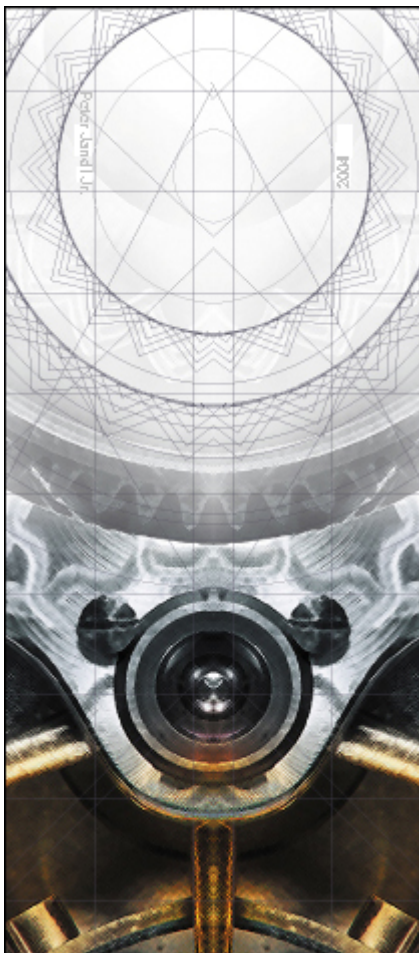


Notas sobre
Sistemas Operacionais



Peter Jandl Jr.

Jandl, Peter, Jr. Notas sobre Sistemas Operacionais/Peter Jandl Jr. Apostila 1. Sistemas operacionais: Computadores : Processamento de dados : 005.43 2004

Histórico

1.1 Fev2004 Revisão Geral. Threads. Escalonamento por prioridades. Escalonamento com múltiplas filas.

1.0 Ago1999 Versão Inicial.

*” O homem pode se tornar culto
pela cultura dos outros,
mas só pode se tornar sábio
pelas próprias experiências.”*
(Provérbio Árabe)

Sumário

Prefácio	1
1 Introdução	3
1.1 Definindo os sistemas operacionais	3
1.2 Objetivos de um sistema operacional	6
1.3 Breve histórico	7
1.3.1 O início	7
1.3.2 Década de 1940	7
1.3.3 Década de 1950	9
1.3.4 Década de 1960	11
1.3.5 Década de 1970 e 1980	13
1.3.6 Década de 1990	14
1.3.7 Década de 2000	15
1.4 Tipos de sistemas operacionais	15
1.5 Recursos e ambiente operacional	17
2 Processos	21
2.1 O que é um processo computacional	21
2.1.1 Subdivisão dos processos	22
2.2 Ocorrência de processos	23
2.2.1 Processos seqüenciais	24
2.2.2 Processos Paralelos	24
2.3 Estado dos processos	25
2.4 PCB e tabelas de processos	28
2.4.1 PCB	29
2.4.2 Tabelas de processos	30
2.5 Operações sobre processos	30
2.6 Funções do núcleo de sistema operacional	31
2.7 Competição por recursos	32
2.7.1 Regiões críticas	32
2.7.2 Código reentrante	33
2.8 Protocolos de acesso	33
2.8.1 Solução com instruções TST ou TSL	36

2.8.2	Situações de corrida	37
2.8.3	Requisitos de um protocolo de acesso	40
2.9	A solução de Dekker	40
2.10	A solução de Peterson	43
2.11	Deadlocks	45
2.11.1	Diagramas de processos e recursos	46
2.11.2	Condições para ocorrência de <i>deadlocks</i>	49
2.11.3	Recuperação de <i>deadlocks</i>	50
2.11.4	Prevenção de <i>deadlocks</i>	51
2.11.5	Estados seguros e inseguros	51
2.11.6	Algoritmo do banqueiro	53
2.12	Comunicação de processos	53
2.12.1	<i>Buffers</i> e operações de <i>sleep</i> e <i>wakeup</i>	54
2.12.2	Semáforos	57
2.12.3	Memória compartilhada	58
2.12.4	Outros mecanismos de IPC	60
2.13	Threads	62
2.13.1	Modelos de <i>multithreading</i>	64
2.13.2	Benefícios do uso	64
3	Escalonamento de Processos	67
3.1	Objetivos do escalonamento	68
3.2	Níveis de escalonamento	69
3.3	Escalonamento preemptivo e não preemptivo	69
3.4	Qualidade do escalonamento	71
3.5	Algoritmos de escalonamento	72
3.5.1	Escalonamento FIFO (<i>First In First Out</i>)	73
3.5.2	Escalonamento HPF (<i>Highest Priority First</i>)	74
3.5.3	Escalonamento SJF (<i>Shortest Job First</i>)	76
3.5.4	Escalonamento HRN (<i>Highest Response-Ratio Next</i>)	77
3.5.5	Escalonamento SRT (<i>Shortest Remaining Time</i>)	79
3.5.6	Escalonamento RR (<i>Round-Robin</i>)	80
3.5.7	Escalonamento MQ (<i>Multilevel Queues</i>)	83
3.5.8	Escalonamento MFQ (<i>Multilevel Feedback Queues</i>)	84
3.6	Comparação dos algoritmos de escalonamento	87
4	Gerenciamento de Memória	89
4.1	Primeiras considerações	89
4.2	Multiprogramação	92
4.3	Organização da memória	95
4.4	Definição de gerenciamento de memória	97
4.5	Criação de programas	99
4.5.1	Espaços lógicos e físicos	101
4.5.2	Compiladores (<i>compilers</i>)	103

4.5.3	Ligadores (<i>linkers</i>)	107
4.5.4	Carregadores (<i>loaders</i>)	110
4.5.5	Relocadores (<i>swappers</i>)	111
4.6	Memória virtual	113
4.7	Modelos de gerenciamento de memória	118
4.7.1	Monoprogramado com armazenamento real	118
4.7.2	Particionamento fixo	120
4.7.3	Particionamento variável	122
4.7.4	Paginação	123
4.7.5	Segmentação	128
4.7.6	Paginação <i>versus</i> Segmentação	131
4.7.7	Paginação e segmentação combinadas	132
4.7.8	Tabelas de páginas	134
4.7.9	Algoritmos de troca de páginas	139
5	Gerenciamento de I/O	143
5.1	Módulos de I/O	143
5.2	Operação de Módulos de I/O	145
5.2.1	I/O Programado	145
5.2.2	I/O com interrupções	148
5.2.3	I/O com Acesso Direto à Memória (DMA)	150
5.3	Tipos de dispositivos de E/S	152
5.3.1	Conexão de Dados dos I/O	152
5.3.2	Tipos de Transferência de I/O	153
5.3.3	Conexões ponto a ponto e multiponto com I/Os	154
5.4	Dispositivos periféricos típicos	155
5.4.1	Unidades de disco	156
5.4.2	Escalonamento de disco	160
5.4.3	Unidades de fita	165
5.4.4	Terminais	167
5.5	Sistemas de arquivos	171
5.5.1	Arquivos	173
5.5.2	Diretórios	178
5.5.3	Serviços do sistema operacional	181
5.5.4	Implementação Lógica	184
5.5.5	Implementação Física	186
5.5.6	Fragmentação	192
	Bibliografia	195

Prefácio

Este texto representa a condensação do material apresentado durante as aulas da disciplina Sistemas Operacionais, ministradas ao longo de meu trabalho no magistério nos cursos de graduação em Análise de Sistemas e Engenharia da Computação.

Como o próprio título indica, são notas de aulas, organizadas num roteiro bastante tradicional, acompanhadas de diagramas, desenhos, comentários e exemplos que tem como objetivo maior facilitar o entendimento do tema, tão importante dentro da Ciência da Computação assim como em outros cursos correlatos, tais como Engenharia da Computação, Análise de Sistemas e Sistemas de Informação. Desta forma, não se pretendeu colocar como um estudo completo e detalhado dos sistemas operacionais, tão pouco substituir a vasta bibliografia existente, mas apenas oferecer um texto de referência, onde sempre que possível são citadas outras fontes bibliográficas.

O Capítulo 1 apresenta os Sistemas Operacionais, alguns conceitos importantes, um breve histórico de sua evolução e uma classificação de seus tipos. O Capítulo 2 discute os processos computacionais, sua ocorrência e as principais questões associadas ao seu controle.

No Capítulos 3 falamos sobre o escalonamento de processos enquanto que o Capítulo 4 trata o gerenciamento de memória. Finalmente o Capítulo 5 aborda o gerenciamento de dispositivos periféricos.

Que este material possa ser útil aos estudantes desta disciplina, despertando o interesse no tema e principalmente motivando um estudo mais profundo e amplo.

O Autor

Capítulo 1

Introdução

Neste capítulo vamos apresentar os **sistemas operacionais**, seus objetivos e um breve histórico de sua evolução. Também proporemos uma forma simples para sua classificação e forneceremos alguns outros conceitos importantes.

1.1 Definindo os sistemas operacionais

Desde sua criação, os computadores sempre foram sistemas de elevada sofisticação em relação ao estágio tecnológico de suas épocas de desenvolvimento. Ao longo dos últimos 50 anos evoluíram incrivelmente e, embora tenham se tornado mais comuns e acessíveis, sua popularização ainda esconde sua tremenda complexidade interna.

Neste sentido, os sistemas operacionais, em termos de suas origens e desenvolvimento, acompanharam a própria evolução dos computadores. Deitel nos traz a seguinte definição de **sistema operacional**:

Vemos um sistema operacional como os programas, implementados como software ou firmware, que tornam o hardware utilizável. O hardware oferece capacidade computacional bruta. Os sistemas operacionais disponibilizam convenientemente tais capacidades aos usuários, gerenciando cuidadosamente o hardware para que se obtenha uma performance adequada. [DEI92, p. 3]

Nesta definição surgem alguns novos termos explicados a seguir. O *hardware* é o conjunto de dispositivos elétricos, eletrônicos, ópticos e eletromecânicos que compõe o computador, sendo a máquina física propriamente dita. O *hardware*, aparentemente identificável pelos dispositivos ou módulos que compõe um sistema computacional, determina as capacidades deste sistema. O *software* é o conjunto de todos os programas de computador em operação num dado computador. Já o *firmware* é representado por programas especiais armazenados de forma permanente no *hardware* do computador que

permitem o funcionamento elementar e a realização de operações básicas em certos dispositivos do computador, geralmente associadas a alguns periféricos e a execução de outros programas também especiais.

Na Figura 1.1 podemos identificar o *hardware* como sendo os dispositivos físicos, sua microprogramação e o *firmware* existente neste computador. Como exemplos de dispositivos existentes num sistema podemos citar os circuitos integrados de memória, as unidades de disco flexível ou rígido e o processador do sistema, sendo este último um dispositivo microprogramado. O *firmware* geralmente vem acondicionado em circuitos de memória não volátil (ROM, PROM ou EPROM) sendo os programas ali gravados escritos geralmente em linguagem de máquina e destinados a execução de operações especiais tal como a auto-verificação inicial do sistema (POST ou *power on self test*) e a carga do sistema operacional a partir de algum dispositivo adequado (*bootstrap*).

