

# SISTEMAS OPERACIONAIS

---

APOSTILA por Lucília Ribeiro



*“Nada lhe posso dar que já não exista em você mesmo. Não posso abrir-lhe outro mundo de imagens, além daquele que há em sua própria alma. Nada lhe posso dar a não ser a oportunidade, o impulso, a chave. Eu o ajudarei a tornar visível o seu próprio mundo, e isso é tudo.”*

*(Hermann Hesse)*

# Índice

*"A alma tem terrenos infranqueáveis, onde retemos nossos sonhos dourados ou um tenebroso dragão. Assim fazemos por medo de que ambos nos escapem, temendo que nos roubem o ouro ou nos devore o dragão"* (Ralph Emerson)

<b>ÍNDICE .....</b>	<b>2</b>
<b>VISÃO GERAL .....</b>	<b>7</b>
1.1    INTRODUÇÃO .....	7
1.2    O SISTEMA OPERACIONAL - FUNCIONALIDADES .....	7
1.2.2 <i>O SISTEMA OPERACIONAL VISTO COMO UM GERENTE DE RECURSOS</i> .....	7
1.3    MÁQUINA VIRTUAL .....	8
1.4    HISTÓRIA DOS SISTEMAS OPERACIONAIS.....	8
1.4.1 <i>DÉCADA DE 1940 (Ausência de SOs)</i> .....	9
1.4.2 <i>DÉCADA DE 1950 (Processamento em Batch)</i> .....	9
1.4.3 <i>DÉCADA DE 1960 (Processamento Time-sharing)</i> .....	9
1.4.4 <i>DÉCADA DE 1970 (Multiprocessamento)</i> .....	9
1.4.5 <i>DÉCADA DE 1980 (Computadores Pessoais)</i> : .....	10
1.4.6 <i>DÉCADA DE 1990 (Sistemas Distribuídos)</i> : .....	10
1.5    CLASSIFICAÇÃO DOS SISTEMAS OPERACIONAIS .....	10
1.5.1 <i>SISTEMAS MONOPROGRAMÁVEIS OU MONOTAREFAS</i> .....	10
1.5.2 <i>SISTEMAS MULTIPROGRAMÁVEIS OU MULTITAREFAS</i> .....	11
a) <i>Sistemas Batch</i> .....	11
b) <i>Sistemas de Tempo Compartilhado</i> .....	11
c) <i>Sistemas de Tempo Real</i> .....	12
1.5.3 <i>SISTEMAS COM MÚLTIPOS PROCESSADORES</i> .....	12
1.6    EXERCÍCIOS .....	14
<b>CONCEITOS DE HARDWARE E SOFTWARE .....</b>	<b>16</b>
2.1    INTRODUÇÃO .....	16
2.2    UNIDADES MÉTRICAS.....	16
2.3    HARDWARE.....	16
2.3.1 <i>PROCESSADOR</i> .....	17
2.3.2 <i>MEMÓRIA PRINCIPAL</i> .....	17
2.3.3 <i>MEMÓRIA CACHE</i> .....	18
2.3.4 <i>MEMÓRIA SECUNDÁRIA</i> .....	19
2.3.5 <i>DISPOSITIVOS DE ENTRADA/SAÍDA</i> .....	19
2.3.6 <i>BARRAMENTO</i> .....	20
2.3.7 <i>PIPELINING</i> .....	21
2.3.8 <i>ARQUITETURAS RISC e CISC</i> .....	22
2.3.9 <i>ANÁLISE DE DESEMPENHO</i> .....	22
2.4    SOFTWARE .....	23
2.4.1 <i>TRADUTOR</i> .....	23
2.4.2 <i>INTERPRETADOR</i> .....	24
2.4.3 <i>LINKEDITOR</i> .....	24
2.4.4 <i>INTERPRETADOR DE COMANDOS e LINGUAGEM DE CONTROLE</i> .....	24
2.4.5 <i>ATIVAÇÃO / DESATIVAÇÃO DO SISTEMA</i> .....	24
2.5    EXERCÍCIOS .....	25
<b>CONCORRÊNCIA .....</b>	<b>27</b>
3.1    INTRODUÇÃO .....	27
3.2    INTERRUPÇÃO E EXCEÇÃO.....	28
3.3    OPERAÇÕES DE ENTRADA/SAÍDA .....	29
3.4    BUFERIZAÇÃO .....	30
3.5    SPOOLING .....	30
3.6    REENTRÂNCIA.....	31
3.7    EXERCÍCIOS .....	31

<b>ESTRUTURA DO SISTEMA OPERACIONAL.....</b>	<b>32</b>
4.1    INTRODUÇÃO .....	32
4.2    CHAMADAS DE SISTEMA (SYSTEM CALLS) .....	32
4.3    MODOS DE ACESSO .....	33
4.4    ARQUITETURA MONOLÍTICA.....	34
4.5    ARQUITETURA DE CAMADAS.....	34
4.6    MÁQUINA VIRTUAL .....	35
4.7    EXERCÍCIOS .....	35
<b>PROCESSOS .....</b>	<b>36</b>
5.1    INTRODUÇÃO .....	36
5.2    ESTRUTURA DO PROCESSO .....	36
5.2.1    CONTEXTO DE HARDWARE .....	37
5.2.2    CONTEXTO DE SOFTWARE .....	37
5.2.3    ESPAÇO DE ENDEREÇAMENTO .....	38
5.2.4    BLOCO DE CONTROLE DE PROCESSO .....	39
5.3    ESTADOS DO PROCESSO .....	39
5.4    MUDANÇA DE ESTADO DO PROCESSO .....	39
5.5    CRIAÇÃO E ELIMINAÇÃO DE PROCESSOS.....	40
5.6    PROCESSOS INDEPENDENTES, SUBPROCESSOS E THREADS .....	41
5.7    PROCESSOS PRIMEIRO ( <i>FOREGROUND</i> ) E SEGUNDO PLANO ( <i>BACKGROUND</i> ) .....	42
5.8    PROCESSOS LIMITADOS POR CPU (CPU-BOUND) E POR E/S (I/O-BOUND).....	43
5.9    SELEÇÃO DE PROCESSOS.....	43
5.9.1    FILAS PARA SELEÇÃO DE PROCESSOS .....	43
5.9.2    ESCALONADORES.....	43
5.10    EXERCÍCIOS .....	44
<b>THREADS.....</b>	<b>46</b>
6.1    INTRODUÇÃO .....	46
6.2    THREADS .....	46
6.3    AMBIENTE MONOTHREAD .....	47
6.4    AMBIENTE MULTITHREAD.....	47
6.5    ARQUITETURA E IMPLEMENTAÇÃO .....	49
6.5.1    THREADS EM MODO USUÁRIO .....	49
6.5.2    THREADS EM MODO KERNEL .....	50
6.5.3    THREADS EM MODO HÍBRIDO .....	50
6.6    MODELOS DE PROGRAMAÇÃO.....	51
6.7    EXERCÍCIOS .....	51
<b>SINCRONIZAÇÃO E COMUNICAÇÃO ENTRE PROCESSOS .....</b>	<b>52</b>
7.1    INTRODUÇÃO .....	52
7.2    SINCRONIZAÇÃO .....	52
7.3    ESPECIFICAÇÃO DE CONCORRÊNCIA EM PROGRAMAS.....	54
7.4    PROBLEMAS DE COMPARTILHAMENTO DE RECURSOS.....	54
7.5    EXCLUSÃO MÚTUA .....	56
7.6    SOLUÇÕES DE HARDWARE .....	56
7.6.1    DESABILITAÇÃO DAS INTERRUPÇÕES.....	56
7.6.2    INSTRUÇÃO TEST-AND-SET.....	57
7.7    SOLUÇÕES DE SOFTWARE .....	58
7.7.1    PRIMEIRO ALGORITMO: ESTRITA ALTERNÂNCIA .....	58
7.7.2    SEGUNDO ALGORITMO:.....	58
7.7.3    TERCEIRO ALGORITMO: .....	59
7.7.4    QUARTO ALGORITMO: .....	60
7.7.5    ALGORITMO DE PETERSON.....	60
7.8    PROBLEMA DO PRODUTOR-CONSUMIDOR OU SINCRONIZAÇÃO CONDICIONAL .....	61
7.9    SEMÁFOROS .....	63
7.9.1    EXCLUSÃO MÚTUA UTILIZANDO SEMÁFOROS.....	63
7.9.2    SINCRONIZAÇÃO CONDICIONAL UTILIZANDO SEMÁFOROS .....	64
7.10    MONITORES .....	65
7.10.1    EXCLUSÃO MÚTUA UTILIZANDO MONITORES.....	65
7.10.2    SINCRONIZAÇÃO CONDICIONAL UTILIZANDO MONITORES .....	66
7.11    TROCA DE MENSAGENS .....	68

7.12	PROBLEMAS CLÁSSICOS DE SINCRONIZAÇÃO .....	68
7.12.1	<i>PROBLEMA DOS FILÓSOFOS GLUTÕES</i> .....	68
7.12.2	<i>PROBLEMA DO BARBEIRO DORMINHO CO</i> .....	69
7.13	EXERCÍCIOS .....	69
	<b>DEADLOCK.....</b>	<b>72</b>
8.1	INTRODUÇÃO .....	72
8.2	EXEMPLOS DE DEADLOCKS .....	72
8.2.1	<i>DEADLOCK DE TRÁFEGO</i> .....	72
8.2.2	<i>DEADLOCK SIMPLES DE RECURSOS</i> .....	72
8.2.3	<i>DEADLOCK EM SISTEMAS DE SPOOLING</i> .....	72
8.2.4	<i>ADIAMENTO INDEFINIDO</i> .....	73
8.3	RECURSOS .....	73
8.4	QUATRO CONDIÇÕES NECESSÁRIAS PARA DEADLOCKS .....	74
8.5	O MODELO DO DEADLOCK .....	74
8.6	MÉTODOS PARA LIDAR COM DEADLOCKS.....	76
8.7	O ALGORITMO DO AVESTRUZ .....	76
8.8	DETECÇÃO DE DEADLOCKS .....	76
8.8.1	<i>DETECÇÃO DO DEADLOCK COM UM RECURSO DE CADA TIPO</i> .....	76
8.8.2	<i>DETECÇÃO DO DEADLOCK COM VÁRIOS RECURSOS DE CADA TIPO</i> .....	77
8.9	RECUPERAÇÃO DE DEADLOCKS.....	78
8.9.1	<i>RECUPERAÇÃO ATRAVÉS DA PREEMPÇÃO</i> .....	78
8.9.2	<i>RECUPERAÇÃO ATRAVÉS DE VOLTA AO PASSADO</i> .....	78
8.9.3	<i>RECUPERAÇÃO ATRAVÉS DE ELIMINAÇÃO DE PROCESSOS</i> .....	78
8.10	TENTATIVAS DE EVITAR O DEADLOCK .....	79
8.10.1	<i>ESTADOS SEGUROS E INSEGUROS</i> .....	79
8.10.2	<i>ALGORITMO DO BANQUEIRO PARA UM ÚNICO TIPO DE RECURSO</i> .....	79
8.11	PREVENÇÃO DE DEADLOCKS .....	80
8.11.1	<i>ATACANDO O PROBLEMA DA EXCLUSÃO MÚTUAS</i> .....	80
8.11.2	<i>ATACANDO O PROBLEMA DA POSSE E DA ESPERA</i> .....	80
8.11.3	<i>ATACANDO O PROBLEMA DA CONDIÇÃO DE NÃO-PREEMPÇÃO</i> .....	80
8.11.4	<i>ATACANDO O PROBLEMA DA ESPERA CIRCULAR</i> .....	80
8.12	EXERCÍCIOS .....	81
	<b>GERÊNCIA DO PROCESSADOR .....</b>	<b>83</b>
9.1	INTRODUÇÃO .....	83
9.2	FUNÇÕES BÁSICAS .....	83
9.3	CRITÉRIOS DE ESCALONAMENTO .....	83
9.4	ESTRATÉGIAS DE ESCALONAMENTO .....	84
9.5	ESCALONAMENTO FIRST COME FIRST SERVED (FCFS) OU FIFO .....	84
9.6	ESCALONAMENTO MENOR JOB PRIMEIRO OU SJF (SHORTEST JOB FIRST) .....	85
9.7	ESCALONAMENTO CIRCULAR OU ROUND ROBIN .....	86
9.8	ESCALONAMENTO POR PRIORIDADES .....	87
9.9	ESCALONAMENTO POR MÚLTIPHAS FILAS .....	88
9.10	EXERCÍCIOS .....	89
	<b>GERÊNCIA DE MEMÓRIA .....</b>	<b>93</b>
10.1	INTRODUÇÃO .....	93
10.2	FUNÇÕES BÁSICAS .....	93
10.3	ALOCAÇÃO CONTÍGUA SIMPLES .....	93
10.4	TÉCNICA DE <i>OVERLAY</i> .....	93
10.5	ALOCAÇÃO PARTICIONADA .....	94
10.5.1	<i>ALOCAÇÃO PARTICIONADA ESTÁTICA</i> .....	94
10.5.2	<i>ALOCAÇÃO PARTICIONADA DINÂMICA</i> .....	95
10.5.3	<i>ESTRATÉGIAS DE ALOCAÇÃO DE PARTIÇÃO</i> .....	97
10.6	SWAPPING .....	99
10.7	EXERCÍCIOS .....	100
	<b>MEMÓRIA VIRTUAL.....</b>	<b>103</b>
11.1	INTRODUÇÃO .....	103
11.2	ESPAÇO DE ENDEREÇAMENTO VIRTUAL .....	103
11.3	MAPEAMENTO .....	104
11.4	PAGINAÇÃO .....	104

11.4.1	<i>PAGINAÇÃO MULTINÍVEL</i> .....	107
11.4.2	<i>POLÍTICAS DE BUSCA DE PÁGINAS</i> .....	108
11.4.3	<i>POLÍTICAS DE ALOCAÇÃO DE PÁGINAS</i> .....	109
11.4.4	<i>POLÍTICAS DE SUBSTITUIÇÃO DE PÁGINAS</i> .....	109
11.4.5	<i>WORKING SET</i> .....	110
11.4.6	<i>ALGORITMOS DE SUBSTITUIÇÃO DE PÁGINAS</i> .....	111
11.5	SEGMENTAÇÃO.....	114
11.6	SEGMENTAÇÃO PAGINADA .....	115
11.7	EXERCÍCIOS .....	116
	<b>SISTEMA DE ARQUIVOS.....</b>	<b>122</b>
12.1	INTRODUÇÃO .....	122
12.2	ARQUIVOS .....	122
12.2.1	<i>MÉTODO DE ACESSO</i> .....	122
12.2.2	<i>IMPLEMENTAÇÃO DE ARQUIVOS</i> .....	123
12.3	MÉTODOS DE ALOCAÇÃO.....	124
12.3.1	<i>ALOCAÇÃO CONTÍGUA</i> .....	124
12.3.2	<i>ALOCAÇÃO COM LISTA LIGADA</i> .....	125
12.3.3	<i>ALOCAÇÃO COM LISTA LIGADA USANDO UM ÍNDICE</i> .....	125
12.4	GERÊNCIA DE ESPAÇO EM DISCO .....	126
12.5	CONTROLE DE BLOCOS LIVRES .....	126
12.5.1	<i>MAPA DE BITS</i> .....	126
12.5.2	<i>LISTA ENCADEADA</i> .....	126
12.5.3	<i>AGRUPAMENTO</i> .....	126
12.5.4	<i>CONTADORES</i> .....	126
12.6	CONSISTÊNCIA DO SISTEMA DE ARQUIVOS .....	127
12.6.1	<i>CONSISTÊNCIA EM BLOCOS</i> .....	127
12.7	PERFORMANCE DO SISTEMA DE ARQUIVOS.....	127
12.7.1	<i>CACHE</i> .....	127
12.7.2	<i>LEITURA ANTECIPADA DE BLOCOS</i> .....	127
12.7.3	<i>REDUÇÃO DO MOVIMENTO DO BRAÇO DO DISCO</i> .....	128
12.8	EXERCÍCIOS .....	128
	<b>GERÊNCIA DE DISPOSITIVOS.....</b>	<b>129</b>
13.1	INTRODUÇÃO .....	129
13.2	ACESSO AO SUBSISTEMA DE ENTRADA E SAÍDA.....	129
13.3	SUBSISTEMA DE ENTRADA E SAÍDA .....	130
13.4	DEVICE DRIVERS .....	131
13.5	CONTROLADORES .....	131
13.6	DISPOSITIVOS DE ENTRADA E SAÍDA .....	132
13.7	DISCOS RÍGIDOS .....	132
13.7.1	<i>TEMPO DE ACESSO</i> .....	133
13.7.2	<i>ENTRELAÇAMENTO (interleaving)</i> .....	133
13.7.3	<i>ESCALONAMENTO DO BRAÇO DO DISCO</i> .....	134
13.8	EXERCÍCIOS .....	136
	<b>MULTIMÍDIA.....</b>	<b>137</b>
14.1	INTRODUÇÃO .....	137
14.2	ARQUIVOS MULTIMÍDIA .....	138
14.2.1	<i>CODIFICAÇÃO DE ÁUDIO</i> .....	139
14.2.2	<i>CODIFICAÇÃO DE VÍDEO</i> .....	139
14.3	COMPRESSÃO DE VÍDEO.....	140
14.3.1	<i>O PADRÃO JPEG</i> .....	140
14.3.2	<i>O PADRÃO MPEG</i> .....	142
14.4	ESCALONAMENTO DE PROCESSOS MULTIMÍDIA .....	143
14.4.1	<i>ESCALONAMENTO DE PROCESSOS HOMOGÊNEOS</i> .....	143
14.4.2	<i>ESCALONAMENTO GERAL DE TEMPO REAL</i> .....	144
14.4.3	<i>ESCALONAMENTO POR TAXA MONOTÔNICA</i> .....	145
14.4.4	<i>ESCALONAMENTO PRAZO MAIS CURTO PRIMEIRO</i> .....	146
14.5	PARADIGMAS DE SISTEMAS DE ARQUIVOS MULTIMÍDIA .....	147
14.5.1	<i>FUNÇÕES DE CONTROLE VCR</i> .....	147
14.5.2	<i>VÍDEO QUASE SOB DEMANDA</i> .....	148
14.6	ALOCAÇÃO DE ARQUIVOS EM DISCOS .....	149

14.6.1	<i>ALOCAÇÃO DE UM ARQUIVO EM UM ÚNICO DISCO</i>	149
14.6.2	<i>DUAS ESTRATÉGIAS ALTERNATIVAS DE ORGANIZAÇÃO DE ARQUIVOS</i>	150
14.6.3	<i>ALOCAÇÃO DE ARQUIVOS PARA VÍDEO QUASE SOB DEMANDA</i>	151
14.7	<b>EXERCÍCIOS</b>	151
	<b>SEGURANÇA.....</b>	<b>153</b>
15.1	<b>INTRODUÇÃO .....</b>	<b>153</b>
15.2	<b>CRİPTOGRAFIA.....</b>	<b>153</b>
15.2.1	<i>CRİPTOGRAFIA POR CHAVE SECRETA .....</i>	154
15.2.2	<i>CRİPTOGRAFIA POR CHAVE PÚBLICA.....</i>	156
15.3	<b>AUTENTICAÇÃO.....</b>	<b>157</b>
15.3.1	<i>AUTENTICAÇÃO BÁSICA.....</i>	158
15.3.2	<i>BIOMETRIA E CARTÕES INTELIGENTES.....</i>	159
15.3.3	<i>KERBEROS.....</i>	160
15.3.4	<i>ASSINATURA ÚNICA.....</i>	160
15.4	<b>CONTROLE DE ACESSO .....</b>	<b>161</b>
15.4.1	<i>DIREITO DE ACESSO E DOMÍNIOS DE PROTEÇÃO.....</i>	161
15.4.2	<i>MODELOS E POLÍTICAS DE CONTROLE DE ACESSO.....</i>	162
15.4.3	<i>MECANISMOS DE CONTROLE DE ACESSO.....</i>	163
15.5	<b>ATAQUES À SEGURANÇA .....</b>	<b>165</b>
15.5.1	<i>CRİPTOANÁLISE.....</i>	165
15.5.2	<i>VÍRUS E VERMES.....</i>	165
15.5.3	<i>ATAQUES DE RECUSA DE SERVIÇO (DoS).....</i>	167
15.5.4	<i>EXPLORAÇÃO DE SOFTWARE.....</i>	168
15.5.5	<i>INVASÃO DE SISTEMA.....</i>	168
15.6	<b>PREVENÇÃO DE ATAQUES E SOLUÇÕES DE SEGURANÇA.....</b>	<b>169</b>
15.6.1	<i>FIREWALLS .....</i>	169
15.6.2	<i>SISTEMAS DE DETECÇÃO DE INTRUSOS (IDSs).....</i>	170
15.6.3	<i>SOFTWARE ANTIVÍRUS.....</i>	170
15.6.4	<i>CORREÇÕES DE SEGURANÇA.....</i>	172
15.6.5	<i>SISTEMA DE ARQUIVOS SEGUROS.....</i>	172
15.6.6	<i>O LIVRO LARANJA DA SEGURANÇA.....</i>	173
15.7	<b>COMUNICAÇÃO SEGURA .....</b>	<b>173</b>
15.8	<b>ESTEGANOGRAFIA .....</b>	<b>174</b>
15.9	<b>ESTUDO DE CASO: SEGURANÇA DE SISTEMAS UNIX .....</b>	<b>174</b>
15.10	<b>EXERCÍCIOS .....</b>	<b>176</b>
	<b>MULTIPROCESSAMENTO .....</b>	<b>179</b>
16.1	<b>INTRODUÇÃO .....</b>	<b>179</b>
16.2	<b>ARQUITETURA DE MULTIPROCESSADOR.....</b>	<b>179</b>
16.2.1	<i>CLASSIFICAÇÃO DE ARQUITETURAS SEQUENCIAIS E PARALELAS.....</i>	179
16.2.2	<i>ESQUEMA DE INTERCONEXÃO DE PROCESSADORES .....</i>	180
16.2.3	<i>SISTEMAS FRACAMENTE x FRACAMENTE ACOPLADOS .....</i>	184
16.3	<b>ORGANIZAÇÃO DE SISTEMAS OPERACIONAIS MULTIPROCESSADORES .....</b>	<b>185</b>
16.3.1	<i>MESTRE/ESCRAVO .....</i>	186
16.3.2	<i>NÚCLEOS SEPARADOS.....</i>	186
16.3.3	<i>ORGANIZAÇÃO SIMÉTRICA .....</i>	187
16.4	<b>ARQUITETURAS DE ACESSO À MEMÓRIA .....</b>	<b>187</b>
16.4.1	<i>ACESSO UNIFORME À MEMÓRIA .....</i>	188
16.4.2	<i>ACESSO NÃO UNIFORME À MEMÓRIA .....</i>	188
16.4.3	<i>ARQUITETURA DE MEMÓRIA SOMENTE DE CACHE .....</i>	189
16.4.4	<i>SEM ACESSO À MEMÓRIA REMOTA .....</i>	190
16.5	<b>COMPARTILHAMENTO DE MEMÓRIA EM MULTIPROCESSADORES .....</b>	<b>191</b>
16.5.1	<i>COERÊNCIA DE CACHE .....</i>	191
16.5.2	<i>REPLICAÇÃO E MIGRAÇÃO DE PÁGINAS .....</i>	192
16.5.3	<i>MEMÓRIA VIRTUAL COMPARTILHADA .....</i>	193
16.6	<b>ESCALONAMENTO DE MULTIPROCESSADORES .....</b>	<b>194</b>
16.6.1	<i>ESCALONAMENTO DE MULTIPROCESSADORES CEGOS AO JOB .....</i>	195
16.6.2	<i>ESCALONAMENTO DE MULTIPROCESSADORES CIENTES AO JOB .....</i>	196
16.7	<b>MIGRAÇÃO DE PROCESSOS .....</b>	<b>198</b>
16.7.1	<i>FLUXO DE MIGRAÇÃO DE PROCESSOS .....</i>	198
16.7.2	<i>CONCEITOS DE MIGRAÇÃO DE PROCESSOS .....</i>	199
16.7.3	<i>ESTRATÉGIAS DE MIGRAÇÃO DE PROCESSOS .....</i>	200

16.8	BALANCEAMENTO DE CARGA .....	201
16.8.1	<i>BALANCEAMENTO ESTÁTICO DE CARGA</i> .....	201
16.8.2	<i>BALANCEAMENTO DINÂMICO DE CARGA</i> .....	202
16.9	EXCLUSÃO MÚTUA EM MULTIPROCESSADORES .....	204
16.9.1	<i>TRAVAS GIRATÓRIAS</i> .....	205
16.9.2	<i>TRAVAS DORMIR/ACORDAR</i> .....	205
16.9.3	<i>TRAVAS DE LEITURA/ESCRITA</i> .....	206
16.10	EXERCÍCIOS .....	206
	<b>BIBLIOGRAFIA.....</b>	<b>210</b>

# 1

## Visão Geral

*“As coisas são sempre melhores no começo” (Blaise Pascal)*

### 1.1 INTRODUÇÃO

Um computador sem software nada seria. O software pode ser dividido, a grosso modo, em duas categorias: os programas do sistema, que gerenciam a operação do próprio computador, e os programas de aplicação, que resolvem problemas para seus usuários. O mais importante dos programas de sistema é o **sistema operacional**, que controla todos os recursos do computador, e fornece a base sobre a qual os programas aplicativos são escritos.

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.

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.

### 1.2 O SISTEMA OPERACIONAL - FUNCIONALIDADES

O Sistema Operacional é o *software* que controla todos os recursos do computador e fornece a base sobre a qual os programas aplicativos são escritos e suas principais funcionalidades são: Máquina Virtual ou Estendida e Gerente de Recursos.

#### 1.2.1 O SISTEMA OPERACIONAL VISTO COMO UMA MÁQUINA ESTENDIDA

Programar em nível de arquitetura é uma tarefa ingrata para qualquer usuário de um sistema de computador. Para tratar da entrada/saída numa unidade de disco flexível, por exemplo, o usuário deveria trabalhar com comandos relativos a: leitura/escrita de dados, movimento da cabeça, formatação de trilhas e inicialização, sensoriamento, reinicialização e calibração do controlador e do driver de disquete.

É óbvio dizer que o programador não vai querer lidar com esse tipo de detalhes, desejando portanto uma abstração de alto nível. Assim, deve haver um programa que esconda do usuário o verdadeiro hardware e apresente-lhe um esquema simples de arquivos identificados que possam ser lidos ou escritos. Tal programa é o sistema operacional. Do ponto de vista de máquina estendida, o sistema operacional trata ainda de outras questões incômodas como: interrupções, temporizadores, gerência de memória etc.

A função do sistema operacional, nesse ponto de vista, é apresentar ao usuário uma máquina virtual equivalente ao hardware, porém muito mais fácil de programar.

#### 1.2.2 O SISTEMA OPERACIONAL VISTO COMO UM GERENTE DE RECURSOS

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.

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.

Assim, o sistema operacional deve gerenciar o uso dos recursos, contabilizando o uso de cada um e garantindo acesso ordenado de usuários a recursos através da mediação de conflitos entre as diversas requisições existentes.

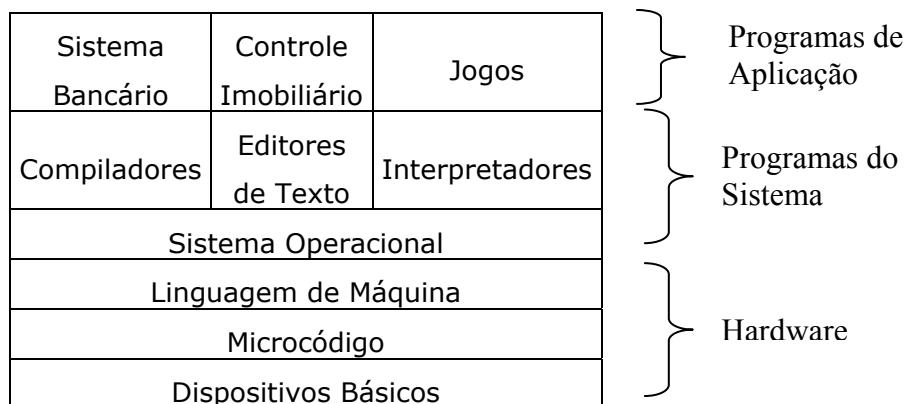
### 1.3 MÁQUINA VIRTUAL

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 através do software que serviços são oferecidos ao usuário, como armazenamento de dados em discos, impressão de relatórios, geração de gráficos, 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.

Nos primeiros computadores, a programação era realizada em painéis através de fios, exigindo, consequentemente, um grande conhecimento da arquitetura do hardware e da sua linguagem de máquina. Isso era uma grande dificuldade para os programadores da época.

Para afastar o usuário da complexidade do hardware, foi encontrada uma forma: colocar uma camada de software em cima do hardware para gerenciar todos os componentes do sistema, aparecendo ao usuário como uma interface muito simples de entender e programar. Tal interface é a **máquina virtual**, e a camada de software é o **Sistema Operacional**.



Num sistema computacional podem atuar diferentes usuários, cada qual tentando solucionar seus problemas. Assim, há uma diversidade enorme de programas aplicativos. O sistema operacional controla e coordena o uso do hardware entre os vários programas aplicativos, para os vários usuários.

### 1.4 HISTÓRIA DOS SISTEMAS OPERACIONAIS

A evolução dos Sistemas Operacionais está intimamente relacionada com o desenvolvimento dos computadores. Objetivando permitir um conhecimento geral, o histórico dos Sistemas Operacionais aqui apresentados será dividido em décadas. Em cada uma das

décadas serão discutidas as principais características do *hardware* e do Sistema Operacional da época.

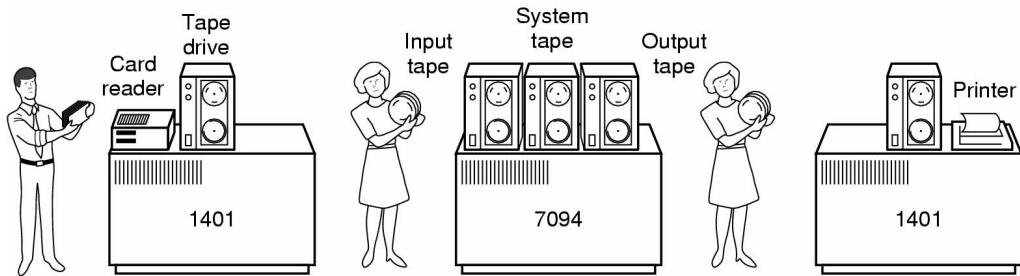
#### **1.4.1 DÉCADA DE 1940 (Ausência de SOs)**

A Segunda Guerra Mundial acelerou as pesquisas para o desenvolvimento dos primeiros computadores (Mark I, ENIAC etc.), objetivando dinamizar o processo de realização de cálculos. Os computadores, então desenvolvidos, eram baseados em válvulas. Eles ocupavam salas inteiras e não possuíam um SO. Com isso, programar, por exemplo, o ENIAC para realizar um determinado cálculo poderia levar dias, pois era necessário conhecer profundamente o funcionamento de seu hardware e utilizar linguagem de máquina;

#### **1.4.2 DÉCADA DE 1950 (Processamento em Batch)**

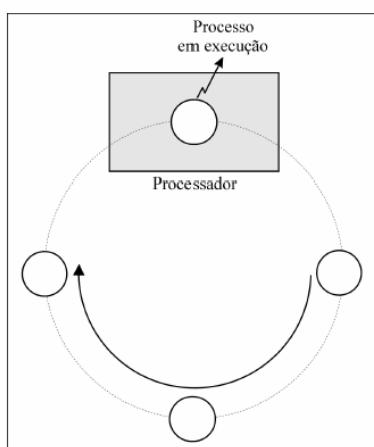
O desenvolvimento do transistor permitiu um grande avanço da informática. Assim, os computadores tornaram-se menores, mais confiáveis e mais rápidos. Nesta década observa-se o surgimento dos primeiros SOs e a programação das máquinas se dava através de cartões perfurados.

Os processos a serem executados entravam seqüencialmente no processador e rodavam até terminar. Este tipo de processamento ficou conhecido como processamento em batch ou lote, o qual é ilustrado pela figura abaixo:



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 vez só, o que diminuía o tempo de espera entre a execução dos programas, permitindo, assim, melhor aproveitamento do processador.

#### **1.4.3 DÉCADA DE 1960 (Processamento Time-sharing)**

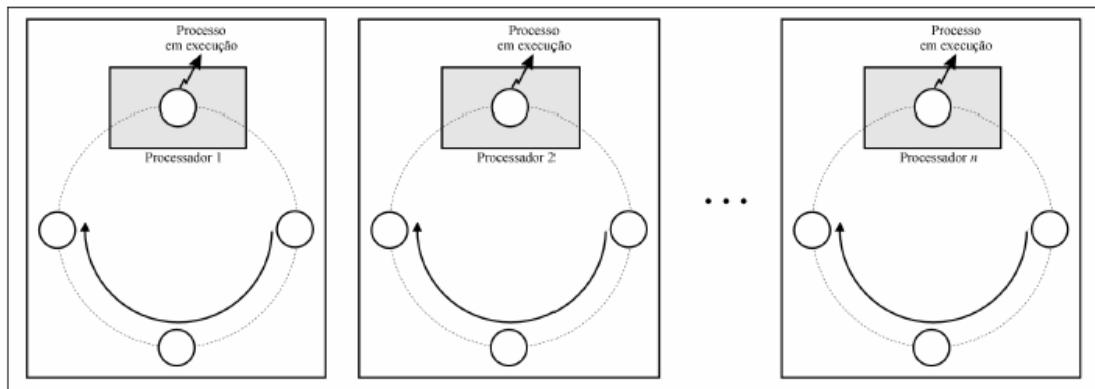


Nesta década entram em cena os circuitos integrados (CIs), o que permitiu que os computadores se tornassem mais baratos e, portanto, mais acessíveis às organizações empresariais. Na época em questão os processos não mais monopolizavam o uso do processador mas sim compartilhavam o mesmo. Dessa forma, o tempo de processamento passou a ser igualmente dividido entre os processos existentes. Esta técnica de processamento acabou ficando conhecida como processamento time-sharing ou processamento de tempo compartilhado.

#### **1.4.4 DÉCADA DE 1970 (Multiprocessamento)**

Nesta década a integração em larga escala (Large Scale Integration – LSI) e a integração em muito grande escala (Very Large Scale Integration - VLSI) permitiram uma

redução significativa no tamanho e no custo dos computadores. Com isso, os computadores com múltiplos processadores tornaram-se acessíveis e os processos passaram a ser executados em paralelo. Este tipo de processamento ficou conhecido como multiprocessamento, o qual é ilustrado pela Figura abaixo:



#### **1.4.5 DÉCADA DE 1980 (Computadores Pessoais):**

Os computadores pessoais tornam-se mais acessíveis comercialmente e a microcomputação se consolidou. Assim, surgiram os SOs voltados especificamente para os então chamados microcomputadores (ex.: CP/M6, MS-DOS7, OS/28 etc.). É interessante destacar que neste período a rápida evolução dos processadores dos computadores pessoais permitiu que seus SOs incorporassem importantes recursos como interface gráfica e multimídia. O baixo custo dos equipamentos permitiu, também, o surgimento das redes locais e com isso desenvolveram-se os Sistemas Operacionais de Rede (ex.: Novell Netware, LAN Manager etc.);

#### **1.4.6 DÉCADA DE 1990 (Sistemas Distribuídos):**

Durante a década de 1990 o crescimento das redes de computadores, especialmente a Internet, propiciou o surgimento de sistemas computacionais novos que se caracterizam por possuírem uma coleção de processadores que não compartilham memória ou barramento e que se comunicam via rede. Estes sistemas acabaram sendo chamados de sistemas fracamente acoplados que possibilitaram o surgimento dos sistemas distribuídos, onde diversos SOs presentes nos computadores de uma rede interagem o suficiente para dar a impressão de que existe um único Sistema Operacional controlando toda a rede e os recursos computacionais ligados a ela. Atualmente, os pesquisadores da área de Sistemas Operacionais concentram boa parte de seus esforços nos estudos sobre os Sistemas Operacionais Distribuídos.

### **1.5 CLASSIFICAÇÃO DOS SISTEMAS OPERACIONAIS**

A evolução dos sistemas operacionais acompanhou a evolução do hardware e das aplicações por ele suportadas. Muitos termos inicialmente introduzidos para definir conceitos e técnicas foram substituídos por outros. Isto fica muito claro quando tratamos da unidade de execução do processador. Inicialmente, eram utilizados os termos programas ou job, depois surgiu o conceito de processo, e agora, o conceito de *thread*.



#### **1.5.1 SISTEMAS MONOPROGRAMÁVEIS OU MONOTAREFAS**

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 se caracterizam por permitir que o processador, a memória e os periféricos permaneçam exclusivamente dedicados à execução de um único programa.

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.

Comparados a outros sistemas, os sistemas monoprogramáveis ou monotarefas são de simples implementação, não existindo muita preocupação com problemas decorrentes do compartilhamento de recursos.

### **1.5.2 SISTEMAS MULTIPROGRAMÁVEIS OU MULTITAREFAS**

Neste tipo de sistema, os recursos computacionais são compartilhados entre os diversos usuários e aplicações.

Neste caso, 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.

As vantagens do uso de tais sistemas são a redução do tempo de resposta das aplicações processadas no ambiente e de custos, a partir do compartilhamento dos diversos recursos do sistema entre as diferentes aplicações.

A partir do número de usuários que interagem com o sistema, podemos classificar os sistemas multiprogramáveis como monousuário ou multiusuário.

Os sistemas multiprogramáveis podem também 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.



#### **a) Sistemas Batch**

Foram os primeiros tipos de Sistemas Operacionais multiprogramáveis a serem implementados. O processamento em batch tem a característica de não exigir a interação do usuário com a aplicação. Todas as entradas e saídas de dados são implementadas por algum tipo de memória secundária, geralmente arquivos em disco.

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.

#### **b) Sistemas de Tempo Compartilhado**

Tais sistemas, também conhecidos como *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, esse é interrompido pelo Sistema Operacional e submetido por um outro, enquanto fica aguardando por uma nova fatia de tempo. O sistema cria um ambiente de trabalho próprio, dando a impressão de que todo o sistema está dedicado, exclusivamente, para cada usuário.

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.

A maioria das aplicações comerciais atualmente são processadas em sistemas de tempo compartilhado, que oferecem tempos baixos de respostas a seus usuários e menores custos, em função da utilização compartilhada dos diversos recursos do sistema.

### c) Sistemas de Tempo Real

Os sistemas de tempo real (*real-time*) diferem dos de tempo compartilhado no 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.

Não existe a idéia de fatia de tempo. Um programa utiliza o processador o tempo que for necessário ou até que apareça outro mais prioritário. Essa prioridade é definida pela própria aplicação. Estão presentes em aplicações de controle de usinas termoelétricas e nucleares, controle de tráfego aéreo, ou em qualquer outra aplicação onde o tempo de processamento é fator fundamental.

## 1.5.3 SISTEMAS COM MÚLTIPLOS PROCESSADORES

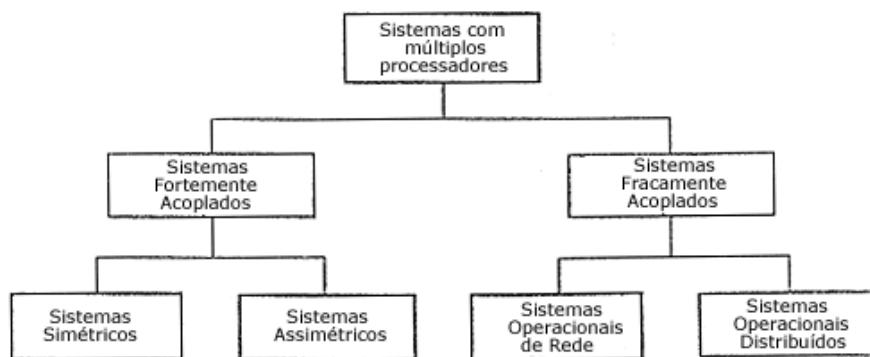
Um sistema com múltiplos processadores possui duas ou mais CPUs interligadas trabalhando em conjunto. A vantagem desse tipo de sistema é permitir que vários programas sejam executados ao mesmo tempo ou que um mesmo programa seja subdividido em partes para serem executadas simultaneamente em mais de um processador.

Os conceitos aplicados ao projeto de sistemas com múltiplos processadores incorporam os mesmos princípios básicos e benefícios apresentados na multiprogramação, além de outras vantagens específicas como escalabilidade, disponibilidade e balanceamento de carga.

**Escalabilidade** é a capacidade de ampliar o poder computacional do sistema apenas adicionando outros 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.

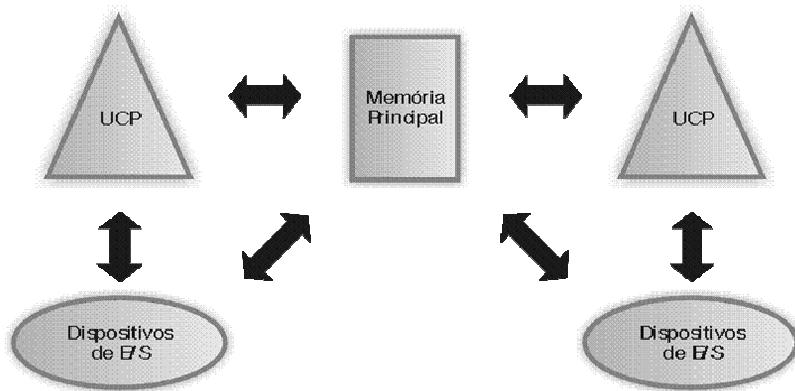
**Balanceamento de Carga** é a possibilidade de distribuir o processamento entre os diversos processadores da configuração a partir da carga de trabalho de cada processador, melhorando assim, o desempenho do sistema como um todo.



Esses sistemas podem ser classificados quanto à forma de comunicação entre as CPUs e quanto ao grau de compartilhamento da memória e dos dispositivos de E/S. Assim, podemos classificá-los em **fortemente acoplados** e **fracamente acoplados**.

### a) Sistemas Fortemente Acoplados

Num sistema fortemente acoplado dois ou mais processadores compartilham uma única memória e são controlados por um único sistema operacional. Um sistema fortemente acoplado é utilizado geralmente em aplicações que fazem uso intensivo da CPU e cujo processamento é dedicado à solução de um único problema.



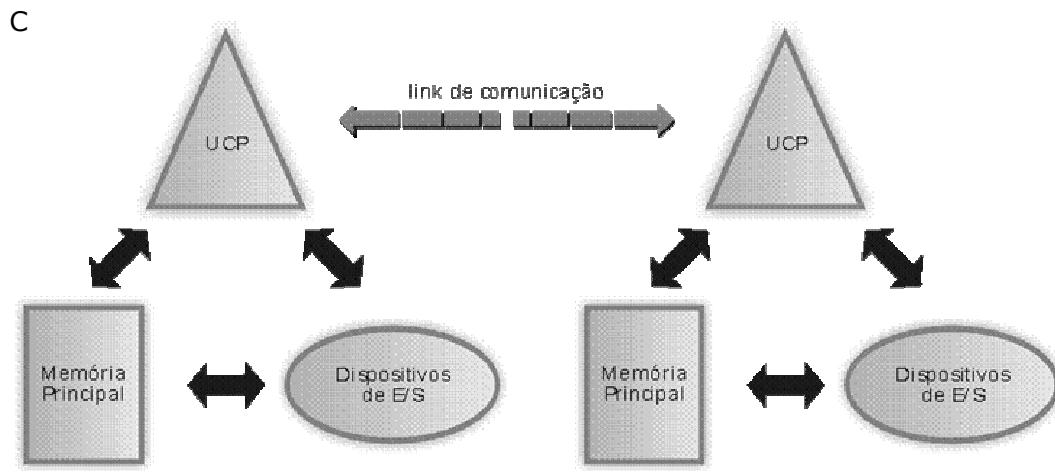
Os sistemas fortemente acoplados podem ser divididos em simétricos ou assimétricos. Os simétricos caracterizam-se pelo tempo uniforme de acesso à memória principal pelos diversos processadores.

Inicialmente, tais sistemas 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 Windows 2000, implementa esta funcionalidade.

### b) Sistemas Fracamente Acoplados

Num sistema fracamente acoplado dois ou mais sistemas de computação são conectados através do uso de linhas de comunicação. Nesses sistemas, ocorre o processamento distribuído entre os diversos computadores. Cada sistema funciona de forma independente, possuindo seu(s) próprio(s) processador(es). Em função destas características, também são conhecidos como *multicomputadores*.

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 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** permitem que um *host* compartilhe seus recursos, como 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.

Enquanto nos Sistemas Operacionais de Rede os usuários têm 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.

## 1.6 EXERCÍCIOS

- 1) Como seria utilizar um computador sem um Sistema Operacional?
- 2) O que é um Sistema Operacional? Fale sobre suas principais funções.
- 3) Defina o conceito de uma máquina de níveis ou camadas.
- 4) Quais os tipos de Sistemas Operacionais existentes?
- 5) Por que dizemos que existe uma subutilização de recursos em sistemas monoprogramáveis?
- 6) Qual a grande diferença entre sistemas monoprogramáveis e multiprogramáveis?
- 7) Quais as vantagens dos sistemas multiprogramáveis?
- 8) Um sistema monousuário pode ser um sistema multiprogramável? Dê um exemplo.
- 9) Quais são os tipos de sistemas multiprogramáveis?

- 10) O que caracteriza um sistema *batch*? Quais aplicações podem ser processadas neste tipo de ambiente?
- 11) Como os processos são executados em um sistema *time-sharing*? Quais as vantagens em utilizá-los?
- 12) Qual a grande diferença entre sistemas de tempo compartilhado e tempo real? Quais aplicações são indicadas para sistemas de tempo real?
- 13) O que são sistemas com múltiplos processadores e quais as vantagens em utilizá-los?
- 14) Qual a grande diferença entre sistemas fortemente e fracamente acoplados?
- 15) O que é um sistema fracamente acoplado? Qual a diferença entre Sistemas Operacionais de rede e Sistemas Operacionais distribuídos?
- 16) Cite dois exemplos de Sistemas Operacionais de rede.

-X-

2

# **Conceitos de Hardware e Software**

*“Nossa vida é desperdiçada em detalhes... Simplifique, simplifique.”* (Henry Thoreau)

## 2.1 INTRODUÇÃO

O Sistema Operacional está intimamente ligado ao hardware do computador no qual ele é executado. Ele estende o conjunto de instruções do computador e gerencia seus recursos. Para funcionar, ele deve ter um grande conhecimento do hardware, pelo menos do ponto de vista do programador.

Neste capítulo serão apresentados conceitos básicos de hardware e de software relativos à arquitetura de computadores e necessários para a compreensão dos demais tópicos.

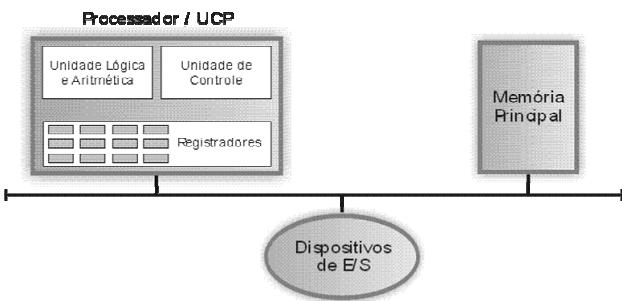
## 2.2 UNIDADES MÉTRICAS

Para evitar maior confusão, é melhor deixar claro, como na ciência da computação em geral, são usadas unidades métricas. Os prefixos métricos estão relacionados na tabela abaixo. Normalmente esses prefixos são abreviados por suas primeiras letras, com as unidades maiores que 1 em letras maiúsculas. Assim, um banco de dados de 1 TB ocupa  $10^{12}$  bytes de memória e um tique de relógio de 100 ps ocorre a cada  $10^{-10}$  segundos. Como ambos os prefixos, mili e micro, começam com a letra "m", foi necessário fazer-se uma escolha. Normalmente "m" é para mili e " $\mu$ " (a letra grega *my*) é para micro.

Convém também recordar que para medir tamanhos de memória, as unidades têm significados um pouco diferentes. O quilo corresponde a  $2^{10}$  (1024) e não a  $10^3$  (1000), pois as memórias são sempre expressas em potências de 2. Assim, uma memória de 1 KB contém 1024 bytes, e não 1000 bytes. De maneira similar, uma memória de 1 MB contém  $2^{20}$  (1.048.576) bytes e uma memória de 1 GB contém  $2^{30}$  (1.073.741.824) bytes. Contudo, uma linha de comunicação de 1 Kbps transmite a 1000 bits por segundo, uma rede local (LAN) de 10 Mbps transmite a 10.000.000 bits por segundo e um processador de 233 MHz executa 233.000 ciclos de clock por segundo, pois essas velocidades não são potências de 2.

## 2.3 HARDWARE

Conceitualmente, um computador pessoal simples pode ser abstraído para um modelo semelhante ao da figura ao lado. A CPU, a memória e os dispositivos de E/S estão todos conectados por um barramento, o qual proporciona a comunicação de uns com os outros.



### 2.3.1 PROCESSADOR

A CPU é o “cérebro” do computador. Ela busca instruções na memória e as executa: busca a primeira instrução da memória, decodifica para determinar seus operandos e qual operação executar com os mesmos, executa e busca, decodifica e executa as próximas operações.

Cada processador é composto por uma unidade de controle, unidade lógica e aritmética, e registradores. A **unidade de controle** 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**, 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 freqüência estável e bem determinada.

Cada CPU tem um conjunto específico de instruções que pode executar. Assim, um Pentium não executa programas SPARC e vice-versa. Como o tempo de acesso à memória é muito menor que o tempo para executá-la, todas as CPUs têm registradores internos para armazenar variáveis e resultados temporários. Por isso, o conjunto de instruções geralmente contém instruções para carregar uma palavra da memória em um registrador e armazenar uma palavra de um registrador na memória.

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 ao da memória principal. O número de registradores e sua capacidade de armazenamento variam em função da arquitetura de cada processador.

Além dos registradores de propósito geral, a maioria dos computadores tem vários registradores especiais visíveis ao programador. Um deles é o contador de programa, que contém o endereço de memória da próxima instrução a ser buscada.

Outro registrador especial é o ponteiro de pilha, que aponta para o topo da pilha atual na memória. A pilha contém uma estrutura para cada procedimento chamado, mas que ainda não encerrou. Uma estrutura de pilha do procedimento contém os parâmetros de entrada, as variáveis temporárias que não são mantidas nos registradores.

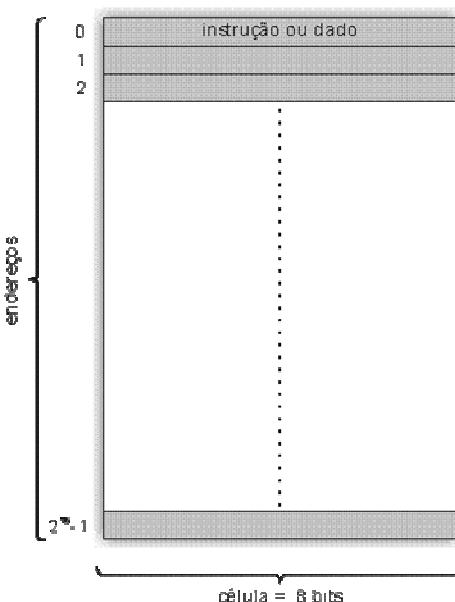
Existe ainda o Registrador de Status, ou PSW (*program status word* – palavra de estado do programa). Esse registrador contém os bits do código de condições, os quais são alterados pelas instruções de comparação, pelo nível de prioridade da CPU, pelo modo de execução (usuário ou núcleo) e por vários outros bits de controle.

A maioria das CPUs - exceto aquelas muito simples dos sistemas embarcados - apresenta dois modos de funcionamento: o modo núcleo e o modo usuário. Tal modo de funcionamento é controlado por um bit do registrador PSW. Funcionando em modo núcleo, a CPU pode executar qualquer instrução do seu conjunto e usar cada atributo de seu hardware. É o caso do sistema operacional. Por outro lado, programas de usuários são executados em modo usuário, o que permite a execução de apenas um subconjunto das instruções e acesso apenas a um subconjunto de atributos.

### 2.3.2 MEMÓRIA PRINCIPAL

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. A grande maioria dos computadores utiliza o byte (8 bits) como tamanho da célula. Observe na figura ao lado, uma memória de 64KB.

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 através de um registrador denominado **registrador de endereço de memória** (*memory address register – MAR*). Através do conteúdo deste registrador, a unidade de controle sabe qual célula será acessada. Outro registrador usado em operações com a memória é o **registrar 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 gravação. Este ciclo de leitura e gravação é apresentado na tabela abaixo:

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

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  $(2^n - 1)$ .

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.

### 2.3.3 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 do que se este dado estivesse na memória principal. O propósito do uso da memória cache é minimizar a disparidade existente entre a velocidade com que o processador executa as instruções e a velocidade com que dados são acessados na memória principal.

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 ele se encontra na 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 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. Ele garante que, após a transferência de um novo bloco da memória principal para a cache a probabilidade de futuras referências é alta, otimizando assim, o tempo de acesso à informação.

Apesar de ser uma memória de acesso rápido, a capacidade de armazenamento das memórias cache é limitada em função do seu alto custo.

### 2.3.4 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 as informações, a memória secundária não precisa de alimentação.

O acesso à memória secundária é lento, se comparado 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, a acesso à memória principal é de nanosegundos. Podemos citar como exemplos de memórias secundárias, a fita magnética, os discos magnéticos, rígidos e óticos.

A figura abaixo mostra a relação entre os diversos tipos de dispositivos de armazenamento apresentados:

Tempo de Acesso	Capacidade Típica
1 ns	< 1 KB
2 ns	1 MB
10 ns	64–512 MB
10 ms	5–50 GB
100 s	20–100 GB

```

graph TD
    Reg[Reg] --- Cache[Cache]
    Cache --- MP[Memória Principal]
    MP --- DR[Disco Rígido (magnético)]
    DR --- FM[Fita Magnética]
  
```

O armazenamento em disco é duas ordens de magnitude mais barato, por bit, que o da RAM e, muitas vezes, duas ordens de magnitude maior também. O único problema é que o tempo de acesso aleatório aos dados é perto de três ordens de magnitude mais lento. Essa baixa velocidade é devida ao fato de o disco ser um dispositivo mecânico.

A última camada é a **fita magnética**. Meio muito utilizado como cópia de segurança para abrigar grandes quantidades de dados. A fita deverá ser percorrida seqüencialmente até chegar ao bloco requisitado. No mínimo, isso levaria alguns minutos. A grande vantagem da fita é que ela tem um custo por bit muito baixo e é também removível.

A hierarquia de memória que vimos é o padrão mais comum, mas alguns sistemas não têm todas essas camadas ou algumas são diferentes delas (como discos ópticos). No entanto, em todas, conforme se desce na hierarquia, o tempo de acesso aleatório cresce muito, a capacidade, da mesma maneira, também aumenta bastante e o custo por bit cai enormemente. Em vista disso, é bem provável que as hierarquias de memória ainda perdurem por vários anos.

### 2.3.5 DISPOSITIVOS DE ENTRADA/SAÍDA

A memória não é o único recurso que o sistema operacional tem de gerenciar. Os dispositivos de E/S também interagem intensivamente com o sistema operacional. Os dispositivos são constituídos, geralmente, de duas partes: o controlador e o dispositivo propriamente dito.

O **controlador** é um chip ou um conjunto deles que controla fisicamente o dispositivo. Ele recebe comandos do sistema operacional, por exemplo, para ler dados do dispositivo e para enviá-los. Cabe ao controlador apresentar uma interface mais simples para o sistema operacional. Por exemplo, um controlador de disco, a o receber um comando para ler o setor 11206 do disco 2, deve então converter esse número linear de setor em

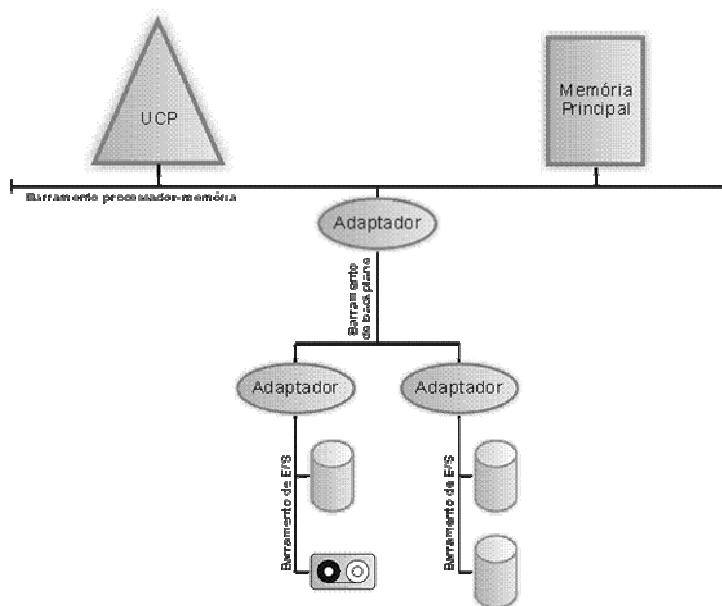
números de cilindro, setor e cabeça. Essa conversão pode ser muito complexa, já que os cilindros mais externos têm mais setores que os internos e que alguns setores danificados podem ter sido remapeados para outros. Então, o controlador precisa determinar sobre qual cilindro o braço do acionador está e emitir uma seqüência de pulsos, correspondente à distância em número de cilindros, para mover-lo em direção ao centro do disco ou à borda.

A outra parte é o dispositivo real. Eles possuem interfaces bastante simples porque não fazem nada muito diferente, e isso ajuda a torná-los padronizados.

Visto que cada controlador é diferente, diferentes programas são necessários para controlá-los. O programa que se comunica com um controlador, emitindo comandos a ele e aceitando respostas, é denominado **driver do dispositivo**. Cada fabricante deve fornecer um driver específico para cada sistema operacional suportado.

Para ser usado, o driver deve ser colocado dentro do sistema operacional para que possa ser executado em modo núcleo. Há três maneiras de colocar o driver dentro do núcleo. A primeira é religar (*linkeditar*) o núcleo com o novo driver e então reiniciar o sistema. Muitos sistemas Unix funcionam assim. A segunda é fazer uma entrada de arquivo e o sistema operacional vê que precisa do driver e então reinicia o sistema. No momento da iniciação, o sistema operacional busca e encontra os drivers de que ele precisa e os carrega. O Windows funciona assim. A terceira maneira é capacitar o sistema operacional e aceitar novos drivers enquanto estiver em execução e instalá-los sem a necessidade de reiniciar. O Windows XP faz assim.

### 2.3.6 BARRAMENTO



O **barramento** ou **bus** é um meio físico de 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 memória. Diferentemente, os **barramentos de E/S** possuem maior extensão, são lentos e permitem a conexão de diferentes dispositivos.

Alguns sistemas de alto desempenho utilizam uma arquitetura com um terceiro barramento, conhecido como **barramento de backplane**. 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 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.

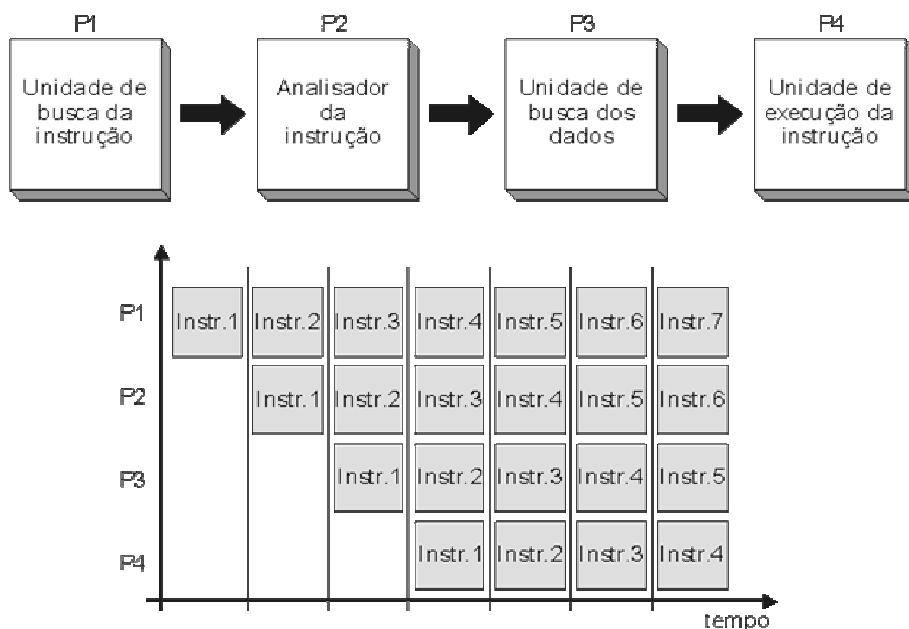
Existem três barramentos mais utilizados: IDE, USB e SCSI. O barramento **IDE** (*Integrated Drive Electronics*) serve para acoplar ao sistema dispositivos periféricos como discos e CD-ROMs. Ele é uma extensão da interface controladora de discos do PC/AT e atualmente constitui um padrão para disco rígido e muitas vezes também para CD-ROM em quase todos os sistemas baseados em Pentium.

O **USB** (*Universal Serial Bus* – barramento serial universal) foi inventado para conectar ao computador todos os dispositivos lento de E/S, como teclado e mouse. É um barramento centralizado no qual um dispositivo-raiz interroga os dispositivos de E/S a cada 1ms para verificar se eles têm algo a ser transmitido. Ele pode tratar uma carga acumulada de 1,5 MB/s. todos os dispositivos USB compartilham um único driver de dispositivo USB, tornando desnecessário instalar um novo driver para cada novo dispositivo. Consequentemente, os dispositivos USB podem ser adicionados ao computador sem precisar reiniciá-lo.

O barramento **SCSI** (*Small Computer System Interface* – interface de pequeno sistema de computadores) é um barramento de alto desempenho destinado a discos rápidos, scanners e outros dispositivos que precisem de considerável largura de banda. Pode funcionar em até 160 Mb/s.

### 2.3.7 PIPELINING

Para melhorar o desempenho, os projetistas de CPU abandonaram o modelo simples de busca, decodificação e execução de uma instrução por vez. Muitas CPUs modernas têm recursos para executar mais de uma instrução ao mesmo tempo. Por exemplo, uma CPU pode ter unidades separadas de busca, decodificação e execução, de modo que, enquanto ela estiver executando a instrução  $n$ , ela pode também estar decodificando a instrução  $n+1$  e buscando a instrução  $n+2$ . Essa organização é chamada **pipeline** e está ilustrada na figura para um *pipeline* de quatro estágios.



É uma técnica que permite ao processador executar múltiplas instruções paralelamente em estágios diferentes. O conceito de processamento *pipeline* se assemelha muito ao de uma linha de montagem, onde uma tarefa é dividida em uma seqüência de sub-tarefas, executadas dentro da linha de produção.

O *pipeline* 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.

### 2.3.8 ARQUITETURAS RISC e CISC

A linguagem de máquina de um computador é a linguagem de programação 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*) se caracteriza por possuir poucas instruções de máquina, em geral bastante simples, 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 (Compac) e Rx000 (MIPS).

Os processadores com arquitetura CISC (*Complex Instruction Set Computer*) 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).

Nos processadores RISC, um programa em linguagem de máquina é executado diretamente pelo hardware. Já na arquitetura CISC, entre os níveis da linguagem de máquina e dos circuitos eletrônicos, existem um nível intermediário que é o da **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.

### 2.3.9 ANÁLISE DE DESEMPENHO

Para avaliar o desempenho de processadores, diversas variáveis devem ser consideradas, entre as quais o intervalo de tempo entre os pulsos de um sinal de clock, conhecido como **ciclo de clock**. A **frequência de clock** é o inverso do ciclo de clock e indica o número de pulsos elétricos gerados em um segundo. A medida da frequência é dada em Hertz (hz) ou seus múltiplos como Kilohertz (Khz) ou Megahertz (Mhz).

O desempenho de um processador pode ser avaliado pela comparação dos tempos que processadores distintos levam para executar um mesmo programa. Este tempo é denominado *tempo de CPU*, que leva em consideração apenas o tempo para executar

instruções pelo processador, não incluindo a espera em operações de E/S. O tempo de CPU para um determinado programa é dado por:

$$\text{TEMPO DE CPU} = \text{nº de ciclos de clock} \times \text{ciclo de clock para execução do programa}$$

Considerando que a freqüência do clock é o inverso do ciclo, podemos ter:

$$\text{TEMPO DE CPU} = \text{nº de ciclos de clock para execução do programa} / \text{freqüência do clock}$$

É possível perceber, observando as fórmulas anteriores, que o desempenho do processador pode ser otimizado reduzindo o número de ciclos de clock utilizados para executar um programa, ou aumentando a freqüência do clock.

A técnica conhecida como *benchmark* permite a análise de desempenho comparativa entre sistemas computacionais. Neste método, um conjunto de programas é executado em cada sistema avaliado e o tempo de execução é comparado. A escolha dos programas deve ser criteriosa para refletir os diferentes tipos de aplicação.

## 2.4 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.

Usaremos **utilitário** para fazer referência a softwares relacionados mais diretamente com serviços complementares do sistema operacional, como compiladores, linkeditores e depuradores. Os softwares desenvolvidos pelos usuários serão denominados **aplicativos**.

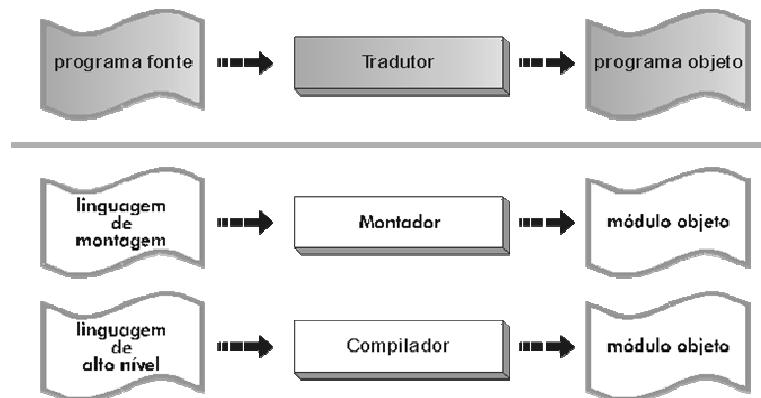
### 2.4.1 TRADUTOR

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, a documentação e a manutenção de programas.

Apesar das inúmeras vantagens proporcionadas pelas linguagens de montagem e de alto nível, os programas não estão prontos para ser diretamente executados pelo processador. 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. Isto 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.



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

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, Cobol e C, não têm nenhuma relação direta com a máquina, ficando essa preocupação exclusivamente com o compilador. Assim, os programas-fonte podem ser portados entre computadores de diversos fabricantes, permitindo o desenvolvimento de aplicações independente do equipamento.

#### **2.4.2 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 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.

#### **2.4.3 LINKEDITOR**

É o utilitário responsável por gerar, a partir de um ou mais módulos-objeto, um único programa executável. Suas funções básicas são resolver todas as referências simbólicas existentes entre os módulos e reservar memória para a execução do programa.

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

#### **2.4.4 INTERPRETADOR DE COMANDOS e LINGUAGEM DE CONTROLE**

O **interpretador de comandos** ou **Shell** permite que o usuário se comunique diretamente com o Sistema Operacional. Através de comandos simples, o usuário pode ter acesso a diversas funções e rotinas específicas do sistema. Quando digitadas pelo usuário, os comandos são interpretados pelo shell, que, ao reconhecer a linha de comando, verifica sua sintaxe, envia mensagens de aviso ou erro e faz chamadas a rotinas do sistema. Dessa forma, o usuário dispõe de uma interface direta com o Sistema Operacional para realizar tarefas como criar, ler ou eliminar arquivos, consultar diretórios ou verificar a data e hora armazenada no sistema.

O conjunto de comandos disponíveis pelo interpretador é conhecido como **linguagem de controle**. Algumas linguagens são poderosas a ponto de oferecer a possibilidade de criar programas com estruturas de decisão e iteração. Esses programas nada mais são que uma seqüência de comandos armazenados em um arquivo texto, denominados arquivos de comandos ou *scripts*, que podem ser executados sempre que necessário.

#### **2.4.5 ATIVAÇÃO / DESATIVAÇÃO DO SISTEMA**

Inicialmente, todo o código do Sistema Operacional reside na memória secundária como discos e fitas. Toda vez que um computador é ligado, o Sistema Operacional tem que ser carregado da memória secundária para a memória principal. Esse procedimento, denominado **ativação do sistema** (*boot*), é realizado por um programa localizado em um bloco específico do disco (área de boot). O procedimento de ativação varia em função do equipamento, podendo ser realizado através do teclado, de um terminal ou no painel do gabinete do processador.

Além da carga do Sistema Operacional, a ativação do sistema também consiste na execução de arquivos de inicialização. Nestes arquivos são especificados procedimentos de customização e configuração de hardware e software específicos para cada ambiente.

Na maioria dos sistemas, também existe o processo de desativação (*shutdown*). Este procedimento permite que as aplicações e componentes do Sistema Operacional sejam desativados de forma ordenada, garantindo sua integridade.

## 2.5 EXERCÍCIOS

- 17) Pratique conversão de unidades:
- Quanto dura um microano em segundos?
  - Micrômetros são chamados de mícrons. Qual o tamanho de um gigamícron?
  - Quantos bytes há em 1 TB de memória?
  - A massa da Terra é de 6000 yottagramas. Qual é esse peso em quilogramas?
- 18) Quais são as unidades funcionais de um sistema computacional?
- 19) Quais os componentes de um processador e quais são suas funções?
- 20) Como a memória principal de um computador é organizada?
- 21) Descreva os ciclos de leitura e gravação da memória principal.
- 22) Qual o número máximo de células endereçadas em arquiteturas com MAR de 16, 32 e 64 bits?
- 23) O que são memórias voláteis e não-voláteis?
- 24) Conceitue memória cache e apresente as principais vantagens do seu uso.
- 25) Quais as diferenças entre memória principal e memória secundária?
- 26) Caracterize os barramentos processador-memória, E/S e backplane.
- 27) Como a técnica de pipelining melhora o desempenho dos sistemas computacionais?
- 28) Compare as arquiteturas de processadores RISC e CISC.
- 29) Conceitue a técnica de benchmark e explique sua realização.
- 30) Por que o código-objeto gerado pelo tradutor ainda não pode ser executado?
- 31) Por que a execução de programas interpretados é mais lenta que a de compilados?
- 32) Qual a função do linkeditor?
- 33) Pesquise comandos disponíveis em linguagens de controle de Sistemas Operacionais.
- 34) Explique o processo de ativação (boot) do Sistema Operacional.
- 35) Uma razão para a demora da adoção das interfaces gráficas era o custo do hardware necessário para suportá-las. De quanta RAM de vídeo se precisa para suportar uma tela de texto monocromático com 25 linhas x 80 colunas de caracteres? Quanto é necessário para suportar um mapa de bits com 1024 x 768 pixels de 24 bits? Qual é o custo dessa RAM em preços de 1980 (5 dólares / Kb)? Quanto custa agora?
- 36) Das instruções a seguir, quais só podem ser executadas em modo núcleo?
- Desabilite todas as interrupções;
  - Leia o horário do relógio;
  - Altere o horário do relógio;
  - Altere o mapa de memória.

- 37) Um computador tem um pipeline de quatro estágios. Cada estágio leva o mesmo tempo para fazer seu trabalho – digamos 1 ns. Quantas instruções por segundo essa máquina pode executar?
- 38) Um revisor alerta sobre um erro de ortografia no original de um livro-texto sobre Sistemas Operacionais que está para ser impresso. O livro tem aproximadamente 700 páginas, cada uma com 50 linhas de 80 caracteres. Quanto tempo será preciso para percorrer eletronicamente o texto no caso de a cópia estar em cada um dos níveis de memória da tabela da página 18? Para métodos de armazenamento interno, considere que o tempo de acesso é dado por byte (caractere); para discos, considere que o tempo é por bloco de 1KB (1024 caracteres); e, para fitas, que o tempo dado é a partir do início dos dados com acesso subsequente na mesma velocidade que o acesso a discos.

-X-

# 3

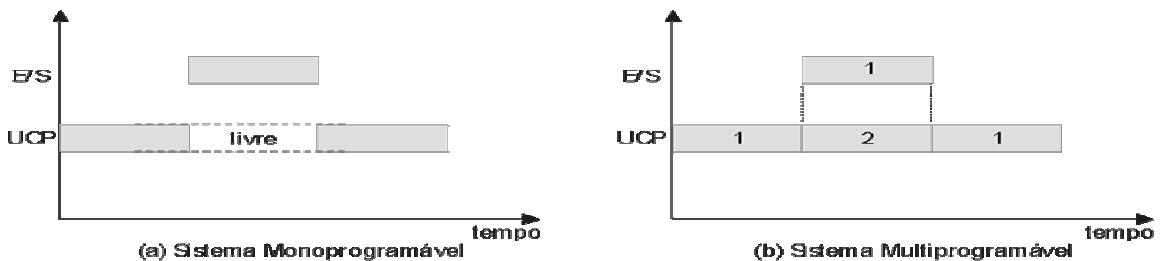
## Concorrência

*“O segredo do sucesso é a perseverança em atingir o objetivo.” (Benjamim Disraeli)*

### 3.1 INTRODUÇÃO

**Concorrência** é a possibilidade de o processador executar instruções em paralelo com operações de E/S permitindo que diversas tarefas sejam executadas concorrentemente. Este é o princípio básico para o projeto e a implementação de sistemas multiprogramáveis.

Os sistemas multiprogramáveis surgiram a partir de limitações existentes nos monoprogramáveis. Nesse, muitos recursos computacionais de alto custo permaneciam muitas vezes ociosos por longo período de tempo, pois enquanto uma leitura de disco é realizada, o processador fica parado. O tempo de espera é longo, já que as operações com dispositivos de entrada e saída são muito lentas se comparadas com a velocidade com que o processador executa as instruções.



A tabela abaixo 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.

Leitura de um registro	0,0015 s
Execução de 100 instruções	0,0001 s
Total	0,0016 s
% utilização da Cpu (0,0001 / 0,0016) = 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 multiprogramáveis, vários programas podem estar residentes em memória, concorrendo pela utilização do processador.

A utilização concorrente da CPU deve ser implementada de maneira que, quando um programa perde o uso do processador e depois retorna para continuar sua execução, seu estado deve ser idêntico ao do momento em que foi interrompido. O programa deverá continuar sua execução exatamente de onde parou, aparentando ao usuário que nada aconteceu.

As vantagens proporcionadas pela multiprogramação podem ser percebidas onde existem um sistema computacional por exemplo, com um disco, um terminal e uma impressora, como mostra a tabela abaixo. Neste ambiente são executados três programas, que possuem características de processamento distintas.

Em um ambiente monoprogramável, os programas são executados seqüencialmente. Sendo assim, o Prog1 é processado em 5 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 exe-

cução dos três programas. No caso de os programas serem executados concorrentemente, em um sistema multiprogramável, o ganho na utilização dos recursos e também no tempo de resposta é considerável.

CARACTERÍSTICAS	PROG1	PROG2	PROG3
Utilização da CPU	alta	baixa	baixa
Operações de E/S	poucas	muitas	muitas
Tempo de processamento	5 min	15 min	10 min
Memória utilizada	50 KB	100 KB	80 KB
Utilização do disco	não	não	sim
Utilização do terminal	não	sim	não
Utilização da impressora	não	não	sim

### 3.2 INTERRUPÇÃO e EXCEÇÃO

Durante a execução de um programa, alguns eventos inesperados podem ocorrer, 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 (interrupção) ou da execução de instruções do próprio programa (exceção).

Uma **interrupção** é sempre gerada por algum evento externo ao programa e, neste 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. Neste 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. Neste 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.



Para cada tipo de interrupção existe uma rotina de tratamento associada, para 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.

Existem dois métodos utilizados para o tratamento de interrupções. O primeiro 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. O outro

método utiliza um registrador de status que armazena o tipo de 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 relacionadas à instrução do programa corrente, portanto são imprevisíveis e podem ocorrer múltiplas vezes de forma simultânea. Uma maneira de evitar esta situação é a rotina de tratamento inibir as demais interrupções.

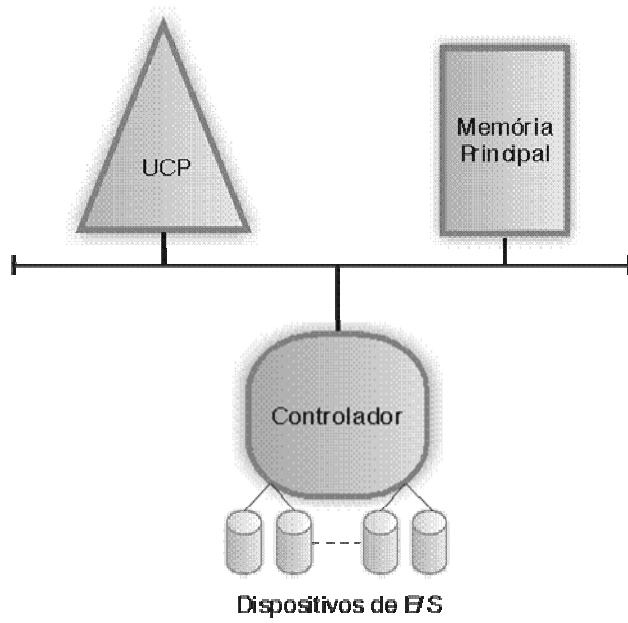
Porém alguns processadores não permitem que interrupções sejam desabilitadas, fazendo com que exista um tratamento para a ocorrência de múltiplas interrupções. Normalmente, existe um dispositivo denominado **controlador de pedidos de interrupção**, responsável por avaliar as interrupções geradas e suas 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 pode ocorrer um único de cada vez.

### 3.3 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. 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. Isso simplificou as instruções de E/S, por não ser mais necessário especificar detalhes de operações dos periféricos, tarefa esta realizada pelo 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 ficava permanentemente testando o estado do periférico para verificar o final da operação. Este controle **E/S controlada por programa**, mantinha o processador ocupado até o término da E/S (espera ocupada ou *busy wait*).

A evolução do modelo anterior foi permitir que, após o início da transferência dos dados, o processador ficasse livre para outras tarefas. Assim, em determinados espaços de tempo, o Sistema Operacional deveria testar cada dispositivo para saber do término da operação de E/S (*Polling*). O problema dessa implementação é que, no caso de existir um grande número de periféricos, o processamento é interrompido freqüentemente para testar os diversos componentes.

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 do 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 registradores próprios. Em seguida, o controlador sinaliza uma interrupção ao processador. Quando o processador atende tal interrupção, a rotina responsável transfere os dados dos registradores do controlador para a memória principal.

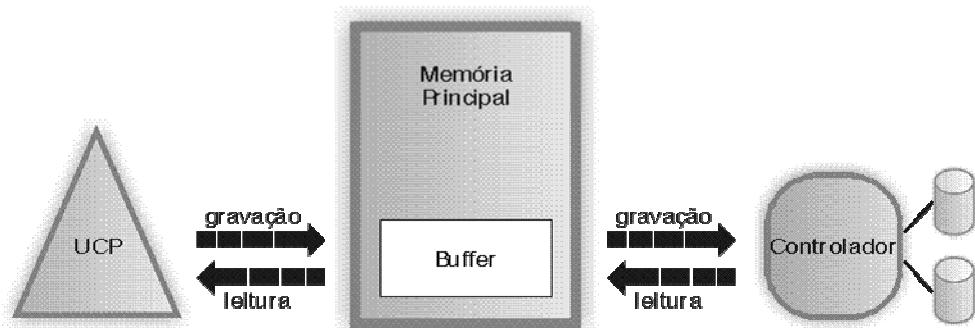
A operação de E/S controlada por interrupção é muito mais eficiente, porém, a transferência de grande volume de dados exige muitas intervenções do processador, reduzindo sua eficiência. Para resolver esse problema, implementou-se uma técnica de transferência de dados denominada DMA (*Direct Memory Access*).

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 a localização, o dispositivo de E/S, a posição inicial da memória onde os dados serão lidos ou gravados e o tamanho do bloco. O controlador realiza então a transferência entre o periférico e a memória principal, e o processador é interrompido somente no final da operação. A área de memória utilizada pelo controlador na técnica DMA é chamada **buffer de E/S**.

### 3.4 BUFERIZAÇÃO

Consiste na utilização de uma área da memória principal, 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. Neste 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.

A buferização permite minimizar o problema da disparidade da velocidade de processamento existente entre o processador e os dispositivos de E/S. O objetivo principal dessa técnica é manter, na maior parte do tempo, processador e dispositivos ocupados.



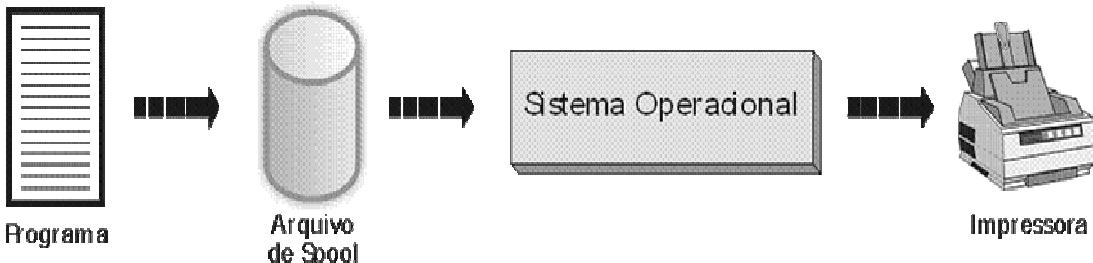
### 3.5 SPOOLING

A técnica de **spooling** (*simultaneous peripheral operation on-line*) foi introduzida no final de 1950 para aumentar o grau de concorrência e eficiência nos Sistemas Operacionais.

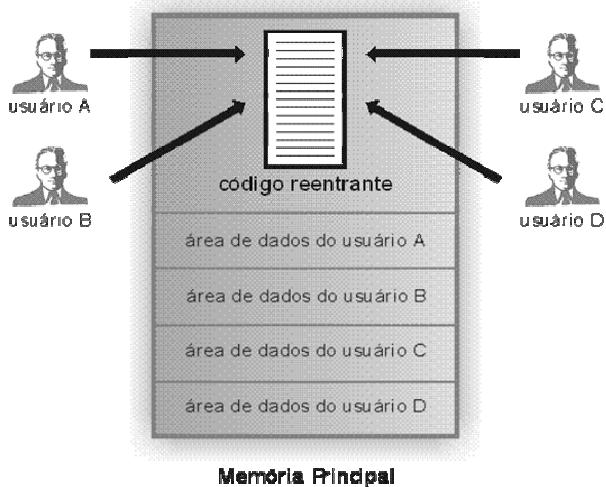
A técnica de spooling, semelhante à técnica de buferização, utiliza uma área em disco como se fosse um grande buffer. Neste caso, dados podem ser lidos ou gravados em disco, enquanto programas são executados concorrentemente. Atualmente essa técnica está presente na maioria dos Sistemas Operacionais sendo utilizada no gerenciamento de impressão.

