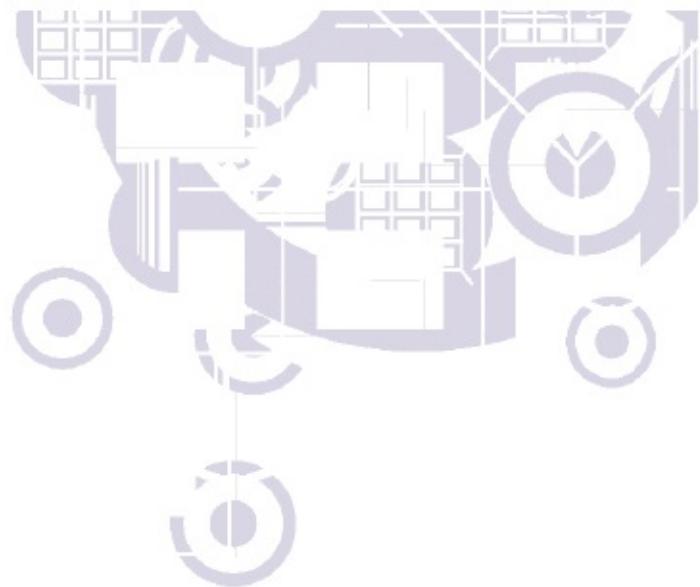


FRANCIS BERENGER MACHADO
LUIZ PAULO MAIA

ARQUITETURA DE SISTEMAS OPERACIONAIS

**INCLUINDO EXERCÍCIOS COM O SIMULADOR SOSIM
E QUESTÕES DO ENADE**





Arquitetura de Sistemas Operacionais



Respeite o direito autoral.

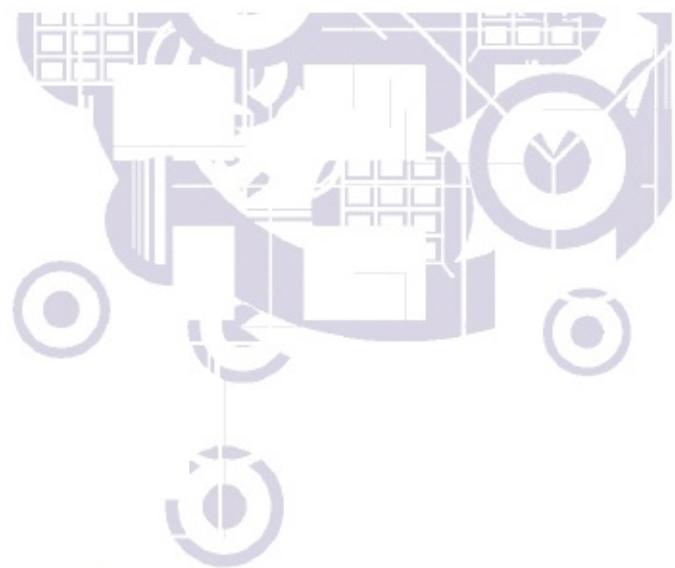


O GEN | Grupo Editorial Nacional reúne as editoras Guanabara Koogan, Santos, Roca, AC Farmacêutica, Forense, Método, LTC, E.P.U. e Forense Universitária, que publicam nas áreas científica, técnica e profissional.

Essas empresas, respeitadas no mercado editorial, construíram catálogos inigualáveis, com obras que têm sido decisivas na formação acadêmica e no aperfeiçoamento de várias gerações de profissionais e de estudantes de Administração, Direito, Enfermagem, Engenharia, Fisioterapia, Medicina, Odontologia, Educação Física e muitas outras ciências, tendo se tornado sinônimo de seriedade e respeito.

Nossa missão é prover o melhor conteúdo científico e distribuí-lo de maneira flexível e conveniente, a preços justos, gerando benefícios e servindo a autores, docentes, livreiros, funcionários, colaboradores e acionistas.

Nosso comportamento ético incondicional e nossa responsabilidade social e ambiental são reforçados pela natureza educacional de nossa atividade, sem comprometer o crescimento contínuo e a rentabilidade do grupo.



Arquitetura de Sistemas Operacionais

5ª edição

Francis Berenger Machado (berenger@pobox.com)

Mestre em Administração de Empresas na PUC-Rio, graduado em Engenharia Eletrônica pelo CEFET-RJ e em Informática e Psicologia pela PUC-Rio. Atualmente é sócio da Descartes Consultoria (www.descartes-consultoria.com.br), além de lecionar na PUC-Rio e IBMEC. É autor do livro *Fundamentos de Sistemas Operacionais*.

Luiz Paulo Maia (lpmaia13@gmail.com)

Mestre em Informática pela UFRJNCE, graduado em Informática pela PUC-Rio e pós-graduado em Marketing pela mesma instituição. Atualmente é professor no Instituto Infnet e consultor na área de sistemas operacionais, redes e segurança. É autor dos livros *Arquitetura de Redes de Computadores* e *Fundamentos de Sistemas Operacionais*, e autor do software SOsim.



Os autores e a editora empenharam-se para citar adequadamente e dar o devido crédito a todos os detentores dos direitos autorais de qualquer material utilizado neste livro, dispondo-se a possíveis acertos caso, inadvertidamente, a identificação de algum deles tenha sido omitida.

Não é responsabilidade da editora nem dos autores a ocorrência de eventuais perdas ou danos a pessoas ou bens que tenham origem no uso desta publicação.

Apesar dos melhores esforços dos autores, do editor e dos revisores, é inevitável que surjam erros no texto. Assim, são bem-vindas as comunicações de usuários sobre correções ou sugestões referentes ao conteúdo ou ao nível pedagógico que auxiliem o aprimoramento de edições futuras. Os comentários dos leitores podem ser encaminhados à **LTC – Livros Técnicos e Científicos Editora** pelo e-mail ltc@grupogen.com.br.

Direitos exclusivos para a língua portuguesa

Copyright © 2013 by Francis Berenger Machado e Luiz Paulo Maia

LTC — Livros Técnicos e Científicos Editora Ltda.

Uma editora integrante do GEN | Grupo Editorial Nacional

Travessa do Ouvidor, 11

Rio de Janeiro, RJ — CEP 20040-040

Tels.: 21-3543-0770 / 11-5080-0770

Fax: 21-3543-0896

ltc@grupogen.com.br

www.ltceditora.com.br

Capa: Leonidas Leite

Produção Digital: Geethik

1^a edição: 1992 – Reimpressões: 1993 (três), 1994, 1995 e 1996

2^a edição: 1997 – Reimpressões: 1997, 1998, 1999 (duas) e 2000

3^a edição: 2002 – Reimpressões: 2004 (duas)

4^a edição: 2007 – Reimpressões: 2008, 2009, 2010 (duas), 2011 e 2012

5^a edição: 2013 – Reimpressões: 2014

**CIP-BRASIL. CATALOGAÇÃO-NA-FONTE
SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ**

M131a

5. ed.

Machado, Francis B. (Francis Berenger)

Arquitetura de sistemas operacionais / Francis Berenger Machado, Luiz Paulo Maia. - 5. ed. - [Reimpr.]. - Rio de Janeiro : LTC, 2014.

il. ; 28 cm

Inclui bibliografia e índice

ISBN 978-85-216-2287-1

1. Sistemas operacionais (Computação). I. Maia, Luiz Paulo. II. Título.

12-8605.

CDD: 005.43

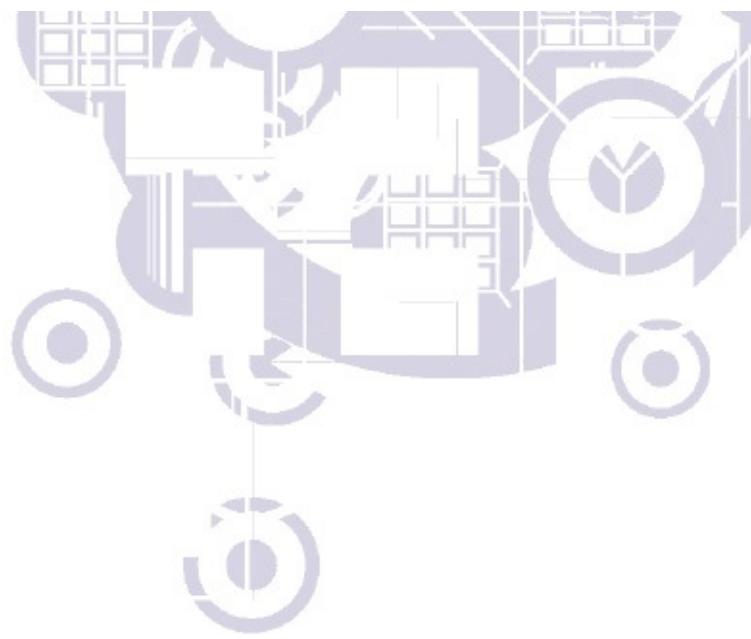
CDU: 004.451

*A Maria Luiza e Felipe
Luiz Paulo Maia*

*Aos meus pais, pela educação e instrução,
à Andréa, à minha filha Tess
e à Amora, que chegou com sua doçura para deixar
minha vida com mais sorrisos.*

Francis Berenger Machado

Prefácio



Este livro aborda a arquitetura e o funcionamento dos sistemas operacionais multiprogramáveis, de forma atual, abrangente e didática. Seu conteúdo é direcionado a estudantes e a profissionais de informática de todas as áreas. Como pré-requisitos básicos para sua leitura, é necessário apenas algum conhecimento de organização de computadores e estrutura de dados. O livro pode ser utilizado integralmente em disciplinas universitárias de graduação ou parcialmente em cursos de extensão.

O estudo de sistemas operacionais ganha importância à medida que diversos profissionais da área de computação necessitam deste conhecimento para exercer suas atividades, como administradores de sistemas, programadores de aplicações concorrentes e gerentes de segurança. Outro fator importante é o relacionamento dos sistemas operacionais nas redes de comunicação de dados, o que faz seu estudo necessário para os administradores de rede.

O texto não foi baseado em nenhuma arquitetura específica, mas são utilizados, como exemplos, sistemas operacionais já consagrados. No website do livro (www.ltceditora.com.br) são apresentados estudos de caso, onde é possível encontrar uma descrição da arquitetura dos sistemas operacionais MS Windows, Unix e OpenVMS.

Alguns termos originais do inglês foram conservados sem tradução de modo a não prejudicar a legibilidade do texto e mantê-lo padronizado com a literatura internacional. Sempre que for necessário o uso de programas será utilizada a linguagem de programação Pascal, por sua clareza e fluência no meio acadêmico, assim como a linguagem Java, por sua ampla aceitação no mercado. Por vezes, quando a estrutura da linguagem não permitir certas construções, uma sintaxe não convencional será utilizada, a fim de facilitar a legibilidade e compreensão dos exemplos.

► Estrutura do Livro

O livro está dividido em três partes e um anexo:

- A Parte I oferece uma visão geral de um sistema operacional, definindo conceitos que frequentemente são referenciados no decorrer do livro.
- A Parte II apresenta os conceitos de processos e threads, além dos mecanismos de comunicação e sincronização utilizados.
- A Parte III descreve os principais subsistemas de gerência de recursos que compõem um sistema operacional como processador, memória e dispositivos de entrada e saída.
- O Anexo contém questões sobre sistemas operacionais do ENADE (Exame Nacional de Desempenho de Estudante) dos cursos de Análise e Desenvolvimento de Sistemas e Ciência da Computação de diversos anos.

► Nova Edição

Nesta nova edição houve uma detalhada revisão técnica da edição anterior. Alguns exemplos de código foram adicionados na linguagem Java. Os estudos de caso sobre MS Windows e Unix foram retirados da versão impressa e disponibilizados no website do livro. Além disso, foi incluído um Anexo contendo questões do ENADE sobre sistemas operacionais.

A seguir são apresentadas as principais alterações desta quinta edição:

- No Capítulo 1 – Visão Geral: o item “Histórico” foi atualizado.
- No Capítulo 2 – Conceitos de Hardware e Software: foram inseridas informações sobre ciclo de busca e instrução no item “Hardware”.
- No Capítulo 3 – Concorrência: foram introduzidos exemplos em Java para exemplificar alguns assuntos.
- No Capítulo 4 – Estrutura do Sistema Operacional: foram apresentados os conceitos de chamadas a rotinas do sistema operacional de forma explícita e implícita no item “Rotinas dos Sistemas Operacionais e System Calls”. Além disso, o item “Arquitetura do Núcleo” foi ampliado com maior conteúdo sobre máquina virtual.
- No Capítulo 5 – Processo: foram introduzidos exemplos em Java para exemplificar alguns assuntos.
- No Capítulo 6 – Thread: foram introduzidos exemplos em Java para exemplificar alguns assuntos.
- Os Capítulos 7, 10 e 11 sofreram pequenas atualizações.
- Os Capítulos 14 e 15 foram retirados da publicação e disponibilizados no website do livro.
- Inclusão de um Anexo com questões do ENADE sobre sistemas operacionais.

► Website e Facebook

No website www.ltceditora.com.br estarão disponíveis diversas informações adicionais sobre o livro. Nosso objetivo é colaborar com professores e alunos para tornar o ensino e aprendizado da disciplina de sistemas operacionais mais fácil e agradável. O website do livro oferece:

- Slides em MS-PowerPoint para apoio de aulas;
- Soluções dos exercícios propostos;

- Estudos de caso dos sistemas operacionais Microsoft Windows, Unix e OpenVMS;
- Errata.

Há uma página do livro no Facebook em que os autores estarão publicando informações relevantes sobre a área de computação e sistemas operacionais.

► Simulador SOsim

Alguns capítulos do livro apresentam laboratórios que fazem uso do simulador SOsim como os que tratam de processos, gerência do processador e gerência de memória virtual. O simulador foi desenvolvido pelo prof. Luiz Paulo Maia como parte de sua tese de mestrado no Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro, defendida em 2001 e orientada pelo prof. Ageu Pacheco. O objetivo do trabalho foi desenvolver uma ferramenta gratuita que permitisse facilitar e melhorar as aulas de sistemas operacionais para alunos e professores.

O SOsim permite que o professor apresente os conceitos e mecanismos de um sistema operacional multitarefa de forma simples e animada. O simulador permite visualizar os conceitos de multiprogramação, processo e suas mudanças de estado, gerência do processador e a gerência memória virtual. A partir das opções de configuração, é possível selecionar diferentes políticas e alterar o funcionamento do simulador. Mais informações sobre o simulador podem ser obtidas no website www.training.com.br/sosim.

► Agradecimentos

Aos alunos, que sempre nos proporcionam um aprendizado contínuo e motivação para publicar mais uma edição revisada deste projeto.

À Editora LTC, mais uma vez pelo apoio dado à realização desta obra.

Material Suplementar

Este livro conta com os seguintes materiais suplementares:

- Ilustrações da obra em formato de apresentação (restrito a docentes)
- Soluções dos exercícios propostos (acesso livre)
- Estudo de caso dos sistemas operacionais MS Windows, Unix e Open VMS (acesso livre)

O acesso ao material suplementar é gratuito, bastando que o leitor se cadastre em: <http://genio.grupogen.com.br>.



GEN-IO (GEN | Informação Online) é o repositório de materiais suplementares e de serviços relacionados com livros publicados pelo GEN | Grupo Editorial Nacional, maior conglomerado brasileiro de editoras do ramo científico-técnico-profissional, composto por Guanabara Koogan, Santos, Roca, AC Farmacêutica, Forense, Método, LTC, E.P.U. e Forense Universitária.

Os materiais suplementares ficam disponíveis para acesso durante a vigência das edições atuais dos livros a que eles correspondem.

Sumário



Parte I Conceitos Básicos

Capítulo 1 Visão Geral

- 1.1** Introdução
- 1.2** Funções Básicas
- 1.3** Máquina de Camadas
- 1.4** Histórico
 - 1.4.1** Década de 1940
 - 1.4.2** Década de 1950
 - 1.4.3** Década de 1960
 - 1.4.4** Década de 1970
 - 1.4.5** Década de 1980
 - 1.4.6** Década de 1990
 - 1.4.7** Década de 2000
 - 1.4.8** Década de 2010
- 1.5** Tipos de Sistemas Operacionais
 - 1.5.1** Sistemas Monoprogramáveis/Monotarefa
 - 1.5.2** Sistemas Multiprogramáveis/Multitarefa
 - 1.5.3** Sistemas com Múltiplos Processadores
- 1.6** Exercícios

Capítulo 2 Conceitos de Hardware e Software

- 2.1** Introdução
- 2.2** Hardware
 - 2.2.1** Processador
 - 2.2.2** Memória Principal
 - 2.2.3** Memória Cache
 - 2.2.4** Memória Secundária
 - 2.2.5** Dispositivos de Entrada e Saída
 - 2.2.6** Barramento

- 2.2.7** Pipelining
- 2.2.8** Arquiteturas RISC e CISC

2.3 Software

- 2.3.1** Tradutor
- 2.3.2** Interpretador
- 2.3.3** Linker
- 2.3.4** Loader
- 2.3.5** Depurador

2.4 Exercícios

Capítulo 3 Concorrência

- 3.1** Introdução
- 3.2** Sistemas Monoprogramáveis × Multiprogramáveis
- 3.3** Interrupções e Exceções
- 3.4** Operações de Entrada/Saída
- 3.5** Buffering
- 3.6** Spooling
- 3.7** Reentrância
- 3.8** Exercícios

Capítulo 4 Estrutura do Sistema Operacional

- 4.1** Introdução
- 4.2** Funções do Núcleo
- 4.3** Modo de Acesso
- 4.4** Rotinas do Sistema Operacional e System Calls
- 4.5** Chamada a Rotinas do Sistema Operacional
- 4.6** Linguagem de Comandos
- 4.7** Ativação/Desativação do Sistema
- 4.8** Arquiteturas do Núcleo
 - 4.8.1** Arquitetura Monolítica
 - 4.8.2** Arquitetura de Camadas
 - 4.8.3** Máquina Virtual
 - 4.8.4** Arquitetura Microkernel
- 4.9** Exercícios

Parte II Processos e Threads

Capítulo 5 Processo

- 5.1** Introdução
- 5.2** Estrutura do Processo
 - 5.2.1** Contexto de Hardware
 - 5.2.2** Contexto de Software
 - 5.2.3** Espaço de Endereçamento
 - 5.2.4** Bloco de Controle do Processo
- 5.3** Estados do Processo
- 5.4** Mudanças de Estado do Processo
- 5.5** Criação e Eliminação de Processos
- 5.6** Processos CPU-bound e I/O-bound
- 5.7** Processos Foreground e Background

- 5.8** Formas de Criação de Processos
- 5.9** Processos Independentes, Subprocessos e Threads
- 5.10** Processos do Sistema Operacional
- 5.11** Sinais
- 5.12** Exercícios
- 5.13** Laboratório com o Simulador SOsim

Capítulo 6 Thread

- 6.1** Introdução
- 6.2** Ambiente Monothread
- 6.3** Ambiente Multithread
- 6.4** Programação Multithread
- 6.5** Arquitetura e Implementação
 - 6.5.1** Threads em Modo Usuário
 - 6.5.2** Threads em Modo Kernel
 - 6.5.3** Threads em Modo Híbrido
 - 6.5.4** Scheduler Activations
- 6.6** Exercícios

Capítulo 7 Sincronização e Comunicação entre Processos

- 7.1** Introdução
- 7.2** Aplicações Concorrentes
- 7.3** Especificação de Concorrência em Programas
- 7.4** Problemas de Compartilhamento de Recursos
- 7.5** Exclusão Mútua
 - 7.5.1** Soluções de Hardware
 - 7.5.2** Soluções de Software
- 7.6** Sincronização Condicional
- 7.7** Semáforos
 - 7.7.1** Exclusão Mútua Utilizando Semáforos
 - 7.7.2** Sincronização Condicional Utilizando Semáforos
 - 7.7.3** Problema dos Filósofos
 - 7.7.4** Problema do Barbeiro
- 7.8** Monitores
 - 7.8.1** Exclusão Mútua Utilizando Monitores
 - 7.8.2** Sincronização Condicional Utilizando Monitores
- 7.9** Troca de Mensagens
- 7.10** Deadlock
 - 7.10.1** Prevenção de Deadlock
 - 7.10.2** Detecção do Deadlock
 - 7.10.3** Correção do Deadlock
- 7.11** Exercícios

Parte III Gerência de Recursos

Capítulo 8 Gerência do Processador

- 8.1** Introdução
- 8.2** Funções Básicas
- 8.3** Critérios de Escalonamento

- 8.4** Escalonamentos Não Preemptivos e Preemptivos
- 8.5** Escalonamento First-In-First-Out (FIFO)
- 8.6** Escalonamento Shortest-Job-First (SJF)
- 8.7** Escalonamento Cooperativo
- 8.8** Escalonamento Circular
- 8.9** Escalonamento por Prioridades
- 8.10** Escalonamento Circular com Prioridades
- 8.11** Escalonamento por Múltiplas Filas
- 8.12** Escalonamento por Múltiplas Filas com Realimentação
- 8.13** Política de Escalonamento em Sistemas de Tempo Compartilhado
- 8.14** Política de Escalonamento em Sistemas de Tempo Real
- 8.15** Exercícios
- 8.16** Laboratório com o Simulador SOsim

Capítulo 9 Gerência de Memória

- 9.1** Introdução
- 9.2** Funções Básicas
- 9.3** Alocação Contígua Simples
- 9.4** Técnica de Overlay
- 9.5** Alocação Particionada
 - 9.5.1** Alocação Particionada Estática
 - 9.5.2** Alocação Particionada Dinâmica
 - 9.5.3** Estratégias de Alocação de Partição
- 9.6** Swapping
- 9.7** Exercícios

Capítulo 10 Gerência de Memória Virtual

- 10.1** Introdução
- 10.2** Espaço de Endereçamento Virtual
- 10.3** Mapeamento
- 10.4** Memória Virtual por Paginação
 - 10.4.1** Políticas de Busca de Páginas
 - 10.4.2** Políticas de Alocação de Páginas
 - 10.4.3** Políticas de Substituição de Páginas
 - 10.4.4** Working Set
 - 10.4.5** Algoritmos de Substituição de Páginas
 - 10.4.6** Tamanho de Página
 - 10.4.7** Paginação em Múltiplos Níveis
 - 10.4.8** Translation Lookaside Buffer
 - 10.4.9** Proteção de Memória
 - 10.4.10** Compartilhamento de Memória
- 10.5** Memória Virtual por Segmentação
- 10.6** Memória Virtual por Segmentação com Paginação
- 10.7** Swapping em Memória Virtual
- 10.8** Thrashing
- 10.9** Exercícios
- 10.10** Laboratório com o Simulador SOsim

Capítulo 11 Sistema de Arquivos

- 11.1** Introdução

11.2 Arquivos

- 11.2.1 Organização de Arquivos**
- 11.2.2 Métodos de Acesso**
- 11.2.3 Operações de Entrada/Saída**
- 11.2.4 Atributos**

11.3 Diretórios

- 11.4 Gerência de Espaço Livre em Disco**
- 11.5 Gerência de Alocação de Espaço em Disco**

- 11.5.1 Alocação Contígua**
- 11.5.2 Alocação Encadeada**
- 11.5.3 Alocação Indexada**

11.6 Proteção de Acesso

- 11.6.1 Senha de Acesso**
- 11.6.2 Grupos de Usuário**
- 11.6.3 Lista de Controle de Acesso**

11.7 Implementação de Caches

11.8 Exercícios

Capítulo 12 Gerência de Dispositivos

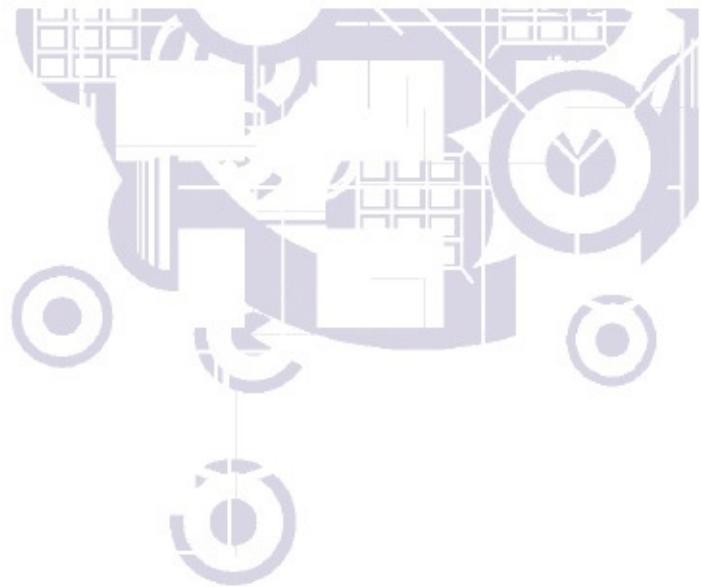
- 12.1 Introdução**
- 12.2 Subsistema de Entrada e Saída**
- 12.3 Device Driver**
- 12.4 Controlador de Entrada e Saída**
- 12.5 Dispositivos de Entrada e Saída**
- 12.6 Discos Magnéticos**
 - 12.6.1 Desempenho, Redundância e Proteção de Dados**
- 12.7 Exercícios**

Capítulo 13 Sistemas com Múltiplos Processadores

- 13.1 Introdução**
- 13.2 Vantagens e Desvantagens**
- 13.3 Tipos de Sistemas Computacionais**
- 13.4 Sistemas Fortemente e Fracamente Acoplados**
- 13.5 Sistemas com Multiprocessadores Simétricos**
 - 13.5.1 Evolução dos Sistemas Simétricos**
 - 13.5.2 Arquitetura dos Sistemas Simétricos**
- 13.6 Sistemas NUMA**
- 13.7 Clusters**
- 13.8 Sistemas Operacionais de Rede**
- 13.9 Sistemas Distribuídos**
 - 13.9.1 Transparência**
 - 13.9.2 Tolerância a Falhas**
 - 13.9.3 Imagem Única do Sistema**
- 13.10 Exercícios**

Anexo – Questões do ENADE

Bibliografia



*Arquitetura de Sistemas
Operacionais*



Parte I

Conceitos Básicos

“Não basta ensinar ao homem uma especialidade, porque se tornará assim uma máquina utilizável, mas não uma personalidade. É necessário que adquira um sentimento, um senso prático daquilo que vale a pena ser empreendido, daquilo que é belo, do que é moralmente correto. A não ser assim, ele se assemelhará, com seus conhecimentos profissionais, mais a um cão ensinado do que a uma criatura harmoniosamente desenvolvida. Deve aprender a compreender as motivações dos homens, suas quimeras e suas angústias, para determinar com exatidão seu lugar preciso em relação a seus próximos e à comunidade.”

Albert Einstein (1879-1955)
Físico alemão

*“Se plantarmos para um ano, devemos plantar cereais.
Se plantarmos para uma década, devemos plantar árvores.
Se plantarmos para toda a vida, devemos treinar e educar o homem.”*

Kwantsu (séc. III a.C.)

“Eduai as crianças e não será preciso punir os homens.”

Abraham Lincoln (1809-1865)
Presidente americano

“Ninguém nasce feito, ninguém nasce marcado para ser isso ou aquilo. Pelo contrário, nos tornamos isso ou aquilo. Somos programados, mas para aprender. A nossa inteligência se inventa e se promove no exercício social de nosso corpo consciente. Se constrói. Não é um dado que, em nós, seja um a priori da nossa história individual e social.”

Paulo Freire (1921-1997)
Educador brasileiro

“O que não conhecemos, não nos pertence.”

Goethe (1749-1832)
Escritor alemão



Visão Geral

► 1.1 Introdução

Um *sistema operacional*, por mais complexo que possa parecer, é apenas um conjunto de rotinas executado pelo processador, de forma semelhante aos programas dos usuários. Sua principal função é controlar o funcionamento de um computador, gerenciando a utilização e o compartilhamento dos seus diversos recursos, como processadores, memórias e dispositivos de entrada e saída.

Sem o sistema operacional, um usuário para interagir com o computador deveria conhecer profundamente diversos detalhes sobre hardware do equipamento, o que tornaria seu trabalho lento e com grandes possibilidades de erros. O sistema operacional tem como objetivo funcionar como uma interface entre o usuário e o computador, tornando sua utilização mais simples, rápida e segura.

A grande diferença entre um sistema operacional e aplicações convencionais é a maneira como suas rotinas são executadas em função do tempo. Um sistema operacional não é executado de forma linear como na maioria das aplicações, com início, meio e fim. Suas rotinas são executadas concorrentemente em função de eventos assíncronos, ou seja, eventos que podem ocorrer a qualquer momento.

O nome sistema operacional, apesar de ser o mais empregado atualmente, não é o único para designar esse conjunto de rotinas. Denominações como monitor, executivo, supervisor ou controlador possuem, normalmente, o mesmo significado.

Neste capítulo serão apresentadas as funções básicas de um sistema operacional, o conceito de máquina de camadas, um histórico da evolução dos sistemas operacionais e seus diferentes tipos e classificações.

► 1.2 Funções Básicas

Um sistema operacional possui inúmeras funções, mas antes de começar o estudo dos conceitos e dos seus principais componentes é importante saber primeiramente quais são suas funções básicas. Nesta

introdução, as funções de um sistema operacional são resumidas em duas, descritas a seguir:

■ **Facilidade de acesso aos recursos do sistema**

Um computador ou sistema computacional possui diversos dispositivos, como monitores de vídeo, impressoras, unidades de CD/DVD, discos e fitas magnéticas. Quando utilizamos um desses dispositivos, não nos preocupamos com a maneira como é realizada esta comunicação e os inúmeros detalhes envolvidos nas operações de leitura e gravação.

Para a maioria dos usuários, uma operação como a leitura de um arquivo em disco pode parecer simples. Na realidade, existe um conjunto de rotinas específicas, controladas pelo sistema operacional, responsável pelo acionamento do mecanismo de leitura e gravação da unidade de disco, posicionamento na trilha e setor corretos, transferência dos dados para a memória e, finalmente, informar ao programa a conclusão da operação. Cabe, então, ao sistema operacional servir de interface entre os usuários e os recursos disponíveis no sistema computacional, tornando esta comunicação transparente, além de permitir um trabalho mais eficiente e com menores chances de erros. Este conceito de ambiente simulado, criado pelo sistema operacional, é denominado máquina virtual e está presente na maioria dos sistemas modernos (Fig. 1.1).

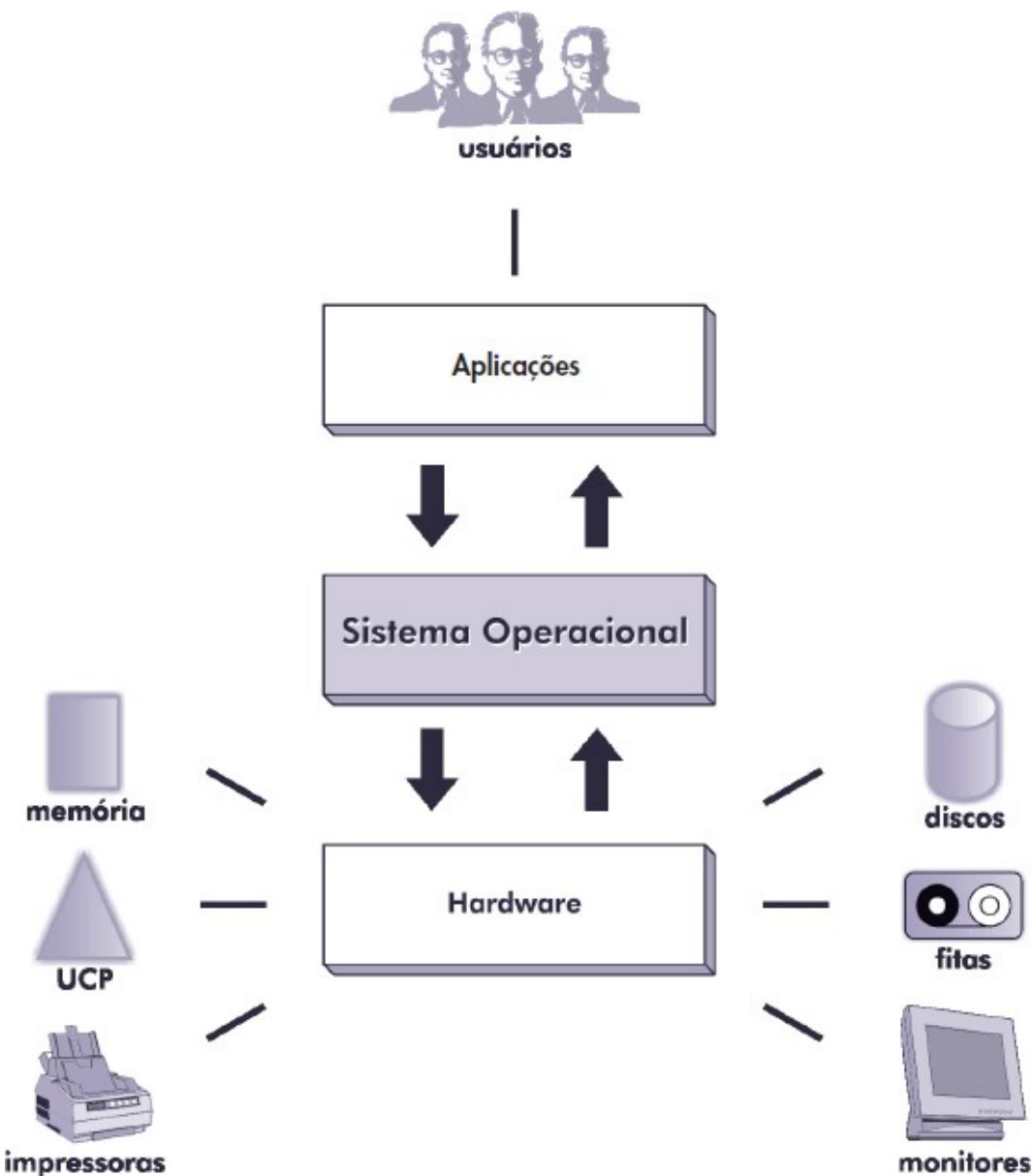


Fig. 1.1 Visão do sistema operacional.

É comum pensar-se que compiladores, linkers, bibliotecas, depuradores e outras ferramentas fazem parte do sistema operacional, mas, na realidade, estes recursos são apenas utilitários, destinados a facilitar a interação do usuário com o computador.

■ Compartilhamento de recursos de forma organizada e protegida

Em sistemas onde diversos usuários compartilham recursos do sistema computacional, é necessário controlar o uso concorrente desses recursos. Se imaginarmos uma impressora sendo compartilhada, deverá existir algum tipo de controle para que a impressão de um usuário não interfira nas dos demais. Novamente é o sistema operacional que tem a responsabilidade de permitir o acesso concorrente a esse e a outros recursos de forma organizada e protegida.

O compartilhamento de recursos permite, também, a diminuição de custos, na medida em que mais de um usuário pode utilizar as mesmas facilidades concorrentemente, como discos, impressoras, linhas de comunicação etc.

Não é apenas em sistemas multiusuário que o sistema operacional é importante. Se pensarmos que um

computador pessoal nos permite executar diversas tarefas ao mesmo tempo, como imprimir um documento, copiar um arquivo pela Internet ou processar uma planilha, o sistema operacional deve ser capaz de controlar a execução concorrente de todas essas atividades.

► 1.3 Máquina de Camadas

Um sistema computacional visto somente sob a ótica do *hardware*, ou seja, como um conjunto de circuitos eletrônicos, placas, cabos e fontes de alimentação, tem pouca utilidade. É por intermédio do *software* que serviços são oferecidos aos usuários, como armazenamento de dados em discos, impressão de relatórios, geração de gráficos, acesso à Internet, entre outras funções.

Uma operação efetuada pelo software pode ser implementada em hardware, enquanto uma instrução executada pelo hardware pode ser simulada via software. Esta decisão fica a cargo do projetista do sistema computacional em função de aspectos como custo, confiabilidade e desempenho. Tanto o hardware quanto o software são logicamente equivalentes, interagindo de uma forma única para o usuário (Tanenbaum, 2007).

Nos primeiros computadores, a programação era realizada em linguagem de máquina, em painéis através de fios, exigindo, consequentemente, um grande conhecimento da arquitetura do hardware. Isso era uma grande dificuldade para os programadores da época (Fig. 1.2a). O surgimento do sistema operacional minimizou esse problema, tornando a interação entre usuário e computador mais simples, confiável e eficiente. A partir desse momento, não existia mais a necessidade de o programador se envolver com a complexidade do hardware para poder trabalhar; ou seja, a parte física do computador tornou-se transparente para o usuário (Fig. 1.2b).

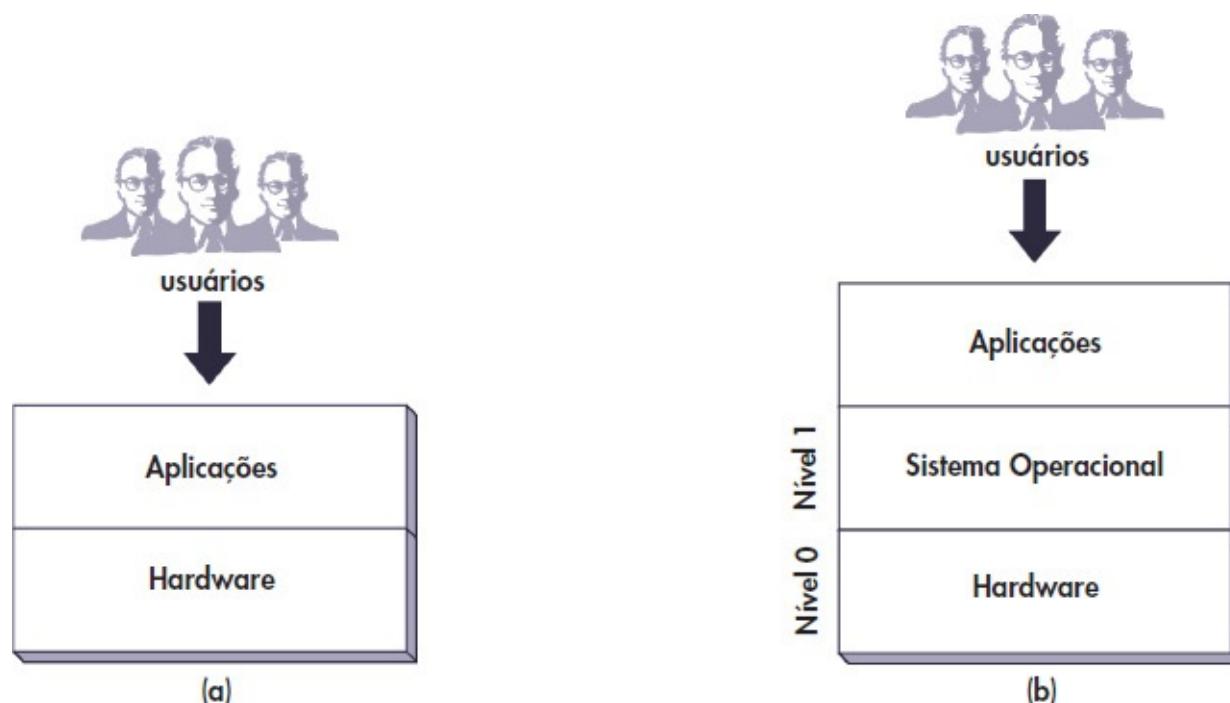


Fig. 1.2 Visão do computador pelo usuário.

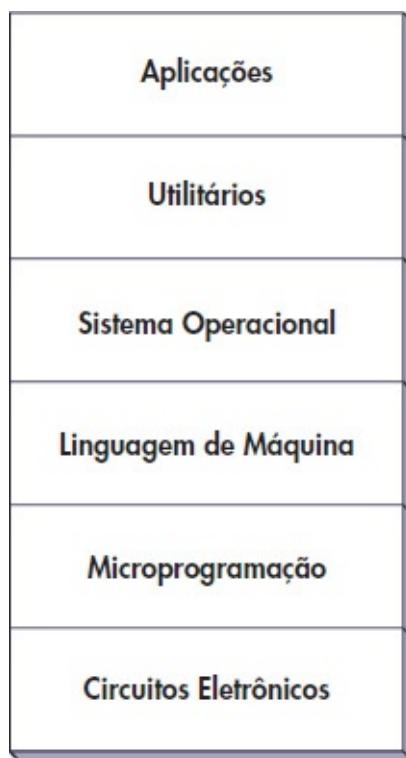


Fig. 1.3 Máquinas de camadas.

O computador pode ser compreendido como uma *máquina de camadas* ou *máquina de níveis*, em que inicialmente existem dois níveis: hardware (nível 0) e sistema operacional (nível 1). Desta forma, a aplicação do usuário interage diretamente com o sistema operacional, ou seja, como se o hardware não existisse. Esta visão modular e abstrata é chamada de *máquina virtual*.

Na realidade, um computador não possui apenas dois níveis, e sim tantos níveis quantos forem necessários para adequar o usuário às suas diversas aplicações. Quando o usuário está trabalhando em um desses níveis não necessita saber da existência das outras camadas, acima ou abaixo de sua máquina virtual.

Atualmente, a maioria dos computadores possui a estrutura básica apresentada na Fig. 1.3, podendo conter um número maior ou menor de camadas. A linguagem utilizada em cada um desses níveis é diferente, variando da mais elementar (baixo nível) à mais sofisticada (alto nível).

► 1.4 Histórico

A evolução dos sistemas operacionais está, em grande parte, relacionada ao desenvolvimento dos computadores. Neste histórico dividimos essa evolução em décadas, nas quais destacamos em cada uma suas principais características de hardware, software, interação com o sistema e aspectos de conectividade.

Antes da década de 1940, inúmeros esforços foram feitos para criar uma máquina que pudesse realizar cálculos de forma mais rápida e precisa. Em 1642, o matemático francês Blaise Pascal inventou uma máquina de somar para auxiliar seu pai no processo de arrecadação de impostos. Em 1673, o matemático e filósofo alemão Gottfried Leibniz foi além e criou uma máquina capaz de somar e multiplicar, utilizando o conceito de acumulador. Mais tarde, em 1820, o francês Charles Colmar inventaria

finalmente uma máquina capaz de executar as quatro operações.

Em 1822 o matemático inglês Charles Babbage criou uma máquina para cálculos de equações polinomiais. Mais tarde, em 1833, Babbage evoluiria esta ideia para uma máquina capaz de executar qualquer tipo de operação, conhecida como Máquina Analítica (*Analytical Engine*). Seu invento é o que mais se aproxima de um computador atual, possuindo os conceitos de unidade central de processamento, memória, unidade de controle e dispositivos de entrada/saída. Enquanto Babbage se preocupava com as características mecânicas do seu invento (hardware), sua discípula Augusta Ada Byron era responsável pela sequência de instruções executadas pela máquina (software). Pelo trabalho realizado na época, Ada Byron é considerada a primeira programadora da história. Devido às limitações técnicas da época, a Máquina Analítica nunca funcionou de forma adequada, mesmo assim Babbage é considerado o “pai do computador”.

Em 1854, o também matemático inglês George Boole criaria a lógica booleana, base para o modelo de computação digital utilizado até hoje. O conceito de lógica binária seria utilizado no desenvolvimento de dispositivos como relés e válvulas, implementados nos primeiros computadores da década de 1940.

No final do século XIX, Herman Hollerith criou um mecanismo utilizando cartões perfurados para acelerar o processamento do censo de 1890 nos EUA. Hollerith fundaria em 1896 a Tabulating Machine Company, que se tornaria a International Business Machine (IBM) em 1924. A utilização de cartões perfurados na computação perduraria por grande parte do século XX, e o nome Hollerith tornar-se-ia sinônimo de cartão perfurado.

Na década de 1930 surgem as primeiras tentativas reais de criar-se uma calculadora eletrônica. Na Alemanha, Konrad Zuse desenvolveu o Z-1, baseado em relés e que utilizava lógica binária. Nos EUA, John Vincent Atanasoff e Clifford Berry desenvolveram uma máquina para o cálculo de equações lineares. Para muitos, o ABC (Atanasoff-Berry Computer) é considerado o primeiro computador eletrônico da história.

Em 1937, o matemático inglês Alan Turing desenvolveu a ideia de Máquina Universal ou Máquina de Turing, capaz de executar qualquer sequência de instruções (algoritmo). Apesar de ser um modelo teórico, a Máquina Universal criou a ideia de “processamento de símbolos”, base da ciência da computação moderna.

1.4.1 Década de 1940

A Segunda Guerra Mundial desencadeou o desenvolvimento de máquinas que pudessem acelerar os procedimentos manuais realizados para fins militares. Nesse período surgiram os primeiros computadores eletromecânicos (calculadoras), formados por milhares de válvulas, que ocupavam áreas enormes, sendo de funcionamento lento e duvidoso.

Em 1943, na Inglaterra, Alan Turing estava envolvido no desenvolvimento de uma máquina chamada Colossus para decifrar o código das mensagens alemãs, conhecido como Enigma. Este computador foi desenvolvido e utilizado naquebra de diversos códigos nazistas, como o do “Dia D”, e significou uma grande vantagem para os aliados na Segunda Guerra.

Nos EUA, em 1944, foi construído o primeiro computador eletromecânico, batizado de Mark I. Desenvolvido pelo professor Howard Aiken, da Universidade de Harvard, e com apoio da IBM, foi utilizado para cálculos matemáticos pela Marinha. O Mark I utilizava os mesmos princípios da Máquina Analítica, criada por Babbage cem anos antes.

O ENIAC (Electronic Numerical Integrator And Calculator) é considerado o primeiro computador digital e eletrônico. Desenvolvido pelos engenheiros J. Presper Eckert e John W. Mauchly na Universidade da Pensilvânia, foi criado para a realização de cálculos balísticos e, posteriormente, utilizado no projeto da bomba de hidrogênio, ficando em operação no período de 1946 a 1955. Sua estrutura possuía 17 mil válvulas, 10 mil capacitores, 70 mil resistores e pesava 30 toneladas. Quando em operação, consumia cerca de 140 quilowatts e era capaz de realizar 5 mil adições por segundo.

Para trabalhar com o ENIAC era necessário conhecer profundamente o funcionamento do hardware, pois a programação era feita em painéis, através de 6 mil conectores, utilizando linguagem de máquina. Esta tarefa poderia facilmente levar alguns dias. Corretamente programado, um cálculo que levasse vinte e quatro horas manualmente era resolvido em menos de trinta segundos. A diferença entre a velocidade de processamento e o tempo necessário para codificar um programa passou a ser um grande problema a ser resolvido.

O professor John von Neumann, consultor no projeto do ENIAC, imaginou uma máquina de propósito geral na qual tanto instruções quanto dados fossem armazenados em uma mesma memória, tornando o processo de programação muito mais rápido e flexível. Este conceito, aparentemente simples, conhecido como “programa armazenado”, é a base da arquitetura de computação atual, batizada de “Arquitetura von Neumann”. Finalmente, as mesmas ideias de Babbage (Máquina Analítica) e Turing (Máquina Universal) puderam ser colocadas em prática em um mesmo sistema computacional.

O primeiro computador a implementar o conceito de “programa armazenado” foi o EDSAC (Electronic Delay Storage Automatic Calculator), desenvolvido pelo professor Maurice Wilkes, na Universidade de Cambridge, na Inglaterra, em 1949. Outros computadores foram construídos nessa mesma época com base no mesmo princípio, como o EDVAC (Electronic Discrete Variable Automatic Computer) na Universidade da Pensilvânia, IAS (Institute for Advanced Studies) em Princeton pelo próprio von Neumann, Manchester Mark I, ORDVAC e ELLIAC na Universidade de Illinois, JOHNIAC pela Rand Corp., MANIAC em Los Alamos e WEIZAC em Israel. A maioria destas máquinas foi utilizada apenas em universidades e órgãos militares para cálculos matemáticos.

Nessa fase, os computadores não possuíam ainda dispositivos com função de interface com os usuários, como teclados e monitores, e o conceito de sistema operacional surgiria apenas na década seguinte.

1.4.2 Década de 1950

O uso do transistor e da memória magnética contribuiu para o enorme avanço dos computadores da época. O transistor permitiu o aumento da velocidade e da confiabilidade no processamento, e as memórias magnéticas permitiram o acesso mais rápido aos dados, maior capacidade de armazenamento e computadores menores. Apesar de o seu invento datar do final da década de 1940, os primeiros computadores transistorizados foram lançados comercialmente apenas no final da década de 1950.

Com o desenvolvimento da indústria de computadores, muitas empresas foram criadas ou investiram no setor, como Raytheon, RCA, Burroughs e IBM, o que levou à criação dos primeiros computadores para utilização em aplicações comerciais.

Em 1946, Eckert e Mauchly deixaram a Universidade da Pensilvânia para formar a primeira empresa de computadores, a Eckert-Mauchly Computer Corp. (EMCC), com a intenção de construir o UNIVAC (Universal Automatic Computer). Devido a problemas financeiros, a EMCC foi adquirida pela Remington Rand Corp., possibilitando a conclusão do projeto a tempo de ser utilizado no censo dos EUA em 1951. O UNIVAC I foi o primeiro computador bem-sucedido fabricado para fins comerciais.

Em 1951, o Massachusetts Institute of Technology (MIT) colocou em operação o que é considerado o primeiro computador voltado para o processamento em tempo real, o Whirlwind I. Entre diversas inovações, o Whirlwind introduziu a tecnologia de memória magnética.

Os programas ou jobs passaram a ser perfurados em cartões, que, submetidos a uma leitora, eram gravados em uma fita de entrada (Fig. 1.4a). A fita, então, era lida pelo computador, que executava um programa de cada vez, gravando o resultado do processamento em uma fita de saída (Fig. 1.4b). Ao término de todos os programas, a fita de saída era lida e impressa (Fig. 1.4c). A esse tipo de processamento, em que um conjunto de programas era submetido ao computador, deu-se o nome de *processamento batch*.

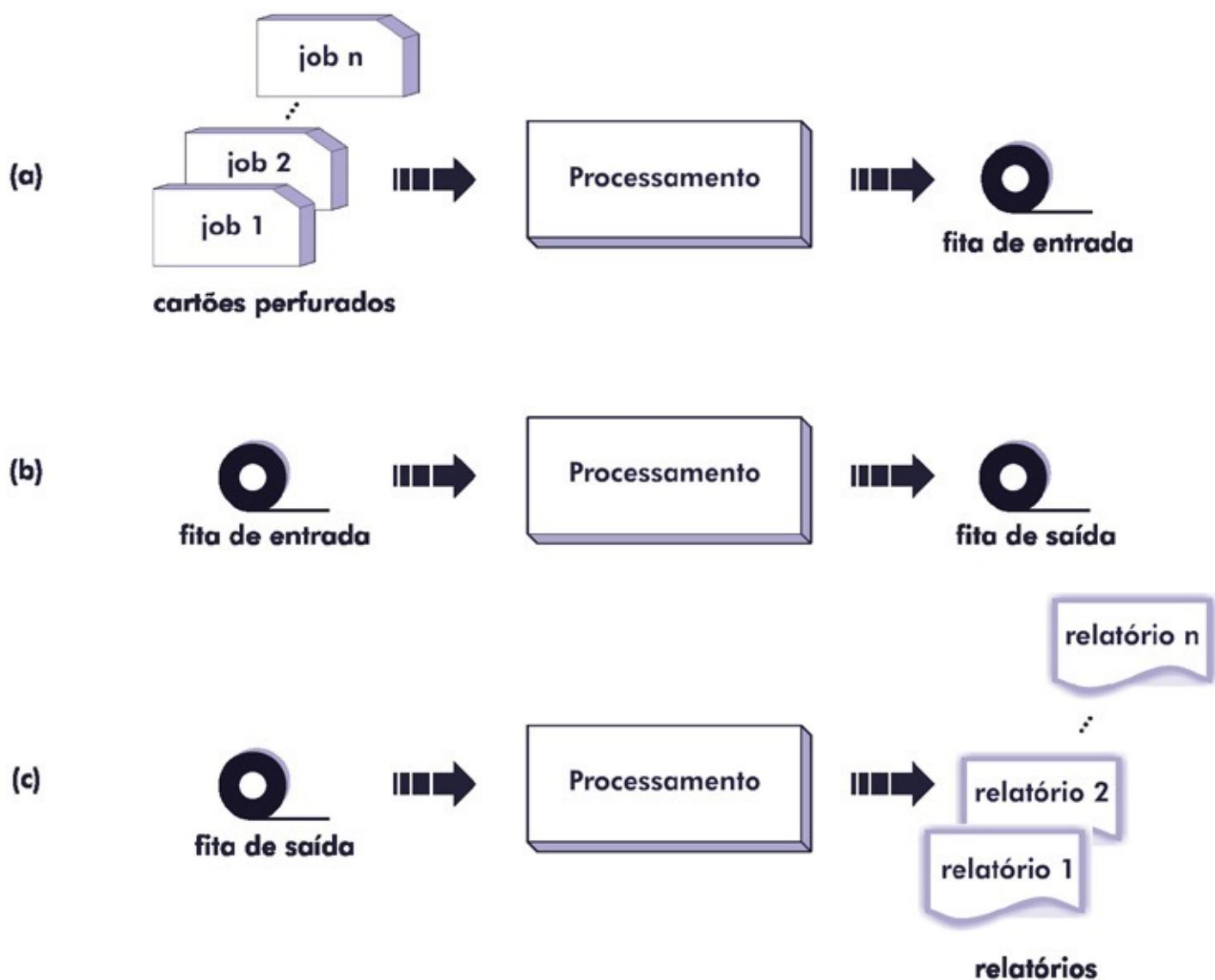


Fig. 1.4 Processamento batch.

Pode não parecer um avanço, mas anteriormente os programas eram submetidos pelo operador, um a um, fazendo com que o processador ficasse ocioso entre a execução de um job e outro. Com o processamento batch, um conjunto de programas era submetido de uma só vez, o que diminuía o tempo de espera entre a execução dos programas, permitindo, assim, melhor aproveitamento do processador.

O primeiro sistema operacional, chamado monitor por sua simplicidade, foi desenvolvido em 1953 pelos usuários do computador IBM 701 do Centro de Pesquisas da General Motors, justamente para tentar automatizar as tarefas manuais até então utilizadas. Posteriormente, este sistema seria reescrito para um computador IBM 704 pelo grupo de usuários da IBM (Weizer, 1981).

Com o surgimento das primeiras linguagens de programação de alto nível, como FORTRAN, ALGOL e COBOL, os programas deixaram de ter relação direta com o hardware dos computadores, o que facilitou e agilizou enormemente o desenvolvimento e a manutenção de programas.

Da mesma maneira que as linguagens de programação, os sistemas operacionais evoluíram no sentido de facilitar o trabalho de codificação, submissão, execução e depuração de programas. Para isso, os sistemas operacionais incorporaram seu próprio conjunto de rotinas para operações de entrada/saída (Input/Output Control System — IOCS). O IOCS eliminou a necessidade de os programadores

desenvolverem suas próprias rotinas de leitura/gravação específicas para cada dispositivo. Essa facilidade de comunicação criou o conceito de *independência de dispositivos*, introduzido pelos sistemas operacionais SOS (SHARE Operating System), FMS (FORTRAN Monitor System) e IBSYS, todos para máquinas IBM.

No final da década de 1950, a Universidade de Manchester, na Inglaterra, desenvolveu o sistema operacional Atlas, que introduziu a ideia de memória hierarquizada, base do conceito de *memória virtual*, presente hoje na maioria dos sistemas operacionais atuais. O Atlas implementava o esquema de paginação por demanda para transferir informações da memória secundária para a principal.

1.4.3 Década de 1960

A partir do surgimento dos circuitos integrados, foi possível viabilizar e difundir o uso de sistemas computacionais nas empresas, devido à redução de seus custos de aquisição. Além disso, houve grande aumento do poder de processamento e diminuição no tamanho dos equipamentos. A década de 1960 foi palco de inúmeras inovações na área de sistemas operacionais, tendo sido implementadas várias técnicas presentes até hoje, como multiprogramação, multiprocessamento, time-sharing e memória virtual.

Uma das características mais importantes surgidas nessa época foi a introdução do conceito de *multiprogramação*. Antes deste conceito, sempre que um programa realizava uma operação de entrada/saída o processador ficava ocioso, aguardando o término da operação. A multiprogramação permitiu que vários programas compartilhassem a memória ao mesmo tempo e, enquanto um programa esperava por uma operação de leitura/gravação, o processador executava um outro programa.

Com a substituição das fitas por discos no processo de submissão dos programas, o processamento batch, juntamente com a multiprogramação, tornou os sistemas mais rápidos e eficientes, pois permitia a carga mais rápida dos programas na memória e a alteração na ordem de execução das tarefas, até então puramente sequencial.

Em 1963, a Burroughs lançou o computador B-5000 com o sistema operacional Master Control Program (MCP), que oferecia multiprogramação, memória virtual com segmentação e multiprocessamento assimétrico, além de ser o primeiro sistema a ser desenvolvido em uma linguagem de alto nível. No mesmo ano, a Control Data Corporation anunciou o lançamento do primeiro supercomputador, o CDC 6600, projetado por Seymour Cray.

A IBM lançou em 1964 o System/360, que causaria uma revolução na indústria de informática, pois introduzia um conceito de máquinas de portes diferentes, porém com uma mesma arquitetura, permitindo a total compatibilidade entre os diferentes modelos. Desta forma, uma empresa poderia adquirir um modelo mais simples e barato e, conforme suas necessidades, migrar para modelos com mais recursos, sem comprometer suas aplicações já existentes. Para essa série, foi desenvolvido o sistema operacional OS/360.

Os sistemas batch implementando multiprogramação, como o OS/360, foram um grande avanço para os programadores, porém o trabalho de desenvolvimento e depuração das aplicações ainda era lento e tedioso. Caso o programador cometesse apenas um erro de digitação, o tempo de resposta do sistema

poderia levar horas. A evolução foi oferecer aos usuários tempos de respostas menores e uma interface que permitisse interagir rapidamente com o sistema. Para tal, cada programa poderia utilizar o processador por pequenos intervalos de tempo. A esse sistema de divisão de tempo chamou-se *tempo compartilhado* (*time-sharing*). Para que a interface entre o computador e usuários fosse possível, foram introduzidos novos dispositivos de entrada/saída, como o terminal de vídeo e o teclado, possibilitando a interação do usuário com a aplicação no decorrer da sua execução (*sistema on-line*).

Um dos primeiros sistemas operacionais de tempo compartilhado foi o CTSS (Compatible Time-Sharing System). Desenvolvido pelo MIT em 1962 para um computador IBM 7094, suportava no máximo 32 usuários interativos, e através de comandos em um terminal permitia compilar e executar seus programas. O CTSS foi a base para outros sistemas operacionais de tempo compartilhado, como o MULTICS (Corbató, 1962).

Em 1965, o MIT, a Bell Labs e a General Electric estavam envolvidos no projeto do sistema operacional MULTICS (Multiplexed Information and Computing Service) para um computador GE 645. Este sistema deveria oferecer vários serviços de forma contínua e confiável, similar aos serviços de luz e telefonia. O MULTICS implementava memória virtual com segmentação e paginação, multiprogramação e deveria suportar múltiplos processadores e usuários. A maior parte do sistema seria desenvolvida em PL/I, uma linguagem de alto nível, para torná-lo portável, ou seja, independente da plataforma de hardware (Corbató, 1965). Apesar de o MULTICS não ter alcançado seus objetivos, suas ideias influenciariam inúmeros sistemas posteriormente.

A Digital Equipment Corp. (DEC) lançou o PDP-8 em 1965, também revolucionário, pois representava a primeira linha de computadores de pequeno porte e baixo custo, comparativamente aos mainframes até então comercializados, criando o mercado de minicomputadores. Em 1969, Ken Thompson, que trabalhara no projeto do MULTICS, utilizou um PDP-7 para fazer sua própria versão de um sistema operacional que viria a ser conhecido como Unix.

1.4.4 Década de 1970

A integração em larga escala (Large Scale Integration — LSI) e a integração em muito larga escala (Very Large Scale Integration — VLSI) levaram adiante o projeto de miniaturização e barateamento dos equipamentos. Seguindo esta tendência, a Digital lançou uma nova linha de minicomputadores, o PDP-11 em 1970 e, posteriormente, o sistema VAX/VMS (Virtual Memory System) de 32 bits.

Em 1971, a Intel Corp. produziu seu primeiro microprocessador, o Intel 4004 e, três anos depois, o Intel 8080, utilizado no primeiro microcomputador, o Altair. Posteriormente, a Zilog lançaria um processador concorrente ao da Intel, o Z80. Com a evolução dos microprocessadores, os microcomputadores ganharam rapidamente o mercado por serem muito mais baratos que qualquer um dos computadores até então comercializados. Em 1976, Steve Jobs e Steve Wozniak produziram o Apple II de 8 bits, tornando-se um sucesso imediato. Neste mesmo ano, as empresas Apple e a Microsoft foram fundadas. O sistema operacional dominante nos primeiros microcomputadores foi o CP/M (Control Program Monitor) da Digital Research.

Para acelerar o processamento foram desenvolvidas arquiteturas com diversos processadores, exigindo

dos sistemas operacionais novos mecanismos de controle e sincronismo. O *multiprocessamento* possibilitou a execução de mais de um programa simultaneamente ou até de um mesmo programa por mais de um processador. Além de equipamentos com múltiplos processadores, foram introduzidos processadores vetoriais e técnicas de paralelismo em diferentes níveis, tornando os computadores ainda mais poderosos. Em 1976, o Cray-1 foi lançado contendo 200.000 circuitos integrados e realizando 100 milhões de operações de ponto flutuante por segundo (100 MFLOPS).

As redes distribuídas (Wide Area Network — WANs) difundiram-se, permitindo o acesso a outros sistemas de computação, independentemente da distância geográfica. Nesse contexto foram desenvolvidos inúmeros protocolos de rede, alguns proprietários, como o DECnet da Digital e o SNA (System Network Architecture) da IBM, e outros de domínio público, como o NCP (predecessor do TCP/IP) e o X.25. Surgiram as primeiras redes locais (Local Area Network — LANs) interligando computadores restritos a pequenas áreas. Os sistemas operacionais passaram a estar intimamente relacionados aos softwares de rede.

Duas importantes linguagens de programação foram desenvolvidas nessa década. Em 1971, o professor Niklaus Wirth desenvolveu a linguagem Pascal, voltada para o ensino de técnicas de programação. Em 1975, Dennis Ritchie desenvolveu a linguagem C e, juntamente com Ken Thompson, portou o sistema Unix para um PDP-11, concebido inicialmente em assembly.

1.4.5 Década de 1980

Em 1981, a IBM entrou no mercado de microcomputadores com o IBM PC (Personal Computer), criando a filosofia dos computadores pessoais. O primeiro PC utilizava o processador Intel 8088 de 16 bits e o sistema operacional DOS (Disk Operating System) da Microsoft, muito semelhante ao CP/M.

Na área dos mini e superminicomputadores ganharam impulso os sistemas *multusuário*, com destaque para os sistemas compatíveis com o Unix. A Universidade de Berkeley, na Califórnia, desenvolveu sua própria versão do sistema Unix (Berkeley Software Distribution – BSD) e introduziu inúmeros melhoramentos, merecendo destaque o protocolo de rede TCP/IP (Transmission Control Protocol/Internet Protocol).

Surgiram as estações de trabalho (workstations) que, apesar de serem sistemas monousuário, permitiram que fossem executadas diversas tarefas concorrentemente (multitarefa). Em 1982, foi fundada a Sun Microsystems, que passaria a atuar fortemente neste setor, lançando as primeiras estações RISC com o sistema operacional SunOS e, posteriormente, Sun Solaris.

Com a evolução dos microprocessadores, principalmente da família Intel, surgiram os primeiros sistemas operacionais comerciais que ofereciam interface gráfica, como o Microsoft Windows e o OS/2. O software de rede passou a estar fortemente relacionado ao sistema operacional, e surgiram os *sistemas operacionais de rede*, com destaque para o Novell Netware e o Microsoft LAN Manager.

Entre os anos de 1982 e 1986, foi desenvolvido no Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro (NCE/UFRJ) o sistema operacional PLURIX para o computador PEGASUS, também construído no NCE. Na década seguinte, o PLURIX seria transportado para a linha de

processadores Intel, dando origem ao TROPIX, um sistema operacional multiusuário e multitarefa, de filosofia Unix, disponível gratuitamente na Internet (TROPIX, 2002).

1.4.6 Década de 1990

Grandes avanços em termos de hardware, software e telecomunicações foram obtidos nessa década. Essas mudanças foram consequência da evolução das aplicações, que necessitavam cada vez mais de capacidade de processamento e armazenamento de dados, como em sistemas especialistas, sistemas multimídia, banco de dados distribuídos, inteligência artificial e redes neurais.

A evolução da microeletrônica permitiu o desenvolvimento de processadores e memórias cada vez mais velozes e baratos, além de dispositivos de E/S menores, mais rápidos e com maior capacidade de armazenamento. Os componentes baseados em tecnologia VLSI evoluíram rapidamente para o ULSI (Ultra Large Scale Integration).

Com o surgimento e a evolução da Internet, o protocolo TCP/IP passou a ser um padrão de mercado, obrigando os fabricantes de sistemas operacionais a oferecer suporte a este protocolo. Devido ao crescimento acelerado da Internet, problemas de gerência, segurança e desempenho tornaram-se fatores importantes relacionados ao sistema operacional e à rede.

A arquitetura cliente/servidor, aplicada basicamente a redes locais, passou a ser utilizada em redes distribuídas como a Internet, permitindo que qualquer pessoa tivesse acesso a todo tipo de informação, independentemente de onde esteja armazenada. A partir deste modelo de computação foram criados diversos sistemas dedicados a oferecer serviços, como servidores web, de correio, de arquivos e servidores de banco de dados.

A década de 1990 foi definitiva para a consolidação dos sistemas operacionais baseados em interfaces gráficas. Os conceitos e implementações só vistos em sistemas considerados de grande porte foram introduzidos na maioria dos sistemas para desktop, como na família Windows da Microsoft e no Unix. Em 1991, o finlandês Linus Torvalds começou o desenvolvimento do Linux, que evoluiu a partir da colaboração de vários programadores que ajudaram no desenvolvimento do kernel, utilitários e vários aplicativos. Atualmente, o Linux é utilizado tanto para fins acadêmicos como comerciais. Em 1993, a Microsoft lançou o Windows NT para competir no mercado de servidores corporativos e ao mesmo tempo substituir as versões anteriores do MS-DOS e MS-Windows utilizadas nos computadores pessoais. Durante essa década, o MS-Windows NT e o Unix (HP-UX, IBM-AIX e Sun Solaris) consolidaram-se como sistemas para ambientes corporativos.

Outro fato importante nessa década foi o amadurecimento e a popularização do software aberto. Com a evolução da Internet, inúmeros produtos foram desenvolvidos e disponibilizados para uso gratuito, como sistemas operacionais (Linux), banco de dados (MySQL), servidores web (Apache), servidores de correio (Sendmail), dentre outros.

1.4.7 Década de 2000

Essa década foi caracterizada por uma grande evolução nos processadores e equipamentos de comunicação. Os computadores tornaram-se muito mais eficientes e com custos reduzidos, transformando-se em equipamentos acessíveis a diversas camadas da população. A tendência de integração de componentes em alta escala e miniaturização fez com que notebooks, netbooks e PDAs tivessem grande disseminação no mercado.

As redes de comunicação cresceram em abrangência e velocidade, possibilitando a integração de computadores e sistemas de governos, empresas e pessoas. A Internet se consolidou em todo o mundo como uma rede de integração entre pessoas por meio de redes sociais como Orkut, Facebook, Twitter, Flickr, MySpace e Google+, assim como um canal de comércio com a difusão do comércio eletrônico. Os navegadores, como o Internet Explorer, Firefox e Chrome, evoluíram no sentido de possibilitar uma interação fácil e segura com a Internet.

Os sistemas operacionais tornaram-se mais intuitivos e presentes em dispositivos móveis como telefones celulares, handhelds e palmtops. Novas interfaces surgiram, fazendo com que a comunicação usuário-computador se tornasse mais inteligente, simples e eficiente. Os sistemas passaram a ser proativos, incorporando mecanismos automáticos para recuperação de erros e atualização de correções e novas versões.

Nessa década, os sistemas operacionais tiraram proveito das arquiteturas de processadores de 64 bits, especialmente do aumento do espaço de endereçamento, o que permitiu melhorar o desempenho de aplicações que manipulam grandes volumes de dados. O conceito de *processamento distribuído* foi explorado, de modo que diversas funções foram espalhadas por vários processadores através de redes locais e distribuídas.

A evolução do hardware dos computadores permitiu que o modelo de virtualização, surgido na década de 1960, fosse utilizado em baixas plataformas. Com isso, a virtualização trouxe uma série de benefícios como a portabilidade de código, consolidação de servidores, aumento da disponibilidade dos sistemas, facilidade de escalabilidade e balanceamento de carga, além de facilitar o desenvolvimento de software.

Com a importância cada vez maior da TI para os negócios, a disponibilidade passou a ser de grande importância para as corporações. Para atender a essa demanda, os fornecedores de TI desenvolveram equipamentos e soluções robustas de alta disponibilidade e sistemas em cluster.

Nessa década, a Microsoft evoluiu seu sistema operacional Windows, unificando suas diferentes versões para servidores, desktops e smartphones, assim como a Apple investiu em melhorias sucessivas para o Mac OS. O Linux ganhou robustez suficiente, tornando-se um sistema de uso tanto para desktops como para servidores. Nos servidores corporativos, os sistemas Unix também avançaram com desenvolvimento dos seus fornecedores, HP para o HP-UX, IBM no AIX e Oracle no Solaris.

1.4.8 Década de 2010

A década de 2010 iniciou com a consolidação dos modelos computacionais em nuvem e da popularização de smartphones e tablets. O modelo de computação em nuvem (cloud computing) vem mudando os modelos econômicos da TI tanto do lado do consumidor quanto do lado dos fornecedores. A

comercialização de software como serviço (SaaS – Software as a Service) vem sendo uma alternativa ao modelo anterior de aquisições de licenças de software com instalação em infraestrutura própria. Outros modelos de serviços na nuvem são a infraestrutura como serviço (IaaS – Infrastructure as a Service) e a plataforma como serviço (PaaS – Platform as a Service).

A evolução da tecnologia dos dispositivos móveis, como smartphones e tablets, está proporcionando uma difusão e popularização do uso destes dispositivos em altíssima escala global. Os fabricantes de sistemas operacionais estão investindo muito em sistemas para este tipo de dispositivo com diversas funcionalidades de comunicação, interface intuitiva e compatibilidade com seus sistemas para servidores e desktops.

Desde que chegaram ao mercado, os smartphones já passaram por três gerações (1G, 2G e 3G) e chegam atualmente a 4G. A primeira geração (1G) permitia somente a comunicação por voz e era feita em aparelhos analógicos. A segunda geração (2G), além da comunicação de voz, também já era possível transmitir digitalmente dados, enviar mensagens de texto (SMS) e acessar a internet. A terceira geração (3G) apresentou melhor desempenho e eficiência na transmissão de dados, além de ótima qualidade na transmissão de voz. Além disso, a terceira geração também possui suporte a conteúdos multimídia como vídeos, imagens, jogos e arquivos de música. A quarta geração (4G) tem como base equipamentos que operam com velocidade mínima de 1 Mbps, podendo chegar a 200 Mbps, possibilitando conexões mais rápidas e qualidades de vídeo e áudio superiores.

Com relação aos sistemas operacionais de smartphones, o mercado apresenta diferentes opções entre sistemas proprietários e abertos. Muitos aplicativos têm sido desenvolvidos para estas plataformas. Atualmente os principais sistemas operacionais para smartphones existentes são: Symbian OS, Windows Mobile, Android, Blackberry OS e iOS.

O Symbian OS é o sistema operacional que predomina entre os celulares e smartphones da Nokia, que decidiu por liberar o seu código para que qualquer empresa pudesse desenvolver suas próprias aplicações. Como características deste sistema estão a sua capacidade de ser multitarefa e lidar com aplicações em tempo real, grande estabilidade, recurso de proteção de memória, eficiência e a boa integração que ele estabelece entre telefone e computador.

A Microsoft desenvolveu o Windows Mobile com o objetivo de transferir características da versão desktop para os celulares. O Windows Mobile contém integração com o Microsoft Exchange Server, possibilitando a sincronia de e-mails e arquivos pessoais entre desktops/notebooks e smartphones. Uma característica interessante neste sistema é a sua conectividade com outros dispositivos da Microsoft, como o Zune e o Xbox 360.

O Android é um sistema operacional desenvolvido a partir de um consórcio chamado Open Handset Alliance formado por 47 empresas e liderado pelo Google. O objetivo deste grupo foi desenvolver um novo sistema operacional para celulares que fosse um sistema aberto. O Android suporta diversos padrões de conectividade como Bluetooth, EDGE, 3G e Wi-Fi. O navegador disponível no sistema é baseado no framework de código aberto.

O BlackBerry OS é um sistema operacional proprietário desenvolvido pela Research In Motion (RIM) para sua linha de smartphones BlackBerry. A plataforma foi pioneira no uso de smartphones para

usuários corporativos, suportando sincronização com Microsoft Exchange, Lotus Domino ou Novell GroupWise, e-mail, calendário, tarefas, notas e contatos, quando utilizado em conjunto com o BlackBerry Enterprise Server.

O iOS é um sistema operacional desenvolvido pela Apple a partir do Mac OS X. O sistema foi difundido no uso do iPhone. Dividido em três domínios: o machine level software – utilizado para atender às necessidades de trabalho de todos os usuários; o system level software – utilizado para atender às funções críticas do sistema; e o user level software – utilizado para atender às necessidades de um usuário específico. A partir da versão 4, o iOS passou a ser multitarefa. Uma das características que ajudaram na popularidade deste sistema operacional é a sua interface intuitiva e amigável, além do navegador Safari.

► 1.5 Tipos de Sistemas Operacionais

Os tipos de sistemas operacionais e sua evolução estão relacionados diretamente com a evolução do hardware e das aplicações por ele suportadas. Muitos termos inicialmente introduzidos para definir conceitos e técnicas foram substituídos por outros, na tentativa de refletir uma nova maneira de interação ou processamento. Isto fica muito claro quando tratamos da unidade de execução do processador. Inicialmente, os termos programa ou job eram os mais utilizados, depois surgiu o conceito de processo e subprocesso e, posteriormente, o conceito de thread.

A seguir, abordaremos os diversos tipos de sistemas operacionais, suas características, vantagens e desvantagens (Fig. 1.5).



Fig. 1.5 Tipos de sistemas operacionais.

1.5.1 Sistemas Monoprogramáveis/Monotarefa

Os primeiros sistemas operacionais eram tipicamente voltados para a execução de um único programa. Qualquer outra aplicação, para ser executada, deveria aguardar o término do programa corrente. Os *sistemas monoprogramáveis*, como vieram a ser conhecidos, se caracterizam por permitir que o

processador, a memória e os periféricos permaneçam exclusivamente dedicados à execução de um único programa.

Os sistemas monoprogramáveis estão tipicamente relacionados ao surgimento dos primeiros computadores na década de 1960. Posteriormente, com a introdução dos computadores pessoais e estações de trabalho na década de 1970, este tipo de sistema voltou a ser utilizado para atender máquinas que, na época, eram utilizadas por apenas um usuário. Os *sistemas monotarefa*, como também são chamados, caracterizam-se por permitir que todos os recursos do sistema fiquem exclusivamente dedicados a uma única tarefa.

Neste tipo de sistema, enquanto um programa aguarda por um evento, como a digitação de um dado, o processador permanece ocioso, sem realizar qualquer tipo de processamento. A memória é subutilizada caso o programa não a preencha totalmente, e os periféricos, como discos e impressoras, estão dedicados a um único usuário, nem sempre utilizados de forma integral (Fig. 1.6).

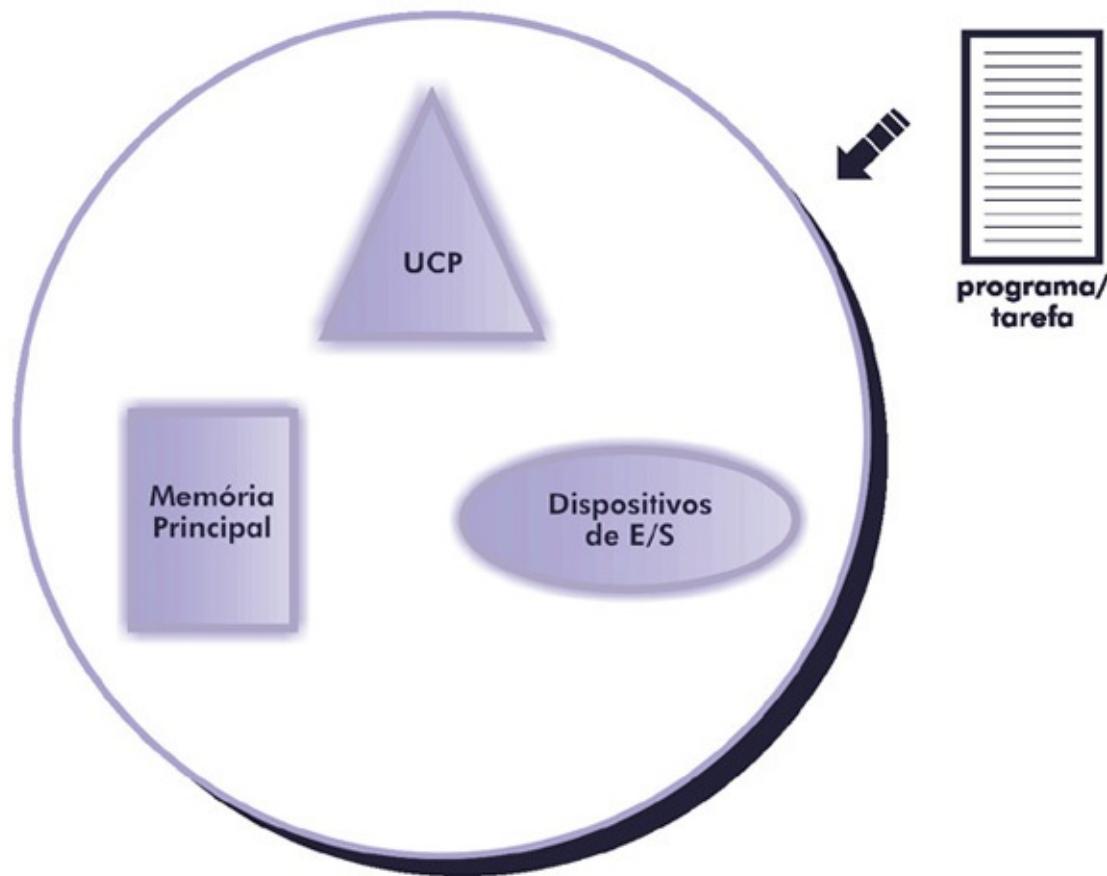


Fig. 1.6 Sistemas monoprogramáveis/monotarefa.

Comparados a outros sistemas, os sistemas monoprogramáveis ou monotarefa são de simples implementação, não existindo muita preocupação com problemas decorrentes do compartilhamento de recursos, como memória, processador e dispositivos de E/S.

1.5.2 Sistemas Multiprogramáveis/Multitarefa

Os sistemas multiprogramáveis ou multitarefa são uma evolução dos sistemas monoprogramáveis. Neste tipo de sistema, os recursos computacionais são compartilhados entre os diversos usuários e aplicações. Enquanto em sistemas monoprogramáveis existe apenas um programa utilizando os recursos disponíveis, nos multiprogramáveis várias aplicações compartilham esses mesmos recursos.

Neste tipo de sistema, por exemplo, enquanto um programa espera por uma operação de leitura ou gravação em disco, outros programas podem estar sendo processados neste mesmo intervalo de tempo. Nesse caso, podemos observar o compartilhamento da memória e do processador. O sistema operacional se preocupa em gerenciar o acesso concorrente aos seus diversos recursos, como memória, processador e periféricos, de forma ordenada e protegida, entre os diversos programas (Fig. 1.7).

A principal vantagem dos sistemas multiprogramáveis é a redução de custos em função da possibilidade do compartilhamento dos diversos recursos entre as diferentes aplicações. Além disso, sistemas multiprogramáveis possibilitam na média a redução total do tempo de execução das aplicações. Os sistemas multiprogramáveis, apesar de mais eficientes que os monoprogramáveis, são de implementação muito mais complexa.

A partir do número de usuários que interagem com o sistema operacional, podemos classificar os sistemas multiprogramáveis como monousuário ou multiusuário. *Sistemas multiprogramáveis monousuário* são encontrados em computadores pessoais e estações de trabalho, onde há apenas um único usuário interagindo com o sistema. Neste caso existe a possibilidade da execução de diversas tarefas ao mesmo tempo, como a edição de um texto, uma impressão e o acesso à Internet. *Sistemas multiprogramáveis multiusuário* são ambientes interativos que possibilitam a diversos usuários conectarem-se ao sistema simultaneamente. A Tabela 1.1 relaciona os tipos de sistemas em função do número de usuários.

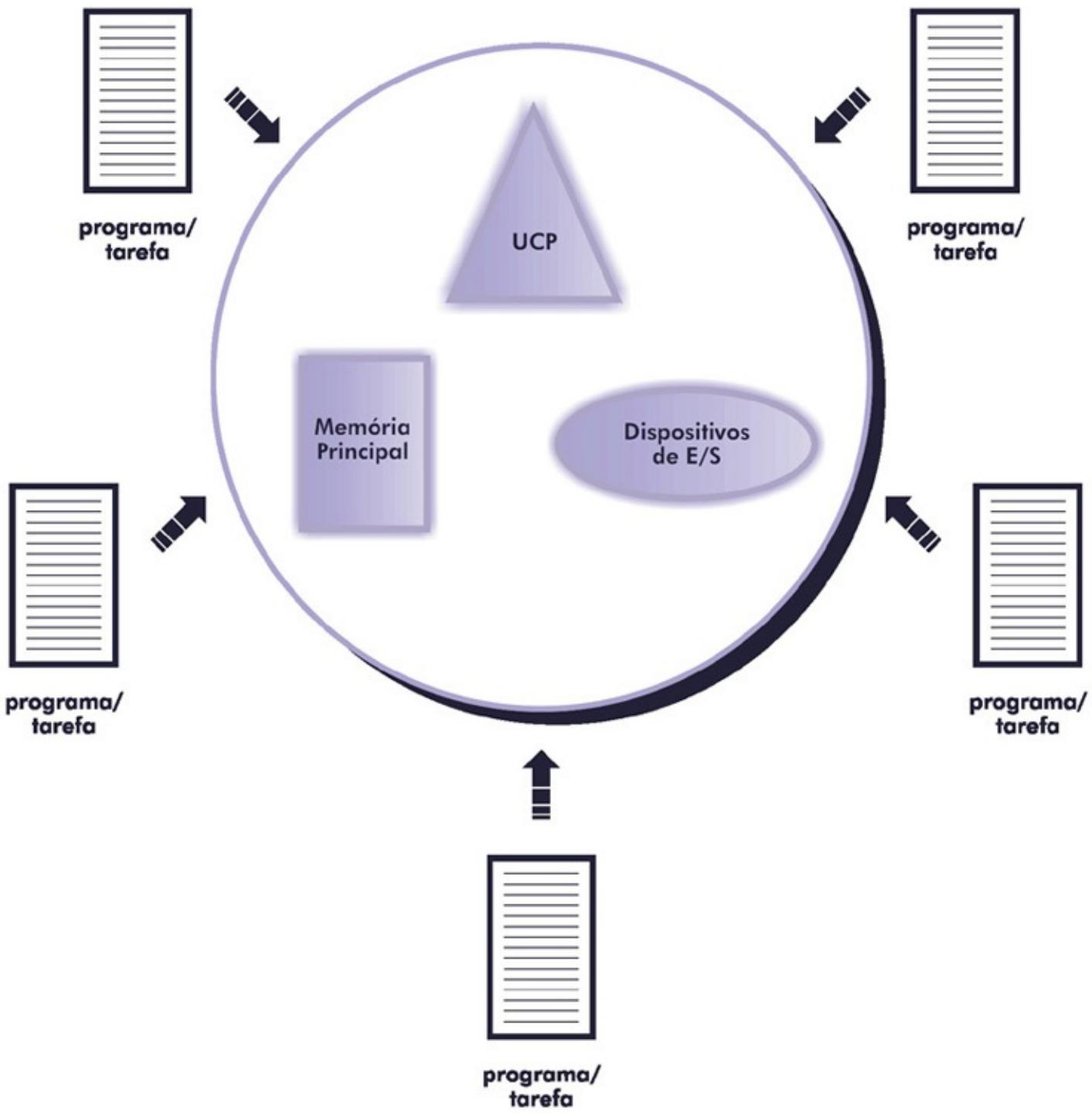


Fig. 1.7 Sistemas multiprogramáveis/multitarefa.

Tabela 1.1 Sistemas × usuários

	Um usuário	Dois ou mais usuários
Monoprogramação/Monotarefa	Monousuário	N/A
Multiprogramação/Multitarefa	Monousuário	Multiusuário

Os sistemas multiprogramáveis ou multitarefa podem ser classificados pela forma com que suas aplicações são gerenciadas, podendo ser divididos em sistemas batch, de tempo compartilhado ou de tempo real. Um sistema operacional pode suportar um ou mais desses tipos de processamento,

dependendo de sua implementação (Fig. 1.8).



Fig. 1.8 Tipos de sistemas multiprogramáveis/multitarefa.

1.5.2.1 Sistemas batch

Os *sistemas batch* foram os primeiros tipos de sistemas operacionais multiprogramáveis a serem implementados na década de 1960. Os programas, também chamados de *jobs*, eram submetidos para execução através de cartões perfurados e armazenados em disco ou fita, onde aguardavam para ser processados. Posteriormente, em função da disponibilidade de espaço na memória principal, os jobs eram executados, produzindo uma saída em disco ou fita.

O *processamento batch* tem a característica de não exigir a interação do usuário com a aplicação. Todas as entradas e saídas de dados da aplicação são implementadas por algum tipo de memória secundária, geralmente arquivos em disco. Alguns exemplos de aplicações originalmente processadas em batch são programas envolvendo cálculos numéricos, compilações, ordenações, backups e todos aqueles onde não é necessária a interação com o usuário.

Esses sistemas, quando bem projetados, podem ser bastante eficientes, devido à melhor utilização do processador, entretanto podem oferecer tempos de resposta longos. Atualmente, os sistemas operacionais implementam ou simulam o processamento batch, não existindo sistemas exclusivamente dedicados a este tipo de processamento.

1.5.2.2 Sistemas de tempo compartilhado

Os *sistemas de tempo compartilhado* (*time-sharing*) permitem que diversos programas sejam executados a partir da divisão do tempo do processador em pequenos intervalos, denominados *fatia de tempo* (*time-slice*). Caso a fatia de tempo não seja suficiente para a conclusão do programa, ele é interrompido pelo sistema operacional e substituído por um outro, enquanto fica aguardando por uma nova fatia de tempo. O sistema cria para cada usuário um ambiente de trabalho próprio, dando a impressão de que todo o sistema está dedicado exclusivamente a ele.

Geralmente, sistemas de tempo compartilhado permitem a interação dos usuários com o sistema através de terminais que incluem vídeo, teclado e mouse. Esses sistemas possuem uma linguagem de controle que permite ao usuário comunicar-se diretamente com o sistema operacional através de comandos. Desta forma é possível verificar arquivos armazenados em disco ou cancelar a execução de um programa. O sistema, normalmente, responde em poucos segundos à maioria desses comandos. Devido a esse tipo de interação, os sistemas de tempo compartilhado também ficaram conhecidos como *sistemas on-line*.

A maioria das aplicações comerciais atualmente é processada em sistemas de tempo compartilhado, pois elas oferecem tempos de respostas razoáveis a seus usuários e custos mais baixos, em função da utilização compartilhada dos diversos recursos do sistema.

1.5.2.3 Sistemas de tempo real

Os *sistemas de tempo real (real-time)* são implementados de forma semelhante aos sistemas de tempo compartilhado. O que caracteriza a diferença entre os dois tipos de sistemas é o tempo exigido no processamento das aplicações. Enquanto em sistemas de tempo compartilhado o tempo de processamento pode variar sem comprometer as aplicações em execução, nos sistemas de tempo real os tempos de processamento devem estar dentro de limites rígidos, que devem ser obedecidos, caso contrário poderão ocorrer problemas irreparáveis.

Nos sistemas de tempo real não existe a ideia de fatia de tempo, implementada nos sistemas de tempo compartilhado. Um programa utiliza o processador o tempo que for necessário ou até que apareça outro mais prioritário. A importância ou prioridade de execução de um programa é definida pela própria aplicação e não pelo sistema operacional.

Esses sistemas, normalmente, estão presentes em aplicações de controle de processos, como no monitoramento de refinarias de petróleo, controle de tráfego aéreo, de usinas termoelétricas e nucleares, ou em qualquer aplicação em que o tempo de processamento é fator fundamental.

1.5.3 Sistemas com Múltiplos Processadores

Os *sistemas com múltiplos processadores* caracterizam-se por possuir duas ou mais UCPs interligadas e trabalhando em conjunto. A vantagem deste tipo de sistema é permitir que vários programas sejam executados ao mesmo tempo ou que um mesmo programa seja subdividido em partes para serem executadas simultaneamente em mais de um processador.

Com múltiplos processadores foi possível a criação de sistemas computacionais voltados, principalmente, para processamento científico, aplicado, por exemplo, no desenvolvimento aeroespacial, prospecção de petróleo, simulações, processamento de imagens e CAD. A princípio qualquer aplicação que faça uso intensivo da UCP será beneficiada pelo acréscimo de processadores ao sistema. A evolução desses sistemas deve-se, em grande parte, ao elevado custo de desenvolvimento de processadores de alto desempenho.

Os conceitos aplicados ao projeto de sistemas com múltiplos processadores incorporam os mesmos

princípios básicos e benefícios apresentados na multiprogramação, além de outras características e vantagens específicas como escalabilidade, disponibilidade e balanceamento de carga.

Escalabilidade é a capacidade de ampliar o poder computacional do sistema apenas adicionando novos processadores. Em ambientes com um único processador, caso haja problemas de desempenho, seria necessário substituir todo o sistema por uma outra configuração com maior poder de processamento. Com a possibilidade de múltiplos processadores, basta acrescentar novos processadores à configuração.

Disponibilidade é a capacidade de manter o sistema em operação mesmo em casos de falhas. Neste caso, se um dos processadores falhar, os demais podem assumir suas funções de maneira transparente aos usuários e suas aplicações, embora com menor capacidade de computação.

Balanceamento de carga é a possibilidade de distribuir o processamento entre os diversos processadores da configuração a partir da carga de trabalho de cada processador, melhorando, assim, o desempenho do sistema como um todo.

Um fator-chave no desenvolvimento de sistemas operacionais com múltiplos processadores é a forma de comunicação entre as UCPs e o grau de compartilhamento da memória e dos dispositivos de entrada e saída. Em função desses fatores, podemos classificar os sistemas com múltiplos processadores em fortemente acoplados ou fracamente acoplados (Fig. 1.9).

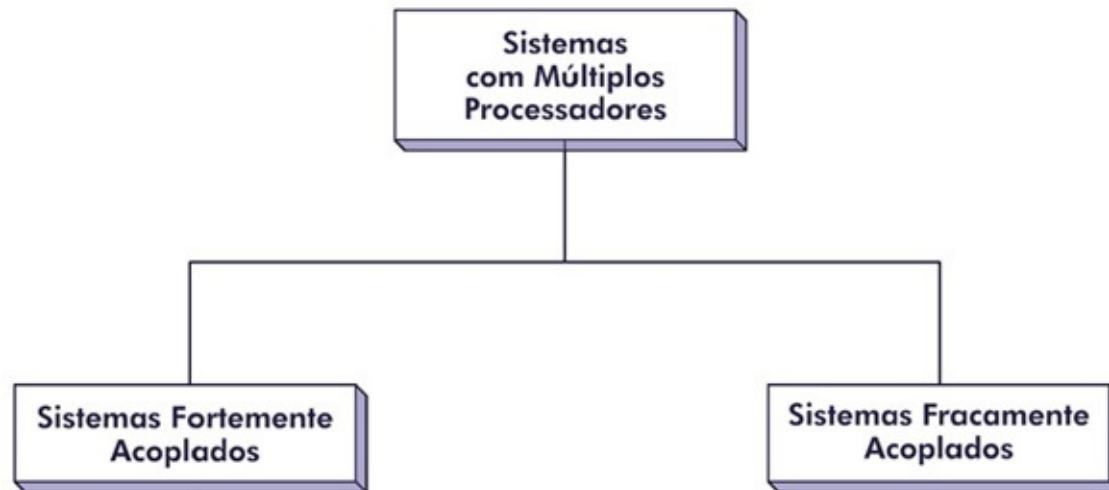


Fig. 1.9 Tipos de sistemas com múltiplos processadores.

A grande diferença entre os dois tipos de sistemas é que em sistemas fortemente acoplados existe apenas uma memória principal sendo compartilhada por todos os processadores, enquanto nos fracamente acoplados cada sistema tem sua própria memória individual. Além disso, a taxa de transferência entre processadores e memória em sistemas fortemente acoplados é muito maior que nos fracamente acoplados.

1.5.3.1 Sistemas fortemente acoplados

Nos *sistemas fortemente acoplados* (tightly coupled) existem vários processadores compartilhando uma única memória física (shared memory) e dispositivos de entrada/saída sendo gerenciados por apenas um

sistema operacional (Fig. 1.10). Em função destas características, os sistemas fortemente acoplados também são conhecidos como *multiprocessadores*.

Os sistemas fortemente acoplados podem ser divididos em SMP (Symmetric Multiprocessors) e NUMA (Non-Uniform Memory Access). Os *sistemas SMP* caracterizam-se pelo tempo uniforme de acesso à memória principal pelos diversos processadores. Os *sistemas NUMA* apresentam diversos conjuntos reunindo processadores e memória principal, sendo que cada conjunto é conectado aos outros através de uma rede de interconexão. O tempo de acesso à memória pelos processadores varia em função da sua localização física.

Nos *sistemas SMP* e *NUMA* todos os processadores têm as mesmas funções. Inicialmente, os sistemas com múltiplos processadores estavam limitados aos sistemas de grande porte, restritos ao ambiente universitário e às grandes corporações. Com a evolução dos computadores pessoais e das estações de trabalho, os sistemas multitarefa evoluíram para permitir a existência de vários processadores no modelo simétrico. Atualmente, a grande maioria dos sistemas operacionais, como o Unix e o Microsoft Windows, implementa esta funcionalidade.

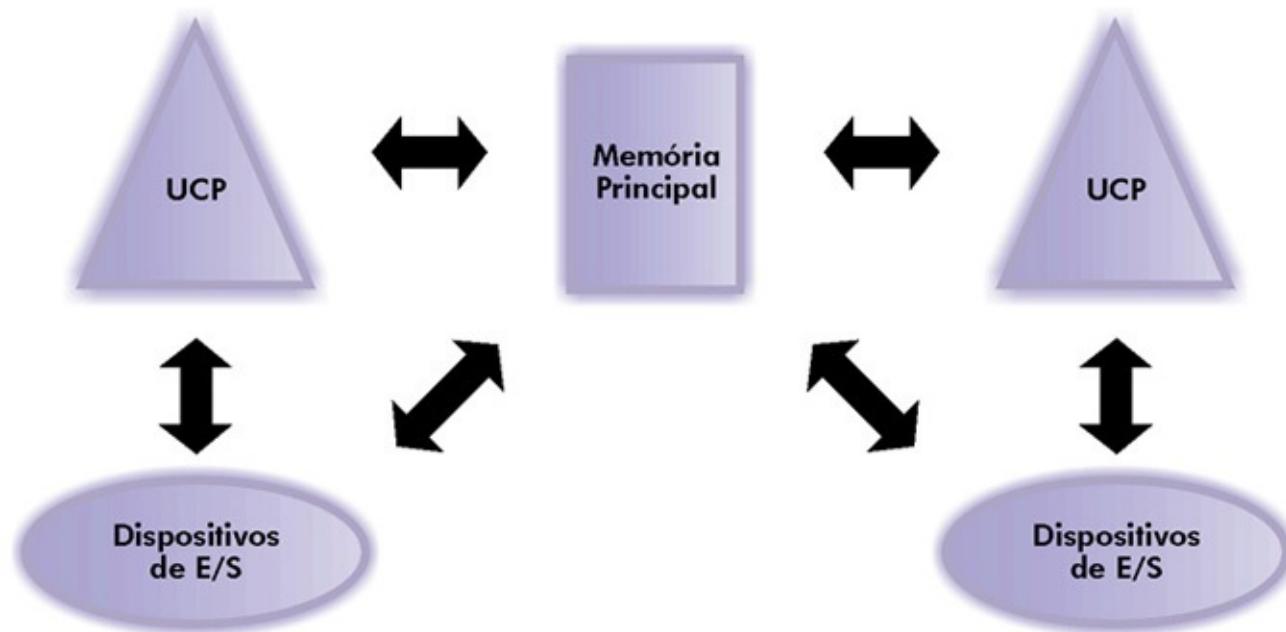


Fig. 1.10 Sistemas fortemente acoplados.

1.5.3.2 Sistemas fracamente acoplados

Os *sistemas fracamente acoplados* (loosely coupled) caracterizam-se por possuir dois ou mais sistemas computacionais conectados através de linhas de comunicação. Cada sistema funciona de forma independente, possuindo seu próprio sistema operacional e gerenciando seus próprios recursos, como UCP, memória e dispositivos de entrada/saída (Fig. 1.11). Em função destas características, os sistemas fracamente acoplados também são conhecidos como *multicomputadores*. Neste modelo, cada sistema computacional também pode ser formado por um ou mais processadores.

Até meados da década de 1980, as aplicações eram tipicamente centralizadas em sistemas de grande

porte, com um ou mais processadores. Neste tipo de configuração, os usuários utilizam terminais não inteligentes conectados a linhas seriais dedicadas ou linhas telefônicas públicas para a comunicação interativa com esses sistemas. No modelo centralizado, os terminais não têm capacidade de processamento. Sempre que um usuário deseja alguma tarefa, o pedido é encaminhado ao sistema, que realiza o processamento e retorna uma resposta, utilizando as linhas de comunicação.

Com a evolução dos computadores pessoais e das estações de trabalho, juntamente com o avanço das telecomunicações e da tecnologia de redes, surgiu um novo modelo de computação, chamado modelo de rede de computadores. Em uma rede existem dois ou mais sistemas independentes (hosts), interligados através de linhas de comunicação, que oferecem algum tipo de serviço aos demais. Neste modelo, a informação deixa de ser centralizada em poucos sistemas de grande porte e passa a ser distribuída pelos diversos sistemas da rede.

Com base no grau de integração dos hosts da rede, podemos dividir os sistemas fracamente acoplados em sistemas operacionais de rede e sistemas distribuídos. A grande diferença entre os dois modelos é a capacidade do sistema operacional em criar uma imagem única dos serviços disponibilizados pela rede.

Os *sistemas operacionais de rede* (SOR) permitem que um host compartilhe seus recursos, como uma impressora ou diretório, com os demais hosts da rede. Um exemplo deste tipo de sistema são as redes locais, onde uma estação pode oferecer serviços de arquivos e impressão para as demais estações da rede, dentre outros serviços.

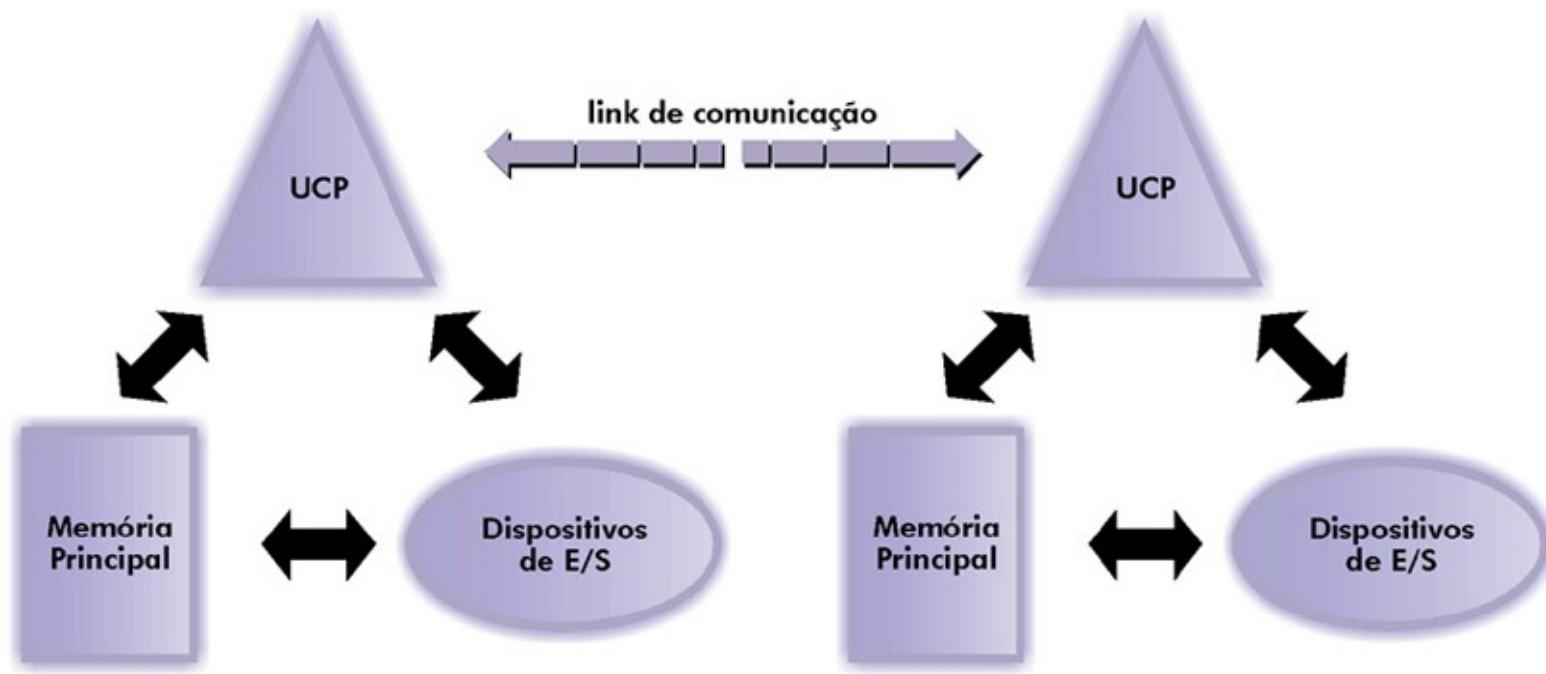


Fig. 1.11 Sistemas fracamente acoplados.

Enquanto nos SORs os usuários têm o conhecimento dos hosts e seus serviços, nos *sistemas distribuídos* o sistema operacional esconde os detalhes dos hosts individuais e passa a tratá-los como um conjunto único, como se fosse um sistema fortemente acoplado. Os sistemas distribuídos permitem, por exemplo, que uma aplicação seja dividida em partes e que cada parte seja executada por hosts diferentes da rede de computadores. Para o usuário e suas aplicações é como se não existisse a rede de computadores, mas

sim um único sistema centralizado.

Outro exemplo de sistema distribuído são os *clusters*. Em um cluster existem dois ou mais servidores ligados, normalmente, por algum tipo de conexão de alto desempenho. O usuário não conhece os nomes dos membros do cluster e não sabe quantos são. Quando ele precisa de algum serviço, basta solicitar ao cluster para obtê-lo. Atualmente, sistemas em cluster são utilizados para serviços de banco de dados e Web, garantindo alta disponibilidade, escalabilidade e balanceamento de carga à solução.

► 1.6 Exercícios

1. Como seria utilizar um computador sem um sistema operacional? Quais são suas duas principais funções?
2. Quais as principais dificuldades que um programador teria no desenvolvimento de uma aplicação em um ambiente sem um sistema operacional?
3. Explique o conceito de máquina virtual. Qual a grande vantagem em utilizar este conceito?
4. Defina o conceito de uma máquina de camadas.
5. Quais os tipos de sistemas operacionais existentes?
6. Por que dizemos que existe uma subutilização de recursos em sistemas monoprogamáveis?
7. Qual a grande diferença entre sistemas monoprogramáveis e sistemas multiprogramáveis?
8. Quais as vantagens dos sistemas multiprogramáveis?
9. Um sistema monousuário pode ser um sistema multiprogramável? Dê um exemplo.
10. Quais são os tipos de sistemas multiprogramáveis?
11. O que caracteriza o processamento batch? Quais aplicações podem ser processadas neste tipo de ambiente?
12. Como funcionam os sistemas de tempo compartilhado? Quais as vantagens em utilizá-los?
13. Qual a grande diferença entre sistemas de tempo compartilhado e tempo real? Quais aplicações são indicadas para sistemas de tempo real?
14. O que são sistemas com múltiplos processadores e quais as vantagens em utilizá-los?
15. Qual a grande diferença entre sistemas fortemente acoplados e fracamente acoplados?
16. O que é um sistema SMP? Qual a diferença para um sistema NUMA?
17. O que é um sistema fracamente acoplado? Qual a diferença entre sistemas operacionais de rede e sistemas operacionais distribuídos?
18. Quais os benefícios de um sistema com múltiplos processadores em um computador pessoal?
19. Qual seria o tipo de sistema operacional recomendável para uso como servidor de aplicações em um ambiente corporativo?
20. Qual seria o tipo de sistema operacional recomendável para executar uma aplicação que manipula grande volume de dados e necessita de um baixo tempo de processamento?



2

Conceitos de Hardware e Software

► 2.1 Introdução

Neste capítulo serão apresentados conceitos básicos de hardware e software relativos à arquitetura de computadores e necessários para a compreensão dos demais capítulos. Os assuntos não serão abordados com profundidade, porém podem ser consultadas referências como Patterson e Hennesy (2012), Stallings (2012) e Tanenbaum (2007) para mais informações.

► 2.2 Hardware

Um *sistema computacional* é um conjunto de circuitos eletrônicos interligados, formado por processadores, memórias, registradores, barramentos, monitores de vídeo, impressoras, mouse, discos magnéticos, além de outros dispositivos físicos (*hardware*). Todos esses dispositivos manipulam dados na forma digital, o que proporciona uma maneira confiável de representação e transmissão de dados.

Todos os componentes de um sistema computacional são agrupados em três subsistemas básicos, chamados *unidades funcionais*: processador ou unidade central de processamento, memória principal e dispositivos de entrada/saída (Fig. 2.1). Estes subsistemas estão presentes em qualquer tipo de computador digital, independentemente da arquitetura ou fabricante. Neste item serão descritos os conceitos básicos dos principais componentes desses subsistemas.

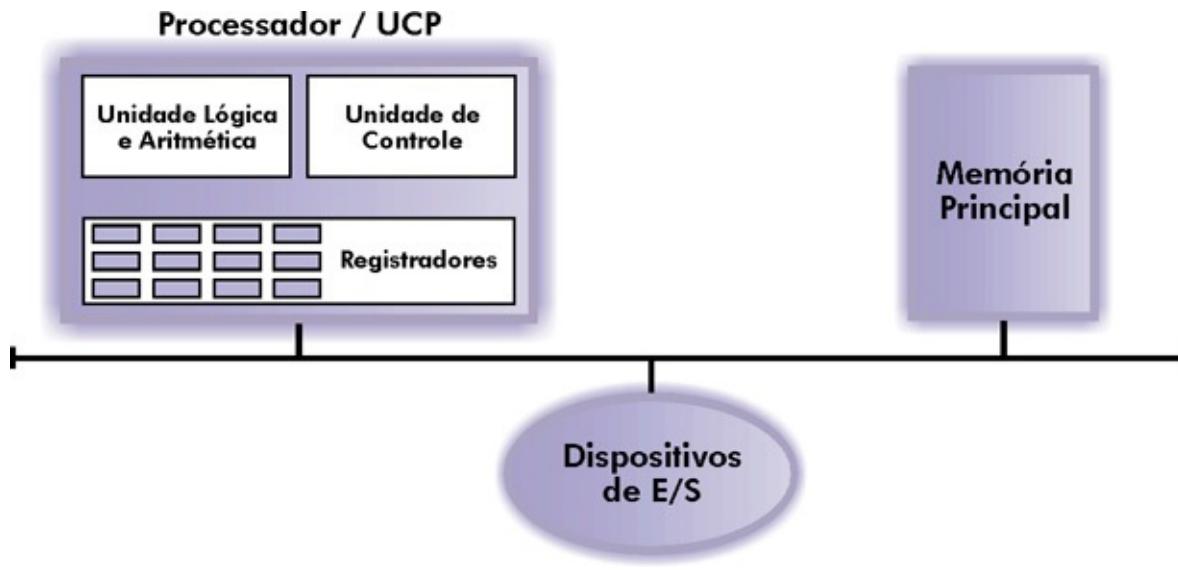


Fig. 2.1 Sistema computacional.

2.2.1 Processador

O *processador*, também denominado *unidade central de processamento* (UCP), gerencia todo o sistema computacional controlando as operações realizadas por cada unidade funcional. A principal função do processador é controlar e executar instruções presentes na memória principal, por meio de operações básicas como somar, subtrair, comparar e movimentar dados.

Cada processador é composto por unidade de controle, unidade lógica e aritmética, e registradores. A *unidade de controle* (UC) é responsável por gerenciar as atividades de todos os componentes do computador, como a gravação de dados em discos ou a busca de instruções na memória. A *unidade lógica e aritmética* (ULA), como o nome indica, é responsável pela realização de operações lógicas (testes e comparações) e aritméticas (somas e subtrações).

A sincronização de todas as funções do processador é realizada através de um *sinal de clock*. Este sinal é um pulso gerado ciclicamente que altera variáveis de estado do processador. O sinal de clock é gerado a partir de um cristal de quartzo que, devidamente polarizado, oscila em uma determinada frequência estável e bem determinada.

Os *registradores* são dispositivos com a função principal de armazenar dados temporariamente. O conjunto de registradores funciona como uma memória de alta velocidade interna do processador, porém com uma capacidade de armazenamento reduzida e custo maior que o da memória principal. O número de registradores e sua capacidade de armazenamento variam em função da arquitetura de cada processador.

Alguns registradores podem ser manipulados diretamente por instruções (registradores de uso geral), enquanto outros são responsáveis por armazenar informações de controle do processador e do sistema operacional (registradores de uso específico). Entre os registradores de uso específico, merecem destaque:

- o *contador de instruções* (CI) ou program counter (PC) contém o endereço da próxima instrução que o processador deve buscar e executar. Toda vez que o processador busca uma nova instrução, este

registrador é atualizado com o endereço de memória da instrução seguinte a ser executada;

- o *apontador da pilha* (AP) ou stack pointer (SP) contém o endereço de memória do topo da pilha, que é a estrutura de dados onde o sistema mantém informações sobre programas que estão sendo executados;
- o *registraror de instruções* (RI) é responsável por armazenar a instrução que será decodificada e executada pelo processador;
- o *registraror de status* ou program status word (PSW) é responsável por armazenar informações sobre a execução de instruções, como a ocorrência de overflow. A maioria das instruções, quando executadas, altera o registraror de status conforme o resultado.

O *ciclo de busca e instrução* é a maneira pela qual o processador executa os programas armazenados na memória principal segundo os passos descritos a seguir:

- a) O processador busca na memória principal a instrução armazenada no endereço indicado pelo CI e a armazena no RI.
- b) O processador incrementa o CI para que o registrado contenha o endereço da próxima instrução.
- c) O processador decodifica a instrução armazenada no RI.
- d) O processador busca os operandos na memória, se houver.
- e) O processador executa a instrução decodificada.

Após este último passo, o ciclo é reiniciado do passo (a).

2.2.2 Memória Principal

A *memória principal, primária ou real* é o local onde são armazenados instruções e dados. A memória é composta por unidades de acesso chamadas células, sendo cada célula composta por um determinado número de bits. O bit é a unidade básica de memória, podendo assumir o valor lógico 0 ou 1.

Atualmente, a grande maioria dos computadores utiliza o byte (8 bits) como tamanho de célula, porém encontramos computadores de gerações passadas com células de 16, 32 e até mesmo 60 bits. Podemos concluir, então, que a memória é formada por um conjunto de células, onde cada célula possui um determinado número de bits (Fig. 2.2).

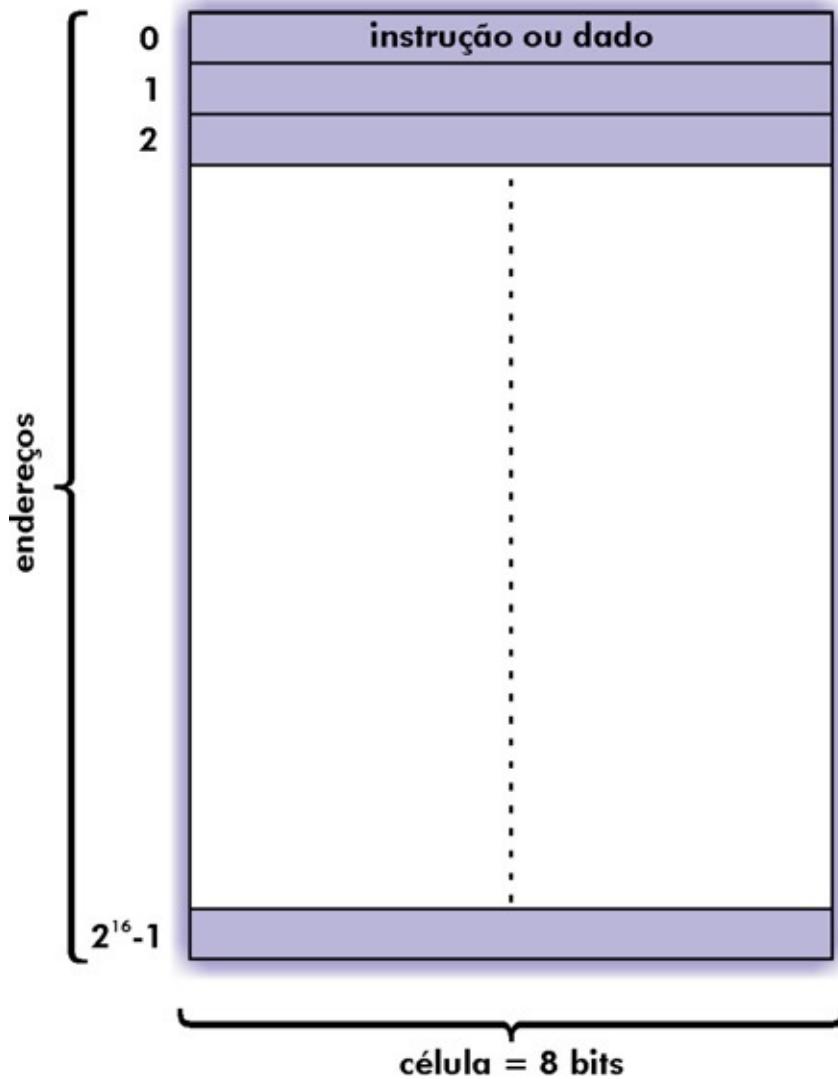


Fig. 2.2 Memória principal com 64 Kbytes.

O acesso ao conteúdo de uma célula é realizado através da especificação de um número chamado *endereço*. O endereço é uma referência única que podemos fazer a uma célula de memória. Quando um programa deseja ler ou escrever um dado em uma célula, deve primeiro especificar qual o endereço de memória desejado, para depois realizar a operação.

A especificação do endereço é realizada por um registrador denominado *registrador de endereço de memória* (memory address register — MAR). Pelo conteúdo deste registrador, a unidade de controle sabe qual a célula de memória que será acessada. Outro registrador usado em operações com a memória é o *registrador de dados da memória* (memory buffer register — MBR). Este registrador é utilizado para guardar o conteúdo de uma ou mais células de memória, após uma operação de leitura, ou para guardar o dado que será transferido para a memória em uma operação de gravação. Este ciclo de leitura e gravação é apresentado na Tabela 2.1.

O número de células endereçadas na memória principal é limitado pelo tamanho do MAR. No caso de o registrador possuir n bits, a memória poderá no máximo endereçar 2^n células, isto é, do endereço 0 ao endereço $(2^n - 1)$.

Tabela 2.1 Ciclo de leitura e gravação

Operação de leitura

Operação de gravação

1. A UCP armazena no MAR o endereço da célula a ser lida.
2. A UCP gera um sinal de controle para a memória principal, indicando que uma operação de leitura deve ser realizada.
3. O conteúdo da(s) célula(s), identificada(s) pelo endereço contido no MAR, é transferido para o MBR.
4. O conteúdo do MBR é transferido para um registrador da UCP

1. A UCP armazena no MAR o endereço da célula que será gravada.
2. A UCP armazena no MBR a informação que deverá ser gravada.
3. A UCP gera um sinal de controle para a memória principal, indicando que uma operação de gravação deve ser realizada.
4. A informação contida no MBR é transferida para a célula de memória endereçada pelo MAR.

A memória principal pode ser classificada em função de sua volatilidade, que é a capacidade de a memória preservar o seu conteúdo mesmo sem uma fonte de alimentação ativa. Memórias do tipo RAM (Random Access Memory) são voláteis, enquanto as memórias ROM (Read-Only Memory) e EPROM (Erasable Programmable ROM) são do tipo não voláteis.

2.2.3 Memória Cache

A *memória cache* é uma memória volátil de alta velocidade, porém com pequena capacidade de armazenamento. O tempo de acesso a um dado nela contido é muito menor que se o mesmo estivesse na memória principal. O propósito do uso da memória cache é minimizar a disparidade existente entre a velocidade com que o processador executa instruções e a velocidade com que dados são lidos e gravados na memória principal. Apesar de ser uma memória de acesso rápido, a capacidade de armazenamento das memórias cache é limitada em função do seu alto custo. A Tabela 2.2 mostra a relação entre memórias cache e principal em alguns equipamentos.

Tabela 2.2 Relação entre memórias cache e principal

	HP 9000/855S	IBM 3090/600S	VAX 9000/440
Tamanho máximo memória principal	25 Mb	512 Mb	512 Mb
Tamanho máximo memória cache	256 Kb	128 Kb por UCP	128 Kb por UCP

A memória cache armazena uma pequena parte do conteúdo da memória principal. Toda vez que o processador faz referência a um dado armazenado na memória, é verificado, primeiramente, se o mesmo encontra-se na memória cache. Caso o processador encontre o dado (cache hit), não há necessidade do acesso à memória principal, diminuindo assim o tempo de acesso.

