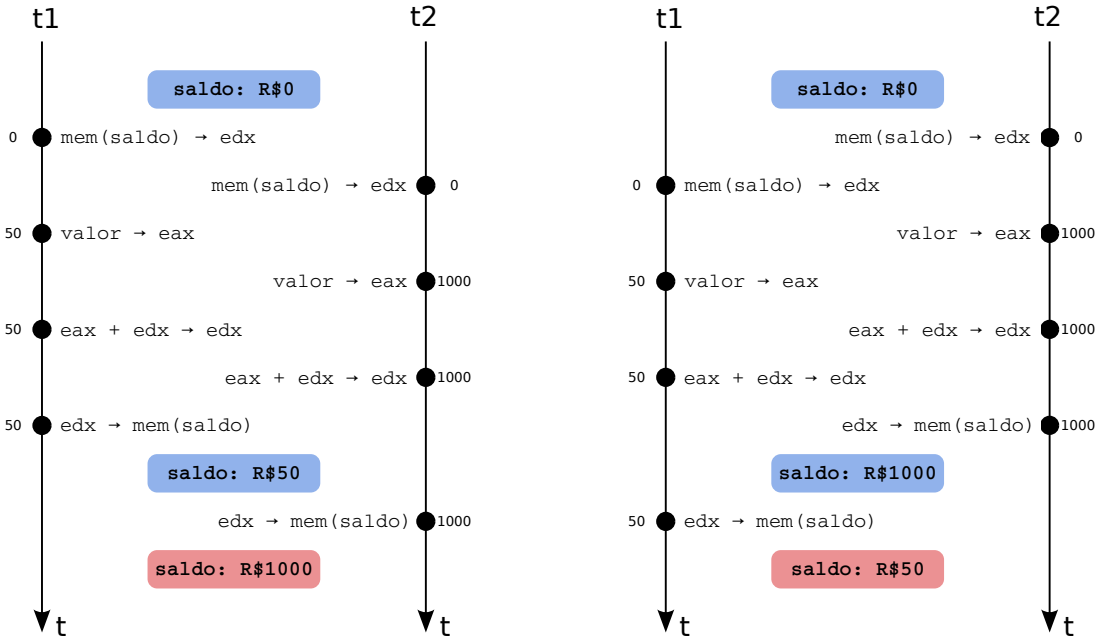


Figura 10.3: Operações de depósito concorrentes.

e ser difícil de reproduzir durante a depuração. Por isso, é importante conhecer técnicas que previnam a ocorrência de condições de disputa.

10.1.3 Condições de Bernstein

Condições de disputa entre tarefas paralelas podem ser formalizadas através das chamadas **condições de Bernstein** [Bernstein, 1966], assim definidas: dadas duas tarefas t_1 e t_2, sendo $\mathcal{R}(t_i)$ o conjunto de variáveis lidas por t_i e $\mathcal{W}(t_i)$ o conjunto de variáveis escritas por t_i, essas tarefas podem executar em paralelo sem risco de condição de disputa ($t_1 \parallel t_2$) se e somente se as seguintes condições forem atendidas:

$$t_1 \parallel t_2 \iff \begin{cases} \mathcal{R}(t_1) \cap \mathcal{W}(t_2) = \varnothing & (t_1 \text{ não lê as variáveis escritas por } t_2) \\ \mathcal{R}(t_2) \cap \mathcal{W}(t_1) = \varnothing & (t_2 \text{ não lê as variáveis escritas por } t_1) \\ \mathcal{W}(t_1) \cap \mathcal{W}(t_2) = \varnothing & (t_1 \text{ e } t_2 \text{ não escrevem nas mesmas variáveis}) \end{cases}$$

Percebe-se claramente que as condições de Bernstein não são respeitadas na aplicação bancária usada como exemplo neste texto, pois ambas as tarefas podem ler e escrever simultaneamente na mesma variável (o saldo). Por isso, elas não devem ser executadas em paralelo.

Outro ponto importante evidenciado pelas condições de Bernstein é que as condições de disputa somente ocorrem se pelo menos uma das operações envolvidas for de escrita; acessos de leitura concorrentes às mesmas variáveis respeitam as condições de Bernstein e portanto não geram condições de disputa entre si.

10.1.4 Seções críticas

Na seção anterior vimos que tarefas acessando dados compartilhados de forma concorrente podem ocasionar condições de disputa. Os trechos de código que acessam dados compartilhados em cada tarefa são denominados **seções críticas** (ou *regiões*

críticas). No caso da Figura 10.1, as seções críticas das tarefas t_1 e t_2 são idênticas e resumidas à seguinte linha de código:

```
1 (*saldo) += valor ;
```

De modo geral, seções críticas são todos os trechos de código que manipulam dados compartilhados onde podem ocorrer condições de disputa. Um programa pode ter várias seções críticas, relacionadas entre si ou não (caso manipulem dados compartilhados distintos). Para assegurar a correção de uma implementação, deve-se impedir o entrelaçamento de seções críticas: dado um conjunto de regiões críticas relacionadas, apenas uma tarefa pode estar em sua seção crítica a cada instante, excluindo o acesso das demais às suas respectivas regiões críticas. Essa propriedade é conhecida como **exclusão mútua**.

10.2 Exclusão mútua

Diversos mecanismos podem ser definidos para garantir a exclusão mútua, impedindo o entrelaçamento de seções críticas. Todos eles exigem que o programador defina os limites (início e o final) de cada seção crítica. Dada uma seção crítica cs_i podem ser definidas as primitivas *enter*(cs_i), para que uma tarefa indique sua intenção de entrar na seção crítica cs_i, e *leave*(cs_i), para que uma tarefa que está na seção crítica cs_i informe que está saindo da mesma. A primitiva *enter*(cs_i) é bloqueante: caso uma tarefa já esteja ocupando a seção crítica cs_i, as demais tarefas que tentarem entrar deverão aguardar até que a primeira libere cs_i através da primitiva *leave*(cs_i).

Usando as primitivas *enter*() e *leave*(), o código da operação de depósito visto na Seção 10.1 pode ser reescrito como segue:

```
1 void depositar (long conta, long *saldo, long valor)
2 {
3    enter (conta) ;          // entra na seção crítica "conta"
4    (*saldo) += valor ;      // usa as variáveis compartilhadas
5    leave (conta) ;          // sai da seção crítica
6 }
```

Nesta seção serão apresentadas algumas soluções para a implementação das primitivas de exclusão mútua. As soluções propostas devem atender a alguns critérios básicos enumerados a seguir:

Exclusão mútua: somente uma tarefa pode estar dentro da seção crítica em cada instante.

Espera limitada: uma tarefa que aguarda acesso a uma seção crítica deve ter esse acesso garantido em um tempo finito, ou seja, não pode haver inanição.

Independência de outras tarefas: a decisão sobre o uso de uma seção crítica deve depender somente das tarefas que estão tentando entrar na mesma. Outras tarefas, que no momento não estejam interessadas em entrar na região crítica, não podem influenciar sobre essa decisão.

Independência de fatores físicos: a solução deve ser puramente lógica e não depender da velocidade de execução das tarefas, de temporizações, do número de processadores no sistema ou de outros fatores físicos.

10.2.1 Inibição de interrupções

Uma solução simples para a implementação da exclusão mútua consiste em impedir as trocas de contexto dentro da seção crítica. Ao entrar em uma seção crítica, a tarefa desativa as interrupções que possam provocar trocas de contexto, e as reativa ao sair da seção crítica. Apesar de simples, essa solução raramente é usada para a construção de aplicações devido a vários problemas:

- Ao desligar as interrupções, a preempção por tempo ou por recursos deixa de funcionar; caso a tarefa entre em um laço infinito dentro da seção crítica, o sistema inteiro será bloqueado. Assim, uma tarefa mal intencionada poderia desativar as interrupções e travar o sistema.
- Enquanto as interrupções estão desativadas, os dispositivos de entrada/saída deixam de ser atendidos pelo núcleo, o que pode causar perdas de dados ou outros problemas. Por exemplo, uma placa de rede pode perder novos pacotes se seus *buffers* estiverem cheios e não forem tratados pelo núcleo em tempo hábil.
- A tarefa que está na seção crítica não pode realizar operações de entrada/saída, pois os dispositivos não irão responder.
- Esta solução só funciona em sistemas monoprocessados; em uma máquina multiprocessada ou multicore, duas tarefas concorrentes podem executar simultaneamente em processadores separados, acessando a seção crítica ao mesmo tempo.

Devido a esses problemas, a inibição de interrupções é uma operação privilegiada e somente utilizada em algumas seções críticas dentro do núcleo do sistema operacional e nunca pelas aplicações.

10.2.2 A solução trivial

Uma solução trivial para o problema da seção crítica consiste em usar uma variável *busy* para indicar se a seção crítica está livre ou ocupada. Usando essa abordagem, a implementação das primitivas *enter*() e *leave*() poderia ser escrita assim:

```
1  int busy = 0 ;          // a seção está inicialmente livre
2
3  void enter ()
4  {
5     while (busy) ;       // espera enquanto a seção estiver ocupada
6     busy = 1 ;           // marca a seção como ocupada
7  }
8
9  void leave ()
10 {
11    busy = 0 ;           // libera a seção (marca como livre)
12 }
```

Infelizmente, essa solução simples **não funciona**! Seu grande defeito é que o teste da variável *busy* (na linha 5) e sua atribuição (na linha 6) são feitos em momentos distintos; caso ocorra uma troca de contexto entre as linhas 5 e 6 do código, poderá ocorrer

uma condição de disputa envolvendo a variável *busy*, que terá como consequência a violação da exclusão mútua: duas ou mais tarefas poderão entrar simultaneamente na seção crítica (conforme demonstra o diagrama de tempo da Figura 10.4). Em outras palavras, as linhas 5 e 6 da implementação formam uma seção crítica que também deve ser protegida.

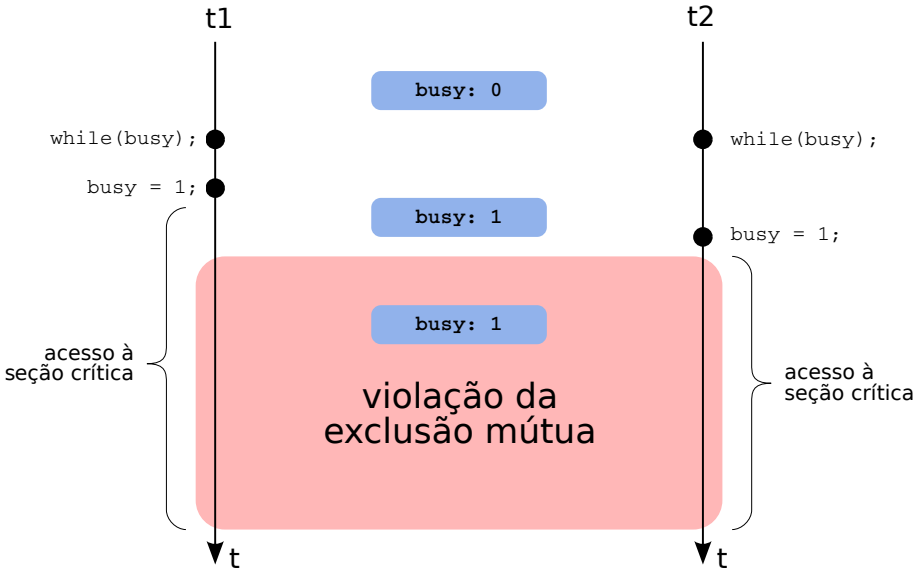


Figura 10.4: Condição de disputa no acesso à variável *busy*.

Outro problema importante com essa solução ocorre no laço da linha 5 do código: o teste contínuo da variável *busy* consome muito processador. Se houverem muitas tarefas tentando entrar em uma seção crítica, muito tempo de processamento será gasto nesse teste. O teste contínuo de uma condição é denominado **espera ocupada** (*busy wait*) e deve ser evitado, por conta de sua ineficiência.

10.2.3 Alternância de uso

Outra solução simples para a implementação da exclusão mútua consiste em definir uma variável *turno*, que indica de quem é a vez de entrar na seção crítica. Essa variável deve ser ajustada cada vez que uma tarefa sai da seção crítica, para indicar a próxima tarefa a usá-la. A implementação das duas primitivas fica assim:

```
1  int num_tasks ;
2  int turn = 0 ;                    // inicia pela tarefa 0
3
4  void enter (int task)             // task vale 0, 1, ..., num_tasks-1
5  {
6     while (turn != task) ;         // a tarefa espera seu turno
7  }
8
9  void leave (int task)
10 {
11    turn = (turn + 1) % num_tasks ; // passa para a próxima tarefa
12 }
```

Nessa solução, cada tarefa aguarda seu turno de usar a seção crítica, em uma sequência circular: $t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_{n-1} \rightarrow t_0$. Essa abordagem garante a exclusão

mútua entre as tarefas e independe de fatores externos, mas não atende os demais critérios: caso uma tarefa t_i não deseje usar a seção crítica, todas as tarefas t_j com $j > i$ ficarão impedidas de fazê-lo, pois a variável *turno* não irá evoluir.

10.2.4 O algoritmo de Peterson

Uma solução correta para a exclusão mútua no acesso a uma seção crítica por duas tarefas foi proposta inicialmente por Dekker em 1965. Em 1981, Gary Peterson propôs uma solução mais simples e elegante para o mesmo problema [Raynal, 1986]. O algoritmo de Peterson pode ser resumido no código a seguir:

```
1  int turn = 0 ;           // indica de quem éa vez
2  int wants[2] = {0, 0} ; // indica se a tarefa i quer acessar a seção crítica
3
4  void enter (int task)   // task pode valer 0 ou 1
5  {
6     int other = 1 - task ; // indica a outra tarefa
7     wants[task] = 1 ;     // task quer acessar a seção crítica
8     turn = task ;
9     while ((turn == task) && wants[other]) ; // espera ocupada
10 }
11
12 void leave (int task)
13 {
14   wants[task] = 0 ;      // task libera a seção crítica
15 }
```

Os algoritmos de Dekker e de Peterson foram desenvolvidos para garantir a exclusão mútua entre **duas tarefas**, garantindo também o critério de espera limitada[2]. Diversas generalizações para $n > 2$ tarefas podem ser encontradas na literatura [Raynal, 1986], sendo a mais conhecida delas o *algoritmo do padeiro*, proposto por Leslie Lamport [Lamport, 1974].

10.2.5 Operações atômicas

O uso de uma variável *busy* para controlar a entrada em uma seção crítica é uma ideia interessante, que infelizmente não funciona porque o teste da variável *busy* e sua atribuição são feitos em momentos distintos do código, permitindo condições de disputa. Para resolver esse problema, projetistas de hardware criaram instruções em código de máquina que permitem testar e atribuir um valor a uma variável de forma *atômica* (indivisível, sem possibilidade de troca de contexto entre essas duas operações). A execução atômica das operações de teste e atribuição impede a ocorrência de condições de disputa sobre a variável *busy*.

Um exemplo de operação atômica simples é a instrução de máquina *Test-and-Set Lock* (TSL), que é executada atomicamente pelo processador e cujo comportamento é descrito pelo seguinte pseudocódigo:

[2] Este algoritmo **pode falhar** em arquiteturas que permitam execução fora de ordem, ou seja, onde a ordem das operações de leitura e de escrita na memória possa ser trocada pelo processador para obter mais desempenho, como é o caso dos processadores Intel x86. Nesse caso, é necessário incluir uma instrução de barreira de memória logo antes do laço `while`.

$$TSL(x) = \begin{cases} x \rightarrow old & \text{// guarda o valor de } x \\ 1 \rightarrow x & \text{// atribui 1 a } x \\ return(old) & \text{// devolve o valor anterior de } x \end{cases}$$

A implementação das primitivas *enter*() e *leave*() usando a instrução TSL assume a seguinte forma:

```
1  int lock = 0 ;            // variável de trava
2
3  void enter (int *lock)    // passa o endereço da trava
4  {
5     while ( TSL (*lock) ) ; // espera ocupada sobre a trava
6  }
7
8  void leave (int *lock)
9  {
10   (*lock) = 0 ;           // libera a seção crítica
11 }
```

A instrução TSL esteve disponível apenas em processadores antigos, como o IBM System/360. Processadores modernos oferecem diversas operações atômicas com o mesmo objetivo, conhecidas coletivamente como **instruções RMW** (de *Read-Modify-Write*, Lê-Modifica-Escreve), como CAS (*Compare-And-Swap*) e XCHG (*Exchange*). A instrução XCHG, disponível nos processadores Intel e AMD, efetua a troca atômica de conteúdo (*swapping*) entre dois registradores, ou entre um registrador e uma posição de memória:

$$XCHG\ op_1, op_2 :\ op_1 \rightleftarrows op_2$$

A implementação das primitivas *enter*() e *leave*() usando a instrução XCHG é um pouco mais complexa:

```
1  int lock ;                // variável de trava
2
3  enter (int *lock)
4  {
5     int key = 1 ;          // variável auxiliar (local)
6     while (key)            // espera ocupada
7       XCHG (lock, &key) ;  // alterna valores de lock e key
8  }
9
10 leave (int *lock)
11 {
12    (*lock) = 0 ;          // libera a seção crítica
13 }
```

Os mecanismos de exclusão mútua usando instruções atômicas são amplamente usados no interior do sistema operacional, para controlar o acesso a seções críticas dentro do núcleo, como descritores de tarefas, *buffers* de arquivos ou de conexões de rede, etc. Nesse contexto, eles são muitas vezes denominados *spinlocks*. Todavia, mecanismos de espera ocupada são inadequados para a construção de aplicações de usuário, como será visto na próxima seção.

10.3 Problemas

O acesso concorrente de diversas tarefas aos mesmos recursos pode provocar problemas de consistência, as chamadas *condições de disputa*. Uma forma de eliminar esses problemas é forçar o acesso a esses recursos em exclusão mútua, ou seja, uma tarefa por vez. Neste capítulo foram apresentadas algumas formas de implementar a exclusão mútua. Contudo, apesar dessas soluções garantirem a exclusão mútua (com exceção da solução trivial), elas sofrem de problemas que impedem seu uso em larga escala nas aplicações de usuário:

Ineficiência: as tarefas que aguardam o acesso a uma seção crítica ficam testando continuamente uma condição, consumindo tempo de processador sem necessidade. O procedimento adequado seria suspender essas tarefas até que a seção crítica solicitada seja liberada.

Injustiça: a não ser na solução de alternância, não há garantia de ordem no acesso à seção crítica; dependendo da duração de *quantum* e da política de escalonamento, uma tarefa pode entrar e sair da seção crítica várias vezes, antes que outras tarefas consigam acessá-la.

Dependência: na solução por alternância, tarefas desejando acessar a seção crítica podem ser impedidas de fazê-lo por tarefas que não têm interessa na seção crítica naquele momento.

Por estas razões, as soluções com espera ocupada são pouco usadas na construção de aplicações. Seu maior uso se encontra na programação de estruturas de controle de concorrência dentro do núcleo do sistema operacional e na construção de sistemas de computação dedicados, como controladores embarcados mais simples. O próximo capítulo apresentará estruturas de controle de sincronização mais sofisticadas, que resolvem os problemas indicados acima.

Referências

A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, Oct 1966. ISSN 0367-7508.

L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, 1986.

Capítulo 11

Mecanismos de coordenação

O capítulo anterior descreveu o problema das condições de disputa entre tarefas concorrentes e a necessidade de executar operações em exclusão mútua para evitá-las. No entanto, as soluções apresentadas não são adequadas para a construção de aplicações, devido à ineficiência e à falta de justiça na distribuição do acesso à seção crítica pelas tarefas.

Este capítulo apresenta mecanismos de sincronização mais sofisticados, como os semáforos e *mutexes*, que atendem os requisitos de eficiência e justiça. Esses mecanismos são amplamente usados na construção de aplicações concorrentes.

11.1 Semáforos

Em 1965, o matemático holandês E. Dijkstra propôs um mecanismo de coordenação eficiente e flexível para o controle da exclusão mútua entre n tarefas: o **semáforo** [Raynal, 1986]. Apesar de antigo, o semáforo continua sendo o mecanismo de sincronização mais utilizado na construção de aplicações concorrentes, sendo usado de forma explícita ou como base na construção de mecanismos de coordenação mais abstratos, como os monitores.

Um semáforo pode ser visto como uma variável s, que representa uma determinada seção crítica e cujo conteúdo interno não é acessível ao programador. Internamente, cada semáforo contém um contador inteiro $s.counter$ e uma fila de tarefas $s.queue$, inicialmente vazia. As tarefas podem invocar **operações atômicas** sobre os semáforos, descritas a seguir:

down(s)**:** solicita acesso à seção crítica associada ao semáforo s, equivalendo à primitiva *enter*() discutida na Seção 10.2. Caso a seção crítica esteja livre, a chamada retorna imediatamente e a tarefa continua sua execução, entrando na seção crítica. Caso contrário, a tarefa solicitante é suspensa e adicionada à fila do semáforo[1]; o contador associado ao semáforo é decrementado. Dijkstra denominou essa operação $P(s)$ (do holandês *probeer*, que significa *tentar*).

up(s)**:** libera a seção crítica associada ao semáforo s, de forma similar à primitiva *leave*(). O contador associado ao semáforo é incrementado; caso a fila do semáforo não esteja vazia, a primeira tarefa da fila é acordada, sai da fila do semáforo

[1] Alguns sistemas implementam também a chamada *try_down*(s), cuja semântica é não-bloqueante: caso o semáforo esteja ocupado, a chamada retorna imediatamente com um código de erro.

e volta à fila de tarefas prontas para retomar sua execução. Essa operação foi inicialmente denominada $V(s)$ (do holandês *verhoog*, que significa *incrementar*). Deve-se observar que esta chamada não é bloqueante: a tarefa não precisa ser suspensa ao executá-la.

As operações $down(s)$ e $up(s)$ estão especificadas no Algoritmo 1.

Algoritmo 1 Operações sobre semáforos

Require: as operações devem executar atomicamente

t: tarefa que invocou a operação
s: semáforo, contendo um contador e uma fila

```
 1: procedure DOWN(t, s)
 2:     s.counter ← s.counter − 1
 3:     if s.counter < 0 then
 4:         append (t, s.queue)              ▷ põe t no final de s.queue
 5:         suspend (t)                      ▷ a tarefa t perde o processador
 6:     end if
 7: end procedure

 8: procedure UP(s)
 9:     s.counter ← s.counter + 1
10:     if s.counter ≤ 0 then
11:         u = first (s.queue)              ▷ retira a primeira tarefa de s.queue
12:         awake(u)                         ▷ devolve u à fila de tarefas prontas
13:     end if
14: end procedure
```

As operações de acesso aos semáforos são geralmente implementadas pelo núcleo do sistema operacional, como chamadas de sistema. É importante observar que a execução dessas operações deve ser **atômica**, para evitar condições de disputa sobre as variáveis internas do semáforo. Para garantir a atomicidade dessas operações em um sistema monoprocessador, seria suficiente inibir as interrupções durante a execução das mesmas; no caso de sistemas multiprocessados, devem ser usados outros mecanismos de controle de concorrência, como as operações atômicas estudadas na Seção 10.2.5, para proteger a integridade do semáforo. Neste caso, a espera ocupada não constitui um problema, pois a execução dessas operações é muito rápida.

Usando semáforos, o código de depósito em conta bancária apresentado na Seção 10.1 poderia ser reescrito da seguinte forma:

```
1 // s: semáforo associado à conta
2
3 void depositar (semaphore s, int *saldo, int valor)
4 {
5    down (s) ;              // solicita acesso a conta
6    (*saldo) += valor ;     // seção crítica
7    up (s) ;                // libera o acesso a conta
8 }
```

Por sua forma de funcionamento, os semáforos resolvem os problemas encontrados nas soluções vistas no Capítulo 10:

Eficiência: as tarefas que aguardam o semáforos são suspensas e não consomem processador; quando o semáforo é liberado, somente a primeira tarefa da fila de semáforos é acordada.

Justiça: a fila de tarefas do semáforo obedece uma política FIFO, garantindo que as tarefas receberão o semáforo na ordem das solicitações[2].

Independência: somente as tarefas que solicitaram o semáforo através da operação *down*(*s*) são consideradas na decisão de quem irá acessá-lo.

O semáforo é um mecanismo de sincronização muito poderoso, seu uso vai muito além de controlar a exclusão mútua no acesso a seções críticas. Por exemplo, o valor inteiro associado ao semáforo funciona como um contador de recursos: caso seja positivo, indica quantas instâncias daquele recurso estão disponíveis. Caso seja negativo, indica quantas tarefas estão aguardando aquele recurso. Seu valor inicial permite expressar diferentes situações de sincronização, como será visto no Capítulo 12.

Um semáforo pode ser usado, por exemplo, para gerenciar a entrada de veículos em um estacionamento controlado por cancelas. O valor inicial do semáforo representa o número de total de vagas no estacionamento. Quando um carro deseja entrar no estacionamento, ele solicita uma vaga; enquanto o semáforo for positivo não haverão bloqueios, pois há vagas livres. Caso não existam mais vagas livres, o carro ficará aguardando o semáforo até que uma vaga seja liberada, o que ocorre quando outro carro sair do estacionamento. A listagem a seguir representa o princípio de funcionamento dessa solução. Observa-se que essa solução funciona para um número qualquer de cancelas de entrada e de saída do estacionamento.

```
1  semaphore vagas = 100 ;     // estacionamento tem 100 vagas
2
3  // cancela de entrada invoca esta operacao para cada carro
4  void obtem_vaga()
5  {
6     down (vagas) ;           // solicita uma vaga
7  }
8
9  // cancela de saída invoca esta operacao para cada carro
10 void libera_vaga ()
11 {
12    up (vagas) ;             // libera uma vaga
13 }
```

Semáforos estão disponíveis na maioria dos sistemas operacionais e linguagens de programação. O padrão POSIX define várias chamadas para a criação e manipulação de semáforos, sendo estas as mais frequentemente utilizadas:

[2] Algumas implementações de semáforos acordam uma tarefa aleatória da fila, não necessariamente a primeira tarefa. Essas implementações são chamadas de *semáforos fracos*, por não garantirem a justiça no acesso à seção crítica nem a ausência de inanição (*starvation*) de tarefas.

```c
#include <semaphore.h>

// inicializa um semáforo apontado por "sem", com valor inicial "value"
int sem_init (sem_t *sem, int pshared, unsigned int value);

// Operação up(s)
int sem_post (sem_t *sem);

// Operação down(s)
int sem_wait (sem_t *sem);

// Operação try_down(s), retorna erro se o semáforo estiver ocupado
int sem_trywait (sem_t *sem);
```

11.2 Mutexes

Muitos ambientes de programação, bibliotecas de threads e até mesmo núcleos de sistema proveem uma versão simplificada de semáforos, na qual o contador só assume dois valores possíveis: *livre* ou *ocupado*. Esses semáforos simplificados são chamados de **mutexes** (uma abreviação de *mutual exclusion*), semáforos binários ou simplesmente **locks** (travas). Algumas das funções definidas pelo padrão POSIX [Gallmeister, 1994; Barney, 2005] para criar e usar *mutexes* são:

```c
#include <pthread.h>

// inicializa uma variável do tipo mutex, usando um struct de atributos
int pthread_mutex_init (pthread_mutex_t *restrict mutex,
                        const pthread_mutexattr_t *restrict attr);

// destrói uma variável do tipo mutex
int pthread_mutex_destroy (pthread_mutex_t *mutex);

// solicita acesso à seção crítica protegida pelo mutex;
// se a seção estiver ocupada, bloqueia a tarefa
int pthread_mutex_lock (pthread_mutex_t *mutex);

// solicita acesso à seção crítica protegida pelo mutex;
// se a seção estiver ocupada, retorna com status de erro
int pthread_mutex_trylock (pthread_mutex_t *mutex);

// libera o acesso à seção crítica protegida pelo mutex
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

Os sistemas Windows oferecem chamadas em C/C++ para gerenciar *mutexes*, como `CreateMutex`, `WaitForSingleObject` e `ReleaseMutex`. *Mutexes* estão disponíveis na maior parte das linguagens de programação de uso geral, como C, C++, Python, Java, C#, etc.

11.3 Variáveis de condição

Outro mecanismo de sincronização de uso frequente são as *variáveis de condição*, ou variáveis condicionais. Uma variável de condição está associada a uma condição lógica que pode ser aguardada por uma tarefa, como a conclusão de uma operação, a chegada de um pacote de rede ou o preenchimento de um *buffer*. Quando uma tarefa aguarda uma condição, ela é colocada para dormir até que outra tarefa a avise que aquela condição se tornou verdadeira. Assim, a tarefa não precisa testar continuamente uma condição, evitando esperas ocupadas.

O uso de variáveis de condição é simples: a condição desejada é associada a uma variável de condição c. Uma tarefa aguarda essa condição através do operador *wait*(c), ficando suspensa enquanto espera. A tarefa em espera será acordada quando outra tarefa sinalizar que a condição se tornou verdadeira, através do operador *signal*(c) (ou *notify*(c)).

Internamente, uma variável de condição possui uma fila de tarefas *c.queue* que aguardam a condição. Além disso, a variável de condição deve ser usada em conjunto com um *mutex*, para garantir a exclusão mútua sobre o estado da condição representada por c.

O Algoritmo 2 descreve o funcionamento das operações *wait*, *signal* e *broadcast* (que sinaliza todas as tarefas que estão aguardando a condição c). Assim como os operadores sobre semáforos, os operadores sobre variáveis de condição também devem ser executados de forma atômica.

Algoritmo 2 Operadores sobre variáveis de condição

Require: as operações devem executar atomicamente

```
t: tarefa que invocou a operação
c: variável de condição
m: mutex associado à condição

procedure WAIT(t, c, m)
    append (t, c.queue)          ▷ põe t no final de c.queue
    unlock (m)                   ▷ libera o mutex
    suspend (t)                  ▷ a tarefa t é suspensa
    lock (m)                     ▷ ao acordar, requer o mutex
end procedure

procedure SIGNAL(c)
    u = first (c.queue)          ▷ retira a primeira tarefa de c.queue
    awake(u)                     ▷ devolve u à fila de tarefas prontas
end procedure

procedure BROADCAST(c)
    while c.queue ≠ ∅ do         ▷ acorda todas as tarefas de c.queue
        u = first (c.queue)
        awake (u)
    end while
end procedure
```

Deve-se terem mente que a variável de condição **não contém** a condição propriamente dita, apenas permite efetuar a sincronização sobre essa condição. Por exemplo, se em um dado programa a condição a testar for um *buffer* ficar vazio (`buffer==0`), a variável de condição apenas permite esperar que essa condição seja verdadeira e sinalizar quando isso ocorre. As operações sobre o buffer (`buffer++`, etc) e os testes (`if (buffer == 0) {...}`) devem ser feitas pelo próprio programa.

No exemplo a seguir, a tarefa `produce_data` obtém dados de alguma fonte (rede, disco, etc) e os deposita em um *buffer* compartilhado. Enquanto isso, a tarefa `consume_data` aguarda por novos dados nesse *buffer* para consumi-los. Uma variável de condição é usada para a tarefa produtora sinalizar a presença de novos dados no *buffer*. Por sua vez, o *mutex* protege o *buffer* de condições de disputa.

```
1  condition c ;
2  mutex m ;
3
4  task produce_data ()
5  {
6    while (1)
7    {
8      // obtem dados de alguma fonte (rede, disco, etc)
9      retrieve_data (data) ;
10
11     // insere dados no buffer
12     lock (m) ;                      // acesso exclusivo ao buffer
13     put_data (buffer, data) ;       // poe dados no buffer
14     signal (c) ;                    // sinaliza que o buffer tem dados
15     unlock (m) ;                    // libera o buffer
16   }
17 }
18
19 task consume_data ()
20 {
21   while (1)
22   {
23     // aguarda presença de dados no buffer
24     lock (m) ;                      // acesso exclusivo ao buffer
25     while (buffer.size == 0)        // enquanto buffer estiver vazio
26       wait (c, m) ;                 // aguarda a condição
27
28     // retira os dados do buffer e o libera
29     get_data (buffer, data) ;
30     unlock (m) ;
31
32     // trata os dados recebidos
33     process_data (data) ;
34   }
35 }
```

É importante observar que na definição original de variáveis de condição, a operação *signal(c)* fazia com que a tarefa sinalizadora perdesse imediatamente o *mutex* e o processador, que eram entregues à primeira tarefa da fila de *c*. Esse comportamento, conhecido como *semântica de Hoare* [Lampson and Redell, 1980], interfere diretamente no escalonador de processos, sendo indesejável em sistemas operacionais de uso geral.

As implementações modernas de variáveis de condição adotam outro comportamento, denominado *semântica Mesa*, que foi inicialmente proposto na linguagem programação concorrente *Mesa*. Nessa semântica, a operação *signal(c)* apenas "acorda" uma tarefa que espera pela condição, sem suspender a execução da tarefa corrente. Cabe ao programador garantir que a tarefa corrente vai liberar o *mutex* logo em seguida e que não vai alterar a condição representada pela variável de condição.

As variáveis de condição estão presentes no padrão POSIX, através de operadores como `pthread_cond_wait`, `pthread_cond_signal` e `pthread_cond_broadcast`. O padrão POSIX adota a semântica *Mesa*.

11.4 Monitores

Ao usar semáforos ou *mutexes*, um programador precisa identificar explicitamente os pontos de sincronização necessários em seu programa. Essa abordagem é eficaz para programas pequenos e problemas de sincronização simples, mas se torna inviável e suscetível a erros em sistemas mais complexos. Por exemplo, se o programador esquecer de liberar um semáforo previamente alocado, o programa pode entrar em um impasse (vide Seção 13). Por outro lado, se ele esquecer de requisitar um semáforo, a exclusão mútua sobre um recurso pode ser violada.

Em 1972, os cientistas Per Brinch Hansen e Charles Hoare definiram o conceito de *monitor* [Lampson and Redell, 1980]. Um monitor é uma estrutura de sincronização que requisita e libera a seção crítica associada a um recurso de forma transparente, sem que o programador tenha de se preocupar com isso. Um monitor consiste dos seguintes elementos:

- um recurso compartilhado, visto como um conjunto de variáveis internas ao monitor.
- um conjunto de procedimentos e funções que permitem o acesso a essas variáveis;
- um *mutex* ou semáforo para controle de exclusão mútua; cada procedimento de acesso ao recurso deve obter o *mutex* antes de iniciar e liberá-lo ao concluir;
- um invariante sobre o estado interno do recurso.

O pseudocódigo a seguir define um monitor para operações sobre uma conta bancária (observe sua semelhança com a definição de uma classe em programação orientada a objetos). Esse exemplo está também ilustrado na Figura 11.1.

```
1  monitor conta
2  {
3     string numero ;
4     float saldo = 0.0 ;
5     float limite ;
6
7     void depositar (float valor)
8     {
9        if (valor >= 0)
10          conta->saldo += valor ;
11       else
12          error ("erro: valor negativo\n") ;
13    }
14
15    void retirar (float saldo)
16    {
17       if (valor >= 0)
18          conta->saldo -= valor ;
19       else
20          error ("erro: valor negativo\n") ;
21    }
22 }
```

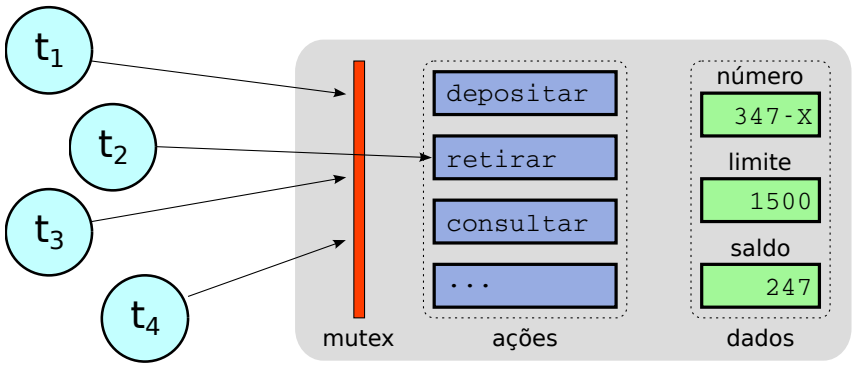


Figura 11.1: Estrutura básica de um monitor de sincronização.

A definição formal de monitor prevê e existência de um *invariante*, ou seja, uma condição sobre as variáveis internas do monitor que deve ser sempre verdadeira. No caso da conta bancária, esse invariante poderia ser o seguinte: "*O saldo atual deve ser a soma de todos os depósitos efetuados menos todas as retiradas efetuadas*". Entretanto, a maioria das implementações de monitor não suporta a definição de invariantes (com exceção da linguagem Eiffel).

De certa forma, um monitor pode ser visto como um objeto que encapsula o recurso compartilhado, com procedimentos (métodos) para acessá-lo. No monitor, a execução dos procedimentos é feita com exclusão mútua entre eles. As operações de obtenção e liberação do *mutex* são inseridas automaticamente pelo compilador do programa em todos os pontos de entrada e saída do monitor (no início e final de cada procedimento), liberando o programador dessa tarefa e assim evitando erros.

Monitores estão presentes em várias linguagens, como Ada, C#, Eiffel, Java e Modula-3. Em Java, a cláusula `synchronized` faz com que um semáforo seja associado aos métodos de um objeto (ou de uma classe, se forem métodos de classe). O código a

seguir mostra um exemplo simplificado de uso de monitor em Java, no qual apenas um depósito ou retirada de cada vez poderá ser feito sobre cada objeto da classe `Conta`.

```
1  class Conta
2  {
3     private float saldo = 0;
4
5     public synchronized void depositar (float valor)
6     {
7        if (valor >= 0)
8           saldo += valor ;
9        else
10          System.err.println("valor negativo");
11    }
12
13    public synchronized void retirar (float valor)
14    {
15       if (valor >= 0)
16          saldo -= valor ;
17       else
18          System.err.println("valor negativo");
19    }
20 }
```

Variáveis de condição podem ser usadas no interior de monitores (na verdade, os dois conceitos nasceram juntos). Todavia, devido às restrições da semântica Mesa, um procedimento que executa a operação *signal* em uma variável de condição deve concluir e sair imediatamente do monitor, para garantir que o invariante associado ao estado interno do monitor seja respeitado [Birrell, 2004].

Referências

B. Barney. POSIX threads programming. http://www.llnl.gov/computing/tutorials/pthreads, 2005.

A. Birrell. Implementing condition variables with semaphores. *Computer Systems Theory, Technology, and Applications*, pages 29–37, December 2004.

B. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly Media, Inc, 1994.

B. Lampson and D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, February 1980.

M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, 1986.

Capítulo 12

Problemas clássicos

Algumas situações de coordenação entre atividades ocorrem com muita frequência na programação de sistemas complexos. Os *problemas clássicos de coordenação* retratam muitas dessas situações e permitem compreender como podem ser implementadas suas soluções.

Este capítulo apresenta alguns problemas clássicos: o problema dos *produtores/consumidores*, o problema dos *leitores/escritores* e o *jantar dos filósofos*. Diversos outros problemas clássicos são frequentemente descritos na literatura, como o *problema dos fumantes* e o do *barbeiro dorminhoco*, entre outros [Raynal, 1986; Ben-Ari, 1990]. Uma extensa coletânea de problemas de coordenação e suas soluções é apresentada em [Downey, 2016], disponível *online*.

12.1 Produtores/consumidores

Este problema também é conhecido como o problema do *buffer limitado*, e consiste em coordenar o acesso de tarefas (processos ou threads) a um *buffer* compartilhado com capacidade de armazenamento limitada a N itens (que podem ser inteiros, registros, mensagens, etc.). São considerados dois tipos de processos com comportamentos cíclicos e simétricos:

Produtor: produz e deposita um item no *buffer*, caso o mesmo tenha uma vaga livre. Caso contrário, deve esperar até que surja uma vaga. Ao depositar um item, o produtor "consome" uma vaga livre.

Consumidor: retira um item do *buffer* e o consome; caso o *buffer* esteja vazio, aguarda que novos itens sejam depositados pelos produtores. Ao consumir um item, o consumidor "produz" uma vaga livre no *buffer*.

Deve-se observar que o acesso ao *buffer* é bloqueante, ou seja, cada processo fica bloqueado até conseguir fazer seu acesso, seja para produzir ou para consumir um item. A Figura 12.1 ilustra esse problema, envolvendo vários produtores e consumidores acessando um *buffer* com capacidade para 9 ítens. É interessante observar a forte similaridade dessa figura com o *Mailbox* da Figura 8.10; na prática, a implementação de *mailboxes* e de *pipes* é geralmente feita usando um esquema de sincronização produtor/consumidor.

A solução do problema dos produtores/consumidores envolve três aspectos de coordenação distintos e complementares:

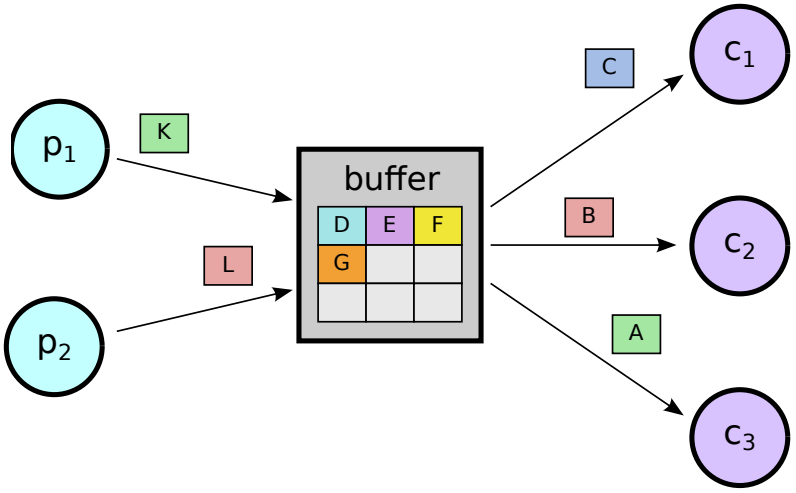


Figura 12.1: O problema dos produtores/consumidores.

- A exclusão mútua no acesso ao *buffer*, para evitar condições de disputa entre produtores e/ou consumidores que poderiam corromper o conteúdo do *buffer*.
- A suspensão dos produtores no caso do *buffer* estar cheio: os produtores devem esperar até que surjam vagas livres no *buffer*.
- A suspensão dos consumidores no caso do *buffer* estar vazio: os consumidores devem esperar até que surjam novos itens a consumir no *buffer*.

12.1.1 Solução usando semáforos

Pode-se resolver o problema dos produtores/consumidores de forma eficiente usando um *mutex* e dois semáforos, um para cada aspecto de coordenação envolvido. O código a seguir ilustra de forma simplificada uma solução para esse problema, considerando um buffer com capacidade para N itens, inicialmente vazio:

```
1  mutex mbuf ;          // controla o acesso ao buffer
2  semaphore item ;      // controla os itens no buffer (inicia em 0)
3  semaphore vaga ;      // controla as vagas no buffer (inicia em N)
4
5  task produtor ()
6  {
7     while (1)
8     {
9        ...                    // produz um item
10       down (vaga) ;          // espera uma vaga no buffer
11       lock (mbuf) ;          // espera acesso exclusivo ao buffer
12       ...                    // deposita o item no buffer
13       unlock (mbuf) ;        // libera o acesso ao buffer
14       up (item) ;            // indica a presença de um novo item no buffer
15    }
16 }
17
18 task consumidor ()
19 {
20    while (1)
21    {
22       down (item) ;          // espera um novo item no buffer
23       lock (mbuf) ;          // espera acesso exclusivo ao buffer
24       ...                    // retira o item do buffer
25       unlock (mbuf) ;        // libera o acesso ao buffer
26       up (vaga) ;            // indica a liberação de uma vaga no buffer
27       ...                    // consome o item retirado do buffer
28    }
29 }
```

É importante observar que essa solução é genérica, pois não depende do tamanho do buffer, do número de produtores nem do número de consumidores.

12.1.2 Solução usando variáveis de condição

O problema dos produtores/consumidores também pode ser resolvido com variáveis de condição. Além do *mutex* para acesso exclusivo ao *buffer*, são necessárias variáveis de condição para indicar a presença de itens e de vagas no *buffer*. A listagem a seguir ilustra uma solução, lembrando que `N` é a capacidade do *buffer* e `num_itens` é o número de itens no *buffer* em um dado instante.

```
1  mutex mbuf ;                        // controla o acesso ao buffer
2  condition item ;                    // condição: existe item no buffer
3  condition vaga ;                    // condição: existe vaga no buffer
4
5  task produtor ()
6  {
7     while (1)
8     {
9        ...                           // produz um item
10       lock (mbuf) ;                 // obtem o mutex do buffer
11       while (num_itens == N)        // enquanto o buffer estiver cheio
12          wait (vaga, m) ;           // espera uma vaga, liberando o buffer
13       ...                           // deposita o item no buffer
14       signal (item) ;               // sinaliza um novo item
15       unlock (mbuf) ;               // libera o buffer
16    }
17 }
18
19 task consumidor ()
20 {
21    while (1)
22    {
23       lock (mbuf) ;                 // obtem o mutex do buffer
24       while (num_items == 0)        // enquanto o buffer estiver vazio
25          wait (item, m) ;           // espera um item, liberando o buffer
26       ...                           // retira o item no buffer
27       signal (vaga) ;               // sinaliza uma vaga livre
28       unlock (mbuf) ;               // libera o buffer
29       ...                           // consome o item retirado do buffer
30    }
31 }
```

12.2 Leitores/escritores

Outra situação que ocorre com frequência em sistemas concorrentes é o problema dos *leitores/escritores*. Neste problema, um conjunto de tarefas acessam de forma concorrente uma área de memória compartilhada, na qual podem fazer leituras ou escritas de valores. De acordo com as condições de Bernstein (Seção 10.1.3), as leituras podem ser feitas em paralelo, pois não interferem umas com as outras, mas as escritas têm de ser feitas com acesso exclusivo à área compartilhada, para evitar condições de disputa. A Figura 12.2 mostra leitores e escritores acessando de forma concorrente uma matriz de números inteiros M.

O estilo de sincronização *leitores/escritores* é encontrado com muita frequência em aplicações com múltiplas *threads*. O padrão POSIX define mecanismos para a criação e uso de travas com essa funcionalidade, acessíveis através de chamadas como `pthread_rwlock_init()`, entre outras.

12.2.1 Solução simplista

Uma solução simplista para esse problema consistiria em proteger o acesso à área compartilhada com um *mutex* ou semáforo inicializado em 1; assim, somente um

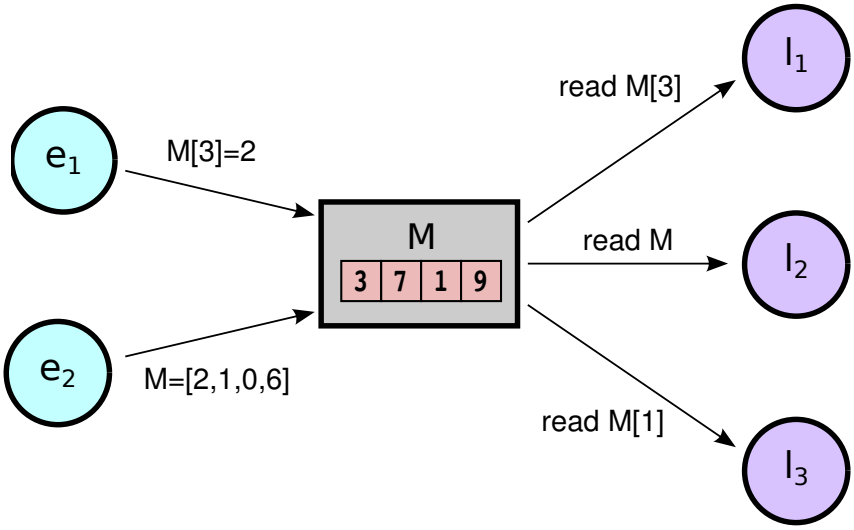


Figura 12.2: O problema dos leitores/escritores.

processo por vez poderia acessar a área, garantindo a integridade de todas as operações. O código a seguir ilustra essa abordagem:

```
1  mutex marea ;              // controla o acesso à área
2
3  task leitor ()
4  {
5     while (1)
6     {
7        lock (marea) ;        // requer acesso exclusivo à área
8        ...                   // lê dados da área compartilhada
9        unlock (marea) ;      // libera o acesso à área
10       ...
11    }
12 }
13
14 task escritor ()
15 {
16    while (1)
17    {
18       lock (marea) ;        // requer acesso exclusivo à área
19       ...                   // escreve dados na área compartilhada
20       unlock (marea) ;      // libera o acesso à área
21       ...
22    }
23 }
```

Essa solução deixa muito a desejar em termos de desempenho, porque restringe desnecessariamente o acesso dos leitores à área compartilhada: como a operação de leitura não altera os valores armazenados, não haveria problema em permitir o acesso paralelo de vários leitores à área compartilhada, desde que as escritas continuem sendo feitas de forma exclusiva.

12.2.2 Solução com priorização dos leitores

Uma solução melhor para o problema dos leitores/escritores, considerando a possibilidade de acesso paralelo pelos leitores, seria a indicada na listagem a seguir. Nela, os leitores dividem a responsabilidade pelo *mutex* de controle da área compartilhada (`marea`): o primeiro leitor a entrar obtém esse *mutex*, que só será liberado pelo último leitor a sair da área. Um contador de leitores permite saber se um leitor é o primeiro a entrar ou o último a sair. Como esse contador pode sofrer condições de disputa, o acesso a ele é controlado por outro *mutex* (`mcont`).

```
1  mutex marea ;              // controla o acesso à área
2  mutex mcont ;              // controla o acesso ao contador
3
4  int num_leitores = 0 ;     // número de leitores acessando a área
5
6  task leitor ()
7  {
8     while (1)
9     {
10       lock (mcont) ;        // requer acesso exclusivo ao contador
11       num_leitores++ ;      // incrementa contador de leitores
12       if (num_leitores == 1) // sou o primeiro leitor a entrar?
13          lock (marea) ;     // requer acesso à área
14       unlock (mcont) ;      // libera o contador
15
16       ...                   // lê dados da área compartilhada
17
18       lock (mcont) ;        // requer acesso exclusivo ao contador
19       num_leitores-- ;      // decrementa contador de leitores
20       if (num_leitores == 0) // sou o último leitor a sair?
21          unlock (marea) ;   // libera o acesso à área
22       unlock (mcont) ;      // libera o contador
23       ...
24    }
25 }
26
27 escritor ()
28 {
29    while (1)
30    {
31       lock (marea) ;        // requer acesso exclusivo à área
32       ...                   // escreve dados na área compartilhada
33       unlock (marea) ;      // libera o acesso à área
34       ...
35    }
36 }
```

Essa solução melhora o desempenho das operações de leitura, pois permite que vários leitores a acessem simultaneamente. Contudo, introduz um novo problema: a *priorização dos leitores*. De fato, sempre que algum leitor estiver acessando a área compartilhada, outros leitores também podem acessá-la, enquanto eventuais escritores têm de esperar até a área ficar livre (sem leitores). Caso existam muito leitores em atividade, os escritores podem ficar impedidos de acessar a área, pois ela nunca ficará vazia (inanição).

Soluções com priorização para os escritores e soluções equitativas entre ambos podem ser facilmente encontradas na literatura [Raynal, 1986; Ben-Ari, 1990].

12.3 O jantar dos selvagens

Uma variação curiosa do problema dos produtores/consumidores foi proposta em [Andrews, 1991] com o nome de *Jantar dos Selvagens*: uma tribo de selvagens está jantando ao redor de um grande caldeirão contendo N porções de missionário cozido. Quando um selvagem quer comer, ele se serve de uma porção no caldeirão, a menos que este esteja vazio. Nesse caso, o selvagem primeiro acorda o cozinheiro da tribo e espera que ele encha o caldeirão de volta, para então se servir novamente. Após encher o caldeirão, o cozinheiro volta a dormir.

```
1  task cozinheiro ()
2  {
3     while (1)
4     {
5        encher_caldeirao () ;
6        dormir () ;
7     }
8  }
9
10 task selvagem ()
11 {
12    while (1)
13    {
14       servir () ;
15       comer () ;
16    }
17 }
```

As restrições de sincronização deste problema são as seguintes:

- Selvagens não podem se servir ao mesmo tempo (mas podem comer ao mesmo tempo);
- Selvagens não podem se servir se o caldeirão estiver vazio;
- O cozinheiro só pode encher o caldeirão quando ele estiver vazio.

Uma solução simples para esse problema, apresentada em [Downey, 2016], é a seguinte:

```
1  int porcoes = 0 ;           // porções no caldeirão
2  mutex mc ;                  // controla acesso ao caldeirão
3  semaphore cald_vazio ;      // indica caldeirão vazio (inicia em 0)
4  semaphore cald_cheio ;      // indica caldeirão vazio (inicia em 0)
5
6  task cozinheiro ()
7  {
8     while (1)
9     {
10       down (cald_vazio) ;   // aguarda o caldeirão esvaziar
11       porcoes += M ;        // enche o caldeirão (M porções)
12       up (cald_cheio) ;     // avisa que encheu o caldeirão
13    }
14 }
15
16
17 task selvagem ()
18 {
19    while (1)
20    {
21       lock (mc) ;           // tenta acessar o caldeirão
22       if (porcoes == 0)     // caldeirão vazio?
23       {
24          up (cald_vazio) ;  // avisa que caldeirão esvaziou
25          down (cald_cheio) ; // espera ficar cheio de novo
26       }
27       porcoes-- ;           // serve uma porção
28       unlock (mc) ;         // libera o caldeirão
29       comer () ;
30    }
31 }
```

12.4 O jantar dos filósofos

Um dos problemas clássicos de coordenação mais conhecidos é o *jantar dos filósofos*, proposto inicialmente por Dijkstra [Raynal, 1986; Ben-Ari, 1990]. Neste problema, um grupo de cinco filósofos chineses alterna suas vidas entre meditar e comer.

Ha uma mesa redonda com um lugar fixo para cada filósofo, com um prato, cinco palitos (*hashis*) compartilhados e um grande prato de arroz ao meio[1]. Para comer, um filósofo f_i precisa pegar o palito à sua direita (p_i) e à sua esquerda (p_{i+1}), um de cada vez. Como os palitos são compartilhados, dois filósofos vizinhos não podem comer ao mesmo tempo. Os filósofos não conversam entre si nem podem observar os estados uns dos outros. A Figura 12.3 ilustra essa situação.

O problema do jantar dos filósofos é representativo de uma grande classe de problemas de sincronização entre vários processos e vários recursos sem usar um coordenador central. Resolver o problema do jantar dos filósofos consiste em encontrar uma forma de coordenar suas atividades de maneira que todos os filósofos consigam meditar e comer. A listagem a seguir é o pseudocódigo de uma implementação do

[1]Na versão inicial de Dijkstra, os filósofos compartilhavam garfos e comiam spaghetti; neste texto os filósofos são chineses e comem arroz...

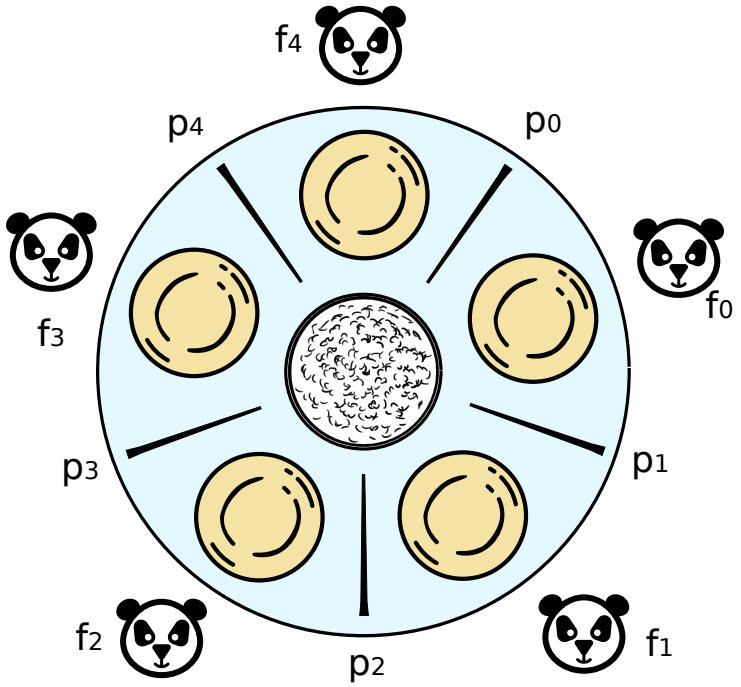


Figura 12.3: O jantar dos filósofos chineses.

comportamento básico dos filósofos, na qual cada filósofo é uma tarefa e palito é um semáforo:

```
1  #define NUMFILO 5
2  semaphore hashi [NUMFILO] ; // um semáforo para cada palito (iniciam em 1)
3
4  task filosofo (int i)              // filósofo i (entre 0 e 4)
5  {
6     int dir = i ;
7     int esq = (i+1) % NUMFILO] ;
8
9     while (1)
10    {
11       meditar () ;
12       down (hashi [dir]) ;          // pega palito direito
13       down (hashi [esq]) ;          // pega palito esquerdo
14       comer () ;
15       up (hashi [dir]) ;            // devolve palito direito
16       up (hashi [esq]) ;            // devolve palito esquerdo
17    }
18 }
```

Soluções simples para esse problema podem provocar impasses, ou seja, situações nas quais todos os filósofos ficam bloqueados (impasses serão estudados na Seção 13). Outras soluções podem provocar inanição (*starvation*), ou seja, alguns dos filósofos nunca conseguem comer. A Figura 12.4 apresenta os filósofos em uma situação de impasse: cada filósofo obteve o palito à sua direita e está esperando o palito à sua esquerda (indicado pelas setas tracejadas). Como todos os filósofos estão esperando, ninguém mais consegue executar.

Uma solução trivial para o problema do jantar dos filósofos consiste em colocar um "saleiro" hipotético sobre a mesa: quando um filósofo deseja comer, ele deve obter o

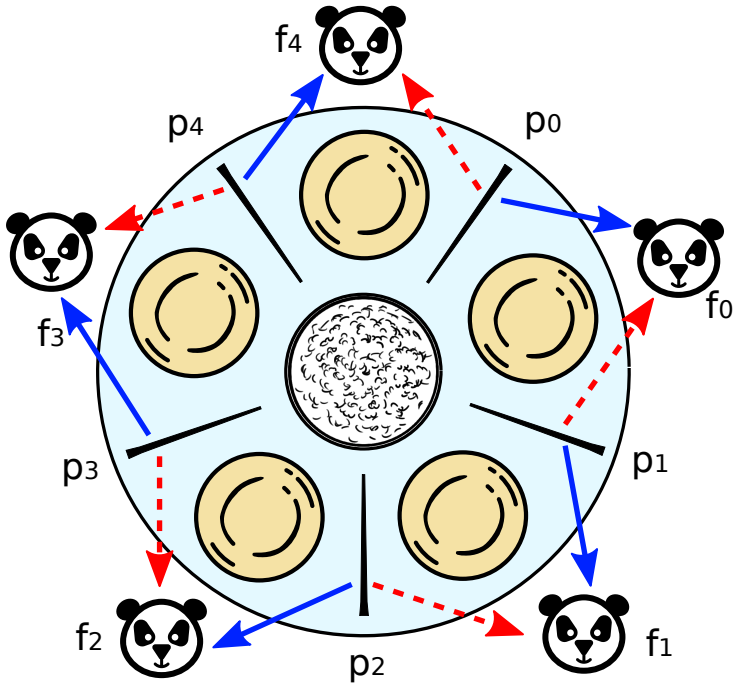


Figura 12.4: Um impasse no jantar dos filósofos chineses.

saleiro antes de obter os palitos; assim que tiver ambos os palitos, ele devolve o saleiro à mesa e pode comer:

```
#define NUMFILO 5
semaphore hashi [NUMFILO] ; // um semáforo para cada palito (iniciam em 1)
semaphore saleiro ;         // um semáforo para o saleiro

task filosofo (int i)               // filósofo i (entre 0 e 4)
{
   int dir = i ;
   int esq = (i+1) % NUMFILO] ;

   while (1)
   {
      meditar () ;
      down (saleiro) ;              // pega saleiro
      down (hashi [dir]) ;          // pega palito direito
      down (hashi [esq]) ;          // pega palito esquerdo
      up (saleiro) ;                // devolve saleiro
      comer () ;
      up (hashi [dir]) ;            // devolve palito direito
      up (hashi [esq]) ;            // devolve palito esquerdo
   }
}
```

Obviamente, a solução do saleiro serializa o acesso aos palitos e por isso tem baixo desempenho se houverem muitos filósofos disputando o mesmo saleiro. Diversas soluções eficientes podem ser encontradas na literatura para esse problema [Tanenbaum, 2003; Silberschatz et al., 2001].

Referências

G. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.

M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice-Hall, 1990.

A. Downey. *The Little Book of Semaphores*. Green Tea Press, 2016. urlhttp://greenteapress.com/wp/semaphores/.

M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, 1986.

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

A. Tanenbaum. *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall, 2003.

Capítulo 13

Impasses

O controle de concorrência entre tarefas acessando recursos compartilhados implica em suspender algumas tarefas enquanto outras acessam os recursos, de forma a garantir a consistência dos mesmos. Para isso, a cada recurso é associado um semáforo ou outro mecanismo equivalente. Assim, as tarefas solicitam e aguardam a liberação de cada semáforo para poder acessar o recurso correspondente.

Em alguns casos, o uso de semáforos ou *mutexes* pode levar a situações de **impasse** (ou *deadlock*), nas quais todas as tarefas envolvidas ficam bloqueadas aguardando a liberação de semáforos, e nada mais acontece. Este capítulo visa compreender os impasses e como tratá-los.

13.1 Exemplo de impasse

Para ilustrar uma situação de impasse, será utilizado o exemplo de acesso a uma conta bancária apresentado na Seção 10.1. O código a seguir implementa uma operação de transferência de fundos entre duas contas bancárias. A cada conta está associado um *mutex*, usado para prover acesso exclusivo aos dados da conta e assim evitar condições de disputa:

```
1  typedef struct conta_t
2  {
3     int saldo ;                    // saldo atual da conta
4     mutex m ;                      // mutex associado à conta
5     ...                            // outras informações da conta
6  } conta_t ;
7
8  void transferir (conta_t* contaDeb, conta_t* contaCred, int valor)
9  {
10    lock (contaDeb->m) ;           // obtém acesso a contaDeb
11    lock (contaCred->m) ;          // obtém acesso a contCred
12
13    if (contaDeb->saldo >= valor)
14    {
15       contaDeb->saldo -= valor ;  // debita valor de contaDeb
16       contaCred->saldo += valor ; // credita valor em contaCred
17    }
18    unlock (contaDeb->m) ;         // libera acesso a contaDeb
19    unlock (contaCred->m) ;        // libera acesso a contaCred
20 }
```

Caso dois clientes do banco (representados por duas tarefas t_1 e t_2) resolvam fazer simultaneamente operações de transferência entre suas contas (t_1 transfere um valor v_1 de c_1 para c_2 e t_2 transfere um valor v_2 de c_2 para c_1), poderá ocorrer uma situação de impasse, como mostra o diagrama de tempo da Figura 13.1.

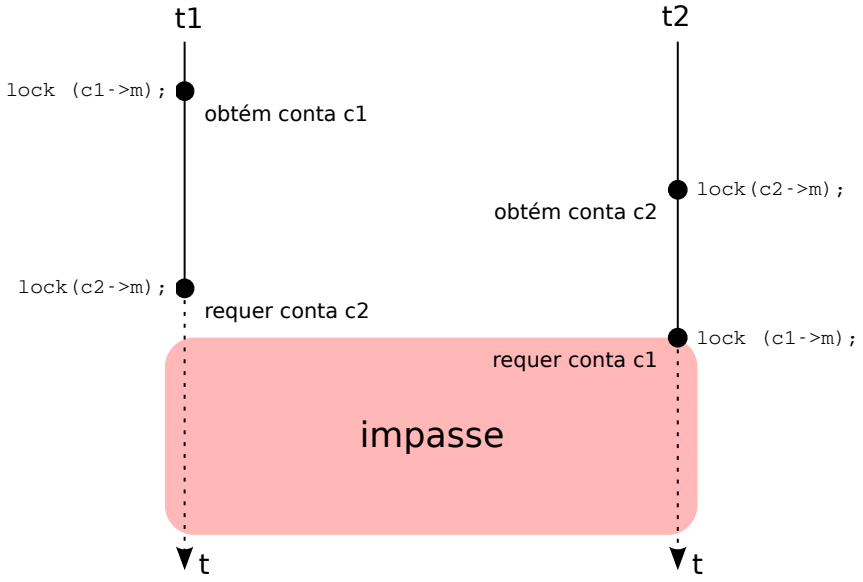


Figura 13.1: Impasse entre duas transferências.

Nessa situação, a tarefa t_1 detém o *mutex* de c_1 e solicita o *mutex* de c_2, enquanto t_2 detém o *mutex* de c_2 e solicita o *mutex* de c_1. Como nenhuma das duas tarefas poderá prosseguir sem obter o *mutex* desejado, nem poderá liberar o *mutex* de sua conta antes de obter o outro *mutex* e realizar a transferência, se estabelece um impasse.

Impasses são situações muito frequentes em programas concorrentes, mas também podem ocorrer em sistemas distribuídos e mesmo em situações fora da informática. A Figura 13.2 mostra como exemplo uma situação de impasse ocorrida em um cruzamento de São Paulo SP, no início de 2017. Antes de conhecer as técnicas

de tratamento de impasses, é importante compreender suas principais causas e saber caracterizá-los adequadamente, o que será estudado nas próximas seções.

Figura 13.2: Uma situação de impasse no trânsito.

13.2 Condições para impasses

Em um impasse, duas ou mais tarefas se encontram bloqueadas, aguardando eventos que dependem somente delas, como a liberação de semáforos. Em outras palavras, não existe influência de entidades externas em uma situação de impasse. Além disso, como as tarefas envolvidas detêm alguns recursos compartilhados (representados por semáforos), outras tarefas que vierem a requisitar esses recursos também ficarão bloqueadas, aumentando gradativamente o impasse, o que pode levar o sistema inteiro a parar de funcionar.

Formalmente, um conjunto de N tarefas se encontra em um impasse se cada uma das tarefas aguarda um evento que somente outra tarefa do conjunto poderá produzir. Quatro condições fundamentais são necessárias para que os impasses possam ocorrer [Coffman et al., 1971; Ben-Ari, 1990]:

Exclusão mútua: o acesso aos recursos deve ser feito de forma mutuamente exclusiva, controlada por semáforos ou mecanismos equivalentes. No exemplo da conta corrente, apenas uma tarefa por vez pode acessar cada conta.

Posse e espera: uma tarefa pode solicitar o acesso a outros recursos sem ter de liberar os recursos que já detém. No exemplo da conta corrente, cada tarefa detém o semáforo de uma conta e solicita o semáforo da outra conta para poder prosseguir.

Não-preempção: uma tarefa somente libera os recursos que detém quando assim o decidir, e não os perde de forma imprevista (ou seja, o sistema operacional não

retira à força os recursos alocados às tarefas). No exemplo da conta corrente, cada tarefa detém os *mutexes* obtidos até liberá-los explicitamente.

Espera circular: existe um ciclo de esperas pela liberação de recursos entre as tarefas envolvidas: a tarefa t_1 aguarda um recurso retido pela tarefa t_2 (formalmente, $t_1 \rightarrow t_2$), que aguarda um recurso retido pela tarefa t_3, e assim por diante, sendo que a tarefa t_n aguarda um recurso retido por t_1. Essa dependência circular pode ser expressa formalmente da seguinte forma: $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \cdots \rightarrow t_n \rightarrow t_1$. No exemplo da conta corrente, pode-se observar claramente que $t_1 \rightarrow t_2 \rightarrow t_1$.

Deve-se observar que essas quatro condições são **necessárias** para a formação de impasses; se uma delas não for verificada, não existem impasses no sistema. Por outro lado, não são condições **suficientes** para a existência de impasses, ou seja, a verificação dessas quatro condições não garante a presença de um impasse no sistema. Essas condições somente são suficientes se existir apenas uma instância de cada tipo de recurso, como será discutido na próxima seção.

13.3 Grafos de alocação de recursos

É possível representar graficamente a alocação de recursos entre as tarefas de um sistema concorrente. A representação gráfica provê uma visão mais clara da distribuição dos recursos e permite detectar visualmente a presença de esperas circulares que podem caracterizar impasses. Em um *grafo de alocação de recursos* [Holt, 1972], as tarefas são representadas por círculos (○) e os recursos por retângulos (□). A posse de um recurso por uma tarefa é representada como □ → ○ (lido como "o recurso pertence à tarefa"), enquanto a requisição de um recurso por uma tarefa é indicada por ○ ⇢ □ (lido como "a tarefa requer o recurso"). Para facilitar a leitura, as setas

A Figura 13.3 apresenta o grafo de alocação de recursos da situação de impasse ocorrida na transferência de valores entre contas bancárias da Figura 13.1. Nessa figura percebe-se claramente a dependência cíclica entre tarefas e recursos no ciclo $t_1 \dashrightarrow c_2 \rightarrow t_2 \dashrightarrow c_1 \rightarrow t_1$, que neste caso evidencia um impasse. Como há um só recurso de cada tipo (apenas uma conta c_1 e uma conta c_2), as quatro condições necessárias se mostram também suficientes para caracterizar um impasse.

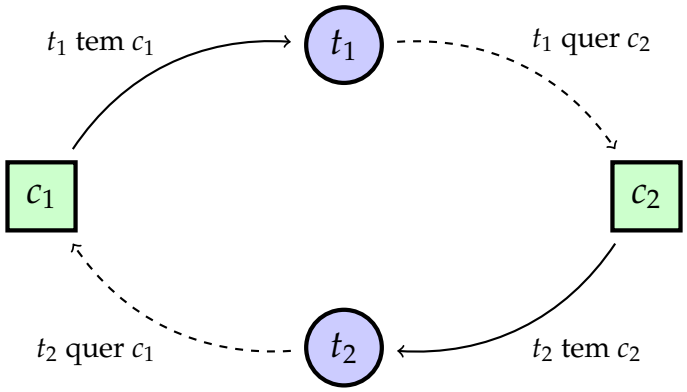


Figura 13.3: Grafo de alocação de recursos com impasse.

Alguns recursos lógicos ou físicos de um sistema computacional podem ter múltiplas instâncias: por exemplo, um sistema pode ter duas impressoras idênticas

instaladas, o que constituiria um recurso (impressora) com duas instâncias equivalentes, que podem ser alocadas de forma independente. No grafo de alocação de recursos, a existência de múltiplas instâncias de um recurso é representada através de "fichas" dentro dos retângulos. Por exemplo, as duas instâncias de impressora seriam indicadas no grafo como [••] . A Figura 13.4 indica apresenta um grafo de alocação de recursos considerando alguns recursos com múltiplas instâncias.

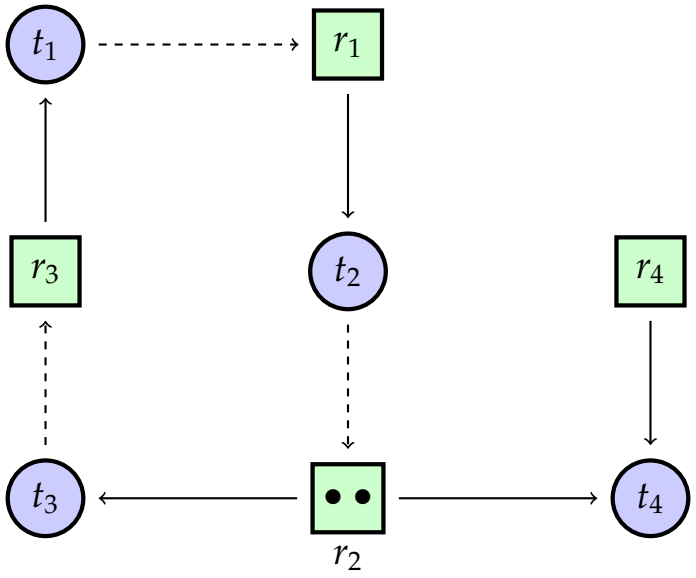


Figura 13.4: Grafo de alocação com múltiplas instâncias de recursos.

É importante observar que a ocorrência de ciclos em um grafo de alocação, envolvendo recursos com múltiplas instâncias, **pode indicar** a presença de um impasse, mas **não garante** sua existência. Por exemplo, o ciclo $t_1 \dashrightarrow r_1 \rightarrow t_2 \dashrightarrow r_2 \rightarrow t_3 \dashrightarrow r_3 \rightarrow t_1$ presente no diagrama da Figura 13.4 não representa um impasse, porque a qualquer momento a tarefa t_4 (que não está esperando recursos) pode liberar uma instância do recurso r_2, solicitado por t_2, permitindo atender a demanda de t_2 e desfazendo assim o ciclo. Um algoritmo de detecção de impasses envolvendo recursos com múltiplas instâncias é apresentado em [Tanenbaum, 2003].

13.4 Técnicas de tratamento de impasses

Como os impasses paralisam tarefas que detêm recursos, sua ocorrência pode gerar consequências graves, como a paralisação gradativa de todas as tarefas que dependam dos recursos envolvidos, o que pode levar à paralisação de todo o sistema. Devido a esse risco, diversas técnicas de tratamento de impasses foram propostas. Essas técnicas podem definir regras estruturais que previnam impasses, podem atuar de forma proativa, se antecipando aos impasses e impedindo sua ocorrência, ou podem agir de forma reativa, detectando os impasses que se formam no sistema e tomando medidas para resolvê-los.

Embora o risco de impasses seja uma questão importante, os sistemas operacionais de mercado (Windows, Linux, MacOS, etc.) adotam a solução mais simples: **ignorar o risco**, na maioria das situações. Devido ao custo computacional necessário ao tratamento de impasses e à sua forte dependência da lógica das aplicações envolvidas, os

projetistas de sistemas operacionais normalmente preferem deixar a gestão de impasses por conta dos desenvolvedores de aplicações.

As principais técnicas usadas para tratar impasses em um sistema concorrente são: **prevenir** impasses através, de regras rígidas para a programação dos sistemas, **impedir** impasses, por meio do acompanhamento contínuo da alocação dos recursos às tarefas, e **detectar e resolver** impasses. Essas técnicas serão detalhadas nas próximas seções.

13.4.1 Prevenção de impasses

As técnicas de prevenção de impasses buscam garantir que impasses nunca possam ocorrer no sistema. Para alcançar esse objetivo, a estrutura do sistema e a lógica das aplicações devem ser construídas de forma que as quatro condições fundamentais para a ocorrência de impasses, apresentadas na Seção 13.2, nunca sejam integralmente satisfeitas. Se ao menos uma das quatro condições for quebrada por essas regras estruturais, os impasses não poderão ocorrer. A seguir, cada uma das condições necessárias é analisada de acordo com essa premissa:

Exclusão mútua: se não houver exclusão mútua no acesso a recursos, não poderão ocorrer impasses. Mas, como garantir a integridade de recursos compartilhados sem usar mecanismos de exclusão mútua? Uma solução interessante é usada na gerência de impressoras: um processo *servidor de impressão* (*printer spooler*) gerencia a impressora e atende as solicitações dos demais processos. Com isso, os processos que desejam usar a impressora não precisam obter acesso exclusivo a ela. A técnica de *spooling* previne impasses envolvendo as impressoras, mas não é facilmente aplicável a certos tipos de recurso, como arquivos em disco e áreas de memória compartilhada.

Posse e espera: caso as tarefas usem apenas um recurso de cada vez, solicitando-o e liberando-o logo após o uso, impasses não poderão ocorrer. No exemplo da transferência de fundos da Figura 13.1, seria possível separar a operação de transferência em duas operações isoladas: débito em c_1 e crédito em c_2 (ou vice-versa), sem a necessidade de acesso exclusivo simultâneo às duas contas. Com isso, a condição de posse e espera seria quebrada e o impasse evitado.

Outra possibilidade seria somente permitir a execução de tarefas que detenham todos os recursos necessários antes de iniciar. Todavia, essa abordagem poderia levar as tarefas a reter os recursos por muito mais tempo que o necessário para suas operações, degradando o desempenho do sistema.

Uma terceira possibilidade seria associar um prazo (*time-out*) às solicitações de recursos: ao solicitar um recurso, a tarefa define um tempo máximo de espera por ele; caso o prazo expire, a tarefa pode tentar novamente ou desistir, liberando os demais recursos que detém.

Não-preempção: normalmente uma tarefa obtém e libera os recursos de que necessita, de acordo com sua lógica interna. Se for possível "arrancar" um recurso da tarefa, sem que esta o libere explicitamente, impasses envolvendo aquele recurso não poderão ocorrer. Essa técnica é frequentemente usada em recursos cujo estado interno pode ser salvo e restaurado de forma transparente para a tarefa, como páginas de memória e o próprio processador (nas trocas de contexto).

No entanto, é de difícil aplicação sobre recursos como arquivos ou áreas de memória compartilhada, porque a preempção viola a exclusão mútua e pode provocar inconsistências no estado interno do recurso.

Espera circular: um impasse é uma cadeia de dependências entre tarefas e recursos que forma um ciclo. Ao prevenir a formação de tais ciclos, impasses não poderão ocorrer. A estratégia mais simples para prevenir a formação de ciclos é ordenar todos os recursos do sistema de acordo com uma ordem global única, e forçar as tarefas a solicitar os recursos obedecendo a essa ordem. No exemplo da transferência de fundos da Figura 13.1, o número de conta bancária poderia definir uma ordem global ($c_1 < c_2$, por exemplo). Assim, todas as tarefas deveriam solicitar primeiro o acesso à conta mais antiga e depois à mais recente (ou vice-versa, mas sempre na mesma ordem para todas as tarefas). Com isso, elimina-se a possibilidade de impasses.

As técnicas de prevenção de impasses devem ser consideradas na construção de aplicações multitarefas complexas, pois permitem prevenir impasses sem muito esforço computacional durante a execução. Frequentemente, uma reorganização de pequenos trechos do código da aplicação é suficiente para prevenir impasses, como ocorre no exemplo da transferência bancária.

13.4.2 Impedimento de impasses

Outra forma de tratar os impasses preventivamente consiste em acompanhar a alocação dos recursos às tarefas e, de acordo com algum algoritmo, negar acessos de recursos que possam levar a impasses. Uma noção essencial nas técnicas de impedimento de impasses é o conceito de **estado seguro**. Cada estado do sistema é definido pela distribuição dos recursos entre as tarefas.

O conjunto de todos os estados possíveis do sistema durante sua execução forma um grafo de estados, no qual as arestas indicam as alocações e liberações de recursos. Um determinado estado é considerado seguro se, a partir dele, é possível concluir as tarefas pendentes. Caso o estado em questão somente leve a impasses, ele é considerado um **estado inseguro**. As técnicas de impedimento de impasses devem portanto manter o sistema sempre em um estado seguro, evitando entrar em estados inseguros.

A Figura 13.5 ilustra o grafo de estados do sistema de transferência de valores com duas tarefas analisado no início deste capítulo. Cada estado desse grafo é a combinação dos estados individuais das duas tarefas[1]. Pode-se observar no grafo que o estado e_{10} corresponde a um impasse, pois a partir dele não há mais nenhuma possibilidade de evolução do sistema a outros estados. Além disso, os estados e_4, e_7 e e_8 são considerados estados inseguros, pois levam invariavelmente na direção do impasse em e_{10}. Os demais estados são considerados seguros, pois a partir de qualquer um deles é possível continuar a execução e retornar ao estado inicial e_0. Obviamente, operações que levem a estados inseguros devem ser impedidas, como $e_1 \dashrightarrow e_4$ e $e_2 \dashrightarrow e_4$.

A técnica de impedimento de impasses mais conhecida é o *algoritmo do banqueiro*, criado por Dijkstra em 1965 [Tanenbaum, 2003]. Esse algoritmo faz uma analogia entre as tarefas de um sistema e os clientes de um banco, tratando os recursos como créditos

[1] Este grafo de estados é simplificado; o grafo completo, detalhando cada solicitação, alocação e liberação de recursos, tem cerca de 40 estados possíveis.

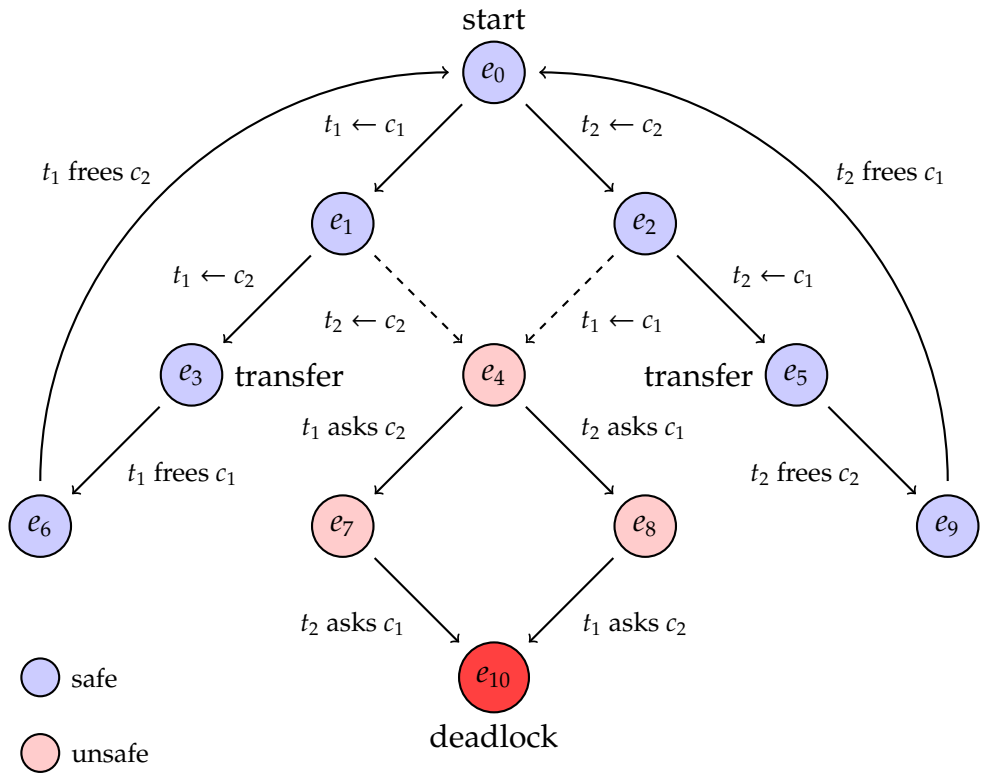


Figura 13.5: Grafo de estados do sistema de transferências com duas tarefas.

emprestados às tarefas para a realização de suas atividades. O banqueiro decide que solicitações de empréstimo deve atender para conservar suas finanças em um estado seguro.

As técnicas de impedimento de impasses necessitam de algum conhecimento prévio sobre o comportamento das tarefas para poder operar. Normalmente é necessário conhecer com antecedência que recursos serão acessados por cada tarefa, quantas instâncias de cada um serão necessárias e qual a ordem de acesso aos recursos. Por essa razão, são pouco utilizadas na prática.

13.4.3 Detecção e resolução de impasses

Nesta abordagem, nenhuma medida preventiva é adotada para prevenir ou evitar impasses. As tarefas executam normalmente suas atividades, alocando e liberando recursos conforme suas necessidades. Quando ocorrer um impasse, o sistema deve detectá-lo, determinar quais as tarefas e recursos envolvidos e tomar medidas para desfazê-lo. Para aplicar essa técnica, duas questões importantes devem ser respondidas: como detectar os impasses? E como resolvê-los?

A **detecção de impasses** pode ser feita através da inspeção do grafo de alocação de recursos (Seção 13.3), que deve ser mantido pelo sistema e atualizado a cada alocação ou liberação de recurso. Um algoritmo de detecção de ciclos no grafo deve ser executado periodicamente, para verificar a presença das dependências cíclicas que podem indicar impasses.

Alguns problemas decorrentes dessa estratégia são o custo de manutenção contínua do grafo de alocação e, sobretudo, o custo de sua análise: algoritmos de busca de ciclos em grafos têm custo computacional elevado, portanto sua ativação com muita

frequência poderá prejudicar o desempenho do sistema. Por outro lado, se a detecção for ativada apenas esporadicamente, impasses podem demorar muito para ser detectados, o que também é ruim para o desempenho do sistema.

Uma vez detectado um impasse e identificadas as tarefas e recursos envolvidos, o sistema deve proceder à **resolução do impasse**, que pode ser feita de duas formas:

Eliminar tarefas: uma ou mais tarefas envolvidas no impasse são eliminadas, liberando seus recursos para que as demais tarefas possam prosseguir. A escolha das tarefas a eliminar deve levar em conta vários fatores, como o tempo de vida de cada uma, a quantidade de recursos que cada tarefa detém, o prejuízo para os usuários que dependem dessas tarefas, etc.

Retroceder tarefas: uma ou mais tarefas envolvidas no impasse têm sua execução parcialmente desfeita (uma técnica chamada *rollback*), de forma a fazer o sistema retornar a um estado seguro anterior ao impasse. Para retroceder a execução de uma tarefa, é necessário salvar periodicamente seu estado, de forma a poder recuperar um estado anterior quando necessário[2]. Além disso, operações envolvendo a rede ou interações com o usuário podem ser muito difíceis ou mesmo impossíveis de retroceder: como desfazer o envio de um pacote de rede, ou a reprodução de um arquivo de áudio na tela do usuário?

A detecção e resolução de impasses é uma abordagem interessante, mas relativamente pouco usada fora de situações muito específicas, porque o custo de detecção pode ser elevado e as alternativas de resolução sempre implicam perder tarefas ou parte das execuções já realizadas. Essa técnica é aplicada, por exemplo, no gerenciamento de transações em sistemas de bancos de dados, pois são providos mecanismos para criar *checkpoints* dos registros envolvidos antes da transação e para efetuar o *rollback* da mesma em caso de impasse.

Referências

M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice-Hall, 1990.

E. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3 (2):67–78, 1971.

R. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3): 179–196, september 1972.

A. Tanenbaum. *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall, 2003.

[2] Essa técnica é conhecida como *checkpointing* e os estados anteriores salvos são denominados *checkpoints*.

Parte IV

Gestão da memória

Capítulo 14

Conceitos básicos

A memória principal é um componente fundamental em qualquer sistema de computação. Ela constitui o "espaço de trabalho" do sistema, no qual são mantidos os processos, threads e bibliotecas compartilhadas, além do próprio núcleo do sistema operacional, com seu código e suas estruturas de dados. O hardware de memória pode ser bastante complexo, envolvendo diversas estruturas, como memórias RAM, caches, unidade de gerência, barramentos, etc, o que exige um esforço de gerência significativo por parte do sistema operacional. Neste capítulo serão estudados os conceitos básicos de memória sob a ótica do usuário e do sistema operacional.

14.1 Tipos de memória

Existem diversos tipos de memória em um sistema de computação, cada um com suas próprias características e particularidades, mas todos com um mesmo objetivo: armazenar informação. Observando um sistema computacional típico, pode-se identificar vários locais onde dados são armazenados: os registradores e o cache interno do processador (denominado *cache L1*), o cache externo da placa mãe (*cache L2*) e a memória principal (RAM). Além disso, discos e unidades de armazenamento externas (*pendrives*, CD-ROMs, DVD-ROMs, fitas magnéticas, etc.) também podem ser considerados memória em um sentido mais amplo, pois também têm como função o armazenamento de informação.

Esses componentes de hardware são construídos usando diversas tecnologias e por isso têm características distintas, como a capacidade de armazenamento, a velocidade de operação, o consumo de energia, o custo por byte armazenado e a volatilidade. Essas características permitem definir uma *hierarquia de memória*, geralmente representada na forma de uma pirâmide (Figura 14.1).

Nessa pirâmide, observa-se que memórias mais rápidas, como os registradores da CPU e os caches, são menores (têm menor capacidade de armazenamento), mais caras e consomem mais energia que memórias mais lentas, como a memória principal (RAM) e os discos. Além disso, as memórias mais rápidas são *voláteis*, ou seja, perdem seu conteúdo ao ficarem sem energia, quando o computador é desligado. Memórias que preservam seu conteúdo mesmo quando não tiverem energia, como as unidades *Flash* e os discos rígidos, são denominadas *memórias não-voláteis*.

Outra característica importante das memórias é a rapidez de seu funcionamento, que pode ser traduzida em duas grandezas: o *tempo de acesso* (ou *latência*) e a *taxa de transferência*. O tempo de acesso caracteriza o tempo necessário para iniciar uma

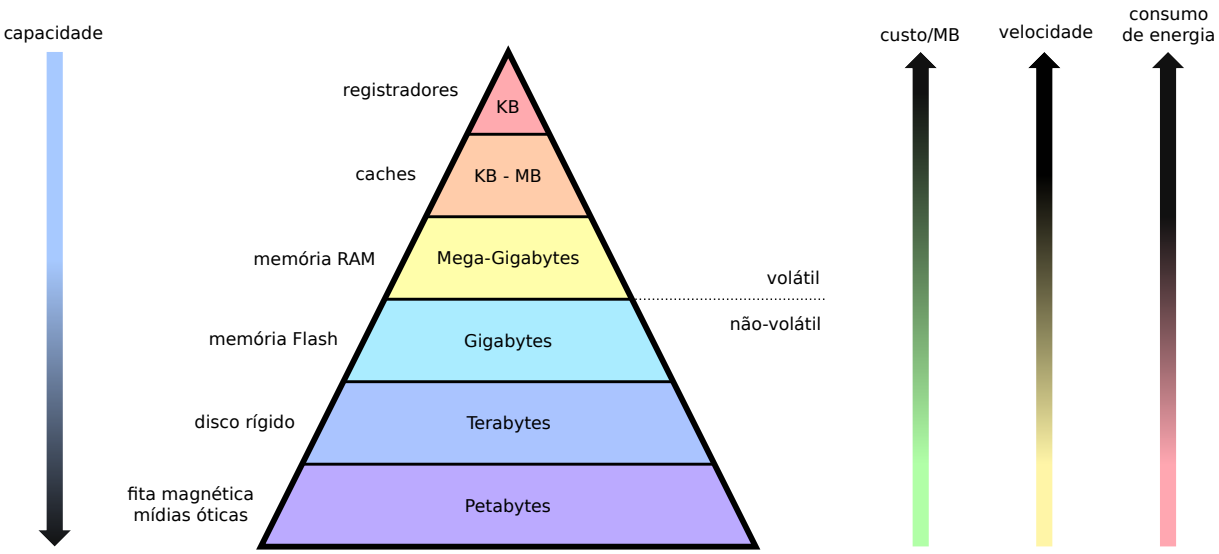


Figura 14.1: Hierarquia de memória.

transferência de dados de/para um determinado meio de armazenamento. Por sua vez, a taxa de transferência indica quantos bytes por segundo podem ser lidos/escritos naquele meio, uma vez iniciada a transferência de dados.

Para ilustrar esses dois conceitos complementares, a Tabela 14.1 traz valores de tempo de acesso e taxa de transferência típicos de alguns meios de armazenamento usuais.

Meio	**Tempo de acesso**	**Taxa de transferência**
Cache L2	1 ns	1 GB/s (1 ns/byte)
Memória RAM	60 ns	1 GB/s (1 ns/byte)
Memória *flash* (NAND)	2 ms	10 MB/s (100 ns/byte)
Disco rígido SATA	5 ms (tempo para o ajuste da cabeça de leitura e a rotação do disco até o setor desejado)	80 MB/s (12 ns/byte)
DVD-ROM	de 100 ms a vários minutos (caso a gaveta do leitor esteja aberta ou o disco não esteja no leitor)	10 MB/s (100 ns/byte)

Tabela 14.1: Tempos de acesso e taxas de transferência típicas [Patterson and Henessy, 2005].

Este e os próximos capítulos são dedicados aos mecanismos envolvidos na gerência da memória principal do computador, que geralmente é constituída por um grande espaço de memória do tipo RAM (*Random Access Memory*). Os mecanismos de gerência dos caches L1 e L2 geralmente são implementados em hardware e são independentes do sistema operacional. Detalhes sobre seu funcionamento podem ser obtidos em [Patterson and Henessy, 2005].

14.2 A memória de um processo

Cada processo é implementado pelo sistema operacional como uma "cápsula" de memória isolada dos demais processos, ou seja, uma área de memória exclusiva que só o próprio processo e o núcleo do sistema podem acessar. A área de memória do processo contém as informações necessárias à sua execução: código binário, variáveis, bibliotecas, buffers, etc. Essa área é dividida em seções ou segmentos, que são intervalos de endereços que o processo pode acessar. A lista de seções de memória de cada processo é mantida pelo núcleo, no descritor do mesmo.

As principais seções de memória de um processo são:

TEXT: contém o código binário a ser executado pelo processo, gerado durante a compilação e a ligação com as bibliotecas e armazenado no arquivo executável. Esta seção se situa no início do espaço de endereçamento do processo e tem tamanho fixo, calculado durante a compilação.

DATA: esta seção contém as variáveis estáticas inicializadas, ou seja, variáveis que estão definidas do início ao fim da execução do processo e cujo valor inicial é declarado no código-fonte do programa. Esses valores iniciais são armazenados no arquivo do programa executável, sendo então carregados para esta seção de memória quando o processo inicia. Nesta seção são armazenadas tanto variáveis globais quanto variáveis locais estáticas (por exemplo, declaradas como `static` em C).

BSS: historicamente chamada de *Block Started by Symbol*, esta seção contém as variáveis estáticas não-inicializadas. Esta seção é separada da seção `DATA` porque as variáveis inicializadas precisam ter seu valor inicial armazenado no arquivo executável, o que não é necessário para as variáveis não-inicializadas. Com essa separação de variáveis, o arquivo executável fica menor.

HEAP: esta seção é usada para armazenar variáveis alocadas dinamicamente, usando operadores como `malloc()`, `new()` e similares. O final desta seção é definido por um ponteiro chamado *Program Break*, ou simplesmente *break*, que pode ser ajustado através de chamadas de sistema para aumentar ou diminuir o tamanho da mesma.

STACK: esta seção é usada para manter a pilha de execução do processo, ou seja, a estrutura responsável por gerenciar o fluxo de execução nas chamadas de função e também para armazenar os parâmetros, variáveis locais e o valor de retorno das funções. Geralmente a pilha cresce "para baixo", ou seja, inicia em endereços maiores e cresce em direção aos endereços menores da memória. O tamanho total desta seção pode ser fixo ou variável, dependendo do sistema operacional.

Em programas com múltiplas *threads*, esta seção contém somente a pilha do programa principal. Como *threads* podem ser criadas e destruídas dinamicamente, a pilha de cada *thread* é mantida em uma seção de memória própria, alocada dinamicamente no *heap* ou em blocos de memória alocados na área livre para esse fim.

Cada uma dessas seções tem um conteúdo específico e por isso devem ser definidas permissões distintas para os acessos às mesmas. Por exemplo, a seção `TEXT` contém o código binário, que pode ser lido e executado, mas não deve ser modificado. As demais seções normalmente devem permitir acessos somente em leitura e escrita (sem execução), mas alguns programas que executam código interpretado ou com compilação dinâmica (como Java) podem gerar código dinamicamente e com isso necessitar de acessos em execução à pilha ou ao *heap*.

A Figura 14.2 apresenta a organização das seções de memória de um processo. Nela, observa-se que as duas seções de tamanho variável (*stack* e *heap*) estão dispostas em posições opostas e vizinhas à memória livre. Dessa forma, a memória livre disponível ao processo pode ser aproveitada da melhor forma possível, tanto pelo *heap* quanto pelo *stack*, ou por ambos.

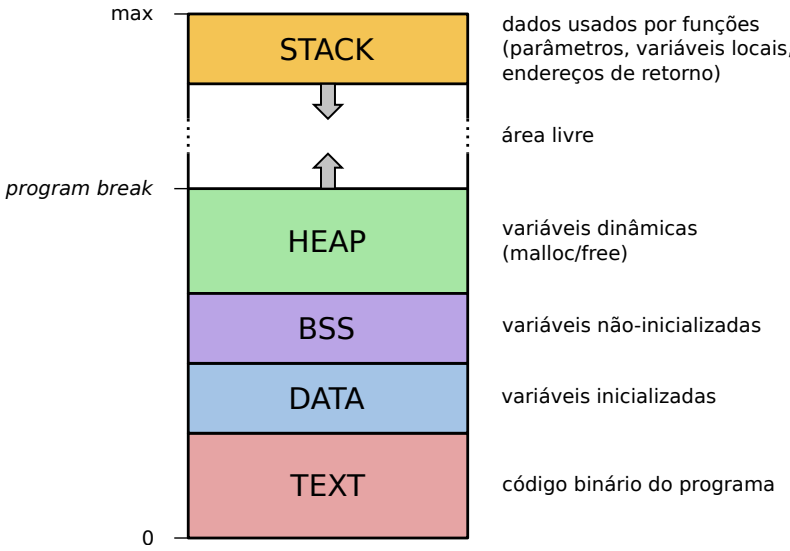


Figura 14.2: Organização da memória de um processo.

No sistema Linux, o comando `pmap` (*process map*) permite observar o mapa de memória de um processo, ou seja, a distribuição das seções de memória que o compõem. O exemplo a seguir ilustra o resultado (simplificado) desse comando aplicado a um processo que executa um simples *Hello World* em linguagem C (27505 é o PID, identificador numérico do processo):

```
# pmap -x 27505

27505:   /usr/bin/hello
Address           Kbytes     Mode    Mapping
0000000000400000     808     r-x--   /usr/bin/hello          (TEXT)
00000000006c9000      12     rw---   /usr/bin/hello          (DATA)
00000000006cc000       8     rw---   [ anon ]                (BSS)
000000000092e000     140     rw---   [ anon ]                (HEAP)
00007ffe6a5df000     132     rw---   [ stack ]               (STACK)
```

Na listagem acima, a coluna *Address* indica o endereço inicial da seção de memória, *Kbytes* indica seu tamanho, *Mode* indica suas permissões de acesso e *Mapping* indica o tipo de conteúdo da seção: `/usr/bin/hello` indica que o conteúdo da seção provém diretamente do arquivo indicado, `anon` indica uma seção *anônima* (que não está relacionada a nenhum arquivo em disco) e `stack` indica que a seção contém uma pilha.

14.3 Alocação de variáveis

Um programa em execução armazena suas informações em variáveis, que são basicamente espaços de memória nomeados. Por exemplo, uma declaração "`int soma`" em linguagem C indica uma área de memória de 4 bytes que pode armazenar um número inteiro e cujo nome é `soma`. Em linguagens como C e C++, variáveis podem ser alocadas na memória usando três abordagens usuais: a alocação *estática*, a alocação *automática* e a alocação *dinâmica*. Estas formas de alocação serão descritas a seguir.

14.3.1 Alocação estática

Na alocação estática, o espaço necessário para a variável é definido durante a compilação do programa. O espaço correspondente em memória RAM é reservado no início da execução do processo e mantido até o encerramento deste. Variáveis com alocação estática são alocadas na seção de memória `DATA`, se forem inicializadas no código-fonte, ou na seção `BSS`, caso contrário.

Na linguagem C, esta forma de alocação é usada para variáveis globais ou variáveis locais estáticas[1], como a variável `soma` no exemplo a seguir:

```
1  #include <stdio.h>
2
3  int soma = 0 ;
4
5  int main ()
6  {
7    int i ;
8
9    for (i=0; i<1000; i++)
10     soma += i ;
11   printf ("Soma de inteiros até 1000: %d\n", soma) ;
12
13   return (0) ;
14 }
```

14.3.2 Alocação automática

Por default, as variáveis definidas dentro de uma função (variáveis locais e parâmetros) são alocadas de forma automática na pilha de execução do programa (seção `STACK`). O espaço usado para armazenar essas variáveis é alocado quando a função é invocada e liberado quando a função termina, de forma transparente para o programador. Isso é o que ocorre por exemplo com a variável `i` no código anterior.

Se uma função for chamada recursivamente, as variáveis locais e parâmetros serão novamente alocados na pilha, em áreas distintas para cada nível de recursão. Isso permite preservar os valores das mesmas em cada um dos níveis. O exemplo a seguir permite observar a existência de múltiplas instâncias de variáveis locais em chamadas recursivas:

[1] Na linguagem C, variáveis locais estáticas são aquelas definidas como `static` dentro de uma função, que preservam seu valor entre duas invocações da mesma função.

```c
#include <stdio.h>

long int fatorial (int n)
{
  long int parcial ;

  printf ("inicio: n = %d\n", n) ;
  if (n < 2)
    parcial = 1 ;
  else
    parcial = n * fatorial (n - 1) ;
  printf ("final : n = %d, parcial = %ld\n", n, parcial) ;
  return (parcial) ;
}

int main ()
{
  printf ("Fatorial (4) = %ld\n", fatorial (4)) ;
  return 0 ;
}
```

A execução do código acima gera o resultado apresentado na listagem a seguir. Pode-se observar claramente que, durante as chamadas recursivas à função `fatorial(n)`, vários valores distintos para as variáveis `n` e `parcial` são armazenados na memória:

```
inicio: n = 4
inicio: n = 3
inicio: n = 2
inicio: n = 1
final : n = 1, parcial = 1
final : n = 2, parcial = 2
final : n = 3, parcial = 6
final : n = 4, parcial = 24
Fatorial (4) = 24
```

14.3.3 Alocação dinâmica

Na alocação dinâmica de memória, o processo requisita explicitamente blocos de memória para armazenar dados, os utiliza e depois os libera, quando não forem mais necessários (ou quando o programa encerrar). Esses blocos de memória são alocados na seção `HEAP`, que pode aumentar de tamanho para acomodar mais alocações quando necessário.

A requisição de blocos de memória dinâmicos é feita através de funções específicas, que retornam uma referência (ou ponteiro) para o bloco de memória alocado. Um exemplo de alocação e liberação de memória dinâmica na linguagem C pode ser visto no trecho de código a seguir:

```
1  char * prt ;                 // ponteiro para caracteres
2
3  ptr = malloc (4096) ;        // solicita um bloco de 4.096 bytes;
4                               // ptr aponta para o início do bloco
5
6  if (ptr == NULL)             // se ptr for nulo, ocorreu um erro
7    abort () ;                 // e a área não foi alocada
8
9  ...                          // usa ptr para acessar o bloco alocado
10
11 free (ptr) ;                 // libera o bloco alocado na linha 3
```

Alocações dinâmicas são muito usadas para armazenar objetos em linguagens orientadas a objetos. O trecho de código a seguir ilustra a criação dinâmica de objetos em Java:

```
1 Rectangle rect1 = new Rectangle (10, 30) ;
2 Rectangle rect2 = new Rectangle (3, 2) ;
3 Triangle tr1  = new Triangle (3, 4, 5) ;
```

A memória alocada dinamicamente por um processo é automaticamente liberada quando sua execução encerra. Contudo, pode ser necessário liberar blocos de memória dinâmicos sem uso durante uma execução, sobretudo se ela for longa, como um servidor Web ou um gerenciador de ambiente *desktop*. Programas que só alocam memória e não a liberam podem acabar consumindo toda a memória disponível no sistema, impedindo os demais programas de funcionar.

A liberação dos blocos de memória dinâmicos durante a execução pode ser manual ou automática, dependendo da linguagem de programação usada. Em linguagens mais simples, como C e C++, a liberação dos blocos de memória alocados deve ser feita pelo programador, usando funções como `free()` ou `delete()`. Linguagens mais sofisticadas, como Java, Python e C#, possuem um mecanismo de "coleta de lixo" (*garbage collection*) que automaticamente varre os blocos de memória alocados e libera os que não forem mais necessários [Wilson et al., 1995].

14.4 Atribuição de endereços

Ao escrever um programa usando uma linguagem de programação, como C, C++ ou Java, o programador usa nomes para referenciar entidades abstratas como variáveis, funções, parâmetros e valores de retorno. Com esses nomes, não há necessidade do programador definir ou manipular endereços de memória explicitamente. O trecho de código em C a seguir (`soma.c`) ilustra esse conceito; nele, são usados símbolos para referenciar posições na memória contendo dados (`i` e `soma`) ou trechos de código (`main`, `printf` e `exit`):

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

int soma = 0 ;

int main ()
{
   int i ;

   for (i=0; i< 10; i++)
   {
      soma += i ;
      printf ("i vale %d e soma vale %d\n", i, soma) ;
   }
   exit(0) ;
}
```

Todavia, o processador do computador precisa acessar endereços de memória para buscar as instruções a executar e seus operandos e para escrever os resultados do processamento dessas instruções. Por isso, quando programa `soma.c` for compilado, ligado a bibliotecas, carregado na memória e executado pelo processador, cada referência a uma variável, procedimento ou função no programa terá de ser transformada em um ou mais endereços específicos na área de memória do processo.

A listagem a seguir apresenta o código *Assembly* correspondente à compilação do programa em linguagem C `soma.c`. Nele, pode-se observar que não há mais referências a nomes simbólicos, apenas a endereços:

```
0000000000000000 <main>:
   0:  55                          push  %rbp
   1:  48 89 e5                    mov   %rsp,%rbp
   4:  48 83 ec 10                 sub   $0x10,%rsp
   8:  c7 45 fc 00 00 00 00        movl  $0x0,-0x4(%rbp)
   f:  eb 2f                       jmp   40 <main+0x40>
  11:  8b 15 00 00 00 00           mov   0x0(%rip),%edx
  17:  8b 45 fc                    mov   -0x4(%rbp),%eax
  1a:  01 d0                       add   %edx,%eax
  1c:  89 05 00 00 00 00           mov   %eax,0x0(%rip)
  22:  8b 15 00 00 00 00           mov   0x0(%rip),%edx
  28:  8b 45 fc                    mov   -0x4(%rbp),%eax
  2b:  89 c6                       mov   %eax,%esi
  2d:  bf 00 00 00 00              mov   $0x0,%edi
  32:  b8 00 00 00 00              mov   $0x0,%eax
  37:  e8 00 00 00 00              callq 3c <main+0x3c>
  3c:  83 45 fc 01                 addl  $0x1,-0x4(%rbp)
  40:  83 7d fc 04                 cmpl  $0x4,-0x4(%rbp)
  44:  7e cb                       jle   11 <main+0x11>
  46:  bf 00 00 00 00              mov   $0x0,%edi
  4b:  e8 00 00 00 00              callq 50 <main+0x50>
```

Dessa forma, os nomes simbólicos das variáveis e blocos de código usados por um programa devem ser traduzidos em endereços de memória em algum momento entre a escrita do código pelo programador e sua execução pelo processador. A atribuição de endereços aos nomes simbólicos pode ser dar em diversos momentos da vida do programa:

Durante a edição: o programador escolhe o endereço de cada uma das variáveis e do código do programa na memória. Esta abordagem normalmente só é usada na programação de sistemas embarcados simples, programados diretamente em *Assembly*.

Durante a compilação: ao traduzir o código-fonte, o compilador escolhe as posições das variáveis na memória. Para isso, todos os códigos-fontes necessários ao programa devem ser conhecidos no momento da compilação, para evitar conflitos de endereços entre variáveis em diferentes arquivos ou bibliotecas. Essa restrição impede o uso de bibliotecas precompiladas. Esta abordagem era usada em programas executáveis com extensão `.COM` do MS-DOS e Windows.

Durante a ligação: na fase de compilação, o compilador traduz o código fonte em código binário, mas não define os endereços das variáveis e funções, gerando como saída um *arquivo objeto* (*object file*)[2], que contém o código binário e uma *tabela de símbolos* descrevendo as variáveis e funções usadas, seus tipos, onde estão definidas e onde são usadas. A seguir, o ligador (*linker*) pega os arquivos objetos com suas tabelas de símbolos, define os endereços de memória dos símbolos e gera o programa executável [Levine, 2000].

Durante a carga: também é possível definir os endereços de variáveis e de funções durante a carga do código em memória para o lançamento de um novo processo. Nesse caso, um *carregador* (*loader*) é responsável por carregar o código do processo na memória e definir os endereços de memória que devem ser utilizados. O carregador pode ser parte do núcleo do sistema operacional ou uma biblioteca ligada ao executável, ou ambos. Esse mecanismo normalmente é usado na carga de bibliotecas dinâmicas (DLL - *Dynamic Linking Libraries*).

Durante a execução: os endereços emitidos pelo processador durante a execução do processo são analisados e convertidos nos endereços efetivos a serem acessados na memória real. Por exigir a análise e a conversão de cada endereço gerado pelo processador, este método só é viável com o auxílio do hardware.

A maioria dos sistemas operacionais atuais usa uma combinação de técnicas, envolvendo a tradução durante a ligação (para o código principal do programa e a construção de bibliotecas), durante a carga (para bibliotecas dinâmicas) e durante a execução (para todo o código). A tradução direta de endereço durante a edição ou compilação só é usada na programação de sistemas mais simples, como microcontroladores e sistemas embarcados. A Figura 14.3 ilustra os momentos de tradução de endereços acima descritos.

[2]Os arquivos com extensão `.o` em UNIX ou `.obj` em Windows são exemplos de arquivos-objeto obtidos da compilação de arquivos em C ou outra linguagem compilada.

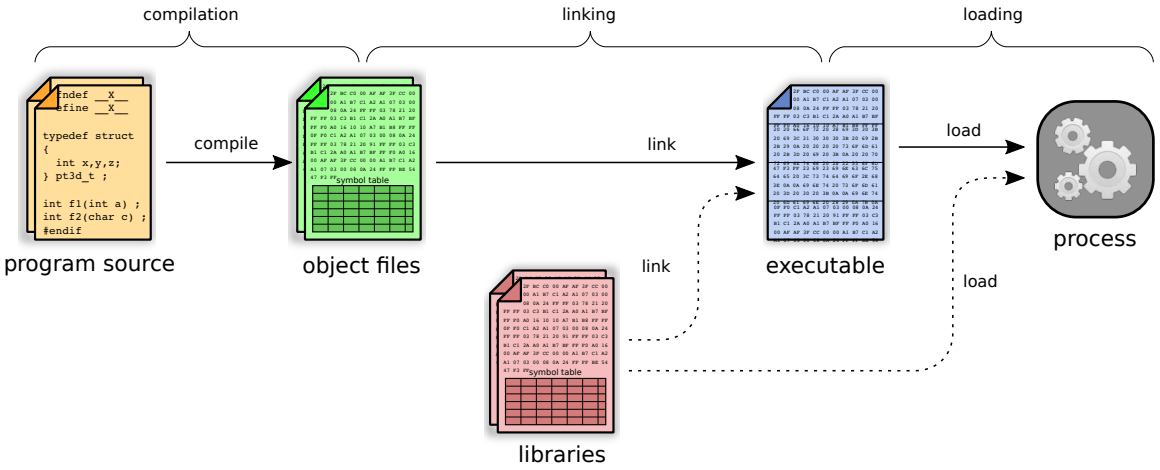


Figura 14.3: Momentos da tradução de endereços.