O *software* deste sistema ou os programas do sistema são representados pelo sistema operacional e todos os seus componentes (bibliotecas de funções e programas utilitários) além de todos os outros programas acessórios do sistema, tais como editores de texto, programas gráficos, compiladores, interpretadores de comando (*shells*), aplicativos de comunicação e ferramentas de administração e manutenção do sistema.

Os programas de aplicação são todos os demais *softwares*, desenvolvidos com finalidades particulares, que são utilizados num dado sistema computacional sob suporte e supervisão do sistema operacional, tais como planilhas eletrônicas, programas de correio eletrônico, navegadores (browsers), jogos, aplicações multimídia etc.

Jogos	Sistemas Específicos	Outros Sistemas
Editores	Compiladores	Shells
Sistema Operacional		
Firmware		
Microprogramação		
Dispositivos Físicos		

Figura 1.1: *Hardware*, *software*, *firmware* e o SO

Por si só, o *hardware* do computador dificilmente poderia ser utilizado diretamente e mesmos assim, exigindo grande conhecimento e esforço para execução de tarefas muito simples. Neste nível, o computador somente é ca-

paz de entender programas diretamente escritos em linguagem de máquina. Além disso, cada diferente tipo de computador possui uma arquitetura interna distinta que pode se utilizar de diferentes processadores que por sua vez requisitarão diferentes linguagens de máquina, tornando penosa e cansativa a tarefa dos programadores.

Desta forma, é adequada a existência de uma camada intermediária entre o *hardware* e os programas de aplicação que pudesse não apenas oferecer um ambiente de programação mais adequado mas também um ambiente de trabalho mais simples, seguro e eficiente.

Um **sistema operacional** é um programa, ou conjunto de programas, especialmente desenvolvido para oferecer, da forma mais simples e transparente possível, os recursos de um sistema computacional aos seus usuários, controlando e organizando o uso destes recursos de maneira que se obtenha um sistema eficiente e seguro.

Stallings, ao tratar dos objetivos e funções dos sistemas operacionais, afirma que:

Um sistema operacional é um programa que controla a execução dos programas de aplicação e atua como uma interface entre o usuário do computador o hardware do computador. Um sistema operacional pode ser pensado como tendo dois objetivos ou desempenhando duas funções: conveniência, pois faz o sistema computacional mais conveniente de usar; e eficiência, pois permite que os recursos do sistema computacional sejam usados de maneira eficiente. [STA96, p. 222]

Silberschatz utiliza praticamente a mesma definição, indicando que um sistema operacional é um ambiente intermediário entre o usuário e o *hardware* do computador no qual programas podem ser executados de forma conveniente e eficiente [SG00, p. 23]. Davis [DAV91], Shay [SHA96] e outros também apresentam idéias semelhantes.

Tanenbaum, por sua vez, define um sistema operacional através de uma ótica ligeiramente diferente:

O mais fundamental de todos os programas do sistema é o sistema operacional que controla todos os recursos computacionais e provê uma base sobre a qual programas de aplicação podem ser escritos. [TAN92, p. 1]

Os sistemas operacionais são uma camada de *software* que "envolve" os componentes físicos de um computador, intermediando as interações entre estes componentes e os usuários ou os programas dos usuários. Neste sentido é apropriado considerar que os sistemas operacionais podem ser vistos como uma extensão do próprio computador ou como gerenciadores dos recursos existentes neste computador.

Ao invés de lidar com a complexidade inerente ao *hardware*, o sistema operacional oferece a funcionalidade disponível no *hardware* através de uma interface de programação orientada a operação de cada tipo de recurso, proporcionando não apenas transparência, mas também isolando o *hardware* das aplicações. Assim temos que o comportamento do sistema operacional é como uma extensão do próprio *hardware* ou como uma máquina virtual, que possui características diferentes da máquina física real.

Imaginando que múltiplos programas em execução desejem fazer uso dos diversos recursos do *hardware*, não é razoável que o controle destes recursos seja transferido aos programadores pois isto acrescentaria uma sobrecarga desnecessária a cada programa, sem que fosse possível otimizar o uso dos recursos. Além disso, erros ou omissões, mesmo que involuntárias, poderiam provocar erros de dimensões catastróficas, acarretando perda de grandes quantidades de dados, violações importantes de segurança etc. O sistema operacional deve se encarregar de controlar os recursos do computador, garantindo seu uso adequado, buscando também otimizar tal uso objetivando um melhor eficiência do sistema, assim sendo, o sistema operacional se comporta como gerente dos recursos do computador.

1.2 Objetivos de um sistema operacional

A despeito do tipo, sofisticação ou capacidades do computador, um sistema operacional deve atender aos seguintes princípios:

1. Oferecer os recursos do sistema de forma simples e transparente;
2. Gerenciar a utilização dos recursos existentes buscando seu uso eficiente em termos do sistema; e
3. Garantir a integridade e a segurança dos dados armazenados e processados no sistema e também de seus recursos físicos.

Além destes objetivos, um sistema operacional também deve proporcionar uma interface adequada para que ele possa ser utilizado pelos seus usuários. Historicamente as primeiras interfaces dos sistemas operacionais eram baseadas em um conjunto de palavras-chave (comandos) e mensagens de diálogo que permitiam a execução de tarefas e a comunicação entre homem (o operador) e máquina. Estes comandos e mensagens definiam a Interface Humano-Computador (IHC) daquele sistema. Atualmente as interfaces baseadas em modo texto estão em desuso, sendo substituídas por interfaces gráficas mais modernas e simples que buscam facilitar a utilização do computador através de sua aparência atraente e uso intuitivo.

1.3 Breve histórico

O desenvolvimento dos sistemas operacionais pode ser melhor observado e compreendido quando acompanhamos a história do próprio computador. No breve resumo que segue pretende-se focar os principais eventos e movimentos relacionados ao desenvolvimento destas máquinas.

1.3.1 O início

Por volta de 1643, Blaise Pascal projeta e constrói uma máquina de calcular mecânica, que deu origem as calculadoras mecânicas utilizadas até meados do século XX. No final do século XVIII, Joseph-Marie Jacquard constrói um tear que utiliza cartões de papelão perfurado para determinar o padrão a ser desenhado no tecido, criando a primeira máquina programável.

Charles Babbage constrói em 1822 a máquina de diferenças e depois, partindo das idéias de Jacquard, projeta a máquina analítica, a qual não foi terminada. Recentemente comprovou-se que suas idéias eram corretas, sendo ele hoje reconhecido como pai do computador moderno. Em 1870 é construído uma máquina analógica para previsão de marés por William Thomson, que origina os computadores analógicos. Em 1890 o Censo Americano utiliza com grande sucesso as máquinas de tabular de Herman Hollerith, que funda em 1896 a *Tabulating Machine Company*. Alguns anos depois esta companhia se transformaria na IBM (*International Business Machines*).

Na década de 30 começam a surgir, em diferentes pontos do mundo, projetos de máquinas eletromecânicas e eletrônicas de calcular:

1934 Máquina eletromecânica programável do engenheiro Konrad Zuse.

1935 Início do projeto da máquina eletrônica ABC, baseada em válvulas, para resolução de sistemas, proposta pelo físico John Vicent Atanasoft.

1937 John Von Neumann, matemático húngaro, propõe uma arquitetura genérica para o computador, utilizada até hoje (veja Figura 1.2).

1939 Desenvolvida a primeira calculadora eletromecânica dos laboratórios Bell.

As descobertas da época motivaram cientistas de diversas especialidades a trabalhar no desenvolvimento dos então chamados cérebros eletrônicos.

1.3.2 Década de 1940

Os primeiros computadores eram realmente grandes máquinas de calcular. Compostas por circuitos baseados em relês e outros dispositivos eletromecânicos, estas máquinas eram muito grandes, lentas, consumiam muita

energia elétrica e eram de difícil operação. Esta tecnologia foi progressivamente substituída pelas válvulas eletrônicas, um pouco mais confiáveis e rápidas, embora muito mais caras. Com isso os computadores da época eram caríssimos, restringindo seu uso à organismos militares, agências governamentais e grandes universidades. O uso do computador era claramente experimental.

Um dos primeiros sistemas programáveis construídos foi o computador eletromecânicos Mark I, projetado em conjunto pela IBM e pela Universidade de Harvard, apresentado em 1944. Em 1946 o Exército Americano revela seu computador eletrônico digital, o ENIAC, utilizado para cálculos de balística externa, que vinha sendo utilizado a alguns anos.

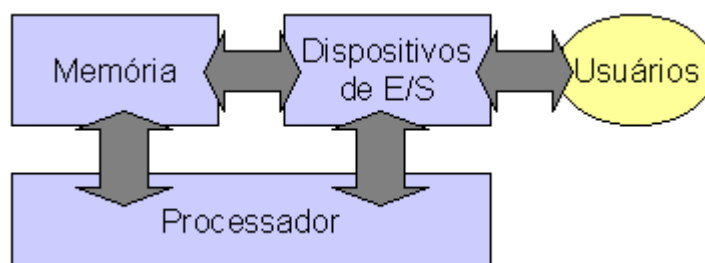


Figura 1.2: Arquitetura de Von Neumann

O uso destes sistemas exigia um grande grau de conhecimento de sua arquitetura e funcionamento. Os engenheiros exerciam o papel de programadores, determinando quais módulos deveriam ser interligados e em que ordem. Um grupo de técnicos treinados, de posse de esquemas de ligação, realizavam a conexão de tais módulos, de forma que a máquina pudesse ser energizada e os cálculos realizados. Nesta época o computador não era programável pois seu *hardware* era modificado para cada tipo de problema diferente, o que representava uma tarefa complexa e delicada.

Nesta década o matemático Von Neumann propôs a construção de sistema computacional baseada numa arquitetura composta por três blocos básicos (Figura 1.2) onde a seqüência de passos a ser executada pela máquina fosse armazenada nela própria sem necessidade de modificação de seu *hardware*. O computador de Von Neumann era uma máquina genérica, cujo bloco processador seria capaz de realizar um conjunto de operações matemáticas e lógicas além de algumas operações de movimentação de dados entre os blocos da máquina. Estas operações, chamadas de instruções, seriam armazenadas no bloco memória enquanto o bloco de dispositivos de E/S (Entrada e Saída) ou I/O (*Input and Output*) seria responsável pela entrada e saída dos dados, instruções e controle do sistema.

A seguir temos a Figura 3 representando o esquema básico de funcionamento dos processadores. Após ligado, um processador efetua um ciclo de busca por uma instrução na memória (*fetch*), a qual é decodificada

do ciclo seguinte (*decode*), que determina quais as ações necessárias para sua execução no último ciclo (*execute*). Após a execução repetem-se os ciclos de busca, decodificação e execução indefinidamente. A repetição desta seqüência só é interrompida através da execução de uma instrução de parada (*halt*) ou pela ocorrência de um erro grave.