Se a informação desejada não estiver presente na cache, o acesso à memória principal é obrigatório (cache miss). Neste caso, o processador, a partir do dado referenciado, transfere um bloco de dados da memória principal para a cache. Apesar de existir neste mecanismo um tempo adicional para a transferência de dados entre as diferentes memórias, este tempo é compensado pela melhora do desempenho, justificado pelo alto percentual de referências a endereços que são resolvidos na cache. Isto ocorre devido ao *princípio da localidade* existente em códigos que são estruturados de forma modular.

A *localidade* é a tendência de o processador, ao longo da execução de um programa, referenciar

instruções e dados na memória principal localizados em endereços próximos. Esta tendência é justificada pela alta incidência de estruturas de repetição, chamadas a sub-rotinas e acesso a estruturas de dados como vetores e tabelas. O princípio da localidade garante, então, que após a transferência de um novo bloco da memória principal para a cache haverá uma alta probabilidade de cache hits em futuras referências, otimizando, assim, o tempo de acesso ao dado.

A maioria dos processadores apresenta uma arquitetura de memória cache com múltiplos níveis. O funcionamento desta arquitetura tem como base o princípio de que quanto menor é a capacidade de armazenamento da memória cache, mais rápido é o acesso ao dado; contudo, a probabilidade da ocorrência de cache hits é menor.

A hierarquização da cache em múltiplos níveis é uma solução para aumentar o desempenho no funcionamento das memórias caches. O nível de cache mais alto é chamado de L1 (*Level 1*), com baixa capacidade de armazenamento e com altíssima velocidade de acesso. O segundo nível, L2 (*Level 2*), possui maior capacidade de armazenamento, porém com velocidade de acesso inferior a L1. Quando a UCP necessita acessar um dado na memória principal, primeiramente é verificado se o dado encontra-se na cache L1. Caso o dado seja encontrado, obtém-se um excelente desempenho no acesso à informação, porém, caso o dado não seja encontrado, a busca prossegue para a L2.

Um processador pode ser projetado com diversos níveis de cache, conforme especificação do fabricante. Como exemplos, o processador da Motorola Power PC G4 possui cache de nível 2, enquanto o processador Intel Xeon MP possui cache de nível 3.

2.2.4 Memória Secundária

A *memória secundária* é um meio permanente, isto é, não volátil de armazenamento de programas e dados. Enquanto a memória principal precisa estar sempre energizada para manter suas informações, a memória secundária não precisa de alimentação.

O acesso à memória secundária é lento, se comparado com o acesso à memória principal, porém seu custo é baixo e sua capacidade de armazenamento é bem superior. Enquanto a unidade de acesso à memória secundária é da ordem de milissegundos, o acesso à memória principal é de nanossegundos. Podemos citar, como exemplos de memórias secundárias, a fita magnética, o disco magnético e o disco óptico.

A Fig. 2.3 mostra a relação entre os diversos tipos de dispositivos de armazenamento apresentados, comparando custo, velocidade e capacidade de armazenamento.

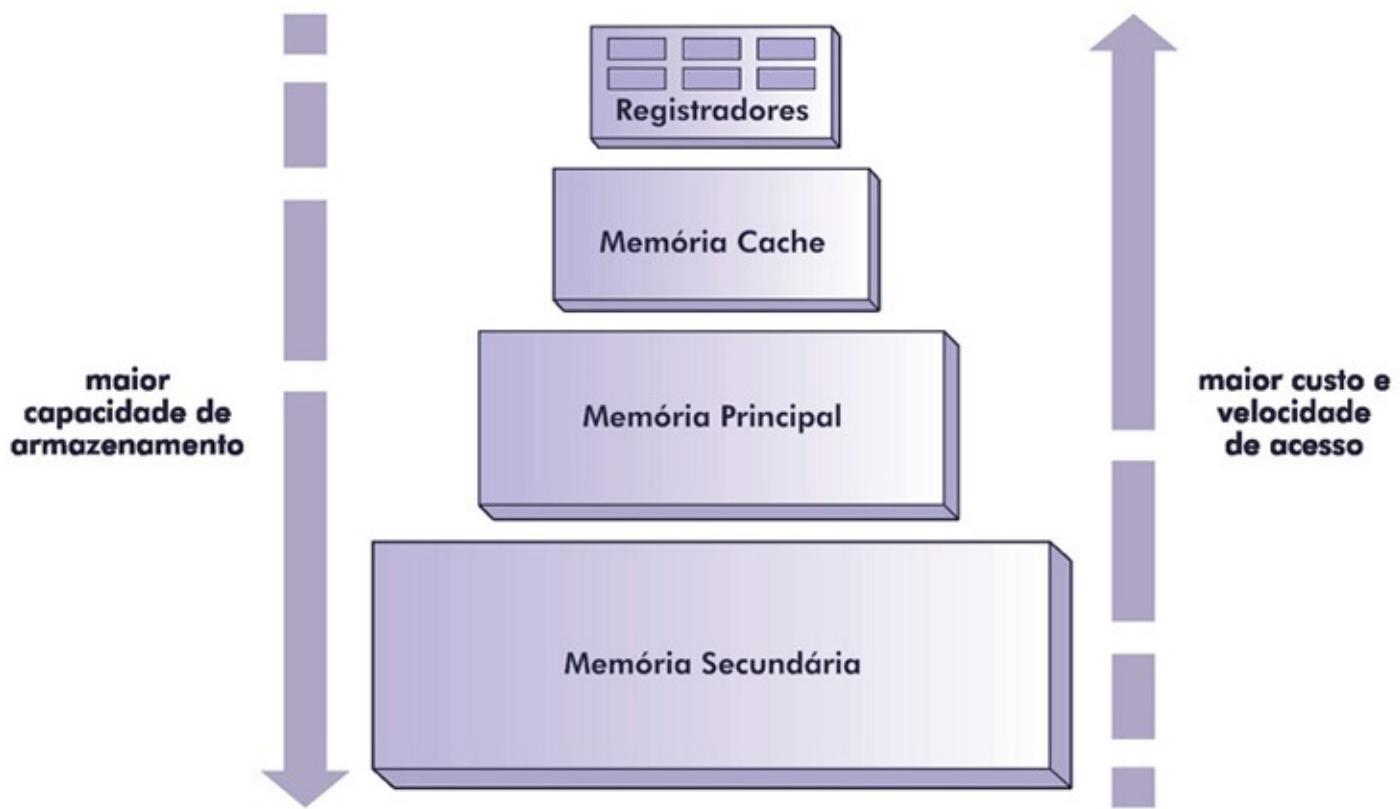


Fig. 2.3 Relação entre dispositivos de armazenamento.

2.2.5 Dispositivos de Entrada e Saída

Os *dispositivos de entrada e saída (E/S)* são utilizados para permitir a comunicação entre o sistema computacional e o mundo externo, e podem ser divididos em duas categorias: os que são utilizados como memória secundária e os que servem para a interface usuário-máquina.

Os dispositivos utilizados como memória secundária (discos e fitas magnéticas) caracterizam-se por ter capacidade de armazenamento bastante superior ao da memória principal. Seu custo é relativamente baixo, porém o tempo de acesso à memória secundária é bem superior ao da memória principal.

Outros dispositivos têm como finalidade a comunicação usuário-máquina, como teclados, monitores de vídeo, impressoras e plotters. A implementação de interfaces mais amigáveis permite, cada vez mais, que usuários pouco especializados utilizem computadores de maneira intuitiva. Scanner, caneta ótica, mouse, dispositivos sensíveis a voz humana e tato são alguns exemplos desses tipos de dispositivos.

2.2.6 Barramento

O *barramento ou bus* é um meio de comunicação compartilhado que permite a comunicação entre as unidades funcionais de um sistema computacional. Através de condutores, informações como dados, endereços e sinais de controle trafegam entre processadores, memórias e dispositivos de E/S.

Em geral, um barramento possui linhas de controle e linhas de dados. Através das linhas de controle trafegam informações de sinalização como, por exemplo, o tipo de operação que está sendo realizada.

Pelas linhas de dados, informações como instruções, operandos e endereços são transferidos entre unidades funcionais.

Os barramentos são classificados em três tipos: *barramentos processador-memória*, *barramentos de E/S* e *barramentos de backplane*. Os barramentos processador-memória são de curta extensão e alta velocidade para que seja otimizada a transferência de informação entre processadores e memórias. Diferentemente, os barramentos de E/S possuem maior extensão, são mais lentos e permitem a conexão de diferentes dispositivos.

A Fig. 2.4 ilustra a arquitetura de um sistema computacional utilizando barramentos processador-memória e de E/S. É importante notar na ilustração que na conexão dos barramentos existe um adaptador (bus adapter) que permite compatibilizar as diferentes velocidades dos barramentos.

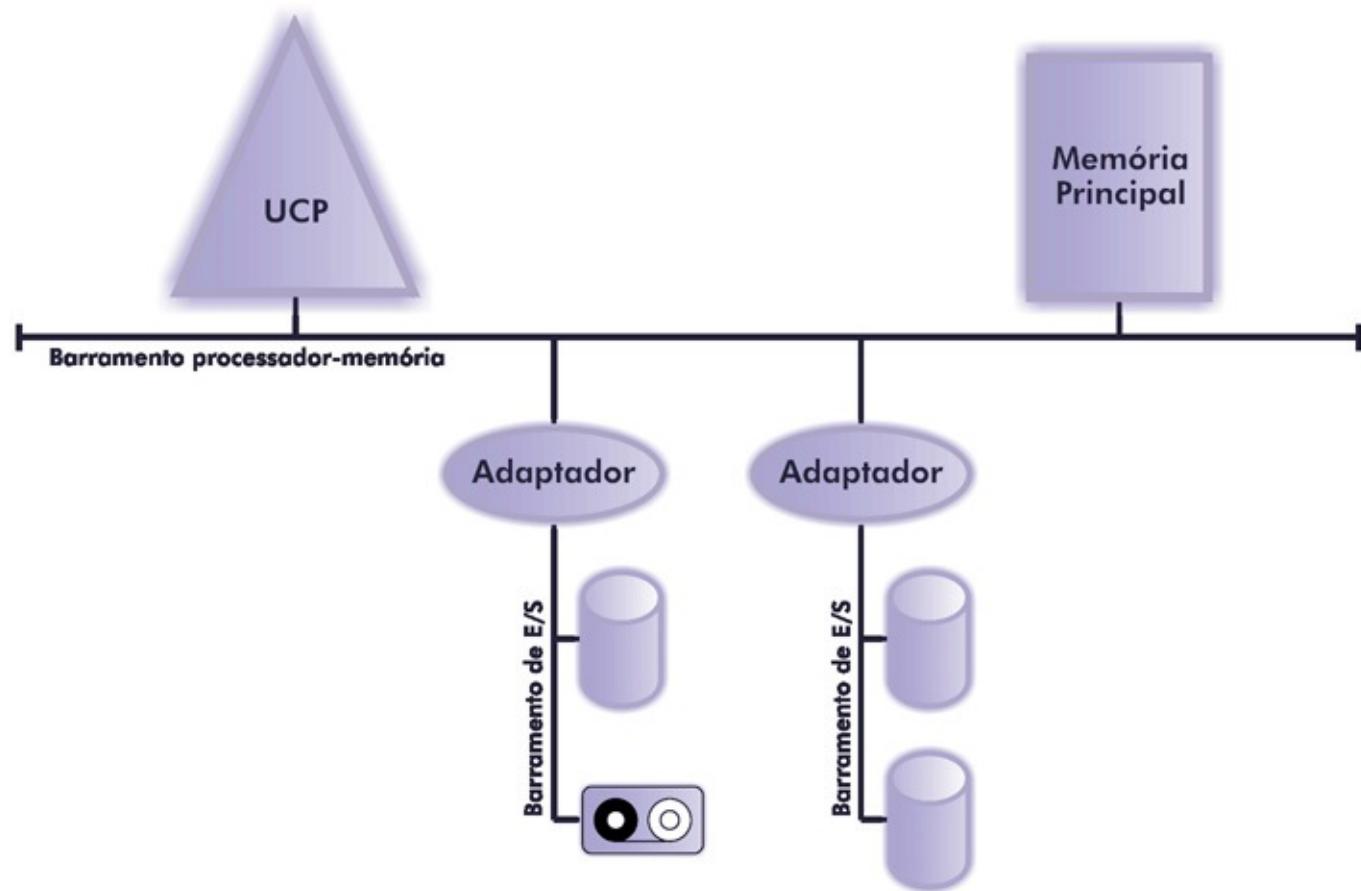


Fig. 2.4 Barramentos processador-memória e de E/S.

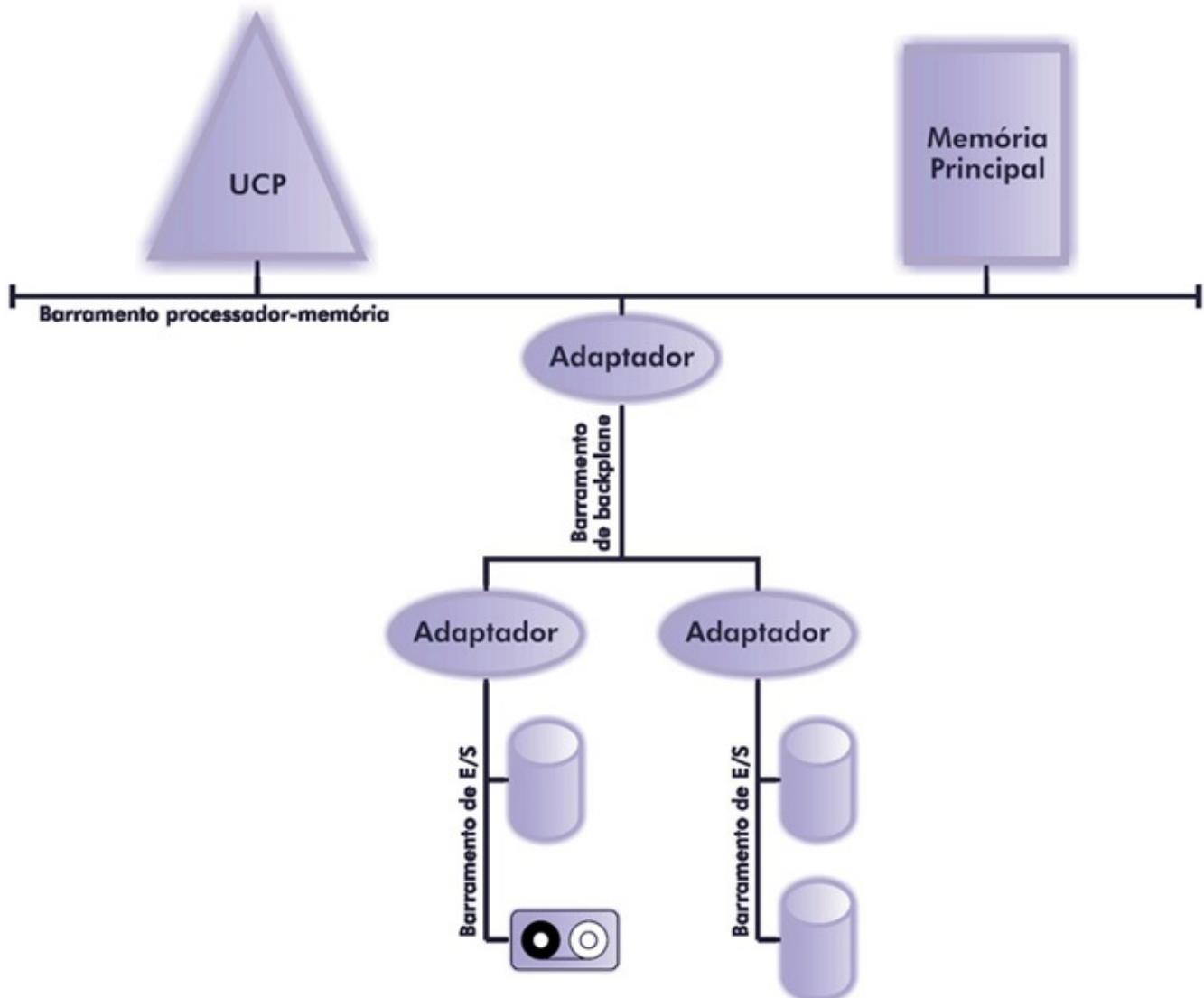


Fig. 2.5 Barramento de backplane.

Alguns sistemas de alto desempenho utilizam uma arquitetura com um terceiro barramento, conhecido como barramento de backplane (Fig. 2.5). Nesta organização, o barramento de E/S não se conecta diretamente ao barramento processador-memória, tendo o barramento de backplane a função de integrar os dois barramentos. A principal vantagem desta arquitetura é reduzir o número de adaptadores existentes no barramento processador-memória e, desta forma, otimizar seu desempenho.

Os barramentos processador-memória são frequentemente utilizados em arquiteturas proprietárias. Como exemplo, a Intel introduziu o barramento PCI (Peripheral Component Interconnect) junto com o seu processador Pentium. Os barramentos de E/S em geral seguem padrões preestabelecidos, pois, desta forma, os dispositivos de E/S podem ser conectados a sistemas computacionais de diferentes plataformas e fabricantes. O padrão SCSI (Small Computer System Interface) é um dos mais utilizados.

2.2.7 Pipelining

Pipelining é uma técnica que permite ao processador executar múltiplas instruções paralelamente em estágios diferentes. O conceito de processamento pipeline se assemelha muito a uma linha de montagem, onde uma tarefa é dividida em uma sequência de subtarefas, executadas dentro da linha de produção.

Da mesma forma que em uma linha de montagem, a execução de uma instrução pode ser dividida em subtarefas, como as fases de busca da instrução e dos operandos, execução e armazenamento dos resultados. O processador, através de suas várias unidades funcionais pipeline, funciona de forma a permitir que, enquanto uma instrução se encontra na fase de execução, uma outra instrução possa estar na fase de busca simultaneamente. A Fig. 2.6 ilustra um processador com quatro unidades funcionais pipeline, permitindo que quatro instruções sejam executadas em paralelo, porém em estágios diferentes.

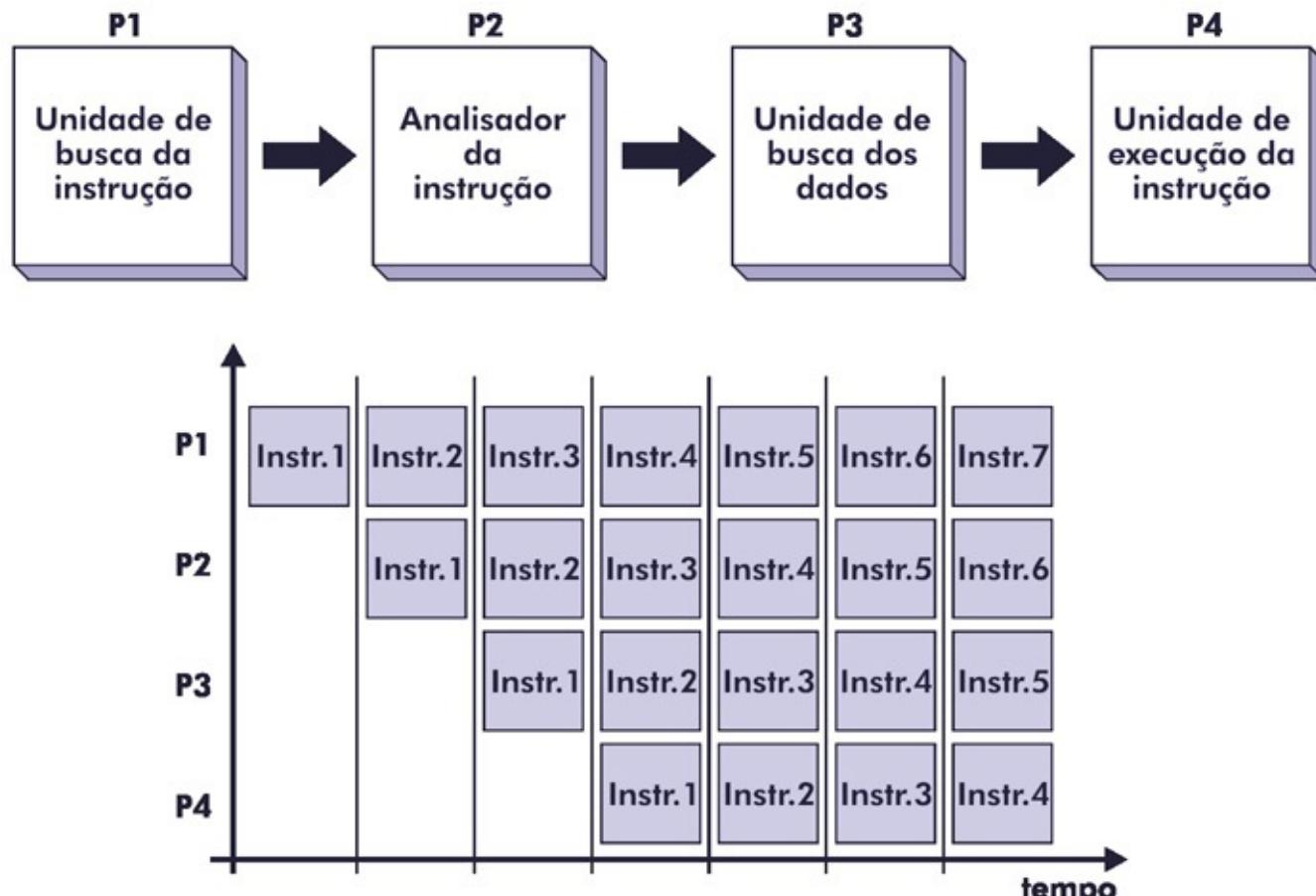


Fig. 2.6 Arquitetura pipeline com quatro estágios.

O pipelining pode ser empregado em sistemas com um ou mais processadores, em diversos níveis, e tem sido a técnica de paralelismo mais utilizada para aumentar o desempenho dos sistemas computacionais.

2.2.8 Arquiteturas RISC e CISC

A *linguagem de máquina* de um computador é a linguagem de programação que é realmente entendida pelo processador. Cada processador possui um conjunto definido de instruções de máquina, definido por seu fabricante. As instruções de máquina fazem referências a detalhes, como registradores, modos de endereçamento e tipos de dados, que caracterizam um processador e suas funcionalidades.

Um programa em linguagem de máquina pode ser diretamente executado pelo processador, não requerendo qualquer tipo de tradução ou relocação. Quando escrito em linguagem de máquina de um determinado processador, um programa não pode ser executado em outra máquina de arquitetura diferente, visto que o conjunto de instruções de um processador é característica específica de cada

arquitetura.

Um processador com arquitetura *RISC* (Reduced Instruction Set Computer) caracteriza-se por possuir poucas instruções de máquina, em geral bastante simples, que são executadas diretamente pelo hardware. Na sua maioria, estas instruções não acessam a memória principal, trabalhando principalmente com registradores que, neste tipo de processador, se apresentam em grande número. Estas características, além de permitirem que as instruções sejam executadas rapidamente, facilitam a implementação do pipelining. Como exemplos de processadores RISC podemos citar o SPARC (Sun), RS-6000 (IBM), PA-RISC (HP), Alpha AXP (Compaq) e Rx000 (MIPS).

Os processadores com arquitetura *CISC* (Complex Instruction Set Computers) já possuem instruções complexas que são interpretadas por microprogramas. O número de registradores é pequeno, e qualquer instrução pode referenciar a memória principal. Neste tipo de arquitetura, a implementação do pipelining é mais difícil. São exemplos de processadores CISC o VAX (DEC), Pentium (Intel) e 68xxx (Motorola). A Tabela 2.3 compara características presentes nas arquiteturas RISC e CISC.

Tabela 2.3 Arquitetura RISC × Arquitetura CISC

Arquitetura RISC	Arquitetura CISC
Poucas instruções	Muitas instruções
Instruções executadas pelo hardware	Instruções executadas por microcódigo
Instruções com formato fixo	Instruções com diversos formatos
Instruções utilizam poucos ciclos de máquina	Instruções utilizam múltiplos ciclos
Instruções com poucos modos de endereçamento	Instruções com diversos modos de endereçamento
Arquitetura com muitos registradores	Arquitetura com poucos registradores
Arquitetura pipelining	Pouco uso da técnica de pipelining

Nos processadores RISC, um programa em linguagem de máquina é executado diretamente pelo hardware, porém isto não ocorre nos processadores CISC. Podemos observar, na Fig. 2.7, que entre os níveis da linguagem de máquina e de circuitos eletrônicos existe um nível intermediário que é o de *microprogramação*.

Os microprogramas definem a linguagem de máquina de um computador CISC. Apesar de cada computador possuir níveis de microprogramação diferentes, existem muitas semelhanças nessa camada se compararmos os diferentes equipamentos. Um computador possui, aproximadamente, 25 microinstruções básicas, que são interpretadas pelos circuitos eletrônicos.

Na realidade, o código executável de um processador CISC é interpretado por microprogramas durante sua execução, gerando *microinstruções*, que, finalmente, são executadas pelo hardware. Para cada instrução em linguagem de máquina existe um microprograma associado.

Os processadores chamados microprogramáveis são aqueles que permitem que novas instruções de máquina possam ser criadas através da criação de microprogramas. Alguns equipamentos, como os microprocessadores, não dispõem desse recurso, já que os microprogramas vêm gravados em memória do tipo ROM.



Fig. 2.7 Máquina de níveis.

► 2.3 Software

Para que o hardware tenha utilidade prática, deve existir um conjunto de programas utilizado como interface entre as necessidades do usuário e as capacidades do hardware. A utilização de softwares adequados às diversas tarefas e aplicações torna o trabalho dos usuários muito mais simples e eficiente.

No decorrer do texto, utilizaremos o termo *utilitário* sempre que desejarmos fazer referência a softwares relacionados mais diretamente com serviços complementares do sistema operacional, como compiladores, linkers e depuradores. Os softwares desenvolvidos pelos usuários serão denominados aplicativos ou apenas aplicações.

2.3.1 Tradutor

Nos primeiros sistemas computacionais, o ato de programar era bastante complicado, já que o programador deveria possuir conhecimento da arquitetura da máquina e programar em painéis através de fios. Esses programas eram desenvolvidos em linguagem de máquina e carregados diretamente na memória principal para execução.

Com o surgimento das primeiras *linguagens de montagem* ou *assembly* e das *linguagens de alto nível*, o programador deixou de se preocupar com muitos aspectos pertinentes ao hardware, como em qual região da memória o programa deveria ser carregado ou quais endereços de memória seriam reservados para as variáveis. A utilização dessas linguagens facilitou a construção de programas, documentação e manutenção.

Apesar das inúmeras vantagens proporcionadas pelas linguagens de montagem e de alto nível, os programas escritos nessas linguagens não estão prontos para ser diretamente executados pelo processador (programas-fonte). Para isso, eles têm de passar por uma etapa de conversão, onde toda representação simbólica das instruções é traduzida para código de máquina. Esta conversão é realizada por um utilitário denominado *tradutor*.

O módulo gerado pelo tradutor é denominado *módulo-objeto*, que, apesar de estar em código de máquina, na maioria das vezes não pode ser ainda executado. Isso ocorre em função de um programa poder chamar sub-rotinas externas, e, neste caso, o tradutor não tem como associar o programa principal às sub-rotinas chamadas. Esta função é realizada por outro utilitário denominado *linker*, e será apresentado adiante.

Dependendo do tipo do programa-fonte, existem dois tipos distintos de tradutores que geram módulos-objeto: montador e compilador (Fig. 2.8).

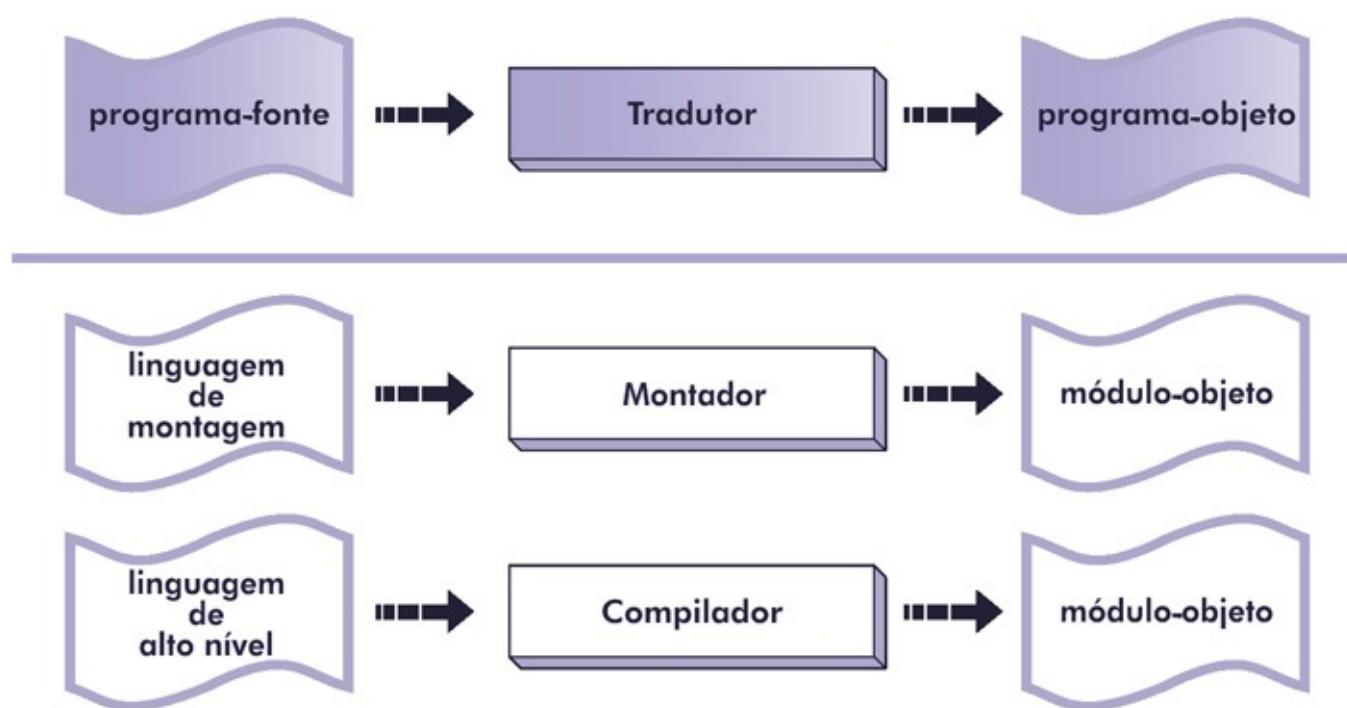


Fig. 2.8 Tradutor.

O *montador (assembler)* é o utilitário responsável por traduzir um programa-fonte em linguagem de montagem em um programa-objeto não executável (módulo-objeto). A linguagem de montagem é particular para cada processador, assim como a linguagem de máquina, o que não permite que programas assembly possam ser portados entre máquinas diferentes.

O *compilador* é o utilitário responsável por gerar, a partir de um programa escrito em uma linguagem de alto nível, um programa em linguagem de máquina não executável. As linguagens de alto nível, como

Pascal, FORTRAN e COBOL, não têm nenhuma relação direta com a máquina, ficando essa preocupação exclusivamente com o compilador. Os programadores de alto nível devem se preocupar apenas com o desenvolvimento de suas aplicações, não tendo que se envolver com detalhes sobre a arquitetura do processador. Assim, os programas-fonte podem ser portados entre computadores de diversos fabricantes, desde que existam padrões para a sintaxe da linguagem. Isso permite o desenvolvimento de aplicações independentes do equipamento.

Um compilador é um utilitário que opera de modo integrado aos componentes do sistema de programação disponíveis, sob a supervisão do sistema operacional (Neto, 1987). Podemos visualizar, então, o compilador como uma interface entre o sistema operacional e o usuário, de maneira que é possível acessar diversos serviços do sistema sem a necessidade da utilização de linguagem de controle ou de outros utilitários.

2.3.2 Interpretador

O *interpretador* é considerado um tradutor que não gera módulo-objeto. A partir de um programa-fonte escrito em linguagem de alto nível, o interpretador, durante a execução do programa, traduz cada instrução e a executa imediatamente. Algumas linguagens tipicamente interpretadas são o Basic e o Perl.

A maior desvantagem na utilização de interpretadores é o tempo gasto na tradução das instruções de um programa toda vez que este for executado, já que não existe a geração de um código executável. A vantagem é permitir a implementação de tipos de dados dinâmicos, ou seja, que podem mudar de tipo durante a execução do programa, aumentando, assim, sua flexibilidade.

2.3.3 Linker

O *linker* ou *editor de ligação* é o utilitário responsável por gerar, a partir de um ou mais módulos-objeto, um único programa executável (Fig. 2.9). Suas funções básicas são resolver todas as referências simbólicas existentes entre os módulos e reservar memória para a execução do programa.

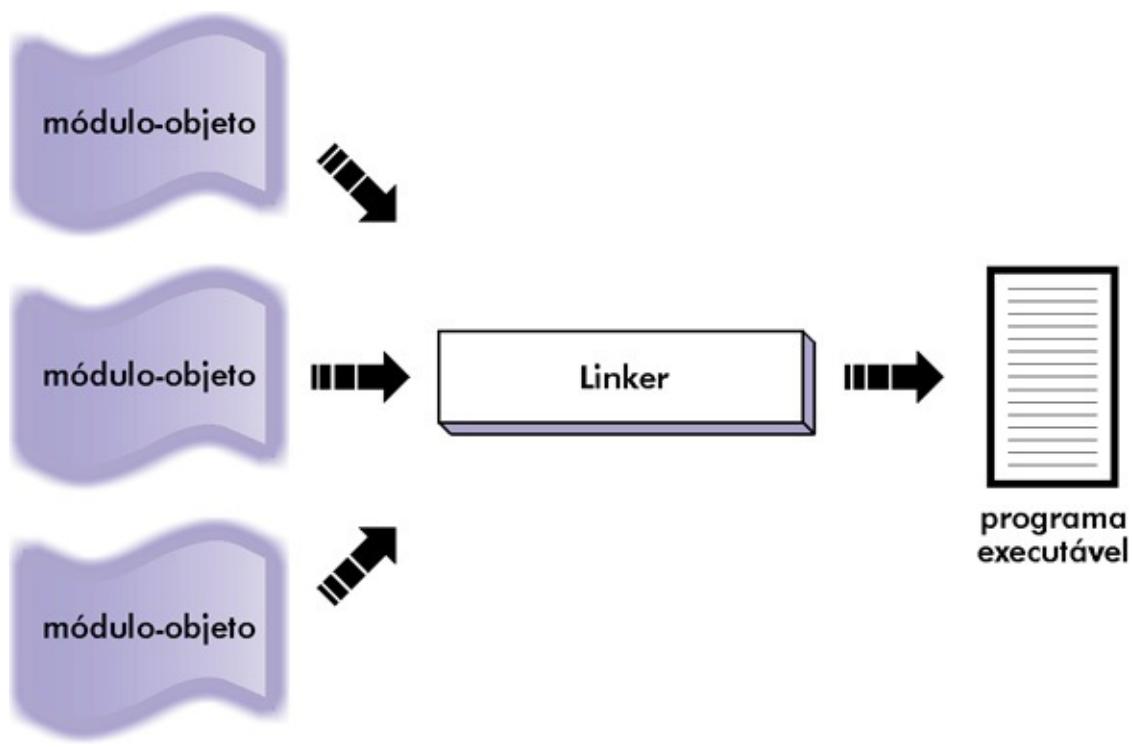


Fig. 2.9 Linker.

Para resolver todas as referências a símbolos, o linker também pode pesquisar em bibliotecas do sistema ou do próprio usuário. *Bibliotecas* são arquivos que contêm diversos módulos-objeto e/ou definições de símbolos.

Outra função importante do linker é a *relocação*, que determina a região de memória na qual o programa será carregado para execução. Nos primeiros sistemas operacionais, a relocação era realizada somente uma vez, na etapa de linkedição. Todos os endereços simbólicos do programa eram traduzidos para endereços físicos, e o programa executável gerado podia ser carregado a partir de uma posição prefixada na memória (código absoluto). Nesse tipo de relocação, o programa poderia ser carregado, apenas, a partir de uma única posição na memória.

Em sistemas multiprogramáveis esse tipo de relocação é de difícil implementação, já que a memória é compartilhada entre diversos programas, e é pouco provável que, no momento em que o sistema carrega um programa, sua área de memória prefixada esteja disponível. A solução para este problema é permitir que o programa seja carregado em regiões diferentes toda vez que for trazido para a memória (código relocável). Esse tipo de relocação não é realizado pelo linker, mas sim por outro utilitário denominado *loader*, responsável por carregar os programas na memória.

Em alguns sistemas, o compilador pode realizar mais do que suas funções básicas, como resolver referências de endereço, sendo que, dessa forma, o programa não necessita passar pela etapa de linkedição independente. Nesse caso, um outro utilitário (linking loader) fica responsável pela carga do programa na memória e pela sua execução. Nesse tipo de sistema, o programa passa por um processo de *link dinâmico*, onde as referências globais são resolvidas no momento da execução. A grande desvantagem dessa implementação é a sobrecarga de processamento (overhead) existente em cada execução do programa, problema que não ocorre em sistemas que possuem linkers independentes (Levy, 1980).

2.3.4 Loader

O *loader* ou *carregador* é o utilitário responsável por carregar na memória principal um programa para ser executado. O procedimento de carga varia com o código gerado pelo linker e, em função deste, o loader é classificado como do tipo absoluto ou relocável.

Se o código executável for do tipo absoluto, o loader só necessita conhecer o endereço de memória inicial e o tamanho do módulo para realizar o carregamento. Então, o loader transfere o programa da memória secundária para a memória principal e inicia sua execução (*loader absoluto*).

No caso do código relocável, o programa pode ser carregado em qualquer posição de memória, e o loader é responsável pela relocação no momento do carregamento (*loader relocável*).

2.3.5 Depurador

O desenvolvimento de programas está sujeito a erros de lógica, independentemente de metodologias utilizadas pelo programador. A depuração é um dos estágios desse desenvolvimento, e a utilização de ferramentas adequadas é essencial para acelerar o processo de correção dos programas. O *depurador* (debugger) é o utilitário que permite ao usuário acompanhar toda a execução de um programa a fim de detectar erros na sua lógica. Este utilitário oferece ao usuário recursos como:

- acompanhar a execução de um programa instrução por instrução;
- possibilitar a alteração e a visualização do conteúdo de variáveis;
- implementar pontos de parada dentro do programa (breakpoint), de forma que, durante a execução, o programa pare nesses pontos;
- especificar que, toda vez que o conteúdo de uma variável for modificado, o programa envie uma mensagem (watchpoint).

► 2.4 Exercícios

1. Quais são as unidades funcionais de um sistema computacional?
2. Quais os componentes de um processador e quais são suas funções?
3. Como a memória principal de um computador é organizada?
4. Descreva os ciclos de leitura e gravação da memória principal.
5. Qual o número máximo de células endereçadas em arquiteturas com MAR de 16, 32 e 64 bits?
6. O que são memórias voláteis e não voláteis?
7. Conceitue memória cache e apresente as principais vantagens no seu uso.
8. Qual a importância do princípio da localidade na eficiência da memória cache?
9. Quais os benefícios de uma arquitetura de memória cache com múltiplos níveis?
10. Quais as diferenças entre a memória principal e a memória secundária?

11. Diferencie as funções básicas dos dispositivos de E/S.
12. Caracterize os barramentos processador-memória, E/S e backplane.
13. Como a técnica de pipelining melhora o desempenho dos sistemas computacionais?
14. Compare as arquiteturas de processadores RISC e CISC.
15. Conceitue a técnica de benchmark e como é sua realização.
16. Por que o código-objeto gerado pelo tradutor ainda não pode ser executado?
17. Por que a execução de programas interpretados é mais lenta que a de programas compilados?
18. Quais as funções do linker?
19. Qual a principal função do loader?
20. Quais as facilidades oferecidas pelo depurador?

3

Concorrência

► 3.1 Introdução

Sistemas operacionais podem ser vistos como um conjunto de rotinas executadas de forma concorrente e ordenada (Pinkert, 1990). A possibilidade de o processador executar instruções ao mesmo tempo que outras operações, como, por exemplo, operações de E/S, permite que diversas tarefas sejam executadas concorrentemente pelo sistema. O conceito de concorrência é o princípio básico para o projeto e a implementação dos sistemas multiprogramáveis.

Neste capítulo serão apresentados mecanismos, técnicas e dispositivos que possibilitam a implementação da concorrência como interrupções e exceções, buffering, spooling e reentrância. Todos estes conceitos são fundamentais para a arquitetura de um sistema operacional multiprogramável. Inicialmente será apresentada uma comparação entre os sistemas monoprogramáveis e multiprogramáveis para ilustrar a importância do conceito de concorrência.

► 3.2 Sistemas Monoprogramáveis x Multiprogramáveis

Os sistemas multiprogramáveis surgiram a partir de limitações existentes nos sistemas operacionais monoprogramáveis. Neste tipo de sistema, os recursos computacionais, como processador, memória e dispositivos de E/S, eram utilizados de maneira pouco eficiente, limitando o desempenho destas arquiteturas. Muitos destes recursos de alto custo permaneciam muitas vezes ociosos por longos períodos de tempo.

Nos sistemas monoprogramáveis somente um programa pode estar em execução por vez, permanecendo o processador dedicado, exclusivamente, a essa tarefa. Podemos observar que, nesse tipo de sistema, ocorre um desperdício na utilização do processador (Fig. 3.1a), pois enquanto uma leitura em disco é realizada, o processador permanece ocioso. O tempo de espera é relativamente longo, já que as operações com dispositivos de entrada e saída são muito lentas, se comparadas com a velocidade do processador em executar instruções.

A Tabela 3.1 apresenta um exemplo de um programa que lê registros de um arquivo e executa, em média, 100 instruções por registro lido. Neste caso, o processador gasta aproximadamente 93% do tempo esperando o dispositivo de E/S concluir a operação para continuar o processamento (Stallings, 1997). Em sistemas monoprogramáveis o processador é subutilizado, permanecendo livre grande parte do tempo.

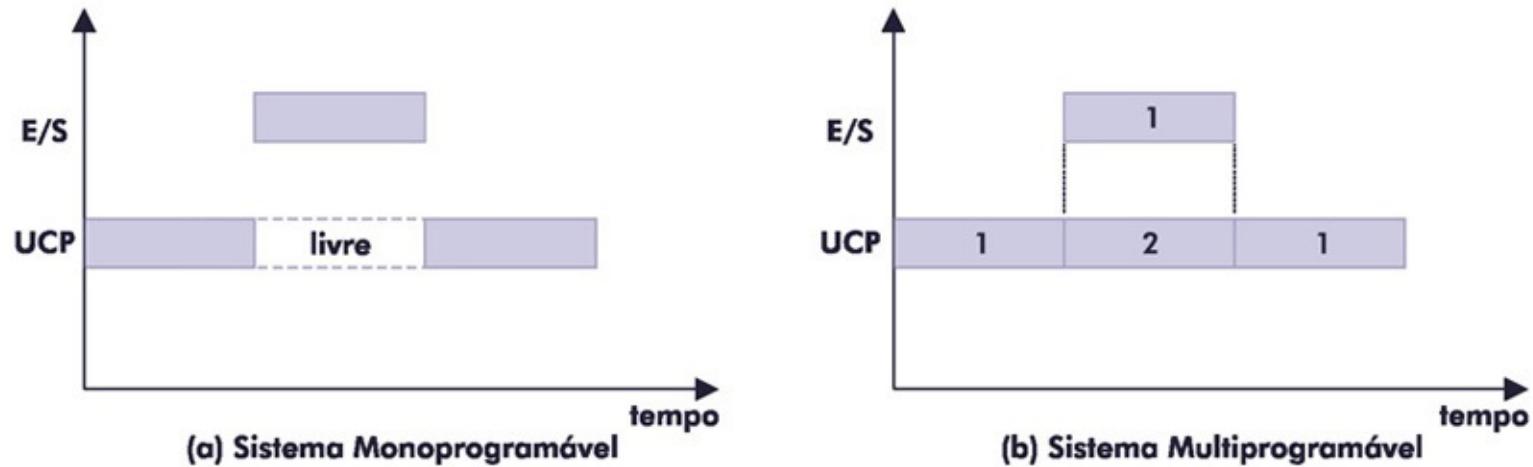


Fig. 3.1 Sistema monoprocessável x sistema multiprocessável.

Tabela 3.1 Exemplo de utilização do sistema

Leitura de um registro	0,0015 s
Execução de 100 instruções	0,0001 s
Total	0,0016 s
$\% \text{ utilização da CPU} (0,0001 / 0,0015) = 0,066 = 6,6\%$	

Outro aspecto a ser considerado é a subutilização da memória principal. Um programa que não ocupe totalmente a memória ocasiona a existência de áreas livres sem utilização. Nos sistemas multiprocessáveis, vários programas podem estar residentes em memória, concorrendo pela utilização do processador. Dessa forma, quando um programa solicita uma operação de E/S outros programas poderão utilizar o processador. Nesse caso, a UCP permanece menos tempo ociosa (Fig. 3.1b) e a memória principal é utilizada de forma mais eficiente, pois existem vários programas residentes se revezando na utilização do processador.

A utilização concorrente da UCP deve ser implementada de maneira que, quando um programa perde o uso do processador e depois retorna para continuar o processamento, seu estado deve ser idêntico ao do momento em que foi interrompido. O programa deverá continuar sua execução exatamente na instrução seguinte àquela em que havia parado, aparentando ao usuário que nada aconteceu. Em sistemas de tempo compartilhado existe a impressão de que o computador está inteiramente dedicado ao usuário, ficando esse mecanismo totalmente transparente.

No caso de periféricos, é comum termos, em sistemas monoprogramáveis, impressoras paradas por um

grande período de tempo e discos com acesso restrito a um único usuário. Esses problemas são solucionados em sistemas multiprogramáveis, onde é possível compartilhar dispositivos de E/S, como impressoras e discos, entre diversos aplicativos.

As vantagens proporcionadas pela multiprogramação podem ser percebidas onde existe um sistema computacional com um disco, um terminal e uma impressora, como mostra a Tabela 3.2. Neste ambiente são executados três programas (Prog1, Prog2 e Prog3), que possuem características de processamento descritas na Tabela 3.2, na qual podemos notar que o Prog1 não realiza operações de E/S, ao contrário de Prog2 e Prog3 (Stallings, 1997).

Em um ambiente monoprogramável, os programas são executados sequencialmente. Assim, o Prog1 é processado em cinco minutos, enquanto o Prog2 espera para começar sua execução, que leva 15 minutos. Finalmente, o Prog3 inicia sua execução após 20 minutos e completa seu processamento em 10 minutos, totalizando 30 minutos na execução dos três programas. No caso de os programas serem executados concorrentemente, em um sistema multiprogramável, o ganho na utilização do processador, memória, periféricos e também no tempo de resposta é considerável, como mostra a Tabela 3.3.

Tabela 3.2 Características de execução dos programas

Características	Prog1	Prog2	Prog3
Utilização da UCP	Alta	Baixa	Baixa
Operação de E/S	Poucas	Muitas	Muitas
Tempo de processamento	5 min	15 min	10 min
Memória utilizada	50 Kb	100 Kb	80 Kb
Utilização de disco	Não	Não	Sim
Utilização de terminal	Não	Sim	Não
Utilização de impressora	Não	Não	Sim

Tabela 3.3 Comparação entre monoprogramação e multiprogramação

	Monoprogramação	Multiprogramação
Utilização da UCP	17%	33%
Utilização da memória	30%	67%
Utilização de disco	33%	67%
Utilização de impressora	33%	67%
Tempo total de processamento	30 min	15 min
Taxa de throughput	6 prog./hora	12 prog./hora

► 3.3 Interrupções e Exceções

Durante a execução de um programa podem ocorrer alguns eventos inesperados, ocasionando um desvio forçado no seu fluxo de execução. Estes tipos de eventos são conhecidos por *interrupção* ou *exceção* e podem ser consequência da sinalização de algum dispositivo de hardware externo ao processador ou da execução de instruções do próprio programa. A diferença entre interrupção e exceção é dada pelo tipo de evento ocorrido, porém alguns autores e fabricantes não fazem esta distinção.

A interrupção é o mecanismo que tornou possível a implementação da concorrência nos computadores, sendo o fundamento básico dos sistemas multiprogramáveis. É em função desse mecanismo que o sistema operacional sincroniza a execução de todas as suas rotinas e dos programas dos usuários, além de controlar dispositivos.

Uma interrupção é sempre gerada por algum evento externo ao programa e, nesse caso, independe da instrução que está sendo executada. Um exemplo de interrupção ocorre quando um dispositivo avisa ao processador que alguma operação de E/S está completa. Nesse caso, o processador deve interromper o programa para tratar o término da operação.

Ao final da execução de cada instrução, a unidade de controle verifica a ocorrência de algum tipo de interrupção. Nesse caso, o programa em execução é interrompido e o controle desviado para uma rotina responsável por tratar o evento ocorrido, denominada *rotina de tratamento de interrupção*. Para que o programa possa posteriormente voltar a ser executado é necessário que, no momento da interrupção, um conjunto de informações sobre a sua execução seja preservado. Essas informações consistem no conteúdo de registradores, que deverão ser restaurados para a continuação do programa (Fig. 3.2).



Fig. 3.2 Mecanismos de interrupção e exceção.

A Tabela 3.4 descreve como é o mecanismo de interrupção. Este mecanismo é realizado tanto por hardware quanto por software.

Tabela 3.4 Mecanismo de interrupção

	<ol style="list-style-type: none">1. Um sinal de interrupção é gerado para o processador.2. Após o término da execução da instrução corrente, o processador identifica o pedido de interrupção.3. Os conteúdos dos registradores PC e de status são salvos.4. O processador identifica qual a rotina de tratamento que será executada e carrega o PC com o endereço inicial desta rotina.
Via hardware	<ol style="list-style-type: none">5. A rotina de tratamento salva o conteúdo dos demais registradores do processador na pilha de controle do programa.6. A rotina de tratamento é executada.
Via software	<ol style="list-style-type: none">7. Após o término da execução da rotina de tratamento, os registradores de uso geral são restaurados, além do registrador de status e o PC, retornando à execução do programa interrompido.

Para cada tipo de interrupção existe uma rotina de tratamento associada, para a qual o fluxo de execução deve ser desviado. A identificação do tipo de evento ocorrido é fundamental para determinar o endereço da rotina de tratamento. No momento da ocorrência de uma interrupção, o processador deve saber para qual rotina de tratamento deve ser desviado o fluxo de execução.

Existem dois métodos para o tratamento de interrupções. O primeiro método utiliza uma estrutura de dados chamada *vetor de interrupção*, que contém o endereço inicial de todas as rotinas de tratamento existentes associadas a cada tipo de evento. Um segundo método utiliza um registrador de status que armazena o tipo do evento ocorrido. Neste método só existe uma única rotina de tratamento que, no seu início, testa o registrador para identificar o tipo de interrupção e tratá-la de maneira adequada.

As interrupções são decorrentes de eventos *assíncronos*, ou seja, não relacionados à instrução do programa corrente. Esses eventos, por serem imprevisíveis, podem ocorrer múltiplas vezes, como no caso de diversos dispositivos de E/S informarem ao processador que estão prontos para receber ou transmitir dados. Isso possibilita a ocorrência de múltiplas interrupções simultâneas. Uma maneira de evitar esta situação é a rotina de tratamento inibir as demais interrupções. Nesse caso, na ocorrência de outras interrupções durante a execução da rotina de tratamento elas serão ignoradas, ou seja, não receberão tratamento. Interrupções com esta característica são denominadas *interrupções mascaráveis*.

Alguns processadores não permitem que interrupções sejam desabilitadas, fazendo com que exista um tratamento para a ocorrência de múltiplas interrupções. Nesse caso, o processador necessita saber qual a ordem de atendimento que deverá seguir. Para isso, as interrupções deverão possuir prioridades, em função da importância no atendimento de cada uma. Normalmente, existe um dispositivo denominado *controlador de pedidos de interrupção*, responsável por avaliar as interrupções geradas e suas prioridades de atendimento.

Uma exceção é semelhante a uma interrupção, sendo a principal diferença o motivo pelo qual o evento é gerado. A exceção é resultado direto da execução de uma instrução do próprio programa, como a divisão de um número por zero ou a ocorrência de overflow em uma operação aritmética.

A diferença fundamental entre exceção e interrupção é que a primeira é gerada por um evento síncrono, enquanto a segunda é gerada por eventos assíncronos. Um evento é *síncrono* quando é resultado direto da execução do programa corrente. Tais eventos são previsíveis e, por definição, só podem ocorrer um de cada vez. Se um programa que causa esse tipo de evento for reexecutado com a mesma entrada de dados, a exceção ocorrerá sempre na mesma instrução.

Da mesma forma que na interrupção, sempre que uma exceção é gerada o programa em execução é interrompido e o controle é desviado para uma *rotina de tratamento de exceção* (Fig. 3.2). Para cada tipo de exceção existe uma rotina de tratamento adequada, para a qual o fluxo de execução deve ser desviado. O mecanismo de tratamento de exceções, muitas vezes, pode ser escrito pelo próprio programador. Desta forma, é possível evitar que um programa seja encerrado no caso de ocorrer, por exemplo, um overflow.

O programa DivisãoZero, escrito em Java, apresenta um exemplo de uma exceção do tipo divisão por zero. No momento em que o comando de divisão é executado, ocorre uma exceção, o programa é interrompido e a mensagem de erro “Exception in thread “main” java.lang.ArithmaticException: / by zero at DivisaoZero.main(DivisaoZero.java:7)” é apresentada pela máquina virtual Java.

```
public class DivisaoZero {  
    public static void main(String[] args) {  
        int a, b, c;  
        a = 2;  
        b = 0;  
        c = a / b;  
        System.out.print("Divisão de a por b = " + c);  
    }  
}
```

O programa DivisãoZeroTratamento, escrito em Java, também apresenta um exemplo de uma exceção do tipo divisão por zero, porém com tratamento de exceção a partir dos comandos try e catch. No momento em que o comando de divisão é executado, ocorre uma exceção, o programa é interrompido e a mensagem de erro “Erro: divisão por zero” é apresentada pelo programa. O mesmo programa poderia ser modificado para, além de identificar a exceção, solicitar ao usuário a entrada de outro número diferente de zero.

```
public class DivisaoZeroTratamento {  
    public static void main(String[] args) {  
        int a, b, c;  
        a = 2;  
        b = 0;  
        try {  
            c = a / b;  
            System.out.print("Divisão de a por b = " + c);  
        }  
        catch (ArithmaticException arithmeticException) {  
            System.out.print("Erro: divisão por zero");  
        }  
    }  
}
```

► 3.4 Operações de Entrada/Saída

Nos primeiros sistemas computacionais, a comunicação entre o processador e os periféricos era controlada por um conjunto de instruções especiais, denominadas *instruções de entrada/saída*, executadas pelo próprio processador. Essas instruções continham detalhes específicos de cada periférico, como em qual trilha e setor de um disco deveria ser lido ou gravado um determinado bloco de dados. Esse modelo criava uma forte dependência entre o processador e os dispositivos de E/S.

O surgimento do *controlador* ou *interface* permitiu ao processador agir de maneira independente dos dispositivos de E/S. Com esse novo elemento, o processador não se comunicava mais diretamente com os periféricos, mas sim através do controlador (Fig. 3.3). Isso simplificou as instruções de E/S por não ser mais necessário especificar detalhes de operação dos periféricos, tarefa esta realizada pelo controlador.

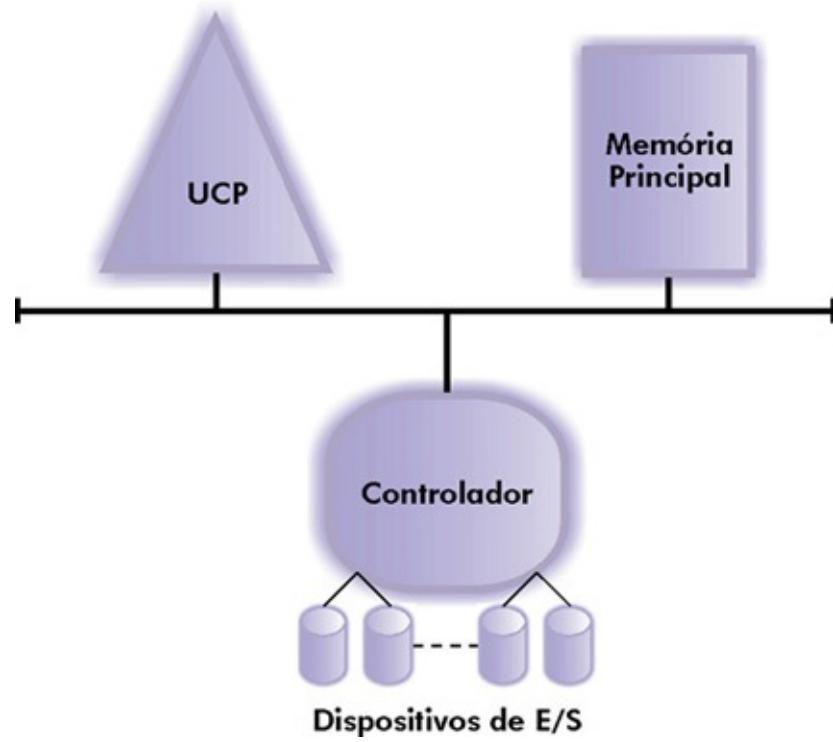


Fig. 3.3 Controlador.

Com a utilização do controlador, existiam duas maneiras básicas pelas quais o processador gerenciava as operações de E/S. Na primeira, o processador sincronizava-se com o periférico para o início da transferência de dados e, após iniciada a transferência, o sistema ficava permanentemente testando o estado do periférico para saber quando a operação chegaria ao seu final. Este controle, chamado *E/S controlada por programa*, mantinha o processador ocupado até o término da operação de E/S (*busy wait*). Como o processador executa uma instrução muito mais rapidamente que a realização de uma operação de E/S pelo controlador, havia um enorme desperdício de tempo da UCP.

A evolução do modelo anterior foi permitir que, após o início da transferência dos dados, o processador permanecesse livre para realizar outras tarefas. Assim, em determinados intervalos de tempo o sistema operacional deveria testar cada dispositivo para saber do término da operação de E/S (*polling*). Esse tipo de operação introduziu certo grau de paralelismo, visto que um programa poderia ser processado

enquanto outro esperava pelo término de uma operação de E/S (Guimarães, 1980). Isso permitiu o surgimento dos primeiros sistemas multiprogramáveis, em que vários programas poderiam ser executados concorrentemente, já que o tempo de uma operação de E/S é relativamente grande. O problema dessa implementação é que, no caso de existir um grande número de periféricos, o processamento é interrompido frequentemente para testar os diversos periféricos, já que é difícil determinar o momento exato do término das operações de E/S em andamento.

Com a implementação do mecanismo de interrupção, as operações de E/S puderam ser realizadas de uma forma mais eficiente. Em vez de o sistema periodicamente verificar o estado de uma operação pendente, o próprio controlador interrompia o processador para avisar do término da operação. Com esse mecanismo, denominado *E/S controlada por interrupção*, o processador, após a execução de um comando de leitura ou gravação, permanece livre para o processamento de outras tarefas. O controlador por sua vez, ao receber, por exemplo, um sinal de leitura, fica encarregado de ler os blocos do disco e armazená-los em memória ou registradores próprios. Em seguida, o controlador sinaliza uma interrupção ao processador. Quando o processador atende à interrupção, a rotina responsável pelo tratamento transfere os dados dos registradores do controlador para a memória principal. Ao término da transferência, o processador pode voltar a executar o programa interrompido e o controlador fica novamente disponível para outra operação.

A operação de E/S controlada por interrupção é muito mais eficiente que a controlada por programa, já que elimina a necessidade de o processador esperar pelo término da operação, além de permitir que várias operações de E/S sejam executadas simultaneamente. Apesar disso, a transferência de grande volume de dados exige muitas intervenções do processador, reduzindo sua eficiência. A solução para esse problema foi a implementação de uma técnica de transferência de dados denominada DMA (Direct Memory Access).

A *técnica de DMA* permite que um bloco de dados seja transferido entre a memória principal e dispositivos de E/S sem a intervenção do processador, exceto no início e no final da transferência. Quando o sistema deseja ler ou gravar um bloco de dados, o processador informa ao controlador sua localização, o dispositivo de E/S, a posição inicial da memória de onde os dados serão lidos ou gravados e o tamanho do bloco. Com estas informações, o controlador realiza a transferência entre o periférico e a memória principal, e o processador é somente interrompido no final da operação. A área de memória utilizada pelo controlador na técnica de DMA é chamada de *buffer de entrada e saída*.

No momento em que uma transferência de dados através da técnica de DMA é realizada, o controlador deve assumir, momentaneamente, o controle do barramento. Como a utilização do barramento é exclusiva de um dispositivo, o processador deve suspender o acesso ao bus, temporariamente, durante a operação de transferência. Este procedimento não gera uma interrupção, e o processador pode realizar tarefas, desde que sem a utilização do barramento, como um acesso à memória cache.

A extensão do conceito do DMA possibilitou o surgimento do *canal de entrada e saída* introduzido pela IBM no Sistema 7094. O canal é um processador com capacidade de executar programas de E/S, permitindo o controle total sobre operações de E/S. As instruções de E/S são armazenadas na memória principal pelo processador, porém o canal é responsável pela sua execução. Assim, o processador realiza uma operação de E/S, instruindo o canal para executar um programa localizado na memória (programa de canal). Este programa especifica os dispositivos para transferência, buffers e ações a

serem tomadas em caso de erros. O canal de E/S realiza a transferência e, ao final, gera uma interrupção, avisando do término da operação. Um canal de E/S pode controlar múltiplos dispositivos através de diversos controladores (Fig. 3.4). Cada dispositivo, ou conjunto de dispositivos, é manipulado por um único controlador. O canal atua como um elo entre o processador principal e o controlador.

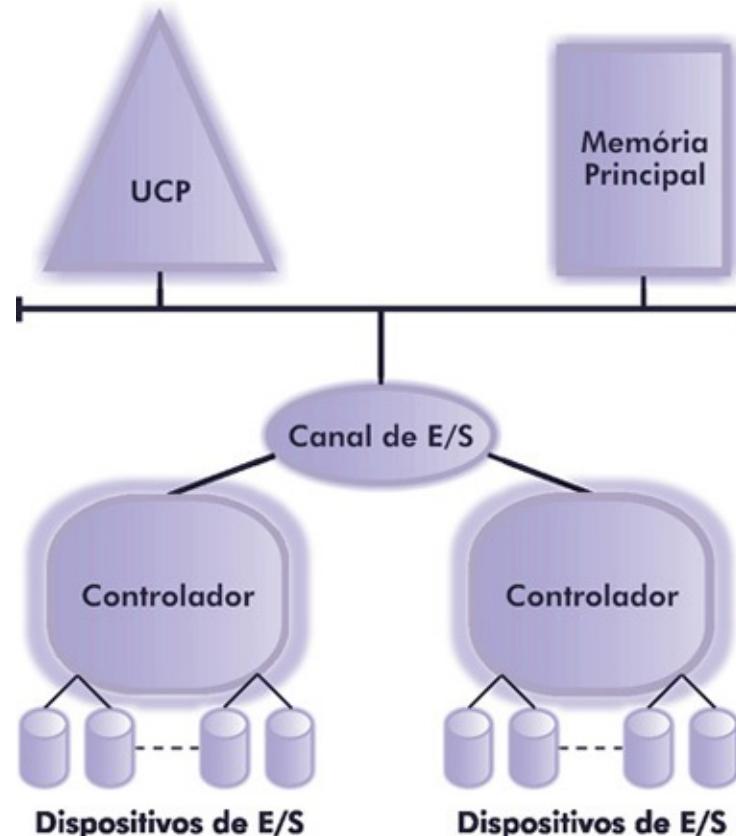


Fig. 3.4 Canal de E/S.

A evolução do canal permitiu que este possuísse sua própria memória, eliminando a necessidade de os programas de E/S serem carregados para a memória principal. Com essa nova arquitetura, várias funções de E/S puderam ser controladas com mínima intervenção da UCP. Este último estágio do canal é também denominado *processador de entrada e saída*, embora seja comum encontrarmos os dois termos empregados indistintamente (Stallings, 1997).

► 3.5 Buffering

A técnica de *buffering* consiste na utilização de uma área na memória principal, denominada buffer, para a transferência de dados entre os dispositivos de E/S e a memória. Esta técnica permite que em uma operação de leitura o dado seja transferido primeiramente para o buffer, liberando imediatamente o dispositivo de entrada para realizar uma nova leitura. Nesse caso, enquanto o processador manipula o dado localizado no buffer, o dispositivo realiza outra operação de leitura no mesmo instante. Este mesmo mecanismo pode ser aplicado nas operações de gravação (Fig. 3.5).

O buffering permite minimizar o problema da disparidade da velocidade de processamento existente entre o processador e os dispositivos de E/S. O objetivo principal desta técnica é manter, na maior parte

do tempo, processador e dispositivos de E/S ocupados.

A unidade de transferência usada no mecanismo de buffering é o *registro*. O tamanho do registro pode ser especificado em função da natureza do dispositivo, como uma linha gerada por uma impressora ou um caractere de um teclado, ou da aplicação, como um registro lógico definido em um arquivo.

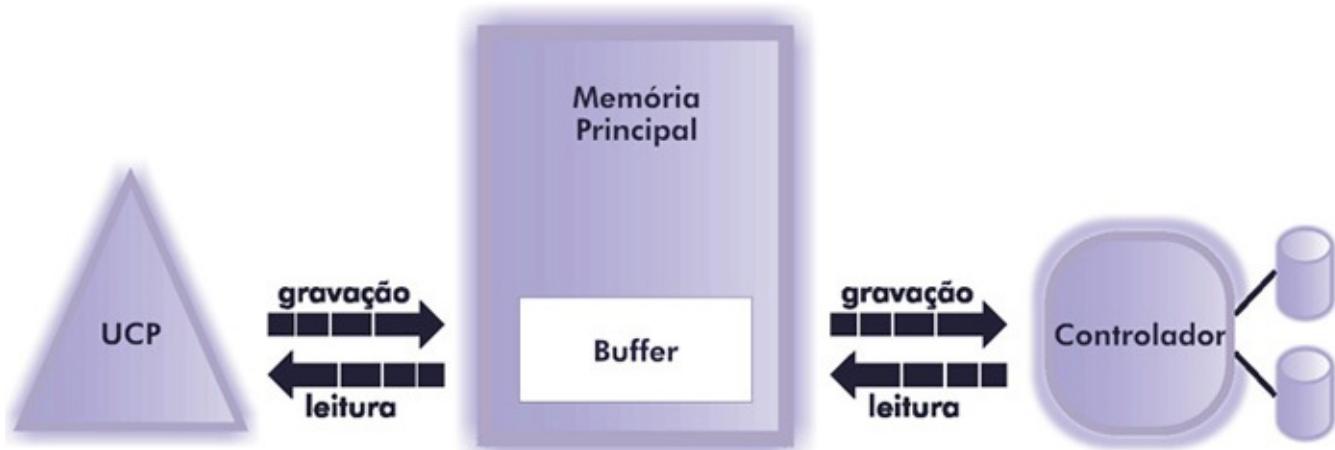


Fig. 3.5 Operações de E/S utilizando buffer.

O buffer deve permitir armazenar diversos registros, de forma que existam dados lidos, mas ainda não processados (operação de leitura), ou processados, mas ainda não gravados (operação de gravação). Desta forma, o dispositivo de entrada poderá ler diversos registros antes que o processador manipule os dados, ou o processador poderá manipular diversos registros antes de o dispositivo de saída realizar a gravação. Isso é extremamente eficiente, pois, dessa maneira, é possível compatibilizar a diferença existente entre o tempo em que o processador executa instruções e o tempo em que o dispositivo de E/S realiza as operações de leitura e gravação.

► 3.6 Spooling

A técnica de *spooling* (simultaneous peripheral operation on-line) foi introduzida no final dos anos 1950 para aumentar o grau de concorrência e a eficiência dos sistemas operacionais. Naquela época, os programas dos usuários eram submetidos um a um ao processamento pelo operador. Como a velocidade de operação dos dispositivos de E/S é muito menor que a do processador, era comum que a UCP permanecesse ociosa à espera de programas e dados de entrada ou pelo término de uma impressão.

A solução foi armazenar os vários programas e seus dados, também chamados de *jobs*, em uma fita magnética e, em seguida, submetê-los a processamento. Dessa forma, o processador poderia executar sequencialmente cada job, diminuindo o tempo de processamento e transição entre eles. Da mesma forma, em vez de um job gravar suas saídas diretamente na impressora, poderia direcioná-las para uma outra fita, que depois seria impressa integralmente. Esta forma de processamento é chamada de *spooling*, e foi a base dos sistemas batch.

A utilização de fitas magnéticas obrigava o processamento a ser estritamente sequencial, ou seja, o primeiro job a ser gravado na fita era o primeiro a ser processado. Assim, se um job que levasse várias

horas antecedesse pequenos jobs, seus tempos de resposta ficariam seriamente comprometidos. Com o surgimento de dispositivos de acesso direto, como discos, foi possível tornar o spooling mais eficiente e, principalmente, possibilitar o processamento não sequencial dos jobs.

A técnica de spooling, semelhante à técnica de buffering já apresentada, utiliza uma área em disco como se fosse um grande buffer. Neste caso, dados podem ser lidos ou gravados em disco, enquanto programas são executados concorrentemente.

Atualmente essa técnica está presente na maioria dos sistemas operacionais, sendo utilizada no gerenciamento de impressão. No momento em que um comando de impressão é executado, as informações que serão impressas são gravadas antes em um arquivo em disco, conhecido como *arquivo de spool*, liberando imediatamente o programa para outras atividades. Posteriormente, o sistema operacional encarrega-se de direcionar o conteúdo do arquivo de spool para a impressora (Fig. 3.6).

O uso do spooling permite desvincular o programa do dispositivo de impressão, impedindo que um programa reserve a impressora para uso exclusivo. O sistema operacional é o responsável por gerenciar a sequência de impressões solicitadas pelos programas, seguindo critérios que garantam a segurança e o uso eficiente das impressoras.

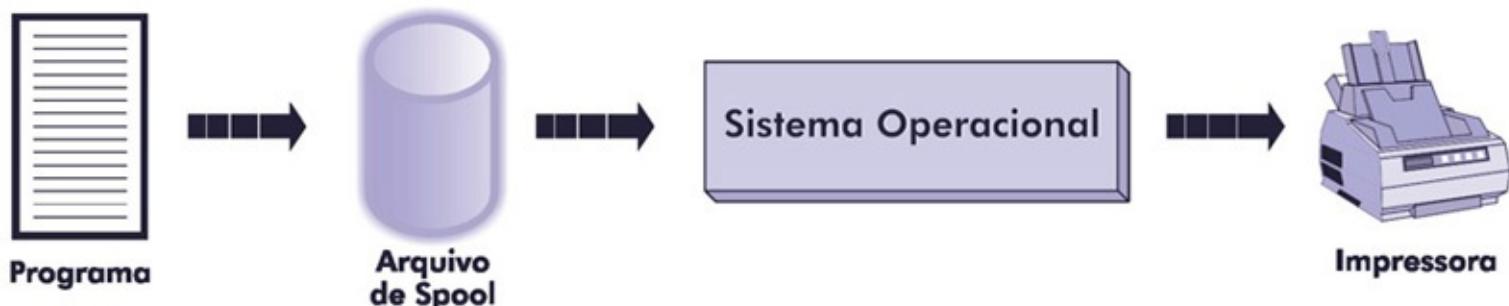


Fig. 3.6 Técnica de spooling.

► 3.7 Reentrância

É comum, em sistemas multiprogramáveis, vários usuários utilizarem os mesmos aplicativos simultaneamente, como editores de textos e compiladores. Se cada usuário que utilizasse um desses aplicativos trouxesse o código executável para a memória, haveria diversas cópias de um mesmo programa na memória principal, o que ocasionaria um desperdício de espaço.

Reentrância é a capacidade de um código executável (*código reentrante*) ser compartilhado por diversos usuários, exigindo que apenas uma cópia do programa esteja na memória. A reentrância permite que cada usuário possa estar em um ponto diferente do código reentrante, manipulando dados próprios, exclusivos de cada usuário (Fig. 3.7).

Os utilitários do sistema, como editores de texto, compiladores e linkers, são exemplos de código reentrante que proporcionam uma utilização mais eficiente da memória principal e aumento no desempenho do sistema. Em alguns sistemas operacionais existe a possibilidade de implementar o conceito de reentrância em aplicativos desenvolvidos pelos próprios usuários.

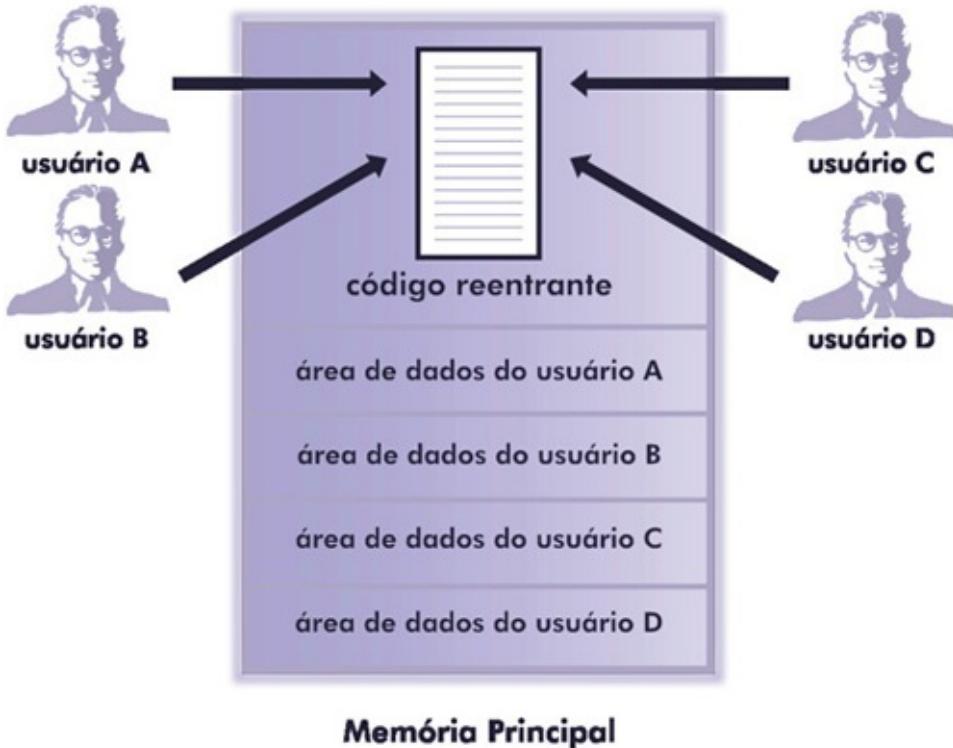


Fig. 3.7 Reentrância.

► 3.8 Exercícios

1. O que é concorrência e como este conceito está presente nos sistemas operacionais multiprogramáveis?
2. Por que o mecanismo de interrupção é fundamental para a implementação da multiprogramação?
3. Explique o mecanismo de funcionamento das interrupções.
4. O que são eventos síncronos e assíncronos? Como estes eventos estão relacionados ao mecanismo de interrupção e exceção?
5. Dê exemplos de eventos associados ao mecanismo de exceção.
6. Qual a vantagem da E/S controlada por interrupção comparada com a técnica de spooling?
7. O que é DMA e qual a vantagem desta técnica?
8. Como a técnica de buffering permite aumentar a concorrência em um sistema computacional?
9. Explique o mecanismo de spooling de impressão.
10. Em um sistema multiprogramável, seus usuários utilizam o mesmo editor de textos (200 Kb), compilador (300 Kb), software de correio eletrônico (200 Kb) e uma aplicação corporativa (500 Kb). Caso o sistema não implemente reentrância, qual o espaço de memória principal ocupado pelos programas quando 10 usuários estiverem utilizando todas as aplicações simultaneamente? Qual o espaço liberado quando o sistema implementa reentrância em todas as aplicações?



Estrutura do Sistema Operacional

► 4.1 Introdução

O sistema operacional é formado por um conjunto de rotinas que oferece serviços aos usuários e às suas aplicações. Esse conjunto de rotinas é denominado *núcleo do sistema*, ou *kernel*. A maioria dos sistemas operacionais é fornecida acompanhada de utilitários e linguagem de comandos, que são ferramentas de apoio ao usuário, porém não são parte do núcleo do sistema. Na Fig. 4.1 é apresentado um modelo de camadas detalhando a estrutura do sistema operacional e suas interfaces.

Há três maneiras distintas de os usuários se comunicarem com o kernel do sistema operacional. Uma delas é por intermédio das chamadas rotinas do sistema realizadas por aplicações. Além disso, os usuários podem interagir com o núcleo mais amigavelmente por meio de utilitários ou linguagem de comandos. Cada sistema operacional oferece seus próprios utilitários, como compiladores e editores de texto. A linguagem de comandos também é particular de cada sistema, com estruturas e sintaxe próprias.

A estrutura do núcleo, ou seja, a maneira como o código do sistema é organizado e o inter-relacionamento entre seus diversos componentes, pode variar conforme a concepção do projeto. Existem diversas abordagens em relação à estrutura de sistemas operacionais que serão apresentadas na parte final deste capítulo.

Inicialmente serão apresentadas as funções do núcleo e os conceitos relativos à segurança, proteção do sistema, modos de acesso, rotinas do sistema, system calls, linguagem de comandos e ativação/desativação do sistema.

► 4.2 Funções do Núcleo

A compreensão da estrutura e do funcionamento de um sistema operacional não é simples. Diferentemente de uma aplicação convencional, com sequenciamento de início, meio e fim, as rotinas do sistema são executadas concorrentemente sem uma ordem predefinida, com base em eventos dissociados do tempo

(eventos assíncronos). Muitos desses eventos estão relacionados ao hardware e a tarefas internas do próprio sistema operacional.

As principais funções do núcleo encontradas nos sistemas operacionais estão listadas a seguir e, no decorrer do livro, serão abordadas com detalhes:

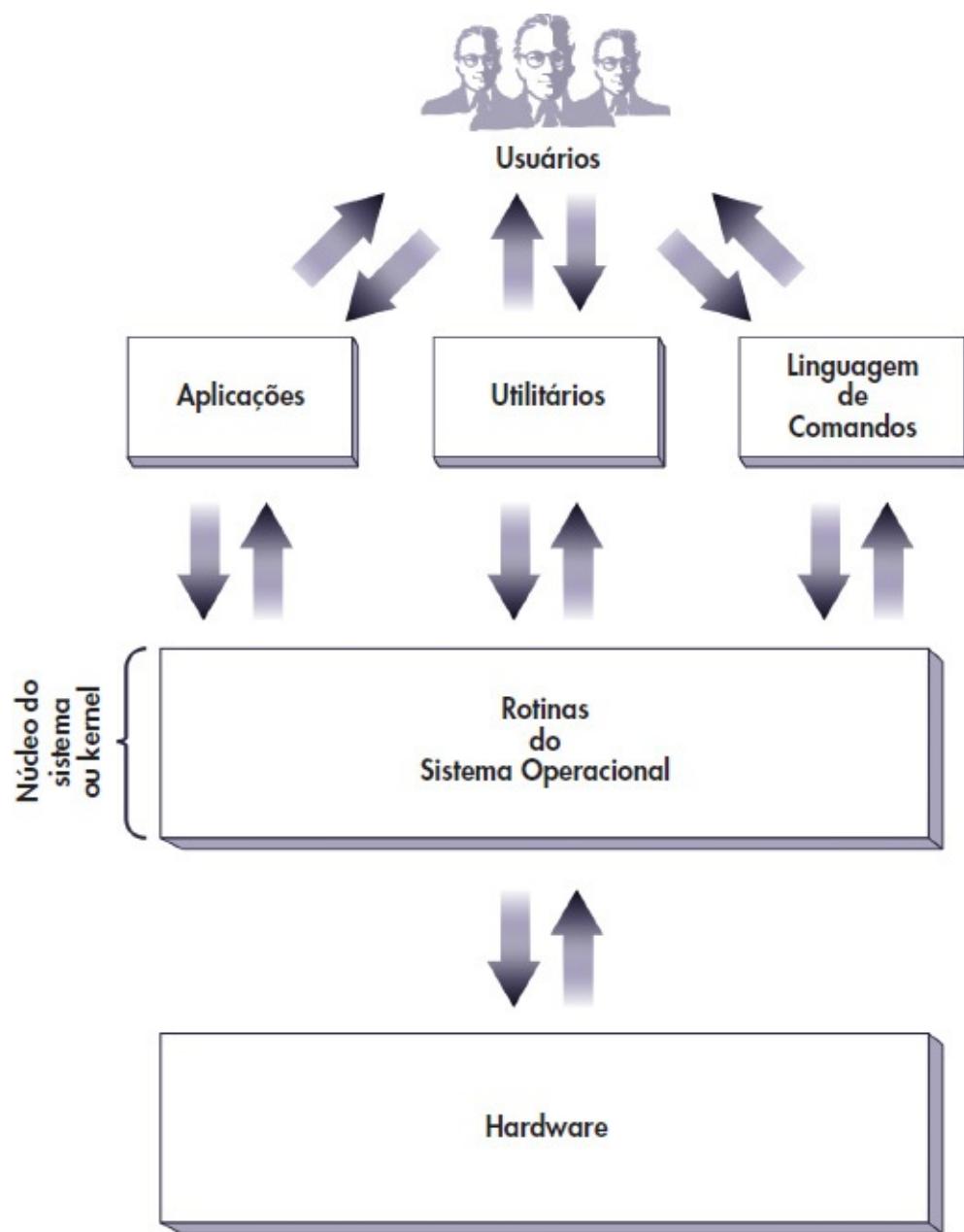


Fig. 4.1 Estrutura do sistema operacional.

- tratamento de interrupções e exceções;
- criação e eliminação de processos e threads;
- sincronização e comunicação entre processos e threads;
- escalonamento e controle dos processos e threads;
- gerência de memória;
- gerência do sistema de arquivos;
- gerência de dispositivos de E/S;

- suporte a redes locais e distribuídas;
- contabilização do uso do sistema;
- auditoria e segurança do sistema.

Como decorrência da complexidade da arquitetura de um sistema multiprogramável, é natural que surjam problemas relativos à segurança no inter-relacionamento dos diversos subsistemas existentes. Como exemplo, podemos considerar a situação em que diversos usuários compartilham os mesmos recursos, como memória, processador e dispositivos de E/S. Esta situação exige que o sistema operacional garanta a confiabilidade na execução concorrente de todos os programas e nos dados dos usuários, além da garantia da integridade do próprio sistema operacional.

Uma das principais características da multiprogramação é permitir que vários programas compartilhem o processador. O sistema operacional deve ser responsável pelo controle da utilização da UCP, de forma a impedir que algum programa monopolize o seu uso inadequadamente.

Como vários programas ocupam a memória simultaneamente, cada usuário deve possuir uma área reservada onde seus dados e código são armazenados. O sistema operacional implementa mecanismos de proteção, de forma a preservar estas informações de maneira reservada. Caso um programa tente acessar uma posição de memória fora de sua área ocorrerá um erro indicando a violação de acesso. Para que diferentes programas tenham o direito de compartilhar uma mesma área de memória, o sistema operacional deve oferecer mecanismos para que a comunicação seja feita de forma sincronizada e controlada, evitando, desta forma, problemas de inconsistência.

De modo semelhante ao compartilhamento de memória, um disco pode armazenar arquivos de diferentes usuários. Mais uma vez o sistema operacional deve garantir a integridade e a confidencialidade dos dados, permitindo ainda que dois ou mais usuários possam ter acesso simultâneo ao mesmo arquivo.

Para solucionar esses diversos problemas originados pelo ambiente multiprogramável, o sistema operacional deve implementar mecanismos de proteção que controlem o acesso concorrente aos diversos recursos do sistema.

► 4.3 Modo de Acesso

Uma preocupação que surge nos projetos de sistemas operacionais é a implementação de mecanismos de proteção ao núcleo do sistema e de acesso aos seus serviços. Caso uma aplicação, que tenha acesso ao núcleo, realize uma operação que altere sua integridade, todo o sistema poderá ficar comprometido e inoperante. Muitas das principais implementações de segurança de um sistema operacional utilizam um mecanismo presente no hardware dos processadores, conhecido como *modo de acesso*.

Em geral, os processadores possuem dois modos de acesso: *modo usuário* e *modo kernel*. Quando o processador trabalha no modo usuário, uma aplicação só pode executar instruções conhecidas como não privilegiadas, tendo acesso a um número reduzido de instruções, enquanto no modo kernel a aplicação pode ter acesso ao conjunto total de instruções do processador. O modo de acesso é determinado por um conjunto de bits, localizado no registrador de status do processador, que indica o modo de acesso corrente. Por intermédio desse registrador, o hardware verifica se a instrução pode ou não ser executada.

As *instruções privilegiadas* não devem ser utilizadas de maneira indiscriminada pelas aplicações, pois isso poderia ocasionar sérios problemas à integridade do sistema. Suponha que uma aplicação atualize um arquivo em disco. O programa, por si só, não deve especificar diretamente as instruções que acessam seus dados no disco. Como o disco é um recurso compartilhado, sua utilização deverá ser gerenciada unicamente pelo sistema operacional, evitando que a aplicação possa ter acesso a qualquer área do disco sem autorização, o que poderia comprometer a segurança e a integridade do sistema de arquivos. Como visto, fica claro que existem certas instruções que só devem ser executadas pelo sistema operacional ou sob sua supervisão, impedindo, assim, a ocorrência de problemas de segurança e integridade do sistema. As instruções privilegiadas só podem ser executadas quando o modo de acesso do processador encontra-se em kernel, caso contrário o hardware irá impedir a execução da instrução. As *instruções não privilegiadas* são as que não oferecem risco ao sistema e podem ser executadas em modo não privilegiado, ou seja, modo usuário.

Outro exemplo do uso dos modos de acessos é a proteção do próprio núcleo do sistema residente na memória principal. Caso uma aplicação tenha acesso a áreas de memória onde está carregado o sistema operacional, um programador mal-intencionado ou um erro de programação poderia gravar nesta área, violando o sistema. Com o mecanismo de modo de acesso, para uma aplicação escrever numa área onde resida o sistema operacional o programa deve estar sendo executado no modo kernel.

► 4.4 Rotinas do Sistema Operacional e System Calls

As *rotinas do sistema operacional* compõem o núcleo do sistema, oferecendo serviços aos usuários e suas aplicações. Todas as funções do núcleo são implementadas por rotinas do sistema que necessariamente possuem em seu código instruções privilegiadas. A partir desta condição, para que estas rotinas possam ser executadas o processador deve estar obrigatoriamente em modo kernel, o que exige a implementação de mecanismos de proteção para garantir a confiabilidade do sistema.

Todo o controle de execução de rotinas do sistema operacional é realizado pelo mecanismo conhecido como *system call*. Toda vez que uma aplicação desejar chamar uma rotina do sistema operacional, o mecanismo de system call é ativado. Inicialmente, o sistema operacional verificará se a aplicação possui privilégios necessários para executar a rotina desejada. Em caso negativo, o sistema operacional impedirá o desvio para a rotina do sistema, sinalizando ao programa chamador que a operação não é possível. Este é um *mecanismo de proteção por software*, no qual o sistema operacional garante que as aplicações só poderão executar rotinas do sistema que estão previamente autorizadas. Esta autorização prévia é realizada pelo administrador do sistema, e será vista com mais detalhes no Cap. 5 — Processo.

Considerando que a aplicação possua o devido privilégio para chamar a rotina do sistema desejada, o sistema operacional primeiramente salva o conteúdo corrente dos registradores, troca o modo de acesso do processador de usuário para kernel e realiza o desvio para a rotina alterando o registrador PC com o endereço da rotina chamada (Fig. 4.2). Ao término da execução da rotina do sistema, o modo de acesso é alterado de kernel para usuário e o contexto dos registradores restaurados para que a aplicação continue a execução a partir da instrução que chamou a rotina do sistema.

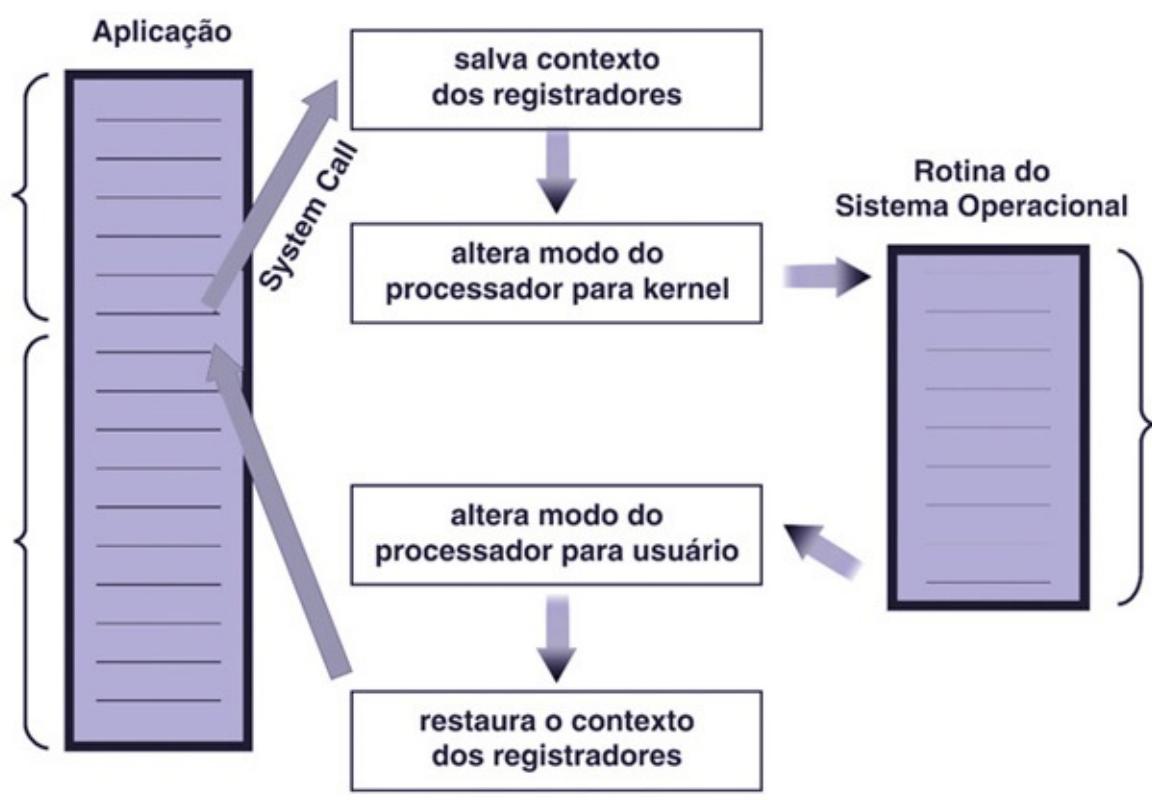


Fig. 4.2 Chamada a uma rotina do sistema (a).

Uma aplicação sempre deve executar com o processador em modo usuário. Caso uma aplicação tente executar diretamente uma instrução privilegiada sem ser por intermédio de uma chamada à rotina do sistema, um *mecanismo de proteção por hardware* garantirá a segurança do sistema, impedindo a operação. Nesta situação, em que a aplicação tenta executar uma instrução privilegiada em modo usuário e sem a supervisão do sistema operacional, o próprio hardware do processador sinalizará um erro. Uma exceção é gerada e a execução do programa é interrompida, protegendo desta forma o núcleo do sistema.

Utilizando o mesmo problema apresentado anteriormente do acesso ao disco, para o programa atualizar um arquivo, a aplicação deve solicitar a operação de E/S ao sistema operacional por meio de uma chamada a uma rotina do sistema. O mecanismo de system call verificará se a aplicação possui privilégio para a operação e, em caso afirmativo, irá alterar o modo de acesso do processador de usuário para kernel. Após executar a rotina de leitura, o modo de acesso volta ao estado usuário para continuar a execução do programa.

Os mecanismos de system call e de proteção por hardware garantem a segurança e a integridade do sistema. Com isso, as aplicações estão impedidas de executarem instruções privilegiadas sem a autorização e a supervisão do sistema operacional.

► 4.5 Chamada a Rotinas do Sistema Operacional

As rotinas do sistema e o mecanismo de system call podem ser entendidos como uma porta de entrada para o núcleo do sistema operacional e a seus serviços. Sempre que uma aplicação desejar algum serviço do sistema, deve ser realizada uma chamada a uma de suas rotinas através de uma system call (Fig. 4.3). Por intermédio dos parâmetros fornecidos na system call, a solicitação é processada e uma resposta é retornada à aplicação juntamente com um estado de conclusão indicando se houve algum erro. O

mechanismo de ativação e comunicação entre o programa e o sistema operacional é semelhante ao mecanismo implementado quando um programa chama uma sub-rotina.

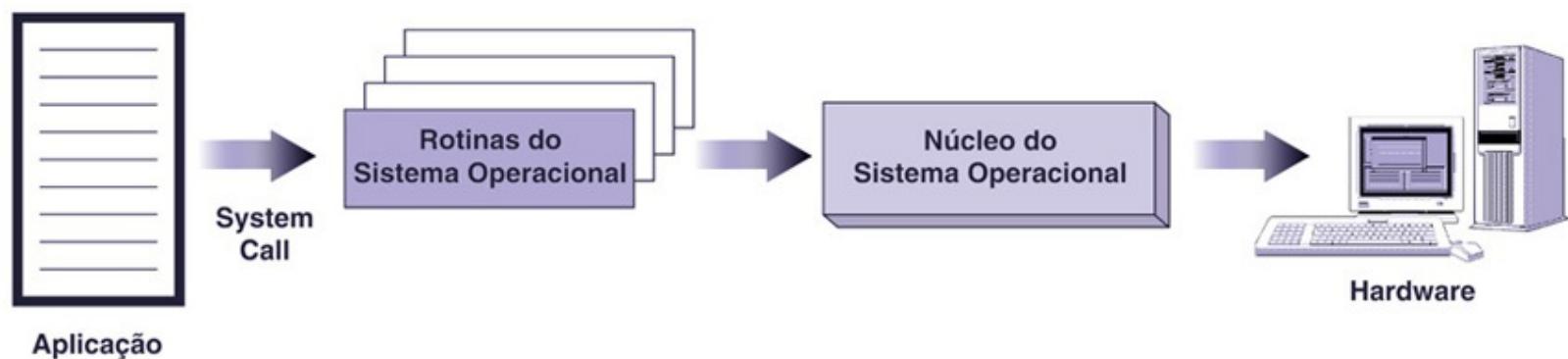


Fig. 4.3 Chamada a uma rotina do sistema (b).

O termo system call é tipicamente utilizado em sistemas Unix, porém em outros sistemas o mesmo conceito é apresentado com diferentes nomes, como system services, no OpenVMS, e Application Program Interface (API), no MS Windows. No exemplo a seguir, desenvolvido em Delphi, é utilizada a API GetSystemTime para obter a data e a hora do sistema MS Windows. A função SystemTimeToDateTime converte a data e a hora para o formato DataHoraT do Delphi e, em seguida, para o formato texto utilizando a função DateTimeToStr. A última linha exibe a data e a hora do sistema em uma janela previamente criada.

```
GetSystemTime(SystemTime);  
DataHoraT := SystemTimeToDateTime(SystemTime);  
DataHoraS := DateTimeToStr(DataHoraT);  
RichEdit1.Lines.Add(DataHoraS);
```

A maioria dos programadores e usuários desconhece os detalhes envolvidos, por exemplo, em um simples comando de leitura a um arquivo utilizando uma linguagem de alto nível. De forma simplificada, o comando da linguagem de alto nível é convertido pelo compilador para uma chamada a uma rotina específica que, quando executada, verifica a ocorrência de erros e retorna os dados ao programa de forma transparente ao usuário. As rotinas do sistema podem ser divididas por grupos de função (Tabela 4.1).

Na realidade existem duas maneiras distintas de chamada a uma rotina do sistema operacional: *explícita* e *implícita*. A *chamada explícita* é a descrita anteriormente, em que no código do programa há uma função explicitando a chamada a rotina do sistema com passagem de parâmetro. Já a *chamada implícita* é realizada por intermédio de um comando da própria linguagem de programação. Quando este comando é traduzido para uma instrução de mais baixo nível, há uma conversão do comando em uma chamada da função.

Tabela 4.1 Funções das system calls

Gerência de processos e threads	Criação e eliminação de processos e threads Alteração das características de processos e threads Sincronização e comunicação entre processos e threads Obtenção de informações sobre processos e threads
Gerência de memória	Alocação e desalocação de memória
Gerência do sistema de arquivos	Criação e eliminação de arquivos e diretórios Alteração das características de arquivos e diretórios Abertura e fechamento de arquivos Leitura e gravação em arquivos Obtenção de informações sobre arquivos e diretórios
Gerência de dispositivos	Alocação e desalocação de dispositivos Operações de entrada/saída em dispositivos Obtenção de informações sobre dispositivos

Cada sistema operacional possui seu próprio conjunto de rotinas, com nomes, parâmetros e formas de ativação específicos. Consequentemente, uma aplicação desenvolvida utilizando serviços de um determinado sistema operacional não pode ser portada diretamente para um outro sistema, exigindo algumas correções no código-fonte. Uma tentativa de criar uma biblioteca de chamadas objetivando uma padronização foi elaborada pelos institutos ISO e IEEE, resultando um conjunto conhecido como POSIX (Portable Operating System Interface for Unix). Inicialmente voltado para a unificação das várias versões do sistema operacional Unix, o POSIX estabeleceu uma biblioteca-padrão, permitindo que uma aplicação desenvolvida seguindo este conjunto de chamadas possa ser portada para os demais sistemas. A maioria dos sistemas operacionais modernos oferece algum suporte ao padrão POSIX como o MS Windows, IBM-AIX, HP-UX e o SUN-Solaris.

► 4.6 Linguagem de Comandos

A *linguagem de comandos*, ou *linguagem de controle*, permite que o usuário se comunique de uma forma simples com o sistema operacional, capacitando-o a executar diversas tarefas específicas do sistema como criar, ler ou eliminar arquivos, consultar diretórios ou verificar a data e a hora armazenadas no sistema. Dessa forma, o usuário dispõe de uma interface direta com o sistema operacional. A Tabela 4.2 apresenta exemplos de comandos disponíveis no sistema Microsoft Windows. Cada sistema operacional possui a sua linguagem de comandos como, por exemplo, a DCL (Digital Command Language) utilizada no OpenVMS, JCL (Job Control Language) no MVS da IBM e os comandos do shell disponíveis nos diversos sistemas Unix.

Tabela 4.2 Exemplos de comandos do MS Windows

Comando	Descrição
---------	-----------

dir	Lista o conteúdo de um diretório
cd	Altera o diretório default
type	Exibe o conteúdo de um arquivo
del	Elimina arquivos
mkdir	Cria um diretório
ver	Mostra a versão do Windows

Cada comando, depois de digitado pelo usuário, é interpretado pelo *shell* ou *interpretador de comandos*, que verifica a sintaxe do comando, faz chamadas a rotinas do sistema e apresenta um resultado ou uma mensagem informativa (Fig. 4.4a). Em geral, o interpretador de comandos não faz parte do núcleo do sistema operacional, possibilitando maior flexibilidade na criação de diferentes linguagens de controle para um mesmo sistema. Por exemplo, o Unix oferece, basicamente, três interpretadores de comandos: Bourne Shell, C Shell e Korn Shell.

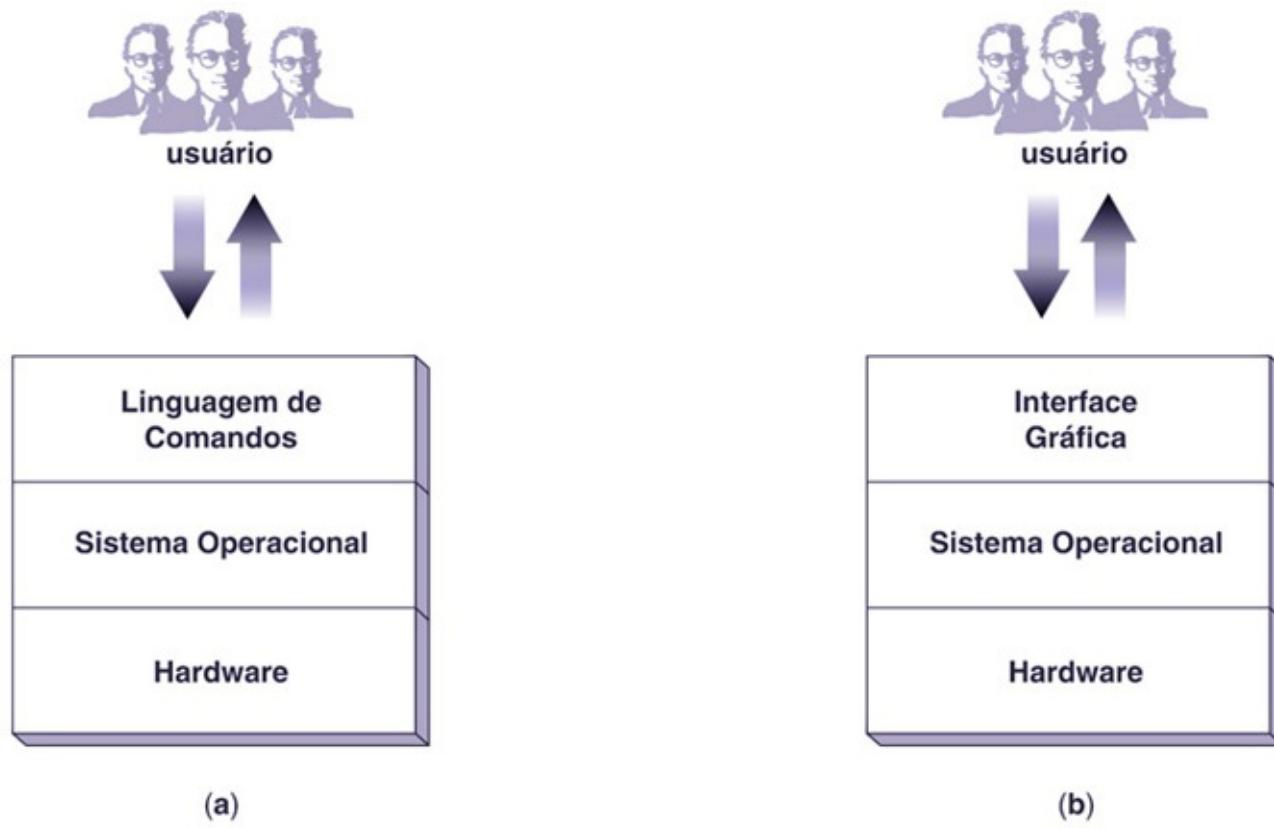


Fig. 4.4 Interface do usuário com o sistema operacional.

Na maioria dos sistemas operacionais, as linguagens de comandos evoluíram no sentido de permitir a interação mais amigável com os usuários, utilizando interfaces gráficas como janelas e ícones, a exemplo dos sistemas MS Windows. Na maioria dos casos, a interface gráfica é apenas mais um nível de abstração entre o usuário e os serviços do sistema operacional (Fig. 4.4b).

As linguagens de comandos são poderosas a ponto de oferecer a possibilidade de criar programas com

estruturas de decisão e iteração. Esses programas nada mais são do que uma sequência de comandos armazenados em um arquivo texto, denominados *arquivos de comandos*, *arquivos batch* ou *shell scripts*, que podem ser executados sempre que necessário. Uma das principais vantagens no uso de arquivos de comandos é possibilitar a automatização de diversas tarefas ligadas à gerência do sistema.

► 4.7 Ativação/Desativação do Sistema

Inicialmente, quando um computador é ligado não há sistema operacional carregado na memória da máquina. Em geral, o sistema operacional reside em um disco rígido, podendo também estar armazenado em outros dispositivos de memória secundária, como CD ou DVD. Os componentes do sistema operacional devem ser carregados para a memória principal toda vez que o computador é ligado por intermédio de um procedimento denominado *ativação do sistema* ou *boot*.

O procedimento de ativação se inicia com a execução de um programa chamado *boot loader*, que se localiza em um endereço fixo de uma memória ROM da máquina. Este programa chama a execução de outro programa conhecido como POST (Power-On Self Test), que identifica possíveis problemas de hardware no equipamento. Após esta fase, o procedimento de ativação verifica se há no sistema computacional algum dispositivo de armazenamento onde haja um sistema operacional residente. Caso nenhum dispositivo seja encontrado, uma mensagem de erro é apresentada e o procedimento de ativação é interrompido. Se um dispositivo com o sistema operacional é encontrado, um conjunto de instruções é carregado para memória e localizado em um bloco específico do dispositivo conhecido como *setor de boot* (*boot sector*). A partir da execução deste código, o sistema operacional é finalmente carregado para a memória principal. Além da carga, a ativação do sistema também consiste na execução de arquivos de inicialização onde são especificados procedimentos de customização e configuração de hardware e software específicos para cada ambiente (Fig. 4.5).

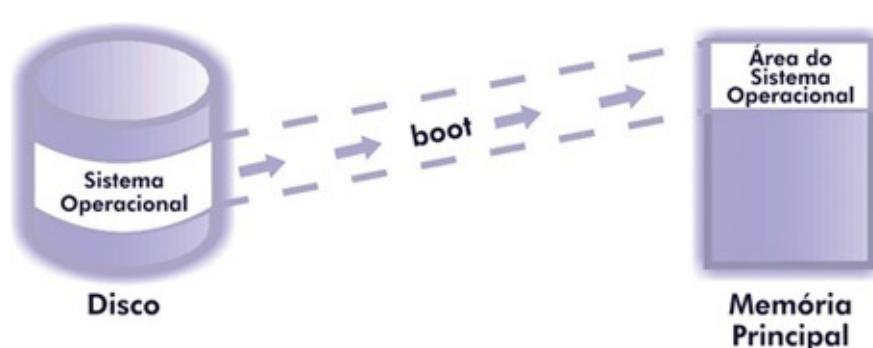


Fig. 4.5 Ativação do sistema.

Na maioria dos sistemas também existe o processo de *desativação* ou *shutdown*. Este procedimento permite que as aplicações e componentes do sistema operacional sejam desativados ordenadamente, garantindo, desta forma, sua integridade.

► 4.8 Arquiteturas do Núcleo

O projeto de um sistema operacional é bastante complexo e deve atender a diversos requisitos, algumas

vezes conflitantes, como confiabilidade, portabilidade, fácil manutenção, flexibilidade e desempenho. O projeto do sistema irá depender muito da arquitetura do hardware a ser utilizado e do tipo de sistema que se deseja construir: batch, tempo compartilhado, monousuário ou multiusuário, tempo real etc.

Os primeiros sistemas operacionais foram desenvolvidos integralmente em assembly, e o código possuía cerca de um milhão de instruções (IBM OS/360). Com a evolução dos sistemas e o aumento do número de linhas de código para algo perto de 20 milhões (MULTICS), técnicas de programação modular foram incorporadas ao projeto, além de linguagens de alto nível, como PL/I e Algol. Nos sistemas operacionais atuais, o número de linhas de código pode chegar a mais de 40 milhões (Windows 2000), sendo que grande parte do código é escrita em Linguagem C/C++.

Além de facilitar o desenvolvimento e a manutenção do sistema, a utilização de linguagens de alto nível permite uma maior portabilidade, ou seja, que o sistema operacional seja facilmente alterado em outra arquitetura de hardware. Uma desvantagem do uso de linguagens de alto nível em relação à programação assembly é a perda de desempenho. Por isso, as partes críticas do sistema, como os device drivers, o escalonador e as rotinas de tratamento de interrupções, são desenvolvidas em assembly.

Uma tendência no projeto de sistemas operacionais modernos é a utilização de técnicas de orientação por objetos, o que leva para o projeto do núcleo do sistema todas as vantagens deste modelo de desenvolvimento de software. Existe uma série de vantagens na utilização de programação por objetos no projeto e na implementação de sistemas operacionais. A seguir, os principais benefícios são apresentados:

- melhoria na organização das funções e recursos do sistema;
- redução no tempo de desenvolvimento;
- maior facilidade na manutenção e extensão do sistema;
- facilidade de implementação do modelo de computação distribuída.

A estrutura do núcleo do sistema operacional, ou seja, a maneira como o código do sistema é organizado e o inter-relacionamento de seus diversos componentes, pode variar conforme a concepção do projeto. A seguir serão abordadas as principais arquiteturas dos sistemas operacionais: arquitetura monolítica, arquitetura de camadas, máquina virtual e arquitetura microkernel.

4.8.1 Arquitetura Monolítica

A *arquitetura monolítica* pode ser comparada com uma aplicação formada por vários módulos que são compilados separadamente e depois linkados, formando um grande e único programa executável, onde os módulos podem interagir livremente (Fig. 4.6).

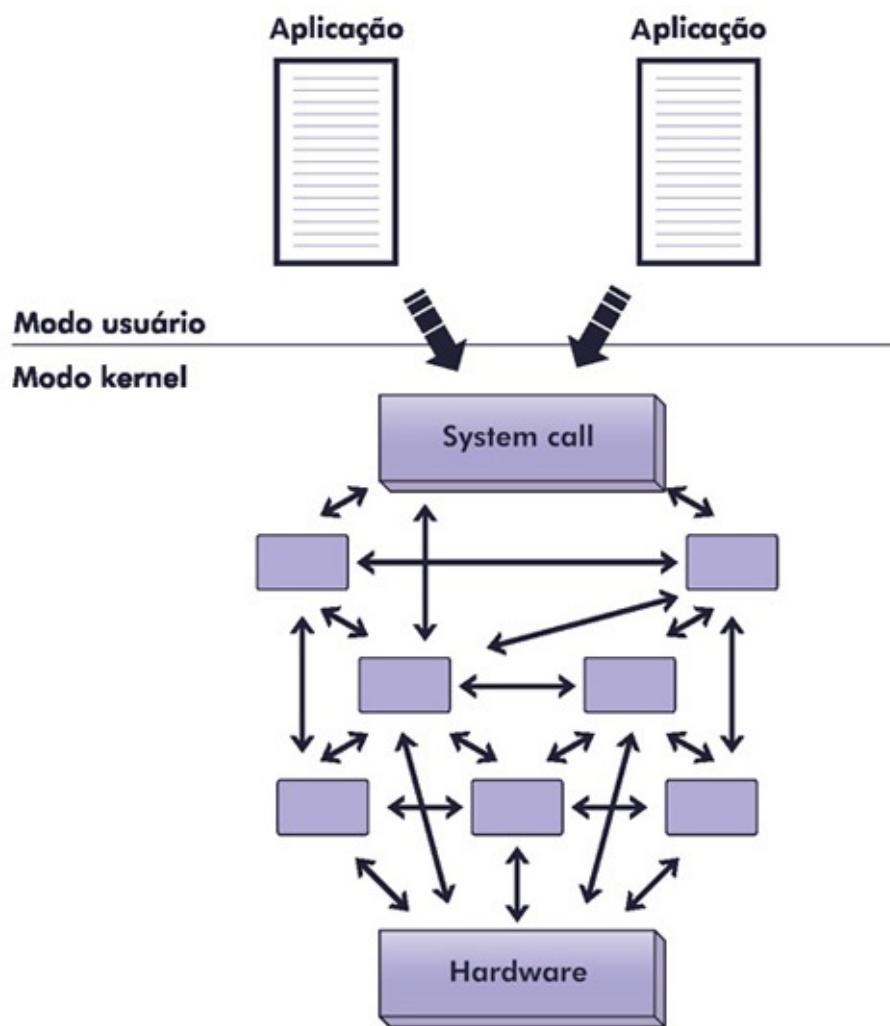


Fig. 4.6 Arquitetura monolítica.

Os primeiros sistemas operacionais foram desenvolvidos com base neste modelo, o que tornava seu desenvolvimento e, principalmente, sua manutenção bastante difíceis. Devido a sua simplicidade e bom desempenho, a estrutura monolítica foi adotada no projeto do MS-DOS e nos primeiros sistemas Unix.

4.8.2 Arquitetura de Camadas

Com o aumento da complexidade e do tamanho do código dos sistemas operacionais, técnicas de programação estruturada e modular foram incorporadas ao seu projeto. Na *arquitetura de camadas*, o sistema é dividido em níveis sobrepostos. Cada camada oferece um conjunto de funções que podem ser utilizadas apenas pelas camadas superiores.

O primeiro sistema com base nesta abordagem foi o sistema THE (Technische Hogeschool Eindhoven), construído por Dijkstra na Holanda em 1968 e que utilizava seis camadas. Posteriormente, os sistemas MULTICS e OpenVMS também implementaram o conceito de camadas, sendo estas concêntricas (Fig. 4.7). Neste tipo de implementação, as camadas mais internas são mais privilegiadas que as mais externas.

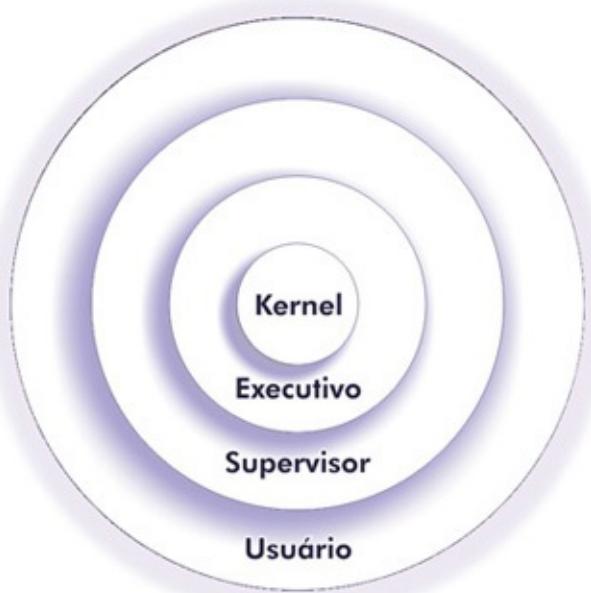


Fig. 4.7 Arquitetura em camadas do OpenVMS.

A vantagem da estruturação em camadas é isolar as funções do sistema operacional, facilitando sua manutenção e depuração, além de criar uma hierarquia de níveis de modos de acesso, protegendo as camadas mais internas. Uma desvantagem para o modelo de camadas é o desempenho. Cada nova camada implica uma mudança no modo de acesso. Por exemplo, no caso do OpenVMS, para se ter acesso aos serviços oferecidos pelo kernel é preciso passar por três camadas ou três mudanças no modo de acesso.

Atualmente, a maioria dos sistemas comerciais utiliza o modelo de duas camadas, onde existem os modos de acesso usuário (não privilegiado) e kernel (privilegiado). A maioria das versões do Unix e o Windows da Microsoft está baseada neste modelo.

4.8.3 Máquina Virtual

Um sistema computacional é formado por níveis, onde a camada de nível mais baixo é o hardware. Acima desta camada encontramos o sistema operacional que oferece suporte para as aplicações, como visto na Fig. 4.1. O modelo de *máquina virtual*, ou *virtual machine* (VM), cria um nível intermediário entre o hardware e o sistema operacional, denominado gerência de máquinas virtuais (Fig. 4.8). Este nível cria diversas máquinas virtuais independentes, onde cada uma oferece uma cópia virtual do hardware, incluindo os modos de acesso, interrupções, dispositivos de E/S etc.

Como cada máquina virtual é independente das demais, é possível que cada VM tenha seu próprio sistema operacional e que seus usuários executem suas aplicações como se todo o computador estivesse dedicado a cada um deles. Na década de 1960, a IBM implementou este modelo no sistema VM/370, permitindo que aplicações batch, originadas de antigos sistemas OS/360, e aplicações de tempo compartilhado pudessem conviver na mesma máquina de forma transparente a seus usuários e aplicações.

Este modelo cria o isolamento total entre cada VM, oferecendo grande segurança para cada máquina virtual. Se, por exemplo, uma VM executar uma aplicação que comprometa o funcionamento do seu sistema operacional, as demais máquinas virtuais não sofrerão qualquer problema. Apesar do isolamento

das aplicações, as máquinas virtuais não estão livres de problemas, como, por exemplo, vírus e erros de software do sistema operacional ou aplicações da VM. Além dessas vantagens, existem diversas aplicações para a utilização de máquinas virtuais:

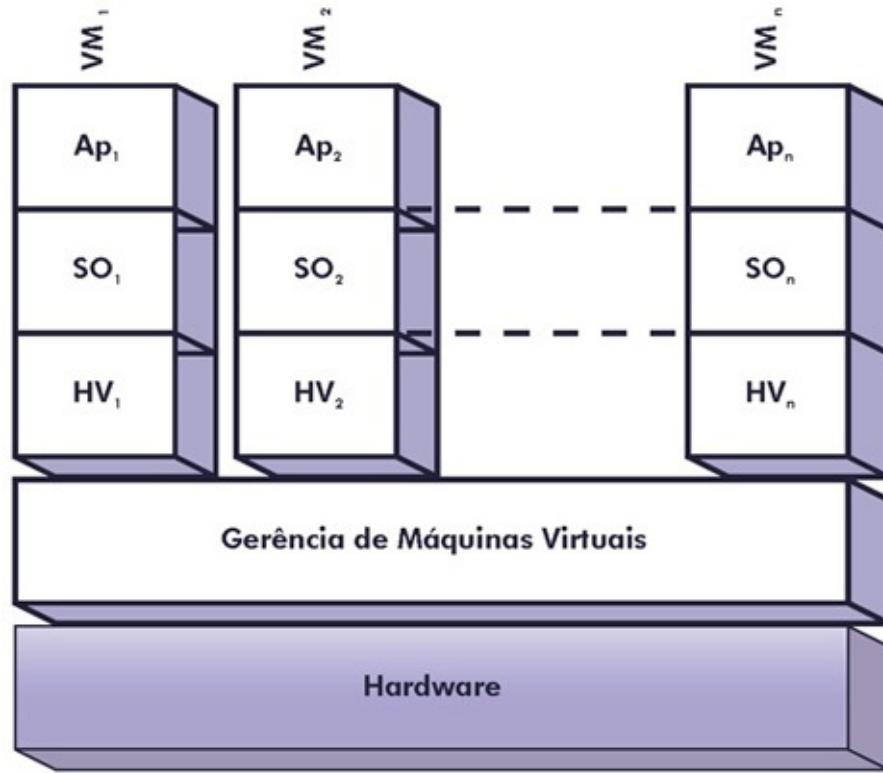


Fig. 4.8 Máquina virtual.