Referências

J. Levine. *Linkers and Loaders*. Morgan Kaufmann, 2000.

D. Patterson and J. Henessy. *Organização e Projeto de Computadores*. Campus, 2005.

P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Memory Management*, pages 1–116. Springer, 1995.

Capítulo 15

Hardware de memória

Neste capítulo serão apresentados os principais elementos de hardware que compõe o sistema de memória de um computador e os mecanismos básicos implementados pelo hardware e controlados pelo sistema operacional para a sua gerência.

15.1 A memória física

A memória principal do computador é composta por um grande conjunto de bytes, que é a menor unidade de memória usada pelo processador. Cada byte da memória RAM possui um endereço, que é usado para acessá-lo. Um computador convencional atual possui alguns GBytes de memória RAM, usados para conter o sistema operacional e os processos em execução, além de algumas áreas para finalidades específicas, como buffers de dispositivos de entrada/saída. A quantidade de memória RAM disponível em um computador constitui seu **espaço de memória física**.

A Figura 15.1 ilustra a organização (simplificada) da memória RAM de um computador PC atual com 16 GBytes de memória RAM instalados. Nessa figura, as áreas livres (*free RAM*) podem ser usadas pelo sistema operacional e as aplicações; as demais áreas têm finalidades específicas e geralmente só são acessadas pelo hardware e pelo sistema operacional, para gerenciar o computador e realizar operações de entrada/saída.

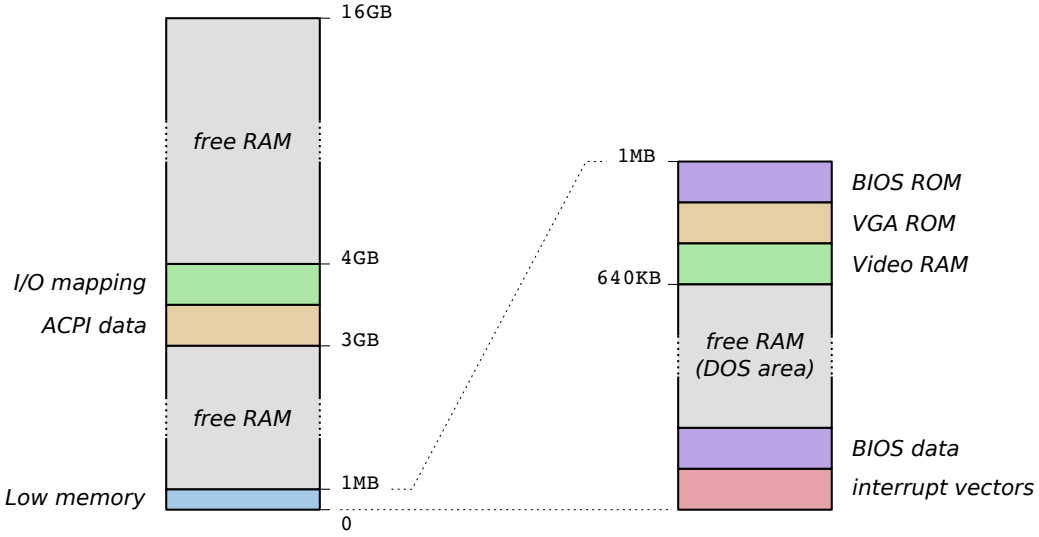


Figura 15.1: Layout da memória física de um computador.

Nos sistemas atuais, o layout da memória física apresentado na figura 15.1 não é visível ao usuário. Como regra geral, a execução de programas diretamente sobre a memória física é pouco usada, com exceção de sistemas muito simples, como em sistemas embarcados baseados em microcontroladores, ou muito antigos (como o MS-DOS – *Disk Operating System*). Os sistemas atuais mais sofisticados usam os conceitos de espaços de endereçamento e de memória virtual, vistos nas próximas seções, para desacoplar a visão da memória pelos processos da estrutura física da memória no hardware. Esse desacoplamento visa tornar mais simples e flexível o uso da memória pelos processos e pelo sistema operacional.

15.2 Espaço de endereçamento

O processador acessa a memória RAM através de barramentos de dados, de endereços e de controle. O barramento de endereços (como os demais) possui um número fixo de vias, que define a quantidade total de endereços de memória que podem ser gerados pelo processador: um barramento de dados com n vias consegue gerar 2^n endereços distintos (pois cada via define um bit do endereço), no intervalo $[0 \ldots 2^n - 1]$. Por exemplo, um processador Intel 80386 possui 32 vias de endereços, o que o permite acessar até 2^{32} bytes (4 GBytes) de memória, no intervalo $[0 \ldots 2^{32} - 1]$. Já um processador Intel Core i7 usa 48 vias para endereços e portanto pode endereçar até 2^{48} bytes, ou seja, 256 Terabytes de memória física. O conjunto de endereços de memória que um processador pode produzir é chamado de **espaço de endereçamento**.

É fundamental ter em mente que o espaço de endereçamento do processador é independente da quantidade de memória RAM disponível no sistema, podendo ser muito maior que esta. Assim, um endereço gerado pelo processador pode ser válido, quando existe um byte de memória RAM acessível naquele endereço, ou inválido, quando não há memória instalada naquele endereço. Dependendo da configuração da memória RAM, o espaço de endereçamento pode conter diversas áreas válidas e outras inválidas.

15.3 A memória virtual

Para ocultar a organização complexa da memória física e simplificar os procedimentos de alocação da memória aos processos, os sistemas de computação modernos implementam a noção de **memória virtual**, na qual existem dois tipos de endereços de memória **distintos**:

Endereços físicos (ou reais) são os endereços dos bytes de memória física do computador. Estes endereços são definidos pela quantidade de memória disponível na máquina, de acordo com o diagrama da figura 15.1.

Endereços lógicos (ou virtuais) são os endereços de memória usados pelos processos e pelo sistema operacional e, portanto, usados pelo processador durante a execução. Estes endereços são definidos de acordo com o espaço de endereçamento do processador.

Ao executar, os processos "enxergam" somente a memória virtual. Assim, durante a execução de um programa, o processador gera endereços lógicos para acessar

a memória. Esses endereços devem então ser traduzidos para os endereços físicos correspondentes na memória RAM, onde as informações desejadas se encontram. Por questões de desempenho, a tradução de endereços lógicos em físicos é feita por um componente específico do hardware do computador, denominado **Unidade de Gerência de Memória** (MMU – *Memory Management Unit*). Na maioria dos processadores atuais, a MMU se encontra integrada ao chip da própria CPU.

A MMU intercepta os endereços lógicos emitidos pelo processador e os traduz para os endereços físicos correspondentes na memória da máquina, permitindo então seu acesso pelo processador. Caso o acesso a um determinado endereço lógico não seja possível (por não estar associado a um endereço físico, por exemplo), a MMU gera uma interrupção de hardware para notificar o processador sobre a tentativa de acesso indevido. O comportamento da MMU e as regras de tradução de endereços são configurados pelo núcleo do sistema operacional.

O funcionamento básico da MMU está ilustrado na Figura 15.2. Observa-se que a MMU intercepta o acesso do processador ao barramento de endereços, recebendo os endereços lógicos gerados pelo mesmo e enviando os endereços físicos correspondentes ao barramento de endereços. Além disso, a MMU também tem acesso ao barramento de controle, para identificar operações de leitura e de escrita na memória.

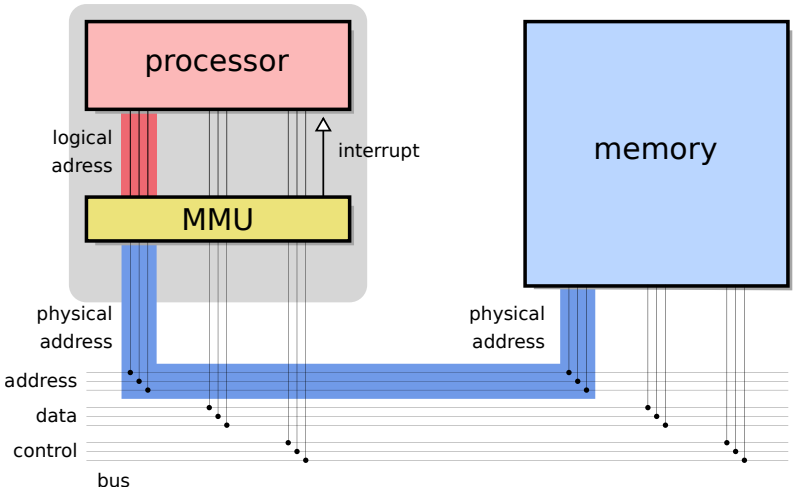


Figura 15.2: Funcionamento básico de uma MMU.

Além de desacoplar os endereços lógicos dos endereços físicos e realizar a tradução entre ambos, a noção de memória virtual também permite implementar a proteção de memória do núcleo e dos processos entre si, fundamentais para a segurança e estabilidade do sistema. Para implementar a proteção de memória entre processos, o núcleo mantém regras distintas de tradução de endereços lógicos para cada processo e reconfigura a MMU a cada troca de contexto. Assim, o processo em execução em cada instante tem sua própria área de memória e é impedido pela MMU de acessar áreas de memória dos demais processos.

Além disso, a configuração das MMUs mais sofisticadas inclui a definição de permissões de acesso às áreas de memória. Essa funcionalidade permite implementar as permissões de acesso às diversas áreas de cada processo (conforme visto na Seção 14.2), bem como impedir os processos de acessar áreas exclusivas do núcleo do sistema operacional.

Nas próximas seções serão estudadas as principais estratégias de tradução de endereços usadas pelas MMUs: por partições (usada nos primeiros sistemas de memória virtual), por segmentos e por páginas, usada nos sistemas atuais.

15.4 Memória virtual por partições

Uma das formas mais simples de organização da memória e tradução de endereços lógicos em físicos consiste em dividir a memória física em N partições, que podem ter tamanhos iguais ou distintos, fixos ou variáveis. Em cada partição da memória física é carregado um processo. O processo ocupando uma partição de tamanho T bytes terá um espaço de endereçamento com até T bytes, com endereços lógicos no intervalo $[0 \ldots T-1]$.

Na tradução de endereços lógicos em um esquema por partições, a MMU possui dois registradores: um *registrador base* (B), que define o endereço físico inicial da partição ativa[1], e um *registrador limite* (L), que define o tamanho em bytes dessa partição. O algoritmo implementado pela MMU é simples: cada endereço lógico e_l gerado pelo processador é comparado ao valor do registrador limite; caso seja menor que este ($e_l < L$), o endereço lógico é somado ao valor do registrador base, para a obtenção do endereço físico correspondente ($e_f = e_l + B$). Caso contrário ($e_l \geq L$), uma interrupção é gerada pela MMU indicando um endereço lógico inválido.

A Figura 15.3 apresenta uma visão geral dessa estratégia. Na Figura, o processo ativo, ocupando a partição 3, tenta acessar o endereço lógico 14.257. Esse endereço é válido, pois é inferior ao limite da partição (15.000). Em seguida o endereço lógico é somado à base da partição (41.000) para obter o endereço físico correspondente (55.257). A tabela de partições reside na memória RAM, sendo usada para atualizar os registradores de base e limite da MMU quando houver troca de contexto.

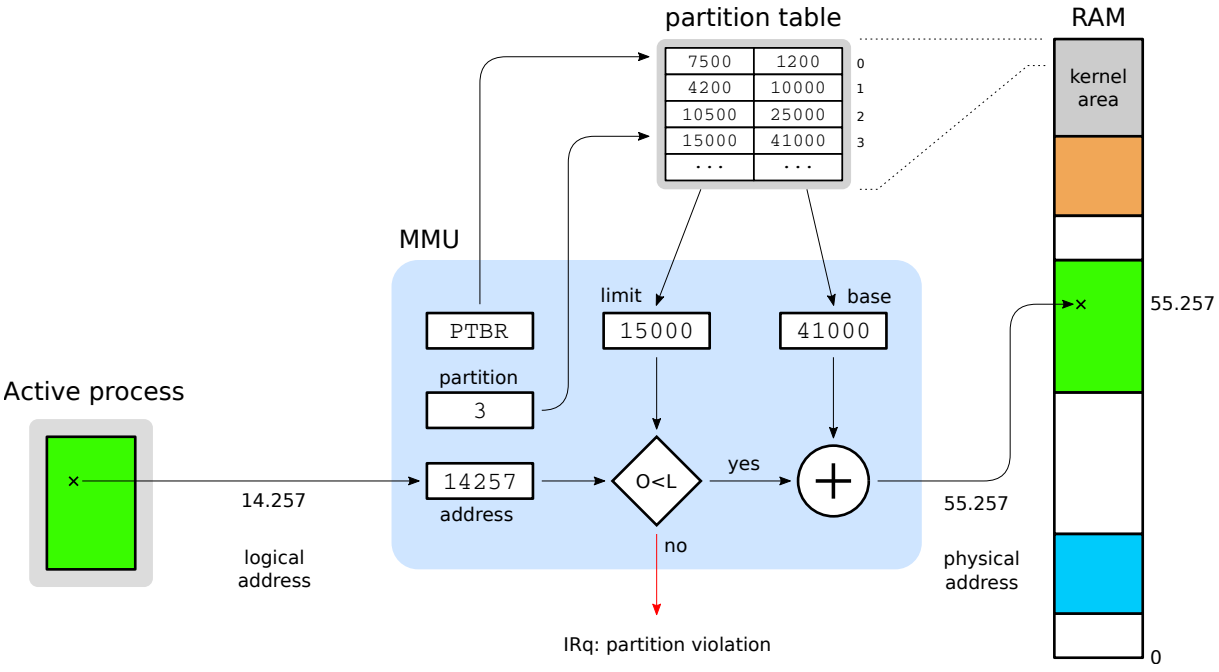


Figura 15.3: MMU com partições.

[1] A partição ativa é aquela onde se encontra o código em execução naquele instante.

Os valores dos registradores base e limite da MMU devem ser ajustados pelo núcleo sempre que for necessário trocar de espaço de endereçamento, ou seja, a cada troca de contexto. Os valores de base e limite para cada processo do sistema podem estar armazenados em uma tabela de partições ou no TCB do processo (*Task Control Block*, vide Seção 5.1). Quando o núcleo estiver executando, os valores de base e limite podem ser ajustados respectivamente para 0 e ∞, permitindo o acesso direto a toda a memória física.

Além de traduzir endereços lógicos nos endereços físicos correspondentes, a ação da MMU propicia a proteção de memória entre os processos: quando um processo p_i estiver executando, ele só pode acessar endereços lógicos no intervalo $[0 \ldots L(p_i) - 1]$, que correspondem a endereços físicos no intervalo $[B(p_i) \ldots B(p_i) + L(p_i) - 1]$. Ao detectar uma tentativa de acesso a um endereço lógico fora desse intervalo, a MMU irá gerar uma solicitação de interrupção (IRq - *Interrupt Request*, vide Seção 2.2.2) para o processador, sinalizado um acesso a endereço inválido. Ao receber a interrupção, o processador interrompe a execução do processo p_i, retorna ao núcleo e ativa a rotina de tratamento da interrupção, que poderá abortar o processo ou tomar outras providências.

A maior vantagem da estratégia de tradução por partições é sua simplicidade: por depender apenas de dois registradores e de uma lógica simples para a tradução de endereços, ela pode ser implementada em hardware de baixo custo, ou mesmo incorporada a processadores mais simples. Todavia, é uma estratégia pouco flexível e está sujeita a um fenômeno denominado *fragmentação externa*, que será discutido na Seção 16.3. Ela foi usada no OS/360, um sistema operacional da IBM usado nas décadas de 1960-70 [Tanenbaum, 2003].

15.5 Memória virtual por segmentos

A tradução por segmentos é uma extensão da tradução por partições, na qual as seções de memória do processo (`TEXT`, `DATA`, etc.) são mapeadas em áreas separadas na memória física. Além das seções funcionais básicas da memória do processo discutidas na Seção 14.2, também podem ser definidas áreas para itens específicos, como bibliotecas compartilhadas, vetores, matrizes, pilhas de *threads*, buffers de entrada/saída, etc.

Nesta abordagem, o espaço de endereçamento de cada processo não é mais visto como uma sequência linear de endereços lógicos, mas como uma coleção de áreas de tamanhos diversos e políticas de acesso distintas, denominadas *segmentos*. Cada segmento se comporta como uma partição de memória independente, com seus próprios endereços lógicos. A Figura 15.4 apresenta a visão lógica da memória de um processo e a sua forma de mapeamento para a memória física.

No modelo de memória virtual por segmentos, os endereços lógicos gerados pelos processos são compostos por pares [*segmento* : *offset*], onde *segmento* indica o número do segmento e *offset* indica a posição dentro daquele segmento. Os valores de *offset* em um segmento S variam no intervalo $[0 \ldots T(S) - 1]$, onde $T(S)$ é o tamanho do segmento. A Figura 15.4 mostra o endereço lógico [3 : 6.914], que corresponde ao *offset* 6.914 no segmento 3 do processo p_b. Nada impede de existir outros endereços 6.914 em outros segmentos do mesmo processo.

A tradução de endereços lógicos por segmentos é similar à tradução por partições. Contudo, como os segmentos são partições, cada segmento terá seus próprios valores de base e limite, o que leva à necessidade de definir uma *tabela de segmentos* para cada processo do sistema. Essa tabela contém os valores de base e limite para cada

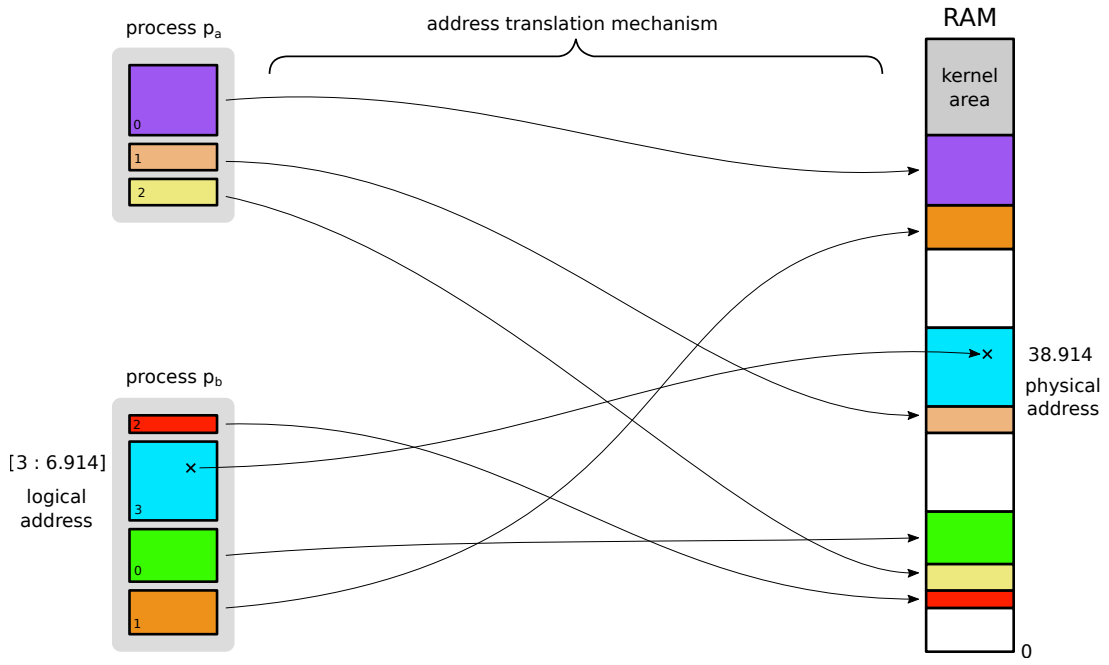


Figura 15.4: Memória virtual por segmentos.

segmento usado pelo processo, além de *flags* com informações sobre o segmento, como permissões de acesso, etc. (vide Seção 15.6.2). A MMU possui dois registradores para indicar a localização da tabela de segmentos ativa na memória RAM e seu tamanho: *STBR* (*Segment Table Base Register*) e *STLR* (*Segment Table Limit Register*). A Figura 15.5 apresenta os principais elementos envolvidos na tradução de endereços lógicos em físicos usando segmentos.

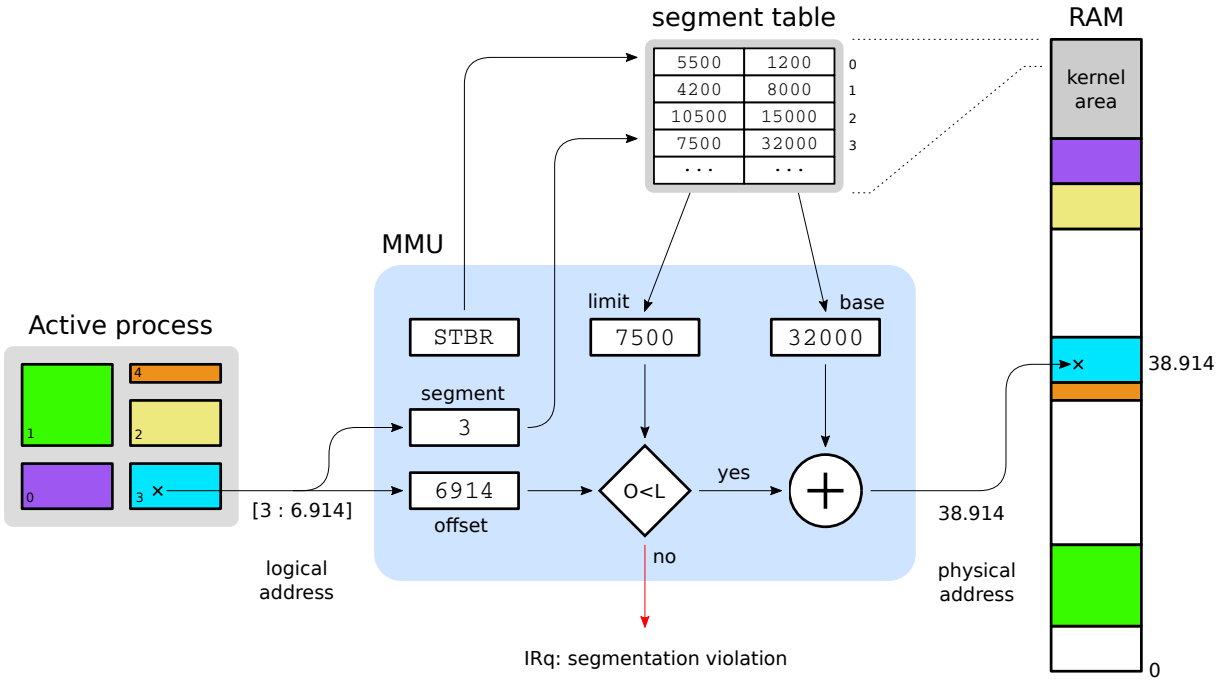


Figura 15.5: MMU com segmentação.

A implementação da tabela de segmentos varia conforme a arquitetura de hardware considerada. Caso o número de segmentos usados por cada processo seja pequeno, a tabela pode residir em registradores especializados do processador. Por outro lado, caso o número de segmentos por processo seja elevado, será necessário manter as tabelas na memória RAM. O processador Intel 80386 usa duas tabelas em RAM: a LDT (*Local Descriptor Table*), que define os segmentos locais (exclusivos) de cada processo, e a GDT (*Global Descriptor Table*), usada para descrever segmentos globais que podem ser compartilhados entre processos distintos (vide Seção 18.1). Cada uma dessas duas tabelas comporta até 8.192 segmentos. A cada troca de contexto, os registradores que indicam a tabela de segmentos ativa são atualizados para refletir as áreas de memória usadas pelo processo que será ativado.

Para cada endereço de memória acessado pelo processo em execução, é necessário acessar a tabela de segmentos para obter os valores de base e limite correspondentes ao endereço lógico acessado. Todavia, como as tabelas de segmentos normalmente se encontram na memória principal, esses acessos têm um custo significativo: considerando um sistema de 32 bits, para cada acesso à memória seriam necessárias pelo menos duas leituras adicionais na memória (para ler os valores de base e limite), o que tornaria cada acesso à memória três vezes mais lento. Para contornar esse problema, os processadores definem alguns *registradores de segmentos*, que permitem armazenar os valores de base e limite dos segmentos mais usados pelo processo ativo. Assim, caso o número de segmentos em uso simultâneo seja pequeno, não há necessidade de consultar a tabela de segmentos o tempo todo, o que mantém o desempenho de acesso à memória em um nível satisfatório. O processador Intel 80386 define os seguintes registradores de segmentos:

- **CS**: *Code Segment*, indica o segmento onde se encontra o código atualmente em execução; este valor é automaticamente ajustado no caso de chamadas de funções de bibliotecas, chamadas de sistema, interrupções ou operações similares.

- **SS**: *Stack Segment*, indica o segmento onde se encontra a pilha em uso pelo processo atual; caso o processo tenha várias threads, este registrador deve ser ajustado a cada troca de contexto entre threads.

- **DS**, **ES**, **FS** e **GS**: *Data Segments*, indicam quatro segmentos com dados usados pelo processo atual, que podem conter variáveis globais, vetores ou áreas de memória alocadas dinamicamente. Esses registradores podem ser ajustados em caso de necessidade, para acessar outros segmentos de dados.

O conteúdo desses registradores é preservado no TCB (*Task Control Block*) de cada processo a cada troca de contexto, tornando o acesso à memória bastante eficiente caso poucos segmentos sejam usados simultaneamente. Portanto, o compilador tem uma grande responsabilidade na geração de código executável: minimizar o número de segmentos necessários à execução do processo a cada instante, para não prejudicar o desempenho de acesso à memória.

O modelo de memória virtual por segmentos foi muito utilizado nos anos 1970-90, sobretudo nas arquiteturas Intel e AMD de 32 bits. Hoje em dia esse modelo é raramente utilizado em processadores de uso geral, sendo dada preferência ao modelo baseado em páginas (apresentado na próxima seção).

15.6 Memória virtual por páginas

Conforme visto na seção anterior, a organização da memória por segmentos exige o uso de endereços bidimensionais na forma [*segmento*:*offset*], o que é pouco intuitivo para o programador e torna mais complexa a construção de compiladores. Além disso, é uma forma de organização bastante suscetível à fragmentação externa, conforme será discutido na Seção 16.3. Essas deficiências levaram os projetistas de hardware a desenvolver outras técnicas para a organização da memória principal.

Na organização da memória por páginas, ou *memória paginada*, o espaço de endereçamento lógico dos processos é mantido linear e unidimensional. Internamente, de forma transparente para o processador, o espaço de endereçamento lógico é dividido em pequenos blocos de mesmo tamanho, denominados *páginas*. Nas arquiteturas atuais, as páginas geralmente têm 4 KBytes (4.096 bytes), mas podem ser encontradas arquiteturas com páginas de outros tamanhos[2]. A memória física também é dividida em blocos de mesmo tamanho que as páginas, denominados *quadros* (do inglês *frames*).

O mapeamento do espaço de endereçamento lógico na memória física é então feito simplesmente indicando em que quadro da memória física se encontra cada página, conforme ilustra a Figura 15.6. É importante observar que uma página pode estar em qualquer posição da memória física disponível, ou seja, pode estar associada a qualquer quadro, o que permite uma grande flexibilidade no uso da memória física.

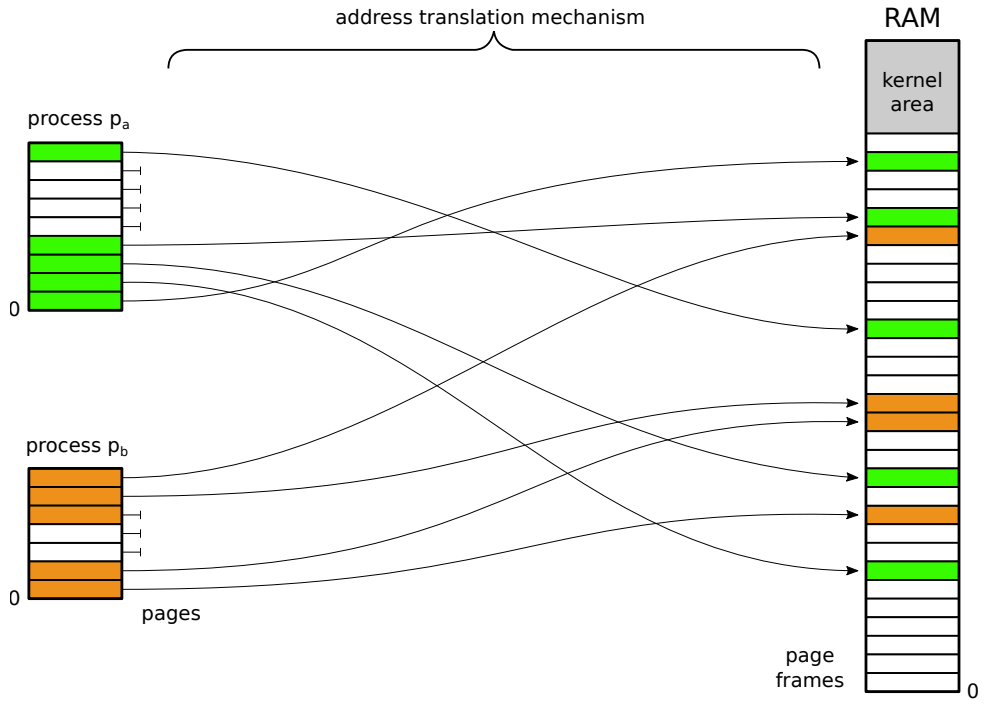


Figura 15.6: Organização da memória em páginas.

[2]As arquiteturas de processador mais recentes suportam diversos tamanhos de páginas, inclusive páginas muito grandes, as chamadas *superpáginas* (*hugepages*, *superpages* ou *largepages*). Uma superpágina tem geralmente entre 1 e 16 MBytes, ou mesmo acima disso; seu uso em conjunto com as páginas normais permite obter mais desempenho no acesso à memória, mas torna os mecanismos de gerência de memória mais complexos. O artigo [Navarro et al., 2002] traz uma discussão mais detalhada sobre esse tema.

15.6.1 A tabela de páginas

O mapeamento entre as páginas e os quadros correspondentes na memória física é feita através de *tabelas de páginas* (*page tables*), nas quais cada entrada corresponde a uma página do processo e contém o número do quadro onde ela se encontra. Cada processo possui sua própria tabela de páginas; a tabela de páginas ativa, que corresponde ao processo em execução no momento, é referenciada por um registrador da MMU denominado PTBR – *Page Table Base Register*. A cada troca de contexto, esse registrador deve ser atualizado com o endereço da tabela de páginas do novo processo ativo.

A divisão do espaço de endereçamento lógico de um processo em páginas pode ser feita de forma muito simples: como as páginas sempre têm 2^n bytes de tamanho (por exemplo, 2^{12} bytes para páginas de 4 KBytes) os n bits menos significativos de cada endereço lógico definem a posição daquele endereço dentro da página (deslocamento ou *offset*), enquanto os bits restantes (mais significativos) são usados para definir o número da página.

Por exemplo, o processador Intel 80386 usa endereços lógicos de 32 bits e páginas com 4 KBytes; um endereço lógico de 32 bits é decomposto em um *offset* de 12 bits, que representa uma posição entre 0 e 4.095 dentro da página, e um número de página com 20 bits. Dessa forma, podem ser endereçadas 2^{20} páginas com 2^{12} bytes cada (1.048.576 páginas com 4.096 bytes cada). Eis um exemplo de decomposição do endereço lógico $01803E9A_h$ nesse sistema[3]:

$$
\begin{aligned}
01803E9A_h \quad &\rightarrow \quad \overbrace{0000\ 0001\ 1000\ 0000\ 0011}^{page:\ 20\ bits}\ \overbrace{1110\ 1001\ 1010_2}^{offset:\ 12\ bits} \\
&\rightarrow \quad \overbrace{0000\ 0001\ 1000\ 0000\ 0011}^{page=01803_h}\ \overbrace{1110\ 1001\ 1010_2}^{offset=E9A_h} \\
&\rightarrow \quad page = 01803_h \quad offset = E9A_h
\end{aligned}
$$

Para traduzir um endereço lógico no endereço físico correspondente, a MMU efetua os seguintes passos, que são ilustrados na Figura 15.7:

1. decompor o endereço lógico em número de página e *offset*;
2. obter o número do quadro onde se encontra a página desejada;
3. construir o endereço físico, compondo o número do quadro com o *offset*; como páginas e quadros têm o mesmo tamanho, o valor do *offset* é preservado na conversão.

Pode-se observar que as páginas de memória não utilizadas pelo processo são representadas por entradas vazias na tabela de páginas e portanto não são mapeadas em quadros de memória física. Se o processo tentar acessar essas páginas, a MMU irá gerar uma interrupção de falta de página (*page fault*). Essa interrupção provoca o desvio da execução para o núcleo do sistema operacional, que deve então tratar a falta de página, abortando o processo ou tomando outra medida.

[3] A notação NNN_h indica um número em hexadecimal.

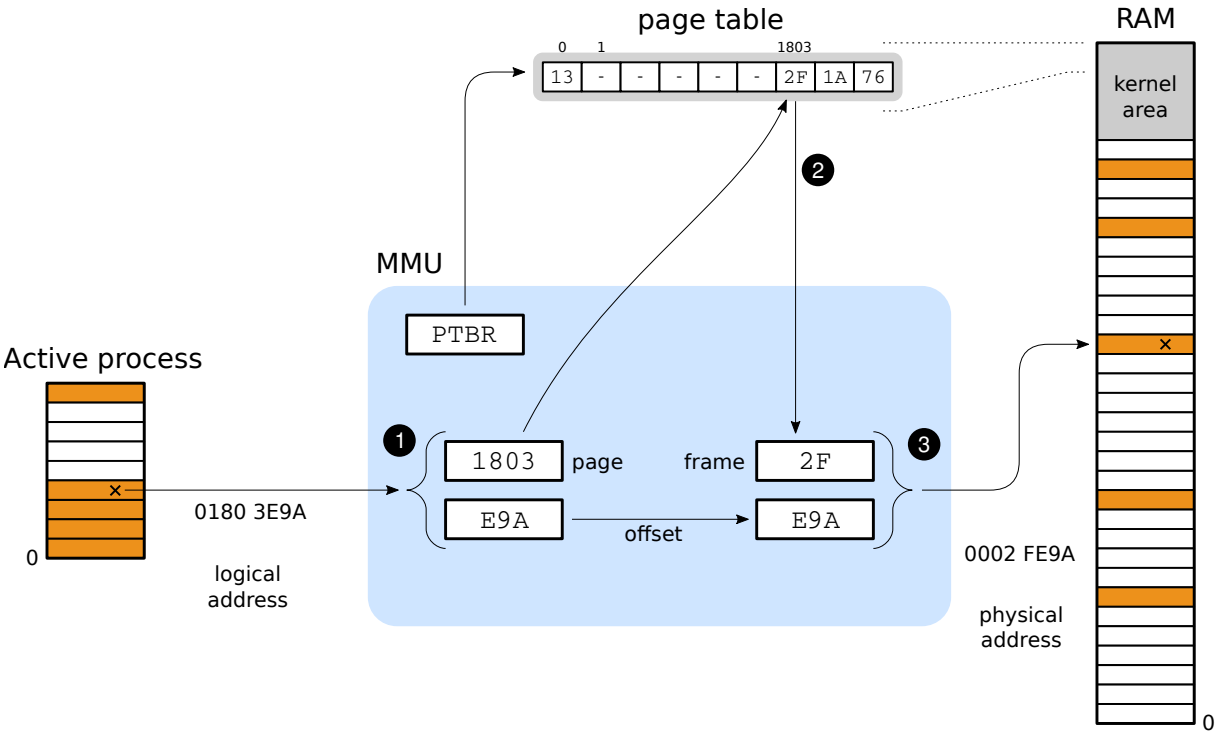


Figura 15.7: MMU com paginação.

15.6.2 Flags de status e controle

Além do número do quadro correspondente na memória física, cada entrada de uma tabela de páginas contém um conjunto de *flags* (bits) de status ou de controle relativos à página, com diversas finalidades. Os mais usuais são:

- *Valid*: indica se a página é válida, ou seja, se existe no espaço de endereçamento daquele processo; se este bit estiver em 0, tentativas de acesso à pagina irão gerar uma interrupção de falta de página (*page fault*);
- *Writable*: controla se a página pode ser acessada em leitura e escrita (1) ou somente em leitura (0);
- *User*: se estiver ativo (1), código executando em modo usuário pode acessar a página; caso contrário, a página só é acessível ao núcleo do sistema;
- *Present*: indica se a página está presente na memória RAM ou se foi transferida para um armazenamento secundário, como ocorre nos sistemas com paginação em disco (Seção 17.2);
- *Accessed*: indica se a página foi acessada recentemente; este bit é ativado pela MMU a cada acesso à página e pode ser desativado pelo núcleo quando desejado; essa informação é usada pelos algoritmos de paginação em disco;
- *Dirty*: este bit é ativado pela MMU após uma escrita na página, para informar que ela foi modificada (que foi "suja"); também é usado pelos algoritmos de paginação em disco.

Além destes, podem existir outros bits, indicando a política de *caching* aplicável à página, se a página pode ser movida para disco, o tamanho da página (no caso de sistemas que permitam mais de um tamanho de página), além de bits genéricos que podem ser usados pelos algoritmos do núcleo. O conteúdo exato de cada entrada da tabela de páginas depende da arquitetura do hardware considerado.

15.6.3 Tabelas multiníveis

Em uma arquitetura de 32 bits com páginas de 4 KBytes, cada entrada na tabela de páginas ocupa cerca de 32 bits, ou 4 bytes (20 bits para o número de quadro e os 12 bits restantes para informações e *flags* de controle). Considerando que cada tabela de páginas tem 2^{20} páginas, uma tabela ocupará 4 MBytes de memória (4×2^{20} bytes) se for armazenada de forma linear na memória. No caso de processos pequenos, com muitas páginas não mapeadas, uma tabela de páginas linear poderá ocupar mais espaço na memória que o próprio processo.

A Figura 15.8 mostra a tabela de páginas de um processo pequeno, com 100 páginas mapeadas no início de seu espaço de endereçamento (para as seções `TEXT`, `DATA` e `HEAP`) e 20 páginas mapeadas no final (para a seção `STACK`). Esse processo ocupa 120 páginas em RAM, ou 480 KBytes, enquanto sua tabela de páginas é quase 10 vezes maior, ocupando 4 MBytes. Além disso, a maior parte das entradas da tabela é vazia, ou seja, não aponta para quadros válidos.

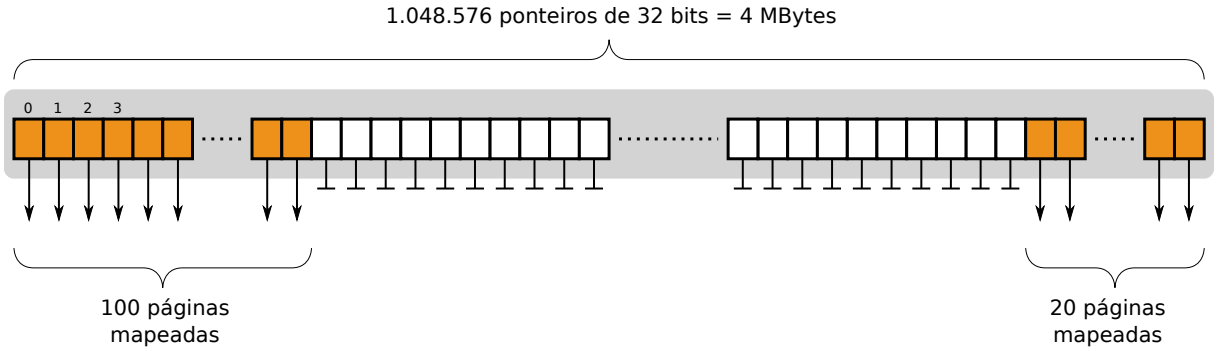


Figura 15.8: Tabela de páginas linear.

Para resolver esse problema, são usadas *tabelas de páginas multiníveis*, estruturadas na forma de árvores: uma primeira tabela de páginas (ou *diretório de páginas*) contém ponteiros para tabelas de páginas secundárias e assim por diante, até chegar à tabela que contém o número do quadro desejado. Quando uma tabela secundária não contiver entradas válidas, ela não precisa ser alocada; isso é representado por uma entrada nula na tabela principal.

A Figura 15.9 apresenta uma tabela de páginas com dois níveis que armazena as mesmas informações que a tabela linear da Figura 15.8, mas de forma muito mais compacta (12 KBytes ao invés de 4 MBytes). Cada sub-tabela contém 1.024 entradas.

Para percorrer essa árvore, o número de página precisa ser dividido em duas ou mais partes, que são usadas como índices em cada nível de tabela, até encontrar o número de quadro desejado. Um exemplo permite explicar melhor esse mecanismo: considerando uma arquitetura de 32 bits com páginas de 4 KBytes, 20 bits são usados para acessar a tabela de páginas. Esses 20 bits são divididos em dois grupos de 10 bits (p_1 e p_2) que são usados como índices em uma tabela de páginas com dois níveis:

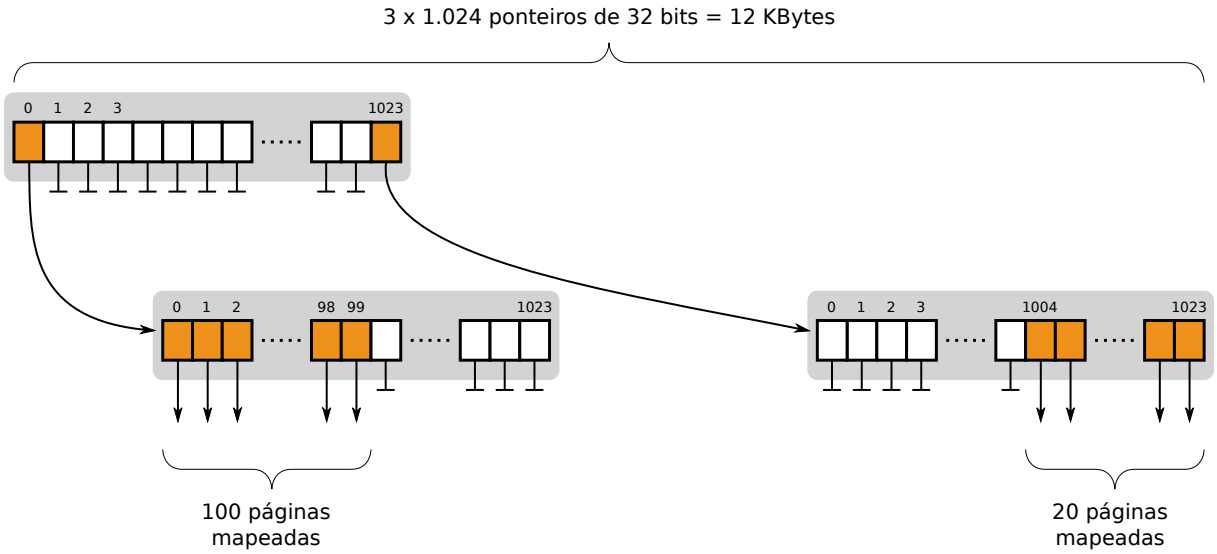


Figura 15.9: Tabela de páginas com dois níveis.

$$
\begin{aligned}
01803E9A_h \;\rightarrow\; & \overbrace{0000\ 0001\ 10}^{p_2\text{:10 bits}}\overbrace{00\ 0000\ 0011}^{p_1\text{:10 bits}}\overbrace{1110\ 1001\ 1010}^{\textit{offset}\text{:12bits}} \\
\rightarrow\; & \overbrace{0000\ 0001\ 10}^{p_2=0006_h}\overbrace{00\ 0000\ 0011}^{p_1=0003_h}\overbrace{1110\ 1001\ 1010}^{\textit{offset}=E9A_h} \\
\rightarrow\; & p_2 = 0006_h \quad p_1 = 0003_h \quad \textit{offset} = E9A_h
\end{aligned}
$$

A tradução de endereços lógicos em físicos usando uma tabela de páginas com dois níveis é efetuada através dos seguintes passos, que são ilustrados na Figura 15.10:

1. o endereço lógico 0180 $3E9A_h$ é decomposto em um *offset* de 12 bits ($E9A_h$) e dois números de página de 10 bits cada, que serão usados como índices: índice do nível externo p_2 (006_h) e o índice do nível interno p_1 (003_h);

2. o índice p_2 é usado como índice na tabela de páginas externa, para encontrar o endereço de uma tabela de páginas interna;

3. em seguida, o índice p_1 é usado na tabela de páginas interna indicada por p_2, para encontrar a entrada contendo o número de quadro ($2F_h$) que corresponde a $[p_2 p_1]$;

4. o número de quadro é combinado ao *offset* para obter o endereço físico (0002 $FE9A_h$) correspondente ao endereço lógico solicitado.

A estruturação da tabela de páginas em níveis reduz significativamente a quantidade de memória necessária para armazená-la, sobretudo no caso de processos pequenos. As Figuras 15.8 e 15.9 evidenciam essa redução, de 4 MBytes para 12 KBytes. Por outro lado, se um processo ocupar todo o seu espaço de endereçamento, seriam

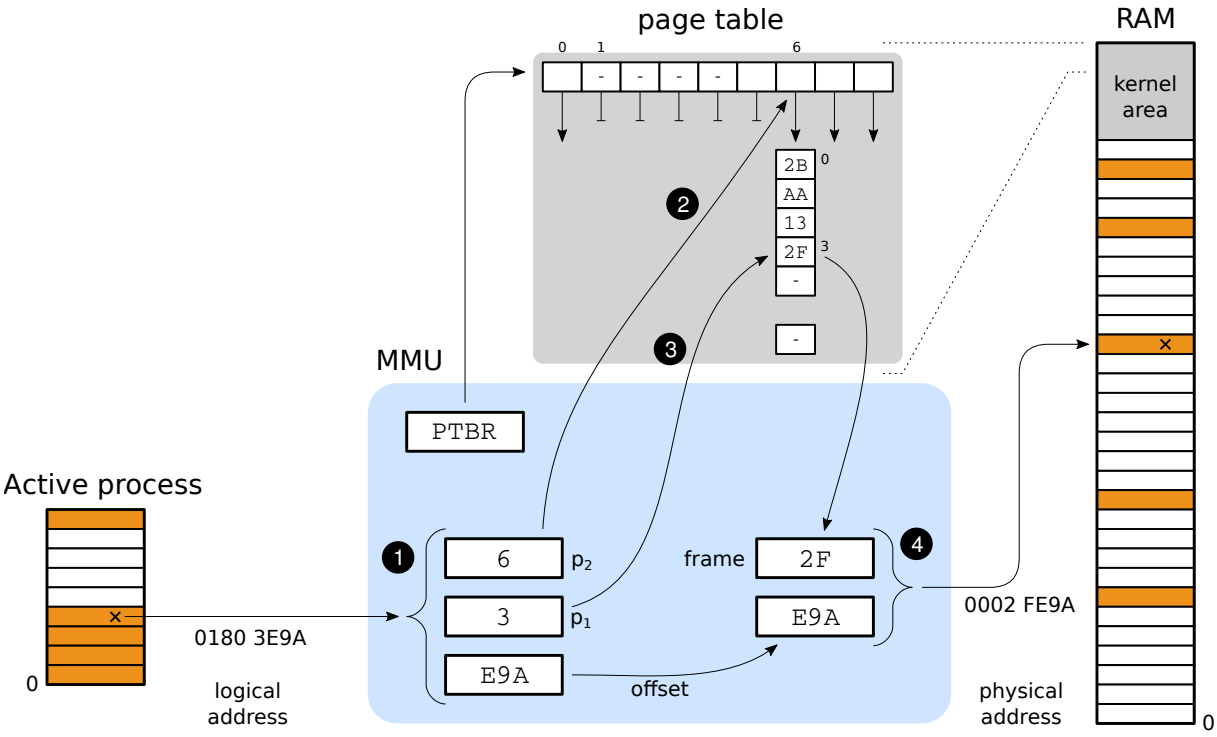


Figura 15.10: MMU com paginação multinível.

necessárias uma tabela de primeiro nível e 1.024 tabelas de segundo nível, que ocupariam $(1 + 1.024) \times 4KB$, ou seja, 0,098% a mais que se a tabela linear ($1.024 \times 4KB$).

O número de níveis da tabela de páginas depende da arquitetura considerada: processadores *Intel 80386* usam tabelas com dois níveis, cada tabela com 1.024 entradas de 4 bytes. Processadores de 64 bits mais recentes, como o *Intel Core i7*, usam tabelas com 4 níveis, cada tabela contendo 512 entradas de 8 bytes. Em ambos os casos, cada subtabela ocupa exatamente uma página de 4 KBytes.

15.6.4 Cache da tabela de páginas

A estruturação das tabelas de páginas em vários níveis resolve o problema do espaço ocupado pelas tabelas, mas tem um efeito colateral nocivo: aumenta fortemente o tempo de acesso à memória. Como as tabelas de páginas são armazenadas na memória RAM, cada acesso a um endereço de memória implica em mais acessos para percorrer a árvore de tabelas e encontrar o número de quadro desejado. Em um sistema com tabelas de dois níveis, cada acesso à memória solicitado pelo processador implica em mais dois acessos, para percorrer os dois níveis de tabelas. Com isso, o tempo efetivo de acesso à memória se torna três vezes maior.

Para atenuar esse problema, consultas recentes à tabela de páginas podem ser armazenadas em um *cache* dentro da própria MMU, evitando ter de repeti-las e assim diminuindo o tempo de acesso à memória RAM. O cache de tabela de páginas na MMU, denominado TLB (*Translation Lookaside Buffer*) ou *cache associativo*, armazena pares [*página*, *quadro*] obtidos em consultas recentes às tabelas de páginas do processo ativo. Esse cache funciona como uma tabela de *hash*: dado um número de página p em sua entrada, ele apresenta em sua saída o número de quadro q correspondente, ou um erro, caso não contenha informação sobre p.

A tradução de endereços lógicos em físicos usando TLB se torna mais rápida, mas também mais complexa. Os seguintes passos são necessários, ilustrados na Figura 15.11, são necessários:

1. A MMU decompõe o endereço lógico em números de página e *offset*;
2. a MMU consulta os números de página em seu cache TLB;
3. caso o número do quadro correspondente seja encontrado (*TLB hit*), ele é usado para compor o endereço físico;
4. caso contrário (*TLB miss*), uma busca completa na tabela de páginas deve ser realizada para obter o número do quadro (passos 4-6);
7. o número de quadro obtido é usado para compor o endereço físico;
8. o número de quadro é adicionado ao TLB para agilizar as próximas consultas.

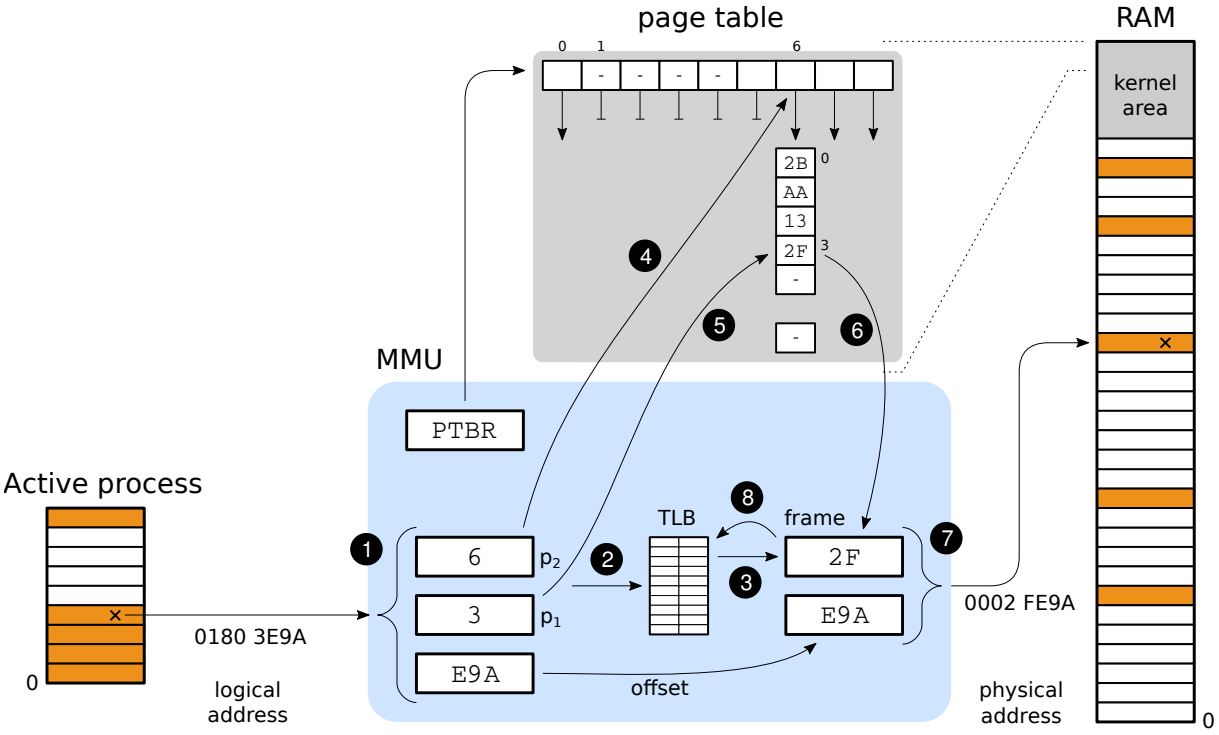


Figura 15.11: MMU com paginação e TLB.

O benefício do TLB pode ser estimado através do cálculo do tempo médio de acesso à memória, que é a média ponderada entre o tempo de acesso com acerto de TLB (*hit*) e o tempo de acesso com erro (*miss*). Deve-se observar, entretanto, que o uso do TLB também adiciona custos em tempo: um *TLB hit* custa cerca de 1 ciclo de relógio da CPU, enquanto um *TLB miss* pode custar entre 10 e 30 ciclos (incluindo o custo para atualizar o TLB).

Considerando como exemplo um sistema operando a 2 GHz (relógio de 0,5 ns) com tempo de acesso à memória RAM de 50 *ns*, tabelas de páginas com 3 níveis e um TLB com custo de acerto de 0,5 *ns* (um ciclo de relógio), custo de erro de 10 *ns* (20 ciclos

de relógio) e uma taxa de acerto de TLB de 95%, o tempo médio de acesso à memória pode ser estimado como segue:

$$\begin{aligned} t_{médio} &= 95\% \times 0,5ns && \text{\# custo do acerto no TLB} \\ &+ 5\% \times 10ns && \text{\# custo do erro no TLB} \\ &+ 5\% \times 3 \times 50ns && \text{\# custo da consulta às 3 tabelas} \\ &+ 50ns && \text{\# custo do acesso ao quadro} \\ t_{médio} &= 58,475ns \end{aligned}$$

Este resultado indica que o sistema de paginação multinível aumenta em 8,475 *ns* (16,9%) o tempo de acesso à memória, o que é razoável considerando-se os benefícios e flexibilidade que esse sistema traz. Todavia, esse custo é muito dependente da taxa de acerto do TLB: no cálculo anterior, caso a taxa de acerto caísse a 90%, o custo adicional seria de 32,9%; caso a taxa subisse a 99%, o custo adicional cairia para 4,2%.

Percebe-se então que, quanto maior a taxa de acertos do TLB (*TLB hit ratio*), melhor é o desempenho dos acessos à memória física. A taxa de acertos de um TLB é influenciada por diversos fatores:

Tamanho do TLB: quanto mais entradas houverem no TLB, melhor será sua taxa de acerto. Contudo, trata-se de um hardware caro e volumoso, por isso os processadores atuais geralmente têm TLBs com poucas entradas (geralmente entre 16 e 256 entradas). Por exemplo, a arquitetura *Intel 80386* tinha um TLB com 64 entradas para páginas de dados e 32 entradas para páginas de código; por sua vez, o *Intel Core i7* possui 96 entradas para páginas de dados e 142 entradas para páginas de código.

Padrão de acessos à memória: processos que concentram seus acessos em poucas páginas de cada vez farão um uso eficiente do TLB, enquanto processos que acessam muitas páginas distintas em um curto período irão gerar frequentes erros de TLB, prejudicando seu desempenho no acesso à memória. Essa propriedade é conhecida como *localidade de referência* e será abordada na Seção 15.9.

Trocas de contexto: o conteúdo do TLB reflete a tabela de páginas do processo ativo em um dado momento. A cada troca de contexto, a tabela de páginas é substituída e portanto o TLB deve ser esvaziado, pois seu conteúdo não é mais válido. Em consequência, trocas de contexto muito frequentes prejudicam a eficiência de acesso à memória, tornando o sistema mais lento.

Política de substituição de entradas: o que ocorre quando há um erro de TLB e não há mais entradas livres no TLB? Em alguns processadores, a associação [*página*, *quadro*] que gerou o erro é adicionada ao TLB, substituindo a entrada mais antiga; todavia, na maioria dos processadores mais recentes, cada erro de TLB provoca uma interrupção, que transfere ao sistema operacional a tarefa de gerenciar o conteúdo do TLB [Patterson and Henessy, 2005].

15.7 Segmentos e páginas

Cada uma das principais formas de organização de memória vistas até agora tem suas vantagens: a organização por partições prima pela simplicidade e rapidez;

a organização por segmentos oferece múltiplos espaços de endereçamento para cada processo, oferecendo flexibilidade ao programador; a organização por páginas oferece um grande espaço de endereçamento linear, enquanto elimina a fragmentação externa.

Vários processadores permitem combinar mais de uma forma de organização. Por exemplo, os processadores *Intel x86* permitem combinar a organização por segmentos com a organização por páginas, visando oferecer a flexibilidade dos segmentos com a baixa fragmentação das páginas. Nessa abordagem, os processos veem a memória estruturada em segmentos, conforme indicado na Figura 15.4. A MMU inicialmente converte os endereços lógicos na forma [*segmento*:*offset*] em endereços lógicos lineares (unidimensionais), usando as tabelas de descritores de segmentos (Seção 15.5). Em seguida, converte esse endereços lógicos lineares nos endereços físicos correspondentes, usando as tabelas de páginas.

Apesar do processador *Intel x86* oferecer as duas formas de organização de memória, a maioria dos sistemas operacionais que o suportam não fazem uso de todas as suas possibilidades: os sistemas da família Windows NT (2000, XP, Vista) e também os da família UNIX (Linux, FreeBSD) usam somente a organização por páginas. O antigo DOS e o Windows 3.* usavam somente a organização por segmentos. O OS/2 da IBM foi um dos poucos sistemas operacionais comerciais a fazer uso pleno das possibilidades de organização de memória nessa arquitetura, combinando segmentos e páginas.

15.8 Espaço de endereçamento de um processo

Na maioria dos sistemas atuais, o espaço de endereços virtuais de cada processo é organizado da seguinte forma: a parte inicial dos endereços é reservada para uso do processo, enquanto a parte final é reservada para o núcleo do sistema operacional. A Figura 15.12 ilustra a organização usual do espaço de endereçamento de cada processo em sistemas Windows e Linux de 32 e 64 bits [Love, 2010; Russinovich et al., 2008]. No sistema Linux 32 bits, por exemplo, os endereços iniciais (faixa `0x00000000-0xbfffffff`, com 3 GB) são usados pelo processo e os endereços finais (faixa `0xc0000000-0xffffffff`, com 1 GB) são usados pelo núcleo do sistema.

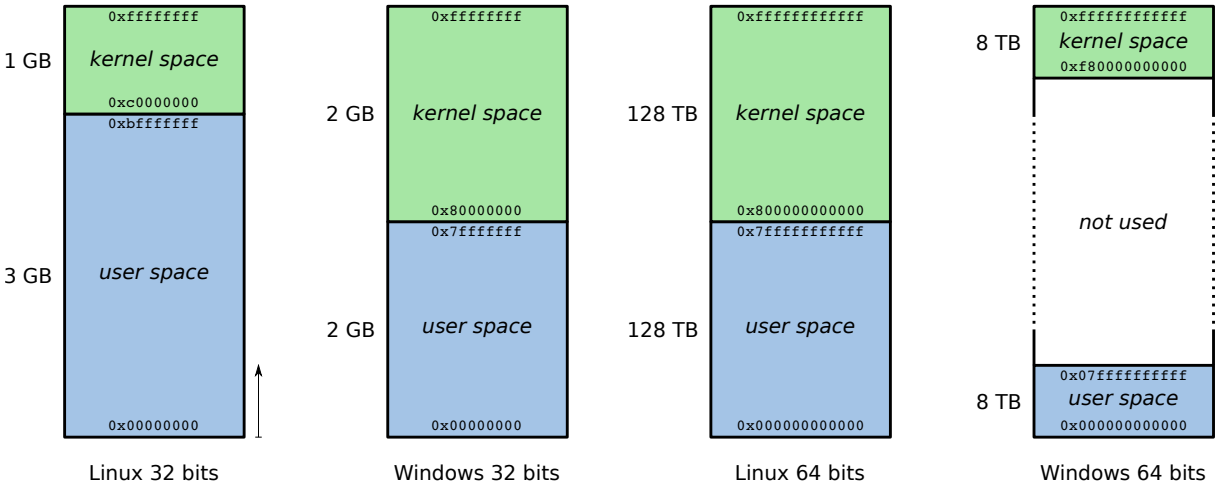


Figura 15.12: Organização do espaço de endereçamento em SOs atuais.

Observa-se que o núcleo faz parte do espaço de endereçamento de cada processo, mas não é acessível ao mesmo, por conta dos *flags* de controle da tabela de páginas

vistos na seção 15.6.2. As páginas do processo são marcadas com o flag *user*, enquanto as páginas do núcleo não o são. Com isso, o código no espaço do usuário não pode acessar as páginas do núcleo, mas o código do núcleo pode acessar as páginas do processo.

Com esse layout de endereços, as páginas do núcleo são mapeadas em todos os processos nas mesmas posições, o que otimiza o uso do cache TLB, pois os endereços das páginas importantes do núcleo são sempre mantidos no TLB. Dessa forma, a execução de uma chamada de sistema não exige a mudança dos mapas de memória ativos nem o esvaziamento do cache TLB, o que contribui para um bom desempenho.

O bug de segurança *Meltdown*, descoberto em 2018 nos mecanismos de memória virtual dos processadores Intel e ARM [Lipp et al., 2018], levou à revisão da organização do espaço de endereçamento virtual dos processos nos sistemas operacionais mais usados. Esse bug permite a um código de usuário ler partes da memória do núcleo, possivelmente expondo dados sensíveis lidos/escritos por outros processos, como senhas ou chaves de criptografia. A solução encontrada pelos principais SOs foi separar completamente os espaços de endereçamento dos processos e do núcleo, em tabelas de páginas independentes.

15.9 Localidade de referências

A forma como os processos acessam a memória tem um impacto direto na eficiência dos mecanismos de gerência de memória, sobretudo os caches de memória física, o cache da tabela de páginas (TLB, Seção 15.6.4) e o mecanismo de paginação em disco (Capítulo 17). Processos que concentram seus acessos em poucas páginas de cada vez farão um uso eficiente desses mecanismos, enquanto processos que acessam muitas páginas distintas em um curto período irão gerar frequentes erros de cache, de TLB e faltas de página, prejudicando seu desempenho no acesso à memória.

A propriedade de um processo ou sistema concentrar seus acessos em poucas áreas da memória a cada instante é chamada *localidade de referências* [Denning, 2006]. Existem ao menos três formas de localidade de referências:

Localidade temporal: um recurso usado há pouco tempo será provavelmente usado novamente em um futuro próximo;

Localidade espacial: um recurso será mais provavelmente acessado se outro recurso próximo a ele já foi acessado;

Localidade sequencial: é um caso particular da localidade espacial, no qual há uma predominância de acesso sequencial aos recursos (útil na otimização de sistemas de arquivos).

A Figura 15.13 ilustra o conceito de localidade de referências. Ela mostra as páginas acessadas durante uma execução do visualizador gráfico *gThumb*, ao abrir um arquivo de imagem JPEG. O gráfico da esquerda dá uma visão geral da distribuição dos acessos na memória, enquanto o gráfico da direita detalha os acessos da parte inferior, que corresponde às seções de código, dados e *heap* do processo. Nessa execução, pode-se observar que os acessos à memória em cada momento da execução são concentrados em certas áreas do espaço de endereçamento. Quanto maior a concentração de acessos em poucas áreas, melhor a localidade de referências de um programa.

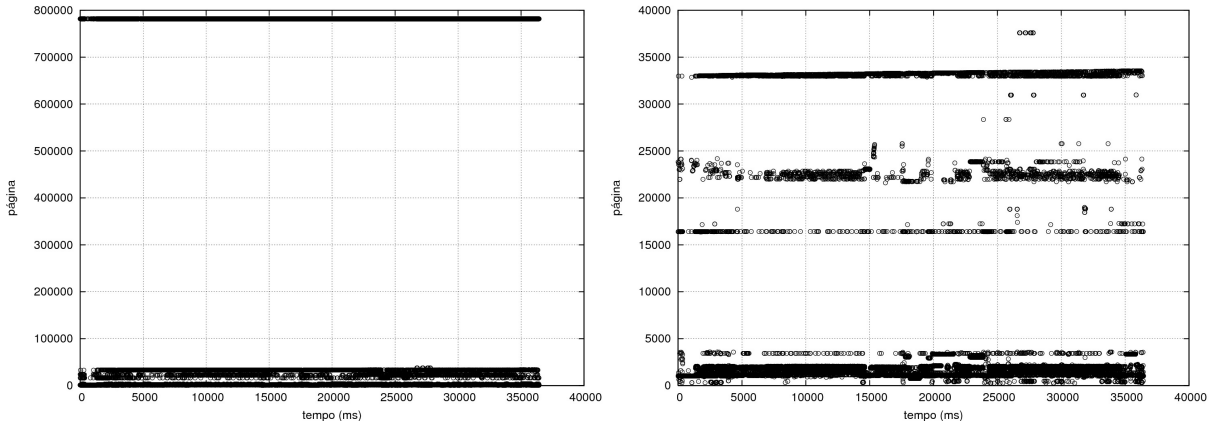


Figura 15.13: Distribuição dos acessos à memória do programa *gThumb*: visão geral (à esquerda) e detalhe da parte inferior (à direita).

Como exemplo da importância da localidade de referências, consideremos um programa para o preenchimento de uma matriz de 4.096 × 4.096 bytes, onde cada linha da matriz está alocada em uma página distinta (considerando páginas de 4.096 bytes). O trecho de código a seguir implementa essa operação, percorrendo a matriz linha por linha:

```
1  unsigned char buffer[4096][4096] ;
2
3  int main ()
4  {
5     int i, j ;
6
7     for (i=0; i<4096; i++)              // percorre as linhas do buffer
8        for (j=0; j<4096; j++)           // percorre as colunas do buffer
9           buffer[i][j]= (i+j) % 256 ;   // preenche com algum valor
10 }
```

Também é possível preencher a matriz percorrendo-a coluna por coluna:

```
1  unsigned char buffer[4096][4096] ;
2
3  int main ()
4  {
5     int i, j ;
6
7     for (j=0; j<4096; j++)              // percorre as colunas do buffer
8        for (i=0; i<4096; i++)           // percorre as linhas do buffer
9           buffer[i][j]= (i+j) % 256 ;   // preenche com algum valor
10 }
```

Embora percorram a matriz de forma distinta, os dois programas são equivalentes e geram o mesmo resultado. Entretanto, eles não têm o mesmo desempenho: a primeira implementação (percurso linha por linha) usa de forma eficiente o cache da tabela de páginas, porque só gera um erro de cache a cada nova linha acessada. Por outro lado, a implementação com percurso por colunas gera um erro de cache TLB a cada célula acessada, pois o cache TLB não tem tamanho suficiente para armazenar as

4.096 entradas referentes às páginas usadas pela matriz. A Figura 15.14 mostra o padrão de acesso à memória dos dois programas.

A diferença de desempenho entre as duas implementações pode ser grande: em processadores *Intel* e AMD, versões 32 e 64 bits, o primeiro código executa cerca de 5 vezes mais rapidamente que o segundo! Além disso, caso o sistema não tenha memória suficiente para manter as 4.096 páginas em memória, o mecanismo de memória virtual será ativado, fazendo com que a diferença de desempenho seja muito maior.

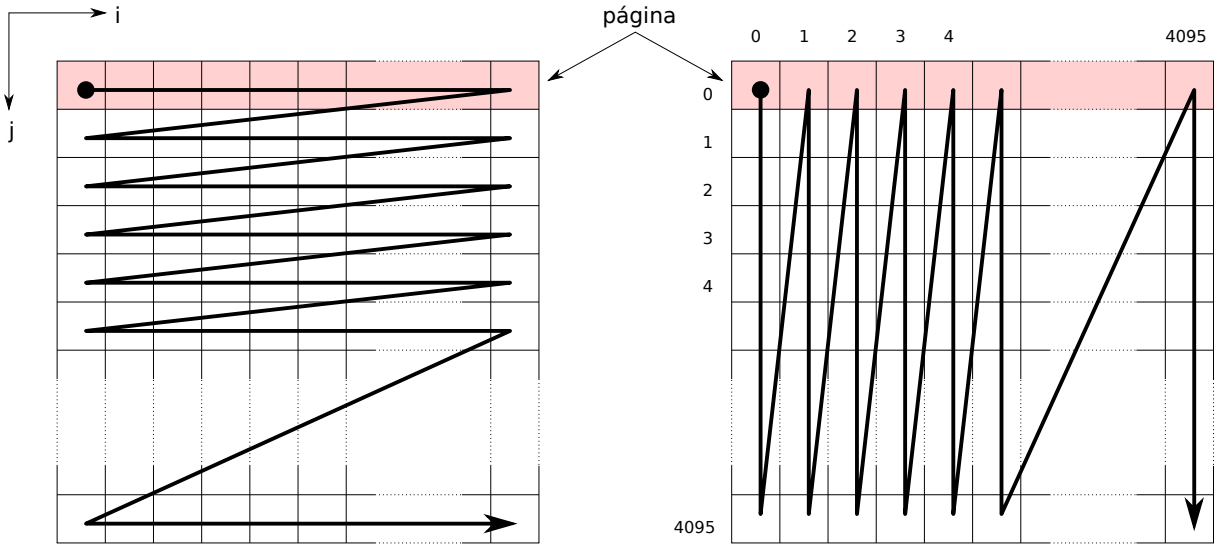


Figura 15.14: Comportamento dos programas no acesso à memória.

A diferença de comportamento das duas execuções pode ser observada na Figura 15.15, que mostra a distribuição dos endereços de memória acessados pelos dois códigos[4]. Nos gráficos, percebe-se claramente que a primeira implementação tem uma localidade de referências muito melhor que a segunda: enquanto a primeira execução usa em média 5 páginas distintas em cada 100.000 acessos à memória, na segunda execução essa média sobe para 3.031 páginas distintas.

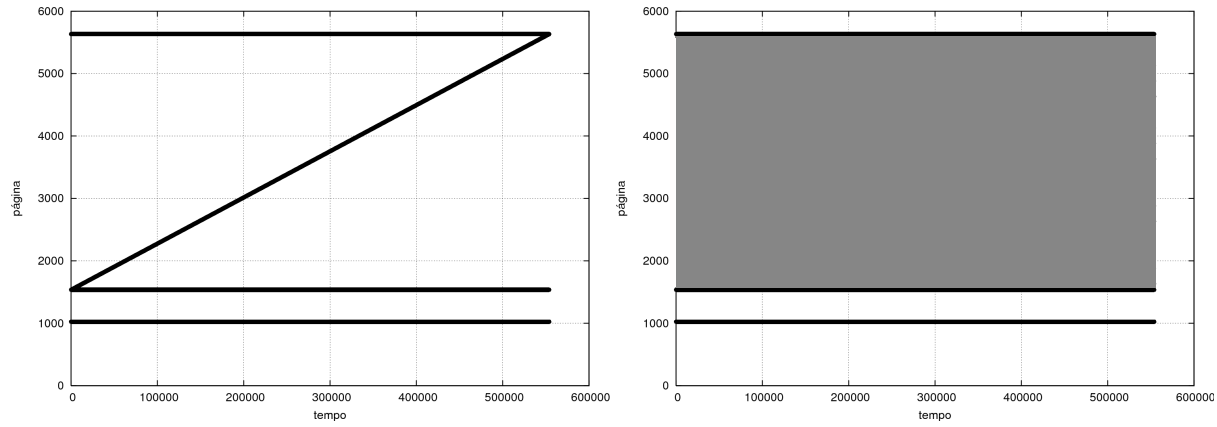


Figura 15.15: Localidade de referências nas duas execuções.

[4]Como a execução total de cada código gera mais de 500 milhões de referências à memória, foi feita uma amostragem da execução para construir os gráficos.

A localidade de referência de uma implementação depende de um conjunto de fatores, que incluem:

- As estruturas de dados usadas pelo programa: estruturas como vetores e matrizes têm seus elementos alocados de forma contígua na memória, o que leva a uma localidade de referências maior que estruturas mais dispersas, como listas encadeadas e árvores;
- Os algoritmos usados pelo programa: o comportamento do programa no acesso à memória é definido pelos algoritmos que ele implementa;
- A qualidade do compilador: cabe ao compilador analisar quais variáveis e trechos de código são usadas com frequência juntos e colocá-los nas mesmas páginas de memória, para aumentar a localidade de referências do código gerado.

A localidade de referências é uma propriedade importante para a construção de programas eficientes. Ela também é útil em outras áreas da computação, como a gerência das páginas armazenadas nos caches de navegadores *web* e servidores *proxy*, nos mecanismos de otimização de leituras/escritas em sistemas de arquivos, na construção da lista "arquivos recentes" dos menus de aplicações interativas, etc.

Referências

P. J. Denning. The locality principle. In J. Barria, editor, *Communication Networks and Computer Systems*, chapter 4, pages 43–67. Imperial College Press, 2006.

M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

R. Love. *Linux Kernel Development, Third Edition*. Addison-Wesley, 2010.

J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. In *5th USENIX Symposium on Operating Systems Design and Implementation*, pages 89–104, December 2002.

D. Patterson and J. Henessy. *Organização e Projeto de Computadores*. Campus, 2005.

M. Russinovich, D. Solomon, and A. Ionescu. *Microsoft Windows Internals, Fifth Edition*. Microsoft Press, 2008.

A. Tanenbaum. *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall, 2003.

Capítulo 16

Alocação de memória

As aplicações, utilitários e o próprio sistema operacional precisam de memória para executar. Como ocorre com os demais recursos de hardware, a memória disponível no sistema deve ser gerenciada pelo SO, para evitar conflitos entre aplicações e garantir justiça no seu uso. Fazendo uso dos mecanismos de hardware de memória apresentados no capítulo 15, o sistema operacional aloca e libera áreas de memória para os processos (ou para o próprio núcleo), conforme a necessidade. Este capítulo apresenta os principais conceitos relacionados à alocação de memória.

16.1 Alocadores de memória

Alocar memória significa reservar áreas de memória RAM que podem ser usadas por um processo, por um descritor de *socket* ou de arquivo no núcleo, por um cache de blocos de disco, etc. Ao final de seu uso, cada área de memória alocada é liberada pela entidade que a solicitou e colocada à disposição do sistema para novas alocações.

O mecanismo responsável pela alocação e liberação de áreas de memória é chamado um *alocador de memória*. Em linhas gerais, o alocador reserva ou libera partes da memória RAM, de acordo com o fluxo de solicitações que recebe (de processos ou do núcleo do sistema operacional). Para tal, o alocador deve manter um registro contínuo de quais áreas estão sendo usadas e quais estão livres. Para ser eficiente, ele deve realizar a alocações rapidamente e minimizar o desperdício de memória [Wilson et al., 1995].

Alocadores de memória podem existir em diversos contextos:

Alocador de memória física : organiza a memória física do computador, alocando e liberando grandes áreas de memória para carregar processos ou para atender requisições do núcleo.

Alocador de espaço de núcleo: o núcleo do SO continuamente cria e destrói muitas estruturas de dados relativamente pequenas, como descritores de arquivos abertos, de processos, *sockets* de rede, *pipes*, etc. O alocador de núcleo obtém áreas de memória do alocador físico e as utiliza para alocar essas estruturas para o núcleo.

Alocador de espaço de usuário: um processo pode solicitar blocos de memória para armazenar estruturas de dados dinâmicas, através de operações como `malloc` e `free`. O alocador de memória do processo geralmente é implementado por bibliotecas providas pelo sistema operacional, como a `LibC`. Essas bibliotecas

interagem com o núcleo para solicitar o redimensionamento da seção `HEAP` do processo quando necessário (Seção 14.2).

A Figura 16.1 apresenta uma visão geral dos mecanismos de alocação de memória em um sistema operacional típico. Na figura pode-se observar os três alocadores: de memória física, de núcleo e do espaço de usuário. O esquema apresentado nessa figura é genérico, pois as implementações variam muito entre sistemas operacionais distintos.

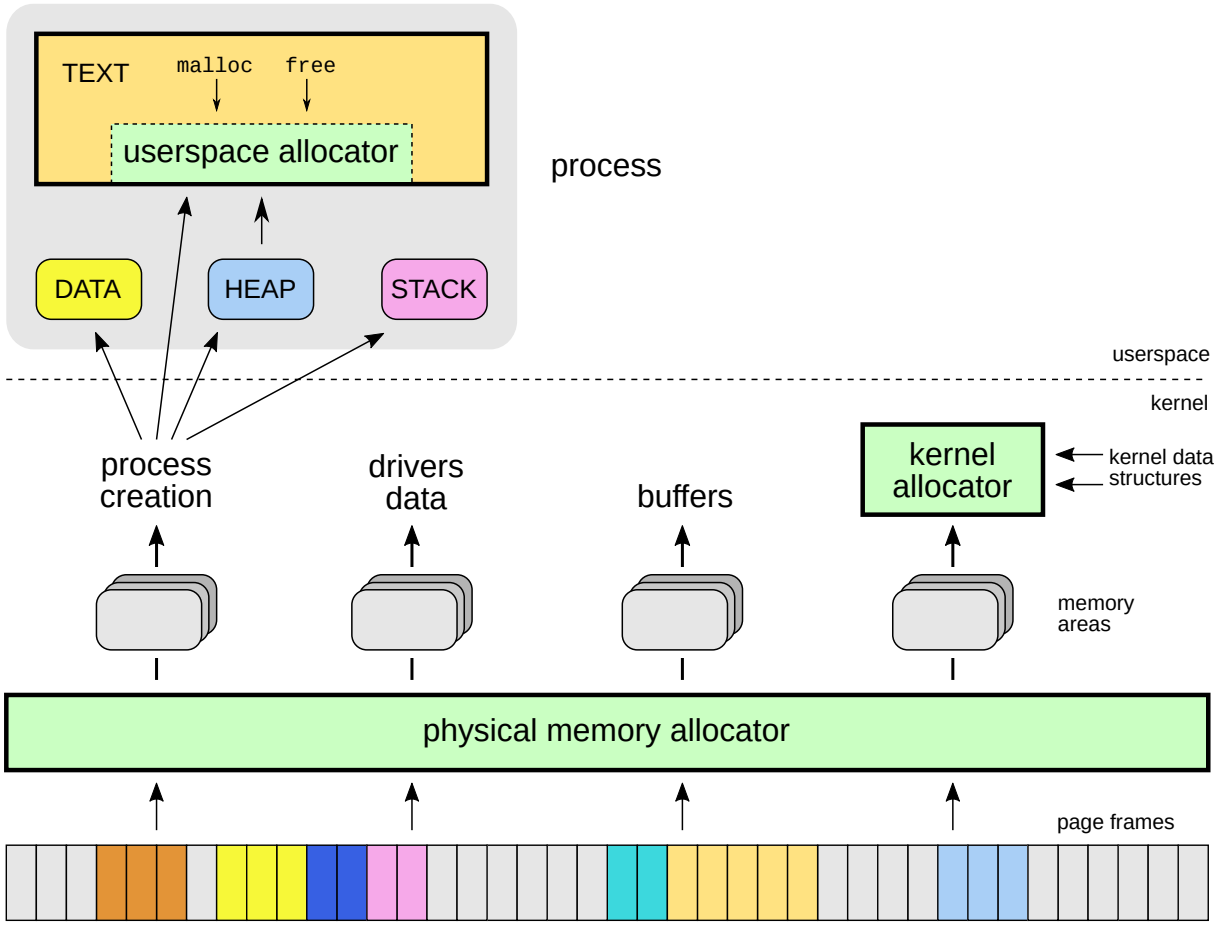


Figura 16.1: Mecanismos de alocação de memória.

16.2 Alocação básica

O problema básico de alocação consiste em manter uma grande área de memória RAM e atender um fluxo de requisições de alocação e liberação de partes dessa área para o sistema operacional ou as aplicações. Essas requisições ocorrem o tempo todo, em função das atividades em execução no sistema, e devem ser atendidas rapidamente.

Vejamos um exemplo simples: considere um sistema hipotético com 1 GB de memória RAM livre em uma área única[1]. O alocador de memória recebe a seguinte sequência de requisições: *aloca 200 MB* (a_1), *aloca 100 MB* (a_2), *aloca 100 MB* (a_3), *libera* a_1, *aloca 300 MB* (a_4) e *libera* a_3. O alocador atende essas requisições em sequência,

[1]Sistemas reais, como os computadores PC, podem ter várias áreas de memória livre distintas e não-contíguas.

reservando e liberando áreas de memória conforme necessário. A Figura 16.2 apresenta uma evolução possível das áreas de memória com as ações do alocador.

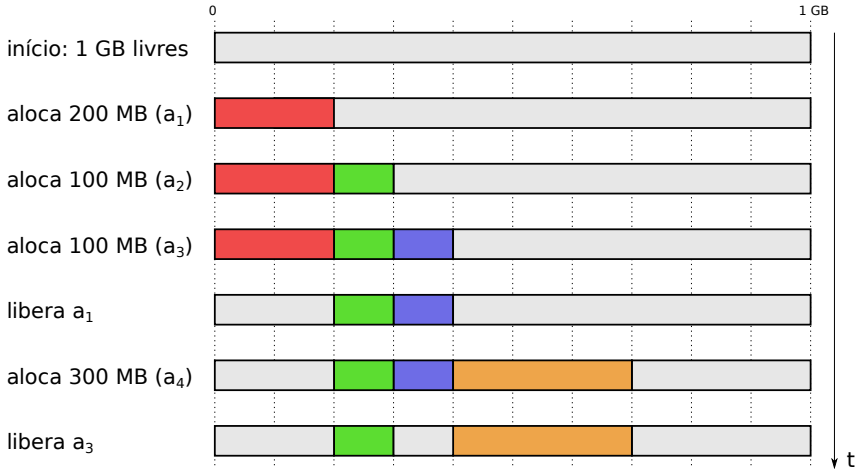


Figura 16.2: Sequência de alocações e liberações de memória.

Na Figura 16.2 pode-se observar que, como efeito das alocações e liberações, a área de memória inicialmente vazia se transforma em uma sequência de áreas ocupadas (alocadas) e áreas livres, que evolui a cada nova requisição. Essas informações são geralmente mantidas em uma ou mais listas duplamente encadeadas (ou árvores) de áreas de memória.

16.3 Fragmentação

Ao longo da vida de um sistema, áreas de memória são alocadas e liberadas continuamente. Com isso, podem surgir áreas livres ("buracos" na memória) entre as áreas alocadas. Por exemplo, na Figura 16.2 pode-se observar que o sistema ainda tem 600 MB de memória livre após a sequência de operações, mas somente requisições de alocação de até 300 MB pode ser aceitas, pois esse é o tamanho da maior área livre contínua disponível. Esse fenômeno se chama *fragmentação externa*, pois fragmenta a memória livre, fora das áreas alocadas.

A fragmentação externa é muito prejudicial, porque limita a capacidade de alocação de memória do sistema. Além disso, quanto mais fragmentada estiver a memória livre, maior o esforço necessário para gerenciá-la, pois mais longas serão as listas encadeadas de área de memória livres. Pode-se enfrentar o problema da fragmentação externa de duas formas: *minimizando* sua ocorrência, através de estratégias de alocação, *desfragmentando* periodicamente a memória do sistema, ou permitindo a fragmentação interna.

16.3.1 Estratégias de alocação

Para minimizar a ocorrência de fragmentação externa, cada pedido de alocação pode ser analisado para encontrar a área de memória livre que melhor o atenda. Essa análise pode ser feita usando um dos seguintes critérios:

***First-fit* (primeiro encaixe):** consiste em escolher a primeira área livre que satisfaça o pedido de alocação; tem como vantagem a rapidez, sobretudo se a lista de áreas livres for muito longa. É a estratégia adotada na Figura 16.2.

***Best-fit* (melhor encaixe):** consiste em escolher a menor área possível que possa receber a alocação, minimizando o desperdício de memória. Contudo, algumas áreas livres podem ficar pequenas demais e portanto inúteis.

***Worst-fit* (pior encaixe):** consiste em escolher sempre a maior área livre possível, de forma que a "sobra" seja grande o suficiente para ser usada em outras alocações.

***Next-fit* (próximo encaixe):** variante da estratégia *first-fit* que consiste em percorrer a lista de áreas a partir da última área alocada ou liberada, para que o uso das áreas livres seja distribuído de forma mais homogênea no espaço de memória.

Diversas pesquisas [Johnstone and Wilson, 1999] demonstraram que as abordagens mais eficientes são a de melhor encaixe e a *first-fit*, sendo esta última bem mais rápida. A Figura 16.3 ilustra essas estratégias na alocação de um bloco de 80 MB dentro da área de 1 GB.

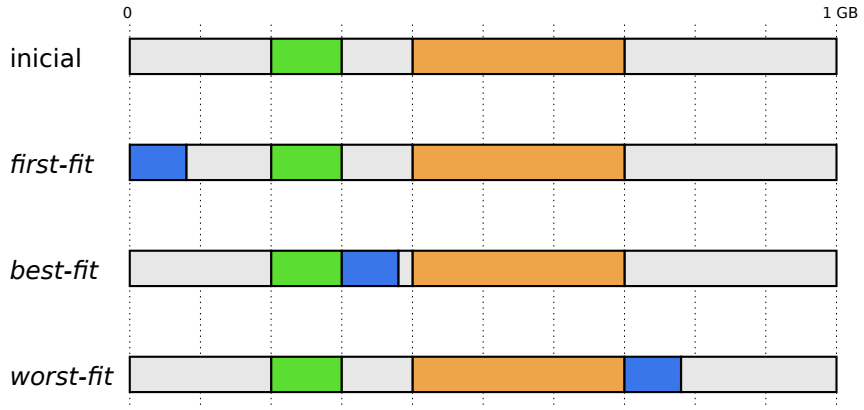


Figura 16.3: Estratégias para minimizar a fragmentação externa.

16.3.2 Desfragmentação

Outra forma de tratar a fragmentação externa consiste em *desfragmentar* a memória periodicamente. Para tal, as áreas de memória usadas pelos processos devem ser movidas na memória de forma a concatenar as áreas livres e assim diminuir a fragmentação. Ao mover um processo na memória, suas informações de endereçamento virtual (registrador base/limite, tabela de segmentos ou de páginas) devem ser devidamente ajustadas para refletir a nova posição do processo na memória RAM. Por essa razão, a desfragmentação só pode ser aplicada a áreas de memória físicas, pois as mudanças de endereço das áreas de memória serão ocultadas pelo hardware. Ela não pode ser aplicada, por exemplo, para a gestão da seção *heap* de um processo.

Como as áreas de memória não podem ser acessadas durante a desfragmentação, é importante que esse procedimento seja executado rapidamente e com pouca frequência, para não interferir nas atividades normais do sistema. As possibilidades de movimentação de áreas podem ser muitas, portanto a desfragmentação deve ser

tratada como um problema de otimização combinatória. A Figura 16.4 ilustra três possibilidades de desfragmentação de uma determinada situação de memória; as três alternativas produzem o mesmo resultado (uma área livre contínua com 450 MB), mas têm custos distintos.

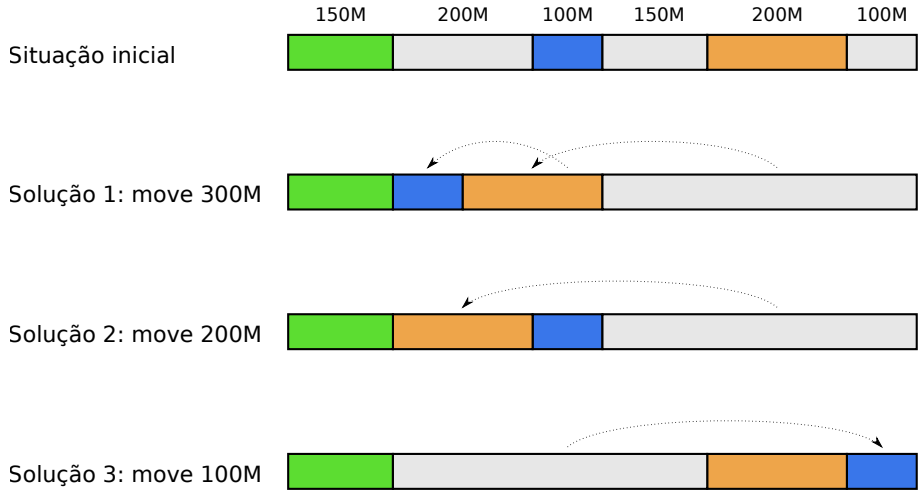


Figura 16.4: Possibilidades de desfragmentação.

16.3.3 Fragmentação interna

Uma alternativa para minimizar o impacto da fragmentação externa consiste em arredondar algumas requisições de alocação, para evitar sobras muito pequenas. Por exemplo, na alocação com *best-fit* da Figura 16.3, a área alocada poderia ser arredondada de 80 MB para 100 MB, evitando a sobra da área de 20 MB (Figura 16.5). Dessa forma, evita-se a geração de um fragmento de memória livre, mas a memória adicional alocada provavelmente não será usada por quem a requisitou. Esse desperdício de memória dentro da área alocada é denominado *fragmentação interna* (ao contrário da fragmentação externa, que ocorre nas áreas livres).

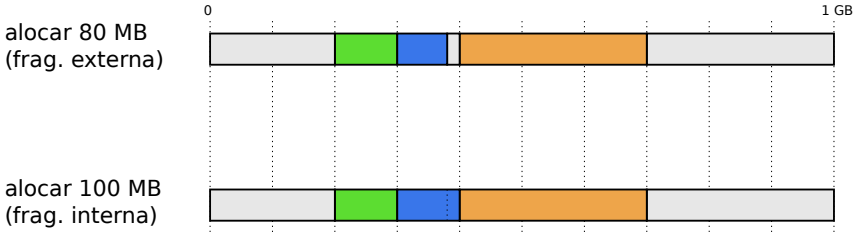


Figura 16.5: Fragmentação interna.

A fragmentação interna afeta todas as formas de organização de memória; as partições e segmentos sofrem menos com esse problema, pois o nível de arredondamento das áreas de memória pode ser decidido caso a caso. No caso da memória paginada, essa decisão não é possível, pois as alocações são sempre feitas em múltiplos inteiros de páginas. Assim, em um sistema com páginas de 4 KBytes (4.096 bytes), um processo que solicite 550.000 bytes (134,284 páginas) receberá 552.960 bytes (135 páginas), ou seja, 2.960 bytes a mais que o solicitado.

Em média, para cada processo haverá uma perda de 1/2 página de memória por fragmentação interna. Uma forma de minimizar a perda por fragmentação interna seria usar páginas de menor tamanho (2K, 1K, 512 bytes ou ainda menos). Todavia, essa abordagem implica em ter mais páginas por processo, o que geraria tabelas de páginas maiores e com maior custo de gerência.

16.4 O alocador Buddy

Existem estratégias de alocação mais sofisticadas e eficientes que as apresentadas na seção 16.3.1. Um algoritmo de alocação muito conhecido é o chamado *Buddy Allocator*, ou alocador por pares [Wilson et al., 1995], explicado a seguir. Em sua versão mais simples, a estratégia *Buddy* sempre aloca blocos de memória de tamanho 2^n, com n inteiro e ajustável. Por exemplo, para uma requisição de 85 KBytes será alocado um bloco de memória com 128 KBytes (2^7 KBytes ou 2^{17} bytes), e assim por diante. O uso de blocos de tamanho 2^n reduz a fragmentação externa, mas pode gerar muita fragmentação interna.

O valor de n pode variar entre os limites n_{min} e n_{max}, ou seja, $n_{min} < n < n_{max}$. n_{min} define o menor bloco que pode ser alocado, para evitar custo computacional e desperdício de espaço com a alocação de blocos muito pequenos. Valores entre 1 KByte e 64 KBytes são usuais. Por sua vez, n_{max} define o tamanho do maior bloco alocável, sendo limitado pela quantidade de memória RAM disponível. Em um sistema com 2.000 MBytes de memória RAM livre, o maior bloco alocável teria 1.024 MBytes (ou 2^{20} Bytes). Os 976 MBytes restantes também podem ser alocados, mas em blocos menores.

O funcionamento do alocador Buddy binário (blocos de 2^n bytes) é simples:

- Ao receber uma requisição de alocação de memória de tamanho 40 KBytes (por exemplo), o alocador procura um bloco livre com 64 KBytes (pois 64 KBytes é o menor bloco com tamanho 2^n que pode conter 40 KBytes). Caso não encontre um bloco com 64 KBytes, procura um bloco livre com 128 KBytes, o divide em dois blocos de 64 KBytes (os *buddies*) e usa um deles para a alocação. Caso não encontre um bloco livre com 128 KBytes, procura um bloco com 256 KBytes para dividir em dois, e assim sucessivamente.

- Ao liberar uma área de memória alocada, o alocador verifica se o par (*buddy*) do bloco liberado também está livre; se estiver, funde os dois em um bloco maior, analisa o novo bloco em relação ao seu par e continua as fusões de blocos, até encontrar um par ocupado ou chegar ao tamanho máximo de bloco permitido. A fusão entre dois blocos vizinhos também é chamada de *coalescência*.

A Figura 16.6 apresenta um exemplo de funcionamento do alocador *Buddy*. Nesse exemplo didático, o tamanho da memória é de 1 MBytes (1.024 KBytes) e o menor bloco alocável é de 64 KBytes. O alocador de memória recebe a seguinte sequência de requisições: *aloca 200 KB* (a_1), *aloca 100 KB* (a_2), *aloca 150 KB* (a_3), *libera* a_1 e *libera* a_2.

Além da estratégia binária apresentada aqui, existem variantes de alocador *Buddy* que usam outras formas de dividir blocos, como a estratégia Buddy com Fibonacci, na qual os tamanhos dos blocos seguem a sequência de Fibonacci[2] e a estratégia Buddy

[2]Na sequência de Fibonacci, cada termo corresponde à soma dos dois termos anteriores: 0, 1, 1, 2, 3, 5, 8, 13, 21, etc. Em termos matemáticos, $F_0 = 0, F_1 = 1, F_{n>1} = F_{n-1} + F_{n-2}$.

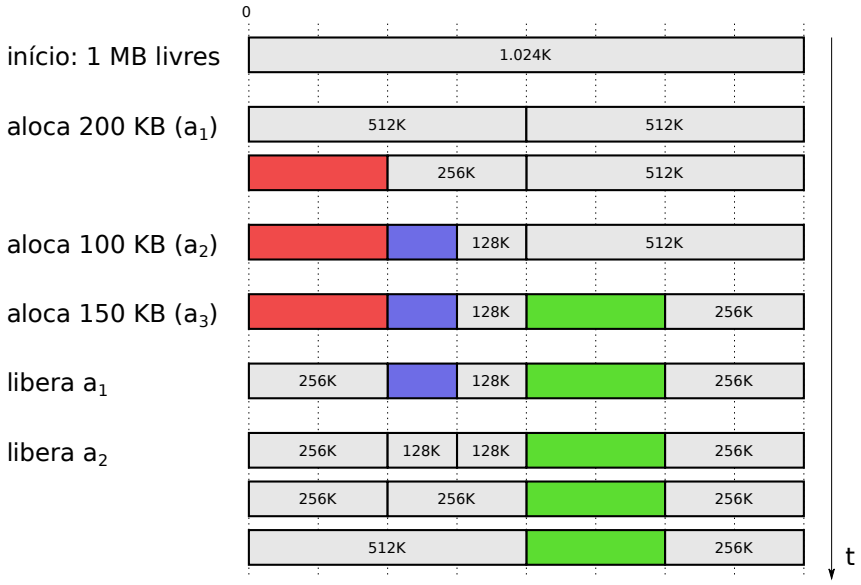


Figura 16.6: O alocador Buddy binário.

com pesos, na qual os blocos têm tamanhos 2^n mas são divididos em sub-blocos de tamanho distintos (por exemplo, um bloco de 64 KB seria dividido em dois blocos, de 48 KB e 16KB).

O alocador Buddy é usado em vários sistemas. Por exemplo, no núcleo Linux ele é usado para a alocação de memória física (*page frames*), entregando áreas de memória RAM para a criação de processos, para o alocador de objetos do núcleo e para outros subsistemas. O arquivo `/proc/buddyinfo` permite consultar informações das alocações existentes.

16.5 O alocador Slab

O alocador Slab foi inicialmente proposto para o núcleo do sistema operacional SunOS 5.4 [Bonwick, 1994]. Ele é especializado na alocação de "objetos de núcleo", ou seja, as pequenas estruturas de dados que são usadas para representar descritores de processos, de arquivos abertos, *sockets* de rede, *pipes*, etc. Esses objetos de núcleo são continuamente criados e destruídos durante a operação do sistema, são pequenos (dezenas ou centenas de bytes) e têm tamanhos relativamente padronizados.

Alocar e liberar memória para objetos de núcleo usando um alocador básico ou Buddy implicaria em um custo computacional elevado, além de desperdício de memória em fragmentação. Por isso é necessário um alocador especializado, capaz de fornecer memória para esses objetos rapidamente e com baixo custo. Outra questão importante é a inicialização dos objetos de núcleo: pode-se economizar custos de inicialização se os objetos liberados forem mantidos na memória e reutilizados, ao invés daquela área de memória ser liberada.

O alocador Slab usa uma estratégia baseada no *caching* de objetos. É definido um *cache* para cada tipo de objeto usado pelo núcleo: descritor de processo, de arquivo, de *socket*, etc[3]. Cada cache é então dividido em *slabs* (lajes ou placas) que contêm objetos daquele tipo, portanto todos com o mesmo tamanho. Um slab pode estar **cheio**, quando

[3]No kernel Linux 4.1 há mais de 140 caches de objetos; eles estão listados no arquivo `/proc/slabinfo`.

todos os seus objetos estão em uso, **vazio**, quando todos os seus objetos estão livres, ou **parcial**. A Figura 16.7 ilustra a estrutura dos caches de objetos.

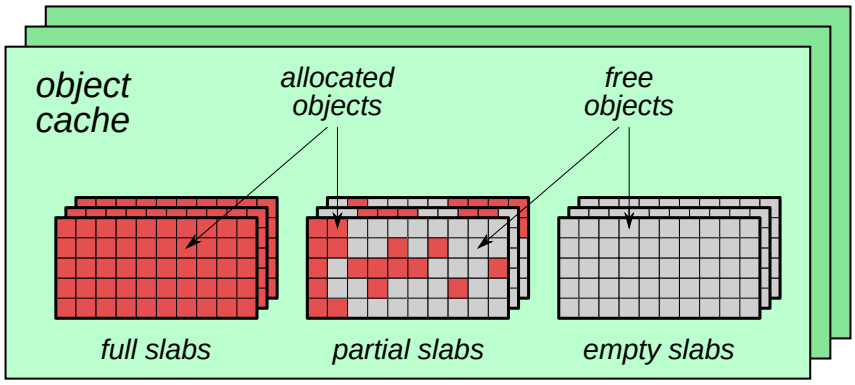


Figura 16.7: Estrutura de caches, slabs e objetos do alocador Slab.

A estratégia de alocação é a seguinte: quando um novo objeto de núcleo é requisitado, o alocador analisa o cache daquele tipo de objeto e entrega um objeto livre de um slab parcial; caso não hajam slabs parciais, entrega um objeto livre de um slab vazio (o que altera o status desse slab para parcial). Caso não existam slabs vazios, o alocador pede mais páginas de RAM ao alocador de memória física para criar um novo slab, inicializar seus objetos e marcá-los como livres. Quando um objeto é liberado, ele é marcado como livre; caso todos os objetos de um slab fiquem livres, este é marcado como vazio. Caso o sistema precise liberar memória para outros usos, o alocador pode descartar os slabs vazios, liberando suas áreas junto ao alocador de memória física.

O alocador Slab é usado para a gestão de objetos de núcleo em muitos sistemas operacionais, como Linux, Solaris, FreeBSD e Horizon (usado no console Nintendo Switch).

16.6 Alocação no espaço de usuário

Da mesma forma que o núcleo, aplicações no espaço de usuário podem ter necessidade de alocar memória durante a execução para armazenar estruturas de dados dinâmicas (conforme discutido na Seção 14.3). Ao ser criado, cada processo recebe uma área para alocação dinâmica de variáveis, chamada `HEAP`. O tamanho dessa seção pode ser ajustado através de chamadas de sistema que modifiquem o ponteiro *Program Break* (vide Seção 14.2).

A gerência da seção `HEAP` pode ser bastante complexa, caso a aplicação use variáveis dinâmicas para construir listas, pilhas, árvores ou outras estruturas de dados mais sofisticadas. Por isso, ela usualmente fica a cargo de bibliotecas de sistema, como a biblioteca C padrão (*LibC*), que oferecem funções básicas de alocação de memória como `malloc` e `free`. A biblioteca então fica encarregada de alocar/liberar blocos de memória na seção `HEAP`, gerenciar quais blocos estão livres ou ocupados, e solicitar ao núcleo do SO o aumento ou redução dessa seção, conforme necessário.

Existem várias implementações de alocadores de uso geral para o espaço de usuário. As implementações mais simples seguem o esquema apresentado na seção 16.2, com estratégia *best-fit*. Implementações mais sofisticadas, como a *DLmalloc* (*Doug*

Lea's Malloc), usada nos sistemas GNU/Linux, usam diversas técnicas para evitar a fragmentação e agilizar a alocação de blocos de memória.

Além dos alocadores de uso geral, podem ser desenvolvidos alocadores customizados para aplicações específicas. Uma técnica muito usada em sistemas de tempo real, por exemplo, é o *memory pool* (reserva de memória). Nessa técnica, um conjunto de blocos de mesmo tamanho é pré-alocado, constituindo um *pool*. A aplicação pode então obter e liberar blocos de memória desse *pool* com rapidez, pois o alocador só precisa registrar quais blocos estão livres ou ocupados.

Referências

J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer Conference*, volume 16. Boston, MA, USA, 1994.

M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: solved? *ACM SIGPLAN Notices*, 34(3):26–36, 1999.

P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Memory Management*, pages 1–116. Springer, 1995.

Capítulo 17

Paginação em disco

A memória RAM sempre foi um recurso escasso em sistemas de computação. Por isso, seu uso deve ser gerenciado de forma eficiente, para que todos os processos possam ter memória suficiente para operar. Além disso, a crescente manipulação de informações multimídia (imagens, áudio, vídeo) contribui para esse problema, uma vez que essas informações são geralmente volumosas e seu tratamento exige grandes quantidades de memória livre.

Como a memória RAM é um recurso caro (cerca de U$10/GByte no mercado americano, em 2018) e que consome uma quantidade significativa de energia, aumentar a quantidade de memória nem sempre é uma opção factível. No entanto, o computador geralmente possui discos rígidos ou SSD maiores, mais baratos e mais lentos que a memória RAM. Em valores de 2018, 1 GByte de disco rígido custa cerca de U$0,05, enquanto 1 GByte de SSD custa cerca de U$0,30 (valores apenas indicativos, variando de acordo com o fabricante e a tecnologia envolvida).

Este capítulo apresenta técnicas usadas para usar um dispositivo de armazenamento secundário como extensão da memória RAM, de forma transparente para as aplicações.

17.1 Estendendo a memória RAM

Os mecanismos de de memória virtual suportados pelo hardware, apresentados no Capítulo 15, permitem usar dispositivos de armazenamento secundário como extensão da memória RAM. Com isso, partes ociosas da memória podem ser transferidas para um disco, liberando a memória RAM para outros usos. Caso algum processo tente acessar esse conteúdo posteriormente, ele deverá ser trazido de volta à memória. A transferência dos dados entre memória e disco é feita pelo sistema operacional, de forma transparente para os processos.

Existem diversas técnicas para usar um espaço de armazenamento secundário como extensão da memória RAM, com ou sem o auxílio do hardware. As mais conhecidas são:

Overlays: o programador organiza seu programa em módulos que serão carregados em uma mesma região de memória em momentos distintos. Esses módulos, chamados de *overlays*, são gerenciados através de uma biblioteca específica. Por exemplo, o código de um compilador pode separar o analisador léxico, o analisador sintático e o gerador de código em *overlays*, que serão ativados

em momentos distintos. Esta abordagem, popular em alguns ambientes de desenvolvimento nos anos 1970-90, como o Turbo Pascal da empresa Borland, é raramente usada hoje em dia, pois exige maior esforço por parte do programador.

Swapping: consiste em mover um processo ocioso da memória RAM para um disco (*swap-out*), liberando a memória para outros processos. Mais tarde, quando esse processo for acordado (entrar na fila de prontos do escalonador), ele é carregado de volta na memória (*swap-in*). A técnica de *swapping* foi muito usada até os anos 1990, mas hoje é pouco empregada em sistemas operacionais de uso geral.

Paging: consiste em mover páginas individuais, conjuntos de páginas ou mesmo segmentos da memória para o disco (*page-out*). Se o processo tentar acessar uma dessas páginas mais tarde, a MMU gera uma interrupção de falta de página e o núcleo do SO recarrega a página faltante na memória (*page-in*). Esta é a técnica mais usada nos sistemas operacionais atuais, por sua flexibilidade, rapidez e eficiência.

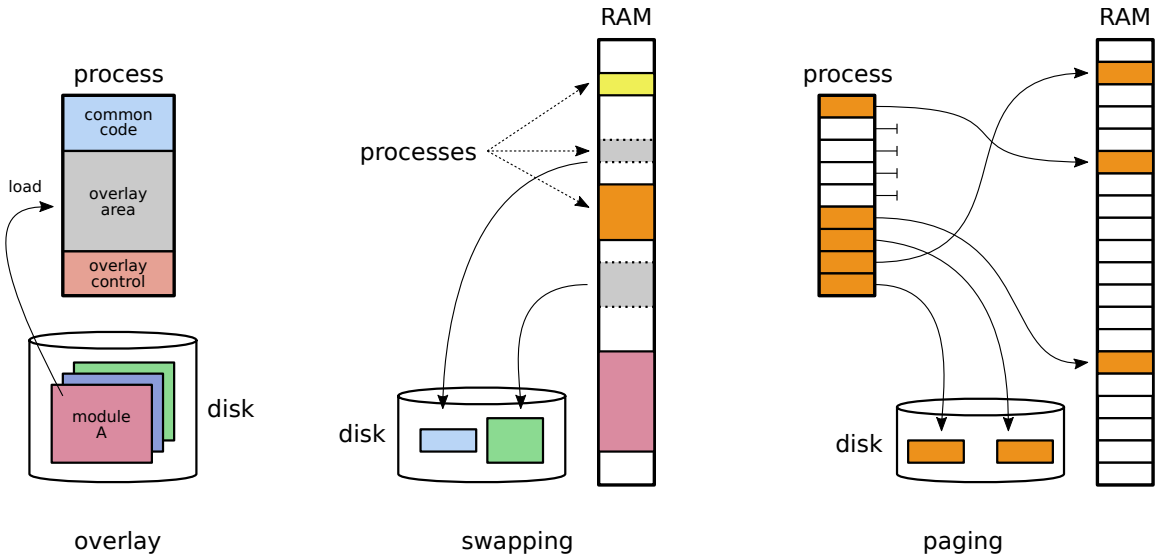


Figura 17.1: Abordagens de extensão da memória em disco.

17.2 A paginação em disco

A ideia central da paginação em disco consiste em transferir páginas ociosas da memória RAM para uma área em disco, liberando memória para outras páginas. Esta seção explica o funcionamento básico desse mecanismo e discute sobre sua eficiência.

17.2.1 Mecanismo básico

A transferência de páginas entre a memória e o disco é realizada pelo núcleo do sistema operacional. As páginas a retirar da memória são escolhidas por ele, de acordo com algoritmos de substituição de páginas, discutidos na Seção 17.3. Quando um processo tentar acessar uma página que está em disco, o núcleo é alertado pela MMU e traz a página de volta à memória para poder ser acessada.

Para cada página transferida para o disco, a tabela de páginas do processo é ajustada: o *flag* de presença da página em RAM é desligado e a posição da página no disco é registrada, ao invés do quadro. Essa situação está ilustrada de forma simplificada na Figura 17.2.