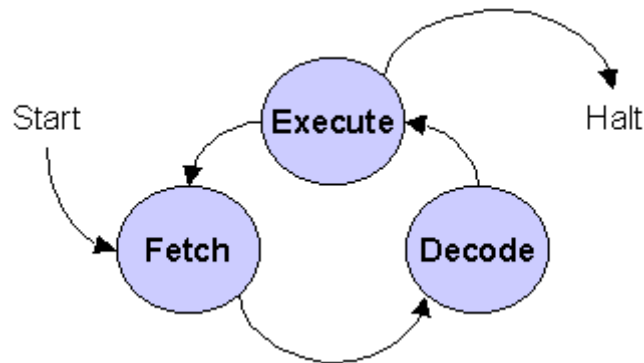


Figura 1.3: Funcionamento básico dos processadores

Assim sendo, um mesmo *hardware* poderia resolver diferentes tipos de problemas sem necessidade de qualquer modificação, bastando que uma seqüência adequada de instruções fosse carregada no computador. Com isto nascem os conceitos de programa, computador programável e com eles a programação de computadores. Apesar de todas as restrições tecnológicas da época, as idéias de Von Neumann revolucionaram a construção dos computadores e ditaram a nova direção a ser seguida.

1.3.3 Década de 1950

A descoberta do transistor deu novo impulso à eletrônica e aos computadores. Apesar de seu custo ainda alto, já era possível fabricar e vender computadores para grandes empresas e organismos governamentais, tanto que em 1951 surge o primeiro computador comercial, o Univac-I (*Universal Automatic Computer*) e em 1953 a IBM lança seu primeiro computador digital, o IBM 701.

Para programá-los ainda era necessário conhecer detalhes sobre seus circuitos e sobre o funcionamento de seus dispositivos. Tais exigências faziam que os computadores só pudessem ser utilizados por especialistas em eletrônica e programação. Mesmo tais especialistas tinham dificuldade em lidar com diferentes computadores, dado que cada computador possuía uma estrutura e funcionamento particulares.

A programação era feita em *assembly*, ou seja, diretamente em linguagem de máquina. Com a evolução dos computadores, tornou-se necessário criar pequenos programas que os controlassem na execução de tarefas cotidianas, tais como acionar certos dispositivos em operações repetitivas ou mesmo

simplificar a execução de novos programas. Surgiram assim os primeiros sistemas operacionais.

O uso individual do computador (conceito de *open shop*) era pouco produtivo, pois a entrada de programas constituía uma etapa muito lenta e demorada que na prática representava o computador parado.

Para otimizar a entrada de programas surgiram as máquinas leitoras de cartão perfurado (semelhantes as máquinas de tabular construídas por Herman Hollerith) que aceleravam muito a entrada de dados. Os programadores deveriam então escrever seus programas e transcrevê-los em cartões perfurados. Cada programa e seus respectivos dados eram organizados em conjuntos denominados *jobs* que poderiam ser processados da seguinte forma: os vários *jobs* de diferentes usuários eram lidos por uma máquina leitora de cartões que gravava os dados numa fita magnética. Esta fita era levada para o computador propriamente dito que lia os *jobs*, um a um, gravando uma outra fita magnética com os resultados de cada job. Esta fita de saída era levada a outro máquina que lia a fita e imprimia as listagens, devolvidas aos usuário juntamente com seus cartões.



Figura 1.4: Sistema Univac, 1951 (processamento em lote - *batch*)

Apesar da natureza seqüencial do processamento, para os usuários era como se um lote de *jobs* fosse processado a cada vez, originando o termo processamento em lote (*batch processing*). Os sistemas *batch* viabilizaram o uso comercial dos computadores, época em que grandes fabricantes de computadores começam a surgir.

Ainda na década de 50 surgiram a primeira linguagem de programação de alto nível (o IBM FORTRAN - *Formula Translator* - em 1957), a primeira unidade de disquetes comercialmente disponível no modelo IBM 305 e os mecanismos de interrupção implementados diretamente no *hardware* dos processadores. Em 1959 a DEC (*Digital Equipment Corporation*) apresenta seu minicomputador, o PDP-I, que fez grande sucesso comercial, originando uma grande linha de equipamentos.

1.3.4 Década de 1960

Buscando uma utilização mais eficiente e segura dos computadores, os sistemas operacionais foram se tornando cada vez mais complexos, passando a administrar os recursos do computador de forma cada vez mais sofisticada. Ao mesmo tempo em que se buscava um uso mais eficiente e seguro do computador, estudavam-se alternativas para que pessoas menos especializadas nos aspectos construtivos da máquina pudessem utilizar o computador, se concentrando em suas verdadeiras tarefas e ampliando as possibilidades de uso dos computadores.

Nesta década aparece o COBOL (*Commom Business Oriented Language*), linguagem de programação especialmente desenvolvida para o Pentágono americano para auxiliar o desenvolvimento de sistemas comerciais. Em 1961 a Farchild inicia a comercialização dos primeiros circuitos integrados. Em 1963 a DEC introduz o uso de terminais de vídeo e no ano seguinte surge o *mouse*. Um dos primeiros avanços ocorridos na década de 60 foi a utilização da multiprogramação. Segundo Deitel:

Multiprogramação é quando vários jobs estão na memória principal simultaneamente, enquanto o processador é chaveado de um job para outro job fazendo-os avançarem enquanto os dispositivos periféricos são mantidos em uso quase constante. [DEI92, p. 4]

Enquanto o processamento chamado científico era muito bem atendido pelo processamento em lote comum o mesmo não acontecia com processamento dito comercial. No processamento científico ocorre a execução de grande quantidade de cálculos com quantidades relativamente pequenas de dados, mantendo o processador ocupado na maior parte do tempo sendo que o tempo gasto com I/O (entrada e saída) era insignificante, daí este comportamento ser chamado *CPU Bounded*. Já no processamento comercial o processador permanece bastante ocioso dado que os cálculos são relativamente simples e o uso de I/O é freqüente dada a quantidade de dados a ser processada, temos um comportamento *I/O Bounded*.

A multiprogramação permitiu uma solução para este problema através da divisão da memória em partes, chamadas partições, onde em cada divisão um *job* poderia ser mantido em execução. Com vários *jobs* na memória o processador permaneceria ocupado o suficiente para compensar o tempo das operações mais lentas de I/O.

A utilização de circuitos integrados na construção de computadores comerciais e a criação de famílias de computadores compatíveis iniciadas com o IBM System/360 inaugurou uma nova era na computação e a expansão de sua utilização. Outra técnica utilizada era o *spooling* (*simultaneous peripheral operation on line*), isto é, a habilidade de certos sistemas operacionais em ler novos *jobs* de cartões ou fitas armazenado-os em uma área temporária

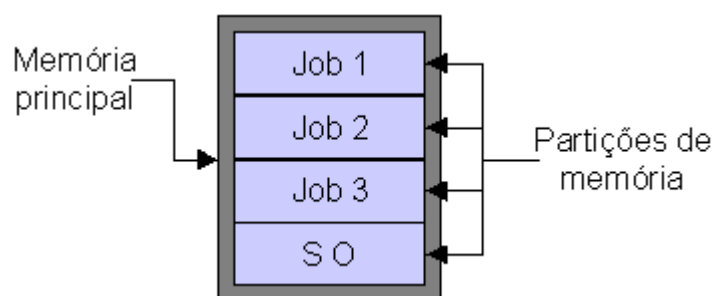


Figura 1.5: Partições de memória num sistema multiprogramado

do disco rígido interno para uso posterior quando uma partição de memória fosse liberada [TAN92, p. 9].

Apesar destas novas técnicas, os sistemas da época operavam basicamente em lote. Assim, enquanto satisfaziam as necessidades mais comuns de processamento comercial e científico, não ofereciam boas condições para o desenvolvimento de novos programas. Num sistema em lote, a correção de um problema simples de sintaxe poderia levar horas devido a rotina imposta: preparação dos cartões, submissão do job no próximo lote e a retirada dos resultados várias horas ou até mesmo dias depois.

Tais problemas associados ao desenvolvimento de *software* motivaram a concepção de sistemas multiprogramados, isto é, sistemas que permitissem o uso simultâneo do computador por diversos usuários através do pseudoparalelismo. O pseudoparalelismo poderia ser obtido com o chaveamento do processador entre vários processos que poderiam atender os usuários desde que fossem dotados de interfaces interativas. A idéia central destes sistemas era dividir o poder computacional de um computador entre seus vários usuários, fornecendo uma máquina virtual para cada um destes. Esta máquina virtual deveria passar a impressão de que o computador estava integralmente disponível para cada usuário, mesmo que isto não fosse verdade.

Nestes sistemas, denominados de sistemas em tempo repartido (*time sharing systems*), o tempo do processador era dividido em pequenos intervalos denominados *quanta* de tempo ou janelas temporais, como ilustra a (Figura 1.6). Tais *quanta* de tempo eram distribuídos seqüencialmente entre os processos de cada usuário, de forma que a espera entre os intervalos fosse imperceptível para os usuários. Depois que um *quanta* de tempo era distribuído para cada processo, se iniciava um novo ciclo de trabalho. A um dado processo seriam concedidos tantos *quanta* de tempo quanto necessário mas apenas um a cada ciclo. Com isto, a capacidade de processamento da máquina ficava dividida entre os usuários do sistema a razão de $\frac{1}{n}$, onde n é o número de usuários do sistema.

O mecanismo central de funcionamento destes sistemas é a interrupção. Uma interrupção é uma forma de parar a execução de um programa qual-

quer, conduzindo o processador a execução de uma outra rotina, previamente especificada que após ser executada por completo, permite o retorno ao programa original, sem prejuízo para execução do primeiro.

Os sistemas em tempo repartido também criaram as necessidades dos mecanismos de identificação de usuário (*userid* e *password*), dos mecanismos de início e término de sessão de trabalho (*login* ou *logon*), da existência de contas de usuário, etc.

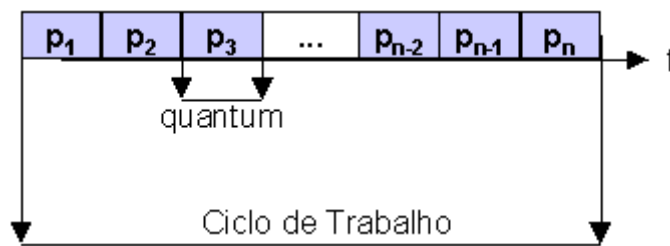


Figura 1.6: Ciclo de trabalho em sistemas de tempo repartido

Um dos processos de cada usuário poderia ser um interpretador de comandos (*shell*) que representaria a interface do computador para o usuário. Inicialmente estas interfaces eram implementadas como linhas de comando em terminais de teletipo e depois em terminais de vídeo. Como utilizavam exclusivamente caracteres também são conhecidas como interfaces em modo texto. As maiores dificuldades que surgiram durante o desenvolvimento destes sistemas foram a solução dos problemas associados ao compartilhamento dos recursos, à organização dos usuários e das tarefas do sistema, além dos mecanismos de segurança e privacidade.