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

Observe a figura na página seguinte:



### 3.6 REENTRÂNCIA



É comum, em sistemas multiprogramáveis, vários usuários utilizarem os mesmos aplicativos simultaneamente, como editores de texto e compiladores. Se cada usuário que utilizasse um destes 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 (reentrante) ser compartilhado por diversos usuários, exigindo que apenas uma cópia do programa esteja na memória.

### 3.7 EXERCÍCIOS

- 39) O que é concorrência e como este conceito está presente nos Sistemas Operacionais multiprogramáveis?
- 40) Por que o mecanismo de interrupção é fundamental para a implementação da multiprogramação?
- 41) Explique o mecanismo de funcionamento das interrupções.
- 42) O que são eventos síncronos e assíncronos? Como estes eventos estão relacionados ao mecanismo de interrupção e exceção?
- 43) Dê exemplos de eventos associados ao mecanismo de exceção.
- 44) Qual a vantagem da E/S por interrupção comparada com a técnica de polling?
- 45) O que é DMA e qual a vantagem desta técnica?
- 46) Como a técnica de buferização permite aumentar a concorrência em um sistema?
- 47) Explique o mecanismo de spooling de impressão.
- 48) 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?

-X-

# 4

## Estrutura do Sistema Operacional

*“Inteligência... é a faculdade de fazer objetos artificiais, especialmente ferramentas para fazer ferramentas.” (Henry Bergson)*

### 4.1 INTRODUÇÃO

O Sistema Operacional é formado por um conjunto de rotinas que oferecem serviços aos usuários, às suas aplicações, e também ao próprio sistema. Esse conjunto de rotinas é denominado **núcleo do sistema operacional**.



É importante não confundir o núcleo do sistema com aplicações, utilitários ou o interpretador de comandos, que acompanham o Sistema Operacional. As aplicações são utilizadas pelos usuários e escondem todos os detalhes da interação com o sistema. Os utilitários, como compiladores, editores de texto e interpretadores de comandos permitem aos usuários uma interação amigável com o sistema para o desenvolvimento de aplicativos.

A maior dificuldade em compreender a estrutura e o funcionamento de um Sistema Operacional está no fato que ele não é executado como uma aplicação tipicamente seqüencial, com início, meio e fim. Os procedimentos do sistema são executados concorrentemente sem uma ordem, com base em eventos assíncronos.

As principais funções do núcleo encontradas na maioria dos sistemas comerciais são:

- Tratamento de Interrupções
- Tratamento de 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

### 4.2 CHAMADAS DE SISTEMA (System Calls)

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.

As chamadas de sistema podem ser entendidas como uma porta de entrada para o acesso ao núcleo do sistema operacional e a seus serviços. Sempre que um usuário ou aplicação desejar algum serviço do sistema, é realizada uma chamada a uma de suas rotinas através de uma *system call* (chamada ao sistema). O termo *system call* é tipicamente utilizado em sistemas Unix, porém em outros sistemas o mesmo conceito é apresentado com diferentes nomes, como *Application Program Interface* (API) no Windows.

Através dos parâmetros fornecidos na chamada de sistema, a solicitação é processada e uma resposta é retornada à aplicação juntamente com um estado de conclusão indi-

cando se houve algum erro. O mecanismo de ativação e comunicação entre o programa e o sistema operacional é semelhante ao mecanismo implementado quando um programa chama uma sub-rotina.

Para cada serviço disponível existe uma chamada de sistema associada e cada sistema operacional tem seu próprio conjunto de chamadas, com nomes, parâmetros e formas de ativação específicos. Isto explica por que uma aplicação desenvolvida utilizando serviços de um determinado sistema operacional não pode ser portada diretamente para outro sistema.

Uma tentativa de criar uma biblioteca de chamadas padronizadas foi proposta pelos institutos ISO e IEEE. O padrão POSIX (Portable Operating System Interface for Unix), como foi definido, permitiu que uma aplicação desenvolvida seguindo esse conjunto de chamadas pudesse ser executada em qualquer sistema operacional que oferecesse suporte ao padrão.

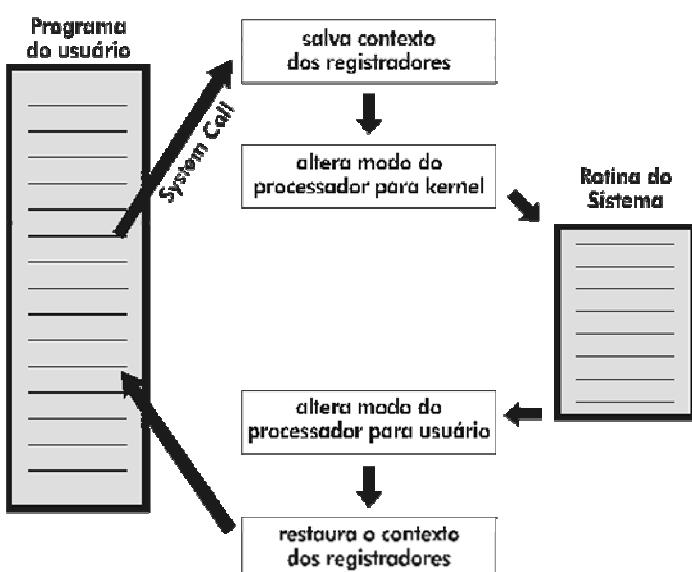
#### 4.3 MODOS DE ACESSO

Existem certas instruções que não podem ser colocadas diretamente à disposição das aplicações, pois a sua utilização indevida ocasionaria sérios problemas à integridade do sistema. Suponha que uma aplicação atualize um arquivo em disco. O programa, por si só, não pode 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 indiscriminadamente, o que poderia comprometer a segurança e integridade do sistema de arquivos.

Portanto, fica claro que existem certas instruções que só devem ser executadas pelo sistema operacional ou sob sua supervisão. As instruções que têm o poder de comprometer o sistema são conhecidas como **instruções privilegiadas**, enquanto as **instruções não-privilegiadas** são as que não oferecem risco ao sistema.

Para que uma aplicação possa executar uma instrução privilegiada, é necessário que no processador seja implementado o mecanismo de proteção conhecido como **modos de acesso**. Existem, basicamente, dois modos de acesso implementados pelos processadores: modo usuário e modo kernel (núcleo) ou supervisor. Quando o processador trabalha no **modo usuário**, uma aplicação só pode executar instruções não-privilegiadas, tendo acesso a um número reduzido de instruções, enquanto no **modo kernel** ou **supervisor** a aplicação pode ter acesso ao conjunto total de instruções do processador.

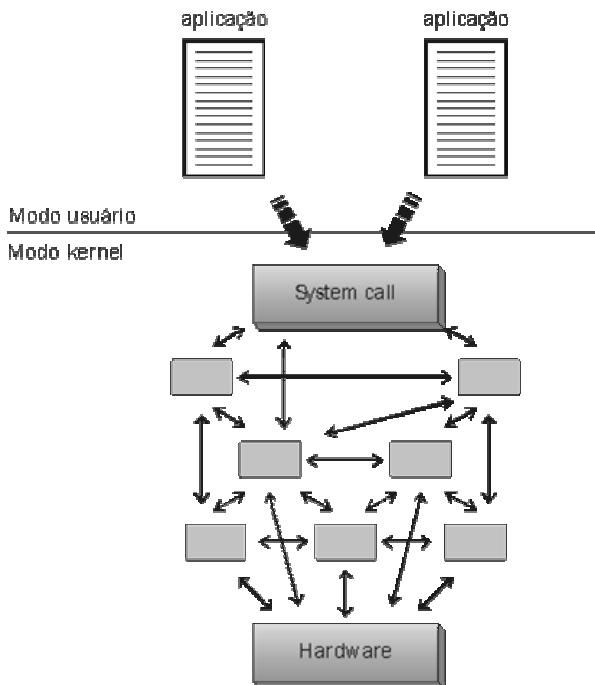
O modo de acesso de uma aplicação é determinado por um conjunto de bits, localizados no registrador de status do processador, ou PSW (Program Status Word), que indica o modo de acesso corrente. Através desse registrador, o hardware verifica se a instrução pode ou não ser executada pela aplicação.



interrompida.

O mecanismo de modos de acesso também é uma boa forma de proteger o próprio núcleo do sistema residente na memória principal. Suponha que uma aplicação tenha acesso a áreas de memória onde está o sistema operacional qualquer programador mal-intencionado ou um erro de programação poderia gravar nesta área, violando o sistema. Com o mecanismo de modos de acesso, para uma aplicação escrever numa área onde resida o Sistema Operacional o programa deve estar sendo executado no modo kernel.

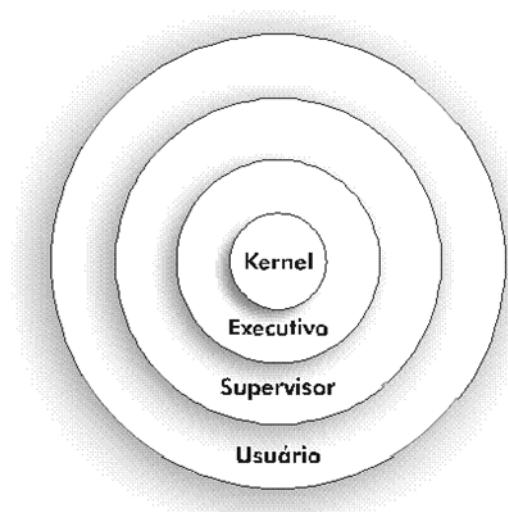
#### 4.4 ARQUITETURA MONOLÍTICA



Pode ser comparada com uma aplicação formada por vários módulos que são compilados separadamente e depois linkados, formando um grande e único programa executável. 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, foi adotada no projeto do MS-DOS e nos primeiros sistemas UNIX.

#### 4.5 ARQUITETURA DE CAMADAS

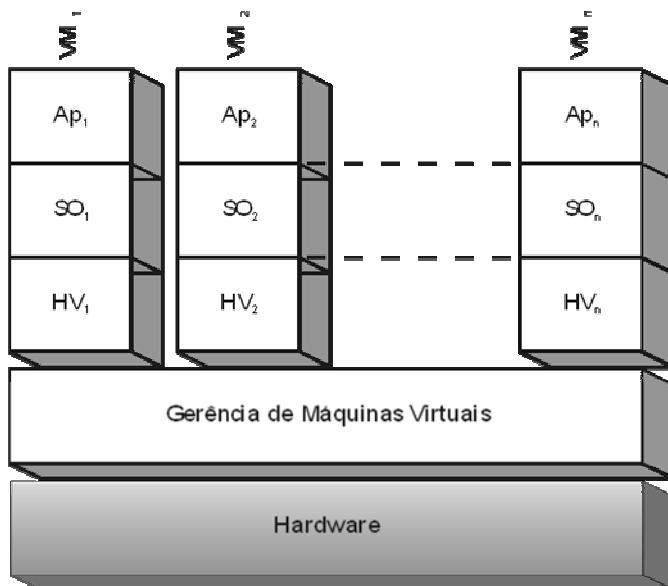
Com o aumento da complexidade e tamanho do código dos sistemas operacionais, técnicas de programação estruturada e modular foram incorporadas ao seu projeto. Na **arquitetura de camadas**, o sistema é dividido em níveis sobrepostos. Cada camada oferece um conjunto de funções que podem ser utilizadas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



A vantagem da estruturação em camadas é isolá-las das 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 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 usuário (não privilegiado) e kernel (privilegiado). A maioria das versões do Unix e dos Windows está baseada nesse modelo.

#### 4.6 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. O modelo de **máquina virtual** cria um nível intermediário entre o hardware e o Sistema Operacional, denominado gerência de máquinas virtuais. 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.

Além de permitir a convivência de Sistemas Operacionais diferentes no mesmo computador, este modelo cria o isolamento total entre cada VM, oferecendo grande segurança para cada uma delas. A desvantagem dessa arquitetura é sua grande complexidade.

Outro exemplo de utilização dessa arquitetura ocorre na linguagem Java. Para se executar um programa em Java é necessário uma máquina virtual Java (JVM). Qualquer sistema operacional pode suportar uma aplicação Java, desde que exista uma JVM desenvolvida para ele. Desta forma, a aplicação não precisa ser recompilada para cada sistema, tornando-se independente do hardware e Sistema Operacional utilizados. A desvantagem é seu menor desempenho se comparado a uma aplicação compilada e executada diretamente em uma arquitetura específica.

#### 4.7 EXERCÍCIOS

- 49) O que é o núcleo do sistema e quais são suas principais funções?
- 50) O que é uma chamada de sistema e qual sua importância para a segurança do sistema? Como as System Calls são utilizadas por um programa?
- 51) O que são instruções privilegiadas e não-privilegiadas? Qual a relação delas com os modos de acesso?
- 52) Explique como funciona a mudança de modos de acesso e dê um exemplo de como um programa faz uso desse mecanismo.
- 53) Como o kernel do Sistema Operacional pode ser protegido pelo mecanismo de modo de acesso?
- 54) Compare as arquiteturas monolítica e de camadas. Quais as vantagens e desvantagens de cada uma?
- 55) Quais as vantagens do modelo de máquina virtual?

-X-

# 5

## Processos

*“Muito do que eu tinha, não consegui libertar;  
Muito do que eu libertei voltou pra mim.” (Lee Wilson Dodd)*

### 5.1 INTRODUÇÃO

O conceito de processo é a base para a implementação de um sistema multiprogramável. O processador é projetado apenas para executar instruções, não sendo capaz de distinguir qual programa se encontra em execução.

A gerência de processos é uma das principais funções de um Sistema Operacional. Através de processos, um programa pode alocar recursos, compartilhar dados, trocar informações e sincronizar sua execução. Nos sistemas multiprogramáveis, os processos são executados concorrentemente, compartilhando, dentro outros recursos, o uso do processador, da memória principal e dos dispositivos de E/S. Nos sistemas com múltiplos processadores, não só existe a concorrência de processos pelo uso do processador, como também a execução simultânea de processos nos diferentes processadores.

### 5.2 ESTRUTURA DO PROCESSO

Nos primeiros sistemas computacionais apenas um programa podia ser executado de cada vez. Tal programa tinha acesso a todos os recursos do sistema. Atualmente, diversos programas executam simultaneamente num sistema computacional. Entretanto, para que isso seja possível é necessário um controle maior na divisão de tarefas entre os vários programas. Daí resultou o conceito de processo.

Um processo é um programa em execução, incluindo os valores correntes de todos os registradores do hardware, e das variáveis manipuladas por ele no curso de sua execução. Embora um processador execute vários processos ao mesmo tempo, dando ao usuário a impressão de paralelismo de execução, num dado instante apenas um processo progride em sua execução. Em outras palavras, em cada momento o processador está executando apenas um processo e seu uso é compartilhado entre os vários processos existentes. Um processo que num instante está usando o processador depois de um certo tempo é interrompido e o processador é cedido a outro processo. Assim, num grande intervalo de tempo todos os processos terão progredido. O rápido chaveamento do processador entre vários processos é chamado multiprogramação. Para que haja esse chaveamento é necessário o uso de algum algoritmo de escalonamento para determinar quando o trabalho de um processo deve ser interrompido e qual o próximo processo a ser servido.

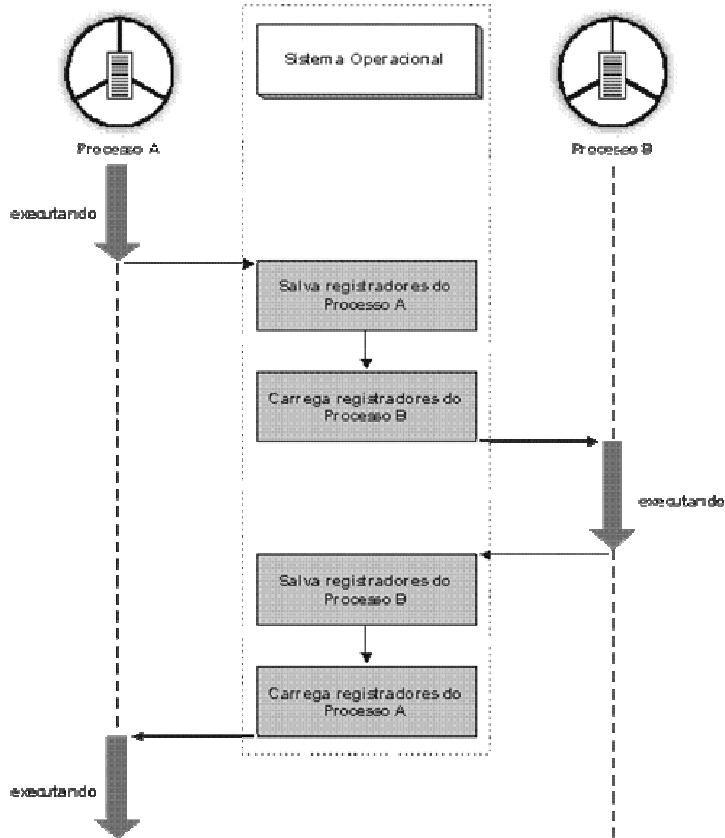
A diferença entre um processo e um programa é sutil, mas crucial. Imagine um cientista de programação que tem excelentes dotes culinários e esteja preparando um bolo de aniversário para a filha. Ele tem à sua disposição uma receita de bolo, e uma dispensa com os ingredientes necessários. A receita é o programa, ou seja, um algoritmo em alguma linguagem conhecida, o cientista de programação é o processador e os ingredientes os dados de entrada. Já o processo vem a ser a atividade resultante da preparação do bolo: leitura da receita, busca dos ingredientes, mistura dos ingredientes, etc. Imagine agora que a filha do nosso cientista apareça chorando, por haver sido picada por uma abelha, enquanto o pai está fazendo o bolo. Neste caso, o cientista deve guardar em que ponto da receita ele estava (o estado do processo corrente é salvo), apanhar um livro de primeiros socorros e seguir as instruções contidas no capítulo do livro sobre picada de abelhas. Aqui vemos o processador, mais precisamente o cientista da computação, sendo chaveado de um processo (preparação do bolo), para outro de maior prioridade (administração de cuidados médicos), cada um deles constituído de um programa diferente (receita versus livro de primeiros socorros). Quando o ferrão da abelha tiver sido retirado da filha do cientista, este poderá voltar à preparação do bolo, continuando seu trabalho a partir do ponto onde ele foi interrompido.

A idéia principal é que um processo constitui uma atividade. Ele possui programa, entrada, saída e um estado. Um único processador pode ser compartilhado entre os vários processos, com algum algoritmo de escalonamento usado para determinar quando parar o trabalho sobre um processo e servir outro.

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

Um processo é formado por três partes, que juntas mantêm todas as informações necessárias à execução de um programa: Contexto de Hardware, Contexto de Software e Espaço de Endereçamento.

### 5.2.1 CONTEXTO DE HARDWARE



Armaزena o conteúdo dos registradores gerais da CPU, além dos registradores de uso específico, como o Contador de Programas, o Ponteiro de Pilha e o Registrador de Status.

A troca de um processo por outro no processador, comandada pelo Sistema Operacional, é denominada troca de contexto. A troca de contexto consiste em salvar o conteúdo dos registradores do processo que está deixando a CPU e carregá-los com os valores referentes ao do novo processo que será executado. Essa operação resume-se em substituir o contexto de hardware de um processo pelo de outro.

### 5.2.2 CONTEXTO DE SOFTWARE

No contexto de software são especificadas características e limites dos recursos que podem ser alocados pelo processo, como o número máximo de arquivos abertos simultaneamente, prioridade de execução e o tamanho do buffer para operação 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 são provenientes de um arquivo do Sistema Operacional, conhecido como arquivo de contas. Neste arquivo, gerenciado pelo administrador do sistema, são especificados os limites dos recursos que cada processo pode alocar.

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

- **Identificação:** cada processo criado pelo Sistema Operacional recebe uma identificação única (PID – Process Identification) representada por um número. Através da PID o Sistema Operacional e outros processos podem fazer referência a qualquer processo existente.

- **Quotas:** são os limites de cada recurso do sistema que um processo pode alocar. Caso uma quota seja insuficiente, o processo será executado lentamente, interrompido durante seu processamento ou mesmo não será executado. Exemplos:

- 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 e threads que podem ser criados.

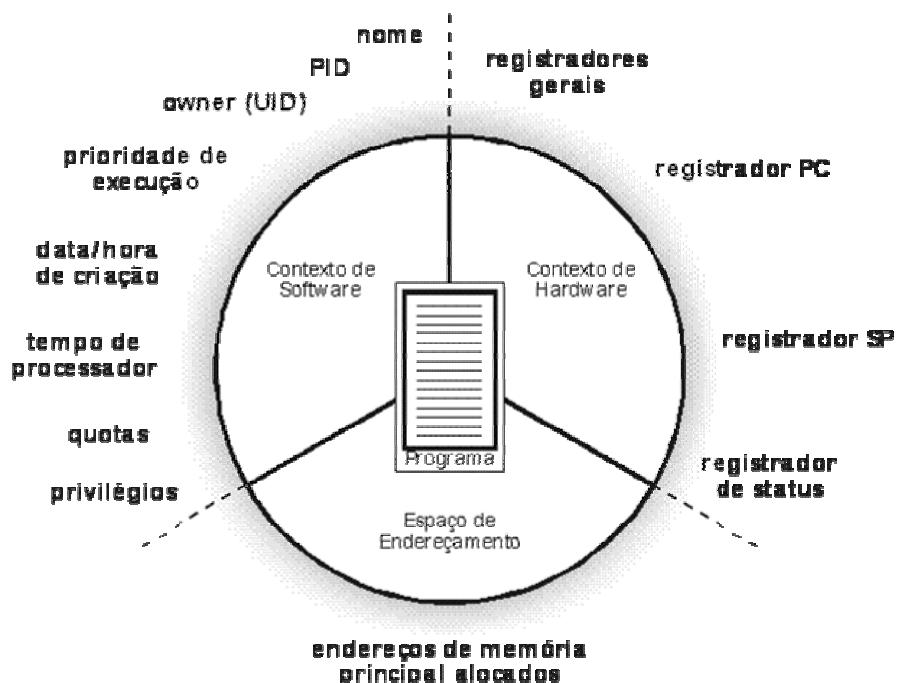
- **Privilégios:** os privilégios ou direitos definem as ações que um processo pode fazer em relação a ele mesmo, aos demais processos e ao Sistema Operacional.

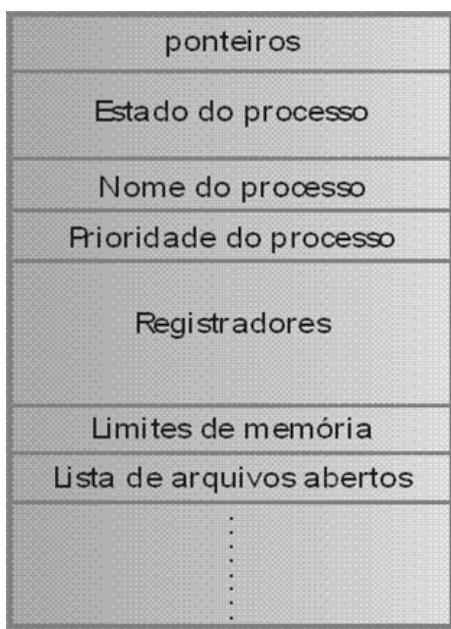
Privilégios que afetam o próprio processo permitem que suas características possam ser alteradas, como prioridade de execução, limites alocados na memória principal e secundária etc. Já os privilégios que afetam os demais processos permitem, além da alteração de suas próprias características, alterar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

Privilégios que afetam o sistema são os mais amplos e poderosos, pois estão relacionados à operação e 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, dentro outros. A maioria dos Sistemas Operacionais possui 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 Windows a conta “administrador” e no OpenVMS existe a conta “system” com este perfil.

### 5.2.3 ESPAÇO DE ENDEREÇAMENTO

É a área de memória pertencente ao processo onde as instruções e os 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.





#### 5.2.4 BLOCO DE CONTROLE DE PROCESSO

O processo é implementado pelo Sistema Operacional através de uma estrutura de dados chamada **Bloco de Controle de 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.

### 5.3 ESTADOS DO PROCESSO

Em um sistema multiprogramável, um processo não deve alocar a CPU com exclusividade, de forma que possa existir 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 estados diferentes:

- **Rodando** (Execução ou *running*): nesse estado, o processo está sendo processado pela CPU. Somente um processo pode estar sendo executado em um dado instante. Os processos se alternam na utilização do processador.
- **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. Em geral existem vários processos no sistema no estado de pronto organizados em listas encadeadas.
- **Bloqueado** (Espera ou *wait*): nesse estado, um processo aguarda por algum evento externo ou por algum recurso para prosseguir seu processamento. Como exemplo, podemos citar o término de uma operação de E/S ou a espera de uma determinada entrada de dados para continuar sua execução. O sistema organiza os vários processos no estado de espera também em listas encadeadas, separados por tipo de evento. Neste caso, quando um evento acontece, todos os processos da lista associada ao evento são transferidos para o estado de pronto.

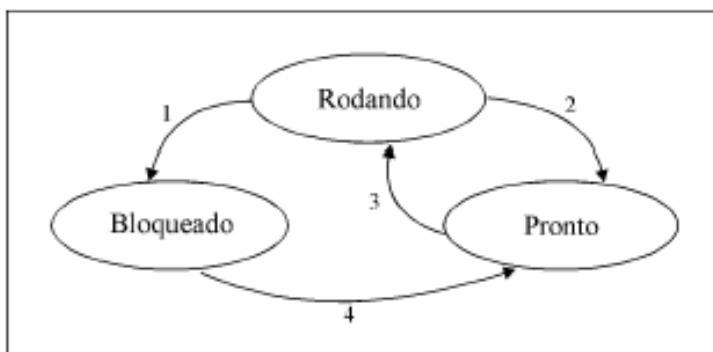
### 5.4 MUDANÇA DE ESTADO DO PROCESSO

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:

- 1) **Rodando – Bloqueado**: essa transição ocorre por eventos gerados pelo próprio processo, como uma operação de E/S.
- 2) **Rodando – 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. Nesse caso, o processo volta para a fila de

pronto, onde aguarda por uma nova oportunidade para continuar seu processamento.

- 3) **Pronto – Rodando:** o Sistema Operacional seleciona um processo da fila de prontos para executar.
- 4) **Bloqueado – Pronto:** um processo passa de bloqueado para pronto quando a operação solicitada é atendida ou o recurso esperado é concedido.



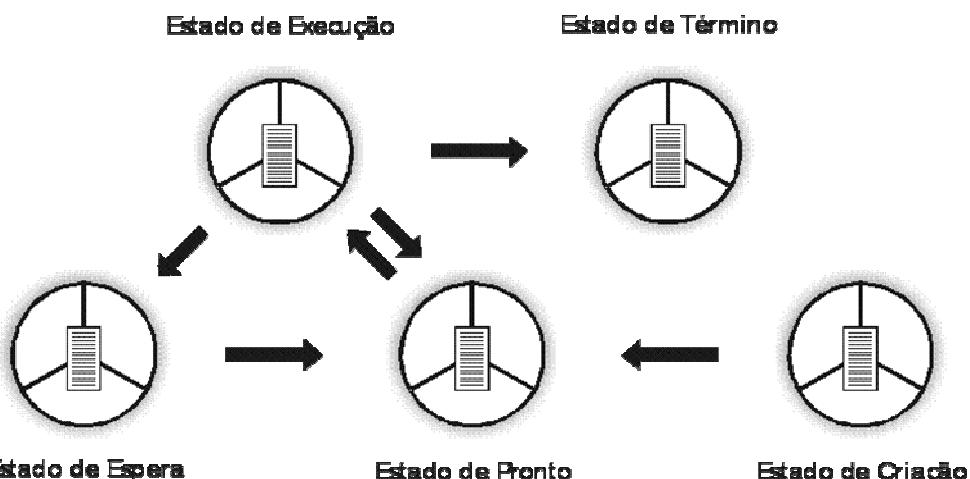
Logicamente, os dois primeiros estados são similares. Em ambos os casos o processo vai executar, só que no segundo não há, temporariamente, CPU disponível para ele. O terceiro estado é diferente dos dois primeiros, pois o processo não pode executar, mesmo que a CPU não tenha nada para fazer.

Quatro transições são possíveis entre esses três estados. A transição 1 ocorre quando um processo descobre que ele não pode prosseguir.

As transições 2 e 3 são causadas pelo escalonador de processos sem que o processo saiba disso. A transição 2 ocorre quando o escalonador decide que o processo em execução deve dar lugar a outro processo para utilizar a CPU. A transição 3 ocorre quando é hora do processo ocupar novamente a CPU. O escalonamento – isto é, a decisão sobre quando e por quanto tempo cada processo deve executar – é um tópico muito importante. Muitos algoritmos vêm sendo desenvolvidos na tentativa de equilibrar essa competição que exige eficiência para o sistema como um todo e igualdade para os processos individuais.

A transição 4 ocorre quando acontece um evento externo pelo qual um processo estava aguardando (como a chegada de alguma entrada).

## 5.5 CRIAÇÃO E ELIMINAÇÃO DE PROCESSOS



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. No caso da eliminação de um processo, todos os recursos associados ao processo são desalocados e o PCB eliminado pelo Sistema Operacional.

A criação de processos pode ocorrer por diferentes razões: 1) *logon* interativo: desta forma, um processo é criado através do estabelecimento de uma sessão interativa por um usuário a partir de um terminal; 2) criação por um outro processo: um processo já existente pode criar outros processos, sendo estes novos processos independentes ou subprocessos; 3) criação pelo sistema operacional: o Sistema Operacional pode criar novos processos com o intuito de oferecer algum tipo de serviço.

Depois de criado, um processo começa a executar e faz seu trabalho. Contudo, nada é para sempre, nem mesmo os processos. Mais cedo ou mais tarde o novo processo terminará, normalmente em razão de alguma das seguintes condições: 1) Saída normal (voluntária); 2) Saída por erro (voluntária); 3) Erro fatal (involuntário) e 4) Cancelamento por um outro processo (involuntário).

Na maioria das vezes, os processos terminam porque fizeram seu trabalho. Quando acaba de compilar o programa atribuído a ele, o compilador avisa ao sistema operacional que ele terminou. Programas baseados em tela suportam também o término voluntário. Processadores de texto, visualizadores da Web (*browsers*) e programas similares tem um ícone que o usuário pode clicar para dizer ao processo que remova quaisquer arquivos temporários que ele tenha aberto e então termine.

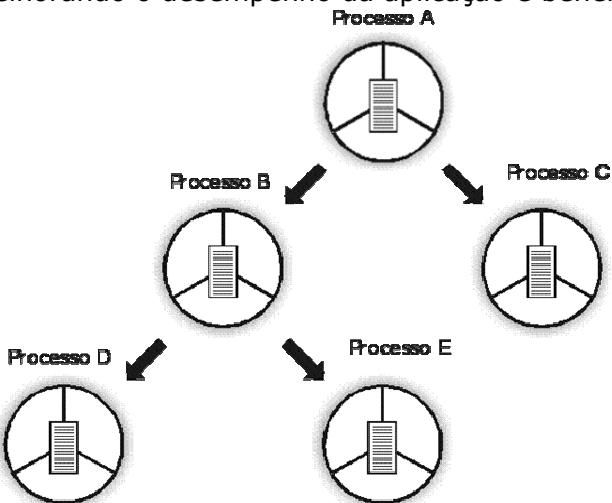
O segundo motivo para término é que o processo descobre um erro. Por exemplo, se um usuário digita um comando para compilar um arquivo que não existe, o compilador simplesmente emite uma chamada de saída ao sistema. Processos interativos com base na tela geralmente não fecham quando parâmetros errados são fornecidos. Em vez disso, uma caixa de diálogo aparece e pergunta ao usuário se ele quer tentar novamente.

A terceira razão para o término é um erro causado pelo processo, muitas vezes por um erro de programa. Entre os vários exemplos estão a execução de uma instrução ilegal, a referência à memória inexistente ou a divisão por zero.

A última razão se dá quando um processo executa uma chamada ao sistema dizendo ao sistema operacional para cancelar algum outro processo. Em alguns sistemas, quando um processo termina, voluntariamente ou não, todos os processos criados por ele são imediatamente cancelados também. Contudo, nem o Unix nem o Windows funcionam dessa maneira.

## 5.6 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 freqüentes 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. Neste caso não existe vínculo do processo criado com seu criador. A criação de um processo independente exige a alocação de um PCB próprio.

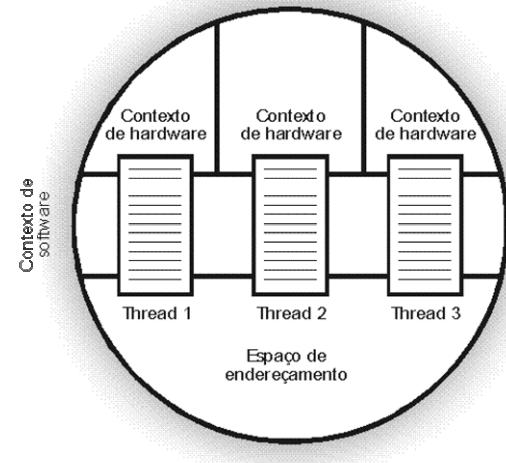
**Subprocessos** são processos criados dentro de uma estrutura hierárquica. Neste modo, o processo criador é denominado **processo pai** enquanto o novo processo é chamado de **subprocesso** ou **processo filho**. O subprocesso, por sua vez, pode criar outras estruturas de subprocessos. Uma característica 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.

Rodrigo Alves  
Sistemas Operacionais – Lucilia Ribeiro

Uma outra característica neste tipo de implementação é que subprocessos podem compartilhar quotas com o processo pai. 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, de software e espaço de endereçamento), consumindo tempo de CPU. No momento do término dos processos, o Sistema Operacional também dispensa tempo para desalocar recursos previamente alocados. Outro problema é a comunicação e sincronização entre processos, considerada pouco eficiente, visto que cada processo possui seu próprio espaço de endereçamento.

O conceito de thread foi introduzido na tentativa de reduzir o tempo gasto na 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. 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.

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 melhor detalhado no próximo capítulo.



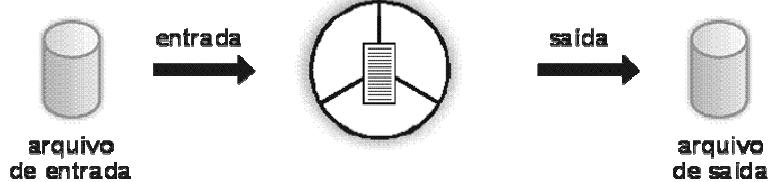
## 5.7 PROCESSOS PRIMEIRO (foreground) E SEGUNDO PLANO (background)

Um **processo de primeiro plano** (interativo) é aquele que permite a comunicação direta do usuário com o processo durante o seu processamento. Um **processo de segundo plano** (*batch*) é aquele onde não existe a comunicação com o usuário durante o seu processamento. 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.

(a) Processo Foreground



(b) Processo Background



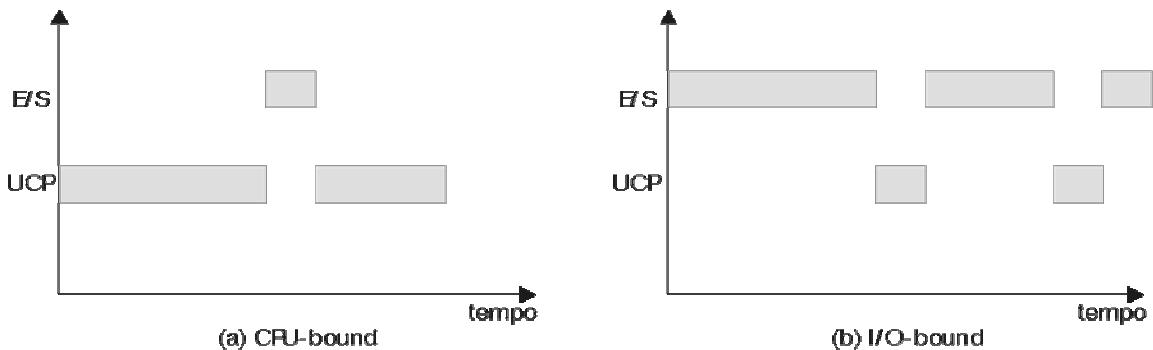
É possível associar o canal de saída de um processo ao canal de entrada de um outro. Neste caso dizemos que existe um *pipe* ligando os dois processos. Se um processo A gera uma listagem e o processo B tem como função ordená-la, basta associar o canal de saída do processo A ao canal de entrada do processo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



## 5.8 PROCESSOS LIMITADOS POR CPU (CPU-BOUND) E POR E/S (I/O-BOUND)

Esta classificação se dá de acordo com a utilização do processador e dos dispositivos de E/S. Um processo é definido como **limitado por CPU** quando passa a maior parte do tempo no estado de execução, ou seja, utilizando o processador. Como exemplo temos os processos utilizados em aplicações científicas que efetuam muitos cálculos.

Um processo é **limitado por E/S** quando passa a maior parte do tempo no estado bloqueado, pois realiza um elevado número de operações de E/S. Esse tipo de processo é encontrado em aplicações comerciais, que se baseiam em leitura, processamento e gravação. Os processos interativos também são exemplos de processos limitados por E/S.



## 5.9 SELEÇÃO DE PROCESSOS

Uma alocação eficiente da CPU é importante para a eficácia global de um sistema. Para tal, diversos algoritmos foram desenvolvidos na tentativa de se ter uma alocação justa do processador. A multiprogramação tem como objetivo maior maximizar a utilização da CPU, ou seja, evitar sempre que possível que o processador fique ocioso.

### 5.9.1 FILAS PARA SELEÇÃO DE PROCESSOS

Num sistema com multiprogramação, os diversos processos existentes no estado de pronto são colocados numa fila conhecida como fila de processos prontos. Um novo processo criado é inicialmente colocado na fila de processos prontos. Ele (assim como os outros processos no estado de pronto) espera nessa fila até ser selecionado para execução. Essa fila pode ser armazenada como uma lista ligada.

É bom salientar que o uso de dispositivos como, por exemplo, uma unidade de fita, pode implicar a necessidade de espera por parte do processo, já que o dispositivo pode estar sendo usado por outro processo. Assim, cada dispositivo tem sua própria fila, chamada fila de dispositivo, com descrições dos processos que estão esperando para usar o dispositivo.

### 5.9.2 ESCALONADORES

Como foi dito anteriormente, um processo passa por várias filas de seleção durante sua execução. Para realizar a alocação de recursos, o sistema operacional deve selecionar os processos dessas filas de alguma maneira. Um escalonador é um programa responsável pelo trabalho de escolher processos e de escalar processos para execução. O

escalonador de processos escolhe processos guardados em disco e os carrega na memória para execução. Pode-se dizer que ele seleciona processos que passaram a poder competir pela CPU. Já o escalonador da CPU escolhe dentre os processos prontos aquele que será executado, alocando a CPU a ele.

A execução do escalonador da CPU deve ser muito rápida. Se levar 10 ms para escolher um processo que será executado por 100 ms, então  $10/(100 + 10) = 9\%$  da CPU será desperdiçada unicamente para a alocação de recursos para realizar o trabalho.

Outra diferença marcante entre os dois tipos de escalonadores é que o escalonador de processos é executado com freqüência bem inferior, já que entre a criação de dois processos podem transcorrer vários minutos. O escalonador de processos controla o número de processos na memória (grau de multiprogramação).

Um escalonador de processos deve manter na memória uma mistura balanceada entre processos que realizam muitas E/S (processos interativos) e processos que usam bastante a CPU. Se todos os processos dependerem de E/S, a fila de prontos estará sempre vazia e o escalonador de CPU terá pouco o que fazer. Caso tenha processos que dependam mais de processamento, a fila de processos em espera pela realização de operações de E/S estará sempre vazia, os dispositivos de E/S ficarão inativos e o sistema terá um desempenho ruim.

## 5.10 EXERCÍCIOS

- 56) O que é um processo? Diferencie processo de programa.
- 57) Por que o conceito de processo é tão importante no projeto de sistemas multiprogramáveis?
- 58) É possível que um programa execute no contexto de um processo e não execute no contexto de um outro? Por quê?
- 59) Quais partes compõem um processo?
- 60) O que é contexto de hardware de um processo e como é a implementação da troca de contexto?
- 61) Qual a função do contexto de software? Exemplifique cada grupo de informação.
- 62) O que é o espaço de endereçamento de um processo?
- 63) Como o Sistema Operacional implementa o conceito de processo? Qual a estrutura de dados indicada para organizar os diversos processos na memória principal?
- 64) Cada processo é descrito por um bloco de controle de processos. Quais são as informações contidas no BCP?
- 65) Defina os três estados possíveis de um processo.
- 66) Faça o diagrama de estados de um processo demonstrando e explicando as transições de um estado para outro.
- 67) Dê um exemplo que apresente todas as mudanças de estado de um processo, juntamente com o evento associado a cada mudança.
- 68) Na teoria, com três estados poderia haver seis transições, duas para cada estado. Contudo, somente quatro transições são mostradas. Por que?
- 69) Diferencie processos multithreads, subprocessos e processos independentes.

- 70) Explique a diferença entre processos de primeiro plano e de segundo plano.
- 71) Dê exemplos de aplicação limitada por CPU e limitada por E/S.
- 72) Trace um paralelo entre o Escalonador de Processos e o Escalonador de CPU.

-X-

# 6

## Threads

*“O que sabemos é uma gota, o que ignoramos é um oceano.”* (Isaac Newton)

### 6.1 INTRODUÇÃO

O conceito de thread surgiu em meados de 1980, com o desenvolvimento do Sistema Operacional Mach na Universidade de Carnegie Mellon.

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 em paralelo.

Atualmente, o conceito de multithread pode ser encontrado em diversos Sistemas Operacionais, como no Sun Solaris e Windows 2000 e posteriores. 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 THREADS

*Threads* são fluxos de execução que rodam dentro de um processo. O que os *threads* acrescentam ao modelo de processo é permitir que múltiplas execuções ocorram no mesmo ambiente do processo com um grande grau de independência uma da outra. Ter múltiplos *threads* executando em paralelo em um processo é análogo a múltiplos processos executando em paralelo em um único computador. No primeiro caso, os *threads* compartilham um mesmo espaço de endereçamento, arquivos abertos e outros recursos. No último, processos compartilham um espaço físico de memória, discos, impressoras e recursos semelhantes.

Exemplo de uso de *thread*: A maioria dos processadores de texto mostra o documento em criação na tela formatado exatamente como ele aparecerá em uma página impressa. Suponha que o usuário esteja escrevendo um livro em um arquivo único. Agora, imagine o que acontece quando o usuário remove, de repente, uma sentença da página 1 de um documento de 800 páginas. Depois de verificar a página alterada para se assegurar se está correta ou não, o usuário quer fazer outra mudança na página 600 e digita um comando dizendo para o processador de textos ir até aquela página (possivelmente buscando por uma frase que apareça somente lá). O processador de textos é então forçado a reformatar todo o conteúdo até a página 600 – uma situação difícil, porque ele não sabe qual será a primeira linha da página 600 enquanto não tiver processado todas as páginas anteriores. Haverá uma demora substancial antes que a página 600 possa ser mostrada, deixando o usuário descontente.

*Threads* nesse caso, podem ajudar. Suponha que o processador de textos seja escrito como um programa de dois *threads*. Um interage com o usuário e o outro faz a reformatação em segundo plano. Logo que uma sentença é removida da página 1, o *thread* interativo diz ao *thread* de reformatação para reformatar todo o livro. Enquanto isso, o *thread* interativo continua atendendo o teclado, o mouse e os comandos simples, como rolar a página 1, enquanto o outro *thread* está processando a todo vapor em segundo plano.

Enquanto estamos nesse exemplo, por que não adicionar um terceiro *thread*? Muitos processadores de texto são capacitados para salvar automaticamente todo o arquivo a cada intervalo de tempo em minutos a fim de proteger o usuário contra a perda do trabalho de um dia, caso ocorra uma falha no programa ou no sistema ou mesmo uma queda de energia. O terceiro *thread* pode fazer os *backups* em disco sem interferir nos outros dois.

Se o programa tivesse apenas um *thread* e se um *backup* de disco iniciasse, os comandos do teclado e do mouse seriam ignorados enquanto o *backup* não terminasse. O usuário perceberia isso como uma queda de desempenho. Por outro lado, os eventos do

teclado e do mouse poderiam interromper o *backup* em disco, permitindo um bom desempenho, mas levando a um complexo modelo de programação orientado à interrupção. Com três *threads*, o modelo de programação fica muito mais simples. O primeiro *thread* apenas interage com o usuário. O segundo reformata o documento quando pedido. O terceiro escreve periodicamente o conteúdo da RAM no disco.

Deve estar claro que três processos separados não funcionariam no exemplo dado, pois todos os três *threads* precisam operar sobre o documento. Em vez de três processos, são três *threads* que compartilham uma memória comum e desse modo têm todo o acesso ao documento que está sendo editado.

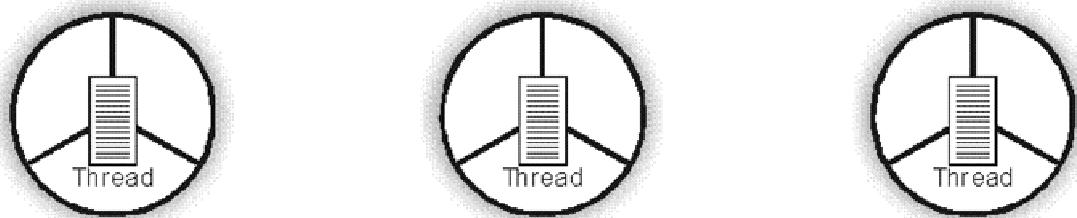
### 6.3 AMBIENTE MONOTHREAD

Um programa é uma seqüência 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.

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-mail. Neste ambiente, um usuário pode estar lendo suas mensagens antigas, ao mesmo tempo em que pode estar enviando e-mails e recebendo novas mensagens. Como uso de múltiplos processos, cada funcionalidade do software implicaria a criação de um novo processo para atendê-la, aumentando o desempenho da aplicação.

Existem dois problemas neste tipo de implementação: 1) o alto consumo de diversos recursos do sistema. Sempre que um novo processo é criado, o sistema deve alocar recursos para cada processo, consumindo tempo de processador neste trabalho e 2) compartilhamento do espaço de endereçamento. Como cada processo possui seu próprio espaço de endereçamento, a comunicação entre processos torna-se lenta e difícil. Além disto, o compartilhamento de recursos comuns aos processos concorrentes, como memória e arquivos abertos, não é simples.

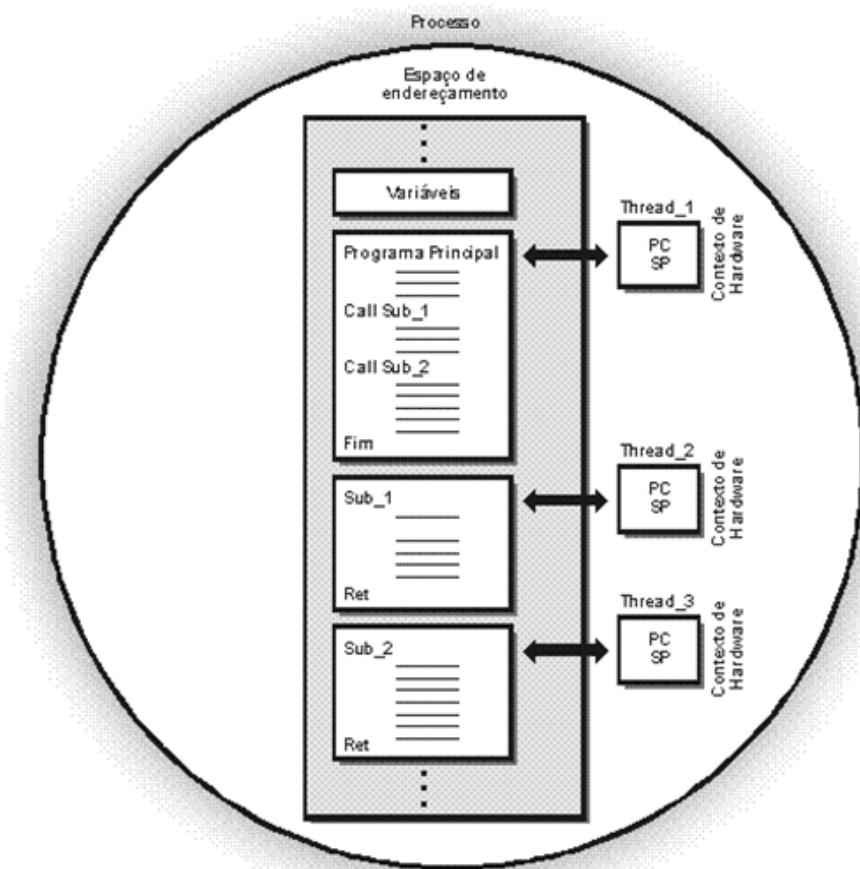
São exemplos o MS-DOS e as primeiras versões do Windows.



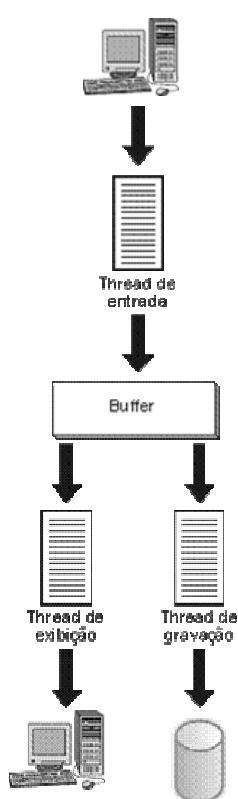
### 6.4 AMBIENTE MULTITHREAD

Neste caso, não existe a idéia de programas associados a processos, mas 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.

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 paralelamente 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 figura, existe um programa principal que realiza a chamada de duas sub-rotinas (Sub1 e Sub2). Inicialmente, o processo é criado apenas com o Thread1 para a execução do programa principal. Quando o programa principal chama as sub-rotinas Sub1 e Sub2, são criados os Thread2 e Thread3, 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.

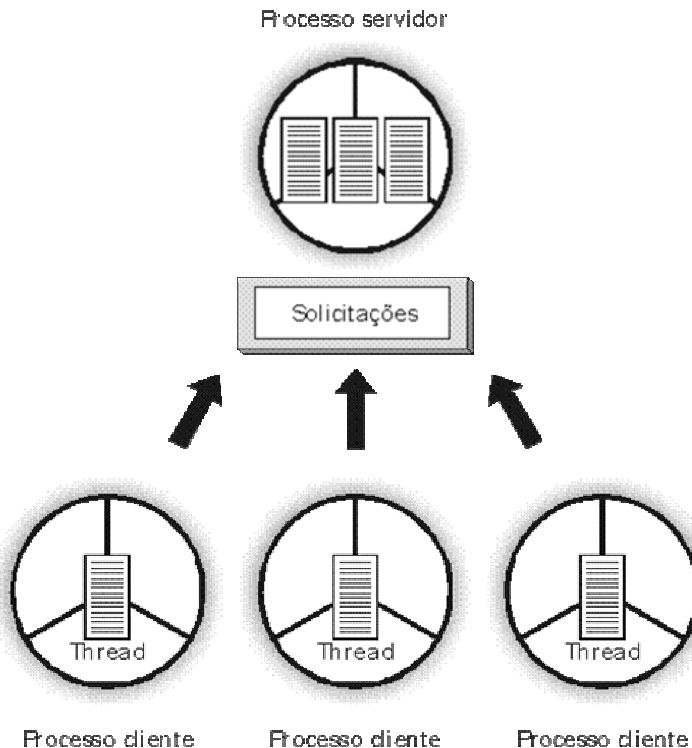
Dentro de um mesmo processo, threads compartilham o mesmo contexto de software e espaço de endereçamento com os demais threads, porém cada um possui seu contexto de hardware individual. Threads são implementados internamente através de uma estrutura de dados denominada **Bloco de controle do thread**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

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.

A utilização do processador, dos discos e de outros periféricos pode ser feita de forma concorrente pelos diversos threads, significando melhor utilização dos recursos computacionais disponíveis. Em algu-

mas aplicações, a utilização de threads pode melhorar o desempenho da aplicação apenas executando tarefas em *background* enquanto operações de E/S estão sendo processadas, conforme mostra a figura. Aplicações como editores de texto, planilhas, aplicativos gráficos e processadores de imagens são especialmente beneficiados quando desenvolvidos 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.



## 6.5 ARQUITETURA E IMPLEMENTAÇÃO

O conjunto de rotinas disponíveis para que uma aplicação utilize as facilidades dos threads é chamado de **pacote de threads**.

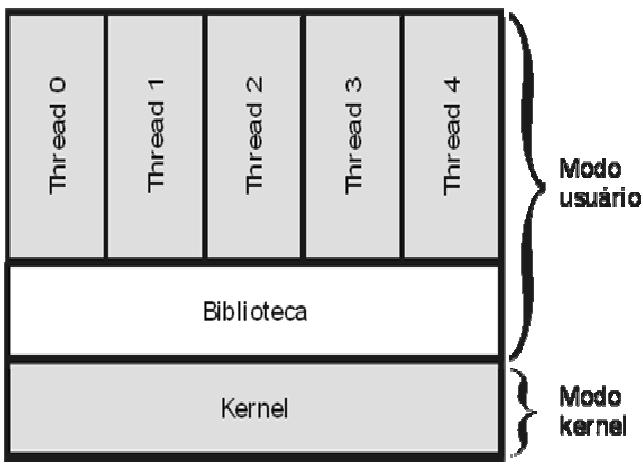
Threads podem ser oferecidos por uma biblioteca de rotinas fora do núcleo do Sistema Operacional (modo usuário), pelo próprio núcleo do sistema (modo kernel) ou por uma combinação de ambos (modo híbrido).

### 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. TMU são rápidos e eficientes por dispensarem acesso ao kernel do Sistema Operacional, evitando assim a mudança de modo de acesso (usuário-kernel-usuário).

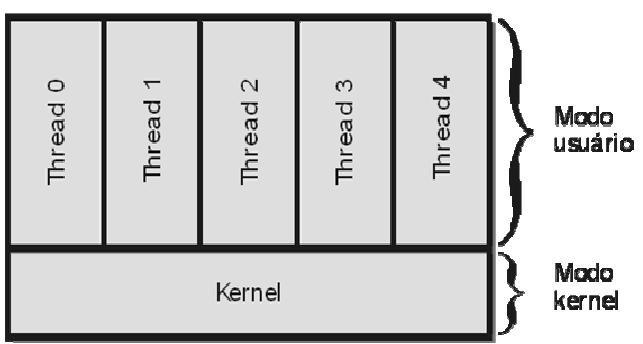
TMU possuem uma grande limitação, pois o Sistema Operacional gerencia cada processo como se existisse apenas um único thread. No momento em que um thread chama uma rotina do sistema que o coloca em estado bloqueado (rotina bloqueante), todo o processo é colocado no estado de espera, mesmo havendo outros 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 o 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 multiprocessados, não é possível que múltiplos threads de um processo possam ser executados em diferentes CPUs 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 só podem ser executados em um processador de cada vez.

### 6.5.2 THREADS EM MODO KERNEL

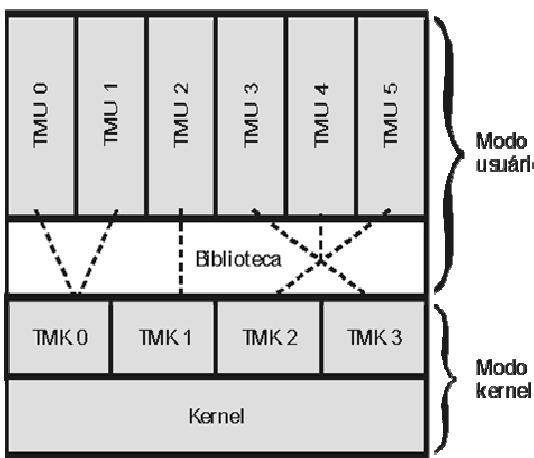


Threads em modo kernel (TMK) são implementados diretamente pelo núcleo do Sistema Operacional, através de chamadas a rotinas do sistema que oferecem todas as funções de gerenciamento e sincronização. O Sistema Operacional sabe da existência de cada thread e pode escaloná-los individualmente. No caso de múltiplos processadores, os threads de um mesmo processo podem ser executados simultaneamente.

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.

### 6.5.3 THREADS EM MODO HÍBRIDO

Combina as vantagens do TMU e TMK. Um processo pode ter vários TMK e, por sua vez, um TMK pode ter vários TMU. O núcleo do sistema reconhece os TMK 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 TMU e especifica quantos TMK estão associados ao processo. Os TMU são mapeados em TMK enquanto o processo está sendo executado. O programador pode utilizar apenas TMK, TMU ou uma combinação de ambos.

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

## 6.6 MODELOS DE PROGRAMAÇÃO

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

Para obter os benefícios do uso de threads, uma aplicação deve permitir que partes diferentes do seu código sejam executadas em paralelo 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 gerenciados de banco de dados (SGBDs), servidores de arquivos ou impressão são exemplos onde o uso de múltiplos threads proporciona grandes vantagens e benefícios.

## 6.7 EXERCÍCIOS

- 73) Como uma aplicação pode implementar concorrência em ambiente monothread?
- 74) Quais os problemas de aplicações concorrentes desenvolvidas em ambientes monothreads?
- 75) O que é um ambiente multithread e quais as vantagens de sua utilização?
- 76) Quais as vantagens e desvantagens do compartilhamento do espaço de endereçamento entre threads de um mesmo processo?
- 77) Compare os pacotes de threads em modo usuário e modo kernel.
- 78) Dê exemplos do uso de threads no desenvolvimento de aplicativos como editores de texto e planilhas eletrônicas.
- 79) Como o uso de threads pode melhorar o desempenho de aplicações paralelas em ambientes com múltiplos processadores?
- 80) Quais os benefícios do uso de threads em ambientes cliente-servidor?
- 81) Relacione processos e threads.

-X-

# 7

## Sincronização e Comunicação entre Processos

*"... não é a posse da força que produz a vitória, mas a sua aparição.  
Um comandante habilidoso manipula não exércitos, mas percepções." (Steven Pressfield)*

### 7.1 INTRODUÇÃO

Em um sistema multiprogramável com um 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 no 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.

### 7.2 SINCRONIZAÇÃO

O Problema de Espaço na Geladeira		
Hora	Pessoa A	Pessoa B
6:00	Olha a geladeira: sem leite	-
6:05	Sai para a padaria	-
6:10	Chega na padaria	Olha a geladeira: sem leite
6:15	Sai da padaria	Sai para a padaria
6:20	Chega em casa: guarda o leite	Chega na padaria
6:25	-	Sai da padaria
6:30	-	Chega em casa: Ah! Não!

**Primeira tentativa** de resolver o Problema de Espaço na Geladeira:

Processos A e B
<pre> if (SemLeite) {     if (SemAviso) {         Deixa Aviso;         Compra Leite;         Remove Aviso;     } } </pre>

E se A for atropelada quando for comprar o leite?

**Segunda tentativa** de resolver o Problema de Espaço na Geladeira: mudar o significado do aviso. **A** vai comprar se não tiver aviso; **B** vai comprar se tiver aviso.

Processo A	Processo B
<pre>if (SemAviso) {     if (SemLeite) {         Compra Leite;     }     Deixa Aviso; }</pre>	<pre>if (Aviso) {     if (SemLeite) {         Compra Leite;     }     Remove Aviso; }</pre>

- somente **A** deixa um aviso, e somente se já não existe um aviso
- somente **B** remove um aviso, e somente se houver um aviso
- portanto, ou existe um aviso, ou nenhum
- se houver um aviso, **B** compra leite, se não houver aviso, **A** compra leite
- portanto, apenas uma pessoa (processo) vai comprar leite

Agora suponha que **B** saiu de férias. **A** vai comprar leite uma vez e não poderá comprar mais até que **B** retorne. Portanto essa "solução" não é boa; em particular ela pode levar a "inanição".

**Terceira tentativa** de resolver o Problema de Espaço na Geladeira: utilizar dois avisos diferentes.

Processo A	Processo B
<pre>Deixa AvisoA; if (SemAvisoB) {     if (SemLeite) {         Compra Leite;     } } Remove AvisoA;</pre>	<pre>Deixa AvisoB; if (SemAvisoA) {     if (SemLeite) {         Compra Leite;     } } Remove AvisoB;</pre>

A solução está quase correta. Só precisamos de uma maneira de decidir quem vai comprar o leite quando ambos deixarem avisos.

**Quarta tentativa** de resolver o Problema de Espaço na Geladeira: em caso de empate, **A** vai comprar o leite.

Processo A	Processo B
<pre>Deixa AvisoA; if (SemAvisoB) {     if (SemLeite) {         Compra Leite;     } } } else {     while (AvisoB) {         EsperaAToa;         /* pode sentar */     }     if (SemLeite) {         CompraLeite;     } } Remove AvisoA;</pre>	<pre>Deixa AvisoB; if (SemAvisoA) {     if (SemLeite) {         Compra Leite;     } } Remove AvisoB;</pre>

Esta solução funciona, mas não é muito elegante. Ela ainda tem os seguintes problemas:

- **A** pode ter que esperar enquanto **B** está na padaria
- Enquanto **A** está esperando ele está consumindo recursos.

Além disso, seria muito difícil estender esta solução para muitos processos com diferentes pontos de sincronização.

### 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 concurrentemente. 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.

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, concurrentemente 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.

```
PROGRAM A;
.
.
.
FORK B;
.
.
.
JOIN B;
.
.
.
END.

PROGRAM B;
.
.
.
END.
```

Uma das implementações mais claras e simples de expressar concorrência em um programa é a utilização dos comandos PARBEGIN e PAREND (Dijkstra, 1965). O comando PARBEGIN especifica que a seqüência de comandos seja executada concurrentemente em uma ordem imprevisível, através da criação de um processo ( $P_1, P_2, P_n$ ) para cada comando ( $C_1, C_2, C_n$ ). 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.

Para exemplificar, o programa *Expressao* realiza o cálculo do valor da expressão aritmética. Os comandos de atribuição simples situados entre PARBEGIN e PAREND são executados concurrentemente. O cálculo final de X só poderá ser realizado quando todas as variáveis dentro da estrutura tiverem sido calculadas.

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

PROGRAM Expressao;
  VAR X, Temp1, Temp2, Temp3 : REAL;
BEGIN
  PARBEGIN
    Temp1 := SQRT (1024);
    Temp2 := 35.4 * 0.23;
    Temp3 := 302 / 7;
  PAREND;
  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 comparti-

lhamento 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 corrente 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.

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. 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.

```
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);
.

.

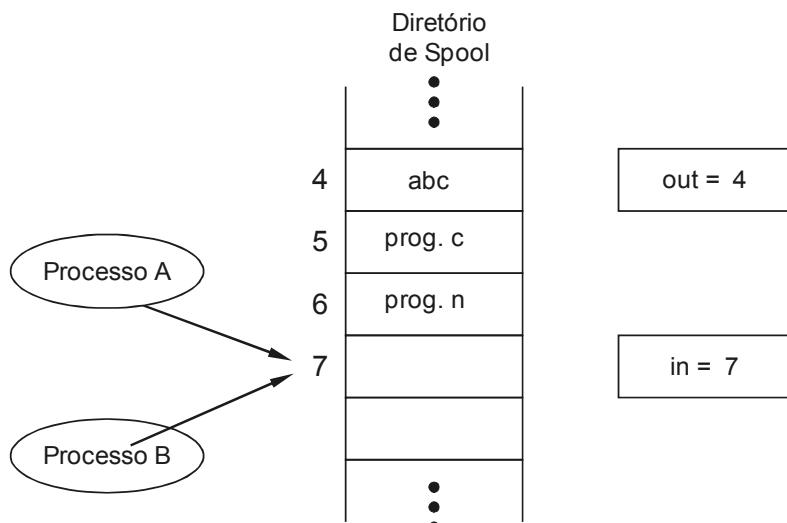
END.
```

Caixa	Comando	Saldo Arquivo	Valor dep/ret	Saldo Memória
1	READ	1000	*	1000
1	READLN	1000	-200	1000
1	:=	1000	-200	800
2	READ	1000	*	1000
2	READLN	1000	+300	1000
2	:=	1000	+300	1300
1	WRITE	800	-200	800
2	WRITE	1300	+300	1300

Um outro exemplo, seria o *spool* de impressão. Quando um processo deseja imprimir um arquivo, ele entra com o nome do arquivo em um diretório especial de *spool*. Outro processo, o impressor, verifica periodicamente este diretório, para ver se há arquivos a serem impressos. Havendo, eles são impressos e seus nomes são retirados do diretório.

Imagine que nosso diretório tenha as entradas de 0 a infinito, onde cada uma delas armazena um arquivo. Duas variáveis compartilhadas: *out* aponta para o nome do arquivo a ser impresso e *in* para a próxima entrada livre do diretório.

Mais ou menos simultaneamente, os processos A e B decidem colocar um arquivo na fila de impressão. Só que ocorre o seguinte: A lê a variável entrada e armazena o valor 7. Depois disso ocorre uma interrupção de tempo, onde o processo A deve ceder o processador para o processo B. Este lê o valor de entrada 7, mesmo valor que A obteve quando fez a leitura. B guarda o nome do seu arquivo na entrada 7 do diretório, atualizando entrada para 8, continuando seu processamento



Em algum momento seguinte, A vai reiniciar seu processamento no exato ponto onde parou. Ao examinar o valor da variável local onde guardou a entrada livre no spool de impressão, encontrará 7 e escreverá o nome do arquivo na entrada 7 do diretório, apagando o nome do arquivo escrito por B e ainda não impresso. Atualiza a variável para 8 e continua o processamento. Acontecerá que o processo B nunca terá seu arquivo impresso.

Analizando 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.

Situações como esta, onde dois ou mais processos estão acessando dados compartilhados, e o resultado final do processamento depende de quem roda quando, são chamadas de **condições de corrida**.

## 7.5 EXCLUSÃO MÚTUA

Como evitar a ocorrência de condições de corrida? A questão chave para evitar qualquer problema em compartilhamento de recursos é encontrar formas de proibir que mais de um processo acesse o dado compartilhado ao mesmo tempo.

Ou seja: implementando a **exclusão mútua**, uma forma de se ter certeza de que se um processo estiver usando uma variável ou arquivo compartilhados, os demais serão impedidos de fazer a mesma coisa.

As partes do programa, cujo processamento pode levar à ocorrência de condições de corrida, são denominadas **regiões críticas**. Evitando que dois processos estejam processando suas seções críticas ao mesmo tempo, evitaria a ocorrência de condições de corrida.

Embora essa solução impeça as condições de corrida, isso não é suficiente para que processos paralelos que usam dados compartilhados executem corretamente. Precisamos evitar também outras situações indesejadas:

1. Nenhum processo que esteja fora de sua região crítica pode bloquear a execução de outro processo
2. Nenhum processo pode esperar indefinidamente para entrar em sua região crítica (*starvation* ou espera indefinida)

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.

## 7.6 SOLUÇÕES DE HARDWARE

A exclusão mútua pode ser implementada através de mecanismos de hardware:

### 7.6.1 DESABILITAÇÃO DAS INTERRUPÇÕES

Nesta solução, ao adentrar sua região crítica, o processo desabilita as interrupções da máquina. Isto garante que outro processo não será escalonado para rodar. Imediatamente após deixar a sua região crítica, o processo habilita novamente as interrupções. Esta é uma solução funcional, porém ela compromete o nível de segurança do sistema computacional, pois se um processo desabilitar as interrupções e, por um motivo qualquer, não habilitá-las novamente, todo o sistema estará 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.

```

BEGIN

    Desabilita_Interrupcoes;
    Regiao_Critica;
    Habilita_Interrupcoes;

.

END.

```

### 7.6.2 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 **test-and-set** e tem como característica ser executada sem interrupção, ou seja, trata-se de uma instrução atômica ou 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;
BEGIN
    Bloqueio := False;
    PARBEGIN
        Processo_A;
        Processo_B;
    PARENDE;
END.

```

<pre> 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; </pre>	<pre> 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; </pre>
--	--

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.

## 7.7 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. Veremos os algoritmos de forma evolutiva até alcançar uma solução definitiva para a exclusão mútua entre N processos.

### 7.7.1 PRIMEIRO ALGORITMO: ESTRITA ALTERNÂNCIA

Este 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 Algoritmo1;
  VAR Vez : CHAR;
BEGIN
  Vez := 'A';
  PARBEGIN
    Processo_A;
    Processo_B;
  PARENDE;
END.
```

<pre>PROCEDURE Processo_A; BEGIN   REPEAT     WHILE (Vez = 'B') DO;     Regiao_Critica_A;     Vez := 'B';     Processamento_A;   UNTIL False; END;</pre>	<pre>PROCEDURE Processo_B; BEGIN   REPEAT     WHILE (Vez = 'A') DO;     Regiao_Critica_B;     Vez := 'A';     Processamento_B;   UNTIL False; END;</pre>
--	--

Este algoritmo apresenta duas limitações: 1) a primeira surge do próprio mecanismo de controle. O acesso ao recurso compartilhado só pode ser realizado por dois processos e sempre de maneira alternada. 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 mais 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. 2) 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.

### 7.7.2 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 Algoritmo2;
  VAR CA, CB : BOOLEAN;
BEGIN
  CA := false;
  CB := false;
  PARBEGIN
    Processo_A;
```

```

Processo_B;
PAREND;
END.
```

<pre> PROCEDURE Processo_A; BEGIN REPEAT   WHILE (CB) DO;   CA := true;   Regiao_Critica_A;   CA := false;   Processamento_A; UNTIL False; END;</pre>	<pre> PROCEDURE Processo_B; BEGIN REPEAT   WHILE (CA) DO;   CB := true;   Regiao_Critica_B;   CB := false;   Processamento_B; UNTIL False; END;</pre>
---	---

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. Observe abaixo:

Processo_A	Processo_B	CA	CB
WHILE (CB)	WHILE (CA)	false	false
CA := true	CB := true	true	true
Regiao_Critica_A	Regiao_Critica_B	true	true

### 7.7.3 TERCEIRO ALGORITMO:

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

```

PROGRAM Algoritmo3;
  VAR CA, CB : BOOLEAN;
BEGIN
  PARBEGIN
    Processo_A;
    Processo_B;
  PAREND;
END.
```

<pre> PROCEDURE Processo_A; BEGIN REPEAT   CA := true;   WHILE (CB) DO;   CA := true;   Regiao_Critica_A;   CA := false;   Processamento_A; UNTIL False; END;</pre>	<pre> PROCEDURE Processo_B; BEGIN REPEAT   CB := true;   WHILE (CA) DO;   CB := true;   Regiao_Critica_B;   CB := false;   Processamento_B; UNTIL False; END;</pre>
---	---

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, ambos os processos não poderão entrar em suas regiões críticas, como se o recurso já estivesse alocado.

#### 7.7.4 QUARTO ALGORITMO:

No terceiro algoritmo, cada processo altera o estado da sua variável indicando que irá entrar 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 Algoritmo4;
  VAR CA, CB : BOOLEAN;
BEGIN
  CA := false;
  CB := false;
  PARBEGIN
    Processo_A;
    Processo_B;
  PARENDE;
END.
```

<pre>PROCEDURE Processo_A; BEGIN   REPEAT     CA := true;     WHILE (CB) DO       BEGIN         CA := false;         {pequeno intervalo de tempo}         CA := true;       END       Regiao_Critica_A;       CA := false;     UNTIL False;   END;</pre>	<pre>PROCEDURE Processo_B; BEGIN   REPEAT     CB := true;     WHILE (CA) DO       BEGIN         CB := false;         {pequeno intervalo de tempo}         CB := true;       END       Regiao_Critica_B;       CB := false;     UNTIL False;   END;</pre>
--	--

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 com esta situação não sendo permanente, pode gerar alguns problemas.

#### 7.7.5 ALGORITMO DE PETERSON

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 quarto algoritmos. Esse algoritmo possui uma lógica bastante complexa. Posteriormente, G. L. Peterson propôs outra solução mais simples.

Este algoritmo apresenta uma solução para 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;
```

```

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

```

<pre> PROCEDURE Processo_A; BEGIN     REPEAT         CA := true;         Vez := 'B';         WHILE (CB and Vez = 'B') DO;         Regiao_Critica_A;         CA := false;         Processamento_A;     UNTIL False; END; </pre>	<pre> PROCEDURE Processo_B; BEGIN     REPEAT         CB := true;         Vez := 'A';         WHILE (CA and Vez = 'A') DO;         Regiao_Critica_B;         CB := false;         Processamento_B;     UNTIL False; END; </pre>
--	--

Apesar de todas soluções até então apresentadas implementarem a exclusão mútua, todas possuíam uma deficiência conhecida como **espera ocupada** (*busy waiting*). Todas as soluções baseadas na espera ocupada possuem dois problemas em comum e eles são:

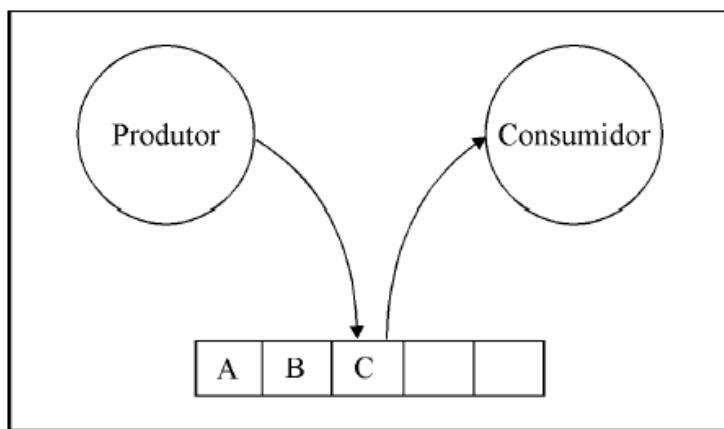
a) **Perda de tempo de processamento**: todas as soluções utilizam-se de um laço de espera ocupada para impedir que dois processos entrem ao mesmo tempo em suas regiões críticas e tais laços consomem, desnecessariamente, o tempo do processador;

b) **Prioridade invertida**: este problema ocorre quando existem dois processos executando, um "A" de alta prioridade, e outro "B" de baixa prioridade e o Gerente de Processos está programado para colocar para rodar qualquer processo de alta prioridade que esteja no estado de pronto. Objetivando verificar tal problema, considera-se que o processo "A" começa a rodar e solicita uma operação de entrada/saída e migra para o estado de bloqueado. Nesse momento, é escalonado para rodar o processo "B", que logo entra em sua região crítica. Em seguida, a operação de entrada/saída solicitada por "A" é concluída e ele, então, migra do estado de bloqueado para o de pronto e tenta adentrar em sua região crítica e fica preso no laço de espera ocupada, pois o processo "B" já se encontra dentro de sua região crítica correspondente. Diante disso, o processo "B" jamais será escalonado novamente para rodar, pois "A" está executando o seu laço de espera ocupada e o Gerente de Processos não irá escalonar "A" que, por sua vez, não deixará a sua região crítica, caracterizando, dessa forma, uma situação de bloqueio conhecida como prioridade invertida.

## 7.8 PROBLEMA DO PRODUTOR-CONSUMIDOR ou SINCRONIZAÇÃO CONDICIONAL

**Sincronização Condicional** é uma situação onde 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-la deverá permanecer bloqueado até que o recurso fique disponível.

O problema do produtor-consumidor, ilustrado na figura abaixo, consiste em dois processos que compartilham um buffer de tamanho fixo. Um dos processos, o produtor, coloca informação no buffer, e o outro, o consumidor, as retira. Assim sendo, é fácil perceber a existência de duas situações que precisam ser tratadas. A primeira delas ocorre quando o produtor deseja colocar informação no buffer e este se encontra cheio. A segunda situação acontece quando o consumidor quer retirar uma informação do buffer que está vazio.



Conforme já foi mencionado, tanto uma quanto outra situação necessita ser adequadamente tratada. Diante disso, tem-se que uma solução para a primeira das situações (buffer cheio) é fazer com que o processo produtor interrompa sua execução enquanto não existir ao menos uma posição vazia no buffer. Uma solução para a segunda situação (buffer vazio) é manter o consumidor parado enquanto não houver pelo menos uma informação no buffer.

O programa a seguir exemplifica o problema. 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   : Tipo_Dado;
      Cont   : 0..TamBuf;
BEGIN
  Cont := 0;
  PARBEGIN
    Produtor;
    Consumidor;
  PARENDE;
END.

```

<pre> PROCEDURE Produtor; BEGIN   REPEAT     Produz_Dado (Dado);     WHILE (Cont = TamBuf) DO;       Grava_Buffer (Dado, Cont);       Cont++;     UNTIL False;   END; </pre>	<pre> PROCEDURE Consumidor; BEGIN   REPEAT     WHILE (Cont = 0) DO;       Le_Buffer (Dado);       Consome_Dado (Dado, Cont);       Cont--;     UNTIL False;   END; </pre>
--	---

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

## 7.9 SEMÁFOROS

Um semáforo é uma variável inteira, não-negativa, que só pode ser manipulada por duas instruções atômicas: DOWN e UP. A instrução UP incrementa uma unidade ao valor do semáforo, enquanto DOWN decremente 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.

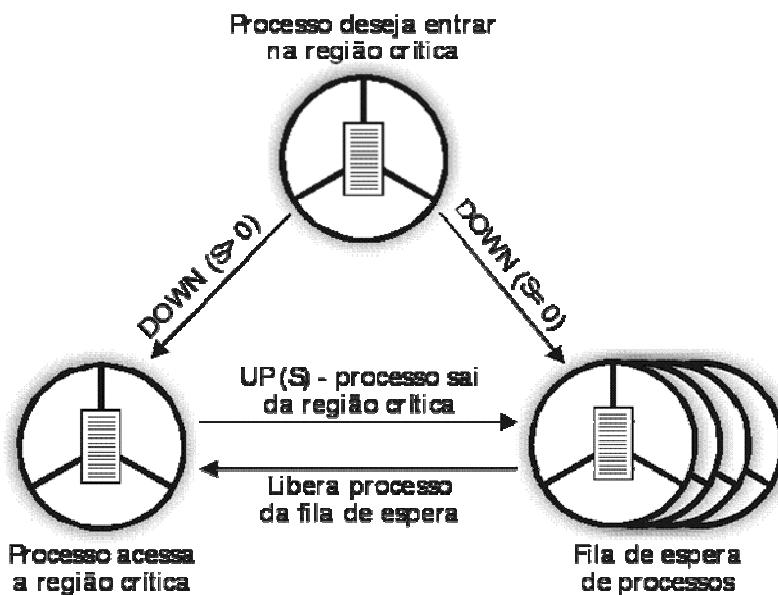
Podem ser classificados como binários ou contadores. Os **binários**, também chamados de **mutexes** (*mutual exclusion semaphores*), só podem assumir os valores 0 e 1, enquanto os **contadores** podem assumir qualquer valor inteiro positivo, além do 0.

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, isso 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*.

### 7.9.1 EXCLUSÃO MÚTUA UTILIZANDO SEMÁFOROS

A exclusão mútua pode ser implementada através de um semáforo binário associado a um 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, o sistema selecionará um processo na fila de espera associada ao recurso e alterará o seu estado para pronto.

As instruções DOWN e UP, aplicadas a um semáforo S, podem ser representadas pelas definições a seguir, em uma sintaxe Pascal não convencional:

```

TYPE Semaforo = RECORD
  Valor : INTEGER;
  Fila_Espera : (*Lista de processos pendentes*);
END;

```

<pre> PROCEDURE DOWN (Var S : Semaforo); BEGIN   IF (S = 0) THEN Coloca_Proc_Fila   ELSE     S := S + 1; END; </pre>	<pre> PROCEDURE UP (Var S : Semaforo); BEGIN   S := S + 1;   IF(Tem_Proc_Espera) THEN(Tira_Proc); END; </pre>
--	---

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;
BEGIN
  PARBEGIN
    Processo_A;
    Processo_B;
  PARENDE;
END.

```

<pre> PROCEDURE Processo_A; BEGIN   REPEAT     DOWN (S);     Regiao_Critica_A;     UP (S);     Processamento_A;   UNTIL False; END; </pre>	<pre> PROCEDURE Processo_B; BEGIN   REPEAT     DOWN (S);     Regiao_Critica_B;     UP (S);     Processamento_B;   UNTIL False; END; </pre>
--	--

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, mas como seu valor é igual a 0, ficará aguardando até que o Processo A execute a instrução UP, ou seja, volte o valor semáforo para 1.

### 7.9.2 SINCRONIZAÇÃO CONDICIONAL UTILIZANDO SEMÁFOROS

O problema do produtor/consumidor é um exemplo de como a exclusão mútua e a sincronização condicional podem ser implementadas com o uso de semáforos. O programa ProdutorConsumidor2 apresenta uma solução para esse problema, utilizando um buffer de apenas duas posições. O programa utiliza três semáforos, sendo um do tipo binário, utilizado 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;
    Dado1 : Tipo_Dado;
    Dado2 : Tipo_Dado;
BEGIN
  PARBEGIN
    Produtor;
    Consumidor;
  PAREND;
END.

```

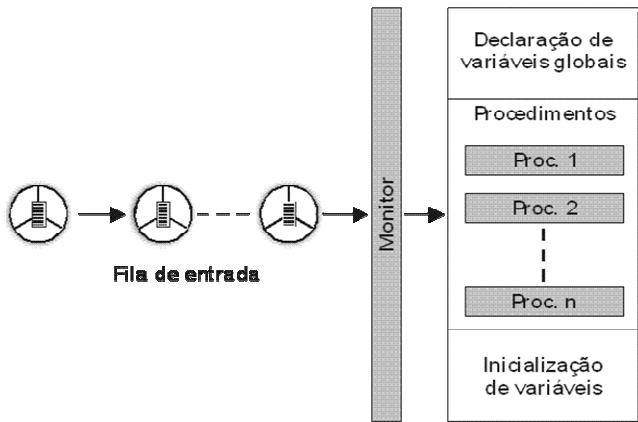
<pre> PROCEDURE Produtor; BEGIN   REPEAT     Produz_Dado (Dado_1);     DOWN (Vazio);     DOWN (Mutex);     Grava_Buffer (Dado_1, Buffer);     UP (Mutex);     UP (Cheio);   UNTIL False; END; </pre>	<pre> PROCEDURE Consumidor; BEGIN   REPEAT     DOWN (Cheio);     DOWN (Mutex);     Lie_Buffer (Dado_2, Buffer);     UP (Mutex);     UP (Vazio);     Consome_Dado (Dado_2);   UNTIL False; END; </pre>
--	---

Por exemplo, consideremos 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.

