

Sistemas Operacionais: Conceitos e Mecanismos

Prof. Carlos A. Maziero, Dr.
DAINF – UTFPR

3 de junho de 2013

Sistemas Operacionais: Conceitos e Mecanismos

© Carlos Alberto Maziero, 2013

Sobre o autor: Carlos Alberto Maziero é professor adjunto do Departamento Acadêmico de Informática da Universidade Tecnológica Federal do Paraná (UTFPR) desde julho de 2011. Anteriormente, foi professor titular do Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná (PUCPR), entre 1998 e 2011, e professor adjunto do Departamento de Automação e Sistemas 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-Doutorado em Segurança da Informação (Università degli Studi di Milano – Italia, 2009). Tem atuação em pesquisa nas áreas de sistemas operacionais, segurança de sistemas e sistemas distribuídos.



Este texto está licenciado sob a Licença *Attribution-NonCommercial-ShareAlike 3.0 Unported* da *Creative Commons* (CC). Em resumo, você deve creditar a obra da forma especificada pelo autor ou licenciante (mas não de maneira que sugira que estes concedem qualquer aval a você ou ao seu uso da obra). Você não pode usar esta obra para fins comerciais. Se você alterar, transformar ou criar com base nesta obra, você poderá distribuir a obra resultante apenas sob a mesma licença, ou sob uma licença similar à presente. Para ver uma cópia desta licença, visite <http://creativecommons.org/licenses/by-nc-sa/3.0/>.

Este texto foi produzido usando exclusivamente software livre: Sistema Operacional *GNU/Linux* (distribuições *Fedora* e *Ubuntu*), compilador de texto *L^AT_EX 2_E*, gerenciador de referências *BibTeX*, editor gráfico *Inkscape*, criadores de gráficos *GNUPlot* e *GraphViz* e processador PS/PDF *GhostScript*, entre outros.

*à Lucia, Henrique e Daniel,
as estrelas mais brilhantes do
meu firmamento...*

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 excelentes, 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 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 da computação, vindas de todas as classes sociais. Nossa país não pode mais se dar ao luxo de desperdiçar pessoas inteligentes só porque são pobres.

Prof. Carlos Maziero, Dr.

Curitiba PR, Outubro de 2011

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, Carlos Silla, Diogo Olsen, Douglas da Costa, Fabiano Beraldo, Jeferson Amend, Marcos Pchek Laureano, Paulo Resende, Rafael Hamasaki, Richard Reichardt, Tadeu Ribeiro Reis, Thayse Solis, Thiago Ferreira, 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 primeiros 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 atuo desde julho de 2011, pelas mesmas razões.

Dedico o Capítulo 8 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/MEC¹. O Capítulo 9 deste livro é dedicado 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 convidado².

Carlos Maziero

Curitiba PR, Outubro de 2011

¹Dedico il Capitolo 8 ai colleghi docenti e ricercatori del Dipartimento di Technologie dell'Informazione de l'Università degli Studi di Milano à Crema, dove sono stato per un soggiorno sabbatico nell 2009, con una borsa di post-dottorato CAPES/MEC.

²Je dédie le Chapitre 9 à l'équipe ADEPT IRISA/INRIA Rennes - France, au sein de laquelle j'ai pu passer trois mois très agréables et productifs dans l'hiver 2007-08, en tant qu'enseignant/chercheur invité.

Sumário

1	Conceitos básicos	1
1.1	Objetivos	1
1.1.1	Abstração de recursos	2
1.1.2	Gerência de recursos	3
1.2	Tipos de sistemas operacionais	4
1.3	Funcionalidades	6
1.4	Estrutura de um sistema operacional	8
1.5	Conceitos de hardware	9
1.5.1	Interrupções	11
1.5.2	Proteção do núcleo	14
1.5.3	Chamadas de sistema	15
1.6	Arquiteturas de Sistemas Operacionais	17
1.6.1	Sistemas monolíticos	17
1.6.2	Sistemas em camadas	18
1.6.3	Sistemas micro-núcleo	19
1.6.4	Máquinas virtuais	20
1.7	Um breve histórico dos sistemas operacionais	24
2	Gerência de atividades	26
2.1	Objetivos	26
2.2	O conceito de tarefa	27
2.3	A gerência de tarefas	28
2.3.1	Sistemas mono-tarefa	29
2.3.2	Sistemas multi-tarefa	30
2.3.3	Sistemas de tempo compartilhado	31
2.3.4	Ciclo de vida das tarefas	32
2.4	Implementação de tarefas	34
2.4.1	Contextos	34
2.4.2	Trocas de contexto	35
2.4.3	Processos	36
2.4.4	Threads	41
2.5	Escalonamento de tarefas	46
2.5.1	Objetivos e métricas	47
2.5.2	Escalonamento preemptivo e cooperativo	48
2.5.3	Escalonamento FCFS (<i>First-Come, First Served</i>)	49

2.5.4	Escalonamento SJF (<i>Shortest Job First</i>)	52
2.5.5	Escalonamento por prioridades	53
2.5.6	Outros algoritmos de escalonamento	61
2.5.7	Um escalonador real	62
3	Comunicação entre tarefas	63
3.1	Objetivos	63
3.2	Escopo da comunicação	64
3.3	Características dos mecanismos de comunicação	65
3.3.1	Comunicação direta ou indireta	65
3.3.2	Sincronismo	66
3.3.3	Formato de envio	68
3.3.4	Capacidade dos canais	68
3.3.5	Confiabilidade dos canais	70
3.3.6	Número de participantes	71
3.4	Exemplos de mecanismos de comunicação	72
3.4.1	Filas de mensagens UNIX	72
3.4.2	Pipes	74
3.4.3	Memória compartilhada	75
4	Coordenação entre tarefas	79
4.1	Objetivos	79
4.2	Condições de disputa	79
4.3	Seções críticas	82
4.4	Inibição de interrupções	83
4.5	Soluções com espera ocupada	84
4.5.1	A solução óbvia	84
4.5.2	Alternância de uso	85
4.5.3	O algoritmo de Peterson	86
4.5.4	Instruções <i>Test-and-Set</i>	86
4.5.5	Problemas	88
4.6	Semáforos	88
4.7	Variáveis de condição	91
4.8	Monitores	93
4.9	Problemas clássicos de coordenação	95
4.9.1	O problema dos produtores/consumidores	95
4.9.2	O problema dos leitores/escritores	97
4.9.3	O jantar dos filósofos	99
4.10	Impasses	101
4.10.1	Caracterização de impasses	103
4.10.2	Grafos de alocação de recursos	104
4.10.3	Técnicas de tratamento de impasses	105

5 Gerência de memória	110
5.1 Estruturas de memória	110
5.2 Endereços, variáveis e funções	112
5.2.1 Endereços lógicos e físicos	114
5.2.2 Modelo de memória dos processos	116
5.3 Estratégias de alocação	117
5.3.1 Partições fixas	117
5.3.2 Alocação contígua	119
5.3.3 Alocação por segmentos	120
5.3.4 Alocação paginada	123
5.3.5 Alocação segmentada paginada	132
5.4 Localidade de referências	132
5.5 Fragmentação	135
5.6 Compartilhamento de memória	139
5.7 Memória virtual	141
5.7.1 Mecanismo básico	142
5.7.2 Eficiência de uso	145
5.7.3 Algoritmos de substituição de páginas	146
5.7.4 Conjunto de trabalho	153
5.7.5 A anomalia de Belady	155
5.7.6 Thrashing	156
6 Gerência de arquivos	159
6.1 Arquivos	159
6.1.1 O conceito de arquivo	159
6.1.2 Atributos	160
6.1.3 Operações	161
6.1.4 Formatos	162
6.1.5 Arquivos especiais	166
6.2 Uso de arquivos	167
6.2.1 Abertura de um arquivo	167
6.2.2 Formas de acesso	169
6.2.3 Controle de acesso	170
6.2.4 Compartilhamento de arquivos	172
6.2.5 Exemplo de interface	174
6.3 Organização de volumes	176
6.3.1 Diretórios	177
6.3.2 Caminhos de acesso	179
6.3.3 Atalhos	182
6.3.4 Montagem de volumes	183
6.4 Sistemas de arquivos	184
6.4.1 Arquitetura geral	185
6.4.2 Blocos físicos e lógicos	186
6.4.3 Alocação física de arquivos	187
6.4.4 O sistema de arquivos virtual	199

6.5	Tópicos avançados	200
7	Gerência de entrada/saída	201
7.1	Introdução	201
7.2	Dispositivos de entrada/saída	203
7.2.1	Componentes de um dispositivo	203
7.2.2	Barramentos	203
7.2.3	Interface de acesso	205
7.2.4	Endereçamento	208
7.2.5	Interrupções	210
7.3	Software de entrada/saída	213
7.3.1	Classes de dispositivos	215
7.3.2	Estratégias de interação	217
7.4	Discos rígidos	225
7.4.1	Estrutura física	226
7.4.2	Interface de hardware	226
7.4.3	Escalonamento de acessos	227
7.4.4	Caching de blocos	230
7.4.5	Sistemas RAID	231
7.5	Interfaces de rede	236
7.6	Dispositivos USB	236
7.7	Interfaces de áudio	236
7.8	Interface gráfica	237
7.9	Mouse e teclado	237
7.10	Outros tópicos	237
8	Segurança de sistemas	238
8.1	Introdução	238
8.2	Conceitos básicos	239
8.2.1	Propriedades e princípios de segurança	239
8.2.2	Ameaças	241
8.2.3	Vulnerabilidades	241
8.2.4	Ataques	243
8.2.5	Malwares	245
8.2.6	Infraestrutura de segurança	247
8.3	Fundamentos de criptografia	249
8.3.1	Cifragem e decifragem	249
8.3.2	Criptografia simétrica e assimétrica	250
8.3.3	Resumo criptográfico	252
8.3.4	Assinatura digital	253
8.3.5	Certificado de chave pública	254
8.4	Autenticação	256
8.4.1	Usuários e grupos	257
8.4.2	Técnicas de autenticação	258
8.4.3	Senhas	258
8.4.4	Senhas descartáveis	259

8.4.5	Técnicas biométricas	260
8.4.6	Desafio-resposta	261
8.4.7	Certificados de autenticação	263
8.4.8	Kerberos	263
8.4.9	Infra-estruturas de autenticação	266
8.5	Controle de acesso	267
8.5.1	Políticas, modelos e mecanismos de controle de acesso	268
8.5.2	Políticas discricionárias	269
8.5.3	Políticas obrigatórias	274
8.5.4	Políticas baseadas em domínios e tipos	276
8.5.5	Políticas baseadas em papéis	278
8.5.6	Mecanismos de controle de acesso	279
8.5.7	Mudança de privilégios	286
8.6	Auditória	289
8.6.1	Coleta de dados	289
8.6.2	Análise de dados	292
8.6.3	Auditória preventiva	293
9	Virtualização de sistemas	295
9.1	Conceitos básicos	295
9.1.1	Um breve histórico	295
9.1.2	Interfaces de sistema	296
9.1.3	Compatibilidade entre interfaces	297
9.1.4	Virtualização de interfaces	299
9.1.5	Virtualização versus abstração	301
9.2	A construção de máquinas virtuais	303
9.2.1	Definição formal	303
9.2.2	Suporte de hardware	305
9.2.3	Formas de virtualização	308
9.3	Tipos de máquinas virtuais	309
9.3.1	Máquinas virtuais de processo	310
9.3.2	Máquinas virtuais de sistema operacional	313
9.3.3	Máquinas virtuais de sistema	315
9.4	Técnicas de virtualização	317
9.4.1	Emulação completa	317
9.4.2	Virtualização da interface de sistema	318
9.4.3	Tradução dinâmica	319
9.4.4	Paravirtualização	320
9.4.5	Aspectos de desempenho	321
9.5	Aplicações da virtualização	323
9.6	Ambientes de máquinas virtuais	325
9.6.1	VMware	325
9.6.2	FreeBSD Jails	326
9.6.3	Xen	327
9.6.4	User-Mode Linux	329

9.6.5	QEMU	330
9.6.6	Valgrind	330
9.6.7	JVM	331
Referências Bibliográficas		332
A O <i>Task Control Block</i> do Linux		342

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 multi-facetada e complexa, denominada genericamente de Sistema Operacional. 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

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 dores de cabeça para o desenvolvedor de aplicativos. Portanto, torna-se desejável oferecer aos programas aplicativos uma forma de acesso homogênea aos dispositivos físicos, que permita abstrair 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. O sistema operacional é 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 arquitetura geral de um sistema de computação típico. Nela, podemos observar elementos de hardware, o sistema operacional e alguns programas aplicativos.

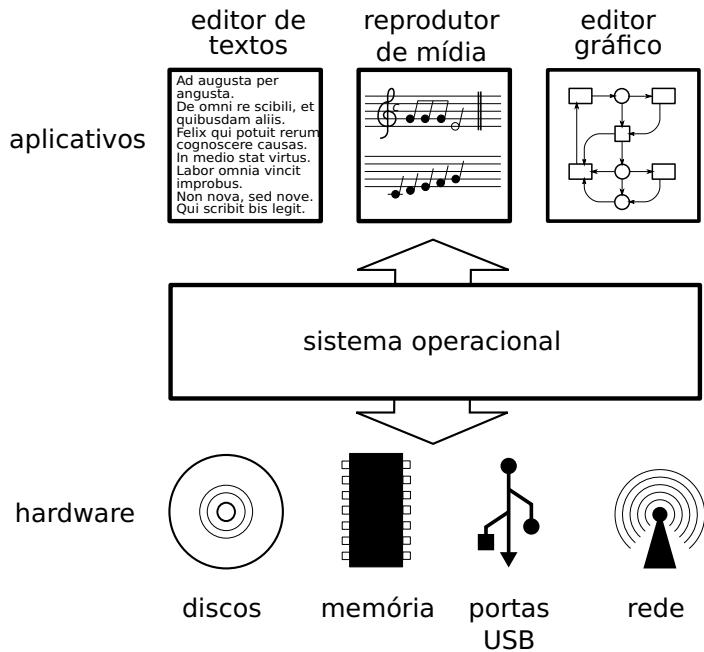


Figura 1.1: Estrutura de um sistema de computação típico

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

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 leitor de disquete:

1. verificar se os parâmetros informados estão corretos (nome do arquivo, identificador do leitor de disquete, buffer de leitura, etc.);
2. verificar se o leitor de disquetes está disponível;
3. verificar se o leitor contém um disquete;
4. ligar o motor do leitor e aguardar atingir a velocidade de rotação correta;
5. posicionar a cabeça de leitura sobre a trilha onde está a tabela de diretório;
6. ler a tabela de diretório e localizar o arquivo ou subdiretório desejado;
7. mover a cabeça de leitura para a posição do bloco inicial do arquivo;
8. 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, uma aplicação usa um conceito chamado *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, teria de 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 rígidos 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 IDE, 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 a 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 Tipos de sistemas operacionais

Os sistemas operacionais podem ser classificados segundo diversos parâmetros e perspectivas, como tamanho, 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 em bancos de dados. 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 desses sistemas incluem o OS/360 e 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 máquina estão disponíveis globalmente, de forma transparente aos usuários. Ao lançar uma aplicação, o usuário interage com sua janela, mas não sabe onde ela está executando ou armazenando seus arquivos: o sistema é quem decide, de forma transparente. Os sistemas operacionais distribuídos já existem há tempos (Amoeba [Tanenbaum et al., 1991] e Clouds [Dasgupta et al., 1991], por exemplo), mas ainda não são uma realidade de mercado.

Multi-usuário : um sistema operacional multi-usuário 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 multi-usuários.

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 mídias simples. 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, etc.), o MacOS X e Linux.

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 multi-usuários.

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 telefones celulares, sistemas de automação industrial e controladores automotivos, equipamentos eletrônicos de uso doméstico (leitores de DVD, TVs, fornos-micro-ondas, centrais de alarme, etc.). Muitas vezes um sistema operacional embarcado se apresenta na forma de uma biblioteca a ser ligada ao programa da aplicação (que é fixa). LynxOS, µC/OS, Xilinx e VxWorks são exemplos de sistemas operacionais embarcados para controle e automação. Sistemas operacionais para telefones celulares inteligentes (*smartphones*) incluem o Symbian e o Android, entre outros.

Tempo real : ao contrário da concepção usual, um sistema operacional de tempo real não precisa ser necessariamente ultra-rápido; sua característica essencial é ter um comportamento temporal previsível (ou seja, seu tempo de resposta deve ser conhecido no melhor e 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.

Existem duas classificações de sistemas de tempo real: *soft real-time systems*, nos quais a perda de prazos implica na degradação do serviço prestado. Um exemplo seria o suporte à gravação de CDs ou à reprodução de músicas. Caso o sistema se atrasasse, pode ocorrer a perda da mídia em gravação ou falhas na música que está sendo tocada. Por outro lado, nos *hard real-time systems* a perda de prazos pelo sistema pode perturbar o objeto controlado, com graves consequências humanas, econômicas ou ambientais. Exemplos desse tipo de sistema seriam o controle de funcionamento de uma turbina de avião a jato ou de uma caldeira industrial.

Exemplos de sistemas de tempo real incluem o QNX, RT-Linux e VxWorks. Muitos sistemas embarcados têm características de tempo real, e vice-versa.

1.3 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 : também conhecida como gerência de processos ou de atividades, esta funcionalidade visa distribuir a capacidade de processamento de forma justa¹ entre as aplicações, evitando que uma aplicação monopolize esse recurso e respeitando as prioridades dos 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 inter-dependentes 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 núcleo do sistema. 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 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 peculiaridades; assim, o procedimento de interação com uma placa de rede é completamente diferente da interação com um disco rígido SCSI. 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 *pen-drives*, discos SCSI ou IDE, disquetes, etc., na forma de um vetor de blocos de dados. A função da gerência de dispositivos (também conhecida como *gerência de*

¹Distribuir de forma justa, mas não necessariamente igual, pois as aplicações têm demandas de processamento distintas; 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.

entrada/saída) é implementar a interação com cada dispositivo por meio de *drivers* e criar modelos abstratos que permitam agrupar vários dispositivos distintos 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 do núcleo do sistema (como o diretório /proc do UNIX). No sistema operacional experimental *Plan 9* [Pike et al., 1993], 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, gerência de energia, etc.

As funcionalidades do sistema operacional geralmente são inter-dependentes: 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. Alguns autores [Silberschatz et al., 2001, Tanenbaum, 2003] representam a estrutura do sistema operacional conforme indicado na Figura 1.2. Nela, o núcleo central implementa o acesso de baixo nível ao hardware, enquanto os módulos externos representam as várias funcionalidades do sistema.

Uma regra importante a ser observada na construção de um sistema operacional é a separação entre os conceitos de política e mecanismo². 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.

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

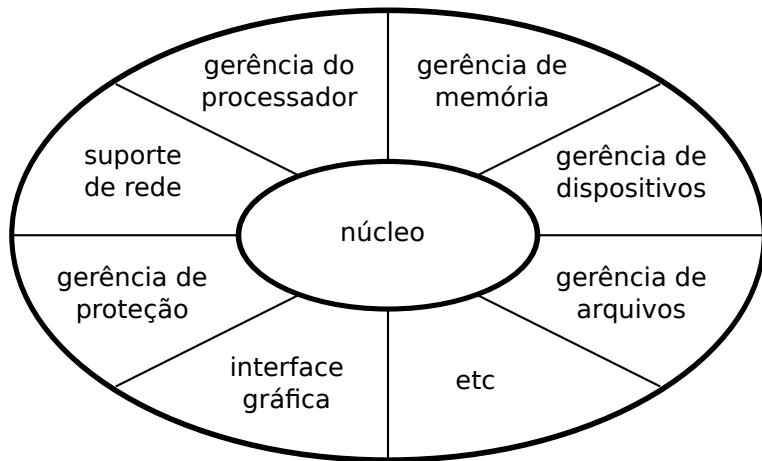


Figura 1.2: Funcionalidades do sistema operacional

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

1.4 Estrutura de um 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 programas aplicativos.

Drivers : módulos de código específicos para acessar os dispositivos físicos. Existe um driver para cada tipo de dispositivo, como discos rígidos IDE, SCSI, portas USB, placas de vídeo, 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.

Código de inicialização : 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.

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 1.3. A forma como esses diversos componentes são interligados e se relacionam varia de sistema para sistema; algumas possibilidades são discutidas na Seção 1.6.

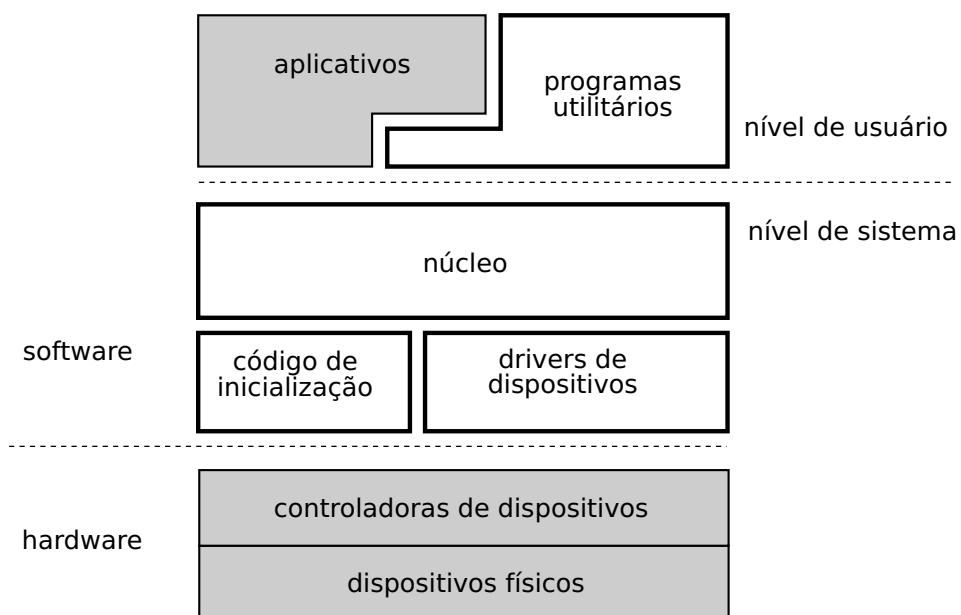


Figura 1.3: Estrutura de um sistema operacional

1.5 Conceitos de hardware

O sistema operacional interage diretamente com o hardware para fornecer serviços às aplicações. Para a compreensão dos conceitos implementados pelos sistemas operacionais, é necessário ter uma visão clara dos recursos fornecidos pelo hardware e a forma de acessá-los. Esta seção apresenta uma revisão dos principais aspectos do hardware de um computador pessoal convencional.

Um sistema de computação típico é constituído de um ou mais processadores, responsáveis pela execução das instruções das aplicações, uma área de memória que armazena 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 rígidos, terminais e teclados. A maioria dos computadores mono-processados

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 1.4 ilustra a arquitetura de um computador típico.

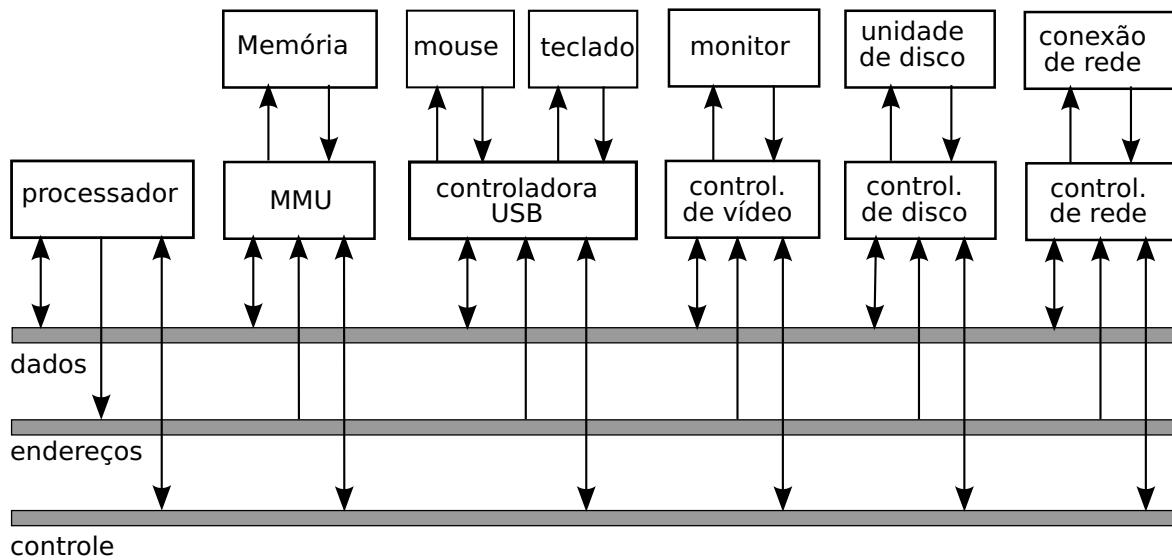


Figura 1.4: Arquitetura de um computador típico

O núcleo do sistema de computação é o processador. Ele é responsável por continuamente ler instruções e dados da memória ou de periféricos, processá-los e enviar os resultados de volta à memória ou a outros periféricos. Um processador convencional é normalmente 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.).

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 indicada entre o processador e a memória ou um controlador de dispositivo.

O acesso à memória é geralmente mediado por um controlador específico (que pode estar fisicamente dentro do próprio processador): a **Unidade de Gerência de Memória** (MMU - *Memory Management Unit*). Ela é responsável por analisar cada endereço solicitado pelo processador, validá-lo, efetuar as conversões de endereçamento necessárias e executar a operação solicitada pelo processador (leitura ou escrita de uma posição de memória).

Os periféricos do computador (discos, teclado, monitor, etc.) são acessados através de circuitos específicos genericamente 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. Para o processador, 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 1.1 a seguir apresenta alguns endereços portas de entrada/saída para acessar controladores em um PC típico:

dispositivo	endereços de acesso
teclado	0060h-006Fh
barramento IDE primário	0170h-0177h
barramento IDE secundário	01F0h-01F7Fh
porta serial COM1	02F8h-02FFh
porta serial COM2	03F8h-03FFh

Tabela 1.1: Endereços de acesso a dispositivos

1.5.1 Interrupções

Quando um controlador de periférico tem uma informação importante a fornecer ao processador, ele tem duas alternativas de comunicação:

- 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 através do barramento de controle, enviando a ele uma *requisição de interrupção* (IRQ – *Interrupt ReQuest*).

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 1.5 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);
2. Um pacote vindo da rede é recebido pela placa Ethernet;
3. A placa 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 receber as informações da placa de rede (via barramentos de dados e de endereços) e atualizar as estruturas de dados do sistema operacional;

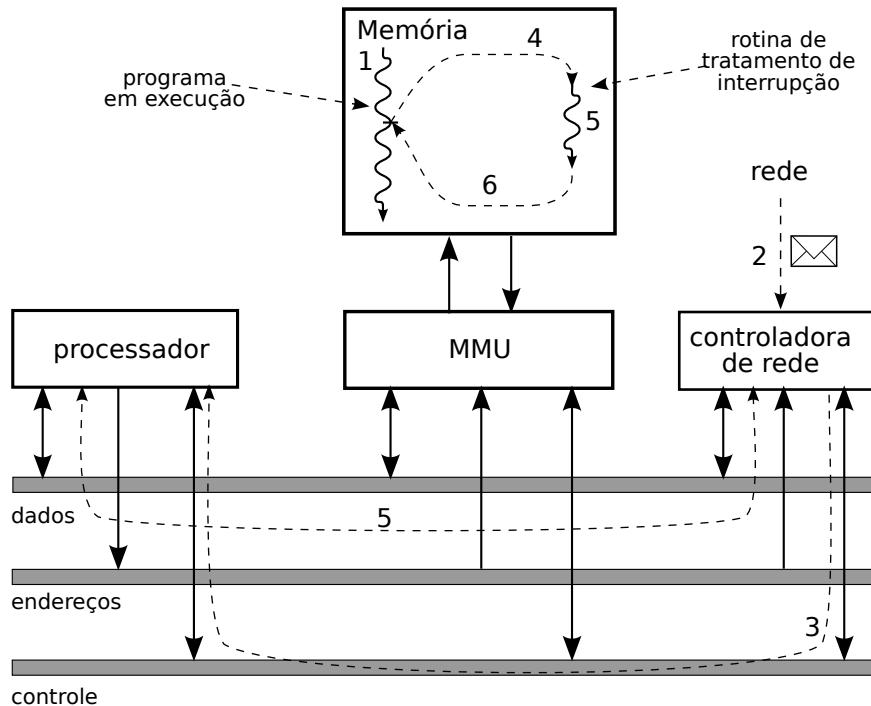


Figura 1.5: Roteiro típico de um tratamento de interrupção

6. A rotina de tratamento da interrupção é finalizada e o processador retorna à execução do programa que havia sido interrompido.

Esse roteiro de ações ocorre a cada requisição de interrupção recebida pelo processador. Cada interrupção geralmente corresponde a um evento ocorrido em um dispositivo periférico: a chegada de um pacote de rede, um click no mouse, uma operação concluída pelo controlador de disco, etc. Isso representa centenas ou mesmo milhares de interrupções recebidas por segundo, dependendo da carga e da configuração do sistema (número e natureza dos periféricos). Por isso, as rotinas de tratamento de interrupção devem ser curtas e realizar suas tarefas rapidamente (para não prejudicar o desempenho do sistema).

Normalmente o processador recebe e trata cada interrupção recebida, mas nem sempre isso é possível. Por exemplo, receber e tratar uma interrupção pode ser problemático caso o processador já esteja tratando outra interrupção. Por essa razão, o processador pode decidir ignorar temporariamente algumas interrupções, se necessário. Isso é feito ajustando o bit correspondente à interrupção em um registrador específico do processador.

Para distinguir interrupções geradas por dispositivos distintos, cada interrupção é identificada por um inteiro, normalmente com 8 bits. 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 um vetor de endereços de funções denominado *Vetor de Interrupções* (*IV - Interrupt Vector*); cada entrada desse vetor aponta para a rotina de tratamento da interrupção correspondente. Por exemplo, se a entrada 5 do vetor contém o valor 3C20h,

então a rotina de tratamento da IRQ 5 iniciará na posição 3C20h da memória RAM. O vetor 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.

As 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*. Eventos como instruções ilegais (inexistentes ou com operandos inválidos), tentativa 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 o mesmo vetor de endereços de funções). A Tabela 1.2 representa o vetor de interrupções do processador Intel Pentium (extraída de [Patterson and Henessy, 2005]).

Tabela 1.2: Vetor 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. Interrupções não são raras, pelo contrário: em um computador pessoal, o processador trata de centenas a milhares de interrupções por segundo, dependendo da carga do sistema e dos periféricos instalados.

1.5.2 Proteção do núcleo

Um sistema operacional deve gerenciar os recursos do hardware, fornecendo-os às aplicações conforme suas necessidades. Para assegurar a integridade dessa gerência, é essencial garantir que as aplicações não consigam acessar o hardware diretamente, mas sempre através de pedidos ao sistema operacional, que avalia e intermedeia todos os acessos ao hardware. Mas como impedir as aplicações de acessar o hardware diretamente?

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 utilitários e os aplicativos devem ter acesso mais restrito a ele, para não interferir nas configurações e na gerência, o que acabaria desestabilizando o sistema inteiro. Além disso, aplicações com acesso pleno ao hardware tornariam inúteis os mecanismos de segurança e controle de acesso aos recursos (tais como arquivos, diretórios e áreas de memória).

Para permitir diferenciar os privilégios de execução dos diferentes tipos de software, os processadores modernos contam com dois ou mais *níveis de privilégio de execução*. Esses níveis são controlados por flags especiais nos processadores, e as formas de mudança de um nível de execução para outro são controladas estritamente pelo processador. O processador Pentium, por exemplo, conta com 4 níveis de privilégio (sendo 0 o nível mais privilegiado), embora a maioria dos sistemas operacionais construídos para esse processador só use os níveis extremos (0 para o núcleo e drivers do sistema operacional e 3 para utilitários e aplicações). 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, todo o processador está acessível: todos os recursos internos do processador (registradores e portas de entrada/saída) e áreas de memória podem ser acessados. Além disso, todas as instruções do processador podem ser executadas. Ao ser ligado, o processador entra em operação neste nível.

Nível usuário (ou *userspace*): neste nível, somente um sub-conjunto das instruções do processador, registradores e portas de entrada/saída estão disponíveis. Instruções “perigosas” como HALT (parar o processador) e RESET (reiniciar o processador) são proibidas para todo código executando neste nível. Além disso, o hardware restringe o uso da memória, permitindo o acesso somente a áreas previamente definidas. Caso o código em execução tente executar uma instrução proibida ou acessar uma área de memória inacessível, o hardware irá gerar uma exceção, desviando a execução para uma rotina de tratamento dentro do núcleo, que

provavelmente irá abortar o programa em execução (e também gerar a famosa frase “este programa executou uma instrução ilegal e será finalizado”, no caso do Windows).

É fácil perceber que, em um sistema operacional convencional, o núcleo e os drivers operam no nível núcleo, enquanto os utilitários e as aplicações operam no nível usuário, confinados em áreas de memória distintas, conforme ilustrado na Figura 1.6. Todavia, essa separação nem sempre segue uma regra tão simples; outras opções de organização de sistemas operacionais serão abordadas na Seção 1.6.

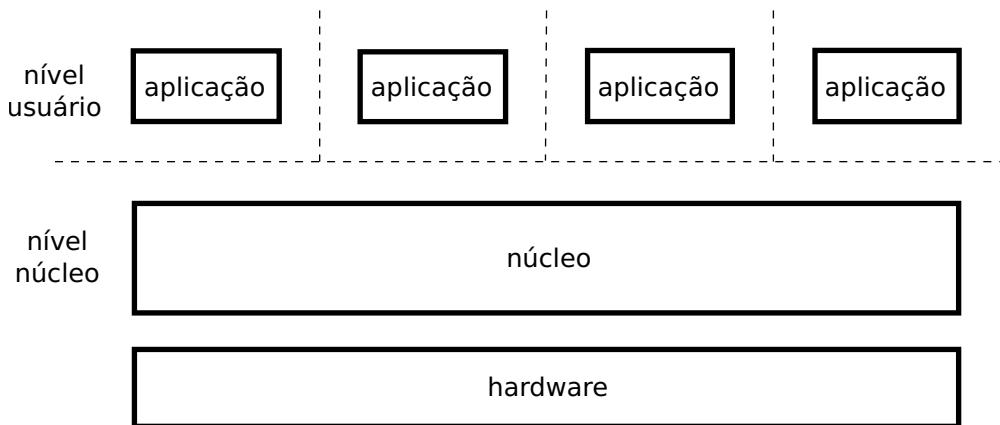


Figura 1.6: Separação entre o núcleo e as aplicações

1.5.3 Chamadas de sistema

O confinamento de cada aplicação em sua área de memória, imposto pelos mapeamentos de memória realizados pela MMU nos acessos 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 chamar, a partir de uma aplicação, as rotinas oferecidas pelo núcleo para o acesso ao hardware e suas abstrações? Em outras palavras, como uma aplicação pode acessar a placa de rede para enviar/receber dados, se não tem privilégio para acessar as portas de entrada/saída correspondentes nem pode invocar o código do núcleo que implementa esse acesso (pois esse código reside em outra área de memória)?

A resposta a esse problema está no mecanismo de interrupção, apresentado na Seção 1.5.1. Os processadores implementam uma instrução especial que permite acionar o mecanismo de interrupção de forma intencional, sem depender de eventos externos ou internos. Ao ser executada, essa instrução (int no Pentium, syscall no MIPS) comuta o processador para o nível privilegiado e procede de forma similar ao tratamento de uma interrupção. Por essa razão, esse mecanismo é denominado *interrupção de software, ou trap*. Processadores modernos oferecem instruções específicas para entrar/sair do modo privilegiado, como SYSCALL e SYSRET (nos processadores Pentium 64 bits) e também um conjunto de registradores específicos para essa operação, o que permite a

transferência rápida do controle para o núcleo, com custo menor que o tratamento de uma interrupção.

A ativação de procedimentos do núcleo usando interrupções de software (ou outros mecanismos correlatos) 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 finalização de tarefas, operadores de sincronização e comunicação, 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 interrupção de software e retorna à aplicação os resultados obtidos.

A Figura 1.7 ilustra o funcionamento básico de uma chamada de sistema (a chamada `read`, que lê dados de um arquivo previamente aberto). Os seguintes passos são realizados:

1. No nível usuário, a aplicação invoca a função `read(fd, &buffer, bytes)` da biblioteca de sistema (no Linux é a biblioteca *GNUC Library*, ou *glibc*; no Windows, essas funções são implementadas pela API Win32).
2. A função `read` preenche uma área de memória com os parâmetros recebidos e escreve o endereço dessa área em um registrador da CPU. Em outro registrador, ela escreve o código da chamada de sistema desejada (no caso do Linux, seria `03h` para a *syscall read*).
3. A função `read` invoca uma interrupção de software (no caso do Linux, sempre é invocada a interrupção `80h`).
4. O processador comuta para o nível privilegiado (*kernel level*) e transfere o controle para a rotina apontada pela entrada `80h` do vetor de interrupções.
5. A rotina obtém o endereço dos parâmetros, verifica a validade de cada um deles e realiza (ou agenda para execução posterior) a operação desejada pela aplicação.
6. Ao final da execução da rotina, eventuais valores de retorno são escritos na área de memória da aplicação e o processamento retorna à função `read`, em modo usuário.
7. A função `read` finaliza sua execução e retorna o controle à aplicação.
8. Caso a operação solicitada não possa ser realizada imediatamente, a rotina de tratamento da interrupção de software passa o controle para a gerência de atividades, ao invés de retornar diretamente da interrupção de software para a aplicação solicitante. Isto ocorre, por exemplo, quando é solicitada a leitura de uma entrada do teclado.
9. Na sequência, a gerência de atividades devolve o controle do processador a outra aplicação que também esteja aguardando o retorno de uma interrupção de software, e cuja operação solicitada já tenha sido concluída.

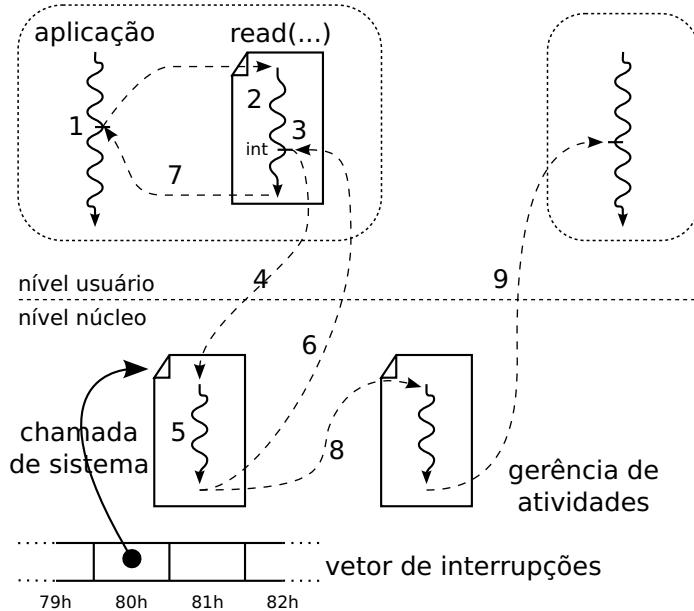


Figura 1.7: Roteiro típico de uma chamada de sistema

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*) desse 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.

1.6 Arquiteturas de Sistemas Operacionais

Embora a definição de níveis de privilégio (Seção 1.5.3) imponha uma estruturação mínima a um sistema operacional, as várias partes que compõem o sistema podem ser organizadas de diversas formas, separando suas funcionalidades e modularizando seu projeto. Nesta seção serão apresentadas as arquiteturas mais populares para a organização de sistemas operacionais.

1.6.1 Sistemas monolíticos

Em um sistema monolítico, todos os componentes do núcleo operam em modo núcleo e se inter-relacionam conforme suas necessidades, sem restrições de acesso entre si, pois o código no nível núcleo tem acesso pleno a todos os recursos e áreas de memória. A Figura 1.8 ilustra essa arquitetura.

A grande vantagem dessa arquitetura é seu desempenho: qualquer componente do núcleo pode acessar os demais componentes, toda a memória ou mesmo dispositivos periféricos diretamente, pois não há barreiras impedindo esse acesso. A interação direta entre componentes também leva a sistemas mais compactos.

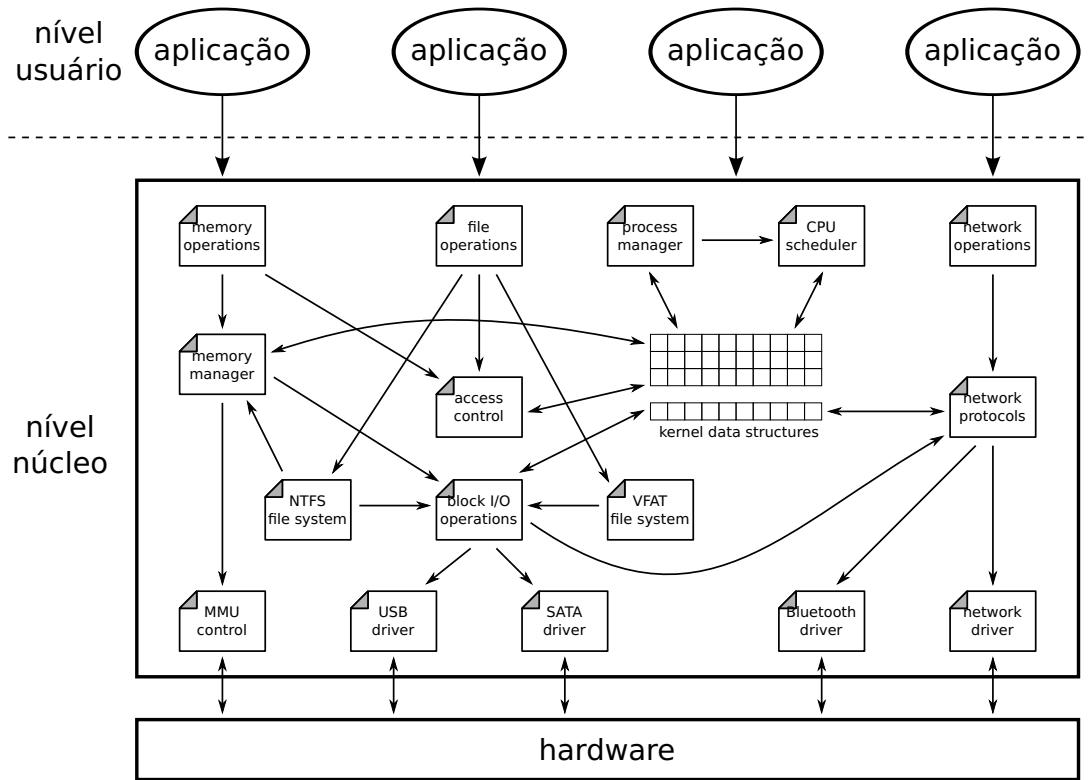


Figura 1.8: Uma arquitetura monolítica

Todavia, a arquitetura monolítica pode pagar um preço elevado por seu desempenho: a robustez e a facilidade de desenvolvimento. Caso um componente do núcleo perca o controle devido a algum erro, esse problema pode se alastrar rapidamente por todo o núcleo, levando o sistema ao colapso (travamento, reinicialização ou funcionamento errático). Além disso, a manutenção e evolução do núcleo se tornam mais complexas, porque as dependências e pontos de interação entre os componentes podem não ser evidentes: pequenas alterações na estrutura de dados de um componente podem ter um impacto inesperado em outros componentes, caso estes acessem aquela estrutura diretamente.

A arquitetura monolítica foi a primeira forma de organizar os sistemas operacionais; sistemas UNIX antigos e o MS-DOS seguiam esse modelo. Atualmente, apenas sistemas operacionais embarcados são totalmente monolíticos, devido às limitações do hardware sobre o qual executam. O núcleo do Linux nasceu monolítico, mas vem sendo paulatinamente estruturado e modularizado desde a versão 2.0 (embora boa parte de seu código ainda permaneça no nível de núcleo).

1.6.2 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 provêem 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). Essa abordagem de estruturação de software fez muito 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 sua aceitação nesse contexto:

- 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 do sistema.
- Não é óbvio como dividir as funcionalidades de um núcleo de sistema operacional em camadas horizontais de abstração crescente, pois essas funcionalidades são inter-dependentes, embora tratem muitas vezes de recursos distintos.

Em decorrência desses inconvenientes, a estruturação em camadas é apenas parcialmente adotada hoje em dia. Muitos sistemas 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 sub-sistemas como a gerência de arquivos e o suporte de rede (segundo o modelo OSI). Como exemplos de sistemas fortemente estruturados em camadas podem ser citados o IBM OS/2 e o MULTICS [Corbató and Vyssotsky, 1965].

1.6.3 Sistemas micro-núcleo

Uma outra possibilidade de estruturação consiste em retirar do núcleo todo o código de alto nível (normalmente associado às políticas de gerência de recursos), deixando no núcleo somente o código de baixo nível necessário para interagir com o hardware e criar as abstrações fundamentais (como a noção de atividade). Por exemplo, usando essa abordagem o código de acesso aos blocos de um disco rígido seria mantido no núcleo, enquanto as abstrações de arquivo e diretório seriam criadas e mantidas por um código fora do núcleo, executando da mesma forma que uma aplicação do usuário.

Por fazer os núcleos de sistema ficarem menores, essa abordagem foi denominada *micro-núcleo* (ou *μ-kernel*). Um micro-núcleo normalmente implementa somente a noção de atividade, de espaços de memória protegidos e de comunicação entre atividades. Todos os aspectos de alto nível, como políticas de uso do processador e da memória, o sistema de arquivos e o controle de acesso aos recursos são implementados fora do núcleo, em processos que se comunicam usando as primitivas do núcleo. A Figura 1.9 ilustra essa abordagem.

Em um sistema micro-núcleo, as interações entre componentes e aplicações são feitas através de trocas de mensagens. Assim, se uma aplicação deseja abrir um arquivo no disco rígido, envia uma mensagem para o gerente de arquivos que, por sua vez, se comunica com o gerente de dispositivos para obter os blocos de dados relativos ao arquivo desejado. Os processos não podem se comunicar diretamente, devido às restrições impostas pelos mecanismos de proteção do hardware. Por isso, todas as mensagens são transmitidas através de serviços do micro-núcleo, como mostra a Figura

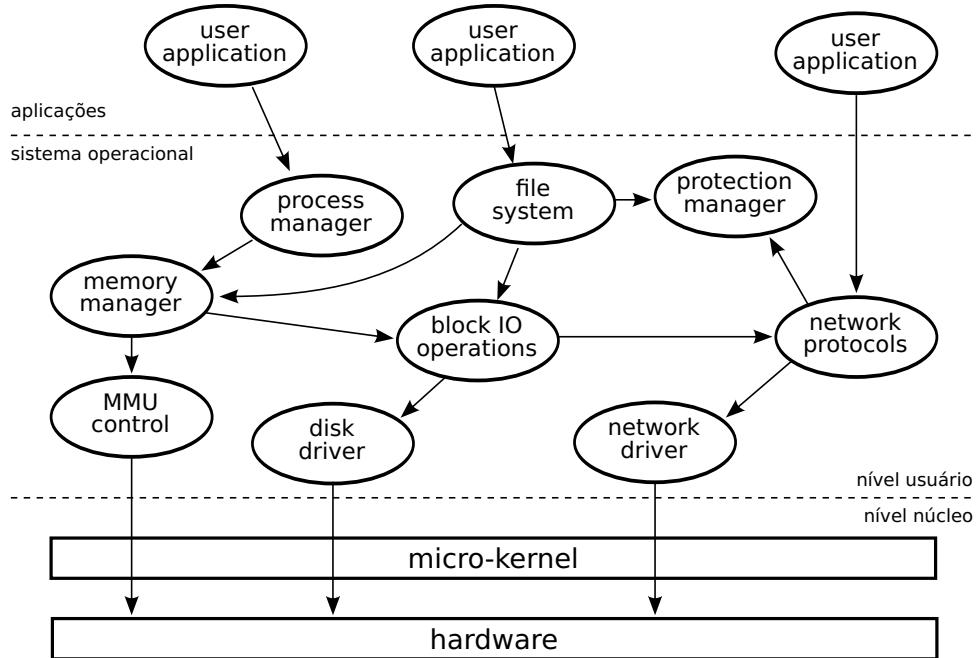


Figura 1.9: Visão geral de uma arquitetura micro-núcleo

1.9. Como os processos têm de solicitar “serviços” uns dos outros, para poder realizar suas tarefas, essa abordagem também foi denominada *cliente-servidor*.

O micro-núcleos foram muito investigados durante os anos 80. Dois exemplos clássicos dessa abordagem são os sistemas Mach [Rashid et al., 1989] e Chorus [Rozier et al., 1992]. As principais vantagens dos sistemas micro-núcleo são sua robustez e flexibilidade: caso um sub-sistema tenha problemas, os mecanismos de proteção de memória e níveis de privilégio irão confiná-lo, impedindo que a instabilidade se alastre ao restante do sistema. Além disso, é possível customizar o sistema operacional, iniciando somente os componentes necessários ou escolhendo os componentes mais adequados às aplicações que serão executadas.

Vários sistemas operacionais atuais adotam parcialmente essa estruturação; por exemplo, o MacOS X da Apple tem suas raízes no sistema Mach, ocorrendo o mesmo com o Digital UNIX. Todavia, o custo associado às trocas de mensagens entre componentes pode ser bastante elevado, o que prejudica seu desempenho e diminui a aceitação desta abordagem. O QNX é um dos poucos exemplos de micro-núcleo amplamente utilizado, sobretudo em sistemas embarcados e de tempo-real.

1.6.4 Máquinas virtuais

Para que programas e bibliotecas possam executar sobre uma determinada plataforma computacional, é necessário que tenham sido compilados para ela, respeitando o conjunto de instruções do processador e o conjunto de chamadas do sistema operacional. Da mesma forma, um sistema operacional só poderá executar sobre uma plataforma de hardware se for compatível com ela. Nos sistemas atuais, as interfaces de baixo nível são pouco flexíveis: geralmente não é possível criar novas instruções de processador ou

novas chamadas de sistema, ou mudar sua semântica. Por isso, um sistema operacional só funciona sobre o hardware para o qual foi construído, uma biblioteca só funciona sobre o hardware e sistema operacional para os quais foi projetada e as aplicações também têm de obedecer a interfaces pré-definidas.

Todavia, é possível contornar os problemas de compatibilidade entre os componentes de um sistema através de técnicas de *virtualização*. Usando os serviços oferecidos por um determinado componente do sistema, é possível construir uma camada de software que ofereça aos demais componentes serviços com outra interface. Essa camada permitirá assim o acoplamento entre interfaces distintas, de forma que um programa desenvolvido para uma plataforma *A* possa executar sobre uma plataforma distinta *B*. O sistema computacional visto através dessa camada é denominado *máquina virtual*.

A Figura 1.10, extraída de [Smith and Nair, 2004], mostra um exemplo de máquina virtual, onde uma camada de virtualização permite executar um sistema operacional Windows e suas aplicações sobre uma plataforma de hardware Sparc, distinta daquela para a qual foi projetado (Intel/AMD).

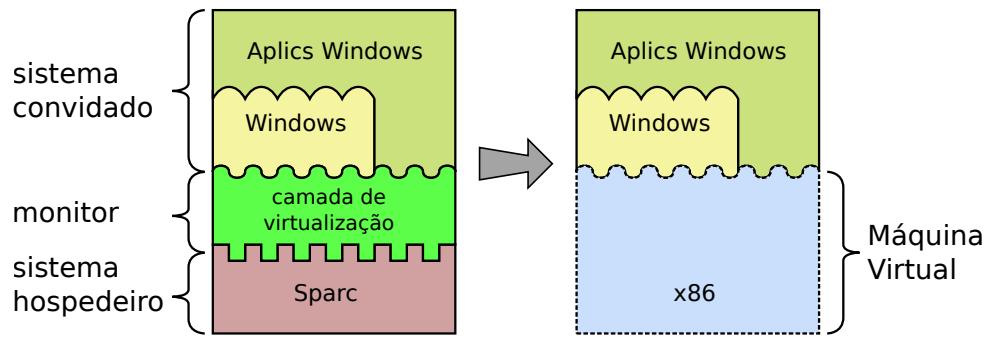


Figura 1.10: Uma máquina virtual [Smith and Nair, 2004].

Um ambiente de máquina virtual consiste de três partes básicas, que podem ser observadas na Figura 1.10:

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

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]. 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 são compilados para execução sobre um hardware abstrato denominado *P-Machine*. Com o aumento de desempenho e funcionalidades do hardware PC e o surgimento da linguagem Java, no início dos anos 90, o interesse pelas tecnologias de virtualização voltou à tona. Atualmente, as soluções de virtualização de linguagens e de plataformas vêm despertando grande interesse do mercado. Várias linguagens são compiladas para máquinas virtuais portáveis e os processadores mais recentes trazem um suporte nativo à virtualização de hardware, finalmente respeitando as condições conceituais definidas no início dos anos 70.

Existem diversas possibilidades de implementação de sistemas de máquinas virtuais. De acordo com o tipo de sistema convidado suportado, os ambientes de máquinas virtuais podem ser divididos em duas grandes famílias (Figura 1.11):

Máquinas virtuais de aplicação : são ambientes de máquinas virtuais destinados a suportar apenas um processo ou aplicação convidada específica. A máquina virtual Java é um exemplo desse tipo de ambiente.

Máquinas virtuais de sistema : são construídos para suportar sistemas operacionais convidados completos, com aplicações convidadas executando sobre eles. Como exemplos podem ser citados os ambientes *VMWare*, *Xen* e *VirtualBox*.

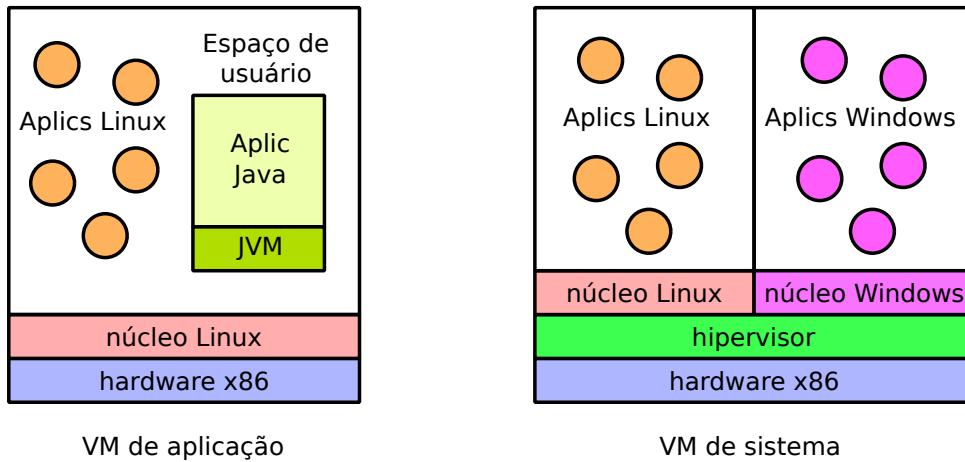


Figura 1.11: Máquinas virtuais de aplicação e de sistema.

As máquinas virtuais de aplicação são geralmente usadas como suporte de execução de linguagens de programação. As primeiras experiências com linguagens usando máquinas virtuais remontam aos anos 1970, com a linguagem *UCSD Pascal* (da Universidade da Califórnia em San Diego). Na época, um programa escrito em Pascal era compilado em um código binário denominado *P-Code*, que executava sobre o processador abstrato *P-Machine*. O interpretador de *P-Codes* era bastante compacto e facilmente portável, o que tornou o sistema P muito popular. 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.

Máquinas virtuais de sistema suportam um ou mais sistemas operacionais convidados, com suas respectivas aplicações, que executam de forma isolada e independente. Em uma máquina virtual, cada sistema operacional convidado tem a ilusão de executar sozinho sobre uma plataforma de hardware exclusiva. Como o sistema operacional convidado e o ambiente de execução dentro da máquina virtual são idênticos ao da máquina real, é possível usar os softwares já construídos para a máquina real dentro das máquinas virtuais. Essa transparência evita ter de construir novas aplicações ou adaptar as já existentes.

As máquinas virtuais de sistema constituem a primeira abordagem usada para a construção de hipervisores, desenvolvida na década de 1960. No que diz respeito à sua arquitetura, existem basicamente dois tipos de hipervisores de sistema, apresentados na Figura 1.12:

Hipervisores nativos (ou de tipo I): o hipervisor executa diretamente sobre o hardware da máquina real, sem um sistema operacional subjacente. A função do hipervisor é multiplexar os recursos de hardware (memória, discos, interfaces de rede, etc.) de forma que cada máquina virtual veja um conjunto de recursos próprio e independente. Alguns exemplos de sistemas que empregam esta abordagem são o *IBM OS/370*, o *VMWare ESX Server* e o ambiente *Xen*.

Hipervisores convidados (ou de tipo II): o hipervisor executa como um processo normal sobre um sistema operacional hospedeiro. O hipervisor usa os recursos oferecidos pelo sistema operacional real para oferecer recursos virtuais ao sistema operacional convidado que executa sobre ele. Exemplos de sistemas que adotam esta estrutura incluem o *VMWare Workstation*, o *QEmu* e o *VirtualBox*.

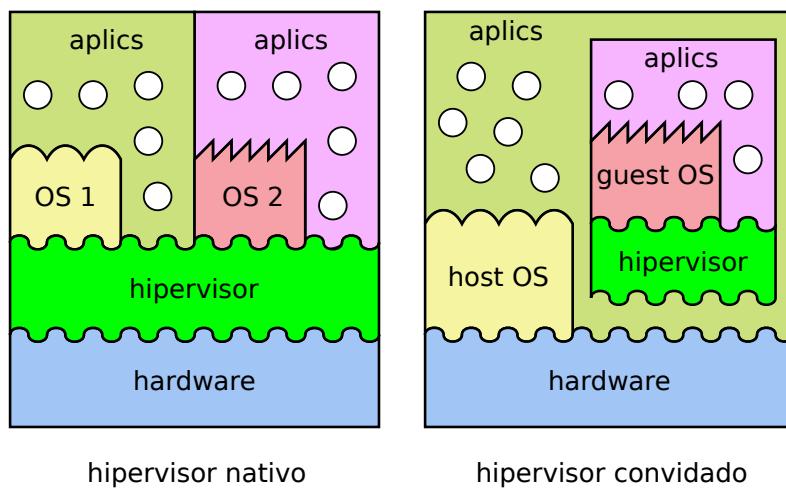


Figura 1.12: Arquiteturas de máquinas virtuais de sistema.

Os trabalhos [Goldberg, 1973, Blunden, 2002] relacionam algumas vantagens para a utilização de máquinas virtuais em sistemas de computação:

- Aperfeiçoamento e testes de novos sistemas operacionais;

- Ensino prático de sistemas operacionais e programação de baixo nível;
- Executar diferentes sistemas operacionais sobre o mesmo hardware, simultaneamente;
- Simular configurações e situações diferentes do mundo real, como por exemplo, mais memória disponível, outros dispositivos de E/S;
- Simular alterações e falhas no hardware para testes ou reconfiguração de um sistema operacional, provendo confiabilidade e escalabilidade para as aplicações;
- Garantir a portabilidade das aplicações legadas (que executariam sobre uma VM simulando o sistema operacional original);
- Desenvolvimento de novas aplicações para diversas plataformas, garantindo a portabilidade destas aplicações;
- Diminuir custos com hardware.

A principal desvantagem do uso de máquinas virtuais é o custo adicional de execução dos processos na máquina virtual em comparação com a máquina real. Esse custo é muito variável, podendo passar de 50% em plataformas sem suporte de hardware à virtualização, como os PCs de plataforma Intel mais antigos [Dike, 2000, Blunden, 2002]. Todavia, pesquisas recentes têm obtido a redução desse custo a patamares abaixo de 20%, graças sobretudo a ajustes no código do sistema hospedeiro [King et al., 2003]. Esse problema não existe em ambientes cujo hardware oferece suporte à virtualização, como é o caso dos mainframes e dos processadores Intel/AMD mais recentes.

1.7 Um breve histórico dos sistemas operacionais

Os primeiros sistemas de computação, no final dos anos 40 e início dos anos 50, não possuíam sistema operacional. 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 [Foundation, 2005]:

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 Macintosh OS 1.0, 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.
- 1993** : lançamento dos UNIX de código aberto FreeBSD e NetBSD.
- 2001** : a Apple lança o MacOS X, um sistema operacional derivado da família UNIX BSD.
- 2001** : lançamento do Windows XP.
- 2004** : lançamento do núcleo Linux 2.6.
- 2006** : lançamento do Windows Vista.

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*, *Chorus*, *QNX* e *Plan 9*, não estão representados.

Capítulo 2

Gerência de atividades

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 módulo apresenta os principais conceitos, estratégias e mecanismos empregados na gestão do processador e das atividades em execução em um sistema de computação.

2.1 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, centenas 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.

No entanto, um processador convencional somente trata um fluxo de instruções de cada vez. Até mesmo computadores com vários processadores (máquinas *Dual Pentium* ou processadores 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 ingênuia 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. Por multiplexar entendemos compartilhar o uso do processador entre as várias tarefas, de forma a atendê-las da melhor maneira possível.

Os principais conceitos abordados neste capítulo compreendem:

- Como as tarefas são definidas;
- Quais os estados possíveis de uma tarefa;
- Como e quando o processador muda de uma tarefa para outra;
- Como ordenar (escalonar) as tarefas para usar o processador.

2.2 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*:

Um 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). Por exemplo, os arquivos C:\Windows\notepad.exe e /usr/bin/nano são programas de edição de texto.

Uma tarefa é a execução, pelo processador, das sequências 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) e interage com outras entidades: o usuário, os periféricos e/ou outras tarefas. Tarefas podem ser implementadas de várias formas, como processos (Seção 2.4.3) ou *threads* (Seção 2.4.4).

Fazendo uma analogia clássica, 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 e o modo de preparo da torta. Por sua vez, a ação de “executar” a receita, providenciando os ingredientes e seguindo os passos definidos na receita, 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 variáveis internas da tarefa).

Assim como uma receita de torta pode definir várias atividades inter-dependentes para elaborar a torta (preparar a massa, fazer o recheio, decorar, etc.), um programa

também pode definir várias sequências de execução inter-dependentes para atingir seus objetivos. Por exemplo, o programa do navegador Web ilustrado na Figura 2.1 define várias tarefas que uma janela de navegador deve executar simultaneamente, para que o usuário possa navegar na Internet:

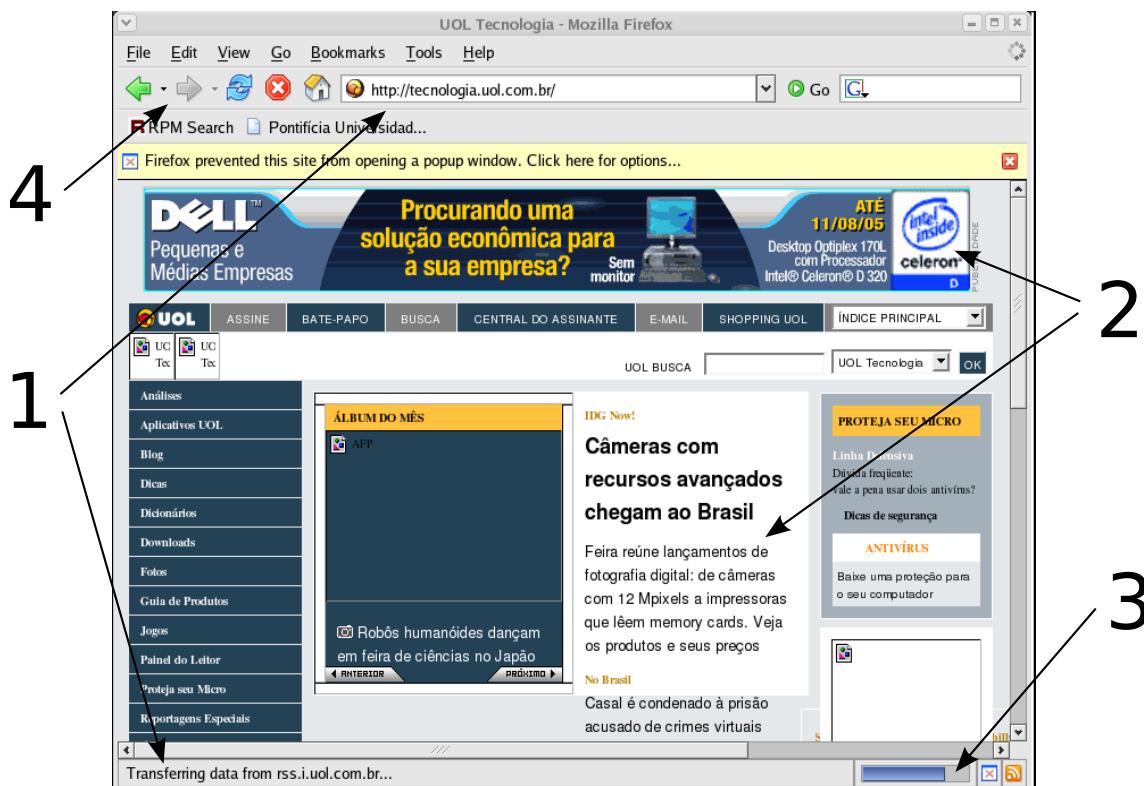


Figura 2.1: Tarefas de um navegador Internet

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

2.3 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.

2.3.1 Sistemas mono-tarefa

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 juntamente com a 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 coordenadas 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 2.2 a seguir ilustra um sistema desse tipo.

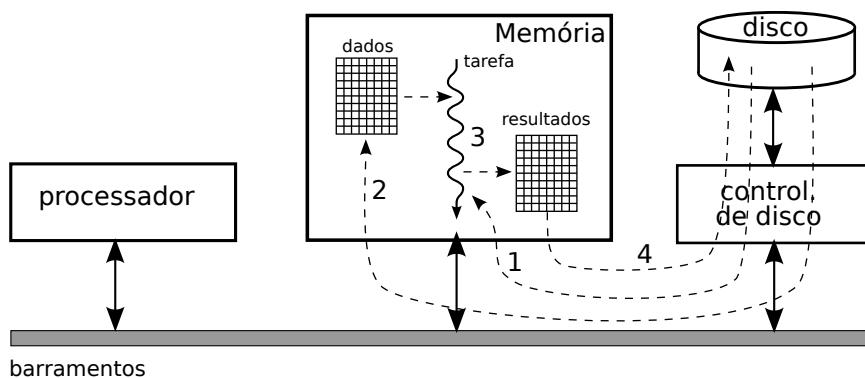


Figura 2.2: Sistema mono-tarefa: 1) carga do código na memória, 2) carga dos dados na memória, 3) processamento, consumindo dados e produzindo resultados, 4) ao término da execução, a descarga dos resultados no disco.

Nesse método de processamento de tarefas é possível delinejar um diagrama de estados para cada tarefa executada pelo sistema, que está representado na Figura 2.3.



Figura 2.3: Diagrama de estados de uma tarefa em um sistema mono-tarefa.

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 coordenar 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.

2.3.2 Sistemas multi-tarefa

O uso do programa monitor 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 saída¹, o processador ficava ocioso durante os períodos de transferência de informação entre disco e memória. Se a operação de entrada/saída envolvia fitas magnéticas, o processador podia ficar vários minutos parado, esperando. O custo dos computadores era elevado demais (e sua capacidade de processamento muito baixa) para permitir deixá-los ociosos por tanto tempo.

A solução encontrada para resolver esse problema foi permitir ao processador suspender a execução da tarefa que espera dados externos e passar a executar outra tarefa. Mais tarde, quando os dados de que necessita estiverem disponíveis, a tarefa suspensa 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 definir procedimentos para suspender uma tarefa e retomá-la mais tarde. O ato de retirar um recurso de uma tarefa (neste caso o recurso é o processador) é denominado *preempção*. Sistemas que implementam esse conceito são chamados *sistemas preemptivos*.

A adoção da preempção levou a sistemas mais produtivos (e complexos), nos quais várias tarefas podiam estar em andamento simultaneamente: uma estava ativa e as demais suspensas, esperando dados externos ou outras condições. Sistemas que suportavam essa funcionalidade foram denominados *monitores multi-tarefas*. O diagrama de estados da Figura 2.4 ilustra o comportamento de uma tarefa em um sistema desse tipo:

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

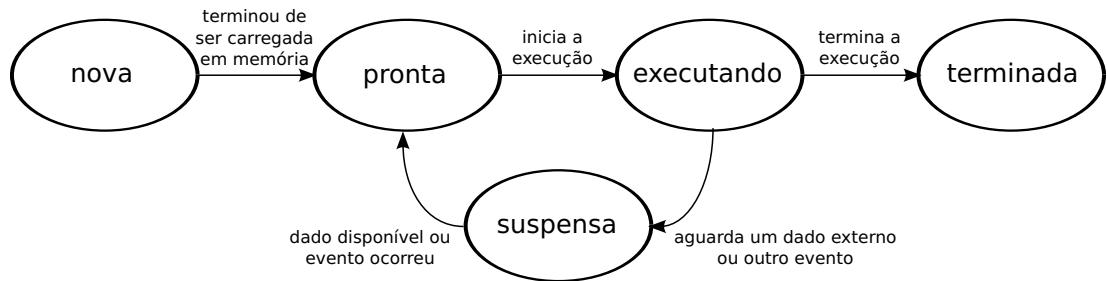


Figura 2.4: Diagrama de estados de uma tarefa em um sistema multi-tarefas.

2.3.3 Sistemas de tempo compartilhado

Solucionado o problema de evitar a ociosidade do processador, restavam no entanto vários outros problemas a resolver. Por exemplo, um programa que contém um laço infinito jamais encerra; como fazer para abortar a tarefa, ou ao menos transferir o controle ao monitor para que ele decida o que fazer? Situações como essa podem ocorrer a qualquer momento, por erros de programação ou intencionalmente, como mostra o exemplo a seguir:

```

1 void main ()
2 {
3     int i = 0, soma = 0 ;
4
5     while (i < 1000)
6         soma += i ; // erro: o contador i não foi incrementado
7
8     printf ("A soma vale %d\n", soma);
9 }
```

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, um terminal de comandos pode ser suspenso a cada leitura de teclado, perdendo o processador. Se ele 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, cada atividade que detém o processador recebe um limite de tempo de processamento, denominado *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.

Em um sistema operacional típico, a implementação da preempção por tempo tem como base as interrupções geradas pelo temporizador programável do hardware. Esse temporizador normalmente é programado para gerar interrupções em intervalos

²A duração atual 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, 2004].

regulares (a cada milissegundo, por exemplo) que são recebidas por um tratador de interrupção (*interrupt handler*); as ativações periódicas do tratador de interrupção são normalmente chamadas de *ticks* do relógio. 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*, o contador é decrementado; quando ele chegar a zero, a tarefa perde o processador e volta à fila de prontas. Essa dinâmica de execução está ilustrada na Figura 2.5.

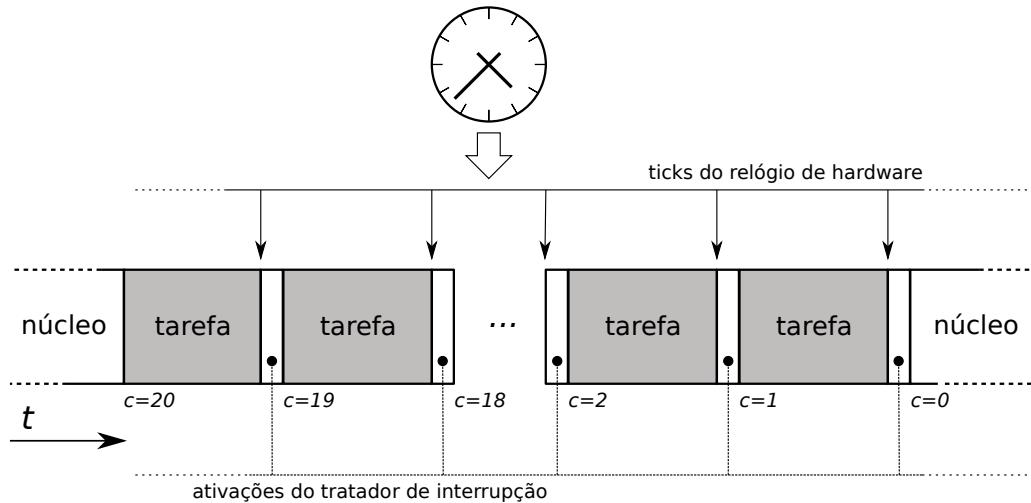


Figura 2.5: Dinâmica da preempção por tempo (*quantum* igual a 20 ticks).

O diagrama de estados das tarefas deve ser reformulado para incluir a preempção por tempo que implementa a estratégia de tempo compartilhado. A Figura 2.6 apresenta esse novo diagrama.

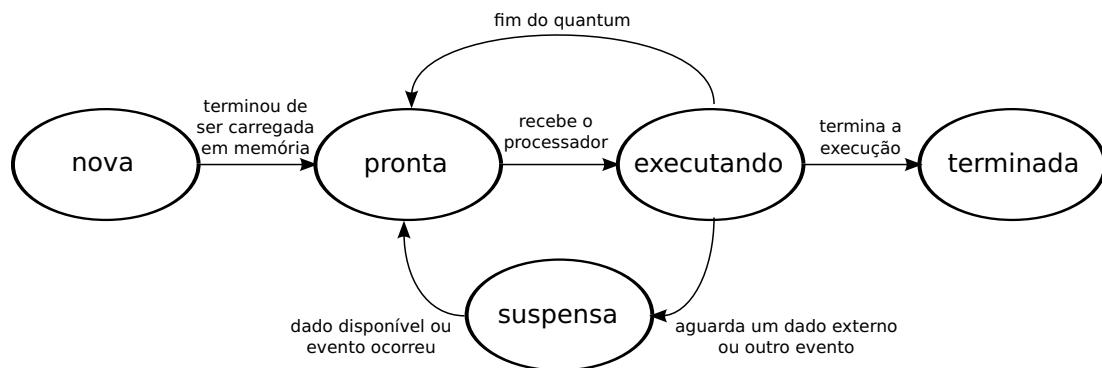


Figura 2.6: Diagrama de estados de uma tarefa em um sistema de tempo compartilhado.

2.3.4 Ciclo de vida das tarefas

O diagrama apresentado na Figura 2.6 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 executar (ou para continuar sua execução), apenas aguardando a disponibilidade do processador. Todas as tarefas prontas são organizadas em uma fila cuja ordem é determinada por algoritmos de escalonamento, que serão estudados na Seção 2.5.

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 2.6 são as *transições* entre esses estados, que são explicadas a seguir:

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

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

Pronta → **Executando** : esta transição ocorre quando a tarefa é escolhida pelo escalonador para ser executada, 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, para esperar novamente o processador.

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 apagadas.

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.

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: “blockeado”, 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 memória virtual (vide Seção 5.7). 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.

2.4 Implementação de tarefas

Conforme apresentado, uma tarefa é uma unidade básica de atividade dentro de um sistema. Tarefas podem ser implementadas de várias formas, como processos, *threads*, *jobs* e transações. Nesta seção 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 eficiente e transparente.

2.4.1 Contextos

Na Seção 2.2 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 arquivos 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.

Cada tarefa presente no sistema possui um *descritor* associado, ou seja, uma estrutura de dados que a representa no núcleo. Nessa estrutura são armazenadas as informações relativas ao seu contexto e os demais dados necessários à sua gerência. 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 identificador opaco);
- Estado da tarefa (nova, pronta, executando, suspensa, terminada, ...);

- 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 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 o conceito de TCB, o Apêndice A apresenta o TCB do núcleo Linux (versão 2.6.12), representado pela estrutura `task_struct` definida no arquivo `include/linux/sched.h`.

2.4.2 Trocas de contexto

Para que o processador possa interromper a execução de uma tarefa e retornar a ela mais tarde, sem corromper seu estado interno, é necessário definir operações para salvar e restaurar o contexto da tarefa. O ato de salvar os valores do contexto atual em seu TCB e possivelmente restaurar o contexto de outra tarefa, previamente salvo em outro TCB, é denominado **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 1.3). 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 2.5). Assim, o despachante implementa os mecanismos da gerência de tarefas, enquanto o escalonador implementa suas políticas.

A Figura 2.7 apresenta um diagrama temporal com os principais passos envolvidos em uma troca de contexto. A realização de uma troca de contexto completa, envolvendo a interrupção de uma tarefa, o salvamento de seu contexto, o escalonamento e a reativação da tarefa escolhida, é uma operação relativamente rápida (de dezenas a centenas de micro-segundos, dependendo do hardware e do sistema operacional).

A parte potencialmente mais demorada de uma troca de contexto é a execução do escalonador; por esta razão, muitos sistemas operacionais executam o escalonador apenas esporadicamente, quando há necessidade de reordenar a fila de tarefas prontas. Nesse caso, o despachante sempre ativa a primeira tarefa da fila.

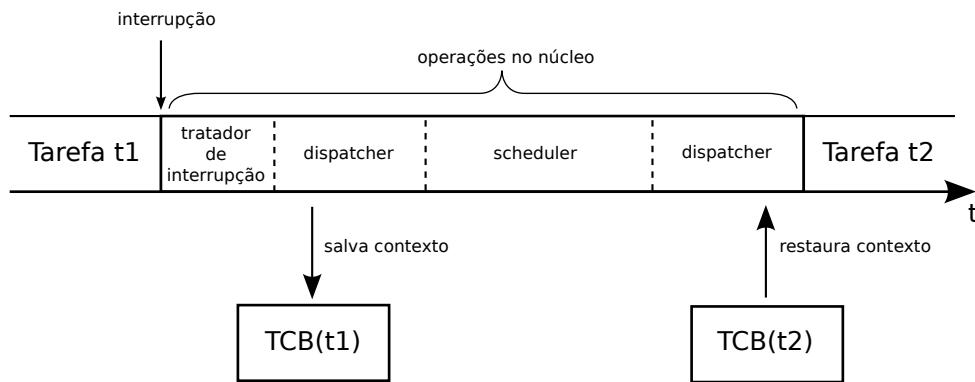


Figura 2.7: Passos de uma troca de contexto.

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 (também através de uma interrupção do hardware) ou pela execução de uma chamada de sistema pela tarefa corrente (ou seja, por uma interrupção de software) ou até mesmo por algum erro de execução que possa provocar uma exceção no processador.

2.4.3 Processos

Além de seu próprio código executável, cada tarefa ativa em um sistema de computação necessita de um conjunto de recursos para executar e cumprir seu objetivo. Entre esses recursos estão as áreas de memória usadas pela tarefa para armazenar seu código, dados e pilha, seus arquivos abertos, conexões de rede, etc. O conjunto dos recursos alocados a uma tarefa para sua execução é denominado **processo**.

Historicamente, os conceitos de tarefa e processo se confundem, sobretudo porque os sistemas operacionais mais antigos, até meados dos anos 80, somente suportavam uma tarefa para cada processo (ou seja, uma atividade associada a cada contexto). Essa visão vem sendo mantida por muitas referências até os dias de hoje. Por exemplo, os livros [Silberschatz et al., 2001] e [Tanenbaum, 2003] ainda apresentam processos como equivalentes de tarefas. No entanto, quase todos os sistemas operacionais contemporâneos suportam mais de uma tarefa por processo, como é o caso do Linux, Windows XP e os UNIX mais recentes.

Os sistemas operacionais convencionais atuais associam por *default* uma tarefa a cada processo, o que corresponde à execução de um programa sequencial (um único fluxo de instruções dentro do processo). Caso se deseje associar mais tarefas ao mesmo contexto (para construir o navegador Internet da Figura 2.1, por exemplo), cabe ao desenvolvedor escrever o código necessário para tal. Por essa razão, muitos livros ainda usam de forma equivalente os termos *tarefa* e *processo*, o que não corresponde mais à realidade.

Assim, 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. 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) e pela própria gerência de tarefas, que atribui os recursos aos processos (e não às tarefas), impedindo que uma tarefa em execução no processo p_a acesse um recurso atribuído ao processo p_b . A Figura 2.8 ilustra o conceito de processo, visto como um contêiner de recursos.

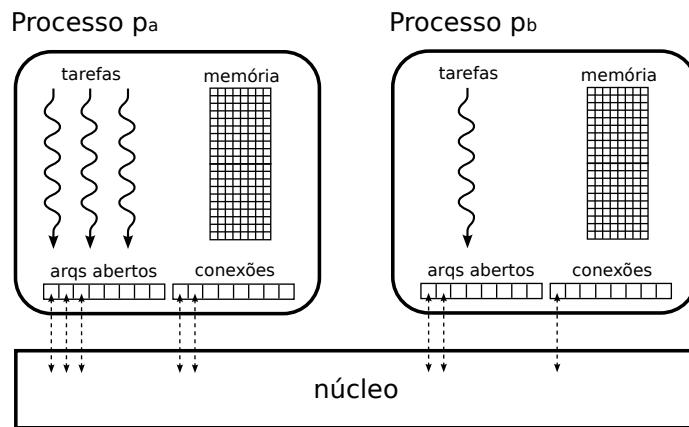


Figura 2.8: O processo visto como um contêiner de recursos.

O núcleo do sistema operacional mantém descritores de processos, denominados PCBs (*Process Control Blocks*), para armazenar as informações referentes aos processos ativos. Cada processo possui um identificador único no sistema, o PID – *Process IDentifier*. 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 ao qual a tarefa está vinculada. Disto observa-se também que a troca de contexto entre tarefas vinculadas ao mesmo processo é muito mais simples e rápida que entre tarefas vinculadas a processos distintos, pois somente os registradores do processador precisam ser salvos/restaurados (as áreas de memória e demais recursos são comuns às duas tarefas). Essas questões são aprofundadas na Seção 2.4.4.

Criação 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 criação e remoção de processos. No caso do UNIX, processos são criados através da chamada de sistema `fork`, que cria uma réplica do processo solicitante: todo o espaço de memória do processo é replicado, incluindo o código da(s) tarefa(s) associada(s) e os descritores dos arquivos e demais recursos associados ao mesmo. A Figura 2.9 ilustra o funcionamento dessa chamada.

A chamada de sistema `fork` é invocada por um processo (o pai), mas dois processos recebem seu retorno: o *processo pai*, que a invocou, e o *processo filho*, recém-criado, que

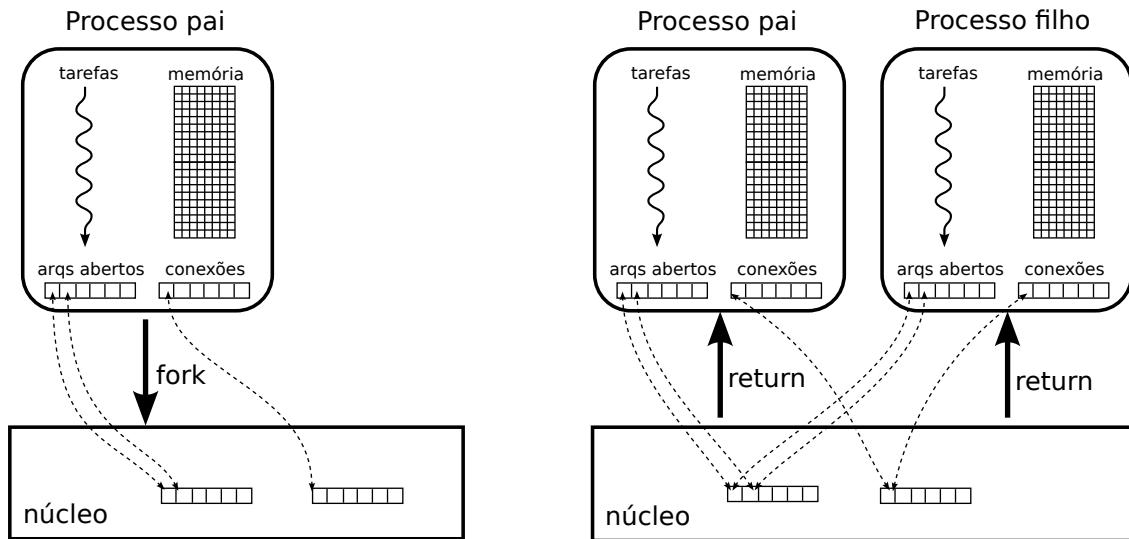


Figura 2.9: A chamada de sistema `fork`: antes (esq) e depois (dir) de sua execução pelo núcleo do sistema UNIX.

possui o mesmo estado interno que o pai (ele também está aguardando o retorno da chamada de sistema). Ambos os processos têm os mesmos recursos associados, embora em áreas de memória distintas. Caso o processo filho deseje abandonar o fluxo de execução herdado do processo pai e executar outro código, poderá fazê-lo através da chamada de sistema `execve`. 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 dessas duas chamadas de sistema:

```

1 #include <unistd.h>
2 #include <sys/types.h>
3 #include <sys/wait.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main (int argc, char *argv[], char *envp[])
8 {
9     int pid ;           /* identificador de processo */
10    pid = fork () ;    /* replicação do processo */
11
12    if ( pid < 0 )      /* fork não funcionou */
13    {
14        perror ("Erro: ") ;
15        exit (-1) ;      /* encerra o processo */
16    }
17    else if ( pid > 0 ) /* sou o processo pai */
18    {
19        wait (0) ;       /* vou esperar meu filho concluir */
20    }
21    else                 /* sou o processo filho*/
22    {
23        /* carrega outro código binário para executar */
24        execve ("/bin/date", argv, envp) ;
25        perror ("Erro: ") ; /* execve não funcionou */
26    }
27    printf ("Tchau !\n") ;
28    exit(0) ;            /* encerra o processo */
29 }
30 }
```

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.