1.3.5 Década de 1970 e 1980

Estas décadas são marcadas especialmente pelo crescimento em tamanho, sofisticação e complexidade dos sistemas computacionais. Aparece o termo *mainframe* (computador principal) e tornam-se cada vez mais comuns os centros de processamento de dados (CPDs) corporativos de uso privativo e bureaus de processamento de dados que vendiam serviços para terceiros. Surge o primeiro microprocessador comercial integrado em um único chip, o Intel 4004 em 1971. Na década de 70 começam a surgir pequenos computadores destinados ao uso pessoal ou doméstico, começando pelos microcomputadores da Xerox em Palo Alto em 1973, o Altair 8.800 destinado ao consumo em massa em 1974, o IBM 5100 em 1975 e o sucesso de vendas do Apple 2 lançado em 1976. Começam a ser desenvolvidas as primeiras aplicações comerciais dos microcomputadores.

A década de 80 viu o surgimento da geração do microcomputadores, o boom dos sistemas desktop a partir do lançamento do IBM-PC (Personal Computer) em 1981 e o desenvolvimento de uma enorme indústria de *hard-*

ware, *software* e serviços originadas nestes eventos. Na mesma década de 80 surgem empresas especializadas em arquiteturas de alto desempenho e são desenvolvidos os primeiros supercomputadores tais como os modelos 1-A ou X-MP da Cray Research.

1.3.6 Década de 1990

Esta década é marcada pelas estações de trabalho (*workstations*), pela computação pessoal portátil e pela interoperabilidade. Dispõe-se de microcomputadores cujo poder de processamento é maior do que os *mainframes* da década de 1970. Os supercomputadores, fabricados quase que artesanalmente, começam a ser tornar máquinas fantásticas, tal como o Cray Y-MP capaz de processar até 2.667 GFlops¹[NCA04]. Tais equipamentos são frequentemente empregados para simulações complexas, tais como as requeridas pela meteorologia, astrofísica, física atômica, farmacêutica, engenharia entre outras.

Os *notebooks*, *palmtops* e PDAs (*personal digital assistants*) representam o máximo em portabilidade e flexibilidade, e estão grandemente incorporados no cotidiano de executivos, profissionais de *design*, vendedores e outros profissionais liberais, mesmo aqueles cuja atividade fim não é a informática. Mesmo estes pequenos aparelhos possuem versões simplificadas de sistemas operacionais, permitindo que uma gama maior de serviços e funcionalidades sejam oferecidas. Além disso a eletrônica embarcada começa a ser tornar presente em automóveis e utensílios domésticos, além das tradicionais aplicações industriais.

Ferramentas de desenvolvimento rápido (RAD tools), ferramentas CASE sofisticadas para modelagem de dados e sistemas, complexos sistemas de CAD-CAE-CAM (Computer Aided Design, Engineering ou Manufacturing). Internet. Sistemas de correio eletrônico, grupos de discussão, educação à distância, multimídia, ATM, Java, Corba, sistemas distribuídos, processamento vetorial, processamento paralelo etc. Inúmeras tecnologias de projeto, desenvolvimento, aplicação, integração e interoperabilidade, adicionando diversidade, novas necessidades e maior complexidade aos sistemas operacionais contemporâneos.

Com isto, alguns milhões de linhas de programa são necessárias para a criação de um sistema operacional que atenda em parte tais necessidades, sendo necessário adaptá-los aos mais diversos tipos de aplicação e usuário. E para dar suporte a tremenda complexidade destes programas, a indústria de *hardware* vem continuamente desenvolvendo novos processadores também com velocidade cada vez maior, circuitos de memória de maior capacidade

¹A capacidade de processamento computacional (*computer power*) é usualmente medida em termos do número de operações em ponto flutuante completas por segundo (Flops).



Figura 1.7: Servidor, *workstation* e *notebook*

e velocidade bem como dispositivos periféricos cada vez mais sofisticados, tudo através das novas tecnologias.

1.3.7 Década de 2000

A supercomputação se torna superlativa. Exemplos disso são os novos sistemas da IBM e da Cray Research, conhecidos como IBM BlueSky e Cray X1 (Figura 1.8) que, respectivamente, tem capacidades de processamento de 6.323 TFlops [NCA04] e 52.4 TFlops [CRA04]. Graças à pesquisa e a evolução dos sistemas distribuídos, tal como as técnicas de *grid computing*, o emprego de *clusters* (agrupamentos) de computadores geograficamente dispersos e interligados através da Internet ou conectados através de redes locais de alto desempenho também vem se tornando mais comum, como alternativa mais econômica e de desempenho semelhante aos supercomputadores.

O desenvolvimento e miniaturização de novos componentes eletrônicos em adição e a utilização de novos processos de fabricação associados à grande expansão das redes digitais de telefonia móvel permitem que muitos milhões de usuários se beneficiem de equipamentos que somam as características de telefones portáteis com PDAs cada vez mais sofisticados.

Os computadores dedicados e os de uso genérico estão se tornando cada vez mais populares em contraste com as necessidades e exigências mais sofisticadas de seus usuários. O número de usuários de computador também cresce vertiginosamente, impulsionando os fabricantes a criarem novos produtos, serviços e soluções.

Com tudo isso a computação é, sem dúvida alguma, imprescindível como ferramenta para o homem moderno, tornando o desenvolvimento contínuo dos sistemas operacionais uma tarefa essencial.

1.4 Tipos de sistemas operacionais

Identificamos através da história dos sistemas operacionais alguns tipos de sistemas operacionais, os quais são comparados segundo alguns aspectos considerados importantes como pode ser visto na Tabela 1.1.



Figura 1.8: Supercomputador Cray X1

Tabela 1.1: Tipos de sistemas operacionais

Tipo de SO	Interativo	Tempo de Resposta	Produtividade (Throughput)	Multiusuário
Open Shop	Sim	Baixo Irregular	Baixa	Não
Batch Simples	Não	Alto Regular	Média Alta	Sim
Batch com Spooling	Não	Alto Regular	Média Alta	Sim
Tempo Repartido	Sim	Baixo Previsível	Média	Sim
Tempo Real	Sim	Baixo Previsível	Média	Sim

A **interatividade** é o aspecto que considera se o usuário utiliza diretamente o sistema computacional, podendo receber as respostas deste, sem intermediação e dentro de intervalos de tempo razoáveis.

O **tempo de resposta** (*response time*) é, desta forma, uma medida de interatividade, que representa o intervalo de tempo decorrido entre um pedido ou solitação de processamento (por exemplos, a entrada de um comando ou execução de um programa) e a resposta produzida pelo sistema (realização das operações solicitadas ou finalização do programa após sua execução completa). Tempos de resposta da ordem de alguns segundos configuram sistemas interativos, embora sejam admitidas esperas mais longas. Por exemplo, uma resposta produzida num intervalo de 30 segundos pode ser considerada normal levando-se em conta a natureza e complexidade da operação solicitada. Embora normal, intervalos desta ordem de grandeza ou superiores tornam enfadonho e cansativo o trabalho com computadores.

O **tempo de reação** (*reaction time*) também é outra medida de interatividade a qual considera o tempo decorrido entre a solicitação de uma ação e seu efetivo processamento.

Já a **produtividade** (*throughput*) é uma medida de trabalho relativa do sistema, expressa usualmente em tarefas completas por unidade de tempo, ou seja, é uma medida que relaciona o trabalho efetivamente produzido e o tempo utilizado para realização deste trabalho. Unidades possíveis do *throughput* são: programas por hora, tarefas por hora, *jobs* por dia etc.

A produtividade não deve ser confundida com o desempenho bruto do processador do sistema (sua capacidade de processamento), pois depende muito da arquitetura do sistema e do sistema operacional o quanto desta capacidade é efetivamente convertida em trabalho útil e o quanto é dispendida nas tarefas de controle e gerência do próprio sistema computacional.

Podemos notar que dentre estas medidas de performance (existem ainda diversas outras), algumas são orientadas ao usuário, tal como o tempo de resposta ou o tempo de reação; enquanto outras são orientadas ao sistema em si, tal como a taxa de utilização do processador ou a produtividade [DEI92, p. 423].

1.5 Recursos e ambiente operacional

O *hardware* do computador, ou seja, sua parte física, determina suas capacidades brutas, isto é, seus verdadeiros limites. Todos os elementos funcionais do computador são considerados **recursos** do sistema computacional e são, geralmente, representados pelos dispositivos que o compõe e que podem ser utilizados pelos usuários, ou seja: monitores de vídeo, teclado, *mouse*, mesas digitalizadoras, portas de comunicação serial e paralela, placas de rede ou comunicação, impressoras, *scanners*, unidades de disco flexível ou rígido, unidades de fita, unidades leitoras/gravadoras de CD, DVDs etc.

O sistema operacional aparece como uma camada sobre o *hardware* e *firmware*, mas simultaneamente envoltória deste. O sistema operacional está sobre o *hardware* e *firmware* pois deles depende para sua própria execução. Ao mesmo tempo é uma camada envoltória pois pretende oferecer os recursos do computador ao usuário do sistema minimizando os aspectos de como são tais dispositivos ou como serão feitas as operações que os utilizam. Desta forma o sistema operacional, através de sua interface, define uma nova máquina que é a combinação de um certo *hardware* com este sistema operacional.

O conjunto de *hardware* e sistema operacional, usualmente chamado de **plataforma** ou **ambiente operacional**, é aparentemente capaz de realizar tarefas de um modo específico ditado pela própria interface. Note que o ambiente operacional é distinto do *hardware*, pois o *hardware* do computador, por si só, não é capaz de copiar um determinado arquivo de uma unidade

de disco rígido para uma unidade de disquete. Para realizar esta cópia, uma série de procedimentos devem ser executados, indo desde o acionamento das unidades utilizadas, passando pela localização das partes do arquivo origem e das áreas disponíveis no disquete de destino, até a transferência efetiva dos dados.

Através da interface do sistema operacional, tal ação poderia ser possível através de um comando fornecido dentro de um console (uma interface de modo texto, tal como mostra a Figura 1.9, por exemplo:

```
copy so.dvi a:\
```

ou, se dispusermos de uma interface gráfica (Figura 1.10), através de uma operação visual frequentemente denominada *arrastar e soltar* (*drag and drop*) devido as ações realizadas com o *mouse*.

```
Prompt do MS-DOS: Miktex
<imagens/Note2Server.png> [15] <imagens/Cray81.png> [16]
Underfull \vbox (badness 10000) has occurred while \output is active [17]
Overfull \hbox (6.64293pt too wide) in paragraph at lines 524--527
\OTI/cw/r/n/10.95 operaçao vi-sual fre-quente-mente de-not-i-nada arrastar-e
-soltar <drag-and-drop>
<imagens/space300x200.png> <imagens/space300x200.png> [18]
Overfull \hbox (2.41502pt too wide) in paragraph at lines 357--358
[[\OTI/cw/r/n/10.95 Determinaçao de um am-bi-ente de tra-balho equiv-a-lente
para os usuarios,
[19]) <so_processos.tex [20]
Cap\vi }tulo 2.
) <so_escalamento.tex [21] [22]
Cap\vi }tulo 3.
) <so_memoria.tex [23] [24]
Cap\vi }tulo 4.
) <so_io.tex [25] [26]
Cap\vi }tulo 5.
) <so_rl.tex [27] [28]
Apêndice A.
) [29] [30] <so.bb1> [31] <so.aux> )
<see the transcript file for additional information>
Output written on so.dvi (41 pages, 77504 bytes).
Transcript written on so.log.
C:\1\Escolas\Faj\SO\texto>copy so.dvi a:\
```

Figura 1.9: Interface em modo texto (linha de comando ou console)

Enquanto o *hardware* isolado não permitia a realização direta da tarefa de copiar um arquivo, através do sistema operacional temos a impressão de que o computador se tornou capaz de realizar a tarefa desejada através do comando dado ou da operação visual efetuada, sem a necessidade do conhecimento de detalhes de como a tarefa é verdadeiramente realizada. A aparentemente expansão das capacidades do *hardware* do computador é, na verdade, uma simplificação obtida através da automatização da tarefa através de sua programação no sistema operacional. O *hardware* é realmente o realizador da tarefa, enquanto o sistema operacional apenas intermedia esta operação através de sua interface.