## 7.10 MONITORES

São mecanismos de sincronização de alto nível que tornam mais simples o desenvolvimento de aplicações concorrentes. 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.

Monitores são estruturados em função de serem implementados pelo compilador. Assim, o desenvolvimento de programas concorrentes fica mais fácil e as chances de erro são menores. Atualmente, a maioria das linguagens de programação disponibiliza rotinas para uso de monitores.



O monitor é formado por procedimentos e variáveis encapsuladas 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.

**7.10.1 EXCLUSÃO MÚTUA UTILIZANDO MONITORES**

A implementação 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 seguinte problema: 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;
  PAREN;
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.

### **7.10.2 SINCRONIZAÇÃO CONDICIONAL UTILIZANDO MONITORES**

Monitores também podem ser utilizados na implementação da sincronização condicional. Através 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 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.

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).

Sempre que o processo produtor deseja gravar um dado no buffer, deverá fazê-lo através de uma chamada ao procedimento Produz do monitor (Condisional.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 pro-

cesso consumidor executar um SIGNAL na variável Cheio, indicando que um elemento do buffer for 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 (Condisional.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 recuperção 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 Condisional;
VAR Vazio, cheio : (* Variáveis de condição *);
    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;

BEGIN
    PARBEGIN
        Produtor;
        Consumidor;
    PARENDE;
END.

```

<pre> PROCEDURE Produtor; BEGIN     REPEAT         Produz_Dado (Dado);         Condisional.Produz;     UNTIL False; END; </pre>	<pre> PROCEDURE Consumidor; BEGIN     REPEAT         Condisional.Consome         Consome_Dado (Dado);     ;     UNTIL False; END; </pre>
---	--

## 7.11 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.

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

```
PROGRAM Produtor_Consumidor_4;
BEGIN
    PARBEGIN
        Produtor;
        Consumidor;
    PARENDE;
END.
```

<pre>PROCEDURE Produtor; VAR Msg : Tipo_Msg; BEGIN     REPEAT         Produz_Mensagem (Msg);         SEND (Msg);     UNTIL False; END;</pre>	<pre>PROCEDURE Consumidor; VAR Msg : Tipo_Msg; BEGIN     REPEAT         RECEIVE (Msg);         Consome_Mensagem (Msg);     UNTIL False; END;</pre>
--	--

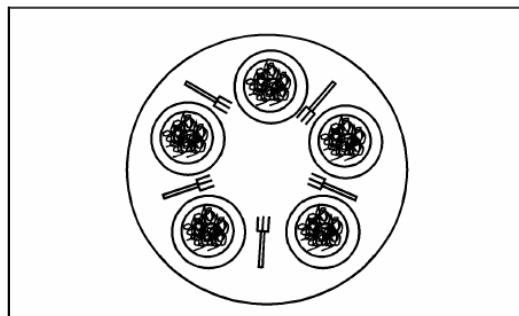
## 7.12 PROBLEMAS CLÁSSICOS DE SINCRONIZAÇÃO

A literatura sobre sistemas operacionais está repleta de problemas interessantes que têm sido amplamente discutidos e analisados a partir de vários métodos de sincronização.

### 7.12.1 PROBLEMA DOS FILÓSOFOS GLUTÕES

Este problema, conforme ilustra a figura abaixo, considera a existência de cinco filósofos sentados em torno de uma mesa redonda. Cada filósofo tem à sua frente um prato de macarrão.

Um filósofo que deseja comer precisa contar com o auxílio de 2 garfos e entre cada prato existe um garfo. A vida de um filósofo consiste em períodos nos quais ele alternadamente come e pensa. Quando um filósofo sente fome, ele tenta tomar posse dos garfos à sua direita e à sua esquerda, um por vez, em qualquer ordem. Se ele tiver sucesso nesta empreitada, come um pouco do macarrão, libera os garfos e continua a pensar. A questão a ser respondida é: como escrever um programa para simular a ação de cada filósofo que nunca leve a situações de bloqueio?



### 7.12.2 PROBLEMA DO BARBEIRO DORMINHOCO

Neste problema, considera-se uma barbearia onde existe um barbeiro, uma cadeira de barbeiros e diversos lugares onde os clientes que estiverem esperando, poderão sentar. Se não houver nenhum cliente presente, o barbeiro simplesmente senta em sua cadeira e cai no sono. Quando chegarem fregueses, enquanto o barbeiro estiver cortando o cabelo de outro, estes devem ou sentar, se houver cadeira vazia, ou ir embora, se não houver nenhuma cadeira disponível.

#### 7.13 EXERCÍCIOS

- 82) Defina o que é uma aplicação concorrente e dê um exemplo de sua utilização.
- 83) Relacione Região Crítica com Condição de Corrida.
- 84) 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?
- 85) O que é exclusão mútua e como é implementada?
- 86) Como seria possível resolver os problemas decorrentes do compartilhamento da matriz, apresentado anteriormente, utilizando o conceito de exclusão mútua?
- 87) O que é *starvation* e como podemos solucionar esse problema?
- 88) Qual o problema com a solução que desabilita as interrupções para implementar a exclusão mútua?
- 89) O que é espera ocupada e qual o seu problema?
- 90) Quais são os problemas comuns a todas as soluções que se utilizam da espera ocupada para garantir a exclusão mútua de execução?
- 91) Explique o que é sincronização condicional e dê um exemplo de sua utilização.
- 92) Explique o que são semáforos e dê dois exemplos de sua utilização: um para a solução da exclusão mútua e outro para sincronização condicional.
- 93) Explique o que são monitores e dê dois exemplos de sua utilização: um para a solução da exclusão mútua e outro para sincronização condicional.
- 94) 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)	Processo 2 (Cliente B)
<pre>/* saque em A */ 1a. x := saldo_cliente_A; 1b. x := x - 200; 1c. saldo_cliente_A := x;  /* deposito em B */ 1d. x := saldo_cliente_B; 1e. x := x + 100; 1f. saldo_cliente_B := x;</pre>	<pre>/* saque em A */ 2a. y := saldo_cliente_A; 2b. y := y - 100; 2c. saldo_cliente_A := y;  /* deposito em B */ 2d. y := saldo_cliente_B; 2e. y := y + 200; 2f. saldo_cliente_B := y;</pre>

Supondo que os valores dos saldos de A e B sejam, respectivamente, 500 e 900, antes de os processos executarem, pede-se:

- Quais os valores corretos esperados para os saldos dos clientes A e B após o término da execução dos processos?
- Quais os valores finais dos saldos dos clientes se a seqüência temporal de execução das operações for: 1a, 2a, 1b, 2b, 1c, 2c, 1d, 2d, 1e, 2e, 1f, 2f?
- 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.

95) 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;
```

<pre>PROCEDURE Escritor; BEGIN     ProduzDado;     DOWN (Acesso);     Escreve;     UP (Acesso); END;</pre>	<pre>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;</pre>
--	---

- Suponha que exista apenas um leitor fazendo acesso à base. Enquanto este processo realiza a leitura, quais os valores das três variáveis?
- 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?
- Chega mais um leitor enquanto o primeiro ainda não acabou de ler e o escritor está bloqueado. Descreva os valores das três variáveis enquanto o segundo leitor inicia a leitura.
- Os dois leitores terminam simultaneamente a leitura. É possível haver problemas quanto à integridade do valor da variável Nleitores? Justifique.
- 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.
- 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.
- Quando o escritor houver terminado a atualização, é possível prever qual dos processos bloqueados (leitor ou escritor) terá acesso primeiro à base?
- Descreva uma situação onde os escritores sofram (adiamento indefinido) *starvation*.

96) Um outro problema clássico de comunicação entre processos é o problema dos fumantes (Patil, 1971). Três classes de fumantes estão juntas em uma sala acompanhadas do vendedor de ingredientes para a fabricação de cigarros. Para fazer e fumar um cigarro, cada fumante precisa de três ingredientes: tabaco, papel e fósforos, todos eles disponíveis em quantidades infinitamente grandes, guardadas pelo vendedor. Uma das categorias de fumantes tem seu próprio tabaco, ou-

tra seu papel e a última seus fósforos. A ação começa quando o vendedor coloca dois ingredientes sobre a mesa, para permitir que um dos fumantes inicie um procedimento extremamente nocivo à sua saúde (fumar). Quando este fumante estiver servido, ele acorda o vendedor para que ele coloque mais dois ingredientes na mesa, escolhido randomicamente, desbloqueando outro fumante. Tente pensar nesse problema e veja se encontra alguns dilemas.

- 97) Outro problema clássico é o da padaria, definido por Lamport (1974). Neste problema, uma padaria tem vários pães e tortas sendo vendidos por  $n$  vendedores. Cada cliente que entra na padaria recebe um número e espera até que seu número seja chamado. Sempre que um vendedor fica desocupado, o próximo número é chamado. Pense nos procedimentos relativos aos vendedores e aos clientes.

-X-

# 8

## Deadlock

*"A educação é o nosso passaporte para o futuro  
pois o amanhã pertence a quem se prepara hoje." (Malcom X)*

### 8.1 INTRODUÇÃO

Um conjunto de N processos está em *deadlock* quando cada um dos N processos está bloqueado à espera de um evento que somente pode ser causado por um dos N processos do conjunto. Obviamente, essa situação somente pode ser alterada por alguma iniciativa que parte de um processo fora do conjunto dos N processos.

Ou ainda: um processo em um sistema multiprogramado é dito estar em uma situação de *deadlock* quando ele está esperando por um evento particular que jamais ocorrerá. Em um *deadlock* em um sistema, um ou mais processos estão em *deadlock*.

Em sistemas multiprogramados, o compartilhamento de recursos é uma das principais metas dos sistemas operacionais. Quando recursos são compartilhados entre uma população de usuários, e cada usuário mantém controle exclusivo sobre os recursos particulares a ele alocados, é possível que haja a ocorrência de *deadlocks* no sentido em que alguns usuários jamais sejam capazes de terminar seu processamento.

Talvez a forma de ilustrar um *deadlock* seja com um exemplo de uma lei aprovada pela assembléia norte-americana no início deste século: "Quando dois trens se aproximarem um do outro em um cruzamento, ambos deverão parar completamente e nenhum dos dois deverá ser acionado até que o outro tenha partido."

### 8.2 EXEMPLOS DE DEADLOCKS

#### 8.2.1 DEADLOCK DE TRÁFEGO

Um certo número de automóveis está tentando atravessar uma parte da cidade bastante movimentada, mas o tráfego ficou completamente paralisado. O tráfego chegou numa situação onde somente a polícia pode resolver a questão, fazendo com que alguns carros recuem na área congestionada. Eventualmente o tráfego volta a fluir normalmente, mas a essa altura os motoristas já se aborreceram e perderam tempo considerável.

#### 8.2.2 DEADLOCK SIMPLES DE RECURSOS

Muitos *deadlocks* ocorrem em sistemas de computação devido à natureza de recursos dedicados (isto é, os recursos somente podem ser usados por um usuário por vez).

Suponha que em um sistema o processo A detém um recurso 1, e precisa alocar o recurso 2 para poder prosseguir. O processo B, por sua vez, detém o recurso 2, e precisa do recurso 1 para poder prosseguir. Nesta situação, temos um *deadlock*, porque um processo está esperando pelo outro. Esta situação de espera mútua é chamada muitas vezes de **espera circular**.

#### 8.2.3 DEADLOCK EM SISTEMAS DE SPOOLING

Um sistema de *spooling* serve, por exemplo, para agilizar as tarefas de impressão do sistema. Ao invés do aplicativo mandar linhas para impressão diretamente para a impressora, ele as manda para o *spool*, que se encarregará de enviá-las para a impressora. Assim o aplicativo é rapidamente liberado da tarefa de imprimir.

Em alguns sistemas de *spool*, todo o *job* de impressão deve ser gerado antes do início da impressão. Isto pode gerar uma situação de *deadlock*, uma vez que o espaço disponível em disco para a área de *spooling* é limitado. Se vários processos começarem a gerar seus dados para o *spool*, é possível que o espaço disponível para o *spool* fique cheio antes mesmo de um dos *jobs* de impressão tiver terminado de ser gerado. Neste caso, todos os processos ficarão esperando pela liberação do espaço em disco, o que jamais vai

acontecer. A solução nesse caso seria o operador do sistema cancelar um dos *jobs* parcialmente gerados.

Para resolver o problema sem a intervenção do operador, o SO poderia alocar uma área maior de *spooling*, ou a área de *spooling* poderia ser variável dinamicamente. Alguns sistemas, como o do Windows 3.x/95, utilizam todo o espaço em disco disponível. Entretanto, pode acontecer de o disco possuir pouco espaço e o problema ocorrer da mesma forma.

A solução definitiva seria implementar um sistema de *spooling* que começasse a imprimir assim que algum dado estivesse disponível, sem a necessidade de se esperar por todo o *job*.

#### **8.2.4 ADIAMENTO INDEFINIDO**

Em sistemas onde processos ficam esperando pela alocação de recursos ou pelas decisões de escalonamento, é possível que ocorra adiamento indefinido também chamado de bloqueamento indefinido (ou starvation).

Adiamento indefinido pode ocorrer devido às políticas de escalonamento de recursos do sistema, principalmente quando o esquema utiliza prioridades (conforme já vimos). Isto pode ser evitado permitindo que a prioridade de um processo em espera cresça conforme ele espera por um recurso.

### **8.3 RECURSOS**

Um sistema operacional pode ser visto de forma mais ampla como um gerenciador de recursos. Ele é responsável pela alocação de vários recursos de diversos tipos.

Aos objetos que os processos podem adquirir daremos o nome de recursos, para generalizar. Um recurso pode ser um dispositivo de hardware, como por exemplo, uma unidade de fita, ou uma informação, tal como um registro em uma base de dados. Um recurso é algo que só pode ser usado por um único processo em um determinado instante de tempo.

Existem os recursos preemptíveis e os não-preemptíveis. Um recurso preemptível é aquele que pode ser tomado do processo que estiver usando o recurso, sem nenhum prejuízo para o processo. A memória e a CPU são exemplos.

Recursos não-preemptíveis não podem ser tomados de processos aos quais foram alocados. Têm que ser executados até o fim. Exemplo, se um processo iniciou a impressão de resultados, a impressora não poderá ser tomada dele e entregue a outro processo, sem que haja um prejuízo considerável no processamento do primeiro processo.

Alguns recursos são compartilhados entre vários processos. Unidades de disco são compartilhadas em geral. Memória principal e CPU são compartilhadas; apesar de que em um instante a CPU pertence a um único processo, mas sua multiplexação entre os vários processos transmite a idéia de que está sendo compartilhada.

Dados e programas são certamente os recursos que mais precisam ser controlados e alocados. Em um sistema multiprogramado, vários usuários podem querer simultaneamente usar um programa editor. Seria desperdício de memória ter-se uma cópia do editor para cada programa executando. Ao contrário, uma única cópia do código é trazida para a memória e várias cópias dos dados são feitas, uma para cada usuário. Uma vez que o código está em uso por vários usuários simultaneamente, ele não pode mudar. Códigos que não podem mudar enquanto estão em uso são ditos reentrantes. Código que pode ser mudado mas que é reinicializado cada vez que é usado é dito seriamente reusável. Código reentrante pode ser compartilhado entre vários processos, enquanto código seriamente reusável só pode ser alocado a um processo por vez.

Recursos que tendem a estar envolvidos em deadlocks são aqueles que podem ser usados por vários processos, mas um de cada vez, ou seja: o recurso não-preemptivo.

Um sistema possui um número finito de recursos para serem distribuídos entre processos concorrentes. Os recursos são classificados segundo vários tipos, sendo que cada tipo pode consistir de uma quantidade de instâncias idênticas. Por exemplo, se conside-

rarmos o tipo de recurso CPU, em uma máquina com dois processadores, temos duas instâncias do recurso CPU.

Se um processo requisita uma instância de um tipo de recurso, a alocação de qualquer instância daquele tipo irá satisfazer a requisição. Se em um determinado sistema esta satisfação não ocorrer, isto significa que as instâncias não são idênticas, e que as classes de tipos de recursos não estão definidas corretamente. Por exemplo, suponha que um sistema possui duas impressoras. Elas poderiam ser definidas como instâncias de um mesmo tipo de recurso. Entretanto se uma estivesse instalada no quarto andar, e outra no térreo do prédio, os usuários do andar térreo podem não enxergar as duas impressoras como sendo equivalentes. Neste caso, classes de recursos separadas precisariam ser definidas para cada impressora.

Um processo pode requisitar um recurso antes de usá-lo, e deve liberá-lo depois de seu uso. Um processo pode requisitar quantos recursos precisar para desempenhar a tarefa para a qual foi projetado. Obviamente, o número de recursos requisitados não pode exceder o número total de recursos disponíveis no sistema.

Em uma situação de operação normal, um processo pode utilizar um recurso somente nesta seqüência:

1) **Requisitar**: se a requisição não pode ser atendida imediatamente (por exemplo, o recurso está em uso por outro processo), então o processo requisitante deve esperar até obter o recurso; 2) **Usar**: O processo pode operar sobre o recurso (por exemplo, se o recurso é uma impressora, ele pode imprimir) e 3) **Liberar**: O processo libera o recurso

#### 8.4 QUATRO CONDIÇÕES NECESSÁRIAS PARA DEADLOCKS

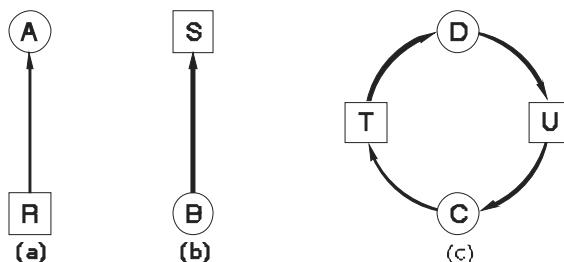
Coffman, Elphick e Shosani (1971) enumeraram as seguintes quatro condições necessárias que devem estar em efeito para que um deadlock exista:

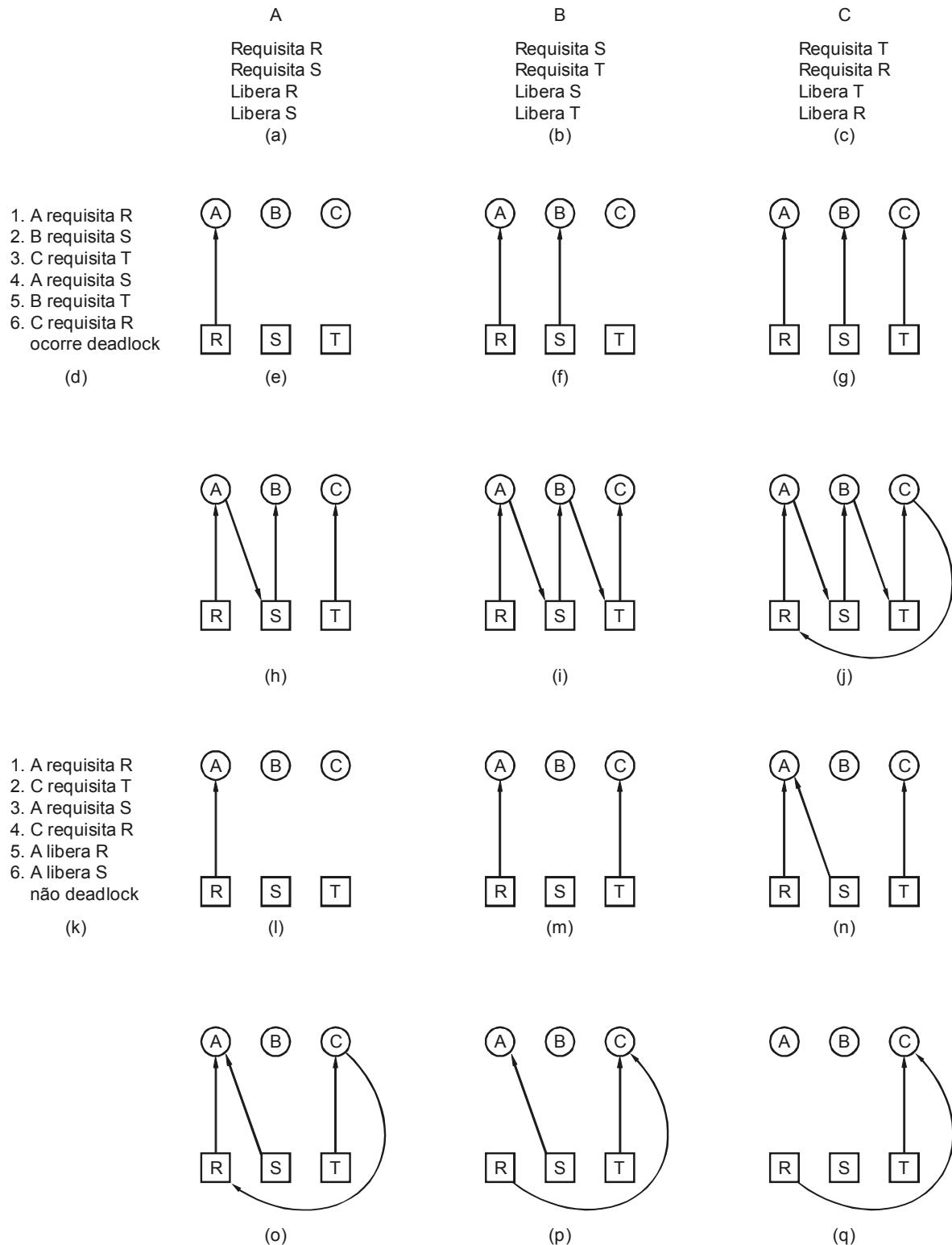
- 1) **Condição de exclusão mútua**: cada recurso ou está alocado a exatamente um processo ou está disponível
- 2) **Condição de posse e de espera**: processos que estejam de posse de recursos obtidos anteriormente podem solicitar novos recursos
- 3) **Condição de não-preempção**: recursos já alocados a processos não podem ser tomados à força. Eles precisam ser liberados explicitamente pelo processo que detém sua posse.
- 4) **Condição de espera circular**: deve existir uma cadeia circular de dois ou mais processos, cada um dos quais esperando por um recurso que está com o próximo membro da cadeia

Todas as quatro condições acima citadas devem estar presentes para que possa ocorrer *deadlock*. Se uma delas estiver ausente, não há a menor possibilidade de ocorrer uma situação de *deadlock*. Isto nos ajudará a desenvolver esquemas para prevenir *deadlocks*.

#### 8.5 O MODELO DO DEADLOCK

As condições de deadlock podem, de acordo com Holt (1972), ser modeladas com o uso de grafos dirigidos. Tais grafos tem dois tipos de nós: processos, representados por círculos e recursos, representados por quadrados.





Grafos de alocação de recursos. (a) Um processo de posse de um recurso.  
 (b) Um processo solicitando um recurso. (c) Deadlock

## 8.6 MÉTODOS PARA LIDAR COM DEADLOCKS

**Evitar** dinamicamente o *deadlock*, pela cuidadosa alocação dos recursos aos processos. Isto requer que seja fornecida ao sistema operacional informações adicionais sobre quais recursos um processo irá requisitar e usar durante sua execução.

**Prevenir** o *deadlock* através da negação de uma das quatro condições necessárias para que ocorra um *deadlock*

**Detectar e recuperar** uma situação de *deadlock*. Se não são usadas estratégias de prevenção ou para evitar *deadlocks*, existe a possibilidade de ocorrência destes. Neste ambiente, o sistema operacional pode possuir um algoritmo que consiga determinar se ocorreu um *deadlock*, além de um algoritmo que faça a recuperação da situação.

**Ignorar** completamente o problema. Se um sistema que nem previne, evita ou recupera situações de *deadlock*, se um ocorrer, não haverá maneira de saber o que aconteceu exatamente. Neste caso, o *deadlock* não detectado causará a deterioração do desempenho do sistema, porque recursos estão detidos por processos que não podem continuar, e porque mais e mais processos, conforme requisitam recursos, entram em *deadlock*. Eventualmente o sistema irá parar de funcionar e terá que ser reinicializado manualmente.

Apesar do método de ignorar os *deadlocks* não parecer uma abordagem viável para o problema da ocorrência de *deadlocks*, ele é utilizado na maioria dos sistemas operacionais, inclusive o UNIX. Em muitos sistemas, *deadlocks* não ocorrem de forma freqüente, como por exemplo, uma vez por ano. Assim, é muito mais simples e "barato" usar este método do que os dispendiosos meios de prevenir, evitar, detectar e recuperar *deadlocks*.

## 8.7 O ALGORITMO DO AVESTRUZ

É a abordagem mais simples: "enterre a cabeça na areia e pense que não há nenhum problema acontecendo".

A estratégia do UNIX em relação aos *deadlocks* é simplesmente ignorar o problema, pressupondo que a maioria dos usuários vai preferir a ocorrência de *deadlocks* ocasionais, em vez de regras que forcem todos os usuários a só usar um processo, um arquivo aberto, enfim, um exemplar de cada recurso. Se os *deadlocks* pudessem ser eliminados de graça, não haveria nenhuma discussão a respeito da conveniência de tal abordagem. O problema é que o preço de sua eliminação é alto, sobretudo pelo fato de se estabelecerem restrições inconvenientes ao uso dos recursos do sistema. Desta forma, estamos diante de uma decisão não muito agradável, entre a conveniência e a correção.

## 8.8 DETECÇÃO DE DEADLOCKS

Uma segunda técnica é detectar e recuperar os *deadlocks*. Quando ela é usada, o sistema não se preocupa em prevenir a ocorrência de *deadlocks*. Em vez disso, ele permite que os mesmos ocorram, tenta detectar as ocorrências, e age no sentido de normalizar a situação, após sua ocorrência.

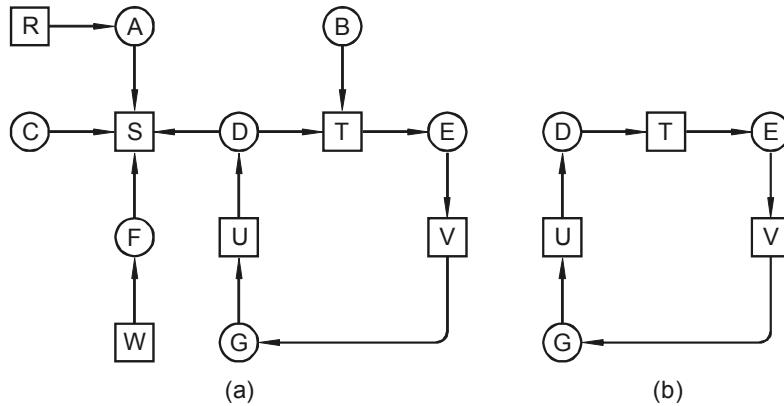
### 8.8.1 DETECÇÃO DO DEADLOCK COM UM RECURSO DE CADA TIPO

Vamos começar com a existência de um recurso de cada tipo. Ou seja, o sistema pode ter uma impressora, um *plotter* e uma unidade de fita, mas não duas impressoras, por exemplo.

Para esse método, construímos um grafo de recursos conforme já foi visto. Se tal grafo contiver um ou mais ciclos, estará garantida pelo menos uma situação de *deadlock*. Se não houver ciclos, o sistema não estará em *deadlock*.

Na figura, observamos que o grafo possui um ciclo, que pode ser notado pela simples inspeção visual. Portanto, os processos D, E e G estão em *deadlock*. Apesar de ser relativamente simples eliminar a condição de *deadlock*, através do gráfico, esse método não pode ser usado nos sistemas reais, onde há necessidade de um algoritmo para essa tarefa.

O algoritmo que usaremos como ilustração, é um dos mais simples, que inspeciona um grafo e termina quando encontra um ciclo ou quando não encontra nenhum. Ele usa uma estrutura de dados  $L$ , que é uma lista de nós. Durante a execução do algoritmo, os arcos serão marcados para indicar que já foram inspecionados.



Ele toma cada um dos nós, em ordem, como se fosse a raiz de uma árvore e faz uma pesquisa profunda nela. Se alguma vez ele visitar um nó onde já tenha estado anteriormente, então ele encontrou um ciclo. Ao exaurir todos os arcos de um determinado nó, ele volta para o nó anterior. Se esta propriedade valer para todos, o grafo inteiro estará livre de ciclos, de forma que o sistema não apresenta condição de deadlock.

Começando por R, inicializando L como uma lista vazia. Acrescentamos R à lista, e nos deslocamos para o único nó possível, A, adicionando-o também a L, fazendo então  $L=[R,A]$ . De A vamos para S, formando  $L=[R,A,S]$ . Como não há nenhum arco saindo dele, significa que chegamos a um beco sem saída, de maneira que somos forçados a voltar para A. Como A não tem nenhum arco desmarcado saindo dele, voltamos para R, completando nossa inspeção a partir de R.

Agora recomeçamos o algoritmo partindo de A, fazendo com que L volte a ser uma lista vazia. Esta busca também vai terminar rapidamente, de modo que vamos recomeçar, só que agora partindo de B. Daí continuamos seguindo os arcos que saem deste nó, até encontrarmos D, quando  $L=[B,T,E,V,G,U,D]$ . Neste instante, devemos fazer uma escolha aleatória entre S e T. Se escolhermos S, alcançamos um beco sem saída, e voltamos a D, de onde voltamos a T, caracterizando um ciclo, razão pela qual o algoritmo deve parar. Se escolhermos T, estaremos visitando este nó pela segunda vez, e ao atualizarmos a lista para  $L=[B,T,E,V,G,U,D,T]$ , identificaremos um ciclo e o algoritmo pára.

## **8.8.2 DETECÇÃO DO DEADLOCK COM VÁRIOS RECURSOS DE CADA TIPO**

Quando existem várias cópias de alguns dos recursos, é necessária a adoção de uma abordagem diferente para a detecção de *deadlocks*. Usaremos um algoritmo baseado numa matriz.

Vamos denominar **E** de vetor de recursos existentes. Ele fornece o número total de instâncias de cada recurso existente. Por exemplo, se a classe 1 refere-se à unidade de fita,  $E_1 = 2$ , significa que há duas unidades de fitas no sistema.

Seja  $\mathbf{D}$  o vetor de recursos disponíveis, com  $D_i$  fornecendo o número de instâncias do recurso  $i$  atualmente disponíveis. Se ambas as unidades de fita estiverem alocadas,  $D_1 = 0$ .

Agora necessitamos de duas matrizes: **C**, a matriz de alocação corrente e **R**, a matriz de requisções.

Como exemplo, vamos considerar a figura abaixo, onde existem três processos e quatro classes de recursos, as quais chamamos arbitrariamente de unidades de fita, *plotters*, impressoras e unidades de CD-ROM.

O processo 1 está de posse de uma impressora, o processo 2 de duas unidades de fita e de uma de CD-ROM, e o processo 3 tem um *plotter* e duas impressoras. Cada um dos processos precisa de recursos adicionais, conforme mostra a matriz R.

Recursos existentes				Recursos disponíveis			
Unidade de fita	Processadores	Impressoras	Unidades de CD ROM	Unidade de fita	Processadores	Impressoras	Unidades de CD ROM
$E = (4 \quad 2 \quad 3 \quad 1)$				$D = (2 \quad 1 \quad 0 \quad 0)$			
Matriz de alocação corrente				Matriz de requisições			
$C = \begin{bmatrix} 0 & 0 & 1 \\ 2 & 0 & 0 \\ 0 & 1 & 2 \end{bmatrix}$	0	1	0	$R = \begin{bmatrix} 2 & 0 & 0 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{bmatrix}$	1	0	0

Para rodar este algoritmo, procuramos por um processo cujas necessidades de recursos possam ser satisfeitas. As necessidades do processo 1 não poderão ser atendidas, pois não há unidades de CD ROM disponíveis. O processo 2 também não terá suas requisições atendidas por falta de impressora livre. Felizmente, as requisições do processo 3 poderão ser satisfeitas, de maneira que ele será posto para rodar, ficando  $D = (0,0,0,0)$ , e ao final de seu processamento, devolverá seus recursos ao pool de recursos disponíveis, fazendo com que  $D = (2,2,2,0)$ .

Neste momento o processo 2 pode rodar, ficando  $D=(1,2,1,0)$ , e ao devolver seus recursos fará  $D=(4,2,2,1)$ . Nesta situação o último processo poderá rodar, fazendo com que não haja *deadlock* no sistema.

Se acontecer que algum processo necessite de um recurso a mais do que os disponíveis, todo o sistema estará em *deadlock*.

Bem, agora já sabemos como detectar *deadlocks*, mas, quando devemos procurar por eles? Podemos fazer isso a cada vez que um novo recurso for solicitado, mas custará muito caro, apesar de detectarmos o *deadlock* o mais cedo possível. Uma estratégia alternativa é a de verificar a cada k minutos, ou quando o nível de utilização do processador atingir um determinado nível muito baixo.

## 8.9 RECUPERAÇÃO DE DEADLOCKS

Já vimos como detectar deadlocks, e agora, como recuperar o sistema?

### 8.9.1 RECUPERAÇÃO ATRAVÉS DA PREEMPÇÃO

Algumas vezes é possível tomar temporariamente um recurso de seu atual dono e entregá-lo a outro processo. Outras vezes é necessária a intervenção manual. A capacidade de tomar um recurso de um processo, deixar que outro processo o utilize e devolvê-lo ao processo original, sem que este tenha conhecimento do que está acontecendo, depende muito do recurso.

Por exemplo, para tomar uma impressora de seu proprietário atual, o operador deve recolher as folhas impressas e colocá-las numa pilha. O processo pode então ser suspenso. A impressora pode ser entregue a um outro processo. Quando este terminar, a pilha de folhas impressas do processo anterior deve ser colocada de novo na bandeja de saída, sendo o processo reiniciado.

Agora, interromper a gravação em uma unidade de CD-ROM pode se tornar muito complicado.

### 8.9.2 RECUPERAÇÃO ATRAVÉS DE VOLTA AO PASSADO

Consiste em escrever históricos dos processos em andamento, contendo todas as informações necessárias para que em caso de *deadlock*, o sistema possa voltar atrás e alocar um ou outro recurso diferente para evitar o impasse novamente.

### 8.9.3 RECUPERAÇÃO ATRAVÉS DE ELIMINAÇÃO DE PROCESSOS

É a maneira mais rude, mas também a mais simples de se eliminar a situação de *deadlock*.

Por exemplo, um processo pode estar de posse de uma impressora e estar precisando de um *plotter*, e outro processo pode estar precisando da impressora, estando com o *plotter*. Estão em *deadlock*. A eliminação de um outro processo que possui tanto a impressora quanto o *plotter* pode solucionar esse impasse.

O ideal é que seja eliminado um processo que possa rodar de novo desde o início sem produzir nenhum efeito negativo ao sistema. Por exemplo, uma compilação pode sempre recomeçar, pois tudo o que ela faz é ler um arquivo-fonte e produzir um arquivo-objeto. Sua eliminação fará com que a primeira rodada não tenha influência alguma na segunda. Por outro lado, um processo que atualiza uma base de dados não pode ser eliminado e voltar a rodar uma segunda vez em condições seguras.

## 8.10 TENTATIVAS DE EVITAR O DEADLOCK

Se o sistema for capaz de decidir se a alocação de determinado recurso é ou não segura, e só fazer a alocação quando ela for segura, teremos como evitar *deadlocks*.

### 8.10.1 ESTADOS SEGUROS E INSEGUROS

Para escrever algoritmos que evitem *deadlocks*, precisamos usar informações como os Recursos existentes e os disponíveis, bem como as Matrizes de alocação corrente e a de Requisições.

Um estado é dito seguro se não provocar *deadlock*, e houver uma maneira de satisfazer todas as requisições pendentes partindo dos processos em execução.

Vamos exemplificar usando apenas um recurso. Na figura, temos um estado no qual A tem três instâncias de um recurso, mas pode eventualmente precisar de nove. No momento, B tem duas e pode precisar ao todo de quatro. De maneira similar, C tem duas, mas pode precisar adicionalmente de mais cinco. Existe um total de 10 instâncias deste recurso, estando sete alocadas e três livres.

Possui Máximo			Possui Máximo			Possui Máximo			Possui Máximo			Possui Máximo		
A	3	9	A	3	9	A	3	9	A	3	9	A	3	9
B	2	4	B	4	4	B	0	-	B	0	-	B	0	-
C	2	7	C	2	7	C	2	7	C	7	7	C	0	-
Livre: 3			Livre: 1			Livre: 5			Livre: 0			Livre: 7		
(a)			(b)			(c)			(d)			(e)		

O estado mostrado primeiramente (Fig. a) é seguro, pois existe uma seqüência de alocações que permitem que todos os processos venham a terminar seus processamentos. Para tanto, o escalonador pode rodar B exclusivamente, até que ele precise de mais duas instâncias do recurso, levando ao estado da Fig. b. Quando B termina, chegamos ao estado da Fig. c. Então o escalonador pode rodar C, chegando ao estado da Fig. d. Quando C termina, atingimos o estado ilustrado na Fig. 3E. Agora A pode obter as seis instâncias adicionais do recurso, necessárias a que ele termine sua execução. Então o estado inicial é considerado seguro, pois o sistema, através de um escalonamento cuidadoso, pode evitar a ocorrência de *deadlocks*.

Se por acaso, A requisita e ganha um outro recurso na Fig. b. Podemos encontrar uma seqüência de alocações que funcione garantidamente. Não. Podemos, então, concluir que a requisição de A por um recurso adicional não deveria ser atendida naquele momento.

Um estado inseguro não leva necessariamente a um *deadlock*. A diferença entre um estado seguro e um inseguro é que, de um estado seguro, o sistema pode garantir que todos os processos vão terminar, enquanto que a partir de um inseguro esta garantia não pode ser dada.

### 8.10.2 ALGORITMO DO BANQUEIRO PARA UM ÚNICO TIPO DE RECURSO

Dijkstra (1965) idealizou um algoritmo de escalonamento que pode evitar a ocorrência de *deadlocks*, conhecido como algoritmo do banqueiro. Ele baseia-se nas mesmas

premissas adotadas por um banqueiro de um pequeno banco para garantir ou não crédito a seus correntistas.

Usando a mesma analogia mostrada no item acima, considerando que os clientes são os processos e as unidades de crédito representam os recursos, e o banqueiro faz o papel do sistema operacional.

	Possui Máximo		
	A	B	C
A	0	6	
B	0	5	
C	0	4	
D	0	7	

Livre: 10

(a)

	Possui Máximo		
	A	B	C
A	1	6	
B	1	5	
C	2	4	
D	4	7	

Livre: 2

(b)

	Possui Máximo		
	A	B	C
A	1	6	
B	2	5	
C	2	4	
D	4	7	

Livre: 1

(c)

Três cenários de alocação de recursos: (a) Seguro. (b) Seguro. (c) Inseguro.

O mesmo algoritmo pode ser generalizado para tratar com diversos tipos de recursos, utilizando matrizes.

Todos os livros sérios de Sistemas Operacionais descrevem tal algoritmo com detalhes. Apesar de teoricamente maravilhoso, este algoritmo na prática é inútil, pois os processos quase nunca sabem com antecedência a quantidade de recursos de que vão precisar. Além do mais, tal quantidade não é fixa e os recursos que estavam disponíveis em determinado momento podem "sumir" repentinamente (uma unidade de fita pode quebrar).

## 8.11 PREVENÇÃO DE DEADLOCKS

Se pudermos garantir que pelo menos uma das quatro condições necessárias para que um *deadlock* ocorra nunca será satisfeita, poderemos garantir que será estruturalmente impossível a ocorrência de *deadlocks* (Havender, 1968).

### 8.11.1 ATACANDO O PROBLEMA DA EXCLUSÃO MÚTUA

Se não houver possibilidade de nenhum recurso ser entregue exclusivamente a um único processo, nunca teremos configurada uma situação de *deadlock*. No entanto, está claro também que dar permissão a dois processos para acessar a mesma impressora ao mesmo tempo vai levar a uma situação caótica.

Evitar então, entregar um recurso, quando ele não for absolutamente necessário, e tentar estar certo de que o mínimo possível de processos está precisando de recursos.

### 8.11.2 ATACANDO O PROBLEMA DA POSSE E DA ESPERA

Se pudermos impedir processos de manter a posse de recursos enquanto esperam por outros recursos, poderemos eliminar os *deadlocks*. Uma forma de se alcançar este objetivo é exigir que todos os processos requisitem todos os recursos de que precisam, antes de iniciar a execução.

Um problema imediato que pode ocorrer com esta abordagem é o fato de muitos processos não conhecerem com antecedência quantos e quais os recursos necessários à sua execução.

### 8.11.3 ATACANDO O PROBLEMA DA CONDIÇÃO DE NÃO-PREEMPÇÃO

A análise da terceira condição (não-preempção) revela-se ainda menos promissora do que a segunda. Se um processo tem alocado uma impressora e está no meio da impressão de seus resultados, tomar à força a impressora porque um *plotter* de que ele vai precisar não está disponível é uma grande besteira.

### 8.11.4 ATACANDO O PROBLEMA DA ESPERA CIRCULAR

Resta-nos somente uma condição para tentar resolver a questão dos *deadlocks*. A condição de espera circular pode ser eliminada de diversas maneiras. Uma delas é sim-

plesmente seguindo a regra de que um processo só está autorizado a usar apenas um recurso por vez. Se ele precisar de um segundo recurso, deve liberar o primeiro.

Outra forma é utilizar uma numeração global para todos os recursos. Agora a regra é: processos podem solicitar recursos sempre que necessário, mas todas as solicitações precisam ser feitas em ordem numérica.

Resumindo:

CONDIÇÃO	ABORDAGEM
Exclusão Mútua	Alocar todos os recursos usando <i>spool</i>
Posse e Espera	Requisitar todos os recursos inicialmente
Não-Preempção	Tomar recursos de volta
Espera Circular	Ordenar numericamente os recursos

## 8.12 EXERCÍCIOS

- 98) Quando é que um processo está em *deadlock*?
- 99) Descreva o *deadlock* em sistemas de *spooling*.
- 100) Dentro do contexto de *deadlock*, o que é a espera circular?
- 101) Um sistema operacional pode ser visto de forma mais ampla como um gerenciador de recursos. Ele é responsável pela alocação de vários recursos de diversos tipos. O que seriam recursos nesse contexto?
- 102) Diferencie recursos preemptíveis e não-preemptíveis.
- 103) Em sistemas onde processos ficam esperando pela alocação de recursos ou pelas decisões de escalonamento, é possível que ocorra adiamento indefinido (ou *starvation*) e consequentemente um *deadlock*. O que vem a ser isso?
- 104) Existem quatro condições necessárias que devem estar presentes para que possa ocorrer deadlock. Se uma delas estiver ausente, não há a menor possibilidade de ocorrer uma situação de impasse. Quais são elas? Cite e explique cada uma.
- 105) De acordo com Havender, 1968, se pudermos garantir que pelo menos uma das quatro condições necessárias para que um deadlock ocorra (exclusão mútua, posse e espera, não-preempção e espera circular) nunca sejam satisfeitas, podemos garantir que será estruturalmente impossível a ocorrência dos mesmos. Faile como isso poderia ser feito considerando o contexto descrito.
- 106) Existem três formas de recuperar o sistema depois da detecção de um deadlock: recuperação através da preempção, através da volta ao passado e através da eliminação de processos. Fale sobre cada uma delas.
- 107) Considerando a seguinte matriz de alocação de recursos, onde  $p$  é o número de recursos que determinado processo possui,  $m$  é o número máximo de recursos que esse processo pode requerer e  $L$  o número de recursos Livres no sistema computacional diga se o estado é seguro ou não e demonstre isso através das matrizes. Considere que o sistema computacional tenha 10 recursos disponíveis:

	p	m
A	1	7
B	1	5
C	2	4
D	4	9
L=	2	

- 108) Quando existem várias cópias de alguns recursos em um sistema computacional, é necessário a adoção de uma abordagem que utiliza vetores e matrizes para a detecção de deadlocks. Supondo que em um sistema computacional, existam três processos e quatro classes de recursos: 5 unidades de fita, 3 plotters, 4 impressoras e 2 unidades de CD-ROM. Considerando E o vetor de recursos existentes, D o vetor de recursos disponíveis, C a matriz de alocação corrente dos processos e R a matriz de requisições, demonstre, através da alocação cuidadosa de recursos, se existe ou não a ocorrência de deadlock. Lembre-se que é preciso inicialmente, preencher o vetor D.

$$E = (5 \ 3 \ 4 \ 2) \quad D = ( ) \\ C = \begin{pmatrix} 2 & 1 & 0 & 1 \\ 1 & 0 & 2 & 0 \\ 0 & 2 & 1 & 0 \end{pmatrix} \quad R = \begin{pmatrix} 3 & 0 & 2 & 1 \\ 1 & 0 & 0 & 1 \\ 4 & 1 & 1 & 2 \end{pmatrix}$$

- 109) Quando existem várias cópias de vários recursos em um sistema computacional, é necessária a adoção de uma abordagem que utiliza vetores e matrizes para a detecção de deadlocks. Considerando E o vetor de recursos existentes, D o vetor de recursos disponíveis, C a matriz de alocação corrente dos processos e R a matriz de requisições, responda:

- Se existir cinco processos sendo executados, quantas linhas terão as matrizes C e R?
- Em que situação o vetor E é igual ao vetor D?
- O que está acontecendo se a primeira linha da matriz C e a primeira linha da matriz R estiverem zeradas?
- Respondendo em função dos valores das matrizes e dos vetores, em que situação o sistema detectaria um deadlock?

-X-

# 9

## Gerência do Processador

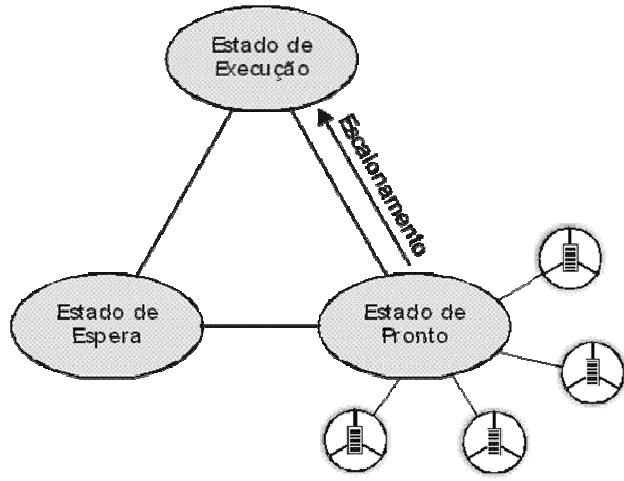
*"Precisamos nos especializar para entrar mais profundamente em certos aspectos separados da realidade. Mas se a especialização é absolutamente necessária, pode ser absolutamente fatal, se levada longe demais." (Aldous Huxley).*

### 9.1 INTRODUÇÃO

Nos Sistemas Operacionais Multiprogramados, sempre que o processador fica sem utilização, ocorre o acionamento do Gerente de Processos ou Escalonador de Processos que tem como principal função selecionar o processo que deve ocupar o processador. Dessa maneira, o trabalho do Escalonador de Processos pode permitir que o computador se torne mais produtivo através da otimização do uso do processador.

Assim sendo, o escalonamento de processos constitui a base dos Sistemas Operacionais Multiprogramados e, portanto, faz-se importante conhecer suas principais características, bem como os algoritmos de escalonamento mais utilizados.

### 9.2 FUNÇÕES BÁSICAS



A política de escalonamento de um Sistema Operacional possui diversas funções básicas, como a de manter o processador ocupado a maior parte do tempo, balancear o uso da CPU 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, possuem requisitos de escalonamento distintos dos sistemas de tempo real.

A rotina do Sistema Operacional que tem como principal função implementar os critérios da política de escalonamento é chamada **escalonador** (*scheduler*). Em um sistema multiprogramável, o escalonador é fundamental, pois todo o compartilhamento do processador é dependente dessa rotina.

Outra rotina importante na gerência do processador é conhecida como **despachante** (*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 despachante**.

### 9.3 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*. Já em sistemas de tempo real, o escalonamento deve priorizar a execução de processos críticos em detrimento da execução de outros processos.

a) **Utilização do processador:** o processador deve permanecer ocupado o máximo de tempo possível. Deve ser frisado que quando o sistema computacional trabalha com

uma taxa de utilização do processador por volta de 30%, ele é considerado um sistema com carga baixa de processamento. Porém, quando o mesmo trabalha com uma taxa na faixa de 90%, ele é considerado um sistema computacional bastante carregado;

b) **Throughput ou vazão:** representa o número máximo de processos executados em um determinado intervalo de tempo o número de processos que são executados em uma determinada fração de tempo, geralmente uma hora, deve ser elevado ao máximo.

c) **Tempo de processador ou Tempo de CPU:** é o tempo que o 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.

c) **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.

d) **Tempo de retorno (turnaround):** é o tempo que um processo leva desde sua criação até o seu término, considerando-se o tempo gasto na alocação de memória, na fila de processos prontos, nas operações de entrada/saída e, ainda, o seu tempo de processamento. Deve ser o menor possível;

e) **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.

De uma maneira geral, qualquer política de escalonamento busca otimizar a utilização do processador e a vazão, enquanto tenta diminuir os tempos de retorno, 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.

#### 9.4 ESTRATÉGIAS DE ESCALONAMENTO

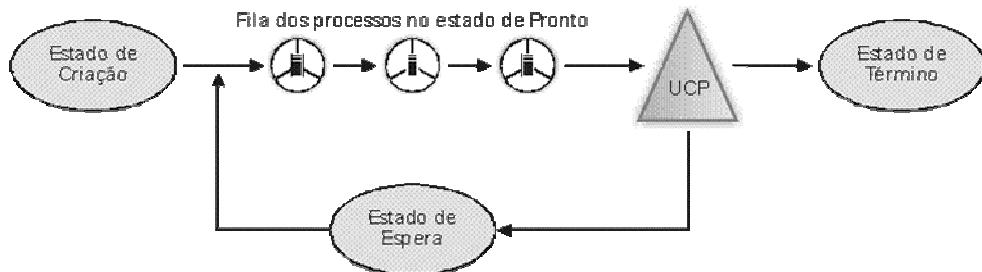
As estratégias de escalonamento de processos são definidas em função da atividade de preempção, a qual consiste na capacidade de permitir a interrupção da execução de um processo para executar um outro, sem prejuízo à lógica de execução de ambos.

As estratégias de escalonamento são duas:

a) **Escalonamento não-preemptivo:** com esta estratégia um processo que entra no processador roda até terminar, sem jamais ser interrompido. Este foi o primeiro tipo de escalonamento desenvolvido e foi utilizado nos SOs de processamento em batch;

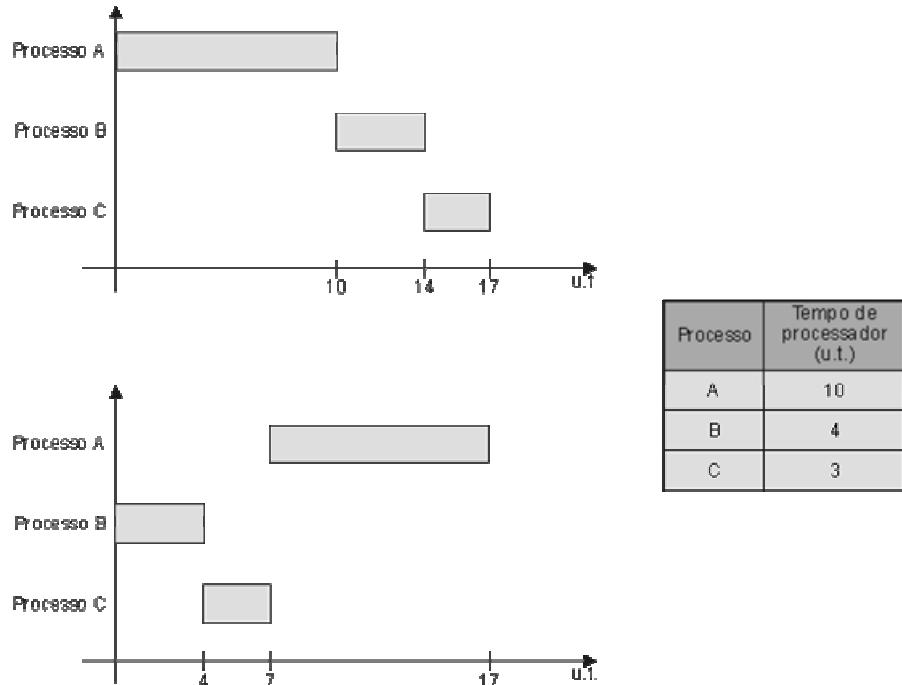
b) **Escalonamento preemptivo:** esta estratégia é baseada na atividade de preempção, ou seja, permite a suspensão temporária da execução de um processo para outro rodar, sem prejuízo lógico de execução a ambos. A maioria dos SOs da atualidade utiliza esta estratégia de escalonamento.

#### 9.5 ESCALONAMENTO FIRST COME FIRST SERVED (FCFS) ou FIFO



Neste algoritmo os processos são organizados em uma fila por ordem de chegada e toda vez que for necessário escalonar um processo para rodar, o Gerente de Processos seleciona o primeiro da fila. Os processos que forem chegando para serem executados são colocados no final da fila, conforme ilustrado pela Figura anterior.

A próxima Figura apresenta um exemplo de utilização do algoritmo FIFO. Nela é possível observar a existência de três processos a serem executados, obedecendo à ordem de chegada de cada um, e, ainda, os seus respectivos tempos estimados e finais de execução.



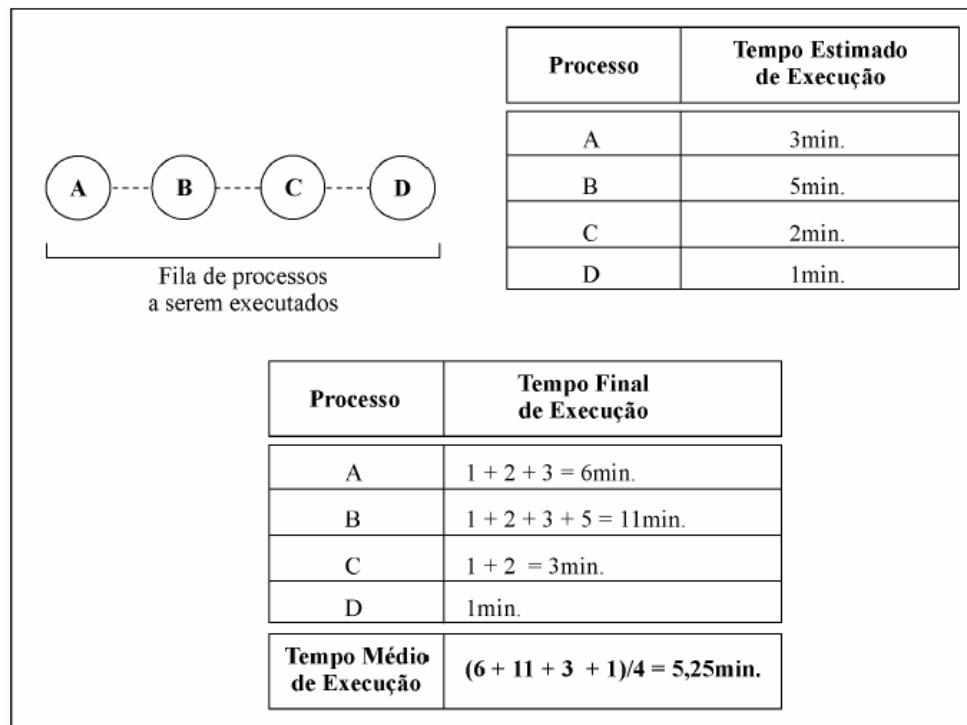
A principal vantagem de utilização do FIFO é a sua simplicidade de implementação, haja visto que o Gerente de Processos necessita apenas manter uma fila de processos e escalonar para rodar um após o outro. Contudo, o algoritmo em questão apresenta algumas deficiências e a mais importante delas é a impossibilidade de se saber ao certo qual é o tempo de *turnaround* de um processo. Um outro problema existente é que o FIFO é um algoritmo não preemptivo e, portanto, a sua aplicação em sistemas computacionais que possuam muitos processos de usuários interativos, como os da atualidade, é considerada ineficiente.

#### 9.6 ESCALONAMENTO MENOR JOB PRIMEIRO ou SJF (Shortest Job First)

O algoritmo de escalonamento do menor trabalho primeiro (*Shortest Job First - SJF*) possui uma política que estabelece, conforme evidenciado na Figura abaixo, que o processo com menor tempo estimado de execução é aquele que deve rodar primeiro. Assim sendo, este algoritmo parte do pressuposto que o tempo de execução de cada processo é conhecido.

Na sua concepção inicial, o escalonamento SJF é preemptivo. Sua vantagem sobre o escalonamento FIFO está na redução do tempo médio de retorno 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 **escalonamento de tempo remanescente**. Nessa 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.



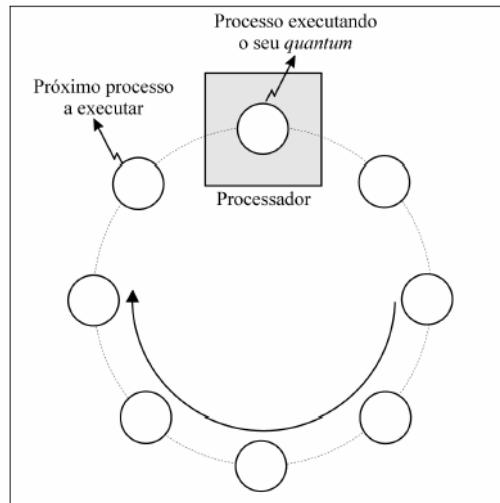
A principal vantagem do algoritmo em questão é que ele privilegia processos de usuários interativos. Assim, estes usuários podem, por exemplo, conseguir obter respostas rápidas aos seus comandos.

Já sua desvantagem principal é que, normalmente, não se conhece, a priori, o tempo que um processo irá ocupar o processador e este fato dificulta a implementação do SJF em sua forma original.

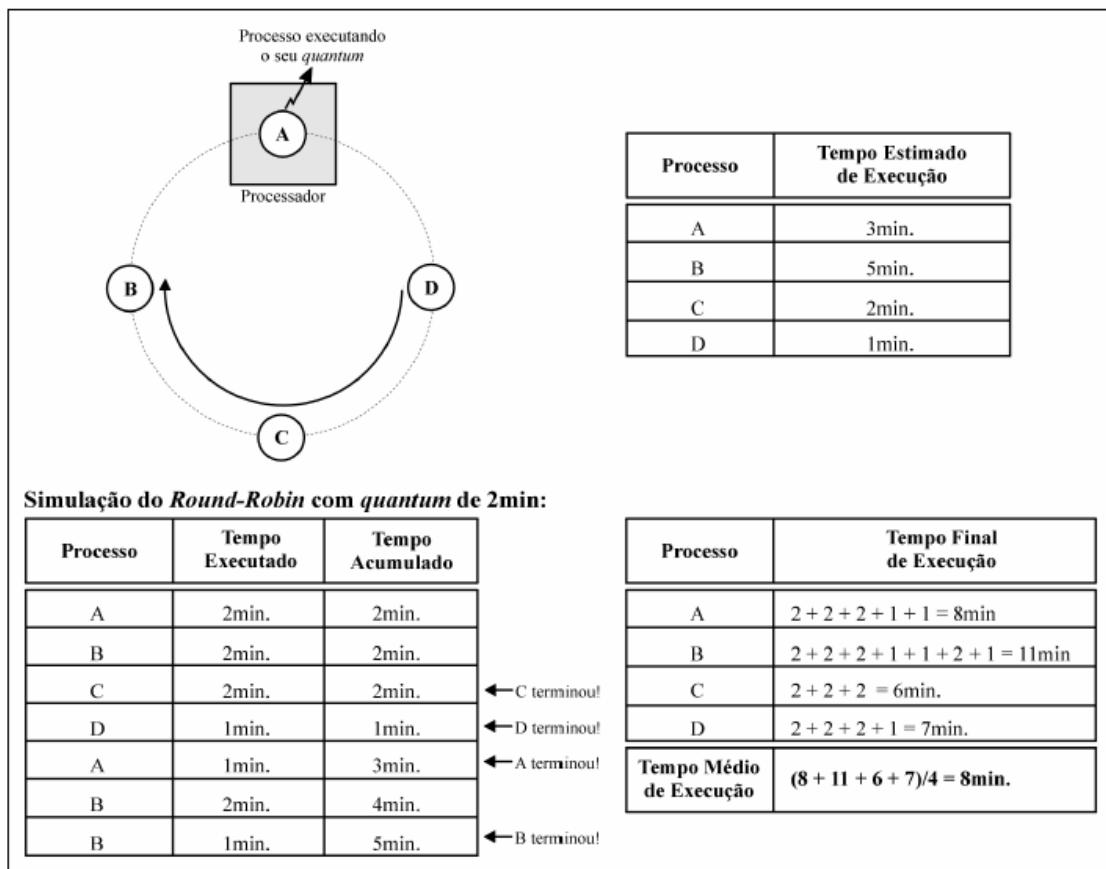
#### 9.7 ESCALONAMENTO CIRCULAR ou ROUND ROBIN

É um escalonamento do tipo preemptivo, projetado especialmente para sistemas de tempo compartilhado. Neste algoritmo, a cada processo atribui-se um intervalo de tempo, chamado de fatia de tempo (*time-slice*) ou quantum, durante o qual ele poderá usar o processador.

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. Este mecanismo é conhecido como **preempção por tempo**.



Objetivando facilitar a compreensão do funcionamento do *Round-Robin*, a Figura a seguir apresenta um exemplo de utilização do mesmo.

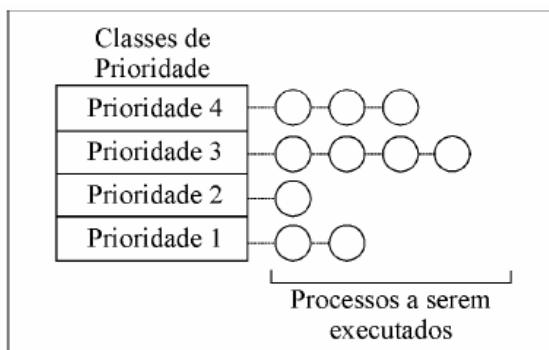


O Round-Robin apresenta como vantagens principais a sua simplicidade de implementação e, ainda, o fato de não permitir que processos monopolizem o processador. Já a sua desvantagem mais evidente é que ele não faz distinção entre a prioridade dos processos. Assim sendo, um processo de tempo-real vai competir pelo processador em condições de igualdade com um processo que seja um jogo de entretenimento, por exemplo. Outra desvantagem é a dificuldade de se definir o tamanho do quantum, pois se este for muito pequeno, ocorrerão sucessivas trocas de contexto, baixando a eficiência do SO. Por outro lado, se o quantum for muito grande, os usuários interativos ficarão insatisfeitos.

Obs.: O valor do quantum depende do projeto de cada SO e, geralmente, ele se encontra entre 10 e 100 milissegundos.

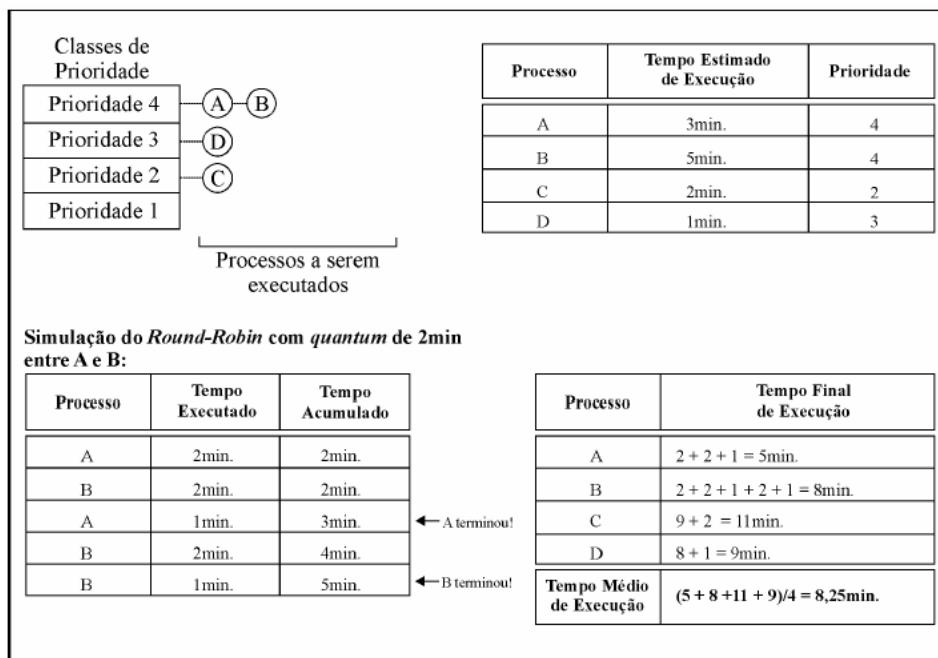
## 9.8 ESCALONAMENTO POR PRIORIDADES

O algoritmo em questão é preemptivo e considera fatores externos para escolher qual processo irá rodar em um dado momento. Assim, cada processo deve possuir uma prioridade de execução a fim de que o Gerente de Processos defina qual processo irá rodar.



Normalmente, o processo com maior prioridade é aquele que deve ser executado primeiro. Contudo, existindo processos com prioridades iguais, os mesmos devem ser agrupados em classes e o Round-Robin é, geralmente, aplicado nas mesmas para determinar qual processo de uma classe irá rodar. A figura mostrada como tal algoritmo funciona.

A próxima Figura apresenta um exemplo de uso do algoritmo de escalonamento com prioridade.

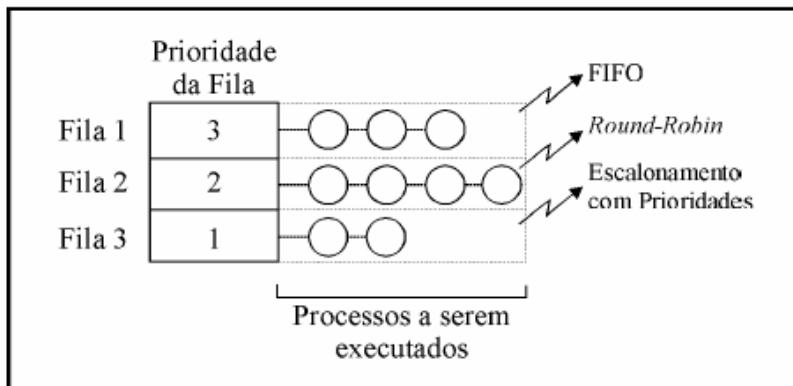


O escalonamento com prioridades apresenta como principal ponto positivo o fato de permitir a diferenciação dos processos segundo critérios de importância. Este fato torna-se bastante relevante quando se considera o projeto de um Sistema Operacional de Tempo Real, por exemplo.

A desvantagem de utilização do algoritmo em estudo é a possibilidade de os processos de baixa prioridade nunca serem escalonados, caracterizando uma situação conhecida como *starvation* (adiamento indefinido). Contudo, esta desvantagem pode ser superada através do uso da técnica de *aging* (envelhecimento), a qual consiste em incrementar gradativamente a prioridade dos processos que ficam muito tempo sem rodar.

## 9.9 ESCALONAMENTO POR MÚLTIPLAS FILAS

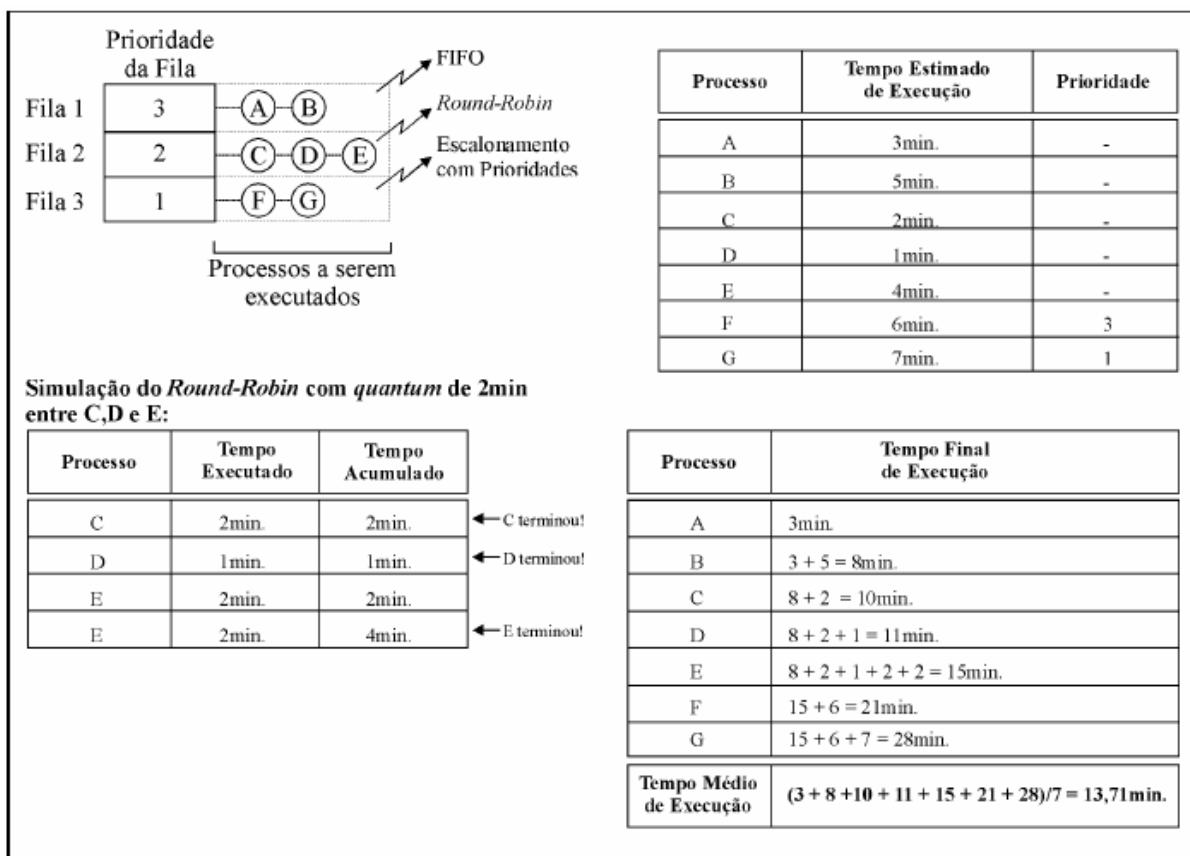
Este algoritmo preemptivo, como o próprio nome sugere, considera a existência de várias filas de processos no estado de pronto, cada uma delas com sua prioridade de execução e seu algoritmo de escalonamento. Os processos são associados às filas em função de características particulares como, por exemplo, tipo de processo, área de memória ocupada etc.



Na Figura a seguir, é encontrado um exemplo de utilização do escalonamento com filas múltiplas.

A principal vantagem deste algoritmo é que ele permite que se utilize um algoritmo de escalonamento diferente para cada fila. Assim, é possível que uma fila utilize o FIFO, enquanto outra usa o Round-Robin, por exemplo.

Como desvantagem deste algoritmo tem-se o fato de que sua implementação é mais complexa que a dos outros já estudados, pois cada uma das filas de prioridade pode utilizar um algoritmo de escalonamento diferente.



## 9.10 EXERCÍCIOS

- 110) O que é política de escalonamento de um Sistema Operacional?
- 111) Qual é a função do escalonador e do despachante?
- 112) Quais os principais critérios utilizados em uma política de escalonamento?
- 113) Diferencie os tempos de processador, espera, retorno e resposta.
- 114) Qual é a diferença entre escalonamento preemptivo e não-preemptivo?
- 115) Qual a diferença entre os escalonamentos FIFO e Circular?
- 116) O que é fatia de tempo?
- 117) Descreva o escalonamento SJF e o escalonamento por prioridades.
- 118) 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 CPU. 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
procA	P	P	E	E	E	P	P	P	E	W
procB	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
procA	P	E	P	P	E	E	W	W	P	E
procB	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.
- 119) Tomando como base os dados da tabela apresentada a seguir, ignorando a perda de tempo causada pelas trocas de contexto e considerando que nenhum processo realiza entrada/saída, informe o tempo que cada um dos processos efetivamente levará para executar se os seguintes algoritmos de escalonamento forem utilizados: a) Round-Robin com quantum de 2 minutos; b) SJF; c) Escalonamento com Prioridade com quantum de 2 minutos.

Processo	Tempo CPU	Prioridade
A	1 min	4
B	7 min	2
C	5 min	3
D	4 min	4
E	3 min	1

- 120) Explique o funcionamento da técnica de envelhecimento (*aging*) e quando ela pode ser utilizada.
- 121) Verificamos uma situação na qual um processo A, de alta prioridade, e um processo B de baixa prioridade interagem de forma a levar A a um loop eterno. Tal situação persistiria se utilizássemos o escalonamento round robin em vez do escalonamento com prioridade?
- 122) Os escalonadores round robin normalmente mantêm uma fila com todos os processos prontos, com cada processo aparecendo uma vez nesta fila. O que pode acontecer se determinado processo aparecer duas vezes na fila de prontos? Você pode encontrar alguma razão para que isto seja permitido?
- 123) Que tipos de critérios devem ser utilizados no momento da definição da fatia de tempo a ser empregada em um determinado sistema?
- 124) Em um sistema operacional, o escalonador de curto prazo utiliza duas filas. A fila "A" contém os processos do pessoal do CPD e a fila "B" contém os processos dos alunos. O algoritmo entre filas é fatia de tempo. De cada 11 unidades de tempo de processador, 7 são fornecidas para os processos da fila "A", e 4 para os processos da fila "B". O tempo de cada fila é dividido entre os processos também por fatias de tempo, com fatias de 2 unidades para todos. A tabela abaixo mostra o conteúdo das duas filas no instante zero. Considere que está iniciando um ciclo de 11 unidades, e agora a fila "A" vai receber as suas 7 unidades de tempo. Mostre a sequência de execução dos processos, com os momentos em que é feita a troca.  
OBS: Se terminar a fatia de tempo da fila "X" no meio da fatia de tempo de um dos processos, o processador passa para a outra fila. Entretanto, esse processo permanece como primeiro da fila "X", até que toda sua fatia de tempo seja consumida.

Fila	Processo	Tempo de CPU
A	P1	6
	P2	5
	P3	7
B	P4	3
	P5	8
B	P6	4

- 125) Quatro programas devem ser executados em um computador. Todos os programas são compostos por 2 ciclos de processador e 2 ciclos de E/S. A entrada e saída de todos os programas é feita sobre a mesma unidade de disco. Os tempos para cada ciclo de cada programas são mostrados na tabela. Construa um diagrama de tempo mostrando qual programa está ocupando o processador e o disco a cada momento, até que os 4 programas terminem. Suponha que o algoritmo de escalonamento utilizado seja fatia de tempo, com fatias de 4 unidades. Qual a taxa de ocupação do processador e do disco?

Programa	Processador	Disco	Processador	Disco
P1	3	10	3	12
P2	4	12	6	8
P3	7	8	8	10
P4	6	14	2	10

- 126) Um algoritmo de escalonamento de CPU determina a ordem para execução dos seus processos escalonados. Considerando  $n$  processos a serem escalonados em um processador, quantos escalonamentos diferentes são possíveis? Apresente uma fórmula em termos de  $n$ .
- 127) Considere um Sistema Operacional que implemente escalonamento circular com quantum igual a 10 ut. Em um determinado instante de tempo, existem apenas três processos (P1, P2 e P3) na fila de pronto, e o tempo de CPU de cada processo é 18, 4 e 13 ut, 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)  $T = 8$  ut      b)  $T = 11$  ut      c)  $T = 33$  ut
- 128) Considere um Sistema Operacional que implemente escalonamento circular com quantum igual a 10 ut. Em um determinado instante de tempo, existem apenas três processos (P1, P2 e P3) na fila de pronto, e o tempo de CPU de cada processo é 14, 4 e 12 ut, 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 ut e consome 10 ut.  
 a)  $T = 8$  ut      b)  $T = 18$  ut      c)  $T = 28$  ut
- 129) Considere as seguintes tabelas. Elabore o gráfico de Gantt para os algoritmos SJF preemptivo, RR e prioridade. Calcule os tempos médios de retorno, resposta e espera para cada uma das situações. Considere uma fatia de tempo de 2 instantes.

PROC	Tcheg	Prior	Tcpu	Tresp	Tret	Tesp
A	12	4	7			
B	5	3	3			
C	2	1	2			
D	13	2	4			
E	14	5	1			

PROC	Tcheg	Prior	Tcpu	Tresp	Tret	Tesp
A	8	5	7			
B	2	1	3			
C	15	2	2			
D	2	4	4			
E	3	3	1			
F	11	6	5			

PROC	Tcheg	Prior	Tcpu	Tresp	Tret	Tesp
A	12	1	5			
B	5	4	9			
C	2	3	4			
D	13	5	3			
E	8	0	6			
F	6	6	1			
G	1	2	7			

130) Qual a vantagem em ter diferentes tamanhos de quantum em diferentes níveis de um sistema múltiplas filas?

131) Considere as seguintes tabelas. Elabore o gráfico de Gantt para o algoritmo Múltipla Fila. Calcule os tempos médios de retorno, resposta e espera para cada uma das situações.

- a) Algoritmos: Fila 1 – FCFS; Fila 2 – Prioridade; Fila 3 – RR, com quantum = 3  
Algoritmo entre filas: RR com quantum = 2

PROC	FILA	Tcheg	Tcpu	Prior	Tret	Tresp	Tesp
A	1	0	10	1			
B	2	5	8	1			
C	1	0	7	1			
D	3	9	4	2			
E	2	0	3	0			
F	3	8	9	3			
G	3	1	7	4			
H	2	10	5	3			

- b) Algoritmos: Fila 1 – SJF; Fila 2 – FIFO; Fila 3 – RR, com quantum = 2  
Algoritmo entre filas: prioridade. Fila 1 = 2, Fila 2 = 3 e Fila 3 = 1

PROC	FILA	Tcheg	Tcpu	Prior	Tret	Tresp	Tesp
A	1	5	10				
B	2	4	8				
C	1	8	5				
D	3	7	7				
E	1	1	1				
F	3	0	2				

-X-

# 10

## Gerência de Memória

*“Nunca implore por aquilo que você tem o poder de conquistar.”* (Miguel de Cervantes)

### 10.1 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 com a redução do custo e consequente aumento da capacidade, seu gerenciamento é um dos fatores mais importantes no projeto de Sistemas Operacionais.

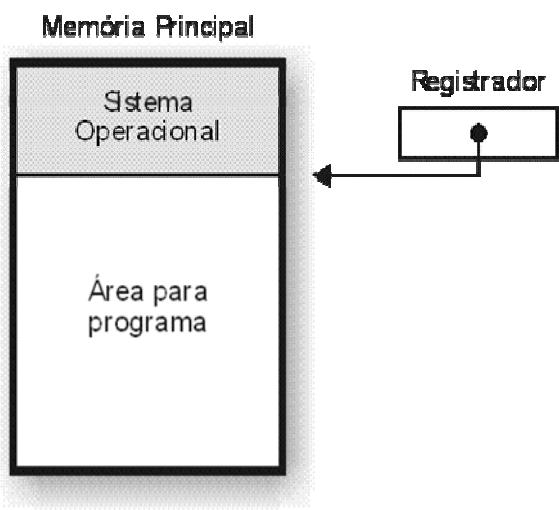
### 10.2 FUNÇÕES BÁSICAS

O objetivo principal de um sistema computacional é executar programas. Tais programas devem estar, ainda que parcialmente, localizados na memória principal para executarem. Contudo, a memória principal, normalmente, não possui tamanho suficiente para armazenar todos os programas a serem executados, bem como os dados por eles utilizados. Este fato levou a maioria dos Sistemas Operacionais modernos a utilizarem principalmente, dos discos rígidos como dispositivos secundários de apoio a memória.

Assim, faz-se necessária a presença do software de gerência de memória que é, também, conhecido como Gerente de Memória e possui como objetivos principais controlar quais partes da memória estão ou não em uso e tratar as dificuldades inerentes ao processo de *swapping* – processo de troca de dados entre a memória principal e o disco rígido.

### 10.3 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. Dessa forma, o programador deve desenvolver suas aplicações preocupado apenas, em não ultrapassar o espaço de memória disponível.



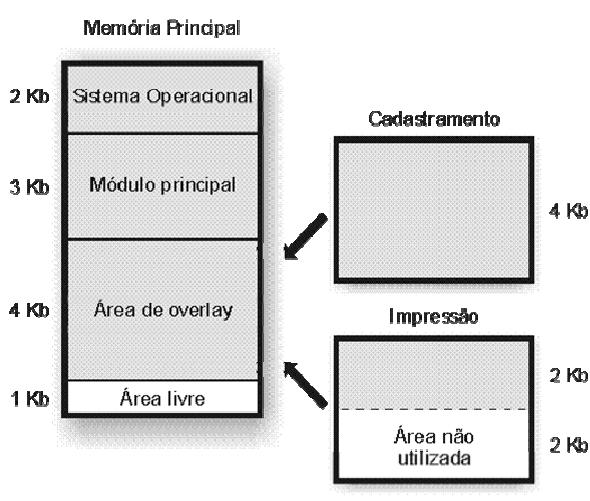
Nesse esquema, o usuário tem controle sobre toda a memória principal, podendo ter acesso a qualquer posição de memória, inclusive a área do Sistema Operacional. Para proteger o sistema desse tipo de acesso, que pode ser intencional ou não, alguns sistemas implementam proteção através de um registrador que delimita as áreas do Sistema Operacional e do usuário. 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.

### 10.4 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 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 de 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 figura, a memória é insuficiente para armazenar todo o programa, que totaliza 9KB. A técnica de *overlay* utiliza uma área de memória comum, onde os módulos de cadastramento e de impressão poderão compartilhar a mesma área de memória. Sempre que um dos dois módulos for referenciado pelo módulo principal, o módulo será carregado da memória secundária para a área de *overlay*.