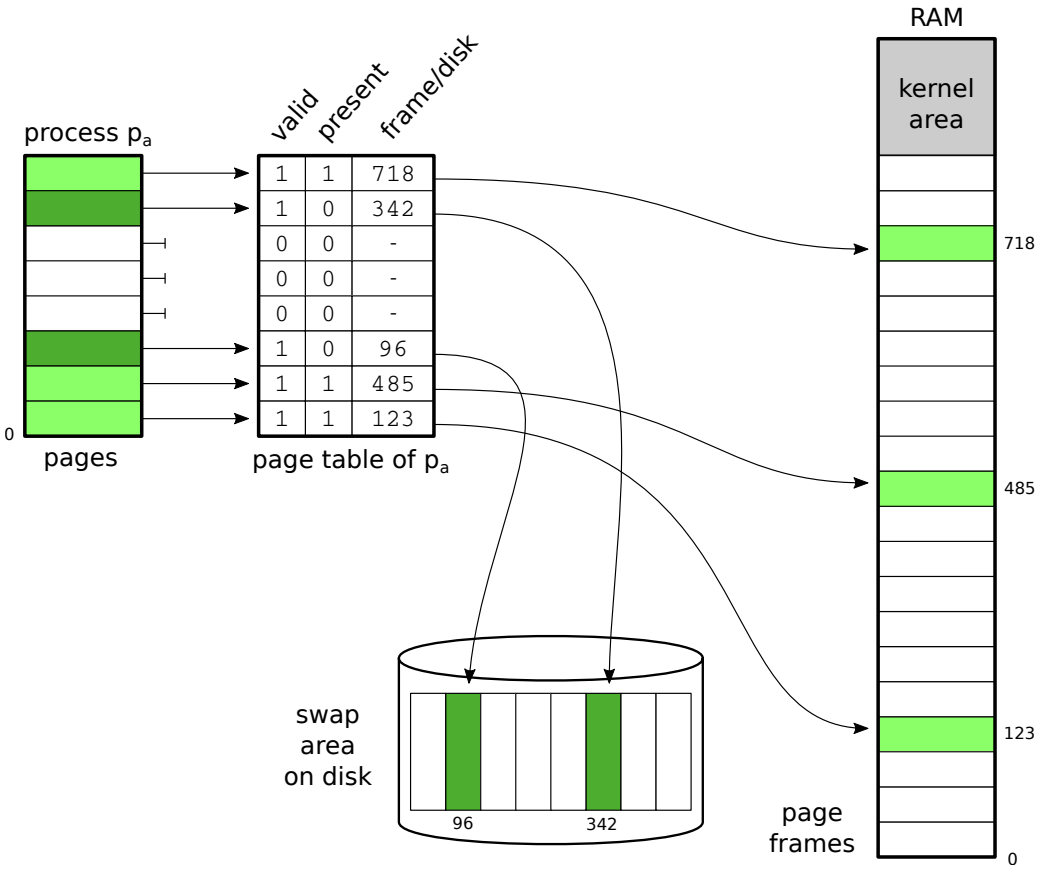


Figura 17.2: A paginação em disco.

O armazenamento externo das páginas pode ser feito em um disco exclusivo (usual em servidores de maior porte), em uma partição do disco principal (usual no Linux e outros UNIX) ou em um arquivo reservado dentro do sistema de arquivos (como no Windows NT e sucessores). Em alguns sistemas, é possível usar uma área de troca remota, em um servidor de rede; todavia, essa solução apresenta baixo desempenho. Por razões históricas, essa área de disco é geralmente denominada *área de troca* (*swap area*), embora armazene páginas. No caso de um disco exclusivo ou uma partição de disco, essa área geralmente é formatada usando uma estrutura de sistema de arquivos otimizada para a transferência rápida das páginas.

Páginas que foram transferidas da memória para o disco possivelmente serão acessadas no futuro por seus processos. Quando um processo tentar acessar uma página que está em disco, esta deve ser transferida de volta para a memória para possibilitar o acesso, de forma transparente ao processo. Conforme exposto na Seção 15.6, quando um processo acessa uma página, a MMU verifica se a mesma está presente na memória RAM; em caso positivo, faz o acesso ao endereço físico correspondente. Caso contrário, a MMU gera uma interrupção de falta de página (*page fault*) que desvia a execução para o sistema operacional.

O sistema operacional verifica se a página é válida, usando os *flags* de controle da tabela de páginas. Caso a página seja inválida, o processo tentou acessar um endereço

inválido e deve ser abortado. Por outro lado, caso a página solicitada seja válida, o processo deve ser suspenso enquanto o sistema transfere a página de volta para a memória RAM e faz os ajustes necessários na tabela de páginas. Uma vez a página carregada em memória, o processo pode continuar sua execução. O fluxograma da Figura 17.3 apresenta as principais ações desenvolvidas pelo mecanismo de paginação em disco.

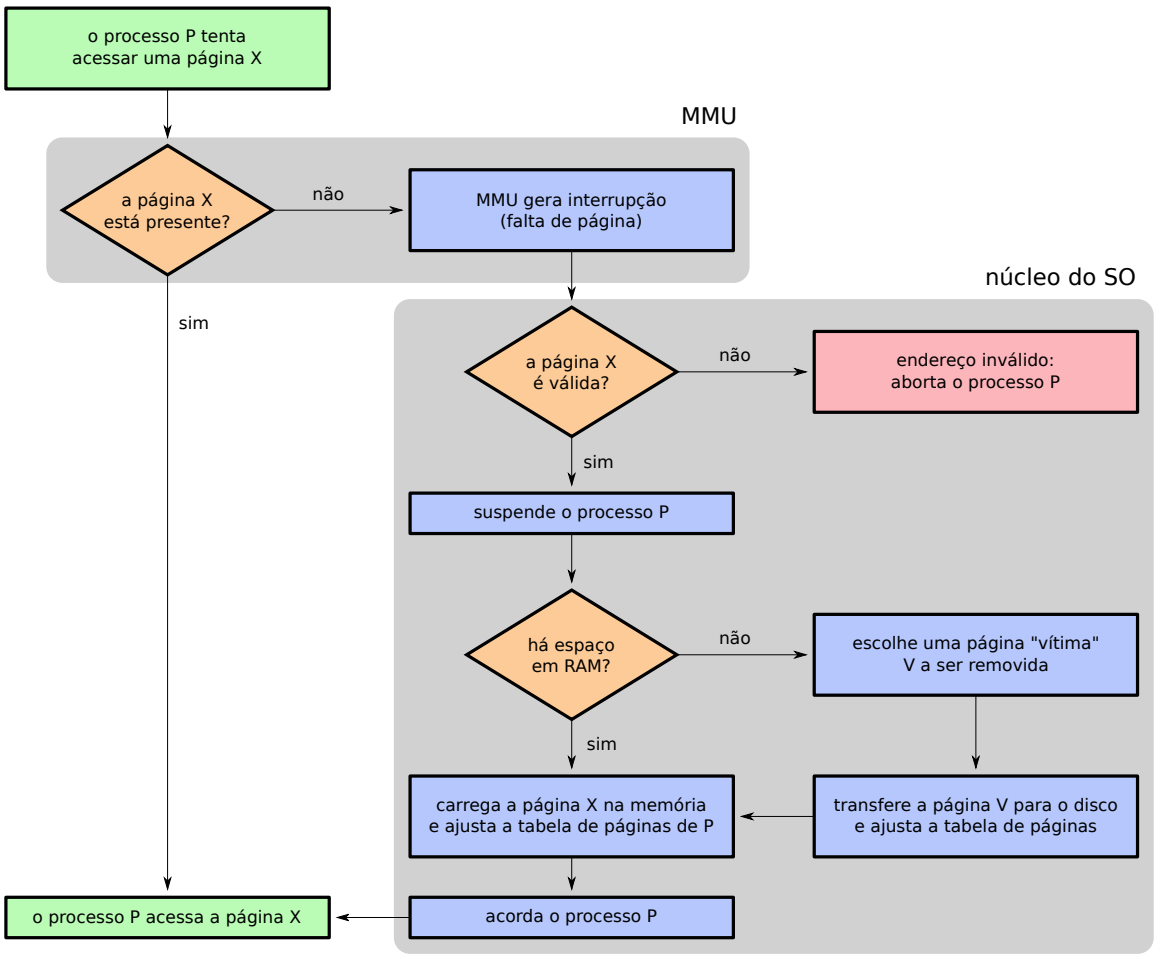


Figura 17.3: Ações do mecanismo de paginação em disco.

Caso a memória principal já esteja cheia, uma página deverá ser movida para o disco antes de trazer de volta a página faltante. Isso implica em mais operações de leitura e escrita no disco e portanto em mais demora para atender o pedido do processo. Muitos sistemas, como o Linux e o Solaris, evitam essa situação mantendo um *daemon*[1] responsável por escolher e transferir páginas para o disco, sempre que a quantidade de memória livre estiver abaixo de um certo limiar.

Retomar a execução do processo que gerou a falta de página pode ser uma tarefa complexa. Como a instrução que gerou a falta de página não foi completada, ela deve ser reexecutada. No caso de instruções simples, envolvendo apenas um endereço de memória sua reexecução é trivial. Todavia, no caso de instruções que envolvam várias ações e vários endereços de memória, deve-se descobrir qual dos endereços gerou a falta de página, que ações da instrução foram executadas e então executar somente o

[1] *Daemons* são processos que executam continuamente, sem interação com o usuário, para prover serviços ao sistema operacional, como gestão de impressoras, de conexões de rede, etc.

que estiver faltando. A maioria dos processadores atuais provê registradores especiais que auxiliam nessa tarefa.

17.2.2 Eficiência

O mecanismo de paginação em disco permite usar o disco como uma extensão de memória RAM, de forma transparente para os processos. Seria a solução ideal para as limitações da memória principal, se não houvesse um problema importante: o tempo de acesso dos discos utilizados. Conforme os valores indicados na Tabela 14.1, um disco rígido típico tem um tempo de acesso cerca de 100.000 vezes maior que a memória RAM. Cada falta de página provocada por um processo implica em um acesso ao disco, para buscar a página faltante (ou dois acessos, caso a memória RAM esteja cheia e outra página tenha de ser removida antes). Assim, faltas de página muito frequentes irão gerar muitos acessos ao disco, aumentando o tempo médio de acesso à memória e, em consequência, diminuindo o desempenho geral do sistema.

Para demonstrar o impacto das faltas de página no desempenho, consideremos um sistema cuja memória RAM tem um tempo de acesso de 60 ns ($60 \times 10^{-9}s$) e cujo disco de troca tem um tempo de acesso de 6 ms ($6 \times 10^{-3}s$), no qual ocorre uma falta de página a cada milhão de acessos (10^6 acessos). Caso a memória não esteja saturada, o tempo médio de acesso será:

$$\begin{aligned} t_{médio} &= \frac{(999.999 \times 60ns) + 6ms + 60ns}{1.000.000} \\ &= \frac{10^6 \times 60 \times 10^{-9} + 6 \times 10^{-3}}{10^6} \\ t_{médio} &= 66ns \end{aligned}$$

Caso a memória esteja saturada, o tempo médio será maior:

$$\begin{aligned} t_{médio} &= \frac{(999.999 \times 60ns) + 2 \times 6ms + 60ns}{1.000.000} \\ &= \frac{10^6 \times 60 \times 10^{-9} + 2 \times 6 \times 10^{-3}}{10^6} \\ t_{médio} &= 72ns \end{aligned}$$

Caso a frequência de falta de páginas aumente para uma falta a cada 100.000 acessos (10^5 acessos), o tempo médio de acesso à memória subirá para 120 ns no primeiro caso (memória não saturada) e 180 ns no segundo caso (memória saturada).

A frequência de faltas de página depende de vários fatores, como:

- O tamanho da memória RAM, em relação à demanda dos processos em execução: sistemas com memória insuficiente, ou muito carregados, podem gerar muitas faltas de página, prejudicando o seu desempenho e podendo ocasionar o fenômeno conhecido como *thrashing* (Seção 17.7).

- o comportamento dos processos em relação ao uso da memória: processos que agrupem seus acessos a poucas páginas em cada momento, respeitando a localidade de referências (Seção 15.9), necessitam usar menos páginas simultaneamente e geram menos faltas de página.
- A escolha das páginas a remover da memória: caso sejam removidas páginas usadas com muita frequência, estas serão provavelmente acessadas pouco tempo após sua remoção, gerando mais faltas de página. A escolha das páginas a remover é responsabilidade dos algoritmos apresentados na Seção 17.3.

17.2.3 Critérios de seleção

Vários critérios podem ser usados para escolher páginas "vítimas", ou seja, páginas a transferir da memória RAM para o armazenamento secundário:

Idade da página: há quanto tempo a página está na memória; páginas muito antigas talvez sejam pouco usadas.

Frequência de acessos à página: páginas muito acessadas em um passado recente possivelmente ainda o serão em um futuro próximo.

Data do último acesso: páginas há mais tempo sem acessar possivelmente serão pouco acessadas em um futuro próximo (sobretudo se os processos respeitarem o princípio da localidade de referências).

Prioridade do processo proprietário: processos de alta prioridade, ou de tempo real, podem precisar de suas páginas de memória rapidamente; se elas estiverem no disco, seu desempenho ou tempo de resposta poderão ser prejudicados.

Conteúdo da página: páginas cujo conteúdo seja código executável exigem menos esforço do mecanismo de paginação, porque seu conteúdo já está mapeado no disco (dentro do arquivo executável correspondente ao processo). Por outro lado, páginas de dados que tenham sido alteradas precisam ser salvas na área de troca.

Páginas especiais: páginas contendo *buffers* de operações de entrada/saída podem ocasionar dificuldades ao núcleo caso não estejam na memória no momento em que ocorrer a transferência de dados entre o processo e o dispositivo físico. O processo também pode solicitar que certas páginas contendo informações sensíveis (como senhas ou chaves criptográficas) não sejam copiadas na área de troca, por segurança.

A escolha correta das páginas a retirar da memória física é um fator essencial para a eficiência do mecanismo de paginação. Más escolhas poderão remover da memória páginas muito usadas, aumentando a taxa de faltas de página e diminuindo o desempenho do sistema.

17.3 Algoritmos clássicos

Existem vários algoritmos para a escolha de páginas a substituir na memória, visando reduzir a frequência de falta de páginas, que levam em conta alguns dos fatores acima enumerados. Os principais algoritmos serão apresentados na sequência.

17.3.1 Cadeia de referências

Uma ferramenta importante para o estudo dos algoritmos de substituição de páginas é a *cadeia de referências* (*reference string*), que indica a sequência de páginas acessadas por um processo ao longo de sua execução, considerando todos os endereços acessados pelo processo nas várias áreas de memória que o compõem (código, dados, pilha, *heap*, etc). Ao submeter a cadeia de referências de uma execução aos vários algoritmos, podemos calcular quantas faltas de página cada um geraria naquela execução em particular, permitindo assim comparar suas eficiências.

Cadeias de referências de execuções reais podem ser muito longas: considerando um tempo de acesso à memória de 50 ns, em apenas um segundo de execução ocorrem por volta de 20 milhões de acessos à memória. Além disso, a obtenção de cadeias de referências confiáveis é uma área de pesquisa importante, por envolver técnicas complexas de coleta, filtragem e compressão de dados de execução de sistemas [Uhlig and Mudge, 1997]. Para possibilitar a comparação dos algoritmos de substituição de páginas apresentados na sequência, será usada a seguinte cadeia de referências fictícia, obtida de [Silberschatz et al., 2001]:

$$7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1$$

Deve-se observar que acessos consecutivos a uma mesma página não são relevantes para a análise dos algoritmos, porque somente o primeiro acesso em cada grupo de acessos consecutivos provoca uma falta de página.

17.3.2 Algoritmo Ótimo

Idealmente, a melhor página a remover da memória em um dado instante é aquela que ficará mais tempo sem ser usada pelos processos. Esta ideia simples define o *algoritmo ótimo* (OPT). Entretanto, como o comportamento futuro dos processos não pode ser previsto com precisão, este algoritmo não é implementável. Mesmo assim ele é importante, porque define um limite mínimo conceitual: se, para uma dada cadeia de referências, o algoritmo ótimo gera X faltas de página, nenhum outro algoritmo irá gerar menos de X faltas de página ao tratar essa mesma cadeia. Assim, seu resultado serve como parâmetro para a avaliação dos demais algoritmos.

A aplicação do algoritmo ótimo à cadeia de referências apresentada na Seção anterior, considerando uma memória física com 3 quadros, é apresentada na Tabela 17.1. Nesse caso, o algoritmo OPT gera 9 faltas de página.

17.3.3 Algoritmo FIFO

Um critério simples e factível a considerar para a escolha das páginas a substituir poderia ser sua "idade", ou seja, o tempo em que estão na memória. Assim, páginas mais antigas podem ser removidas para dar lugar a novas páginas. Esse algoritmo é muito simples de implementar: basta organizar as páginas em uma fila de números de páginas com política FIFO (*First In, First Out*). Os números das páginas recém carregadas na memória são registrados no final da lista, enquanto os números das próximas páginas a substituir na memória são obtidos no início da lista.

A aplicação do algoritmo FIFO à cadeia de referências apresentada na Seção anterior, considerando uma memória física com 3 quadros, é apresentada na Tabela 17.2. Nesse caso, o algoritmo gera no total 15 faltas de página, 6 a mais que o algoritmo ótimo.

t	página acessada	quadros q_o	q_1	q_2	falta de página?	ação realizada
0						situação inicial, quadros vazios
1	7	7			✓	p_7 é carregada em q_0
2	0	7	0		✓	p_0 é carregada em q_1
3	1	7	0	1	✓	p_1 é carregada em q_2
4	2	2	0	1	✓	p_2 substitui p_7 (que só será acessada em $t = 18$)
5	0	2	0	1		p_0 já está na memória
6	3	2	0	3	✓	p_3 substitui p_1
7	0	2	0	3		p_0 já está na memória
8	4	2	4	3	✓	p_4 substitui p_0
9	2	2	4	3		p_2 já está na memória
10	3	2	4	3		p_3 já está na memória
11	0	2	0	3	✓	p_0 substitui p_4
12	3	2	0	3		p_3 já está na memória
13	2	2	0	3		p_2 já está na memória
14	1	2	0	1	✓	p_1 substitui p_3
15	2	2	0	1		p_2 já está na memória
16	0	2	0	1		p_0 já está na memória
17	1	2	0	1		p_1 já está na memória
18	7	7	0	1	✓	p_7 substitui p_2
19	0	7	0	1		p_0 já está na memória
20	1	7	0	1		p_1 já está na memória

Tabela 17.1: Aplicação do algoritmo de substituição ótimo.

Apesar de ter uma implementação simples, na prática este algoritmo não oferece bons resultados. Seu principal defeito é considerar somente a idade da página, sem levar em conta sua importância. Páginas carregadas na memória há muito tempo podem estar sendo frequentemente acessadas, como é o caso de áreas de memória contendo bibliotecas dinâmicas compartilhadas por muitos processos, ou páginas de processos servidores lançados durante a inicialização (*boot*) da máquina.

17.3.4 Algoritmo LRU

Uma aproximação implementável do algoritmo ótimo é proporcionada pelo algoritmo LRU (*Least Recently Used*, menos recentemente usado). Neste algoritmo, a escolha recai sobre as páginas que estão na memória há mais tempo **sem ser acessadas**. Assim, páginas antigas e menos usadas são as escolhas preferenciais. Páginas antigas mas de uso frequente não são penalizadas por este algoritmo, ao contrário do que ocorre no algoritmo FIFO. Pode-se observar facilmente que este algoritmo é simétrico do algoritmo OPT em relação ao tempo: enquanto o OPT busca as páginas que serão

t	página acessada	quadros q_o	q_1	q_2	falta de página?	ação realizada
0						situação inicial, quadros vazios
1	7	7			✓	p_7 é carregada em q_0
2	0	7	0		✓	p_0 é carregada em q_1
3	1	7	0	1	✓	p_1 é carregada em q_2
4	2	2	0	1	✓	p_2 substitui p_7 (carregada em $t = 1$)
5	0	2	0	1		p_0 já está na memória
6	3	2	3	1	✓	p_3 substitui p_0
7	0	2	3	0	✓	p_1 substitui p_1
8	4	4	3	0	✓	p_4 substitui p_2
9	2	4	2	0	✓	p_2 substitui p_3
10	3	4	2	3	✓	p_3 substitui p_0
11	0	0	2	3	✓	p_0 substitui p_4
12	3	0	2	3		p_3 já está na memória
13	2	0	2	3		p_2 já está na memória
14	1	0	1	3	✓	p_1 substitui p_2
15	2	0	1	2	✓	p_2 substitui p_3
16	0	0	1	2		p_0 já está na memória
17	1	0	1	2		p_1 já está na memória
18	7	7	1	2	✓	p_7 substitui p_0
19	0	7	0	2	✓	p_0 substitui p_1
20	1	7	0	1	✓	p_1 substitui p_2

Tabela 17.2: Aplicação do algoritmo de substituição FIFO.

acessadas "mais longe" no futuro do processo, o algoritmo LRU busca as páginas que foram acessadas "mais longe" no seu passado.

A aplicação do algoritmo LRU à cadeia de referências apresentada na Seção anterior, considerando uma memória física com 3 quadros, é apresentada na Tabela 17.3. Nesse caso, o algoritmo gera 12 faltas de página (três faltas a mais que o algoritmo ótimo).

O algoritmo LRU parte do pressuposto que páginas recentemente acessadas no passado provavelmente serão acessadas em um futuro próximo, e então evita removê-las da memória. Esta hipótese geralmente se verifica na prática, sobretudo se os processos respeitam o princípio da localidade de referência (Seção 15.9). Todavia, o desempenho do algoritmo LRU é prejudicado no caso de acessos com um padrão fortemente sequencial, ou seja, um certo número de páginas são acessadas em sequência e repetidamente (por exemplo: $p_1, p_2, p_3, \cdots, p_n, p_1, p_2, p_3, \cdots p_n, \cdots$). Nessa situação, o desempenho do algoritmo LRU será similar do FIFO.

t	página acessada	quadros q_o	q_1	q_2	falta de página?	ação realizada
0						situação inicial, quadros vazios
1	7	7			✓	p_7 é carregada em q_0
2	0	7	0		✓	p_0 é carregada em q_1
3	1	7	0	1	✓	p_1 é carregada em q_2
4	2	2	0	1	✓	p_2 substitui p_7 (há mais tempo sem acesso)
5	0	2	0	1		p_0 já está na memória
6	3	2	0	3	✓	p_3 substitui p_1
7	0	2	0	3		p_0 já está na memória
8	4	4	0	3	✓	p_4 substitui p_2
9	2	4	0	2	✓	p_2 substitui p_3
10	3	4	3	2	✓	p_3 substitui p_0
11	0	0	3	2	✓	p_0 substitui p_4
12	3	0	3	2		p_3 já está na memória
13	2	0	3	2		p_2 já está na memória
14	1	1	3	2	✓	p_1 substitui p_0
15	2	1	3	2		p_2 já está na memória
16	0	1	0	2	✓	p_0 substitui p_3
17	1	1	0	2		p_1 já está na memória
18	7	1	0	7	✓	p_7 substitui p_2
19	0	1	0	7		p_0 já está na memória
20	1	1	0	7		p_1 já está na memória

Tabela 17.3: Aplicação do algoritmo de substituição LRU.

17.3.5 Algoritmo RANDOM

Um algoritmo interessante consiste em escolher aleatoriamente as páginas a retirar da memória. O algoritmo aleatório pode ser útil em situações onde as abordagens LRU e FIFO tem desempenho ruim, como os padrões de acesso fortemente sequenciais discutidos na seção anterior. De fato, alguns sistemas operacionais mais antigos usavam essa abordagem em situações onde os demais algoritmos funcionavam mal.

17.3.6 Comparação entre algoritmos

O gráfico da Figura 17.4 permite a comparação dos algoritmos OPT, FIFO, LRU e RANDOM sobre a cadeia de referências apresentada na Seção 17.3.1, em função do número de quadros existentes na memória física. Pode-se observar que o melhor desempenho é do algoritmo OPT, enquanto o pior desempenho é proporcionado pelo algoritmo RANDOM.

A Figura 17.5 permite comparar o desempenho desses mesmos algoritmos de substituição de páginas em um cenário mais realista. A cadeia de referências

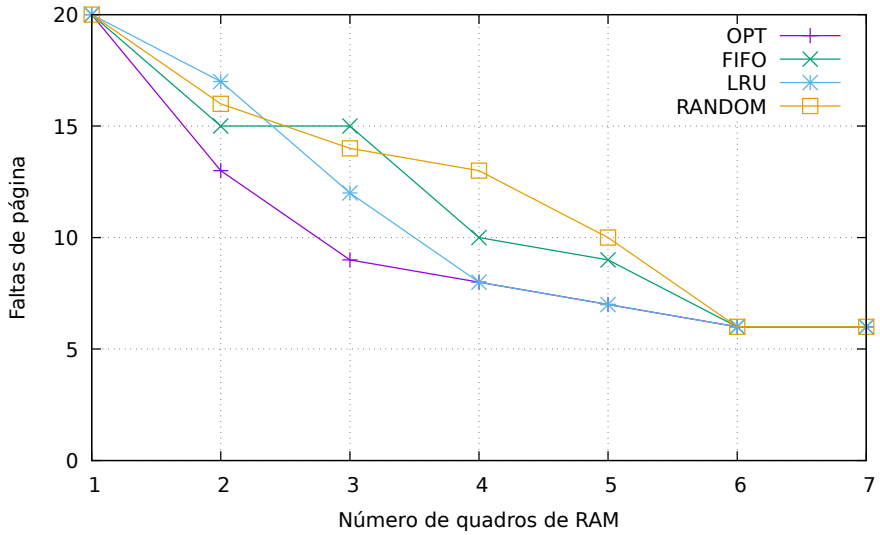


Figura 17.4: Comparação dos algoritmos de substituição de páginas.

usada corresponde a uma execução do compilador GCC (`http://gcc.gnu.org`) com 10^6 referências a 1.260 páginas distintas. Pode-se perceber as mesmas relações entre os desempenhos dos algoritmos.

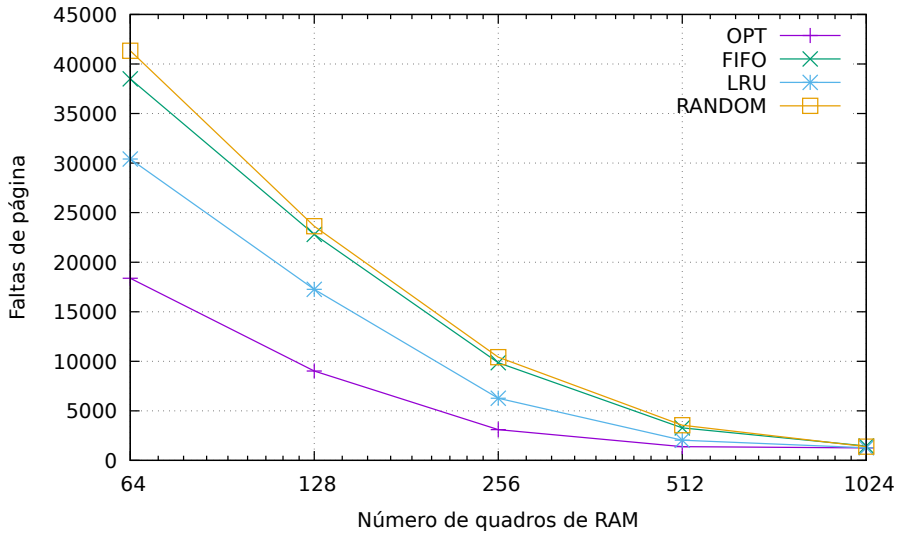


Figura 17.5: Comparação dos algoritmos, usando uma execução do compilador GCC.

17.4 Aproximações do algoritmo LRU

Embora possa ser implementado, o algoritmo LRU básico é pouco usado na prática, porque sua implementação exigiria registrar as datas de acesso às páginas a cada leitura ou escrita na memória, o que é difícil de implementar de forma eficiente em software e com custo proibitivo para implementar em hardware. Além disso, sua implementação exigiria varrer as datas de acesso de todas as páginas para buscar a página com acesso mais antigo (ou manter uma lista de páginas ordenadas por data

de acesso), o que exigiria muito processamento. Portanto, a maioria dos sistemas operacionais reais implementa algoritmos baseados em aproximações do LRU.

Esta seção apresenta alguns algoritmos simples que permitem se aproximar do comportamento LRU. Por sua simplicidade, esses algoritmos têm desempenho limitado e por isso somente são usados em sistemas operacionais mais simples. Como exemplos de algoritmos de substituição de páginas mais sofisticados e com maior desempenho podem ser citados o LIRS [Jiang and Zhang, 2002] e o ARC [Bansal and Modha, 2004].

17.4.1 Algoritmo da segunda chance

O algoritmo FIFO (Seção 17.3.3) move para a área de troca as páginas há mais tempo na memória, sem levar em conta seu histórico de acessos. Uma melhoria simples desse algoritmo consiste em analisar o bit de referência (Seção 15.6.2) de cada página candidata, para saber se ela foi acessada recentemente. Caso tenha sido, essa página recebe uma "segunda chance", voltando para o fim da fila com seu bit de referência ajustado para zero. Dessa forma, evita-se substituir páginas antigas mas muito acessadas. Todavia, caso todas as páginas sejam muito acessadas, o algoritmo vai varrer todas as páginas, ajustar todos os bits de referência para zero e acabará por escolher a primeira página da fila, como faria o algoritmo FIFO.

Uma forma eficiente de implementar este algoritmo é através de uma lista circular de números de página, ordenados de acordo com seu ingresso na memória. Um ponteiro percorre a lista sequencialmente, analisando os bits de referência das páginas e ajustando-os para zero à medida em que avança. Quando uma página vítima é encontrada, ela é movida para o disco e a página desejada é carregada na memória no lugar da vítima, com seu bit de referência ajustado para um (pois acaba de ser acessada). Essa implementação é conhecida como *algoritmo do relógio* e pode ser vista na Figura 17.6.

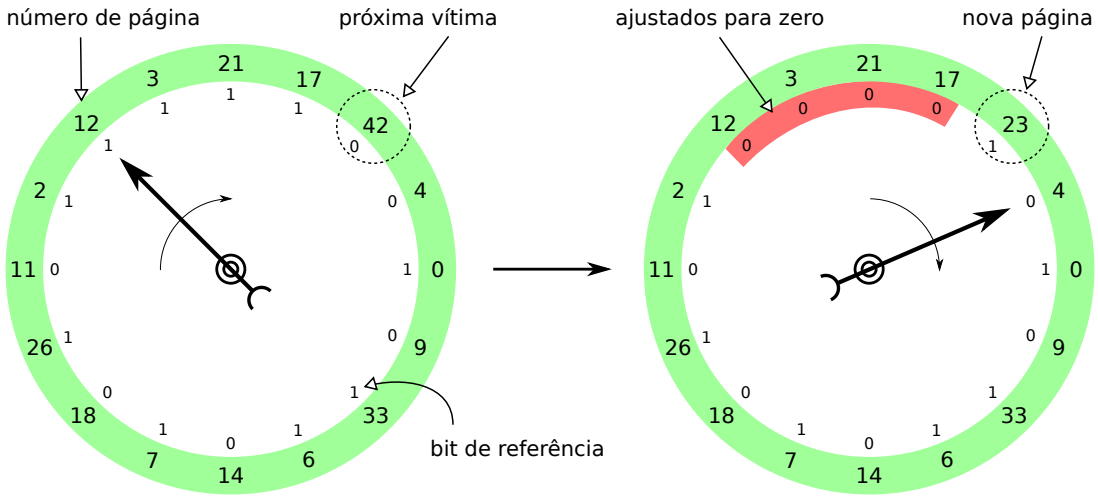


Figura 17.6: Algoritmo da segunda chance (ou do relógio).

17.4.2 Algoritmo NRU

O algoritmo da segunda chance leva em conta somente o bit de referência de cada página ao escolher as vítimas para substituição. O algoritmo NRU (*Not Recently*

Used, ou *não usada recentemente*) melhora essa escolha, ao considerar também o bit de modificação (*dirty bit*, vide Seção 15.6.2), que indica se o conteúdo de uma página foi modificado após ela ter sido carregada na memória.

Usando os bits R (referência) e M (modificação), é possível classificar as páginas em memória em quatro níveis de importância:

- 00 ($R = 0, M = 0$): páginas que não foram referenciadas recentemente e cujo conteúdo não foi modificado. São as melhores candidatas à substituição, pois podem ser simplesmente retiradas da memória.
- 01 ($R = 0, M = 1$): páginas que não foram referenciadas recentemente, mas cujo conteúdo já foi modificado. Não são escolhas tão boas, porque terão de ser gravadas na área de troca antes de serem substituídas.
- 10 ($R = 1, M = 0$): páginas referenciadas recentemente, cujo conteúdo permanece inalterado. São provavelmente páginas de código que estão sendo usadas ativamente e serão referenciadas novamente em breve.
- 11 ($R = 1, M = 1$): páginas referenciadas recentemente e cujo conteúdo foi modificado. São a pior escolha, porque terão de ser gravadas na área de troca e provavelmente serão necessárias em breve.

O algoritmo NRU consiste simplesmente em tentar substituir primeiro páginas do nível 0; caso não encontre, procura candidatas no nível 1 e assim sucessivamente. Pode ser necessário percorrer várias vezes a lista circular até encontrar uma página adequada para substituição.

17.4.3 Algoritmo do envelhecimento

Outra possibilidade de melhoria do algoritmo da segunda chance consiste em usar os bits de referência das páginas para construir *contadores de acesso* às mesmas. A cada página é associado um contador inteiro com N bits (geralmente 8 bits são suficientes). Periodicamente, o algoritmo varre as tabelas de páginas, lê os bits de referência e agrega seus valores aos contadores de acessos das respectivas páginas. Uma vez lidos, os bits de referência são ajustados para zero, para registrar as referências de páginas que ocorrerão durante próximo período.

O valor lido de cada bit de referência não deve ser simplesmente somado ao contador, por duas razões: o contador chegaria rapidamente ao seu valor máximo (*overflow*) e a simples soma não permitiria diferenciar acessos recentes dos mais antigos. Por isso, outra solução foi encontrada: cada contador é deslocado para a direita 1 bit, descartando o bit menos significativo (LSB - *Least Significant Bit*). Em seguida, o valor do bit de referência é colocado na primeira posição à esquerda do contador, ou seja, em seu bit mais significativo (MSB - *Most Significant Bit*). Dessa forma, acessos mais recentes têm um peso maior que acessos mais antigos, e o contador nunca ultrapassa seu valor máximo.

O exemplo a seguir mostra a evolução dos contadores para quatro páginas distintas, usando os valores dos respectivos bits de referência R. Os valores decimais dos contadores estão indicados entre parênteses, para facilitar a comparação. Observe que as páginas acessadas no último período (p_2 e p_4) têm seus contadores aumentados, enquanto aquelas não acessadas (p_1 e p_3) têm seus contadores diminuídos.

$$\begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{array}\!\!\begin{bmatrix} R \\ \boxed{0} \\ \boxed{1} \\ \boxed{0} \\ \boxed{1} \end{bmatrix} \text{ com } \begin{bmatrix} \textit{contadores} & \\ \boxed{0000\ 0011} & (3) \\ \boxed{0011\ 1101} & (61) \\ \boxed{1010\ 1000} & (168) \\ \boxed{1110\ 0011} & (227) \end{bmatrix} \Longrightarrow \begin{bmatrix} R \\ \boxed{0} \\ \boxed{0} \\ \boxed{0} \\ \boxed{0} \end{bmatrix} \text{ e } \begin{bmatrix} \textit{contadores} & \\ \boxed{\mathbf{0}000\ 0001} & (1) \\ \boxed{\mathbf{1}001\ 1110} & (158) \\ \boxed{\mathbf{0}101\ 0100} & (84) \\ \boxed{\mathbf{1}111\ 0001} & (241) \end{bmatrix}$$

O contador construído por este algoritmo constitui uma aproximação razoável do algoritmo LRU: páginas menos acessadas "envelhecerão", ficando com contadores menores, enquanto páginas mais acessadas permanecerão "jovens", com contadores maiores. Por essa razão, esta estratégia é conhecida como *algoritmo do envelhecimento* [Tanenbaum, 2003], ou *algoritmo dos bits de referência adicionais* [Silberschatz et al., 2001].

17.5 Conjunto de trabalho

A localidade de referências (estudada na Seção 15.9) mostra que os processos normalmente acessam apenas uma pequena fração de suas páginas a cada instante. O conjunto de páginas acessadas na história recente de um processo é chamado *Conjunto de Trabalho* (*Working Set*, ou *ws*) [Denning, 1980, 2006]. A composição do conjunto de trabalho é dinâmica, variando à medida em que o processo executa e evolui seu comportamento, acessando novas páginas e deixando de acessar outras. Para ilustrar esse conceito, consideremos a cadeia de referências apresentada na Seção 17.3.1. Considerando como história recente as últimas n páginas acessadas pelo processo, a evolução do conjunto de trabalho *ws* do processo que gerou aquela cadeia é apresentada na Tabela 17.4.

O tamanho e a composição do conjunto de trabalho dependem do número de páginas consideradas em sua história recente (o valor n na Tabela 17.4). Em sistemas reais, essa dependência não é linear, mas segue uma proporção exponencial inversa, devido à localidade de referências. Por essa razão, a escolha precisa do tamanho da história recente a considerar não é crítica. Esse fenômeno pode ser observado na Tabela 17.4: assim que a localidade de referências se torna mais forte (a partir de $t = 12$), os três conjuntos de trabalho ficam muito similares. Outro exemplo é apresentado na Figura 17.7, que mostra o tamanho médio dos conjuntos de trabalhos observados na execução do programa *gThumb* (analisado na Seção 15.9), em função do tamanho da história recente considerada (em número de páginas referenciadas).

Se um processo tiver todas as páginas de seu conjunto de trabalho carregadas na memória, ele sofrerá poucas faltas de página, pois somente acessos a novas páginas poderão gerar faltas. Essa constatação permite delinear um algoritmo simples para substituição de páginas: só substituir páginas que não pertençam ao conjunto de trabalho de nenhum processo ativo. Contudo, esse algoritmo é difícil de implementar, pois exigiria manter atualizado o conjunto de trabalho de cada processo a cada acesso à memória, o que teria um custo computacional proibitivo.

Uma alternativa mais simples e eficiente de implementar seria verificar que páginas cada processo acessou recentemente, usando a informação dos respectivos bits de referência. Essa é a base do algoritmo *WSClock* (*Working Set Clock*) [Carr and Hennessy, 1981], uma modificação do algoritmo do relógio (Seção 17.4.1). Como no

t	**página**	$ws(n = 3)$	$ws(n = 4)$	$ws(n = 5)$
1	7	7	7	7
2	0	0, 7	0, 7	0, 7
3	1	0, 1, 7	0, 1, 7	0, 1, 7
4	2	0, 1, 2	0, 1, 2, 7	0, 1, 2, 7
5	0	0, 1, 2	0, 1, 2	0, 1, 2, 7
6	3	0, 2, 3	0, 1, 2, 3	0, 1, 2, 3
7	0	0, 3	0, 2, 3	0, 1, 2, 3
8	4	0, 3, 4	0, 3, 4	0, 2, 3, 4
9	2	0, 2, 4	0, 2, 3, 4	0, 2, 3, 4
10	3	2, 3, 4	0, 2, 3, 4	0, 2, 3, 4
11	0	0, 2, 3	0, 2, 3, 4	0, 2, 3, 4
12	3	0, 3	0, 2, 3	0, 2, 3, 4
13	2	0, 2, 3	0, 2, 3	0, 2, 3
14	1	1, 2, 3	0, 1, 2, 3	0, 1, 2, 3
15	2	1, 2	1, 2, 3	0, 1, 2, 3
16	0	0, 1, 2	0, 1, 2	0, 1, 2, 3
17	1	0, 1, 2	0, 1, 2	0, 1, 2
18	7	0, 1, 7	0, 1, 2, 7	0, 1, 2, 7
19	0	0, 1, 7	0, 1, 7	0, 1, 2, 7
20	1	0, 1, 7	0, 1, 7	0, 1, 7

Tabela 17.4: Conjuntos de trabalho *ws* para $n = 3$, $n = 4$ e $n = 5$.

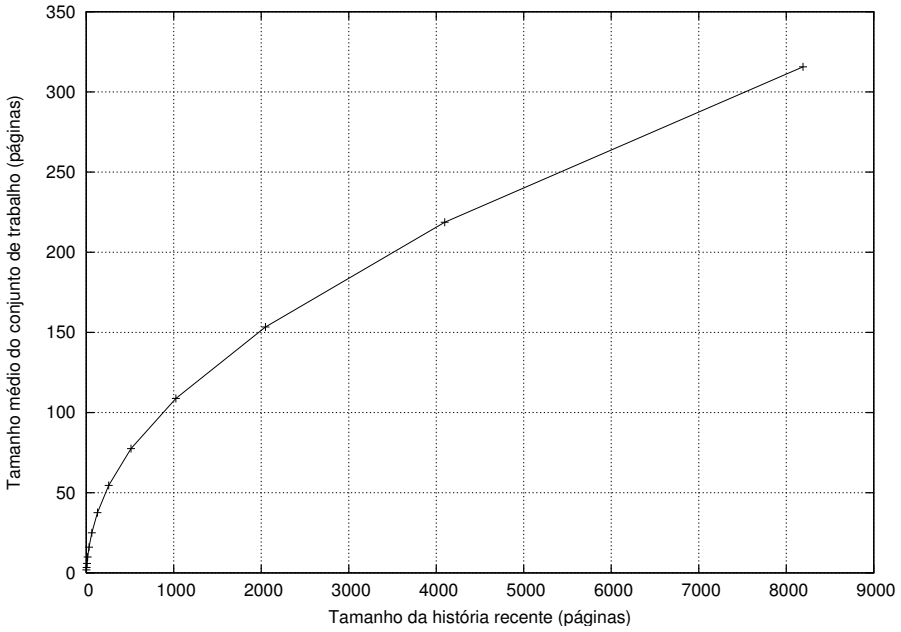


Figura 17.7: Tamanho do conjunto de trabalho do programa *gThumb*.

algoritmo do relógio, as páginas carregadas na memória também são organizadas em uma lista circular, por ordem de instante de carga na memória. Cada página p dessa lista tem uma data de último acesso $t_a(p)$.

No *WSClock*, define-se um prazo de validade τ para as páginas, entre dezenas e centenas de milissegundos; a idade $i(p)$ de uma página p na memória é definida como a diferença entre a data de seu último acesso $t_a(p)$ e o instante atual t_{now} ($i(p) = t_{now} - t_a(p)$). Quando há necessidade de substituir páginas, o ponteiro percorre a lista buscando páginas "vítimas":

1. Ao encontrar uma página p referenciada (com $R(p) = 1$), a data de seu último acesso é atualizada ($t_a(p) = t_{now}$), seu bit de referência é limpo ($R(p) = 0$) e o ponteiro do relógio avança, ignorando-a.

2. Ao encontrar uma página p não-referenciada (com $R(p) = 0$):

   (a) se a idade de p for válida ($i(p) \leq \tau$), a página está no conjunto de trabalho e deve ser ignorada;

   (b) caso contrário ($i(p) > \tau$), p está fora do conjunto de trabalho. Neste caso:

      i. Se $M(p) = 0$, a página p não foi modificada e pode ser substituída;

      ii. caso contrário ($M(p) = 1$), agenda-se uma escrita dessa página em disco e o ponteiro do relógio avança, ignorando-a.

3. Caso o ponteiro dê uma volta completa na lista e não encontre página com $i(p) > \tau$, $R = 0$ e $M = 0$:

   (a) substituir a página mais antiga (com o menor $t_a(p)$) com $R = 0$ e $M = 0$;

   (b) se não achar, substituir a página mais antiga com $M = 0$ e $R = 1$;

   (c) se não achar, substituir a página mais antiga com $R = 0$;

   (d) se não achar, substituir a página mais antiga.

O algoritmo *WSClock* pode ser implementado de forma eficiente, porque a data último acesso de cada página não precisa ser atualizada a cada acesso à memória, mas somente quando a referência da página na lista circular é visitada pelo ponteiro do relógio (caso $R = 1$). Todavia, esse algoritmo não é uma implementação "pura" do conceito de conjunto de trabalho, mas uma composição de conceitos de vários algoritmos: FIFO e segunda chance (estrutura e percurso do relógio), Conjuntos de trabalho (divisão das páginas em dois grupos conforme sua idade), LRU (escolha das páginas com datas de acesso mais antigas) e NRU (preferência às páginas não modificadas).

17.6 A anomalia de Belady

Espera-se que, quanto mais memória física um sistema possua, menos faltas de página ocorram. Todavia, esse comportamento intuitivo não se verifica em todos os algoritmos de substituição de páginas. Alguns algoritmos, como o FIFO, podem apresentar um comportamento estranho: ao aumentar o número de quadros de memória, o número de faltas de página geradas pelo algoritmo aumenta, ao invés de diminuir. Esse

comportamento atípico de alguns algoritmos foi estudado pelo matemático húngaro Laslo Belady nos anos 60, sendo por isso denominado *anomalia de Belady*.

A seguinte cadeia de referências permite observar esse fenômeno; o comportamento dos algoritmos OPT, FIFO e LRU ao processar essa cadeia pode ser visto na Figura 17.8, que exibe o número de faltas de página em função do número de quadros de memória disponíveis no sistema. A anomalia pode ser observada no algoritmo FIFO: ao aumentar a memória de 4 para 5 quadros, esse algoritmo passa de 22 para 24 faltas de página.

$$0, 1, 2, 3, 4, 0, 1, 2, 5, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 0, 1, 2, 5, 0, 1, 2, 3, 4, 5$$

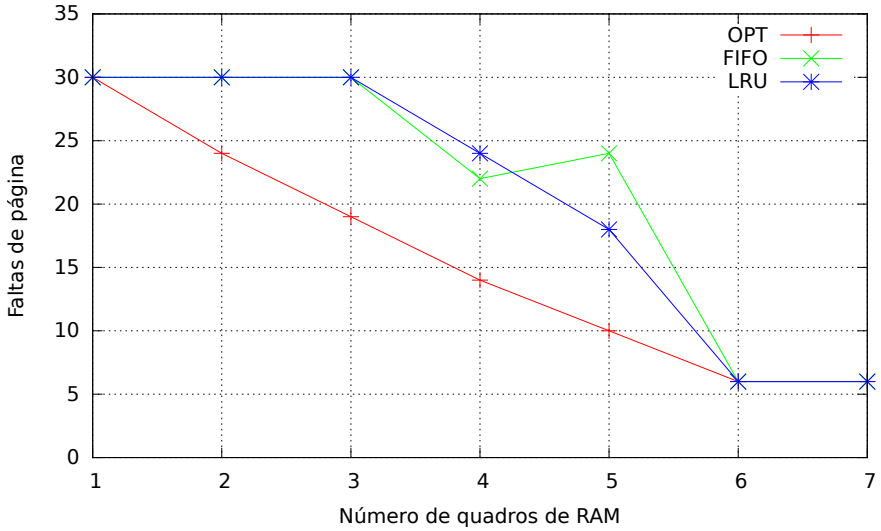


Figura 17.8: A anomalia de Belady.

Estudos demonstraram que uma família de algoritmos denominada *algoritmos de pilha* (à qual pertencem os algoritmos OPT e LRU, entre outros) não apresenta a anomalia de Belady [Tanenbaum, 2003].

17.7 Thrashing

Na Seção 17.2.2, foi demonstrado que o tempo médio de acesso à memória RAM aumenta significativamente à medida em que aumenta a frequência de faltas de página. Caso a frequência de faltas de páginas seja muito elevada, o desempenho do sistema como um todo pode ser severamente prejudicado.

Conforme discutido na Seção 17.5, cada processo tem um conjunto de trabalho, ou seja, um conjunto de páginas que devem estar na memória para sua execução naquele momento. Se o processo tiver uma boa localidade de referência, esse conjunto é pequeno e varia lentamente. Caso a localidade de referência seja ruim, o conjunto de trabalho geralmente é grande e muda rapidamente. Enquanto houver espaço na memória RAM para os conjuntos de trabalho dos processos ativos, não haverão problemas. Contudo, caso a soma de todos os conjuntos de trabalho dos processos prontos para execução seja maior que a memória RAM disponível no sistema, poderá ocorrer um fenômeno conhecido como *thrashing* [Denning, 1980, 2006].

No *thrashing*, a memória RAM não é suficiente para todos os processos ativos, portanto muitos processos não conseguem ter seus conjuntos de trabalho totalmente carregados na memória. Cada vez que um processo recebe o processador, executa algumas instruções, gera uma falta de página e volta ao estado suspenso, até que a página faltante seja trazida de volta à RAM. Todavia, para trazer essa página à RAM será necessário abrir espaço na memória, transferindo algumas páginas (de outros processos) para o disco. Quanto mais processos estiverem nessa situação, maior será a atividade de paginação e maior o número de processos no estado suspenso, aguardando páginas.

A Figura 17.9 ilustra o conceito de *thrashing*: ela mostra a taxa de uso do processador (quantidade de processos na fila de prontos) em função do número de processos ativos no sistema. Na zona à esquerda não há *thrashing*, portanto a taxa de uso do processador aumenta com o aumento do número de processos. Caso esse número aumente muito, a memória RAM não será suficiente para todos os conjuntos de trabalho e o sistema entra em uma situação de *thrashing*: muitos processos passarão a ficar suspensos aguardando a paginação, diminuindo a taxa de uso do processador. Quanto mais processos ativos, menos o processador será usado e mais lento ficará o sistema. Pode-se observar que um sistema ideal com memória infinita não teria esse problema, pois sempre haveria memória suficiente para todos os processos ativos.

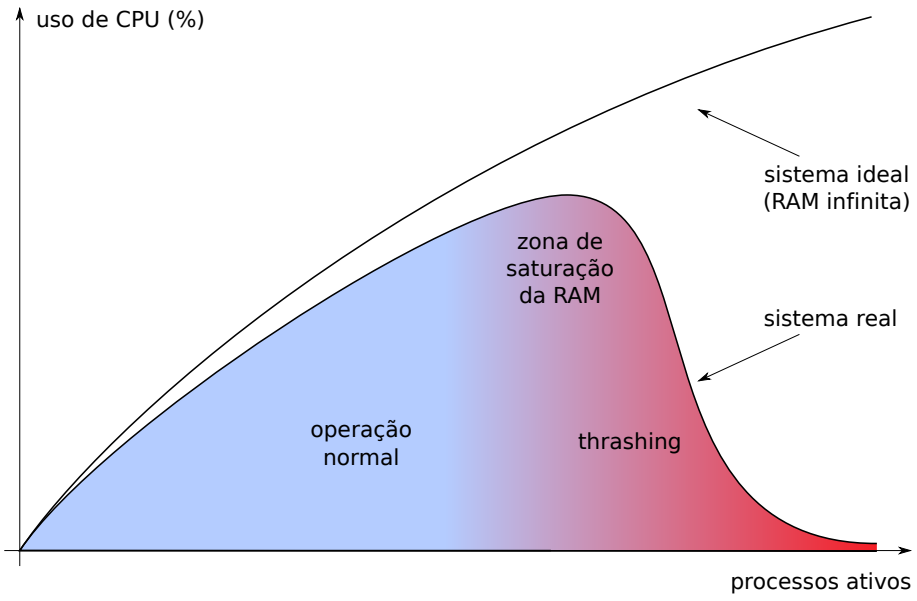


Figura 17.9: Comportamento de um sistema com *thrashing*.

Um sistema operacional sob *thrashing* tem seu desempenho muito prejudicado, a ponto de parar de responder ao usuário e se tornar inutilizável. Por isso, esse fenômeno deve ser evitado. Para tal, pode-se aumentar a quantidade de memória RAM do sistema, limitar a quantidade máxima de processos ativos, ou mudar a política de escalonamento dos processos durante o *thrashing*, para evitar a competição pela memória disponível. Vários sistemas operacionais adotam medidas especiais para situações de *thrashing*, como suspender em massa os processos ativos, adotar uma política de escalonamento de processador que considere o uso da memória, aumentar o *quantum* de processador para cada processo ativo, ou simplesmente abortar os processos com maior alocação de memória ou com maior atividade de paginação.

Referências

S. Bansal and D. Modha. CAR: Clock with adaptive replacement. In *USENIX Conference on File and Storage Technologies*, April 2004.

R. Carr and J. Hennessy. WSclock - a simple and effective algorithm for virtual memory management. In *ACM symposium on Operating systems principles*, 1981.

P. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, 6 (1):64–84, January 1980.

P. J. Denning. The locality principle. In J. Barria, editor, *Communication Networks and Computer Systems*, chapter 4, pages 43–67. Imperial College Press, 2006.

S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS Intl Conference on Measurement and Modeling of Computer Systems*, pages 31–42, 2002.

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

A. Tanenbaum. *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall, 2003.

R. Uhlig and T. Mudge. Trace-driven memory simulation: a survey. *ACM Computing Surveys*, 29(2):128–170, June 1997.

Capítulo 18

Tópicos em gestão de memória

Este capítulo traz tópicos de estudo específicos, que aprofundam ou complementam os temas apresentados nesta parte do livro, mas cuja leitura não é essencial para a compreensão do conteúdo principal da disciplina. Algumas seções deste capítulo podem estar incompletas ou não ter sido revisadas.

18.1 Compartilhamento de memória

A memória RAM é um recurso escasso, que deve ser usado de forma eficiente. Nos sistemas atuais, é comum ter várias instâncias do mesmo programa em execução, como várias instâncias de editores de texto, de navegadores, etc. Em servidores, essa situação pode ser ainda mais frequente, com centenas ou milhares de instâncias do mesmo programa carregadas na memória. Por exemplo, em um servidor de e-mail UNIX, cada cliente que se conecta através de protocolos de rede como POP3 ou IMAP terá um processo correspondente no servidor, para atender suas consultas de e-mail (Figura 18.1). Todos esses processos operam com dados distintos (pois atendem a usuários distintos), mas executam o mesmo código de servidor. Assim, centenas ou milhares de cópias do mesmo código executável poderão coexistir na memória do sistema.

Conforme visto na Seção 14.2, a estrutura típica da memória de um processo contém seções separadas para código, dados, pilha e *heap*. Normalmente, a seção de código não tem seu conteúdo modificado durante a execução, portanto geralmente essa seção é protegida contra escritas (*read-only*). Assim, seria possível *compartilhar* essa seção entre todos os processos que executam o mesmo código, economizando memória RAM.

O compartilhamento de código entre processos pode ser implementado de forma muito simples e transparente para os processos envolvidos, através dos mecanismos de tradução de endereços oferecidos pela MMU, como segmentação e paginação. No caso da segmentação, basta fazer com que os segmentos de código dos processos apontem para o mesmo trecho da memória física, como indica a Figura 18.2. É importante observar que o compartilhamento é transparente para os processos: cada processo continua a acessar endereços lógicos em seu próprio segmento de código.

No caso da paginação, a unidade básica de compartilhamento é a página. Assim, a tabela de páginas de cada processo envolvido é ajustada para referenciar os mesmos quadros de memória física. É importante observar que, embora referenciem os mesmos endereços físicos, as páginas compartilhadas podem ter endereços lógicos distintos. A Figura 18.3 ilustra o compartilhamento de páginas entre dois processos.

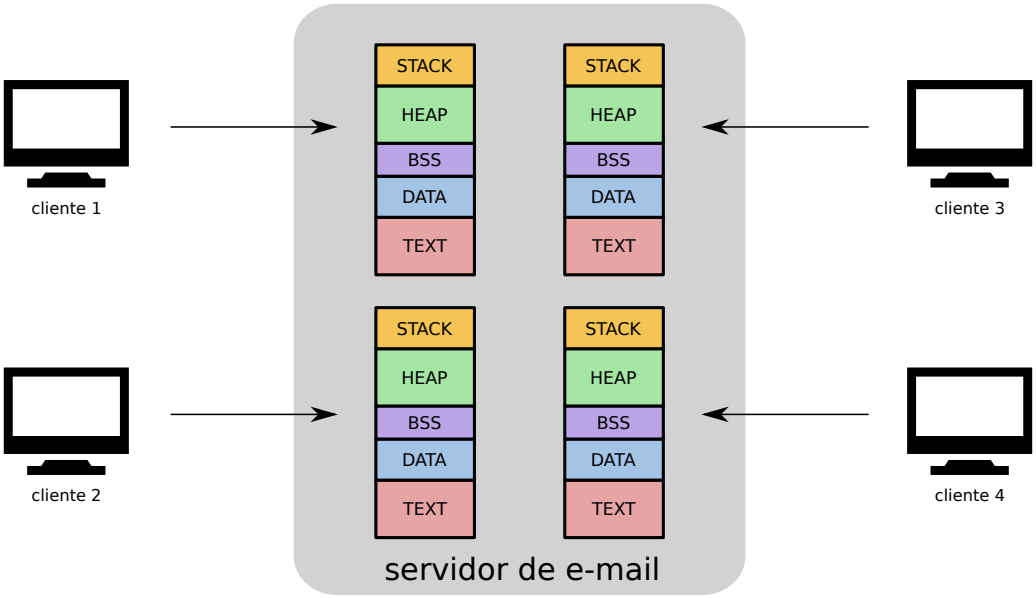


Figura 18.1: Várias instâncias do mesmo processo.

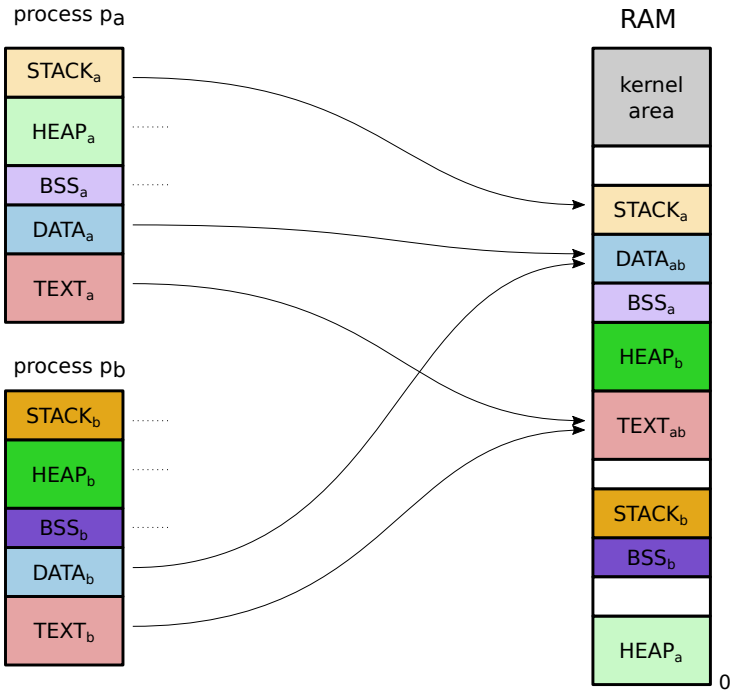


Figura 18.2: Compartilhamento de segmentos.

O compartilhamento das seções de código permite proporcionar uma grande economia no uso da memória física, sobretudo em servidores e sistemas multiusuários. Por exemplo: consideremos um editor de textos que necessite de 100 MB de memória para executar, dos quais 60 MB são ocupados por código executável e bibliotecas. Sem o compartilhamento das seções de código, 10 instâncias do editor consumiriam 1.000 MB de memória; com o compartilhamento, esse consumo cairia para 460 MB: uma área compartilhada com 60 MB contendo o código e mais 10 áreas de 40 MB cada uma, contendo os dados e pilha de cada processo.

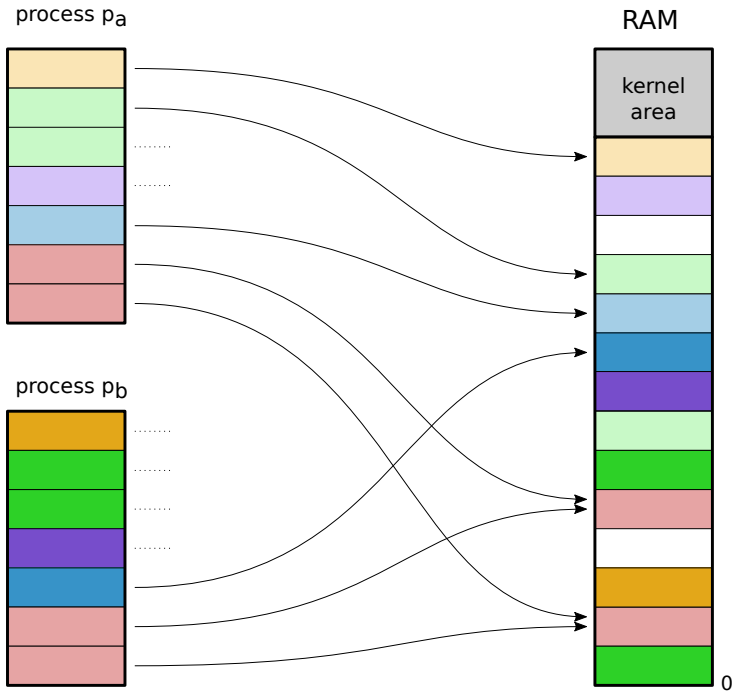


Figura 18.3: Compartilhamento de páginas.

Áreas de memória compartilhada também podem ser usadas para permitir a comunicação entre processos. Para tal, dois ou mais processos solicitam ao núcleo o mapeamento de uma área de memória sobre a qual ambos podem ler e escrever. Como os endereços lógicos acessados nessa área serão mapeados sobre a mesma área de memória física, o que cada processo escrever nessa área poderá ser lido pelos demais, imediatamente. A Seção 9.3 traz informações mais detalhadas sobre a comunicação entre processos através de memória compartilhada.

18.2 Copy-on-write (COW)

O mecanismo de compartilhamento de memória não é restrito apenas às seções de código. Em princípio, toda área de memória protegida contra escrita pode ser compartilhada, o que pode incluir áreas de dados constantes, como tabelas de constantes, textos de ajuda, etc., proporcionando ainda mais economia de memória.

A técnica conhecida como *copy-on-write* (CoW, ou *copiar ao escrever*), é uma estratégia usada quando o núcleo do SO precisa copiar páginas de um espaço de memória para outro. Ela pode ser aplicada, por exemplo, quando um processo cria outro através da chamada de sistema `fork()` (vide Seção 5.3). Nessa chamada, deve ser feita uma cópia do conteúdo de memória do processo pai para o processo filho.

Na técnica CoW, ao invés de copiar as páginas do processo pai para o filho, o núcleo compartilha as páginas e as marca como "somente leitura" e "copy-on-write", usando os flags das tabelas de página (Seção 15.6.2). Quando um dos processos tentar escrever em uma dessas páginas, a proteção contra escrita irá provocar uma falta de página, ativando o núcleo. Este então irá verificar que se trata de uma página "copy on write", criará uma cópia separada daquela página para o processo que deseja fazer a escrita, e removerá a proteção contra escrita dessa cópia.

Os principais passos dessa estratégia estão ilustrados na Figura 18.4 e detalhados a seguir:

1. Dois processos têm páginas compartilhadas pelo mecanismo CoW; as páginas somente podem ser acessadas em leitura;
2. o processo p_a tenta escrever em uma página compartilhada, provocando uma falta de página que ativa o núcleo do SO;
3. o núcleo faz uma cópia da página em outro quadro de memória RAM;
4. o núcleo ajusta a tabela de páginas de p_a para apontar para a cópia da página, limpa o flag *CoW* e permite a escrita na cópia;
5. o processo p_a continua a executar e consegue completar sua operação de escrita;
6. p_b continua a acessar o conteúdo original da página.

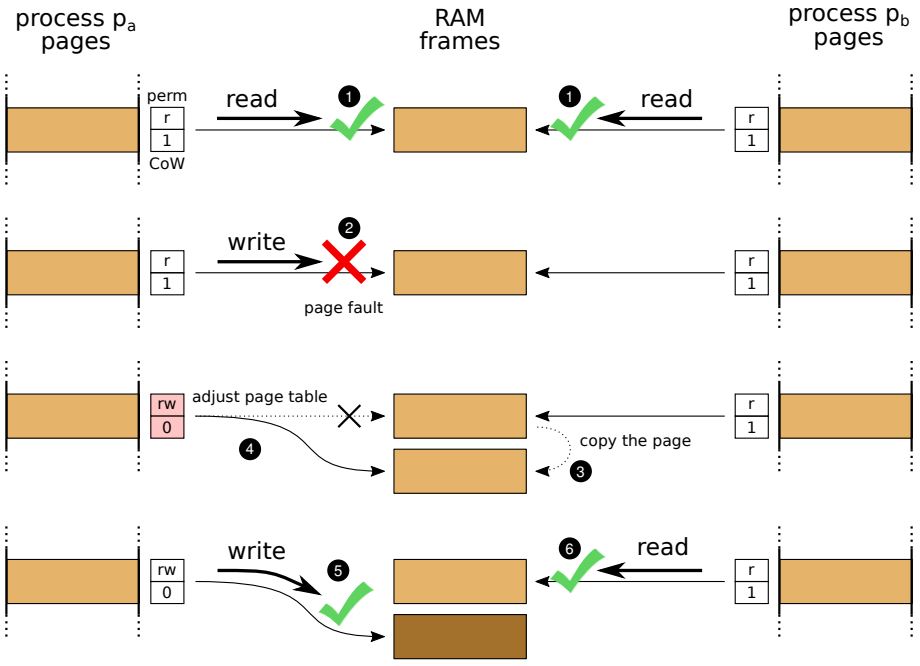


Figura 18.4: Estratégia de compartilhamento com *Copy-on-Write*.

Todo esse procedimento é feito de forma transparente para os processos envolvidos, visando compartilhar ao máximo as áreas de memória dos processos e assim otimizar o uso da RAM. Esse mecanismo é mais efetivo em sistemas baseados em páginas, porque normalmente as páginas são menores que os segmentos. A maioria dos sistemas operacionais atuais (Linux, Windows, Solaris, FreeBSD, etc.) usa esse mecanismo.

18.3 Mapeamento de arquivo em memória

Uma funcionalidade importante dos sistemas operacionais atuais é o mapeamento de arquivos em memória, que surgiu no SunOS 4.0 [Vahalia, 1996]. Esse

mapeamento consiste em associar uma área específica de memória do processo a um arquivo em disco: cada byte no arquivo corresponderá então a um byte naquela área de memória, sequencialmente. A Figura 18.5 ilustra esse conceito.

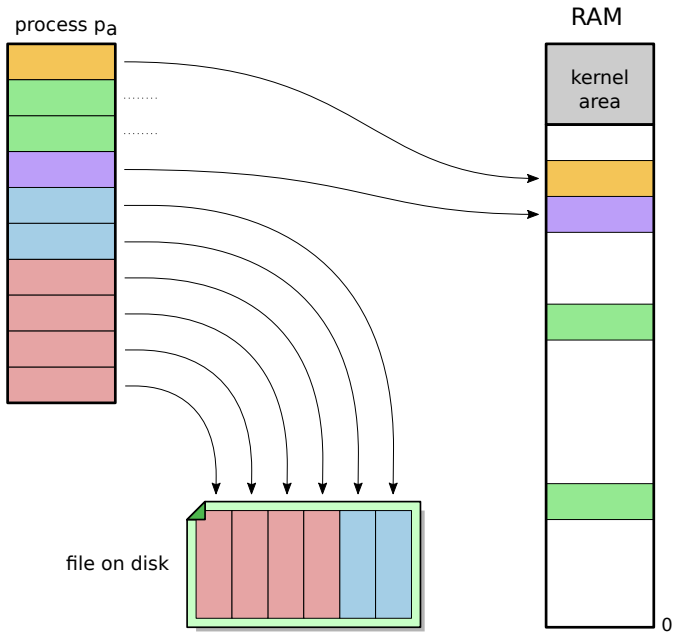


Figura 18.5: Mapeamento de arquivo em memória.

O conteúdo do arquivo mapeado pode ser acessado pelo processo através de leitura e escrita de bytes naquela área de memória que o mapeia. É importante observar que o mapeamento não implica em carregar o arquivo inteiro na memória. Ao invés disso, é criada uma área de memória com o mesmo tamanho do arquivo e suas páginas são ajustadas para "acesso proibido", usando os flags da tabela de páginas (Seção 15.6.2). Quando o processo tentar acessar uma posição de memória naquela área, a MMU irá gerar uma falta da página para o núcleo, que irá então buscar e carregar naquela página o conteúdo correspondente do arquivo mapeado. Esse procedimento é denominado *paginação sob demanda* (*demand paging*).

Os principais passos da paginação sob demanda estão ilustrados na Figura 18.6 e detalhados a seguir:

1. Um arquivo é mapeado em uma área de memória do processo; as páginas dessa área são marcadas como inacessíveis;

2. o processo p_a tenta acessar um dado em uma página da área mapeada, provocando uma falta de página que ativa o núcleo do SO;

3. o núcleo carrega o conteúdo correspondente do arquivo na memória;

4. o núcleo ajusta a tabela de páginas de p_a;

5. o processo p_a continua a executar e consegue completar seu acesso.

Em relação às operações de escrita, os mapeamentos podem ser *compartilhados* ou *privados*. Em um mapeamento compartilhado, as escritas feitas pelo processo na área mapeada são copiadas no arquivo, para que possam ser vistas por outros processos que

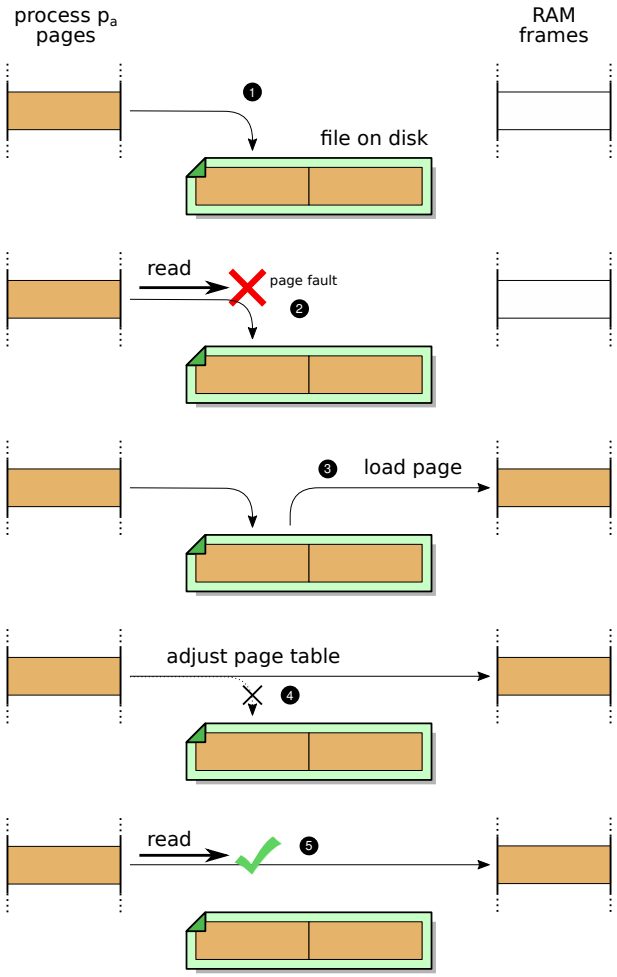


Figura 18.6: Paginação sob demanda.

abrirem aquele arquivo. No mapeamento privado, é usada a técnica *copy on write* para não propagar as escritas ao arquivo mapeado, que permanece intacto.

A paginação sob demanda é particularmente interessante para o lançamento de processos. Um arquivo executável é estruturado em seções, que contêm código, dados inicializados, etc., refletindo as seções de memória de um processo (Seção 14.2). A carga do código de um novo processo pode então ser feita pelo simples mapeamento das seções correspondentes do programa executável na memória do processo. À medida em que o processo executar, ele irá gerar faltas de páginas e provocar a carga das páginas correspondentes do programa executável. Com isso, somente os trechos de código efetivamente executados serão carregados na memória, o que é particularmente útil em programas grandes.

Outra vantagem do mapeamento de arquivos em memória é o compartilhamento de código executável. Caso mais processos que executem o mesmo programa sejam lançados, estes poderão usar as páginas já carregadas em memória pelo primeiro processo, gerando economia de memória e rapidez de carga.

Referências

U. Vahalia. *UNIX Internals – The New Frontiers*. Prentice-Hall, 1996.

Parte V

Gestão de entrada/saída

Capítulo 19

Hardware de entrada/saída

19.1 Introdução

Um computador é constituído basicamente de processadores, memória RAM e **dispositivos de entrada e saída**, também chamados de **periféricos**. Os dispositivos de entrada/saída permitem a interação do computador com o mundo exterior de várias formas, como por exemplo:

- interação com os usuários através de *mouse*, teclado, tela gráfica, tela de toque e *joystick*;
- escrita e leitura de dados em discos rígidos, SSDs, CD-ROMs, DVD-ROMs e *pen-drives*;
- impressão de informações através de impressoras e plotadoras;
- captura e reprodução de áudio e vídeo, como câmeras, microfones e alto-falantes;
- comunicação com outros computadores, através de redes LAN, *wifi*, *Bluetooth* e de telefonia celular.

Esses exemplos são típicos de computadores pessoais e de computadores menores, como os *smartphones* (Figura 19.1). Já em ambientes industriais, é comum encontrar dispositivos de entrada/saída específicos para a monitoração e controle de máquinas e processos de produção, como tornos de comando numérico, braços robotizados e controladores de processos químicos. Por sua vez, o computador embarcado em um carro conta com dispositivos de entrada para coletar dados do combustível e do funcionamento do motor e dispositivos de saída para controlar a injeção eletrônica e a tração dos pneus, por exemplo.

É bastante óbvio que um computador não tem muita utilidade sem dispositivos periféricos, pois o objetivo básico da imensa maioria dos computadores é receber dados, processá-los e devolver resultados aos seus usuários, sejam eles seres humanos, outros computadores ou processos físicos/químicos externos.

Os primeiros sistemas de computação, construídos nos anos 1940, eram destinados a cálculos matemáticos e por isso possuíam dispositivos de entrada/saída rudimentares, que apenas permitiam carregar/descarregar programas e dados diretamente na memória principal. Em seguida surgiram os terminais compostos de teclado e monitor de texto, para facilitar a leitura e escrita de dados, e os discos rígidos, como

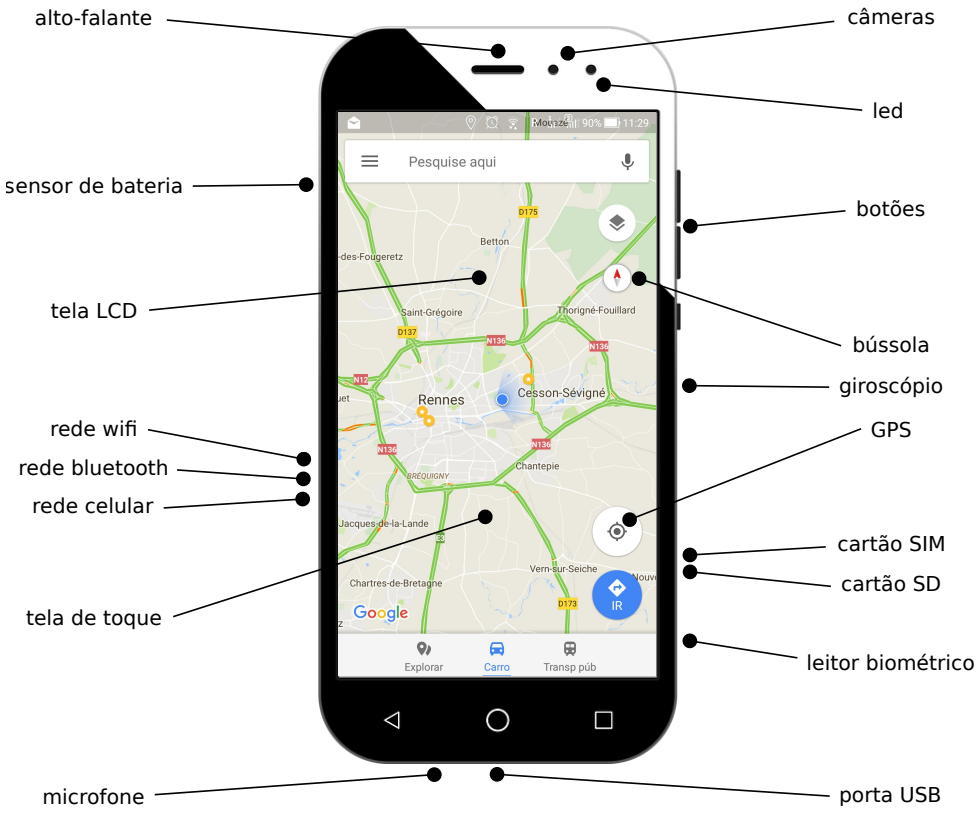


Figura 19.1: Um *smartphone* com seus dispositivos de entrada e saída.

meio de armazenamento persistente de dados e programas. Hoje, dispositivos de entrada/saída dos mais diversos tipos podem estar conectados a um computador. A grande diversidade de dispositivos periféricos é um dos maiores desafios presentes na construção e manutenção de um sistema operacional, pois cada um deles tem especificidades e exige mecanismos de acesso específicos.

Este capítulo apresenta uma visão geral das estruturas de hardware associadas aos dispositivos de entrada/saída presentes em computadores convencionais para a interação com o usuário, armazenamento de dados e comunicação.

19.2 Componentes de um dispositivo

Conceitualmente, a entrada de dados em um computador inicia com um **sensor** capaz de converter uma informação externa (física ou química) em um sinal elétrico analógico. Como exemplos de sensores temos o microfone, as chaves internas das teclas de um teclado ou o foto-diodo de um leitor de DVDs. O sinal elétrico analógico fornecido pelo sensor é então aplicado a um **conversor analógico-digital** (CAD), que o transforma em informação digital (sequências de bits). Essa informação digital é armazenada em um *buffer* que pode ser acessado pelo processador através de um **controlador de entrada**.

Uma saída de dados inicia com o envio de dados do processador a um **controlador de saída**, através do barramento. Os dados enviados pelo processador são armazenados em um *buffer* interno do controlador e a seguir convertidos em um sinal elétrico analógico, através de um **conversor digital-analógico** (CDA). Esse sinal será

aplicado a um **atuador**[1] que irá convertê-lo em efeitos físicos perceptíveis ao usuário. Exemplos simples de atuadores são a cabeça de impressão e os motores de uma impressora, um alto-falante, uma tela gráfica, etc.

Vários dispositivos combinam funcionalidades tanto de entrada quanto de saída, como os discos rígidos: o processador pode ler dados gravados no disco rígido (entrada), mas para isso precisa ativar o motor que faz girar o disco e posicionar adequadamente a cabeça de leitura (saída). O mesmo ocorre com uma placa de áudio de um PC convencional, que pode tanto capturar quanto reproduzir sons. A Figura 19.2 mostra a estrutura básica de captura e reprodução de áudio em um computador pessoal, que é um bom exemplo de dispositivo de entrada/saída.

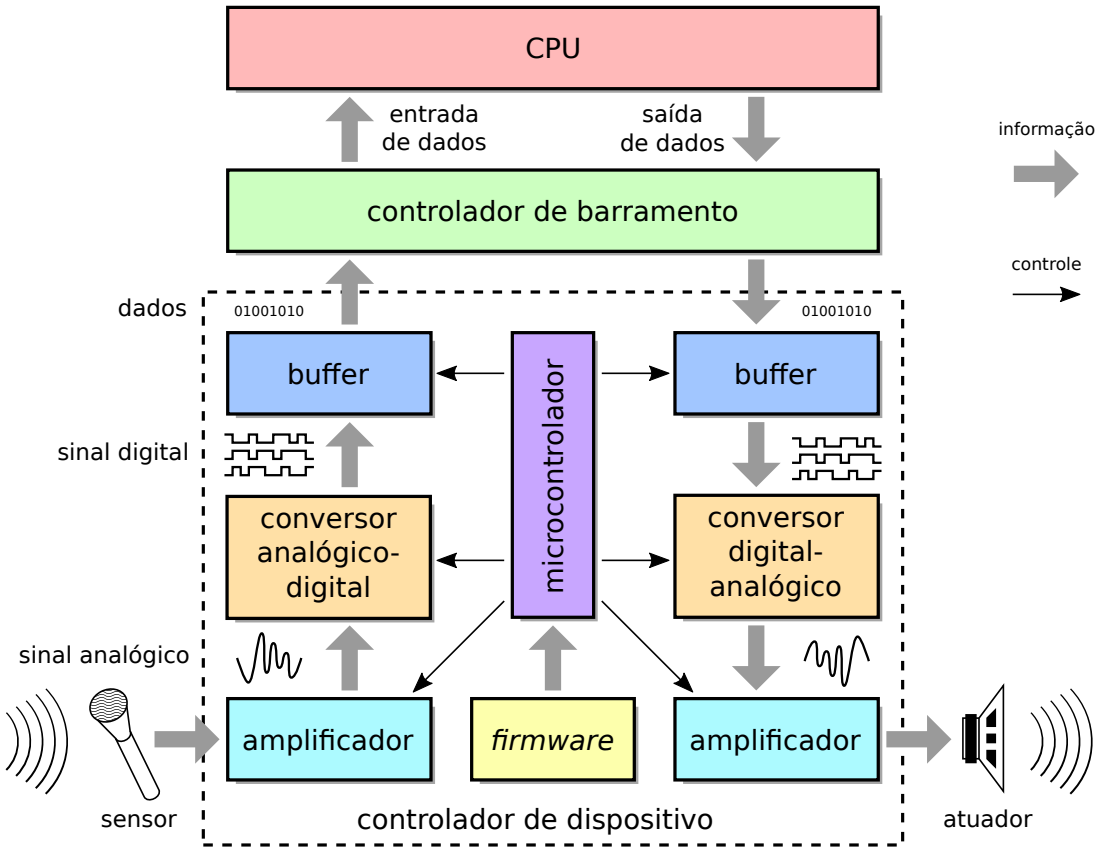


Figura 19.2: Estrutura básica da entrada e saída de áudio.

Os dispositivos de entrada/saída mais complexos, como discos rígidos, placas de rede e placas gráficas, possuem um processador ou microcontrolador interno para gerenciar sua operação. Esse processador embutido no dispositivo executa um código criado pelo fabricante do mesmo, denominado *firmware*. O código do *firmware* é independente do sistema operacional do computador e contém as instruções necessárias para operar o restante do hardware do dispositivo, permitindo realizar as operações solicitadas pelo sistema operacional.

[1]Sensores e atuadores são denominados genericamente *dispositivos transdutores*, pois transformam energia externa (como luz, calor, som ou movimento) em sinais elétricos, ou vice-versa.

19.3 Barramentos de acesso

Historicamente, o acoplamento dos dispositivos de entrada/saída ao computador é feito através de barramentos, seguindo o padrão estabelecido pela arquitetura de *Von Neumann*. Enquanto o barramento dos primeiros sistemas era um simples agrupamento de fios, os barramentos dos sistemas atuais são estruturas de hardware bastante complexas, com circuitos específicos para seu controle. Além disso, a diversidade de velocidades e volumes de dados suportados pelos dispositivos fez com que o barramento único dos primeiros sistemas fosse gradativamente estruturado em um conjunto de barramentos com características distintas de velocidade e largura de dados.

O controle dos barramentos em um sistema *desktop* moderno está a cargo de dois controladores de hardware que fazem parte do *chipset*[2] da placa-mãe: a *north bridge* e a *south bridge*. A *north bridge*, diretamente conectada ao processador, é responsável pelo acesso à memória RAM e aos dispositivos de alta velocidade, através de barramentos dedicados como AGP (*Accelerated Graphics Port*) e PCI-Express (*Peripheral Component Interconnect*).

Por outro lado, a *south bridge* é o controlador responsável pelos barramentos e portas de baixa ou média velocidade do computador, como as portas seriais e paralelas, e pelos barramentos dedicados como o PCI padrão, o USB e o SATA. Além disso, a *south bridge* costuma integrar outros componentes importantes do computador, como controladores de áudio e rede *on-board*, controlador de interrupções, controlador DMA (*Direct Memory Access*), temporizadores (responsáveis pelas interrupções de tempo usadas pelo escalonador de processos), relógio de tempo real (que mantém a informação de dia e hora atuais), controle de energia e acesso à memória BIOS. O processador se comunica com a *south bridge* indiretamente, através da *north bridge*.

A Figura 19.3 traz uma visão da arquitetura típica de um computador pessoal moderno. A estrutura detalhada e o funcionamento dos barramentos e seus respectivos controladores estão fora do escopo deste texto; informações mais detalhadas podem ser encontradas em [Patterson and Henessy, 2005].

Como existem muitas possibilidades de interação do computador com o mundo exterior, também existem muitos tipos de dispositivos de entrada/saída, com características diversas de velocidade de transferência, forma de transferência dos dados e método de acesso. A **velocidade de transferência de dados** de um dispositivo pode ir de alguns bytes por segundo, no caso de dispositivos simples como teclados e *mouses*, a gigabytes por segundo, para algumas placas de interface gráfica ou de acesso a discos de alto desempenho. A Tabela 19.1 traz alguns exemplos de dispositivos de entrada/saída com suas velocidades típicas de transferência de dados.

[2] O *chipset* de um computador é um conjunto de controladores e circuitos auxiliares de hardware integrados à placa-mãe, que proveem serviços fundamentais ao funcionamento do computador, como o controle dos barramentos, acesso à BIOS, controle de interrupções, temporizadores programáveis e controladores *on-board* para alguns periféricos, como discos rígidos, portas paralelas e seriais e entrada/saída de áudio.

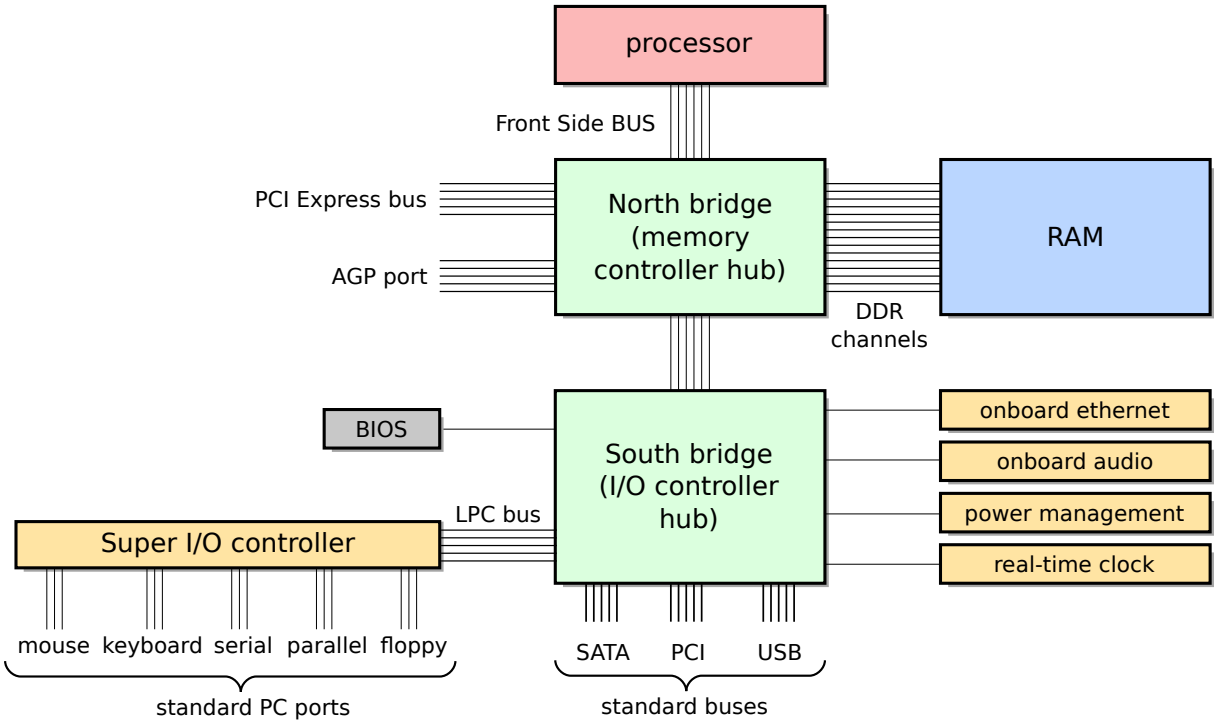


Figura 19.3: Arquitetura típica de um PC atual.

Tabela 19.1: Velocidades típicas de alguns dispositivos de entrada/saída.

Dispositivo	**velocidade**
Teclado	10 B/s
Mouse ótico	100 B/s
Interface infravermelho (IrDA-SIR)	14 KB/s
Interface paralela padrão	125 KB/s
Interface de áudio digital S/PDIF	384 KB/s
Interface de rede *Fast Ethernet*	11.6 MB/s
Chave ou disco USB 2.0	60 MB/s
Interface de rede *Gigabit Ethernet*	116 MB/s
Disco rígido SATA 2	300 MB/s
Interface gráfica *high-end*	4.2 GB/s

19.4 Interface de acesso

Para o sistema operacional, o aspecto mais relevante de um dispositivo de entrada/saída é sua interface de acesso, ou seja, a abordagem a ser usada para acessar o dispositivo, configurá-lo e enviar dados para ele (ou receber dados dele). Normalmente, cada dispositivo oferece um conjunto de registradores acessíveis através do barramento, também denominados **portas de entrada/saída**, que são usados para a comunicação entre o dispositivo e o processador. As portas oferecidas para acesso a cada dispositivo de entrada/saída podem ser divididas nos seguintes grupos (conforme ilustrado na Figura 19.4):

Portas de entrada (*data-in ports*): usadas pelo processador para receber dados provindos do dispositivo; são escritas pelo dispositivo e lidas pelo processador;

Portas de saída (*data-out ports*): usadas pelo processador para enviar dados ao dispositivo; essas portas são escritas pelo processador e lidas pelo dispositivo;

Portas de status (*status ports*): usadas pelo processador para consultar o estado interno do dispositivo ou verificar se uma operação solicitada ocorreu sem erro; essas portas são escritas pelo dispositivo e lidas pelo processador;

Portas de controle (*control ports*): usadas pelo processador para enviar comandos ao dispositivo ou modificar parâmetros de sua configuração; essas portas são escritas pelo processador e lidas pelo dispositivo.

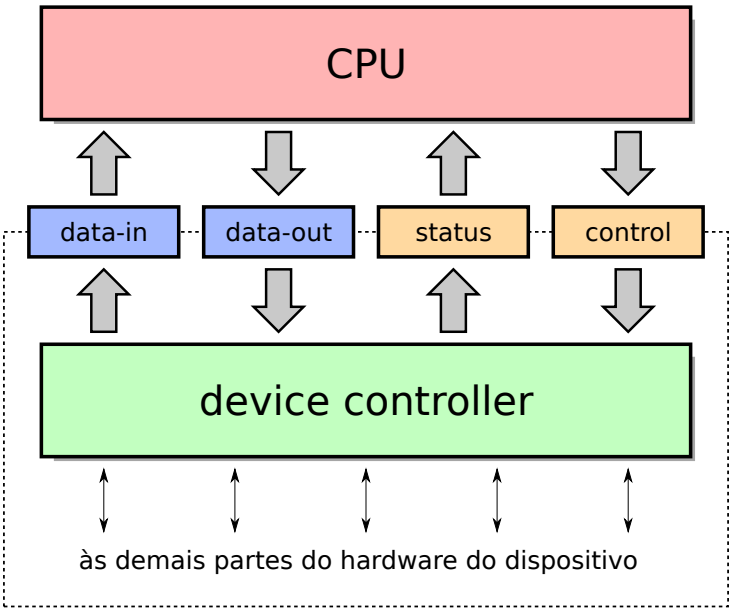


Figura 19.4: Portas de interface de um dispositivo de entrada/saída.

O número exato de portas e o significado específico de cada uma dependem do tipo de dispositivo considerado. Um exemplo simples de interface de acesso a dispositivo é a interface paralela, geralmente usada para acessar impressoras mais antigas. As portas de uma interface paralela operando no modo padrão (SPP - *Standard Parallel Port*) têm um byte cada e estão descritas a seguir [Patterson and Henessy, 2005]:

- P_0 (*data port*): porta de saída, usada para enviar dados à impressora; pode ser usada também como porta de entrada, se a interface estiver operando em modo bidirecional;
- P_1 (*status port*): porta de status, permite ao processador consultar vários indicadores de status da interface paralela ou do dispositivo ligado a ela. O significado de cada um de seus 8 bits é:
  0. reservado;
  1. reservado;
  2. $\overline{\texttt{nIRq}}$: se 0, indica que o controlador gerou uma interrupção (Seção 19.6);
  3. `error`: há um erro interno na impressora;
  4. `select`: a impressora está pronta (*online*);
  5. `paper_out`: falta papel na impressora;
  6. $\overline{\texttt{ack}}$: um pulso em 0 indica que o dado na porta P_0 foi recebido pelo controlador (pulso com duração $t \geq 1\mu s$);
  7. `busy`: indica que o controlador está ocupado processando um comando.
- P_2 (*control port*): porta de controle, usada para configurar a interface paralela e para solicitar operações de saída (ou entrada) de dados através da mesma. Seus 8 bits têm o seguinte significado:
  0. $\overline{\texttt{strobe}}$: um pulso em 0 informa o controlador que há um dado disponível em P_0 (pulso com duração $t \geq 0,5\mu s$);
  1. `auto_lf`: a impressora deve inserir um *line feed* a cada *carriage return* recebido;
  2. `reset`: a impressora deve ser reiniciada;
  3. `select`: a impressora está selecionada para uso;
  4. `enable_IRq`: permite ao controlador gerar interrupções (Seção 19.6);
  5. `bidirectional`: informa que a interface será usada para entrada e para saída de dados;
  6. reservado;
  7. reservado.
- P_3 a P_7: estas portas são usadas nos modos estendidos de operação da interface paralela, como EPP (*Enhanced Paralel Port*) e ECP (*Extended Capabilities Port*).

O algoritmo básico implementado pelo hardware interno do controlador da interface paralela para coordenar suas interações com o processador está ilustrado no fluxograma da Figura 19.5. Considera-se que os valores iniciais dos flags de status da porta P_1 são `nIRq=1`, `error=0`, `select=1`, `paper_out=0`, `ack=1` e `busy=0`.

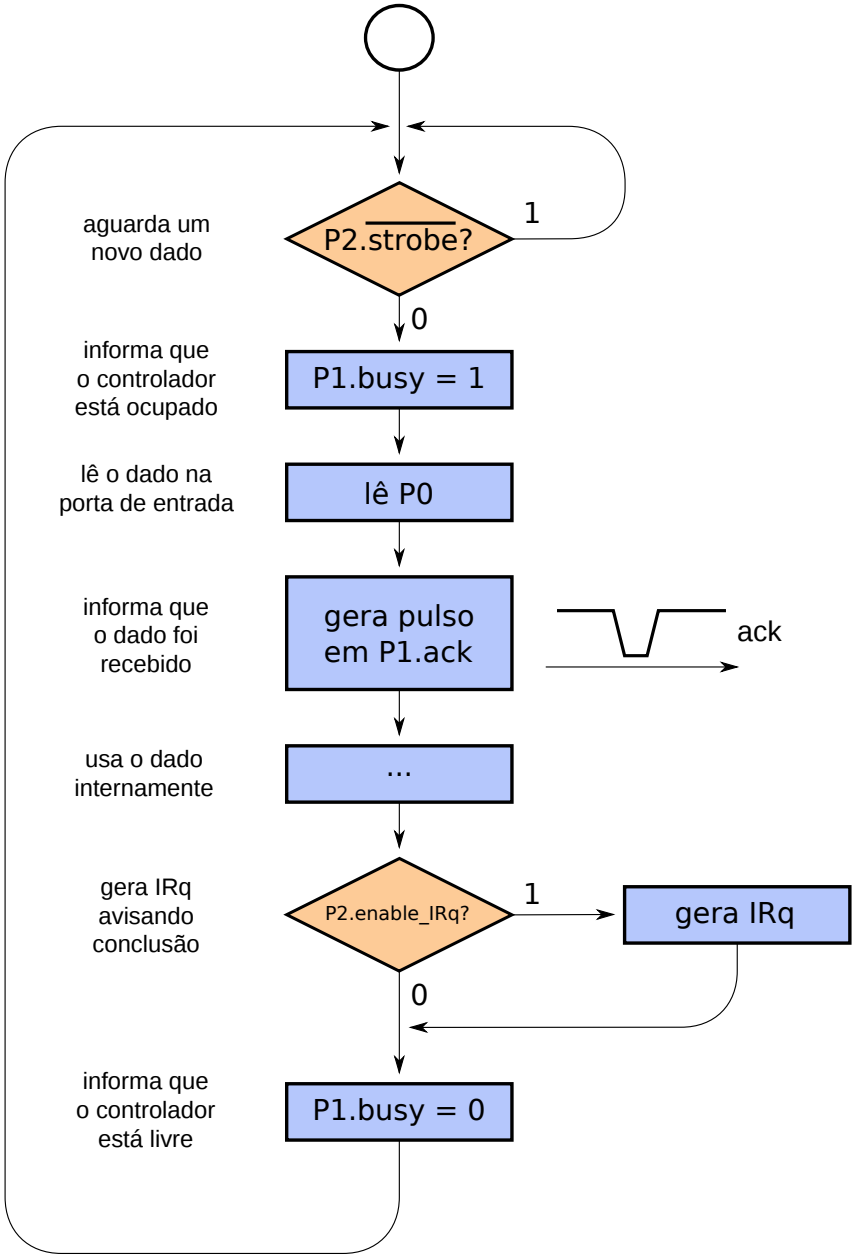


Figura 19.5: Comportamento básico do controlador da porta paralela.

19.5 Endereçamento de portas

A forma de acesso às portas que compõem a interface de um dispositivo varia de acordo com a arquitetura do computador. Alguns sistemas utilizam **entrada/saída mapeada em portas** (*port-mapped I/O*), onde as portas que compõem a interface são acessadas pelo processador através de instruções específicas para operações de entrada/saída. Por exemplo, os processadores da família Intel usam a instrução "`IN` *reg port*" para ler o valor presente na porta "*port*" do dispositivo e depositá-lo no registrador "*reg*" do processador, enquanto a instrução "`OUT` *port reg*" é usada para escrever na porta "*port*" o valor contido no registrador "*reg*".

A entrada/saída mapeada em portas usa um espaço de endereços de entrada/saída (*I/O address space*) independente da memória principal, normalmente compreendido

entre 0 e 64KB. Ou seja, o endereço de porta de E/S 001*Fh* e o endereço de memória 001*Fh* apontam para informações distintas. Para distinguir entre endereços de memória e endereços de portas de entrada/saída, o barramento de controle do processador possui uma linha $IO/\overline{M}$, que indica se o endereço presente no barramento de endereços se refere a uma posição de memória (se $IO/\overline{M} = 0$) ou a uma porta de entrada/saída (se $IO/\overline{M} = 1$). A Tabela 19.2 apresenta os endereços típicos de algumas portas de entrada/saída de dispositivos em computadores pessoais que seguem o padrão IBM-PC.

Dispositivo	**Endereços das portas**
teclado e mouse PS/2	0060h e 0064h
barramento IDE primário	0170h a 0177h
barramento IDE secundário	01F0h a 01F7h
relógio de tempo real	0070h e 0071h
porta serial COM1	02F8h a 02FFh
porta serial COM2	03F8h a 03FFh
porta paralela LPT1	0378h a 037Fh

Tabela 19.2: Endereços de portas de E/S de alguns dispositivos.

Uma outra forma de acesso aos dispositivos de entrada/saída é a **entrada/saída mapeada em memória** (*memory-mapped I/O*). Nesta abordagem, uma parte não ocupada do espaço de endereços de memória é reservado para mapear as portas de acesso aos dispositivos. Dessa forma, as portas são vistas como se fossem parte da memória principal e podem ser lidas e escritas através das mesmas instruções usadas para acessar o restante da memória, sem a necessidade de instruções especiais como `IN` e `OUT`. Algumas arquiteturas de computadores, como é caso do IBM-PC padrão, usam uma abordagem híbrida para certos dispositivos como interfaces de rede e de áudio: as portas de controle e status são mapeadas no espaço de endereços de entrada/saída, sendo acessadas através de instruções específicas, enquanto as portas de entrada e saída de dados são mapeadas em memória (normalmente na faixa de endereços entre 640 KB e 1MB) [Patterson and Henessy, 2005].

Finalmente, uma abordagem mais sofisticada para o controle de dispositivos de entrada/saída é o uso de um hardware independente, com processador dedicado, que comunica com o processador principal através de um barramento específico. Em sistemas de grande porte (*mainframes*) essa abordagem é denominada **canais de entrada/saída** (*IO channels*); em computadores pessoais, essa abordagem costuma ser usada em interfaces para vídeo ou áudio de alto desempenho, como é o caso das placas gráficas com aceleração, nas quais um processador gráfico (GPU – *Graphics Processing Unit*) realiza a parte mais pesada do processamento da saída de vídeo, como a renderização de imagens em 3 dimensões e texturas, deixando o processador principal livre para outras tarefas.

19.6 Interrupções

O acesso aos controladores de dispositivos através de suas portas é conveniente para a comunicação no sentido *processador* → *controlador*, ou seja, para as interações iniciadas pelo processador. Entretanto, essa forma de acesso é inviável para interações no iniciadas pelo controlador, pois o processador pode demorar a acessar suas portas, caso esteja ocupado em outras atividades.

Frequentemente um controlador de dispositivo precisa informar ao processador com rapidez sobre um evento interno, como a chegada de um pacote de rede, um clique de mouse ou a conclusão de uma operação de disco. Nesse caso, o controlador pode notificar o processador sobre o evento ocorrido através de uma **requisição de interrupção** (IRq - *Interrupt Request*).

As requisições de interrupção são sinais elétricos veiculados através do barramento de controle do computador. Cada interrupção está geralmente associada a um número inteiro que permite identificar sua origem. A Tabela 19.3 informa os números de interrupção associados a alguns dispositivos periféricos típicos em um PC no padrão *Intel x86*.

Dispositivo	**Interrupção**
teclado	1
porta serial COM2	3
porta serial COM1	4
porta paralela LPT1	7
relógio de tempo real	8
mouse PS/2	12
barramento ATA primário	14
barramento ATA secundário	15

Tabela 19.3: Interrupções geradas por alguns dispositivos em um PC.

Ao receber uma determinada requisição de interrupção, o processador suspende seu fluxo de instruções corrente e desvia a execução para um endereço pré-definido, onde se encontra uma **rotina de tratamento de interrupção** (*interrupt handler*). Essa rotina é responsável por tratar aquela requisição de interrupção, ou seja, executar as ações necessárias para acessar o controlador de dispositivo e tratar o evento que a gerou. Ao final da rotina de tratamento da interrupção, o processador retoma o código que estava executando quando foi interrompido. A Figura 19.6 representa os principais passos associados ao tratamento de uma interrupção envolvendo o controlador de teclado, detalhados a seguir:

1. O processador está executando um programa qualquer;
2. O usuário pressiona uma tecla no teclado;
3. O controlador do teclado identifica a tecla pressionada, armazena seu código em um *buffer* interno e envia uma requisição de interrupção (IRq) ao processador;

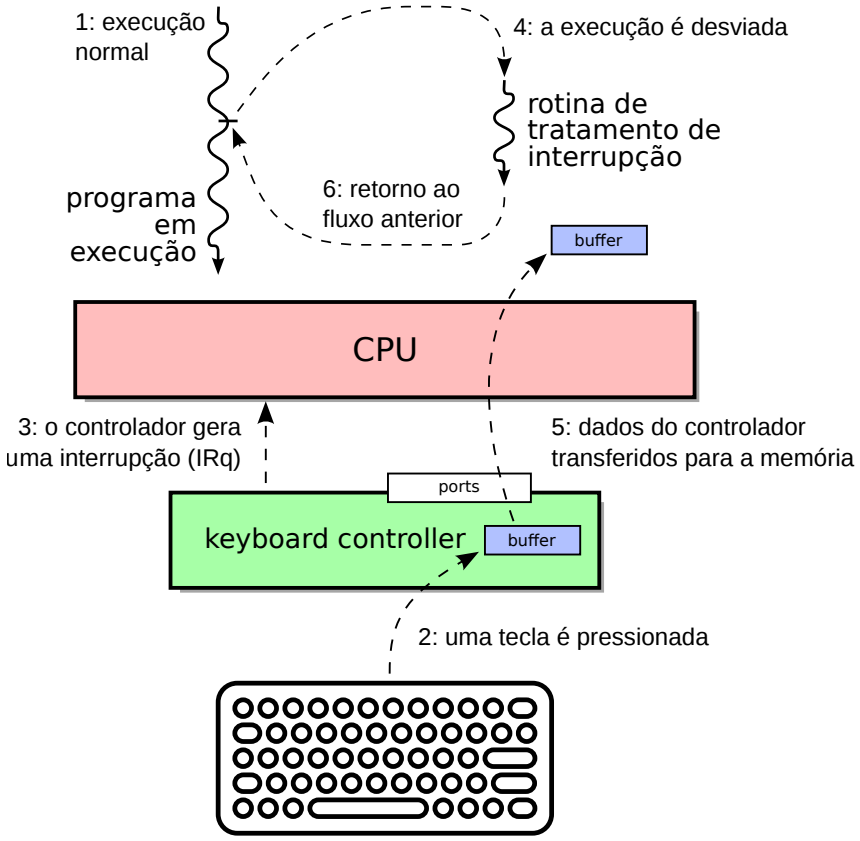


Figura 19.6: Roteiro típico de um tratamento de interrupção

4. O processador recebe a interrupção, salva na pilha seu estado atual (o conteúdo de seus registradores) e desvia sua execução para a rotina de tratamento da interrupção associada ao teclado;

5. Ao executar, essa rotina acessa as portas do controlador de teclado para transferir o conteúdo de seu *buffer* para uma área de memória do núcleo. Depois disso, ela pode executar outras ações, como acordar algum processo ou *thread* que esteja esperando por entradas do teclado;

6. Ao concluir a execução da rotina de tratamento da interrupção, o processador retorna à execução do fluxo de instruções que havia sido interrompido, usando a informação de estado salva no passo 4.

Essa sequência de ações ocorre a cada requisição de interrupção recebida pelo processador. Como cada interrupção corresponde a um evento ocorrido em um dispositivo periférico (chegada de um pacote de rede, movimento do mouse, conclusão de operação do disco, etc.), podem ocorrer centenas ou mesmo milhares de interrupções por segundo, dependendo da carga de trabalho e da configuração do sistema (número e tipo de periféricos). Por isso, as rotinas de tratamento de interrupção devem realizar suas tarefas rapidamente, para não prejudicar o funcionamento do restante do sistema.

Como cada tipo de interrupção pode exigir um tipo de tratamento diferente (pois os dispositivos são diferentes), cada requisição de interrupção deve disparar uma rotina de tratamento específica. A maioria das arquiteturas de processador atuais define uma tabela de endereços de funções denominada *Tabela de Interrupções* (IVT - *Interrupt*

Vector Table); cada entrada dessa tabela contém o endereço da rotina de tratamento da interrupção correspondente. Por exemplo, se a entrada 5 da tabela contém o valor `3C20h`, então a rotina de tratamento da IRq 5 iniciará na posição `3C20h` da memória RAM. Dependendo do hardware, a tabela de interrupções pode residir em uma posição fixa da memória RAM, definida pelo fabricante do processador, ou ter sua posição indicada pelo conteúdo de um registrador da CPU específico para esse fim.

Além das requisições de interrupção geradas pelos controladores de dispositivos, eventos internos ao processador também podem ocasionar o desvio da execução usando o mesmo mecanismo de interrupção: são as **exceções**. Eventos como instruções inválidas, divisão por zero ou outros erros de software disparam exceções internas no processador, que resultam na ativação de rotinas de tratamento de exceção registradas na tabela de interrupções. A Tabela 19.4 apresenta algumas exceções previstas pelo processador Intel Pentium (extraída de [Patterson and Henessy, 2005]).

Tabela 19.4: Algumas exceções dos processadores *Intel x86*.

Exceção	**Descrição**
0	erro de divisão por zero
3	breakpoint (parada de depurador)
5	erro de faixa de valores
6	operação inválida
7	dispositivo não disponível
11	segmento de memória ausente
12	erro de pilha
14	falta de página
16	erro de ponto flutuante
19-31	valores reservados pela Intel

Nas arquiteturas de hardware atuais, as interrupções geradas pelos dispositivos de entrada/saída não são transmitidas diretamente ao processador, mas a um **controlador de interrupções programável** (PIC - *Programmable Interrupt Controller*, ou APIC - *Advanced Programmable Interrupt Controller*), que faz parte do *chipset* do computador. As linhas de interrupção dos controladores de periféricos são conectadas aos pinos desse controlador de interrupções, enquanto suas saídas são conectadas às entradas de interrupção do processador.

O controlador de interrupções recebe as interrupções dos dispositivos e as encaminha ao processador em sequência, uma a uma. Ao receber uma interrupção, o processador deve acessar a interface do PIC para identificar a origem da interrupção e depois "reconhecê-la", ou seja, indicar ao PIC que aquela interrupção foi tratada e pode ser descartada pelo controlador. Interrupções já ocorridas mas ainda não reconhecidas pelo processador são chamadas de **interrupções pendentes**.

O PIC pode ser programado para bloquear ou ignorar algumas das interrupções recebidas, impedindo que cheguem ao processador. Além disso, ele permite definir prioridades entre as interrupções. A Figura 19.7 mostra a operação básica de um

controlador de interrupções; essa representação é simplificada, pois as arquiteturas de computadores mais recentes podem contar com vários controladores de interrupções interligados.