- Portabilidade de código: um programa executável, em linguagem de máquina, precisa de uma plataforma de hardware específica para ser executado, tornando-o dependente desse ambiente. Se o programa for gerado em uma linguagem intermediária e uma VM fizer a tradução dos comandos para a plataforma onde o programa está sendo executado, a aplicação pode ser portada para qualquer ambiente sem a necessidade de se reescrever o programa. A máquina virtual Java (Java Virtual Machine – JVM) funciona dessa forma. O compilador Java gera um código intermediário, chamado bytecode, que é interpretado pela JVM e convertido para o ambiente onde está sendo executado. Dessa forma, o programa é desenvolvido apenas uma vez e pode ser processado em qualquer hardware ou sistema operacional, bastando que exista uma JVM para tal.
- Consolidação de servidores: é muito comum que os servidores de uma empresa funcionem com apenas parte de sua capacidade total de processamento, ou seja, existe a subutilização da UCP, memória principal e dispositivos de E/S dos sistemas computacionais. A ideia da consolidação é poder executar diversas aplicações que estariam sendo processadas em diferentes servidores, cada qual com o seu hardware e sistema operacional, em uma única máquina, utilizando uma VM para cada uma das aplicações. Dessa forma, a consolidação permite a melhor utilização dos recursos computacionais e, além disso, também permite a redução de custos de infraestrutura, como energia elétrica, refrigeração, espaço físico e cabeamento de rede.
- Aumento da disponibilidade: uma vez que cada VM pode ser copiada facilmente para a memória secundária, como um disco, caso haja algum problema com uma VM ou com o próprio sistema onde está sendo processada, basta restaurar a cópia da VM em outro servidor e rapidamente o ambiente

pode ser novamente disponibilizado.

- Facilidade de escalabilidade e balanceamento de carga: caso o sistema no qual a VM está sendo processada fique sobrecarregado e apresente problemas de desempenho, a VM pode ser facilmente migrada para um novo ambiente que possua maior capacidade de processamento. Dessa forma, é possível balancear a carga dos servidores e garantir o bom desempenho das aplicações.
- Facilidade no desenvolvimento de software: as VMs permitem a criação de um ambiente independente para o desenvolvimento e teste de software sem o comprometimento das máquinas de produção. Além disso, é possível o teste do software em diferentes sistemas operacionais e suas versões sem a necessidade de um hardware dedicado.

4.8.4 Arquitetura Microkernel

Uma tendência nos sistemas operacionais modernos é tornar o núcleo do sistema operacional o menor e mais simples possível. Para implementar esta ideia, os serviços do sistema são disponibilizados através de processos, onde cada um é responsável por oferecer um conjunto específico de funções, como gerência de arquivos, gerência de processos, gerência de memória e escalonamento.

Sempre que uma aplicação deseja algum serviço, é realizada uma solicitação ao processo responsável. Neste caso, a aplicação que solicita o serviço é chamada de *cliente*, enquanto o processo que responde à solicitação é chamado de *servidor*. Um cliente, que pode ser uma aplicação de um usuário ou um outro componente do sistema operacional, solicita um serviço enviando uma mensagem para o servidor. O servidor responde ao cliente através de uma outra mensagem. A principal função do núcleo é realizar a comunicação, ou seja, a troca de mensagens entre cliente e servidor (Fig. 4.9).

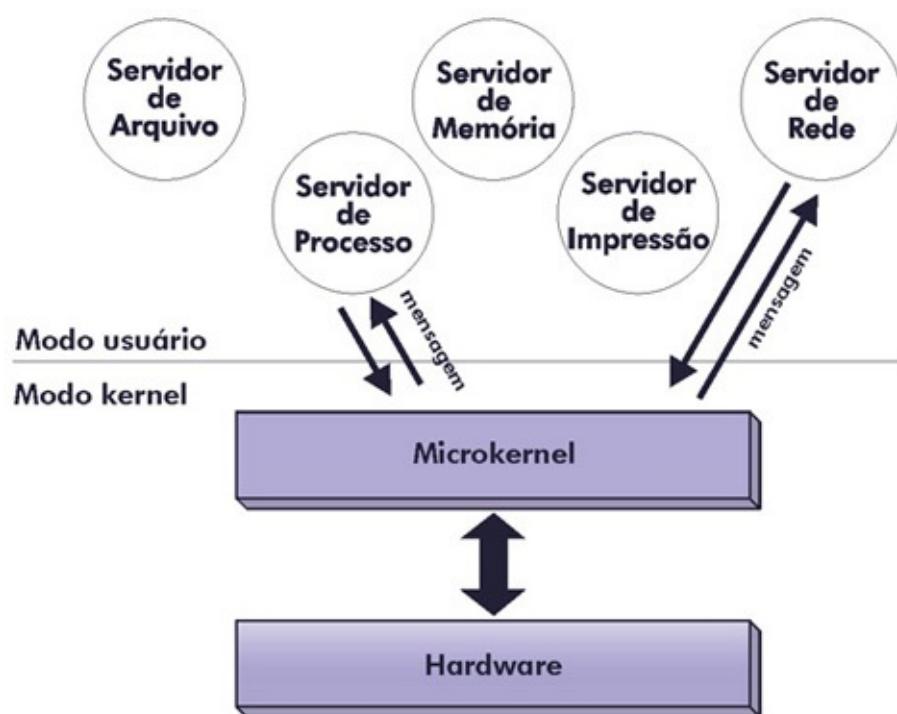


Fig. 4.9 Arquitetura microkernel.

O conceito da *arquitetura microkernel* surgiu no sistema operacional Mach, na década de 1980, na

Universidade Carnegie-Mellon. O núcleo do sistema Mach oferece basicamente quatro serviços: gerência de processos, gerência de memória, comunicação por troca de mensagens e operações de E/S, todos em modo usuário.

A utilização deste modelo permite que os servidores executem em modo usuário, ou seja, não tenham acesso direto a certos componentes do sistema. Apenas o núcleo do sistema, responsável pela comunicação entre clientes e servidores, executa no modo kernel. Como consequência, se ocorrer um erro em um servidor, este poderá parar, mas o sistema não ficará inteiramente comprometido, aumentando assim a sua disponibilidade.

Como os servidores se comunicam através de trocas de mensagens, não importa se os clientes e servidores são processados em um sistema com um único processador, com múltiplos processadores (fortemente acoplado) ou ainda em um ambiente de sistema distribuído (fracamente acoplado). A implementação de sistemas microkernel em ambientes distribuídos permite que um cliente solicite um serviço e a resposta seja processada remotamente. Esta característica permite acrescentar novos servidores à medida que o número de clientes aumenta, conferindo uma grande escalabilidade ao sistema operacional.

Além disso, a arquitetura microkernel permite isolar as funções do sistema operacional por diversos processos servidores pequenos e dedicados a serviços específicos, tornando o núcleo menor, mais fácil de depurar e, consequentemente, aumentando sua confiabilidade. Na arquitetura microkernel, o sistema operacional passa a ser de mais fácil manutenção, flexível e de maior portabilidade.

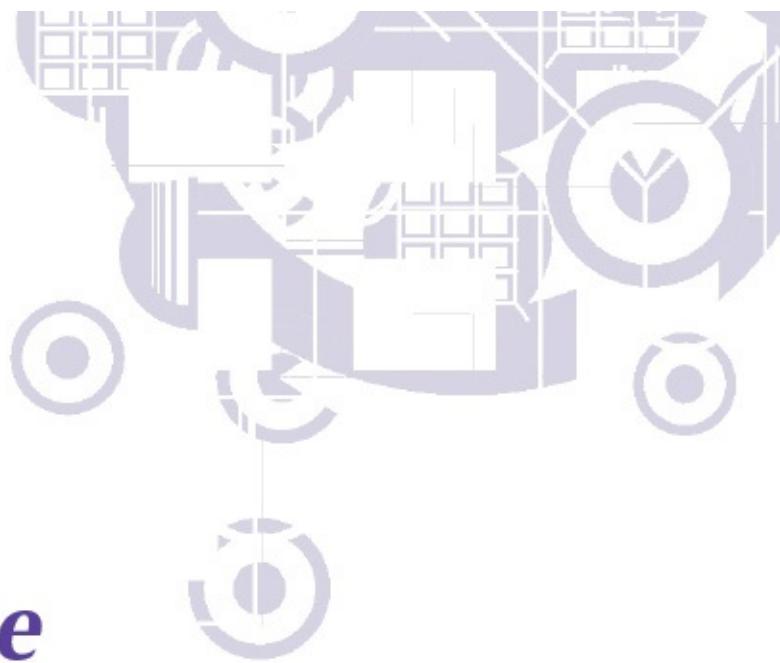
Apesar de todas as vantagens deste modelo, sua implementação, na prática, é muito difícil. Primeiro existe o problema de desempenho, devido à necessidade de mudança de modo de acesso a cada comunicação entre clientes e servidores. Outro problema é que certas funções do sistema operacional exigem acesso direto ao hardware, como operações de E/S. Na realidade, o que é implementado mais usualmente é uma combinação do modelo de camadas com a arquitetura microkernel. O núcleo do sistema, além de ser responsável pela comunicação entre cliente e servidor, passa a incorporar outras funções críticas do sistema, como escalonamento, tratamento de interrupções e gerência de dispositivos.

Existem vários projetos baseados em sistemas microkernel, principalmente em instituições de ensino e centros de pesquisa, como o Exokernel, do MIT (Massachusetts Institute of Technology), L4, da Universidade de Dresden, e Amoeba, da Vrije Universiteit. A maioria das iniciativas nesta área está relacionada ao desenvolvimento de sistemas operacionais distribuídos.

► 4.9 Exercícios

1. O que é o núcleo do sistema e quais são suas principais funções?
2. O que são instruções privilegiadas e não privilegiadas? Qual a relação dessas instruções com os modos de acesso?
3. Explique como funciona a mudança de modos de acesso e dê um exemplo de como um programa faz uso desse mecanismo.
4. Como o kernel do sistema operacional pode ser protegido pelo mecanismo de modos de acesso?

5. Por que as rotinas do sistema operacional possuem instruções privilegiadas?
6. O que é uma system call e qual sua importância para a segurança do sistema? Como as system calls são utilizadas por um programa?
7. Quais das instruções a seguir devem ser executadas apenas em modo kernel? Desabilitar todas as interrupções, consultar a data e a hora do sistema, alterar a data e a hora do sistema, alterar informações residentes no núcleo do sistema, somar duas variáveis declaradas dentro do programa, realizar um desvio para uma instrução dentro do próprio programa e acessar diretamente posições no disco.
8. Pesquise comandos disponíveis em linguagens de controle de sistemas operacionais.
9. Explique o processo de ativação (boot) do sistema operacional.
10. Compare as arquiteturas monolítica e de camadas. Quais as vantagens e desvantagens de cada arquitetura?
11. Quais as vantagens do modelo de máquina virtual?
12. Como funciona o modelo cliente-servidor na arquitetura microkernel? Quais as vantagens e desvantagens dessa arquitetura?
13. Por que a utilização da programação orientada a objetos é um caminho natural para o projeto de sistemas operacionais?



Parte II

Processos e Threads

*“Anyone who stops learning is old, whether twenty or eighty.
Anyone who keeps learning stays young.
The greatest thing in life is to keep your mind young.”*

Henry Ford (1863-1947)
Empreendedor americano

“O homem nasceu para aprender, aprender tanto quanto a vida lhe permita.”

Guimarães Rosa (1908-1967)
Escritor e diplomata brasileiro

“O reconhecimento da ignorância é o princípio da sabedoria.”

Sócrates (470 a.C.-399 a.C.)
Filósofo grego

“Feliz aquele que transfere o que sabe e aprende o que ensina.”

Cora Coralina (1889-1985)
Poetisa brasileira

“Onde há educação não há distinção de classe.”

Confúcio (550 a.C.-479 a.C.)
Filósofo chinês



Processo

► 5.1 Introdução

A gerência de um ambiente multiprogramável é função exclusiva do sistema operacional que deve controlar a execução dos diversos programas e o uso concorrente do processador e demais recursos. Para isso, um programa ao ser executado deve estar sempre associado a um processo. O conceito de processo é a base para a implementação de um sistema multiprogramável.

A gerência de processos é uma das principais funções de um sistema operacional, possibilitando aos programas alocar recursos, compartilhar dados, trocar informações e sincronizar suas execuções. Nos sistemas multiprogramáveis os processos são executados concurrentemente, compartilhando o uso do processador, memória principal e dispositivos de E/S, dentre outros recursos. Nos sistemas com múltiplos processadores não só existe a concorrência de processos pelo uso do processador como também a possibilidade de execução simultânea de processos nos diferentes processadores.

Neste capítulo serão abordados os principais conceitos relacionados a processos, como sua estrutura, estados de execução, tipos de processos e sinais.

► 5.2 Estrutura do Processo

O processador é projetado para executar instruções a partir do ciclo de busca e execução. Neste ciclo, o processador busca a instrução a ser executada na memória principal, armazena-a no registrador de instruções para, finalmente, decodificar seus bits e realizar a execução. O registrador PC tem a função de armazenar sempre o endereço da próxima instrução a ser executada, e as alterações do seu conteúdo determinam o sequenciamento de execução das instruções armazenadas na memória principal.

Na visão da camada de hardware, o processador executa instruções sem distinguir qual programa encontra-se em processamento. É de responsabilidade do sistema operacional implementar a concorrência entre programas gerenciando a alternância da execução de instruções na UCP de maneira

controlada e segura. Neste sentido, o conceito de processo torna-se essencial para que os sistemas multiprogramáveis implementem a concorrência de diversos programas e atendam a múltiplos usuários simultaneamente.

Um *processo* pode ser entendido inicialmente como um programa em execução, só que seu conceito é mais abrangente. Para que a concorrência entre os programas ocorra sem problemas, é necessário que todas as informações do programa interrompido sejam guardadas para que, quando este voltar a ser executado, não lhe falte nenhuma informação necessária à continuação do processamento. Estas informações são fundamentais para que o sistema operacional possa gerenciar a execução concorrente de programas, e é a base de qualquer ambiente multiprogramável. O conceito de processo pode ser definido como sendo o conjunto necessário de informações para que o sistema operacional implemente a concorrência de programas.

A Fig. 5.1 ilustra a concorrência de três programas (PROG_1, PROG_2, PROG_3) associados aos Processos X, Y e Z. No intervalo de tempo Δt_1 , o processador executa instruções do PROG_1. No instante de tempo t_4 , o sistema operacional decide interromper temporariamente a execução do PROG_1 e salva o conteúdo dos registradores do processador, armazenando-os no Processo X. A seguir, o PROG_2 é iniciado e executado ao longo do intervalo Δt_2 . No instante t_7 , o sistema operacional decide interromper o PROG_2 e salva o conteúdo dos registradores no Processo Y. Neste momento o PROG_3 é iniciado, executa no intervalo de tempo Δt_3 até que o sistema operacional decide interrompê-lo, salvar seus registradores no Processo Z e retomar a execução de PROG_1. Para isso, no instante t_{12} o conteúdo dos registradores do processo X é carregado no processador, fazendo com que PROG_1 continue sua execução como se não tivesse sido interrompido. A troca de um processo por outro no processador, comandada pelo sistema operacional, é denominada *mudança de contexto*. É dessa maneira que o sistema operacional implementa e gerencia um ambiente multiprogramável.

Em um sistema multusuário, cada usuário tem seu programa associado a um processo. Ao executar um programa, o usuário tem a impressão de possuir o processador e todos os demais recursos reservados exclusivamente para seu uso. De fato isto não é verdade, visto que todos os recursos estão sendo compartilhados, inclusive a UCP. Nesse caso, o processador executa o programa de um usuário durante um intervalo de tempo e, conforme observado, no instante seguinte estará processando um outro programa.

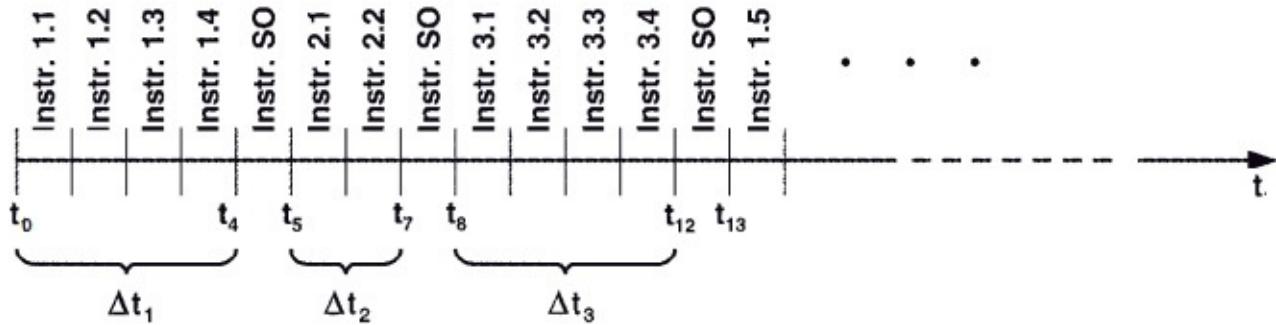
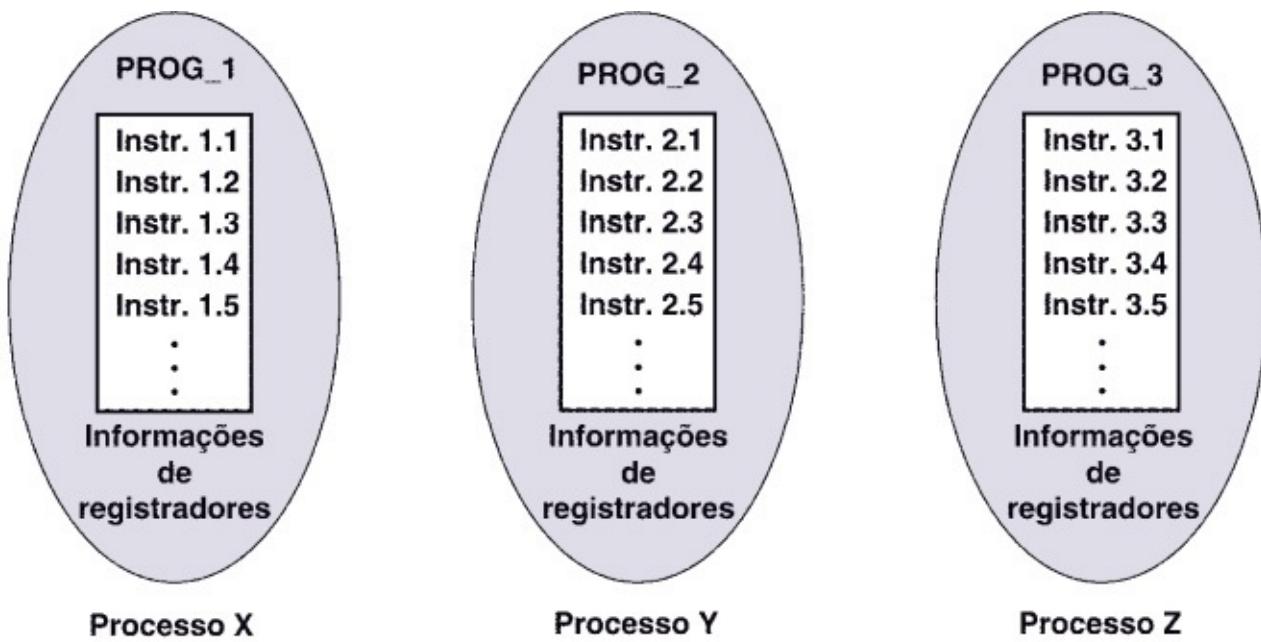


Fig. 5.1 Concorrência de programas e processos.

Um processo também pode ser definido como o ambiente onde um programa é executado. Este ambiente, além das informações sobre a execução, possui também a quantidade de recursos do sistema que cada programa pode utilizar, como o espaço de endereçamento da memória principal, tempo de processador e área em disco.

O resultado da execução de um mesmo programa pode variar, dependendo do processo em que é executado, ou seja, em função dos recursos que são disponibilizados para o programa. A falta de recursos pode impedir a execução com sucesso de um programa. Caso um programa, por exemplo, necessite utilizar uma área em disco superior ao seu limite, o sistema operacional irá interromper sua execução por falta de recursos disponíveis.

Um processo é formado por três partes, conhecidas como contexto de hardware, contexto de software e espaço de endereçamento, que juntos mantêm todas as informações necessárias à execução de um programa. A Fig. 5.2 ilustra de maneira abstrata os componentes da estrutura do processo.

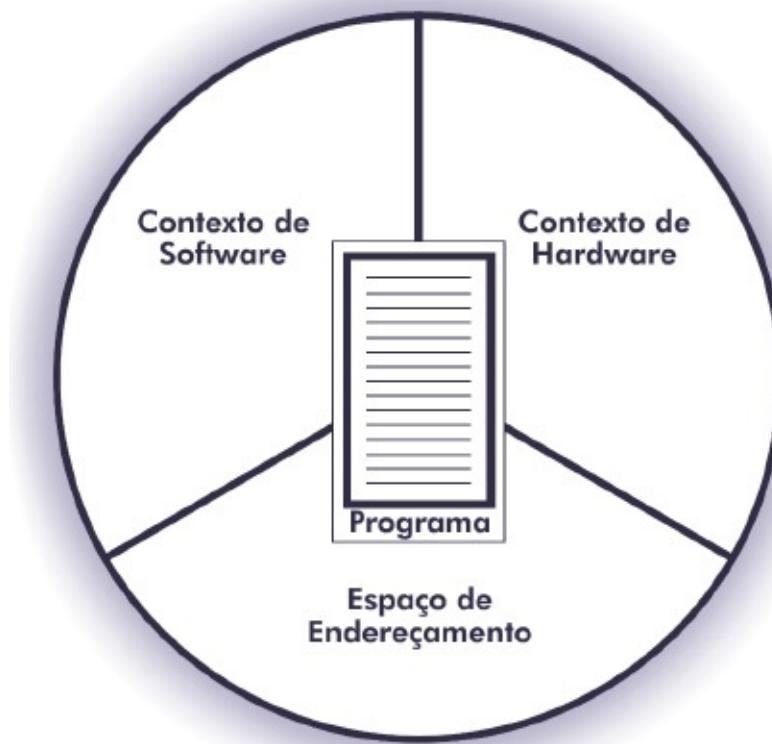


Fig. 5.2 Estrutura do processo.

5.2.1 Contexto de Hardware

O *contexto de hardware* de um processo armazena o conteúdo dos registradores gerais da UCP, além dos registradores de uso específico, como program counter (PC), stack pointer (SP) e registrador de status. Quando um processo está em execução, o seu contexto de hardware está armazenado nos registradores do processador. No momento em que o processo perde a utilização da UCP, o sistema salva as informações no contexto de hardware do processo.

O contexto de hardware é fundamental para a implementação dos sistemas multiprogramáveis, onde os processos se alternam na utilização da UCP, podendo ser interrompidos e, posteriormente, restaurados. O sistema operacional gerencia a mudança de contexto, base para a implementação da concorrência, que consiste em salvar o conteúdo dos registradores do processo que está deixando a UCP e carregá-lo com os valores referentes ao do novo processo que será executado. Essa operação se resume em substituir o contexto de hardware de um processo pelo de outro (Fig. 5.3).

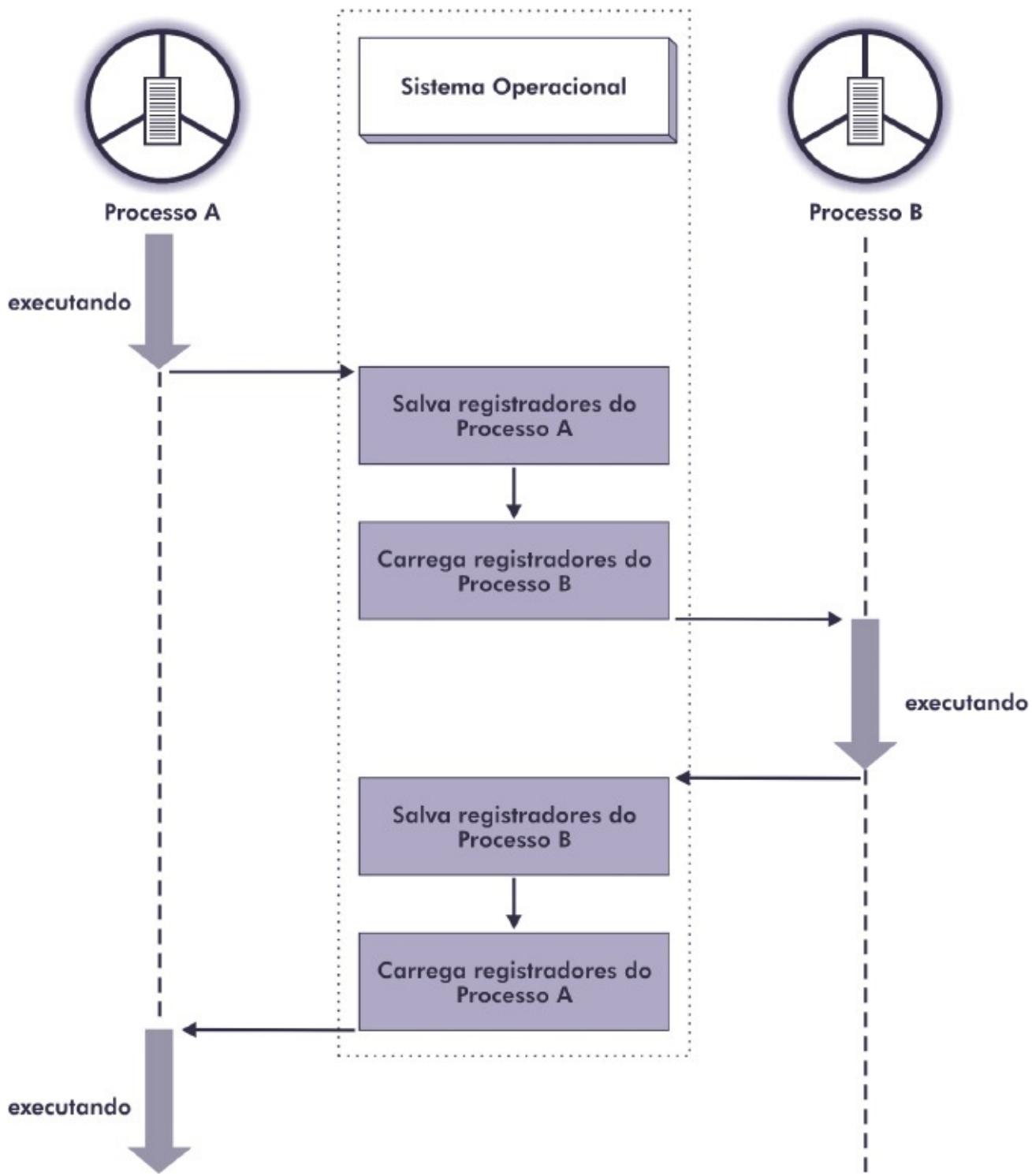


Fig. 5.3 Mudança de contexto.

5.2.2 Contexto de Software

No *contexto de software* de um processo são especificados limites e características dos recursos que podem ser alocados pelo processo, como o número máximo de arquivos abertos simultaneamente, prioridade de execução e tamanho do buffer para operações de E/S. Muitas destas características são determinadas no momento da criação do processo, enquanto outras podem ser alteradas durante sua existência.

A maior parte das informações do contexto de software do processo provém de um arquivo do sistema operacional, conhecido como *arquivo de usuários*. Neste arquivo são especificados os limites dos

recursos que cada processo pode alocar, sendo gerenciado pelo administrador do sistema. Outras informações presentes no contexto de software são geradas dinamicamente ao longo da execução do processo.

O contexto de software é composto por três grupos de informações sobre o processo: identificação, quotas e privilégios.

■ Identificação

Cada processo criado pelo sistema recebe uma identificação única (PID — process identification) representada por um número. Através do PID, o sistema operacional e outros processos podem fazer referência a qualquer processo existente, consultando seu contexto ou alterando uma de suas características. Alguns sistemas, além do PID, identificam o processo através de um nome.

O processo também possui a identificação do usuário ou processo que o criou (owner). Cada usuário possui uma identificação única no sistema (UID — user identification), atribuída ao processo no momento de sua criação. A UID permite implementar um modelo de segurança, onde apenas os objetos (processos, arquivos, áreas de memória etc.) que possuem a mesma UID do usuário (processo) podem ser acessados.

■ Quotas

As quotas são os limites de cada recurso do sistema que um processo pode alocar. Caso uma quota seja insuficiente, o processo poderá ser executado lentamente, interrompido durante seu processamento ou mesmo não ser executado. Alguns exemplos de quotas presentes na maioria dos sistemas operacionais são:

- número máximo de arquivos abertos simultaneamente;
- tamanho máximo de memória principal e secundária que o processo pode alocar;
- número máximo de operações de E/S pendentes;
- tamanho máximo do buffer para operações de E/S;
- número máximo de processos, subprocessos e threads que podem ser criados.

■ Privilégios

Os privilégios ou direitos definem as ações que um processo pode fazer em relação a ele mesmo, aos demais processos e ao sistema operacional.

Privilégios que afetam o próprio processo permitem que suas características possam ser alteradas, como prioridade de execução, limites alocados na memória principal e secundária etc. Já os privilégios que afetam os demais processos permitem, além da alteração de suas próprias características, alterar as de outros processos.

Privilégios que afetam o sistema são os mais amplos e poderosos, pois estão relacionados à operação e à gerência do ambiente, como a desativação do sistema, alteração de regras de segurança, criação de outros processos privilegiados, modificação de parâmetros de configuração do sistema, entre outros. A maioria dos sistemas operacionais disponibiliza uma conta de acesso com todos estes privilégios

disponíveis, com o propósito de o administrador gerenciar o sistema operacional. No sistema Unix existe a conta “root”, no MS Windows a conta “administrator” e no OpenVMS existe a conta “system” com este mesmo perfil.

5.2.3 Espaço de Endereçamento

O espaço de endereçamento é a área de memória pertencente ao processo onde instruções e dados do programa são armazenados para execução. Cada processo possui seu próprio espaço de endereçamento, que deve ser devidamente protegido do acesso dos demais processos. No Capítulo 9 — Gerência de Memória e no Capítulo 10 — Gerência de Memória Virtual serão apresentados diversos mecanismos de implementação e administração do espaço de endereçamento.

A Fig. 5.4 ilustra as características da estrutura de um processo.



Fig. 5.4 Características da estrutura de um processo.

5.2.4 Bloco de Controle do Processo

O processo é implementado pelo sistema operacional através de uma estrutura de dados chamada *bloco de controle do processo* (*Process Control Block* — *PCB*). A partir do *PCB*, o sistema operacional mantém todas as informações sobre o contexto de hardware, contexto de software e espaço de

endereçamento de cada processo (Fig. 5.5).

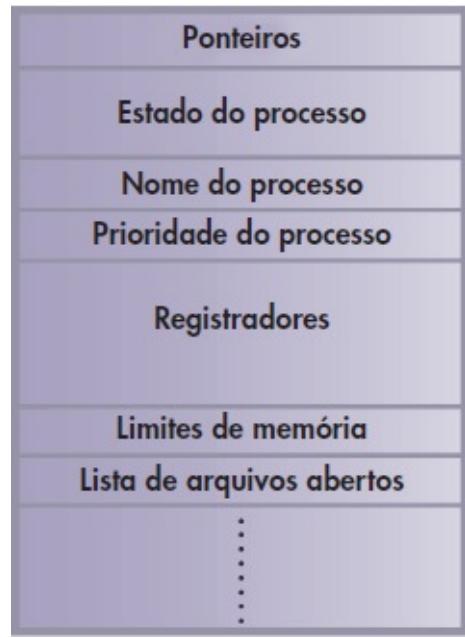


Fig. 5.5 Bloco de Controle do Processo (PCB).

Os PCBs de todos os processos ativos residem na memória principal em uma área exclusiva do sistema operacional. O tamanho desta área, geralmente, é limitado por um parâmetro do sistema operacional que permite especificar o número máximo de processos que podem ser suportados simultaneamente pelo sistema.

#	ps	-l	-A	F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
4	S	0	1	0	0	75	0	-	378	schedu	?		00:00:04	init			
1	S	0	2	1	0	75	0	-	0	contex	?		00:00:00	keventd			
1	S	0	3	1	0	94	19	-	0	ksofti	?		00:00:00	ksoftirqd/0			
1	S	0	6	1	0	85	0	-	0	bdflus	?		00:00:00	bdflush			
1	S	0	4	1	0	75	0	-	0	schedu	?		00:05:35	kswapd			
1	S	0	5	1	0	75	0	-	0	schedu	?		00:03:45	kscand			
1	S	0	7	1	0	75	0	-	0	schedu	?		00:00:00	kupdated			
1	S	0	8	1	0	85	0	-	0	md_thr	?		00:00:00	mdrecoveryd			
1	S	0	21	1	0	75	0	-	0	end	?		00:05:40	kjournald			
1	S	0	253	1	0	75	0	-	0	end	?		00:00:00	kjournald			
1	S	0	254	1	0	75	0	-	0	end	?		00:00:00	kjournald			
1	S	0	255	1	0	75	0	-	0	end	?		00:55:28	kjournald			
1	S	0	579	1	0	75	0	-	399	schedu	?		00:02:00	syslogd			
5	S	0	583	1	0	75	0	-	383	do_sys	?		00:00:00	klogd			
5	S	32	600	1	0	75	0	-	414	schedu	?		00:00:00	portmap			
5	S	29	619	1	0	85	0	-	416	schedu	?		00:00:00	rpc.statd			
1	S	0	631	1	0	75	0	-	393	schedu	?		00:00:00	mdadm			
5	S	0	702	1	0	75	0	-	917	schedu	?		00:00:30	sshd			
5	S	0	716	1	0	75	0	-	539	schedu	?		00:00:00	xinetd			
5	S	0	745	1	0	75	0	-	398	schedu	?		00:00:00	gpm			
5	S	0	765	1	0	75	0	-	607	schedu	?		00:00:16	crond			

Fig. 5.6 Visualização de processos no Linux.

A Fig. 5.6 exibe, a partir da execução do comando `ps` (process status), a listagem de alguns dos processos existentes em uma estação de trabalho com sistema operacional Linux. Diversas características do contexto de software do processo são apresentadas. Por exemplo, a coluna PID indica a identificação do processo, PRI indica a prioridade e TIME o tempo de utilização do processador. De forma semelhante, é possível visualizar os processos existentes em um sistema MS Windows utilizando o Gerenciador de Tarefas (Task Manager).

Toda a gerência dos processos é realizada por intermédio de chamadas a rotinas do sistema operacional que realizam operações como criação, alteração de características, visualização, eliminação, sincronização, suspensão de processos, dentre outras.

► 5.3 Estados do Processo

Em um sistema multiprogramável, um processo não deve alocar exclusivamente a UCP, de forma que exista um compartilhamento no uso do processador. Os processos passam por diferentes estados ao longo do seu processamento, em função de eventos gerados pelo sistema operacional ou pelo próprio processo. Um processo ativo pode encontrar-se em três diferentes estados:

■ Execução (running)

Um processo é dito no *estado de execução* quando está sendo processado pela UCP. Em sistemas com apenas uma UCP, somente um processo pode estar sendo executado em um dado instante de tempo. Os processos se alternam na utilização do processador seguindo uma política estabelecida pelo sistema operacional.

Em sistemas com múltiplos processadores, existe a possibilidade de mais de um processo ser executado ao mesmo tempo. Neste tipo de sistema, também é possível um mesmo processo ser executado simultaneamente em mais de uma UCP (processamento paralelo).

■ Pronto (ready)

Um processo está no *estado de pronto* quando aguarda apenas para ser executado. O sistema operacional é responsável por determinar a ordem e os critérios pelos quais os processos em estado de pronto devem fazer uso do processador. Este mecanismo é conhecido como escalonamento.

Em geral existem vários processos no sistema no estado de pronto organizados em listas encadeadas. Os processos devem estar ordenados pela sua importância, permitindo que processos mais prioritários sejam selecionados primeiramente para execução (Fig. 5.7).

■ Espera (wait)

Um processo no *estado de espera* aguarda por algum evento externo ou por algum recurso para prosseguir seu processamento. Como exemplo, podemos citar o término de uma operação de entrada/saída ou a espera de uma determinada data e/ou hora para continuar sua execução. Em alguns sistemas operacionais, o estado de espera pode ser chamado de bloqueado (blocked).

O sistema organiza os vários processos no estado de espera também em listas encadeadas. Em geral, os processos são separados em listas de espera associadas a cada tipo de evento (Fig. 5.7). Nesse caso, quando um evento acontece, todos os processos da lista associada ao evento são transferidos para o estado de pronto.

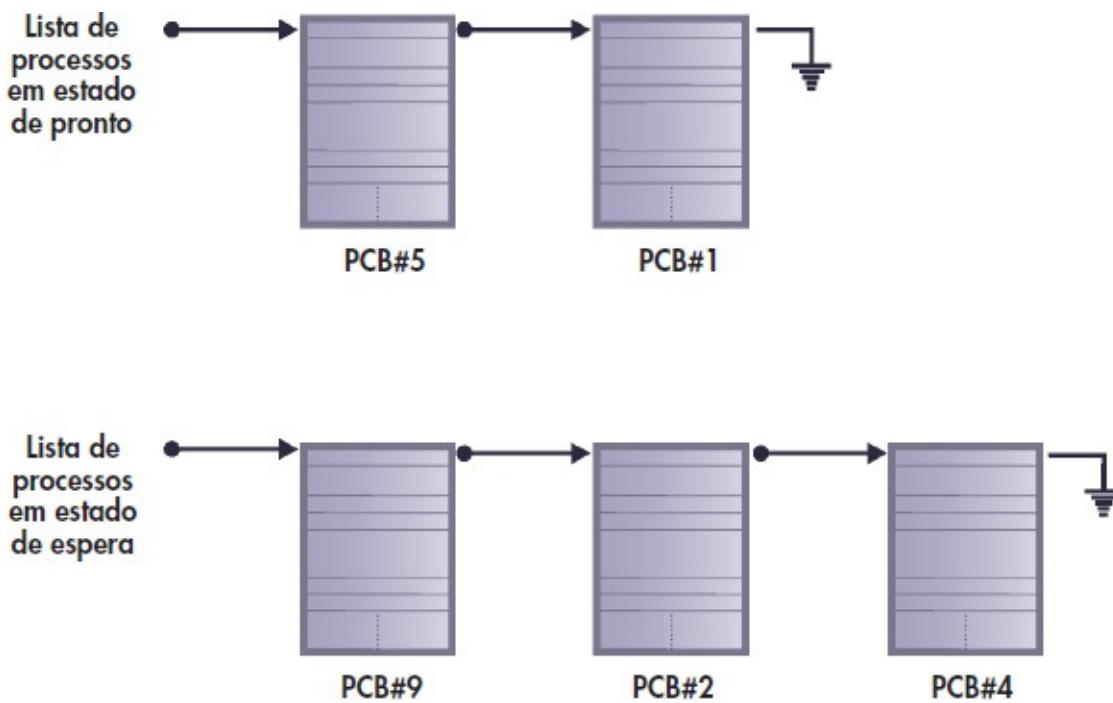


Fig. 5.7 Lista de PCBs nos estados de pronto e de espera.

► 5.4 Mudanças de Estado do Processo

Um processo muda de estado durante seu processamento em função de eventos originados por ele próprio (*eventos voluntários*) ou pelo sistema operacional (*eventos involuntários*). Basicamente, existem quatro mudanças de estado que podem ocorrer a um processo:

■ Pronto → Execução

Após a criação de um processo, o sistema o coloca em uma lista de processos no estado de pronto, onde aguarda por uma oportunidade para ser executado (Fig. 5.8a). Cada sistema operacional tem seus próprios critérios e algoritmos para a escolha da ordem em que os processos serão executados (política de escalonamento). No Capítulo 8 — Gerência do Processador, esses critérios e seus algoritmos serão analisados com detalhes.

■ Execução → Espera

Um processo em execução passa para o estado de espera por eventos gerados pelo próprio processo, como uma operação de E/S, ou por eventos externos (Fig. 5.8b). Um evento externo é gerado, por exemplo, quando o sistema operacional suspende por um período de tempo a execução de um processo.

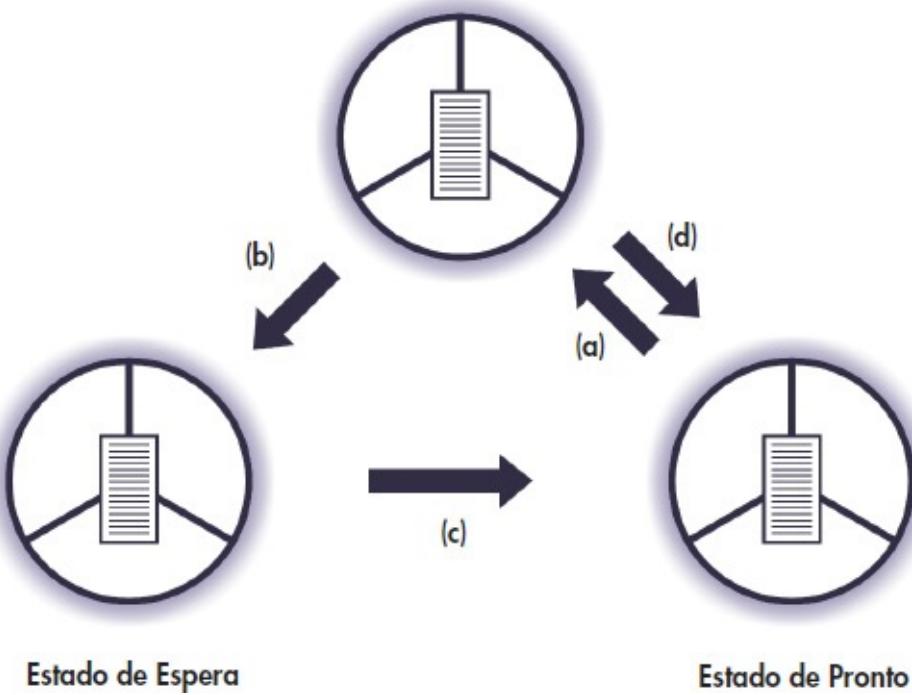


Fig. 5.8 Mudanças de estado do processo.

■ Espera → Pronto

Um processo no estado de espera passa para o estado de pronto quando a operação solicitada é atendida ou o recurso esperado é concedido. Um processo no estado de espera sempre terá de passar pelo estado de pronto antes de poder ser novamente selecionado para execução. Não existe a mudança do estado de espera para o estado de execução diretamente (Fig. 5.8c).

■ Execução → Pronto

Um processo em execução passa para o estado de pronto por eventos gerados pelo sistema, como o término da fatia de tempo que o processo possui para sua execução (Fig. 5.8d). Nesse caso, o processo volta para a fila de pronto, onde aguarda por uma nova oportunidade para continuar seu processamento.

Um processo em estado de pronto ou de espera pode não se encontrar na memória principal. Esta condição ocorre quando não existe espaço suficiente para todos os processos na memória principal e parte do contexto do processo é levado para memória secundária. A técnica conhecida como swapping, na condição citada, retira processos da memória principal (*swap out*) e os traz de volta (*swap in*) seguindo critérios de cada sistema operacional. Neste caso, os processos em estados de espera e pronto podem estar residentes ou não residentes (*outswapped*) na memória principal (Fig. 5.9).

► 5.5 Criação e Eliminação de Processos

Processos são criados e eliminados por motivos diversos. A criação de um processo ocorre a partir do momento em que o sistema operacional adiciona um novo PCB à sua estrutura e aloca um espaço de

endereçamento na memória para uso. A partir da criação do PCB, o sistema operacional já reconhece a existência do processo, podendo gerenciá-lo e associar programas ao seu contexto para serem executados. No caso da eliminação de um processo, todos os recursos associados ao processo são desalocados e o PCB eliminado pelo sistema operacional.

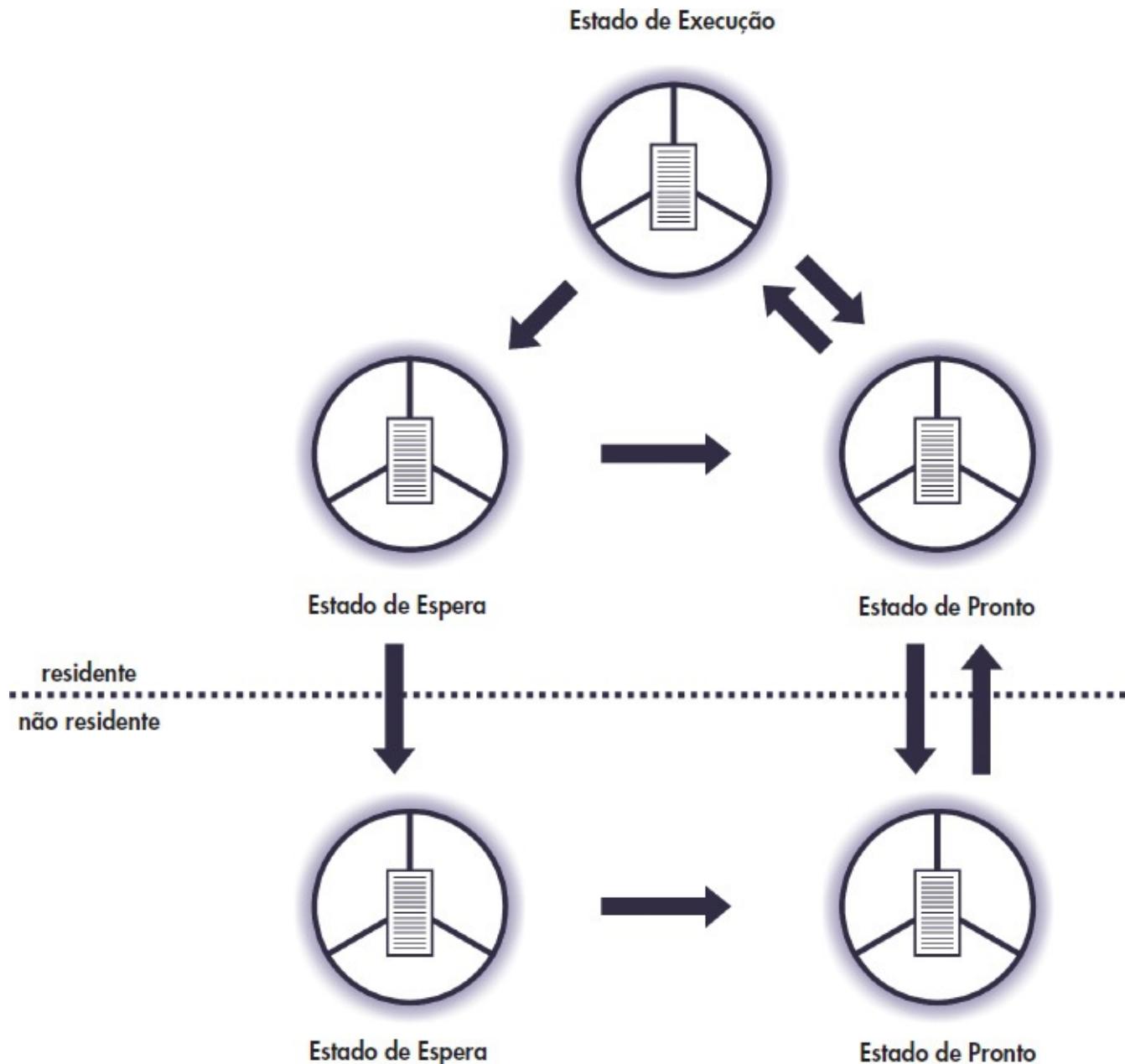


Fig. 5.9 Mudanças de estado do processo (2).

Além dos três estados apresentados anteriormente para o processo, a maioria dos sistemas operacionais estabelece para os momentos de criação e eliminação de um processo dois estados adicionais. A Fig. 5.10 ilustra as mudanças de estado do processo considerando estes dois novos estados.

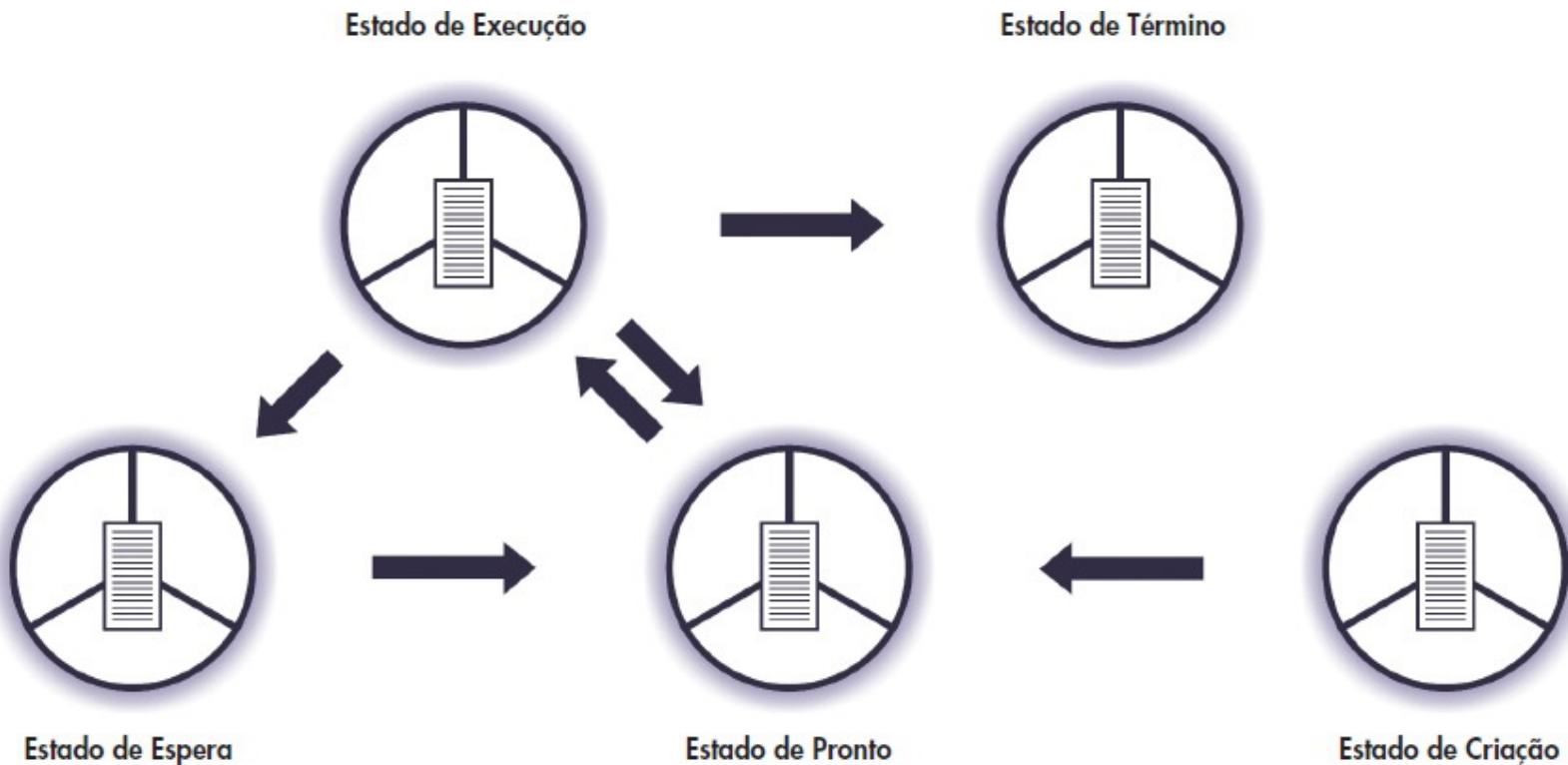


Fig. 5.10 Mudanças de estado do processo (3).

■ Criação (new)

Um processo é dito no *estado de criação* quando o sistema operacional já criou um novo PCB, porém ainda não pode colocá-lo na lista de processos do estado de pronto. Alguns sistemas operacionais limitam o número de processos ativos em função dos recursos disponíveis ou de desempenho. Esta limitação pode ocasionar que processos criados permaneçam no estado de criação até que possam passar para ativos. No item 5.8 são descritas diferentes maneiras de criação de processos.

■ Terminado (exit)

Um processo no *estado de terminado* não poderá ter mais nenhum programa executado no seu contexto, porém o sistema operacional ainda mantém suas informações de controle presentes em memória. Um processo neste estado não é considerado mais ativo, mas como o PCB ainda existe, o sistema operacional pode recuperar informações sobre a contabilização de uso de recursos do processo, como o tempo total do processador. Após as informações serem extraídas, o processo pode deixar de existir.

O término de processo pode ocorrer por motivos como:

- término normal de execução;
- eliminação por um outro processo;
- eliminação forçada por ausência de recursos disponíveis no sistema.

► 5.6 Processos CPU-bound e I/O-bound

Processos podem ser classificados como CPU-bound ou I/O-bound de acordo com a utilização do processador e dos dispositivos de E/S.

Um processo é definido como *CPU-bound* (ligado à UCP) quando passa a maior parte do tempo no estado de execução, utilizando o processador, ou pronto (Fig. 5.11a). Esse tipo de processo realiza poucas operações de leitura e gravação, e é encontrado em aplicações científicas que efetuam muitos cálculos.

Um processo é classificado como *I/O-bound* (ligado à E/S) quando passa a maior parte do tempo no estado de espera, pois realiza um elevado número de operações de E/S (Fig. 5.11b). Esse tipo de processo é encontrado em aplicações comerciais, que se baseiam em leitura, processamento e gravação. Os processos interativos também são bons exemplos de processos I/O-bound, pela forma de comunicação entre o usuário e o sistema, normalmente lenta, devido à utilização de terminais.

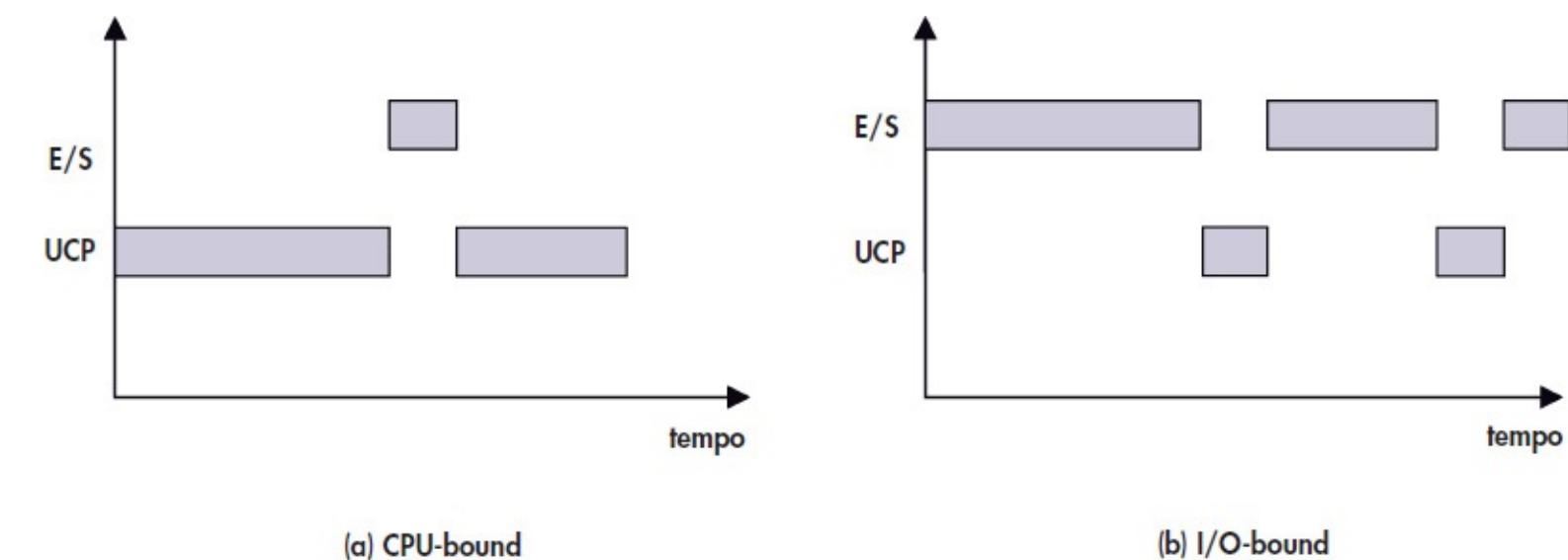


Fig. 5.11 Processos CPU-bound × I/O-bound.

► 5.7 Processos Foreground e Background

Um processo possui sempre pelo menos dois canais de comunicação associados a sua estrutura, pelos quais são realizadas todas as entradas e saídas de dados ao longo do seu processamento. Os canais de entrada (input) e saída (output) de dados podem estar associados a terminais, arquivos, impressoras e até mesmo a outros processos.

Um *processo foreground* é aquele que permite a comunicação direta do usuário com o processo durante o seu processamento. Neste caso, tanto o canal de entrada quanto o de saída estão associados a um terminal com teclado, mouse e monitor, permitindo, assim, a interação com o usuário (Fig. 5.12a). O processamento interativo tem como base processos foreground.

Um *processo background* é aquele onde não existe a comunicação com o usuário durante o seu processamento (Fig. 5.12b). Neste caso, os canais de E/S não estão associados a nenhum dispositivo de E/S interativo, mas em geral a arquivos de E/S. O processamento do tipo batch é realizado através de

processos background.

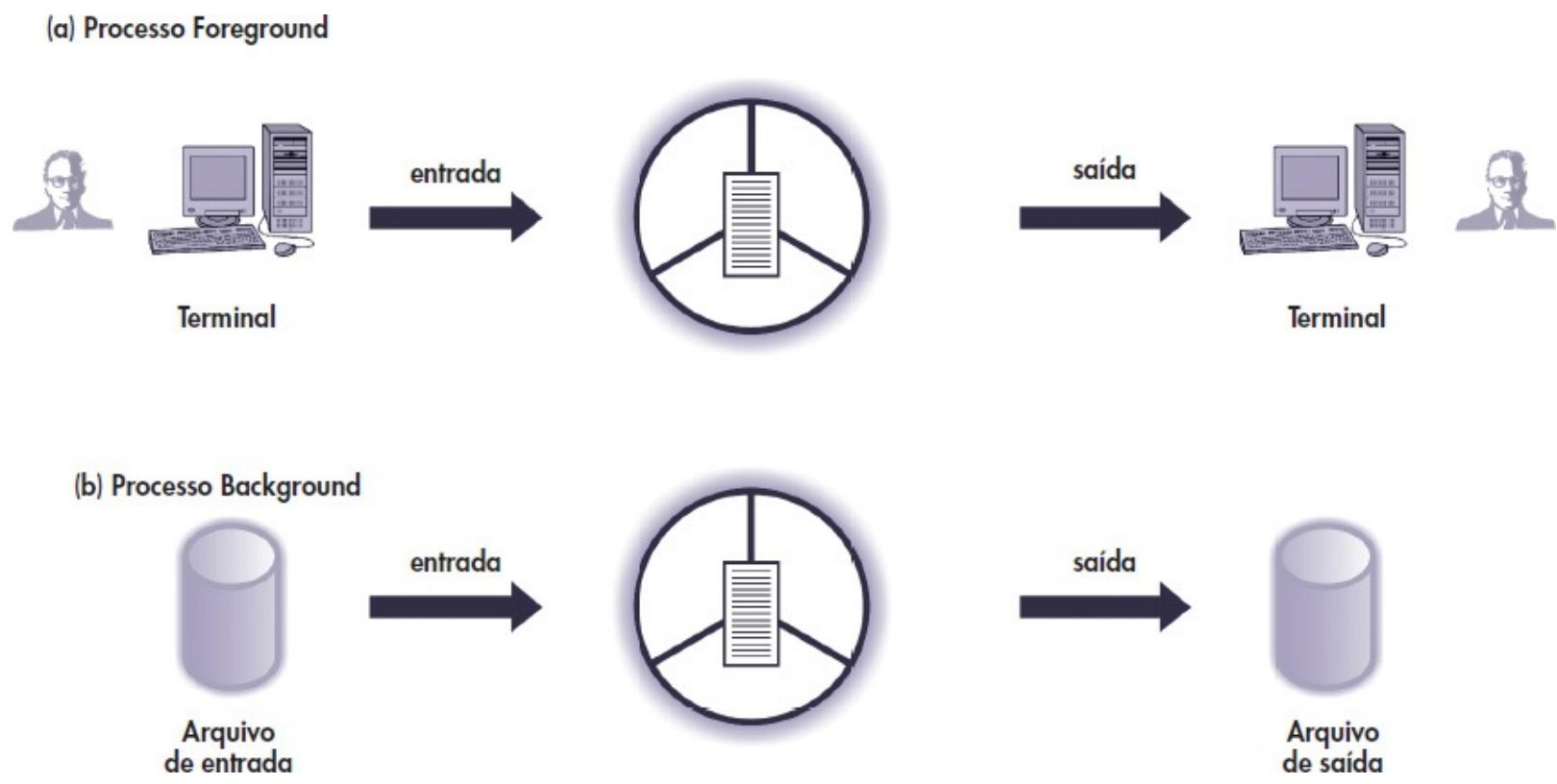


Fig. 5.12 Processos foreground e background.

É possível associar o canal de saída de um processo ao canal de entrada de um outro processo. Neste caso dizemos que existe um *pipe* ligando os dois processos. Se um Processo A gera uma listagem e o Processo B tem como função ordená-la, basta associar o canal de saída do processo A ao canal de entrada do processo B (Fig. 5.13).

► 5.8 Formas de Criação de Processos

Um processo pode ser criado de diversas maneiras. A seguir, são apresentadas as três principais formas de criação de processos:

■ Logon Interativo

No *logon interativo* o usuário, por intermédio de um terminal, fornece ao sistema um nome de identificação (username ou logon) e uma senha (password). O sistema operacional autentica estas informações verificando se estão corretamente cadastradas no arquivo de usuários. Em caso positivo, um processo foreground é criado, possibilitando ao usuário interagir com o sistema utilizando uma linguagem de comandos. Na Fig. 5.14 é apresentado um exemplo de logon interativo em um sistema Unix.

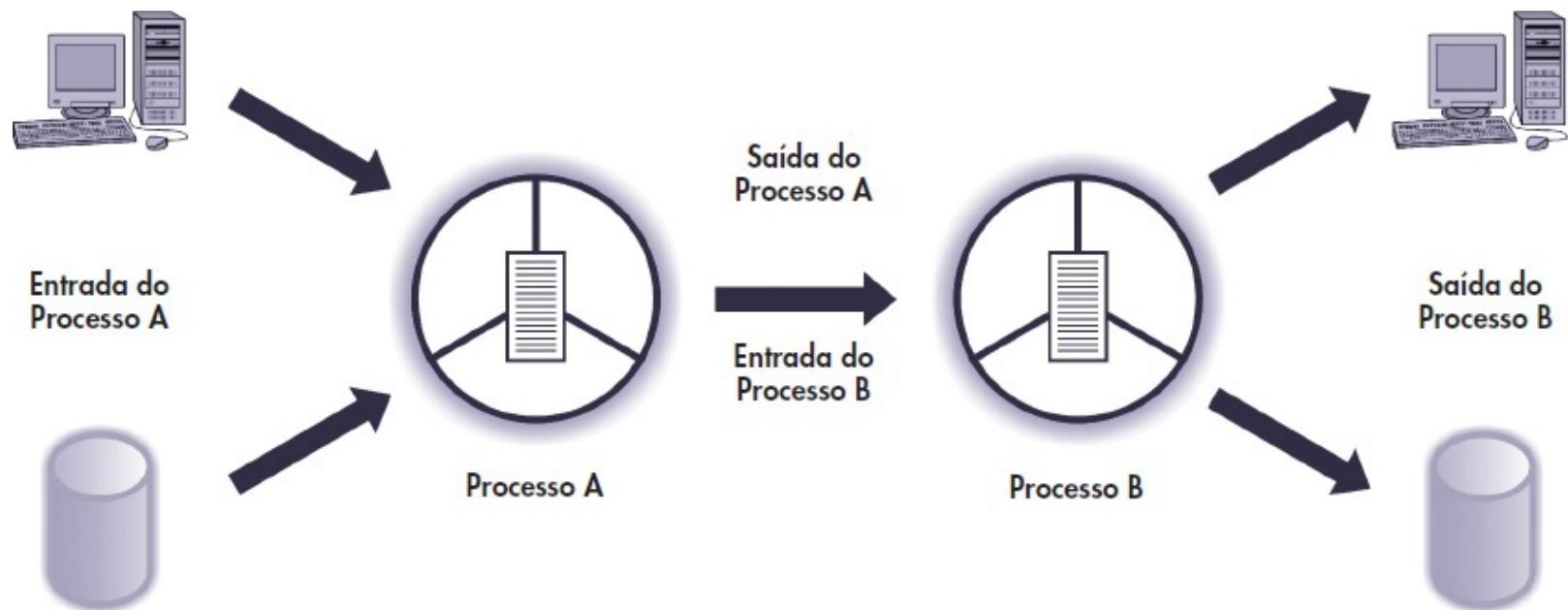


Fig. 5.13 Pipe.

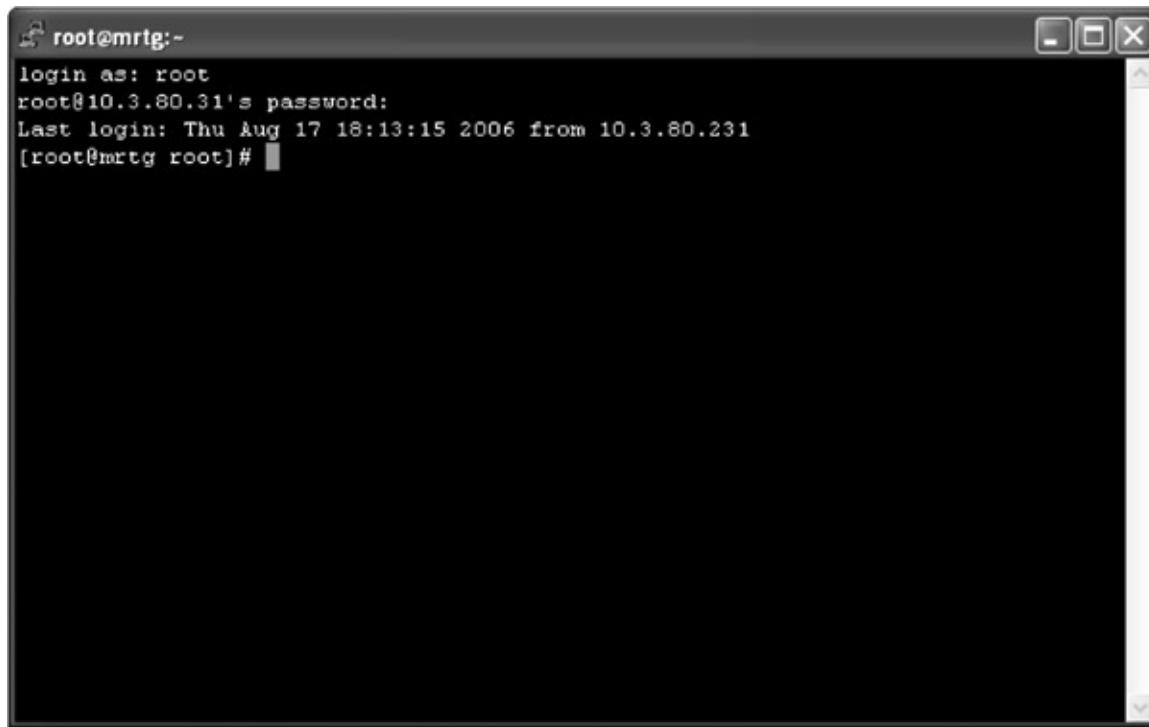


Fig. 5.14 Tela de logon interativo.

Conforme apresentado anteriormente, o *arquivo de usuários* é um arquivo do sistema operacional onde são armazenados todos os usuários autorizados a ter acesso ao sistema. Para cada usuário existe um registro com informações como, por exemplo, username, senha, quotas e privilégios. Apenas o administrador do sistema pode criar, modificar e eliminar registros no arquivo de usuários. Grande parte das informações do contexto de software de um processo provém do arquivo de usuários.

O processo também pode ser eliminado interativamente quando o usuário realiza o procedimento de saída do sistema, também chamado *logout* ou *logoff*, digitando um comando da linguagem de comandos para encerramento da sessão.

■ Via Linguagem de Comandos

Um usuário pode, a partir do seu processo, criar novos processos por intermédio de comandos da linguagem de comandos. O principal objetivo para que um usuário crie diversos processos é a possibilidade de execução de programas concurrentemente. Por exemplo, no sistema OpenVMS o comando *spawn* permite executar uma outra tarefa de forma concorrente. Esta é uma maneira de criar um processo a partir de outro processo já existente. O processo criado pode ser foreground ou background, dependendo do comando de criação utilizado.

■ Via Rotina do Sistema Operacional

Um processo pode ser criado a partir de qualquer programa executável com o uso de rotinas do sistema operacional. A criação deste processo possibilita a execução de outros programas concurrentemente ao programa chamador. A rotina de criação de processos depende do sistema operacional e possui diversos parâmetros, como nome do processo a ser criado, nome do programa executável que será executado dentro do contexto do processo, prioridade de execução, estado do processo, se o processo é do tipo foreground ou background etc.

Como exemplos de rotinas do sistema para a criação de processos temos `sys$createprocess` no OpenVMS, `fork` no Unix e `CreateProcess` no MS Windows. No trecho de programa em Delphi, a seguir, desenvolvido para o MS Windows, há um exemplo da criação de um processo, sendo que o executável é o programa `NOTEPAD.EXE` (bloco de notas). Neste caso, a API abrirá uma janela para a execução concorrente do bloco de notas independente do programa chamador.

```
procedure TForm1.CriaProcesso(Sender: TObject);
var
  status: boolean;
  si: STARTUPINFO;
  pi: PROCESS_INFORMATION;
  Handle: THandle;
  NomeExe: PChar;
begin
  NomeExe := PChar('\WINNT\NOTEPAD.EXE');
  FillChar(si, SizeOf(si), 0);
  si.cb := SizeOf(si);
  status := CreateProcess(NomeExe, nil, nil, nil, TRUE,
    NORMAL_PRIORITY_CLASS, nil, nil, si, pi);
  if (not status) then MessageBox(Handle, 'Erro na criação do
  processo', nil, MB_OK);
end;
```

No exemplo do procedimento `TForm1.CriaProcesso`, o primeiro parâmetro da API `CreateProcess` indica o nome do executável, ou seja, no exemplo será criado um processo para a execução do programa `NOTEPAD.EXE`. O sexto parâmetro especifica a prioridade do processo, o nono dá informações para a criação do processo e o último retorna informações sobre o processo criado. A instrução seguinte à rotina `CreateProcess` testa se o processo foi criado com sucesso.

O programa CriaProcesso, escrito em Java, também cria um processo para a execução do NOTEPAD.EXE, porém a chamada ao sistema é feita implicitamente pela máquina virtual Java. O programa utiliza um tratador de exceção, caso haja algum problema na criação do processo.

```
import java.io.IOException;
public class CriaProcesso {
    public static void main(String[] args) {
        try {
            Process processo =
                Runtime.getRuntime().exec("notepad.exe");
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

► 5.9 Processos Independentes, Subprocessos e Threads

Processos independentes, subprocessos e threads são maneiras diferentes de implementar a concorrência dentro de uma aplicação. Neste caso, busca-se subdividir o código em partes para trabalharem de forma cooperativa. Considere um banco de dados com produtos de uma grande loja, onde vendedores fazem frequentes consultas. Neste caso, a concorrência na aplicação proporciona um tempo de espera menor entre as consultas, melhorando o desempenho da aplicação e beneficiando os usuários.

O uso de *processos independentes* é a maneira mais simples de implementar a concorrência em sistemas multiprogramáveis. Neste caso não existe vínculo do processo criado com o seu criador. A criação de um processo independente exige a alocação de um PCB, possuindo contextos de hardware, contextos de software e espaços de endereçamento próprios.

Subprocessos são processos criados dentro de uma estrutura hierárquica. Neste modo, o processo criador é denominado *processo-pai*, enquanto o novo processo é chamado de *subprocesso* ou *processo-filho*. O subprocesso, por sua vez, pode criar outras estruturas de subprocessos. Uma característica desta implementação é a dependência existente entre o processo criador e o subprocesso. Caso um processo-pai deixe de existir, os subprocessos subordinados são automaticamente eliminados. De modo semelhante aos processos independentes, subprocessos possuem seu próprio PCB. A Fig. 5.15 ilustra cinco processos em uma estrutura hierárquica, cada qual com seu próprio contexto de hardware, contexto de software e espaço de endereçamento.

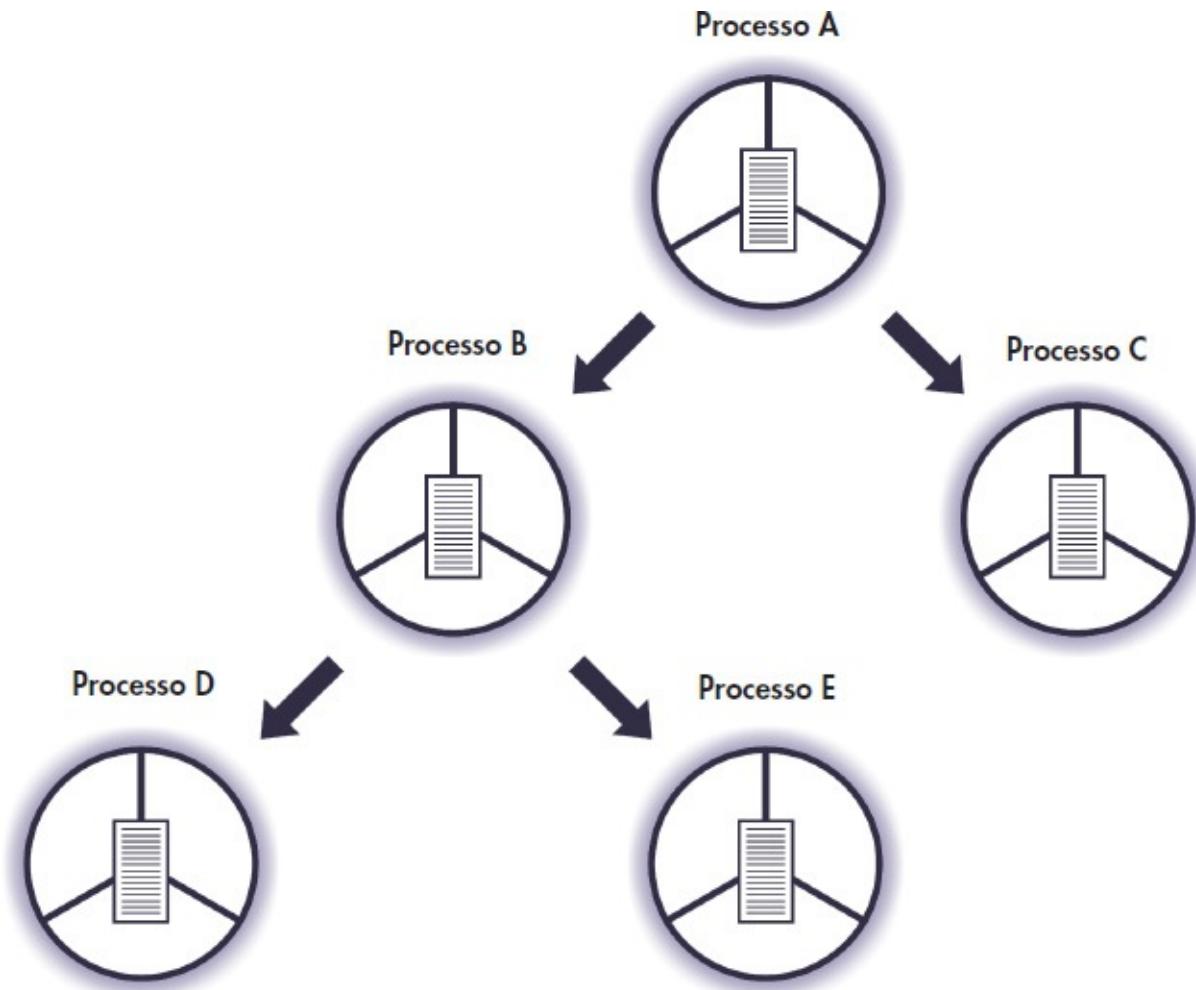


Fig. 5.15 Estrutura de processos e subprocessos.

Além da dependência hierárquica entre processos e subprocessos, uma outra característica neste tipo de implementação é que subprocessos podem compartilhar quotas com o processo-pai. Neste caso, quando um subprocesso é criado o processo-pai cede parte de suas quotas ao processo-filho.

O uso de processos independentes e subprocessos no desenvolvimento de aplicações concorrentes demanda consumo de diversos recursos do sistema. Sempre que um novo processo é criado, o sistema deve alocar recursos (contexto de hardware, contexto de software e espaço de endereçamento), consumindo tempo de UCP neste trabalho. No momento do término dos processos, o sistema operacional também dispensa tempo para desalocar recursos previamente alocados. Outro problema é a comunicação e a sincronização entre processos consideradas pouco eficientes, visto que cada processo possui seu próprio espaço de endereçamento.

O conceito de *thread* foi introduzido na tentativa de reduzir o tempo gasto em criação, eliminação e troca de contexto de processos nas aplicações concorrentes, bem como economizar recursos do sistema como um todo. Em um ambiente multithread, um único processo pode suportar múltiplos threads, cada qual associado a uma parte do código da aplicação (Fig. 5.16). Neste caso não é necessário haver diversos processos para a implementação da concorrência. Threads compartilham o processador da mesma maneira que um processo, ou seja, enquanto um thread espera por uma operação de E/S, outro thread pode ser executado.

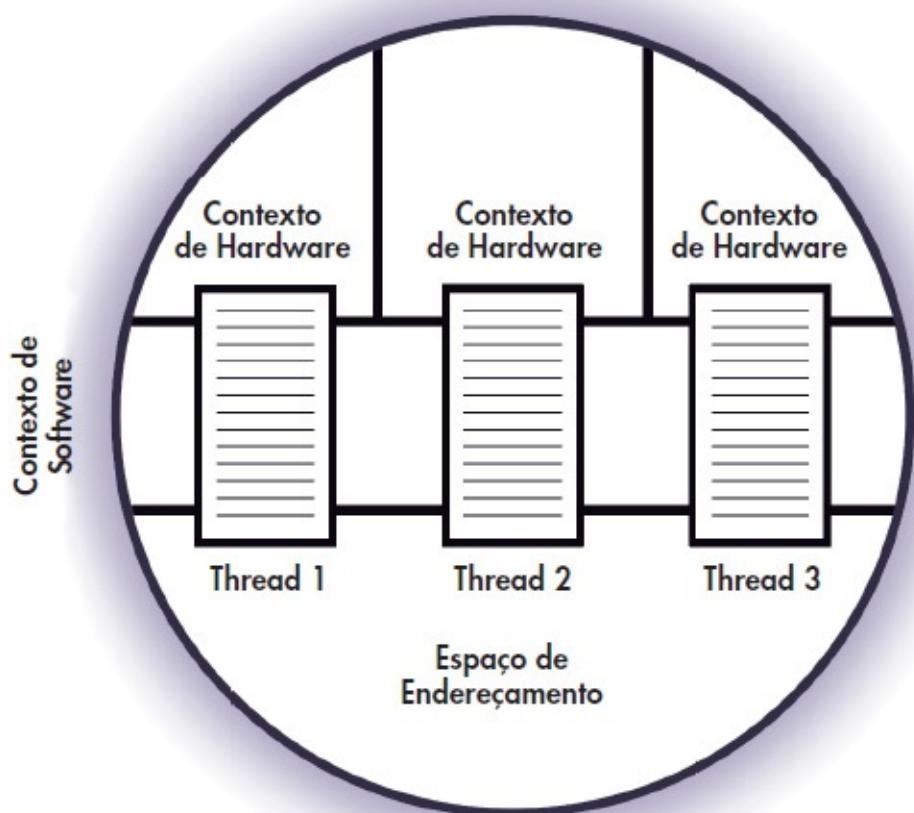


Fig. 5.16 Processo multithread.

Cada thread possui seu próprio contexto de hardware, porém compartilha o mesmo contexto de software e espaço de endereçamento com os demais threads do processo. O compartilhamento do espaço de endereçamento permite que a comunicação de threads dentro de um mesmo processo seja realizada de forma simples e rápida. Este assunto é mais bem detalhado no Capítulo 6 — Thread.

► 5.10 Processos do Sistema Operacional

O conceito de processo, além de estar associado a aplicações de usuários, pode também ser implementado na própria arquitetura do sistema operacional. Como visto no Capítulo 4 — Estrutura do Sistema Operacional, a arquitetura microkernel implementa o uso intensivo de processos que disponibilizam serviços para processos das aplicações e do próprio sistema operacional.

Quando processos são utilizados para a implementação de serviços do sistema, estamos retirando código do seu núcleo, tornando-o menor e mais estável. No caso de um ou mais serviços não serem desejados, basta não ativar os processos responsáveis, o que permitirá liberar memória para os processos dos usuários.

A seguir, são apresentados alguns serviços que o sistema operacional pode implementar através de processos:

- auditoria e segurança;
- serviços de rede;

- contabilização do uso de recursos;
- contabilização de erros;
- gerência de impressão;
- gerência de jobs batch;
- temporização;
- comunicação de eventos;
- interface de comandos (shell).

► 5.11 Sinais

Sinais é um mecanismo que permite notificar processos de eventos gerados pelo sistema operacional ou por outros processos. O uso de sinais é fundamental para a gerência de processos, além de possibilitar a comunicação e sincronização entre processos.

Um exemplo de uso de sinais é quando um usuário utiliza uma sequência de caracteres do teclado, como [Ctrl-C], para interromper a execução de um programa. Neste caso, o sistema operacional gera um sinal a partir da digitação desta combinação de teclas, sinalizando ao processo a ocorrência do evento. No momento que o processo identifica a chegada do sinal, uma rotina específica de tratamento é executada (Fig. 5.17).

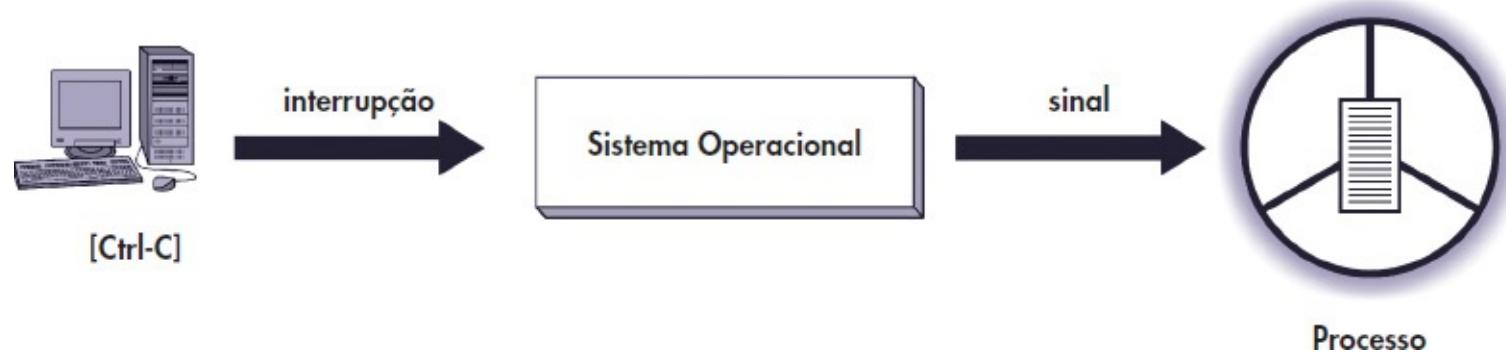


Fig. 5.17 Uso de sinais.

Sinais podem ser utilizados em conjunto com temporizadores, no intuito de sinalizar ao processo algum evento associado ao tempo. Como exemplo, podemos citar a situação em que um processo deve ser avisado periodicamente para realizar alguma tarefa, como monitorar uma fila de pedidos. Depois de realizada a tarefa, o processo deve voltar a esperar pelo próximo sinal de temporização.

A maior parte dos eventos associados a sinais são gerados pelo sistema operacional ou pelo hardware, como a ocorrência de exceções, interrupções geradas por terminais, limites de quotas excedidos durante a execução dos processos e alarmes de tempo. Em outras situações, os eventos são gerados a partir de outros processos com o propósito de sincronizar suas execuções.

A geração de um sinal ocorre quando o sistema operacional, a partir da ocorrência de eventos síncronos ou assíncronos, notifica ao processo através de bits de sinalização localizados no seu PCB. Um processo

não responde instantaneamente a um sinal. Os sinais ficam pendentes até que o processo seja escalonado, quando então serão tratados. Por exemplo, quando um processo é eliminado o sistema ativa o bit associado a este evento. O processo somente será excluído do sistema quando for selecionado para execução. Neste caso, é possível que o processo demore algum tempo até ser eliminado de fato.

O tratamento de um sinal é muito semelhante ao mecanismo de interrupções. Quando um sinal é tratado, o contexto do processo é salvo e a execução desviada para um código de tratamento de sinal (signal handler), geralmente no núcleo do sistema. Após a execução do tratador de sinais, o programa pode voltar a ser processado do ponto onde foi interrompido. Em certas implementações, o próprio processo pode tratar o sinal através de um tratador de sinais definido no código programa. É possível também que um processo bloqueie temporariamente ou ignore por completo alguns sinais.

O mecanismo de sinais assemelha-se ao tratamento de interrupções e exceções vistos no Capítulo 3 — Concorrência, porém com propósitos diferentes. O sinal está para o processo assim como as interrupções e exceções estão para o sistema operacional (Fig. 5.18).

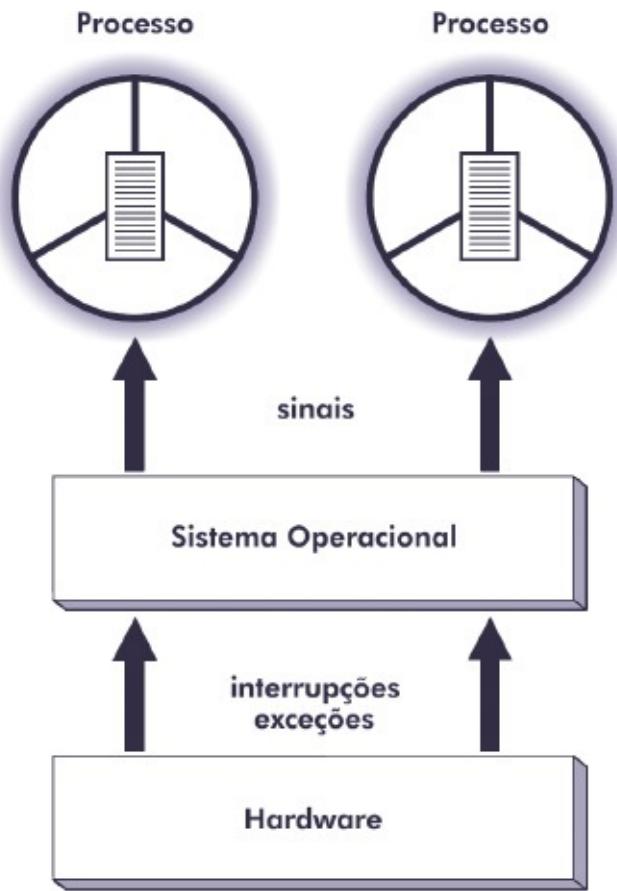


Fig. 5.18 Sinais, interrupções e exceções.

► 5.12 Exercícios

1. Defina o conceito de processo.
2. Por que o conceito de processo é tão importante no projeto de sistemas multiprogramáveis?
3. É possível que um programa execute no contexto de um processo e não execute no contexto de um

outro? Por quê?

4. Quais partes compõem um processo?
5. O que é o contexto de hardware de um processo e como é a implementação da troca de contexto?
6. Qual a função do contexto de software? Exemplifique cada grupo de informação.
7. O que é o espaço de endereçamento de um processo?
8. Como o sistema operacional implementa o conceito de processo? Qual a estrutura de dados indicada para organizar os diversos processos na memória principal?
9. Defina os cinco estados possíveis de um processo.
10. Dê um exemplo que apresente todas as mudanças de estado de um processo, juntamente com o evento associado a cada mudança.
11. Diferencie processos multithreads, subprocessos e processos independentes.
12. Explique a diferença entre processos foreground e background.
13. Qual a relação entre processo e arquitetura microkernel?
14. Dê exemplos de aplicações CPU-bound e I/O-bound.
15. Justifique com um exemplo a frase “o sinal está para o processo assim como as interrupções e exceções estão para o sistema operacional”.
16. Explique como a eliminação de um processo utiliza o mecanismo de sinais.

► 5.13 Laboratório com o Simulador SOsim

Atividade 1: Criação de Processos

a) Práticas de simulação

- Execute o simulador SOsim e identifique as quatro janelas que são abertas na inicialização.
- Crie um processo: janela *Gerência de Processos / Criar* – janela *Criação de Processos / Criar*.

b) Análise prática

- Na janela *Gerência de Processos*, observe algumas informações sobre o contexto de software do processo como PID, prioridade, estado do processo e tempo de processador.
- Na janela *Gerência de Processador*, observe o processo transicionado entre estados.
- Na janela *Gerência de Processador*, movimente a barra de *Clock de UCP* e observe as variações ocorridas.

c) Questão teórica para responder com a ajuda do simulador

Com base na observação do comportamento do processo criado, identifique se o processo é I/O-bound ou CPU-bound. Justifique a resposta.

Atividade 2: Tipos de Processos

a) Práticas de simulação

- Reinicialize o simulador.
- Crie um processo do tipo CPU-bound: janela *Gerência de Processos / Criar* – janela *Criação de Processos / Criar* (*tipo de processo deve ser CPU-bound*).
- Crie outro processo do tipo I/O-bound: janela *Gerência de Processos / Cria* – janela *Criação de Processos / Criar* (*tipo de processo deve ser I/O-bound*).

b) Análise prática

- Na janela *Gerência de Processos*, observe as mudanças de estado dos dois processos.
- Na janela *Gerência de Processador*, observe o comportamento dos processos e as mudanças de contexto em função do tipo I/O-bound e CPU-bound.
- Na janela *Gerência de Processos*, compare a taxa de crescimento do tempo de processador dos dois processos.

c) Questão teórica para responder com a ajuda do simulador

Analise os efeitos gerados no caso de redução do tempo gasto na operação de E/S pelo processo I/O-bound.

Atividade 3: PCB

a) Práticas de simulação

- Reinicialize o simulador.
- Crie dois novos processos: janela *Gerência de Processos / Criar* – janela *Criação de Processos / Criar*.

b) Análise prática

- Na janela *Gerência de Processos / PCB*, observe as informações sobre o contexto de software e hardware dos processos criados.

c) Questão teórica para responder com a ajuda do simulador

Identifique quais informações do PCB são estáticas ou dinâmicas e quais fazem parte do contexto de software e do contexto de hardware.

Atividade 4: Estatísticas

a) Práticas de simulação

- Reinicialize o simulador.
- Ative a janela de Estatísticas em *Console SOsim / Janelas / Estatísticas*.
- Crie dois novos processos: janela *Gerência de Processos / Criar* – janela *Criação de Processos / Criar*.

b) Análise prática

- Na janela *Estatísticas*, observe as informações: número de processos, estados dos processos e processos escalonados.

c) Questão teórica para responder com a ajuda do simulador

Observe que em alguns momentos existem processos no estado de pronto, porém nenhum em estado de execução. Explique a razão dessa situação.

Atividade 5: Log de Execução dos Processos

a) Práticas de simulação

- Reinicialize o simulador.
- Ative a janela de Log em *Console SOsim / Janelas / Log*.
- Crie dois novos processos do tipo CPU-bound: janela *Gerência de Processos / Cria* — janela *Criação de Processos / Criar* (*tipo de processo deve ser CPU-bound*).

b) Análise prática

- Na janela *Log*, observe as informações sobre as mudanças de estado dos processos levando em conta o tempo que cada processo permanece nos estados de Execução e Pronto.
- Configure o simulador com um valor de fatia de tempo diferente.
- Reinicie o simulador SOsim para que a nova parametrização passe a ser válida.
- Observe as diferenças na janela *Log*.

c) Questão teórica para responder usando o simulador

Analise comparativamente a concorrência de dois processos CPU-bound executando em dois sistemas operacionais que se diferenciam apenas pelo valor da fatia de tempo.

Atividade 6: Suspensão e Eliminação de Processos

a) Práticas de simulação

- Reinicialize o simulador.
- Crie dois novos processos: janela *Gerência de Processos / Cria* – janela *Criação de Processos / Criar*.

b) Análise prática

- Na janela *Gerência de Processos*, observe as informações sobre o contexto de software dos processos criados.
- Na janela *Gerência de Processador*, observe a concorrência no uso do processador pelos dois processos.
- Compare percentualmente os tempos de uso do processador entre os dois processos.
- Suspenda temporariamente um dos processos na janela *Gerência de Processos/Suspender*.
- Observe os estados dos processos, a concorrência no uso do processador e novamente compare percentualmente os tempos de uso do processador entre os dois processos.
- Libere o processo do estado de espera (suspenso) na janela *Gerência de Processos/Prosseguir*.
- Elimine um dos processos na janela *Gerência de Processos/Finalizar*.

c) Questão teórica para responder com a ajuda do simulador

Ao se eliminar um processo em estado de suspenso, o processo não é eliminado imediatamente. Reproduza essa situação no simulador e explique a razão da situação.

6

Thread

► 6.1 Introdução

Até o final da década de 1970, sistemas operacionais, como Tops-10 (DEC), MVS (IBM) e Unix (Bell Labs), suportavam apenas processos com um único thread (monothread), ou seja, um processo com apenas um único programa fazendo parte do seu contexto. Em 1979, durante o desenvolvimento do sistema operacional Toth, foi introduzido o conceito de processos lightweight (peso leve), onde o espaço de endereçamento de um processo era compartilhado por vários programas. Apesar do conceito revolucionário, a ideia não foi utilizada comercialmente e somente em meados de 1980, com o desenvolvimento do sistema operacional Mach, na Universidade de Carnegie Mellon, ficou clara a separação entre o conceito de processo e thread.

A partir do conceito de múltiplos threads (multithread) é possível projetar e implementar aplicações concorrentes de forma eficiente, pois um processo pode ter partes diferentes do seu código sendo executadas concorrentemente, com um menor overhead do que utilizando múltiplos processos. Como os threads de um mesmo processo compartilham o mesmo espaço de endereçamento, a comunicação entre threads não envolve mecanismos lentos de intercomunicação entre processos, aumentando, consequentemente, o desempenho da aplicação.

O desenvolvimento de programas que exploram os benefícios da programação multithread não é simples. A presença do paralelismo introduz um novo conjunto de problemas, como a comunicação e sincronização de threads. Existem diferentes modelos para a implementação de threads em um sistema operacional, onde desempenho, flexibilidade e custo devem ser avaliados.

Atualmente, o conceito de multithread pode ser encontrado em diversos sistemas operacionais, como no Sun Solaris e MS Windows. A utilização comercial de sistemas operacionais multithread é crescente em função do aumento de popularidade dos sistemas com múltiplos processadores, do modelo cliente-servidor e dos sistemas distribuídos.

► 6.2 Ambiente Monothread

Um programa é uma sequência de instruções, composta por desvios, repetições e chamadas a procedimentos e funções. Em um *ambiente monothread*, um processo suporta apenas um programa no seu espaço de endereçamento. Neste ambiente, aplicações concorrentes são implementadas apenas com o uso de múltiplos processos independentes ou subprocessos.

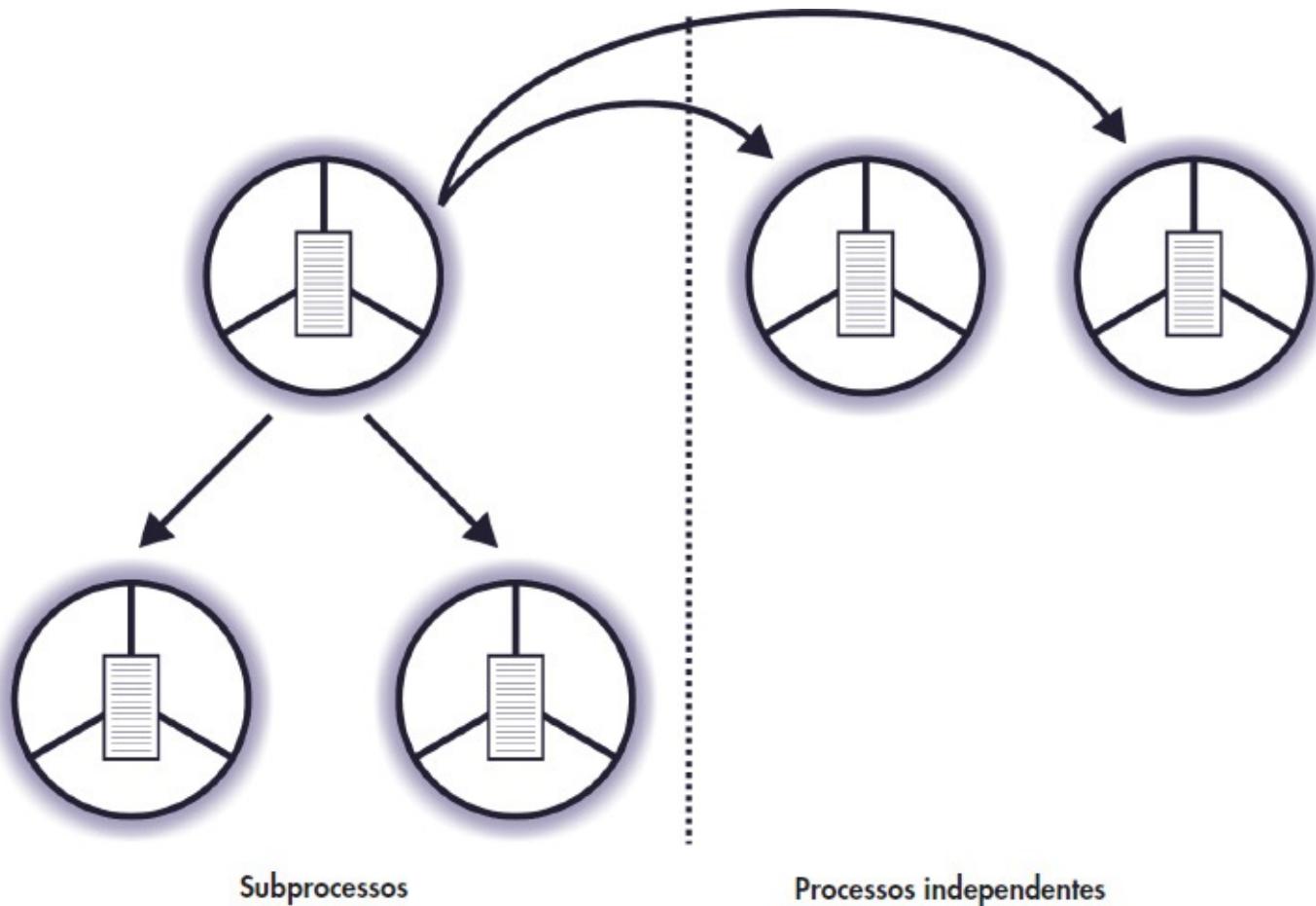


Fig. 6.1 Concorrência com subprocessos e processos independentes.

A utilização de processos independentes e subprocessos permite dividir uma aplicação em partes que podem trabalhar de forma concorrente. Um exemplo do uso de concorrência pode ser encontrado nas aplicações com interface gráfica, como em um software de gerenciamento de e-mails. Neste ambiente, um usuário pode estar lendo suas mensagens antigas, ao mesmo tempo que pode estar enviando e-mails e recebendo novas mensagens. Com o uso de múltiplos processos, cada funcionalidade do software implicaria a criação de um novo processo para atendê-la, aumentando o desempenho da aplicação (Fig. 6.1).

O problema neste tipo de implementação é que o uso de processos no desenvolvimento de aplicações concorrentes demanda consumo de diversos recursos do sistema. Sempre que um novo processo é criado, o sistema deve alojar recursos para cada processo, consumindo tempo de processador neste trabalho. No caso do término do processo, o sistema dispensa tempo para desalocar recursos previamente alocados.

Outro problema a ser considerado é quanto ao compartilhamento do espaço de endereçamento. Como cada processo possui seu próprio espaço de endereçamento, a comunicação entre processos torna-se

difícil e lenta, pois utiliza mecanismos como pipes, sinais, semáforos, memória compartilhada ou troca de mensagem. Além disso, o compartilhamento de recursos comuns aos processos concorrentes, como memória e arquivos abertos, não é simples. Na Fig. 6.2 existem três processos monothreads, cada um com seu próprio contexto de hardware, contexto de software e espaço de endereçamento.



Fig. 6.2 Ambiente monothread.

São exemplos de sistemas monothread o MS-DOS e as primeiras versões do MS Windows. Mesmo em ambientes multiprogramáveis e multiusuário, encontramos exemplos de implementações monothread, como nas versões mais antigas dos sistemas VAX/VMS e Unix.

► 6.3 Ambiente Multithread

Em um *ambiente multithread*, ou seja, com múltiplos threads, não existe a ideia de programas associados a processos, mas, sim, a threads. O processo, neste ambiente, tem pelo menos um thread de execução, mas pode compartilhar o seu espaço de endereçamento com inúmeros outros threads. Na Fig. 6.3 existe apenas um processo com três threads em execução compartilhando o mesmo espaço de endereçamento.

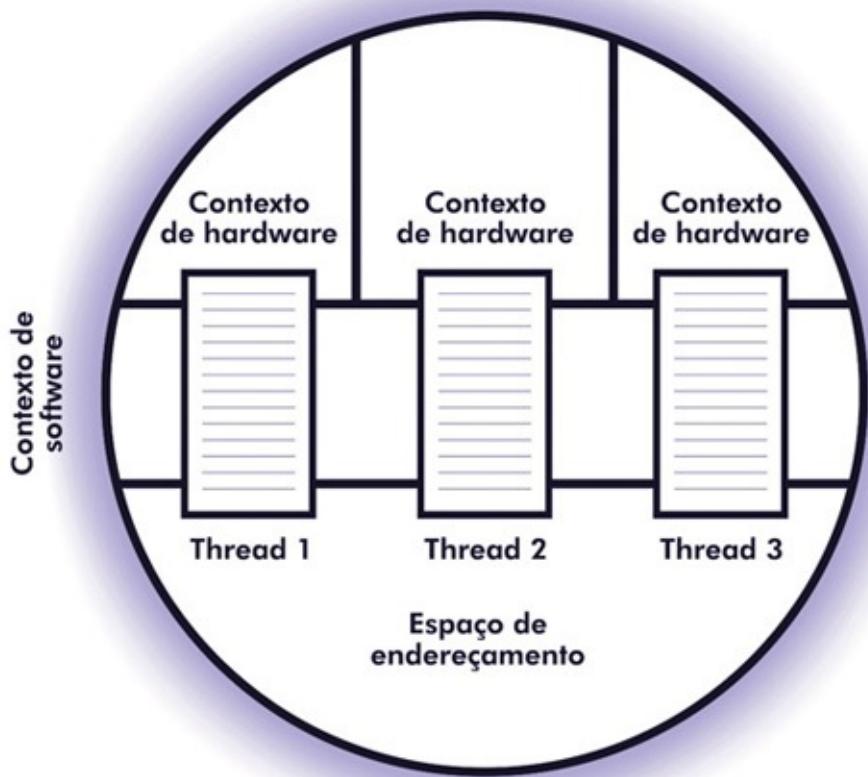


Fig. 6.3 Ambiente multithread.

De forma simplificada, um thread pode ser definido como uma sub-rotina de um programa que pode ser executada de forma assíncrona, ou seja, executada concorrentemente ao programa chamador. O programador deve especificar os threads, associando-os às sub-rotinas assíncronas. Desta forma, um ambiente multithread possibilita a execução concorrente de sub-rotinas dentro de um mesmo processo.

Na Fig. 6.4 existe um programa principal que realiza a chamada de duas sub-rotinas assíncronas (Sub_1 e Sub_2). Inicialmente, o processo é criado apenas com o Thread_1 para a execução do programa principal. Quando o programa principal chama as sub-rotinas Sub_1 e Sub_2, são criados os Thread_2 e Thread_3, respectivamente, e executados independentemente do programa principal. Neste processo, os três threads são executados concorrentemente.

No ambiente multithread, cada processo pode responder a várias solicitações concorrentemente ou mesmo simultaneamente, caso haja mais de um processador. A grande vantagem no uso de threads é a possibilidade de minimizar a alocação de recursos do sistema, além de diminuir o overhead na criação, troca e eliminação de processos.

Threads compartilham o processador da mesma maneira que processos e passam pelas mesmas mudanças de estado (execução, espera e pronto). Por exemplo, enquanto um thread espera por uma operação de E/S, outro thread pode ser executado. Para permitir a troca de contexto entre os diversos threads, cada thread possui seu próprio contexto de hardware, com o conteúdo dos registradores gerais, PC e SP. Quando um thread está sendo executado, seu contexto de hardware está armazenado nos registradores do processador. No momento em que o thread perde a utilização da UCP, as informações são atualizadas no seu contexto de hardware.

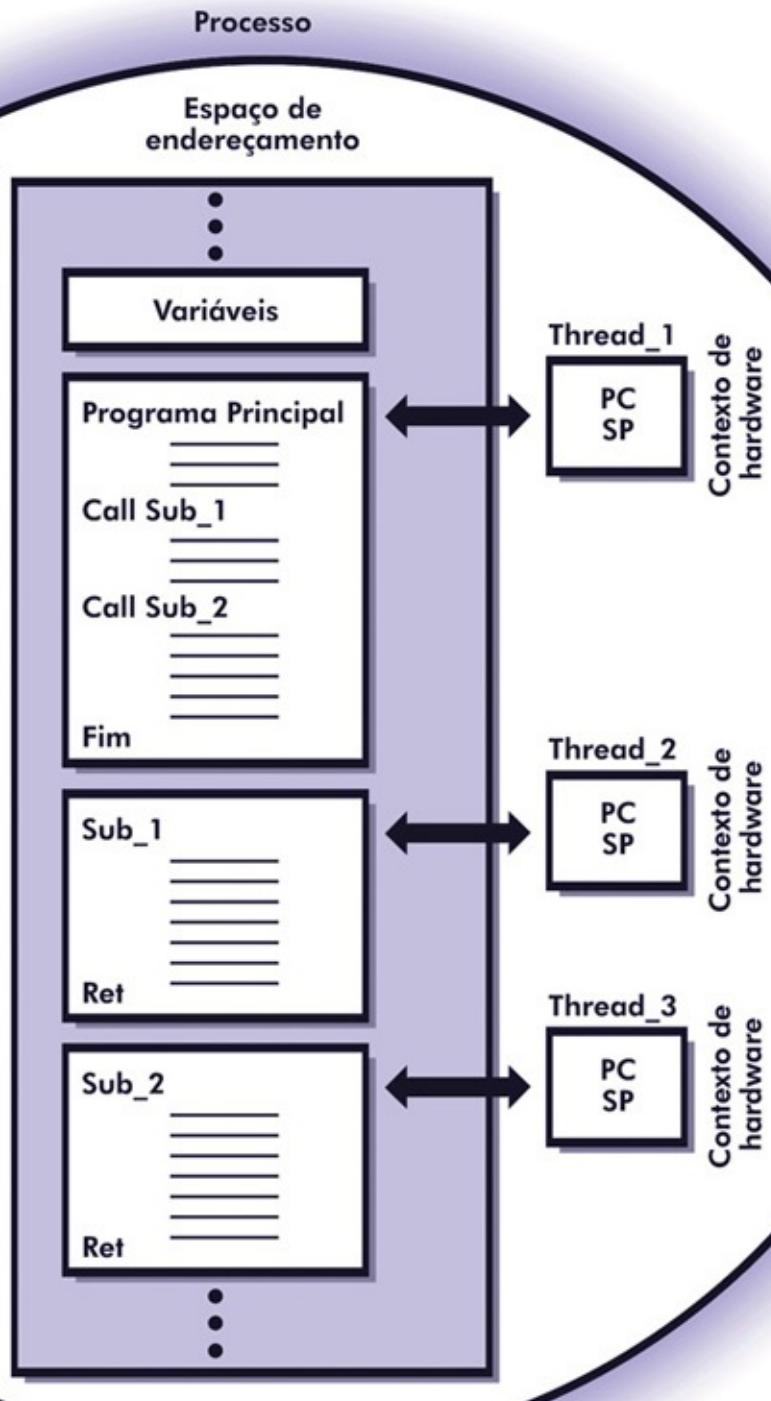


Fig. 6.4 Aplicação multithread.

Dentro de um mesmo processo, threads compartilham o mesmo contexto de software e espaço de endereçamento com os demais threads, porém cada thread possui seu contexto de hardware individual. Threads são implementados internamente por meio de uma estrutura de dados denominada *bloco de controle do thread* (*Thread Control Block – TCB*). O TCB armazena, além do contexto de hardware, mais algumas informações relacionadas exclusivamente ao thread, como prioridade, estado de execução e bits de estado.

Em ambientes monothread, o processo é ao mesmo tempo a unidade de alocação de recursos e a unidade de escalonamento. A independência entre os conceitos de processo e thread permite separar a unidade de

alocação de recursos da unidade de escalonamento, que em ambientes monothread estão fortemente relacionadas. Em um ambiente multithread, a unidade de alocação de recursos é o processo onde todos os seus threads compartilham o espaço de endereçamento, descritores de arquivos e dispositivos de E/S. Por outro lado, cada thread representa uma unidade de escalonamento independente e, neste caso, o sistema não seleciona um processo para a execução, mas sim um de seus threads.

A grande diferença entre aplicações monothread e multithread está no uso do espaço de endereçamento. Processos independentes e subprocessos possuem espaços de endereçamento individuais e protegidos, enquanto threads compartilham o espaço dentro de um mesmo processo. Esta característica permite que o compartilhamento de dados entre threads de um mesmo processo seja mais simples e rápida, se comparado a ambientes monothread.

Como threads de um mesmo processo compartilham o mesmo espaço de endereçamento, não existe qualquer proteção no acesso à memória, permitindo que um thread possa alterar facilmente dados de outros. Para que threads trabalhem de forma cooperativa, é fundamental que a aplicação implemente mecanismos de comunicação e sincronização entre threads, a fim de garantir o acesso seguro aos dados compartilhados na memória.

O uso de multithreads proporciona uma série de benefícios. Programas concorrentes com múltiplos threads são mais rápidos do que implementados com múltiplos processos, pois operações de criação, troca de contexto e eliminação dos threads geram menor overhead (Tabela 6.1). Como os threads dentro de um processo dividem o mesmo espaço de endereçamento, a comunicação entre eles pode ser realizada de forma rápida e eficiente. Além disso, threads em um mesmo processo podem compartilhar facilmente outros recursos, como descritores de arquivos, temporizadores, sinais, atributos de segurança etc.

Tabela 6.1 Latência de processos e threads (Vahalia, 1996)

Implementação	Tempo de criação (μs)	Tempo de sincronização (μs)
Processo	1700	200
Processo Lightweight	350	390
Thread	52	66

A utilização do processador, discos e outros periféricos pode ser feita de forma concorrente pelos diversos threads, significando uma melhor utilização dos recursos computacionais disponíveis. Em algumas aplicações, a utilização de threads pode melhorar o desempenho da aplicação apenas executando tarefas em background, enquanto operações E/S estão sendo processadas (Fig. 6.5). Aplicações como editores de texto, planilhas, aplicativos gráficos e processadores de imagens são especialmente beneficiadas quando desenvolvidas com base em threads.

Em ambientes cliente-servidor, threads são essenciais para solicitações de serviços remotos. Em um ambiente monothread, se uma aplicação solicita um serviço remoto ela pode ficar esperando indefinidamente, enquanto aguarda pelo resultado. Em um ambiente multithread, um thread pode solicitar o serviço remoto, enquanto a aplicação pode continuar realizando outras atividades. Já para o processo

que atende a solicitação, múltiplos threads permitem que diversos pedidos sejam atendidos simultaneamente (Fig. 6.6).

Não apenas aplicações tradicionais podem fazer uso dos benefícios do multithreading. O núcleo do sistema operacional também pode ser implementado com o uso desta técnica de forma vantajosa, como na arquitetura microkernel, apresentada no Cap. 4 — Estrutura do Sistema Operacional.

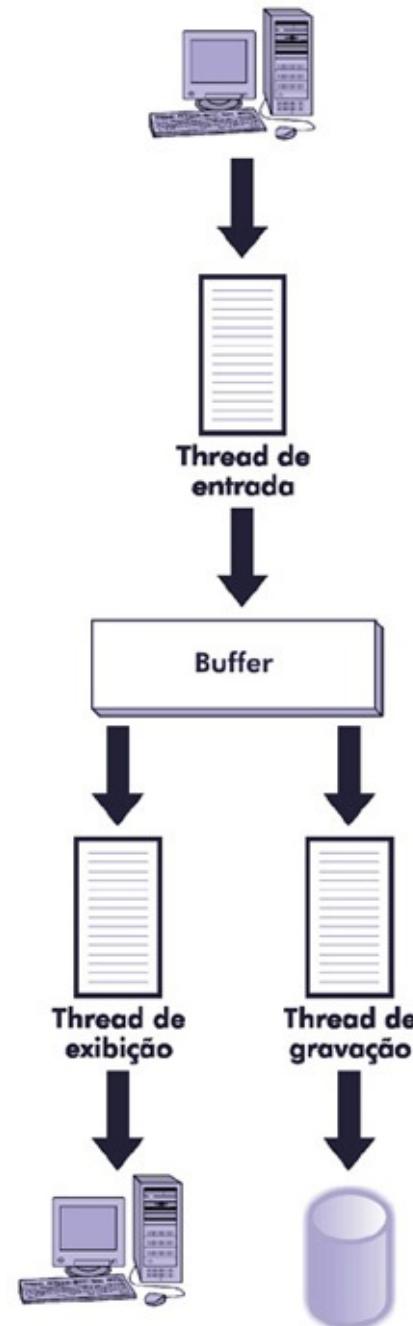


Fig. 6.5 Aplicação multithread (a).

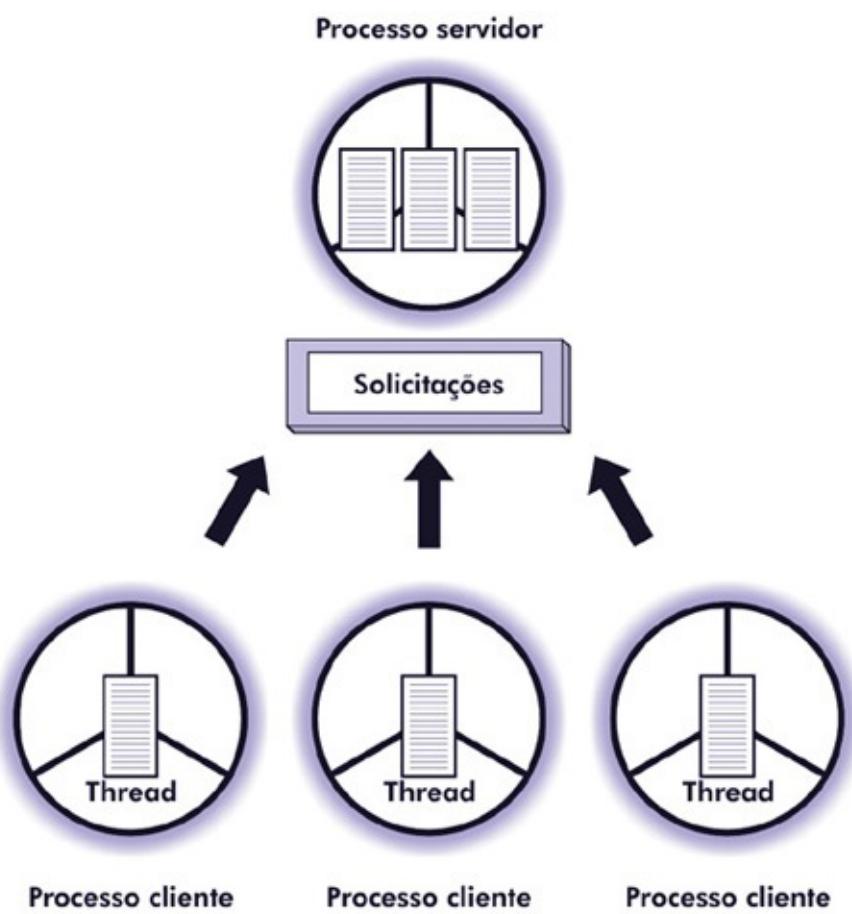


Fig. 6.6 Aplicação multithread (b).

► 6.4 Programação Multithread

O programa CriaThreads, escrito em Java, cria três threads a partir da classe Loop. Para cada thread criado, o programa exibe uma mensagem “Thread x criado”, em que x pode ser 1, 2 ou 3, dependendo do thread criado. Cada thread executa uma iteração infinita, exibindo uma mensagem “Thread x executado”, em que x pode ser 1, 2 ou 3, dependendo do thread executado. Em seguida, o próprio thread se coloca no estado de espera por um tempo aleatório entre zero e cinco segundos e depois volta ao estado de pronto para ser escalonado.

```

import java.util.*;
public class CriaThreads {
    public static void main(String [] args) {
        int i, n = 3;

        for(i = 1; i <= n; i++) {
            Loop loop = new Loop(i);
            Thread t = new Thread(loop);
            t.start();
            System.out.println("Thread " + i + " criado");
        }
    }

    class Loop implements Runnable {

```

```

int j;

public Loop(int i) {
j = i;
}
public void run() {
Random random = new Random();
while (true) {
System.out.println("Thread " + j + " executado");
try {
Thread.sleep(random.nextInt(5000));
} catch (InterruptedException iex) {}
}
}
}

```

A Fig. 6.7 apresenta uma possível saída para a execução do programa CriaThreads. Como pode ser observado, os threads 1 e 2 são criados, executados e apenas depois o thread 3 é criado e executado. Após a criação dos três threads, não é possível determinar qual o thread será selecionado para a execução. O escalonamento dos threads é realizado pela máquina virtual Java, independentemente da ordem em que foram criados.