Observamos que a criação de uma interface, gráfica ou não, para um sistema operacional é extremamente importante ², pois determinam a criação de um **ambiente operacional** consistente. Tais possibilidades são bastante atraentes, pois se cada sistema operacional tem a capacidade de "transformar" o *hardware* de um computador em um determinado ambiente operacional, isto equivale a dizer que computadores diferentes, dotados de um mesmo sistema operacional, oferecerão transparentemente um ambiente operacional

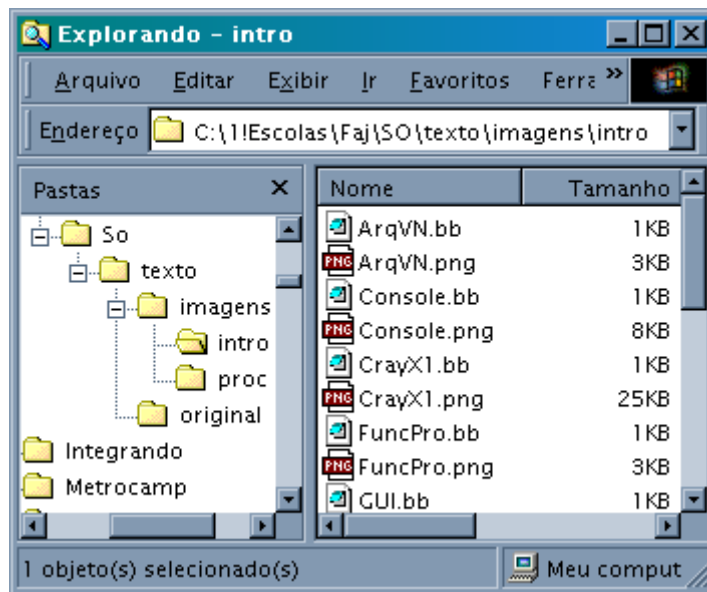


Figura 1.10: Interface gráfica

idêntico, ou seja, se comportarão como se fossem máquinas de um mesmo tipo. Por exemplo, o emprego de uma distribuição do Linux em microcomputadores baseados em processadores na Intel oferece o mesmo ambiente que *workstations* IBM utilizando este mesmo sistema operacional.

Esta possibilidade propicia as seguintes vantagens:

1. Determinação de um ambiente de trabalho equivalente para os usuários, que podem desconhecer as diferenças entre os diversos computadores dotados de um mesmo sistema operacional (plataforma de operação).
2. Criação de um ambiente de desenvolvimento semelhante, onde outros programas podem ser desenvolvidos, testados e utilizados em diferentes computadores dotados de um mesmo SO, com a conseqüente criação de uma plataforma de desenvolvimento.
3. Redução das necessidades de treinamento e aceleração do processo de familiarização e aprendizado no novo ambiente.

Para estudarmos os sistemas operacionais devemos primeiro nos familiarizar com o conceito de processo, fundamental para uma compreensão mais ampla dos problemas envolvidos e de suas soluções. Depois serão estudadas a administração da execução dos programas no sistema, o gerenciamento de memória e também dos dispositivos de entrada e saída.

²O projeto de interfaces consistentes, convenientes e uso simplificado, bem como sua avaliação, é uma disciplina compreendida como IHC ou Interface Humano-Computador.

Capítulo 2

Processos

O estudo e o desenvolvimento dos sistemas operacionais requer a compreensão de um conceito fundamental: processo computacional. Veremos que os processos computacionais constituem a unidade básica de administração de um sistema e, que junto deste importante conceito, surgem uma série de problemas que devem ser adequadamente equacionados dentro de qualquer sistema operacional.

2.1 O que é um processo computacional

Um **processo computacional** ou simplesmente **processo** pode ser entendido como uma atividade que ocorre em meio computacional, usualmente possuindo um objetivo definido, tendo duração finita e utilizando uma quantidade limitada de recursos computacionais.

Esta definição traz algumas implicações: apenas as atividades que acontecem num sistema computacional são compreendidas como sendo processos computacionais. Outro ponto importante é a duração finita, pois isto implica que um processo computacional, por mais rápido ou curto que possa ser tem sempre uma duração maior que zero, ou seja, não existem processos instantâneos. Além disso, um processo utiliza ao menos um dos recursos computacionais existentes para caracterizar seu estado.

Simplificando, podemos entender um processo como um programa em execução, o que envolve o código do programa, os dados em uso, os registradores do processador, sua pilha (*stack*) e o contador de programa além de outras informações relacionadas a sua execução.

Desta forma, temos que a impressão de um documento é um processo computacional assim como a cópia de um arquivo, a compilação de um programa ou a execução de uma rotina qualquer. Todas as atividades, manuais ou automáticas, que ocorrem dentro de um computador podem ser descritas como processos computacionais.

Atualmente quase todos os computadores são capazes de realizar diversas

tarefas ao mesmo tempo, onde cada uma destas tarefas pode representar um ou mesmo mais processos. Para funcionarem desta forma tais computadores são multiprogramados, ou seja, o processador é chaveado de processo em processo, em pequenos intervalos de tempo, isto é, o processador executa um programa durante um pequeno intervalo de tempo, para depois executar outro programa por outro pequeno intervalo de tempo e assim sucessivamente. Num instante de tempo qualquer, o processador estará executando apenas um dado programa, mas durante um intervalo de tempo maior ele poderá ter executado trechos de muitos programas criando a ilusão de paralelismo.

Este comportamento é, algumas vezes, chamado de paralelismo virtual ou pseudoparalelismo. Em computadores com dois ou mais processadores é possível a existência de paralelismo verdadeiro pois cada processador pode executar um processo independentemente.

A administração de vários diferentes programas em execução concomitante é o que permite o funcionamento eficiente dos computadores modernos, ao mesmo tempo conferindo-lhe complexa organização e estrutura pois tal administração não é simples e requer a consideração de muitos fatores e situações diferentes, mesmo que improváveis.

O termo processo (*process*) é muitas vezes substituído pelo termo tarefa (*task*) e pode assumir um dos seguintes significados:

- um programa em execução;
- uma atividade assíncrona;
- o espírito ativo de um procedimento;
- uma entidade que pode utilizar um processador ou
- uma unidade que pode ser despachada para execução.

2.1.1 Subdivisão dos processos

Outro ponto importante é que os processos computacionais podem ser divididos em **sub-processos**, ou seja, podem ser decompostos em processos componentes mais simples que o processo como um todo, o que permite um detalhamento da realização de sua tarefa ou do seu modo de operação. Esta análise aprofundada dos processos através de sua decomposição em sub-processos pode ser feita quase que indefinidamente, até o exagerado limite das micro-instruções do processador que será utilizado. O nível adequado de divisão de um processo é aquele que permite um entendimento preciso dos eventos em estudo, ou seja, depende do tipo de problema em questão e também da solução pretendida.

Processos tipicamente também podem criar novos processos. O processo criador é chamado de **processo-pai** (*parent process*) enquanto os processos

criados são denominados de **processos filhos** (*child process*). Um processo-filho também pode criar novos processos, permitindo a criação de árvores de processos hierarquicamente relacionados, como exemplificado na Figura 2.1.

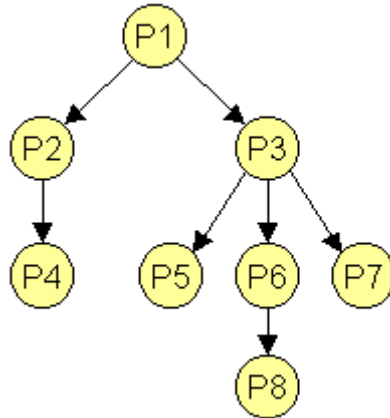


Figura 2.1: Subdivisão de um processo

2.2 Ocorrência de processos

É importante estudarmos os processos computacionais porque a razão de ser dos computadores é a realização de certas atividades de maneira mais rápida e confiável do que seria possível para o homem. Como cada processo precisa de recursos para ser executado e concluído, a ocorrência de processos significa a utilização de recursos do computador. Sendo assim, para que um sistema operacional possa cumprir com seu papel de gerente de recursos de um sistema computacional é fundamental um entendimento mais profundo dos processos computacionais e de suas particularidades como forma efetiva de criar-se sistemas operacionais capazes de lidar com as exigências dos processos em termos de recursos.

Um critério muito importante de análise dos processos computacionais é aquele que considera os processos segundo sua ocorrência, isto é, a observação de seu comportamento considerando o tempo. Neste caso teríamos os seguintes tipos de processos:

Seqüenciais São aqueles que ocorrem um de cada vez, um a um no tempo, serialmente, como que de forma exclusiva.

Paralelos aqueles que, durante um certo intervalo de tempo, ocorrem simultaneamente, ou seja, aqueles que no todo ou em parte ocorrem ao mesmo tempo.

2.2.1 Processos seqüenciais

Dados que os **processos seqüenciais** são aqueles que ocorrem um de cada vez no tempo, como numa série de eventos (veja Figura 2.2), temos que para um dado processo, todos os recursos computacionais estão disponíveis, ou seja, como só ocorre um processo de cada vez, os recursos computacionais podem ser usados livremente pelos processos, não sendo disputados entre processos diferentes, mas apenas utilizados da maneira necessária por cada processo.

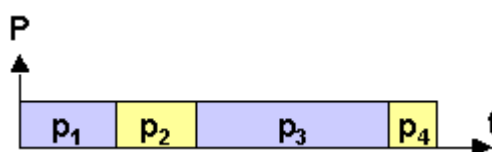


Figura 2.2: Diagrama-exemplo de processos seqüenciais

Esta aparente situação de simplicidade esconde um outro fato alarmante: como é muito improvável que um processo utilize mais do que alguns poucos recursos do sistema, todos os demais recursos não utilizados ficarão ociosos por todo o tempo de execução deste processo. No geral, com a execução de um único processo, temos que a ociosidade dos diversos recursos computacionais é muito alta, sugerindo que sua utilização é pouco efetiva, ou, em outros termos, inviável economicamente.