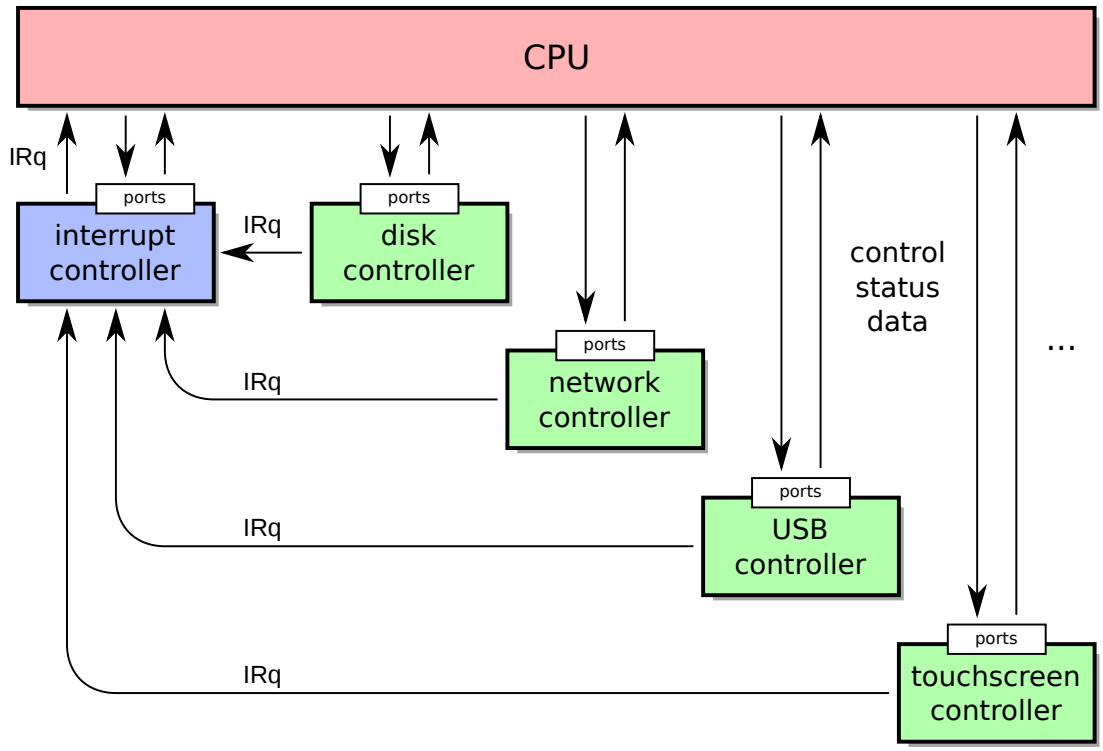


Figura 19.7: Uso de um controlador de interrupções.

O mecanismo de interrupção torna eficiente a interação do processador com os dispositivos periféricos. Se não existissem interrupções, o processador perderia muito tempo consultando todos os dispositivos do sistema para verificar se há eventos a serem tratados. Além disso, as interrupções permitem construir funções de entrada/saída assíncronas: o processador não precisa esperar a conclusão de cada operação solicitada, pois o dispositivo emitirá uma interrupção para "avisar" o processador quando a operação estiver concluída.

Referências

D. Patterson and J. Henessy. *Organização e Projeto de Computadores*. Campus, 2005.

Capítulo 20

Software de entrada/saída

20.1 Introdução

O sistema operacional é responsável por oferecer acesso aos dispositivos de entrada/saída às aplicações e, em consequência, aos usuários do sistema. Prover acesso eficiente, rápido e confiável a um conjunto de periféricos com características diversas de comportamento, velocidade de transferência, volume de dados produzidos/consumidos e diferentes interfaces de hardware é um enorme desafio. Além disso, como cada dispositivo define sua própria interface e modo de operação, o núcleo do sistema operacional deve implementar o código necessário para interagir com milhares de tipos de dispositivos distintos. Como exemplo, cerca de 70% das 20 milhões de linhas de código do núcleo Linux na versão 4.3 pertencem a código de *drivers* de dispositivos de entrada/saída.

Este capítulo apresenta uma visão geral da organização e do funcionamento dos componentes do sistema operacional responsáveis por interagir com os dispositivos de hardware de entrada/saída, e como o acesso a esses dispositivos é provido às aplicações em modo usuário.

20.2 Arquitetura de software de entrada/saída

Para simplificar o uso e a gerência dos dispositivos de entrada/saída, o código do sistema operacional é estruturado em camadas, que levam da interação direta com o hardware (como o acesso às portas de entrada/saída, interrupções e operações de DMA) às interfaces de acesso abstratas e genéricas oferecidas às aplicações, como arquivos e *sockets* de rede.

Uma visão conceitual dessa estrutura em camadas pode ser vista na Figura 20.1. Nessa figura, a camada inferior corresponde aos dispositivos físicos propriamente ditos, como discos rígidos, teclados, etc. A camada logo acima corresponde aos controladores de dispositivos (discos SATA, USB, etc.) e aos controladores de DMA e de interrupções implementados no *chipset* do computador.

A primeira camada de software no núcleo do sistema operacional corresponde aos *drivers* de dispositivos, ou simplesmente *drivers*[1], que são os componentes de código que interagem diretamente com cada controlador, para realizar as operações

[1] Em Portugal se utiliza o termo "piloto", que eu aprecio, mas optei por não utilizá-lo neste texto porque o termo em inglês é omnipresente no Brasil.

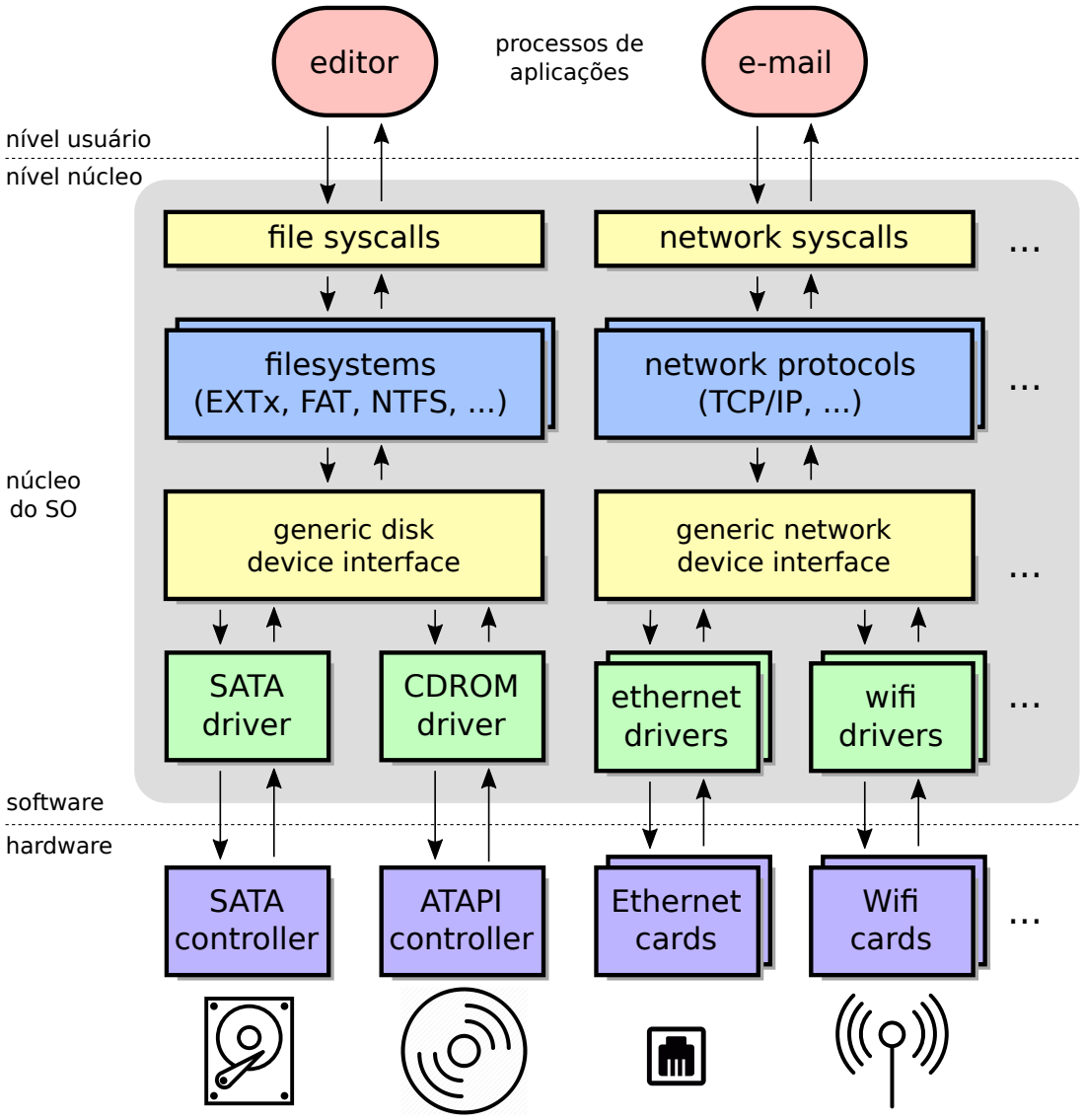


Figura 20.1: Estrutura em camadas do software de entrada/saída.

de entrada/saída, receber as requisições de interrupção e fazer o gerenciamento do dispositivo correspondente. Dentro de um mesmo grupo de dispositivos similares, como as placas de rede, há centenas de modelos de diferentes fabricantes, com interfaces distintas. Para cada dispositivo é então necessário construir um *driver* específico.

Acima dos *drivers* existe uma camada de código, denominada *generic device interface*, cuja finalidade é construir uma visão genérica de dispositivos similares, para que o restante do sistema operacional não precise ter consciência das peculiaridades de cada dispositivo, mas possa tratá-los por famílias ou classes, como dispositivos de armazenamento, interfaces de rede, de vídeo, etc.

Acima da camada de interface genérica de dispositivos, uma ou mais camadas de código estão presentes, para implementar abstrações mais complexas, como sistemas de arquivos e protocolos de rede. Finalmente, no topo da arquitetura de software, são implementadas as chamadas de sistema fornecidas às aplicações para acessar as abstrações construídas pelas camadas inferiores, como arquivos, diretórios e *sockets* de rede, etc.

20.3 Classes de dispositivos

Para simplificar a construção de aplicações e das camadas mais elevadas do próprio sistema operacional, os dispositivos de entrada/saída são geralmente agrupados em classes ou famílias com características similares, para os quais uma interface genérica pode ser definida. Por exemplo, discos rígidos SATA, discos SSD e DVD-ROMs têm características mecânicas e elétricas distintas, mas servem basicamente para o mesmo propósito: armazenar arquivos. O mesmo pode ser afirmado sobre interfaces de rede *Ethernet* e *Wifi*: embora usem tecnologias distintas, ambas permitem a comunicação entre computadores.

Nos sistemas de padrão UNIX os dispositivos são geralmente agrupados em quatro grandes famílias[2]:

Dispositivos orientados a caracteres: são aqueles cujas transferências de dados são sempre feitas byte por byte, em sequência. Um dispositivo orientado a caracteres pode ser visto como um fluxo contínuo de entrada ou de saída de bytes. A característica sequencial faz com que não seja possível alterar o valor de um byte que já foi enviado. Dispositivos ligados às interfaces paralelas e seriais do computador, como *mouse* e teclado, são os exemplos mais clássicos desta família. Os terminais de texto e *modems* de transmissão de dados por linhas seriais (como as linhas telefônicas) também são considerados dispositivos orientados a caracteres.

Dispositivos orientados a blocos: são aqueles dispositivos em que as operações de entrada ou saída de dados são feitas usando blocos de bytes de tamanho fixo. Esses blocos são lidos ou escritos em posições específicas do dispositivo, ou seja, são endereçáveis. Conceitualmente, um dispositivo orientado a blocos pode ser visto como um vetor de blocos de bytes de mesmo tamanho. Discos rígidos, CDROMs, fitas magnéticas e outros dispositivos de armazenamento são exemplos típicos desta família.

Dispositivos de rede: estes dispositivos permitem enviar e receber mensagens entre processos e computadores distintos. As mensagens são blocos de dados de tamanho variável, com envio e recepção feitas de forma sequencial (não é possível alterar o conteúdo de uma mensagem que já foi enviada). As interfaces *Ethernet*, *Wifi*, *Bluetooth* e GPRS são bons exemplos desta classe de dispositivos.

Dispositivos gráficos: permitem a renderização de texto e gráficos em terminais de vídeo. Devido aos requisitos de desempenho, sobretudo para jogos e filmes, estes dispositivos exigem um alto desempenho na transferência de dados. Por isso, sua interface genérica é constituída por funções para consultar e configurar o dispositivo gráfico e uma área de memória compartilhada entre o processador e o dispositivo, usualmente denominada *frame buffer*, que permite acesso direto à memória de vídeo. Programas ou bibliotecas que interagem diretamente com o dispositivo gráfico têm acesso a essa área de memória através de bibliotecas específicas, como *DirectX* em ambientes Windows ou *DRI – Direct Rendering Engine* no Linux.

[2]Nos sistemas *Windows* os dispositivos são agrupados em um número maior de categorias.

Vários dispositivos não se enquadram diretamente nas categorias acima, como receptores de GPS, sensores de temperatura e interfaces de áudio. Nestes casos, alguns sistemas operacionais optam por criar classes adicionais para esses dispositivos (como o Windows), enquanto outros buscam enquadrá-los em uma das famílias já existentes (como os UNIX). No Linux, por exemplo, os dispositivos de áudio são acessados pelas aplicações como dispositivos orientados a caracteres: uma sequência de bytes enviada ao dispositivo é tratada como um fluxo de áudio a ser reproduzido, geralmente em formato PCM. Cabe ao driver do dispositivo e à camada de interface genérica transformar essa interface orientada a caracteres nas operações de baixo nível necessárias para reproduzir o fluxo de áudio desejado.

20.4 *Drivers* de dispositivos

Um *driver* de dispositivo, ou simplesmente *driver*, é um componente do sistema operacional responsável por interagir com um controlador de dispositivo. Cada tipo de dispositivo possui seu próprio *driver*, muitas vezes fornecido pelo fabricante do mesmo. Cada *driver* é geralmente capaz de tratar um único tipo de dispositivo, ou uma família de dispositivos correlatos do mesmo fabricante. Por exemplo, o *driver* RTL8110SC(L), da empresa *Realtek Corp.*, serve somente para as interfaces de rede RTL8110S, RTL8110SB(L), RTL8169SB(L), RTL8169S(L) e RTL8169 desse fabricante.

Internamente, um *driver* consiste de um conjunto de funções que são ativadas pelo núcleo do sistema operacional conforme necessário. Existem basicamente três grupos de funções implementadas por um *driver*, ilustradas na Figura 20.2:

Funções de entrada/saída: responsáveis pela transferência de dados entre o dispositivo e o sistema operacional; essas funções recebem e enviam dados de acordo com a classe dos dispositivo: caracteres (bytes), blocos de tamanho fixo (discos), blocos de tamanho variável (pacotes de rede) ou áreas de memória compartilhadas entre o dispositivo e a CPU (imagens/vídeo e outros).

Funções de gerência: responsáveis pela gestão do dispositivo e do próprio *driver*. Além de funções para coordenar a inicialização e finalização do *driver* e do dispositivo, geralmente são fornecidas funções para configurar o dispositivo, para desligar ou colocar em espera o dispositivo quando este não for usado, e para tratar erros no *dispositivo*. Algumas dessas funções podem ser disponibilizadas aos processos no espaço de usuário, através de chamadas de sistema específicas como `ioctl` (UNIX) e `DeviceIoControl` (Windows).

Funções de tratamento de eventos: estas funções são ativadas quando uma requisição de interrupção é gerada pelo dispositivo. Conforme apresentado na Seção 19.6, toda requisição de interrupção gerada pelo dispositivo é encaminhada ao controlador de interrupções do hardware, que a entrega ao núcleo do sistema operacional. No núcleo, um tratador de interrupções (*IRq handler*) reconhece e identifica a interrupção junto ao controlador e em seguida envia uma notificação de evento a uma função do *driver*, para o devido tratamento.

Além das funções acima descritas, um *driver* mantém estruturas de dados locais, para armazenar informações sobre o dispositivo e as operações em andamento.

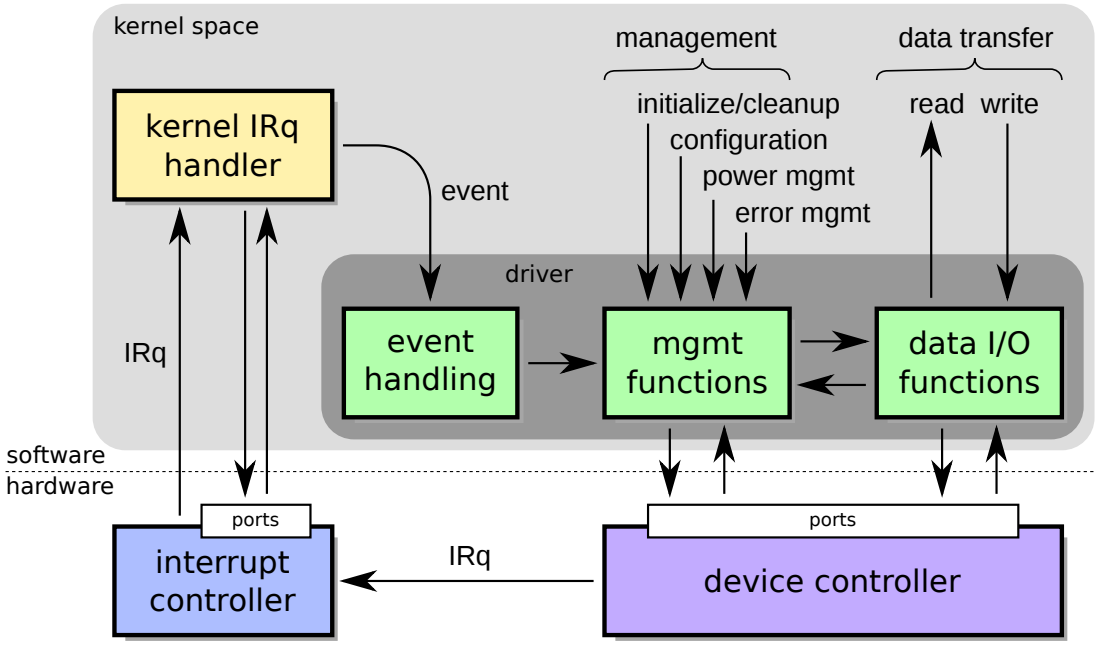


Figura 20.2: Visão geral de um *driver* de dispositivo.

Os *drivers* normalmente executam dentro do núcleo do sistema operacional, em modo privilegiado. Por ser código de terceiros executando com acesso total ao hardware, eles constituem um dos maiores riscos à estabilidade e segurança do sistema operacional. *Drivers* mal construídos ou mal configurados são fontes frequentes de problemas como travamentos ou reinicializações inesperadas.

20.5 Estratégias de interação

Cada *driver* deve interagir com seu respectivos dispositivo de entrada/saída para realizar as operações desejadas, através das portas de seu controlador. Esta seção aborda as três estratégias mais frequentemente usadas pelos *drivers* para essa interação, que são a entrada/saída controlada por programa, a controlada por eventos e o acesso direto à memória, detalhados a seguir.

20.5.1 Interação controlada por programa

A estratégia de entrada/saída mais simples, usada com alguns tipos de dispositivos, é a interação controlada por programa, também chamada **varredura**, **polling** ou PIO – *Programmed I/O*. Nesta abordagem, o *driver* solicita uma operação ao controlador do dispositivo, usando as portas *control* e *data-out* (ou *data-in*) de sua interface, e aguarda a conclusão da operação solicitada, monitorando continuamente os bits da respectiva porta de status.

Considerando as portas da interface paralela descrita na Seção 19.4, o comportamento do *driver* em uma operação de saída na porta paralela usando essa abordagem é descrito (simplificadamente) pelos trechos de código a seguir. O primeiro trecho de código contém definições de macros C úteis para o restante do código:

```
// portas da interface paralela LPT1 (endereço inicial em 0378h)
#define P0    0x0378        # porta de dados
#define P1    0x0379        # porta de status
#define P2    0x037A        # porta de controle

// bits de controle e status das portas
#define BUSY  7             # bit 7 da porta de status
#define ACK   6             # bit 6 da porta de status
#define STROBE 0            # bit 0 da porta de controle

// operações em bits individuais de bytes
#define BIT_SET(a,n) (a |= 1 << n)       // muda n-esimo bit de a para 1
#define BIT_CLR(a,n) (a &= 1 << n)       // muda n-esimo bit de a para 0
#define BIT_TST(a,n) (a & 1 << n)        // testa n-esimo bit de a
```

O código a seguir contém a implementação simplificada de uma operação de saída de caractere. As leituras e escritas de bytes nas portas do dispositivo são representadas respectivamente pelas funções `in(port)` e `out(port,value)`:

```
// saída de dados por programa
void polling_output (char c)
{
  // espera o controlador ficar livre, testando a porta de status (P1)
  while (BIT_TST (in(P1), BUSY) ;

  // escreve o byte "c" a enviar na porta de dados (P0)
  out (P0, c) ;

  // gera pulso em 0 no bit STROBE da porta de controle (P2),
  // para indicar ao controlador que há um novo dado em P0
  out (P2, BIT_CLR (in (P2), STROBE) ;    // poe bit STROBE de P2 em 0
  usleep (1) ;                            // aguarda 1 us
  out (P2, BIT_SET (in (P2), STROBE) ;    // poe bit STROBE de P2 em 0

  // espera controlador receber o dado (pulso em 0 no bit ACK de P1)[3]
  while (BIT_TST (in (P1), ACK)) ;

  // espera o controlador concluir a operação solicitada
  while (BIT_TST (in(P1), BUSY) ;
}
```

Observa-se que o processador e o controlador executam ações coordenadas e complementares: o processador espera que o controlador esteja livre antes de lhe enviar um dado; por sua vez, o controlador espera que o processador lhe envie um novo dado para processar, e assim por diante. Essa interação é ilustrada na Figura 20.3.

Nessa estratégia, o controlador pode ficar esperando por novos dados, pois só precisa trabalhar quando há dados a processar, ou seja, quando o bit *strobe* de sua porta P_2 indicar que há um novo dado em sua porta P_0. Entretanto, manter o processador esperando até que a operação seja concluída é indesejável, sobretudo se a operação solicitada for demorada. Além de constituir uma situação clássica de desperdício de recursos por **espera ocupada**, manter o processador esperando pela resposta de um

[3] Este laço de espera sobre o bit `ACK` é desnecessário, pois o próximo laço testa o bit `BUSY`, que indica a conclusão da operação solicitada. Contudo, ele foi mantido no código para maior clareza didática.

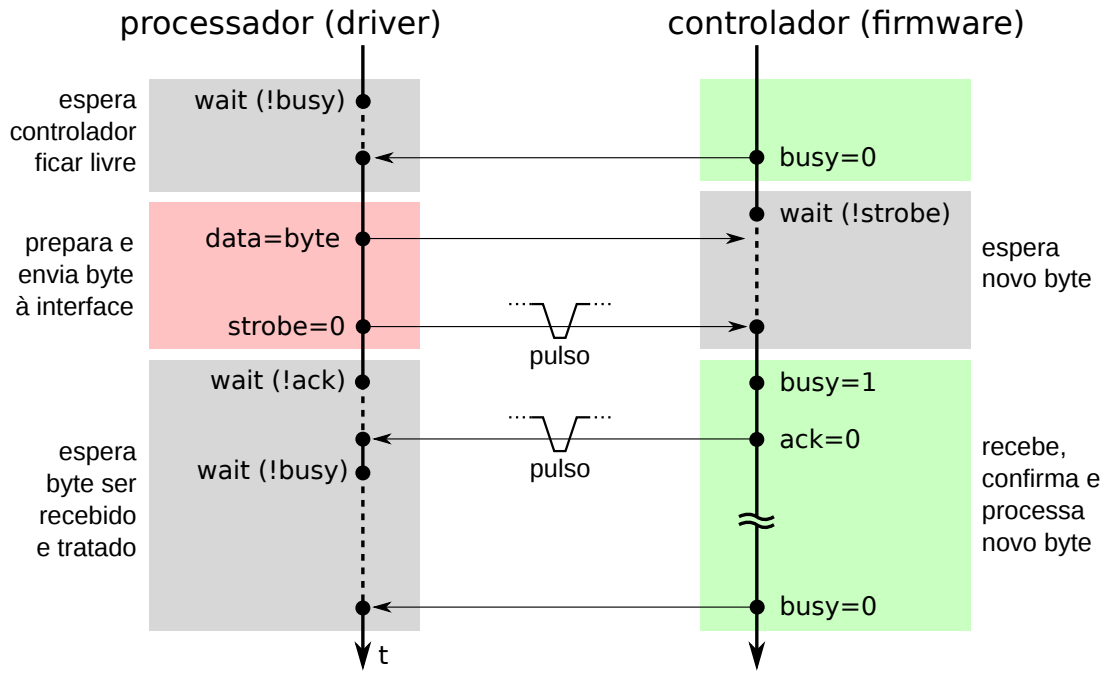


Figura 20.3: Entrada/saída controlada por programa.

controlador lento pode prejudicar o andamento de outras atividades importantes do sistema, como a interação com o usuário.

O problema da espera ocupada torna a estratégia de entrada/saída por programa pouco eficiente, sobretudo se o dispositivo for lento. Por isso, ela é pouco usada em sistemas operacionais de propósito geral. Seu uso se concentra sobretudo em sistemas embarcados dedicados, nos quais o processador só tem uma atividade (ou poucas) a realizar. A estratégia básica de varredura pode ser modificada, substituindo o teste contínuo do status do dispositivo por um teste periódico (por exemplo, a cada 1*ms*), e permitindo ao processador executar outras tarefas enquanto o dispositivo estiver ocupado. Todavia, essa abordagem implica em uma menor taxa de transferência de dados para o dispositivo e, por essa razão, só é usada em dispositivos com baixa vazão de dados.

20.5.2 Interação controlada por eventos

Uma forma mais eficiente de interagir com dispositivos de entrada/saída consiste em efetuar a requisição da operação desejada e suspender o fluxo de execução corrente, liberando o processador para tratar outras tarefas. Quando o dispositivo tiver terminado de processar a operação solicitada, seu controlador irá gerar uma requisição de interrupção (IRq) para notificar o respectivo *driver*, que poderá então retomar a execução daquele fluxo de instruções. Essa estratégia de ação é denominada **interação controlada por eventos** ou por interrupções, pois as interrupções têm um papel fundamental em sua implementação.

Na estratégia de entrada/saída por eventos, uma operação de entrada ou saída é dividida em dois blocos de instruções: um bloco que inicia a operação, ativado pelo *driver* a pedido de um processo ou *thread*, e uma **rotina de tratamento de interrupção** (*interrupt handler*), ativada a cada interrupção, para informar sobre a conclusão da última operação solicitada.

Considerando novamente a interface paralela descrita na Seção 19.4, o código a seguir representa o lançamento da operação de E/S pelo *driver* e a rotina de tratamento de interrupção (subentende-se as constantes e variáveis definidas na listagem anterior). Conforme visto na Seção 19.4, o controlador da interface paralela pode ser configurado para gerar uma interrupção através do flag *Enable_IRq* de sua porta de controle (porta P_2).

```
1  // saída de dados por evento: solicitação de operação
2  void event_output (char c)
3  {
4    // espera o controlador ficar livre, testando a porta de status (P1)
5    while (BIT_TST (in(P1), BUSY) ;
6
7    // escreve o byte "c" a enviar na porta de dados (P0)
8    out (P0, c) ;
9
10   // gera pulso em 0 no bit STROBE da porta de controle (P2),
11   // para indicar ao controlador que há um novo dado em P0
12   out (P2, BIT_CLR (in (P2), STROBE) ;    // poe bit STROBE de P2 em 0
13   usleep (1) ;                            // aguarda 1 us
14   out (P2, BIT_SET (in (P2), STROBE) ;    // poe bit STROBE de P2 em 0
15
16   // espera controlador receber o dado (pulso em 0 no bit ACK de P1)
17   while (BIT_TST (in (P1), ACK)) ;
18
19   // suspende a execução, liberando o processador para outras tarefas
20   // enquanto o controlador está ocupado processando o dado recebido.
21   suspend_task () ;
22 }
23
24 // saída de dados por evento: tratamento da interrupção
25 void event_handle ()
26 {
27   // o controlador concluiu sua operação, acordar a tarefa solicitante.
28   awake_task () ;
29 }
```

Nesse exemplo, percebe-se que o *driver* inicia a transferência de dados para a interface paralela e suspende a tarefa solicitante (chamada *suspend_task*), liberando o processador para outras atividades. Ao ocorrer uma interrupção, a rotina de tratamento do *driver* é ativada pelo SO e acorda a tarefa solicitante. O diagrama da Figura 20.4 ilustra de forma simplificada a estratégia de entrada/saída usando interrupções.

Uma variante mais eficiente da operação por eventos consistiria em fornecer ao *driver* um *buffer* com N bytes a enviar, como mostra o código a seguir:

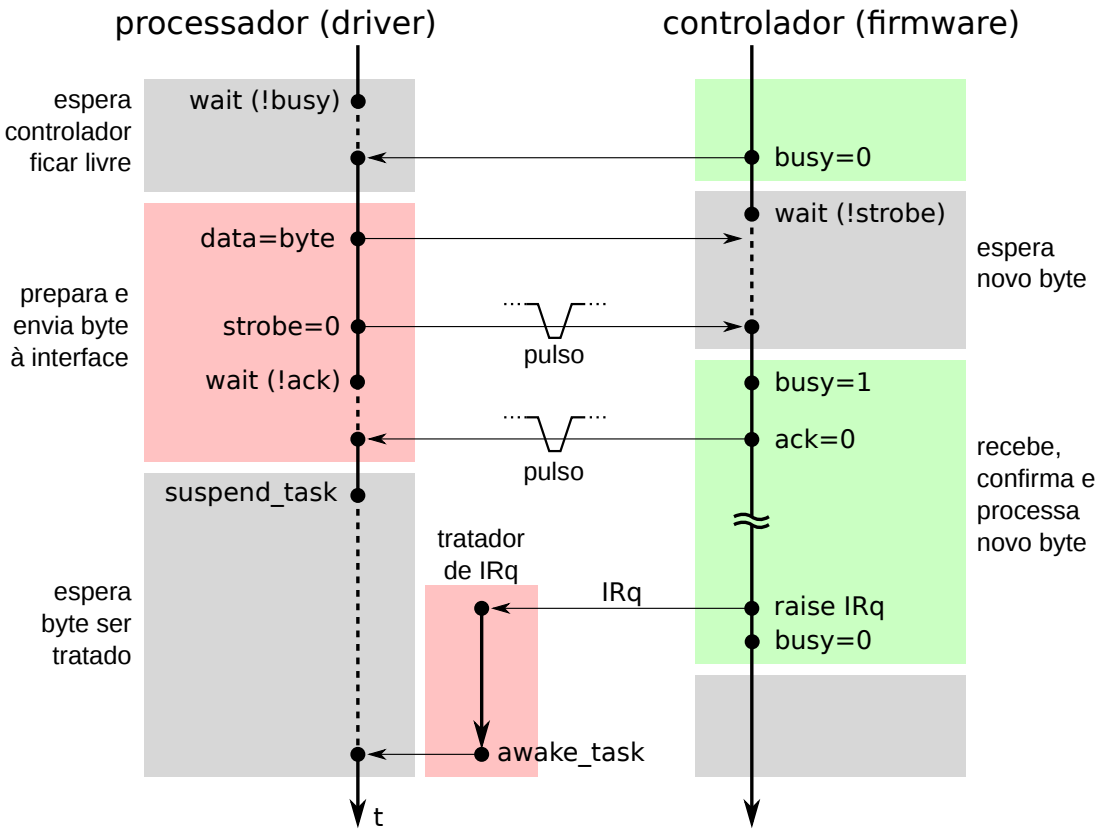


Figura 20.4: Entrada/saída controlada por eventos (interrupções).

```
1  // buffer de bytes a enviar
2  char *buffer ;
3  int bufsize, pos ;
4
5  // envia um byte à porta paralela
6  void send_byte (char c)
7  {
8    // espera o controlador ficar livre, testando a porta de status (P1)
9    while (BIT_TST (in(P1), BUSY) ;
10
11   // escreve o byte a enviar na porta de dados (P0)
12   out (P0, c) ;
13
14   // gera pulso em 0 no bit STROBE da porta de controle (P2),
15   // para indicar ao controlador que há um novo dado em P0
16   out (P2, BIT_CLR (in (P2), STROBE) ;    // poe bit STROBE de P2 em 0
17   usleep (1) ;                            // aguarda 1 us
18   out (P2, BIT_SET (in (P2), STROBE) ;    // poe bit STROBE de P2 em 0
19
20   // espera controlador receber o dado (pulso em 0 no bit ACK de P1)
21   while (BIT_TST (in (P1), ACK)) ;
22 }
```

```
23 // lançamento da operação de saída de dados
24 void event_output ()
25 {
26   // envia o primeiro byte do buffer
27   pos = 0 ;
28   send_byte (buffer[pos]) ;
29
30   // suspende a execução, liberando o processador para outras tarefas
31   // enquanto o controlador está ocupado processando o dado recebido.
32   suspend_task () ;
33 }
34
35 // rotina de tratamento de interrupções da interface paralela
36 void event_handle ()
37 {
38   pos++ ;                          // avança posição de envio no buffer
39   if (pos >= bufsize)              // o buffer terminou?
40     awake_task () ;                // sim, acorda a tarefa solicitante
41   else
42     send_byte (buffer[pos]) ;      // não, envio o próximo byte
43 }
```

20.5.3 Tratamento de interrupções

Durante a execução de uma rotina de tratamento de interrupção, é usual inibir novas interrupções, para evitar a execução aninhada de tratadores de interrupção, o que tornaria o código dos *drivers* (e do núcleo) bem mais complexo e suscetível a erros. Entretanto, manter interrupções inibidas durante muito tempo pode ocasionar perdas de dados ou outros problemas. Por exemplo, uma interface de rede gera uma interrupção quando recebe um pacote vindo da rede; esse pacote fica em seu buffer interno e deve ser transferido dali para a memória principal antes que outros pacotes cheguem, pois esse buffer tem uma capacidade limitada. Por essa razão, o tratamento das interrupções deve ser feito de forma muito rápida.

Para obter rapidez, a maioria dos sistemas operacionais implementa o tratamento de interrupções em dois níveis: um **tratador primário** (FLIH - *First-Level Interrupt Handler*) e um **tratador secundário** (SLIH - *Second-Level Interrupt Handler*). O tratador primário, também chamado *hard/fast interrupt handler* ou ainda *top-half handler*, é ativado a cada interrupção recebida e executa rapidamente, com as demais interrupções desabilitadas. Sua tarefa consiste em reconhecer a ocorrência da interrupção junto ao controlador de interrupções, criar um *descritor de evento* contendo os dados da interrupção ocorrida, inserir esse descritor em uma fila de eventos pendentes junto ao *driver* do respectivo dispositivo e notificar o *driver*.

O tratador secundário, também conhecido como *soft/slow interrupt handler* ou ainda *bottom-half handler*, tem por objetivo tratar os eventos pendentes registrados pelo tratador primário. Ele geralmente é escalonado como uma *thread* de núcleo com alta prioridade, executando quando um processador estiver disponível. Embora mais complexa, esta estrutura em dois nível traz vantagens: ao permitir um tratamento mais rápido de cada interrupção, ela minimiza o risco de perder interrupções simultâneas; além disso, a fila de eventos pendentes pode ser analisada para remover eventos redundantes (como atualizações consecutivas de posição do *mouse*).

No Linux, cada interrupção possui sua própria fila de eventos pendentes e seus próprios *top-half* e *bottom-half*. Os tratadores secundários são lançados pelos respectivos tratadores primários, sob a forma de *threads* de núcleo especiais (denominadas *tasklets* ou *workqueues*) [Bovet and Cesati, 2005]. O núcleo Windows NT e seus sucessores implementam o tratamento primário através de rotinas de serviço de interrupção (ISR - *Interrupt Service Routine*). Ao final de sua execução, cada ISR agenda o tratamento secundário da interrupção através de um procedimento postergado (DPC - *Deferred Procedure Call*) [Russinovich et al., 2008]. O sistema *Symbian* usa uma abordagem similar a esta.

Por outro lado, os sistemas Solaris, FreeBSD e MacOS X usam uma abordagem denominada *interrupt threads* [Mauro and McDougall, 2006]. Cada interrupção provoca o lançamento de uma *thread* de núcleo, que é escalonada e compete pelo uso do processador de acordo com sua prioridade. As interrupções têm prioridades que podem estar acima da prioridade do escalonador ou abaixo dela. Como o próprio escalonador também é uma *thread*, interrupções de baixa prioridade podem ser interrompidas pelo escalonador ou por outras interrupções. Por outro lado, interrupções de alta prioridade não são interrompidas pelo escalonador, por isso devem executar rapidamente.

20.5.4 Acesso direto à memória

Na maioria das vezes, o tratamento de operações de entrada/saída é uma operação lenta, pois os dispositivos periféricos são mais lentos que o processador. Além disso, o uso do processador principal para intermediar essas operações é ineficiente, pois implica em transferências adicionais (e desnecessárias) de dados: por exemplo, para transportar um byte de um *buffer* da memória para a interface paralela, esse byte precisa antes ser carregado em um registrador do processador, para em seguida ser enviado ao controlador da interface. Para resolver esse problema, a maioria dos computadores atuais, com exceção de pequenos sistemas embarcados dedicados, oferece mecanismos de **acesso direto à memória** (DMA - *Direct Memory Access*), que permitem transferências diretas entre a memória principal e os controladores de entrada/saída.

O funcionamento do mecanismo de acesso direto à memória em si é relativamente simples. Como exemplo, a seguinte sequência de passos seria executada para a escrita de dados de um *buffer* em memória RAM para o controlador de um disco rígido:

1. o processador acessa as portas do controlador de DMA associado ao dispositivo desejado, para informar o endereço inicial e o tamanho da área de memória RAM contendo os dados a serem escritos no disco. O tamanho da área de memória deve ser um múltiplo do tamanho dos blocos físicos do disco rígido (512 ou 4096 bytes);

2. o controlador de DMA solicita ao controlador do disco a transferência de dados da RAM para o disco e aguarda a conclusão da operação;

3. o controlador do disco recebe os dados da memória;

4. a operação anterior pode ser repetida mais de uma vez, caso a quantidade de dados a transferir seja maior que o tamanho máximo de cada transferência feita pelo controlador de disco;

5. a final da transferência de dados, o controlador de DMA notifica o processador sobre a conclusão da operação, através de uma requisição de interrupção (IRq).

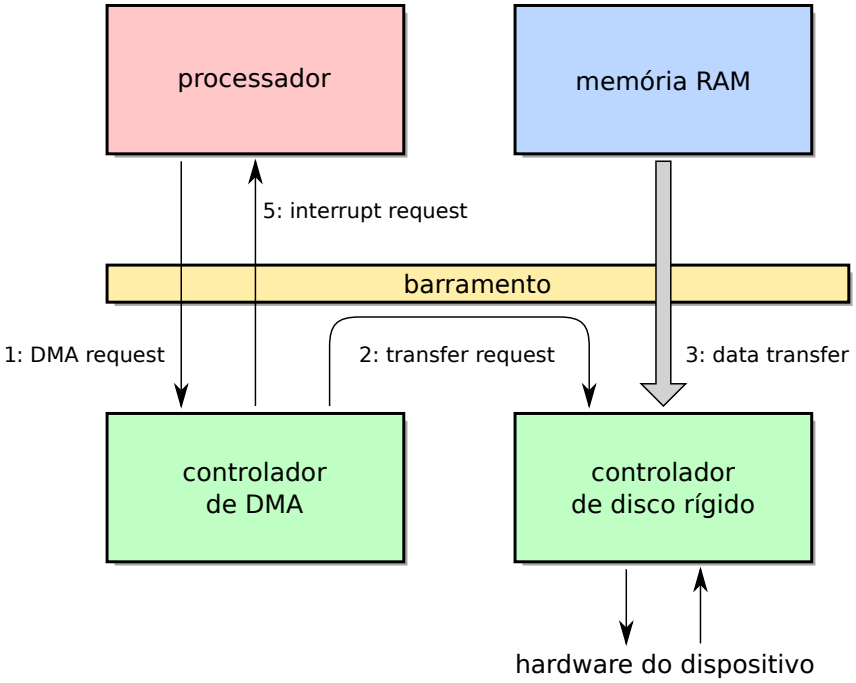


Figura 20.5: Funcionamento do acesso direto à memória.

A dinâmica dessas operações é ilustrada de forma simplificada na Figura 20.5. Uma vez efetuado o passo 1, o processador fica livre para outras atividades[4], enquanto o controlador de DMA e o controlador do disco se encarregam da transferência de dados propriamente dita. O código a seguir representa uma implementação hipotética de rotinas para executar uma operação de entrada/saída através de DMA.

[4]Obviamente pode existir uma contenção (disputa) entre o processador e os controladores no acesso aos barramentos e à memória principal, mas esse problema é atenuado pelo fato do processador poder acessar o conteúdo das memórias *cache* enquanto a transferência DMA é executada. Além disso, circuitos de *arbitragem* intermedeiam o acesso à memória para evitar conflitos. Mais detalhes sobre contenção de acesso à memória durante operações de DMA podem ser obtidos em [Patterson and Henessy, 2005].

```
// requisição da operação de saída através de DMA
void dma_output ()
{
  // solicita uma operação DMA, informando os dados da transferência
  // através da estrutura "dma_data".
  request_dma (dma_data) ;

  // suspende o processo solicitante, liberando o processador.
  suspend_task () ;
}

// rotina de tratamento da interrupção do controlador de DMA
void interrupt_handle ()
{
  // informa o controlador de interrupções que a IRq foi tratada.
  acknowledge_irq () ;

  // saída terminou, acordar o processo solicitante.
  awake_task (...) ;
}
```

A implementação dos mecanismos de DMA depende da arquitetura e do barramento considerado. Computadores mais antigos dispunham de um controlador de DMA único, que oferecia vários **canais de DMA** distintos, permitindo a realização de transferências DMA simultâneas. Já os computadores pessoais usando barramento PCI não possuem um controlador DMA central; ao invés disso, cada controlador de dispositivo conectado ao barramento pode assumir o controle do mesmo para efetuar transferências DMA sem depender do processador principal [Corbet et al., 2005], gerenciando assim seu próprio canal.

No exemplo anterior, a ativação do mecanismo de DMA é dita **síncrona**, pois é feita explicitamente pelo processador, provavelmente em decorrência de uma chamada de sistema. Contudo, a ativação também pode ser **assíncrona**, quando ativada por um dispositivo de entrada/saída que dispõe de dados a serem transferidos para a memória, como ocorre com uma interface de rede ao receber dados provindos da rede.

O mecanismo de DMA é utilizado para transferir grandes blocos de dados diretamente entre a memória RAM e as portas dos dispositivos de entrada/saída, liberando o processador para outras atividades. Todavia, como a configuração de cada operação de DMA é complexa, para pequenas transferências de dados acaba sendo mais rápido e simples usar o processador principal [Bovet and Cesati, 2005]. Por essa razão, o mecanismo de DMA é usado sobretudo nas operações de entrada/saída envolvendo dispositivos que produzem ou consomem grandes volumes de dados, como discos, interfaces de rede, entradas e saídas de áudio, interfaces gráficas e discos.

Referências

D. Bovet and M. Cesati. *Understanding the Linux Kernel, 3rd edition*. O'Reilly Media, Inc, 2005.

J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc, 2005.

J. Mauro and R. McDougall. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*. Prentice-Hall PTR, 2006.

D. Patterson and J. Henessy. *Organização e Projeto de Computadores*. Campus, 2005.

M. Russinovich, D. Solomon, and A. Ionescu. *Microsoft Windows Internals, Fifth Edition*. Microsoft Press, 2008.

Capítulo 21

Discos rígidos

21.1 Introdução

Discos rígidos estão presentes na grande maioria dos computadores pessoais e servidores. Um disco rígido permite o armazenamento persistente (não-volátil) de grandes volumes de dados com baixo custo e tempos de acesso razoáveis. Além disso, a leitura e escrita de dados em um disco rígido é mais simples e flexível que em outros meios, como fitas magnéticas ou discos óticos (CDs, DVDs). Por essas razões, eles são intensivamente utilizados em computadores para o armazenamento de arquivos do sistema operacional, das aplicações e dos dados dos usuários. Os discos rígidos também são frequentemente usados como área de armazenamento de páginas em sistemas de paginação em disco (*swapping* e *paging*, Capítulo 17).

Este capítulo inicialmente apresenta alguns aspectos de hardware relacionados aos discos rígidos, como sua estrutura física e os principais padrões de interface entre o disco e sua controladora no computador. Em seguida, detalha aspectos de software que estão sob a responsabilidade direta do sistema operacional o escalonamento de operações de leitura/escrita no disco. Por fim, apresenta a estratégia RAID para a composição de discos rígidos, que visam melhorar seu desempenho e/ou confiabilidade.

21.2 Estrutura física

Um disco rígido é composto por um ou mais discos metálicos que giram juntos em alta velocidade (usualmente entre 4.200 e 15.000 RPM), acionados por um motor elétrico. Para cada face de cada disco há uma cabeça de leitura móvel, responsável por ler e escrever dados através da magnetização de pequenas áreas da superfície metálica. Cada face é dividida logicamente em **trilhas** (ou **cilindros**) e **setores**; a interseção de uma trilha e um setor em uma face define um **bloco físico**[1], que é a unidade básica de armazenamento e transferência de dados no disco. Até 2010, os discos rígidos usavam blocos físicos de 512 bytes, mas o padrão da indústria migrou nos últimos anos para blocos de 4.096 bytes. A Figura 21.1 apresenta os principais elementos que compõem a estrutura de um disco rígido.

Um disco típico contém várias faces e milhares de trilhas e de setores por face [Patterson and Henessy, 2005], resultando em milhões de blocos de dados disponíveis. Cada bloco pode ser individualmente acessado (lido ou escrito) através de seu *endereço*.

[1] Alguns autores denominam essa intersecção como "setor de trilha" (*track sector*).

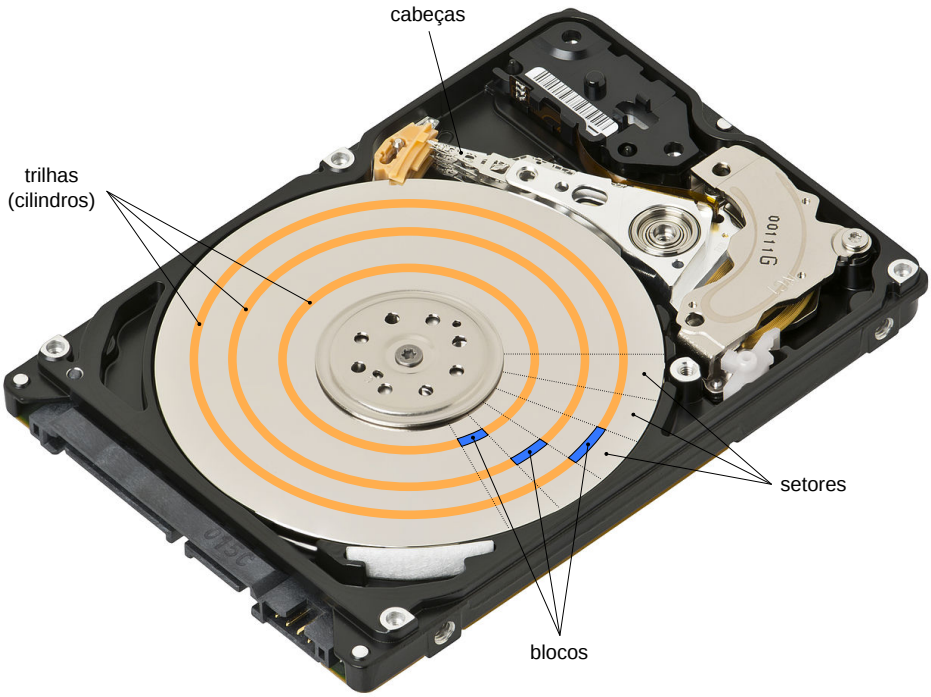


Figura 21.1: Elementos da estrutura de um disco rígido.

Historicamente, o endereçamento dos blocos usava um padrão denominado CHS (*Cylinder-Head-Sector*): para acessar cada bloco, era necessário informar a cabeça (ou seja, a face), o cilindro (trilha) e o setor do disco onde se encontra o bloco. Esse sistema foi mais tarde substituído pelo padrão LBA (*Logical Block Addressing*), no qual os blocos são endereçados linearmente (0, 1, 2, 3, ...), o que é muito mais fácil de gerenciar pelo sistema operacional. Como a estrutura física do disco rígido continua a ter faces, trilhas e setores, uma conversão entre endereços LBA e CHS é feita pelo firmware do disco rígido, de forma transparente para o restante do sistema.

Por serem dispositivos eletromecânicos, os discos rígidos são extremamente lentos, se comparados à velocidade da memória ou do processador. Para cada bloco a ser lido/escrito, a cabeça de leitura deve se posicionar na trilha desejada e aguardar o disco girar até encontrar o setor desejado. Esses dois passos definem o *tempo de busca* (t_s – *seek time*), que é o tempo necessário para a cabeça de leitura se posicionar sobre uma determinada trilha, e a *latência rotacional* (t_r – *rotation latency*), que é o tempo necessário para o disco girar até que o setor desejado esteja sob a cabeça de leitura. Valores médios típicos desses atrasos para discos de uso doméstico são $t_s \approx 10ms$ e $t_r \approx 5ms$. Juntos, esses dois atrasos podem ter um forte impacto no desempenho do acesso a disco.

21.3 Interface de acesso

Como visto na seção 20.2, o sistema operacional deve dispor de *drivers* para interagir com cada controlador de disco e solicitar operações de escrita e leitura de dados. O disco rígido é um dispositivo orientado a blocos, então cada operação de leitura/escrita é feita sobre blocos físicos individuais ou *clusters* (grupos de 2^n blocos físicos contíguos). O acesso ao disco rígido normalmente é feito usando interação por

eventos (Seção 20.5.2): o *driver* solicita uma operação de E/S ao controlador do disco rígido e suspende o fluxo de execução solicitante; quando a operação é completada, o controlador de disco gera uma interrupção que é encaminhada ao *driver*, para retomar a execução e/ou solicitar novas operações. A estratégia de acesso direto à memória (DMA, Seção 20.5.4) também é frequentemente usada, para aumentar o desempenho de transferência de dados.

O *controlador* de disco rígido normalmente é conectado a um barramento do computador. Por sua vez, os discos são conectados ao controlador através de uma interface de conexão que pode usar diversas tecnologias. As mais comuns estão descritas a seguir:

- **IDE**: *Integrated Drive Electronics*, padrão também conhecido como PATA (*Parallell ATA - Advanced Technology Attachment*); surgiu nos anos 1980 e durante muito tempo foi o padrão de interface de discos mais usado em computadores pessoais. Suporta velocidades de até 133 MB/s, através de cabos paralelos de 40 ou 80 vias. Cada barramento IDE suporta até dois dispositivos, em uma configuração mestre/escravo.

- **SATA**: *Serial ATA*, é o padrão de interface de discos em *desktops* e *notebooks* atuais. A transmissão dos dados entre o disco e a controladora é serial, atingindo taxas entre 150 MB/s e 300 MB/s através de cabos com 7 vias.

- **SCSI**: *Small Computer System Interface*, padrão de interface desenvolvida nos anos 1980, foi muito usada em servidores e estações de trabalho de alto desempenho. Um barramento SCSI suporta até 16 dispositivos e atinge taxas de transferência de até 640 MB/s (divididos entre os dispositivos conectados no mesmo barramento).

- **SAS**: *Serial Attached SCSI*, é uma evolução do padrão SCSI, permitindo atingir taxas de transferência de até 600 MB/s em cada dispositivo conectado ao controlador. É usado em equipamentos de alto desempenho, como servidores.

É importante observar que esses padrões de interface não são de uso exclusivo em discos rígidos, muito pelo contrário. Há vários tipos de dispositivos que podem se conectar ao computador através dessas interfaces, como discos de estado sólido (SSD), leitores óticos (CD, DVD), unidades de fita magnética, *scanners*, etc.

21.4 Escalonamento de acessos

Em um sistema operacional multitarefas, várias aplicações e processos podem solicitar acessos ao disco simultaneamente, para escrita e leitura de dados. Devido à sua estrutura mecânica, um disco rígido só pode atender a uma requisição de acesso por vez, o que torna necessário criar uma fila de acessos pendentes. Cada nova requisição de acesso ao disco é colocada nessa fila e o processo solicitante é suspenso até seu pedido ser atendido. Sempre que o disco concluir um acesso, ele informa o sistema operacional, que deve buscar nessa fila a próxima requisição de acesso a ser atendida. A ordem de atendimento das requisições pendentes na fila de acesso ao disco é denominada **escalonamento de disco** e pode ter um grande impacto no desempenho do sistema operacional.

Na sequência do texto serão apresentados alguns algoritmos de escalonamento de disco clássicos. Para exemplificar seu funcionamento, será considerado um disco hipotético com 1.000 blocos (enumerados de 0 ao 999), cuja cabeça de leitura se encontra inicialmente sobre o bloco 500. A fila de pedidos de acesso pendentes contém pedidos de acesso aos seguintes blocos do disco, em sequência:

$$278,\ 914,\ 447,\ 71,\ 161,\ 659,\ 335$$

Todos esses pedidos são de processos distintos, portanto podem ser atendidos em qualquer ordem. Para simplificar, considera-se que nenhum pedido de acesso novo chegará à fila durante a execução do algoritmo de escalonamento.

FCFS (*First Come, First Served*): este algoritmo consiste em atender as requisições na ordem da fila, ou seja, na ordem em foram pedidas pelos processos. É a estratégia mais simples de implementar, mas raramente oferece um bom desempenho. Se os pedidos de acesso estiverem muito espalhados pelo disco, este irá perder muito tempo movendo a cabeça de leitura de um lado para o outro. A sequência de blocos percorridos pela cabeça de leitura é:

$$500 \xrightarrow{222} 278 \xrightarrow{636} 914 \xrightarrow{467} 447 \xrightarrow{376} 71 \xrightarrow{90} 161 \xrightarrow{498} 659 \xrightarrow{324} 335 \text{ (2.613 blocos)}$$

Percebe-se que, para atender os pedidos de leitura na ordem indicada pelo algoritmo FCFS, a cabeça de leitura teve de deslocar-se por 2.613 blocos do disco ($222 + 636 + 467 + \cdots$).

SSTF (*Shortest Seek Time First – Menor Tempo de Busca Primeiro*): esta estratégia de escalonamento de disco consiste em sempre atender o pedido que está mais próximo da posição atual da cabeça de leitura (que é geralmente a posição do último pedido atendido). Dessa forma, ela busca reduzir os movimentos da cabeça de leitura, e com isso o tempo perdido entre os acessos.

A sequência de acesso efetuadas pelo algoritmo SSTF está indicada a seguir. Pode-se observar uma grande redução da movimentação da cabeça de leitura em relação à estratégia FCFS, que passou de 2.613 para 1.272 blocos percorridos. Contudo, a estratégia SSTF não garante obter sempre um percurso mínimo.

$$500 \xrightarrow{53} 447 \xrightarrow{112} 335 \xrightarrow{57} 278 \xrightarrow{117} 161 \xrightarrow{90} 71 \xrightarrow{588} 659 \xrightarrow{255} 914 \text{ (1.272 blocos)}$$

Apesar de oferecer um ótimo desempenho, a estratégia SSTF pode levar à inanição (*starvation*) de requisições de acesso: caso existam muitas requisições em uma determinada região do disco, pedidos de acesso a blocos distantes dessa região podem ficar esperando indefinidamente. Para resolver esse problema, torna-se necessário implementar uma estratégia de *envelhecimento* dos pedidos pendentes.

SCAN: neste algoritmo, a cabeça "varre" (*scan*) continuamente o disco, do início ao final, atendendo os pedidos que encontra pela frente; ao atingir o final do disco, ela inverte seu sentido de movimento e volta, atendendo os próximos pedidos. Apesar de ser mais lento que SSTF, este algoritmo atende os pedidos

de forma mais uniforme ao longo do disco, eliminando o risco de inanição de pedidos e mantendo um desempenho equilibrado para todos os processos. Ele é adequado para sistemas com muitos pedidos simultâneos de acesso a disco, como servidores de arquivos. O comportamento deste algoritmo para a sequência de requisições de exemplo está indicado a seguir:

$$500 \xrightarrow{159} 659 \xrightarrow{255} 914 \xrightarrow{85} 999 \xrightarrow{552} 447 \xrightarrow{112} 335 \xrightarrow{57} 278 \xrightarrow{117} 161 \xrightarrow{90} 71 \text{ (1.337 blocos)}$$

C-SCAN: esta é uma variante "circular" do algoritmo SCAN, na qual a cabeça de leitura varre o disco somente em uma direção. Ao atingir o final do disco, ela retorna diretamente ao início do disco, sem atender os pedidos intermediários, e recomeça a varredura. O nome "circular" é devido ao disco ser visto pelo algoritmo como uma lista circular de blocos. Sua vantagem em relação ao algoritmo SCAN é prover um tempo de espera mais homogêneo aos pedidos pendentes, o que é importante em servidores. O comportamento deste algoritmo para a sequência de requisições de exemplo está indicado a seguir:

$$500 \xrightarrow{159} 659 \xrightarrow{255} 914 \xrightarrow{85} 999 \xrightarrow{999} 0 \xrightarrow{71} 71 \xrightarrow{90} 161 \xrightarrow{117} 278 \xrightarrow{57} 335 \xrightarrow{112} 447 \text{ (1.776 blocos)}$$

LOOK: é uma otimização do algoritmo SCAN, na qual a cabeça do disco não avança até o final do disco, mas inverte seu movimento assim que tiver tratado o último pedido em cada sentido do movimento:

$$500 \xrightarrow{159} 659 \xrightarrow{255} 914 \xrightarrow{467} 447 \xrightarrow{112} 335 \xrightarrow{57} 278 \xrightarrow{117} 161 \xrightarrow{90} 71 \text{ (1.257 blocos)}$$

C-LOOK: idem, otimizando o algoritmo C-SCAN:

$$500 \xrightarrow{159} 659 \xrightarrow{255} 914 \xrightarrow{843} 71 \xrightarrow{90} 161 \xrightarrow{117} 278 \xrightarrow{57} 335 \xrightarrow{112} 447 \text{ (1.644 blocos)}$$

Os algoritmos SCAN, C-SCAN e suas variantes LOOK e C-LOOK são denominados coletivamente de *Algoritmo do Elevador*, pois seu comportamento reproduz o comportamento do elevador em um edifício: a cabeça de leitura avança em um sentido, atendendo as requisições dos usuários; ao chegar ao final, inverte seu sentido e retorna [Silberschatz et al., 2001].

A Figura 21.2 apresenta graficamente o comportamento dos seis algoritmos apresentados acima (FCFS, SSTF, SCAN, C-SCAN, LOOK e C-LOOK). No gráfico, as linhas pontilhadas representam os blocos a serem lidos/escritos, as linhas inclinadas representam a movimentação da cabeça do disco, e as barras horizontais mais grossas representam as operações de leitura/escrita de blocos.

Sistemas operacionais reais, como Solaris, Windows e Linux, utilizam escalonadores de disco bem mais sofisticados. No caso do sistema Linux, por exemplo, os seguintes escalonadores de disco estão presentes no núcleo, podendo ser configurados pelo administrador do sistema em função das características dos discos e da carga de trabalho do sistema [Love, 2010; Bovet and Cesati, 2005]:

Noop (*No-Operation*): é o escalonador mais simples, baseado em FCFS, que não reordena os pedidos de acesso, apenas agrupa os pedidos direcionados ao mesmo bloco ou a blocos adjacentes. Este escalonador é voltado para discos de estado sólido (SSD, baseados em memória *flash*) ou sistemas de armazenamento que façam seu próprio escalonamento interno, como sistemas RAID (vide Seção 21.5).

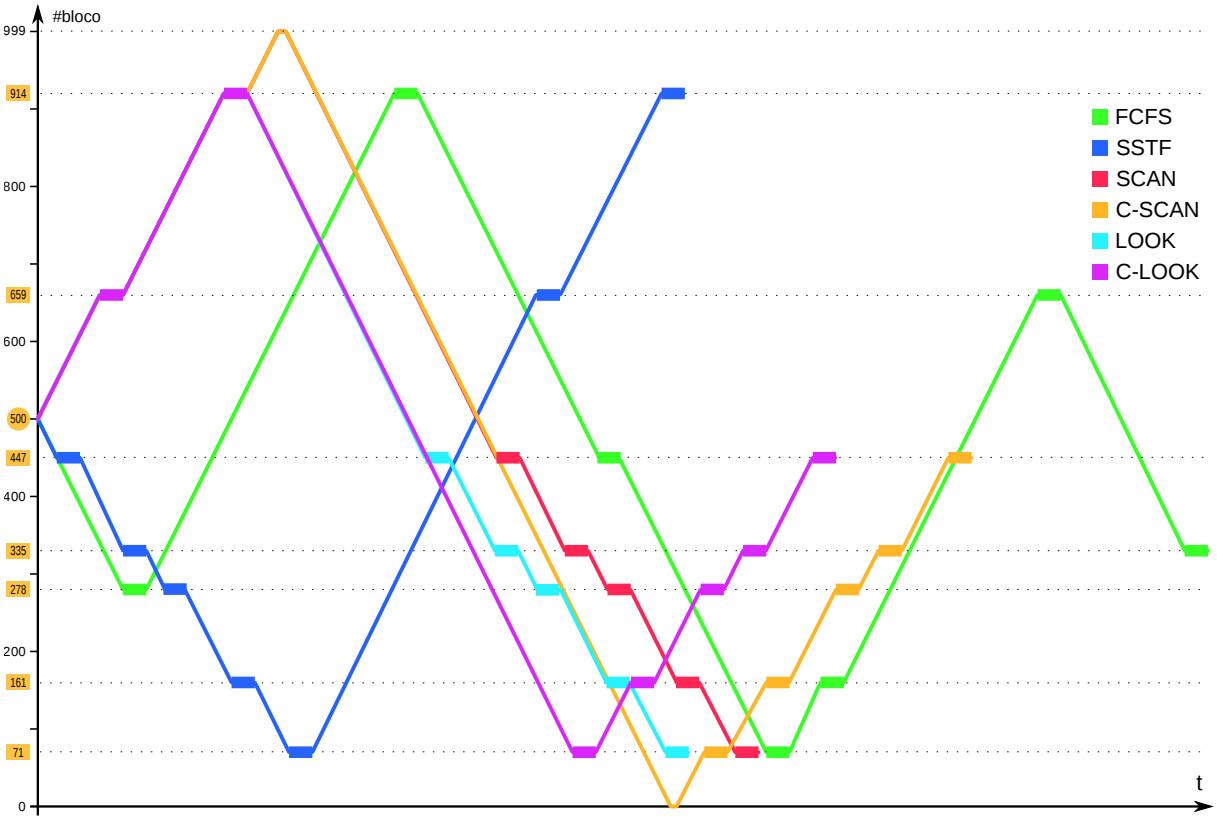


Figura 21.2: Algoritmos de escalonamento de disco.

Deadline: este escalonador é baseado no algoritmo do elevador circular (C-SCAN), mas associa um prazo (*deadline*) a cada requisição, para evitar problemas de inanição. Como os pedidos de leitura implicam no bloqueio dos processos solicitantes, eles recebem um prazo de 500 *ms*; pedidos de escrita podem ser executados de forma assíncrona, sem bloquear o processo solicitante, por isso recebem um prazo maior, de 5 segundos. O escalonador processa os pedidos usando o algoritmo do elevador, mas prioriza os pedidos cujos prazos estejam esgotando.

Anticipatory: este algoritmo é baseado no anterior (*deadline*), mas busca se antecipar às operações de leitura de dados feitas pelos processos. Como as operações de leitura são geralmente feitas de forma sequencial (em blocos contíguos ou próximos), a cada operação de leitura realizada o escalonador aguarda um certo tempo (por default 6 *ms*) por um novo pedido de leitura naquela mesma região do disco, que é imediatamente atendido. Caso não surja nenhum pedido novo, o escalonador volta a tratar a fila de pedidos pendentes normalmente. Essa espera por pedidos adjacentes melhora o desempenho das operações de leitura emitidas pelo mesmo processo.

CFQ (*Completely Fair Queuing*): os pedidos dos processos são divididos em várias filas (64 filas por default); cada fila recebe uma fatia de tempo para acesso ao disco, que varia de acordo com a prioridade de entrada/saída dos processos contidos na mesma. Este é o escalonador default do Linux na maioria das distribuições mais recentes.

21.5 Sistemas RAID

Apesar dos avanços dos sistemas de armazenamento em estado sólido (como os dispositivos baseados em memórias *flash*), os discos rígidos continuam a ser o principal meio de armazenamento não-volátil de grandes volumes de dados. Os discos atuais têm capacidades de armazenamento impressionantes: encontram-se facilmente no mercado discos rígidos com capacidade da ordem de terabytes para computadores domésticos.

Entretanto, o desempenho dos discos rígidos evolui a uma velocidade muito menor que a observada nos demais componentes dos computadores, como processadores, memórias e barramentos. Com isso, o acesso aos discos constitui um dos maiores gargalos de desempenhos nos sistemas de computação. Boa parte do baixo desempenho no acesso aos discos é devida aos aspectos mecânicos do disco, como a latência rotacional e o tempo de posicionamento da cabeça de leitura do disco (vide Seção 21.4) [Chen et al., 1994].

Outro problema relevante associado aos discos rígidos diz respeito à sua confiabilidade. Os componentes internos do disco podem falhar, levando à perda de dados. Essas falhas podem estar localizadas no meio magnético, ficando restritas a alguns setores, ou podem estar nos componentes mecânicos/eletrônicos do disco, levando à corrupção ou mesmo à perda total dos dados armazenados.

Buscando soluções eficientes para os problemas de desempenho e confiabilidade dos discos rígidos, pesquisadores da Universidade de Berkeley, na Califórnia, propuseram em 1988 a construção de discos virtuais compostos por conjuntos de discos físicos, que eles denominaram RAID – *Redundant Array of Inexpensive Disks*[2] [Patterson et al., 1988], que em português pode ser traduzido como *Conjunto Redundante de Discos Econômicos*.

Um sistema RAID é constituído de dois ou mais discos rígidos que são vistos pelo sistema operacional e pelas aplicações como um único disco lógico, ou seja, um grande espaço contíguo de armazenamento de dados. O objetivo central de um sistema RAID é proporcionar mais desempenho nas operações de transferência de dados, através do paralelismo no acesso aos vários discos, e também mais confiabilidade no armazenamento, usando mecanismos de redundância dos dados armazenados nos discos, como cópias de dados ou códigos corretores de erros.

Um sistema RAID pode ser construído "por hardware", usando uma placa controladora dedicada a esse fim, à qual estão conectados os discos rígidos. Essa placa controladora oferece a visão de um disco lógico único ao restante do computador. Também pode ser usada uma abordagem "por software", na qual são usados *drivers* apropriados dentro do sistema operacional para combinar os discos rígidos conectados ao computador em um único disco lógico. Obviamente, a solução por software é mais flexível e econômica, por não exigir uma placa controladora dedicada, enquanto a solução por hardware é mais robusta e tem um desempenho melhor. É importante observar que os sistemas RAID operam abaixo dos sistemas de arquivos, ou seja, eles se preocupam apenas com o armazenamento e recuperação de blocos de dados.

Há várias formas de se organizar um conjunto de discos rígidos em RAID, cada uma com suas próprias características de desempenho e confiabilidade. Essas formas de organização são usualmente chamadas *Níveis RAID*. Os níveis RAID padronizados pela *Storage Networking Industry Association* são [SNIA]:

[2] Mais recentemente alguns autores adotaram a expressão *Redundant Array of Independent Disks* para a sigla RAID, buscando evitar a subjetividade da palavra *Inexpensive* (econômico).

RAID 0 (linear): neste nível os discos físicos (ou partições) são simplesmente concatenados em sequência para construir um disco lógico. Essa abordagem, ilustrada na Figura 21.3, é denominada por alguns autores de *RAID 0 linear*, enquanto outros a denominam JBoD (*Just a Bunch of Disks* – apenas um punhado de discos). Na figura, $d_i.b_j$ indica o bloco j do disco físico i.

Em teoria, esta estratégia oferece maior velocidade de leitura e de escrita, pois acessos a blocos em discos físicos distintos podem ser feitos em paralelo. Entretanto, esse ganho pode ser pequeno caso os acessos se concentrem em uma área pequena do disco lógico, pois ela provavelmente estará mapeada em um mesmo disco físico (o acesso a cada disco é sempre sequencial). Além disso, alguns discos tendem a ser mais usados que outros.

Esta abordagem não oferece nenhuma redundância de dados, o que a torna suscetível a erros de disco: caso um disco falhe, todos os blocos armazenados nele serão perdidos. Como a probabilidade de falhas aumenta com o número de discos, esta abordagem acaba por reduzir a confiabilidade do sistema de discos.

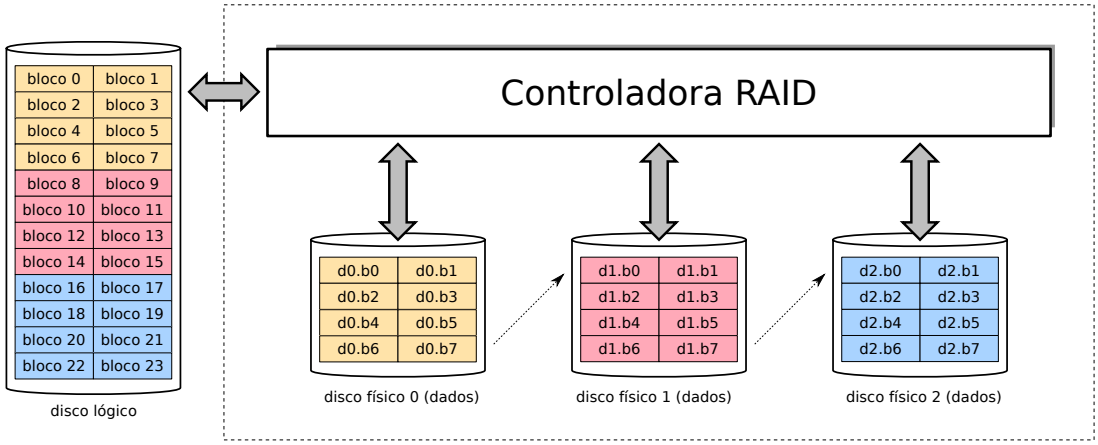


Figura 21.3: RAID nível 0 (*linear*).

RAID 0 (striping): neste nível os discos físicos são divididos em áreas de tamanhos fixo chamadas *fatias* ou *faixas* (*stripes*). Cada fatia de disco físico armazena um ou mais blocos do disco lógico (tipicamente são usadas faixas de 32, 64 ou 128 KBytes). As fatias são concatenadas usando uma estratégia *round-robin* para construir o disco lógico, como mostra a Figura 21.4.

O maior espalhamento dos blocos sobre os discos físicos contribui para distribuir melhor a carga de acessos entre eles e assim ter um melhor desempenho. Suas características de suporte a grande volume de dados e alto desempenho em leitura/escrita tornam esta abordagem adequada para ambientes que precisam processar grandes volumes de dados temporários, como os sistemas de computação científica [Chen et al., 1994].

RAID 1: neste nível, cada disco físico possui um "espelho", ou seja, outro disco com a cópia de seu conteúdo, sendo por isso comumente chamado de *espelhamento de discos*. A Figura 21.5 mostra uma configuração simples deste nível, com dois discos físicos. Caso hajam mais de dois discos, devem ser incorporadas técnicas de RAID 0 para organizar a distribuição dos dados sobre eles (o que leva a configurações denominadas RAID 0+1, RAID 1+0 ou RAID 1E).

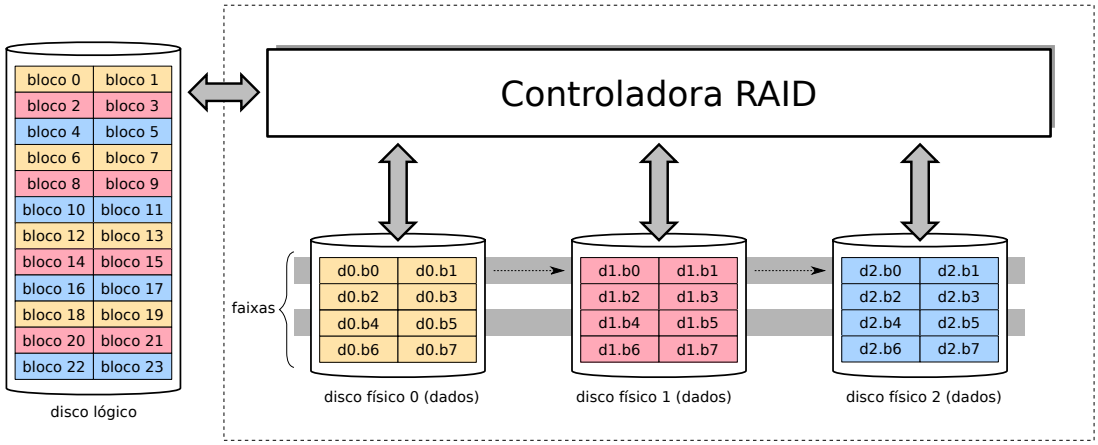


Figura 21.4: RAID nível 0 (*striping*).

Esta abordagem oferece uma excelente confiabilidade, pois cada bloco lógico está escrito em dois discos distintos; caso um deles falhe, o outro continua acessível. O desempenho em leituras também é beneficiado, pois a controladora pode distribuir as leituras entre as cópias. Contudo, não há ganho de desempenho em escrita, pois cada operação de escrita deve ser replicada em todos os discos. Além disso, seu custo de implantação é elevado, pois são necessários dois discos físicos para cada disco lógico.

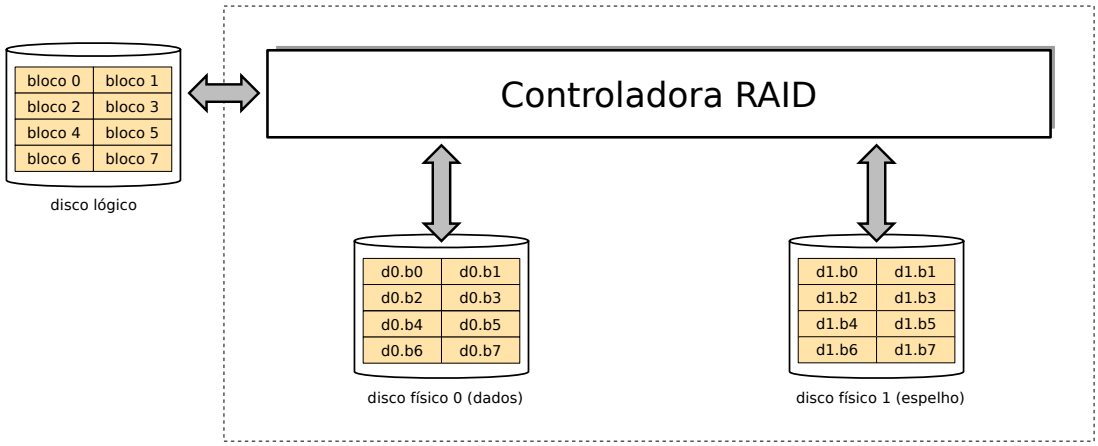


Figura 21.5: RAID nível 1 (espelhamento).

RAID 2: neste nível os dados são "fatiados" em bits individuais que são escritos nos discos físicos em sequência; discos adicionais são usados para armazenar códigos corretores de erros (*Hamming Codes*), em um arranjo similar ao usado nas memórias RAM. Esses códigos corretores de erros permitem resgatar dados no caso de falha em blocos ou discos de dados. Por ser pouco eficiente e complexo de implementar, este nível não é usado na prática.

RAID 3: de forma similar ao RAID 2, este nível fatia os dados em bits escritos nos discos em sequência. Um disco adicional é usado para armazenar o bit de paridade de cada fatia de bits, sendo usado para a recuperação de erros nos demais discos. A cada leitura ou escrita, os dados de paridade devem ser atualizados, o que transforma o disco de paridade em um gargalo de desempenho.

RAID 4: esta abordagem é similar ao RAID 3, com a diferença de que o fatiamento é feito em blocos ao invés de bits, como mostra a Figura 21.6. Ela sofre dos mesmos problemas de desempenho que o RAID 3, sendo por isso pouco usada. Todavia, ela serve como base conceitual para o RAID 5.

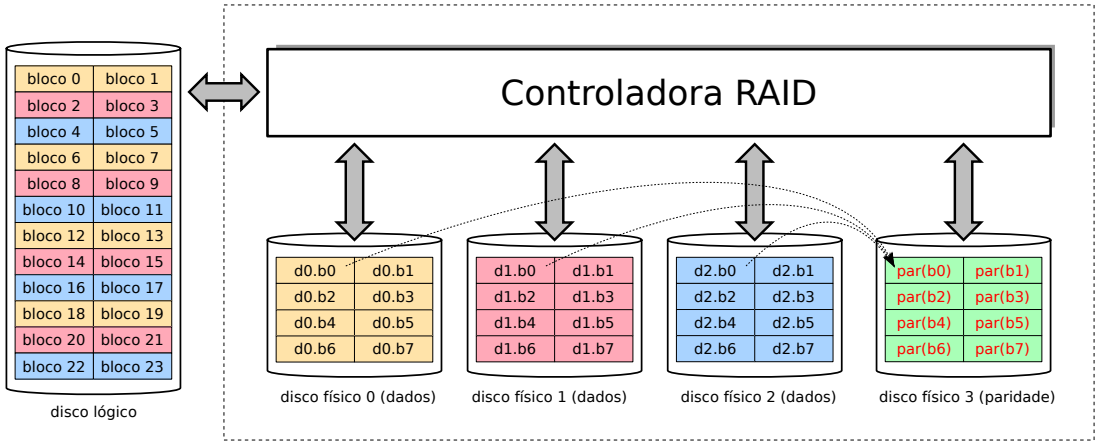


Figura 21.6: RAID nível 4 (disco de paridade).

RAID 5: assim como o RAID 4, esta abordagem também armazena informações de paridade para tolerar falhas em blocos ou discos. Todavia, essas informações não ficam concentradas em um único disco físico, sendo distribuídas uniformemente entre eles. A Figura 21.7 ilustra uma possibilidade de distribuição das informações de paridade. Essa estratégia elimina o gargalo de desempenho no acesso aos dados de paridade visto no RAID 4. Esta é a abordagem de RAID mais popular, por oferecer um bom desempenho e redundância de dados, desperdiçando menos espaço que o espelhamento (RAID 1).

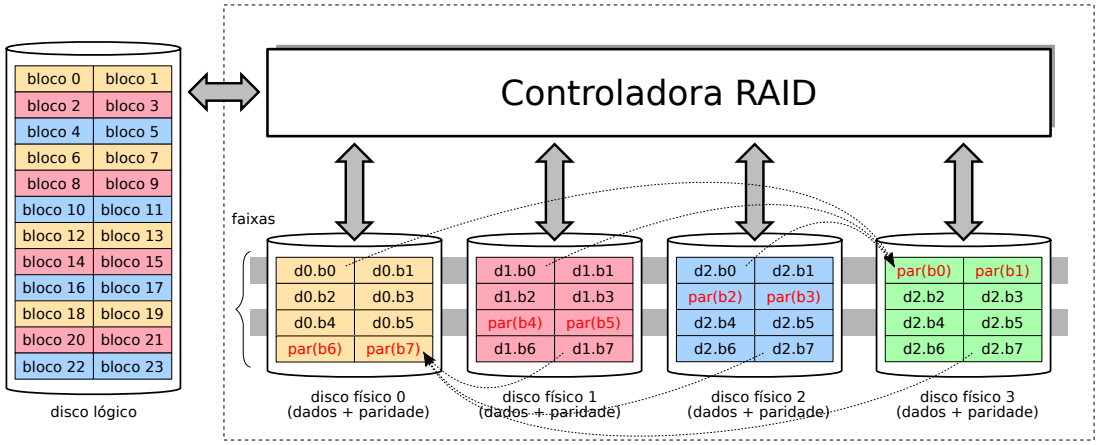


Figura 21.7: RAID nível 5 (paridade distribuída).

RAID 6: é uma extensão do nível RAID 5 que utiliza blocos com códigos corretores de erros de *Reed-Solomon*, além dos blocos de paridade. Esta redundância extra demanda dois discos adicionais, mas permite tolerar falhas simultâneas de até dois discos.

Além dos níveis padronizados, no mercado podem ser encontrados produtos oferecendo outros níveis RAID, como 1+0, 0+1, 50, 100, etc., que muitas vezes implementam combinações dos níveis básicos ou então soluções proprietárias. Outra observação importante é que os vários níveis de RAID não têm necessariamente uma relação hierárquica entre si, ou seja, um sistema RAID 5 não é necessariamente melhor que um sistema RAID 1, pois isso depende do campo de aplicação do sistema. Uma descrição mais aprofundada dos vários níveis RAID, de suas variantes e características pode ser encontrada em [Chen et al., 1994] e [SNIA].

A Tabela 21.1 traz um comparativo entre as principais características das diversas estratégias de RAID apresentadas nesta seção. As velocidades de leitura e de escrita são analisadas em relação às que seriam observadas em um disco isolado de mesmo tipo. A coluna *Espaço* indica a fração do espaço total dos N discos que está disponível para o armazenamento de dados. A coluna *Falhas* indica a quantidade de discos que pode falhar sem causar perda de dados. A coluna *Discos* define o número mínimo de discos necessários para a configuração.

Tabela 21.1: Comparativo das principais estratégias de RAID

Estratégia	Velocidade em leitura	Velocidade em escrita	Espaço	Falhas	Discos
RAID 0 linear	maior (blocos em discos distintos são acessados em paralelo)	maior (blocos em discos distintos são acessados em paralelo)	100%	0	≥ 2
RAID 0 striping	idem ao anterior	idem ao anterior	100%	0	≥ 2
RAID 1	maior (leituras podem ser paralelizadas sobre os discos)	igual (todas cópias devem ser atualizadas)	50%	$N/2$	≥ 2
RAID 4	idem ao anterior	menor (gargalo no acesso ao disco de paridade)	$\frac{(N-1)}{N}$%	1	≥ 3
RAID 5	idem ao anterior	maior (blocos em discos distintos são acessados em paralelo, inclusive os dados de paridade)	$\frac{(N-1)}{N}$%	1	≥ 3
RAID 6	idem ao anterior	idem ao anterior	$\frac{(N-2)}{N}$%	2	≥ 4

Referências

D. Bovet and M. Cesati. *Understanding the Linux Kernel, 3rd edition*. O'Reilly Media, Inc, 2005.

P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26:145–185, June 1994.

R. Love. *Linux Kernel Development, Third Edition*. Addison-Wesley, 2010.

D. Patterson and J. Henessy. *Organização e Projeto de Computadores*. Campus, 2005.

D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD International Conference on Management of Data*, pages 109–116. ACM, 1988.

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

SNIA. *Common RAID Disk Data Format Specification*. SNIA – Storage Networking Industry Association, March 2009. Version 2.0 Revision 19.

Parte VI

Gestão de arquivos

Capítulo 22

O conceito de arquivo

Desde os primórdios da computação, percebeu-se a necessidade de armazenar informações para uso posterior, como programas e dados. Hoje, parte importante do uso de um computador consiste em recuperar e apresentar informações previamente armazenadas, como documentos, fotografias, músicas e vídeos. O próprio sistema operacional também precisa manter informações armazenadas para uso posterior, como programas, bibliotecas e configurações. Para simplificar o armazenamento e busca dessas informações, surgiu o conceito de *arquivo*.

Este e os próximos capítulos apresentam os principais conceitos relacionados a arquivos, seu uso e a forma como são implementados e gerenciados pelo sistema operacional.

22.1 Elementos básicos

Um **arquivo** é essencialmente uma sequência de bytes armazenada em um dispositivo físico não volátil, como um disco rígido ou de estado sólido, que preserva seu conteúdo mesmo quando desligado. Cada arquivo possui um nome, ou outra referência, que permite sua localização e acesso. Do ponto de vista do usuário e das aplicações, o arquivo é a unidade básica de armazenamento de informação em um dispositivo não volátil, pois para eles não há forma mais simples de armazenamento persistente de dados. Arquivos são extremamente versáteis em conteúdo e capacidade: podem conter desde um texto com alguns poucos caracteres até vídeos com dezenas de gigabytes, ou ainda mais.

Como um dispositivo de armazenamento pode conter milhões de arquivos, estes são organizados em estruturas hierárquicas denominadas **diretórios**, para facilitar sua localização e acesso pelos usuários. Essa estrutura hierárquica está ilustrada no exemplo da Figura 22.1, que mostra diretórios (`Administrativo`, `Ensino`), subdiretórios (`avaliacoes`, `documentos`) e arquivos (`ci067.txt`, `programa.odt`, `cpp.png`). Os diretórios serão estudados no Capítulo 25.

A organização do conteúdo dos arquivos e diretórios dentro de um dispositivo físico é denominada **sistema de arquivos**. Um sistema de arquivos pode ser visto como uma imensa estrutura de dados armazenada de forma persistente no dispositivo físico. Existe um grande número de sistemas de arquivos, dentre os quais podem ser citados o NTFS (nos sistemas Windows), Ext2/Ext3/Ext4 (Linux), HPFS (MacOS), FFS (Solaris) e FAT (usado em *pendrives* USB, câmeras fotográficas digitais e leitores MP3). A organização dos sistemas de arquivos será discutida no Capítulo 24.

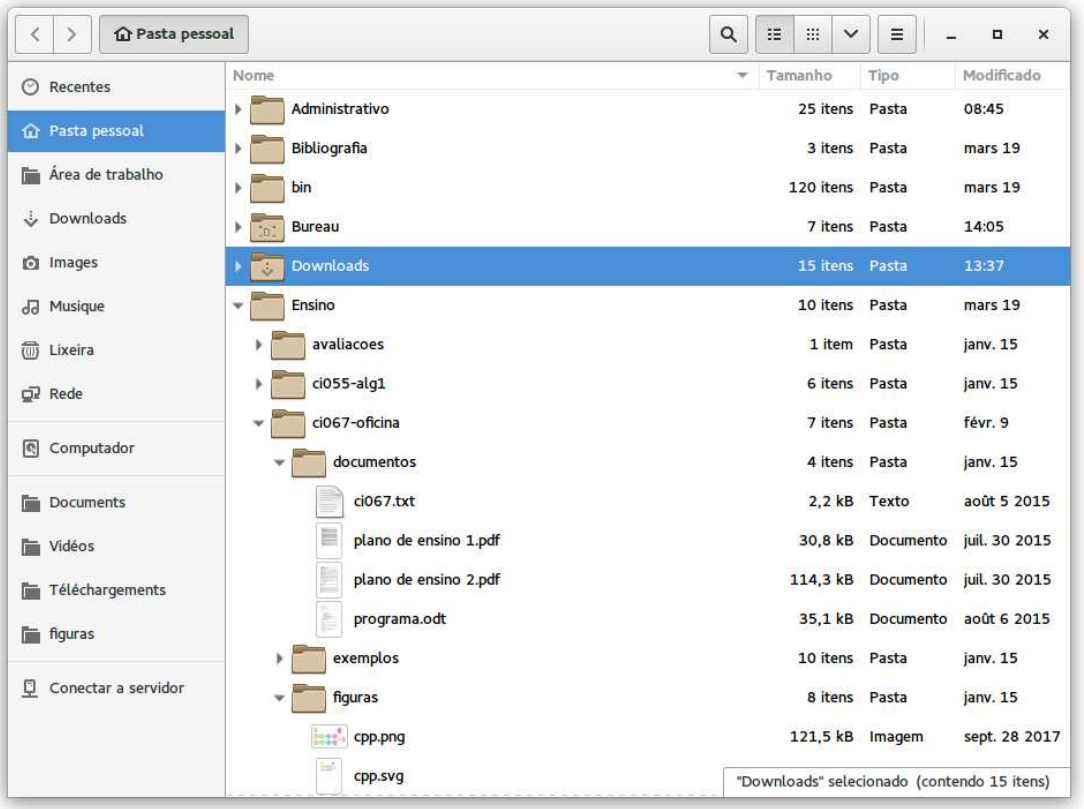


Figura 22.1: Estrutura hierárquica de arquivos e diretórios.

Finalmente, Um dispositivo físico é estruturado em um ou mais **volumes** (ou partições); cada volume pode armazenar um sistema de arquivos próprio. Assim, um mesmo disco rígido pode conter volumes com diferentes sistemas de arquivos, como FAT, NTFS ou EXT4, por exemplo.

22.2 Atributos e operações

Um arquivo é uma unidade de armazenamento de informações que podem ser documentos, imagens, código executável, etc. Além de seu conteúdo, um arquivo é caracterizado por *atributos*, que são informações adicionais relativas ao conteúdo, e *operações*, ou ações que podem ser realizadas sobre o conteúdo e/ou sobre os atributos.

Os **atributos** dos arquivos variam de acordo com o sistema de arquivos utilizado. Os atributos mais usuais, presentes na maioria dos sistemas, são:

Nome: *string* que identifica o arquivo para o usuário, como "`foto1.jpg`", "`relatÃşrio.pdf`", "`hello.c`", etc.;

Tipo: indicação do formato dos dados contidos no arquivo, como áudio, vídeo, imagem, texto, etc. Muitos sistemas operacionais usam parte do nome do arquivo para identificar o tipo de seu conteúdo, na forma de uma extensão: "`.doc`", "`.jpg`", "`.mp3`", etc.;

Tamanho: indicação do tamanho do conteúdo do arquivo, geralmente em bytes;

Datas: para fins de gerência, é importante manter as datas mais importantes relacionadas ao arquivo, como suas datas de criação, de último acesso e de última modificação do conteúdo;

Proprietário: em sistemas multiusuários, cada arquivo tem um proprietário, que deve estar corretamente identificado;

Permissões de acesso: indicam que usuários têm acesso àquele arquivo e que formas de acesso são permitidas (leitura, escrita, remoção, etc.);

Localização: indicação do dispositivo físico onde o arquivo se encontra e da posição do arquivo dentro do mesmo;

Além destes, vários outros atributos podem ser associados a um arquivo, por exemplo para indicar se ele é um arquivo de sistema, se está visível aos usuários, se tem conteúdo binário ou textual, etc. Cada sistema de arquivos normalmente define seus próprios atributos específicos, além dos atributos usuais acima.

Nem sempre os atributos oferecidos por um sistema de arquivos são suficientes para exprimir todas as informações a respeito de um arquivo. Nesse caso, a "solução" encontrada pelos usuários é usar o nome do arquivo para registrar a informação desejada. Por exemplo, em muitos sistemas a parte final do nome do arquivo (sua extensão) é usada para identificar o formato de seu conteúdo. Outra situação frequente é usar parte do nome do arquivo para identificar diferentes versões do mesmo conteúdo[1]: `relat-v1.txt`, `relat-v2.txt`, etc.

As aplicações e o sistema operacional usam arquivos para armazenar e recuperar dados. O acesso aos arquivos é feito através de um conjunto de **operações**, geralmente implementadas sob a forma de chamadas de sistema e funções de bibliotecas. As operações básicas envolvendo arquivos são:

Criar: a criação de um novo arquivo implica em alocar espaço para ele no dispositivo de armazenamento e definir valores para seus atributos (nome, localização, proprietário, permissões de acesso, datas, etc.);

Abrir: antes que uma aplicação possa ler ou escrever dados em um arquivo, ela deve solicitar ao sistema operacional a "abertura" desse arquivo. O sistema irá então verificar se o arquivo desejado existe, verificar se as permissões associadas ao arquivo permitem aquele acesso, localizar seu conteúdo no dispositivo de armazenamento e criar uma referência para ele na memória da aplicação;

Ler: permite transferir dados presentes no arquivo para uma área de memória da aplicação;

Escrever: permite transferir dados na memória da aplicação para o arquivo no dispositivo físico; os novos dados podem ser adicionados ao final do arquivo ou sobrescrever dados já existentes;

Fechar: ao concluir o uso do arquivo, a aplicação deve informar ao sistema operacional que o mesmo não é mais necessário, a fim de liberar as estruturas de gerência do arquivo mantidas na memória do núcleo;

[1] Alguns sistemas operacionais, como o *TOPS-20* e o *OpenVMS*, possuem sistemas de arquivos com suporte automático a múltiplas versões do mesmo arquivo.

Remover: para eliminar o arquivo do dispositivo, descartando seus dados e liberando o espaço ocupado por ele.

Alterar atributos: para modificar os valores dos atributos do arquivo, como nome, proprietário, permissões, datas, etc.

Além dessas operações básicas, outras operações podem ser definidas, como truncar, copiar, mover ou renomear arquivos. Todavia, essas operações geralmente podem ser construídas usando as operações básicas acima. Por exemplo, para copiar um arquivo *A* em um novo arquivo *B*, os seguintes passos seriam necessários: abrir *A*; criar e abrir *B*; ler conteúdo de *A* e escrever em *B*; fechar *A* e *B*.

22.3 Formatos de arquivos

Arquivos permitem armazenar dados para uso posterior. A forma como esses dados são estocados dentro do arquivo é denominada *formato do arquivo*. Esta seção discute os formatos de arquivos mais usuais.

22.3.1 Sequência de bytes

Em sua forma mais simples, um arquivo contém basicamente uma sequência de bytes. Essa sequência de bytes pode ser estruturada de forma a representar diferentes tipos de informação, como imagens, música, textos ou código executável. Essa estrutura interna do arquivo pode ser entendida pelo núcleo do sistema operacional ou somente pelas aplicações que acessam esse conteúdo.

O núcleo do sistema geralmente reconhece apenas alguns poucos formatos de arquivos, como binários executáveis e bibliotecas. Os demais formatos de arquivos são vistos pelo núcleo apenas como sequências de bytes opacas, sem um significado específico, cabendo às aplicações interessadas interpretá-las.

Uma aplicação pode definir um formato próprio para armazenar seus dados, ou pode seguir formatos padronizados. Por exemplo, há um grande número de formatos padronizados para o armazenamento de imagens, como JPEG, GIF, PNG e TIFF, mas também existem formatos de arquivos proprietários, definidos por algumas aplicações específicas, como os formatos PSD e XCF (dos editores gráficos *Adobe Photoshop* e *GIMP*, respectivamente). A adoção de um formato proprietário ou exclusivo limita o uso das informações armazenadas, pois somente aplicações que reconheçam aquele formato específico conseguem ler corretamente as informações contidas no arquivo.

22.3.2 Arquivos de registros

Alguns núcleos de sistemas operacionais oferecem arquivos com estruturas internas que vão além da simples sequência de bytes. Por exemplo, o sistema *OpenVMS* [Rice, 2000] proporciona *arquivos baseados em registros*, cujo conteúdo é visto pelas aplicações como uma sequência linear de registros de tamanho fixo ou variável, e também *arquivos indexados*, nos quais podem ser armazenados pares {*chave*/*valor*}, de forma similar a um banco de dados relacional. A Figura 22.2 ilustra a estrutura interna desses dois tipos de arquivos.

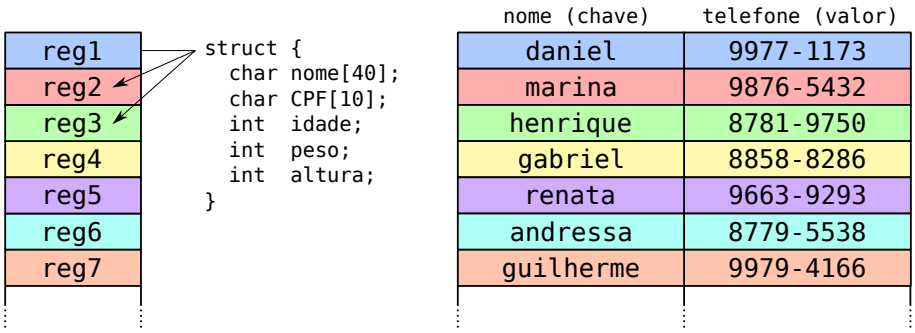


Figura 22.2: Arquivos estruturados: registros em sequência e registros indexados.

Nos sistemas operacionais cujo núcleo não suporta arquivos estruturados em registros, essa funcionalidade pode ser facilmente obtida através de bibliotecas específicas ou do suporte de execução de algumas linguagens de programação. Por exemplo, as bibliotecas *Berkeley DB* e *SQLite*, disponíveis em plataformas UNIX, oferecem suporte à indexação de registros sobre arquivos UNIX convencionais.

22.3.3 Arquivos de texto

Um tipo de arquivo de uso muito frequente é o arquivo de *texto puro* (ou *plain text*). Esse tipo de arquivo é usado para armazenar informações textuais simples, como códigos-fonte de programas, arquivos de configuração, páginas HTML, dados em XML, etc. Um arquivo de texto é formado por linhas de caracteres de tamanho variável, separadas por caracteres de controle. Nos sistemas UNIX, as linhas são separadas por um caractere *New Line* (ASCII 10 ou "\n"). Já nos sistemas DOS/Windows, as linhas de um arquivo de texto são separadas por dois caracteres: o caractere *Carriage Return* (ASCII 13 ou "\r") seguido do caractere *New Line*.

Por exemplo, considere o seguinte programa em C armazenado em um arquivo `hello.c` (os caracteres "␣" indicam espaços em branco):

```
1 int␣main()
2 {
3 ␣␣printf("Hello,␣world\n");
4 ␣␣exit(0);
5 }
```

O arquivo de texto `hello.c` seria armazenado usando a seguinte sequência de bytes[2] em um sistema UNIX:

```
1 0000  69 6e 74 20 6d 61 69 6e 28 29 0a 7b 0a 20 20 70
2        i  n  t  ␣  m  a  i  n  (  ) \n  {  \n ␣  ␣  p
3 0010  72 69 6e 74 66 28 22 48 65 6c 6c 6f 2c 20 77 6f
4        r  i  n  t  f  (  "  H  e  l  l  o  ,  ␣  w  o
5 0020  72 6c 64 5c 6e 22 29 3b 0a 20 20 65 78 69 74 28
6        r  l  d  \  n  "  )  ;  \n ␣  ␣  e  x  i  t  (
7 0030  30 29 3b 0a 7d 0a
8        0  )  ;  \n  }  \n
```

[2]Listagem obtida através do comando `hd` do Linux, que apresenta o conteúdo de um arquivo em hexadecimal e seus caracteres ASCII correspondentes, byte por byte.

Por outro lado, o mesmo arquivo `hello.c` seria armazenado usando a seguinte sequência de bytes em um sistema DOS/Windows:

```
0000  69 6e 74 20 6d 61 69 6e 28 29 0d 0a 7b 0d 0a 20
       i  n  t  ␣  m  a  i  n  (  ) \r \n  {  \r \n  ␣
0010  20 70 72 69 6e 74 66 28 22 48 65 6c 6c 6f 2c 20
       ␣  p  r  i  n  t  f  (  "  H  e  l  l  o  ,  ␣
0020  77 6f 72 6c 64 5c 6e 22 29 3b 0d 0a 20 20 65 78
       w  o  r  l  d  \  n  "  )  ;  \r \n  ␣  ␣  e  x
0030  69 74 28 30 29 3b 0d 0a 7d 0d 0a
       i  t  (  0  )  ;  \r \n  }  \r \n
```

Essa diferença na forma de representação da separação entre linhas pode provocar problemas em arquivos de texto transferidos entre sistemas Windows e UNIX, caso não seja feita a devida conversão.

22.3.4 Arquivos de código

Em um sistema operacional moderno, um arquivo de código (programa executável ou biblioteca) é dividido internamente em várias seções, para conter código, tabelas de símbolos (variáveis e funções), listas de dependências (bibliotecas necessárias) e outras informações de configuração. A organização interna de um arquivo de código depende do sistema operacional para o qual foi definido. Os formatos de código mais usuais atualmente são [Levine, 2000]:

- **ELF** (*Executable and Linking Format*): formato de arquivo usado para programas executáveis e bibliotecas na maior parte das plataformas UNIX modernas. É composto por um cabeçalho e várias seções de dados, contendo código executável, tabelas de símbolos e informações sobre relocação de código, usadas quando o código executável é carregado na memória.
- **PE** (*Portable Executable*): é o formato usado para executáveis e bibliotecas na plataforma Windows. Consiste basicamente em uma extensão do formato COFF (*Common Object File Format*), usado em plataformas UNIX mais antigas.

A Figura 22.3 ilustra de forma simplificada a estrutura interna de um arquivo de código no formato ELF, usado tipicamente para armazenar código executável ou bibliotecas em sistemas UNIX (Linux, Solaris, etc.). Esse arquivo é composto por:

- *ELF header*: descreve o conteúdo do restante do arquivo.
- *Section header table*: descreve cada uma das seções do código (nome, tipo, tamanho, etc).
- *Sections*: trechos que compõem o arquivo, como código binário, constantes, tabela de símbolos, tabela de relocações, etc. As seções são agrupadas em segmentos.
- *Program header table*: informações sobre como o código deve ser carregado na memória ao lançar o processo ou carregar a biblioteca.

- *Segments*: conteúdo que deve ser carregado em cada segmento de memória ao lançar o processo; um segmento pode conter várias seções do programa ou biblioteca, para agilizar a carga do programa na memória.

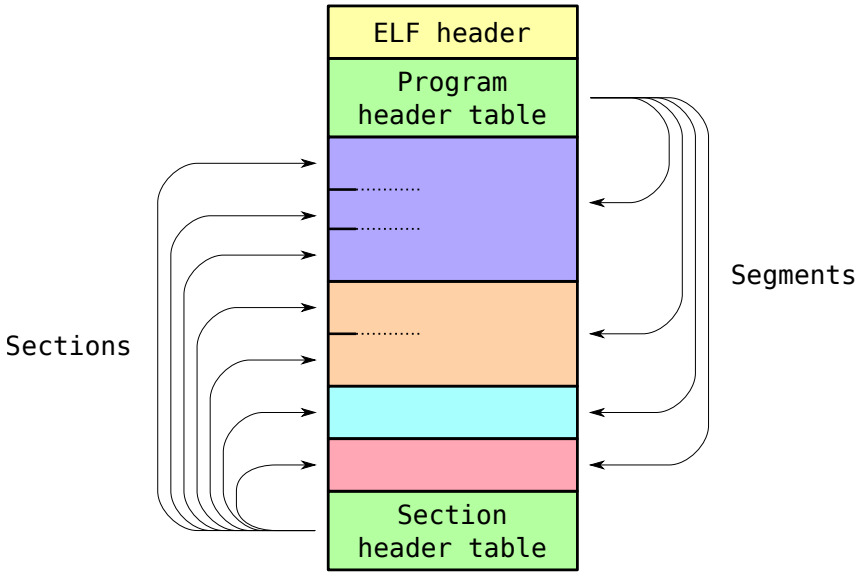


Figura 22.3: Estrutura interna de um arquivo em formato ELF [Levine, 2000].

22.3.5 Identificação de conteúdo

Um problema importante relacionado aos formatos de arquivos é a correta identificação de seu conteúdo pelos usuários e aplicações. Já que um arquivo de dados pode ser visto como uma simples sequência de bytes, como é possível reconhecer que tipo de informação essa sequência representa?

Uma solução simples para esse problema consiste usar parte do nome do arquivo para indicar o tipo do conteúdo: assim, um arquivo "`praia.jpg`" provavelmente contém uma imagem em formato JPEG, enquanto um arquivo "`entrevista.mp3`" contém áudio em formato MP3. A estratégia de **extensão do nome**, utilizada ainda hoje na maioria dos sistemas operacionais, foi introduzida nos anos 1980 pelo sistema operacional DOS. Naquele sistema, os arquivos eram nomeados usando um padrão denominado "8.3": até 8 caracteres para o nome, seguidos de um ponto (".") e de uma extensão com até 3 caracteres, para o tipo do conteúdo.

Outra abordagem, frequentemente usada em sistemas UNIX, é usar alguns bytes no início do conteúdo do arquivo para a definição de seu tipo. Esses bytes iniciais do conteúdo são denominados **números mágicos** (*magic numbers*), e são convencionados para muitos tipos de arquivos, como mostra a Tabela 22.1.

Nos sistemas UNIX, o utilitário `file` permite identificar o tipo de arquivo através da análise de seus bytes iniciais e do restante de sua estrutura interna, sem levar em conta o nome do arquivo. Por isso, constitui uma ferramenta importante para identificar arquivos sem extensão ou com extensão errada.

Além do uso de extensões no nome do arquivo e de números mágicos, alguns sistemas operacionais definem **atributos adicionais** no sistema de arquivos para identificar o conteúdo de cada arquivo. Por exemplo, o sistema operacional MacOS 9 define

Tabela 22.1: Números mágicos de alguns tipos de arquivos

Tipo de arquivo	bytes iniciais	Tipo de arquivo	bytes iniciais
Documento PostScript	`%!`	Documento PDF	`%PDF`
Imagem GIF	`GIF89a`	Imagem JPEG	`0xFF D8 FF`
Música MIDI	`MThd`	Classes Java (JAR)	`0xCA FE BA BE`
Arquivo ZIP	`0x50 4B 03 04`	Documento RTF	`{\rtf1`

um atributo com 4 bytes para identificar o tipo de cada arquivo (*file type*), e outro atributo com 4 bytes para indicar a aplicação que o criou (*creator application*). Os tipos de arquivos e aplicações são definidos em uma tabela mantida pelo fabricante do sistema. Assim, quando o usuário solicitar a abertura de um determinado arquivo, o sistema irá escolher a aplicação que o criou, se ela estiver presente. Caso contrário, pode indicar ao usuário uma relação de aplicações aptas a abrir aquele tipo de arquivo.

Recentemente, a necessidade de transferir arquivos através de e-mail e de páginas Web levou à definição de um padrão de tipagem de arquivos conhecido como **Tipos MIME** (da sigla *Multipurpose Internet Mail Extensions*) [Freed and Borenstein, 1996]. O padrão MIME define tipos de arquivos através de uma notação uniformizada na forma "`tipo/subtipo`". Alguns exemplos de tipos de arquivos definidos segundo o padrão MIME são apresentados na Tabela 22.2.

Tabela 22.2: Tipos MIME correspondentes a alguns formatos de arquivos

Tipo MIME	Significado
`application/java-archive`	Arquivo de classes Java (JAR)
`application/msword`	Documento do Microsoft Word
`application/vnd.oasis.opendocument.text`	Documento do OpenOffice
`audio/midi`	Áudio em formato MIDI
`audio/mpeg`	Áudio em formato MP3
`image/jpeg`	Imagem em formato JPEG
`image/png`	Imagem em formato PNG
`text/csv`	Texto em formato CSV (*Comma-separated Values*)
`text/html`	Texto HTML
`text/plain`	Texto puro
`text/rtf`	Texto em formato RTF (*Rich Text Format*)
`text/x-csrc`	Código-fonte em C
`video/quicktime`	Vídeo no formato *Quicktime*

O padrão MIME é usado para identificar arquivos transferidos como anexos de e-mail e conteúdos obtidos de páginas Web. Alguns sistemas operacionais, como o BeOS e o MacOS X, definem atributos adicionais usando esse padrão para identificar o conteúdo de cada arquivo dentro do sistema de arquivos.

22.4 Arquivos especiais

O conceito de arquivo é ao mesmo tempo simples e poderoso, o que motivou sua utilização de forma quase universal. Além do armazenamento de dados do sistema operacional e de aplicações, como mostrado na seção anterior, o conceito de arquivo também pode ser usado como:

Abstração de dispositivos de entrada/saída: os sistemas UNIX costumam mapear as interfaces de acesso a vários dispositivos físicos como arquivos dentro do diretório `/dev` (da palavra *devices*), como por exemplo:

- `/dev/ttyS0`: porta de comunicação serial (COM1);
- `/dev/audio`: placa de som;
- `/dev/sda1`: primeira partição do primeiro disco SCSI (ou SATA).

Essa abstração dos dispositivos é muito conveniente, pois permite que aplicações percebam e acessem os dispositivos físicos como se fossem arquivos. Por exemplo, uma aplicação que escrever dados no arquivo `/dev/ttyS0` estará enviando esses dados para a saída serial COM1 do computador.

Abstração de interfaces do núcleo: em sistemas UNIX, os diretórios `/proc` e `/sys` permitem consultar e/ou modificar informações internas do núcleo do sistema operacional, dos processos em execução e dos *drivers* de dispositivos. Por exemplo, alguns arquivos oferecidos pelo Linux:

- `/proc/cpuinfo`: informações sobre os processadores disponíveis no sistema;
- `/proc/3754/maps`: disposição das áreas de memória alocadas para o processo cujo identificador (PID) é 3754;
- `/sys/block/sda/queue/scheduler`: definição da política de escalonamento de disco (vide Capítulo 21.4) a ser usada no acesso ao disco `/dev/sda`.

Canais de comunicação: na família de protocolos de rede TCP/IP, a abstração de arquivo é usada como interface para os canais de comunicação: uma conexão TCP é apresentada aos dois processos comunicantes como um arquivo, sobre o qual eles podem escrever (enviar) e ler (receber) dados entre si. Vários mecanismos de comunicação local entre processos de um sistema também são vistos como arquivos, como é o caso dos *pipes* em UNIX.

Em alguns sistemas operacionais experimentais, como o *Plan 9* [Pike et al., 1993, 1995] e o *Inferno* [Dorward et al., 1997], todos os recursos e entidades físicas e lógicas do sistema operacional são mapeadas sob a forma de arquivos: processos, *threads*, conexões de rede, usuários, sessões de usuários, janelas gráficas, áreas de memória alocadas, etc. Assim, para finalizar um determinado processo, encerrar uma conexão de rede ou desconectar um usuário, basta remover o arquivo correspondente.

Embora o foco deste texto esteja concentrado em arquivos convencionais, que visam o armazenamento de informações (bytes ou registros), muitos dos conceitos aqui expostos são igualmente aplicáveis aos arquivos não convencionais descritos neste capítulo.

Referências

S. Dorward, R. Pike, D. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. The Inferno operating system. *Bell Labs Technical Journal*, 2(1):5–18, 1997.

N. Freed and N. Borenstein. RFC 2046: Multipurpose Internet Mail Extensions (MIME) part two: Media types, Nov 1996.

J. Levine. *Linkers and Loaders*. Morgan Kaufmann, 2000.

R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in Plan 9. *Operating Systems Review*, 27(2):72–76, April 1993.

R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Journal of Computing Systems*, 8(3):221–254, 1995.