```

Thread 1 criado
Thread 2 criado
Thread 1 executado
Thread 2 executado
Thread 3 criado
Thread 3 executado
Thread 3 executado
Thread 1 executado
Thread 2 executado

```

Fig. 6.7 Programa Multithread.

O desenvolvimento de aplicações multithread não é simples, pois exige que a comunicação e o compartilhamento de recursos entre os diversos threads sejam feitos de forma sincronizada para evitar problemas de inconsistências e deadlock. Além das dificuldades naturais no desenvolvimento de aplicações concorrentes, o procedimento de depuração é bastante complexo.

Um fator importante em aplicações multithread é o número total de threads e a forma como são criados e eliminados. Se uma aplicação cria um número excessivo de threads poderá ocorrer um overhead no sistema, ocasionando uma queda de desempenho. Dependendo da implementação, a definição do número de threads pode ser dinâmica ou estática. Quando a criação/eliminação é dinâmica, os threads são criados/eliminados conforme a demanda da aplicação, oferecendo grande flexibilidade. Já em ambientes estáticos, o número de threads é definido na criação do processo onde a aplicação será executada.

Para obter os benefícios do uso de threads, uma aplicação deve permitir que partes diferentes do seu código sejam executadas concorrentemente de forma independente. Se um aplicativo realiza várias operações de E/S e trata eventos assíncronos, a programação multithread aumenta seu desempenho até

mesmo em ambientes com um único processador. Sistemas gerenciadores de banco de dados, servidores de arquivos ou impressão, servidores web e de correio são exemplos onde o uso de múltiplos threads proporciona grandes vantagens e benefícios.

► 6.5 Arquitetura e Implementação

Existem diferentes abordagens na implementação dos pacotes de threads em um sistema operacional, o que influenciará no desempenho, na concorrência e na modularidade das aplicações multithread. Threads podem ser oferecidos por uma biblioteca de rotinas fora do núcleo do sistema operacional (modo usuário), pelo próprio núcleo do sistema (modo kernel), uma combinação de ambos (modo híbrido) ou por um modelo conhecido como scheduler activations. A Tabela 6.2 sumariza as diversas arquiteturas para diferentes ambientes operacionais.

Tabela 6.2 Arquitetura de threads para diversos ambientes operacionais

Ambientes	Arquitetura
Distributed Computing Environment (DCE)	Modo usuário
Compaq OpenVMS versão 6	Modo usuário
Microsoft Windows 2000	Modo kernel
Compaq Unix	Modo kernel
Compaq OpenVMS versão 7	Modo kernel
Sun Solaris versão 2	Modo híbrido
University of Washington FastThreads	Scheduler activations

Uma das grandes dificuldades para a utilização de threads foi a ausência de um padrão. Em 1995, o padrão POSIX P1003.1c foi aprovado e posteriormente atualizado para a versão POSIX 1003.4a. Com este padrão, também conhecido como Pthreads, aplicações comerciais multithreading tornaram-se mais simples e de fácil implementação. O padrão Pthreads é largamente utilizado em ambientes Unix, como o Sun Solaris Pthreads e o DECthreads para Digital OSF/1.

6.5.1 Threads em Modo Usuário

Threads em modo usuário (TMU) são implementados pela aplicação e não pelo sistema operacional. Para isso, deve existir uma biblioteca de rotinas que possibilite à aplicação realizar tarefas como criação/eliminação de threads, troca de mensagens entre threads e uma política de escalonamento. Neste modo, o sistema operacional não sabe da existência de múltiplos threads, sendo responsabilidade exclusiva da aplicação gerenciar e sincronizar os diversos threads existentes.

A vantagem deste modelo é a possibilidade de implementar aplicações multithreads mesmo em sistemas

operacionais que não suportam threads. Utilizando a biblioteca, múltiplos threads podem ser criados, compartilhando o mesmo espaço de endereçamento do processo, além de outros recursos (Fig. 6.8). TMU são rápidos e eficientes por dispensarem acessos ao kernel do sistema operacional, evitando assim a mudança de modo de acesso (usuário-kernel-usuário).

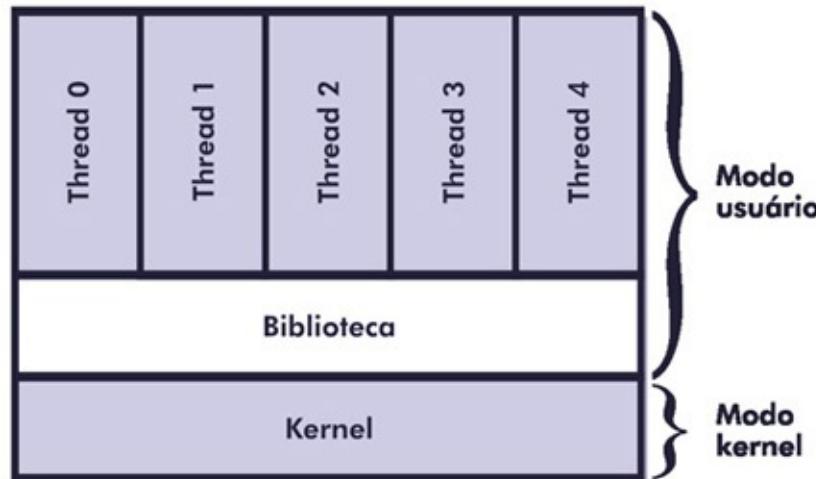


Fig. 6.8 Threads em modo usuário.

TMU possuem uma grande limitação, pois o sistema operacional gerencia cada processo como se existisse apenas um único thread. No momento em que um thread chama uma rotina do sistema que o coloca em estado de espera (rotina bloqueante), todo o processo é colocado no estado de espera, mesmo havendo outros threads prontos para execução. Para contornar esta limitação, a biblioteca tem que possuir rotinas que substituam as rotinas bloqueantes por outras que não possam causar o bloqueio de um thread (rotinas não bloqueantes). Todo este controle é transparente para o usuário e para o sistema operacional.

Talvez um dos maiores problemas na implementação de TMU seja o tratamento individual de sinais. Como o sistema reconhece apenas processos e não threads, os sinais enviados para um processo devem ser reconhecidos e encaminhados a cada thread para tratamento. No caso do recebimento de interrupções de clock, fundamental para a implementação do tempo compartilhado, esta limitação é crítica. Neste caso, os sinais de temporização devem ser interceptados, para que se possa interromper o thread em execução e realizar a troca de contexto.

Em relação ao escalonamento em ambientes com múltiplos processadores, não é possível que múltiplos threads de um processo possam ser executados em diferentes UCPs simultaneamente, pois o sistema seleciona apenas processos para execução e não threads. Esta restrição limita drasticamente o grau de paralelismo da aplicação, já que os threads de um mesmo processo podem ser executados em somente um processador de cada vez.

6.5.2 Threads em Modo Kernel

Threads em modo kernel (TMK) são implementados diretamente pelo núcleo do sistema operacional, através de chamadas a rotinas do sistema que oferecem todas as funções de gerenciamento e

sincronização (Fig. 6.9). O sistema operacional sabe da existência de cada thread e pode escaloná-los individualmente. No caso de múltiplos processadores, os threads de um mesmo processo podem ser executados simultaneamente.



Fig. 6.9 Threads em modo kernel.

O grande problema para pacotes em modo kernel é o seu baixo desempenho. Enquanto nos pacotes em modo usuário todo tratamento é feito sem a ajuda do sistema operacional, ou seja, sem a mudança do modo de acesso (usuário-kernel-usuário), pacotes em modo kernel utilizam chamadas a rotinas do sistema e, consequentemente, várias mudanças no modo de acesso. A Tabela 6.3 compara o desempenho de duas operações distintas envolvendo criação, escalonamento, execução e eliminação de um processo/thread.

Tabela 6.3 Comparação entre tempos de latência (Anderson et al., 1992)

Implementação	Operação 1 (μs)	Operação 2 (μs)
Subprocessos	11.300	1.840
Threads em Modo Kernel	948	441
Threads em Modo Usuário	34	37

6.5.3 Threads em Modo Híbrido

A arquitetura de *threads em modo híbrido* combina as vantagens de threads implementados em modo usuário (TMU) e modo kernel (TMK). Um processo pode ter vários TMKs e, por sua vez, um TMK pode ter vários TMUs. O núcleo do sistema reconhece os TMKs e pode escaloná-los individualmente. Um TMU pode ser executado em um TMK, em um determinado momento, e no instante seguinte ser executado em outro.

O programador desenvolve a aplicação em termos de TMUs e especifica quantos TMKs estão associados ao processo. Os TMU são mapeados em TMK enquanto o processo está sendo executado. O programador pode utilizar apenas TMKs, TMUs ou uma combinação de ambos (Fig. 6.10).

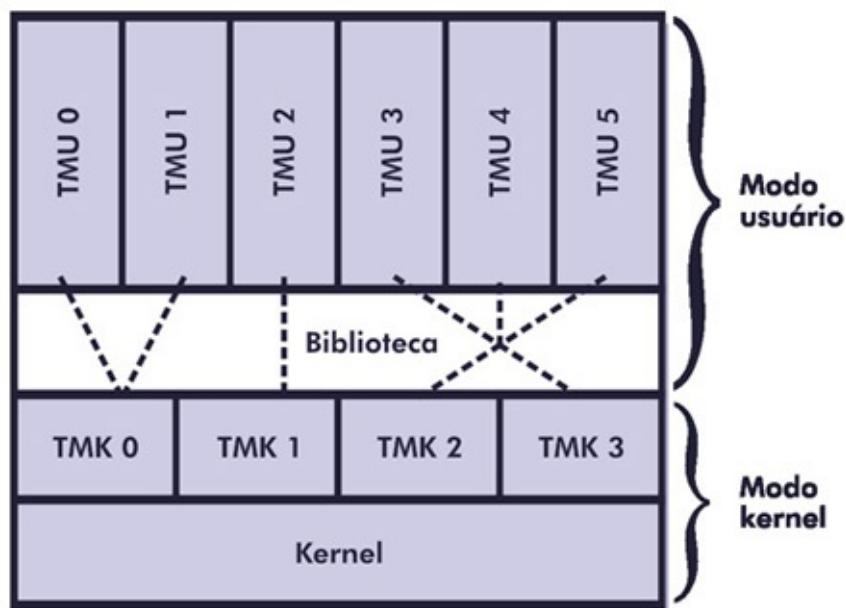


Fig. 6.10 Threads em modo híbrido.

O modo híbrido, apesar da maior flexibilidade, apresenta problemas herdados de ambas as implementações. Por exemplo, quando um TMK realiza uma chamada bloqueante, todos os TMUs são colocados no estado de espera. TMUs que desejam utilizar vários processadores devem utilizar diferentes TMKs, o que influenciará no desempenho.

6.5.4 Scheduler Activations

Os problemas apresentados no pacote de threads em modo híbrido existem devido à falta de comunicação entre os threads em modo usuário e modo kernel. O modelo ideal deveria utilizar as facilidades do pacote em modo kernel com o desempenho e a flexibilidade do modo usuário.

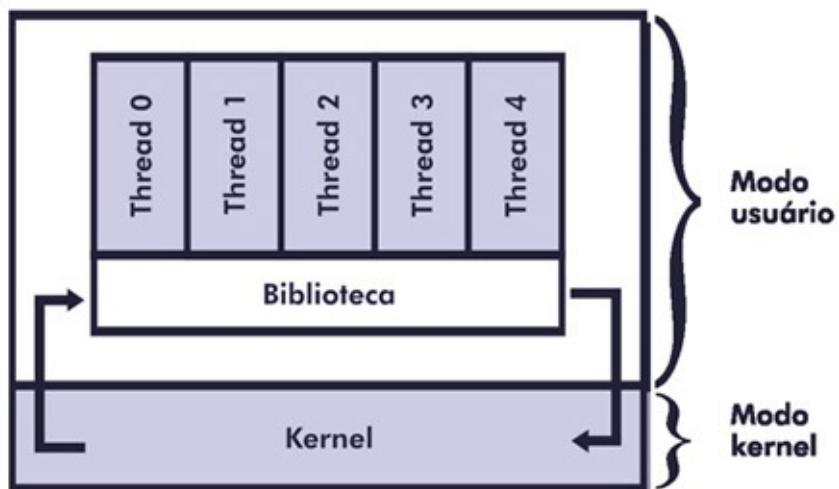


Fig. 6.11 Scheduler activations.

Introduzido no início da década de 1990 na Universidade de Washington, este pacote combina o melhor das duas arquiteturas, mas ao contrário de dividir os threads em modo usuário entre os de modo kernel o núcleo do sistema troca informações com a biblioteca de threads utilizando uma estrutura de dados

chamada *scheduler activations* (Fig. 6.11).

A maneira de alcançar um melhor desempenho é evitar as mudanças de modos de acesso desnecessárias (usuário-kernel-usuário). Caso um thread utilize uma chamada ao sistema que o coloque no estado de espera, não é necessário que o kernel seja ativado, bastando que a própria biblioteca em modo usuário escalone outro thread. Isto é possível porque a biblioteca em modo usuário e o kernel se comunicam e trabalham de forma cooperativa. Cada camada implementa seu escalonamento de forma independente, porém trocando informações quando necessário.

► 6.6 Exercícios

1. Como uma aplicação pode implementar concorrência em um ambiente monothread?
2. Quais os problemas de aplicações concorrentes desenvolvidas em ambientes monothread?
3. O que é um ambiente multithread e quais as vantagens de sua utilização?
4. Explique a diferença entre unidade de alocação de recursos e unidade de escalonamento.
5. Quais as vantagens e desvantagens do compartilhamento do espaço de endereçamento entre threads de um mesmo processo?
6. No programa multithread do item 6.4, qual dos três threads será executado primeiro, depois de criados?
7. No programa multithread do item 6.4, é possível que o thread 3 seja executado mais que três vezes seguidas?
8. Altere o programa do item 6.4 de forma a criar mais um thread, totalizando uma aplicação concorrente com quatro threads.
9. Compare os pacotes de threads em modo usuário e modo kernel.
10. Qual a vantagem do *scheduler activations* comparado ao pacote híbrido?
11. Dê exemplos do uso de threads no desenvolvimento de aplicativos, como editores de textos e planilhas eletrônicas.
12. Como o uso de threads pode melhorar o desempenho de aplicações paralelas em ambientes com múltiplos processadores?
13. Quais os benefícios do uso de threads em ambientes cliente-servidor?
14. Como o uso de threads pode ser útil em arquiteturas microkernel?



Sincronização e Comunicação entre Processos

► 7.1 Introdução

Na década de 1960, com o surgimento dos sistemas multiprogramáveis, passou a ser possível estruturar aplicações de maneira que partes diferentes do código do programa pudessem executar concorrentemente. Este tipo de aplicação, denominado *aplicação concorrente*, tem como base a execução cooperativa de múltiplos processos ou threads, que trabalham em uma mesma tarefa na busca de um resultado comum.

Em um sistema multiprogramável com único processador, os processos alternam sua execução segundo critérios de escalonamento estabelecidos pelo sistema operacional. Mesmo não havendo neste tipo de sistema um paralelismo na execução de instruções, uma aplicação concorrente pode obter melhorias no seu desempenho. Em sistemas com múltiplos processadores, a possibilidade do paralelismo na execução de instruções somente estende as vantagens que a programação concorrente proporciona.

É natural que processos de uma aplicação concorrente compartilhem recursos do sistema, como arquivos, registros, dispositivos de E/S e áreas de memória. O compartilhamento de recursos entre processos pode ocasionar situações indesejáveis, capazes até de comprometer a execução das aplicações. Para evitar esse tipo de problema, os processos concorrentes devem ter suas execuções sincronizadas, a partir de mecanismos oferecidos pelo sistema operacional, com o objetivo de garantir o processamento correto dos programas.

Este capítulo apresenta como a concorrência de processos pode ser implementada, os problemas do compartilhamento de recursos, soluções e mecanismos do sistema operacional para sincronizar processos como semáforos e monitores. No final do capítulo é também apresentado o problema do deadlock, suas consequências e análise de soluções.

► 7.2 Aplicações Concorrentes

Muitas vezes, em uma aplicação concorrente, é necessário que processos se comuniquem entre si. Esta comunicação pode ser implementada através de diversos mecanismos, como variáveis compartilhadas na memória principal ou trocas de mensagens. Nesta situação, é necessário que os processos concorrentes tenham sua execução sincronizada através de mecanismos do sistema operacional.

A Fig. 7.1 apresenta um exemplo onde dois processos concorrentes compartilham um buffer para trocar informações através de operações de gravação e leitura. Neste exemplo, um processo só poderá gravar dados no buffer caso este não esteja cheio. Da mesma forma, um processo só poderá ler dados armazenados do buffer caso exista algum dado para ser lido. Em ambas as situações, os processos deverão aguardar até que o buffer esteja pronto para as operações, seja de gravação, seja de leitura.

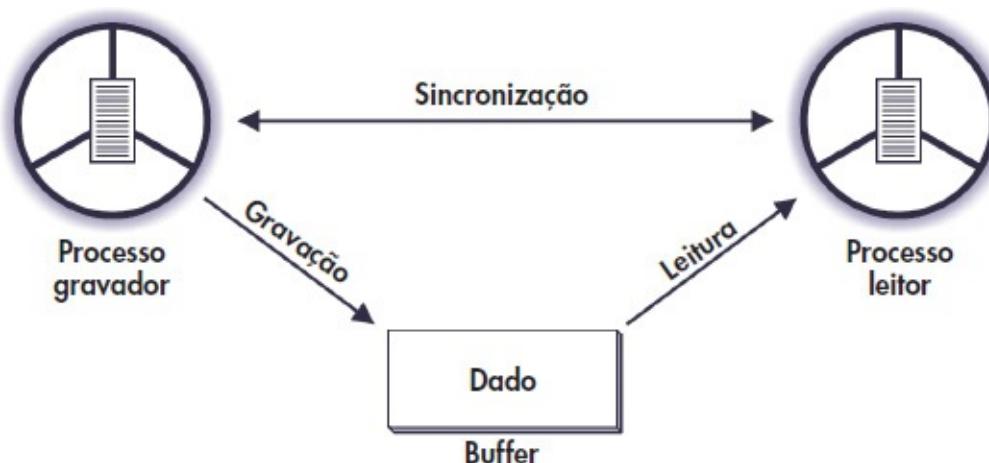


Fig. 7.1 Sincronização e comunicação entre processos.

Os mecanismos que garantem a comunicação entre processos concorrentes e o acesso a recursos compartilhados são chamados *mecanismos de sincronização*. No projeto de sistemas operacionais multiprogramáveis, é fundamental a implementação destes mecanismos para garantir a integridade e a confiabilidade na execução de aplicações concorrentes.

Apesar de o termo processo ser utilizado na exemplificação de aplicações concorrentes, os termos subprocesso e thread têm o mesmo significado nesta abordagem.

► 7.3 Especificação de Concorrência em Programas

Existem várias notações utilizadas para especificar a concorrência em programas, ou seja, as partes de um programa que devem ser executadas concorrentemente. As técnicas mais recentes tentam expressar a concorrência no código dos programas de uma forma mais clara e estruturada.

A primeira notação para a especificação da concorrência em um programa foram os comandos FORK e JOIN, introduzidos por Conway (1963) e Dennis e Van Horn (1966). O exemplo a seguir apresenta a implementação da concorrência em um programa com uma sintaxe simplificada:

```
PROGRAM A; PROGRAM B;
```

```
..
```

```
..
```

```

FORK B; .
.
.
JOIN B;END.
.
.
END.

```

O programa A começa a ser executado e, ao encontrar o comando FORK, faz com que seja criado um outro processo para a execução do programa B, concorrentemente ao programa A. O comando JOIN permite que o programa A sincronize-se com B, ou seja, quando o programa A encontrar o comando JOIN só continuará a ser processado após o término da execução do programa B. Os comandos FORK e JOIN são bastante poderosos e práticos, sendo utilizados de forma semelhante no sistema operacional Unix.

Uma das implementações mais claras e simples de expressar concorrência em um programa é a utilização dos comandos PARBEGIN e PAREND (Dijkstra, 1965a), que, posteriormente, foram chamados de COBEGIN e COEND. No decorrer deste capítulo, os comandos PARBEGIN e PAREND serão utilizados nos algoritmos apresentados.

O comando PARBEGIN especifica que a sequência de comandos seja executada concorrentemente em uma ordem imprevisível, através da criação de um processo (Processo_1, Processo_2, Processo_n) para cada comando (Comando_1, Comando_2, Comando_n). O comando PAREND define um ponto de sincronização, onde o processamento só continuará quando todos os processos ou threads criados já tiverem terminado suas execuções. Os comandos delimitados pelos comandos PARBEGIN e PAREND podem ser comandos simples, como atribuições ou chamadas a procedimentos (Fig. 7.2).

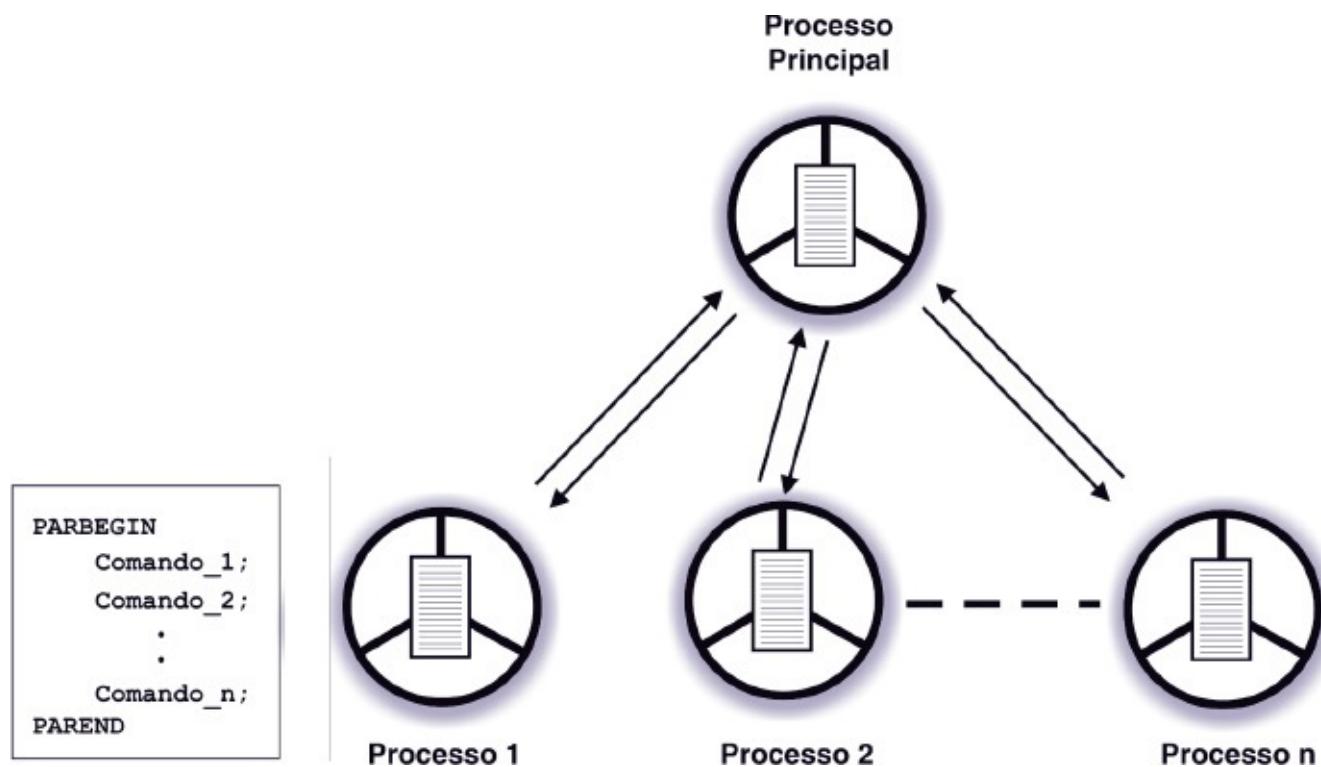


Fig. 7.2 Concorrência em programas.

Para exemplificar o funcionamento dos comandos PARBEGIN e PAREND em uma aplicação

concorrente, o programa Expressao realiza o cálculo do valor da expressão aritmética descrita a seguir:

```
X := SQRT (1024) + (35.4 * 0.23) - (302 / 7)
```

Os comandos de atribuição situados entre PARBEGIN e PARENTESES são executados concorrentemente. O cálculo final de X só pode ser realizado quando todas as variáveis dentro da estrutura tiverem sido calculadas.

```
PROGRAM Expressao;
VAR X, Temp1, Temp2, Temp3 : REAL;
BEGIN
PARBEGIN
Temp1 := SQRT (1024);
Temp2 := 35.4 * 0.23;
Temp3 := 302 / 7;
PARENTESES;
X := Temp1 + Temp2 - Temp3;
WRITELN ('x = ', X);
END.
```

► 7.4 Problemas de Compartilhamento de Recursos

Para a compreensão de como a sincronização entre processos concorrentes é fundamental para a confiabilidade dos sistemas multiprogramáveis, são apresentados alguns problemas de compartilhamento de recursos. A primeira situação envolve o compartilhamento de um arquivo em disco; a segunda apresenta uma variável na memória principal sendo compartilhada por dois processos.

O primeiro problema é analisado a partir do programa Conta_Corrente, que atualiza o saldo bancário de um cliente após um lançamento de débito ou crédito no arquivo de contas-correntes Arq_Contas. Neste arquivo são armazenados os saldos de todos os correntistas do banco. O programa lê o registro do cliente no arquivo (Reg_Cliente), lê o valor a ser depositado ou retirado (Valor_Dep_Ret) e, em seguida, atualiza o saldo no arquivo de contas.

```
PROGRAM Conta_Corrente;
.
.
.
READ (Arq_Contas, Reg_Cliente);
READLN (Valor_Dep_Ret);
Reg_Cliente.Saldo := Reg_Cliente.Saldo + Valor_Dep_Ret;
WRITE (Arq_Contas, Reg_Cliente);
.
.
.
END.
```

Considerando processos concorrentes pertencentes a dois funcionários do banco que atualizam o saldo de um mesmo cliente simultaneamente, a situação de compartilhamento do recurso pode ser analisada no exemplo da Tabela 7.1. O processo do primeiro funcionário (Caixa 1) lê o registro do cliente e soma ao campo Saldo o valor do lançamento de débito. Antes de gravar o novo saldo no arquivo, o processo do

segundo funcionário (Caixa 2) lê o registro do mesmo cliente, que está sendo atualizado, para realizar outro lançamento, desta vez de crédito. Independentemente de qual dos processos atualize primeiro o saldo no arquivo, o dado gravado estará inconsistente.

Tabela 7.1 Problema de Concorrência I

Caixa	Instrução	Saldo arquivo	Valor dep/ret	Saldo memória
1	READ	1.000	*	1.000
1	READLN	1.000	-200	1.000
1	:=	1.000	-200	800
2	READ	1.000	*	1.000
2	READLN	1.000	+300	1.000
2	:=	1.000	+300	1.300
1	WRITE	800	-200	800
2	WRITE	1.300	+300	1.300

Outro exemplo, ainda mais simples, onde o problema da concorrência pode levar a resultados inesperados é a situação onde dois processos (A e B) executam um comando de atribuição. O Processo A soma 1 à variável X e o Processo B diminui 1 da mesma variável que está sendo compartilhada. Inicialmente, a variável X possui o valor 2.

Processo A

X := X + 1;

Processo B

X := X - 1;

Seria razoável pensar que o resultado final da variável X, após a execução dos Processos A e B, continuasse 2, porém isto nem sempre será verdade. Os comandos de atribuição, em uma linguagem de alto nível, podem ser decompostos em comandos mais elementares, como visto a seguir:

Processo A

LOAD x, R_a

ADD1, R_a

STORE R_a, x

Processo B

LOAD x, R_b

SUB1, R_b

STORE R_b, x

Utilizando o exemplo da Tabela 7.2, considere que o Processo A carregue o valor de X no registrador R_a, some 1 e, no momento em que vai armazenar o valor em X, seja interrompido. Nesse instante, o Processo B inicia sua execução, carrega o valor de X em R_b e subtrai 1. Dessa vez, o Processo B é interrompido e o Processo A volta a ser processado, atribuindo o valor 3 à variável X e concluindo sua execução.

Finalmente, o Processo B retorna a execução, atribui o valor 1 a X, e sobrepõe o valor anteriormente gravado pelo Processo A. O valor final da variável X é inconsistente em função da forma concorrente com que os dois processos executaram.

Tabela 7.2 Problema de Concorrência II

Processo	Instrução	X	R _a	R _b
A	LOAD X, R _a	2	2	*
A	ADD 1, R _a	2	3	*
B	LOAD X, R _b	2	*	2
B	SUB 1, R _b	2	*	1
A	STORE R _a , X	3	3	*
B	STORE R _b , X	1	*	1

Analisando os dois exemplos apresentados, é possível concluir que em qualquer situação, onde dois ou mais processos compartilham um mesmo recurso, devem existir mecanismos de controle para evitar esses tipos de problemas, conhecidos como *condições de corrida (race conditions)*.

► 7.5 Exclusão Mútua

A solução mais simples para evitar os problemas de compartilhamento apresentados no item anterior é impedir que dois ou mais processos acessem um mesmo recurso simultaneamente. Para isso, enquanto um processo estiver acessando determinado recurso, todos os demais processos que queiram acessá-lo deverão esperar pelo término da utilização do recurso. Essa ideia de exclusividade de acesso é chamada *exclusão mútua (mutual exclusion)*.

A exclusão mútua deve afetar apenas os processos concorrentes somente quando um deles estiver fazendo acesso ao recurso compartilhado. A parte do código do programa onde é feito o acesso ao recurso compartilhado é denominada *região crítica (critical region)*. Se for possível evitar que dois processos entrem em suas regiões críticas ao mesmo tempo, ou seja, se for garantida a execução mutuamente exclusiva das regiões críticas, os problemas decorrentes do compartilhamento serão evitados.

Os mecanismos que implementam a exclusão mútua utilizam protocolos de acesso à região crítica. Toda vez que um processo desejar executar instruções de sua região crítica, obrigatoriamente deverá executar antes um protocolo de entrada nessa região. Da mesma forma, ao sair da região crítica um protocolo de saída deverá ser executado. Os protocolos de entrada e saída garantem a exclusão mútua da região crítica de um programa.

BEGIN

.

.

```
Entra_Regiao_Critica; (* Protocolo de Entrada *)
Regiao_Critica;
Sai_Regiao_Critica; (* Protocolo de Saída *)
.
.
END.
```

Utilizando o programa Conta_Corrente apresentado no item anterior, a aplicação dos protocolos para dois processos (A e B) pode ser implementada no compartilhamento do arquivo de contas-correntes. Sempre que o Processo A for atualizar o saldo de um cliente, antes de ler o registro o acesso exclusivo ao arquivo deve ser garantido através do protocolo de entrada da sua região crítica. O protocolo indica se já existe ou não algum processo acessando o recurso. Caso o recurso esteja livre, o Processo A pode entrar em sua região crítica para realizar a atualização. Durante este período, caso o Processo B tente acessar o arquivo, o protocolo de entrada faz com que esse processo permaneça aguardando até que o Processo A termine o acesso ao recurso. Quando o Processo A terminar a execução de sua região crítica, deve sinalizar aos demais processos concorrentes que o acesso ao recurso foi concluído. Isso é realizado através do protocolo de saída, que informa aos outros processos que o recurso já está livre e pode ser utilizado de maneira exclusiva por um outro processo.

Como é possível, então, observar, para garantir a implementação da exclusão mútua os processos envolvidos devem fazer acesso aos recursos de forma sincronizada. Diversas soluções foram desenvolvidas com esse propósito; porém, além da garantia da exclusão mútua, duas situações indesejadas também devem ser evitadas.

A primeira situação indesejada é conhecida como *starvation* ou *espera indefinida*. Starvation é a situação em que um processo nunca consegue executar sua região crítica e, consequentemente, acessar o recurso compartilhado. No momento em que um recurso alocado é liberado, o sistema operacional possui um critério para selecionar, dentre os processos que aguardam pelo uso do recurso, qual será o escolhido. Em função do critério de escolha, o problema do starvation pode ocorrer. Os critérios de escolha aleatória ou com base em prioridades são exemplos de situações que podem gerar o problema. No primeiro caso, não é garantido que um processo em algum momento fará o uso do recurso, pois a seleção é randômica. No segundo caso, a baixa prioridade de um processo em relação aos demais pode impedir o processo de acessar o recurso compartilhado. Uma solução bastante simples para o problema é a criação de filas de pedidos de alocação para cada recurso, utilizando o esquema FIFO. Sempre que um processo solicita um recurso, o pedido é colocado no final da fila associada ao recurso. Quando o recurso é liberado, o sistema seleciona o primeiro processo da fila.

Outra situação indesejada na implementação da exclusão mútua é aquela em que um processo fora da sua região crítica impede que outros processos entrem nas suas próprias regiões críticas. No caso de esta situação ocorrer, um recurso estaria livre, porém alocado a um processo. Com isso, vários processos estariam sendo impedidos de utilizar o recurso, reduzindo o grau de compartilhamento.

Diversas soluções foram propostas para garantir a exclusão mútua de processos concorrentes. A seguir, são apresentadas algumas soluções de hardware e de software com discussões sobre benefícios e problemas de cada proposta.

7.5.1 Soluções de Hardware

A exclusão mútua pode ser implementada através de mecanismos de hardware. Neste item são apresentadas as soluções de desabilitação de interrupções e instruções test-and-set.

■ Desabilitação de interrupções

A solução mais simples para o problema da exclusão mútua é fazer com que o processo desabilite todas as interrupções antes de entrar em sua região crítica, e as reabilite após deixar a região crítica. Como a mudança de contexto de processos só pode ser realizada através de interrupções, o processo que as desabilitou terá acesso exclusivo garantido.

```
BEGIN
.
.
.
Desabilita_Interrupcoes;
Regiao_Critica;
Habilita_Interrupcoes;
.
.
.
END .
```

Esta solução, apesar de simples, apresenta algumas limitações. Primeiro, a multiprogramação pode ficar seriamente comprometida, já que a concorrência entre processos tem como base o uso de interrupções. Um caso mais grave poderia ocorrer caso um processo desabilitasse as interrupções e não tornasse a habilitá-las. Neste caso, é provável que o sistema tivesse seu funcionamento seriamente comprometido.

Em sistemas com múltiplos processadores, essa solução torna-se ineficiente devido ao tempo de propagação quando um processador sinaliza aos demais que as interrupções devem ser habilitadas ou desabilitadas. Outra consideração é que o mecanismo de clock do sistema é implementado através de interrupções, devendo esta solução ser utilizada com bastante critério.

Apesar das limitações apresentadas, essa solução pode ser útil quando se deseja que a execução de parte do núcleo do sistema operacional ocorra sem que haja interrupção. Dessa forma, o sistema pode garantir que não ocorrerão problemas de inconsistência em suas estruturas de dados durante a execução de algumas rotinas.

■ Instrução test-and-set

Muitos processadores possuem uma instrução de máquina especial que permite ler uma variável, armazenar seu conteúdo em uma outra área e atribuir um novo valor à mesma variável. Essa instrução especial é chamada *instrução test-and-set* e tem como característica ser executada sem interrupção, ou seja, trata-se de uma instrução indivisível. Dessa forma, é garantido que dois processos não manipulem uma variável compartilhada ao mesmo tempo, possibilitando a implementação da exclusão mútua.

A instrução test-and-set possui o formato a seguir, e quando executada o valor lógico da variável Y é

copiado para X, sendo atribuído à variável Y o valor lógico verdadeiro.

Test-and-Set (X, Y) ;

Para coordenar o acesso concorrente a um recurso, a instrução test-and-set utiliza uma variável lógica global, que no programa Test_and_Set é denominada Bloqueio. Quando a variável Bloqueio for falsa, qualquer processo poderá alterar seu valor para verdadeiro através da instrução test-and-set e, assim, acessar o recurso de forma exclusiva. Ao terminar o acesso, o processo deve simplesmente retornar o valor da variável para falso, liberando o acesso ao recurso.

```
PROGRAM Test_and_Set;
VAR Bloqueio : BOOLEAN;

PROCEDURE Processo_A;
VAR Pode_A : BOOLEAN;
BEGIN
REPEAT
Pode_A := true;
WHILE (Pode_A) DO
Test_and_Set (Pode_A, Bloqueio);
Regiao_Critica_A;
Bloqueio := false;
UNTIL false;
END;

PROCEDURE Processo_B;
VAR Pode_B : BOOLEAN;
BEGIN
REPEAT
Pode_B := true;
WHILE (Pode_B) DO
Test_and_Set (Pode_B, Bloqueio);
Regiao_Critica_B;
Bloqueio := false;
UNTIL false;
END;

BEGIN
Bloqueio := false;
PARBEGIN
Processo_A;
Processo_B;
PAREND;
END.
```

No exemplo da Tabela 7.3, o Processo B executa a instrução Test_and_Set antes do processo A, alterando o valor da variável Bloqueio para true e executando a Regiao_Critica_B. Enquanto o Processo B não alterar o valor da variável Bloqueio para false, o Processo A ficará bloqueado. Quando o Processo B desbloquear o acesso ao recurso, o Processo A poderá executar a Regiao_Critica_A.

Tabela 7.3 Execução do Programa Test_and_Set

Processo	Instrução	Pode_A	Pode_B	Bloqueio
A	REPEAT	*	*	false
B	REPEAT	*	*	false
A	Pode_A := true	true	*	false
B	Pode_B := true	*	true	false
B	WHILE	*	true	false
B	Test_and_Set	*	false	true
A	WHILE	true	*	true
A	Test_and_Set	true	*	true
B	Regiao_Critica_B	*	false	true
A	WHILE	true	*	true
A	Test_and_Set	true	*	true
B	Bloqueio := false	*	false	false
B	UNTIL false	*	false	false
B	REPEAT	*	false	false
A	WHILE	true	*	false
A	Test_and_Set	false	*	true
A	Regiao_Critica_A	false	*	true

O uso de uma instrução especial de máquina oferece algumas vantagens, como a simplicidade de implementação da exclusão mútua em múltiplas regiões críticas e o uso da solução em arquiteturas com múltiplos processadores. A principal desvantagem é a possibilidade do starvation, pois a seleção do processo para acesso ao recurso é arbitrária.

7.5.2 Soluções de Software

Diversos algoritmos foram propostos na tentativa de implementar a exclusão mútua através de soluções de software. As primeiras soluções tratavam apenas da exclusão mútua para dois processos e, inicialmente, apresentavam alguns problemas. A seguir apresenta-se de forma evolutiva como foi o desenvolvimento de uma solução definitiva para a exclusão mútua entre N processos.

■ Primeiro algoritmo

Este algoritmo apresenta uma solução para exclusão mútua entre dois processos, onde um mecanismo de controle alterna a execução das regiões críticas. Cada processo (A e B) é representado por um procedimento que possui um loop infinito (REPEAT/UNTIL), onde é feito o acesso a um recurso por diversas vezes. A sequência de comandos, dentro do loop, é formada por um protocolo de entrada, uma região crítica e um protocolo de saída. A região crítica é representada por uma rotina, onde o acesso ao recurso realmente acontece. Após o acesso à região crítica, tanto o Processo A quanto o Processo B realizam processamentos individuais.

O mecanismo utiliza uma variável de bloqueio, indicando qual processo pode ter acesso à região crítica. Inicialmente, a variável global Vez é igual a ‘A’, indicando que o Processo A pode executar sua região crítica. O Processo B, por sua vez, fica esperando enquanto Vez for igual a ‘A’. O Processo B só executará sua região crítica quando o Processo A atribuir o valor ‘B’ à variável de bloqueio Vez. Desta forma, estará garantida a exclusão mútua entre os dois processos.

```
PROGRAM Algoritmo_1;
VAR Vez : CHAR;

PROCEDURE Processo_A;
BEGIN
REPEAT
WHILE (Vez = 'B') DO (* Nao faz nada *);
Regiao_Critica_A;
Vez := 'B';
Processamento_A;
UNTIL false;
END;

PROCEDURE Processo_B;
BEGIN
REPEAT
WHILE (Vez = 'A') DO (* Nao faz nada *);
Regiao_Critica_B;
Vez := 'A';
Processamento_B;
UNTIL false;
END;

BEGIN
Vez := 'A';
PARBEGIN
Processo_A;
Processo_B;
PAREND;
END.
```

Este algoritmo, apesar de implementar a exclusão mútua, apresenta duas limitações, das quais a primeira surge do próprio mecanismo de controle utilizado. O acesso ao recurso compartilhado só pode ser realizado por dois processos e sempre de maneira alternada; com isso, um processo que necessite utilizar o recurso mais vezes do que outro permanecerá grande parte do tempo bloqueado.

No exemplo apresentado na Tabela 7.4, caso o Processo A permaneça muito tempo na rotina Processamento_A, é possível que o Processo B queira executar sua região crítica e não consiga, mesmo que o Processo A não esteja utilizando o recurso. Como o Processo B pode executar seu loop mais rapidamente que o Processo A, a possibilidade de executar sua região crítica fica limitada pela velocidade de processamento do Processo A.

Tabela 7.4 Execução do Programa Algoritmo_1

Processo	Instrução	Vez
B	WHILE (Vez = 'A')	A
A	WHILE (Vez = 'B')	A
A	Regiao_Critica_A	A
A	Vez := 'B'	B
A	Processamento_A	B
B	Regiao_Critica_B	B
B	Vez := 'A'	A
B	Processamento_B	A
B	WHILE (Vez = 'A')	A

Outro problema existente nesta solução é que, no caso da ocorrência de algum problema com um dos processos, de forma que a variável de bloqueio não seja alterada, o outro processo permanecerá indefinidamente bloqueado, aguardando o acesso ao recurso.

■ Segundo algoritmo

O problema principal do primeiro algoritmo é que ambos os processos trabalham com uma mesma variável global, cujo conteúdo indica qual processo tem o direito de entrar na região crítica. Para evitar esta situação, o segundo algoritmo introduz uma variável para cada processo (CA e CB) que indica se o processo está ou não em sua região crítica. Neste caso, toda vez que um processo desejar entrar em sua região crítica a variável do outro processo é testada para verificar se o recurso está livre para uso.

```

PROGRAM Algoritmo_2;
VAR CA, CB : BOOLEAN;

PROCEDURE Processo_A;
BEGIN
REPEAT
WHILE (CB) DO (* Nao faz nada *);
CA := true;
Regiao_Critica_A;

```

```

CA := false;
Processamento_A;
UNTIL false;
END;

PROCEDURE Processo_B;
BEGIN
REPEAT
WHILE (CA) DO (* Nao faz nada *);
CB := true;
Regiao_Critica_B;
CB := false;
Processamento_B;
UNTIL false;
END;

BEGIN
CA := false;
CB := false;
PARBEGIN
Processo_A;
Processo_B;
PAREND;
END.

```

Neste segundo algoritmo, o uso do recurso não é realizado necessariamente alternado. Caso ocorra algum problema com um dos processos fora da região crítica, o outro processo não ficará bloqueado, o que, porém, não resolve por completo o problema. Caso um processo tenha um problema dentro da sua região crítica ou antes de alterar a variável, o outro processo permanecerá indefinidamente bloqueado. Mais grave que o problema do bloqueio é que esta solução, na prática, é pior do que o primeiro algoritmo apresentado, pois nem sempre a exclusão mútua é garantida. O exemplo apresentado na Tabela 7.5 ilustra o problema, considerando que cada processo executa cada instrução alternadamente.

Tabela 7.5 Execução do Programa Algoritmo_2

Processo	Instrução	CA	CB
A	WHILE (CB)	false	false
B	WHILE (CA)	false	false
A	CA := true	true	false
B	CB := true	true	true
A	Regiao_Critica_A	true	true
B	Regiao_Critica_B	true	true

■ Terceiro algoritmo

O terceiro algoritmo tenta solucionar o problema apresentado no segundo, colocando a instrução de atribuição das variáveis CA e CB antes do loop de teste.

```
PROGRAM Algoritmo_3;
VAR CA, CB : BOOLEAN;

PROCEDURE Processo_A;
BEGIN
REPEAT
CA := true;
WHILE (CB) DO (* Nao faz nada *);
Regiao_Critica_A;
CA := false;
Processamento_A;
UNTIL false;
END;

PROCEDURE Processo_B;
BEGIN
REPEAT
CB := true;
WHILE (CA) DO (* Nao faz nada *);
Regiao_Critica_B;
CB := false;
Processamento_B;
UNTIL false;
END;
```

Esta alteração resulta na garantia da exclusão mútua, porém introduz um novo problema, que é a possibilidade de bloqueio indefinido de ambos os processos. Caso os dois processos alterem as variáveis CA e CB antes da execução da instrução WHILE, nenhum dos dois processos poderá entrar em suas regiões críticas, como se o recurso já estivesse alocado.

■ Quarto algoritmo

No terceiro algoritmo, cada processo altera o estado da sua variável indicando que entrará na região crítica sem conhecer o estado do outro processo, o que acaba resultando no problema apresentado. O quarto algoritmo apresenta uma implementação onde o processo, da mesma forma, altera o estado da variável antes de entrar na sua região crítica, porém existe a possibilidade de esta alteração ser revertida.

```
PROGRAM Algoritmo_4;
VAR CA,CB : BOOLEAN;

PROCEDURE Processo_A;
BEGIN
REPEAT
CA:= true;
```

```

WHILE (CB) DO
BEGIN
CA:= false;
{ pequeno intervalo de tempo aleatório }
CA:= true
END;
Regiao_Critica_A;
CA := false;
UNTIL false;
END;

PROCEDURE Processo_B;
BEGIN
REPEAT
CB := true;
WHILE (CA) DO
BEGIN
CB := false;
{ pequeno intervalo de tempo aleatório }
CB := true;
END;
Regiao_Critica_B;
CB := false;
UNTIL false;
END;

BEGIN
CA := false;
CB := false;
PARBEGIN
Processo_A;
Processo_B;
PAREND;
END.

```

Apesar de esta solução garantir a exclusão mútua e não gerar o bloqueio simultâneo dos processos, uma nova situação indesejada pode ocorrer eventualmente. No caso de os tempos aleatórios serem próximos e a concorrência gerar uma situação onde os dois processos alterem as variáveis CA e CB para falso antes do término do loop, nenhum dos dois processos conseguirá executar sua região crítica. Mesmo esta situação não sendo permanente, pode gerar alguns problemas.

■ Algoritmo de Dekker

A primeira solução de software que garantiu a exclusão mútua entre dois processos sem a incorreção de outros problemas foi proposta pelo matemático holandês T. Dekker, com base no primeiro e no quarto algoritmos. O algoritmo de Dekker possui uma lógica bastante complexa e pode ser encontrado em Ben-Ari (1990) e Stallings (1997). Posteriormente, G. L. Peterson propôs outra solução mais simples para o mesmo problema, conforme apresentado a seguir.

■ Algoritmo de Peterson

O algoritmo proposto por G. L. Peterson apresenta uma solução para o problema da exclusão mútua entre dois processos que pode ser facilmente generalizada para o caso de N processos. Similar ao terceiro algoritmo, esta solução, além das variáveis de condição (CA e CB) que indicam o desejo de cada processo entrar em sua região crítica, introduz a variável Vez para resolver os conflitos gerados pela concorrência.

Antes de acessar a região crítica, o processo sinaliza esse desejo através da variável de condição, porém o processo cede o uso do recurso ao outro processo, indicado pela variável Vez. Em seguida, o processo executa o comando WHILE como protocolo de entrada da região crítica. Neste caso, além da garantia da exclusão mútua, o bloqueio indefinido de um dos processos no loop nunca ocorrerá, já que a variável Vez sempre permitirá a continuidade da execução de um dos processos.

```
PROGRAM Algoritmo_Peterson;
VAR CA,CB: BOOLEAN;
Vez : CHAR;

PROCEDURE Processo_A;
BEGIN
REPEAT
CA := true;
Vez := 'B';
WHILE (CB and Vez = 'B') DO (* Nao faz nada *);
Regiao_Critica_A;
CA := false;
Processamento_A;
UNTIL false;
END;

PROCEDURE Processo_B;
BEGIN
REPEAT
CB := true;
Vez := 'A';
WHILE (CA and Vez = 'A') DO (* Nao faz nada *);
Regiao_Critica_B;
CB := false;
Processamento_B;
UNTIL false;
END;

BEGIN
CA := false;
CB := false;
PARBEGIN
Processo_A;
Processo_B;
PAREND;
END.
```

■ Algoritmo para exclusão mútua entre N processos

O algoritmo de Dekker e a versão inicial do algoritmo de Peterson garantem a exclusão mútua de dois processos. Posteriormente, o algoritmo de Peterson foi generalizado para o caso de N processos Hofri (1990). O algoritmo do padeiro (bakery algorithm), proposto por Lamport (1974), é uma solução clássica para o problema da exclusão mútua entre N processos e pode ser encontrado em Ben-Ari (1990) e Silberschatz, Galvin e Gagne (2001). Outros algoritmos que também foram apresentados para exclusão mútua de N processos podem ser consultados em Peterson (1983) e Lamport (1987).

Apesar de todas as soluções até então apresentadas implementarem a exclusão mútua, todas possuíam uma deficiência conhecida como *espera ocupada* (*busy wait*). Na espera ocupada, toda vez que um processo não consegue entrar em sua região crítica, por já existir outro processo acessando o recurso, o processo permanece em looping, testando uma condição, até que lhe seja permitido o acesso. Dessa forma, o processo em looping consome tempo do processador desnecessariamente, podendo ocasionar problemas ao desempenho do sistema.

A solução para o problema da espera ocupada foi a introdução de mecanismos de sincronização que permitiram que um processo, quando não pudesse entrar em sua região crítica, fosse colocado no estado de espera. Esses mecanismos, conhecidos como semáforos e monitores, são apresentados posteriormente.

► 7.6 Sincronização Condicional

Sincronização condicional é uma situação em que o acesso ao recurso compartilhado exige a sincronização de processos vinculada a uma condição de acesso. Um recurso pode não se encontrar pronto para uso devido a uma condição específica. Nesse caso, o processo que deseja acessá-lo deverá permanecer bloqueado até que o recurso fique disponível.

Um exemplo clássico desse tipo de sincronização é a comunicação entre dois processos por meio de operações de gravação e leitura em um buffer, como foi visto na Fig. 7.1, em que processos geram informações (*processos produtores*) utilizadas por outros processos (*processos consumidores*). Nessa comunicação, enquanto um processo grava dados em um buffer, o outro lê os dados, concorrentemente. Os processos envolvidos devem estar sincronizados a uma variável de condição, de forma que um processo não tente gravar dados em um buffer cheio ou realizar uma leitura em um buffer vazio.

O programa a seguir exemplifica o problema da sincronização condicional, também conhecido como *problema do produtor/consumidor* ou do *buffer limitado*. O recurso compartilhado é um buffer, definido no algoritmo com o tamanho TamBuf e sendo controlado pela variável Cont. Sempre que a variável Cont for igual a 0, significa que o buffer está vazio e o processo Consumidor deve permanecer aguardando até que se grave um dado. Da mesma forma, quando a variável Cont for igual a TamBuf, significa que o buffer está cheio e o processo Produtor deve aguardar a leitura de um novo dado. Nessa solução, a tarefa de colocar e retirar os dados do buffer é realizada, respectivamente, pelos procedimentos Grava_Buffer e Le_Buffer, executados de forma mutuamente exclusiva. Não está sendo considerada, neste algoritmo, a implementação da exclusão mútua na variável compartilhada Cont.

```
PROGRAM Produtor_Consumidor_1;
```

```

CONST TamBuf= (* Tamanho qualquer *);
TYPE Tipo_Dado = (* Tipo qualquer *);
VAR Buffer : ARRAY [1..TamBuf] OF Tipo_Dado;
Dado_1 : Tipo_Dado;
Dado_2 : Tipo_Dado;
Cont: 0..TamBuf;

PROCEDURE Produtor;
BEGIN
REPEAT
Produz_Dado (Dado_1);
WHILE (Cont = TamBuf) DO (* Nao faz nada *);
Grava_Buffer (Dado_1, Buffer);
Cont := Cont + 1;
UNTIL false;
END;

PROCEDURE Consumidor;
BEGIN
REPEAT
WHILE (Cont = 0) DO (* Nao faz nada *);
Le_Buffer (Dado_2, Buffer);
Consome_Dado (Dado_2);
Cont := Cont - 1;
UNTIL false;
END;

BEGIN
Cont := 0;
PARBEGIN
Produtor;
Consumidor;
PAREND;
END.

```

Apesar de o algoritmo apresentado resolver a questão da sincronização condicional, o problema da espera ocupada também se faz presente, sendo somente solucionado pelos mecanismos de sincronização semáforos e monitores.

► 7.7 Semáforos

O conceito de *semáforos* foi proposto por E. W. Dijkstra em 1965, sendo apresentado como um mecanismo de sincronização que permitia implementar, de forma simples, a exclusão mútua e a sincronização condicional entre processos. De fato, o uso de semáforos tornou-se um dos principais mecanismos utilizados em projetos de sistemas operacionais e em aplicações concorrentes. Atualmente, a maioria das linguagens de programação disponibiliza rotinas para uso de semáforos.

Um semáforo é uma variável inteira, não negativa, que só pode ser manipulada por duas instruções: DOWN e UP, também chamadas originalmente por Dijkstra de instruções P (*proberen*, teste em holandês)

e V (*verhogen*, incremento em holandês). As instruções DOWN e UP são indivisíveis, ou seja, a execução destas instruções não pode ser interrompida. A instrução UP incrementa uma unidade ao valor do semáforo, enquanto a DOWN decrementa a variável. Como, por definição, valores negativos não podem ser atribuídos a um semáforo, a instrução DOWN executada em um semáforo com valor 0 faz com que o processo entre no estado de espera. Em geral, essas instruções são implementadas no processador, que deve garantir todas essas condições.

Os semáforos podem ser classificados como binários ou contadores. Os *semáforos binários*, também chamados *mutexes (mutual exclusion semaphores)*, só podem assumir os valores 0 e 1, enquanto os *semáforos contadores* podem assumir qualquer valor inteiro positivo, além do 0.

A seguir, será apresentado o uso de semáforos na implementação da exclusão mútua e da sincronização condicional. Ao final deste item, alguns problemas clássicos de sincronização são também apresentados.

7.7.1 Exclusão Mútua Utilizando Semáforos

A exclusão mútua pode ser implementada através de um semáforo binário associado ao recurso compartilhado. A principal vantagem desta solução em relação aos algoritmos anteriormente apresentados é a não ocorrência da espera ocupada.

As instruções DOWN e UP funcionam como protocolos de entrada e saída, respectivamente, para que um processo possa entrar e sair de sua região crítica. O semáforo fica associado a um recurso compartilhado, indicando quando o recurso está sendo acessado por um dos processos concorrentes. O valor do semáforo igual a 1 indica que nenhum processo está utilizando o recurso, enquanto o valor 0 indica que o recurso está em uso.

Sempre que deseja entrar na sua região crítica, um processo executa uma instrução DOWN. Se o semáforo for igual a 1, este valor é decrementado, e o processo que solicitou a operação pode executar as instruções da sua região crítica. De outra forma, se uma instrução DOWN é executada em um semáforo com valor igual a 0, o processo fica impedido do acesso, permanecendo em estado de espera e, consequentemente, não gerando overhead no processador.

O processo que está acessando o recurso, ao sair de sua região crítica, executa uma instrução UP, incrementando o valor do semáforo e liberando o acesso ao recurso. Se um ou mais processos estiverem esperando pelo uso do recurso (operações DOWN pendentes), o sistema selecionará um processo na fila de espera associada ao recurso e alterará o seu estado para pronto (Fig. 7.3).

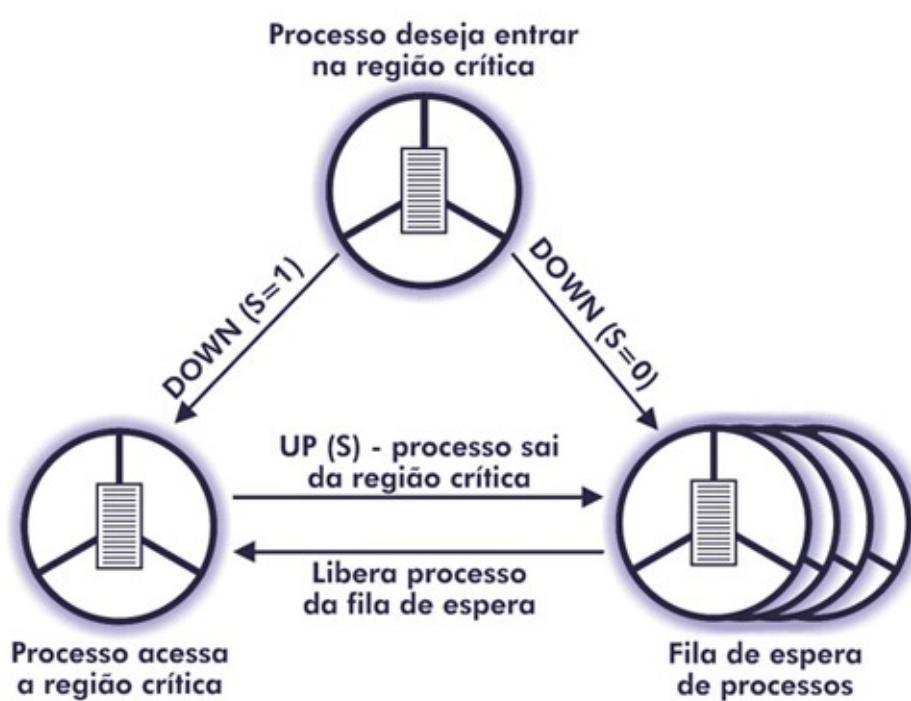


Fig. 7.3 Utilização do semáforo binário na exclusão mútua.

O programa Semaforo_1 apresenta uma solução para o problema da exclusão mútua entre dois processos utilizando semáforos. O semáforo é inicializado com o valor 1, indicando que nenhum processo está executando sua região crítica.

```

PROGRAM Semaforo_1;
VAR s : Semaforo := 1; (* inicializacao do semaforo *)

PROCEDURE Processo_A;
BEGIN
REPEAT
DOWN (s);
Regiao_Critica_A;
UP (s);
UNTIL false;
END;

PROCEDURE Processo_B;
BEGIN
REPEAT
DOWN (s);
Regiao_Critica_B;
UP (s);
UNTIL false;
END;

BEGIN
PARBEGIN
Processo_A;
Processo_B;
PAREND;
END.

```

O exemplo apresentado na Tabela 7.6 ilustra a execução do programa Semaforo_1, onde o Processo A executa a instrução DOWN, fazendo com que o semáforo seja decrementado de 1 e passe a ter o valor 0. Em seguida, o Processo A ganha o acesso à sua região crítica. O Processo B também executa a instrução DOWN no semáforo, mas como seu valor é igual a 0 ficará aguardando até que o Processo A execute a instrução UP, ou seja, volte o valor do semáforo para 1.

Tabela 7.6 Execução do Programa Semaforo_1

Processo	Instrução	S	Espera
A	REPEAT	1	*
B	REPEAT	1	*
A	DOWN (s)	0	*
B	REPEAT	0	*
A	Regiao_Critica_A	0	*
B	DOWN (s)	0	Processo_B
A	UP (s)	1	Processo_B
B	DOWN (s)	1	Processo_B
A	REPEAT	1	Processo_B
B	Regiao_Critica_B	0	*

7.7.2 Sincronização Condicional Utilizando Semáforos

Além de permitirem a implementação da exclusão mútua, os semáforos podem ser utilizados nos casos onde a sincronização condicional é exigida. Um exemplo desse tipo de sincronização ocorre quando um processo solicita uma operação de E/S. O pedido faz com que o processo execute uma instrução DOWN no semáforo associado ao evento e fique no estado de espera, até que a operação seja completada. Quando a operação termina, a rotina de tratamento da interrupção executa um UP no semáforo, liberando o processo do estado de espera.

O problema do produtor/consumidor já apresentado é um exemplo de como a exclusão mútua e a sincronização condicional podem ser implementadas com o uso de semáforos. O programa Produtor_Consumidor_2 apresenta uma solução para esse problema, utilizando um buffer de apenas duas posições. O programa utiliza três semáforos, sendo um do tipo binário, empregado para implementar a exclusão mútua, e dois semáforos contadores para a sincronização condicional. O semáforo binário Mutex permite a execução das regiões críticas Grava_Buffer e Le_Buffer de forma mutuamente exclusiva. Os semáforos contadores Vazio e Cheio representam, respectivamente, se há posições livres no buffer para serem gravadas e posições ocupadas a serem lidas. Quando o semáforo Vazio for igual a 0, significa que o buffer está cheio e o processo produtor deverá aguardar até que o consumidor leia algum dado. Da

mesma forma, quando o semáforo Cheio for igual a 0, significa que o buffer está vazio e o consumidor deverá aguardar até que o produtor grave algum dado. Em ambas as situações, o semáforo sinaliza a impossibilidade de acesso ao recurso.

```
PROGRAM Produtor_Consumidor_2;
CONST TamBuf = 2;
TYPE Tipo_Dado = (* Tipo qualquer *);
VAR Vazio : Semaforo := TamBuf;
Cheio : Semaforo := 0;
Mutex : Semaforo := 1;
Buffer : ARRAY [1..TamBuf] OF Tipo_Dado;
Dado_1 : Tipo_Dado;
Dado_2 : Tipo_Dado;
PROCEDURE Produtor;
BEGIN
REPEAT
Produz_Dado (Dado_1);
DOWN (Vazio);
DOWN (Mutex);
Grava_Buffer (Dado_1, Buffer);
UP (Mutex);
UP (Cheio);
UNTIL false;
END;

PROCEDURE Consumidor;
BEGIN
REPEAT
DOWN (Cheio);
DOWN (Mutex);
Le_Buffer (Dado_2, Buffer);
UP (Mutex);
UP (Vazio);
Consome_Dado (Dado_2);
UNTIL false;
END;

BEGIN
PARBEGIN
Produtor;
Consumidor;
PAREND;
END.
```

O exemplo apresentado na Tabela 7.7 considera que o processo Consumidor é o primeiro a ser executado. Como o buffer está vazio ($Cheio = 0$), a operação DOWN ($Cheio$) faz com que o processo Consumidor permaneça no estado de espera. Em seguida, o processo Produtor grava um dado no buffer e, após executar o UP ($Cheio$), libera o processo Consumidor, que estará apto para ler o dado do buffer.

Tabela 7.7 Execução do Programa Produtor_Consumidor_2

Processo	Instrução	Vazio	Cheio	Mutex	Espera
Produtor	Produz_Dado	2	0	1	*
Consumidor	DOWN (Cheio)	2	0	1	Consumidor
Produtor	DOWN (Vazio)	1	0	1	Consumidor
Produtor	DOWN (Mutex)	1	0	0	Consumidor
Produtor	Grava_Buffer	1	0	0	Consumidor
Produtor	UP (Mutex)	1	0	1	Consumidor
Produtor	UP (Cheio)	1	1	1	Consumidor
Consumidor	DOWN (Cheio)	1	0	1	*
Produtor	Produz_Dado	1	0	1	Produtor
Consumidor	DOWN (Mutex)	1	0	0	Produtor
Consumidor	Le_Buffer	1	0	0	Produtor
Consumidor	UP (Mutex)	1	0	1	Produtor
Consumidor	UP (Vazio)	2	0	1	Produtor

Semáforos contadores são bastante úteis quando aplicados em problemas de sincronização condicional onde existem processos concorrentes alocando recursos do mesmo tipo (pool de recursos). O semáforo é inicializado com o número total de recursos do pool, e, sempre que um processo deseja alocar um recurso, executa um DOWN, subtraindo 1 do número de recursos disponíveis. Da mesma forma, sempre que o processo libera um recurso para o pool, executa um UP. Se o semáforo contador ficar com o valor igual a 0, significa que não existem mais recursos a serem utilizados, e o processo que solicita um recurso permanece em estado de espera até que outro processo libere algum recurso para o pool.

7.7.3 Problema dos Filósofos

O problema dos filósofos é um exemplo clássico de sincronização de processos proposto por Dijkstra. Nesse problema, há uma mesa com cinco pratos e cinco garfos onde os filósofos podem sentar, comer e pensar. Toda vez que um filósofo para de pensar e deseja comer é necessário que ele utilize dois garfos, posicionados à sua direita e à sua esquerda.

O algoritmo Filosofo_1 apresenta uma solução que não resolve o problema totalmente, pois se todos os filósofos estiverem segurando apenas um garfo cada um, nenhum filósofo conseguirá comer. Esta situação é conhecida como deadlock.

```
PROGRAM Filosofo_1;
VAR Garfos:ARRAY [0..4] of Semaforo := 1;
I:INTEGER;
```

```

PROCEDURE Filosofo (I : INTEGER);
BEGIN
REPEAT
Pensando;
DOWN (Garfos[I]);
DOWN (Garfos[(I+1) MOD 5]);
Comendo;
UP (Garfos[I]);
UP (Garfos[(I+1) MOD 5]);
UNTIL false;
END;

BEGIN
PARBEGIN
FOR I := 0 TO 4 DO
Filosofo (I);
PAREND;
END.

```

Existem várias soluções para resolver o problema dos filósofos sem a ocorrência do deadlock, como as apresentadas a seguir. A solução (a) é descrita no algoritmo Filosofo_2.

- (a) Permitir que apenas quatro filósofos sentem à mesa simultaneamente;
- (b) Permitir que um filósofo pegue um garfo apenas se o outro estiver disponível;
- (c) Permitir que um filósofo ímpar pegue primeiro o seu garfo da esquerda e depois o da direita, enquanto um filósofo par pegue o garfo da direita e, em seguida, o da esquerda.

```

PROGRAMFilosofo_2;
VARGarfos :ARRAY [0..4] of Semaforo := 1;
Lugares:Semaforo := 4;
I:INTEGER;

PROCEDURE Filosofo (I : INTEGER);
BEGIN
REPEAT
Pensando;
DOWN (Lugares);
DOWN (Garfos[I]);
DOWN (Garfos[(I+1) MOD 5]);
Comendo;
UP (Garfos[I]);
UP (Garfos[(I+1) MOD 5]);
UP (Lugares);
UNTIL false;
END;

BEGIN
PARBEGIN

```

```

FOR I := 0 TO 4 DO
Filosofo (I);
PAREND;
END.

```

7.7.4 Problema do Barbeiro

O problema do barbeiro é outro exemplo clássico de sincronização de processos. Neste problema, um barbeiro recebe clientes para cortar o cabelo. Na barbearia há uma cadeira de barbeiro e apenas cinco cadeiras para clientes esperarem. Quando um cliente chega, caso o barbeiro esteja trabalhando ele se senta, se houver cadeira vazia, ou vai embora, se todas as cadeiras estiverem ocupadas. No caso de o barbeiro não ter nenhum cliente para atender, ele se senta na cadeira e dorme até que um novo cliente apareça.

```

PROGRAM Barbeiro;
CONST Cadeiras = 5;
VAR Clientes: Semaforo:= 0;
Barbeiro: Semaforo:= 0;
Mutex: Semaforo:= 1;
Espera: integer:= 0;

PROCEDURE Barbeiro;
BEGIN
REPEAT
DOWN (Clientes);
DOWN (Mutex);
Espera := Espera - 1;
UP (Barbeiro);
UP (Mutex);
Corta_Cabelo;
UNTIL false;
END;

PROCEDURE Cliente;
BEGIN
DOWN (Mutex);
IF (Espera < Cadeiras)
BEGIN
Espera := Espera + 1;
UP (Clientes);
UP (Mutex);
DOWN (Barbeiro);
Ter_Cabelo_Cortado;
END
ELSE UP (Mutex);
END;

```

A solução utiliza o semáforo contador Clientes e os semáforos binários Barbeiro e Mutex. No processo Barbeiro é executado um DOWN no semáforo Clientes com o objetivo de selecionar um cliente para o

corte. Caso não exista nenhum cliente aguardando, o processo Barbeiro permanece no estado de espera. No caso de existir algum cliente, a variável Espera é decrementada, e para isso é utilizado o semáforo Mutex visando garantir o uso exclusivo da variável. Uma instrução UP é executada no semáforo Barbeiro para indicar que o recurso barbeiro está trabalhando.

No processo Cliente, o semáforo Mutex garante a exclusão mútua da variável Espera, que permite verificar se todas as cadeiras já estão ocupadas. Nesse caso, o cliente vai embora, realizando um UP no Mutex. Caso contrário, a variável Espera é incrementada e o processo solicita o uso do recurso barbeiro através da instrução DOWN no semáforo Barbeiro.

► 7.8 Monitores

Monitores são mecanismos de sincronização de alto nível que tornam mais simples o desenvolvimento de aplicações concorrentes. O conceito de monitores foi proposto por Brinch Hansen, em 1972, e desenvolvido por C.A.R. Hoare em 1974, como um mecanismo de sincronização estruturado, ao contrário dos semáforos, que são considerados não estruturados.

O uso de semáforos exige do desenvolvedor bastante cuidado, pois qualquer engano pode levar a problemas de sincronização imprevisíveis e difíceis de serem reproduzidos devido à execução concorrente dos processos. Monitores são considerados mecanismos de alto nível e estruturados em função de serem implementados pelo compilador. Assim, o desenvolvimento de programas concorrentes fica mais fácil e as chances de erro são menores. Inicialmente, algumas poucas linguagens de programação, como Concurrent Pascal, Modula-2 e Modula-3, ofereciam suporte ao uso de monitores. Atualmente, a maioria das linguagens de programação disponibiliza rotinas para uso de monitores.

O monitor é formado por procedimentos e variáveis encapsulados dentro de um módulo. Sua característica mais importante é a implementação automática da exclusão mútua entre os procedimentos declarados, ou seja, somente um processo pode estar executando um dos procedimentos do monitor em um determinado instante. Toda vez que algum processo faz uma chamada a um desses procedimentos, o monitor verifica se já existe outro processo executando algum procedimento do monitor. Caso exista, o processo ficará aguardando a sua vez em uma fila de entrada. As variáveis globais de um monitor são visíveis apenas aos procedimentos da sua estrutura, sendo inacessíveis fora do contexto do monitor. Toda a inicialização das variáveis é realizada por um bloco de comandos do monitor, sendo executado apenas uma vez, na ativação do programa onde está declarado o monitor (Fig. 7.4).

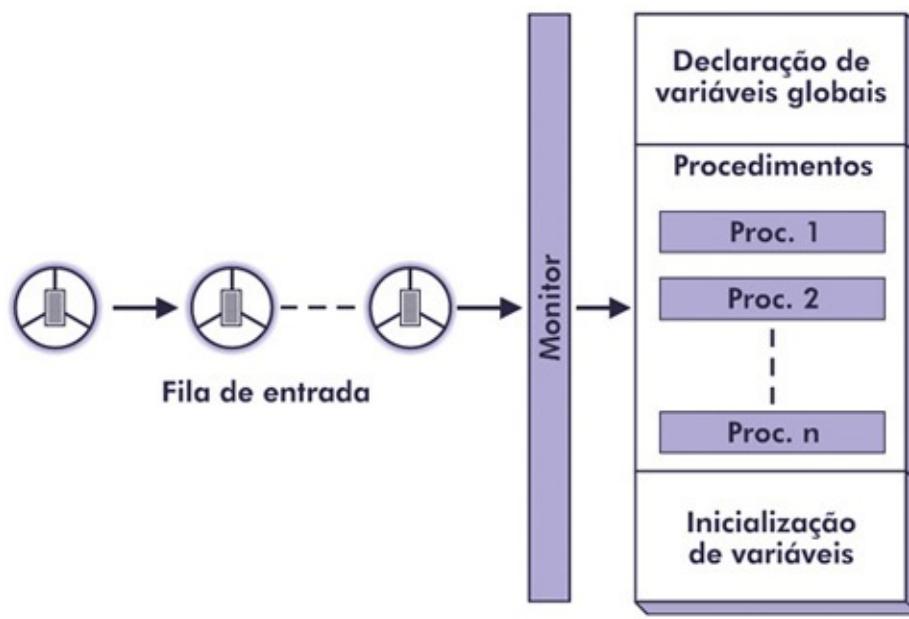


Fig. 7.4 Estrutura do monitor.

Um monitor é definido especificando-se um nome, declarando-se variáveis locais, procedimentos e um código de inicialização. A estrutura de um monitor é mostrada a seguir, utilizando uma sintaxe Pascal não convencional:

```

MONITOR Exclusao_Mutua;
(* Declaracao das variaveis do monitor *)
PROCEDURE Regiao_Critica_1;

BEGIN
.
.
END;

PROCEDURE Regiao_Critica_2;
BEGIN
.
.
END;

PROCEDURE Regiao_Critica_3;
BEGIN
.
.
END;

BEGIN
(* Codigo de inicializacao *)
END;

```

7.8.1 Exclusão Mútua Utilizando Monitores

A implementação da exclusão mútua utilizando monitores não é realizada diretamente pelo programador, como no caso do uso de semáforos. As regiões críticas devem ser definidas como procedimentos no monitor, e o compilador se encarregará de garantir a exclusão mútua entre esses procedimentos.

A comunicação do processo com o monitor é feita unicamente através de chamadas a seus procedimentos e dos parâmetros passados. O programa Monitor_1 apresenta a solução para o problema anteriormente apresentado, onde dois processos somam e diminuem, concorrentemente, o valor 1 da variável compartilhada X.

```
PROGRAM Monitor_1;
MONITOR Regiao_Critica;
VAR X : INTEGER;

PROCEDURE Soma;
BEGIN
  X := X + 1;
END;

PROCEDURE Diminui;
BEGIN
  X := X - 1;
END;

BEGIN
  X := 0;
END;

BEGIN
  PARBEGIN
    Regiao_Critica.Soma;
    Regiao_Critica.Diminui;
  PAREND;
END.
```

A inicialização da variável X com o valor zero só acontecerá uma vez, no momento da inicialização do monitor Regiao_Critica. Neste exemplo, podemos garantir que o valor de X ao final da execução concorrente de Soma e Diminui será igual a zero, porque, como os procedimentos estão definidos dentro do monitor, estará garantida a execução mutuamente exclusiva.

7.8.2 Sincronização Condicional Utilizando Monitores

Monitores também podem ser utilizados na implementação da sincronização condicional. Por meio de *variáveis especiais de condição*, é possível associar a execução de um procedimento que faz parte do monitor a uma determinada condição, garantindo a sincronização condicional.

As variáveis especiais de condição são manipuladas por intermédio de duas instruções, conhecidas como WAIT e SIGNAL. A instrução WAIT faz com que o processo seja colocado no estado de espera, até que algum outro processo sinalize com a instrução SIGNAL que a condição de espera foi satisfeita. Caso a

instrução SIGNAL seja executada e não haja processo aguardando a condição, nenhum efeito surtirá.

É possível que vários processos estejam com suas execuções suspensas, aguardando a sinalização de diversas condições. Para isso, o monitor organiza os processos em espera utilizando filas associadas às condições de sincronização. A execução da instrução SIGNAL libera apenas um único processo da fila de espera da condição associada. Um processo pode executar um procedimento de um monitor, mesmo quando um ou mais processos estão na fila de espera de condições (Fig. 7.5).

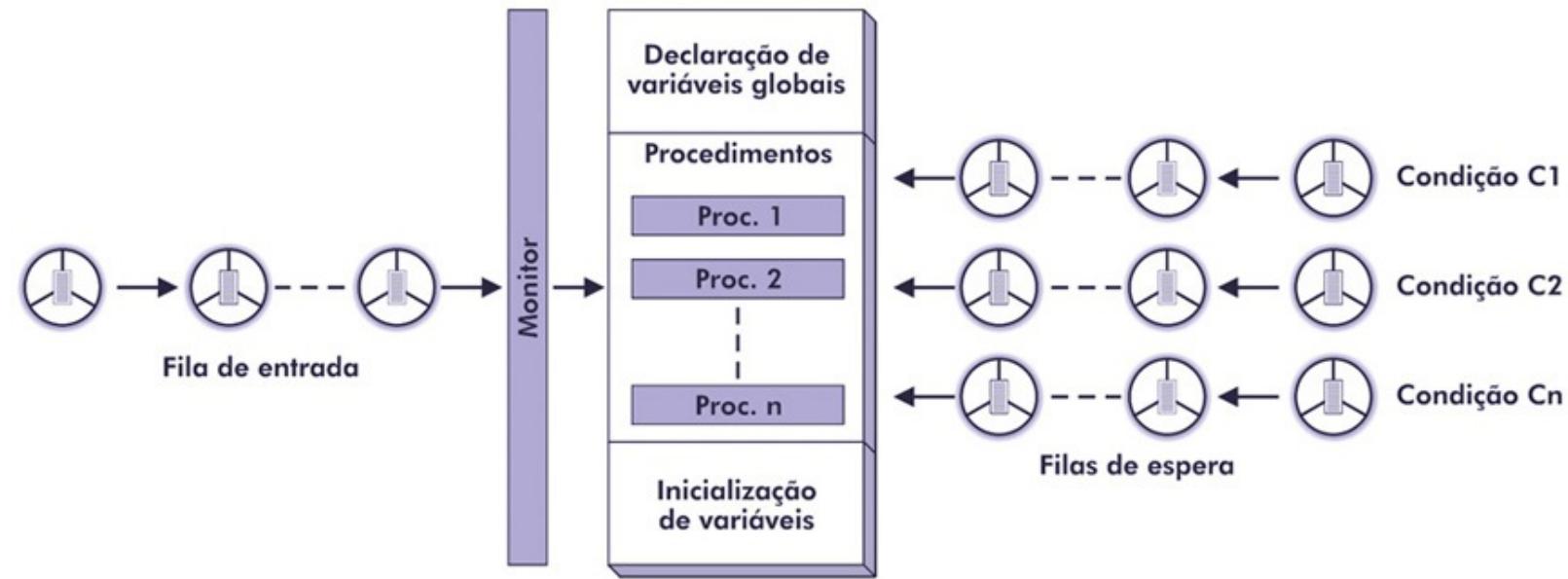


Fig. 7.5 Estrutura do monitor com variáveis de condição.

O problema do produtor/consumidor é mais uma vez apresentado, desta vez com o uso de um monitor para a solução. O monitor Condisional é estruturado com duas variáveis especiais de condição (Cheio e Vazio) e dois procedimentos (Produz e Consome).

```
MONITOR Condisional;
VAR Cheio, Vazio : (* Variaveis especiais de condicao *);

PROCEDURE Produz;
BEGIN
IF (Cont = TamBuf) THEN WAIT (Cheio);
.

IF (Cont = 1) THEN SIGNAL (Vazio);
END;

PROCEDURE Consome;
BEGIN
IF (Cont = 0) THEN WAIT (Vazio);
.

IF (Cont = TamBuf - 1) THEN SIGNAL (Cheio);
END;
```

```
BEGIN  
END;
```

Sempre que o processo produtor desejar gravar um dado no buffer, deverá fazê-lo através de uma chamada ao procedimento Produz do monitor (Condicional.Produz). Neste procedimento deve ser testado se o número de posições ocupadas no buffer é igual ao seu tamanho, ou seja, se o buffer está cheio. Caso o teste seja verdadeiro, o processo deve executar um WAIT em Cheio, indicando que a continuidade da sua execução depende dessa condição, e permanecer aguardando em uma fila de espera associada à variável Cheio. O processo bloqueado só poderá prosseguir sua execução quando um processo consumidor executar um SIGNAL na variável Cheio, indicando que um elemento do buffer foi consumido.

No caso do processo consumidor, sempre que desejar ler um dado do buffer deverá fazê-lo através de uma chamada ao procedimento Consome do monitor (Condicional.Consome). Sua condição de execução é existir ao menos um elemento no buffer. Caso o buffer esteja vazio, o processo deve executar um WAIT em Vazio, indicando que a continuidade da sua execução depende dessa condição, e permanecer aguardando em uma fila de espera associada à variável Vazio. O processo produtor que inserir o primeiro elemento no buffer deverá sinalizar esta condição através do comando SIGNAL na variável Vazio, liberando o consumidor.

O programa Produtor_Consumidor_3 descreve a solução completa com o uso do monitor Condicional apresentado. Os procedimentos Produz_Dado e Consome_Dado servem, respectivamente, como entrada e saída de dados para o programa. Os procedimentos Grava_Dado e Le_Dado são responsáveis, respectivamente, pela transferência e pela recuperação do dado no buffer.

```
PROGRAM Produtor_Consumidor_3;  
CONST TamBuf= (* Tamanho qualquer *);  
TYPE Tipo_Dado = (* Tipo qualquer*);  
VAR Buffer : ARRAY [1..TamBuf] OF Tipo_Dado;  
Dado: Tipo_Dado;  
  
MONITOR Condicional;  
VAR Vazio, Cheio : (* Variaveis de condicao *);  
Cont : INTEGER;  
  
PROCEDURE Produz;  
BEGIN  
IF (Cont = TamBuf) THEN WAIT (Cheio);  
Grava_Dado (Dado, Buffer);  
Cont = Cont + 1;  
IF (Cont = 1) THEN SIGNAL (Vazio);  
END;  
  
PROCEDURE Consome;  
BEGIN  
IF (Cont = 0) THEN WAIT (Vazio);  
Le_Dado (Dado, Buffer);  
Cont := Cont - 1;  
IF (Cont = TamBuf - 1) THEN SIGNAL(Cheio);  
END;
```

```

BEGIN
Cont := 0;
END;

PROCEDURE Produtor;
BEGIN
REPEAT
Produz_Dado (Dado);
Condicional.Produz;
UNTIL false;
END;

PROCEDURE Consumidor;
BEGIN
REPEAT
Condicional.Consome;
Consome_Dado (Dado);
UNTIL false;
END;

BEGIN
PARBEGIN
Produtor;
Consumidor;
PAREND;
END.

```

► 7.9 Troca de Mensagens

Troca de mensagens é um mecanismo de comunicação e sincronização entre processos. O sistema operacional possui um subsistema de mensagem que suporta esse mecanismo sem que haja necessidade do uso de variáveis compartilhadas. Para que haja a comunicação entre os processos deve existir um canal de comunicação, podendo esse meio ser um buffer ou um link de uma rede de computadores.

Os processos cooperativos podem fazer uso de um buffer para trocar mensagens através de duas rotinas: SEND (receptor, mensagem) e RECEIVE (transmissor, mensagem). A rotina SEND permite o envio de uma mensagem para um *processo receptor*, enquanto a rotina RECEIVE possibilita o recebimento de mensagem enviada por um *processo transmissor* (Fig. 7.6).



Fig. 7.6 Transmissão de mensagem.

O mecanismo de troca de mensagens exige que os processos envolvidos na comunicação tenham suas execuções sincronizadas. A sincronização é necessária, já que uma mensagem somente pode ser tratada por um processo após ter sido recebida, ou o envio de mensagem pode ser uma condição para a continuidade de execução de um processo.

A troca de mensagens entre processos pode ser implementada de duas maneiras distintas, comunicação direta e comunicação indireta. A *comunicação direta* entre dois processos exige que, ao enviar ou receber uma mensagem, o processo enderece explicitamente o nome do processo receptor ou transmissor. Uma característica deste tipo de comunicação é só permitir a troca de mensagem entre dois processos. O principal problema da comunicação direta é a necessidade da especificação do nome dos processos envolvidos na troca de mensagens. No caso de mudança na identificação dos processos, o código do programa deve ser alterado e recompilado (Fig. 7.7).



Fig. 7.7 Comunicação direta.

A *comunicação indireta* entre processos utiliza uma área compartilhada, onde as mensagens podem ser colocadas pelo processo transmissor e retiradas pelo receptor. Esse tipo de buffer é conhecido como *mailbox* ou *port*, e suas características, como identificação e capacidade de armazenamento de mensagens, são definidas no momento de criação. Na comunicação indireta, vários processos podem estar associados a mailbox, e os parâmetros dos procedimentos SEND e RECEIVE passam a ser nomes de mailboxes, e não mais nomes de processos (Fig. 7.8).

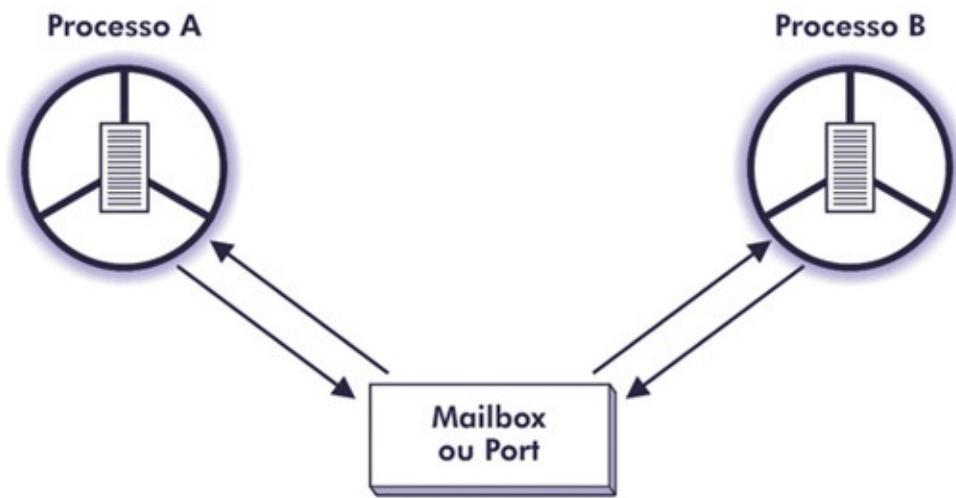


Fig. 7.8 Comunicação indireta.

Independentemente do mecanismo de comunicação utilizado, processos que estão trocando mensagens devem ter suas execuções sincronizadas em função do fluxo de mensagens. Um processo não pode tratar uma mensagem até que esta tenha sido enviada por outro processo, ou um processo não pode receber uma mesma mensagem mais de uma vez. Existem três diferentes esquemas para implementar a sincronização entre processos que trocam mensagens.

O primeiro esquema de sincronização é garantir que um processo, ao enviar uma mensagem, permaneça esperando até que o processo receptor a leia. Na mesma condição, um processo, ao tentar receber uma mensagem ainda não enviada, deve permanecer aguardando até que o processo transmissor faça o envio. Esse tipo de comunicação implementa uma forma síncrona de comunicação entre os processos conhecida como *rendezvous*. O problema desta implementação é que a execução dos processos fica limitada ao tempo de processamento no tratamento das mensagens.

Uma variação mais eficiente do *rendezvous* é permitir que o processo transmissor não permaneça bloqueado aguardando a leitura da mensagem pelo processo receptor. Neste caso, um processo enviará mensagens para diversos destinatários tão logo seja possível.

O terceiro esquema implementa uma forma assíncrona de comunicação, onde nem o processo receptor nem o processo transmissor permanecem aguardando o envio e o recebimento de mensagens. Nesse caso, além da necessidade de buffers para armazenar as mensagens deve haver outros mecanismos de sincronização que permitam ao processo identificar se uma mensagem já foi enviada ou recebida. A grande vantagem deste mecanismo é aumentar a eficiência de aplicações concorrentes.

Uma solução para o problema do produtor/consumidor é apresentada utilizando a troca de mensagens.

```

PROGRAM Produtor_Consumidor_4;
PROCEDURE Produtor;
VAR Msg : Tipo_Msg;
BEGIN
REPEAT
Produz_Mensagem (Msg) ;
SEND (Msg) ;
UNTIL false;
END;
```

```

PROCEDURE Consumidor;
VAR Msg : Tipo_Msg;
BEGIN
REPEAT
RECEIVE (Msg) ;
Consome_Mensagem (Msg) ;
UNTIL false;
END;

BEGIN
PARBEGIN
Produtor;
Consumidor;
PAREND;
END .

```

► 7.10 Deadlock

Deadlock é a situação em que um processo aguarda por um recurso que nunca estará disponível ou um evento que não ocorrerá. Essa situação é consequência, na maioria das vezes, do compartilhamento de recursos, como dispositivos, arquivos e registros, entre processos concorrentes em que a exclusão mútua é exigida. O problema do deadlock torna-se cada vez mais frequente e crítico na medida em que os sistemas operacionais evoluem no sentido de implementar o paralelismo de forma intensiva e permitir a alocação dinâmica de um número ainda maior de recursos.

A Fig. 7.9 ilustra graficamente o problema do deadlock entre os processos PA e PB, quando utilizam os recursos R1 e R2. Inicialmente, PA obtém acesso exclusivo de R1, da mesma forma que PB obtém de R2. Durante o processamento, PA necessita de R2 para poder prosseguir. Como R2 está alocado a PB, PA ficará aguardando que o recurso seja liberado. Em seguida, PB necessita utilizar R1 e, da mesma forma, ficará aguardando até que PA libere o recurso. Como cada processo está esperando que o outro libere o recurso alocado é estabelecida uma condição conhecida por *espera circular*, caracterizando uma situação de deadlock.

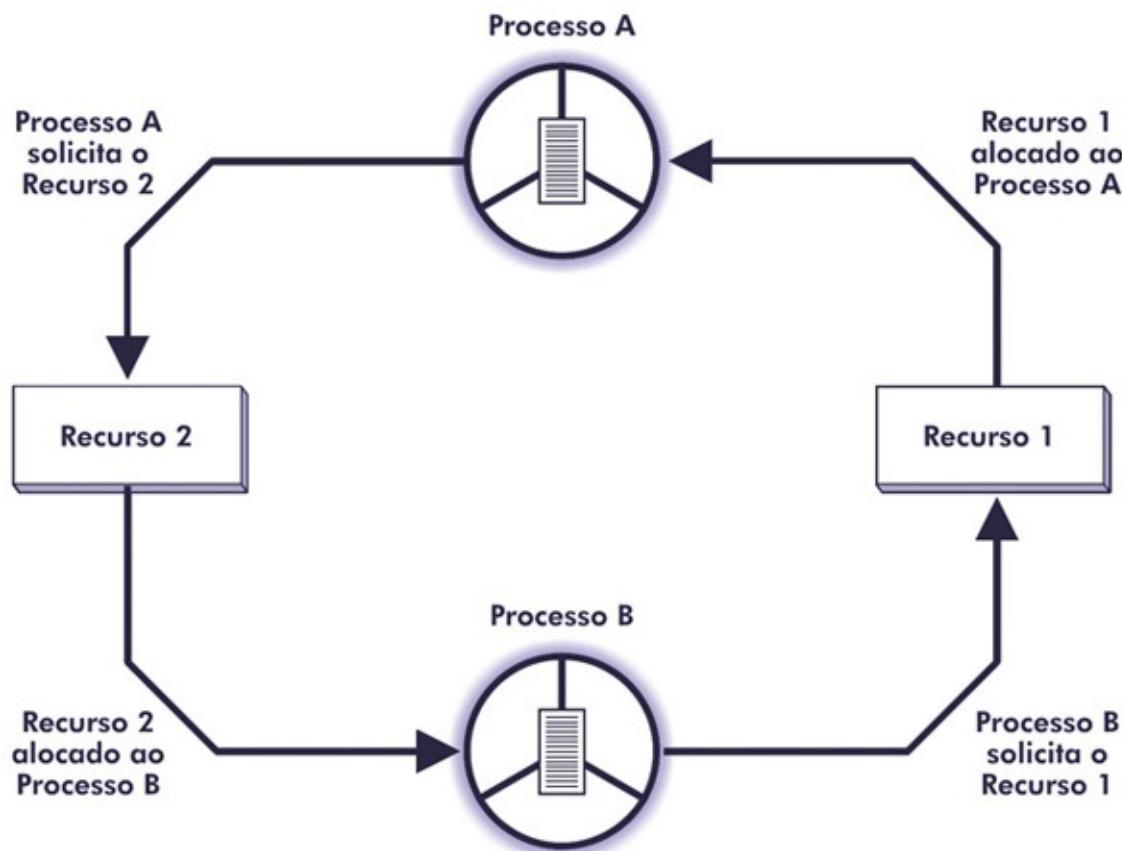


Fig. 7.9 Espera circular.

Para que ocorra a situação de deadlock, quatro condições são necessárias simultaneamente (Coffman, Elphick e Shoshani, 1971):

1. Exclusão mútua: cada recurso só pode estar alocado a um único processo em um determinado instante;
2. Espera por recurso: um processo, além dos recursos já alocados, pode estar esperando por outros recursos;
3. Não preempção: um recurso não pode ser liberado de um processo só porque outros processos desejam o mesmo recurso;
4. Espera circular: um processo pode ter de esperar por um recurso alocado a outro processo, e vice-versa.

O problema do deadlock existe em qualquer sistema multiprogramável; no entanto as soluções implementadas devem considerar o tipo do sistema e o impacto em seu desempenho. Por exemplo, um deadlock em um sistema de tempo real, que controla uma usina nuclear, deve ser tratado com mecanismos voltados para esse tipo de aplicação, diferentes dos adotados por um sistema de tempo compartilhado comum.

A seguir, são examinados os principais mecanismos de prevenção, detecção e correção de deadlocks.

7.10.1 Prevenção de Deadlock

Para prevenir a ocorrência de deadlocks, é preciso garantir que uma das quatro condições apresentadas, necessárias para sua existência, nunca se satisfaça.

A ausência da primeira condição (exclusão mútua) certamente acaba com o problema do deadlock, pois nenhum processo terá que esperar para ter acesso a um recurso, mesmo que já esteja sendo utilizado por outro processo. Os problemas decorrentes do compartilhamento de recursos entre processos concorrentes sem a exclusão mútua já foram apresentados, inclusive com a exemplificação das sérias inconsistências geradas.

Para evitar a segunda condição (espera por recurso), processos que já possuam recursos garantidos não devem requisitar novos recursos. Uma maneira de implementar esse mecanismo de prevenção é que, antes do início da execução, um processo deve pré-alocar todos os recursos necessários. Nesse caso, todos os recursos necessários ao processo devem estar disponíveis para o início da execução, caso contrário nenhum recurso será alocado e o processo permanecerá aguardando. Esse mecanismo pode produzir um grande desperdício na utilização dos recursos do sistema, pois um recurso pode permanecer alocado por um grande período de tempo, sendo utilizado apenas por um breve momento. Outra dificuldade, decorrente desse mecanismo, é a determinação do número de recursos que um processo deve alocar antes da sua execução. O mais grave, no entanto, é a possibilidade de um processo sofrer starvation caso os recursos necessários à sua execução nunca estejam disponíveis ao mesmo tempo.

A terceira condição (não preempção) pode ser evitada quando é permitido que um recurso seja retirado de um processo no caso de outro processo necessitar do mesmo recurso. A liberação de recursos já garantidos por um processo pode ocasionar sérios problemas, podendo até impedir a continuidade de execução do processo. Outro problema desse mecanismo é a possibilidade de um processo sofrer starvation, pois, após garantir os recursos necessários à sua execução, um processo pode ter que liberá-los sem concluir seu processamento.

A última maneira de evitar um deadlock é excluir a possibilidade da quarta condição (espera circular). Uma forma de implementar esse mecanismo é forçar o processo a ter apenas um recurso por vez. Caso o processo necessite de outro recurso, o recurso já alocado deve ser liberado. Esta condição restringiria muito o grau de compartilhamento e o processamento de programas.

A prevenção de deadlocks evitando-se a ocorrência de qualquer uma das quatro condições é bastante limitada e, por isso, não é utilizada na prática. É possível evitar o deadlock mesmo se todas as condições necessárias à sua ocorrência estejam presentes. A solução mais conhecida para essa situação é o algoritmo do banqueiro (Banker's Algorithm) proposto por Dijkstra (1965). Basicamente, este algoritmo exige que os processos informem o número máximo de cada tipo de recurso necessário à sua execução. Com essas informações, é possível definir o estado de alocação de um recurso, que é a relação entre o número de recursos alocados e disponíveis e o número máximo de processos que necessitam desses recursos. Com base nessa relação, o sistema pode fazer a alocação dos recursos de forma segura e evitar a ocorrência de deadlocks. Muitos livros abordam esse algoritmo com detalhes, e ele pode ser encontrado em Tanenbaum (2001).

Apesar de evitar o problema do deadlock, essa solução possui também várias limitações. A maior delas é a necessidade de um número fixo de processos ativos e de recursos disponíveis no sistema. Essa limitação impede que a solução seja implementada na prática, pois é muito difícil prever o número de usuários no sistema e o número de recursos disponíveis.

7.10.2 Detecção do Deadlock

Em sistemas que não possuam mecanismos que previnam a ocorrência de deadlocks, é necessário um esquema de detecção e correção do problema. A *detecção do deadlock* é o mecanismo que determina, realmente, a existência da situação de deadlock, permitindo identificar os recursos e processos envolvidos no problema.

Para detectar deadlocks, os sistemas operacionais devem manter estruturas de dados capazes de identificar cada recurso do sistema, o processo que o está alocando e os processos que estão à espera da liberação do recurso. Toda vez que um recurso é alocado ou liberado por um processo, a estrutura deve ser atualizada. Geralmente, os algoritmos que implementam esse mecanismo verificam a existência da espera circular, percorrendo toda a estrutura sempre que um processo solicita um recurso e ele não pode ser imediatamente garantido.

Dependendo do tipo de sistema, o ciclo de busca por um deadlock pode variar. Em sistemas de tempo compartilhado, o tempo de busca pode ser maior, sem comprometer o desempenho e a confiabilidade do sistema. Sistemas de tempo real, por sua vez, devem constantemente certificar-se da ocorrência de deadlocks, porém a maior segurança gera maior overhead.

7.10.3 Correção do Deadlock

Após a detecção do deadlock, o sistema operacional deverá de alguma forma corrigir o problema. Uma solução bastante utilizada pela maioria dos sistemas é, simplesmente, eliminar um ou mais processos envolvidos no deadlock e desalocar os recursos já garantidos por eles, quebrando, assim, a espera circular.

A eliminação dos processos envolvidos no deadlock e, consequentemente, a liberação de seus recursos podem não ser simples, dependendo do tipo do recurso envolvido. Se um processo estiver atualizando um arquivo ou imprimindo uma listagem, o sistema deve garantir que esses recursos sejam liberados sem problemas. Os processos eliminados não têm como ser recuperados, porém outros processos, que antes estavam em deadlock, poderão prosseguir a execução.

A escolha do processo a ser eliminado é feita, normalmente, de forma aleatória ou, então, com base em algum tipo de prioridade. Este esquema, porém, pode consumir considerável tempo do processador, ou seja, gerar elevado overhead ao sistema.

Uma solução menos drástica envolve a liberação de apenas alguns recursos alocados aos processos para outros processos, até que o ciclo de espera termine. Para que esta solução seja realmente eficiente, é necessário que o sistema possa suspender um processo, liberar seus recursos e, após a solução do problema, retornar à execução do processo, sem perder o processamento já realizado. Este mecanismo é conhecido como *rollback* e, além do overhead gerado, é muito difícil de ser implementado, por ser bastante dependente da aplicação que está sendo processada.

► 7.11 Exercícios

- Defina o que é uma aplicação concorrente e dê um exemplo de sua utilização.
- Considere uma aplicação que utilize uma matriz na memória principal para a comunicação entre vários processos concorrentes. Que tipo de problema pode ocorrer quando dois ou mais processos acessam uma mesma posição da matriz?
- O que é exclusão mútua e como é implementada?
- Como seria possível resolver os problemas decorrentes do compartilhamento da matriz, apresentado anteriormente, utilizando o conceito de exclusão mútua?
- O que é starvation e como podemos solucionar esse problema?
- Qual o problema com a solução que desabilita as interrupções para implementar a exclusão mútua?
- O que é espera ocupada e qual o seu problema?
- Explique o que é sincronização condicional e dê um exemplo de sua utilização.
- Explique o que são semáforos e dê dois exemplos de sua utilização: um para a solução da exclusão mútua e outro para a sincronização condicional.
- Apresente uma solução para o problema dos filósofos que permita que os cinco pensadores sentem à mesa, porém evite a ocorrência de starvation e deadlock.
- Explique o que são monitores e dê dois exemplos de sua utilização: um para a solução da exclusão mútua e outro para a sincronização condicional.
- Qual a vantagem da forma assíncrona de comunicação entre processos e como esta pode ser implementada?
- O que é deadlock, quais as condições para obtê-lo e quais as soluções possíveis?
- Em uma aplicação concorrente que controla saldo bancário em contas-correntes, dois processos compartilham uma região de memória onde estão armazenados os saldos dos clientes A e B. Os processos executam concorrentemente os seguintes passos:

Processo 1 (Cliente A)

```
/* saque em A */
1a. x := saldo_do_cliente_A;
1b. x := x - 200;
1c. saldo_do_cliente_A := x;

/* deposito em B */
1d. x := saldo_do_cliente_B;
1e. x := x + 100;
1f. saldo_do_cliente_B := x;
```

Processo 2 (Cliente B)

```
/*saque em A */
2a. y := saldo_do_cliente_A;
2b. y := y - 100;
2c. saldo_do_cliente_A := y;

/* deposito em B */
2d. y := saldo_do_cliente_B;
2e. y := y + 200;
2f. saldo_do_cliente_B := y;
```

Supondo que os valores dos saldos de A e B sejam, respectivamente, 500 e 900, antes de os

processos executarem, pede-se:

- a) Quais os valores corretos esperados para os saldos dos clientes A e B após o término da execução dos processos?
- b) Quais os valores finais dos saldos dos clientes se a sequência temporal de execução das operações for: 1a, 2a, 1b, 2b, 1c, 2c, 1d, 2d, 1e, 2e, 1f, 2f?
- c) Utilizando semáforos, proponha uma solução que garanta a integridade dos saldos e permita o maior compartilhamento possível dos recursos entre os processos, não esquecendo a especificação da inicialização dos semáforos.

15. O problema dos leitores/escritores, apresentado a seguir, consiste em sincronizar processos que consultam/atualizam dados em uma base comum. Pode haver mais de um leitor lendo ao mesmo tempo; no entanto, enquanto um escritor está atualizando a base, nenhum outro processo pode ter acesso a ela (nem mesmo leitores).