Na operação de criação de processos do UNIX aparece de maneira bastante clara a noção de **hierarquia** entre 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 Linux permite visualizar a árvore de processos do sistema, como mostra a listagem a seguir.

```
1 init---aacraid
2   |-ahc_dv_0
3   |-atd
4   |-avaliacao_horac
5   |-bdflush
6   |-crond
7   |-gpm
8   |-kdm---X
9     '-kdm---kdm_greet
10  |-keventd
11  |-khubd
12  |-2*[kjournald]
13  |-klogd
14  |-ksoftirqd_CPU0
15  |-ksoftirqd_CPU1
16  |-kswapd
17  |-kupdated
18  |-lockd
19  |-login---bash
20  |-lpd---lpd---lpd
21  |-5*[mingetty]
22  |-8*[nfsd]
23  |-nmbd
24  |-nrpe
25  |-oafd
26  |-portmap
27  |-rhnsd
28  |-rpc.mountd
29  |-rpc.statd
30  |-rpctod
31  |-scsi_eh_0
32  |-scsi_eh_1
33  |-smbd
34  |-sshd---sshd---tcsh---top
35    |-sshd---bash
36    '-sshd---tcsh---pstree
37  |-syslogd
38  |-xfs
39  |-xinetc
40  '-ypbind---ypbind---2*[ypbind]
```

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 inter-processos (IPC – *Inter-Process Communication*). Esse tema será estudado em profundidade no Capítulo 3.

2.4.4 Threads

Conforme visto na Seção 2.4.3, os primeiros sistemas operacionais suportavam apenas uma tarefa por processo. À medida em que as aplicações se tornavam mais complexas, essa limitação se tornou um claro inconveniente. Por exemplo, um editor de textos geralmente executa tarefas simultâneas de edição, formatação, paginação e verificação ortográfica sobre a mesma massa de dados (o texto sob 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 no mesmo contexto, ou seja, dentro do mesmo processo.

De forma geral, cada fluxo de execução do sistema, seja associado a um processo ou no interior do núcleo, é denominado **thread**. *Threads* executando dentro de um processo são chamados de **threads de usuário** (*user-level threads* ou simplesmente *user threads*). Cada *thread* de usuário corresponde a uma tarefa a ser executada dentro de um processo. Por sua vez, os fluxos de execução reconhecidos e gerenciados pelo núcleo do sistema operacional são chamados de **threads de núcleo** (*kernel-level threads* ou *kernel threads*). Os *threads* de núcleo representam tarefas que o núcleo deve realizar. Essas tarefas podem corresponder à execução dos processos no espaço de usuário, ou a atividades internas do próprio núcleo, como *drivers* de dispositivos ou tarefas de gerência.

Os sistemas operacionais mais antigos não ofereciam suporte a *threads* para a construção de aplicações. Sem poder contar com o sistema operacional, os desenvolvedores de aplicações contornaram o problema construindo bibliotecas que permitiam criar e gerenciar *threads* dentro de cada processo, sem o envolvimento do núcleo do sistema. Usando essas bibliotecas, uma aplicação pode lançar vários *threads* conforme sua necessidade, mas o núcleo do sistema irá sempre perceber (e gerenciar) apenas um fluxo de execução dentro de cada processo. Por essa razão, esta forma de implementação de *threads* é nomeada **Modelo de Threads N:1**: N *threads* no processo, mapeados em um único *thread* de núcleo. A Figura 2.10 ilustra esse modelo.

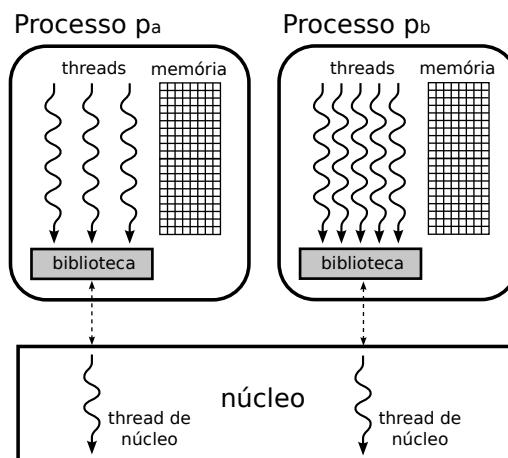


Figura 2.10: O modelo de *threads* N:1.

O modelo de *threads* N:1 é muito utilizado, por ser leve e de fácil implementação. Como o núcleo somente considera 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 (a simulação detalhada do tráfego viário de uma cidade grande, por exemplo, pode exigir um *thread* para cada veículo, resultando em centenas de milhares ou mesmo milhões de *threads*). Um exemplo de implementação desse modelo é a biblioteca *GNU Portable Threads* [Engeschall, 2005].

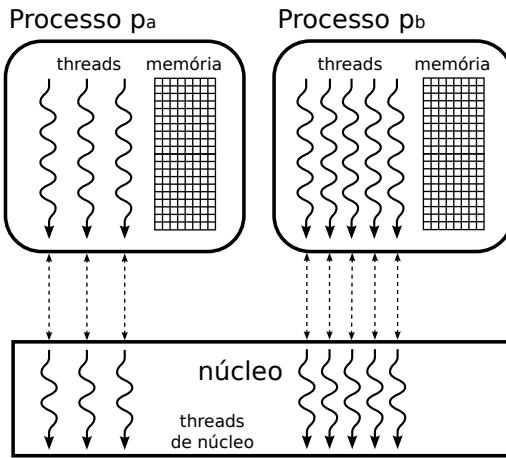
Entretanto, o modelo de *threads* N:1 apresenta problemas em algumas situações, sendo o mais grave deles relacionado às operações de entrada/saída. Como essas operações são intermediadas pelo núcleo, se um *thread* de usuário solicitar uma operação de E/S (recepção de um pacote de rede, por exemplo) o *thread* de núcleo correspondente será suspenso até a conclusão da operação, fazendo com que todos os *threads* de usuário associados ao processo parem de executar enquanto a operação não for concluída.

Outro problema desse modelo diz respeito à divisão de recursos entre as tarefas. O núcleo do sistema divide o tempo do processador entre os fluxos de execução que ele conhece e gerencia: as *threads* de núcleo. Assim, uma aplicação com 100 *threads* de usuário irá receber o mesmo tempo de processador que outra aplicação com apenas um *thread* (considerando que ambas as aplicações têm a mesma prioridade). Cada *thread* da primeira aplicação irá portanto receber 1/100 do tempo que recebe o *thread* único da segunda aplicação, o que não pode ser considerado uma divisão justa desse recurso.

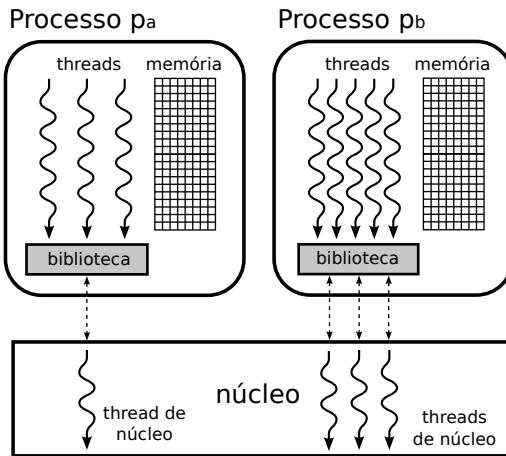
A necessidade de suportar aplicações com vários *threads* (*multithreaded*) levou os desenvolvedores de sistemas operacionais a incorporar a gerência dos *threads* de usuário ao núcleo do sistema. Para cada *thread* de usuário foi então definido um *thread* correspondente dentro do núcleo, suprimindo com isso a necessidade de bibliotecas de *threads*. Caso um *thread* de usuário solicite uma operação bloqueante (leitura de disco ou recepção de pacote de rede, por exemplo), somente seu respectivo *thread* de núcleo será suspenso, sem afetar os demais *threads*. Além disso, caso o hardware tenha mais de um processador, mais *threads* da mesma aplicação podem executar ao mesmo tempo, o que não era possível no modelo anterior. Essa forma de implementação, denominada **Modelo de Threads 1:1** e apresentada na Figura 2.11, é a mais frequente nos sistemas operacionais atuais, incluindo o Windows NT e seus descendentes, além da maioria dos UNIXes.

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

Para resolver o problema da escalabilidade, alguns sistemas operacionais implementam um modelo híbrido, que agrupa características dos modelos anteriores. Nesse novo modelo, uma biblioteca gerencia um conjunto de *threads* de usuário (dentro do processo), que é mapeado em um ou mais *threads* do núcleo. O conjunto de *threads* de núcleo associados a um processo é geralmente composto de um *thread* para cada tarefa bloqueada e mais um *thread* para cada processador disponível, podendo ser ajustado dinamicamente conforme a necessidade da aplicação. Essa abordagem híbrida

Figura 2.11: O modelo de *threads* 1:1.

é denominada **Modelo de Threads N:M**, onde N *threads* de usuário são mapeados em $M \leq N$ *threads* de núcleo. A Figura 2.12 apresenta esse modelo.

Figura 2.12: O modelo de *threads* N:M.

O modelo N:M é implementado pelo Solaris e também pelo projeto KSE (*Kernel-Scheduled Entities*) do FreeBSD [Evans and Elischer, 2003] baseado nas idéias apresentadas 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 comparado ao modelo 1:1. A Tabela 2.1 resume os principais aspectos dos três modelos de implementação de *threads* e faz um comparativo entre eles.

Modelo	N:1	1:1	N:M
Resumo	Todos os N <i>threads</i> do processo são mapeados em um único <i>thread</i> de núcleo	Cada <i>thread</i> do processo tem um <i>thread</i> correspondente no núcleo	Os N <i>threads</i> do processo são mapeados em um conjunto de M <i>threads</i> de núcleo
Local da implementação	bibliotecas no nível usuário	dentro do núcleo	em ambos
Complexidade	baixa	média	alta
Custo de gerência para o núcleo	nulo	médio	alto
Escalabilidade	alta	baixa	alta
Suporte a vários processadores	não	sim	sim
Velocidade das trocas de contexto entre <i>threads</i>	rápida	lenta	rápida entre <i>threads</i> no mesmo processo, lenta entre <i>threads</i> de processos distintos
Divisão de recursos entre tarefas	injusta	justa	variável, pois o mapeamento <i>thread</i> →processador é dinâmico
Exemplos	GNU Portable Threads	Windows XP, Linux	Solaris, FreeBSD KSE

Tabela 2.1: Quadro comparativo dos modelos de *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 *PThreads* [Nichols et al., 1996], que busca definir uma interface padronizada para a criação e manipulação 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* (para compilar deve ser usada a opção de ligação -lpthread).

```

1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #define NUM_THREADS 5
6
7 /* cada thread vai executar esta função */
8 void *PrintHello (void *threadid)
9 {
10     printf ("%d: Hello World!\n", (int) threadid);
11     pthread_exit (NULL);
12 }
13
14 /* thread "main" (vai criar as demais threads) */
15 int main (int argc, char *argv[])
16 {
17     pthread_t thread[NUM_THREADS];
18     int status, i;
19
20     /* cria as demais threads */
21     for(i = 0; i < NUM_THREADS; i++)
22     {
23         printf ("Creating thread %d\n", i);
24         status = pthread_create (&thread[i], NULL, PrintHello, (void *) i);
25
26         if (status) /* ocorreu um erro */
27         {
28             perror ("pthread_create");
29             exit (-1);
30         }
31     }
32
33     /* encerra a thread "main" */
34     pthread_exit (NULL);
35 }
```

O conceito de *threads* também pode ser utilizado em outras linguagens de programação, como Java, Python, Perl, C++ e C#. O código a seguir traz um exemplo simples de criação de *threads* em Java (extraído da documentação oficial da linguagem):

```

1 public class MyThread extends Thread {
2     int threadID;
3
4     MyThread (int ID) {
5         threadID = ID;
6     }
7
8     public void run () {
9         int i ;
10
11        for (i = 0; i< 100 ; i++)
12            System.out.println ("Hello from t" + threadID + "!" );
13    }
14
15    public static void main (String args[]) {
16        MyThread t1 = new MyThread (1);
17        MyThread t2 = new MyThread (2);
18        MyThread t3 = new MyThread (3);
19
20        t1.start ();
21        t2.start ();
22        t3.start ();
23    }
24 }
```

2.5 Escalonamento de tarefas

Um dos componentes mais importantes da gerência de tarefas é o **escalonador** (*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.

Antes de se definir o algoritmo usado por um escalonador, 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 anti-vírus, cálculos numéricos longos, 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 numéricos 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 despendem boa parte de suas existências no estado *suspensa*, 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.

2.5.1 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 2.4.2. 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 “vida” de cada 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).

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, o tempo decorrido entre apertar uma tecla e o caractere correspondente aparecer na tela, em um editor de textos. 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 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 similar devem receber tempos de processamento similares e ter durações de execução similares.

Eficiência : a eficiência \mathcal{E} , conforme definido na Seção 2.4.2, 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).

2.5.2 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, 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 pode reavaliar todas as tarefas da fila de prontas e decidir se mantém ou substitui a tarefa atualmente em execução.

Sistemas cooperativos : a tarefa em execução permanece no processador tanto quanto possível, só abandonando o mesmo caso termine de executar, solicite uma operação de entrada/saída ou libere explicitamente o processador, voltando à fila de tarefas prontas (isso normalmente é feito através de uma chamada de sistema `sched_yield()` ou similar). Esses sistemas são chamados de *cooperativos* por exigir a cooperação das tarefas entre si na gestão do processador, para que todas possam executar.

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, 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.

2.5.3 Escalonamento FCFS (*First-Come, First Served*)

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 chegada na fila de tarefas prontas). Esse algoritmo é conhecido como FCFS – *First Come - First Served* – e tem como principal vantagem sua simplicidade.

Para dar um exemplo do funcionamento do algoritmo FCFS, consideremos as tarefas na fila de tarefas prontas, com suas durações previstas de processamento e datas de ingresso no sistema, descritas na tabela a seguir:

tarefa	t_1	t_2	t_3	t_4
ingresso	0	0	1	3
duração	5	2	4	3

O diagrama da Figura 2.13 mostra o escalonamento do processador 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).

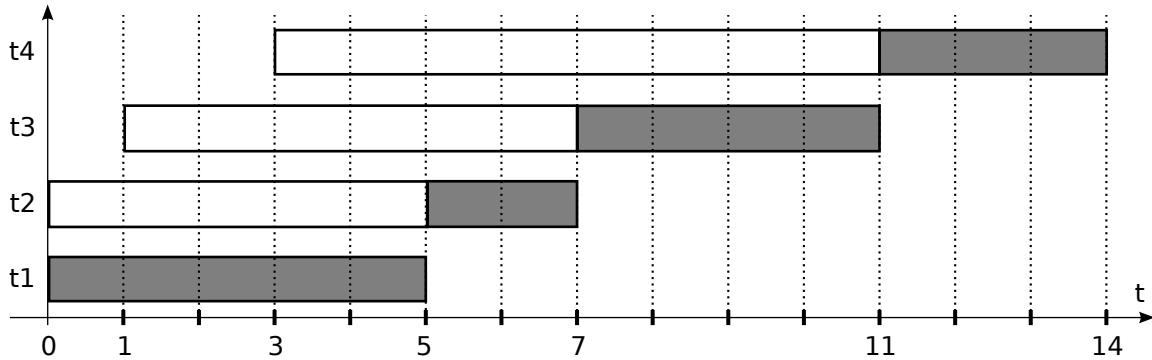


Figura 2.13: Escalonamento FCFS.

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 o algoritmo FCFS, temos:

$$\begin{aligned} T_t &= \frac{t_t(t_1) + t_t(t_2) + t_t(t_3) + t_t(t_4)}{4} = \frac{(5 - 0) + (7 - 0) + (11 - 1) + (14 - 3)}{4} \\ &= \frac{5 + 7 + 10 + 11}{4} = \frac{33}{4} = 8.25s \end{aligned}$$

$$\begin{aligned} T_w &= \frac{t_w(t_1) + t_w(t_2) + t_w(t_3) + t_w(t_4)}{4} = \frac{(0 - 0) + (5 - 0) + (7 - 1) + (11 - 3)}{4} \\ &= \frac{0 + 5 + 6 + 8}{4} = \frac{19}{4} = 4.75s \end{aligned}$$

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 2.14.

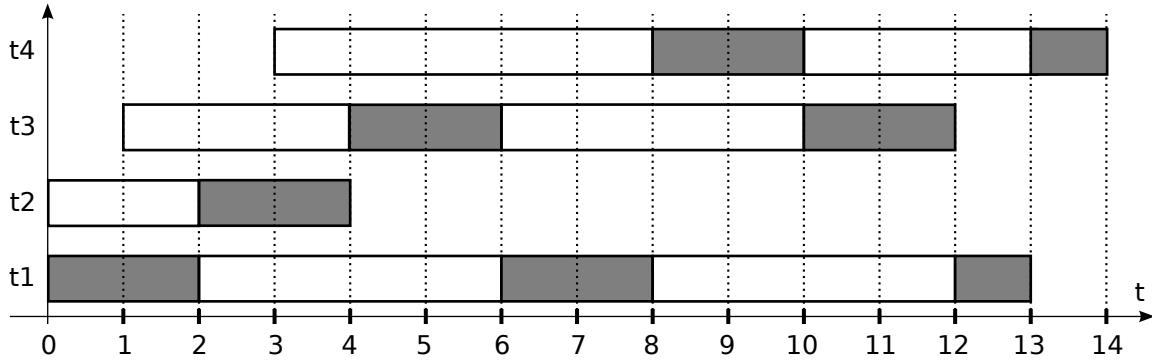


Figura 2.14: Escalonamento *Round-Robin*.

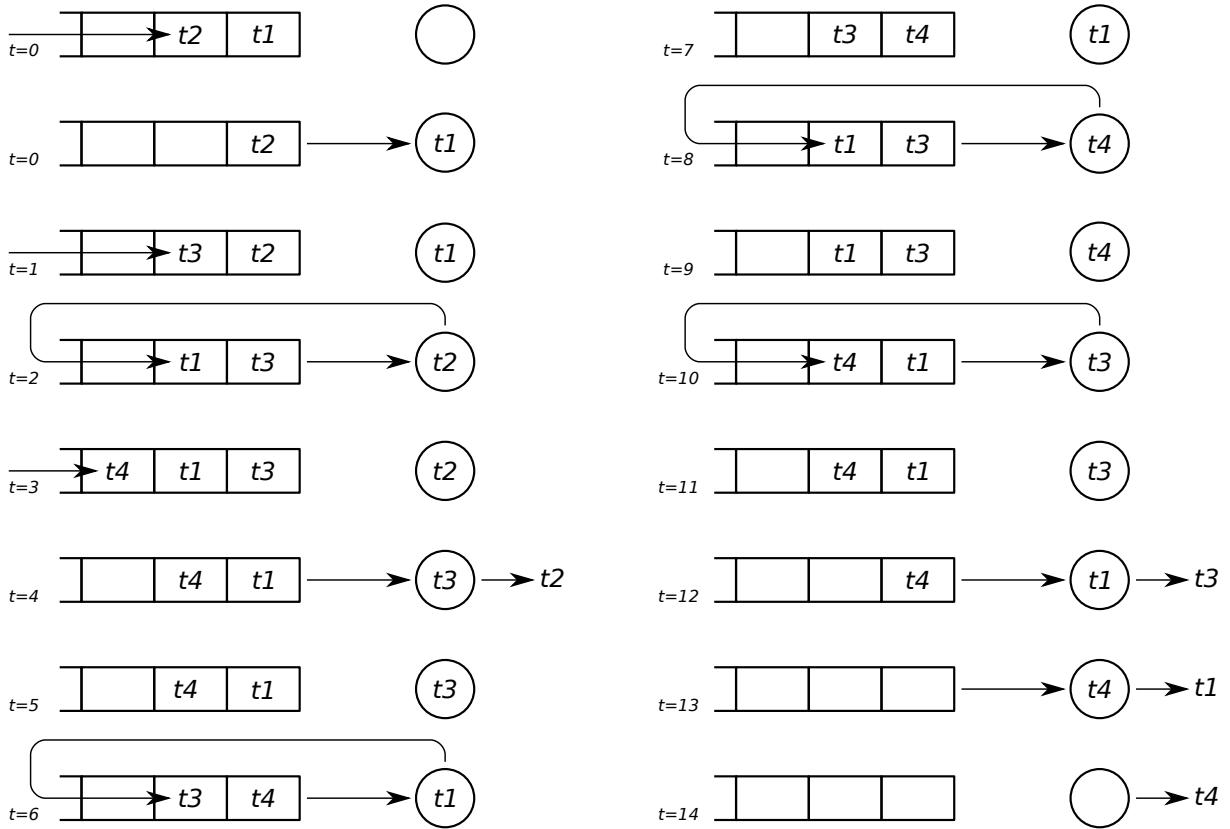
Na Figura 2.14, é importante observar que a execução das tarefas não obedece uma sequência óbvia como $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \rightarrow t_1 \rightarrow t_2 \rightarrow \dots$, mas uma sequência bem mais complexa: $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_1 \rightarrow t_4 \rightarrow t_3 \rightarrow \dots$. 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 2.15 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 o algoritmo *round-robin*, temos:

$$\begin{aligned} T_t &= \frac{t_t(t_1) + t_t(t_2) + t_t(t_3) + t_t(t_4)}{4} = \frac{(13 - 0) + (4 - 0) + (12 - 1) + (14 - 3)}{4} \\ &= \frac{13 + 4 + 11 + 11}{4} = \frac{39}{4} = 9.75s \\ T_w &= \frac{t_w(t_1) + t_w(t_2) + t_w(t_3) + t_w(t_4)}{4} = \frac{8 + 2 + 7 + 8}{4} = \frac{25}{4} = 6.25s \end{aligned}$$

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 pode proporcionar tempos de resposta bem melhores às aplicações interativas.

O aumento da quantidade de trocas de contexto também tem um impacto negativo 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} :

Figura 2.15: Evolução da fila de tarefas prontas no escalonamento *Round-Robin*.

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

Por exemplo, um sistema no qual as trocas de contexto duram $1ms$ e cujo *quantum* médio é de $20ms$ terá uma eficiência $\mathcal{E} = \frac{20}{20+1} = 95,2\%$. Caso a duração do quantum seja reduzida para $2ms$, a eficiência cairá para $\mathcal{E} = \frac{2}{2+1} = 66,7\%$. 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).

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.

2.5.4 Escalonamento SJF (*Shortest Job First*)

O algoritmo de escalonamento que proporciona os menores tempos médios de execução e de espera é conhecido como *menor tarefa primeiro*, ou SJF (*Shortest Job First*). Como o nome indica, ele consiste em atribuir o processador à menor (mais curta) tarefa da fila de tarefas prontas. Pode ser provado matematicamente que esta estratégia sempre proporciona os menores tempos médios de espera. Aplicando-se este algoritmo às tarefas da tabela anterior, obtém-se o escalonamento apresentado na Figura 2.16.

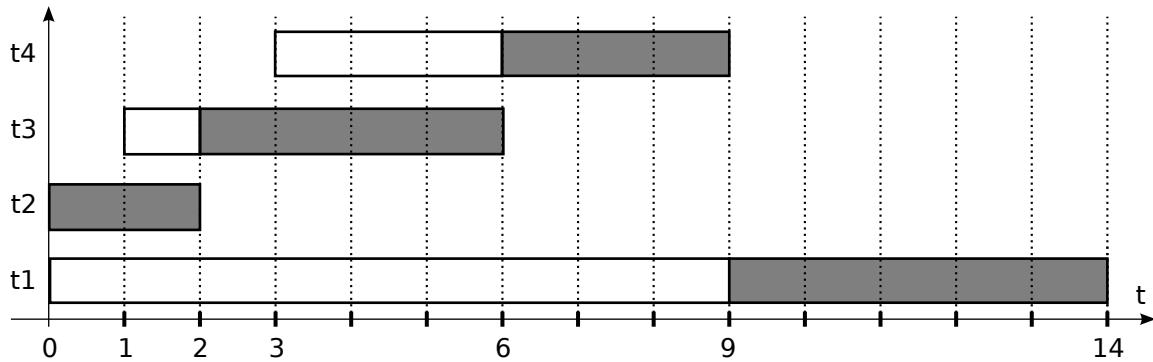


Figura 2.16: Escalonamento SJF.

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para o algoritmo SJF, temos:

$$\begin{aligned} T_t &= \frac{t_t(t_1) + t_t(t_2) + t_t(t_3) + t_t(t_4)}{4} = \frac{(14 - 0) + (2 - 0) + (6 - 1) + (9 - 3)}{4} \\ &= \frac{14 + 2 + 5 + 6}{4} = \frac{27}{4} = 6.75s \\ T_w &= \frac{t_w(t_1) + t_w(t_2) + t_w(t_3) + t_w(t_4)}{4} = \frac{(9 - 0) + (0 - 0) + (2 - 1) + (6 - 3)}{4} \\ &= \frac{9 + 0 + 1 + 3}{4} = \frac{13}{4} = 3.25s \end{aligned}$$

Deve-se observar que o comportamento expresso na Figura 2.16 corresponde à versão cooperativa do algoritmo SJF: o escalonador aguarda a conclusão de cada tarefa para decidir quem irá receber o processador. No caso preemptivo, o escalonador deve comparar a duração prevista de cada nova tarefa que ingressa no sistema com o tempo restante de processamento das demais tarefas presentes, inclusive aquela que está executando no momento. Essa abordagem é denominada por alguns autores de *menor tempo restante primeiro* (SRTF – *Short Remaining Time First*) [Tanenbaum, 2003].

A maior dificuldade no uso do algoritmo SJF consiste em estimar a priori a duração de cada tarefa, ou seja, 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 à preempção por

tempo, esse algoritmo pode ser de grande valia, sobretudo para tarefas orientadas a entrada/saída.

Suponha 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 duração da execução 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 priorizar 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 2.5.5.

2.5.5 Escalonamento por prioridades

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 prioridades, a cada tarefa é associada uma prioridade, geralmente na forma de um número inteiro. Os valores de prioridade são então usados para escolher a próxima tarefa a receber o processador, a cada troca de contexto. O algoritmo de escalonamento por prioridades define um modelo genérico de escalonamento, que permite modelar várias abordagens, entre as quais o FCFS e o SJF.

Para ilustrar o funcionamento do escalonamento por prioridades, serão usadas as tarefas descritas na tabela a seguir, que usam uma escala de prioridades positiva (ou seja, onde valores maiores indicam uma prioridade maior):

tarefa	t_1	t_2	t_3	t_4
ingresso	0	0	1	3
duração	5	2	4	3
prioridade	2	3	1	4

O diagrama da Figura 2.17 mostra o escalonamento do processador usando o algoritmo por prioridades em modo cooperativo (ou seja, sem *quantum* ou outras interrupções).

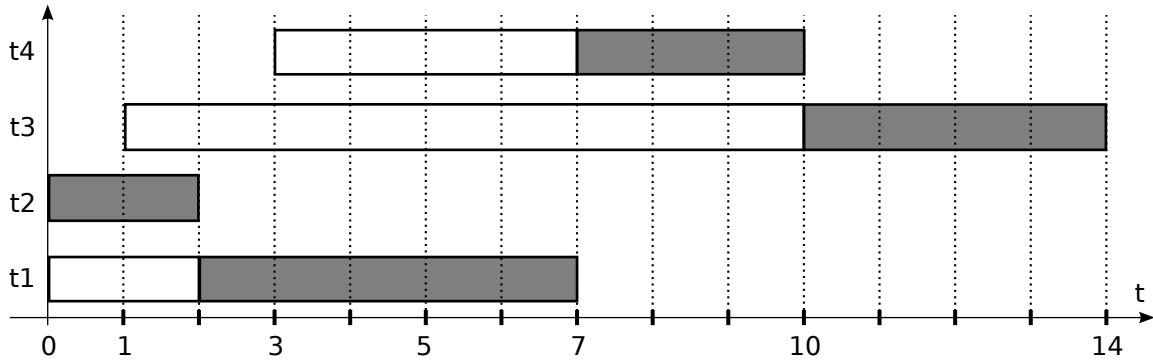


Figura 2.17: Escalonamento por prioridades (cooperativo).

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para esse algoritmo, temos:

$$\begin{aligned} T_t &= \frac{t_t(t_1) + t_t(t_2) + t_t(t_3) + t_t(t_4)}{4} = \frac{(7 - 0) + (2 - 0) + (14 - 1) + (10 - 3)}{4} \\ &= \frac{7 + 2 + 13 + 7}{4} = \frac{29}{4} = 7.25s \end{aligned}$$

$$\begin{aligned} T_w &= \frac{t_w(t_1) + t_w(t_2) + t_w(t_3) + t_w(t_4)}{4} = \frac{(2 - 0) + (0 - 0) + (10 - 1) + (7 - 3)}{4} \\ &= \frac{2 + 0 + 9 + 4}{4} = \frac{15}{4} = 3.75s \end{aligned}$$

Quando uma tarefa de maior prioridade se torna disponível para execução, o escalonador pode decidir entregar o processador a ela, trazendo a tarefa atual de volta para a fila de prontas. Nesse caso, temos um escalonamento por prioridades *preemptivo*, cujo comportamento é apresentado na Figura 2.18 (observe que, quando t_4 ingressa no sistema, ela recebe o processador e t_1 volta a esperar na fila de prontas).

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para esse algoritmo, temos:

$$\begin{aligned} T_t &= \frac{t_t(t_1) + t_t(t_2) + t_t(t_3) + t_t(t_4)}{4} = \frac{(10 - 0) + (2 - 0) + (14 - 1) + (6 - 3)}{4} \\ &= \frac{10 + 2 + 13 + 3}{4} = \frac{28}{4} = 7s \end{aligned}$$

$$T_w = \frac{t_w(t_1) + t_w(t_2) + t_w(t_3) + t_w(t_4)}{4} = \frac{5 + 0 + 9 + 0}{4} = \frac{14}{4} = 3.5s$$

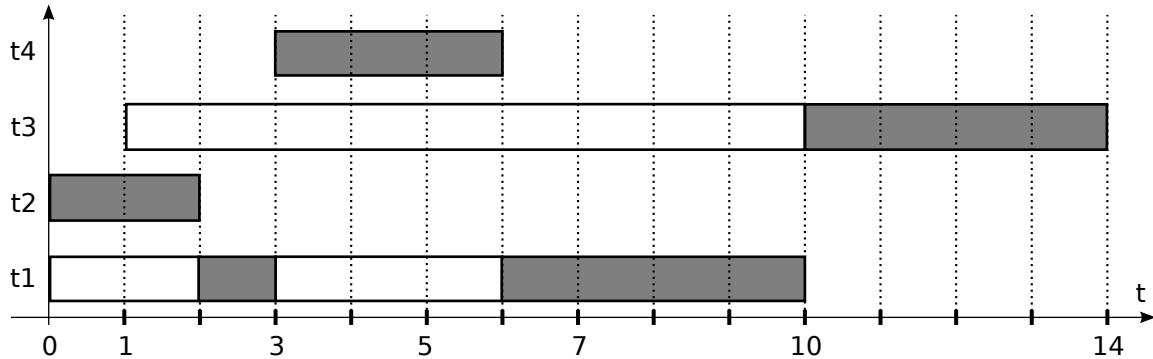


Figura 2.18: Escalonamento por prioridades (preemptivo).

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 : são 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 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 : são informações que podem ser obtidas ou estimadas pelo escalonador, com base em dados disponíveis no sistema local. Os fatores internos mais utilizados são a idade da tarefa, sua duração estimada, sua interatividade, seu uso de memória ou de outros recursos, etc.

Todos esses fatores devem ser combinados para produzir um valor de prioridade para cada tarefa. Todos os fatores externos são expressos por valor inteiro denominado **prioridade estática** (ou *prioridade de base*), que resume a “opinião” do usuário ou administrador sobre aquela tarefa. Os fatores internos mudam continuamente e devem ser recalculados periodicamente pelo escalonador. A combinação da prioridade estática com os fatores internos resulta na **prioridade dinâmica** ou final, que é usada pelo escalonador para ordenar as tarefas prontas. A Figura 2.19 resume esse procedimento.

Em geral, cada família de sistemas operacionais define sua própria escala de prioridades estáticas. Alguns exemplos de escalas comuns são:

Windows 2000 e sucessores : 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*:

- 4: baixa ou ociosa

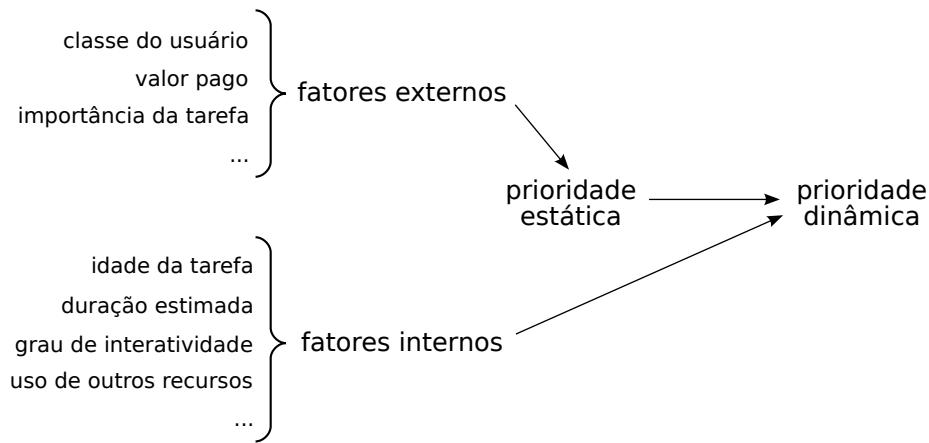


Figura 2.19: Composição da prioridade dinâmica.

- 6: *abaixo do normal*
- 8: *normal*
- 10: *acima do normal*
- 13: *alta*
- 24: *tempo-real*

Geralmente a prioridade da tarefa responsável pela janela ativa recebe um incremento de prioridade (+1 ou +2, conforme a configuração do sistema).

No Linux (núcleo 2.4 e sucessores) há duas escalas de prioridades:

- *Tarefas interativas*: a escala de prioridades é negativa: a prioridade de cada tarefa vai de -20 (mais importante) a +19 (menos importante) e pode ser ajustada através dos comandos `nice` e `renice`. Esta escala é padronizada em todos os sistemas UNIX.
- *Tarefas de tempo-real*: a prioridade de cada tarefa vai de 1 (mais importante) a 99 (menos importante). As tarefas de tempo-real têm precedência sobre as tarefas interativas e são escalonadas usando políticas distintas. Somente o administrador pode criar tarefas de tempo-real.

Inanição e envelhecimento de tarefas

No escalonamento por prioridades básico, as tarefas de baixa prioridade só recebem o processador na ausência de tarefas de maior prioridade. Caso existam tarefas de maior prioridade frequentemente ativas, as de baixa prioridade podem sofrer de inanição (*starvation*), ou seja, nunca ter acesso ao processador.

Além disso, em sistemas de tempo compartilhado, as prioridades estáticas definidas pelo usuário estão intuitivamente relacionadas à *proporcionalidade* na divisão do tempo de processamento. Por exemplo, se um sistema recebe duas tarefas iguais com a mesma prioridade, espera-se que cada uma receba 50% do processador e que ambas concluam ao

mesmo tempo. Caso o sistema 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). Entretanto, se aplicarmos o algoritmo de prioridades básico, as tarefas irão executar de forma sequencial, sem distribuição proporcional do processador. Esse resultado indesejável ocorre porque, a cada fim de quantum, sempre a mesma tarefa é escolhida para processar: a mais prioritária. Essa situação está ilustrada na Figura 2.20.

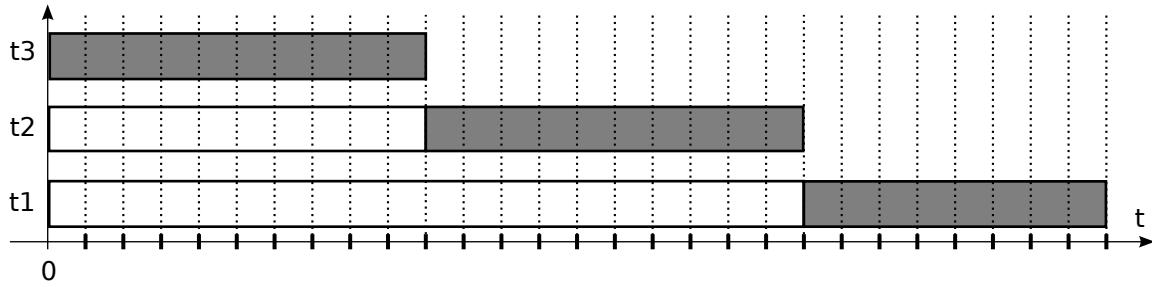


Figura 2.20: Escalonamento por prioridades.

Para evitar a inanição e garantir a proporcionalidade expressa através das prioridades estáticas, um fator interno denominado **envelhecimento** (*task aging*) deve ser definido. O envelhecimento indica há quanto tempo uma tarefa está aguardando o processador e aumenta sua prioridade proporcionalmente. Dessa forma, o envelhecimento evita a inanição dos processos de baixa prioridade, permitindo a eles obter o processador periodicamente. Uma forma simples de implementar o envelhecimento está resumida no seguinte algoritmo (que considera uma escala de prioridades positiva):

Definições:

- t_i : tarefa i
- pe_i : prioridade estática de t_i
- pd_i : prioridade dinâmica de t_i
- N : número de tarefas no sistema

Quando uma tarefa nova t_n ingressa no sistema:

$$\begin{aligned} pe_n &\leftarrow \text{prioridade inicial default} \\ pd_n &\leftarrow pe_n \end{aligned}$$

Para escolher a próxima tarefa a executar t_p :

$$\begin{aligned} \text{escolher } t_p \mid pd_p &= \max_{i=1}^N (pd_i) \\ pd_p &\leftarrow pe_p \\ \forall i \neq p : pd_i &\leftarrow pd_i + \alpha \end{aligned}$$

Em outras palavras, a cada turno o escalonador escolhe como próxima tarefa (t_p) aquela com a maior prioridade dinâmica (pd_p). A prioridade dinâmica dessa tarefa é igualada à sua prioridade estática ($pd_p \leftarrow pe_p$) e então ela recebe o processador. A prioridade dinâmica das demais tarefas é aumentada de α , ou seja, elas “envelhecem”

e no próximo turno terão mais chances de ser escolhidas. A constante α é conhecida como *fator de envelhecimento*.

Usando o algoritmo de envelhecimento, a divisão do processador entre as tarefas se torna proporcional às suas prioridades. A Figura 2.21 ilustra essa proporcionalidade na execução das três tarefas t_1 , t_2 e t_3 com $p(t_1) < p(t_2) < p(t_3)$, usando a estratégia de envelhecimento. Nessa figura, 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 .

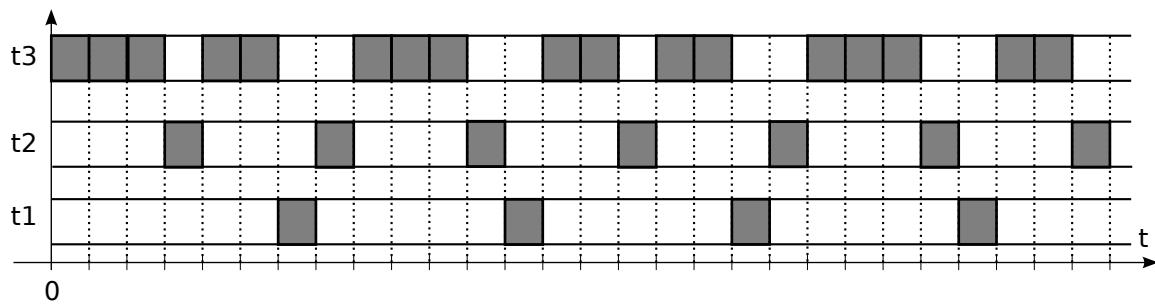


Figura 2.21: Escalonamento por prioridades com envelhecimento.

Inversão e herança de prioridades

Outro problema relevante que pode ocorrer em sistemas baseados em prioridades é a *inversão de prioridades* [Sha et al., 1990]. Este tipo de problema é mais complexo que o anterior, pois envolve o conceito de *exclusão mútua*: alguns recursos do sistema devem ser usados por um processo de cada vez, para evitar problemas de consistência de seu estado interno. Isso pode ocorrer com arquivos, portas de entrada saída e conexões de rede, por exemplo. Quando um processo obtém acesso a um recurso com exclusão mútua, os demais processos 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 4.

A inversão de prioridades consiste em processos de alta prioridade serem impedidos de executar por causa de um processo de baixa prioridade. Para ilustrar esse problema, pode ser considerada a seguinte situação: um determinado sistema possui um processo de alta prioridade p_a , um processo de baixa prioridade p_b e alguns processos de prioridade média p_m . Além disso, há um recurso R que deve ser acessado em exclusão mútua; para simplificar, somente p_a e p_b estão interessados em usar esse recurso. A seguinte sequência de eventos, ilustrada na Figura 2.22, é um exemplo de como pode ocorrer uma inversão de prioridades:

1. Em um dado momento, o processador está livre e é alocado a um processo de baixa prioridade p_b ;
2. durante seu processamento, p_b obtém o acesso exclusivo a um recurso R e começa a usá-lo;

3. p_b perde o processador, pois um processo com prioridade maior que a dele (p_m) foi acordado devido a uma interrupção;
4. p_b volta ao final da fila de tarefas prontas, aguardando o processador; enquanto ele não voltar a executar, o recurso R permanecerá alocado a ele e ninguém poderá usá-lo;
5. Um processo de alta prioridade p_a recebe o processador e solicita acesso ao recurso R ; como o recurso está alocado ao processo p_b , p_a é suspenso até que o processo de baixa prioridade p_b libere o recurso.

Neste momento, o processo de alta prioridade p_a não pode continuar sua execução, porque o recurso de que necessita está nas mãos do processo de baixa prioridade p_b . Dessa forma, p_a deve esperar que p_b execute e libere R , o que justifica o nome *inversão de prioridades*. A espera de p_a pode ser longa, pois p_b tem baixa prioridade e pode demorar a receber o processador novamente, caso existam outros processos em execução no sistema (como p_m). Como tarefas de alta prioridade são geralmente críticas para o funcionamento de um sistema, a inversão de prioridades pode ter efeitos graves.

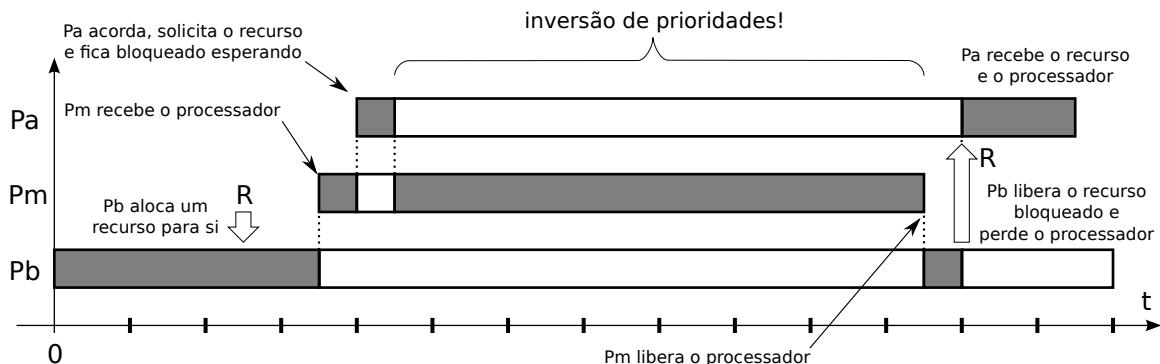


Figura 2.22: Cenário de uma inversão de prioridades.

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 do processo p_b que detém o recurso de uso exclusivo R . Caso esse recurso seja requisitado por um processo de maior prioridade p_a , o processo p_b “herda” temporariamente a prioridade de p_a , para que possa voltar a executar e liberar o recurso R mais rapidamente. Assim que liberar o recurso, p_b retorna à sua prioridade anterior. Essa estratégia está ilustrada na Figura 2.23.

Provavelmente o melhor exemplo real de inversão de prioridades tenha ocorrido na sonda espacial *Mars Pathfinder*, enviada pela NASA em 1996 para explorar o solo marciano (Figura 2.24) [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. Essas tarefas se comunicavam através de uma área de transferência em memória compartilhada, com acesso mutuamente exclusivo controlado por semáforos (semáforos são estruturas de sincronização discutidas na Seção 4.6).

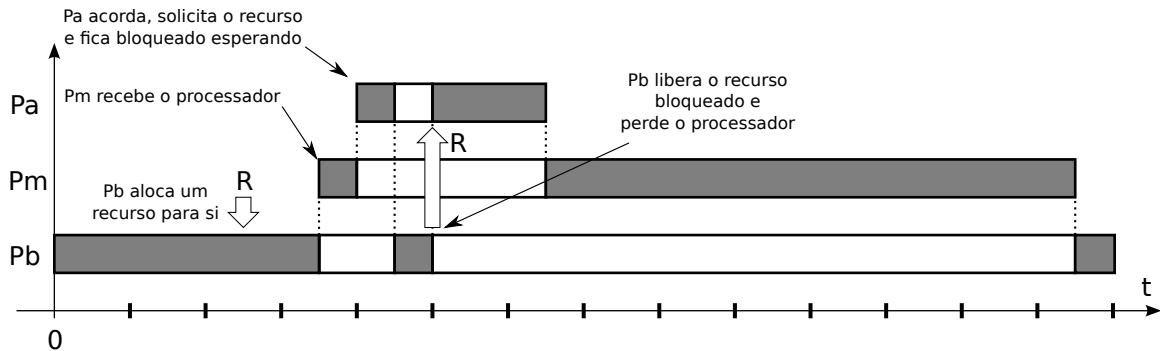


Figura 2.23: Um protocolo de herança de prioridade.

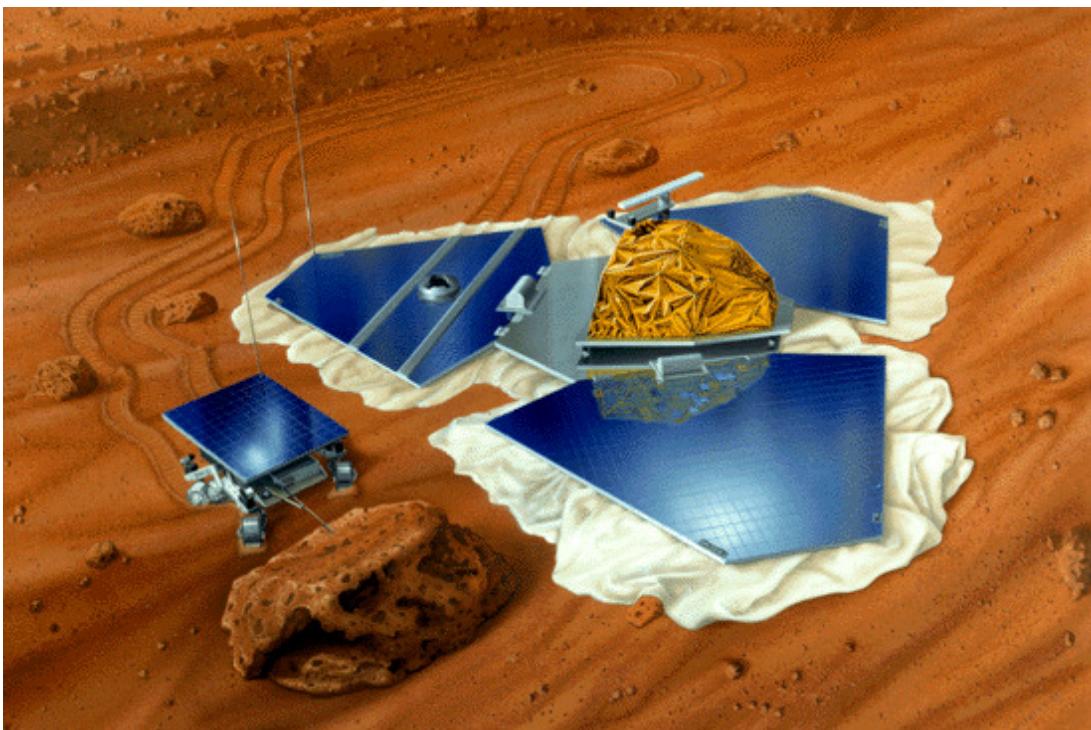
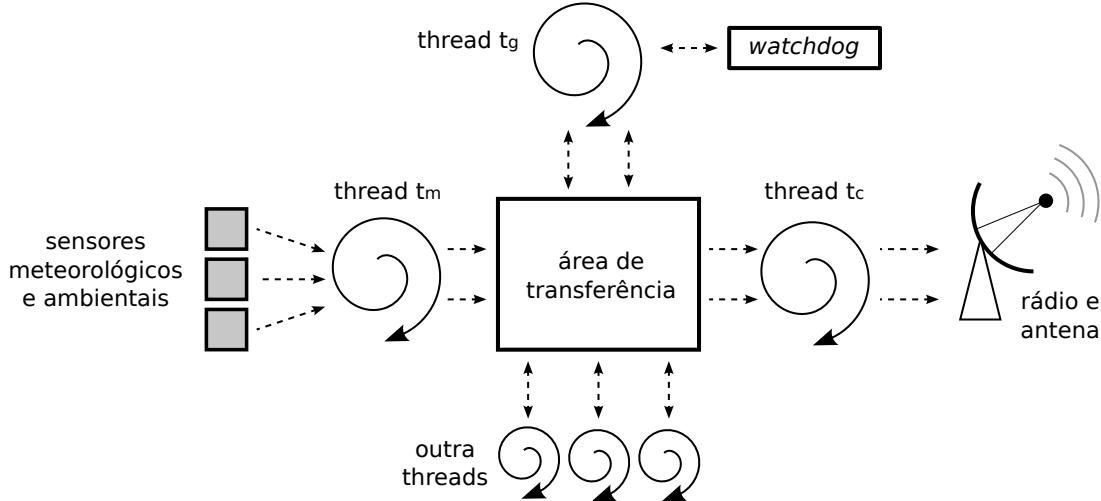


Figura 2.24: Sonda Mars Pathfinder com o robô Sojourner (NASA).

A gerência da área de transferência estava a cargo de uma tarefa t_g , rápida mas 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_m 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_c de prioridade média e potencialmente demorada (Tabela 2.2 e Figura 2.25).

Como o sistema VxWorks define prioridades preemptivas, 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_m perdesse o processador sem liberar a área de transferência, a tarefa de gerência t_g teria de ficar esperando até que t_m voltasse a

tarefa	função	prioridade	duração
t_g	gerência da área de transferência	alta	curta
t_m	coleta de dados meteorológicos	baixa	curta
t_c	comunicação com a Terra	média	longa

Tabela 2.2: Algumas tarefas do software da sonda *Mars Pathfinder*.Figura 2.25: Principais tarefas do software embarcado da sonda *Mars Pathfinder*.

executar para liberar a área. Isso poderia poderia demorar se, por azar, a tarefa de comunicação estivesse executando, pois ela tinha mais prioridade que t_m .

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 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 da herança de prioridades: caso a tarefa de gerência t_g fosse bloqueada pela tarefa de coleta de dados t_m , esta última herdava a alta prioridade de t_g para poder liberar rapidamente a área de transferência, mesmo se a tarefa de comunicação t_c estivesse em execução.

2.5.6 Outros algoritmos de escalonamento

Além dos algoritmos de escalonamento vistos nesta seção, diversos outros podem ser encontrados na literatura e em sistemas de mercado, como os escalonadores de tempo-real [Farines et al., 2000], os escalonadores multimídia [Nieh and Lam, 1997], os escalonadores justos [Kay and Lauder, 1988, Ford and Susarla, 1996], os escalonadores multi-processador [Black, 1990] e multi-core [Boyd-Wickizer et al., 2009].

2.5.7 Um escalonador real

Na prática, os sistemas operacionais de mercado implementam mais de um algoritmo de escalonamento. A escolha do escalonador adequado é feita com base na *classe de escalonamento* atribuída a cada tarefa. Por exemplo, o núcleo Linux implementa dois escalonadores (Figura 2.26): um escalonador de tarefas de tempo-real (classes SCHED_FIFO e SCHED_RR) e um escalonador de tarefas interativas (classe SCHED_OTHER) [Love, 2004]. Cada uma dessas classes de escalonamento está explicada a seguir:

Classe SCHED_FIFO : as tarefas associadas a esta classe são escalonadas usando uma política FCFS sem preempção (sem *quantum*) e usando apenas suas prioridades estáticas (não há envelhecimento). Portanto, uma tarefa desta classe executa até bloquear por recursos ou liberar explicitamente o processador (através da chamada de sistema `sched_yield()`).

Classe SCHED_RR : implementa uma política similar à anterior, com a inclusão da preempção por tempo. O valor do *quantum* é proporcional à prioridade atual de cada tarefa, variando de 10ms a 200ms.

Classe SCHED_OTHER : suporta tarefas interativas em lote, através de uma política baseada em prioridades dinâmicas com preempção por tempo com *quantum* variável. Tarefas desta classe somente são escalonadas se não houverem tarefas prontas nas classes SCHED_FIFO e SCHED_RR.

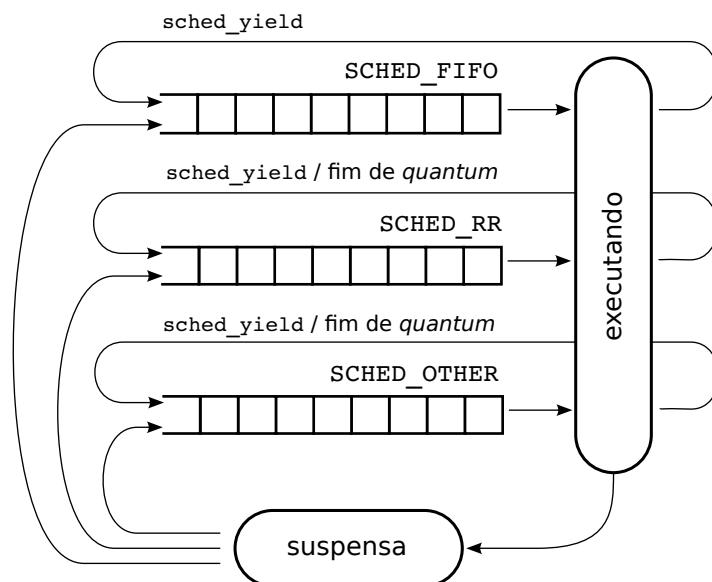


Figura 2.26: O escalonador multi-filas do Linux.

As classes de escalonamento SCHED_FIFO e SCHED_RR são reservadas para tarefas de tempo-real, que só podem ser lançadas pelo administrador do sistema. Todas as demais tarefas, ou seja, a grande maioria das aplicações e comandos dos usuários, executa na classe de escalonamento SCHED_OTHER.

Capítulo 3

Comunicação entre tarefas

Muitas implementações de sistemas complexos são estruturadas como várias tarefas inter-dependentes, 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 uma 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.

3.1 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 inter-dependentes 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 multi-processador : 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 inter-dependentes. 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.

3.2 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 do suporte de rede disponível. A Figura 3.1 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 de IPC” (*Inter-Process Communication mechanisms*).

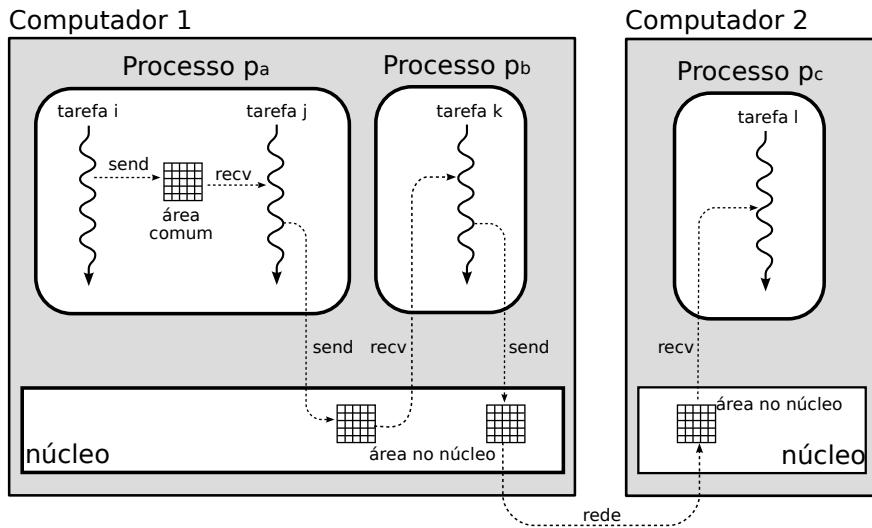


Figura 3.1: Comunicação intra-processo ($t_i \rightarrow t_j$), inter-processos ($t_j \rightarrow t_k$) e inter-sistemas ($t_k \rightarrow t_l$).

3.3 Características dos mecanismos de 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.

3.3.1 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 3.2 ilustra essas duas formas de comunicação.

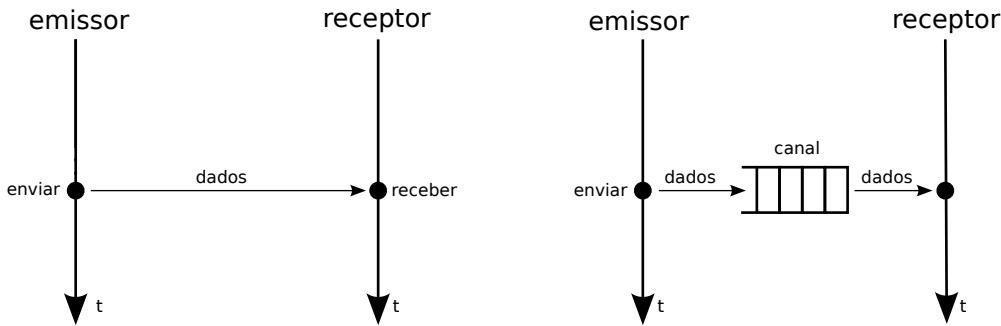


Figura 3.2: Comunicação direta (esquerda) e indireta (direita).

3.3.2 Síncronismo

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

Síncrona : 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. Esta modalidade de interação também é conhecida como **comunicação bloqueante**. A Figura 3.3 apresenta os diagramas de tempo de duas situações frequentes em sistemas com comunicação síncrona.

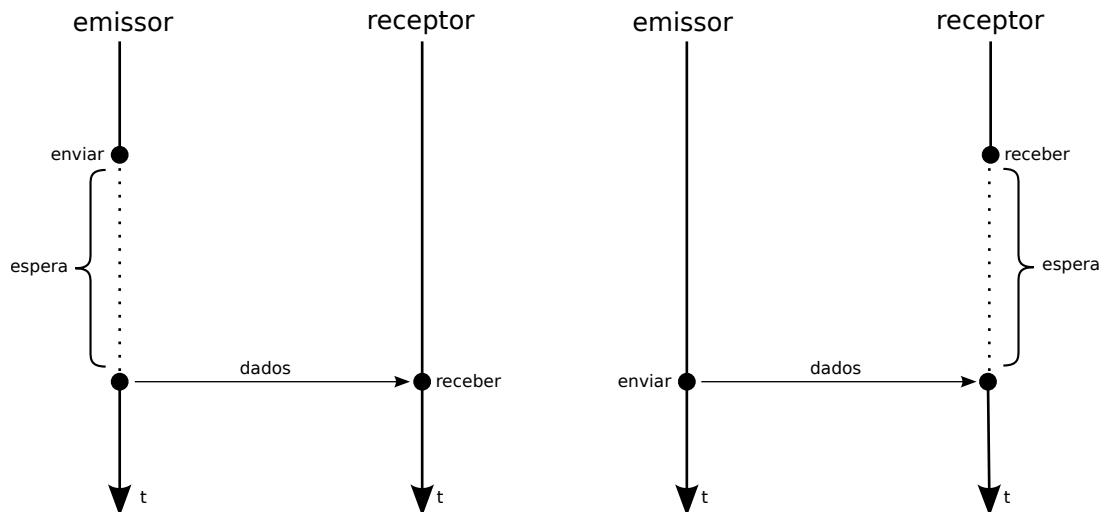


Figura 3.3: Comunicação síncrona.

Assíncrona : 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, também conhecida como **comunicação não-bloqueante**, está representada no diagrama de tempo da Figura 3.4.

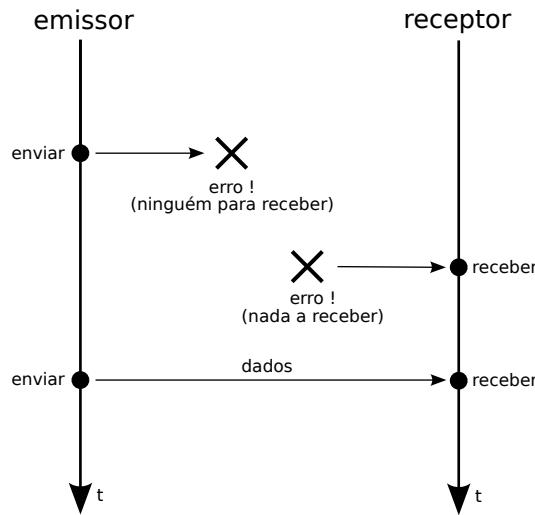


Figura 3.4: Comunicação assíncrona.

Semi-síncrona : primitivas de comunicação semi-síncronas (ou semi-bloqueantes) 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: *enviar (dados, destino, prazo)* e *receber (dados, origem, prazo)*. A Figura 3.5 ilustra duas situações em que ocorre esse comportamento.

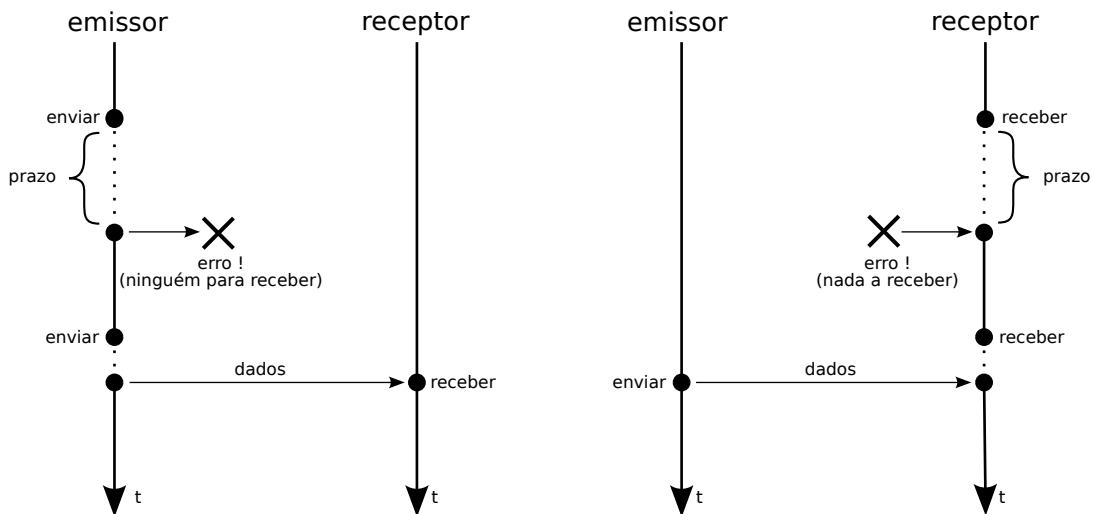


Figura 3.5: Comunicação semi-síncrona.

3.3.3 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 3.6). 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 3.4.

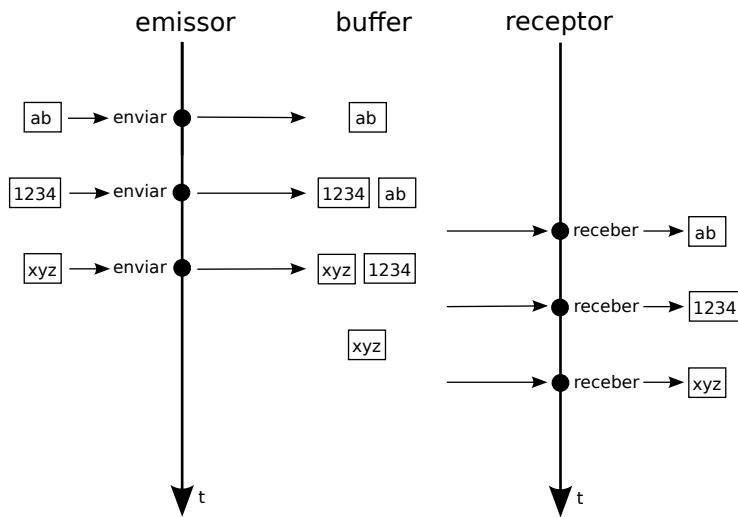


Figura 3.6: Comunicação baseada em mensagens.

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 3.7 apresenta o comportamento dessa forma de comunicação.

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 3.4.

3.3.4 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

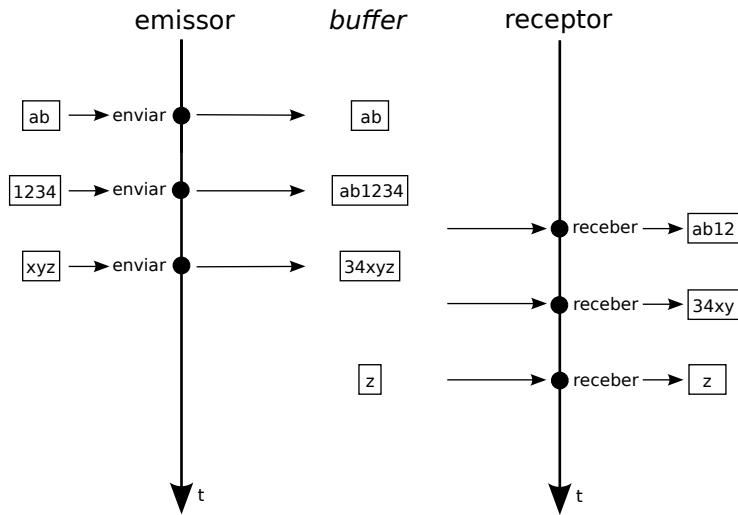


Figura 3.7: Comunicação baseada em fluxo de dados.

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 3.3 ilustra dois casos de *Rendez-Vous*. Por outro lado, a comunicação assíncrona torna-se inviável usando canais de capacidade nula (conforme discutido na Seção 3.3.2).

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 3.8 apresenta o comportamento de duas tarefas trocando dados através de um canal de comunicação com capacidade para duas mensagens e comportamento síncrono (bloqueante).

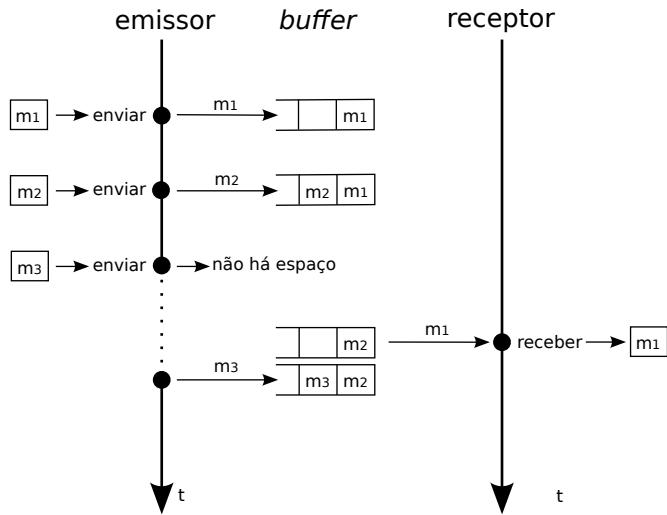


Figura 3.8: Comunicação síncrona usando um canal com capacidade 2.

3.3.5 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:

- *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.

3.3.6 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:

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* 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 3.9), sendo implementada nas *message queues* UNIX e 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 **todos os receptores** (cada receptor recebe uma cópia da mensagem). Essa abordagem é conhecida pelos nomes de *difusão* (*multicast*) ou *canal de eventos* (Figura 3.10), sendo implementada por exemplo no protocolo UDP.

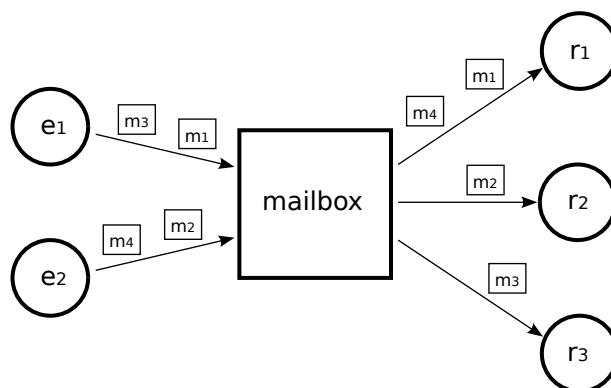


Figura 3.9: Comunicação M:N através de um *mailbox*.

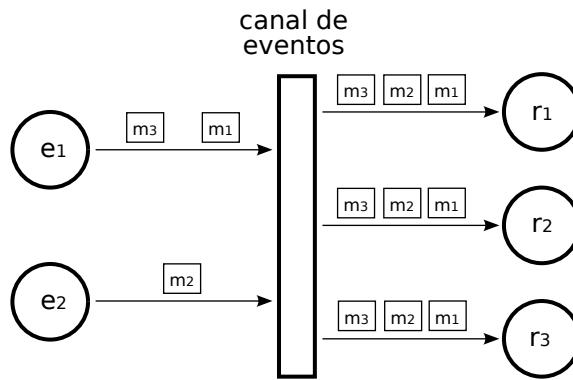


Figura 3.10: Difusão através de um canal de eventos.

3.4 Exemplos de mecanismos de comunicação

Nesta seção serão apresentados alguns mecanismos de comunicação usados com frequência 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].

3.4.1 Filas de mensagens UNIX

As filas de mensagens foram definidas inicialmente na implementação UNIX *System V*, sendo atualmente suportadas pela maioria dos sistemas. Além do padrão *System V*, o padrão *POSIX* também define uma interface para manipulação de filas de mensagens. Esse mecanismo é um bom exemplo de implementação do conceito de *mailbox* (vide Seção 3.3.6), permitindo o envio e recepção ordenada de mensagens tipadas entre processos locais. As operações de envio e recepção podem ser síncronas ou assíncronas, a critério do programador.

As principais chamadas para uso de filas de mensagens *POSIX* são:

- `mq_open`: abre uma fila já existente ou cria uma nova fila;
- `mq_setattr` e `mq_getattr`: permitem ajustar ou obter atributos 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-rev.c: recebe mensagens de uma fila de mensagens Posix.
2 // Em Linux, compile usando: cc -o mq-rev -lrt mq-rev.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;               // mensagens contendo um inteiro
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     umask (0);             // mascara de permissões (umask)
23
24     // abre ou cria a fila com permissões 0666
25     if ((queue = mq_open (QUEUE, O_RDWR|O_CREAT, 0666, &attr)) < 0)
26     {
27         perror ("mq_open");
28         exit (1);
29     }
30
31     // recebe cada mensagem e imprime seu conteúdo
32     for (;;)
33     {
34         if ((mq_receive (queue, (void*) &msg, sizeof(msg), 0)) < 0)
35         {
36             perror("mq_receive:");
37             exit (1);
38         }
39         printf ("Received msg value %d\n", msg);
40     }
41 }
```

A listagem a seguir implementa o programa produtor das mensagens consumidas pelo programa anterior:

```

1 // Arquivo mq-send.c: envia mensagens para uma fila de mensagens Posix.
2 // Em Linux, compile usando: cc -o mq-send -lrt mq-send.c
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <mqqueue.h>
7 #include <unistd.h>
8
9 #define QUEUE "/my_queue"
10
11 int main (int argc, char *argv[])
12 {
13     mqd_t queue;           // descritor da fila
14     int msg;               // mensagem a enviar
15
16     // abre a fila de mensagens, se existir
17     if((queue = mq_open (QUEUE, O_RDWR)) < 0)
18     {
19         perror ("mq_open");
20         exit (1);
21     }
22
23     for (;;)
24     {
25         msg = random() % 100 ; // valor entre 0 e 99
26
27         // envia a mensagem
28         if (mq_send (queue, (void*) &msg, sizeof(msg), 0) < 0)
29         {
30             perror ("mq_send");
31             exit (1);
32         }
33         printf ("Sent message with value %d\n", msg);
34         sleep (1) ;
35     }
36 }
```

O produtor de mensagens deve ser executado após o consumidor, pois é este último quem cria a fila de mensagens. Deve-se observar também que o arquivo /fila 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 3.4.3).

3.4.2 Pipes

Um dos mecanismos de comunicação entre processos mais simples de usar no ambiente UNIX é o *pipe*, ou tubo. Na interface de linha de comandos, o *pipe* é frequentemente usado para conectar a saída padrão (*stdout*) de um comando à entrada

padrão (*stdin*) de outro comando, permitindo assim a comunicação entre eles. A linha de comando a seguir traz um exemplo do uso de *pipes*:

```
# who | grep marcos | sort > login-marcos.txt
```

A saída do comando *who* é uma listagem de usuários conectados ao computador. Essa saída é encaminhada através de um *pipe* (indicado pelo caractere “|”) ao comando *grep*, que a filtra e gera como saída somente as linhas contendo a string “marcos”. Essa saída é encaminhada através de outro *pipe* ao comando *sort*, que ordena a listagem recebida e a deposita no arquivo *login-marcos.txt*. Deve-se observar que todos os processos envolvidos são lançados simultaneamente; suas ações são coordenadas pelo comportamento síncrono dos *pipes*. A Figura 3.11 detalha essa sequência de ações.

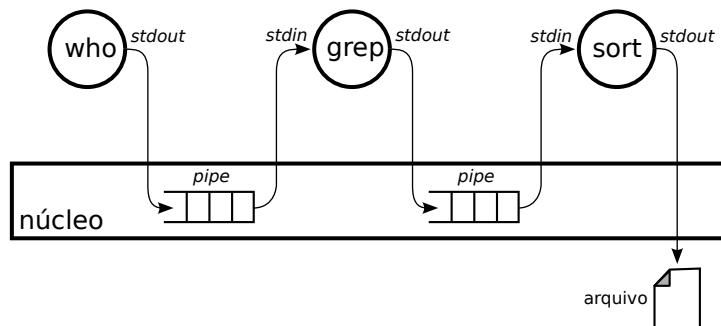


Figura 3.11: Comunicação através de *pipes*.

O *pipe* é um canal de comunicação unidirecional entre dois processos (1:1), com capacidade finita (os *pipes* do Linux armazenam 4 KBytes por default), visto pelos processos como um arquivo, ou seja, a comunicação que ele oferece é baseada em fluxo. O envio e recepção de dados são feitos pelas chamadas de sistema *write* e *read*, que podem operar em modo síncrono (bloqueante, por default) ou assíncrono.

O uso de *pipes* na linha de comando é simples, mas seu uso na construção de programas pode ser complexo. Vários exemplos do uso de *pipes* UNIX na construção de programas são apresentados em [Robbins and Robbins, 2003].

3.4.3 Memória compartilhada

A comunicação entre tarefas situadas em processos distintos deve ser feita através do núcleo, usando chamadas de sistema, porque não existe a possibilidade de acesso a variáveis comuns a ambos. No entanto, essa abordagem pode ser ineficiente caso a comunicação seja muito volumosa e frequente, ou envolva muitos processos. 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 pelo 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 utilizam são gerenciados pelo núcleo, mas o acesso ao conteúdo de cada área é feito diretamente pelos processos, sem intermediação ou

coordenação do núcleo. Por essa razão, mecanismos de coordenação (apresentados no Capítulo 4) podem ser necessários para garantir a consistência dos dados armazenados nessas áreas.

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 3.12:

1. O processo p_a solicita ao núcleo a criação de uma área de memória compartilhada, informando o tamanho e as permissões de acesso; o retorno dessa operação é um identificador (id) da área criada.
2. O processo p_a solicita ao núcleo que a área recém-criada seja anexada ao seu espaço de endereçamento; esta operação retorna um ponteiro para a nova área de memória, que pode então ser acessada pelo processo.
3. O processo p_b obtém o identificador id da área de memória criada por p_a .
4. O processo p_b solicita ao núcleo que a área de memória seja anexada ao seu espaço de endereçamento e recebe um ponteiro para o acesso à mesma.
5. Os processos p_a e p_b acessam a área de memória compartilhada através dos ponteiros informados pelo núcleo.

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 . A Listagem 3.1 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.

Listagem 3.1: Memória Compartilhada

```

1 // Arquivo shmem.c: cria e usa uma área de memória compartilhada.
2 // Em Linux, compile usando: cc -o shmem -lrt shmem.c
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <fcntl.h>
7 #include <sys/stat.h>
8 #include <sys/mman.h>
9
10 int main (int argc, char *argv[])
11 {
12     int fd, value, *ptr;
13
14     // Passos 1 a 3: abre/cria uma area de memoria compartilhada
15     fd = shm_open("/sharedmem", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
16     if(fd == -1) {
17         perror ("shm_open");
18         exit (1) ;
19     }
20
21     // Passos 1 a 3: ajusta o tamanho da area compartilhada
22     if (ftruncate(fd, sizeof(value)) == -1) {
23         perror ("ftruncate");
24         exit (1) ;
25     }
26
27     // Passos 2 a 4: mapeia a area no espaco de enderecamento deste processo
28     ptr = mmap(NULL, sizeof(value), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
29     if(ptr == MAP_FAILED) {
30         perror ("mmap");
31         exit (1);
32     }
33
34     for (;;) {
35         // Passo 5: escreve um valor aleatorio na area compartilhada
36         value = random () % 1000 ;
37         (*ptr) = value ;
38         printf ("Wrote value %i\n", value) ;
39         sleep (1);
40
41         // Passo 5: le e imprime o conteudo da area compartilhada
42         value = (*ptr) ;
43         printf("Read value %i\n", value);
44         sleep (1) ;
45     }
46 }
```

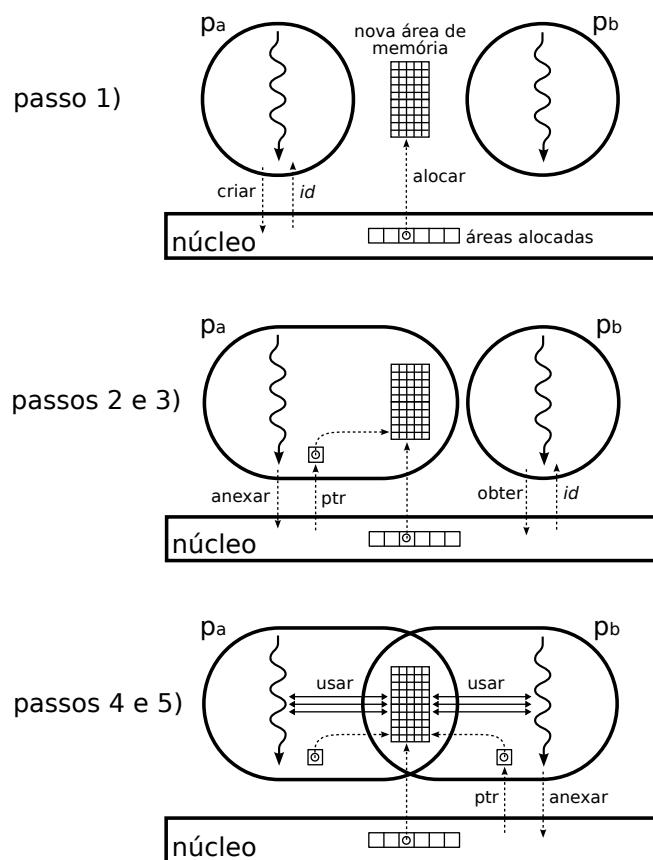


Figura 3.12: Criação e uso de uma área de memória compartilhada.

Capítulo 4

Coordenação entre tarefas

Muitas implementações de sistemas complexos são estruturadas como várias tarefas inter-dependentes, 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 uma 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 coordenação entre tarefas.

4.1 Objetivos

Em um sistema multi-tarefas, várias tarefas podem executar simultaneamente, acessando recursos compartilhados como áreas de memória, arquivos, conexões de rede, etc. Neste capítulo serão estudados os problemas que podem ocorrer quando duas ou mais tarefas acessam os mesmos recursos de forma concorrente; também serão apresentadas as principais técnicas usadas para coordenar de forma eficiente os acessos das tarefas aos recursos compartilhados.

4.2 Condições de disputa

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.

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

```

1 typedef struct conta_t
2 {
3     int saldo ;      // saldo atual da conta
4     ...             // outras informações da conta
5 } conta_t ;
6
7 void depositar (conta_t* conta, int valor)
8 {
9     conta->saldo += valor ;
10}

```

Após a compilação em uma plataforma *Intel i386*, a função `depositar` assume a seguinte forma em código de máquina (nos comentários ao lado do código, reg_i é um registrador e $mem(x)$ é a posição de memória onde está armazenada a variável x):

```

00000000 <depositar>:
push %ebp          # guarda o valor do "stack frame"
mov  %esp,%ebp    # ajusta o stack frame para executar a função

mov  0x8(%ebp),%ecx   # mem(saldo) → reg1
mov  0x8(%ebp),%edx   # mem(saldo) → reg2
mov  0xc(%ebp),%eax   # mem(valor) → reg3
add   (%edx),%eax      # [reg1,reg2] + reg3 → [reg1,reg2]
mov  %eax,(%ecx)       # [reg1,reg2] → mem(saldo)

leave              # restaura o stack frame anterior
ret               # retorna à função anterior

```

Considere que a função `depositar` faz parte de um sistema mais amplo de controle de contas bancárias, que pode ser acessado simultaneamente por centenas ou milhares de usuários em terminais distintos. Caso dois clientes em terminais diferentes tentem depositar valores na mesma conta ao mesmo tempo, existirão duas tarefas acessando os dados (variáveis) da conta de forma concorrente. A Figura 4.1 ilustra esse cenário.

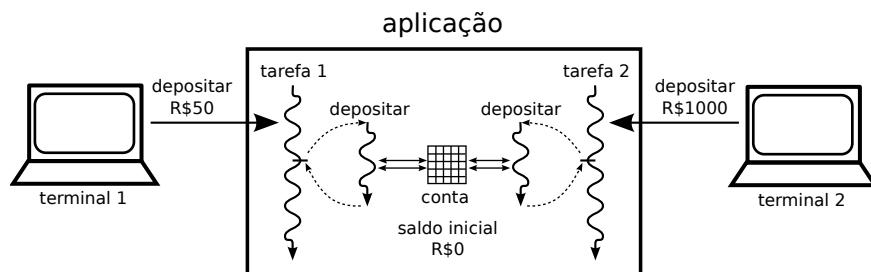


Figura 4.1: Acessos concorrentes a variáveis compartilhadas.

O comportamento dinâmico da aplicação 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 4.2. Em ambas as execuções o saldo inicial da conta passou de R\$ 0,00 para R\$ 1050,00, conforme esperado.

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 mono-

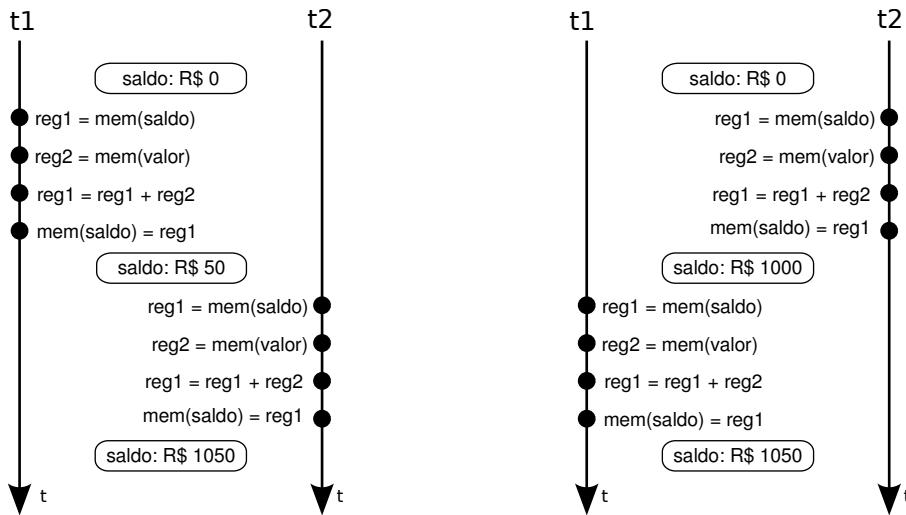


Figura 4.2: Operações de depósitos não-concorrentes.

processados, a sobreposição pode acontecer caso ocorram trocas de contexto durante a execução do depósito. Em sistemas multi-processados a situação é ainda mais complexa, pois cada tarefa poderá estar executando em um processador distinto.

Os diagramas de tempo apresentados na Figura 4.3 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. No caso, apenas é concretizado o depósito da tarefa que realizou a operação $mem(saldo) \leftarrow reg1$ por último¹.

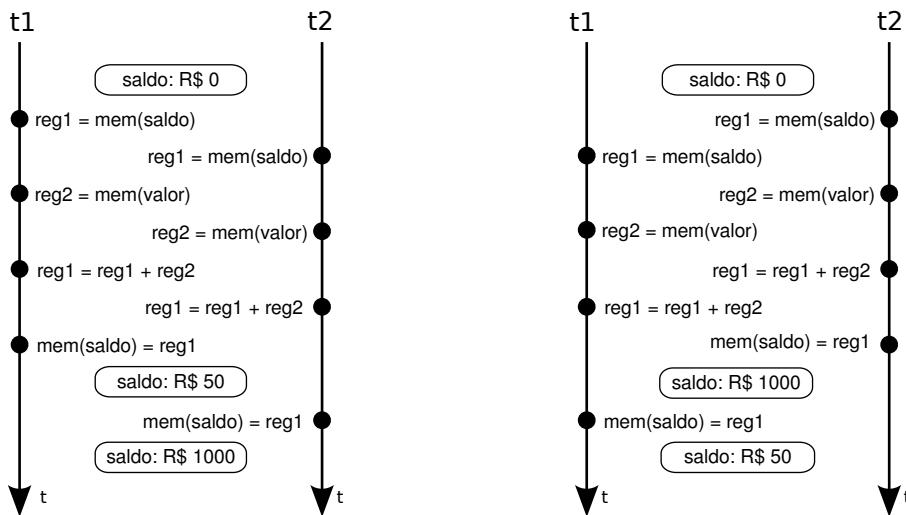


Figura 4.3: Operações de depósito concorrentes.

Os erros e inconsistências gerados por acessos concorrentes a dados compartilhados, como os ilustrados na Figura 4.3, são denominados **condições de disputa**, ou condições

¹Não há problema em ambas as tarefas usarem os mesmos registradores $reg1$ e $reg2$, pois os valores de todos os registradores são salvos/restaurados a cada troca de contexto entre tarefas.

de corrida (do inglês *race conditions*). Condições de disputa podem ocorrer em qualquer sistema onde várias tarefas (processos ou *threads*) acessam de forma concorrente recursos compartilhados (variáveis, áreas de memória, arquivos abertos, etc.). Finalmente, condições de disputa somente existem caso ao menos uma das operações envolvidas seja de escrita; acessos de leitura concorrentes entre si não geram condições de disputa.

É importante observar que condições de disputa são erros *dinâmicos*, ou seja, que não aparecem no código fonte e que só se manifestam durante a execução, sendo dificilmente detectáveis através da 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 e ser difícil de reproduzir durante a depuração. Por isso, é importante conhecer técnicas que previnam a ocorrência de condições de disputa.

4.3 Seções críticas

Na seção anterior vimos que tarefas acessando dados compartilhados de forma concorrente podem ocasionar condições de disputa. Os trechos de código de cada tarefa que acessam dados compartilhados são denominados **seções críticas** (ou *regiões críticas*). No caso da Figura 4.1, as seções críticas das tarefas t_1 e t_2 são idênticas e resumidas à seguinte linha de código:

```
1 conta.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: apenas uma tarefa pode estar na seção crítica a cada instante. Essa propriedade é conhecida como **exclusão mútua**.