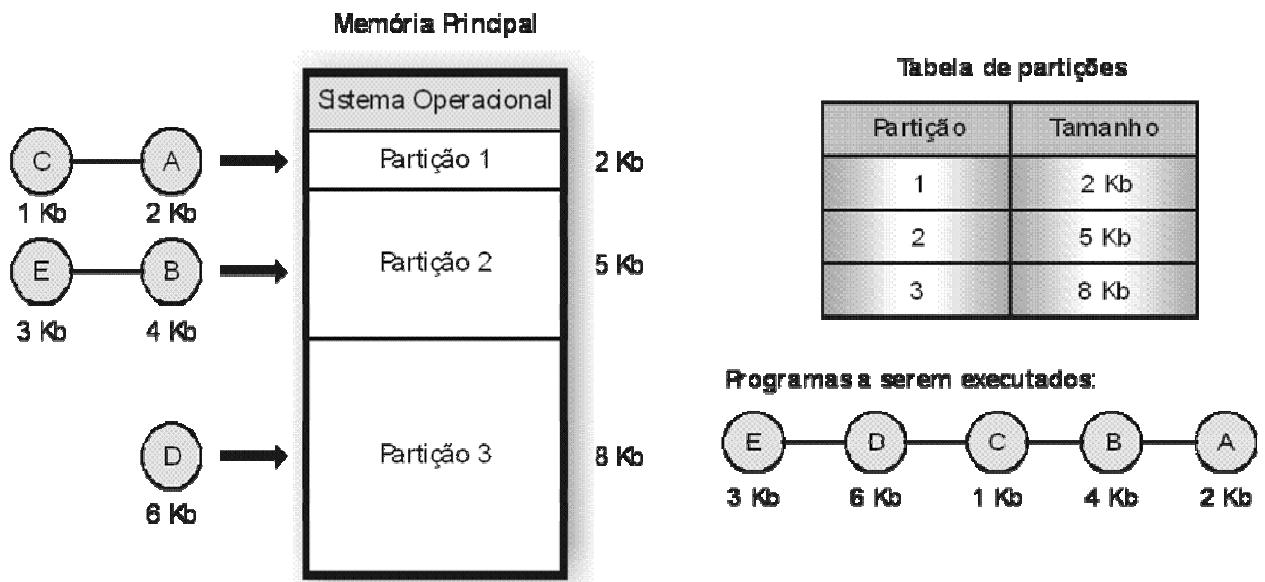
A definição das áreas de *overlay* é função do programador, através de comandos específicos da linguagem de programação utilizada. O tamanho de uma área de *overlay* é estabelecido a partir do tamanho do maior módulo.

## 10.5 ALOCAÇÃO PARTICIONADA

Os sistemas multiprogramáveis 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 de memória sejam implementadas.

### 10.5.1 ALOCAÇÃO PARTICIONADA ESTÁTICA

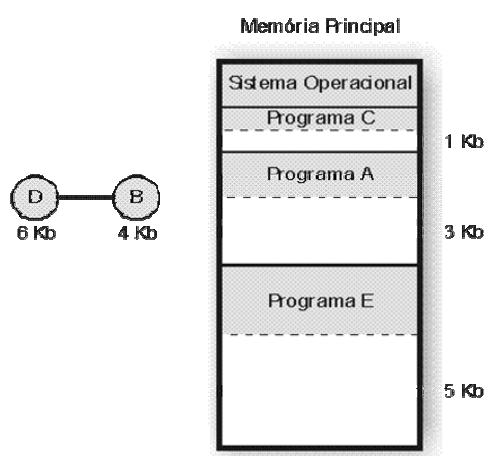
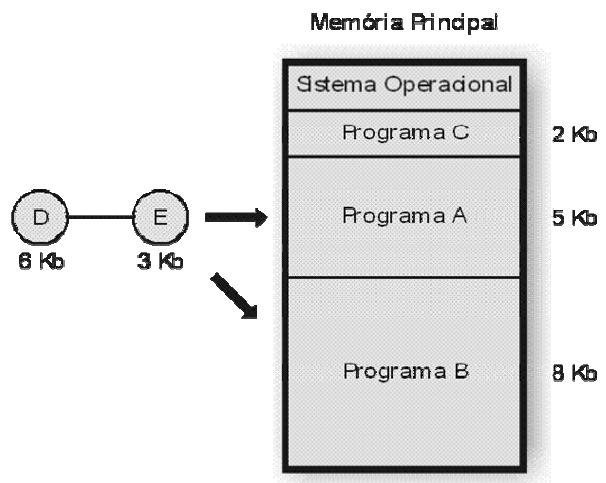
Nos primeiros sistemas multiprogramáveis, a memória era dividida em pedaços de tamanho fixo, chamados **partições**. O tamanho das partições, estabelecido na fase de inicialização do sistema, era definido em função do tamanho dos programas que executariam no ambiente. Sempre que fosse necessária a alteração 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 **alocação fixa**.



Inicialmente, os programas só podiam ser carregados 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. A esse tipo de gerência de memória chamou-se **alocação particionada estática absoluta**.

Com a evolução dos compiladores, o código gerado deixou de ser absoluto e passa 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 partições. Esse tipo de gerência é denominado **alocação particionada estática relocável**.

Para manter 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. 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.

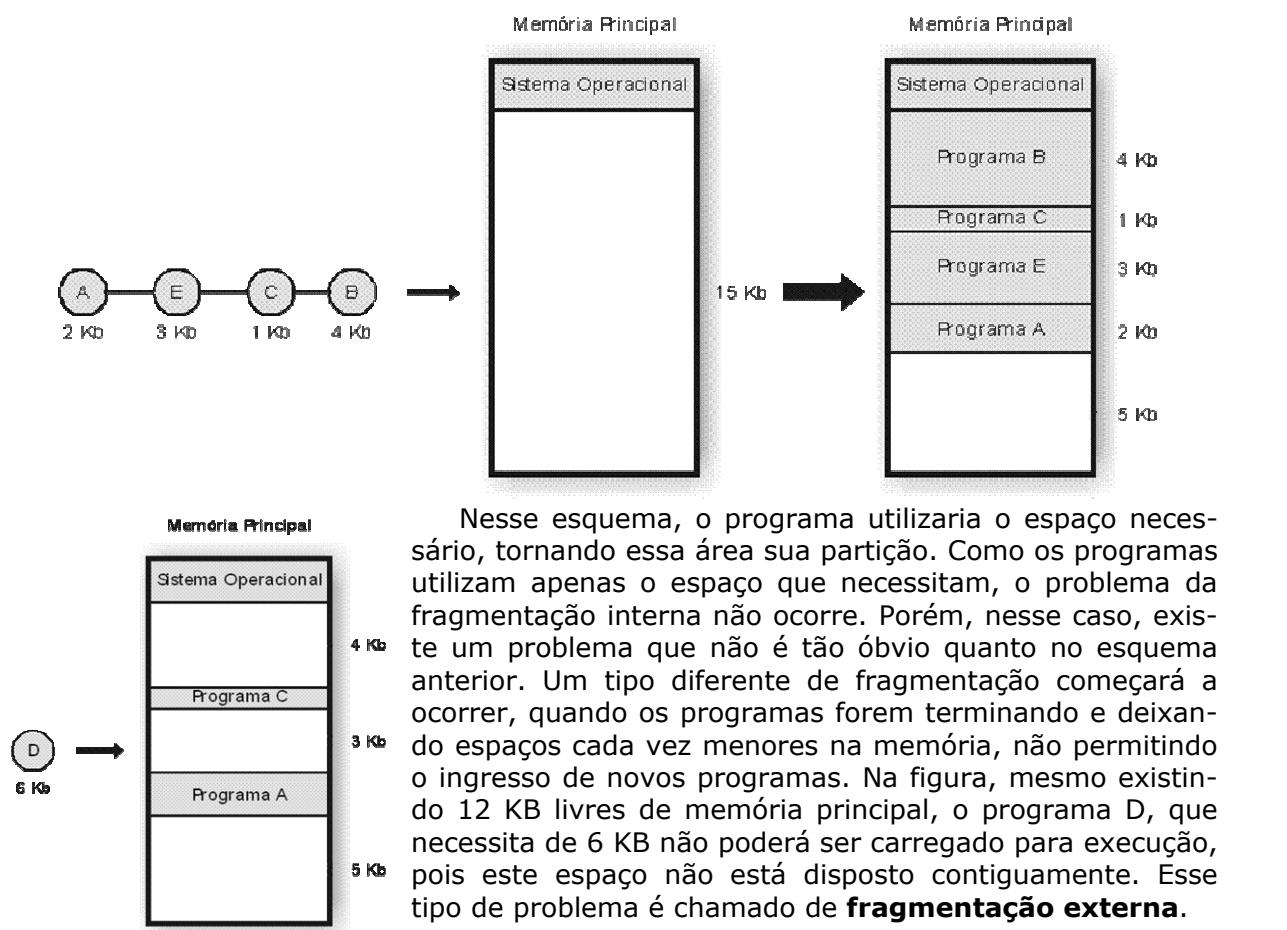


Nesse esquema de alocação de memória, a proteção baseia-se em dois registradores, que indicam os limites inferior e superior da partição onde o programa está sendo executado.

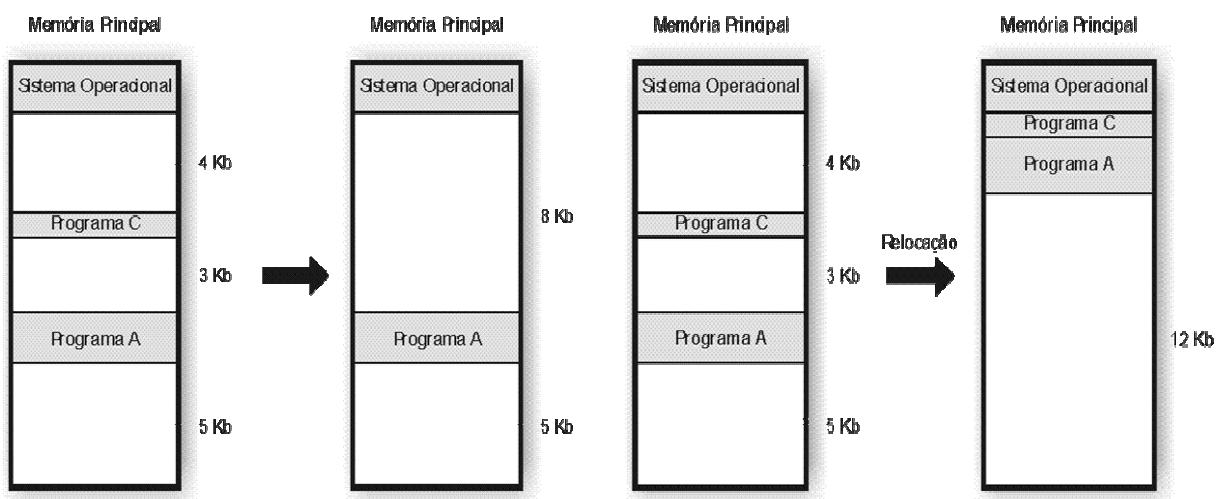
Tanto nos sistemas de alocação absoluta quanto nos de alocação relocável, os programas, normalmente, não preenchem totalmente as partições onde são carregados. Por exemplo, os programas C, A e E não ocupam integralmente o espaço das partições onde estão alocados, deixando 1 KB, 3 KB e 5 KB de áreas livres. Este tipo de problema, decorrente da alocação fixa das partições, é conhecido como **fragmentação interna**.

### 10.5.2 ALOCAÇÃO PARTICIONADA DINÂMICA

Na tentativa de evitar o desperdício de memória ocasionado pelo método tratado na subseção anterior, foi desenvolvido um outro esquema de multiprogramação. Este outro esquema considera que o tamanho das partições de memória não são fixos e é conhecido como **alocação particionada dinâmica** ou **partições variáveis**.



Existem duas soluções para o problema da fragmentação externa. Na primeira solução, conforme os programas terminam, apenas os espaços livres adjacentes são reunidos, produzindo áreas livres de tamanho maior. A segunda solução envolve a relocação de todas as partições ocupadas, eliminando todos os espaços entre elas e criando uma única área livre contígua. Para que esse processo seja possível, é necessário que o sistema tenha a capacidade de mover os diversos programas na memória principal, realizar **relocação dinâmica**. A complexidade do seu algoritmo e o consumo de recursos Sistema Operacional sistema para a relocação dinâmica, podem torná-la inviável.



### 10.5.3 ESTRATÉGIAS DE ALOCAÇÃO DE PARTIÇÃO

Para que diversos programas estejam na memória principal ao mesmo tempo, novas formas de gerência de memória devem ser implementadas. Os Sistemas Operacionais implementam algumas estratégias para determinar em qual área livre um programa será carregado para execução. Essas estratégias visam 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 acessados no ambiente. Independentemente do algoritmo utilizado, o sistema deve possuir formas de saber quais áreas estão livres e quais não estão. Existem duas estratégias principais: Mapa de Bits e Lista Encadeada.

a) **MAPA DE BITS:** esta estratégia divide a memória em pequenas unidades de alocação e a cada uma delas é associado um bit no mapa de bits. Convenciona-se que um "0" no mapa indica que a unidade correspondente da memória está livre e "1" informa que está ocupada. A principal dificuldade em se utilizar o mapa de bits ocorre quando for necessário, por exemplo, trazer para a memória um processo que ocupa  $k$  unidades de alocação. O Gerente de Memória deve, então, procurar por  $k$  bits "0" consecutivos no mapa. Esta procura é lenta, de forma que, na prática, os mapas de bits raramente são usados, ainda que eles sejam muito simples de serem implementados.

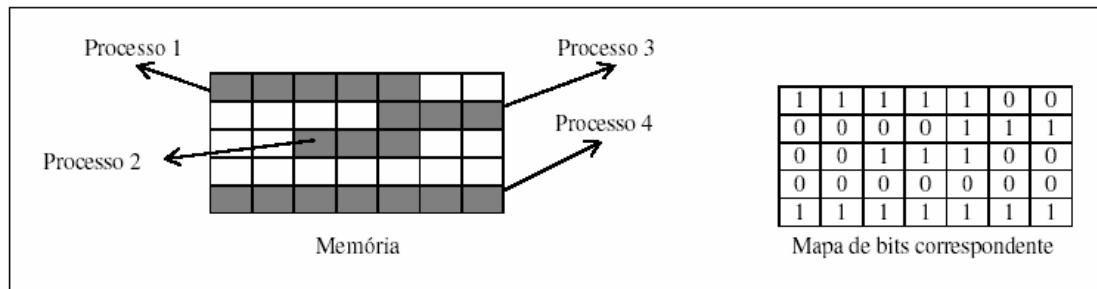
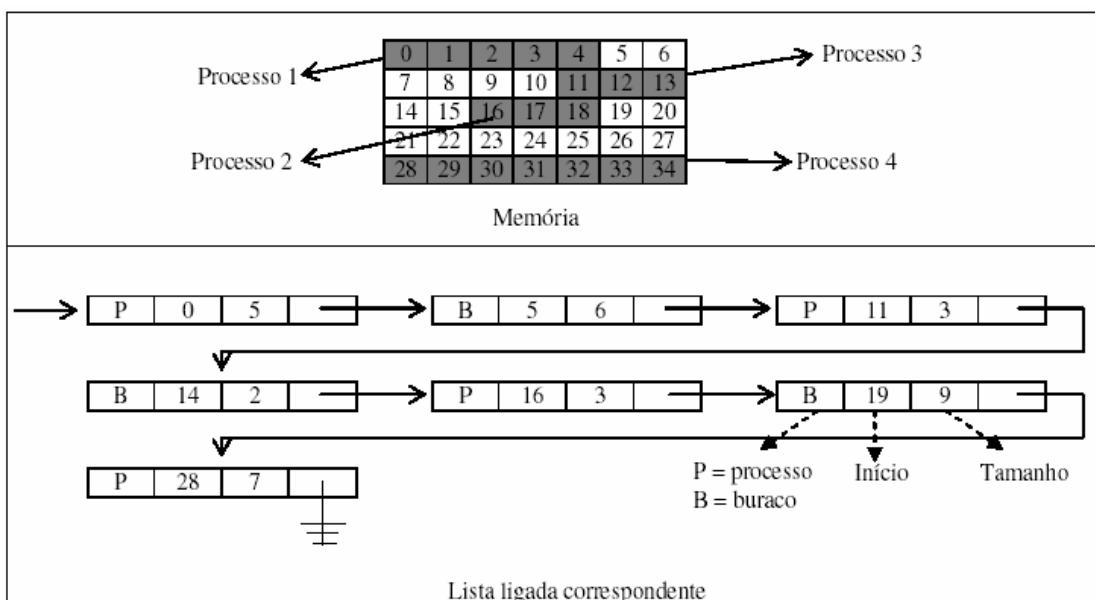


Figura 3.5: Estratégia de alocação/liberação de memória que utiliza mapa de bits.

b) **LISTAS LIGADAS:** a presente estratégia utiliza uma lista ligada para controlar a alocação/liberação de memória, onde tal lista contém os segmentos livres e ocupados da memória. Um segmento pode ser um processo ou um buraco, conforme mostrado na Figura.



No esquema de listas ligadas podem ser utilizados os algoritmos de alocação de memória apresentados na Tabela abaixo:

ALGORITMO	DESCRÍÇÃO
Primeira Alocação	<p>O Gerente de Memória procura ao longo da lista de segmentos até encontrar um buraco que seja suficientemente grande para abrigar o processo. O buraco é então quebrado em 2 pedaços, um para o processo e outro para o espaço não-utilizado, exceto no caso, altamente improvável, do tamanho do buraco corresponder exatamente ao do processo.</p> <p>Vantagem: Este algoritmo é extremamente rápido, pois procura o mínimo de tempo possível.</p>
Próxima Alocação	<p>É uma pequena variação do algoritmo da primeira alocação. Funciona exatamente igual ao algoritmo anterior, exceto pelo fato de guardar a posição onde ele encontra um buraco conveniente. Da próxima vez que o algoritmo for chamado, ele inicia sua busca deste ponto, em vez de começar de novo no início da lista.</p> <p>Desvantagem: Este algoritmo tem uma performance um pouco pior do que o da primeira alocação.</p>
Melhor Alocação	<p>Este algoritmo busca na lista inteira a melhor posição para armazenar o processo que está precisando de espaço. Em vez de parar ao encontrar um buraco grande, que poderá ser necessário mais tarde, ele busca um buraco cujo tamanho seja o mais próximo possível do tamanho do processo.</p> <p>Desvantagem: Este algoritmo é mais lento do que o da primeira alocação, pois precisa pesquisar toda lista cada vez que for chamado. Ele também resulta em maior desperdício de memória que o da primeira alocação, ou mesmo o da próxima alocação, pois ele tende a dividir a memória em buracos muito pequenos, que se tornam difíceis de serem utilizados. O algoritmo da primeira alocação gera, em média, buracos maiores.</p>
Pior Alocação	<p>É aquele que sempre aloca ao processo o maior buraco disponível, de forma que tal buraco, quando dividido, resulta em novo buraco suficientemente grande para abrigar outro processo.</p> <p>Desvantagem: Este algoritmo não apresenta bons resultados práticos.</p>
Alocação Rápida	<p>Este algoritmo mantém listas separadas para alguns dos tamanhos de buracos mais requisitados. Por exemplo, poderá ser criada uma tabela com n entradas, na qual a primeira entrada é um ponteiro para o início de uma lista de buracos de 4K, a segunda para uma lista de buracos de 8K, a terceira para buracos de 12K e assim por diante. Com a Alocação Rápida, a busca de um buraco de determinado tamanho é muito rápida.</p> <p>Desvantagem: sua complexidade é maior, uma vez que devem ser gerenciadas várias listas de buracos.</p>

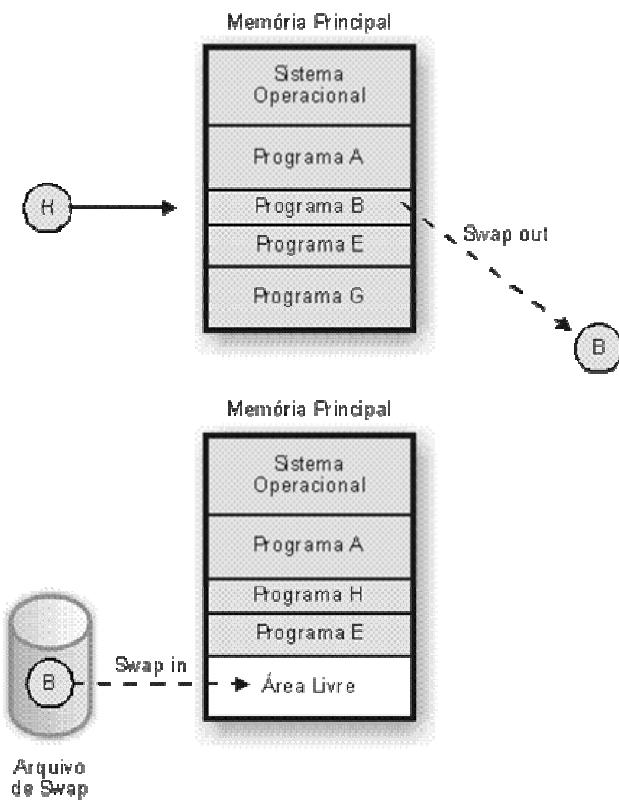
**SISTEMA BUDDY:** o sistema *buddy* é uma estratégia que tira vantagem do fato de os computadores usarem números binários para o endereçamento, como uma forma de acelerar a junção dos buracos adjacentes quando um processo termina ou quando é retirado da memória. A Figura traz um exemplo do funcionamento da referida estratégia.

Apesar de extremamente eficiente sob o aspecto da velocidade, o sistema *buddy* não é eficiente em termos de utilização da memória. O problema decorre, obviamente, da

necessidade de se arredondar a requisição feita pelo processo para a próxima potência inteira de 2. Assim, a um processo de 35K deve ser alocado 54K. Os 29K excedentes serão perdidos, ocasionando a fragmentação interna.

	Memória									
	0	128 K	256 K	384 K	512 K	640 K	768 K	896 K	1 M	Buracos
Inicialmente										1
Requisição de 70	A	128		256			512			3
Requisição de 35	A	B	64		256			512		3
Requisição de 80	A	B	64	C	128			512		3
Devolução de A	128	B	64	C	128			512		4
Requisição de 60	128	B	D	C	128			512		4
Devolução de B	128	64	D	C	128			512		4
Devolução de D		256		C	128			512		3
Devolução de C								1024		1

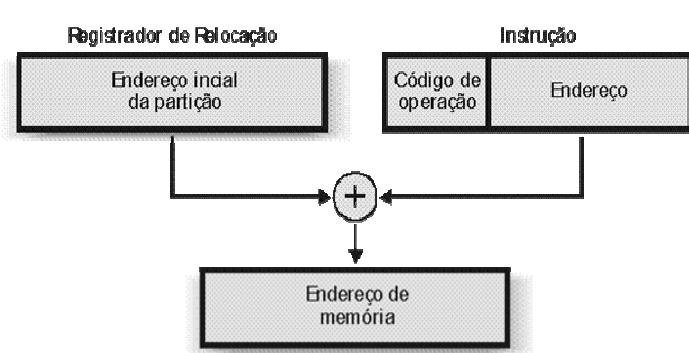
## 10.6 SWAPPING



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 E/S. 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.

O algoritmo de escolha do processo a ser retirado da memória principal deve priorizar aquele com menores chances de ser executado. Para que essa técnica seja utilizada, é essencial que o sistema ofereça um *loader* que implemente a relocação dinâmica.



No momento em que o programa é carregado na memória, o registrador recebe o endereço inicial da posição de memória que o programa irá ocupar. 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. Dessa forma, um programa pode ser carregado em qualquer posição de memória.

## 10.7 EXERCÍCIOS

- 132) Quais as funções básicas da gerência de memória?
- 133) 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?
- 134) Suponha que um sistema computacional de 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?
- 135) Considerando o exercício anterior, se o módulo de 30 KB tivesse seu tamanho aumentado para 40 KB, seria possível executar o programa? Caso não possa, como o problema poderia ser contornado?
- 136) Qual a diferença entre fragmentação interna e externa da memória principal?
- 137) Suponha um sistema computacional com 128 KB de memória principal e que utilize um Sistema Operacional de 64 KB que implemente alocação particionada 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.
- P1 <- PA (6 KB); P2 <- PB (20 KB); P3 <- PC (28 KB)
  - P1 <- PA (4 KB); P2 <- PB (16 KB); P3 <- PC (26 KB)
  - P1 <- PA (8 KB); P2 <- PB (24 KB); P3 <- PC (32 KB)
- 138) Considerando o exercício anterior, seria possível executar quatro programas concorrentemente utilizando apenas a técnica de alocação particionada estática relocável? Se for possível, como? Considerando ainda o mesmo exercício, seria possível executar um programa de 32 KB? Se for possível, como?
- 139) Qual a limitação da alocação particionada absoluta em relação à alocação estática relocável?
- 140) 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 quando existem duas ou mais partições livres adjacentes.

PROCESSOS	MEMÓRIA	TEMPO
1	30 KB	5
2	6 KB	10
3	36 KB	5

5 KB	Programa A
3 KB	Programa B
10 KB	Livre
6 KB	Programa C
26 KB	Livre

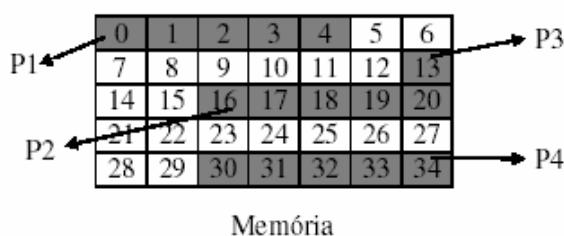
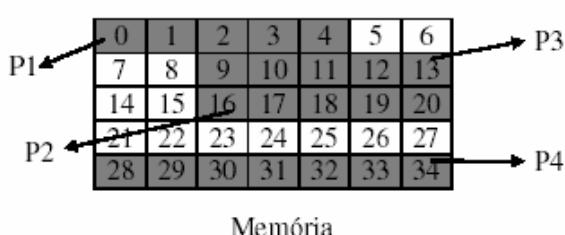
Realize as operações abaixo seqüencialmente, mostrando o estado da memória após cada uma delas. Resolva a questão utilizando as estratégias “Melhor Alocação”, “Pior Alocação”, “Primeira Alocação” e “Próxima Alocação”.

- 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.

- 144) O que é swapping e para que é utilizada essa técnica?

145) Por que é importante o uso de um loader com relocação dinâmica para que a técnica de swapping possa ser implementada?

146) Apresente o mapa de bits e o esquema de listas ligadas para os dois layouts de memória apresentados abaixo:



- 147) Considere um sistema com swapping no qual os seguintes buracos estão na memória, na ordem apresentada: 17K, 4K, 20K, 18K, 7K, 9K, 11K e 15K. Considere, ainda, que foram sucessivamente carregados para a memória os processos A, B e C de tamanhos 18K, 9K e 14K, respectivamente. Assim sendo, aplique os algoritmos da Primeira Alocação e o da Próxima Alocação.
- 148) Um minicomputador usa o sistema buddy para gerenciar sua memória. Inicialmente, ele tem um bloco de 512K no endereço 0. Demonstre graficamente a situação da memória após cada passo e responda ao final de todos os passos: a) Quantos blocos livres restaram, b) quais os seus tamanhos e c) quais seus endereços.
- a) Alocação do Processo A de 12 KB
  - b) Alocação do Processo B de 74 KB
  - c) Alocação do Processo C de 30 KB
  - d) Finalização do Processo B
  - e) Alocação do Processo D de 200 KB
  - f) Finalização do Processo A
  - g) Alocação do Processo E de 7 KB

-X-

# 11

## Memória Virtual

*“Nunca deixe o inimigo perceber que você está sangrando.” (James Bond)*

### 11.1 INTRODUÇÃO

As implementações vistas anteriormente no gerenciamento de memória 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, onde 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 estão residentes. Isto leva a uma utilização mais eficiente também do processador. Além disso, essa técnica possibilita minimizar o problema da fragmentação da memória principal.

### 11.2 ESPAÇO DE ENDEREÇAMENTO VIRTUAL

#### Endereço Físico

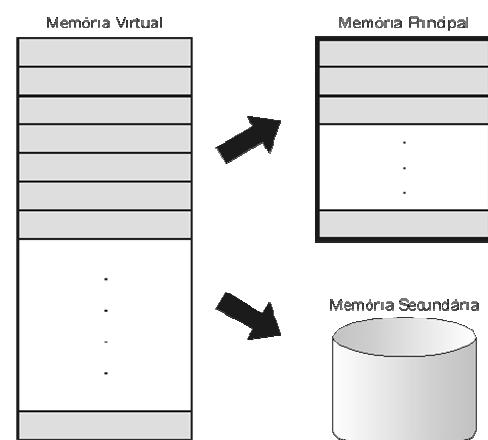
500	VET [1]
501	VET [2]
502	VET [3]
503	VET [4]
504	VET [5]
.	.
.	.
.	.
599	VET [100]

O conceito de memória virtual se aproxima muito da idéia 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 implementem esse mecanismo, tornando-o totalmente transparente ao programador.

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 o endereço físico é chamado de **mapemento**.

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



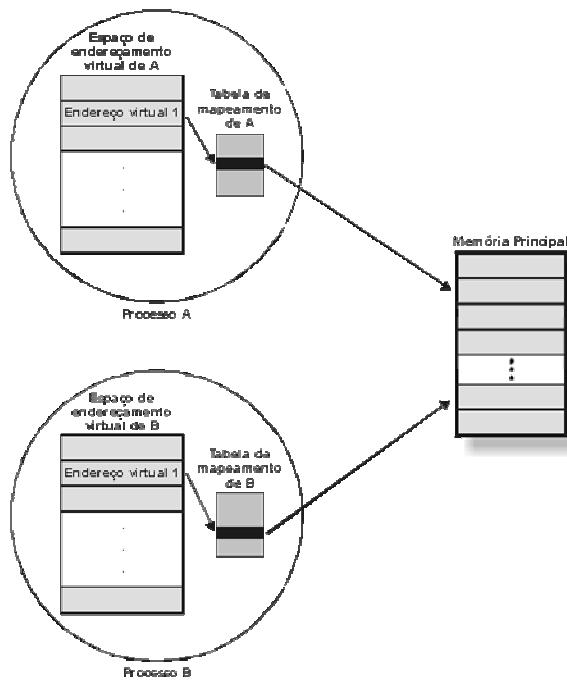
restante na memória secundária até o momento de ser referenciado. Esta condição permite aumentar o compartilhamento da memória principal entre muitos processos.

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.

### 11.3 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 reais. Esse mecanismo é chamado **mapeamento**.

Nos sistemas atuais, o mapeamento é realizado por hardware juntamente com o Sistema Operacional. O dispositivo de hardware responsável por esta tradução é conhecido como **Unidade de Gerenciamento de Memória** (*Memory Management Unit* – MMU), sendo acionado sempre que se faz referência um endereço virtual. Depois de traduzido, o endereço real pode ser utilizado pelo processador para 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.

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 na memória.

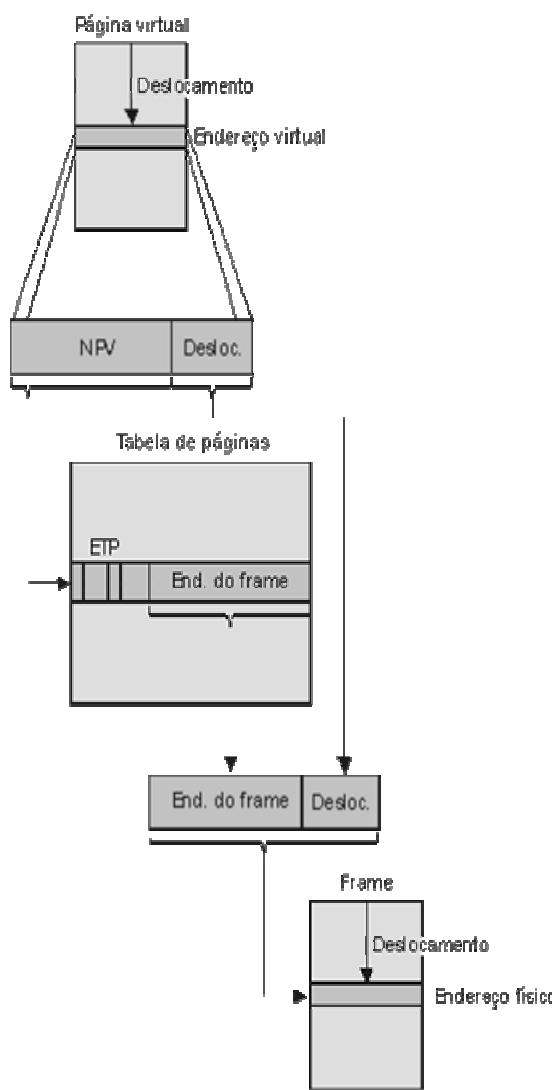
Espaço de Endereçamento Virtual	Tamanho do Bloco	Número de Blocos	Número de entradas na tabela de mapeamento
$2^{32}$ endereços	512 bytes	$2^{23}$	$2^{23}$
$2^{32}$ endereços	4 KB	$2^{20}$	$2^{20}$
$2^{64}$ endereços	4 KB	$2^{52}$	$2^{52}$
$2^{64}$ endereços	64 KB	$2^{48}$	$2^{48}$

Como veremos a seguir, existem Sistemas Operacionais que trabalham apenas com blocos de tamanho fixo (paginação), enquanto outros utilizam blocos de tamanho variável (segmentação). Existe ainda um terceiro tipo de sistema que implementa ambas as técnicas (segmentação paginada).

### 11.4 PAGINAÇÃO

É a técnica de gerência de memória onde o espaço de endereçamento virtual e o real são divididos em blocos do mesmo tamanho chamados **páginas**. A definição do **tamanho da 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 16MB. Páginas

no espaço virtual são denominadas **páginas virtuais**, enquanto as páginas no espaço real são chamadas de **páginas reais, molduras ou frames**.



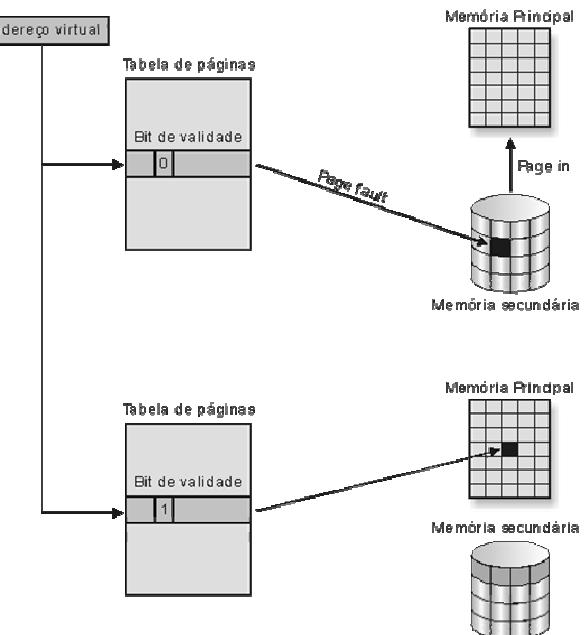
está ou não na memória principal. Caso a página não esteja na memória, dizemos que ocorre uma **falta de página (page fault)**. Neste caso, o sistema transfere a página da memória secundária para a memória principal, realizando uma operação de E/S conhecida como **page in** ou **paginação**. O número de faltas de página gerado por um processo depende de como o programa foi desenvolvido, além da política de gerência de memória implementada pelo Sistema Operacional. O número de falta de páginas geradas por cada processo em um determinado intervalo de tempo é definido como **taxa de paginação**.

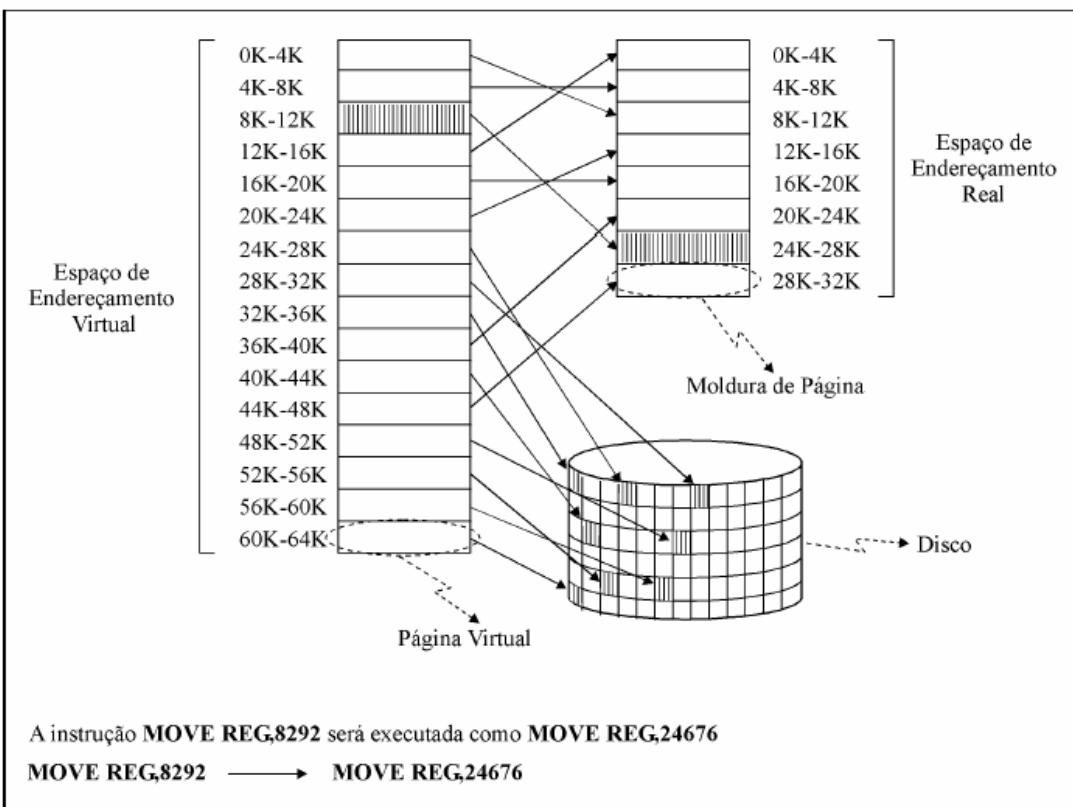
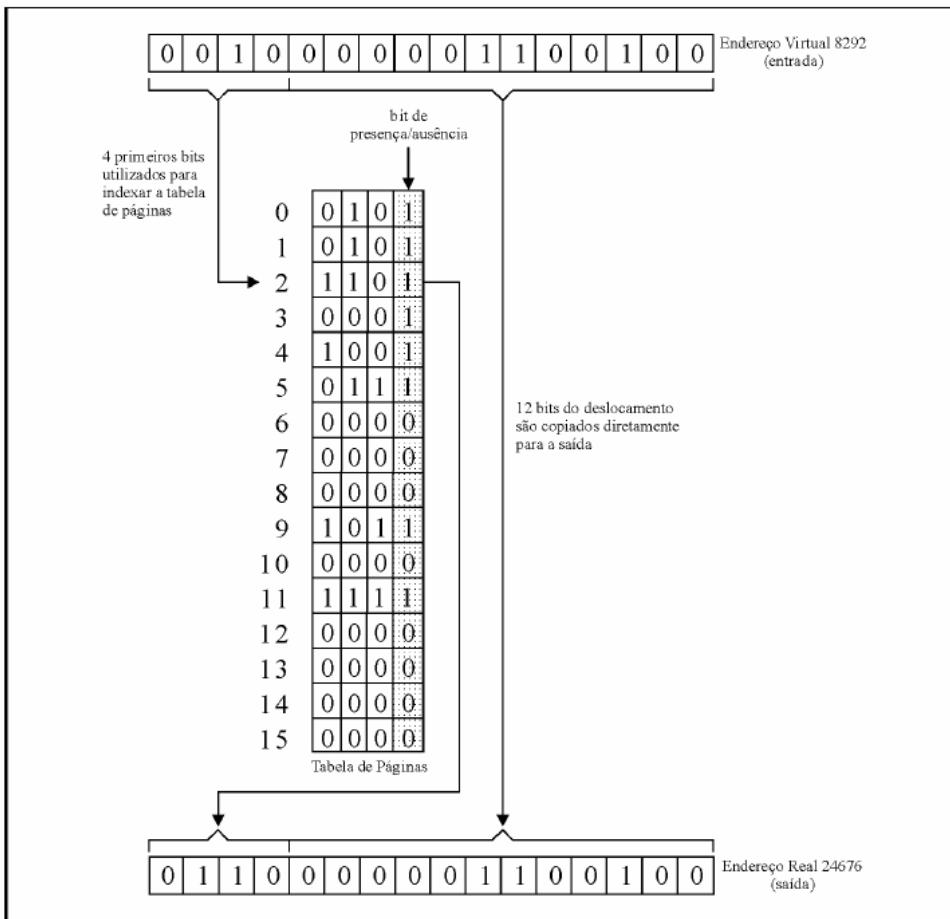
Todo o mapeamento de endereço virtual em real é realizado através de **tabelas de páginas**. Cada processo possui sua própria tabela e cada página virtual do processo possui uma **entrada na tabela de páginas (ETP)**, com informações de mapeamento que permitem ao sistema localizar a página real correspondente.

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.

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.

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

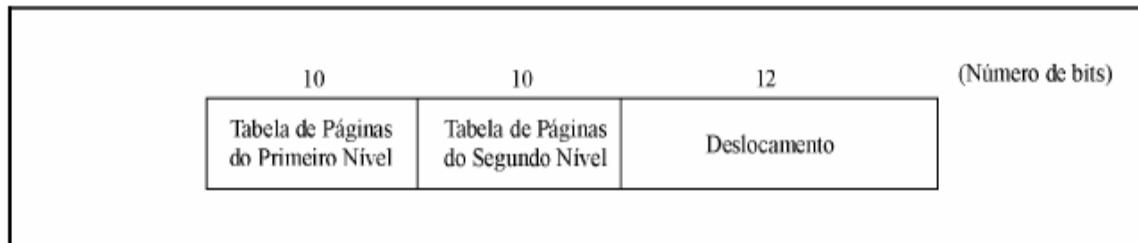




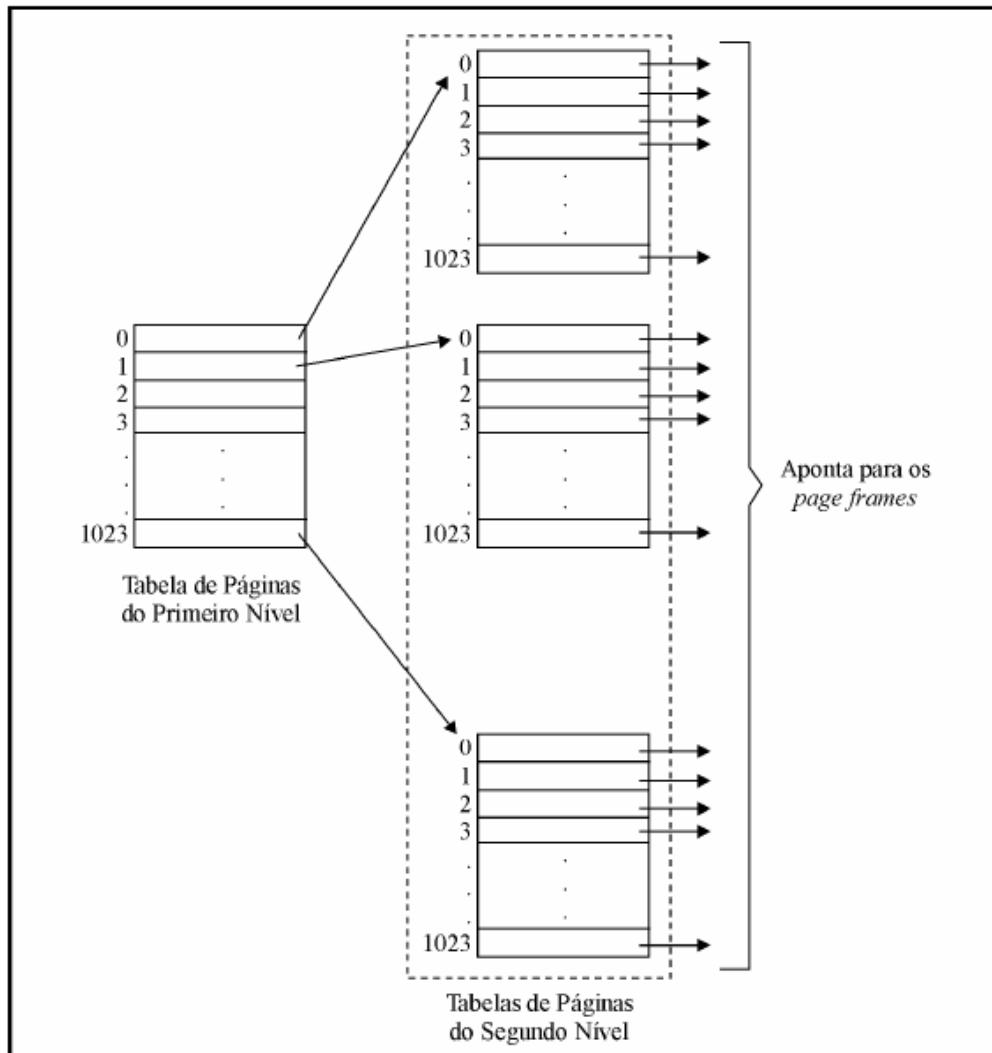
### 11.4.1 PAGINAÇÃO MULTINÍVEL

Em sistemas que implementam apenas um nível de paginação, o tamanho das tabelas de páginas pode ser um problema. Em uma arquitetura de 32 bits para endereçamento e páginas com 4KB 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. Imaginando vários processos residentes na memória principal, manter tabelas desse tamanho para cada processo certamente seria de difícil gerenciamento.

Uma boa solução para contornar o problema é a utilização de **tabelas de páginas multinível**. Com a finalidade de propiciar um melhor entendimento do mencionado conceito, considere-se um sistema computacional com palavra de 32 bits, 4 GB de espaço de endereçamento virtual e páginas de tamanho 4K. Nesta configuração, a palavra que chega à MMU é dividida em três partes, como indica a Figura:

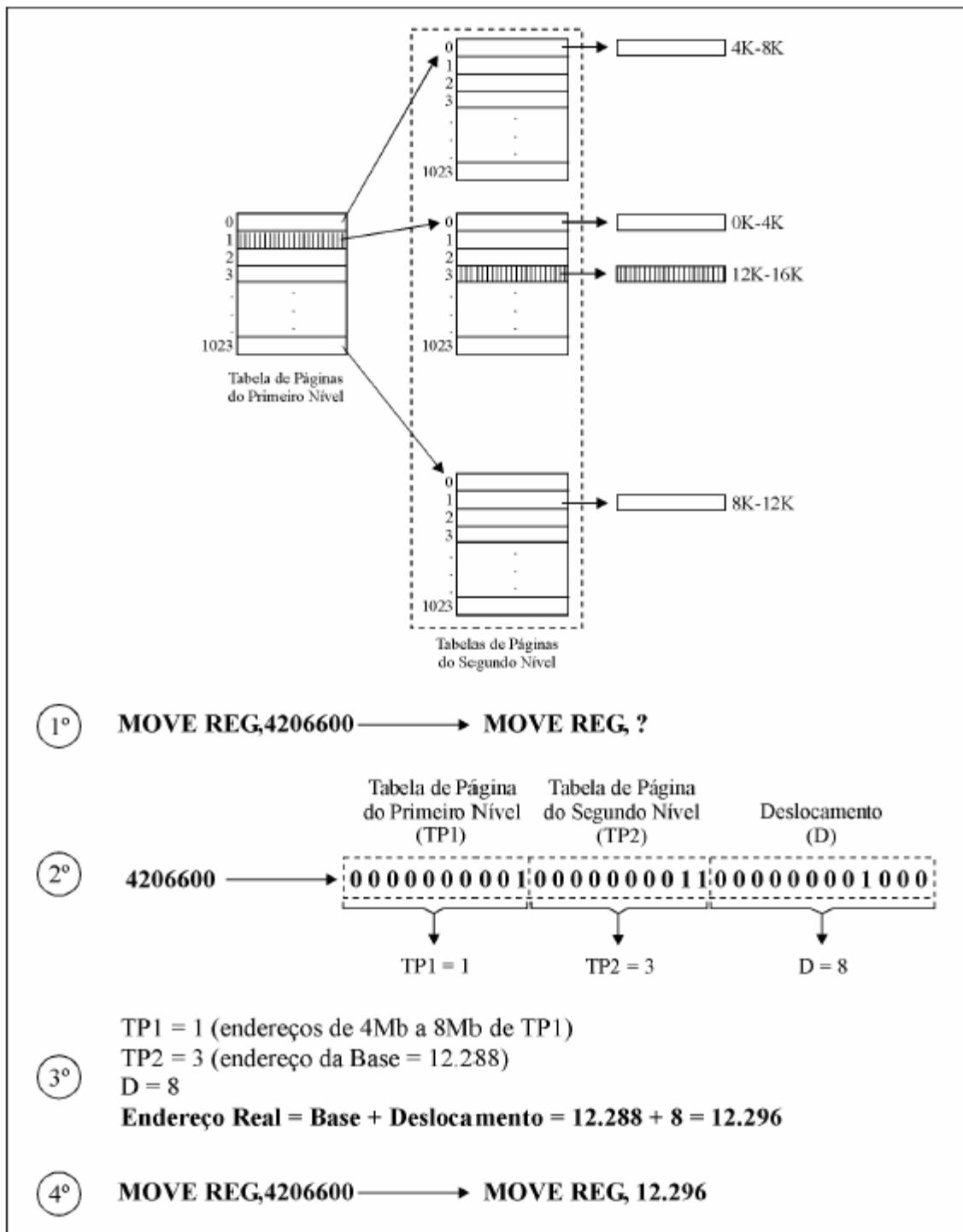


Então, baseando-se nos dados do sistema computacional e no layout da palavra de endereçamento mostrado, tem-se o esquema de tabela de páginas multinível apresentado abaixo:



Dessa forma, cada entrada do primeiro nível gerencia 4 MB de espaço e cada entrada do segundo mapeia 4KB, totalizando 4GB.

A Figura abaixo apresenta um exemplo de funcionamento da MMU para o caso de tabelas de páginas multinível.



Como exemplos práticos de hardware paginação pode-se citar: PDP-11 (paginação de um nível), VAX (paginação em dois níveis), SPARC (paginação em três níveis) e 68030 (paginação em quatro níveis).

#### 11.4.2 POLÍTICAS DE BUSCA DE PÁGINAS

Determina quando uma página deve ser carregada para a memória. Basicamente, existem duas estratégias para este propósito: paginação sob demanda e pré-paginação ou paginação antecipada.

**Na paginação sob demanda**, 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 pré-paginação**, o sistema carrega para a memória principal, além da página referenciada, outras páginas que podem ou não ser necessárias ao processo ao longo do processamento. Se imaginarmos que o programa está armazenado seqüencialmente 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.

#### 11.4.3 POLÍTICAS DE ALOCAÇÃO DE PÁGINAS

Determina quantas molduras (*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 molduras 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 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 página apresenta dois problemas. Se o número máximo de páginas alocadas for muito pequeno, o processo tenderá a ter um elevado número de falta de página, 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.

**Na política de alocação variável**, o número máximo de páginas pode variar durante sua execução em função de sua taxa de paginação e da ocupação da memória principal. Este mecanismo, apesar de ser mais flexível, exige que o Sistema Operacional monitore constantemente o comportamento dos processos, gerando maior *overhead*.

#### 11.4.4 POLÍTICAS DE SUBSTITUIÇÃO DE PÁGINAS

Em algumas situações, quando um processo atinge o seu limite de alocação de molduras 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. A partir dessa situação, qualquer estratégia de substituição de páginas deve considerar se uma página foi ou não modificada antes de liberá-la. Se a página tiver sido modificada, 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**.

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**. 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 deste escopo, pode ser definida como local ou global.

**Na política de substituição local**, apenas as páginas do processo que gerou a falta de página são candidatas a realocação. 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 a substituição, independente do processo que gerou a falta de pági-

na. Na verdade, nem todas as páginas podem ser candidatas a substituição. Algumas páginas, como as do núcleo do sistema, são marcadas como bloqueadas e não podem ser realocadas.

#### 11.4.5 WORKING SET

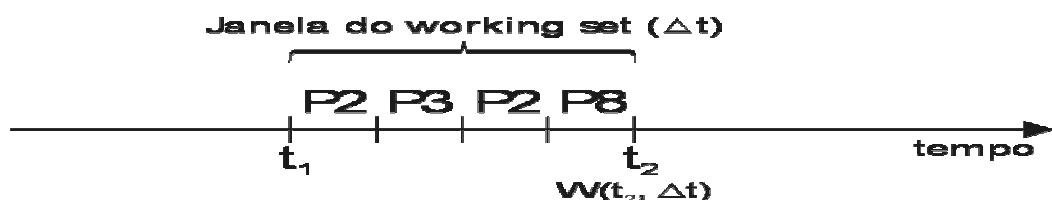
Apesar de suas diversas vantagens, o mecanismo de memória virtual introduz um sério problema: caso os processos tenham na memória principal um número insuficiente de páginas para a execução do programa, é provável que diversos *frames* referenciados ao longo do seu processamento não estejam na memória. Esta situação provoca a ocorrência de um número elevado de falta de página e, consequentemente, inúmeras operações de E/S. Neste caso, ocorre um problema conhecido como ***trashing***, provocando sérias consequências ao desempenho do sistema.

O conceito de *working set* surgiu com o objetivo de reduzir o problema do *trashing* 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.

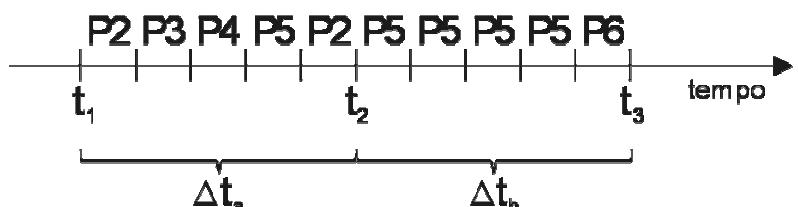
Página 0	Inicialização
Página 1	WHILE () DO BEGIN
Página 2	(Círculo com seta apontando para baixo)
Página 3	BND;
Página 4	Imprime resultados

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.

A partir da observação do princípio da localidade, Peter Denning (1968), formulou o **modelo de working set**. *Working set* é definido como sendo o conjunto das páginas referenciadas por um processo durante determinado intervalo de tempo. A figura ilustra que no instante  $t_2$ , o *working set* do processo  $W(t_2, \Delta t)$ , são as páginas referenciadas no intervalo  $\Delta t$  ( $t_2 - t_1$ ), isto é, as páginas P2, P3 e P8. O intervalo de tempo  $\Delta t$  é denominado **janela do working set**. Podemos observar, então, que o *working set* de um processo é função do tempo e do tamanho da janela do *working set*.



Dentro da janela do *working set*, o número de páginas distintas referenciadas é conhecido como **tamanho do working set**. Na figura são apresentadas as referências às páginas de um processo nas janelas  $\Delta t_a$  ( $t_2 - t_1$ ) e  $\Delta t_b$  ( $t_3 - t_2$ ). O *working set* do processo no instante  $t_2$ , com a janela  $\Delta t_a$ , 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  $\Delta t_b$ , o *working set* corresponde às páginas P5 e P6, e o tamanho é igual a duas páginas.



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 O-

peracional deverá manter as páginas do *working set* de cada processo residente 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 *trashing*. Na prática, o modelo de *working set* serve como base para inúmeros algoritmos de substituição de páginas, como os apresentados a seguir.

#### 11.4.6 ALGORITMOS DE SUBSTITUIÇÃO DE PÁGINAS

A melhor estratégia de substituição de páginas seria aquela que escolhesse uma moldura que não fosse mais utilizada no futuro ou levasse mais tempo para ser novamente referenciada. Porém, quanto mais sofisticado o algoritmo, maior *overhead* para o Sistema Operacional implementá-lo. O algoritmo deve tentar manter o *working set* dos processos na memória principal e, ao mesmo tempo, não comprometer o desempenho do sistema.

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

O algoritmo ótimo simplesmente diz que a página com o maior rótulo deve ser removida, adiando-se o máximo possível a próxima falta de página. (A exemplo das pessoas, os computadores também tendem a adiar o quanto possível a ocorrência de eventos desagradáveis).

O único problema com este algoritmo é que ele não é realizável. No momento da falta de página, o sistema operacional não tem como saber quando cada uma das páginas será referenciada de novo. No máximo podemos executar um programa em um simulador e, mantendo uma lista de todas as páginas referenciadas, implementar o algoritmo na segunda execução (usando informações coletadas na primeira execução).

b) **FIFO:** Para ilustrar seu funcionamento, considere um supermercado que tem prateleiras suficientes para armazenar exatamente  $k$  produtos diferentes. Certo dia, alguma indústria introduz um novo tipo de alimento que faz um tremendo sucesso comercial. Nosso supermercado deve, então, arranjar um jeitinho para vendê-lo, eliminando de suas prateleiras algum outro produto.

Uma possibilidade é descobrir qual dos produtos este supermercado vem estocando há mais tempo (isto é, algo que ele vem vendendo há 120 anos) e livrar-se dele. De fato, tal decisão é tomada com facilidade, visto que o supermercado mantém uma lista de todos os produtos vendidos atualmente, na ordem em que eles entraram pela primeira vez no estoque. O mais novo está no fim da fila, e o mais velho no início, devendo ser eliminado.

Em algoritmos de substituição de páginas, a mesma idéia pode ser aplicada. O sistema operacional mantém uma fila de todas as páginas que estão na memória, com a página no topo da fila sendo a mais antiga e a do fim da fila a que chegou há menos tempo. Na ocorrência de uma falta de página, a página do início deve ser removida, sendo a nova página adicionada ao fim desta fila. Quando aplicado ao problema do supermercado, o algoritmo FIFO tanto pode remover um dos itens mais vendidos, como sal ou manteiga, quanto um dos menos vendidos, como sacos de lixo. Quando aplicada aos computadores, o mesmo problema ocorre. Por isso, o algoritmo FIFO, em sua forma pura, nunca é usado. A Figura traz uma simulação simplificada deste algoritmo;

$page$ $frames \rightarrow$	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>5</td><td>0</td><td>1</td><td>4</td><td>4</td><td>2</td><td>3</td><td>0</td><td>2</td><td>3</td><td>1</td><td>2</td></tr> <tr><td>X</td><td>5</td><td>0</td><td>1</td><td>4</td><td>4</td><td>2</td><td>3</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>X</td><td>X</td><td>5</td><td>0</td><td>1</td><td>1</td><td>4</td><td>2</td><td>3</td><td>3</td><td>3</td><td>0</td></tr> <tr><td>X</td><td>X</td><td>X</td><td>5</td><td>0</td><td>0</td><td>1</td><td>4</td><td>2</td><td>2</td><td>2</td><td>0</td></tr> <tr><td>P</td><td>P</td><td>P</td><td>P</td><td>P</td><td>P</td><td>P</td><td>P</td><td>P</td><td>P</td><td>P</td><td>P</td></tr> </table>	5	0	1	4	4	2	3	0	2	3	1	2	X	5	0	1	4	4	2	3	0	0	0	1	X	X	5	0	1	1	4	2	3	3	3	0	X	X	X	5	0	0	1	4	2	2	2	0	P	P	P	P	P	P	P	P	P	P	P	P	← Referências ← Faltas de página
5	0	1	4	4	2	3	0	2	3	1	2																																																			
X	5	0	1	4	4	2	3	0	0	0	1																																																			
X	X	5	0	1	1	4	2	3	3	3	0																																																			
X	X	X	5	0	0	1	4	2	2	2	0																																																			
P	P	P	P	P	P	P	P	P	P	P	P																																																			
		Total de faltas de página: 9																																																												

c) **Segunda Chance:**

Este algoritmo é uma variação do FIFO, a única diferença é que existe um bit "R" associado a cada página. Se "R" for 0 a página é considerada velha e não referenciada, de modo que ela deve ser removida da memória. Ao passo que, se "R" for 1, "R" deve ser zerado e a página colocada no fim da fila (torna-se jovem novamente). Contudo, se todas as páginas tiverem sido recentemente referenciadas, este algoritmo irá se comportar exatamente como o FIFO;

d) **Relógio:**

O algoritmo da segunda chance está sempre movendo páginas do início para o final da lista. Então, com a finalidade de solucionar este problema, desenvolveu-se o algoritmo do relógio, que possui uma lista ligada circular e um ponteiro que aponta para a página mais velha. Quando uma falta de página acontece, a página que está sendo apontada é testada e, caso o seu bit "R" seja zero, ela deve abandonar a memória, porém se "R" for 1, "R" deve ser zerado e o ponteiro avança para o próximo nó da lista. Este processo deve se repetir até que seja encontrado um nó com "R" igual a zero;

e) **NUR (Not Recently Used):**

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

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

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

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

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

O algoritmo NRU remove uma página aleatória da classe de numeração mais baixa não vazia. Implícito neste algoritmo é que é melhor remover uma página modificada que não foi referenciada pelo menos no último *clock*, que uma página não modificada, mas muito usada.

As características principais do NRU é que ele é fácil de entender, eficiente de se implementar, e gera um desempenho que, embora não ótimo, é geralmente tido como adequado.

#### f) LRU (*Least Recently Used*):

Uma boa aproximação para o algoritmo ótimo é baseada em uma observação comum que as páginas muito usadas nas últimas instruções, provavelmente o serão nas próximas instruções. Da mesma forma, páginas que não têm sido usadas por um longo tempo provavelmente continuarão sem uso. Esta observação sugere um algoritmo realizável. Na ocorrência de uma falta de página, este algoritmo irá remover as páginas menos referenciadas nas últimas instruções, pois ele parte do princípio que as páginas que foram referenciadas nas últimas instruções continuarão sendo acessadas.

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

Manipular uma lista ligada a toda instrução é proibitivo, até mesmo em hardware. Entretanto, há outras maneiras de implementar LRU com um hardware especial. Vamos considerar o caminho mais simples primeiro. Este método requer equipar o hardware com um contador de 64 bits, C, que é automaticamente incrementado após cada instrução. Além disso, cada entrada na tabela de páginas deve também ter um campo grande o bastante para conter o contador. Após cada referência de memória, o corrente valor de C é armazenado na entrada da tabela de páginas para a página referenciada. Quando ocorre uma falta de página, o sistema operacional examina todos os contadores na tabela de páginas para achar o menor deles. A página correspondente é a menos recentemente usada.

Agora vejamos um segundo algoritmo LRU, também em hardware. Para uma máquina com N *page frames*, o LRU deve manter uma matriz de N x N bits, inicialmente todos zero.

Sempre que uma moldura k for referenciada, o hardware coloca todos os bits da linha k em 1, e depois zera todos os bits da coluna k. Em cada instante, a linha com o menor valor binário armazenado será correspondente à página usada há mais tempo; aquela com o próximo valor será a próxima usada há mais tempo, e assim por diante.

Um exemplo do funcionamento deste algoritmo aparece ilustrada na figura para quatro molduras de página e a seguinte ordem de referências às páginas: 5 0 1 2 3 2 1 0 3 2 3. Neste exemplo a página usada há mais tempo é a 1

page frames	→	← Referências											
		5	0	1	4	4	2	3	0	2	3	1	2
X	5	0	1	4	4	2	3	0	2	3	1	2	
X	X	5	0	1	1	4	2	3	0	2	3	1	
X	X	X	5	0	0	1	4	2	3	0	2	3	
	P	P	P	P	P	P	P	P	P	P	P	P	← Faltas de página
Total de faltas de página: 8													

Considerando-se o funcionamento dos algoritmos de substituição de páginas, é possível pensar, de início, que quanto maior for o número de *page frames*, menor será a ocorrência de faltas de páginas durante o período de execução de um processo. Entretanto, estudos demonstraram que este pensamento nem sempre é verdadeiro e este fato ficou conhecido como **anomalia de Belady**.

Um exemplo de ocorrência da mencionada anomalia encontra-se detalhado na Figura 3.16, onde é utilizado o algoritmo de substituição de páginas FIFO que, inicialmente, é simulado com 3 *page frames* e apresenta 9 faltas de páginas. Em seguida, é realizada a

simulação do FIFO com 4 molduras de páginas e é observado que o número de falta de páginas se eleva para 10.

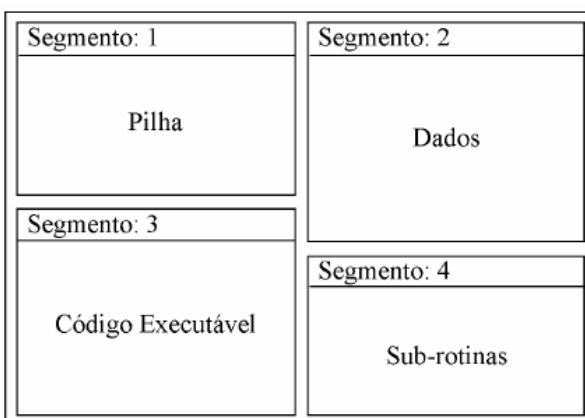
Dessa forma, é possível verificar a presença da anomalia de *Belady*, pois a memória física foi aumentada e o número de falta de páginas também.

<i>page frames</i>	$\rightarrow \left\{ \begin{array}{ccccccccccccc} 0 & 1 & 2 & 3 & 0 & 1 & 4 & 0 & 1 & 2 & 3 & 4 \\ \hline X & 0 & 1 & 2 & 3 & 0 & 1 & 4 & 4 & 4 & 2 & 3 & 3 \\ X & X & 0 & 1 & 2 & 3 & 0 & 1 & 1 & 1 & 4 & 2 & 2 \\ X & X & X & 0 & 1 & 2 & 3 & 0 & 0 & 0 & 1 & 4 & 4 \end{array} \right\}$	$\leftarrow$ Referências										
		$\leftarrow$ Faltas de página										
Total de faltas de página: 9												
<i>page frames</i>	$\rightarrow \left\{ \begin{array}{ccccccccccccc} 0 & 1 & 2 & 3 & 0 & 1 & 4 & 0 & 1 & 2 & 3 & 4 \\ \hline X & 0 & 1 & 2 & 3 & 3 & 3 & 4 & 0 & 1 & 2 & 3 & 4 \\ X & X & 0 & 1 & 2 & 2 & 2 & 3 & 4 & 0 & 1 & 2 & 3 \\ X & X & X & 0 & 1 & 1 & 1 & 2 & 3 & 4 & 0 & 1 & 2 \\ X & X & X & X & 0 & 0 & 0 & 1 & 2 & 3 & 4 & 0 & 1 \end{array} \right\}$	$\leftarrow$ Referências										
		$\leftarrow$ Faltas de página										
Total de faltas de página: 10												

Após a verificação da anomalia de *Belady*, muitos estudos foram desenvolvidos e foi observado que alguns algoritmos não apresentavam tal anomalia e estes foram chamados de algoritmos de pilha.

Obs.: 1) O LRU é um algoritmo de pilha e não apresenta a anomalia de *Belady*; 2) O FIFO, como foi visto anteriormente, apresenta a anomalia de *Belady*.

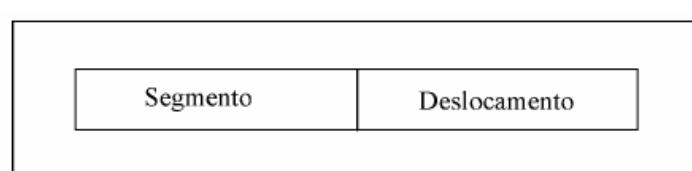
## 11.5 SEGMENTAÇÃO



É a técnica de gerência de memória onde o espaço de endereçamento virtual é dividido em blocos de tamanhos diferentes chamados **segmentos**. Cada segmento tem um número e um tamanho, conforme pode ser observado na Figura. Nesta técnica um programa é dividido logicamente em sub-rotinas e estruturas de dados, que são alocados em segmentos na memória principal.

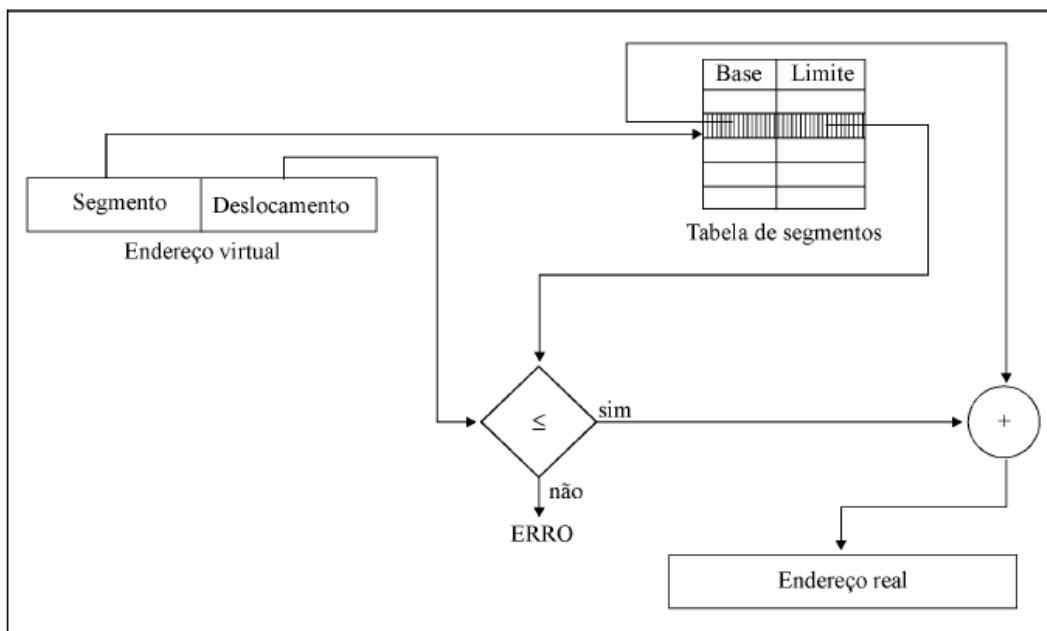
Enquanto na técnica de paginação o programa é dividido em páginas de tamanho fixo, sem qualquer ligação com sua estrutura, na segmentação existe uma relação entre a lógica do programa e sua alocação na memória principal. Normalmente, a definição dos segmentos é realizada pelo compilador, a partir do código fonte do programa, e cada segmento pode representar um procedimento, uma função, vetor ou pilha.

Na segmentação, os endereços especificam o número do segmento e o deslocamento dentro do mesmo.



Assim, para mapear um endereço virtual composto pelo par <segmento, deslocamento> o hardware de segmentação considera a existência de uma tabela de segmentos. Cada entrada da tabela de segmentos possui a base e o limite de cada segmento. A base contém o endereço físico de início do segmento e o limite especifica o seu tamanho.

A figura apresentada a seguir ilustra o funcionamento do mapeamento de um endereço virtual em um sistema que utiliza a segmentação.



Os segmentos podem se tornar muito grandes e, às vezes, pode ser impossível manter todos na memória ao mesmo tempo. Para resolver este problema implementa-se a paginação em cada um dos segmentos, dando origem, então, à segmentação paginada.

A seguir, um quadro comparando as técnicas de paginação e segmentação em função de suas principais características:

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

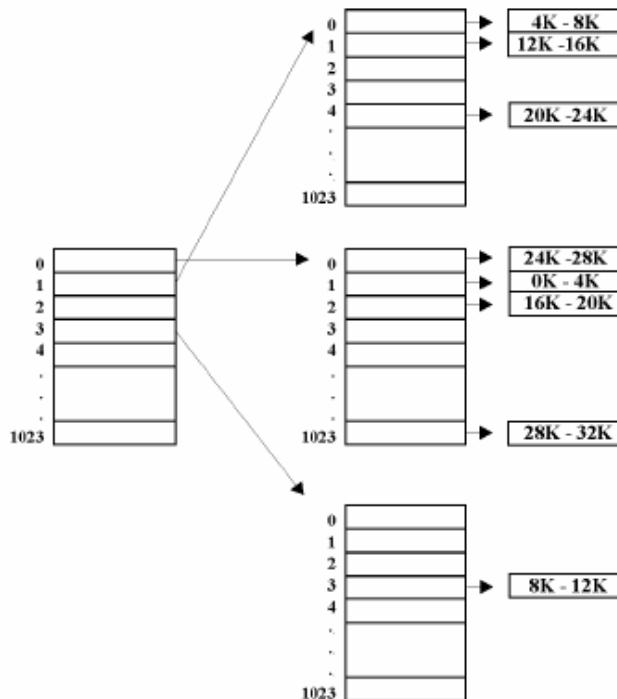
## 11.6 SEGMENTAÇÃO PAGINADA

É a técnica de gerência de memória onde o espaço de endereçamento é dividido em segmentos e, por sua vez, cada segmento é dividido em páginas. Esse esquema de gerência de memória tem o objetivo de oferecer as vantagens de ambas as técnicas.

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 do mesmo tamanho, mapeadas por uma tabela de páginas associada ao segmento. Desta forma, um segmento não precisa estar contíguo na memória principal, eliminando o problema da fragmentação externa encontrado na segmentação pura.

## 11.7 EXERCÍCIOS

- 149) 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?
- 150) Explique como um endereço virtual de um processo é traduzido para um endereço real na memória principal?
- 151) Por que o mapeamento deve ser feito em blocos e não sobre células individuais?
- 152) Qual a principal diferença entre os sistemas que implementam paginação e os que implementam segmentação?
- 153) Diferencie página virtual de página real.
- 154) Para que serve o bit de validade nas tabelas de página?
- 155) Apresente o funcionamento binário da MMU de uma máquina hipotética para encontrar a instrução correspondente de: MOVE REG, 700

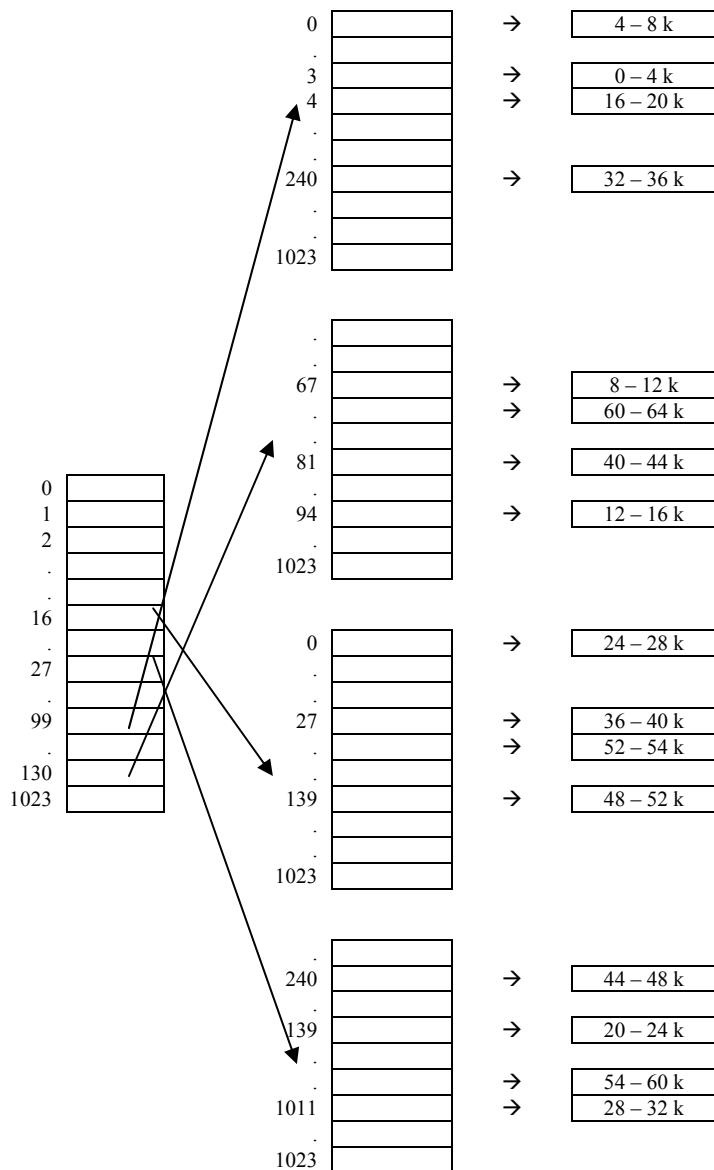


- 156) O que é page fault , quando ocorre e quem controla a sua ocorrência? Como uma elevada taxa de falta de página pode comprometer o Sistema Operacional?
- 157) Descreva como ocorre a fragmentação interna em sistemas que implementam a paginação.
- 158) Compare as políticas de busca de páginas apresentadas.
- 159) Quais as vantagens e desvantagens da política de alocação de páginas variável comparada à alocação fixa?
- 160) 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 tabela abaixo:

Endereço Fixo	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 desse 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?
- 161) Encontre o endereço físico correspondente aos seguintes endereços virtuais:  
 a) 67.219.712      b) 113.819.552      c) 545.591.437      d) 416.219.160



162) Um Sistema Operacional implementa gerência de memória virtual por paginação, com frames de 2KB. A partir da tabela abaixo, 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
6	Sim	1000

- 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?

163) 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 abaixo 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 nesse 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

164) Uma memória virtual possui páginas de 1024 endereços, existem 8 páginas virtuais e 4096 bytes de memória real. A tabela de páginas de um processo está descrita abaixo. O asterisco indica que a página não está na memória principal:

Página Virtual	Página Real
0	3
1	1
2	*
3	*
4	2
5	*
6	0
7	*

- a) Faça a lista/faixa de todos os endereços virtuais que irão causar falta de página.
- b) Indique o endereço real correspondente aos seguintes endereços virtuais 0, 1023, 1024, 6500 e 3728.

165) Porque existe a necessidade de uma política de substituição de páginas? Compare as políticas de substituição local e global.

166) Para que serve o bit de modificação nas tabelas de páginas?

167) Como o princípio da localidade viabiliza a implementação da gerência de memória virtual por paginação?

168) Por que programas não estruturados estão sujeitos a uma alta taxa de paginação?

169) Cite os principais algoritmos de substituição de páginas estudados.

- 170) Explique o funcionamento do algoritmo de substituição de páginas NUR.
- 171) Explique o funcionamento do algoritmo de substituição de páginas da segunda chance.
- 172) Informe a principal desvantagem de se utilizar o algoritmo de substituição de páginas da segunda chance.
- 173) Informe a principal vantagem que o algoritmo do relógio possui sobre o da segunda chance.
- 174) Considere uma máquina com três molduras de página (page frames), as quais estão inicialmente vazias. Assim sendo, informe quantas faltas de páginas serão geradas com a utilização do algoritmo de substituição de páginas FIFO se a seqüência de referência às páginas for: 0, 1, 2, 3, 0, 1, 4, 1, 2, 3, 1 e 4.
- 175) Considere uma máquina com quatro molduras de página (page frames), as quais estão inicialmente vazias. Assim sendo, informe quantas faltas de páginas serão geradas com a utilização do algoritmo de substituição de páginas LRU se a seqüência de referência às páginas for: 0, 1, 2, 3, 0, 1, 4, 1, 2, 3, 1, 4 e 5.
- 176) Considere uma máquina com quatro molduras de página (page frames), as quais estão inicialmente vazias, e uma seqüência de referência às páginas igual a: 0, 1, 2, 3, 0, 1, 4, 1, 2, 3, 1, 4, 5, 4 e 1. Assim sendo, informe, justificando sua resposta, quantas faltas de páginas serão geradas com a utilização de cada um dos algoritmos de substituição de páginas abaixo relacionados:
- a) FIFO
  - b) LRU
  - c) Segunda Chance
- 177) Qual é a principal desvantagem de se utilizar no projeto de um sistema paginado um tamanho de página considerado muito grande? E muito pequeno?
- 178) O que é a anomalia de Belady?

- 179) 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, 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)<sub>16</sub>?
- e) Caso ocorra uma falta de página e uma das páginas do processo deva ser descartada, quais páginas poderiam sofrer *page out*?

- 180) Considere um sistema de memória virtual que implemente paginação, onde o limite de frames por processo é igual a três. Descreva para os itens abaixo, onde é apresentada uma seqüência de referências a páginas pelo processo, o número total de faltas de páginas 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

181) 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 8 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 seqüência de acessos, a tabela de páginas completa desse programa tem a configuração abaixo. 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 seqüência de acessos utilizando a estratégia de remoção FIFO. Mostre o estado final da tabela de páginas.

Página Referenciada	Página Removida	Falta de Página (Sim / Não)
5		
15		
12		
8		
0		

182) Em um computador, o endereço virtual é de 16 bits e as páginas têm tamanho de 2K. 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.

- a) Quantos bits do endereço virtual destinam-se ao número da página? E ao deslocamento?  
 b) Ilustre o comportamento da política de substituição LRU mostrando, a cada referência, quais páginas estão em memória, as faltas de páginas e as páginas escolhidas para descarte.

183) 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 (Hexa)
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 abaixo, 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 na página 3 com *page fault*.  
 (T + 5): O processo referencia um endereço na página 6 com *page fault*.

- a) Em quais instantes de tempo ocorrem um *page out*?  
 b) Em que instantes de tempo o limite de páginas do processo na memória principal é atingido?  
 c) Caso a política de realocação de páginas utilizada seja o 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 falta de página?
- 184) Um sistema possui quatro frames. A tabela abaixo apresenta para cada página o momento da carga, o momento do último acesso, o bit de referência e o bit de modificação. Responda:
- | 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 pagina será substituída utilizando o algoritmo NRU?  
 b) Qual pagina será substituída utilizando o algoritmo FIFO?  
 c) Qual pagina será substituída utilizando o algoritmo LRU?
- 185) Considere um processo com limite de páginas reais igual a quatro e um sistema que implemente a política de substituição de páginas FIFO. Quantos *page faults* ocorrerão considerando que as páginas virtuais são referenciadas na seguinte ordem: 0172327103. Repita o problema utilizando a política LRU e Segunda Chance
- 186) O que é *trashing*?

-X-

# 12

## Sistema de Arquivos

*“Se o conhecimento pode criar problemas, não é através da ignorância que podemos solucioná-los.” (Isaac Asimov)*

### 12.1 INTRODUÇÃO

O sistema de arquivos é a parte do Sistema Operacional mais visível para os usuários. Por isso o sistema de arquivos deve apresentar uma interface coerente e simples. Ao mesmo tempo, arquivos são normalmente implementados a partir de discos magnéticos e como um acesso a disco demora cerca de 10000 vezes mais tempo do que um acesso à memória principal. São necessárias estruturas de dados e algoritmos que optimizem os acessos a disco gerados pela manipulação de arquivos.

É importante observar que sistemas de arquivos implementam um recurso em software que não existe no hardware. O hardware oferece simplesmente espaço em disco, na forma de setores que podem ser acessados individualmente, em uma ordem aleatória. O conceito de arquivo, muito mais útil que o simples espaço em disco, é uma abstração criada pelo Sistema Operacional. Neste caso, temos o Sistema Operacional criando um recurso lógico a partir dos recursos físicos existentes no sistema computacional.

### 12.2 ARQUIVOS

São recipientes que contém dados. Cada arquivo é identificado por um nome e por uma série de outros atributos que são mantidos pelo sistema operacional: tipo do conteúdo, tamanho, último acesso, última alteração.

O Sistema Operacional suporta diversas operações sobre arquivos, como criação, destruição, leitura, alteração, etc. Em geral, essas operações correspondem a chamadas de sistema que os programas de usuários podem usar para manipular arquivos.

Existe, na prática, uma enorme quantidade de diferentes tipos de arquivos, cada tipo com sua estrutura interna particular. Não é viável para o sistema operacional conhecer todos os tipos de arquivos existentes. Em geral, os sistemas operacionais ignoram a estrutura interna dos arquivos.