```
VAR Acesso: Semaforo := 1;
Exclusao: Semaforo := 1;
Nleitores: integer := 0;

PROCEDURE Escritor;
BEGIN
ProduzDado;
DOWN (Acesso);
Escreve;
UP (Acesso);
END;

PROCEDURE Leitor;
BEGIN
DOWN (Exclusao);
Nleitores := Nleitores + 1;
IF ( Nleitores = 1 ) THEN DOWN (Acesso);
UP (Exclusao);
Leitura;
DOWN (Exclusao);
Nleitores := Nleitores - 1;
IF ( Nleitores = 0 ) THEN UP (Acesso);
UP (Exclusao);
ProcessaDado;
END;
```

- a) Suponha que exista apenas um leitor fazendo acesso à base. Enquanto este processo realiza a leitura, quais os valores das três variáveis?
- b) Chega um escritor enquanto o leitor ainda está lendo. Quais os valores das três variáveis após o bloqueio do escritor? Sobre qual(is) semáforo(s) se dá o bloqueio?
- c) Chega mais um leitor enquanto o primeiro ainda não acabou de ler e o escritor está bloqueado. Descreva os valores das três variáveis quando o segundo leitor inicia a leitura.
- d) Os dois leitores terminam simultaneamente a leitura. É possível haver problemas quanto à integridade do valor da variável Nleitores? Justifique.

- e) Descreva o que acontece com o escritor quando os dois leitores terminam suas leituras. Descreva os valores das três variáveis quando o escritor inicia a escrita.
- f) Enquanto o escritor está atualizando a base, chegam mais um escritor e mais um leitor. Sobre qual(is) semáforo(s) eles ficam bloqueados? Descreva os valores das três variáveis após o bloqueio dos recém-chegados.
- g) Quando o escritor tiver terminado a atualização, é possível prever qual dos processos bloqueados (leitor ou escritor) terá acesso primeiro à base?
- h) Descreva uma situação onde os escritores sofram starvation (adiamento indefinido).



Parte III

Gerência de Recursos

“O que sabemos é uma gota, o que ignoramos é um oceano.”

Isaac Newton (1642-1727)
Físico e matemático inglês

“Senhor, dai-nos coragem para mudar as coisas que podem ser mudadas, paciência para aceitar as imutáveis e sabedoria para distinguir umas das outras.”

Abraham Lincoln (1809-1865)
Presidente americano

“A educação seria muito mais eficaz se sua finalidade fosse assegurar que cada estudante, ao sair da escola, tivesse a ciência do quanto ainda não sabe e imbuída com um desejo de saber.”

William Haley (1901-1987)
Profissional de televisão americano

“Somente os instruídos são livres.”

Epicteto (50 a.C.-130 a.C.)
Filósofo frígio

“Quem abre uma escola, fecha uma prisão.”

Victor Hugo (1802-1885)
Escritor e poeta francês

*“A leitura faz o homem completo.
A história torna-o sábio e prudente;*

a poesia, espiritual;

a matemática, útil;

a filosofia, profundo;

a moral, grave;

a lógica e a retórica, apto para discutir.

Ler é conversar com os sábios.”

Francis Bacon (1561-1626)



8

Gerência do Processador

► 8.1 Introdução

Com o surgimento dos sistemas multiprogramáveis, nos quais múltiplos processos poderiam permanecer na memória principal compartilhando o uso da UCP, a gerência do processador tornou-se uma das atividades mais importantes em um sistema operacional. A partir do momento em que diversos processos podem estar no estado de pronto, critérios devem ser estabelecidos para determinar qual processo será escolhido para fazer uso do processador. Os critérios utilizados para esta seleção compõem a chamada *política de escalonamento*, que é a base da gerência do processador e da multiprogramação em um sistema operacional.

Neste capítulo serão abordadas as funções básicas de escalonamento, políticas e algoritmos, descrevendo os mecanismos existentes e implementados na gerência do processador.

► 8.2 Funções Básicas

A política de escalonamento de um sistema operacional tem diversas funções básicas, como a de manter o processador ocupado a maior parte do tempo, balancear o uso da UCP entre processos, privilegiar a execução de aplicações críticas, maximizar o throughput do sistema e oferecer tempos de resposta razoáveis para usuários interativos. Cada sistema operacional possui sua política de escalonamento adequada ao seu propósito e às suas características. Sistemas de tempo compartilhado, por exemplo, têm requisitos de escalonamento distintos dos sistemas de tempo real.

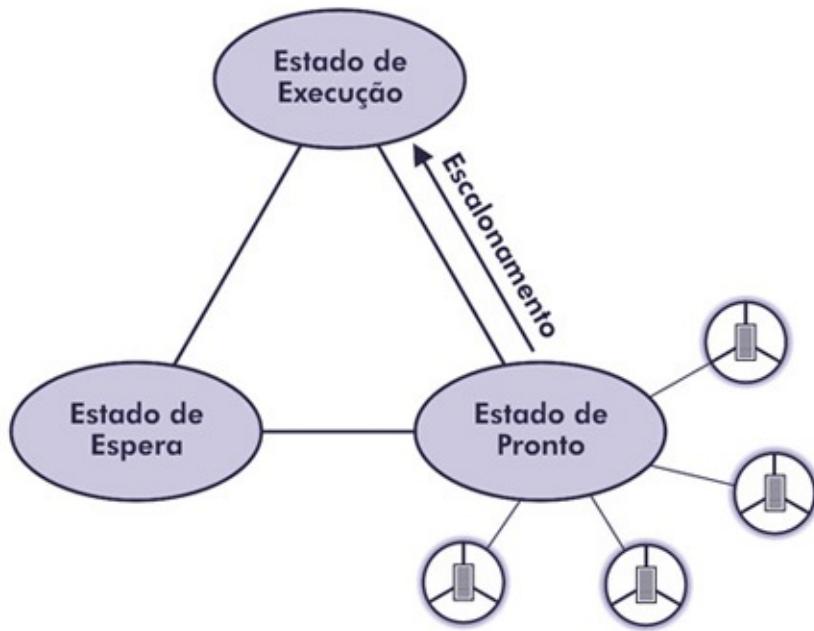


Fig. 8.1 Escalonamento.

A rotina do sistema operacional que tem como principal função implementar os critérios da política de escalonamento é denominada *escalonador (scheduler)*. Em um sistema multiprogramável, o escalonador é fundamental, pois todo o compartilhamento do processador é dependente desta rotina.

Outra rotina importante na gerência do processador é conhecida como *dispatcher*, responsável pela troca de contexto dos processos após o escalonador determinar qual processo deve fazer uso do processador. O período de tempo gasto na substituição de um processo em execução por outro é denominado latência do dispatcher.

Em ambientes que implementam apenas processos, o escalonamento é realizado com base nos processos prontos para execução. Em sistemas que implementam threads, o escalonamento é realizado considerando os threads no estado de pronto, independentemente do processo. Neste caso, podemos considerar o processo como a unidade de alocação de recursos, enquanto o thread é a unidade de escalonamento. No decorrer do capítulo, utilizaremos o termo processo de forma genérica como o elemento selecionado pelo escalonador.

► 8.3 Critérios de Escalonamento

As características de cada sistema operacional determinam quais são os principais aspectos para a implementação de uma política de escalonamento adequada. Por exemplo, sistemas de tempo compartilhado exigem que o escalonamento trate todos os processos de forma igual, evitando, assim, a ocorrência de starvation, ou seja, que um processo fique indefinidamente esperando pela utilização do processador. Já em sistemas de tempo real, o escalonamento deve priorizar a execução de processos críticos em detrimento da execução de outros processos.

A seguir são apresentados os principais critérios que devem ser considerados em uma política de escalonamento:

■ Utilização do processador

Na maioria dos sistemas é desejável que o processador permaneça a maior parte do seu tempo ocupado. Uma utilização na faixa de 30% indica um sistema com uma carga de processamento baixa, enquanto na faixa de 90% indica um sistema bastante carregado, próximo da sua capacidade máxima.

■ Throughput

Throughput representa o número de processos executados em um determinado intervalo de tempo. Quanto maior o throughput, maior o número de tarefas executadas em função do tempo. A maximização do throughput é desejada na maioria dos sistemas.

■ Tempo de Processador / Tempo de UCP

Tempo de processador ou *tempo de UCP* é o tempo que um processo leva no estado de execução durante seu processamento. As políticas de escalonamento não influenciam o tempo de processador de um processo, sendo este tempo função apenas do código da aplicação e da entrada de dados.

■ Tempo de Espera

Tempo de espera é o tempo total que um processo permanece na fila de pronto durante seu processamento, aguardando para ser executado. A redução do tempo de espera dos processos é desejada pela maioria das políticas de escalonamento.

■ Tempo de Turnaround

Tempo de turnaround é o tempo que um processo leva desde a sua criação até seu término, levando em consideração todo o tempo gasto na espera para alocação de memória, espera na fila de pronto (tempo de espera), processamento na UCP (tempo de processador) e na fila de espera, como nas operações de E/S. As políticas de escalonamento buscam minimizar o tempo de turnaround.

■ Tempo de Resposta

Tempo de resposta é o tempo decorrido entre uma requisição ao sistema ou à aplicação e o instante em que a resposta é exibida. Em sistemas interativos, podemos entender como o tempo decorrido entre a última tecla digitada pelo usuário e o início da exibição do resultado no monitor. Em geral, o tempo de resposta não é limitado pela capacidade de processamento do sistema computacional, mas pela velocidade dos dispositivos de E/S. Em sistemas interativos, como aplicações on-line ou acesso à Web, os tempos de resposta devem ser da ordem de poucos segundos.

De maneira geral, qualquer política de escalonamento busca otimizar a utilização do processador e o throughput, enquanto tenta diminuir os tempos de turnaround, espera e resposta. Apesar disso, as funções que uma política de escalonamento deve possuir são muitas vezes conflitantes. Dependendo do tipo de sistema operacional, um critério pode ter maior importância do que outros, como nos sistemas interativos, onde o tempo de resposta tem grande relevância.

► 8.4 Escalonamentos Não Preemptivos e Preemptivos

As políticas de escalonamento podem ser classificadas segundo a possibilidade de o sistema operacional interromper um processo em execução e substituí-lo por um outro, atividade esta conhecida como *preempção*. Sistemas operacionais que implementam escalonamento com preempção são mais complexos, contudo possibilitam políticas de escalonamento mais flexíveis.

O *escalonamento não preemptivo* foi o primeiro tipo de escalonamento implementado nos sistemas multiprogramáveis, onde predominava tipicamente o processamento batch. Nesse tipo de escalonamento, quando um processo está em execução nenhum evento externo pode ocasionar a perda do uso do processador. O processo somente sai do estado de execução caso termine seu processamento ou execute instruções do próprio código que ocasionem uma mudança para o estado de espera.

No *escalonamento preemptivo*, o sistema operacional pode interromper um processo em execução e passá-lo para o estado de pronto, com o objetivo de alocar outro processo na UCP. Com o uso da preempção, é possível ao sistema priorizar a execução de processos, como no caso de aplicações de tempo real, em que o fator tempo é crítico. Outro benefício é a possibilidade de implementar políticas de escalonamento que compartilhem o processador de uma maneira mais uniforme, distribuindo de forma balanceada o uso da UCP entre os processos.

Atualmente, a maioria dos sistemas operacionais implementa políticas de escalonamento preemptivas que, apesar de tornarem os sistemas mais complexos, possibilitam a implementação dos diversos critérios de escalonamento apresentados.

► 8.5 Escalonamento First-In-First-Out (FIFO)

No *escalonamento first-in-first-out (FIFO scheduling)*, também conhecido como *first-come-first-served (FCFS scheduling)*, o processo que chegar primeiro ao estado de pronto é o selecionado para execução. Este algoritmo é bastante simples, sendo necessária apenas uma fila, onde os processos que passam para o estado de pronto entram no seu final e são escalonados quando chegam ao seu início. Quando um processo vai para o estado de espera, o primeiro processo da fila de pronto é escalonado. Todos os processos quando saem do estado de espera entram no final da fila de pronto (Fig. 8.2).

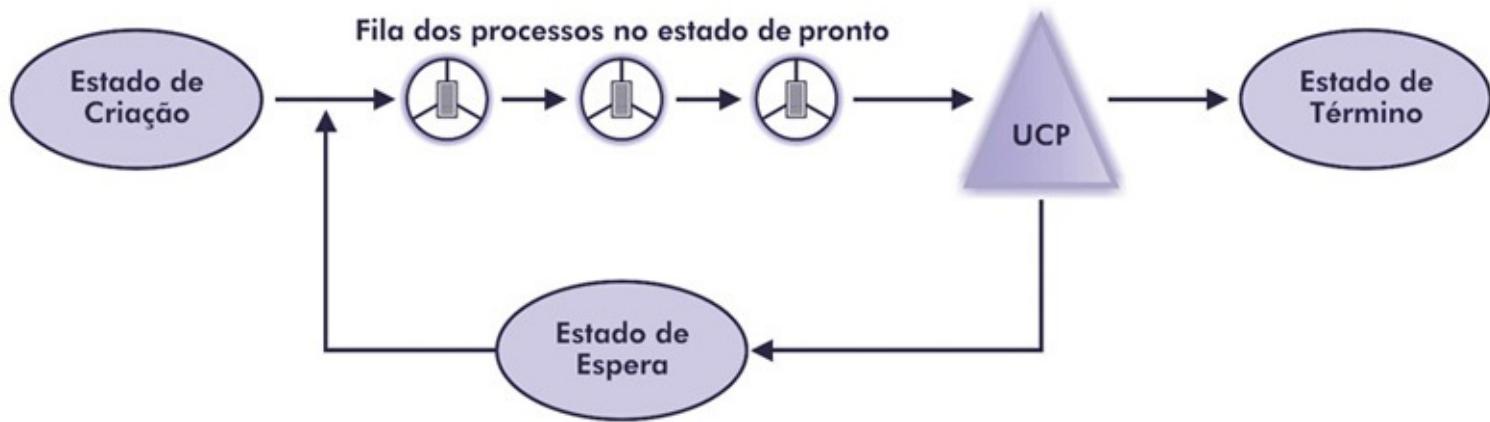


Fig. 8.2 Escalonamento FIFO.

Na Fig. 8.3 é possível comparar o uso do escalonamento FIFO em duas situações distintas, em que os processos A, B e C são criados no instante de tempo 0, com os tempos de processador 10, 4 e 3, respectivamente. A diferença entre os exemplos da Fig. 8.3 é o posicionamento dos processos na fila de pronto. O tempo médio de espera dos três processos no exemplo (a) é igual a $(0 + 10 + 14)/3 = 8$ u.t., enquanto no exemplo (b) é de aproximadamente $(7 + 0 + 4)/3 = 3,7$ u.t. A diferença entre as médias é bastante significativa, justificada pela diferença dos tempos de processador entre os processos.

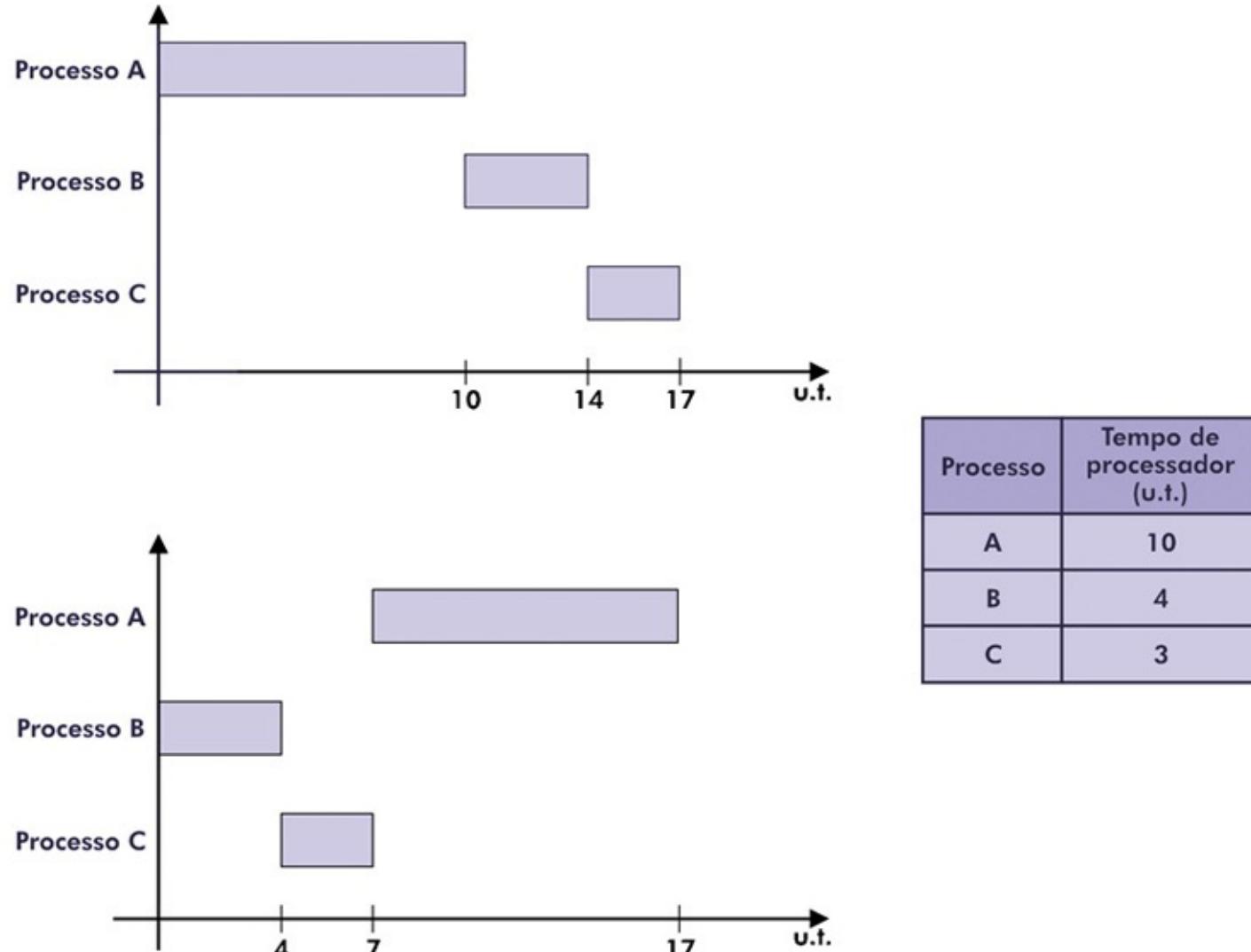


Fig. 8.3 Escalonamento FIFO (exemplo).

Apesar de simples, o escalonamento FIFO apresenta algumas deficiências. O principal problema é a impossibilidade de prever-se quando um processo terá sua execução iniciada, já que isso varia em função do tempo de execução dos demais processos posicionados à sua frente na fila de pronto. Como vimos no exemplo da Fig. 8.3, o algoritmo de escalonamento não se preocupa em melhorar o tempo médio de espera dos processos, utilizando apenas a ordem de chegada dos processos à fila de pronto. Este problema pode ser mais bem percebido nos tempos de turnaround dos processos que demandam menor tempo de UCP.

Outro problema neste tipo de escalonamento é que processos CPU-bound levam vantagem no uso do processador sobre processos I/O-bound. No caso de existirem processos I/O-bound mais importantes do que os CPU-bound, não é possível tratar este tipo de diferença.

O escalonamento FIFO é do tipo não preemptivo e foi inicialmente implementado em sistemas monoprogramáveis com processamento batch, sendo ineficiente, se aplicado da forma original em sistemas interativos de tempo compartilhado. Atualmente, sistemas de tempo compartilhado utilizam o escalonamento FIFO com variações, permitindo, assim, parcialmente sua implementação.

► 8.6 Escalonamento Shortest-Job-First (SJF)

No *escalonamento shortest-job-first (SJF scheduling)*, também conhecido como *shortest-process-next (SPN scheduling)*, o algoritmo de escalonamento seleciona o processo que tiver o menor tempo de processador ainda por executar. Dessa forma, o processo em estado de pronto que necessitar de menos tempo de UCP para terminar seu processamento é selecionado para execução.

Na Fig. 8.4 é mostrado como ficaria o escalonamento utilizando o algoritmo SJF a partir dos mesmos processos utilizados no exemplo da Fig. 8.3. O tempo médio de espera dos três processos, igual a $(7 + 3 + 0)/3 = 3,3$ u.t., é inferior ao apresentado no escalonamento FIFO.

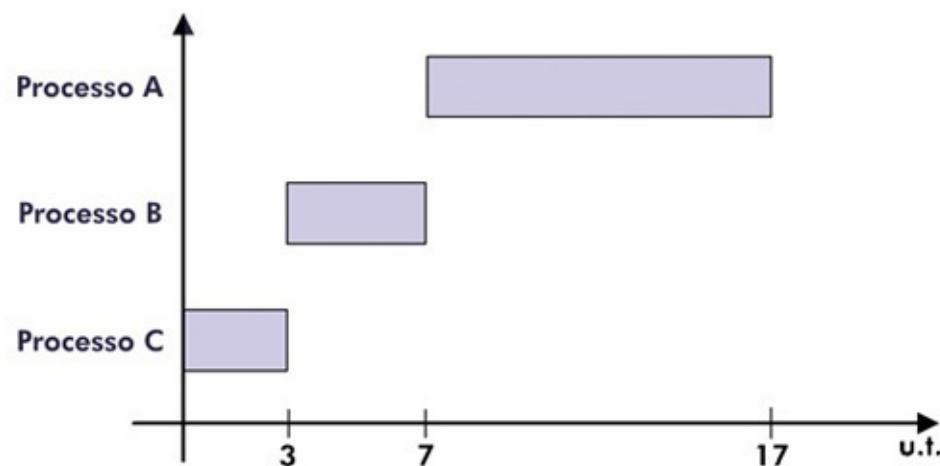


Fig. 8.4 Escalonamento SJF (exemplo).

Esta implementação foi utilizada nos primeiros sistemas operacionais com processamento exclusivamente batch. Para cada novo processo admitido no sistema, um tempo de processador era associado ao seu contexto de software. Como não é possível precisar o tempo de processador previamente para cada processo, uma estimativa era realizada com base em análises estatísticas de execuções passadas dos programas. O tempo estimado era informado ao sistema operacional, que o utilizava como no algoritmo de escalonamento. Caso o tempo informado fosse muito inferior ao tempo real, o sistema operacional poderia interromper a execução do processo. O principal problema nesta implementação é a impossibilidade de estimar o tempo de processador para processos interativos, já que a entrada de dados é uma ação imprevisível.

Uma maneira de implementar o escalonamento SJF em sistemas interativos foi considerar o comportamento do processo neste ambiente. A característica de um processo interativo é ser escalonado, executar algumas poucas instruções e fazer uma operação de E/S. Após o término da operação, o processo é posteriormente escalonado, executa mais algumas instruções e faz outra operação de E/S. Este comportamento se repete até o término do processamento. Neste caso, o escalonamento é realizado com

base no tempo que um processo irá utilizar a UCP na próxima vez em que for escalonado, e não mais no tempo total de processador que utilizará até o término do seu processamento.

Um problema existente nesta implementação é não ser possível ao sistema operacional saber quanto tempo um processo permanecerá utilizando a UCP na próxima vez em que for escalonado, contudo é possível prever o tempo com base no seu comportamento passado. O tempo pode ser estimado com base na média exponencial dos tempos passados, onde t_n é o tempo do processador utilizado no último escalonamento e δ_{n+1} o valor estimado que será utilizado no escalonamento seguinte:

$$\delta_{n+1} = \alpha t_n + (1 - \alpha) \delta_n$$

O valor de t_n representa a informação mais recente, enquanto δ_n possui o valor histórico. Com a determinação do valor da constante α ($0 < \alpha < 1$) é possível ponderar uma maior importância na estimativa para os valores recentes ou passados. Um valor de $\alpha = 1/2$ considera os valores recentes e passados com o mesmo peso.

Na sua concepção inicial, o escalonamento SJF é um escalonamento não preemptivo. Sua vantagem sobre o escalonamento FIFO está na redução do tempo médio de turnaround dos processos, porém no SJF é possível haver starvation para processos com tempo de processador muito longo ou do tipo CPU-bound.

Uma implementação do escalonamento SJF com preempção é conhecida como *escalonamento shortest remaining time (SRT scheduling)*. Nesta política, toda vez que um processo no estado de pronto tem um tempo de processador estimado menor do que o processo em execução o sistema operacional realiza uma preempção, substituindo-o pelo novo processo. De modo semelhante ao SJF, o sistema operacional deve ser o responsável por estimar os tempos de processador dos processos, e o risco de starvation continua presente.

► 8.7 Escalonamento Cooperativo

O *escalonamento cooperativo* é uma implementação que busca aumentar o grau de multiprogramação em políticas de escalonamentos que não possuam mecanismos de preempção, como o FIFO e o SJF não preemptivo. Neste caso, um processo em execução pode voluntariamente liberar o processador, retornando à fila de pronto e possibilitando que um novo processo seja escalonado, permitindo assim uma melhor distribuição no uso do processador.

A principal característica do escalonamento cooperativo está no fato de a liberação do processador ser uma tarefa realizada exclusivamente pelo processo em execução, que de uma maneira cooperativa libera a UCP para um outro processo. Neste mecanismo, o processo em execução verifica periodicamente uma fila de mensagens para determinar se existem outros processos na fila de pronto.

Como a interrupção do processo em execução não é responsabilidade do sistema operacional, algumas situações indesejadas podem ocorrer. No caso de um processo não verificar a fila de mensagens, os demais não terão chance de ser executados até a liberação da UCP pelo processo em execução. Isto pode ocasionar sérios problemas para a multiprogramação cooperativa, na medida em que um programa pode permanecer por um longo período de tempo alocando o processador.

Um exemplo deste tipo de escalonamento pode ser encontrado nos primeiros sistemas operacionais da família Microsoft Windows, sendo conhecido como multitarefa cooperativa.

► 8.8 Escalonamento Circular

O *escalonamento circular* (*round robin scheduling*) é um escalonamento do tipo preemptivo, projetado especialmente para sistemas de tempo compartilhado. Esse algoritmo é bastante semelhante ao FIFO, porém quando um processo passa para o estado de execução existe um tempo-limite para o uso contínuo do processador denominado *fatia de tempo* (*time-slice*) ou *quantum*.

No escalonamento circular, toda vez que um processo é escalonado para execução uma nova fatia de tempo é concedida. Caso a fatia de tempo expire, o sistema operacional interrompe o processo em execução, salva seu contexto e direciona-o para o final da fila de pronto. Esse mecanismo é conhecido como *preempção por tempo*.

A Fig. 8.5 ilustra o escalonamento circular, em que a fila de processos em estado de pronto é tratada como uma fila circular. O escalonamento é realizado alocando a UCP ao primeiro processo da fila de pronto. O processo permanecerá no estado de execução até que termine seu processamento, voluntariamente passe para o estado de espera ou que sua fatia de tempo expire, sofrendo, neste caso, uma preempção pelo sistema operacional. Após isso, um novo processo é escalonado com base na política de FIFO.

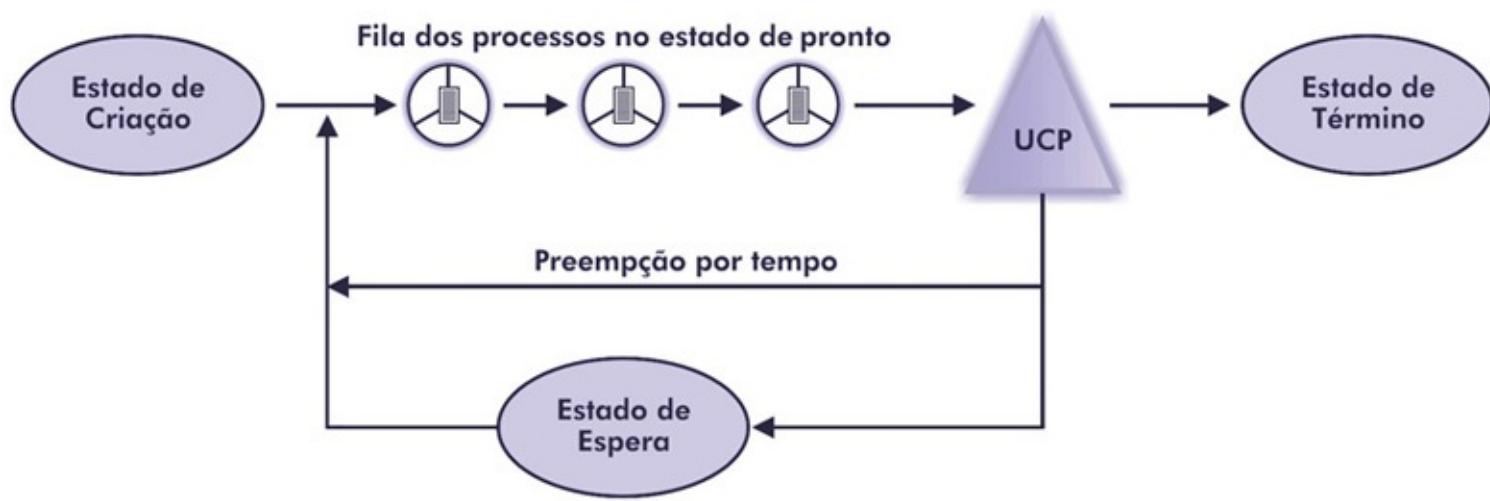


Fig. 8.5 Escalonamento circular.

A Fig. 8.6 exemplifica o escalonamento circular com três processos, em que a fatia de tempo é igual a 2 u.t. No exemplo não está sendo levado em consideração o tempo de latência do dispatcher, ou seja, o tempo de troca de contexto entre os processos.

O valor da fatia de tempo depende da arquitetura de cada sistema operacional e, em geral, varia entre 10 e 100 milissegundos. Este valor afeta diretamente o desempenho da política de escalonamento circular. Caso a fatia de tempo tenha um valor muito alto, este escalonamento tenderá a ter o mesmo

comportamento do escalonamento FIFO. Caso o valor do time slice seja pequeno, a tendência é que haja um grande número de preempções, o que ocasionaria excessivas mudanças de contexto, prejudicando o desempenho do sistema e afetando o tempo de turnaround dos processos.

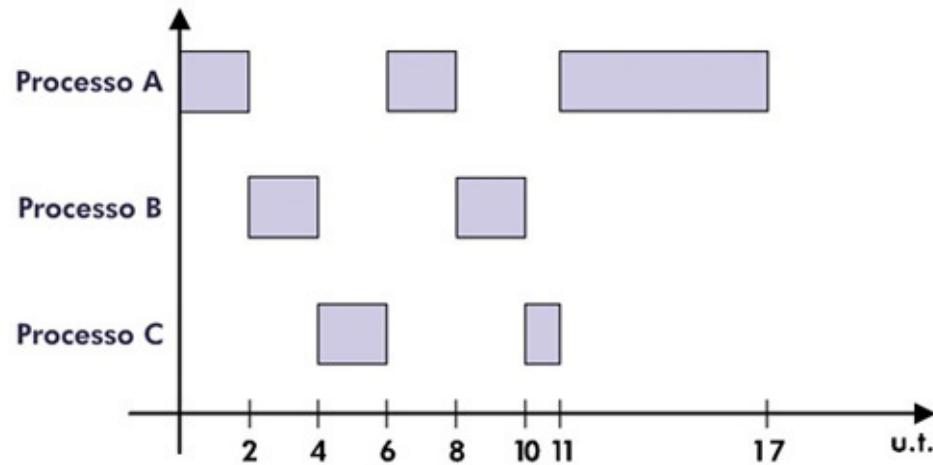


Fig. 8.6 Escalonamento circular (exemplo).

A principal vantagem do escalonamento circular é não permitir que um processo monopolize a UCP, sendo o tempo máximo alocado continuamente igual à fatia de tempo definido no sistema. No caso de sistemas de tempo compartilhado, onde existem diversos processos interativos concorrendo pelo uso do processador, o escalonamento circular é adequado.

Um problema presente nesta política é que processos CPU-bound são beneficiados no uso do processador em relação aos processos I/O-bound. Devido às suas características, os processos CPU-bound tendem a utilizar por completo a fatia de tempo, enquanto os processos I/O-bound têm mais chances de passar para o estado de espera antes de sofrerem preempção por tempo. Estas características distintas ocasionam um balanceamento desigual no uso do processador entre os processos.

Um refinamento do escalonamento circular, que busca reduzir este problema, é conhecido como *escalonamento circular virtual*, ilustrado na Fig. 8.7. Neste esquema, processos que saem do estado de espera vão para uma fila de pronto auxiliar. Os processos da fila auxiliar possuem preferência no escalonamento em relação à fila de pronto, e o escalonador só seleciona processos na fila de pronto quando a fila auxiliar estiver vazia. Quando um processo é escalonado a partir da fila auxiliar, sua fatia de tempo é calculada como sendo o valor da fatia de tempo do sistema menos o tempo de processador que o processo utilizou na última vez em que foi escalonado a partir da fila de pronto. Estudos comprovam que, apesar da maior complexidade na implementação, o balanceamento do uso do processador neste escalonamento é mais equilibrado.



Fig. 8.7 Escalonamento circular virtual.

Esse esquema em que o sistema operacional busca ajustar dinamicamente a ordem de escalonamento dos processos objetivando o balanceamento no uso do processador é denominado *mecanismo adaptativo*. Ao longo deste capítulo alguns outros exemplos deste tipo de mecanismo serão apresentados.

► 8.9 Escalonamento por Prioridades

O *escalonamento por prioridades* é um escalonamento do tipo preemptivo realizado com base em um valor associado a cada processo denominado *prioridade de execução*. O processo com maior prioridade no estado de pronto é sempre o escolhido para execução, e processos com valores iguais são escalonados seguindo o critério de FIFO. Neste escalonamento, o conceito de fatia de tempo não existe, consequentemente um processo em execução não pode sofrer preempção por tempo.

No escalonamento por prioridades, a perda do uso do processador só ocorrerá no caso de uma mudança voluntária para o estado de espera ou quando um processo de prioridade maior passa para o estado de pronto. Neste caso, o sistema operacional deverá interromper o processo corrente, salvar seu contexto e colocá-lo no estado de pronto. Esse mecanismo é conhecido como *preempção por prioridade*. Após isso, o processo de maior prioridade é escalonado.

A preempção por prioridade é implementada através de uma interrupção de clock, gerada em determinados intervalos de tempo, para que a rotina de escalonamento reavalie as prioridades dos processos no estado de pronto. Caso haja processos na fila de pronto com maior prioridade do que o processo em execução, o sistema operacional realiza a preempção.

A Fig. 8.8 ilustra o escalonamento por prioridades, no qual para cada prioridade existe uma fila de processos em estado de pronto que é tratada como uma fila circular. O escalonamento é realizado alocando o processador ao primeiro processo da fila de prioridade mais alta. O processo permanecerá no estado de execução até que termine seu processamento, voluntariamente passe para o estado de espera

ou sofra uma preempção devido a um processo de maior prioridade.

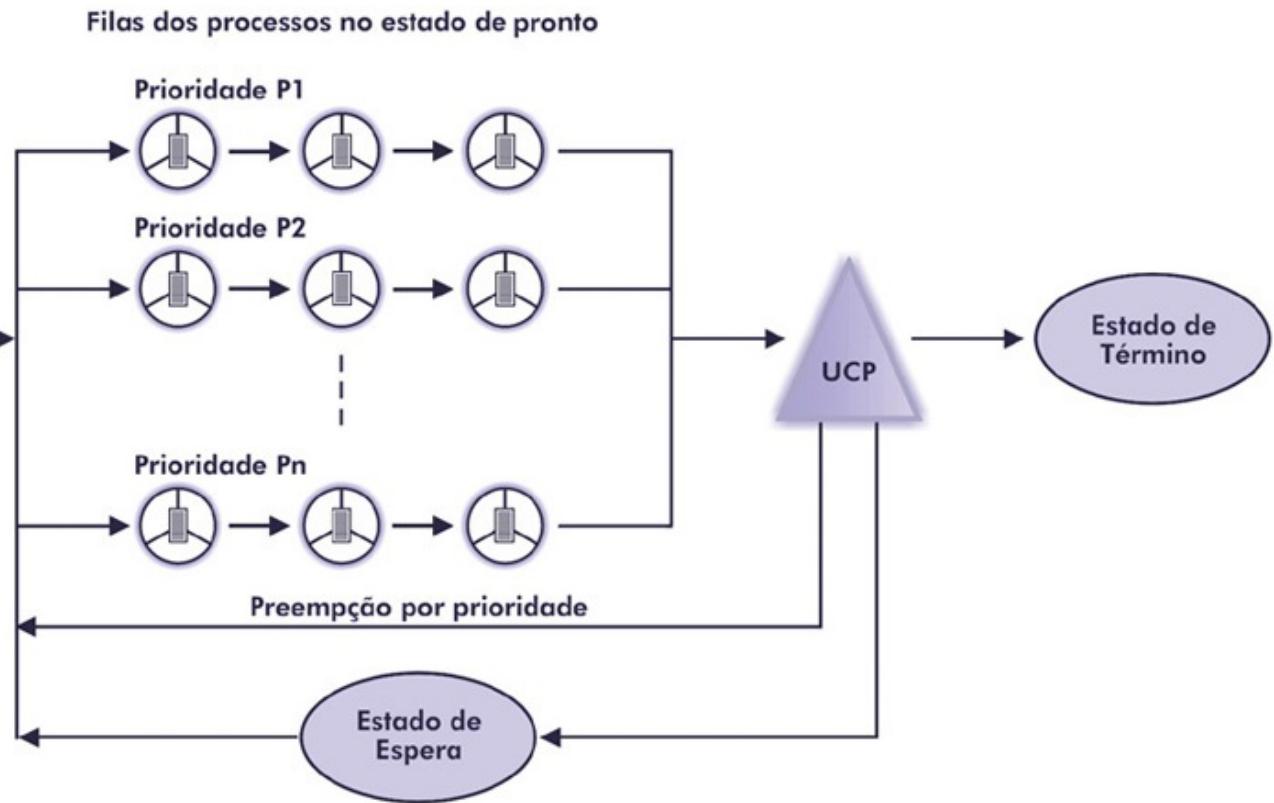


Fig. 8.8 Escalonamento por prioridades.

A Fig. 8.9 exemplifica o escalonamento por prioridades com três processos de prioridades diferentes, onde 5 é o valor mais alto de prioridade.

O escalonamento por prioridades também pode ser implementado de uma maneira não preemptiva. Neste caso, processos que passem para o estado de pronto com prioridade maior do que a do processo em execução não ocasionam preempção, sendo apenas colocados no início da fila de pronto.

Cada sistema operacional implementa sua faixa de valores para as prioridades de execução. Alguns sistemas associam as maiores prioridades a valores altos, enquanto outros sistemas utilizam valores baixos. No caso do OpenVMS, a prioridade do processo pode variar de 0 a 31, sendo 31 a maior prioridade. No IBM-AIX a prioridade varia de 0 a 127, porém os valores mais baixos possuem maior prioridade de execução.

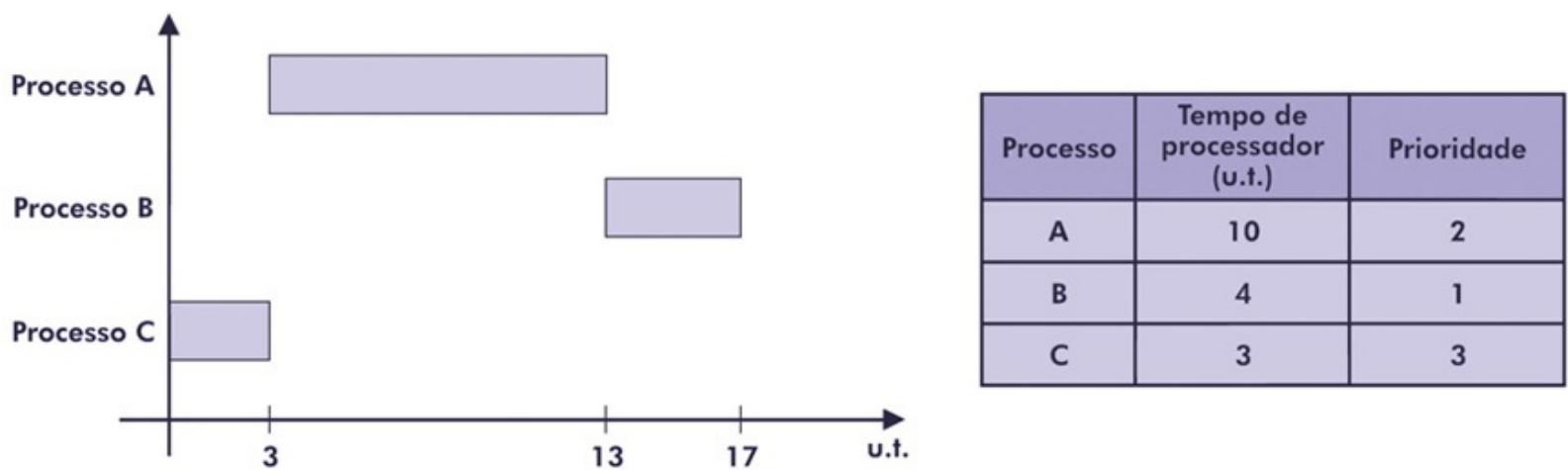


Fig. 8.9 Escalonamento por prioridades (exemplo).

A prioridade de execução é uma característica do contexto de software de um processo, e pode ser classificada como estática ou dinâmica. A *prioridade estática* não tem o seu valor alterado durante a existência do processo, enquanto a *prioridade dinâmica* pode ser ajustada de acordo com critérios definidos pelo sistema operacional. A possibilidade de alterar o valor da prioridade de um processo ao longo de seu processamento permite ajustar o critério de escalonamento em função do comportamento de cada processo no sistema.

Um dos principais problemas no escalonamento por prioridades é o starvation. Processos de baixa prioridade podem não ser escalonados, permanecendo indefinidamente na fila de pronto. Uma solução para este problema, possível em sistemas que implementam prioridade dinâmica, é a técnica de *aging*. Este mecanismo incrementa gradualmente a prioridade de processos que permanecem por muito tempo na fila de pronto.

O escalonamento por prioridades possibilita diferenciar os processos segundo critérios de importância. Com isso, processos de maior prioridade são escalonados preferencialmente. Isto é bastante útil, tanto em sistemas de tempo real e nas aplicações de controle de processo como também em aplicações de sistemas de tempo compartilhado, em que, às vezes, é necessário priorizar o escalonamento de determinados processos.

► 8.10 Escalonamento Circular com Prioridades

O *escalonamento circular com prioridades* implementa o conceito de fatia de tempo e de prioridade de execução associada a cada processo. Neste tipo de escalonamento, um processo permanece no estado de execução até que termine seu processamento, voluntariamente passe para o estado de espera ou sofra uma preempção por tempo ou prioridade (Fig. 8.10).

A principal vantagem deste escalonamento é permitir o melhor balanceamento no uso do processador em sistemas de tempo compartilhado. Processos com o perfil I/O-bound devem receber do administrador do sistema prioridades com valores maiores que os dos processos CPU-bound. Isso permite ao sistema operacional praticar uma política compensatória entre processos de perfis distintos, compartilhando o processador de forma mais igualitária. Este tipo de escalonamento é amplamente utilizado em sistemas de tempo compartilhado, como o Windows e o Unix.

O escalonamento circular com prioridades possui duas variações, dependendo se a prioridade é do tipo estática ou dinâmica. No *escalonamento circular com prioridades estáticas*, a prioridade definida no contexto de software de cada processo permanece inalterada ao longo da sua existência. No caso do *escalonamento circular com prioridades dinâmicas*, é possível que a prioridade de um processo seja alterada dinamicamente pelo administrador do sistema ou, em algumas políticas, pelo próprio sistema operacional.

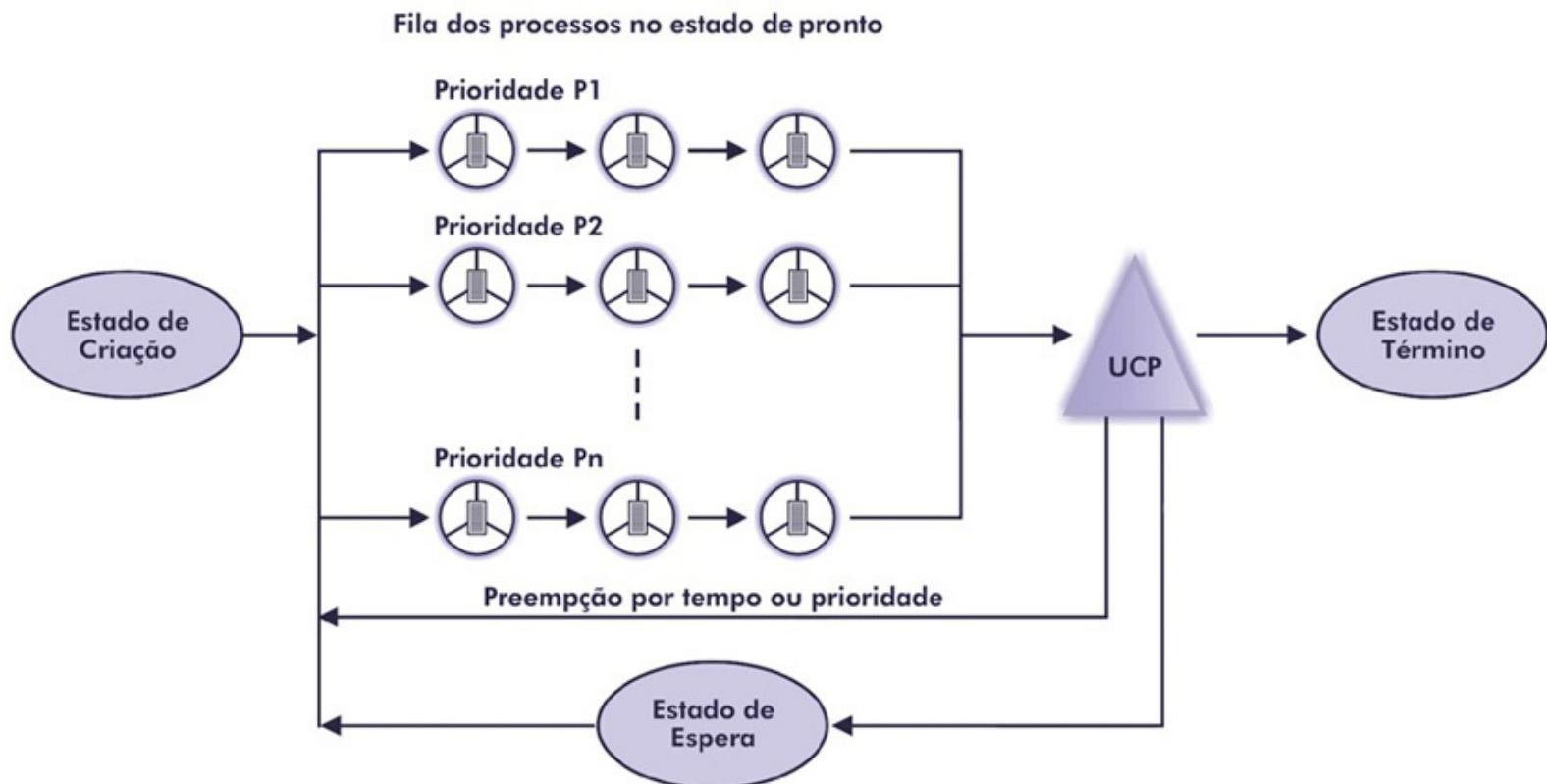


Fig. 8.10 Escalonamento circular com prioridades.

O ajuste dinâmico das prioridades dos processos pelo sistema operacional com base no seu comportamento é outro exemplo de um mecanismo adaptativo. Neste caso específico, o mecanismo possibilita um refinamento no balanceamento do uso do processador por processos de diferentes perfis com base no tipo de operação de E/S realizada. Algumas destas operações são mais lentas, fazendo com que um processo permaneça mais tempo no estado de espera sem chances de competir pelo uso do processador. Neste caso, quando o processo sai do estado de espera, um acréscimo temporário à sua prioridade é concedido pelo sistema operacional. Dessa forma, os processos têm mais chances de serem escalonados rapidamente, compensando parcialmente o tempo gasto no estado de espera. É importante perceber que os processos CPU-bound não são prejudicados com esta política, pois podem ser executados durante o período em que os processos I/O-bound estão no estado de espera.

► 8.11 Escalonamento por Múltiplas Filas

No *escalonamento por múltiplas filas (multilevel queue scheduling)* existem diversas filas de processos no estado de pronto, cada qual com uma prioridade específica. Os processos são associados às filas em função de características próprias, como importância para a aplicação, tipo de processamento ou área de

mémoria necessária.

Como processos possuem características de processamento distintas, é difícil que um único mecanismo de escalonamento seja adequado a todos. A principal vantagem de múltiplas filas é a possibilidade da convivência de mecanismos de escalonamento distintos em um mesmo sistema operacional. Cada fila possui um mecanismo próprio, permitindo que alguns processos sejam escalonados pelo mecanismo FIFO, enquanto outros pelo circular.

Neste mecanismo, o processo não possui prioridade, ficando esta característica associada à fila. O processo em execução sofre preempção caso um outro processo entre em uma fila de maior prioridade. O sistema operacional só pode escalar processos de uma determinada fila caso todas as outras filas de maior prioridade estejam vazias. Uma boa prática é classificar os processos em função do tipo de processamento realizado e associá-los adequadamente às respectivas filas.

Para exemplificarmos esse escalonamento, considere que os processos sejam divididos em três grupos: sistema, interativo e batch (Fig. 8.11). Os processos do sistema devem ser colocados em uma fila de prioridade mais alta que a dos outros processos, implementando um algoritmo de escalonamento baseado em prioridades. Os processos de usuários interativos devem estar em uma fila de prioridade intermediária, implementando o escalonamento circular. O mesmo mecanismo de escalonamento pode ser utilizado na fila de processos batch, com a diferença de que esta fila deve possuir uma prioridade mais baixa.

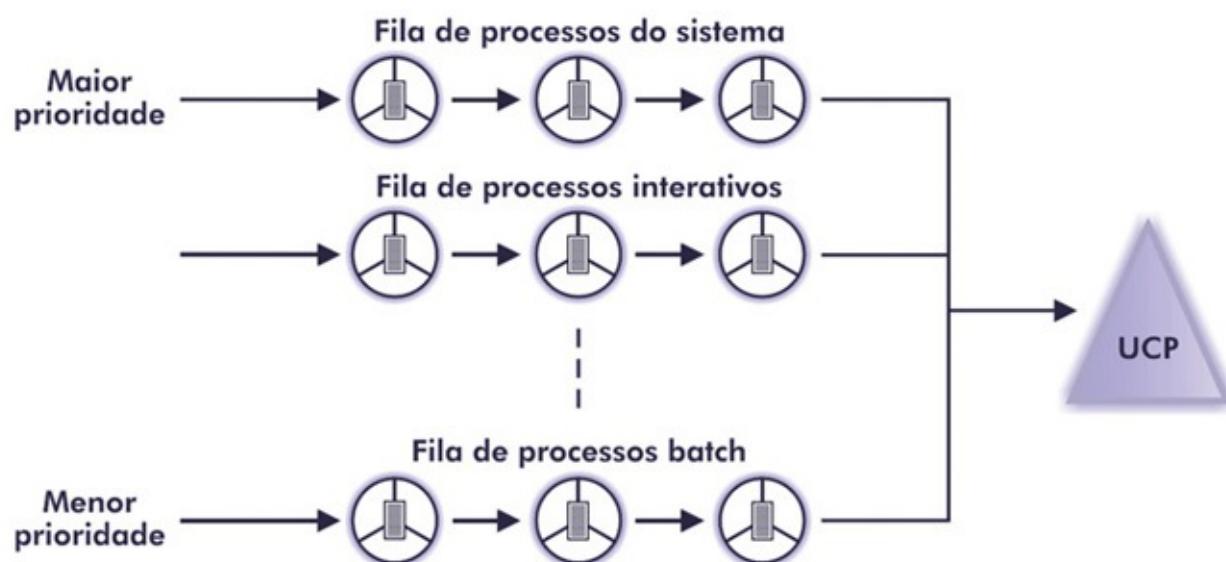


Fig. 8.11 Escalonamento por múltiplas filas.

Uma desvantagem deste escalonamento é que no caso de um processo alterar seu comportamento no decorrer do tempo o processo não poderá ser redirecionado para uma outra fila mais adequada. A associação de um processo à fila é determinada na criação do processo, permanecendo até o término do seu processamento.

► 8.12 Escalonamento por Múltiplas Filas com Realimentação

O escalonamento por múltiplas filas com realimentação (multilevel feedback queues scheduling) é semelhante ao escalonamento por múltiplas filas, porém os processos podem trocar de filas durante seu processamento. Sua grande vantagem é permitir ao sistema operacional identificar dinamicamente o comportamento de cada processo, direcionando-o para filas com prioridade de execução e mecanismo de escalonamento mais adequados ao longo de seu processamento.

Esse esquema permite que os processos sejam redirecionados entre as diversas filas, fazendo com que o sistema operacional implemente um mecanismo adaptativo. Os processos não são previamente associados às filas de pronto, e, sim, direcionados pelo sistema para as filas existentes com base no seu comportamento.

Um mecanismo FIFO adaptado com fatia de tempo é implementado para escalonamento em todas as filas, com exceção da fila de menor prioridade, que utiliza o escalonamento circular. O escalonamento de um processo em uma fila ocorre apenas quando todas as outras filas de prioridades mais altas estiverem vazias. A fatia de tempo em cada fila varia em função da sua prioridade, ou seja, quanto maior a prioridade da fila menor a sua fatia de tempo (Fig. 8.12). Assim, a fatia de tempo concedida aos processos varia em função da fila de pronto na qual ele se encontra. Um processo, quando criado, entra no final da fila de maior prioridade, porém durante sua execução, a cada preempção por tempo, o processo é redirecionado para uma fila de menor prioridade.

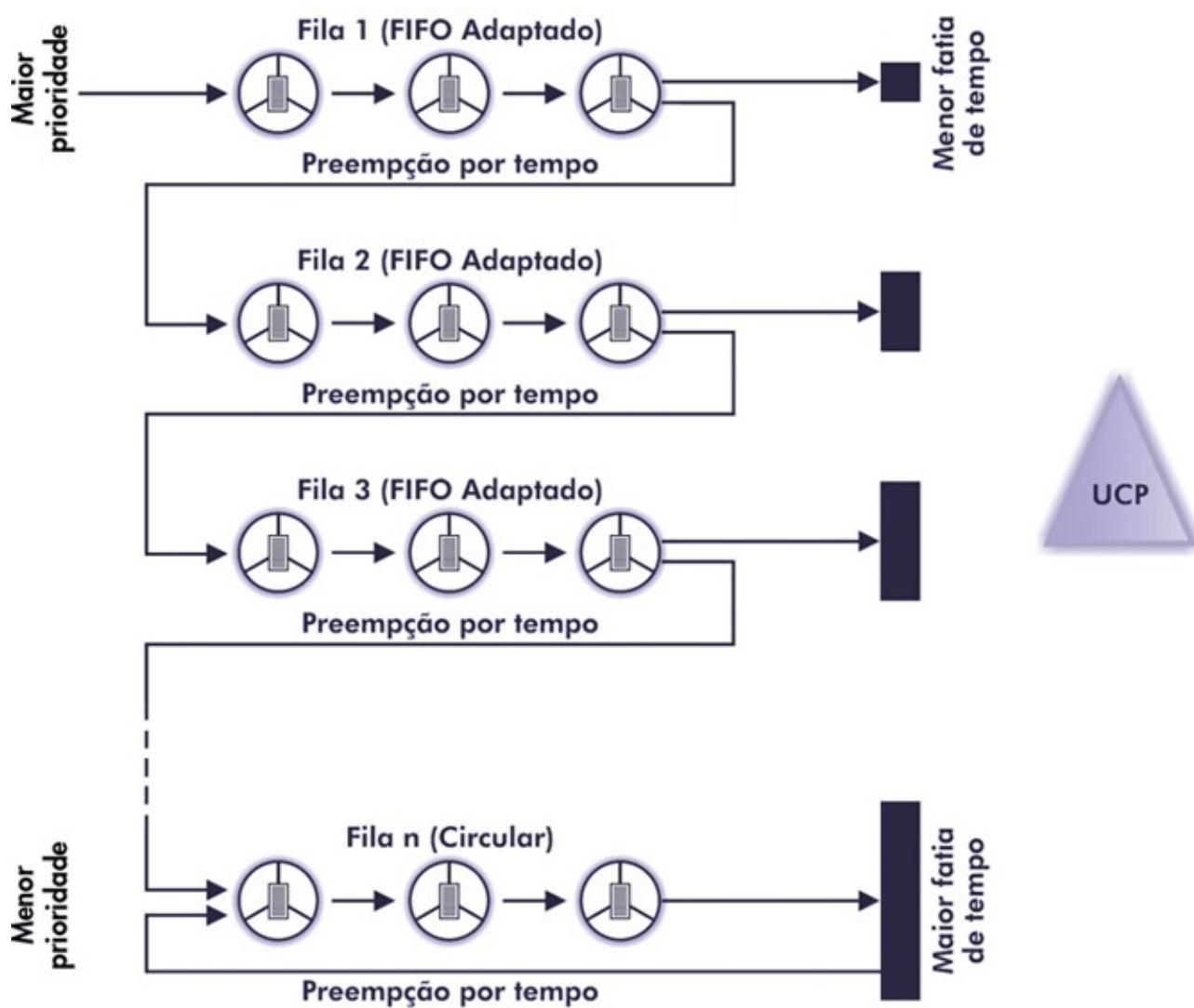


Fig. 8.12 Escalonamento por múltiplas filas com realimentação.

Esse escalonamento atende às necessidades dos diversos tipos de processos. No caso de processos I/O-bound, um tempo de resposta adequado é obtido, já que esses processos têm prioridades mais altas por permanecerem a maior parte do tempo nas filas de maior prioridade, pois dificilmente sofrerão preempção por tempo. Por outro lado, em processos CPU-bound a tendência é de que, ao entrar na fila de mais alta prioridade, o processo ganhe o processador, gaste sua fatia de tempo de tempo e seja direcionado para uma fila de menor prioridade. Dessa forma, quanto mais tempo do processador um processo utiliza, mais ele vai caindo para filas de menor prioridade.

O escalonamento por múltiplas filas com realimentação é um algoritmo de escalonamento generalista, podendo ser implementado em qualquer tipo de sistema operacional. Um dos problemas encontrados nesta política é que a mudança de comportamento de um processo CPU-bound para I/O-bound pode comprometer seu tempo de resposta. Outro aspecto a ser considerado é sua complexidade de implementação, ocasionando um grande overhead ao sistema.

► 8.13 Política de Escalonamento em Sistemas de Tempo Compartilhado

Em geral, sistemas de tempo compartilhado caracterizam-se pelo processamento interativo, no qual usuários interagem com as aplicações exigindo tempos de respostas baixos. A escolha de uma política de escalonamento para atingir este propósito deve levar em consideração o compartilhamento dos recursos de forma equitativa para possibilitar o uso balanceado da UCP entre processos. Neste item, serão analisados os comportamentos de um processo CPU-bound e outro I/O-bound, nos principais escalonamento apresentados.

Na Fig. 8.13, os processos A (CPU-bound) e B (I/O-bound) são escalonados segundo o mecanismo FIFO. Se contabilizarmos o uso da UCP para cada processo no instante de tempo 27, constataremos que o processador está sendo distribuído de forma bastante desigual (21 u.t. para o Processo A e 6 u.t. para o Processo B). Como a característica do Processo B é realizar muitas operações de E/S, em grande parte do tempo o processo permanecerá no estado de espera.

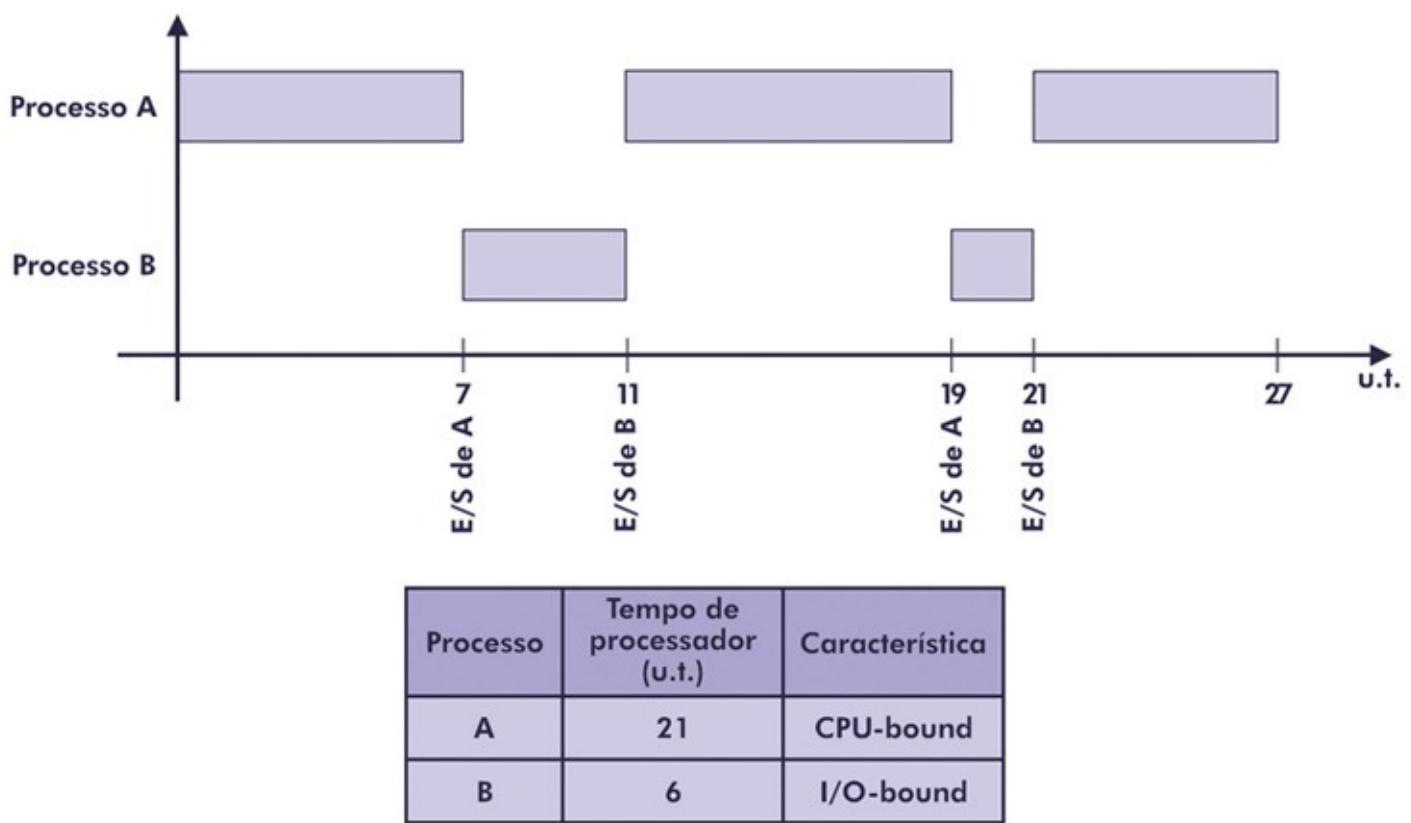


Fig. 8.13 Escalonamento FIFO (exemplo).

A Fig. 8.14 apresenta os mesmos processos A e B, só que aplicando o escalonamento circular com fatia de tempo igual a cinco unidades de tempo. Com isso, o escalonamento circular consegue melhorar a distribuição do tempo de processador (15 u.t. para o Processo A e 10 u.t. para o Processo B) em relação ao escalonamento FIFO, porém ainda não consegue implementar um compartilhamento equitativo entre os diferentes tipos de processos. Esta deficiência no escalonamento circular deve-se ao fato de todos os processos serem tratados de uma maneira igual, o que nem sempre é desejável.

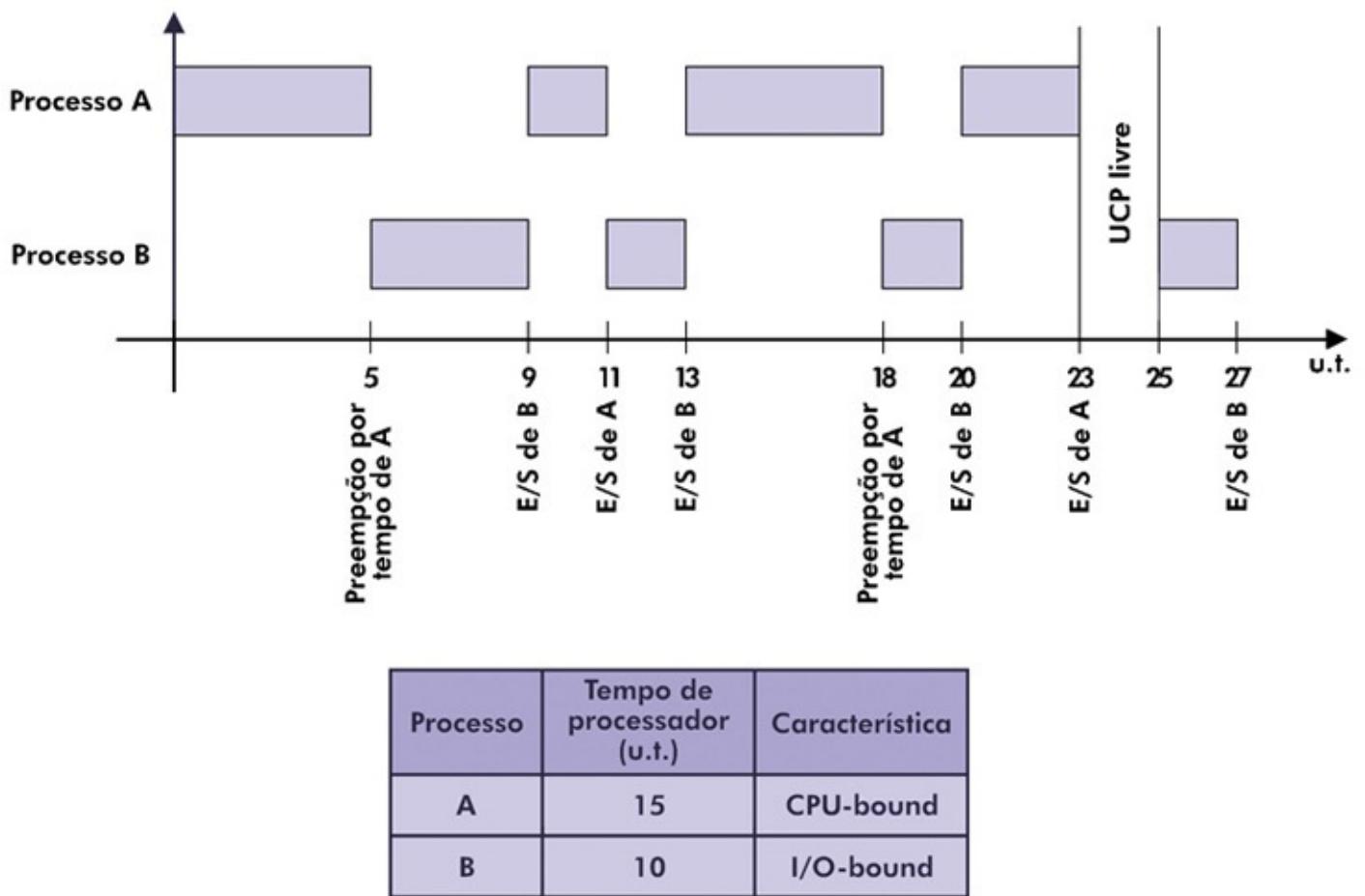


Fig. 8.14 Escalonamento circular (exemplo).

Em um escalonamento onde todos os processos são tratados igualmente, processos CPU-bound sempre levam vantagem sobre processos I/O-bound no uso do processador. Como no escalonamento circular um processo I/O-bound compete pelo processador da mesma forma que um processo CPU-bound e o processo I/O-bound passa a maior parte do tempo no estado de espera, o processo CPU-bound tem mais oportunidade de ser executado.

No escalonamento circular com prioridades é possível associar prioridades maiores aos processos I/O-bound, a fim de compensar o excessivo tempo gasto por este tipo de processo no estado de espera. No exemplo da Fig. 8.15, o Processo B tem uma prioridade superior à do Processo A, obtendo-se, desta forma, um maior grau de compartilhamento no uso do processador (12 u.t. para o Processo A e 13 u.t. para o Processo B).

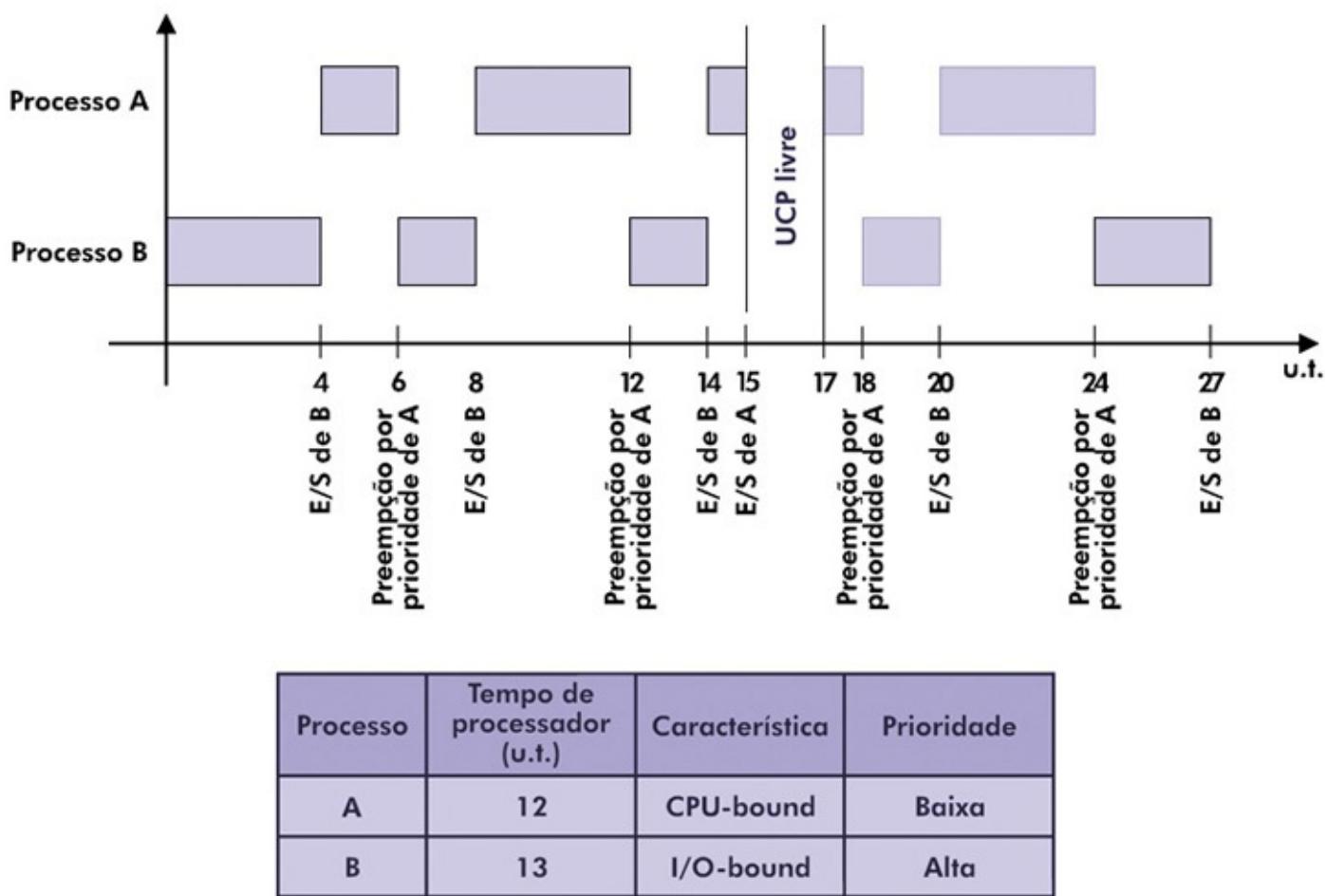


Fig. 8.15 Escalonamento circular com prioridades (exemplo).

Embora os sistemas com prioridade dinâmica sejam mais complexos de implementar que os sistemas com prioridade estática, o tempo de resposta oferecido compensa. Atualmente, a maioria dos sistemas operacionais de tempo compartilhado utiliza o escalonamento circular com prioridades dinâmicas.

► 8.14 Política de Escalonamento em Sistemas de Tempo Real

Diferentemente dos sistemas de tempo compartilhado, nos quais a aplicação não é prejudicada pela variação no tempo de resposta, algumas aplicações específicas exigem respostas imediatas para a execução de determinadas tarefas. Neste caso, a aplicação deve ser executada em sistemas operacionais de tempo real, onde é garantida a execução de processos dentro de limites rígidos de tempo, sem o risco de a aplicação ficar comprometida. Aplicações de controle de processos, como sistemas de controle de produção de bens industriais e controle de tráfego aéreo, são exemplos de aplicação de tempo real.

O escalonamento em sistemas de tempo real deve levar em consideração a importância relativa de cada tarefa na aplicação. Em função disso, o escalonamento por prioridades é o mais adequado, já que para cada processo uma prioridade é associada em função da importância do processo dentro da aplicação. No escalonamento para sistemas de tempo real não deve existir o conceito de fatia de tempo, e a prioridade de cada processo deve ser estática.

O sistema OpenVMS possui uma faixa de valores de prioridade de 0 a 31, sendo que os valores entre 16 e 31 devem ser utilizados exclusivamente para processos de aplicações em tempo real. Nesta subfaixa, o

escalonamento é realizado com base apenas no valor da prioridade do processo, não havendo preempção por tempo.

► 8.15 Exercícios

1. O que é política de escalonamento de um sistema operacional?
2. Quais as funções do escalonador e do dispatcher?
3. Quais os principais critérios utilizados em uma política de escalonamento?
4. Diferencie os tempos de processador, espera, turnaround e resposta.
5. Diferencie os escalonamentos preemptivos e não preemptivos.
6. Qual a diferença entre os escalonamentos FIFO e circular?
7. Descreva o escalonamento SJF e o escalonamento por prioridades.
8. Qual a diferença entre preempção por tempo e preempção por prioridade?
9. O que é um mecanismo de escalonamento adaptativo?
10. Que tipo de escalonamento aplicações de tempo real exigem?
11. O escalonamento por múltiplas filas com realimentação favorece processos CPU-bound ou I/O-bound? Justifique.
12. Considere que cinco processos sejam criados no instante de tempo 0 (P1, P2, P3, P4 e P5) e possuam as características descritas na tabela a seguir:

Processo	Tempo de UCP	Prioridade
P1	10	3
P2	14	4
P3	5	1
P4	7	2
P5	20	5

Desenhe um diagrama ilustrando o escalonamento dos processos e seus respectivos tempos de turnaround, segundo as políticas especificadas a seguir. O tempo de troca de contexto deve ser desconsiderado.

- a) FIFO
- b) SJF
- c) Prioridade (número menor implica prioridade maior)
- d) Circular com fatia de tempo igual a 2 u.t.
13. Considere um sistema operacional com escalonamento por prioridades onde a avaliação do escalonamento é realizada em um intervalo mínimo de 5 ms. Neste sistema, os processos A e B competem por uma única UCP. Desprezando os tempos de processamento relativo às funções do

sistema operacional, a tabela a seguir fornece os estados dos processos A e B ao longo do tempo, medido em intervalos de 5 ms (E = execução, P = pronto e W = espera). O processo A tem menor prioridade que o processo B.

	00-04	05-09	10-14	15-19	20-24	25-29	30-34	35-39	40-44	45-49
Processo A	P	P	E	E	E	P	P	P	E	W
Processo B	E	E	W	W	P	E	E	E	W	W

	50-54	55-59	60-64	65-69	70-74	75-79	80-84	85-89	90-94	95-99	100-105
Processo A	P	E	P	P	E	E	W	W	P	E	E
Processo B	W	P	E	E	W	W	P	E	E	-	-

- a) Em que tempos A sofre preempção?
- b) Em que tempos B sofre preempção?
- c) Refaça a tabela anterior supondo que o processo A é mais prioritário que o processo B.
14. Como o valor do quantum pode afetar o grau de multiprogramação em um sistema operacional? Qual a principal desvantagem de um quantum com um valor muito pequeno?
15. Considere um sistema operacional que implemente escalonamento circular com fatia de tempo igual a 10 u.t. Em um determinado instante de tempo, existem apenas três processos (P1, P2 e P3) na fila de pronto, e o tempo de UCP de cada processo é 18, 4 e 13 u.t., respectivamente. Qual o estado de cada processo no instante de tempo T, considerando a execução dos processos P1, P2 e P3, nesta ordem, e que nenhuma operação de E/S é realizada?
- a) T = 8 u.t.
b) T = 11 u.t.
c) T = 33 u.t.
16. Considere um sistema operacional que implemente escalonamento circular com fatia de tempo igual a 10 u.t. Em um determinado instante de tempo existem apenas três processos (P1, P2 e P3) na fila de pronto, e o tempo de UCP de cada processo é 14, 4 e 12 u.t., respectivamente. Qual o estado de cada processo no instante de tempo T, considerando a execução dos processos P1, P2 e P3, nesta ordem, e que apenas o processo P1 realiza operações de E/S? Cada operação de E/S é executada após 5 u.t. e consome 10 u.t.
- a) T = 8 u.t.
b) T = 18 u.t.
c) T = 28 u.t.
17. Existem quatro processos (P1, P2, P3 e P4) na fila de pronto, com tempos de UCP estimados em 9, 6, 3 e 5, respectivamente. Em que ordem os processos devem ser executados para minimizar o tempo de turnaround dos processos?
18. Considere a tabela a seguir, na qual:

Processo	Tempo de UCP	Prioridade
P1	40	4
P2	20	3
P3	50	1
P4	30	3

Calcule o tempo de turnaround médio dos processos considerando que todos foram criados no instante de tempo 0 u.t. e que o tempo de troca de contexto é igual a 0 para os seguintes escalonamentos:

- a) FIFO
- b) SJF
- c) Circular com fatia de tempo igual a 20 u.t.
- d) Prioridades

19. Refaça o Exercício 18 considerando que o tempo de troca de contexto é igual a 5 u.t.

► 8.16 Laboratório com o Simulador SOsim

Atividade 1: Escalonamento Circular

a) Práticas de simulação

- Execute o simulador SOsim e configure-o para trabalhar com Escalonamento Circular: janela *Console SOsim/Opções/Parâmetros do Sistema* na guia *Processador*.

b) Análise prática

- Crie dois processos com a mesma prioridade (um CPU-bound e outro I/O-bound): janela *Gerência de Processos / Criar* – janela *Criação de Processos / Criar*.
- Na janela *Gerência de Processos*, observe o tempo de processador de cada processo durante dois minutos e as mudanças de estado. Após esse período, anote o tempo de processador de cada processo. Analise o balanceamento no uso do processador pelos dois processos.
- Na janela *Gerência de Processos*, finalize os dois processos.
- Na janela *Gerência de Processador*, aumente a fatia de tempo movimentando a barra de *Fatia de Tempo*.
- Na janela *Gerência de Processos*, observe mais uma vez o tempo de processador de cada processo durante dois minutos e as mudanças de estado. Após esse período, anote o tempo de processador de cada processo. Compare os tempos anotados nas duas e analise o resultado do balanceamento no uso do processador pelos dois processos. Identifique as causas da variação.

c) Questão teórica para responder com a ajuda do simulador

Considere a concorrência, nesse tipo de escalonamento, com dois processos CPU-bound que não realizam operações de E/S. Qual o efeito da variação da fatia de tempo sobre o balanceamento no uso do processador?

Atividade 2: Escalonamento Circular com Prioridades Estáticas I

a) Práticas de simulação

- Execute o simulador SOsim e configure-o para trabalhar com Escalonamento Circular com Prioridades Estáticas: janela *Console SOsim/Opções/Parâmetros do Sistema* na guia *Processador*.

b) Análise prática

- Crie um processo CPU-bound com prioridade 3 e um outro I/O-bound com prioridade 4: janela *Gerência de Processos/Criar* – janela *Criação de Processos/Criar*.
- Na janela *Gerência de Processos*, observe o tempo de processador de cada processo durante dois minutos e as mudanças de estado. Após esse período anote o tempo de processador de cada processo.
- Verifique a preempção por prioridade que ocorre toda vez que o processo I/O-bound de maior prioridade passa para o estado de pronto.
- Analise o balanceamento no uso do processador pelos dois processos comparativamente à Atividade 1.

c) Questões teóricas para responder com a ajuda do simulador

Quais devem ser os critérios para determinar as prioridades dos processos?

Caso, nesse escalonamento, todos os processos sejam criados com a mesma prioridade, qual o benefício dessa política sobre o Escalonamento Circular?

Atividade 3: Escalonamento Circular com Prioridades Estáticas II

a) Práticas de simulação

- Execute o simulador SOsim e configure-o para trabalhar com Escalonamento Circular com Prioridades Estáticas: janela *Console SOsim/Opções/Parâmetros do Sistema* na guia *Processador*.

b) Análise prática

- Crie um processo CPU-bound com prioridade 4 e um outro I/O-bound com prioridade 3: janela *Gerência de Processos/Criar* – janela *Criação de Processos/Criar*.
- Na janela *Gerência de Processos*, observe o escalonamento dos dois processos. Analise o

problema do starvation.

c) Questões teóricas para responder com a ajuda do simulador

Por que o problema do starvation pode ocorrer?

Cite duas ações que o administrador do sistema pode realizar quando é identificada a situação de starvation em um processo.

Atividade 4: Escalonamento Circular com Prioridades Dinâmica

a) Práticas de simulação

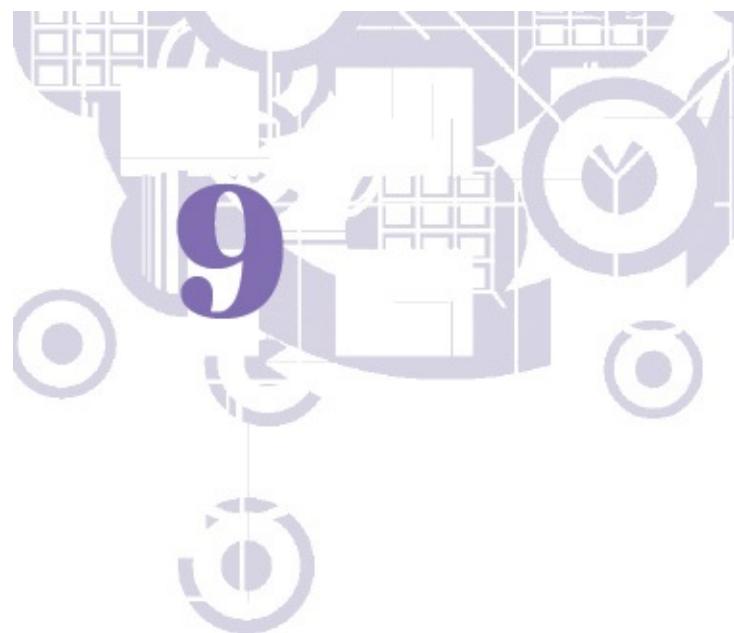
- Execute o simulador SOsim e configure-o para trabalhar com Escalonamento Circular com Prioridades Dinâmicas: janela *Console SOsim/Opções/Parâmetros do Sistema* na guia *Processador*.
- Habilite as janelas de log e estatísticas: janela *Console SOsim/Janelas*.
- Na janela *Gerência do Processador* desloque a barra *Frequência clock* para a metade da escala.

b) Análise prática

- Crie um processo CPU-bound com prioridade base 3 e mais três processos I/O-bound com prioridade base 4, porém com perfis diferentes (tipo 1, 2 e 3): janela *Gerência de Processos/Criar* — janela *Criação de Processos/Criar*.
- Observe as prioridades base e dinâmica dos quatro processos na janela *Gerência de Processos*.
- Identifique os motivos de as prioridades dinâmicas dos processos variarem ao longo do tempo.
- Observe na janela de log o valor do incremento recebido na prioridade de cada processo, Identifique o porquê das diferenças nos valores do incremento.
- Observe na janela de estatísticas o percentual de utilização da UCP.
- Suspenda o processo CPU-bound: janela *Gerência de Processos/Suspender*.
- Observe na janela de estatísticas as mudanças no percentual de utilização da UCP e identifique a razão.
- Libere o processo CPU-bound do estado de suspenso: janela *Gerência de Processos/Proseguir*.

c) Questão teórica para responder com a ajuda do simulador

Qual o critério utilizado pelo sistema operacional para determinar diferentes valores de incremento à prioridade-base de um processo quando há uma mudança do estado de espera para pronto?



9

Gerência de Memória

► 9.1 Introdução

Historicamente, a memória principal sempre foi vista como um recurso escasso e caro. Uma das maiores preocupações dos projetistas foi desenvolver sistemas operacionais que não ocupassem muito espaço de memória e, ao mesmo tempo, otimizassem a utilização dos recursos computacionais. Mesmo atualmente, com a redução de custo e consequente aumento da capacidade da memória principal, seu gerenciamento é um dos fatores mais importantes no projeto de sistemas operacionais.

Enquanto nos sistemas monoprogramáveis a gerência da memória não é muito complexa, nos sistemas multiprogramáveis ela se torna crítica, devido à necessidade de se maximizar o número de usuários e aplicações utilizando eficientemente o espaço da memória principal.

Neste capítulo apresentaremos os esquemas básicos de gerência da memória principal, mostrando suas vantagens, desvantagens e implementações, numa sequência quase evolutiva. Esses conhecimentos serão úteis para a compreensão das motivações que levaram à ampla adoção do mecanismo de gerência de memória virtual nos sistemas operacionais modernos, apresentado no próximo capítulo.

► 9.2 Funções Básicas

Em geral, programas são armazenados em memórias secundárias, como disco ou fita, por ser um meio não volátil, abundante e de baixo custo. Como o processador somente executa instruções localizadas na memória principal, o sistema operacional deve sempre transferir programas da memória secundária para a memória principal antes de serem executados. Como o tempo de acesso à memória secundária é muito superior ao tempo de acesso à memória principal, o sistema operacional deve buscar reduzir o número de operações de E/S à memória secundária, caso contrário podem ser ocasionados sérios problemas no desempenho do sistema.

A gerência de memória deve tentar manter na memória principal o maior número de processos residentes,

permitindo maximizar o compartilhamento do processador e demais recursos computacionais. Mesmo na ausência de espaço livre, o sistema deve permitir que novos processos sejam aceitos e executados. Isso é possível através da transferência temporária de processos residentes na memória principal para a memória secundária, liberando espaço para novos processos. Esse mecanismo é conhecido como swapping, e será detalhado posteriormente.

Outra preocupação na gerência de memória é permitir a execução de programas que sejam maiores que a memória física disponível, implementada através de técnicas como o overlay e a memória virtual.

Em um ambiente de multiprogramação, o sistema operacional deve proteger as áreas de memória ocupadas por cada processo, além da área onde reside o próprio sistema. Caso um programa tente realizar algum acesso indevido à memória, o sistema de alguma forma deve impedi-lo. Apesar de a gerência de memória garantir a proteção de áreas da memória, mecanismos de compartilhamento devem ser oferecidos para que diferentes processos possam trocar dados de forma protegida.

► 9.3 Alocação Contígua Simples

A *alocação contígua simples* foi implementada nos primeiros sistemas operacionais, porém ainda está presente em alguns sistemas monoprogramáveis. Nesse tipo de organização, a memória principal é subdividida em duas áreas: uma para o sistema operacional e outra para o programa do usuário (Fig. 9.1). Dessa forma, o programador deve desenvolver suas aplicações preocupado, apenas, em não ultrapassar o espaço de memória disponível, ou seja, a diferença entre o tamanho total da memória principal e a área ocupada pelo sistema operacional.



Fig. 9.1 Alocação contígua simples.

Nesse esquema, o usuário tem controle sobre toda a memória principal, podendo ter acesso a qualquer posição de memória, inclusive a área do sistema operacional. Para proteger o sistema desse tipo de acesso, que pode ser intencional ou não, alguns sistemas implementam proteção através de um registrador que delimita as áreas do sistema operacional e do usuário (Fig. 9.2). Dessa forma, sempre que um programa faz referência a um endereço na memória, o sistema verifica se o endereço está dentro dos limites permitidos. Caso não esteja, o programa é cancelado e uma mensagem de erro é gerada, indicando

que houve uma violação no acesso à memória principal.

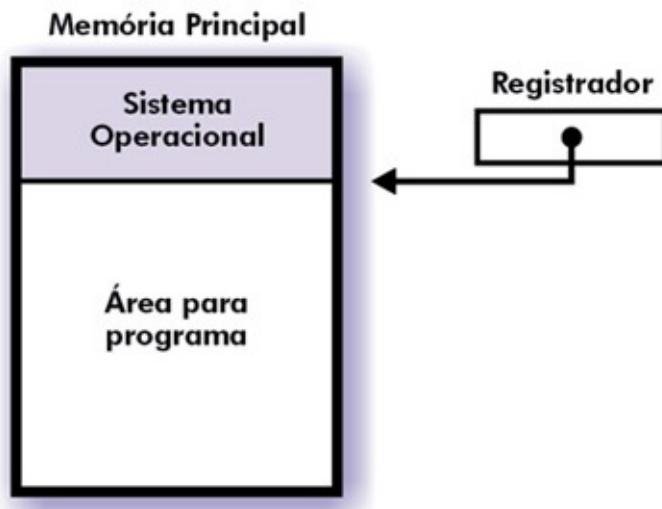


Fig. 9.2 Proteção na alocação contígua simples.

Apesar da fácil implementação e do código reduzido, a alocação contígua simples não permite a utilização eficiente dos recursos computacionais, pois apenas um usuário pode dispor desses recursos. Em relação à memória principal, caso o programa do usuário não a preencha totalmente existirá um espaço de memória livre sem utilização (Fig. 9.3).

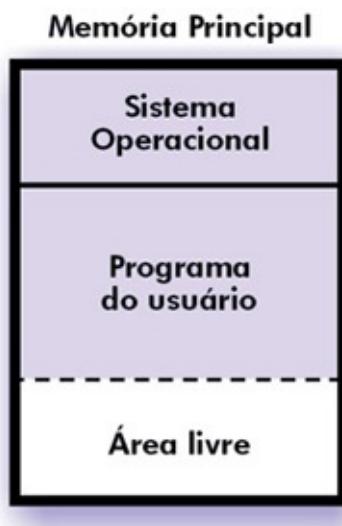


Fig. 9.3 Subutilização da memória principal.

► 9.4 Técnica de Overlay

Na alocação contígua simples todos os programas estão limitados ao tamanho da área de memória principal disponível para o usuário. Uma solução encontrada para o problema é dividir o programa em módulos, de forma que seja possível a execução independente de cada módulo, utilizando uma mesma área de memória. Essa técnica é chamada de *overlay*.

Considere um programa que tenha três módulos: um principal, um de cadastramento e outro de impressão, sendo os módulos de cadastramento e impressão independentes. A independência do código significa que quando um módulo estiver na memória para execução o outro não precisa necessariamente estar presente. O módulo principal é comum aos dois módulos; logo, deve permanecer na memória durante todo o tempo da execução do programa.

Como podemos verificar na Fig. 9.4, a memória é insuficiente para armazenar todo o programa. A técnica de overlay utiliza uma área de memória comum, onde os módulos de cadastramento e impressão poderão compartilhar a mesma área de memória (área de overlay). Sempre que um dos dois módulos for referenciado pelo módulo principal, o módulo será carregado da memória secundária para a área de overlay. No caso de uma referência a um módulo que já esteja na área de overlay, a carga não é realizada; caso contrário, o novo módulo irá sobrepor-se ao que já estiver na memória principal.

A definição das áreas de overlay é função do programador, através de comandos específicos da linguagem de programação utilizada. O tamanho de uma área de overlay é estabelecido a partir do tamanho do maior módulo. Por exemplo, se o módulo de cadastramento tem 4 Kb e o módulo de impressão 2 Kb, a área de overlay deverá ter o tamanho do maior módulo, ou seja, 4 Kb.

A técnica de overlay tem a vantagem de permitir ao programador expandir os limites da memória principal. A utilização dessa técnica exige muito cuidado, pois pode trazer implicações tanto na sua manutenção quanto no desempenho das aplicações, devido à possibilidade de transferência excessiva dos módulos entre as memórias principal e secundária.

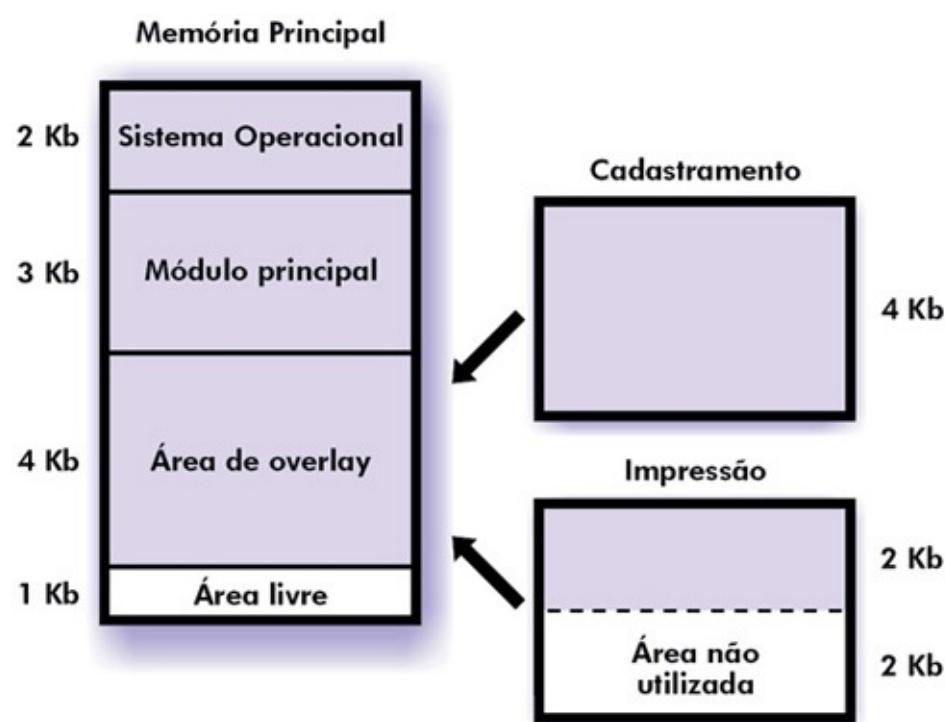


Fig. 9.4 Técnica de overlay.

► 9.5 Alocação Particionada

Os sistemas operacionais evoluíram no sentido de proporcionar melhor aproveitamento dos recursos disponíveis. Nos sistemas monoprogramáveis, o processador permanece grande parte do tempo ocioso e a memória principal é subutilizada. Os sistemas multiprogramáveis já são muito mais eficientes no uso do processador, necessitando, assim, que diversos programas estejam na memória principal ao mesmo tempo e que novas formas de gerência da memória sejam implementadas.

9.5.1 Alocação Particionada Estática

Nos primeiros sistemas multiprogramáveis, a memória era dividida em pedaços de tamanho fixo, chamados *partições*. O tamanho das partições, estabelecido na fase de inicialização do sistema, era definido em função do tamanho dos programas que executariam no ambiente (Fig. 9.5). Sempre que fosse necessária a alteração do tamanho de uma partição, o sistema deveria ser desativado e reinicializado com uma nova configuração. Esse tipo de gerência de memória é conhecido como *alocação particionada estática* ou *fixa*.

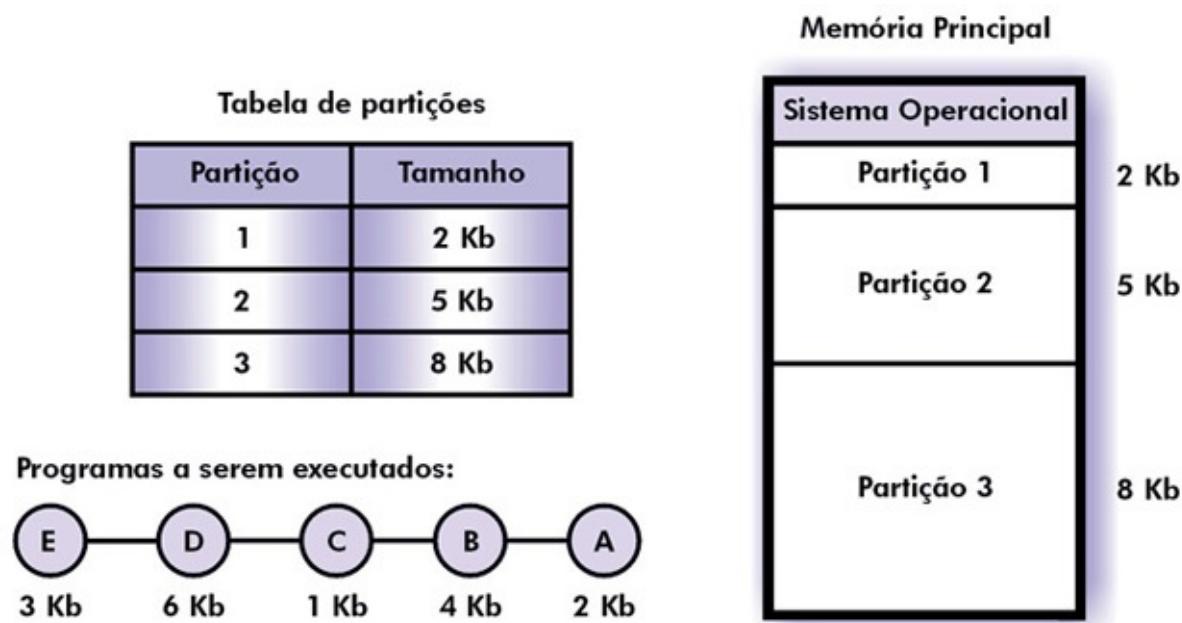


Fig. 9.5 Alocação particionada estática.

Inicialmente, os programas só podiam ser carregados e executados em apenas uma partição específica, mesmo se outras estivessem disponíveis. Essa limitação se devia aos compiladores e montadores que geravam apenas código absoluto. No *código absoluto* todas as referências a endereços no programa são posições físicas na memória principal, ou seja, o programa só poderia ser carregado a partir do endereço de memória especificado no seu próprio código. Se, por exemplo, os programas A e B estivessem sendo executados e a terceira partição estivesse livre, os programas C e E não poderiam ser processados (Fig. 9.6). A esse tipo de gerência de memória chamou-se *alocação particionada estática absoluta*.

Memória Principal

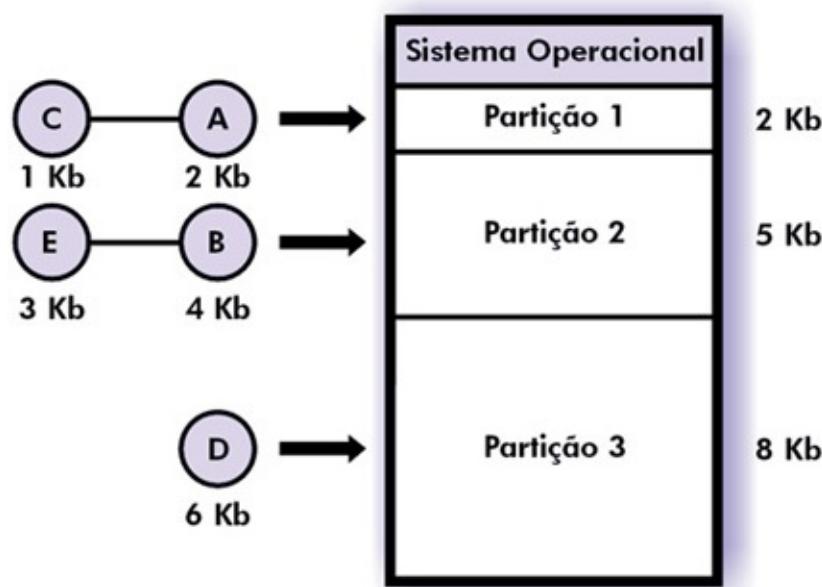


Fig. 9.6 Alocação particionada estática absoluta.

Com a evolução dos compiladores, montadores, linkers e loaders, o código gerado deixou de ser absoluto e passou a ser relocável. No *código relocável*, todas as referências a endereços no programa são relativas ao início do código, e não a endereços físicos de memória. Desta forma, os programas puderam ser executados a partir de qualquer partição. Quando o programa é carregado, o loader calcula todos os endereços a partir da posição inicial onde o programa foi alocado. Caso os programas A e B terminassem, o programa E poderia ser executado em qualquer uma das duas partições (Fig. 9.7). Esse tipo de gerência de memória é denominado *alocação particionada estática relocável*.

Memória Principal

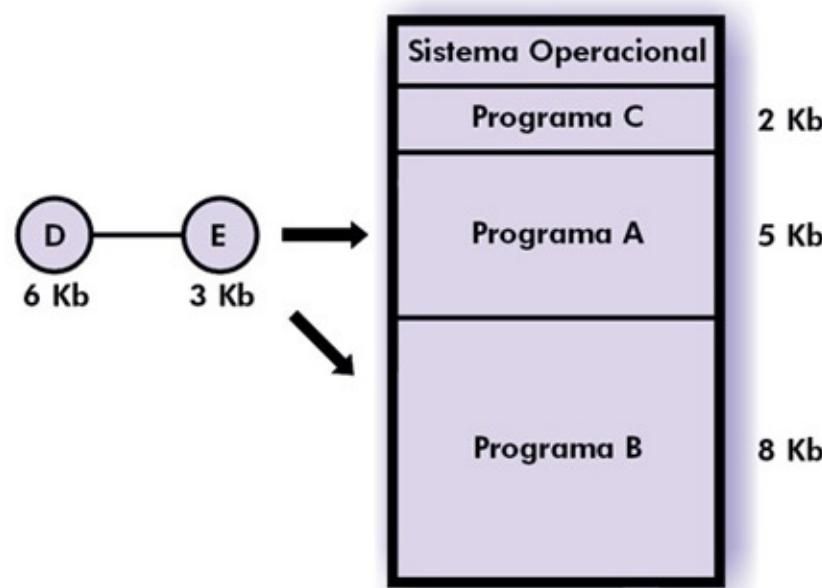


Fig. 9.7 Alocação particionada estática relocável.

Para manter o controle sobre quais partições estão alocadas, a gerência de memória mantém uma tabela com o endereço inicial de cada partição, seu tamanho e se está em uso (Fig. 9.8). Sempre que um

programa é carregado para a memória, o sistema percorre a tabela, na tentativa de localizar uma partição livre onde o programa possa ser carregado.

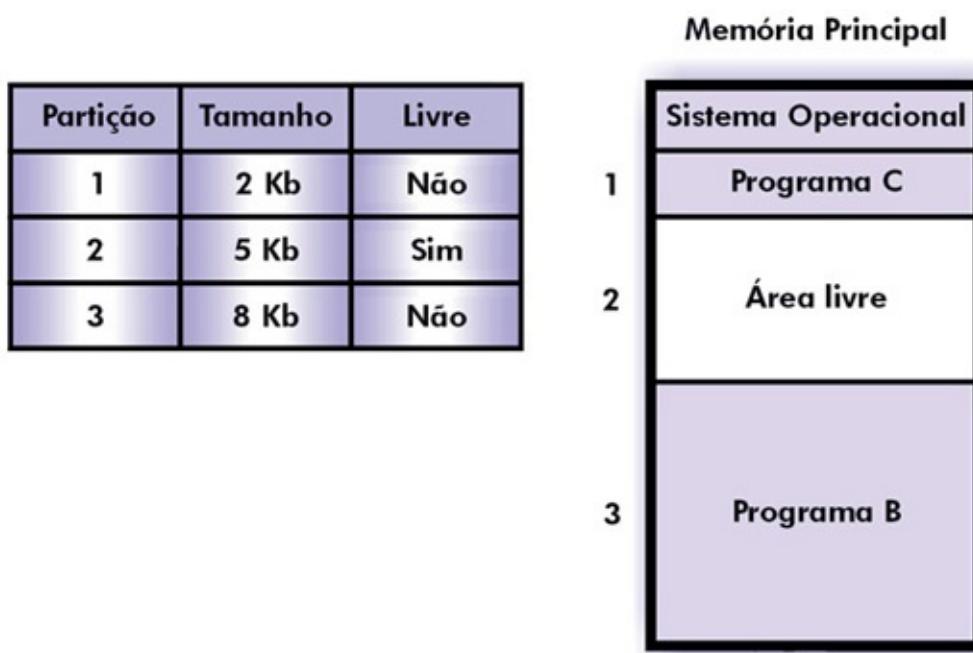


Fig. 9.8 Tabela de alocação de partições.

Nesse esquema de alocação de memória a proteção baseia-se em dois registradores, que indicam os limites inferior e superior da partição onde o programa está sendo executado (Fig. 9.9). Caso o programa tente acessar uma posição de memória fora dos limites definidos pelos registradores, ele é interrompido e uma mensagem de violação de acesso é gerada pelo sistema operacional.

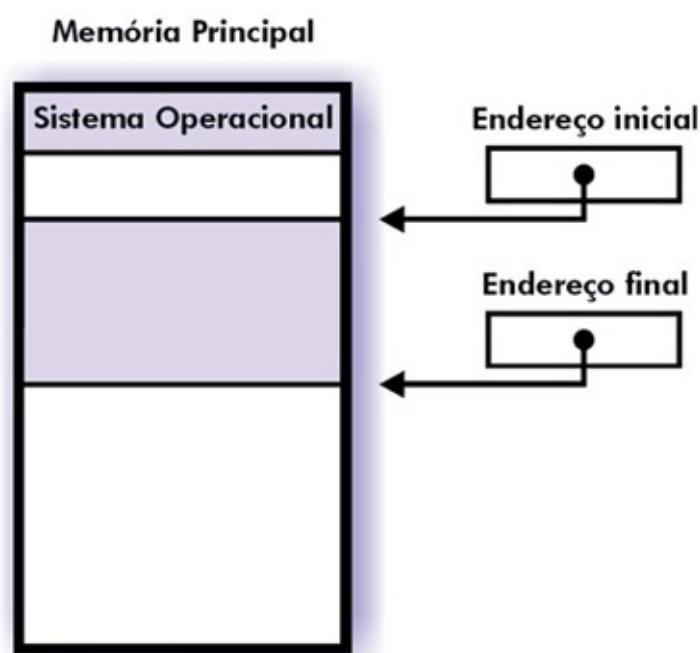


Fig. 9.9 Proteção na alocação particionada.

Tanto nos sistemas de alocação absoluta quanto nos de alocação relocável os programas, normalmente,

não preenchem totalmente as partições onde são carregados. Por exemplo, os programas C, A e E não ocupam integralmente o espaço das partições onde estão alocados, deixando 1 Kb, 3 Kb e 5 Kb de áreas livres, respectivamente (Fig. 9.10). Esse tipo de problema, decorrente da alocação fixa das partições, é conhecido como *fragmentação interna*.

Um exemplo de sistema operacional que implementou esse tipo de gerência de memória é o OS/MFT (Multiprogramming with a Fixed Number of Tasks) da IBM.

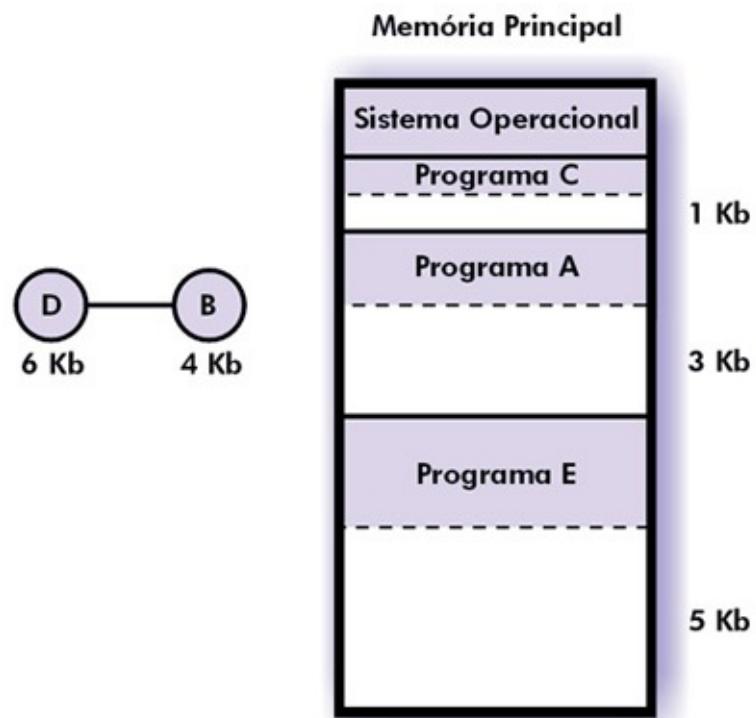


Fig. 9.10 Fragmentação interna.

9.5.2 Alocação Particionada Dinâmica

A alocação particionada estática, analisada anteriormente, deixou evidente a necessidade de uma nova forma de gerência da memória principal, em que o problema da fragmentação interna fosse reduzido e, consequentemente, o grau de compartilhamento da memória aumentado.

Na *alocação particionada dinâmica*, ou *variável*, foi eliminado o conceito de partições de tamanho fixo. Nesse esquema, cada programa utilizaria o espaço necessário, tornando essa área sua partição (Fig. 9.11). Como os programas utilizam apenas o espaço de que necessitam, no esquema de alocação particionada dinâmica o problema da fragmentação interna não ocorre.

A princípio, o problema da fragmentação interna está resolvido, porém nesse caso existe um problema que não é tão óbvio quanto no esquema anterior. Um diferente tipo de fragmentação começará a ocorrer, quando os programas forem terminando e deixando espaços cada vez menores na memória, não permitindo o ingresso de novos programas. No caso da Fig. 9.12, mesmo existindo 12 Kb livres de memória principal o programa D, que necessita de 6 Kb de espaço, não poderá ser carregado para execução, pois esse espaço não está disposto contiguamente. Esse tipo de problema é chamado de

fragmentação externa.

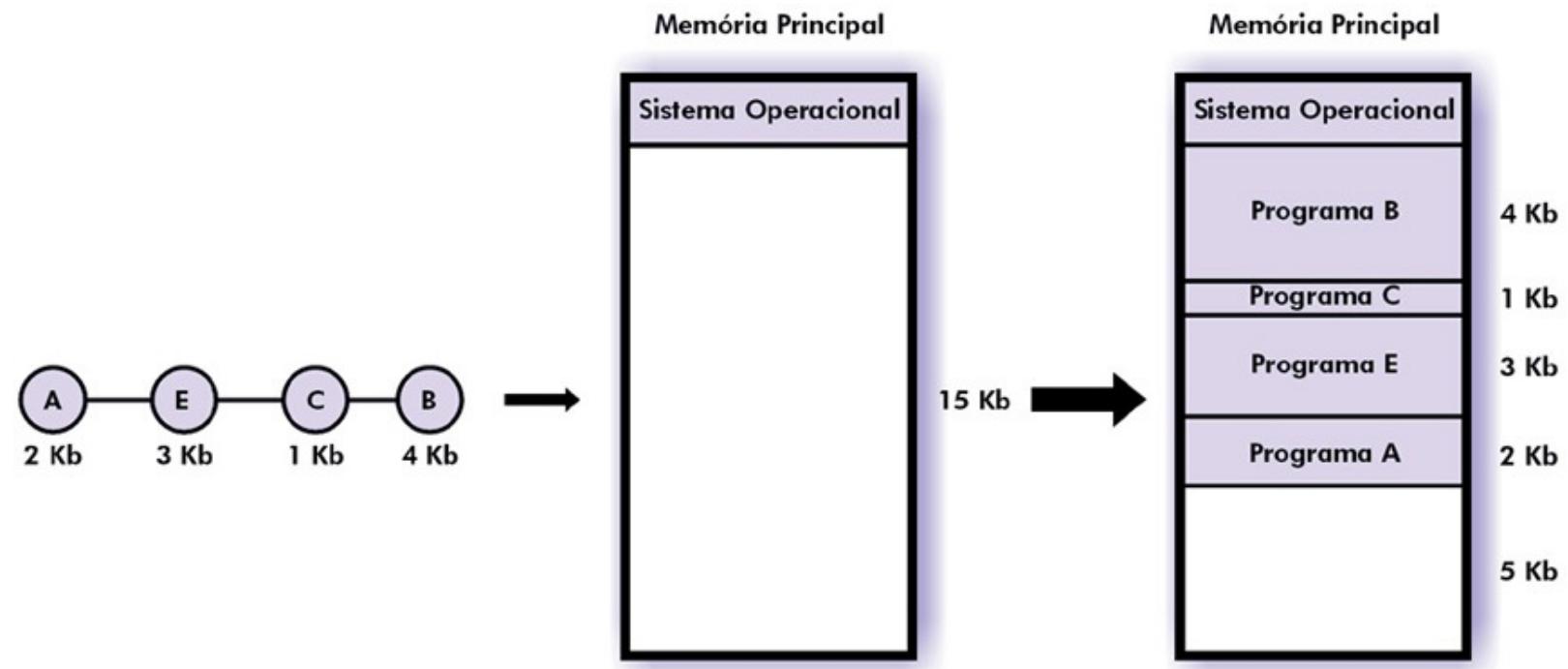


Fig. 9.11 Alocação particionada dinâmica.

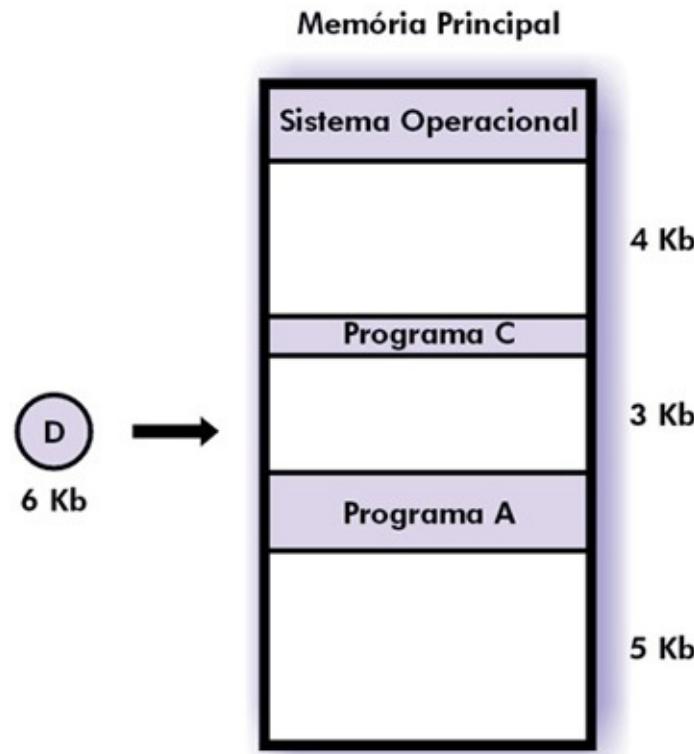


Fig. 9.12 Fragmentação externa.

Existem duas soluções para o problema da fragmentação externa da memória principal. Na primeira solução, conforme os programas terminam apenas os espaços livres adjacentes são reunidos, produzindo áreas livres de tamanho maior. Na Fig. 9.13, caso o programa C termine, uma área de 8 Kb é criada.

Memória Principal

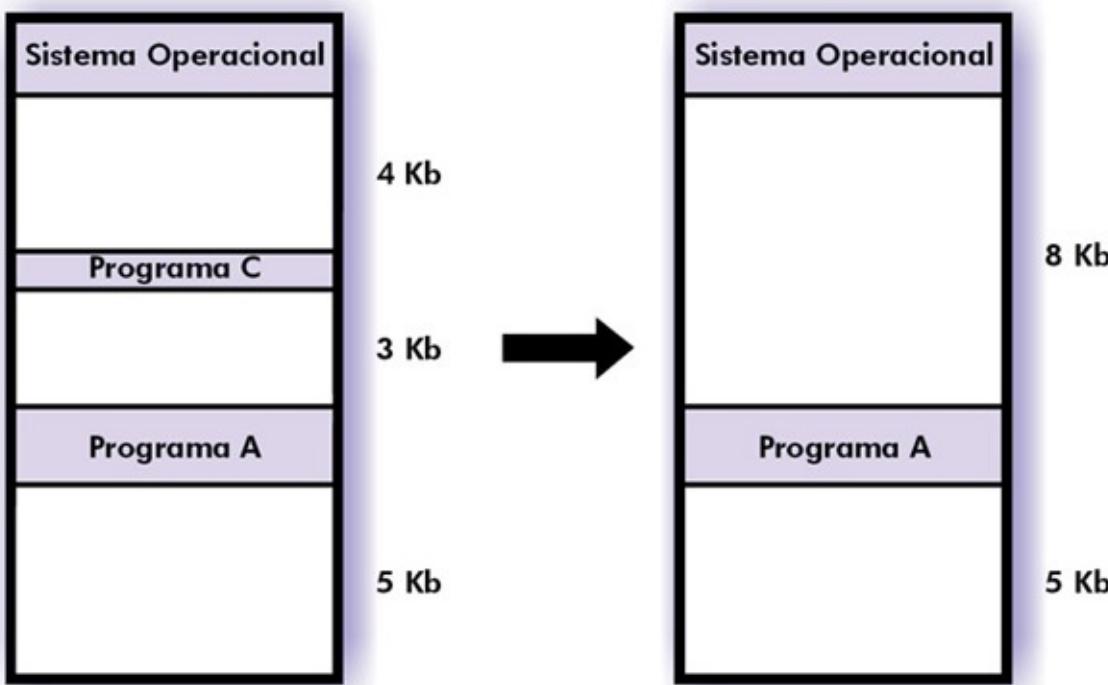


Fig. 9.13 Solução para a fragmentação externa (a).

A segunda solução envolve a relocação de todas as partições ocupadas, eliminando todos os espaços entre elas e criando uma única área livre contígua (Fig. 9.14). Para que esse processo seja possível é necessário que o sistema tenha a capacidade de mover os diversos programas na memória principal, ou seja, realizar *relocação dinâmica*. Esse mecanismo de compactação, também conhecido como *alocação particionada dinâmica com relocação*, reduz em muito o problema da fragmentação, porém a complexidade do seu algoritmo e o consumo de recursos do sistema, como processador e área em disco, podem torná-lo inviável.

Memória Principal

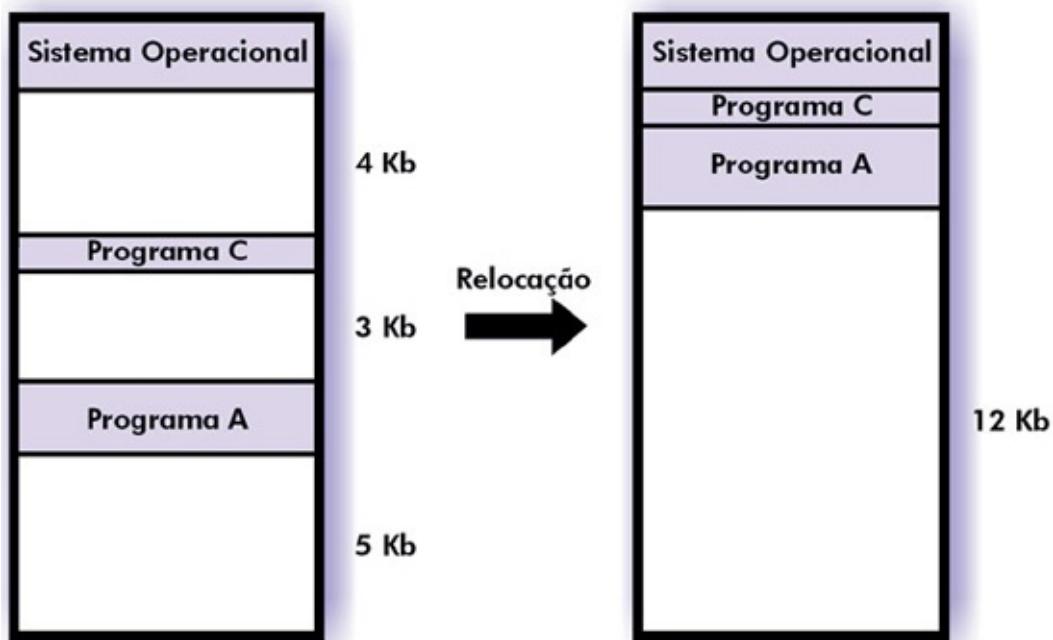


Fig. 9.14 Solução para a fragmentação externa (b).

Um exemplo de sistema operacional que implementou esse tipo de gerência de memória é o OS/MVT (Multiprogramming with a Variable Number of Tasks) da IBM.

9.5.3 Estratégias de Alocação de Partição

Os sistemas operacionais implementam, basicamente, três estratégias para determinar em qual área livre um programa será carregado para execução. Essas estratégias tentam evitar ou diminuir o problema da fragmentação externa.

A melhor estratégia a ser adotada por um sistema depende de uma série de fatores, sendo o mais importante o tamanho dos programas processados no ambiente. Independentemente do algoritmo utilizado, o sistema possui uma lista de áreas livres, com o endereço e tamanho de cada área (Fig. 9.15).

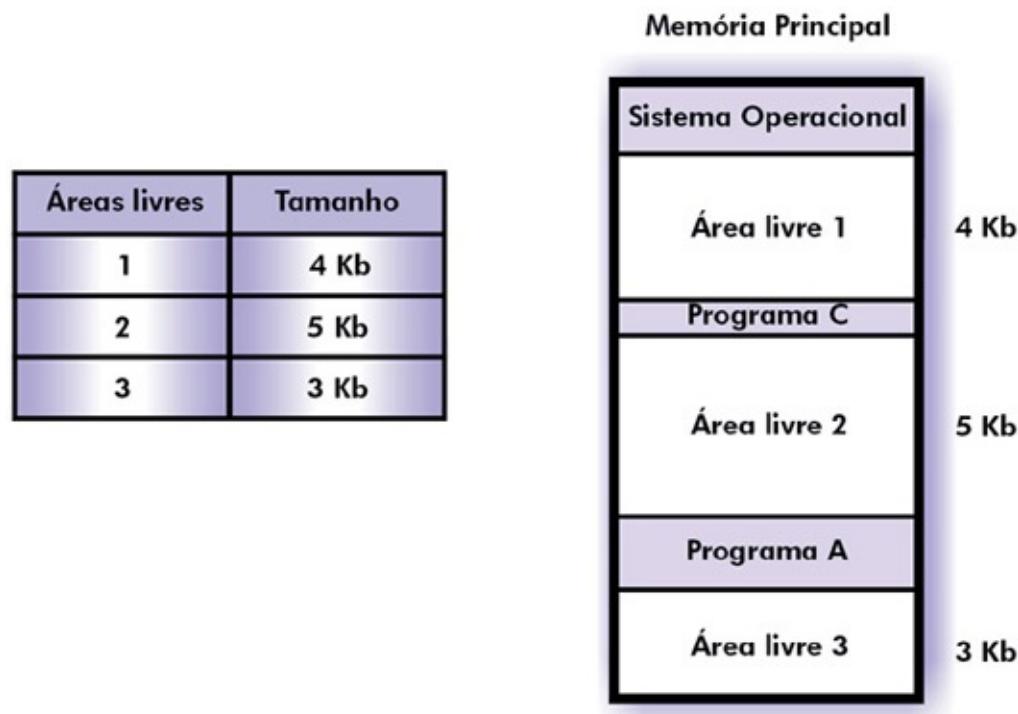


Fig. 9.15 Lista de áreas livres.

■ Best-fit

Na estratégia *best-fit*, a melhor partição é escolhida, ou seja, aquela em que o programa deixa o menor espaço sem utilização (Fig. 9.16a). Nesse algoritmo, a lista de áreas livres está ordenada por tamanho, diminuindo o tempo de busca por uma área desocupada. Uma grande desvantagem desse método é consequência do próprio algoritmo. Como é alocada a partição que deixa a menor área livre, a tendência é que cada vez mais a memória fique com pequenas áreas não contíguas, aumentando o problema da fragmentação.

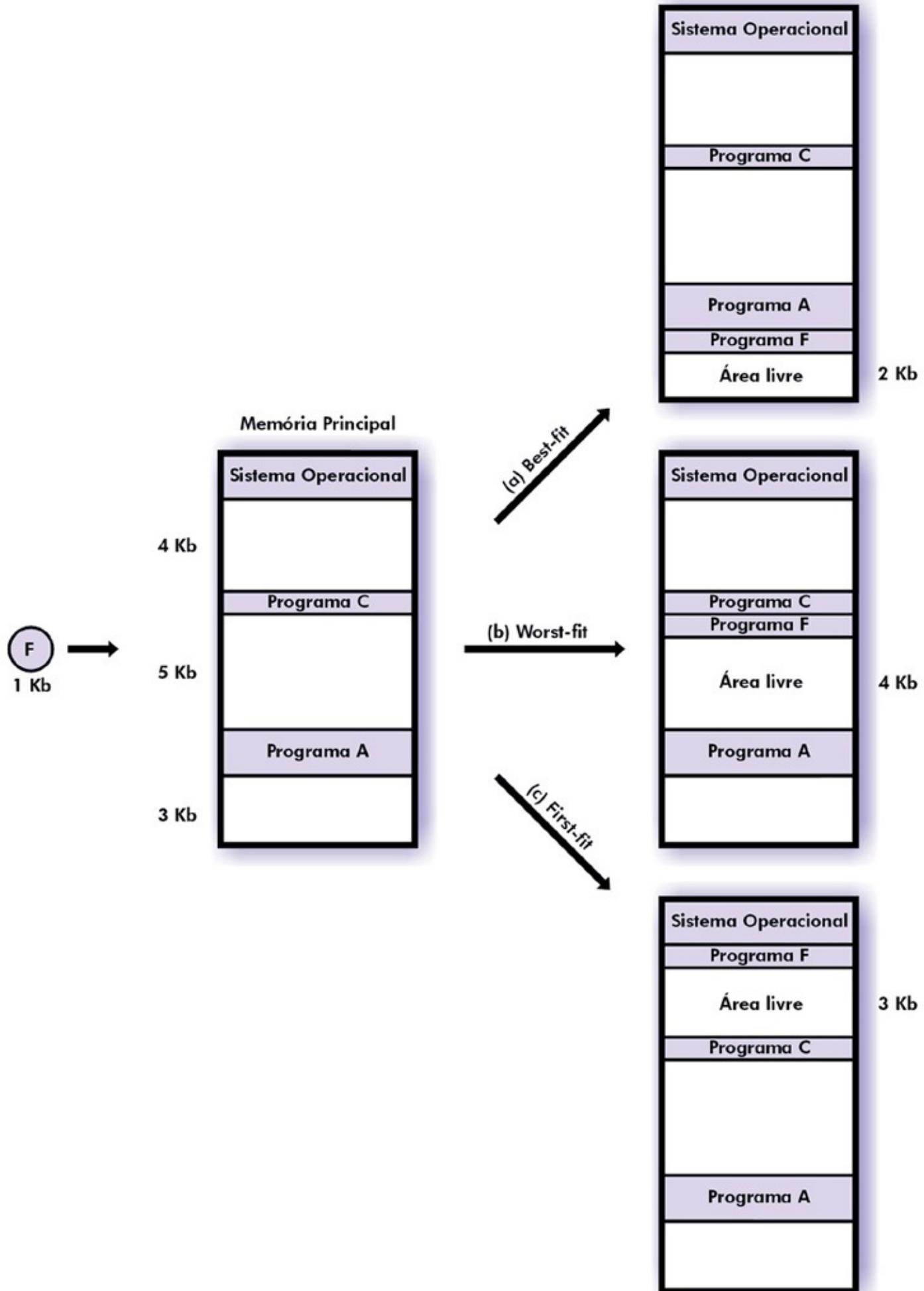


Fig. 9.16 Estratégias para a escolha da partição.

■ Worst-fit

Na estratégia *worst-fit*, a pior partição é escolhida, ou seja, aquela em que o programa deixa o maior espaço sem utilização (Fig. 9.16b). Apesar de utilizar as maiores partições, a técnica de worst-fit deixa espaços livres maiores que permitem a um maior número de programas utilizar a memória, diminuindo o problema da fragmentação.

■ First-fit

Na estratégia *first-fit*, a primeira partição livre de tamanho suficiente para carregar o programa é escolhida (Fig. 9.16c). Nesse algoritmo, a lista de áreas livres está ordenada por endereços crescentemente. Como o método tenta primeiro utilizar as áreas livres de endereços mais baixos, existe uma grande chance de se obter uma grande partição livre nos endereços de memória mais altos. Das três estratégias apresentadas, a first-fit é a mais rápida, consumindo menos recursos do sistema.

► 9.6 Swapping

Mesmo com o aumento da eficiência da multiprogramação e, particularmente, da gerência de memória, muitas vezes um programa não podia ser executado por falta de uma partição livre disponível. A técnica de *swapping* foi introduzida para contornar o problema da insuficiência de memória principal.

Em todos os esquemas apresentados anteriormente, um processo permanecia na memória principal até o final da sua execução, inclusive nos momentos em que esperava por um evento, como uma operação de leitura ou gravação. O swapping é uma técnica aplicada à gerência de memória para programas que esperam por memória livre para serem executados. Nesta situação, o sistema escolhe um processo residente, que é transferido da memória principal para a memória secundária (*swap out*), geralmente disco. Posteriormente, o processo é carregado de volta da memória secundária para a memória principal (*swap in*) e pode continuar sua execução como se nada tivesse ocorrido (Fig. 9.17).

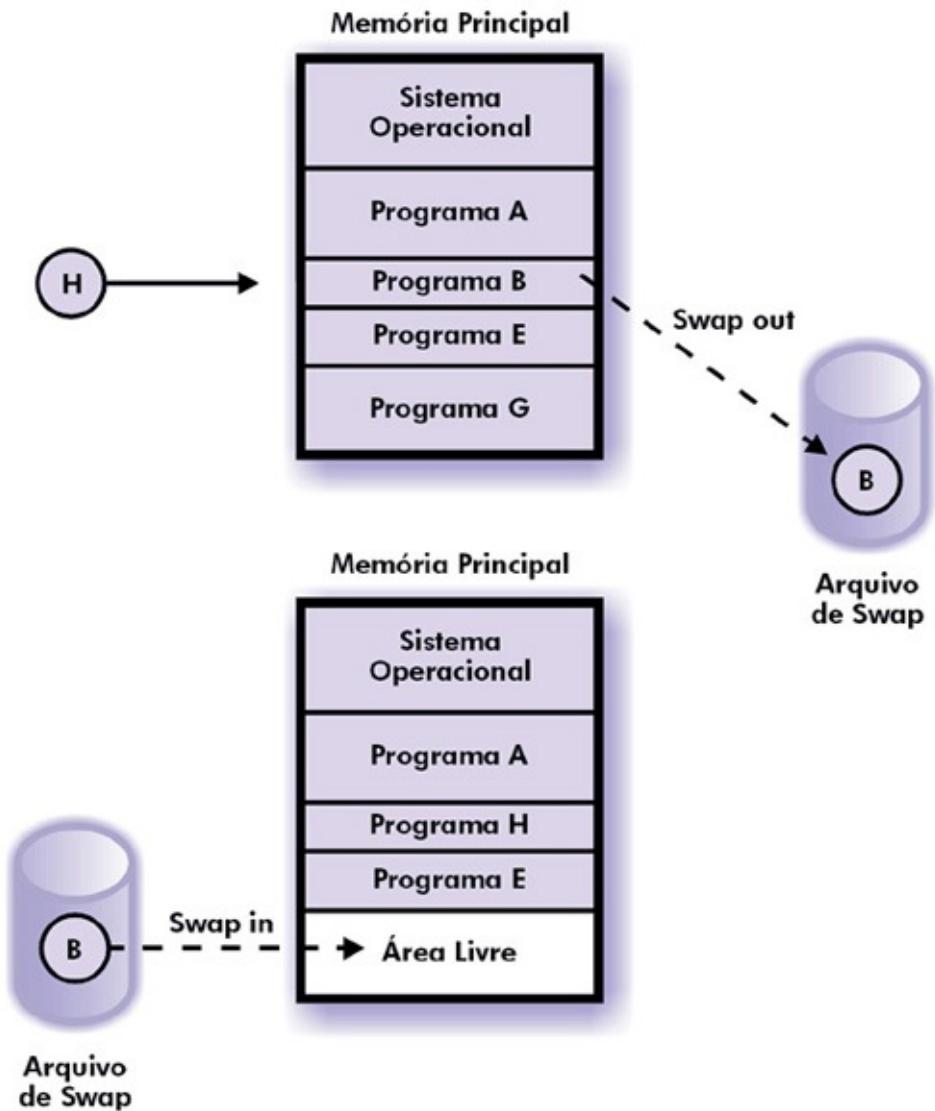


Fig. 9.17 Swapping.

O algoritmo de escolha do processo a ser retirado da memória principal deve priorizar aquele com menores chances de ser escalonado para evitar o swapping desnecessário de um processo que será executado logo em seguida. Os processos retirados da memória estão geralmente no estado de espera, mas ainda existe a possibilidade de um processo no estado de pronto também ser selecionado para swapping. Os processos no estado de espera ou pronto quando fora de memória são ditos não residentes (outswapped).

Para que a técnica de swapping seja implementada é essencial que o sistema ofereça um loader que implemente a *relocação dinâmica* de programas. Um loader relocável que não ofereça esta facilidade permite que um programa seja colocado em qualquer posição de memória, porém a relocação é apenas realizada no momento do carregamento. No caso do swapping, um programa pode sair e voltar diversas vezes para a memória, sendo necessário que a relocação seja realizada pelo loader a cada carregamento.

A relocação dinâmica é realizada através de um registrador especial denominado *registrador de relocação*. No momento em que o programa é carregado na memória, o registrador recebe o endereço inicial da posição de memória que o programa ocupará. Toda vez que ocorrer uma referência a algum endereço, o endereço contido na instrução será somado ao conteúdo do registrador, gerando, assim, o endereço físico (Fig. 9.18). Dessa forma, um programa pode ser carregado em qualquer posição de

memória.

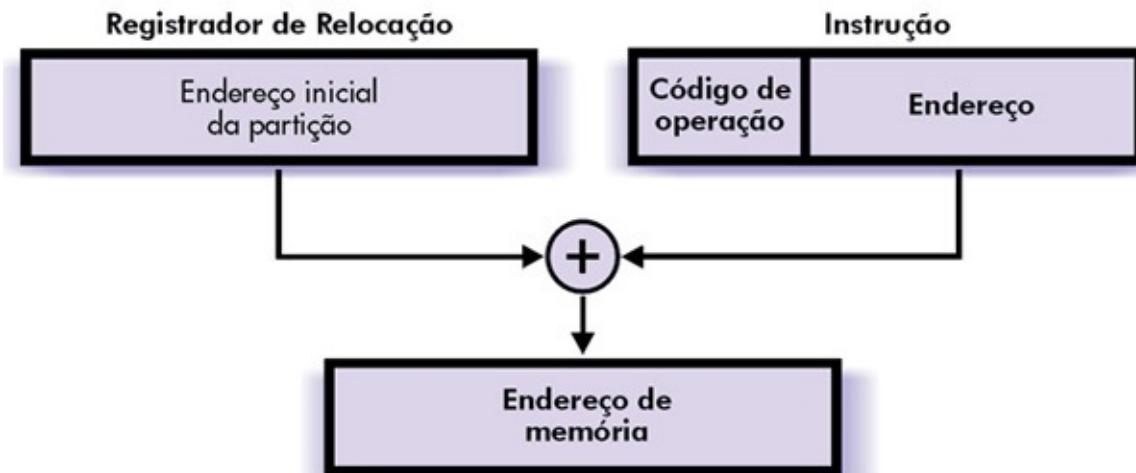


Fig. 9.18 Relocação dinâmica.

O conceito de swapping permite um maior compartilhamento da memória principal e, consequentemente, uma maior utilização dos recursos do sistema computacional. Seu maior problema é o elevado custo das operações de entrada/saída (swap in/out). Em momentos em que há pouca memória disponível, o sistema pode ficar quase que dedicado à realização de swapping, deixando de executar outras tarefas e impedindo a execução dos demais processos residentes. Esta situação é denominada *thrashing*, e é considerada um problema crítico na gerência de memória dos sistemas operacionais.

Os primeiros sistemas operacionais que implementaram esta técnica surgiram na década de 1960, como o CTSS do MIT e OS/360 da IBM. Com a evolução dos sistemas operacionais, novos esquemas de gerência de memória passaram a incorporar a técnica de swapping, como a gerência de memória virtual.

► 9.7 Exercícios

1. Quais as funções básicas da gerência de memória?
2. Considere um sistema computacional com 40 Kb de memória principal e que utilize um sistema operacional de 10 Kb que implemente alocação contígua de memória. Qual a taxa de subutilização da memória principal para um programa que ocupe 20 Kb de memória?
3. Suponha um sistema computacional com 64 Kb de memória principal e que utilize um sistema operacional de 14 Kb que implemente alocação contígua de memória. Considere também um programa de 90 Kb, formado por um módulo principal de 20 Kb e três módulos independentes, cada um com 10 Kb, 20 Kb e 30 Kb. Como o programa poderia ser executado utilizando-se apenas a técnica de overlay?
4. Considerando o exercício anterior, se o módulo de 30 Kb tivesse seu tamanho aumentado para 40 Kb, seria possível executar o programa? Caso não seja possível, como o problema poderia ser contornado?
5. Qual a diferença entre fragmentação interna e externa da memória principal?
6. Suponha um sistema computacional com 128 Kb de memória principal e que utilize um sistema

operacional de 64 Kb que implementa alocação partionada estática relocável. Considere também que o sistema foi inicializado com três partições: P1 (8 Kb), P2 (24 Kb) e P3 (32 Kb). Calcule a fragmentação interna da memória principal após a carga de três programas: PA, PB e PC.

- a) $P1 \leftarrow PA (6 \text{ Kb})$; $P2 \leftarrow PB (20 \text{ Kb})$; $P3 \leftarrow PC (28 \text{ Kb})$
- b) $P1 \leftarrow PA (4 \text{ Kb})$; $P2 \leftarrow PB (16 \text{ Kb})$; $P3 \leftarrow PC (26 \text{ Kb})$
- c) $P1 \leftarrow PA (8 \text{ Kb})$; $P2 \leftarrow PB (24 \text{ Kb})$; $P3 \leftarrow PC (32 \text{ Kb})$

7. Considerando o exercício anterior, seria possível executar quatro programas concorrentemente utilizando apenas a técnica de alocação partionada estática relocável? Se for possível, como? Considerando ainda o mesmo exercício, seria possível executar um programa de 36 Kb? Se for possível, como?
8. Qual a limitação da alocação partionada absoluta em relação à alocação estática relocável?
9. Considere que os processos da tabela a seguir estão aguardando para serem executados e que cada um permanecerá na memória durante o tempo especificado. O sistema operacional ocupa uma área de 20 Kb no início da memória e gerencia a memória utilizando um algoritmo de particionamento dinâmico modificado. A memória total disponível no sistema é de 64 Kb e é alocada em blocos múltiplos de 4 Kb. Os processos são alocados de acordo com sua identificação (em ordem crescente), e irão aguardar até obter a memória de que necessitam. Calcule a perda de memória por fragmentação interna e externa sempre que um processo é colocado ou retirado da memória. O sistema operacional compacta a memória apenas quanto existem duas ou mais partições livres adjacentes.

Processos	Memória	Tempo
1	30 Kb	5
2	6 Kb	10
3	36 Kb	5

10. Considerando as estratégias para escolha da partição dinamicamente, conceitue as estratégias best-fit e worst-fit especificando prós e contras de cada uma.
11. Considere um sistema que possua as seguintes áreas livres na memória principal, ordenadas crescentemente: 10 Kb, 4 Kb, 20 Kb, 18 Kb, 7 Kb, 9 Kb, 12 Kb e 15 Kb. Para cada programa a seguir, qual seria a partição alocada utilizando-se as estratégias first-fit, best-fit e worst-fit?
 - a) 12 Kb
 - b) 10 Kb
 - c) 9 Kb
12. Um sistema utiliza alocação partionada dinâmica como mecanismo de gerência de memória. O sistema operacional aloca uma área de memória total de 50 Kb e possui, inicialmente, os programas da tabela a seguir:

3 Kb	Programa B
10 Kb	Livre
6 Kb	Programa C
26 Kb	Livre

Realize as operações a seguir sequencialmente, mostrando o estado da memória após cada uma delas. Resolva a questão utilizando as estratégias best-fit, worst-fit e first-fit.

- a) alocar uma área para o programa D que possui 6 Kb;
- b) liberar a área do programa A;
- c) alocar uma área para o programa E que possui 4 Kb.

13. O que é swapping e para que é utilizada esta técnica?

14. Por que é importante o uso de um loader com relocação dinâmica para que a técnica de swapping possa ser implementada?

10

Gerência de Memória Virtual

► 10.1 Introdução

No capítulo anterior foram apresentadas diversas técnicas de gerenciamento de memória que evoluíram no sentido de maximizar o número de processos residentes na memória principal e reduzir o problema da fragmentação, porém os esquemas vistos se mostraram muitas vezes ineficientes. Além disso, o tamanho de um programa e de suas estruturas de dados estava limitado ao tamanho da memória disponível. A utilização da técnica de overlay para contornar este problema é de difícil implementação na prática, e nem sempre uma solução garantida.