Diversos mecanismos podem ser definidos para impedir o entrelaçamento de seções críticas e assim prover a exclusão mútua. Todos eles exigem que o programador defina os limites (início e o final) de cada seção crítica. Genericamente, cada seção crítica i pode ser associada a um identificador cs_i e são definidas as primitivas $enter(t_a, cs_i)$, para que a tarefa t_a indique que deseja entrar na seção crítica cs_i , e $leave(t_a, cs_i)$, para que t_a informe que está saindo da seção crítica cs_i . 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 a seção crítica, através da primitiva $leave(cs_i)$.

Usando as primitivas *enter* e *leave*, o código da operação de depósito visto na Seção 4.2 pode ser reescrito como segue:

```

1 typedef struct conta_t
2 {
3     int saldo ;           // saldo atual da conta
4     int numero ;        // identificação da conta (seção crítica)
5     ...
6 } conta_t ;
7
8 void depositar (conta_t* conta, int valor)
9 {
10    enter (conta->numero) ; // tenta entrar na seção crítica
11    conta->saldo += valor ; // está na seção crítica
12    leave (conta->numero) ; // sai da seção crítica
13 }

```

Nas próximas seções serão estudadas várias soluções para a implementação das primitivas *enter* e *leave*, bem como abordagens alternativas. As soluções propostas devem atender a alguns critérios básicos que são 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.

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 do sistema, que no momento não estejam interessadas em entrar na região crítica, não podem ter influência 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.

4.4 Inibição de interrupções

Uma solução simples para a implementação das primitivas *enter* e *leave* consiste em impedir as trocas de contexto dentro da seção crítica. Ao entrar em uma seção crítica, a tarefa desativa (mascara) 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. Uma tarefa mal-intencionada pode forçar essa situação 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 mono-processados; em uma máquina multi-processada ou multi-core, 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.

4.5 Soluções com espera ocupada

Uma primeira classe de soluções para o problema da exclusão mútua no acesso a seções críticas consiste em testar continuamente uma condição que indica se a seção desejada está livre ou ocupada. Esta seção apresenta algumas soluções clássicas usando essa abordagem.

4.5.1 A solução óbvia

Uma solução aparentemente trivial para o problema da seção crítica consiste em usar uma variável *busy* para indicar se a seção crítica desejada 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 (int task)
4 {
5     while (busy) ;       // espera enquanto a seção estiver ocupada
6     busy = 1 ;           // marca a seção como ocupada
7 }
8
9 void leave (int task)
10{
11    busy = 0 ;           // libera a seção (marca como livre)
12}
```

Infelizmente, essa solução óbvia e 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 (vide o diagrama de tempo da Figura 4.4). Em outras palavras, as linhas 5 e 6 da implementação também formam uma seção crítica, que deve ser protegida.

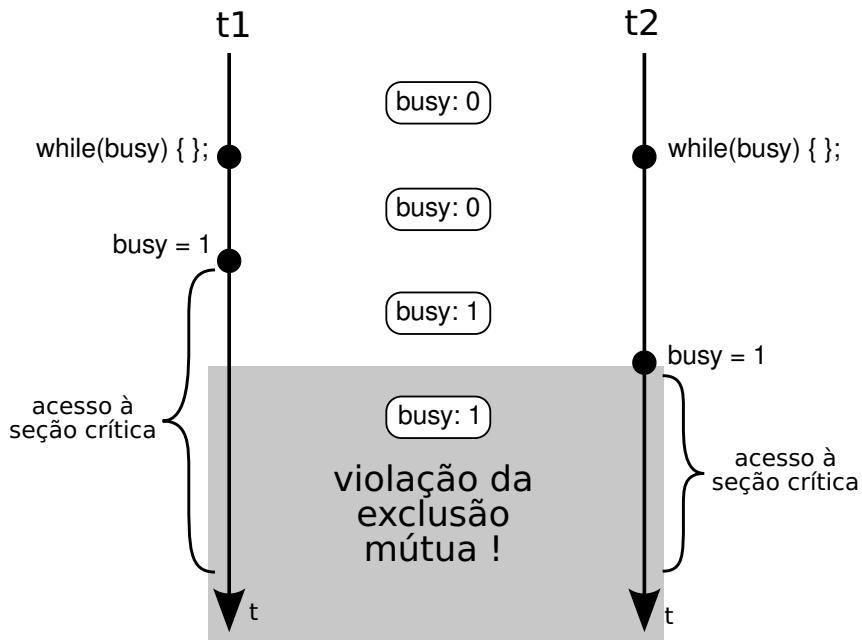


Figura 4.4: Condição de disputa no acesso à variável *busy*.

4.5.2 Alternância de uso

Outra solução simples para a implementação das primitivas *enter* e *leave* 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 turn = 0 ;
2 int num_tasks ;

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     if (turn < num_tasks-1) // o turno é da próxima tarefa
12         turn ++ ;
13     else
14         turn = 0 ;           // volta à primeira tarefa
15 }
```

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

4.5.3 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 é a 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 limitada². Diversas generalizações para n tarefas podem ser encontradas na literatura [Raynal, 1986], sendo a mais conhecida delas o *algoritmo da padaria*, proposto por Leslie Lamport [Lamport, 1974].

4.5.4 Instruções *Test-and-Set*

O uso de uma variável *busy* para controlar a entrada em uma seção crítica é uma idéia interessante, que infelizmente não funciona porque o teste da variável *busy* e seu ajuste são feitos em momentos distintos do código, permitindo condições de corrida.

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* ou indivisível, sem possibilidade de troca de contexto entre essas duas operações. A execução atômica das operações de teste e atribuição (*Test & Set instructions*) impede a ocorrência de condições de disputa. Uma implementação básica dessa idéia está na instrução de máquina *Test-and-Set Lock* (TSL), cujo comportamento é descrito pelo seguinte pseudo-código (que é executado atomicamente pelo processador):

²Este algoritmo **pode falhar** quando usado em algumas máquinas multi-núcleo ou multi-processadas, pois algumas arquiteturas permitem acesso fora de ordem à memória, ou seja, permitem que operações de leitura na memória se antecipem a operações de escrita executadas posteriormente, para obter mais desempenho. Este é o caso dos processadores Pentium e AMD.

$$\begin{aligned} TSL(x) : \quad & old \leftarrow x \\ & x \leftarrow 1 \\ & return(old) \end{aligned}$$

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
6 }
7
8 void leave (int *lock)
9 {
10    (*lock) = 0 ;          // libera a seção crítica
11 }
```

Outros processadores oferecem uma instrução que efetua a troca atômica de conteúdo (*swapping*) entre dois registradores, ou entre um registrador e uma posição de memória. No caso da família de processadores Intel i386 (incluindo o Pentium e seus sucessores), a instrução de troca se chama XCHG (do inglês *exchange*) e sua funcionalidade pode ser resumida assim:

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

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 no estilo TSL são amplamente usados no interior do sistema operacional, para controlar o acesso a seções críticas internas 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 a seguir.

4.5.5 Problemas

Apesar das soluções para o problema de acesso à seção crítica usando espera ocupada garantirem a exclusão mútua, elas sofrem de alguns 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 : 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.

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 (onde se chamam *spinlocks*), e na construção de sistemas de computação dedicados, como controladores embarcados mais simples.

4.6 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 implícita (na construção de mecanismos de coordenação mais abstratos, como os monitores).

Um semáforo pode ser visto como uma variável s , que representa uma seção crítica e cujo conteúdo não é diretamente acessível ao programador. Internamente, cada semáforo contém um contador inteiro $s.counter$ e uma fila de tarefas $s.queue$, inicialmente vazia. Sobre essa variável podem ser aplicadas duas operações atômicas, descritas a seguir:

Down(s) : usado para solicitar acesso à seção crítica associada a s . Caso a seção esteja livre, a operação retorna imediatamente e a tarefa pode continuar sua execução; caso contrário, a tarefa solicitante é suspensa e adicionada à fila do semáforo; o contador associado ao semáforo é decrementado³. Dijkstra denominou essa operação $P(s)$ (do holandês *probeer*, que significa *tentar*).

³Alguns sistemas implementam também a chamada *TryDown(s)*, cuja semântica é não-bloqueante: caso o semáforo solicitado esteja ocupado, a chamada retorna imediatamente, com um código de erro.

```

Down(s): // a executar de forma atômica
s.counter ← s.counter - 1
if s.counter < 0 then
    põe a tarefa corrente no final de s.queue
    suspende a tarefa corrente
end if

```

Up(s) : invocado para liberar a seção crítica associada a *s*; 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 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.

```

Up(s): // a executar de forma atômica
s.counter ← s.counter + 1
if s.counter ≤ 0 then
    retira a primeira tarefa t de s.queue
    devolve t à fila de tarefas prontas (ou seja, acorda t)
end if

```

As operações de acesso aos semáforos são geralmente implementadas pelo núcleo do sistema operacional, na forma de chamadas de sistema. É importante observar que a execução das operações *Down(s)* e *Up(s)* deve ser **atômica**, ou seja, não devem ocorrer acessos concorrentes às variáveis internas do semáforo, para evitar condições de disputa sobre as mesmas. Para garantir a atomicidade dessas operações em um sistema monoprocessador, seria suficiente inibir as interrupções durante a execução das mesmas; no caso de sistemas com mais de um núcleo, torna-se necessário usar outros mecanismos de controle de concorrência, como operações TSL, para proteger a integridade interna do semáforo. Nestes casos, 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 4.2 poderia ser reescrito da seguinte forma:

```

1 typedef struct conta_t
2 {
3     int saldo ;           // saldo atual da conta
4     sem_t s = 1;         // semáforo associado à conta, valor inicial 1
5     ...
6 } conta_t ;
7
8 void depositar (conta_t * conta, int valor)
9 {
10    down (conta->s) ;   // solicita acesso à conta
11    conta->saldo += valor ; // seção crítica
12    up (conta->s) ;     // libera o acesso à conta
13 }

```

A suspensão das tarefas que aguardam o acesso à seção crítica elimina a espera ocupada, o que torna esse mecanismo mais eficiente no uso do processador que os anteriores. A fila de tarefas associada ao semáforo contém todas as tarefas que foram suspensas ao solicitar acesso à seção crítica usando a chamada *Down(s)*. Como a fila obedece uma política FIFO, garante-se a também a justiça no acesso à seção crítica, pois todos os processos que aguardam no semáforo serão atendidos em sequência⁴. Por sua vez, 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 para usar aquele recurso. Seu valor inicial permite expressar diferentes situações de sincronização, como será visto na Seção 4.9.

A listagem a seguir apresenta um exemplo hipotético de uso de um semáforo para controlar o acesso a um estacionamento. O valor inicial do semáforo *vagas* representa o número de vagas inicialmente livres no estacionamento (500). 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, a chamada *carro_entra()* ficará bloqueada até que alguma vaga seja liberada, o que ocorre quando outro carro acionar a chamada *carro_sai()*. Esta solução simples pode ser aplicada a um estacionamento com várias entradas e várias saídas simultâneas.

```

1 sem_t vagas = 500 ;
2
3 void carro_entra ()
4 {
5     down (vagas) ;           // solicita uma vaga de estacionamento
6     ...                      // demais ações específicas da aplicação
7 }
8
9 void carro_sai ()
10 {
11    up (vagas) ;            // libera uma vaga de estacionamento
12    ...                      // demais ações específicas da aplicação
13 }
```

A API POSIX define várias chamadas para a criação e manipulação de semáforos. As chamadas mais frequentemente utilizadas estão indicadas a seguir:

⁴Algumas 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.

```

1 #include <semaphore.h>
2
3 // inicializa um semáforo apontado por "sem", com valor inicial "value"
4 int sem_init(sem_t *sem, int pshared, unsigned int value);
5
6 // Operação Up(s)
7 int sem_post(sem_t *sem);
8
9 // Operação Down(s)
10 int sem_wait(sem_t *sem);
11
12 // Operação TryDown(s), retorna erro se o semáforo estiver ocupado
13 int sem_trywait(sem_t *sem);

```

Os semáforos nos quais o contador inteiro pode assumir qualquer valor são denominados **semáforos genéricos** e constituem um mecanismo de coordenação muito poderoso. No entanto, Muitos ambientes de programação, bibliotecas de threads e até mesmo núcleos de sistema provêem uma versão simplificada de semáforos, na qual o contador só assume dois valores possíveis: *livre* (1) ou *ocupado* (0). Esses semáforos simplificados são chamados de **mutexes** (uma abreviação de *mutual exclusion*) ou semáforos binários. Por exemplo, algumas das funções definidas pelo padrão POSIX [Gallmeister, 1994, Barney, 2005] para criar e usar mutexes são:

```

1 #include <pthread.h>
2
3 // inicializa uma variável do tipo mutex, usando um struct de atributos
4 int pthread_mutex_init (pthread_mutex_t *restrict mutex,
5                         const pthread_mutexattr_t *restrict attr);
6
7 // destrói uma variável do tipo mutex
8 int pthread_mutex_destroy (pthread_mutex_t *mutex);
9
10 // solicita acesso à seção crítica protegida pelo mutex;
11 // se a seção estiver ocupada, bloqueia a tarefa
12 int pthread_mutex_lock (pthread_mutex_t *mutex);
13
14 // solicita acesso à seção crítica protegida pelo mutex;
15 // se a seção estiver ocupada, retorna com status de erro
16 int pthread_mutex_trylock (pthread_mutex_t *mutex);
17
18 // libera o acesso à seção crítica protegida pelo mutex
19 int pthread_mutex_unlock (pthread_mutex_t *mutex);

```

4.7 Variáveis de condição

Além dos semáforos, outro mecanismo de sincronização de uso frequente são as *variáveis de condição*, ou variáveis condicionais. Uma variável de condição não representa um valor, mas uma condição, que pode ser aguardada por uma tarefa. Quando uma tarefa aguarda uma condição, ela é colocada para dormir até que a condição seja

verdadeira. Assim, a tarefa não precisa testar continuamente aquela condição, evitando uma espera ocupada.

Uma tarefa aguardando uma condição representada pela variável de condição c pode ficar suspensa através do operador $wait(c)$, para ser notificada mais tarde, quando a condição se tornar verdadeira. Essa notificação ocorre quando outra tarefa chamar o operador $notify(c)$ (também chamado $signal(c)$). Por definição, uma variável de condição c está sempre associada a um semáforo binário $c.mutex$ e a uma fila $c.queue$. O $mutex$ garante a exclusão mútua sobre a condição representada pela variável de condição, enquanto a fila serve para armazenar em ordem as tarefas que aguardam aquela condição.

Uma implementação hipotética⁵ para as operações $wait$, $notify$ e $broadcast$ (que notifica todas as tarefas na espera da condição) para uma tarefa t , seria a seguinte:

```

1 wait ( $c$ ):
2    $c.queue \leftarrow t$            // coloca a tarefa  $t$  no fim de  $c.queue$ 
3   unlock ( $c.mutex$ )        // libera o  $mutex$ 
4   suspend ( $t$ )            // põe a tarefa atual para dormir
5   lock ( $c.mutex$ )         // quando acordar, obtém o  $mutex$  imediatamente
6
7 notify ( $c$ ):
8   awake (first ( $c.queue$ )) // acorda a primeira tarefa da fila  $c.queue$ 
9
10 broadcast ( $c$ ):
11   awake ( $c.queue$ )       // acorda todas as tarefas da fila  $c.queue$ 
```

No exemplo a seguir, a tarefa A espera por uma condição que será sinalizada pela tarefa B . A condição de espera pode ser qualquer: um novo dado em um buffer de entrada, a conclusão de um procedimento externo, a liberação de espaço em disco, etc.

⁵Assim como os operadores sobre semáforos, os operadores sobre variáveis de condição também devem ser implementados de forma atômica.

```

1 Task A ()
2 {
3     ...
4     lock (c.mutex)
5     while (not condition)
6         wait (c) ;
7     ...
8     unlock (c.mutex)
9     ...
10 }
11
12 Task B ()
13 {
14     ...
15     lock (c.mutex)
16     condition = true
17     notify (c)
18     unlock (c.mutex)
19     ...
20 }
```

É importante observar que na definição original de variáveis de condição (conhecida como *semântica de Hoare*), a operação *notify(c)* fazia com que a tarefa notificadora perdesse imediatamente o semáforo e o controle do processador, que eram devolvidos à primeira tarefa da fila de *c*. Como essa semântica é complexa de implementar e interfere diretamente no escalonador de processos, as implementações modernas de variáveis de condição normalmente adotam a *semântica Mesa* [Lampson and Redell, 1980], proposta na linguagem de mesmo nome. Nessa semântica, a operação *notify(c)* apenas “acorda” as tarefas que esperam pela condição, sem suspender a execução da tarefa corrente. Cabe ao programador garantir que a tarefa corrente vai liberar o *mutex* e não vai alterar o estado associado à 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*.

4.8 Monitores

Ao usar semáforos, um programador define 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 4.10). 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 de:

- um recurso compartilhado, visto como um conjunto de variáveis internas ao monitor.
- um conjunto de procedimentos 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 semáforo antes de iniciar e liberar o semáforo ao concluir;
- um invariante sobre o estado interno do recurso.

O pseudo-código 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):

```

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

A definição formal de monitor prevê a 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 e todas as retiradas efetuadas (com sinal negativo)*”. 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 entanto, a execução dos procedimentos é feita com exclusão mútua entre eles. As operações de obtenção e liberação do semáforo 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. O código a seguir mostra um exemplo simplificado de uso de monitor em Java:

```

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.out.println("valor negativo");
11    }
12
13    public synchronized void retirar (float valor)
14    {
15        if (valor >= 0)
16            saldo -= valor ;
17        else
18            System.out.println("valor negativo");
19    }
20 }
```

Em Java, a cláusula `synchronized` faz com que um semáforo seja associado aos métodos indicados, para cada objeto (ou para cada classe, se forem métodos de classe). No exemplo anterior, apenas um depósito ou retirada de cada vez poderá ser feito sobre cada objeto da classe `Conta`.

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 `notify` 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].

4.9 Problemas clássicos de coordenação

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. Nesta seção serão estudados três 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, 2008] (disponível *online*).

4.9.1 O problema dos 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 simétricos:

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

Consumidor : continuamente 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.

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 4.5 ilustra esse problema, envolvendo vários produtores e consumidores acessando um buffer com capacidade para 12 entradas. É interessante observar a forte similaridade dessa figura com a Figura 3.9; na prática, a implementação de *mailboxes* e de *pipes* é geralmente feita usando um esquema de sincronização produtor/consumidor.

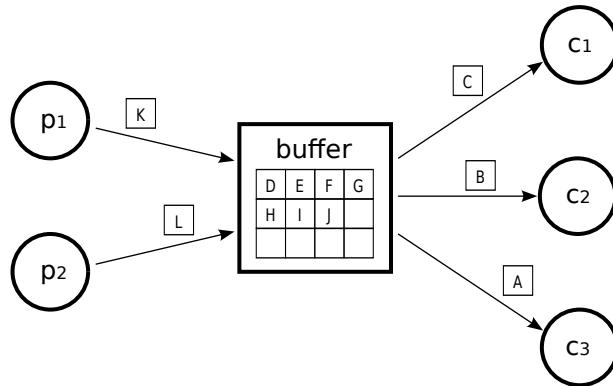


Figura 4.5: O problema dos produtores/consumidores.

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

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

A solução para esse problema exige três semáforos, um para atender cada aspecto de coordenação acima descrito. O código a seguir ilustra de forma simplificada uma solução para esse problema, considerando um buffer com capacidade para N itens, inicialmente vazio:

```

1 sem_t mutex ; // controla o acesso ao buffer (inicia em 1)
2 sem_t item ; // número de itens no buffer (inicia em 0)
3 sem_t vaga ; // número de vagas livres no buffer (inicia em N)
4
5 produtor () {
6     while (1) {
7         ...
8         sem_down(&vaga) ; // aguarda uma vaga no buffer
9         sem_down(&mutex) ; // aguarda acesso exclusivo ao buffer
10        ...
11        sem_up(&mutex) ; // deposita o item no buffer
12        sem_up(&item) ; // libera o acesso ao buffer
13    }
14 }
15
16 consumidor () {
17     while (1) {
18         sem_down(&item) ; // aguarda um novo item no buffer
19         sem_down(&mutex) ; // aguarda acesso exclusivo ao buffer
20         ...
21         sem_up(&mutex) ; // retira o item do buffer
22         sem_up(&vaga) ; // libera o acesso ao buffer
23         ...
24     }
25 }
```

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

4.9.2 O problema dos leitores/escritores

Outra situação que ocorre com frequência em sistemas concorrentes é o problema dos *leitores/escritores*. Neste caso, um conjunto de processos ou threads acessam de forma concorrente uma área de memória comum (compartilhada), na qual podem fazer leituras ou escritas de valores. As leituras podem ser feitas simultaneamente, 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. No exemplo da Figura 4.6, os leitores e escritores acessam de forma concorrente uma matriz de inteiros M .

Uma solução trivial para esse problema consistiria em proteger o acesso à área compartilhada com um semáforo inicializado em 1; assim, somente um processo por vez poderia acessar a área, garantindo a integridade de todas as operações. O código a seguir ilustra essa abordagem simplista:

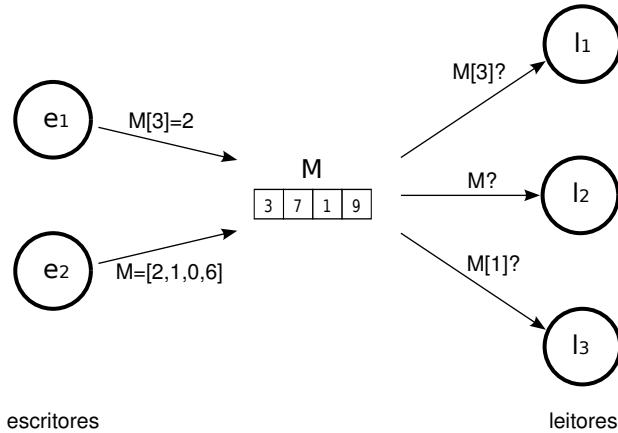


Figura 4.6: O problema dos leitores/escritores.

```

1 sem_t mutex_area ;           // controla o acesso à área (inicia em 1)
2
3 leitor () {
4     while (1) {
5         sem_down (&mutex_area) ;    // requer acesso exclusivo à área
6         ...
7         sem_up (&mutex_area) ;    // libera o acesso à área
8         ...
9     }
10 }
11
12 escritor () {
13     while (1) {
14         sem_down (&mutex_area) ;    // requer acesso exclusivo à área
15         ...
16         sem_up (&mutex_area) ;    // libera o acesso à área
17         ...
18     }
19 }
```

Todavia, essa solução deixa 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 simultâneo de vários leitores à área compartilhada, desde que as escritas continuem sendo feitas de forma exclusiva. Uma nova solução para o problema, considerando a possibilidade de acesso simultâneo pelos leitores, seria a seguinte:

```

1 sem_t mutex_area ;           // controla o acesso à área (inicia em 1)
2 int conta_leitores = 0 ;     // número de leitores acessando a área
3 sem_t mutex_conta ;         // controla o acesso ao contador (inicia em 1)
4
5 leitor () {
6     while (1) {
7         sem_down (&mutex_conta) ;    // requer acesso exclusivo ao contador
8         conta_leitores++ ;        // incrementa contador de leitores
9         if (conta_leitores == 1)   // sou o primeiro leitor a entrar?
10            sem_down (&mutex_area) ; // requer acesso à área
11            sem_up (&mutex_conta) ; // libera o contador
12
13            ...                   // lê dados da área compartilhada
14
15            sem_down (&mutex_conta) ; // requer acesso exclusivo ao contador
16            conta_leitores-- ;    // decrementa contador de leitores
17            if (conta_leitores == 0) // sou o último leitor a sair?
18                sem_up (&mutex_area) ; // libera o acesso à área
19                sem_up (&mutex_conta) ; // libera o contador
20            ...
21     }
22 }
23
24 escritor () {
25     while (1) {
26         sem_down(&mutex_area) ;    // requer acesso exclusivo à área
27         ...
28         sem_up(&mutex_area) ;    // libera o acesso à área
29         ...
30     }
31 }
```

Essa solução melhora o desempenho das operações de leitura, mas 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. Soluções com priorização para os escritores e soluções equitativas entre ambos podem ser facilmente encontradas na literatura [Raynal, 1986, Ben-Ari, 1990].

O relacionamento de sincronização *leitor/escritor* 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 (com priorização de escritores), acessíveis através de chamadas como `pthread_rwlock_init`, entre outras.

4.9.3 O jantar dos filósofos

Um dos problemas clássicos de coordenação mais conhecidos é o *jantar dos filósofos*, que foi inicialmente proposto por Dijkstra [Raynal, 1986, Ben-Ari, 1990]. Neste problema, um grupo de cinco filósofos chineses alterna suas vidas entre meditar e comer. Na mesa há um lugar fixo para cada filósofo, com um prato, cinco palitos (*hashis* ou *chopsticks*)

compartilhados e um grande prato de comida ao meio (na versão inicial de Dijkstra, os filósofos compartilhavam garfos e comiam spaghetti). A Figura 4.7 ilustra essa situação.

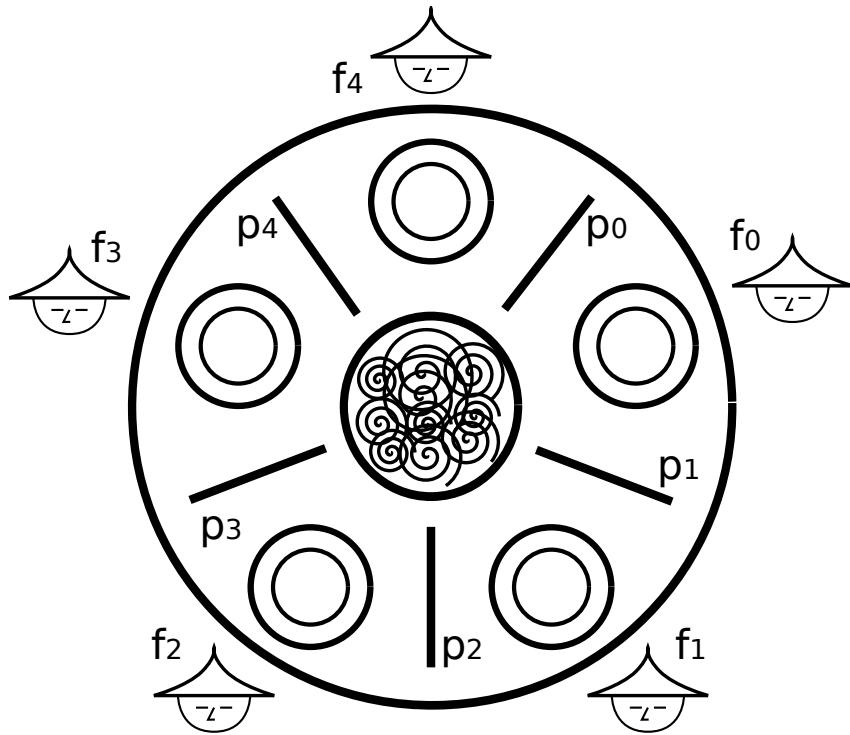


Figura 4.7: O jantar dos filósofos chineses.

Para comer, um filósofo f_i precisa pegar os palitos à sua direita (p_i) e à sua esquerda (p_{i+1}), um de cada vez. Como os palitos são compartilhados, dois filósofos vizinhos nunca podem comer ao mesmo tempo. Os filósofos não conversam entre si nem podem observar os estados uns dos outros. 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. A listagem a seguir representa uma implementação do comportamento básico dos filósofos, na qual cada palito é representado por um semáforo:

```

1 #define NUMFILO 5
2 sem_t hashi [NUMFILO] ; // um semáforo para cada palito (iniciais em 1)
3
4 filosofo (int i) {
5     while (1) {
6         medita () ;
7         sem_down (&hashi [i]) ; // obtém palito i
8         sem_down (&hashi [(i+1) % NUMFILO]) ; // obtém palito i+1
9         come () ;
10        sem_up (&hashi [i]) ; // devolve palito i
11        sem_up (&hashi [(i+1) % NUMFILO]) ; // devolve palito i+1
12    }
13 }
```

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. As soluções mais simples para esse problema podem provocar impasses, nos quais todos os filósofos ficam bloqueados (impasses serão estudados na Seção 4.10). Outras soluções podem provocar inanição (*starvation*), ou seja, alguns dos filósofos nunca conseguem comer. A Figura 4.8 apresenta os filósofos em uma situação de impasse: cada filósofo obteve o palito à sua direita e está aguardando o palito à sua esquerda (indicado pelas setas tracejadas). Como todos os filósofos estão aguardando, ninguém mais consegue executar.

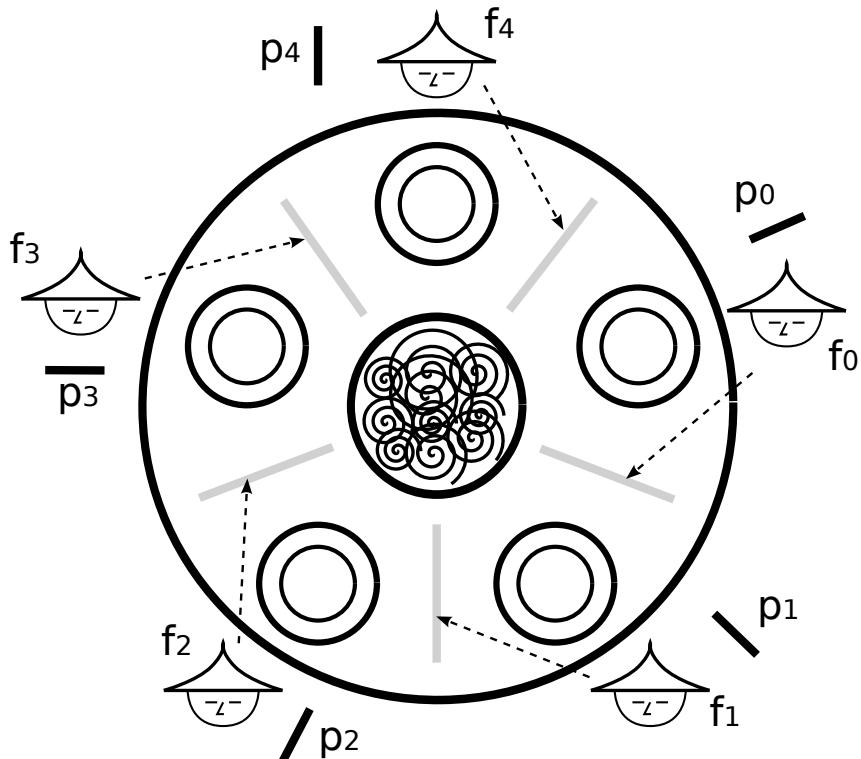


Figura 4.8: Um impasse no jantar dos filósofos chineses.

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 pegar o saleiro antes de obter os palitos; assim que tiver ambos os palitos, ele devolve o saleiro à mesa e pode comer. Obviamente, essa solução serializa o acesso aos palitos e por isso tem baixo desempenho. Imagine como essa solução funcionaria em uma situação com 1000 filósofos e 1000 palitos? Diversas outras soluções podem ser encontradas na literatura [Tanenbaum, 2003, Silberschatz et al., 2001].

4.10 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 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. Para ilustrar uma situação de impasse, será utilizado o exemplo de acesso a uma conta bancária apresentado na Seção 4.2. O código a seguir implementa uma operação de transferência de fundos entre duas contas bancárias. A cada conta está associado um semáforo, 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     sem_t lock ;                   // semáforo associado à conta
5     ...
6 } conta_t ;
7
8 void transferir (conta_t* contaDeb, conta_t* contaCred, int valor)
9 {
10    sem_down (contaDeb->lock) ;      // obtém acesso a contaDeb
11    sem_down (contaCred->lock) ;      // obtém acesso a contaCred
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    sem_up (contaDeb->lock) ;       // libera acesso a contaDeb
19    sem_up (contaCred->lock) ;      // 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 4.9.

Nessa situação, a tarefa t_1 detém o semáforo de c_1 e solicita o semáforo de c_2 , enquanto t_2 detém o semáforo de c_2 e solicita o semáforo de c_1 . Como nenhuma das duas tarefas poderá prosseguir sem obter o semáforo desejado, nem poderá liberar o semáforo de sua conta antes de obter o outro semáforo e realizar a transferência, se estabelece um impasse (ou *deadlock*).

Impasses são situações muito frequentes em programas concorrentes, mas também podem ocorrer em sistemas distribuídos. 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.

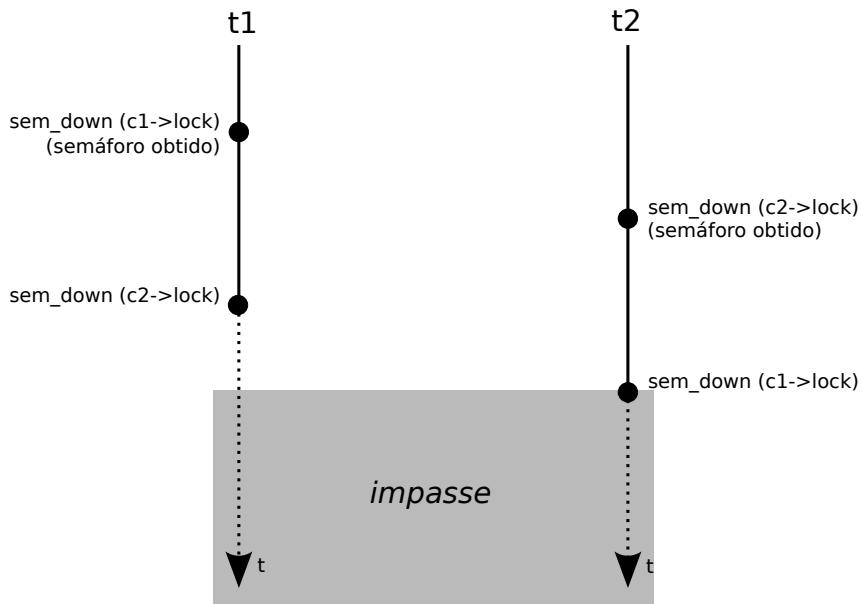


Figura 4.9: Impasse entre duas transferências.

4.10.1 Caracterização de 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 requisitá-los 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]:

C1 – 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.

C2 – 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.

C3 – Não-preempção : uma tarefa somente libera os recursos que detém quando assim o decidir, e não pode perdê-los contra a sua vontade (ou seja, o sistema operacional não retira os recursos já alocados às tarefas). No exemplo da conta corrente, cada tarefa detém indefinidamente os semáforos que já obteve.

C4 – 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: $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \dots \rightarrow t_n \rightarrow t_1$. 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 existirão 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.

4.10.2 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 (\circ) e os recursos por retângulos (\square). A posse de um recurso por uma tarefa é representada como $\square \rightarrow \circ$, enquanto a requisição de um recurso por uma tarefa é indicada por $\circ \rightarrow \square$.

A Figura 4.10 apresenta o grafo de alocação de recursos da situação de impasse ocorrida na transferência de valores entre contas bancárias da Figura 4.9. Nessa figura percebe-se claramente a dependência cíclica entre tarefas e recursos $t_1 \rightarrow c_2 \rightarrow t_2 \rightarrow c_1 \rightarrow t_1$, 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.

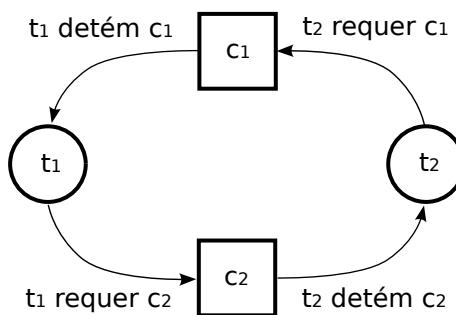


Figura 4.10: Grafo de alocação de recursos com impasse.

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 $\boxed{\bullet \bullet}$. A Figura 4.11 indica apresenta um grafo de alocação de recursos considerando alguns recursos com múltiplas instâncias.

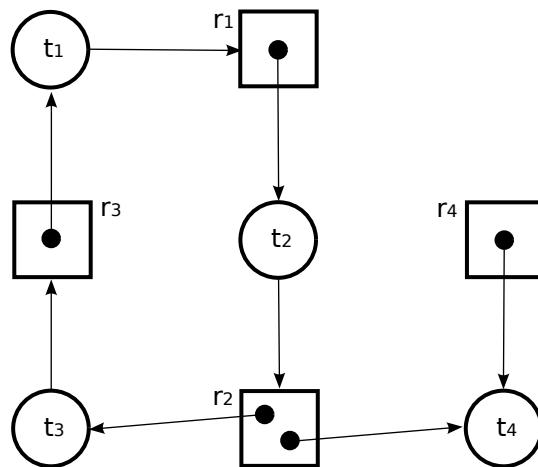


Figura 4.11: Grafo de alocação com múltiplas instâncias de recursos.

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 $t_1 \rightarrow r_1 \rightarrow t_2 \rightarrow r_2 \rightarrow t_3 \rightarrow r_3 \rightarrow t_1$ presente no diagrama da Figura 4.11 não representa um impasse, porque a qualquer momento a tarefa t_4 pode liberar uma instância do recurso r_2 , solicitado por t_2 , desfazendo assim o ciclo. Um algoritmo de detecção de impasses envolvendo recursos com múltiplas instâncias é apresentado em [Tanenbaum, 2003].

4.10.3 Técnicas de tratamento de impasses

Como os impasses paralisam tarefas que detêm recursos, sua ocorrência pode incorrer em 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 pró-ativa, 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, Solaris, 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.

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 4.10.1, nunca sejam 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 esse recurso. A técnica de *spooling* previne impasses envolvendo as impressoras, mas não é facilmente aplicável a outros 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 4.9, 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, e devolvê-lo mais tarde, 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 contexto do processador. No entanto, é de difícil aplicação sobre recursos como arquivos ou áreas de memória compartilhada, porque a preempção viola a exclusão mútua e pode deixar 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 4.9, o número de conta bancária poderia definir uma ordem global. 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.

Impedimento de impasses

Uma 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 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 **inseguro**. As técnicas de impedimento de impasses devem portanto manter o sistema sempre em um estado seguro, evitando entrar em estados inseguros.

A Figura 4.12 ilustra o grafo de estados do sistema de transferência de valores com duas tarefas discutido anteriormente. Nesse grafo, cada estado é a combinação dos estados individuais das duas tarefas. Pode-se observar no grafo que o estado E_{13} 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_{12} , E_{14} e E_{15} são considerados estados inseguros, pois levam invariavelmente na direção do impasse. Os demais estados são considerados seguros, pois a partir de cada um deles é possível continuar a execução e retornar ao estado inicial E_0 . Obviamente, transições que levem de um estado seguro a um inseguro devem ser evitadas, como $E_9 \rightarrow E_{14}$ ou $E_{10} \rightarrow E_{12}$.

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 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 poderem 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.

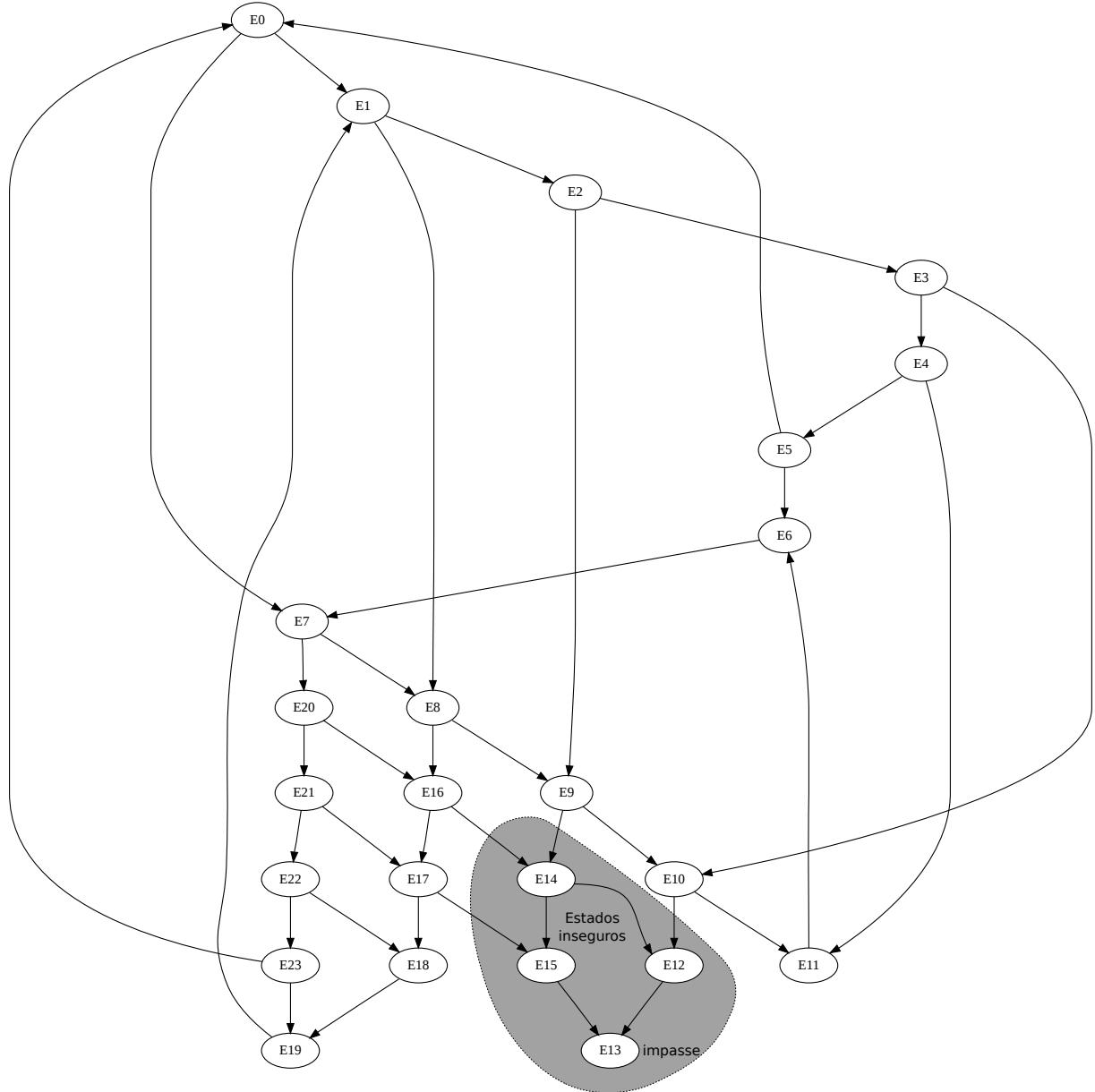


Figura 4.12: Grafo de estados do sistema de transferências com duas tarefas.

Detectão 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 o detecta, determina quais as tarefas e recursos envolvidos e toma 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 4.10.2), 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.

Uma vez detectado um impasse e identificadas as tarefas e recursos envolvidos, o sistema deve proceder à sua resolução, 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, etc.

Retroceder tarefas : uma ou mais tarefas envolvidas no impasse têm sua execução parcialmente desfeita, 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 necessário⁶. 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?

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.

⁶Essa técnica é conhecida como *checkpointing* e os estados anteriores salvos são denominados *checkpoints*.

Capítulo 5

Gerência de memória

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, bibliotecas compartilhadas e canais de comunicação, 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 caches, unidade de gerência, etc, o que exige um esforço de gerência significativo por parte do sistema operacional.

Uma gerência adequada da memória é essencial para o bom desempenho de um computador. Neste capítulo serão estudados os elementos de hardware que compõe a memória de um sistema computacional e os mecanismos implementados ou controlados pelo sistema operacional para a gerência da memória.

5.1 Estruturas 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 dados. 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 rígidos 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 dados.

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 e o custo por byte armazenado. Essas características permitem definir uma *hierarquia de memória*, representada na forma de uma pirâmide (Figura 5.1).

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 rígidos. Além disso, as memórias mais rápidas são *voláteis*, ou seja, perdem

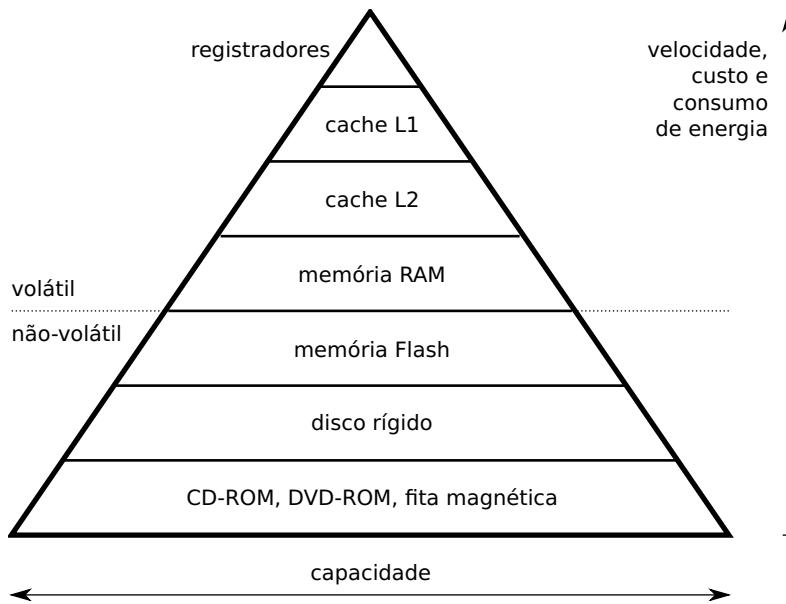


Figura 5.1: Hierarquia de memória.

seu conteúdo ao ficarem sem energia. Memórias que preservam seu conteúdo mesmo quando não tiverem energia são denominadas *não-voláteis*.

Outra característica importante das memórias é a rapidez de seu funcionamento, que pode ser detalhada em duas dimensões: *tempo de acesso* (ou *latência*) e *taxa de transferência*. O tempo de acesso caracteriza o tempo necessário para iniciar uma 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 5.1 traz valores de tempo de acesso e taxa de transferência 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 IDE	10 ms (tempo necessário para o deslocamento da cabeça de leitura e 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 5.1: Tempos de acesso e taxas de transferência típicas [Patterson and Henessy, 2005].

Neste capítulo serão estudados os 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* ou memória de leitura/escrita). Também será estudado o uso do disco rígido como extensão da memória principal, através de mecanismos de memória virtual (Seção 5.7). A gerência dos espaços de armazenamento em disco rígido é abordada no Capítulo 6. 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].

5.2 Endereços, variáveis e funções

Ao escrever um programa usando uma linguagem de alto nível, como C, C++ ou Java, o programador usa apenas referências a entidades abstratas, como variáveis, funções, parâmetros e valores de retorno. 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 de dados (`i` e `soma`) ou de trechos de código (`main`, `printf` e `exit`):

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main ()
5 {
6     int i, soma = 0 ;
7
8     for (i=0; i< 5; i++)
9     {
10         soma += i ;
11         printf ("i vale %d e soma vale %d\n", i, soma) ;
12     }
13     exit(0) ;
14 }
```

Todavia, o processador do computador acessa endereços de memória para buscar as instruções a executar e seus operandos; acessa também outros endereços de memória para escrever os resultados do processamento das instruções. Por isso, quando programa `soma.c` for compilado, ligado a bibliotecas, carregado na memória e executado pelo processador, cada variável ou trecho de código definido pelo programador deverá ocupar um espaço específico e exclusivo na memória, com seus próprios endereços. A listagem a seguir apresenta o código *assembly* correspondente à compilação do programa `soma.c`. Nele, pode-se observar que não há mais referências a nomes simbólicos, apenas a endereços:

Dessa forma, os endereços das variáveis e trechos de código usados por um programa devem ser definidos em algum momento entre a escrita do código e sua execução pelo processador, que pode ser:

Durante a edição : o programador escolhe a posição 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 linguagem de máquina.

Durante a compilação : o compilador escolhe as posições das variáveis na memória. Para isso, todos os códigos-fonte que fazem parte do programa devem ser conhecidos no momento da compilação, para evitar conflitos de endereços entre variáveis. Uma outra técnica bastante usada é a geração de código independente de posição (PIC - *Position-Independent Code*), no qual todas as referências a variáveis são feitas usando endereços relativos (como “3.471 bytes após o início do módulo”, ou “15 bytes após o *program counter*”, por exemplo).

Durante a ligação : o compilador gera símbolos que representam as variáveis mas não define seus endereços finais, gerando um arquivo que contém as instruções em linguagem de máquina e as definições das variáveis utilizadas, denominado

*arquivo objeto*¹. 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 de alto nível. O ligador (ou *link-editor*) então lê todos os arquivos-objeto e as bibliotecas e gera um arquivo-objeto executável, no qual os endereços de todas as variáveis estão corretamente definidos.

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 das 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 uso de hardware dedicado para esse tratamento. Esta é a abordagem usada na maioria dos sistemas computacionais atuais (como os computadores pessoais), e será descrita nas próximas seções.

A Figura 5.2 ilustra os diferentes momentos da vida de um processo em que pode ocorrer a resolução dos endereços de variáveis e de código.

5.2.1 Endereços lógicos e físicos

Ao executar uma sequência de instruções, o processador escreve endereços no barramento de endereços do computador, que servem para buscar instruções e operandos, mas também para ler e escrever valores em posições de memória e portas de entrada/saída. Os endereços de memória gerados pelo processador à medida em que executa algum código são chamados de *endereços lógicos*, porque correspondem à lógica do programa, mas não são necessariamente iguais aos endereços reais das instruções e variáveis na memória real do computador, que são chamados de *endereços físicos*.

Os endereços lógicos emitidos pelo processador são interceptados por um hardware especial denominado *Unidade de Gerência de Memória* (MMU - *Memory Management Unit*), que pode fazer parte do próprio processador (como ocorre nos sistemas atuais) ou constituir um dispositivo separado (como ocorria na máquinas mais antigas). A MMU faz a análise dos endereços lógicos emitidos pelo processador e determina os endereços físicos correspondentes na memória da máquina, permitindo então seu acesso pelo processador. Caso o acesso a um determinado endereço solicitado pelo processador não

¹Arquivos-objeto são formatos de arquivo projetados para conter código binário e dados provenientes de uma compilação de código-fonte. Existem diversos formatos de arquivos-objeto; os mais simples, como os arquivos .com do DOS, apenas definem uma sequência de bytes a carregar em uma posição fixa da memória; os mais complexos, como os formatos UNIX ELF (*Executable and Library Format*) e Microsoft PE (*Portable Executable Format*), permitem definir seções internas, tabelas de relocação, informação de depuração, etc. [Levine, 2000].

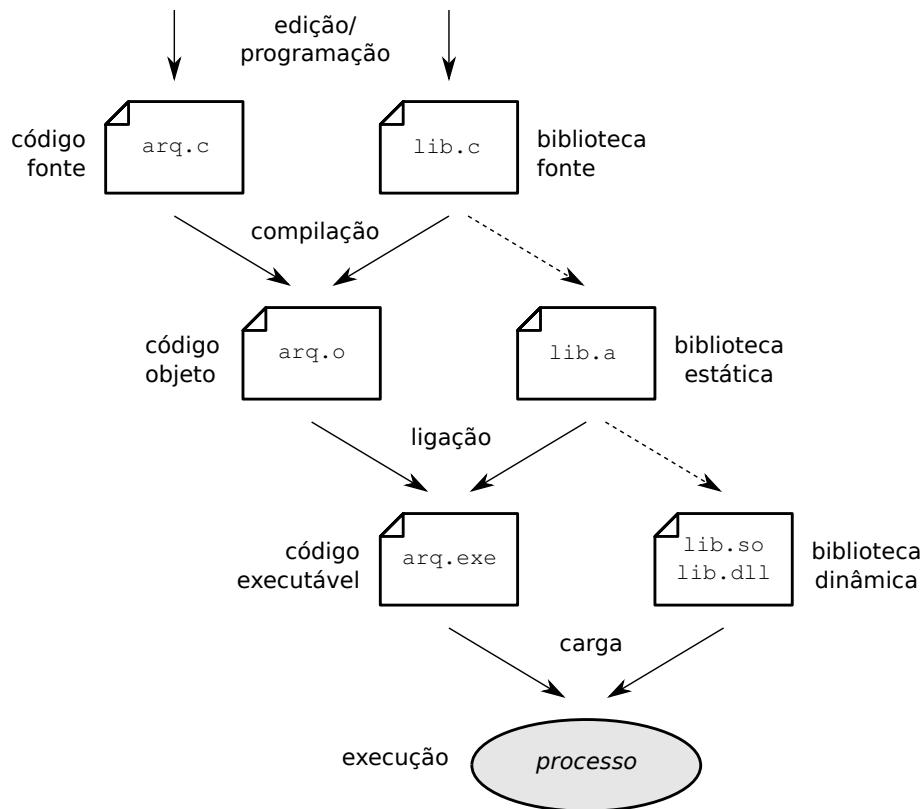


Figura 5.2: Momentos de atribuição de endereços.

seja possível, a MMU gera uma interrupção de hardware para notificar o processador sobre a tentativa de acesso indevido. O funcionamento básico da MMU está ilustrado na Figura 5.3.

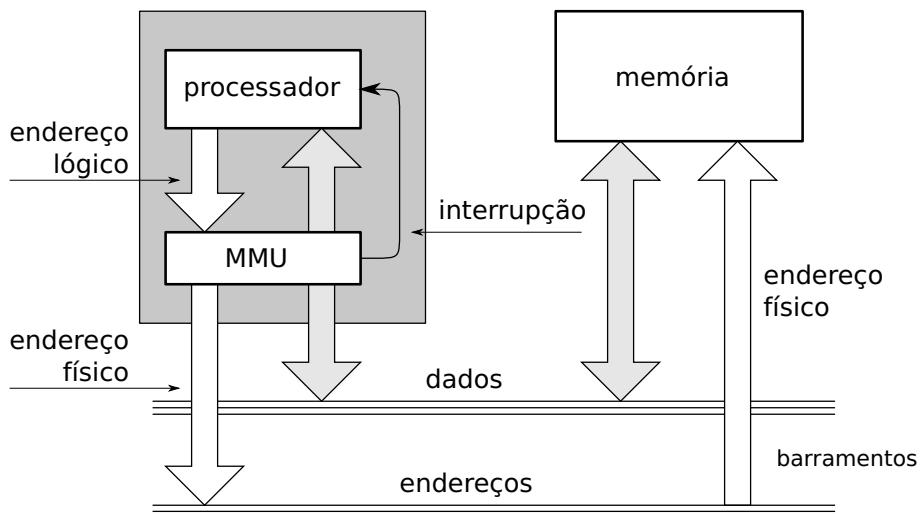


Figura 5.3: Funcionamento básico de uma MMU.

A proteção de memória entre processos é essencial para a segurança e estabilidade dos sistemas mais complexos, nos quais centenas ou milhares de processos podem estar

na memória simultaneamente. A MMU pode ser rapidamente ajustada para mudar a forma de conversão entre endereços lógicos e físicos, o que permite implementar uma área de memória exclusiva para cada processo do sistema. Assim, a cada troca de contexto entre processos, as regras de conversão da MMU devem ser ajustadas para somente permitir o acesso à área de memória definida para cada novo processo corrente.

5.2.2 Modelo de memória dos processos

Cada processo é visto pelo sistema operacional como uma cápsula isolada, ou seja, uma área de memória exclusiva que só ele e o núcleo do sistema podem acessar. Essa área de memória contém todas as informações necessárias à execução do processo, divididas nas seguintes seções:

TEXT : contém o código a ser executado pelo processo, gerado durante a compilação e a ligação com as bibliotecas. Esta área tem tamanho fixo, calculado durante a compilação, e normalmente só deve estar acessível para leitura e execução.

DATA : esta área contém os dados estáticos usados pelo programa, ou seja, suas variáveis globais e as variáveis locais estáticas (na linguagem C, são as variáveis definidas como `static` dentro das funções). Como o tamanho dessas variáveis pode ser determinado durante a compilação, esta área tem tamanho fixo; deve estar acessível para leituras e escritas, mas não para execução.

HEAP : área usada para armazenar dados através de alocação dinâmica, usando operadores como `malloc` e `free` ou similares. Esta área tem tamanho variável, podendo aumentar/diminuir conforme as alocações/liberações de memória feitas pelo processo. Ao longo do uso, esta área pode se tornar fragmentada, ou seja, pode conter lacunas entre os blocos de memória alocados. São necessários então algoritmos de alocação que minimizem sua fragmentação.

STACK : área 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 elevados e cresce em direção aos endereços menores da memória. No caso de programas com múltiplas *threads*, esta área contém somente a pilha do programa principal. Como *threads* podem ser criadas e destruídas dinamicamente, a pilha de cada *thread* é mantida em uma área própria, geralmente alocada no *heap*.

A Figura 5.4 apresenta a organização da memória de um processo. Nela, observa-se que as duas áreas de tamanho variável (*stack* e *heap*) estão dispostas em posições opostas e vizinhas à memória livre (não alocada). Dessa forma, a memória livre disponível ao processo pode ser aproveitada da melhor forma possível, tanto pelo *heap* quanto pelo *stack*, ou por ambos.

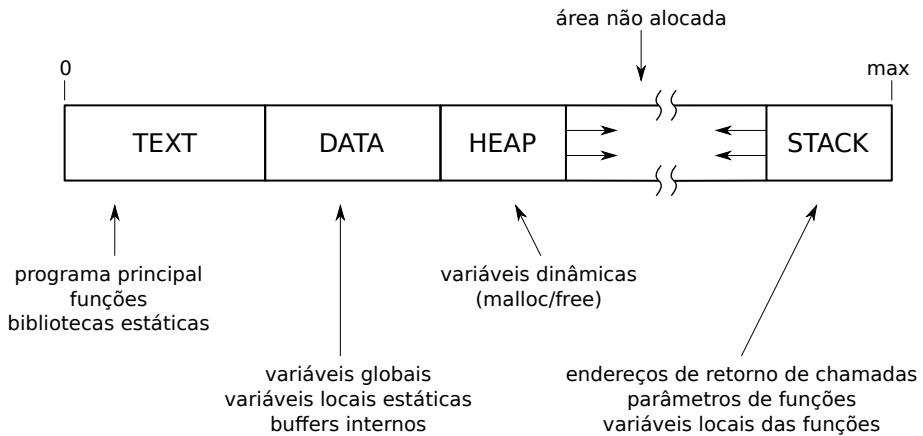


Figura 5.4: Organização da memória de um processo.

5.3 Estratégias de alocação

Em um sistema mono-processo, em que apenas um processo por vez é carregado em memória para execução, a alocação da memória principal é um problema simples de resolver: basta reservar uma área de memória para o núcleo do sistema operacional e alocar o processo na memória restante, respeitando a disposição de suas áreas internas, conforme apresentado na Figura 5.4.

A memória reservada para o núcleo do sistema operacional pode estar no início ou no final da área de memória física disponível. Como a maioria das arquiteturas de hardware define o vetor de interrupções (vide Seção 1.5.1) nos endereços iniciais da memória (também chamados *endereços baixos*), geralmente o núcleo também é colocado na parte inicial da memória. Assim, toda a memória disponível após o núcleo do sistema é destinada aos processos no nível do usuário (*user-level*). A Figura 5.5 ilustra essa organização da memória.

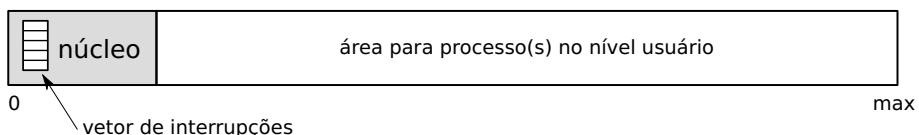


Figura 5.5: Organização da memória do sistema.

Nos sistemas multi-processos, vários processos podem ser carregados na memória para execução simultânea. Nesse caso, o espaço de memória destinado aos processos deve ser dividido entre eles usando uma estratégia que permita eficiência e flexibilidade de uso. As principais estratégias de alocação da memória física serão estudadas nas próximas seções.

5.3.1 Partições fixas

A forma mais simples de alocação de memória consiste em dividir a memória destinada aos processos em N partições fixas, de tamanhos iguais ou distintos. Em

cada partição pode ser carregado um processo. Nesse esquema, a tradução entre os endereços lógicos vistos pelos processos e os endereços físicos é feita através de um simples registrador de relocação, cujo valor é somado ao endereço lógico gerado pelo processador, a fim de obter o endereço físico correspondente. Endereços lógicos maiores que o tamanho da partição em uso são simplesmente rejeitados pela MMU. O exemplo da Figura 5.6 ilustra essa estratégia. No exemplo, o processo da partição 3 está executando e deseja acessar o endereço lógico 14.257. A MMU recebe esse endereço e o soma ao valor do registrador de relocação (110.000) para obter o endereço físico 124.257, que então é acessado. Deve-se observar que o valor contido no registrador de relocação é o endereço de início da partição ativa (partição 3); esse registrador deve ser atualizado a cada troca de processo ativo.

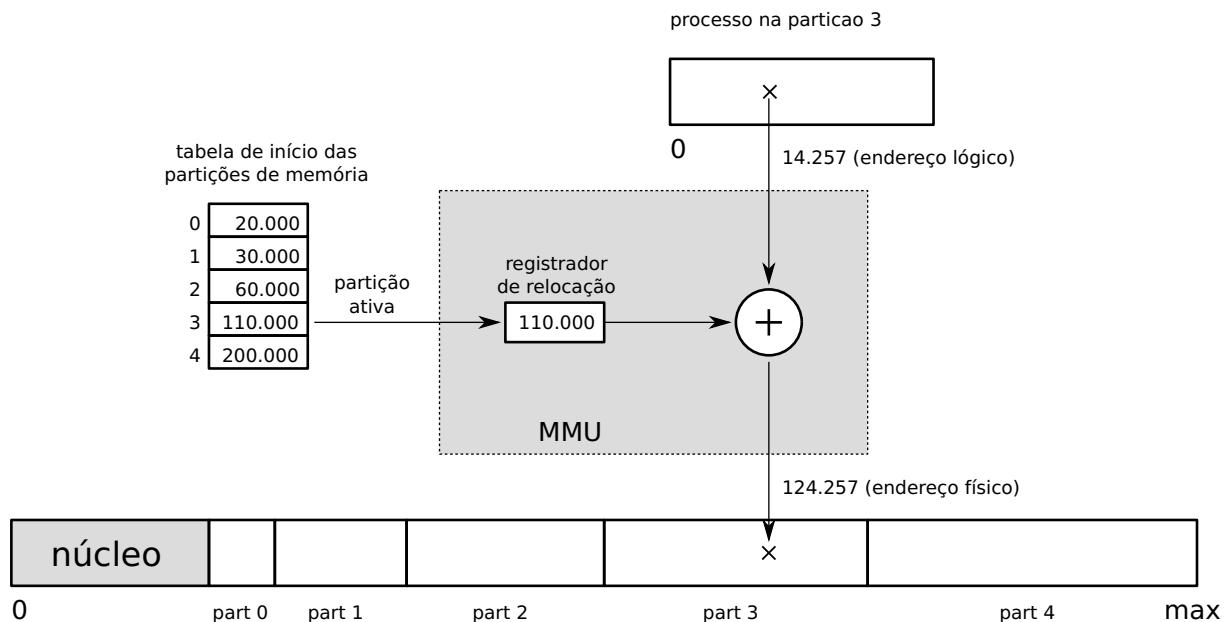


Figura 5.6: Alocação em partições fixas.

Essa abordagem é extremamente simples, todavia sua simplicidade não compensa suas várias desvantagens:

- Os processos podem ter tamanhos distintos dos tamanhos das partições, o que implica em áreas de memória sem uso no final de cada partição.
- O número máximo de processos na memória é limitado ao número de partições, mesmo que os processos sejam pequenos.
- Processos maiores que o tamanho da maior partição não poderão ser carregados na memória, mesmo se todas as partições estiverem livres.

Por essas razões, esta estratégia de alocação é pouco usada atualmente; ela foi muito usada no OS/360, um sistema operacional da IBM usado nas décadas de 1960-70 [Tanenbaum, 2003].

5.3.2 Alocação contígua

A estratégia anterior, com partições fixas, pode ser tornar bem mais flexível caso o tamanho de cada partição possa ser ajustado para se adequar à demanda específica de cada processo. Nesse caso, a MMU deve ser projetada para trabalhar com dois registradores próprios: um *registrador base*, que define o endereço inicial da partição ativa, e um *registrador limite*, que define o tamanho em bytes dessa partição. O algoritmo de tradução de endereços lógicos em físicos é bem simples: cada endereço lógico gerado pelo processo em execução é comparado ao valor do registrador limite; caso seja maior ou igual a este, uma interrupção é gerada pela MMU de volta para o processador, indicando um endereço inválido. Caso contrário, o endereço lógico é somado ao valor do registrador base, para a obtenção do endereço físico correspondente. A Figura 5.7 apresenta uma visão geral dessa estratégia. Na Figura, o processo p_3 tenta acessar o endereço lógico 14.257.

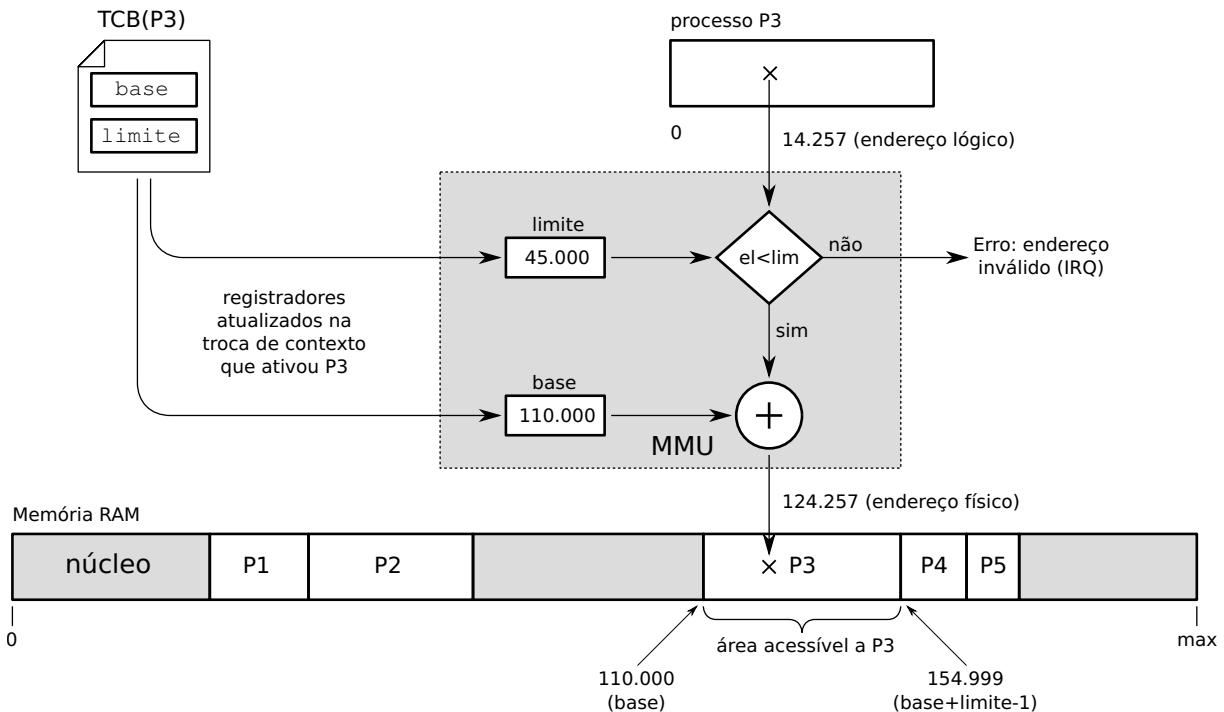


Figura 5.7: Alocação contígua de memória.

Os valores dos registradores base e limite da MMU devem ser ajustados pelo despachante (*dispatcher*) a cada troca de contexto, ou seja, cada vez que o processo ativo é substituído. Os valores de base e limite para cada processo do sistema devem estar armazenados no respectivo TCB (*Task Control Block*, vide Seção 2.4.1). Obviamente, quando o núcleo estiver executando, os valores de base e limite devem ser ajustados respectivamente para 0 e ∞ , para permitir 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 $[0, \text{limite}(p_i) - 1]$, que correspondem a endereços físicos no intervalo $[\text{base}(p_i), \text{base}(p_i) + \text{limite}(p_i) - 1]$.

Ao detectar uma tentativa de acesso a um endereço fora desse intervalo, a MMU irá gerar uma solicitação de interrupção (IRQ - *Interrupt ReQuest*, vide Seção 1.5.1) para o processador, indicando o endereço inválido. Ao receber a interrupção, o processador interrompe o fluxo de 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 alocação contígua é sua simplicidade: por depender apenas de dois registradores e de uma lógica simples para a tradução de endereços, pode ser implementada em hardware de baixo custo, ou mesmo incorporada a processadores mais simples. Todavia, é uma estratégia pouco flexível e está muito sujeita à fragmentação externa, conforme será discutido na Seção 5.5.

5.3.3 Alocação por segmentos

A alocação por segmentos, ou *alociação segmentada*, é uma extensão da alocação contígua, na qual o espaço de memória de um processo é fracionado em áreas, ou *segmentos*, que podem ser alocados separadamente na memória física. Além das quatro áreas funcionais básicas da memória do processo discutidas na Seção 5.2.2 (*text*, *data*, *stack* e *heap*), também podem ser definidos segmentos para ítems específicos, como bibliotecas compartilhadas, vetores, matrizes, pilhas de *threads*, buffers de entrada/saída, etc.

Ao estruturar a memória em segmentos, o espaço de memória de cada processo não é mais visto como uma sequência linear de endereços lógicos, mas como uma coleção de segmentos de tamanhos diversos e políticas de acesso distintas. A Figura 5.8 apresenta a visão lógica da memória de um processo e a sua forma de mapeamento para a memória física.

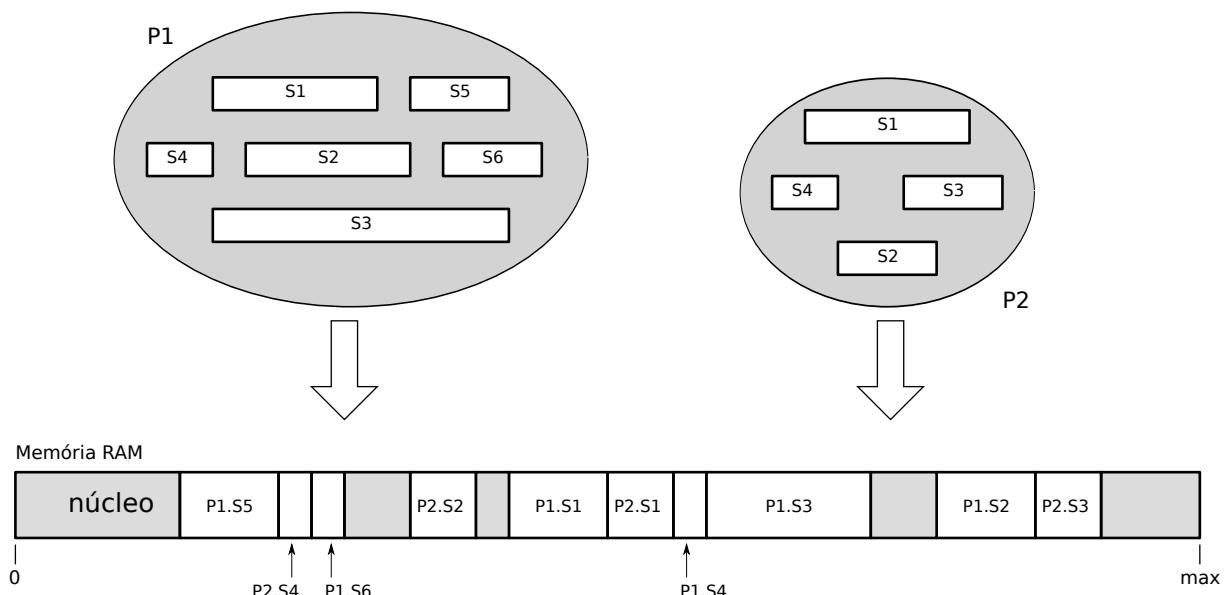


Figura 5.8: Alocação de memória por segmentos.

No modelo de memória alocada por segmentos, os endereços gerados pelos processos devem indicar as posições de memória e os segmentos onde elas se encontram. Em

outras palavras, este modelo usa endereços lógicos *bidimensionais*, compostos por pares *[segmento:offset]*, onde *segmento* indica o número do segmento desejado e *offset* indica a posição desejada dentro do segmento. Os valores de *offset* variam de 0 (zero) ao tamanho do segmento. A Figura 5.9 mostra alguns exemplos de endereços lógicos usando alocação por segmentos.

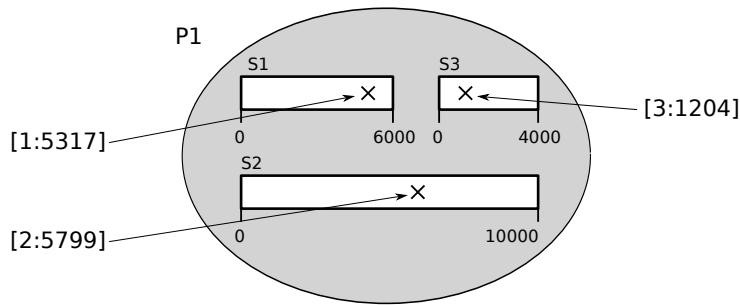


Figura 5.9: Endereços lógicos em segmentos.

Na alocação de memória por segmentos, a forma de tradução de endereços lógicos em físicos é similar à da alocação contígua. Contudo, como os segmentos podem ter tamanhos distintos e ser alocados separadamente na memória física, 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 segmento usado pelo processo, além de *flags* com informações sobre cada segmento, como permissões de acesso, etc. (vide Seção 5.3.4). A Figura 5.10 apresenta os principais elementos envolvidos na tradução de endereços lógicos em físicos usando memória alocada por segmentos. Nessa figura, “ST reg” indica o registrador que aponta para a tabela de segmentos ativa.

Cabe ao compilador colocar os diversos trechos do código-fonte de cada programa em segmentos separados. Ele pode, por exemplo, colocar cada vetor ou matriz em um segmento próprio. Dessa forma, erros frequentes como acessos a índices além do tamanho de um vetor irão gerar endereços fora do respectivo segmento, que serão detectados pelo hardware de gerência de memória e notificados ao sistema operacional.

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 alocar as tabelas na memória RAM. O processador 80.386 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 5.6). Cada uma dessas duas tabelas comporta até 8.192 segmentos. As tabelas em uso pelo processo em execução são indicadas por registradores específicos do processador. A cada troca de contexto, os registradores que indicam a tabela de segmentos ativa devem ser 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

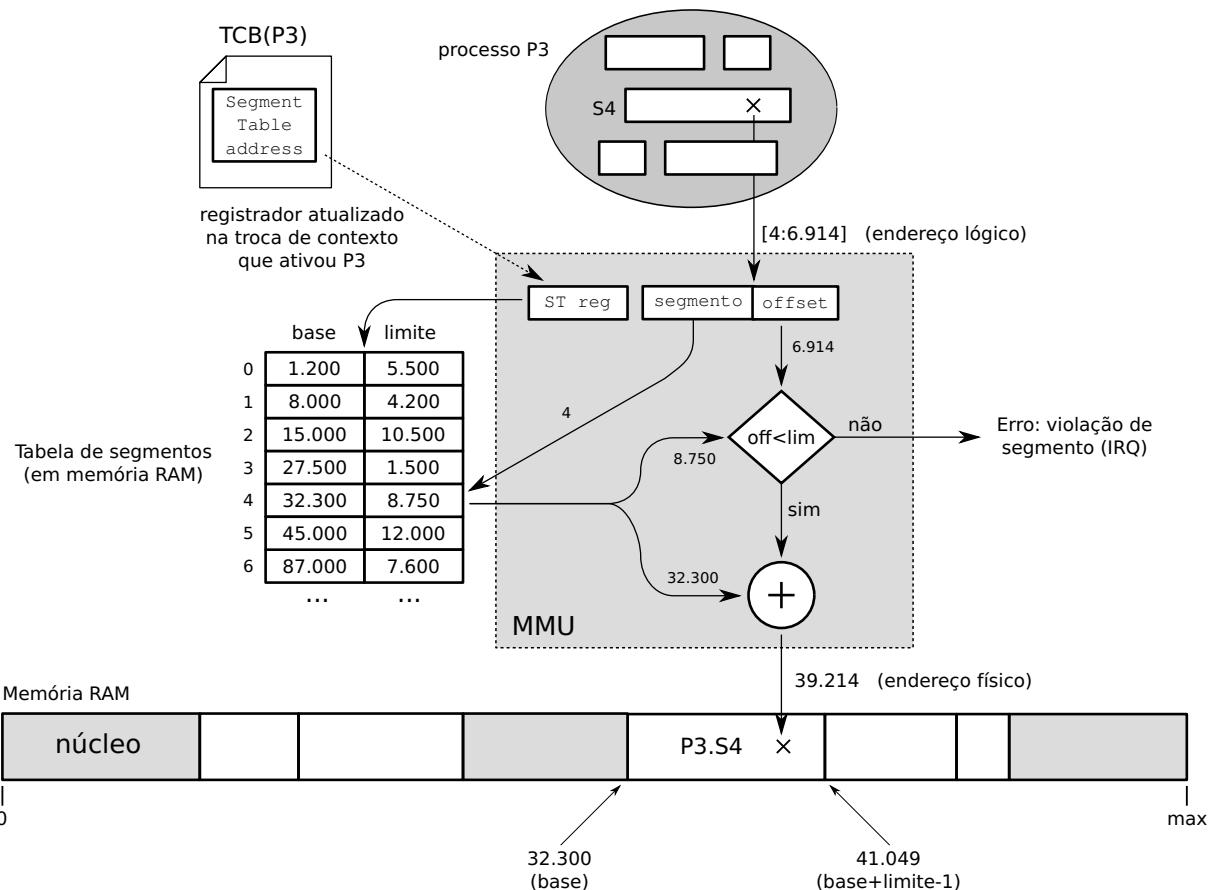


Figura 5.10: Tradução de endereços em memória alocada por segmentos.

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 com excessiva frequência, o que mantém o desempenho de acesso à memória em um nível satisfatório. O processador 80.386 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 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.

Exemplos de processadores que utilizam a alocação por segmentos incluem o 80.386 e seus sucessores (486, Pentium, Athlon e processadores correlatos).

5.3.4 Alocação paginada

Conforme visto na Seção anterior, a alocação de 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 alocação bastante suscetível à fragmentação externa, conforme será discutido na Seção 5.5. Essas deficiências levaram os projetistas de hardware a desenvolver outras técnicas para a alocação da memória principal.

Na alocação de memória por páginas, ou *alociação paginada*, o espaço de endereçamento lógico dos processos é mantido linear e unidimensional (ao contrário da alocação por segmentos, que usa endereços bidimensionais). Internamente, e de forma transparente para os processos, o espaço de endereços lógicos 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 tamanhos². O espaço de memória física destinado aos processos também é dividido em blocos de mesmo tamanho que as páginas, denominados *quadros* (do inglês *frames*). A alocação dos processos na memória física é então feita simplesmente indicando em que quadro da memória física se encontra cada página de cada processo, conforme ilustra a Figura 5.11. É importante observar que as páginas de um processo podem estar em qualquer posição da memória física disponível aos processos, ou seja, podem estar associadas a quaisquer quadros, o que permite uma grande flexibilidade de alocação. Além disso, as páginas não usadas pelo processo não precisam estar mapeadas na memória física, o que proporciona maior eficiência no uso da mesma.

O mapeamento entre as páginas de um processo e os quadros correspondentes na memória física é feita através de uma *tabela de páginas* (*page table*), na qual cada entrada corresponde a uma página e contém o número do quadro onde ela se encontra. Cada processo possui sua própria tabela de páginas; a tabela de páginas ativa, que

²As arquiteturas de processador mais recentes suportam diversos tamanhos de páginas, inclusive páginas muito grandes, as chamadas *super-páginas* (*hugepages*, *superpages* ou *largepages*). Uma super-página 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 bem mais complexos. O artigo [Navarro et al., 2002] traz uma discussão mais detalhada sobre esse tema.

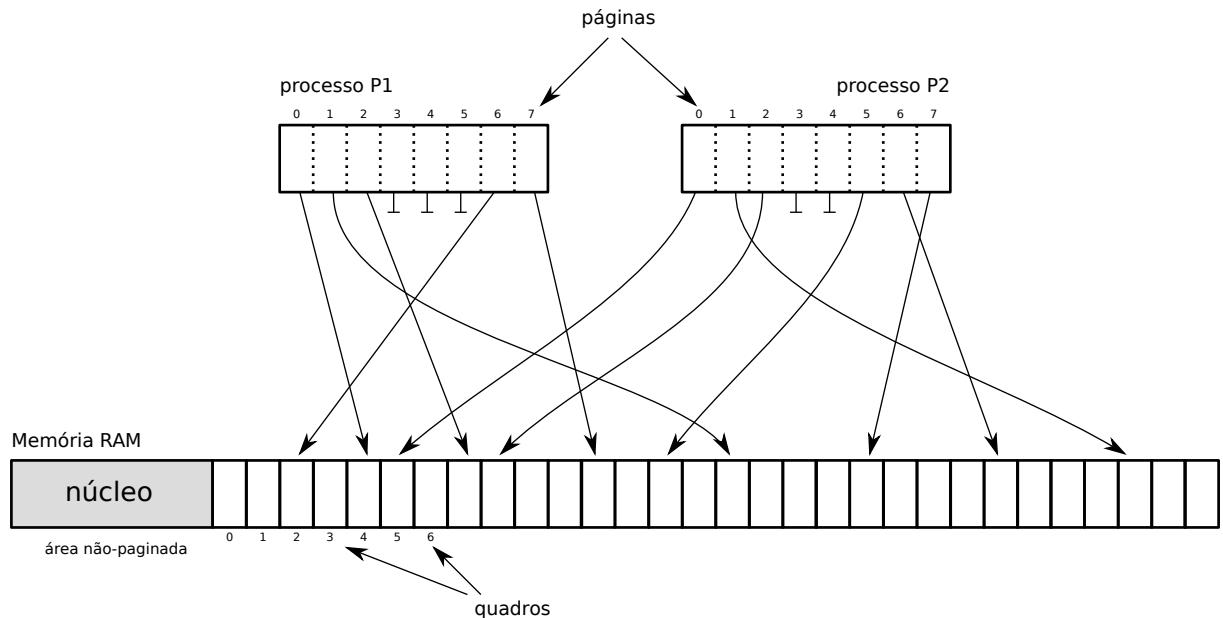


Figura 5.11: Alocação de memória por páginas.

corresponde ao processo em execução no momento, é referenciada por um registrador do processador 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 80.386 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 de um endereço lógico nesse sistema:

$$\begin{aligned}
 01805E9A_H &\rightarrow 0000\ 0001\ 1000\ 0000\ 0101\ 1110\ 1001\ 1010_2 \\
 &\rightarrow \underbrace{0000\ 0001\ 1000\ 0000}_\text{20 bits} \underbrace{0101_2}_\text{e} \underbrace{1110\ 1001\ 1010_2}_\text{12 bits} \\
 &\rightarrow \underbrace{01805_H}_\text{página} \text{ e } \underbrace{E9A_H}_\text{offset} \\
 &\rightarrow \text{página } 01805_H \text{ e offset } E9A_H
 \end{aligned}$$

Para traduzir um endereço lógico no endereço físico correspondente, a MMU precisa efetuar os seguintes passos:

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;
4. caso a página solicitada não esteja mapeada em um quadro da memória física, a MMU deve gerar uma interrupção de *falta de página* (*page fault*) para o processador;
5. essa interrupção provoca o desvio da execução para o núcleo do sistema operacional, que deve então tratar a falta de página.

A Figura 5.12 apresenta os principais elementos que proporcionam a tradução de endereços em um sistema paginado com páginas de 4.096 bytes.

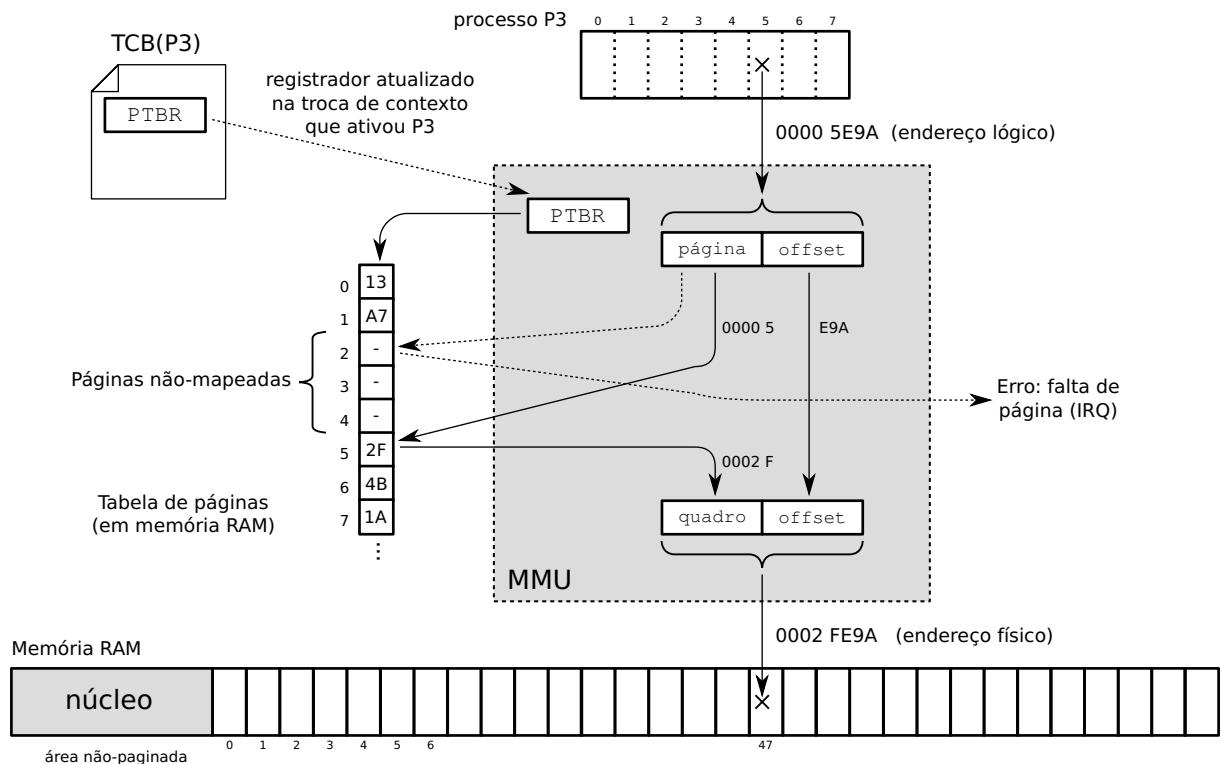


Figura 5.12: Tradução de endereços usando paginação.

Flags de controle

Como o espaço de endereçamento lógico de um processo pode ser extremamente grande (por exemplo, o espaço de endereços lógicos de cada processo em uma arquitetura

de 32 bits pode ir até 2^{32} bytes), uma parte significativa das páginas de um processo pode não estar mapeada em quadros de memória física. Áreas de memória não usadas por um processo não precisam estar mapeadas na memória física, o que permite um uso mais eficiente da memória. Assim, a tabela de páginas de cada processo indica as áreas não mapeadas com um flag adequado (*válido/inválido*). Cada entrada da tabela de páginas de um processo contém o número do quadro correspondente e um conjunto de flags (bits) de controle, com diversas finalidades:

- *Presença*: indica se a página está presente (mapeada) no espaço de endereçamento daquele processo;
- *Proteção*: bits indicando os direitos de acesso do processo à página (basicamente leitura, escrita e/ou execução);
- *Referência*: indica se a página foi referenciada (acessada) recentemente, sendo ajustado para 1 pelo próprio hardware a cada acesso à página. Este bit é usado pelos algoritmos de memória virtual (apresentados na Seção 5.7);
- *Modificação*: também chamado de *dirty bit*, é ajustado para 1 pelo hardware a cada escrita na página, indicando se a página foi modificada após ser carregada na memória; é usado pelos algoritmos de memória virtual.

Além destes, podem ser definidos outros bits, indicando a política de *caching* da página, se é uma página de usuário ou de sistema, se a página pode ser movida para disco, o tamanho da página (no caso de sistemas que permitam mais de um tamanho de página), etc. O conteúdo exato de cada entrada da tabela de páginas depende da arquitetura do hardware considerado.

Tabelas multi-níveis

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 flags). Considerando que cada tabela de páginas tem 2^{20} páginas, cada 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 ocupará mais espaço na memória que o próprio processo, como mostra a Figura 5.13, o que torna seu uso pouco interessante.

Para resolver esse problema, são usadas *tabelas de páginas multi-níveis*, estruturadas na forma de árvores: uma *tabela de páginas de primeiro nível* (ou *diretório de páginas*) contém ponteiros para *tabelas de páginas de segundo nível*, e assim por diante, até chegar à tabela que contém os números dos quadros desejados. Para percorrer essa árvore, o número de página é dividido em duas ou mais partes, que são usadas de forma sequencial, um para cada nível de tabela, até encontrar o número de quadro desejado. O número de níveis da tabela depende da arquitetura considerada: os processadores *Intel 80.386* usam tabelas com dois níveis, os processadores *Sun Sparc* e *DEC Alpha* usam tabelas com 3 níveis; processadores mais recentes, como *Intel Itanium*, podem usar tabelas com 3 ou 4 níveis.

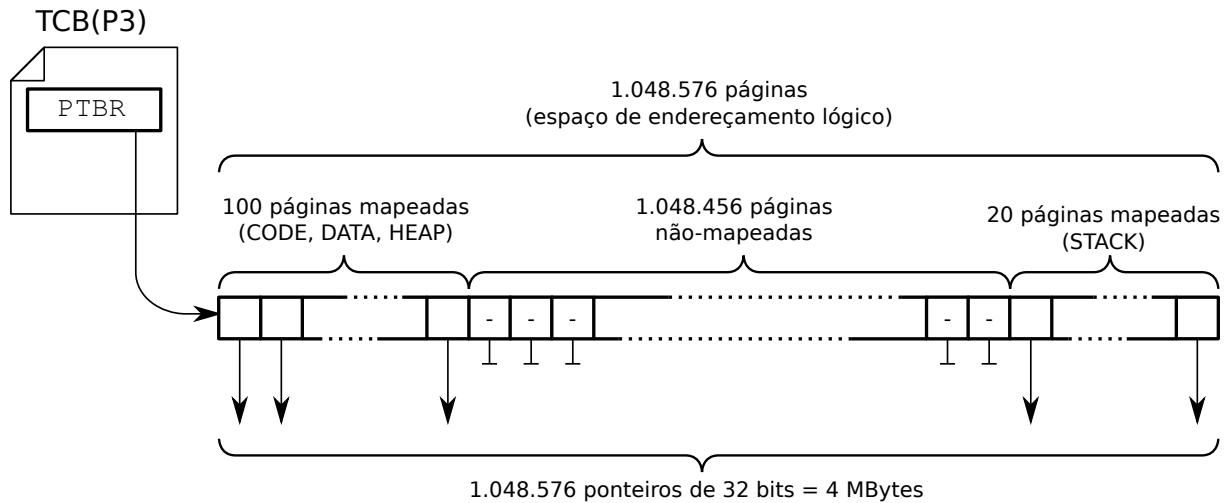


Figura 5.13: Inviabilidade de tabelas de página lineares.

Um exemplo permite explicar melhor esse conceito: considerando uma arquitetura de 32 bits com páginas de 4 Kbytes, 20 bits são usados para acessar a tabela de páginas. Esses 20 bits podem ser divididos em dois grupos de 10 bits que são usados como índices em uma tabela de páginas com dois níveis:

$$\begin{aligned}
 01805E9A_H &\rightarrow 0000\ 0001\ 1000\ 0000\ 0101\ 1110\ 1001\ 1010_2 \\
 &\rightarrow \underbrace{0000\ 0001\ 10}_2 \text{ e } \underbrace{00\ 0000\ 0101}_2 \text{ e } \underbrace{1110\ 1001\ 1010}_2 \\
 &\quad \text{10 bits} \qquad \text{10 bits} \qquad \text{12 bits} \\
 &\rightarrow 0006_H \text{ e } 0005_H \text{ e } E9A_H \\
 \\
 &\rightarrow p_1\ 0006_H, p_2\ 0005_H \text{ e offset } E9A_H
 \end{aligned}$$

A tradução de endereços lógicos em físicos usando uma tabela de páginas estruturada em dois níveis é efetuada através dos seguintes passos, que são ilustrados na Figura 5.14:

1. o endereço lógico el ($0180\ 5E9A_H$) é decomposto em um *offset* de 12 bits o ($E9A_H$) e dois números de página de 10 bits cada: o número de página de primeiro nível p_1 (006_H) e o número de página de segundo nível p_2 (005_H);
2. o número de página p_1 é usado como índice na tabela de página de primeiro nível, para encontrar o endereço de uma tabela de página de segundo nível;
3. o número de página p_2 é usado como índice na tabela de página de segundo nível, para encontrar o número de quadro q ($2F_H$) que corresponde a $[p_1p_2]$;
4. o número de quadro q é combinado ao *offset* o para obter o endereço físico ef ($0002\ FE9A_H$) correspondente ao endereço lógico solicitado el .

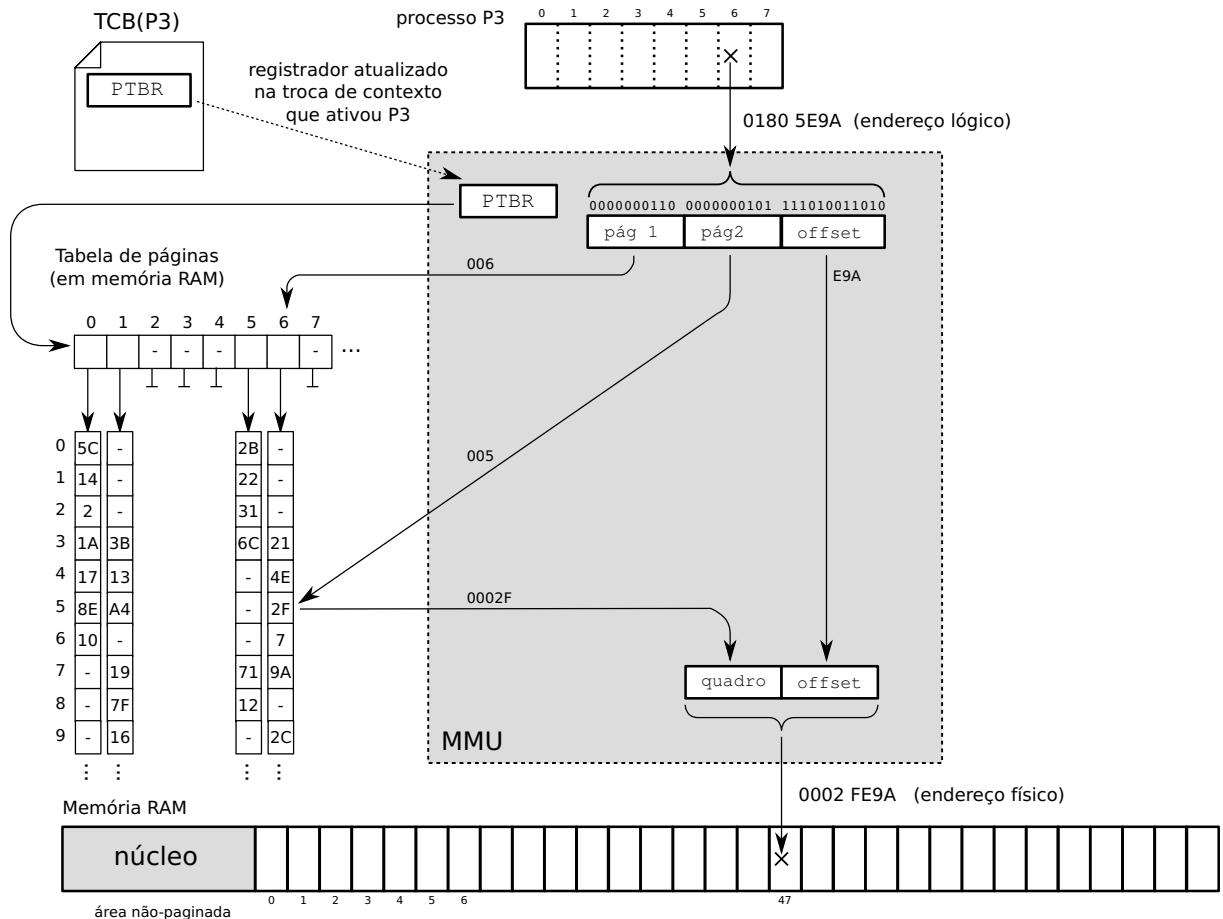


Figura 5.14: Tabela de página multi-nível.

Com a estruturação da tabela de páginas em níveis, a quantidade de memória necessária para armazená-la diminui significativamente, sobretudo no caso de processos pequenos. Considerando o processo apresentado como exemplo na Figura 5.13, ele faria uso de uma tabela de primeiro nível e somente duas tabelas de segundo nível (uma para mapear suas primeiras páginas e outra para mapear suas últimas páginas); todas as demais entradas da tabela de primeiro nível estariam vazias. Assumindo que cada entrada de tabela ocupa 4 bytes, serão necessários somente 12 Kbytes para armazenar essas três tabelas ($4 \times 3 \times 2^{10}$ bytes). Na situação limite onde um processo ocupa toda a memória possível, seriam necessárias uma tabela de primeiro nível e 1.024 tabelas de segundo nível. Essas tabelas ocupariam de $4 \times (2^{10} \times 2^{10} + 2^{10})$ bytes, ou seja, 0,098% a mais que se a tabela de páginas fosse estruturada em um só nível (4×2^{20} bytes). Essas duas situações extremas de alocação estão ilustradas na Figura 5.15.

Cache da tabela de páginas

A estruturação das tabelas de páginas em vários níveis resolve o problema do espaço ocupado pelas tabelas de forma muito eficiente, mas tem um efeito colateral muito nocivo: aumenta drasticamente o tempo de acesso à memória. Como as tabelas de páginas são armazenadas na memória, cada acesso a um endereço de memória implica

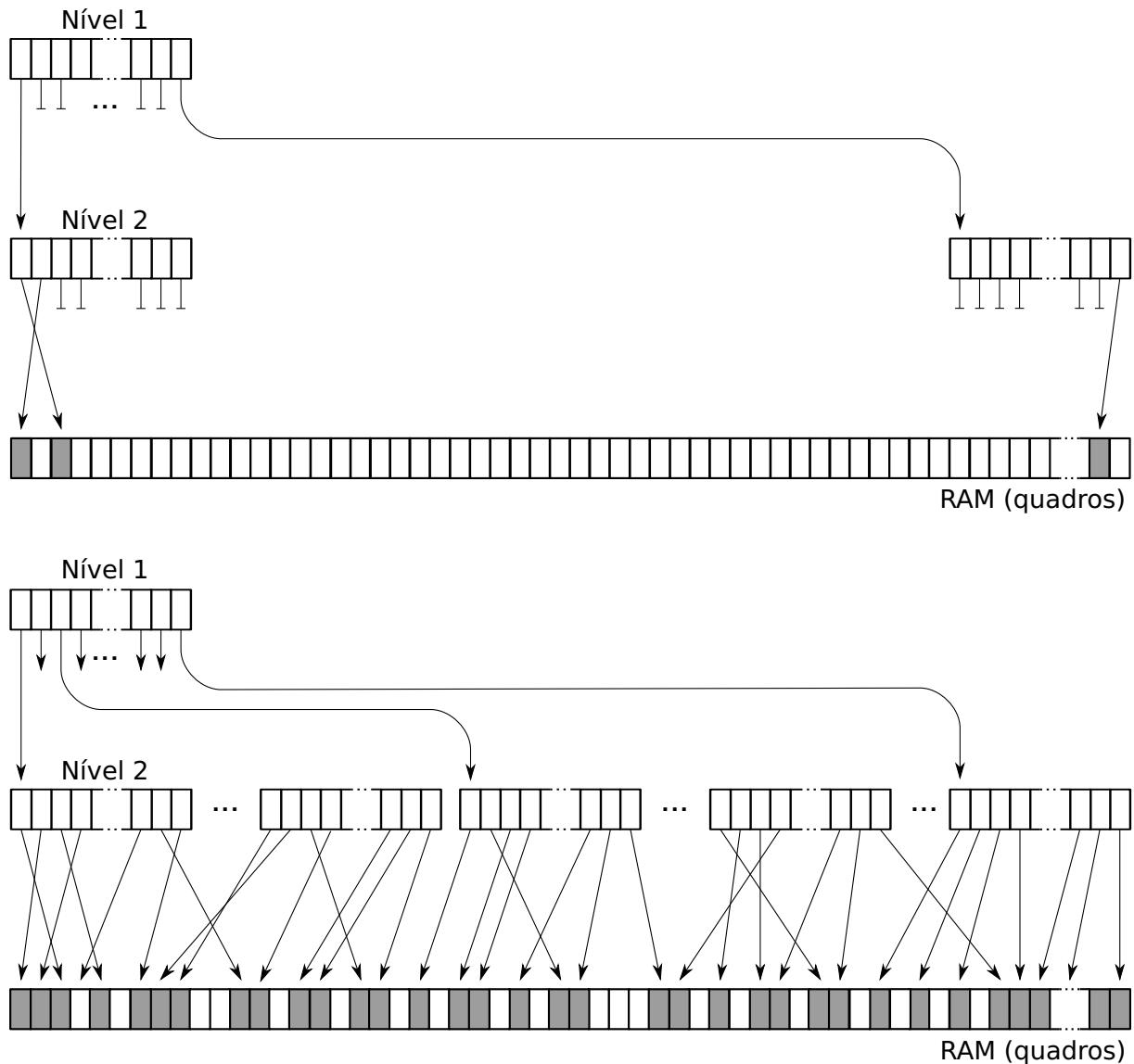


Figura 5.15: Tabela de páginas multi-nível vazia (alto) e cheia (baixo).

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.

Quando um processo executa, ele acessa endereços de memória para buscar instruções e operandos e ler/escrever dados. Em cada instante, os acessos tendem a se concentrar em poucas páginas, que contém o código e as variáveis usadas naquele instante³. Dessa forma, a MMU terá de fazer muitas traduções consecutivas de endereços nas mesmas páginas, que irão resultar nos mesmos quadros de memória física. Por isso, consultas recentes à tabela de páginas são armazenadas em um *cache* dentro da própria MMU,

³Esse fenômeno é conhecido como *localidade de referências* e será detalhado na Seção 5.4.

evitando ter de repeti-las constantemente e assim diminuindo o tempo de acesso à memória física.

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 flag de erro chamado *erro de cache* (*cache miss*). Por ser implementado em hardware especial rápido e caro, geralmente esse cache é pequeno: TLBs de processadores típicos têm entre 16 e 256 entradas. Seu tempo de acesso é pequeno: um acerto custa cerca de 1 ciclo de relógio da CPU, enquanto um erro pode custar entre 10 e 30 ciclos.

A tradução de endereços lógicos em físicos usando TLBs se torna mais rápida, mas também mais complexa. Ao receber um endereço lógico, a MMU consulta o TLB; caso o número do quadro correspondente esteja em cache, ele é usado para compor o endereço físico e o acesso à memória é efetuado. Caso contrário, uma busca normal (completa) na tabela de páginas deve ser realizada. O quadro obtido nessa busca é usado para compor o endereço físico e também é adicionado ao TLB para agilizar as consultas futuras. A Figura 5.16 apresenta os detalhes desse procedimento.

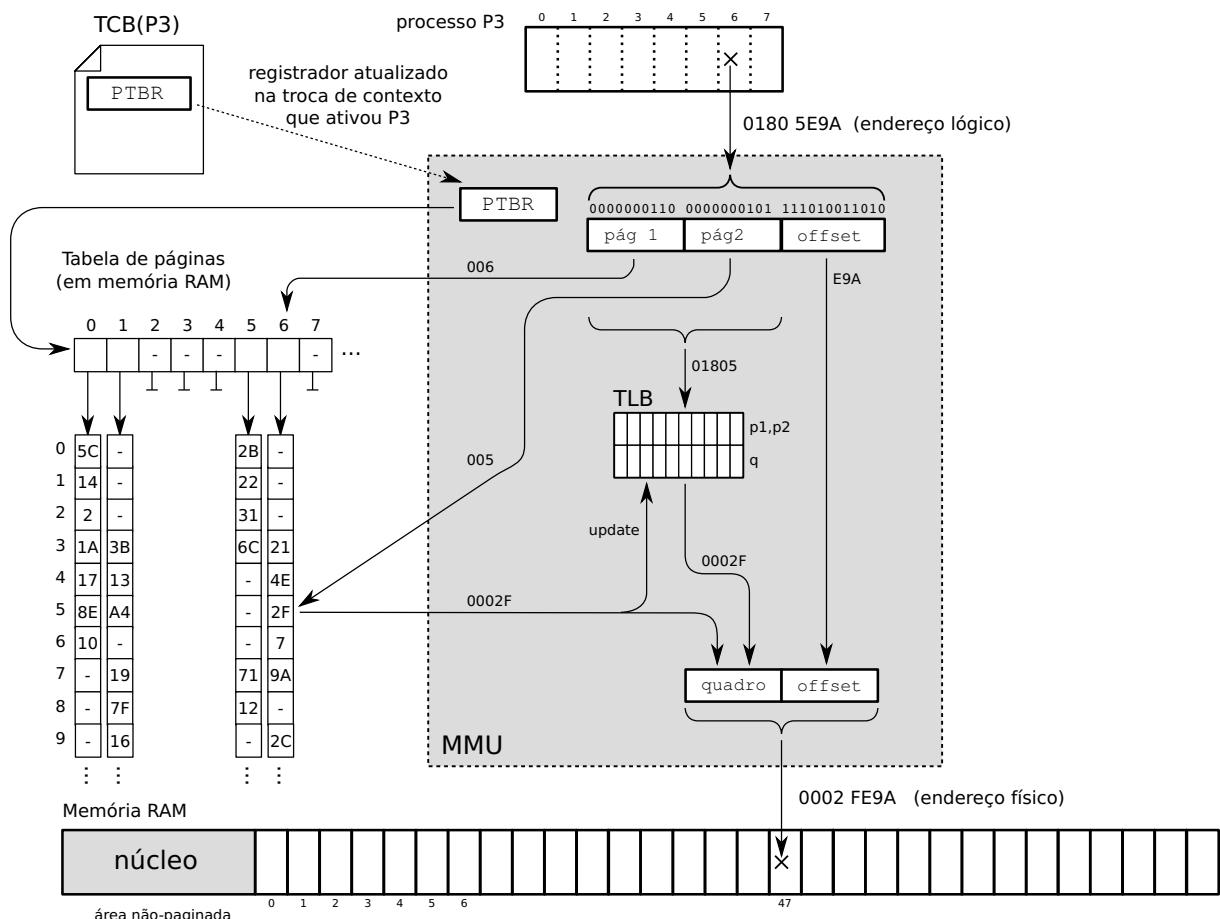


Figura 5.16: Uso da TLB.

É fácil perceber que, quanto maior a taxa de acertos do TLB (*cache hit ratio*), melhor é o desempenho dos acessos à memória física. O tempo médio de acesso à memória pode então ser determinado pela média ponderada entre o tempo de acesso com acerto de cache e o tempo de acesso no caso de erro. Por exemplo, considerando um sistema operando a 2 GHz (relógio de 0,5 ns) com tempo de acesso à 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 taxa de acerto de 95%, o tempo médio de acesso à memória pode ser estimado como segue:

$$\begin{aligned} t_{\text{médio}} &= 95\% \times 0,5\text{ns} && // \text{em caso de acerto} \\ &+ 5\% \times (10\text{ns} + 3 \times 50\text{ns}) && // \text{em caso de erro, consultar as tabelas} \\ &+ 50\text{ns} && // \text{acesso ao quadro desejado} \end{aligned}$$

$$t_{\text{médio}} = 58,475\text{ns}$$

Este resultado indica que o sistema de paginação multi-nível 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 fosse de 90%, o custo adicional seria de 32,9%; caso a taxa subisse a 99%, o custo adicional cairia para 4,2%.

Obviamente, 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, o *Intel i386* tem um TLB com 64 entradas para páginas de dados e 32 entradas para páginas de código; por sua vez, o *Intel Itanium* tem 128 entradas para páginas de dados e 96 entradas para páginas de código.

O tamanho do TLB é um fator que influencia a sua taxa de acertos, mas há outros fatores importantes a considerar, como a política de substituição das entradas do TLB. Essa política define o que ocorre quando há um erro de cache e não há entradas livres no TLB: em alguns processadores, a associação [*página, quadro*] que gerou o erro é adicionada ao cache, substituindo a entrada mais antiga; todavia, na maioria dos processadores mais recentes, cada erro de cache provoca uma interrupção, que transfere ao sistema operacional a tarefa de gerenciar o conteúdo do TLB [Patterson and Henessy, 2005].

Outro aspecto que influencia significativamente a taxa de acerto do TLB é a forma como cada processo acessa a memória. Processos que concentram seus acessos em poucas páginas de cada vez farão um uso eficiente desse cache, enquanto processos que acessam muitas páginas distintas em um curto período irão gerar frequentes erros de cache, prejudicando seu desempenho no acesso à memória. Essa propriedade é conhecida como *localidade de referência* (Seção 5.4).

Finalmente, é importante observar que o conteúdo do TLB reflete a tabela de páginas ativa, que indica as páginas de memória pertencentes ao processo em execução naquele momento. A cada troca de contexto, a tabela de páginas é substituída e portanto o cache TLB deve ser esvaziado, pois seu conteúdo não é mais válido. Isso permite concluir que trocas de contexto muito frequentes prejudicam a eficiência de acesso à memória, tornando o sistema mais lento.

5.3.5 Alocação segmentada paginada

Cada uma das principais formas de alocação de memória vistas até agora tem suas vantagens: a alocação contígua prima pela simplicidade e rapidez; a alocação por segmentos oferece múltiplos espaços de endereçamento para cada processo, oferecendo flexibilidade ao programador; a alocação por páginas oferece um grande espaço de endereçamento linear, enquanto elimina a fragmentação externa. Alguns processadores oferecem mais de uma forma de alocação, deixando aos projetistas do sistema operacional a escolha da forma mais adequada de organizar a memória usada por seus processos.

Vários processadores permitem combinar mais de uma forma de alocação. Por exemplo, os processadores *Intel i386* permitem combinar a alocação com segmentos com a alocação por páginas, visando oferecer a flexibilidade da alocação por segmentos com a baixa fragmentação da alocação por páginas.

Nessa abordagem, os processos vêm a memória estruturada em segmentos, conforme indicado na Figura 5.9. O hardware da MMU converte os endereços lógicos na forma `[segmento:offset]` para endereços lógicos lineares (unidimensionais), usando as tabelas de descritores de segmentos (Seção 5.3.3). Em seguida, esse endereços lógicos lineares são convertidos nos endereços físicos correspondentes através do hardware de paginação (tabelas de páginas e TLB), visando obter o endereço físico correspondente.

Apesar do processador *Intel i386* oferecer as duas formas de alocação 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 alocação por páginas. O antigo DOS e o Windows 3.* usavam somente a alocação por segmentos. O OS/2 da IBM foi um dos poucos sistemas operacionais comerciais a fazer uso pleno das possibilidades de alocação de memória nessa arquitetura, combinando segmentos e páginas.

5.4 Localidade de referências

A forma como os processos acessam a memória tem um impacto direto na eficiência dos mecanismos de gerência de memória, sobretudo o cache de páginas (TLB, Seção 5.3.4) e o mecanismo de memória virtual (Seção 5.7). 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 (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 (esta propriedade é usada pelos algoritmos de gerência de memória virtual);

Localidade espacial : um recurso será mais provavelmente acessado se outro recurso próximo a ele já foi acessado (é a propriedade verificada na primeira execução);

Localidade sequencial : é um caso particular da localidade espacial, no qual há uma predominância de acesso sequencial aos recursos (esta propriedade é útil na otimização de sistemas de arquivos).

Como exemplo prático da importância da localidade de referências, considere 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 ;
10 }
```