Para o sistema operacional, cada arquivo corresponde a uma seqüência de bytes, cujo significado é conhecido pelo usuário que criou o arquivo. A única exceção são os arquivos que contêm programas executáveis. Nesse caso, a estrutura interna é definida pelo próprio sistema operacional, responsável pela carga do programa para a memória quando esse deve ser executado.

Como o conceito de tipo de arquivo é útil para os usuários, muitos sistemas operacionais suportam nomes de arquivos onde o tipo é indicado. A forma usual é acrescentar uma extensão no nome que identifique o tipo do arquivo em questão.

#### 12.2.1 MÉTODO DE ACESSO

O acesso a arquivos pode se dar de duas maneiras:

**Seqüencial:** o conteúdo do arquivo pode ser lido seqüencialmente, pedaço a pedaço. O acesso seqüencial é muito usado. Por exemplo, compiladores fazem uma leitura seqüencial dos programas fontes. A impressão de um arquivo é feita também a partir de sua leitura seqüencial. Copiar o conteúdo de um arquivo corresponde a fazer uma leitura seqüencial do arquivo origem e uma escrita seqüencial do arquivo destino.

**Relativo:** muitas aplicações não podem ser implementadas como acesso seqüencial, pelo menos não com um desempenho aceitável. Nesse método de acesso, o programa inclui na chamada de sistema qual a posição do arquivo a ser lida.

### 12.2.2 IMPLEMENTAÇÃO DE ARQUIVOS

A forma básica de implementar arquivos é criar, para cada arquivo no sistema, um descritor de arquivo. O descritor de arquivo é um registro no qual são mantidas as informações a respeito do arquivo: nome, extensão, tamanho, datas, usuários, etc.

O descritor de arquivo deve ficar em disco e de preferência na mesma partição do arquivo. Só que, se a cada acesso ao descritor, o SO precisasse ler do disco, o sistema de arquivos teria um péssimo desempenho.

Para tornar mais rápido o acesso aos arquivos, o sistema de arquivos mantém na memória uma tabela contendo todos os descritores dos arquivos em uso, chamado Tabela dos Descritores de Arquivos Abertos (TDAA). Quando um arquivo entra em uso, o seu descritor é copiado do disco para a memória.

Quando o processo que está sendo executado faz uma chamada para abrir um arquivo (`open`), o sistema de arquivos realiza as seguintes tarefas:

Localiza no disco o descritor de arquivo cujo nome foi fornecido. Caso o arquivo não exista, é retornado um código de erro, que é passado para o processo que chamou o `open`. No caso de sucesso, retorna o par <partição, endereço do descritor>.

Verifica se o arquivo solicitado já se encontra aberto. Isso é feito através de uma pesquisa na TDAA. Tal pesquisa é feita tendo como chave não o nome do arquivo, mas sim o número da partição e o endereço do descritor.

Caso o arquivo não esteja aberto, aloca uma entrada livre na TDAA e copia o descritor do arquivo que está no disco para a entrada alocada na TDAA.

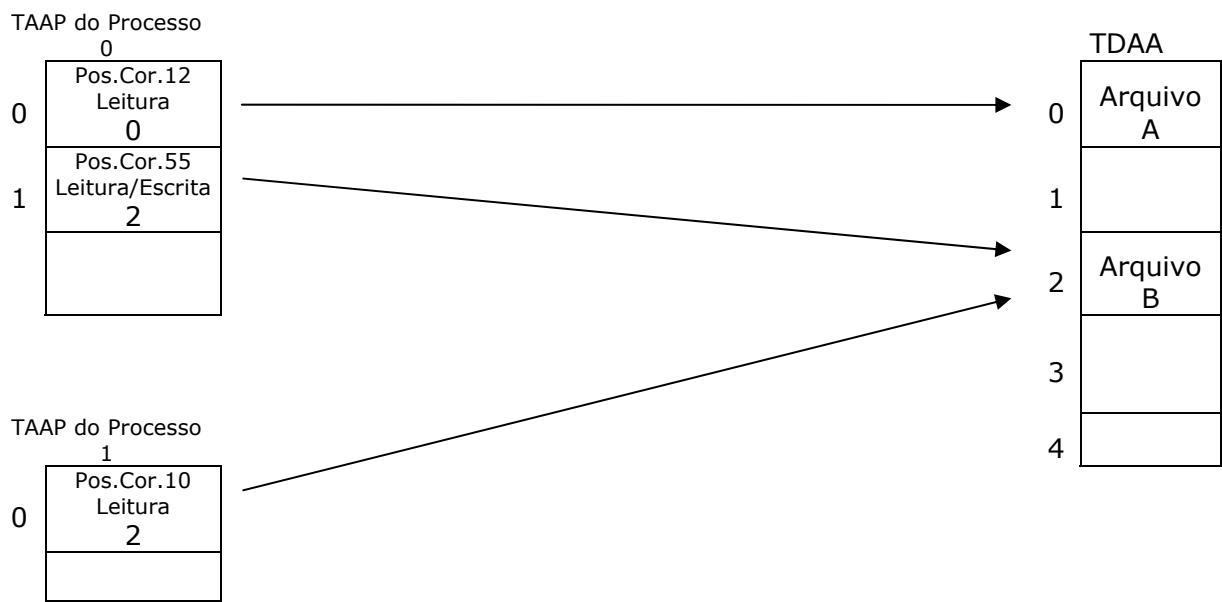
Uma vez que o descritor do arquivo foi copiado para a memória, verifica se o processo em questão tem o direito de abrir o arquivo conforme solicitado. Se não tiver, a entrada na TDAA é liberada e o `open` retorna um código de erro.

A partir desse momento, o arquivo está aberto e pode ser acessado. Quando um processo realiza a chamada de sistema `close`, o número de processos utilizando o arquivo em questão é decrementado na sua respectiva entrada da TDAA. Quando esse número chega a zero, significa que nenhum processo está usando o arquivo. Nesse caso, o descritor do arquivo é atualizado em disco e a entrada da TDAA liberada.

As entradas da TDAA armazenam informações que não variam conforme o processo que está acessando o arquivo. Por exemplo, o tamanho do arquivo é o mesmo, não importa qual processo execute o `read` ou `write`. Entretanto, existem informações diretamente associadas com o processo que acessa o arquivo. Como por exemplo a posição corrente no arquivo. Em um dado instante, cada processo deseja acessar uma parte diferente do arquivo. Da mesma forma, alguns processos podem abrir o arquivo para apenas leitura, enquanto outros abrem para leitura e escrita.

Essas informações não podem ser mantidas na TDAA pois, como vários processos podem acessar o mesmo arquivo, elas possuirão um valor diferente para cada processo. A solução é criar, para cada processo, uma Tabela de Arquivos Abertos por Processo (TAAP). Cada processo possui a sua TAAP. Cada entrada ocupada na TAAP corresponde a um arquivo aberto pelo processo correspondente. No mínimo, a TAAP contém em cada entrada: posição corrente no arquivo, tipo de acesso e apontador para a entrada correspondente na TDAA.

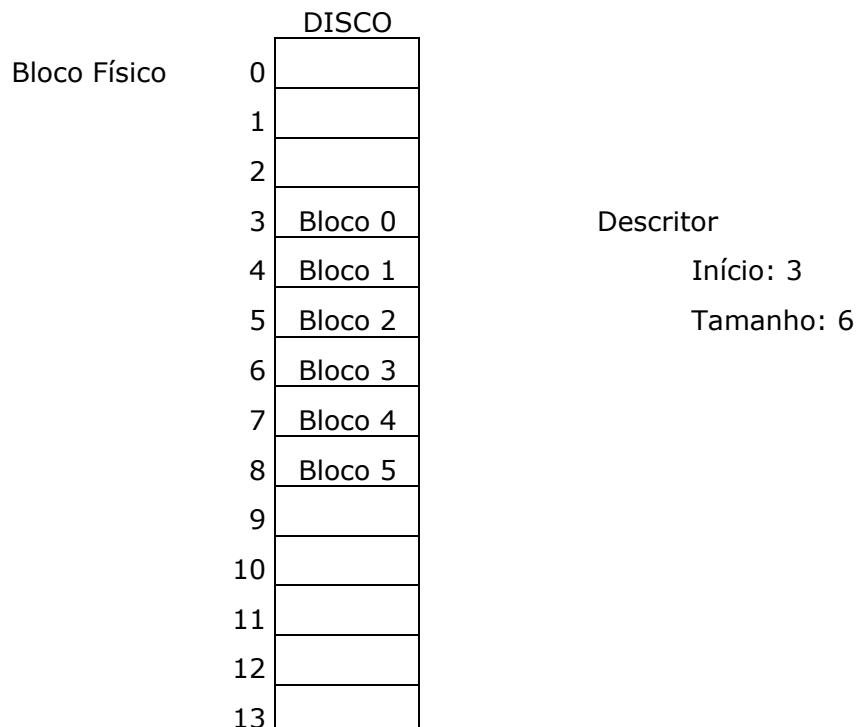
No exemplo, o processo 0 acessa o arquivo B através de referências a entrada 1 da sua TAAP. Muitos sistemas operacionais chamam esse número de *handle* do arquivo.



### 12.3 MÉTODOS DE ALOCAÇÃO

Para o SO, cada arquivo corresponde a uma seqüência de blocos lógicos, numerados a partir do zero. É preciso mapear os números de blocos lógicos desse arquivo em números de blocos físicos. A maneira como o mapeamento é realizado depende de como é mantida a informação “onde o arquivo está no disco”.

#### 12.3.1 ALOCAÇÃO CONTÍGUA



Dos métodos que objetivam associar blocos de disco a arquivos existentes, a alocação contígua é o mais simples. Neste método, os arquivos são armazenados no disco como um bloco contíguo de dados. Assim, para armazenar um arquivo de 70K em um disco com blocos de 1K seriam necessários 70 blocos consecutivos.

Vantagens:

- Simplicidade de implementação, pois para se acessar todo o arquivo basta que seja conhecido o endereço de seu primeiro bloco;
- Performance excelente, pois a leitura do arquivo em disco pode acontecer em uma única operação.

Vantagens:

- O tamanho máximo do arquivo tem que ser conhecido no momento em que ele for criado;
- Ocasiona fragmentação no disco.

### 12.3.2 ALOCAÇÃO COM LISTA LIGADA

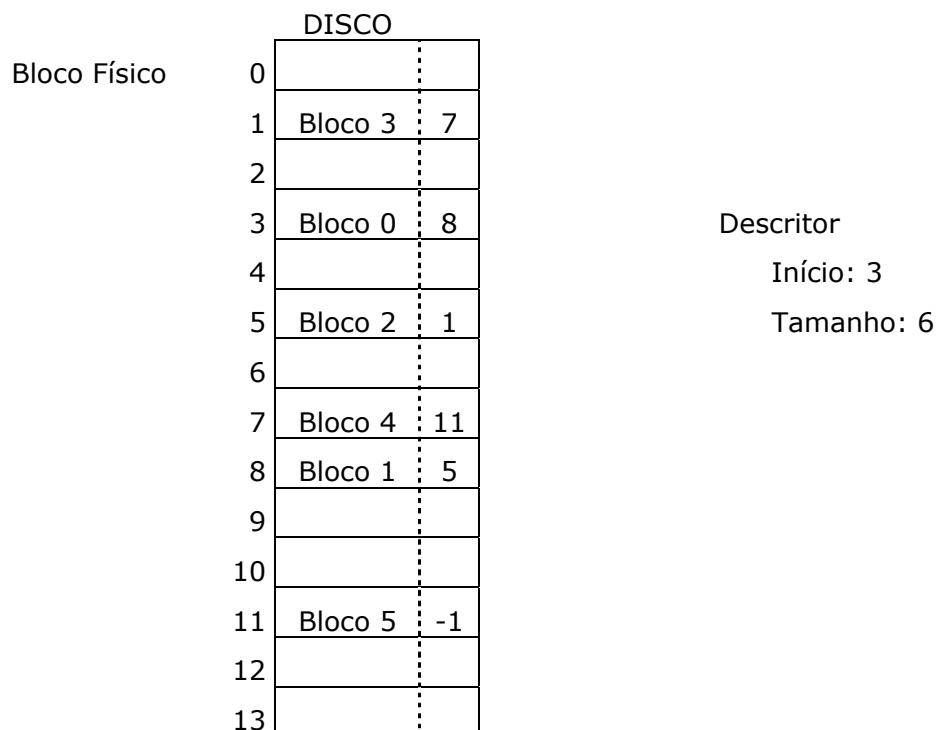
Neste método, cada bloco físico contém o endereço do próximo bloco físico alocado a esse mesmo arquivo.

Vantagens:

- Evita-se a perda de espaço em disco ocasionado pela fragmentação;
- Para se acessar todo o arquivo, basta que seja conhecido o endereço de seu primeiro bloco.

Desvantagens:

- Acesso randômico lento e difícil de ser implementado



### 12.3.3 ALOCAÇÃO COM LISTA LIGADA USANDO UM ÍNDICE

Com a finalidade de se eliminar os problemas encontrados no caso anterior, nesse método, cada arquivo possui uma tabela de índices. Cada entrada da tabela de índices contém o endereço de um dos blocos físicos que forma o arquivo.

Vantagens:

- Facilidade para se implementar um acesso randômico rápido, pois as entradas a serem seguidas encontram-se na memória
  - Para se acessar todo o arquivo basta que a entrada de diretório contenha o endereço de seu primeiro bloco.

Desvantagens:

- Toda a tabela deve estar na memória durante todo o tempo.

Bloco Físico	DISCO	DESCRITOR
0		Tamanho: 5
1	Bloco 3	Índices: 0
2		1 8
3	Bloco 0	2 5
4		3 1
5	Bloco 2	4 7
6		5
7	Bloco 4	6
8	Bloco 1	7
9		8

## 12.4 GERÊNCIA DE ESPAÇO EM DISCO

Existem alguns métodos que objetivam associar blocos de disco a arquivos existentes, no entanto, algumas considerações necessitam ser feitas com relação ao tamanho do bloco (unidade de alocação):

Unidade de alocação muito grande: causará desperdício de espaço dentro da unidade de alocação quando o arquivo for menor que a mesma;

Unidade de alocação muito pequena: um arquivo de tamanho razoável será composto de várias unidades de alocação (blocos) e a sua leitura do disco será lenta, pois a leitura de cada um dos blocos leva um certo tempo, devido ao tempo de *seek* e à latência rotacional do disco.

Obs.: Normalmente, temos blocos de tamanhos que variam de 512 bytes até 2KB.

## 12.5 CONTROLE DE BLOCOS LIVRES

Existem quatro maneiras mais usuais de se realizar o controle dos blocos não utilizados do disco:

### 12.5.1 MAPA DE BITS

Cada bloco é representado por 1 bit. Se o bloco estiver livre, o bit será 1; se ele estiver alocado, o bit será 0.

### 12.5.2 LISTA ENCADEADA

Outra abordagem é encadear os blocos de discos livres, mantendo um ponteiro ao primeiro bloco livre. Tal bloco contém um ponteiro ao próximo bloco livre, e assim por diante.

### 12.5.3 AGRUPAMENTO

Uma modificação da abordagem de lista livre é armazenar os endereços de n blocos livres no primeiro bloco livre. Os primeiros n-1 desses blocos estão realmente livres. O bloco final contém os endereços de outros n blocos livres, e assim por diante.

### 12.5.4 CONTADORES

Outra abordagem é aproveitar o fato de que, em geral, vários blocos contíguos podem ser alocados ou liberados simultaneamente, sobretudo quando o espaço é alocado com o algoritmo de alocação contígua. Portanto, em vez de manter uma lista de n endereços de disco livres, podemos manter o endereço do primeiro bloco livre e o número n de blocos contíguos livres que seguem esse primeiro bloco.

## 12.6 CONSISTÊNCIA DO SISTEMA DE ARQUIVOS

É importante que após uma pane no computador, o seu sistema de arquivos possa permanecer consistente. Para tal, alguns sistemas operacionais rodam um programa utilitário que faz a verificação do sistema de arquivos sempre que alguma pane ocorre. O UNIX utiliza o *fsck* e o Windows usa o *scandisk*.

O referido programa utilitário realiza, por exemplo, a verificação de consistência em blocos.

### 12.6.1 CONSISTÊNCIA EM BLOCOS

Neste tipo de verificação, o programa utilitário constrói uma tabela possuindo dois contadores: um indica quantas vezes um determinado bloco está presente em um arquivo e o outro informa quantas vezes um determinado bloco aparece como estando livre.

#### Situações indesejadas possíveis:

- Um bloco não possui ocorrência em nenhum arquivo e ele também não aparece na lista de blocos livres (bloco perdido).

Solução: colocar o bloco perdido na lista de blocos livres.

- Um bloco aparece duas vezes na lista de blocos livres.

Solução: reconstruir a lista de blocos livres.

- Um bloco está presente em mais de um arquivo.

Solução: alocar um novo bloco e copiar nele o conteúdo do bloco problemático. Em seguida, fazer um dos arquivos apontar para este novo bloco.

- Um bloco está presente em um arquivo e também na lista de blocos livres.

Solução: remover o bloco da lista de blocos livres.

## 12.7 PERFORMANCE DO SISTEMA DE ARQUIVOS

Sabemos que acessar arquivos localizados em discos é uma tarefa que consome um tempo relevante de um sistema computacional. Sendo assim, o projeto de um sistema de arquivos que possua uma boa performance deve procurar reduzir ao máximo possível o acesso a discos.

Dentre as técnicas mais utilizadas para diminuir o acesso ao disco, podemos destacar:

### 12.7.1 CACHE

A cache corresponde a um agrupamento de blocos que pertencem logicamente ao disco porém, se encontram armazenados em uma memória de alta velocidade, objetivando, assim, elevar a performance do sistema de arquivos.

Os algoritmos de substituição de páginas estudados (ótimo, NUR, FIFO, segunda chance, relógio e LRU) podem ser utilizados para gerenciar a cache.

Problema: Em caso de pane, os blocos que se encontram na cache podem não ser salvos no disco.

Solução: UNIX: implementa a chamada SYNC, que faz com que todos os blocos que sofreram modificações sejam imediatamente copiados para o disco.

**Obs.**: Quando o sistema é inicializado, um programa em background realiza uma chamada SYNC a cada 30 segundos.

MS-DOS: implementa a estratégia *write-through*, na qual a alteração em um bloco implica em sua cópia imediata para o disco.

### 12.7.2 LEITURA ANTECIPADA DE BLOCOS

A segunda técnica para melhorar o desempenho do sistema de arquivos é tentar transferir os blocos para a cache antes que eles sejam necessários para aumentar a taxa de acertos.

Problema: Tal estratégia só funciona para arquivos que estejam sendo lidos seqüencialmente. Para um arquivo com acesso aleatório, a leitura antecipada piora a situação, fazendo leituras em blocos não usados e removendo blocos potencialmente úteis da cache.

Solução: Para verificar se vale a pena fazer a leitura antecipada, o sistema de arquivos pode monitorar os padrões de acesso de cada arquivo aberto. Por exemplo, um bit associado a cada arquivo indica se o arquivo está em “modo de acesso seqüencial” ou em um “modo de acesso aleatório”.

### 12.7.3 REDUÇÃO DO MOVIMENTO DO BRAÇO DO DISCO

Esta técnica objetiva gravar os blocos que serão mais acessados o mais perto possível uns dos outros, e será detalhada no próximo capítulo.

## 12.8 EXERCÍCIOS

- 187) Considerando que a unidade de alocação física é de 2k, o tamanho da memória secundária é de 64k e a primeira unidade de alocação está ocupada com a tabela de arquivos, faça o mapa de bits final depois de todos os passos a seguir. Em seguida, faça a tabela de contadores, de agrupamento e a lista ligada mostrando os espaços livres:
- Escrever o arquivo A de 11k
  - Escrever o arquivo B de 6k
  - Remover o arquivo A
  - Alocar o arquivo C seqüencial, de 15k.
  - Escrever o arquivo D com 31K, não seqüencial.
  - Remover o arquivo B
- 188) Qual a diferença entre fragmentação externa e fragmentação interna? Demonstre graficamente.
- 189) Relacione a Tabela de Descritores de Arquivos Abertos (TDAA) e a Tabela de Arquivos Abertos por Processo (TAAP), ilustrando e citando exemplos. Simule situações de processos sendo executados e abrindo ou criando arquivos.
- 190) Cite as vantagens e desvantagens dos métodos de alocação: contígua, encadeada e indexada. Demonstre através de figuras.
- 191) Existem alguns métodos que objetivam associar blocos de disco a arquivos existentes. No entanto, algumas considerações necessitam ser feitas com relação ao tamanho do bloco. Porque não se deve ter unidades de alocação nem tão grandes nem tão pequenas?
- 192) Considere que o sistema operacional XYZ utiliza contadores para o controle de seus blocos livres. A lista está reproduzida abaixo. Monte o mapa de bits, agrupamento e a lista encadeada partindo dos contadores observados:

26	4
10	7
02	3

- 193) Partindo agora de um mapa de bits utilizado para controle de blocos livres, monte uma tabela de agrupamento, considerando n = 5: 11101101 – 10101000 – 11001100 – 11100000 – 11111011 – 00100100 – 00000000

-X-

# 13

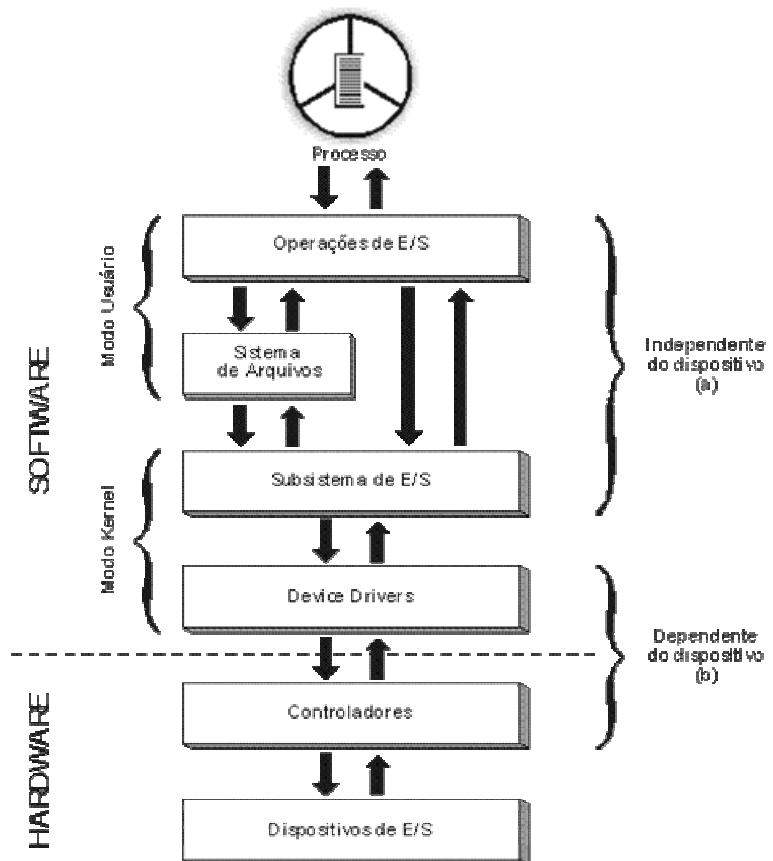
## Gerência de Dispositivos

*“A teoria sempre acaba, mais cedo ou mais tarde, assassinada pela experiência.”*  
 (Albert Einstein)

### 13.1 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. As camadas de mais baixo nível escondem características dos dispositivos das camadas superiores, oferecendo uma interface simples e confiável ao usuário e suas aplicações.

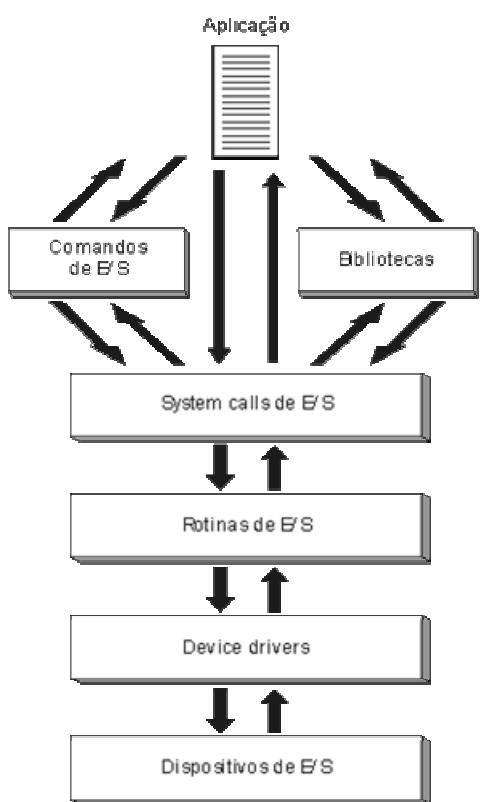
As camadas são divididas em dois grupos, onde o primeiro grupo visualiza os diversos tipos de dispositivos do sistema de um modo único (Fig. a), enquanto o segundo é específico para cada dispositivo (Fig. b). A maior parte das camadas trabalha de forma independente do dispositivo.



### 13.2 ACESSO AO 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. Para isso, o sistema possui um conjunto de rotinas, denominado *rotinas de entrada/saída*, que faz parte do subsistema de E/S e 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 devem ser realizadas através de chamadas de sistemas que chamam as rotinas de E/S do núcleo do Sistema Operacional. Dessa forma, é possível



escrever um programa que manipule arquivos, estesjam eles em disquetes, discos rígidos ou CD-Rom, sem ter que alterar o código para cada tipo de dispositivo.

A maneira mais simples de ter acesso a um dispositivo é através de comandos de leitura/gravação e chamadas a bibliotecas de rotinas oferecidas por linguagens de alto nível, como C. A comunicação entre os comandos de E/S oferecidos pelas linguagens de programação de alto nível e as chamadas de sistema de E/S é feita simplesmente através de passagem de parâmetros.

Um dos objetivos principais das chamadas de sistema de E/S é simplificar a interface entre as aplicações e os dispositivos. Com isso, elimina-se a necessidade de duplicação de rotinas idênticas nos diversos aplicativos, além de esconder do programador características específicas associadas à programação de cada dispositivo.

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. Assíncrona é 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.

### 13.3 SUBSISTEMA DE ENTRADA E SAÍDA

O subsistema de E/S é responsável por realizar as funções comuns a todos os tipos de dispositivos. É a parte do Sistema Operacional que oferece uma interface uniforme com as camadas superiores.

Independência de Dispositivos: 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 do dispositivo e repassá-la para os níveis superiores, sem o conhecimento do conteúdo da informação.

Tratamento de Erros: 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 nesse nível.

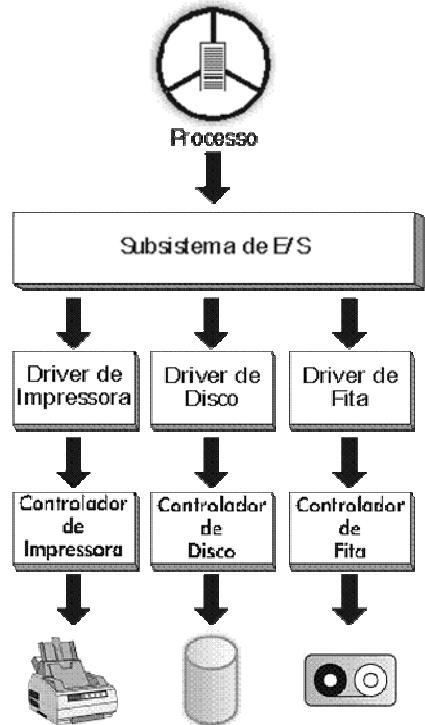
Compartilhamento: Todos os dispositivos de E/S são controlados, com o objetivo de obter o maior compartilhamento possível entre os diversos usuários de forma segura e confiável. Alguns dispositivos podem ser compartilhados simultaneamente, outros, como a impressora, devem ter acesso exclusivo, e o sistema deve controlar seu compartilhamento de forma organizada. É responsável também por implementar todo um mecanismo de proteção de acesso aos dispositivos. No momento que o usuário realiza uma operação de E/S, é verificado se o seu processo possui permissão para realizar a operação.

Bufferização: Essa técnica permite 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.

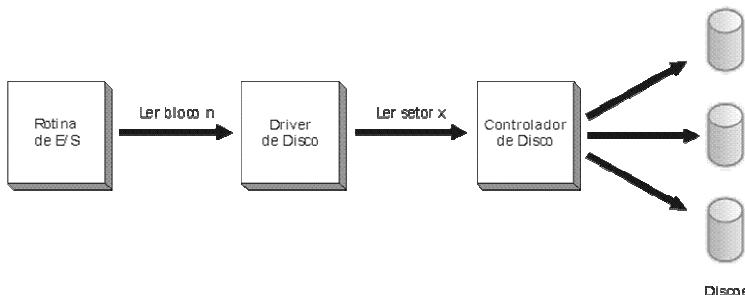
Interface Padronizada: O subsistema de E/S deve oferecer uma interface padronizada que permita a inclusão de novos drivers sem a necessidade de alteração da camada de subsistema de E/S.

#### 13.4 DEVICE DRIVERS

Tem como função implementar a comunicação do subsistema de E/S com os dispositivos, através de controladores. Os drivers tratam de aspectos particulares dos dispositivos. Recebem comandos gerais sobre acessos aos dispositivos e traduzem para comandos específicos, que poderão ser executados pelos controladores. Além disso, o driver pode realizar outras funções, como a inicialização do dispositivo e seu gerenciamento.

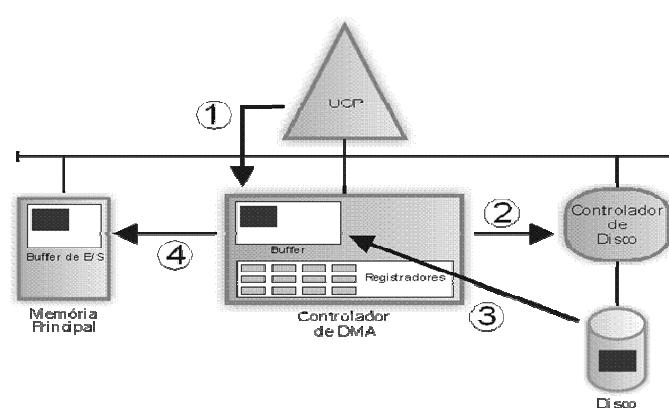


Por exemplo, na leitura síncrona de um dado em disco, o driver recebe a solicitação de um determinado bloco e informa ao controlador o disco, cilindro, trilha e setor que o bloco se localiza, iniciando, dessa forma, a operação. Enquanto se realiza a leitura, o processo que solicitou a operação é colocado 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 driver. Após verificar a inexistência de erros, o 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.



Os drivers fazem parte do núcleo do Sistema Operacional, sendo escritos geralmente em *assembly*. Devido ao alto grau de dependência entre os drivers e o restante do núcleo do SO, os fabricantes desenvolvem, para um mesmo dispositivo, diferentes drivers, um para cada Sistema Operacional. Sempre que um novo dispositivo é instalado, o driver 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.

#### 13.5 CONTROLADORES



São componentes de hardware responsáveis por manipular diretamente os dispositivos de E/S. Possuem memória e registradores próprios utilizados na execução de instruções enviadas pelo driver. 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 trans-

ferência do bloco pode ser realizado pela UCP ou por um controlador de DMA (Acesso Direto à Memória). O uso da técnica de DMA evita que o processador fique ocupado com a transferência do bloco para a memória.

De forma simplificada, uma operação de leitura em disco utilizando DMA teria os seguintes passos: 1) A UCP, através do driver, inicializa os registradores do controlador de DMA e, a partir desse ponto, fica livre para realizar outras atividades. 2) O controlador de DMA solicita ao controlador de disco a transferência do bloco do disco para o seu buffer interno. 3) Terminada a transferência, o controlador de disco verifica a existência de erros. 4) Caso não haja erros, o controlador de DMA transfere o bloco para o buffer de E/S na memória principal. 5) Ao término da transferência, o controlador de DMA gera uma interrupção avisando ao processador que o dado já se encontra na memória principal.

### 13.6 DISPOSITIVOS DE ENTRADA E SAÍDA

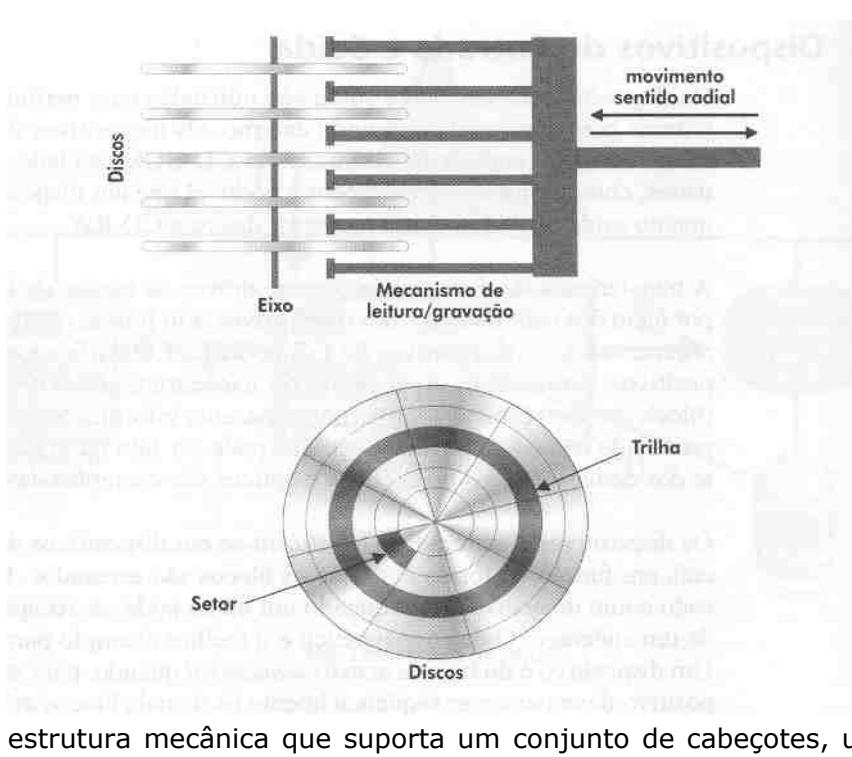
São utilizados para permitir a comunicação entre o sistema computacional e o mundo externo.

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: **estruturados** e **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.

Os dispositivos estruturados classificam-se em dispositivos de acesso direto e seqüencial, em função da forma com que os blocos são acessados. **Acesso direto** é quando um bloco pode ser recuperado diretamente através de um endereço, como por exemplo o disco magnético. **Acesso seqüencial** é quando, para se acessar um bloco, o dispositivo deve percorrer sequencialmente os demais blocos até encontrá-lo. A fita magnética é um bom exemplo.

Os dispositivos não-estruturados (*character devices*) são aqueles que enviam ou recebem uma seqüência de caracteres sem estar estruturada no formato de um bloco. A seqüência de caracteres não é endereçável, não permitindo operações de acesso direto ao dado. Terminais, impressoras e interfaces de rede são exemplos de tais dispositivos.

### 13.7 DISCOS RÍGIDOS



Fisicamente, um disco rígido pode ser visto como composto por dois grandes blocos. O primeiro bloco é um conjunto de discos metálicos (aço ou alumínio) superpostos e dispostos em alturas diferentes com auxílio de um eixo central. As duas superfícies de cada um desses discos são recobertas por uma película magnética na qual os dados são gravados. No momento de acesso ao disco, essa estrutura é mantida em uma rotação constante (36000 rpm, por exemplo). O segundo bloco é uma estrutura mecânica que suporta um conjunto de cabeçotes, um para cada superfície de

disco. Essa estrutura é capaz de realizar movimentos de vai-e-vem de maneira que os cabeçotes possam ser deslocados desde a borda do disco até o centro. Graças ao movimento de rotação dos discos e ao movimento retilíneo dos cabeçotes, toda a superfície de cada disco pode ser alcançada por seu respectivo cabeçote.

Do ponto de vista da organização lógica, cada superfície de um disco é dividida em circunferências concêntricas denominadas trilhas. Cada trilha, por sua vez, é subdividida radialmente em unidades chamadas setores. Em geral todos os setores tem o mesmo tamanho, o qual varia entre 512 a 4096 bytes. O setor constitui a unidade mínima de leitura e gravação em um disco. O conjunto de trilhas de todas as superfícies do disco que ficam exatamente à mesma distância do eixo central forma o cilindro. A definição de trilhas e de setores em um disco chama-se formatação física e é um procedimento realizado no fabricante.

Outros termos bastante comuns associados a discos rígidos são formatação lógica e partições. Ambos os conceitos estão mais relacionados com o sistema de arquivos do que com o disco propriamente dito. A formatação lógica consiste em gravar informações no disco de forma que arquivos possam ser escritos, lidos e localizados pelo sistema operacional.

### 13.7.1 TEMPO DE ACESSO

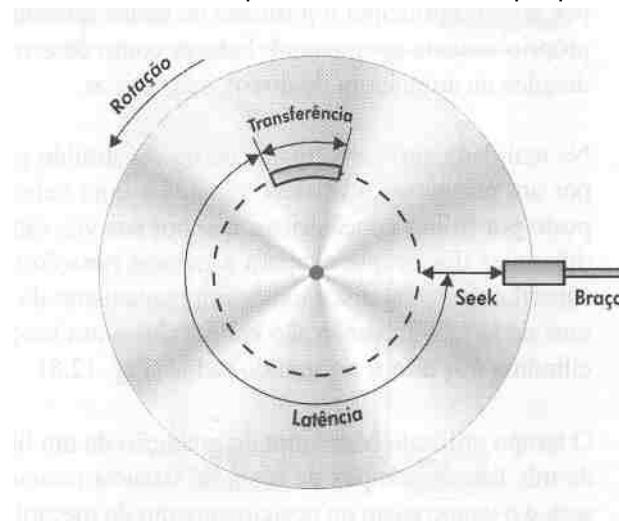
Para realizar um acesso a um disco rígido, é necessário posicionar o cabeçote de leitura e escrita sob um determinado setor e trilha onde o dado será lido ou escrito. Considerando-se a organização de um disco, esse procedimento de posicionamento implica um certo tempo: o tempo de acesso. O tempo de acesso é definido por três fatores:

$$\text{Tacesso} = \text{tseek} + \text{tlatência} + \text{ttransferência}$$

Tempo de Seek: tempo de locomoção do braço do disco até o cilindro desejado;

Latência Rotacional: tempo para que o setor desejado passe sob a cabeça de leitura;

Tempo de Transferência: tempo de transferência propriamente dito.



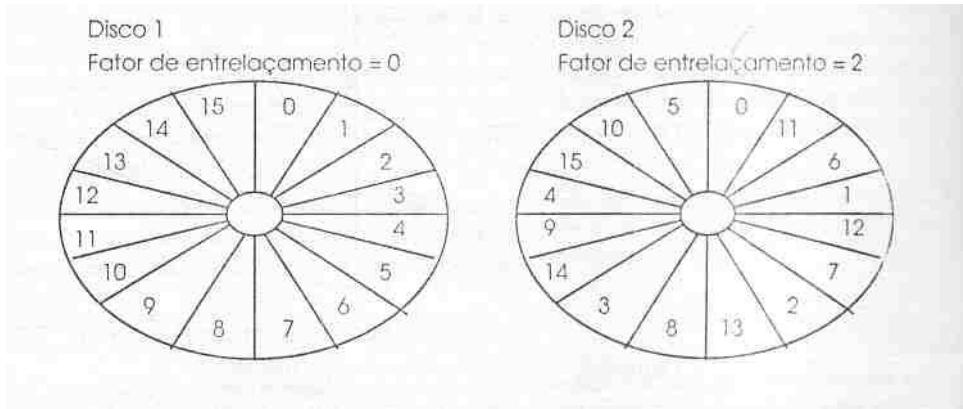
### 13.7.2 ENTRELAÇAMENTO (interleaving)

É muito comum o acesso a vários setores contíguos em uma trilha do disco. Suponha que se deseja ler os setores 4 e 5 de uma determinada trilha. O sistema operacional envia à controladora de disco o comando para ler o setor 4. Após o seek apropriado, o cabeçote passa sobre o setor 4, a transferência ocorre. Quando o cabeçote sai do setor 4, os dados são transferidos do buffer do controlador de disco para a memória, provocando uma interrupção no processador para informar o término da leitura do setor 4. Nesse momento, o processador (via sistema operacional) envia um novo comando de leitura para o setor 5. Um novo seek não será necessário, pois o cabeçote de leitura já se encontra sobre o cilindro desejado. Entretanto, devido à rotação do disco, o cabeçote provavelmente não se encontra mais no início do setor 5. Será necessário esperar que o disco dê uma volta completa (tempo de latência) para então efetuar a leitura do setor 5.

A forma usual para evitar esse problema é realizar um entrelaçamento dos setores (interleaving). Essa técnica numera os setores não mais de forma contígua mas sim com um espaço entre eles. Um disco com fator de entrelaçamento ou interleaving 2, significa que entre o setor k e o setor k+1, existem dois outros setores.

Voltando ao exemplo no qual os setores 4 e 5 são lidos, mas agora considerando um entrelaçamento de 2, observamos que, após o cabeçote sair do setor 4, ele passa por outros dois setores antes de chegar no início do setor 5. desta forma, a transferência dos dados relativos ao setor 4 pode ser concluída, e o processador tem a chance de mandar o comando de leitura do setor 5 antes que o cabeçote passe sobre o início do mesmo.

O melhor fator de entrelaçamento para uma determinada unidade de disco depende da velocidade do processador, do barramento, do controlador e da velocidade de rotação do disco.



### 13.7.3 ESCALONAMENTO DO BRAÇO DO DISCO

Como já vimos, uma das principais funções do sistema operacional é gerenciar os recursos do sistema de forma eficaz. O problema no caso do disco rígido está em como ordenar e atender os pedidos de entrada e saída de forma a maximizar o atendimento e minimizar o tempo em que os processos permanecerão bloqueados.

O tempo necessário a uma operação de entrada e saída com discos é fortemente influenciado pelo tempo de acesso ao disco. O objetivo então é minimizar os movimentos da cabeça de leitura e maximizar o número de bytes transferidos (throughput) de forma a atender o maior número possível de requisições no menor intervalo de tempo possível. Para resolver esse problema, existe um conjunto de algoritmos para realizar a movimentação do cabeçote de leitura do disco, alguns tentando otimizar a movimentação entre as trilhas e outros tentando aproveitar o percurso das cabeças de leitura:

#### a) 7.3.1. Primeiro a Entrar, Primeiro a Ser Servido (FCFS)

Neste algoritmo, o driver só aceita uma requisição de cada vez. O atendimento às requisições é feito considerando-se a ordem de chegada das mesmas.

Requisição : 11, 1, 36, 16, 34, 9 e 12

Deslocamento do Braço: 10, 35, 20, 18, 25 e 3

Total Percorrido: 111

#### b) 7.3.2. Menor Seek Primeiro (SSF)

Neste algoritmo, a próxima requisição a ser atendida será aquela que exigir o menor deslocamento do braço do disco.

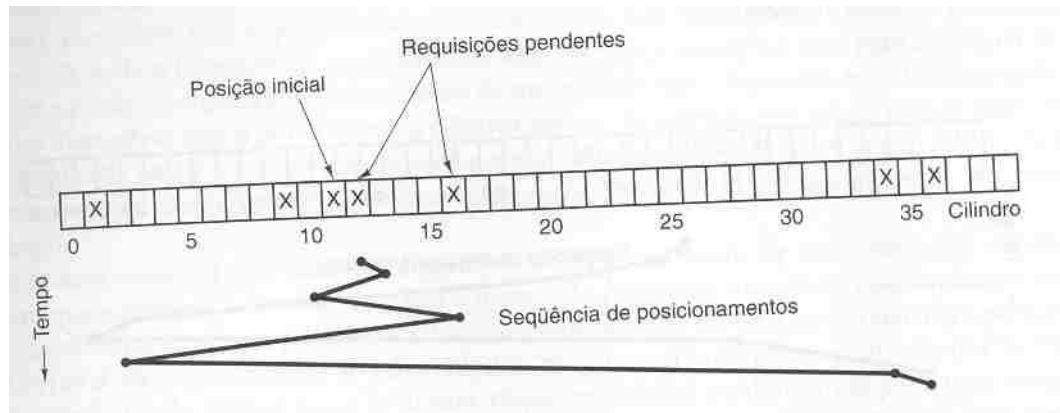
Requisição: 11, 1, 36, 16, 34, 9 e 12

Seqüência de acesso: 11, 12, 9, 16, 1, 34 e 36

Deslocamento do Braço: 1, 3, 7, 15, 33 e 2

Total Percorrido: 61

Obs.: O SSF reduz quase metade do movimento do braço do disco, quando comparado com o FCFS, porém apresenta tendência em acessar o meio do disco, podendo levar um pedido de acesso à postergação indefinida.



#### c) 7.3.3. Algoritmo do Elevador (Scan)

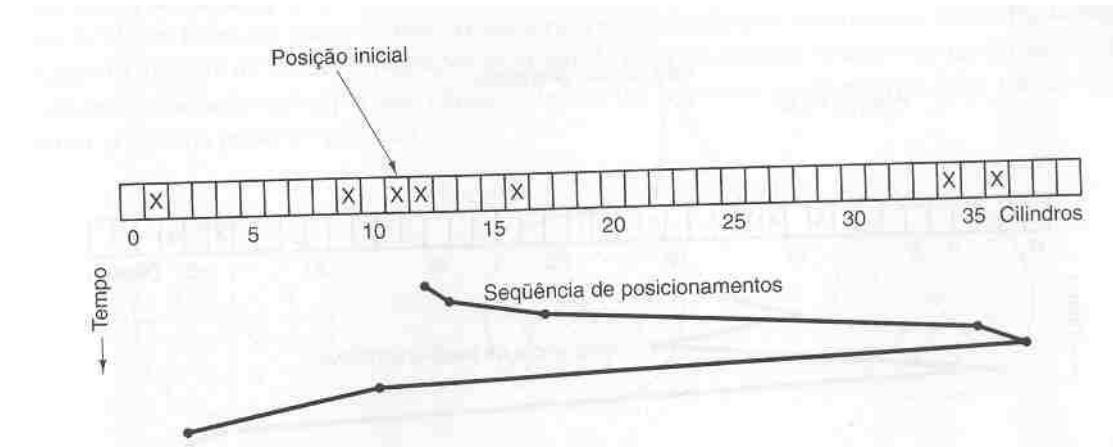
Como o próprio nome do algoritmo sugere, este faz com que todas as requisições em um determinado sentido sejam atendidas. Depois, as demais requisições em sentido oposto serão atendidas.

Requisição: 11, 1, 36, 16, 34, 9 e 12

Seqüência de acesso: 11, 12, 16, 34, 36, 9 e 1

Deslocamento do Braço: 1, 4, 18, 2, 27 e 8

Total Percorrido: 60



#### d) 7.3.4. Algoritmo C-Scan

O algoritmo Scan, imediatamente após inverter a varredura, inicia o atendimento privilegiando os pedidos correspondentes aos cilindros recém servidos, por consequência os pedidos dos cilindros do outro extremo da varredura, feitos anteriormente, devem esperar. O algoritmo C-Scan é uma variação do Scan com o objetivo de eliminar esse privilégio.

Nesse caso, quando a varredura atinge o último cilindro, o cabeçote é posicionado no primeiro cilindro onde reinicia a varredura. Em outros termos, os pedidos são atendidos em um só sentido da varredura.

Requisição: 11, 1, 36, 16, 34, 9 e 12

Seqüência de acesso: 11, 12, 16, 34, 36, 1 e 9

Deslocamento do Braço: 1, 4, 18, 2, 35 e 8

Total Percorrido: 68

### 13.8 EXERCÍCIOS

- 194) Explique o modelo de camadas aplicado na gerência de dispositivos
- 195) Qual a principal finalidade das rotinas de E/S?
- 196) Quais as diferentes formas de um programa chamar rotinas de E/S?
- 197) Quais as principais funções do subsistema de E/S?
- 198) Qual a principal função de um driver?
- 199) Por que o sistema de E/S deve criar uma interface padronizada com os device drivers?
- 200) Explique o funcionamento da técnica de DMA e sua principal vantagem.
- 201) Diferencie os dispositivos de E/S estruturados dos não-estruturados.
- 202) Desenhe um disco com 21 setores e um fator 3 de entrelaçamento. Mostre agora porque há uma minimização no tempo de acesso utilizando esse artifício.
- 203) As requisições do disco chegam ao driver do disco na seguinte ordem dos cilindros: 10, 22, 20, 2, 40, 6 e 38. Um posicionamento leva 6ms por cilindro movido. Quanto tempo é necessário para os algoritmos FCFS, SSF, Scan e C-Scan, considerando que o braço está inicialmente no cilindro 20?
- 204) Monte um quadro mostrando a seqüência do braço, o deslocamento a cada instante e o deslocamento total percorrido pelo braço utilizando os algoritmos FCFS, SSF, Scan e C-Scan. Considere as seguintes requisições e que o braço estava em repouso:

Chegada	Requisições
0	1, 6, 20
2	13
3	5, 19
4	7, 9, 15
5	18, 32

-X-

# 14

## Multimídia

*“A precisão numérica é a própria alma da ciência.”*  
 (D’arcy Wentworth Thompson)

### 14.1 INTRODUÇÃO

Filmes, fotos e música digitais estão se tornando, cada vez mais, meios comuns de apresentar informação e entretenimento usando um computador. Arquivos de áudio e vídeo podem ser armazenados em um disco e reproduzidos sob demanda. Contudo, suas características são muito diferentes dos tradicionais arquivos de texto para os quais foram projetados os atuais sistemas de arquivos. Como consequência, são necessários novos tipos de sistema de arquivos. Mais ainda: o armazenamento e a reprodução de áudio e vídeo impõem novas exigências ao escalonador e também a outras partes do sistema operacional.

A grande busca do mundo multimídia atualmente, é o vídeo sob demanda, que implica a capacidade de um consumidor em casa, selecionar um filme usando o controle remoto do televisor (ou o mouse) e ter esse filme imediatamente exibido na tela de sua televisão (ou monitor de seu computador). Para viabilizar o vídeo sob demanda é necessária uma infra-estrutura especial. Na Figura vemos duas dessas infra-estruturas. Cada uma tem três componentes essenciais: um ou mais servidores de vídeo, uma rede de distribuição e uma caixa digital (*set-top box*) em cada casa para decodificar o sinal.

O servidor de vídeo é um computador potente que armazena muitos filmes em seu sistema de arquivos e os reproduz sob demanda. Algumas vezes, computadores de grande porte são usados como servidores de vídeo, pois conectar, digamos, mil discos de grande capacidade a um computador de grande porte é bem menos complicado que conectar mil discos a qualquer tipo de computador pessoal.

A rede de distribuição entre o usuário e o servidor de vídeo deve ser capaz de transmitir dados em altas taxas e em tempo real.

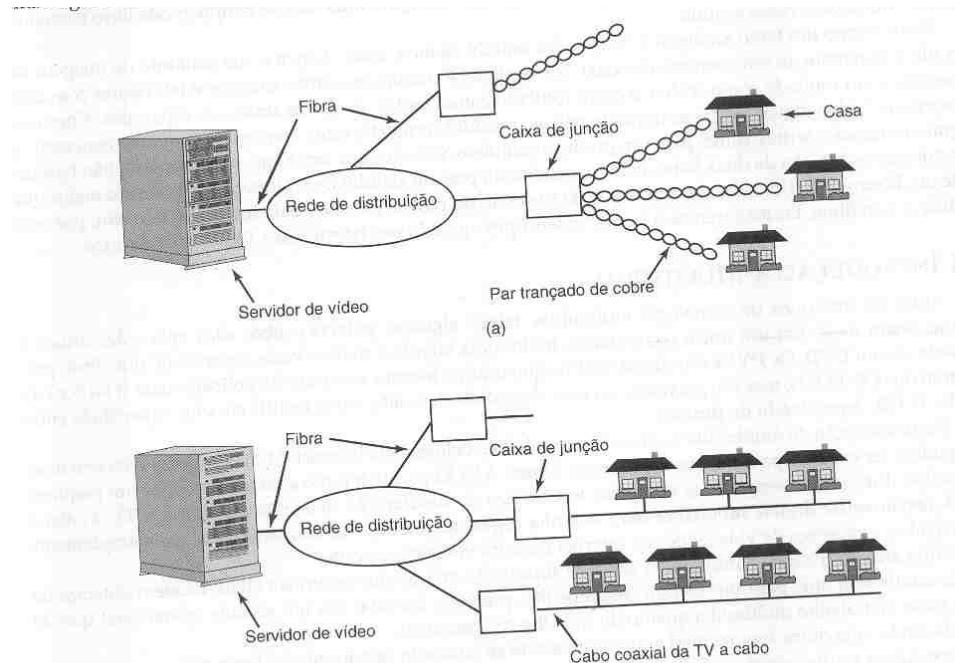


Figura: Vídeo sob demanda usando (a) ADSL e (b) TV a cabo

A última peça do sistema é a caixa digital, ou **set-top box**, ou seja, aonde chega o ADSL ou a TV a cabo. Esse dispositivo é, na verdade, um computador normal, com certos

chips especiais para decodificação e descompressão de vídeo. No mínimo, esse tipo de aparelho contém CPU, RAM, ROM e a interface para ADSL ou TV a cabo.

Há duas características fundamentais que devem ser muito bem entendidas para tratar a multimídia adequadamente:

- Usa taxas de dados extremamente altas.
- Requer reprodução em tempo real.

As altas taxas de dados resultam da natureza da informação visual e acústica. Os olhos e os ouvidos podem processar quantidades prodigiosas de informações por segundo e devem ser alimentados nessa taxa para produzir uma experiência sensorial aceitável. As taxas de dados de algumas fontes digitais multimídia e de certos dispositivos comuns de hardware são apresentados na tabela abaixo. O que se deve observar é a alta taxa de dados que a multimídia requer, a necessidade de compressão e a quantidade necessária de memória.

Origem	Mbps	GB/h	Dispositivo	Mbps
Telefone (PCM)	0,064	0,03	Fast Ethernet	100
Música em MP3	0,14	0,06	Disco EIDE	133
Áudio de CD	1,4	0,62	Rede ATM OC-3	156
Filme MPEG-2 (640 x 480)	4	1,76	Disco SCSI UltraWide	320
Câmera digital (720 x 480)	25	11	FireWire	400
TV (640 X 480)	221	97	Gigabit Ethernet	1000
HDTV (1280 X 720)	648	288	Disco SCSI Ultra-160	1280

A segunda exigência que a multimídia impõe sobre um sistema é a necessidade da entrega de dados em tempo real. A porção de vídeo de um filme digital consistem em alguns quadros por segundo. O sistema NTSC, usado na América do Norte, do Sul (exceto o Brasil) e no Japão), executa em 30 quadros/s; já os sistemas PAL e Secam, usados em grande parte dos demais países (no Brasil usa-se PAL/M), executam 25 quadros/s. Os quadros devem ser entregues em intervalos precisos de 33,3 ms ou 40 ms respectivamente, do contrário a imagem parecerá fragmentada.

NTSC é a sigla para *National Television Standards Committee* (Comissão Nacional para Padrões Televisivos). PAL significa *Phase Alternating Line* (Linha de Fase Alternativa). Tecnicamente é o melhor dos sistemas.

Nos seres humanos os ouvidos são mais sensíveis que os olhos; portanto, uma variação de até mesmo alguns milissegundos na exibição será notada. A variabilidade nas taxas de entrega é chamada **jitter** e deve ser estritamente limitada para se obter um bom desempenho. Observe que jitter não é o mesmo que atraso. Se a rede de distribuição atrasar uniformemente todos os bits por exatamente 5000 s, o filme começará um pouco mais tarde, mas será visto perfeitamente. Por outro lado, se os quadros forem aleatoriamente atrasados entre 100 e 200 ms, o filme se parecerá com filmes antigos.

As propriedades de tempo real necessárias para reproduzir multimídia de maneira aceitável são muitas vezes representadas por parâmetros de **qualidade de serviço**. Entre esses parâmetros estão largura da banda média disponível, pico de largura de banda, atraso mínimo e máximo e a probabilidade de perda de bit.

## 14.2 ARQUIVOS MULTIMÍDIA

Na maioria dos sistemas, um arquivo de texto comum é formado por uma seqüência de bytes sem qualquer estrutura que o Sistema Operacional possa reconhecer ou com ela se importar. Com a multimídia, a situação é mais complicada. Para começar, vídeo e áudio são completamente diferentes. Eles são capturados por dispositivos específicos (chip de CCD *versus* microfone), possuem estruturas internas diferentes (o vídeo tem 25-30 quadros/s; o áudio tem 44100 amostras/s) e são reproduzidos por dispositivos diferentes (monitor *versus* alto-falante).

Além disso, a maioria dos filmes almeja uma audiência mundial que, na sua maioria, não fala inglês. Isso pode ser resolvido de duas maneiras. Para alguns países é produzida uma trilha sonora adicional com vozes dubladas no idioma local. Em outros países, é usada a trilha sonora original com legendas no idioma local.

### 14.2.1 CODIFICAÇÃO DE ÁUDIO

Uma onda sonora (áudio) é uma onda acústica (de pressão) unidimensional. Quando uma onda acústica adentra o ouvido, o tímpano vibra fazendo com que os pequenos ossos do ouvido interno vibrem também, enviando pulsos ao cérebro. Esses pulsos são percebidos como sons pelo ouvinte. De uma maneira semelhante, quando uma onda acústica atinge um microfone, este gera um sinal elétrico representando a amplitude do som como uma função do tempo.

As ondas de áudio podem ser convertidas para a forma digital por um CAD (Conversor Analógico-Digital). Um CAD toma uma voltagem elétrica como entrada e gera um número binário como saída.

O intervalo de alcance de freqüência do ouvido humano vai de 20 a 20 mil Hz. O áudio dos CDs é digital com uma taxa de amostragem de 44100 amostras/s, suficiente para capturar freqüências de até 22050 Hz. As amostras são, cada uma, de 16 bits e lineares de acordo com o intervalo de amplitudes.

### 14.2.2 CODIFICAÇÃO DE VÍDEO

O olho humano funciona do seguinte modo: ao atingir a retina, uma imagem é retida por alguns milissegundos antes de desaparecer. Se uma seqüência de imagens atinge a retina em 50 ou mais imagens/s, o olho não percebe que estão sendo exibidas imagens discretas. Todos os sistemas de imagens em movimento baseados em filme ou em vídeo exploram esse princípio para produzir imagens em movimento.

Para entender sistemas de Vídeo é melhor começar com a simples e antiga televisão em preto-e-branco. Para representar a imagem bidimensional na tela da televisão como uma função unidimensional da voltagem em relação ao tempo, a câmera percorre um feixe de elétrons rapidamente de um lado para outro da imagem e lentamente de cima para baixo, registrando a intensidade luminosa conforme seu percurso. No final da varredura, chamada de **quadro**, o feixe volta à origem da tela (*retrace*). Essa intensidade como uma função do tempo é transmitida e os receptores repetem o processo de varredura para reconstruir a imagem.

Embora 25 quadros/s sejam suficientes para capturar movimentos suaves, nessa taxa de quadros muitas pessoas, especialmente as mais idosas, perceberão que a imagem tremula (porque a imagem anterior desaparece da retina antes que uma nova aparecesse). Em vez de aumentar a taxa de quadros, que requereria usar mais da escassa largura de banda, foi adotada uma abordagem diferente. Em vez de mostrar as linhas de varredura de cima para baixo, primeiro são exibidas todas as linhas de varredura ímpares e depois são exibidas as linhas pares. Cada um desses meio-quadros é chamado de **campo**. Experimentos mostraram que as pessoas percebem tremulação em 25 quadros/s, mas não a percebem em 50 campos/s. Essa técnica é chamada de **entrelaçamento**.

Vídeo em cores usa o mesmo padrão de varredura do monocromático, só que, em vez de mostrar a imagem com um feixe em movimento, são empregados três feixes movendo-se em uníssono. Um feixe é usado para cada uma das três cores primárias: vermelho, verde e azul (RGB – Red, Green e Blue).

Para permitir que transmissões em cores sejam vistas em receptores em preto-e-branco, todos os três sistemas combinam linearmente os sinais RGB em um sinal de **luminância** (brilho) e dois sinais de **crominância** (cor). O interessante é que o olho é muito mais sensível ao sinal de luminância do que aos sinais de crominância e, portanto, esses últimos não precisam ser transmitidos de modo tão preciso.

Até aqui vimos o vídeo analógico. Agora nos voltaremos para o vídeo digital. A representação mais simples de vídeo digital é uma seqüência de quadros, cada um constituído de uma grade retangular de elementos de imagem ou **pixels**. Para vídeo em cores, são usados 8 bits por pixel para cada uma das cores RGB, resultando em 16 milhões de co-

res, que é o suficiente. O olho humano não consegue nem mesmo distinguir essa quantidade de cores, o que dirá mais que isso.

Para produzir movimentos suaves, o vídeo digital, assim como o vídeo analógico, deve mostrar pelo menos 25 quadros/s. Contudo, como os bons monitores dos computadores atuais percorrem a tela atualizando as imagens armazenadas na RAM de vídeo 75 vezes por segundo ou mais, o entrelaçamento não é necessário. Conseqüentemente, todos os monitores de computadores usam varredura progressiva. Apenas redesenhar (isto é, renovar) o mesmo quadro três vezes em uma linha é suficiente para eliminar a tremulação.

Em outras palavras, a suavidade do movimento é determinada pelo número de imagens diferentes por segundo, enquanto a tremulação é estipulada pelo número de vezes que a tela é desenhada por segundo. Esses dois parâmetros são diferentes. Uma imagem parada desenhada a 20 quadros/s não mostrará movimentos espasmódicos, mas tremulará, pois o quadro desaparecerá da retina antes que o próximo apareça. Um filme com 20 quadros diferentes por segundo, cada qual desenhado quatro vezes seguidas a 80 Hz, não tremerá, mas o movimento parecerá espasmódico.

A importância desses dois parâmetros torna-se clara quando consideramos a largura de banda necessária para transmitir vídeo digital por uma rede. Os monitores atuais de computadores têm, todos, a taxa de aspecto 4:3 e, assim, podem usar aqueles tubos de imagem baratos e produzidos em massa para o mercado de televisores. Configurações comuns são 640 x 480 (VGA), 800 x 600 (SVGA) e 1024 x 768 (XGA). Uma tela XGA com 24 bits por pixel e 25 quadros/s precisa ser alimentado a 472 Mbps. Dobrar essa taxa para evitar tremulações não é uma medida interessante. Uma solução melhor é transmitir 25 quadros/s e fazer com que o computador armazene cada um deles e então os desenhe duas vezes. A transmissão de televisão não usa essa estratégia porque os televisores não têm memória e, além disso, sinais analógicos não podem ser armazenados em RAM sem que antes sejam convertidos para a forma digital, o que requer um hardware a mais. Como consequência, o entrelaçamento é necessário para difusão de televisão, mas não é necessário para o vídeo digital.

### 14.3 COMPRESSÃO DE VÍDEO

Agora, deve estar óbvio que manipular material multimídia sem compressão está fora de questão – esse material é muito extenso. A única esperança é que uma compressão macia seja possível.

Todos os sistemas de compressão precisam de dois algoritmos: um para comprimir os dados na origem e outro para descomprimi-los no destino. Na literatura, esses algoritmos são conhecidos como algoritmos de **codificação** e **decodificação**, respectivamente.

Esses algoritmos têm certas assimetrias que precisam ser compreendidas. Primeiramente, para muitas aplicações, um documento multimídia – por exemplo, um filme – será codificado somente uma vez (quando for armazenado no servidor multimídia), mas será decodificado milhares de vezes (quando for assistido pelos clientes). Essa assimetria revela que é aceitável que o algoritmo de codificação seja lento e que exija hardware sofisticado desde que o algoritmo de decodificação seja rápido e não requeira hardware sofisticado. Por outro lado, para multimídias em tempo real, como videoconferência, é inaceitável uma codificação lenta. A codificação deve ocorrer durante a própria videoconferência, em tempo real.

Uma segunda assimetria é que o processo codifica/decodifica não é reversível. Ou seja, no processo de compressão, transmissão e então descompressão de um arquivo, o usuário espera voltar ao arquivo original, precisamente até o último bit. Para multimídia essa exigência não existe. É perfeitamente aceitável que o sinal de vídeo, depois da codificação e da consequente decodificação, resulte em um sinal um pouco diferente do original.

#### 14.3.1 O PADRÃO JPEG

O padrão JPEG (*Joint Photographic Experts Goup* – grupo conjunto de especialistas em fotografia) para compressão de imagens paradas de tons contínuos (por exemplo,

fotografias) é importante para multimídia porque, de modo geral, o padrão multimídia para imagens em movimento, MPEG, é apenas a codificação JPEG de cada quadro em separado, mais alguns aspectos adicionais para compressão entre os quadros e de compensação de movimento.

O passo 1 da codificação de uma imagem em JPEG é a **preparação do bloco**. Para ser mais específico, suponha que uma entrada JPEG seja uma imagem RGB de 640 x 480 com 24 bits por pixel, conforme ilustra a Figura (a). Como o uso de luminância e de crominância oferece uma melhor compressão, são calculados a luminância e dois sinais de crominância a partir dos valores RGB. Para o NTSC, esses sinais são chamados  $Y$ ,  $I$  e  $Q$ , respectivamente. Para o PAL, são chamados respectivamente de  $Y$ ,  $U$  e  $V$  e as fórmulas são diferentes. A seguir usaremos os nomes correspondentes ao NTSC, mas o algoritmo de compressão é o mesmo.

São construídas matrizes separadas para  $Y$ ,  $I$  e  $Q$ , cada uma com elementos entre 0 e 255. Em seguida, é calculada a média de todos os blocos quadrados de 4 pixels nas matrizes  $I$  e  $Q$ , reduzindo-as a matrizes 320 x 240. Essa redução apresenta perdas, mas isso dificilmente é captado pela visão, pois o olho responde mais à luminância que à crominância. Mesmo assim, a compressão de dados é por um fator de dois. Então, de cada elemento de todas as três matrizes é subtraído 128 para que o 0 fique na metade do intervalo. Por fim, cada matriz é dividida em blocos de 8 x 8. A matriz  $Y$  tem 4800 blocos; as outras duas têm 1200 blocos cada, conforme mostra a Figura (b).

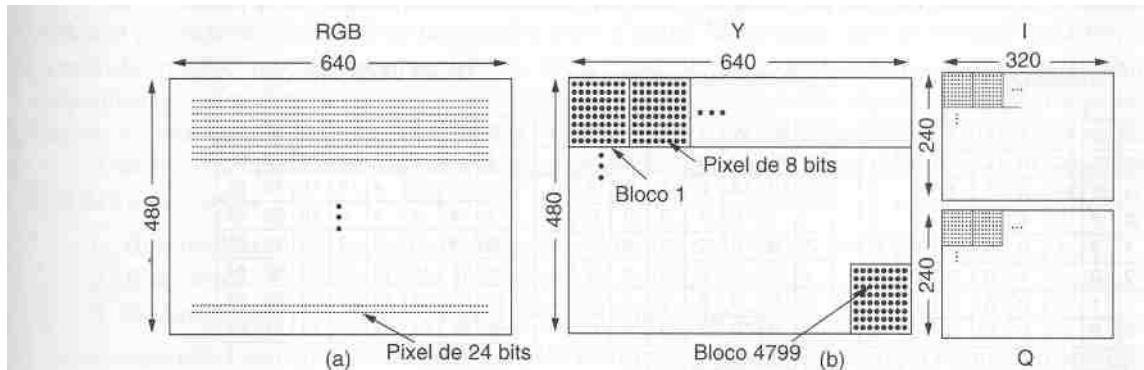


Figura: (a) Entrada de dados RGB. (b) Depois da preparação do bloco.

O passo 2 do JPEG é **aplicar uma transformação DCT** (*Discrete Cosine Transformation* – Transformação Discreta de Co-seno) para cada um dos 7200 blocos separadamente. A saída de cada OCT é uma matriz 8x8 de coeficientes DCT.

Uma vez terminada a DCT, o JPEG passará ao passo 3, que é chamado de **quantização**, no qual os coeficientes DCT menos importantes são eliminados. Essa transformação (com perda) é realizada dividindo-se cada um dos coeficientes da matriz DCT 8x8 por um peso tomado a partir de uma tabela. Se todos os pesos forem 1, a transformação não fará nada. Contudo, se os pesos crescerem abruptamente a partir da origem, as freqüências espaciais mais altas serão rapidamente reduzidas.

Um exemplo desse passo é mostrado na Figura abaixo, em que vemos a matriz DCT inicial, a tabela de quantização e o resultado obtido dividindo-se cada elemento DCT pelo elemento correspondente da tabela de quantização. Os valores na tabela de quantização não fazem parte do padrão JPEG. Cada aplicação deve fornecer sua tabela de quantização particular, possibilitando a essa aplicação controlar seu próprio compromisso entre perda e compressão.

O passo 4 **reduz o valor (0,0) de cada bloco** (o primeiro no canto superior esquerdo) substituindo-o pelo tanto que ele difere do elemento correspondente no bloco anterior. Como são valores médios de seus respectivos blocos, esses elementos devem mudar lentamente; portanto, assumir os valores diferenciais deve reduzir a maioria deles a pequenos valores. Nenhum diferencial é calculado a partir dos demais valores. Os valores (0,0) são conhecidos como componentes DC; os outros, como componentes AC.

Coeficientes DCT								Coeficientes quantizados								Tabela de quantização							
150	80	40	14	4	2	1	0	150	80	20	4	1	0	0	0	1	1	2	4	8	16	32	64
92	75	36	10	6	1	0	0	92	75	18	3	1	0	0	0	1	1	2	4	8	16	32	64
52	38	26	8	7	4	0	0	26	19	13	2	1	0	0	0	2	2	2	4	8	16	32	64
12	8	6	4	2	1	0	0	3	2	2	1	0	0	0	0	4	4	4	4	8	16	32	64
4	3	2	0	0	0	0	0	1	0	0	0	0	0	0	0	8	8	8	8	8	16	32	64
2	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	16	16	16	16	16	16	32	64
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	32	32	32	32	32	32	32	64
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	64	64	64	64	64	64	64	64

Figura: Computação dos coeficientes DCT quantizados.

O passo 5 **lineariza** os 64 elementos e aplica a codificação *run-length* a essa lista de elementos. Percorrer o bloco da esquerda para a direita e de cima para baixo não agrupará os zeros; assim, é aplicado um padrão de varredura em ziguezague, como mostra a Figura abaixo. Nesse exemplo, o padrão ziguezague resulta em 38 zeros consecutivos no final da matriz. Essa cadeia pode ser reduzida a apenas um contador indicando que há 38 zeros.

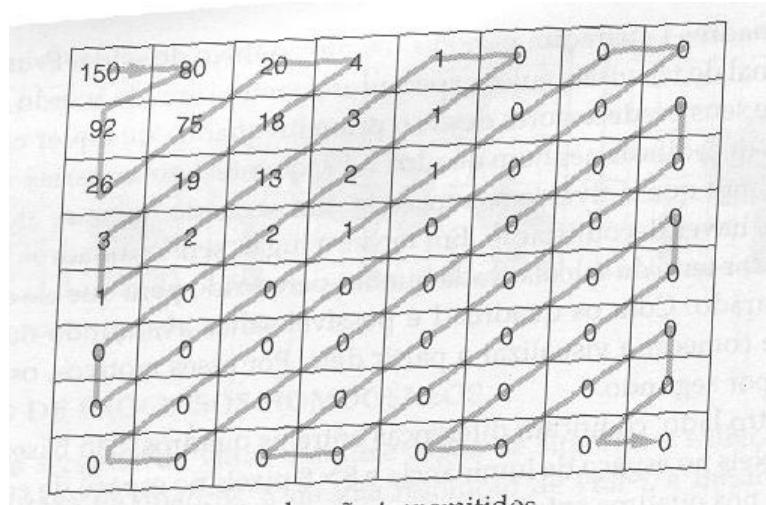


Figura: A ordem na qual os valores quantizados são transmitidos

Agora temos uma lista de números que representa a imagem (no espaço de transformação). O passo 6 **codifica, usando o código de Huffman**, para armazenamento ou transmissão.

O JPEG pode parecer complicado; isso ocorre porque o JPEG é complicado. Ainda assim, como produz uma compressão de 20:1, ou até melhor, o JPEG é amplamente usado. Decodificar uma imagem JPEG requer a execução do algoritmo de compressão de trás para a frente. O JPEG apresenta uma certa simetria: leva quase o mesmo tempo decodificar e codificar uma imagem.

#### 14.3.2 O PADRÃO MPEG

Finalmente, chegamos ao cerne do que nos interessa: os padrões MPEG (*Motion Picture Experts Group* – grupo de especialistas em imagens em movimento). Trata-se dos principais algoritmos usados para comprimir vídeos e são padrões internacionais desde 1993 e é adequado para canais de transmissão NTSC ou PAL.

Para cenas em que a câmera e o fundo sejam estacionários e um ou dois atores movam-se lentamente, quase todos os pixels serão idênticos de um quadro para o outro. Nesse caso, apenas subtrair cada frame do anterior e executar o JPEG na diferença funcionaria bem. Contudo, para cenas em que a câmera faz panorâmicas ou aproximações, essa técnica não se mostra muito adequada. É necessário algum modo de compensar esse movimento, e é exatamente isso que o MPEG faz - na verdade, essa é a diferença entre o MPEG e o JPEG.

A saída do MPEG consiste em três tipos diferentes de quadros que devem ser processados pelo programa de visualização:

I (Intracodificados): Imagens paradas autocontidas codificadas por JPEG.

P (Preditivos): Diferença bloco por bloco com o último quadro.

B (Bidirecionais): Diferenças entre o último e o próximo quadro.

Os quadros I são apenas imagens paradas codificadas pelo JPEG, usando também o sinal de luminância com resolução completa e a crominância com a metade da resolução ao longo de cada eixo.

Os quadros P, por outro lado, codificam diferenças entre os quadros. São baseados na idéia de macroblocos, que cobrem 16x16 pixels no espaço de luminância e 8x8 pixels no espaço de crominância. Um macrobloco é codificado buscando-se, nos quadros anteriores, um macrobloco idêntico ou com alguma pequena diferença.

Um exemplo da utilidade dos quadros P é ilustrado na Figura. Nela vemos três quadros consecutivos que têm o mesmo fundo, mas são diferentes quanto à posição de uma pessoa. Os macroblocos que contêm a cena de fundo serão idênticos, mas os macroblocos contendo a pessoa terão suas posições deslocadas por uma quantidade desconhecida e deverão ser acompanhados.

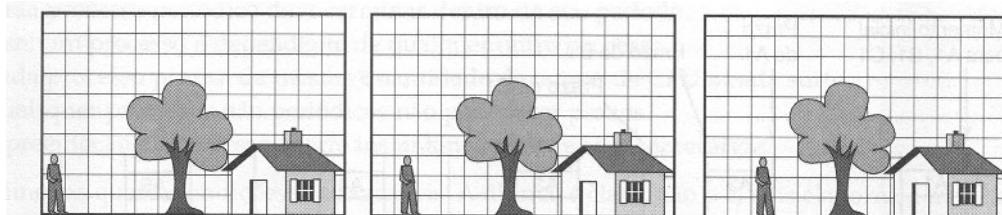


Figura: Três quadros consecutivos de vídeo

Os quadros B são similares aos quadros P, só que eles permitem que o macrobloco de referência esteja em um dos seguintes quadros: anterior ou sucessor, I ou P. Essa liberdade adicional permite a compensação melhorada do movimento e é útil também quando objetos passam pela frente ou por trás de outros objetos. Por exemplo, em um jogo de beisebol, quando o terceiro jogador da base arremessa a bola para a primeira base, pode haver alguns quadros nos quais a bola obscurece a cabeça do jogador que está na segunda base, ao fundo. No próximo quadro, a cabeça pode estar parcialmente visível, à esquerda da bola, com a aproximação seguinte da cabeça derivando-se do quadro posterior, quando a bola, então, já passou pela cabeça. Os quadros B permitem que um quadro seja baseado no quadro futuro.

Para fazer uma codificação do quadro B, o codificador precisa guardar, ao mesmo tempo, três quadros codificados na memória: o passado, o atual e o futuro. Para simplificar a decodificação, os quadros devem estar presentes, no fluxo MPEG, em ordem de dependência, e não em ordem de exibição. Portanto, mesmo com uma temporização perfeita, quando um vídeo é visto pela rede, faz-se necessário um armazenamento temporário na máquina do usuário para reordenar os quadros e resultar em uma exibição apropriada. Por causa dessa diferença entre a ordem de dependência e a ordem de exibição, tentar exibir um filme de trás para a frente não funcionará, a não ser que se disponha de um considerável armazenamento temporário e que se lance mão de algoritmos complexos.

#### 14.4 ESCALONAMENTO DE PROCESSOS MULTIMÍDIA

Os sistemas operacionais que suportam multimídia diferem dos tradicionais por três aspectos principais: pelo escalonamento de processos, pelo sistema de arquivos e pelo escalonamento do disco. Iniciaremos, com o escalonamento de processos e prosseguiremos com os outros tópicos nas seções subsequentes.

##### 14.4.1 ESCALONAMENTO DE PROCESSOS HOMOGÊNEOS

O tipo mais simples de servidor de vídeo é aquele capaz de suportar a exibição de um número fixo de filmes, todos com a mesma taxa de quadros, a mesma resolução de ví-

deo, a mesma taxa de dados e outros parâmetros. Sob essas circunstâncias, um algoritmo de escalonamento simples mas efetivo é como o seguinte. Para cada filme, há um único processo (ou thread) cujo trabalho é ler o filme do disco, um quadro por vez, e então transmitir esse quadro para o usuário. Como todos os processos são igualmente importantes, têm a mesma quantidade de trabalho por quadro e bloqueiam quando terminam de processar o quadro atual, o escalonamento por alternância circular (*round-robin*) realiza esse trabalho sem problemas. O único incremento necessário aos algoritmos comuns de escalonamento é o mecanismo de temporização para assegurar que cada processo execute na freqüência correta.

Um modo de conseguir a temporização apropriada é ter um relógio-mestre que pulse, por exemplo, 30 vezes por segundo (para NTSC). A cada pulso, todos os processos são executados seqüencialmente, na mesma ordem. Quando um processo termina seu trabalho, ele emite uma chamada ao sistema *suspend* que libera a CPU até que o relógio-mestre pulse novamente. Quando isso acontece, todos os processos são reexecutados na mesma ordem. Enquanto o número de processos for pequeno o bastante para que todo o trabalho possa ser feito em um intervalo de tempo, o escalonamento circular será suficiente.

#### 14.4.2 ESCALONAMENTO GERAL DE TEMPO REAL

Infelizmente trata-se de um modelo raramente aplicável à realidade. O número de usuários se altera conforme os espectadores vêm e vão, os tamanhos dos quadros variam exageradamente por causa da natureza da compressão de vídeo (quadros I são muito maiores que os quadros P ou B) e filmes diferentes podem ter resoluções diferentes. Como consequência, processos diferentes podem ser obrigados a executar em freqüências diferentes, com quantidades diferentes de trabalho e com prazos diferentes para terminar o trabalho.

Essas considerações levam a um modelo diferente: múltiplos processos competindo pela CPU, cada qual com seu próprio trabalho e seu próprio prazo. Nos próximos modelos vamos supor que o sistema saiba a freqüência na qual cada processo deve executar, quanto trabalho deve fazer e qual é seu prazo (*deadline*). O escalonamento de múltiplos processos em competição – alguns dos quais obrigados a cumprir prazos – é chamado de **escalonamento de tempo real**.

Um exemplo do tipo de ambiente com o qual um escalonador multimídia de tempo real trabalha: considere os três processos, A, B e C, mostrados na Figura. O processo A executa a cada 30 ms (aproximadamente a taxa do NTSC). Cada quadro requer 10 ms de tempo de CPU. Na ausência de competição, o processo executaria nos surtos A1, A2, A3 etc., cada um iniciando 30 ms depois do anterior. Cada surto da CPU trata um quadro e tem um prazo: ele deve terminar antes que um próximo inicie.

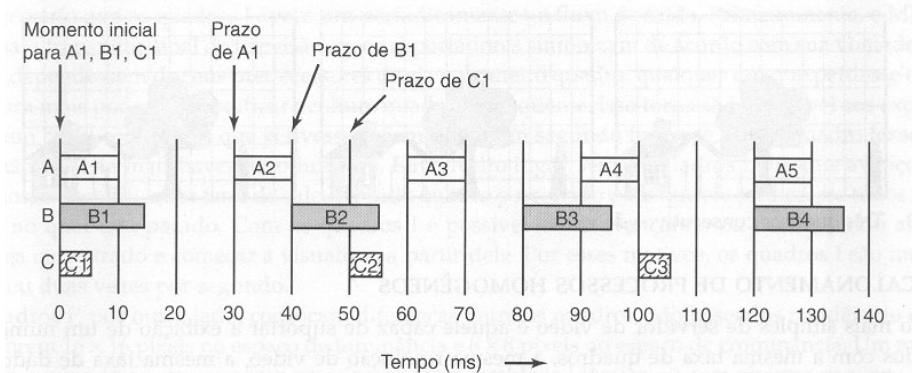


Figura: Três processos periódicos, cada um exibindo um filme. As taxas de quadro e os requisitos de processamento por quadro são diferentes para cada filme.

Também ilustrados na Figura estão dois outros processos, B e C. O processo B executa 25 vezes/s (por exemplo, PAL) e o processo C executa 20 vezes/s (por exemplo, um fluxo NTSC ou PAL mais lento, destinado a um usuário com uma conexão de baixa largura de banda para o servidor de vídeo). O tempo de computação por quadro é mostrado como 15 ms e 5 ms para B e C, respectivamente, apenas para tornar o problema de escalonamento mais geral do que tê-los todos iguais.

O problema do escalonamento agora é como escalonar A, B e C assegurando que eles cumpram seus respectivos prazos. Até agora presumimos que há um processo por fluxo. Na realidade, pode haver dois (ou mais) processos por fluxo - por exemplo, um para o áudio e outro para o vídeo. Eles podem executar em taxas diferentes e podem consumir quantidades diferentes de tempo de cpu por surto. Adicionar processos de áudio ao conjunto não alteraria o modelo geral, contudo, estamos presumindo que há  $m$  processos, cada um executando em uma freqüência específica, com uma quantidade fixa de trabalho necessário para cada surto de CPU.

Algoritmos de tempo real podem ser estáticos ou dinâmicos. Os algoritmos estáticos atribuem antecipadamente uma prioridade fixa a cada processo e então fazem o escalonamento preemptivo priorizado utilizando essas prioridades. Os algoritmos dinâmicos não apresentam prioridades fixas.