Memória virtual é uma técnica sofisticada e poderosa de gerência de memória, em que as memórias principal e secundária são combinadas dando ao usuário a ilusão de existir uma memória muito maior que a capacidade real da memória principal. O conceito de memória virtual fundamenta-se em não vincular o endereçamento feito pelo programa dos endereços físicos da memória principal. Desta forma, programas e suas estruturas de dados deixam de estar limitados ao tamanho da memória física disponível, pois podem possuir endereços associados à memória secundária.

Outra vantagem da técnica de memória virtual é permitir um número maior de processos compartilhando a memória principal, já que apenas partes de cada processo estarão residentes. Isto leva a uma utilização mais eficiente também do processador. Além disso, essa técnica possibilita minimizar o problema da fragmentação da memória principal.

A primeira implementação de memória virtual foi realizada no início da década de 1960, no sistema Atlas, desenvolvido na Universidade de Manchester (Kilburn, 1962). Posteriormente, a IBM introduziria este conceito comercialmente na família System/370, em 1972. Atualmente, a maioria dos sistemas implementa memória virtual, com exceção de alguns sistemas operacionais de supercomputadores.

Existe um forte relacionamento entre a gerência da memória virtual e a arquitetura de hardware do sistema computacional. Por motivos de desempenho, é comum que algumas funções da gerência de memória virtual sejam implementadas diretamente no hardware. Além disso, o código do sistema

operacional deve levar em consideração várias características específicas da arquitetura, especialmente o esquema de endereçamento do processador.

Este capítulo apresenta conceitos relacionados à memória virtual e aborda as três técnicas que permitem sua implementação: paginação, segmentação e segmentação com paginação. Apesar da ênfase na gerência de memória virtual por paginação, diversos dos conceitos apresentados também podem ser aplicados nas técnicas de segmentação.

► 10.2 Espaço de Endereçamento Virtual

O conceito de memória virtual se aproxima muito da ideia de um vetor, existente nas linguagens de alto nível. Quando um programa faz referência a um elemento do vetor, não há preocupação em saber a posição de memória daquele dado. O compilador se encarrega de gerar instruções que implementam esse mecanismo, tornando-o totalmente transparente ao programador (Fig. 10.1).

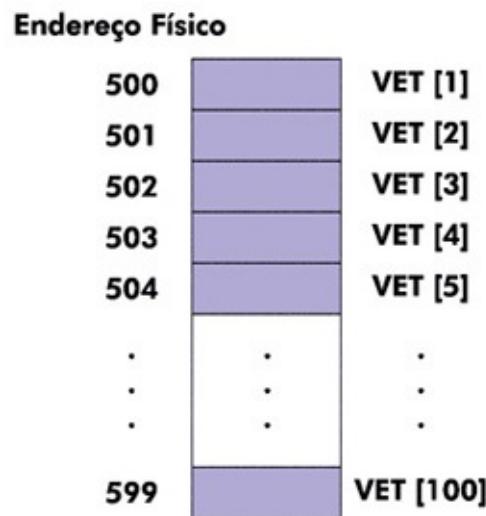


Fig. 10.1 Vetor de 100 posições.

A memória virtual utiliza abstração semelhante, só que em relação aos endereços dos programas e dados. Um programa no ambiente de memória virtual não faz referência a endereços físicos de memória (endereços reais), mas apenas a endereços virtuais. No momento da execução de uma instrução, o endereço virtual referenciado é traduzido para um endereço físico, pois o processador manipula apenas posições da memória principal. O mecanismo de tradução do endereço virtual para endereço físico é denominado mapeamento.

Um processo, conforme apresentado, é formado pelo contexto de hardware, pelo contexto de software e pelo espaço de endereçamento. Em ambientes que implementam memória virtual, o espaço de endereçamento do processo é conhecido como espaço de endereçamento virtual e representa o conjunto de endereços virtuais que o processo pode endereçar. Analogamente, o conjunto de endereços reais que o processador pode referenciar é chamado de espaço de endereçamento real (Fig. 10.2).

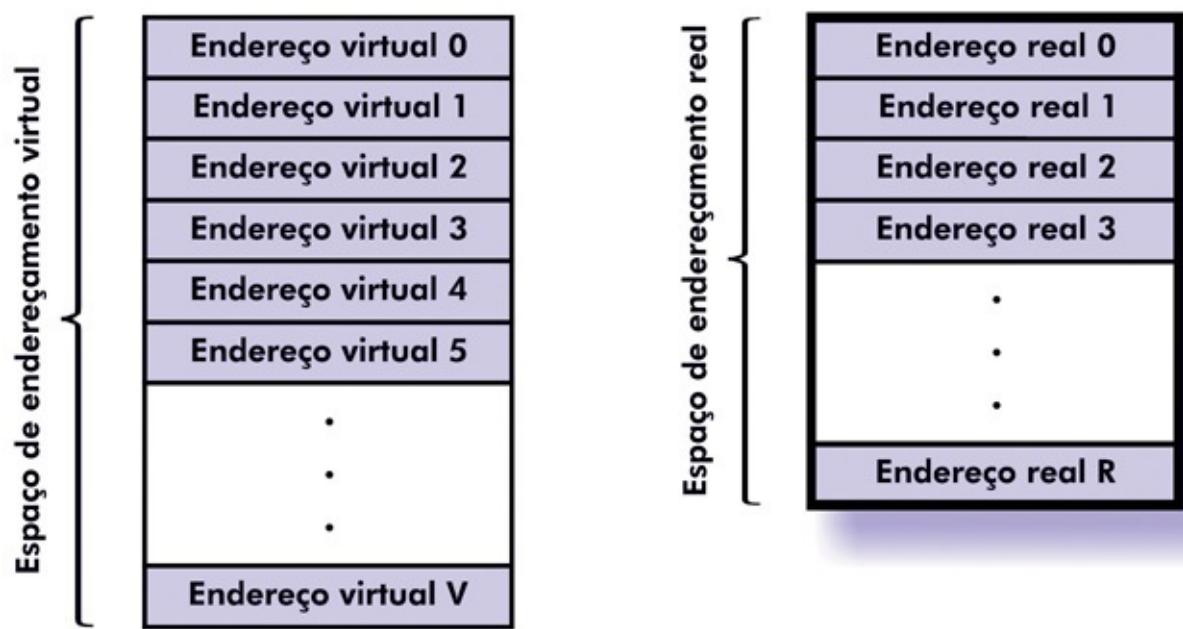


Fig. 10.2 Espaço de endereçamentos virtual e real.

Como o espaço de endereçamento virtual não tem nenhuma relação direta com os endereços no espaço real, um programa pode fazer referência a endereços virtuais que estejam fora dos limites da memória principal, ou seja, os programas e suas estruturas de dados não estão mais limitados ao tamanho da memória física disponível. Para que isso seja possível, o sistema operacional utiliza a memória secundária como extensão da memória principal. Quando um programa é executado, somente uma parte do seu código fica residente na memória principal, permanecendo o restante na memória secundária até o momento de ser referenciado. Esta condição permite aumentar o compartilhamento da memória principal entre muitos processos (Fig. 10.3).

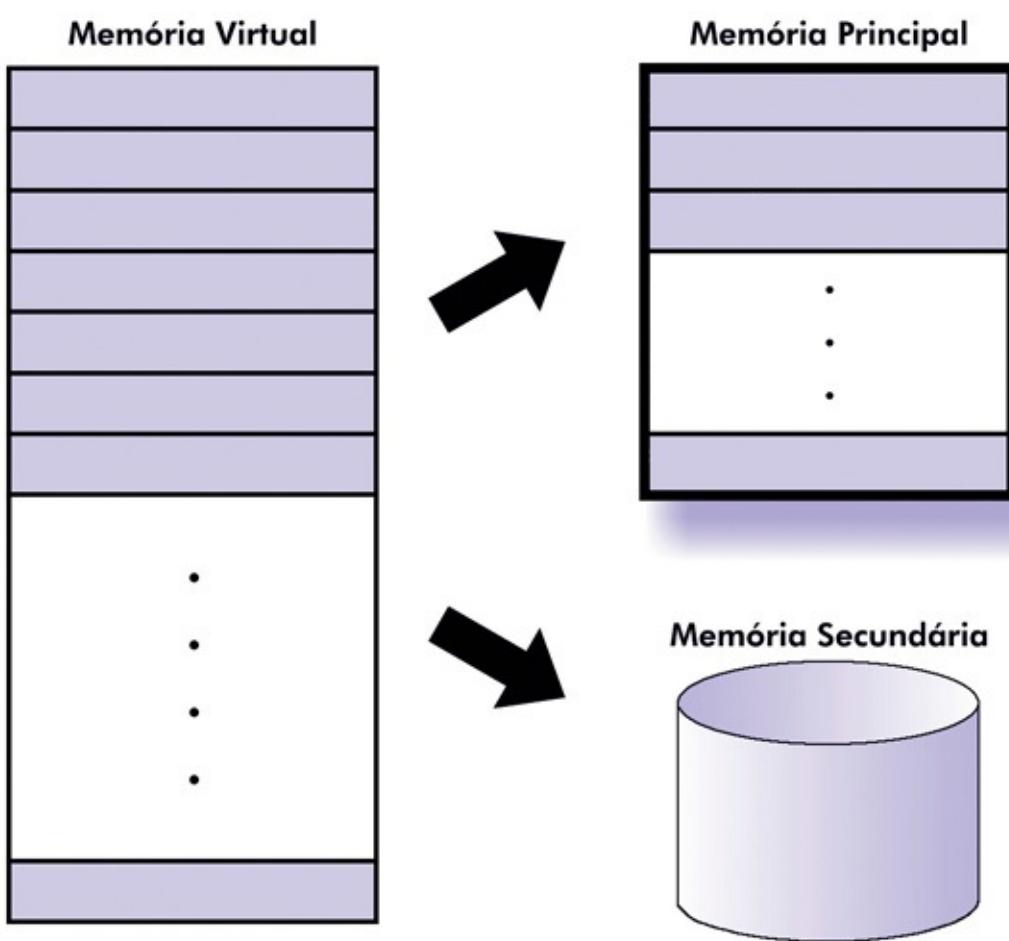


Fig. 10.3 Espaço de endereçamento virtual.

No desenvolvimento de aplicações, a existência dos endereços virtuais é ignorada pelo programador. Os compiladores e linkers se encarregam de gerar o código executável em função do espaço de endereçamento virtual, e o sistema operacional cuida dos detalhes durante sua execução.

► 10.3 Mapeamento

O processador apenas executa instruções e referencia dados residentes no espaço de endereçamento real; portanto, deve existir um mecanismo que transforme os endereços virtuais em endereços reais. Esse mecanismo, conhecido por mapeamento, permite traduzir um endereço localizado no espaço virtual para um associado no espaço real. Como consequência do mapeamento, um programa não mais precisa estar necessariamente em endereços contíguos na memória principal para ser executado (Fig. 10.4).

Nos sistemas modernos, a tarefa de tradução de endereços virtuais é realizada por hardware juntamente com o sistema operacional, de forma a não comprometer seu desempenho e torná-lo transparente a usuários e suas aplicações. O dispositivo de hardware responsável por esta tradução é conhecido como unidade de gerência de memória (*Memory Management Unit* — MMU), sendo acionado sempre que se faz referência a um endereço virtual. Depois de traduzido, o endereço real pode ser utilizado pelo processador para o acesso à memória principal.

Cada processo tem o seu espaço de endereçamento virtual como se possuísse sua própria memória. O mecanismo de tradução se encarrega, então, de manter tabelas de mapeamento exclusivas para cada

processo, relacionando os endereços virtuais do processo às suas posições na memória real (Fig. 10.5).

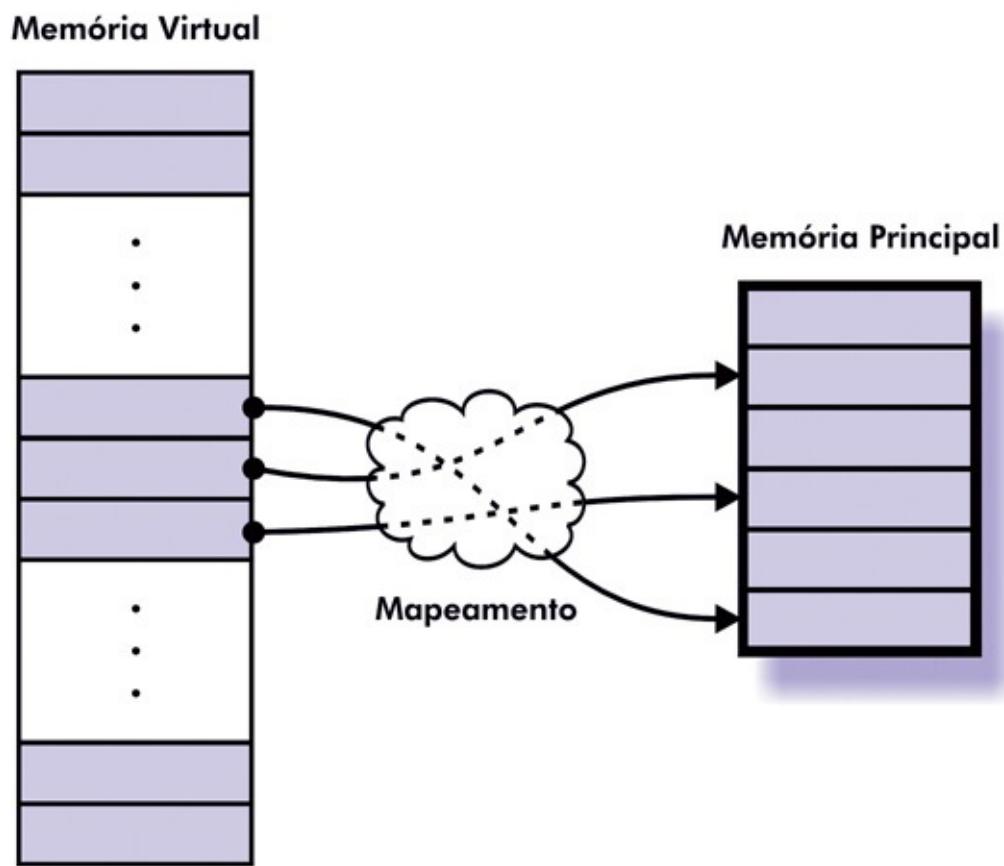


Fig. 10.4 Mapeamento.

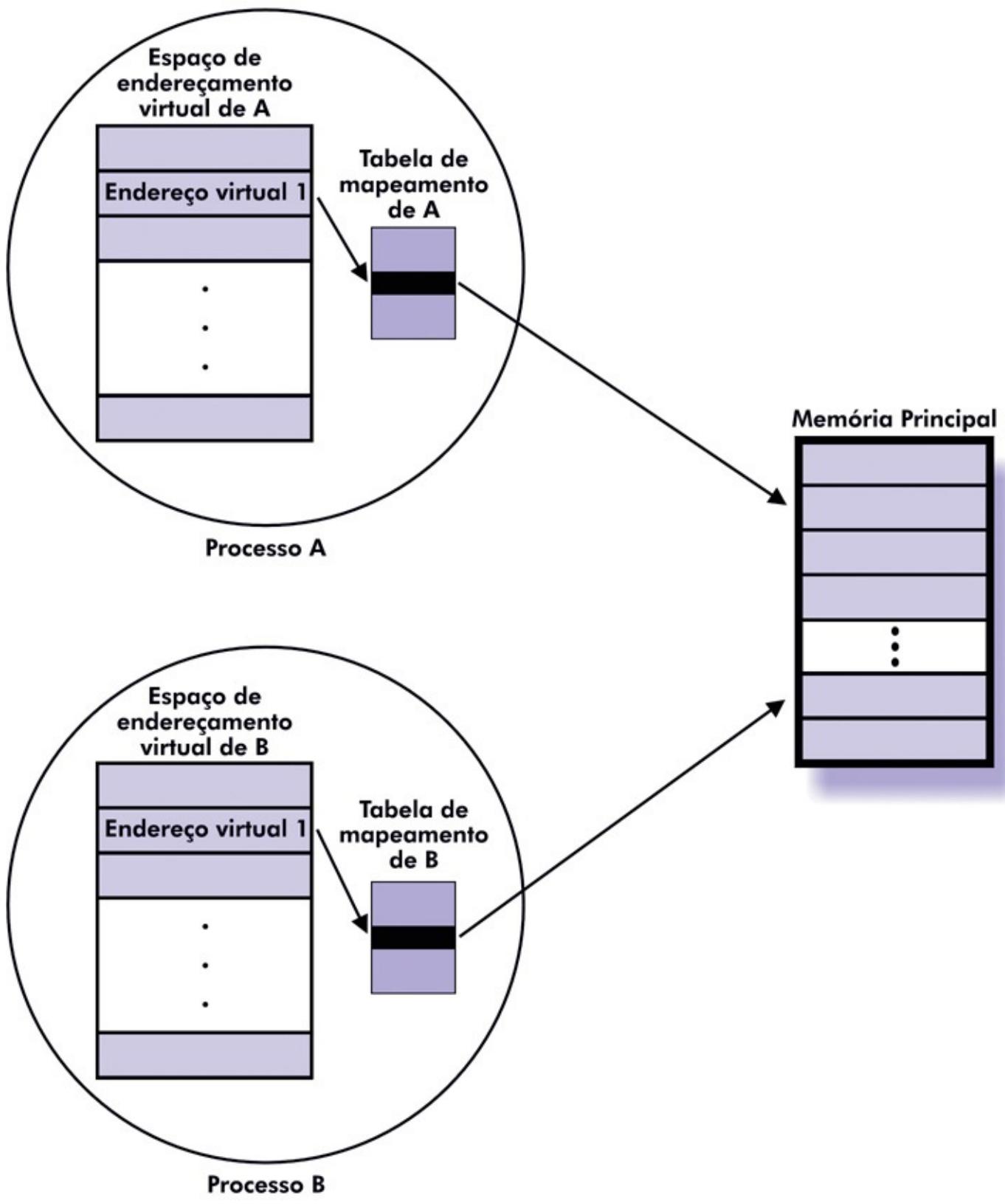


Fig. 10.5 Tabela de mapeamento.

A tabela de mapeamento é uma estrutura de dados existente para cada processo. Quando um determinado processo está sendo executado, o sistema utiliza a tabela de mapeamento do processo em execução para realizar a tradução de seus endereços virtuais. Se um outro processo vai ser executado, o sistema deve passar a referenciar a tabela de mapeamento do novo processo. A troca de tabelas de mapeamento é realizada através de um registrador, que indica a posição inicial da tabela corrente, onde, toda vez que há mudança de contexto, o registrador é atualizado com o endereço da nova tabela.

Caso o mapeamento fosse realizado para cada célula na memória principal, o espaço ocupado pelas tabelas seria tão grande quanto o espaço de endereçamento virtual de cada processo, o que inviabilizaria a implementação do mecanismo de memória virtual. Em função disso, as tabelas mapeiam blocos de dados, cujo tamanho determina o número de entradas existentes nas tabelas de mapeamento. Quanto maior o bloco, menos entradas nas tabelas de mapeamento e, consequentemente, tabelas de mapeamento que ocupam um menor espaço de memória (Tabela 10.1).

Tabela 10.1 Espaço virtual × tamanho do bloco

Espaço de endereçamento virtual	Tamanho do bloco	Número de blocos	Número de entradas na tabela de mapeamento
2^{32} endereços	512 endereços	2^{23}	2^{23}
2^{32} endereços	4 K endereços	2^{20}	2^{20}
2^{64} endereços	4 K endereços	2^{52}	2^{52}
2^{64} endereços	64 K endereços	2^{48}	2^{48}

Como veremos a seguir, existem sistemas operacionais que trabalham apenas com blocos de tamanho fixo (técnica de paginação), enquanto outros utilizam blocos de tamanho variável (técnica de segmentação). Existe ainda um terceiro tipo de sistema que implementa ambas as técnicas (segmentação com paginação).

► 10.4 Memória Virtual por Paginação

A memória virtual por paginação é a técnica de gerência de memória em que o espaço de endereçamento virtual e o espaço de endereçamento real são divididos em blocos de mesmo tamanho chamados páginas. As páginas no espaço virtual são denominadas páginas virtuais, enquanto as páginas no espaço real são chamadas de páginas reais ou frames.

Todo o mapeamento de endereço virtual em real é realizado através de tabelas de páginas. Cada processo possui sua própria tabela de páginas, e cada página virtual do processo possui uma entrada na tabela (entrada na tabela de páginas — ETP), com informações de mapeamento que permitem ao sistema localizar a página real correspondente (Fig. 10.6).

Quando um programa é executado, as páginas virtuais são transferidas da memória secundária para a memória principal e colocadas nos frames. Sempre que um programa fizer referência a um endereço virtual, o mecanismo de mapeamento localizará na ETP da tabela do processo o endereço físico do frame no qual se encontra o endereço real correspondente.

Nessa técnica, o endereço virtual é formado pelo número da página virtual (NPV) e por um deslocamento. O NPV identifica unicamente a página virtual que contém o endereço, funcionando como um índice na tabela de páginas. O deslocamento indica a posição do endereço virtual em relação ao início da página na qual se encontra. O endereço físico é obtido, então, combinando-se o endereço do

frame, localizado na tabela de páginas, com o deslocamento, contido no endereço virtual (Fig. 10.7).

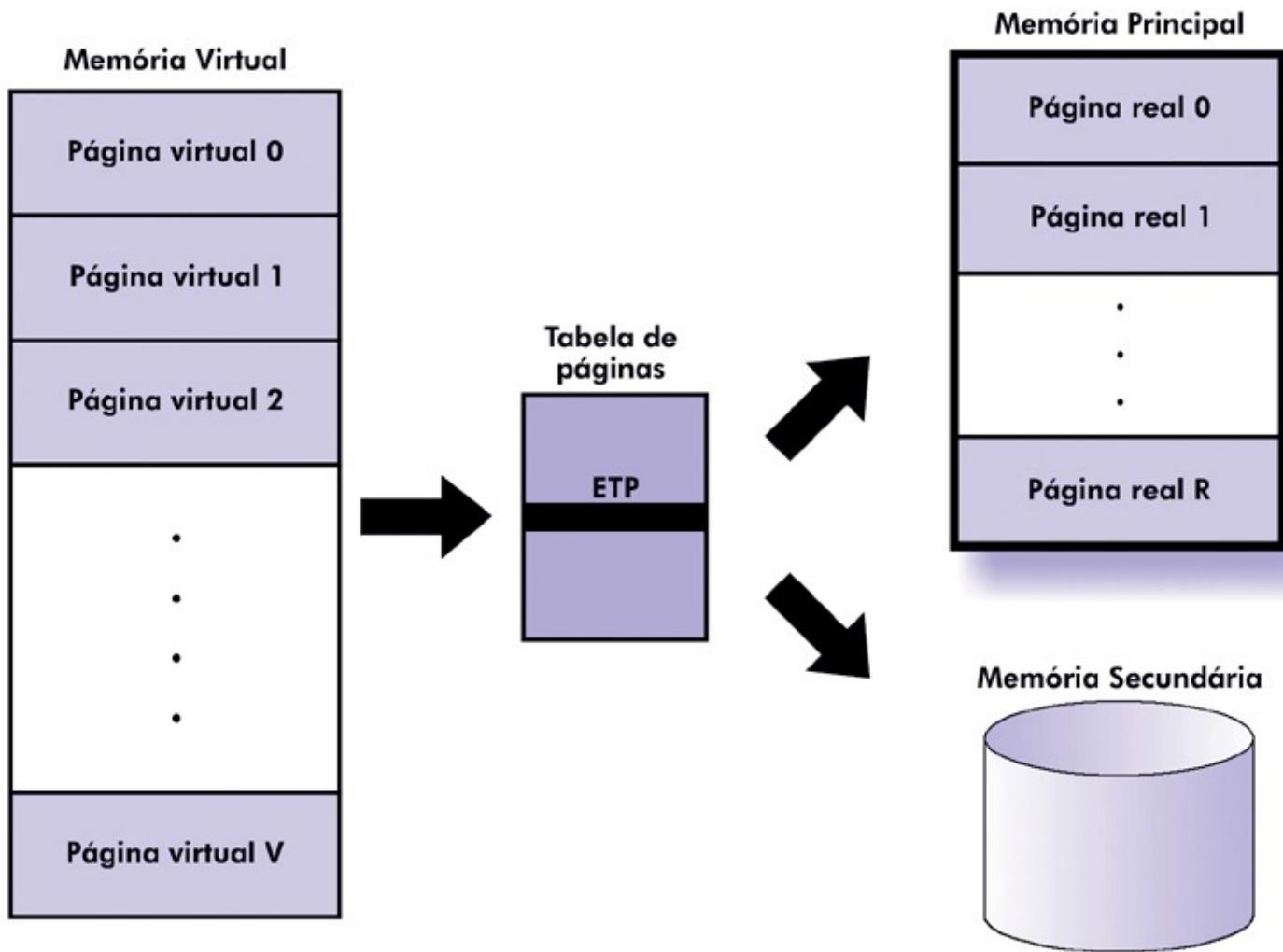


Fig. 10.6 Tabela de páginas.

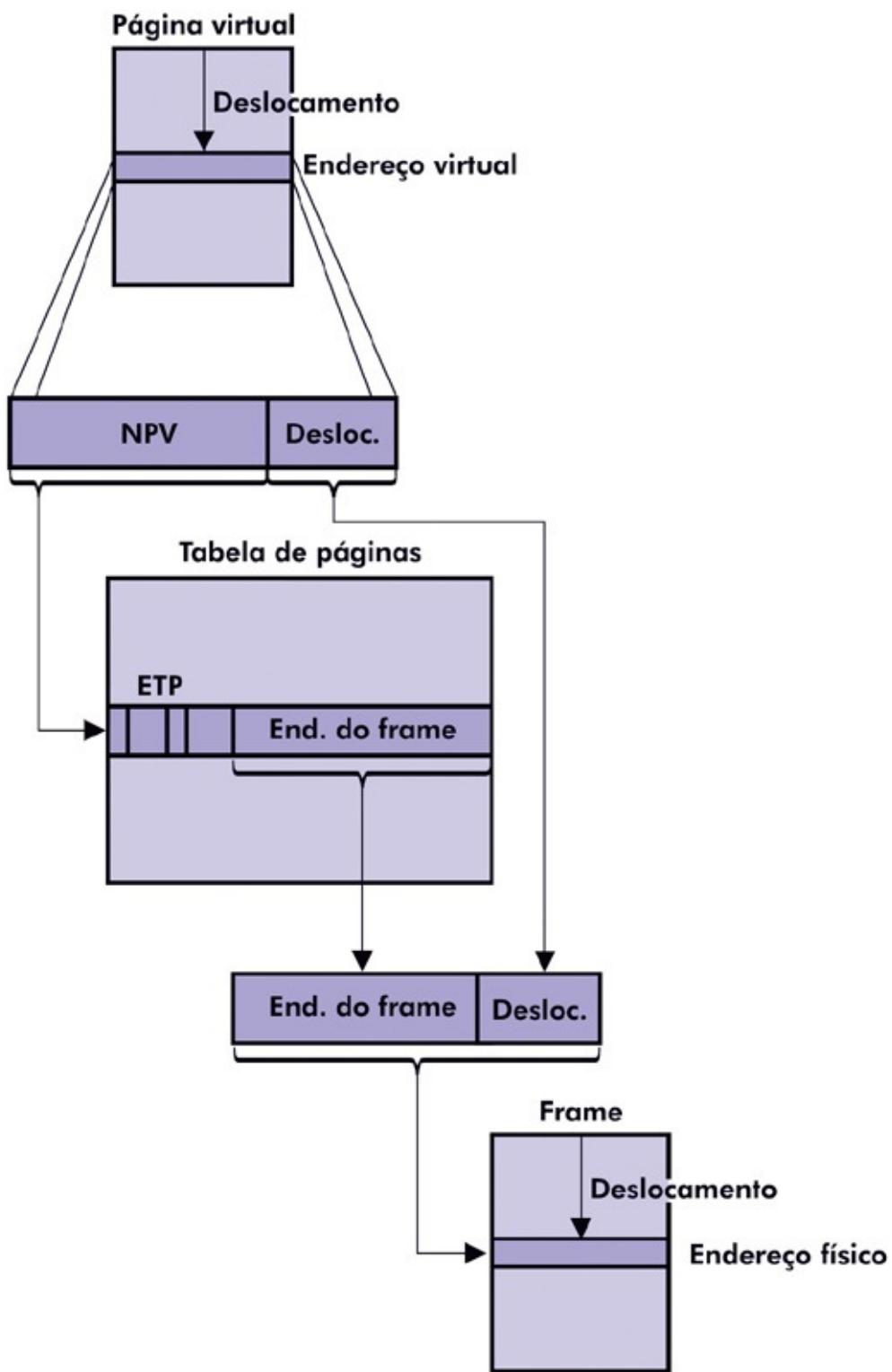


Fig. 10.7 Tradução do endereço virtual.

Além da informação sobre a localização da página virtual, a ETP possui outras informações, como o bit de validade (*valid bit*), que indica se uma página está ou não na memória principal. Se o bit tem o valor 0, isto indica que a página virtual não está na memória principal, mas se é igual a 1, a página está localizada na memória.

Sempre que o processo referencia um endereço virtual, a unidade de gerência de memória verifica, através do bit de validade, se a página que contém o endereço referenciado está ou não na memória principal. Caso a página não esteja na memória, dizemos que ocorreu um *page fault*. Neste caso, o sistema transfere a página da memória secundária para a memória principal, realizando uma operação de

E/S conhecida como page in, ou paginação. O número de page faults gerado por um processo depende de como o programa foi desenvolvido, além da política de gerência de memória implementada pelo sistema operacional (Fig. 10.8).

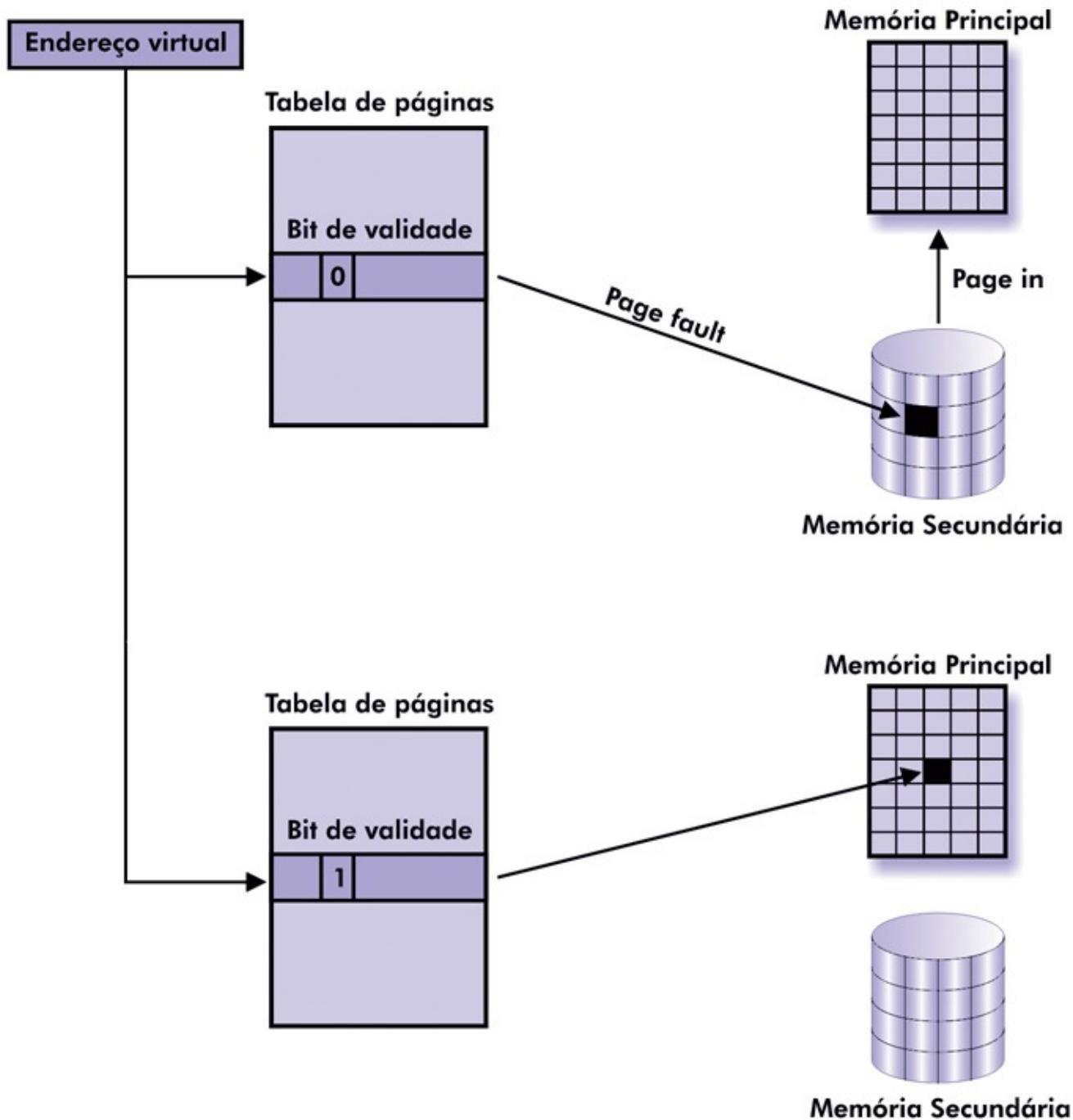


Fig. 10.8 Mecanismo de tradução.

O número de page faults gerados por cada processo em um determinado intervalo de tempo é definido como taxa de paginação do processo. O overhead gerado pelo mecanismo de paginação é inerente à gerência de memória virtual, porém se a taxa de paginação dos processos atingir valores elevados, o excesso de operações de E/S poderá comprometer o desempenho do sistema.

Quando um processo referencia um endereço e ocorre um page fault, o processo em questão passa do estado de execução para o estado de espera, até que a página seja transferida do disco para a memória principal. Na troca de contexto, as informações sobre a tabela de mapeamento são salvas e as

informações do novo processo escalonado são restauradas. Após a transferência da página para a memória principal, o processo é recolocado na fila de processos no estado de pronto, e quando for reescalonado poderá continuar sua execução.

10.4.1 Políticas de Busca de Páginas

O mecanismo de memória virtual permite a execução de um programa sem que seu código esteja completamente residente na memória principal. A política de busca de páginas determina quando uma página deve ser carregada para a memória. Basicamente, existem duas estratégias para este propósito: paginação por demanda e paginação antecipada.

Na paginação por demanda (*demand paging*), as páginas dos processos são transferidas da memória secundária para a principal apenas quando são referenciadas. Este mecanismo é conveniente, na medida em que leva para a memória principal apenas as páginas realmente necessárias à execução do programa. Desse modo, é possível que partes não executadas do programa, como rotinas de tratamento de erros, nunca sejam carregadas para a memória.

Na paginação antecipada (*anticipatory paging* ou *prepaging*), o sistema carrega para a memória principal, além da página referenciada, outras páginas que podem ou não ser necessárias ao processo ao longo do seu processamento. Se imaginarmos que o programa está armazenado sequencialmente no disco, existe uma grande economia de tempo em levar um conjunto de páginas da memória secundária, ao contrário de carregar uma de cada vez. Por outro lado, caso o processo não precise das páginas carregadas antecipadamente, o sistema terá perdido tempo e ocupado memória principal desnecessariamente.

A técnica de paginação antecipada pode ser empregada no momento da criação de um processo ou na ocorrência de um page fault. Quando um processo é criado, diversas páginas do programa na memória secundária devem ser carregadas para a memória principal, gerando um elevado número de page faults e várias operações de leitura em disco. Na medida em que as páginas são carregadas para a memória, a taxa de paginação tende a diminuir. Se o sistema carregar não apenas uma, mas um conjunto de páginas, a taxa de paginação do processo deverá cair imediatamente e estabilizar-se durante um certo período de tempo. Seguindo o mesmo raciocínio, sempre que houver um page fault, o sistema poderá carregar para a memória, além da página referenciada, páginas adicionais, na tentativa de evitar novos page faults e sucessivas operações de leitura em disco.

10.4.2 Políticas de Alocação de Páginas

A política de alocação de páginas determina quantos frames cada processo pode manter na memória principal. Existem, basicamente, duas alternativas: alocação fixa e alocação variável.

Na política de alocação fixa, cada processo tem um número máximo de frames que pode ser utilizado durante a execução do programa. Caso o número de páginas reais seja insuficiente, uma página do processo deve ser descartada para que uma nova seja carregada. O limite de páginas reais pode ser igual para todos os processos ou definido individualmente. Apesar de parecer justo, alocar o mesmo número

de páginas para todos os processos pode não ser uma boa opção, pois a necessidade de memória de cada processo raramente é a mesma. O limite de páginas deve ser definido no momento da criação do processo, com base no tipo da aplicação que será executada. Essa informação faz parte do contexto de software do processo.

Apesar de sua simplicidade, a política de alocação fixa de páginas apresenta dois problemas. Se o número máximo de páginas alocadas for muito pequeno, o processo tenderá a ter um elevado número de page faults, o que pode impactar no desempenho de todo o sistema. Por outro lado, caso o número de páginas seja muito grande, cada processo irá ocupar na memória principal um espaço maior do que o necessário, reduzindo o número de processos residentes e o grau de multiprogramação. Nesse caso, o sistema pode implementar a técnica de swapping, retirando e carregando processos da/para a memória principal.

Na política de alocação variável, o número máximo de páginas alocadas ao processo pode variar durante sua execução em função de sua taxa de paginação e da ocupação da memória principal. Nesse modelo, processos com elevadas taxas de paginação podem ampliar o limite máximo de frames, a fim de reduzir o número de page faults. Da mesma forma, processos com baixas taxas de paginação podem ter páginas realocadas para outros processos. Este mecanismo, apesar de mais flexível, exige que o sistema operacional monitore constantemente o comportamento dos processos, gerando maior overhead.

10.4.3 Políticas de Substituição de Páginas

Em algumas situações, quando um processo atinge o seu limite de alocação de frames e necessita alocar novas páginas na memória principal, o sistema operacional deve selecionar, dentre as diversas páginas alocadas, qual deverá ser liberada. Este mecanismo é chamado de política de substituição de páginas. Uma página real, quando liberada por um processo, está livre para ser utilizada por qualquer outro processo. A partir dessa situação, qualquer estratégia de substituição de páginas deve considerar se uma página foi ou não modificada antes de liberá-la; caso contrário, os dados armazenados na página podem ser perdidos. No caso de páginas contendo código executável, que não sofrem alterações, não existe essa preocupação, pois existe uma cópia do código no arquivo executável em disco. As páginas modificáveis, que armazenam variáveis e estruturas de dados, podem sofrer alterações. Neste caso, o sistema deverá gravá-la na memória secundária antes do descarte, preservando seu conteúdo para uso em futuras referências. Este mecanismo é conhecido como page out (Fig. 10.9). Com este propósito, o sistema mantém um arquivo de paginação (*page file*), onde todas as páginas modificadas e descartadas são armazenadas. Sempre que uma página modificada for novamente referenciada ocorrerá um page in, carregando-a para a memória principal a partir do arquivo de paginação.

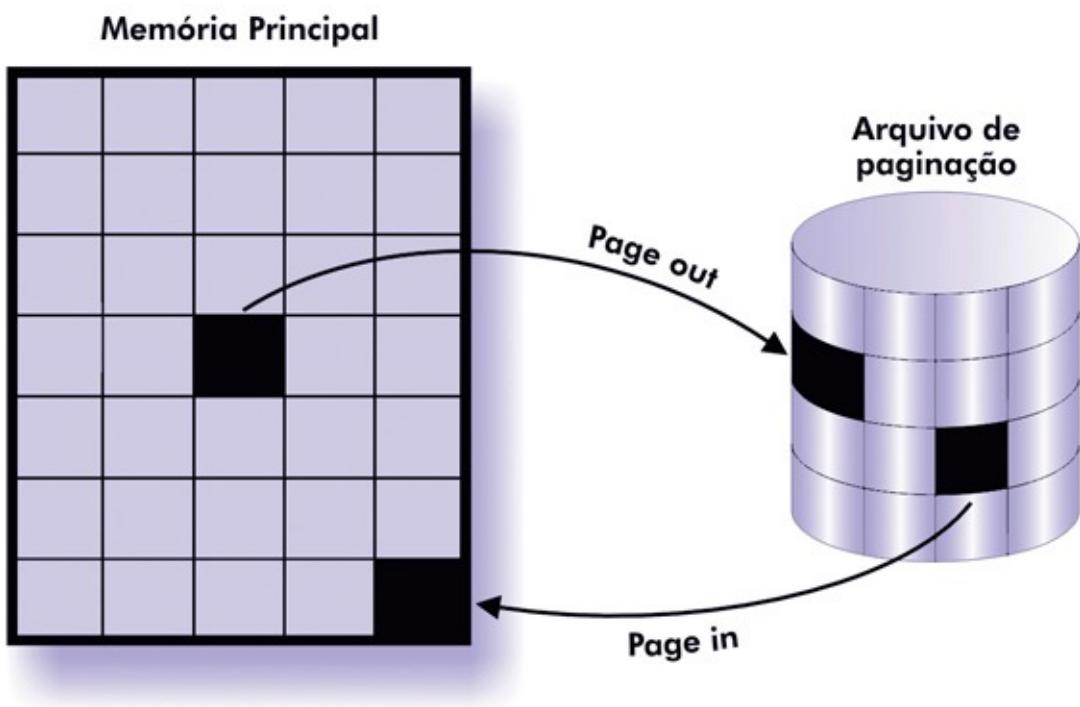


Fig. 10.9 Substituição de páginas.

O sistema operacional consegue identificar as páginas modificadas através de um bit que existe em cada entrada da tabela de páginas, chamado bit de modificação (*dirty bit* ou *modify bit*). Sempre que uma página sofre uma alteração, o valor do bit de modificação é alterado, indicando que a página foi modificada.

A política de substituição de páginas pode ser classificada conforme seu escopo, ou seja, dentre os processos residentes na memória principal quais são candidatos a ter páginas realocadas. Em função desse escopo, a política de substituição pode ser definida como local ou global.

Na política de substituição local, apenas as páginas do processo que gerou o page fault são candidatas a realocação. Nesse modelo, sempre que um processo precisar de uma nova página o sistema deverá selecionar, dentre os frames alocados pelo processo, a página a ser substituída. Os frames dos demais processos não são avaliados para substituição.

Já na política de substituição global, todas as páginas alocadas na memória principal são candidatas à substituição, independente do processo que gerou o page fault. Como qualquer processo pode ser escolhido, é possível que o processo selecionado sofra um aumento na sua taxa de paginação, em função da redução do número de páginas alocadas na memória. Na verdade, nem todas as páginas podem ser candidatas à substituição. Algumas páginas, como as do núcleo do sistema, são marcadas como bloqueadas e não podem ser realocadas.

Existe uma relação entre o escopo da política de substituição e a política de alocação de páginas, apresentada anteriormente. A política de alocação fixa permite apenas a utilização de uma política de substituição local. Nesse caso, sempre que um processo necessita de uma nova página, o sistema deverá selecionar um frame do próprio processo para ser realocado, mantendo assim o seu limite de páginas.

A política de alocação variável permite uma política de substituição tanto local quanto global. Na política de alocação variável com substituição global, quando um processo necessita de uma nova página

o sistema poderá selecionar um frame dentre todas as páginas na memória principal, independente do processo. Nesse caso, o processo escolhido perde uma de suas páginas, reduzindo assim o número de frames alocados na memória pelo processo. Por outro lado, o processo que gerou o page fault recebe um novo frame e tem seu número de páginas aumentado.

Na política de alocação variável com substituição local, quando um processo necessita de nova página o sistema deverá selecionar uma página do próprio processo para substituição. Em função do comportamento do processo e do nível de utilização do sistema, o número de páginas alocadas ao processo pode ser aumentado ou diminuído, a fim de melhorar o desempenho do sistema.

10.4.4 Working Set

Apesar de suas diversas vantagens, o mecanismo de memória virtual introduz um sério problema. Como cada processo possui na memória principal apenas algumas páginas alocadas, o sistema deve manter um conjunto mínimo de frames buscando uma baixa taxa de paginação. Ao mesmo tempo, o sistema operacional deve impedir que os processos tenham um número excessivo de páginas na memória, de forma a aumentar o grau de compartilhamento da memória principal.

Caso os processos tenham na memória principal um número insuficiente de páginas para a execução do programa, é provável que diversos frames referenciados ao longo do seu processamento não estejam na memória. Esta situação provoca a ocorrência de um número elevado de page faults e, como resultado, inúmeras operações de E/S. Neste caso, ocorre um problema conhecido como *thrashing*, provocando sérias consequências ao desempenho do sistema.

O conceito de working set surgiu com o objetivo de reduzir o problema do thrashing e está relacionado ao princípio da localidade. Existem dois tipos de localidade que são observados durante a execução da maioria dos programas. A localidade espacial é a tendência de que após uma referência a uma posição de memória sejam realizadas novas referências a endereços próximos. A localidade temporal é a tendência de que após a referência a uma posição de memória esta mesma posição seja novamente referenciada em um curto intervalo de tempo.

O princípio da localidade significa, na prática, que o processador tenderá a concentrar suas referências a um conjunto de páginas do processo durante um determinado período de tempo. Imaginando um loop, cujo código ocupe três páginas, a tendência de essas três páginas serem referenciadas diversas vezes é muito alta (Fig. 10.10).

No início da execução de um programa observa-se um elevado número de page faults, pois não existe nenhum frame do processo na memória principal. Com o decorrer da sua execução, as páginas são carregadas para a memória e o número de page faults diminui. Após um período de estabilidade, o programa gera novamente uma elevada taxa de paginação, que depois de algum tempo volta a se estabilizar. Esse fenômeno pode repetir-se inúmeras vezes durante a execução de um processo e está relacionado com a forma com que a aplicação foi escrita. Normalmente, se um programa foi desenvolvido utilizando técnicas estruturadas, o conceito da localidade quase sempre é válido. Nesse caso, a localidade será percebida, por exemplo, durante a execução de repetições e sub-rotinas.



Fig. 10.10 Conceito de localidade.

O princípio da localidade é indispensável para que a gerência de memória virtual funcione eficientemente. Como as referências aos endereços de um processo concentram-se em um determinado conjunto de páginas, é possível manter apenas parte do código de cada um dos diversos programas na memória principal, sem prejudicar a execução dos processos. Caso contrário, o sistema teria que manter integralmente o código de todos os programas na memória para evitar o problema do thrashing. Considerando um programa com rotinas de inicialização, um loop principal e rotinas de finalização, manter o programa inteiro na memória principal seria ineficiente. A má utilização da memória fica mais clara quando o programa possui rotinas de tratamento de erros na memória que, muitas vezes, nunca serão executadas.

A partir da observação do princípio da localidade, Peter Denning formulou o modelo de working set (Denning, 1968, 1970, 1980). O conceito de working set é definido como o conjunto das páginas referenciadas por um processo durante determinado intervalo de tempo. A Fig. 10.11 ilustra que no instante t_2 o working set do processo, $W(t_2, \Delta_t)$, consiste nas páginas referenciadas no intervalo Δ_t ($t_2 - t_1$), isto é, as páginas P2, P3 e P8. O intervalo de tempo Δ_t é denominado janela do working set. Podemos observar, então, que o working set de um processo é uma função do tempo e do tamanho da janela do working set.

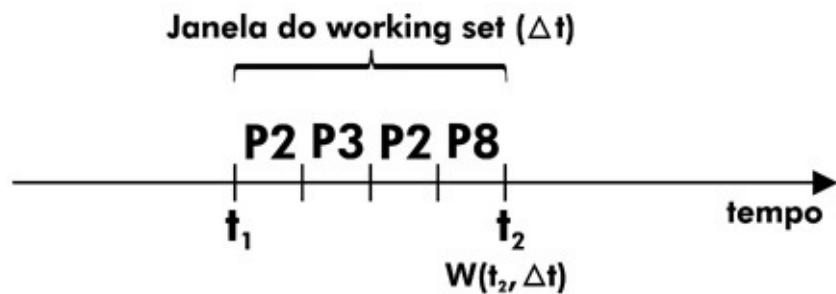


Fig. 10.11 Modelo de working set.

Dentro da janela do working set, o número de páginas distintas referenciadas é conhecido como tamanho do working set. Na Fig. 10.12 são apresentadas as referências às páginas de um processo nas janelas Δt_a ($t_2 - t_1$) e Δt_b ($t_3 - t_2$). O working set do processo no instante t_2 , com a janela Δt_a , corresponde às páginas P2, P3, P4 e P5, e o tamanho do working set é igual a quatro páginas. No instante t_3 , com a janela Δt_b , o working set corresponde às páginas P5 e P6, e o tamanho do working set é igual a duas páginas.

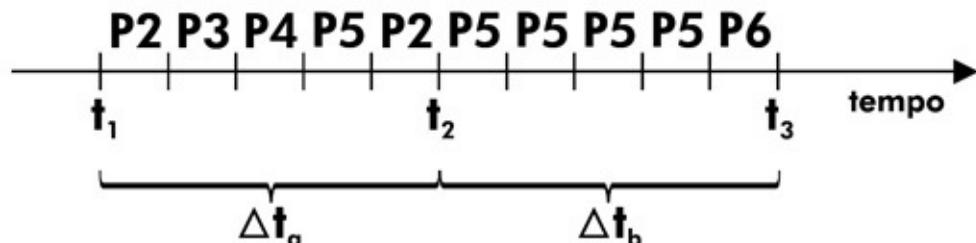


Fig. 10.12 Tamanho do working set.

O modelo de working set proposto por Denning possibilita prever quais páginas são necessárias à execução de um programa de forma eficiente. Caso a janela do working set seja apropriadamente selecionada, em função da localidade do programa, o sistema operacional deverá manter as páginas do working set de cada processo residentes na memória principal. Considerando que a localidade de um programa varia ao longo da sua execução, o tamanho do working set do processo também varia, ou seja, o seu limite de páginas reais deve acompanhar esta variação. O working set refletirá a localidade do programa, reduzindo a taxa de paginação dos processos e evitando, consequentemente, o thrashing.

Caso o limite de páginas reais de um processo seja maior do que o tamanho do working set, menor será a chance de ocorrer uma referência a uma página que não esteja na memória principal. Por outro lado, as páginas dos processos ocuparão espaço excessivo, reduzindo o grau de compartilhamento da memória. No caso de o limite de páginas reais ser menor, a taxa de paginação será alta, pois parte do working set não estará residente na memória principal. Outro fato que pode ser observado é a existência de um ponto onde o aumento do limite de páginas reais do processo não implica a diminuição significativa da taxa de paginação, sendo este ponto alcançado muito antes de o programa ser totalmente carregado para a memória (Fig. 10.13).

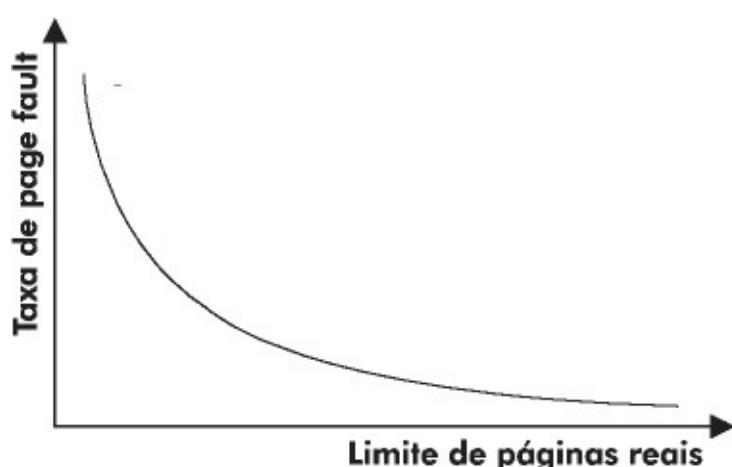


Fig. 10.13 Taxa de page fault × limite de páginas reais.

Apesar de o conceito de working set ser bastante intuitivo, sua implementação não é simples por questões de desempenho. Para implementar esse modelo, o sistema operacional deve garantir que o working set de cada processo permaneça na memória principal, determinando quais páginas devem ser mantidas e retiradas em função da última janela de tempo. Em função disso, o modelo de working set deve ser implementado somente em sistemas que utilizam a política de alocação de páginas variável, onde o limite de páginas reais não é fixo.

Uma maneira de implementar o modelo de working set é analisar a taxa de paginação de cada processo, conhecida como estratégia de frequência de page fault. Caso um processo tenha uma taxa de paginação acima de um limite definido pelo sistema, o processo deverá aumentar o seu limite de páginas reais na tentativa de alcançar o seu working set. Por outro lado, se o processo tem uma taxa de paginação abaixo de um certo limite, o sistema poderá reduzir o limite de páginas sem comprometer seu desempenho. O sistema operacional OpenVMS implementa uma estratégia semelhante a esta. Na prática, o modelo de working set serve como base para inúmeros algoritmos de substituição de páginas, como os apresentados neste capítulo.

10.4.5 Algoritmos de Substituição de Páginas

O maior problema na gerência de memória virtual por paginação não é decidir quais páginas carregar para a memória principal, mas quais liberar (Denning, 1968). Quando um processo necessita de uma nova página e não existem frames disponíveis, o sistema deverá selecionar, dentre as diversas páginas alocadas na memória, qual deverá ser liberada pelo processo.

Os algoritmos de substituição de páginas têm o objetivo de selecionar os frames que tenham as menores chances de serem referenciados em um futuro próximo; caso contrário, o frame poderia retornar diversas vezes para a memória principal, gerando vários page faults e acessos à memória secundária. A partir do princípio da localidade, a maioria dos algoritmos tenta prever o comportamento futuro das aplicações em função do comportamento passado, avaliando o número de vezes que uma página foi referenciada, o momento em que foi carregada para a memória principal e o intervalo de tempo da última referência.

A melhor estratégia de substituição de páginas seria aquela que escolhesse um frame que não fosse mais utilizado no futuro ou levasse mais tempo para ser novamente referenciado. Porém, quanto mais sofisticado o algoritmo de substituição, maior overhead para o sistema operacional implementá-lo. O algoritmo de substituição deve tentar manter o working set dos processos na memória principal e, ao mesmo tempo, não comprometer o desempenho do sistema. A seguir, analisaremos os principais algoritmos existentes para a substituição de páginas.

■ Ótimo

O algoritmo ótimo seleciona para substituição uma página que não será mais referenciada no futuro ou aquela que levará o maior intervalo de tempo para ser novamente utilizada. Apesar de este algoritmo garantir as menores taxas de paginação para os processos, na prática é impossível de ser implementado, pois o sistema operacional não tem como conhecer o comportamento futuro das aplicações. Essa estratégia é utilizada apenas como modelo comparativo na análise de outros algoritmos de substituição.

■ Aleatório

O algoritmo aleatório, como o nome já sugere, não utiliza critério algum de seleção. Todas as páginas alocadas na memória principal têm a mesma chance de serem selecionadas, inclusive os frames que são frequentemente referenciados. Apesar de ser uma estratégia que consome poucos recursos do sistema, é raramente implementada, em função de sua baixa eficiência.

■ FIFO (First-In-First-Out)

No algoritmo FIFO, a página que primeiro foi utilizada será a primeira a ser escolhida, ou seja, o algoritmo seleciona a página que está há mais tempo na memória principal. O algoritmo pode ser implementado associando-se a cada página o momento em que foi carregada para a memória ou utilizando-se uma estrutura de fila, onde as páginas mais antigas estão no início e as mais recentes no final (Fig. 10.14).

Parece razoável pensar que uma página que esteja há mais tempo na memória seja justamente aquela que deva ser selecionada, porém isto nem sempre é verdadeiro. No caso de uma página que seja constantemente referenciada, como é o caso de páginas que contêm dados, o fator tempo torna-se irrelevante e o sistema tem que referenciar a mesma página diversas vezes ao longo do processamento.

O algoritmo FIFO é raramente implementado sem algum outro mecanismo que minimize o problema da seleção de páginas antigas que são constantemente referenciadas.

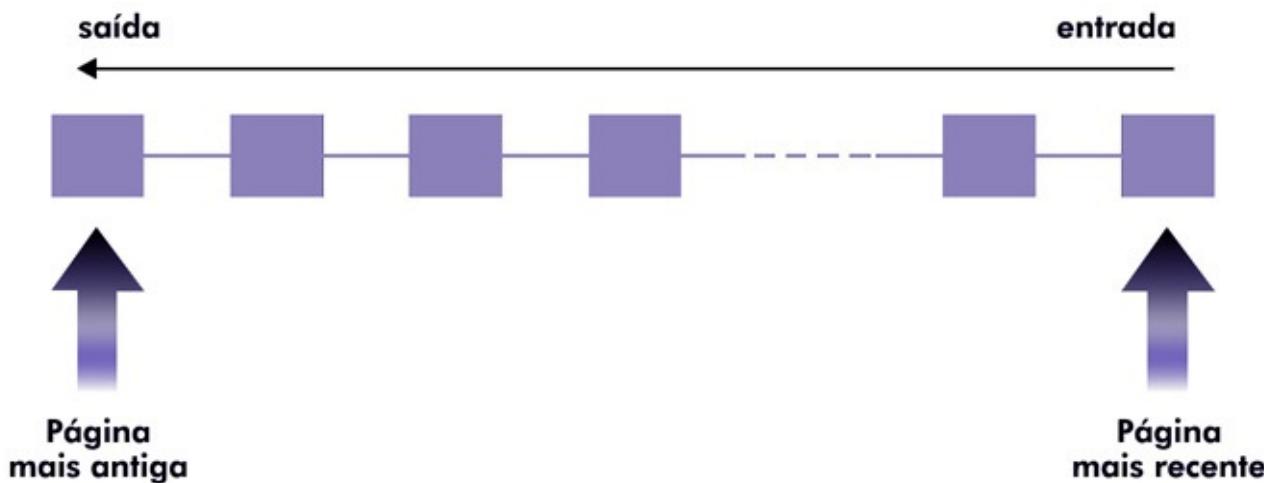


Fig. 10.14 FIFO.

■ LFU (Least-Frequently-Used)

O algoritmo LFU seleciona a página menos referenciada, ou seja, o frame menos utilizado. Para isso, é mantido um contador com o número de referências para cada página na memória principal. A página que possuir o contador com o menor número de referências será escolhida, ou seja, o algoritmo evita selecionar páginas que são bastante utilizadas.

Inicialmente, esta parece ser uma boa estratégia, porém as páginas que estão há pouco tempo na

memória principal podem ser justamente aquelas selecionadas, pois seus contadores estarão com o menor número de referências. É possível também que uma página muito utilizada no passado não seja mais referenciada no futuro. Nesse caso, como o contador possuiria um número elevado de referências, a página não seria selecionada para substituição. Este esquema, como apresentado, é raramente implementado, servindo apenas de base para outros algoritmos de substituição.

■ LRU (Least-Recently-Used)

O algoritmo LRU seleciona a página na memória principal que está há mais tempo sem ser referenciada. Se considerarmos o princípio da localidade, é provável que uma página que não foi utilizada recentemente não seja referenciada novamente em um futuro próximo.

Para implementar esse algoritmo, é necessário que cada página tenha associado o momento do último acesso, que deve ser atualizado a cada referência a um frame. Quando for necessário substituir uma página, o sistema fará uma busca por um frame que esteja há mais tempo sem ser referenciado. Outra maneira de implementar o LRU seria através de uma lista encadeada, onde todas as páginas estariam ordenadas pelo momento da última referência. Neste caso, cada acesso à memória exigiria um acesso à lista.

Apesar de ser uma estratégia com uma eficiência comparável ao algoritmo ótimo, é pouco empregada na prática devido ao seu elevado custo de implementação.

■ NRU (Not-Recently-Used)

O algoritmo NRU é bastante semelhante ao LRU, porém com menor sofisticação. Para a implementação deste algoritmo é necessário um bit adicional, conhecido como bit de referência (BR). O bit indica se a página foi utilizada recentemente e está presente em cada entrada da tabela de páginas.

Quando uma página é carregada para a memória principal, o bit de referência é alterado pelo hardware, indicando que a página foi referenciada ($BR = 1$). Periodicamente, o sistema altera o valor do bit de referência ($BR = 0$), e à medida que as páginas são utilizadas, o bit associado a cada frame retorna para 1. Desta forma, é possível distinguir quais frames foram recentemente referenciados. No momento da substituição de uma página, o sistema seleciona um dos frames que não tenha sido utilizado recentemente, ou seja, com o bit de referência igual a zero.

Tabela 10.2 Bits de referência e modificação

Categorias	Bits avaliados	Resultado
1	$BR = 0$ e $BM = 0$	Página não referenciada e não modificada.
2	$BR = 0$ e $BM = 1$	Página não referenciada e modificada.
3	$BR = 1$ e $BM = 0$	Página referenciada e não modificada.
4	$BR = 1$ e $BM = 1$	Página referenciada e modificada.

O algoritmo NRU torna-se mais eficiente se o bit de modificação for utilizado em conjunto com o bit de referência. Neste caso, é possível classificar as páginas em quatro categorias (Tabela 10.2). O algoritmo, inicialmente, seleciona as páginas que não foram utilizadas recentemente e não foram modificadas, evitando assim um page out. O próximo passo é substituir as páginas que não tenham sido referenciadas recentemente, porém modificadas. Neste caso, apesar de existir um acesso à memória secundária para a gravação da página modificada, seguindo o princípio da localidade, há pouca chance de essa página ser novamente referenciada.

■ FIFO com buffer de páginas

O algoritmo FIFO, anteriormente apresentado, é de fácil implementação e baixo overhead, porém apresenta uma séria limitação quando páginas antigas são constantemente referenciadas. Uma maneira de torná-lo uma política de substituição eficiente é implementar variações no algoritmo.

O algoritmo FIFO com buffer de páginas combina uma lista de páginas alocadas (LPA) com uma lista de páginas livres (LPL). A LPA organiza todas as páginas que estão sendo utilizadas na memória principal, podendo ser implementada como uma lista única para todos os processos ou uma lista individual para cada processo. Independentemente da política utilizada, a LPA organiza as páginas alocadas há mais tempo na memória no início da lista, enquanto as páginas mais recentes ficam no seu final. Da mesma forma, a LPL organiza todos os frames livres da memória principal, sendo que as páginas livres há mais tempo estão no início e as mais recentes no final. Sempre que um processo necessita alocar uma nova página, o sistema utiliza a primeira página da LPL, colocando-a no final da LPA (Fig. 10.15a). Caso o processo tenha que liberar uma página, o mecanismo de substituição seleciona o frame em uso há mais tempo na memória, isto é, o primeiro da LPA, colocando-o no final da LPL (Fig. 10.15b).

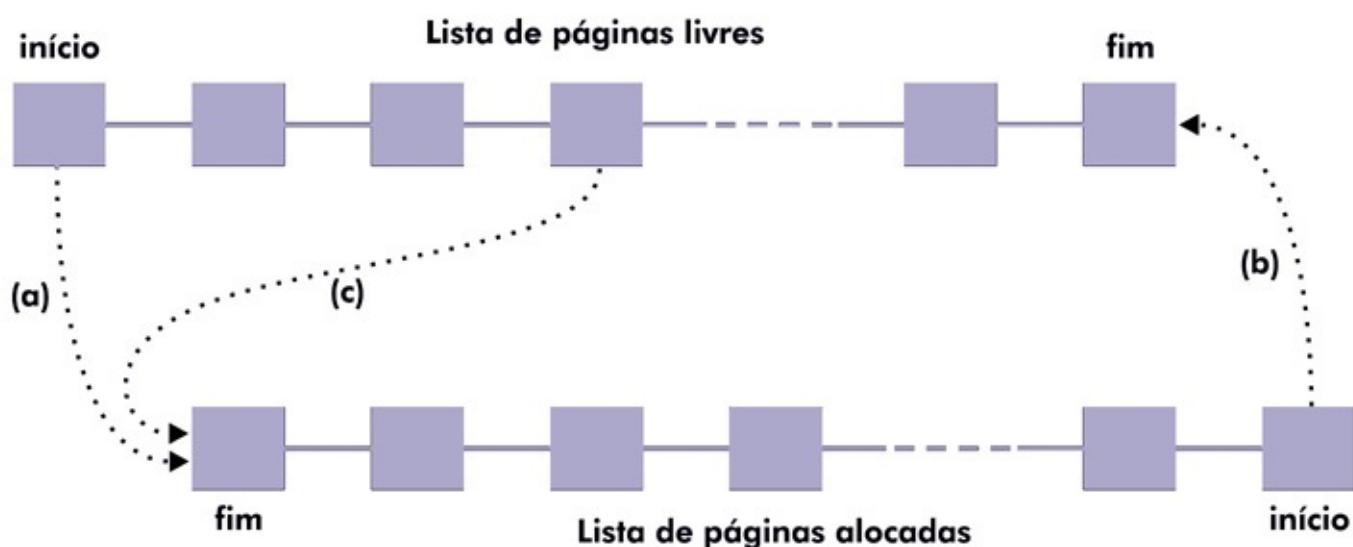


Fig. 10.15 FIFO com buffer de páginas.

É importante notar que a página selecionada e que entrou na LPL continua disponível na memória principal por um determinado intervalo de tempo. Caso esta página seja novamente referenciada e

ainda não tenha sido alocada, basta retirá-la da LPL e devolvê-la ao processo (Fig. 10.15c). Nesse caso, a LPL funciona como um buffer de páginas, evitando o acesso à memória secundária. Por outro lado, se a página não for mais referenciada, com o passar do tempo irá chegar ao início da LPL, quando será utilizada para um outro processo. Caso a página seja posteriormente referenciada, o sistema terá que carregá-la novamente da memória secundária.

O buffer de páginas permite criar um algoritmo de substituição de páginas simples e eficiente, sem o custo de outras implementações. O sistema operacional Mach utiliza o esquema de buffer de páginas único, enquanto o OpenVMS e o MS Windows utilizam dois buffers: um para páginas livres (free page list) e outro para páginas modificadas (modified page list).

■ FIFO circular (clock)

O algoritmo FIFO circular utiliza como base o FIFO, porém as páginas alocadas na memória estão em uma estrutura de lista circular, semelhante a um relógio. Este algoritmo é implementado, com pequenas variações, na maioria dos sistemas Unix.

Para a implementação do algoritmo existe um ponteiro que guarda a posição da página mais antiga na lista (Fig. 10.16a). Cada página possui associado um bit de referência, indicando se a página foi recentemente referenciada. Quando é necessário substituir uma página, o sistema verifica se o frame apontado tem o bit de referência desligado ($BR = 0$). Nesse caso, a página é selecionada para descarte, pois, além de ser a mais antiga, não foi utilizada recentemente. Por outro lado, se a página apontada tem o bit de referência ligado ($BR = 1$), o bit é desligado e o ponteiro incrementado, pois, apesar de ser a página mais antiga, foi utilizada recentemente. O processo se repete até ser encontrada uma página com bit de referência igual a zero (Fig. 10.16b).

Neste algoritmo, existe a possibilidade de todos os frames possuírem o bit de referência ligado. Nesse caso, o ponteiro percorrerá toda a lista, desligando o bit de referência de cada página. Ao final, a página mais antiga é selecionada. A utilização do bit de referência permite conceder a cada página uma segunda chance antes de ser substituída. É possível melhorar a eficiência do algoritmo utilizando o bit de modificação, juntamente com o bit de referência, como apresentado no esquema NRU.

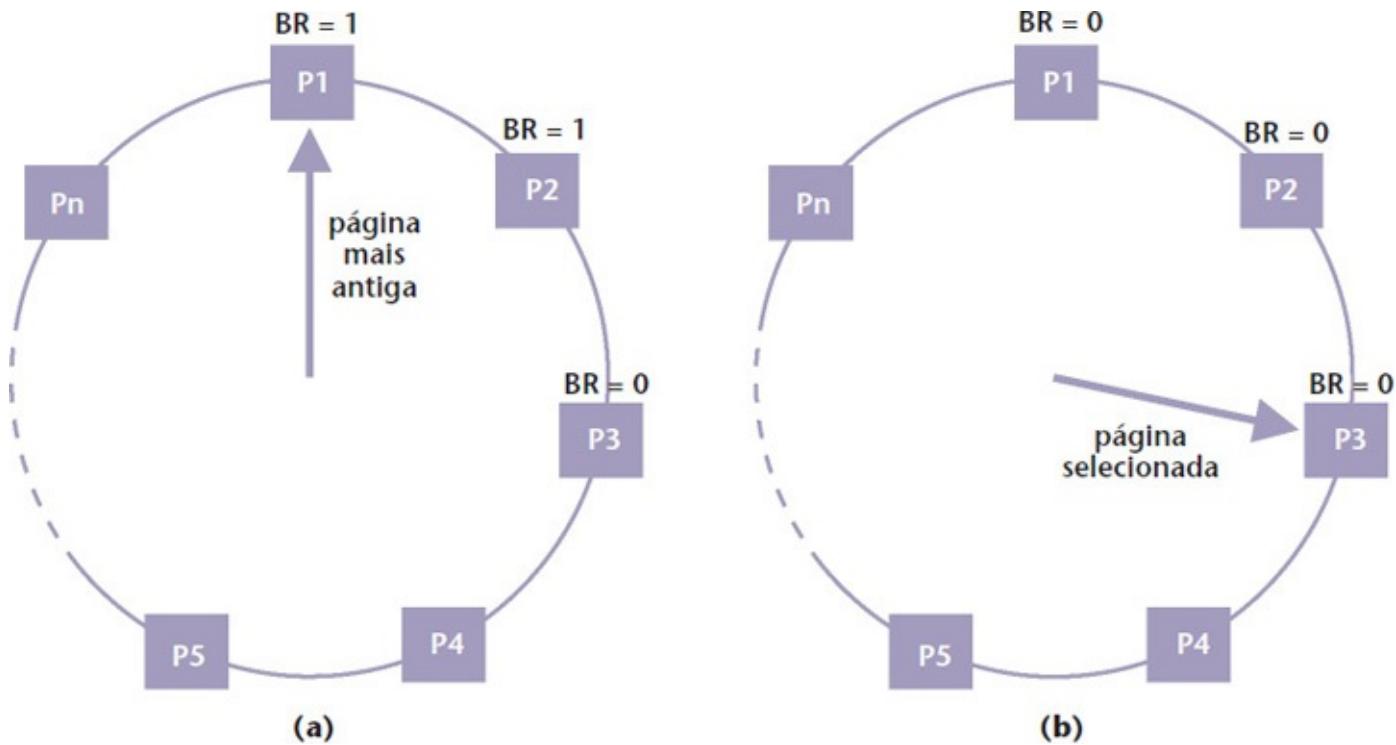


Fig. 10.16 FIFO circular.

10.4.6 Tamanho de Página

A definição do tamanho de página é um fator importante no projeto de sistemas que implementam memória virtual por paginação. O tamanho da página está associado à arquitetura do hardware e varia de acordo com o processador, mas normalmente está entre 512 e 16 M endereços. Algumas arquiteturas permitem a configuração do tamanho de página, oferecendo assim maior flexibilidade.

O tamanho da página tem impacto direto sobre o número de entradas na tabela de páginas e, consequentemente, no tamanho da tabela e no espaço ocupado na memória principal. Por exemplo, em uma arquitetura de 32 bits para endereçamento e páginas de 4 K endereços, teríamos tabelas de páginas de até 2^{20} entradas. Se cada entrada ocupasse 4 bytes, poderíamos ter tabelas de páginas de 4 Mb por processo. Logo, páginas pequenas necessitam de tabelas de mapeamento maiores, provocam maior taxa de paginação e aumentam o número de acessos à memória secundária.

Apesar de páginas grandes tornarem menor o tamanho das tabelas de páginas, ocorre o problema da fragmentação interna. Como podemos observar na Fig. 10.17, o programa ocupa quase que integralmente todas as páginas. A fragmentação só é encontrada, realmente, na última página, quando o código não ocupa o frame por completo.

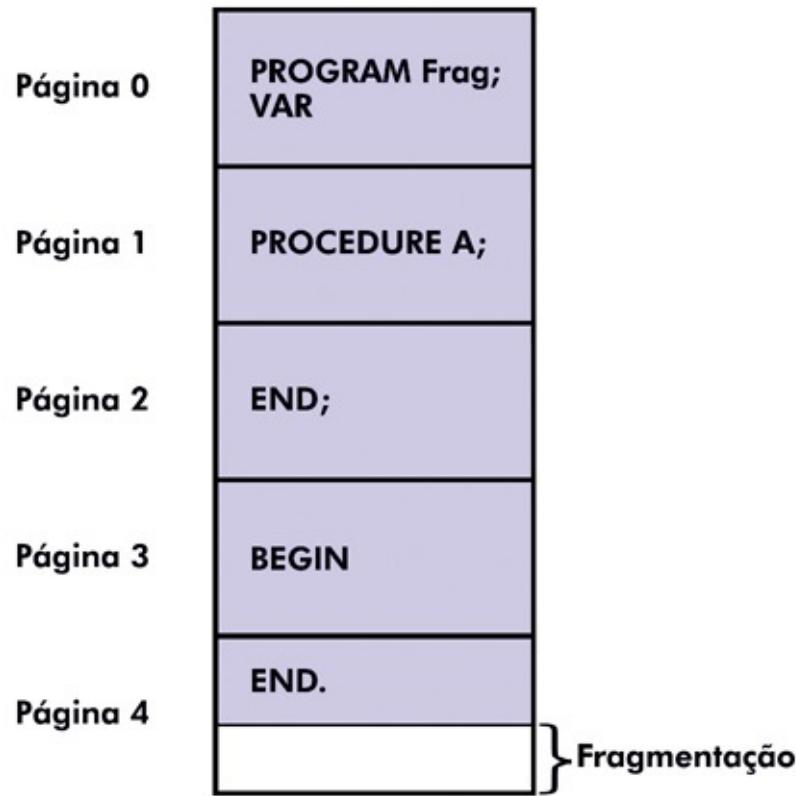


Fig. 10.17 Fragmentação interna.

O principal argumento a favor do uso de páginas pequenas é a melhor utilização da memória principal. A partir do princípio da localidade, com páginas pequenas teríamos na memória apenas as partes dos programas com maiores chances de serem executadas. Quanto maior o tamanho de página, maiores as chances de ter na memória código pouco referenciado, ocupando espaço desnecessariamente. Além disso, páginas pequenas reduzem o problema da fragmentação interna.

Outra preocupação quanto ao tamanho da página é a relacionada aos tempos de leitura e gravação na memória secundária. Devido ao modo de funcionamento dos discos, o tempo de operações de E/S com duas páginas de 512 bytes é muito maior do que em uma página de 1024 bytes.

Com o aumento do espaço de endereçamento e da velocidade de acesso à memória principal, a tendência no projeto de sistemas operacionais com memória virtual por paginação é a adoção de páginas maiores, apesar dos problemas citados.

10.4.7 Paginação em Múltiplos Níveis

Em sistemas que implementam apenas um nível de paginação, o tamanho das tabelas de páginas pode ser um problema. Como já visto, em uma arquitetura de 32 bits para endereçamento e páginas com 4 K endereços por processo, onde cada entrada na tabela de páginas ocupe 4 bytes, a tabela de páginas poderia ter mais de um milhão de entradas e ocuparia 4 Mb de espaço (Fig. 10.18). Imaginando vários processos residentes na memória principal, manter tabelas desse tamanho para cada processo certamente seria de difícil gerenciamento.

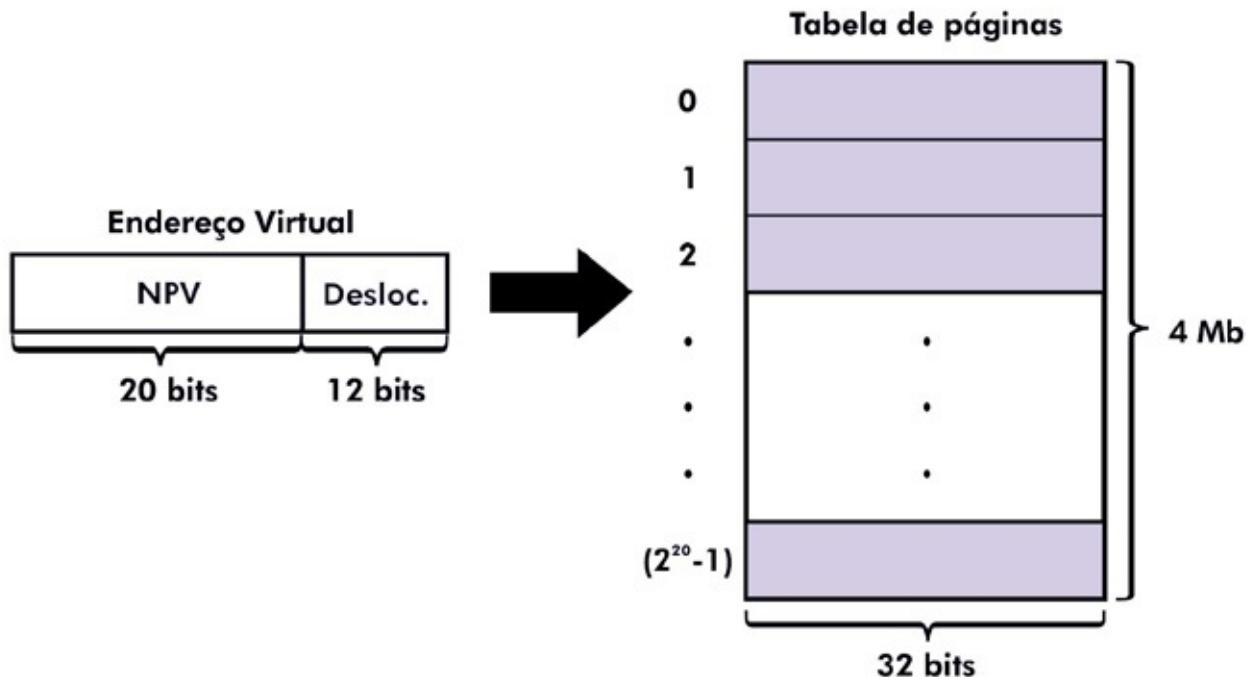


Fig. 10.18 Paginação em um nível.

Uma boa solução para contornar o problema apresentado é a utilização de tabelas de páginas em múltiplos níveis. A ideia é que o princípio da localidade seja aplicado também às tabelas de mapeamento — apenas as informações sobre páginas realmente necessárias aos processos estariam residentes na memória principal.

No esquema de paginação em dois níveis existe uma tabela-diretório, onde cada entrada aponta para uma tabela de página. A partir do exemplo anterior, podemos dividir o campo NPV em duas partes: número da página virtual de nível 1 (NPV1) e número da página virtual de nível 2 (NPV2), cada um com 10 bits. O NPV1 permite localizar a tabela de páginas na tabela-diretório; por sua vez, o NPV2 permite localizar o frame desejado na tabela de páginas (Fig. 10.19).

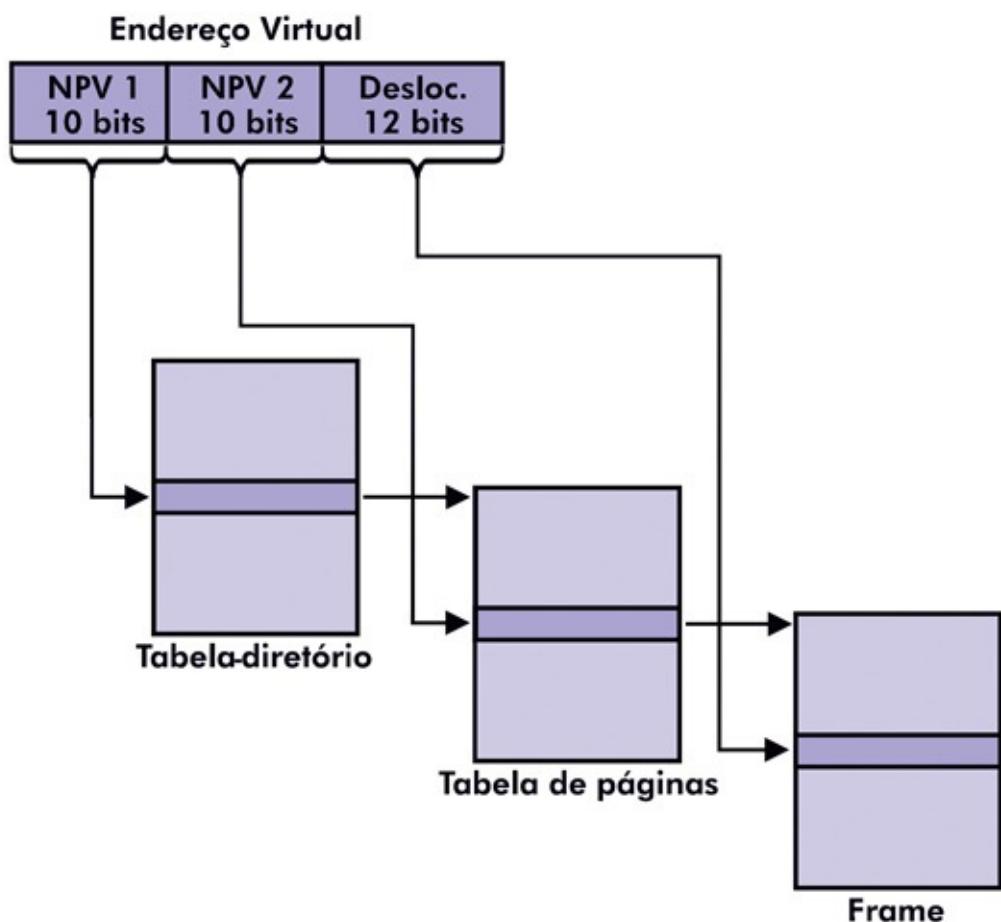


Fig. 10.19 Endereço virtual em dois níveis.

Utilizando-se o exemplo anterior, é possível que existam 1024 tabelas de páginas para cada processo (Fig. 10.20). A grande vantagem da paginação em múltiplos níveis é que apenas estarão residentes na memória principal as tabelas realmente necessárias aos processos, reduzindo, dessa forma, o espaço ocupado na memória. A arquitetura VAX da Compaq/Digital é um exemplo de implementação de paginação em dois níveis.

Em uma arquitetura de 64 bits, a estrutura em dois níveis já não é mais adequada devido ao espaço de endereçamento de 2^{64} . Considerando páginas de 4 K endereços, NPV2 com 10 bits e NPV1 com 42 bits, teríamos tabelas com 2^{42} entradas. Novamente a solução passa por dividir a tabela-diretório, criando uma estrutura em três níveis. Por exemplo, NPV1 com 32 bits, NPV2 com 10 bits e NPV3 com 10 bits. Esse tipo de implementação pode ser encontrado em alguns processadores da família SPARC, da Sun Microsystems.

A técnica de paginação em múltiplos níveis pode ser estendida para quatro níveis, como nos processadores Motorola 68030, cinco ou mais níveis. A cada nível introduzido há, pelo menos, mais um acesso à memória principal, o que sem dúvida gera problemas de desempenho. Tais problemas podem ser contornados utilizando-se caches, como será apresentado a seguir.

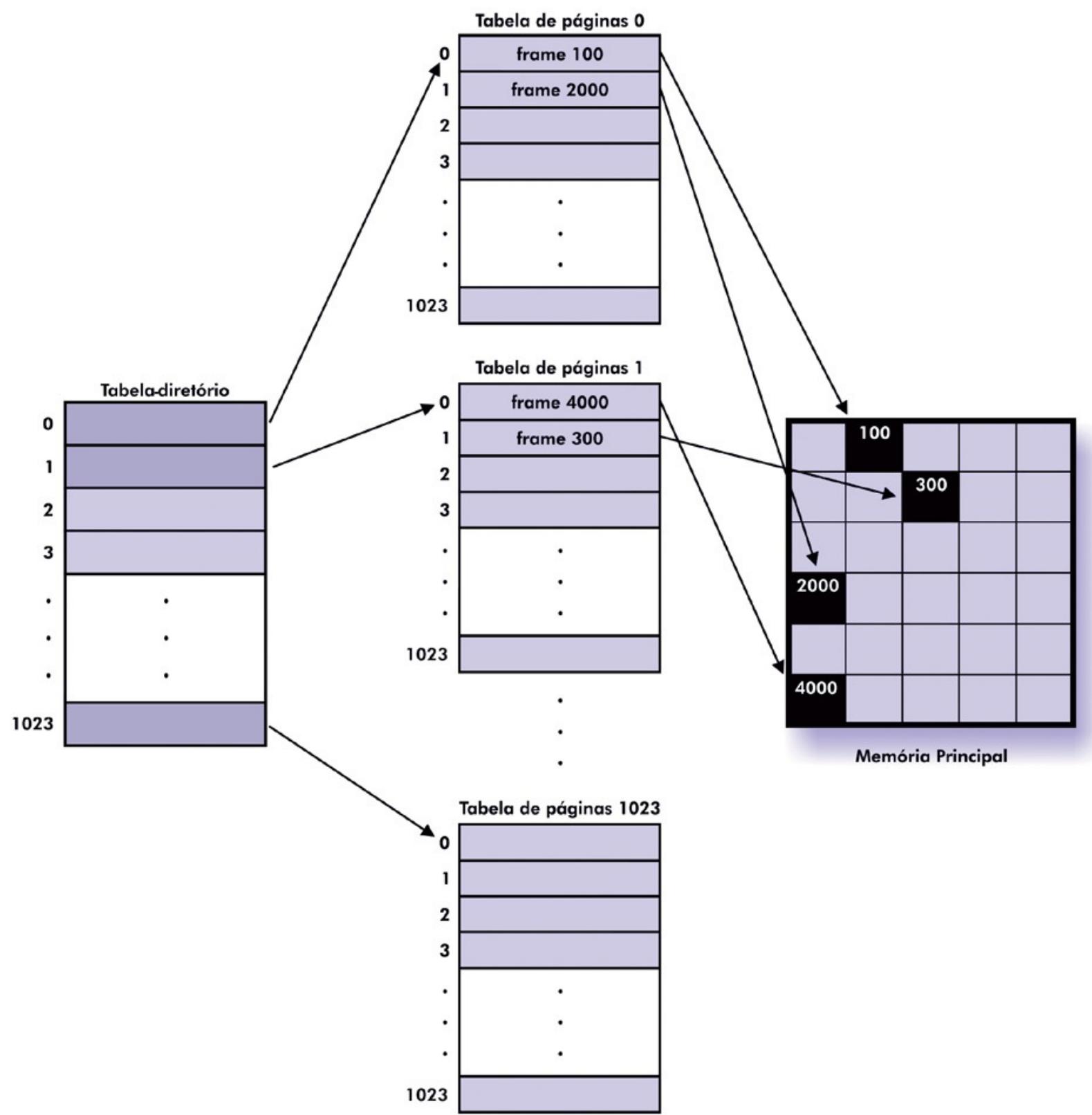


Fig. 10.20 Paginação em dois níveis.

10.4.8 Translation Lookaside Buffer

A gerência de memória virtual utiliza a técnica de mapeamento para traduzir endereços virtuais em endereços reais, porém o mapeamento implica pelo menos dois acessos à memória principal: o primeiro à tabela de páginas e o outro à própria página. Sempre que um endereço virtual precisa ser traduzido, a tabela de mapeamento deve ser consultada para se obter o endereço do frame e, posteriormente, acessar

o dado na memória principal.

Como a maioria das aplicações referencia um número reduzido de frames na memória principal, seguindo o princípio da localidade, somente uma pequena fração da tabela de mapeamento é realmente necessária. Com base neste princípio, foi introduzida uma memória especial chamada translation lookaside buffer (TLB), com o intuito de mapear endereços virtuais em endereços físicos sem a necessidade do acesso à tabela de páginas (Fig. 10.21).

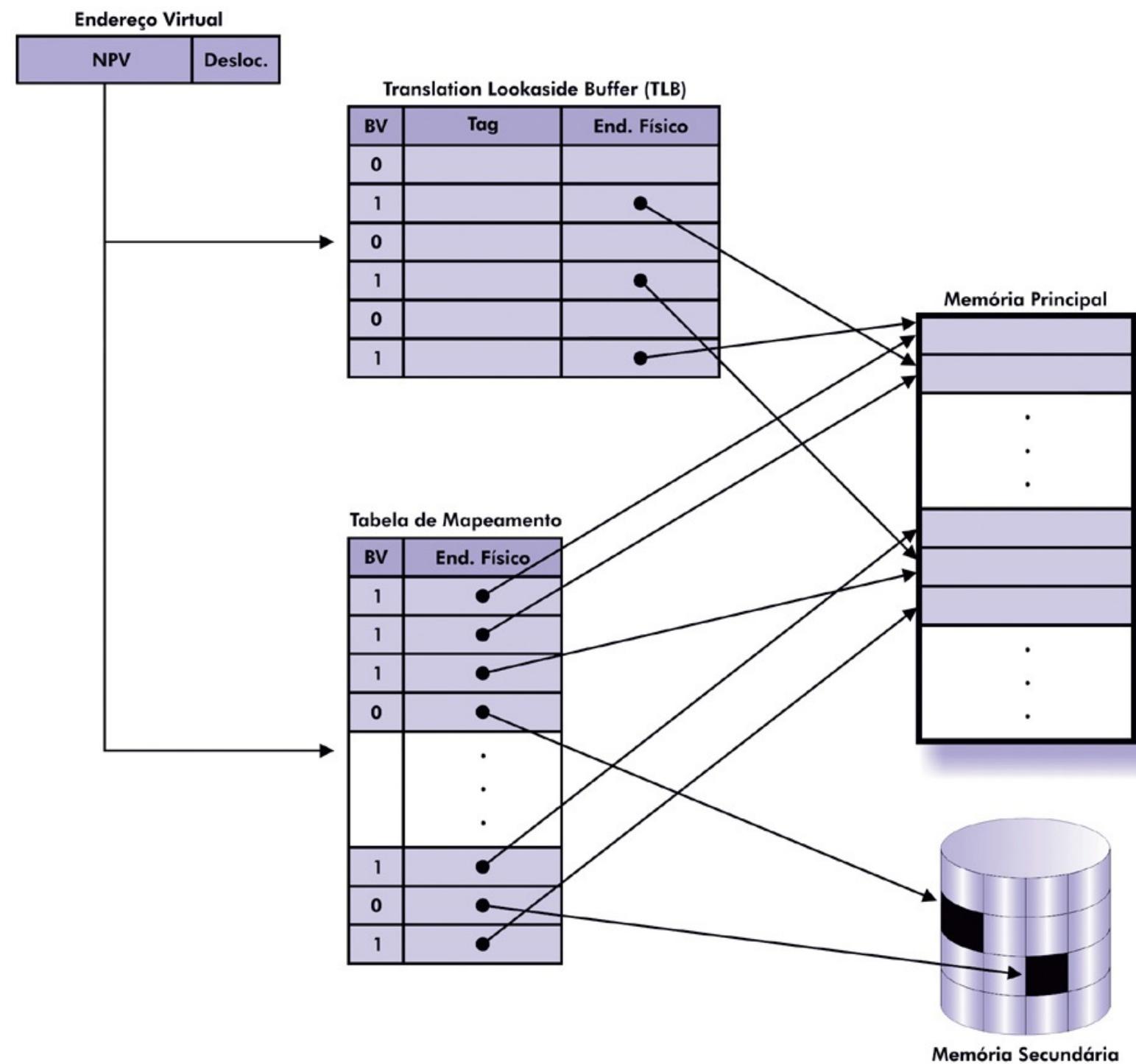


Fig. 10.21 Translation lookaside buffer (TLB).

O TLB funciona como uma memória cache, mantendo apenas as traduções dos endereços virtuais das páginas mais recentemente referenciadas. Em geral, o TLB utiliza o esquema de mapeamento associativo,

que permite verificar simultaneamente em todas as suas entradas a presença do endereço virtual. Dessa forma, para localizar uma entrada não é necessário realizar uma pesquisa em todo o TLB. Além disso, as traduções dos endereços virtuais podem ser armazenadas em qualquer posição da cache.

Na tradução de um endereço virtual, o sistema verifica primeiro o TLB. Caso o endereço virtual (tag) esteja na cache, o endereço físico é utilizado, eliminando o acesso à tabela de mapeamento (TLB hit). Caso o endereço não esteja na cache, a tabela de mapeamento deve ser consultada (TLB miss). Se a página estiver na memória principal, a tradução do endereço virtual é colocada no TLB e o endereço é traduzido. Caso contrário, ocorre um page fault, a página é carregada para a memória, a tabela de mapeamento é atualizada e a informação é carregada para a TLB.

Como a TLB pode eliminar o acesso à tabela de mapeamento, as informações de um endereço virtual contidas na entrada da tabela de páginas devem também estar na cache. A Tabela 10.3 descreve os campos de uma entrada típica de uma TLB.

Tabela 10.3 Campos da TLB

Campo	Descrição
Tag	Endereço virtual sem o deslocamento.
Modificação	Bit que indica se a página foi alterada.
Referência	Bit que indica se a página foi recentemente referenciada, sendo utilizada para a realocação de entradas na TLB.
Proteção	Define a permissão de acesso à página.
Endereço físico	Posição do frame na memória principal.

Geralmente, a TLB contém apenas informações sobre endereços do processo em execução, ou seja, quando há uma mudança de contexto a TLB deve ser reinicializada com informações da tabela de mapeamento do novo processo. Caso a TLB possa armazenar endereços de vários processos, deve existir um campo adicional em cada entrada da TLB para a identificação do processo.

Sempre que a TLB fica com todas as suas entradas ocupadas e um novo endereço precisa ser armazenado, uma entrada da cache deve ser liberada. Geralmente, a política de realocação de entradas da TLB é aleatória e, mais raramente, utiliza-se a política NRU.

A TLB é essencial para reduzir o número de operações de acesso à memória principal em sistemas que implementam memória virtual. Devido ao conceito de localidade, a TLB pode ser implementada com poucas entradas, mapeando de 8 a 2048 endereços. Mesmo pequena, a taxa de TLB hits é muito alta, reduzindo significativamente o impacto da gerência de memória virtual no desempenho do sistema.

10.4.9 Proteção de Memória

Em qualquer sistema multiprogramável, onde diversas aplicações compartilham a memória principal,

devem existir mecanismos para preservar as áreas de memória do sistema operacional e dos diversos processos dos usuários. O sistema operacional deve impedir que um processo tenha acesso ou modifique uma página do sistema sem autorização. Caso uma página do sistema operacional seja indevidamente alterada, é possível que, como consequência, haja uma instabilidade no funcionamento do sistema ou sua parada completa. Até mesmo páginas dos processos de usuários precisam ser protegidas contra alteração, como no caso de frames contendo código executável.

Um primeiro nível de proteção é inerente ao próprio mecanismo de memória virtual por paginação. Neste esquema, cada processo tem a sua própria tabela de mapeamento e a tradução dos endereços é realizada pelo sistema. Desta forma, não é possível a um processo acessar áreas de memória de outros processos, a menos que haja compartilhamento explícito de páginas entre processos. A proteção de acesso é realizada individualmente em cada página da memória principal, utilizando-se as entradas das tabelas de mapeamento, onde alguns bits especificam os acessos permitidos (Fig. 10.22).



Fig. 10.22 Proteção para páginas.

De forma resumida, definiremos dois tipos de acessos básicos realizados em uma página: leitura e gravação. O acesso de leitura permite a leitura da página, enquanto o de gravação a sua alteração. Os dois tipos de acessos combinados produzem um mecanismo de proteção simples e eficiente, permitindo desde o total acesso à página, passando por acessos intermediários, até a falta completa de acesso (Fig. 10.23).

LG	Descrição
00	Sem acesso
10	Acesso de leitura
11	Acesso para leitura/gravação

Fig. 10.23 Mecanismo de proteção.

Sempre que uma página é referenciada, o sistema operacional verifica na tabela de mapeamento do processo a proteção do frame e determina se a operação é permitida. Caso a tentativa de uma operação de gravação seja realizada em uma página com proteção apenas de leitura, o sistema gera um erro indicando a ocorrência do problema.

10.4.10 Compartilhamento de Memória

Em sistemas que implementam memória virtual, é bastante simples a implementação da reentrância,

possibilitando compartilhamento de código entre os diversos processos. Para isso, basta que as entradas das tabelas de mapeamento dos processos apontem para os mesmos frames na memória principal, evitando, assim, várias cópias de um mesmo programa na memória (Fig. 10.24). Apesar de os processos compartilharem as mesmas páginas de código, cada um possui sua própria área de dados em páginas independentes.

O compartilhamento de memória também é extremamente importante em aplicações que precisam compartilhar dados na memória principal. Similar ao compartilhamento de código, o mecanismo de paginação permite que processos façam o mapeamento de uma mesma área na memória e, consequentemente, tenham acesso compartilhado de leitura e gravação. A única preocupação da aplicação é garantir o sincronismo no acesso à região compartilhada, evitando problemas de inconsistência.

► 10.5 Memória Virtual por Segmentação

Memória virtual por segmentação é a técnica de gerência de memória onde o espaço de endereçamento virtual é dividido em blocos de tamanhos diferentes chamados segmentos. Na técnica de segmentação, um programa é dividido logicamente em sub-rotinas e estruturas de dados, que são alocadas em segmentos na memória principal (Fig. 10.25).

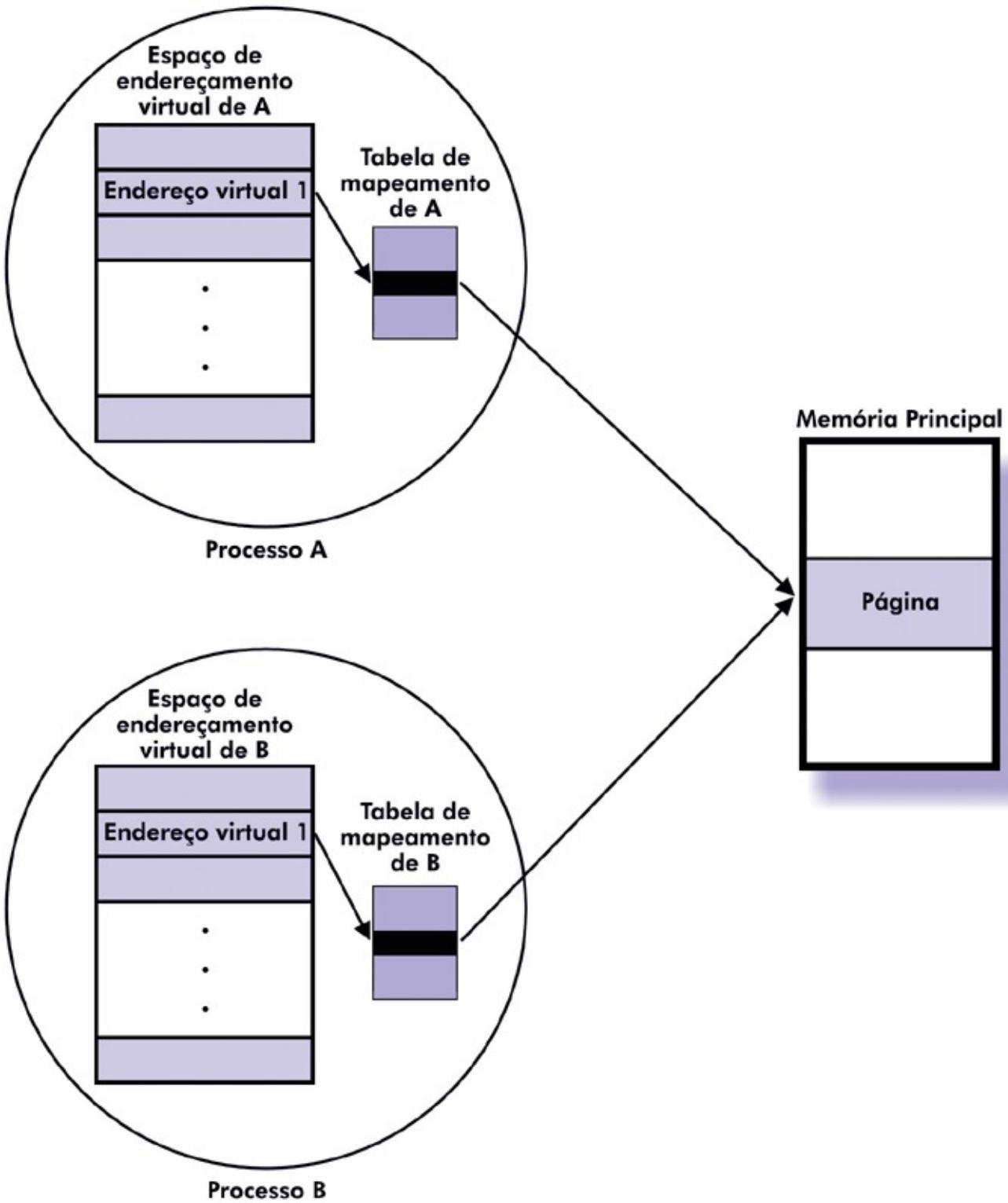


Fig. 10.24 Compartilhamento de memória.

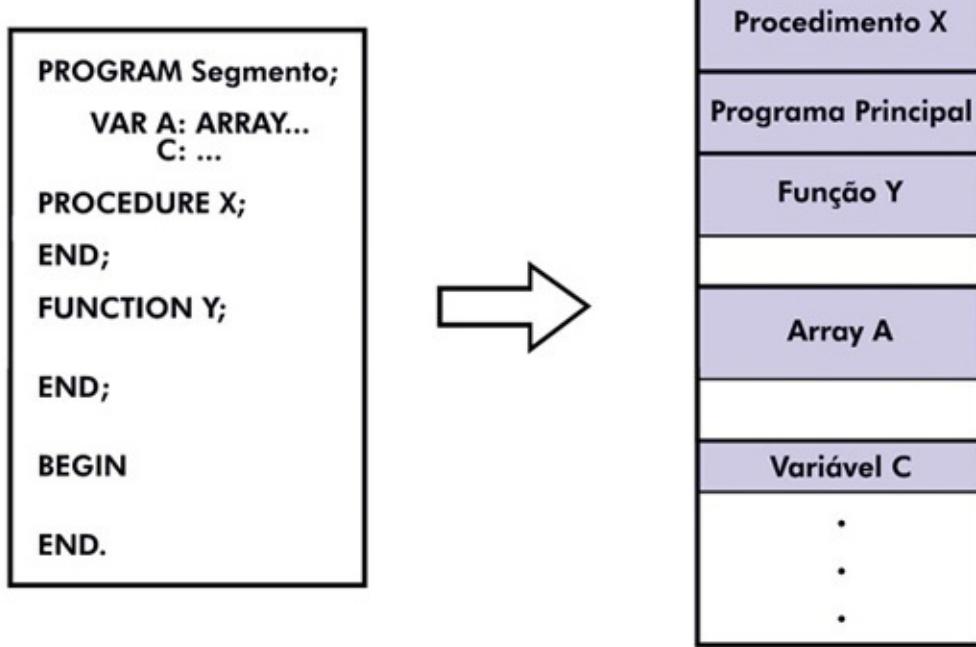


Fig. 10.25 Segmentação.

Enquanto na técnica de paginação o programa é dividido em páginas de tamanho fixo, sem qualquer ligação com sua estrutura, na segmentação existe uma relação entre a lógica do programa e sua alocação na memória principal. Normalmente, a definição dos segmentos é realizada pelo compilador, a partir do código-fonte do programa, e cada segmento pode representar um procedimento, função, vetor ou pilha.

O espaço de endereçamento virtual de um processo possui um número máximo de segmentos que podem existir, onde cada segmento pode variar de tamanho dentro de um limite. O tamanho do segmento pode ser alterado durante a execução do programa, facilitando a implementação de estruturas de dados dinâmicas. Espaços de endereçamento independentes permitem que uma sub-rotina seja alterada sem a necessidade de o programa principal e todas as suas sub-rotinas serem recompiladas e religadas. Em sistemas que implementam paginação, a alteração de uma sub-rotina do programa implica recompilar e religar a aplicação por completo.

O mecanismo de mapeamento é muito semelhante ao de paginação. Os segmentos são mapeados através de tabelas de mapeamento de segmentos (TMS), e os endereços são compostos pelo número do segmento virtual (NSV) e por um deslocamento. O NSV identifica unicamente o segmento virtual que contém o endereço, funcionando como um índice na TMS. O deslocamento indica a posição do endereço virtual em relação ao início do segmento no qual se encontra. O endereço físico é obtido, então, combinando-se o endereço do segmento, localizado na TMS, com o deslocamento, contido no endereço virtual (Fig. 10.26).

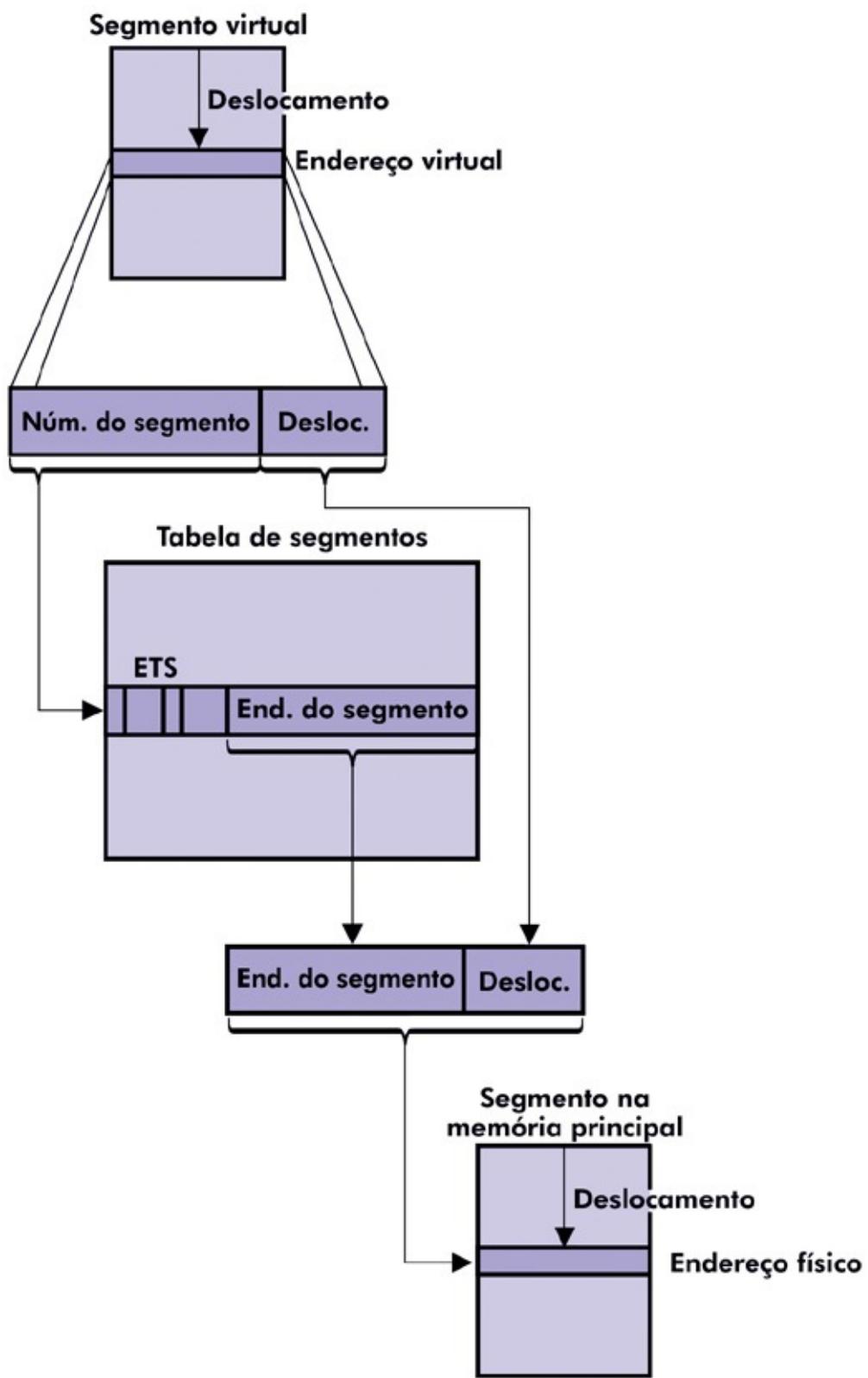


Fig. 10.26 Tradução do endereço virtual.

Cada ETS possui, além do endereço do segmento na memória principal, informações adicionais, como ilustra a Tabela 10.4.

Tabela 10.4 Campos da ETS

Campo	Descrição
Tamanho	Especifica o tamanho do segmento.

Bit de validade

Indica se o segmento está na memória principal.

Bit de modificação

Indica se o segmento foi alterado.

Bit de referência

Indica se o segmento foi recentemente referenciado, sendo utilizado pelo algoritmo de substituição.

Proteção

Indica a proteção do segmento.

Uma grande vantagem da segmentação em relação à paginação é a sua facilidade em lidar com estruturas de dados dinâmicas. Como o tamanho do segmento pode ser facilmente alterado na ETS, estruturas de dados, como pilhas e listas encadeadas, podem aumentar e diminuir dinamicamente, oferecendo grande flexibilidade ao desenvolvedor. Enquanto na paginação a expansão de um vetor implica a alocação de novas páginas e, consequentemente, o ajuste da tabela de paginação, na segmentação deve ser alterado apenas o tamanho do segmento.

Na técnica de segmentação, apenas os segmentos referenciados são transferidos da memória secundária para a memória principal. Se as aplicações não forem desenvolvidas em módulos, grandes segmentos estarão na memória desnecessariamente, reduzindo o compartilhamento da memória e o grau de multiprogramação. Logo, para que a segmentação funcione de forma eficiente, os programas devem estar bem modularizados.

Para alocar os segmentos na memória principal, o sistema operacional mantém uma tabela com as áreas livres e ocupadas da memória. Quando um novo segmento é referenciado, o sistema seleciona um espaço livre suficiente para que o segmento seja carregado na memória. A política de alocação de páginas pode ser a mesma utilizada na alocação particionada dinâmica (best-fit, worst-fit ou first-fit), apresentada no Cap. 9 — Gerência de Memória.

Enquanto na paginação existe o problema da fragmentação interna, na segmentação surge o problema da fragmentação externa. Este problema ocorre sempre que há diversas áreas livres na memória principal, mas nenhuma é grande o suficiente para alocar um novo segmento. Neste caso, é necessário que os segmentos sejam realocados na memória de forma que os espaços livres sejam agrupados em uma única área maior.

Em sistemas com segmentação, a proteção de memória é mais simples de ser implementada do que em sistemas com paginação. Como cada segmento possui um conteúdo bem definido, ou seja, instruções ou dados, basta especificar a proteção do segmento na ETS, onde alguns bits podem especificar os tipos de acesso ao segmento.

Na segmentação é mais simples o compartilhamento de memória do que na paginação, pois a tabela de segmentos mapeia estruturas lógicas e não páginas. Para compartilhar um segmento, basta que as ETS dos diversos processos apontem para o mesmo segmento na memória principal. Por exemplo, enquanto o mapeamento de um vetor pode necessitar de várias entradas na tabela de páginas, na tabela de segmentos é necessária apenas uma única entrada.

A Tabela 10.5 compara as técnicas de paginação e segmentação em função de suas principais características.

Tabela 10.5 Paginação x segmentação

Característica	Paginação	Segmentação
Tamanho dos blocos de memória	Iguais	Diferentes
Proteção	Complexa	Mais simples
Compartilhamento	Complexo	Mais simples
Estruturas de dados dinâmicas	Complexo	Mais simples
Fragmentação interna	Pode existir	Não existe
Fragmentação externa	Não existe	Pode existir
Programação modular	Dispensável	Indispensável
Alteração do programa	Mais trabalhosa	Mais simples

► 10.6 Memória Virtual por Segmentação com Paginação

Memória virtual por segmentação com paginação é a técnica de gerência de memória na qual o espaço de endereçamento é dividido em segmentos e, por sua vez, cada segmento dividido em páginas. Esse esquema de gerência de memória tem o objetivo de oferecer as vantagens tanto da técnica de paginação quanto da técnica de segmentação.

Nessa técnica, um endereço virtual é formado pelo número do segmento virtual (NSV), um número de página virtual (NPV) e um deslocamento. Através do NSV, obtém-se uma entrada na tabela de segmentos, que contém informações da tabela de páginas do segmento. O NPV identifica unicamente a página virtual que contém o endereço, funcionando como um índice na tabela de páginas. O deslocamento indica a posição do endereço virtual em relação ao início da página na qual se encontra. O endereço físico é obtido, então, combinando-se o endereço do frame, localizado na tabela de páginas, com o deslocamento, contido no endereço virtual (Fig. 10.27).

Na visão do programador, sua aplicação continua sendo mapeada em segmentos de tamanhos diferentes, em função das sub-rotinas e estruturas de dados definidas no programa. Por outro lado, o sistema trata cada segmento como um conjunto de páginas de mesmo tamanho, mapeadas por uma tabela de páginas associada ao segmento. Dessa forma, um segmento não precisa estar contíguo na memória principal, eliminando o problema da fragmentação externa encontrado na segmentação pura.

► 10.7 Swapping em Memória Virtual

A técnica de swapping, apresentada no Cap. 9 — Gerência de Memória, também pode ser aplicada em sistemas com memória virtual, permitindo aumentar o número de processos que compartilham a memória principal e, consequentemente, o grau de multiprogramação do sistema.

Quando existem novos processos para serem executados e não há memória principal livre suficiente para alocação, o sistema utiliza o swapping, selecionando um ou mais processos para saírem da memória e oferecer espaço para novos processos. Depois de escolhidos, o sistema retira os processos da memória principal para a memória secundária (*swap out*), onde as páginas ou segmentos são gravados em um arquivo de swap (*swap file*). Com os processos salvos na memória secundária, os frames ou segmentos alocados são liberados para novos processos. Posteriormente, os processos que foram retirados da memória devem retornar para a memória principal (*swap in*) para serem novamente executados (Fig. 10.28).

Há várias políticas que podem ser aplicadas na escolha dos processos que devem ser retirados da memória principal. Independente do algoritmo utilizado, o sistema tenta selecionar os processos com as menores chances de serem executados em um futuro próximo. Na maioria das políticas, o critério de escolha considera o estado do processo e sua prioridade.

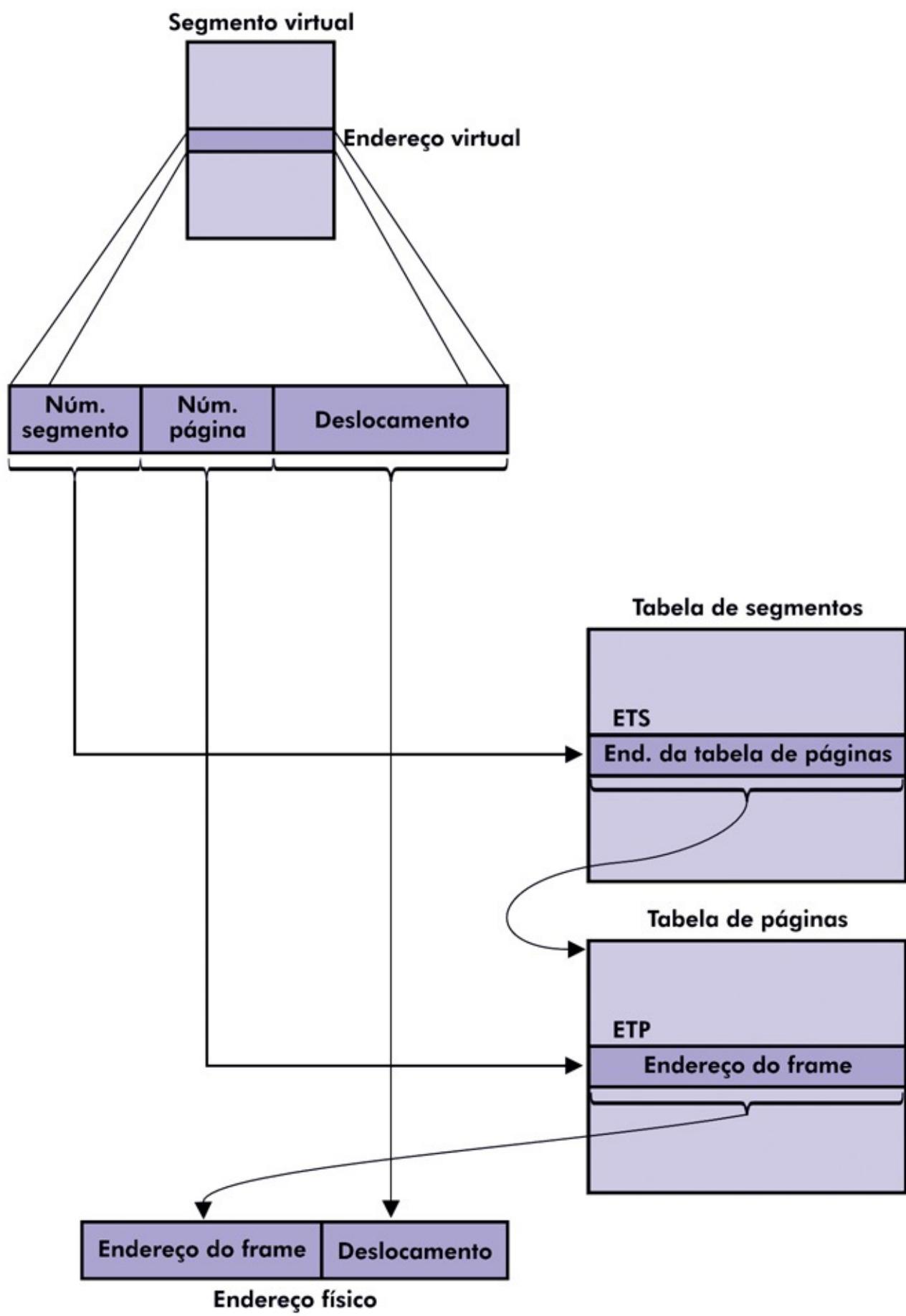


Fig. 10.27 Segmentação com paginação.

O swapping com base no estado dos processos seleciona, inicialmente, os processos que estão no estado de espera. A seleção pode ser refinada em função do tipo de espera de cada processo. É possível que não existam processos suficientes no estado de espera para atender as necessidades de memória do sistema.

Nesse caso, os processos no estado de pronto com menor prioridade deverão ser selecionados.

O arquivo de swap é compartilhado por todos os processos que estão sendo executados no ambiente. Quando um processo é criado, o sistema reserva um espaço no arquivo de swap para o processo. Da mesma forma, quando um processo é eliminado o sistema libera a área alocada. Alguns sistemas operacionais utilizam um único arquivo para uso como arquivo de paginação e de swap.

Em alguns sistemas operacionais, o arquivo de swap é, na verdade, uma área em disco reservada exclusivamente para esta função. Independentemente da implementação, o arquivo de swap deve oferecer o melhor desempenho possível para as operações de swapping.

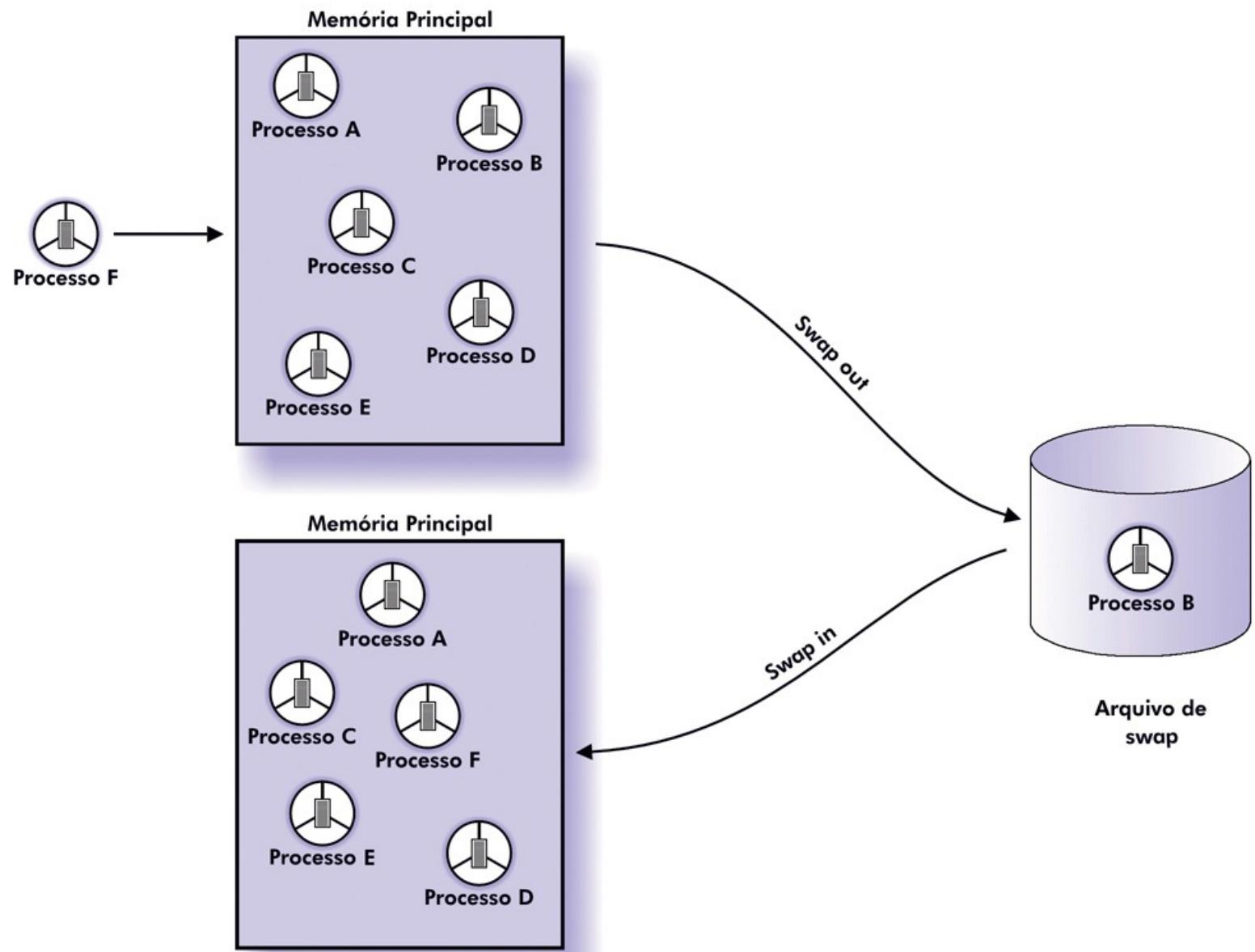


Fig. 10.28 Swapping em memória virtual.

► 10.8 Thrashing

Thrashing pode ser definido como a excessiva transferência de páginas/segmentos entre a memória

principal e a memória secundária. Esse problema está presente em sistemas que implementam tanto paginação como segmentação.

Na memória virtual por paginação, o thrashing ocorre em dois níveis: no do próprio processo e no do sistema. No nível do processo, a excessiva paginação ocorre devido ao elevado número de page faults gerado pelo programa em execução. Esse problema faz com que o processo passe mais tempo esperando por páginas que realmente正在 sendo executado. Existem dois motivos que levam um processo a sofrer esse tipo de thrashing. O primeiro é o dimensionamento incorreto do limite máximo de páginas do processo, pequeno demais para acomodar seu working set. O segundo é a ausência do princípio da localidade.

O thrashing no sistema ocorre quando existem mais processos competindo por memória principal que espaço disponível. Nesse caso, o primeiro passo é a redução do número de páginas de cada processo na memória; porém, como já analisamos, esse mecanismo leva ao thrashing do processo. Caso a redução não seja suficiente, o sistema inicia o swapping, retirando processos da memória principal para a memória secundária. Se esse mecanismo for levado ao extremo, o sistema passará mais tempo realizando swapping que atendendo aos processos.

O thrashing em sistemas que implementam segmentação também ocorre em dois níveis. No nível do processo, a transferência excessiva de segmentos é devida à modularização extrema do programa. O thrashing no sistema é semelhante ao da paginação, com a ocorrência de swapping de processos para liberar memória para os demais.

Independentemente das soluções apresentadas, se existirem mais processos para serem executados que memória real disponível a única solução é a expansão da memória principal. É importante ressaltar que este problema não ocorre apenas em sistemas que implementam memória virtual, mas também em sistemas com outros mecanismos de gerência de memória.

► 10.9 Exercícios

1. Quais os benefícios oferecidos pela técnica de memória virtual? Como este conceito permite que um programa e seus dados ultrapassem os limites da memória principal?
2. Explique como um endereço virtual de um processo é traduzido para um endereço real na memória principal?
3. Por que o mapeamento deve ser feito em blocos e não sobre células individuais? Apresente um exemplo numérico.
4. Qual a principal diferença entre os sistemas que implementam paginação e os que implementam segmentação?
5. Diferencie página virtual de página real.
6. O que são tabelas de páginas e tabelas de segmentos?
7. Para que serve o bit de validade nas tabelas de páginas e segmentos?
8. O que é um page fault, quando ocorre e quem controla a sua ocorrência? Como uma elevada taxa de page fault pode comprometer o sistema operacional?

Nos sistemas com paginação, a rotina para tratamento de page faults está residente na memória principal. Esta rotina pode ser removida da memória em algum momento? O que aconteceria se esta rotina não estivesse na memória principal durante a ocorrência de um page fault?

9. Descreva como ocorre a fragmentação interna em um sistema que implementa paginação.

10. Compare as políticas de busca de páginas apresentadas.

11. Quais as vantagens e desvantagens da política de alocação de páginas variável comparada à alocação fixa?

12. Um sistema com gerência de memória virtual por paginação possui tamanho de página com 512 posições, espaço de endereçamento virtual com 512 páginas endereçadas de 0 a 511 e memória real com 10 páginas numeradas de 0 a 9. O conteúdo atual da memória real contém apenas informações de um único processo e é descrito resumidamente na seguinte tabela:

Endereço Físico	Conteúdo
1536	Página Virtual 34
2048	Página Virtual 9
3072	Tabela de Páginas
3584	Página Virtual 65
4608	Página Virtual 10

- a) Considere que a entrada da tabela de páginas contém, além do endereço do frame, o número da página virtual. Mostre o conteúdo da tabela de páginas deste processo.
- b) Mostre o conteúdo da tabela de páginas após a página virtual 49 ser carregada na memória a partir do endereço real 0 e a página virtual 34 ser substituída pela página virtual 12.
- c) Como é o formato do endereço virtual deste sistema?
- d) Qual endereço físico está associado ao endereço virtual 4613?

13. Um sistema operacional implementa gerência de memória virtual por paginação, com frames de 2 Kb. A partir da tabela que se segue, que representa o mapeamento de páginas de um processo em um determinado instante de tempo, responda:

Página	Residente	Frame
0	Sim	20
1	Sim	40
2	Sim	100
3	Sim	10
4	Não	50
5	Não	70

- a) Qual o endereço físico de uma variável que ocupa o último byte da página 3?
 b) Qual o endereço físico de uma variável que ocupa o primeiro byte da página 2?
 c) Qual o endereço físico de uma variável que tem deslocamento 10 na página 3?
 d) Quais páginas do processo estão na memória?

15. Um sistema operacional implementa gerência de memória virtual por paginação. Considere endereços virtuais com 16 bits, referenciados por um mesmo processo durante sua execução, e sua tabela de páginas a seguir com no máximo 256 entradas. Estão representadas apenas as páginas presentes na memória real. Indique para cada endereço virtual a seguir a página virtual em que o endereço se encontra, o respectivo deslocamento e se a página se encontra na memória principal neste momento.

- a) $(307)_{10}$
 b) $(2049)_{10}$
 c) $(2304)_{10}$

Página	Endereço Físico
0	8 K
1	4 K
2	24 K
3	0 K
4	16 K
5	12 K
9	20 K
11	28 K

16. Uma memória virtual possui páginas de 1024 endereços, existem oito páginas virtuais e 4096 bytes de memória real. A tabela de páginas de um processo está descrita a seguir. O asterisco indica que a página não está na memória principal:

Página Virtual	Página Real
0	3
1	1
2	*
3	*
4	2

- a) Faça a lista/faixa de todos os endereços virtuais que irão causar page fault.
- b) Indique o endereço real correspondente aos seguintes endereços virtuais 0, 1023, 1024, 6500 e 3728.
17. Por que existe a necessidade de uma política de substituição de páginas? Compare as políticas de substituição local e global.
18. Para que serve o bit de modificação nas tabelas de páginas e segmentos?
19. Como o princípio da localidade viabiliza a implementação da gerência de memória virtual por paginação?
20. Por que programas não estruturados estão sujeitos a uma alta taxa de paginação?
21. Descreva os algoritmos de substituição de páginas FIFO e LRU, apresentando vantagens e desvantagens.
22. Considere um sistema com memória virtual por paginação com endereço virtual com 24 bits e página com 2048 endereços. Na tabela de páginas a seguir, de um processo em determinado instante de tempo, o bit de validade 1 indica página na memória principal e o bit de modificação 1 indica que a página sofreu alteração.

Página	BV	BM	End. do Frame
0	1	1	30.720
1	1	0	0
2	1	1	10.240
3	0	1	*
4	0	0	*
5	1	0	6.144

- a) Quantos bits possui o campo deslocamento do endereço virtual?
- b) Qual o número máximo de entradas que a tabela de páginas pode ter?
- c) Qual o endereço físico que ocupa o último endereço da página 2?
- d) Qual o endereço físico traduzido do endereço virtual (00080A)16?
- e) Caso ocorra um page fault e uma das páginas do processo deva ser descartada, quais páginas poderiam sofrer page out?
23. Considere um sistema de memória virtual que implemente paginação no qual o limite de frames por processo é igual a três. Descreva para os itens a seguir, para os quais é apresentada uma sequência de referências a páginas pelo processo, o número total de page faults para as estratégias de realocação

de páginas FIFO e LRU. Indique qual a mais eficaz para cada item.

a) 1 / 2 / 3 / 1 / 4 / 2 / 5 / 3 / 4 / 3

b) 1 / 2 / 3 / 1 / 4 / 1 / 3 / 2 / 3 / 3

24. Em um sistema de memória virtual que implementa paginação, as páginas têm 4 K endereços, a memória principal possui 32 Kb e o limite de páginas na memória principal é de oito páginas. Um programa faz referência a endereços virtuais situados nas páginas 0, 2, 1, 9, 11, 4, 5, 2, 3, 1, nesta ordem. Após essa sequência de acessos, a tabela de páginas completa desse programa tem a configuração que se segue. As entradas em branco correspondem a páginas ausentes.

Página	End. Físico
0	8 K
1	4 K
2	24 K
3	0 K
4	16 K
5	12 K
6	*
7	*
8	*
9	20 K
10	*
11	28 K
12	*
13	*
14	*
15	*

a) Qual o tamanho (em bits) e o formato do endereço virtual?

b) O processo faz novas referências a endereços virtuais situados nas páginas 5, 15, 12, 8 e 0, nesta ordem. Complete o quadro a seguir, que ilustra o processamento dessa sequência de acessos utilizando a estratégia de remoção FIFO. Mostre o estado final da tabela de páginas.

Página Referenciada

Página Removida

Page Fault (sim/não)

15

12

8

0

25. Em um computador, o endereço virtual é de 16 bits e as páginas têm tamanho de 2 K endereços. O limite de páginas reais de um processo qualquer é de quatro páginas. Inicialmente, nenhuma página está na memória principal. Um programa faz referência a endereços virtuais situados nas páginas 0, 7, 2, 7, 5, 8, 9, 2 e 4, nesta ordem.

- Quantos bits do endereço virtual destinam-se ao número da página? E ao deslocamento?
- Ilustre o comportamento da política de substituição LRU mostrando, a cada referência, quais páginas estão em memória, os page faults causados e as páginas escolhidas para descarte.

26. Um sistema trabalha com gerência de memória virtual por paginação. Para todos os processos do sistema, o limite de páginas na memória principal é igual a 10. Considere um processo que esteja executando um programa e em um determinado instante de tempo (T) a sua tabela de páginas possui o conteúdo a seguir. O bit de validade igual a 1 indica página na memória principal e o bit de modificação igual a 1 indica que a página sofreu alteração.

Número da Página	BV	BM	Endereço do Frame (hexadecimal)
0	1	0	3303A5
1	1	0	AA3200
2	1	0	111111
3	1	1	BFDCCA
4	1	0	765BFC
5	1	0	654546
6	1	1	B6B7B0
7	1	1	999950
8	1	0	888BB8
9	0	0	N/A
10	0	0	N/A

Responda às perguntas que se seguem, considerando que os seguintes eventos ocorrerão nos instantes de tempo indicados:

- (T + 1) O processo referencia um endereço na página 9 com page fault.
- (T + 2) O processo referencia um endereço na página 1.

(T + 3) O processo referencia um endereço na página 10 com page fault.

(T + 4) O processo referencia um endereço da página 3 com page fault.

(T + 5) O processo referencia um endereço da página 6 com page fault.

- a) Em quais instantes de tempo ocorre um page out?
- b) Em que instante de tempo o limite de páginas do processo na memória principal é atingido?
- c) Caso a política de realocação de páginas utilizada seja FIFO, no instante (T + 1), qual a página que está há mais tempo na memória principal?
- d) Como o sistema identifica que no instante de tempo (T + 2) não há ocorrência de page fault?

27. Um sistema possui quatro frames. A tabela seguinte apresenta para cada página o momento da carga, o momento do último acesso, o bit de referência e o bit de modificação.

Frame	Carga	Referência	BR	BM
0	126	279	0	0
1	230	260	1	0
2	120	272	1	1
3	160	280	1	1

- a) Qual página será substituída utilizando o algoritmo NRU?
- b) Qual página será substituída utilizando o algoritmo FIFO?
- c) Qual página será substituída utilizando o algoritmo LRU?