Outra implementação possível seria percorrer a matriz coluna por coluna, conforme o código a seguir:

```

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 ;
10 }
```

Embora percorram a matriz de forma distinta, os dois programas geram o mesmo resultado e são conceitualmente equivalentes (a Figura 5.17 mostra o padrão de acesso à memória dos dois programas). 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 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 10 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.

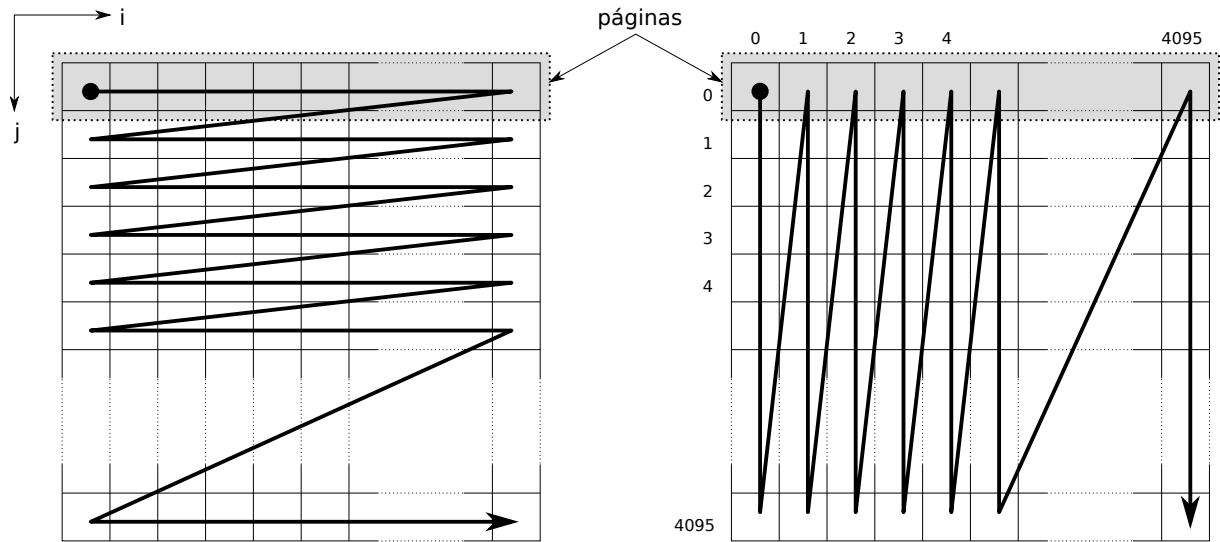


Figura 5.17: Comportamento dos programas no acesso à memória.

A diferença de comportamento das duas execuções pode ser observada na Figura 5.18, que mostra a distribuição dos endereços de memória acessados pelos dois códigos⁴. Nos gráficos, percebe-se claramente que a primeira implementação tem uma localidade de referências muito mais forte 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.

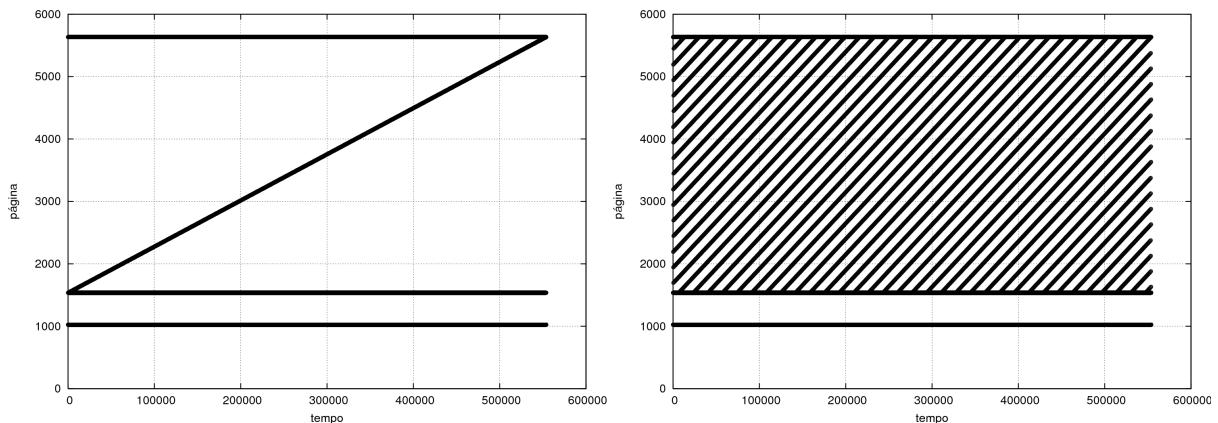


Figura 5.18: Localidade de referências nas duas execuções.

A Figura 5.19 traz outro exemplo de boa localidade de referências. Ela mostra as páginas acessadas durante uma execução do visualizador gráfico *gThumb*, ao abrir um arquivo de imagem. 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 áreas de código, dados e *heap* do processo.

⁴Como 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.

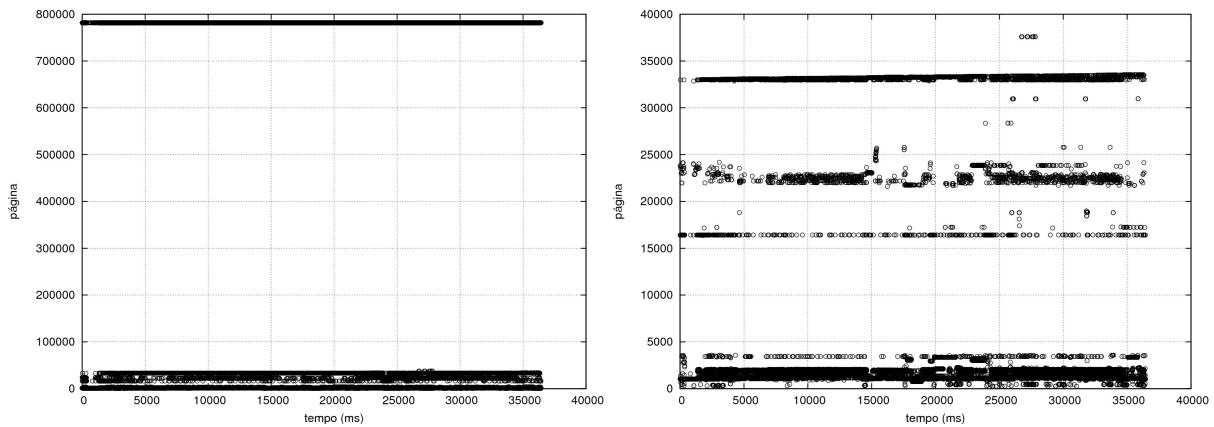


Figura 5.19: Distribuição dos acessos à memória do programa *gThumb*: visão geral (à esquerda) e detalhe da parte inferior (à direita).

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 muitas aplicações interativas, etc.

5.5 Fragmentação

Ao longo da vida de um sistema, áreas de memória são liberadas por processos que concluem sua execução e outras áreas são alocadas por novos processos, de forma contínua. Com isso, podem surgir áreas livres (vazios ou *buracos* na memória) entre os processos, o que constitui um problema conhecido como *fragmentação externa*. Esse problema somente afeta as estratégias de alocação que trabalham com blocos de tamanho variável, como a alocação contígua e a alocação segmentada. Por outro lado, a alocação paginada sempre trabalha com blocos de mesmo tamanho (os quadros e páginas), sendo por isso imune à fragmentação externa.

A fragmentação externa é prejudicial porque limita a capacidade de alocação de memória no sistema. A Figura 5.20 apresenta um sistema com alocação contígua de memória no qual ocorre fragmentação externa. Nessa Figura, observa-se que existem 68 MBytes de memória livre em quatro áreas separadas ($A_1 \dots A_4$), mas somente processos com até 28 MBytes podem ser alocados (usando a maior área livre, A_4). Além disso, quanto mais fragmentada estiver a memória livre, maior o esforço necessário para gerenciá-la: as áreas livres são mantidas em uma lista encadeada de área de memória, que é manipulada a cada pedido de alocação ou liberação de memória.

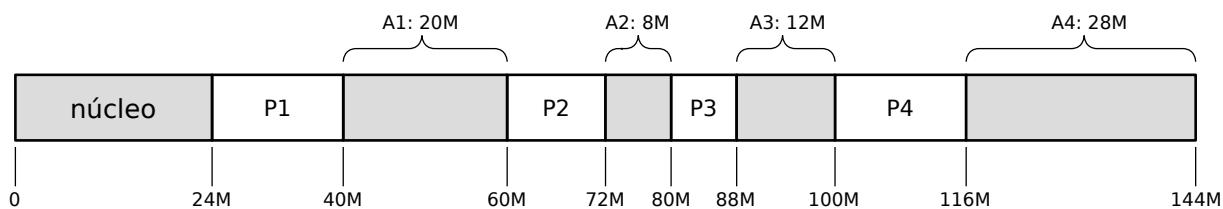


Figura 5.20: Memória com fragmentação externa.

Pode-se enfrentar o problema da fragmentação externa de duas formas: *minimizando* sua ocorrência, através de critérios de escolha das áreas a alocar, ou *desfragmentando* periodicamente a memória do sistema. Para minimizar a ocorrência de fragmentação externa, cada pedido de alocação deve ser analisado para encontrar a área de memória livre que melhor o atenda. Essa análise pode ser feita usando um dos seguintes critérios:

Melhor encaixe (best-fit) : consiste em escolher a menor área possível que possa atender à solicitação de alocação. Dessa forma, as áreas livres são usadas de forma otimizada, mas eventuais resíduos (sobras) podem ser pequenos demais para ter alguma utilidade.

Pior encaixe (worst-fit) : consiste em escolher sempre a maior área livre possível, de forma que os resíduos sejam grandes e possam ser usados em outras alocações.

Primeiro encaixe (first-fit) : 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.

Próximo encaixe (next-fit) : variante da anterior (*first-fit*) que consiste em percorrer a lista 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 de primeiro encaixe, sendo esta última bem mais rápida. A Figura 5.21 ilustra essas estratégias.

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 alocação (registrador base ou tabela de segmentos) devem ser ajustadas para refletir a nova posição do processo.

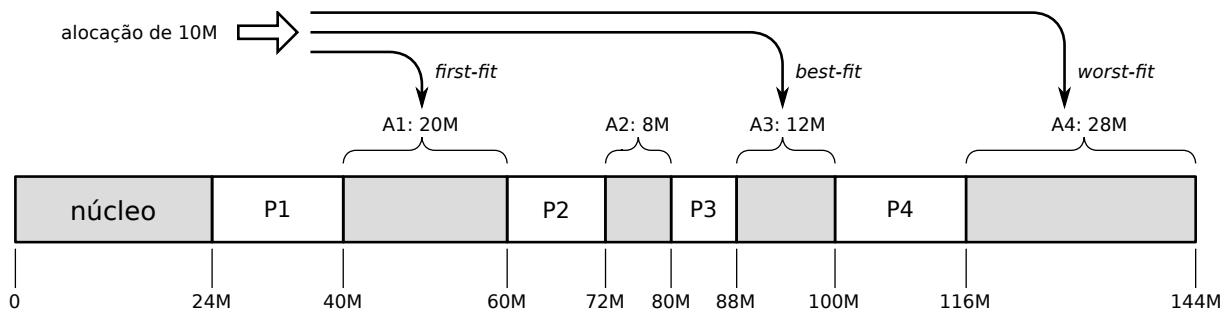


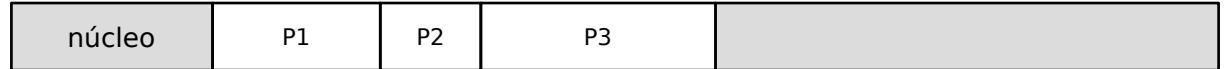
Figura 5.21: Estratégias para minimizar a fragmentação externa.

Obviamente, nenhum processo pode executar durante a desfragmentação. Portanto, é importante que esse procedimento seja executado rapidamente e com pouca frequência, para não interferir nas atividades normais do sistema. Como as possibilidades de movimentação de processos podem ser muitas, a desfragmentação deve ser tratada como um problema de otimização combinatória, cuja solução ótima pode ser difícil de calcular. A Figura 5.22 ilustra três possibilidades de desfragmentação de uma determinada situação de memória; as três alternativas produzem o mesmo resultado, mas apresentam custos distintos.

Situação inicial



Solução 1: deslocar P2 e P3 (custo: mover 60M)



Solução 2: deslocar P3 (custo: mover 40M)



Solução 3: deslocar P3 (custo: mover 20M)



Figura 5.22: Possibilidades de desfragmentação.

Além da fragmentação externa, que afeta as áreas livres entre os processos, as estratégias de alocação de memória também podem apresentar a *fragmentação interna*, que pode ocorrer dentro das áreas alocadas aos processos. A Figura 5.23 apresenta uma situação onde ocorre esse problema: um novo processo requisita uma área de memória com 4.900 Kbytes. Todavia, a área livre disponível tem 5.000 Kbytes. Se for

alocada exatamente a área solicitada pelo processo (situação A), sobrará um fragmento residual com 100 Kbytes, que é praticamente inútil para o sistema, pois é muito pequeno para acomodar novos processos. Além disso, essa área residual de 100 Kbytes deve ser incluída na lista de áreas livres, o que representa um custo de gerência desnecessário. Outra possibilidade consiste em “arredondar” o tamanho da área solicitada pelo processo para 5.000 Kbytes, ocupando totalmente aquela área livre (situação B). Assim, haverá uma pequena área de 100 Kbytes no final da memória do processo, que provavelmente não será usada por ele.

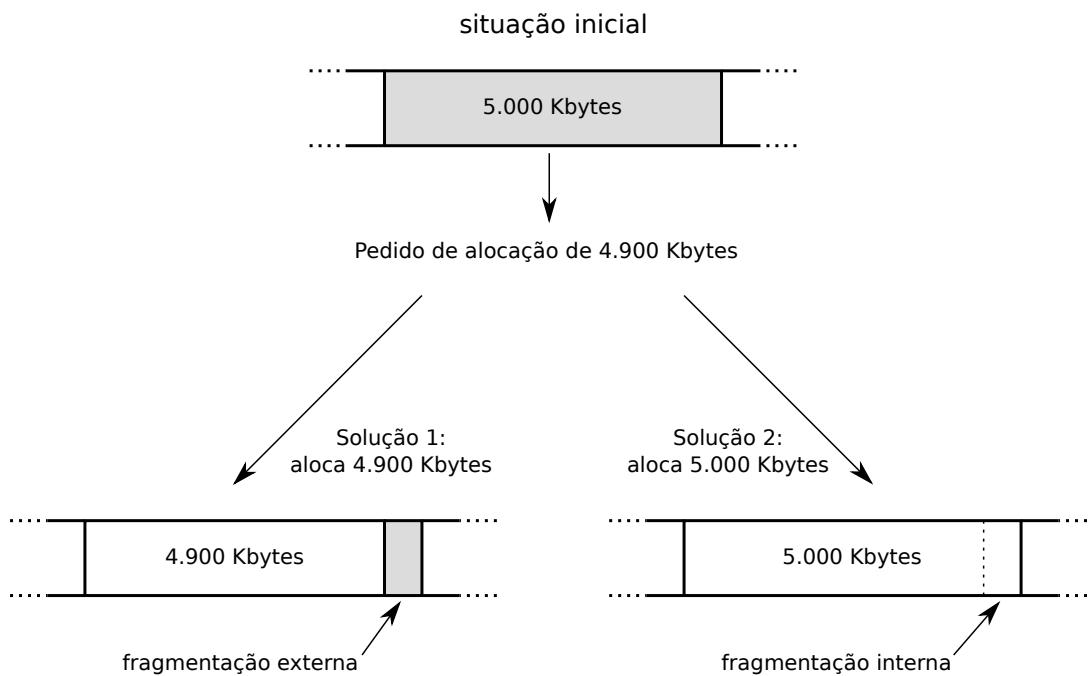


Figura 5.23: Fragmentação interna.

A fragmentação interna afeta todas as formas de alocação; as alocações contígua e segmentada sofrem menos com esse problema, pois o nível de arredondamento das alocações pode ser decidido caso a caso. No caso da alocação paginada, essa decisão não é possível, pois as alocações são feitas em páginas inteiras. Assim, em um sistema com páginas de 4 Kbytes (4.096 bytes), um processo que solicite a alocação de 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. Assim, 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.

5.6 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 dos protocolos POP3 ou IMAP terá um processo correspondente no servidor, para atender suas consultas de e-mail (Figura 5.24). Todos esses processos operam com dados distintos (pois atendem a usuários distintos), mas executam o mesmo código. Assim, centenas ou milhares de cópias do mesmo código executável poderão coexistir na memória do sistema.

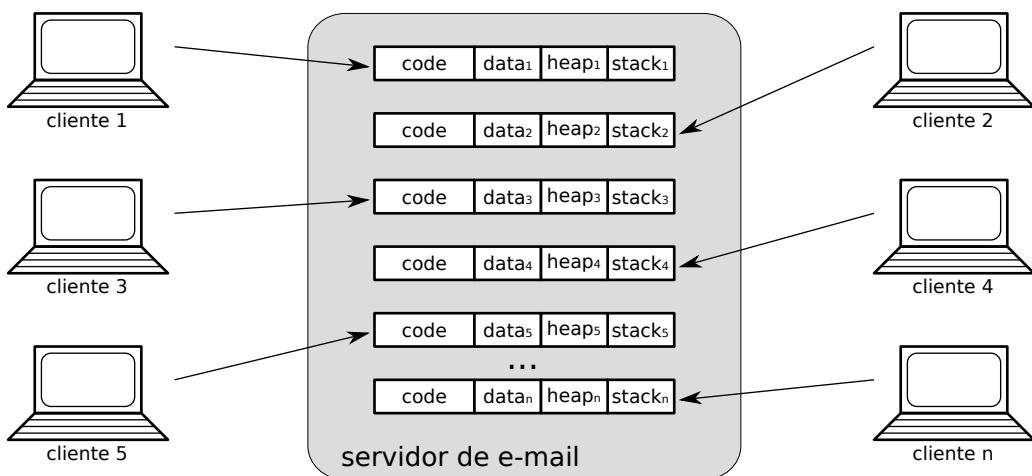


Figura 5.24: Várias instâncias do mesmo processo.

Conforme visto na Seção 5.2.2, a estrutura típica da memória de um processo contém áreas separadas para código, dados, pilha e *heap*. Normalmente, a área de código não precisa ter seu conteúdo modificado durante a execução, portanto geralmente essa área é protegida contra escritas (*read-only*). Assim, seria possível *compartilhar* essa área entre todos os processos que executam o mesmo código, economizando memória física.

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, bastaria fazer com que todos os segmentos de código dos processos apontem para o mesmo segmento da memória física, como indica a Figura 5.25. É importante observar que o compartilhamento é transparente para os processos: cada processo continua a acessar endereços lógicos em seu próprio segmento de código, buscando suas instruções a executar.

No caso da paginação, a unidade básica de compartilhamento é a página. Assim, as entradas das tabelas de páginas dos processos envolvidos são ajustadas 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 5.26 ilustra o compartilhamento de páginas entre processos.

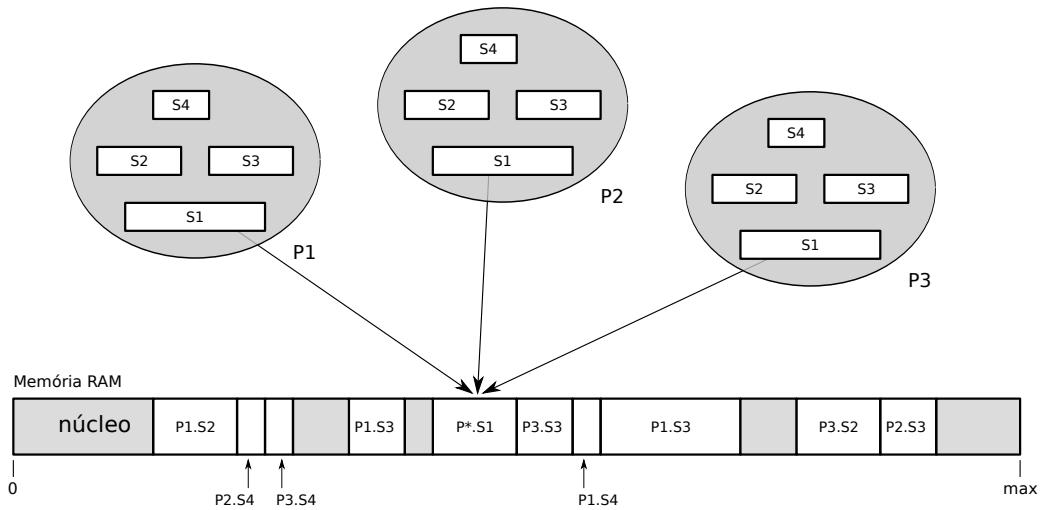


Figura 5.25: Compartilhamento de segmentos.

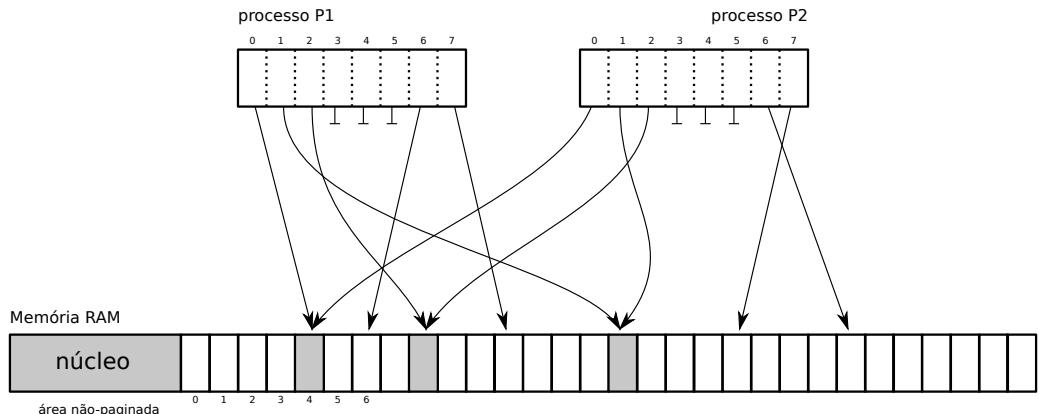


Figura 5.26: Compartilhamento de páginas.

O compartilhamento das áreas de código permite proporcionar uma grande economia no uso da memória física, sobretudo em servidores e sistemas multi-usuários. Por exemplo: consideremos um processador de textos que necessite de 100 MB de memória para executar, dos quais 60 MB são ocupados por código executável. Sem o compartilhamento de áreas de código, 10 instâncias do editor consumiriam 1.000 MB de memória; com o compartilhamento, esse consumo cairia para 460 MB ($60MB + 10 \times 40MB$).

O mecanismo de compartilhamento de memória não é usado apenas com áreas de código; em princípio, toda área de memória protegida contra escrita pode ser compartilhada, o que poderia incluir áreas de dados constantes, como tabelas de constantes, textos de ajuda, etc., proporcionando ainda mais economia de memória.

Uma forma mais agressiva de compartilhamento de memória é proporcionada pelo mecanismo denominado *copiar-ao-escrever* (COW - Copy-On-Write). Nele, todas as áreas de memória de um processo (segmentos ou páginas) são passíveis de compartilhamento por outros processos, à condição que ele ainda não tenha modificado seu conteúdo. A idéia central do mecanismo é simples:

1. ao carregar um novo processo em memória, o núcleo protege todas as áreas de memória do processo contra escrita (inclusive dados, pilha e *heap*), usando os flags da tabela de páginas (ou de segmentos);
2. quando o processo tentar escrever na memória, a MMU gera uma interrupção (negação de escrita) para o núcleo do sistema operacional;
3. o sistema operacional ajusta então os flags daquela área para permitir a escrita e devolve a execução ao processo, para ele poder continuar;
4. processos subsequentes idênticos ao primeiro, ao serem carregados em memória, serão mapeados sobre as mesmas áreas de memória física do primeiro processo que ainda estiverem protegidas contra escrita, ou seja, que ainda não foram modificadas por ele;
5. se um dos processos envolvidos tentar escrever em uma dessas áreas compartilhadas, a MMU gera uma interrupção para o núcleo;
6. o núcleo então faz uma cópia separada daquela área física para o processo que deseja escrever nela e desfaz seu compartilhamento, ajustando as tabelas do processo que provocou a interrupção. Os demais processos continuam compartilhando a área inicial.

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.

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 comum, sobre a qual 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. É importante observar que os endereços lógicos em cada processo poderão ser distintos, pois isso depende do mapeamento feito pela tabela de páginas (ou de segmentos) de cada processo; apenas os endereços físicos serão iguais. Portanto, ponteiros (variáveis que contêm endereços lógicos) armazenados na área compartilhada terão significado para o processo que os escreveu, mas não necessariamente para os demais processos que acessam aquela área. A Seção 3.4.3 traz informações mais detalhadas sobre a comunicação entre processos através de memória compartilhada.

5.7 Memória virtual

Um problema constante nos computadores é a disponibilidade de memória física: os programas se tornam cada vez maiores e cada vez mais processos executam simultaneamente, ocupando a memória disponível. 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\$50/GByte no mercado americano, em 2007) e que consome uma quantidade significativa de energia, aumentar sua capacidade nem sempre é uma opção factível.

Observando o comportamento de um sistema computacional, constata-se que nem todos os processos estão constantemente ativos, e que nem todas as áreas de memória estão constantemente sendo usadas. Por isso, as áreas de memória pouco acessadas poderiam ser transferidas para um meio de armazenamento mais barato e abundante, como um disco rígido (U\$0,50/GByte) ou um banco de memória *flash* (U\$10/GByte)⁵, liberando a memória RAM para outros usos. Quando um processo proprietário de uma dessas áreas precisar acessá-la, ela deve ser transferida de volta para a memória RAM. O uso de um armazenamento externo como extensão da memória RAM se chama *memória virtual*; essa estratégia pode ser implementada de forma eficiente e transparente para processos, usuários e programadores.

5.7.1 Mecanismo básico

Nos primeiros sistemas a implementar estratégias de memória virtual, processos inteiros eram transferidos da memória para o disco rígido e vice-versa. Esse procedimento, denominado *troca* (*swapping*) permite liberar grandes áreas de memória a cada transferência, e se justifica no caso de um armazenamento com tempo de acesso muito elevado, como os antigos discos rígidos. Os sistemas atuais raramente transferem processos inteiros para o disco; geralmente as transferências são feitas por páginas ou grupos de páginas, em um procedimento denominado *paginação* (*paging*), detalhado a seguir.

Normalmente, o mecanismo de memória virtual se baseia em páginas ao invés de segmentos. As páginas têm um tamanho único e fixo, o que permite simplificar os algoritmos de escolha de páginas a remover, os mecanismos de transferência para o disco e também a formatação da área de troca no disco. A otimização desses fatores seria bem mais complexa e menos efetiva caso as operações de troca fossem baseadas em segmentos, que têm tamanho variável.

A idéia central do mecanismo de memória virtual em sistemas com memória paginada consiste em retirar da memória principal as páginas menos usadas, salvando-as em uma área do disco rígido reservada para esse fim. Essa operação é feita periodicamente, de modo reativo (quando a quantidade de memória física disponível cai abaixo de um certo limite) ou pró-ativo (aproveitando os períodos de baixo uso do sistema para retirar da memória as páginas pouco usadas). As páginas a retirar são escolhidas de acordo com algoritmos de substituição de páginas, discutidos na Seção 5.7.3. As entradas das tabelas de páginas relativas às páginas transferidas para o disco devem então ser ajustadas de forma a referenciar os conteúdos correspondentes no disco rígido. Essa situação está ilustrada de forma simplificada na Figura 5.27.

O armazenamento externo das páginas pode ser feito em um disco exclusivo para esse fim (usual em servidores de maior porte), em uma partição do disco principal

⁵Estes valores são apenas indicativos, variando de acordo com o fabricante e a tecnologia envolvida.

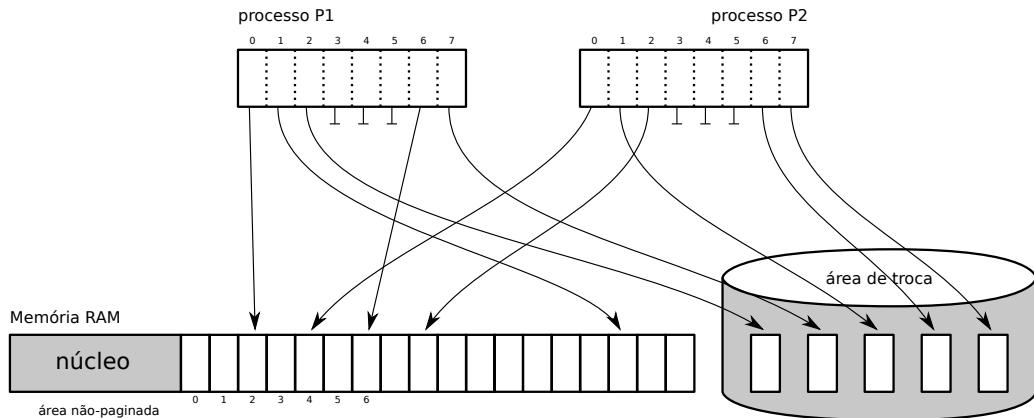


Figura 5.27: Memória virtual paginada.

(usual no Linux e outros UNIX) ou em um arquivo reservado dentro do sistema de arquivos do disco principal da máquina, geralmente oculto (como no Windows NT e sucessores). Em alguns sistemas, é possível usar uma área de troca remota, em um servidor de arquivos 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 partição de disco, essa área geralmente é formatada usando uma estrutura de sistema de arquivos otimizada para o armazenamento e recuperação rápida das páginas.

As páginas que foram transferidas da memória para o disco provavelmente serão necessárias no futuro, pois seus processos proprietários provavelmente continuam vivos. Quando um processo tentar acessar uma página ausente, esta deve ser transferida de volta para a memória para permitir seu acesso, de forma transparente ao processo. Conforme exposto na Seção 5.3.4, quando um processo acessa uma página, a MMU verifica se a mesma está mapeada na memória RAM e, em caso positivo, faz o acesso ao endereço físico correspondente. Caso contrário, a MMU gera uma interrupção de falta de página (*page fault*) que força o desvio da execução para o sistema operacional. Nesse instante, o sistema deve verificar se a página solicitada não existe ou se foi transferida para o disco, usando os flags de controle da respectiva entrada da tabela de páginas. Caso a página não exista, o processo tentou acessar um endereço inválido e deve ser abortado. Por outro lado, caso a página solicitada tenha sido transferida para o disco, 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 5.28 apresenta as principais ações desenvolvidas pelo mecanismo de memória virtual.

Nesse procedimento aparentemente simples há duas questões importantes. Primeiro, caso a memória principal já esteja cheia, uma ou mais páginas deverão ser removidas 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, mantém um processo *daemon*

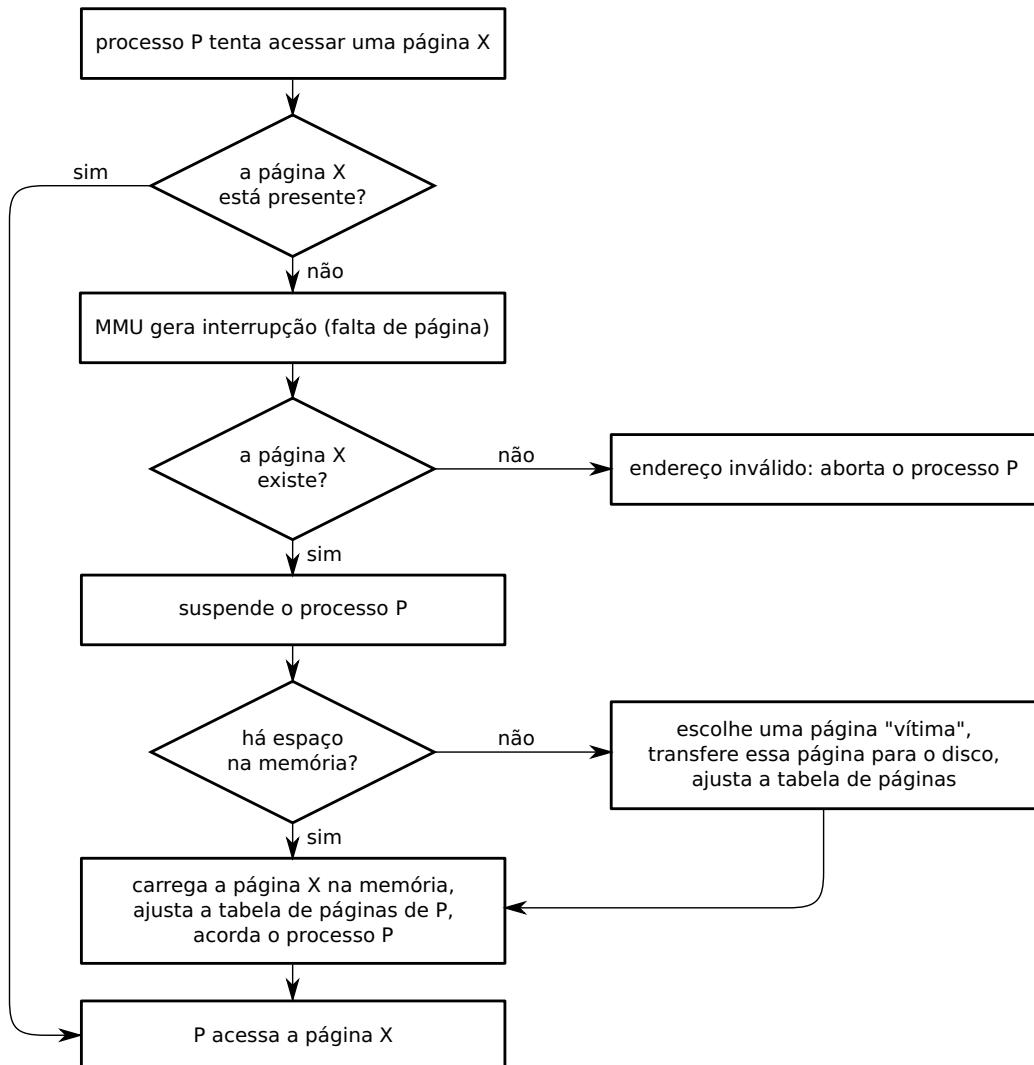


Figura 5.28: Ações do mecanismo de memória virtual.

com a finalidade de escolher e transferir páginas para o disco, ativado sempre que a quantidade de memória livre estiver abaixo de um limite mínimo.

Segundo, 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 re-executada. No caso de instruções simples, envolvendo apenas um endereço de memória sua re-execução 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 que estiver faltando. A maioria dos processadores atuais provê registradores especiais que auxiliam nessa tarefa.

5.7.2 Eficiência de uso

O mecanismo de memória virtual 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 5.1, 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 (60×10^{-9} s) e cujo disco de troca tem um tempo de acesso de 6 ms (6×10^{-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:

$$\begin{aligned} t_{\text{médio}} &= \frac{(999.999 \times 60\text{ns}) + 6\text{ms} + 60\text{ns}}{1.000.000} \\ &= \frac{10^6 \times 60 \times 10^{-9} + 6 \times 10^{-3}}{10^6} \\ t_{\text{médio}} &= 66\text{ns} \end{aligned}$$

Caso a memória esteja saturada, o tempo médio será maior:

$$\begin{aligned} t_{\text{médio}} &= \frac{(999.999 \times 60\text{ns}) + 2 \times 6\text{ms} + 60\text{ns}}{1.000.000} \\ &= \frac{10^6 \times 60 \times 10^{-9} + 2 \times 6 \times 10^{-3}}{10^6} \\ t_{\text{médio}} &= 72\text{ns} \end{aligned}$$

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 180 ns, ou seja, três vezes mais lento. 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 5.7.6).
- 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 5.4), 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 tarefa dos algoritmos apresentados na Seção 5.7.3.

5.7.3 Algoritmos de substituição de páginas

A escolha correta das páginas a remover da memória física é um fator essencial para a eficiência do mecanismo de memória virtual. Mais escolhas poderão remover da memória páginas muito usadas, aumentando a taxa de faltas de página e diminuindo o desempenho do sistema. Vários critérios podem ser usados para escolher “vítimas”, ou seja, páginas a transferir da memória para a área de troca no disco:

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 memória virtual, 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.

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 serão apresentados na sequência.

Uma ferramenta importante para o estudo desses algoritmos é a *cadeia de referências* (*reference string*), que indica a sequência de páginas acessadas por um processo durante

sua execução. 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 e assim comparar sua eficiência.

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, extraída de [Tanenbaum, 2003]:

0, 2, 1, 3, 5, 4, 6, 3, 7, 4, 7, 3, 3, 5, 5, 3, 1, 1, 1, 7, 1, 3, 4, 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.

Algoritmo FIFO

Um critério básico 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 5.2. Nesse caso, o algoritmo gera no total 15 faltas de página.

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 páginas contendo bibliotecas dinâmicas compartilhadas por muitos processos, ou páginas de processos servidores lançados durante a inicialização (*boot*) da máquina.

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 idéia 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 17 faltas de página, nenhum outro algoritmo irá

t	antes			página	depois			faltas	ação realizada
	q_0	q_1	q_2		q_0	q_1	q_2		
1	-	-	-	0	0	-	-	★	p_0 é carregada no quadro vazio q_0
2	0	-	-	2	0	2	-	★	p_2 é carregada em q_1
3	0	2	-	1	0	2	1	★	p_1 é carregada em q_2
4	0	2	1	3	3	2	1	★	p_3 substitui p_0 (a mais antiga na memória)
5	3	2	1	5	3	5	1	★	p_5 substitui p_2 (idem)
6	3	5	1	4	3	5	4	★	p_4 substitui p_1
7	3	5	4	6	6	5	4	★	p_6 substitui p_3
8	6	5	4	3	6	3	4	★	p_3 substitui p_5
9	6	3	4	7	6	3	7	★	p_7 substitui p_4
10	6	3	7	4	4	3	7	★	p_4 substitui p_6
11	4	3	7	7	4	3	7		p_7 está na memória
12	4	3	7	3	4	3	7		p_3 está na memória
13	4	3	7	3	4	3	7		p_3 está na memória
14	4	3	7	5	4	5	7	★	p_5 substitui p_3
15	4	5	7	5	4	5	7		p_5 está na memória
16	4	5	7	3	4	5	3	★	p_3 substitui p_7
17	4	5	3	1	1	5	3	★	p_1 substitui p_4
18	1	5	3	1	1	5	3		p_1 está na memória
19	1	5	3	1	1	5	3		p_1 está na memória
20	1	5	3	7	1	7	3	★	p_7 substitui p_5
21	1	7	3	1	1	7	3		p_1 está na memória
22	1	7	3	3	1	7	3		p_3 está na memória
23	1	7	3	4	1	7	4	★	p_4 substitui p_3
24	1	7	4	1	1	7	4		p_1 está na memória

Tabela 5.2: Aplicação do algoritmo de substituição FIFO.

gerar menos de 17 faltas de página ao tratar a 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 5.3. Nesse caso, o algoritmo gera 11 faltas de página, ou seja, 4 a menos que o algoritmo FIFO.

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 acessadas “mais longe” no futuro do processo, o algoritmo LRU busca as páginas que foram acessadas “mais longe” no seu passado.

t	antes			página	depois			faltas	ação realizada
	q_0	q_1	q_2		q_0	q_1	q_2		
1	-	-	-	0	0	-	-	★	p_0 é carregada no quadro vazio q_0
2	0	-	-	2	0	2	-	★	p_2 é carregada em q_1
3	0	2	-	1	0	2	1	★	p_1 é carregada em q_2
4	0	2	1	3	3	2	1	★	p_3 substitui p_0 (não será mais acessada)
5	3	2	1	5	3	5	1	★	p_5 substitui p_2 (não será mais acessada)
6	3	5	1	4	3	5	4	★	p_4 substitui p_1 (só será acessada em $t = 17$)
7	3	5	4	6	3	6	4	★	p_6 substitui p_5 (só será acessada em $t = 14$)
8	3	6	4	3	3	6	4		p_3 está na memória
9	3	6	4	7	3	7	4	★	p_7 substitui p_6 (não será mais acessada)
10	3	7	4	4	3	7	4		p_4 está na memória
11	3	7	4	7	3	7	4		p_7 está na memória
12	3	7	4	3	3	7	4		p_3 está na memória
13	3	7	4	3	3	7	4		p_3 está na memória
14	3	7	4	5	3	7	5	★	p_5 substitui p_4 (só será acessada em $t = 23$)
15	3	7	5	5	3	7	5		p_5 está na memória
16	3	7	5	3	3	7	5		p_3 está na memória
17	3	7	5	1	3	7	1	★	p_1 substitui p_5 (não será mais acessada)
18	3	7	1	1	3	7	1		p_1 está na memória
19	3	7	1	1	3	7	1		p_1 está na memória
20	3	7	1	7	3	7	1		p_7 está na memória
21	3	7	1	1	3	7	1		p_1 está na memória
22	3	7	1	3	3	7	1		p_3 está na memória
23	3	7	1	4	4	7	1	★	p_4 substitui p_3 (não será mais acessada)
24	4	7	1	1	4	7	1		p_1 está na memória

Tabela 5.3: Aplicação do algoritmo de substituição ótimo.

A aplicação do algoritmo LRU à cadeia de referências apresentada na Seção anterior, considerando uma memória física com 3 quadros, é apresentada na Tabela 5.4. Nesse caso, o algoritmo gera 14 faltas de página (três faltas a mais que o algoritmo ótimo).

O gráfico da Figura 5.29 permite a comparação dos algoritmos OPT, FIFO e LRU sobre a cadeia de referências em estudo, 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 FIFO.

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 se verifica na prática, sobretudo se os processos respeitam o princípio da localidade de referência (Seção 5.4).

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

t	antes			página	depois			faltas	ação realizada
	q_0	q_1	q_2		q_0	q_1	q_2		
1	-	-	-	0	0	-	-	★	p_0 é carregada no quadro vazio q_0
2	0	-	-	2	0	2	-	★	p_2 é carregada em q_1
3	0	2	-	1	0	2	1	★	p_1 é carregada em q_2
4	0	2	1	3	3	2	1	★	p_3 substitui p_0 (há mais tempo sem acessos)
5	3	2	1	5	3	5	1	★	p_5 substitui p_2 (idem)
6	3	5	1	4	3	5	4	★	p_4 substitui p_1 (...)
7	3	5	4	6	6	5	4	★	p_6 substitui p_3
8	6	5	4	3	6	3	4	★	p_3 substitui p_5
9	6	3	4	7	6	3	7	★	p_7 substitui p_4
10	6	3	7	4	4	3	7	★	p_4 substitui p_6
11	4	3	7	7	4	3	7		p_7 está na memória
12	4	3	7	3	4	3	7		p_3 está na memória
13	4	3	7	3	4	3	7		p_3 está na memória
14	4	3	7	5	5	3	7	★	p_5 substitui p_4
15	5	3	7	5	5	3	7		p_5 está na memória
16	5	3	7	3	5	3	7		p_3 está na memória
17	5	3	7	1	5	3	1	★	p_1 substitui p_7
18	5	3	1	1	5	3	1		p_1 está na memória
19	5	3	1	1	5	3	1		p_1 está na memória
20	5	3	1	7	7	3	1	★	p_7 substitui p_5
21	7	3	1	1	7	3	1		p_1 está na memória
22	7	3	1	3	7	3	1		p_3 está na memória
23	7	3	1	4	4	3	1	★	p_4 substitui p_7
24	4	3	1	1	4	3	1		p_1 está na memória

Tabela 5.4: Aplicação do algoritmo de substituição LRU.

exigiria muito processamento. Portanto, a maioria dos sistemas operacionais reais implementa algoritmos baseados em aproximações do LRU.

As próximas seções apresentam 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].

Algoritmo da segunda chance

O algoritmo FIFO 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 5.3.4) 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

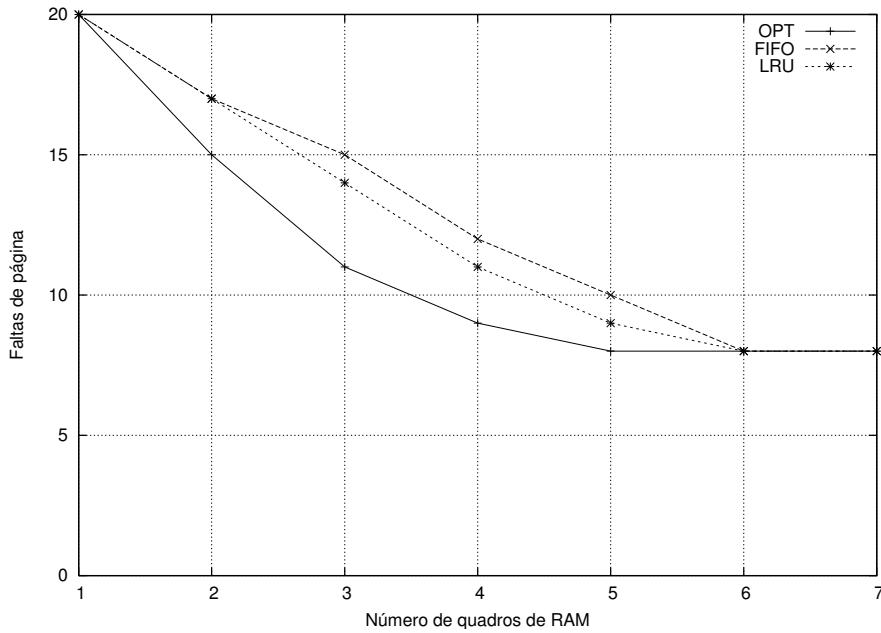


Figura 5.29: Comparação dos algoritmos de substituição de páginas.

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 fila circular de números de página, ordenados de acordo com seu ingresso na memória. Um ponteiro percorre a fila 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 zero. Essa implementação é conhecida como *algoritmo do relógio* e pode ser vista na Figura 5.30.

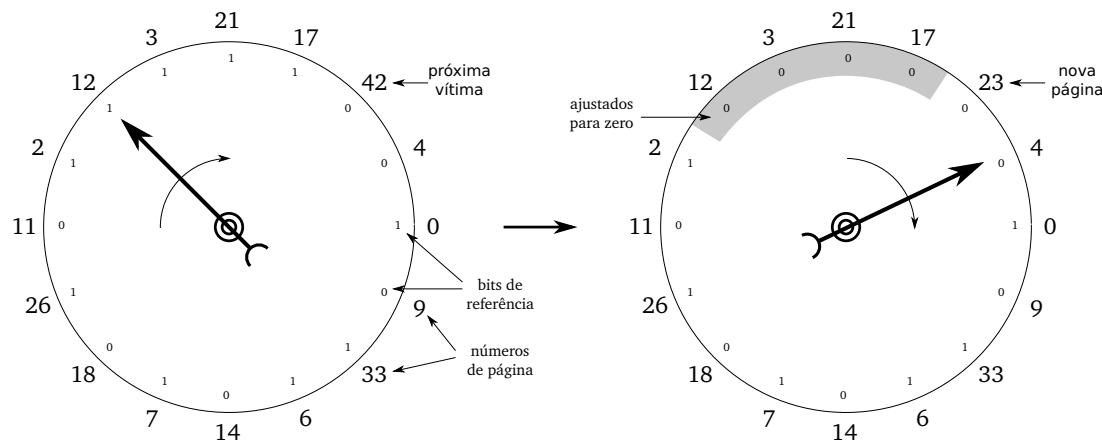


Figura 5.30: Algoritmo da segunda chance (ou do relógio).

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 5.3.4), 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 usadasativamente 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.

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 agrupa 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.

p_1	$\begin{bmatrix} R \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$	com	$\begin{bmatrix} contadores \\ 0000\ 0011 \\ 0011\ 1101 \\ 1010\ 1000 \\ 1110\ 0011 \end{bmatrix} \quad (3) \quad (61) \quad (168) \quad (227)$	\Rightarrow	$\begin{bmatrix} R \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$	e	$\begin{bmatrix} contadores \\ 0000\ 0001 \\ 1001\ 1110 \\ 0101\ 0100 \\ 1111\ 0001 \end{bmatrix} \quad (1) \quad (158) \quad (84) \quad (241)$
-------	---	-----	---	---------------	---	---	---

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].

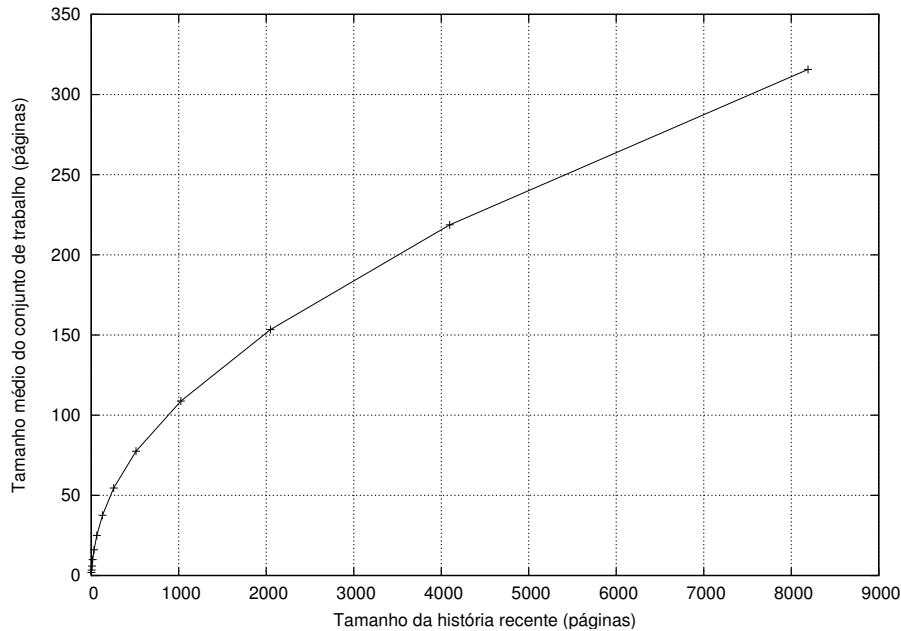
5.7.4 Conjunto de trabalho

A localidade de referências (estudada na Seção 5.4) 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, Denning, 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 5.7.3. 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 5.5.

O tamanho e a composição do conjunto de trabalho dependem do número de páginas consideradas em sua história recente (o valor n na Tabela 5.5). 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 5.5: 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 5.31, que mostra o tamanho médio dos conjuntos de trabalhos observados na execução do programa *gThumb* (analisado na Seção 5.4), 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

t	página	$ws(n = 3)$	$ws(n = 4)$	$ws(n = 5)$
1	0	{0}	{0}	{0}
2	2	{0, 2}	{0, 2}	{0, 2}
3	1	{0, 2, 1}	{0, 2, 1}	{0, 2, 1}
4	3	{2, 1, 3}	{0, 2, 1, 3}	{0, 2, 1, 3}
5	5	{1, 3, 5}	{2, 1, 3, 5}	{0, 2, 1, 3, 5}
6	4	{3, 5, 4}	{1, 3, 5, 4}	{2, 1, 3, 5, 4}
7	6	{5, 4, 6}	{3, 5, 4, 6}	{1, 3, 5, 4, 6}
8	3	{4, 6, 3}	{5, 4, 6, 3}	{5, 4, 6, 3}
9	7	{6, 3, 7}	{4, 6, 3, 7}	{5, 4, 6, 3, 7}
10	4	{3, 7, 4}	{6, 3, 7, 4}	{6, 3, 7, 4}
11	7	{4, 7}	{3, 4, 7}	{6, 3, 4, 7}
12	3	{4, 7, 3}	{4, 7, 3}	{4, 7, 3}
13	3	{7, 3}	{4, 7, 3}	{4, 7, 3}
14	5	{3, 5}	{7, 3, 5}	{4, 7, 3, 5}
15	5	{3, 5}	{3, 5}	{7, 3, 5}
16	3	{5, 3}	{5, 3}	{5, 3}
17	1	{5, 3, 1}	{5, 3, 1}	{5, 3, 1}
18	1	{3, 1}	{5, 3, 1}	{5, 3, 1}
19	1	{1}	{3, 1}	{5, 3, 1}
20	7	{1, 7}	{1, 7}	{3, 1, 7}
21	1	{7, 1}	{7, 1}	{3, 7, 1}
22	3	{7, 1, 3}	{7, 1, 3}	{7, 1, 3}
23	4	{1, 3, 4}	{7, 1, 3, 4}	{7, 1, 3, 4}
24	1	{3, 4, 1}	{3, 4, 1}	{7, 3, 4, 1}

Tabela 5.5: Conjuntos de trabalho ws para $n = 3$, $n = 4$ e $n = 5$.Figura 5.31: Tamanho do conjunto de trabalho do programa *gThumb*.

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], que modifica o algoritmo do relógio (Seção 5.7.3) da seguinte forma:

1. Uma data de último acesso $t_a(p)$ é associada a cada página p da fila circular; essa data pode ser atualizada quando o ponteiro do relógio visita a página.
2. Define-se um prazo de validade τ para as páginas, geralmente entre dezenas e centenas de mili-segundos; a “idade” $i(p)$ de uma página p é definida como a diferença entre sua data de último acesso $t_a(p)$ e o instante corrente t_c ($i(p) = t_c - t_a(p)$).
3. Quando há necessidade de substituir páginas, o ponteiro percorre a fila buscando páginas candidatas:
 - (a) Ao encontrar uma página referenciada (com $R = 1$), sua data de último acesso é atualizada com o valor correto do tempo ($t_a(p) = t_c$), seu bit de referência é limpo ($R = 0$) e o ponteiro do relógio avança, ignorando aquela página.
 - (b) Ao encontrar uma página não referenciada (com $R = 0$), se sua idade for menor que τ , a página está no conjunto de trabalho; caso contrário, ela é considerada fora do conjunto de trabalho e pode ser substituída.
4. Caso o ponteiro dê uma volta completa na fila e não encontre páginas com idade maior que τ , a página mais antiga (que tiver o menor $t_a(p)$) encontrada na volta anterior é substituída.
5. Em todas as escolhas, dá-se preferência a páginas não-modificadas ($M = 0$), pois seu conteúdo já está salvo no disco.

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 fila 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).

5.7.5 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 5.32, 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.