#### 14.4.3 ESCALONAMENTO POR TAXA MONOTÔNICA

O algoritmo clássico de escalonamento estático de tempo real para processos preemptivos e periódicos é o **escalonamento por taxas monotônicas** (*Rate Monotonic Scheduling* - RMS). Pode ser usado para processos que satisfaçam as seguintes condições:

1. Cada processo periódico deve terminar dentro de seu período.
2. Nenhum processo é dependente de qualquer outro processo.
3. Cada processo precisa da mesma quantidade de tempo de CPU a cada surto.
4. Quaisquer processos não periódicos não podem ter prazos.
5. A preempção de processo ocorre instantaneamente e sem sobrecargas.

O RMS funciona atribuindo a cada processo uma prioridade fixa igual à freqüência de ocorrência de seu evento de disparo. Por exemplo, um processo que deva executar a cada 30 ms (33 vezes/s) recebe prioridade 33; outro que tenha de executar a cada 40 ms (25 vezes/s), recebe prioridade 25; e um processo que execute a cada 50 ms (20 vezes/s) recebe prioridade 20. Portanto, as prioridades são lineares em relação à freqüência (número de vezes/segundo que o processo executa). Por isso, o escalonamento é chamado monotônico. Em tempo de execução, o escalonador sempre executa o processo que estiver pronto e com a prioridade mais alta, fazendo a preempção do processo em execução se for necessário.

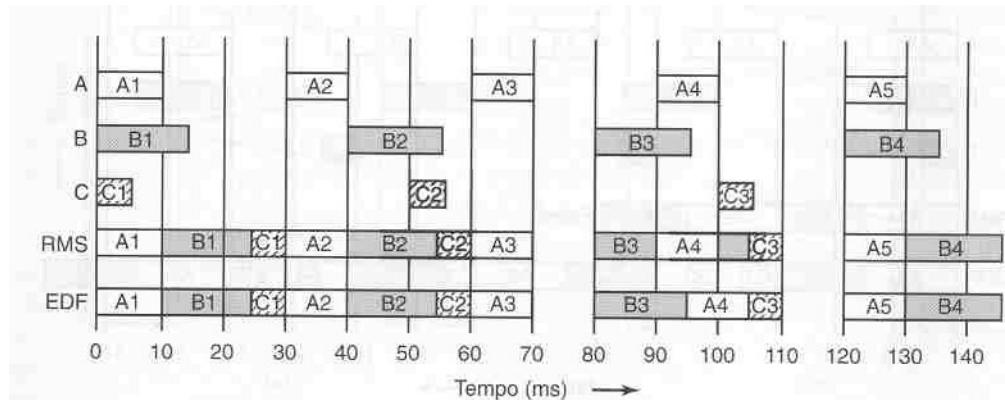


Figura: Um exemplo dos escalonamentos de tempo real RMS e EDF.

A Figura acima mostra como o escalonamento por taxas monotônicas funciona para o exemplo da Figura anterior. Os processos A, B e C têm prioridades estáticas 33, 25 e 20, respectivamente, o que significa que, se A precisa executar, ele executa, fazendo a preempção de qualquer outro processo que esteja usando a CPU. O processo B pode fazer a preempção de C, mas não de A. O processo C deve esperar até que a CPU fique ociosa para que possa executar.

Na Figura, inicialmente todos os três processos estão prontos para executar. O de prioridade mais alta, A, é escolhido e autorizado a executar até que termine em 15 ms, conforme ilustrado pela linha RMS. Depois de terminar, B e C executam naquela ordem.

Juntos esses processos levam 30 ms para executar e, assim, quando C terminar será a vez de A executar novamente. Essa rotação prossegue até que o sistema fique ocioso em  $t = 70$ .

Em  $t = 80$ , B fica pronto e executa. Contudo, em  $t = 90$ , um processo de prioridade mais alta, A, fica pronto e então faz a preempção de B e executa até que termine, em  $t = 100$ . Nesse ponto, o sistema pode escolher entre terminar B ou iniciar C e assim ele escolhe o processo de prioridade mais alta, ou seja, B.

#### 14.4.4 ESCALONAMENTO PRAZO MAIS CURTO PRIMEIRO

Outro algoritmo popular de escalonamento de tempo real é o do prazo mais curto primeiro (*Earliest Deadline First - EDF*). O EDF é um algoritmo dinâmico que não requer que os processos sejam periódicos, como no algoritmo do RMS. Também não exige o mesmo tempo de execução por surto de CPU, como o RMS. Se precisar de tempo de CPU, o processo anunciará sua presença e seu prazo. O escalonador tem uma lista de processos executáveis, ordenados por vencimentos de prazo. O algoritmo executa o primeiro processo da lista, aquele cujo prazo é o mais curto (mais próximo de vencer). Se um novo processo tornar-se pronto, o sistema verificará se seu prazo vence antes do prazo do processo em execução. Em caso afirmativo, o novo processo faz a preempção do que estiver executando.

Um exemplo de EDF é visto também na Figura anterior. Inicialmente, todos os três processos estão prontos. Eles são executados na ordem de vencimento de seus prazos. A deve terminar em  $t = 30$ , B deve terminar em  $t = 40$  e C deve terminar em  $t = 50$ ; portanto, o prazo de A vence antes e assim executa antes. Até  $t = 90$ , as escolhas são as mesmas do RMS. Em  $t = 90$ , A torna-se pronto novamente e o vencimento de seu prazo é  $t = 120$  – o mesmo vencimento de B. O escalonador poderia legitimamente escolher um ou outro para executar, mas, na prática, a preempção de B apresenta algum custo associado. É melhor, portanto, deixar B executando.

Para dissipar a idéia de que o RMS e o EDF sempre chegam aos mesmos resultados, estudemos um outro exemplo, mostrado na Figura abaixo. Nele, os períodos de A, B e C são os mesmos de antes, mas agora A precisa de 15 ms de tempo de cpu por surto, e não de apenas 10 ms.

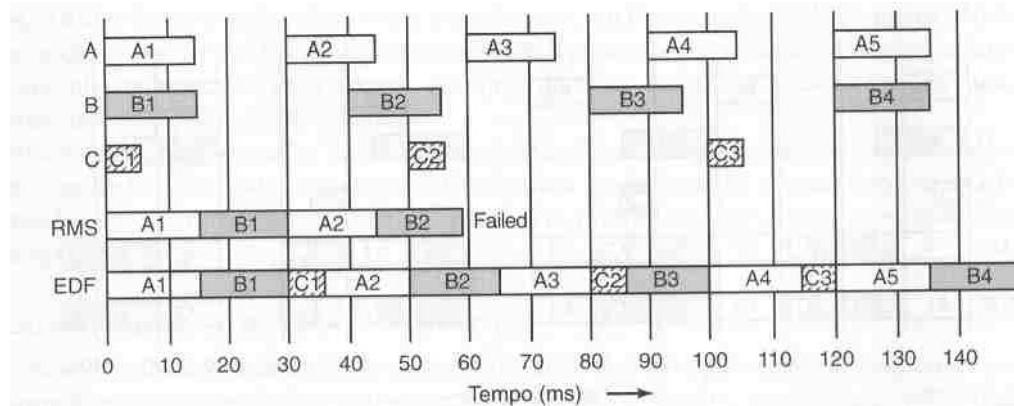


Figura: Outro exemplo de escalonamento em tempo real RMS e EDF.

Para o RMS, as prioridades dos três processos ainda são 33, 25 e 20, já que somente o período é o que interessa, e não o tempo de execução. Nesse caso, B1 não termina antes de  $t = 30$ , instante em que A está pronto para executar novamente. No momento em que A termina, em  $t = 45$ , B está pronto de novo e, portanto, com prioridade mais alta que C, B executa e o prazo de C vence. O RMS falha.

Agora vejamos como o EDF lida com esse problema. Em  $t = 30$ , há uma disputa entre A2 e C1. Como o prazo de C1 vence em 50 e o prazo de A2 vence em 60, C é escalonado. Isso é diferente do que ocorre no RMS, em que A ganha porque tem a prioridade mais alta.

Em  $t = 90$ , A fica pronto pela quarta vez. O vencimento do prazo de A é igual ao do processo em execução (120), portanto o escalonador deve escolher entre fazer ou não a

preempção. Assim como antes, se não for necessário, é melhor não fazer a preempção; desse modo, B3 é autorizado a terminar.

#### 14.5 PARADIGMAS DE SISTEMAS DE ARQUIVOS MULTIMÍDIA

Agora que cobrimos o escalonamento de processos em sistemas multimídia, prosseguiremos estudando os sistemas de arquivos multimídia (esses sistemas de arquivos usam um paradigma diferente dos sistemas de arquivos tradicionais). Para ter acesso a um arquivo, um processo emite, antes de tudo, uma chamada ao sistema `open`. Se não houver problemas, será dada uma espécie de identificador a quem chamou. Nesse ponto, o processo pode emitir uma chamada ao sistema `read`, fornecendo como parâmetros o identificador, o endereço do buffer e o número de bytes. O sistema operacional retorna então os dados requisitados no buffer. Podem ser feitas outras chamadas `read`, até que o processo termine. É então o momento de chamar `close` para fechar o arquivo e devolver seus recursos.

Esse modelo não funciona bem para multimídia por causa da necessidade de um comportamento de tempo real. Funciona ainda pior para mostrar arquivos multimídia que saem de um servidor remoto de vídeo. Um problema é que o usuário deve fazer as chamadas `read` precisamente espaçadas no tempo. Um segundo problema é que o servidor de vídeo deve ser capaz de fornecer os blocos de dados sem atraso - algo difícil de se conseguir quando as requisições vierem de modo não planejado e não houver recursos reservados antecipadamente.

Para resolver esses problemas, os servidores de arquivos multimídia usam um paradigma completamente diferente: eles agem como se fossem aparelhos de videocassete (*Video Cassette Recorders - VCR*). Para ler um arquivo multimídia, um processo-usuário emite uma chamada ao sistema `start`, especificando o arquivo a ser lido e vários outros parâmetros - por exemplo, quais trilhas de áudio e de legenda devem ser usadas. O servidor de vídeo começa, então, a enviar os quadros na taxa de quadros requisitada. Fica para o usuário tratá-los na taxa que os quadros vierem. Se o usuário ficar cansado do filme, a chamada ao sistema `stop` terminará o fluxo. Servidores de arquivos com esse modelo de fluxo são chamados de **servidores push** (pois eles empurram os dados para o usuário) e são diferentes dos tradicionais **servidores pull**, nos quais o usuário deve puxar os dados, um bloco de cada vez, chamando repetidamente o `read` para obter um bloco após o outro. A diferença entre esses dois modelos é ilustrada na Figura:

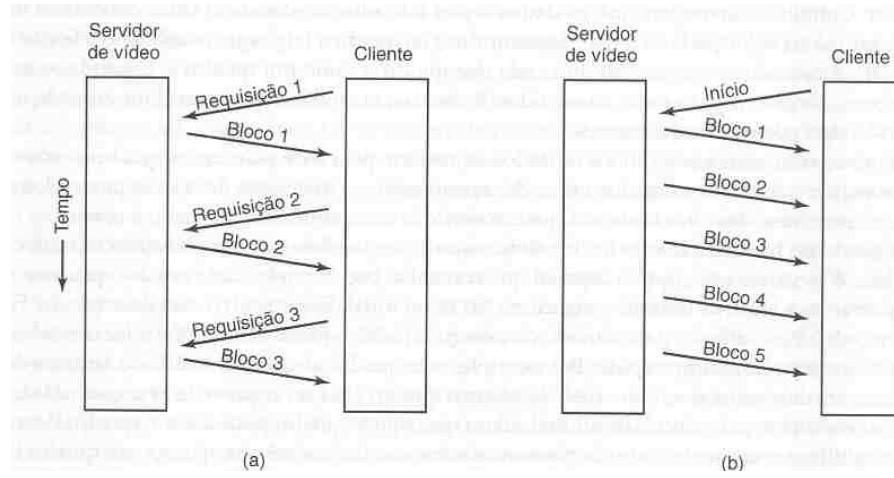


Figura: (a) Um servidor pull. (b) Um servidor push.

##### 14.5.1 FUNÇÕES DE CONTROLE VCR

A maioria dos servidores de vídeo implementa também funções-padrão de controle VCR, inclusive pausa, avanço rápido e rebobinamento. A **pausa** é bastante clara. O usuário envia uma mensagem para o servidor de vídeo pedindo que pare. Nesse caso, o que o servidor deve fazer é lembrar qual é o próximo quadro. Quando o usuário pedir para que o servidor prossiga, ele apenas continuará de onde parou.

Contudo, há uma complicação. Para obter um desempenho aceitável, o servidor pode reservar recursos como largura de banda de disco e buffers de memória para cada fluxo de saída. Continuar retendo esses recursos enquanto um filme está em pausa desperdiça recursos, especialmente se o usuário demorar. Claro que os recursos podem ser facilmente liberados quando estiverem em pausa, mas, se o usuário tentar continuar, haverá o perigo de não poder mais readquiri-los.

O **rebobinamento** é bem fácil. Tudo o que o servidor precisa fazer é lembrar que o próximo quadro a ser enviado é o 0. O que poderia ser mais simples? Contudo, o avanço e o retrocesso rápidos (isto é, reproduzir enquanto rebobina) são mais complicados. Se não fosse pela compressão, uma maneira de avançar em dez vezes a velocidade normal seria apenas exibir um quadro a cada dez. Para avançar em 20 vezes seria preciso mostrar um quadro a cada 20. Na verdade, sem compressão, avançar ou retroceder em qualquer velocidade é fácil. Para reproduzir em  $k$  vezes a velocidade normal, é só mostrar um quadro a cada  $k$  quadros. Para fazer o retrocesso rápido  $k$  vezes a velocidade normal é só fazer a mesma coisa na outra direção. Esse sistema funciona igualmente bem, tanto para os servidores pull quanto para os servidores push.

A compressão complica o movimento rápido para frente ou para trás. Com uma fita de vídeo digital, para a qual cada quadro é comprimido independentemente de todos os outros, é possível usar essa estratégia, desde que o quadro necessário possa ser encontrado rapidamente. Como a compressão de um quadro depende de seu conteúdo, cada quadro tem um tamanho diferente; portanto, saltar  $k$  quadros no arquivo não pode ser feito por um cálculo numérico. Além disso, a compressão de áudio é independente da compressão de vídeo; assim, para cada quadro de vídeo exibido no modo de alta velocidade deve ser encontrado também o quadro correto de áudio (a menos que o som seja desligado quando estiver reproduzindo em um modo mais rápido que o normal). Portanto, o avanço rápido de um arquivo de vídeo digital requer um índice que permita que os quadros sejam localizados rapidamente. Isso até na teoria é bastante complicado.

#### 14.5.2 VÍDEO QUASE SOB DEMANDA

Ter  $k$  usuários assistindo ao mesmo filme impõe, essencialmente, a mesma carga sobre o servidor que têm os assistindo a  $k$  filmes diferentes. Contudo, com uma pequena mudança no modelo são possíveis grandes ganhos de desempenho. O problema do vídeo sob demanda é que os usuários podem começar o fluxo de um novo filme em um momento arbitrário; portanto, se houver cem usuários, todos começando a ver algum novo filme por volta das 20h, há a possibilidade de que nunca dois usuários iniciem exatamente no mesmo instante - assim, eles não podem compartilhar um fluxo. A alteração que possibilita a otimização consiste em informar a todos os usuários que os filmes começam somente na hora cheia e depois de cada cinco minutos, por exemplo. Portanto, se um usuário quiser ver um filme às 20h02, ele terá de esperar até as 20h05.

A vantagem é que, para um filme de duas horas, são necessários somente 24 fluxos, não importando o número de consumidores. Conforme ilustra a Figura a seguir, o primeiro fluxo começa às 20h. Às 20h05, quando o primeiro fluxo estiver no quadro 9000, o fluxo 2 se iniciará. Às 20h10, quando o primeiro fluxo estiver no quadro 18000 e o fluxo 2 no quadro 9000, o fluxo 3 começará e assim irá até o fluxo 24, que se inicia às 21h55. Às 22h, o fluxo 1 termina e começa com o quadro 0. Esse esquema é chamado de **vídeo quase sob demanda** (*near video on demand*) porque o vídeo não começa no momento da requisição, mas um pouco depois dela.

De certo modo, o vídeo sob demanda é como usar um táxi: você chama e ele vem. O vídeo quase sob demanda é como usar um ônibus: há uma escala fixa de horários e é preciso esperar pelo próximo horário. Mas o trânsito em massa só faz sentido se houver a massa. No centro das cidades, um ônibus que passe a cada cinco minutos pode contar com pelo menos alguns passageiros. Um ônibus viajando pelas estradas do interior pode estar quase sempre vazio. Da mesma maneira, o lançamento do último filme do David Linch pode atrair clientes suficientes para garantir o início de um novo fluxo a cada cinco minutos; em contrapartida, em relação a "E o Vento Levou" talvez seja melhor simplesmente oferecê-lo sob demanda.

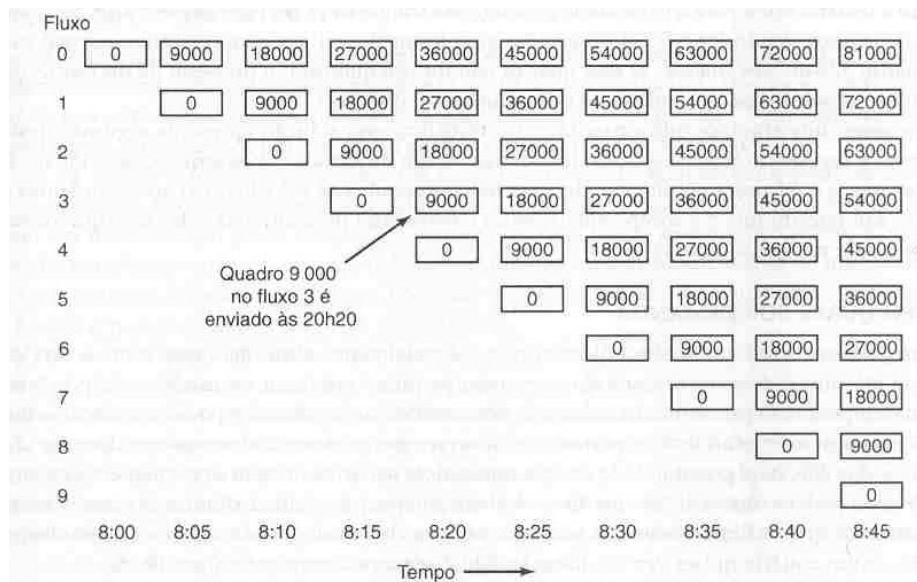


Figura: O vídeo quase sob demanda tem um novo fluxo iniciando em intervalos regulares; no exemplo dado, a cada cinco minutos (9000 quadros)

No vídeo quase sob demanda, os usuários não têm controles VCR. Nenhum usuário pode dar uma pausa no filme para uma excursão à cozinha. O melhor que pode ser feito é, quando retornar da cozinha, entrar em um fluxo que tenha começado depois, por isso alguns minutos do vídeo estarão repetidos.

#### 14.6 ALOCAÇÃO DE ARQUIVOS EM DISCOS

Os arquivos multimídia são bastante grandes e com freqüência são escritos somente uma vez, mas lidos muitas vezes e em geral sofrem acessos seqüenciais. Suas reproduções devem cumprir critérios estritos de qualidade de serviço. Juntas, essas exigências sugerem esquemas de sistemas de arquivos diferentes dos usados pelos sistemas operacionais tradicionais. Discutiremos alguns desses assuntos a seguir, partindo do contexto de um único disco e então para múltiplos discos.

##### 14.6.1 ALOCAÇÃO DE UM ARQUIVO EM UM ÚNICO DISCO

A exigência mais importante é que os dados, na forma de fluxos, possam fluir para a rede ou para um dispositivo de saída, na velocidade necessária e sem *jitter*. Por isso, ter várias buscas em disco (*seeks*) durante um quadro é altamente indesejável. Um modo de eliminar buscas intra-arquivos em servidores de vídeo é usar arquivos contíguos. Normalmente, arquivos contíguos não funcionam bem, mas em servidores de vídeo, que são carregados antecipadamente com filmes que não mudarão mais, isso pode funcionar.

No entanto, uma complicação é a presença de vídeo, áudio e texto. Mesmo que esses elementos estejam armazenados separadamente em arquivos contíguos, será necessária uma busca para ir do arquivo de vídeo para um arquivo de áudio e de lá para um arquivo de texto se este último for necessário. Isso sugere um segundo arranjo possível, com o vídeo, o áudio e o texto intercalados conforme mostra a Figura, mas todo o arquivo ainda é contíguo. Na figura, o vídeo para o quadro 1 é seguido diretamente pelas diversas trilhas de áudio do quadro 1 e então pelas várias trilhas de texto do quadro 1. Dependendo de quantas trilhas de áudio e texto houver, poderá ser mais simples apenas ler todas as partes de cada quadro em uma única operação de leitura de disco e transmitir somente as partes que o usuário precisa.

Essa organização requer uma E/S adicional de disco para leitura de áudio e texto que não sejam desejados e um espaço extra de buffer em memória para armazená-los. Contudo, ela elimina todas as buscas (em um sistema monousuário) e não requer custo extra para saber onde no disco está localizado determinado quadro, pois todo o filme é um arquivo contíguo. O acesso aleatório é impossível com esse esquema, mas, como ele não é necessário, sua perda não é grave. Da mesma maneira, o 'avanço rápido' e o 'retrocesso rápido' tomam-se impossíveis sem o acréscimo de estruturas de dados e complexidade.

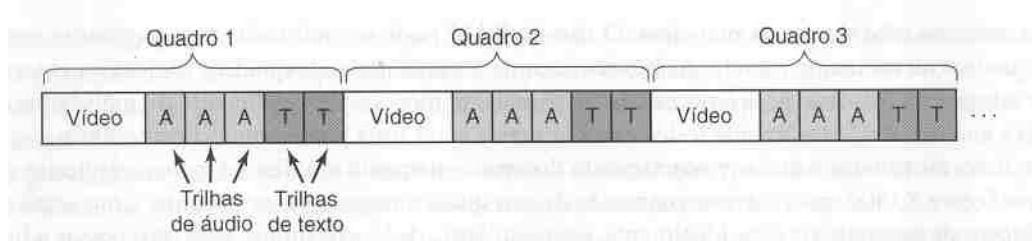


Figura: Intercalação de vídeo, áudio e texto em um único arquivo contínuo por filme

A vantagem de um filme inteiro como um único arquivo contíguo é perdida em um servidor de vídeo com vários fluxos concorrentes de saída, pois, depois de ler um quadro de um filme, os quadros de muitos outros filmes terão de ser lidos antes do primeiro ser lido novamente. Além disso, é difícil - e nada útil - implementar um sistema em que os filmes estejam sendo escritos e lidos (por exemplo, um sistema usado para produção ou edição de vídeo) usando enormes arquivos contíguos.

#### 14.6.2 DUAS ESTRATÉGIAS ALTERNATIVAS DE ORGANIZAÇÃO DE ARQUIVOS

Essas observações levam a duas outras organizações de alocação de arquivos multimídia. A primeira dessas organizações, o modelo de bloco pequeno, é ilustrada na Figura (a). Nela, o tamanho do bloco de disco é consideravelmente menor que o tamanho médio de um quadro. A idéia é ter uma estrutura de dados - o índice de quadros - por filme com uma entrada para cada quadro apontando para seu início. Cada quadro é constituído de todas as trilhas de vídeo, áudio e texto daquele quadro, dispostas em blocos de discos contíguos, conforme ilustrado. Dessa maneira, a leitura de  $k$  quadros consiste em encontrar a  $k^{\text{a}}$  entrada no índice de quadros e então ler o quadro inteiro em uma operação de disco. Como diferentes quadros têm tamanhos diferentes, é necessário que o tamanho do quadro (em blocos) esteja no índice, mas, mesmo que os blocos de disco fossem de 1 KB, um campo de 8 bits conseguiria tratar um quadro de até 255 KB que é o suficiente para um quadro NTSC sem compressão e até mesmo com muitas trilhas de áudio.

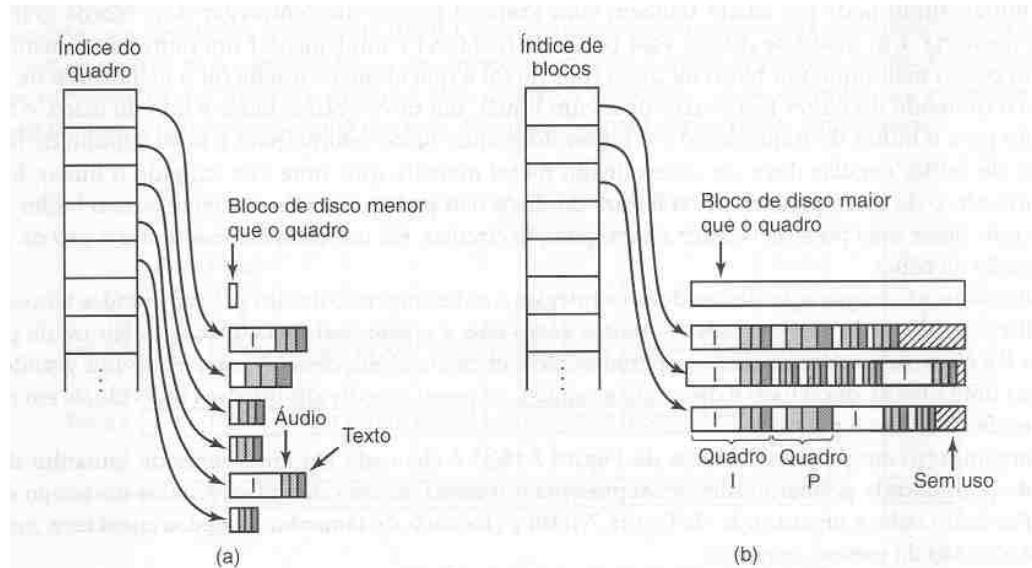


Figura: Armazenamento não-contíguo de filmes. (a) Pequenos blocos de disco. (b) Grandes blocos de disco.

Outra maneira de armazenar o filme é usar um bloco de disco grande (por exemplo, 256 KB) e colocar vários quadros em cada bloco, conforme ilustra a Figura (b). Ainda é necessário um índice, mas agora trata-se de um índice de blocos e não de um índice de quadros. Em geral, um bloco não conterá um número inteiro de quadros; portanto, algo deve ser feito para lidar com esse problema. Existem duas opções.

Na primeira opção, que é ilustrada na Figura (b), se o próximo quadro não couber no bloco atual, deixa-se o restante do bloco vazio. O espaço desperdiçado é a fragmentação interna, a mesma dos sistemas de memória virtual com páginas de tamanho fixo. Por outro lado, nunca será necessário fazer uma busca no meio de um quadro.

A outra opção é preencher cada bloco até o final, dividindo-se os quadros entre os blocos. Essa opção gera necessidade de buscas no meio dos quadros, o que pode prejudicar o desempenho, mas economiza espaço em disco, eliminando a fragmentação interna.

### 14.6.3 ALOCAÇÃO DE ARQUIVOS PARA VÍDEO QUASE SOB DEMANDA

Até agora temos estudado estratégias de alocação para vídeo sob demanda. Para vídeo quase sob demanda, há uma estratégia diferente de alocação que é mais eficiente. Lembre-se de que o mesmo filme está saindo em fluxos de vários estágios. Mesmo que um filme seja armazenado como um arquivo contíguo, é necessário buscar cada fluxo. Chen e Thapar (1997) inventaram uma estratégia de alocação de arquivos para eliminar quase todas as buscas. Sua aplicação é ilustrada na Figura abaixo, para um filme sendo reproduzido em 30 quadros/s com um novo fluxo se iniciando a cada cinco minutos. Com esses parâmetros, são necessários 24 fluxos concorrentes para um filme de duas horas.

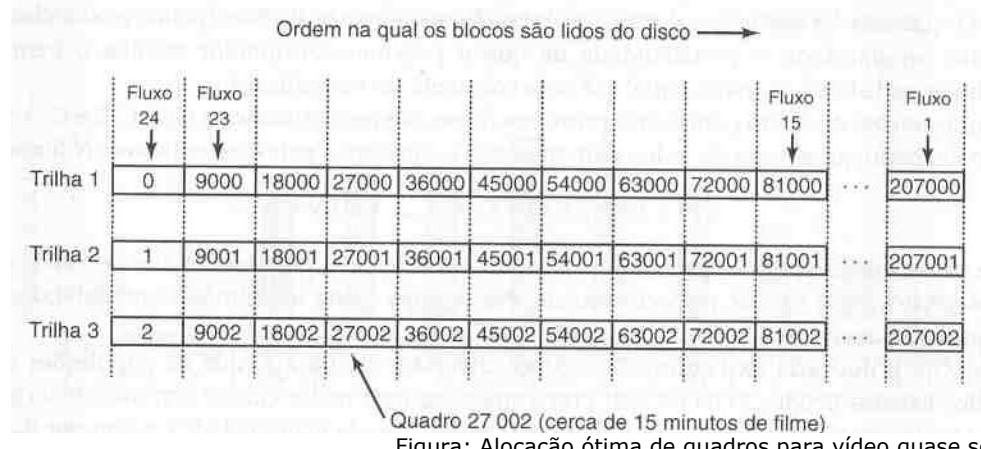


Figura: Alocação ótima de quadros para vídeo quase sob demanda.

Nessa alocação, conjuntos de 24 quadros são concatenados e escritos no disco como um único registro. Eles podem também ser lidos em uma só operação de leitura. Considere o instante em que o fluxo 24 tenha acabado de começar. O quadro 0 será necessário. O fluxo 23, que começou cinco minutos antes, precisará do quadro 9000. O fluxo 22 precisará do quadro 18000 e assim por diante até o fluxo 0, que precisará do quadro 207000. Dispondo esses quadros consecutivamente em uma trilha de disco, o servidor pode satisfazer todos os 24 fluxos em ordem inversa somente com uma busca (para o quadro 0). Claro que os quadros podem estar em ordem inversa no disco se houver algum motivo para servir os fluxos em ordem ascendente. Depois de servir o último fluxo, o braço do disco pode mover-se para a trilha 2 e preparar todo o serviço novamente. Esse esquema não exige que todo o arquivo seja contíguo, mas ainda permite um bom desempenho para vários fluxos simultâneos.

## 14.7 EXERCÍCIOS

- 205) Filmes, fotos e música digitais estão se tornando, cada vez mais, meios comuns de apresentar informação e entretenimento usando um computador. Para isso são utilizados arquivos de áudio e vídeo, contudo suas características são muito diferentes dos tradicionais arquivos de texto. Por quê?
- 206) Para viabilizar o vídeo sob demanda é necessária uma infra-estrutura especial. Quais os componentes dessa infra-estrutura?
- 207) Há duas características fundamentais que devem ser muito bem entendidas para trabalhar a multimídia adequadamente: (a) taxa de dados extremamente altas e (b) reprodução em tempo real. Fale sobre tais características.
- 208) Diferencie NTSC de PAL/M.
- 209) O que é *jitter*?

- 210) Fale brevemente sobre a codificação de áudio.
- 211) Vídeo em cores usa o mesmo padrão de varredura do monocromático, só que, em vez de mostrar a imagem com um feixe em movimento, são empregados três feixes. Fale sobre isso.
- 212) Porque é aceitável que o algoritmo de codificação de vídeo seja lento e que exija hardware sofisticado?
- 213) Descreva os passos para compressão JPEG.
- 214) Simule um escalonamento geral de tempo real, com os seguintes processos:

PROCESSO	Taxa de Execução	Tcpu/quadro
A	NTSC	10 ms
B	PAL	15 ms
C	NTSC	5 ms
D	20 vezes/s	20 ms

- 215) O algoritmo RMS pode ser usado para processo que satisfaçam quais condições?
- 216) Simule os escalonamentos RMS e EDF para os seguintes quadros:

PROCESSO	Taxa de Execução	Tcpu/quadro
A	NTSC	10 ms
B	PAL	15 ms
C	10 vezes/s	5 ms
D	20 vezes/s	20 ms

PROCESSO	Taxa de Execução	Tcpu/quadro
A	33 vezes/s	20 ms
B	25 vezes/s	15 ms
C	30 vezes/s	25 ms
D	20 vezes/s	30 ms

- 217) Diferencie “Servidores Push” de “Servidores Pull”
- 218) A maioria dos servidores de vídeo implementa também funções-padrão de controle VCR, inclusive pausa, avanço rápido e rebobinamento. Descreva o funcionamento dessas três funções.
- 219) Considere um padrão NTSC utilizando o vídeo quase sob demanda, com a transmissão iniciando a partir das 19:00 h e a cada 10 minutos. Quantos fluxos serão necessários para toda a transmissão? E para o padrão PAL/M?
- 220) Fale sobre as duas estratégias alternativas de organização de arquivos: o modelo do bloco pequeno e o do bloco grande.

-X-

# 15

## Segurança

*“É intelecto fraco, passarinho”, eu gritei,  
“ou é um verme duro nas suas entranhas tão pequenas”?  
(W. S. Gilbert)*

### 15.1 INTRODUÇÃO

À medida que a computação pessoal e o e-business se expandem, organizações armazenam informações altamente confidenciais em seus computadores e transmitem informações sensíveis pela Internet. Consumidores apresentam números de cartões de crédito a sites de e-commerce, e empresas expõem dados proprietários na Web. Ao mesmo tempo, organizações estão sofrendo um número cada vez maior de violações da segurança. Ambos, indivíduos e empresas, são vulneráveis a roubos de dados e a ataques que podem comprometer dados, corromper arquivos e provocar a queda de sistemas. A indústria da computação procura atender essas necessidades com suas organizações, trabalhando para melhorar a segurança da Internet e de redes. Por exemplo, a iniciativa “computação confiável” proposta pelo presidente da Microsoft, Bill Gates, é um esforço para concentrar a prioridade das empresas no fornecimento de aplicações confiáveis, disponíveis e seguras.

Segurança de computadores aborda a questão da prevenção do acesso não autorizado a recursos e informações mantidos em computadores. Sistemas de computador devem fornecer mecanismos para gerenciar ameaças à segurança originadas tanto externamente ao computador (via conexão de rede), quanto internamente (via usuários e softwares mal-intencionados). Segurança de computador comumente abrange garantir a privacidade e a integridade de dados sensíveis, restringir a utilização de recursos de computação e oferecer proteção contra tentativas de incapacitar o sistema. Proteção abrange mecanismos que protegem recursos como hardware e serviços de sistemas operacionais contra ataques. A segurança está se tornando rapidamente um dos tópicos mais ricos e desafiadores da computação; a segurança do sistema operacional está no âmago de um sistema de computação seguro.

### 15.2 CRIPTOGRAFIA

Uma meta importante da computação é disponibilizar quantidades maciças de dados com facilidade. Transmissão eletrônica de dados, especialmente por fiação pública, é inherentemente insegura. Uma solução para esse problema é tornar os dados ilegíveis, por meio de criptografia, para qualquer usuário não autorizado. Muitos mecanismos de segurança dependem da criptografia para proteger dados sensíveis, como senhas.

Criptografia trata da codificação e decodificação de dados de modo que eles somente possam ser interpretados pelos receptores pretendidos. Dados são transformados pela utilização de uma **cifra**, ou **sistema criptográfico** – um algoritmo matemático para criptografar mensagens. Uma **chave**, representada por uma cadeia de caracteres, é a entrada para a cifra. O algoritmo transforma os dados não criptografados, ou **texto comum**, em dados criptografados, ou **texto cifrado**, usando as chaves como entrada – chaves diferentes resultam em textos cifrados diferentes. O objetivo é fazer com que os dados fiquem incompreensíveis para quaisquer receptores não pretendidos (os que não possuem a chave de decriptação). Apenas os receptores pretendidos devem ter a chave para decriptar o texto cifrado, transformando-o em texto comum.

A criptografia é usada para proporcionar o seguinte:

- Confidencialidade. Para garantir que os dados permaneçam privados. Geralmente, a confidencialidade é obtida com a criptografia. Os algoritmos de criptografia (que usam chaves de criptografia) são usados para converter texto sem formatação em texto codificado e o algoritmo de descriptografia equivalente é usado para converter o texto codifi-

cado em texto sem formatação novamente. Os algoritmos de criptografia simétricos usam a mesma chave para a criptografia e a descriptografia, enquanto que os algoritmos assimétricos usam um par de chaves pública/privada.

- **Integridade de dados.** Para garantir que os dados sejam protegidos contra modificação accidental ou deliberada (mal-intencionada). A integridade, geralmente, é fornecida por códigos de autenticação de mensagem ou hashes. Um valor de hash é um valor numérico de comprimento fixo derivado de uma seqüência de dados. Os valores de hash são usados para verificar a integridade dos dados enviados por canais não seguros. O valor do hash de dados recebidos é comparado ao valor do hash dos dados, conforme eles foram enviados para determinar se foram alterados.

- **Autenticação.** Para garantir que os dados se originem de uma parte específica. Os certificados digitais são usados para fornecer autenticação. As assinaturas digitais geralmente são aplicadas a valores de hash, uma vez que eles são significativamente menores que os dados de origem que representam.

Sistemas criptográficos modernos dependem de algoritmos que operam nos bits individuais ou blocos (um grupo de bits) de dados, e não em letras do alfabeto. Chaves criptográficas e de decriptação são cadeias binárias com uma chave de tamanho determinado. Por exemplo, em sistemas criptográficos de 128 bits, o tamanho da chave é 128 bits. Chaves maiores têm mais força criptográfica; é preciso mais tempo e mais capacidade de computação para “quebrar” a cifra. Chaves maiores também exigem mais tempo de processamento para criptografar e decriptar dados, reduzindo o desempenho sistema. O crescimento das redes de computadores tornou mais desafiadora a permuta segura/desempenho.

### 15.2.1 CRIPTOGRAFIA POR CHAVE SECRETA

**Criptografia simétrica**, também conhecida como criptografia por chave secreta, usa a mesma chave secreta para criptografar e decriptar uma mensagem (Figura 15.1). Nesse caso, o emissor criptografa a mensagem utilizando a chave secreta, envia a mensagem criptografada ao receptor pretendido, que decifra a mensagem usando a mesma chave secreta. Uma limitação da criptografia por chave secreta é que, antes que as duas partes possam se comunicar com segurança, elas precisam encontrar um meio seguro de passar a chave secreta uma para a outra.

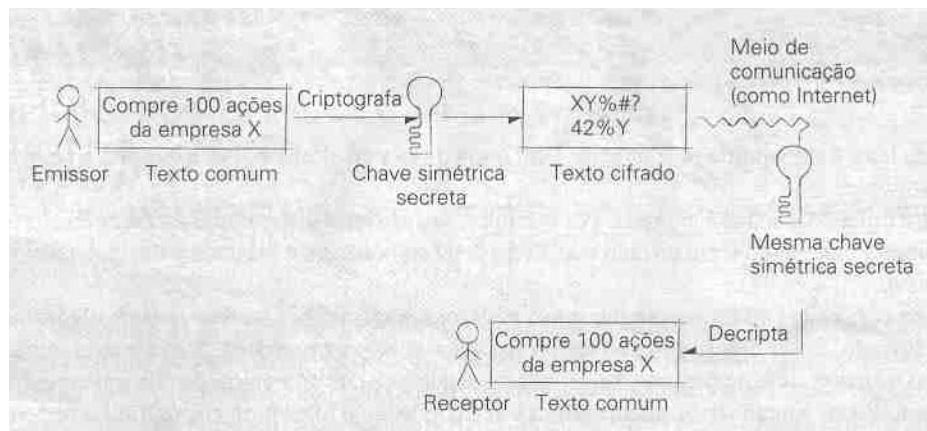


Figura 15.1: Criptografia e decriptação de uma mensagem usando uma chave secreta

Uma solução é a chave ser entregue por um *courier* ou outro serviço de correio. Embora essa abordagem talvez seja viável quando apenas duas partes se comunicam, não é fácil ampliá-la para grandes redes. Além do mais, a criptografia por chave secreta não pode ser considerada completamente segura porque a privacidade e a integridade da mensagem podem ser comprometidas se a chave for interceptada no caminho entre o emissor e o receptor. E, também, porque ambas as partes da transação usam a mesma chave para criptografar e decriptar uma mensagem, não é possível determinar qual das partes criou uma mensagem, o que habilita uma terceira parte, que finge ser uma das partes autorizadas, a criar uma mensagem após capturar a chave. Por fim, para manter privadas as comunicações, um emissor precisa de uma chave diferente para cada receptor. Conseqüentemente, para ter computação segura em grandes organizações, seria

preciso manter grandes números de chaves secretas para cada usuário, o que exigiria significativo armazenamento de dados.

Uma abordagem alternativa ao problema do intercâmbio de chaves é criar uma autoridade central denominada **central de distribuição de chaves** (*Key Distribution Center* - KDC), que compartilha uma chave secreta diferente com cada usuário da rede. Uma central de distribuição de chaves gera uma **chave de sessão** a ser usada para uma transação (Figura 15.2), então entrega ao emissor e ao receptor a chave da sessão criptografada com a chave secreta, que cada um compartilha com a central de distribuição de chaves.

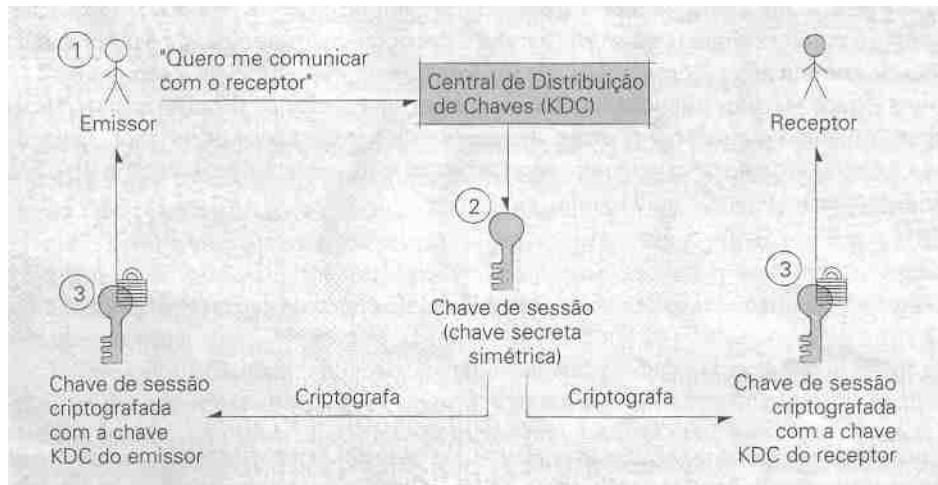


Figura 15.2: Distribuição de uma chave de sessão com uma Central de Distribuição de Chaves

Por exemplo, suponha que um comerciante e um cliente desejem realizar uma transação segura. Cada um compartilha a chave secreta exclusiva com a central de distribuição de chaves. A KDC gera uma chave de sessão para o comerciante e o cliente usarem na transação; ela envia ao comerciante a chave de sessão para a transação criptografada usando a chave reta que o comerciante já compartilha com a central. A KDC envia a mesma chave de sessão ao cliente para a transação criptografada usando a chave secreta que o cliente já compartilha com a KDC. Assim que o comerciante e o cliente obtêm a chave de sessão para a transação, eles podem se comunicar um com o outro criptografando suas mensagens por meio da chave de sessão compartilhada.

Uma central de distribuição de chaves reduz o número de entregas de chaves secretas por *courier* a cada usuário de a rede. Além disso, dá aos usuários uma nova chave secreta para cada comunicação com outros usuários da rede. Todavia, se a segurança da central de distribuição de chaves for comprometida, a segurança de toda a rede também estará comprometida.

Um algoritmo de criptografia simétrica comumente usado é o **padrão para criptografia de dados** (*Data Encryption Standard* - DES). Horst Feistel da IBM criou o **algoritmo de Lúcifer**, que foi escolhido como o DES pelo governo e pela Agência Nacional de Segurança (*National Security Agency* - NSA) dos Estados Unidos na década de 1970. O tamanho da chave do DES é de 56 bits, e o algoritmo criptografa dados em blocos de 64 bits. Esse tipo de criptografia é conhecido como **cifra de bloco**, porque cria grupos de bits por meio de uma mensagem e, então, aplica um algoritmo criptográfico ao bloco como um todo. Essa técnica reduz a quantidade de capacidade de processamento do computador e o tempo exigido para criptografar os dados.

Durante muitos anos, o DES foi o padrão de criptografia determinado pelo governo norte-americano e pelo *American National Standards Institute* (ANSI). Contudo, graças aos avanços da tecnologia e da velocidade de computação, o DES já não é mais considerado seguro - no final da década de 1990 foram construídas máquinas especializadas em quebrar DES que recuperavam as chaves do padrão para criptografia de dados (DES) após um período de várias horas. Consequentemente, o padrão para a criptografia simétrica foi substituído por **Triple DES** ou **3DES**, uma variante do DES que consiste, essencialmente, em três sistemas DES em série, cada um com uma chave secreta diferente que opera sobre um bloco. Embora o 3DES seja mais seguro, as três passagens pelo al-

goritmo DES aumentam a sobrecarga de criptografia, resultando em redução do desempenho.

Em outubro de 2000, o governo dos Estados Unidos selecionou um padrão mais seguro para criptografia simétrica para substituir o DES, denominado **padrão avançado de criptografia** (*Advanced Encryption Standard - AES*). O *National Institute of Standards and Technology* (NIST) - que estabelece os padrões criptográficos para o governo norte-americano - escolheu o **Rijndael** como o método de criptografia da AES. O Rijndael é uma cifra de bloco desenvolvida pela Dra. Joan Daemen e pelo Dr. Vincent Rijmen, da Bélgica. O Rijndael pode ser usado com chaves e blocos de 128, 192 ou 256 bits e foi escolhido como AES, concorrendo com outros quatro finalistas pelo seu alto nível de segurança, desempenho, eficiência e flexibilidade e baixo requisito de memória para sistemas computacionais.

### 15.2.2 CRIPTOGRAFIA POR CHAVE PÚBLICA

Em 1976, Whitfield Diffie e Martin Hellman, pesquisadores da Stanford University, desenvolveram a **criptografia por chave pública** para resolver o problema do intercâmbio seguro de chaves simétricas. A criptografia por chave pública é assimétrica no sentido de que emprega duas chaves inversamente relacionadas: uma **chave pública** e uma **chave privada**. A chave privada é mantida em segredo pelo seu proprietário, e a chave pública é distribuída livremente. Se a mensagem for cifrada com uma chave pública, somente a chave privada correspondente poderá decifrá-la (Figura 15.3) e vice-versa. Cada parte da transação possui a chave pública e a chave privada. Para transmitir uma mensagem com segurança, o emissor usa a chave pública do receptor para criptografar a mensagem. Então o receptor decifra a mensagem utilizando sua chave privada exclusiva. Admitindo que a chave privada seja mantida em segredo, a mensagem não pode ser lida por ninguém, exceto pelo receptor pretendido. Assim, o sistema garante a privacidade da mensagem.

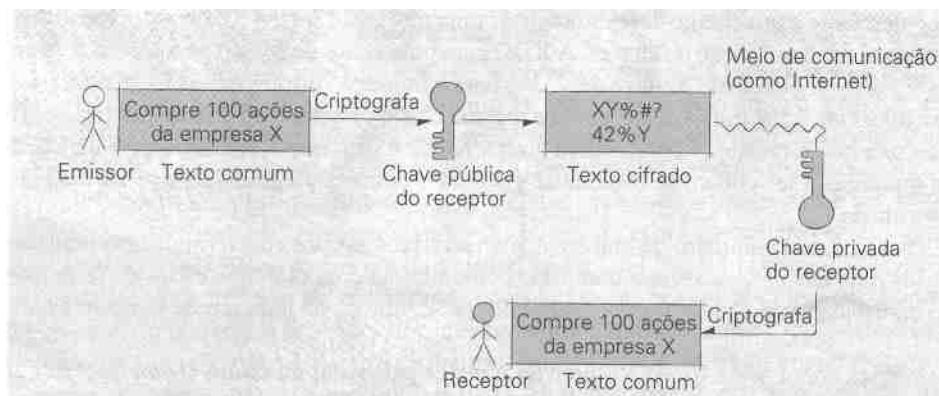


Figura 15.3: Criptografia e decriptação de uma mensagem usando chave pública

Algoritmos de chave pública seguros dependem de uma chave privada que, em termos computacionais, é quase impossível decifrar por meio da chave pública em um período de tempo razoável. Esses algoritmos são denominados funções de "uma via" ou "armadilha", porque cifrar a mensagem usando a chave pública requer pouca capacidade de computação, mas decifrá-la sem conhecer a chave privada exige capacidade computacional e tempo consideráveis. Um único computador levaria anos para decifrar um método criptográfico moderadamente seguro. [Nota: O concurso RC5 da Distributed.net oferece um prêmio em dinheiro para quem decifrar uma mensagem criptografada por meio da função de uma via. O concurso atrai milhares de usuários que trabalham em paralelo para descobrir a chave. Recentemente, cerca de 300 mil participantes dedicaram capacidade computacional para descobrir uma mensagem cifrada usando uma chave de 64 bits. A chave foi descoberta em agosto de 2002, após 1.757 dias de computação equivalente a 46 mil processadores de 2 GHz trabalhando em paralelo. O computador mais potente na época era o Earth Simulator, que continha 5.120 processadores de 500 MHz.]

A segurança do sistema depende do segredo das chaves privadas. Portanto, se um terceiro obtiver a chave privada usada na criptografia, a segurança de todo o sistema estará comprometida. Se o sistema ficar comprometido, o usuário pode simplesmente trocar a chave, em vez de trocar todo o algoritmo criptográfico ou de decriptação.

Tanto a chave pública quanto a privada podem ser usadas para criptografar ou decriptar uma mensagem. Por exemplo, se um cliente usar a chave pública de um comerciante para criptografar uma mensagem, só o comerciante poderá decifrá-la, com a sua própria chave privada. Assim, a identidade do comerciante pode ser autenticada, porque somente ele conhece a chave privada. Entretanto, o comerciante não pode validar a identidade do cliente porque a chave criptográfica que o cliente usou está disponível publicamente.

Se a chave de decriptação for a chave pública do emissor e a chave criptográfica for a chave privada do emissor, o emissor da mensagem poderá ser autenticado. Por exemplo, suponha que um cliente envie a um comerciante uma mensagem criptografada usando a chave privada do cliente. O comerciante decripta a mensagem por meio da chave pública do cliente. Porque o cliente criptografou a mensagem usando a sua própria chave privada, o comerciante poderá ter certeza da identidade do cliente. Esse processo autentica o emissor, mas não garante confidencialidade, pois qualquer terceiro poderia decifrar a mensagem com a chave pública do emissor. O problema de provar a propriedade de uma chave pública será discutido posteriormente.

Esses dois métodos de criptografia de chave pública podem ser combinados para autenticar ambos os participantes de uma comunicação (Figura 15.4). Suponha que um comerciante deseje enviar uma mensagem com segurança a um cliente, de modo que somente o cliente possa ler a mensagem, e também provar ao cliente que foi ele, o comerciante, quem enviou a mensagem. Primeiro, o comerciante criptografa a mensagem usando a chave pública do cliente. Essa etapa garante que somente o cliente pode ler a mensagem. Então o comerciante criptografa o resultado por meio de sua própria chave privada, o que comprova a sua identidade. O cliente decripta a mensagem em ordem inversa. Primeiro, ele utiliza a chave pública do comerciante. Porque somente o comerciante poderia ter criptografado a mensagem com a chave privada inversamente relacionada, essa etapa autentica o comerciante. O cliente então usa sua própria chave privada para decriptar o próximo nível da criptografia. Essa etapa assegura que o conteúdo da mensagem continuou privado durante a transmissão, pois somente o cliente tem a chave para decriptar a mensagem.

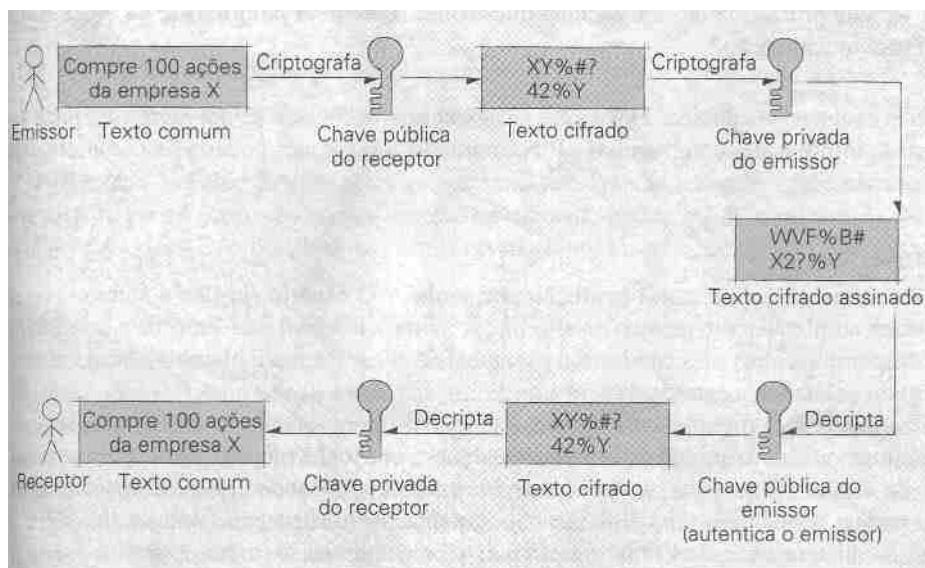


Figura 15.4: Autenticação com um algoritmo de chave pública

O algoritmo de chave pública mais comumente usado é o **RSA**, um sistema criptográfico desenvolvido em 1977. Seus produtos criptográficos estão inseridos em centenas de milhões de cópias das aplicações mais populares da Internet incluindo navegadores Web, servidores comerciais e sistemas de e-mail. A maioria das transações seguras de e-commerce e das comunicações pela Internet usa produtos RSA.

### 15.3 AUTENTICAÇÃO

Identificar usuários e as ações que eles têm permissão de realizar é vital para manter a segurança de um sistema computacional. Um usuário pode ser identificado por:

- Uma característica exclusiva da pessoa (impressões digitais, impressões vocais, varredura de retina e assinaturas);
- Propriedades de um item (crachás, carteiras de identidade, chaves e cartões inteligentes); e
- Conhecimento do usuário (senhas, números de identificação pessoal (PINs) e combinações de travas).

### 15.3.1 AUTENTICAÇÃO BÁSICA

O esquema de autenticação mais comum é uma simples proteção por senha. O usuário escolhe e memoriza uma senha e a registra no sistema para obter admissão a um recurso ou sistema. A maioria dos sistemas suprime a exibição da senha na tela, substituindo seu texto por caracteres mascaradores ou ocultando a entrada da senha.

A proteção por senha introduz diversos pontos fracos em um sistema seguro. Usuários tendem a escolher senhas fáceis de lembrar, como o nome de um cônjuge ou de um animal de estimação. Alguém que tenha obtido informações pessoais do usuário poderia tentar o acesso ao sistema (login) várias vezes, usando senhas correspondentes às características do usuário; várias tentativas repetidas podem resultar em uma violação da segurança. Alguns sistemas antigos limitavam o usuário a senhas curtas; esses eram facilmente comprometidos simplesmente pela tentativa de todas as senhas possíveis - uma técnica conhecida por **quebra por força bruta**.

A maioria dos sistemas de hoje exige senhas mais longas que incluem caracteres alfabéticos e numéricos para frustrar as tentativas de invasão. Alguns sistemas até mesmo proíbem a utilização de palavras do dicionário como valores de senhas. Contudo, senhas longas não melhoraram necessariamente a segurança de um sistema; se elas forem difíceis de lembrar os usuários ficarão mais inclinados a anotá-las, facilitando a obtenção da senha por um intruso.

A invasão das defesas de um sistema operacional não precisa resultar necessariamente em um comprometimento significativo da segurança. Por exemplo, suponha que um intruso consiga obter acesso à lista mestra de senhas de um sistema. Se o arquivo de senhas estivesse armazenado em texto comum, essa invasão permitiria ao intruso o acesso a qualquer informação do sistema, assumindo a identidade de qualquer usuário. Para reduzir a efetividade de um arquivo de senhas roubado, muitos sistemas operacionais criptografam o arquivo de senhas ou armazenam apenas valores de hash para cada senha. Nesse caso, uma cópia do arquivo de senhas é de pouca utilidade, a menos que o intruso possa decriptá-las. Para frustrar ainda mais as tentativas de obter senhas, vários sistemas empregam o salpicamento de senha; uma técnica que insere caracteres em várias posições da senha antes de criptografá-la (Tabela 15.1). Note como uma pequena quantidade de salpicamento pode alterar significativamente um texto cifrado, mesmo quando se estiver utilizando uma cifra fraca, como uma cifra de substituição (codificação base 64). O salpicamento pode evitar que intrusos obtenham uma chave criptográfica com base em padrões produzidos por senhas comuns após a criptografia.

Texto Comum	Texto Cifrado
password	cGFzc3dvcmQ=
psassawlortd	cHNhc2Fzd2xvcnRK
newpassword	bmV3cGFzc3dvcmQ=
nsewaplatssewodrd	bnN1d2FwbGFOc3N1d29kcmQ=

Tabela 15.1: Salpicando senhas (codificação base 64)

Usuários são incentivados a trocar suas senhas freqüentemente; mesmo que um intruso obtenha uma senha, ela pode ser trocada antes que algum dano real seja causado. Alguns sistemas exigem que os usuários escolham novas senhas periodicamente. Infelizmente, alguns usuários reutilizarão duas ou três senhas ciclicamente, o que reduz a

segurança geral. Como resposta, vários sistemas de autenticação proíbem a reutilização das senhas escolhidas mais recentemente por um usuário.

Uma defesa simples contra a quebra por força bruta e tentativas repetidas de entrar a senha é limitar o número de tentativas de acesso ao sistema que podem ser iniciadas em um certo período de tempo de um único terminal ou estação de trabalho (ou de uma única conta). É claro que as pessoas cometem erros ao tentarem obter acesso ao sistema, mas não é razoável que alguém que saiba a senha correta tenha de fazer dezenas, centenas ou milhares de tentativas. Portanto, um sistema pode permitir três ou quatro tentativas e, então, desconectar o terminal durante vários segundos ou minutos. Após um período de espera, o terminal pode ser reconectado.

### **15.3.2 BIOMETRIA E CARTÕES INTELIGENTES**

Uma inovação na segurança que era limitada aos filmes, mas que está se tornando cada vez mais comum nos sistemas seguros atuais, é a biometria. A biometria utiliza informações pessoais exclusivas - como impressões digitais, varreduras da íris ou varreduras da face - para identificar um usuário. O número de senhas que um usuário médio tem de lembrar aumentou em razão da proliferação de dados sensíveis transmitidos por canais não seguros. O resultado é que as senhas se tornaram uma carga cada vez mais pesada para a computação segura. Essa tendência pode ser prejudicial à segurança, pois usuários registram senhas no papel ou empregam a mesma senha para diversas contas. Conseqüentemente, a biometria tornou-se uma alternativa atraente para as senhas, e o custo dos dispositivos biométricos caiu de modo significativo.

Dispositivos de varredura de impressões digitais, varredura da face e varredura da íris estão substituindo a utilização de senhas para entrar em sistemas, verificar e-mail ou ter acesso a informações seguras em uma rede. As impressões digitais, a varredura da face ou a varredura da íris de cada usuário são armazenadas em um banco de dados seguro. Cada vez que o usuário entra no sistema, a sua varredura é comparada com o banco de dados. Se houver correspondência, o acesso é bem-sucedido.

Embora as senhas sejam correntemente o meio predominante de autenticação em sistemas de computador e e-commerce, várias plataformas adotaram a biometria. Em 2000, a Microsoft anunciou a *Biometric Application Programming Interface* (BAPI) incluída nos seus sistemas operacionais Windows 2000 e Windows XP para simplificar a integração da biometria em sistemas pessoais e corporativos.

Um cartão inteligente (*smart card*), projetado para parecer um cartão de crédito, normalmente é usado para autenticação e armazenar dados. Os cartões inteligentes mais populares são os de memória e os microprocessadores. Cartões de memória podem armazenar e transferir dados; cartões microprocessadores contêm componentes de computador gerenciados por um sistema operacional que fornecem segurança e armazenamento. Cartões inteligentes também são caracterizados por sua interface com dispositivos de leitura. Uma delas é a interface de contato, pela qual os cartões inteligentes são inseridos em um dispositivo de leitura que requer o contato físico entre o dispositivo e o cartão para transferir dados. Como alternativa, uma interface sem contato permite que dados sejam transferidos sem contato físico entre a leitora e o cartão, quase sempre realizado por um dispositivo sem fio embutido no cartão.

Cartões inteligentes podem armazenar chaves privadas, certificados digitais e outras informações. Também podem armazenar números de cartões de crédito, informações pessoais para contato e coisas semelhantes. Cada cartão inteligente pode ser usado em combinação com um PIN. Essa característica fornece dois níveis de segurança, pois requer que o usuário possua um cartão inteligente e conheça o PIN correspondente para ter acesso às informações armazenadas no cartão. Para reforçar ainda mais a segurança, alguns cartões microprocessadores apagarão ou corromperão dados armazenados caso haja tentativa de violação do cartão.

**Autenticação de dois fatores** emprega dois meios para autenticar o usuário, como biometria ou um cartão inteligente combinado com uma senha. Embora esse sistema possa ser potencialmente comprometido, usar dois métodos de autenticação normalmente proporciona melhor segurança do que apenas senhas.

### 15.3.3 KERBEROS

Ataques internos a computadores (ataques que se originam de um usuário válido) são comuns e podem ser extremamente danosos. Por exemplo, funcionários descontentes com acesso à rede podem desabilitar a rede de uma organização ou roubar informações proprietárias valiosas. Estima-se que 70% a 90% dos ataques a redes corporativas são internos. Um sistema de autenticação centralizado, seguro, pode facilitar uma reação rápida a esses ataques à segurança. **Kerberos**, um protocolo de código-fonte aberto, desenvolvido no MIT, pode fornecer proteção contra ataques internos à segurança; emprega criptografia de chave secreta para autenticar usuários de uma rede e manter a integridade e a privacidade das comunicações da rede.

No Kerberos, a autenticação é tratada por um servidor de autenticação e um serviço secundário de **concessão de bilhetes de entrada** (*Ticket Granting Service* - TGS). Esse sistema é semelhante às centrais de distribuição descritas na Seção 15.2.1, "Criptografia por chave secreta". O servidor de autenticação autentica a identidade de um cliente para o TGS; o TGS autentica os direitos de acesso do cliente a serviços específicos de rede.

Cada cliente da rede compartilha uma chave secreta com o sistema Kerberos. Essa chave secreta pode ser armazenada por vários TGSs do sistema Kerberos. O sistema Kerberos funciona da seguinte maneira:

1. O cliente começa apresentando um nome de usuário e uma senha ao servidor de autenticação Kerberos.
2. O servidor de autenticação mantém um banco de dados de todos os clientes da rede. Se o nome e a senha forem válidos, o servidor de autenticação retomará um bilhete de concessão de entrada (*Ticket-Granting Ticket* - TGT) criptografado com a chave secreta do cliente. Porque a chave secreta é conhecida unicamente pelo servidor de autenticação e pelo cliente, apenas o cliente pode decriptar o TGT, autenticando, assim, a identidade do cliente.
3. O cliente envia o TGT decriptado ao serviço de concessão de bilhetes de entrada e requisita um bilhete de serviço. O bilhete de serviço autoriza o acesso do cliente a serviços específicos de rede. Bilhetes de serviço têm um prazo de expiração designado e podem ser renovados ou revogados pelo TGS. Se o TGT for válido, o TGS emitirá um bilhete de serviço criptografado com a chave de sessão do cliente.
4. O cliente então decripta o bilhete de serviço e o apresenta para obter acesso aos recursos da rede.

### 15.3.4 ASSINATURA ÚNICA

Sistemas de **assinatura única** simplificam o processo de autenticação, permitindo que o usuário acesse o sistema somente uma vez, usando uma única senha. Usuários autenticados via sistema de assinatura única, então, podem ter acesso a várias aplicações em vários computadores. Senhas de assinatura única devem ser guardadas com muita segurança porque, se intrusos se apossarem de uma senha, todas as aplicações protegidas por aquela senha podem ser acessadas e atacadas.

**Scripts de acesso de estações de trabalho** são as formas mais simples de assinatura única. Usuários entram no sistema por suas estações de trabalho e escolhem aplicações em um menu. O script de acesso envia a senha do usuário aos servidores de aplicação, autenticando o usuário para futuro acesso a essas aplicações. Muitos scripts de acesso de estações de trabalho não proporcionam um nível de segurança suficiente porque as senhas de usuários em geral são armazenadas no computador cliente em texto comum. Mesmo que o script use uma criptografia simples para a senha, aquele algoritmo deve estar presente no sistema, o que significa que qualquer intruso que conseguir acesso ao computador poderá comprometer a criptografia.

**Scripts de servidor de autenticação** autenticam usuários via servidor central. O servidor central controla conexões entre o usuário e as aplicações que o usuário deseja acessar. Scripts de servidor de autenticação são mais seguros do que scripts de acesso de estações de trabalho, pois as senhas são mantidas no servidor, que geralmente é

mais seguro do que o computador cliente. Contudo, se a segurança do servidor for comprometida, a segurança do sistema inteiro também será.

Os sistemas mais avançados de assinatura única empregam **autenticação por ficha (token)**. Quando um usuário é autenticado, é emitida uma única **ficha** que habilita o usuário a acessar aplicações específicas. A segurança do acesso ao sistema (login) que cria a ficha é garantida por criptografia ou senha única. Por exemplo, o Kerberos usa autenticação por ficha na qual o bilhete de serviço age como a ficha. Um problema fundamental da autenticação por ficha é que as aplicações devem ser modificadas para aceitar fichas em vez das tradicionais senhas de acesso ao sistema.

Correntemente, os três líderes do desenvolvimento da tecnologia de assinatura única são o Liberty Alliance Project, a Microsoft e a Novell. O Liberty é um consórcio de organizações de tecnologia e segurança que trabalham para criar uma solução de assinatura única aberta. O .NET Passport da Microsoft e o SecureLogin da Novell também são soluções viáveis, embora sejam proprietárias. Para proteger a privacidade de informações regidas por assinatura única e outras aplicações, a **plataforma para preferências privadas (Platform for Privacy Preferences - P3P)** permite que os usuários controlem as informações pessoais que os sites coletam.

## 15.4 CONTROLE DE ACESSO

Na qualidade de gerenciador de recursos, o sistema operacional deve se defender cuidadosamente contra utilizações não intencionais e maliciosas de recursos de computação. Conseqüentemente, hoje os sistemas operacionais são projetados para proteger serviços de sistema operacional e informações sensíveis contra usuários e/ou softwares que obtiveram acesso aos recursos do computador. Direitos de acesso protegem recursos e serviços do sistema contra usuários potencialmente perigosos, restringindo ou limitando as ações que podem ser executadas no recurso. Esses direitos normalmente são gerenciados por listas de controle de acesso ou listas de capacidades.

### 15.4.1 DIREITO DE ACESSO E DOMÍNIOS DE PROTEÇÃO

A chave para a segurança do sistema operacional é controlar o acesso a dados e recursos internos. **Direitos de acesso** definem como vários sujeitos podem acessar vários objetos. Sujeitos podem ser usuários, processos, programas ou outras entidades. Objetos são recursos como hardware, software e dados; podem ser objetos físicos como discos, processadores ou memória principal. Também podem ser objetos abstratos como estruturas de dados, processos ou serviços. Sujeitos também podem ser objetos do sistema; um sujeito pode ter direitos de acesso a um outro. Sujeitos são entidades ativas; objetos são entidades passivas. À medida que um sistema executa, sua população de sujeitos e objetos tende a mudar. A maneira pela qual um sujeito pode ter acesso a um objeto é denominada **privilegio** e pode incluir leitura, escrita e impressão.

Objetos devem ser protegidos contra sujeitos. Caso fosse permitido que um processo tivesse acesso a todos os recursos de um sistema, um usuário poderia inadvertidamente ou mal-intencionadamente, comprometer a segurança do sistema ou causar a queda de outros programas. Para impedir que tais eventos ocorram, cada sujeito deve obter autorização para acessar objetos dentro de um sistema.

Um **domínio de proteção** é uma coleção de direitos de acesso. Cada direito de acesso de um domínio de proteção é representado como um par ordenado com campos para o nome do objeto e seus privilégios correspondentes. Um domínio de proteção é exclusivo de um sujeito. Por exemplo, se um usuário puder ler e escrever para o arquivo exemplo.txt, o par ordenado correspondente para o direito de acesso desse usuário poderá ser representado por <example.txt, {read, write}>.

Os direitos de acesso mais comuns são ler, escrever e executar. Alguns sujeitos também podem conceder direitos de acesso a outros sujeitos. Na maioria dos sistemas de computação, o administrador possui todos os direitos de acesso e é responsável pelo gerenciamento dos direitos de outros usuários.

Direitos de acesso podem ser copiados, transferidos ou propagados de um domínio para outro. Copiar um direito de acesso implica simplesmente conceder um direito de um

usuário a outro usuário. Quando um direito de acesso é transferido do sujeito A para o sujeito B, o direito de acesso do sujeito A é revogado quando a transferência for concluída. Propagar um direito de acesso é similar a copiar um direito de acesso; contudo, além de compartilhar o direito de acesso original, ambos os sujeitos também podem copiar o direito para outros sujeitos.

Quando um sujeito não precisa mais do acesso a um objeto, os direitos de acesso podem ser revogados. Nesse caso surgem várias questões - a revogação deve ser imediata ou tardia? A revogação deve se aplicar a todos os objetos ou a uns poucos selecionados? A revogação deve-se aplicar a sujeitos específicos ou a um domínio inteiro? A revogação deve ser permanente ou temporária? Cada implementação de gerenciamento de direitos de acesso aborda a revogação de maneira diferente

#### **15.4.2 MODELOS E POLÍTICAS DE CONTROLE DE ACESSO**

Controle de acesso pode ser dividido em três níveis conceituais: modelos, políticas e mecanismos. Um **modelo de segurança** define os sujeitos, objetos e privilégios de um sistema. Uma **política de segurança**, que normalmente é especificada pelo usuário e/ou administrador do sistema, define quais privilégios a objetos são designados a sujeitos. O **mecanismo de segurança** é o método pelo qual o sistema implementa a política de segurança. Em muitos sistemas, a política muda ao longo do tempo à medida que mudam o conjunto de recursos e os usuários do sistema, mas o modelo de segurança e os mecanismos que implementam o controle de acesso não requerem modificação, portanto, a política de segurança é separada do mecanismo e do modelo.

Um modelo de segurança popular organiza usuários em classes. Uma desvantagem é que os direitos de acesso são armazenados em cada arquivo e especificam um único proprietário e grupo, portanto, no máximo um grupo pode acessar um arquivo particular. Além disso, o sistema poderia precisar modificar permissões de grupo para diversos arquivos ao designar novos direitos de acesso a um grupo de usuários - Um processo demorado e sujeito a erros.

No **modelo de controle de acesso por função** (*Role-based Access Control - RBAC*) são designadas **funções** aos usuários, cada uma representando normalmente um conjunto de tarefas designadas a um membro de uma organização. A cada função é designado um conjunto de privilégios que definem os objetos que os usuários podem acessar em cada função. Usuários podem pertencer a várias funções; os administradores precisam apenas modificar permissões para uma única função para alterar os direitos de acesso de um grupo de usuários. A atratividade do RBAC é que ele designa relações significativas entre sujeitos e objetos que não são limitadas por classes como proprietários e grupos.

Considere, por exemplo, um sistema formado por computadores de uma universidade com as seguintes funções: os professores da universidade criam e dão notas aos trabalhos escolares, estudantes apresentam seus trabalhos concluídos e o pessoal administrativo transfere as notas para os históricos dos estudantes. Sob o modelo RBAC esse sistema consiste em três funções (estudantes, professores e pessoal administrativo), dois objetos (trabalhos escolares e notas) e três permissões (ler, modificar e criar). Nesse exemplo, os professores têm permissão para criar, ler e modificar trabalhos escolares e notas; estudantes têm permissão para ler e modificar cópias de seus trabalhos e ler notas; o pessoal administrativo tem permissão de ler e modificar notas.

Embora as políticas de segurança variem para atender às necessidades dos usuários de um sistema, a maioria delas incorpora o princípio do mínimo privilégio - um sujeito recebe permissão de acesso somente aos objetos de que necessita para executar suas tarefas. Políticas também podem implementar controle de acesso discricionário ou obrigatório, dependendo das necessidades de segurança do ambiente. A maioria dos sistemas baseados em UNIX segue o modelo de **controle de acesso discricionário** (*Discretionary Access Control - DAC*), segundo o qual o criador de um objeto controla as permissões para aquele objeto. As políticas de **controle de acesso obrigatório** (*Mandatory Access Control - MAC*) definem previamente um esquema de permissão central pelo qual todos os sujeitos e objetos são controlados. O MAC é encontrado em muitas instalações de alta segurança, como sistemas governamentais confidenciais.

### 15.4.3 MECANISMOS DE CONTROLE DE ACESSO

Nesta seção, discutiremos várias técnicas que um sistema operacional pode empregar para gerenciar direitos de acesso. Matrizes de controle de acesso compatibilizam sujeitos e objetos com os direitos de acesso adequados. O conceito que fundamenta o modelo é simples; contudo, a maioria dos sistemas contém muitos sujeitos e objetos, resultando em uma matriz de grande porte que é um meio ineficiente de controle de acesso. Listas de controle de acesso e listas de capacidades são derivadas do princípio de mínimo privilégio, e muitas vezes são métodos mais eficientes e flexíveis de gerenciar direitos de acesso.

#### Matrizes de Controle de Acesso

Um modo de gerenciar direitos de acesso é através de uma **matriz de controle de acesso**. Os vários sujeitos são listados nas linhas, e os objetos aos quais requerem acesso são listados nas colunas. Cada célula da matriz especifica as ações que um sujeito (definido pela linha) pode executar sobre um objeto (definido pela coluna). Em uma matriz de controle de acesso, os direitos de acesso são concedidos com base no mínimo privilégio, portanto, se um direito de acesso não estiver descrito explicitamente na matriz, o usuário não terá nenhum direito de acesso ao objeto.

Porque uma matriz de controle de acesso coloca todas as informações de permissão em uma localização central, ela deve ser uma das entidades mais bem protegidas de um sistema operacional. Se a matriz for comprometida, quaisquer recursos que eram protegidos pelos direitos de acesso definidos na matriz também serão suscetíveis a ataques.

A matriz de controle de acesso da Figura 15.6 representa os direitos de acesso dos usuários (André, Bruna, Celina, Geraldo e Convidado) aos objetos (Arquivo A, Arquivo B e Impressora). Os privilégios que um usuário pode obter para um objeto são ler, escrever e imprimir. Os direitos de acesso a ler e escrever aplicam-se somente aos arquivos do sistema, nesse caso, Arquivo A e Arquivo B. Em alguns ambientes, nem todos os usuários têm acesso à Impressora - um usuário tem de ter o privilégio explícito de imprimir para poder enviar conteúdo à Impressora. Qualquer direito de acesso marcado por um asterisco (\*) pode ser copiado de um usuário para outro. Nessa matriz de controle de acesso, André tem todos os direitos de acesso, bem como a capacidade de atribuir esses direitos a outros usuários. Geraldo não pode acessar o Arquivo A porque não há nenhuma entrada na célula correspondente na matriz. A conta Convidado não contém nenhum direito de acesso por *default*. Um Convidado pode acessar recursos somente quando o direito for explicitamente concedido por um outro usuário. Embora gerar e interpretar uma matriz de controle de acesso seja uma operação direta, a matriz pode tornar-se grande e esparsamente povoada.

	Arquivo A	Arquivo B	Impressora
André	Ler* Escrever	Ler* Escrever*	Imprimir*
Bruna	Ler* Escrever	Ler* Escrever	Imprimir
Celina	Ler		Imprimir
Geraldo		Ler	
Convidado			

Figura 15.6: Matriz de controle de acesso para um pequeno grupo de sujeitos e objetos

#### Listas de Controle de Acesso

Uma **lista de controle de acesso** armazena os mesmos dados que uma matriz de controle de acesso, mas mantém um registro somente das entradas que especificam um direito de acesso. A lista de controle de acesso de um sistema pode ser baseada nas linhas (os sujeitos) ou nas colunas (os objetos) de uma matriz. Para cada objeto de um sistema operacional, uma lista de controle de acesso contém entradas para cada sujeito e os privilégios associados àquele sujeito em relação àquele objeto. Quando um sujeito tenta acessar um objeto, o sistema procura pela lista de controle de acesso para aquele objeto para identificar os privilégios daquele sujeito.

O problema desse método está na ineficiência com a qual o sistema operacional determina os privilégios de usuário para determinado objeto. A lista de controle de acesso para cada objeto contém uma entrada para cada sujeito com privilégios para aquele objeto - uma lista potencialmente grande. Toda vez que um objeto for acessado, o sistema deve pesquisar a lista de sujeitos para encontrar os privilégios adequados. Quando se usam listas de controle de acesso, é difícil determinar quais direitos de acesso pertencem a um certo domínio de proteção; é preciso pesquisar a lista de acesso para cada objeto, procurando entradas relativas àquele objeto particular.

A lista de controle de acesso da Figura 15.7 representa um conjunto de direitos de acesso que foram estabelecidos na matriz de controle de acesso da Figura 15.6. A implementação é menor porque as entradas vazias da matriz não estão presentes. Se um objeto não contiver uma entrada para um usuário particular, esse usuário não terá nenhum privilégio para o objeto.

```

1 Arquivo A:
2 <André, {ler*, escrever*}>
3 <Bruna, {ler*, escrever}>
4 <Celina, {ler}>
5 Arquivo B:
6 <André, {ler*, escrever*}>
7 <Bruna, {ler*, escrever}>
8 <Geraldo, {ler}>
9 Impressora:
10 <André, {imprimir*}>
11 <Bruna, {imprimir}>
12 <Celina, {imprimir}>
```

Figura 15.7: Lista de controle de acesso derivada da matriz de controle de acesso

### Listas de Capacidades

Uma capacidade é um ponteiro, ou ficha, que concede privilégios a um sujeito que a possui. É análoga a um bilhete de entrada usado para obter acesso a um evento esportivo. Capacidades normalmente não são modificadas, mas podem ser reproduzidas. Lembre-se de que um domínio de proteção define o conjunto de privilégios entre sujeitos e objetos. Como alternativa, pode-se definir o domínio de proteção como o conjunto de capacidades pertencentes a um sujeito.

Uma capacidade freqüentemente é implementada como um identificador exclusivo de um objeto. Capacidades são concedidas a um sujeito, o qual apresenta a ficha para todos os acessos subsequentes ao objeto. Capacidades são criadas por rotinas de sistemas operacionais cuidadosamente protegidas. Um sujeito que possua uma capacidade pode realizar certas operações, entre elas criar cópias da capacidade ou passá-la como um parâmetro.

Quando um objeto é criado, também é criada uma capacidade para esse objeto. Essa capacidade original inclui privilégios totais para o novo objeto. O sujeito que cria o objeto pode passar cópias da capacidade a outros sujeitos. Da mesma maneira, um sujeito que recebe a capacidade pode usá-la para acessar o objeto ou pode criar cópias adicionais e passá-las para outros sujeitos. Quando um sujeito passa uma capacidade para um outro sujeito, pode reduzir os privilégios associados. Assim, à medida que uma capacidade se propaga pelo sistema, o tamanho do seu conjunto de privilégios pode permanecer o mesmo ou ser reduzido.

Usuários devem ser impedidos de criar capacidades arbitrariamente, o que pode ser conseguido armazenando capacidades em segmentos que os processos usuários não podem acessar.

O identificador de uma capacidade pode ser implementado como um ponteiro para o objeto desejado ou pode ser uma seqüência exclusiva de bits (uma ficha). Ponteiros simplificam o acesso ao endereço no qual o objeto está armazenado, mas, se o objeto for movimentado, todos os ponteiros desse tipo do sistema devem ser atualizados, o que pode degradar o desempenho. Quando se usam fichas, as capacidades não dependem da

localização do objeto na memória. Entretanto, porque uma ficha não especifica a localização de seu objeto correspondente, as fichas exigem que o endereço do objeto seja determinado quando a capacidade for usada pela primeira vez. Um mecanismo de hash implementa eficientemente capacidades por ficha; caches de alta velocidade em geral reduzem a sobrecarga das referências repetidas ao mesmo objeto.

Sistemas que empregam capacidades podem sofrer o problema do 'objeto perdido'. Se a última capacidade remanescente para um objeto for destruída, o objeto associado não poderá mais ser acessado. Para evitar esse problema, muitos sistemas operacionais garantem que o sistema mantenha sempre, no mínimo, uma capacidade para cada objeto.

Controlar a propagação de capacidades é um problema difícil. Em geral, os sistemas não permitem manipulação direta de capacidades pelos usuários; a manipulação da capacidade é executada pelo sistema operacional em nome dos usuários. Monitorar capacidades é uma tarefa importante que se torna difícil em sistemas multiusuários que contêm um grande número de capacidades. Muitos sistemas empregam uma estrutura de diretório para gerenciar suas capacidades.

## 15.5 ATAQUES À SEGURANÇA

Recentes ciberataques contra empresas de comércio eletrônico foram notícias de primeira página em jornais do mundo inteiro. Ataques de recusa de serviço (*Denial-of-Service - DoS*), vírus e vermes têm custado bilhões de dólares às empresas e causado incontáveis horas de frustração. Muitos desses ataques permitem que o perpetrador invada uma rede ou sistema, o que pode levar a roubo de dados, corrupção de dados e outros ataques. Nesta seção, discutiremos diversos tipos de ataques contra sistemas de computador.

### 15.5.1 CRIPTOANÁLISE

Um **ataque criptoanalítico** tenta decriptar texto cifrado sem possuir a chave de decriptação. A forma mais comum de ataque criptoanalítico é aquela em que o algoritmo criptográfico é analisado para descobrir relações entre bits da chave criptográfica e bits do texto cifrado. A meta desse tipo de ataque é determinar a chave do texto cifrado.

Tendências estatísticas fracas entre texto cifrado e chaves podem ser exploradas para tentar descobrir a chave. Gerenciamento adequado de chaves e datas de expiração de chaves podem reduzir a suscetibilidade a ataques criptoanalíticos. Quanto mais tempo uma chave criptográfica for usada, mais texto cifrado um invasor pode usar para derivar a chave. Se uma chave for recuperada ocultamente por um invasor, poderá ser usada para decriptar todas as mensagens que usarem aquela chave.

### 15.5.2 VÍRUS E VERMES

Um **vírus** é um código executável - freqüentemente enviado como um anexo de uma mensagem por e-mail ou oculto em arquivos como clipes de áudio, clipes de vídeo e jogos -, que fica anexado a um arquivo ou sobrescreve outros arquivos para se reproduzir. Vírus podem corromper arquivos, controlar aplicações ou até mesmo apagar um disco rígido. Hoje, os vírus podem ser propagados através de uma rede simplesmente compartilhando arquivos 'infectados' inseridos em anexos de e-mail, documentos ou programas.