28. Considere um processo com limite de páginas reais igual a quatro e um sistema que implemente a política de substituição de páginas FIFO. Indique quantos page faults ocorrerão, considerando que as páginas virtuais são referenciadas na seguinte ordem: 0172327103. Repita o problema utilizando a política LRU.

29. Os sistemas operacionais OpenVMS e MS Windows utilizam dois buffers de páginas: um buffer de páginas livres e outro para páginas modificadas. Qual a vantagem de implementar um buffer de páginas modificadas?

30. Explique por que páginas pequenas podem aumentar a taxa de paginação.

31. A arquitetura VAX-11 utiliza 32 bits para endereçamento e páginas de 512 bytes. Calcule o número de bits para cada parte do endereço virtual, sabendo que o espaço de endereçamento é dividido em quatro partes: P0, P1, S0 e S1, sendo que cada uma possui sua própria tabela de páginas.

32. Um sistema computacional com espaço de endereçamento de 32 bits utiliza uma tabela de páginas de dois níveis. Os endereços virtuais são divididos em um campo de 9 bits para o primeiro nível da tabela, outro de 11 bits para o segundo nível e um último campo para o deslocamento. Qual o tamanho das páginas? Quantas páginas podem existir no espaço de endereçamento virtual?

33. Na arquitetura SPARC, o espaço de endereçamento virtual de 4 G pode ser dividido para cada processo em páginas de 4 Kb. A busca do endereço real correspondente ao endereço virtual gerado pelo processador envolve, em caso de falha na TLB, três níveis de acesso à memória principal. No primeiro nível, é feito um acesso a uma tabela única por processo de 256 entradas. Essa tabela gera o

endereço de uma das 256 possíveis tabelas de nível 2. Cada tabela de nível 2 possui 64 entradas e, quando acessada, gera o endereço da tabela de nível 3 que deve ser consultada. Essa tabela, que também possui 64 entradas, gera o endereço real procurado. As tabelas de níveis 1, 2 e 3 formam basicamente uma árvore de busca na memória e vão sendo criadas dinamicamente, à medida que novas páginas na memória vão sendo alocadas para aquele processo. Qual a vantagem de se ter esse esquema de tabelas em múltiplos níveis, criadas dinamicamente sob demanda, em vez de uma tabela única criada integralmente quando da carga do processo? Justifique sua resposta com um exemplo.

34. Em um sistema computacional, a busca do endereço real correspondente ao endereço virtual gerado pelo processador envolve, em caso de falha na TLB, dois níveis de acesso à memória principal. Supondo que não existe memória cache, que a taxa de falha da TLB é de 2%, que o tempo de acesso à TLB é desprezível e que o tempo de acesso à memória principal é de 100 ns, calcule o tempo médio em acesso à memória gasto no processamento completo de uma instrução de soma de dois operandos. Considere que o primeiro operando é endereçado na memória em modo direto, o segundo operando é endereçado na memória em modo indireto e o operando destino é um registrador interno do processador.
35. Descreva o mecanismo de tradução de um endereço virtual em um endereço real em sistemas que implementam gerência de memória virtual utilizando segmentação com paginação.
36. Na técnica de swapping, que critérios o sistema operacional pode utilizar para selecionar os processos que sofrerão swap out?
37. Existe fragmentação em sistemas que implementam gerência de memória virtual? Se existe, que tipo de fragmentação é encontrado em sistemas com paginação? Que tipo de fragmentação é encontrado em sistemas com segmentação?
38. O que é o thrashing em sistemas que implementam memória virtual?

► 10.10 Laboratório com o Simulador SOsim

Atividade 1: Política de Busca — Paginação Antecipada

a) Práticas de simulação

- Execute o simulador SOsim e configure-o para trabalhar com Escalonamento Circular: janela *Console SOsim / Opções / Parâmetros do Sistema* na guia *Processador*.
- Configure a política de busca de páginas antecipada: janela *Console SOsim / Opções / Parâmetros do Sistema* na guia *Memória*.
- Reinicie o simulador SOsim para que a nova parametrização passe a ser válida.

b) Análise prática

- Crie um processo CPU-bound: janela *Gerência de Processos / Criar* – janela *Criação de Processos / Criar*.
- Ative a janela *Contexto do Processo* para visualizar a tabela de páginas do processo criado: *Gerência de Processos / PCB* na guia *Tab. de Pag.*

- Verifique os valores do Bit de Validade (Bit V) nas Entradas das Tabelas de Páginas (ETP).

Atividade 2: Política de Busca — Paginação sob Demanda

a) Práticas de simulação

- Execute o simulador SOsim e configure-o para trabalhar com Escalonamento Circular: janela *Console SOsim / Opções / Parâmetros do Sistema* na guia *Processador*.
- Configure a política de busca de páginas sob demanda: janela *Console SOsim / Opções / Parâmetros do Sistema* na guia *Memória*.
- Reinicie o simulador SOsim para que a nova parametrização passe a ser válida.

b) Análise prática

- Crie um processo CPU-bound: janela *Gerência de Processos / Criar* – janela *Criação de Processos / Criar*.
- Ative a janela *Contexto do Processo* para visualizar a tabela de páginas do processo criado: *Gerência de Processos / PCB* na guia *Tab. de Pag.*.
- Verifique os valores do Bit de Validade (Bit V) nas Entradas das Tabelas de Páginas (ETP) e o local em que se encontram as páginas.

c) Questão teórica para responder com ajuda do simulador

- Considerando as atividades práticas 1 e 2, quais as diferenças encontradas nas ETPs do processo criado? Justifique o motivo.

Atividade 3: Espaço de Endereçamento Virtual

a) Práticas de simulação

- Execute o simulador SOsim e configure-o para trabalhar com Escalonamento Circular: janela *Console SOsim / Opções / Parâmetros do Sistema* na guia *Processador*.
- Configure a política de busca de páginas sob demanda: janela *Console SOsim / Opções / Parâmetros do Sistema* na guia *Memória*.
- Reinicie o simulador SOsim para que a nova parametrização passe a ser válida.

b) Análise prática

- Crie dois processos CPU-bound: janela *Gerência de Processos / Criar* – janela *Criação de Processos / Criar*.
- Ative a janela *Contexto do Processo* para visualizar a tabela de páginas do processo criado: *Gerência de Processos / PCB* na guia *Tab. de Pag.*.
- Na janela *Gerência de Memória* observe a alocação dos frames na memória principal.
- Na janela *Contexto do Processo* observe as alterações nas tabelas de páginas dos dois processos

navegando com as setas inferiores.

c) Questões teóricas para responder com a ajuda do simulador

- Qual o espaço de endereçamento real máximo de um processo?
- Qual o espaço de endereçamento real mínimo de um processo?
- Qual o tamanho da página virtual?

Atividade 4: Limite de Frames e FIFO com Buffer de Páginas

a) Práticas de simulação

- Execute o simulador SOsim e configure-o para trabalhar com Escalonamento Circular: janela *Console SOsim / Opções / Parâmetros do Sistema* na guia *Processador*.
- Configure a política de busca de páginas sob demanda: janela *Console SOsim / Opções / Parâmetros do Sistema* na guia *Memória*.
- Reinicie o simulador SOsim para que a nova parametrização passe a ser válida.

b) Análise prática

- Crie um processo CPU-bound com limite de três frames: janela *Gerência de Processos / Criar*.
- Ative a janela *Contexto do Processo* para visualizar a tabela de páginas do processo criado: *Gerência de Processos / PCB* na guia *Tab. de Pag.*
- Ative a janela *Arquivo de Paginação* para visualizar o arquivo de paginação do sistema: *Console SOsim / Janelas / Arquivo de Paginação*.
- Observe na janela *Gerência de Memória* a alocação dos frames na memória principal, e na janela *Contexto do Processo*, a tabela de páginas do processo.

c) Questão teórica para responder com a ajuda do simulador

- O que acontece quando a página virtual 3 (quarta página) é referenciada?
- E a página virtual 4?
- O que acontece quando a página virtual 0 é novamente referenciada?
- Observe se ao longo da execução do programa ocorre page out para o arquivo de paginação. Justifique.

Atividade 5: Swapping

a) Práticas de simulação

- Execute o simulador SOsim e configure-o para trabalhar com Escalonamento Circular: janela *Console SOsim / Opções / Parâmetros do Sistema* na guia *Processador*.
- Configure a política de busca de páginas sob demanda: janela *Console SOsim / Opções / Parâmetros do Sistema* na guia *Memória*.

- Configurar a memória livre para possuir sempre 20% de frames livres: janela *Console SOsim / Opções / Parâmetros do Sistema* na guia *Memória*.
- Reinicie o simulador SOsim para que a nova parametrização passe a ser válida.

b) Análise prática

- Crie dois processos CPU-bound e três I/O-bound com limite de cinco frames para cada processo: janela *Gerência de Processos / Criar*.
- Suspenda um dos processos I/O-bound: janela *Gerência de Processos / Suspender*.
- Ative a janela *Arquivo de Paginação* para visualizar o arquivo de paginação do sistema: *Console SOsim / Janelas / Arquivo de Paginação*.
- Crie mais dois processos CPU-bound: janela *Gerência de Processos / Criar*.
- Observe os estados dos processos outswapped.

c) Questão teórica para responder com a ajuda do simulador

- Quais os critérios utilizados pelo simulador para selecionar o processo a ser transferido para o arquivo de paginação (swap out)?
- Quando o processo deve ser transferido novamente para a memória principal (swap in)?



Sistema de Arquivos

► 11.1 Introdução

O armazenamento e a recuperação de informações são atividades essenciais para qualquer tipo de aplicação. Um processo deve ser capaz de ler e gravar de forma permanente grande volume de dados em dispositivos como fitas e discos, além de poder compartilhá-los com outros processos. A maneira pela qual o sistema operacional estrutura e organiza estas informações é por intermédio da implementação de arquivos.

Os arquivos são gerenciados pelo sistema operacional de maneira a facilitar o acesso dos usuários ao seu conteúdo. A parte do sistema responsável por essa gerência é denominada *sistema de arquivos*. O sistema de arquivos é a parte mais visível de um sistema operacional, pois a manipulação de arquivos é uma atividade frequentemente realizada pelos usuários, devendo sempre ocorrer de maneira uniforme, independente dos diferentes dispositivos de armazenamento.

Neste capítulo serão apresentados aspectos presentes nos sistemas de arquivos, como identificação, organização, compartilhamento, métodos de acesso, proteção e operações de entrada e de saída.

► 11.2 Arquivos

Um *arquivo* é constituído por informações logicamente relacionadas. Estas informações podem representar instruções ou dados. Um arquivo executável, por exemplo, contém instruções compreendidas pelo processador, enquanto um arquivo de dados pode ser estruturado livremente como um arquivo-texto ou de forma mais rígida, como em um banco de dados relacional. Na realidade, um arquivo é um conjunto de registros definidos pelo sistema de arquivos, tornando seu conceito abstrato e generalista. A partir dessa definição, o conteúdo do arquivo pode ser manipulado segundo conceitos preestabelecidos.

Os arquivos são armazenados pelo sistema operacional em diferentes dispositivos físicos, como fitas magnéticas, discos magnéticos e discos ópticos. O tipo de dispositivo no qual o arquivo é armazenado

deve ser isolado pelo sistema operacional, de forma que exista uma independência entre os arquivos a serem manipulados e o meio de armazenamento.

Um arquivo é identificado por um nome, composto por uma sequência de caracteres. Em alguns sistemas de arquivos é feita distinção entre caracteres alfabéticos maiúsculos e minúsculos. Regras como extensão máxima do nome e quais são os caracteres válidos também podem variar.

Em alguns sistemas operacionais, a identificação de um arquivo é composta por duas partes separadas com um ponto. A parte após o ponto é denominada extensão do arquivo e tem como finalidade identificar o conteúdo do arquivo. Assim é possível convencionar que uma extensão TXT identifica um arquivo-texto, enquanto EXE indica um arquivo executável. Na Tabela 11.1 são apresentadas algumas extensões de arquivos.

Tabela 11.1 Extensão de arquivos

Extensão	Descrição
ARQUIVO.BAS	Arquivo-fonte em BASIC
ARQUIVO.COB	Arquivo-fonte em COBOL
ARQUIVO.EXE	Arquivo executável
ARQUIVO.OBJ	Arquivo-objeto
ARQUIVO.PAS	Arquivo-fonte em Pascal
ARQUIVO.TXT	Arquivo-texto

11.2.1 Organização de Arquivos

A *organização de arquivos* consiste em como os seus dados estão internamente armazenados. A estrutura dos dados pode variar em função do tipo de informação contida no arquivo. Arquivos-textos possuem propósitos completamente distintos de arquivos executáveis, consequentemente estruturas diferentes podem adequar-se melhor a um tipo do que a outro.

No momento da criação de um arquivo, seu criador pode definir qual a organização adotada. Esta organização pode ser uma estrutura suportada pelo sistema operacional ou definida pela própria aplicação.

A forma mais simples de organização de arquivos é através de uma sequência não estruturada de bytes (Fig. 11.1a). Neste tipo de organização, o sistema de arquivos não impõe nenhuma estrutura lógica para os dados. A aplicação deve definir toda a organização, estando livre para estabelecer seus próprios critérios. A grande vantagem deste modelo é a flexibilidade para criar diferentes estruturas de dados, porém todo o controle de acesso ao arquivo é de inteira responsabilidade da aplicação.

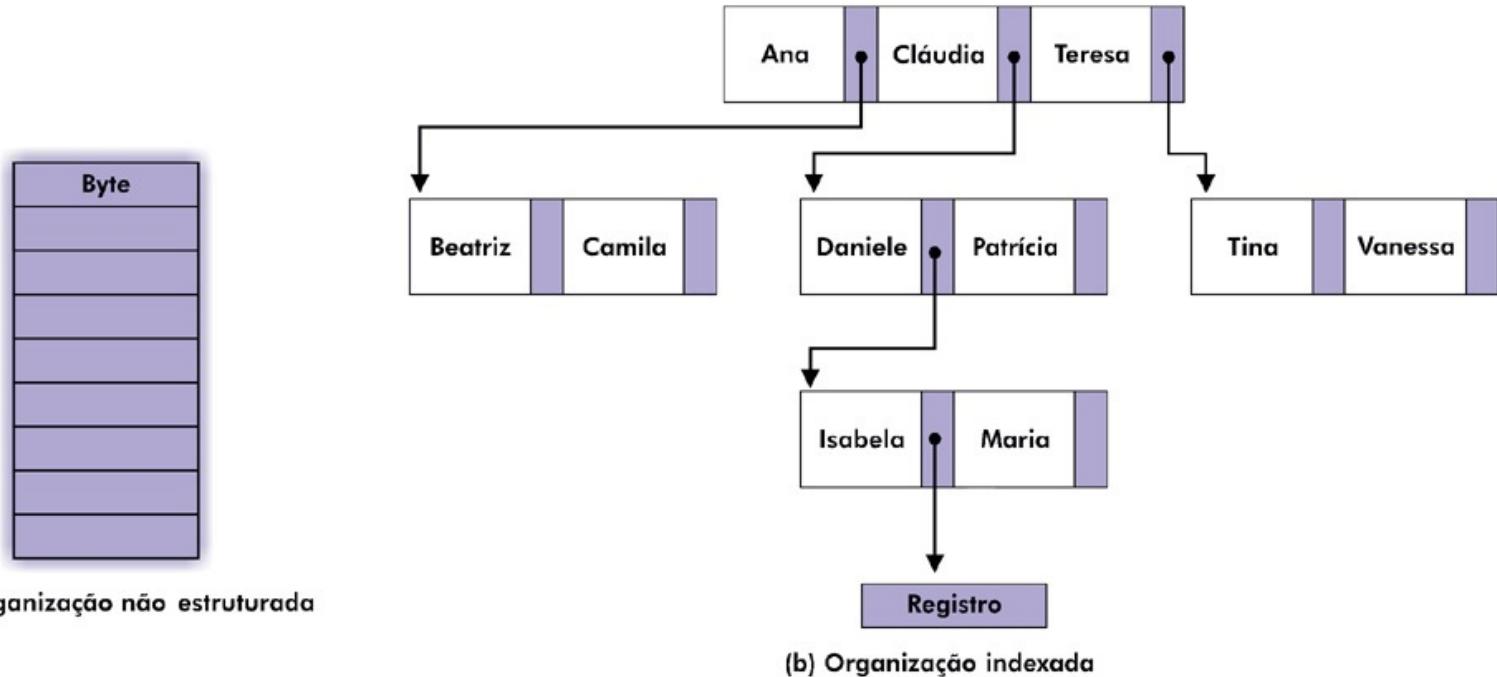


Fig. 11.1 Organização de arquivos.

Alguns sistemas operacionais possuem diferentes organizações de arquivos. Neste caso, cada arquivo criado deve seguir um modelo suportado pelo sistema de arquivos. As organizações mais conhecidas e implementadas são a sequencial, a relativa e a indexada (Fig. 11.1b). Nestes tipos de organização, podemos visualizar um arquivo como um conjunto de registros. Os registros podem ser classificados em registros de tamanho fixo, quando possuírem sempre o mesmo tamanho, ou registros de tamanho variável.

11.2.2 Métodos de Acesso

Em função de como o arquivo está organizado, o sistema de arquivos pode recuperar registros de diferentes maneiras. Inicialmente, os primeiros sistemas operacionais só armazenavam arquivos em fitas magnéticas. Com isso, o acesso era restrito à leitura dos registros na ordem em que eram gravados, e a gravação de novos registros só era possível no final do arquivo. Este tipo de acesso, chamado de *acesso sequencial*, era próprio da fita magnética, que, como meio de armazenamento, possuía esta limitação.

Com o advento dos discos magnéticos, foi possível a introdução de métodos de acesso mais eficientes. O primeiro a surgir foi o *acesso direto*, que permite a leitura/gravação de um registro diretamente na sua posição. Este método é realizado através do número do registro, que é a sua posição relativa ao início do arquivo. No acesso direto não existe restrição à ordem em que os registros são lidos ou gravados, sendo sempre necessária a especificação do número do registro. É importante notar que o acesso direto somente é possível quando o arquivo é definido com registros de tamanho fixo (Fig. 11.2).

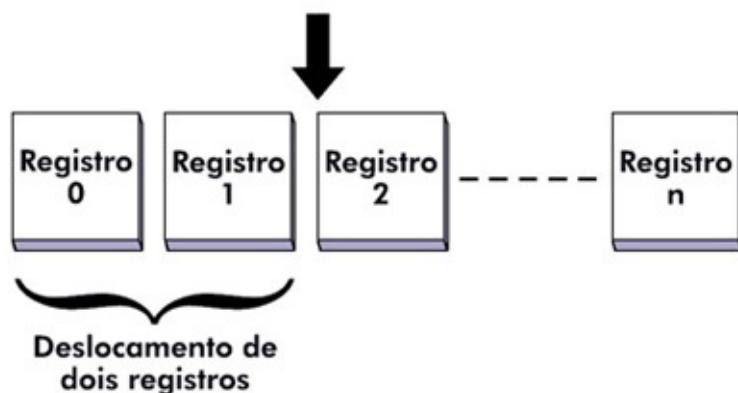


Fig. 11.2 Acesso direto.

O acesso direto pode ser combinado com o acesso sequencial. Com isso é possível acessar diretamente um registro qualquer de um arquivo e, a partir deste, acessar sequencialmente os demais.

Um método de acesso mais sofisticado, que tem como base o acesso direto, é o chamado *acesso indexado* ou *acesso por chave*. Para este acesso, o arquivo deve possuir uma área de índice onde existam ponteiros para os diversos registros. Sempre que a aplicação desejar acessar um registro, deverá ser especificada uma chave através da qual o sistema pesquisará na área de índice o ponteiro correspondente. A partir desta informação é realizado um acesso direto ao registro desejado.

11.2.3 Operações de Entrada/Saída

O sistema de arquivos disponibiliza um conjunto de rotinas que permite às aplicações realizarem operações de E/S, como tradução de nomes em endereços, leitura e gravação de dados e criação/eliminação de arquivos. Na realidade, as rotinas de E/S têm como função disponibilizar uma interface simples e uniforme entre a aplicação e os diversos dispositivos. A Fig. 11.3 ilustra a comunicação entre aplicação e dispositivos de maneira simplificada.

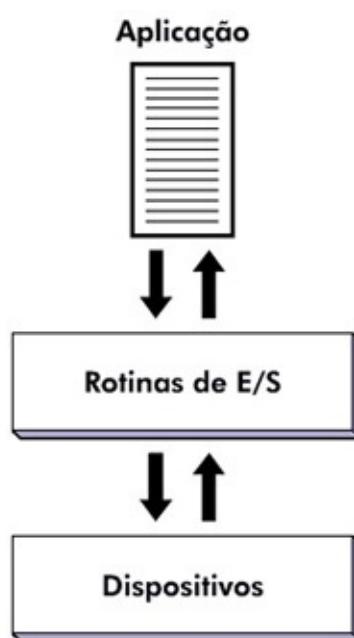


Fig. 11.3 Operações de entrada/saída.

A Tabela 11.2 apresenta algumas destas rotinas encontradas na maioria das implementações de sistemas de arquivos.

Tabela 11.2 Rotinas de entrada/saída

Rotina	Descrição
CREATE	Criação de arquivos.
OPEN	Abertura de um arquivo.
READ	Leitura de um arquivo.
WRITE	Gravação em um arquivo.
CLOSE	Fechamento de um arquivo.
DELETE	Eliminação de um arquivo.

11.2.4 Atributos

Cada arquivo possui informações de controle denominadas *atributos*. Os atributos variam dependendo do sistema de arquivos, porém alguns, como tamanho do arquivo, proteção, identificação do criador e data de criação, estão presentes em quase todos os sistemas.

Alguns atributos especificados na criação do arquivo não podem ser modificados em função de sua própria natureza, como organização e data/hora de criação. Outros são alterados pelo próprio sistema operacional, como tamanho e data/hora do último backup realizado. Existem ainda atributos que podem ser modificados pelo próprio usuário, como proteção do arquivo, tamanho máximo e senha de acesso. Na Tabela 11.3 são apresentados os principais atributos presentes nos sistemas de arquivos.

► 11.3 Diretórios

A estrutura de *diretórios* é como o sistema organiza logicamente os diversos arquivos contidos em um disco. O diretório é uma estrutura de dados que contém entradas associadas aos arquivos em que cada entrada armazena informações como localização física, nome, organização e demais atributos.

Tabela 11.3 Atributos de arquivos

Atributos	Descrição
Tamanho	Especifica o tamanho do arquivo.
Proteção	Código de proteção de acesso.

Dono

Identifica o criador do arquivo.

Criação

Data e hora de criação do arquivo.

Backup

Data e hora do último backup realizado.

Organização

Indica a organização lógica dos registros.

Senha

Senha necessária para acessar o arquivo.

Quando um arquivo é aberto, o sistema operacional procura a sua entrada na estrutura de diretórios, armazenando as informações sobre atributos e localização do arquivo em uma tabela mantida na memória principal. Esta tabela contém todos os arquivos abertos, sendo fundamental para aumentar o desempenho das operações com arquivos. É importante que ao término do uso de arquivos estes sejam fechados, ou seja, que se libere o espaço na tabela de arquivos abertos.

A implementação mais simples de uma estrutura de diretórios é chamada de nível único (single-level directory). Neste caso, somente existe um único diretório contendo todos os arquivos do disco (Fig. 11.4). Este modelo é bastante limitado, já que não permite que usuários criem arquivos com o mesmo nome, o que ocasionaria um conflito no acesso aos arquivos.

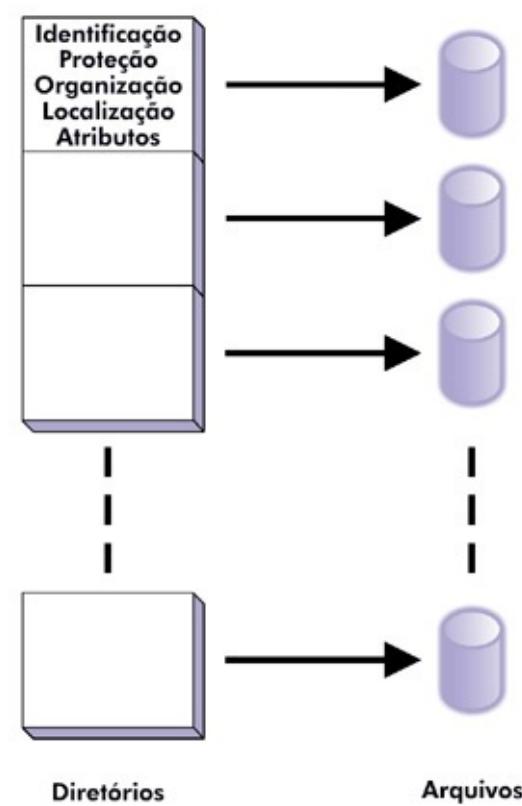


Fig. 11.4 Estrutura de diretórios de nível único.

Como o sistema de nível único é bastante limitado, uma evolução do modelo foi a implementação de uma estrutura em que para cada usuário existiria um diretório particular denominado User File Directory (UFD). Com esta implementação, cada usuário passa a poder criar arquivos com qualquer nome, sem a preocupação de conhecer os demais arquivos do disco.

Para que o sistema possa localizar arquivos nessa estrutura, deve haver um nível de diretório adicional para controlar os diretórios individuais dos usuários. Este nível, denominado Master File Directory (MFD), é indexado pelo nome do usuário, e nele cada entrada aponta para o diretório pessoal. A Fig. 11.5 ilustra este modelo de estrutura de diretórios com dois níveis (two-level directory).

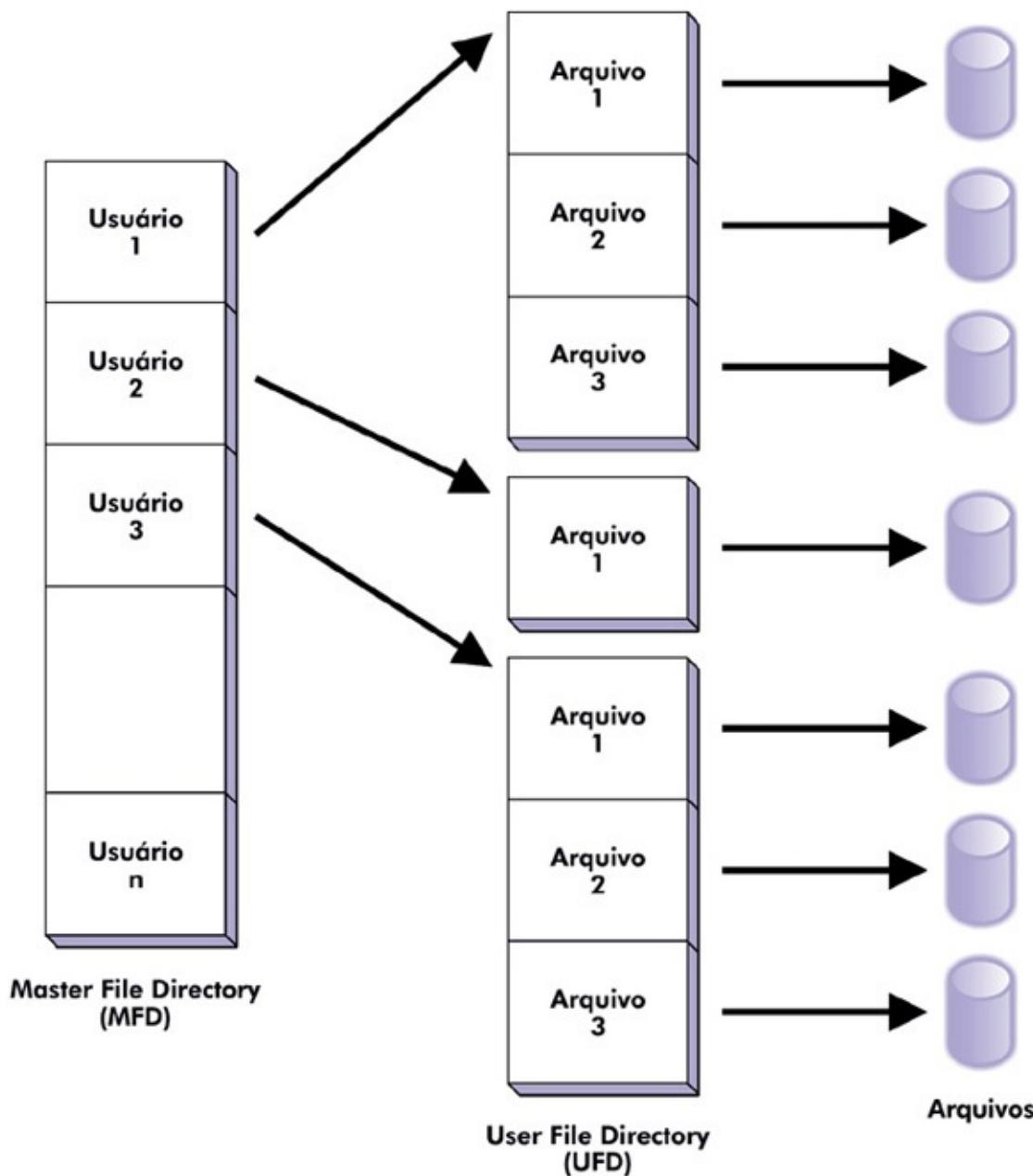


Fig. 11.5 Estrutura de diretórios com dois níveis.

A estrutura de diretórios com dois níveis é análoga a uma estrutura de dados em árvore, em que o MFD é a raiz, os galhos são os UFD e os arquivos são as folhas. Neste tipo de estrutura, quando se referencia um arquivo é necessário especificar, além do seu nome, o diretório onde ele se localiza. Esta referência é chamada de path (caminho). Como exemplo, caso o usuário CARLOS necessite acessar um arquivo próprio chamado DOCUMENTO.TXT, este pode ser referenciado como /CARLOS/DOCUMENTO.TXT. Cada sistema de arquivos possui sua própria sintaxe para especificação de diretórios e arquivos.

Sob o ponto de vista do usuário, a organização dos seus arquivos em um único diretório não permite uma organização adequada. A extensão do modelo de dois níveis para um de múltiplos níveis permitiu que os

arquivos fossem logicamente mais bem organizados. Este novo modelo, chamado estrutura de diretórios em árvore (tree-structured directory), é adotado pela maioria dos sistemas (Fig. 11.6).

Na estrutura em árvore, cada usuário pode criar diversos níveis de diretórios, também chamados subdiretórios. Cada diretório pode conter arquivos ou outros diretórios. O número de níveis de uma estrutura em árvore é dependente do sistema de arquivos de cada sistema operacional. No Open VMS são possíveis oito níveis de diretórios.

Um arquivo, nesta estrutura em árvore, pode ser especificado unicamente por meio de um path absoluto, descrevendo todos os diretórios percorridos a partir da raiz (MFD) até o diretório no qual o arquivo está ligado. Na Fig. 11.7, o path absoluto do arquivo SOMA.EXE é /PAULO/PROGRAMAS. Na maioria dos sistemas, os diretórios também são tratados como arquivos, possuindo identificação e atributos, como proteção, identificador do criador e data de criação.

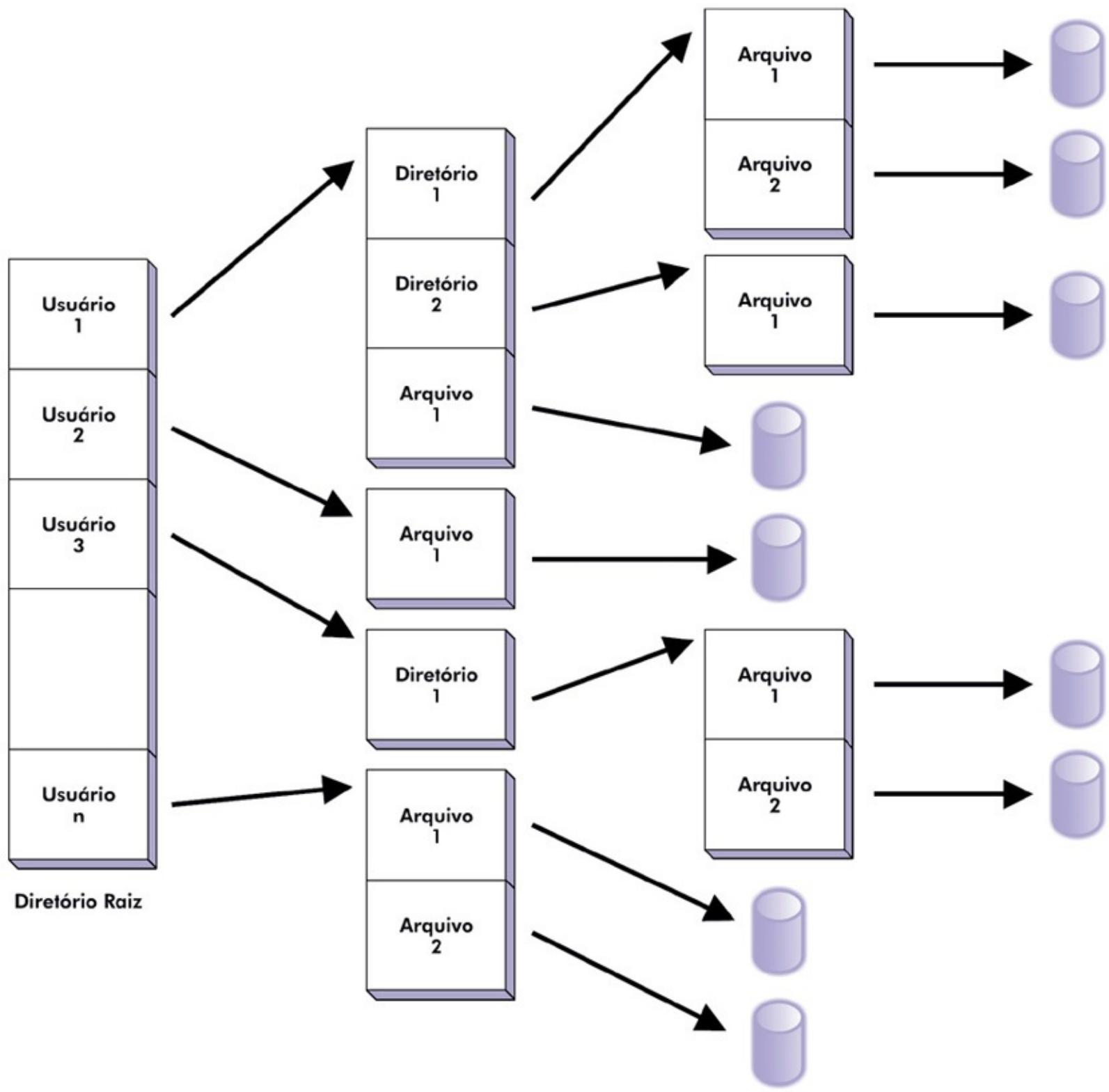


Fig. 11.6 Estrutura de diretórios em árvore.

► 11.4 Gerência de Espaço Livre em Disco

A criação de arquivos em disco exige que o sistema operacional tenha o controle de quais áreas ou blocos no disco estão livres. Este controle é realizado utilizando-se alguma estrutura de dados que armazena informações que possibilitam ao sistema de arquivos gerenciar o espaço livre do disco. Nesta estrutura, geralmente uma lista ou tabela, é possível identificar blocos livres que poderão ser alocados a um novo arquivo. Neste caso, o espaço é removido da estrutura para que não seja reutilizado. No momento em que um arquivo é eliminado, todos os seus blocos são liberados para a lista de espaços

livres.

A forma mais simples de implementar uma estrutura de espaços livres é através de uma tabela denominada mapa de bits (bit map). Cada entrada na tabela é associada a um bloco do disco representado por um bit, podendo assumir valor igual a 0 (indicando bloco livre) ou 1 (indicando bloco alocado). Na Fig. 11.8a podemos observar um exemplo desta implementação, que apresenta como principal problema um excessivo gasto de memória, já que para cada bloco do disco deve existir uma entrada na tabela.

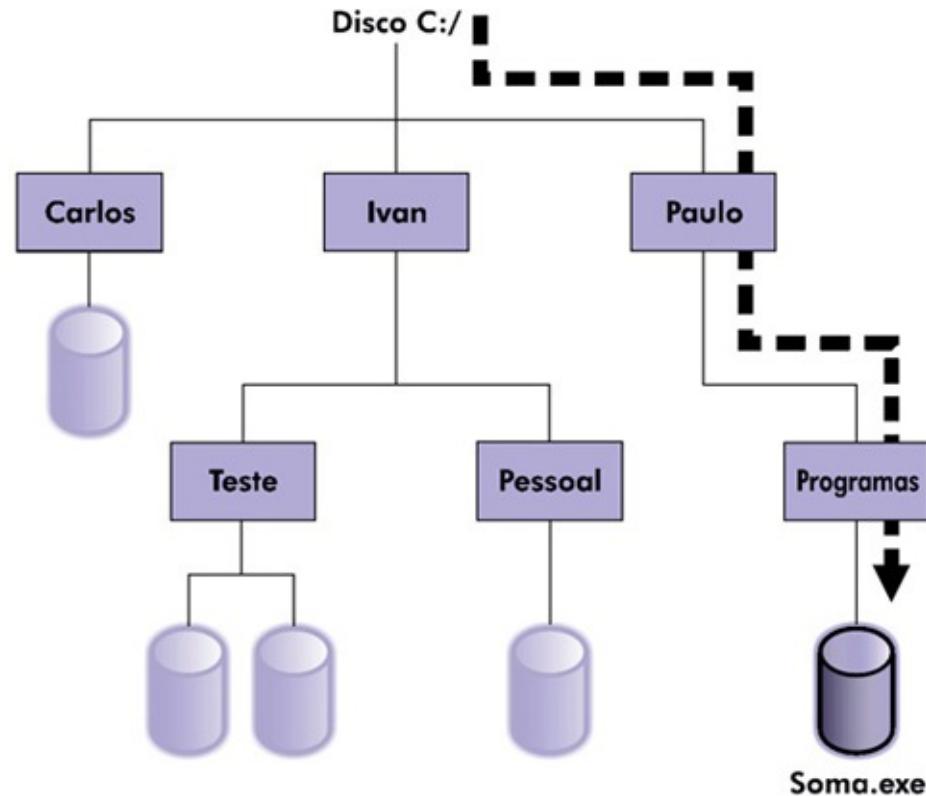


Fig. 11.7 Path de um arquivo.

Uma segunda maneira de realizar este controle é com uma estrutura de lista encadeada de todos os blocos livres do disco. Para que isto seja possível, cada bloco possui uma área reservada para armazenamento do endereço do próximo bloco. A partir do primeiro bloco livre é, então, possível o acesso sequencial aos demais de forma encadeada (Fig. 11.8b). Este esquema apresenta algumas restrições se considerarmos que, além do espaço utilizado no bloco com informação de controle, o algoritmo de busca de espaço livre sempre deve realizar uma pesquisa sequencial na lista.

Outra solução leva em consideração que blocos contíguos são geralmente alocados ou liberados simultaneamente. Podemos, desta forma, enxergar o disco como um conjunto de segmentos de blocos livres. Com base neste conceito, é possível manter uma tabela com o endereço do primeiro bloco de cada segmento e o número de blocos livres contíguos que se seguem. Esta técnica de gerência de espaço livre é conhecida como tabela de blocos livres (Fig. 11.8c).

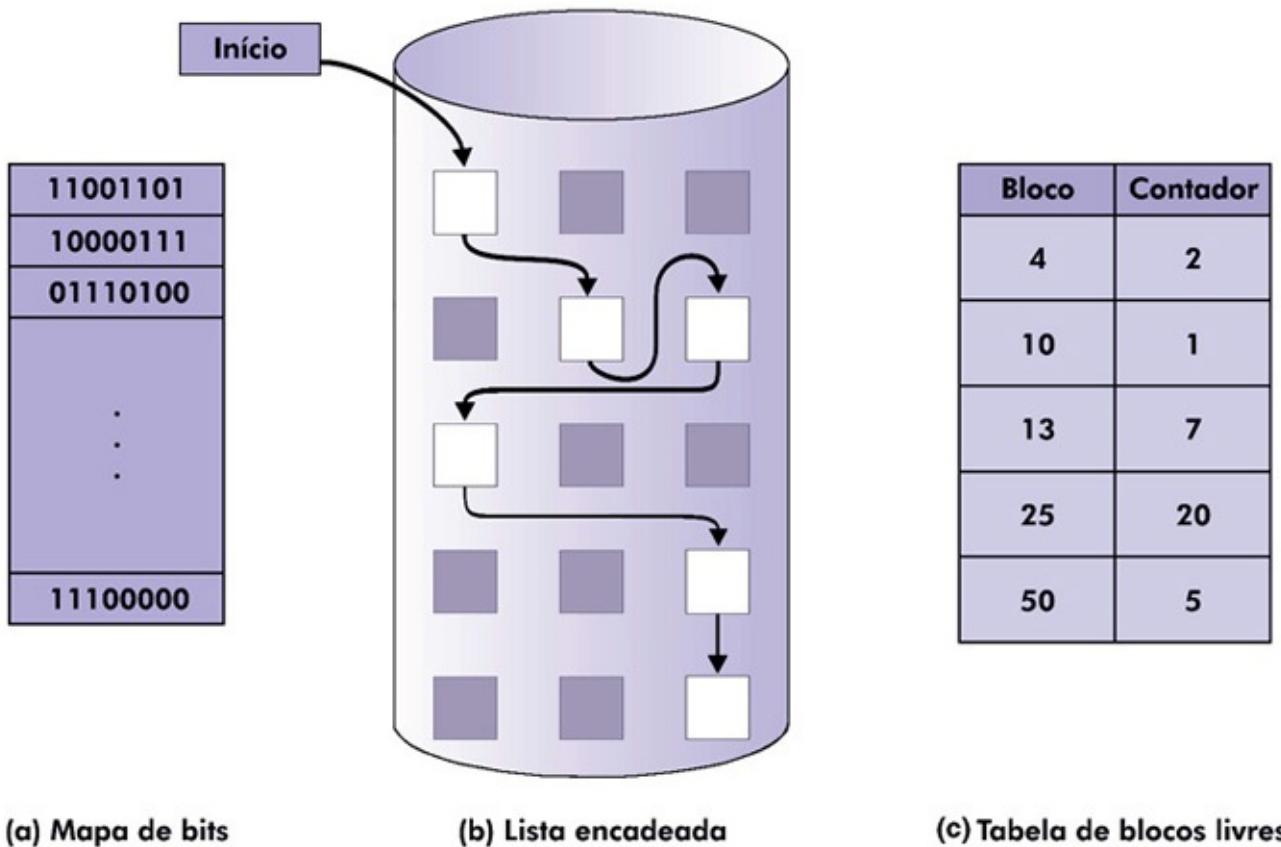


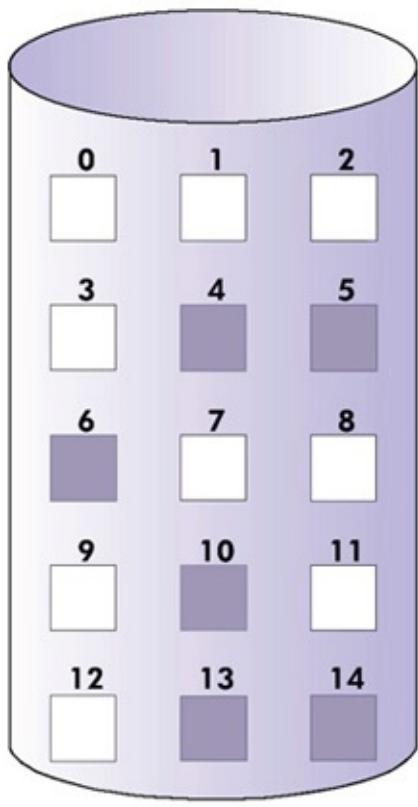
Fig. 11.8 Alocação de espaço em disco.

► 11.5 Gerência de Alocação de Espaço em Disco

Da mesma forma que o sistema operacional gerencia os espaços livres no disco, a gerência dos espaços alocados aos arquivos é de fundamental importância em um sistema de arquivos. A seguir, as principais técnicas de alocação serão apresentadas.

11.5.1 Alocação Contígua

A *alocação contígua* consiste em armazenar um arquivo em blocos sequencialmente dispostos no disco. Neste tipo de alocação, o sistema localiza um arquivo através do endereço do primeiro bloco e da sua extensão em blocos (Fig. 11.9).



Arquivo	Bloco	Extensão
A.TXT	4	3
B.TXT	10	1
C.TXT	13	2

Fig. 11.9 Alocação contígua.

O acesso a arquivos dispostos contiguamente no disco é bastante simples tanto para a forma sequencial quanto para a direta. Seu principal problema é a alocação de espaço livre para novos arquivos. Caso um arquivo deva ser criado com determinado tamanho, é necessário existir uma quantidade suficiente de blocos contíguos no disco para realizar a alocação.

Para este tipo de alocação, podemos enxergar o disco como um grande vetor, onde os elementos podem ser considerados segmentos com tamanhos diferentes de blocos contíguos. Estes segmentos estão dispostos alternadamente entre segmentos ocupados (blocos alocados) e segmentos livres (blocos livres). No momento em que o sistema operacional deseja alocar espaço para armazenar um novo arquivo, pode existir mais de um segmento livre disponível com o tamanho exigido. Neste caso, é necessário que alguma estratégia de alocação seja adotada para selecionar qual o segmento na lista de blocos livres deve ser escolhido. Analisaremos a seguir as principais estratégias:

■ First-fit

Neste caso, o primeiro segmento livre com tamanho suficiente para alocar o arquivo é selecionado. A busca na lista é sequencial, sendo interrompida tão logo se localize um segmento com tamanho adequado.

■ Best-fit

A alocação best-fit seleciona o menor segmento livre disponível com tamanho suficiente para armazenar o arquivo. A busca em toda a lista se faz necessária para a seleção do segmento, a não ser que a lista esteja ordenada por tamanho.

■ Worst-fit

Neste caso, o maior segmento é alocado. Mais uma vez a busca em toda a lista se faz necessária, a menos que exista uma ordenação por tamanho.

Independentemente da estratégia utilizada, a alocação contígua apresenta um problema chamado fragmentação dos espaços livres. Como os arquivos são criados e eliminados frequentemente, os segmentos livres vão se fragmentando em pequenos pedaços por todo o disco. O problema pode tornar-se crítico quando um disco possui blocos livres disponíveis, porém não existe um segmento contíguo em que o arquivo possa ser alocado.

O problema da fragmentação pode ser contornado através de rotinas que reorganizem todos os arquivos no disco de maneira que só exista um único segmento de blocos livres. Este procedimento, denominado desfragmentação, geralmente utiliza uma área de trabalho no próprio disco ou em fita magnética (Fig. 11.10). Existe um grande consumo de tempo neste tipo de operação. É importante também ressaltar que a desfragmentação é um procedimento com efeito temporário e deve, portanto, ser realizada periodicamente.



Fig. 11.10 Desfragmentação.

Podemos concluir que a alocação contígua apresenta alguns inconvenientes, sendo o principal problema a determinação do espaço em disco necessário a um arquivo. Nem sempre, no momento da criação de um arquivo, é possível determinar qual o seu tamanho em definitivo, podendo posteriormente existir a necessidade de extensão. Por esta operação ser complexa na alocação contígua, a pré-alocação de espaço é uma solução que, apesar de resolver o problema, pode ocasionar que parte do espaço alocado permaneça ocioso por um longo período de tempo.

11.5.2 Alocação Encadeada

Na *alocação encadeada*, um arquivo pode ser organizado como um conjunto de blocos ligados logicamente no disco, independente da sua localização física. Cada bloco deve possuir um ponteiro para o bloco seguinte do arquivo, e assim sucessivamente (Fig. 11.11).

A fragmentação dos espaços livres, apresentada no método anterior, não ocasiona nenhum problema na

alocação encadeada, pois os blocos livres alocados para um arquivo não precisam necessariamente estar contíguos. O que ocorre neste método é a fragmentação de arquivos, que é a quebra do arquivo em diversos pedaços denominados extents.

A fragmentação resulta no aumento do tempo de acesso aos arquivos, pois o mecanismo de leitura/gravação do disco deve se deslocar diversas vezes sobre sua superfície para acessar cada extent (excessivo tempo de seek). Para otimizar o tempo das operações de E/S neste tipo de sistema, é importante que o disco seja periodicamente desfragmentado. Apesar de ter propósitos diferentes, o procedimento de desfragmentação é idêntico ao já apresentado na alocação contígua.

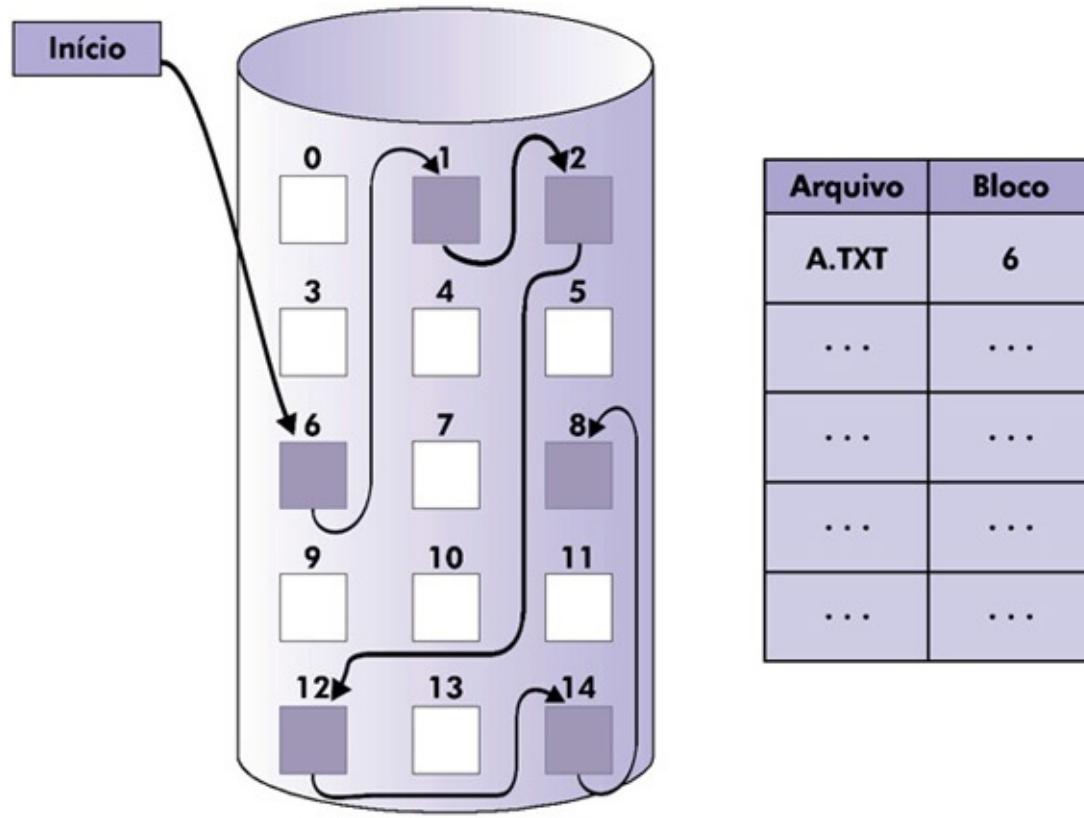


Fig. 11.11 Alocação encadeada.

A alocação encadeada só permite que se realize acesso sequencial aos blocos dos arquivos. Isto constitui uma das principais desvantagens desta técnica, já que não é possível o acesso direto aos blocos. Além disso, essa técnica desperdiça espaço nos blocos com o armazenamento de ponteiros.

11.5.3 Alocação Indexada

A *alocação indexada* soluciona uma das principais limitações da alocação encadeada, que é a impossibilidade do acesso direto aos blocos dos arquivos. O princípio desta técnica é manter os ponteiros de todos os blocos do arquivo em uma única estrutura denominada bloco de índice.

A alocação indexada, além de permitir o acesso direto aos blocos do arquivo, não utiliza informações de controle nos blocos de dados, como existente na alocação encadeada (Fig. 11.12).

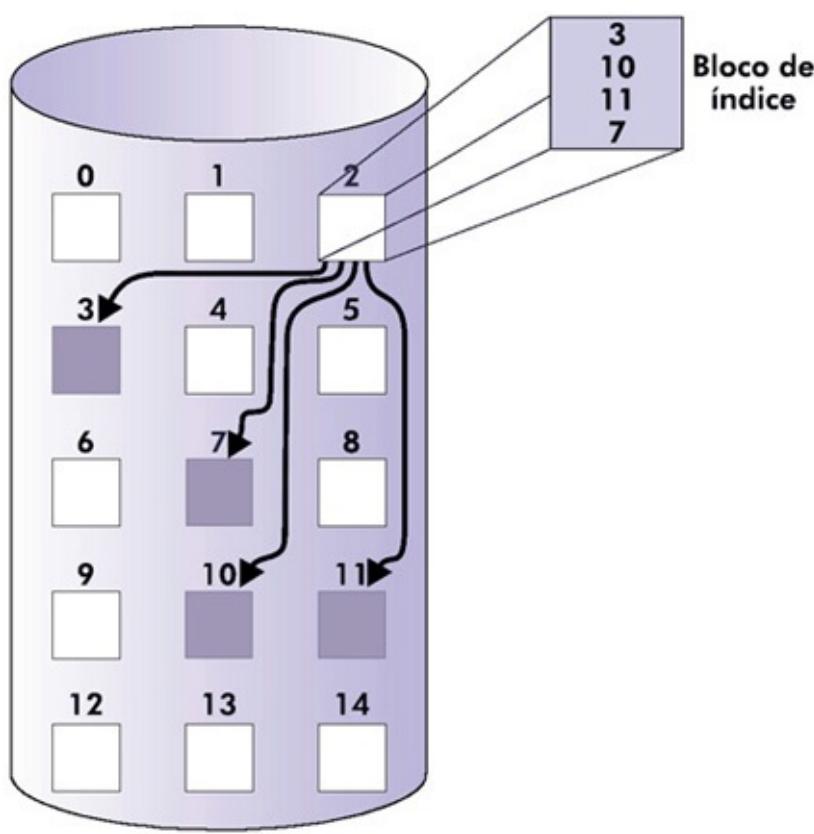


Fig. 11.12 Alocação indexada.

► 11.6 Proteção de Acesso

Considerando que os meios de armazenamento são compartilhados entre diversos usuários, é de fundamental importância que mecanismos de proteção sejam implementados para garantir a proteção individual de arquivos e diretórios. Qualquer sistema de arquivos deve possuir mecanismos próprios para proteger o acesso às informações gravadas em discos e fitas, além de possibilitar o compartilhamento de arquivos entre usuários, quando desejado.

Em geral, o tipo de acesso a arquivos é implementado mediante a concessão ou não dos diferentes acessos que podem ser realizados, como leitura (read), gravação (write), execução (execute) e eliminação (delete). O acesso de leitura está relacionado com qualquer tipo de operação em que o arquivo possa ser visualizado, como a exibição do seu conteúdo, edição através de um editor de textos ou até mesmo a cópia para criação de um novo arquivo. O acesso de gravação está relacionado à alteração no conteúdo do arquivo, como inclusão ou alteração de registros. O acesso de execução só tem sentido quando associado a arquivos executáveis ou arquivos de comandos, indicando o direito de execução do arquivo. Finalmente, o acesso de eliminação expressa a permissão para eliminação do arquivo.

O controle de acesso às operações realizadas com diretórios possui diferenças em relação às operações com arquivos. Controle da criação/eliminação de arquivos nos diretórios, visualização do seu conteúdo e eliminação do próprio diretório são operações que também devem ser protegidas.

Existem diferentes mecanismos e níveis de proteção, cada qual com suas vantagens e desvantagens — para cada tipo de sistema, um modelo é mais adequado do que outro. A seguir, apresentaremos três diferentes mecanismos de proteção, presentes na maioria dos sistemas de arquivos.

11.6.1 Senha de Acesso

A associação de uma *senha de acesso* a um arquivo é um princípio bastante simples. O controle de acesso se resume ao usuário ter o conhecimento da senha e, consequentemente, ter a liberação do acesso ao arquivo concedida pelo sistema.

Como cada arquivo possui apenas uma senha, o acesso é liberado ou não na sua totalidade. Isto significa que não é possível determinar quais tipos de operação podem ou não ser concedidas. Outra desvantagem deste método é a dificuldade de compartilhamento de arquivos, pois, além do dono do arquivo, todos os demais usuários teriam que conhecer a senha de acesso.

11.6.2 Grupos de Usuário

A proteção baseada em *grupos de usuários* é implementada por diversos sistemas operacionais. Este tipo de proteção tem como princípio a associação de cada usuário do sistema a um grupo. Os grupos de usuários são organizados logicamente com o objetivo de compartilhar arquivos e diretórios — os usuários que desejam compartilhar arquivos entre si devem pertencer a um mesmo grupo.

Esse mecanismo implementa três níveis de proteção ao arquivo: owner (dono), group (grupo) e all (todos). Na criação do arquivo, o usuário especifica se o arquivo deve ser acessado somente pelo seu criador, pelos usuários do grupo ao qual ele pertence ou por todos os usuários do sistema. Nessa especificação é necessário associar o tipo de acesso (leitura, escrita, execução e eliminação) aos três níveis de proteção.

Vejamos um exemplo na Fig. 11.13, onde a proteção do arquivo DADOS.TXT libera completamente o acesso para o dono, permite somente o compartilhamento para leitura dentro do grupo e não libera qualquer tipo de acesso para os demais usuários do sistema. Em geral, somente o dono ou usuários privilegiados é que podem modificar a proteção dos arquivos.

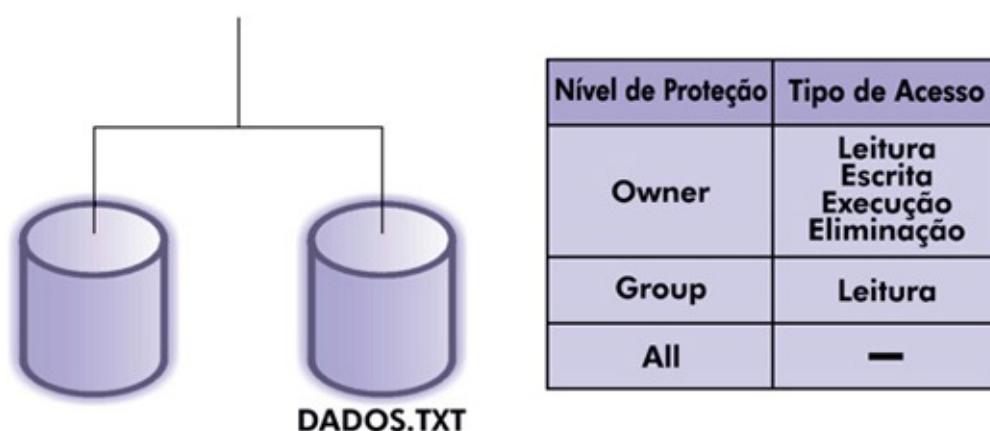


Fig. 11.13 Proteção por grupos de usuários.

11.6.3 Lista de Controle de Acesso

A *Lista de Controle de Acesso* (*Access Control List* — ACL) consiste em uma lista associada a cada arquivo, onde são especificados quais os usuários e os tipos de acesso permitidos (Fig. 11.14). Nesse caso, quando um usuário tenta acessar um arquivo, o sistema operacional verifica se a lista de controle autoriza a operação desejada.

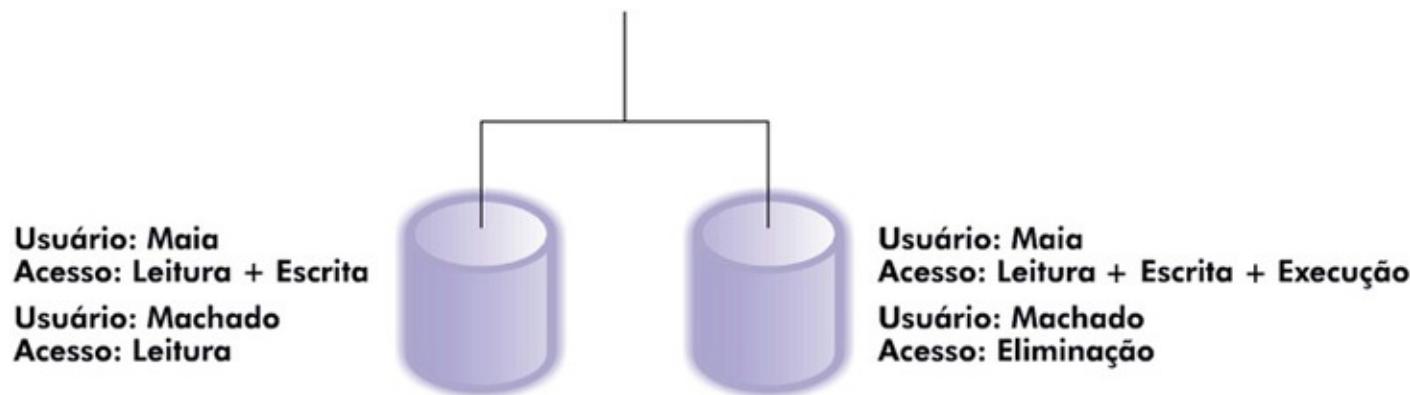


Fig. 11.14 Lista de controle de acessos.

O tamanho dessa estrutura de dados pode ser bastante extenso, se considerarmos que um arquivo pode ter seu acesso compartilhado por diversos usuários. Além deste fato, existe um overhead adicional, se comparado com o mecanismo de proteção por grupo de usuários, devido à pesquisa sequencial que o sistema deverá realizar na lista sempre que um acesso for solicitado.

Em determinados sistemas de arquivos é possível encontrar tanto o mecanismo de proteção por grupos de usuários quanto o de lista de controle de acesso, oferecendo, desta forma, maior flexibilidade ao mecanismo de proteção de arquivos e diretórios.

► 11.7 Implementação de Caches

O acesso a disco é bastante lento se comparado ao acesso à memória principal, devido à arquitetura dos discos magnéticos. Este é o principal motivo das operações de E/S com discos serem um problema para o desempenho do sistema.

Com o objetivo de minimizar este problema, a maioria dos sistemas de arquivos implementa uma técnica denominada *buffer cache*. Neste esquema, o sistema operacional reserva uma área da memória para que se tornem disponíveis caches utilizados em operações de acesso ao disco. Quando uma operação é realizada, seja leitura ou gravação, o sistema verifica se a informação desejada se encontra no buffer cache. Em caso positivo, não é necessário o acesso ao disco. Caso o bloco requisitado não se encontre no cache, a operação de E/S é realizada e o cache é atualizado. Como existe uma limitação no tamanho do cache, cada sistema adota políticas para substituição de blocos como a FIFO ou a LRU.

Apesar de esta implementação melhorar o desempenho do sistema, aspectos de segurança devem ser levados em consideração. No caso de blocos de dados permanecerem por um longo período de tempo na memória principal, a ocorrência de problemas de energia pode ocasionar a perda de tarefas já realizadas e consideradas já salvas em disco.

Existem duas maneiras distintas de tratar este problema. No primeiro caso, o sistema operacional possui uma rotina que executa periodicamente em um intervalo de tempo, atualizando em disco todos os blocos modificados do cache. Uma segunda alternativa é, toda vez que um bloco do cache for modificado, que seja realizada imediatamente uma atualização no disco (write-through caches).

Analisando comparativamente as duas técnicas, podemos concluir que a primeira implica menor quantidade de operações de E/S, porém o risco de perda de dados é maior. Apesar de tal probabilidade ser pequena, pode ocorrer que dados atualizados em um arquivo e ainda no cache sejam perdidos no caso de falta de energia. Isto já não aconteceria nos caches do tipo write-through, em função do seu próprio funcionamento, porém o aumento considerável nas operações de E/S tornam este método menos eficiente. Atualmente, a maioria dos sistemas utiliza a primeira técnica de otimização.

► 11.8 Exercícios

1. O que é um arquivo?
2. Como arquivos podem ser organizados?
3. Diferencie os métodos de acesso a registros sequencial, direto e indexado.
4. Qual a função das rotinas de E/S?
5. Quais as diferentes formas de implementação de uma estrutura de diretório?
6. Descreva as vantagens e desvantagens das técnicas para gerência de espaços livres.
7. O que é alocação contígua de blocos e quais benefícios a desfragmentação pode proporcionar quando esta técnica é utilizada?
8. Descreva as vantagens e desvantagens das técnicas de alocação encadeada e indexada na gerência de alocação de espaço em disco.
9. Quais os tipos de proteção de acesso a arquivos existentes e quais suas principais vantagens?
10. O que é a técnica denominada buffer cache?

12

Gerência de Dispositivos

► 12.1 Introdução

A gerência de dispositivos de entrada/saída é uma das principais e mais complexas funções de um sistema operacional. Sua implementação é estruturada através de camadas em um modelo semelhante ao apresentado para o sistema operacional como um todo. As camadas de mais baixo nível escondem características dos dispositivos das camadas superiores, oferecendo uma interface simples e confiável ao usuário e suas aplicações (Fig. 12.1).

A diversidade dos dispositivos de E/S exige que o sistema operacional implemente uma camada, chamada de *subsistema de E/S*, com a função de isolar a complexidade dos dispositivos físicos. Dessa forma, é possível ao sistema operacional ser flexível, permitindo a comunicação das aplicações com qualquer tipo de periférico. Aspectos como velocidade de operação, unidade de transferência, representação dos dados, tipos de operações e demais detalhes de cada periférico são tratados pela camada de *device driver*, oferecendo uma interface uniforme entre o subsistema de E/S e todos os dispositivos.

As camadas são divididas em dois grupos, com o primeiro grupo visualizando os diversos tipos de dispositivos do sistema de um modo único (Fig. 12.1a), enquanto o segundo é específico para cada dispositivo (Fig. 12.1b). A maior parte das camadas trabalha de forma independente do dispositivo físico.

Neste capítulo serão examinadas as diversas camadas que compõem a gerência de dispositivos de E/S, além dos conceitos envolvidos em cada nível. No final do capítulo serão abordados aspectos relacionados especificamente aos discos magnéticos, como desempenho e segurança de dados.

► 12.2 Subsistema de Entrada e Saída

O sistema operacional deve tornar as operações de E/S o mais simples possível para o usuário e suas

aplicações. O *subsistema de E/S* isola a complexidade de operações específicas para cada tipo de dispositivo da camada de sistema de arquivo, do sistema gerenciador de banco de dados (SGBD) ou diretamente da aplicação. Dessa forma, é possível que as aplicações manipulem qualquer tipo de periférico com mais simplicidade.

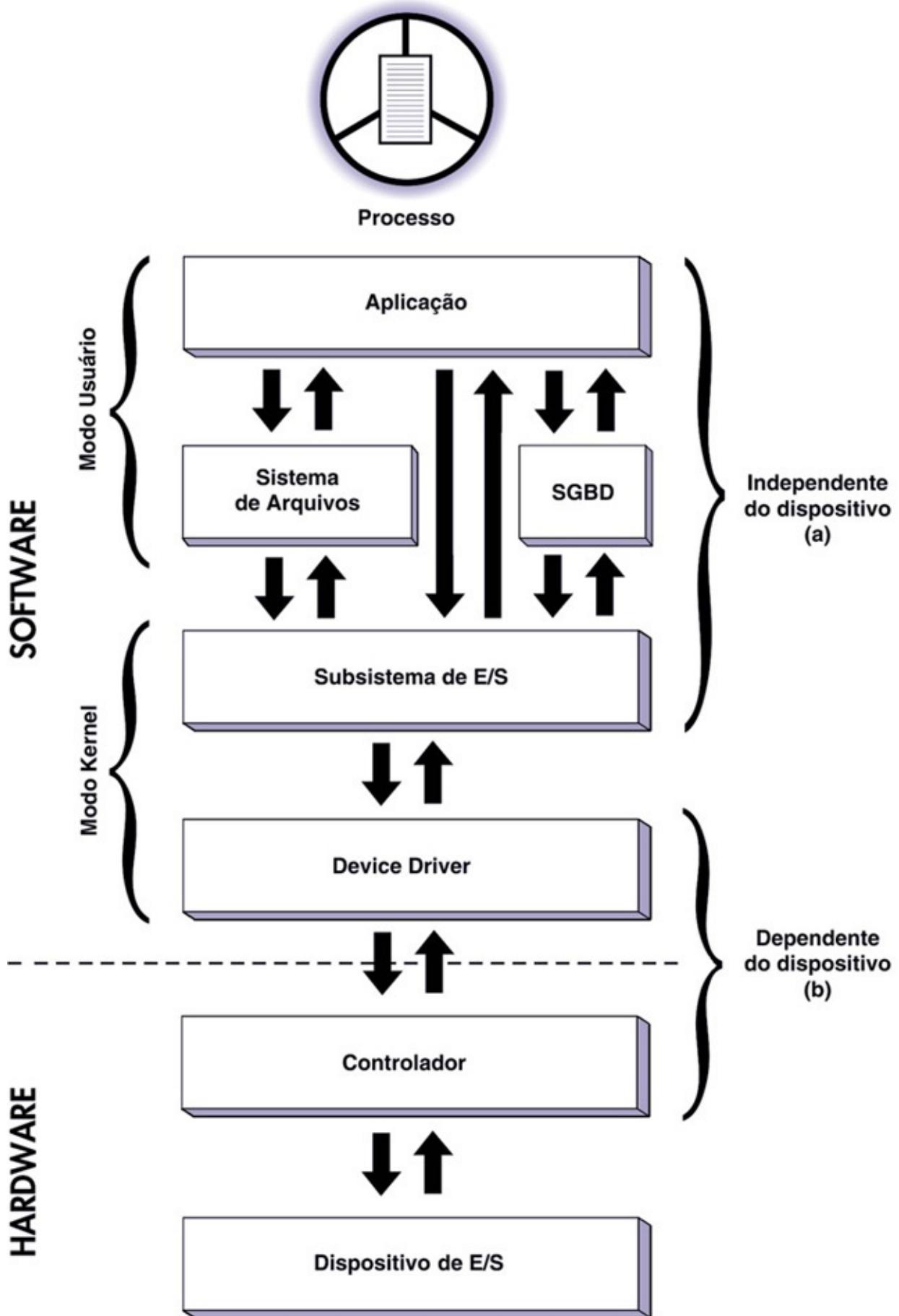


Fig. 12.1 Arquitetura de camadas da gerência de dispositivos.

O subsistema de E/S é composto por um conjunto de rotinas que possibilita a comunicação com qualquer dispositivo que possa ser conectado ao computador. Esse conjunto de rotinas, denominado *rotinas de*

entrada/saída, permite ao usuário realizar operações de E/S sem se preocupar com detalhes do dispositivo que está sendo acessado. Nesse caso, quando um usuário cria um arquivo em disco não lhe interessa como é a formatação do disco, nem em que trilha ou setor o arquivo será gravado.

As operações de E/S são realizadas por intermédio de chamadas às rotinas de E/S, possibilitando a independência da aplicação com relação a características específicas das arquiteturas dos diferentes dispositivos. Com isso, é possível escrever um programa que manipule arquivos, estejam eles em disquetes, discos rígidos ou fita magnética, sem ter que alterar o código para cada tipo de dispositivo.

Aplicações podem realizar operações de E/S de dados que podem estar logicamente estruturados em um dispositivo como também em dados armazenados de forma não estruturada. A possibilidade de estruturação de dados vai depender do tipo de periférico envolvido na operação. Para dispositivos em que é possível o armazenamento não volátil da informação, como os discos magnéticos ou discos ópticos, os dados podem estar estruturados conforme organização do sistema de arquivos disponibilizado pelo sistema operacional ou em um sistema gerenciador de banco de dados. Neste caso a aplicação realiza operações de E/S que manipulam os dados segundo a organização utilizada. Quando o dispositivo de E/S não possibilita a estruturação de dados no periférico, como em monitores, as operações de E/S estão ligadas diretamente às rotinas de E/S do sistema operacional.

A Fig. 12.1 ilustra as diferentes maneiras de uma aplicação interagir com o subsistema de E/S. A primeira forma é a *chamada explícita*, em que a partir de um código de alto nível é chamada uma rotina de E/S do sistema operacional diretamente. A outra forma é denominada *chamada implícita*, que é uma maneira mais simples de ter acesso a um dispositivo com o uso de comandos de leitura/gravação e chamadas a bibliotecas de rotinas oferecidas por linguagens de alto nível, como Pascal ou C. A comunicação entre os comandos de E/S oferecidos pelas linguagens de programação de alto nível e as rotinas de E/S é feita simplesmente através de passagem de parâmetros. O relacionamento entre o comando e a rotina de E/S é criado na geração do código executável do programa.

As operações de E/S podem ser classificadas conforme o seu sincronismo. Uma operação é dita síncrona quando o processo que realizou a operação fica aguardando no estado de espera pelo seu término. A maioria dos comandos das linguagens de alto nível funciona desta forma. Uma operação é dita assíncrona quando o processo que realizou a operação não aguarda pelo seu término e continua pronto para ser executado. Neste caso, o sistema deve oferecer algum mecanismo de sinalização que avise ao processo que a operação foi terminada.

O subsistema de entrada e saída é responsável por realizar as funções comuns a todos os tipos de dispositivos, ficando os aspectos específicos de cada periférico como responsabilidade dos device drivers. Dessa forma, o subsistema de E/S é a parte do sistema operacional que oferece uma interface uniforme com as camadas superiores.

Cada dispositivo trabalha com unidades de informação de tamanhos diferentes, como caracteres ou blocos. O subsistema de E/S é responsável por criar uma unidade lógica de transferência independente do dispositivo e repassá-la para os níveis superiores, sem o conhecimento do conteúdo da informação. No caso de a camada superior ser o sistema de arquivos, esta informação poderá ser interpretada como um registro lógico de um arquivo, devendo obedecer a uma certa organização e método de acesso estabelecidos.

Normalmente, o tratamento de erros nas operações de E/S é realizado pelas camadas mais próximas ao hardware. Existem, porém, certos erros que podem ser tratados e reportados de maneira uniforme pelo sistema de arquivos, independentemente do dispositivo. Erros como a gravação em dispositivos de entrada, leitura em dispositivos de saída e operações em dispositivos inexistentes podem ser tratados neste nível.

Todos os dispositivos de E/S são controlados, com o objetivo de obter o maior compartilhamento possível entre os diversos usuários de forma segura e confiável. Alguns dispositivos, como os discos, podem ser compartilhados, simultaneamente, entre diversos usuários, sendo o sistema operacional responsável pela integridade dos dados acessados. Outros, como as impressoras, devem ter acesso exclusivo, e o sistema deve controlar seu compartilhamento de forma organizada. O subsistema de E/S é responsável também por implementar todo um mecanismo de proteção de acesso aos dispositivos. No momento em que o usuário realiza uma operação de E/S, é verificado se o seu processo possui permissão para realizar a operação.

A técnica de *buffering* é outra tarefa implementada por esse subsistema, permitindo reduzir o número de operações de E/S, utilizando uma área de memória intermediária chamada de *buffer*. Por exemplo, quando um dado é lido do disco o sistema traz para a área de buffer não só o dado solicitado, mas um bloco de dados. Caso haja uma solicitação de leitura de um novo dado que pertença ao bloco anteriormente lido, não existe a necessidade de uma nova operação de E/S, melhorando desta forma a eficiência do sistema.

Uma das principais funções do subsistema de E/S é criar uma interface padronizada com os device drivers. Sempre que um novo dispositivo é instalado no computador, é necessário que um novo driver seja adicionado ao sistema. O subsistema de E/S deve oferecer uma interface padronizada que permita a inclusão de novos drivers sem a necessidade de alteração da camada de subsistema de E/S.

► 12.3 Device Driver

O device driver, ou somente driver, tem como função implementar a comunicação do subsistema de E/S com os dispositivos, por intermédio dos controladores de E/S. Enquanto o subsistema de E/S trata de funções ligadas a todos os dispositivos, os drivers tratam apenas dos seus aspectos particulares.

Os drivers têm como função receber comandos gerais sobre acessos aos dispositivos e traduzi-los para comandos específicos, que poderão ser executados pelos controladores (Fig. 12.2). Cada device driver manipula somente um tipo de dispositivo ou grupo de dispositivos semelhantes. Normalmente, um sistema possui diferentes drivers, como drivers para disco, fita magnética, rede e vídeo.

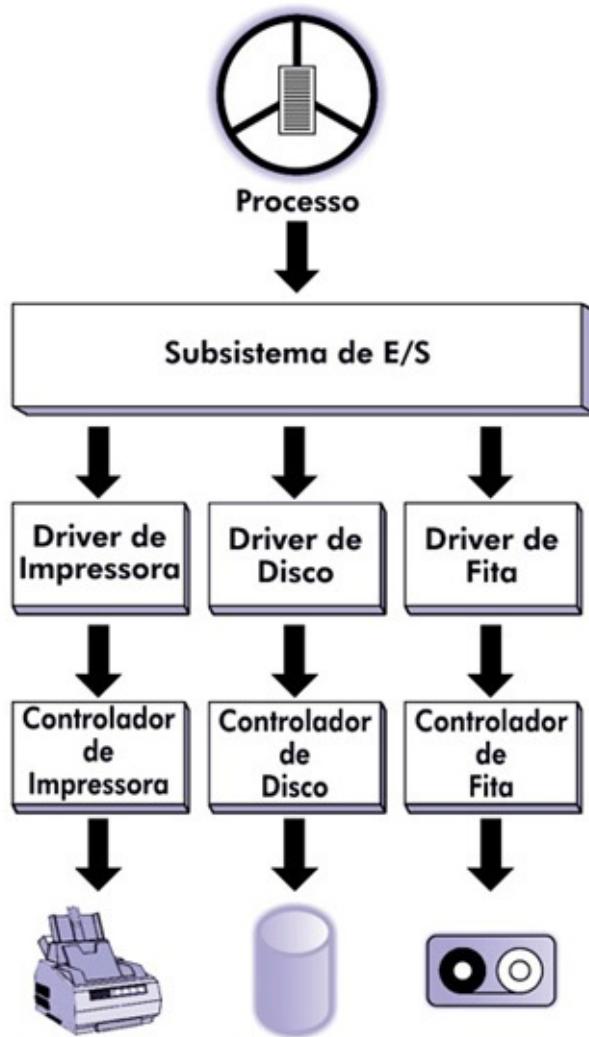


Fig. 12.2 Device drivers.

O driver está integrado diretamente às funções do controlador, sendo o componente do sistema que reconhece as características particulares do funcionamento de cada dispositivo de E/S, como número de registradores do controlador, funcionamento e comandos do dispositivo. Sua função principal é receber comandos abstratos do subsistema de E/S e traduzi-los para comandos que o controlador possa entender e executar. Além disso, o driver pode realizar outras funções, como a inicialização do dispositivo e seu gerenciamento.

Por exemplo, na leitura síncrona de um dado em disco o driver recebe a solicitação de leitura de um determinado bloco e informa ao controlador o disco, cilindro, trilha e setor em que o bloco se localiza, iniciando, dessa forma, a operação (Fig. 12.3). Enquanto se realiza a leitura, o processo que solicitou a operação é colocado no estado de espera até que o controlador avise a UCP do término da operação através de uma interrupção que, por sua vez, ativa novamente o device driver. Após verificar a inexistência de erros, o device driver transfere as informações para a camada superior. Com os dados disponíveis, o processo pode ser retirado do estado de espera e retornar ao estado de pronto para continuar seu processamento.

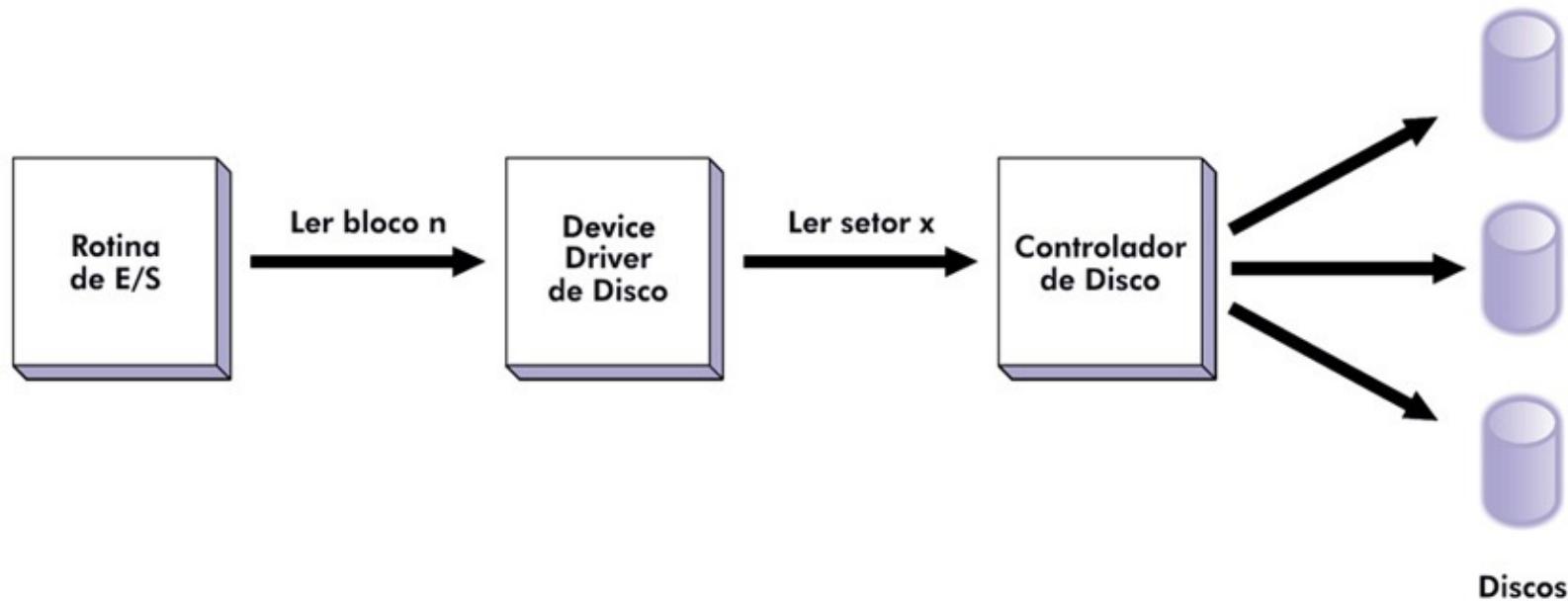


Fig. 12.3 Driver de disco.

Os device drivers fazem parte do núcleo do sistema operacional, sendo escritos geralmente em Assembly. Como os drivers são códigos reentrantes que executam em modo kernel, qualquer erro de programação pode comprometer por completo o funcionamento do sistema. Por isso, um device driver deve ser cuidadosamente desenvolvido e testado.

Devido ao alto grau de dependência entre os drivers e o restante do núcleo do sistema, os fabricantes desenvolvem, para um mesmo dispositivo, diferentes device drivers, um para cada sistema operacional. Sempre que um novo dispositivo é instalado, o driver do dispositivo deve ser adicionado ao núcleo do sistema. Nos sistemas mais antigos, a inclusão de um novo driver significava a recompilação do kernel, uma operação complexa e que exigia a reinicialização do sistema. Atualmente, alguns sistemas permitem a fácil instalação de novos drivers sem a necessidade de reinicialização.

► 12.4 Controlador de Entrada e Saída

Os controladores de E/S são componentes de hardware responsáveis por manipular diretamente os dispositivos de E/S. O sistema operacional, mais exatamente o device driver, comunica-se com os dispositivos através dos controladores (Fig. 12.4). Em geral, o controlador pode ser uma placa independente conectada a um slot do computador ou implementado na mesma placa do processador.

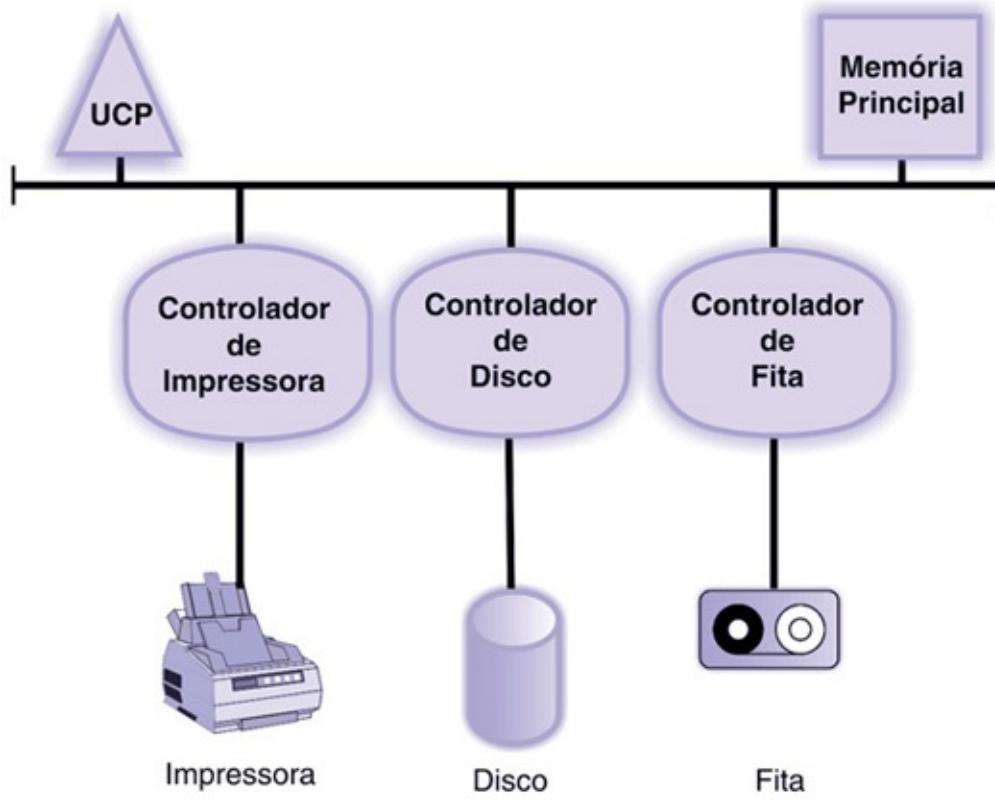


Fig. 12.4 UCP, memória e controladores.

O controlador possui memória e registradores próprios utilizados na execução de instruções enviadas pelo device driver. Essas instruções de baixo nível são responsáveis pela comunicação entre o controlador e o dispositivo de E/S. Em operações de leitura, o controlador deve armazenar em seu buffer interno uma sequência de bits proveniente do dispositivo até formar um bloco. Após verificar a ocorrência de erros, o bloco pode ser transferido para um buffer de E/S na memória principal. A transferência do bloco do buffer interno do controlador para o buffer de E/S da memória principal pode ser realizada pela UCP ou por um controlador de DMA. O uso da técnica de DMA evita que o processador fique ocupado com a transferência do bloco para a memória. O controlador de DMA é um dispositivo de hardware que pode fazer parte do controlador ou ser um dispositivo independente.

De forma simplificada, uma operação de leitura em disco utilizando DMA teria os seguintes passos. A UCP, através do device driver, inicializa os registradores do controlador de DMA e, a partir deste ponto, fica livre para realizar outras atividades. O controlador de DMA, por sua vez, solicita ao controlador de disco a transferência do bloco do disco para o seu buffer interno. Terminada a transferência, o controlador de disco verifica a existência de erros e, caso não haja erros, o controlador de DMA transfere o bloco para o buffer de E/S na memória principal. Ao término da transferência, o controlador de DMA gera uma interrupção avisando ao processador que o dado já se encontra na memória principal (Fig. 12.5).

Alguns controladores, particularmente os de discos, implementam técnicas de cache semelhantes às implementadas pelos sistemas de arquivos, na tentativa de melhorar o desempenho das operações de E/S. Normalmente, o controlador avisa ao sistema operacional do término de uma operação de gravação, quando os dados no buffer do controlador são gravados no disco (*write-through caching*). O controlador também pode ser configurado para avisar do término da gravação, mesmo quando os dados ainda se encontram no buffer do controlador e a operação de gravação no disco não foi realizada (*write-back*).

caching). Desta forma é possível obter ganhos consideráveis de desempenho.

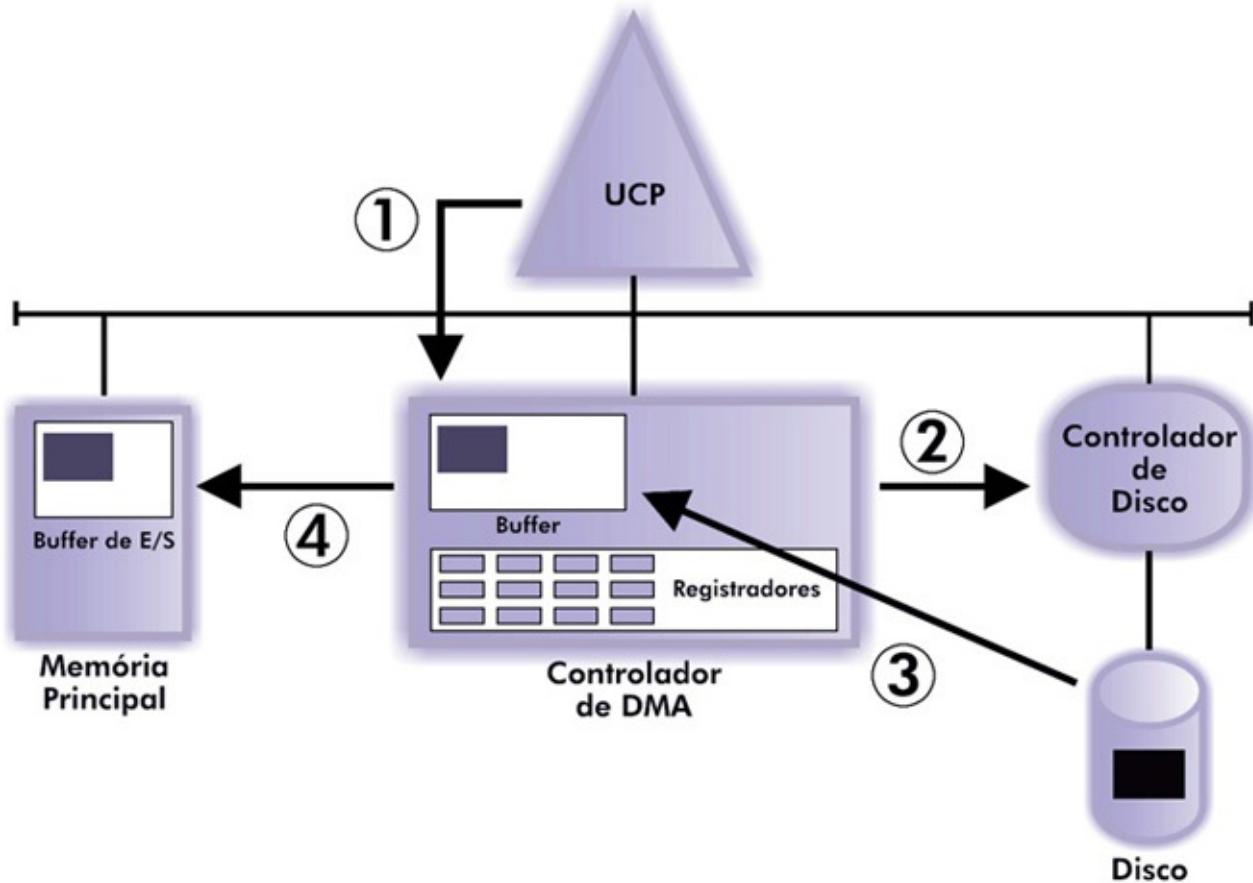


Fig. 12.5 Técnica de DMA.

O padrão mais popular para a conexão de dispositivos a um computador é o SCSI (Small Computer Systems Interface). Inicialmente utilizado em estações de trabalho RISC, o SCSI pode ser encontrado hoje em sistemas de computação de todos os portes, desde computadores pessoais até servidores de grande porte. O SCSI define padrões de hardware e software que permitem conectar ao sistema computacional dispositivos de fabricantes diferentes, como discos, CD-ROMs, scanners e unidades de fita, desde que sigam o padrão estabelecido. Para que isso seja possível, deve-se configurar o sistema operacional com um driver SCSI e o hardware com um controlador SCSI, onde os periféricos são conectados (Fig. 12.6).

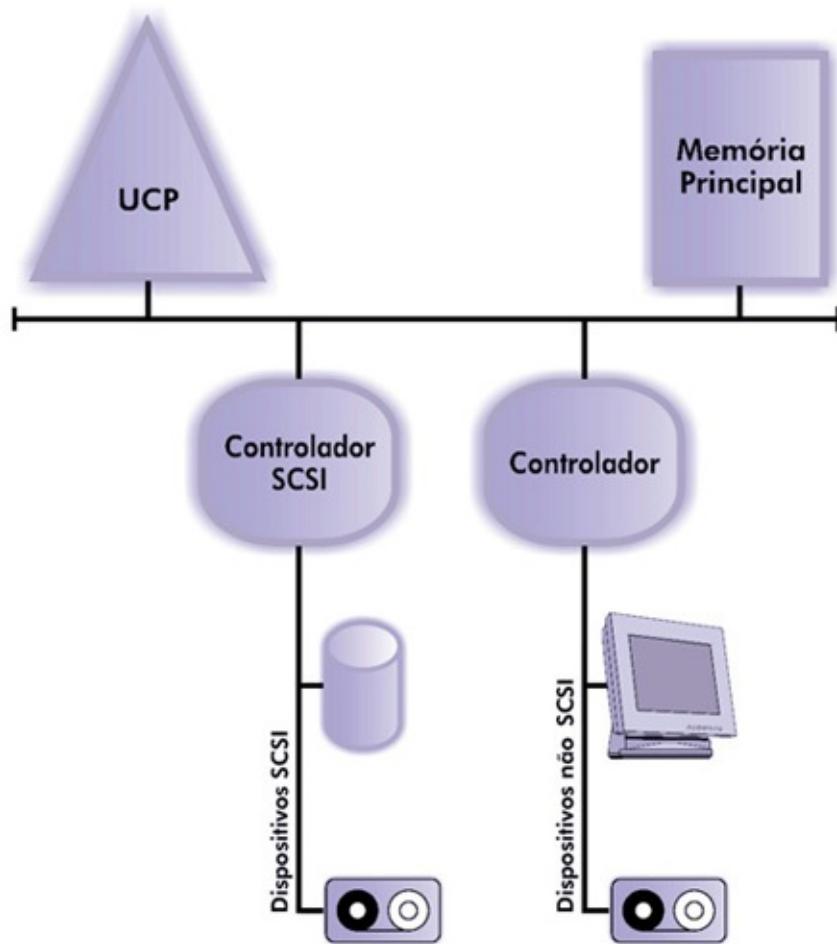


Fig. 12.6 Controlador SCSI.

► 12.5 Dispositivos de Entrada e Saída

Os dispositivos de entrada e saída são utilizados para permitir a comunicação entre o sistema computacional e o mundo externo. Os dispositivos de E/S podem ser classificados como de entrada de dados, como CD-ROM, teclado e mouse, ou de saída de dados, como impressoras. Também é possível que um dispositivo realize tanto entrada quanto saída de dados, como modems, discos e CD-RW.

A transferência de dados pode ocorrer através de blocos de informação ou caracteres, por meio dos controladores dos dispositivos. Em função da forma com que os dados são armazenados os dispositivos de E/S podem ser classificados em duas categorias: dispositivos estruturados e dispositivos não estruturados. Os dispositivos estruturados (*block devices*) caracterizam-se por armazenar informações em blocos de tamanho fixo, possuindo cada qual um endereço que pode ser lido ou gravado de forma independente dos demais. Discos magnéticos e ópticos são exemplos de dispositivos estruturados.

Os dispositivos estruturados classificam-se em dispositivos de acesso direto e sequencial, em função da forma com que os blocos são acessados. Um dispositivo é classificado como de acesso direto quando um bloco pode ser recuperado diretamente através de um endereço. O disco magnético é o melhor exemplo para esse tipo de dispositivo. Um dispositivo é do tipo de acesso sequencial quando, para se acessar um bloco, o dispositivo deve percorrer sequencialmente os demais blocos até encontrá-lo. A fita magnética é exemplo deste tipo de acesso.

Os dispositivos não estruturados (*character devices*) são aqueles que enviam ou recebem uma sequência de caracteres sem estar estruturada no formato de um bloco. Desse modo, a sequência de caracteres não é endereçável, não permitindo operações de acesso direto ao dado. Dispositivos como terminais, impressoras e interfaces de rede são exemplos de dispositivos não estruturados.

► 12.6 Discos Magnéticos

Entre os diversos dispositivos de E/S, os discos magnéticos merecem atenção especial, por serem o principal repositório de dados utilizado pela maioria das aplicações e pelo próprio sistema operacional. Fatores como desempenho e segurança devem ser considerados na arquitetura de discos magnéticos.

Na realidade, um disco magnético é constituído por vários discos sobrepostos, unidos por um mesmo eixo vertical, girando a uma velocidade constante. Cada disco é composto por trilhas concêntricas, que por sua vez são divididas em setores. As trilhas dos diferentes discos que ocupam a mesma posição vertical formam um cilindro. Para a superfície de cada disco existe um mecanismo de leitura/gravação. Todos os mecanismos de leitura/gravação são conectados a um braço que se movimenta entre os vários cilindros dos discos no sentido radial (Fig. 12.7).

O tempo utilizado para leitura e gravação de um bloco de dados em um disco é função de três fatores: tempos de seek, de latência rotacional e de transferência. O tempo de seek é o tempo gasto no posicionamento do mecanismo de leitura e gravação até o cilindro onde o bloco se encontra. O tempo de latência rotacional é o tempo de espera até que o setor desejado se posicione sob o mecanismo de leitura/gravação. O tempo de transferência corresponde ao tempo necessário para a transferência do bloco entre memória principal e o setor do disco (Fig. 12.8).

Como todos esses fatores envolvem aspectos mecânicos, o tempo total das operações de E/S é extremamente longo, se comparado ao número de instruções que o processador pode executar no mesmo intervalo de tempo. Para a maioria dos discos magnéticos, o tempo de seek é o fator de maior impacto no acesso a seus dados.

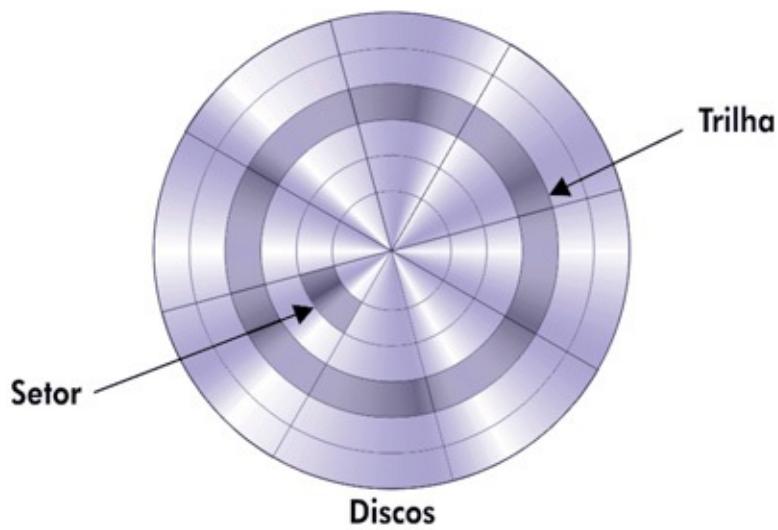
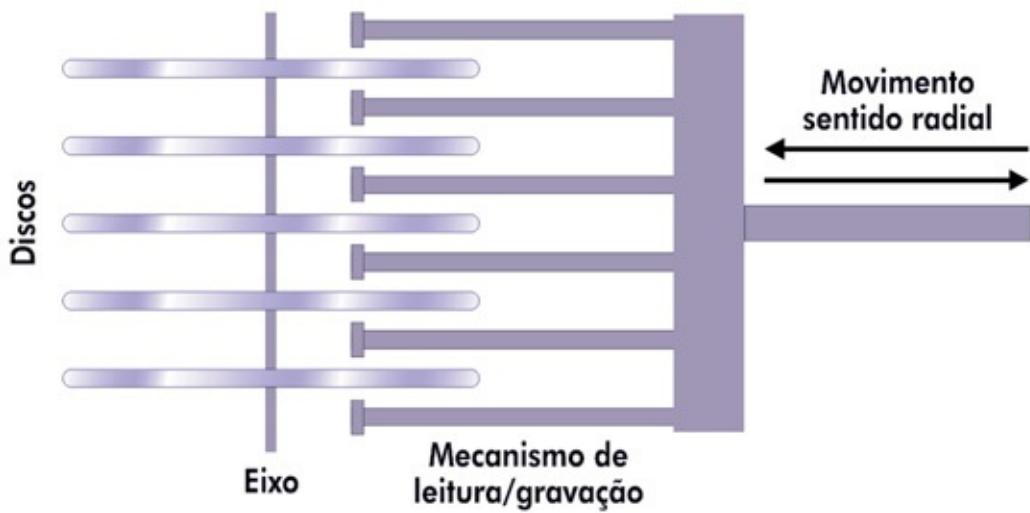


Fig. 12.7 Estrutura de um disco magnético.

Uma forma de eliminar parcialmente os tempos de seek e latência é copiar parte dos dados do disco para a memória principal, utilizando uma área conhecida como disco RAM. Alguns sistemas gerenciadores de banco de dados (SGBD) implementam essa técnica com vistas a aumentar o desempenho no acesso a grandes bases de dados.

12.6.1 Desempenho, Redundância e Proteção de Dados

No final da década de 1980, pesquisadores da Universidade da Califórnia em Berkeley desenvolveram técnicas de gerenciamento de discos que otimizavam as operações de E/S e implementavam redundância e proteção de dados conhecidas como RAID (Redundant Arrays of Inexpensive Disk). As diferentes técnicas, utilizando múltiplos discos, foram publicadas em seis níveis (RAID 1-6). Estas técnicas tiveram grande aceitação no mercado e, posteriormente, um novo nível foi introduzido e denominado RAID 0.

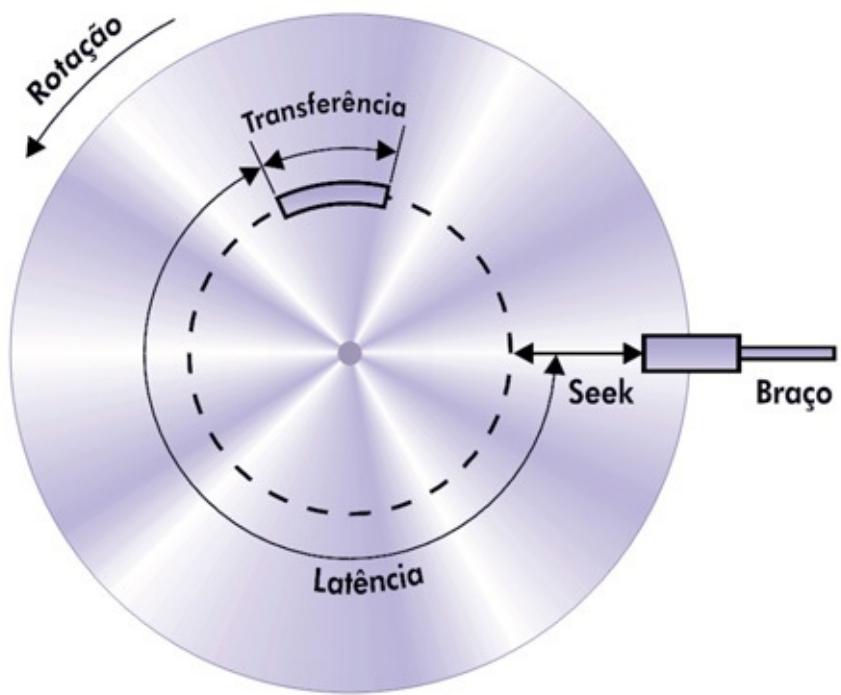


Fig. 12.8 Tempo de acesso.

As técnicas de RAID podem ser implementadas diretamente nos controladores de discos, conhecidos como subsistema RAID externo, ou por software, através do sistema operacional ou um produto gerenciador de discos, denominado subsistema JBOD (Just a Bunch of Disks). A Fig. 12.9 ilustra a diferença entre as duas possíveis formas de implantação das técnicas de RAID.

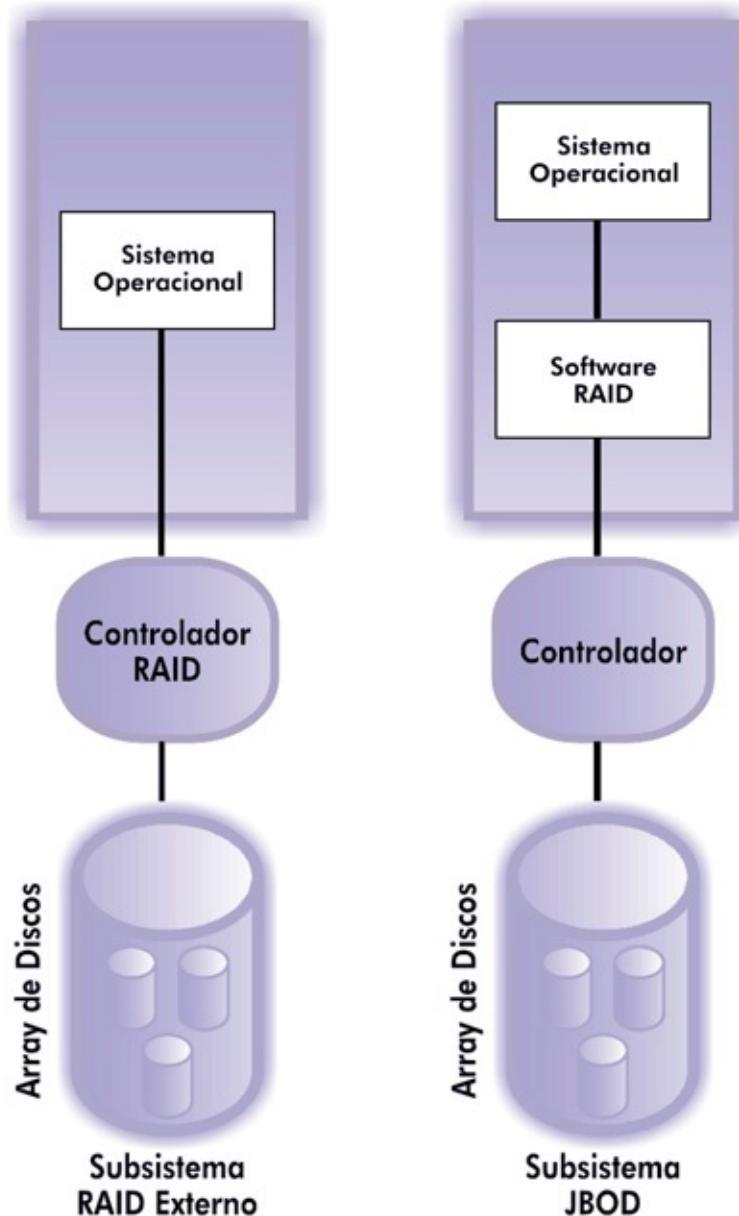


Fig. 12.9 Subsistema de discos.

Uma característica fundamental na técnica de RAID é a criação de um dispositivo virtual conhecido como array de discos. Este array consiste em um grupo de discos físicos que são tratados pelo sistema operacional como se fossem um único disco. Este disco tem como características uma grande capacidade de armazenamento, alto desempenho e confiabilidade nos dados armazenados. A seguir são apresentadas as técnicas de RAID 0, RAID 1 e RAID 5, por sua ampla utilização em diversos sistemas computacionais:

■ RAID 0: Striping

A técnica de RAID 0, também conhecida como striping, consiste na implementação do chamado disk striping, que é distribuir as operações de E/S entre os diversos discos físicos contidos no array com o intuito de otimizar o desempenho. Como os dados são divididos entre os diversos discos, as operações de E/S podem ser processadas paralelamente.

Para poder implementar o striping é preciso formar um conjunto de discos chamado de stripe set, onde cada disco é dividido em pedaços (stripes). Sempre que um arquivo é gravado, seus dados são

divididos em pedaços iguais e espalhados simultaneamente pelos stripes dos diversos discos (Fig. 12.10).

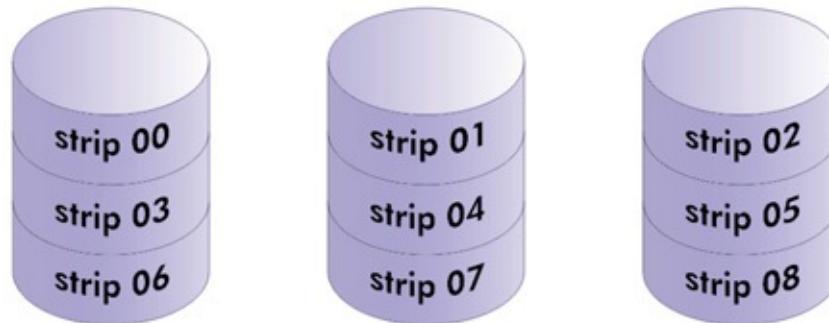


Fig. 12.10 RAID 0.

Apesar da denominação RAID, esta técnica não implementa nenhum tipo de redundância, só sendo vantajosa no ganho de desempenho das operações de E/S. Caso haja uma falha em qualquer disco do stripe set, os dados serão perdidos. Aplicações multimídia são beneficiadas com o uso desta técnica, pois necessitam de alto desempenho nas operações com discos.

■ RAID 1: Espelhamento

A técnica de RAID 1, também conhecida como espelhamento ou mirroring, consiste em replicar todo o conteúdo do disco principal, chamado primário, em um ou mais discos denominados espelhos ou secundários. A redundância oferecida por essa técnica garante que, no caso de falha no disco principal os discos-espelhos sejam utilizados de forma transparente pelo sistema de arquivos (Fig. 12.11).

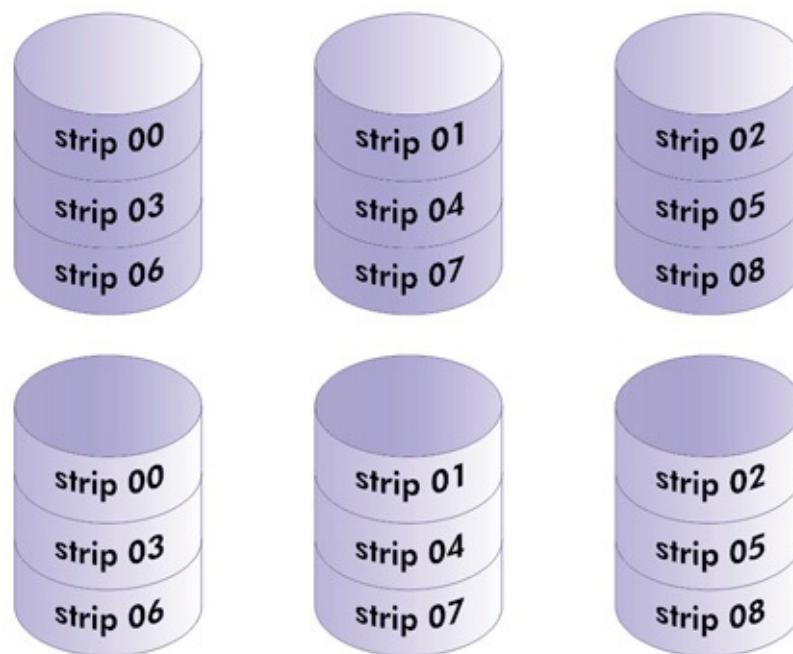


Fig. 12.11 RAID 1.

Para que a sincronização entre o conteúdo do disco principal e dos discos-espelhos ocorra, toda

operação de escrita realizada no disco primário é replicada para os discos-espelhos. O overhead adicional exigido nesta operação é pequeno, e o benefício da proteção justifica a implementação. Apesar da vantagem proporcionada pela redundância oferecida por esta técnica, a capacidade útil do subsistema de discos com a implementação do RAID 1 é de apenas 50%.

A técnica de RAID 1 pode ser implementada por software em um subsistema JBOD ou por hardware diretamente pelo controlador de disco em um subsistema RAID externo. A implementação por software necessita que o sistema operacional ou algum produto gerenciador de discos ofereça esta facilidade.

■ RAID 5: Acesso Independente com Paridade Distribuída

A técnica de RAID 5 consiste em distribuir os dados entre os discos do array e implementar redundância baseada em paridade. Este mecanismo de redundância é implementado através de cálculos do valor da paridade dos dados, que são armazenados nos discos do array junto com os dados (Fig. 12.12).



Fig. 12.12 RAID 5.

Caso haja uma falha em qualquer um dos discos do array, os dados podem ser recuperados por um algoritmo de reconstrução que utiliza as informações de paridade dos demais discos. Esta recuperação ocorre automaticamente e é transparente ao sistema de arquivos.

A principal vantagem de uma técnica de redundância que utiliza paridade é que esta requer um espaço adicional menor para armazenar informação de controle que a técnica de espelhamento. A capacidade útil do subsistema de discos com a técnica de RAID 5 implementada é de aproximadamente 80%. Apesar disso, esta técnica de redundância requer um overhead nas operações de gravação no disco em função do cálculo da paridade.

► 12.7 Exercícios

1. Explique o modelo de camadas aplicado na gerência de dispositivos.
2. Qual a principal finalidade das rotinas de E/S?
3. Quais as diferentes formas de um programa chamar rotinas de E/S?
4. Quais as principais funções do subsistema de E/S?

5. Qual a principal função de um device driver?
6. Por que o sistema de E/S deve criar uma interface padronizada com os device drivers?
7. Explique o funcionamento da técnica de DMA e sua principal vantagem.
8. Diferencie os dispositivos de E/S estruturados dos não estruturados.
9. Qual a principal razão de as operações de E/S em fitas e discos magnéticos serem tão lentas se comparadas à velocidade com que o processador executa instruções?
10. O que são técnicas de redundância em discos magnéticos?
11. Diferencie as técnicas RAID 0, RAID 1 e RAID 5, apresentando vantagens e desvantagens.

13

Sistemas com Múltiplos Processadores

► 13.1 Introdução

Sistemas com múltiplos processadores são arquiteturas que possuem duas ou mais UCPs interligadas e que funcionam em conjunto na execução de tarefas independentes ou no processamento simultâneo de uma mesma tarefa. Inicialmente, os computadores eram vistos como máquinas sequenciais, em que o processador executava as instruções de um programa uma de cada vez. Com a implementação de sistemas com múltiplos processadores, o conceito de paralelismo pôde ser expandido a um nível mais amplo.

A evolução desses sistemas se deve, em grande parte, ao elevado custo de desenvolvimento de processadores mais rápidos. Em função disso, passou-se a dar ênfase a computadores com múltiplos processadores, em vez de arquiteturas com um único processador de alto desempenho. Outra motivação são aplicações que necessitam de grande poder computacional, como sistemas de previsão do tempo, dinâmica dos fluidos, genoma humano, modelagens e simulações. Com múltiplos processadores, é possível reduzir drasticamente o tempo de processamento destas aplicações. Inicialmente, as configurações limitavam-se a poucos processadores, mas atualmente existem sistemas com milhares de processadores.

Os primeiros sistemas com múltiplos processadores surgiram na década de 1960, com o objetivo principal de acelerar a execução de aplicações que lidavam com um grande volume de cálculos. Por muito tempo, esses sistemas foram utilizados quase que exclusivamente em ambientes acadêmicos e centros de pesquisas voltados para o processamento científico. O mercado corporativo começou realmente a utilizar os sistemas com múltiplos processadores na década de 1980, para melhorar o desempenho de suas aplicações comerciais e reduzir o tempo de resposta dos usuários interativos. Posteriormente, as empresas reconheceram também nesse tipo de sistema uma maneira de aumentar a confiabilidade, a escalabilidade e a disponibilidade, além da possibilidade do balanceamento de carga de suas aplicações. Atualmente, a maioria dos servidores de banco de dados, servidores de arquivos e servidores Web utiliza sistemas com múltiplos processadores. Além disso, sistemas com múltiplos processadores estão sendo utilizados em estações de trabalho e, até mesmo, em computadores pessoais. Isso foi possível devido à redução de custo dessas arquiteturas e à evolução dos sistemas operacionais,

que passaram a oferecer suporte a múltiplos processadores.

Este capítulo apresenta as vantagens e desvantagens de sistemas com múltiplos processadores, os tipos de sistemas, suas classificações e características.

► 13.2 Vantagens e Desvantagens

Existem inúmeras vantagens em sistemas com múltiplos processadores, como desempenho, escalabilidade, relação custo/desempenho, tolerância a falhas, disponibilidade e balanceamento de carga. A seguir, são apresentadas as principais vantagens:

■ Desempenho

A princípio, sempre que novos processadores são adicionados à arquitetura de uma máquina, melhor é o *desempenho* do sistema. Apesar de esta relação não ser linear, o aumento de desempenho pode ser observado pelo maior throughput do sistema, pela redução no tempo de resposta e pelo menor tempo de processamento das aplicações.

O desempenho pode ser formalmente medido utilizando os padrões especificados pelo consórcio Transaction Processing Performance Council (TPC), criado em 1988 e formado por inúmeras empresas. O TPC oferece três benchmarks (TPC-A, TPC-B e TPC-C) que podem ser aplicados para determinar o desempenho comparativo de diversos sistemas, além da relação custo/desempenho.

O ganho de desempenho com múltiplos processadores pode ser obtido em dois níveis. No primeiro nível, múltiplos processadores permitem a execução simultânea de diversas tarefas independentes, aumentando o throughput do sistema. Servidores de banco de dados e servidores Web são bons exemplos de ambientes onde o aumento do número de processadores permite atender um número maior de usuários simultaneamente.

No segundo nível, múltiplos processadores permitem a execução de uma mesma tarefa por vários processadores simultaneamente (processamento paralelo). Neste caso, o ganho de desempenho dependerá de diferentes fatores, como a organização dos processadores, a linguagem de programação utilizada e o grau de paralelismo possível na aplicação. O processamento paralelo é mais difícil de ser implementado, e apenas algumas aplicações oferecem ganhos reais com aumento do número de processadores (aplicação concorrente).

■ Escalabilidade

Escalabilidade é a capacidade de adicionar novos processadores ao hardware do sistema. Em ambientes que permitam apenas um processador, para aumentar a capacidade computacional é necessário substituir a UCP por uma outra com maior poder de processamento. Com múltiplos processadores, é possível ampliar a capacidade de computação apenas adicionando-se novos processadores, com um custo inferior à aquisição de um outro sistema com maior desempenho.

• Relação custo/desempenho

Sistemas com um único processador, por mais poderosos que sejam, apresentam limitações de desempenho inerentes à sua arquitetura, devido às limitações existentes na comunicação da UCP com as demais unidades funcionais, principalmente a memória principal. Além disso, o custo do desenvolvimento de um processador que ofereça desempenho semelhante a um sistema com múltiplos processadores é muito elevado.

Sistemas com múltiplos processadores permitem utilizar UCPs convencionais de baixo custo, interligadas às unidades funcionais através de mecanismos de interconexão. Desta forma é possível oferecer sistemas de alto desempenho com custo aceitável.

■ Tolerância a falhas e disponibilidade

Tolerância a falhas é a capacidade de manter o sistema em operação mesmo em casos de falha em algum componente. Nesta situação, se um dos processadores falhar, os demais podem assumir suas funções de maneira transparente aos usuários e suas aplicações, embora com menor capacidade computacional.

A *disponibilidade* é medida em número de minutos por ano que o sistema permanece em funcionamento de forma ininterrupta, incluindo possíveis falhas de hardware ou software, manutenções preventivas e corretivas. Na Tabela 13.1 são apresentados diversos níveis de disponibilidade em função do tempo de indisponibilidade do sistema (downtime), considerando uma operação de 24 horas por sete dias.

Tabela 13.1 Níveis de disponibilidade

Disponibilidade	Downtime em minutos	Impacto ao usuário
90%	50.000	Mais de um mês
99%	5.000	Menos de 4 dias
99,9%	500	Menos de 9 horas
99,99%	50	Cerca de 1 hora
99,999%	5	Pouco mais de 5 minutos
99,9999%	0,5	Cerca de 30 segundos
99,99999%	0,05	Cerca de 3 segundos
100%	0	0

Uma alta disponibilidade é obtida com sistemas com maior tolerância a falhas. Sistemas de alta disponibilidade são utilizados em aplicações de missão crítica, como sistemas de tráfego aéreo e de comércio eletrônico na Internet.

■ Balanceamento de carga

Balanceamento de carga é a distribuição do processamento entre os diversos componentes da configuração, a partir da carga de cada processador, melhorando, assim, o desempenho do sistema como um todo. Servidores de banco de dados, que oferecem esse tipo de facilidade, permitem que as solicitações dos diversos usuários sejam distribuídas entre os vários processadores disponíveis.

Apesar das inúmeras vantagens, sistemas com múltiplos processadores também possuem desvantagens. Com múltiplos processadores, novos problemas de comunicação e sincronização são introduzidos, pois vários processadores podem estar acessando as mesmas posições de memória. Além disso, existe o problema de organizar os processadores, memórias e periféricos de uma forma eficiente, que permita uma relação custo/desempenho aceitável. Dependendo do tipo de sistema, a tolerância contra falhas é dependente do sistema operacional e não apenas do hardware, sendo difícil de ser implementada.

► 13.3 Tipos de Sistemas Computacionais

Os sistemas computacionais podem ser classificados conforme o grau de paralelismo existente no processamento de suas instruções e dados. O modelo proposto por Flynn (1966) define quatro tipos de sistema e, apesar de antigo, continua sendo amplamente utilizado para comparar e compreender a evolução das arquiteturas de computadores.

■ SISD (Single Instruction Single Data)

Esta arquitetura engloba os sistemas que suportam uma única sequência de instruções e apenas uma sequência de dados. A grande maioria dos computadores com apenas um processador se enquadra nesta categoria, desde computadores pessoais até servidores de grande porte. Mesmo em sistemas com um único processador é possível encontrar algum nível de paralelismo, como na utilização da técnica de pipeline.

■ SIMD (Single Instruction Multiple Data)

O segundo tipo trata de sistemas com uma única sequência de instruções e múltiplas sequências de dados. Os computadores SISD trabalham com dados escalares e, portanto, processam vetores sequencialmente, ou seja, um componente de cada vez. Computadores SIMD permitem a manipulação de vetores inteiros simultaneamente, possibilitando a execução de uma mesma instrução sobre diferentes elementos de um ou mais vetores.

Vejamos o exemplo da soma de dois vetores, A e B, em que o resultado é atribuído ao vetor C. Em uma linguagem de alto nível, a operação poderia ser implementada conforme o loop descrito no trecho de código a seguir. Em uma arquitetura SISD, a soma dos vetores A e B corresponderia ao ciclo de busca da instrução, ao cálculo dos endereços dos operandos e à execução das instruções de controle da repetição (FOR) para cada soma dos N elementos dos vetores. Para a mesma situação, um computador SIMD utiliza uma única instrução, em que o vetor A é somado ao vetor B e o resultado atribuído ao vetor C. A grande vantagem dessa implementação é que somente uma instrução é executada, ocorrendo apenas um ciclo de busca e execução.