$0, 1, 2, 3, 4, 0, 1, 2, 5, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 0, 1, 2, 5, 0, 1, 2, 3, 4, 5$

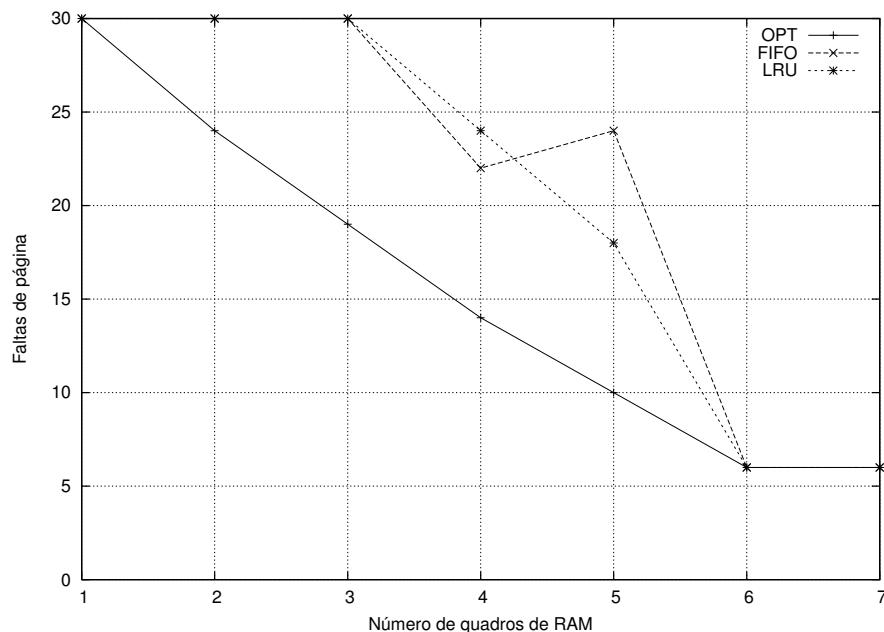


Figura 5.32: A anomalia de Belady.

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].

5.7.6 Thrashing

Na Seção 5.7.2, 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 5.7.4, 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, Denning, 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 5.33 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 região 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.

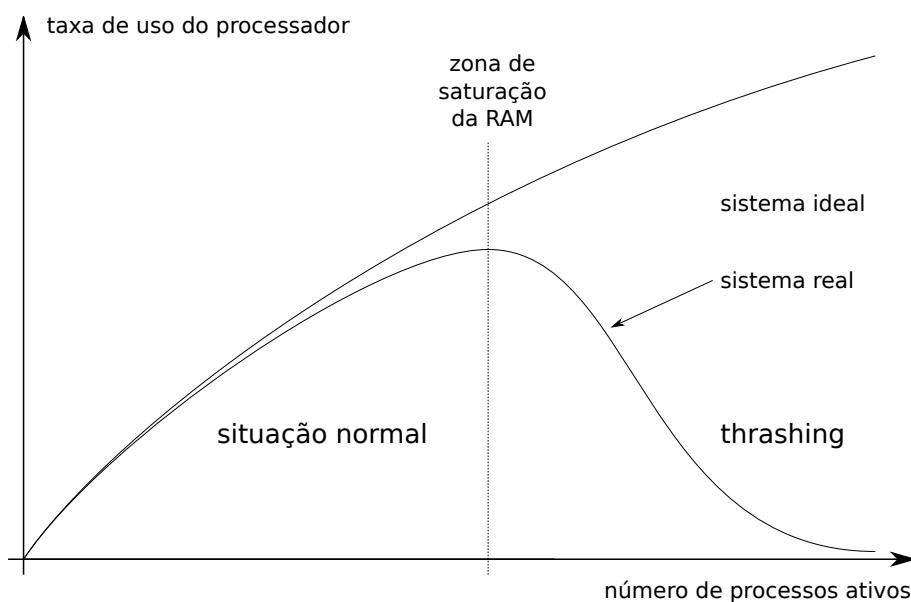


Figura 5.33: *Thrashing* no uso da memória RAM.

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.

Capítulo 6

Gerência de arquivos

Um sistema operacional tem por finalidade permitir que o usuários do computador executem aplicações, como editores de texto, jogos, reprodutores de áudio e vídeo, etc. Essas aplicações processam informações como textos, músicas e filmes, armazenados sob a forma de arquivos em um disco rígido ou outro meio. Este módulo apresenta a noção de arquivo, suas principais características e formas de acesso, a organização de arquivos em diretórios e as técnicas usadas para criar e gerenciar arquivos nos dispositivos de armazenamento.

6.1 Arquivos

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. Geralmente essas informações devem ser armazenadas em um dispositivo não-volátil, que preserve seu conteúdo mesmo quando o computador estiver desligado. Para simplificar o armazenamento e busca de informações, surgiu o conceito de *arquivo*, que será discutido a seguir.

6.1.1 O conceito de arquivo

Um arquivo é basicamente um conjunto de dados armazenados em um dispositivo físico não-volátil, com um nome ou outra referência que permita sua localização posterior. 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 ASCII com alguns bytes até sequências de vídeo com dezenas de gigabytes, ou mesmo mais.

Como um dispositivo de armazenamento pode conter milhões de arquivos, estes são organizados em estruturas hierárquicas denominadas *diretórios* (conforme ilustrado na Figura 6.1 e discutido mais detalhadamente na Seção 6.3.1). A organização física

e lógica dos arquivos e diretórios dentro de um dispositivo é denominada *sistema de arquivos*. Um sistema de arquivos pode ser visto como uma imensa estrutura de dados armazenada de forma persistente em um 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, máquinas fotográficas digitais e leitores MP3). A organização dos sistemas de arquivos será discutida na Seção 6.4.

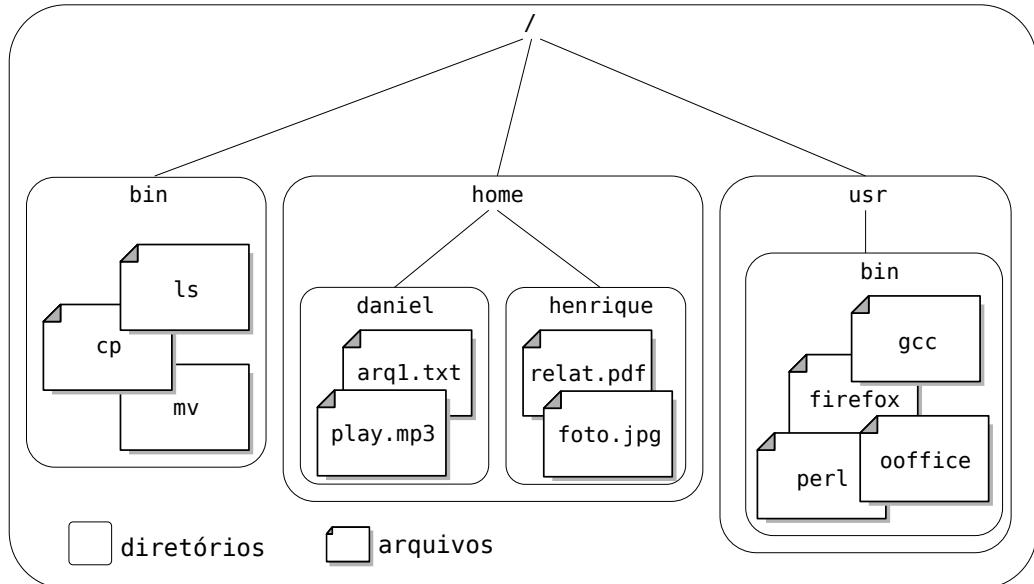


Figura 6.1: Arquivos organizados em diretórios dentro de um dispositivo.

6.1.2 Atributos

Conforme apresentado, um arquivo é uma unidade de armazenamento de informações que podem ser dados, código executável, etc. Cada arquivo é caracterizado por um conjunto de *atributos*, que podem variar de acordo com o sistema de arquivos utilizado. Os atributos mais usuais são:

Nome: string de caracteres que identifica o arquivo para o usuário, como "foto1.jpg", "relatório.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, em bytes ou registros;

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 multi-usuários, 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;

Outros atributos: vários outros atributos podem ser associados a um arquivo, por exemplo para indicar se 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.

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 exprimir 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 conteúdo¹: `relat-v1.txt`, `relat-v2.txt`, etc.

6.1.3 Operações

As aplicações e o sistema operacional usam arquivos para armazenar e recuperar dados. O uso dos 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 seus atributos (nome, localização, proprietário, permissões de acesso, 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 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 no final do arquivo ou sobreescriver dados já existentes;

¹Alguns sistemas operacionais, como o *TOPS-20* e o *OpenVMS*, possuem sistemas de arquivos com suporte automático a múltiplas versões do mesmo arquivo.

Mudar atributos: para modificar outras características do arquivo, como nome, localização, proprietário, permissões, etc.

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 na memória do núcleo;

Remover: para eliminar o arquivo do dispositivo, descartando seus dados e liberando o espaço ocupado por ele.

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.

6.1.4 Formatos

Em sua forma mais simples, um arquivo contém basicamente uma sequência de bytes, que pode estar estruturada de diversas formas para representar diferentes tipos de informação. O formato ou estrutura interna de um arquivo pode ser definido – e reconhecido – pelo núcleo do sistema operacional ou somente pelas aplicações. 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 sem um significado específico, cabendo às aplicações interpretá-los.

Os arquivos de dados convencionais são estruturados pelas aplicações para armazenar os mais diversos tipos de informações, como imagens, sons e documentos. Uma aplicação pode definir um formato próprio de armazenamento ou seguir formatos padronizados. Por exemplo, há um grande número de formatos públicos 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 o formato PSD (do editor *Adobe Photoshop*) e o formato XCF (do editor gráfico *GIMP*). A adoção de um formato proprietário ou exclusivo dificulta a ampla utilização das informações armazenadas, pois somente aplicações que reconheçam aquele formato conseguem ler corretamente as informações contidas no arquivo.

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 6.2 ilustra a estrutura interna desses dois tipos de arquivos.

Nos sistemas operacionais cujo núcleo não suporta arquivos estruturados como registros, essa funcionalidade pode ser facilmente obtida através de bibliotecas específicas

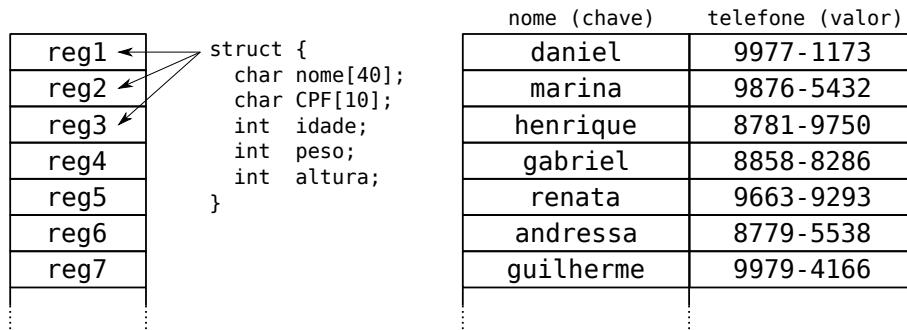


Figura 6.2: Arquivos estruturados: registros em sequência e registros indexados.

ou do suporte de execução de algumas linguagens de programação. Por exemplo, a biblioteca *Berkeley DB* disponível em plataformas UNIX oferece suporte à indexação de registros sobre arquivos UNIX convencionais.

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 muito 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 ASCII 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 da seguinte forma² em um ambiente 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 r l d "
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 }
```

²Listagem 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 da seguinte forma em um sistema DOS/Windows:

```

1 0000 69 6e 74 20 6d 61 69 6e 28 29 0d 0a 7b 0d 0a 20
2      i n t _ _ m a i n ( ) \r \n { \r \n _ 
3 0010 20 70 72 69 6e 74 66 28 22 48 65 6c 6c 6f 2c 20
4      _ p r i n t f ( " H e l l o , _ 
5 0020 77 6f 72 6c 64 5c 6e 22 29 3b 0d 0a 20 20 65 78
6      w o r l d \ n " ) ; \r \n _ _ e x
7 0030 69 74 28 30 29 3b 0d 0a 7d 0d 0a
8      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.

Arquivos executáveis

Um arquivo executável 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 executável ou biblioteca depende do sistema operacional para o qual foi definido. Os formatos de executáveis mais populares 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 de relocação de código.
- **PE (*Portable Executable*)**: é o formato usado para executáveis e bibliotecas na plataforma Windows. Consiste basicamente em uma adaptação do antigo formato COFF usado em plataformas UNIX.

A Figura 6.3 ilustra a estrutura interna de um arquivo executável no formato ELF, usado tipicamente em sistemas UNIX (Linux, Solaris, etc.). Esse arquivo é dividido em seções, que representam trechos de código e dados sujeitos a ligação dinâmica e relocação; as seções são agrupadas em segmentos, de forma a facilitar a carga em memória do código e o lançamento do processo.

Além de executáveis e bibliotecas, o núcleo de um sistema operacional costuma reconhecer alguns tipos de arquivos não convencionais, como diretórios, atalhos (*links*), dispositivos físicos e estruturas de comunicação do núcleo, como *sockets*, *pipes* e filas de mensagens (vide Seção 6.1.5).

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

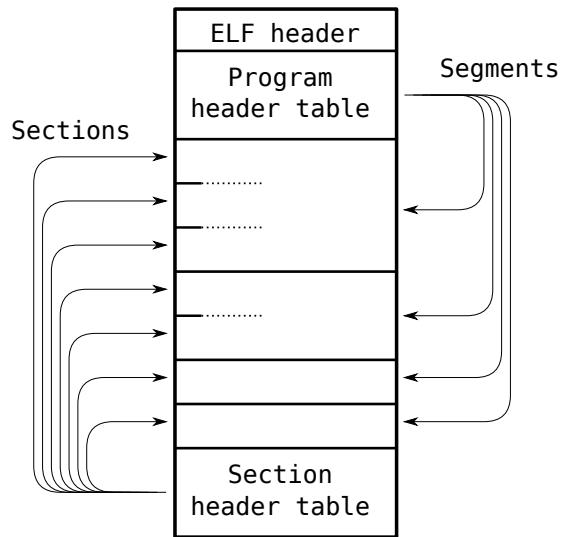


Figura 6.3: Estrutura interna de um arquivo executável em formato ELF [Levine, 2000].

pode ser visto como uma simples sequência de bytes, como é possível saber que tipo de informação essa sequência representa? Uma solução simples para esse problema consiste em indicar o tipo do conteúdo como parte do nome do arquivo: um arquivo “praia.jpg” provavelmente contém uma imagem em formato JPEG, enquanto um arquivo “entrevista.mp3” contém áudio em formato MP3. Essa estratégia, amplamente utilizada em muitos sistemas operacionais, foi introduzida nos anos 1980 pelo sistema operacional DOS. Naquele sistema, os arquivos eram nomeados segundo uma abordagem denominada “8.3”, ou seja, 8 caracteres seguidos de um ponto (“.”) e mais 3 caracteres de extensão, para definir o tipo do arquivo.

Outra abordagem, frequentemente usada em sistemas UNIX, é o uso de alguns bytes no início de cada arquivo para a definição de seu tipo. Esses bytes iniciais são denominados “números mágicos” (*magic numbers*), e são usados em muitos tipos de arquivos, como exemplificado na Tabela 6.1:

Tabela 6.1: Números mágicos de alguns tipos de arquivos

Tipo de arquivo	bytes iniciais	Tipo de arquivo	bytes iniciais
Documento PostScript	%!	Documento PDF	%PDF
Imagen GIF	GIF89a	Imagen JPEG	0xFFD8
Música MIDI	MThd	Classes Java (JAR)	0xCAFEBAE

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 desconhecidos 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 indicar o conteúdo de cada arquivo. Por exemplo, o sistema operacional MacOS 9 definia 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 6.2.

Tabela 6.2: 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 (<i>Comma-separated Values</i>)
text/html	Texto HTML
text/plain	Texto puro
text/rtf	Texto em formato RTF (<i>Rich Text Format</i>)
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 recuperados de páginas Web. Alguns sistemas operacionais, como o BeOS e o MacOS X, definem atributos de acordo com esse padrão para identificar o conteúdo de cada arquivo dentro do sistema de arquivos.

6.1.5 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 código e dados, arquivos também podem ser usados como:

Abstração de dispositivos de baixo nível: os sistemas UNIX costumam mapear as interfaces de acesso de vários dispositivos físicos em arquivos dentro do diretório /dev (de *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).

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`: posição 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 Seção 7.4.3) a ser usada no acesso ao disco `/dev/sda`.

Canais de comunicação: na família de protocolos de rede TCP/IP, a metáfora de arquivo é usada como interface para os canais de comunicação: uma conexão TCP é apresentada aos dois processos envolvidos 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 usam a metáfora do arquivo, como é o caso dos *pipes* em UNIX.

Em alguns sistemas operacionais experimentais, como o *Plan 9* [Pike et al., 1993, Pike et al., 1995] e o *Inferno* [Dorward et al., 1997], todos os recursos e entidades físicas e lógicas do sistema 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 nesta seção.

6.2 Uso de arquivos

Arquivos são usados por processos para ler e escrever dados de forma não-volátil. Para usar arquivos, um processo tem à sua disposição uma *interface de acesso*, que depende da linguagem utilizada e do sistema operacional subjacente. Essa interface normalmente é composta por uma representação lógica de cada arquivo usado pelo processo (uma *referência* ao arquivo) e por um conjunto de funções (ou métodos) para realizar operações sobre esses arquivos. Através dessa interface, os processos podem localizar arquivos no disco, ler e modificar seu conteúdo, entre outras operações.

Na sequência desta seção serão discutidos aspectos relativos ao uso de arquivos, como a abertura do arquivo, as formas de acesso aos seus dados, o controle de acesso e problemas associados ao compartilhamento de arquivos entre vários processos.

6.2.1 Abertura de um arquivo

Para poder ler ou escrever dados em um arquivo, cada aplicação precisa antes “abri-lo”. A **abertura de um arquivo** consiste basicamente em preparar as estruturas de

memória necessárias para acessar os dados do arquivo em questão. Assim, para abrir um arquivo, o núcleo do sistema operacional deve realizar as seguintes operações:

1. Localizar o arquivo no dispositivo físico, usando seu nome e caminho de acesso (vide Seção 6.3.2);
2. Verificar se a aplicação tem permissão para usar aquele arquivo da forma desejada (leitura e/ou escrita);
3. Criar uma estrutura na memória do núcleo para representar o arquivo aberto;
4. Inserir uma referência a essa estrutura na lista de arquivos abertos mantida pelo sistema, para fins de gerência;
5. Devolver à aplicação uma referência a essa estrutura, para ser usada nos acessos subsequentes ao arquivo recém-aberto.

Concluída a abertura do arquivo, o processo solicitante recebe do núcleo uma referência para o arquivo recém-aberto, que deve ser informada pelo processo 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 gerência criadas durante sua abertura. Normalmente, os arquivos abertos são automaticamente fechados quando do encerramento do processo, 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.

As referências a arquivos abertos usadas pelas aplicações dependem da linguagem de programação utilizada para construí-las. Por exemplo, em um programa escrito na linguagem C, cada arquivo aberto é representado por uma variável dinâmica do tipo **FILE***, que é denominada **um ponteiro de arquivo** (*file pointer*). Essa variável dinâmica é alocada no momento da abertura do arquivo e serve como uma referência ao mesmo nas operações de acesso subsequentes. Já em Java, as referências a arquivos abertos são objetos instanciados a partir da classe **File**. Na linguagem Python existem os **file objects**, criados a partir da chamada **open**.

Por outro lado, cada sistema operacional tem sua própria convenção para a representação de arquivos abertos. Por exemplo, 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. Por outro lado, em sistemas UNIX os arquivos abertos por um processo são representados por **descritores de arquivos** (*file descriptors*). Um descritor de arquivo aberto é um número inteiro não-negativo, usado como índice em uma tabela que relaciona os arquivos abertos por aquele processo, mantida pelo núcleo. 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.

6.2.2 Formas de acesso

Uma vez aberto um arquivo, 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 direto* (ou acesso aleatório).

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 ainda o são, 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 6.4 traz um exemplo de acesso sequencial em leitura a um arquivo, mostrando a evolução do ponteiro do arquivo durante uma sequência de leituras. A primeira leitura no arquivo traz a *string* “Qui_scribit_bis”, a segunda leitura 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.

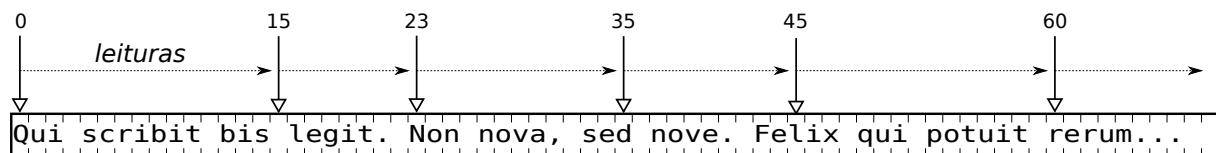


Figura 6.4: Leituras sequenciais em um arquivo de texto.

Por outro lado, no método de **acesso direto** (ou aleatório), pode-se indicar a posição no arquivo onde cada leitura ou escrita deve ocorrer, sem a necessidade de um ponteiro. 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 do arquivo caso necessário, o que permite então o acesso direto a qualquer registro do arquivo. Nos sistemas POSIX, o reposicionamento do ponteiro do arquivo é efetuado através das chamadas `lseek` e `fseek`.

Uma forma particular de acesso direto ao conteúdo de um arquivo é o **mapeamento em memória** do mesmo, que faz uso dos mecanismos de memória virtual (paginação). Nessa modalidade de acesso, um 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 ainda não acessada é lida, é gerada uma falta de página. Nesse momento, o mecanismo de paginação da memória virtual intercepta o acesso à memória, lê o conteúdo correspondente no arquivo e o deposita no vetor, de forma transparente à aplicação. 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 6.5 ilustra essa forma de acesso.

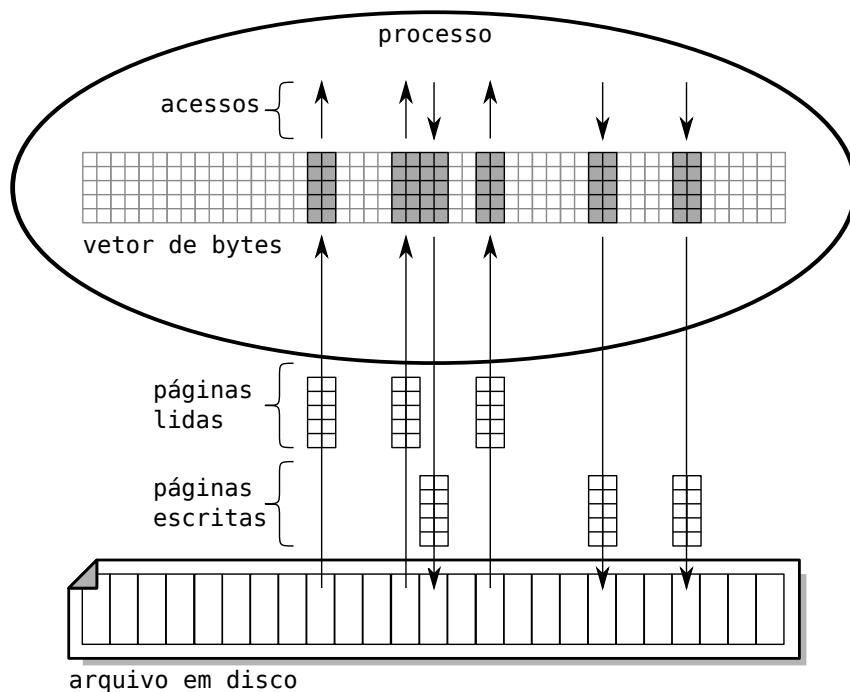


Figura 6.5: Arquivo mapeado em memória.

Finalmente, 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 um conjunto de pares *chave/valor*. Os dados do arquivo são armazenados e recuperados de acordo com suas chaves correspondentes, como em um banco de dados relacional. Como o próprio núcleo do 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.

6.2.3 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 d rwx --- --- 2 maziero prof      4096 2008-09-27 08:43 figuras
3 - rwx r-x --- 1 maziero prof      7248 2008-08-23 09:54 hello-unix
4 - rw- r-- r-- 1 maziero prof       54 2008-08-23 09:54 hello-unix.c
5 - rw- --- --- 1 maziero prof       59 2008-08-23 09:49 hello-windows.c
6 - rw- r-- r-- 1 maziero prof    195780 2008-09-26 22:08 main.pdf
7 - rw- --- --- 1 maziero prof     40494 2008-09-27 08:44 main.tex
```

Nessa listagem, o arquivo *hello-unix.c* (linha 4) 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* (linha 3) 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.

No mundo Windows, o sistema de arquivos NTFS implementa um controle de acesso bem mais flexível que o do UNIX, 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 8.

É importante destacar que o controle de acesso é normalmente 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 arquivo enquanto o mantiver aberto, mesmo que as permissões do arquivo sejam alteradas para impedir esse acesso. O controle contínuo de acesso aos arquivos é pouco frequentemente 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.

6.2.4 Compartilhamento de arquivos

Em um sistema multi-tarefas, é frequente ter arquivos acessados por mais de um processo, ou mesmo mais de um usuário, caso as permissões de acesso ao mesmo o permitam. Conforme estudado no Capítulo 4, 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 simultâneas tem de ser prevista e tratada de forma adequada.

Travas em arquivos

A solução mais simples e mais frequentemente utilizada para gerenciar o acesso compartilhado a arquivos é o uso de travas de exclusão mútua (*mutex locks*), estudadas no Capítulo 4. 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 necessidade de sincronização entre eles.

As travas oferecidas pelo sistema operacional podem ser **obrigatórias** (*mandatory locks*) ou **recomendadas** (*advisory locks*). As travas obrigatórias são impostas pelo núcleo de forma incontornável: se um processo obtiver a trava do arquivo para si, outros processos que solicitarem acesso ao arquivo serão suspensos até que a respectiva trava seja liberada. Por outro lado, as travas recomendadas não são impostas pelo núcleo do sistema operacional. Neste caso, um processo pode acessar um arquivo mesmo sem ter sua trava. Caso sejam usadas travas recomendadas, cabe ao programador implementar

os controles de trava necessários em suas aplicações, para impedir acessos conflitantes aos arquivos.

As travas sobre arquivos também podem ser **exclusivas** ou **compartilhadas**. Uma trava exclusiva, também chamada *trava de escrita*, garante acesso exclusivo ao arquivo: enquanto uma trava exclusiva estiver ativa, nenhum outro processo poderá obter uma trava sobre aquele arquivo. Já uma trava compartilhada (ou *trava de leitura*) impede outros processos de criar travas exclusivas sobre aquele arquivo, mas permite a existência de outras travas compartilhadas. Em conjunto, as travas exclusivas e compartilhadas implementam um modelo de sincronização *leitores/escritores* (descrito na Seção 4.9.2), no qual os leitores acessam o arquivo usando travas compartilhadas e os escritores o fazem usando travas exclusivas.

É importante observar que normalmente as travas de arquivos são atribuídas a processos, portanto um processo só pode ter um tipo de trava sobre um mesmo arquivo. Além disso, todas as suas travas são liberadas quando o processo fecha o arquivo ou encerra sua execução. No UNIX, a manipulação de travas em arquivos é feita através das chamadas de sistema flock e fcntl. Esse sistema oferece por default travas recomendadas exclusivas ou compartilhadas sobre arquivos ou trechos de arquivos. Sistemas Windows oferecem por default travas obrigatórias sobre arquivos, que podem ser exclusivas ou compartilhadas, ou travas recomendadas sobre trechos de arquivos.

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 lêem 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 arquivo é chamada de *semântica de compartilhamento*. Existem várias semânticas possíveis, mas as mais usuais são [Silberschatz et al., 2001]:

Semântica UNIX: toda modificação em um arquivo é imediatamente visível a todos os processos que mantêm aquele arquivo aberto; existe também a possibilidade de vários processos compartilharem o mesmo ponteiro de posicionamento do arquivo. Essa semântica é a mais comum em sistemas de arquivos locais, ou seja, para acesso a arquivos nos dispositivos locais;

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

Esta semântica é normalmente aplicada a sistemas de arquivos de rede, usados para acesso a arquivos em outros computadores;

Semântica imutável: de acordo com esta semântica, se um arquivo pode ser compartilhado por vários processos, ele é marcado como imutável, ou seja, seu conteúdo 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.

A Figura 6.6 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. 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 escrito por p_2 . Da mesma forma, o valor 71 escrito por p_2 não é percebido por p_3 , mas somente por p_4 .

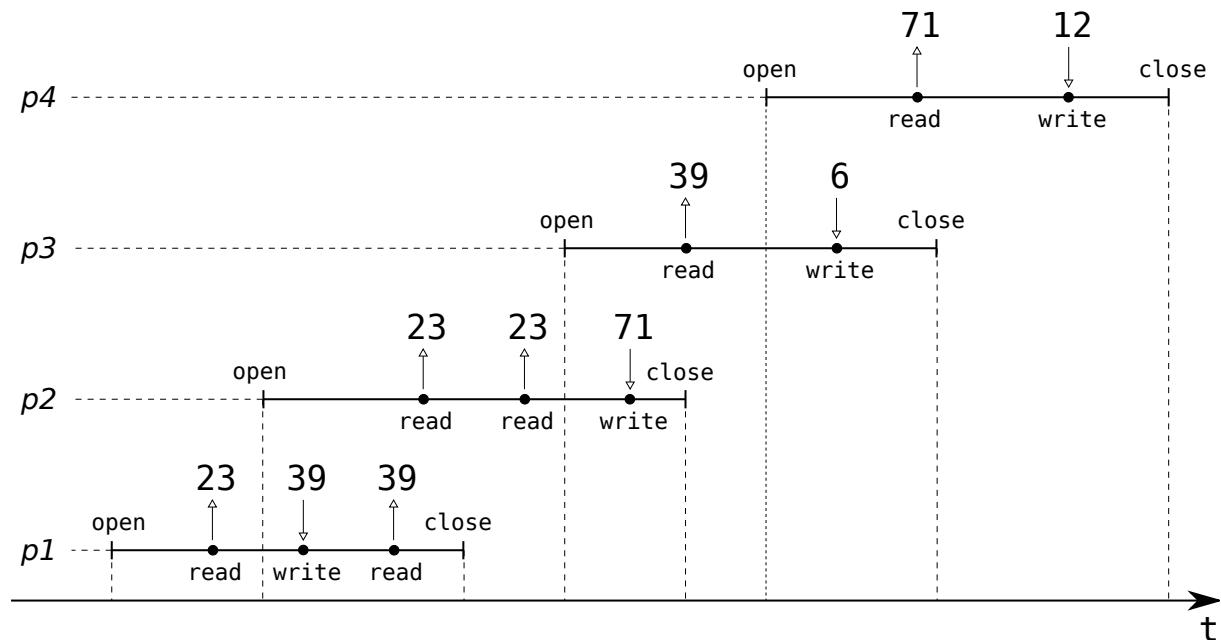


Figura 6.6: Compartilhamento de arquivo usando a semântica de sessão.

6.2.5 Exemplo de interface

Como visto na Seção 6.2.1, 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*, definida no arquivo de cabeçalho `stdio.h`). As funções mais usuais dessa biblioteca são apresentadas a seguir:

- 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 por `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 ;
- Repositionamento do ponteiro do arquivo:
 - `long int ftell (FILE *f)`: indica a posição corrente do ponteiro 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 do arquivo à sua posição inicial;
 - `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 reservado por outro processo;
 - `void funlockfile (FILE *f)`: libera o 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 23) e finalmente o fecha (linha 25).

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main (int argc, char *argv[], char* envp[])
5 {
6     FILE *arq ;
7     char c ;
8
9     arq = fopen ("infos.dat", "r") ; /* abertura do arquivo em leitura */
10
11    if (! arq) /* referencia de arquivo invalida */
12    {
13        perror ("Erro ao abrir arquivo") ;
14        exit (1) ;
15    }
16
17    while (1)
18    {
19        c = getc (arq) ; /* le um caractere do arquivo */
20        if (feof (arq)) /* chegou ao final do arquivo? */
21            break ;
22        putchar(c) ; /* imprime o caractere na tela */
23    }
24
25    fclose (arq) ; /* fecha o arquivo */
26    exit (0) ;
27 }
```

6.3 Organização de volumes

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 será discutida em detalhes no Capítulo 7; por enquanto, um disco rígido pode ser visto basicamente como um grande vetor de blocos de bytes. Esses blocos de dados, também denominados *setores*, têm tamanho fixo geralmente entre 512 e 4.096 bytes e são 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*).

Em um computador no padrão PC, o espaço de armazenamento de cada dispositivo é dividido em uma pequena área inicial de configuração e uma ou mais *partições*, que podem ser vistas como espaços independentes. A área de configuração é denominada MBR - *Master Boot Record*, e contém uma *tabela de partições* com informações sobre o particionamento do dispositivo. Além disso, contém também um pequeno código executável, usado no processo de inicialização do sistema operacional. No início de cada partição geralmente há um bloco reservado, utilizado para a descrição do conteúdo daquela partição e para armazenar o código de lançamento do sistema operacional, se for uma partição inicializável (*bootable partition*). Esse bloco reservado é denominado

bloco de inicialização ou VBR - *Volume Boot Record*. O restante dos blocos da partição está disponível para o armazenamento de arquivos. A Figura 6.7 ilustra a organização básica do espaço de armazenamento em um dispositivo de blocos típico: um disco rígido.

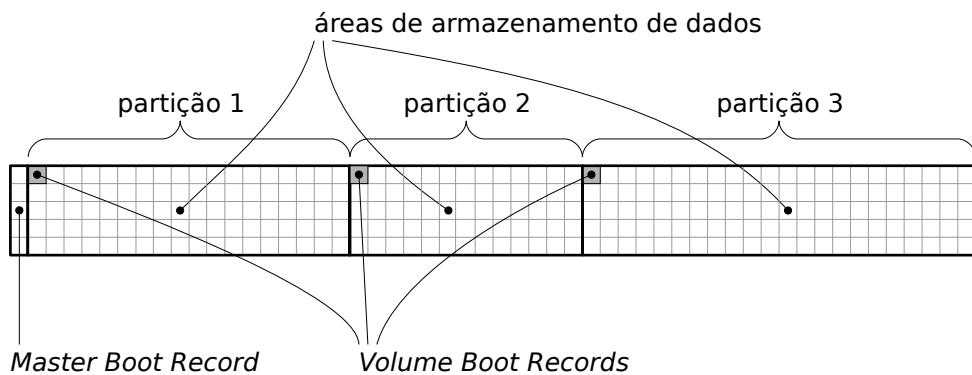


Figura 6.7: Organização em partições de um disco rígido típico.

Cada partição deve ser *formatada*, ou seja, estruturada para conter um sistema de arquivos, que pode conter arquivos, diretório, atalhos e outras entradas. Cada dispositivo ou partição devidamente preparado e formatado para receber um sistema de arquivos é designado como um *volume*.

6.3.1 Diretórios

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. Embora o sistema operacional possa tratar facilmente essa imensa quantidade de arquivos, essa tarefa não é tão simples para os usuários: identificar e localizar de forma inequívoca um arquivo específico em meio a milhões de outros arquivos pode ser impraticável.

Para permitir a organização de arquivos dentro de uma partição, são usados *diretórios*. Um diretório, também chamado de *pasta* (*folder*), representa um contêiner de informações, que pode conter arquivos ou mesmo outros diretórios. Da mesma forma que os arquivos, diretórios têm nome e atributos, que são usados na localização e acesso aos arquivos neles contidos.

Cada espaço de armazenamento possui ao menos um diretório principal, denominado *diretório raiz* (*root directory*). 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 de inicialização aponta para a posição do diretório raiz dentro do sistema de arquivos, permitindo que este tenha um número muito maior de entradas.

O uso de diretórios permite construir uma estrutura hierárquica (em árvore) de armazenamento dentro de um volume, sobre a qual os arquivos são distribuídos. A Figura 6.8 representa uma pequena parte da árvore de diretórios típica de um

sistema Linux, cuja estrutura é definida nas normas *Filesystem Hierarchy Standard* [Russell et al., 2004].

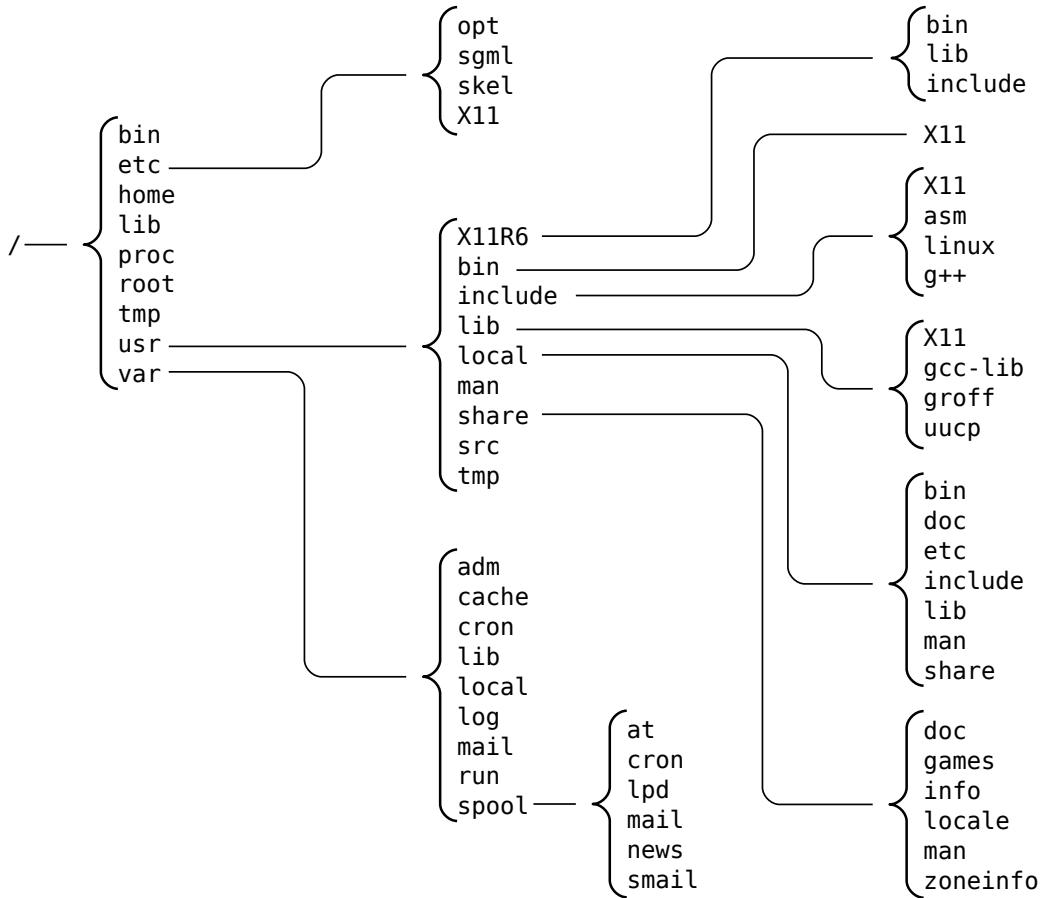


Figura 6.8: Estrutura de diretórios típica de um sistema Linux.

Os primeiros sistemas de arquivos implementavam apenas o diretório raiz, que continha todos os arquivos do volume. Posteriormente, ofereceram sub-diretórios, ou seja, um nível de diretórios abaixo do diretório raiz. Os sistemas atuais oferecem uma estrutura muito mais flexível, com um número de níveis de diretórios muito mais elevado, ou mesmo ilimitado (como no NTFS e no Ext3).

A implementação de diretórios é relativamente simples: um diretório é implementado como um arquivo estruturado, cujo conteúdo é uma relação de entradas. Os tipos de entradas normalmente considerados nessa relação são arquivos normais, diretórios, atalhos (vide Seção 6.3.3) e entradas associadas a arquivos especiais, como os discutidos na Seção 6.1.1. Cada entrada contém ao menos o nome do arquivo (ou do diretório), seu tipo e a localização física do mesmo no volume. Deve ficar claro que um diretório não contém fisicamente os arquivos e sub-diretórios, ele apenas os relaciona.

Duas entradas padronizadas são 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 próprio.

A Figura 6.9 apresenta uma possibilidade de implementação de parte da estrutura de diretórios apresentada na Figura 6.8. Os tipos das entradas em cada diretório são: “A” para arquivos normais e “D” para diretórios.

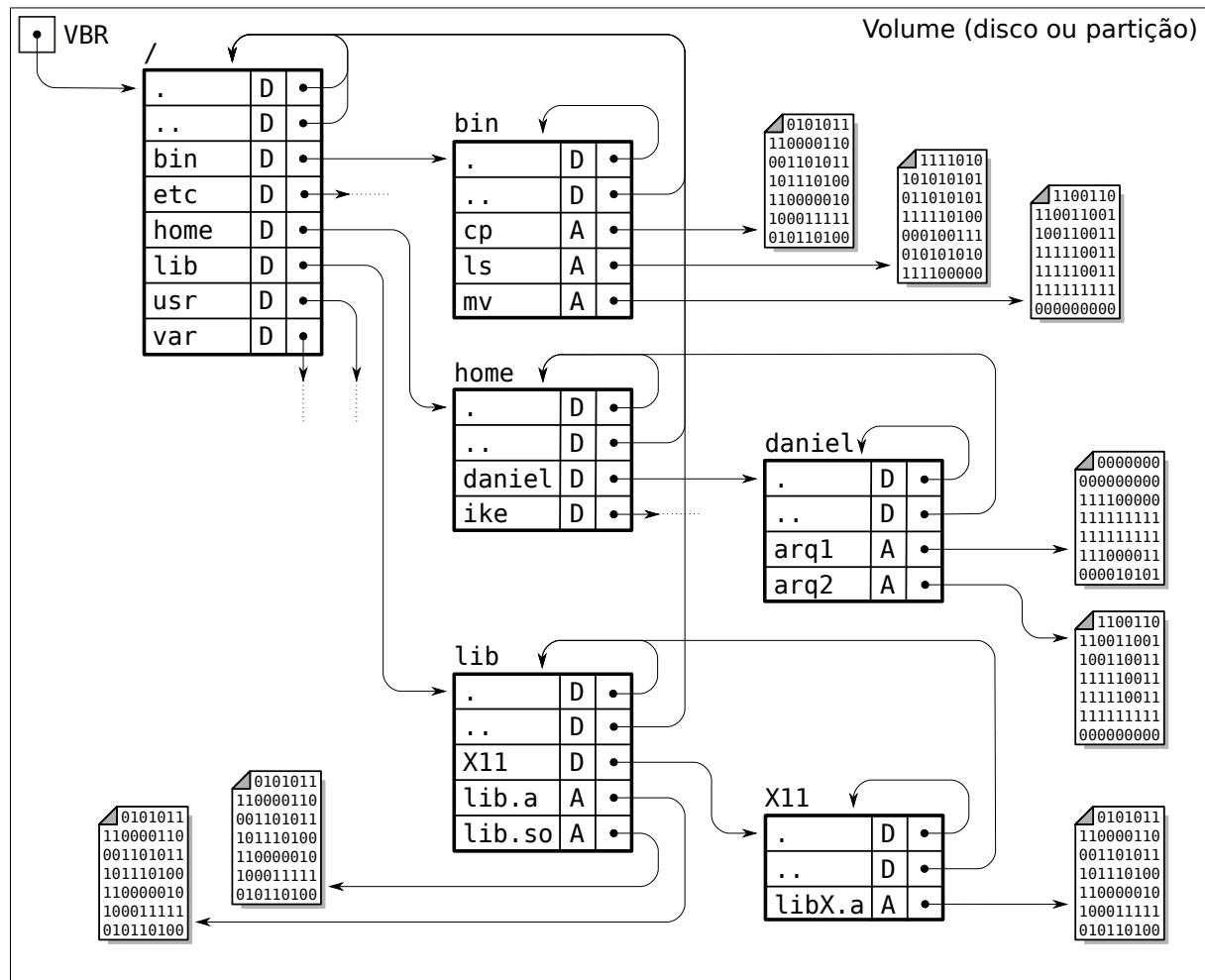


Figura 6.9: Implementação de uma estrutura de diretórios.

A relação de entradas em um diretório, também chamada de *índice do diretório*, pode ser implementada como uma lista linear, como no caso do MS-DOS e do Ext2 (Linux) ou como algum tipo de tabela *hash* ou árvore, o que é feito no NTFS e no Ext3, entre outros. A implementação em lista linear é mais simples, mas tem baixo desempenho. A implementação em tabela *hash* ou árvore provê um melhor desempenho quando é necessário percorrer a estrutura de diretórios em busca de arquivos, o que ocorre frequentemente.

6.3.2 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, separadas por um caractere específico. 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 seriam \Windows\system32\ole32.dll (no Windows) e /usr/bin/bash (em sistemas UNIX).

A maioria dos sistemas implementa o conceito de *diretório de trabalho* ou diretório corrente de um processo (*working directory*). Ao ser criado, cada novo processo recebe um diretório de trabalho, 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 pai, mas pode mudar de diretório através de chamadas de sistema (como `chdir` nos sistemas UNIX).

Existem basicamente três formas de se referenciar arquivos em um sistema de arquivos:

Referência direta: somente o nome do arquivo é informado; neste caso, considera-se que o arquivo está (ou será criado) no diretório de trabalho do processo. Exemplos:

- 1 prova1.doc
- 2 materiais.pdf
- 3 uma-bela-foto.jpg

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. 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 a seguir, os dois primeiros são caminhos canônicos, enquanto os dois últimos não o são:

- 1 \Windows\system32\drivers\etc\hosts.lm
- 2 /usr/local/share/fortunes/brasil.dat
- 3 \Documents and Settings\Carlos Maziero..\All Users\notas.xls
- 4 /home/maziero/bin/scripts/.../docs/proj1.pdf

Referência relativa: o caminho de acesso ao arquivo tem como início o diretório de trabalho do processo, e indica sub-diretórios ou diretórios anteriores, através de referências "..."; eis alguns exemplos:

- 1 imagens\satelite\brasil\geral.jpg
- 2 ..\users\maziero\documentos\prova-2.doc
- 3 public_html/static/fotografias/rennes.jpg
- 4 ../../share/icons/128x128/calculator.svg

Durante a abertura de um arquivo, o sistema operacional deve encontrar a localização do mesmo no dispositivo de armazenamento, a partir do nome e caminho informados pelo processo. Para isso, é necessário percorrer as estruturas definidas pelo caminho do arquivo até encontrar sua localização, em um procedimento denominado *localização de arquivo (file lookup)*. Por exemplo, para abrir o arquivo /usr/lib/X11/libX.a da Figura 6.9 seria necessário executar os seguintes passos³:

1. Acessar o disco para ler o VBR (*Volume Boot Record*) do volume;
2. Nos dados lidos, descobrir onde se encontra o diretório raiz (/) daquele sistema de arquivos;
3. Acessar o disco para ler o diretório raiz;
4. Nos dados lidos, descobrir onde se encontra o diretório usr;
5. Acessar o disco para ler o diretório usr;
6. Nos dados lidos, descobrir onde se encontra o diretório lib;
7. Acessar o disco para ler o diretório lib;
8. Nos dados lidos, descobrir onde se encontra o diretório X11;
9. Acessar o disco para ler o diretório X11;
10. Nos dados lidos, descobrir onde se encontra o arquivo libX11.a;
11. Acessar o disco para ler o bloco de controle do arquivo libX11.a, que contém seus atributos;
12. Criar as estruturas em memória que representam o arquivo aberto;
13. Retornar uma referência ao arquivo para o processo solicitante.

Pode-se perceber que a localização de arquivo é um procedimento trabalhoso. Neste exemplo, foram necessárias 5 leituras no disco (passos 1, 3, 5, 7 e 9) apenas para localizar a posição do bloco de controle do arquivo desejado no disco. Assim, o tempo necessário para localizar um arquivo pode ser muito elevado, pois discos rígidos são dispositivos lentos. Para evitar esse custo e melhorar o desempenho do mecanismo de localização de arquivos, é mantido em memória um cache de entradas de diretório localizadas recentemente, gerenciado de acordo com uma política LRU (*Least Recently Used*). Cada entrada desse cache contém um nome de arquivo ou diretório e sua localização no dispositivo físico. Esse cache geralmente é organizado na forma de uma tabela *hash*, o que permite localizar rapidamente os arquivos ou diretórios recentemente utilizados.

³Para simplificar, foram omitidas as verificações de existência de entradas, de permissões de acesso e os tratamentos de erro.

6.3.3 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 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* (*links*).

Existem basicamente duas abordagens para a construção de atalhos:

Atalhos simbólicos (*soft links*): cada “cópia” do arquivo original é na verdade 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 ao arquivo original é indicado de forma simbólica, este pode estar localizado em outro dispositivo físico (outro disco ou uma unidade de rede). O arquivo original e seus atalhos simbólicos são totalmente independentes: caso o arquivo original seja movido, renomeado ou removido, os atalhos simbólicos apontarão para um arquivo inexistente; neste caso, diz-se que aqueles atalhos estão “quebrados” (*broken links*).

Atalhos físicos (*hard links*): 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 6.10 traz exemplos de implementação de atalhos simbólicos e físicos a arquivos em um sistema de arquivos UNIX. As entradas de diretórios indicadas como “L” correspondem a atalhos simbólicos (de *links*). Nessa figura, pode-se constatar que as entradas `/bin/ls` e `/usr/bin/dir` são atalhos físicos para o mesmo conteúdo no disco, enquanto a entrada `/bin/shell` é um atalho simbólico para o arquivo `/usr/bin/sh` e `/lib` é um atalho simbólico para o diretório `/usr/lib`.

Sistemas UNIX suportam atalhos físicos e simbólicos, com algumas limitações: atalhos físicos geralmente só podem ser feitos para arquivos dentro do mesmo sistema de arquivos (mesmo volume) e não são permitidos atalhos físicos para diretórios⁴. 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.

⁴Atalhos físicos para diretórios geralmente são proibidos porque permitiriam diretórios recursivos, tornando muito complexa a implementação de rotinas de verificação e gerência do sistema de arquivos.

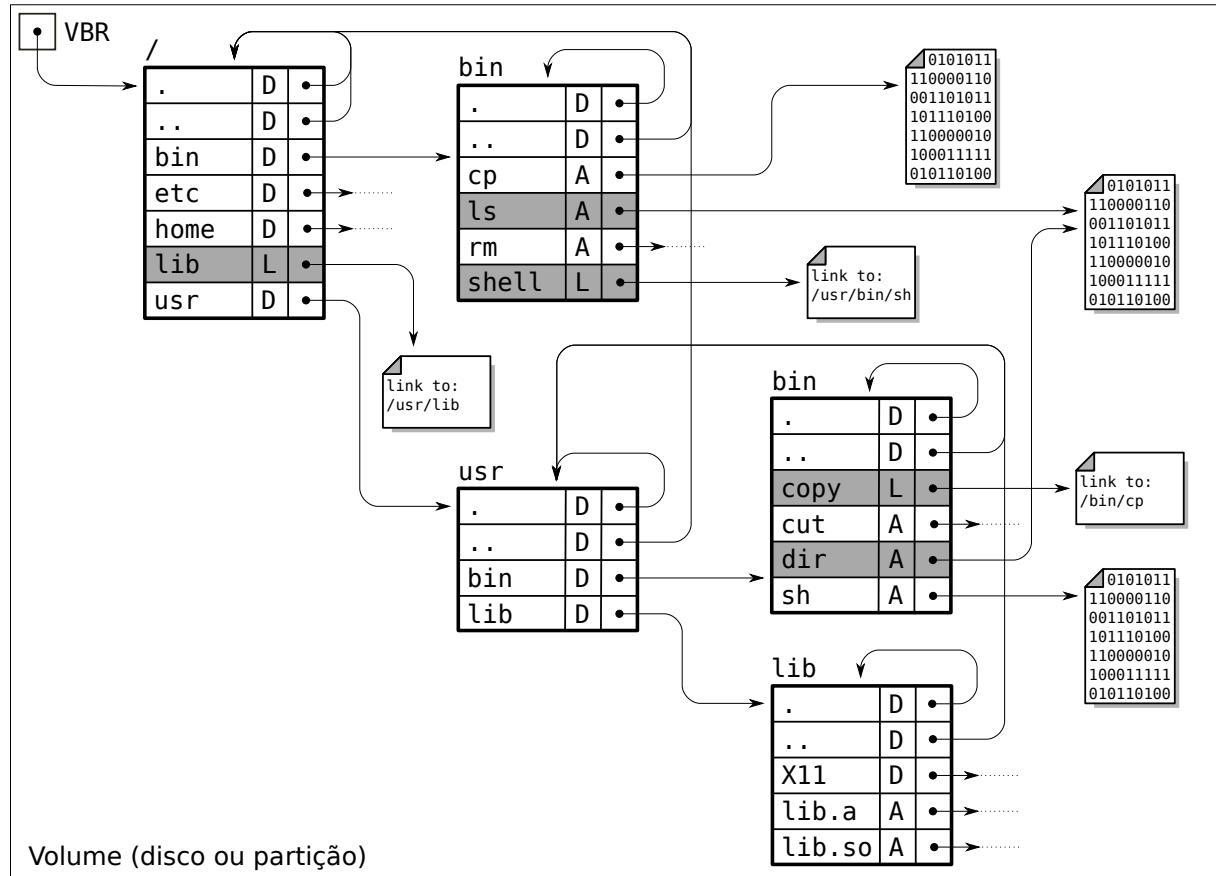


Figura 6.10: Atalhos simbólicos e físicos a arquivos em UNIX.

6.3.4 Montagem de volumes

Para que o sistema operacional possa acessar o sistema de arquivos presente em um determinado volume, ele deve ler os dados presentes em seu bloco de inicialização, que descrevem o tipo de sistema de arquivos presente, e criar as estruturas em memória que representam esse volume dentro do núcleo. 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 seus 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.

A montagem é um procedimento frequente no caso de mídias móveis, como CD-ROMs, DVD-ROMs e pendrives USB. Neste caso, a desmontagem do volume inclui também ejectar 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 sub-diretórios na árvore do volume principal, através do comando `mount`. A Figura 6.11 apresenta um exemplo de montagem de volumes em plataformas UNIX. Nessa figura, o disco rígido 1 contém o sistema operacional e foi montado como raiz da árvore de diretórios durante a inicialização do sistema. O disco rígido 2 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 uma mídia removível (CD-ROM), com sua árvore de diretórios própria.

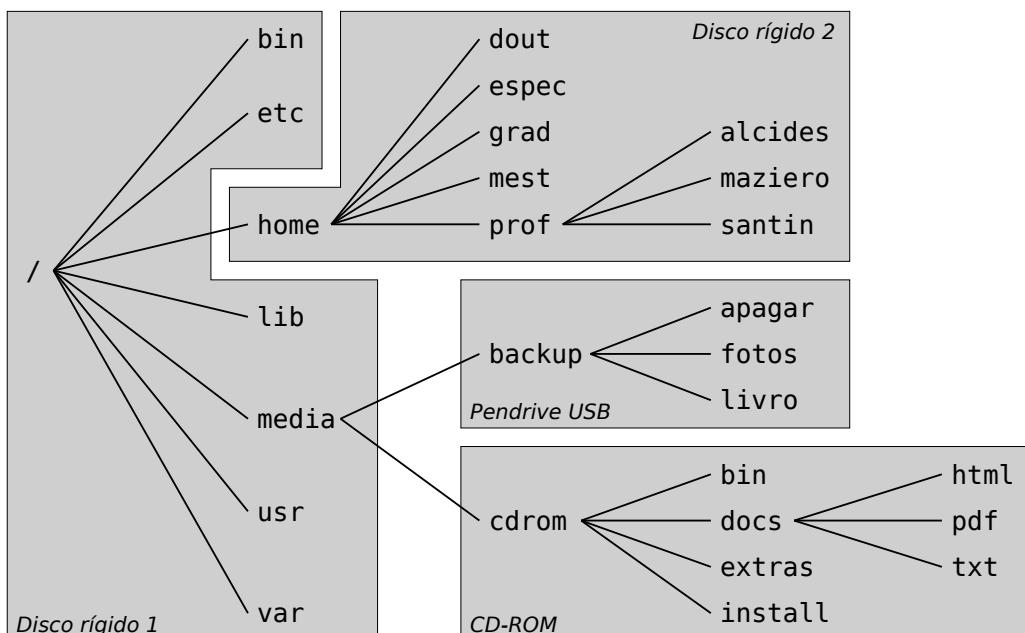


Figura 6.11: Montagem de volumes em UNIX.

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 sub-diretórios, da mesma forma que o UNIX.

6.4 Sistemas de arquivos

Vários problemas importantes devem ser resolvidos na construção de um sistema de arquivos, que vão do acesso de baixo nível aos dispositivos físicos de armazenamento à implementação da interface de acesso a arquivos para os programadores. Na implementação de um sistema de arquivos, considera-se que cada arquivo possui **dados** e **meta-dados**. Os dados de um arquivo são o seu conteúdo em si (uma música, uma fotografia, um documento ou uma planilha); por outro lado, os meta-dados 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.

Nesta seção serão discutidos os principais elementos que compõem a gerência de arquivos em um sistema operacional típico.

6.4.1 Arquitetura geral

Os principais elementos que constituem a gerência de arquivos estão organizados em camadas, conforme apresentado na Figura 6.12. No nível mais baixo dessa arquitetura estão os **dispositivos de armazenamento**, como discos rígidos ou bancos de memória *flash*, responsáveis pelo armazenamento dos dados e meta-dados dos arquivos. Esses dispositivos são acessados através de **controladores**, que são circuitos eletrônicos dedicados ao controle e interface dos dispositivos. A interface entre controladores e dispositivos de armazenamento segue padrões como SATA, ATAPI, SCSI, USB e outros.

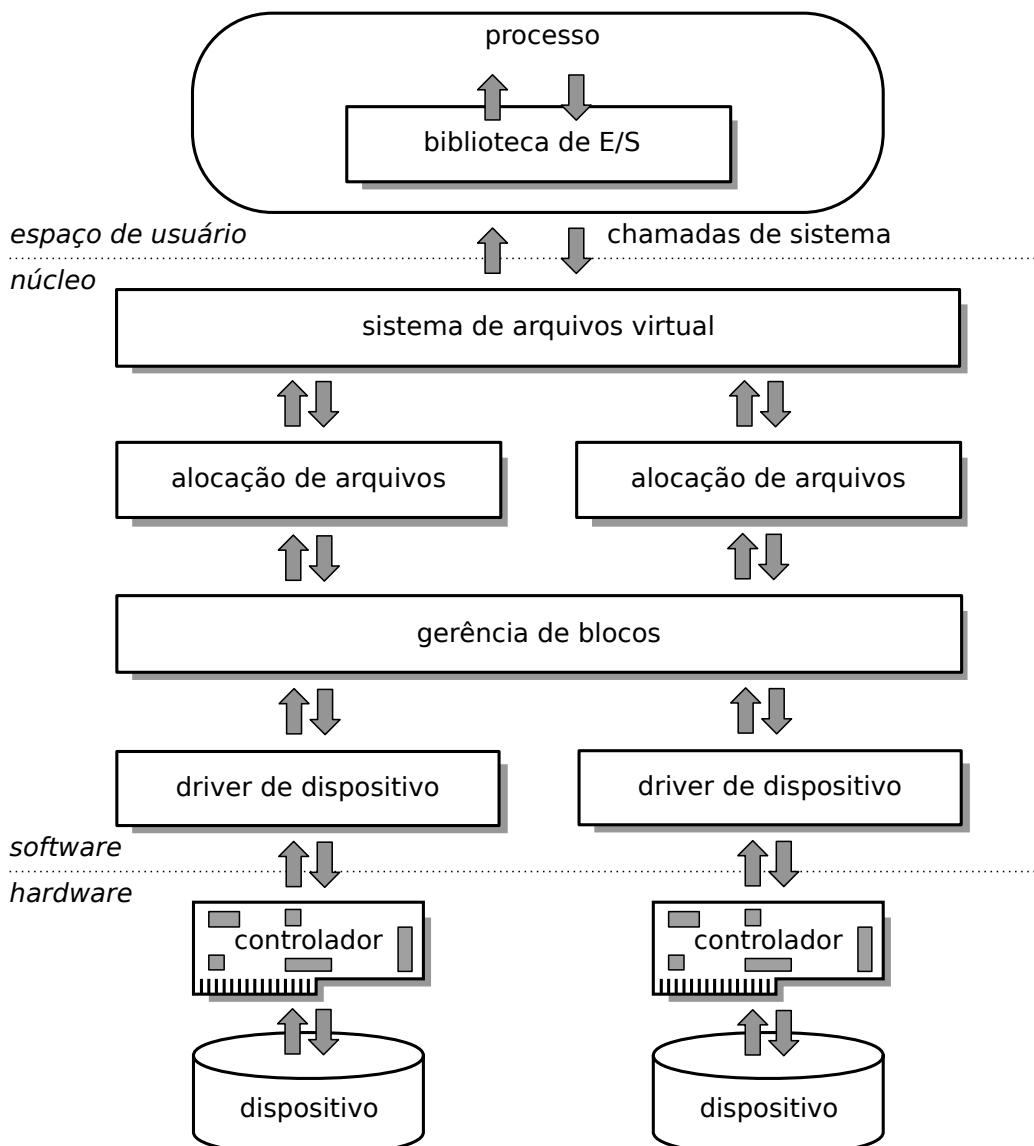


Figura 6.12: Camadas da implementação da gerência de arquivos.

Os controladores de dispositivos são configurados e acessados pelo núcleo do sistema operacional através de **drivers de dispositivos**, que são componentes de software capazes de interagir com os controladores. Os *drivers* usam portas de entrada/saída, interrupções e canais de acesso direto à memória (DMA) para interagir com os controladores e realizar as operações de controle e de entrada/saída de dados. Como cada controlador define sua própria interface, também possui um *driver* específico. Os *drivers* ocultam essas interfaces e fornecem às camadas superiores do núcleo uma interface padronizada para acesso aos dispositivos de armazenamento. Desta forma, os detalhes tecnológicos e particularidades de cada dispositivo são isolados, tornando o restante do sistema operacional independente da tecnologia subjacente.

Acima dos *drivers* está a camada de **gerência de blocos**, que gerencia o fluxo de blocos de dados entre a memória e os dispositivos de armazenamento. É importante lembrar que os discos são dispositivos orientados a blocos, ou seja, 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, oferecer às camadas superiores a abstração de cada dispositivo físico como sendo um imenso vetor de blocos lógicos, independente de sua configuração real, e também efetuar o *caching/buffering* de blocos (Seção 7.4.4).

A seguir está a camada de **alocação de arquivos**, que tem como função principal alocar os arquivos sobre os blocos lógicos oferecidos pela 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 6.4.3.

Acima da alocação de arquivos está o **sistema de arquivos virtual** (VFS - *Virtual File System*), que provê uma interface de acesso a arquivos independente dos dispositivos físicos e das estratégias de alocação de arquivos empregadas pelas camadas inferiores. O sistema de arquivos virtual normalmente gerencia as permissões associadas aos arquivos e as travas de acesso compartilhado, além de construir as abstrações de diretórios e atalhos. Outra responsabilidade importante desta camada é manter informações sobre 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. A interface de acesso ao sistema de arquivos virtual é oferecida aos processos através de um conjunto de **chamadas de sistema**.

Finalmente, as **bibliotecas de entrada/saída** usam as chamadas de sistema oferecidas pelo sistema operacional para construir funções padronizadas de acesso a arquivos para cada linguagem de programação, como aquelas apresentadas na Seção 6.2.5 para a linguagem C ANSI.

6.4.2 Blocos físicos e lógicos

Um dos aspectos mais importantes dos sistemas de arquivos é a forma como o conteúdo dos arquivos é disposto dentro do disco rígido ou outro dispositivo de armazenamento secundário. Conforme visto na Seção 6.3, um disco rígido pode ser visto como um conjunto de blocos de tamanho fixo (geralmente de 512 bytes). Os blocos

do disco rígido são normalmente denominados *blocos físicos*. Como esses blocos são pequenos, o número de blocos físicos em um disco rígido recente pode ser imenso: um disco rígido de 250 GBytes contém mais de 500 milhões de blocos físicos! Para simplificar a gerência dessa quantidade de blocos físicos e melhorar o desempenho das operações de leitura/escrita, os sistemas operacionais costumam trabalhar com *blocos lógicos* ou *clusters*, que são grupos de 2ⁿ blocos físicos consecutivos. Blocos lógicos com 4K, 8K, 16K e 32K bytes são frequentemente usados. 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 de uma partição é 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 muito pequenos implicam em ter mais blocos a gerenciar e menos bytes transferidos em cada operação de leitura/escrita, o que tem impacto negativo sobre o desempenho do sistema. Por outro lado, blocos lógicos muito grandes podem levar à *fragmentação interna*: um arquivo com 200 bytes armazenado em um sistema de arquivos com blocos lógicos de 32.768 bytes (32K) ocupará um bloco lógico, do qual 32.568 bytes serão desperdiçados, pois ficarão alocados ao arquivo sem serem usados. A fragmentação interna diminui o espaço útil do disco rígido, por isso deve ser evitada. Uma forma de evitá-la é escolher um tamanho de bloco lógico adequado ao tamanho médio dos arquivos a armazenar no disco, ao formatá-lo. Além disso, 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].

6.4.3 Alocação física de arquivos

Um dispositivo de armazenamento é visto pelas camadas superiores como um grande vetor de blocos lógicos de tamanho fixo. O problema da alocação de arquivos consiste em dispor (alocar) o conteúdo e os meta-dados dos arquivos dentro desses blocos lógicos. Como os blocos lógicos são pequenos, cada arquivo poderá precisar de muitos blocos lógicos para ser armazenado no disco (Figura 6.13). Os dados e meta-dados de um arquivo devem estar dispostos nesses blocos de forma a permitir um acesso rápido e confiável. Como um arquivo pode ocupar milhares ou mesmo milhões de blocos, a forma de alocação dos arquivos nos blocos do disco tem um impacto importante sobre o desempenho e a robustez do sistema de arquivos.

A alocação de um arquivo no disco tem como ponto de partida a definição de um **bloco de controle de arquivo** (FCB - *File Control Block*), que nada mais é que uma estrutura contendo os meta-dados do arquivo e a localização de seu conteúdo no disco. Em alguns sistemas de arquivos mais simples, como o sistema FAT (*File Allocation Table*) usado em plataformas MS-DOS, o FCB é bastante pequeno e cabe na entrada correspondente ao arquivo, na tabela de diretório onde ele se encontra definido. Em sistemas de arquivos mais complexos, os blocos de controle de arquivos são definidos

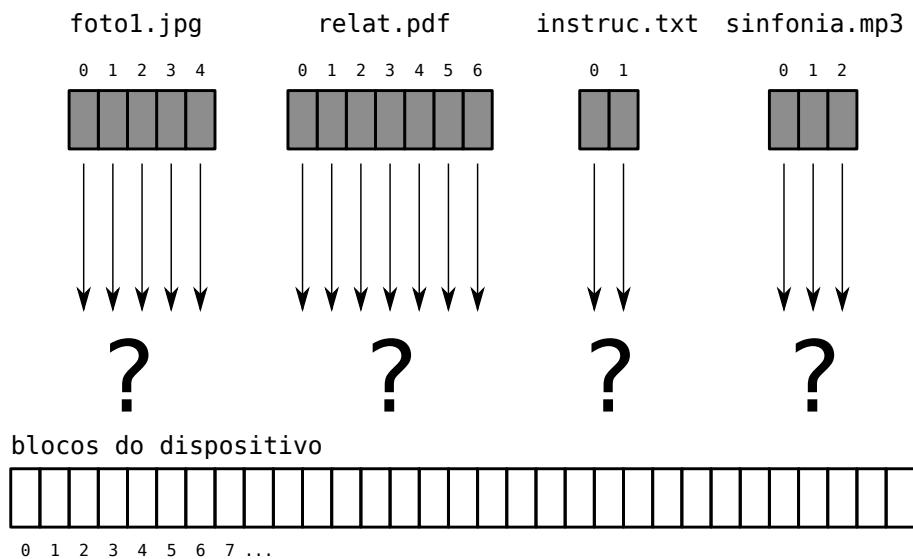


Figura 6.13: O problema da alocação de arquivos.

em estruturas separadas, como a *Master File Table* do sistema NTFS e os *i-nodes* dos sistemas UNIX.

Há três estratégias usuais de alocação de arquivos nos blocos lógicos do disco, que serão apresentadas a seguir: as alocações **contígua**, **encadeada** e **indexada**. Como diretórios são usualmente implementados na forma de arquivos, as estratégias de alocação discutidas aqui são válidas também para a alocação de diretórios. 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 diretos; 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 e diretórios.

Alocação contígua

Na alocação contígua, os dados do arquivo são dispostos de forma ordenada 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 6.14 apresenta um exemplo dessa estratégia de alocação (para simplificar o exemplo, considera-se que a tabela de diretórios contém os meta-dados 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 direto a posições específicas do arquivo 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 1. De acordo com esse algoritmo, o byte de número 14.372 do arquivo *relat.pdf* da Figura 6.14 estará na posição 2.084 do bloco 16 do disco rígido.

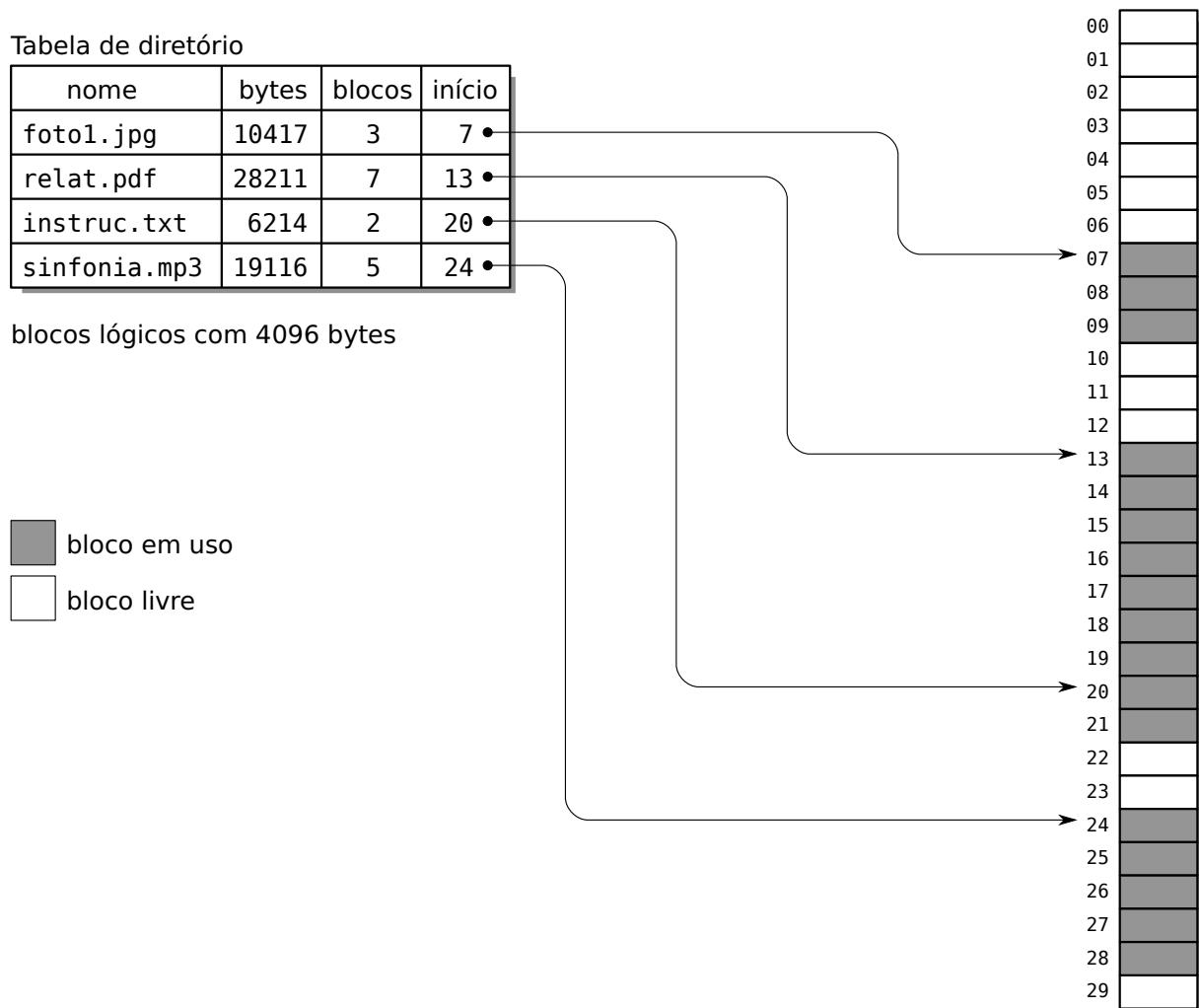


Figura 6.14: Estratégia de alocação contígua.

Esta estratégia apresenta uma boa robustez a falhas de disco: caso um bloco do disco apresente defeito e não permita a leitura de seus dados, apenas o conteúdo daquele bloco é perdido: o conteúdo do arquivo nos blocos anteriores e posteriores ao bloco defeituoso ainda poderão ser acessados sem dificuldades. Por outro lado, o ponto fraco desta estratégia é sua baixa flexibilidade, pois o tamanho final de cada arquivo precisa ser conhecido no momento de sua criação. Além disso, esta estratégia está sujeita à fragmentação externa, de forma similar à técnica de alocação contígua estudada nos mecanismos de alocação de memória (vide Seção 5.3.2): à medida em que arquivos são criados e destruídos, as áreas livres do disco vão sendo fracionadas em pequenas áreas isoladas (os fragmentos) que diminuem a capacidade de alocação de arquivos maiores. Por exemplo, na situação da Figura 6.14 há 13 blocos livres no disco, mas somente podem ser criados arquivos com até 7 blocos de tamanho. As técnicas de alocação *first/best/worst-fit* utilizadas em gerência de memória também podem ser aplicadas para atenuar este problema. Contudo, a desfragmentação de um disco é problemática, pois

Algoritmo 1 Localizar a posição do i -ésimo byte do arquivo no disco

i : número do byte a localizar
 B : tamanho dos blocos lógicos, em bytes
 b_0 : número do bloco do disco onde o arquivo inicia
 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)

$$\begin{aligned} b_i &= b_0 + i \div B \\ o_i &= i \bmod B \\ \text{return } &(b_i, o_i) \end{aligned}$$

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 os 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 diretos aos dados. Um exemplo dessa situação são sistemas dedicados para reprodução de dados multimídia, como áudio e vídeo.

Alocação encadeada

Esta forma de alocação foi proposta para contornar a pouca flexibilidade da alocação contígua e eliminar a fragmentação externa. Nela, cada bloco do arquivo no disco contém dados do arquivo e também um ponteiro para o próximo bloco, ou seja, um campo indicando o número do próximo bloco do arquivo no disco. 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 são utilizáveis sem restrições, e permite que arquivos sejam criados sem a necessidade de definir seu tamanho final. A Figura 6.15 ilustra um exemplo dessa abordagem.

Nesta abordagem, o acesso sequencial aos dados do arquivo é simples e rápido, pois cada bloco contém o ponteiro do próximo bloco do arquivo. 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 direto a posições específicas do arquivo fica muito prejudicado com esta abordagem: caso se deseje acessar um bloco no meio do arquivo, todos os blocos anteriores terão de ser lidos em sequência, para poder seguir os ponteiros que levam ao bloco desejado. O algoritmo 2 mostra claramente esse problema, indicado através do laço *while*. Essa dependência dos blocos anteriores 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

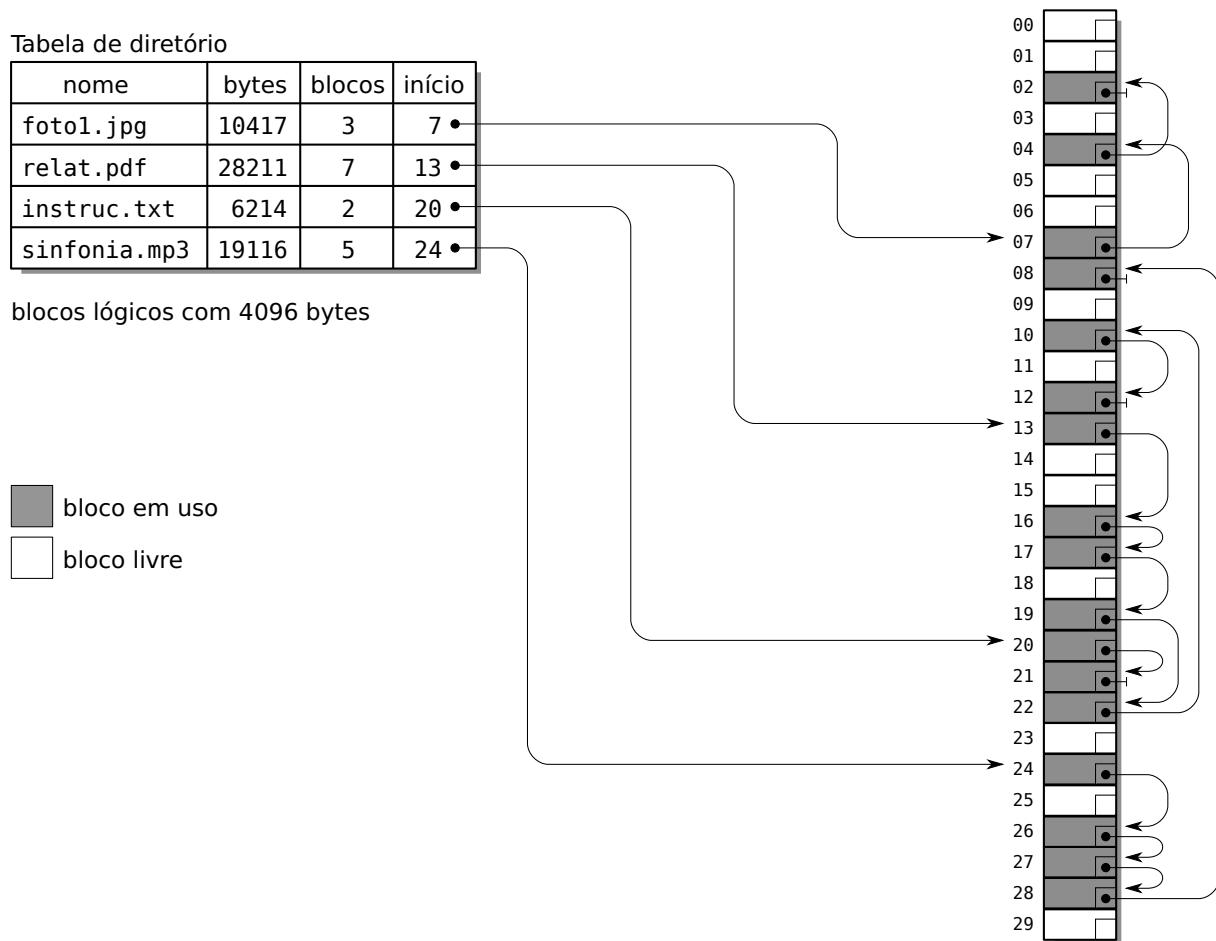


Figura 6.15: Estratégia de alocação encadeada.

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 usado por qualquer arquivo, eliminando a fragmentação externa.

Os principais problemas da alocação encadeada são o baixo desempenho nos acessos diretos 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 *pen-drives*, reprodutores MP3 e câmeras fotográficas digitais.

Na abordagem da 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 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

Algoritmo 2 Localizar a posição do i -ésimo byte do arquivo no disco

i : número do byte a localizar
 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
 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*)

```
// define bloco inicial do percurso
 $b_{aux} = b_0$ 
// calcula número de blocos a percorrer
 $b = i \div (B - P)$ 
while  $b > 0$  do
     $block = read\_block(b_{aux})$ 
     $b_{aux} =$  ponteiro extraído de  $block$ 
     $b = b - 1$ 
end while
 $b_i = b_{aux}$ 
 $o_i = i \bmod (B - P)$ 
return  $(b_i, o_i)$ 
```

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 6.16 apresenta o conteúdo da tabela de alocação de arquivos para o exemplo apresentado anteriormente na Figura 6.15.

Alocação indexada

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 6.17 apresenta um exemplo de alocação indexada.

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 entradas, só poderão ser armazenados arquivos com até 256 Kbytes. 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.

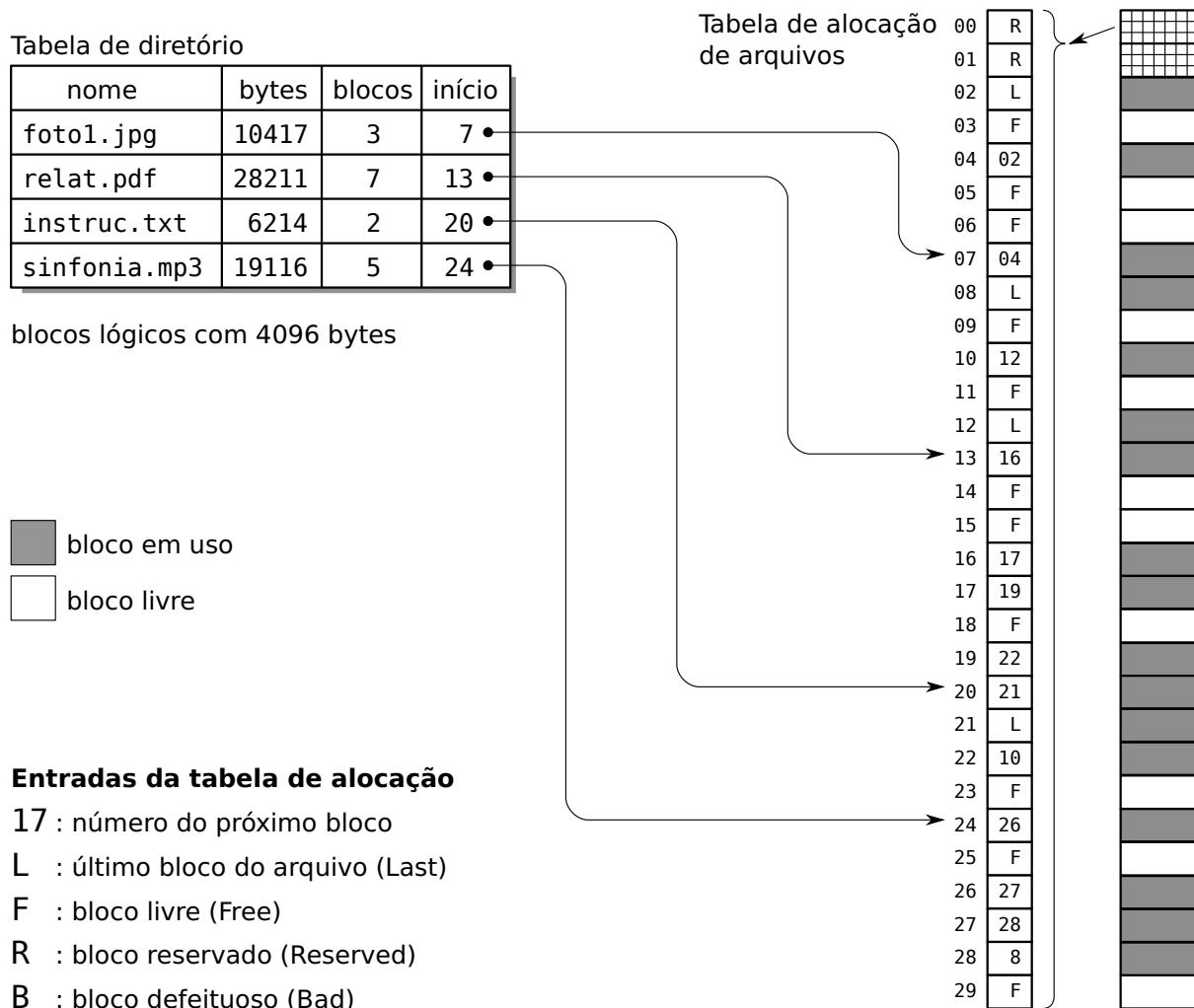


Figura 6.16: Uma tabela de alocação de arquivos.