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

Capítulo 23

Uso de arquivos

23.1 Introdução

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

23.2 Interface de acesso

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

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

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

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

Tabela 23.1: Chamadas de sistema para arquivos

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

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

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

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

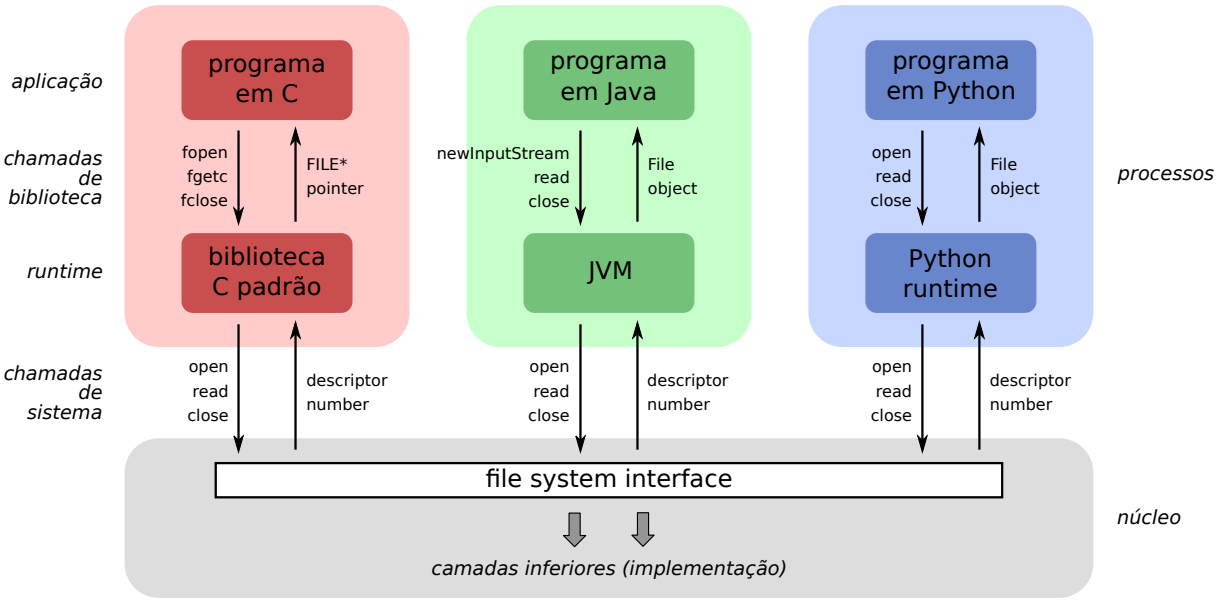


Figura 23.1: Relação entre funções de linguagem e chamadas de sistema.

23.2.1 Descritores de arquivos

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

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

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

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

23.2.2 A abertura de um arquivo

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

1. No processo:
   (a) a aplicação solicita a abertura do arquivo (`fopen()`, se for um programa C);
   (b) o suporte de execução da aplicação recebe a chamada de função, trata os parâmetros recebidos e invoca uma chamada de sistema para abrir o arquivo.
2. No núcleo:
   (a) o núcleo recebe a chamada de sistema;
   (b) localiza o arquivo no dispositivo físico, usando seu nome e caminho de acesso;
   (c) verifica se o processo tem as permissões necessárias para usar aquele arquivo da forma desejada (vide Seção 23.5);

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

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

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

3. No processo:

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

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

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

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

23.3 Formas de acesso

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

23.3.1 Acesso sequencial

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

A Figura 23.2 traz um exemplo de acesso sequencial em leitura a um arquivo, mostrando a evolução do ponteiro do arquivo durante uma sequência de leituras. Uma primeira leitura de 15 bytes do arquivo traz os caracteres "`Qui␣scribit␣bis`"; uma segunda leitura de mais 9 bytes traz "`␣legit.␣`", e assim sucessivamente. O acesso sequencial é implementado em praticamente todos os sistemas operacionais de mercado e constitui a forma mais usual de acesso a arquivos, usada pela maioria das aplicações.

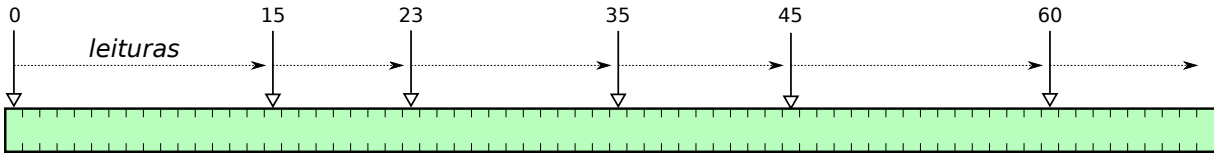


Figura 23.2: Leituras sequenciais em um arquivo de texto.

23.3.2 Acesso aleatório

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

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

23.3.3 Acesso mapeado em memória

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

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

23.3.4 Acesso indexado

Alguns sistemas operacionais oferecem também a possibilidade de **acesso indexado** aos dados de um arquivo, como é o caso do *OpenVMS* [Rice, 2000]. Esse sistema implementa arquivos cuja estrutura interna pode ser vista como uma tabela de pares *chave/valor*. Os dados do arquivo são armazenados em registros com chaves (índices) associados a eles, e podem ser recuperados usando essas chaves, como em um banco de dados relacional.

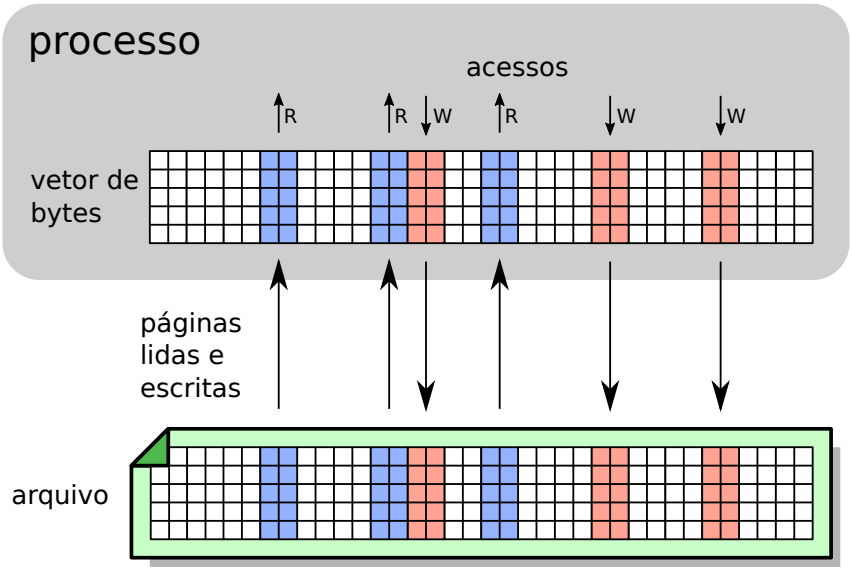


Figura 23.3: Arquivo mapeado em memória.

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

23.4 Compartilhamento de arquivos

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

23.4.1 Travas em arquivos

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

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

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

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

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

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

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

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

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

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

23.4.2 Semântica de acesso

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

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

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

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

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

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

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

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

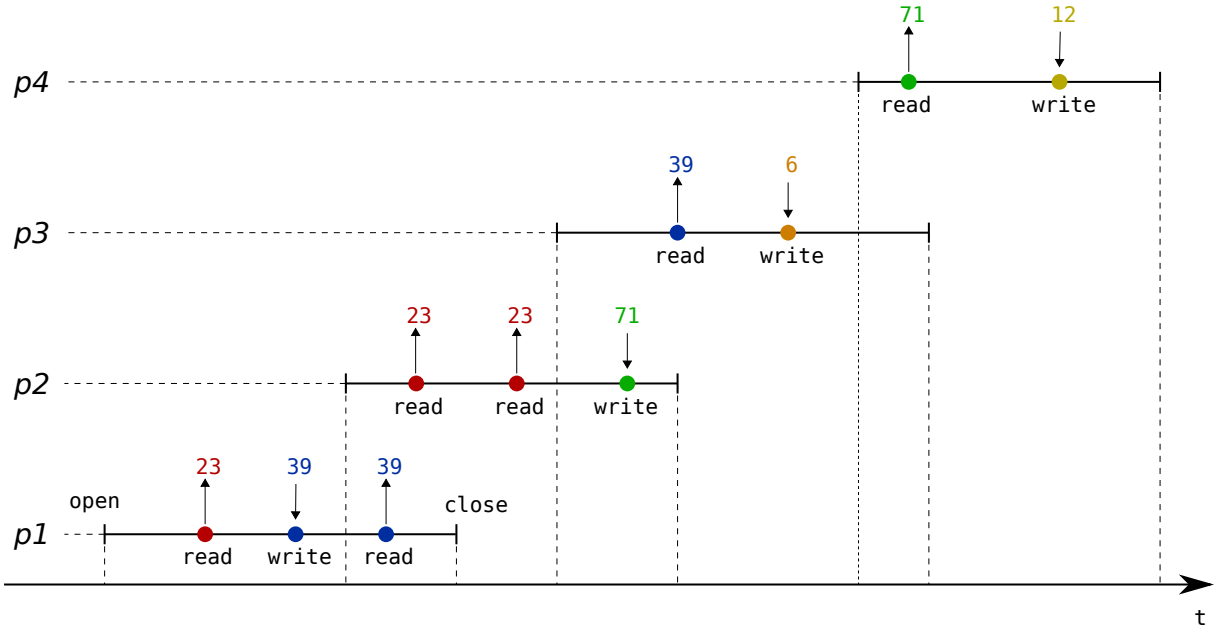


Figura 23.4: Compartilhamento de arquivo usando a semântica de sessão.

23.5 Controle de acesso

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

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

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

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

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

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

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

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

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

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

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

23.6 Interface de acesso

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

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

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

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

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

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

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

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

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

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

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

Referências

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

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

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

Capítulo 24

Sistemas de arquivos

24.1 Introdução

Vários problemas importantes devem ser resolvidos para a implementação eficiente de arquivos e diretórios, que vão dos aspectos de baixo nível, como o acesso aos dispositivos físicos, a aspectos mais abstratos, como a implementação da interface de acesso a arquivos para os programadores.

Neste capítulo são discutidos os principais elementos dos sistemas de arquivos, que são os subsistemas do sistema operacional que implementam o conceito de arquivo sobre o armazenamento bruto proporcionado pelos dispositivos físicos.

24.2 Arquitetura geral

Os principais elementos que realizam a implementação de arquivos no sistema operacional estão organizados em camadas, sendo apresentados na Figura 24.1 e detalhados a seguir:

Dispositivos: como discos rígidos ou bancos de memória *flash*, são os responsáveis pelo armazenamento de dados.

Controladores: são os circuitos eletrônicos dedicados ao controle dos dispositivos físicos. Eles são acessados através de portas de entrada/saída, interrupções e canais de acesso direto à memória (DMA).

Drivers: interagem com os controladores de dispositivos para configurá-los e realizar as transferências de dados entre o sistema operacional e os dispositivos. Como cada controlador define sua própria interface, também possui um *driver* específico. Os *drivers* ocultam as diferenças entre controladores e fornecem às camadas superiores do núcleo uma interface padronizada para acesso aos dispositivos de armazenamento.

Gerência de blocos: gerencia o fluxo de blocos de dados entre as camadas superiores e os dispositivos de armazenamento. Como os discos são dispositivos orientados a blocos, as operações de leitura e escrita de dados são sempre feitas com blocos de dados, e nunca com bytes individuais. As funções mais importantes desta camada são efetuar o mapeamento de blocos lógicos nos blocos físicos do dispositivo (Seção 24.4.1) e o *caching/buffering* de blocos (Seção 24.4.2).

Alocação de arquivos: realiza a alocação dos arquivos sobre os blocos lógicos oferecidos pela camada de gerência de blocos. Cada arquivo é visto como uma sequência de blocos lógicos que deve ser armazenada nos blocos dos dispositivos de forma eficiente, robusta e flexível. As principais técnicas de alocação de arquivos são discutidas na Seção 24.5.

Sistema de arquivos virtual: o VFS (*Virtual File System*) constrói as abstrações de diretórios e atalhos, além de gerenciar as permissões associadas aos arquivos e as travas de acesso compartilhado. Outra responsabilidade importante desta camada é manter o registro de cada arquivo aberto pelos processos, como a posição da última operação no arquivo, o modo de abertura usado e o número de processos que estão usando o arquivo.

Interface do sistema de arquivos: conjunto de chamadas de sistema oferecidas aos processos do espaço de usuários para a criação e manipulação de arquivos.

Bibliotecas de entrada/saída: usam as chamadas de sistema da interface do núcleo para construir funções padronizadas de acesso a arquivos para cada linguagem de programação (como aquelas apresentadas na Seção 23.6 para a linguagem C ANSI).

Na Figura 24.1, a maior parte da implementação do sistema de arquivos está localizada dentro do núcleo, mas isso obviamente varia de acordo com a arquitetura do sistema operacional. Em sistemas micronúcleo (Seção 3.2), por exemplo, as camadas de gerência de blocos e as superiores provavelmente estariam no espaço de usuário.

Na implementação de um sistema de arquivos, considera-se que cada arquivo possui **dados** e **metadados**. Os dados de um arquivo são o seu conteúdo em si (uma música, uma fotografia, um documento ou uma planilha); por outro lado, os metadados do arquivo são seus atributos (nome, datas, permissões de acesso, etc.) e todas as informações de controle necessárias para localizar e manter seu conteúdo no disco. Também são considerados metadados as informações necessárias à gestão do sistema de arquivos, como os mapas de blocos livres, etc.

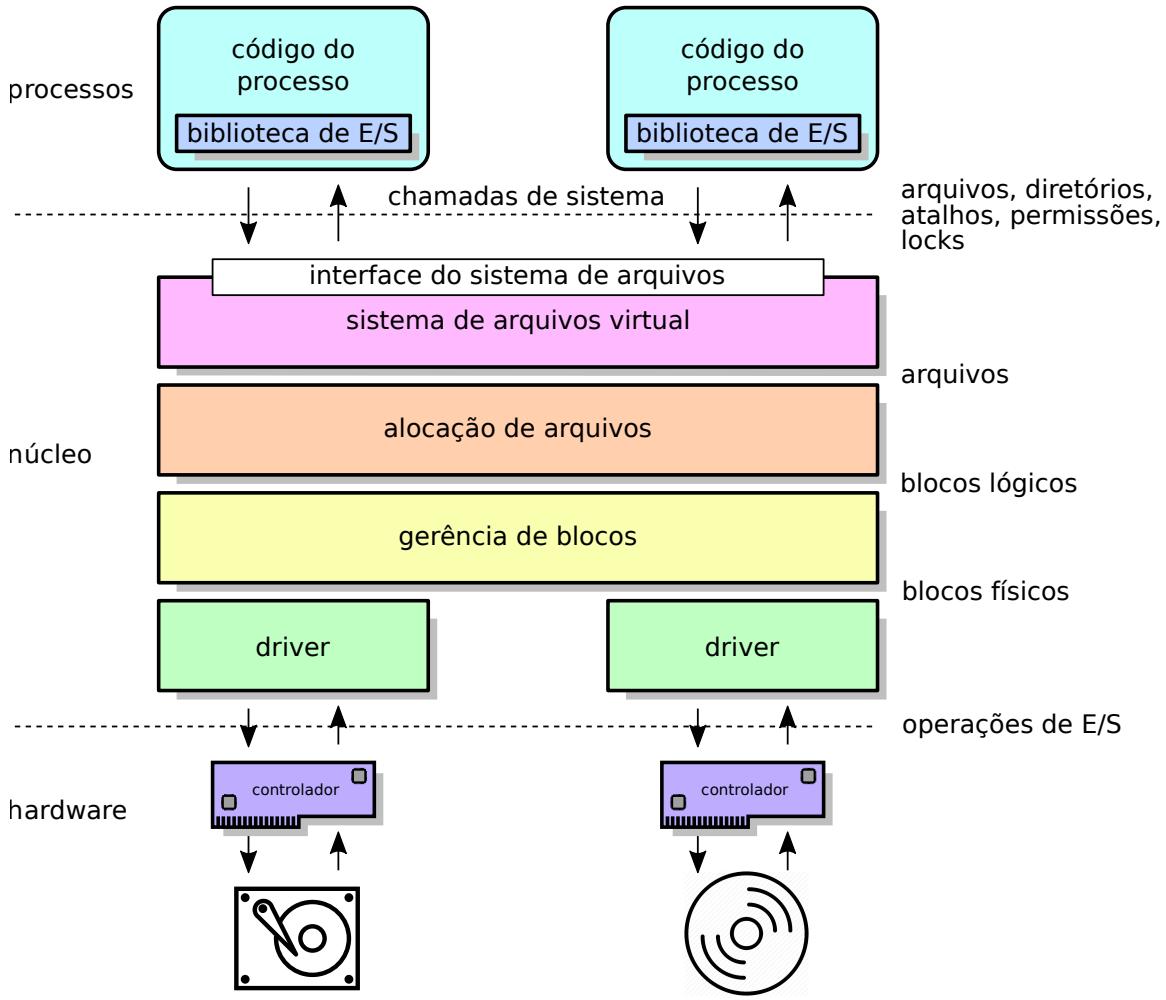


Figura 24.1: Camadas da implementação da gerência de arquivos.

24.3 Espaços de armazenamento

Um computador normalmente possui um ou mais dispositivos para armazenar arquivos, que podem ser discos rígidos, discos óticos (CD-ROM, DVD-ROM), discos de estado sólido (baseados em memória *flash*, como *pendrives* USB), etc. A estrutura física dos discos rígidos e demais dispositivos é discutida em detalhes no Capítulo 20.

24.3.1 Discos e partições

Em linhas gerais, um disco é visto pelo sistema operacional como um grande vetor de blocos de dados de tamanho fixo, numerados sequencialmente. As operações de leitura e escrita de dados nesses dispositivos são feitas bloco a bloco, por essa razão esses dispositivos são chamados *dispositivos de blocos* (*block devices* ou *block-oriented devices*).

O espaço de armazenamento de cada dispositivo é dividido em uma pequena área de configuração reservada, no início do disco, e uma ou mais *partições*, que podem ser vistas como espaços independentes. A área de configuração contém uma *tabela de partições* com informações sobre o particionamento do dispositivo (número do bloco inicial, quantidade de blocos e outras informações sobre cada partição). Além disso, essa

área contém também um pequeno código executável usado no processo de inicialização do sistema operacional (*boot*), por isso ela é usualmente chamada de *boot sector* ou MBR (*Master Boot Record*, nos PCs).

No início de cada partição do disco há também um ou mais blocos reservados, usados para a descrição do conteúdo daquela partição e para armazenar o código de lançamento do sistema operacional, se aquela for uma partição inicializável (*bootable partition*). Essa área reservada é denominada *bloco de inicialização* ou VBR - *Volume Boot Record*. O restante dos blocos da partição está disponível para o armazenamento de arquivos. A Figura 24.2 ilustra a organização básica do espaço de armazenamento em um disco rígido típico, com três partições.

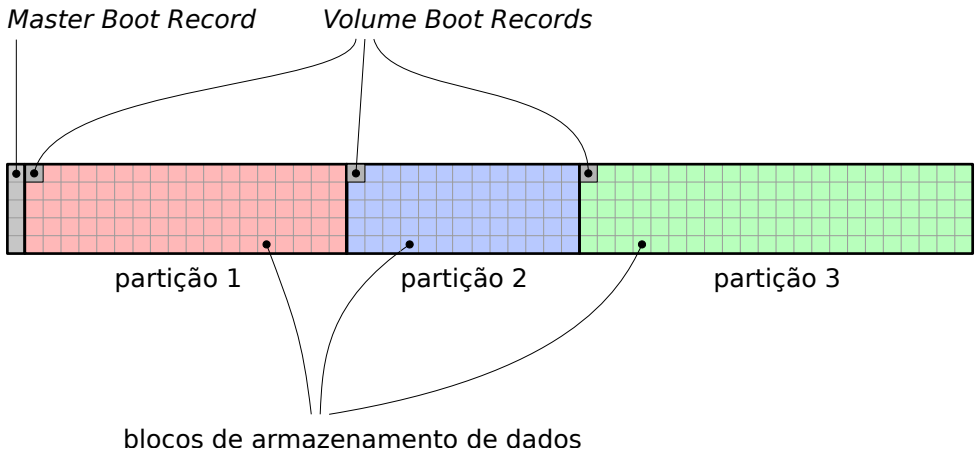


Figura 24.2: Organização em partições de um disco rígido.

É importante lembrar que existem diversos formatos possíveis para os blocos de inicialização de disco e de partição, além da estrutura da própria tabela de partição. Esses formatos devem ser reconhecidos pelo código de inicialização do computador (BIOS) e pelo sistema operacional instalado, para que os dados do disco possam ser acessados.

Um termo frequentemente utilizado em sistemas de arquivos é o **volume**, que significa um espaço de armazenamento de dados, do ponto de vista do sistema operacional. Em sua forma mais simples, cada volume corresponde a uma partição, mas configurações mais complexas são possíveis. Por exemplo, o subsistema LVM (*Logical Volume Manager*) do Linux permite construir volumes lógicos combinando vários discos físicos, como nos sistemas RAID (Seção 21.5).

Antes de ser usado, cada volume ou partição deve ser *formatado*, ou seja, preenchido com as estruturas de dados necessárias para armazenar arquivos, diretórios, atalhos e outras entradas. Cada volume pode ser formatado de forma independente e receber um sistema de arquivos distinto dos demais volumes.

24.3.2 Montagem de volumes

Para que o sistema operacional possa acessar os arquivos armazenados em um volume, ele deve ler os dados presentes em seu bloco de inicialização, que descrevem o tipo de sistema de arquivos do volume, e criar as estruturas em memória que representam esse volume dentro do núcleo do SO. Além disso, ele deve definir um identificador para o volume, de forma que os processos possam acessar seus arquivos. Esse procedimento

é denominado *montagem* do volume, e seu nome vem do tempo em que era necessário montar fisicamente os discos rígidos ou fitas magnéticas nos leitores, antes de poder acessar os dados. O procedimento oposto, a *desmontagem*, consiste em fechar todos os arquivos abertos no volume e remover as estruturas de memória usadas para gerenciá-lo.

Apesar de ser realizada para todos os volumes, a montagem é um procedimento particularmente frequente no caso de mídias removíveis, como CD-ROMs, DVD-ROMs e *pendrives* USB. Neste caso, a desmontagem do volume inclui também ejetar a mídia (CD, DVD) ou avisar o usuário que ela pode ser removida (discos USB).

Ao montar um volume, deve-se fornecer aos processos e usuários uma referência para seu acesso, denominada *ponto de montagem* (*mounting point*). Sistemas UNIX normalmente definem os pontos de montagem de volumes como posições dentro da árvore principal do sistema de arquivos. Dessa forma, há um volume principal, montado durante a inicialização do sistema operacional, onde normalmente reside o próprio sistema operacional e que define a estrutura básica da árvore de diretórios. Os volumes secundários são montados como subdiretórios na árvore do volume principal, através do comando `mount`.

A Figura 24.3 apresenta um exemplo de montagem de volumes em plataformas UNIX. Nessa figura, o disco SSD contém o sistema operacional e foi montado como raiz da árvore de diretórios, durante a inicialização do sistema. O disco rígido contém os diretórios de usuários e seu ponto de montagem é o diretório `/home`. Já o diretório `/media/cdrom` é o ponto de montagem de um CD-ROM, com sua árvore de diretórios própria, e o diretório `/media/backup` é o ponto de montagem de um *pendrive*.

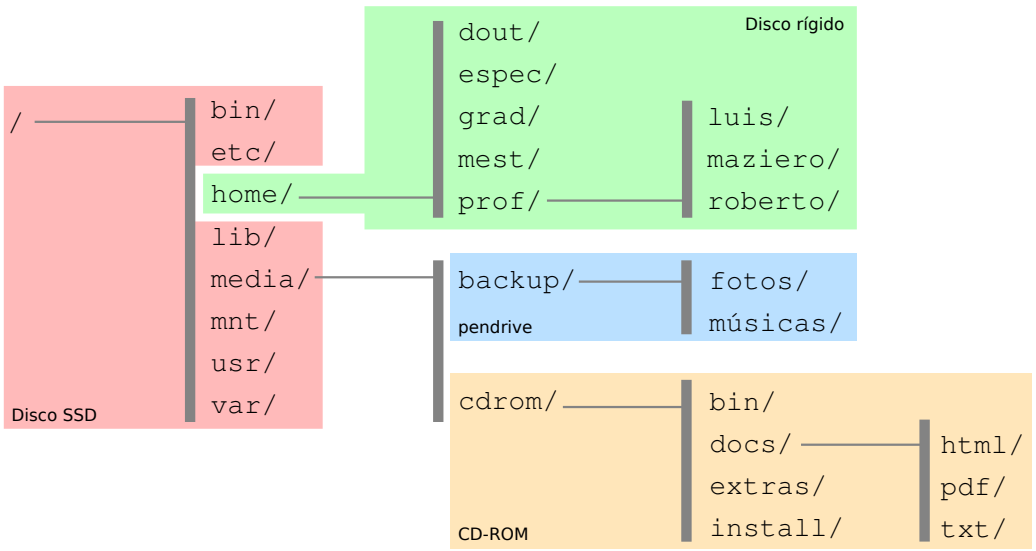


Figura 24.3: Montagem de volumes em UNIX.

Em sistemas de arquivos de outras plataformas, como DOS e Windows, é comum definir cada volume montado como um disco lógico distinto, chamado simplesmente de disco ou *drive* e identificado por uma letra ("`A:`", "`C:`", "`D:`", etc.). Todavia, o sistema de arquivos NTFS do Windows também permite a montagem de volumes como subdiretórios da árvore principal, da mesma forma que o UNIX.

24.4 Gestão de blocos

A função primordial da camada de gestão de blocos é interagir com os *drivers* de dispositivos para realizar as operações de leitura e escrita de blocos de dados. Neste nível do subsistema ainda não existe a noção de arquivo, que só será implementada nas camadas superiores. Portanto, todas as operações nesta camada dizem respeito a blocos de dados.

Além da interação com os *drivers*, esta camada também é responsável pelo mapeamento entre blocos físicos e blocos lógicos e pelo mecanismo de *caching* de blocos, abordados a seguir.

24.4.1 Blocos físicos e lógicos

Conforme visto na Seção 21.2, um disco rígido pode ser visto como um conjunto de blocos de tamanho fixo (geralmente de 512 ou 4.096 bytes). Os blocos do disco rígido são normalmente denominados *blocos físicos*. Como esses blocos são pequenos, seu número em um disco rígido recente pode ser imenso: um disco rígido de 500 GBytes contém mais de um bilhão de blocos físicos!

Para simplificar a gerência da imensa quantidade de blocos físicos e melhorar o desempenho das operações de leitura/escrita, os sistemas operacionais costumam agrupar os blocos físicos em *blocos lógicos* ou *clusters*, que são grupos de 2^n blocos físicos consecutivos. A maior parte das operações e estruturas de dados definidas nos discos pelos sistemas operacionais são baseadas em blocos lógicos, que também definem a unidade mínima de alocação de arquivos e diretórios: cada arquivo ou diretório ocupa um ou mais blocos lógicos para seu armazenamento.

O número de blocos físicos em cada bloco lógico é fixo e definido pelo sistema operacional ao formatar a partição, em função de vários parâmetros, como o tamanho da partição, o sistema de arquivos usado e o tamanho das páginas de memória RAM. Blocos lógicos com tamanhos de 4 K a 64 KBytes são frequentemente usados.

Blocos lógicos maiores (32 KB ou 64 KB) levam a uma menor quantidade de blocos lógicos a gerenciar pelo SO em cada disco e implicam em mais eficiência de entrada/saída, pois mais dados são transferidos em cada operação. Entretanto, blocos grandes podem gerar muita *fragmentação interna*. Por exemplo, um arquivo com 200 bytes de dados ocupará um bloco lógico inteiro. Se esse bloco lógico tiver 32 KBytes (32.768 bytes), serão desperdiçados 32.568 bytes, que ficarão alocados ao arquivo sem ser usados.

Pode-se concluir que blocos lógicos menores diminuiriam a perda de espaço útil por fragmentação interna. Todavia, usar blocos menores implica em ter mais blocos a gerenciar por disco e menos bytes transferidos em cada operação de leitura/escrita, o que tem impacto negativo sobre o desempenho do sistema.

A fragmentação interna diminui o espaço útil do disco, por isso deve ser evitada. Uma forma de tratar esse problema é escolher um tamanho de bloco lógico adequado ao tamanho médio dos arquivos a armazenar no disco, no momento de sua formatação. Alguns sistemas de arquivos (como o UFS do Solaris e o ReiserFS do Linux) permitem a alocação de partes de blocos lógicos, através de técnicas denominadas *fragmentos de blocos* ou *alocação de sub-blocos* [Vahalia, 1996].

24.4.2 *Caching* de blocos

Discos são dispositivos lentos, portanto as operações de leitura e escrita de blocos podem ter latências elevadas. O desempenho nos acessos ao disco pode ser melhorado através de um *cache*, ou seja, uma área de memória RAM na camada de gerência de blocos, onde o conteúdo dos blocos lidos/escritos pode ser mantido para acessos posteriores.

É possível fazer *caching* de leitura e de escrita. No *caching* de leitura (*read caching*), blocos lidos anteriormente do disco são mantidos em memória, para acelerar leituras subsequentes desses mesmos blocos. No *caching* de escrita (*write caching*, também chamado *buffering*), blocos a escrever no disco são mantidos em memória, para agrupar várias escritas pequenas em poucas escritas maiores (e mais eficientes) e para agilizar leituras posteriores desses dados.

Quatro estratégias básicas de *caching* são usuais (ilustradas na Figura 24.4):

***Read-through*:** quando um processo solicita uma leitura, o cache é consultado; caso o dado esteja no cache ele é entregue ao processo; caso contrário, o bloco é lido do disco, copiado no cache e entregue ao processo. Leituras subsequentes do mesmo bloco (pelo mesmo ou outro processo) poderão acessar o conteúdo diretamente do cache, sem precisar acessar o disco.

***Read-ahead*:** ao atender uma requisição de leitura, são trazidos para o cache mais dados que os solicitados pela requisição; além disso, leituras de dados ainda não solicitados podem ser agendadas em momentos de ociosidade dos discos. Dessa forma, futuras requisições podem ser beneficiadas pela leitura antecipada dos dados. Essa política pode melhorar muito o desempenho de acesso sequencial a arquivos e em outras situações com boa localidade de referências (Seção 15.9).

***Write-through*:** quando um processo solicita uma escrita, o conteúdo é escrito diretamente no disco, enquanto o processo solicitante aguarda a conclusão da operação. Uma cópia do conteúdo é mantida no cache, para beneficiar leituras futuras. Não há ganho de desempenho na operação de escrita. Esta estratégia também é denominada *escrita síncrona*.

***Write-back*:** (ou *write-behind*) quando um processo solicita uma escrita, os dados são copiados para o cache e o processo solicitante é liberado. A escrita efetiva dos dados no disco é efetuada posteriormente. Esta estratégia melhora o desempenho de escrita, pois libera mais cedo os processos que solicitam escritas (eles não precisam esperar pela escrita real dos dados no disco) e permite agrupar as operações de escrita, gerando menos acessos a disco. Todavia, existe o risco de perda de dados, caso ocorra um erro no sistema ou falta de energia antes que os dados sejam efetivamente transferidos do cache para o disco.

A estratégia de *caching* utilizada em cada operação pode ser definida pelas camadas superiores (alocação de arquivos, etc) e depende do tipo de informação a ser lida ou escrita. Usualmente, dados de arquivos são escritos usando a estratégia *write-back*, para obter um bom desempenho, enquanto metadados (estruturas de diretórios e outras informações de controle do sistema de arquivos) são escritas usando *write-through* para garantir mais confiabilidade (a perda de metadados tem muito mais impacto na confiabilidade do sistema que a perda de dados dentro de um arquivo específico).

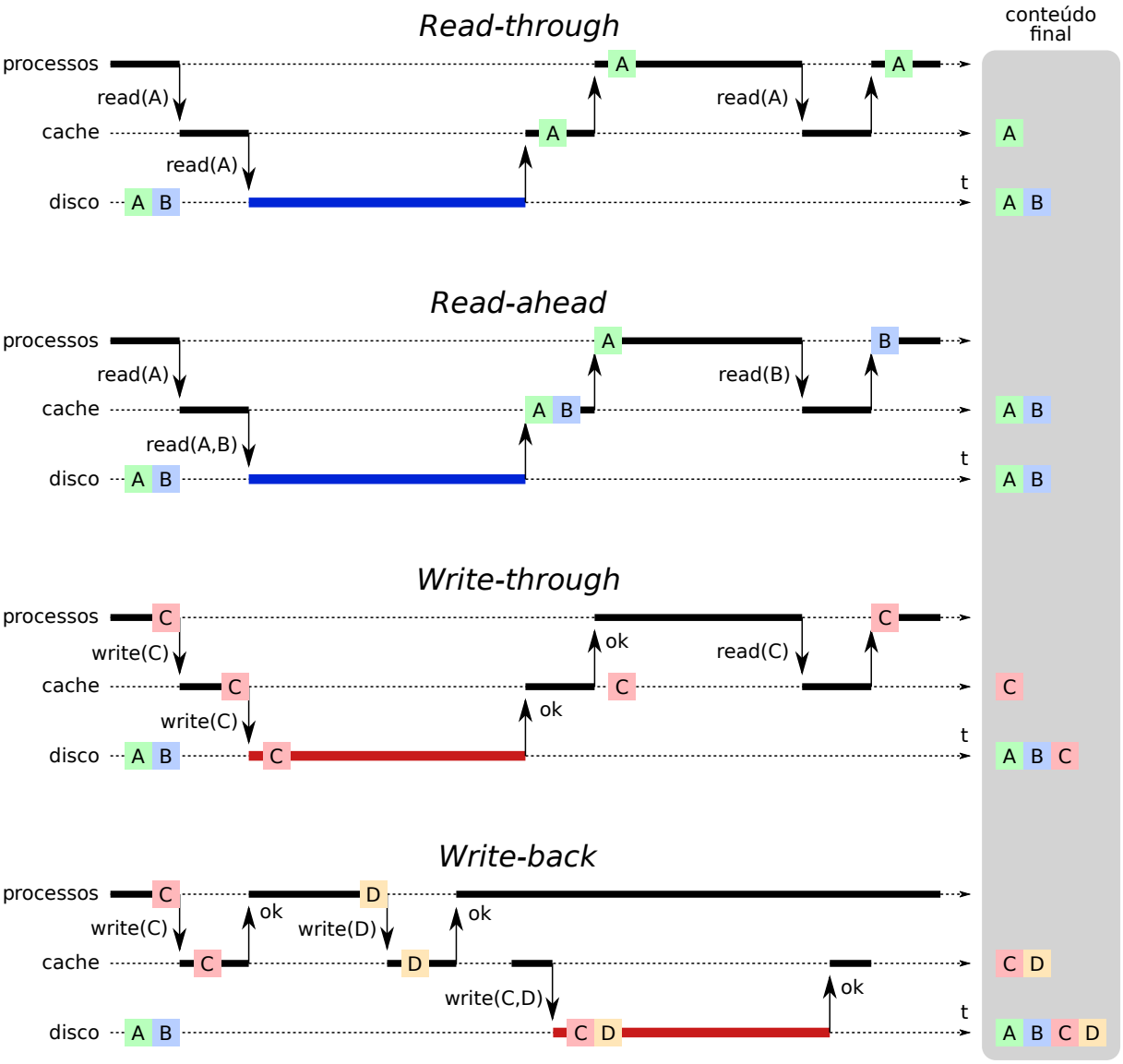


Figura 24.4: Estratégias de *caching* de blocos ($A \ldots D$ indicam blocos de disco).

Outro aspecto importante do *cache* é a gestão de seu tamanho e conteúdo. Obviamente, o *cache* de blocos reside em RAM e é menor que os discos subjacentes, portanto ele pode ficar cheio. Nesse caso, é necessário definir uma política para selecionar que conteúdo pode ser removido do *cache*. Políticas de substituição clássicas, como FIFO, LRU (*Least Recently Used*) e de segunda chance são frequentemente utilizadas.

24.5 Alocação de arquivos

Como visto nas seções anteriores, um espaço de armazenamento é visto pelas camadas superiores do sistema operacional como um grande vetor de blocos lógicos de tamanho fixo. O problema da alocação de arquivos consiste em dispor (alocar) o conteúdo e os metadados dos arquivos dentro desses blocos (Figura 24.5). Como os blocos são pequenos, um arquivo pode precisar de muitos blocos para ser armazenado no disco: por exemplo, um arquivo de filme em formato MP4 com 1 GB de tamanho ocuparia 262.144 blocos de 4 KBytes. O conteúdo do arquivo deve estar disposto nesses

blocos de forma a permitir um acesso rápido, flexível e confiável. Por isso, a forma de alocação dos arquivos nos blocos do disco tem um impacto importante sobre o desempenho e a robustez do sistema de arquivos.

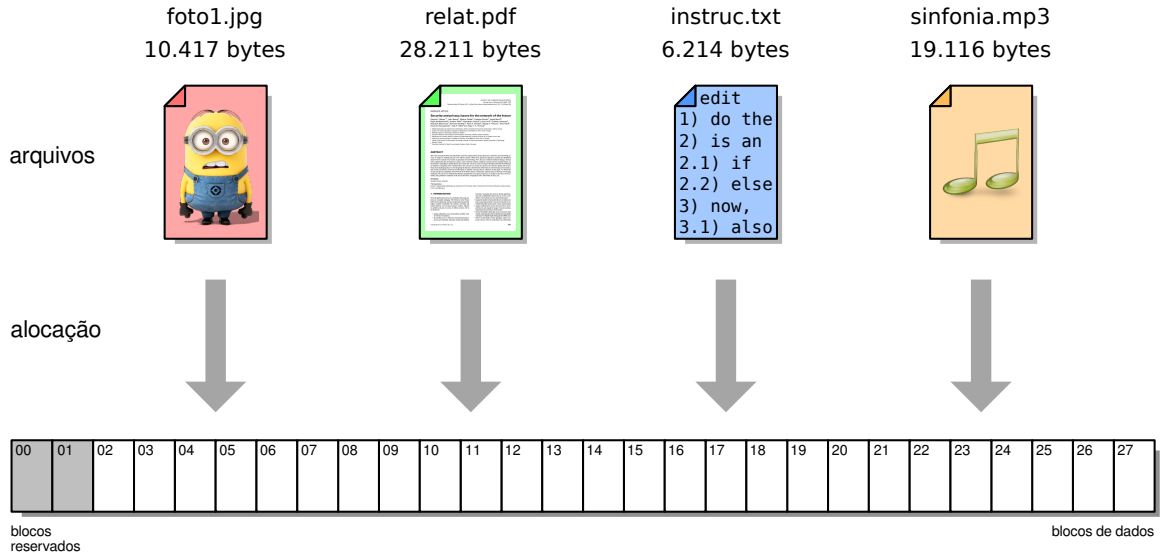


Figura 24.5: O problema da alocação de arquivos.

Há três estratégias básicas de alocação de arquivos nos blocos lógicos do disco, que serão apresentadas a seguir: as alocações **contígua**, **encadeada** e **indexada**. Essas estratégias serão descritas e analisadas à luz de três critérios: a **rapidez** oferecida por cada estratégia no acesso aos dados do arquivo, tanto para acessos sequenciais quanto para acessos aleatórios; a **robustez** de cada estratégia frente a erros, como blocos de disco defeituosos (*bad blocks*) e dados corrompidos; e a **flexibilidade** oferecida por cada estratégia para a criação, modificação e exclusão de arquivos.

Um conceito importante na alocação de arquivos é o **bloco de controle de arquivo** (FCB - *File Control Block*), que nada mais é que uma estrutura contendo os metadados do arquivo e uma referência para a localização de seu conteúdo no disco. A implementação do FCB depende do sistema de arquivos: em alguns pode ser uma simples entrada na tabela de diretório, enquanto em outros é uma estrutura de dados separada, como a *Master File Table* do sistema NTFS e os *i-nodes* dos sistemas UNIX.

24.5.1 Alocação contígua

Na alocação contígua, os dados do arquivo são dispostos de forma sequencial sobre um conjunto de blocos consecutivos no disco, sem "buracos" entre os blocos. Assim, a localização do conteúdo do arquivo no disco é definida pelo endereço de seu primeiro bloco. A Figura 24.6 apresenta um exemplo dessa estratégia de alocação (para simplificar o exemplo, considera-se que a tabela de diretórios contém os metadados de cada arquivo, como nome, tamanho em bytes e número do bloco inicial).

Como os blocos de cada arquivo se encontram em sequência no disco, o acesso sequencial aos dados do arquivo é rápido, por exigir pouca movimentação da cabeça de leitura do disco. O acesso aleatório também é rápido, pois a posição de cada byte do arquivo pode ser facilmente calculada a partir da posição do bloco inicial, conforme indica o algoritmo 3.

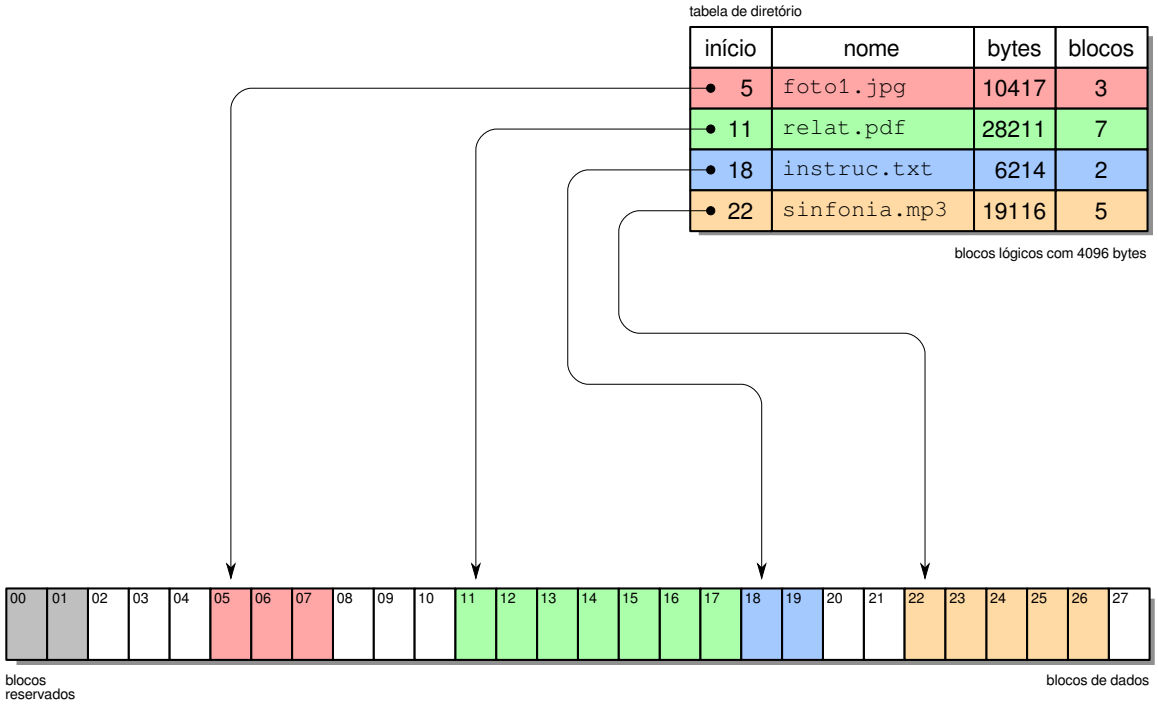


Figura 24.6: Estratégia de alocação contígua.

De acordo com esse algoritmo, o byte de número 14.372 do arquivo `relat.pdf` da Figura 24.6 estará na posição 2.084 do bloco 14 do disco rígido:

$$\begin{aligned} b_i &= b_0 + i \div B = 11 + 14.372 \div 4.096 = 11 + 3 = 14 \\ o_i &= i \bmod B = 14.372 \bmod 4.096 = 2.084 \end{aligned}$$

A estratégia de alocação contígua apresenta uma boa robustez a falhas de disco: caso um bloco do disco apresente defeito e não permita a leitura dos dados contidos nele, apenas o conteúdo daquele bloco é perdido: o conteúdo do arquivo nos blocos anteriores e posteriores ao bloco defeituoso ainda poderão ser acessados normalmente. Além disso, seu desempenho é muito bom, pois os blocos de cada arquivo estão próximos entre si no disco, minimizando a movimentação da cabeça de leitura do disco.

O ponto fraco desta estratégia é sua baixa flexibilidade, pois o tamanho máximo de cada arquivo precisa ser conhecido no momento de sua criação. No exemplo da Figura 24.6, o arquivo `relat.pdf` não pode aumentar de tamanho, pois não há blocos livres imediatamente após ele (o bloco 18 está ocupado). Para poder aumentar o tamanho desse arquivo, ele teria de ser movido (ou o arquivo `instruc.txt`) para liberar os blocos necessários.

Outro problema desta estratégia é a fragmentação externa, de forma similar à que ocorre nos mecanismos de alocação de memória (Capítulo 16): à medida em que arquivos são criados e destruídos, as áreas livres do disco vão sendo divididas em pequenas áreas isoladas (os fragmentos) que diminuem a capacidade de alocação de arquivos maiores. Por exemplo, na situação da Figura 24.6 há 9 blocos livres no disco, mas somente podem ser criados arquivos com até 3 blocos. As técnicas de alocação *first/best/worst-fit* utilizadas em gerência de memória também podem ser aplicadas para

Algoritmo 3 Localizar a posição do i-ésimo byte do arquivo no disco

Entrada:
i: número do byte a localizar no arquivo
B: tamanho dos blocos lógicos, em bytes
P: tamanho dos ponteiros de blocos, em bytes
b_0: número do bloco do disco onde o arquivo inicia

Saída:
b_i: número do bloco do disco onde se encontra o byte i
o_i: posição do byte i dentro do bloco b_i (*offset*)

$\div$: divisão inteira
mod: módulo (resto da divisão inteira)

$b_i = b_0 + i \div B$
$o_i = i \bmod B$
return(b_i, o_i)

atenuar este problema. Contudo, a desfragmentação de um disco é problemática, pois pode ser uma operação muito lenta e os arquivos não devem ser usados durante sua realização.

A baixa flexibilidade desta estratégia e a possibilidade de fragmentação externa limitam muito seu uso em sistemas operacionais de propósito geral, nos quais arquivos são constantemente criados, modificados e destruídos. Todavia, ela pode encontrar uso em situações específicas, nas quais os arquivos não sejam modificados constantemente e seja necessário rapidez nos acessos sequenciais e aleatórios aos dados. Um exemplo dessa situação são sistemas dedicados para reprodução de dados multimídia, como áudio e vídeo. O sistema ISO 9660, usado em CD-ROMs, é um exemplo de sistema de arquivos que usa a alocação contígua.

24.5.2 Alocação encadeada simples

Esta forma de alocação foi proposta para resolver os principais problemas da alocação contígua: sua baixa flexibilidade e a fragmentação externa. Na alocação encadeada, cada bloco do arquivo no disco contém dados do arquivo e também um ponteiro para o próximo bloco, ou seja, um campo indicando a posição no disco do próximo bloco do arquivo. Desta forma é construída uma lista encadeada de blocos para cada arquivo, não sendo mais necessário manter os blocos do arquivo lado a lado no disco. Esta estratégia elimina a fragmentação externa, pois todos os blocos livres do disco podem ser utilizados sem restrições, e permite que arquivos sejam criados sem a necessidade de definir seu tamanho final. A Figura 24.7 ilustra um exemplo dessa abordagem.

Nesta abordagem, o acesso sequencial aos dados do arquivo é simples e rápido, pois cada bloco do arquivo contém um "ponteiro" para o próximo bloco. Todavia, caso os blocos estejam muito espalhados no disco, a cabeça de leitura terá de fazer muitos deslocamentos, diminuindo o desempenho de acesso ao disco. Já o acesso aleatório ao arquivo fica muito prejudicado com esta abordagem: caso se deseje acessar o n-ésimo

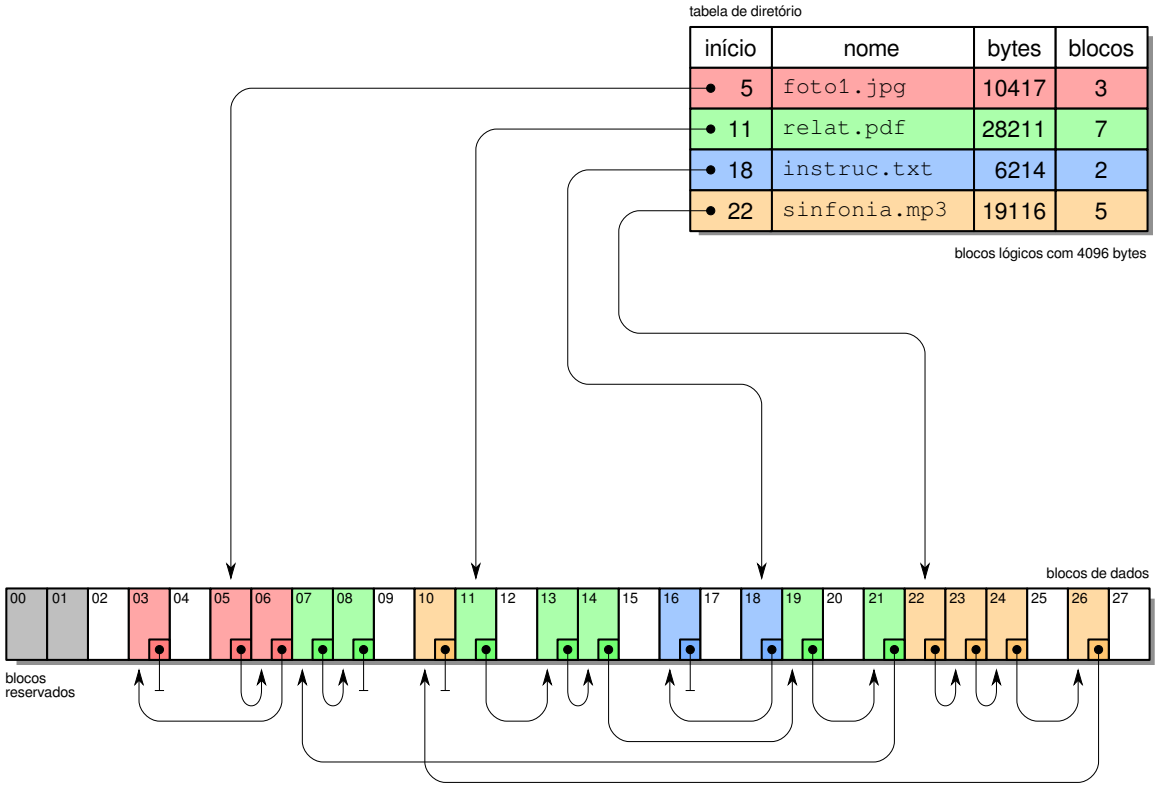


Figura 24.7: Estratégia de alocação encadeada simples.

bloco do arquivo, os $n-1$ blocos anteriores terão de ser lidos em sequência, para poder encontrar os ponteiros que levam ao bloco desejado. O algoritmo 4 mostra claramente esse problema, indicado através do laço *while*.

A dependência dos ponteiros de blocos também acarreta problemas de robustez: caso um bloco do arquivo seja corrompido ou se torne defeituoso, todos os blocos posteriores a este também ficarão inacessíveis. Por outro lado, esta abordagem é muito flexível, pois não há necessidade de se definir o tamanho máximo do arquivo durante sua criação, e arquivos podem ser expandidos ou reduzidos sem maiores dificuldades. Além disso, qualquer bloco livre do disco pode ser usados por qualquer arquivo, eliminando a fragmentação externa.

24.5.3 Alocação encadeada FAT

Os principais problemas da alocação encadeada simples são o baixo desempenho nos acessos aleatórios e a relativa fragilidade em relação a erros nos blocos do disco. Ambos os problemas provêm do fato de que os ponteiros dos blocos são armazenados nos próprios blocos, junto dos dados do arquivo. Para resolver esse problema, os ponteiros podem ser retirados dos blocos de dados e armazenados em uma tabela separada. Essa tabela é denominada **Tabela de Alocação de Arquivos** (FAT - *File Allocation Table*), sendo a base dos sistemas de arquivos FAT12, FAT16 e FAT32 usados nos sistemas operacionais MS-DOS, Windows e em muitos dispositivos de armazenamento portáteis, como *pendrives*, reprodutores MP3 e câmeras fotográficas digitais.

Na abordagem FAT, os ponteiros dos blocos de cada arquivo são mantidos em uma tabela única, armazenada em blocos reservados no início da partição. Cada entrada dessa tabela corresponde a um bloco lógico do disco e contém um ponteiro indicando o

Algoritmo 4 Localizar a posição do i-ésimo byte do arquivo no disco

Entrada:
i: número do byte a localizar no arquivo
B: tamanho dos blocos lógicos, em bytes
P: tamanho dos ponteiros de blocos, em bytes
b_0: número do primeiro bloco do arquivo no disco

Saída:
b_i: número do bloco do disco onde se encontra o byte i
o_i: posição do byte i dentro do bloco b_i (*offset*)

$b_{aux} = b_0$ ▷ define bloco inicial do percurso
$b = i \div (B - P)$ ▷ calcula número de blocos a percorrer

while $b > 0$ **do**
  $block = read\_block\ (b_{aux})$ ▷ lê um bloco do disco
  b_{aux} = ponteiro extraído de $block$
  $b = b - 1$
end while
$b_i = b_{aux}$
$o_i = i \bmod (B - P)$
return(b_i, o_i)

próximo bloco do mesmo arquivo. As entradas da tabela também podem conter valores especiais para indicar o último bloco de cada arquivo, blocos livres, blocos defeituosos e blocos reservados. Uma cópia dessa tabela é mantida em cache na memória durante o uso do sistema, para melhorar o desempenho na localização dos blocos dos arquivos. A Figura 24.8 apresenta o conteúdo da tabela de alocação de arquivos para o exemplo apresentado anteriormente na Figura 24.7.

A alocação com FAT resolve o problema de desempenho da alocação sequencial simples, mantendo sua flexibilidade de uso. A tabela FAT é um dado crítico para a robustez do sistema, pois o acesso aos arquivos ficará comprometido caso ela seja corrompida. Para minimizar esse problema, cópias da FAT são geralmente mantidas na área reservada.

24.5.4 Alocação indexada simples

Nesta abordagem, a estrutura em lista encadeada da estratégia anterior é substituída por um vetor contendo um *índice de blocos* do arquivo. Cada entrada desse índice corresponde a um bloco do arquivo e aponta para a posição desse bloco no disco. O índice de blocos de cada arquivo é mantido no disco em uma estrutura denominada *nó de índice* (*index node*) ou simplesmente *nó-i* (*i-node*). O *i-node* de cada arquivo contém, além de seu índice de blocos, os principais atributos do mesmo, como tamanho, permissões, datas de acesso, etc. Os *i-nodes* de todos os arquivos são agrupados em uma tabela de *i-nodes*, mantida em uma área reservada do disco, separada dos blocos de dados dos arquivos. A Figura 24.9 apresenta um exemplo de alocação indexada.

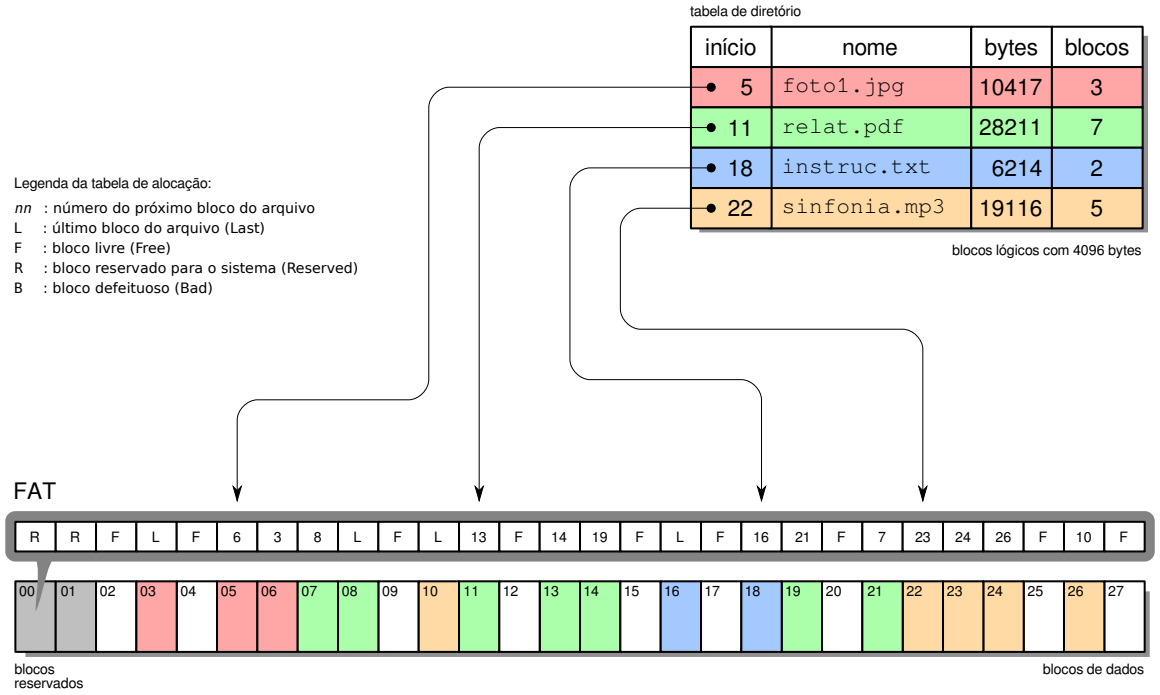


Figura 24.8: Estratégia de alocação encadeada com FAT.

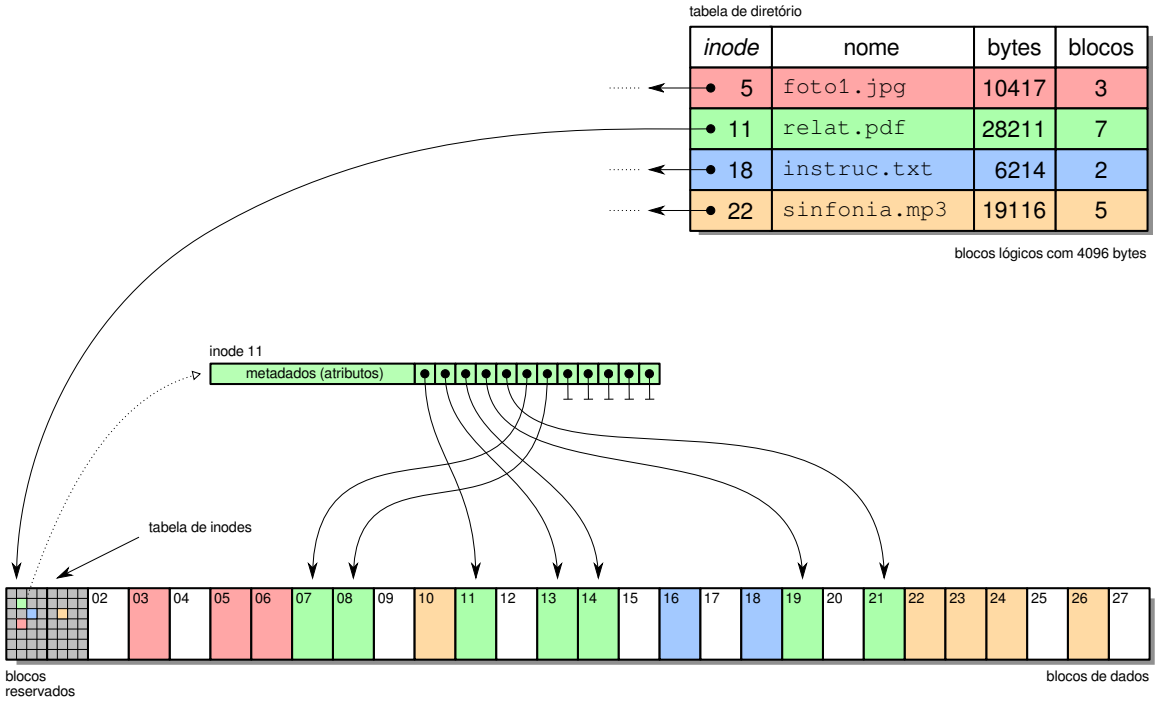


Figura 24.9: Estratégia de alocação indexada simples.

Como os *i-nodes* também têm tamanho fixo, o número de entradas no índice de blocos de um arquivo é limitado. Por isso, esta estratégia de alocação impõe um tamanho máximo para os arquivos. Por exemplo, se o sistema usar blocos de 4 KBytes e o índice de blocos suportar 64 ponteiros, só poderão ser armazenados arquivos com até 256 KBytes (64×4). Além disso, a tabela de *i-nodes* também tem um tamanho fixo, determinado durante a formatação do sistema de arquivos, o que limita o número

máximo de arquivos ou diretórios que podem ser criados na partição (pois cada arquivo ou diretório consome um *i-node*).

24.5.5 Alocação indexada multinível

Para aumentar o tamanho máximo dos arquivos armazenados, algumas das entradas do índice de blocos podem ser transformadas em ponteiros indiretos. Essas entradas apontam para blocos do disco que contém outros ponteiros, criando assim uma estrutura em árvore. Considerando um sistema com blocos lógicos de 4 Kbytes e ponteiros de 32 bits (4 bytes), cada bloco lógico pode conter 1.024 ponteiros, o que aumenta muito a capacidade do índice de blocos. Além de ponteiros indiretos, podem ser usados ponteiros dupla e triplamente indiretos. Por exemplo, os sistemas de arquivos Ext2/Ext3 do Linux (apresentado na Figura 24.10) usam *i-nodes* com 12 ponteiros diretos (que apontam para blocos de dados), um ponteiro indireto, um ponteiro duplamente indireto e um ponteiro triplamente indireto.

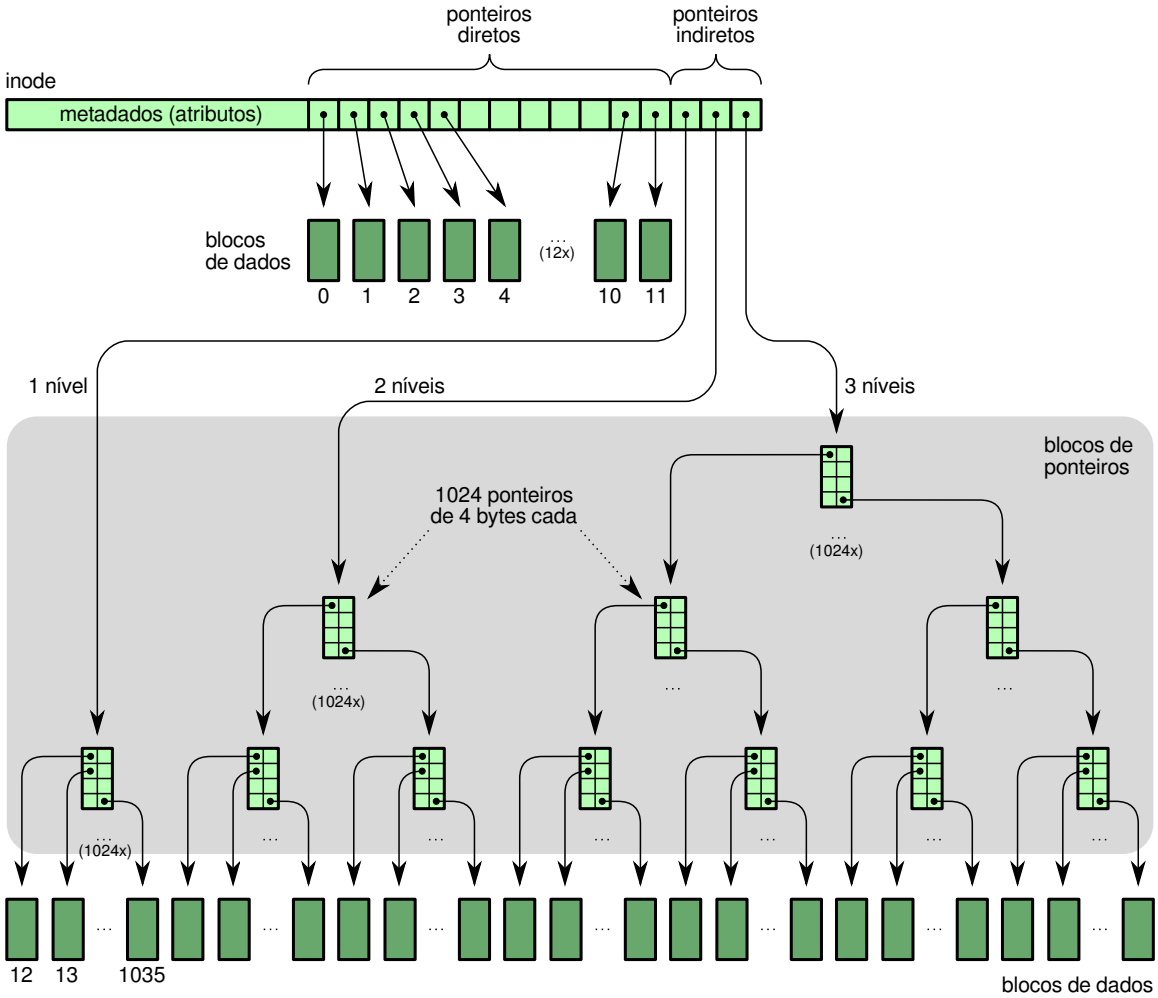


Figura 24.10: Estratégia de alocação indexada multi-nível.

Considerando blocos lógicos de 4 Kbytes e ponteiros de 4 bytes, cada bloco de ponteiros contém 1.024 ponteiros. Dessa forma, o cálculo do tamanho máximo de um arquivo nesse sistema é simples:

$$
\begin{aligned}
max &= 4.096 \times 12 && \text{(ponteiros diretos)} \\
&+ 4.096 \times 1.024 && \text{(ponteiro 1-indireto)} \\
&+ 4.096 \times 1.024 \times 1.024 && \text{(ponteiro 2-indireto)} \\
&+ 4.096 \times 1.024 \times 1.024 \times 1.024 && \text{(ponteiro 3-indireto)} \\
&= 4.402.345.721.856 \textit{ bytes} \\
max &\approx 4T \textit{ bytes}
\end{aligned}
$$

Apesar dessa estrutura aparentemente complexa, a localização e acesso de um bloco do arquivo no disco permanece relativamente simples, pois a estrutura homogênea de ponteiros permite calcular rapidamente a localização dos blocos. A localização do bloco lógico de disco correspondente ao *i*-ésimo bloco lógico de um arquivo segue o algoritmo 5.

Em relação ao desempenho, pode-se afirmar que esta estratégia é bastante rápida, tanto para acessos sequenciais quanto para acessos aleatórios a blocos, devido aos índices de ponteiros dos blocos presentes nos *i-nodes*. Contudo, no caso de blocos situados no final de arquivos muito grandes, podem ser necessários três ou quatro acessos a disco adicionais para localizar o bloco desejado, devido à necessidade de ler os blocos com ponteiros indiretos.

Defeitos em blocos de dados não afetam os demais blocos de dados, o que torna esta estratégia robusta. Todavia, defeitos nos metadados (o *i-node* ou os blocos de ponteiros) podem danificar grandes extensões do arquivo; por isso, muitos sistemas que usam esta estratégia implementam técnicas de redundância de *i-nodes* e metadados para melhorar a robustez.

Em relação à flexibilidade, pode-se afirmar que esta forma de alocação é tão flexível quanto a alocação encadeada, não apresentando fragmentação externa e permitindo o uso de todas as áreas livres do disco para armazenar dados. Todavia, são impostos limites para o tamanho máximo dos arquivos criados e para o número máximo de arquivos no volume.

Algoritmo 5 Localizar a posição do i-ésimo byte do arquivo no disco

Entrada:
i: número do byte a localizar no arquivo
B: tamanho dos blocos lógicos, em bytes (4.096 neste exemplo)
ptr[0...14]: vetor de ponteiros contido no *i-node*
$block$[0...1023]: bloco de ponteiros para outros blocos (1.024 ponteiros de 4 bytes)

Saída:
b_i: número do bloco do disco onde se encontra o byte i
o_i: posição do byte i dentro do bloco b_i (*offset*)

$o_i = i \bmod B$
$pos = i \div B$
if $pos < 12$ **then** ▷ usar ponteiros diretos
    $b_i = ptr[pos]$ ▷ o ponteiro é o número do bloco b_i
else
    $pos = pos - 12$
    if $pos < 1024$ **then** ▷ usar ponteiro 1-indireto
        $block_1 = read\_block\ (ptr[12])$ ▷ ler 1º bloco de ponteiros
        $b_i = block_1[pos]$ ▷ achar endereço do bloco b_i
    else
        $pos = pos - 1024$
        if $pos < 1024^2$ **then** ▷ usar ponteiro 2-indireto
            $block_1 = read\_block\ (ptr[13])$ ▷ ler 1º bloco de ponteiros
            $block_2 = read\_block\ (block_1[pos \div 1024])$ ▷ ler 2º bloco de ponteiros
            $b_i = block_2[pos \bmod 1024]$ ▷ achar endereço do bloco b_i
        else ▷ usar ponteiro 3-indireto
            $pos = pos - 1024^2$
            $block_1 = read\_block\ (ptr[14])$ ▷ ler 1º bloco de ponteiros
            $block_2 = read\_block\ (block_1[pos \div (1024^2)])$ ▷ ler 2º bloco
            $block_3 = read\_block\ (block_2[(pos \div 1024) \bmod 1024])$ ▷ ler 3º bloco
            $b_i = block_3[pos \bmod 1024]$ ▷ achar endereço do bloco b_i
        end if
    end if
end if
return(b_i, o_i)

24.5.6 Análise comparativa

A Tabela 24.1 traz um comparativo entre as principais formas de alocação estudadas aqui, sob a ótica de suas características de rapidez, robustez e flexibilidade de uso.

Estratégia	**Contígua**	**Encadeada**	**FAT**	**Indexada**
Rapidez (acesso sequencial)	Alta, os blocos do arquivo estão sempre em sequência no disco.	Alta, se os blocos do arquivo estiverem próximos no disco.	Alta, se os blocos do arquivo estiverem próximos no disco.	Alta, se os blocos do arquivo estiverem próximos no disco.
Rapidez (acesso aleatório)	Alta, as posições dos blocos podem ser calculadas sem acessar o disco.	Baixa, é necessário ler todos os blocos a partir do início do arquivo até encontrar o bloco desejado.	Alta, se os blocos do arquivo estiverem próximos no disco.	Alta, se os blocos do arquivo estiverem próximos no disco.
Robustez	Alta, blocos com erro não impedem o acesso aos demais blocos do arquivo.	Baixa: erro em um bloco leva à perda dos dados daquele bloco e de todos os blocos subsequentes do arquivo.	Alta, desde que não ocorram erros na tabela de alocação.	Alta, desde que não ocorram erros no *i-node* nem nos blocos de ponteiros.
Flexibilidade	Baixa, o tamanho máximo dos arquivos deve ser conhecido a priori; nem sempre é possível aumentar o tamanho de um arquivo existente.	Alta, arquivos podem ser criados em qualquer local do disco, sem risco de fragmentação externa.	Alta, arquivos podem ser criados em qualquer local do disco, sem risco de fragmentação externa.	Alta, arquivos podem ser criados em qualquer local do disco, sem risco de fragmentação externa.
Limites	O tamanho de um arquivo é limitado ao tamanho do disco.	O número de bits do ponteiro limita o número de blocos endereçáveis e o tamanho do arquivo.	O número de bits do ponteiro limita o número de blocos endereçáveis e o tamanho do arquivo.	Número de ponteiros no *i-node* limita o tamanho do arquivo; tamanho da tabela de *i-nodes* limita número de arquivos.

Tabela 24.1: Quadro comparativo das estratégias de alocação de arquivos.

24.6 Gestão do espaço livre

Além de manter informações sobre que blocos são usados por cada arquivo no disco, a camada de alocação de arquivos deve manter um registro atualizado de quais blocos estão livres, ou seja não estão ocupados por nenhum arquivo ou metadado. Isto é importante para obter rapidamente blocos no momento de criar um novo arquivo ou aumentar um arquivo existente. Algumas técnicas de gerência de blocos são sugeridas na literatura [Silberschatz et al., 2001; Tanenbaum, 2003]: o mapa de bits, a lista de blocos livres e a tabela de grupos de blocos livres.

Na abordagem de **mapa de bits**, um pequeno conjunto de blocos na área reservada do volume é usado para manter um mapa de bits. Nesse mapa, cada bit representa um bloco lógico da partição, que pode estar livre ou ocupado. Para o exemplo da Figura 24.9, o mapa de bits de blocos livres seria: 1101 0111 1011 0110 1011 0111 1010 (considerando 0 para bloco livre e incluindo os blocos reservados no início da partição). O mapa de bits tem como vantagens ser simples de implementar e ser bem compacto: em um disco de 500 GBytes com blocos lógicos de 4.096 bytes, seriam necessários 131.072.000 bits no mapa, o que representa 16.384.000 bytes, ocupando 4.000 blocos (ou seja, 0,003% do total de blocos lógicos do disco).

Na abordagem de **lista de blocos livres**, cada bloco livre contém um ponteiro para o próximo bloco livre do disco, de forma similar à alocação encadeada de arquivos vista na Seção 24.5.2. Essa abordagem é ineficiente, por exigir um acesso a disco para cada bloco livre requisitado. Uma melhoria simples consiste em armazenar em cada bloco livre um vetor de ponteiros para outros blocos livres; o último ponteiro desse vetor apontaria para um novo bloco livre contendo mais um vetor de ponteiros, e assim sucessivamente. Essa abordagem permite obter um grande número de blocos livre a cada acesso a disco.

Outra melhoria similar consiste em manter uma **tabela de grupos de blocos livres**, contendo a localização e o tamanho de um conjunto de blocos livres contíguos no disco. Cada entrada dessa tabela contém o número do bloco inicial e o número de blocos no grupo, de forma similar à alocação contígua de arquivos (Seção 24.5.1). Para o exemplo da figura 24.6, a tabela de grupos de blocos livres teria o seguinte conteúdo: $\{[2,3],[8,3],[20,2],[27,1]\}$, com entradas na forma *[bloco, tamanho]*.

Por outro lado, a abordagem de alocação FAT (Seção 24.5.3) usa a própria tabela de alocação de arquivos para gerenciar os blocos livres, que são indicados por flags específicos; no exemplo da Figura 24.8, os blocos livres estão indicados com o flag "`F`" na tabela. Para encontrar blocos livres ou liberar blocos usados, basta consultar ou modificar as entradas da tabela.

É importante lembrar que, além de manter o registro dos blocos livres, na alocação indexada também é necessário gerenciar o uso dos *inodes*, ou seja, manter o registros de quais *inodes* estão livres. Isso geralmente também é feito através de um mapa de bits.

Referências

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

A. Tanenbaum. *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall, 2003.

U. Vahalia. *UNIX Internals – The New Frontiers*. Prentice-Hall, 1996.

Capítulo 25

Diretórios e atalhos

A quantidade de arquivos em um sistema atual pode ser muito grande, chegando facilmente a milhões deles em um computador *desktop* típico, e muito mais em servidores. O sistema operacional pode tratar facilmente essa imensa quantidade de arquivos, mas essa tarefa não é tão simples para os usuários. Para simplificar a gestão dos arquivos é possível organizá-los em grupos e hierarquias, usando diretórios e atalhos, cujo conceito e implementação são explicados neste capítulo.

Além de arquivos, um sistema de arquivos também precisa implementar outros elementos, como diretórios e atalhos, e precisa construir mecanismos eficientes para permitir a localização de arquivos nas árvores de diretórios, que podem ser imensas. Esta seção apresenta as linhas gerais da implementação de diretórios, de atalhos e do mecanismo de localização de arquivos nos diretórios.

25.1 Diretórios

Um diretório, também chamado de *pasta* ou *folder*, representa um contêiner de arquivos e de outros diretórios. Da mesma forma que os arquivos, os diretórios têm nome e atributos, que são usados na localização e acesso ao seu conteúdo.

Cada sistema de arquivos possui um diretório principal, denominado *diretório raiz* (*root directory*). Os primeiros sistemas de arquivos implementavam apenas o diretório raiz, que continha todos os arquivos. Posteriormente foram implementados subdiretórios, ou seja, um nível de diretórios abaixo do diretório raiz. Os sistemas de arquivos atuais oferecem uma estrutura muito mais flexível, com um número de níveis de diretórios muito mais elevado, ou mesmo ilimitado (como nos sistemas de arquivos NTFS e Ext4).

O uso de diretórios permite construir uma estrutura hierárquica (em árvore) de armazenamento dentro de um volume, sobre a qual os arquivos são organizados. A Figura 25.1 representa uma pequena parte da árvore de diretórios típica de sistemas Linux, cuja estrutura é definida nas normas *Filesystem Hierarchy Standard* (FHS) [Russell et al., 2004].

A maioria dos sistemas operacionais implementa o conceito de *diretório de trabalho* ou diretório corrente de um processo (*working directory*): ao ser criado, cada novo processo é associado a um diretório, que será usado por ele como local default para criar novos arquivos ou abrir arquivos existentes, quando não informar os respectivos caminhos de acesso. Cada processo geralmente herda o diretório de trabalho de seu

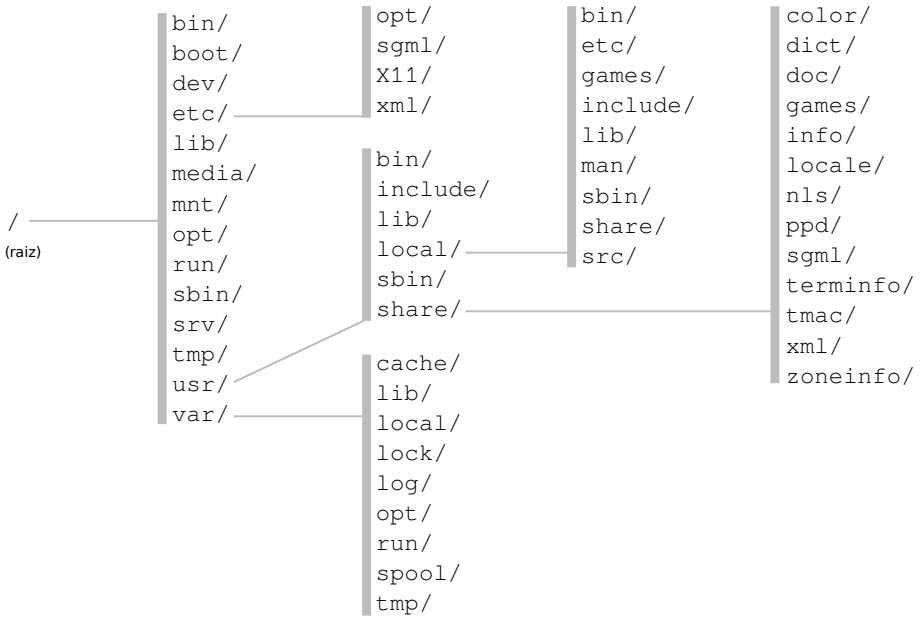


Figura 25.1: Estrutura de diretórios FHS de um sistema Linux.

pai, mas pode mudar de diretório através de chamadas de sistema (como `chdir` nos sistemas UNIX).

25.2 Caminhos de acesso

Em um sistema de arquivos, os arquivos estão dispersos ao longo da hierarquia de diretórios. Para poder abrir e acessar um arquivo, torna-se então necessário conhecer sua localização completa, ao invés de somente seu nome. A posição de um arquivo dentro do sistema de arquivos é chamada de **caminho de acesso** ao arquivo. Normalmente, o caminho de acesso a um arquivo é composto pela sequência de nomes de diretórios que levam até ele a partir da raiz, separados por um caractere específico. Cada elemento do caminho representa um nível de diretório, a partir da raiz. Por exemplo, considerando a estrutura de diretórios da figura 25.2, o arquivo `index.html` teria o seguinte caminho de acesso: `C:\Users\Maziero\public_html\index.html`.

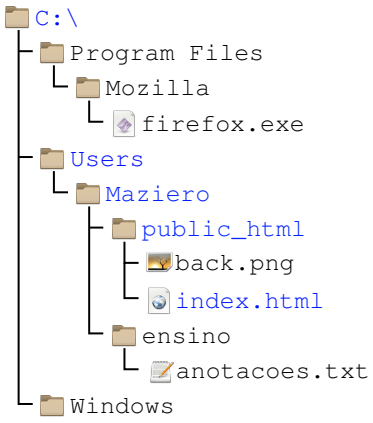


Figura 25.2: Caminho de acesso em um sistema Windows

O caractere separador de nomes no caminho depende do sistema operacional. Por exemplo, o sistema Windows usa como separador o caractere "\", enquanto sistemas UNIX usam o caractere "/"; outros sistemas podem usar caracteres como "`:`" e "`!`". Exemplos de caminhos de acesso a arquivos em sistemas UNIX seriam `/bin/bash` e `/var/log/mail.log`.

Em muitos sistemas de arquivos, um caminho de acesso pode conter elementos especiais, como "`..`" e "`.`". O elemento "`..`" indica o diretório anterior, ou seja, retorna um nível na hierarquia; por sua vez, o elemento "`.`" indica o próprio diretório atual. Esses elementos são úteis na construção de referências de acesso a arquivos fora do diretório corrente do processo.

Para acessar um arquivo, o processo precisa fornecer uma referência a ele. Existem basicamente três formas de se referenciar arquivos armazenados em um sistema de arquivos:

Referência direta: somente o nome do arquivo é informado pelo processo; neste caso, considera-se que o arquivo está (ou será criado) no diretório de trabalho do processo. Exemplos:

```
1 firefox.exe
2 back.png
3 index.html
```

Em um sistema UNIX, se o processo estiver trabalhando no diretório `/home/prof/maziero`, então o arquivo `materiais.pdf` pode ser localizado pelo sistema operacional através do caminho `/home/prof/maziero/materiais.pdf`.

Referência absoluta: o caminho de acesso ao arquivo é indicado a partir do diretório raiz do sistema de arquivos, e não depende do diretório de trabalho do processo; uma referência absoluta a um arquivo sempre inicia com o caractere separador, indicando que o nome do arquivo está referenciado a partir do diretório raiz do sistema de arquivos. Exemplos de referências absolutas:

```
C:\Users\Maziero\ensino\anotacoes.txt
/usr/local/share/fortunes/brasil.dat
C:\Program Files\Mozilla\firefox.exe
/home/maziero/bin/scripts/../../docs/proj1.pdf
```

O caminho de acesso mais curto a um arquivo a partir do diretório raiz é denominado *caminho canônico* do arquivo. Nos exemplos de referências absolutas acima, os dois primeiros são caminhos canônicos, enquanto os dois últimos não.

Referência relativa: o caminho de acesso ao arquivo tem como início o diretório de trabalho do processo, e indica subdiretórios ou diretórios anteriores, através de elementos "`..`". Eis alguns exemplos:

```
Mozilla\firefox.exe
public_html\index.html
public_html/static/fotografias/rennes.jpg
../../../share/icons/128x128/calculator.svg
```

25.3 Atalhos

Em algumas ocasiões, pode ser necessário ter um mesmo arquivo ou diretório replicado em várias posições dentro do sistema de arquivos. Isso ocorre frequentemente com arquivos de configuração de programas e arquivos de bibliotecas, por exemplo. Nestes casos, seria mais econômico (em espaço de armazenamento) armazenar apenas uma instância dos dados do arquivo no sistema de arquivos e criar referências indiretas (ponteiros) para essa instância, para representar as demais cópias do arquivo. O mesmo raciocínio pode ser aplicado a diretórios duplicados. Essas referências indiretas a arquivos ou diretórios são denominadas *atalhos* (ou *links*).

A listagem (simplificada) a seguir apresenta algumas entradas do diretório `/usr/lib/` em um sistema Linux. Nela podem ser observados alguns atalhos: as entradas `libcryptui.so` e `libcryptui.so.0` são atalhos para o arquivo `libcryptui.so.0.0.0`, enquanto a entrada `libcrypt.so` é um atalho para o arquivo `/lib/x86_64-linux-gnu/libcrypt.so.1`.

```
~> ls -l /usr/lib/

...
lrwxrwxrwx  1 root root libcrypt.so -> /lib/x86_64-linux-gnu/libcrypt.so.1
lrwxrwxrwx  1 root root libcryptui.so -> libcryptui.so.0.0.0
lrwxrwxrwx  1 root root libcryptui.so.0 -> libcryptui.so.0.0.0
-rw-r--r--  1 root root libcryptui.so.0.0.0
...
```

Assim, quando um processo solicitar acesso ao arquivo `/usr/lib/libcrypt.so`, ele na verdade estará acessando o arquivo `/lib/x86_64-linux-gnu/libcrypt.so.1`, e assim por diante. Atalhos criam múltiplos caminhos para acessar o mesmo conteúdo, o que simplifica a organização de sistemas com muitos arquivos replicados (arquivos de configuração, ícones de aplicações, bibliotecas compartilhadas, etc).

25.4 Implementação de diretórios

A implementação de diretórios em um sistema de arquivos é relativamente simples: um diretório é implementado como um arquivo cujo conteúdo é uma relação de entradas (ou seja, uma tabela). Os tipos de entradas normalmente considerados nessa relação são arquivos normais, outros diretórios, atalhos (vide Seção 25.3) e entradas associadas a arquivos especiais, como os discutidos na Seção 22.4. Cada entrada contém ao menos o nome do arquivo (ou do diretório), seu tipo e a localização física do mesmo no volume (número do *i-node* ou número do bloco inicial). Deve ficar claro que um diretório não contém fisicamente os arquivos e subdiretórios, ele apenas os relaciona.

Em sistemas de arquivos mais antigos e simples, o diretório raiz de um volume estava definido em seus blocos de inicialização, normalmente reservados para informações de gerência. Todavia, como o número de blocos reservados era pequeno e fixo, o número de entradas no diretório raiz era limitado. Nos sistemas mais recentes, um registro específico dentro dos blocos reservados aponta para a posição do diretório raiz dentro do sistema de arquivos, permitindo que este tenha um número maior de entradas.

A Figura 25.3 apresenta a implementação de parte de uma estrutura de diretórios de um sistema UNIX. O endereço do diretório raiz (/) é indicado por uma entrada específica no VBR – *Volume Boot Record*, dentro da área reservada do disco ou partição. Neste exemplo, as entradas em cada diretório pode ser arquivos de dados (`A`) ou diretórios (`D`). Em uma implementação real podem existir outras entradas, como atalhos (`L` - *links*) e arquivos especiais (Seção 22.4). A informação sobre o tipo de cada entrada pode estar presente na própria tabela (como no exemplo da figura), ou pode estar nos metadados do *i-node* da entrada correspondente.

Na figura, podem também ser observadas duas entradas usualmente definidas em cada diretório: a entrada "`.`" (ponto), que representa o próprio diretório, e a entrada "`..`" (ponto-ponto), que representa seu diretório pai (o diretório imediatamente acima dele na hierarquia de diretórios). No caso do diretório raiz, ambas as entradas apontam para ele mesmo.

Internamente, a lista ou índice do diretório pode ser implementada como uma tabela simples, como no caso do MS-DOS e do Ext2 (Linux). Essa implementação é mais

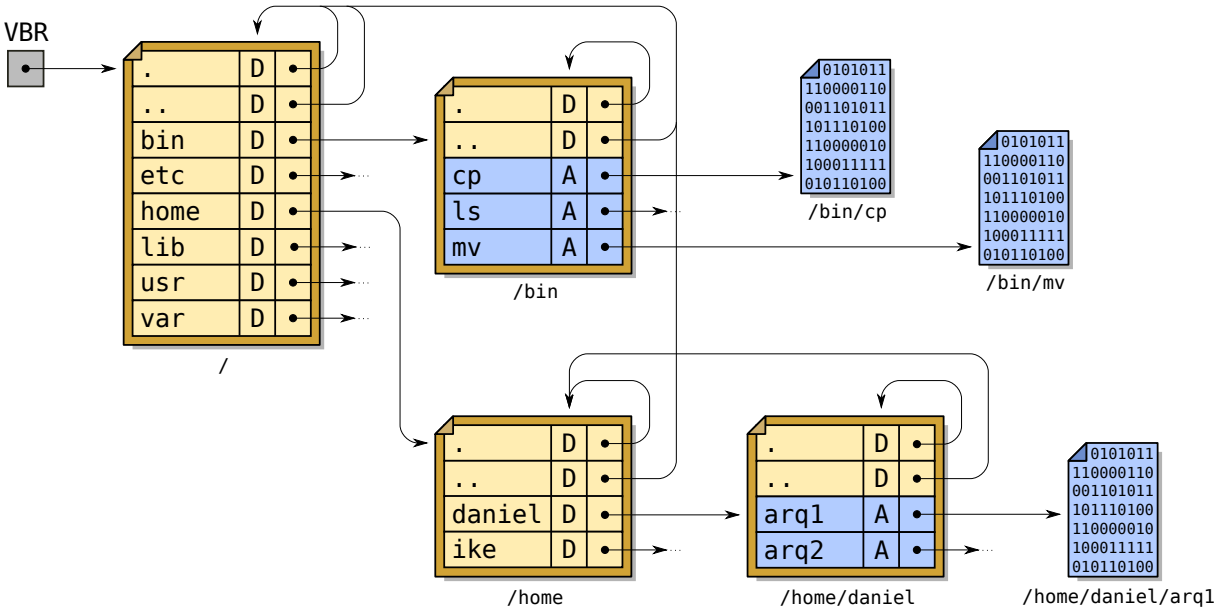


Figura 25.3: Implementação de uma estrutura de diretórios.

simples, mas tem baixo desempenho, sobretudo em diretórios contendo muitos ítens (a complexidade da busca em uma lista linear com n elementos é $O(n)$). Sistemas de arquivos mais sofisticados, como o Ext4, ZFS e NTFS usam estruturas de dados com maior desempenho de busca, como *hashes* e árvores.

25.5 Implementação de atalhos

Como apresentado na Seção 25.3, atalhos são entradas no sistema de arquivos que apontam para outras entradas. Existem basicamente duas abordagens para a implementação de atalhos:

Atalho simbólico (*soft link*): é implementado como um pequeno arquivo de texto contendo uma *string* com o caminho até o arquivo original (pode ser usado um caminho simples, absoluto ou relativo à posição do atalho). Como o caminho é uma *string*, o "alvo" do atalho pode estar localizado em outro dispositivo físico (outro disco ou uma unidade de rede). O arquivo apontado e seus atalhos simbólicos são totalmente independentes: caso o arquivo seja movido, renomeado ou removido, os atalhos simbólicos apontarão para um local inexistente; neste caso, diz-se que aqueles atalhos estão "quebrados" (*broken links*).

Atalho físico (*hard link*): várias referências do arquivo no sistema de arquivos apontam para a mesma localização do dispositivo físico onde o conteúdo do arquivo está de fato armazenado. Normalmente é mantido um contador de referências a esse conteúdo, indicando quantos atalhos físicos apontam para o mesmo: somente quando o número de referências ao arquivo for zero, aquele conteúdo poderá ser removido do dispositivo. Como são usadas referências à posição do arquivo no dispositivo, atalhos físicos só podem ser feitos para arquivos dentro do mesmo sistema de arquivos (o mesmo volume).

A Figura 25.4 traz exemplos de implementação de atalhos simbólicos e físicos em um sistema de arquivos UNIX. As entradas de diretórios indicadas como "L" correspondem a atalhos simbólicos (*links*). Nessa figura, pode-se observar que a entrada `/bin/sh` é um atalho simbólico para o arquivo `/bin/bash`. Atalhos simbólicos podem apontar para diretórios (`/lib` → `/usr/lib`) ou mesmo para outros atalhos (`/usr/bin/shell` → `/bin/sh`). Esses atalhos são transparentes para as aplicações, ou seja: uma aplicação que acessar o arquivo `/usr/bin/shell` estará na verdade acessando `/bin/bash`.

Por outro lado as entradas `/bin/cp` e `/usr/bin/copy` apontam para o mesmo conteúdo no disco, por isso são consideradas atalhos físicos a esse conteúdo. Atalhos físicos geralmente só podem ser feitos para arquivos dentro do mesmo sistema de arquivos (mesmo volume) e não são permitidos atalhos físicos para diretórios[1].

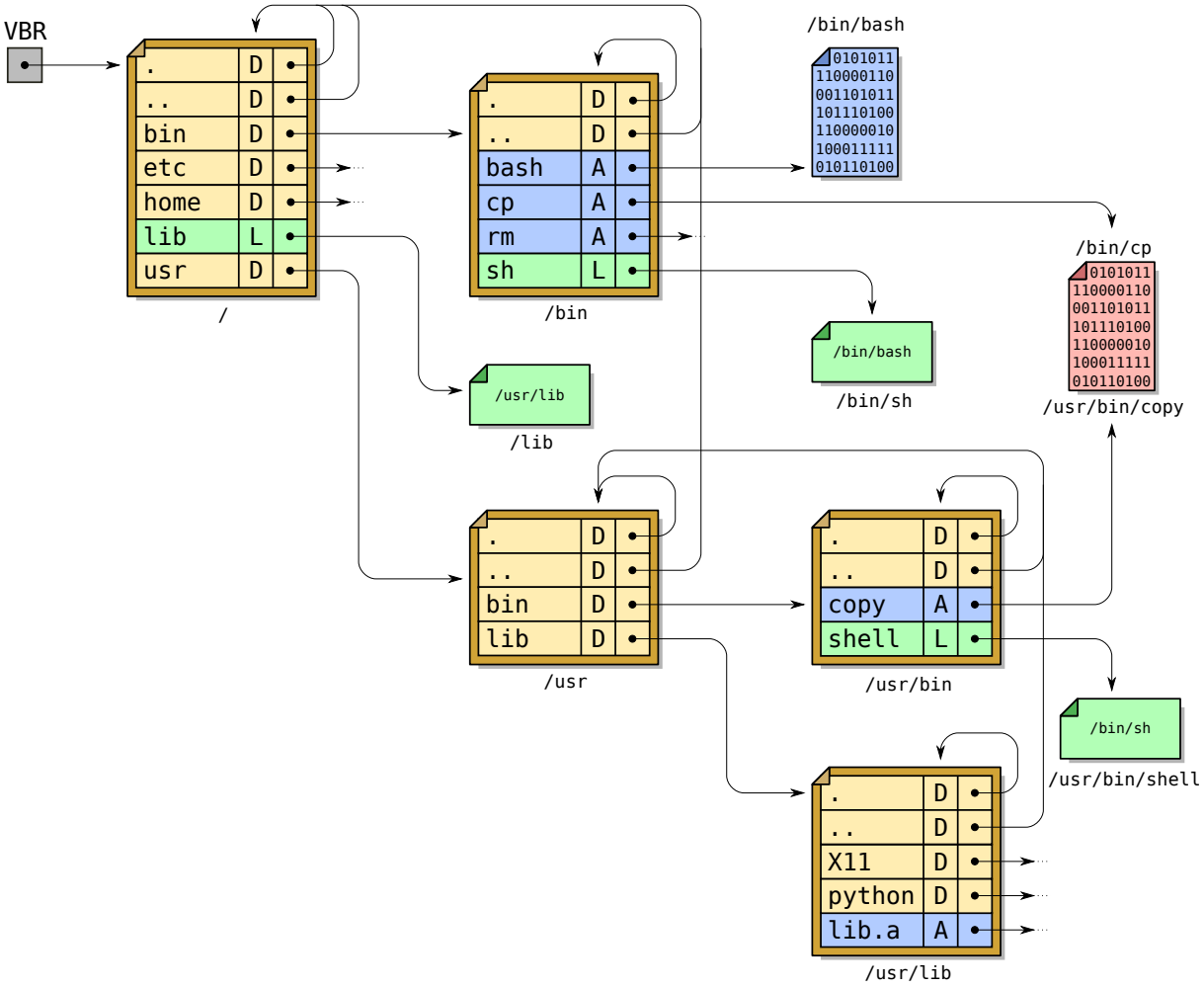


Figura 25.4: Atalhos simbólicos e físicos a arquivos em UNIX.

Em ambientes Windows, o sistema de arquivos NTFS suporta ambos os tipos de atalhos (embora atalhos simbólicos só tenham sido introduzidos no Windows Vista), com limitações similares.

[1] Atalhos físicos de diretórios transformariam a árvore de diretórios em um grafo, tornando mais complexa a implementação de rotinas que percorrem o sistema de arquivos recursivamente, como utilitários de *backup* e de gerência.

25.6 Tradução dos caminhos de acesso

A estrutura do sistema de arquivos em diretórios facilita a organização dos arquivos pelo usuário, mas complica a implementação da abertura de arquivos pelo sistema operacional. Para abrir um arquivo, o núcleo deve encontrar a localização do mesmo no dispositivo de armazenamento, a partir do nome de arquivo informado pelo processo. Para isso, é necessário percorrer o caminho do arquivo até encontrar sua localização, em um procedimento denominado *localização de arquivo* (*file lookup*).

Por exemplo, para abrir o arquivo `/home/daniel/arq1` da Figura 25.3 em um sistema UNIX (com alocação indexada) seria necessário executar os seguintes passos, ilustrados também na figura 25.5[2]:

1. Descobrir a localização do *i-node* do diretório raiz (/); essa informação pode estar no VBR (*Volume Boot Record*) do volume ou pode ser um valor fixo (por exemplo, o *i-node* 0).
2. Ler o *i-node* de / para:
   (a) Verificar se o processo tem permissão de acessar o conteúdo de /.
   (b) Descobrir em que bloco(s) está localizado o conteúdo de / (ou seja, a tabela de diretório contida em /).
3. Ler o conteúdo de / para encontrar o número do *i-node* correspondente à entrada `/home`.
4. Ler o *i-node* de /home para:
   (a) Verificar se o processo tem permissão de acessar o conteúdo de `/home`.
   (b) Descobrir em que bloco(s) está localizado o conteúdo de `/home`.
5. Ler o conteúdo de `/home` para encontrar o número do *i-node* correspondente à entrada `/home/daniel`.
6. Ler o *i-node* de `/home/daniel` para:
   (a) Verificar se o processo tem permissão de acessar o conteúdo de `/home/daniel`.
   (b) Descobrir em que bloco(s) está localizado o conteúdo de `/home/daniel`.
7. Ler o conteúdo de `/home/daniel` para encontrar o número do *i-node* correspondente à entrada `/home/daniel/arq1`.
8. Devolver o número do *i-node* encontrado.

O número do *i-node* devolvido no passo 8 corresponde ao arquivo desejado (`/home/daniel/arq1`), que pode então ser usado pelo restante do sistema para as operações envolvendo esse arquivo.

[2]Para simplificar, foram omitidos os tratamentos de erro.

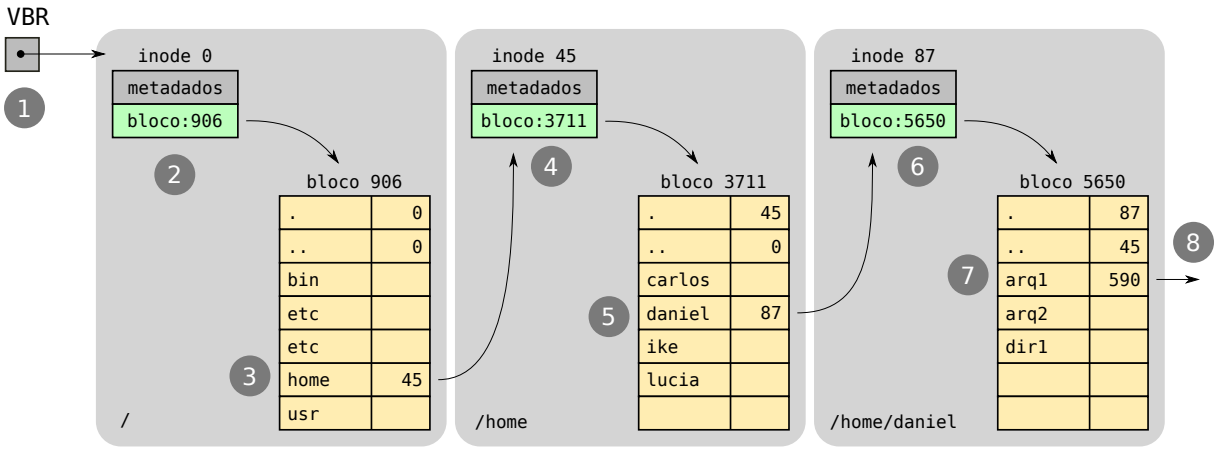


Figura 25.5: Resolução de nomes de arquivos.

Neste exemplo, foram necessárias 6 leituras no disco (passos 2-7) somente para localizar o *i-node* do arquivo `/home/daniel/arq1`, cujo caminho de acesso tem 3 níveis. Um arquivo com um caminho mais longo, como `/usr/share/texlive/texmf/tex/generic/pgf/graphdrawing/lua/pgf/gd/doc/ogdf/energybased/multilevelmixer/SolarMerger.lua` (16 níveis), apresenta um custo de localização bem mais elevado.

Para atenuar o custo de localização e melhorar o desempenho geral do acesso a arquivos, é mantido em memória um cache de entradas de diretório localizadas recentemente, denominado *cache de resolução de nomes* (*name lookup cache*). Cada entrada desse cache contém um nome absoluto de arquivo ou diretório e o número do *i-node* correspondente (ou outra informação que permita localizá-lo no dispositivo físico). A Tabela 25.1 representa as entradas do cache de nomes relativas ao exemplo apresentado nesta seção (Figura 25.5).

caminho	***i-node***
`/home/daniel/arq1`	590
`/home/daniel`	87
`/home`	45
`/`	0

Tabela 25.1: Conteúdo parcial do cache de resolução de nomes.

Esse cache geralmente é organizado na forma de uma tabela *hash* e gerenciado usando uma política LRU (*Least Recently Used*). A consulta ao cache é feita de forma iterativa, partindo do nome completo do arquivo e removendo a última parte a cada consulta, até encontrar uma localização conhecida. Considerando o conteúdo de cache da Tabela 25.1, eis alguns exemplos de consultas:

`/home/daniel/arq1` → *inode* 590

`/home/daniel/arq2` → não
`/home/daniel` → *inode* 87

```
/home/maziero/imagens/foto.jpg → não
/home/maziero/imagens          → não
/home/maziero                  → não
/home                          → inode 45

/usr/bin/bash                  → não
/usr/bin                       → não
/usr                           → não
/                              → inode 0
```

A resolução de caminho de acesso para um atalho físico segue exatamente o mesmo procedimento, pois um atalho físico é simplesmente um caminho altarnativo para um arquivo. Por outro lado, se o caminho a resolver apontar para um atalho simbólico, é necessário ler o conteúdo desse atalho e aplicar o procedimento de resolução sobre o novo caminho encontrado.

Referências

R. Russell, D. Quinlan, and C. Yeoh. Filesystem Hierarchy Standard, January 2004.

Parte VII

Segurança

Capítulo 26

Conceitos básicos de segurança

A segurança de um sistema de computação diz respeito à garantia de algumas propriedades fundamentais associadas às informações e recursos presentes no sistema. Por "informação", compreende-se todos os recursos disponíveis no sistema, como registros de bancos de dados, arquivos, áreas de memória, dados de entrada/saída, tráfego de rede, configurações, etc.

Em Português, a palavra "segurança" abrange muitos significados distintos e por vezes conflitantes. Em Inglês, as palavras "security", "safety" e "reliability" permitem definir mais precisamente os diversos aspectos da segurança: a palavra "security" se relaciona a ameaças intencionais, como intrusões, ataques e roubo de informações; a palavra "safety" se relaciona a problemas que possam ser causados pelo sistema aos seus usuários ou ao ambiente, como acidentes provocados por erros de programação; por fim, o termo "reliability" é usado para indicar sistemas confiáveis, construídos para tolerar erros de software, de hardware ou dos usuários [Avizienis et al., 2004]. Neste capítulo serão considerados somente os aspectos de segurança relacionados à palavra inglesa "security", ou seja, a proteção do sistema contra ameaças intencionais.

Este capítulo trata dos principais conceitos de segurança, como as propriedades e princípios de segurança, ameaças, vulnerabilidades e ataques típicos em sistemas operacionais, concluindo com uma descrição da infraestrutura de segurança típica de um sistema operacional. Grande parte dos tópicos de segurança apresentados neste e nos próximos capítulos não são exclusivos de sistemas operacionais, mas se aplicam a sistemas de computação em geral.

26.1 Propriedades e princípios de segurança

A segurança de um sistema de computação pode ser expressa através de algumas propriedades fundamentais [Amoroso, 1994]:

Confidencialidade: os recursos presentes no sistema só podem ser consultados por usuários devidamente autorizados a isso;

Integridade: os recursos do sistema só podem ser modificados ou destruídos pelos usuários autorizados a efetuar tais operações;

Disponibilidade: os recursos devem estar disponíveis para os usuários que tiverem direito de usá-los, a qualquer momento.

Além destas, outras propriedades importantes estão geralmente associadas à segurança de um sistema:

Autenticidade: todas as entidades do sistema são autênticas ou genuínas; em outras palavras, os dados associados a essas entidades são verdadeiros e correspondem às informações do mundo real que elas representam, como as identidades dos usuários, a origem dos dados de um arquivo, etc.;

Irretratabilidade: Todas as ações realizadas no sistema são conhecidas e não podem ser escondidas ou negadas por seus autores; esta propriedade também é conhecida como *irrefutabilidade* ou *não-repúdio*.

É função do sistema operacional garantir a manutenção das propriedades de segurança para todos os recursos sob sua responsabilidade. Essas propriedades podem estar sujeitas a violações decorrentes de erros de software ou humanos, praticadas por indivíduos mal intencionados (maliciosos), internos ou externos ao sistema.

Além das técnicas usuais de engenharia de software para a produção de sistemas corretos, a construção de sistemas computacionais seguros é pautada por uma série de princípios específicos, relativos tanto à construção do sistema quanto ao comportamento dos usuários e dos atacantes. Alguns dos princípios mais relevantes, compilados a partir de [Saltzer and Schroeder, 1975; Lichtenstein, 1997; Pfleeger and Pfleeger, 2006], são indicados a seguir:

Privilégio mínimo: todos os usuários e programas devem operar com o mínimo possível de privilégios ou permissões de acesso necessários para poder funcionar. Dessa forma, os danos provocados por erros ou ações maliciosas intencionais serão minimizados.

Separação de privilégios: sistemas de proteção baseados em mais de um controle são mais robustos, pois se o atacante conseguir burlar um dos controles, mesmo assim não terá acesso ao recurso. Um exemplo típico é o uso de mais de uma forma de autenticação para acesso ao sistema (como um cartão e uma senha, nos sistemas bancários). Se o atacante obtiver somente um dos privilégios (o cartão ou a senha), não conseguirá concretizar o ataque.

Mediação completa: todos os acessos a recursos, tanto diretos quanto indiretos, devem ser verificados pelos mecanismos de segurança. Eles devem estar dispostos de forma a ser impossível contorná-los.

Default seguro: o mecanismo de segurança deve identificar claramente os acessos permitidos; caso um certo acesso não seja explicitamente permitido, ele deve ser negado. Este princípio impede que acessos inicialmente não previstos no projeto do sistema sejam inadvertidamente autorizados.

Economia de mecanismo: o projeto de um sistema de proteção deve ser pequeno e simples, para que possa ser facilmente e profundamente analisado, testado e validado.

Compartilhamento mínimo: mecanismos compartilhados entre usuários são fontes potenciais de problemas de segurança, devido à possibilidade de fluxos de

informação imprevistos entre usuários. Por isso, o uso de mecanismos compartilhados deve ser minimizado, sobretudo se envolver áreas de memória compartilhadas. Por exemplo, caso uma certa funcionalidade do sistema operacional possa ser implementada como chamada ao núcleo ou como função de biblioteca, deve-se preferir esta última forma, pois envolve menos compartilhamento.

Projeto aberto: a robustez do mecanismo de proteção não deve depender da ignorância dos atacantes; ao invés disso, o projeto deve ser público e aberto, dependendo somente do segredo de poucos itens, como listas de senhas ou chaves criptográficas. Um projeto aberto também torna possível a avaliação por terceiros independentes, provendo confirmação adicional da segurança do mecanismo.

Proteção adequada: cada recurso computacional deve ter um nível de proteção coerente com seu valor intrínseco. Por exemplo, o nível de proteção requerido em um servidor Web de serviços bancário é bem distinto daquele de um terminal público de acesso à Internet.

Facilidade de uso: o uso dos mecanismos de segurança deve ser fácil e intuitivo, caso contrário eles serão evitados pelos usuários.

Eficiência: os mecanismos de segurança devem ser eficientes no uso dos recursos computacionais, de forma a não afetar significativamente o desempenho do sistema ou as atividades de seus usuários.

Elo mais fraco: a segurança do sistema é limitada pela segurança de seu elemento mais vulnerável, seja ele o sistema operacional, as aplicações, a conexão de rede ou o próprio usuário.

Esses princípios devem pautar a construção, configuração e operação de qualquer sistema computacional com requisitos de segurança. A imensa maioria dos problemas de segurança dos sistemas atuais provém da não observação desses princípios.

26.2 Ameaças

Como ameaça, pode ser considerada qualquer ação que coloque em risco as propriedades de segurança do sistema descritas na seção anterior. Alguns exemplos de ameaças às propriedades básicas de segurança seriam:

- *Ameaças à confidencialidade*: um processo vasculhar as áreas de memória de outros processos, arquivos de outros usuários, tráfego de rede nas interfaces locais ou áreas do núcleo do sistema, buscando dados sensíveis como números de cartão de crédito, senhas, e-mails privados, etc.;

- *Ameaças à integridade*: um processo alterar as senhas de outros usuários, instalar programas, *drivers* ou módulos de núcleo maliciosos, visando obter o controle do sistema, roubar informações ou impedir o acesso de outros usuários;

- *Ameaças à disponibilidade*: um usuário alocar para si todos os recursos do sistema, como a memória, o processador ou o espaço em disco, para impedir que outros usuários possam utilizá-lo.

Obviamente, para cada ameaça possível, devem existir estruturas no sistema operacional que impeçam sua ocorrência, como controles de acesso às áreas de memória e arquivos, quotas de uso de memória e processador, verificação de autenticidade de *drivers* e outros softwares, etc.

As ameaças podem ou não se concretizar, dependendo da existência e da correção dos mecanismos construídos para evitá-las ou impedi-las. As ameaças podem se tornar realidade à medida em que existam vulnerabilidades que permitam sua ocorrência.