```
FOR i := 1 TO N DO  
C[i] := A[i] + B[i];
```

Arquiteturas SIMD são vantajosas apenas quando as aplicações executadas têm um elevado grau de *parallelismo de dados*, como problemas numéricos, processamento de imagens, estudos meteorológicos e de física nuclear. Para processar esses tipos de problemas, que realizam repetidas operações aritméticas de ponto flutuante em grandes vetores, foram desenvolvidos os chamados supercomputadores ou computadores vetoriais. Essas máquinas são capazes de processar, em segundos, o que um computador SISD levaria horas ou mesmo dias. O custo do seu alto desempenho é sua organização complexa e preços muito superiores aos dos computadores convencionais. Um exemplo de computador vetorial é o Cray T90, que possui oito registradores vetoriais, com 128 elementos por vetor, e máximo de 32 processadores.

■ MISD (Multiple Instruction Single Data)

A terceira categoria permite múltiplas sequências de instruções e uma única sequência de dados. Não existe, até o momento, nenhum computador desenvolvido com esta arquitetura.

■ MIMD (Multiple Instruction Multiple Data)

A quarta categoria trata múltiplas sequências de instruções e múltiplas sequências de dados, nas quais se enquadram os sistemas com múltiplos processadores.

► 13.4 Sistemas Fortemente e Fracamente Acoplados

As arquiteturas MIMD podem ser classificadas em função de diversos fatores, como compartilhamento da memória principal, distância física entre processadores, tempo de acesso à memória principal, mecanismos de comunicação e sincronização utilizados e velocidade de comunicação dos processadores. A partir desses fatores, é possível determinar o grau de acoplamento de um sistema, permitindo que arquiteturas MIMD sejam classificadas em sistemas fortemente acoplados ou fracamente acoplados.

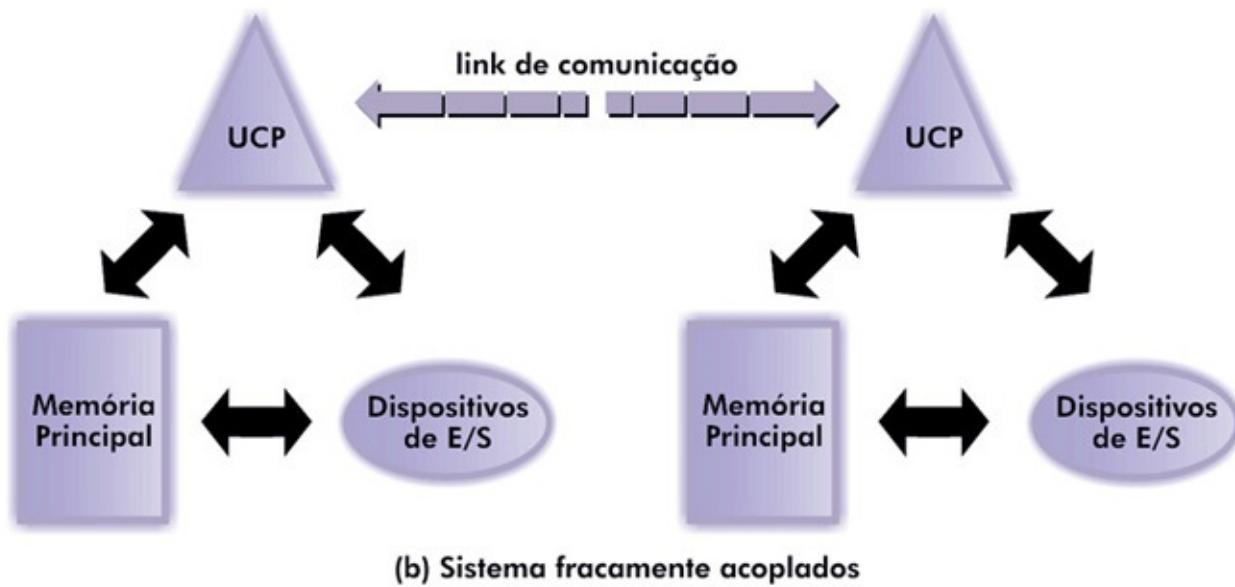
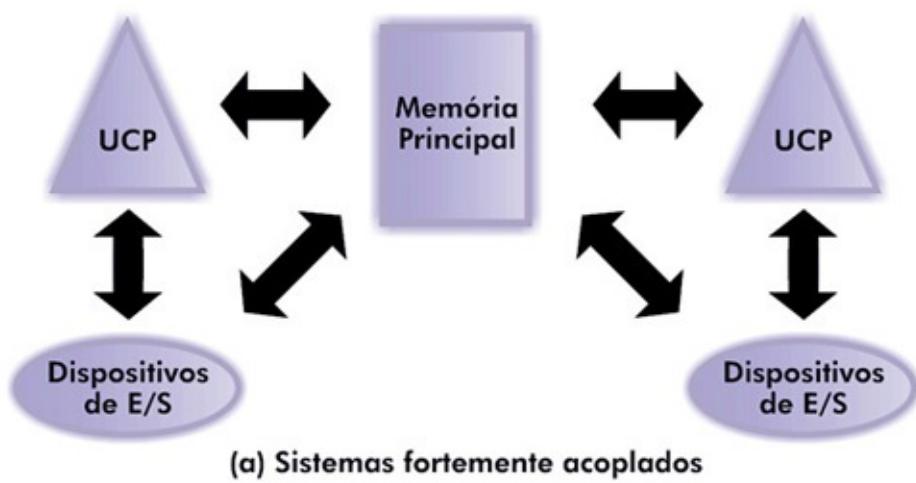


Fig. 13.1 Sistemas fortemente e fracamente acoplados.

Nos *sistemas fortemente acoplados*, os processadores compartilham uma única memória principal e são controlados por apenas um único sistema operacional. Os *sistemas fracamente acoplados* caracterizam-se por possuir dois ou mais sistemas computacionais independentes, conectados por uma rede de comunicação, tendo cada sistema seus próprios processadores, memória principal, dispositivos E/S e sistema operacional (Fig. 13.1).

A grande diferença entre os dois tipos de sistemas é que em sistemas fortemente acoplados existe apenas um espaço de endereçamento compartilhado por todos os processadores, também chamado de *memória compartilhada*. A comunicação entre os processadores é feita através de variáveis na memória principal, utilizando operações de leitura e escrita. Nos sistemas fracamente acoplados, cada sistema tem seu próprio espaço de endereçamento individual e a comunicação entre os sistemas é realizada através de mecanismos de troca de mensagens, utilizando operações de send e receive. A Tabela 13.2 apresenta diversas características encontradas em sistemas fortemente e fracamente acoplados e compara suas principais diferenças.

Outra maneira de subdividir as arquiteturas MIMD foi proposta por Bell (1985), considerando apenas o grau de acoplamento da memória principal. A subdivisão proposta separa arquiteturas que compartilham um único espaço de endereçamento, chamadas *multiprocessadores*, das que possuem seu próprio espaço

de endereçamento individual, chamadas *multicomputadores*. De certa forma, podemos relacionar os sistemas fortemente acoplados aos multiprocessadores e os sistemas fracamente acoplados aos multicomputadores.

A Fig. 13.2 apresenta uma subdivisão para os sistemas fortemente acoplados e fracamente acoplados, para facilitar a identificação dos sistemas intermediários.

Tabela 13.2 Sistemas fortemente acoplados × sistemas fracamente acoplados

Características	Sistemas fortemente acoplados	Sistemas fracamente acoplados
Espaço de endereçamento	Centralizado	Distribuído
Comunicação entre os processadores	Muito rápida	Lenta
Arquitetura de hardware	Complexa	Mais simples
Sistema operacional	Homogêneo	Geralmente heterogêneo
Cópias do sistema operacional	Existe apenas uma cópia	Existem várias cópias, uma para cada sistema
Programação paralela	Mais fácil	Mais difícil
Número de processadores	Centenas	Não existe limite
Escalabilidade	Baixa	Alta
Disponibilidade	Baixa/Média	Média/Alta
Administração	Simples	Complexa
Custo de software	Menor	Maior
Intercomunicação	Proprietária	Padronizada
Segurança	Centralizada	Distribuída

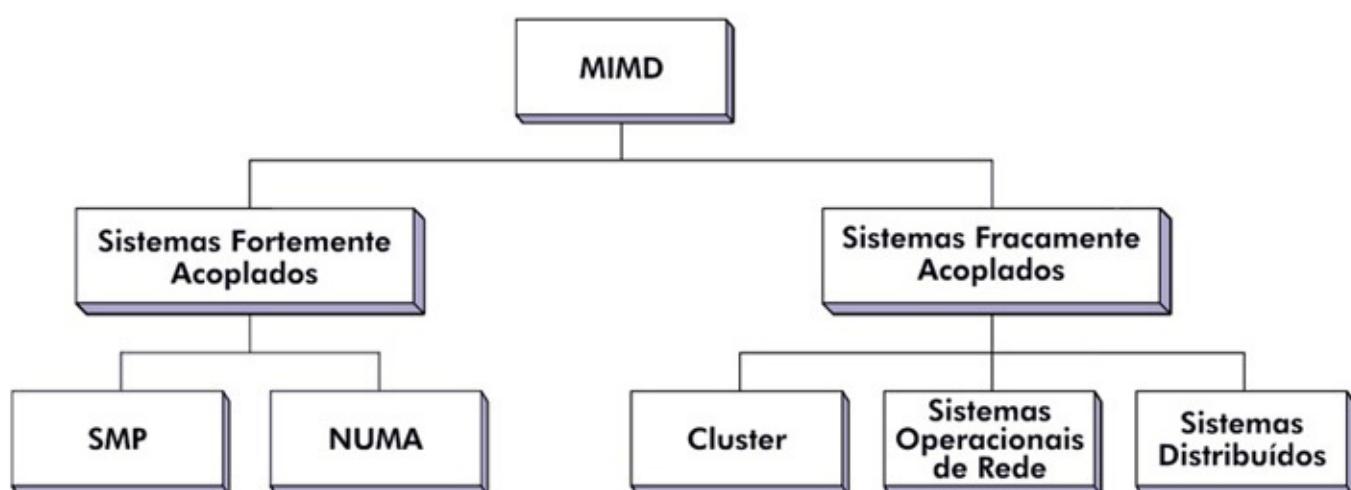


Fig. 13.2 Sistemas com múltiplos processadores.

► 13.5 Sistemas com Multiprocessadores Simétricos

Os sistemas com multiprocessadores simétricos (*Symmetric Multiprocessors — SMP*) possuem dois ou mais processadores compartilhando um único espaço de endereçamento e gerenciados por apenas um sistema operacional.

Uma característica importante nos sistemas SMP é que o tempo de acesso à memória principal pelos vários processadores é uniforme, independentemente da localização física do processador. Este tipo de arquitetura também é conhecido como *UMA (Uniform Memory Access)*.

Inicialmente, os sistemas simétricos estavam limitados aos ambientes de grande porte, restritos aos centros de pesquisas e às grandes corporações. Com a evolução dos computadores pessoais e das estações de trabalho, os sistemas operacionais para esses tipos de ambiente passaram a suportar vários multiprocessadores simétricos.

13.5.1 Evolução dos Sistemas Simétricos

Os sistemas *SMP* são uma evolução dos sistemas com múltiplos processadores assimétricos. Na organização *assimétrica* ou *mestre/escravo* somente um processador, denominado mestre, pode executar serviços do sistema operacional, como, por exemplo, operações de E/S. Sempre que um processador do tipo escravo precisa realizar uma operação de E/S, tem de requisitar o serviço ao processador-mestre. Dependendo do volume de operações de E/S destinadas aos processadores-escravos, o sistema pode se tornar ineficiente, devido ao elevado número de interrupções que deverão ser tratadas pelo mestre.

Outra consequência dessa organização é que, se o processador-mestre falhar, todo o sistema ficará incapaz de continuar o processamento. Neste caso, o sistema deve ser reconfigurado, fazendo um dos processadores-escravos assumir o papel de mestre. Mesmo sendo uma organização simples de implementar e uma extensão dos sistemas multiprogramáveis, esse tipo de sistema não é eficiente devido à assimetria dos processadores, que não realizam as mesmas funções.

Sistemas *SMP*, por outro lado, implementam a simetria dos processadores, ou seja, todos os processadores realizam as mesmas funções. Apenas algumas poucas funções ficam a cargo de um único processador, como, por exemplo, a inicialização do sistema. Os sistemas simétricos são mais poderosos que os assimétricos, permitindo um melhor balanceamento do processamento e das operações de E/S, apesar de sua implementação ser mais complexa. Atualmente, a grande maioria dos sistemas operacionais oferece suporte para *SMP*.

13.5.2 Arquitetura dos Sistemas Simétricos

A organização interna das unidades funcionais (processadores, memória principal e dispositivos de E/S) é fundamental no projeto de sistemas *SMP*, pois determina quantos processadores o sistema poderá

suportar e como será o mecanismo de acesso à memória principal.

A forma mais simples de comunicação entre múltiplos processadores e outras unidades funcionais é interligar todos os componentes a um *barramento único*. O maior problema dessa organização é que somente uma unidade funcional pode estar utilizando o barramento em determinado instante, o que pode produzir um gargalo quando várias unidades tentam acessá-lo simultaneamente. Além disso, caso ocorra algum problema no barramento, todo o sistema fica comprometido (Fig. 13.3).

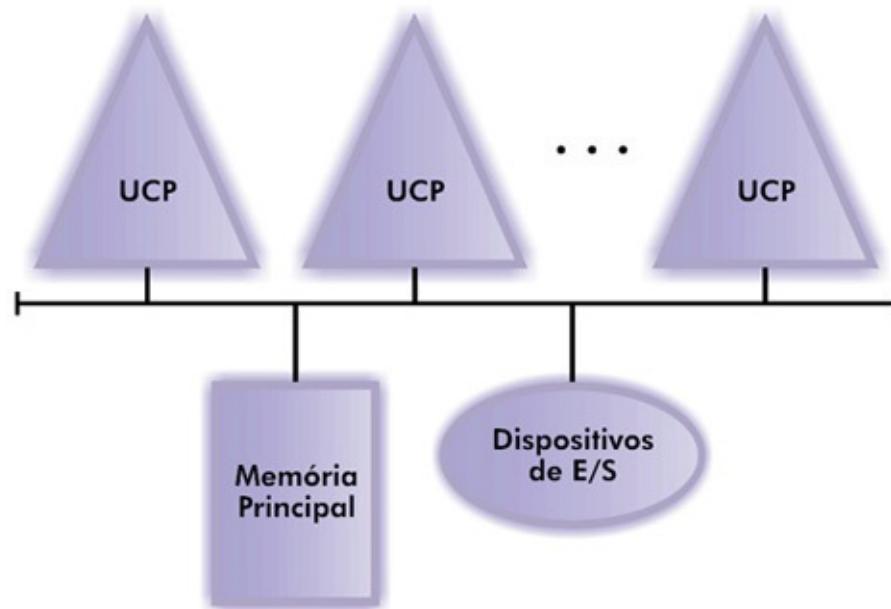


Fig. 13.3 Barramento único.

Apesar de ser uma arquitetura simples, econômica e flexível, sistemas desse tipo estão limitados a poucos processadores, dependendo da velocidade de transmissão do barramento. A maioria das arquiteturas que implementa o barramento único utiliza o esquema de cache para reduzir a latência das operações de acesso à memória principal. Apesar de serem largamente utilizados, caches introduzem um novo problema, conhecido como *coerência de cache*. Nesse esquema, cada processador possui seu cache individual, para leitura e escrita de dados de maneira eficiente. Quando os dados são apenas para leitura, como instruções, não há problema que cada processador tenha sua própria cópia do dado, já que não haverá alteração. O problema aparece quando dois ou mais processadores desejam compartilhar um dado, não apenas para leitura, mas também para escrita, como no caso de uma variável (Fig. 13.4).

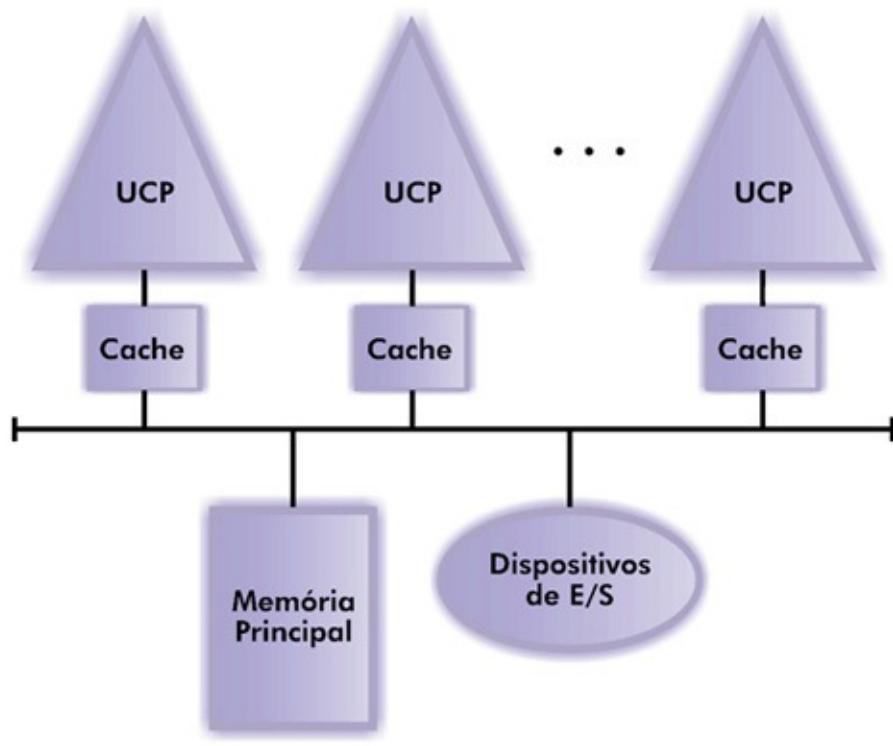


Fig. 13.4 Barramento único com cache.

Suponha que um processador PA leia uma variável X na memória e seu conteúdo seja copiado para o cache. Se o conteúdo da variável X for alterado, o cache terá um valor e a memória principal outro diferente, logo estarão inconsistentes. Caso um processador PB leia a mesma variável, PB terá no cache um valor que não é idêntico ao de PA. Uma solução simples para o problema apresentado é que, sempre que um dado no cache for alterado, a memória principal também seja atualizada. Este esquema é conhecido como *write-through* e, apesar de sua simplicidade, não oferece um bom desempenho, pois incorre em acesso à memória a cada operação de escrita, gerando, consequentemente, tráfego no barramento. Outro esquema amplamente empregado é conhecido como *write-back*, que permite alterar o dado no cache e não alterá-lo imediatamente na memória principal, oferecendo, assim, melhor desempenho, porém não resolvendo o problema da inconsistência.

A técnica mais utilizada para manter a coerência de cache é chamada de *snooping*, sendo implementada em hardware. Cada cache monitora o barramento, verificando se o endereço do dado que trafega no barramento está localmente armazenado. Caso o dado seja acessado apenas para leitura, não há problema de vários caches possuírem suas próprias cópias, mas caso seja um dado que possa ser alterado, a técnica de snooping deve garantir que os processadores obtenham a cópia mais recente.

Existem duas implementações para a técnica de snooping: *write-invalidate* (escreve-invalida) e *write-update* (escreve-atualiza). Na técnica *write-invalidate*, como o nome sugere, sempre que ocorre uma operação de escrita todas as cópias do mesmo dado em outros caches são invalidadas e o novo valor é atualizado na memória principal. Nesse caso, apenas o processador que realizou a operação de escrita possuirá uma cópia válida do dado no cache. Os demais processadores, ao acessarem novamente o dado no cache, terão uma cópia inválida e o dado deverá ser transferido da memória principal para o cache (*cache miss*). Se dois ou mais processadores tentarem realizar a escrita no dado ao mesmo tempo, deve existir um mecanismo que garanta que apenas um processador tenha sucesso na operação e os demais tenham o dado invalidado em seus caches. A Tabela 13.3 apresenta a técnica de *write-invalidate* sendo

aplicada à variável X, inicialmente com o valor zero, e ao cache de dois processadores (PA e PB).

Tabela 13.3 Write-invalidate

Processador	Barramento	Cache do PA	Cache do PB	Memória
PA lê X	Cache miss do PA	0	-	0
PB lê X	Cache miss do PB	0	0	0
PA escreve 1 em X	Invalidação de X	1	-	1
PB lê em X	Cache miss do PB	1	1	1

Na técnica write-update, sempre que ocorre uma operação de escrita a alteração é informada pelo barramento para que todos os processadores que possuam a mesma cópia atualizem seu cache com o novo valor. Esta técnica oferece um desempenho inferior à técnica de write-invalidate, pois além do endereço do dado a ser atualizado deve também informar no barramento o seu novo valor. Por isso, a técnica de write-invalidate é empregada na grande maioria dos sistemas SMP. A Tabela 13.4 apresenta a técnica de write-update sendo aplicada à variável X, inicialmente com o valor zero, e ao cache de dois processadores (PA e PB).

Tabela 13.4 Write-update

Processador	Barramento	Cache do PA	Cache do PB	Memória
PA lê X	Cache miss do PA	0	-	0
PB lê X	Cache miss do PB	0	0	0
PA escreve 1 em X	Atualização de X	1	1	1
PB lê em X		1	1	1

A maioria dos sistemas comerciais com múltiplos processadores adota o esquema de write-back com a técnica de write-invalidate, a fim de reduzir o tráfego no barramento e permitir um número maior de processadores. Apesar de todas as técnicas apresentadas buscarem aumentar o número de processadores, a topologia de barramento único está limitada comercialmente a cerca de 64 UCPs.

Para evitar o problema do gargalo do barramento único, a memória principal pode ser dividida em módulos para permitir múltiplos acessos simultâneos, podendo assim ser compartilhada entre as várias unidades funcionais. As unidades funcionais podem ser conectadas entre si através de um *barramento cruzado comutado (crossbar switch)*, criando uma matriz de interconexão (Fig. 13.5). Nesse esquema, é possível a comunicação simultânea entre diferentes unidades funcionais, ficando a cargo do hardware e do sistema operacional a resolução dos possíveis conflitos de acesso a uma mesma unidade. Um exemplo de máquina que implementa esta arquitetura é o Sun Enterprise 10000, que pode suportar até 64 processadores UltraSPARC. As UCPs são agrupadas em nós de quatro processadores e interligadas através de 16 barramentos cruzados.

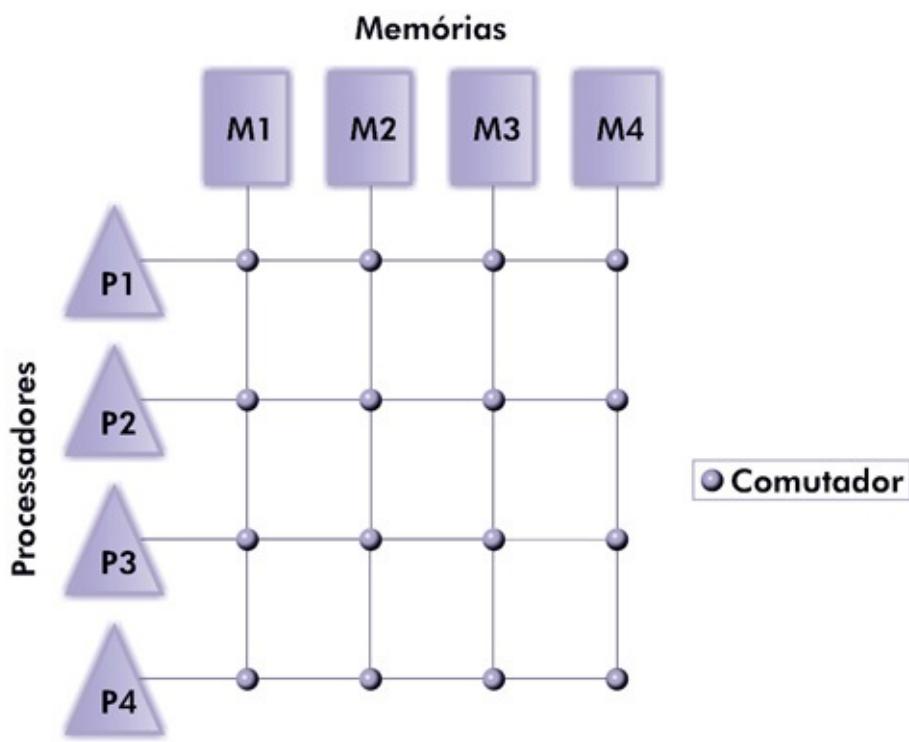


Fig. 13.5 Barramento cruzado comutado.

Em uma arquitetura de barramento cruzado, para cada n processadores e n módulos de memória são necessários n^2 comutadores para interligar todos os pontos. Em uma configuração em que n seja muito grande, o custo tende a ser muito alto.

Uma maneira de reduzir o número de comutadores é implementar uma *rede Ômega*. Nessa topologia, não existe uma ligação exclusiva entre cada processador e um módulo de memória; os caminhos são compartilhados entre as diversas unidades funcionais. Desta forma, é possível que um processador tenha que esperar para poder acessar um determinado módulo de memória devido à falta de um caminho disponível no momento. A grande vantagem dessa arquitetura é o seu menor custo, pois são necessários $(n \log_2 n)/2$ comutadores, número menor que o utilizado no barramento cruzado comutado (Fig. 13.6).

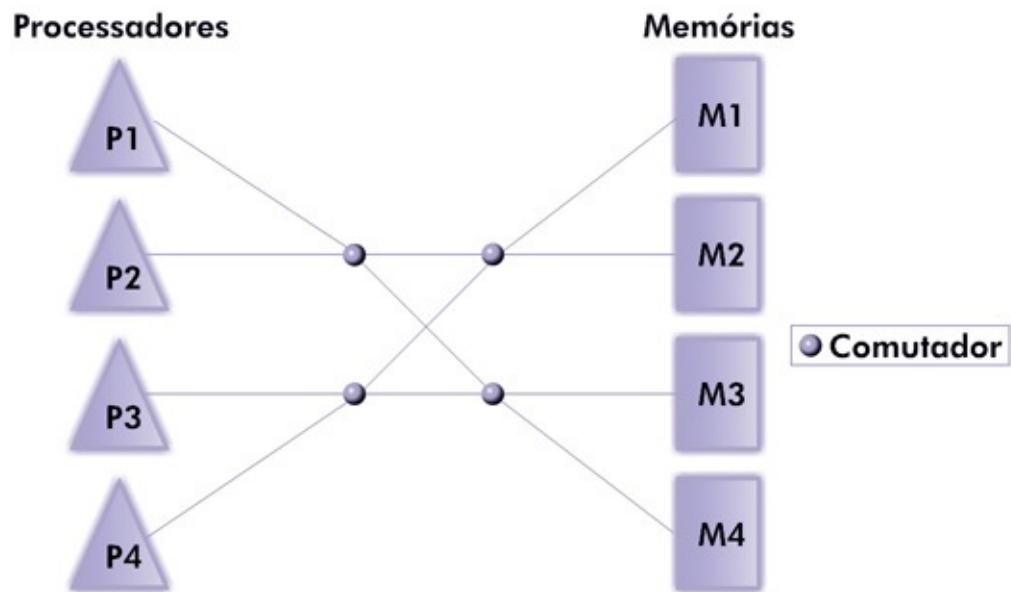


Fig. 13.6 Rede Ômega.

Apesar de a rede Ômega ser implementada com um número menor de comutadores, ainda é possível a qualquer processador acessar todos os módulos de memória utilizando a mesma técnica de comutação das redes de pacotes. Apesar do menor custo, redes Ômega podem introduzir o problema de latência, devido ao longo caminho percorrido entre o ponto de origem e o de destino, principalmente se considerarmos redes de interconexão maiores. Nesse caso, o tempo de acesso à memória poderá ser diferente em função da localização física dos processadores, originando os sistemas NUMA.

► 13.6 Sistemas NUMA

Nos *sistemas NUMA* (*Non-Uniform Memory Access*), o tempo de acesso à memória principal depende da localização física do processador. Nesta arquitetura, existem vários conjuntos, reunindo processadores e memória, sendo cada conjunto conectado aos outros por uma rede de interconexão. Todos os conjuntos compartilham um mesmo sistema operacional e referenciam o mesmo espaço de endereçamento.

A diferença entre os sistemas NUMA e SMP está no desempenho das operações de acesso à memória. Na Fig. 13.7 é apresentado um exemplo de uma arquitetura NUMA, tendo cada conjunto três processadores e uma memória principal, conectados a um mesmo barramento interno. Os conjuntos são interligados através de um barramento interconjunto, criando um modelo hierárquico. Quando um processador faz um acesso local, ou seja, à memória dentro do mesmo conjunto, o tempo de acesso é muito menor que um acesso à memória remota em um outro conjunto. Para manter um nível de desempenho satisfatório, o sistema deve manter a maioria dos acessos locais, evitando acessos à memória remota.

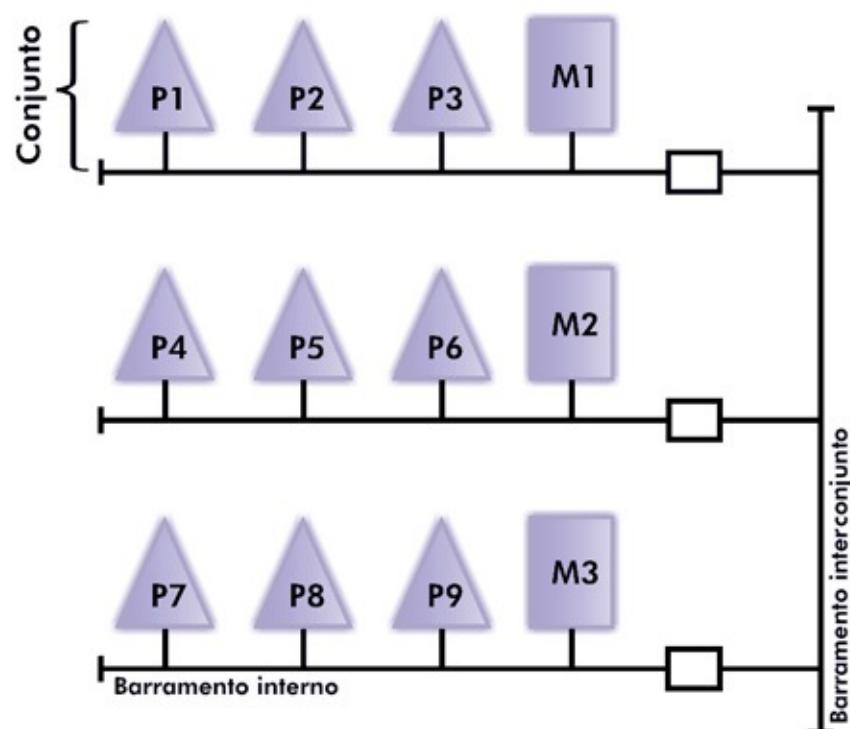


Fig. 13.7 Exemplo de arquitetura NUMA.

O primeiro sistema NUMA, conhecido como Cm*, era formado por vários conjuntos de componentes, cada um com um processador, um módulo de memória principal e dispositivos de E/S opcionais, conectados por um barramento. Os conjuntos eram interconectados por um outro barramento único e cada conjunto possuía um controlador de memória que determinava se o endereço referenciado era local ou remoto. Se o endereço fosse local, o controlador simplesmente acessava o módulo de memória no mesmo barramento. Se o endereço fosse em um outro conjunto, o controlador local solicitava o acesso ao controlador do conjunto remoto, que, por sua vez, realizava a operação.

Existem diversas formas de interligar os vários conjuntos neste tipo de arquitetura, como as topologias em árvore tipo fat-tree, anel, grid 2D, torus 2D e hipercubo, tendo cada topologia um custo e um desempenho associados (Fig. 13.8).

A Tabela 13.5 apresenta alguns sistemas NUMA comerciais e suas principais características (Patterson, 1996).

Tabela 13.5 Características de sistemas NUMA

Sistema	Fabricante	Número de processadores	Topologia
CM-2	Thinking Machines	1.024 a 4.096	Cubo
Paragon	Intel	4 a 2.048	Grid 2D
T3D	Cray	16 a 2.048	Torus 3D
SP-2	IBM	2 a 512	Árvore
CM-5	Thinking Machines	32 a 2.048	Árvore

Na arquitetura NUMA, a memória principal é fisicamente distribuída entre os vários processadores, porém existe apenas um único espaço de endereçamento sendo compartilhado. Quando um processador faz uma referência a um endereço de memória, não importa sua localização física, o endereço será acessado de forma transparente, variando o tempo de acesso conforme sua localização. Este modelo de memória é conhecido como *memória compartilhada distribuída* ou *Distributed Shared Memory* (DSM). A Tabela 13.6 apresenta a relação entre o espaço de endereçamento e a distribuição física da memória principal.

Em relação à memória, as primeiras arquiteturas NUMA não utilizavam qualquer mecanismo de cache. A ausência de cache, apesar de simplificar e reduzir o custo do modelo, oferece uma latência muito alta quando são realizados acessos à memória remota. Implementações mais recentes, como o DASH e o Multiplus (Aude, 1996), utilizam memória cache para reduzir a latência das operações remotas, mas introduzem o problema da coerência de cache. Estas implementações são conhecidas como CC-NUMA (*Cache Coherent NUMA*).

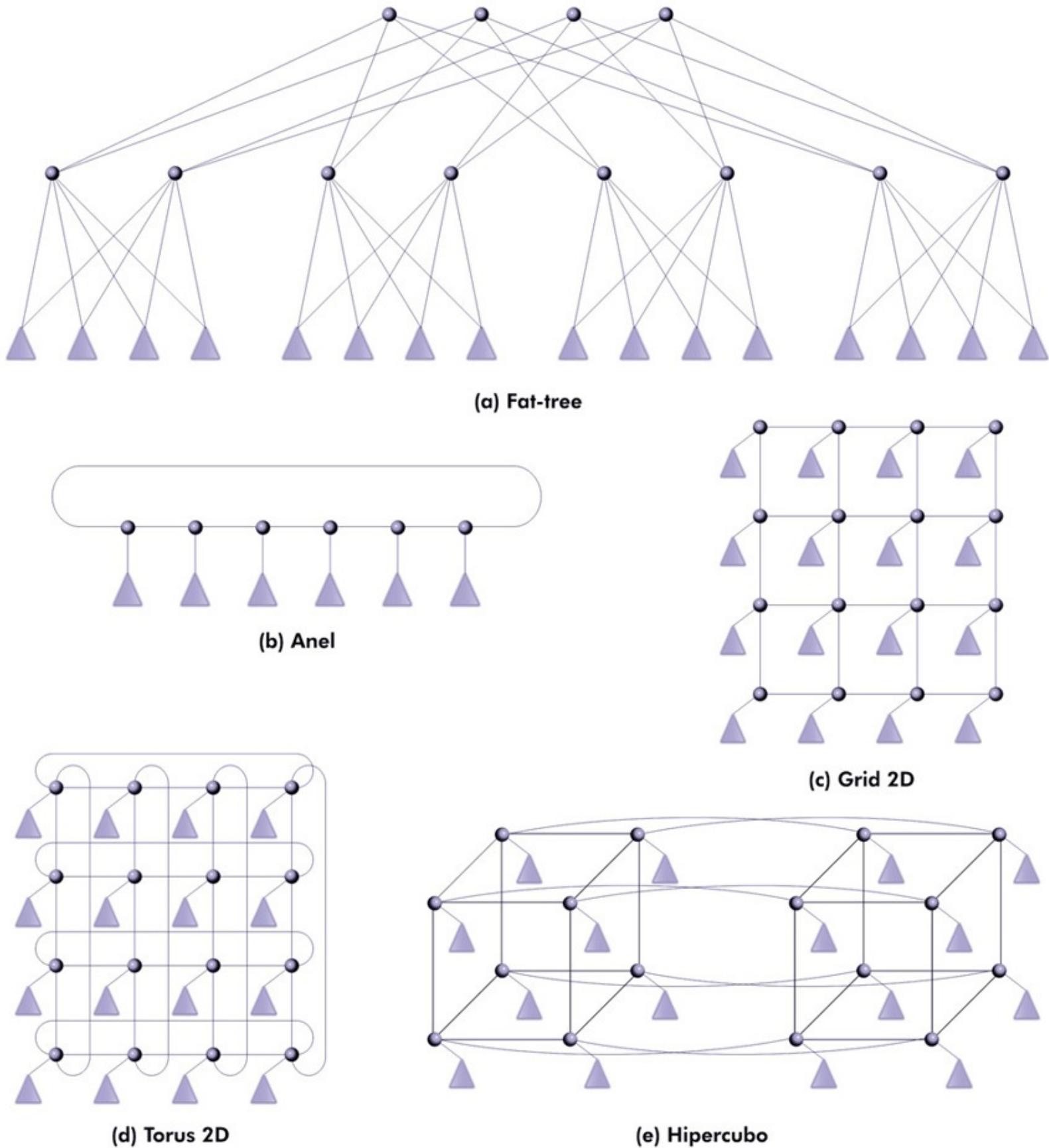


Fig. 13.8 Topologias NUMA.

O DASH (*Directory Architecture for Shared Memory*), desenvolvido na Universidade de Stanford, permite uma configuração inicial com 16 conjuntos, sendo cada conjunto formado por quatro processadores, uma memória principal e dispositivos de E/S opcionais. Os elementos de um mesmo conjunto são ligados através de um barramento comum e os vários conjuntos interligados através de uma

matriz retangular. O espaço total de endereçamento do DASH é 256 Mb, e cada conjunto possui um intervalo de 16 Mb desse espaço. A memória local de cada conjunto é dividida, por sua vez, em blocos de 16 bytes, produzindo um total de 1 Mb bloco por conjunto (Fig. 13.9).

Tabela 13.6 Espaço de endereçamento × memória principal

	Espaço de endereçamento compartilhado	Espaço de endereçamento individual
Memória principal centralizada	SMP	N/A
Memória principal distribuída	DSM	Clusters e SOR

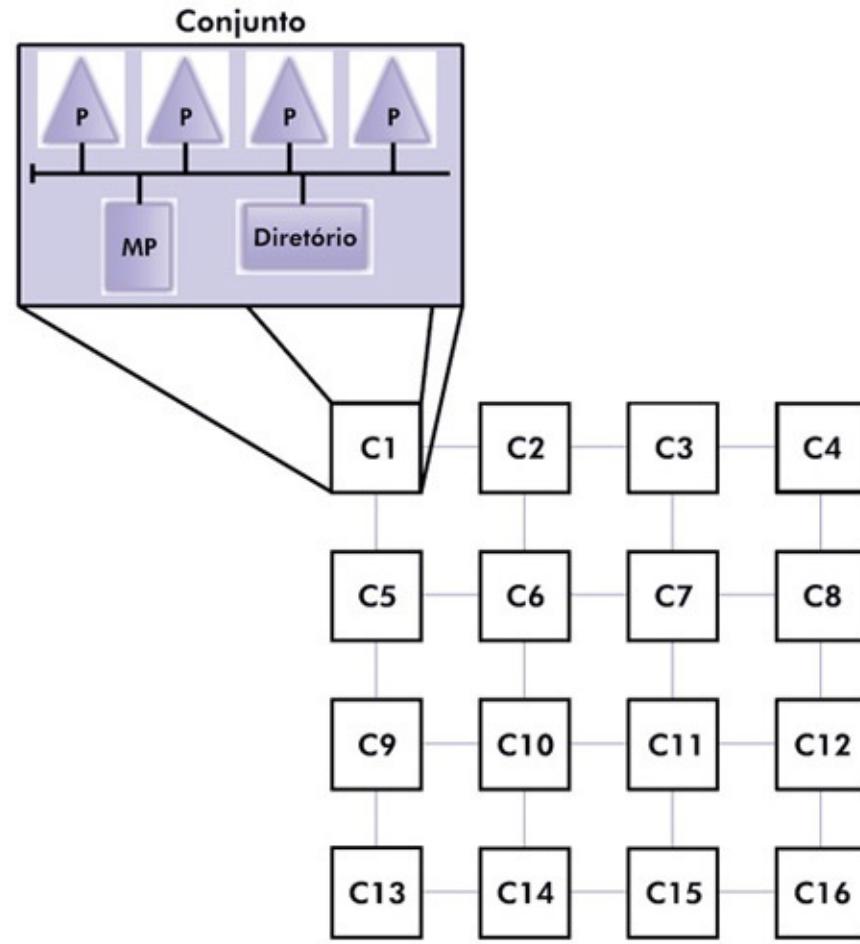


Fig. 13.9 Arquitetura DASH.

O problema de coerência de caches no DASH é tratado em dois níveis: dentro do mesmo conjunto e entre conjuntos diferentes. A coerência de cache dentro do conjunto é implementada utilizando-se a técnica de snooping. A coerência de caches entre conjuntos utiliza o esquema de diretórios. Para cada conjunto existe um diretório que mantém o controle dos blocos que foram copiados para outros conjuntos, permitindo identificar os blocos que estão em cache.

O Multiplus, que vem sendo desenvolvido na UFRJ/NCE, permite até 128 conjuntos, sendo cada conjunto formado por até oito elementos de processamento, conectados por uma rede de interconexão multiestágio n-cube invertido. Um elemento de processamento (EP) é formado por um processador, uma memória

cache e um módulo de memória principal (Fig. 13.10).

Podem existir quatro tipos de acesso à memória no Multiplus: (1) acesso rápido ao cache do EP, (2) acesso à própria memória local do EP, (3) acesso a um módulo de memória em outro EP, porém dentro do mesmo conjunto sem o uso da rede de interconexão, e (4) acesso a um EP em outro conjunto utilizando a rede de interconexão. O problema de coerência de caches no Multiplus é implementado em dois níveis: dentro do mesmo conjunto e entre conjuntos diferentes. A coerência de cache dentro do conjunto é tratada com as técnicas geralmente adotadas em arquiteturas de barramento. A coerência de caches entre conjuntos é baseada no modelo de consistência *lazy release*.

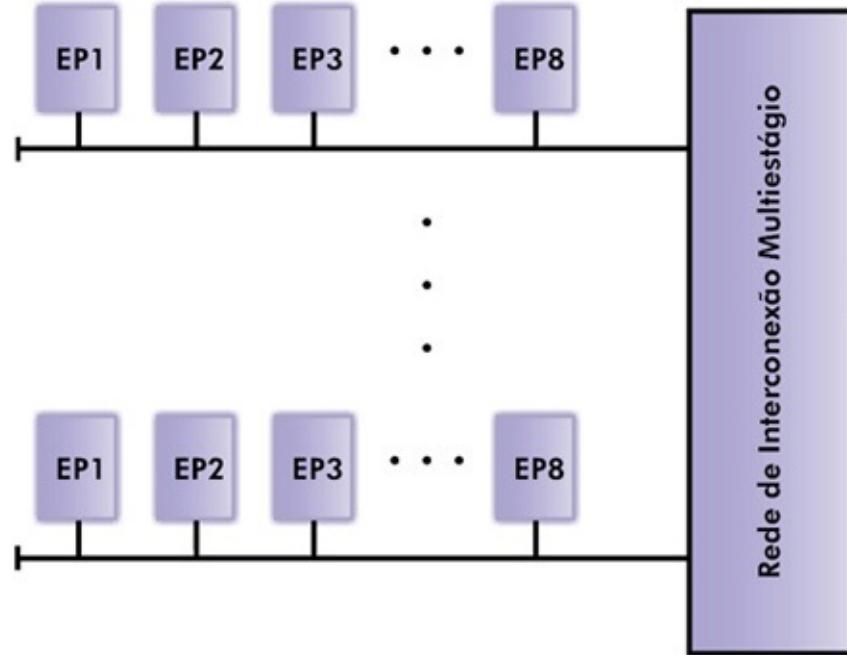


Fig. 13.10 Arquitetura do Multiplus.

Sistemas NUMA são uma alternativa às organizações SMP, pois oferecem uma escalabilidade maior de processadores, menor custo e maior desempenho. Uma grande dificuldade em sistemas NUMA é manter um nível de desempenho satisfatório, devido à possibilidade de existirem tempos de acesso à memória principal bem diferentes.

Neste caso, o sistema deve garantir o maior número possível de acessos locais, evitando acessos à memória remota.

► 13.7 Clusters

Clusters são sistemas fracamente acoplados, formados por nós conectados por uma rede de interconexão de alto desempenho dedicada. Cada nó da rede é denominado membro do cluster e possui seus próprios recursos, como processadores, memória, dispositivos de E/S e sistema operacional. Geralmente, os membros do cluster são de um mesmo fabricante, principalmente por questões de incompatibilidade dos sistemas operacionais.

O primeiro cluster comercial foi desenvolvido pela Digital Equipment Corporation (DEC) em 1983, com o nome de VAXcluster. A Tabela 13.7 apresenta alguns exemplos de clusters comerciais, com o número máximo de membros e processadores.

Tabela 13.7 Exemplos de clusters

Cluster	Número máximo de membros	Número máximo de UCPs por membro	Número total de UCPs
HP 9000 EPS 21	16	4	64
IBM RS/6000 SP2	32	16	512
Sun Enterprise Cluster 6000 HA	2	30	60
Tandem NonStop Himalaya S70000	256	16	4096

Cada membro do cluster possui seu próprio espaço de endereçamento individual, e a comunicação entre os membros se faz, na maioria das implementações, pelo mecanismo de troca de mensagens através da rede de interconexão. Inicialmente, os fabricantes utilizavam esquemas proprietários de interconexão, mas, atualmente, existem diversos padrões estabelecidos, como FDDI, Fibre Channel, ATM e Gigabit Ethernet.

Geralmente, a rede de interconexão é restrita aos membros do cluster, e o acesso aos serviços oferecidos é feito a partir de uma outra rede de acesso. A Fig. 13.11 apresenta um exemplo de cluster com dois membros (Membro 1 e Membro 2), conectados por uma rede de interconexão do tipo Gigabit Ethernet, enquanto a rede de acesso é uma rede local Ethernet 10/100. Os membros do cluster compartilham os dispositivos de armazenamento utilizando, por exemplo, o padrão SCSI ou alguma rede de armazenamento do tipo SAN (*Storage Area Network*).

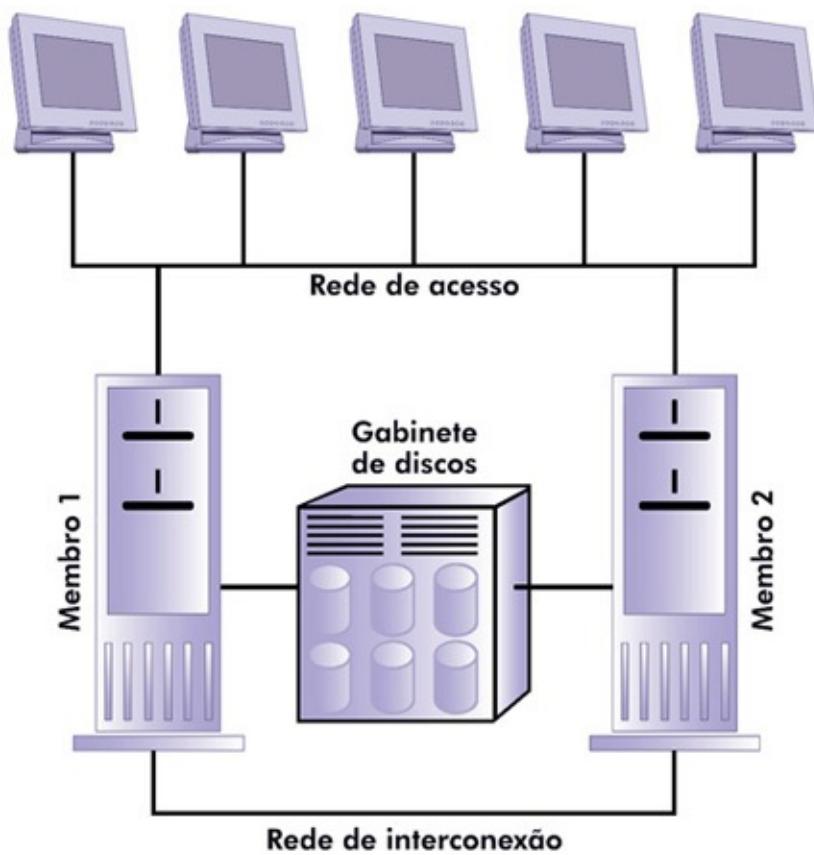


Fig. 13.11 Exemplo de cluster.

A razão para o surgimento e a rápida aceitação de sistemas em cluster foi a maior necessidade de tolerância a falhas e a alta disponibilidade, de forma a reduzir o downtime. O downtime pode ser reduzido não apenas contornando problemas de hardware e software, mas também reduzindo o impacto de manutenções preventivas e atualizações. Outras vantagens de um cluster são sua escalabilidade e o balanceamento de carga. Por essas razões, clusters são utilizados em servidores Web, sistemas de comércio eletrônico, servidores de banco de dados e soluções de firewall.

Quando um usuário tem acesso ao cluster, ele não tem conhecimento do número de membros que compõem o cluster e da identificação individual de cada membro. Para o usuário é como se ele estivesse acessando um único sistema fortemente acoplado. Esta característica, conhecida como imagem única do sistema, é implementada pelo sistema operacional juntamente com as aplicações que estão sendo processadas no ambiente. A maioria dos clusters implementa apenas parcialmente este conceito, principalmente quando o problema envolve o desenvolvimento de aplicações paralelas.

Quando ocorre uma falha em um dos membros do cluster, outro membro verifica o problema e assume as suas funções (*failover*). Após resolvido o problema, a situação inicial pode ser restaurada (*failback*). Para que o processo de failover seja efetivo, as aplicações processadas no ambiente devem ter sido desenvolvidas para utilizar os recursos oferecidos pelo cluster. Neste caso, se houver uma falha em um dos membros do cluster, o usuário não perceberá o problema, pois outro membro assumirá o processamento de forma transparente.

Sistemas em cluster permitem o compartilhamento de dispositivos de E/S, como impressoras, discos e fitas, independentemente do sistema a que o usuário esteja conectado, o que permite melhor utilização dos recursos computacionais. Nesse tipo de configuração, se um dos membros falhar o acesso aos

dispositivos não será interrompido.

Além das vantagens apresentadas, clusters podem ser utilizados também para processamento paralelo. Neste ambiente, conhecido como *cluster farm*, uma aplicação pode ser dividida entre os vários membros do cluster, de forma que possa ser executada em paralelo, reduzindo o seu tempo de processamento. A programação paralela nesse tipo de ambiente exige que a aplicação seja desenvolvida para tirar proveito do ambiente em cluster, não sendo transparente como nos sistemas fortemente acoplados. Existem pacotes voltados para facilitar o desenvolvimento de aplicações paralelas em clusters, como o PVM (*Parallel Virtual Machines*), MPI (*Message Passing Interface*) e Tredmarks.

► 13.8 Sistemas Operacionais de Rede

Os *sistemas operacionais de rede* (*SOR*) são o melhor exemplo de um ambiente fracamente acoplado. Cada sistema, também chamado host ou nó, possui seus próprios recursos de hardware, como processadores, memória e dispositivos de E/S. Os nós são totalmente independentes dos demais, sendo interconectados por uma rede de comunicação de dados formando uma rede de computadores (Fig. 13.12).

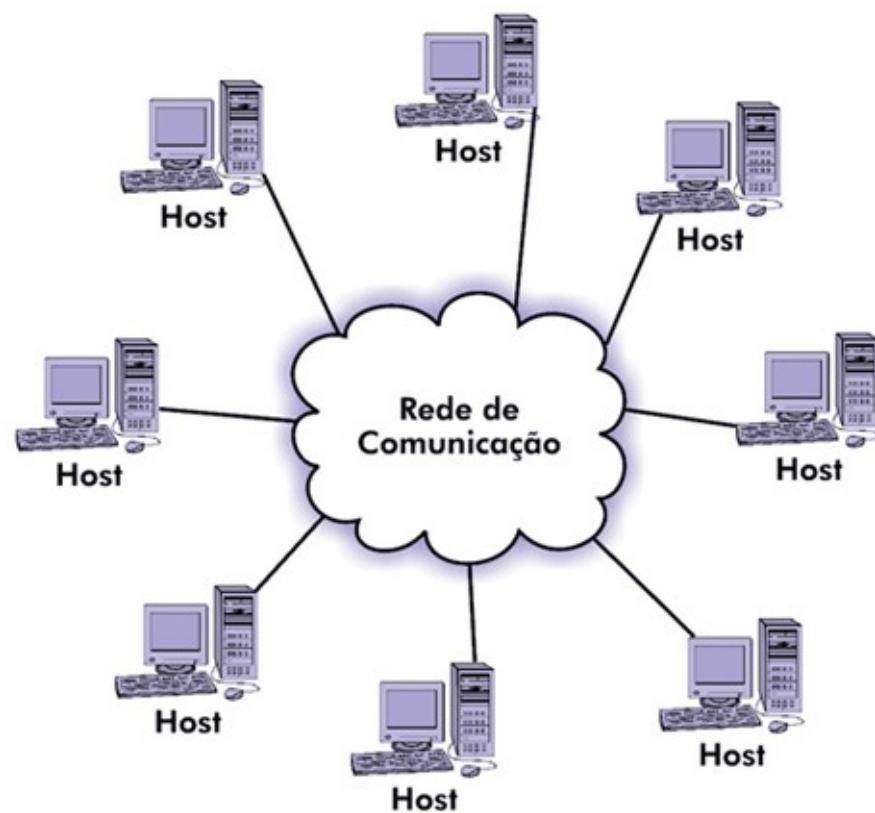


Fig. 13.12 Rede de computadores.

Os SORs são utilizados tanto em *redes locais* (*Local Area Network — LAN*), como Ethernet e Token Ring, quanto em *redes distribuídas* (*Wide Area Network — WAN*), como a Internet. A comunicação entre os diversos nós é feita por uma interface de rede que possibilita o acesso aos demais componentes da rede. A princípio, não existe um limite máximo para o número de nós que podem fazer parte de uma rede de computadores. A Fig. 13.13 apresenta as principais topologias utilizadas em redes de computadores.

Cada nó é totalmente independente dos demais, possuindo seu próprio sistema operacional e espaço de endereçamento. Os sistemas operacionais podem ser heterogêneos, bastando apenas que os nós comuniquem-se utilizando o mesmo protocolo de rede. Na Internet, cada host pode estar processando um sistema operacional diferente, mas todos estão se comunicando através do mesmo protocolo de rede, no caso, os protocolos da família TCP/IP (*Transmission Control Protocol/Internet Protocol*).

O usuário tem acesso aos recursos de outros computadores especificando o nome ou endereço do nó onde o recurso é oferecido. Nesse ambiente, cada estação pode compartilhar seus recursos com o restante da rede. Caso uma estação sofra qualquer problema, os demais componentes da rede podem continuar o processamento, apenas não dispondo dos recursos antes oferecidos. Não existe a ideia de imagem única do sistema, como também não existe a tolerância a falhas e a alta disponibilidade presente nos sistemas distribuídos.

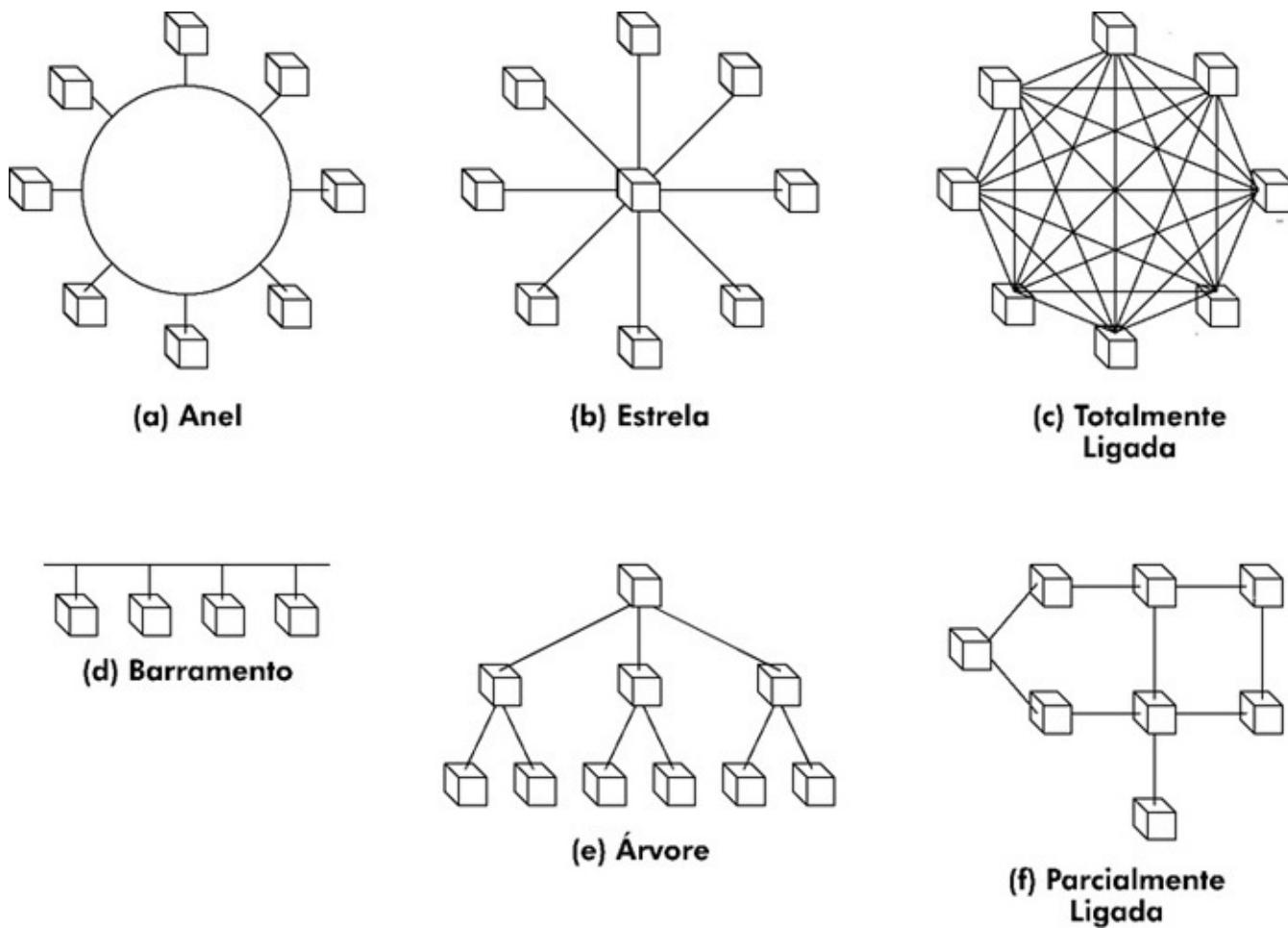


Fig. 13.13 Topologias de redes de computadores.

A grande maioria dos SORs e seus protocolos de rede implementa o modelo *cliente/servidor*. Neste modelo, existem um ou mais servidores que oferecem serviços e respondem às solicitações dos demais clientes da rede. Um bom exemplo deste tipo de ambiente são as redes locais, onde existem servidores de arquivos, de impressão, de banco de dados, de correio eletrônico e de intranet. O Microsoft Windows 2000 e o Novell Netware são dois exemplos de SORs voltados para oferecer este tipo de serviço.

Não apenas em LANs o modelo cliente/servidor é implementado. Este tipo de ambiente pode ser observado quando um usuário acessa uma página Web na Internet. Neste caso, o browser funciona como

um cliente que solicita informações, e o servidor Web, por sua vez, responde às solicitações. A Tabela 13.8 apresenta alguns exemplos de serviços e protocolos utilizados na Internet baseados neste modelo.

Tabela 13.8 Serviços e protocolos Internet

Serviços	Protocolos
Correio eletrônico	Simple Mail Transfer Protocol (SMTP)
	Post-Office Protocol (POP)
Web	Hyper-Text Transfer Protocol (HTTP)
Cópia remota de arquivos	File Transfer Protocol (FTP)
Terminal remoto	Telnet Protocol
Gerência remota	Simple Network Management Protocol (SNMP)
Tradução de nomes	Domain Name System (DNS)

► 13.9 Sistemas Distribuídos

Um *sistema distribuído* é um conjunto de sistemas autônomos, interconectados por uma rede de comunicação e que funciona como se fosse um sistema fortemente acoplado. Cada componente de um sistema distribuído possui seus próprios recursos, como processadores, memória principal, dispositivos de E/S, sistema operacional e espaço de endereçamento. Os tipos de sistemas operacionais que compõem um sistema distribuído não precisam ser necessariamente homogêneos.

O que diferencia um sistema distribuído dos demais sistemas fracamente acoplados é a existência de um relacionamento mais forte entre os seus componentes. Podemos definir um sistema distribuído como sendo um sistema fracamente acoplado pelo aspecto de hardware e fortemente acoplado pelo aspecto de software. Para o usuário e suas aplicações, é como se não existisse uma rede de computadores independente, mas sim um único sistema fortemente acoplado (Fig. 13.14). Este conceito é chamado de *imagem única do sistema* (*single system image*).

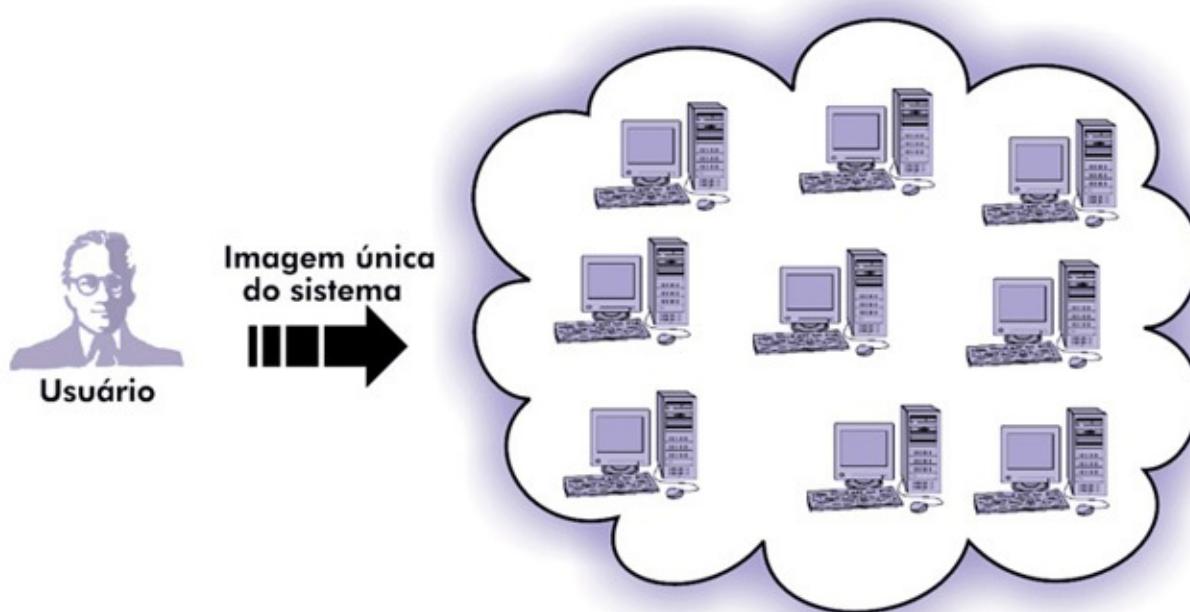


Fig. 13.14 Sistema distribuído.

Os componentes de um sistema distribuído podem estar conectados em um ambiente de uma rede local ou através de uma rede distribuída. Na verdade, a localização de um componente não deve constituir um fator a ser levado em consideração, bem como as características físicas da rede. A escalabilidade dos sistemas distribuídos é, a princípio, ilimitada, pois basta acrescentar novos componentes à rede em função da necessidade.

Os sistemas distribuídos permitem que uma aplicação seja dividida em diferentes partes, que se comunicam através de linhas de comunicação, podendo cada parte ser executada em qualquer processador de qualquer sistema (*aplicações distribuídas*). Para que isto seja possível, o sistema deve oferecer transparência e tolerância a falhas em vários níveis, a fim de criar a ideia de imagem única do sistema.

13.9.1 Transparência

Em sistemas distribuídos, o conceito de *transparência* toma-se fator-chave, pois, a partir dele, um conjunto de sistemas independentes parece ser um sistema único, criando a ideia da imagem única do sistema. A seguir, são apresentados os vários aspectos relativos à transparência em sistemas distribuídos:

- *Transparência de acesso*: é a possibilidade de acesso a objetos locais ou remotos de maneira uniforme;
- *Transparência de localização*: significa que o usuário não deve se preocupar com a localização dos recursos de que necessita. Os recursos são acessados por nomes lógicos e não por nomes físicos, criando assim a independência da localização dos recursos;
- *Transparência de migração*: permite que os recursos sejam fisicamente movidos para outro sistema, sem que os usuários e suas aplicações sejam afetados;

- *Transparência de replicação*: permite a duplicação de informações, com o objetivo de aumentar a disponibilidade e o desempenho do sistema, de forma sincronizada e consistente;
- *Transparência de concorrência*: permite que vários processos sejam executados paralelamente e os recursos sejam compartilhados de forma sincronizada e consistente;
- *Transparência de paralelismo*: possibilita que uma aplicação paralela seja executada em qualquer processador de qualquer sistema, como em um sistema fortemente acoplado;
- *Transparência no desempenho*: oferece aos usuários tempos de resposta independentes de alterações na estrutura do sistema ou na sua carga. Além disso, operações realizadas remotamente não devem apresentar resultados piores do que as realizadas localmente;
- *Transparência de escalabilidade*: permite que o sistema cresça sem a necessidade de alterar as aplicações e seus algoritmos;
- *Transparência a falhas*: garante que, em caso de falha de um sistema, as aplicações continuem disponíveis sem interrupção.

Em um sistema distribuído, quando um usuário se conecta ao sistema não há necessidade de especificar o nome do componente a que estará conectado. Independentemente da localização física dos objetos, o usuário terá acesso a todos os seus arquivos, diretórios e demais recursos de forma transparente. Ao executar uma aplicação, o usuário não saberá em quais ou quantos componentes a sua aplicação foi dividida. Caso um erro ocorra em um desses componentes, o usuário não terá conhecimento, ficando como responsabilidade do sistema operacional a resolução de todos os problemas.

13.9.2 Tolerância a Falhas

Para que um sistema distribuído possa oferecer transparência é preciso que o sistema implemente *tolerância a falhas* de hardware e, principalmente, de software. Neste caso, o sistema deve garantir que, em caso de problema em um de seus componentes, as aplicações continuem sendo processadas sem nenhuma interrupção ou intervenção do usuário, de forma totalmente transparente.

A tolerância a falhas de hardware é facilmente oferecida utilizando-se componentes redundantes, como fontes duplicadas, vários processadores, memória com detecção e correção de erro e técnicas de RAID. Em sistemas distribuídos, a tolerância a falhas de hardware passa pela redundância dos meios de conexão entre os sistemas, como placas de rede, linhas de comunicação e dispositivos de rede.

A tolerância a falhas de software é bem mais complexa de implementar. Quando uma falha deste tipo ocorre, como uma falha no sistema operacional, a aplicação deve continuar sem que o usuário perceba nenhum problema. Enquanto em uma rede de computadores o usuário deverá se reconectar a um outro sistema em funcionamento e reiniciar sua tarefa, em um sistema distribuído o problema deve ser resolvido de forma transparente, mantendo a integridade e consistência dos dados.

Com a tolerância a falhas, é possível também oferecer alta disponibilidade e confiabilidade. Como existem sistemas autônomos, em caso de falha de um dos componentes um outro sistema poderá assumir suas funções, sem a interrupção do processamento. Como as aplicações estão distribuídas por diversos sistemas, caso ocorra algum problema com um dos componentes é possível que um deles assuma de forma transparente o papel do sistema defeituoso.

13.9.3 Imagem Única do Sistema

A maior dificuldade em implementar um sistema distribuído é a complexidade em criar para os usuários e suas aplicações uma imagem única do sistema a partir de um conjunto de sistemas autônomos. Para conseguir criar um ambiente fisicamente distribuído e logicamente centralizado é necessário um sistema operacional capaz de lidar com os diversos problemas de comunicação existentes em um ambiente fracamente acoplado. O sistema precisa oferecer tolerância a falhas de forma transparente, independente do tipo da rede de comunicação.

Um problema encontrado em sistemas fortemente acoplados é o compartilhamento de recursos de forma segura. Em sistemas distribuídos, a utilização de recursos concorrentemente exige mecanismos mais complexos e lentos para manter a integridade e a segurança dos dados.

Um dos grandes desafios para a adoção de sistemas distribuídos é a dificuldade no desenvolvimento de aplicações paralelas. Enquanto a programação em sistemas fortemente acoplados é relativamente transparente, em sistemas distribuídos o desenvolvimento não é tão simples. Apesar de algumas aplicações serem naturalmente paralelas, como ordenações e processamento de imagens, desenvolver aplicações distribuídas exige uma grande interação do programador com detalhes de codificação e escalonamento da aplicação.

► 13.10 Exercícios

1. Por que sistemas SMP que utilizam arquitetura em barramento único possuem sérias limitações quanto ao número de processadores?
2. Como o esquema de cache melhora o desempenho de sistemas SMP com topologia em barramento único?
3. Qual a diferença entre os mecanismos write-through e write-back para coerência de caches? Qual mecanismo apresenta o melhor desempenho?
4. Qual a diferença entre os mecanismos write-invalidate e write-update para snooping?
5. Um sistema com oito processadores precisaria de quantos comutadores em uma arquitetura de barramento cruzado? Este mesmo sistema precisaria de quantos comutadores em uma rede Ômega?
6. Quantos comutadores no máximo uma mensagem deve percorrer em um sistema com 256 processadores organizados em uma matriz 16 X 16?
7. Qual a diferença entre os sistemas SMP e NUMA?
8. Como os sistemas NUMA permitem um número maior de processadores?
9. O que são clusters e como são utilizados?
10. Defina o conceito de imagem única do sistema.
11. O que é transparência em sistemas distribuídos?
12. Considere um sistema com dois processadores (UCP1 e UCP2) e memória compartilhada. A fila de prontos é única e compartilhada entre os processadores. Neste sistema são criados 5 processos com as seguintes características:

Processo	Tempo de UCP	Prioridade	Tempo em que foi criado
P1	10	1	0
P2	6	7	2
P3	12	5	4
P4	11	2	6
P5	7	4	11

O tempo de escalonamento ou troca de contexto entre processos deve ser desconsiderado. No tempo 0, serão buscados processos na fila de prontos para execução nos processadores. A escolha da UCP1 sempre será prioritária sobre a UCP2 no momento do escalonamento. Desenhe um diagrama mostrando o que está acontecendo em cada um dos processadores até o término da execução dos cinco processos, calculando o tempo de turnaround dos processos e considerando os seguintes esquemas de escalonamento:

- a) Circular com fatia de tempo igual a 5;
- b) Prioridade (número menor implica prioridade maior).



Anexo – Questões do ENADE

Neste anexo são apresentadas as questões do ENADE (Exame Nacional de Desempenho dos Estudantes) relacionadas ao tema, retiradas dos exames dos cursos de Análise e Desenvolvimento de Sistemas (ADS) e Ciência da Computação (COMP) de diversos anos. As questões estão vinculadas aos capítulos do livro em que são abordadas e devidamente identificadas com o tipo da prova, ano de realização e número da questão.

► Capítulo 1

1. [COMP, 2005, Q42] Duas possibilidades para a construção de sistemas com múltiplos processadores são: processadores idênticos com um único espaço de endereçamento interligados por um barramento único (SMP); e máquinas monoprocessadas conectadas por uma rede (cluster). Com relação a esses sistemas, assinale a opção correta.
 - a) A comunicação entre processadores de um cluster é, potencialmente, muito mais rápida que a comunicação entre processadores de um sistema SMP, pois redes atuais possuem taxa de transmissão da ordem de gigabits/s, enquanto as melhores memórias operam somente com frequências da ordem de centenas de mega-hertz.
 - b) Comunicação entre processos pode ser implementada de forma muito mais eficiente em um cluster que em um sistema SMP, pois, nesse último, todos os processos precisam compartilhar os mesmos dispositivos de entrada e saída.
 - c) Em um sistema SMP, é mais simples substituir um processador defeituoso, pois, em um cluster, toda a rede de comunicação deve ser desabilitada para que a troca seja efetuada sem prejudicar a troca de mensagens entre os processos.
 - d) Alocação de memória para processos é muito mais simples em um cluster, pois cada processador executa um único processo na sua memória exclusiva e, dessa forma, não existe o problema de

distribuição de processos no espaço de endereçamento único da máquina SMP.

- e) Em um cluster, o custo da escalabilidade é muito menor, pois, para a interconexão entre as máquinas, podem ser utilizados equipamentos comuns usados em uma rede local de computadores, ao passo que um sistema SMP exige conexões extras no barramento e gabinetes especiais.

► Capítulo 3

1. [COMP, 2005, Q41] Processadores atuais incluem mecanismos para o tratamento de situações especiais, conhecidas como interrupções. Em uma interrupção, o fluxo normal de instruções é interrompido para que a causa da interrupção seja tratada. Com relação a esse assunto, assinale a opção correta.
 - a) Controladores de entrada e saída geram interrupções de forma síncrona à execução do processador, para que nenhuma instrução fique incompleta devido à ocorrência da interrupção.
 - b) Quando uma interrupção ocorre, o próprio processador salva todo o seu contexto atual, tais como registradores de dados e endereço e códigos de condição, para que esse mesmo contexto possa ser restaurado pela rotina de atendimento da interrupção.
 - c) O processador pode autointerromper-se para tratar exceções de execução, tais como um erro em uma operação aritmética, uma tentativa de execução de instrução ilegal ou uma falha de página em memória virtual.
 - d) Rotinas de tratamento de interrupção devem ser executadas com o mecanismo de interrupção inibido, pois esse tipo de rotina não permite aninhamento.
 - e) O uso de interrupção para realizar entrada ou saída de dados somente é eficiente quando o periférico trata grandes quantidades de dados, como é o caso de discos magnéticos e discos ópticos. Para periféricos com pouco volume de dados, como teclados e mouses, o uso de interrupção é ineficiente.

► Capítulo 4

1. [ADS, 2008, Q15] O conceito de máquina virtual (MV) foi usado na década de 1970 do século passado no sistema operacional IBM System 370. Atualmente, centros de dados (datacenters) usam MVs para migrar tarefas entre servidores conectados em rede e, assim, equilibrar carga de processamento. Além disso, plataformas atuais de desenvolvimento de software empregam MVs (Java, .NET). Uma MV pode ser construída para emular um processador ou um computador completo. Um código desenvolvido para uma máquina real pode ser executado de forma transparente em uma MV. Com relação a essas informações, assinale a opção correta.
 - a) O conceito de transparência mencionado indica que a MV permite que um aplicativo acesse diretamente o hardware da máquina.
 - b) Uma das vantagens mais significativas de uma MV é a economia de carga de CPU e de memória RAM na execução de um aplicativo.
 - c) Uma MV oferece maior controle de segurança, uma vez que aplicativos são executados em um

ambiente controlado.

- d) Para emular uma CPU dual-core, uma MV deve ser instalada e executada em um computador com CPU dual-core.
- e) Como uma MV não é uma máquina real, um sistema operacional nela executado fica automaticamente imune a vírus.

2. [ADS, 2011, Q28] A virtualização permite que um único computador hospede múltiplas máquinas virtuais, cada uma com seu próprio sistema operacional. Essa técnica tem ganhado importância nos dias atuais e vem sendo utilizada para resolver diversos tipos de problemas. Considerando os diversos aspectos a serem considerados na utilização da virtualização, avalie as afirmações abaixo.

- I. Um sistema operacional sendo executado em uma máquina virtual utiliza um subconjunto da memória disponível na máquina real.
- II. Uma das aplicações da virtualização é a disponibilização de múltiplos sistemas operacionais para teste de software.
- III. A virtualização só pode ser utilizada em sistemas operacionais Linux.
- IV. Um sistema operacional executado em uma máquina virtual apresenta um desempenho superior ao que alcançaria quando executado diretamente na mesma máquina real.

É correto apenas o que se afirma em

- a) I.
- b) III.
- c) I e II.
- d) II e IV.
- e) III e IV.

3. [COMP, 2011, Q18] Um vendedor de artigos de pesca obteve com um amigo o código executável (já compilado) de um programa que gerencia vendas e faz o controle de estoque, com o intuito de usá-lo em sua loja. Segundo o seu amigo, o referido programa foi compilado em seu sistema computacional pessoal (sistema A) e funciona corretamente. O vendedor constatou que o programa executável também funciona corretamente no sistema computacional de sua loja (sistema B). Considerando a situação relatada, analise as afirmações a seguir.

- I. Os computadores poderiam ter quantidades diferentes de núcleos (cores).
- II. As chamadas ao sistema (system call) do sistema operacional no sistema A devem ser compatíveis com as do sistema B.
- III. O conjunto de instruções do sistema A poderia ser diferente do conjunto de instruções do sistema B.
- IV. Se os registradores do sistema A forem de 64 bits, os registradores do sistema B poderiam ser de 32 bits.

É correto o que se afirma em

- a) III, apenas.
- b) I e II, apenas.
- c) III e IV, apenas.
- d) I, II e IV, apenas.
- e) I, II, III e IV.

► Capítulo 6

1. [COMP, 2011, Q29] Uma antiga empresa de desenvolvimento de software resolveu atualizar toda sua infraestrutura computacional adquirindo um sistema operacional multitarefa, processadores multicore (múltiplos núcleos) e o uso de uma linguagem de programação com suporte a threads. O sistema operacional multitarefa de um computador é capaz de executar vários processos (programas) em paralelo. Considerando esses processos implementados com mais de uma thread (multithreads), analise as afirmações abaixo.
- I. Os ciclos de vida de processos e threads são idênticos.
 - II. Threads de diferentes processos compartilham memória.
 - III. Somente processadores multicore são capazes de executar programas multithreads.
 - IV. Em sistemas operacionais multitarefa, threads podem migrar de um processo para outro.

É correto apenas o que se afirma em

- a) I.
- b) II.
- c) I e III.
- d) I e IV.
- e) II e IV.

► Capítulo 7

1. [COMP, 2005, Q53] O problema do buffer limitado de tamanho N é um problema clássico de sincronização de processos: um grupo de processos utiliza um buffer de tamanho N para armazenar temporariamente itens produzidos; processos produtores produzem os itens, um a um, e os armazenam no buffer; processos consumidores retiram os itens do buffer, um a um, para processamento. O problema do buffer limitado de tamanho N pode ser resolvido com a utilização de semáforos, que são mecanismos de software para controle de concorrência entre processos. Duas operações são definidas para um semáforo s: wait(s) e signal(s).

Considere o problema do buffer limitado de tamanho N cujos pseudocódigos dos processos produtor e consumidor estão mostrados na tabela abaixo. Pode-se resolver esse problema com a utilização dos semáforos mutex, cheio e vazio, inicializados, respectivamente, com 1, 0 e N.

Processo produtor

Produz item
Comando_a
Comando_b
Coloca no buffer
Comando_c
Comando_d

Processo consumidor

Comando_e
Comando_f
Retira do buffer
Comando_g
Comando_h
Consome o item

A partir dessas informações, para que o problema do buffer limitado de tamanho N cujos pseudocódigos foram apresentados possa ser resolvido a partir do uso dos semáforos mutex, cheio e vazio, é necessário que comando_a, comando_b, comando_c, comando_d, comando_e, comando_f, comando_g e comando_h correspondam, respectivamente, às operações

- a) wait(vazio), wait(mutex), signal(mutex), signal(cheio), wait(cheio), wait(mutex), signal(mutex) e signal(vazio).
- b) wait(cheio), wait(mutex), signal(mutex), signal(vazio), wait(vazio), signal(mutex), signal(mutex) e wait(cheio).
- c) wait(mutex), wait(vazio), signal(cheio), signal(mutex), wait(mutex), wait(vazio), signal(cheio) e signal(mutex).
- d) wait(mutex), wait(vazio), signal(cheio), signal(mutex), wait(mutex), wait(cheio), signal(vazio) e signal(mutex).
- e) wait(vazio), signal(mutex), signal(cheio), wait(mutex), wait(cheio), signal(mutex), signal(vazio) e signal(mutex).

► Capítulo 8

1. [COMP, 2005, Q71] Sistemas operacionais de tempo real são utilizados em controle de processos automatizados, em que o tempo de resposta a determinados eventos é um fator crítico. Com relação a esse assunto, julgue os itens seguintes.
 - I. Sistemas de tempo real estritos (hard real-time) não utilizam dispositivos de memória secundária (como discos), pois estes não oferecem garantia de término das operações dentro de uma quantidade máxima de tempo.
 - II. Um sistema operacional de propósito geral pode ser modificado para ser de tempo real atribuindo-se prioridades fixas para cada um dos processos.
 - III. O escalonamento mais utilizado por sistemas operacionais de tempo real é o shortest-job-first (tarefa mais curta primeiro).

Assinale a opção correta.

- a) Apenas um item está certo.
- b) Apenas os itens I e II estão certos.
- c) Apenas os itens I e III estão certos.
- d) Apenas os itens II e III estão certos.
- e) Todos os itens estão certos.

► Capítulo 10

1. [COMP, 2005, Q22] Com relação ao gerenciamento de memória com paginação em sistemas operacionais, assinale a opção correta.
 - a) As páginas utilizadas por um processo, sejam de código ou de dados, devem ser obrigatoriamente armazenadas na partição de swap do disco, quando o processo não estiver sendo executado.
 - b) Todas as páginas de um processo em execução devem ser mantidas na memória física enquanto o processo não tiver terminado.
 - c) Um processo somente pode ser iniciado se o sistema operacional conseguir alocar um bloco contíguo de páginas do tamanho da memória necessária para execução do processo.
 - d) O espaço de endereçamento virtual disponível para os processos pode ser maior que a memória física disponível.
 - e) Um processo somente pode ser iniciado se o sistema operacional conseguir alocar todas as páginas de código desse processo.

► Capítulo 11

1. [COMP, 2005, Q52] O método de alocação de espaço de disco utilizado para armazenamento de informações em um sistema de arquivos determina o desempenho desse sistema. Com relação a esse assunto, julgue os itens seguintes.
 - I. A alocação contígua é um método adequado para sistemas em que inserções e remoções de arquivos são frequentes.
 - II. Na alocação indexada, o tamanho máximo de um arquivo depende do número de bits utilizados para representar um índice e do tamanho dos blocos de índices.
 - III. Na alocação encadeada, o tamanho máximo de um arquivo depende do tamanho dos blocos de dados.

Assinale a opção correta.

- a) Apenas um item está certo.
- b) Apenas os itens I e II estão certos.
- c) Apenas os itens I e III estão certos.
- d) Apenas os itens II e III estão certos.

e) Todos os itens estão certos.



Bibliografia

- ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., LEVY, H. M. The Interaction of Architecture and Operating System Design. *Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, p. 108-120, April, 1991.
- ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., LEVY, H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, vol. 10, p. 53-79, Feb., 1992.
- ANDREWS, G. R. *Foundations of Multithread, Parallel, and Distributed Programming*. Addison-Wesley, 2000.
- AUDE, J. S. *The Multiplus/Mulplix Project: Current Status and Perspectives*. NCE/IM/UFRJ, 1996.
- BACH, M. J. *The Design of the Unix Operating System*. Prentice-Hall, 1986.
- BACON, J. *Concurrent Systems: Operating Systems, Database and Distributed Systems*. Addison-Wesley, 1997.
- BEN-ARI, M. *Principles of Concurrent and Distributed Programming*. Prentice-Hall, 1990.
- CHOW, R., JOHNSON, T. *Distributed Operating Systems and Algorithms*. Addison-Wesley, 1997.
- COFFMAN, E. G., ELPHICK, M. J., SHOSHANI, A. System Deadlocks. *Computing Surveys*, vol. 3, p. 67-78, 1971.
- CONWAY, M. A Multiprocessor System Design. *Proceedings of the AFIPS. Fall Joint Computer Conference*, 1963.
- CORBATÓ, F. J., DAGGETT, M. M., DALEY, R. C. An Experimental Time-Sharing System. *Proceedings AFIPS Spring Joint Computer Conference*, p. 334-335, 1962.
- CORBATÓ, F. J., VYSSOTSKY, V. A. Introduction and Overview of the Multics System. *Proceedings AFIPS Fall Joint Computer Conference*, p. 185-196, 1965.
- COULOURIS, G., DOLLIMORE, J., KINDBERG, T. *Distributed Systems: Concepts and Design*. Addison-Wesley, 1995.

- DEITEL, H. M. *Operating Systems*. 2. ed. Addison-Wesley, 1990.
- DENNING, P. J. The Working Set Model of Program Behavior. *CACM*, vol. 11, n. 5, May, 1968.
- DENNING, P. J. Virtual Memory. *Computing Surveys*, vol. 2, n. 3, Sep., 1970.
- DENNING, P. J. Third Generation Computer Systems. *Computing Surveys*, vol. 3, n. 4, December, 1971.
- DENNING, P. J., SCHWARTZ, S. C. Properties of the Working-Set Model. *CACM*, vol. 15, n. 3, March, 1972.
- DENNING, P. J. Working Sets Past and Present. *IEEE Trans. on Software Engineering*, vol. SE-6, p. 64-84, Jan., 1980.
- DENNIS, J. B., VAN HORN, E. C. Programming Semantics for Multiprogrammed Computations. *CACM*, vol. 9, p. 143-155, March, 1966.
- DIJKSTRA, E. W. Solution of a Problem in Concurrent Programming Control. *CACM*, 1965.
- DIJKSTRA, E. W. Cooperating Sequential Processes. *Programming Languages*. Academic Press, 1965.
- ENGLER, D. R., KAASSHOEK, M. F., O'TOOLE, J. Exokernel: An Operating System Architecture for Application-Level Resource Management. *Proc. 15th Symp. On Operating Systems Principles*, ACM, 1995.
- GUIMARÃES, C. C. *Princípios de Sistemas Operacionais*. Campus, 1980.
- HANSEN, P. B., *Operating System Principles*. Prentice-Hall, 1973.
- HANSEN, P. B. The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1975.
- HAVENDER, J. W. Avoiding Deadlock in Multitasking Systems. *IBM System Journal*, vol. 7, p. 74-84, 1968.
- HOARE, C. A. R. Monitors: An Operating System Structuring Concept. *CACM*, vol. 17, p. 549-557, Oct., 1974.
- HOFRI, M. Proof of a Mutual Exclusion Algorithm. *Operating Systems Review*, January, 1990.
- KILBURN, T., EDWARDS, D. B. G., LANIGAN, M. J., SUMMER, F. H. One-level Storage System. *IRE Transactions on Electronic Computers*, EC-11. April, p. 223-35, 1962.
- KLEIMAN, S., SHAH, D., SMAALDERS, B. *Programming with Threads*. SunSoft Press, Prentice-Hall, 1996.
- KNUTH, D. E. Von Neumann's First Computer Program. *Computing Surveys*, vol. 2, n. 4, December 1970.
- LAMPORT, L. A New Solution of Dijkstra's Concurrent Programming Problem. *CACM*, 1974.
- LAMPORT, L. A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 5, 1987.
- LÉVÉNEZ, E. *Unix History*. Disponível na INTERNET via www.levenez.com/unix/. Consultado em 2002.
- LEVY, H. M. ECKHOUSE, R. H. *Computer Programming and Architecture: The VAX-11*. Dec., 1980.
- LINUX. *Linux On-line*. Disponível na INTERNET via www.linux.org. Consultado em 2002.
- MCGUIRE, M. R., EISENBERG, M. A. Further Comments on Dijkstra's Concurrent Programming Control Problem. *CACM*, 1972.
- Microsoft® Windows® Internals, Fourth Edition: Microsoft Windows Server™ 2003, Windows XP, and Windows 2000. By Mark E. Russinovich and David A. Solomon.
- MILLER, D. D. *VAX/VMS Operating System Concepts*. Dec., 1992.

MINIX. *Minix Information Sheet*. Disponível na INTERNET via www.cs.vu.nl/~ast/minix.html. Consultado em 2002.

NETO, J. J. *Introdução à Compilação*. LTC, 1987.

NICHOLS, B., BUTTAR, D., FIRRELL, J. *Pthreads Programming*. O'Reilly Press, 1996.

PATTERSON, D. A., HENNESSY, J. L. *Computer Architecture—A Quantitative Approach*. 5. ed. Morgan Kaufman Publishers, 2012.

PATTERSON, D. A., HENNESSY, J. L. *Computer Organization and Design – The Hardware/Software Interface*. 4. ed. Morgan Kaufman Publishers, 2012.

PETERSON, G. L. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, vol. 12, p. 115-116, June, 1981.

PETERSON, G. L. A New Solution to Lamport's Concurrent Programming Problem Using Small Shared Variables. *ACM Trans. Program. Lang. And Syst.*, 5, 1983.

PFISTER, G. F. *In Search of Clusters*. Prentice-Hall, 1998.

PHAM, T. Q., GANG, P. K. *Multithreaded Programming with Windows NT*. Prentice-Hall, 1996.

PROTIC, J., TOMASEVIC, M., MILUTINOVIC, V. *Distributed Shared Memory: Concepts and Systems*. IEEE Computer Society, 1998.

RHEINGOLD, H. *Tools for Thought*. Prentice-Hall, 1985.

RICHTER, J. *Advanced Windows*. 3. ed. Microsoft Press, 1997.

RITCHIE, D., THOMPSON, K. The Unix Timesharing System. *CACM*, vol. 17, p. 365-375, July, 1974.

ROBBINS, K. A., ROBBINS, S. *Practical Unix Programming: A Guide to Concurrency, Communication, and Multithreading*. Prentice-Hall, 1996.

ROSEN, S. Programming Systems and Languages – A Historical Survey. *Proceedings Eastern Joint Computer Conference, AFIPS*, vol. 25, 1964.

ROSEN, S. Electronic Computers – A Historical Survey. *Computing Surveys*, Vol. 1, n. 1, March 1969.

SILBERSCHATZ, A., GALVIN, P. B. *Operating System Concepts*. 5. ed. Addison-Wesley, 1998.

SIMON, R. J. *Windows NT Win32 SuperBible*: Wait Group Press, 1997.

SOLOMON, D. A. *Inside Windows NT*. 2. ed. Microsoft Press, 1999.

SPEC. Disponível na INTERNET via www.spec.org. Consultado em 2001.

STALLINGS, W. *Operating Systems: Internal and Design Principles*. 3. ed. Prentice-Hall, 1997.

STALLINGS, W. *Computer Organization and Architecture – Design for Performance*. 9. ed. Prentice-Hall, 2012.

STALLINGS, W. *Data and Computer Communications*. 6. ed. Prentice-Hall, 1999.

TANENBAUM, A. S. *Operation Systems: Design and Implementation*. Prentice-Hall, 1987.

TANENBAUM, A. S. *Distributed Operating Systems*. Prentice-Hall, 1995.

TANENBAUM, A. S. *Computer Networks*. 2. ed. Prentice-Hall, 1996.

TANENBAUM, A. S. *Modern Operating Systems*. 2. ed. Prentice-Hall, 2001.

TANENBAUM, A. S. *Organização Estruturada de Computadores*. 5. ed. Pearson Education, 2007.

TROPIX. Disponível na INTERNET via www.tropix.nce.ufrj.br. Consultado em 2002.

VAHALIA, U. *Unix Internal: The New Frontiers*. Prentice-Hall, 1996.

WEIZER, N. A History of Operating System. *Datamation*, Jan., 1981. p. 119-126.