2.2.2 Processos Paralelos

Os **processos paralelos** são aqueles que, durante um certo intervalo de tempo, ocorrem simultaneamente, como mostra Figura 2.3. Se consideramos a existência de processos paralelos, então estamos admitindo a possibilidade de que dois ou mais destes processos passem, a partir de um dado momento, a disputar o uso de um recurso computacional particular.

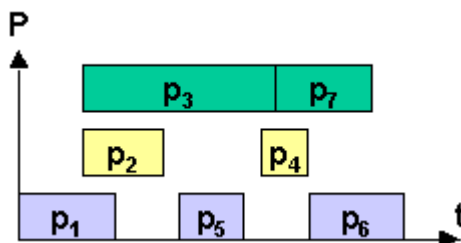


Figura 2.3: Diagrama-exemplo de processos paralelos

Considerando tal possibilidade de disputa por recursos e também sua natureza, os processos paralelos podem ser classificados nos seguintes tipos:

Independentes Quando utilizam recursos completamente distintos, não se envolvendo em disputas com outros processos.

Concorrentes Quando pretendem utilizar um mesmo recurso, dependendo de uma ação do sistema operacional para definir a ordem na qual os processos usarão o recurso.

Cooperantes Quando dois ou mais processos utilizam em conjunto um mesmo recurso para completarem uma dada tarefa.

Como não se pode prever quais os tipos de processos que existirão num sistema computacional, o sistema operacional deve estar preparado para administrar a ocorrência de processos paralelos concorrentes em quantidade, ou seja, deverá assumir a complexidade de administrar e organizar a coexistência de inúmeros processos diferentes disputando todos os tipos de recursos instalados no sistema.

Apesar da maior complexidade, a existência de processos paralelos permitem o melhor aproveitamento dos sistemas computacionais e mais, através do suporte oferecido pelo sistema operacional passa a ser possível a exploração do processamento paralelo e da computação distribuída.

2.3 Estado dos processos

Dado que um processo pode ser considerado como um programa em execução, num sistema computacional multiprogramado poderíamos identificar três **estados** básicos de existência de um processo:

Pronto (*Ready*) Situação em que o processo está apto a utilizar o processador quando este estiver disponível. Isto significa que o processo pode ser executado quando o processador estiver disponível.

Execução (*Running*) Quando o processo está utilizando um processador para seu processamento. Neste estado o processo tem suas instruções efetivamente executadas pelo processador.

Bloqueado (*Blocked*) Quando o processo está esperando ou utilizando um recurso qualquer de E/S (entrada e saída). Como o processo deverá aguardar o resultado da operação de entrada ou saída, seu processamento fica suspenso até que tal operação seja concluída.

Durante o **ciclo de vida** de um processo, isto é, desde seu início até seu encerramento, podem ocorrer diversas transições entre os estados relacionados, como ilustra a Figura 2.4. Devemos observar que entre os três estados básicos existem quatro transições possíveis, isto é, quatro situações de modificação de estado que correspondem às ações específicas do sistema operacional com relação aos processos: **Despachar** (*Dispatch*), **Esgotamento**

(*TimeRunOut*), **Bloqueio** (*Block*) e **Reativar** (*Awake*). Além destas quatro transições existe outras duas correspondentes a **Criação** (*Create*) e **Finalização** (*Terminate*) do processo.

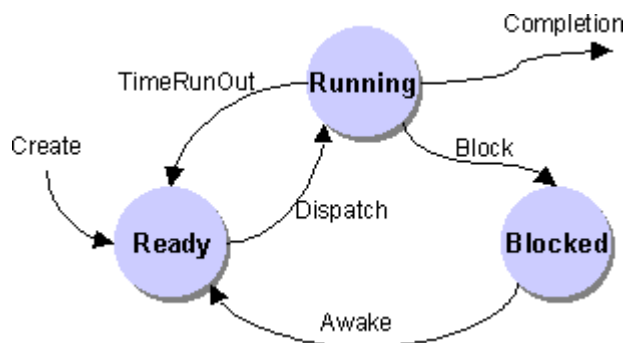


Figura 2.4: Diagrama-exemplo de processos paralelos

Quando solicitamos a execução de um programa, o sistema operacional cria (*Create*) um processo atribuindo a este um **número de identificação** ou seu PID (*Process Identifier*), um valor inteiro que servirá para distinguir este processo dos demais.

Após a criação, o processo é colocado no final de uma fila onde existem apenas processos prontos, ou seja, o estado inicial de um processo é definido como sendo o estado **Pronto** (*Ready*). Quando todos os processos existentes nesta fila, ou seja, criados anteriormente, já tiverem utilizado seu *quantum* (fração de tempo do processador), o sistema operacional acionará uma rotina especial para **Despachar** (*Dispatch*) o processo, ou seja, para efetivamente colocar o processo em execução. Nesta situação ocorrerá uma transição do estado do processo de **Pronto** para **Execução** (*Running*). Quando a fração de tempo destinada ao processo se esgotar, ocorrerá uma interrupção que devolverá o controle para o sistema operacional, fazendo-o acionar uma rotina especial (*TimeRunOut*) para retirar o processo do processador e recolocá-lo na fila de processos prontos. Esta é transição do estado **Execução** para o estado **Pronto**.

Nos casos em que o processo deseje utilizar algum dispositivo de entrada/saída, ou seja, efetuar uma operação de I/O, na solicitação deste recurso o próprio processo sairá do estado de **Execução** entrando voluntariamente no estado **Bloqueado** (*Blocked*) para utilizar ou esperar pela disponibilidade do recurso solicitado. Ao finalizar o uso do recurso, o sistema operacional recolocará o processo na lista de processos prontos, através da transição denominada **Reativação** ou *Awake*, que faz com que o processo passe do estados **Bloqueado** para **Pronto**.

A Tabela 2.1 mostra resumidamente as operações de transição de estado dos processos, indicando os respectivos estados inicial e final.

Tabela 2.1: Operações de transição de estado dos processos

Operação de Transição	Estado Inicial	Estado Final
Create()		Ready
Dispatch(PID)	Ready	Running
TimeRunOut(PID)	Running	Ready
Block(PID)	Running	Blocked
Awake(PID)	Blocked	Ready

Devemos destacar que a transição do estado **Execução** (*Running*) para **Bloqueado** (*Blocked*) é a única causada pelo próprio processo, isto é, voluntária, enquanto as demais são causadas por agentes externos (entidades do sistema operacional). Outro ponto importante é que os estados **Pronto** (*Ready*) e **Execução** (*Running*) são estados considerados ativos enquanto que o estado **Bloqueado** (*Blocked*) é tido como inativo.

Num sistema em tempo repartido a entidade que coordena a utilização do processador por parte dos processos é o **escalonador** (*scheduler*). O *scheduler* é uma função de baixo nível, que se utiliza de um temporizador (*timer*) do sistema para efetuar a divisão de processamento que, em última instância é uma mera divisão de tempo. Sendo assim está intimamente ligada ao hardware do computador.

Regularmente, a cada intervalo de tempo fixo ou variável, este temporizador dispara uma interrupção (um mecanismo especial de chamada de rotinas) que ativa uma rotina que corresponde ao escalonador do sistema. Esta rotina realiza algumas operações com os registradores do processador que forma que o resultado seja o chaveamento do processador para o próximo processo na fila de processos em estado **Pronto** (*Ready*). Ao encerrar-se esta interrupção o novo processo em execução é aquele preparado pelo escalonador.

Praticamente todas as outras funções do sistema operacional são acionadas por chamadas explícitas ou implícitas de suas próprias funções (chamadas de sistema ou *system calls*) enquanto outras entidades do próprio sistema operacional assumem a forma de processos que também compartilham o processamento. Neste sentido, as interrupções são um mecanismo importantíssimo para que os sistemas possa alcançar melhores níveis de produtividade, pois permitem que periféricos do sistema possam trabalhar de forma independente, caracterizando a execução de atividades em paralelo num sistema computacional.

De forma simplificada podemos dizer que o escalonador e a interrupção do temporizador do sistema se relacionam da seguinte maneira:

1. O processador empilha o *program counter* e também o conteúdo dos registradores do processador.

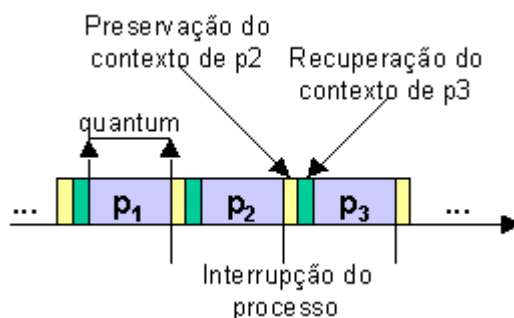


Figura 2.5: Representação do escalonamento de processos

2. O processador carrega um novo valor para o *program counter* a partir do vetor de interrupção onde existe uma rotina que realizará a troca de processos em execução.
3. Rotina modifica estado do processo cuja execução foi interrompida (o processo atual) para **Pronto** (*Ready*).
4. O conteúdo dos registradores e o *program counter* empilhados são copiados para uma área de controle própria do processo interrompido (cada processo tem tal área de controle, como veremos na seção 2.4.1), preservando o contexto do processo interrompido..
5. Rotina consulta escalonador para determinar qual será o próximo processo a ser executado (qual o processo que utilizará processador).
6. Rotina copia o conteúdo dos registradores e do *program counter* armazenado na área de controle do processo para a pilha, restaurando o contexto deste processo e, assim, alterando a seqüência de retorno da interrupção.
7. O temporizador é reprogramado.
8. A rotina é finalizada e, com isso, encerrando a interrupção.
9. Processador restaura seus registradores e o *program counter* com base nos conteúdo da pilha, passando a continuar a execução de um processo que tinha sido interrompido em um momento anterior.

2.4 PCB e tabelas de processos

Quando o modelo de administração de processos é adotado em um sistema operacional, para que este possa efetivamente controlar os processos existentes em um sistema, é comum a criação e manutenção de uma tabela para a organização das informações relativas aos processos chamada de **tabela**

de processos. Esta tabela é usualmente implementada sob a forma de um vetor de estruturas ou uma lista ligada de estruturas. Para cada processo existente existe uma entrada correspondente nesta tabela, ou seja, um elemento da estrutura destinado a armazenar os dados relativos ao respectivo processo, como mostra a Figura 2.6. Tal estrutura, projetada para armazenar as informações, recebe o nome de PCB.

2.4.1 PCB

O PCB (*Process Control Block* ou *Process Descriptor*) é uma estrutura de dados que mantém a representação de um processo para o sistema operacional. O PCB contém todas as informações necessárias para a execução do mesmo possa ser iniciada, interrompida e retomada conforme determinação do sistema operacional, sem prejuízo para o processo.

Apesar de ser dependente do projeto e implementação particulares do sistema operacional, geralmente o PCB contém as seguintes informações:

- identificação do processo (PID);
- estado corrente do processo;
- ponteiro para o processo pai (*parent process*);
- lista de ponteiros para os processos filho (*child processes*);
- prioridade do processo;
- lista de ponteiros para as regiões alocadas de memória;
- informações sobre horário de início, tempo utilizado do processador;
- estatísticas sobre uso de memória e periféricos;
- cópia do conteúdo do contador de programa (*program counter*);
- cópia do conteúdo dos registradores do processador;
- identificador do processador sendo utilizado;
- informações sobre diretórios raiz e de trabalho;
- informações sobre arquivos em uso;
- permissões e direitos.