Um **verme** é um código executável que se propaga infectando arquivos de uma rede. A propagação de vermes raramente requer qualquer ação de usuário e eles também não precisam ser anexados a um outro programa ou arquivo. Uma vez liberado, um vírus ou um verme pode se propagar com rapidez, freqüentemente infectando milhões de computadores no mundo inteiro em minutos ou horas.

Vírus podem ser classificados da seguinte maneira:

1. **vírus de setor de boot**: infecta o setor de boot do disco rígido do computador, o que permite que seja carregado juntamente com o sistema operacional e potencialmente controle o sistema.

2. **vírus transiente**: fica anexado a um programa de computador particular. O vírus é ativado quando o programa é executado, e desativado quando o programa é encerrado.

3. **vírus residente**: uma vez carregado na memória de um computador, funciona até que o computador seja desligado.

4. **bomba lógica**: executa seu **código**, ou **carga explosiva** (*payload*) quando encontra determinada condição. Um exemplo de bomba lógica é uma **bomba-relógio**, ativada quando o relógio do computador coincide com um certo horário ou data.

O **cavalo-de-tróia** é um programa mal-intencionado que se esconde dentro de um programa autorizado ou simula um programa ou característica legítima enquanto causa dano ao computador ou à rede quando é executado. O nome cavalo-de-tróia origina-se da lenda da guerra entre Tróia e Grécia [Nota: Nessa história, guerreiros gregos esconderam-se dentro de um cavalo de madeira que os troianos recolheram para dentro das muralhas da cidade de Tróia. Quando a noite caiu e os troianos estavam dormindo, os guerreiros gregos saíram de dentro do cavalo e abriram os portões da cidade, permitindo que o exército grego entrasse e destruísse a cidade de Tróia.] Pode ser particularmente difícil detectar programas desse tipo porque eles parecem ser aplicações legítimas e operacionais.

**Programas de porta dos fundos** (*backdoor*) são vírus residentes que permitem ao emissor acesso completo, não detectado, aos recursos do computador da vítima. Esses tipos de vírus são especialmente ameaçadores para a vítima, pois podem ser programados para registrar cada toque de teclado (capturando todas as senhas, números de cartões de crédito etc.).

### Vírus de Alta Propagação

Dois vírus que atraíram significativa atenção dos meios de comunicação são os vírus Melissa, que atacou em março de 1999, e o ILOVEYOU, que atacou em maio de 2000. Cada um causou bilhões de dólares de prejuízo. O Melissa propagou-se por documentos Microsoft Word enviados por e-mail. Quando se abria o documento, o vírus era acionado e, então, acessava a agenda de endereços do Microsoft Outlook do usuário (uma lista de endereços de e-mail) daquele computador e enviava o anexo Word infectado por e-mail a um máximo de 50 pessoas que constavam da agenda de endereços do usuário. Toda vez que um outro usuário abria o anexo, o vírus enviava até 50 mensagens adicionais. Uma vez residente no sistema, o vírus continuava infectando quaisquer arquivos salvos usando Microsoft Word.

O vírus ILOVEYOU era enviado como um anexo a um e-mail que fingia ser uma carta de amor. A mensagem do e-mail era: "Favor verificar a carta de amor anexa enviada por mim". Uma vez aberta a carta, o vírus acessava a agenda de endereços do Microsoft Outlook e enviava mensagens a cada endereço da lista, habilitando-o a se espalhar rapidamente pelo mundo inteiro. O vírus corrompia muitos tipos de arquivos, incluindo arquivos de sistemas operacionais. Muitas redes ficaram desativadas durante dias por causa do número maciço de e-mails gerados.

Esse vírus expôs as inadequações da segurança dos e-mails, como a falta de um software para fazer a varredura de anexos de arquivos em busca de ameaças à segurança antes que eles fossem abertos. Também ensinou os usuários a ficar mais atentos a e-mails suspeitos, mesmo aos enviados por alguém conhecido.

### Vermes Sapphire/Slammer: análise e implicações

Vermes se propagam explorando pontos fracos nos canais de comunicação estabelecidos por softwares, seja por aplicações, seja pelo sistema operacional. Uma vez descoberto um ponto fraco, um verme pode produzir tráfego de rede suficiente para desativar um único computador ou uma rede de computadores. Além disso, um verme pode ser projetado para executar código no computador que ele infecta, permitindo, potencialmente, que o criador do verme obtenha ou destrua informações sensíveis.

Entre os ataques de vermes que receberam a atenção dos meios de comunicação estão *Nimda*, *Code Red* e *Sapphire*, também denominado *Slammer*. O *Slammer*, que infectou os computadores mais vulneráveis em dez minutos a partir da sua liberação em 25

de janeiro de 2003, dobrava o número de computadores infectados a cada 8,5 segundos. [Nota: Computadores vulneráveis eram os que executavam o SQL Server 2000 da Microsoft aos quais não se aplicava uma correção de segurança liberada pela Microsoft em julho de 2002.] A taxa de infecção eram duas ordens de magnitude mais rápida do que a do seu famoso predecessor, o vírus *Code Red*. Esse último, um verme de 4 KB, instanciaava vários threads para criar conexões TCP para infectar novos hospedeiros. O *Slammer*, ao contrário, funcionava por UDP e sua carga explosiva estava contida em um único pacote UDP de 404 bytes. O protocolo UDP sem conexão, aliado a um algoritmo de varredura aleatória para gerar endereços IP como alvos do ataque, fazia do *Slammer* um verme particularmente virulento. Um **algoritmo de varredura aleatória** usa números pseudo-aleatórios para gerar uma distribuição ampla de endereços IP como alvos para infectar.

O verme *Slammer* causou quedas de sistemas e redes em razão da saturação causada por seus pacotes UDP. O interessante é que o verme não portava nenhuma carga explosiva mal-intencionada e atacava a vulnerabilidade de uma aplicação de uso mundial relativamente limitado. O verme *Slammer* também continha o que parecia ser um erro lógico no seu algoritmo de varredura que limitava significativamente o número de endereços IP que ele podia alcançar.

Softwares antivírus podem proteger contra vírus e alguns vermes. A maioria dos softwares antivírus é reativa, o que significa que podem atacar vírus conhecidos, mas não proteger contra vírus desconhecidos ou futuros.

### **15.5.3 ATAQUES DE RECUSA DE SERVIÇO (DoS)**

Um ataque de recusa de serviço (Denial-of-Service -DoS) impede um sistema de atender requisições legítimas. Em muitos ataques DoS, tráfego não autorizado satura os recursos de uma rede, restringindo o acesso a usuários legítimos. Normalmente o ataque é executado pela inundação dos servidores com pacotes de dados. Ataques de recusa de serviço em geral requerem que uma rede de computadores trabalhe simultaneamente, embora seja possível realizar alguns ataques habilidosos com uma única máquina. Ataques de recusa de serviço podem fazer com que computadores em rede caiam ou sejam desconectados, destruindo o serviço de um site Web ou até mesmo desativando sistemas críticos como telecomunicações ou centrais de controle de tráfego aéreo.

Um outro tipo de ataque de recusa de serviço visa às tabelas de roteamento de uma rede. Lembre-se de que tabelas de roteamento proporcionam uma visão da topologia da rede e são usadas por um roteador para determinar para onde enviar os dados. Esse tipo de ataque é executado modificando as tabelas de roteamento e, assim, redirecionando propositalmente a atividade da rede. Por exemplo, as tabelas de roteamento podem ser modificadas para que passem a enviar todos os dados que chegam a um único endereço da rede. Um ataque semelhante, o **ataque ao sistema de nome de domínio** (*Domain Name System - DNS*), pode modificar o endereço para o qual o tráfego de um site Web particular é enviado. Ataques desses tipos podem ser usados para redirecionar usuários de um site Web particular para um outro site, potencialmente mal-intencionado. Esses ataques são particularmente perigosos se o site Web ilegítimo se fizer passar por um real, levando usuários a revelar informações sensíveis ao invasor.

Em um **ataque de recusa de serviço distribuído**, a inundação de pacotes vem de vários computadores ao mesmo tempo. Esses ataques normalmente são iniciados por um indivíduo que infectou diversos computadores com um vírus com o intuito de obter acesso não autorizado aos computadores para executar o ataque. Ataques de recusa de serviço distribuído podem ser difíceis de interromper porque não é fácil determinar quais requisições de uma rede são de usuários legítimos e quais são parte do ataque. Também é particularmente difícil identificar o perpetrador desses ataques porque eles não são executados diretamente do computador do invasor.

Em fevereiro de 2000, ataques de recusa de serviço distribuído desativaram vários sites Web de tráfego intenso, entre eles Yahoo, eBay, CNN Interactive e Amazon. Nesse caso, um único usuário utilizou uma rede de computadores para inundar os sites Web com tráfego que afogou os computadores dos sites. Embora os ataques de recusa de serviço meramente interrompam o acesso a um site Web e não afetem os dados da vítima, eles podem ser extremamente custosos. Por exemplo, quando o site Web do eBay

ficou fora do ar por 24 horas, em 6 de agosto de 1999, o valor de suas ações caiu vertiginosamente.

Quem é responsável por vírus e ataques de recusa de serviço? Na maioria das vezes, as partes responsáveis são denominadas **hackers**, mas esse não é um nome adequado. Na área de computadores, hacker se refere a um programador experiente, muitas vezes um programador que faz programas tanto para seu próprio prazer quanto pela funcionalidade da aplicação. O verdadeiro termo para aquele tipo de pessoa é **cracker**, que é alguém que usa um computador de maneira mal-intencionada (e muitas vezes ilegalmente) para invadir um outro sistema ou fazê-lo falhar.

#### **15.5.4 EXPLORAÇÃO DE SOFTWARE**

Um outro problema que assola as empresas de comércio eletrônico é a exploração de software por crackers. Todo programa de uma máquina que funciona em rede deve ser verificado em busca de vulnerabilidades. Entretanto, com milhões de produtos de software disponíveis e vulnerabilidades descobertas diariamente, essa tarefa se torna esmagadora. Um método comum de exploração de vulnerabilidade é um **transbordamento de buffer**, no qual um programa recebe entrada maior do que seu espaço alocado.

Um transbordamento de buffer ocorre quando uma aplicação envia mais dados a um buffer do que ele pode conter. Um ataque de transbordamento de buffer pode deslocar os dados adicionais para buffers adjacentes, corrompendo ou sobrescrevendo dados existentes. Esse tipo de ataque, quando bem projetado, pode substituir código executável da pilha de uma aplicação para alterar seu comportamento. Ataques de transbordamento de buffer podem conter código mal-intencionado que, então, poderá executar com os mesmos direitos de acesso que a aplicação atacada. Dependendo do usuário e da aplicação, o invasor pode obter acesso ao sistema inteiro. BugTraq ([www.securityfocus.com](http://www.securityfocus.com)) foi criado em 1993 para listar vulnerabilidades, como explorá-las e como repará-las.

#### **15.5.5 INVASÃO DE SISTEMA**

O resultado de muitos ataques contra a segurança é a invasão do sistema ou da rede. Segundo um estudo comparativo do Computer Security Institute ([www.gocsi.com](http://www.gocsi.com)), 40% dos entrevistados relataram que um intruso tinha conseguido invadir seus sistemas. Após um invasor explorar um sistema operacional ou o software que está sendo executado no computador, o sistema fica vulnerável a inúmeros ataques - desde roubo e manipulação de dados até uma queda de sistema. A **invasão de sistema** é uma violação bem-sucedida da segurança do computador por um usuário externo não autorizado. Toda invasão de sistema é potencialmente perigosa, embora uma reação rápida em geral consiga frustrar o ataque de um intruso antes que seja causado qualquer dano significativo. Muitos ataques, como roubo de dados e desfiguração da Web, dependem de uma invasão bem-sucedida do sistema como fundamento.

**Desfiguração da Web** é uma forma de ataque popular pela qual os crackers obtêm acesso ilegalmente para modificar o site Web de uma organização e mudar o seu conteúdo. A desfiguração da Web tem atraído significativa atenção dos meios de comunicação. Um caso notável ocorreu em 1996, quando crackers suecos modificaram o site da Central Intelligence Agency (CIA) para "Central Stupidity Agency". Os vândalos inseriram na página obscenidades, mensagens políticas, bilhetes para administradores de sistemas e links com sites de conteúdo adulto. Muitos outros sites Web populares e de grande porte sofreram desfiguração. Hoje, desfigurar sites Web tornou-se imensamente popular entre os crackers, o que provocou o fechamento dos arquivos de registro de sites atacados (que registravam mais de 15 mil sites vandalizados) em razão do volume de sites vandalizados diariamente.

A invasão do sistema muitas vezes ocorre como resultado de um cavalo-de-tróia, um programa de porta dos fundos ou um erro explorado no software ou no sistema operacional. Permitir que usuários externos obtenham acesso a aplicações via Web proporciona um outro canal para o intruso invadir o sistema. Vulnerabilidades em aplicações de servidores Web comuns, como o *Microsoft Internet Information Services* (IIS) e o Apache HTTP Server, dão aos invasores uma rota bem conhecida para invadir um sistema se os administradores não aplicarem as correções necessárias. A invasão de sistemas também

pode ocorrer em computadores pessoais por meio de softwares conectados com a Internet, como navegadores Web.

## 15.6 PREVENÇÃO DE ATAQUES E SOLUÇÕES DE SEGURANÇA

A seção anterior detalhou diversos ataques comuns contra a segurança de computadores. Embora o número de ameaças à segurança de computadores talvez pareça esmagador, na prática, bom senso e diligência podem evitar um grande número de ataques. Para reforçar a segurança, hardware e software especializados em frustrar uma variedade de ataques podem ser instalados em computadores e em redes.

### 15.6.1 FIREWALLS

O **firewall** protege uma rede local (LAN) contra intrusos de fora da rede e polícia o tráfego que chega e que parte da LAN. Firewalls podem proibir todas as transmissões de dados que não sejam expressamente permitidas ou permitir todas as transmissões de dados que não sejam expressamente proibidas. A escolha entre esses dois modelos pode ser determinada pelo administrador de segurança da rede; o primeiro proporciona um alto grau de segurança, mas pode impedir a transferência legítima de dados. O último deixa o sistema mais suscetível a ataques, mas, em geral, não restringe transferências legítimas na rede. Cada LAN pode ser conectada à Internet por meio de um gateway (portal) que normalmente inclui um firewall. Durante anos, as ameaças mais significativas à segurança originaram-se de profissionais que estavam dentro do firewall. Agora que todas as empresas dependem intensamente do acesso à Internet, um número cada vez maior de ameaças à segurança se origina fora do firewall - das centenas de milhões de pessoas conectadas à rede da empresa via Internet.

Há dois tipos primários de firewalls. Um **firewall de filtragem de pacotes** examina todos os dados enviados de fora da LAN e rejeita pacotes de dados com base em regras predefinidas, como rejeitar pacotes que tenham endereços da rede local ou rejeitar pacotes de certos endereços ou portas. Por exemplo, suponha que um hacker de fora da rede obtenha o endereço de um computador que pertence à rede e tente passar um pacote de dados prejudicial através de um firewall enviando um pacote que indique ter sido enviado de um computador de dentro da rede. Nesse caso, um firewall de filtragem de pacotes rejeitará o pacote de dados porque o endereço de retorno do pacote que pretende entrar foi claramente modificado. Uma limitação dos firewalls de filtragem de pacote é que eles consideram apenas a origem dos pacotes de dados; não examinam os dados anexos. O resultado é que vírus mal-intencionados podem ser instalados no computador de um usuário autorizado, permitindo que o invasor tenha acesso à rede sem o conhecimento daquele usuário. A meta de um **gateway de nível de aplicação** é proteger a rede contra os dados contidos em pacotes. Se a mensagem contiver um vírus, o gateway poderá bloqueá-lo e impedir que seja enviado ao receptor pretendido.

Instalar um firewall é um dos modos mais efetivos e mais fáceis de aumentar a segurança de uma rede de pequeno porte. É comum que pequenas empresas ou residências conectadas à Internet por meio de conexões permanentes, como modem por cabo, não empreguem fortes medidas de segurança. O resultado é que seus computadores são alvos de primeira para a exploração dos crackers por ataques de recusa de serviço ou roubo de informações. Entretanto, é importante que todos os computadores conectados à Internet contenham um certo grau de segurança para seus sistemas. Muitos produtos populares de rede, como roteadores, proporcionam capacidades de firewall, e certos sistemas operacionais como o Windows XP fornecem software de firewall. Na verdade, o Windows XP *Internet Connection Firewall* (ICF) é habilitado por default.

A **tecnologia da camada de ar** é uma solução de segurança de rede que complementa o firewall. Proporciona segurança de dados privados contra tráfego externo que tem acesso à rede interna. A camada de ar separa a rede interna da rede externa, e a organização decide quais informações serão disponibilizadas para usuários externos. A *Whale Communications* criou o *e-Gap System*, composto de dois servidores e um banco de memória. O banco de memória não executa um sistema operacional, portanto, os hackers não podem tirar proveito dos pontos fracos comuns dos sistemas operacionais para acessar informações da rede.

A tecnologia da camada de ar não permite que usuários de fora vejam a estrutura da rede, o que impede que os hackers pesquisem a rede em busca de pontos fracos. A tecnologia de camada de ar é usada por organizações de comércio eletrônico para permitir que seus clientes e parceiros acessem informações com segurança transparente, reduzindo, assim, o custo do gerenciamento de estoque. Os setores militar, aeroespacial e governamental, que armazenam informações altamente sensíveis, empregam a tecnologia de camada de ar.

### **15.6.2 SISTEMAS DE DETECÇÃO DE INTRUSOS (IDSs)**

**Sistemas de detecção de intrusos** (*Intrusion Detection Systems - IDSs*) monitoram **arquivos de registro** (log files) de redes e aplicações que registram informações sobre o comportamento da rede, como o horário em que os serviços do sistema operacional são requisitados e o nome do processo que os requisita. IDSs examinam arquivos de registro para alertar os administradores do sistema contra aplicações e/ou comportamentos suspeitos do sistema. Se uma aplicação exibir comportamento errático ou mal-intencionado, um IDSs pode interromper a execução daquele processo.

Sistemas de **detecção de intrusos baseada no hospedeiro** monitoram arquivos de registro do sistema e da aplicação, o que é especialmente útil para detectar cavalos-de-tróia. Softwares de **detecção de intrusos baseada na rede** monitoram o tráfego de uma rede em busca de padrões fora do comum que poderiam indicar ataques DoS ou acesso à rede por um usuário não autorizado. Então, os administradores do sistema podem verificar seus arquivos de registro para determinar se houve uma invasão e, caso isso tenha acontecido, rastrear o criminoso.

A detecção de intrusos via **análise estática** tenta detectar quando aplicações foram corrompidas por um hacker. A técnica da análise estática admite que os hackers tentam atacar um sistema usando chamadas ao sistema. Adotando essa premissa, o primeiro passo para detectar intrusos é construir um modelo do comportamento esperado de uma aplicação (um padrão das chamadas ao sistema normalmente geradas pela aplicação). Então, o padrão de chamadas ao sistema da aplicação é monitorado durante sua execução; um ataque pode ser detectado se esse padrão for diferente do padrão do modelo estático.

O **método OCTAVE** (*Operationally Critical Threat, Asset and Vulnerability Evaluation*), desenvolvido pelo Software Engineering Institute da Carnegie Mellon University, avalia as ameaças à segurança de um sistema. O OCTAVE tem três fases: montagem de perfis de ameaças, identificação de vulnerabilidades e desenvolvimento de soluções e planos de segurança. No primeiro estágio, a organização identifica suas informações e ativos importantes e avalia os níveis de segurança requeridos para sua proteção. Na segunda fase, o sistema é examinado em busca de pontos fracos que poderiam comprometer os dados valiosos. A terceira fase é o desenvolvimento de uma estratégia de segurança aconselhada por uma equipe de análise formada por três a cinco especialistas designados pelo OCTAVE. Essa abordagem é uma das primeiras desse tipo, na qual os proprietários dos sistemas de computador não somente obtêm análise profissional, mas também participam da priorização de informações cruciais.

### **15.6.3 SOFTWARE ANTIVÍRUS**

Como discutimos na Seção 15.5.2, vírus e vermes transformaram-se em uma ameaça para usuários profissionais e também residenciais e custam bilhões de dólares às empresas. O número de vírus reportados vem aumentando continuamente desde meados da década de 1990. Em resposta, softwares antivírus têm sido desenvolvidos e modificados para atender ao número e variedade crescentes de ataques de vírus em sistemas de computador. Software antivírus tenta proteger um computador contra um vírus e/ou identificar e eliminar vírus presentes naquele computador. Há uma variedade de técnicas que os softwares antivírus podem usar para detectar e eliminar vírus presentes em um sistema; contudo, nenhum deles pode oferecer proteção completa.

**Detecção de vírus por verificação de assinatura** depende de conhecer a estrutura do código do vírus de computador. Por exemplo, muitos programas antivírus mantêm uma lista de vírus conhecidos e de seus códigos. Todos os vírus contêm uma região denominada **assinatura do vírus** que não muda durante a propagação do vírus. Na prática,

a maioria das listas de vírus conhecidos mantém uma lista de assinaturas de vírus. Nesse caso, o software de detecção de vírus verifica o computador e compara dados de arquivos com códigos de vírus.

Um ponto fraco das listas de vírus conhecidos é que elas podem se tornar proibitivamente grandes à medida que os vírus proliferam. A lista de vírus deve ser atualizada periodicamente para identificar com sucesso os vírus emergentes. Talvez o ponto fraco mais sério dessas listas seja que elas podem reconhecer somente vírus que já foram previamente identificados pelo provedor da lista. Por isso, elas em geral não protegem contra vírus novos e não identificados.

Uma lista de vírus conhecidos pode ser particularmente ineficaz contra vírus variantes e polimórficos. Um vírus variante é aquele cujo código foi modificado em relação à sua forma original, mas ainda retém sua carga explosiva perniciosa. Um **vírus polimórfico** muda seu código (por exemplo, via criptografia, substituição, inserção e coisas semelhantes) enquanto se propaga, para escapar das listas de vírus conhecidos (Figura 15.8).

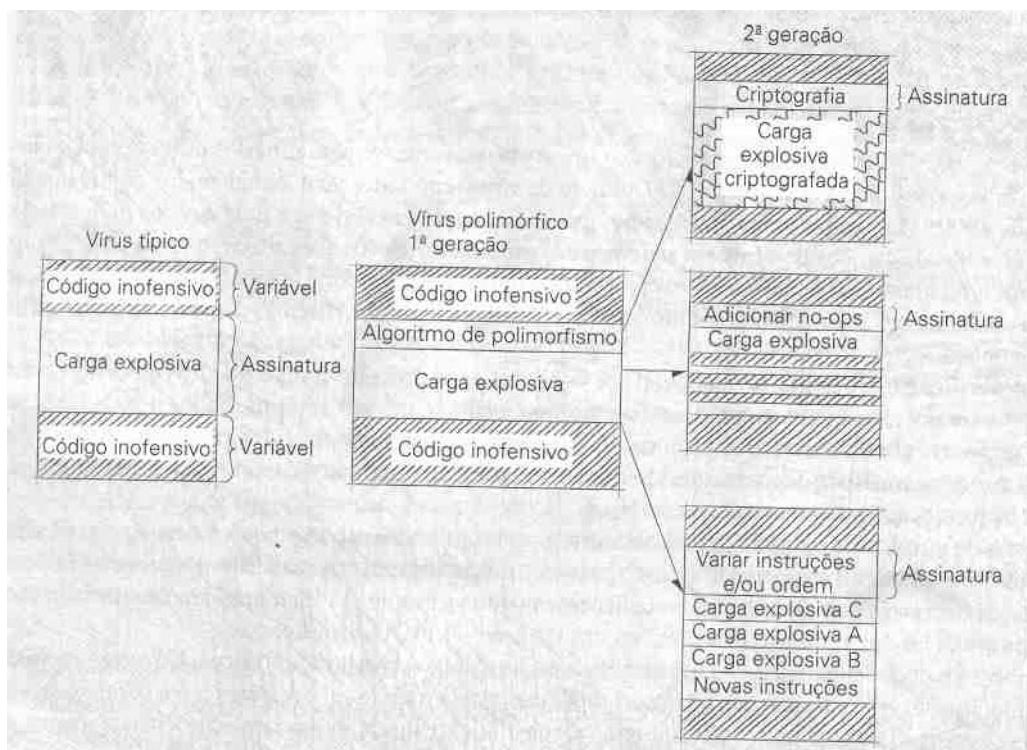


Figura 15.8: Vírus Polimórficos

Embora assinaturas de vírus melhorem a capacidade de um verificador de vírus detectar vírus e suas variantes, também introduzem a possibilidade de detecção falsa positiva ou falsa negativa de vírus. Alertas de vírus falsos positivos indicam incorretamente que um vírus reside em um arquivo, ao passo que alertas falsos negativos determinam incorretamente que um arquivo infectado está limpo. Esses resultados incorretos tomam-se mais freqüentes à medida que o tamanho da assinatura de um vírus fica menor. O problema de evitar leituras falsas positivas ou falsas negativas durante a verificação de vírus é desafiador, pois já se demonstrou que podem ser criadas assinaturas de vírus de apenas dois bytes.

Uma alternativa para a verificação de vírus é a **verificação heurística**. Vírus são caracterizados por replicação, residência na memória e/ou código destrutivo. Verificações heurísticas podem evitar a propagação de vírus detectando ou suspendendo qualquer programa que exiba esse comportamento. A força primordial da verificação heurística é que ela pode detectar vírus que ainda não foram identificados. Entretanto, do mesmo modo que a verificação de assinatura, a verificação heurística também é suscetível a resultados falsos. A maioria dos softwares antivírus emprega uma combinação de verificação de assinatura e verificação heurística.

Todos os vírus (com exceção dos vermes) têm de modificar um arquivo para infectar um computador. Conseqüentemente, uma outra técnica antivírus é monitorar mudanças

em arquivos verificando a sua consistência. A maioria das verificações de consistência é implementada como somas de verificação (checksum) de arquivos protegidos. Manter um registro de consistência de arquivos em um sistema que experimenta um alto volume de E/S de arquivos não é uma opção viável se os usuários esperam tempos de resposta rápidos. Conseqüentemente, muitos programas antivírus asseguram consistência de arquivo para um conjunto limitado de arquivos (normalmente os arquivos do sistema operacional). Entretanto, verificações de consistência de arquivos não podem proteger contra vírus que residem no computador antes da instalação do software antivírus. Alguns sistemas operacionais como o Windows XP executam verificações de consistência em sistemas de arquivos vitais e os substituem se forem alterados para proteger a integridade do sistema.

Além das técnicas de verificação, os softwares antivírus podem ser caracterizados pelo comportamento do seu programa. Por exemplo, **verificadores de tempo real** são residentes na memória e evitam vírus ativamente, ao passo que outros softwares antivírus devem ser carregados manualmente e servem apenas para identificar vírus. Alguns programas antivírus avisam o usuário para entrar em ação quando um vírus é detectado, ao passo que outros eliminam o vírus sem a interação do usuário.

#### **15.6.4 CORREÇÕES DE SEGURANÇA**

Sistemas Operacionais e outros softwares contêm falhas de segurança que não são descobertas senão depois de muitos usuários os terem instalado. Para tratar falhas de segurança de maneira oportuna, os desenvolvedores de software devem:

- Descobrir falhas anteriormente desconhecidas;
- Liberar rapidamente correções (*patches*) para as falhas; e
- Estabelecer fortes linhas de comunicação com o usuário.

Para reduzir o dano causado por falhas de segurança, desenvolvedores devem abordar as que já foram exploradas e procurar e sanar ativamente as que não foram exploradas. A liberação de um código que aborda uma falha de segurança é denominada **correção de segurança** (*security patch*).

Muitas vezes, simplesmente liberar uma correção para uma falha de segurança é insuficiente para melhorar a segurança. Por exemplo, o verme Slammer explorava uma falha de segurança para a qual tinha sido liberada uma correção seis meses antes. portanto, desenvolvedores de software devem tratar falhas de segurança notificando seus usuários rapidamente e disponibilizando software para facilitar o processo de aplicação de correções de segurança.

#### **15.6.5 SISTEMA DE ARQUIVOS SEGUROS**

Acesso a recursos de sistemas, incluindo dados armazenados em um arquivo do sistema, pode ser regulamentado pela política de controle do acesso ao sistema. Todavia, esses mecanismos não evitam, necessariamente, o acesso a dados armazenados no disco rígido quando esse for acessado por um sistema operacional diferente. Como consequência, vários dos sistemas operacionais de hoje suportam sistemas de arquivos seguros que protegem dados sensíveis independentemente do modo como os dados são acessados. O Windows XP emprega o *New Technology File System* (NTFS) que protege arquivos via controle de acesso e criptografia.

O **sistema de criptografia de arquivos** (*Encrypting File System* - EFS) usa criptografia para proteger arquivos e pastas em um sistema de arquivo NTFS. O EFS usa criptografia por chave secreta e por chave pública para garantir a segurança dos arquivos. Cada usuário recebe um par de chaves e um certificado, usados para garantir que somente o usuário que criptografou os arquivos possa ter acesso a eles. Os dados do arquivo serão perdidos se a chave for perdida. O EFS em geral é implementado em sistemas multiusuários ou sistemas móveis para assegurar que os arquivos protegidos não estejam acessíveis a alguém que esteja usando uma máquina roubada ou perdida.

Criptografia pode ser aplicada a arquivos individuais ou a pastas inteiras; no último caso, cada arquivo da pasta é criptografado. Aplicar criptografia no nível de pasta muitas

vezes obtém um nível mais alto de segurança por impedir que os programas criem arquivos temporários em texto comum.

### **15.6.6 O LIVRO LARANJA DA SEGURANÇA**

Para avaliar as características de segurança de sistemas operacionais, o Departamento de Defesa dos Estados Unidos (DoD) publicou um documento intitulado *Department of Defense Trusted Computer System Evaluation Criteria* (Critérios Confiáveis de Avaliação de Sistemas de Computador), também conhecido como *Orange Book* (Livro Laranja) em dezembro de 1985. Esse documento ainda é usado para definir níveis de segurança de sistemas operacionais. O Livro Laranja, originalmente elaborado para avaliar sistemas militares, classifica-os em quatro níveis, A, B, C, e D. O nível mais baixo de segurança é D e o mais alto é A. Os requisitos de cada nível são os seguintes:

- Nível D: Qualquer sistema que não cumpra todos os requisitos de qualquer dos outros níveis. Sistemas categorizados como nível D em geral são inseguros.
- Nível C: Este nível contém dois subníveis. Subnível C 1 requer que o sistema operacional separe usuários de dados, o que significa que indivíduos e grupos devem registrar um nome de usuário ou de grupo e uma senha para usar o sistema. A segurança das informações privadas pertencentes a um indivíduo ou grupo é garantida, impedindo que outros indivíduos e grupos as leiam ou modifiquem. As primeiras versões do UNIX pertencem a esse nível. O Nível C2 suporta apenas acesso individual com senha, o que significa que usuários não podem ter acesso ao sistema com um nome de grupo. Indivíduos autorizados podem acessar somente certos arquivos e programas. Ambos, C1 e C2, permitem que indivíduos controlem o acesso a seus arquivos e informações privadas, o que significa que requerem apenas controle de acesso discricionário. A maioria dos sistemas operacionais, como Windows NT, sistemas UNIX modernos e o IBM OS/400, está nessa categoria.
- Nível B: Neste nível é exigido controle de acesso obrigatório, o que significa que o sistema operacional requer um esquema de permissão central predefinido para determinar as permissões designadas a sujeitos. O criador de um objeto não controla as permissões para aquele objeto. Este nível contém três subníveis. Além dos requisitos de C2, o nível B1 requer que o sistema operacional contenha um esquema de permissão central predefinido e aplique etiquetas de sensibilidade (por exemplo, 'Confidencial') a sujeitos e objetos. O mecanismo de controle de acesso deve usar essas etiquetas de sensibilidade para determinar permissões. Entre os sistemas operacionais que satisfazem os requisitos de B1 estão HP-UX BLS, SEVMS e CS/SX. O nível B2 requer que a linha de comunicação entre o usuário e o sistema operacional para autenticação seja segura, além dos requisitos B1. Exemplos de sistemas que satisfazem os requisitos B2 são o Multics e o VSLAN. O nível B3 requer todas as características presentes em B2 além da implementação de domínios de proteção, fornecendo mecanismos de recuperação seguros (recuperação sem comprometimento da proteção após uma falha de sistema) e monitorando todos os acessos a sujeitos e objetos para análise. O sistema operacional XTS-300 satisfaz os requisitos B3.
- Nível A: Este nível contém dois subníveis. O subnível A1 requer todas as características providas por B3 e exige que a segurança do sistema operacional seja formalmente verificada. Um exemplo de sistema operacional que satisfaz os requisitos A1 é o Boeing MLS LAN. Os requisitos de A2 são reservados para uso futuro.

## **15.7 COMUNICAÇÃO SEGURA**

O comércio eletrônico lucrou muito com o crescimento rápido do número de consumidores cujos computadores têm conexões com a Internet. Todavia, à medida que o número de transações on-line aumenta, também aumenta o volume de dados sensíveis transmitidos pela Internet. Aplicações que processam transações requerem conexões seguras pelas quais dados sensíveis possam ser transmitidos. Diversos métodos para fornecer transações seguras foram desenvolvidos nos últimos anos. Há cinco requisitos para uma transação bem-sucedida e segura:

- privacidade

- integridade
- autenticação
- autorização
- não-rejeição

A questão da **privacidade** é: como garantir que a informação que você transmite pela Internet não seja capturada ou passada a um terceiro sem o seu conhecimento? A questão da **integridade** é: como assegurar que a informação que você envia ou recebe não seja comprometida nem alterada? A questão da **autenticação** é: como o emissor e o receptor de uma mensagem verificam reciprocamente suas identidades? A questão da **autorização** é: como você gerencia o acesso a recursos protegidos com base em credenciais do usuário que consistem em identidade (por exemplo, nome do usuário) e prova de identidade (por exemplo, senha)? A questão da **não-rejeição** é: como você prova legalmente que uma mensagem foi enviada ou recebida? A segurança de redes também deve abordar a questão da disponibilidade: como assegurar que a rede e os sistemas de computador que ela conecta funcionarão continuamente?

## 15.8 ESTEGANOGRAFIA

**Esteganografia**, derivada da raiz grega que significa 'escrita oculta', é a prática de ocultar informação dentro de outra informação. Como a criptografia, a esteganografia tem sido usada desde a Antigüidade. Ela pode ser utilizada para ocultar uma informação, como uma mensagem ou imagem, com uma outra imagem, mensagem ou outra forma de multimídia.

Considere um simples exemplo textual: suponha que o cliente de um corretor de ações deseja realizar uma transação via canal não seguro. O cliente poderia enviar a mensagem "BURIED UNDER YARD" (enterrado no pátio). Se o cliente e o corretor tiverem combinado antes que a mensagem estaria contida nas primeiras letras de cada palavra, o corretor extraeria "BUY" (compre).

Uma aplicação cada vez mais popular da esteganografia são as **marcas-d'água digitais** para proteção da propriedade intelectual. A esteganografia digital explora porções não utilizadas de arquivos codificados usando formatos particulares como em imagens ou de discos removíveis. O espaço insignificante armazena a mensagem oculta, enquanto o arquivo digital mantém sua semântica pretendida. Uma marca-d'água digital pode ser visível ou invisível. Usualmente é o logotipo de uma empresa, uma declaração de direitos autorais ou uma outra marca ou mensagem que indique o proprietário de um documento. O proprietário de um documento poderia mostrar a marca-d'água oculta em tribunais, por exemplo, para provar que o item portador da marca-d'água foi roubado.

A marca-d'água digital pode causar um impacto substancial no comércio eletrônico. Considere a indústria musical. Distribuidores de música e filmes estão preocupados que as tecnologias MP3 e o MPEG facilitem a distribuição ilegal de material protegido por direitos autorais. Conseqüentemente, muito editores hesitam em publicar conteúdo online, pois conteúdo digital é fácil de copiar. E mais, porque CD-ROMs e DVD-ROMs armazenam informações digitais, usuários podem copiar arquivos multimídia e os compartilhar via Web. Usando marcas-d'água digitais, editores musicais podem fazer mudanças imperceptíveis em uma certa parte de uma canção em uma freqüência que não é audível para seres humanos para mostrar que a canção foi, de fato, copiada.

## 15.9 ESTUDO DE CASO: SEGURANÇA DE SISTEMAS UNIX

Sistemas UNIX são projetados para incentivar interação de usuários, o que pode fazer com que fique mais difícil garantir sua segurança. Sistemas UNIX pretendem ser abertos; suas especificações e códigos-fonte são amplamente disponíveis.

O arquivo de senhas do UNIX é criptografado. Quando um usuário digita uma senha, ela é criptografada e comparada com o arquivo de senhas criptografadas. Assim, as senhas são irrecuperáveis até mesmo pelo administrador do sistema. Sistemas UNIX usam salpicamento na criptografia das senhas. O salpicamento é uma cadeia de dois caracteres selecionada aleatoriamente via função do horário e do ID do processo. Então, 12 bits do

salpicamento modificam o algoritmo criptográfico. Assim, usuários que escolhem a mesma senha (por coincidência ou intencionalmente) terão senhas criptografadas diferentes (com alta probabilidade). Algumas instalações modificam o programa de senhas para impedir que usuários utilizem senhas fracas.

O arquivo de senhas deve ser legível para qualquer usuário porque contém outras informações cruciais (por exemplo, nomes de usuários, ID de usuários e assemelhados) requeridas por muitas ferramentas UNIX. Por exemplo, porque diretórios empregam IDs de usuários para registrar propriedade de arquivo, *ls* (a ferramenta que lista conteúdo de diretórios e propriedade de arquivos) precisa ler o arquivo de senhas para determinar nomes de usuários com base em IDs de usuários. Se crackers obtiverem o arquivo de senhas, eles poderão, potencialmente, quebrar a criptografia da senha. Para tratar essa questão, o UNIX protege o arquivo de senhas contra crackers, armazenando quaisquer informações que não sejam as senhas criptografadas no arquivo normal de senhas e armazenando as senhas criptografadas em um arquivo-sombra de senhas que pode ser acessado somente por usuários com privilégio de raiz.

Com a característica de permissão *setuid* do UNIX, um programa pode ser executado usando os privilégios de um outro usuário. Essa característica poderosa contém falhas de segurança, particularmente quando o privilégio resultante é o de um “superusuário” (com acesso a todos os arquivos de um sistema UNIX). Por exemplo, se um usuário regular conseguir executar um interpretador de comandos que pertença ao superusuário e para o qual o bit *setuid* foi configurado, então, essencialmente, o usuário regular se tornará o superusuário. Fica claro que o *setuid* deve ser empregado com cuidado. Usuários, incluindo os que têm privilégios de superusuário, devem examinar periodicamente seus diretórios para confirmar a presença de arquivos *setuid* e detectar quaisquer arquivos que não devam ser *setuid*.

Um meio relativamente simples de comprometer a segurança em sistemas UNIX (e em outros sistemas operacionais) é instalar um programa que imprima o aviso de acesso ao sistema (*login*), copie o que o usuário digitar, finja acesso inválido e permita que o usuário tente novamente. Sem se dar conta, o usuário entregou sua senha! Uma defesa é que, se você tiver certeza de que digitou sua senha corretamente da primeira vez, deverá tentar novamente o acesso ao sistema em um terminal diferente e escolher uma nova senha imediatamente.

Sistemas UNIX incluem o comando *crypt*, que permite que um usuário entre uma chave e texto comum. A saída é texto cifrado. A transformação pode ser revertida trivialmente com a mesma chave. Um problema com essa operação é que usuários tendem a usar a mesma chave repetidamente; uma vez descoberta a chave, todos os outros arquivos criptografados com essa chave podem ser lidos. Às vezes, os usuários se esquecem de apagar seus arquivos de texto comum após produzir versões criptografadas, o que torna a descoberta da chave muito mais fácil.

Muitas vezes, o número de pessoas que recebem privilégios de superusuário é demasiadamente grande. Restringir esses privilégios pode reduzir o risco da obtenção do controle de um sistema por invasores por erros cometidos por usuários inexperientes. Sistemas UNIX fornecem um comando de identidade de usuário substituta (*su*) para habilitar usuários a executar interpretadores de comando com credenciais de usuário diferentes. Toda a atividade *su* deve ser registrada; esse comando permite que qualquer usuário que digite uma senha correta de um outro usuário assuma a identidade daquele usuário, possivelmente até mesmo adquirindo privilégios de superusuário.

Uma técnica comum de cavalo-de-tróia é instalar um programa *su* falso que obtém a senha do usuário, envia a senha ao invasor por e-mail e restaura o programa *su* regular. Nunca permita que outros tenham permissão de escrita para seus arquivos, especialmente para seus diretórios; se o fizer, está convidando alguém a instalar um cavalo-de-tróia.

Sistemas UNIX contêm uma característica denominada envelhecimento de senha (*password aging*) que permite ao administrador determinar por quanto tempo as senhas serão válidas; quando uma senha expira, o usuário recebe uma mensagem solicitando que digite uma nova senha. Essa característica apresenta vários problemas:

1. Usuários freqüentemente fornecem senhas fáceis de quebrar.

2. O sistema muitas vezes impede um usuário de recorrer à antiga (ou a qualquer outra) senha durante uma semana, para que ele não possa fortalecer uma senha fraca.

3. Usuários muitas vezes usam apenas duas senhas intercaladamente.

Senhas devem ser trocadas com freqüência. Um usuário pode monitorar todas as suas datas e horários de acesso ao sistema para determinar se um usuário não autorizado acessou o sistema (o que significa que sua senha foi descoberta). É comum que arquivos de registro de tentativas frustradas de acesso ao sistema armazenem senhas porque, às vezes, os usuários digitam accidentalmente sua senha quando a intenção era digitar o seu nome de usuário.

Alguns sistemas desabilitam contas após um pequeno número de tentativas mal sucedidas de acesso ao sistema. Essa é uma defesa contra o intruso que experimenta todas as senhas possíveis. Um intruso que tenha invadido o sistema pode usar essa característica para desabilitar a conta ou contas de usuários, incluindo a do administrador do sistema, que poderia tentar detectar a intrusão.

O invasor que ganhar superprivilegios temporariamente pode instalar um programaramadilha com características não documentadas. Por exemplo, alguém que tenha acesso ao código-fonte poderia reescrever o programa de acesso ao sistema para aceitar um determinado nome de acesso e conceder a esse usuário privilégios de superusuário sem nem mesmo digitar uma senha.

É possível que usuários individuais 'se apossem' do sistema e impeçam que outros usuários obtenham acesso. Um usuário pode conseguir isso gerando milhares de processos, cada um dos quais abre centenas de arquivos, preenchendo assim todos os espaços (slots) da tabela de arquivos abertos. Uma proteção que as instalações podem implementar para se proteger contra isso é determinar limites razoáveis para o número de processos que um pai pode gerar e para o número de arquivos que um processo pode abrir de uma só vez; porém isso, por sua vez, poderia atrapalhar usuários legítimos que precisam dos recursos adicionais.

## 15.10 EXERCÍCIOS

221) Por que segurança e proteção são importantes mesmo para computadores que não contêm dados sensíveis?

222) Qual a diferença entre segurança e proteção?

223) Considere uma cifra que reordene aleatoriamente as letras de cada palavra de uma mensagem. Por que essa cifra não é apropriada para criptografia?

224) Qual o principal ponto fraco dos algoritmos restritos?

225) Discuta as vantagens e desvantagens da criptografia por chave secreta.

226) O que limita o potencial da maioria dos algoritmos criptográficos?

227) Qual a diferença entre criptografia de chave secreta e criptografia de chave pública?

228) Dos três métodos de identificação (característica exclusiva da pessoa, propriedades de um item, conhecimento do usuário), qual deles é o menos provável de ser comprometido por intrometidos?

229) Como a segurança poderia ser comprometida em um sistema que requer somente a propriedade de um item ou apenas o conhecimento do usuário para autenticação?

230) É verdade que senhas mais longas e mais complicadas garantem maior segurança?

- 231) Como o salpicamento melhora a segurança por senhas?
- 232) Por que é difícil para um usuário não autorizado obter acesso em um sistema que usa biometria para segurança?
- 233) Explique uma desvantagem de armazenar informações do usuário não criptografadas em um cartão inteligente.
- 234) Por que a conexão entre clientes e o servidor deve ser segura na autenticação Kerberos?
- 235) Por que é adequado que os bilhetes tenham prazo de expiração?
- 236) Dos três serviços de assinatura única qual é o mais seguro?
- 237) De que modo os scripts de acesso de estações de trabalho são mais seguros do que os scripts de servidor de autenticação?
- 238) O termo *sujeito* sempre se refere a usuários de um sistema? Justifique.
- 239) Explique a diferença entre copiar direitos de acesso e propagar direitos de acesso.
- 240) Por que políticas e mecanismos de controle de acesso normalmente são separados?
- 241) Como o MAC poderia ser mais seguro do que o DAC?
- 242) Em que tipo de ambiente as listas de controle de acesso são mais apropriadas do que matrizes de controle de acesso?
- 243) Discuta as vantagens e desvantagens das capacidades por ficha e por ponteiros.
- 244) Como datas de expiração reduzem a efetividade de ataques criptoanalíticos?
- 245) Qual a diferença entre vermes e outros vírus?
- 246) Quais os pontos fracos de sistema de computadores que os vírus Melissa e ILOVEYOU expuseram?
- 247) Como os ataques DNS são prejudiciais?
- 248) Por que é difícil detectar e interromper ataques de recusa de serviço distribuído?
- 249) Por que ataques de transbordamento de buffer são perigosos?
- 250) Como ataques de transbordamento de buffer podem ser evitados?
- 251) Cite várias técnicas usadas por crackers para invadir sistemas.
- 252) Compare desfiguração da Web e ataque DNS.
- 253) É mais provável que um usuário residencial use um firewall para proibir todo o fluxo de dados não permitido expressamente ou para permitir todos os dados não proibidos expressamente?
- 254) Discuta a diferença entre firewalls de filtragem de pacotes e gateways de aplicação.

- 255) Cite uma importante desvantagem dos IDSs.
- 256) Explique a diferença entre IDSs baseados em hospedeiro e baseados em rede.
- 257) A verificação de assinatura ou a verificação heurística proporcionam melhor proteção contra vírus novos e não identificados?
- 258) Descreva diversos pontos fracos das listas de vírus conhecidos.
- 259) Por que é importante estabelecer fortes linhas de comunicação entre desenvolvedores de software e clientes na liberação de correções de segurança?
- 260) Por que é insuficiente tratar de falhas de segurança somente depois que elas foram exploradas?
- 261) Quando os mecanismos de controle de acesso são insuficientes para proteger dados de arquivos?
- 262) Qual o risco primordial de implementar um sistema de arquivos criptografado?
- 263) Um sistema que pertence à categoria C2 precisa satisfazer os requisitos do nível B?
- 264) Qual a diferença entre C2 e C1?
- 265) Para quais dos cinco requisitos fundamentais para uma transação segura, bem-sucedida, a criptografia seria útil?
- 266) Quais dos cinco requisitos fundamentais tratam do acesso do usuário ao sistema?
- 267) Por que a característica *setuid* deve ser usada com cautela?
- 268) Discuta como sistemas UNIX protegem contra acessos não autorizados ao sistema e como invasores podem evitar e explorar esses mecanismos.

-X-

# 16

## Multiprocessamento

*“Obedeça à justa medida, pois o momento certo é,  
Entre todas as coisas, o mais importante fator.”  
(Hesíodo)*

### 16.1 INTRODUÇÃO

Empresas e consumidores continuam demandando substancialmente mais capacidade de computação do que um processador pode oferecer. O resultado é que sistemas multiprocessadores – computadores que contêm mais de um processador – são empregados em muitos ambientes de computação.

Earth Simulator (mais poderoso até jun/03) usa 5120 processadores, cada um funcionando a 500Mhz. Executa 35,86 Tflops (trilhões de operações de ponto flutuante por segundo)

Assegurar:

- Todos os processadores fiquem ocupados
- Os processos sejam distribuídos equitativamente por todo o sistema
- A execução de processos relacionados seja sincronizada
- Os processadores operem sobre cópias consistentes de dados armazenados em memória compartilhada
- Seja imposta a exclusão mútua.

### 16.2 ARQUITETURA DE MULTIPROCESSADOR

Engloba qualquer sistema que contenha mais de um processador: computadores pessoais de dois processadores, servidores de alta capacidade que contêm muitos processadores e grupos distribuídos de estações de trabalho que trabalham juntas para executar tarefas.

#### 16.2.1 CLASSIFICAÇÃO DE ARQUITETURAS SEQUENCIAIS E PARALELAS

Flynn desenvolveu esquema para classificar computadores em configurações de paralelismo: consiste em quatro categorias baseadas em tipos diferentes de fluxos usados por processadores.

**Fluxo** é uma sequência de bytes alimentada em um processador. Um processador aceita dois fluxos – um fluxo de instruções e um fluxo de dados.

- **Fluxo único de instruções, fluxo único de dados** (SISD – *Single-Instruction-Stream, Single-Data-Stream*) são os monoprocessadores tradicionais nos quais um único processador busca uma instrução por vez e a executa sobre um único item de dados. Pipeline e multithread podem introduzir paralelismo em computadores SISD.

- **Fluxo múltiplo de instruções, fluxo único de dados** (MISD – *Multiple-Instruction-Stream, Single-Data-Stream*) não são comumente usados. Teria várias unidades de processamento que agiriam sobre um fluxo único de dados. Cada unidade executaria uma instrução diferente nos dados e passaria o resultado para a próxima unidade.

- **Fluxo único de instruções, fluxo múltiplo de dados** (SIMD – *Single-Instruction-Stream, Multiple-Data-Stream*) emitem instruções que agem sobre vários itens de dados. Consiste em uma ou mais unidades de processamento. Um processador executa uma instrução SIMD processando-a em um bloco de dados (por exemplo, adicionando um a todos os elementos de um arranjo). Se houver mais elementos de dados do

que unidades de processamento, essas buscarão elementos de dados adicionais para o ciclo seguinte. Isso pode melhorar o desempenho em relação às arquiteturas SISD, que exigiriam um laço para realizar a mesma operação em um elemento de dados por vez. Um laço contém muitos testes condicionais e requer que o processador SISD decodifique a mesma instrução várias vezes e que o processador SISD leia dados uma palavra por vez. Ao contrário, arquiteturas SIMD lêem um bloco de dados por vez, reduzindo dispensiosas transferências de memória para o registrador. Arquiteturas SIMD são mais efetivas em ambientes em que um sistema aplica a mesma instrução a grandes conjuntos de dados.

Computadores de **fluxo múltiplo de instruções, fluxo múltiplo de dados** (Multiple-Instruction-Stream, Multiple-Data-Stream - MIMD) são multiprocessadores nos quais as unidades processadoras são completamente independentes e operam sobre fluxos de instruções separados. Todavia, esses sistemas normalmente contêm hardware que permite que os processadores sincronizem-se uns com os outros quando necessário tal como ao acessarem um dispositivo periférico compartilhado.

### **16.2.2 ESQUEMA DE INTERCONEXÃO DE PROCESSADORES**

O esquema de interconexão de um sistema multiprocessador descreve de que modo os componentes do sistema, como um processador e módulos de memória, são conectados fisicamente. O esquema de interconexão é uma questão fundamental para projetistas de multiprocessadores porque afeta o desempenho, a confiabilidade e o custo do sistema. Um sistema de interconexão consiste em nodos e enlaces. Nodos são compostos de componentes do sistema e/ou de **chaves** que roteiam mensagens entre componentes. Um enlace é uma conexão entre dois nodos. Em muitos sistemas, um único nodo pode ter um ou mais processadores, seus caches associados, um módulo de memória e uma chave. Em multiprocessadores de grande escala, às vezes abstraímos o conceito de nodo e indicamos um grupo de nodos como um único supernodo.

Os projetistas usam diversos parâmetros para avaliar esquemas de interconexão. O **grau** de um nodo é o número de nodos ao qual ele está conectado. Eles procuram minimizar o grau de um nodo para reduzir sua complexidade e o custo de sua interface de comunicação. Nodos de graus maiores requerem hardware de comunicação mais complexo para suportar comunicação entre o nodo e seus nodos vizinhos (nodos conectados a ele).

Uma técnica para medir a tolerância à falha de um esquema de interconexão é contar o número de enlaces de comunicação que devem falhar antes que a rede não possa mais funcionar adequadamente. Isso pode ser quantificado por meio da **largura de bisseção** - o número mínimo de enlaces que precisam ser cortados para dividir a rede em duas metades não conectadas. Sistemas que têm larguras de bisseção maiores são mais tolerantes à falha do que os que têm larguras de bisseção menores, pois mais componentes têm de falhar antes que o sistema inteiro tenha problemas.

O desempenho de um esquema de interconexão depende, em grande parte, da latência de comunicação entre nodos, que pode ser medida de várias maneiras, sendo uma delas a latência média. Uma outra medição de desempenho é o **diâmetro da rede** - a distância mais curta entre os dois nodos mais remotos do esquema de interconexão. Para determinar o diâmetro da rede, considere todos os pares de nodos da rede e identifique o caminho de comprimento mais curto para cada par - calculado pela soma do número de enlaces percorridos - e então identifique o maior desses caminhos. Um diâmetro de rede pequeno indica baixa latência de comunicação e desempenho mais alto. Por fim, arquitetos de sistemas tentam minimizar o **custo de um esquema de interconexão**, semelhante ao número total de enlaces de uma rede.

Veremos agora diversos modelos bem-sucedidos de interconexão e os avaliamos com base nos critérios precedentes. Muitos sistemas reais implementam variações desses modelos. Por exemplo, eles podem agregar enlaces de comunicação extras para aumentar a tolerância à falha (aumentando a largura de bisseção) e o desempenho (reduzindo o diâmetro da rede).

#### **Barramento Compartilhado**

A organização de rede de barramento compartilhado usa um único caminho de comunicação (a rota pela qual as mensagens transitam) entre todos os processadores e módulos de memória (Figura 16.1). As interfaces de barramento dos componentes manipulam operações de transferência. O barramento é passivo e os componentes arbitram entre eles mesmos para utilizar o barramento. Somente uma transferência por vez pode ocorrer no barramento porque ele não pode transmitir dois sinais elétricos ao mesmo tempo. Portanto, antes de um componente iniciar uma transferência, ele deve verificar se o barramento e também o componente destinatário estão disponíveis. Um problema dos barramentos compartilhados - **contenção** - surge quando vários componentes querem usar o barramento ao mesmo tempo. Para reduzir contenção e tráfego no barramento, cada processador mantém seu próprio cache local, como mostra a Figura 16.1. Quando o sistema puder atender a uma requisição de memória por meio do cache de um processador, o processador não precisará se comunicar com um módulo de memória através do barramento. Uma outra opção é montar uma **arquitetura de barramentos múltiplos compartilhados**, que reduz a contenção fornecendo vários barramentos que atendem às requisições de comunicação. Contudo, esse esquema requer uma lógica complexa de arbitragem de barramento e enlaces adicionais, o que aumenta o custo do sistema.

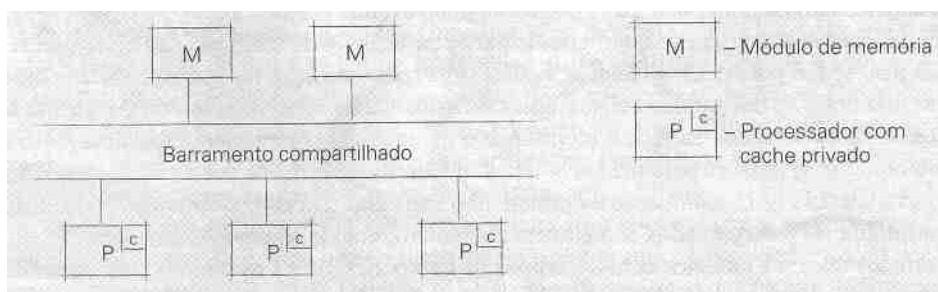


Figura 16.1: Organização de multiprocessador de barramento compartilhado

O barramento compartilhado é um esquema simples e barato para conectar um número pequeno de processadores. Novos componentes podem ser adicionados ao sistema ligando-os ao barramento, e o software manipula a detecção e a identificação dos componentes do barramento. Todavia, devido à contenção pelo único caminho de comunicação, as organizações de barramento compartilhado não podem ser escaladas para mais do que um pequeno número de processadores (na prática 16 ou 32 é o máximo). A contenção é exacerbada pelo fato de a velocidade do processador ter aumentado mais rapidamente do que a largura de banda do barramento. À medida que os processadores tomam-se mais velozes, é preciso menos processadores para saturar um barramento.

Barramentos compartilhados são redes dinâmicas, porque enlaces de comunicação são formados e descartados (por meio do barramento compartilhado) durante a execução. Portanto, os critérios utilizados para avaliar esquemas de interconexão que discutimos anteriormente não se aplicam; esses critérios se baseiam em enlaces estáticos, que não mudam durante a execução. Contudo, em comparação a outros esquemas de interconexão, um barramento compartilhado com diversos processadores é rápido e barato, mas não particularmente tolerante à falha - se o barramento compartilhado falhar, os componentes não poderão se comunicar.

Os projetistas podem alavancar os benefícios de barramentos compartilhados em multiprocessadores com um número maior de processadores. Nesses sistemas, manter um único barramento compartilhado (ou vários barramentos) que conectam todos os processadores não é prático, porque o barramento fica saturado com facilidade. Entretanto, eles podem dividir os recursos do sistema (por exemplo, processadores e memória) em diversos pequenos supernodos. Os recursos contidos em um supernodo comunicam-se via barramento compartilhado, e os supernodos são conectados usando um dos esquemas de interconexão mais escaláveis descritos nas seções seguintes. Tais sistemas tentam manter a maior parte do tráfego de comunicações dentro de um supernodo para explorar a arquitetura veloz do barramento e, ao mesmo tempo, habilitar comunicação entre supernodos. A maioria dos sistemas multiprocessadores com um pequeno número de processadores, como os sistemas de dois processadores Pentium, da Intel, usa uma arquitetura de barramento compartilhado.

### Matriz de Comutação de Barras Cruzadas

Uma matriz de comutação de barras cruzadas fornece um caminho separado de cada processador para cada módulo de memória (Figura 16.2). Por exemplo, se houver  $n$  processadores e  $m$  módulos de memória, haverá um total de  $n \times m$  comutadores, que conectam cada processador a cada módulo de memória. Podemos imaginar os processadores como as linhas da matriz, e os módulos de memória como as colunas. Em redes maiores, os nodos normalmente consistem em processadores e componentes de memória, o que melhora o desempenho de acesso à memória (para os acessos entre um processador e seu módulo de memória associado). No caso de uma matriz de comutação de barras cruzadas, isso reduz o custo do esquema de interconexão. Nesse projeto, cada nodo se conecta com um comutador de grau  $p-1$ , onde  $p$  é o número de nodos processador-memória do sistema (nesse caso  $m = n$  porque cada nodo contém o mesmo número de processadores e módulos de memória).

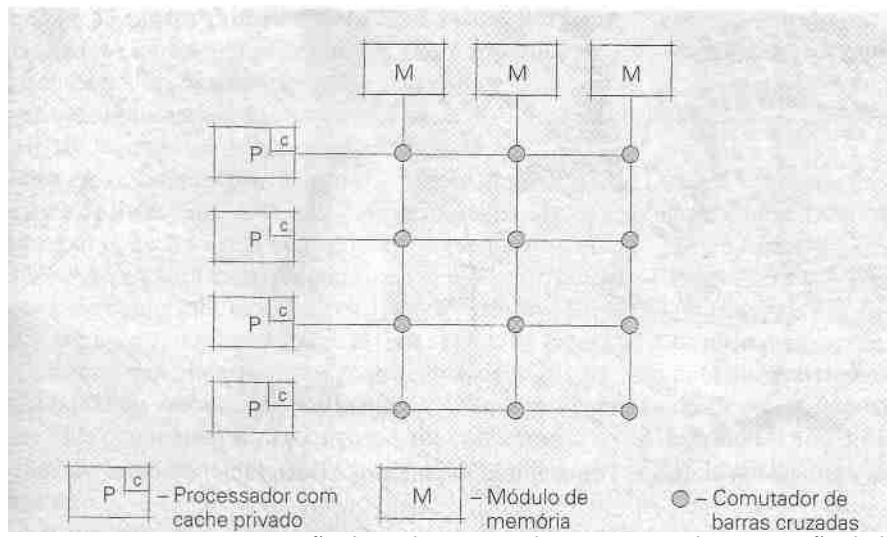


Figura 16.2: Organização de multiprocessador por matriz de comutação de barras cruzadas

Uma matriz de comutação de barras cruzadas pode suportar transmissões de dados para todos os nodos ao mesmo tempo, mas cada nodo pode aceitar, no máximo, uma mensagem por vez. Compare esse esquema com o barramento compartilhado, que suporta somente uma transmissão por vez. Um comutador usa um algoritmo de arbitragem do tipo 'atenda o processador requisitante que tiver sido atendido menos recentemente neste comutador' para resolver requisições múltiplas. O projeto de comutação de barras cruzadas fornece alto desempenho. Pelo fato de todos os nodos estarem ligados a todos os outros nodos e a transmissão através de nodos de comutação ter um custo de desempenho trivial, o diâmetro da rede mede, essencialmente, um. Cada processador está conectado a cada módulo de memória, portanto, para dividir uma matriz de comutação de barras cruzadas em duas metades iguais, é preciso que seja cortada metade dos enlaces entre processadores e módulos de memória. O número de enlaces da matriz é o produto de  $n$  por  $m$ , portanto a largura de bisseção é  $(n \times m)/2$ , resultando em forte tolerância à falha. Como a Figura 16.2 mostra, há muitos caminhos que uma comunicação pode tomar para chegar ao seu destino.

### Rede em Malha 2-D

Em um esquema de interconexão por rede em malha 2-D, cada nodo consiste em um ou mais processadores e um módulo de memória. No caso mais simples (Figura 16.3), os nodos de uma rede em malha são organizados em um retângulo de  $n$  filas e  $m$  colunas, e cada nodo é conectado aos nodos diretamente ao norte, sul, leste e oeste dele. Esse arranjo é denominado rede em malha 2-D de 4 conexões. Esse projeto mantém pequeno o grau de cada nodo, independentemente do número de processadores de um sistema - os nodos dos vértices têm grau dois, os nodos das arestas têm grau três e os nodos internos têm grau quatro. Na Figura 16.3, onde  $n = 4$  e  $m = 5$ , a rede em malha 2-D pode ser dividida em duas metades iguais cortando os cinco enlaces entre a segunda e a terceira linha de nodos. Realmente, se  $n$  for par e  $m$  ímpar, a largura de bisseção será  $m + 1$  se  $m > n$  e, do contrário, será  $n$ . Se a malha 2-D contiver um número par de

linhas e colunas, a largura de bisseção será a menor entre m e n. Embora não seja tão tolerante à falha como uma matriz de comutação de barras cruzadas, uma rede em malha 2-D é mais tolerante do que outros projetos simples, como um barramento compartilhado. Como o grau máximo de um nodo é quatro, o diâmetro de uma rede em malha 2-D será demasiadamente substancial para sistemas de grande escala. Entretanto, redes em malha têm sido usadas em grandes sistemas nos quais a comunicação ocorre principalmente entre nodos vizinhos.

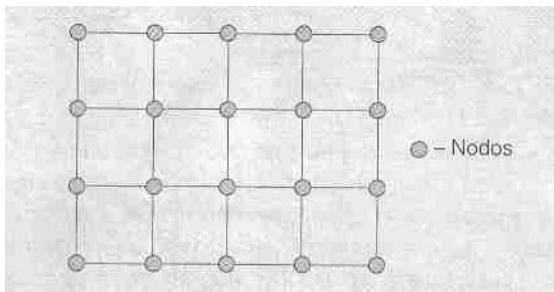


Figura 16.3: Rede em malha 2-D de 4 conexões

### Hipercubo

Um hipercubo n-dimensional consiste em  $2^n$  nodos, cada um ligado a n nodos vizinhos. Portanto, um hipercubo bidimensional é uma rede em malha  $2 \times 2$ , e um hipercubo tridimensional é conceitualmente um cubo. A Figura 16.4 ilustra conexões entre nodos em um hipercubo tridimensional (parte a) e em um hipercubo tetradiimensional (parte b). Note que um hipercubo tridimensional é, na verdade, um par de cubos bidimensionais no qual os nodos correspondentes de cada cubo bidimensional estão conectados. Similarmente, um hipercubo tetradiimensional é, na verdade, um par de hipercubos tridimensionais no qual os nodos correspondentes de cada hipercubo tridimensional estão conectados.

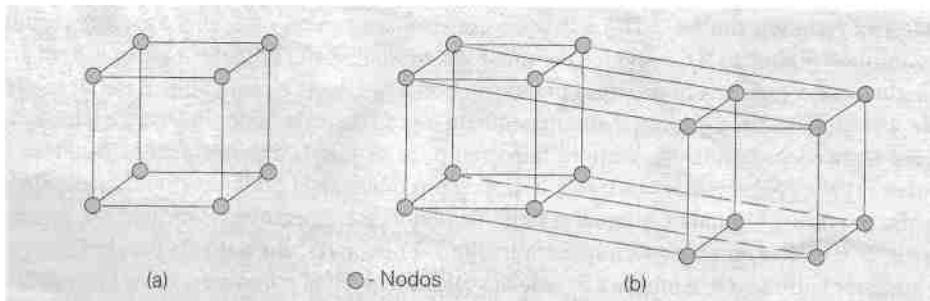


Figura 16.4: Hipercubos tridimensionais e tetradiimensionais

O desempenho de um hipercubo escala melhor do que o de uma rede em malha 2-D, pois cada nodo está conectado a outros nodos por n enlaces, o que reduz o diâmetro relativo da rede a uma rede em malha 2-D. Por exemplo, considere um multiprocessador de 16 nodos implementado ou como uma rede em malha  $4 \times 4$  ou como um hipercubo tetradiimensional. O diâmetro de uma rede em malha  $4 \times 4$  é 6, enquanto o de um cubo tetradiimensional é 4. Em alguns hipercubos, os projetistas adicionam enlaces de comunicação entre nodos não vizinhos para reduzir ainda mais o diâmetro da rede. A tolerância à falha do hipercubo também se compara favoravelmente com a de outros projetos. Todavia, o maior número de enlaces por nodo aumenta o custo de um hipercubo em relação ao de uma rede em malha.

O esquema de interconexão por hipercubo é eficiente para conectar um número modesto de processadores e é mais econômico do que uma matriz de comutação de barras cruzadas. O sistema nCUBE usado para sistemas de média de tempo real e de propaganda digital emprega hipercubos de até 13 dimensões (8.192 nodos).

### Redes Multiestágios

Um esquema alternativo de interconexão de processadores é uma rede multiestágio. Como acontece no projeto de matriz de comutação de barras cruzadas, alguns nodos são comutadores, e não nodos processadores com memória local. Os nodos de comuta-

ção são menores, mais simples e podem ser mais bem compactados, melhorando o desempenho. Para entender os benefícios de uma rede multiestágio sobre uma matriz de comutação de barras cruzadas, considere o problema de voar entre duas cidades. Em vez de oferecer vôos diretos entre cada par de cidades, as linhas aéreas usam cidades grandes como 'centrais de distribuição' (hubs). Um voo entre duas cidades pequenas normalmente consiste em diversas 'pernas', nas quais o passageiro primeiramente voa até uma central distribuidora, possivelmente viaja entre centrais distribuidoras e por fim voa até seu aeroporto de destino. Desse modo, as empresas aéreas podem programar um número menor de vôos no total e ainda assim conectar qualquer cidade pequena que tenha um aeroporto com qualquer outra. Os nodos de comutação de uma rede multiestágio agem como centrais distribuidoras para as comunicações entre processos, exatamente como os aeroportos das grandes cidades para as linhas aéreas.

Há muitos esquemas para construir uma rede multiestágio. A Figura 16.5 mostra uma rede multiestágio popular denominada **rede Ômega**. Cada nodo da esquerda é igual ao nodo da direita. Quando um processador quer se comunicar com um outro, a mensagem viaja por uma série de comutadores. O comutador mais à esquerda corresponde ao bit menos significativo (o mais à direita) do identificador (ID) do processador destinatário; o comutador do meio corresponde ao bit do meio; e o comutador mais à direita corresponde ao bit mais significativo (o mais à esquerda).

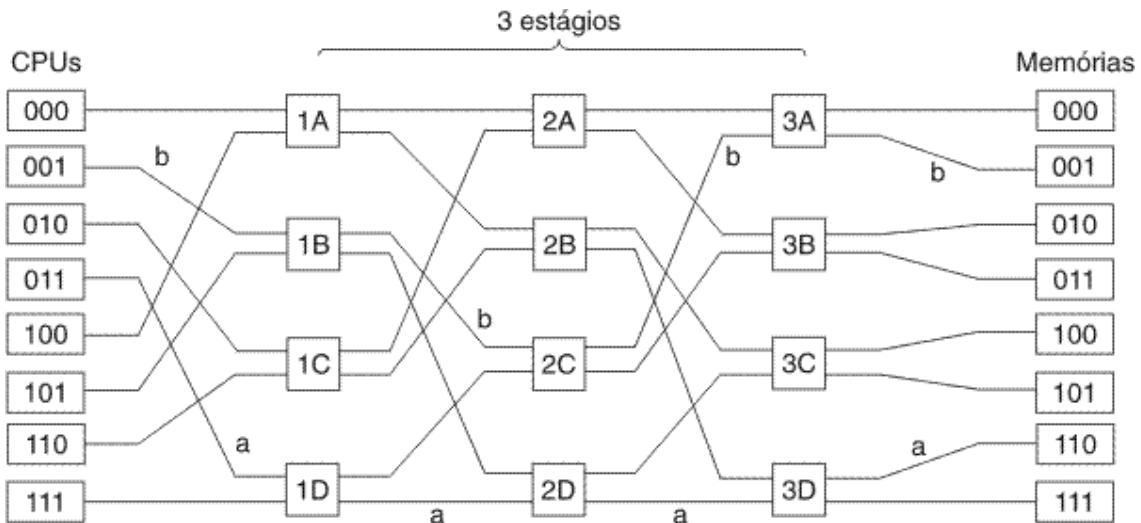


Figura 16.5: Rede Ômega.

Por exemplo, considere que a CPU 011 queira ler uma palavra do módulo de memória 110. A CPU envia uma mensagem para a chave 1D contendo 110. A chave toma o primeiro bit (mais à esquerda) de 110 e usa-o para o roteamento. Um 0 roteia para a saída superior e um 1 roteia para a saída inferior. Visto que esse bit é 1, a mensagem é roteada pela saída inferior rumo a 2D.

Todas as chaves do segundo estágio, incluindo 2D, usam o segundo bit para o roteamento. Este também é 1, de modo que a mensagem agora é encaminhada para a saída inferior rumo a 3D. Nesse estágio, verifica-se que o terceiro bit é 0. Consequentemente, a mensagem vai pela saída superior e chega à memória 110, conforme desejado. O caminho seguido por essa mensagem é marcado na Figura 16.5 pela letra a.

Redes multiestágios representam uma solução de compromisso entre custo e desempenho. Esse projeto emprega hardware simples para conectar grandes números de processadores. Qualquer processador pode se comunicar com qualquer outro sem rotear a mensagem por processadores intermediários. Entretanto, o diâmetro de uma rede multiestágio é maior, portanto, a comunicação é mais lenta do que na matriz de comutação de barras cruzadas - cada mensagem tem de passar por vários comutadores. E, também, pode se desenvolver contenção nos elementos de comutação, o que pode degradar o desempenho.

### 16.2.3 SISTEMAS FRACAMENTE x FRACAMENTE ACOPLADOS

Uma outra característica que define multiprocessadores é como eles compartilham recursos do sistema. Em um **sistema fortemente acoplado** (tightly coupled system) (Figura 16.6), processadores compartilham a maioria dos recursos do sistema. Sistemas fortemente acoplados freqüentemente empregam barramentos compartilhados, e os processadores usualmente se comunicam via memória compartilhada. Normalmente é um sistema operacional centralizado que gerencia os componentes do sistema. **Sistemas fracamente acoplados** (loosely coupled systems) (Figura 16.7) normalmente conectam componentes indiretamente por meio de enlaces de comunicação. Às vezes os processadores compartilham memória, mas muitas vezes cada processador mantém sua própria memória local, à qual o seu acesso é muito mais rápido do que ao resto da memória. Em outros casos, passar mensagens é a única forma de comunicação entre processadores, e a memória não é compartilhada.

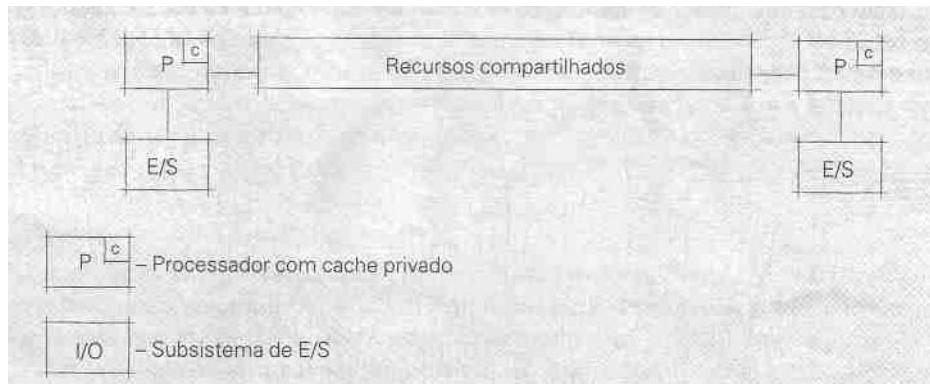


Figura 16.6: Sistema fortemente acoplado

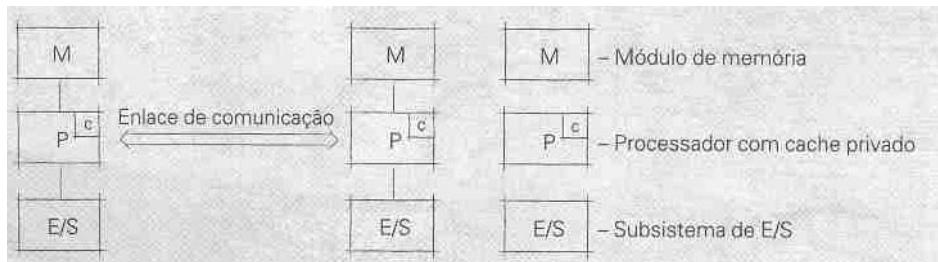


Figura 16.7: Sistema fracamente acoplado

Em geral, sistemas fracamente acoplados são mais flexíveis e escaláveis do que os fortemente acoplados. Quando os componentes são fracamente conectados, projetistas podem adicionar ou remover componentes do sistema com facilidade. A fraca acoplagem também aumenta a tolerância a falha porque os componentes podem funcionar independentemente uns dos outros. Todavia, sistemas fracamente acoplados normalmente são menos eficientes porque se comunicam passando mensagens por um enlace de comunicação, o que é mais lento do que se comunicar por meio de memória compartilhada. Isso também resulta em carga sobre os programadores de sistemas operacionais que normalmente ocultam dos programadores de aplicações a maior parte da complexidade da passagem de mensagens. O Earth Simulator do Japão é um exemplo de sistema fracamente acoplado.

Sistemas fortemente acoplados, ao contrário, executam melhor, mas são menos flexíveis. Esses sistemas não escalam bem, pois a contenção por recursos compartilhados aumenta rapidamente à medida que são adicionados processadores. Por essa razão, a maioria dos sistemas com um grande número de processadores é fracamente acoplada. Em um sistema fortemente acoplado, projetistas podem otimizar interações entre componentes para aumentar o desempenho do sistema. Contudo, isso reduz a flexibilidade e a tolerância a falha do sistema, porque um componente depende de outros componentes. Um sistema Intel Pentium de dois processadores é um exemplo de sistema fortemente acoplado.

### 16.3 ORGANIZAÇÃO DE SISTEMAS OPERACIONAIS MULTIPROCESSADORES

A organização e a estrutura de sistemas operacionais multiprocessadores são significativamente diferentes da organização e estrutura de sistemas operacionais monoprocessadores. Nesta seção categorizaremos multiprocessadores com base no modo como eles compartilham responsabilidades de sistema operacional. As organizações básicas de sistemas operacionais multiprocessadores são mestre/escravo, núcleos separados para cada processador e tratamento simétrico (ou anônimo) de todos os processadores.

### 16.3.1 MESTRE/ESCRAVO

A organização mestre/escravo de sistemas operacionais multiprocessadores designa um processador como o mestre e os outros como escravos (Figura 16.8). O mestre executa código de sistema operacional; os escravos executam somente programas usuários. O mestre executa entrada/saída e cálculos. Os escravos podem executar jobs dirigidos a processador efetivamente, mas a execução de jobs dirigidos a E/S em escravos provoca chamadas freqüentes a serviços que somente o mestre pode executar. Do ponto de vista da tolerância a falha, quando um escravo falha, há uma certa perda de capacidade de computação, mas o sistema continua a funcionar. A falha do processador mestre é catastrófica e pára o sistema. Sistemas mestre/escravo são fortemente acoplados, pois todos os processadores escravos dependem do mestre.

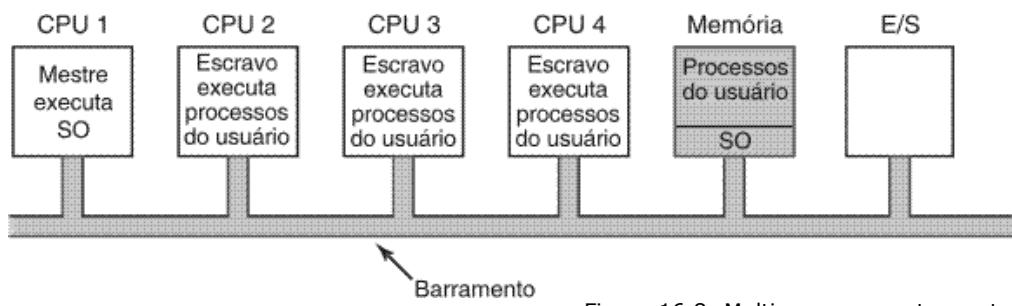


Figura 16.8: Multiprocessamento mestre/escravo

O problema primordial do multiprocessamento mestre/escravo é a assimetria de hardware, porque somente o processador mestre executa o sistema operacional. Quando um processo que está executando em um processador escravo requer a atenção do sistema operacional, o processador escravo gera uma interrupção e espera que o mestre a gerencie.

### 16.3.2 NÚCLEOS SEPARADOS

Na organização de multiprocessadores de núcleos separados, cada processador executa seu próprio sistema operacional e responde a interrupções de processos de modo usuário que estão em execução naquele processador. Um processo atribuído a um processador em particular executa naquele processador até a conclusão. Diversas estruturas de dados do sistema operacional contêm informações globais do sistema, como a lista de processos conhecidos pelo sistema. O acesso a essas estruturas de dados deve ser controlado com técnicas de exclusão mútua. Sistemas que utilizam a organização de núcleos separados são fracamente acoplados. Essa organização é mais tolerante a falha do que a organização mestre/escravo - se apenas um único processador falhar, é improvável que o sistema falhe. Todavia, os processos que estavam em execução no processador que sofreu a falha não podem ser executados até que sejam reiniciados em outro processador.

Na organização de núcleos separados, cada processador controla seus próprios recursos dedicados, como arquivos e dispositivos de E/S. Interrupções de E/S retornam diretamente aos processadores que iniciaram essas interrupções.

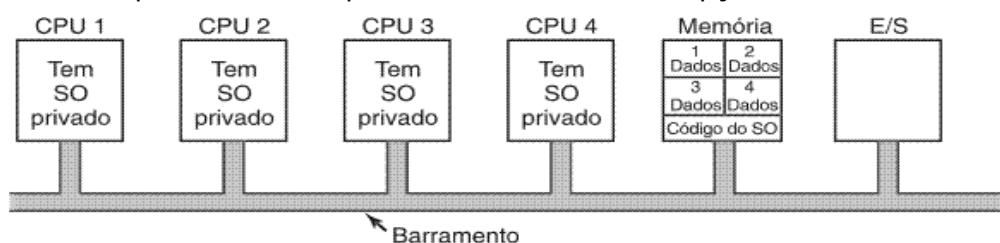


Figura 16.9: Núcleo Separado

Essa organização beneficia-se da mínima contenção pelos recursos do sistema operacional, pois os recursos são distribuídos entre os sistemas operacionais individuais para seu próprio uso. Todavia, processadores não cooperam para executar um processo individual, portanto alguns processadores podem ficar ociosos enquanto um processador executa um processo multithread.

### 16.3.3 ORGANIZAÇÃO SIMÉTRICA

A organização simétrica de multiprocessadores é a mais complexa de implementar, mas também a mais poderosa. O sistema operacional gerencia um repositório de processadores idênticos, cada um dos quais pode controlar qualquer dispositivo de E/S ou se referir a qualquer unidade de armazenamento. A simetria possibilita balancear a carga de trabalho com maior precisão do que em outras organizações.

Pelo fato de muitos processadores poderem executar o sistema operacional ao mesmo tempo, a exclusão mútua deve ser imposta sempre que o sistema operacional modificar as estruturas de dados compartilhadas. As técnicas de resolução de conflitos de hardware e software são importantes.

Organizações simétricas de sistemas multiprocessadores em geral são mais tolerantes a falha. Quando um processador falha, o sistema operacional o elimina de seu repositório de processadores disponíveis. O sistema se degrada graciosamente enquanto os reparos são realizados. E, também, um processo que esteja sendo executado em uma organização simétrica de sistema pode ser despachado para qualquer processador. Consequentemente, um processo não depende de um processador específico como acontece na organização de núcleos separados. O sistema operacional 'flutua' de um processador para o seguinte.

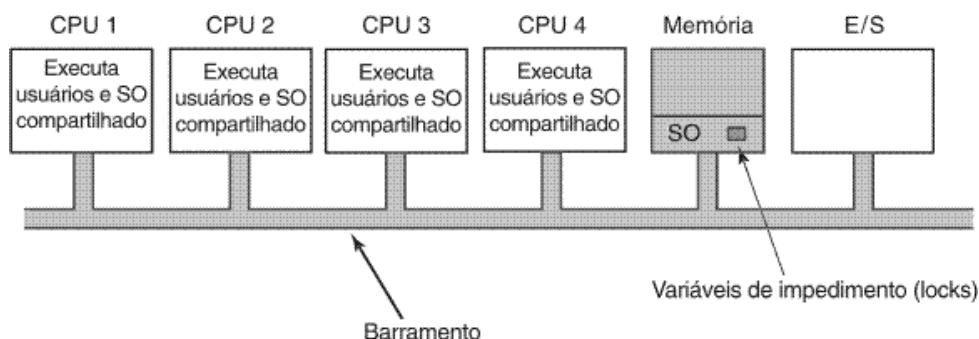


Figura 16.10: Organização Simétrica

Uma desvantagem da organização simétrica de multiprocessadores é a contenção pelos recursos do sistema operacional, como as estruturas de dados compartilhadas. Um projeto cuidadoso das estruturas de dados do sistema é essencial para evitar travamento excessivo que impeça que o sistema operacional execute em vários processadores ao mesmo tempo. Uma técnica que minimiza a contenção é dividir as estruturas de dados do sistema em entidades separadas e independentes que podem ser travadas individualmente.