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 4K bytes e ponteiros de 32 bits (4 bytes), cada bloco lógico pode conter 1024 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 6.18) 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. Considerando blocos lógicos de 4K bytes e ponteiros de 4 bytes, cada bloco de ponteiros contém 1024 ponteiros. Dessa forma, o cálculo do tamanho máximo de um arquivo nesse sistema é simples:

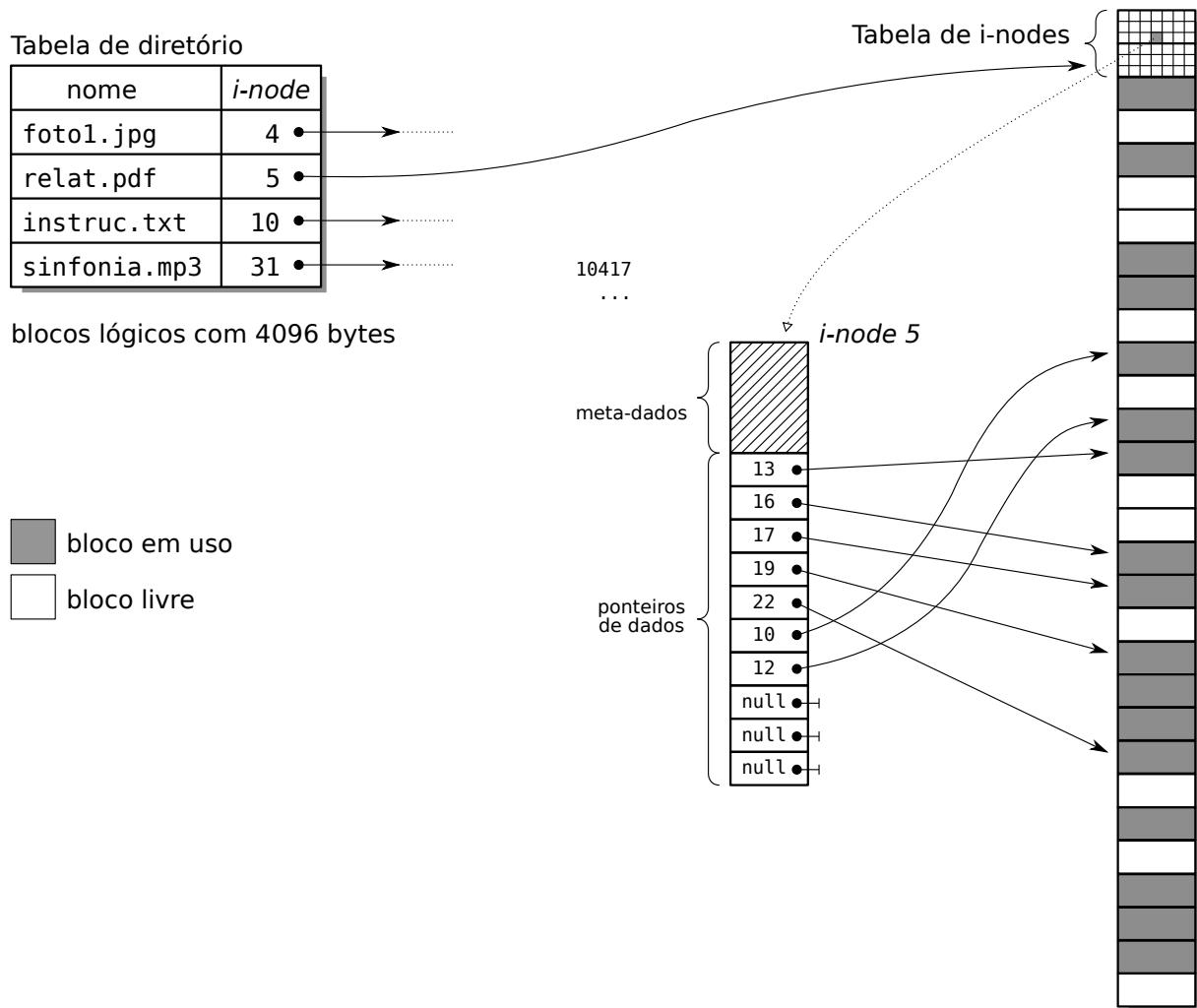


Figura 6.17: Estratégia de alocação indexada simples.

$$\begin{aligned}
 max &= 4096 \times 12 && (\text{ponteiros diretos}) \\
 &+ 4096 \times 1024 && (\text{ponteiro indireto}) \\
 &+ 4096 \times 1024 \times 1024 && (\text{ponteiro indireto duplo}) \\
 &+ 4096 \times 1024 \times 1024 \times 1024 && (\text{ponteiro indireto triplo}) \\
 &= 4.402.345.721.856 \text{ bytes} \\
 max &\approx 4T \text{ bytes}
 \end{aligned}$$

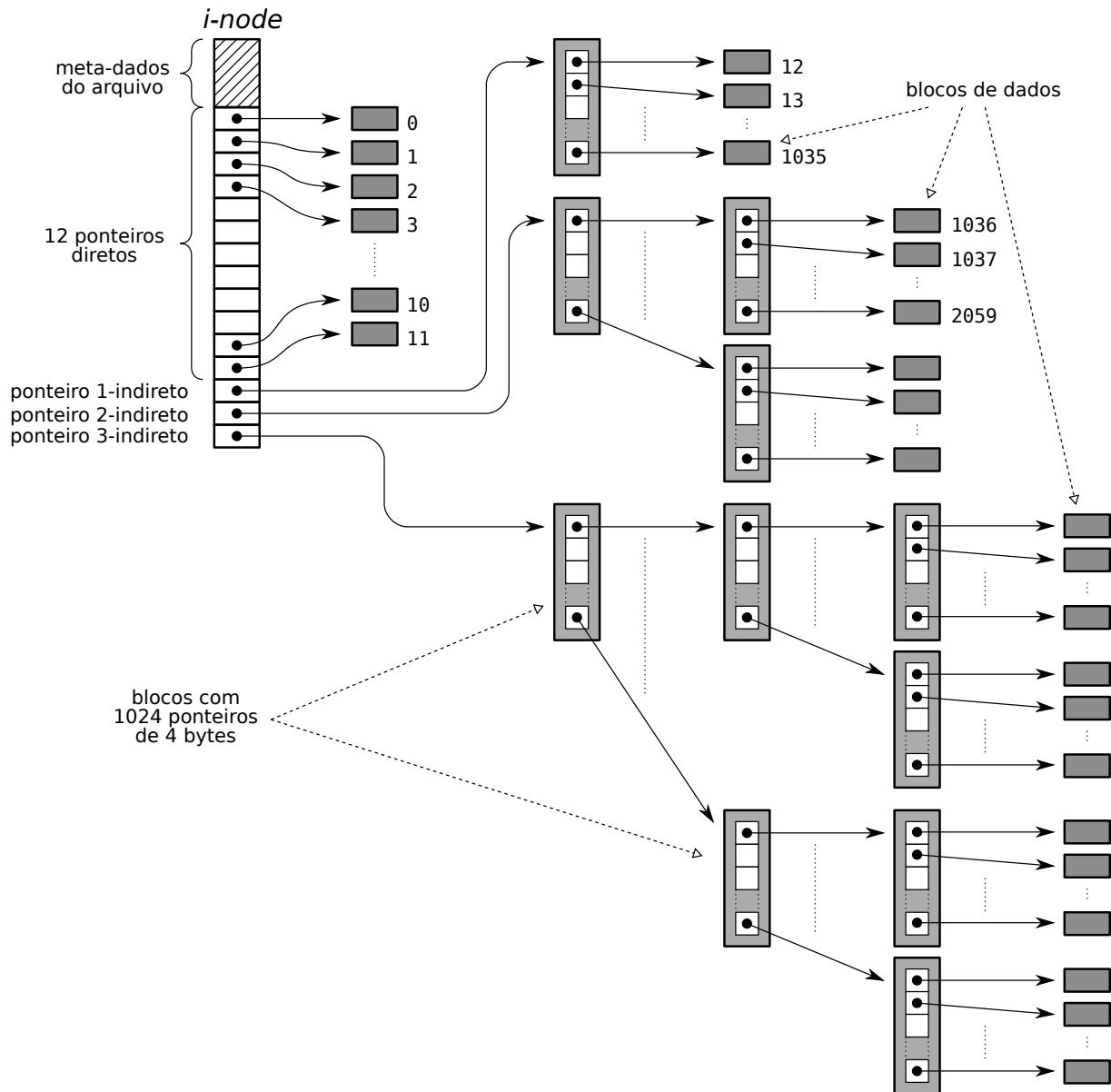


Figura 6.18: Estratégia de alocação indexada multi-nível.

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 a localização dos blocos com exatidão. A localização do bloco lógico de disco correspondente ao i -ésimo bloco lógico de um arquivo segue o algoritmo 3.

Em relação ao desempenho, pode-se afirmar que esta estratégia é bastante rápida, tanto para acessos sequenciais quanto para acessos diretos a blocos, devido aos índices de ponteiros dos blocos presentes nos *i-nodes*. Contudo, no caso de blocos no final de arquivos muito grandes, podem ser necessários três ou quatro acessos a disco adicionais para localizar o bloco desejado, devido aos ponteiros indiretos. Defeitos em blocos de dados não afetam os demais blocos de dados, o que torna esta estratégia robusta. Todavia,

Algoritmo 3 Localizar a posição do i -ésimo byte do arquivo no disco

1. B : tamanho dos blocos lógicos, em bytes
2. b_i : número do bloco do disco onde se encontra o byte i
3. o_i : posição do byte i dentro do bloco b_i (offset)
4. $ptr[0...14]$: vetor de ponteiros do i -node
5. $block[0...1023]$: bloco de ponteiros para outros blocos
- 6.
7. $o_i = i \bmod B$
8. $b_{aux} = i \div B$
9. **if** $b_{aux} < 12$ **then** // ponteiros diretos
10. // o endereço do bloco b_i é o próprio valor do ponteiro
11. $b_i = ptr[b_{aux}]$
12. **else**
13. $b_{aux} = b_{aux} - 12$
14. **if** $b_{aux} < 1024$ **then** // ponteiro indireto simples
15. // ler bloco de ponteiros de nível 1
16. $block_1 = read_block(ptr[12])$
17. // encontrar o endereço do bloco b_i
18. $b_i = block_1[b_{aux}]$
19. **else**
20. $b_{aux} = b_{aux} - 1024$
21. **if** $b_{aux} < 1024 \times 1024$ **then** // ponteiro indireto duplo
22. // ler bloco de ponteiros de nível 1
23. $block_1 = read_block(ptr[13])$
24. // ler bloco de ponteiros de nível 2
25. $block_2 = read_block(block_1[b_{aux} \div 1024])$
26. // encontrar o endereço do bloco b_i
27. $b_i = block_2[b_{aux} \bmod 1024]$
28. **else** // ponteiro indireto triplo
29. $b_{aux} = b_{aux} - (1024 \times 1024)$
30. // ler bloco de ponteiros de nível 1
31. $block_1 = read_block(ptr[14])$
32. // ler bloco de ponteiros de nível 2
33. $block_2 = read_block(block_1[b_{aux} \div (1024 \times 1024)])$
34. // ler bloco de ponteiros de nível 3
35. $block_3 = read_block(block_2[(b_{aux} \div 1024) \bmod 1024])$
36. // encontrar o endereço do bloco b_i
37. $b_i = block_3[b_{aux} \bmod 1024]$
38. **end if**
39. **end if**
40. **end if**
41. **return** (b_i, o_i)

defeitos nos meta-dados (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 meta-dados 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 do disco para armazenar dados. Todavia, o tamanho máximo dos arquivos criados é limitado, bem como o número máximo de arquivos na partição.

Uma característica interessante da alocação indexada é a possibilidade de criar *arquivos esparsos*. Um arquivo esparsos contém áreas mapeadas no disco (contendo dados) e áreas não-mapeadas (sem dados). Somente as áreas mapeadas estão fisicamente alocadas no disco rígido, pois os ponteiros correspondentes a essas áreas no *i-node* apontam para blocos do disco contendo dados do arquivo. Os ponteiros relativos às áreas não-mapeadas têm valor nulo, servindo apenas para indicar que aquela área do arquivo ainda não está mapeada no disco (conforme indicado na Figura 6.19). Caso um processo leia uma área não-mapeada, receberá somente zeros. As áreas não-mapeadas serão alocadas em disco somente quando algum processo escrever nelas. Arquivos esparsos são muito usados por gerenciadores de bancos de dados e outras aplicações que precisem manter arquivos com índices ou tabelas *hash* que possam conter grandes intervalos sem uso.

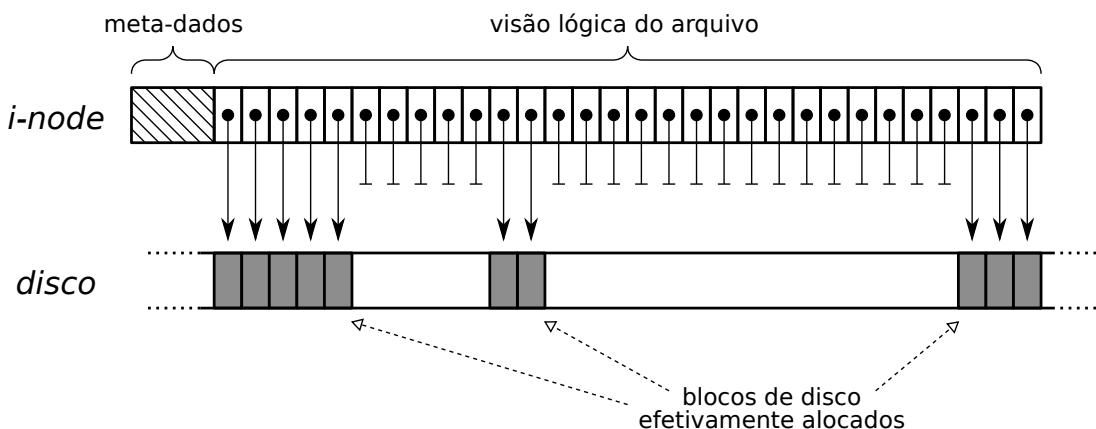


Figura 6.19: Alocação de um arquivo esparsos.

Análise comparativa

A Tabela 6.3 traz um comparativo entre as principais formas de alocação estudadas aqui, sob a ótica de suas características de rapidez, robustez e flexibilidade de uso.

Gerência de 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 meta-dado. Duas

Estratégia	Rapidez	Robustez	Flexibilidade
Contígua	Alta, pois acessos sequencial e direto rápidos, pois os blocos do arquivo estão próximos no disco.	Alta, pois blocos defeituosos não impedem o acesso aos demais blocos do arquivo.	Baixa, pois o tamanho máximo dos arquivos deve ser conhecido a priori; nem sempre é possível aumentar o tamanho de um arquivo existente.
Encadeada	Acesso sequencial é rápido, se os blocos estiverem próximos; o acesso direto é lento, pois é necessário ler todos os blocos a partir do início do arquivo até encontrar o bloco desejado.	Baixa, pois um bloco defeituoso leva à perda dos dados daquele bloco e de todos os blocos subsequentes, até o fim do arquivo.	Alta, pois arquivos podem ser criados em qualquer local do disco, sem risco de fragmentação externa.
FAT	Alta, pois acessos sequencial e direto são rápidos, se os blocos do arquivo estiverem próximos no disco.	Mais robusta que a alocação encadeada, desde que não ocorram erros na tabela de alocação.	Alta, pois arquivos podem ser criados em qualquer local do disco, sem risco de fragmentação externa.
Indexada	Alta, pois os acessos sequencial e direto são rápidos, se os blocos do arquivo estiverem próximos no disco.	Alta, desde que não ocorram erros no <i>i-node</i> nem nos blocos de ponteiros.	Alta, pois arquivos podem ser criados em qualquer local do disco, sem risco de fragmentação externa. No entanto, o tamanho máximo dos arquivos é limitado pelo número de ponteiros definidos nos <i>i-nodes</i> .

Tabela 6.3: Quadro comparativo das estratégias de alocação de arquivos

técnicas de gerência de blocos livres são frequentemente utilizadas: o mapa de bits e a lista de blocos livres [Silberschatz et al., 2001, Tanenbaum, 2003].

Na abordagem de **mapa de bits**, um pequeno conjunto de blocos no início da partição é reservado para manter um mapa de bits. Cada bit nesse mapa de bits representa um bloco lógico da partição, que pode estar livre (o bit vale 1) ou ocupado (o bit vale 0). Essa abordagem como vantagem ser bastante compacta e simples de implementar: em um disco de 80 GBytes com blocos lógicos de 4.096 bytes, seriam necessários 20.971.520 bits no mapa de bits, o que representa 2.621.440 bytes ou 640 blocos (ou seja, 0,003% do total de blocos lógicos do disco).

A abordagem de **lista de blocos livres** pode ser implementada de várias formas. Na forma mais simples, cada bloco livre contém um ponteiro para o próximo bloco livre do disco, de forma similar à alocação encadeada de arquivos vista na Seção 6.4.3. Apesar de simples, essa abordagem é pouco eficiente, por exigir um acesso a disco para cada bloco livre requisitado. A abordagem FAT (Seção 6.4.3) é uma melhoria desta técnica, na qual os blocos livres são indicados por flags específicos na tabela de alocação de arquivos. Outra 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 armazenar uma tabela de *extensões* de blocos livres,

ou seja, a localização e o tamanho de um conjunto de blocos livres consecutivos no disco, de forma similar à alocação contígua (Seção 6.4.3).

6.4.4 O sistema de arquivos virtual

O sistema de arquivos virtual gerencia os aspectos do sistema de arquivos mais próximos do usuário, como a verificação de permissões de acesso, o controle de concorrência (atribuição e liberação travas) e a manutenção de informações sobre os arquivos abertos pelos processos.

Conforme apresentado na Seção 6.2.1, quando um processo abre um arquivo, ele recebe do núcleo uma referência ao arquivo aberto, a ser usada nas operações subsequentes envolvendo aquele arquivo. Em sistemas UNIX, as referências a arquivos abertos são denominadas *descritores de arquivos*, e correspondem a índices de entradas em uma tabela de arquivos abertos pelo processo (*process file table*), mantida pelo núcleo. Cada entrada dessa tabela contém informações relativas ao uso do arquivo por aquele processo, como o ponteiro de posição corrente e o modo de acesso ao arquivo solicitado pelo processo (leitura, escrita, etc.).

Adicionalmente, cada entrada da tabela de arquivos do processo contém uma referência para uma entrada correspondente na tabela global de arquivos abertos (*system file table*) do sistema. Nessa tabela global, cada entrada contém um contador de processos que mantém aquele arquivo aberto, uma trava para controle de compartilhamento e uma referência às estruturas de dados que representam o arquivo no sistema de arquivos onde ele se encontra, além de outras informações [Bach, 1986, Vahalia, 1996, Love, 2004].

A Figura 6.20 apresenta a organização geral das estruturas de controle de arquivos abertos presentes no sistema de arquivos virtual de um núcleo UNIX típico. Essa estrutura é similar em outros sistemas, mas pode ser simplificada em sistemas mais antigos e simples, como no caso do DOS. Deve-se observar que toda essa estrutura é independente do dispositivo físico onde os dados estão armazenados e da estratégia de alocação de arquivos utilizada; por essa razão, esta camada é denominada *sistema de arquivos virtual*. Essa transparência permite que os processos acessem de maneira uniforme, usando a mesma interface, arquivos em qualquer meio de armazenamento e armazenados sob qualquer estratégia de alocação.

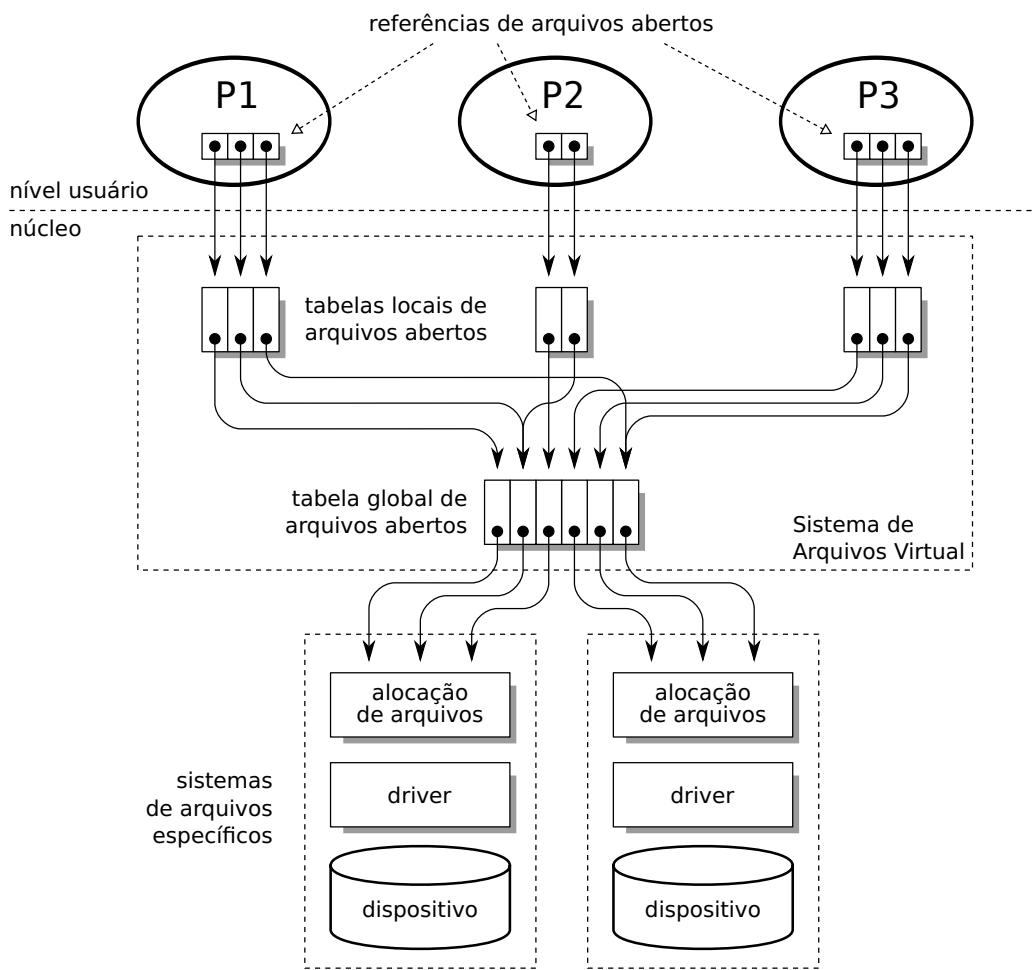


Figura 6.20: Estruturas de controle de arquivos abertos em um núcleo UNIX.

6.5 Tópicos avançados

- Journaling FS
- Extents
- Log-structured Fyle Systems

Capítulo 7

Gerência de entrada/saída

Este conteúdo está em elaboração. Ainda há muito o que escrever aqui...

7.1 Introdução

Um computador é constituído basicamente de um ou mais 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, WLAN, *Bluetooth* e de telefonia celular.

Esses exemplos são típicos de computadores pessoais e de computadores menores, como os *smartphones*. 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 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.

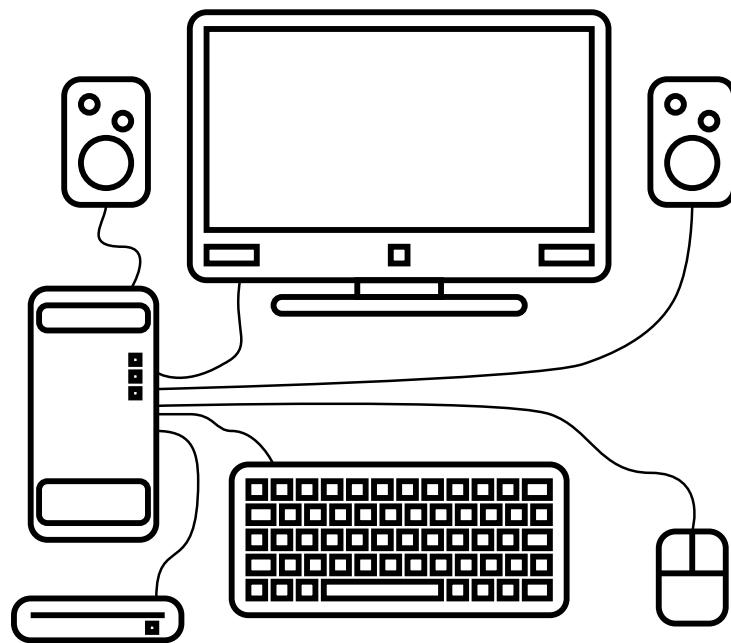


Figura 7.1: Dispositivos de entrada/saída.

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 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 periféricicos 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 da operação dos dispositivos de entrada/saída sob a ótica do sistema operacional. Inicialmente serão discutidas as principais características dos dispositivos de entrada/saída usualmente presentes nos computadores convencionais para a interação com o usuário, armazenamento de dados e comunicação via rede. A seguir, a arquitetura de hardware e software responsável pela operação dos dispositivos de entrada/saída será detalhada. Na sequência, as principais estratégias de interação entre o software e o hardware serão apresentadas. Os *drivers*, componentes de software responsáveis pela interação do sistema operacional com o hardware de entrada/saída, terão sua estrutura e princípio de funcionamento abordados em seguida. Por fim, alguns sub-sistemas de entrada/saída específicos, considerados os mais relevantes nos computadores de uso geral atualmente em uso, serão estudados com maior profundidade.

7.2 Dispositivos de entrada/saída

Um dispositivo de entrada/saída realiza a interação entre os sistemas internos de um computador (processadores e memória) e o mundo exterior.

7.2.1 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 7.2 mostra a estrutura básica de captura e reprodução de áudio em um computador pessoal, que é um bom exemplo de dispositivo de entrada/saída.

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 7.1 traz alguns exemplos de dispositivos de entrada/saída com suas velocidades típicas de transferência de dados.

7.2.2 Barramentos

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

¹Sensores e atuadores são denominados genericamente *dispositivos transdutores*, pois transformam energia externa (como luz, calor, som ou movimento) em sinais elétricos, ou vice-versa.

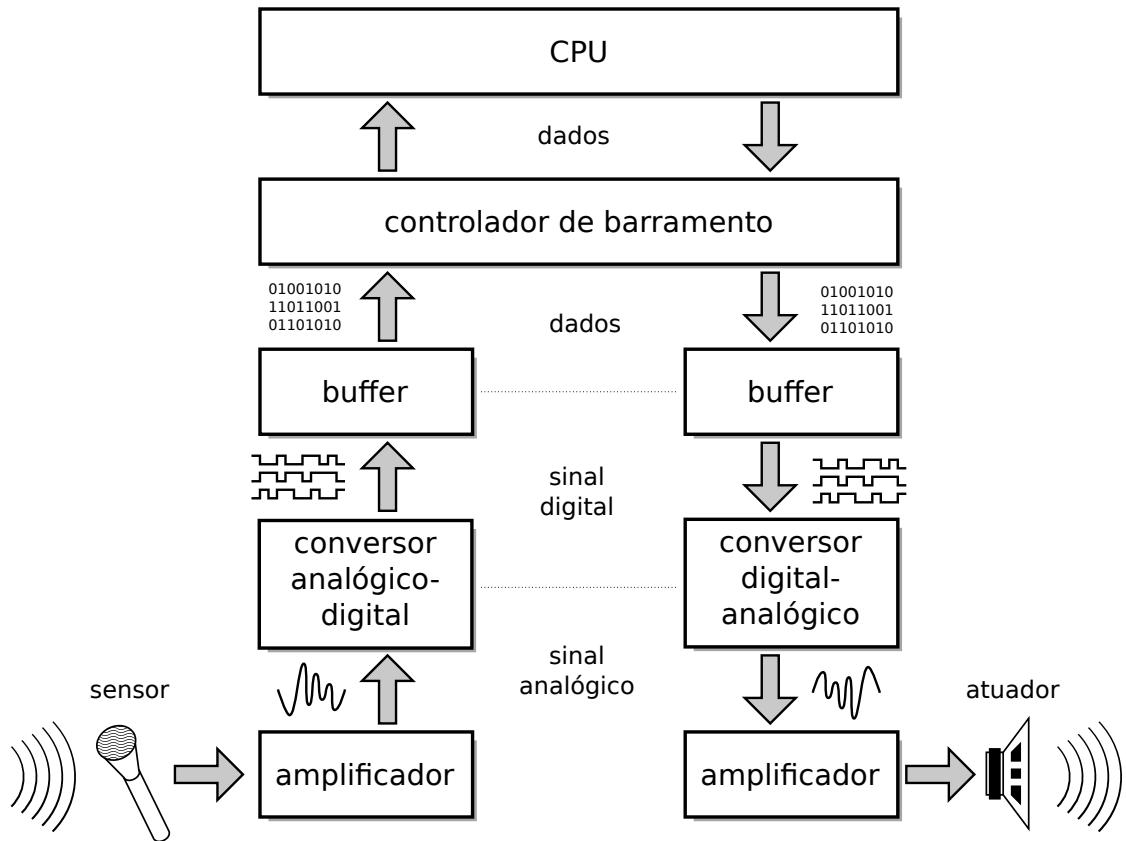


Figura 7.2: Estrutura básica da entrada e saída de áudio.

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

²O *chipset* de um computador é um conjunto de controladores e circuitos auxiliares de hardware integrados à placa-mãe, que provêem 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.

Tabela 7.1: 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 <i>Fast Ethernet</i>	11.6 MB/s
Chave ou disco USB 2.0	60 MB/s
Interface de rede <i>Gigabit Ethernet</i>	116 MB/s
Disco rígido SATA 2	300 MB/s
Interface gráfica <i>high-end</i>	4.2 GB/s

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*), relógio de tempo real (responsável pelas interrupções de tempo usadas pelo escalonador de processos), controle de energia e o acesso à memória BIOS. O processador se comunica com a *south bridge* indiretamente, através da *north bridge*.

A Figura 7.3 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].

7.2.3 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 7.4):

- 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;

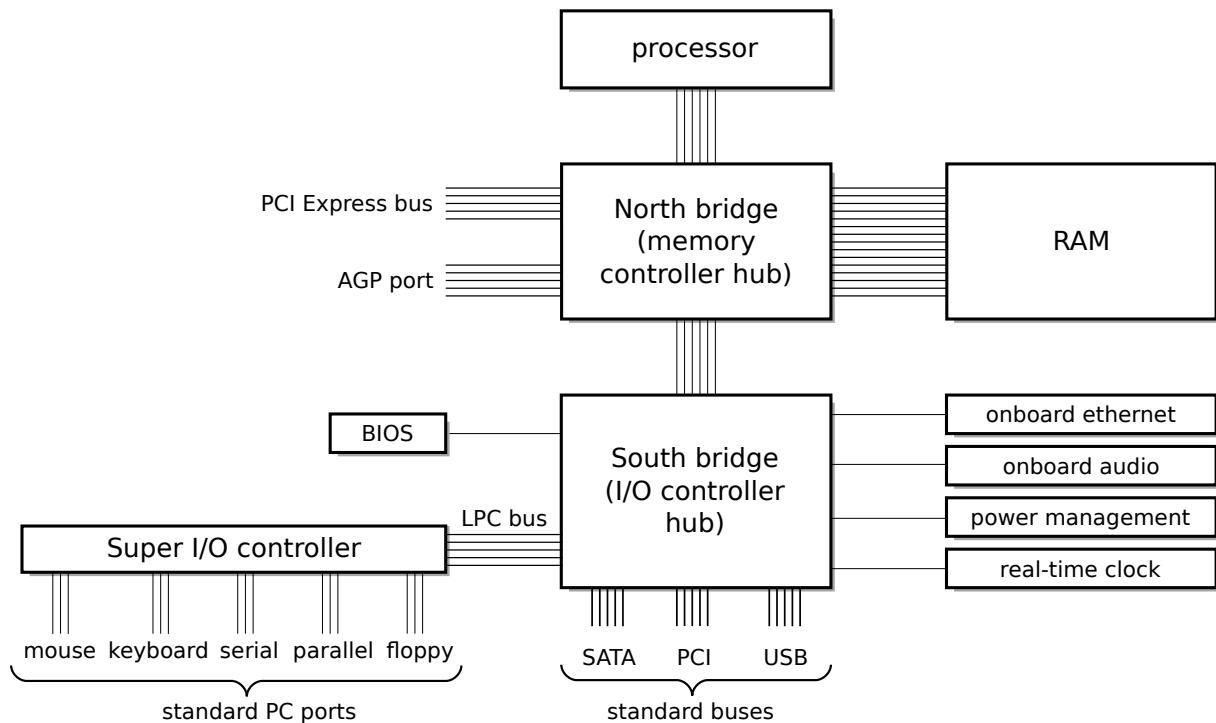


Figura 7.3: Arquitetura típica de um PC atual.

- 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.

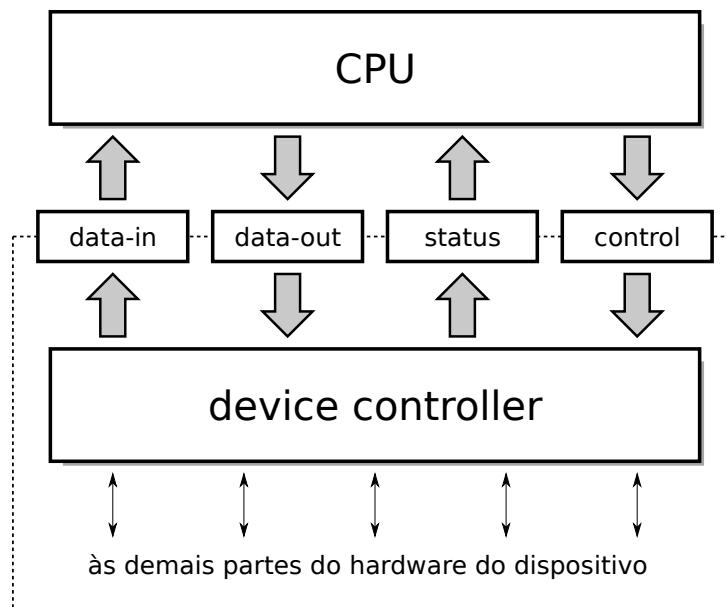


Figura 7.4: Portas de interface de um dispositivo de entrada/saída.

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. As portas de uma interface paralela operando no modo padrão (SPP - *Standard Parallel Port*) estão descritas a seguir [Patterson and Henessy, 2005]:

- P_0 (*data port*): porta de saída, usada para enviar bytes 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{nIRQ} : se 0, indica que o controlador gerou uma interrupção (Seção 7.2.5);
 3. *error*: há um erro interno na impressora;
 4. *select*: a impressora está pronta (*online*);
 5. *paper_out*: falta papel na impressora;
 6. \overline{ack} : se 0, indica que um dado foi recebido (gera um pulso em 0 com duração de ao menos $1\mu s$);
 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{strobe} : informa a interface que há um dado em P_0 (deve ser gerado um pulso em 0, com duração de ao menos $1\mu s$);
 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 7.2.5);
 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 Parallel 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 7.5. 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 $busy = 0$.

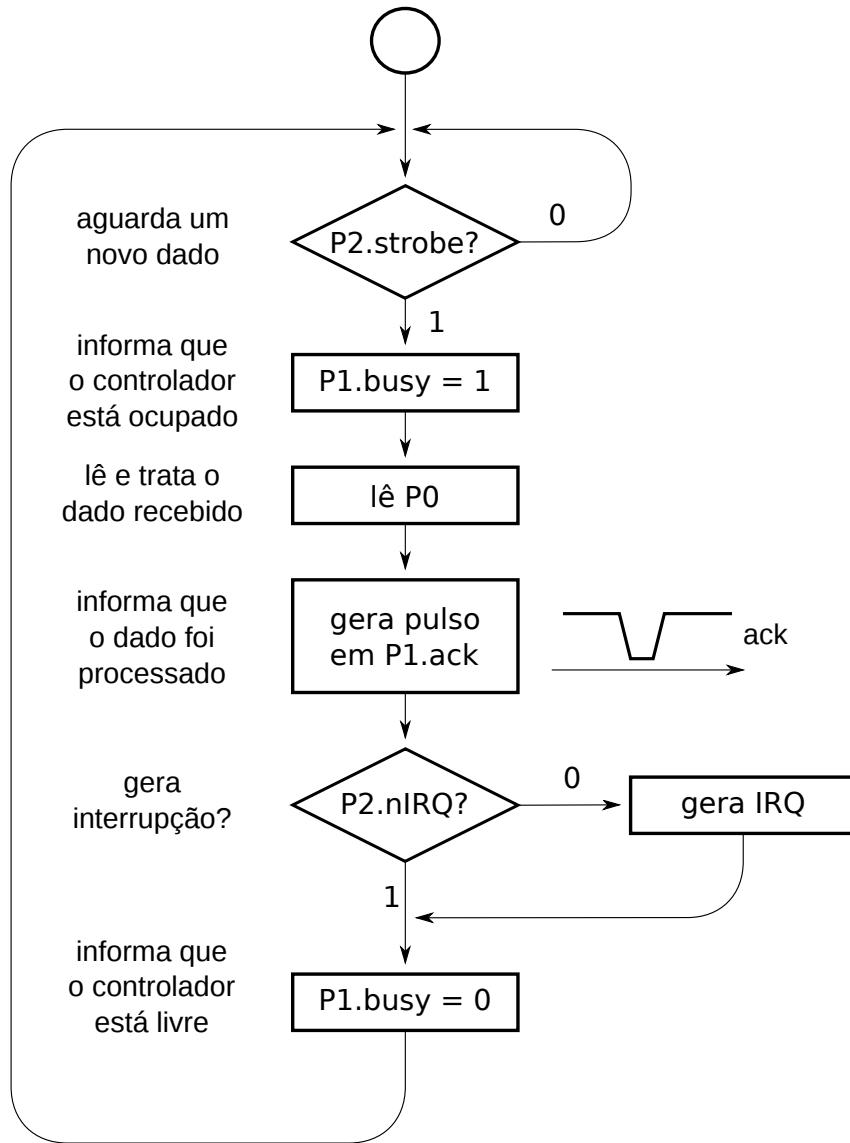


Figura 7.5: Comportamento básico do controlador da porta paralela.

7.2.4 Endereçamento

A forma de acesso aos registradores 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*”.

Na entrada/saída mapeada em portas, é definido um **espaço de endereços de entrada/saída** (*I/O address space*) separado da memória principal e normalmente compreendido entre 0 e 64K. Para distinguir entre endereços de memória e de portas de entrada/saída, o barramento de controle do processador possui uma linha IO/\bar{M} , que indica se o endereço presente no barramento de endereços se refere a uma posição de memória (se $IO/\bar{M} = 0$) ou a uma porta de entrada/saída (se $IO/\bar{M} = 1$). A Tabela 7.2 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 037F

Tabela 7.2: Endereços de portas de E/S de alguns dispositivos.

Uma outra forma de acesso aos dispositivos de entrada/saída, usada frequentemente em interfaces gráficas e de rede, 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 algum tipo de barramento. 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.

7.2.5 Interrupções

O acesso aos controladores de dispositivos através de seus registradores é conveniente para a comunicação no sentido *processador* → *controlador*, ou seja, para as interações iniciadas pelo processador. Entretanto, pode ser problemática no sentido *controlador* → *processador*, caso o controlador precise informar algo ao processador de forma assíncrona, sem que o processador esteja esperando. Nesse caso, o controlador pode utilizar uma **requisição de interrupção** (IRQ - *Interrupt Request*) para notificar o processador sobre algum evento importante, como a conclusão de uma operação solicitada, a disponibilidade de um novo dado ou a ocorrência de algum problema no dispositivo.

As requisições de interrupção são sinais elétricos veiculados através do barramento de controle do computador. Cada interrupção está associada a um número inteiro, geralmente na faixa 0–31 ou 0–63, o que permite identificar o dispositivo que a solicitou. A Tabela 7.3 informa os números de interrupção associados a alguns dispositivos periféricos típicos.

Dispositivo	Interrupção
teclado	1
mouse PS/2	12
barramento IDE primário	14
barramento IDE secundário	15
relógio de tempo real	8
porta serial COM1	4
porta serial COM2	3
porta paralela LPT1	7

Tabela 7.3: Interrupções geradas por alguns dispositivos.

Ao receber uma 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 a interrupção, ou seja, executar as ações necessárias para identificar e atender o dispositivo que gerou a requisição. Ao final da rotina de tratamento da interrupção, o processador retoma o código que estava executando quando foi interrompido. A Figura 7.6 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 solicitação de interrupção (IRQ) ao processador;
4. O processador recebe a interrupção, salva na pilha seu estado atual (o conteúdo de seus registradores) e desvia sua execução para uma rotina de tratamento da interrupção;

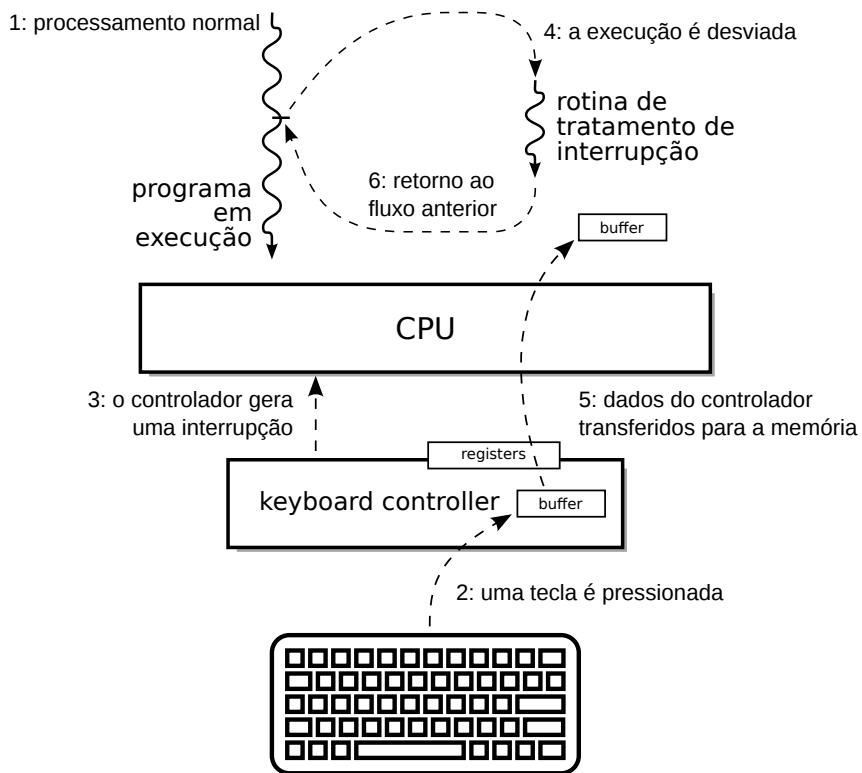


Figura 7.6: Roteiro típico de um tratamento de interrupção

5. Ao executar, essa rotina acessa os registradores 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 milhares de interrupções por segundo, dependendo da carga de trabalho e da configuração do sistema (número e natureza dos 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 atuais define um vetor de endereços de funções denominado *Vetor de Interrupções* (IV - *Interrupt Vector*); cada entrada desse vetor aponta para a rotina de tratamento da interrupção correspondente. Por exemplo, se a entrada 5 do vetor contém o valor 3C20h, então a rotina de tratamento da IRQ

5 iniciará na posição 3C20h da memória RAM. Dependendo do hardware, o vetor 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.

As interrupções recebidas pelo processador têm como origem eventos externos a ele, ocorridos nos dispositivos periféricos e reportados por seus controladores. Entretanto, eventos gerados pelo próprio processador podem ocasionar o desvio da execução usando o mesmo mecanismo de interrupção: são as **exceções**. Eventos como instruções ilegais (inexistentes ou com operandos inválidos), divisão por zero ou outros erros de software disparam exceções internas no processador, que resultam na ativação de rotinas de tratamento de exceção registradas no vetor de interrupções. A Tabela 7.4 apresenta algumas exceções previstas pelo processador Intel Pentium (extraída de [Patterson and Hennessy, 2005]).

Tabela 7.4: Algumas exceções do processador Pentium.

Exceção	Descrição
0	divide error
3	breakpoint
5	bound range exception
6	invalid opcode
9	coprocessor segment overrun
11	segment not present
12	stack fault
13	general protection
14	page fault
16	floating point error

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 7.7 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.

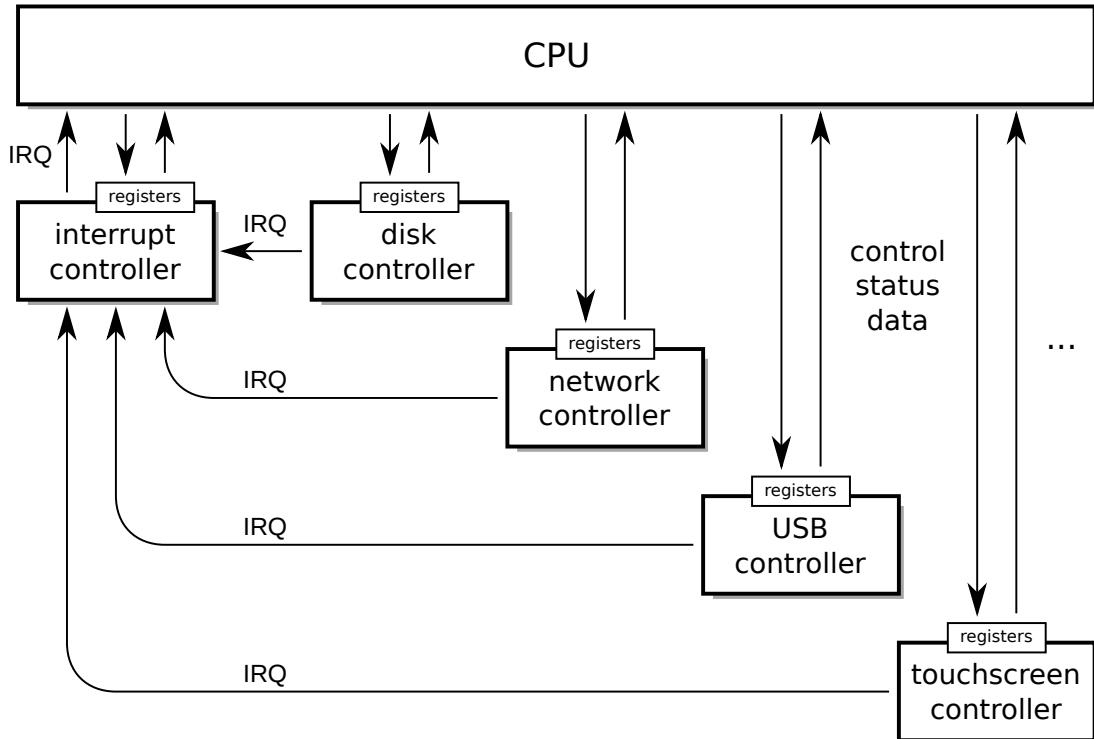


Figura 7.7: Uso de um controlador de interrupções.

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 for concluída.

7.3 Software de entrada/saída

O sistema operacional é responsável por oferecer acesso aos dispositivos de entrada/saída às aplicações e, em última instância, 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 imenso desafio. Além disso, como cada dispositivo define sua própria interface e modo de operação, o núcleo do sistema operacional deve implementar código de baixo nível para interagir com milhares de tipos de dispositivos distintos. Como exemplo, cerca de 60% das 12 milhões de linhas

de código do núcleo Linux 2.6.31 pertencem a código de *drivers* de dispositivos de entrada/saída.

Para simplificar o acesso e a gerência dos dispositivos de entrada/saída, o código do sistema operacional é estruturado em camadas, que levam das portas de entrada/saída, interrupções de operações de DMA a interfaces de acesso abstratas, como arquivos e *sockets* de rede. Uma visão conceitual dessa estrutura em camadas pode ser vista na Figura 7.8. Nessa figura, a camada inferior corresponde aos dispositivos periféricos propriamente ditos, como discos rígidos, teclados, etc. A camada logo acima, implementada em hardware, corresponde ao controlador específico de cada dispositivo (controlador IDE, SCSI, SATA, etc.) e aos controladores de DMA e de interrupções, pertencentes ao *chipset* do computador.

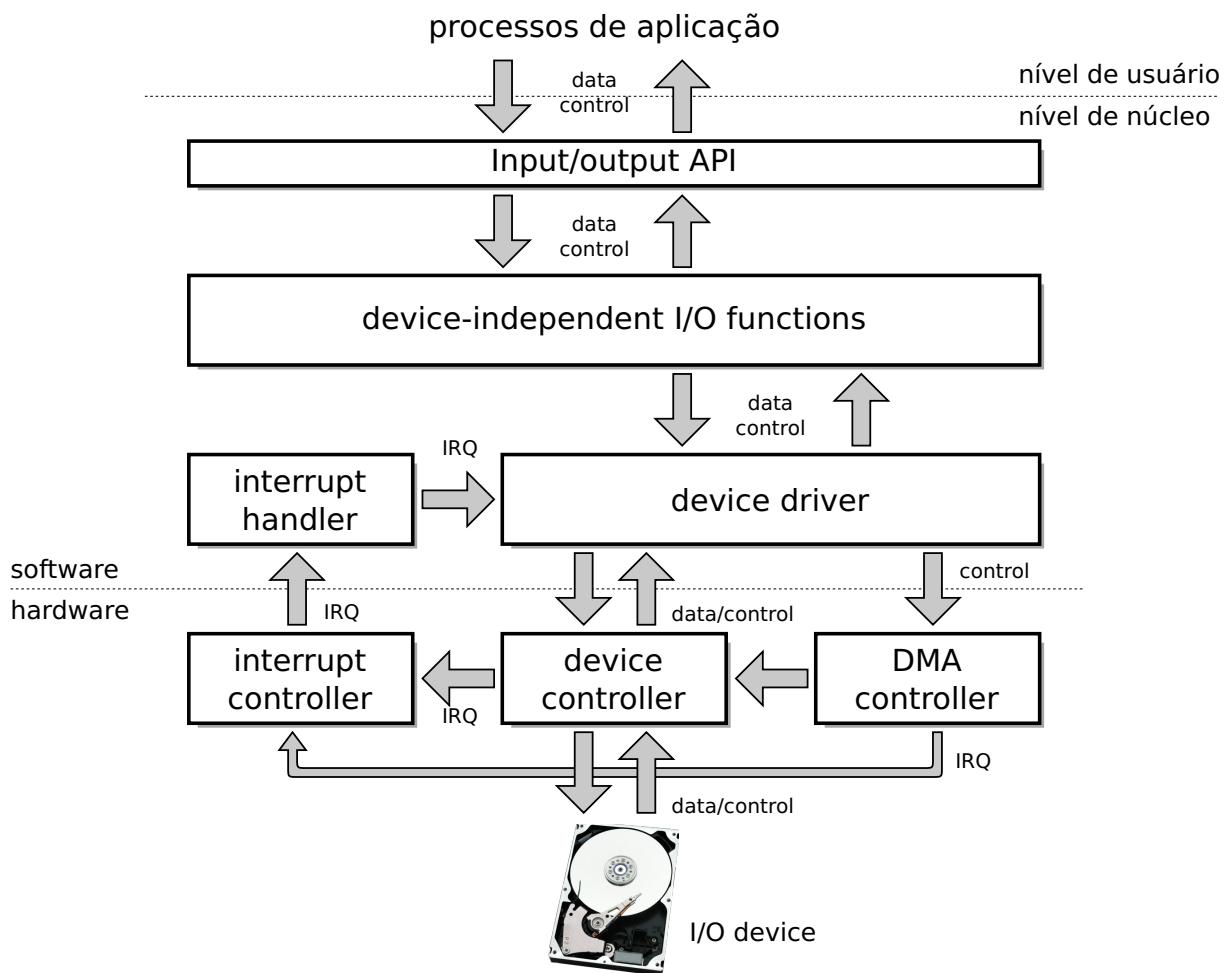


Figura 7.8: Estrutura em camadas do software de entrada/saída.

A primeira camada de software corresponde às rotinas de tratamento de interrupções (*interrupt handles*) e aos *drivers* de entrada/saída. As rotinas de tratamento de interrupção são acionadas pelo mecanismo de interrupção do processador a cada interrupção provinda do controlador de interrupções e servem basicamente para registrar sua ocorrência. A execução dessas rotinas deve ser muito breve, pois durante o tratamento de uma interrupção o processador desabilita a ocorrência de novas interrupções,

conforme discutido na Seção 7.2.5. Assim, quando uma rotina é acionada, ela apenas reconhece a interrupção ocorrida junto ao controlador, cria um **descritor de evento** (*event handler*) contendo os dados da interrupção, o insere em uma fila de eventos pendentes mantida pelo *driver* do dispositivo, notifica o *driver* e conclui.

Os eventos da fila de eventos pendentes mantida por cada driver são tratados posterior, quando o processador estiver livre. A separação do tratamento de interrupções em dois níveis de urgência levar a estruturar o código de tratamento de cada interrupção também em dois níveis: o **bottom-half**, que compreende as ações imediatas a executar quando a interrupção ocorre, e o **top-half**, que compreende o restante das ações de tratamento da interrupção.

7.3.1 Classes de dispositivos

Para simplificar a construção de aplicações (e do próprio sistema operacional), os dispositivos de entrada/saída são agrupados em classes ...

Os **dispositivos orientados a blocos** são aqueles em que as operações de entrada ou saída de dados são feitas usando blocos de bytes e nunca bytes isolados. Discos rígidos, fitas magnéticas e outros dispositivos de armazenamento são exemplos típicos desta categoria.

Os **dispositivos orientados a caracteres** são aqueles cujas transferências de dados são sempre feitas byte por byte, ou usando blocos de bytes de tamanho variável, cujo tamanho mínimo seja um byte. Dispositivos ligados às interfaces paralelas e seriais do computador, como *mouse* e teclado, são os exemplos mais clássicos deste tipo de dispositivo. Os terminais de texto e modems de transmissão de dados por linhas seriais (como as linhas telefônicas) também são vistos como dispositivos orientados a caracteres.

As **interfaces de rede** são colocadas em uma classe particular, pois são vistos como dispositivos orientados a blocos (os pacotes de rede são blocos), esses blocos são endereçáveis (os endereços dos destinos dos pacotes), mas a saída é feita de forma sequencial, bloco após bloco, e normalmente não é possível resgatar ou apagar um bloco enviado ao dispositivo.

sentido dos fluxos:	entrada	saída
granularidade	caractere : os dados são enviados ou recebidos byte por byte	bloco : os dados são enviados/recebidos em blocos de tamanho fixo
exemplos:	terminais, portas paralelas e seriais, mouses, teclados	discos rígidos, interfaces de rede (pacotes), fitas magnéticas
acesso	sequencial : os dados são enviados ou recebidos em sequência, um após o outro	direto : a cada dado enviado ou recebido é associado um endereço (respectivamente de destino ou de origem)
exemplos:	porta paralela/serial, mouse, teclado, fita magnética	disco rígido, interface de rede
persistência:	persistente , se o dado enviado pode ser resgatado diretamente (ou, em outras palavras, se os dados lidos do dispositivo foram anteriormente escritos nele por aquele computador ou algum outro)	volátil , se o dado enviado é “consumido” pelo dispositivo, ou se o dado recebido do dispositivo foi “produzido” por ele e não anteriormente depositado nele.
exemplos:	fita magnética, disco rígido	interface de rede, porta serial/-paralela

- granularidade da informação: byte, bloco, stream
- tipos de dispositivos: a blocos, a caracteres, de rede, blocos sequenciais? (fita, rede)
- tipos de interface: bloqueante, não bloqueante, assíncrona
- arquitetura de E/S do kernel
 - estrutura de E/S do kernel: de devices genéricos a drivers específicos
 - interfaces, drivers, irq handlers, controllers
- Drivers
 - arquitetura geral
 - a estrutura de um driver
 - fluxograma de execução
 - top e bottom half
 - rotinas oferecidas aos processos
 - acesso via /dev
 - acesso ao hardware
 - integração ao kernel (recompilação ou módulos dinâmicos)

7.3.2 Estratégias de interação

O sistema operacional deve interagir com cada dispositivo de entrada/saída para realizar as operações desejadas, através das portas de seu controlador. Esta seção aborda as três estratégias de interação mais frequentemente usadas pelo sistema operacional, que são a entrada/saída controlada por programa, a controlada por eventos e o acesso direto à memória, detalhados a seguir.

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 ou polling**. Nesta abordagem, o sistema operacional 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 7.2.3, o comportamento do processador em uma operação de saída na porta paralela usando essa abordagem seria descrito pelo seguinte pseudo-código, no qual as leituras e escritas nas portas são representadas respectivamente pelas funções `in(port)` e `out(port, value)`³:

```

1 // portas da interface paralela LPT1 (endereço inicial em 0378h)
2 #define P0    0x0378      # porta de dados
3 #define P1    0x0379      # porta de status
4 #define P2    0x037A      # porta de controle
5
6 // máscaras para alguns bits de controle e status
7 #define BUSY  0x80        # 1000 0000 (bit 7)
8 #define ACK   0x40        # 0100 0000 (bit 6)
9 #define STROBE 0x01       # 0000 0001 (bit 0)
10
11 // buffer de bytes a enviar
12 unsigned char buffer[BUFSIZE] ;
13
14 polling_output ()
15 {
16     for (i = 0 ; i < BUFSIZE ; i++)
17     {
18         // espera o controlador ficar livre (bit BUSY deve ser zero).
19         while (in (P1) & BUSY) ;
20
21         // escreve o byte a enviar na porta P0.
22         out (P0, buffer[i]) ;
23
24         // gera pulso de 1 microsegundo no bit STROBE de P2,
25         // para indicar ao controlador que há um novo dado.
26         out (P2, in (P2) | STROBE) ;
27         usleep (1) ;
28         out (P2, in (P2) & ! STROBE) ;
29
30         // espera o byte terminar de ser tratado (pulso "zero" em ACK).
31         while (in (P1) & ACK) ;
32     }
33 }
```

Em conjunto, processador e controlador executam ações coordenadas e complementares: o processador espera que o controlador esteja livre antes de enviar um novo dado; por sua vez, o controlador espera que o processador lhe envie um novo dado para processar. Essa interação é ilustrada na Figura 7.9. 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 do controlador 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 tempo de resposta do dispositivo for longo, sendo por isso pouco usada em sistemas operacionais de propósito geral. Seu uso se concentra

³Como o bit *BUSY* da porta P_1 deve retornar ao valor zero (0) após o pulso no bit *ACK*, o pseudo-código poderia ser simplificado, eliminando o laço de espera sobre *ACK*; contudo, esse laço foi mantido para maior clareza didática.

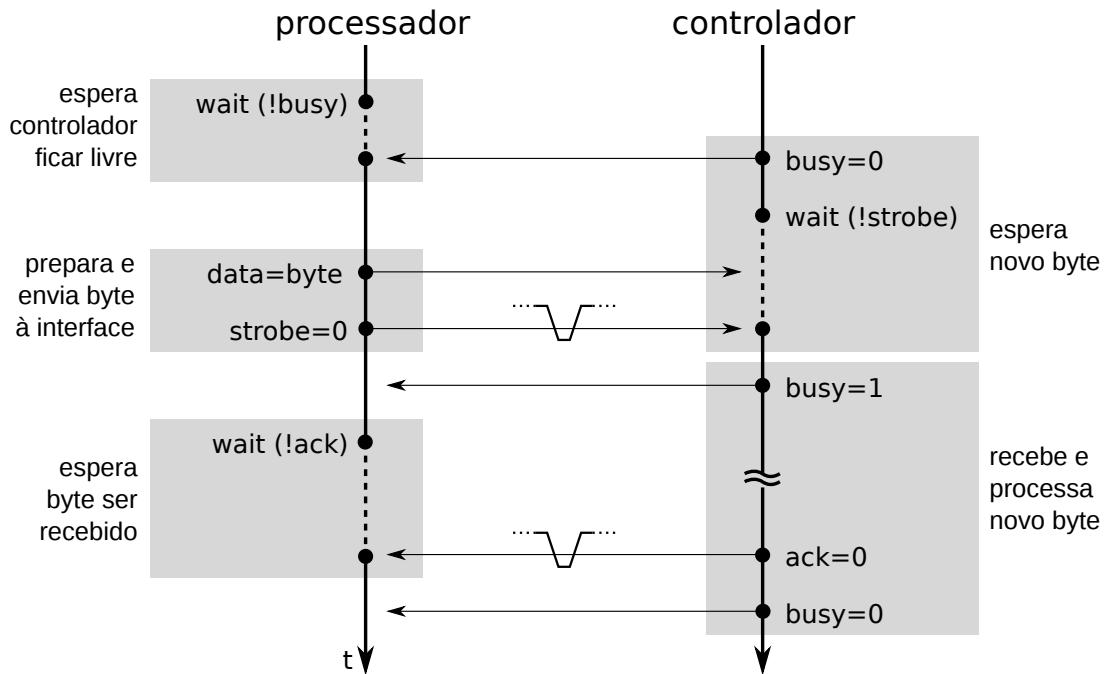


Figura 7.9: Entrada/saída controlada por programa.

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 10ms), 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.

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, desviando o processador para tratar outro processo ou *thread*. Quando o dispositivo tiver terminado de processar a requisição, seu controlador irá gerar uma interrupção (IRQ) para notificar o processador, que poderá então retomar a execução daquele fluxo quando for conveniente. 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 sistema operacional a pedido de um processo ou *thread*, e uma **rotina de tratamento de interrupção** (*interrupt handler*), ativado a cada interrupção, para prosseguir a transferência de dados ou para informar sobre sua conclusão. Considerando novamente a saída de um buffer de N bytes em uma interface paralela (descrita na Seção 7.2.3), o pseudo-código a seguir representa o lançamento da operação de E/S pelo processador e

a rotina de tratamento de interrupção (subentende-se as constantes e variáveis definidas na listagem anterior):

```

1 // lançamento da operação de saída de dados
2 interrupt_driven_output ()
3 {
4     i = 0 ;
5
6     // espera o controlador ficar livre (bit BUSY de P1 deve ser zero).
7     while (in (P1) & BUSY) ;
8
9     // escreve o byte a enviar na porta P0.
10    out (P0, buffer[i]) ;
11
12    // gera pulso de 1 microsegundo no bit STROBE de P2.
13    out (P2, in (P2) | STROBE) ;
14    usleep (1) ;
15    out (P2, in (P2) & ! STROBE) ;
16
17    // suspende o processo solicitante, liberando o processador.
18    schedule () ;
19 }
20
21 // rotina de tratamento de interrupções da interface paralela
22 interrupt_handle ()
23 {
24     i++ ;
25     if (i >= BUFSIZE)
26         // a saída terminou, acordar o processo solicitante.
27         awake_process () ;
28     else
29     {
30         // escreve o byte a enviar na porta P0.
31         out (P0, buffer[i]) ;
32
33         // gera pulso de 1 microsegundo no bit STROBE de P2.
34         out (P2, in (P2) | STROBE) ;
35         usleep (1) ;
36         out (P2, in (P2) & ! STROBE) ;
37     }
38
39     // informa o controlador de interrupções que a IRQ foi tratada.
40     acknowledge_irq () ;
41 }
```

Nesse pseudo-código, percebe-se que o processador inicia a transferência de dados para a interface paralela e suspende o processo solicitante (chamada *schedule*), liberando o processador para outras atividades. A cada interrupção, a rotina de tratamento é ativada para enviar um novo byte à interface paralela, até que todos os bytes do *buffer* tenham sido enviados, quando então sinaliza ao escalonador que o processo solicitante pode retomar sua execução (chamada *resume*). Conforme visto anteriormente, 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 , Seção 7.2.3). O diagrama da Figura 7.10 ilustra de forma simplificada a estratégia de entrada/saída usando interrupções.

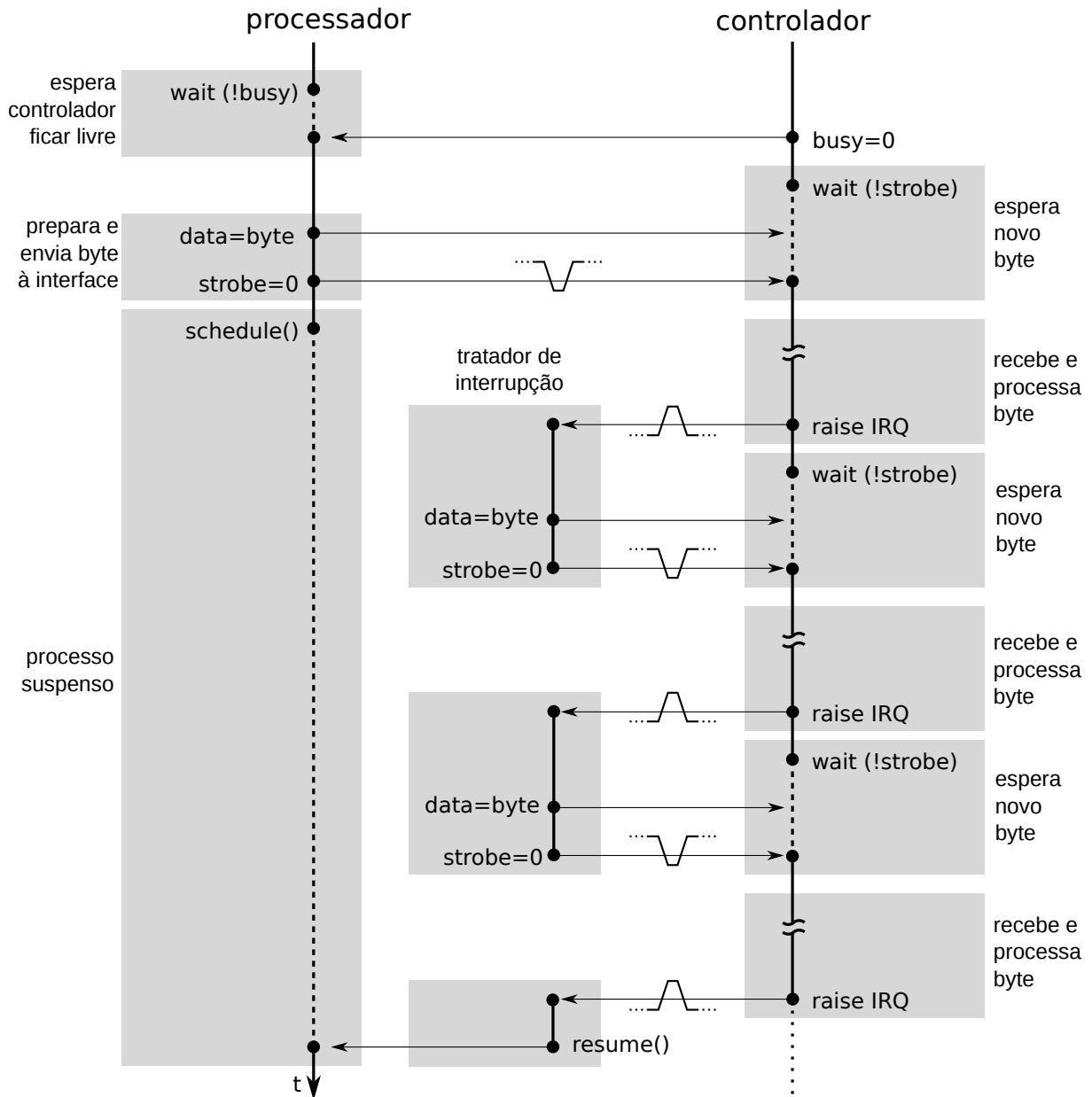


Figura 7.10: Entrada/saída controlada por eventos (interrupções).

Durante a execução da rotina de tratamento de uma 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) 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.

A maioria dos sistemas operacionais implementa o tratamento de interrupções em dois níveis distintos: 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 lançado quando a interrupção ocorre e deve executar rapidamente, pois as demais interrupções estão inibidas. Sua atividade se restringe a ações essenciais, como reconhecer a ocorrência da interrupção junto ao controlador e registrar dados sobre a mesma em uma fila de eventos pendentes, para tratamento posterior.

Por sua vez, o tratador secundário, também conhecido como *soft/slow interrupt handler* ou ainda *bottom-half handler*, tem por objetivo tratar a fila de eventos pendentes registrados pelo tratador primário, quando o processador estiver disponível. Ele pode ser escalonado e suspenso da mesma forma que um processo ou *thread*, embora normalmente execute com maior prioridade. Embora mais complexa, esta estrutura em dois nível traz algumas vantagens: ao permitir um tratamento mais rápido de cada interrupção, minimiza o risco de perder interrupções concomitantes; 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 and Solomon, 2004]. 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.

Acesso direto à memória

Na maioria das vezes, o tratamento de operações de entrada/saída é uma operação lenta, pois os dispositivos 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 os registradores do canal 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 de 512 bytes, que é o tamanho padrão dos setores do disco rígido;
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 interrupção (IRQ).

A dinâmica dessas operações é ilustrada de forma simplificada na Figura 7.11. Uma vez efetuado o passo 1, o processador fica livre para outras atividades⁴, 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.

⁴Obviamente 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 memoria para evitar conflitos. Mais detalhes sobre contenção de acesso à memória durante operações de DMA podem ser obtidos em [Patterson and Henessy, 2005].

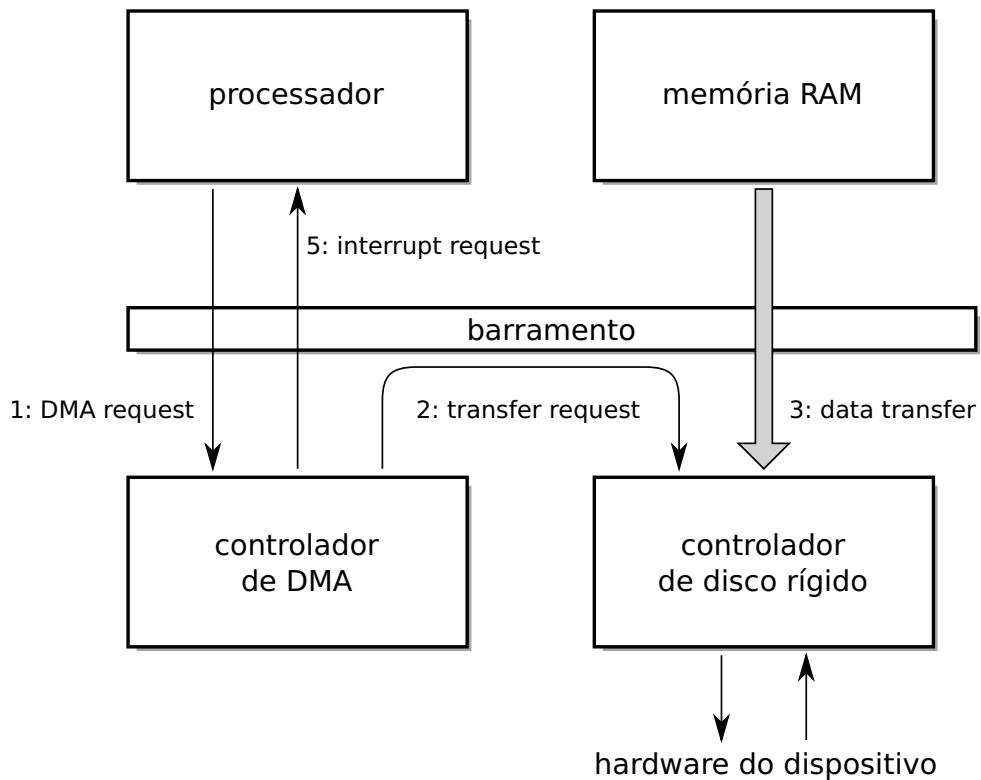


Figura 7.11: Funcionamento do acesso direto à memória.

```

1 // requisição da operação de saída através de DMA
2 dma_driven_output ()
3 {
4     // solicita uma operação DMA, informando os dados da transferência
5     // através da estrutura "dma_data".
6     request_dma (dma_data) ;
7
8     // suspende o processo solicitante, liberando o processador.
9     schedule () ;
10 }
11
12 // rotina de tratamento da interrupção do controlador de DMA
13 interrupt_handle ()
14 {
15     // informa o controlador de interrupções que a IRQ foi tratada.
16     acknowledge_irq () ;
17
18     // saída terminou, acordar o processo solicitante.
19     resume (...) ;
20 }
```

A implementação dos mecanismos de DMA depende da arquitetura e do barramento considerado. Computadores mais antigos dispunham de um controlador de DMA central, que oferecia vários **canais de DMA** distintos, permitindo a realização de transferências de dados por 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 de dados de/para a memória sem depender do processador principal [Corbet et al., 2005], gerenciando assim seu próprio canal DMA.

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 de outro computador através 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 preponderantemente nas operações de entrada/saída envolvendo dispositivos que produzem ou consomem grandes volumes de dados, como interfaces de rede, entradas e saídas de áudio, interfaces gráficas e discos.

Nas próximas seções serão discutidos alguns aspectos relevantes dos dispositivos de entrada/saída mais usados e da forma como estes são acessados e gerenciados nos sistemas de computação pessoal. Sempre que possível, buscar-se-á adotar uma abordagem independente de sistema operacional, concentrando a atenção sobre os aspectos comuns que devem ser considerados por todos os sistemas.

7.4 Discos rígidos

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 ópticos (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 memória virtual (Seção 5.7).

Esta seção inicialmente discute 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, apresenta aspectos de software que estão sob a responsabilidade direta do sistema operacional, como o *caching* de blocos e o escalonamento de operações de leitura/escrita no disco. Por fim, apresenta as técnicas RAID para a composição de discos rígidos, que visam melhorar seu desempenho e/ou confiabilidade.

7.4.1 Estrutura física

Um disco rígido é composto por um ou mais discos metálicos que giram juntos em alta velocidade (entre 4.200 e 15.000 RPM), acionados por um motor elétrico. Para cada face de cada disco há uma cabeça de leitura, responsável por ler e escrever dados através da magnetização de pequenas áreas da superfície metálica. Cada face é dividida logicamente em **trilhas** e **setores**; a interseção de uma trilha e um setor em uma face define um **bloco físico**, que é a unidade básica de armazenamento e transferência de dados no disco. Os discos rígidos atuais (até 2010) usam blocos de 512 bytes, mas o padrão da indústria está migrando para blocos físicos de 4.096 bytes. A Figura 7.12 apresenta os principais elementos que compõem a estrutura de um disco rígido. Um disco típico comporta milhares de trilhas e centenas de setores por face de disco [Patterson and Hennessy, 2005].

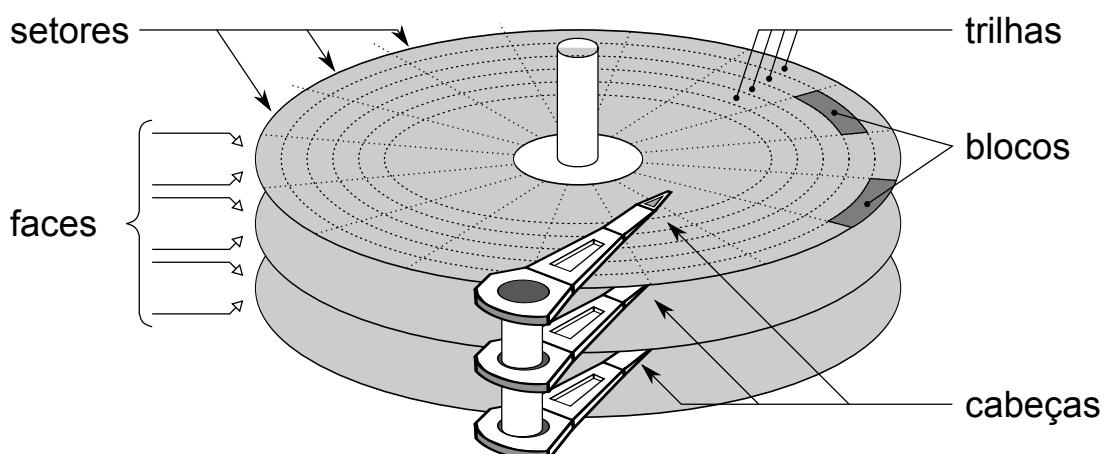


Figura 7.12: Elementos da estrutura de um disco rígido.

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.

7.4.2 Interface de hardware

Conforme estudado anteriormente (Seções 6.4.1 e 7.2), o acesso do processador ao disco rígido é feito através de uma *controladora* em hardware, ligada ao barramento do computador. Por sua vez, o disco rígido é ligado à controladora de disco através de um barramento de interconexão, que pode usar diversas tecnologias. As mais comuns estão descritas na sequência:

- **SATA:** *Serial AT Attachment*, padrão dos *desktops* e *notebooks* atuais; a transmissão dos dados entre disco e controladora é serial, atingindo taxas entre 150 MB/s e 300 MB/s.
- **IDE:**
- **SAS:**
- **SCSI:**

7.4.3 Escalonamento de acessos

Em um sistema multi-tarefas, várias aplicações e processos do sistema podem solicitar acessos concorrentes ao disco rígido, para escrita e leitura de dados. Como o disco só pode atender a uma requisição de acesso por vez, é necessário criar uma fila de acessos pendentes. Cada nova requisição de acesso é colocada nessa fila e o processo solicitante é suspenso até ela ser atendida. Sempre que o disco concluir um acesso, o sistema operacional busca nessa fila a próxima requisição de acesso a ser atendido. A ordem de atendimento das requisições de acesso a disco pendentes 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 1.024 blocos, 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, 71, 447, 161, 659, 335, 524}. Para simplificar, considera-se que nenhum novo pedido de acesso chegará à fila durante a execução dos algoritmos.

FCFS (*First Come, First Served*): esta abordagem consiste em atender as requisições de acesso na ordem em que elas foram pedidas pelos processos. É a estratégia mais simples de implementar, mas raramente oferece um bom desempenho. Se os pedidos de acesso estiverem espalhados ao longo do disco, este irá perder muito tempo movendo a cabeça de leitura de um lado para o outro. A Figura 7.13 mostra os deslocamentos da cabeça de leitura para atender os pedidos de acesso da fila de exemplo. Pode-se perceber que a cabeça de leitura teve de percorrer 3374 blocos do disco para atender todos pedidos da fila.

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 pedido que acabou de ser atendido). Dessa forma, ela busca reduzir os movimentos da cabeça de leitura, e com isso o tempo perdido entre os acessos atendidos. A estratégia SSTF está ilustrada na figura 7.14. Pode-se observar uma forte redução da movimentação da cabeça de leitura em relação à estratégia FCFS, que passou de 3374 para 1320 blocos percorridos. Contudo, essa estratégia não garante um percurso mínimo. Por exemplo, o percurso 500 → 524 → 659 → 914 → 447 → 335 → 278 → 161 → 71 percorreria apenas 1257 blocos.

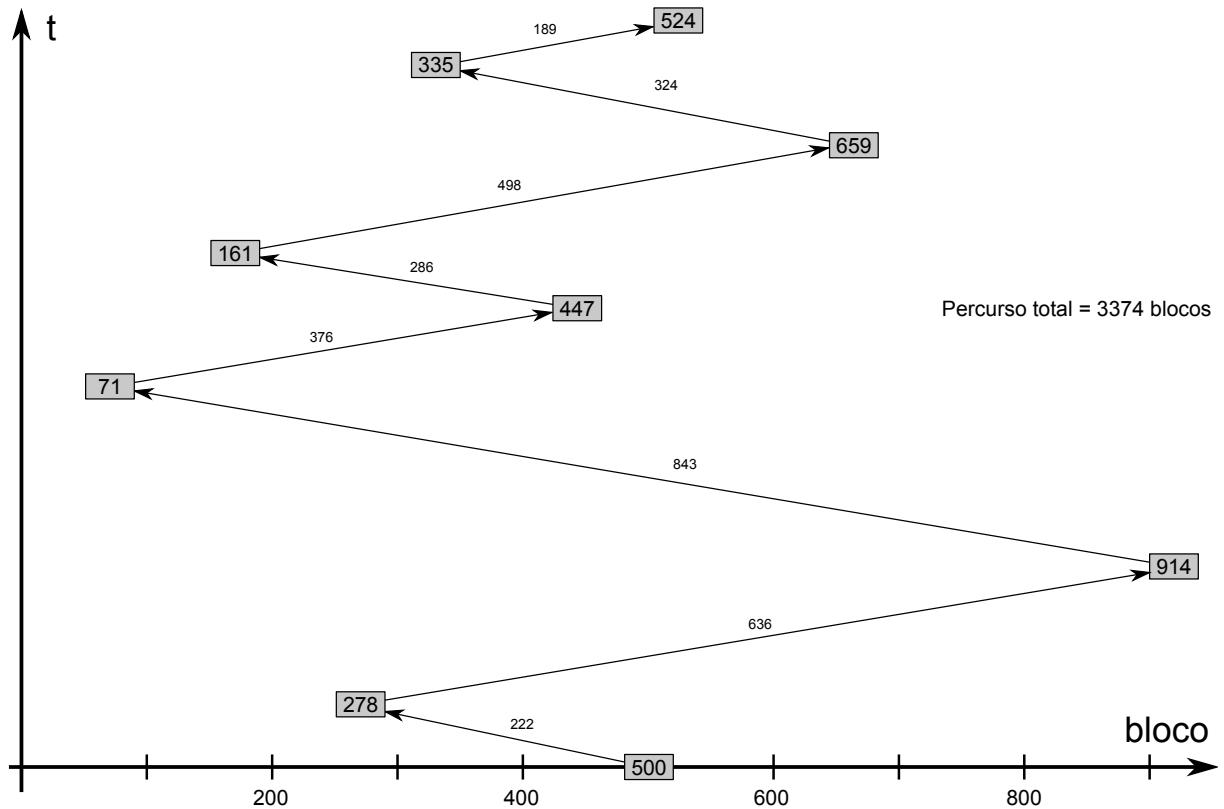


Figura 7.13: Escalonamento de disco FCFS.

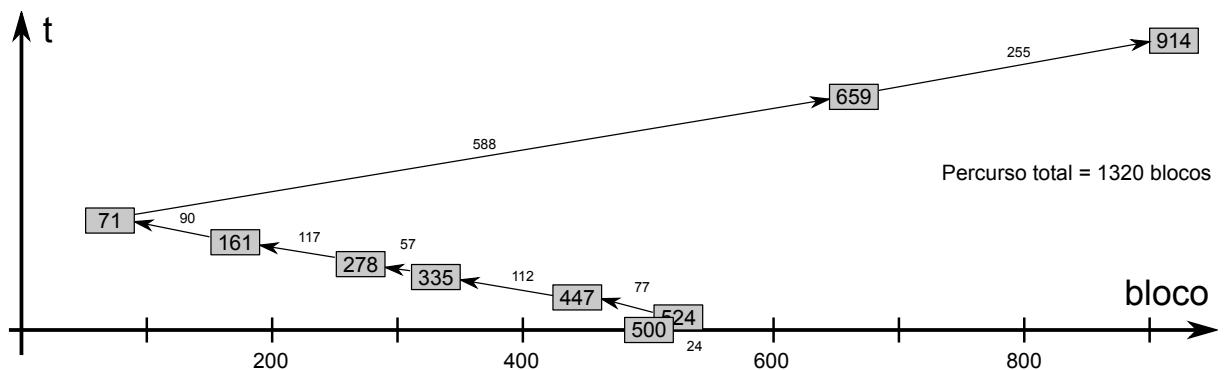


Figura 7.14: Escalonamento de disco SSTF.

Apesar de oferecer um bom desempenho, esta estratégia pode levar à inanição (*starvation*) de requisições de acesso: caso existam muitas requisições em uma determinada região do disco, pedidos de acesso a blocos longe dessa região podem ficar muito tempo esperando. Para resolver esse problema, torna-se necessário implementar uma estratégia de *envelhecimento* dos pedidos pendentes.

Elevador : este algoritmo reproduz o comportamento dos elevadores em edifícios: a cabeça de leitura se move em uma direção e atende os pedidos que encontra pela frente; após o último pedido, ela inverte seu sentido de movimento e atende os próximos pedidos. Esse movimento também é análogo ao dos limpadores de para-

brisas de um automóvel. A Figura 7.15 apresenta o comportamento deste algoritmo para a sequência de requisições de exemplo, considerando que a cabeça de leitura estava inicialmente se movendo para o começo do disco. Pode-se observar que o desempenho deste algoritmo foi melhor que os algoritmos anteriores, mas isso não ocorre sempre. A grande vantagem do algoritmo do elevador é atender os pedidos de forma mais uniforme ao longo do disco, eliminando a possibilidade de inanição de pedidos e mantendo um bom desempenho. Ele é adequado para sistemas com muitos pedidos concorrentes de acesso a disco em paralelo, como servidores de arquivos. Este algoritmo também é conhecido la literatura como SCAN ou LOOK [Silberschatz et al., 2001].

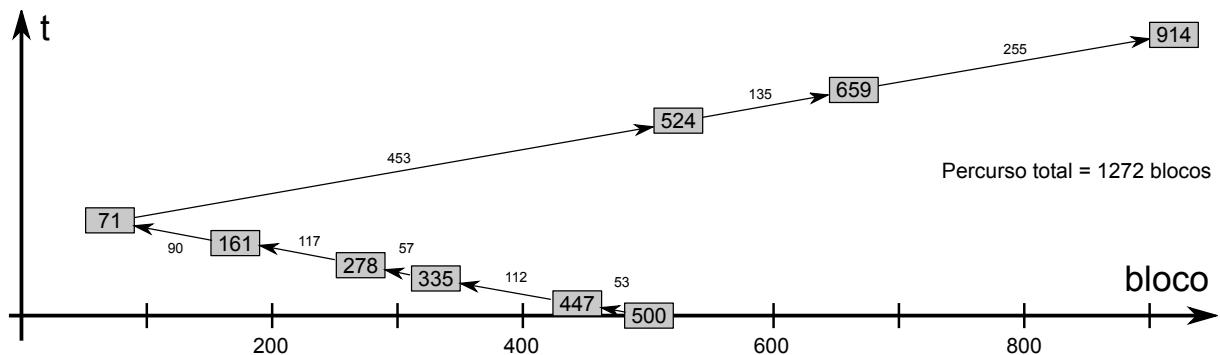


Figura 7.15: Escalonamento de disco com o algoritmo do elevador.

Elevador Circular : esta é uma variante do algoritmo do elevador, na qual a cabeça de leitura varre o disco em uma direção, atendendo os pedidos que encontrar. Ao atender o último pedido em um extremo do disco, ela retorna diretamente ao primeiro pedido no outro extremo, sem atender os pedidos intermediários, e recomeça. O nome “circular” é devido ao disco ser visto como uma lista circular de blocos. A Figura 7.16 apresenta um exemplo deste algoritmo. Apesar de seu desempenho ser pior que o do algoritmo do elevador clássico, sua maior vantagem é prover um tempo de espera mais homogêneo aos pedidos pendentes, o que é importante em servidores. Este algoritmo também é conhecido como C-SCAN ou C-LOOK.