A Figura 2.6 ilustra uma tabela de processos e os PCB associados.

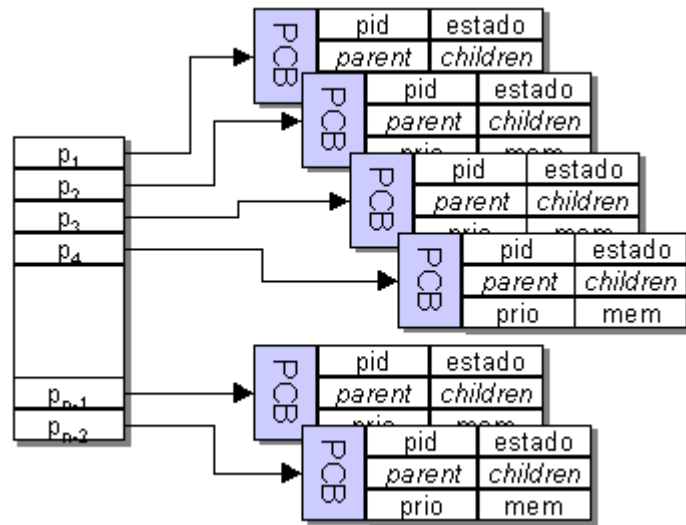


Figura 2.6: Organização de uma tabela de processos

2.4.2 Tabelas de processos

2.5 Operações sobre processos

Considerando os estados possíveis dos processos e as necessidades dos sistema operacional, temos que dispor das seguintes operações a serem realizadas sobre os processos:

- criação (*create*)
- destruição (*destroy*)
- suspensão (*suspend* ou *wait*)
- retomada (*resume*)
- troca de prioridade
- bloqueio (*block*)
- ativação (*activate*)
- execução (*execute* ou *dispatch*)
- comunicação inter-processo

A criação de um processo envolve algumas outras operações particulares:

- identificação do processo (determinação do PID)

- inserção do processo na lista de processos conhecidos do sistema
- determinação da prioridade inicial do processo
- criação do PCB
- alocação inicial dos recursos necessários

2.6 Funções do núcleo de sistema operacional

Todas as operações que envolvem os processos são controladas por uma parte do sistema operacional denominada **núcleo** (*core* ou *kernel*). Apesar do núcleo não representar a maior parte do sistema operacional, é a parcela mais importante e mais intensivamente utilizada, tanto que fica permanentemente alocada na memória primária do sistema.

Uma das mais importantes funções do núcleo do sistema operacional é o gerenciamento das interrupções, pois em grandes sistemas, um grande número delas é constantemente dirigida ao processador e seu efetivo processamento determina quão bem serão utilizados os recursos do sistema e, conseqüentemente, como serão os tempos de resposta para os processos dos usuários.

O núcleo de um sistema operacional constrói, a partir de uma máquina física dotada de um ou mais processadores, n máquinas virtuais, onde cada máquina virtual é designada para um processo. Tal processo, dentro de certos limites impostos pelo sistema operacional, controla sua máquina virtual de modo a realizar suas tarefas.

Resumidamente, o núcleo de um sistema operacional deve conter rotinas para que sejam desempenhadas as seguintes funções:

- gerenciamento de interrupções
- manipulação de processos (criação, destruição, suspensão, retomada etc.)
- manipulação dos PCBs (*Process Control Blocks*)
- troca de estados dos processos (execução, timerunout e ativação)
- intercomunicação de processos
- sincronização de processos
- gerenciamento de memória
- gerenciamento de dispositivos de E/S
- suporte a um ou mais sistemas de arquivos

- suporte à funções de administração do sistema

Devido ao uso intensivo destas rotinas, é comum que boa parte delas sejam programadas diretamente em *assembly*, permitindo assim grande otimização e, portanto, a maior eficiência possível em sua execução.

2.7 Competição por recursos

Quando afirmamos que existirão dois ou mais processos em execução paralela estamos admitindo a possibilidade de que alguns destes processos solicitem a utilização de um mesmo recurso simultaneamente. Conforme o tipo de recurso requisitado e também as operações pretendidas, é desejável que ocorra o compartilhamento de tal recurso ou, como é mais frequente, que seja exigido o uso individual e organizado de recurso. Obviamente o processo que recebe o direito de usar primeiramente o recurso é favorecido em relação aos demais, transformando tal situação numa **competição** pelos recursos do sistema.

A exigência de uso individual ou a possibilidade de compartilhamento caracterizam duas situações importantes: o código reentrante e as regiões críticas, que serão tratadas nas seções seguintes.

2.7.1 Regiões críticas

Quando um dado recurso computacional só pode ser utilizado por um único processo de cada vez, dizemos que este recurso determina uma **região crítica** (ou *critical section*), como representado na Figura 2.7. Sendo assim uma região crítica pode ser uma rotina de *software* especial ou um dispositivo de *hardware* ou uma rotina de acesso para um certo dispositivo do *hardware*.

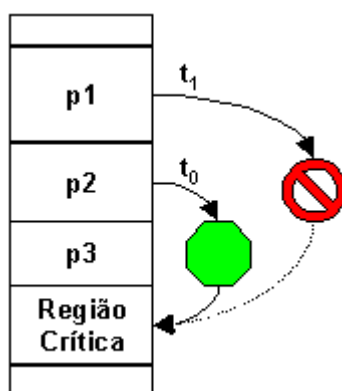


Figura 2.7: Representação de uma região crítica

Segundo Guimarães:

Uma região crítica é, no fundo, uma forma de administrar a concessão e devolução de um recurso comum. [GUI86, p. 81]

A situação que se deseja é a **exclusão mútua**, ou seja, quando um processo qualquer utiliza a região crítica, todos os demais, sejam quais forem, são excluídos, isto é, ficam impossibilitados de utilizá-la. Apenas quando a região crítica for liberada é que um outro processo terá acesso a mesma e também de forma exclusiva. Os processos que desejam utilizar uma região crítica são, portanto, processos paralelos concorrentes.

Quando acidentalmente acontece o acesso de um ou mais processos enquanto região crítica está ocupada ou quando dois ou mais processos entram a região crítica simultaneamente, dizemos estar ocorrendo um **acesso simultâneo**. Tal situação geralmente conduz a perda de dados de um ou mais dos processos participantes e, as vezes, o comprometimento da estabilidade do sistema como um todo.

Para prevenir que mais de um processo faça uso de uma região crítica, não é razoável que tal controle seja realizado pelos próprios processos, pois erros de programação ou ações mal intencionadas poderiam provocar prejuízos aos usuários ou comprometer a estabilidade do sistema.

Assim, o sistema operacional implementa uma rotina de tratamento especial denominada **protocolo de acesso** (seção 2.8) que, além de determinar os critérios de utilização do recurso, resolve as situações de competição bem como a organização de uma eventual lista de espera no caso de disputa do recurso por processos concorrentes.

2.7.2 Código reentrante

Em contraste com as regiões críticas, quando uma certa rotina de *software* pode ser utilizada simultaneamente por uma quantidade qualquer de processos, dizemos que esta rotina é um bloco de **código reentrante** ou **código público** (ou ainda *public code*), como ilustrado na Figura 2.8.

O código reentrante é uma situação bastante desejável, pois representa um compartilhamento benéfico para o sistema, onde um mesmo recurso é utilizado por diversos processos, aumentando a eficiência do sistema como um todo.

2.8 Protocolos de acesso

Um **protocolo de acesso** (*access protocol*) é composto por uma rotina de entrada e uma outra de saída. A rotina de entrada determina se um processo pode ou não utilizar o recurso, organizando um fila de espera (espera inativa) ou apenas bloqueando a entrada do processo (espera ativa). A rotina de saída é executada após o uso do recurso, sinalizando que este se encontra

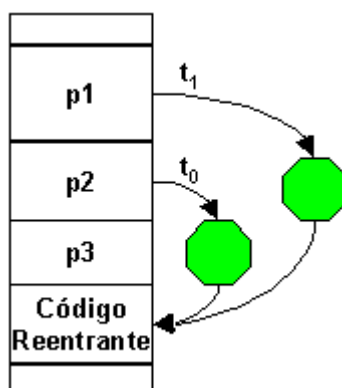


Figura 2.8: Representação de código reentrante

desocupado, ou seja, causando a liberação do recurso para outro processo em espera ou bloqueado.

Não se pode definir o acesso à recursos unicamente pela atribuição de prioridades fixas, pois podem ocorrer situações onde um processo de prioridade mais baixa não consegue utilizar a região crítica dado que sempre existem processos de maior prioridade, configurando um problema de **prioridade estática**. Existem também as situações de **bloqueio simultâneo**, onde os vários processos que estão em disputa pelo uso recurso são bloqueados de modo tal que o recurso continue sem uso, ou ainda de **adiamento infinito**, onde um processo é sistemática e indefinidamente bloqueado de utilizar o recurso. Todas estas situações também são inaceitáveis.

Devemos ainda considerar que o sistema operacional gerencia diferentes tipos de recursos computacionais, isto é, controla dispositivos com características bastante diferenciadas em termos de velocidade, capacidade e, principalmente utilização. O processador possui certas características enquanto a memória possui outras. Acontece o mesmo quando avaliamos a funcionalidade de unidades de disco e fita, impressoras, portas de comunicação e outros dispositivos que podem ser interligados a um sistema computacional. Desta forma o tratamento ideal que deve ser dado a uma impressora não pode ser aceitável para um outro tipo de dispositivo.

Torna-se óbvio que o controle destes diferentes dispositivos por si só adiciona uma razoável complexidade aos sistemas operacionais, mas quando consideramos que diferentes processos computacionais farão uso distinto e imprevisível destes recursos, tal complexidade e as inerentes dificuldades aumentam tremendamente, exigindo especial atenção durante o projeto do sistema.

O sistema operacional deve dispor de um protocolo de acesso diferente para cada tipo de recurso devido suas características próprias, as quais devem ser respeitadas durante sua utilização. Quando tais características são

entendidas e tratadas da forma adequada podemos obter situações que favoreçam o desempenho global do sistema.

De modo resumido, as características desejáveis de um protocolo de acesso estão relacioandas na Tabela 2.2

Tabela 2.2: Funções de um protocolo de acesso

Garantir	Evitar
Exclusão Mútua	Acesso Simultâneo Bloqueio Mútuo ou Simultâneo Adiamento Indefinido Prioridade Estática

Na Figura 2.9 temos um exemplo simplificado de um protocolo de acesso, com destaque as suas partes de entrada e saída. Aparentemente o problema parece resolvido, mas em situações de corrida, isto é, nas situações onde dois ou mais processos procuram acessar um recurso simultaneamente (ou algo próximo a isto), pode ocorrer uma violação do desejado acesso exclusivo: se dois processos executarem o teste da variável de controle X antes que um deles consiga fazer com que $X = 0$, então ambos utilizarão a região crítica.