Mesmo em sistemas multiprocessadores completamente simétricos, adicionar novos processadores não resulta em aumento do rendimento do sistema em virtude das capacidades nominais dos novos processadores. Há muitas razões para isso, entre elas sobrecarga adicional do sistema operacional, aumento da contenção por recursos do sistema e atrasos de hardware nas rotinas de chaveamento e roteamento entre um número maior de componentes.

## 16.4 ARQUITETURAS DE ACESSO À MEMÓRIA

Até aqui classificamos sistemas multiprocessadores segundo as características do hardware e segundo o modo como os processadores compartilham as responsabilidades do sistema operacional. Também podemos classificar sistemas multiprocessadores pelo modo como compartilham a memória. Por exemplo, considere um sistema com poucos processadores e uma pequena quantidade de memória. Se o sistema contiver um grupo de módulos de memória facilmente acessível por todos os processadores (via barramento compartilhado), poderá manter acesso rápido à memória. Todavia, sistemas que têm

muitos processadores e módulos de memória saturarão o barramento que fornece acesso a esses módulos de memória. Nesse caso, uma fração da memória pode ser vinculada a um processador de modo que ele possa acessar sua memória mais eficientemente. Portanto, projetistas devem ponderar as preocupações com desempenho, custo e escalabilidade de um sistema ao determinar a arquitetura de acesso à memória dele.

#### 16.4.1 ACESSO UNIFORME À MEMÓRIA

Arquiteturas de multiprocessador de acesso uniforme à memória (Uniform-Memory-Access multiprocessor - UMA) requerem que todos os processadores compartilhem a memória principal do sistema (Figura 16.11). Essa é uma extensão direta da arquitetura de memória de um monoprocessador, mas com vários processadores e módulos de memória. Normalmente, cada processador mantém seu próprio cache para reduzir a contenção no barramento e aumentar o desempenho. O tempo de acesso à memória é uniforme para qualquer processador que acessar qualquer item de dado, exceto quando esse estiver armazenado no cache de um processador ou quando houver contenção no barramento. Sistemas UMA também são denominados sistemas multiprocessadores simétricos (Symmetric MultiProcessor - SMP) porque qualquer processador pode ser designado para qualquer tarefa, e todos os processadores compartilham todos os recursos (incluindo memória, dispositivos de E/S e processos). Multiprocessadores UMA com um pequeno número de processadores normalmente usam uma interconexão de rede de barramento compartilhado ou de matriz de comutação de barras cruzadas. Os dispositivos de E/S são ligados diretamente à rede de interconexão e igualmente acessíveis a todos os processadores.

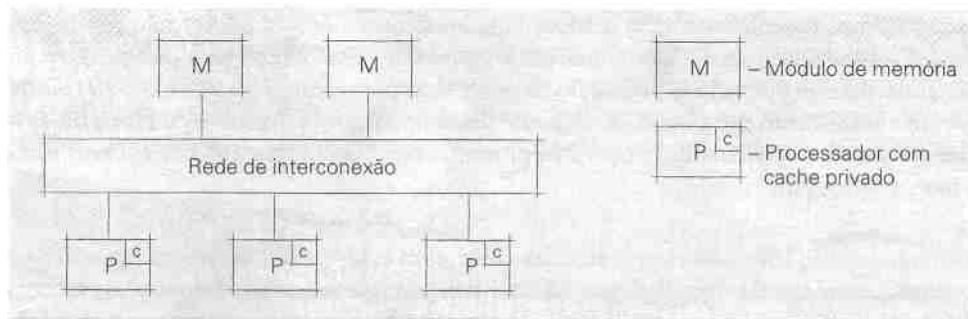


Figura 16.11: Multiprocessador UMA

Arquiteturas UMA são encontradas normalmente em sistemas multiprocessadores pequenos (de dois a oito processadores). Multiprocessadores UMA não escalam bem - um barramento torna-se rapidamente saturado quando mais do que alguns processadores acessam a memória principal simultaneamente, e matrizes de comutação de barras cruzadas ficam muito caras mesmo para sistemas de tamanhos modestos.

#### 16.4.2 ACESSO NÃO UNIFORME À MEMÓRIA

Arquiteturas de multiprocessador de acesso não uniforme à memória (NonUniform-Memory-Access - NUMA) abordam os problemas de escalabilidade da UMA. O garrote primário de um sistema UMA de grande escala é o acesso à memória compartilhada - o desempenho se degrada devido à contenção entre numerosos processadores que estão tentando acessar a memória compartilhada. Se for usada uma matriz de comutação de barras cruzadas, o custo do esquema de interconexão pode aumentar substancialmente para facilitar vários caminhos à memória compartilhada. Multiprocessadores NUMA gerenciam esses problemas relaxando a restrição de uniformidade imposta aos tempos de acesso à memória para todos os processadores que estão acessando qualquer item de dado.

Multiprocessadores NUMA mantêm uma memória global compartilhada que pode ser acessada por todos os processadores. A memória global é fracionada em módulos, e cada nodo usa um desses módulos de memória como a memória local do processador. Na Figura 16.12, cada nodo contém um processador, mas isso não é uma exigência. Embora a implementação do esquema de interconexão possa variar, os processadores são conectados diretamente a seus módulos de memória local e conectados indiretamente ao restante da memória global. Esse arranjo proporciona acesso mais rápido à memória lo-

cal do que ao restante da memória global porque o acesso à memória global requer percorrer a rede de interconexão.

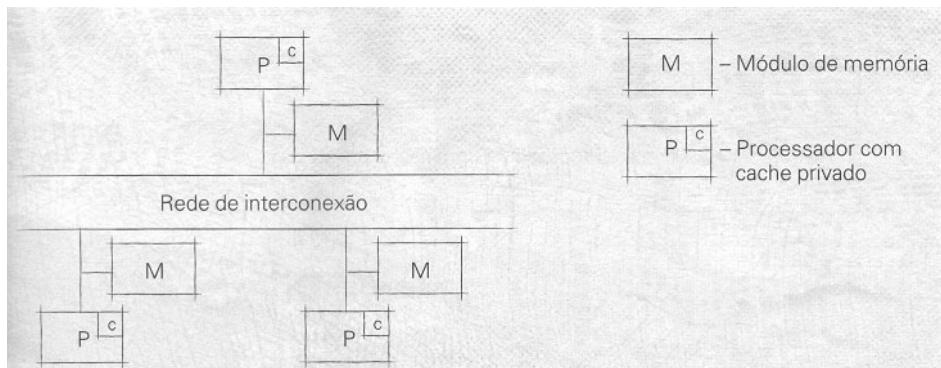


Figura 16.12: Multiprocessador NUMA

A arquitetura NUMA tem alto grau de escalabilidade, pois reduz colisões no barramento quando a memória local de um processador atende à maioria das requisições de memória do processador. Sistemas NUMA podem implementar uma estratégia que transfere páginas para o processador no qual essas páginas são acessadas mais freqüentemente - uma técnica denominada migração de páginas, que veremos posteriormente. Normalmente, sistemas NUMA podem suportar um grande número de processadores, mas seu projeto é mais complexo do que o projeto dos UMA, e a implementação de sistemas com muitos processadores pode ser cara.

#### 16.4.3 ARQUITETURA DE MEMÓRIA SOMENTE DE CACHE

Como descrito anteriormente, cada nodo de um sistema NUMA mantém sua própria memória local, a qual os processadores de outros nodos podem acessar. Muitas vezes, o acesso à memória local é radicalmente mais rápido do que o acesso à memória global (ou seja, acesso a um outro nodo da memória local). A latência de falta de cache (cache-miss latency) - o tempo requerido para recuperar dados que não estão no cache - pode ser significativa quando o dado requisitado não estiver presente na memória local. Um modo de reduzir a latência de falta de cache é reduzir o número de requisições de memória atendidas por nodos remotos. Lembre-se de que sistemas NUMA colocam dados na memória local do processador que acessa esses dados mais freqüentemente, o que não é prático para um compilador ou programador implementar, pois os padrões de acesso aos dados mudam dinamicamente. Sistemas operacionais podem executar essa tarefa, mas podem transferir somente porções de dados do tamanho de uma página, o que pode reduzir a velocidade de migração de dados. E, também, itens de dados diferentes em uma única página muitas vezes são acessados por processadores em nodos diferentes.

Multiprocessadores de arquitetura de memória somente de cache (Cache-Only Memory Architecture - COMA) usam uma ligeira variação da NUMA para abordar essa questão do posicionamento da memória (Figura 16.13). Multiprocessadores COMA têm um ou mais processadores, cada um com seu cache associado e uma fração da memória global compartilhada. Contudo, a memória associada com cada nodo é organizada como um grande cache conhecido como memória de atração (MA), o que permite que o hardware migre dados eficientemente na granularidade de uma linha de memória - equivalente a uma linha de cache, mas na memória principal e, normalmente, de quatro ou oito bytes. E, também, porque a memória local de cada processador é vista como um cache, MAs diferentes podem ter cópias da mesma linha de memória. Com essas modificações de projeto é comum que os dados residam na memória local do processador que usa esses dados mais freqüentemente, o que reduz a latência média de falta de cache. As permutas são sobrecarga de memória devido à duplicação dos itens de dados em vários módulos de memória e hardware e protocolos complicados para garantir que atualizações da memória sejam refletidas em cada MA do processador. Essa sobrecarga resulta em latência mais alta para as faltas de cache atendidas remotamente.

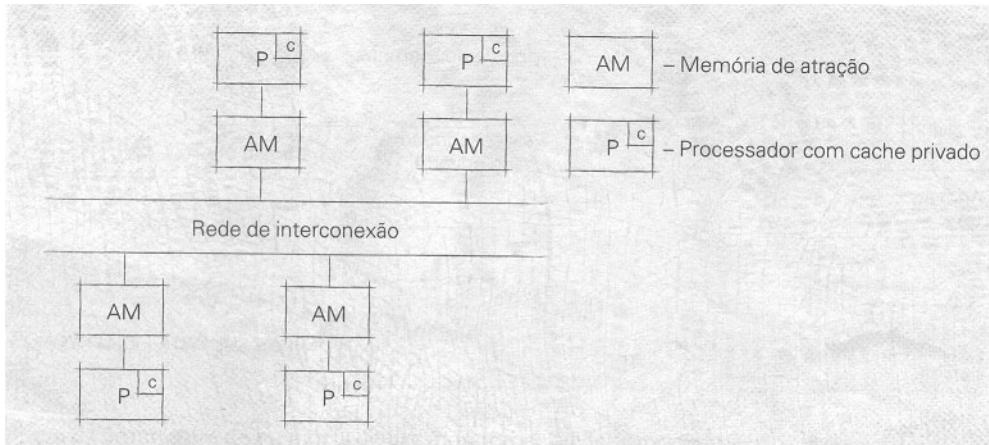


Figura 16.13: Multiprocessador COMA

#### 16.4.4 SEM ACESSO À MEMÓRIA REMOTA

Multiprocessadores UMA, NUMA e COMA são fortemente acoplados. Embora multiprocessadores NUMA (e COMA, em menor extensão) escalem bem, requerem software e hardware complexos. O software controla o acesso a recursos compartilhados como a memória; o hardware implementa o esquema de interconexão. Multiprocessadores sem acesso à memória remota (NO-Remote-Memory-Access - NORMA) são multiprocessadores fracamente acoplados que não fornecem nenhuma memória global compartilhada (Figura 16.14). Cada nodo mantém sua própria memória local, e multiprocessadores NORMA freqüentemente implementam uma memória virtual compartilhada (Shared Virtual Memory - SVM) comum. Em um sistema SVM, quando um processo requisita uma página que não está na memória local do seu processador, o sistema operacional carrega a página na memória local por meio de um outro módulo de memória (de um computador remoto através de uma rede) ou do armazenamento secundário (por exemplo, um disco). Nodos de sistemas NORMA que não suportam SVM devem compartilhar dados por meio de passagem de mensagens explícita. O Google, que alimenta seu serviço usando 15 mil servidores baratos localizados no mundo inteiro, é um exemplo de sistema multiprocessador distribuído NORMA.

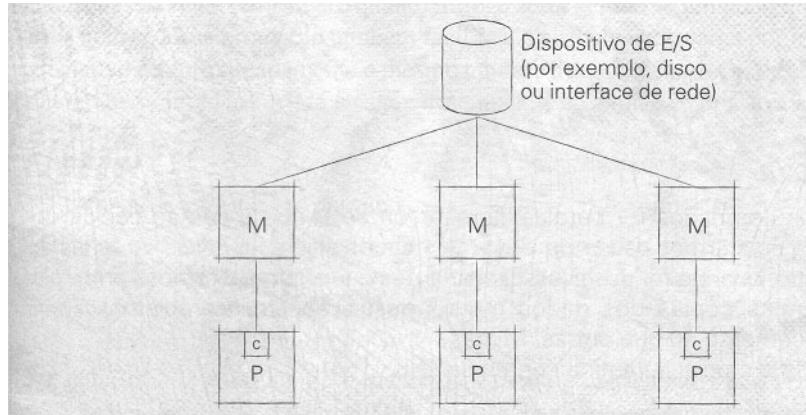


Figura 16.14: Multiprocessador NORMA

Sistemas NORMA são os mais simples de construir, porque não requerem um sistema de interconexão complexo. Todavia, a ausência de memória global compartilhada requer que programadores de aplicações implementem IPC via passagem de mensagem e chamadas remotas de procedimentos. Há muitos sistemas em que usar memória virtual compartilhada é ineficiente, pois o sistema teria de enviar páginas inteiras de dados de um processador para o seguinte, e esses processadores nem sempre estariam na mesma máquina física.

Como os multiprocessadores NORMA são fracamente acoplados, é relativamente fácil remover ou adicionar nodos. Esses multiprocessadores são sistemas distribuídos regidos por um único sistema operacional, em vez de redes de computadores com seu próprio sistema operacional. Se um nodo de um sistema NORMA falhar, basta o usuário

comutar para outro nodo e continuar trabalhando. Se 1% dos nodos falharem em um multiprocessador NORMA, o sistema só executará 1% mais lentamente.

## 16.5 COMPARTILHAMENTO DE MEMÓRIA EM MULTIPROCESSADORES

Quando vários processadores com caches privados ou memórias locais acessam memória compartilhada, os projetistas têm de abordar o problema da coerência de memória. A memória é coerente se o valor obtido da leitura de um endereço de memória for sempre o valor escrito mais recentemente para aquele endereço. Por exemplo, considere dois processadores que mantêm uma cópia separada da mesma página em suas memórias locais. Se um processador modificar sua cópia local, o sistema deverá garantir que a outra cópia da página seja atualizada para refletir as últimas mudanças. Multiprocessadores que permitem que somente uma cópia de página resida no sistema por vez (como sistemas UMA e sistemas NUMA pequenos) ainda assim devem garantir coerência de cache - ler uma entrada de cache reflete a atualização mais recente daqueles dados. Sistemas de grande escala (como sistemas NORMA ou sistemas NUMA de grande escala) muitas vezes permitem que a mesma página de memória resida na memória local de diversos processadores. Esses sistemas devem integrar coerência de memória em sua implementação de memória virtual compartilhada.

Outras importantes considerações de projeto são posicionamento e replicação de páginas. O acesso à memória local é muito mais rápido do que o acesso à memória global, portanto, garantir que os dados acessados por um processador residam na memória local daquele processador pode melhorar o desempenho. Há duas estratégias comuns para abordar essa questão. A replicação de página mantém várias cópias de uma página da memória, de modo que ela possa ser acessada rapidamente em vários nodos. A migração de página transfere páginas para o nodo (ou nodos, quando usada com replicação de página) no qual os processadores mais acessam uma página. Posteriormente, consideraremos implementações dessas duas estratégias.

### 16.5.1 COERÊNCIA DE CACHE

A coerência de memória passou a ser uma consideração de projeto quando surgiram os caches, porque as arquiteturas de computador permitiam caminhos de acesso diferentes aos dados (ou seja, por meio da cópia do cache ou da cópia da memória principal). Em sistemas multiprocessadores a coerência é complicada pelo fato de que cada processador mantém um cache privado.

#### **Coerência de cache UMA**

Implementar protocolos de coerência de cache para multiprocessadores UMA é simples porque os caches são relativamente pequenos e o barramento que conecta a memória compartilhada é relativamente rápido. Quando um processador atualiza um item de dado, o sistema também deve atualizar ou descartar todas as instâncias daquele dado nos caches de outros processadores e na memória principal, o que pode ser realizado por escuta do barramento (também denominado escuta do cache). Nesse protocolo um processador 'escuta' o barramento determinando se uma escrita requisitada de um outro processador é para um item de dado que está no cache do processador. Se o dado residir no cache do processador, esse remove o item de dado do seu cache. A escuta de barramento é simples de implementar, mas gera tráfego adicional no barramento compartilhado. Como alternativa, o sistema pode manter um diretório centralizado que registre os itens que residem em cada cache e indique quando remover dados envelhecidos (dados que não refletem a atualização mais recente) de um cache. Uma outra opção é o sistema permitir que somente um processador faça o cache de um determinado item da memória.

#### **NUMA com cache coerente (CC-NUMA)**

NUMAs com cache coerente (Cache-Coherent NUMAs -CC-NUMAs) são multiprocessadores NUMA que impõem coerência de cache. Em uma arquitetura CC-NUMA típica, cada endereço de memória física está associado a um nodo nativo responsável por armazenar o item de dado com aquele endereço de memória principal. (Muitas vezes o nodo nativo é simplesmente determinado pelos bits de ordem mais alta do endereço). Quando ocorrer uma falta de cache em um nodo, este contacta o nodo hospedeiro associado ao

endereço de memória requisitado. Se o item de dado estiver limpo (se nenhum outro nodo tiver uma versão modificada do item de dado em seu cache), o nodo nativo o despachará para o cache do processador requisitante. Se o item de dado estiver sujo (se um outro nodo escreveu para o item de dado desde a última vez que a entrada da memória principal foi atualizada), o nodo nativo despachará a requisição para o nodo que tem a cópia suja; esse nodo envia o item de dado para o requisitante e também para o nodo nativo. Similarmente, requisições para modificar dados são realizadas via nodo nativo. O nodo que desejar modificar dados em determinado endereço de memória requisita propriedade exclusiva dos dados. A versão mais recente dos dados (se já não estiver no cache do nodo modificador) é obtida da mesma maneira que uma requisição de leitura. Após a modificação, o nodo nativo notifica a outros nodos com cópias dos dados que os dados foram modificados.

Esse protocolo é relativamente simples de implementar porque todas as leituras e escritas contactam primeiramente o nodo nativo e, embora possa parecer ineficiente, esse protocolo de coerência requer o número máximo de apenas três comunicações de rede. (Considere quanto tráfego seria gerado se um nodo que está escrevendo tivesse de contactar todos os outros nodos.) Esse protocolo também facilita a distribuição de carga por todo o sistema - designando cada nodo como o nodo nativo de aproximadamente o mesmo número de endereços -, o que aumenta a tolerância a falha e reduz a contenção. Contudo, esse protocolo pode ter mau desempenho se a maioria dos acessos aos dados vier de nodos remotos.

### **16.5.2 REPLICACÃO E MIGRAÇÃO DE PÁGINAS**

A latência de acesso à memória de sistemas NUMA é mais alta do que a de multiprocessadores UMA, o que limita o desempenho do NUMA. Portanto, maximizar o número de faltas de cache atendidas pela memória local é uma consideração importante do projeto do NUMA. O projeto do COMA é uma tentativa de resolver a questão da latência do NUMA. Sistemas CC-NUMA abordam a questão da latência implementando estratégias de migração ou de replicação de páginas.

Replicar uma página é uma operação direta. O sistema copia todos os dados de uma página remota para uma página da memória local do processador requisitante. Para manter coerência de memória, o sistema usa uma estrutura de dados que registra onde estão todas as páginas replicadas. A migração de páginas ocorre de um modo similar, exceto que, depois de uma página ter sido replicada, o nodo original apaga a página da sua memória e descarrega qualquer TLB ou entradas de cache associadas.

Embora migrar e replicar páginas proporcione benefícios óbvios, essas estratégias podem degradar o desempenho se não forem implementadas corretamente. Por exemplo, referir-se a uma página remotamente é mais rápido do que migrar ou replicar aquela página. Portanto, se um processo se referir a uma página remota apenas uma vez, será mais eficiente não migrar nem replicar a página. Além disso, algumas páginas são melhores candidatas para replicação do que para migração e vice-versa. Por exemplo, páginas lidas freqüentemente por processos em diferentes processadores seriam beneficiadas pela replicação, pois todos os processos ficariam habilitados a acessar aquela página da memória local. E, também, a desvantagem da replicação - manter coerência nas escritas - não seria problema para uma página somente de leitura. Uma página freqüentemente modificada por um processo é uma boa candidata para migração; se outros processos estiverem lendo daquela página não seria viável replicá-la porque esses processos teriam de buscar atualizações continuamente no nodo que está escrevendo. Páginas escritas freqüentemente por processos em mais de um processador não são boas candidatas nem para replicação nem para migração, pois elas serão migradas ou replicadas após cada operação de escrita.

Para ajudar a determinar a melhor estratégia, muitos sistemas mantêm o histórico de informações de acesso de cada página. Um sistema poderia usar essas informações para determinar quais páginas são acessadas freqüentemente - essas devem ser consideradas para replicação ou migração. A sobrecarga envolvida na replicação e na migração de uma página não se justifica para páginas que não são acessadas com freqüência. Um sistema também poderia manter informações sobre quais processadores remotos

estão acessando a página e se esses processadores estão lendo a página ou escrevendo para ela. Quanto mais informações sobre o acesso às páginas o sistema reunir, melhores decisões poderá tomar. Contudo, reunir esse histórico e transferi-lo durante a migração incorre em sobrecarga. Assim, o sistema deve coletar informações suficientes para tomar boas decisões de replicação ou migração sem provocar sobrecarga excessiva da manutenção dessas informações.

### **16.5.3 MEMÓRIA VIRTUAL COMPARTILHADA**

Compartilhar memória em multiprocessadores pequenos, fortemente acoplados, como um multiprocessador UMA, é uma extensão direta do compartilhamento de memória em monoprocessadores, pois todos os processadores acessam os mesmos endereços físicos com igual latência de acesso. Essa estratégia é inviável para multiprocessadores NUMA de grande escala devido à latência de acesso à memória remota, e é impossível para multiprocessadores NORMA, que não compartilham memória física. Como a IPC por meio de memória compartilhada é mais fácil do que a IPC via troca de mensagem, muitos sistemas habilitam processos a compartilhar memória localizada em diferentes nodos (e talvez em diferentes espaços de endereçamento físico) pela memória virtual compartilhada (Shared Virtual Memory - SVM). A SVM estende conceitos de memória virtual de monoprocessador garantindo coerência de memória para páginas acessadas por vários processadores. Duas questões que se apresentam aos projetistas de SVM são selecionar qual protocolo de coerência usar e quando aplicá-lo. Os dois protocolos primordiais de coerência são a invalidação e a difusão de escrita. Primeiramente descreveremos esses protocolos e, então, consideraremos como implementá-los.

#### **Invalidate**

Na abordagem da invalidação da coerência de memória somente um processador pode acessar a página enquanto ela está sendo modificada - o que é denominado propriedade de página. Para obter propriedade de página, um processador deve primeiro invalidar (negar acesso a) todas as outras cópias da página. Após o processador obter a propriedade, o modo de acesso da página é modificado para leitura/escrita e o processador copia a página para sua memória local antes de acessá-la. Para obter acesso de leitura a uma página, um processador deve requisitar que o processador que tem a propriedade de acesso de leitura/escrita modifique seu modo de acesso para somente leitura. Se a requisição for concedida, outros processos podem copiar a página para a memória local e lê-la. Normalmente o sistema emprega a política de sempre conceder requisições, a menos que um processador esteja esperando para obter a propriedade da página; no caso da negação de uma requisição, o requisitante deve esperar até que a página esteja disponível. Note que vários processos que modificam uma única página concorrentemente resultam em mau desempenho porque processadores que estão na disputa invalidam repetidamente as cópias da página de outros processadores.

#### **Difusão de Escrita**

Na abordagem de coerência de memória por difusão de escrita o processador que está escrevendo comunica cada modificação a todo o sistema. Em uma das versões dessa técnica somente um processador pode obter a propriedade de uma página. Em vez de invalidar todas as outras cópias da página existentes, o processador atualiza todas essas outras cópias. Em uma segunda versão, diversos processadores obtêm acesso de escrita para aumentar a eficiência. Como esse esquema não requer que escritores obtenham propriedade de escrita, processadores devem assegurar que as diversas atualizações de uma página sejam aplicadas na ordem correta, o que pode incorrer em sobrecarga significativa.

#### **Implementação de Protocolos de Coerência**

Há diversas maneiras de implementar um protocolo de coerência de memória, embora os projetistas devam tentar limitar o tráfego de acessos à memória no barramento. Garantir que cada escrita seja refletida para todos os nodos logo que possível pode reduzir o desempenho; todavia, relaxar as restrições de coerência pode produzir resultados errôneos se os programas usarem dados ultrapassados. Em geral, uma implementação deve equilibrar desempenho com integridade dos dados.

Sistemas que usam **consistência seqüencial** asseguram que todas as escritas sejam imediatamente refletidas em todas as cópias da página. Esse esquema não escala bem para sistemas grandes devido ao custo de comunicação e ao tamanho da página. A comunicação internodos é lenta em comparação com o acesso à memória local; se um processo atualizar repetidamente a mesma página, o resultado será muitas comunicações desperdiçadas. E também, porque o sistema operacional manipula páginas, as quais muitas vezes contêm muitos itens de dados, essa estratégia poderia resultar em **falso compartilhamento**, no qual processos que estejam sendo executados em nós separados poderiam precisar de acesso a dados não relacionados na mesma página. Nesse caso, a consistência seqüencial efetivamente requer que dois processos compartilhem a página, mesmo que suas modificações não afetem um ao outro.

Sistemas que utilizam **consistência relaxada** executam operações de coerência periodicamente. A premissa por trás dessa estratégia é que o usuário não perceberá se a coerência para dados remotos sofrer um atraso de alguns segundos e ela pode aumentar o desempenho e reduzir o efeito de falso compartilhamento.

## 16.6 ESCALONAMENTO DE MULTIPROCESSADORES

As metas do escalonamento de multiprocessadores são as mesmas do escalonamento de monoprocessadores - o sistema tenta maximizar rendimento e minimizar tempos de resposta para todos os processos. Além disso, ele deve impor prioridade de escalonamento.

Diferentemente dos algoritmos de escalonamento de monoprocessadores, que determinam somente em que ordem os processos são despachados, os algoritmos de escalonamento de multiprocessadores devem assegurar que os processadores não fiquem ociosos enquanto estão esperando que os processos executem.

Ao determinar o processador no qual um processo é executado, o escalonador considera diversos fatores. Por exemplo, algumas estratégias focalizam o máximo paralelismo em um sistema para explorar a concorrência de aplicações. Sistemas freqüentemente agrupam processos colaboradores em um job. Executar os processos de um job em paralelo melhora o desempenho, habilitando esses processos a executar verdadeiramente de maneira simultânea. Esta seção apresenta diversos algoritmos de **escalonamento de compartilhamento de tempo** que tentam explorar tal paralelismo escalonando processos colaborativos em diferentes processadores, o que habilita o processo a sincronizar sua execução concorrente e mais efetivamente.

Outras estratégias focalizam a **afinidade de processador** - a relação de um processo com um processador particular e sua memória local e em cache. Um processo que exibe alta afinidade de processador executa no mesmo processador durante quase todo, ou todo, o seu ciclo de vida. A vantagem é que o processo experimentará mais acertos no cache, e, no caso de um projeto NUMA ou NORMA, possivelmente menos acessos a páginas remotas do que se executasse em diversos processadores durante seu ciclo de vida. Algoritmos de escalonamento que tentam escalonar um processo no mesmo processador durante todo o seu ciclo de vida mantêm uma **afinidade flexível**, ao passo que algoritmos que escalonam um processo somente em um processador mantêm **afinidade restrita**.

Algoritmos de **escalonamento por partição de espaço** tentam maximizar a afinidade do processador escalonando processos colaborativos em um único processador (ou um único conjunto de processadores) sob a premissa de que processos colaborativos acessarão os mesmos dados compartilhados, que provavelmente estão armazenados nos caches e na memória local do processador. Portanto, escalonamento por partição de espaço aumenta acertos de cache e de memória local. Todavia, pode limitar o rendimento, porque esses processos normalmente não executam simultaneamente.

Algoritmos de escalonamento de multiprocessadores em geral são classificados como cegos aos jobs ou cientes de jobs. Políticas de **escalonamento cego ao job** incorrem em sobrecarga mínima de escalonamento porque não tentam melhorar o paralelismo de um job ou a afinidade do processador. **Escalonamento ciente de job** avalia as pro-

priedades do job e tenta maximizar o paralelismo de cada job ou a afinidade do processador, o que aumenta o desempenho ao custo do aumento de sobrecarga.

Muitos algoritmos de escalonamento de multiprocessadores organizam processos em **filas globais de execução**. Cada fila global de execução contém todos os processos do sistema que estão prontos para executar. Essas filas podem ser utilizadas para organizar processos por prioridade, por job ou pelos processos que executaram mais recentemente.

Como alternativa, sistemas podem utilizar uma **fila de execução por processador**, o que é típico de sistemas grandes, fracamente acoplados (como sistemas NORMA), nos quais os acertos de cache e as referências à memória principal devem ser maximizadas. Nesse caso, processos são associados a um processador específico e o sistema implementa uma política de escalonamento para aquele processador. Alguns sistemas usam **filas de execução por nodo**. Cada nodo poderia conter mais do que um processador, o que é apropriado para um sistema no qual um processo está vinculado a um grupo particular de processadores. Descreveremos a questão relacionada à migração de processos, que acarreta a transferência de processos de uma fila por processador ou por nodo para outra.

### **16.6.1 ESCALONAMENTO DE MULTIPROCESSADORES CEGOS AO JOB**

Algoritmos de escalonamento de multiprocessadores cegos ao job escalonam jobs ou processos em qualquer processador disponível. Os três algoritmos descritos nesta seção são exemplos de algoritmos de escalonamento de processadores cegos ao job. Em geral, qualquer algoritmo de escalonamento de multiprocessadores, como os descritos em capítulos anteriores, pode ser implementado como um algoritmo de escalonamento de multiprocessadores cego ao job.

#### **Escalonamento de Processo FCFS**

O escalonamento de processo primeiro-a-chegar-primeiro-a-ser-atendido (First-Come-First-Served - FCFS) coloca processos que chegam em uma fila global de execução. Quando um processador torna-se disponível, o escalonador despacha o processo que está no início da fila e o executa até que ele libere o processador.

FCFS trata todos os processos com justiça, escalonando-os de acordo com seus horários de chegada. Contudo, o FCFS poderia ser considerado injusto, pois processos longos fazem com que processos curtos esperem, e processos de baixa prioridade podem fazer com que processos de alta prioridade esperem - embora uma versão preemptiva do FCFS possa impedir que isso ocorra. Normalmente o FCFS não é útil para processos interativos porque não pode garantir tempos de respostas curtos. Contudo, é um algoritmo fácil de implementar e elimina a possibilidade de adiamento indefinido - uma vez que um processo entre em uma fila, nenhum outro processo entrará na fila à frente dele.

#### **Escalonamento de Multiprocessadores por Alternância Circular (RRprocess)**

O escalonamento de processo por alternância-circular (Round-Robin process - RR-process) coloca cada processo pronto em uma fila global de execução. O escalonamento RRprocess é semelhante ao escalonamento por revezamento de tempo de monoprocessadores - um processo executa durante no máximo um período antes de o escalonador despachar um novo processo para execução. O processo que estava executando anteriormente é colocado no final da fila global de execução. O algoritmo impede adiamento indefinido, mas não promove um alto grau de paralelismo ou de afinidade de processador, pois ignora as relações entre processos.

#### **Escalonamento de Multiprocessadores Processo-mais-curto-primeiro (SJF)**

Um sistema também pode implementar o algoritmo de escalonamento processo-mais-curto-primeiro (Shortest Job First - SJF), que despacha o processo que requer a menor quantidade de tempo para executar até concluir. Ambas as versões do SJF, preemptiva e não preemptiva, exibem médias mais baixas de tempo de espera para processos interativos em relação ao FCFS, porque processos interativos normalmente são processos 'curtos'. Contudo, um processo mais longo pode ser adiado indefinidamente se processos mais curtos chegarem continuamente antes que ele possa obter um processa-

dor. Como acontece com algoritmos cegos ao job, o SJF não considera paralelismo nem afinidade de processador.

### **16.6.2 ESCALONAMENTO DE MULTIPROCESSADORES CIENTES AO JOB**

Embora algoritmos cegos ao job sejam fáceis de implementar e incorram em sobrecarga mínima, eles não consideram questões de desempenho específicas do escalonamento de multiprocessadores. Por exemplo, se dois processos que se comunicam com freqüência não executarem simultaneamente, podem gastar significativa quantidade de tempo em espera ociosa, o que degradará o desempenho geral do sistema. Além do mais, na maioria dos sistemas multiprocessadores cada processador mantém seu próprio cache privado. Processos de um mesmo job muitas vezes acessam os mesmos itens de memória, portanto, escalar os processos de um job no mesmo processador tende a aumentar os acertos de cache e melhorar o desempenho do acesso à memória. Em geral, algoritmos de escalonamento de processos cientes de job tentam maximizar paralelismo ou afinidade de processador à custa de maior complexidade do algoritmo de escalonamento.

#### **Escalonamento de Menor-número-de-processos-primeiro (SNPF)**

O algoritmo de escalonamento menor-número-de-processos-primeiro (Smallest-Number-of-Processes-First - SNPF), que pode ser preemptivo ou não preemptivo, usa uma fila global de prioridade de jobs. A prioridade de um job é inversamente proporcional ao seu número de processos. Se os jobs contiverem o mesmo número de processos disputando um processador, aquele que estiver esperando há mais tempo receberá prioridade. No escalonamento SNPF não preemptivo, quando um processador fica disponível, o escalonador seleciona um processo do job que está no início da fila e permite que ele execute até a conclusão. No escalonamento SNPF preemptivo, se chegar um novo job com menos processos, ele receberá prioridade e seus processos serão despachados imediatamente. Algoritmos SNPF melhoram o paralelismo porque processos que estão associados ao mesmo job muitas vezes podem executar concorrentemente. Contudo, os algoritmos SNPF não tentam melhorar a afinidade de processador. Além disso, é possível que jobs com muitos processos sejam adiados indefinidamente.

#### **Escalonamento por alternância circular de jobs (RRJob)**

O escalonamento por alternância circular de jobs (Round-Robin Job - RRJob) emprega uma fila global de jobs na qual cada job é designado a um grupo de processadores (embora não necessariamente o mesmo grupo cada vez que o job for escalonado). Cada job mantém sua própria fila de processos. Se o sistema contiver p processadores e usar um quantum de tamanho q, um job receberá um total p x q de tempo de processador quando for despachado. Normalmente um job não contém exatamente p processos que exaurem um quantum cada um (por exemplo, um processo pode bloquear antes que seu quantum expire). Portanto, RRJobs usam escalonamento de alternância circular para despachar os processos do job até que este consuma todos os quanta p x q, ou conclua, ou todos os seus processos bloqueiem. O algoritmo também pode dividir os quanta p x q igualmente entre os processos do job, permitindo que cada um execute até exaurir seu quantum, concluir ou bloquear. Como alternativa, se um job tiver mais do que p processos, ele poderá selecionar p processos para executar durante um quantum de tamanho q.

Similarmente ao escalonamento RRprocess, esse algoritmo evita adiamento indefinido. Além disso, como os processos do mesmo job executam concorrentemente, esse algoritmo promove paralelismo. Todavia, a sobrecarga adicional de chaveamento de contexto do escalonamento de alternância circular pode reduzir o rendimento de jobs.

#### **Coescalonamento**

Algoritmos de coescalonamento (ou **escalonamento de bando**) empregam uma fila global de execução que é acessada à maneira da alternância circular. O objetivo dos algoritmos de coescalonamento é executar processos do mesmo job concorrentemente em vez de maximizar a afinidade de processador. Há várias implementações de coescalonamento: matricial, contínuo e não dividido. Apresentaremos somente o algoritmo não dividido porque ele corrige algumas das deficiências dos algoritmos matricial e contínuo.

O **algoritmo de coescalonamento não dividido** coloca processos do mesmo job em entradas adjacentes na fila global de execução (Figura 16.15). O escalonador mantém uma 'janela' igual ao número de processadores do sistema. Todos os processadores em uma janela executam em paralelo por, no mínimo, um quantum. Após escalarar um grupo de processos, a janela passa para o próximo grupo de processos, que também executa em paralelo durante um quantum. Para maximizar a utilização do processador, se um processo da janela for suspenso, o algoritmo estenderá a janela corredíga um processo para a direita para permitir que outro processo executável execute durante o dado quantum, mesmo que ele não faça parte do job que esteja executando no momento.

Como algoritmos de coescalonamento usam uma estratégia de alternância circular, eles impedem adiamento indefinido. Além disso, como os processos do mesmo job com freqüência executam ao mesmo tempo, os algoritmos de coescalonamento permitem que sejam projetados programas para executar em paralelo e tirar proveito de um ambiente de multiprocessamento. Infelizmente, um processo pode ser despachado para um processador diferente a cada vez, o que pode reduzir a afinidade de processador.

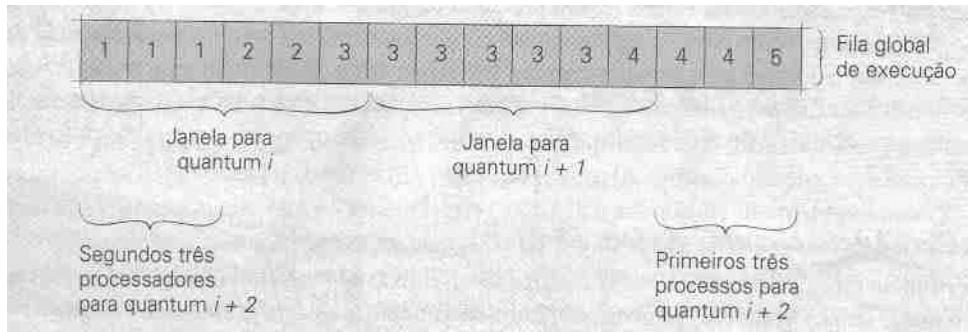


Figura 16.15: Coescalonamento (versão não dividido)

### Partição dinâmica

A partição dinâmica minimiza a penalidade do desempenho associada a faltas de cache mantendo alta afinidade de processador. O escalonador distribui processadores do sistema eqüitativamente entre jobs. O número de processadores alocados a um job é sempre menor ou igual ao número de processos executáveis do job.

Por exemplo, considere um sistema que contenha 32 processadores e execute três jobs - o primeiro com 8 processos executáveis, o segundo com 16 e o terceiro com 20. Se o escalonador dividisse os processadores eqüitativamente entre jobs, um job receberia 10 processadores e os outros dois receberiam 11. No caso em questão, o primeiro job tem somente 8 processos, portanto o escalonador designa 8 processadores àquele job e distribui eqüitativamente os 24 processadores remanescentes (12 para cada) ao segundo e terceiro jobs (Exemplo 1 da Figura 16.16). Portanto, um certo job sempre executa em um certo subconjunto de processadores, desde que nenhum job novo entre no sistema. O algoritmo pode ser estendido de modo que um determinado processo sempre execute no mesmo processador. Se cada job contiver apenas um único processo, a partição dinâmica irá se reduzir a um algoritmo de escalonamento de alternância circular.

À medida que novos jobs entram no sistema, esse atualiza dinamicamente a alocação do processador. Suponha que um quarto job com 10 processos executáveis (Exemplo 2 da Figura 16.16) entre no sistema, o primeiro job retém sua alocação de 8 processadores, mas o segundo e o terceiro jobs devolvem 4 processadores cada para o quarto job. Assim, os processadores são divididos eqüitativamente entre os jobs - 8 processadores por job. O algoritmo atualiza o número de processadores que cada job recebe sempre que jobs entrem ou saiam do sistema, ou um processo dentro de um job mude de estado em execução para o estado de espera ou vice-versa. Embora o número de processadores alocados a jobs mude, um job ainda assim executa ou em um subconjunto ou em um superconjunto de sua alocação prévia, o que ajuda a manter afinidade de processador. Para que a partição dinâmica seja efetiva, o aumento do desempenho pela afinidade de cache deve compensar o custo da repartição.

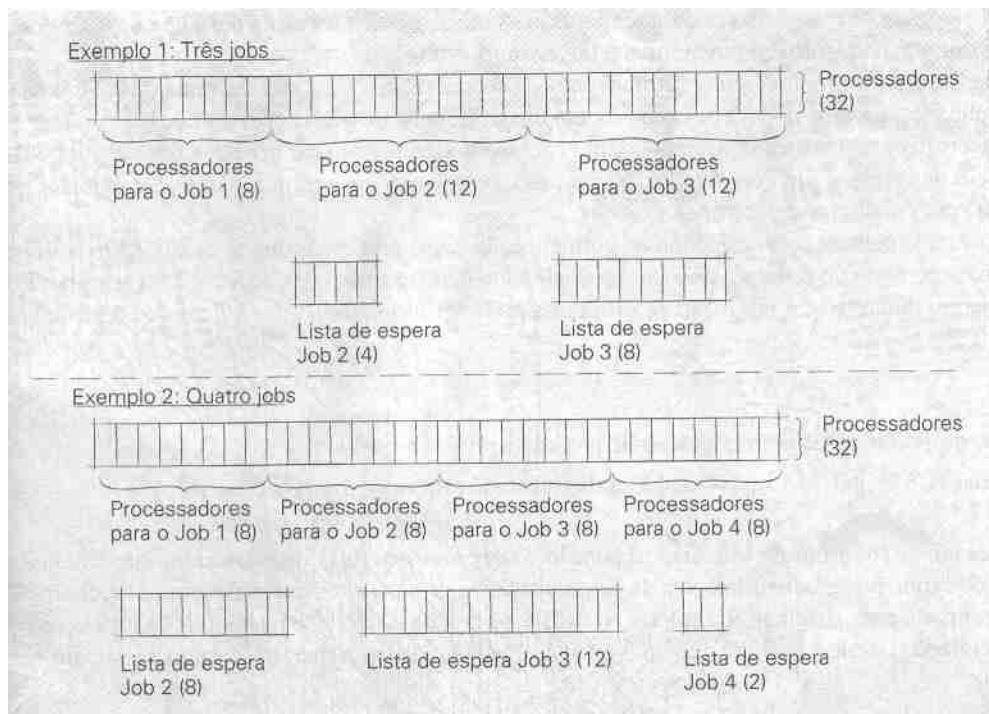


Figura 16.16: Partição dinâmica

## 16.7 MIGRAÇÃO DE PROCESSOS

Migração de processos implica transferir um processo entre dois processadores. Isso pode ocorrer, por exemplo, se um processador falhar ou estiver sobrecarregado.

A capacidade de executar processos em qualquer processador tem muitas vantagens. A mais óbvia é que eles podem passar para processadores subutilizados para reduzir tempos de resposta de processo e aumentar o desempenho e o rendimento. A migração de processos também promove a tolerância a falha. Por exemplo, considere um programa que deva executar cálculos intensivos, sem interrupção. Se a máquina que o estiver executando precisar ser desligada ou ficar instável, o progresso do programa poderá ser perdido. A migração de processos permite que o programa passe para outra máquina e continue o cálculo em um ambiente talvez mais estável.

Além disso, migração de processos promove compartilhamento de recursos. Em sistemas de grande escala alguns recursos podem não estar replicados em cada nodo. Por exemplo, em um sistema NORMA, processos podem executar em máquinas com suporte de dispositivo de hardware diferente. Um processo poderia requisitar acesso a um arranjo RAID que esteja disponível por meio de somente um computador. Nesse caso, o processo deve migrar para o computador que tenha acesso ao arranjo RAID para melhorar o desempenho.

Por fim, migração de processos melhora o desempenho da comunicação. Dois processos que se comunicam com freqüência devem executar no mesmo nodo ou próximo dele para reduzir a latência de comunicação. Pelo fato de enlaces de comunicação muitas vezes serem dinâmicos, a migração de processos pode ser usada para tomar dinâmico o posicionamento de processos.

### 16.7.1 FLUXO DE MIGRAÇÃO DE PROCESSOS

Embora as implementações de migração de processos variem entre arquiteturas, muitas delas seguem as mesmas etapas gerais (Figura 16.17). Primeiro, um nodo emite uma requisição de migração para um nodo remoto. Na maioria dos esquemas, o emissor inicia a migração porque seu nodo está sobrecarregado ou um processo específico precisa acessar um recurso localizado em um nodo remoto. Em alguns esquemas um nodo subutilizado pode requisitar processos de outros nodos. Se o emissor e o receptor concordarem em migrar um processo, o emissor suspenderá o processo migrante e criará uma fila de mensagens para reter todas as mensagens destinadas a ele. Então, extrairá o estado do processo, o que incluirá copiar o conteúdo da memória do processo (ou seja,

páginas marcadas como válidas na memória virtual do processo), conteúdo de registradores, estado de arquivos abertos e outras informações específicas de processo. O emissor transmitirá o estado extraído para um processo 'fictício' criado pelo receptor. Os dois nodos notificarão a todos os processos a nova localização do processo migrante. Por fim, o receptor despachará a nova instância do processo, o emissor transmitirá as mensagens que estão na fila do processo migrado e destruirá a instância local do processo.

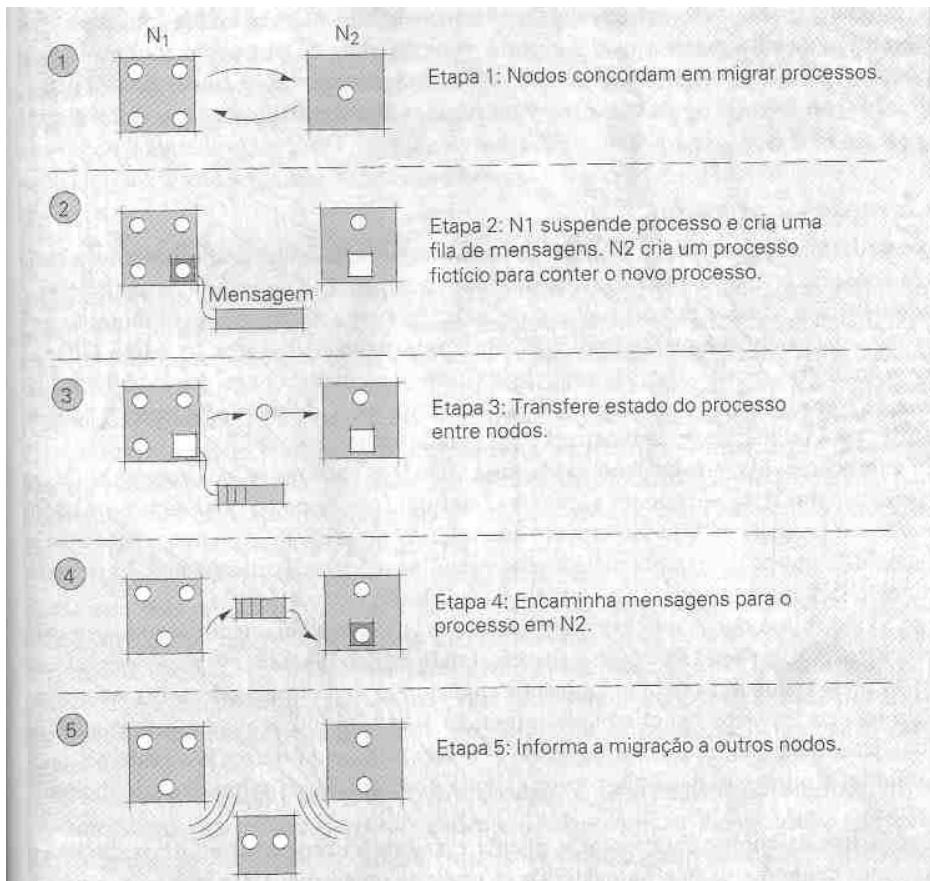


Figura 16.17: Migração de processos

### 16.7.2 CONCEITOS DE MIGRAÇÃO DE PROCESSOS

A transferência de memória é o elemento que mais consome tempo na migração. Para minimizar o custo de desempenho do processo de migração, a dependência residual - dependência do processo em relação ao seu nodo anterior - deve ser minimizada. Por exemplo, o nodo anterior do processo poderia conter parte do grupo de trabalho do processo ou poderia estar executando outros processos com os quais o processo migrado estivesse se comunicando. Se a migração resultar em muitas dependências residuais, um nodo receptor terá de se comunicar com o nodo emissor enquanto executa o processo migrado, o que aumentará a IPC, gerando mais tráfego na rede de interconexão, e degradará o desempenho devido a altas latências. E as dependências residuais também reduzirão a tolerância a falha porque a execução do processo migrado dependerá do funcionamento correto de ambos os nodos.

Muitas vezes, as estratégias que resultam no mais alto grau de dependência residual transferem páginas do emissor somente quando o processo do nodo receptor se refere a eles. Essas estratégias de migração preguiçosa (ou sob demanda) reduzem o tempo de migração inicial do processo - o tempo durante o qual o processo migrante fica suspenso. Na migração preguiçosa, as informações sobre o estado do processo (e outras requeridas) são transferidas para o receptor, mas o nodo original retém as páginas do processo. Enquanto executa no nodo remoto, o processo tem de iniciar uma transferência de memória para cada acesso a uma página que permanecer no nodo emissor. Embora essa técnica resulte em uma rápida migração inicial do processo, o acesso à memória pode reduzir seriamente o desempenho de uma aplicação. A migração preguiçosa é muito útil quando o processo não requer acesso freqüente ao espaço de endereçamento remoto.

Para que a migração seja bem-sucedida, os processos devem exibir diversas características. Um processo migrado deve exibir transparência - não deve ser afetado adversamente pela migração (exceto, talvez, um leve atraso no tempo de resposta). Em outras palavras, o processo não deve perder nenhuma mensagem interprocessos nem os arquivos abertos que manuseia. Um sistema também deve ser escalável - se as dependências residuais crescerem com cada migração, o sistema poderá rapidamente ficar saturado pelo tráfego da rede à medida que o processo requisitar páginas remotas. Por fim, avanços na tecnologia de comunicação entre várias arquiteturas criaram a necessidade de migrações heterogêneas - os processos devem poder migrar entre duas arquiteturas de processador diferentes em sistemas distribuídos. Isso implica que o estado do processo deve ser armazenado em um formato independente de plataforma.

### **16.7.3 ESTRATÉGIAS DE MIGRAÇÃO DE PROCESSOS**

Estratégias de migração de processos devem equilibrar a penalidade incidente sobre o desempenho na transferência de grandes quantidades de dados de processo com o benefício da minimização da dependência residual de um processo. Em alguns sistemas, projetistas admitem que a maior parte do espaço de endereçamento de um processo migrado será acessada no seu novo nodo. Esses sistemas freqüentemente implementam a migração ávida, que transfere todas as páginas do processo durante a migração inicial, o que o habilita a executar tão eficientemente em seu novo nodo quanto executava no nodo anterior. Todavia, se o processo não acessar grande parte do seu espaço de endereçamento, a latência inicial e a largura de banda requeridas pela migração ávida incorrerão em grande sobrecarga.

Para atenuar o custo inicial da migração ávida, a migração ávida suja transfere somente as páginas sujas de um processo. Essa estratégia admite que haja um armazenamento secundário comum (um disco ao qual ambos os nodos têm acesso). Todas as páginas limpas são trazidas do armazenamento secundário comum à medida que o processo se refere a elas no nodo remoto, o que reduz o tempo de transferência inicial e elimina a dependência residual. Entretanto, cada acesso a uma página não residente leva mais tempo do que levaria usando a migração ávida.

Uma desvantagem da migração ávida suja é que o processo migrado deve usar armazenamento secundário para recuperar as páginas limpas do processo. A migração de cópia-sob-referência é semelhante à migração preguiçosa suja, exceto que o processo migrado pode requisitar páginas limpas ou do seu nodo anterior ou do armazenamento secundário comum. Essa estratégia tem os mesmos benefícios da migração ávida suja, mas dá ao gerenciador de memória mais controle sobre a localização da qual requisitar páginas - o acesso à memória remota pode ser mais rápido do que acesso ao disco. Contudo, a migração de cópia-sob-referência pode adicionar sobrecarga de memória ao emissor.

A implementação de cópia preguiçosa da cópia-sob-referência transfere apenas informações mínimas durante o tempo da migração inicial; muitas vezes nenhuma página é transferida, o que cria uma grande dependência residual, força o nodo anterior a manter páginas na memória para o processo migrado e aumenta a latência de acesso à memória em relação às estratégias que migram páginas sujas. Contudo, a estratégia de cópia preguiçosa elimina grande parte da latência inicial da migração. Essa latência pode ser inaceitável para processos de tempo real e é inadequada para a maioria dos processos.

Todas as estratégias discutidas até aqui criam dependência residual ou incorrem em grande latência inicial de migração. Um método que elimina grande parte dessa latência é a estratégia de descarga. Nessa estratégia o emissor escreve todas as páginas da memória para um armazenamento secundário quando a migração inicia; o processo migrante deve ser suspenso enquanto os dados estão sendo gravados no armazenamento secundário e, então, o processo acessa as páginas do armazenamento secundário conforme o necessário. Portanto, não ocorre nenhuma migração real de página que reduza a velocidade da migração inicial e o processo não tem nenhuma dependência residual do nodo anterior. Todavia, a estratégia de descarga reduz o desempenho do processo no

seu novo nodo porque ele não tem páginas na memória principal, o que resulta em falta de páginas.

Uma outra estratégia para eliminar grande parte da latência da imigração inicial e a dependência residual é o método da cópia prévia, no qual o nodo emissor começa a transferir páginas sujas antes da suspensão do processo original. Qualquer página transferida que o processo modificar antes da migração é marcada para retransmissão. Para garantir que um processo eventualmente migre, o sistema define um patamar inferior para o número de páginas sujas que devem permanecer antes que o processo migre. Quando esse patamar é alcançado o processo é suspenso e migrado para um outro processador. Com essa técnica, o processo não precisa ficar suspenso por longo tempo (em comparação com outras técnicas como cópia preguiçosa e cópia-sob-referência) e o acesso à memória no novo nodo é fácil, pois a maioria dos dados já foi copiada. E, também, a dependência residual é mínima. O conjunto de trabalho do processo é duplicado durante um curto período de tempo (existe em ambos os nodos envolvidos na migração), mas essa é uma desvantagem mínima.

## 16.8 BALANCEAMENTO DE CARGA

Uma medida da eficiência de sistemas multiprocessadores é a utilização geral dos processadores. Em grande parte dos casos, se a utilização dos processadores for alta, o sistema estará desempenhando com mais eficiência. A maioria dos sistemas multiprocessadores (especialmente sistemas NUMA e NORMA) tenta maximizar a afinidade de processadores. Isso aumenta a eficiência porque os processos não precisam acessar recursos remotos com tanta freqüência, mas podem reduzir a utilização se todos os processos designados a um determinado processador forem concluídos. Esse processador ficará ocioso enquanto processos são despachados para outros processadores para explorar a afinidade. **Balanceamento de carga** é uma técnica pela qual o sistema tenta distribuir cargas de processamento eqüitativamente entre processadores, o que aumenta a utilização de processadores e reduz filas de execução de processadores sobrecarregados, diminuindo os tempos médios de resposta dos processos.

Um algoritmo de balanceamento de carga pode designar um número fixo de processadores a um job quando esse for escalonado pela primeira vez, o que é denominado **balanceamento estático de carga**. Esse método resulta em baixa sobrecarga de tempo de execução porque os processadores gastam pouco tempo determinando os processadores nos quais um job deve executar. Contudo, o balanceamento estático de carga não leva em conta as populações variáveis de processos dentro de um job. Por exemplo, um job pode incluir muitos processos inicialmente, mas manter apenas alguns durante o restante da sua execução, o que pode levar a filas de execução desbalanceadas, que podem resultar na ociosidade do processador.

O **balanceamento dinâmico de carga** tenta abordar essa questão ajustando o número de processadores designados a um job durante sua vida. Estudos demonstraram que o balanceamento dinâmico de carga resulta em melhor desempenho do que o balanceamento estático de carga quando o tempo de chaveamento de contexto é baixo e a carga do sistema é alta.

### 16.8.1 BALANCEAMENTO ESTÁTICO DE CARGA

O balanceamento estático de carga é útil em ambientes nos quais os jobs repetem certos testes ou instruções e, portanto, exibem padrões previsíveis (por exemplo, computação científica). Esses padrões podem ser representados por grafos usados para modelar o escalonamento. Considere os processos de um sistema como vértices de um grafo, e as comunicações entre os processos como arestas. Por exemplo, se houver uma aplicação na qual um processo armazene continuamente dados similares e, então, passe aqueles dados para outro processo, essa operação poderá ser modelada como dois nodos ligados por uma aresta. Pelo fato de essa relação ser consistente durante toda a vida da aplicação, não haverá necessidade de ajustar o grafo.

Devido ao compartilhamento da memória cache e da memória física, a comunicação entre processos no mesmo processador é muito mais rápida do que a comunicação entre processos em processadores diferentes. Portanto, os algoritmos de balanceamento

estático de carga tentam dividir o grafo em subgrafos de tamanhos semelhantes (cada processador tem um número similar de processos) minimizando, ao mesmo tempo, as arestas entre os subgrafos para reduzir a comunicação entre processadores. Entretanto, essa técnica pode incorrer em sobrecarga significativa quando houver um grande número de jobs. Considere o grafo da Figura 16.18. As duas linhas tracejadas representam os possíveis cortes para dividir processos, até certo ponto, equitativamente. O Corte #1 resulta em quatro canais de comunicação interprocessos, ao passo que o Corte #2 resulta em apenas dois, representando, assim, um melhor agrupamento de processos.

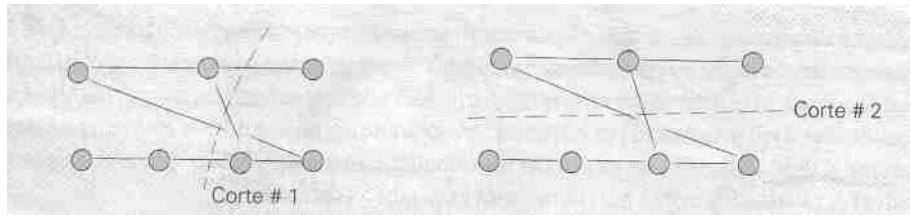


Figura 16.18: Balanceamento estático de carga usando grafos

O balanceamento estático de carga pode ser inadequado quando padrões de comunicação mudam dinamicamente e quando processos são concluídos sem haver nenhum processo para ocupar os seus lugares. O primeiro caso pode ter desempenho menos eficientemente em virtude da alta latência de comunicação. No segundo caso, a utilização do processador poderia diminuir mesmo quando houvesse processos aguardando para obter um processador. Nesses casos o balanceamento dinâmico de carga pode melhorar o desempenho.

### 16.8.2 BALANCEAMENTO DINÂMICO DE CARGA

Algoritmos de balanceamento dinâmico de carga migram processos depois de eles terem sido criados em resposta à carga do sistema. O sistema operacional mantém informações estatísticas sobre a carga do processador, como o número de processos ativos e bloqueados em um processador, a utilização média do processador, o tempo de retorno e a latência. Se muitos dos processos de um processador estiverem bloqueados ou tiverem tempos de retorno altos, o processador muito provavelmente estará sobrecarregado. Se um processador não tiver uma taxa de utilização alta, provavelmente estará subcarregado.

Diversas políticas podem ser utilizadas para determinar quando migrar processos no balanceamento dinâmico de carga. A **política iniciada pelo emissor** entra em ação quando o sistema determina que um processador contém uma carga pesada. Somente então o sistema pesquisará processadores subutilizados e migrará alguns dos jobs dos processadores sobrecarregados para eles. Essa política é melhor para sistemas com cargas leves, pois a migração de processos é dispendiosa e, nesse caso, a política raramente será ativada.

Ao contrário, a **política iniciada pelo receptor** é melhor para sistemas sobrecarregados. Nesse ambiente, o sistema inicia migração de processos quando a utilização de um processador for baixa.

A maioria dos sistemas experimenta cargas leves e pesadas ao longo do tempo. Para esses sistemas a **política simétrica**, que combina os dois métodos anteriores, proporciona o máximo de versatilidade para adaptação às condições ambientais. Por fim, a **política aleatória**, na qual o sistema escolhe arbitrariamente um processador para receber um processo migrado, tem mostrado resultados decentes devido à sua implementação simples e (na média) distribuição uniforme de processos. A motivação por trás da política aleatória é que o destino de um processo migrante terá, provavelmente, uma carga menor do que sua origem, considerando que o processador original esteja seriamente sobrecarregado.

As subseções seguintes descrevem algoritmos comuns que determinam como processos são migrados. Para todos os propósitos desta discussão, considere que o sistema multiprocessador pode ser representado por um grafo no qual cada processador e sua memória são um vértice e cada ligação, uma aresta.

## Algoritmo de Licitação

O algoritmo de licitação é uma política simples de migração iniciada pelo emissor. Processadores com cargas menores 'licitam' ('dão lances a') processos de processadores sobrecarregados, algo muito parecido com o que se faz em leilões. O valor de um lance é baseado na carga corrente do processador licitante e na distância entre os processadores sobrecarregado e subcarregado no grafo. Para reduzir o custo do processo de migração, os caminhos mais diretos de comunicação com o processador sobrecarregado recebem os lances de valores mais altos. O processador sobrecarregado aceita os lances dos processadores que estão dentro de uma certa distância no grafo. Se o processador sobrecarregado receber demasiados lances, ele reduzirá a distância; se receber muito poucos, aumentará a distância e verificará novamente. O processo é enviado para o processador que der o lance mais alto.

## Algoritmos de Recrutamento

O algoritmo de recrutamento é uma política iniciada pelo receptor que classifica a carga de cada processador como baixa, normal ou alta. Cada processador mantém uma tabela que descreve as cargas dos outros processadores usando essas classificações. Muitas vezes, em sistemas de grande escala ou sistemas distribuídos, processadores mantêm somente informações sobre seus vizinhos. Toda vez que a carga de um processador muda de classificação, o processador transmite suas informações atualizadas aos processadores que estão na sua tabela de carga. Quando um processador recebe uma dessas mensagens, ele anexa suas próprias informações e transmite a mensagem aos seus vizinhos. Desse modo, informações sobre níveis de carga eventualmente chegam a todos os nodos da rede. Processadores subutilizados usam essas informações para requisitar processos de processadores sobrecarregados.

## Questões de Comunicação

Estratégias de comunicação ineficientes ou incorretas podem saturar um sistema. Por exemplo, algumas implementações de migração empregam transmissão no âmbito do sistema. O dilúvio de mensagens transmitidas pode saturar os canais de comunicação. Devido a atrasos de comunicação, muitos processadores sobrecarregados poderiam receber a requisição de um processo ao mesmo tempo, e todos poderiam enviar seus processos a um só processador subcarregado.

Diversas estratégias foram elaboradas para evitar esses problemas. Por exemplo, o algoritmo poderia restringir a comunicação dos processadores somente com seus vizinhos imediatos, o que reduziria o número de mensagens transmitidas, mas aumentaria o tempo requerido para a informação chegar a todos os nodos do sistema. Como alternativa, os processadores poderiam selecionar periodicamente um processador aleatório com o qual trocar informações. Nesse caso, os processos seriam migrados de um processador cuja carga seria mais alta para um outro cuja carga seria mais baixa. Em casos em que um processador esteja seriamente sobrecarregado e os restantes subcarregados, haverá uma rápida difusão do processo.

A Figura 16.19 ilustra a difusão de processo em um sistema no qual os nodos se comunicam apenas com seus vizinhos. O processador sobrecarregado, representado pelo vértice do meio, tem 17 processos, enquanto todos os outros têm apenas um processo. Após uma iteração, o processador sobrecarregado comunica-se com seus vizinhos e envia 3 processos para cada um. Agora esses processadores têm 4 processos, três a mais do que alguns de seus vizinhos. Na segunda iteração, os processadores que têm 4 processos enviam alguns a seus vizinhos. Por fim, na terceira iteração, o processador sobre-carregado mais uma vez envia alguns processos a seus vizinhos, de modo que agora o processador cuja carga é a mais pesada tem somente 3 processos. Esse exemplo ilustra que, mesmo quando as comunicações são mantidas entre processadores vizinhos, o balanceamento de carga pode distribuir responsabilidades de processamento efetivamente por todo o sistema.

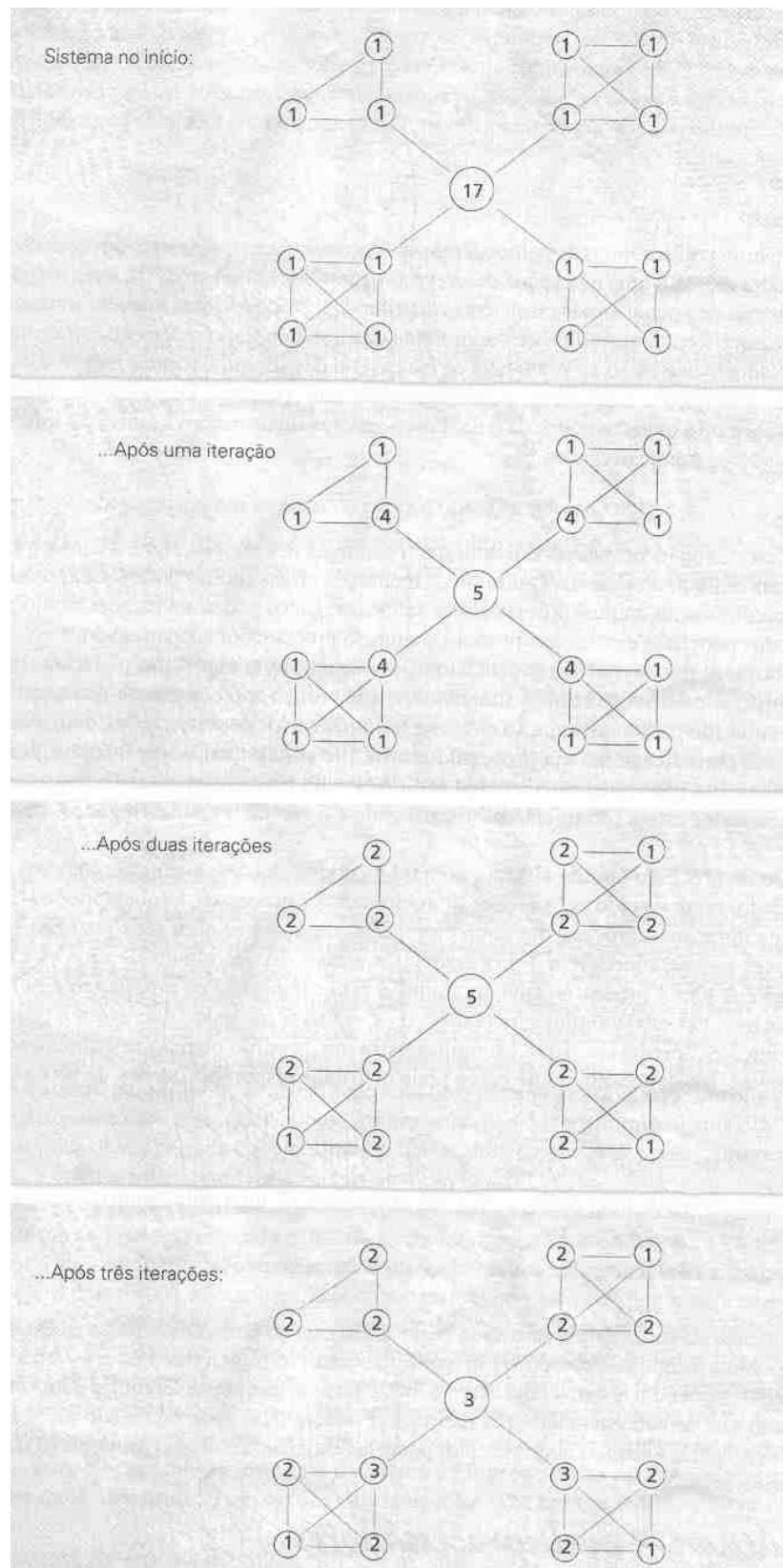


Figura 16.19: Difusão de carga de processador

### 16.9 EXCLUSÃO MÚTUA EM MULTIPROCESSADORES

Muitos dos mecanismos de exclusão mútua descritos anteriormente não são adequados a sistemas multiprocessadores. Por exemplo, a desabilitação de interrupções, que impede que outros processos executem desabilitando preempções em sistemas monoprocessadores, não garante a exclusão mútua em multiprocessadores porque diversos processos podem executar simultaneamente em processadores diferentes. Instruções

como teste-e-atualize (test-and-set) podem ser ineficientes quando os dados a que essas instruções se referem estão localizados em memória remota. A instrução teste-e-atualize tem uma outra vulnerabilidade, a saber, cada uma das operações pendentes requer um acesso separado à memória. Os acessos são realizados seqüencialmente porque, em geral, pode ocorrer apenas um acesso a uma localização de memória por vez. Essas colisões podem saturar rapidamente vários tipos de redes de interconexão, causando sérios problemas de desempenho. Por causa dos custos de comunicações não triviais e do fato de que vários processos executam ao mesmo tempo, projetistas desenvolveram diversas técnicas de exclusão mútua para multiprocessadores.

### 16.9.1 TRAVAS GIRATÓRIAS

Sistemas operacionais multiprocessadores como o Windows XP e o Linux com frequência usam travas giratórias (*spin locks*) para exclusão mútua de multiprocessador. Uma trava giratória é denominada 'trava' porque o processo que tiver uma trava giratória reclamará acesso exclusivo ao recurso que a trava protege (por exemplo, uma estrutura de dados compartilhada na memória ou uma seção crítica de um código). Na verdade, são os outros processos que ficam 'trancados' para fora do recurso. Outros processos que querem usar o recurso 'giram' (ou seja, esperam ociosamente), testando continuamente uma condição para determinar se o recurso está disponível. Em sistemas monoprocessadores as travas giratórias são um desperdício porque consomem ciclos de processador que poderiam ser usados por outros processos, incluindo o processo que detém a trava, para realizar trabalho útil. Em sistemas multiprocessadores, o processo que detém a trava pode liberá-la enquanto outro processo estiver em espera ociosa. Nesse caso, é mais eficiente para um processo esperar ociosamente se o tempo requerido pelo sistema para executar um chaveamento de contexto para escalaronar um outro processo for mais longo do que o tempo médio de espera ociosa. E, também, se um processador não contiver nenhum outro processo em sua fila de execução, fará sentido manter o processo girando para minimizar o tempo de resposta quando a trava giratória for liberada.

Quando um processo detém uma trava giratória durante um longo tempo (em relação ao tempo de chaveamento de contexto), bloquear é mais eficiente. O **bloqueio retardado** é uma técnica na qual o processo gira por um curto período de tempo; se ele não adquirir a trava durante aquele período, será bloqueado. Uma **trava de processo anunciada** (*Advisable Process Lock - APL*) apresenta uma solução alternativa. Quando o processo adquire uma APL, ele especifica a quantidade de tempo durante o qual deterá a trava. Com base no tempo especificado, outros processos que estão à espera para adquirir a APL podem determinar se é mais eficiente esperar ociosamente pela trava ou bloquear.

**Travas adaptativas** (também denominadas travas configuráveis) adicionam flexibilidade a uma implementação de exclusão mútua. Em certos momentos, como quando há poucos processos ativos, as travas giratórias são preferíveis, pois um processo pode adquirir um recurso logo após tomar-se disponível. Em momentos em que a carga do sistema for alta, girar perde valiosos ciclos de processador, e o bloqueio é preferível. Travas adaptativas permitem que um processo mude dinamicamente o tipo de trava que está em uso. Essas travas podem ser ajustadas para bloqueio, giro ou bloqueio retardado e podem incorporar características de APL para personalizar a trava conforme a carga do sistema e as necessidades da aplicação.

Quando vários processos esperam simultaneamente pela liberação de uma trava giratória, pode ocorrer adiamento indefinido. Sistemas operacionais podem evitar adiamento indefinido concedendo a trava giratória a processos na ordem 'primeiro a chegar, primeiro a ser atendido', ou envelhecendo processos que estão à espera pela trava giratória.

### 16.9.2 TRAVAS DORMIR/ACORDAR

Uma trava dormir/acordar fornece sincronização similar à de uma trava giratória, mas reduz o desperdício de ciclos de processador e o tráfego no barramento. Considere os processos P1 e P2 ambos requisitando a utilização de um recurso protegido por uma trava dormir/acordar. P1 requisita o recurso em primeiro lugar e obtém a trava. Quando P2 requisita o recurso possuído por P1 e não o recebe, P2 responde adormecendo (ou

seja, bloqueia). Assim que P1 libera a trava, ele acorda o processo de prioridade mais alta que estiver esperando pelo recurso (nesse caso, P2). Ao contrário das travas giratórias (que dão a trava ao próximo processo que estiver à espera), as travas dormir/acordar podem usar o escalonador do processador para impor prioridades de processo. Elas podem fazer com que processos que estão à espera sejam adiados indefinidamente, dependendo da política de escalonamento do sistema.

Note que, diferentemente de uma implementação de monoprocessador, somente é acordado o processo que tiver a prioridade mais alta. Quando todos os threads são acordados, pode ocorrer uma condição de disputa porque dois ou mais threads podem acessar um recurso associado a uma trava em uma ordem não determinística. Condições de disputa devem ser evitadas porque podem causar erros imperceptíveis em aplicações e são difíceis de depurar. Em monoprocessadores, mesmo que todos os processos sejam alertados, não pode haver uma condição de disputa, pois apenas um processo (o que tiver a prioridade mais alta na maioria dos algoritmos de escalonamento) obterá o controle do processador e adquirirá a trava. Em um ambiente multiprocessador, uma difusão deve acordar muitos processos que estão competindo pela trava, criando uma **condição de corrida**. O resultado disso também seria um processo obtendo a trava e muitos outros testando, bloqueando novamente e, consequentemente, voltando a dormir e, assim, desperdiçando tempo de processador devido ao chaveamento de contexto. Esse fenômeno é conhecido como **estouro da boiada**.

### 16.9.3 TRAVAS DE LEITURA/ESCRITA

Impor acesso mutuamente exclusivo à memória compartilhada em um sistema multiprocessador pode degradar o desempenho. Somente escritores necessitam de acesso exclusivo ao recurso, ao passo que, em geral, vários leitores podem acessar a mesma localização da memória ao mesmo tempo. Portanto, muitos sistemas protegem a memória compartilhada com uma **trava de leitura/escrita** mais versátil, em vez de uma trava genérica de exclusão mútua. Uma trava de leitura/escrita proporciona exclusão mútua similar à apresentada no problema dos leitores/escritores. Uma trava de leitura/escrita permite que vários processos leitores (ou seja, processos que não alterarão os dados compartilhados) entrem em suas seções críticas. Diferentemente do monoprocessamento, todavia, as travas de leitura/escrita exigem que um escritor (um processo que alterará dados compartilhados) espere até não haver mais nenhum processo leitor ou escritor em suas seções críticas antes de entrar na sua própria seção crítica.

Para implementar essa abordagem de travamento de maneira eficiente, o mecanismo de exclusão mútua deve usar memória compartilhada. Todavia, em ambientes nos quais a memória não é compartilhada, como em sistemas NORMA, deve ser utilizada a troca de mensagem.

## 16.10 EXERCÍCIOS

- 269) Por que os multiprocessadores são tão úteis?
- 270) Quais as diferenças entre as responsabilidades de um Sistema Operacional multiprocessador e as de um sistema de um só processador?
- 271) Paralelismo no nível de thread (TLP) refere-se à execução de múltiplos threads independentes em paralelo. Qual quitetura de multiprocessador explora TLP?
- 272) (V /F) Somente arquiteturas SIMD exploram paralelismo.
- 273) Uma rede em malha 2-D com 8 conexões inclui enlaces para nodos diagonais, bem como os enlaces da Figura 16.3. Compare redes em malha 2-D de 8 conexões e de 4 conexões.
- 274) Compare uma rede multiestágio com uma matriz de comutação de barras cruzadas. Quais os benefícios e desvantagens de cada esquema de interconexão?

- 275) Por que muitos multiprocessadores pequenos são construídos como sistemas fortemente acoplados? Por que muitos sistemas de grande escala são fracamente acoplados?
- 276) Alguns sistemas consistem em vários grupos de componentes conectados entre si de um modo fortemente acoplado. Discuta algumas motivações para esse esquema.
- 277) (V/F) Multiprocessadores mestre/escravo escalam bem para sistemas de grande escala.
- 278) Para que tipo de ambientes os multiprocessadores mestre/escravo são mais indicados?
- 279) Por que a organização de núcleos separados é mais tolerante a falha do que a organização mestre/escravo?
- 280) Para que tipo de ambiente os multiprocessadores de núcleos separados seriam úteis?
- 281) Por que dobrar o número de processadores de uma organização simétrica não dobra a capacidade total de processamento?
- 282) Cite alguns benefícios da organização simétrica em relação às organizações mestre/escravo e de núcleos separados.
- 283) Por que redes em malha e hipercubos são esquemas de interconexão inadequados para sistemas UMA?
- 284) Como um sistema UMA é “simétrico”?
- 285) Cite algumas vantagens dos multiprocessadores NUMA em relação aos UMA. Cite algumas desvantagens.
- 286) Quais as questões que o projeto NUMA levanta para programadores e para projetistas de sistemas operacionais?
- 287) Quais os problemas inerentes aos multiprocessadores NUMA abordados pelo projeto COMA?
- 288) (V /F) O projeto COMA sempre aumenta o desempenho em relação a um projeto NUMA.
- 289) Por que multiprocessadores NORMA são ideais para agrupamentos de estações de trabalho (clusters)?
- 290) Em quais ambientes multiprocessadores NORMA não são úteis?
- 291) Por que a coerência de memória é importante?
- 292) Quais as vantagens de permitir que existam várias cópias da mesma página em um sistema?
- 293) Quais as vantagens e desvantagens da escuta do barramento?
- 294) Por que é difícil para o sistema estabelecer um nodo nativo para memória que é referida por diversos nodos diferentes ao longo do tempo? Como o protocolo CC-

NUMA baseado no conceito de nodo nativo pode ser modificado para suportar esse comportamento?

- 295) Como uma estratégia de migração/replicação poderia degradar o desempenho?
- 296) Para que tipos de páginas a replicação é apropriada? Quando a migração é apropriada?
- 297) Cite um benefício da consistência relaxada. E uma desvantagem.
- 298) Em muitas implementações de consistência relaxada os dados podem não estar coerentes durante vários segundos. Isso é um problema para todos os ambientes? Dê um exemplo no qual poderia ser um problema.
- 299) Que tipos de processos se beneficiam do escalonamento por tempo compartilhado? E do escalonamento por partição de espaço?
- 300) Quando filas de execução por nodo são mais apropriadas do que filas globais de execução?
- 301) Um sistema UMA ou NUMA é mais adequado para escalonamento de multiprocessadores cego ao job?
- 302) Qual estratégia de escalonamento cego ao job discutida nesta seção é mais apropriada para sistemas de processamento em lote e por quê?
- 303) Quais as similaridades entre RRJob e coescalonamento não dividido? E as diferenças?
- 304) Descreva algumas das permutas entre implementar uma política global de escalonamento que maximize a afinidade de processador, tal como a partição dinâmica, e filas de execução por processador.
- 305) Cite alguns benefícios proporcionados pela migração de processos.
- 306) Em que tipo de sistemas (UMA, NUMA ou NORMA) a migração de processos é mais apropriada?
- 307) Cite alguns itens que compõem o estado de um processo e que devem migrar com um processo.
- 308) Que sobrecarga é incorrida na migração de processos?
- 309) Por que a dependência residual é indesejável? Por que uma certa dependência residual poderia ser benéfica?
- 310) Como uma estratégia de migração que resulta em dependência residual significativa poderia não ser escalável?
- 311) Quais estratégias de migração devem ser usadas em processos de tempo real?
- 312) Embora o tempo de migração inicial seja mínimo e haja pouca dependência residual na estratégia de cópia prévia, você consegue imaginar alguns custos 'ocultos' incorridos nessa estratégia?
- 313) Cite alguns dos benefícios da implantação do balanceamento de carga.
- 314) Quais algoritmos de escalonamento se beneficiam do balanceamento de carga?

- 315) Quando o balanceamento estático de carga é útil? Quando não é?
- 316) Na Figura 16.18 há uma diferença substancial entre o Corte #1 e o Corte #2. Considerando que é difícil descobrir o corte mais efetivo quando há um grande número de jobs, o que isso significa em relação às limitações do balanceamento estático de carga?
- 317) Em que tipo de ambiente deve ser usada uma política dirigida ao emissor? E uma política dirigida ao receptor? Justifique suas respostas.
- 318) Como uma alta latência de comunicação atrapalha a efetividade do balanceamento de carga?
- 319) De que modo girar pode ser útil em um multiprocessador? Sob quais condições girar não é desejável em um multiprocessador? Por que o giro não é útil em um sistema monoprocessador?
- 320) Por que uma APL pode não ser útil em todas as situações?
- 321) Cite uma vantagem e uma desvantagem da trava dormir/acordar em comparação com a trava giratória.
- 322) De que modo a implementação de uma trava dormir/acordar é diferente em ambientes multiprocessadores e ambientes monoprocessadores?
- 323) Em que situações as travas de leitura/escrita são mais eficientes do que as travas giratórias?
- 324) Uma implementação ingênuia de uma trava de leitura/escrita permite que qualquer leitor entre em sua seção crítica quando nenhum escritor estiver em sua própria seção crítica. Como isso pode levar a adiamento indefinido para os escritores? Qual seria uma implementação mais justa?

-X-

## Bibliografia

"Só sei que nada sei" (Sócrates)

Esta apostila foi totalmente baseada nos livros:

DEITEL H. M., DEITEL P. J. e CHOFFNES D.R. *Sistemas Operacionais*. São Paulo, 2005. Prentice Hall

TANEMBAUM, Andrew S. *Sistemas Operacionais Modernos*. São Paulo, 2003. Prentice Hall

MAIA, Luiz Paulo e MACHADO, Francis Berenger. *Arquitetura de Sistemas Operacionais*. Rio de Janeiro, 2002. LTC.