26.3 Vulnerabilidades

Uma vulnerabilidade é um defeito ou problema presente na especificação, implementação, configuração ou operação de um software ou sistema, que possa ser explorado para violar as propriedades de segurança do mesmo. Alguns exemplos de vulnerabilidades são descritos a seguir:

- um erro de programação no serviço de compartilhamento de arquivos, que permita a usuários externos o acesso a outros arquivos do computador local, além daqueles compartilhados;
- uma conta de usuário sem senha, ou com uma senha pré-definida pelo fabricante, que permita a usuários não autorizados acessar o sistema;
- ausência de quotas de disco, permitindo a um único usuário alocar todo o espaço em disco para si e assim impedir os demais usuários de usar o sistema.

A grande maioria das vulnerabilidades ocorre devido a erros de programação, como, por exemplo, não verificar a conformidade dos dados recebidos de um usuário ou da rede. Em um exemplo clássico, o processo servidor de impressão `lpd`, usado em alguns UNIX, pode ser instruído a imprimir um arquivo e a seguir apagá-lo, o que é útil para imprimir arquivos temporários. Esse processo executa com permissões administrativas pois precisa acessar a porta de entrada/saída da impressora, o que lhe confere acesso a todos os arquivos do sistema. Por um erro de programação, uma versão antiga do processo `lpd` não verificava corretamente as permissões do usuário sobre o arquivo a imprimir; assim, um usuário malicioso podia pedir a impressão (e o apagamento) de arquivos do sistema. Em outro exemplo clássico, uma versão antiga do servidor HTTP Microsoft IIS não verificava adequadamente os pedidos dos clientes; por exemplo, um cliente que solicitasse a URL `http://www.servidor.com/../../../../windows/system.ini`, receberia como resultado o conteúdo do arquivo de sistema `system.ini`, ao invés de ter seu pedido recusado.

Uma classe especial de vulnerabilidades decorrentes de erros de programação são os chamados "estouros" de *buffer* e de pilha (*buffer/stack overflows*). Nesse erro, o programa escreve em áreas de memória indevidamente, com resultados imprevisíveis, como mostra o exemplo a seguir e o resultado de sua execução:

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

int i, j, buffer[20], k;  // declara buffer[0] a buffer[19]

int main()
{
  i = j = k = 0 ;

  for (i = 0; i<= 20; i++) // usa buffer[0] a buffer[20] <-- erro!
     buffer[i] = random() ;

  printf ("i: %d\nj: %d\nk: %d\n", i, j, k) ;

  return(0);
}
```

A execução desse código gera o seguinte resultado:

```
host:~> cc buffer-overflow.c -o buffer-overflow
host:~> buffer-overflow
i: 21
j: 35005211
k: 0
```

Pode-se observar que os valores $i = 21$ e $k = 0$ são os previstos, mas o valor da variável j mudou "misteriosamente" de 0 para 35005211. Isso ocorreu porque, ao acessar a posição `buffer[20]`, o programa extrapolou o tamanho do vetor e escreveu na área de memória sucessiva[1], que pertence à variável j. Esse tipo de erro é muito frequente em linguagens como C e C++, que não verificam os limites de alocação das variáveis durante a execução. O erro de estouro de pilha é similar a este, mas envolve variáveis alocadas na pilha usada para o controle de execução de funções.

Se a área de memória invadida pelo estouro de *buffer* contiver código executável, o processo pode ter erros de execução e ser abortado. A pior situação ocorre quando os dados a escrever no *buffer* são lidos do terminal ou recebidos através da rede: caso o atacante conheça a organização da memória do processo, ele pode escrever inserir instruções executáveis na área de memória invadida, mudando o comportamento do processo ou abortando-o. Caso o *buffer* esteja dentro do núcleo, o que ocorre em *drivers* e no suporte a protocolos de rede como o TCP/IP, um estouro de *buffer* pode travar o sistema ou permitir acessos indevidos a recursos. Um bom exemplo é o famoso *Ping of Death* [Pfleeger and Pfleeger, 2006], no qual um pacote de rede no protocolo ICMP, com um conteúdo específico, podia paralisar computadores na rede local.

Além dos estouros de *buffer* e pilha, há uma série de outros erros de programação e de configuração que podem constituir vulnerabilidades, como o uso descuidado das strings de formatação de operações de entrada/saída em linguagens como C e C++ e condições de disputa na manipulação de arquivos compartilhados. Uma explicação mais detalhada desses erros e de suas implicações pode ser encontrada em [Pfleeger and Pfleeger, 2006].

[1] As variáveis não são alocadas na memória necessariamente na ordem em que são declaradas no código fonte. A ordem de alocação das variáveis varia com o compilador usado e depende de vários fatores, como a arquitetura do processador, estratégias de otimização de código, etc.

26.4 Ataques

Um ataque é o ato de utilizar (ou explorar) uma vulnerabilidade para violar uma propriedade de segurança do sistema. De acordo com [Pfleeger and Pfleeger, 2006], existem basicamente quatro tipos de ataques, representados na Figura 26.1:

Interrupção: consiste em impedir o fluxo normal das informações ou acessos; é um ataque à disponibilidade do sistema;

Interceptação: consiste em obter acesso indevido a um fluxo de informações, sem necessariamente modificá-las; é um ataque à confidencialidade;

Modificação: consiste em modificar de forma indevida informações ou partes do sistema, violando sua integridade;

Fabricação: consiste em produzir informações falsas ou introduzir módulos ou componentes maliciosos no sistema; é um ataque à autenticidade.

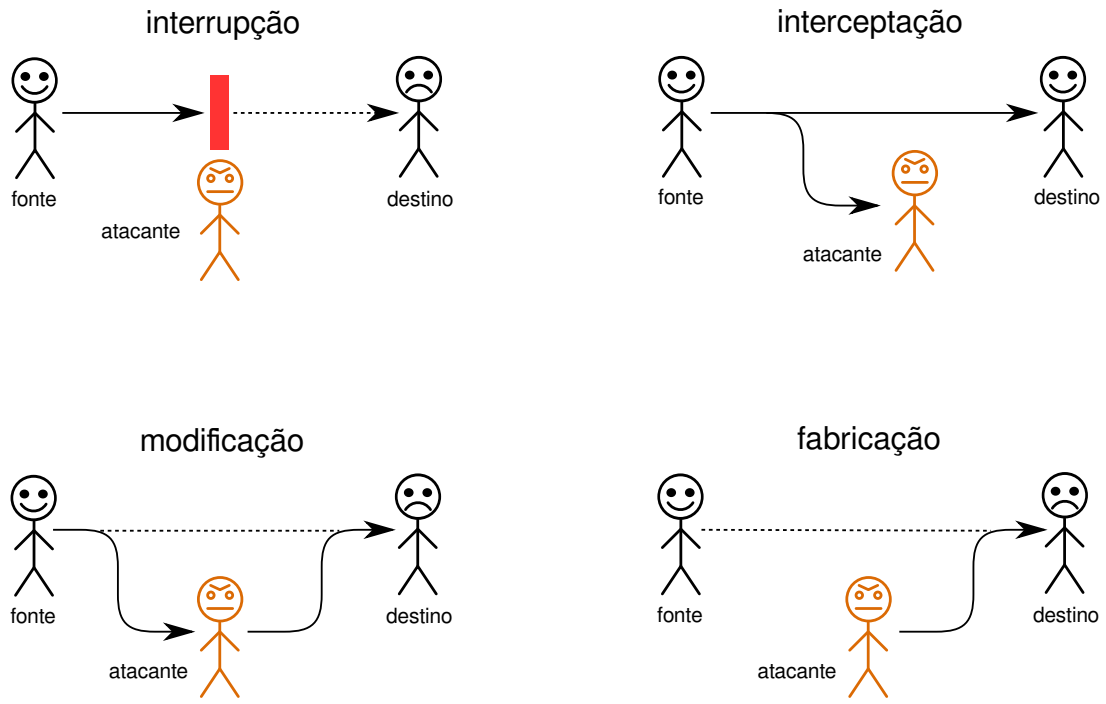


Figura 26.1: Tipos básicos de ataques (inspirado em [Pfleeger and Pfleeger, 2006]).

Existem ataques **passivos**, que visam capturar informações confidenciais, e ataques **ativos**, que visam introduzir modificações no sistema para beneficiar o atacante ou impedir seu uso pelos usuários válidos. Além disso, os ataques a um sistema operacional podem ser **locais**, quando executados por usuários válidos do sistema, ou **remotos**, quando são realizados através da rede, sem fazer uso de uma conta de usuário local. Um programa especialmente construído para explorar uma determinada vulnerabilidade de sistema e realizar um ataque é denominado *exploit*.

Uma intrusão ou invasão é um ataque bem sucedido, que dá ao atacante acesso indevido a um sistema. Uma vez dentro do sistema, o intruso pode usá-lo para fins escusos, como enviar *spam*, atacar outros sistemas ou minerar moedas

digitais. Frequentemente o intruso efetua novos ataques para aumentar seu nível de acesso no sistema, o que é denominado *elevação de privilégio* (*privilege escalation*). Esses ataques exploram vulnerabilidades em programas do sistema (que executam com mais privilégios), ou do próprio núcleo, através de chamadas de sistema, para alcançar os privilégios do administrador.

Por outro lado, os ataques de negação de serviços (DoS – *Denial of Service*) visam prejudicar a disponibilidade do sistema, impedindo que os usuários válidos do sistema possam utilizá-lo, ou seja, que o sistema execute suas funções. Esse tipo de ataque é muito comum em ambientes de rede, com a intenção de impedir o acesso a servidores Web, DNS e de e-mail. Em um sistema operacional, ataques de negação de serviço podem ser feitos com o objetivo de consumir todos os recursos locais, como processador, memória, arquivos abertos, *sockets* de rede ou semáforos, dificultando ou mesmo impedindo o uso desses recursos pelos demais usuários.

O antigo ataque *fork bomb* dos sistemas UNIX é um exemplo trivial de ataque DoS local: ao executar, o processo atacante se reproduz rapidamente, usando a chamada de sistema `fork` (vide código a seguir). Cada processo filho continua executando o mesmo código do processo pai, criando novos processos filhos, e assim sucessivamente. Em consequência, a tabela de processos do sistema é rapidamente preenchida, impedindo a criação de processos pelos demais usuários. Além disso, o grande número de processos solicitando chamadas de sistema mantém o núcleo ocupado, impedindo os a execução dos demais processos.

```
1 #include <unistd.h>
2
3 int main()
4 {
5   while (1)    // laço infinito
6     fork() ;   // reproduz o processo
7 }
```

Ataques similares ao *fork bomb* podem ser construídos para outros recursos do sistema operacional, como memória, descritores de arquivos abertos, *sockets* de rede e espaço em disco. Cabe ao sistema operacional impor limites máximos (quotas) de uso de recursos para cada usuário e definir mecanismos para detectar e conter processos excessivamente "gulosos".

Recentemente têm ganho atenção os ataques à confidencialidade, que visam roubar informações sigilosas dos usuários. Com o aumento do uso da Internet para operações financeiras, como acesso a sistemas bancários e serviços de compras *online*, o sistema operacional e os navegadores manipulam informações sensíveis, como números de cartões de crédito, senhas de acesso a contas bancárias e outras informações pessoais. Programas construídos com a finalidade específica de realizar esse tipo de ataque são denominados *spyware*.

Deve ficar clara a distinção entre *ataques* e *incidentes de segurança*. Um incidente de segurança é qualquer fato intencional ou acidental que comprometa uma das propriedades de segurança do sistema. A intrusão de um sistema ou um ataque de negação de serviços são considerados incidentes de segurança, assim como o vazamento acidental de informações confidenciais.

26.5 Malwares

Denomina-se genericamente *malware* todo programa cuja intenção é realizar atividades ilícitas, como realizar ataques, roubar informações ou dissimular a presença de intrusos em um sistema. Existe uma grande diversidade de *malwares*, destinados às mais diversas finalidades [Shirey, 2000; Pfleeger and Pfleeger, 2006]. As funcionalidades mais comuns dos *malwares* são:

Vírus: um vírus de computador é um trecho de código que se infiltra em programas executáveis existentes no sistema operacional, usando-os como suporte para sua execução e replicação[2]. Quando um programa "infectado" é executado, o vírus também se executa, infectando outros executáveis e eventualmente executando outras ações danosas. Alguns tipos de vírus são programados usando macros de aplicações complexas, como editores de texto, e usam os arquivos de dados dessas aplicações como suporte. Outros tipos de vírus usam o código de inicialização dos discos e outras mídias como suporte de execução.

Worm: ao contrário de um vírus, um "verme" é um programa autônomo, que se propaga sem infectar outros programas. A maioria dos vermes se propaga explorando vulnerabilidades nos serviços de rede, que os permitam invadir e instalar-se em sistemas remotos. Alguns vermes usam o sistema de e-mail como vetor de propagação, enquanto outros usam mecanismos de autoexecução de mídias removíveis (como *pendrives*) como mecanismo de propagação. Uma vez instalado em um sistema, o verme pode instalar *spywares* ou outros programas nocivos.

Trojan horse: de forma análoga ao personagem da mitologia grega, um "cavalo de Tróia" computacional é um programa com duas funcionalidades: uma funcionalidade lícita conhecida de seu usuário e outra ilícita, executada sem que o usuário a perceba. Muitos cavalos de Tróia são usados como vetores para a instalação de outros *malwares*. Um exemplo clássico é o famoso *Happy New Year 99*, distribuído através de e-mails, que usava uma animação de fogos de artifício como fachada para a propagação de um verme. Para convencer o usuário a executar o cavalo de Tróia podem ser usadas técnicas de *engenharia social* [Mitnick and Simon, 2002].

Exploit: é um programa escrito para explorar vulnerabilidades conhecidas, como prova de conceito ou como parte de um ataque. Os *exploits* podem estar incorporados a outros *malwares* (como vermes e *trojans*) ou constituírem ferramentas autônomas, usadas em ataques manuais.

Packet sniffer: um "farejador de pacotes" captura pacotes de rede do próprio computador ou da rede local, analisando-os em busca de informações sensíveis como senhas e dados bancários. A cifragem do conteúdo da rede resolve parcialmente esse problema, embora um *sniffer* na máquina local possa capturar os dados antes que sejam cifrados, ou depois de decifrados.

[2] De forma análoga, um vírus biológico precisa de uma célula hospedeira, pois usa o material celular como suporte para sua existência e replicação.

Keylogger: software dedicado a capturar e analisar as informações digitadas pelo usuário na máquina local, sem seu conhecimento. Essas informações podem ser transferidas a um computador remoto periodicamente ou em tempo real, através da rede.

Rootkit: é um conjunto de programas destinado a ocultar a presença de um intruso no sistema operacional. Como princípio de funcionamento, o *rootkit* modifica os mecanismos do sistema operacional que mostram os processos em execução, arquivos nos discos, portas e conexões de rede, etc., para ocultar o intruso. Os *rootkits* mais simples substituem utilitários do sistema, como `ps` (lista de processos), `ls` (arquivos), `netstat` (conexões de rede) e outros, por versões adulteradas que não mostrem os arquivos, processos e conexões de rede do intruso. Versões mais elaboradas de *rootkits* substituem bibliotecas do sistema operacional ou modificam partes do próprio núcleo, o que torna complexa sua detecção e remoção.

Backdoor: uma "porta dos fundos" é um programa que facilita a entrada posterior do atacante em um sistema já invadido. Geralmente a porta dos fundos é criada através um processo servidor de conexões remotas (usando SSH, telnet ou um protocolo ad-hoc). Muitos *backdoors* são instalados a partir de *trojans*, vermes ou *rootkits*.

Ransomware: categoria recente de malware, que visa sequestrar os dados do usuário. O sequestro é realizado cifrando os arquivos do usuário com uma chave secreta, que só será fornecida pelo atacante ao usuário se este pagar um valor de resgate.

Deve-se ter em mente que existe muita confusão na mídia e na literatura em relação à nomenclatura de *malwares*; além disso, a maioria dos *malwares* atuais são complexos, apresentando várias funcionalidades complementares. Por exemplo, um mesmo *malware* pode se propagar como *worm*, dissimular-se no sistema como *rootkit*, capturar informações locais usando *keylogger* e manter uma *backdoor* para acesso remoto. Por isso, esta seção procurou dar uma definição tecnicamente precisa de cada funcionalidade, sem a preocupação de aspresentar exemplos reais de *malwares* em cada uma dessas categorias.

26.6 Infraestrutura de segurança

De forma genérica, o conjunto de todos os elementos de hardware e software considerados críticos para a segurança de um sistema são denominados **Base Computacional Confiável** (TCB – *Trusted Computing Base*) ou **núcleo de segurança** (*security kernel*). Fazem parte da TCB todos os elementos do sistema cuja falha possa representar um risco à sua segurança. Os elementos típicos de uma base de computação confiável incluem os mecanismos de proteção do hardware (tabelas de páginas/segmentos, modo usuário/núcleo do processador, instruções privilegiadas, etc.) e os diversos subsistemas do sistema operacional que visam garantir as propriedades básicas de segurança, como o controle de acesso aos arquivos, acesso às portas de rede, etc.

O sistema operacional emprega várias técnicas complementares para garantir a segurança de um sistema operacional. Essas técnicas estão classificadas nas seguintes grandes áreas:

Autenticação: conjunto de técnicas usadas para identificar inequivocamente usuários e recursos em um sistema; podem ir de simples pares *login/senha* até esquemas sofisticados de biometria ou certificados criptográficos. No processo básico de autenticação, um usuário externo se identifica para o sistema através de um procedimento de autenticação; no caso da autenticação ser bem-sucedida, é aberta uma *sessão*, na qual são criados uma ou mais entidades (processos, *threads*, transações, etc.) para representar aquele usuário dentro do sistema.

Controle de acesso: técnicas usadas para definir quais ações são permitidas e quais são negadas no sistema; para cada usuário do sistema, devem ser definidas regras descrevendo as ações que este pode realizar no sistema, ou seja, que recursos este pode acessar e sob que condições. Normalmente, essas regras são definidas através de uma *política de controle de acesso*, que é imposta a todos os acessos que os usuários efetuam sobre os recursos do sistema.

Auditoria: técnicas usadas para manter um registro das atividades efetuadas no sistema, visando a contabilização de uso dos recursos, a análise posterior de situações de uso indevido ou a identificação de comportamentos suspeitos.

A Figura 26.2 ilustra alguns dos conceitos vistos até agora. Nessa figura, as partes indicadas em cinza e os mecanismos utilizados para implementá-las constituem a base de computação confiável do sistema.

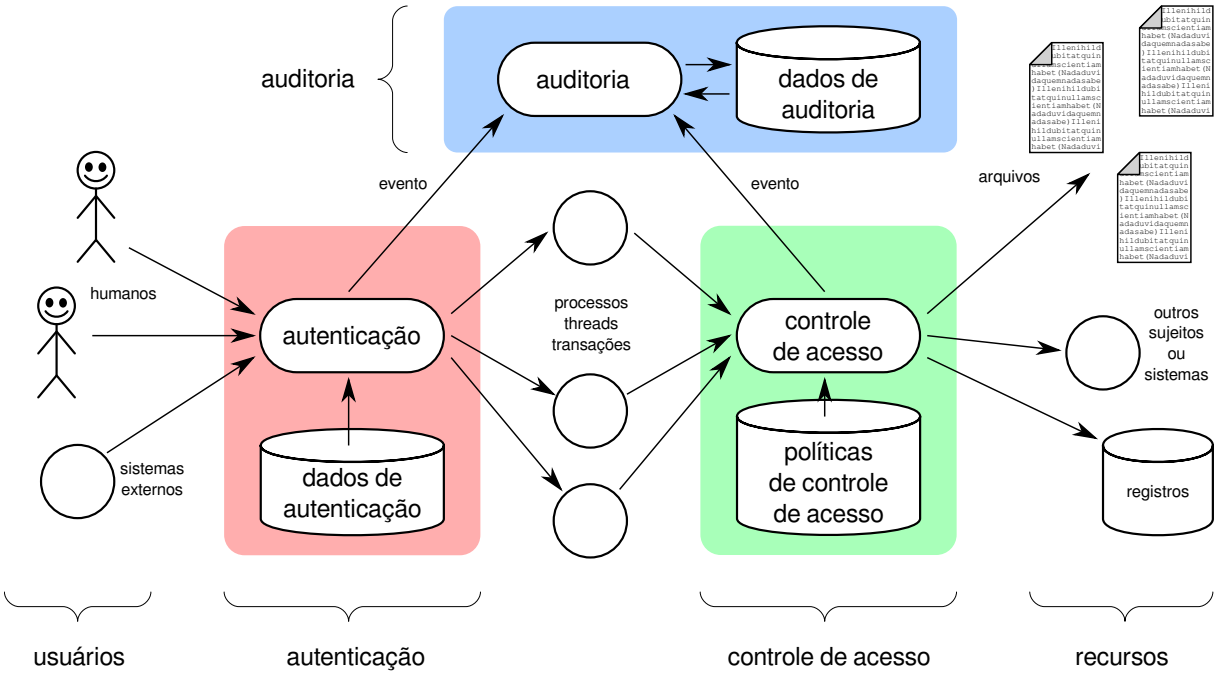


Figura 26.2: Base de computação confiável de um sistema operacional.

Sob uma ótica mais ampla, a base de computação confiável de um sistema informático compreende muitos fatores além do sistema operacional em si. A manutenção das propriedades de segurança depende do funcionamento correto de todos os elementos do sistema, do hardware ao usuário final.

O hardware fornece várias funcionalidades essenciais para a proteção do sistema: os mecanismos de memória virtual (MMU) permitem isolar o núcleo e os processos entre

si; o mecanismo de interrupção de software provê uma interface controlada de acesso ao núcleo; os níveis de execução do processador permitem restringir as instruções e as portas de entrada saída acessíveis aos diversos softwares que compõem o sistema; além disso, muitos tipos de hardware permitem impedir operações de escrita ou execução de código em certas áreas de memória.

No nível do sistema operacional surgem os processos isolados entre si, as contas de usuários, os mecanismos de autenticação e controle de acesso e os registros de auditoria. Em paralelo com o sistema operacional estão os utilitários de segurança, como antivírus, verificadores de integridade, detectores de intrusão, entre outros.

As linguagens de programação também desempenham um papel importante nesse contexto, pois muitos problemas de segurança têm origem em erros de programação. O controle estrito de índices em vetores, a restrição do uso de ponteiros e a limitação de escopo de nomes para variáveis e funções são exemplos de aspectos importantes para a segurança de um programa. Por fim, as aplicações também têm responsabilidade em relação à segurança, no sentido de ter implementações corretas e validar todos os dados manipulados. Isso é particularmente importante em aplicações multi-usuários (como sistemas corporativos e sistemas Web) e processos privilegiados que recebam requisições de usuários ou da rede (servidores de impressão, de DNS, etc.).

Referências

E. Amoroso. *Fundamentals of Computer Security Technology*. Prentice Hall PTR, 1994.

A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1), Mar. 2004.

S. Lichtenstein. A review of information security principles. *Computer Audit Update*, 1997(12):9–24, December 1997.

K. D. Mitnick and W. L. Simon. *The Art of Deception: Controlling the Human Element of Security*. John Wiley & Sons, Inc., New York, NY, USA, 2002. ISBN 0471237124.

C. Pfleeger and S. L. Pfleeger. *Security in Computing, 4th Edition*. Prentice Hall PTR, 2006.

J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278 – 1308, September 1975.

R. Shirey. RFC 2828: Internet security glossary, May 2000.

Capítulo 27

Fundamentos de criptografia

Este capítulo apresenta uma introdução às técnicas clássicas de criptografia frequentemente usadas em sistemas operacionais e redes de computadores. Este texto não tem a mínima pretensão de ser completo sobre esse vasto tema; leitores em busca de uma abordagem mais profunda e completa devem procurar livros específicos sobre criptografia.

27.1 Terminologia

O termo "criptografia" provém das palavras gregas *kryptos* (oculto, secreto) e *graphos* (escrever). Assim, a criptografia foi criada para codificar informações, de forma que somente as pessoas autorizadas pudessem ter acesso ao seu conteúdo. Conceitualmente, a criptografia faz parte de um escopo mais amplo de conhecimento:

Criptografia: técnicas para codificar/decodificar informações, ocultando seu conteúdo de pessoas não autorizadas;

Criptanálise: conjunto de técnicas usadas para "quebrar" uma criptografia, expondo a informação ocultada por ela;

Criptologia: área geral, englobando criptografia e criptanálise.

Criptossistema: conjunto de algoritmos/mecanismos para realizar um tipo específico de criptografia.

As técnicas criptográficas são extensivamente usadas na segurança de sistemas, para garantir a confidencialidade e integridade dos dados. Além disso, elas desempenham um papel importante na autenticação de usuários e recursos.

Alguns conceitos fundamentais para estudar as técnicas criptográficas são [Menezes et al., 1996]:

Texto aberto: a mensagem ou informação a codificar (x);

Texto cifrado: a informação codificada de forma a ocultar seu conteúdo (x');

Chave: informação complementar, necessária para cifrar ou decifrar as informações (k);

Cifrar: transformar o texto aberto em texto cifrado ($x \xrightarrow{k} x'$) ;

Decifrar: transformar o texto cifrado em texto aberto ($x' \xrightarrow{k} x$) ;

Cifrador: mecanismo responsável por cifrar/decifrar as informações;

No restante deste texto, a operação de cifragem de um conteúdo aberto x usando uma chave k e gerando um conteúdo cifrado x' será representada por $x' = \{x\}_k$ e a decifragem de um conteúdo x' usando uma chave k será representada por $x = \{x'\}_k^{-1}$.

27.2 Cifradores, chaves e espaço de chaves

Uma das mais antigas técnicas criptográficas conhecidas é o *cifrador de César*, usado pelo imperador romano Júlio César para se comunicar com seus generais. O algoritmo usado nesse cifrador é bem simples: cada caractere do texto aberto é substituído pelo k-ésimo caractere sucessivo no alfabeto. Assim, considerando $k = 2$, a letra "A" seria substituída pela letra "C", a letra "R" pela "T", e assim por diante.

Usando esse algoritmo, a mensagem secreta "Reunir todos os generais para o ataque" seria cifrada da seguinte forma:

mensagem aberta:	`REUNIR TODOS OS GENERAIS PARA O ATAQUE`
mensagem cifrada com $k = 1$:	`SFVOJS UPEPT PT HFOFSBJT QBSB P BUBRVF`
mensagem cifrada com $k = 2$:	`TGWPKT VQFQU QU IGPGTCKU RCTC Q CVCSWG`
mensagem cifrada com $k = 3$:	`UHXQLU WRGRV RV JHQHUDLV SDUD R DWDTXH`

Para decifrar uma mensagem no cifrador de César, é necessário conhecer a mensagem cifrada e o valor de k utilizado para cifrar a mensagem, que é a *chave criptográfica*. Caso essa chave não seja conhecida, ainda é possível tentar "quebrar" a mensagem cifrada testando todas as chaves possíveis, o que é conhecido como análise exaustiva ou "ataque de força bruta". Considerando o cifrador de César e somente letras maiúsculas, a análise exaustiva é trivial, pois há somente 26 valores possíveis para a chave k (as 26 letras do alfabeto).

O número de chaves possíveis em um algoritmo de cifragem é conhecido como o seu **espaço de chaves** (*keyspace*). Em 1883, muito antes dos computadores eletrônicos, o criptólogo francês Auguste Kerckhoffs enunciou um princípio segundo o qual "*o segredo de uma técnica criptográfica não deve residir no algoritmo em si, mas no espaço de chaves que ela provê*". Obedecendo esse princípio, a criptografia moderna se baseia em algoritmos públicos, extensivamente avaliados pela comunidade científica, para os quais o espaço de chaves é extremamente grande, tornando inviável qualquer análise exaustiva, mesmo por computador.

Um bom exemplo de aplicação do princípio de Kerckhoffs é dado pelo algoritmo de criptografia AES (*Advanced Encryption Standard*), adotado como padrão pelo governo americano. Usando chaves de 128 bits, esse algoritmo oferece um espaço de chaves com 2^{128} possibilidades, ou seja, 340.282.366.920.938.463.463.374.607.431.768.211.456 chaves diferentes... Se pudéssemos testar um bilhão (10^9) de chaves por segundo, ainda assim seriam necessários 10 sextilhões de anos para testar todas as chaves possíveis!

27.3 O cifrador de Vernam-Mauborgne

O cifrador de Vernam-Mauborgne foi proposto em 1917 por Gilbert Vernam, engenheiro da ATT, e melhorado mais tarde por Joseph Mauborgne. Neste criptossistema, a cifragem de um texto aberto x com b bits de comprimento consiste em realizar uma operação XOR (OU-exclusivo, $\oplus$) entre os bits do texto aberto e os bits correspondentes de uma chave k de mesmo tamanho:

$$x' = x \oplus k, \text{ ou seja, } \forall i \in [1...b], x'_i = x_i \oplus k_i$$

Como a operação XOR é uma involução (pois $(x \oplus y) \oplus y = x$), a operação de decifragem consiste simplesmente em reaplicar essa mesma operação sobre o texto cifrado, usando a mesma chave:

$$x = x' \oplus k, \text{ ou seja, } \forall i \in [1...b], x_i = x'_i \oplus k_i$$

O exemplo a seguir ilustra este cifrador, usando caracteres ASCII. Nele, a mensagem x ("TOMATE") é cifrada usando a chave k ("ABCDEF"):

x (texto)	T	O	M	A	T	E
k (chave)	A	B	C	D	E	F
x (ASCII)	84	79	77	65	84	69
k (ASCII)	65	66	67	68	69	70
x (binário)	01010100	01001111	01001101	01000001	01010100	01000101
k (binário)	01000001	01000010	01000011	01000100	01000101	01000110
$x' = x \oplus k$	00010101	00001101	00001110	00000101	00010001	00000011
x' (ASCII)	21	13	14	5	17	3

Para decifrar a mensagem x', basta repetir a operação $\oplus$ com a mesma chave:

x'	00010101	00001101	00001110	00000101	00010001	00000011
k	01000001	01000010	01000011	01000100	01000101	01000110
$x' \oplus k$	01010100	01001111	01001101	01000001	01010100	01000101
(ASCII)	84	79	77	65	84	69
(texto)	T	O	M	A	T	E

A Figura 27.1 ilustra a aplicação do cifrador de Vernam-Mauborgne sobre uma imagem. A chave é uma imagem aleatória com as mesmas dimensões da imagem a cifrar; a operação de OU-exclusivo deve ser aplicada entre os pixels correspondentes na imagem e na chave.

Apesar de extremamente simples, o cifrador de Vernam-Mauborgne é considerado um criptossistema inquebrável, sendo comprovadamente seguro se a chave utilizada for realmente aleatória. Entretanto, ele é pouco usado na prática, porque exige uma chave k do mesmo tamanho do texto a cifrar, o que é pouco prático no caso de mensagens longas. Além disso, essa chave deve ser mantida secreta e deve ser usada uma única vez (ou seja, um atacante não deve ser capaz de capturar dois textos distintos

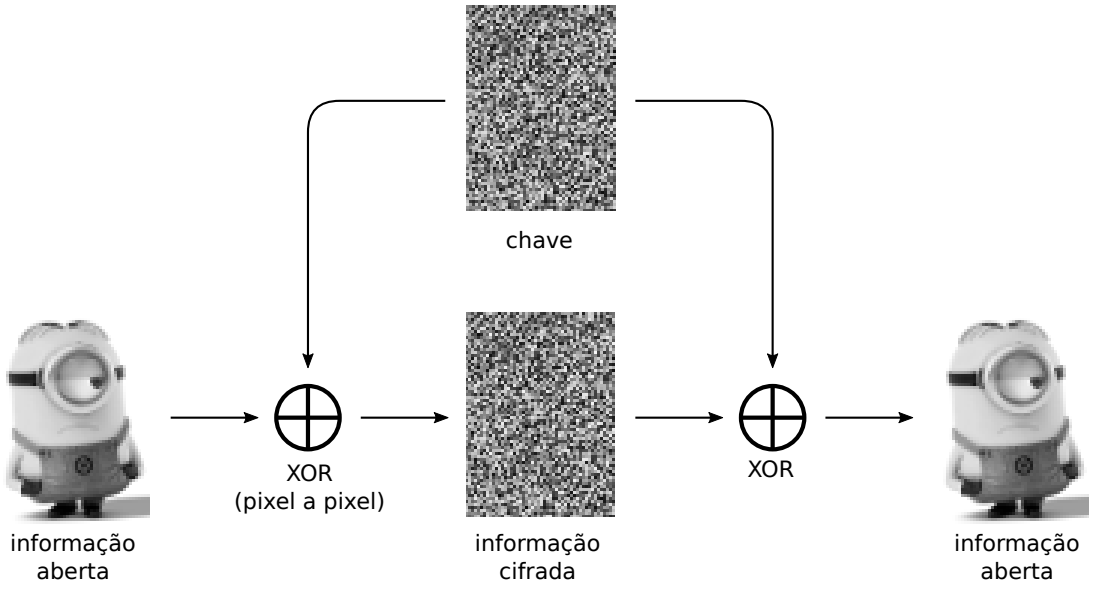


Figura 27.1: Aplicação do cifrador de Vernam sobre uma imagem.

cifrados com a mesma chave). O requisito de uso único da chave levou este cifrador a ser chamado também de *One-Time Pad* [1].

[1] O cifrador conhecido como *One-Time Pad* consiste em uma melhoria do algoritmo original de G. Vernam, proposto por J. Mauborgne pouco tempo depois.

27.4 Criptografia simétrica

De acordo com o tipo de chave utilizada, os algoritmos de criptografia se dividem em dois grandes grupos: *algoritmos simétricos* e *algoritmos assimétricos*. Nos **algoritmos simétricos**, a mesma chave k é usada para cifrar e decifrar a informação. Em outras palavras, se usarmos uma chave k para cifrar um texto, teremos de usar a mesma chave k para decifrá-lo. Essa propriedade pode ser expressa em termos matemáticos:

$$\{ \{ x \}_k \}_{k'}^{-1} = x \iff k' = k$$

Os cifradores de César e de Vernam são exemplos típicos de cifradores simétricos simples. Outros exemplos de cifradores simétricos bem conhecidos são:

- DES (*Data Encryption Standard*): criado pela IBM nos anos 1970, foi usado amplamente até o final do século XX. O algoritmo original usa chaves de 56 bits, o que gera um espaço de chaves insuficiente para a capacidade computacional atual. A variante 3DES (*Triple*-DES) usa chaves de 168 bits e é considerada segura, sendo ainda muito usada.
- AES (*Advanced Encryption Standard*): algoritmo simétrico adotado como padrão de segurança pelo governo americano em 2002. Ele pode usar chaves de 128, 192 ou 256 bits, sendo considerado muito seguro. É amplamente utilizado na Internet e em programas de cifragem de arquivos em disco.
- A5/1, A5/2, A5/3: algoritmos de criptografia simétrica usados em telefonia celular GSM, para cifrar as transmissões de voz.

A Figura 27.2 ilustra o esquema básico de funcionamento de um sistema de criptografia simétrica para a troca segura de informações. Nesse esquema, a chave simétrica deve ter sido previamente compartilhada entre quem envia e quem recebe a informação.

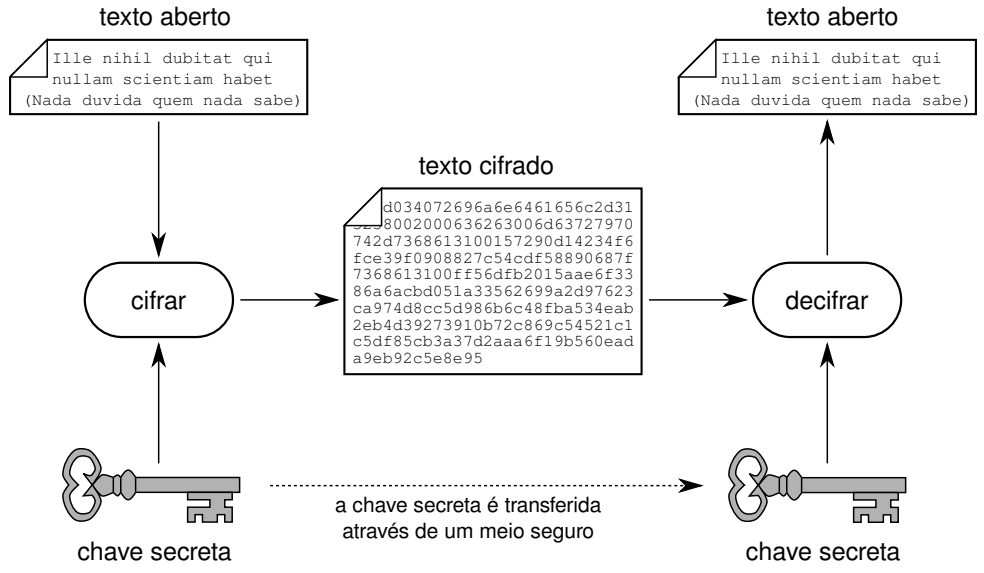


Figura 27.2: Criptografia simétrica.

Os criptossistemas simétricos são muito rápidos e bastante eficientes para a cifragem de grandes volumes de dados, como arquivos em um disco rígido ou o tráfego em uma conexão de rede. Entretanto, se a informação cifrada tiver de ser enviada a outro usuário, a chave criptográfica secreta usada terá de ser transmitida a ele através de algum meio seguro, para mantê-la secreta. Esse problema é conhecido como *o problema da distribuição de chaves*, e será discutido na Seção 27.5.

27.4.1 Cifradores de substituição e de transposição

De acordo com as operações usadas para cifrar os dados, os cifradores simétricos podem ser baseados em operações de *substituição* ou de *transposição*. Os **cifradores de substituição** se baseiam na substituição de caracteres por outros caracteres usando tabelas de substituição (ou *alfabetos*). Esses cifradores podem ser **monoalfabéticos**, quando usam uma única tabela de substituição, ou **polialfabéticos**, quando usam mais de uma tabela.

O cifrador de César é um exemplo trivial de cifrador de substituição monoalfabético. Outro exemplo dessa família, bem conhecido na cultura popular, é a linguagem *Alien*, usada em episódios da série de TV *Futurama*, cuja tabela é apresentada na Figura 27.3.

Figura 27.3: Linguagem *Alien* da série *Futurama*.

Os cifradores de substituição polialfabéticos operam com mais de uma tabela de substituição de caracteres. Um exemplo clássico de cifrador polialfabético é o cifrador de Vigenère, que foi inicialmente proposto por Giovan Battista Bellaso em 1553 e refinado por Blaise de Vigenère no século XIX. Trata-se de um método de cifragem que combina vários cifradores de César em sequência. As operações de cifragem/decifragem usam uma tabela denominada *tabula rasa*, apresentada na Figura 27.4.

Nesse cifrador, para cifrar uma mensagem, primeiro se escolhe uma palavra-chave qualquer, que é repetida até ter o mesmo comprimento da mensagem. Em seguida, cada caractere da mensagem original é codificado usando um cifrador de substituição específico, definido pela linha da *tabula rasa* indicada pela letra correspondente da palavra-chave.

Um exemplo de cifragem usando a palavra-chave “bicicleta” é indicado a seguir. Nele, pode-se observar que a letra “M” da mensagem aberta, combinada à letra “T” da palavra-chave, gera a letra “F” na mensagem cifrada.

Mensagem aberta	ATACARE**M**OS AO AMANHECER DE SEXTA-FEIRA
Palavra-chave	BICICLE**T**AB IC ICLETABIC IC LETAB ICICL
Mensagem cifrada	BBCKCCI**F**OT IQ IOLRAEDMT LG DIQTB-NGQTL

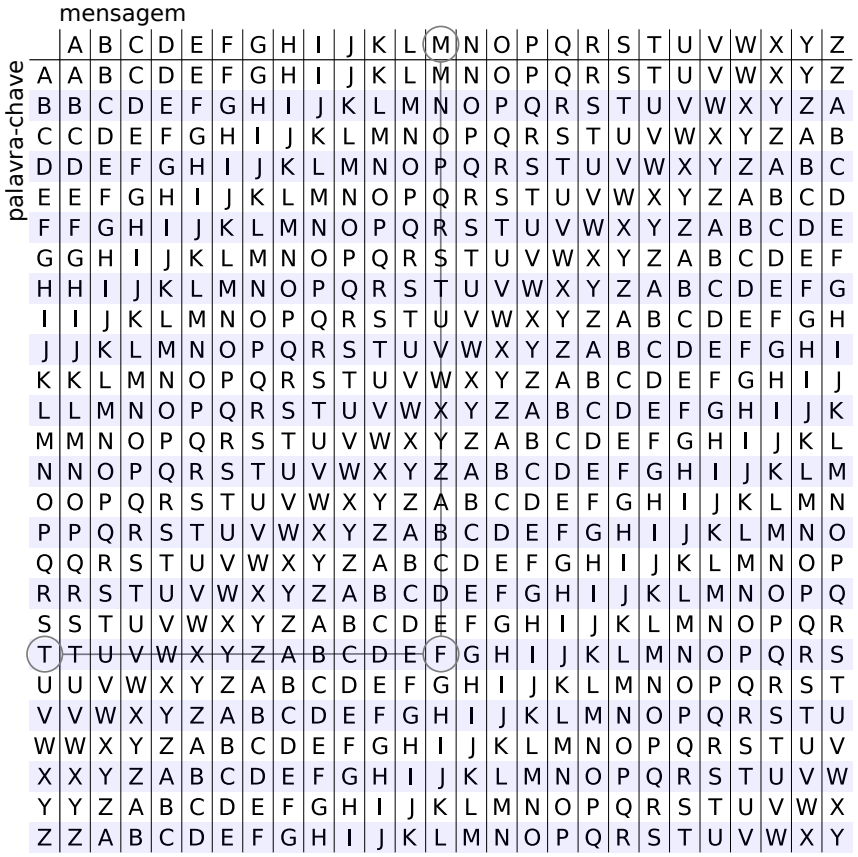


Figura 27.4: Tabula rasa do cifrador de Vigenère.

No cifrador de Vigenère, uma senha com n caracteres distintos irá usar n linhas distintas da *tabula rasa*. Considerando um alfabeto com 26 letras, teremos um espaço de chaves de 26^n, o que é bastante respeitável para a época em que o esquema foi usado.

Os cifradores de substituição podem ainda ser **monográficos** ou **poligráficos**, conforme cifrem caracteres individuais ou grupos de caracteres. Os cifradores de César e de Vigenère são monográficos. O cifrador *Playfair*, que trata o texto aberto em grupos de duas letras, é um bom exemplo de cifrador poligráfico.

Por outro lado, os **cifradores de transposição** (ou permutação) têm como mecanismo básico a troca de posição (ou embaralhamento) dos caracteres que compõem uma mensagem, sem substituí-los. O objetivo básico da operação de transposição é espalhar a informação aberta em toda a extensão do texto cifrado.

Um exemplo clássico de cifrador de transposição é o algoritmo *Rail Fence*, no qual os caracteres da mensagem aberta são distribuídos em várias linhas de uma "cerca" imaginária. Por exemplo, considerando a mesma mensagem do exemplo anterior ("Atacaremos ao amanhecer de sexta-feira") e uma cerca com 4 linhas ($k = 4$), teríamos a seguinte distribuição de caracteres na cerca:

```
A . . . . . E . . . . . A . . . . . C . . . . . E . . . . . I . .
. T . . . R . M . . . O . M . . . E . E . . . S . X . . . E . R .
. . A . A . . . O . A . . . A . H . . . R . E . . . T . F . . . A
. . . C . . . . . S . . . . . N . . . . . D . . . . . A . . . . .
```

A mensagem cifrada é obtida ao percorrer a cerca linha após linha: `AEACEITRMOMEESXERAAOAAHRETFACSNDA`. É interessante observar que os caracteres da

mensagem cifrada são os mesmos da mensagem aberta; uma análise de frequência dos caracteres poderá auxiliar a inferir qual a língua usada no texto, mas será pouco útil para identificar as posições dos caracteres na mensagem original.

Algoritmos de cifragem por substituição e por transposição raramente são utilizados isoladamente. Cifradores simétricos modernos, como o 3DES, AES e outros, são construídos usando vários blocos de substituição e de transposição aplicados de forma alternada, para aumentar a resistência do criptossistema [Stamp, 2011].

27.4.2 Cifradores de fluxo e de bloco

De acordo com a forma de agrupar os dados a cifrar, os cifradores simétricos podem ser classificados em dois grandes grupos: os *cifradores de fluxo* e os *cifradores de bloco*. Os **cifradores de fluxo** (*stream ciphers*) cifram cada byte da mensagem aberta em sequência, produzindo um byte cifrado como saída. Por essa característica sequencial, esses cifradores são importantes para aplicações de mídia em tempo real, como VoIP (voz sobre IP) e comunicações em redes celulares. Exemplos típicos de cifradores de fluxo incluem o RC4, usado até pouco tempo atrás nas redes sem fio, e o A5/1, usado para cifrar fluxo de voz em telefones GSM.

A maioria dos cifradores de fluxo funciona de forma similar: um fluxo contínuo de bytes aleatórios (*keystream*) é gerado por um bloco PRNG (*Pseudo-Random Number Generator*) a partir de uma semente que é a chave simétrica, ou calculada a partir dela. Cada byte desse fluxo é combinado com um byte do fluxo de dados aberto, para produzir um byte do fluxo cifrado. A combinação entre os fluxos geralmente é feita usando a operação XOR, de forma similar ao cifrador de Vernam. No lado do receptor, o mesmo bloco PRNG, inicializado com a mesma semente, refaz a operação XOR para decifrar o fluxo de dados. A figura 27.5 ilustra esse funcionamento.

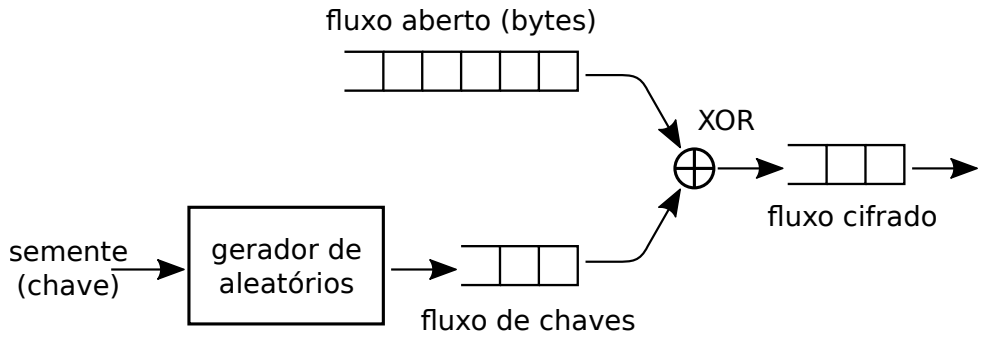


Figura 27.5: Funcionamento básico de um cifrador por fluxo

Como seu próprio nome diz, os **cifradores de bloco** (*block ciphers*) cifram os dados em blocos de mesmo tamanho, geralmente entre 64 e 128 bits. Os dados a serem cifrados são divididos em blocos e o algoritmo de cifragem é aplicado a cada bloco, até o final dos dados. Caso o último bloco não esteja completo, bits de preenchimento (*padding*) são geralmente adicionados para completá-lo. A operação em blocos provê a estes algoritmos uma maior eficiência para cifrar grandes volumes de dados, como tráfego de rede e arquivos em disco. Exemplos comuns de cifradores de bloco incluem o AES (*Advanced Encryption Standard*) e o DES/3DES (*Data Encryption Standard*).

O *modo de operação* de um cifrador de blocos define a forma como algoritmo percorre e considera os blocos de dados a cifrar. Esse modo de operação pode ter

um impacto significativo na segurança do algoritmo. O modo de operação mais simples, chamado ECB (de *Electronic Codebook*), consiste em aplicar o mesmo algoritmo sobre os blocos de dados abertos em sequência, obtendo os blocos de dados cifrados correspondentes. Esse modo de operação está ilustrado na Figura 27.6.

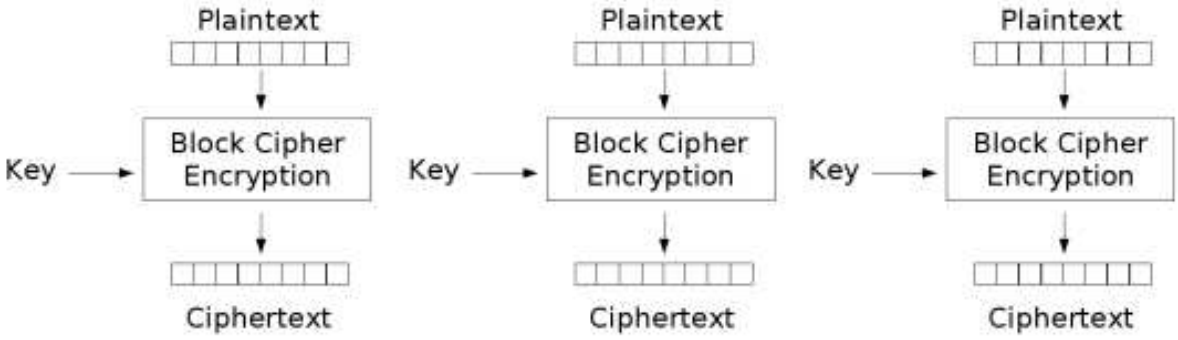


Figura 27.6: Cifragem por blocos em modo ECB [Wikipedia, 2018]

O modo de operação ECB preserva uma forte correlação entre os trechos da mensagem aberta e da mensagem cifrada, o que pode ser indesejável. A figura 27.7 demonstra o efeito dessa correlação indesejada na cifragem por blocos em modo ECB aplicada aos pixels de uma imagem.

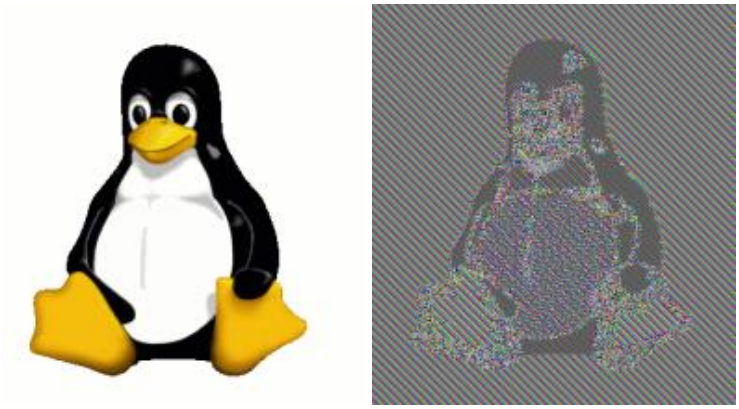


Figura 27.7: Cifragem por blocos em modo ECB: imagem aberta (à esquerda); imagem cifrada (à direita) [Wikipedia, 2018]

Para evitar a correlação direta entre blocos da entrada e da saída, cifradores de bloco modernos usam modos de operação mais sofisticados, que combinam blocos entre si. Um modo de operação bem simples com essa característica é o CBC (*Cipher Block Chaining*), no qual a saída do primeiro bloco é combinada (XOR) com a entrada do segundo bloco e assim por diante, como ilustrado na Figura 27.8. Nessa figura, o IV (*Initialization Vector*) corresponde a um bloco aleatório adicional, que deve ser conhecido ao cifrar e decifrar a mensagem. A Figura 27.9 demonstra o efeito do modo de operação CBC sobre a cifragem em blocos de uma imagem.

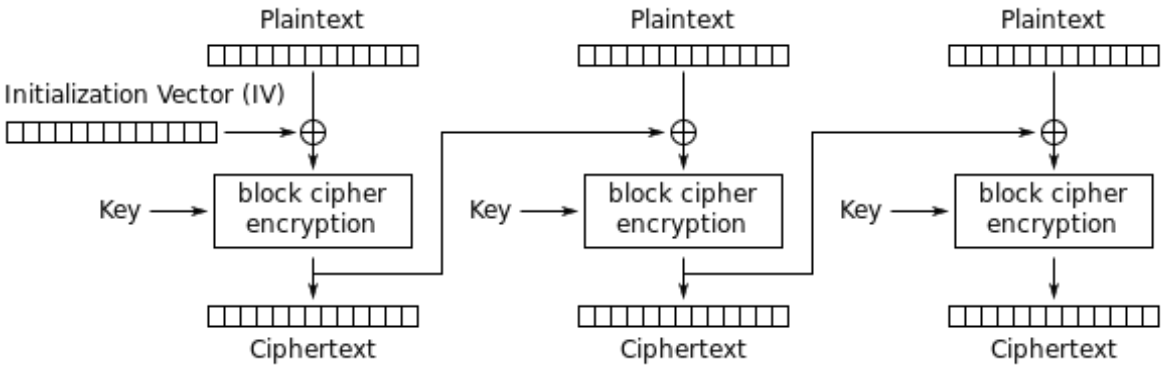


Figura 27.8: Cifragem por blocos em modo CBC [Wikipedia, 2018]

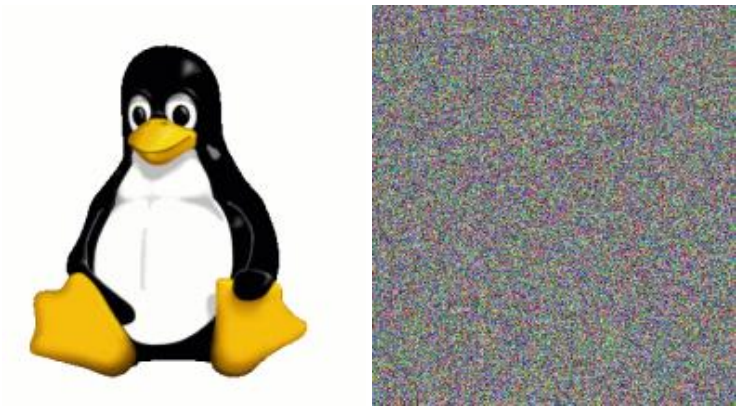


Figura 27.9: Cifragem por blocos em modo CBC: imagem aberta (à esquerda); imagem cifrada (à direita) [Wikipedia, 2018]

27.5 O acordo de chaves de Diffie-Hellman-Merkle

Um dos principais problemas no uso da criptografia simétrica para a criação de um canal de comunicação segura é a troca de chaves, ou seja, o estabelecimento de um segredo comum entre os interlocutores. Caso eles não estejam fisicamente próximos, criar uma senha secreta comum, ou substituir uma senha comprometida, pode ser um processo complicado e demorado.

O protocolo de troca de chaves de Diffie-Hellman-Merkle (*Diffie-Hellman-Merkle Key Exchange Protocol*) [Schneier, 1996; Stallings, 2011] foi proposto em 1976. Ele permite estabelecer uma chave secreta comum entre duas entidades distantes, mesmo usando uma rede insegura. Um atacante que estiver observando o tráfego de rede não poderá inferir a chave secreta a partir das mensagens em trânsito capturadas. Esse protocolo é baseado em aritmética inteira modular e constitui um exemplo muito interessante e didático dos mecanismos básicos de funcionamento da criptografia assimétrica.

Considere-se um sistema com três usuários: Alice e Bob[2] são usuários honestos que desejam se comunicar de forma confidencial; Mallory é uma usuária desonesta, que tem acesso a todas as mensagens trocadas entre Alice e Bob e tenta descobrir seus segredos (ataque de interceptação).

[2]Textos de criptografia habitualmente usam os nomes Alice, Bob, Carol e Dave para explicar algoritmos e protocolos criptográficos, em substituição às letras A, B, C e D. Outros usuários são frequentes, como Mallory (M), que é uma usuária maliciosa (atacante).

A troca de chaves proposta por Diffie-Hellman-Merkle ocorre conforme os passos do esquema a seguir. Sejam p um número primo e g uma raiz primitiva[3] módulo p:

passo	Alice	Mallory	Bob
1	escolhe p e g	$\xrightarrow{(p,g)}$	recebe p e g
2	escolhe a secreto		escolhe b secreto
3	$A = g^a mod\ p$		$B = g^b mod\ p$
4	envia A	$\xrightarrow{A}$	recebe A
5	recebe B	$\xrightarrow{B}$	envia B
6	$k = B^a mod\ p = g^{ba} mod\ p$		$k = A^b mod\ p = g^{ab} mod\ p$
7	$m' = \{m\}_k$	$\xrightarrow{m'}$	$m = \{m'\}_k^{-1}$

Como $g^{ba} mod\ p = g^{ab} mod\ p = k$, após os passos 1–6 do protocolo Alice e Bob possuem uma chave secreta comum k, que pode ser usada para cifrar e decifrar mensagens (passo 7). Durante o estabelecimento da chave secreta k, a usuária Mallory pôde observar as trocas de mensagens entre Alice e Bob e obter as seguintes informações:

- O número primo p
- O número gerador g
- $A = g^a mod\ p$ (aqui chamado *chave pública* de Alice)
- $B = g^b mod\ p$ (aqui chamado *chave pública* de Bob)

Para calcular a chave secreta k, Mallory precisará encontrar a na equação $A = g^a mod\ p$ ou b na equação $B = g^b mod\ p$. Esse cálculo é denominado *problema do logaritmo discreto* e não possui nenhuma solução eficiente conhecida: a solução por força bruta tem complexidade exponencial no tempo, em função do número de dígitos de p; o melhor algoritmo conhecido tem complexidade temporal subexponencial.

Portanto, encontrar a ou b a partir dos dados capturados da rede por Mallory torna-se impraticável se o número primo p for muito grande. Por exemplo, caso seja usado o seguinte número primo de Mersenne[4]:

$$p = 2^{127} - 1 = 170.141.183.460.469.231.731.687.303.715.884.105.727$$

[3] Uma raiz primitiva módulo p é um número inteiro positivo com certas propriedades específicas em aritmética modular.

[4] Um *número primo de Mersenne* é um número primo de forma $N_m = 2^m - 1$ com $m \geq 1$. Esta família de números primos tem propriedades interessantes para a construção de algoritmos de criptografia e geradores de números aleatórios.

o número de passos necessários para encontrar o logaritmo discreto seria aproximadamente de $\sqrt{p} = 13 \times 10^{18}$, usando o melhor algoritmo conhecido. Um computador que calcule um bilhão (10^9) de tentativas por segundo levaria 413 anos para testar todas as possibilidades!

Apesar de ser robusto em relação ao segredo da chave, o protocolo de Diffie-Hellman-Merkle é suscetível a ataques do tipo *man-in-the-middle* (ataque de modificação). Se Mallory puder modificar as mensagens em trânsito, substituindo os valores de p, g, A e B por valores que ela escolher, ela poderá estabelecer uma chave secreta $Alice \rightleftharpoons Mallory$ e outra chave secreta $Mallory \rightleftharpoons Bob$, sem que Alice e Bob percebam. Há versões modificadas do protocolo que resolvem este problema [Stamp, 2011].

27.6 Criptografia assimétrica

O protocolo de acordo de chaves de Diffie-Hellmann (Seção 27.5) revolucionou a criptografia em 1976, ao criar a família de **criptossistemas assimétricos**. Os algoritmos assimétricos se caracterizam pelo uso de um par de chaves complementares: uma **chave pública** *kp* e uma **chave privada** *kv*. Uma informação cifrada com uma determinada chave pública só poderá ser decifrada através da chave privada correspondente, e vice-versa[5]. A Figura 27.10 ilustra o funcionamento básico da criptografia assimétrica.

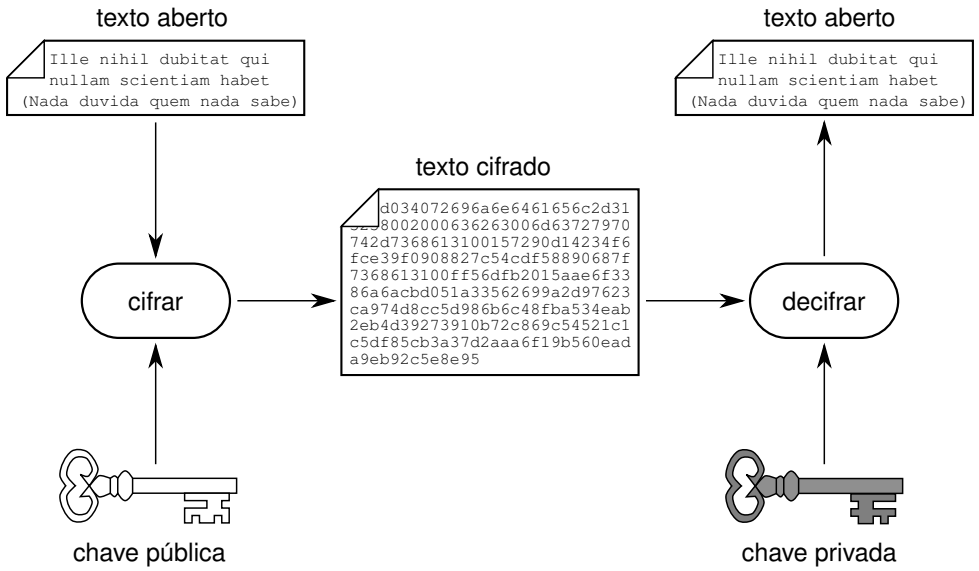


Figura 27.10: Criptografia assimétrica.

Considerando uma chave pública *kp* e sua chave privada correspondente *kv*, temos:

$$\{\{x\}_{kp}\}_k^{-1} = x \iff k = kv$$
$$\{\{x\}_{kv}\}_k^{-1} = x \iff k = kp$$

ou seja

[5]Como bem observado pelo colega Diego Aranha (Unicamp), nem todos os algoritmos assimétricos têm chaves reversíveis, ou seja, o vice-versa não é aplicável a todos os algoritmos assimétricos.

$$x \xrightarrow{kp} x' \xrightarrow{kv} x \qquad \text{e} \qquad x \xrightarrow{kp} x' \xrightarrow{k \neq kv} y \neq x$$
$$x \xrightarrow{kv} x' \xrightarrow{kp} x \qquad \text{e} \qquad x \xrightarrow{kv} x' \xrightarrow{k \neq kp} y \neq x$$

Essas equações deixam claro que as chaves pública e privada estão fortemente relacionadas: para cada chave pública há uma única chave privada correspondente, e vice-versa. Como o próprio nome diz, geralmente as chaves públicas são amplamente conhecidas e divulgadas (por exemplo, em uma página Web ou um repositório de chaves públicas), enquanto as chaves privadas correspondentes são mantidas em segredo por seus proprietários. Por razões óbvias, não é possível calcular a chave privada a partir de sua chave pública.

Além do algoritmo de *Diffie-Hellman*, apresentado na Seção 27.5, outros criptossistemas assimétricos famosos são o RSA (*Rivest-Shamir-Adleman*), que é baseado na fatoração do produto de número primos, e o ElGamal, baseado no cálculo de logaritmos discretos [Stamp, 2011].

Um exemplo prático de uso da criptografia assimétrica é mostrado na Figura 27.11. Nele, a usuária Alice deseja enviar um documento cifrado ao usuário Bob. Para tal, Alice busca a chave pública de Bob previamente divulgada em um chaveiro público (que pode ser um servidor Web, por exemplo) e a usa para cifrar o documento que será enviado a Bob. Somente Bob poderá decifrar esse documento, pois só ele possui a chave privada correspondente à chave pública usada para cifrá-lo. Outros usuários poderão até ter acesso ao documento cifrado, mas não conseguirão decifrá-lo.

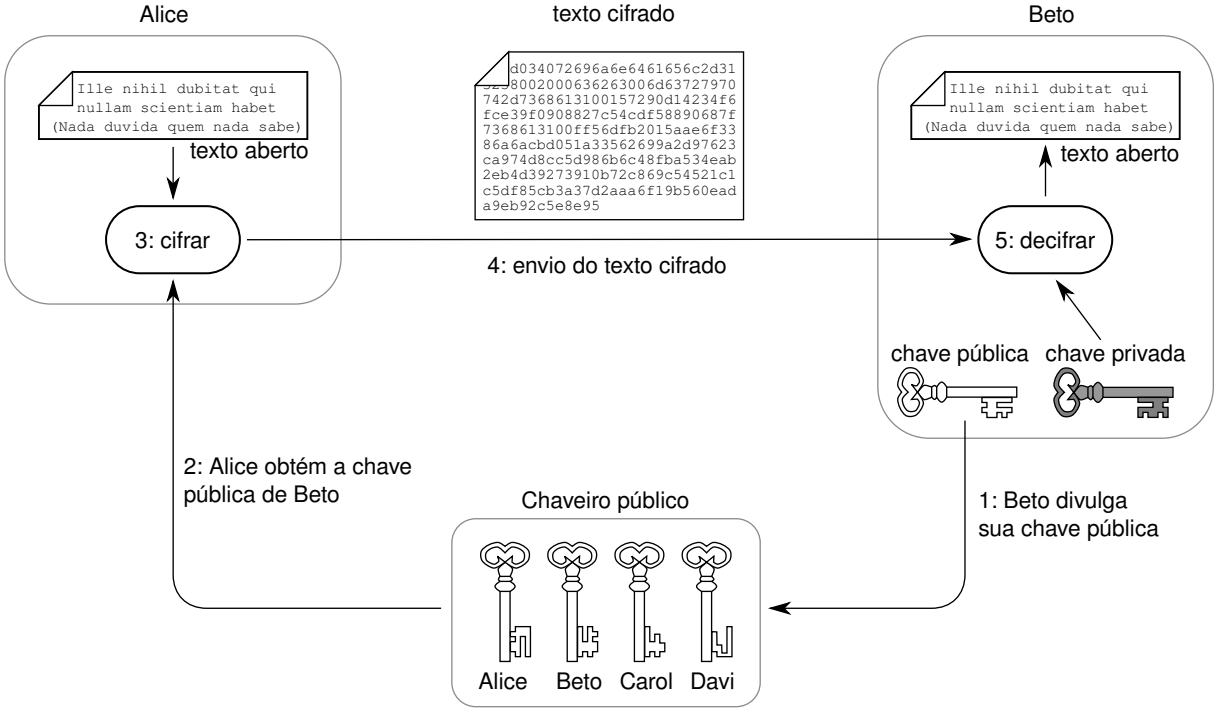


Figura 27.11: Exemplo de uso da criptografia assimétrica.

A criptografia assimétrica também pode ser usada para identificar a autoria de um documento. Por exemplo, se Alice criar um documento e cifrá-lo com sua chave privada, qualquer usuário que tiver acesso ao documento poderá decifrá-lo e lê-lo, pois a chave pública de Alice está publicamente acessível. Todavia, o fato do documento

poder ser decifrado usando a chave pública de Alice significa que ela é a autora legítima do mesmo, pois só ela teria acesso à chave privada que foi usada para cifrá-lo. Esse mecanismo é usado na criação das *assinaturas digitais* (Seção 27.9).

A Tabela 27.1 traz uma análise comparativa das principais características dos cifradores simétricos e assimétricos.

Cifrador	**Simétrico**	**Assimétrico**
Chaves	Uma única chave para cifrar e decifrar.	Chaves complementares para cifrar e decifrar.
Tamanho das chaves	Pequena (AES: 64 a 256 bits).	Grande (RSA: 2.048 a 15.360 bits).
Tamanho dos dados	Qualquer (podem ser tratados em blocos ou em fluxo).	No máximo o tamanho da chave, menos alguns bytes de *padding*.
Velocidade	Alta (centenas de MBytes/s em um PC típico).	Baixa (centenas de KBytes/s em um PC típico).
Uso	Cifragem de grandes quantidades de dados (tráfego de rede, arquivos, áudio, etc).	Cifragem de pequenas quantidades de dados (troca de chaves, assinaturas digitais).
Exemplos	RC4, A/51, DES, 3DES, AES.	Diffie-Hellman, RSA, ElGamal, ECC (Curvas Elípticas).

Tabela 27.1: Quadro comparativo de famílias de cifradores.

27.7 Criptografia híbrida

Embora sejam mais versáteis, os algoritmos assimétricos costumam exigir muito mais processamento que os algoritmos simétricos equivalentes. Além disso, eles necessitam de chaves bem maiores que os algoritmos simétricos e geralmente só podem cifrar informações pequenas (menores que o tamanho da chave). Por isso, muitas vezes os algoritmos assimétricos são usados em associação com os simétricos. Por exemplo, os protocolos de rede seguros baseados em TLS (*Transport Layer Security*), como o SSH e HTTPS, usam criptografia assimétrica somente durante o início de cada conexão, para definir uma chave simétrica comum entre os dois computadores que se comunicam. Essa chave simétrica, chamada *chave de sessão*, é então usada para cifrar/decifrar os dados trocados entre os dois computadores durante aquela conexão, sendo descartada quando a sessão encerra.

O esquema a seguir ilustra um exemplo de criptografia híbrida: a criptografia assimétrica é usada para definir uma chave de sessão comum entre dois usuários. Em seguida, essa chave de sessão é usada para cifrar e decifrar as mensagens trocadas entre eles, usando criptografia simétrica.

Passo	Alice	canal	Bob	significado
1	$k = random()$			sorteia uma chave secreta k
2	$k' = \{k\}_{kp(Bob)}$			cifra a chave k usando $kp(Bob)$
3	$k'' = \{k'\}_{kv(Alice)}$			cifra k' usando $kv(Alice)$
4	$k'' \longrightarrow \cdots$	k''	$\cdots \longrightarrow k''$	envia a chave cifrada k''
5			$k' = \{k''\}^{-1}_{kp(Alice)}$	decifra k'' usando $kp(Alice)$
6			$k = \{k'\}^{-1}_{kv(Bob)}$	decifra k' usando $kv(Bob)$, obtém k
7	$m' = \{m\}_k$			cifra mensagem m usando k
8	$m' \longrightarrow \cdots$	m'	$\cdots \longrightarrow m'$	envia mensagem cifrada m'
9			$m = \{m'\}^{-1}_k$	decifra a mensagem m' usando k

Inicialmente, Alice sorteia uma chave secreta simétrica k (passo 1) e a cifra com a chave pública de Bob (passo 2), para que somente ele possa decifrá-la (garante confidencialidade). Em seguida, ela cifra k' com sua chave privada (passo 3), para que Bob tenha certeza de que a chave foi gerada por Alice (garante autenticidade). Em seguida, A chave duplamente cifrada k'' é enviada a Bob (passo 4), que a decifra (passos 5 e 6) e resgata a chave secreta k. Agora, Alice e Bob podem usar a chave de sessão k para trocar mensagens cifradas entre si (passos 7 a 9).

Se Mallory estiver capturando mensagens no canal de comunicação, ela terá acesso somente a k'' e m' (e às chaves públicas de Alice e Bob), o que não a permite descobrir a chave de sessão k nem a mensagem aberta m.

27.8 Resumo criptográfico

Um *resumo criptográfico* (*cryptographic hash*) [Menezes et al., 1996] é uma função $y = hash(x)$ que gera uma sequência de bytes y de tamanho pequeno e fixo (algumas dezenas ou centenas de bytes) a partir de um conjunto de dados x de tamanho variável aplicado como entrada. Os resumos criptográficos são frequentemente usados para identificar unicamente um arquivo ou outra informação digital, ou para atestar sua integridade: caso o conteúdo de um documento digital seja modificado, seu resumo também será alterado.

Em termos matemáticos, os resumos criptográficos são um tipo de *função unidirecional* (*one-way function*). Uma função $f(x)$ é chamada unidirecional quando seu cálculo direto ($y = f(x)$) é rápido, mas o cálculo de sua inversa ($x = f^{-1}(y)$) é impossível ou computacionalmente inviável. Um exemplo clássico de função unidirecional é a fatoração do produto de dois números primos muito grandes, como os usados no algoritmo RSA. Considerando a função $f(p, q) = p \times q$, onde p e q são inteiros primos, calcular $y = f(p, q)$ é simples e rápido, mesmo se p e q forem grandes. Entretanto, fatorar

y para obter de volta os primos p e q pode ser computacionalmente inviável, se y tiver muitos dígitos[6].

Os algoritmos de resumo criptográfico mais conhecidos e utilizados atualmente são os da família SHA (*Secure Hash Algorithms*). Os algoritmos MD5 e SHA1 foram muito utilizados, mas se mostraram inseguros a colisões e não devem mais ser utilizadom para aplicações criptográficas; seu uso continua viável em outras aplicações, como a detecção de réplicas de arquivos [Menezes et al., 1996; Stamp, 2011]. No Linux, comandos como `md5sum` e `sha1sum` permitem calcular respectivamente resumos criptográficos de arquivos:

```
~:> md5sum livro.*
371f456d68720a3c0ba5950fe2708d37  livro.pdf
d4a593dc3d44f6eae54fc62600581b11  livro.tex

~:> sha1sum livro.*
9664a393b533d5d82cfe505aa3ca12410aa1f3b7  livro.pdf
d5bd8d809bb234ba8d2289d4fa13c319e227ac25  livro.tex

~:> sha224sum livro.*
36049e03abf47df178593f79c3cdd0c018406232a0f300d872351631  livro.pdf
edac4154fe0263da86befa8d5072046b96b75c2f91764cc6b5b2f5c0  livro.tex

~:> sha256sum livro.*
c5fc543d1758301feacdc5c6bfd7bc12ef87036fbc589a902856d306cb999d50  livro.pdf
da5075006c6f951e40c9e99cef3218d8a2d16db28e746d0f4e4b18cf365a8099  livro.tex
```

Uma boa função de resumo criptográfico deve gerar sempre a mesma saída para a mesma entrada ($hash(m_1) = hash(m_2) \iff m_1 = m_2$) e saídas diferentes para entradas diferentes (($hash(m_1) \neq hash(m_2) \iff m_1 \neq m_2$). No entanto, como o número de bytes do resumo é pequeno, podem ocorrer *colisões*: duas entradas distintas m_1 e m_2 gerando o mesmo resumo ($m_1 \neq m_2$ mas $hash(m_1) = hash(m_2)$). Idealmente, uma função de *hash* criptográfico deve apresentar as seguintes propriedades:

Determinismo: para uma dada entrada m, a saída é sempre a mesma.

Rapidez: o cálculo de $x = hash(m)$ é rápido para qualquer m.

Resistência à pré-imagem: dado um valor de x, é difícil encontrar m tal que $x = hash(m)$ (ou seja, a função é difícil de inverter).

Resistência à colisão: é difícil encontrar duas mensagens quaisquer $m_1 \neq m_2$ tal que $hash(m_1) = hash(m_2)$.

Espalhamento: uma modificação em um trecho específico dos dados de entrada m gera modificações em várias partes do resumo $hash(m)$.

Sensibilidade: uma pequena mudança nos dados de entrada m (mesmo um só bit) gera mudanças significativas no resumo $hash(m)$.

[6] Em 2014, um grupo de pesquisadores conseguiu fatorar o inteiro $2^{1199} - 1$ (que tem 361 dígitos), em um projeto que consumiu cerca de 7.500 anos-CPU.

27.9 Assinatura digital

Os mecanismos de criptografia assimétrica e resumos criptográficos previamente apresentados permitem efetuar a *assinatura digital* de documentos eletrônicos. A assinatura digital é uma forma de verificar a autoria e integridade de um documento, sendo por isso o mecanismo básico utilizado na construção dos *certificados digitais*, amplamente empregados para a autenticação de servidores na Internet.

Em termos gerais, a assinatura digital de um documento é um resumo digital do mesmo, cifrado usando a chave privada de seu autor (ou de quem o está assinando). Sendo um documento d emitido pelo usuário u, sua assinatura digital $s(d, u)$ é definida por:

$$s(d, u) = \{\ hash(d)\ \}_{kv(u)}$$

onde $hash(x)$ é uma função de resumo criptográfico conhecida, $\{x\}_k$ indica a cifragem de x usando uma chave k e $kv(u)$ é a chave privada do usuário u. Para verificar a validade da assinatura, basta calcular novamente o resumo $r' = hash(d)$ e compará-lo com o resumo obtido da assinatura, decifrada usando a chave pública de u ($r'' = \{s\}^{-1}_{kp(u)}$). Se ambos forem iguais ($r' = r''$), o documento foi realmente assinado por u e está íntegro, ou seja, não foi modificado desde que u o assinou [Menezes et al., 1996].

A Figura 27.12 ilustra o processo de assinatura digital e verificação de um documento. Os passos do processo são:

1. Alice divulga sua chave pública kp_a em um repositório acessível publicamente;
2. Alice calcula o resumo digital r do documento d a ser assinado;
3. Alice cifra o resumo r usando sua chave privada kv_a, obtendo uma assinatura digital s;
4. A assinatura s e o documento original d, em conjunto, constituem o documento assinado por Alice: $[d, s]$;
5. Bob obtém o documento assinado por Alice ($[d', s']$, com $d' = d$ e $s' = s$ se ambos estiverem íntegros);
6. Bob recalcula o resumo digital $r' = hash(d')$ do documento, usando o mesmo algoritmo empregado por Alice;
7. Bob obtém a chave pública kp_a de Alice e a usa para decifrar a assinatura s' do documento, obtendo um resumo r'' ($r'' = r$ se s foi realmente cifrado com a chave kv_a e se $s' = s$) ;
8. Bob compara o resumo r' do documento com o resumo r'' obtido da assinatura digital; se ambos forem iguais ($r' = r''$), o documento foi assinado por Alice e está íntegro, assim como sua assinatura.

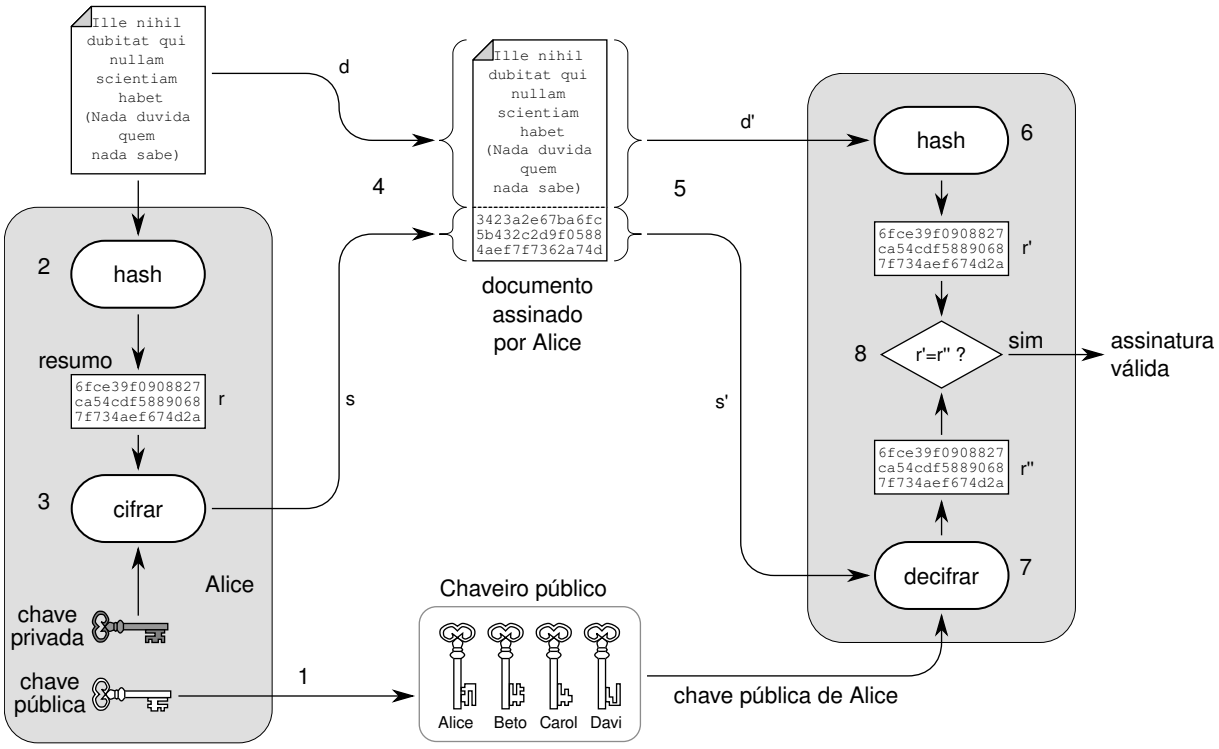


Figura 27.12: Assinatura e verificação de uma assinatura digital.

27.10 Certificado de chave pública

A identificação confiável do proprietário de uma chave pública é fundamental para o funcionamento correto das técnicas de criptografia assimétrica e de assinatura digital. Uma chave pública é composta por uma mera sequência de bytes que não permite a identificação direta de seu proprietário. Por isso, torna-se necessária uma estrutura complementar para fazer essa identificação. A associação entre chaves públicas e seus respectivos proprietários é realizada através dos *certificados digitais*. Um certificado digital é um documento digital assinado, composto das seguintes partes [Menezes et al., 1996]:

- Identidade do proprietário do certificado (nome, endereço, e-mail, URL, número IP e/ou outras informações que permitam identificá-lo unicamente)[7];
- Chave pública do proprietário do certificado;
- Identificação da entidade que emitiu/assinou o certificado;
- Outras informações, como período de validade do certificado, algoritmos de criptografia e resumos utilizados, etc.;
- Uma ou mais assinaturas digitais do conteúdo, emitidas por entidades consideradas confiáveis pelos usuários do certificado.

[7]Deve-se ressaltar que um certificado pode pertencer a um usuário humano, a um sistema computacional ou qualquer módulo de software que precise ser identificado de forma inequívoca.

Dessa forma, um certificado digital "amarra" uma identidade a uma chave pública. Para verificar a validade de um certificado, basta usar a chave pública da entidade que o assinou. Existem vários tipos de certificados digitais com seus formatos e conteúdos próprios, sendo os certificados PGP e X.509 aqueles mais difundidos [Mollin, 2000]. Os certificados no padrão X509 são extensivamente utilizados na Internet para a autenticação de chaves públicas de servidores Web, de e-mail, etc. Um exemplo de certificado X.509, destacando sua estrutura básica e principais componentes, é apresentado na Figura 27.13.

```
Certificate Data:
   Version: 3 (0x2)
   Serial Number: 05:f1:3c:83:7e:0e:bb:86:ed:f8:c4:9b
   Issuer: C=BE, O=GlobalSign nv-sa, CN=GlobalSign Extended Validation CA-SHA256-G3
   Validity
      Not Before: Feb  7 12:41:03 2017 GMT                          informações básicas
      Not After : May  9 23:59:59 2018 GMT
   Subject: businessCategory=Private Organization/serialNumber=00.000.000/7297-44/
            jurisdictionC=BR, C=BR, ST=Distrito Federal, L=Brasilia/
            street=ST STN SN QD 716 CONJ C EDIF SEDE IV ANDAR 1 ASA NORTE,   proprietário
            OU=DITEC, O=Banco do Brasil S.A., CN=www2.bancobrasil.com.br     do certificado
   Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
         Public-Key: (2048 bit)
         Modulus:
            00:db:4a:0e:92:da:5b:f3:38:3f:d5:63:9d:6d:f9:          chave pública
            91:6c:16:fc:24:84:28:e8:aa:86:aa:9c:a3:aa:1a:          do proprietário
            2e:b6:09:74:6a:f8:1e:31:4a:60:81:0f:ac:76:59:          do certificado
            ... (linhas omitidas)
            8e:0b
         Exponent: 65537 (0x10001)
   X509v3 extensions:
      X509v3 Key Usage: critical
          Digital Signature, Key Encipherment                      campos opcionais
      Authority Information Access:
          CA Issuers - URI:http://secure.globalsign.com/cacert/gsextendvalsha2g3r3.crt
          OCSP - URI:http://ocsp2.globalsign.com/gsextendvalsha2g3r3
      X509v3 Extended Key Usage:
          TLS Web Server Authentication, TLS Web Client Authentication
Signature Algorithm: sha256WithRSAEncryption
   94:8e:14:c6:38:30:78:77:80:fc:92:f1:5b:8b:72:6a:b6:b6:
   95:66:c5:7b:ba:be:51:a4:b8:8a:f5:37:0a:4a:74:4d:82:27:          assinatura do emissor
   ... (linhas omitidas)
   b6:44:e8:8c
```

Figura 27.13: Certificado digital no padrão X.509.

27.11 Infraestrutura de chaves públicas

Todo certificado deve ser assinado por alguma entidade considerada confiável pelos usuários do sistema. Essas entidades são normalmente denominadas *Autoridades*

Certificadoras (AC ou CA – *Certification Authorities*). Como as chaves públicas das ACs devem ser usadas para verificar a validade de um certificado, surge um problema: como garantir que uma chave pública realmente pertence a uma dada autoridade certificadora?

A solução para esse problema é simples: basta criar um certificado para essa AC, assinado por outra AC ainda mais confiável. Dessa forma, pode-se construir uma estrutura hierárquica de certificação, na qual a AC mais confiável (denominada "AC raiz") assina os certificados de outras ACs, e assim sucessivamente, até chegar aos certificados dos servidores, usuários e demais entidades do sistema. Uma estrutura de certificação dessa forma se chama *Infraestrutura de Chaves Públicas* (ICP ou PKI - *Public-Key Infrastructure*). Em uma ICP convencional (hierárquica), a chave pública da AC raiz deve ser conhecida de todos e é considerada íntegra [Mollin, 2000].

A Figura 27.14 traz um exemplo de infraestrutura de chaves públicas hierárquica. A chave pública AC raiz (vermelha) é usada para assinar os certificados das chaves verde e azul, e assim por diante. Reciprocamente, o certificado de chave roxo depende da confiança na chave azul, que por sua vez depende da confiança na chave vermelha. A sequência de certificados *roxo* → *azul* → *vermelho* é chamada de **cadeia de certificação** ou *cadeia de confiança*.

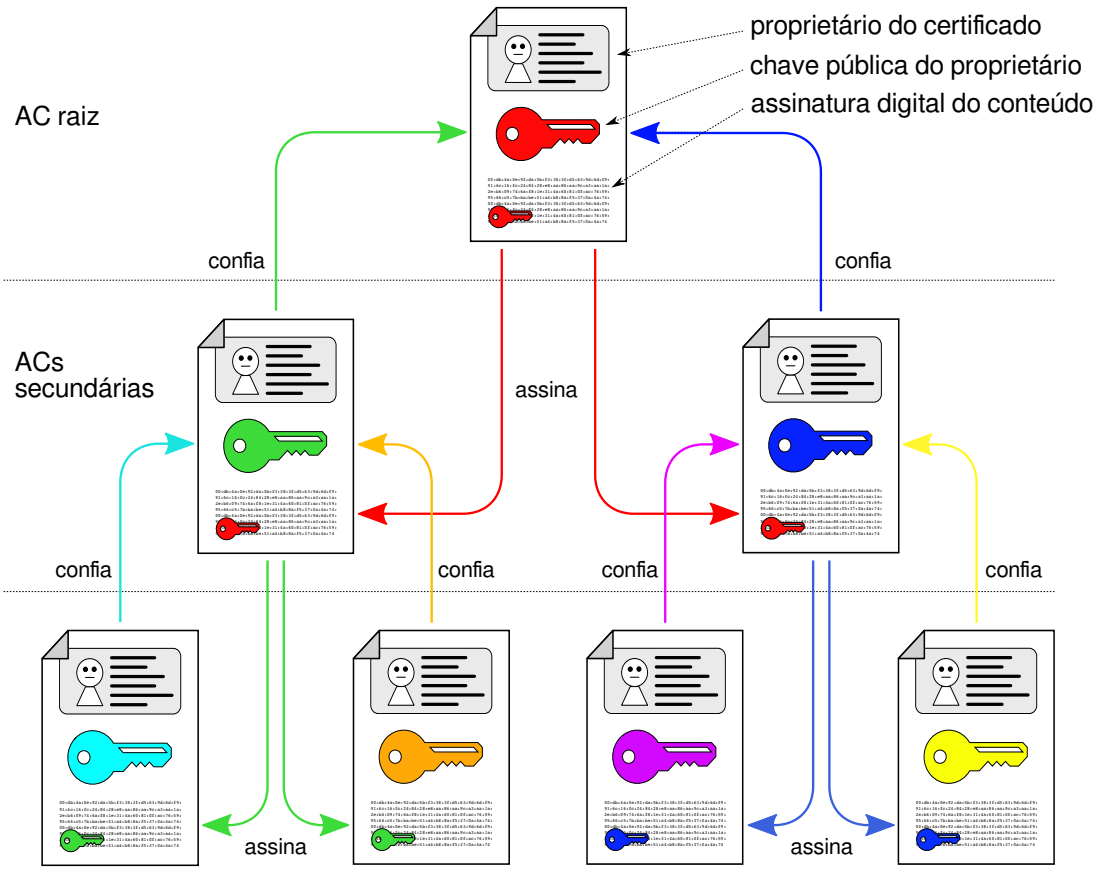


Figura 27.14: Infraestrutura de chaves públicas hierárquica.

O campo *Validity* de um certificado X509 (ver Figura 27.13) diz respeito ao seu prazo de validade, ou seja, o período de tempo em que o certificado é considerado válido. Entretanto, em algumas situações pode ser necessário **revogar certificados** antes do prazo final de validade. Casos típicos de revogação envolvem o vazamento da chave

privada do usuário ou da de alguma autoridade certificadora na cadeia de certificação que valida o certificado, ou então situações mais banais, como a cessação de atividade da empresa proprietária do certificado ou mudanças na finalidade do certificado (campos opcionais `Key Usage` na Figura 27.13).

Existem dois mecanismos básicos para revogar certificados: as CRLs - *Certificate Revocation Lists* são listas de certificados revogados mantidas pelas autoridades certificadoras, que pode ser descarregadas pelo software cliente através de um acesso HTTP. Contudo, em autoridades certificadoras populares, as CRLs podem conter muitos certificados e se tornar muito grandes. Por isso, mais recentemente foi definido o OCSP - *Online Certificate Status Protocol*, que permite ao software consultar junto à CA o status de um certificado digital específico.

Referências

A. Menezes, P. Van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

R. A. Mollin. *An Introduction to Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 2000. ISBN 1584881275.

B. Schneier. *Applied cryptography: protocols, algorithms, and source code in C, 2nd edition*. Wiley, 1996.

W. Stallings. *Cryptography and Network Security – Principles and Practice, 4th edition*. Pearson, 2011.

M. Stamp. *Information Security - Principles and Practice, 2nd edition*. Wiley, 2011.

Wikipedia. Wikipedia online enciclopedia. `http://www.wikipedia.org`, 2018.

Capítulo 28

Autenticação

O objetivo da autenticação consiste em identificar as diversas entidades de um sistema computacional. Através da autenticação, o usuário interessado em acessar o sistema comprova que ele/a realmente é quem afirma ser. Para tal podem ser usadas várias técnicas, sendo as mais relevantes apresentadas neste capítulo.

28.1 Introdução

Autenticação é o procedimento de verificar a autenticidade de uma entidade no sistema computacional, ou seja, comprovar que as informações associadas a essa entidade são verdadeiros e correspondem às informações do mundo real que elas representam, como a identidade de um usuário, o construtor de um software, a origem dos dados de uma página Web, etc.

Inicialmente, a autenticação visava apenas identificar usuários, para garantir que somente usuários previamente registrados teriam acesso ao sistema. Atualmente, em muitas circunstâncias também é necessário o oposto, ou seja, identificar o sistema para o usuário, sobretudo no caso de acessos por rede. Por exemplo, quando um usuário acessa um serviço bancário via Internet, deseja ter certeza de que o sistema acessado é realmente aquele do banco desejado, e não um sistema falso, construído para roubar seus dados bancários. Outro exemplo ocorre durante a instalação de componentes de software como *drivers*: o sistema operacional deve assegurar-se que o software a ser instalado provém de uma fonte confiável.

28.2 Usuários e grupos

A autenticação geralmente é o primeiro passo no acesso de um usuário a um sistema computacional. Caso a autenticação do usuário tenha sucesso, são criados processos para representá-lo dentro do sistema. Esses processos interagem com o usuário através da interface e executam as ações desejadas por ele dentro do sistema, ou seja, agem em nome do usuário. A presença de um ou mais processos agindo em nome de um usuário dentro do sistema é denominada uma *sessão de usuário* (*user session* ou *working session*). A sessão de usuário inicia imediatamente após a autenticação do usuário (*login* ou *logon*) e termina quando seu último processo é encerrado, na desconexão (*logout* ou *logoff*). Um sistema operacional servidor ou *desktop* típico suporta várias sessões de usuários simultaneamente.

A fim de permitir a implementação das técnicas de controle de acesso e auditoria, cada processo deve ser associado a seu respectivo usuário através de um *identificador de usuário* (UID - *User IDentifier*), geralmente um número inteiro usado como chave em uma tabela de usuários cadastrados (como o arquivo `/etc/passwd` dos sistemas UNIX). O identificador de usuário é usado pelo sistema operacional para definir o proprietário de cada entidade e recurso conhecido: processo, arquivo, área de memória, semáforo, etc. É habitual também classificar os usuários em grupos, como *professores*, *alunos*, *contabilidade*, *engenharia*, etc. Cada grupo é identificado através de um *identificador de grupo* (GID - *Group IDentifier*). A organização dos grupos de usuários pode ser hierárquica ou arbitrária. O conjunto de informações que relaciona um processo ao seu usuário e grupo é geralmente denominado *credenciais do processo*.

Normalmente, somente usuários devidamente autenticados podem ter acesso aos recursos de um sistema. Todavia, alguns recursos podem estar disponíveis abertamente, como é o caso de pastas de arquivos públicas em rede e páginas em um servidor Web público. Nestes casos, assume-se a existência de um usuário fictício "convidado" (*guest*, *nobody*, *anonymous* ou outros), ao qual são associados todos os acessos externos não autenticados e para o qual são definidas políticas de segurança específicas.

28.3 Estratégias de autenticação

As técnicas usadas para a autenticação de um usuário podem ser classificadas em três grandes grupos:

SYK – Something You Know ("algo que você sabe"): estas técnicas de autenticação são baseadas em informações conhecidas pelo usuário, como seu nome de *login* e sua senha. São consideradas técnicas de autenticação fracas, pois a informação necessária para a autenticação pode ser facilmente comunicada a outras pessoas, ou mesmo roubada.

SYH – Something You Have ("algo que você tem"): são técnicas que se baseiam na posse de alguma informação mais complexa, como um certificado digital ou uma chave criptográfica, ou algum dispositivo material, como um *smartcard*, um cartão magnético, um código de barras, etc. Embora sejam mais robustas que as técnicas SYK, estas técnicas também têm seus pontos fracos, pois dispositivos materiais, como cartões, também podem ser roubados ou copiados.

SYA – Something You Are ("algo que você é"): se baseiam em características intrinsecamente associadas ao usuário, como seus dados biométricos: impressão digital, padrão da íris, timbre de voz, etc. São técnicas mais complexas de implementar, mas são potencialmente mais robustas que as anteriores.

Muitos sistemas implementam somente a autenticação por login/senha (SYK). Sistemas mais recentes têm suporte a técnicas SYH através de *smartcards* ou a técnicas SYA usando biometria, como os sensores de impressão digital. Alguns serviços de rede, como HTTP e SSH, também podem usar autenticação pelo endereço IP do cliente (SYA) ou através de certificados digitais (SYH).

Sistemas computacionais com fortes requisitos de segurança geralmente implementam mais de uma técnica de autenticação, o que é chamado de **autenticação multifator**. Por exemplo, um sistema militar pode exigir senha e reconhecimento de

íris para o acesso de seus usuários, enquanto um sistema bancário pode exigir uma senha e o cartão emitido pelo banco. Essas técnicas também podem ser usadas de forma gradativa: uma autenticação básica é solicitada para o usuário acessar o sistema e executar serviços simples (como consultar o saldo de uma conta bancária); se ele solicitar ações consideradas críticas (como fazer transferências de dinheiro para outras contas), o sistema pode exigir mais uma autenticação, usando outra técnica.

28.4 Senhas

A grande maioria dos sistemas operacionais de propósito geral implementam a técnica de autenticação SYK baseada em *login/senha*. Na autenticação por senha, o usuário informa ao sistema seu identificador de usuário (nome de *login*) e sua senha, que normalmente é uma sequência de caracteres memorizada por ele. O sistema então compara a senha informada pelo usuário com a senha previamente registrada para ele: se ambas forem iguais, o acesso é consentido.

A autenticação por senha é simples mas muito frágil, pois implica no armazenamento das senhas "em aberto" no sistema, em um arquivo ou base de dados. Caso o arquivo ou base seja exposto devido a algum erro ou descuido, as senhas dos usuários estarão visíveis. Para evitar o risco de exposição indevida das senhas, são usadas funções unidirecionais para armazená-las, como os resumos criptográficos (Seção 27.8).

A autenticação por senhas usando um resumo criptográfico é bem simples: ao registrar a senha s de um novo usuário, o sistema calcula seu resumo ($r = hash(s)$), e o armazena. Mais tarde, quando esse usuário solicitar sua autenticação, ele informará uma senha s'; o sistema então calculará novamente seu resumo $r' = hash(s')$ e irá compará-lo ao resumo previamente armazenado ($r' = r$). Se ambos forem iguais, a senha informada pelo usuário é considerada autêntica e o acesso do usuário ao sistema é permitido. Com essa estratégia, as senhas não precisam ser armazenadas em aberto no sistema, aumentando sua segurança.

Caso um intruso tenha acesso aos resumos das senhas dos usuários, ele não conseguirá calcular de volta as senhas originais (pois o resumo foi calculado por uma função unidirecional), mas pode tentar obter as senhas indiretamente, através do **ataque do dicionário**. Nesse ataque, o invasor usa o algoritmo de resumo para cifrar palavras conhecidas ou combinações delas, comparando os resumo obtidos com aqueles presentes no arquivo de senhas. Caso detecte algum resumo coincidente, terá encontrado a senha correspondente. O ataque do dicionário permite encontrar senhas consideradas "fracas", por serem muito curtas ou baseadas em palavras conhecidas. Por isso, muitos sistemas operacionais definem políticas rígidas para as senhas, impedindo o registro de senhas óbvias ou muito curtas e restringindo o acesso ao repositório dos resumos de senhas.

Uma técnica muito utilizada em sistemas operacionais para dificultar o ataque do dicionário a *hashes* de senhas consiste em "salgar as senhas". O "sal", neste caso, é um número aleatório (*nonce*) concatenado a cada senha antes do cálculo do respectivo *hash*. Ao cadastrar uma senha, um *nonce* aleatório (o sal) é gerado e concatenado à senha e o *hash* dessa concatenação é calculado. Esse *hash* e o sal são então armazenados juntos no sistema, para uso no processo de autenticação. A verificar a senha informada por um usuário, o sal armazenado é concatenado à senha a ser verificada, o *hash* dessa concatenação é calculado e o resultado é comparado ao *hash* previamente armazenado, para autenticar o usuário. A Figura 28.1 ilustra esses procedimentos.

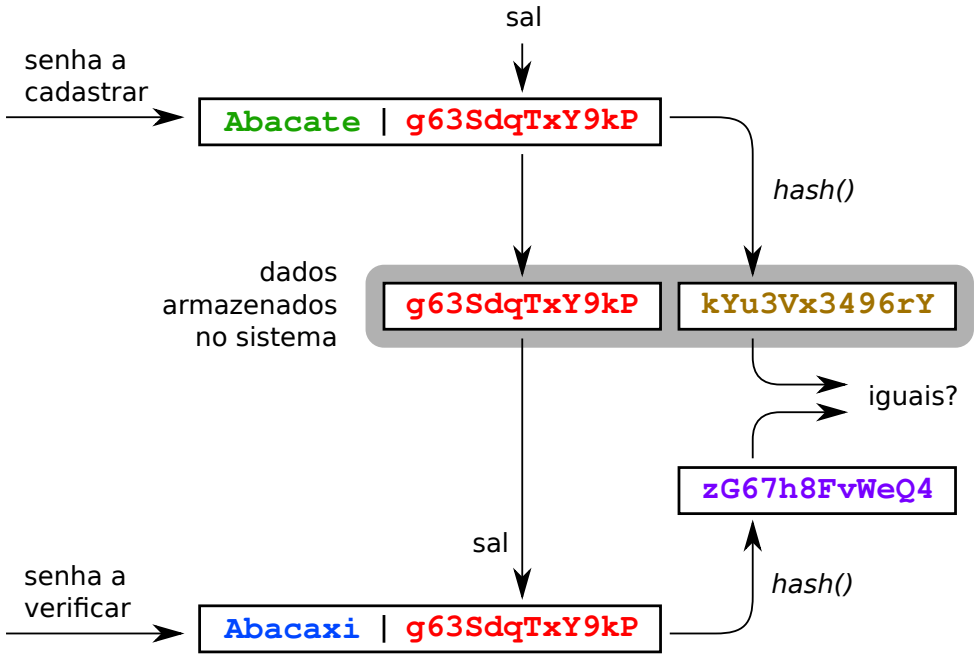


Figura 28.1: Uso de sal na proteção de senhas.

O sal protege os *hashes* das senhas por tornar impraticável o cálculo prévio de tabelas de hashes para o ataque do dicionário. Ao concatenar um sal aleatório com 64 bits de comprimento (8 bytes) a uma senha, essa combinação poderá gerar 2^{64} *hashes* distintos para a mesma senha, o que torna inviável computar e armazenar previamente todos os *hashes* possíveis para cada palavra do dicionário.

28.5 Senhas descartáveis

Um problema importante relacionado à autenticação por senhas reside no risco de roubo da senhas. Por ser uma informação estática, caso uma senha seja roubada, o malfeitor poderá usá-la enquanto o roubo não for percebido e a senha substituída. Para evitar esse problema, são propostas técnicas de senhas descartáveis (OTP - *One-Time Passwords*). Como o nome diz, uma senha descartável só pode ser usada uma única vez, perdendo sua validade após esse uso. O usuário deve então ter em mãos uma lista de senhas pré-definidas, ou uma forma de gerá-las quando necessário. Há várias formas de se produzir e usar senhas descartáveis, entre elas:

- Armazenar uma lista sequencial de senhas (ou seus resumos) no sistema e fornecer essa lista ao usuário, em papel ou outro suporte. Quando uma senha for usada com sucesso, o usuário e o sistema a eliminam de suas respectivas listas. A lista de senhas pode ser entregue ao usuário impressa, ou fornecida por outro meio, como mensagens SMS. A tabela a seguir ilustra um exemplo dessas listas de senhas, ainda usadas por alguns bancos:

```
001 342232   002 038234   003 887123   004 545698   005 323241
006 587812   007 232221   008 772633   009 123812   010 661511
011 223287   012 870910   013 865324   014 986323   015 876876
...
```

- Uma variante da lista de senhas é conhecida como *algoritmo OTP de Lamport* [Menezes et al., 1996]. Ele consiste em criar uma sequência de senhas $s_0, s_1, s_2, \cdots, s_{n-1}, s_n$ com s_0 aleatório e $s_i = hash(s_{i-1})\ \forall i > 0$, sendo $hash(x)$ uma função de resumo criptográfico conhecida:

$$\xrightarrow{random} s_0 \xrightarrow{hash} s_1 \xrightarrow{hash} s_2 \xrightarrow{hash} \cdots \xrightarrow{hash} s_{n-1} \xrightarrow{hash} s_n$$

  O valor de s_n é informado ao servidor previamente. Ao acessar o servidor, o cliente informa o valor de s_{n-1}. O servidor pode então comparar $hash(s_{n-1})$ com o valor de s_n previamente informado: se forem iguais, o cliente está autenticado e ambos podem descartar s_n. Para validar a próxima autenticação será usado s_{n-1} e assim sucessivamente. Um intruso que conseguir capturar uma senha s_i não poderá usá-la mais tarde, pois não conseguirá calcular s_{i-1} (a função $hash(x)$ não é inversível).

- Gerar senhas temporárias sob demanda, através de um dispositivo ou software externo usado pelo cliente; as senhas temporárias podem ser geradas por um algoritmo de resumo que combine uma senha pré-definida com a data/horário corrente. Dessa forma, cliente e servidor podem calcular a senha temporária de forma independente. Como o tempo é uma informação importante nesta técnica, o dispositivo ou software gerador de senhas do cliente deve estar sincronizado com o relógio do servidor. Dispositivos OTP como o mostrado na Figura 28.2 são frequentemente usados em sistemas de *Internet Banking*.

Figura 28.2: Gerador de senhas descartáveis (fotografia de Mazh3101@Wikipedia).

28.6 Técnicas biométricas

A biometria (*biometrics*) consiste em usar características físicas ou comportamentais de um indivíduo, como suas impressões digitais ou seu timbre de voz, para identificá-lo unicamente perante o sistema. Diversas características podem ser usadas para a autenticação biométrica; no entanto, elas devem obedecer a um conjunto de princípios básicos [Jain et al., 2004]:

Universalidade: a característica biométrica deve estar presente em todos os indivíduos que possam vir a ser autenticados;

Singularidade: (ou unicidade) dois indivíduos quaisquer devem apresentar valores distintos para a característica em questão;

Permanência: a característica não deve mudar ao longo do tempo, ou ao menos não deve mudar de forma abrupta;

Mensurabilidade: a característica em questão deve ser facilmente mensurável em termos quantitativos.

As características biométricas usadas em autenticação podem ser *físicas* ou *comportamentais*. Como características físicas são consideradas, por exemplo, o DNA, a geometria das mãos, do rosto ou das orelhas, impressões digitais, o padrão da íris (padrões na parte colorida do olho) ou da retina (padrões de vasos sanguíneos no fundo do olho). Como características comportamentais são consideradas a assinatura, o padrão de voz e a dinâmica de digitação (intervalos de tempo entre teclas digitadas), por exemplo.

Os sistemas mais populares de autenticação biométrica atualmente são os baseados em impressões digitais e no padrão de íris. Esses sistemas são considerados confiáveis, por apresentarem taxas de erro relativamente baixas, custo de implantação/operação baixo e facilidade de coleta dos dados biométricos. A Figura 28.3 apresenta alguns exemplos de características biométricas empregadas nos sistemas atuais.

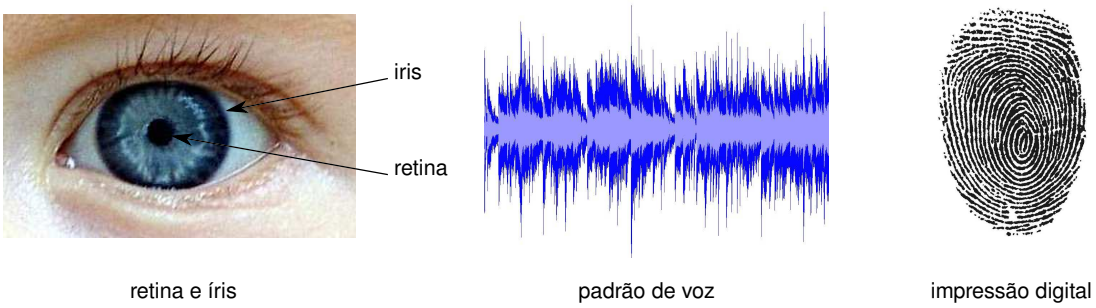


Figura 28.3: Exemplo de características biométricas.

Um sistema biométrico típico é composto de um *sensor*, responsável por capturar dados biométricos de uma pessoa; um *extrator de características*, que processa os dados do sensor para extrair suas características mais relevantes; um *comparador*, cuja função é comparar as características extraídas do indivíduo sob análise com dados previamente armazenados, e um *banco de dados* contendo as características biométricas dos usuários registrados no sistema [Jain et al., 2004].

O sistema biométrico pode funcionar de três modos: no modo de *coleta*, os dados dbiométricos dos usuários são coletados, processados e cadastrados no sistema, junto com a identificação do usuário. No modo de *autenticação*, ele verifica se as características biométricas de um indivíduo (previamente identificado por algum outro método, como login/senha, cartão, etc.) correspondem às suas características biométricas previamente armazenadas. Desta forma, a biometria funciona como uma autenticação complementar. No modo de *identificação*, o sistema biométrico visa identificar o indivíduo a quem correspondem as características biométricas coletadas pelo sensor, dentre todos aqueles presentes no banco de dados. A Figura 28.4 mostra os principais elementos de um sistema biométrico típico.

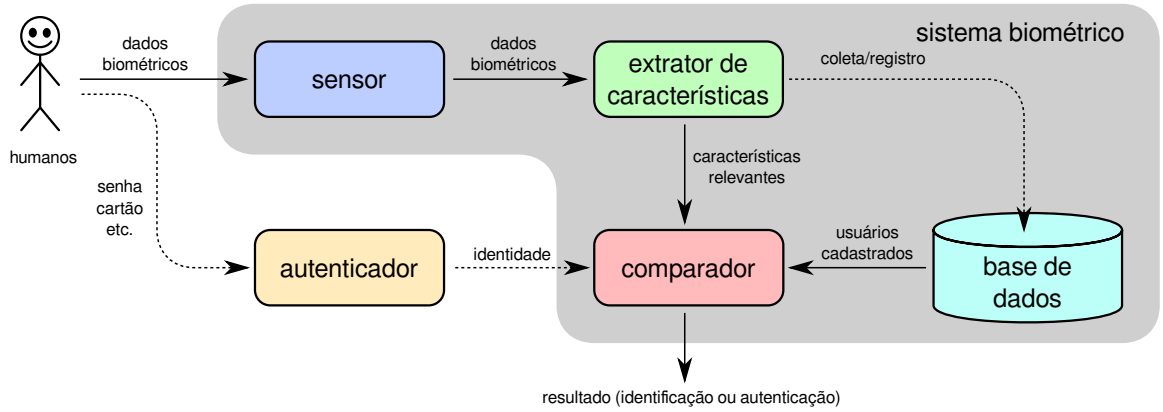


Figura 28.4: Um sistema biométrico típico.

28.7 Desafio/resposta

Em algumas situações o uso de senhas é indesejável, pois sua exposição indevida pode comprometer a segurança do sistema. Um exemplo disso são os serviços via rede: caso o tráfego de rede possa ser capturado por um intruso, este terá acesso às senhas transmitidas entre o cliente e o servidor. Uma técnica interessante para resolver esse problema são os protocolos de *desafio/resposta*.

A técnica de desafio/resposta se baseia sobre um segredo s previamente definido entre o cliente e o servidor (ou o usuário e o sistema), que pode ser uma senha ou uma chave criptográfica, e um algoritmo de cifragem ou resumo $hash(x)$, também previamente definido. No início da autenticação, o servidor escolhe um valor aleatório d e o envia ao cliente, como um *desafio*. O cliente recebe esse desafio, o concatena com seu segredo s, calcula o resumo da concatenação e a devolve ao servidor, como *resposta* ($r = hash(s \parallel d)$). O servidor executa a mesma operação de seu lado, usando o valor do segredo armazenado localmente (s') e compara o resultado obtido $r' = hash(s' \parallel d)$ com a resposta r fornecida pelo cliente. Se ambos os resultados forem iguais, os segredos são iguais ($r = r' \Rightarrow s = s'$) e o cliente é considerado autêntico. A Figura 28.5 apresenta os passos desse algoritmo.

A estratégia de desafio/resposta é robusta, porque o segredo s nunca é exposto fora do cliente nem do servidor; além disso, como o desafio d é aleatório e a resposta é cifrada, intrusos que eventualmente conseguirem capturar d ou r não poderão utilizá-los para se autenticar nem para descobrir s. Variantes dessa técnica são usadas em vários protocolos de rede, como o CHAP (em redes sem fio) e o SSH (para terminais remotos).

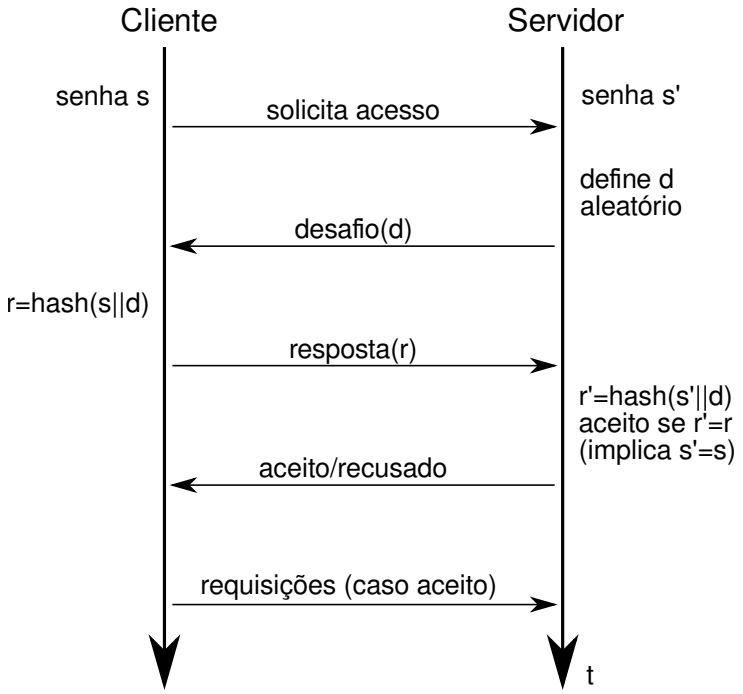


Figura 28.5: Autenticação por desafio/resposta.

28.8 Certificados de autenticação

Uma forma cada vez mais frequente de autenticação envolve o uso de *certificados digitais*. Conforme apresentado na Seção 27.10, um certificado digital é um documento assinado digitalmente, através de técnicas de criptografia assimétrica e resumo criptográfico. Os padrões de certificados PGP e X.509 definem certificados de autenticação (ou de identidade), cujo objetivo é identificar entidades através de suas chaves públicas. Um certificado de autenticação conforme o padrão X.509 contém as seguintes informações [Mollin, 2000]:

- Número de versão do padrão X.509 usado no certificado;
- Chave pública do proprietário do certificado e indicação do algoritmo de criptografia ao qual ela está associada e eventuais parâmetros;
- Número serial único, definido pelo emissor do certificado (quem o assinou);
- Identificação detalhada do proprietário do certificado, definida de acordo com normas do padrão X.509;
- Período de validade do certificado (datas de início e final de validade);
- Identificação da Autoridade Certificadora que emitiu/assinou o certificado;
- Assinatura digital do certificado e indicação do algoritmo usado na assinatura e eventuais parâmetros;

Os certificados digitais são o principal mecanismo usado para verificar a autenticidade de serviços acessíveis através da Internet, como bancos e comércio

eletrônico. Nesse caso, eles são usados para autenticar os sistemas para os usuários. No entanto, é cada vez mais frequente o uso de certificados para autenticar os próprios usuários. Nesse caso, um *smartcard* ou um dispositivo USB contendo o certificado é conectado ao sistema para permitir a autenticação do usuário.

28.9 Infraestruturas de autenticação

A autenticação é um procedimento necessário em vários serviços de um sistema computacional, que vão de simples sessões de terminal em modo texto a serviços de rede, como e-mail, bancos de dados e terminais gráficos remotos. Historicamente, cada forma de acesso ao sistema possuía seus próprios mecanismos de autenticação, com suas próprias regras e informações. Essa situação dificultava a criação de novos serviços, pois estes deveriam também definir seus próprios métodos de autenticação. Além disso, a existência de vários mecanismos de autenticação desconexos prejudicava a experiência do usuário e dificultava a gerência do sistema.

Para resolver esse problema, foram propostas infraestruturas de autenticação (*authentication frameworks*) que unificam as técnicas de autenticação, oferecem uma interface de programação homogênea e usam as mesmas informações (pares *login*/*senha*, dados biométricos, certificados, etc.). Assim, as informações de autenticação são coerentes entre os diversos serviços, novas técnicas de autenticação podem ser automaticamente usadas por todos os serviços e, sobretudo, a criação de novos serviços é simplificada.

A visão genérica de uma infraestrutura de autenticação local é apresentada na Figura 28.6. Nela, os vários mecanismos disponíveis de autenticação são oferecidos às aplicações através de uma interface de programação (API) padronizada. As principais infraestruturas de autenticação em uso nos sistemas operacionais atuais são:

PAM (*Pluggable Authentication Modules*): proposto inicialmente para o sistema Solaris, foi depois adotado em vários outros sistemas UNIX, como FreeBSD, NetBSD, MacOS X e Linux;

XSSO (*X/Open Single Sign-On*): é uma tentativa de extensão e padronização do sistema PAM, ainda pouco utilizada;

BSD Auth: usada no sistema operacional OpenBSD; cada método de autenticação é implementado como um processo separado, respeitando o princípio do privilégio mínimo (vide Seção 29.2);

NSS (*Name Services Switch*): infraestrutura usada em sistemas UNIX para definir as bases de dados a usar para vários serviços do sistema operacional, inclusive a autenticação;

GSSAPI (*Generic Security Services* API): padrão de API para acesso a serviços de segurança, como autenticação, confidencialidade e integridade de dados;

SSPI (*Security Support Provider Interface*): variante proprietária da GSSAPI, específica para plataformas Windows.

Além das infraestruturas de autenticação local, existem também padrões e protocolos para implementar ações de autenticação em redes de computadores e

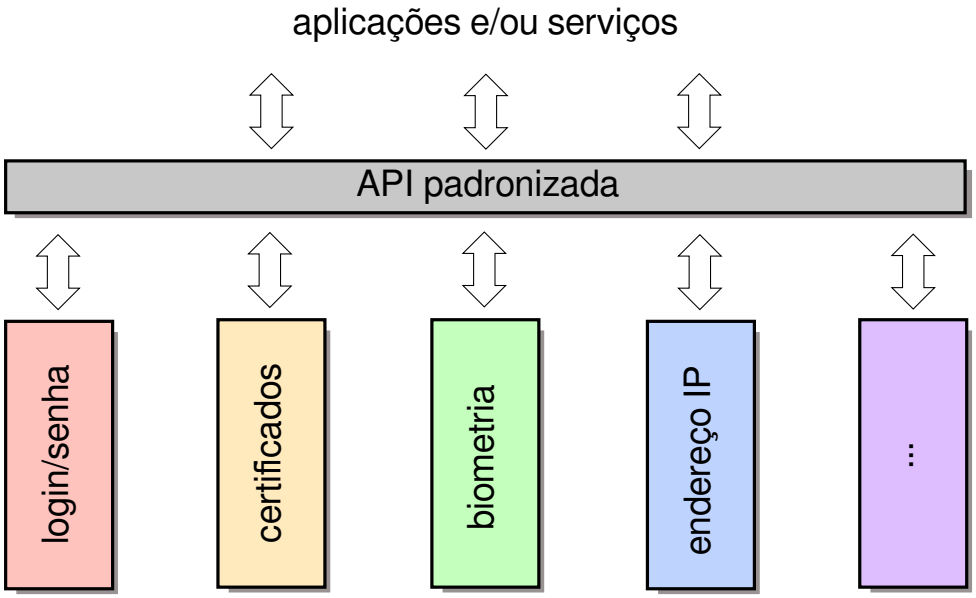


Figura 28.6: Estrutura genérica de uma infraestrutura de autenticação.

sistemas distribuídos, como a Internet. Procolos de autenticação em redes locais incluem o Kerberos (Seção 28.10), Windows NTML, CHAP, Radius/Diameter e LDAP, entre outros. Na Internet, os protocolos de autenticação OpenID e Shibboleth são muito utilizados.

28.10 Kerberos

O sistema de autenticação *Kerberos* foi proposto pelo MIT nos anos 80 [Neuman and Ts'o, 1994]. Hoje, esse sistema é utilizado para centralizar a autenticação de rede em vários sistemas operacionais, como Windows, Solaris, MacOS X e Linux. O sistema Kerberos se baseia na noção de *tickets*, que são obtidos pelos clientes junto a um serviço de autenticação e podem ser usados para acessar os demais serviços da rede. Os tickets são cifrados usando criptografia simétrica DES e têm validade limitada, para aumentar sua segurança.

Os principais componentes de um sistema Kerberos são o Serviço de Autenticação (AS - *Authentication Service*), o Serviço de Concessão de Tickets (TGS - *Ticket Granting Service*), a base de chaves, os clientes e os serviços de rede que os clientes podem acessar. Juntos, o AS e o TGS constituem o *Centro de Distribuição de Chaves* (KDC - *Key Distribution Center*). O funcionamento básico do sistema Kerberos, ilustrado na Figura 28.7, é relativamente simples: o cliente se autentica junto ao AS (passo 1) e obtém um ticket de acesso ao serviço de tickets TGS (passo 2). A seguir, solicita ao TGS um ticket de acesso ao servidor desejado (passos 3 e 4). Com esse novo ticket, ele pode se autenticar junto ao servidor desejado e solicitar serviços (passos 5 e 6).

No Kerberos, cada cliente c possui uma chave secreta k_c registrada no servidor de autenticação AS. Da mesma forma, cada servidor s também tem sua chave k_s registrada no AS. As chaves são simétricas, usando cifragem DES, e somente são conhecidas por seus respectivos proprietários e pelo AS. Os seguintes passos detalham o funcionamento do Kerberos versão 5 [Neuman and Ts'o, 1994]:

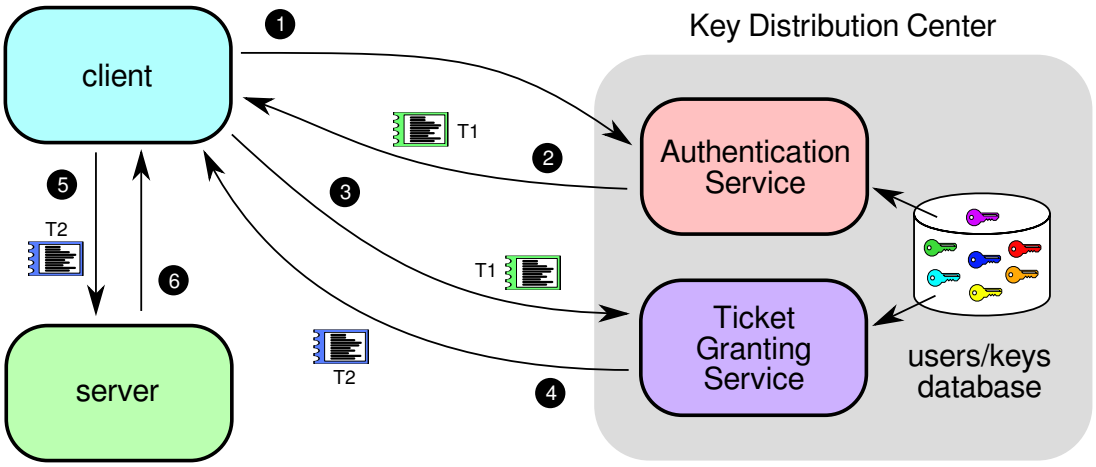


Figura 28.7: Visão geral do serviço Kerberos.

1. Uma máquina cliente c desejando acessar um determinado servidor s envia uma solicitação de autenticação ao serviço de autenticação (AS); essa mensagem m_1 contém sua identidade (c), a identidade do serviço desejado (tgs), um prazo de validade solicitado (ts) e um número aleatório (n_1) que será usado para verificar se a resposta do AS corresponde ao pedido efetuado:

$$m_1 = [c \;\; tgs \;\; ts \;\; n_1]$$

2. A resposta do AS (mensagem m_2) contém duas partes: a primeira parte contém a chave de sessão a ser usada na comunicação com o TGS (k_{c-tgs}) e o número aleatório n_1, ambos cifrados com a chave do cliente k_c registrada no AS; a segunda parte é um ticket cifrado com a chave do TGS (k_{tgs}), contendo a identidade do cliente (c), o prazo de validade do ticket concedido pelo AS (tv) e uma chave de sessão k_{c-tgs}, a ser usada na interação com o TGS:

$$m_2 = [\{k_{c-tgs} \;\; n_1\}_{k_c} \;\; T_{c-tgs}] \quad \text{onde } T_{c-tgs} = \{c \;\; tv \;\; k_{c-tgs}\}_{k_{tgs}}$$

   O ticket T_{c-tgs} fornecido pelo AS para permitir o acesso ao TGS é chamado TGT (*Ticket Granting Ticket*), e possui um prazo de validade limitado (geralmente de algumas horas). Ao receber m_2, o cliente tem acesso à chave de sessão k_{c-tgs} e ao ticket TGT. Todavia, esse ticket é cifrado com a chave k_{tgs} e portanto somente o TGS poderá abrí-lo.

3. A seguir, o cliente envia uma solicitação ao TGS (mensagem m_3) para obter um ticket de acesso ao servidor desejado s. Essa solicitação contém a identidade do cliente (c) e a data atual (t), ambos cifrados com a chave de sessão k_{c-tgs}, o ticket TGT recebido em m_2, a identidade do servidor s e um número aleatório n_2:

$$m_3 = [\{c \;\; t\}_{k_{c-tgs}} \;\; T_{c-tgs} \;\; s \;\; n_2]$$

4. Após verificar a validade do ticket TGT, o TGS devolve ao cliente uma mensagem m_4 contendo a chave de sessão k_{c-s} a ser usada no acesso ao servidor s e o número

aleatório n_2 informado em m_3, ambos cifrados com a chave de sessão k_{c-tgs}, e um ticket T_{c-s} cifrado, que deve ser apresentado ao servidor s:

$$m_4 = [\{k_{c-s} \;\; n\}_{k_{c-tgs}} \;\; T_{c-s}] \quad \text{onde } T_{c-s} = \{c \;\; tv \;\; k_{c-s}\}_{k_s}$$

5. O cliente usa a chave de sessão k_{c-s} e o ticket T_{c-s} para se autenticar junto ao servidor s através da mensagem m_5. Essa mensagem contém a identidade do cliente (c) e a data atual (t), ambos cifrados com a chave de sessão k_{c-s}, o ticket T_{c-s} recebido em m_4 e o pedido de serviço ao servidor (*request*), que é dependente da aplicação:

$$m_5 = [\{c \;\; t\}_{k_{c-s}} \;\; T_{c-s} \;\; request]$$

6. Ao receber m_5, o servidor s decifra o ticket T_{c-s} para obter a chave de sessão k_{c-s} e a usa para decifrar a primeira parte da mensagem e confirmar a identidade do cliente. Feito isso, o servidor pode atender a solicitação e responder ao cliente, cifrando sua resposta com a chave de sessão k_{c-s}:

$$m_6 = [\{reply\}_{k_{c-s}}]$$

Enquanto o ticket de serviço T_{c-s} for válido, o cliente pode enviar solicitações ao servidor sem a necessidade de se reautenticar. Da mesma forma, enquanto o ticket T_{c-tgs} for válido, o cliente pode solicitar tickets de acesso a outros servidores sem precisar se reautenticar. Pode-se observar que em nenhum momento as chaves de sessão k_{c-tgs} e k_{c-s} circularam em aberto através da rede. Além disso, a presença de prazos de validade para as chaves permite minimizar os riscos de uma eventual captura da chave. Informações mais detalhadas sobre o funcionamento do protocolo Kerberos 5 podem ser encontradas em [Neuman et al., 2005].

Referências

A. Jain, A. Ross, and S. Prabhakar. An Introduction to Biometric Recognition. *IEEE Transactions on Circuits and Systems for Video Technology*, 14(1), Apr. 2004.

A. Menezes, P. Van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

R. A. Mollin. *An Introduction to Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 2000. ISBN 1584881275.

B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, September 1994.

C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). RFC 4120 (Proposed Standard), July 2005. URL `http://www.ietf.org/rfc/rfc4120.txt`. Updated by RFCs 4537, 5021.

Capítulo 29

Controle de acesso

Em um sistema computacional, o controle de acesso consiste em mediar cada solicitação de acesso de um usuário autenticado a um recurso ou dado mantido pelo sistema, para determinar se aquela solicitação deve ser autorizada ou negada [Samarati and De Capitani di Vimercati, 2001]. Praticamente todos os recursos de um sistema operacional típico estão submetidos a um controle de acesso, como arquivos, áreas de memória, semáforos, portas de rede, dispositivos de entrada/saída, etc.

29.1 Terminologia

Esta seção define alguns termos usuais na área de controle de acesso:

Sujeito: são todas aquelas entidades que executam ações no sistema, como processos, *threads* ou transações. Normalmente um sujeito opera em nome de um usuário, que pode ser um ser humano ou outro sistema computacional externo.

Objeto: são as entidades passivas sujeitas às ações dos sujeitos, como arquivos, áreas de memória, registros em um banco de dados ou outros recursos. Em alguns casos, um sujeito pode ser visto como objeto por outro sujeito (por exemplo, quando um sujeito deve enviar uma mensagem a outro sujeito).

Acesso: ação realizada por um sujeito sobre um objeto. Por exemplo, um acesso de um processo a um arquivo, um envio de pacote de rede através de uma porta UDP, execução de um programa, etc.

Autorização: é a permissão para que um sujeito realize uma determinada ação sobre um objeto. Existem autorizações positivas (que permitem uma ação) e autorizações negativas (que negam uma ação).

Tanto sujeitos quanto objetos e autorizações podem ser organizados em grupos e hierarquias, para facilitar a gerência da segurança.

29.2 Políticas, modelos e mecanismos

Uma *política de controle de acesso* é uma visão abstrata das permissões de acesso a recursos (objetos) pelos usuários (sujeitos) de um sistema. Essa política consiste

basicamente de um conjunto de regras definindo os acessos possíveis aos recursos do sistema e eventuais condições necessárias para permitir cada acesso. Por exemplo, as regras a seguir poderiam constituir parte da política de segurança de um sistema de informações médicas:

1. Médicos podem consultar os prontuários de seus pacientes;
2. Médicos podem modificar os prontuários de seus pacientes enquanto estes estiverem internados;
3. O supervisor geral pode consultar os prontuários de todos os pacientes;
4. Enfermeiros podem consultar apenas os prontuários dos pacientes de sua seção e somente durante seu período de turno;
5. Assistentes não podem consultar prontuários;
6. Prontuários de pacientes de planos de saúde privados podem ser consultados pelo responsável pelo respectivo plano de saúde no hospital;
7. Pacientes podem consultar seus próprios prontuários (aceitar no máximo 30 pacientes simultâneos).

As regras ou definições individuais de uma política são denominadas *autorizações*. Uma política de controle de acesso pode ter autorizações baseadas em *identidades* (como sujeitos e objetos) ou em outros *atributos* (como idade, sexo, tipo, preço, etc.); as autorizações podem ser *individuais* (a sujeitos) ou *coletivas* (a grupos); também podem existir autorizações *positivas* (permitindo o acesso) ou *negativas* (negando o acesso); por fim, uma política pode ter autorizações dependentes de *condições externas* (como o horário ou a carga do sistema). Além da política de acesso aos objetos, também deve ser definida uma *política administrativa*, que define quem pode modificar/gerenciar as políticas vigentes no sistema [Samarati and De Capitani di Vimercati, 2001].

O conjunto de autorizações de uma política deve ser ao mesmo tempo *completo*, cobrindo todas as possibilidades de acesso que vierem a ocorrer no sistema, e *consistente*, sem regras conflitantes entre si (por exemplo, uma regra que permita um acesso e outra que negue esse mesmo acesso). Além disso, toda política deve buscar respeitar o *princípio do privilégio mínimo* [Saltzer and Schroeder, 1975], segundo o qual um usuário nunca deve receber mais autorizações que aquelas que necessita para cumprir sua tarefa. A construção e validação de políticas de controle de acesso é um tema complexo, que está fora do escopo deste texto, sendo melhor descrito em [di Vimercati et al., 2005, 2007].

As políticas de controle de acesso definem de forma abstrata como os sujeitos podem acessar os objetos do sistema. Existem muitas formas de se definir uma política, que podem ser classificadas em quatro grandes classes: políticas *discricionárias*, políticas *obrigatórias*, políticas *baseadas em domínios* e políticas *baseadas em papéis* [Samarati and De Capitani di Vimercati, 2001]. As próximas seções apresentam com mais detalhe cada uma dessas classes de políticas.

Geralmente a descrição de uma política de controle de acesso é muito abstrata e informal. Para sua implementação em um sistema real, ela precisa ser descrita de uma forma precisa, através de um *modelo de controle de acesso*. Um modelo de controle de acesso é uma representação lógica ou matemática da política, de forma a facilitar

sua implementação e permitir a análise de eventuais erros. Em um modelo de controle de acesso, as autorizações de uma política são definidas como relações lógicas entre *atributos do sujeito* (como seus identificadores de usuário e grupo) *atributos do objeto* (como seu caminho de acesso ou seu proprietário) e eventuais condições externas (como o horário ou a carga do sistema). Nas próximas seções, para cada classe de políticas de controle de acesso apresentada serão discutidos alguns modelos aplicáveis à mesma.

Por fim, os *mecanismos de controle de acesso* são as estruturas necessárias à implementação de um determinado modelo em um sistema real. Como é bem sabido, é de fundamental importância a separação entre políticas e mecanismos, para permitir a substituição ou modificação de políticas de controle de acesso de um sistema sem incorrer em custos de modificação de sua implementação. Assim, um mecanismo de controle de acesso ideal deveria ser capaz de suportar qualquer política de controle de acesso.

29.3 Políticas discricionárias

As políticas discricionárias (DAC - *Discretionary Access Control*) se baseiam na atribuição de permissões de forma individualizada, ou seja, pode-se claramente conceder (ou negar) a um sujeito específico s a permissão de executar a ação a sobre um objeto específico o. Em sua forma mais simples, as regras de uma política discricionária têm a forma $\langle s, o, +a \rangle$ ou $\langle s, o, -a \rangle$, para respectivamente autorizar ou negar a ação a do sujeito s sobre o objeto o (também podem ser definidas regras para grupos de usuários e/ou de objetos devidamente identificados). Por exemplo:

- O usuário `Beto` pode ler e escrever arquivos em `/home/beto`
- Usuários do grupo `admin` podem ler os arquivos em `/suporte`

O responsável pela administração das permissões de acesso a um objeto pode ser o seu proprietário ou um administrador central. A definição de quem estabelece as regras da política de controle de acesso é inerente a uma política administrativa, independente da política de controle de acesso em si[1].

29.3.1 Matriz de controle de acesso

O modelo matemático mais simples e conveniente para representar políticas discricionárias é a *Matriz de Controle de Acesso*, proposta em [Lampson, 1971]. Nesse modelo, as autorizações são dispostas em uma matriz, cujas linhas correspondem aos sujeitos do sistema e cujas colunas correspondem aos objetos. Em termos formais, considerando um conjunto de sujeitos $\mathbb{S} = \{s_1, s_2, \ldots, s_m\}$, um conjunto de objetos $\mathbb{O} = \{o_1, o_2, \ldots, o_n\}$ e um conjunto de ações possíveis sobre os objetos $\mathbb{A} = \{a_1, a_2, \ldots, a_p\}$, cada elemento M_{ij} da matriz de controle de acesso é um subconjunto (que pode ser vazio) do conjunto de ações possíveis, que define as ações que $s_i \in \mathbb{S}$ pode efetuar sobre $o_j \in \mathbb{O}$:

[1] Muitas políticas de controle de acesso discricionárias são baseadas na noção de que cada recurso do sistema possui um proprietário, que decide quem pode acessar o recurso. Isso ocorre por exemplo nos sistemas de arquivos, onde as permissões de acesso a cada arquivo ou diretório são definidas pelo respectivo proprietário. Contudo, a noção de "proprietário" de um recurso não é essencial para a construção de políticas discricionárias [Shirey, 2000].

$$\forall s_i \in \mathbb{S}, \forall o_j \in \mathbb{O}, M_{ij} \subseteq \mathbb{A}$$

Por exemplo, considerando um conjunto de sujeitos $\mathbb{S} = \{Alice, Beto, Carol, Davi\}$, um conjunto de objetos $\mathbb{O} = \{file_1, file_2, program_1, socket_1\}$ e um conjunto de ações $\mathbb{A} = \{read, write, execute, remove\}$, podemos ter uma matriz de controle de acesso como a apresentada na Tabela 29.1.

	$file_1$	$file_2$	$program_1$	$socket_1$
Alice	*read* *write* *remove*	*read* *write*	*execute*	*write*
Beto	*read* *write*	*read* *write* *remove*	*read*	
Carol		*read*	*execute*	*read* *write*
Davi	*read*	*append*	*read*	*read* *append*

Tabela 29.1: Uma matriz de controle de acesso

Apesar de simples, o modelo de matriz de controle de acesso é suficientemente flexível para suportar políticas administrativas. Por exemplo, considerando uma política administrativa baseada na noção de proprietário do recurso, poder-se-ia considerar que cada objeto possui um ou mais proprietários (*owner*), e que os sujeitos podem modificar as entradas da matriz de acesso relativas aos objetos que possuem. Uma matriz de controle de acesso com essa política administrativa é apresentada na Tabela 29.2.

Embora seja um bom modelo conceitual, a matriz de acesso é inadequada para implementação. Em um sistema real, com milhares de sujeitos e milhões de objetos, essa matriz pode se tornar gigantesca e consumir muito espaço. Como em um sistema real cada sujeito tem seu acesso limitado a um pequeno grupo de objetos (e vice-versa), a matriz de acesso geralmente é esparsa, ou seja, contém muitas células vazias. Assim, algumas técnicas simples podem ser usadas para implementar esse modelo, como as tabelas de autorizações, as listas de controle de acesso e as listas de capacidades [Samarati and De Capitani di Vimercati, 2001], explicadas a seguir.

29.3.2 Tabela de autorizações

Na abordagem conhecida como **Tabela de Autorizações**, as entradas não vazias da matriz são relacionadas em uma tabela com três colunas: *sujeitos*, *objetos* e *ações*, onde cada tupla da tabela corresponde a uma autorização. Esta abordagem é muito utilizada em sistemas gerenciadores de bancos de dados (DBMS - *Database Management Systems*), devido à sua facilidade de implementação e consulta nesse tipo de ambiente. A Tabela

	$file_1$	$file_2$	$program_1$	$socket_1$
Alice	*read* *write* *remove* ***owner***	*read* *write*	*execute*	*write*
Beto	*read* *write*	*read* *write* *remove* ***owner***	*read* ***owner***	
Carol		*read*	*execute*	*read* *write*
Davi	*read*	*write*	*read*	*read* *write* ***owner***

Tabela 29.2: Uma matriz de controle de acesso com política administrativa

29.3 mostra como ficaria a matriz de controle de acesso da Tabela 29.2 sob a forma de uma tabela de autorizações.

29.3.3 Listas de controle de acesso

Outra abordagem usual é a **Lista de Controle de Acesso**. Nesta abordagem, para cada objeto é definida uma lista de controle de acesso (ACL - *Access Control List*), que contém a relação de sujeitos que podem acessá-lo, com suas respectivas permissões. Cada lista de controle de acesso corresponde a uma coluna da matriz de controle de acesso. Como exemplo, as listas de controle de acesso relativas à matriz de controle de acesso da Tabela 29.2 seriam:

$$
\begin{aligned}
ACL(file_1) &= \{ \; Alice : (read, write, remove, owner), \\
&\quad\; Beto : (read, write), \\
&\quad\; Davi : (read) \; \} \\
ACL(file_2) &= \{ \; Alice : (read, write), \\
&\quad\; Beto : (read, write, remove, owner), \\
&\quad\; Carol : (read), \\
&\quad\; Davi : (write) \; \} \\
ACL(program_1) &= \{ \; Alice : (execute), \\
&\quad\; Beto : (read, owner), \\
&\quad\; Carol : (execute), \\
&\quad\; Davi : (read) \; \} \\
ACL(socket_1) &= \{ \; Alice : (write),
\end{aligned}
$$

Sujeito	Objeto	Ação	Sujeito	Objeto	Ação
Alice	$file_1$	*read*	Beto	$file_2$	*owner*
Alice	$file_1$	*write*	Beto	$program_1$	*read*
Alice	$file_1$	*remove*	Beto	$socket_1$	*owner*
Alice	$file_1$	*owner*	Carol	$file_2$	*read*
Alice	$file_2$	*read*	Carol	$program_1$	*execute*
Alice	$file_2$	*write*	Carol	$socket_1$	*read*
Alice	$program_1$	*execute*	Carol	$socket_1$	*write*
Alice	$socket_1$	*write*	Davi	$file_1$	*read*
Beto	$file_1$	*read*	Davi	$file_2$	*write*
Beto	$file_1$	*write*	Davi	$program_1$	*read*
Beto	$file_2$	*read*	Davi	$socket_1$	*read*
Beto	$file_2$	*write*	Davi	$socket_1$	*write*
Beto	$file_2$	*remove*	Davi	$socket_1$	*owner*

Tabela 29.3: Tabela de autorizações

$$\begin{aligned} &Carol : (read, write), \\ &Davi : (read, write, owner) \} \end{aligned}$$

Esta forma de implementação é a mais frequentemente usada em sistemas operacionais, por ser simples de implementar e bastante robusta. Por exemplo, o sistema de arquivos associa uma ACL a cada arquivo ou diretório, para indicar quem são os sujeitos autorizados a acessá-lo. Em geral, somente o proprietário do arquivo pode modificar sua ACL, para incluir ou remover permissões de acesso.

29.3.4 Listas de capacidades

Uma terceira abordagem possível para a implementação da matriz de controle de acesso é a **Lista de Capacidades** (CL - *Capability List*), ou seja, uma lista de objetos que um dado sujeito pode acessar e suas respectivas permissões sobre os mesmos. Cada lista de capacidades corresponde a uma linha da matriz de acesso. Como exemplo, as listas de capacidades correspondentes à matriz de controle de acesso da Tabela 29.2 seriam:

$$\begin{aligned} CL(Alice) = \{ \quad & file_1 : (read, write, remove, owner), \\ & file_2 : (read, write), \\ & program_1 : (execute), \\ & socket_1 : (write) \} \\ CL(Beto) = \{ \quad & file_1 : (read, write), \\ & file_2 : (read, write, remove, owner), \end{aligned}$$

$$
\begin{aligned}
& & & program_1 : (read, owner) \} \\
& CL(Carol) & = \{ \; & file_2 : (read), \\
& & & program_1 : (execute), \\
& & & socket_1 : (read, write) \} \\
& CL(Davi) & = \{ \; & file_1 : (read), \\
& & & file_2 : (write), \\
& & & program_1 : (read), \\
& & & socket_1 : (read, write, owner) \}
\end{aligned}
$$

Uma capacidade pode ser vista como uma ficha ou *token*: sua posse dá ao proprietário o direito de acesso ao objeto em questão. Capacidades são pouco usadas em sistemas operacionais, devido à sua dificuldade de implementação e possibilidade de fraude, pois uma capacidade mal implementada pode ser transferida deliberadamente a outros sujeitos, ou modificada pelo próprio proprietário para adicionar mais permissões a ela. Outra dificuldade inerente às listas de capacidades é a administração das autorizações: por exemplo, quem deve ter permissão para modificar uma lista de capacidades, e como retirar uma permissão concedida anteriormente a um sujeito? Alguns sistemas operacionais que implementam o modelo de capacidades são discutidos na Seção 29.7.4.

29.4 Políticas obrigatórias

Nas *políticas obrigatórias* (MAC - *Mandatory Access Control*) o controle de acesso é definido por regras globais incontornáveis, que não dependem das identidades dos sujeitos e objetos nem da vontade de seus proprietários ou mesmo do administrador do sistema [Samarati and De Capitani di Vimercati, 2001]. Essas regras são normalmente baseadas em atributos dos sujeitos e/ou dos objetos, como mostram estes exemplos bancários (fictícios):

- Cheques com valor acima de R$ 5.000,00 devem ser obrigatoriamente depositados e não podem ser descontados;
- Clientes com renda mensal acima de R$3.000,00 não têm acesso ao crédito consignado.

Uma das formas mais usuais de política obrigatória são as *políticas multinível* (MLS - *Multi-Level Security*), que se baseiam na classificação de sujeitos e objetos do sistema em *níveis de segurança* (*clearance levels*, $\mathbb{S}$) e na definição de regras usando esses níveis. Um exemplo bem conhecido de escala de níveis de segurança é aquela usada pelo governo britânico para definir a confidencialidade de um documento:

- *TS*: *Top Secret* (*Ultrassecreto*)
- *S*: *Secret* (*Secreto*)
- *C*: *Confidential* (*Confidencial*)

- *R*: *Restrict* (*Reservado*)
- *U*: *Unclassified* (*Público*)

Em uma política MLS, considera-se que os níveis de segurança estão ordenados entre si (por exemplo, $U < R < C < S < TS$) e são associados a todos os sujeitos e objetos do sistema, sob a forma de *habilitação* dos sujeitos ($h(s_i) \in \mathbb{S}$) e *classificação* dos objetos ($c(o_j) \in \mathbb{S}$). As regras da política são então estabelecidas usando essas habilitações e classificações, como mostram os modelos de Bell-LaPadula e de Biba, descritos a seguir.

Além das políticas multinível, existem também políticas denominadas *multilaterais*, nas quais o objetivo é evitar fluxos de informação indevidos entre departamentos ou áreas distintas em uma organização. *Chinese Wall* e *Clark-Wilson* são exemplos dessa família de políticas [Anderson, 2008].

29.4.1 Modelo de Bell-LaPadula

Um modelo de controle de acesso que permite formalizar políticas multinível é o de *Bell-LaPadula* [Bell and LaPadula, 1974], usado para garantir a confidencialidade das informações. Esse modelo consiste basicamente de duas regras:

No-Read-Up ("não ler acima", ou "propriedade simples"): impede que um sujeito leia objetos que se encontrem em níveis de segurança acima do seu. Por exemplo, um sujeito habilitado como confidencial (*C*) somente pode ler objetos cuja classificação seja confidencial (*C*), reservada (*R*) ou pública (*U*). Considerando um sujeito s e um objeto o, formalmente temos:

$$request(s, o, read) \iff h(s) \geq c(o)$$

No-Write-Down ("não escrever abaixo", ou "propriedade $\star$"): impede que um sujeito escreva em objetos abaixo de seu nível de segurança, para evitar o "vazamento" de informações dos níveis superiores para os inferiores. Por exemplo, um sujeito habilitado como confidencial somente pode escrever em objetos cuja classificação seja confidencial, secreta ou ultrassecreta. Formalmente, temos:

$$request(s, o, write) \iff h(s) \leq c(o)$$

As regras da política de Bell-LaPadula estão ilustradas na Figura 29.1. Pode-se perceber que a política obrigatória representada pelo modelo de Bell-LaPadula visa proteger a *confidencialidade* das informações do sistema, evitando que estas fluam dos níveis superiores para os inferiores. Todavia, nada impede um sujeito com baixa habilitação escrever sobre um objeto de alta classificação, destruindo seu conteúdo.

29.4.2 Modelo de Biba

Para garantir a *integridade* das informações, um modelo dual ao de Bell-LaPadula foi proposto por Kenneth Biba [Biba, 1977]. Esse modelo define níveis de integridade $i(x) \in \mathbb{I}$ para sujeitos e objetos (como *Baixa*, *Média*, *Alta* e *Sistema*, com $B < M < A < S$), e também possui duas regras básicas:

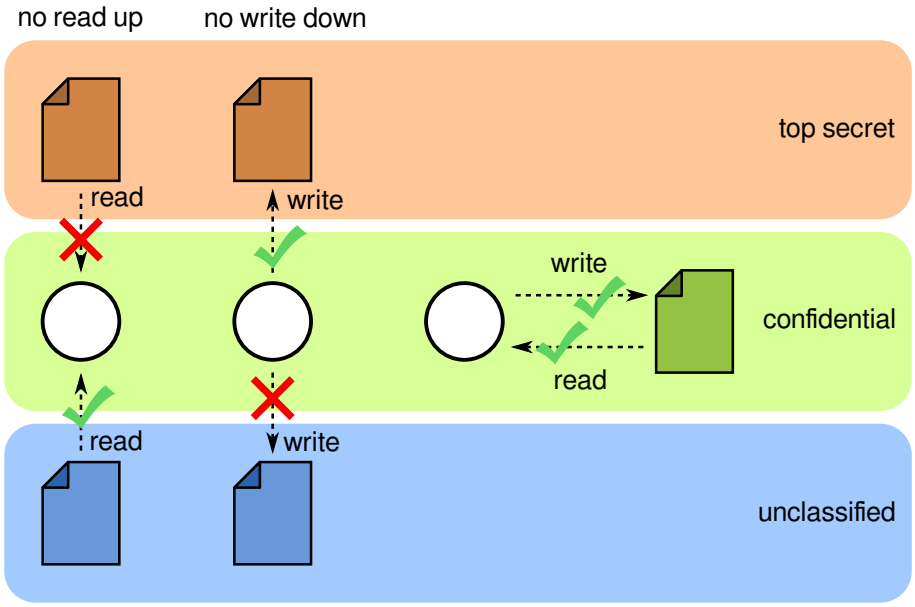


Figura 29.1: Política de Bell-LaPadula.

No-Write-Up ("não escrever acima", ou "propriedade simples de integridade"): impede que um sujeito escreva em objetos acima de seu nível de integridade, preservando-os íntegros. Por exemplo, um sujeito de integridade média (M) somente pode escrever em objetos de integridade baixa (B) ou média (M). Formalmente, temos:

$$request(s, o, write) \iff i(s) \geq i(o)$$

No-Read-Down ("não ler abaixo", ou "propriedade $\star$ de integridade"): impede que um sujeito leia objetos em níveis de integridade abaixo do seu, para não correr o risco de ler informação duvidosa. Por exemplo, um sujeito com integridade alta (A) somente pode ler objetos com integridade alta (A) ou de sistema (S). Formalmente, temos:

$$request(s, o, read) \iff i(s) \leq i(o)$$

As regras da política de Biba estão ilustradas na Figura 29.2. Essa política obrigatória evita violações de integridade, mas não garante a confidencialidade das informações. Para que as duas políticas (confidencialidade e integridade) possam funcionar em conjunto, é necessário portanto associar a cada sujeito e objeto do sistema um nível de confidencialidade e um nível de integridade, possivelmente distintos, ou seja, combinar as políticas de Bell-LaPadula e Biba.

É importante observar que, na maioria dos sistemas reais, **as políticas obrigatórias não substituem as políticas discricionárias**, mas as complementam. Em um sistema que usa políticas obrigatórias, cada acesso a recurso é verificado usando a política obrigatória e também uma política discricionária; o acesso é permitido somente se ambas as políticas o autorizarem. A ordem de avaliação das políticas MAC e DAC obviamente não afeta o resultado final, mas pode ter impacto sobre o desempenho do sistema. Por isso, deve-se primeiro avaliar a política mais restritiva, ou seja, aquela que tem mais probabilidades de negar o acesso.

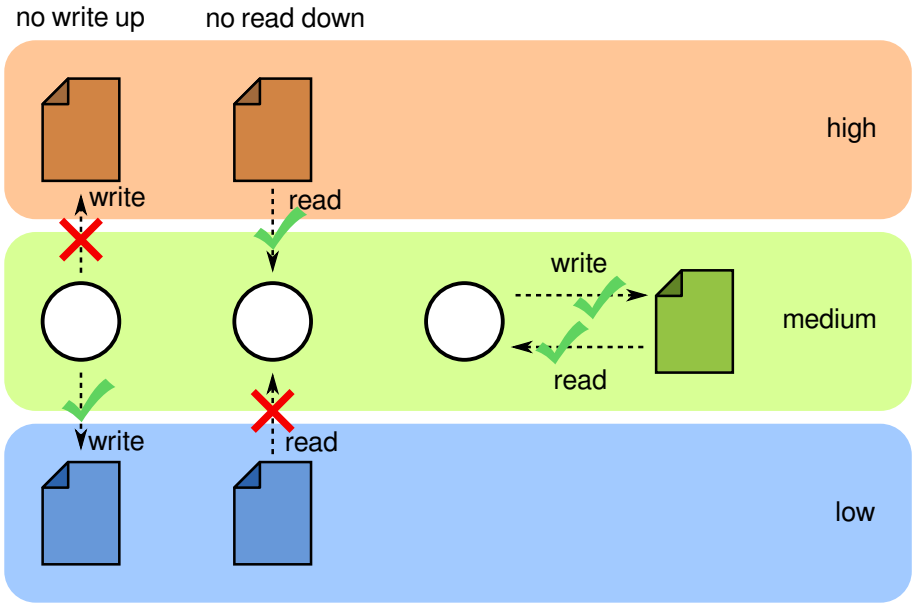


Figura 29.2: Política de Biba.

29.4.3 Categorias

Uma extensão frequente às políticas multinível é a noção de *categorias* ou *compartimentos*. Uma categoria define uma área funcional dentro do sistema computacional, como "pessoal", "projetos", "financeiro", "suporte", etc. Normalmente o conjunto de categorias é estático não há uma ordem hierárquica entre elas. Cada sujeito e cada objeto do sistema são "rotulados" com uma ou mais categorias; a política então consiste em restringir o acesso de um sujeito somente aos objetos pertencentes às mesmas categorias dele, ou a um subconjunto destas. Dessa forma, um sujeito com as categorias $\{suporte, financeiro\}$ só pode acessar objetos rotulados como $\{suporte, financeiro\}$, $\{suporte\}$, $\{financeiro\}$ ou $\{\phi\}$. Formalmente: sendo $\mathbb{C}(s)$ o conjunto de categorias associadas a um sujeito s e $\mathbb{C}(o)$ o conjunto de categorias associadas a um objeto o, s só pode acessar o se $\mathbb{C}(s) \supseteq \mathbb{C}(o)$ [Samarati and De Capitani di Vimercati, 2001].

29.5 Políticas baseadas em domínios e tipos

O *domínio de segurança* de um sujeito define o conjunto de objetos que ele pode acessar e como pode acessá-los. Muitas vezes esse domínio está definido implicitamente nas regras das políticas obrigatórias ou na matriz de controle de acesso de uma política discricionária. As *políticas baseadas em domínios e tipos* (DTE - *Domain/Type Enforcement policies*) [Boebert and Kain, 1985] tornam explícito esse conceito: cada sujeito s do sistema é rotulado com um atributo constante definindo seu domínio $domain(s)$ e cada objeto o é associado a um tipo $type(o)$, também constante.

No modelo de implementação de uma política DTE definido em [Badger et al., 1995], as permissões de acesso de sujeitos a objetos são definidas em uma tabela global chamada *Tabela de Definição de Domínios* (DDT - *Domain Definition Table*), na qual cada linha é associada a um domínio e cada coluna a um tipo; cada célula $DDT[x, y]$ contém as permissões de sujeitos do domínio x a objetos do tipo y:

$$request(s, o, action) \iff action \in DDT[domain(s), type(o)]$$

Por sua vez, as interações entre sujeitos (trocas de mensagens, sinais, etc.) são reguladas por uma *Tabela de Interação entre Domínios* (DIT - *Domain Interaction Table*). Nessa tabela, linhas e colunas correspondem a domínios e cada célula $DIT[x, y]$ contém as interações possíveis de um sujeito no domínio x sobre um sujeito no domínio y:

$$request(s_i, s_j, interaction) \iff interaction \in DIT[domain(s_i), domain(s_j)]$$

Eventuais mudanças de domínio podem ser associadas a programas executáveis rotulados como *pontos de entrada* (*entry points*). Quando um processo precisa mudar de domínio, ele executa o ponto de entrada correspondente ao domínio de destino, se tiver permissão para tal.

O código a seguir define uma política de controle de acesso DTE, usada como exemplo em [Badger et al., 1995]. Essa política está representada graficamente (de forma simplificada) na Figura 29.3.

```
1  /* type definitions */
2  type unix_t,     /* normal UNIX files, programs, etc. */
3      specs_t,     /* engineering specifications */
4      budget_t,    /* budget projections */
5      rates_t;     /* labor rates */
6
7  #define DEFAULT (/bin/sh), (/bin/csh), (rxd->unix_t) /* macro */
8
9  /* domain definitions */
10 domain engineer_d = DEFAULT, (rwd->specs_t);
11 domain project_d = DEFAULT, (rwd->budget_t), (rd->rates_t);
12 domain accounting_d = DEFAULT, (rd->budget_t), (rwd->rates_t);
13 domain system_d  = (/etc/init), (rwxd->unix_t), (auto->login_d);
14 domain login_d   = (/bin/login), (rwxd->unix_t),
15                    (exec-> engineer_d, project_d, accounting_d);
16
17 initial_domain system_d; /* system starts in this domain */
18
19 /* assign resources (files and directories) to types */
20 assign -r unix_t   /;       /* default for all files */
21 assign -r specs_t  /projects/specs;
22 assign -r budget_t /projects/budget;
23 assign -r rates_t  /projects/rates;
```

A implementação direta desse modelo sobre um sistema real pode ser inviável, pois exige a classificação de todos os sujeitos e objetos do mesmo em domínios e tipos. Para atenuar esse problema, [Badger et al., 1995; Cowan et al., 2000] propõem o uso de *tipagem implícita*: todos os objetos que satisfazem um certo critério (como por exemplo ter como caminho `/usr/local/*`) são automaticamente classificados em um dado tipo. Da mesma forma, os domínios podem ser definidos pelos nomes dos programas executáveis que os sujeitos executam (como `/usr/bin/httpd` e `/usr/lib/httpd/plugin/*` para o domínio do servidor Web). Além disso, ambos os autores propõem linguagens para a definição dos domínios e tipos e para a descrição das políticas de controle de acesso.

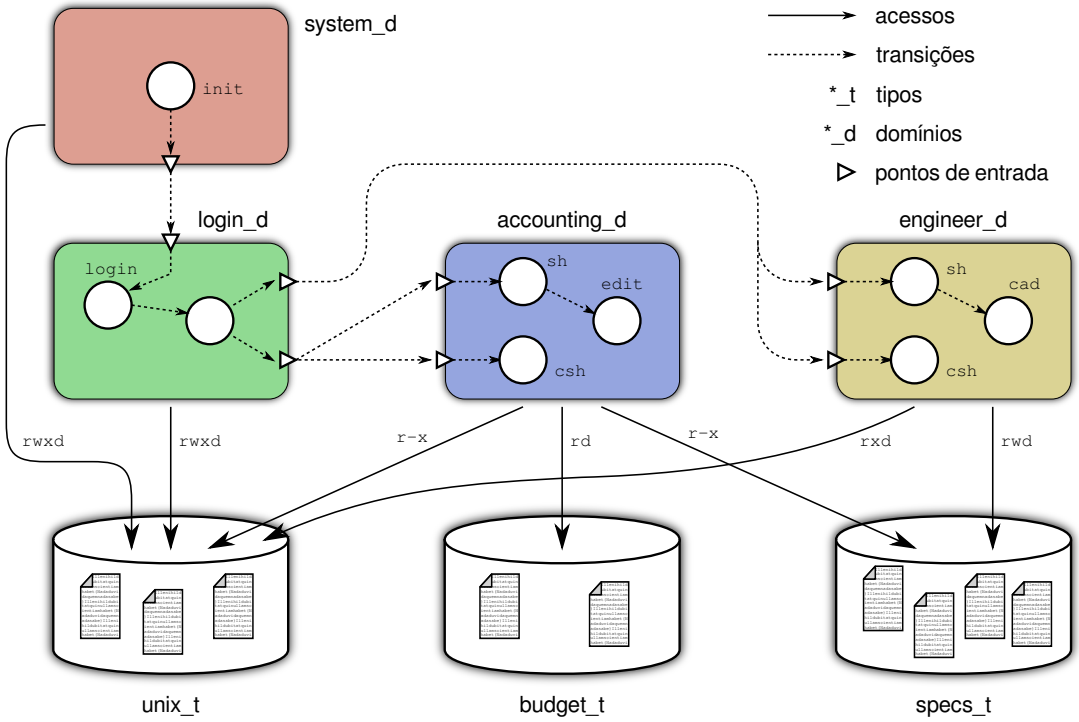


Figura 29.3: Exemplo de política baseada em domínios e tipos.

29.6 Políticas baseadas em papéis

Um dos principais problemas de segurança em um sistema computacional é a administração correta das políticas de controle de acesso. As políticas MAC são geralmente consideradas pouco flexíveis e por isso as políticas DAC acabam sendo muito mais usadas. Todavia, gerenciar as autorizações à medida em que usuários mudam de cargo e assumem novas responsabilidades, novos usuários entram na empresa e outros saem pode ser uma tarefa muito complexa e sujeita a erros.

Esse problema pode ser reduzido através do *controle de acesso baseado em papéis* (RBAC - *Role-Based Access Control*) [Sandhu et al., 1996]. Uma política RBAC define um conjunto de *papéis* no sistema, como "diretor", "gerente", "suporte", "programador", etc. e atribui a cada papel um conjunto de autorizações. Essas autorizações podem ser atribuídas aos papéis de forma discricionária ou obrigatória.

Para cada usuário do sistema é definido um conjunto de papéis que este pode assumir. Durante sua sessão no sistema (geralmente no início), o usuário escolhe os papéis que deseja ativar e recebe as autorizações correspondentes, válidas até este desativar os papéis correspondentes ou encerrar sua sessão. Assim, um usuário autorizado pode ativar os papéis de "professor" ou de "aluno" dependendo do que deseja fazer no sistema.

Os papéis permitem desacoplar os usuários das permissões. Por isso, um conjunto de papéis definido adequadamente é bastante estável, restando à gerência apenas atribuir a cada usuário os papéis a que este tem direito. A Figura 29.4 apresenta os principais componentes de uma política RBAC.

Existem vários modelos para a implementação de políticas baseadas em papéis, como os apresentados em [Sandhu et al., 1996]. Por exemplo, no modelo *RBAC hierárquico* os papéis são classificados em uma hierarquia, na qual os papéis superiores herdam

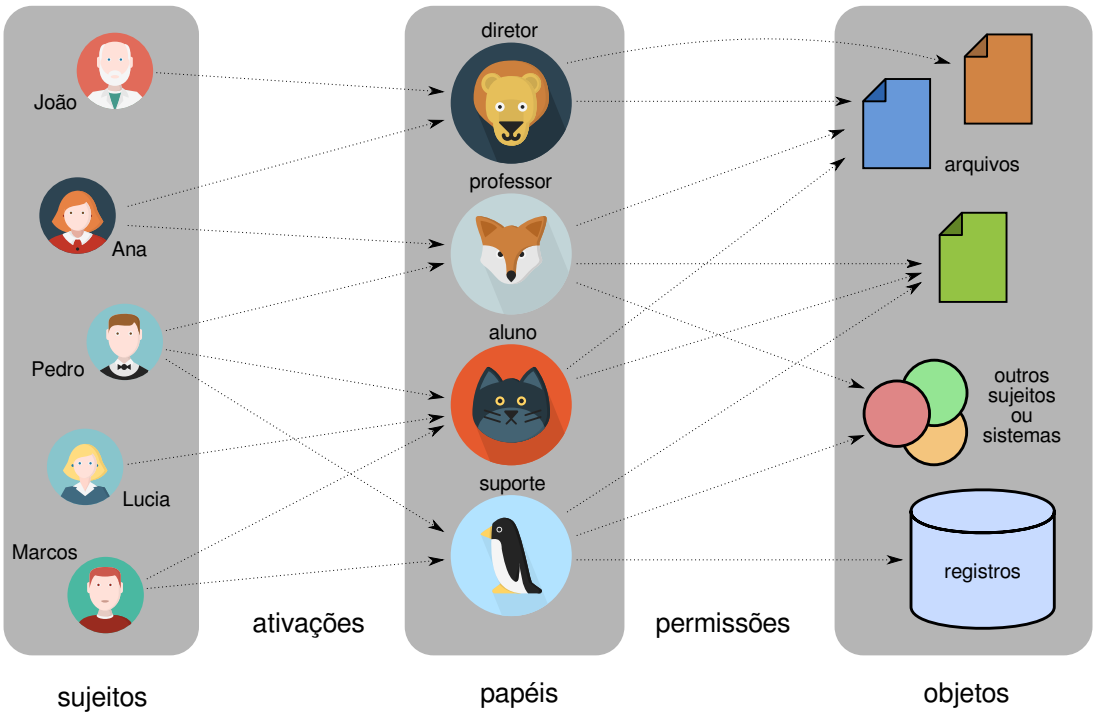


Figura 29.4: Políticas baseadas em papéis.

as permissões dos papéis inferiores. No modelo *RBAC com restrições* é possível definir restrições à ativação de papéis, como o número máximo de usuários que podem ativar um determinado papel simultaneamente ou especificar que dois papéis são conflitantes e não podem ser ativados pelo mesmo usuário simultaneamente.

Existem muitas outras políticas de controle de acesso além das apresentadas neste texto. Uma política que está ganhando popularidade é a ABAC – *Attribute-Based Access Control*, na qual o controle de acesso é feito usando regras baseadas em atributos dos sujeitos e objetos, não necessariamente suas identidades. Essas regras também podem levar em conta informações externas aos sujeitos e objetos, como horário, carga computacional do servidor, etc. Políticas baseadas em atributos são úteis em sistemas dinâmicos e de larga escala, como a Internet, onde a identidade de cada usuário específico é menos relevante que sua região geográfica, seu tipo de subscrição ao serviço desejado, ou outros atributos. O padrão ABAC definido pelo NIST [Hu et al., 2014] pode ser visto como uma estrutura formal genérica que permite construir políticas baseadas em atributos, além de permitir modelar políticas clássicas (discricionárias, obrigatórias), baseadas em papéis ou em domínios e tipos.

29.7 Mecanismos de controle de acesso

A implementação do controle de acesso em um sistema computacional deve ser independente das políticas de controle de acesso adotadas. Como nas demais áreas de um sistema operacional, a separação entre mecanismo e política é importante, por possibilitar trocar a política de controle de acesso sem ter de modificar a implementação do sistema. A infraestrutura de controle de acesso deve ser ao mesmo tempo *inviolável* (impossível de adulterar ou enganar) e *incontornável* (todos os acessos aos recursos do sistema devem passar por ela).

29.7.1 Infraestrutura básica

A arquitetura básica de uma infraestrutura de controle de acesso típica é composta pelos seguintes elementos (Figura 29.5):

Bases de sujeitos e objetos (*User/Object Bases*): relação dos sujeitos e objetos que compõem o sistema, com seus respectivos atributos;

Base de políticas (*Policy Base*): base de dados contendo as regras que definem como e quando os objetos podem ser acessados pelos sujeitos, ou como/quando os sujeitos podem interagir entre si;

Monitor de referências (*Reference monitor*): elemento que julga a pertinência de cada pedido de acesso. Com base em atributos do sujeito e do objeto (como suas respectivas identidades), nas regras da base de políticas e possivelmente em informações externas (como horário, carga do sistema, etc.), o monitor decide se um acesso deve ser permitido ou negado;

Mediador (impositor ou *Enforcer*): elemento que medeia a interação entre sujeitos e objetos; a cada pedido de acesso a um objeto, o mediador consulta o monitor de referências e permite/nega o acesso, conforme a decisão deste último.

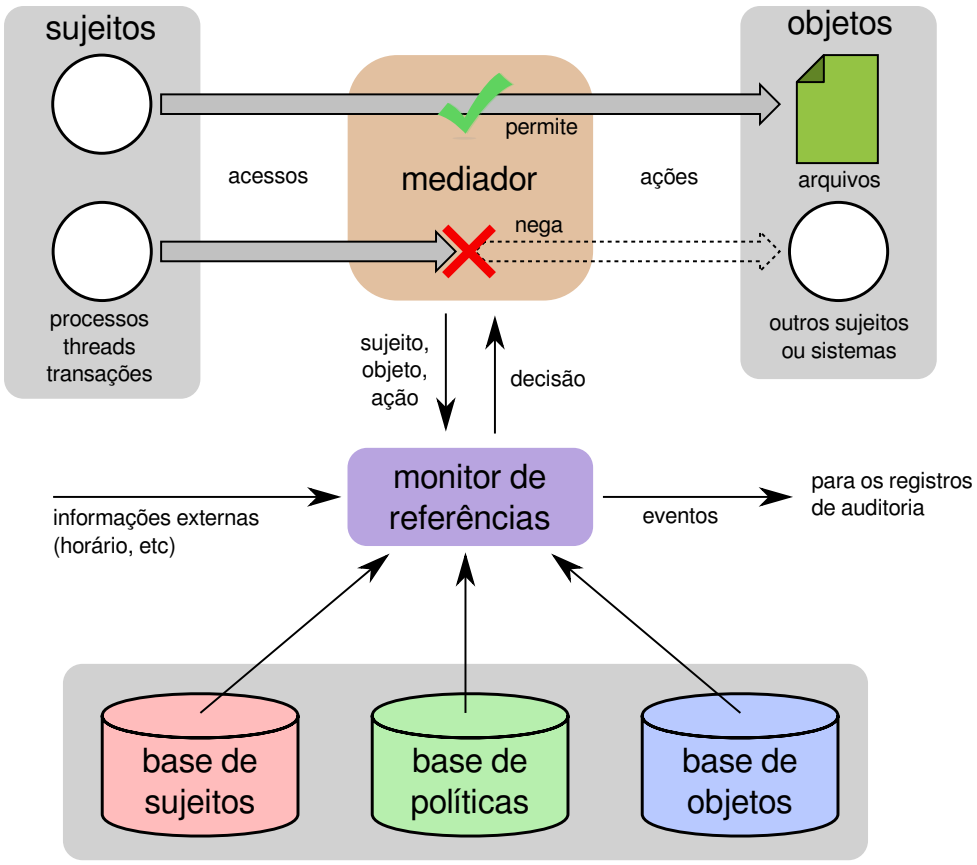


Figura 29.5: Estrutura genérica de uma infraestrutura de controle de acesso.

É importante observar que os elementos dessa estrutura são componentes lógicos, que não impõem uma forma de implementação rígida. Por exemplo, em um

sistema operacional convencional, o sistema de arquivos possui sua própria estrutura de controle de acesso, com permissões de acesso armazenadas nos próprios arquivos, e um pequeno monitor/mediador associado a algumas chamadas de sistema, como `open` e `mmap`. Outros recursos (como áreas de memória ou semáforos) possuem suas próprias regras e estruturas de controle de acesso, organizadas de forma diversa.

29.7.2 Controle de acesso em UNIX

Os sistemas operacionais do mundo UNIX implementam um sistema de ACLs básico bastante rudimentar, no qual existem apenas três sujeitos: *user* (o dono do recurso), *group* (um grupo de usuários ao qual o recurso está associado) e *others* (todos os demais usuários do sistema). Para cada objeto existem três possibilidades de acesso: *read*, *write* e *execute*, cuja semântica depende do tipo de objeto (arquivo, diretório, *socket* de rede, área de memória compartilhada, etc.). Dessa forma, são necessários apenas 9 bits por arquivo para definir suas permissões básicas de acesso.

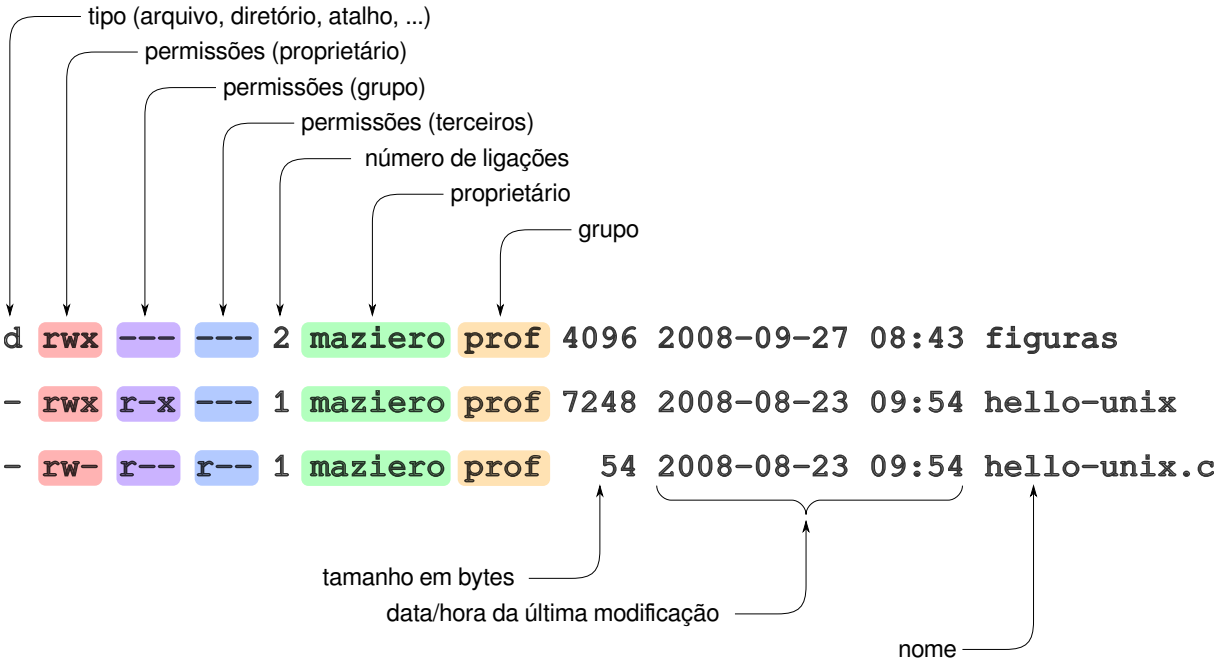


Figura 29.6: Listas de controle de acesso em UNIX.

A Figura 29.6 apresenta uma listagem de diretório típica em UNIX. Nessa listagem, o arquivo `hello-unix.c` pode ser acessado em leitura e escrita por seu proprietário (o usuário `maziero`, com permissões `rw-`), em leitura pelos usuários do grupo `prof` (permissões `r--`) e em leitura pelos demais usuários do sistema (permissões `r--`). Já o arquivo `hello-unix` pode ser acessado em leitura, escrita e execução por seu proprietário (permissões `rwx`), em leitura e execução pelos usuários do grupo `prof` (permissões `r-x`) e não pode ser acessado pelos demais usuários (permissões `---`). No caso de diretórios, a permissão de leitura autoriza a listagem do diretório, a permissão de escrita autoriza sua modificação (criação, remoção ou renomeação de arquivos ou sub-diretórios) e a permissão de execução autoriza usar aquele diretório como diretório de trabalho ou parte de um caminho.

É importante destacar que o controle de acesso é normalmente realizado apenas durante a abertura do arquivo, para a criação de seu descritor em memória. Isso significa

que, uma vez aberto um arquivo por um processo, este terá acesso ao arquivo enquanto o mantiver aberto, mesmo que as permissões do arquivo sejam modificadas para impedir esse acesso. O controle contínuo de acesso a arquivos é pouco frequentemente implementado em sistemas operacionais, porque verificar as permissões de acesso a cada operação de leitura ou escrita teria um forte impacto negativo sobre o desempenho do sistema.

Dessa forma, um descritor de arquivo aberto pode ser visto como uma capacidade (vide Seção 29.3.4), pois a posse do descritor permite ao processo acessar o arquivo referenciado por ele. O processo recebe esse descritor ao abrir o arquivo e deve apresentá-lo a cada acesso subsequente; o descritor pode ser transferido aos processos filhos ou até mesmo a outros processos, outorgando a eles o acesso ao arquivo aberto. A mesma estratégia é usada em *sockets* de rede, semáforos e outros mecanismos de IPC.

O padrão POSIX 1003.1e definiu ACLs mais detalhadas para o sistema de arquivos, que permitem definir permissões para usuários e grupos específicos além do proprietário do arquivo. Esse padrão é parcialmente implementado em vários sistemas operacionais, como o Linux e o FreeBSD. No Linux, os comandos `getfacl` e `setfacl` permitem manipular essas ACLs, como mostra o exemplo a seguir:

```
1  host:~> ll
2  -rw-r--r-- 1 maziero prof 2450791 2009-06-18 10:47 main.pdf
3
4  host:~> getfacl main.pdf
5  # file: main.pdf
6  # owner: maziero
7  # group: maziero
8  user::rw-
9  group::r--
10 other::r--
11
12 host:~> setfacl -m diogo:rw,rafael:rw main.pdf
13
14 host:~> getfacl main.pdf
15 # file: main.pdf
16 # owner: maziero
17 # group: maziero
18 user::rw-
19 user:diogo:rw-
20 user:rafael:rw-
21 group::r--
22 mask::rw-
23 other::r--
```

No exemplo, o comando da linha 12 define permissões de leitura e escrita específicas para os usuários `diogo` e `rafael` sobre o arquivo `main.pdf`. Essas permissões estendidas são visíveis na linha 19 e 20, junto com as permissões UNIX básicas (nas linhas 18, 21 e 23).

29.7.3 Controle de acesso em Windows

Os sistemas Windows baseados no núcleo NT (NT, 2000, XP, Vista e sucessores) implementam mecanismos de controle de acesso bastante sofisticados [Brown, 2000; Russinovich et al., 2008]. Em um sistema Windows, cada sujeito (computador, usuário, grupo ou domínio) é unicamente identificado por um *identificador de segurança* (SID - *Security IDentifier*). Cada sujeito do sistema está associado a um *token de acesso*, criado no momento em que o respectivo usuário ou sistema externo se autentica no sistema. A autenticação e o início da sessão do usuário são gerenciados pelo LSASS (*Local Security Authority Subsystem*), que cria os processos iniciais e os associa ao *token* de acesso criado para aquele usuário. Esse *token* normalmente é herdado pelos processos filhos, até o encerramento da sessão do usuário. Ele contém o identificador do usuário (SID), dos grupos aos quais ele pertence, privilégios a ele associados e outras informações. Privilégios são permissões para realizar operações genéricas, que não dependem de um recurso específico, como reiniciar o computador, carregar um *driver* ou depurar um processo.

Por outro lado, cada objeto do sistema está associado a um *descritor de segurança* (SD - *Security Descriptor*). Como objetos, são considerados arquivos e diretórios, processos, impressoras, serviços e chaves de registros, por exemplo. Um descritor de segurança indica o proprietário e o grupo primário do objeto, uma lista de controle de acesso de sistema (SACL - *System ACL*), uma lista de controle de acesso discricionária (DACL - *Discretionary ACL*) e algumas informações de controle.

A DACL contém uma lista de regras de controle de acesso ao objeto, na forma de ACEs (*Access Control Entries*). Cada ACE contém um identificador de usuário ou grupo, um modo de autorização (positiva ou negativa), um conjunto de permissões (ler, escrever, executar, remover, etc.), sob a forma de um mapa de bits. Quando um sujeito solicita acesso a um recurso, o SRM (*Security Reference Monitor*) compara o *token* de acesso do sujeito com as entradas da DACL do objeto, para permitir ou negar o acesso. Como sujeitos podem pertencer a mais de um grupo e as ACEs podem ser positivas ou negativas, podem ocorrer conflitos entre as ACEs. Por isso, um mecanismo de resolução de conflitos é acionado a cada acesso solicitado ao objeto.

A SACL define que tipo de operações sobre o objeto devem ser registradas pelo sistema, sendo usada basicamente para fins de auditoria (Seção 30). A estrutura das ACEs de auditoria é similar à das ACEs da DACL, embora defina quais ações sobre o objeto em questão devem ser registradas para quais sujeitos. A Figura 29.7 ilustra alguns dos componentes da estrutura de controle de acesso dos sistemas Windows.

29.7.4 Outros mecanismos

As políticas de segurança básicas utilizadas na maioria dos sistemas operacionais são discricionárias, baseadas nas identidades dos usuários e em listas de controle de acesso. Entretanto, políticas de segurança mais sofisticadas vêm sendo gradualmente agregadas aos sistemas operacionais mais complexos, visando aumentar sua segurança. Algumas iniciativas dignas de nota são apresentadas a seguir:

- O SELinux é um mecanismo de controle de acesso multipolíticas, desenvolvido pela NSA (*National Security Agency*, USA) [Loscocco and Smalley, 2001] a partir da arquitetura flexível de segurança *Flask* (*Flux Advanced Security Kernel*) [Spencer et al., 1999]. Ele constitui uma infraestrutura complexa de segurança

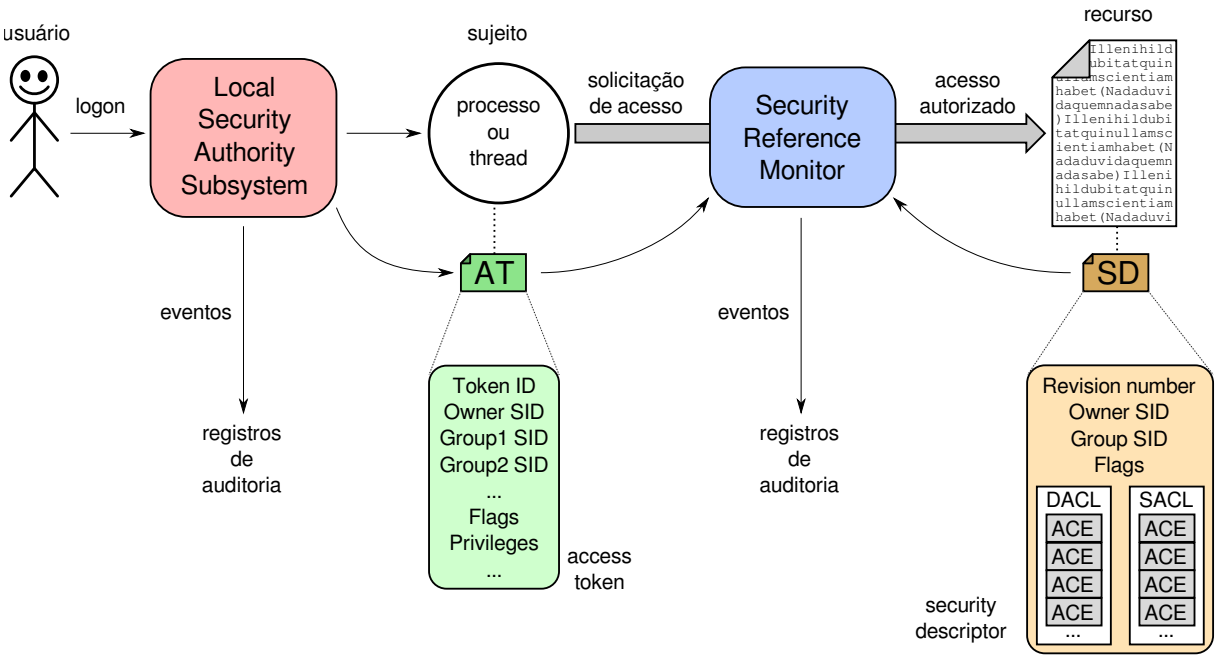


Figura 29.7: Listas de controle de acesso no Windows.

para o núcleo Linux, capaz de aplicar diversos tipos de políticas obrigatórias aos recursos do sistema operacional. A política default do SELinux é baseada em RBAC e DTE, mas ele também é capaz de implementar políticas de segurança multinível. O SELinux tem sido criticado devido à sua complexidade, que torna difícil sua compreensão e configuração. Em consequência, outros projetos visando adicionar políticas MAC mais simples e fáceis de usar ao núcleo Linux têm sido propostos, como *LIDS*, *SMACK* e *AppArmor*.

- O sistema operacional Windows Vista incorpora uma política denominada *Mandatory Integrity Control* (MIC) que associa aos processos e recursos os níveis de integridade *Low*, *Medium*, *High* e *System* [Microsoft], de forma similar ao modelo de Biba (Seção 29.4.2). Os processos normais dos usuários são classificados como de integridade média, enquanto o navegador Web e executáveis provindos da Internet são classificados como de integridade baixa. Além disso, o Vista conta com o UAC (*User Account Control*) que aplica uma política baseada em RBAC: um usuário com direitos administrativos inicia sua sessão como usuário normal, e somente ativa seu papel administrativo quando necessita efetuar uma ação administrativa.

- O projeto TrustedBSD [Watson, 2001] implementa ACLs no padrão POSIX, capacidades POSIX e o suporte a políticas obrigatórias como Bell LaPadula, Biba, categorias e TE/DTE. Uma versão deste projeto foi portada para o MacOS X, sendo denominada *MacOS X MAC Framework*.

- Desenvolvido nos anos 90, o sistema operacional experimental *EROS* (*Extremely Reliable Operating System*) [Shapiro and Hardy, 2002] implementou um modelo de controle de acesso totalmente baseado em capacidades. Nesse modelo, todas as interfaces dos componentes do sistema só são acessíveis através de capacidades, que são usadas para nomear as interfaces e para controlar seu

acesso. O sistema *EROS* deriva de desenvolvimentos anteriores feitos no sistema operacional KeyKOS para a plataforma S/370 [Bomberger et al., 1992].

- Em 2009, o sistema operacional experimental *SeL4*, que estende o sistema micronúcleo L4 [Liedtke, 1996] com um modelo de controle de acesso baseado em capacidades similar ao utilizado no sistema EROS, tornou-se o primeiro sistema operacional cuja segurança foi formalmente verificada [Klein et al., 2009]. A verificação formal é uma técnica de engenharia de software que permite demonstrar matematicamente que a implementação do sistema corresponde à sua especificação, e que a especificação está completa e sem erros.

- O sistema *Trusted Solaris* [Sun Microsystems] implementa várias políticas de segurança: em MLS (*Multi-Level Security*), níveis de segurança são associados aos recursos do sistema e aos usuários. Além disso, a noção de domínios é implementada através de "compartimentos": um recurso associado a um determinado compartimento só pode ser acessado por sujeitos no mesmo compartimento. Para limitar o poder do super-usuário, é usada uma política de tipo RBAC, que divide a administração do sistema em vários papéis que podem ser atribuídos a usuários distintos.

29.8 Mudança de privilégios

Normalmente, os processos em um sistema operacional são sujeitos que representam o usuário que os lançou. Quando um novo processo é criado, ele herda as credenciais de seu processo-pai, ou seja, seus identificadores de usuário e de grupo. Na maioria dos mecanismos de controle de acesso usados em sistemas operacionais, as permissões são atribuídas aos processos em função de suas credenciais. Com isso, normalmente cada novo processo herda as mesmas permissões de seu processo-pai, pois possui as mesmas credenciais dele.

O uso de privilégios fixos é adequado para o uso normal do sistema, pois os processos de cada usuário só devem ter acesso aos recursos autorizados para esse usuário. Entretanto, em algumas situações esse mecanismo se mostra inadequado. Por exemplo, caso um usuário precise executar uma tarefa administrativa, como instalar um novo programa, modificar uma configuração de rede ou atualizar sua senha, alguns de seus processos devem possuir permissões para as ações necessárias, como editar arquivos de configuração do sistema. Os sistemas operacionais atuais oferecem diversas abordagens para resolver esse problema:

Usuários administrativos: são associadas permissões administrativas às sessões de trabalho de alguns usuários específicos, permitindo que seus processos possam efetuar tarefas administrativas, como instalar softwares ou mudar configurações. Esta é a abordagem utilizada em alguns sistemas operacionais de amplo uso. Algumas implementações definem vários tipos de usuários administrativos, com diferentes tipos de privilégios, como acessar dispositivos externos, lançar máquinas virtuais, reiniciar o sistema, etc. Embora simples, essa solução é falha, pois se algum programa com conteúdo malicioso for executado por um usuário administrativo, terá acesso a todas as suas permissões.

Permissões temporárias: conceder sob demanda a certos processos do usuário as permissões de que necessitam para realizar ações administrativas; essas permissões podem ser descartadas pelo processo assim que concluir as ações. Essas permissões podem estar associadas a papéis administrativos (Seção 29.6), ativados quando o usuário tiver necessidade deles. Esta é a abordagem usada pela infraestrutura UAC (*User Access Control*) [Microsoft], na qual um usuário administrativo inicia sua sessão de trabalho como usuário normal, e somente ativa seu papel administrativo quando necessita efetuar uma ação administrativa, desativando-o imediatamente após a conclusão da ação. A ativação do papel administrativo pode impor um procedimento de reautenticação.

Mudança de credenciais: permitir que certos processos do usuário mudem de identidade, assumindo a identidade de algum usuário com permissões suficientes para realizar a ação desejada; pode ser considerada uma variante da atribuição de permissões temporárias. O exemplo mais conhecido de implementação desta abordagem são os flags `setuid` e `setgid` do UNIX, explicados a seguir.

Monitores: definir processos privilegiados, chamados *monitores* ou *supervisores*, recebem pedidos de ações administrativas dos processos não privilegiados, através de uma API pré-definida; os pedidos dos processos normais são validados e atendidos. Esta é a abordagem definida como *separação de privilégios* em [Provos et al., 2003], e também é usada na infra-estrutura *PolicyKit*, usada para autorizar tarefas administrativas em ambientes *desktop* Linux.

Um mecanismo amplamente usado para mudança de credenciais consiste dos flags `setuid` e `setgid` dos sistemas UNIX. Se um arquivo executável tiver o flag `setuid` habilitado (indicado pelo caractere "s" em suas permissões de usuário), seus processos assumirão as credenciais do proprietário do arquivo. Portanto, se o proprietário de um arquivo executável for o usuário *root*, os processos lançados a partir dele terão todos os privilégios do usuário *root*, independente de quem o tiver lançado. De forma similar, processos lançados a partir de um arquivo executável com o flag `setgid` habilitado terão as credenciais do grupo associado ao arquivo. A Figura 29.8 ilustra esse mecanismo: o primeiro caso representa um executável normal (sem esses flags habilitados); um processo filho lançado a partir do executável possui as mesmas credenciais de seu pai. No segundo caso, o executável pertence ao usuário *root* e tem o flag `setuid` habilitado; assim, o processo filho assume a identidade do usuário *root* e, em consequência, suas permissões de acesso. No último caso, o executável pertence ao usuário *root* e tem o flag `setgid` habilitado; assim, o processo filho pertencerá ao grupo *mail*.

Os flags `setuid` e `setgid` são muito utilizados em programas administrativos no UNIX, como troca de senha e agendamento de tarefas, sempre que for necessário efetuar uma operação inacessível a usuários normais, como modificar o arquivo de senhas. Todavia, esse mecanismo pode ser perigoso, pois o processo filho recebe todos os privilégios do proprietário do arquivo, o que contraria o princípio do privilégio mínimo. Por exemplo, o programa `passwd` deveria somente receber a autorização para modificar o arquivo de senhas (`/etc/passwd`) e nada mais, pois o superusuário (*root user*) tem acesso a todos os recursos do sistema e pode efetuar todas as operações que desejar. Se o programa `passwd` contiver erros de programação, ele pode ser induzido pelo seu usuário a efetuar ações não previstas, visando comprometer a segurança do sistema (vide Seção 26.3).

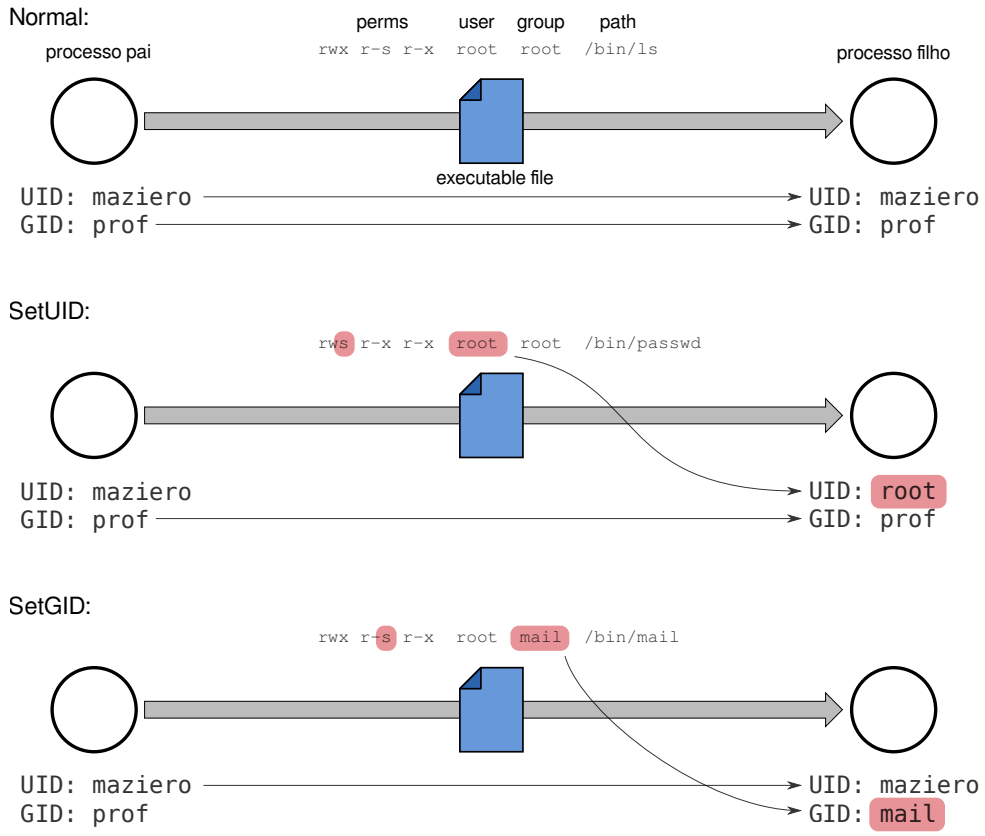


Figura 29.8: Funcionamento dos flags `setuid` e `setgid` do UNIX.

Uma alternativa mais segura aos flags `setuid` e `setgid` são os *privilégios POSIX* (*POSIX Capabilities*[2]), definidos no padrão POSIX 1003.1e [Gallmeister, 1994]. Nessa abordagem, o "poder absoluto" do super usuário é dividido em um grande número de pequenos privilégios específicos, que podem ser atribuídos a certos processos do sistema. Como medida adicional de proteção, cada processo pode ativar/desativar os privilégios que possui em função de sua necessidade. Vários sistemas UNIX implementam privilégios POSIX, como é o caso do Linux, que implementa:

- `CAP_CHOWN`: alterar o proprietário de um arquivo qualquer;
- `CAP_USER_DEV`: abrir dispositivos;
- `CAP_USER_FIFO`: usar *pipes* (comunicação);
- `CAP_USER_SOCK`: abrir *sockets* de rede;
- `CAP_NET_BIND_SERVICE`: abrir portas de rede com número abaixo de 1024;
- `CAP_NET_RAW`: abrir *sockets* de baixo nível (*raw sockets*);
- `CAP_KILL`: enviar sinais para processos de outros usuários.
- ... (outros +30 privilégios)

[2]O padrão POSIX usou indevidamente o termo "capacidade" para definir o que na verdade são privilégios associados aos processos. O uso indevido do termo *POSIX Capabilities* perdura até hoje em vários sistemas, como é o caso do Linux.

Para cada processo são definidos três conjuntos de privilégios: *Permitidos* (*P*), *Efetivos* (*E*) e *Herdáveis* (*H*). Os privilégios permitidos são aqueles que o processo pode ativar quando desejar, enquanto os efetivos são aqueles ativados no momento (respeitando-se $E \subseteq P$). O conjunto de privilégios herdáveis *H* é usado no cálculo dos privilégios transmitidos aos processos filhos. Os privilégios POSIX também podem ser atribuídos a programas executáveis em disco, substituindo os tradicionais (e perigosos) flags `setuid` e `setgid`. Assim, quando um executável for lançado, o novo processo recebe um conjunto de privilégios calculado a partir dos privilégios atribuídos ao arquivo executável e aqueles herdados do processo-pai que o criou [Bovet and Cesati, 2005].

Um caso especial de mudança de credenciais ocorre em algumas circunstâncias, quando é necessário **reduzir** as permissões de um processo. Por exemplo, o processo responsável pela autenticação de usuários em um sistema operacional deve criar novos processos para iniciar a sessão de trabalho de cada usuário. O processo autenticador geralmente executa com privilégios elevados, para poder acessar a bases de dados de autenticação dos usuários, enquanto os novos processos devem receber as credenciais do usuário autenticado, que normalmente tem menos privilégios. Em UNIX, um processo pode solicitar a mudança de suas credenciais através da chamada de sistema `setuid()`, entre outras. Em Windows, o mecanismo conhecido como *impersonation* permite a um processo ou *thread* abandonar temporariamente seu *token* de acesso e assumir outro, para realizar uma tarefa em nome do sujeito correspondente [Russinovich et al., 2008].

Referências

R. Anderson. *Security engineering*. John Wiley & Sons, 2008.

L. Badger, D. Sterne, D. Sherman, K. Walker, and S. Haghighat. Practical Domain and Type Enforcement for UNIX. In *IEEE Symposium on Security and Privacy*, pages 66–77, 1995.

D. E. Bell and L. J. LaPadula. Secure computer systems. mathematical foundations and model. Technical Report M74-244, MITRE Corporation, 1974.

K. Biba. Integrity considerations for secure computing systems. Technical Report MTR-3153, MITRE Corporation, 1977.

W. Boebert and R. Kain. A practical alternative to hierarchical integrity policies. In *8th National Conference on Computer Security*, pages 18–27, 1985.

A. Bomberger, A. Frantz, W. Frantz, A. Hardy, N. Hardy, C. Landau, and J. Shapiro. The KeyKOS nanokernel architecture. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, 1992.

D. Bovet and M. Cesati. *Understanding the Linux Kernel, 3rd edition*. O'Reilly Media, Inc, 2005.

K. Brown. *Programming Windows Security*. Addison-Wesley Professional, 2000.

C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious server security. In *14th USENIX Systems Administration Conference*, 2000.

S. di Vimercati, P. Samarati, and S. Jajodia. Policies, Models, and Languages for Access Control. In *Workshop on Databases in Networked Information Systems*, volume LNCS 3433, pages 225–237. Springer-Verlag, 2005.

S. di Vimercati, S. Foresti, S. Jajodia, and P. Samarati. Access control policies and languages in open environments. In T. Yu and S. Jajodia, editors, *Secure Data Management in Decentralized Systems*, volume 33 of *Advances in Information Security*, pages 21–58. Springer, 2007.

B. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly Media, Inc, 1994.

V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. Guide to attribute based access control (ABAC) definition and considerations. Technical Report 800-162, NIST - National Institute of Standards and Technology, 2014.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. SeL4: Formal verification of an OS kernel. In *22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, Oct 2009.

B. Lampson. Protection. In *5th Princeton Conference on Information Sciences and Systems*, 1971. Reprinted in ACM Operating Systems Rev. 8, 1 (Jan. 1974), pp 18-24.

J. Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996.

P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *USENIX Annual Technical Conference*, pages 29–42, 2001.

Microsoft. *Security Enhancements in Windows Vista*. Microsoft Corporation, May 2007.

N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, 2003.

M. Russinovich, D. Solomon, and A. Ionescu. *Microsoft Windows Internals, Fifth Edition*. Microsoft Press, 2008.

J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278 – 1308, September 1975.

P. Samarati and S. De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*. Springer-Verlag, 2001.

R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, Feb. 1996.

J. Shapiro and N. Hardy. Eros: a principle-driven operating system from the ground up. *Software, IEEE*, 19(1):26–33, Jan/Feb 2002. ISSN 0740-7459. doi: 10.1109/52.976938.

R. Shirey. RFC 2828: Internet security glossary, May 2000.

R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *8th USENIX Security Symposium*, pages 123–139, 1999.

Sun Microsystems. *Trusted Solaris User's Guide*. Sun Microsystems, Inc, June 2000.

R. Watson. TrustedBSD: Adding trusted operating system features to FreeBSD. In *USENIX Technical Conference*, 2001.

Capítulo 30

Mecanismos de auditoria

30.1 Introdução

Na área de segurança de sistemas, o termo "auditar" significa recolher dados sobre o funcionamento de um sistema ou aplicação e analisá-los para descobrir vulnerabilidades ou violações de segurança, ou para examinar violações já constatadas, buscando suas causas e possíveis consequências[1] [Sandhu and Samarati, 1996]. Os dois pontos-chave da auditoria são portanto a *coleta* de dados e a *análise* desses dados, que serão discutidas a seguir.

30.2 Coleta de dados

Um sistema computacional em funcionamento processa uma grande quantidade de eventos. Destes, alguns podem ser de importância para a segurança do sistema, como a autenticação de um usuário (ou uma tentativa mal-sucedida de autenticação), uma mudança de credenciais, o lançamento ou encerramento de um serviço, etc. Os dados desses eventos devem ser coletados a partir de suas fontes e registrados de forma adequada para a análise e arquivamento.

Dependendo da natureza do evento, a coleta de seus dados pode ser feita no nível da aplicação, de subsistema ou do núcleo do sistema operacional:

Aplicação: eventos internos à aplicação, cuja semântica é específica ao seu contexto. Por exemplo, as ações realizadas por um servidor HTTP, como páginas fornecidas, páginas não encontradas, erros de autenticação, pedidos de operações não suportadas, etc. Normalmente esses eventos são registrados pela própria aplicação, muitas vezes usando formatos próprios para os dados.

Subsistema: eventos não específicos a uma aplicação, mas que ocorrem no espaço de usuário do sistema operacional. Exemplos desses eventos são a autenticação de usuários (ou erros de autenticação), lançamento ou encerramento de serviços do sistema, atualizações de softwares ou de bibliotecas, criação ou remoção de usuários, etc. O registro desses eventos normalmente fica a cargo dos processos ou bibliotecas responsáveis pelos respectivos subsistemas.

[1] A análise de violações já ocorridas é comumente conhecida como *análise postmortem*.

Núcleo: eventos que ocorrem dentro do núcleo do sistema, sendo inacessíveis aos processos. É o caso dos eventos envolvendo o hardware, como a detecção de erros ou mudança de configurações, e de outros eventos internos do núcleo, como a criação de *sockets* de rede, semáforos, área de memória compartilhada, reinicialização do sistema, etc.

Um aspecto importante da coleta de dados para auditoria é sua forma de representação. A abordagem mais antiga e comum, amplamente disseminada, é o uso de arquivos de registro (*logfiles*). Um arquivo de registro contém uma sequência cronológica de descrições textuais de eventos associados a uma fonte de dados, geralmente uma linha por evento. Um exemplo clássico dessa abordagem são os arquivos de registro do sistema UNIX; a listagem a seguir apresenta um trecho do conteúdo do arquivo `/var/log/security`, geralmente usado para reportar eventos associados à autenticação de usuários:

```
...
Sep  8 23:02:09 espec sudo: e89602174 : user NOT in sudoers ; TTY=pts/1 ; USER=root ; COMMAND=/bin/su
Sep  8 23:19:57 espec userhelper[20480:] running '/sbin/halt' with user_u:system_r:hotplug_t context
Sep  8 23:34:14 espec sshd[6302:] pam_unix(sshd:auth): failure; rhost=210.210.102.173  user=root
Sep  8 23:57:16 espec sshd[6302:] Failed password for root from 210.103.210.173 port 14938 ssh2
Sep  8 00:08:16 espec sshd[6303:] Received disconnect from 210.103.210.173: 11: Bye Bye
Sep  8 00:35:24 espec gdm[9447:] pam_unix(gdm:session): session opened for user rodr by (uid=0)
Sep  8 00:42:19 espec gdm[857:] pam_unix(gdm:session): session closed for user rafael3
Sep  8 00:49:06 espec userhelper[11031:] running '/sbin/halt' with user_u:system_r:hotplug_t context
Sep  8 00:53:40 espec gdm[12199:] pam_unix(gdm:session): session opened for user rafael3 by (uid=0)
Sep  8 00:53:55 espec gdm[12199:] pam_unix(gdm:session): session closed for user rafael3
Sep  8 01:08:43 espec gdm[9447:] pam_unix(gdm:session): session closed for user rodr
Sep  8 01:12:41 espec sshd[14125:] Accepted password for rodr from 189.30.227.212 port 1061 ssh2
Sep  8 01:12:41 espec sshd[14125:] pam_unix(sshd:session): session opened for user rodr by (uid=0)
Sep  8 01:12:41 espec sshd[14127:] subsystem request for sftp
Sep  8 01:38:26 espec sshd[14125:] pam_unix(sshd:session): session closed for user rodr
Sep  8 02:18:29 espec sshd[17048:] Accepted password for e89062004 from 20.0.0.56 port 54233 ssh2
Sep  8 02:18:29 espec sshd[17048:] pam_unix(sshd:session): session opened for user e89062004 by (uid=0)
Sep  8 02:18:29 espec sshd[17048:] pam_unix(sshd:session): session closed for user e89062004
Sep  8 09:06:33 espec sshd[25002:] Postponed publickey for mzr from 159.71.224.62 port 52372 ssh2
Sep  8 06:06:34 espec sshd[25001:] Accepted publickey for mzr from 159.71.224.62 port 52372 ssh2
Sep  8 06:06:34 espec sshd[25001:] pam_unix(sshd:session): session opened for user mzr by (uid=0)
Sep  8 06:06:57 espec su: pam_unix(su-l:session): session opened for user root by mzr(uid=500)
...
```

A infraestrutura tradicional de registro de eventos dos sistemas UNIX é constituída por um *daemon*[2] chamado `syslogd` (*System Log Daemon*). Esse *daemon* usa um *socket* local e um socket UDP para receber mensagens descrevendo eventos, geradas pelos demais subsistemas e aplicações através de uma biblioteca específica. Os eventos são descritos por mensagens de texto e são rotulados por suas fontes em *serviços* (`AUTH`, `KERN`, `MAIL`, etc.) e *níveis* (`INFO`, `WARNING`, `ALERT`, etc.). A partir de seu arquivo de configuração, o processo `syslogd` registra a data de cada evento recebido e decide seu destino: armazenar em um arquivo, enviar a um terminal, avisar o administrador, ativar um programa externo ou enviar o evento a um *daemon* em outro computador são as principais possibilidades. A Figura 30.1 apresenta os principais componentes dessa arquitetura.

Os sistemas Windows mais recentes usam uma arquitetura similar, embora mais sofisticada do ponto de vista do formato dos dados, pois os eventos são descritos em formato XML (a partir do Windows Vista). O serviço *Windows Event Log* assume o

[2]Processo que executa em segundo plano, sem estar associado a uma interface com o usuário, como um terminal ou janela.

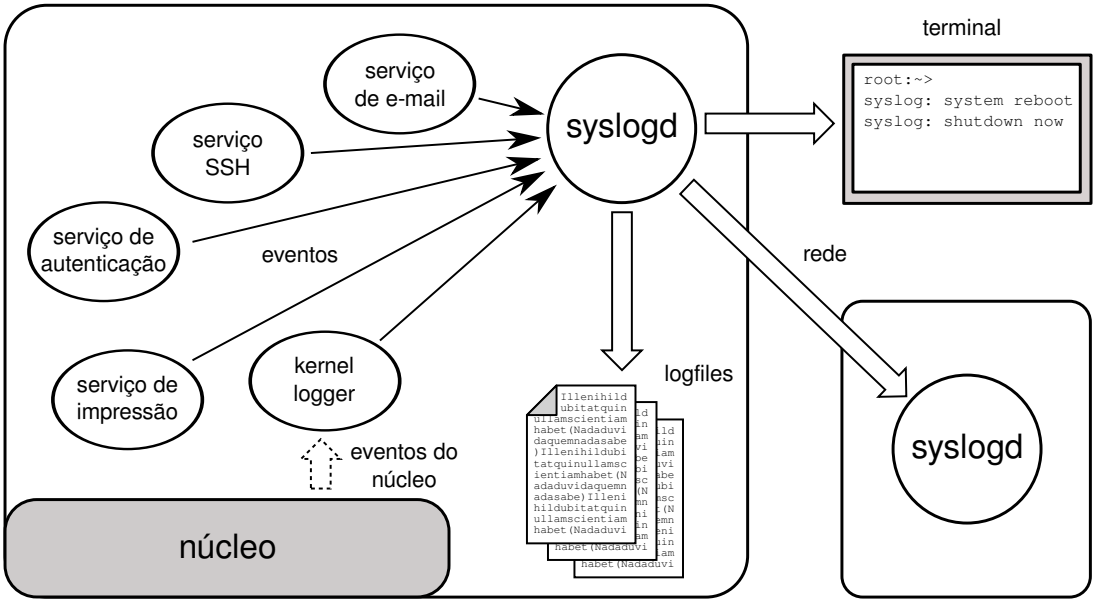


Figura 30.1: O serviço de *logs* em UNIX.

papel de centralizador de eventos, recebendo mensagens de várias fontes, entre elas os componentes do subsistema de segurança (LSASS e SRM, Seção 29.7.3), as aplicações e o próprio núcleo. Conforme visto anteriormente, o componente LSASS gera eventos relativos à autenticação dos usuários, enquanto o SRM registra os acessos a cada objeto de acordo com as regras de auditoria definidas em sua SACL (*System ACLs*). Além disso, aplicações externas podem se registrar junto ao sistema de logs para receber eventos de interesse, através de uma interface de acesso baseada no modelo *publish/subscribe*.

Além dos exemplos aqui apresentados, muitos sistemas operacionais implementam arquiteturas específicas para auditoria, como é o caso do BSM (*Basic Security Module*) do sistema Solaris e sua implementação OpenBSM para o sistema operacional OpenBSD. O sistema MacOS X também provê uma infraestrutura de auditoria, na qual o administrador pode registrar os eventos de seu interesse e habilitar a geração de registros.

Além da coleta de eventos do sistema à medida em que eles ocorrem, outras formas de coleta de dados para auditoria são frequentes. Por exemplo, ferramentas de segurança podem vasculhar o sistema de arquivos em busca de arquivos com conteúdo malicioso, ou varrer as portas de rede para procurar serviços suspeitos.

30.3 Análise de dados

Uma vez registrada a ocorrência de um evento de interesse para a segurança do sistema, deve-se proceder à sua análise. O objetivo dessa análise é sobretudo identificar possíveis violações da segurança em andamento ou já ocorridas. Essa análise pode ser feita sobre os registros dos eventos à medida em que são gerados (chamada análise *online*) ou sobre registros previamente armazenados (análise *offline*). A análise *online* visa detectar problemas de segurança com rapidez, para evitar que comprometam o sistema. Como essa análise deve ser feita simultaneamente ao funcionamento do sistema, é importante que seja rápida e leve, para não prejudicar o desempenho do sistema nem

interferir nas operações em andamento. Um exemplo típico de análise online são os antivírus, que analisam os arquivos à medida em que estes são acessados pelos usuários.

Por sua vez, a análise *offline* é realizada com dados previamente coletados, possivelmente de vários sistemas. Como não tem compromisso com uma resposta imediata, pode ser mais profunda e detalhada, permitindo o uso de técnicas de mineração de dados para buscar correlações entre os registros, que possam levar à descoberta de problemas de segurança mais sutis. A análise offline é usada em sistemas de detecção de intrusão, por exemplo, para analisar a história do comportamento de cada usuário. Além disso, é frequentemente usada em sistemas de informação bancários, para se analisar o padrão de uso dos cartões de débito e crédito dos correntistas e identificar fraudes.

As ferramentas de análise de registros de segurança podem adotar basicamente duas abordagens: análise por assinaturas ou análise por anomalias. Na *análise por assinaturas*, a ferramenta tem acesso a uma base de dados contendo informações sobre os problemas de segurança conhecidos que deve procurar. Se algum evento ou registro se encaixar nos padrões descritos nessa base, ele é considerado uma violação de segurança. Um exemplo clássico dessa abordagem são os programas antivírus: um antivírus típico varre o sistema de arquivos em busca de conteúdos maliciosos. O conteúdo de cada arquivo é verificado junto a uma *base de assinaturas*, que contém descrições detalhadas dos vírus conhecidos pelo software; se o conteúdo de um arquivo coincidir com uma assinatura da base, aquele arquivo é considerado suspeito. Um problema com essa forma de análise é sua incapacidade de detectar novas ameaças, como vírus desconhecidos, cuja assinatura não esteja na base.

Por outro lado, uma ferramenta de *análise por anomalias* conta com uma base de dados descrevendo o que se espera como comportamento ou conteúdo normal do sistema. Eventos ou registros que não se encaixarem nesses padrões de normalidade são considerados como violações potenciais da segurança, sendo reportados ao administrador do sistema. A análise por anomalias, também chamada de análise baseada em heurísticas, é utilizada em certos tipos de antivírus e sistemas de detecção de intrusão, para detectar vírus ou ataques ainda desconhecidos. Também é muito usada em sistemas de informação bancários, para detectar fraudes envolvendo o uso das contas e cartões bancários. O maior problema com esta técnica é caracterizar corretamente o que se espera como comportamento "normal", o que pode ocasionar muitos erros.

30.4 Auditoria preventiva

Além da coleta e análise de dados sobre o funcionamento do sistema, a auditoria pode agir de forma "preventiva", buscando problemas potenciais que possam comprometer a segurança do sistema. Há um grande número de ferramentas de auditoria, que abordam aspectos diversos da segurança do sistema, entre elas [Pfleeger and Pfleeger, 2006]:

- *Vulnerability scanner*: verifica os softwares instalados no sistema e confronta suas versões com uma base de dados de vulnerabilidades conhecidas, para identificar possíveis fragilidades. Pode também investigar as principais configurações do sistema, com o mesmo objetivo. Como ferramentas deste tipo podem ser citadas: *Metasploit*, *Nessus Security Scanner* e SAINT (*System Administrator's Integrated Network Tool*).

- *Port scanner*: analisa as portas de rede abertas em um computador remoto, buscando identificar os serviços de rede oferecidos pela máquina, as versões do softwares que atendem esses serviços e a identificação do próprio sistema operacional subjacente. O *NMap* é provavelmente o *scanner* de portas mais conhecido atualmente.

- *Password cracker*: conforme visto na Seção 28.4, as senhas dos usuários de um sistema são armazenadas na forma de resumos criptográficos, para aumentar sua segurança. Um "quebrador de senhas" tem por finalidade tentar descobrir as senhas dos usuários, para avaliar sua robustez. A técnica normalmente usada por estas ferramentas é o ataque do dicionário, que consiste em testar um grande número de palavras conhecidas, suas variantes e combinações, confrontando seus resumos com os resumos das senhas armazenadas. Quebradores de senhas bem conhecidos são o *John the Ripper* para UNIX e *Cain and Abel* para ambientes Windows.

- *Rootkit scanner*: visa detectar a presença de *rootkits* (vide Seção 26.2) em um sistema, normalmente usando uma técnica *offline* baseada em assinaturas. Como os *rootkits* podem comprometer até o núcleo do sistema operacional instalado no computador, normalmente as ferramentas de detecção devem ser aplicadas a partir de outro sistema, carregado a partir de uma mídia externa confiável (CD ou DVD).

- *Verificador de integridade*: a segurança do sistema operacional depende da integridade do núcleo e dos utilitários necessários à administração do sistema. Os verificadores de integridade são programas que analisam periodicamente os principais arquivos do sistema operacional, comparando seu conteúdo com informações previamente coletadas. Para agilizar a verificação de integridade são utilizadas somas de verificação (*checksums*) ou resumos criptográficos como o MD5 e SHA1. Essa verificação de integridade pode se estender a outros objetos do sistema, como a tabela de chamadas de sistema, as portas de rede abertas, os processos de sistema em execução, o cadastro de softwares instalados, etc. Um exemplo clássico de ferramenta de verificação de integridade é o *Tripwire* [Tripwire, 2003], mas existem diversas outras ferramentas mais recentes com propósitos similares.

Referências

C. Pfleeger and S. L. Pfleeger. *Security in Computing, 4th Edition*. Prentice Hall PTR, 2006.

R. Sandhu and P. Samarati. Authentication, access control, and audit. *ACM Computing Surveys*, 28(1), 1996.

Tripwire. The Tripwire open source project. `http://www.tripwire.org`, 2003.

Parte VIII

Virtualização

Capítulo 31

O conceito de virtualização

As tecnologias de virtualização do ambiente de execução de aplicações ou de plataformas de hardware têm sido objeto da atenção crescente de pesquisadores, fabricantes de hardware/software, administradores de sistemas e usuários avançados. A virtualização de recursos é um conceito relativamente antigo, mas os recentes avanços nessa área permitem usar máquinas virtuais com os mais diversos objetivos, como a segurança, a compatibilidade de aplicações legadas ou a consolidação de servidores. Este capítulo apresenta os principais conceitos relacionados à virtualização.

31.1 Um breve histórico

O conceito de máquina virtual não é recente. Os primeiros passos na construção de ambientes de máquinas virtuais começaram na década de 1960, quando a IBM desenvolveu o sistema operacional experimental M44/44X. A partir dele, a IBM desenvolveu vários sistemas comerciais suportando virtualização, entre os quais o famoso OS/370 [Goldberg, 1973; Goldberg and Mager, 1979]. A tendência dominante nos sistemas naquela época era fornecer a cada usuário um ambiente monousuário completo, com seu próprio sistema operacional e aplicações, completamente independente e desvinculado dos ambientes dos demais usuários.

Na década de 1970, os pesquisadores Popek & Goldberg formalizaram vários conceitos associados às máquinas virtuais, e definiram as condições necessárias para que uma plataforma de hardware suporte de forma eficiente a virtualização [Popek and Goldberg, 1974]; essas condições são discutidas em detalhe no Capítulo 33.1. Nessa mesma época surgem as primeiras experiências concretas de utilização de máquinas virtuais para a execução de aplicações, com o ambiente *UCSD p-System*, no qual programas Pascal eram compilados para execução sobre um hardware virtual denominado *P-Machine*.

Na década de 1980, com a popularização de plataformas de hardware baratas como o PC, a virtualização perdeu importância. Afinal, era mais barato, simples e versátil fornecer um computador completo a cada usuário, que investir em sistemas de grande porte, caros e complexos. Além disso, o hardware do PC tinha desempenho modesto e não provia suporte adequado à virtualização, o que inibiu o uso de ambientes virtuais nessas plataformas.

Com o aumento de desempenho e funcionalidades do hardware PC e o surgimento da linguagem Java, nos anos 90, o interesse pelas tecnologias de virtualização voltou à tona. Apesar da plataforma PC Intel na época não oferecer um suporte ade-

quado à virtualização, soluções engenhosas como as adotadas pela empresa VMware permitiram a virtualização nessa plataforma, embora com desempenho relativamente modesto. Nos anos 2000 as soluções de virtualização de plataformas despertou grande interesse do mercado, devido à consolidação de servidores e à construção das nuvens computacionais. Hoje, várias linguagens são compiladas para máquinas virtuais portáveis e os processadores mais recentes trazem um suporte nativo à virtualização do hardware.

31.2 Interfaces de sistema

Uma máquina real é formada por vários componentes físicos que fornecem operações para o sistema operacional e suas aplicações. Iniciando pelo núcleo do sistema real, o processador central (CPU) e o *chipset* da placa-mãe fornecem um conjunto de instruções e outros elementos fundamentais para o processamento de dados, alocação de memória e processamento de entrada/saída. Os sistemas de computadores são projetados com basicamente três componentes: hardware, sistema operacional e aplicações. O papel do hardware é executar as operações solicitadas pelas aplicações através do sistema operacional. O sistema operacional recebe as solicitações das operações (por meio das chamadas de sistema) e controla o acesso ao hardware – principalmente nos casos em que os componentes são compartilhados, como a memória e os dispositivos de entrada/saída.

Os sistemas de computação convencionais são caracterizados por níveis de abstração crescentes e interfaces bem definidas entre eles. As abstrações oferecidas pelo sistema às aplicações são construídas de forma incremental, em níveis separados por interfaces bem definidas e relativamente padronizadas. Cada interface encapsula as abstrações dos níveis inferiores, permitindo assim o desenvolvimento independente dos vários níveis, o que simplifica a construção e evolução dos sistemas. As interfaces existentes entre os componentes de um sistema de computação típico são:

Conjunto de instruções (*ISA – Instruction Set Architecture*): é a interface básica entre o hardware e o software, sendo constituída pelas instruções de máquina aceitas pelo processador e as operações de acesso aos recursos do hardware (acesso físico à memória, às portas de entrada/saída, ao relógio do hardware, etc.). Essa interface é dividida em duas partes:

- **Instruções de usuário (*User ISA*):** compreende as instruções do processador e demais itens de hardware acessíveis aos programas do usuário, que executam com o processador operando em modo usuário;
- **Instruções de sistema (*System ISA*):** compreende as instruções do processador e demais itens de hardware unicamente acessíveis ao núcleo do sistema operacional, que executa em modo privilegiado;

Chamadas de sistema (*syscalls*): é o conjunto de operações oferecidas pelo núcleo do sistema operacional aos processos no espaço de usuário. Essas chamadas permitem o acesso controlado das aplicações aos dispositivos periféricos, à memória e às instruções privilegiadas do processador.

Chamadas de bibliotecas (*libcalls*): as bibliotecas oferecem um grande número de funções para simplificar a construção de programas; além disso, muitas chamadas de biblioteca encapsulam chamadas do sistema operacional, para tornar seu uso mais simples. Cada biblioteca possui uma interface própria, denominada *Interface de Programação de Aplicações* (*API – Application Programming Interface*). Exemplos típicos de bibliotecas são a *LibC* do UNIX (que oferece funções como `fopen` e `printf`), a GTK+ (*Gimp ToolKit*, que permite a construção de interfaces gráficas) e a SDL (*Simple DirectMedia Layer*, para a manipulação de áudio e vídeo).

A Figura 31.1 apresenta essa visão conceitual da arquitetura de um sistema computacional, com seus vários componentes e as respectivas interfaces entre eles.

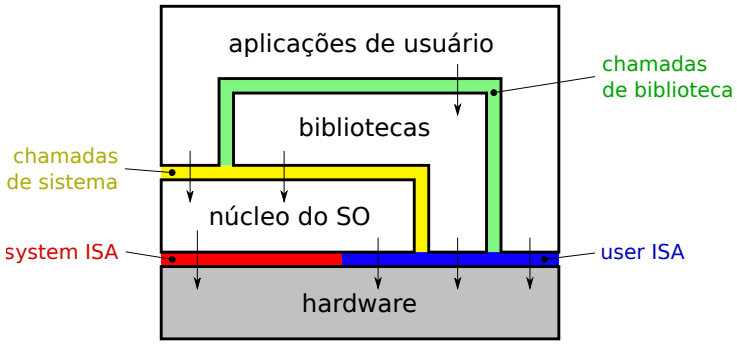


Figura 31.1: Componentes e interfaces de um sistema computacional.

31.3 Compatibilidade entre interfaces

Para que programas e bibliotecas possam executar sobre uma determinada plataforma, é necessário que tenham sido compilados para ela, respeitando o conjunto de instruções do processador em modo usuário (*User ISA*) e o conjunto de chamadas de sistema oferecido pelo sistema operacional. A visão conjunta dessas duas interfaces (*User ISA + syscalls*) é denominada *Interface Binária de Aplicação* (*ABI – Application Binary Interface*). Da mesma forma, um sistema operacional só poderá executar sobre uma plataforma de hardware se tiver sido construído e compilado de forma a respeitar sua interface ISA (*User/System ISA*). A Figura 31.2 representa essas duas interfaces.

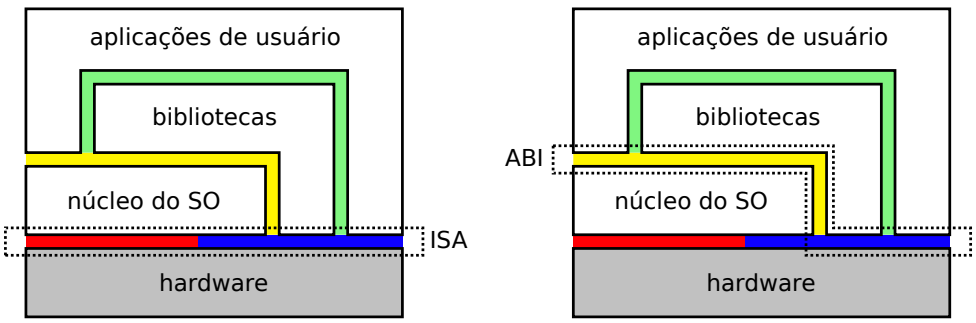


Figura 31.2: Interfaces de sistema ISA e ABI [Smith and Nair, 2004].

Nos sistemas computacionais de mercado atuais, as interfaces de baixo nível ISA e ABI são normalmente fixas, ou pouco flexíveis. Geralmente não é possível criar novas instruções de processador ou novas chamadas de sistema operacional, ou mesmo mudar sua semântica para atender às necessidades específicas de uma determinada aplicação. Mesmo se isso fosse possível, teria de ser feito com cautela, para não comprometer o funcionamento de outras aplicações.

Os sistemas operacionais, assim como as aplicações, são projetados para aproveitar o máximo dos recursos que o hardware fornece. Normalmente os projetistas de hardware, sistema operacional e aplicações trabalham de forma independente (em empresas e tempos diferentes). Por isso, esses trabalhos independentes geraram, ao longo dos anos, várias plataformas computacionais diferentes e incompatíveis entre si.

Observa-se então que, embora a definição de interfaces seja útil, por facilitar o desenvolvimento independente dos vários componentes do sistema, torna pouco flexíveis as interações entre eles: um sistema operacional só funciona sobre o hardware (ISA) para o qual foi construído, uma biblioteca só funciona sobre a ABI para a qual foi projetada e uma aplicação tem de obedecer a ABIs e APIs pré-definidas. A Figura 31.3, extraída de [Smith and Nair, 2004], ilustra esses problemas de compatibilidade entre interfaces.

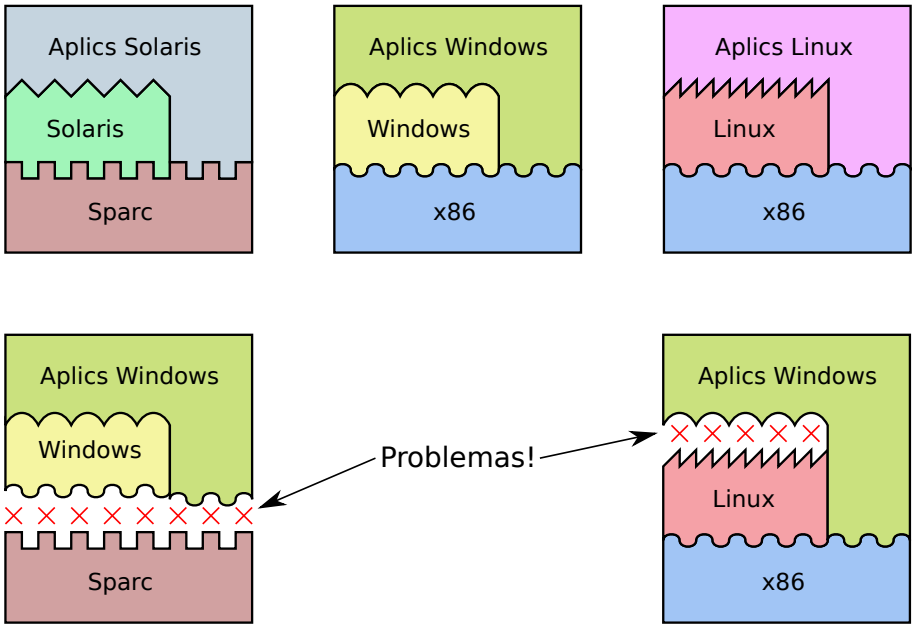


Figura 31.3: Problemas de compatibilidade entre interfaces [Smith and Nair, 2004].

A baixa flexibilidade na interação entre as interfaces dos componentes de um sistema computacional traz vários problemas [Smith and Nair, 2004]:

- *Baixa portabilidade*: a mobilidade de código e sua interoperabilidade são requisitos importantes dos sistemas atuais, que apresentam grande conectividade de rede e diversidade de plataformas. A rigidez das interfaces de sistema atuais dificulta sua construção, por acoplar excessivamente as aplicações aos sistemas operacionais e aos componentes do hardware.

- *Barreiras de inovação*: a presença de interfaces rígidas dificulta a construção de novas formas de interação entre as aplicações e os dispositivos de hardware (e

com os usuários, por consequência). Além disso, as interfaces apresentam uma grande inércia à evolução, por conta da necessidade de suporte às aplicações já existentes.

- *Otimizações intercomponentes*: aplicações, bibliotecas, sistemas operacionais e hardware são desenvolvidos por grupos distintos, geralmente com pouca interação entre eles. A presença de interfaces rígidas a respeitar entre os componentes leva cada grupo a trabalhar de forma isolada, o que diminui a possibilidade de otimizações que envolvam mais de um componente.

Essas dificuldades levaram à investigação de outras formas de relacionamento entre os componentes de um sistema computacional. Uma das abordagens mais promissoras nesse sentido é o uso da virtualização de interfaces, discutida a seguir.

31.4 Virtualização de interfaces

Conforme visto, as interfaces padronizadas entre os componentes do sistema de computação permitem o desenvolvimento independente dos mesmos, mas também são fonte de problemas de interoperabilidade, devido à sua pouca flexibilidade. Por isso, não é possível executar diretamente em um processador Intel/AMD uma aplicação compilada para um processador ARM: as instruções em linguagem de máquina da aplicação não serão compreendidas pelo processador Intel. Da mesma forma, não é possível executar diretamente em Linux uma aplicação escrita para um sistema Windows, pois as chamadas de sistema emitidas pelo programa Windows não serão compreendidas pelo sistema operacional Linux subjacente.

Todavia, é possível contornar esses problemas de compatibilidade através de uma *camada de virtualização* construída em software. Usando os serviços oferecidos por uma determinada interface de sistema, é possível construir uma camada de software que ofereça aos demais componentes uma outra interface. Essa camada de software permitirá o acoplamento entre interfaces distintas, de forma que um programa desenvolvido para a plataforma *A* possa executar sobre uma plataforma distinta *B*.

Usando os serviços oferecidos por uma determinada interface de sistema, a camada de virtualização constrói outra interface de mesmo nível, de acordo com as necessidades dos componentes de sistema que farão uso dela. A nova interface de sistema, vista através dessa camada de virtualização, é denominada *máquina virtual*. A camada de virtualização em si é denominada *hipervisor* (ou *monitor de máquina virtual*).

A Figura 31.4, extraída de [Smith and Nair, 2004], apresenta um exemplo de máquina virtual, onde um hipervisor permite executar um sistema operacional Windows e suas aplicações sobre uma plataforma de hardware Sparc, distinta daquela para a qual esse sistema operacional foi projetado (Intel x86).

Um ambiente de máquina virtual consiste de três partes básicas, que podem ser observadas na Figura 31.4:

- O sistema real, nativo ou hospedeiro (*host system*), que contém os recursos reais de hardware do sistema;

- a camada de virtualização, chamada *hipervisor* ou *monitor* (VMM – *Virtual Machine Monitor*), que constrói a interface virtual a partir da interface real;

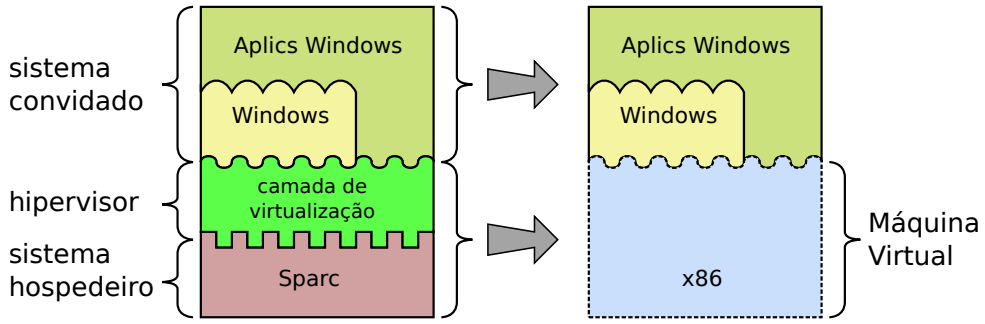


Figura 31.4: Uma máquina virtual [Smith and Nair, 2004].

- o sistema virtual, também chamado *sistema convidado* (*guest system*), que executa sobre a camada de virtualização.

É importante ressaltar a diferença entre os termos *virtualização* e *emulação*. A emulação é na verdade uma forma de virtualização: quando um hipervisor virtualiza integralmente uma interface de hardware ou de sistema operacional, é geralmente chamado de *emulador*. Por exemplo, a máquina virtual Java, que constrói um ambiente completo para a execução de *bytecodes* a partir de um processador real que não executa *bytecodes*, pode ser considerada um emulador.

A virtualização abre uma série de possibilidades interessantes para a composição de um sistema de computação, como por exemplo (Figura 31.5):

- *Emulação de hardware*: um sistema operacional convidado e suas aplicações, desenvolvidas para uma plataforma de hardware A, são executadas sobre uma plataforma de hardware distinta B.
- *Emulação de sistema operacional*: aplicações construídas para um sistema operacional X são executadas sobre outro sistema operacional Y.
- *Otimização dinâmica*: as instruções de máquina das aplicações são traduzidas durante a execução em outras instruções mais eficientes para a mesma plataforma.
- *Replicação de hardware*: são criadas várias instâncias virtuais de um mesmo hardware real, cada uma executando seu próprio sistema operacional convidado e suas respectivas aplicações.

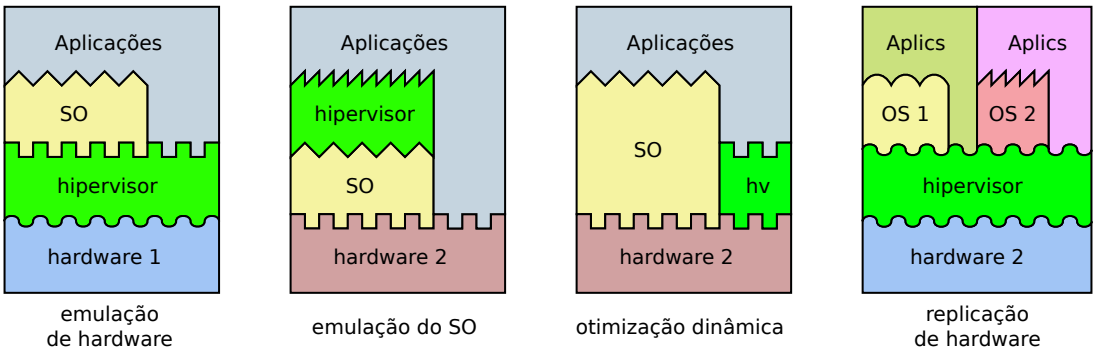


Figura 31.5: Possibilidades de virtualização [Smith and Nair, 2004].

31.5 Virtualização versus abstração

Embora a virtualização possa ser vista como um tipo de abstração, existe uma clara diferença entre os termos "abstração" e "virtualização", no contexto de sistemas operacionais [Smith and Nair, 2004]. Um dos principais objetivos do sistema operacional é oferecer uma visão de alto nível dos recursos de hardware, que seja mais simples de usar e menos dependente das tecnologias subjacentes. Essa visão abstrata dos recursos é construída de forma incremental, em níveis de abstração crescentes.

No subsistema de arquivos, por exemplo, cada nível de abstração trata de um problema: interação com o dispositivo físico, escalonamento de acessos ao dispositivo, gerência de *buffers* e *caches*, alocação de arquivos, diretórios, controle de acesso, etc. Essa estrutura em camadas é discutida em detalhes no capítulo 24.

Por outro lado, a virtualização consiste em criar novas interfaces a partir das interfaces existentes. Na virtualização, os detalhes de baixo nível da plataforma real não são necessariamente ocultos, como ocorre na abstração de recursos. A Figura 31.6 ilustra essa diferença: através da virtualização, um processador Sparc pode ser visto pelo sistema convidado como um processador Intel. Da mesma forma, um disco real no padrão SATA pode ser visto como vários discos menores independentes, com a mesma interface (SATA) ou outra interface (IDE).

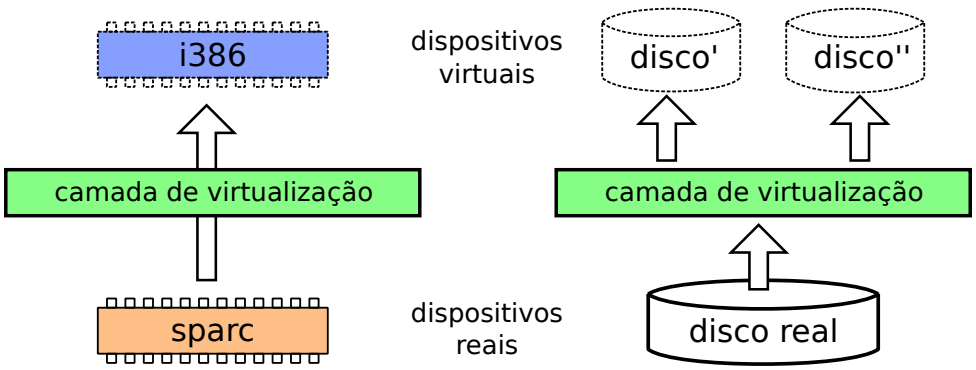


Figura 31.6: Virtualização de recursos do hardware.

A Figura 31.7 ilustra outro exemplo dessa diferença no contexto do armazenamento em disco. A abstração provê às aplicações o conceito de "arquivo", sobre o qual estas podem executar operações simples como `read` ou `write`, por exemplo. Já a virtualização fornece para a camada superior apenas um disco virtual, construído a

partir de um arquivo do sistema operacional real subjacente. Esse disco virtual terá de ser particionado e formatado para seu uso, da mesma forma que um disco real.

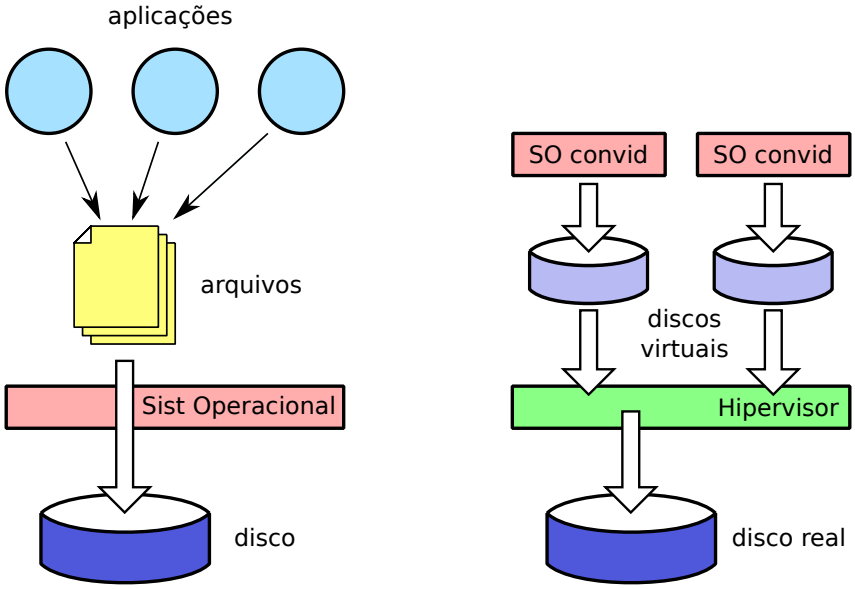


Figura 31.7: Abstração versus virtualização de um disco rígido.

Referências

R. Goldberg. Architecture of virtual machines. In *AFIPS National Computer Conference*, 1973.

R. Goldberg and P. Mager. Virtual machine technology: A bridge from large mainframes to networks of small computers. *IEEE Proceedings Compcon Fall 79*, pages 210–213, 1979.

G. Popek and R. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

J. Smith and R. Nair. *Virtual Machines: Architectures, Implementations and Applications*. Morgan Kaufmann, 2004.

Capítulo 32

Tipos de máquinas virtuais

O principal uso da virtualização de interfaces é a construção de *máquinas virtuais*. Uma máquina virtual é um ambiente de suporte à execução de software, construído usando uma ou mais técnicas de virtualização. Existem vários tipos de máquinas virtuais, que serão discutidos neste capítulo.

32.1 Critérios de classificação

Conforme as características do ambiente virtual proporcionado, as máquinas virtuais podem ser classificadas em três categorias, representadas na Figura 32.1:

Máquinas virtuais de sistema: são ambientes de máquinas virtuais construídos para emular uma plataforma de hardware completa, com processador e periféricos. Este tipo de máquina virtual suporta sistemas operacionais convidados com aplicações convidadas executando sobre eles. Como exemplos desta categoria de máquinas virtuais temos os ambientes *KVM*, *VMware* e *VirtualBox*.

Máquinas virtuais de sistema operacional: são construídas para suportar espaços de usuário distintos sobre um mesmo sistema operacional. Embora compartilhem o mesmo núcleo, cada ambiente virtual possui seus próprios recursos lógicos, como espaço de armazenamento, mecanismos de IPC e interfaces de rede distintas. Os sistemas *Docker*, *Solaris Containers* e *FreeBSD Jails* implementam este conceito.

Máquinas virtuais de processo: também chamadas de máquinas virtuais de aplicação ou de linguagem, são ambientes construídos para prover suporte de execução a apenas um processo ou aplicação convidada específica. A máquina virtual Java e o ambiente de depuração *Valgrind* são exemplos deste tipo de ambiente.

Os ambientes de máquinas virtuais também podem ser classificados de acordo com o nível de similaridade entre as interfaces de hardware do sistema convidado e do sistema real (ISA - *Instruction Set Architecture*, Seção 31.2):

Interfaces equivalentes: a interface virtual oferecida ao ambiente convidado reproduz a interface de hardware do sistema real, permitindo a execução de aplicações construídas para o sistema real. Como a maioria das instruções do sistema convidado pode ser executada diretamente pelo processador (com exceção das

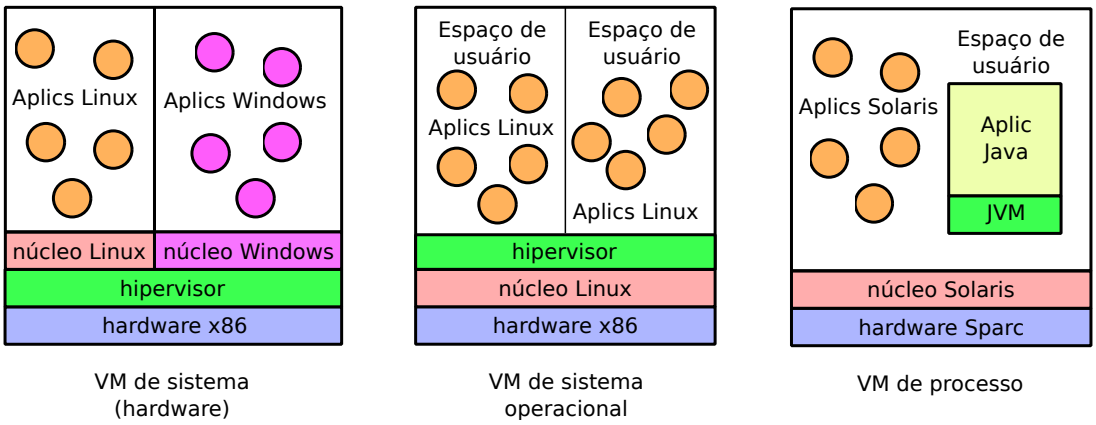


Figura 32.1: Máquinas virtuais de processo, de sistema operacional e de hardware.

instruções sensíveis), o desempenho obtido pelas aplicações convidadas pode ser próximo do desempenho de execução no sistema real. Ambientes como *VMware* são exemplos deste tipo de ambiente.

Interfaces distintas: a interface virtual não tem nenhuma relação com a interface de hardware do sistema real, ou seja, implementa um conjunto de instruções distinto, que deve ser totalmente traduzido pelo hipervisor. Conforme visto na Seção 33.1, a interpretação de instruções impõe um custo de execução significativo ao sistema convidado. A máquina virtual Java e o ambiente *QEmu* são exemplos dessa abordagem.

32.2 Máquinas virtuais de sistema

Uma máquina virtual de sistema (ou de hardware) provê uma interface de hardware completa para um ou mais sistemas operacionais convidados, com suas respectivas aplicações, que executam de forma isolada e independente. Cada sistema operacional convidado tem a ilusão de executar sozinho sobre uma plataforma de hardware exclusiva.

O hipervisor de sistema fornece aos sistemas operacionais convidados uma interface de sistema ISA virtual, que pode ser idêntica ao hardware real, ou distinta. Além disso, ele intercepta e virtualiza o acesso aos recursos, para que cada sistema operacional convidado tenha um conjunto próprio de recursos virtuais, construído a partir dos recursos físicos existentes na máquina real. Assim, cada máquina virtual terá sua própria interface de rede, disco, memória RAM, etc.

Em um ambiente virtual, os sistemas operacionais convidados são fortemente isolados uns dos outros, e normalmente só podem interagir através dos mecanismos de rede, como se estivessem em computadores separados. Todavia, alguns sistemas de máquinas virtuais permitem o compartilhamento controlado de certos recursos. Por exemplo, os sistemas *VMware Workstation* e *VirtualBox* permitem a definição de diretórios compartilhados no sistema de arquivos da máquina real, que podem ser acessados pelas máquinas virtuais.

As máquinas virtuais de sistema constituem a primeira abordagem usada para a construção de hipervisores, desenvolvida na década de 1960 e formalizada por Popek e Goldberg (conforme apresentado na Seção 33.1). Naquela época, a tendência de

desenvolvimento de sistemas computacionais buscava fornecer a cada usuário uma máquina virtual com seus recursos virtuais próprios, sobre a qual o usuário executava um sistema operacional monotarefa e suas aplicações. Assim, o compartilhamento de recursos não era responsabilidade do sistema operacional convidado, mas do hipervisor subjacente.

Ao longo dos anos 70, com o desenvolvimento de sistemas operacionais multitarefas eficientes e robustos como MULTICS e UNIX, as máquinas virtuais de sistema perderam gradativamente seu interesse. Somente no final dos anos 90, com o aumento do poder de processamento dos microprocessadores e o surgimento de novas possibilidades de aplicação, as máquinas virtuais de sistema foram "redescobertas".

Existem basicamente duas arquiteturas de hipervisores de sistema, apresentados na Figura 32.2:

Hipervisor nativo (ou *de tipo I*): executa diretamente sobre o hardware do computador real, sem um sistema operacional subjacente. A função do hipervisor é virtualizar os recursos do hardware (memória, discos, interfaces de rede, etc.) de forma que cada máquina virtual veja um conjunto de recursos próprio e independente. Assim, cada máquina virtual se comporta como um computador completo que pode executar o seu próprio sistema operacional. Esta é a forma mais antiga de virtualização, encontrada nos sistemas computacionais de grande porte dos anos 1960-70. Alguns exemplos de sistemas que empregam esta abordagem são o *IBM OS/370*, o *VMware ESX Server* e o ambiente *Xen*.

Hipervisor convidado (ou *de tipo II*): executa como um processo normal (ou um componente de núcleo) de um sistema operacional nativo subjacente, que gerencia o hardware. O hipervisor utiliza os recursos oferecidos por esse sistema operacional para oferecer recursos virtuais aos sistemas operacionais convidados. Exemplos de sistemas que adotam esta estrutura incluem o *VMware Workstation*, o *KVM* e o *VirtualBox*.

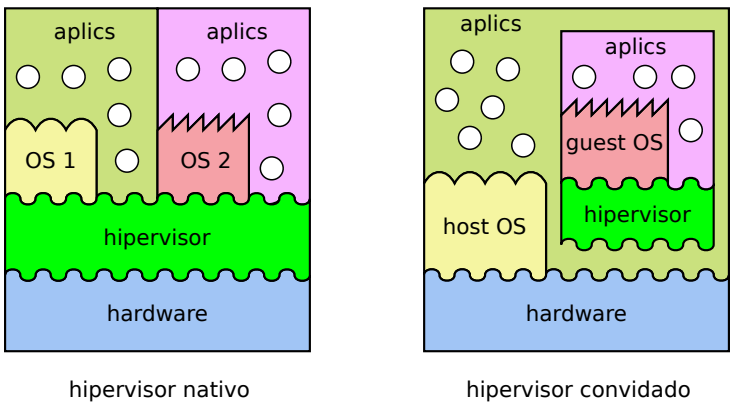


Figura 32.2: Arquiteturas de máquinas virtuais de sistema.

Pode-se afirmar que os hipervisores convidados são mais flexíveis que os hipervisores nativos, pois podem ser facilmente instalados/removidos em máquinas com sistemas operacionais previamente instalados, e podem ser facilmente lançados sob demanda. Por outro lado, hipervisores convidados têm desempenho pior que hipervisores nativos, pois têm de usar os recursos oferecidos pelo sistema operacional

subjacente, enquanto um hipervisor nativo pode acessar diretamente o hardware real. Técnicas para atenuar este problema são discutidas na Seção 33.5.

32.3 Máquinas virtuais de sistema operacional

Em muitas situações, a principal motivação para o uso de máquinas virtuais é o isolamento oferecido pelo ambiente virtual (Seção 33.1). Essa propriedade é importante na segurança de sistemas, por permitir isolar entre si sistemas independentes que executam sobre o mesmo hardware. Por exemplo, a estratégia de *consolidação de servidores* usa máquinas virtuais para abrigar os diversos servidores (de nomes, de arquivos, de e-mail, de Web) de uma empresa em uma mesma máquina física. Dessa forma, pode-se fazer um uso mais eficiente do hardware disponível, preservando o isolamento entre os serviços. Todavia, o custo da virtualização de uma plataforma de hardware ou sistema operacional sobre o desempenho do sistema final pode ser elevado. As principais fontes desse custo são a virtualização dos recursos (periféricos) da máquina real e a eventual necessidade de emular instruções do processador real.

Uma forma simples e eficiente de implementar o isolamento entre aplicações ou subsistemas em um sistema operacional consiste na virtualização do espaço de usuário (*userspace*). Nesta abordagem, denominada *máquinas virtuais de sistema operacional*, *servidores virtuais* ou *contêineres*, o espaço de usuário do sistema operacional é dividido em áreas isoladas denominadas *domínios* ou *contêineres*. A cada domínio é alocada uma parcela dos recursos do sistema operacional, como memória, tempo de processador e espaço em disco.

Para garantir o isolamento entre os domínios, alguns recursos do sistema real são virtualizados, como as interfaces de rede: cada domínio tem sua própria interface de rede virtual e, portanto, seu próprio endereço de rede. Além disso, na maioria das implementações cada domínio tem seu próprio espaço de nomes para os identificadores de usuários, processos e primitivas de comunicação. Assim, é possível encontrar um usuário `pedro` no domínio d_3 e outro usuário `pedro` no domínio d_7, sem relação entre eles nem conflitos. Essa noção de espaços de nomes distintos se estende aos demais recursos do sistema: identificadores de processos, semáforos, árvores de diretórios, etc.

Os processos presentes em um determinado domínio virtual podem interagir entre si, criar novos processos e usar os recursos presentes naquele domínio, respeitando as regras de controle de acesso associadas a esses recursos. Todavia, processos presentes em um domínio não podem ver ou interagir com processos que estiverem em outro domínio, não podem trocar de domínio, criar processos em outros domínios, nem consultar ou usar recursos de outros domínios. Dessa forma, para um determinado domínio, os demais domínios são máquinas distintas, acessíveis somente através de seus endereços de rede.

Para fins de gerência, normalmente é definido um domínio d_0, chamado de *domínio inicial*, *privilegiado* ou *de gerência*, cujos processos têm visibilidade e acesso aos processos e recursos dos demais domínios. Somente processos no domínio d_0 podem migrar para outros domínios; uma vez realizada uma migração, não há possibilidade de retornar ao domínio inicial.

Nesta forma de virtualização, o núcleo do sistema operacional é o mesmo para todos os domínios virtuais, e sua interface (conjunto de chamadas de sistema) é preservada (apenas algumas chamadas de sistema são adicionadas para a gestão dos domínios). A Figura 32.3 mostra a estrutura típica de um ambiente de máquinas

virtuais de sistema operacional. Nela, pode-se observar que um processo pode migrar de d_0 para d_1, mas que os processos em d_1 não podem migrar para outros domínios. A comunicação entre processos confinados em domínios distintos (d_2 e d_3) também é proibida.

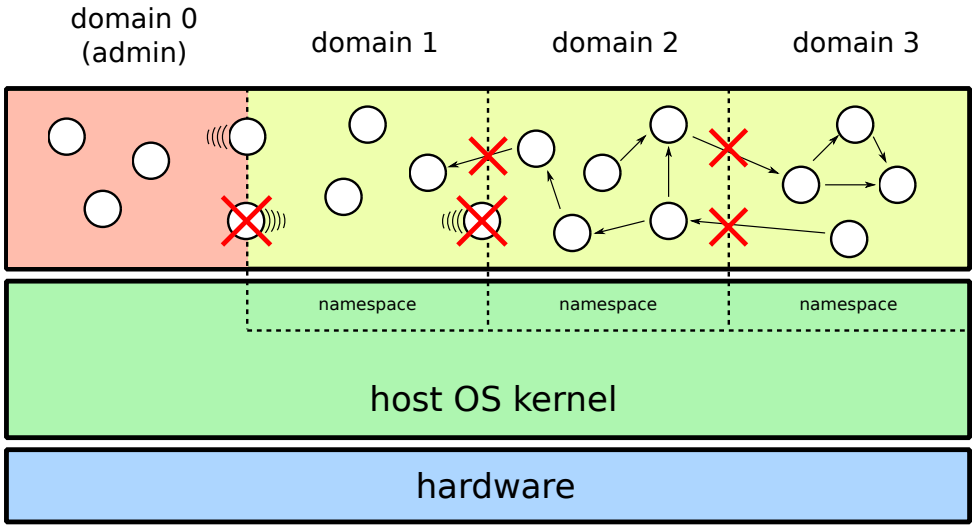


Figura 32.3: Máquinas virtuais de sistema operacional.

Há várias implementações disponíveis de mecanismos para a criação de contêineres. A técnica mais antiga é implementada pela chamada de sistema `chroot()`, disponível na maioria dos sistemas UNIX. Essa chamada de sistema atua sobre o acesso de um processo ao sistema de arquivos: o processo que a executa tem seu acesso ao sistema de arquivos restrito a uma subárvore da hierarquia de diretórios, ou seja, ele fica "confinado" a essa subárvore. Os filhos desse processo herdam essa mesma visão restrita do sistema de arquivos, que não pode ser revertida. Por exemplo, um processo que executa a chamada de sistema `chroot("/var/spool/postfix")` passa a ver somente a hierarquia de diretórios a partir do diretório `/var/spool/postfix`, que se torna o diretório raiz ("/") na visão daquele processo. A Figura 32.4 ilustra essa operação.

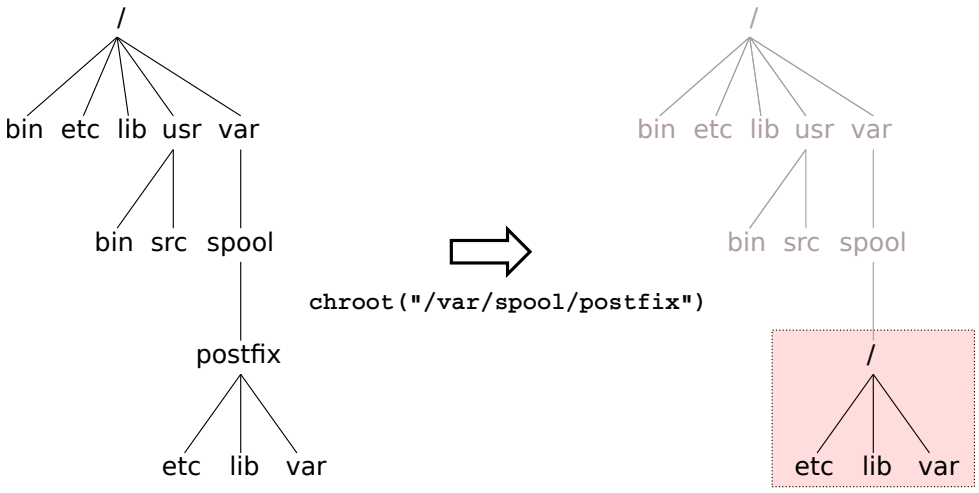


Figura 32.4: Funcionamento da chamada de sistema `chroot`.

A chamada de sistema `chroot` é muito utilizada para isolar processos que oferecem serviços à rede, como servidores DNS e de e-mail. Se um processo servidor

tiver alguma vulnerabilidade e for subvertido por um atacante, este só terá acesso ao conjunto de diretórios visíveis a esse processo, mantendo fora de alcance o restante da árvore de diretórios do sistema.

O sistema operacional *FreeBSD* oferece uma implementação de domínios virtuais mais elaborada, conhecida como *Jails* [McKusick and Neville-Neil, 2005] (aqui traduzidas como *celas*). Um processo que executa a chamada de sistema `jail()` cria uma nova cela e é colocado dentro dela, de onde não pode mais sair, nem seus filhos. Os processos dentro de uma cela estão submetidos a várias restrições:

- o processo só vê processos e recursos associados àquela cela; os demais processos, arquivos e outros recursos do sistema não associados à cela não são visíveis;
- somente são permitidas interações (comunicação e coordenação) entre processos dentro da mesma cela;
- de forma similar à chamada `chroot`, cada cela recebe uma árvore de diretórios própria; operações de montagem/desmontagem de sistemas de arquivos são proibidas;
- cada cela tem um endereço de rede associado, que é o único utilizável pelos processos da cela; a configuração de rede (endereço, parâmetros de interface, tabela de roteamento) não pode ser modificada;
- não podem ser feitas alterações no núcleo do sistema, como reconfigurações ou inclusões/exclusões de módulos.

Essas restrições são impostas a todos os processos dentro de uma cela, mesmo aqueles pertencentes ao administrador (usuário *root*). Assim, uma cela constitui uma unidade de isolamento bastante robusta, que pode ser usada para confinar serviços de rede e aplicações ou usuários considerados "perigosos".

Existem implementações de estruturas similares às celas do *FreeBSD* em outros sistemas operacionais. Por exemplo, o sistema operacional *Solaris* implementa o conceito de *zonas* [Price and Tucker, 2004], que oferecem uma capacidade de isolamento similar à celas, além de prover um mecanismo de controle da distribuição dos recursos entre as diferentes zonas existentes. Outras implementações podem ser encontradas para o sistema Linux, como os ambientes *Virtuozzo*, *Vservers*, *LXC* e *Docker*.

32.4 Máquinas virtuais de processo

Uma máquina virtual de processo, de aplicação ou de linguagem (*Process Virtual Machine*) suporta a execução de um processo ou aplicação individual. Ela é criada sob demanda, no momento do lançamento da aplicação convidada, e destruída quando a aplicação finaliza sua execução. O conjunto *hipervisor* + *aplicação* é normalmente visto como um único processo dentro do sistema operacional hospedeiro (ou um pequeno conjunto de processos), submetido às mesmas condições e restrições que os demais processos nativos.

Os hipervisores que implementam máquinas virtuais de processo normalmente permitem a interação entre a aplicação convidada e as demais aplicações do sistema, através dos mecanismos usuais de comunicação e coordenação entre processos, como

mensagens, *pipes* e semáforos. Além disso, também permitem o acesso normal ao sistema de arquivos e outros recursos locais do sistema. Estas características violam a propriedade de *isolamento* descrita na Seção 33.1, mas são necessárias para que a aplicação convidada se comporte como uma aplicação normal aos olhos do usuário.

Ao criar a máquina virtual para uma aplicação, o hipervisor pode implementar a mesma interface de hardware (ISA, Seção 31.2) da máquina real subjacente, ou implementar uma interface distinta. Quando a interface da máquina real é preservada, boa parte das instruções do processo convidado podem ser executadas diretamente sem perda de desempenho, com exceção das instruções sensíveis, que devem ser interpretadas pelo hipervisor.

Os exemplos mais comuns de máquinas virtuais de aplicação que preservam a interface ISA real são os *sistemas operacionais multitarefas*, os *tradutores dinâmicos* e alguns *depuradores de memória*:

Sistemas operacionais multitarefas: os sistemas operacionais que suportam vários processos simultâneos, estudados no Capítulo 4, também podem ser vistos como ambientes de máquinas virtuais. Em um sistema multitarefas, cada processo recebe um *processador virtual* (simulado através das fatias de tempo do processador real e das trocas de contexto), uma *memória virtual* (através do espaço de endereços mapeado para aquele processo) e *recursos físicos* (acessíveis através de chamadas de sistema). Este ambiente de virtualização é tão antigo e tão presente em nosso cotidiano que costumamos ignorá-lo como tal. No entanto, ele simplifica muito a tarefa dos programadores, que não precisam se preocupar com a gestão do isolamento e do compartilhamento de recursos entre os processos.

Tradutores dinâmicos: um tradutor dinâmico consiste em um hipervisor que analisa e otimiza um código executável, para tornar sua execução mais rápida e eficiente. A otimização não muda o conjunto de instruções da máquina real usado pelo código, apenas reorganiza as instruções de forma a acelerar sua execução. Por ser dinâmica, a otimização do código é feita durante a carga do processo na memória ou durante a execução de suas instruções, sendo transparente ao processo e ao usuário. O artigo [Duesterwald, 2005] apresenta uma descrição detalhada desse tipo de abordagem.

Depuradores de memória: alguns sistemas de depuração de erros de acesso à memória, como o sistema *Valgrind* [Seward and Nethercote, 2005], executam o processo sob depuração em uma máquina virtual. Todas as instruções do programa que manipulam acessos à memória são executadas de forma controlada, a fim de encontrar possíveis erros. Ao depurar um programa, o sistema *Valgrind* inicialmente traduz seu código binário em um conjunto de instruções interno, manipula esse código para inserir operações de verificação de acessos à memória e traduz o código modificado de volta ao conjunto de instruções da máquina real, para em seguida executá-lo e verificar os acessos à memória realizados.

Contudo, as máquinas virtuais de processo mais populares atualmente são aquelas em que a interface binária de aplicação (ABI, Seção 31.2) requerida pela aplicação é diferente daquela oferecida pela máquina real. Como a ABI é composta pelas chamadas do sistema operacional e as instruções de máquina disponíveis à aplicação (*user ISA*), as

diferenças podem ocorrer em ambos esses componentes. Nos dois casos, o hipervisor terá de fazer traduções dinâmicas (durante a execução) das ações requeridas pela aplicação em suas equivalentes na máquina real. Como visto, um hipervisor com essa função é denominado *tradutor dinâmico*.

Caso as diferenças de interface entre a aplicação e a máquina real se limitem às chamadas do sistema operacional, o hipervisor precisa mapear somente as chamadas de sistema e de bibliotecas usadas pela aplicação sobre as chamadas equivalentes oferecidas pelo sistema operacional da máquina real. Essa é a abordagem usada, por exemplo, pelo ambiente *Wine*, que permite executar aplicações Windows em plataformas Unix. As chamadas de sistema Windows emitidas pela aplicação em execução são interceptadas e convertidas em chamadas Unix, de forma dinâmica e transparente (Figura 32.5).

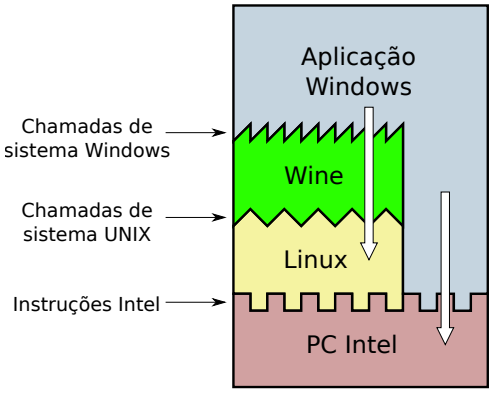


Figura 32.5: Funcionamento do emulador Wine.

Entretanto, muitas vezes a interface ISA utilizada pela aplicação não corresponde a nenhum hardware existente, mas a uma máquina abstrata. Um exemplo típico dessa situação ocorre na linguagem Java: um programa escrito em Java, ao ser compilado, gera um código binário específico para uma máquina abstrata denominada *máquina virtual Java* (*JVM – Java Virtual Machine*). A linguagem de máquina executada pela máquina virtual Java é denominada *bytecode* Java, e não corresponde a instruções de um processador real. A máquina virtual deve então interpretar todas as operações do *bytecode*, utilizando as instruções da máquina real subjacente para executá-las. Além de Java, várias linguagens empregam a mesma abordagem (embora com *bytecodes* distintos), como Lua, Python e C#.

Em termos de desempenho, um programa compilado para um processador abstrato executa mais lentamente que seu equivalente compilado para um processador real, devido ao custo de interpretação do *bytecode*. Todavia, essa abordagem oferece melhor desempenho que linguagens puramente interpretadas. Além disso, técnicas de otimização como a compilação *Just-in-Time* (JIT), na qual blocos de instruções frequentes são traduzidos e mantidos em cache pelo hipervisor, permitem obter ganhos de desempenho significativos.

Referências

E. Duesterwald. Design and engineering of a dynamic binary optimizer. *Proceedings of the IEEE*, 93(2):436–448, Feb 2005.

M. McKusick and G. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2005.

D. Price and A. Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *18th USENIX conference on System administration*, pages 241–254, 2004.

J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference*, 2005.

Capítulo 33

Construção de máquinas virtuais

A construção de máquinas virtuais é bem mais complexa que possa parecer à primeira vista. Caso os conjuntos de instruções (ISA) do sistema real e do sistema virtual sejam diferentes, é necessário usar as instruções da máquina real para simular as instruções da máquina virtual. Além disso, é necessário mapear os recursos de hardware virtuais (periféricos oferecidos ao sistema convidado) sobre os recursos existentes na máquina real (os periféricos reais). Por fim, pode ser necessário mapear as chamadas de sistema emitidas pelas aplicações do sistema convidado em chamadas equivalentes no sistema real, quando os sistemas operacionais virtual e real forem distintos.

Este capítulo aborda inicialmente o conceito formal de virtualização, para em seguida discutir as principais técnicas usadas na construção de máquinas virtuais.

33.1 Definição formal

Em 1974, os pesquisadores americanos Gerald Popek (UCLA) e Robert Goldberg (Harvard) definiram uma máquina virtual da seguinte forma [Popek and Goldberg, 1974]:

> *Uma máquina virtual é vista como uma duplicata eficiente e isolada de uma máquina real. Essa abstração é construída por um "monitor de máquina virtual" (VMM - Virtual Machine Monitor).*

Para funcionar de forma correta e eficiente, o monitor ou hipervisor deve atender a alguns requisitos básicos, também definidos pos Popek e Goldberg:

Equivalência: um hipervisor de prover um ambiente de execução quase idêntico ao da máquina real original. Todo programa executando em uma máquina virtual deve se comportar da mesma forma que o faria em uma máquina real; exceções podem resultar somente de diferenças nos recursos disponíveis (memória, disco, etc.), dependências de temporização e a existência dos dispositivos de entrada/saída necessários à aplicação.

Controle de recursos: o hipervisor deve possuir o controle completo dos recursos da máquina real: nenhum programa executando na máquina virtual deve possuir acesso a recursos que não tenham sido explicitamente alocados a ele pelo hipervisor, que deve intermediar todos os acessos. Além disso, a qualquer instante o hipervisor pode retirar recursos previamente alocados à máquina virtual.

Eficiência: grande parte das instruções do processador virtual (o processador provido pelo hipervisor) deve ser executada diretamente pelo processador da máquina real, sem intervenção do hipervisor. As instruções da máquina virtual que não puderem ser executadas pelo processador real devem ser interpretadas pelo hipervisor e traduzidas em ações equivalentes no processador real. Instruções simples, que não afetem outras máquinas virtuais ou aplicações, podem ser executadas diretamente no processador real.

Além dessas três propriedades básicas, as propriedades derivadas a seguir são frequentemente associadas a hipervisores [Rosenblum, 2004]:

Isolamento: aplicações dentro de uma máquina virtual não podem interagir diretamente (a) com outras máquinas virtuais, (b) com o hipervisor, ou (c) com o sistema real hospedeiro. Todas as interações entre entidades dentro de uma máquina virtual e o mundo exterior devem ser mediadas pelo hipervisor.

Introspecção: o hipervisor tem acesso e controle sobre todas as informações do estado interno da máquina virtual, como registradores do processador, conteúdo de memória, eventos etc.

Recursividade: alguns hipervisores exibem também esta propriedade: deve ser possível executar um hipervisor dentro de uma máquina virtual, produzindo um novo nível de máquinas virtuais. Neste caso, a máquina real é normalmente denominada *máquina de nível 0*.

Essas propriedades básicas caracterizam um hipervisor ideal, que nem sempre pode ser construído sobre as plataformas de hardware existentes. A possibilidade de construção de um hipervisor em uma determinada plataforma é definida através do seguinte teorema, definido por Popek e Goldberg [Popek and Goldberg, 1974]:

> *Para qualquer computador convencional de terceira geração, um hipervisor pode ser construído se o conjunto de instruções sensíveis daquele computador for um subconjunto de seu conjunto de instruções privilegiadas.*

Para compreender melhor as implicações desse teorema, é necessário definir claramente os seguintes conceitos:

- *Computador convencional de terceira geração*: qualquer sistema de computação convencional seguindo a arquitetura de Von Neumann, que suporte memória virtual e dois modos de operação do processador: modo usuário e modo privilegiado.
- *Instruções sensíveis*: são aquelas que podem consultar ou alterar o status do processador, ou seja, os registradores que armazenam o status atual da execução na máquina real;
- *Instruções privilegiadas*: são acessíveis somente por meio de códigos executando em nível privilegiado (código de núcleo). Caso um código não privilegiado tente executar uma instrução privilegiada, uma exceção (interrupção) deve ser gerada, ativando uma rotina de tratamento previamente especificada pelo núcleo do sistema real.

De acordo com esse teorema, toda instrução sensível deve ser também privilegiada. Assim, quando uma instrução sensível for executada por um programa não privilegiado (um núcleo convidado ou uma aplicação convidada), provocará a ocorrência de uma interrupção. Essa interrupção pode ser usada para ativar uma *rotina de interpretação* dentro do hipervisor, que irá simular o efeito da instrução sensível (ou seja, interpretá-la), de acordo com o contexto onde sua execução foi solicitada (máquina virtual ou hipervisor). Obviamente, quanto maior o número de instruções sensíveis, maior o volume de interpretação de código a realizar, e menor o desempenho da máquina virtual.

No caso de processadores que não atendam as restrições de Popek/Goldberg, podem existir instruções sensíveis que executem sem gerar interrupções, o que impede o hipervisor de interceptá-las e interpretá-las. Uma solução possível para esse problema é a *tradução dinâmica* das instruções sensíveis presentes nos programas de usuário: ao carregar um programa na memória, o hipervisor analisa seu código e substitui essas instruções sensíveis por chamadas a rotinas que as interpretam dentro do hipervisor. Isso implica em um tempo maior para o lançamento de programas, mas torna possível a virtualização. Outra técnica possível para resolver o problema é a *paravirtualização*, que se baseia em reescrever parte do sistema convidado para não usar essas instruções sensíveis. Ambas as técnicas são discutidas neste capítulo.

33.2 Suporte de hardware

Na época em que Popek e Goldberg definiram seu principal teorema, o hardware dos mainframes IBM suportava parcialmente as condições impostas pelo mesmo. Esses sistemas dispunham de uma funcionalidade chamada *execução direta*, que permitia a uma máquina virtual acessar nativamente o hardware para a execução de instruções. Esse mecanismo permitia que aqueles sistemas obtivessem, com a utilização de máquinas virtuais, desempenho similar ao de sistemas convencionais equivalentes [Goldberg, 1973; Popek and Goldberg, 1974; Goldberg and Mager, 1979].

O suporte de hardware para a construção de hipervisores eficientes está presente em sistemas de grande porte, como os mainframes, mas permaneceu deficiente nos microprocessadores de mercado por muito tempo. Por exemplo, a família de processadores *Intel Pentium IV* (e anteriores) possuía 17 instruções sensíveis que podiam ser executadas em modo usuário sem gerar exceções, violando o teorema de Goldberg (Seção 33.1) e dificultando a criação de máquinas virtuais [Robin and Irvine, 2000]. Alguns exemplos dessas instruções "problemáticas" são:

- `SGDT/SLDT`: permitem ler o registrador que indica a posição e tamanho das tabelas de segmentos global/local do processo ativo.
- `SMSW`: permite ler o registrador de controle 0, que contém informações de status interno do processador.
- `PUSHF/POPF`: empilha/desempilha o valor do registrador `EFLAGS`, que também contém informações de status interno do processador.

Para controlar o acesso aos recursos do sistema e às instruções privilegiadas, os processadores usam a noção de "anéis de proteção" herdada do sistema MULTICS [Corbató and Vyssotsky, 1965]. Os anéis definem níveis de privilégio: um código

executando no nível 0 (anel central) tem acesso completo ao hardware, enquanto um código executando em um nível $i > 0$ (anéis externos) tem menos privilégio. Quanto mais externo o anel onde um código executa, menor o seu nível de privilégio. Os processadores Intel/AMD atuais suportam 4 anéis ou níveis de proteção, mas a quase totalidade dos sistemas operacionais de mercado somente usa os dois anéis extremos: o anel 0 para o núcleo do sistema e o anel 3 para as aplicações dos usuários.

As técnicas iniciais de virtualização para as plataformas Intel/AMD se baseavam na redução de privilégios do sistema operacional convidado: o hipervisor e o sistema operacional hospedeiro executam no nível 0, o sistema operacional convidado executa no nível 1 ou 2 e as aplicações do sistema convidado executam no nível 3. Essas formas de estruturação de sistema são denominadas respectivamente "modelo 0/1/3" e modelo "0/2/3" (Figura 33.1). Todavia, para que a estratégia de redução de privilégio pudessem funcionar, algumas instruções do sistema operacional convidado precisavam ser reescritas dinamicamente no momento da carga do sistema convidado na memória, pois ele foi construído para executar no nível 0.

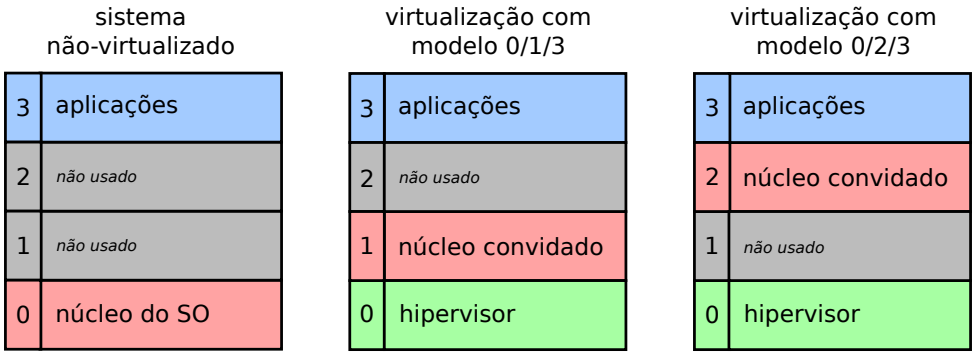


Figura 33.1: Uso dos níveis de proteção em processadores Intel/AMD.

Por volta de 2005, os principais fabricantes de microprocessadores (*Intel* e *AMD*) incorporaram um suporte básico à virtualização em seus processadores, através das tecnologias IVT (*Intel Virtualization Technology*) e AMD-V (*AMD Virtualization*), que são conceitualmente equivalentes [Uhlig et al., 2005]. A ideia central de ambas as tecnologias consiste em definir dois modos possíveis de operação do processador: os modos *root* e *non-root*. O modo *root* equivale ao funcionamento de um processador convencional, e se destina à execução de um hipervisor. Por outro lado, o modo *non-root* se destina à execução de máquinas virtuais. Ambos os modos suportam os quatro níveis de privilégio, o que permite executar os sistemas convidados sem a necessidade de reescrita dinâmica de seu código.

São também definidos dois procedimentos de transição entre modos: *VM entry* (transição *root* → *non-root*) e *VM exit* (transição *non-root* → *root*). Quando operando dentro de uma máquina virtual (ou seja, em modo *non-root*), as instruções sensíveis e as interrupções podem provocar a transição *VM exit*, devolvendo o processador ao hipervisor em modo *root*. As instruções e interrupções que provocam a transição *VM exit* são configuráveis pelo próprio hipervisor.

Para gerenciar o estado do processador (conteúdo dos registradores), é definida uma *Estrutura de Controle de Máquina Virtual* (VMCS - *Virtual-Machine Control Structure*). Essa estrutura de dados contém duas áreas: uma para os sistemas convidados e outra para o hipervisor. Na transição *VM entry*, o estado do processador é lido a partir da área de sistemas convidados da VMCS. Já uma transição *VM exit* faz com que o estado do

processador seja salvo na área de sistemas convidados e o estado anterior do hipervisor, previamente salvo na VMCS, seja restaurado. A Figura 33.2 traz uma visão geral da arquitetura Intel IVT.

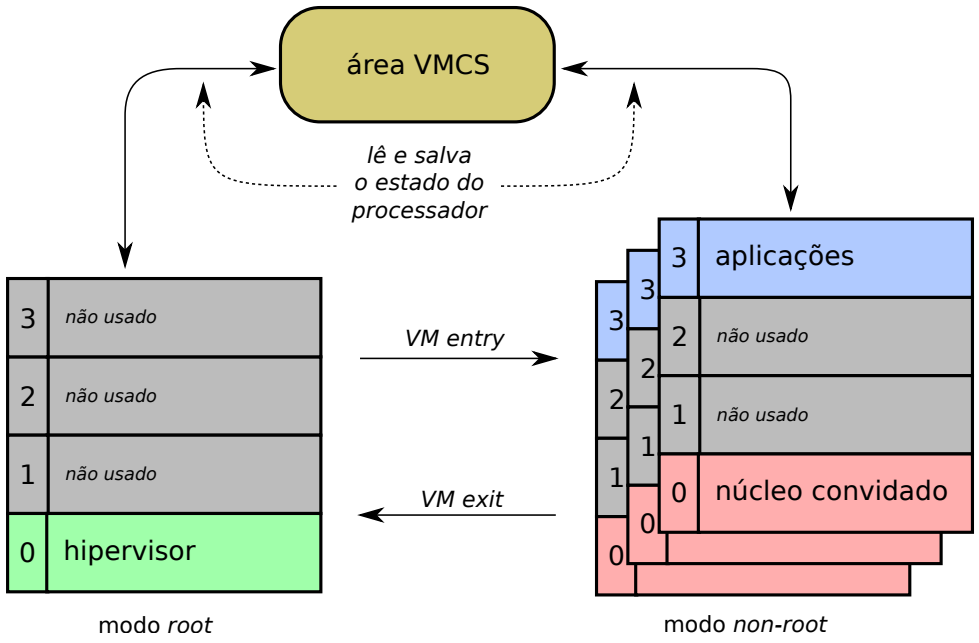


Figura 33.2: Visão geral da arquitetura *Intel IVT*.

Além da Intel e AMD, outros fabricantes de hardware se preocuparam com o suporte à virtualização. Em 2005, a *Sun Microsystems* incorporou suporte nativo à virtualização em seus processadores *UltraSPARC* [Yen, 2007]. Em 2007, a IBM propôs uma especificação de interface de hardware denominada *IBM Power ISA 2.04* [IBM], que respeita os requisitos necessários à virtualização do processador e da gestão de memória.

Conforme apresentado, a virtualização do processador pode obtida por reescrita dinâmica do código executável ou através do suporte nativo em hardware, usando tecnologias como IVT e AMD-V. Por outro lado, a virtualização da memória envolve outros desafios, que exigem modificações significativas dos mecanismos de gestão de memória. Por exemplo, é importante prever o compartilhamento de páginas de código entre máquinas virtuais, para reduzir a quantidade total de memória física necessária a cada máquina virtual. Outros desafios similares surgem na virtualização dos dispositivos de armazenamento e de entrada/saída, alguns deles sendo analisados em [Rosenblum and Garfinkel, 2005].

33.3 Níveis de virtualização

A virtualização implica na reescrita de interfaces de sistema, para permitir a interação entre componentes de sistema construídos para plataformas distintas. Como existem várias interfaces entre os componentes, também há várias possibilidades de aplicação da virtualização em um sistema. De acordo com as interfaces em que são aplicadas, os níveis mais usuais de virtualização são [Rosenblum, 2004; Nanda and Chiueh, 2005]:

Virtualização do hardware: toda a interface ISA, que permite o acesso ao hardware, é virtualizada. Isto inclui o conjunto de instruções do processador e as interfaces de acesso aos dispositivos de entrada/saída. A virtualização de hardware (ou virtualização completa) permite executar um sistema operacional e/ou aplicações em uma plataforma totalmente diversa daquela para a qual estes foram desenvolvidos. Esta é a forma de virtualização mais poderosa e flexível, mas também a de menor desempenho, uma vez que o hipervisor tem de traduzir todas as instruções geradas no sistema convidado em instruções do processador real. A máquina virtual Java (JVM, Seção 34.2.6) é um bom exemplo de virtualização do hardware. Máquinas virtuais implementando esta forma de virtualização são geralmente denominados *emuladores de hardware*.

Virtualização da interface de sistema: virtualiza-se a *System ISA*, que corresponde ao conjunto de instruções sensíveis do processador. Esta forma de virtualização é bem mais eficiente que a anterior, pois o hipervisor emula somente as instruções sensíveis do processador virtual, executadas em modo privilegiado pelo sistema operacional convidado. As instruções não sensíveis podem ser executadas diretamente pelo processador real, sem perda de desempenho. Todavia, apenas sistemas convidados desenvolvidos para o mesmo processador podem ser executados usando esta abordagem. Esta é a abordagem clássica de virtualização, presente nos sistemas de grande porte (*mainframes*) e usada nos ambientes de máquinas virtuais VMware, Xen e KVM.

Virtualização de dispositivos de entrada/saída: virtualizam-se os dispositivos físicos que permitem ao sistema interagir com o mundo exterior. Esta técnica implica na construção de dispositivos físicos virtuais, como discos, interfaces de rede e terminais de interação com o usuário, usando os dispositivos físicos subjacentes. A maioria dos ambientes de máquinas virtuais usa a virtualização de dispositivos para oferecer discos rígidos e interfaces de rede virtuais aos sistemas convidados.

Virtualização do sistema operacional: virtualiza-se o conjunto de recursos lógicos oferecidos pelo sistema operacional, como árvores de diretórios, descritores de arquivos, semáforos, canais de IPC e IDS de processos, usuários e grupos. Nesta abordagem, cada máquina virtual pode ser vista como uma instância distinta do mesmo sistema operacional subjacente. Esta é a abordagem comumente conhecida como *servidores virtuais* ou *contêneres*, da qual são bons exemplos os ambientes *FreeBSD Jails*, *Linux VServers* e *Solaris Zones*.

Virtualização de chamadas de sistema: permite oferecer o conjunto de chamadas de sistema de uma sistema operacional *A* usando as chamadas de sistema de um sistema operacional *B*, permitindo assim a execução de aplicações desenvolvidas para um sistema operacional sobre outro sistema. Todavia, como as chamadas de sistema são normalmente invocadas através de funções de bibliotecas, a virtualização de chamadas de sistema pode ser vista como um caso especial de virtualização de bibliotecas.

Virtualização de chamadas de biblioteca: tem objetivos similares ao da virtualização de chamadas de sistema, permitindo executar aplicações em diferentes sistemas operacionais e/ou com bibliotecas diversas daquelas para as quais foram

construídas. O sistema *Wine*, que permite executar aplicações Windows sobre sistemas UNIX, usa essencialmente esta abordagem.

Na prática, esses vários níveis de virtualização podem ser usados para a resolução de problemas específicos em diversas áreas de um sistema de computação. Por exemplo, vários sistemas operacionais oferecem facilidades de virtualização de dispositivos físicos, como por exemplo: interfaces de rede virtuais que permitem associar mais de um endereço de rede ao computador, discos rígidos virtuais criados em áreas livres da memória RAM (os chamados *RAM disks*); árvores de diretórios virtuais criadas para confinar processos críticos para a segurança do sistema (através da chamada de sistema `chroot`), emulação de operações 3D em uma placa gráfica que não as suporta, etc.

33.4 Técnicas de virtualização

A construção de hipervisores implica na definição de algumas estratégias para a virtualização. As estratégias mais utilizadas atualmente são a emulação completa do hardware, a virtualização da interface de sistema, a tradução dinâmica de código e a paravirtualização. Além disso, algumas técnicas complementares são usadas para melhorar o desempenho dos sistemas de máquinas virtuais. Essas técnicas são discutidas nesta seção.

33.4.1 Emulação completa

Nesta abordagem, toda a interface do hardware é virtualizada, incluindo todas as instruções do processador, a memória e os dispositivos periféricos. Isso permite oferecer ao sistema operacional convidado uma interface de hardware distinta daquela fornecida pela máquina real subjacente, caso seja necessário. O custo de virtualização pode ser muito elevado, pois cada instrução executada pelo sistema convidado tem de ser analisada e traduzida em uma ou mais instruções equivalentes no computador real. No entanto, esta abordagem permite executar sistemas operacionais em outras plataformas, distintas daquela para a qual foram projetados, sem modificação.

Exemplos típicos de emulação completa abordagem são os sistemas de máquinas virtuais *QEMU*, que oferece um processador x86 ao sistema convidado, o *MS VirtualPC for MAC*, que permite executar o sistema Windows sobre uma plataforma de hardware *PowerPC*, e o sistema *Hercules*, que emula um computador *IBM System/390* sobre um PC convencional de plataforma Intel.

Um caso especial de emulação completa consiste nos **hipervisores embutidos no hardware** (*codesigned hypervisors*). Um hipervisor embutido é visto como parte integrante do hardware e implementa a interface de sistema (ISA) vista pelos sistemas operacionais e aplicações daquela plataforma. Entretanto, o conjunto de instruções do processador real somente está acessível ao hipervisor, que reside em uma área de memória separada da memória principal e usa técnicas de tradução dinâmica (vide Seção 33.4.3) para tratar as instruções executadas pelos sistemas convidados.

Um exemplo típico desse tipo de sistema é o processador *Transmeta Crusoe/Efficeon*, que aceita instruções no padrão *Intel 32 bits* e internamente as converte em um conjunto de instruções VLIW (*Very Large Instruction Word*). Como o hipervisor desse processador pode ser reprogramado para criar novas instruções ou modificar as

instruções existentes, ele acabou sendo denominado *Code Morphing Software* (Figura 33.3).

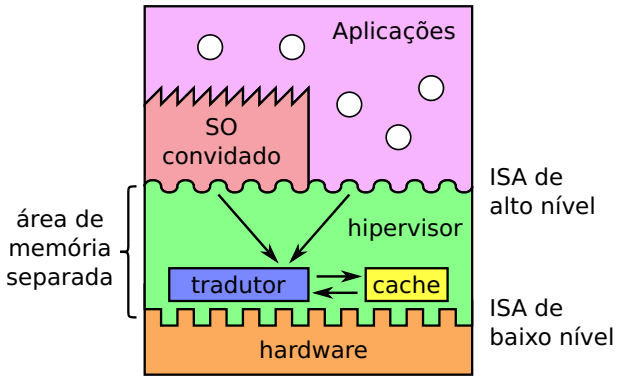


Figura 33.3: Hipervisor embutido no hardware.

33.4.2 Virtualização da interface de sistema

Nesta abordagem, a interface ISA de usuário é mantida, apenas as instruções privilegiadas e os dispositivos (discos, interfaces de rede, etc.) são virtualizados. Dessa forma, o sistema operacional convidado e as aplicações convidadas veem o processador real. Como a quantidade de instruções a virtualizar é reduzida, o desempenho do sistema convidado pode ficar próximo daquele obtido se ele estivesse executando diretamente sobre o hardware real.

Cabe lembrar que a virtualização da interface de sistema só pode ser aplicada diretamente caso o hardware subjacente atenda os requisitos de Goldberg e Popek (cf. Seção 33.1). No caso de processadores que não atendam esses requisitos, podem existir instruções sensíveis que executem sem gerar interrupções, impedindo o hipervisor de interceptá-las. Nesse caso, será necessário o emprego de técnicas complementares, como a *tradução dinâmica* das instruções sensíveis. Obviamente, quanto maior o número de instruções sensíveis, maior o volume de interpretação de código a realizar, e menor o desempenho da máquina virtual. Os processadores mais recentes das famílias Intel e AMD discutidos na Seção 33.2 atendem os requisitos de Goldberg/Popek, e por isso suportam esta técnica de virtualização.

Exemplos de sistemas que implementam esta técnica incluem os ambientes *VMware Workstation*, *VirtualBox*, *MS VirtualPC* e *KVM*.

33.4.3 Tradução dinâmica

Uma técnica frequentemente utilizada na construção de máquinas virtuais é a *tradução dinâmica* (*dynamic translation*) ou recompilação dinâmica (*dynamic recompilation*) de partes do código binário do sistema convidado e suas aplicações. Nesta técnica, o hipervisor analisa, reorganiza e traduz as sequências de instruções emitidas pelo sistema convidado em novas sequências de instruções, à medida em que o código é carregado na memória, ou mesmo durante a execução do sistema convidado.

A tradução binária dinâmica pode ter vários objetivos: (a) adaptar as instruções geradas pelo sistema convidado à interface ISA do sistema real, caso não sejam idênticas; (b) detectar e tratar instruções sensíveis não privilegiadas (que não geram interrupções

ao serem invocadas pelo sistema convidado); ou (c) analisar, reorganizar e otimizar as sequências de instruções geradas pelo sistema convidado, de forma a melhorar o desempenho de sua execução. Neste último caso, os blocos de instruções muito frequentes podem ter suas traduções mantidas em cache, para melhorar ainda mais o desempenho.

A tradução dinâmica é usada em vários tipos de hipervisores. Uma aplicação típica é a construção da máquina virtual Java, onde recebe o nome de JIT – *Just-in-Time Bytecode Compiler*. Outro uso corrente é a construção de hipervisores para plataformas sem suporte adequado à virtualização, como os processadores Intel/AMD 32 bits. Neste caso, o código convidado a ser executado é analisado em busca de instruções sensíveis, que são substituídas por chamadas a rotinas apropriadas dentro do supervisor.

No contexto de virtualização, a tradução dinâmica é composta basicamente dos seguintes passos [Ung and Cifuentes, 2006]:

1. *Desmontagem* (*disassembling*): o fluxo de bytes do código convidado em execução é decomposto em blocos de instruções. Cada bloco é normalmente composto de uma sequência de instruções de tamanho variável, terminando com uma instrução de controle de fluxo de execução;

2. *Geração de código intermediário*: cada bloco de instruções tem sua semântica descrita através de uma representação independente de máquina;

3. *Otimização*: a descrição em alto nível do bloco de instruções é analisada para aplicar eventuais otimizações; como este processo é realizado durante a execução, normalmente somente otimizações com baixo custo computacional são aplicáveis;

4. *Codificação*: o bloco de instruções otimizado é traduzido para instruções da máquina física, que podem ser diferentes das instruções do código original;

5. *Caching*: blocos de instruções com execução muito frequente têm sua tradução armazenada em cache, para evitar ter de traduzi-los e otimizá-los novamente;

6. *Execução*: o bloco de instruções traduzido é finalmente executado nativamente pelo processador da máquina real.

Esse processo pode ser simplificado caso as instruções de máquina do código convidado sejam as mesmas do processador real subjacente, o que torna desnecessário traduzir os blocos de instruções em uma representação independente de máquina.

33.4.4 Paravirtualização

A Seção 33.2 mostrou que as arquiteturas de alguns processadores, como o *Intel x86*, eram difíceis de virtualizar, porque algumas instruções sensíveis não podiam ser interceptadas pelo hipervisor. Essas instruções sensíveis deviam ser então detectadas e interpretadas pelo hipervisor, em tempo de carga do código na memória.

Além das instruções sensíveis em si, outros aspectos da interface software/hardware trazem dificuldades ao desenvolvimento de máquinas virtuais de sistema eficientes. Uma dessas áreas é o mecanismo de entrega e tratamento de interrupções pelo processador, baseado na noção de uma *tabela de interrupções*, que contém uma

função registrada para cada tipo de interrupção a tratar. Outra área da interface software/hardware que pode trazer dificuldades é a gerência de memória, pois o TLB (*Translation Lookaside Buffer*, Seção 15.6.4) dos processadores *x86* é gerenciado diretamente pelo hardware, sem possibilidade de intervenção direta do hipervisor no caso de uma falta de página.

No início dos anos 2000, alguns pesquisadores investigaram a possibilidade de modificar a interface entre o hipervisor e os sistemas operacionais convidados, oferecendo a estes um hardware virtual que é similar, mas não idêntico ao hardware real. Essa abordagem, denominada *paravirtualização*, permite um melhor acoplamento entre os sistemas convidados e o hipervisor, o que leva a um desempenho significativamente melhor das máquinas virtuais.

Modificações na interface de sistema do hardware virtual (*system ISA*) exigem uma adaptação dos sistemas operacionais convidados, para que estes possam executar sobre a plataforma virtual. Em particular, o hipervisor define uma API denominada *chamadas de hipervisor* (*hypercalls*), que cada sistema convidado deve usar para acessar a interface de sistema do hardware virtual. Todavia, a interface de usuário (*user ISA*) do hardware é preservada, permitindo que as aplicações convidadas executem sem necessidade de modificações. A Figura 33.4 ilustra esse conceito.

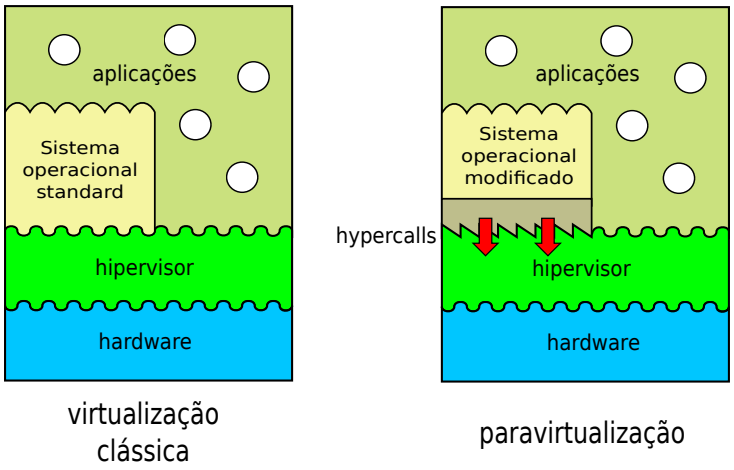


Figura 33.4: Paravirtualização.

Os primeiros ambientes a adotar a paravirtualização foram o *Denali* [Whitaker et al., 2002] e o *Xen* [Barham et al., 2003]. O *Denali* é um ambiente experimental de paravirtualização construído na Universidade de Washington, que pode suportar dezenas de milhares de máquinas virtuais sobre um computador PC convencional. O projeto *Denali* não se preocupa em suportar sistemas operacionais comerciais, sendo voltado à execução maciça de minúsculas máquinas virtuais para serviços de rede. Já o ambiente de máquinas virtuais *Xen* (vide Seção 34.2.2) permite executar sistemas operacionais convencionais como Linux e Windows, modificados para executar sobre um hipervisor.

Embora exija que o sistema convidado seja adaptado ao hipervisor, o que diminui sua portabilidade, a paravirtualização permite que o sistema convidado acesse alguns recursos do hardware diretamente, sem a intermediação ativa do hipervisor. Nesses casos, o acesso ao hardware é apenas monitorado pelo hipervisor, que informa ao sistema convidado seus limites, como as áreas de memória e de disco disponíveis. O acesso aos demais dispositivos, como mouse e teclado, também é direto: o hipervisor apenas

gerencia os conflitos, no caso de múltiplos sistemas convidados em execução simultânea. Apesar de exigir modificações nos sistemas operacionais convidados, a paravirtualização teve sucesso, por conta do desempenho obtido nos sistemas virtualizados, além de simplificar a interface de baixo nível dos sistemas convidados.

33.5 Aspectos de desempenho

De acordo com os princípios de Goldberg e Popek, o hipervisor deve permitir que a máquina virtual execute diretamente sobre o hardware sempre que possível, para não prejudicar o desempenho dos sistemas convidados. O hipervisor deve retomar o controle do processador somente quando a máquina virtual tentar executar operações que possam afetar o correto funcionamento do sistema, o conjunto de operações de outras máquinas virtuais ou do próprio hardware. O hipervisor deve então simular com segurança a operação solicitada e devolver o controle à máquina virtual.

Na prática, os hipervisores nativos e convidados raramente são usados em sua forma conceitual. Várias otimizações são inseridas nas arquiteturas apresentadas, com o objetivo principal de melhorar o desempenho das aplicações nos sistemas convidados. Como os pontos cruciais do desempenho dos sistemas de máquinas virtuais são as operações de entrada/saída, as principais otimizações utilizadas em sistemas de produção dizem respeito a essas operações. Quatro formas de otimização são usuais:

- Em hipervisores nativos (Figura 33.5, esquerda):

  1. O sistema convidado (*guest system*) acessa diretamente o hardware. Essa forma de acesso é implementada por modificações no núcleo do sistema convidado e no hipervisor. Essa otimização é implementada, por exemplo, no subsistema de gerência de memória do ambiente Xen [Barham et al., 2003].

- Em hipervisores convidados (Figura 33.5, direita):

  2. O sistema convidado (*guest system*) acessa diretamente o sistema nativo (*host system*). Essa otimização é implementada pelo hipervisor, oferecendo partes da API do sistema nativo ao sistema convidado. Um exemplo dessa otimização é a implementação do sistema de arquivos no *VMware* [VMware, 2000]: em vez de reconstruir integralmente o sistema de arquivos sobre um dispositivo virtual provido pelo hipervisor, o sistema convidado faz uso da implementação de sistema de arquivos existente no sistema nativo.
  3. O sistema convidado (*guest system*) acessa diretamente o hardware. Essa otimização é implementada parcialmente pelo hipervisor e parcialmente pelo sistema nativo, pelo uso de um *device driver* específico. Um exemplo típico dessa otimização é o acesso direto a dispositivos físicos como leitor de CDs, hardware gráfico e interface de rede provida pelo sistema *VMware* aos sistemas operacionais convidados [VMware, 2000].
  4. O hipervisor acessa diretamente o hardware. Neste caso, um *device driver* específico é instalado no sistema nativo, oferecendo ao hipervisor uma interface de baixo nível para acesso ao hardware subjacente. Essa abordagem,

também ilustrada na Figura 33.6, é usada pelo sistema *VMware* [VMware, 2000].

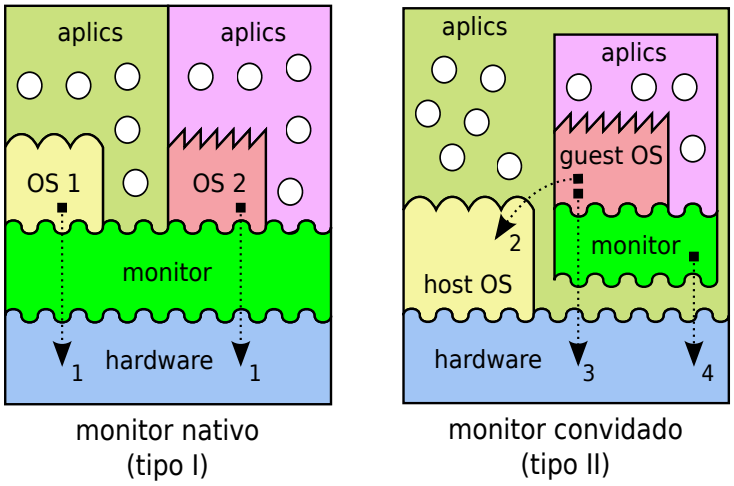


Figura 33.5: Otimizações em sistemas de máquinas virtuais.

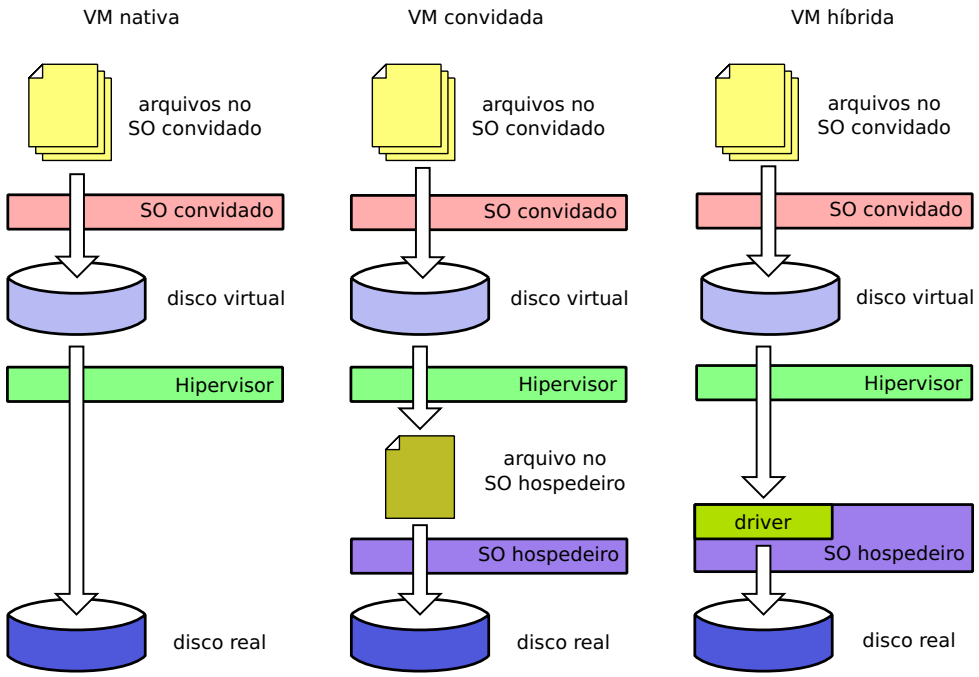


Figura 33.6: Desempenho de hipervisores nativos e convidados.

33.6 Migração de máquinas virtuais

Recentemente, a virtualização vem desempenhando um papel importante na gerência de sistemas computacionais corporativos, graças à facilidade de *migração* de máquinas virtuais implementada pelos hipervisores modernos. Na migração, uma máquina virtual e seu sistema convidado são transferidos através da rede de um hipervisor para outro, executando em equipamentos distintos, sem ter de reiniciá-los. A

máquina virtual tem seu estado preservado e prossegue sua execução no hipervisor de destino assim que a migração é concluída. De acordo com [Clark et al., 2005], as técnicas mais frequentes para implementar a migração de máquinas virtuais são:

- *stop-and-copy*: consiste em suspender a máquina virtual, transferir o conteúdo de sua memória para o hipervisor de destino e retomar a execução em seguida. É uma abordagem simples, mas implica em parar completamente os serviços oferecidos pelo sistema convidado enquanto durar a migração (que pode demorar algumas dezenas de segundos);

- *demand-migration*: a máquina virtual é suspensa apenas durante a cópia das estruturas de memória do núcleo do sistema operacional convidado para o hipervisor de destino, o que dura alguns milissegundos. Em seguida, a execução da máquina virtual é retomada e o restante das páginas de memória da máquina virtual é transferido sob demanda, através dos mecanismos de tratamento de faltas de página. Nesta abordagem a interrupção do serviço tem duração mínima, mas a migração completa pode demorar muito tempo.

- *pre-copy*: consiste basicamente em copiar para o hipervisor de destino todas as páginas de memória da máquina virtual enquanto esta executa; a seguir, a máquina virtual é suspensa e as páginas modificadas depois da cópia inicial são novamente copiadas no destino; uma vez terminada a cópia dessas páginas, a máquina pode retomar sua execução no destino. Esta abordagem, usada no hipervisor *Xen* [Barham et al., 2003], é a que oferece o melhor compromisso entre o tempo de suspensão do serviço e a duração total da migração.

Referências

P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Symposium on Networked Systems Design and Implementation*, 2005.

F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the Multics system. In *AFIPS Conference Proceedings*, pages 185–196, 1965.

R. Goldberg. Architecture of virtual machines. In *AFIPS National Computer Conference*, 1973.

R. Goldberg and P. Mager. Virtual machine technology: A bridge from large mainframes to networks of small computers. *IEEE Proceedings Compcon Fall 79*, pages 210–213, 1979.

IBM. *Power Instruction Set Architecture – Version 2.04*. IBM Corporation, april 2007.

S. Nanda and T. Chiueh. A survey on virtualization technologies. Technical report, University of New York at Stony Brook, 2005. Departament of Computer Science.

G. Popek and R. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

J. Robin and C. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *9th USENIX Security Symposium*, 2000.

M. Rosenblum. The reincarnation of virtual machines. *Queue Focus - ACM Press*, pages 34–40, 2004.

M. Rosenblum and T. Garfinkel. Virtual machine monitors: Current technology and future trends. *IEEE Computer*, May 2005.

R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kägi, F. Leung, and L. Smith. Intel virtualization technology. *IEEE Computer*, May 2005.

D. Ung and C. Cifuentes. Dynamic re-engineering of binary code with run-time feedbacks. *Science of Computer Programming*, 60(2):189–204, April 2006.

VMware. VMware technical white paper. Technical report, VMware, Palo Alto, CA - USA, 2000.

A. Whitaker, M. Shaw, and S. Gribble. Denali: A scalable isolation kernel. In *ACM SIGOPS European Workshop*, September 2002.

C.-H. Yen. Solaris operating system - hardware virtualization product architecture. Technical Report 820-3703-10, Sun Microsystems, November 2007.

Capítulo 34

Virtualização na prática

34.1 Aplicações da virtualização

Por permitir o acoplamento entre componentes de sistema com interfaces distintas, a virtualização tem um grande número de aplicações possíveis. As principais delas serão brevemente discutidas nesta seção:

Construção de binários portáveis: este uso da virtualização começou na década de 1970, com o compilador UCSD Pascal, que traduzia o código fonte Pascal em um código binário *P-Code*, para uma máquina virtual chamada *P-Machine*. A execução do código binário ficava então a cargo de uma implementação da *P-Machine* sobre a máquina alvo. Esse esquema foi posteriormente adotado por linguagens como Java, C#, Perl e Python, nas quais o código fonte é compilado em *bytecodes* para uma máquina virtual. Assim, uma aplicação Java compilada em *bytecode* pode executar em qualquer plataforma onde uma implementação da máquina virtual Java (*JVM - Java Virtual Machine*) esteja disponível.

Compartilhamento de hardware: executar simultaneamente várias instâncias de sistema operacional sobre a mesma plataforma de hardware. Uma área de aplicação dessa possibilidade é a chamada *consolidação de servidores*, que consiste em agrupar vários servidores de rede (web, e-mail, proxy, banco de dados, etc.) sobre o mesmo computador: ao invés de instalar vários computadores fisicamente isolados para abrigar cada um dos serviços, pode ser instalado um único computador, com maior capacidade, para suportar várias máquinas virtuais, cada uma abrigando um sistema operacional convidado e seu respectivo serviço.

Suporte a aplicações legadas: pode-se preservar ambientes virtuais com sistemas operacionais antigos para a execução de aplicações legadas, sem a necessidade de manter computadores reservados para isso.

Experimentação em redes: é possível construir uma rede de máquinas virtuais, comunicando por protocolos de rede como o TCP/IP, sobre um único computador hospedeiro. Isto torna possível o desenvolvimento e implantação de serviços de rede e de sistemas distribuídos sem a necessidade de uma rede real, o que é especialmente interessante em ensino e pesquisa.

Ensino: em disciplinas de rede e de sistema, um aluno deve ter a possibilidade de modificar as configurações da máquina para poder realizar seus experimentos. Essa possibilidade é uma verdadeira "dor de cabeça" para os administradores de laboratórios de ensino. Todavia, um aluno pode lançar uma máquina virtual e ter controle completo sobre ela, mesmo não tendo acesso às configurações da máquina real subjacente.

Segurança: a propriedade de isolamento provida pelo hipervisor torna esta abordagem útil para isolar domínios, usuários e/ou aplicações não confiáveis. As máquinas virtuais de sistema operacional (Seção 32.3) foram criadas justamente com o objetivo de isolar subsistemas particularmente críticos, como servidores Web, DNS e de e-mail. Pode-se também usar máquinas virtuais como plataforma de execução de programas suspeitos, para inspecionar seu funcionamento e seus efeitos sobre o sistema operacional convidado.

Desenvolvimento de baixo nível: o uso de máquinas virtuais para o desenvolvimento partes do núcleo do sistema operacional, módulos e protocolos de rede, tem vários benefícios com o uso de máquinas virtuais. Por exemplo, o desenvolvimento e os testes podem ser feitos sobre a mesma plataforma. Outra vantagem visível é o menor tempo necessário para instalar e lançar um núcleo em uma máquina virtual, quando comparado a uma máquina real. Por fim, a execução em uma máquina virtual pode ser melhor acompanhada e depurada que a execução equivalente em uma máquina real.

Tolerância a faltas: muitos hipervisores oferecem suporte ao *checkpointing*, ou seja, à possibilidade de salvar o estado interno de uma máquina virtual e de poder restaurá-lo posteriormente. Com *checkpoints* periódicos, torna-se possível retornar a execução de uma máquina virtual a um estado salvo anteriormente, em caso de falhas ou incidentes de segurança.

Computação sob demanda: o desacoplamento entre o hardware real e o sistema operacional proporcionado pelas máquinas virtuais as tornou um suporte adequado para a oferta de serviços através da rede. Na computação em nuvem, imensas centrais de processamento de dados alugam máquinas virtuais que podem ser instanciadas, configuradas e destruídas por seus clientes sob demanda, conforme sua necessidade no momento.

Virtualização de redes: máquinas virtuais têm sido usadas para implementar dispositivos de rede como roteadores, *switches* e *firewalls*, em uma abordagem chamada *Virtualização de Funções de Rede* (NFV - *Network Function Virtualization*), diminuindo a quantidade de hardware proprietário na infraestrutura de rede e agilizando operações de reconfiguração da rede (que passam a ser feitas exclusivamente por software).

34.2 Ambientes de máquinas virtuais

Esta seção apresenta alguns exemplos de sistemas de máquinas virtuais de uso corrente. Entre eles há máquinas virtuais de aplicação e de sistema, com virtualização total ou paravirtualização, além de abordagens híbridas. Eles foram escolhidos por estarem entre os mais representativos de suas respectivas classes.

34.2.1 VMware

O hipervisor da *VMware* é um dos mais difundidos, provendo uma implementação completa da interface x86 ao sistema convidado. Embora essa interface seja extremamente genérica para o sistema convidado, acaba conduzindo a um hipervisor mais complexo. Como podem existir vários sistemas operacionais em execução sobre mesmo hardware, o hipervisor tem que emular certas instruções para representar corretamente um processador virtual em cada máquina virtual, fazendo uso intensivo dos mecanismos de tradução dinâmica [VMware, 2000; Newman et al., 2005]. Atualmente, a empresa *VMware* produz vários hipervisores, entre eles:

- *VMware Workstation*: hipervisor convidado para ambientes *desktop*;
- *VMware ESXi Server*: hipervisor nativo para servidores de grande porte, possui um núcleo proprietário chamado *vmkernel* e utiliza *Linux* para prover outros serviços, tais como a gerência de usuários.

O *VMware Workstation* utiliza as estratégias de virtualização total, tradução dinâmica (Seção 33.4) e o suporte de hardware, quando disponível. O *VMware ESXi Server* implementa também a paravirtualização. Por razões de desempenho, o hipervisor do *VMware* utiliza uma abordagem híbrida (Seção 33.5) para implementar a interface do hipervisor com as máquinas virtuais [Sugerman et al., 2001]. O controle de exceção e o gerenciamento de memória são realizados por acesso direto ao hardware, mas o controle de entrada/saída usa o sistema hospedeiro. Para garantir que não ocorra nenhuma colisão de memória entre o sistema convidado e o real, o hipervisor *VMware* aloca uma parte da memória para uso exclusivo de cada sistema convidado.

Para controlar o sistema convidado, o *VMware Workstation* intercepta todas as interrupções do sistema convidado. Sempre que uma exceção é causada no convidado, é examinada primeiro pelo hipervisor. As interrupções de entrada/saída são remetidas para o sistema hospedeiro, para que sejam processadas corretamente. As exceções geradas pelas aplicações no sistema convidado (como as chamadas de sistema, por exemplo) são remetidas para o sistema convidado.

34.2.2 Xen

O ambiente *Xen* é um hipervisor nativo para a plataforma *x86* que implementa a paravirtualização. Ele permite executar sistemas operacionais como Linux e Windows especialmente modificados para executar sobre o hipervisor [Barham et al., 2003]. Versões mais recentes do sistema *Xen* utilizam o suporte de virtualização disponível nos processadores atuais, o que torna possível a execução de sistemas operacionais convidados sem modificações, embora com um desempenho ligeiramente menor que no caso de sistemas paravirtualizados. De acordo com seus desenvolvedores, o custo e impacto das alterações nos sistemas convidados são baixos e a diminuição do custo da virtualização compensa essas alterações: a degradação média de desempenho observada em sistemas virtualizados sobre a plataforma *Xen* não excede 5%. As principais modificações impostas pelo ambiente *Xen* a um sistema operacional convidado são:

- O mecanismo de entrega de interrupções passa a usar um serviço de eventos oferecido pelo hipervisor; o núcleo convidado deve registrar uma tabela de tratadores de exceções junto ao hipervisor;

- as operações de entrada/saída de dispositivos são feitas através de uma interface simplificada, independente de dispositivo, que usa *buffers* circulares de tipo produtor/consumidor;
- o núcleo convidado pode consultar diretamente as tabelas de segmentos e páginas da memória usada por ele e por suas aplicações, mas as modificações nas tabelas devem ser solicitadas ao hipervisor;
- o núcleo convidado deve executar em um nível de privilégio inferior ao do hipervisor;
- o núcleo convidado deve implementar uma função de tratamento das chamadas de sistema de suas aplicações, para evitar que elas tenham de passar pelo hipervisor antes de chegar ao núcleo convidado.

Como o hipervisor deve acessar os dispositivos de hardware, ele deve dispor dos *drivers* adequados. Já os núcleos convidados não precisam de *drivers* específicos, pois eles acessam dispositivos virtuais através de uma interface simplificada. Para evitar o desenvolvimento de *drivers* específicos para o hipervisor, o ambiente *Xen* usa uma abordagem alternativa: a primeira máquina virtual (chamada VM_0) pode acessar o hardware diretamente e provê os *drivers* necessários ao hipervisor. As demais máquinas virtuais ($VM_i, i > 0$) acessam o hardware virtual através do hipervisor, que usa os *drivers* da máquina VM_0 conforme necessário. Essa abordagem, apresentada na Figura 34.1, simplifica muito a evolução do hipervisor, por permitir utilizar os *drivers* desenvolvidos para o sistema Linux.

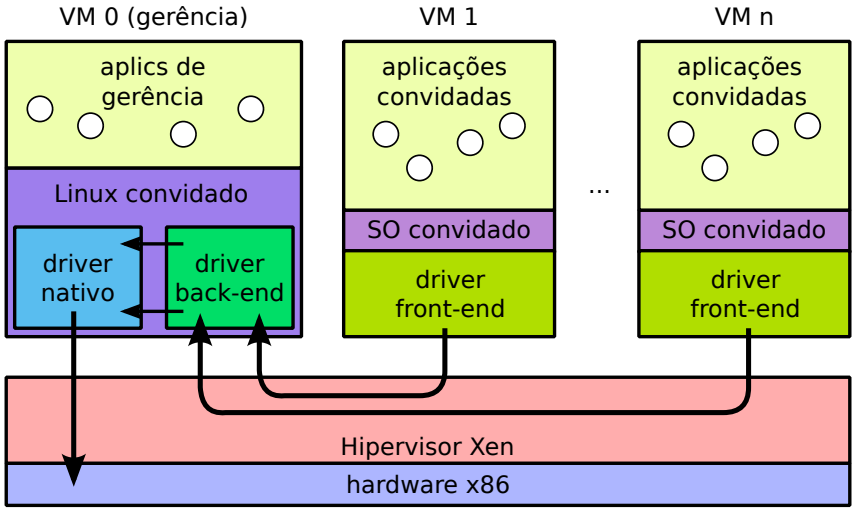


Figura 34.1: O hipervisor *Xen*.

O hipervisor *Xen* pode ser considerado uma tecnologia madura, sendo muito utilizado em sistemas de produção. O seu código fonte está liberado sob a licença *GNU General Public Licence* (GPL). Atualmente, o ambiente *Xen* suporta os sistemas Windows, Linux e NetBSD. Várias distribuições Linux já possuem suporte nativo ao *Xen*.

34.2.3 QEMU

O *QEMU* é um emulador de hardware [Bellard, 2005]. Não requer alterações ou otimizações no sistema hospedeiro, pois utiliza intensivamente a tradução dinâmica

(Seção 33.4) como técnica para prover a virtualização. O *QEMU* oferece dois modos de operação:

- *Emulação total do sistema*: emula um sistema completo, incluindo processador (normalmente um processador no padrão x86) e vários periféricos. Neste modo o emulador pode ser utilizado para executar diferentes sistemas operacionais;
- *Emulação no modo de usuário*: disponível apenas para o sistema Linux. Neste modo o emulador pode executar processos Linux compilados em diferentes plataformas (por exemplo, um programa compilado para um processador *x86* pode ser executado em um processador *PowerPC* e vice-versa).

Durante a emulação de um sistema completo, o *QEMU* implementa uma MMU (*Memory Management Unit*) totalmente em software, para garantir o máximo de portabilidade. Quando em modo usuário, o *QEMU* simula uma MMU simplificada através da chamada de sistema `mmap` (que permite mapear um arquivo em uma região da memória) do sistema hospedeiro.

O *VirtualBox* [VirtualBox, 2008] é um ambiente de máquinas virtuais construído sobre o hipervisor *QEMU*. Ele é similar ao *VMware Workstation* em muitos aspectos. Atualmente, pode tirar proveito do suporte à virtualização disponível nos processadores Intel e AMD. Originalmente desenvolvido pela empresa *Innotek*, o *VirtualBox* foi depois adquirido pela *Sun Microsystems* e liberado para uso público sob a licença *GPLv2*.

34.2.4 KVM

O KVM (*Kernel-based Virtual Machine*) é um hipervisor convidado (de tipo 2) embutido no núcleo Linux desde sua versão 2.6.20 [Kivity et al., 2007]. Ele foi construído com o objetivo de explorar as extensões de virtualização disponíveis nos processadores modernos para construir máquinas virtuais no *userspace* do Linux. Embora seja construído para hospedeiros Linux, o KVM suporta vários sistemas convidados distintos.

No ambiente KVM, cada máquina virtual dispõe de um conjunto de CPUs virtuais (VCPUs) e de recursos paravirtualizados, como interfaces de rede e de disco, que são implementados pelo hipervisor. Além disso, hipervisor reaproveita grande parte dos mecanismos do núcleo Linux subjacente para realizar o gerenciamento de memória, escalonamento de CPUs e de entrada/saída.

Por default, o KVM realiza a paravirtualização, ou seja, emula as operações de entrada/saída e executa as demais instruções do sistema convidado diretamente pelo processador, usando as extensões de virtualização quando necessário. Entretanto, Se o sistema convidado precisar de um processador distinto do usado pelo sistema hospedeiro, o KVM usa o emulador QEMU para virtualizar a execução das instruções do sistema convidado.

Em termos de implementação, o KVM consiste de um conjunto de módulos de núcleo e de um arquivo de controle (`/dev/kvm`). No espaço de usuário, o KVM pode fazer uso do QEMU e também de uma biblioteca para o gerenciamento de máquinas virtuais chamada *LibVirt*. Os sistemas convidados deve acessar os dispositivos paravirtualizados através de uma biblioteca específica chamada *VirtIO*.

34.2.5 Docker

Docker [Merkel, 2014] é um *framework* para a construção e gestão de contêineres (máquinas virtuais de sistema operacional). Nesse *framework*, uma aplicação corporativa complexa pode ser "empacotada" em um contêiner com todas as suas dependências (bibliotecas, serviços auxiliares, etc), agilizando e simplificando sua distribuição, implantação e configuração. Essa abordagem se mostrou excelente para a distribuição e ativação de serviços em nuvens computacionais.

Para construir e manter o ambiente de virtualização, o *framework* Docker usa várias tecnologias providas pelo núcleo Linux, como os suportes de contêiner LXC (*Linux Containers*) e *systemd-nspawn*, o sistema de arquivos em camadas *OverlayFS* e os mecanismos de isolamento e gestão de recursos *cgroups* e *namespaces*.

34.2.6 JVM

Tendo sido originalmente concebida para o desenvolvimento de pequenos aplicativos e programas de controle de aparelhos eletroeletrônicos, a linguagem Java mostrou-se ideal para ser usada na Internet. O que a torna tão atraente é o fato de programas escritos nessa linguagem de programação poderem ser executados em praticamente qualquer plataforma.

A virtualização é o fator responsável pela independência dos programas Java do hardware e dos sistemas operacionais: um programa escrito em Java, ao ser compilado, gera um código binário específico para uma máquina abstrata denominada *máquina virtual Java* (*JVM - Java Virtual Machine*). A linguagem de máquina executada pela máquina virtual Java é denominada *bytecode* Java, e não corresponde a instruções de nenhum processador real. A máquina virtual deve então interpretar todas as operações do *bytecode*, utilizando as instruções da máquina real subjacente para executá-las.

A vantagem mais significativa da abordagem adotada por Java é a *portabilidade* do código executável: para que uma aplicação Java possa executar sobre uma determinada plataforma, basta que a máquina virtual Java esteja disponível ali (na forma de um suporte de execução denominado JRE - *Java Runtime Environment*). Assim, a portabilidade dos programas Java depende unicamente da portabilidade da própria máquina virtual Java.

O suporte de execução Java pode estar associado a um navegador Web, o que permite que código Java seja associado a páginas Web, na forma de pequenas aplicações denominadas *applets*, que são trazidas junto com os demais componentes de página Web e executam localmente no navegador. A Figura 34.2 mostra os principais componentes da plataforma Java.

É importante ressaltar que a adoção de uma máquina virtual como suporte de execução não é exclusividade da linguagem Java, nem foi inventada por seus criadores. As primeiras experiências de execução de aplicações sobre máquinas abstratas remontam aos anos 1970, com a linguagem *UCSD Pascal*. Hoje, muitas linguagens adotam estratégias similares, como Java, C#, Python, Perl, Lua e Ruby. Em C#, o código fonte é compilado em um formato intermediário denominado CIL (*Common Intermediate Language*), que executa sobre uma máquina virtual CLR (*Common Language Runtime*). CIL e CLR fazem parte da infraestrutura .NET da Microsoft.

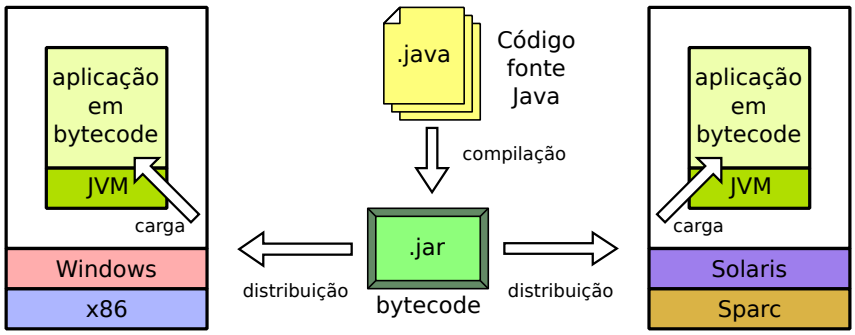


Figura 34.2: Máquina virtual Java.

34.2.7 FreeBSD Jails

O sistema operacional *FreeBSD* oferece um mecanismo de confinamento de processos denominado *Jails*, criado para aumentar a segurança de serviços de rede. Esse mecanismo consiste em criar domínios de execução distintos (denominados *jails* ou celas), conforme descrito na Seção 32.3. Cada cela contém um subconjunto de processos e recursos (arquivos, conexões de rede) que pode ser gerenciado de forma autônoma, como se fosse um sistema separado [McKusick and Neville-Neil, 2005].

Cada domínio é criado a partir de um diretório previamente preparado no sistema de arquivos. Um processo que executa a chamada de sistema `jail` cria uma nova cela e é colocado dentro dela, de onde não pode mais sair, nem seus filhos. Além disso, os processos em um domínio não podem:

- Reconfigurar o núcleo (através da chamada `sysctl`, por exemplo);
- Carregar/retirar módulos do núcleo;
- Mudar configurações de rede (interfaces e rotas);
- Montar/desmontar sistemas de arquivos;
- Criar novos *devices*;
- Realizar modificações de configurações do núcleo em tempo de execução;
- Acessar recursos que não pertençam ao seu próprio domínio.

Essas restrições se aplicam mesmo a processos que estejam executando com privilégios de administrador (*root*).

Pode-se considerar que o sistema *FreeBSD Jails* virtualiza somente partes do sistema hospedeiro, como a árvore de diretórios (cada domínio tem sua própria visão do sistema de arquivos), espaços de nomes (cada domínio mantém seus próprios identificadores de usuários, processos e recursos de IPC) e interfaces de rede (cada domínio tem sua interface virtual, com endereço de rede próprio). Os demais recursos (como as instruções de máquina e chamadas de sistema) são preservadas, ou melhor, podem ser usadas diretamente. Essa virtualização parcial demanda um custo computacional muito baixo, mas exige que todos os sistemas convidados executem sobre o mesmo núcleo.

34.2.8 Valgrind

O *Valgrind* [Nethercote and Seward, 2007] é uma ferramenta de depuração de uso da memória RAM e problemas correlatos. Ele permite investigar vazamentos de memória (*memory leaks*), acessos a endereços inválidos, padrões de uso dos caches e outras operações envolvendo o uso da memória RAM. O *Valgrind* foi desenvolvido para plataforma *x86 Linux*, mas existem versões experimentais para outras plataformas.

Tecnicamente, o *Valgrind* é um hipervisor de aplicação que virtualiza o processador através de técnicas de tradução dinâmica. Ao iniciar a análise de um programa, o *Valgrind* traduz o código executável do mesmo para um formato interno independente de plataforma denominado IR (*Intermediate Representation*). Após a conversão, o código em IR é instrumentado, através da inserção de instruções para registrar e verificar as operações de alocação, acesso e liberação de memória. A seguir, o programa IR devidamente instrumentado é traduzido no formato binário a ser executado sobre o processador virtual. O código final pode ser até 50 vezes mais lento que o código original, mas essa perda de desempenho normalmente não é muito relevante durante a análise ou depuração de um programa.

34.2.9 User-Mode Linux

O *User-Mode Linux* é um hipervisor simples, proposto por Jeff Dike em 2000 [Dike, 2000]. Nele, o núcleo do Linux foi portado de forma a poder executar sobre si mesmo, como um processo do próprio Linux. O resultado é um *userspace* separado e isolado na forma de uma máquina virtual, que utiliza dispositivos de hardware virtualizados a partir dos serviços providos pelo sistema hospedeiro.

Essa máquina virtual é capaz de executar todos os serviços e aplicações disponíveis para o sistema hospedeiro. Além disso, o custo de processamento e de memória das máquinas virtuais *User-Mode Linux* é geralmente menor que aquele imposto por outros hipervisores mais complexos. O *User-Mode Linux* está integrado ao núcleo Linux desde a versão 2.6 deste.

O UML implementa um hipervisor convidado, que executa na forma de um processo no sistema hospedeiro. Os processos em execução na máquina virtual não têm acesso direto aos recursos do sistema hospedeiro. A implementação do UML se baseia na interceptação das chamadas de sistema emitidas pelo convidado. Essa interceptação é realizada através da chamada de sistema `ptrace`, que permite observar e controlar a execução de outros processos. Assim, o hipervisor recebe o controle de todas as chamadas de sistema de entrada/saída geradas pelas máquinas virtuais. Além disso, todos os sinais gerados ou enviados às máquinas virtuais também são interceptados.

Referências

P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, 2005.

J. Dike. A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference*, 2000.

A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

M. McKusick and G. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2005.

D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM Conference on Programming Language Design and Implementation*, San Diego - California - USA, june 2007.

M. Newman, C.-M. Wiberg, and B. Braswell. *Server Consolidation with VMware ESX Server*. IBM RedBooks, 2005. http://www.redbooks.ibm.com.

J. Sugerman, G. Venkitachalam, and B. H. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, pages 1–14, 2001.

I. VirtualBox. The VirtualBox architecture. http://www.virtualbox.org/wiki/VirtualBox_architecture, 2008.

VMware. VMware technical white paper. Technical report, VMware, Palo Alto, CA - USA, 2000.

Capítulo A

O descritor de tarefa do Linux

A estrutura em linguagem C apresentada neste anexo constitui o descritor de tarefas (*Task Control Block*) do Linux (estudado na Seção 5.1). Ela foi extraída do arquivo `include/linux/sched.h` do código fonte do núcleo Linux versão 1.0. Essa versão do núcleo foi escolhida por sua simplicidade; na versão mais recente do código-fonte do Linux (4.18 nesta data), a mesma estrutura possui cerca de 600 linhas de código C.

```
1  // part of the Linux kernel source code, file include/linux/sched.h, version 1.0
2
3  struct task_struct {
4
5  /* these are hardcoded - don't touch */
6         volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
7         long counter;
8         long priority;
9         unsigned long signal;
10        unsigned long blocked; /* bitmap of masked signals */
11        unsigned long flags; /* per process flags, defined below */
12        int errno;
13        int debugreg[8]; /* Hardware debugging registers */
14
15 /* various fields */
16        struct task_struct *next_task, *prev_task;
17
18        struct sigaction sigaction[32];
19        unsigned long saved_kernel_stack;
20        unsigned long kernel_stack_page;
21        int exit_code, exit_signal;
22        int elf_executable:1;
23        int dumpable:1;
24        int swappable:1;
25        int did_exec:1;
26        unsigned long start_code, end_code, end_data, start_brk, brk,
27                start_stack, start_mmap;
28        unsigned long arg_start, arg_end, env_start, env_end;
29        int pid,pgrp,session,leader;
30        int    groups[NGROUPS];
31
32        /* pointers to (original) parent process, youngest child, younger sibling,
33         * older sibling, respectively. (p->father can be replaced with
34         * p->p_pptr->pid */
35        struct task_struct *p_opptr,*p_pptr, *p_cptr, *p_ysptr, *p_osptr;
```

```
36
37        struct wait_queue *wait_chldexit; /* for wait4() */
38        /* For ease of programming... Normal sleeps don't need to
39         * keep track of a wait-queue: every task has an entry of its own */
40
41        /* user/group IDs */
42        unsigned short uid, euid, suid;
43        unsigned short gid, egid, sgid;
44
45        unsigned long timeout;
46        unsigned long it_real_value, it_prof_value, it_virt_value;
47        unsigned long it_real_incr, it_prof_incr, it_virt_incr;
48        long utime,stime,cutime,cstime,start_time;
49        unsigned long min_flt, maj_flt;
50        unsigned long cmin_flt, cmaj_flt;
51        struct rlimit rlim[RLIM_NLIMITS];
52        unsigned short used_math;
53        unsigned short rss; /* number of resident pages */
54        char comm[16];
55        struct vm86_struct * vm86_info; /* virtual memory info */
56        unsigned long screen_bitmap;
57
58 /* file system info */
59        int link_count;
60        int tty;            /* -1 if no tty, so it must be signed */
61        unsigned short umask;
62        struct inode * pwd;
63        struct inode * root;
64        struct inode * executable;
65        struct vm_area_struct * mmap;
66        struct shm_desc *shm;
67        struct sem_undo *semun;
68        struct file * filp[NR_OPEN];
69        fd_set close_on_exec;
70
71 /* ldt for this task - used by Wine. If NULL, default_ldt is used */
72        struct desc_struct *ldt;
73
74 /* tss for this task */
75        struct tss_struct tss;
76
77 #ifdef NEW_SWAP
78        unsigned long old_maj_flt; /* old value of maj_flt */
79        unsigned long dec_flt;     /* page fault count of the last time */
80        unsigned long swap_cnt;    /* number of pages to swap on next pass */
81        short swap_table;          /* current page table */
82        short swap_page;           /* current page */
83 #endif NEW_SWAP
84
85        struct vm_area_struct *stk_vma;
86 };
```