Sistemas reais mais complexos, como Solaris, Windows e Linux, utilizam escalonadores de disco geralmente mais sofisticados. No caso do Linux, os seguintes escalonadores de disco estão presentes no núcleo, podendo ser selecionados pelo administrador do sistema [Love, 2004, 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 (baseados em memória *flash*) ou sistemas de armazenamento que façam seu próprio escalonamento, como sistemas RAID (vide Seção 7.4.5).

Deadline : este escalonador é baseado no algoritmo do elevador circular, mas associa um prazo (*deadline*) a cada requisição, para evitar problemas de inanição. Como os

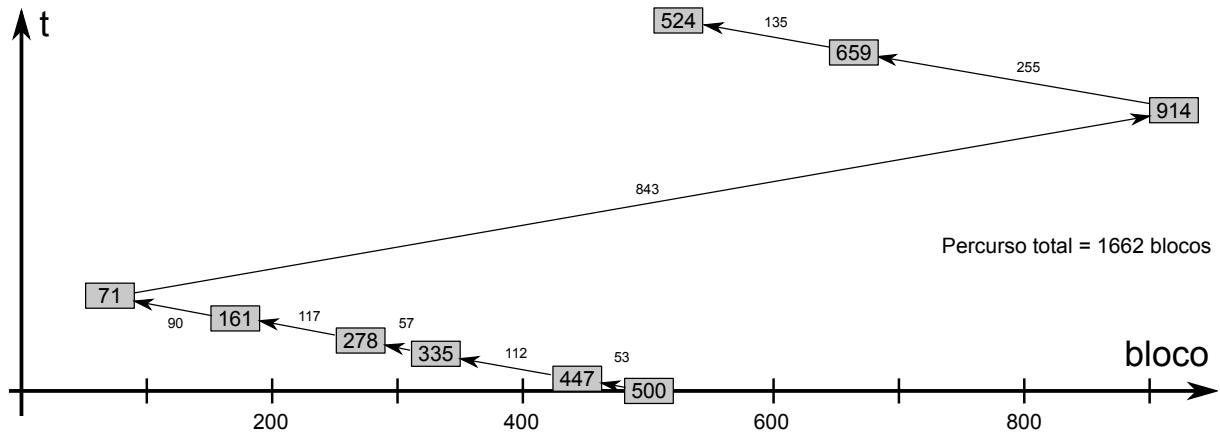


Figura 7.16: Escalonamento de disco com o algoritmo do elevador circular.

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, por isso recebem um prazo maior, de 5 segundos. O escalonador processa os pedidos usando o algoritmo do elevador, mas prioriza os pedidos cujo prazo esteja 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, o escalonador volta a tratar a fila de pedidos pendentes normalmente. Essa espera por pedidos adjacentes melhora o desempenho das operações de leitura.

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 atuais.

7.4.4 Caching de blocos

Como o disco rígido pode apresentar latências elevadas, a funcionalidade de *caching* é muito importante para o bom desempenho dos acessos ao disco. É possível fazer *caching* de leitura e de escrita. No *caching* de leitura (*read caching*), blocos de dados lidos do disco são mantidos em memória, para acelerar leituras posteriores dos mesmos. No *caching* de escrita (*write caching*, também chamado *buffering*), dados a escrever no disco são mantidos em memória para leituras posteriores, ou para concentrar várias escritas pequenas em poucas escritas maiores (e mais eficientes). Quatro estratégias de *caching* são usuais:

- *Read-behind*: esta é a política mais simples, na qual somente os dados já lidos em requisições anteriores são mantidos em cache; outros acessos aos mesmos dados serão beneficiados pelo cache;
- *Read-ahead*: nesta política, 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;
- *Write-through*: nesta política, ao atender uma requisição de escrita, uma cópia dos dados a escrever no disco é mantida em cache, para beneficiar possíveis leituras futuras desses dados;
- *Write-back*: nesta política, além de copiar os dados em cache, sua escrita efetiva no disco é adiada; esta estratégia melhora o desempenho de escrita de duas formas: por liberar mais cedo os processos que solicitam escritas (eles não precisam esperar pela escrita real no disco) e por concentrar as operações de escrita, gerando menos acessos a disco. Todavia, pode ocasionar perda de dados, caso ocorram erros de hardware ou falta de energia antes que os dados sejam efetivamente escritos no disco.

7.4.5 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 7.4.3) [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

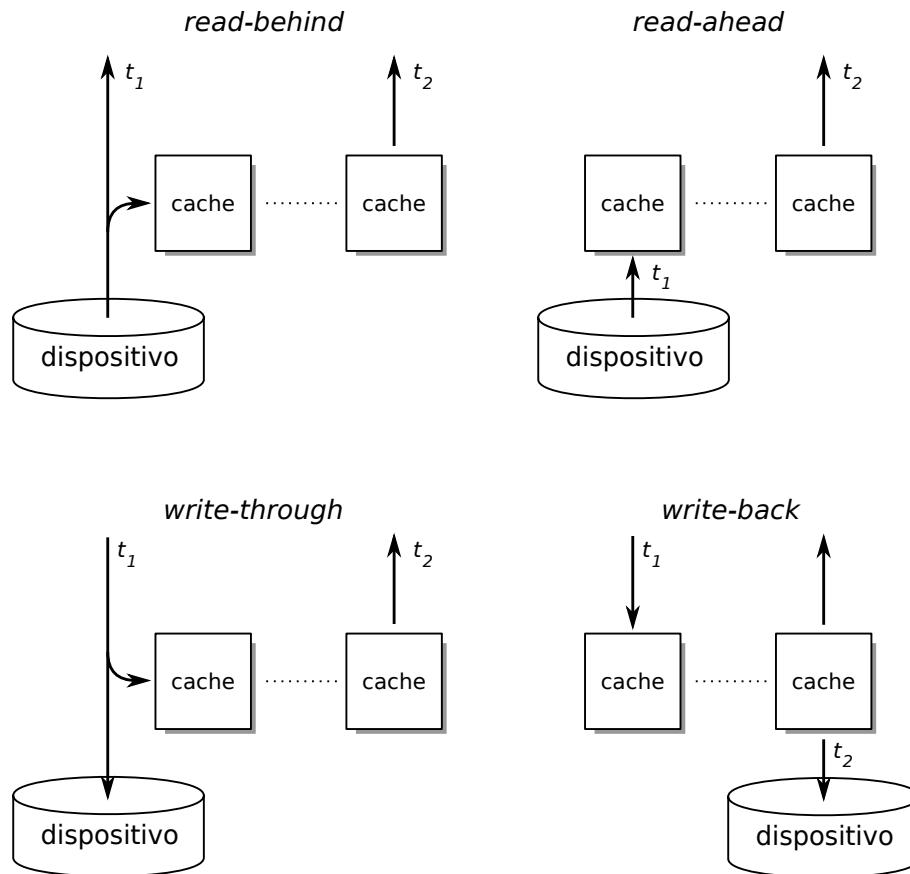


Figura 7.17: Estratégias de *caching* de blocos (t_1 e t_2 indicam dois instantes de tempo).

eles denominaram RAID – *Redundant Array of Inexpensive Disks*⁵ [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

⁵Mais recentemente alguns autores adotaram a expressão *Redundant Array of Independent Disks* para a sigla RAID, buscando evitar a subjetividade da palavra *Inexpensive* (econômico).

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, 2009]:

RAID 0 : 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. As fatias são preenchidas com os blocos usando uma estratégia *round-robin*, como mostra a Figura 7.18. O disco lógico terá como tamanho a soma dos tamanhos dos discos físicos. Esta abordagem oferece um grande ganho de desempenho em operações de leitura e escrita: usando N discos físicos, até N operações podem ser efetuadas em paralelo.

Entretanto, não há nenhuma estratégia de redundância de dados, o que torna este nível mais 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. Suas características de grande volume de dados e alto desempenho em leitura/escrita tornam esta abordagem adequada para ambientes que geram e precisam processar grandes volumes de dados temporários, como os sistemas de computação científica [Chen et al., 1994].

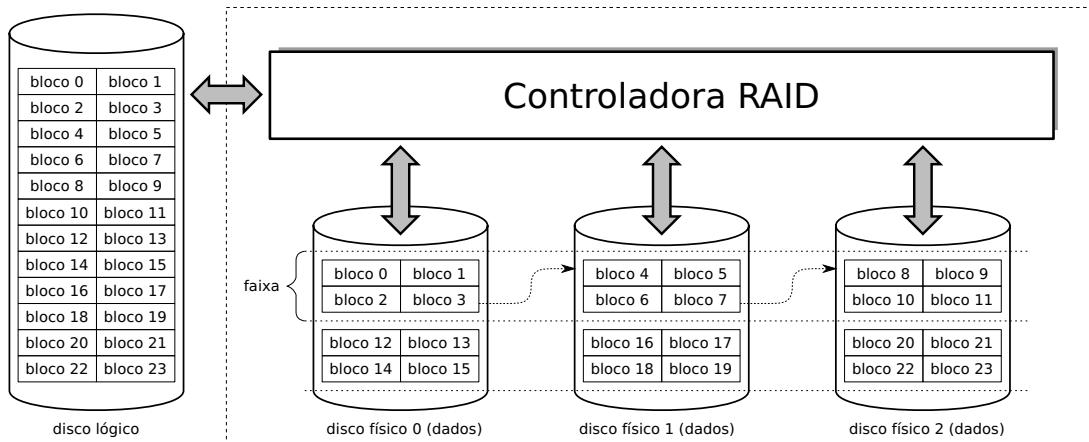
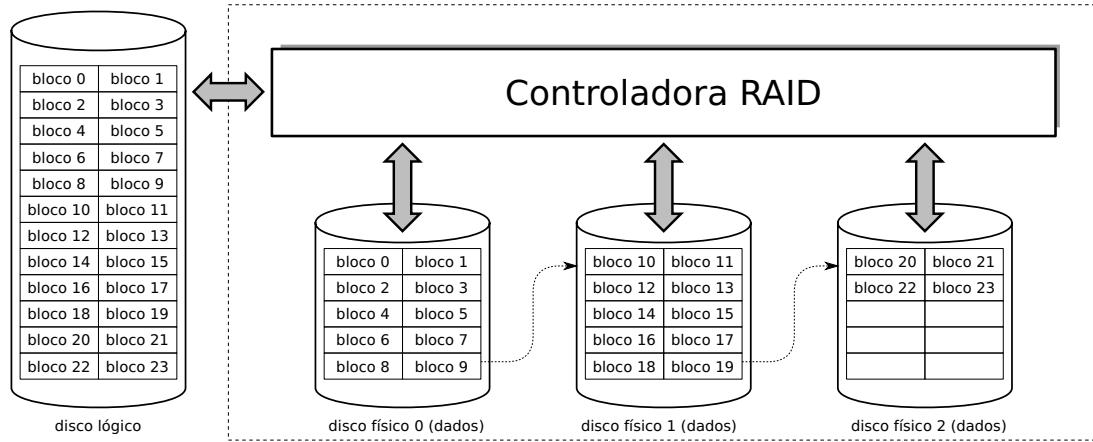


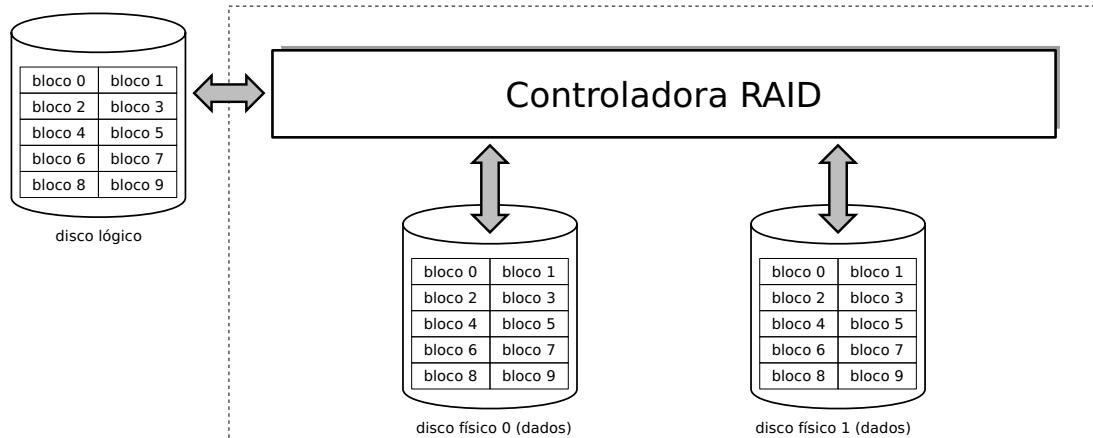
Figura 7.18: RAID nível 0 (*striping*).

Alguns sistemas RAID não dividem os discos físicos em fatias, mas apenas concatenam os espaços dos vários discos em sequência para construir o disco lógico. Essa abordagem, ilustrada na Figura 7.19, é denominada por alguns autores de *RAID 0 linear*, enquanto outros a denominam JBoD (*Just a Bunch of Disks* – apenas um grupo de discos). Como os blocos do disco lógico estão menos uniformemente espalhados sobre os discos físicos, os acessos se concentram em um disco a cada instante, levando a um menor desempenho em leituras e escritas.

Figura 7.19: RAID nível 0 (*linear*).

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 7.20 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).

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 7.20: RAID nível 1 (*mirroring*).

RAID 2 : este nível fatia os dados 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. Este nível não é usado na prática.

RAID 3 : este nível fatia os dados em bytes, que são escritos nos discos em sequência. Um disco separado contém dados de paridade, usados para a recuperação de erros. A cada leitura ou escrita, os dados do disco de paridade devem ser atualizados, o que implica na serialização dos acessos e a consequente queda de desempenho. Por esta razão, esta abordagem é raramente usada.

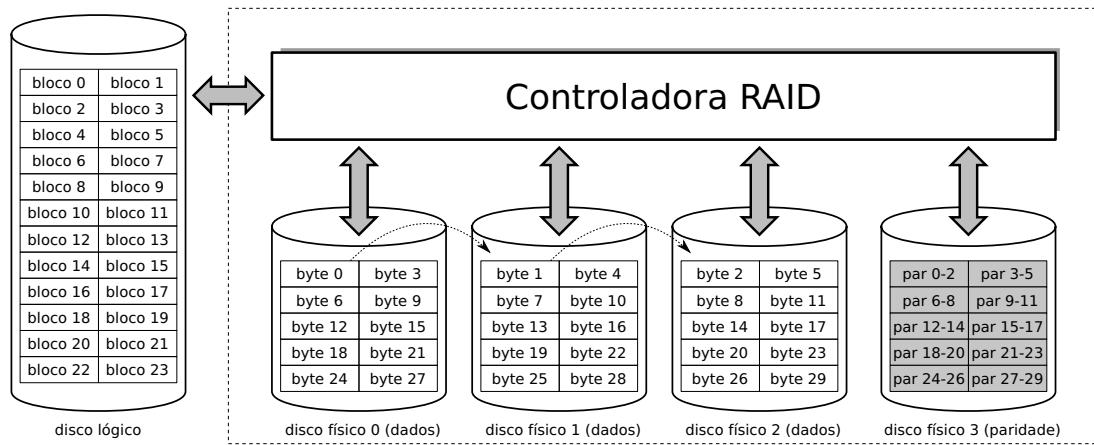


Figura 7.21: RAID nível 3.

RAID 4 : esta abordagem é similar ao RAID 3, com a diferença de que o fatiamento é feito por blocos ao invés de bytes (Figura 7.22). Ela sofre dos mesmos problemas de desempenho que o RAID 3, sendo por isso pouco usada.

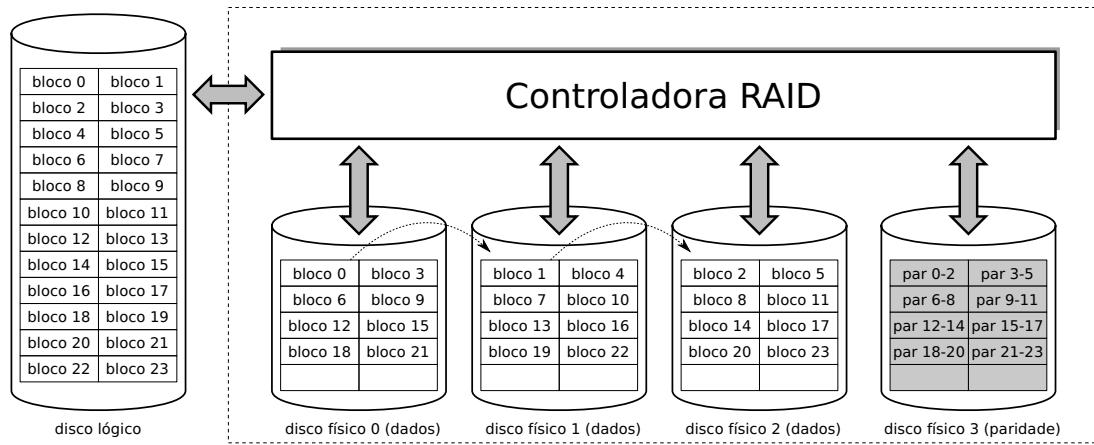


Figura 7.22: RAID nível 4.

RAID 5 : assim como a anterior, esta abordagem também armazena informações de paridade para tolerar falhas em discos. Todavia, essas informações não ficam concentradas em um único disco físico, mas são distribuídas uniformemente entre todos eles. A Figura 7.23 ilustra uma possibilidade de distribuição das informações

de paridade. Essa estratégia elimina o gargalo de desempenho no acesso aos dados de paridade. Esta é sem dúvida a abordagem de RAID mais popular, por oferecer um bom desempenho e redundância de dados com um custo menor que o espelhamento (RAID 1).

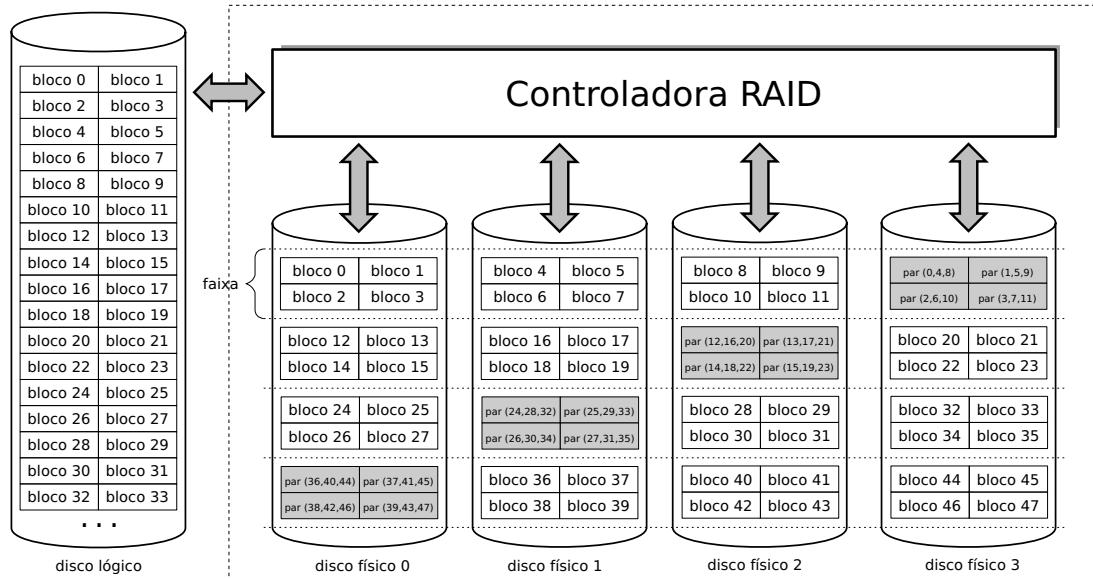


Figura 7.23: RAID nível 5.

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 adicional 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 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 3 não é necessariamente melhor que um sistema RAID 2. Uma descrição mais aprofundada dos vários níveis RAID, de suas variantes e características pode ser encontrada em [Chen et al., 1994] e [SNIA, 2009].

7.5 Interfaces de rede

network I/O (modelo em camadas)

7.6 Dispositivos USB

7.7 Interfaces de áudio

capítulo separado sobre E/S de multimídia (áudio e vídeo, codecs)?

7.8 Interface gráfica

7.9 Mouse e teclado

falar de terminal e e/s serial?

7.10 Outros tópicos

- como deve ser tratada a gerência de energia?
- tratar relógio em separado?
- ioctl
- sistemas de arquivos /dev, /proc e /sys
- major/minor numbers
- gestão de módulos: insmod, lsmod, modprobe
- acesso a barramentos: lsusb, lspci, ...

Capítulo 8

Segurança de sistemas

Este módulo trata dos principais aspectos de segurança envolvidos na construção e utilização de um sistema operacional. Inicialmente são apresentados conceitos básicos de segurança e criptografia; em seguida, são descritos aspectos conceituais e mecanismos relacionados à autenticação de usuários, controle de acesso a recursos e serviços, integridade e privacidade do sistema operacional, das aplicações e dos dados armazenados. Grande parte dos tópicos de segurança apresentados neste capítulo não são exclusivos de sistemas operacionais, mas se aplicam a sistemas de computação em geral.

8.1 Introdução

A segurança de um sistema de computação diz respeito à garantia de algumas propriedades fundamentais associadas às informações e recursos presentes nesse sistema. Por “informação”, compreende-se todos os recursos disponíveis no sistema, como registros de bancos de dados, arquivos, áreas de memória, portas de entrada/saída, conexões 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 erros de programação que possam provocar acidentes; 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 contra ameaças intencionais.

Este capítulo trata dos principais aspectos de segurança envolvidos na construção e operação de um sistema operacional. A primeira parte do capítulo apresentada conceitos básicos de segurança, como as propriedades e princípios de segurança, ameaças, vulnerabilidades e ataques típicos em sistemas operacionais, concluindo com uma descrição da infra-estrutura de segurança típica de um sistema operacional. A seguir, é apresentada uma introdução à criptografia. Na sequência, são descritos

aspectos conceituais e mecanismos relacionados à autenticação de usuários, controle de acesso a recursos e integridade do sistema. Também são apresentados os principais conceitos relativos ao registro de dados de operação para fins de auditoria. Grande parte dos tópicos de segurança apresentados neste capítulo não são exclusivos de sistemas operacionais, mas se aplicam a sistemas de computação em geral.

8.2 Conceitos básicos

Nesta seção são apresentados alguns conceitos fundamentais, importantes para o estudo da segurança de sistemas computacionais. Em particular, são enumeradas as propriedades que caracterizam a segurança de um sistema, são definidos os principais termos em uso na área, e são apresentados os principais elementos que compõe a arquitetura de segurança de um sistema.

8.2.1 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-repudiação*.

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. Dessa forma, os danos provocados por erros ou ações maliciosas intencionais serão minimizados.

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.

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).

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.

8.2.2 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.

8.2.3 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:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int i, j, buffer[20], k; // declara buffer[0] a buffer[19]
5
6 int main()
7 {
8     i = j = k = 0 ;
9
10    for (i = 0; i<= 20; i++) // usa buffer[0] a buffer[20] <-- erro!
11        buffer[i] = random() ;
12
13    printf ("i: %d\nj: %d\nk: %d\n", i, j, k) ;
14
15    return(0);
16 }
```

A execução desse código gera o seguinte resultado:

```

1 host:~> cc buffer-overflow.c -o buffer-overflow
2 host:~> buffer-overflow
3 i: 21
4 j: 35005211
5 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 sucessiva¹, 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].

8.2.4 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 8.1:

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;

¹As 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.

Fabricação : consiste em produzir informações falsas ou introduzir módulos ou componentes maliciosos no sistema; é um ataque à autenticidade.

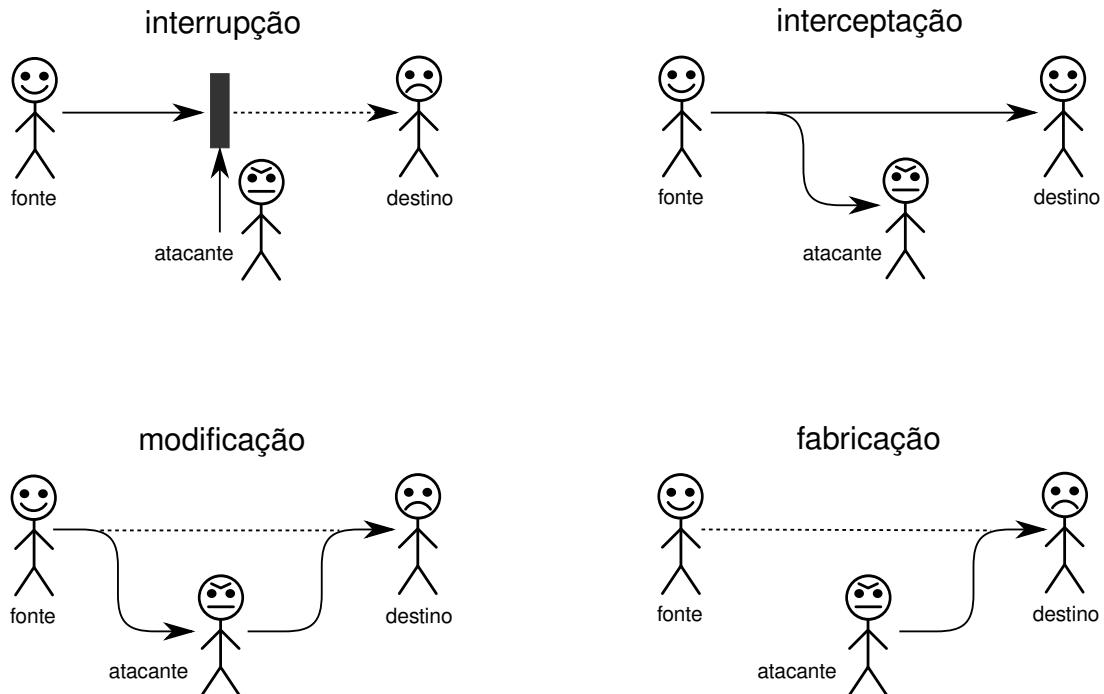


Figura 8.1: Tipos básicos de ataques (inspirado em [Pfleeger and Pfleeger, 2006]).

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*.

A maioria dos ataques a sistemas operacionais visa aumentar o poder do atacante dentro do sistema, o que é denominado *elevação de privilégios* (*privilege escalation*). Esses ataques geralmente exploram vulnerabilidades em programas do sistema (que executam com mais privilégios), ou do próprio núcleo, através de chamadas de sistema, para receber 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.

8.2.5 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], como:

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 replicação². 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

²De forma análoga, um vírus biológico precisa de uma célula hospedeira, pois usa o material celular como suporte para sua existência e replicação.

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 auto-execução 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 criptografia (Seção 8.3) resolve parcialmente esse problema, embora um *sniffer* na máquina local possa capturar os dados antes que sejam cifrados, ou depois de decifrados.

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*.

Deve-se ter em mente que há na mídia e na literatura muita confusão em relação à nomenclatura de *malwares*; além disso, muitos *malwares* têm várias funcionalidades e se encaixam em mais de uma categoria. Esta seção teve como objetivo dar uma definição tecnicamente precisa de cada categoria, sem a preocupação de mapear os exemplos reais nessas categorias.

8.2.6 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 comportamento suspeitos.

A Figura 8.2 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.

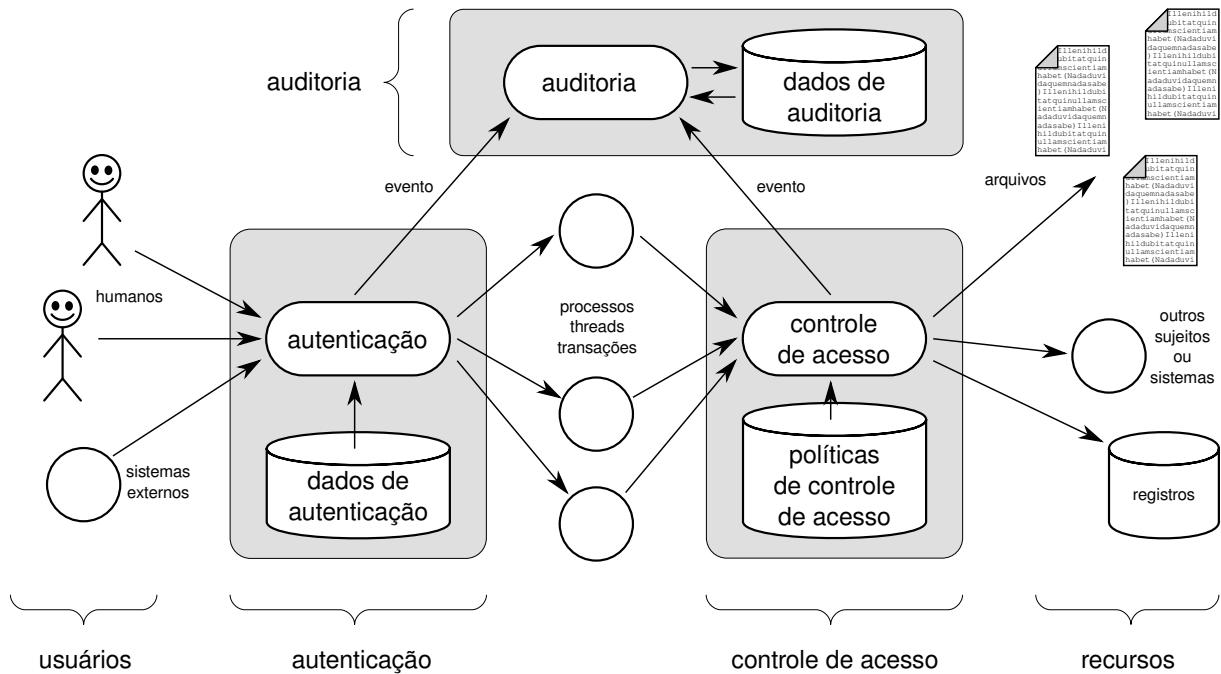


Figura 8.2: Base de computação confiável de um sistema operacional.

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 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 anti-vírus, 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.).

8.3 Fundamentos 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. 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.

Alguns conceitos fundamentais para compreender as técnicas criptográficas são: o *texto aberto*, que é a mensagem ou informação a ocultar; o *texto cifrado*, que é a informação codificada; o *cifrador*, mecanismo responsável por cifrar/decifrar as informações, e as *chaves*, necessárias para poder cifrar ou decifrar as informações [Menezes et al., 1996].

8.3.1 Cifragem e decifragem

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 vqfq qu iga pg tcku rctc q cvcswg
mensagem cifrada com $k = 3$:	Uhxqlu wrgrv rv jhqhudlv sdud r dwdtih

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 denominado *chave criptográfica*. Caso essa chave não seja conhecida, é possível tentar “quebrar” a mensagem cifrada testando todas as chaves possíveis, o que é conhecido como análise exaustiva ou de “força bruta”. No caso do cifrador de César a análise exaustiva é trivial, pois há somente 26 valores possíveis para a chave k .

O número de chaves possíveis para um algoritmo de cifragem é conhecido como o seu *espaço de chaves*. De acordo com princípios enunciados pelo criptógrafo Auguste Kerckhoffs em 1883, o segredo de uma técnica criptográfica não deve residir no algoritmo em si, mas no espaço de chaves que ele provê. Seguindo esses princípios, a criptografia moderna se baseia em algoritmos públicos, bem avaliados pela comunidade científica, para os quais o espaço de chaves é muito grande, tornando inviável qualquer análise exaustiva. Por exemplo, o algoritmo de criptografia AES (*Advanced Encryption Standard*) adotado como padrão pelo governo americano, usando

chaves de 128 bits, 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 analisar um bilhão de chaves por segundo, ainda assim seriam necessários 10 sextilhões de anos para testar todas as chaves possíveis!

No restante do texto, a operação de cifragem de um conteúdo x usando uma chave k será representada por $\{x\}_k$ e a decifragem de um conteúdo x usando uma chave k será representada por $\{x\}_k^{-1}$.

8.3.2 Criptografia simétrica e assimétrica

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 usada para cifrar a informação deve ser usada para decifrá-la. Essa propriedade pode ser expressa em termos matemáticos:

$$\{\{x\}_k\}_{k'}^{-1} = x \iff k' = k$$

O cifrador de César é um exemplo típico de cifrador simétrico: se usarmos $k = 2$ para cifrar um texto, teremos de usar $k = 2$ para decifrá-lo. Os algoritmos simétricos mais conhecidos e usados atualmente são o DES (*Data Encryption Standard*) e o AES (*Advanced Encryption Standard*).

Os algoritmos simétricos são muito úteis para a cifragem de dados em um sistema local, como documentos ou arquivos em um disco rígido. Todavia, se a informação cifrada tiver de ser enviada a outro usuário, a chave criptográfica usada terá de ser informada a ele de alguma forma segura (de forma a preservar seu segredo). A Figura 8.3 ilustra o funcionamento básico de um sistema de criptografia simétrica.

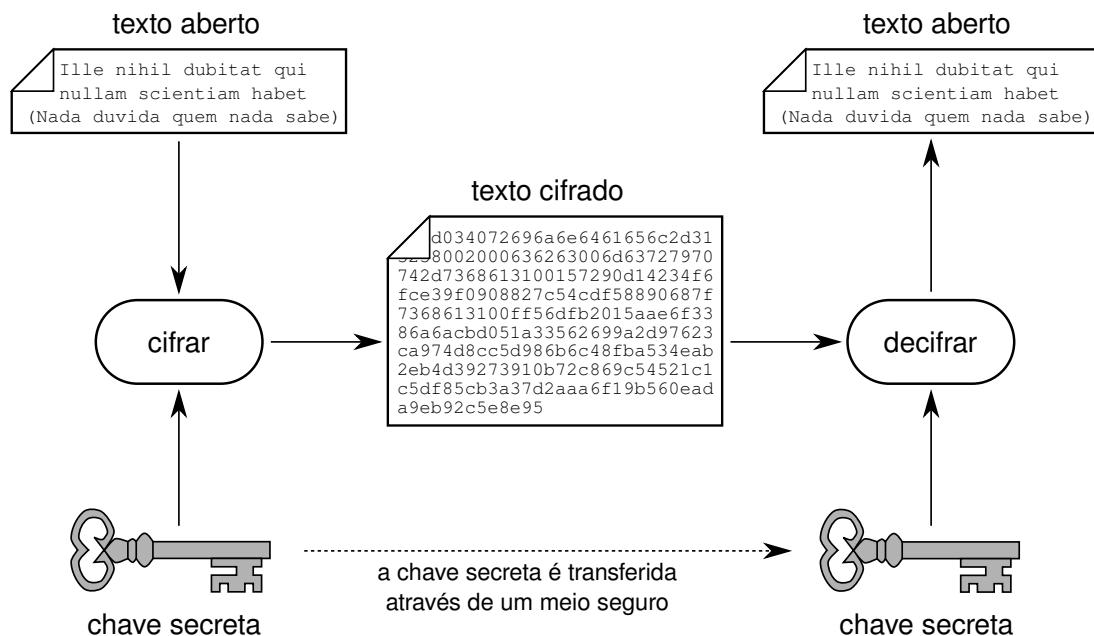


Figura 8.3: Criptografia simétrica.

Por outro lado, 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 uso de uma chave pública só poderá ser decifrada através da chave privada correspondente, e vice-versa. Considerando um usuário u com suas chaves pública $kp(u)$ e privada $kv(u)$, temos:

$$\begin{aligned}\{\{x\}_{kp(u)}\}_k^{-1} &= x \iff k = kv(u) \\ \{\{x\}_{kv(u)}\}_k^{-1} &= x \iff k = kp(u)\end{aligned}$$

Essas duas chaves estão fortemente relacionadas: para cada chave pública há uma única chave privada correspondente, e vice-versa. Todavia, não é possível calcular uma das chaves a partir da outra. 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. Alguns algoritmos assimétricos bem conhecidos são o RSA (*Rivest-Shamir-Adleman*) e o algoritmo de *Diffie-Hellman*. A Figura 8.4 ilustra o funcionamento da criptografia assimétrica.

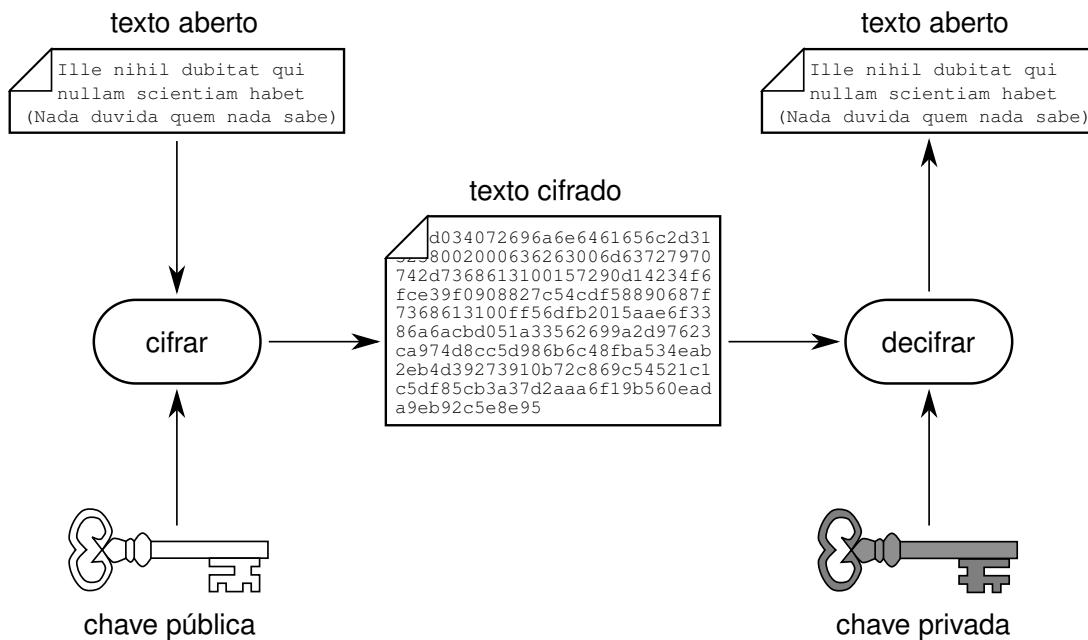


Figura 8.4: Criptografia assimétrica.

Um exemplo de uso da criptografia assimétrica é mostrado na Figura 8.5. Nele, a usuária Alice deseja enviar um documento cifrado ao usuário Beto³. Para tal, Alice busca a chave pública de Beto previamente divulgada em um chaveiro público (que

³Textos em inglês habitualmente usam os nomes Alice, Bob, Carol e Dave para explicar algoritmos e protocolos criptográficos, em substituição às letras A, B, C e D. Neste texto usaremos a mesma abordagem, mas com nomes em português.

pode ser um servidor Web, por exemplo) e a usa para cifrar o documento que será enviado a Beto. Somente Beto poderá decifrar esse documento, pois só ele possui a chave privada correspondente à chave pública usada para cífrá-lo. Outros usuários poderão até ter acesso ao documento cifrado, mas não conseguirão decifrá-lo.

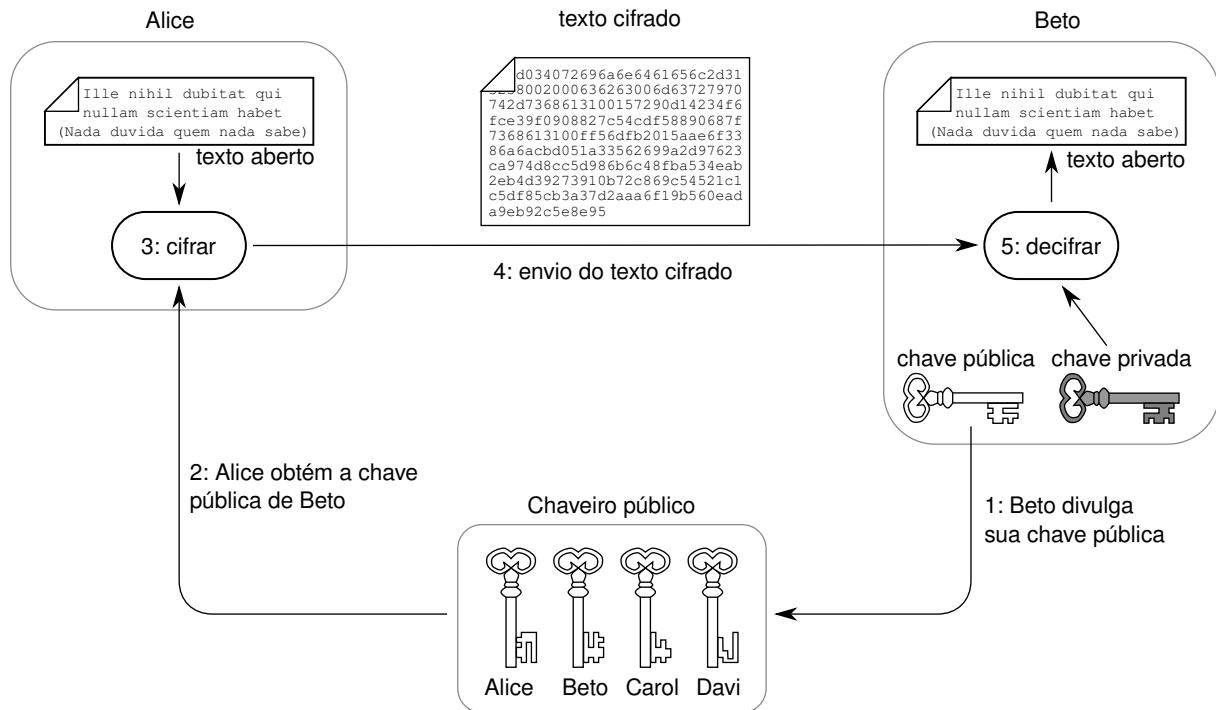


Figura 8.5: Exemplo de uso da criptografia assimétrica.

A criptografia assimétrica também pode ser usada para identificar a autoria de um documento. Por exemplo, se Alice criar um documento e cífrá-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 cífrá-lo. Esse mecanismo é usado na criação das *assinaturas digitais* (Seção 8.3.4).

Embora mais versáteis, os algoritmos de cifragem assimétricos costumam exigir muito mais processamento que os algoritmos simétricos equivalentes. Por isso, muitas vezes ambos são usados em associação. 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 negociar 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.

8.3.3 Resumo criptográfico

Um *resumo criptográfico* (*cryptographic hash*) [Menezes et al., 1996] é uma função que gera uma sequência de bytes de tamanho pequeno e fixo (algumas dezenas ou centenas

de bytes) a partir de um conjunto de dados 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 simples, mas o cálculo de sua inversa ($x = f^{-1}(y)$) é impossível ou inviável em termos computacionais. Um exemplo clássico de função unidirecional é a fatoração do produto de dois números primos grandes: considere 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, fatorizar y para obter de volta os primos p e q pode ser computacionalmente inviável, se y tiver muitos dígitos⁴.

Idealmente, uma função de resumo criptográfico deve gerar sempre a mesma saída para a mesma entrada, e saídas diferentes para entradas diferentes. No entanto, como o número de bytes do resumo é pequeno, podem ocorrer *colisões*. Uma colisão ocorre quando duas entradas distintas x e x' geram o mesmo valor de resumo ($\text{hash}(x) = \text{hash}(x')$ para $x \neq x'$). Obviamente, bons algoritmos de resumo buscam minimizar essa possibilidade. Outras propriedades desejáveis dos resumos criptográficos são o *espalhamento*, em que uma modificação em um trecho específico dos dados de entrada gera modificações em partes diversas do resumo, e a *sensibilidade*, em que uma pequena modificação nos dados de entrada pode gerar grandes mudanças no resumo.

Os algoritmos de resumo criptográfico mais conhecidos e utilizados atualmente são o MD5 e o SHA1 [Menezes et al., 1996]. No Linux, os comandos `md5sum` e `sha1sum` permitem calcular respectivamente os resumos MD5 e SHA1 de arquivos comuns:

```

1 maziero:~> md5sum *
2 62ec3f9ff87f4409925a582120a40131  header.tex
3 0920785a312bd88668930f761de740bf  main.pdf
4 45acbba4b57317f3395c011fb43d68d  main.tex
5 6c332adb037265a2019077e09a024d0c  main.tex~
6
7 maziero:~> sha1sum *
8 742c437692369ace4bf0661a8fe5741f03ecb31a  header.tex
9 9f9f52f48b75fd2f12fa297bdd5e1b13769a3139  main.pdf
10 d6973a71e5c30d0c05d762e9bc26bb073d377a0b  main.tex
11 cf1670f22910da3b9abf06821e44b4ad7efb5460  main.tex~

```

8.3.4 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,

⁴Em 2005, um grupo de pesquisadores alemães fatorizou um inteiro com 200 dígitos, usando 80 processadores Opteron calculando durante mais de cinco meses.

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

$$s(d, u) = \{\text{hash}(d)\}_{kv(u)}$$

onde $\text{hash}(x)$ 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 $r' = \text{hash}(d)$ e compará-lo com o resumo obtido da assinatura, decifrada usando a chave pública de u ($r'' = \{s\}_{kp(u)}^{-1}$). 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 8.6 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. Beto obtém o documento assinado por Alice ($[d', s']$, com $d' = d$ e $s' = s$ se ambos estiverem íntegros);
6. Beto recalcula o resumo digital $r' = \text{hash}(d')$ do documento, usando o mesmo algoritmo empregado por Alice;
7. Beto 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. Beto 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.

8.3.5 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

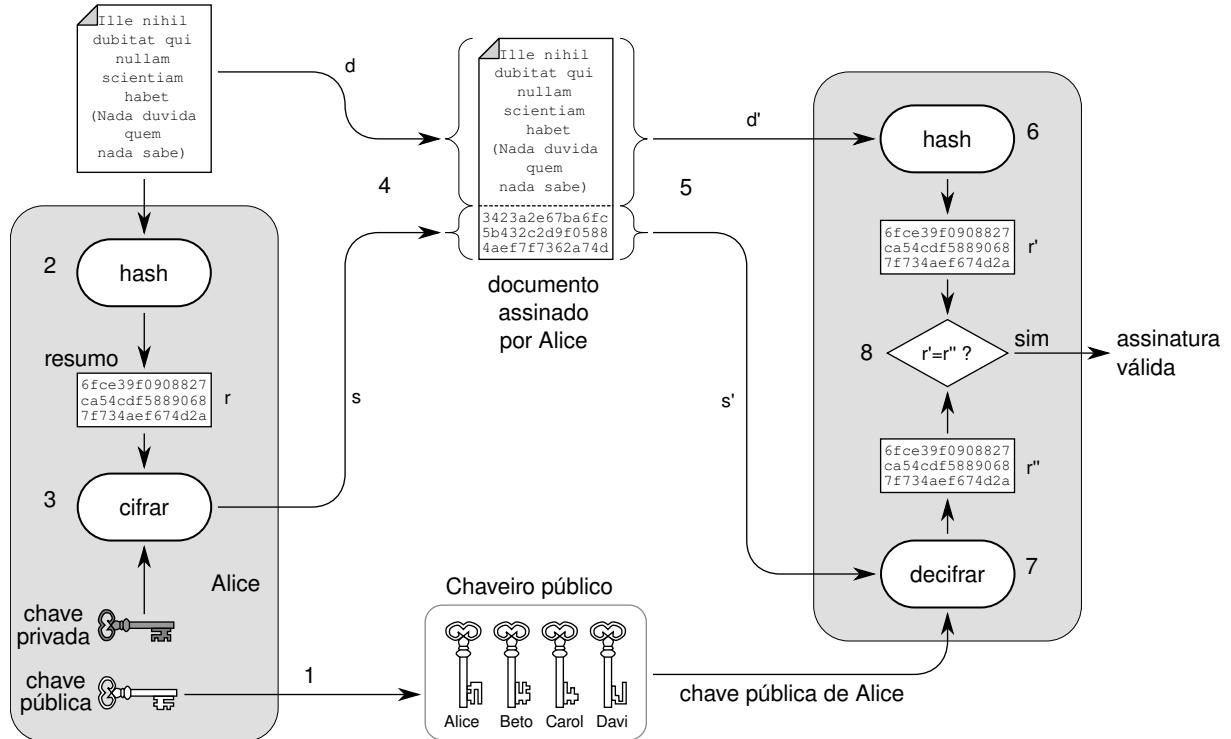


Figura 8.6: Assinatura e verificação de uma assinatura digital.

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]:

- A chave pública do proprietário do certificado;
- Identidade do proprietário do certificado (nome, endereço, e-mail, URL, número IP e/ou outras informações que permitam identificá-lo unicamente)⁵;
- Outras informações, como período de validade do certificado, algoritmos de criptografia e resumos preferidos ou suportados, etc.;
- Uma ou mais assinaturas digitais do conteúdo, emitidas por entidades consideradas confiáveis pelos usuários dos certificados.

Dessa forma, um certificado digital “amarra” uma identidade a uma chave pública. Para verificar a validade de um certificado, basta usar as chaves públicas das entidades que o assinaram. Existem vários tipos de certificados digitais com seus formatos e conteúdos próprios, sendo os certificados PGP e X.509 aqueles mais difundidos [Mollin, 2000].

⁵Deve-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.

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 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 de ordem mais elevada (denominada AC raiz) assina os certificados de outras ACs, e assim sucessivamente, até chegar aos certificados dos usuários e demais entidades do sistema. Uma estrutura de certificação se chama *Infra-estrutura 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].

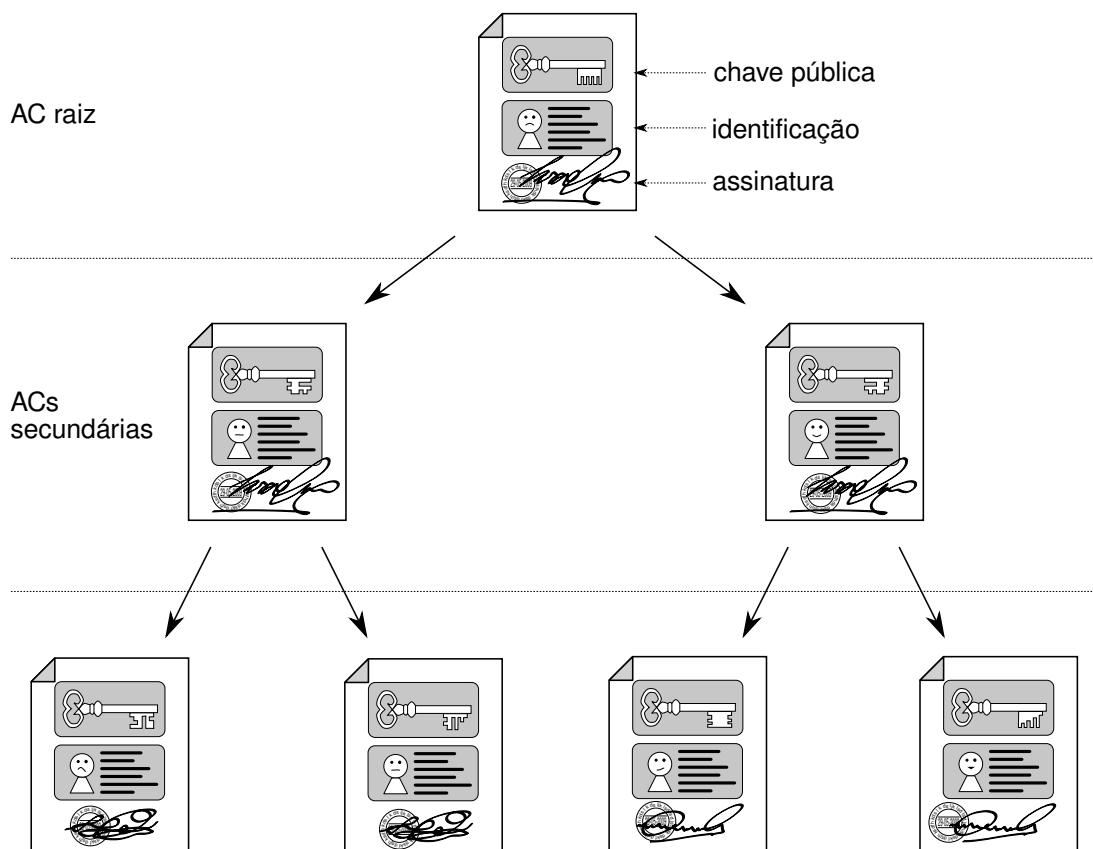


Figura 8.7: Infra-estrutura de chaves públicas hierárquica.

8.4 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 nesta seção.

Inicialmente, a autenticação visava apenas identificar usuários, para garantir que somente usuários devidamente credenciados teriam acesso ao sistema. Atualmente, em muitas circunstâncias também é necessário o oposto, ou seja, identificar o sistema para o usuário, ou mesmo sistemas entre si. 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 e não foi corrompido por algum conteúdo malicioso.

8.4.1 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 públicos 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.

8.4.2 Técnicas 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 multi-fator*. 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.

8.4.3 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 8.3.3).

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 ($r = \text{hash}(s)$), e o armazena. Mais tarde, quando esse usuário solicitar sua autenticação, ele informará uma senha s' ; o sistema então calculará novamente seu resumo $r' = \text{hash}(s')$ 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.

8.4.4 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.
- 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 $s_0, s_1, s_2, \dots, s_n$ com s_0 aleatório e $s_i = \text{hash}(s_{i-1}) \forall i > 0$, sendo $\text{hash}(x)$ uma função de resumo criptográfico conhecida. 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 $\text{hash}(s_{n-1})$ com o valor de s_n previamente informado: se forem iguais, o cliente está autenticado e ambos podem descartar s_n . O servidor

armazena s_{n-1} para validar a próxima autenticação, e assim sucessivamente. Um intruso que conseguir capturar uma senha s_i não poderá usá-la mais tarde, pois não conseguirá calcular $s_{i-1} = \text{hash}^{-1}(s_i)$.

- 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 8.8 são frequentemente usados em sistemas de *Internet Banking*.



Figura 8.8: Um dispositivo gerador de senhas descartáveis.

8.4.5 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 8.9 apresenta alguns exemplos de características biométricas empregadas nos sistemas atuais.

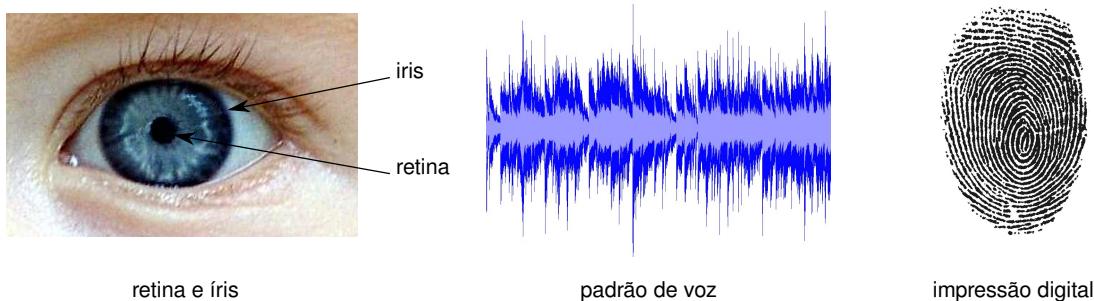


Figura 8.9: Exemplo de características biométricas.

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 pode funcionar de dois modos: 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 8.10 mostra os principais elementos de um sistema biométrico típico.

8.4.6 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*.

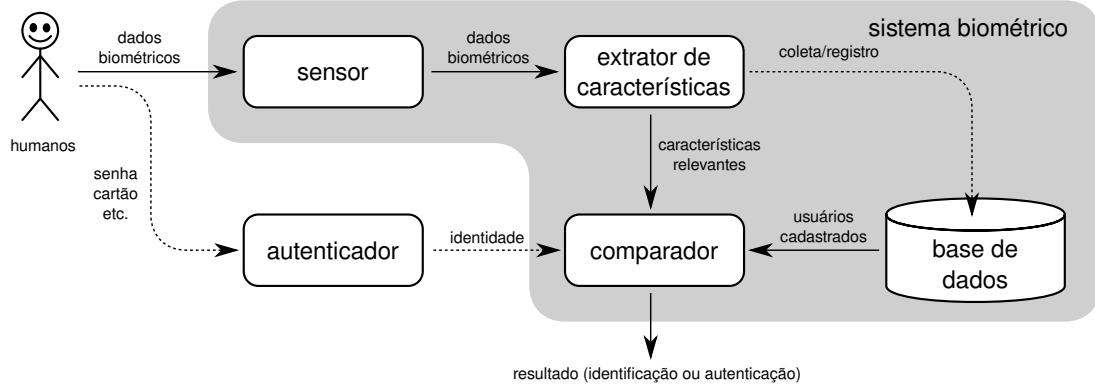


Figura 8.10: Um sistema biométrico típico.

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* ($r = hash(s + d)$). O servidor executa a mesma operação de seu lado, usando o valor do segredo armazenado localmente (s') e compara o resultado obtido $r' = hash(s' + d)$ 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 8.11 apresenta os passos desse algoritmo.

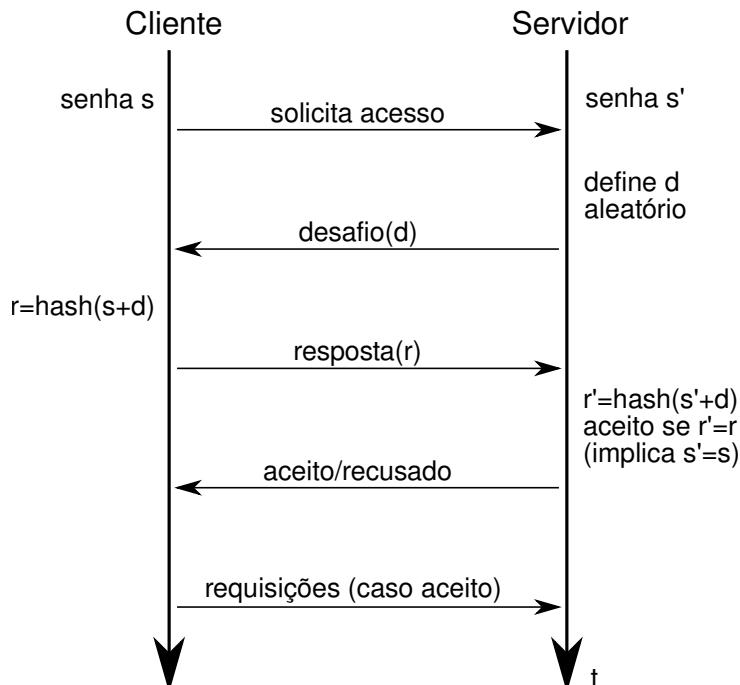


Figura 8.11: Autenticação por desafio-resposta.

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.

8.4.7 Certificados de autenticação

Uma forma cada vez mais frequente de autenticação envolve o uso de *certificados digitais*. Conforme apresentado na Seção 8.3.5, 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.

8.4.8 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 8.12, 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).

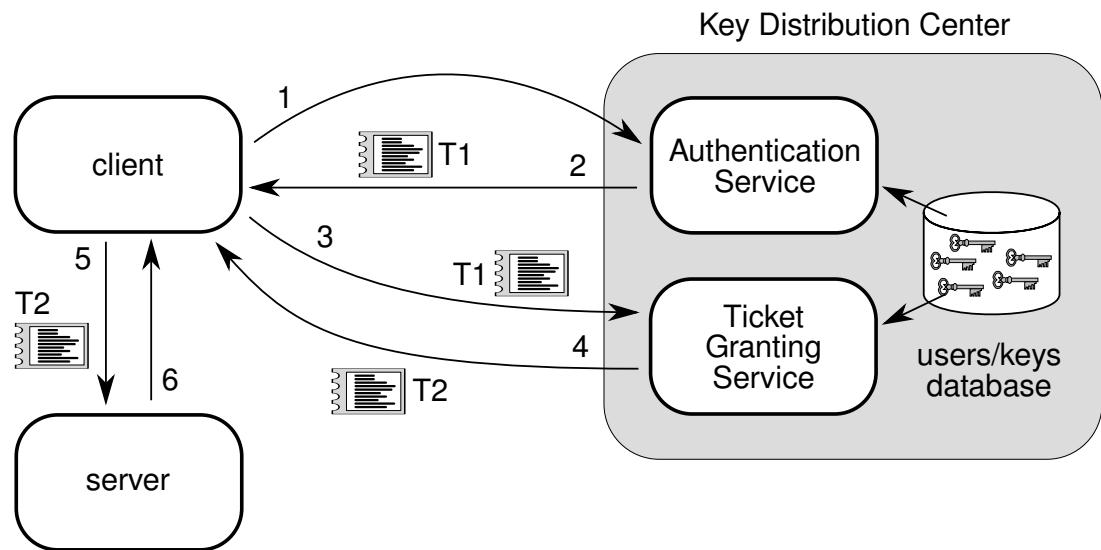


Figura 8.12: Visão geral do serviço Kerberos.

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]:

- 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:

$$m_1 = [c \ tgs \ ts \ n_1]$$

- 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 abri-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 :

$$m_4 = [\{k_{c-s} \ n_2\}_{k_{c-tgs}} \ T_{c-s}] \quad \text{onde } T_{c-s} = \{c \ tv \ k_{c-s}\}_{k_s}$$

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].

8.4.9 Infra-estruturas 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 infra-estruturas 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 infra-estrutura de autenticação é apresentada na Figura 8.13. Nela, os vários mecanismos disponíveis de autenticação são oferecidos às aplicações através de uma interface de programação (API) padronizada. As principais infra-estruturas 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 8.5.1);

NSS (*Name Services Switch*): infra-estrutura 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.

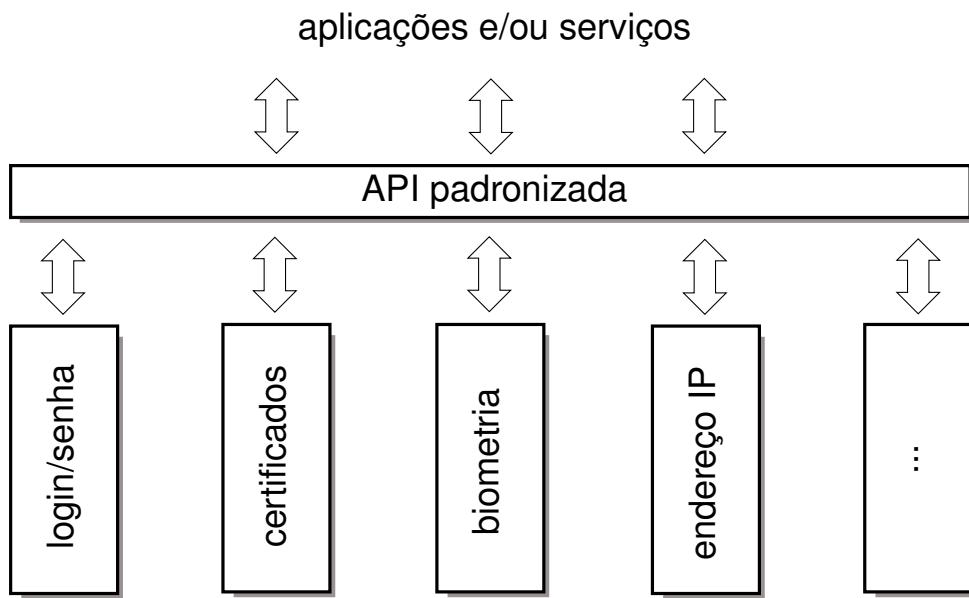


Figura 8.13: Estrutura genérica de uma infra-estrutura de autenticação.

8.5 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. Há alguns conceitos importantes para a compreensão do controle de acesso, como políticas, modelos e mecanismos. Esses conceitos serão estudados nesta seção.

Em controle de acesso, é habitual classificar as entidades de um sistema em dois grupos: os *sujeitos* e os *objetos*. Sujeitos são todas aquelas entidades que exercem um papel ativo 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. Objetos são as entidades passivas utilizadas pelos sujeitos, como arquivos, áreas de memória ou registros em um banco de dados. 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). Tanto sujeitos quanto objetos podem ser organizados em grupos e hierarquias, para facilitar a gerência da segurança.

8.5.1 Políticas, modelos e mecanismos de controle de acesso

Uma *política de controle de acesso* é uma visão abstrata das possibilidades 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:

- Médicos podem consultar os prontuários de seus pacientes;
- Médicos podem modificar os prontuários de seus pacientes enquanto estiverem internados;
- O supervisor geral pode consultar os prontuários de todos os pacientes;
- Enfermeiros podem consultar apenas os prontuários dos pacientes de sua seção e somente durante seu período de turno;
- Assistentes não podem consultar prontuários;
- Prontuários de pacientes de planos de saúde privados podem ser consultados pelo responsável pelo respectivo plano de saúde no hospital;
- 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 tempo 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, di Vimercati et al., 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.

8.5.2 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 si⁶.

⁶Muitas 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].

	<i>file</i> ₁	<i>file</i> ₂	<i>program</i> ₁	<i>socket</i> ₁
Alice	<i>read</i> <i>write</i> <i>remove</i>	<i>read</i> <i>write</i>	<i>execute</i>	<i>write</i>
Beto	<i>read</i> <i>write</i>	<i>read</i> <i>write</i> <i>remove</i>	<i>read</i>	
Carol		<i>read</i>	<i>execute</i>	<i>read</i> <i>write</i>
Davi	<i>read</i>	<i>append</i>	<i>read</i>	<i>read</i> <i>append</i>

Tabela 8.1: Uma matriz de controle de acesso

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 $\mathbb{S} = \{s_1, s_2, \dots, s_m\}$, um conjunto de objetos $\mathbb{O} = \{o_1, o_2, \dots, o_n\}$ e um conjunto de ações possíveis sobre os objetos $\mathbb{A} = \{a_1, a_2, \dots, a_p\}$, cada elemento M_{ij} da matriz de controle de acesso é um sub-conjunto (que pode ser vazio) do conjunto de ações possíveis, que define as ações que $s_i \in \mathbb{S}$ pode efetuar sobre $o_j \in \mathbb{O}$:

$$\forall s_i \in \mathbb{S}, \forall o_j \in \mathbb{O}, M_{ij} \subseteq \mathbb{A}$$

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 8.1.

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 8.2.

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.

	<i>file</i> ₁	<i>file</i> ₂	<i>program</i> ₁	<i>socket</i> ₁
Alice	<i>read</i> <i>write</i> <i>remove</i> <i>owner</i>	<i>read</i> <i>write</i>	<i>execute</i>	<i>write</i>
Beto	<i>read</i> <i>write</i>	<i>read</i> <i>write</i> <i>remove</i> <i>owner</i>	<i>read</i> <i>owner</i>	
Carol		<i>read</i>	<i>execute</i>	<i>read</i> <i>write</i>
Davi	<i>read</i>	<i>write</i>	<i>read</i>	<i>read</i> <i>write</i> <i>owner</i>

Tabela 8.2: Uma matriz de controle de acesso com política administrativa

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 8.3 mostra como ficaria a matriz de controle de acesso da Tabela 8.2 sob a forma de uma tabela de autorizações.

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 8.2 seriam:

$$\begin{aligned}
 ACL(file_1) &= \{ \quad Alice : (read, write, remove, owner), \\
 &\quad Beto : (read, write), \\
 &\quad Davi : (read) \} \\
 ACL(file_2) &= \{ \quad Alice : (read, write), \\
 &\quad Beto : (read, write, remove, owner), \\
 &\quad Carol : (read), \\
 &\quad Davi : (write) \}
 \end{aligned}$$

Sujeito	Objeto	Ação
Alice	$file_1$	<i>read</i>
Alice	$file_1$	<i>write</i>
Alice	$file_1$	<i>remove</i>
Alice	$file_1$	<i>owner</i>
Alice	$file_2$	<i>read</i>
Alice	$file_2$	<i>write</i>
Alice	$program_1$	<i>execute</i>
Alice	$socket_1$	<i>write</i>
Beto	$file_1$	<i>read</i>
Beto	$file_1$	<i>write</i>
Beto	$file_2$	<i>read</i>
Beto	$file_2$	<i>write</i>
Beto	$file_2$	<i>remove</i>
Beto	$file_2$	<i>owner</i>
Beto	$program_1$	<i>read</i>
Beto	$socket_1$	<i>owner</i>
Carol	$file_2$	<i>read</i>
Carol	$program_1$	<i>execute</i>
Carol	$socket_1$	<i>read</i>
Carol	$socket_1$	<i>write</i>
Davi	$file_1$	<i>read</i>
Davi	$file_2$	<i>write</i>
Davi	$program_1$	<i>read</i>
Davi	$socket_1$	<i>read</i>
Davi	$socket_1$	<i>write</i>
Davi	$socket_1$	<i>owner</i>

Tabela 8.3: Tabela de autorizações

$$\begin{aligned}ACL(program_1) &= \{ Alice : (execute), \\&\quad Beto : (read, owner), \\&\quad Carol : (execute), \\&\quad Davi : (read) \} \\ACL(socket_1) &= \{ Alice : (write), \\&\quad Carol : (read, write), \\&\quad Davi : (read, write, owner) \}\end{aligned}$$

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.

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 8.2 seriam:

$$\begin{aligned}CL(Alice) &= \{ file_1 : (read, write, remove, owner), \\&\quad file_2 : (read, write), \\&\quad program_1 : (execute), \\&\quad socket_1 : (write) \} \\CL(Beto) &= \{ file_1 : (read, write), \\&\quad file_2 : (read, write, remove, owner), \\&\quad program_1 : (read, owner) \} \\CL(Carol) &= \{ file_2 : (read), \\&\quad program_1 : (execute), \\&\quad socket_1 : (read, write) \} \\CL(Davi) &= \{ file_1 : (read), \\&\quad file_2 : (write), \\&\quad program_1 : (read), \\&\quad socket_1 : (read, write, owner) \}\end{aligned}$$

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 8.5.6.

8.5.3 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 necessariamente 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 multi-nível* (MLS - *Multi-Level Security*), que se baseiam na classificação de sujeitos e objetos do sistema em *níveis de segurança* (*clearance levels*) e na definição de regras usando esses níveis. Um exemplo bem conhecido de escala de níveis de classificação 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 ($h(s_i)$) e *classificação* dos objetos ($c(o_j)$). As regras da política são então estabelecidas usando essas habilitações e classificações, como mostram os modelos descritos a seguir.

Modelo de Bell-LaPadula

Um modelo de controle de acesso que permite formalizar políticas multi-nível 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:

$$\text{request}(s, o, \text{read}) \iff h(s) \geq c(o)$$

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:

$$\text{request}(s, o, \text{write}) \iff h(s) \leq c(o)$$

Pode-se perceber facilmente 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.

Modelo de Biba

Para garantir a *integridade* das informações, um modelo dual ao de Bell-LaPadula foi proposto por Biba [Biba, 1977]. Esse modelo define níveis de integridade $i(x)$ para sujeitos e objetos (como *Baixa*, *Média*, *Alta* e *Sistema*, com $B < M < A < S$), e também possui duas regras básicas:

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:

$$\text{request}(s, o, \text{write}) \iff i(s) \geq i(o)$$

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:

$$\text{request}(s, o, \text{read}) \iff i(s) \leq i(o)$$

A política obrigatória definida através do modelo de Biba 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.

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.

Categorias

Uma extensão frequente às políticas multi-nível 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 sub-conjunto destas. Dessa forma, um sujeito com as categorias $\{\text{suporte}, \text{financeiro}\}$ só pode acessar objetos rotulados como $\{\text{suporte}, \text{financeiro}\}$, $\{\text{suporte}\}$, $\{\text{financeiro}\}$ ou $\{\emptyset\}$. 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].

8.5.4 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 :

$$request(s, o, action) \iff action \in DDT[\text{domain}(s), \text{type}(o)]$$

Por sua vez, as interações entre sujeitos (trocas de mensagens, sinais, etc.) são reguladas através de 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 :

$$request(s_i, s_j, interaction) \iff interaction \in DIT[\text{domain}(s_i), \text{domain}(s_j)]$$

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 8.14.

```

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), (rwdx->unix_t), (auto->login_d);
14 domain login_d    = (/bin/login), (rwdx->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

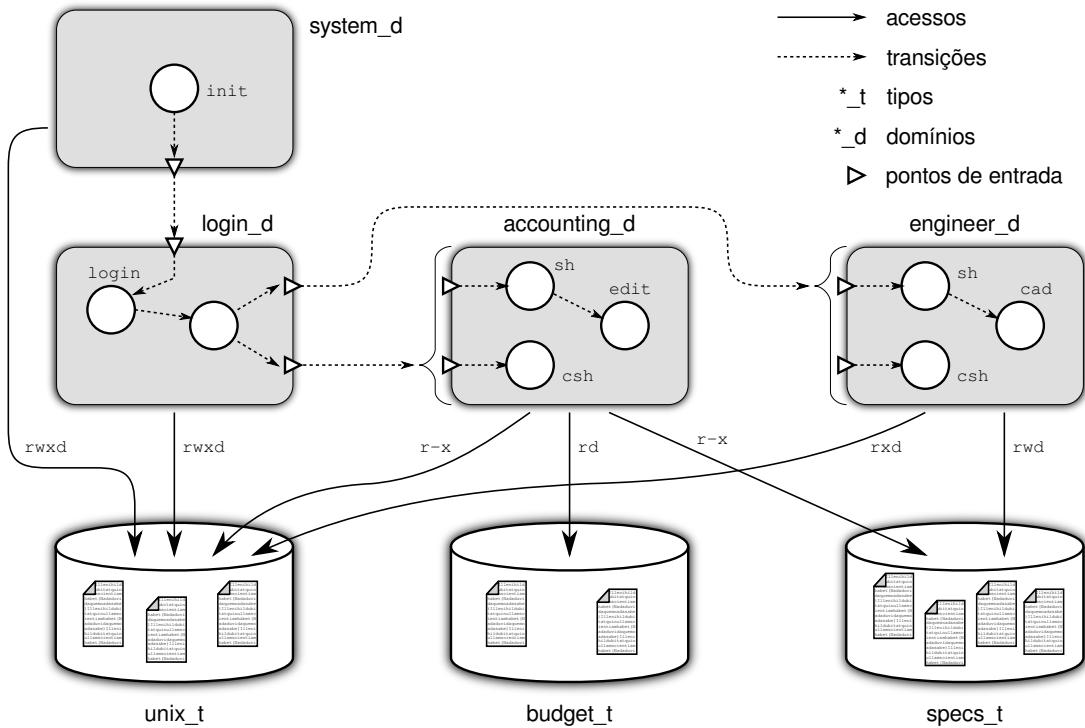


Figura 8.14: Exemplo de política baseada em domínios e tipos.

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.

8.5.5 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 8.15 apresenta os principais componentes de uma política RBAC.

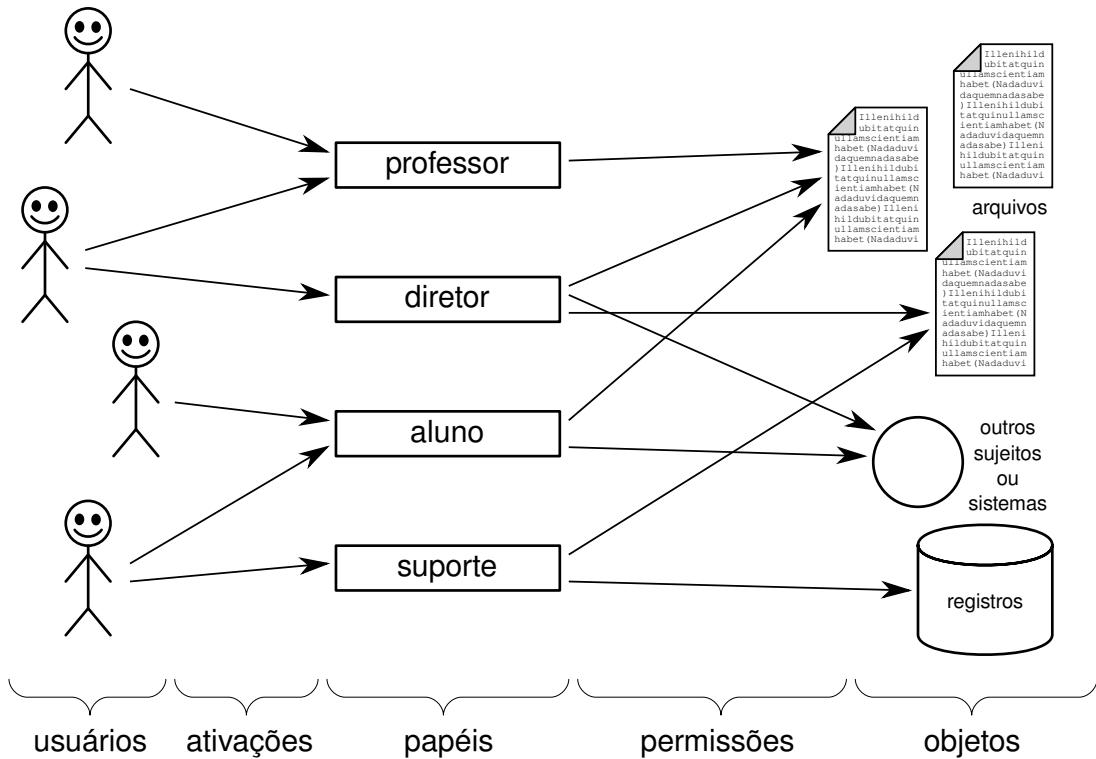


Figura 8.15: Políticas baseadas em papéis.

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 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.

8.5.6 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 infra-estrutura 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).

Infra-estrutura básica

A arquitetura básica de uma infra-estrutura de controle de acesso típica é composta pelos seguintes elementos (Figura 8.16):

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.

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.

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.

A Figura 8.17 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

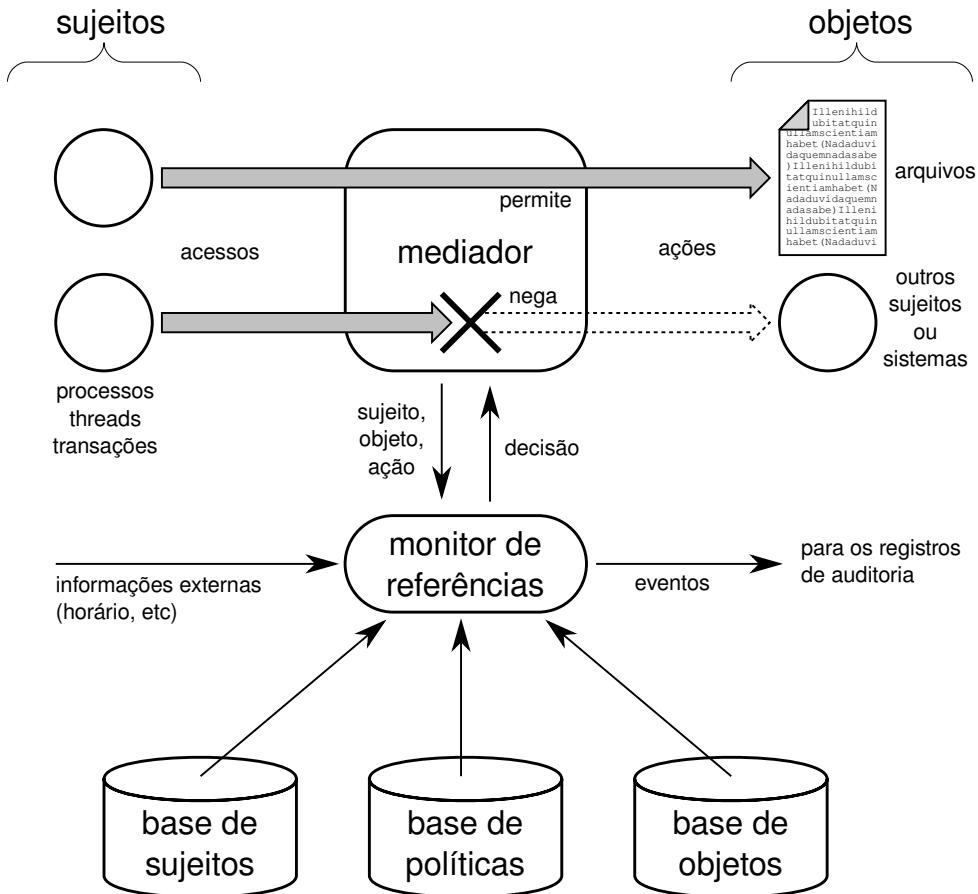


Figura 8.16: Estrutura genérica de uma infra-estrutura de controle de acesso.

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 8.5.2), 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

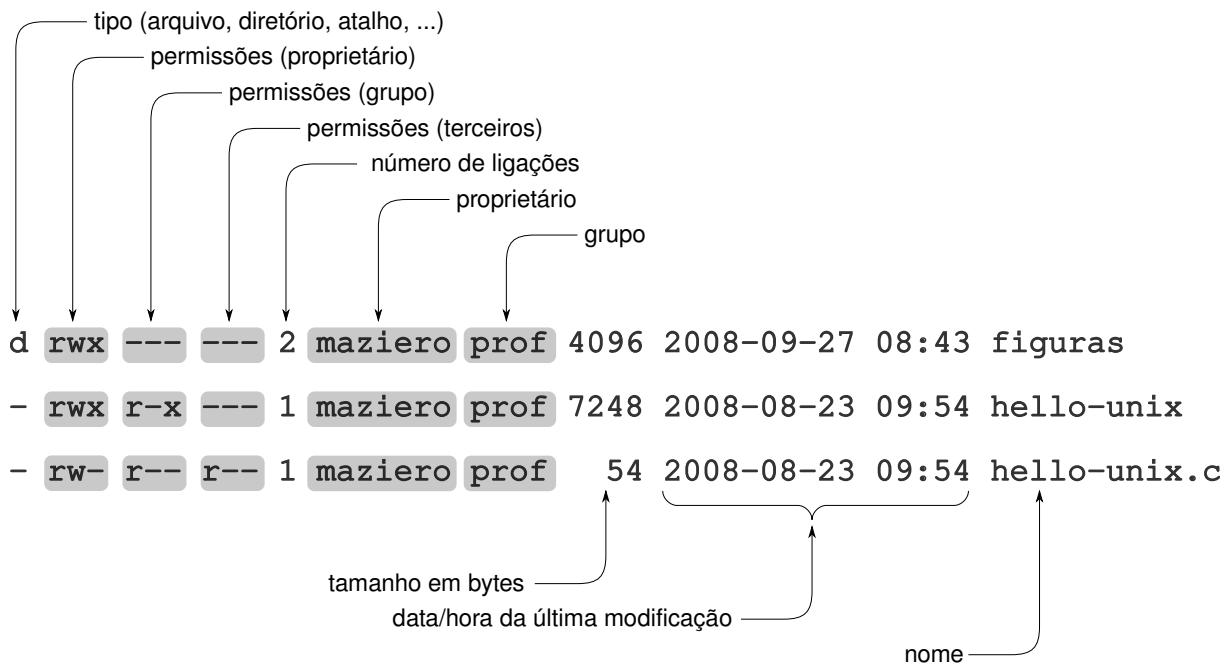


Figura 8.17: Listas de controle de acesso em UNIX.

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).

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 and Solomon, 2004]. 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 8.6). 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 8.18 ilustra alguns dos componentes da estrutura de controle de acesso dos sistemas Windows.

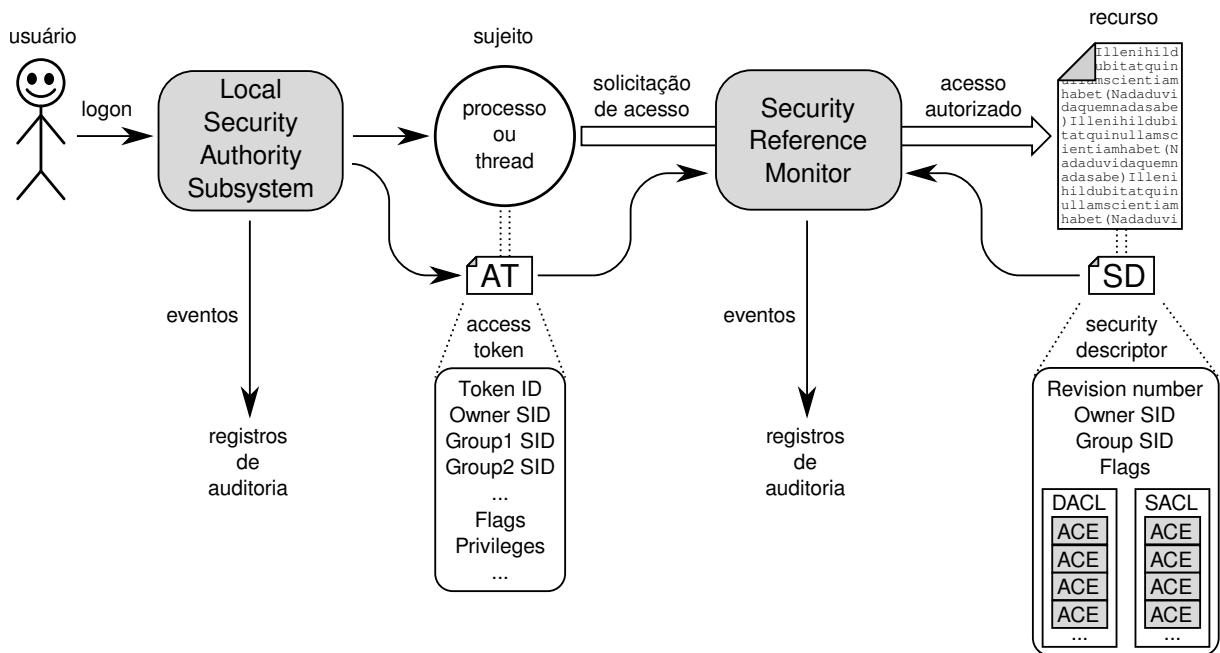


Figura 8.18: Listas de controle de acesso no Windows.

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 multi-políticas, 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 infra-estrutura complexa de segurança 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 multi-nível. 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, 2007], de forma similar ao modelo de Biba (Seção 8.5.3). 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 micro-núcleo 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, 2000] 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 de podem ser atribuídos a usuários distintos.

8.5.7 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 8.5.5), ativados quando o usuário tiver necessidade deles. Esta é a abordagem usada pela infra-estrutura UAC (*User Access Control*) [Microsoft, 2007], 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.

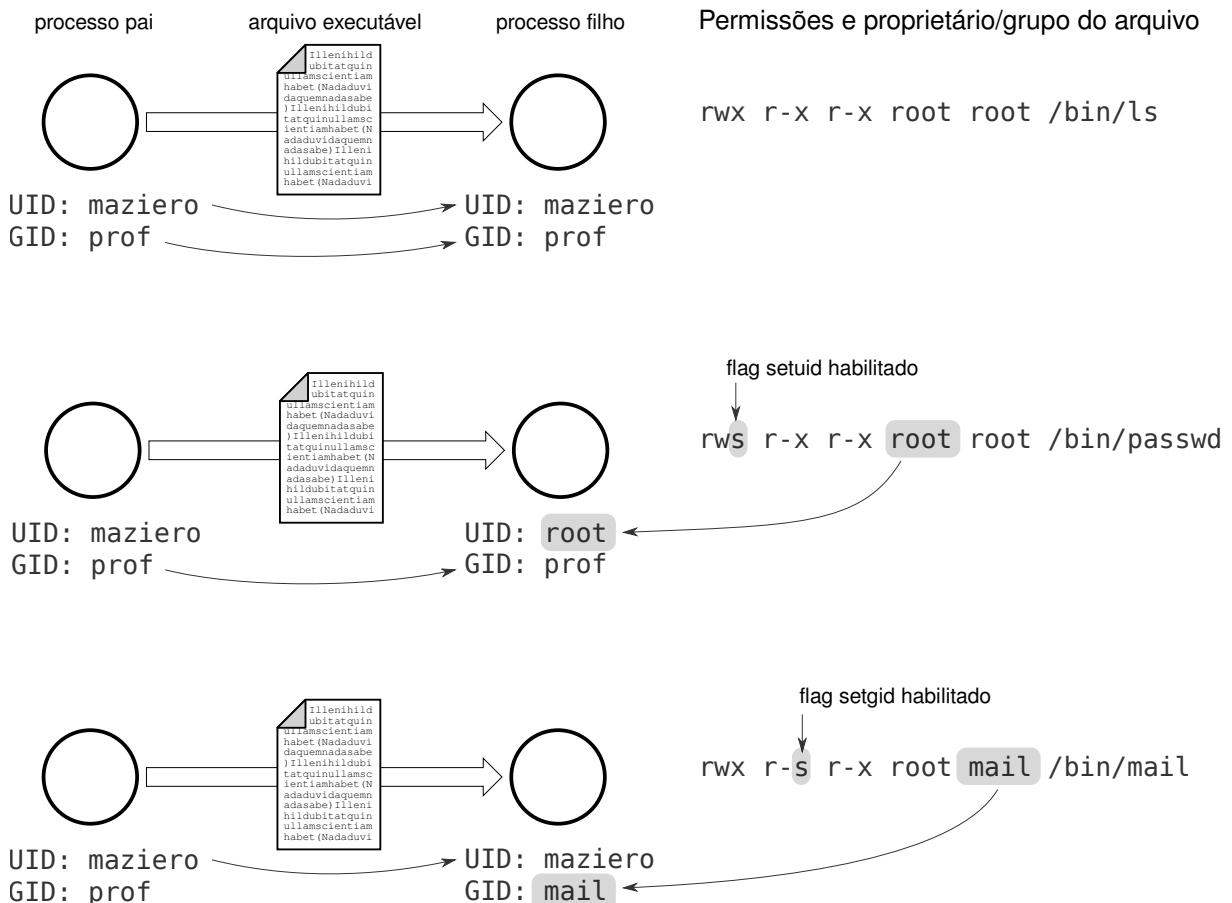
Monitores : definir processos privilegiados, chamados *monitores* ou *supervisores*, recebem pedidos de ações administrativas dos processos não-privilegiados, através de uma API pré-definida; os pedidos dos processos normais são validados e atendidos. Esta é a abordagem definida como *separação de privilégios* 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 8.19 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 super-usuário (*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 8.2.3).

Uma alternativa mais segura aos flags `setuid` e `setgid` são os *privilégios POSIX* (*POSIX Capabilities*⁷), definidos no padrão POSIX 1003.1e [Gallmeister, 1994]. Nessa

⁷O 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.

Figura 8.19: Funcionamento dos flags `setuid` e `setgid` do UNIX.

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)

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 and Solomon, 2004].

8.6 Auditoria

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 consequências⁸ [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.

8.6.1 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 malsucedida 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 sub-sistema 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,

⁸A análise de violações já ocorridas é comumente conhecida como *análise post-mortem*.

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.

Sub-sistema : 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 sub-sistemas.

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:

```

1 ...
2 Sep 8 23:02:09 espec sudo: e89602174 : user NOT in sudoers ; TTY=pts/1 ; USER=root ; COMMAND=/bin/su
3 Sep 8 23:19:57 espec userhelper[20480]: running '/sbin/halt' with user_u:system_r:hotplug_t context
4 Sep 8 23:34:14 espec sshd[6302]: pam_unix(sshd:auth): failure; rhost=210.210.102.173 user=root
5 Sep 8 23:57:16 espec sshd[6302]: Failed password for root from 210.103.210.173 port 14938 ssh2
6 Sep 8 00:08:16 espec sshd[6303]: Received disconnect from 210.103.210.173: 11: Bye Bye
7 Sep 8 00:35:24 espec gdm[9447]: pam_unix(gdm:session): session opened for user rodr by (uid=0)
8 Sep 8 00:42:19 espec gdm[857]: pam_unix(gdm:session): session closed for user rafael3
9 Sep 8 00:49:06 espec userhelper[11031]: running '/sbin/halt' with user_u:system_r:hotplug_t context
10 Sep 8 00:53:40 espec gdm[12199]: pam_unix(gdm:session): session opened for user rafael3 by (uid=0)
11 Sep 8 00:53:55 espec gdm[12199]: pam_unix(gdm:session): session closed for user rafael3
12 Sep 8 01:08:43 espec gdm[9447]: pam_unix(gdm:session): session closed for user rodr
13 Sep 8 01:12:41 espec sshd[14125]: Accepted password for rodr from 189.30.227.212 port 1061 ssh2
14 Sep 8 01:12:41 espec sshd[14125]: pam_unix(sshd:session): session opened for user rodr by (uid=0)
15 Sep 8 01:12:41 espec sshd[14127]: subsystem request for sftp
16 Sep 8 01:38:26 espec sshd[14125]: pam_unix(sshd:session): session closed for user rodr
17 Sep 8 02:18:29 espec sshd[17048]: Accepted password for e89062004 from 20.0.0.56 port 54233 ssh2
18 Sep 8 02:18:29 espec sshd[17048]: pam_unix(sshd:session): session opened for user e89062004 by (uid=0)
19 Sep 8 02:18:29 espec sshd[17048]: pam_unix(sshd:session): session closed for user e89062004
20 Sep 8 09:06:33 espec sshd[25002]: Postponed publickey for mzs from 159.71.224.62 port 52372 ssh2
21 Sep 8 06:06:34 espec sshd[25001]: Accepted publickey for mzs from 159.71.224.62 port 52372 ssh2
22 Sep 8 06:06:34 espec sshd[25001]: pam_unix(sshd:session): session opened for user mzs by (uid=0)
23 Sep 8 06:06:57 espec su: pam_unix(su-l:session): session opened for user root by mzs(uid=500)
24 ...

```

A infra-estrutura tradicional de registro de eventos dos sistemas UNIX é constituída por um *daemon*⁹ chamado *syslogd* (*System Log Daemon*). Esse *daemon* usa um *socket* local

⁹Processo que executa em segundo plano, sem estar associado a uma interface com o usuário, como um terminal ou janela.

e um socket UDP para receber mensagens descrevendo eventos, geradas pelos demais sub-sistemas 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 8.20 apresenta os principais componentes dessa arquitetura.

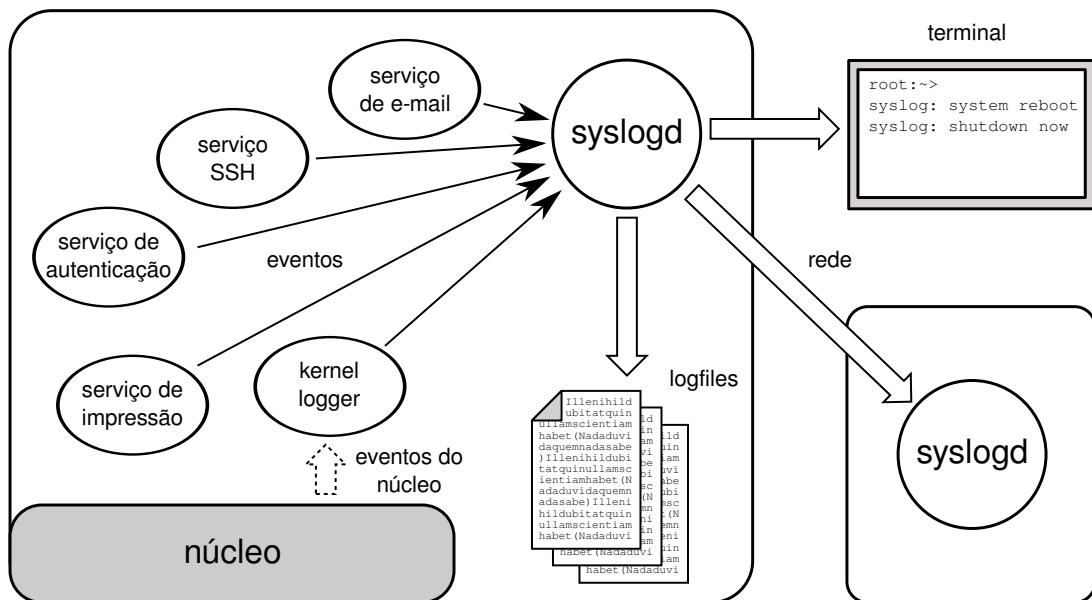


Figura 8.20: O serviço de *logs* em UNIX.

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 papel de centralizador de eventos, recebendo mensagens de várias fontes, entre elas os componentes do subsistema de segurança (LSASS e SRM, Seção 8.5.6), as aplicações e o próprio núcleo. Conforme visto anteriormente, o componente LSASS gera eventos relativos à autenticação dos usuários, enquanto o SRM regista 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 infra-estrutura 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.

8.6.2 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 anti-vírus, 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 correntista 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 anti-vírus: um anti-vírus 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 anti-vírus 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.

8.6.3 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 8.4.3, 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 8.2.2) 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.

Capítulo 9

Virtualização de sistemas

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. 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, arquiteturas e implementações de ambientes virtuais de execução, como máquinas virtuais, emuladores e contêineres.

9.1 Conceitos básicos

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, arquiteturas e técnicas usadas para a implementação de ambientes virtuais de execução.

9.1.1 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 mono-usuário 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 na Seção 9.2.1. 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 são compilados para execução sobre um hardware abstrato 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, no início dos anos 90, o interesse pelas tecnologias de virtualização voltou à tona. Apesar da plataforma PC Intel ainda não oferecer um suporte adequado à virtualização, soluções engenhosas como as adotadas pela empresa VMWare permitiram a virtualização nessa plataforma, embora com desempenho relativamente modesto. Atualmente, as soluções de virtualização de linguagens e de plataformas vêm despertando grande interesse do mercado. Várias linguagens são compiladas para máquinas virtuais portáveis e os processadores mais recentes trazem um suporte nativo à virtualização.

9.1.2 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 o sistema de 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 em código de máquina aceitas pelo processador e todas as operações de acesso aos recursos do hardware

(acesso físico à memória, às portas de entrada/saída, ao relógio do sistema, etc.). Essa interface é dividida em duas partes:

- *Instruções de usuário (User ISA)*: comprehende as instruções do processador e demais itens de hardware acessíveis aos programas do usuário, que executam com o processador operando em modo não-privilegiado;
- *Instruções de sistema (System ISA)*: comprehende 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 dos usuários. Essas chamadas permitem um acesso controlado das aplicações aos dispositivos periféricos, à memória e às instruções privilegiadas do processador.
- *Chamadas de bibliotecas (libcalls)*: 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 9.1 apresenta essa visão conceitual da arquitetura de um sistema computacional, com seus vários componentes e as respectivas interfaces entre eles.

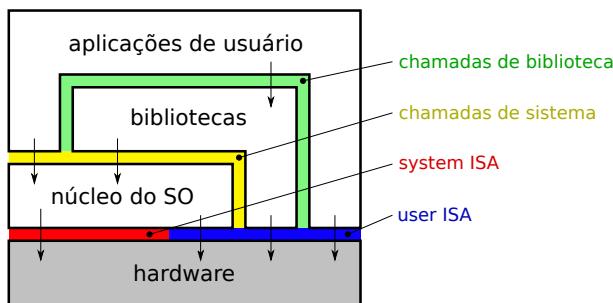


Figura 9.1: Componentes e interfaces de um sistema computacional.

9.1.3 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)*.

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 9.2 representa essas duas interfaces.

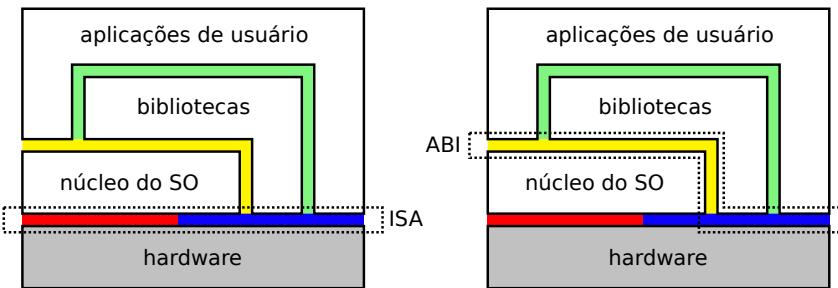


Figura 9.2: Interfaces de sistema ISA e ABI [Smith and Nair, 2004].

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/APIs pré-definidas. A Figura 9.3, extraída de [Smith and Nair, 2004], ilustra esses problemas de compatibilidade entre interfaces.

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.

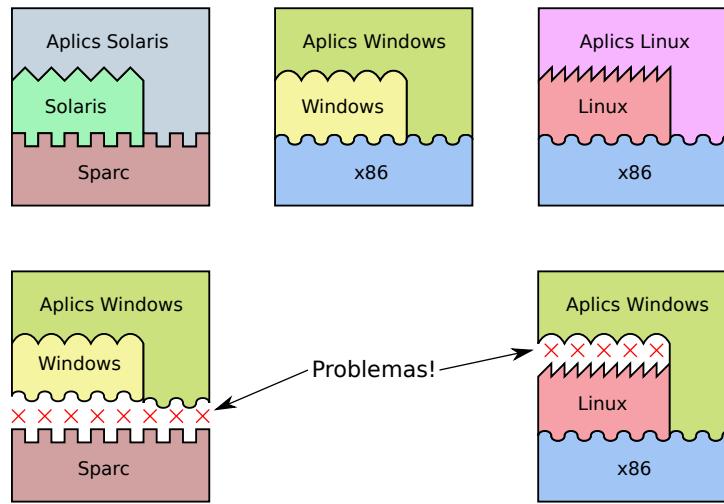


Figura 9.3: Problemas de compatibilidade entre interfaces [Smith and Nair, 2004].

- *Otimizações inter-componentes*: 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, que será apresentada na próxima seção.

9.1.4 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 do programa 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 9.4, 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/AMD).

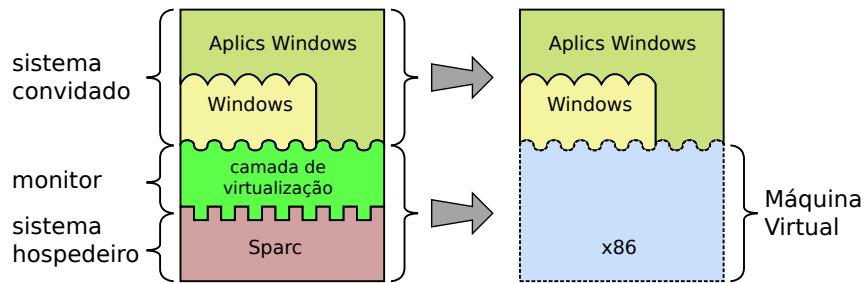


Figura 9.4: Uma máquina virtual [Smith and Nair, 2004].

Um ambiente de máquina virtual consiste de três partes básicas, que podem ser observadas na Figura 9.4:

- O sistema real, nativo ou hospedeiro (*host system*), que contém os recursos reais de hardware e software do sistema;
- o sistema virtual, também denominado *sistema convidado* (*guest system*), que executa sobre o sistema virtualizado; em alguns casos, vários sistemas virtuais podem coexistir, executando simultaneamente sobre o mesmo sistema real;
- a camada de virtualização, chamada *hipervisor* ou *monitor* (*VMM – Virtual Machine Monitor*), que constrói as interfaces virtuais a partir da interface real.

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 9.5):

- *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.

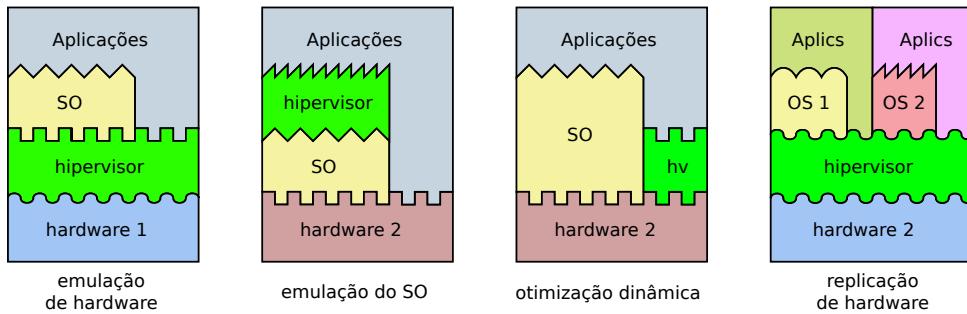


Figura 9.5: Possibilidades de virtualização [Smith and Nair, 2004].

9.1.5 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 dos sistemas operacionais 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. Exemplos típicos dessa estruturação em níveis de abstração são os subsistemas de rede e de disco em um sistema operacional convencional. No sub-sistema de arquivos, cada nível de abstração trata de um problema: interação com o dispositivo físico de armazenamento, escalonamento de acessos ao dispositivo, gerência de *buffers* e *caches*, alocação de arquivos, diretórios, controle de acesso, etc. A Figura 9.6 apresenta os níveis de abstração de um subsistema de gerência de disco típico.

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 9.7 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).

A Figura 9.8 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.

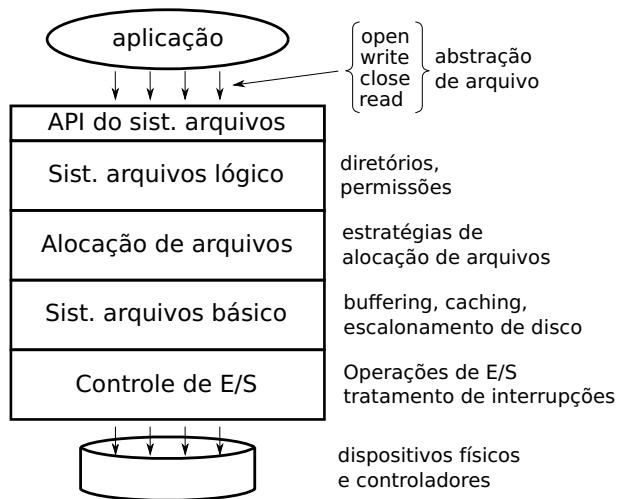


Figura 9.6: Níveis de abstração em um sub-sistema de disco.

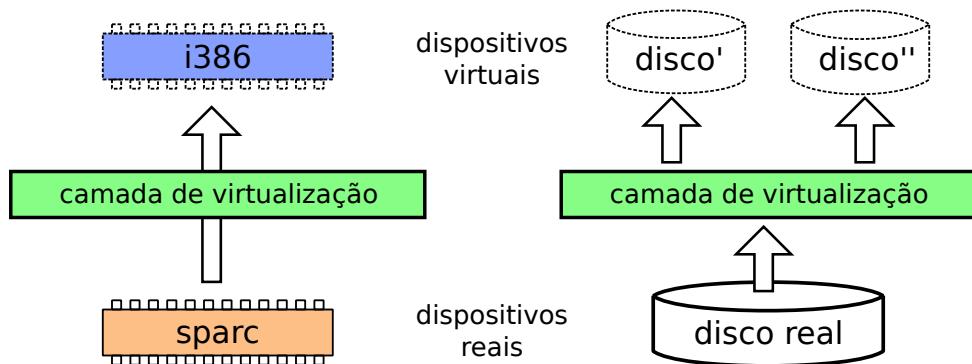


Figura 9.7: Virtualização de recursos do hardware.

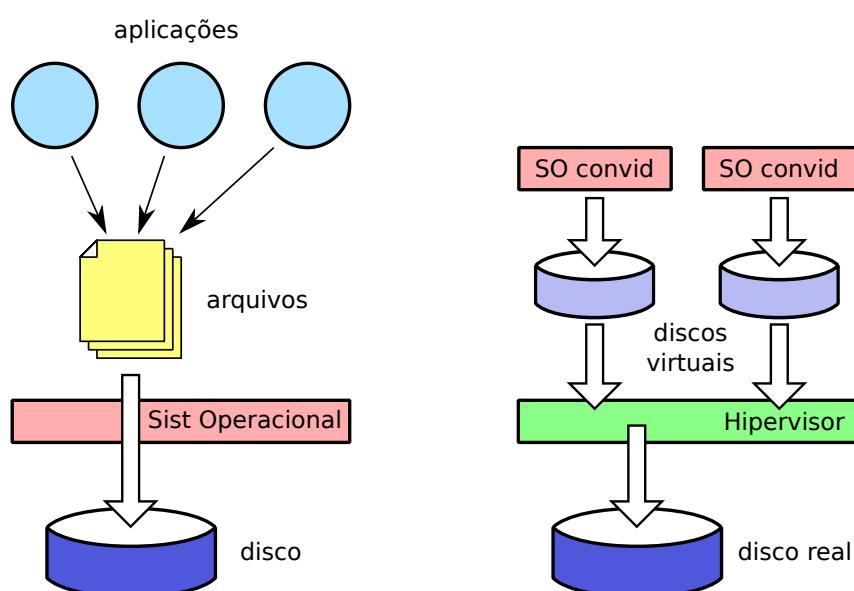


Figura 9.8: Abstração versus virtualização de um disco rígido.

9.2 A construção de máquinas virtuais

Conforme apresentado, a virtualização consiste em reescrever uma ou mais interfaces do sistema computacional, para oferecer novas interfaces e assim permitir a execução de sistemas operacionais ou aplicações incompatíveis com as interfaces originais.

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.

Esta seção aborda inicialmente o conceito formal de virtualização, para em seguida discutir as principais técnicas usadas na construção de máquinas virtuais.

9.2.1 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).

O hipervisor ou monitor de máquina virtual descrito por Popek/Goldberg corresponde à camada de virtualização apresentada na Seção 9.1.4. Para funcionar de forma correta e eficiente, o hipervisor deve atender a alguns requisitos básicos: ele deve prover um ambiente de execução aos programas essencialmente idêntico ao da máquina real, do ponto de vista lógico. Programas executando sobre uma máquina virtual devem apresentar, no pior caso, leves degradações de desempenho. Além disso, o hipervisor deve ter controle completo sobre os recursos do sistema real (o sistema hospedeiro). A partir desses requisitos, foram estabelecidas as seguintes propriedades a serem satisfeitas por um hipervisor ideal:

Equivalência : um hipervisor provê 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 [Popek and Goldberg, 1974, 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.

Recursividade: alguns sistemas de máquinas virtuais 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*.

Inspeção: 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.

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, enunciado e provado por Popek e Goldberg em [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 sub-conjunto 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 *para-virtualização*, que se baseia em reescrever parte do sistema convidado para não usar essas instruções sensíveis. Ambas as técnicas são discutidas a seguir.

9.2.2 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 diretamente o hardware para 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 necessário para a construção de hipervisores eficientes está presente em sistemas de grande porte, como os mainframes, mas é apenas parcial nos micro-processadores de mercado. Por exemplo, a família de processadores *Intel Pentium IV* (e anteriores) possui 17 instruções sensíveis que podem ser executadas em modo usuário sem gerar exceções, o que viola o teorema de Goldberg (Seção 9.2.1) e dificulta a criação de máquinas virtuais em sistemas que usam esses processadores [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 atuais 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 de virtualização para as plataformas Intel/AMD se baseiam 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 “modelo 0/1/3” e modelo “0/2/3”, respectivamente (Figura 9.9). Todavia, para que a estratégia de redução de privilégio possa funcionar, algumas instruções do sistema operacional convidado devem ser reescritas dinamicamente, em tempo de carga do sistema convidado na memória, pois ele foi construído para executar no nível 0.

sistema não-virtualizado		virtualização com modelo 0/1/3		virtualização com modelo 0/2/3	
3	aplicações	3	aplicações	3	aplicações
2	<i>não usado</i>	2	<i>não usado</i>	2	núcleo convidado
1	<i>não usado</i>	1	núcleo convidado	1	<i>não usado</i>
0	núcleo do SO	0	hipervisor	0	hipervisor

Figura 9.9: Uso dos níveis de proteção em processadores Intel/AMD convencionais.

Por volta de 2005, os principais fabricantes de micro-processadores (*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 idéia 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* \rightarrow *non-root*) e *VM exit* (transição *non-root* \rightarrow *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 9.10 traz uma visão geral da arquitetura Intel IVT.

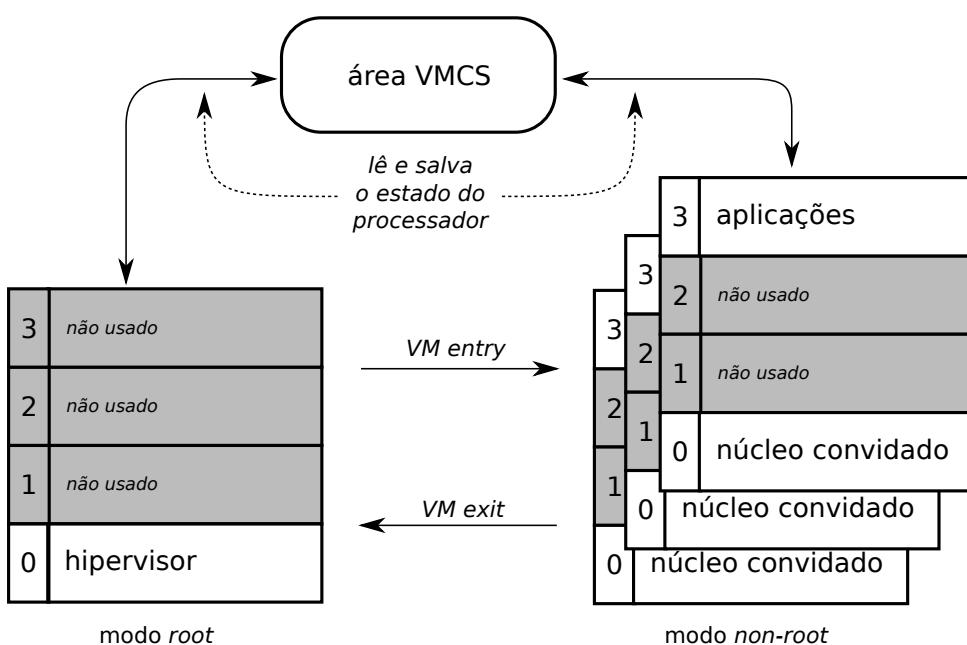


Figura 9.10: Visão geral da arquitetura Intel IVT.

Além da Intel e AMD, outros fabricantes de hardware têm se preocupado 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, 2007], que respeita os requisitos necessários à virtualização do processador e da gestão de memória.

Conforme apresentado, a virtualização do processador pode ser 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 virtual estudados na Seção 5.7. 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].

9.2.3 Formas 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 uso da virtualização em um sistema. De acordo com as interfaces em que são aplicadas, as formas 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 toda as instruções geradas no sistema convidado em instruções do processador real. A máquina virtual Java (JVM, Seção 9.6.7) 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 apenas emula 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, VirtualPC e Xen.

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 nomes de 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*, 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, essas várias formas de virtualização podem ser usadas 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.

9.3 Tipos de máquinas virtuais

Além de resolver problemas em áreas específicas do sistema operacional, as várias formas de virtualização disponíveis podem ser combinadas para a construção de *máquinas virtuais*. Uma máquina virtual é um ambiente de suporte à execução de software, construído usando uma ou mais formas de virtualização. Conforme as características do ambiente virtual proporcionado, as máquinas virtuais podem ser classificadas em três categorias, representadas na Figura 9.11:

Máquinas virtuais de processo (*Process Virtual Machines*): também chamadas de máquinas virtuais de aplicação, 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.

Máquinas virtuais de sistema operacional (*Operating System Virtual Machines*): 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 *Solaris Zones* e *FreeBSD Jails* implementam este conceito.

Máquinas virtuais de sistema (*System Virtual Machines*): 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 *VMware* e *VirtualBox*.

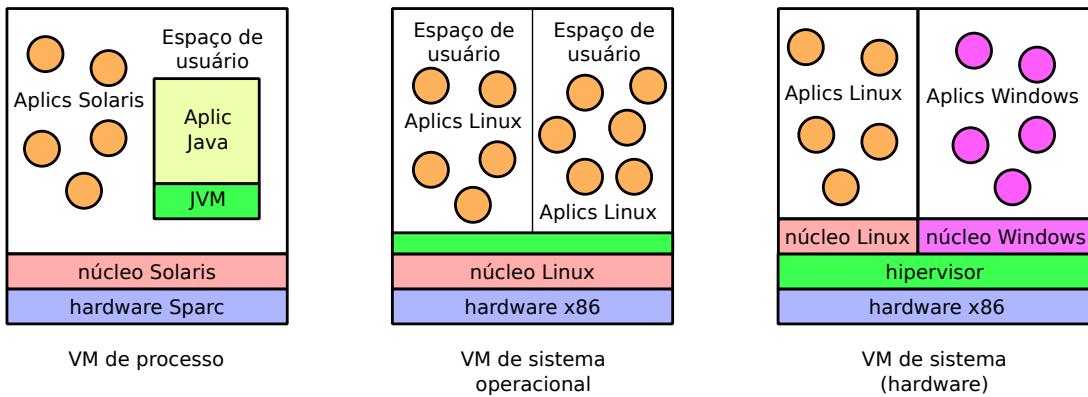


Figura 9.11: Máquinas virtuais de processo, de sistema operacional e de hardware.

Por outro lado, 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 9.1.2):

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 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 9.2.1, 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.

9.3.1 Máquinas virtuais de processo

Uma máquina virtual de processo ou de aplicação (*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 subjacente (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 9.2.1, 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 9.1.2) 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, 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 multi-tarefas*, os *tradutores dinâmicos* e alguns *depuradores de memória*:

Sistemas operacionais multi-tarefas: os sistemas operacionais que suportam vários processos simultâneos, estudados no Capítulo 2, também podem ser vistos como ambientes de máquinas virtuais. Em um sistema multi-tarefas, 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 compartilhamento desses 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, de forma transparente. 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 9.1.2) 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 aplicação e máquina real se restrinjam às chamadas do sistema operacional, o hipervisor precisa apenas mapear 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 transformadas em chamadas Unix, de forma dinâmica e transparente (Figura 9.12).

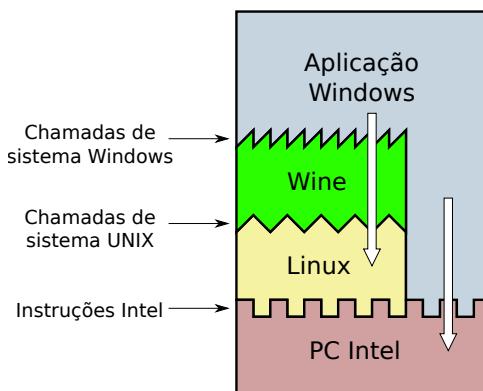


Figura 9.12: Funcionamento do emulador Wine.

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. Várias linguagens empregam a mesma abordagem (embora usando um *bytecode* próprio), como Perl, Python e Smalltalk.

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 repetidos frequentemente são traduzidos e mantidos em cache pelo hipervisor, permitem obter ganhos de desempenho significativos.

9.3.2 Máquinas virtuais de sistema operacional

Em muitas situações, a principal ou mesmo única motivação para o uso de máquinas virtuais é a propriedade de isolamento (Seção 9.2.1). Esta propriedade tem uma grande importância no contexto da segurança de sistemas, por permitir isolar entre si sub-sistemas independentes que executam sobre o mesmo hardware. Por exemplo, a estratégia organizacional conhecida como *consolidação de servidores* advoga o uso de máquinas virtuais para abrigar os diversos servidores (de nomes, de arquivos, de e-mail, de Web) de um determinado domínio. Dessa forma, pode-se fazer um uso mais eficiente do hardware disponível, preservando o isolamento entre os serviços. Todavia, o impacto da virtualização de uma plataforma de hardware ou sistema operacional sobre o desempenho do sistema final pode ser elevado. As principais fontes desse impacto são a virtualização dos recursos (periféricos) da máquina real e a necessidade de tradução binária dinâmica das instruções do processador real.

Uma forma simples e eficiente de implementar o isolamento entre aplicações ou sub-sistemas em um sistema operacional consiste na virtualização do espaço de usuário (*userspace*). Nesta abordagem, denominada *máquinas virtuais de sistema operacional* ou *servidores virtuais*, o espaço de usuário do sistema operacional é dividido em áreas isoladas denominadas *domínios* ou *zonas virtuais*. A cada domínio virtual é alocada uma parcela dos recursos do sistema operacional, como memória, tempo de processador e espaço em disco. Além disso, alguns recursos do sistema real podem ser virtualizados, como é o caso frequente das interfaces de rede: cada domínio tem sua própria interface virtual e, portanto, seu próprio endereço de rede. Em várias implementações, cada domínio virtual define seu próprio espaço de nomes: assim, é possível encontrar um usuário pedro no domínio d_3 e outro usuário pedro no domínio d_7 , sem conflitos. Essa noção de espaços de nomes distintos pode se estender 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 mudar 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 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 anterior.

O núcleo do sistema operacional é o mesmo para todos os domínios virtuais, e sua interface (conjunto de chamadas de sistema) é preservada. Normalmente apenas uma nova chamada de sistema é necessária, para que um processo no domínio inicial d_0 possa solicitar sua migração para um outro domínio d_i . A Figura 9.13 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.

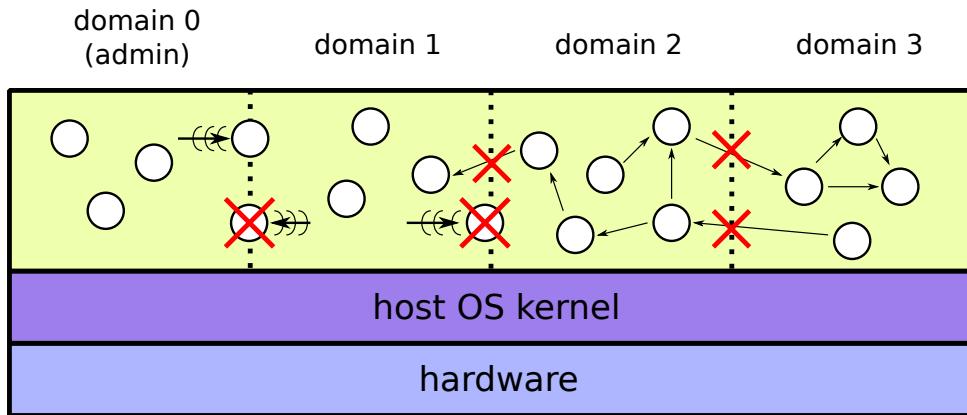


Figura 9.13: Máquinas virtuais de sistema operacional.

Há várias implementações disponíveis de mecanismos para a criação de domínios virtuais. A técnica mais antiga é implementada pela chamada de sistema `chroot`, disponível na maioria dos sistemas UNIX. Essa chamada de sistema atua exclusivamente 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 sub-árvore da hierarquia de diretórios, ou seja, ele fica “confinado” a essa sub-árvore. Os filhos desse processo herdam a mesma visão do sistema de arquivos, que não pode ser revertida. Por exemplo, um processo que executa a chamada `chroot` (“`/var/spool/postfix`”) passa a ver somente a hierarquia de diretórios a partir do diretório `/var/spool/postfix`, que passa a ser seu diretório raiz. A Figura 9.14 ilustra essa operação.

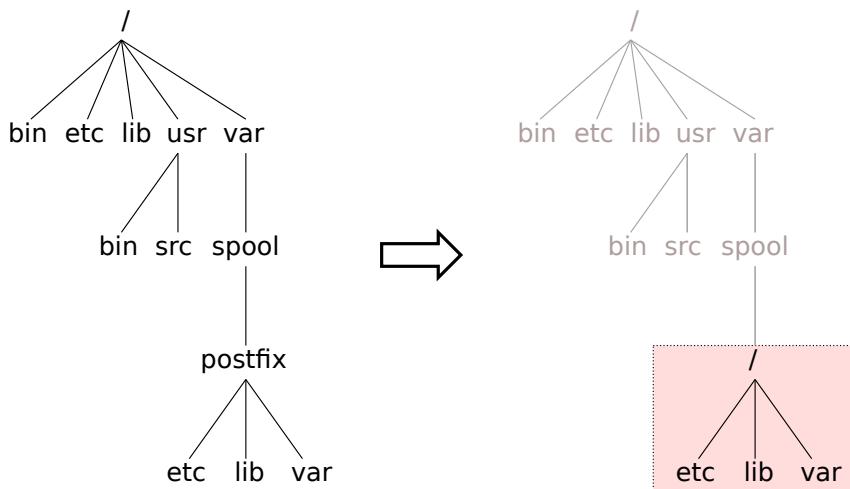


Figura 9.14: A chamada de sistema `chroot` (“`/var/spool/postfix`”).

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 aos processos do servidor, 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 às seguintes restrições:

- a visão do sistema operacional se restringe aos 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 confrinhar 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/OpenVZ* e *Vservers/FreeVPS*.

9.3.3 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 virtualiza o acesso aos recursos, para que cada sistema operacional convidado tenha um conjunto de recursos virtuais próprio, construído a partir dos recursos físicos existentes

na máquina real. Assim, cada máquina virtual terá sua própria interface de rede, seu próprio disco, sua própria 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 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 9.2.1). 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 mono-tarefa e suas aplicações. Assim, o compartilhamento de recursos não era responsabilidade do sistema operacional convidado, mas do hipervisor subjacente. No entanto, ao longo dos anos 70, como o desenvolvimento de sistemas operacionais multi-tarefas 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 micro-processadores 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 9.15:

Hipervisores nativos (ou *de tipo I*): nesta categoria, o hipervisor 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*.

Hipervisores convidados (ou *de tipo II*): nesta categoria, o hipervisor executa como um processo normal sobre um sistema operacional nativo subjacente. O hipervisor utiliza os recursos oferecidos pelo sistema operacional nativo para oferecer recursos virtuais ao sistema operacional convidado que executa sobre ele. Normalmente, um hipervisor convidado suporta apenas uma máquina virtual com uma instância de sistema operacional convidado. Caso mais máquinas sejam necessárias, mais hipervisores devem ser lançados, como processos separados. Exemplos de sistemas que adotam esta estrutura incluem o *VMware Workstation*, o *QEMU* e o *VirtualBox*.

Pode-se afirmar que os hipervisores convidados são mais flexíveis que os hipervisores nativos, pois podem ser facilmente instalados/removidos em máquinas com sistemas

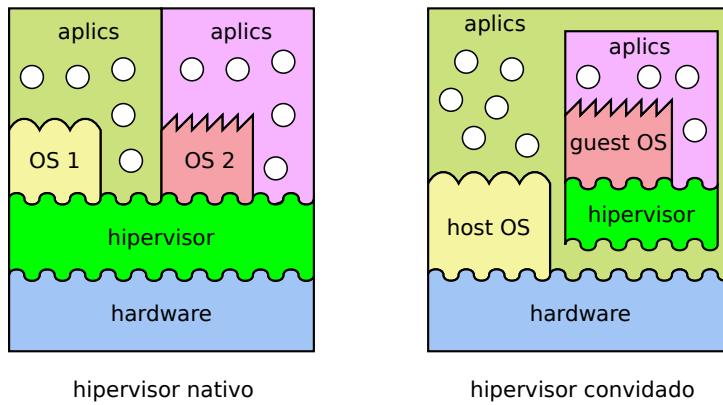


Figura 9.15: Arquiteturas de máquinas virtuais de sistema.

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 9.4.5.

9.4 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 para-virtualização. Além disso, algumas técnicas complementares são usadas para melhorar o desempenho dos sistemas de máquinas virtuais. Essas técnicas são discutidas nesta seção.

9.4.1 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 nenhuma modificação.

Exemplos típicos de emulação completa abordagem são os sistemas de máquinas virtuais QEMU, que oferece um processador *Intel Pentium II* 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 9.4.3) 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 9.16).

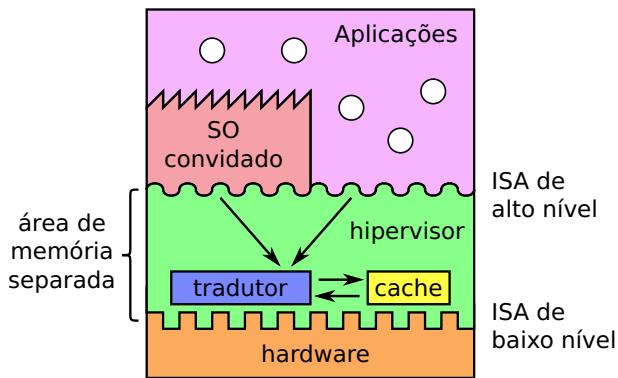


Figura 9.16: Hipervisor embutido no hardware.

9.4.2 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 vêem 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 9.2.1). 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 9.2.2 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*.

9.4.3 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 a execução do sistema convidado avança.

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.

9.4.4 Paravirtualização

A Seção 9.2.2 mostrou que as arquiteturas de alguns processadores, como o *Intel x86*, podem ser difíceis de virtualizar, porque algumas instruções sensíveis não podem ser interceptadas pelo hipervisor. Essas instruções sensíveis devem 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 um *vetor 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 5.3.4) dos processadores *x86* é gerenciado diretamente pelo hardware, sem possibilidade de intervenção direta do hipervisor no caso de uma falta de página.

Em meados 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. As 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 9.17 ilustra esse conceito.

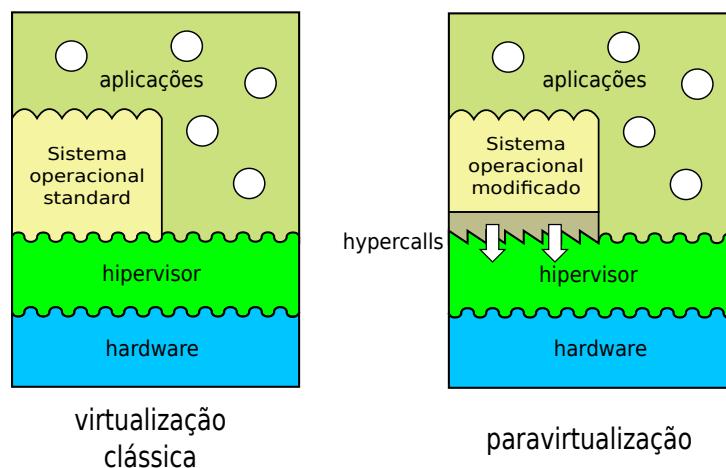


Figura 9.17: Paravirtualização.

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 *x86* 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 9.6.3) 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 tem tido sucesso, por conta do desempenho obtido nos sistemas virtualizados, além de simplificar a interface de baixo nível dos sistemas convidados.

9.4.5 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 9.18):
 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 9.18):

1. 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.
2. 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].
3. 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, ilustrada na Figura 9.19, é usada pelo sistema *VMware* [VMware, 2000].

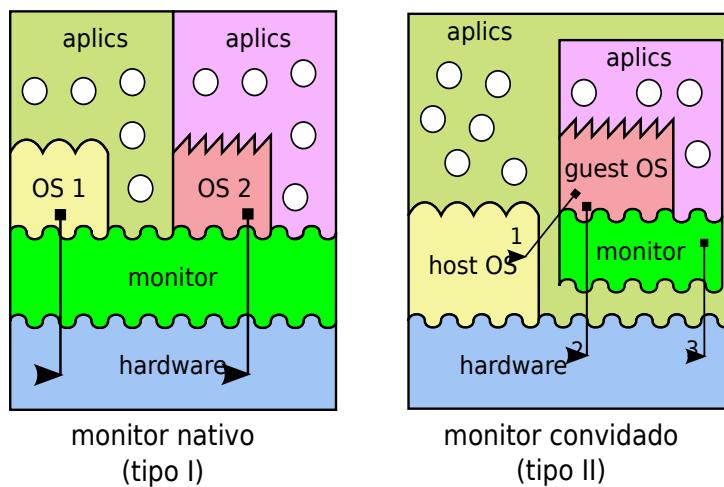


Figura 9.18: Otimizações em sistemas de máquinas virtuais.

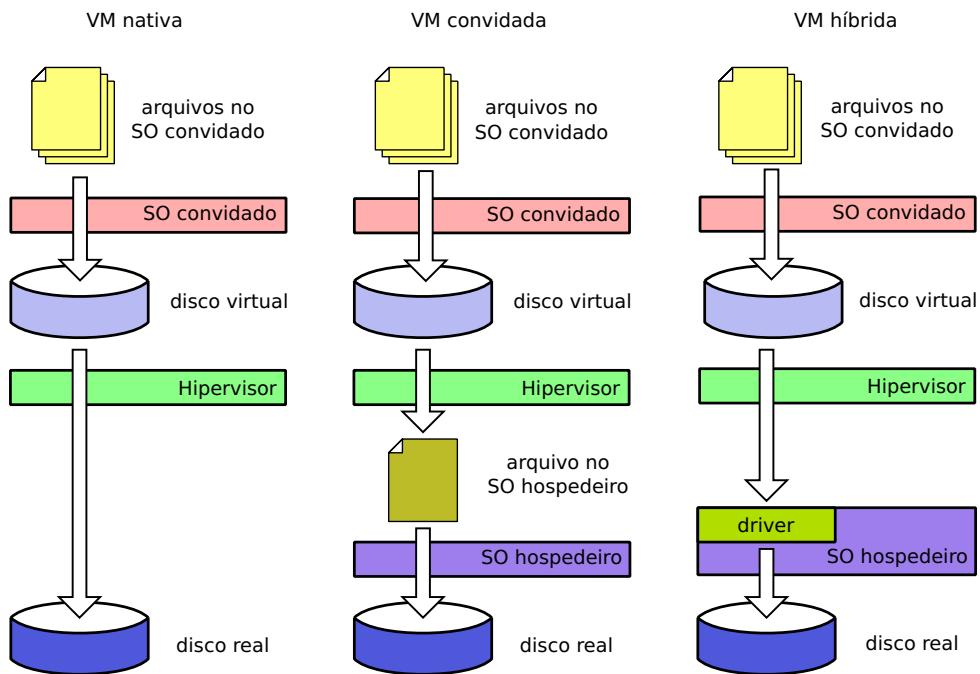


Figura 9.19: Desempenho de hipervisores nativos e convidados.

9.5 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.

O campo de aplicação mais conhecido da virtualização é a **portabilidade de aplicações binárias**. Este uso de máquinas virtuais começou na década de 1970, com o compilador UCSD Pascal. Esse compilador 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. Para executar a aplicação em outra plataforma, bastava portar a implementação da *P-Machine*. Esse esquema foi posteriormente adotado pelas linguagens Java, C#, Perl e Python, entre outras, nas quais o código fonte é compilado em um código binário (*bytecode*) para uma máquina virtual específica. 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.

O **compartilhamento de hardware** é outro uso frequente da virtualização, por tornar possível executar simultaneamente vários sistemas operacionais, ou várias instâncias do mesmo 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 de rede. Essa abordagem visa

aproveitar melhor o hardware existente: como a distribuição dos recursos entre os sistemas convidados pode ser ajustada dinamicamente, pode-se alocar mais recursos aos serviços com maior demanda em um dado instante.

Além dessas aplicações, diversas outras possibilidades de aplicação de ambientes de máquinas virtuais podem ser encontradas, entre as quais:

Suporte a aplicações legadas : pode-se preservar ambientes virtuais para a execução de aplicações legadas, sem a necessidade de manter computadores reservados para isso.

Experimentação com redes e sistemas distribuídos : 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 dos pontos de vista didático e experimental.

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 9.3.2) foram criadas justamente com o objetivo de isolar sub-sistemas 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 software de baixo nível : o uso de máquinas virtuais para o desenvolvimento de software de baixo nível, como 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.

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 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.

9.6 Ambientes de máquinas virtuais

Esta seção apresenta alguns exemplos de sistemas de máquinas virtuais de uso corrente. Serão apresentados os sistemas *VMWare*, *FreeBSD Jails*, *Xen*, *User-Mode Linux*, *QEMU*, *Valgrind* e *JVM*. Entre eles há máquinas virtuais de aplicação e de sistema, com virtualização total ou paravirtualização, além de abordagens hibridas. Eles foram escolhidos por estarem entre os mais representativos de suas respectivas classes.

9.6.1 VMware

Atualmente, o *VMware* é a máquina virtual para a plataforma x86 de uso mais difundido, 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 *VMware* produz vários produtos com hipervisores nativos e convidados:

- Hipervisor convidado:
 - *VMware Workstation*: primeira versão comercial da máquina virtual, lançada em 1999, para ambientes *desktop*;
 - *VMware Fusion*: versão experimental para o sistema operacional *Mac OS* com processadores Intel;
 - *VMware Player*: versão gratuita do *VMware Workstation*, com as mesmas funcionalidades mas limitado a executar máquinas virtuais criadas previamente com versões comerciais;
 - *VMWare Server*: conta com vários recursos do *VMware Workstation*, mas é voltado para pequenas e médias empresas;
- Hipervisor nativo:
 - *VMware ESX Server*: para servidores de grande porte, possui um núcleo proprietário chamado *vmkernel* e Utiliza o *Red Hat Linux* para prover outros serviços, tais como a gerência de usuários.

O *VMware Workstation* utiliza as estratégias de virtualização total e tradução dinâmica (Seção 9.4). O *VMware ESX Server* implementa ainda a paravirtualização. Por razões de desempenho, o hipervisor do *VMware* utiliza uma abordagem híbrida (Seção 9.4.5) 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.

9.6.2 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 9.3.2. 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/remover 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.

9.6.3 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 um vetor 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 9.20, simplifica muito a evolução do hipervisor, por permitir utilizar os *drivers* desenvolvidos para o sistema Linux.

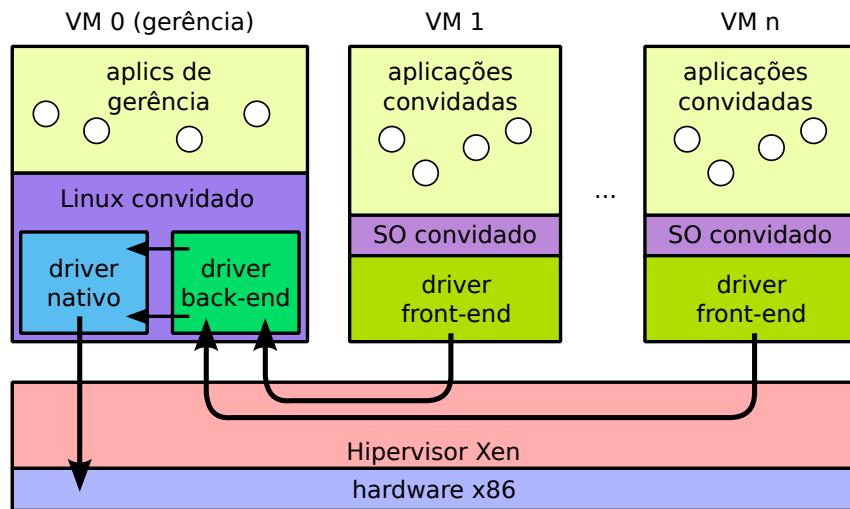


Figura 9.20: O hipervisor *Xen*.

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*.

9.6.4 User-Mode Linux

O *User-Mode Linux* foi proposto por Jeff Dike em 2000, como uma alternativa de uso de máquinas virtuais no ambiente Linux [Dike, 2000]. O núcleo do Linux foi portado de forma a poder executar sobre si mesmo, como um processo do próprio Linux. O resultado é um *user space* 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* é hipervisor convidado, ou seja, 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 maior dificuldade na implementação do *User-Mode Linux* foi encontrar formas de virtualizar as funcionalidades do hardware para as chamadas de sistema do Linux, sobretudo a distinção entre o modo privilegiado do núcleo e o modo não-privilegiado de usuário. Um código somente pode estar em modo privilegiado se é confiável o suficiente para ter pleno acesso ao hardware, como o próprio núcleo do sistema operacional. O *User-Mode Linux* deve possuir uma distinção de privilégios equivalente para permitir que o seu núcleo tenha acesso às chamadas de sistema do sistema hospedeiro quando os seus próprios processos solicitarem este acesso, ao mesmo tempo em que impede os mesmos de acessar diretamente os recursos reais subjacentes.

No hipervisor, a distinção de privilégios foi implementada com o mecanismo de interceptação de chamadas do próprio Linux, fornecido pela chamada de sistema `ptrace`¹. Usando a chamada `ptrace`, o hipervisor recebe o controle de todas as chamadas de sistema de entrada/saída geradas pelas máquinas virtuais. Todos os sinais gerados ou enviados às máquinas virtuais também são interceptados. A chamada `ptrace` também é utilizada para manipular o contexto do sistema convidado.

O *User-Mode Linux* utiliza o sistema hospedeiro para operações de entrada/saída. Como a máquina virtual é um processo no sistema hospedeiro, a troca de contexto entre duas instâncias de máquinas virtuais é rápida, assim como a troca entre dois processos do sistema hospedeiro. Entretanto, modificações no sistema convidado foram necessárias para a otimização da troca de contexto. A virtualização das chamadas de sistema é implementada pelo uso de uma *thread* de rastreamento que intercepta e redireciona todas as chamadas de sistema para o núcleo virtual. Este identifica a chamada de sistema e os seus argumentos, cancela a chamada e modifica estas informações no hospedeiro, onde o processo troca de contexto e executa a chamada na pilha do núcleo.

O *User-Mode Linux* está disponível na versão 2.6 do núcleo Linux, ou seja, ele foi assimilado à árvore oficial de desenvolvimento do núcleo, portanto melhorias na sua arquitetura deverão surgir no futuro, ampliando seu uso em diversos contextos de aplicação.

¹Chamada de sistema que permite observar e controlar a execução de outros processos; o comando `strace` do Linux permite ter uma noção de como a chamada de sistema `ptrace` funciona.

9.6.5 QEMU

O *QEMU* é um hipervisor com virtualização completa [Bellard, 2005]. Não requer alterações ou otimizações no sistema hospedeiro, pois utiliza intensivamente a tradução dinâmica (Seção 9.4) como técnica para prover a virtualização. É um dos poucos hipervisores recursivos, ou seja, é possível chamar o *QEMU* a partir do próprio *QEMU*. O hipervisor *QEMU* oferece dois modos de operação:

- *Emulação total do sistema*: emula um sistema completo, incluindo processador (normalmente um *Intel Pentium II*) 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.

Por meio de um módulo instalado no núcleo do sistema hospedeiro, denominado *KQEMU* ou *QEMU Accelerator*, o hipervisor *QEMU* consegue obter um desempenho similar ao de outras máquinas virtuais como *VMWare* e *User-Mode Linux*. Com este módulo, o *QEMU* passa a executar as chamadas de sistema emitidas pelos processos convidados diretamente sobre o sistema hospedeiro, ao invés de interpretar cada uma. O *KQEMU* permite associar os dispositivos de entrada/saída e o endereçamento de memória do sistema convidado aos do sistema hospedeiro. Processos em execução sobre o núcleo convidado passam a executar diretamente no modo usuário do sistema hospedeiro. O modo núcleo do sistema convidado é utilizado apenas para virtualizar o processador e os periféricos.

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 adquirido pela *Sun Microsystems* e liberado para uso público sob a licença *GPLv2*.

9.6.6 Valgrind

O *Valgrind* [Nethercote and Seward, 2007] é uma ferramenta de depuração de uso da memória RAM e problemas correlatos. Ele permite investigar fugas 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.

9.6.7 JVM

É comum a implementação do suporte de execução de uma linguagem de programação usando uma máquina virtual. Um bom exemplo dessa abordagem ocorre na linguagem Java. 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 o torna tão atraente é o fato de programas escritos em Java 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 9.21 mostra os principais componentes da plataforma Java.

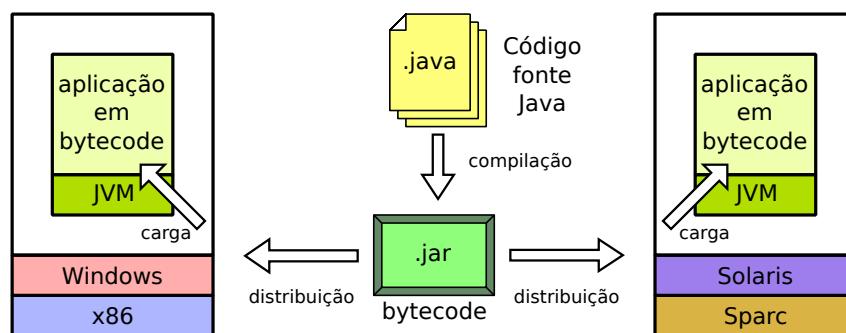


Figura 9.21: Máquina virtual Java.

É importante ressaltar que a adoção de uma máquina virtual como suporte de execução não é exclusividade de 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.

Referências Bibliográficas

- [Amoroso, 1994] Amoroso, E. (1994). *Fundamentals of Computer Security Technology*. Prentice Hall PTR.
- [Anderson et al., 2002] Anderson, D., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. (2002). SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61.
- [Anderson et al., 1992] Anderson, T., Bershad, B., Lazowska, E., and Levy, H. (1992). Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79.
- [Arpaci-Dusseau et al., 2003] Arpaci-Dusseau, A., Arpaci-Dusseau, R., Burnett, N., Denehy, T., Engle, T., Gunawi, H., Nugent, J., and Popovici, F. (2003). Transforming policies into mechanisms with InfoKernel. In *19th ACM Symposium on Operating Systems Principles*.
- [Avizienis et al., 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1).
- [Bach, 1986] Bach, M. J. (1986). *The design of the UNIX operating System*. Prentice-Hall.
- [Badger et al., 1995] Badger, L., Sterne, D., Sherman, D., Walker, K., and Haghhighat, S. (1995). Practical Domain and Type Enforcement for UNIX. In *IEEE Symposium on Security and Privacy*, pages 66–77.
- [Bansal and Modha, 2004] Bansal, S. and Modha, D. (2004). CAR: Clock with adaptive replacement. In *USENIX Conference on File and Storage Technologies*.
- [Barham et al., 2003] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles*, pages 164–177.
- [Barney, 2005] Barney, B. (2005). POSIX threads programming. <http://www.llnl.gov/computing/tutorials/pthreads>.
- [Bell and LaPadula, 1974] Bell, D. E. and LaPadula, L. J. (1974). Secure computer systems. mathematical foundations and model. Technical Report M74-244, MITRE Corporation.

- [Bellard, 2005] Bellard, F. (2005). QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*.
- [Ben-Ari, 1990] Ben-Ari, M. (1990). *Principles of Concurrent and Distributed Programming*. Prentice-Hall.
- [Biba, 1977] Biba, K. (1977). Integrity considerations for secure computing systems. Technical Report MTR-3153, MITRE Corporation.
- [Birrell, 2004] Birrell, A. (2004). Implementing condition variables with semaphores. *Computer Systems Theory, Technology, and Applications*, pages 29–37.
- [Black, 1990] Black, D. L. (1990). Scheduling and resource management techniques for multiprocessors. Technical Report CMU-CS-90-152, Carnegie-Mellon University, Computer Science Dept.
- [Blunden, 2002] Blunden, B. (2002). *Virtual Machine Design and Implementation in C/C++*. Worldware Publishing.
- [Boebert and Kain, 1985] Boebert, W. and Kain, R. (1985). A practical alternative to hierarchical integrity policies. In *8th National Conference on Computer Security*, pages 18–27.
- [Bomberger et al., 1992] Bomberger, A., Frantz, A., Frantz, W., Hardy, A., Hardy, N., Landau, C., and Shapiro, J. (1992). The KeyKOS nanokernel architecture. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112.
- [Bovet and Cesati, 2005] Bovet, D. and Cesati, M. (2005). *Understanding the Linux Kernel, 3rd edition*. O'Reilly Media, Inc.
- [Boyd-Wickizer et al., 2009] Boyd-Wickizer, S., Morris, R., and Kaashoek, M. (2009). Reinventing scheduling for multicore systems. In *12th conference on Hot topics in operating systems*, page 21. USENIX Association.
- [Brown, 2000] Brown, K. (2000). *Programming Windows Security*. Addison-Wesley Professional.
- [Burns and Wellings, 1997] Burns, A. and Wellings, A. (1997). *Real-Time Systems and Programming Languages, 2nd edition*. Addison-Wesley.
- [Carr and Hennessy, 1981] Carr, R. and Hennessy, J. (1981). WSclock - a simple and effective algorithm for virtual memory management. In *ACM symposium on Operating systems principles*.
- [Chen et al., 1994] Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A. (1994). RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26:145–185.
- [Clark et al., 2005] Clark, C., Fraser, K., Hand, S., Hansen, J., Jul, E., Limpach, C., Pratt, I., and Warfield, A. (2005). Live migration of virtual machines. In *Symposium on Networked Systems Design and Implementation*.

- [Coffman et al., 1971] Coffman, E., Elphick, M., and Shoshani, A. (1971). System deadlocks. *ACM Computing Surveys*, 3(2):67–78.
- [Corbató, 1963] Corbató, F. (1963). *The Compatible Time-Sharing System: A Programmer's Guide*. MIT Press.
- [Corbató et al., 1962] Corbató, F., Daggett, M., and Daley, R. (1962). An experimental time-sharing system. In *Proceedings of the Spring Joint Computer Conference*.
- [Corbató and Vyssotsky, 1965] Corbató, F. J. and Vyssotsky, V. A. (1965). Introduction and overview of the Multics system. In *AFIPS Conference Proceedings*, pages 185–196.
- [Corbet et al., 2005] Corbet, J., Rubini, A., and Kroah-Hartman, G. (2005). *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc.
- [Cowan et al., 2000] Cowan, C., Beattie, S., Kroah-Hartman, G., Pu, C., Wagle, P., and Gligor, V. (2000). SubDomain: Parsimonious server security. In *14th USENIX Systems Administration Conference*.
- [Dasgupta et al., 1991] Dasgupta, P., Richard J. LeBlanc, J., Ahamad, M., and Ramachandran, U. (1991). The Clouds distributed operating system. *Computer*, 24(11):34–44.
- [Day, 1983] Day, J. (1983). The OSI reference model. *Proceedings of the IEEE*.
- [Denning, 1980] Denning, P. (1980). Working sets past and present. *IEEE Transactions on Software Engineering*, 6(1):64–84.
- [Denning, 2006] Denning, P. J. (2006). The locality principle. In Barria, J., editor, *Communication Networks and Computer Systems*, chapter 4, pages 43–67. Imperial College Press.
- [di Vimercati et al., 2007] di Vimercati, S., Foresti, S., Jajodia, S., and Samarati, P. (2007). Access control policies and languages in open environments. In Yu, T. and Jajodia, S., editors, *Secure Data Management in Decentralized Systems*, volume 33 of *Advances in Information Security*, pages 21–58. Springer.
- [di Vimercati et al., 2005] di Vimercati, S., Samarati, P., and Jajodia, S. (2005). Policies, Models, and Languages for Access Control. In *Workshop on Databases in Networked Information Systems*, volume LNCS 3433, pages 225–237. Springer-Verlag.
- [Dike, 2000] Dike, J. (2000). A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference*.
- [Dorward et al., 1997] Dorward, S., Pike, R., Presotto, D., Ritchie, D., Trickey, H., and Winterbottom, P. (1997). The Inferno operating system. *Bell Labs Technical Journal*, 2(1):5–18.
- [Downey, 2008] Downey, A. (2008). *The Little Book of Semaphores*. Green Tea Press.
- [Duesterwald, 2005] Duesterwald, E. (2005). Design and engineering of a dynamic binary optimizer. *Proceedings of the IEEE*, 93(2):436–448.

- [Engeschall, 2005] Engeschall, R. (2005). The GNU Portable Threads. <http://www.gnu.org/software/pth>.
- [Evans and Elischer, 2003] Evans, J. and Elischer, J. (2003). Kernel-scheduled entities for FreeBSD. <http://www.aims.net.au/chris/kse>.
- [Farines et al., 2000] Farines, J.-M., da Silva Fraga, J., and de Oliveira, R. S. (2000). *Sistemas de Tempo Real – 12ª Escola de Computação da SBC*. Sociedade Brasileira de Computação.
- [Ford and Susarla, 1996] Ford, B. and Susarla, S. (1996). CPU Inheritance Scheduling. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105.
- [Foundation, 2005] Foundation, W. (2005). Wikipedia online encyclopedia. <http://www.wikipedia.org>.
- [Freed and Borenstein, 1996] Freed, N. and Borenstein, N. (1996). RFC 2046: Multipurpose Internet Mail Extensions (MIME) part two: Media types.
- [Gallmeister, 1994] Gallmeister, B. (1994). *POSIX.4: Programming for the Real World*. O'Reilly Media, Inc.
- [Gnome, 2005] Gnome (2005). Gnome: the free software desktop project. <http://www.gnome.org>.
- [Goldberg, 1973] Goldberg, R. (1973). Architecture of virtual machines. In *AFIPS National Computer Conference*.
- [Goldberg and Mager, 1979] Goldberg, R. and Mager, P. (1979). Virtual machine technology: A bridge from large mainframes to networks of small computers. *IEEE Proceedings Compcon Fall 79*, pages 210–213.
- [Hart, 2004] Hart, J. (2004). *Windows System Programming, 3rd edition*. Addison-Wesley Professional.
- [Holt, 1972] Holt, R. (1972). Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3):179–196.
- [IBM, 2007] IBM (2007). *Power Instruction Set Architecture – Version 2.04*. IBM Corporation.
- [Jain et al., 2004] Jain, A., Ross, A., and Prabhakar, S. (2004). An Introduction to Biometric Recognition. *IEEE Transactions on Circuits and Systems for Video Technology*, 14(1).
- [Jiang and Zhang, 2002] Jiang, S. and Zhang, X. (2002). 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.

- [Johnstone and Wilson, 1999] Johnstone, M. S. and Wilson, P. R. (1999). The memory fragmentation problem: solved? *ACM SIGPLAN Notices*, 34(3):26–36.
- [Jones, 1997] Jones, M. (1997). What really happened on Mars Rover Pathfinder. *ACM Risks-Forum Digest*, 19(49).
- [Kay and Lauder, 1988] Kay, J. and Lauder, P. (1988). A fair share scheduler. *Communications of the ACM*, 31(1):44–55.
- [KDE, 2005] KDE (2005). KDE desktop project. <http://www.kde.org>.
- [Kernighan and Ritchie, 1989] Kernighan, B. and Ritchie, D. (1989). *C: a Linguagem de Programação - Padrão ANSI*. Campus/Elsevier.
- [King et al., 2003] King, S., Dunlap, G., and Chen, P. (2003). Operating system support for virtual machines. In *Proceedings of the USENIX Technical Conference*.
- [Klein et al., 2009] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. (2009). SeL4: Formal verification of an OS kernel. In *22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA.
- [Lamport, 1974] Lamport, L. (1974). A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455.
- [Lampson, 1971] Lampson, B. (1971). Protection. In *5th Princeton Conference on Information Sciences and Systems*. Reprinted in ACM Operating Systems Rev. 8, 1 (Jan. 1974), pp 18-24.
- [Lampson and Redell, 1980] Lampson, B. and Redell, D. (1980). Experience with processes and monitors in Mesa. *Communications of the ACM*.
- [Levine, 2000] Levine, J. (2000). *Linkers and Loaders*. Morgan Kaufmann.
- [Lichtenstein, 1997] Lichtenstein, S. (1997). A review of information security principles. *Computer Audit Update*, 1997(12):9–24.
- [Liedtke, 1996] Liedtke, J. (1996). Toward real microkernels. *Communications of the ACM*, 39(9):70–77.
- [Loscocco and Smalley, 2001] Loscocco, P. and Smalley, S. (2001). Integrating Flexible Support for Security Policies into the Linux Operating System. In *USENIX Annual Technical Conference*, pages 29–42.
- [Love, 2004] Love, R. (2004). *Linux Kernel Development*. Sams Publishing Developer's Library.
- [Mauro and McDougall, 2006] Mauro, J. and McDougall, R. (2006). *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*. Prentice-Hall PTR.

- [McKusick and Neville-Neil, 2005] McKusick, M. and Neville-Neil, G. (2005). *The Design and Implementation of the FreeBSD Operating System*. Pearson Education.
- [Menezes et al., 1996] Menezes, A., Van Oorschot, P., and Vanstone, S. (1996). *Handbook of Applied Cryptography*. CRC Press.
- [Microsoft, 2007] Microsoft (2007). *Security Enhancements in Windows Vista*. Microsoft Corporation.
- [Mitnick and Simon, 2002] Mitnick, K. D. and Simon, W. L. (2002). *The Art of Deception: Controlling the Human Element of Security*. John Wiley & Sons, Inc., New York, NY, USA.
- [Mollin, 2000] Mollin, R. A. (2000). *An Introduction to Cryptography*. CRC Press, Inc., Boca Raton, FL, USA.
- [Nanda and Chiueh, 2005] Nanda, S. and Chiueh, T. (2005). A survey on virtualization technologies. Technical report, University of New York at Stony Brook.
- [Navarro et al., 2002] Navarro, J., Iyer, S., Druschel, P., and Cox, A. (2002). Practical, transparent operating system support for superpages. In *5th USENIX Symposium on Operating Systems Design and Implementation*, pages 89–104.
- [Nethercote and Seward, 2007] Nethercote, N. and Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM Conference on Programming Language Design and Implementation*, San Diego - California - USA.
- [Neuman and Ts'o, 1994] Neuman, B. C. and Ts'o, T. (1994). Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38.
- [Neuman et al., 2005] Neuman, C., Yu, T., Hartman, S., and Raeburn, K. (2005). The Kerberos Network Authentication Service (V5). RFC 4120 (Proposed Standard). Updated by RFCs 4537, 5021.
- [Newman et al., 2005] Newman, M., Wiberg, C.-M., and Braswell, B. (2005). *Server Consolidation with VMware ESX Server*. IBM RedBooks. <http://www.redbooks.ibm.com>.
- [Nichols et al., 1996] Nichols, B., Buttler, D., and Farrell, J. (1996). *PThreads Programming*. O'Reilly Media, Inc.
- [Nieh and Lam, 1997] Nieh, J. and Lam, M. (1997). 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.
- [Patterson and Hennessy, 2005] Patterson, D. and Hennessy, J. (2005). *Organização e Projeto de Computadores*. Campus.
- [Patterson et al., 1988] Patterson, D. A., Gibson, G., and Katz, R. H. (1988). A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD International Conference on Management of Data*, pages 109–116. ACM.

- [Petzold, 1998] Petzold, C. (1998). *Programming Windows, 5th edition*. Microsoft Press.
- [Pfleeger and Pfleeger, 2006] Pfleeger, C. and Pfleeger, S. L. (2006). *Security in Computing, 4th Edition*. Prentice Hall PTR.
- [Pike et al., 1995] Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. (1995). Plan 9 from Bell Labs. *Journal of Computing Systems*, 8(3):221–254.
- [Pike et al., 1993] Pike, R., Presotto, D., Thompson, K., Trickey, H., and Winterbottom, P. (1993). The use of name spaces in Plan 9. *Operating Systems Review*, 27(2):72–76.
- [Popek and Goldberg, 1974] Popek, G. and Goldberg, R. (1974). Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421.
- [Price and Tucker, 2004] Price, D. and Tucker, A. (2004). Solaris zones: Operating system support for consolidating commercial workloads. In *18th USENIX conference on System administration*, pages 241–254.
- [Provos et al., 2003] Provos, N., Friedl, M., and Honeyman, P. (2003). Preventing privilege escalation. In *12th USENIX Security Symposium*.
- [Rashid et al., 1989] Rashid, R., Julin, D., Orr, D., Sanzi, R., Baron, R., Forin, A., Golub, D., and Jones, M. B. (1989). Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON*, pages 176–178, San Francisco, CA, USA. IEEE Comput. Soc. Press.
- [Raynal, 1986] Raynal, M. (1986). *Algorithms for Mutual Exclusion*. The MIT Press.
- [Rice, 2000] Rice, L. (2000). *Introduction to OpenVMS*. Elsevier Science & Technology Books.
- [Robbins and Robbins, 2003] Robbins, K. and Robbins, S. (2003). *UNIX Systems Programming*. Prentice-Hall.
- [Robin and Irvine, 2000] Robin, J. and Irvine, C. (2000). Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *9th USENIX Security Symposium*.
- [Rosenblum, 2004] Rosenblum, M. (2004). The reincarnation of virtual machines. *Queue Focus - ACM Press*, pages 34–40.
- [Rosenblum and Garfinkel, 2005] Rosenblum, M. and Garfinkel, T. (2005). Virtual machine monitors: Current technology and future trends. *IEEE Computer*.
- [Rozier et al., 1992] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaiser, C., Langlois, S., Léonard, P., and Neuhauser, W. (1992). Overview of the Chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle WA (USA).

- [Russell et al., 2004] Russell, R., Quinlan, D., and Yeoh, C. (2004). Filesystem Hierarchy Standard.
- [Russinovich and Solomon, 2004] Russinovich, M. and Solomon, D. (2004). *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000*. Microsoft Press.
- [Saltzer and Schroeder, 1975] Saltzer, J. and Schroeder, M. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278 – 1308.
- [Samarati and De Capitani di Vimercati, 2001] Samarati, P. and De Capitani di Vimercati, S. (2001). Access control: Policies, models, and mechanisms. In Focardi, R. and Gorrieri, R., editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*. Springer-Verlag.
- [Sandhu et al., 1996] Sandhu, R., Coyne, E., Feinstein, H., and Youman, C. (1996). Role-based access control models. *IEEE Computer*, 29(2):38–47.
- [Sandhu and Samarati, 1996] Sandhu, R. and Samarati, P. (1996). Authentication, access control, and audit. *ACM Computing Surveys*, 28(1).
- [Seward and Nethercote, 2005] Seward, J. and Nethercote, N. (2005). Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference*.
- [Sha et al., 1990] Sha, L., Rajkumar, R., and Lehoczky, J. (1990). Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185.
- [Shapiro and Hardy, 2002] Shapiro, J. and Hardy, N. (2002). Eros: a principle-driven operating system from the ground up. *Software, IEEE*, 19(1):26–33.
- [Shirey, 2000] Shirey, R. (2000). RFC 2828: Internet security glossary.
- [Silberschatz et al., 2001] Silberschatz, A., Galvin, P., and Gagne, G. (2001). *Sistemas Operacionais – Conceitos e Aplicações*. Campus.
- [Smith and Nair, 2004] Smith, J. and Nair, R. (2004). *Virtual Machines: Architectures, Implementations and Applications*. Morgan Kaufmann.
- [SNIA, 2009] SNIA (2009). *Common RAID Disk Data Format Specification*. SNIA – Storage Networking Industry Association. Version 2.0 Revision 19.
- [Spencer et al., 1999] Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., and Lepreau, J. (1999). The Flask security architecture: System support for diverse security policies. In *8th USENIX Security Symposium*, pages 123–139.
- [Stevens, 1998] Stevens, R. (1998). *UNIX Network Programming*. Prentice-Hall.

- [Sugerman et al., 2001] Sugerman, J., Venkitachalam, G., and Lim, B. H. (2001). Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, pages 1–14.
- [Sun Microsystems, 2000] Sun Microsystems (2000). *Trusted Solaris User's Guide*. Sun Microsystems, Inc.
- [Tanenbaum, 2003] Tanenbaum, A. (2003). *Sistemas Operacionais Modernos, 2^a edição*. Pearson – Prentice-Hall.
- [Tanenbaum et al., 1991] Tanenbaum, A., Kaashoek, M., van Renesse, R., and Bal, H. (1991). The Amoeba distributed operating system – a status report. *Computer Communications*, 14:324–335.
- [Tripwire, 2003] Tripwire (2003). The Tripwire open source project. <http://www.tripwire.org>.
- [Uhlig and Mudge, 1997] Uhlig, R. and Mudge, T. (1997). Trace-driven memory simulation: a survey. *ACM Computing Surveys*, 29(2):128–170.
- [Uhlig et al., 2005] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A., Martins, F., Anderson, A., Bennett, S., Kägi, A., Leung, F., and Smith, L. (2005). Intel virtualization technology. *IEEE Computer*.
- [Ung and Cifuentes, 2006] Ung, D. and Cifuentes, C. (2006). Dynamic re-engineering of binary code with run-time feedbacks. *Science of Computer Programming*, 60(2):189–204.
- [Vahalia, 1996] Vahalia, U. (1996). *UNIX Internals – The New Frontiers*. Prentice-Hall.
- [VirtualBox, 2008] VirtualBox, I. (2008). The VirtualBox architecture. http://www.virtualbox.org/wiki/VirtualBox_architecture.
- [VMware, 2000] VMware (2000). VMware technical white paper. Technical report, VMware, Palo Alto, CA - USA.
- [Watson, 2001] Watson, R. (2001). TrustedBSD: Adding trusted operating system features to FreeBSD. In *USENIX Technical Conference*.
- [Whitaker et al., 2002] Whitaker, A., Shaw, M., and Gribble, S. (2002). Denali: A scalable isolation kernel. In *ACM SIGOPS European Workshop*.
- [Yen, 2007] Yen, C.-H. (2007). Solaris operating system - hardware virtualization product architecture. Technical Report 820-3703-10, Sun Microsystems.

Apêndice A

O *Task Control Block* do Linux

A estrutura em linguagem C apresentada a seguir constitui o descritor de tarefas (*Task Control Block*) do Linux (estudado na Seção 2.4.1). Ela foi extraída do arquivo `include/linux/sched.h` do código-fonte do núcleo Linux 2.6.12 (o arquivo inteiro contém mais de 1.200 linhas de código em C).

```
1  struct task_struct {
2      volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
3      struct thread_info *thread_info;
4      atomic_t usage;
5      unsigned long flags; /* per process flags, defined below */
6      unsigned long ptrace;
7
8      int lock_depth;      /* BKL lock depth */
9
10     int prio, static_prio;
11     struct list_head run_list;
12     prio_array_t *array;
13
14     unsigned long sleep_avg;
15     unsigned long long timestamp, last_ran;
16     unsigned long long sched_time; /* sched_clock time spent running */
17     int activated;
18
19     unsigned long policy;
20     cpumask_t cpus_allowed;
21     unsigned int time_slice, first_time_slice;
22
23 #ifdef CONFIG_SCHEDSTATS
24     struct sched_info sched_info;
25 #endif
26
27     struct list_head tasks;
28     /*
29      * ptrace_list/ptrace_children forms the list of my children
30      * that were stolen by a ptracer.
31      */
32     struct list_head ptrace_children;
33     struct list_head ptrace_list;
```

```

35     struct mm_struct *mm, *active_mm;
36
37 /* task state */
38     struct linux_binfmt *binfmt;
39     long exit_state;
40     int exit_code, exit_signal;
41     int pdeath_signal; /* The signal sent when the parent dies */
42     /* ??? */
43     unsigned long personality;
44     unsigned did_exec:1;
45     pid_t pid;
46     pid_t tgid;
47     /*
48      * pointers to (original) parent process, youngest child, younger sibling,
49      * older sibling, respectively. (p->father can be replaced with
50      * p->parent->pid)
51      */
52     struct task_struct *real_parent; /* real parent process (when being debugged) */
53     struct task_struct *parent; /* parent process */
54     /*
55      * children/sibling forms the list of my children plus the
56      * tasks I'm ptracing.
57      */
58     struct list_head children; /* list of my children */
59     struct list_head sibling; /* linkage in my parent's children list */
60     struct task_struct *group_leader; /* threadgroup leader */
61
62     /* PID/PID hash table linkage. */
63     struct pid pids[PIDTYPE_MAX];
64
65     struct completion *vfork_done; /* for vfork() */
66     int __user *set_child_tid; /* CLONE_CHILD_SETTID */
67     int __user *clear_child_tid; /* CLONE_CHILD_CLEARTID */
68
69     unsigned long rt_priority;
70     cputime_t utime, stime;
71     unsigned long nvcsw, nivcsw; /* context switch counts */
72     struct timespec start_time;
73 /* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
74     unsigned long min_flt, maj_flt;
75
76     cputime_t it_prof_expires, it_virt_expires;
77     unsigned long long it_sched_expires;
78     struct list_head cpu_timers[3];
79
80 /* process credentials */
81     uid_t uid,euid,suid,fsuid;
82     gid_t gid,egid,sgid,fsgid;
83     struct group_info *group_info;
84     kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
85     unsigned keep_capabilities:1;
86     struct user_struct *user;
87 #ifdef CONFIG_KEYS
88     struct key *thread_keyring; /* keyring private to this thread */
89 #endif

```

```

90     int oomkilladj; /* OOM kill score adjustment (bit shift). */
91     char comm[TASK_COMM_LEN]; /* executable name excluding path
92                             - access with [gs]et_task_comm (which lock
93                             it with task_lock())
94                             - initialized normally by flush_old_exec */
95 /* file system info */
96     int link_count, total_link_count;
97 /* ipc stuff */
98     struct sysv_sem sysvsem;
99 /* CPU-specific state of this task */
100    struct thread_struct thread;
101 /* filesystem information */
102    struct fs_struct *fs;
103 /* open file information */
104    struct files_struct *files;
105 /* namespace */
106    struct namespace *namespace;
107 /* signal handlers */
108    struct signal_struct *signal;
109    struct sighand_struct *sighand;
110
111    sigset_t blocked, real_blocked;
112    struct sigpending pending;
113
114    unsigned long sas_ss_sp;
115    size_t sas_ss_size;
116    int (*notifier)(void *priv);
117    void *notifier_data;
118    sigset_t *notifier_mask;
119
120    void *security;
121    struct audit_context *audit_context;
122    seccomp_t seccomp;
123
124 /* Thread group tracking */
125     u32 parent_exec_id;
126     u32 self_exec_id;
127 /* Protection of (de-)allocation: mm, files, fs, tty, keyrings */
128     spinlock_t alloc_lock;
129 /* Protection of proc_dentry: nesting proc_lock, dcache_lock, write_lock_irq(&tasklist_lock); */
130     spinlock_t proc_lock;
131 /* context-switch lock */
132     spinlock_t switch_lock;
133
134 /* journalling filesystem info */
135     void *journal_info;
136
137 /* VM state */
138     struct reclaim_state *reclaim_state;
139
140     struct dentry *proc_dentry;
141     struct backing_dev_info *backing_dev_info;
142
143     struct io_context *io_context;
144

```

```
145     unsigned long ptrace_message;
146     siginfo_t *last_siginfo; /* For ptrace use. */
147 /*
148  * current io wait handle: wait queue entry to use for io waits
149  * If this thread is processing aio, this points at the waitqueue
150  * inside the currently handled kiocb. It may be NULL (i.e. default
151  * to a stack based synchronous wait) if its doing sync IO.
152 */
153     wait_queue_t *io_wait;
154 /* i/o counters(bytes read/written, #syscalls */
155     u64 rchar, wchar, syscr, syscw;
156 #if defined(CONFIG_BSD_PROCESS_ACCT)
157     u64 acct_rss_mem1; /* accumulated rss usage */
158     u64 acct_vm_mem1; /* accumulated virtual memory usage */
159     clock_t acct_stimexpd; /* clock_t-converted stime since last update */
160 #endif
161 #ifdef CONFIG_NUMA
162     struct mempolicy *mempolicy;
163     short il_next;
164 #endif
165 #ifdef CONFIG_CPUSETS
166     struct cpuset *cpuset;
167     nodemask_t mems_allowed;
168     int cpuset_mems_generation;
169 #endif
170 };
```