Mesmo considerando que, na verdade, os processos não estão sendo executados paralelamente, se o primeiro deles é interrompido após o teste, mas antes da alteração da variável de controle, então qualquer outro poderá adentrar a região crítica. Quando o primeiro tem retomada sua execução, ele continuará seu processamento usando a região crítica, mesmo que esteja ocupada, pois já realizou o teste da variável de controle, provocando os problemas descritos.

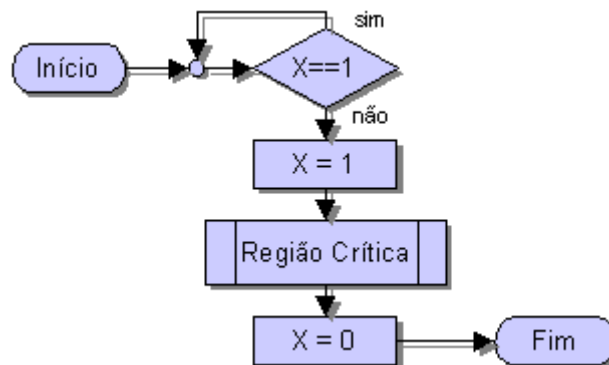


Figura 2.9: Protocolo de acesso simples

O protocolo exemplificado necessita de 2 a 4 instruções para ser implementado numa máquina convencional (vide Exemplo 2.1), portanto pode ocorrer uma interrupção de um dos processos durante a execução do trecho antes da atualização da variável X .

```

Laço:  LDA X   ; Carrega o acumulador com X
        BZ Laço ; Se zero repete o teste
        CLR X   ; Zera variável X
        :      ; Entra na RC

```

Exemplo 2.1 Código *assembly* de protocolo de acesso ineficiente

Uma análise cuidadosa permite ver que X, uma variável comum aos processos, também deveria ser usada de forma exclusiva, ou seja, também constitui uma região crítica e, assim, apenas adiou-se o problema de um nível.

A modificação do protocolo exemplificado utilizando-se uma instrução de troca, isto é, uma instrução que efetua a movimentação de conteúdo entre dois registradores distintos, permite resolver o problema, como ilustrado no Exemplo 2.2.

```

        CLR A   ; Zera o acumulador
Laço:  EXA X   ; Troca acumulador com X
        BZ Laço ; Se zero repete o teste
        :      ; Entra na RC
        :
        :
        :      ; Sai da RC
        MVI A,1 ; Coloca um no acumulador
        EXA X   ; Troca acumulador com X

```

Exemplo 2.2 Protocolo de acesso funcional com instrução EXA

2.8.1 Solução com instruções TST ou TSL

O uso de instruções de troca levou alguns fabricantes a criarem uma instrução especial denominada TST (*Test and Set*) ou TSL (*Test and Set Lock*). As instruções TST ou TSL foram primeiramente implementadas no final da década de 1950 e introduzidas na linha IBM/360. Tais instruções realizavam duas operações atômicas, isto é, de forma indivisível, não podendo ser interrompida durante sua execução, permitindo assim resolver os problemas de exclusão mútua encontrados até então. As operações realizadas através de uma instrução TST são ilustradas abaixo:

```

TST(v, x)  Equivale à:  v ? x
                          x ? 1 ; copia valor de x para v
                          ; seta valor de x para 1

```

Com isto poderíamos implementar um protocolo de acesso eficiente através da construção de duas primitivas de exclusão mútua como indicado

abaixo:

```

Enter_Region:
    TSL reg, flag      ; copia flag p/ reg e a seta para 1
    CMP reg, #0        ; compara flag com zero
    JNZ Enter_Region  ; se não zero loop (travado)
    RET                ; retorna, i.e., entrou na RC
Leave_Region:
    MOV flag, #0       ; armazena zero na flag
    RET                ; retorna, i.e., saiu da RC

```

Exemplo 2.3 Protocolo de acesso eficiente com instrução TST

Apesar da solução oferecida pelas instruções TST, o problema da exclusão mútua não estava completamente resolvido, pois tais instruções só resolvem tal questão em sistemas dotados de um único processador. Embora na época não se pretendesse construir computadores dotados de múltiplos processadores, percebia-se que tais instruções eram apenas um solução parcial e temporária.

2.8.2 Situações de corrida

Dizemos existir uma situação de corrida quando a execução de dois ou mais processos se dá, de tal forma, que tais processos solicitam o uso de uma região crítica simultaneamente ou praticamente nesta condição. As situações de corrida exigem protocolos de acesso extremamente bem elaborados para evitarmos o acesso simultâneo ou o bloqueio mútuo.

Para ilustrar o que são as situações de corrida, imaginemos um sistema multiusuário que dispõe de vários de terminais, onde cada terminal é monitorado por um processo especial que necessita efetuar a contagem das linhas de texto enviadas pelos usuários. A contagem do total das linhas de texto é mantida através de uma variável global de nome `linesEntered`. Assumamos que cada terminal em uso seja representado por um processo que possui uma cópia da rotina exibida no Exemplo 2.4 a qual atualiza o número de linhas de texto enviadas por cada terminal:

```

LOAD linesEntered
ADD 1
STORE linesEntered

```

Exemplo 2.4 Rotina local de contagem de linhas

Imaginando que o valor corrente de `linesEntered` é 21687 quando um processo executa as instruções `LOAD` e `ADD`, sendo interrompido pelo sistema operacional e, assim, deixando a valor 21688 no acumulador. Se um segundo

processo executa a rotina de forma completa, como o valor 21688 ainda não foi armazenado em memória, o valor 21697 será utilizado novamente resultando em 21688. Quando o primeiro processo retomar sua execução o valor 21688, presente no acumulador será armazenado novamente.

Concluimos que o sistema perdeu o controle do número de linhas enviado pois o valor correto deveria ser 21689! Isto ocorreu porque dois processos acessaram *simultaneamente* a região crítica delimitada pelas operações realizadas sobre a variável global `linesEntered`. Para evitar-se este problema cada processo deveria ter recebido acesso exclusivo às operações sobre a variável `linesEntered` que é uma região crítica.

Embora o problema de controle de linhas de texto enviadas pareça inútil, imagine que a contagem se refira a peças produzidas numa fábrica que conta com diversas linhas de produção: ao final de um período não se saberia com certeza quantas peças foram produzidas, sendo necessário contá-las no estoque. O mesmo vale para um sistema de controle de estoque, num problema semelhante envolvendo a contagem da retirada de peças. Quando um número mínimo é atingido, se faz necessária a compra de peças para que a produção não seja interrompida pela falta das mesmas. Se a operação de retirada não é contabilizada corretamente torna-se provável que a produção seja interrompida por falta de peças. Numa instituição financeira, o mesmo tipo de problema poderia se refletir no saldo da conta corrente de um cliente, onde dependendo da ordenação dos eventos, um débito ou um crédito poderiam deixar de ser efetuados, prejudicando a instituição ou o cliente. Torna-se claro que todas estas são situações inadmissíveis.

Percebemos ser muito importante que um sistema operacional ofereça mecanismos de controle adequado para seus recursos e também oferecendo suporte para o desenvolvimento de aplicações. Como o problema do acesso exclusivo também pode surgir em decorrência de aplicações com múltiplos usuários ou que apenas compartilhem dados de maneira especial, seria conveniente dispor-se de primitivas de controle para que os próprios programadores realizassem, com segurança, tal tarefa.

Tais primitivas de programação poderiam ser `EnterMutualExclusion` e `ExitMutualExclusion`, que respectivamente significam a indicação de entrada e saída de uma região crítica. Tais primitivas poderiam ser utilizadas como mostra o Exemplo 2.5, que resolve o problema de acesso à variável global `linesEntered`.

As primitivas `EnterMutualExclusion` e `ExitMutualExclusion` correspondem a implementação das regiões de entrada e saída de um protocolo de acesso genérico. Também poderiam ser utilizadas as instruções TST nesta implementação ou usadas outras soluções mais elegantes, tais como a de Dekker (seção 2.9) ou de Peterson (seção 2.10).

```
program ExclusaoMutua;
  { Variável global para contagem das linhas }
  var linesEntered: integer;

  { Procedimento correspondente ao primeiro terminal }
  procedure ProcessoUm;
  begin
    while true do
      begin
        LeProximaLinhaDoTerminal;
        EnterMutualExclusion;
        linesEntered := linesEntered + 1;
        ExitMutualExclusion;
        ProcessaALinha;
      end;
    end;
  end;

  { Procedimento correspondente ao segundo terminal }
  procedure ProcessoDois;
  begin
    while true do
      begin
        LeProximaLinhaDoTerminal;
        EnterMutualExclusion;
        linesEntered := linesEntered + 1;
        ExitMutualExclusion;
        ProcessaALinha;
      end;
    end;
  end;

  { Programa principal: ativação dos terminais }
  begin
    linesEntered := 0;
    parbegin
      ProcessoUm;
      ProcessoDois;
    parend;
  end.
```

Exemplo 2.5 Uso de primitivas de exclusão mútua

2.8.3 Requisitos de um protocolo de acesso

Antes de nos concentrarmos em outras soluções para o problema da exclusão mútua, é interessante analisar os requisitos desejáveis para um protocolo de acesso eficiente, os quais podem ser expressos através dos postulados de Dijkstra [GUI86, p. 78]:

1. A solução não deve impor uma prioridade estática entre os processos que desejem acessar a região crítica.
2. A única hipótese que pode ser feita quanto a velocidade de execução dos processos paralelos é que ela é não nula e, em particular, quando um processo acessa um região crítica ele sempre a libera depois de um tempo finito.
3. Se um processo é bloqueado fora da região crítica, isto não deve impedir que outros processos acessarem a região crítica.
4. Mesmo em improváveis situações de corrida, são inaceitáveis as situações de bloqueio mútuo ou acesso simultâneo de processos em uma região crítica.

2.9 A solução de Dekker

Até 1964 não se conhecia uma solução geral considerada satisfatória para o problema de exclusão mútua, excetuando-se os mecanismos de *hardware* implementados pelas instruções de troca e TTST. Neste ano o matemático holandês T. J. Dekker propôs uma solução para o problema de exclusão mútua de dois processos, a qual não necessitava instruções especiais implementadas pelo *hardware*, ou seja, uma solução que utilizava apenas os recursos comuns das linguagens de programação.

A engenhosa solução baseia-se em uma variável de controle para cada processo envolvido, no caso **Avez** e **Bvez**. Cada processo possui uma versão ligeiramente diferente do algoritmo do protocolo de acesso, pois cada uma das variáveis de controle só são alteradas pelos processos as quais pertencem. Além destas variáveis de controle individual, existe uma outra para determinação de prioridade (**Pr**), que pode ser manipulada por todos os processos envolvidos mas apenas após o uso da região crítica. Esta elegante solução tornou-se conhecida como *solução de Dekker para exclusão mútua de dois processos* e é ilustrada na Figura 2.10.

O fluxograma ilustrado na Figura 2.10 representa apenas o algoritmo da solução de Dekker correspondente ao processo **A**. Para obtermos a versão deste fluxograma correspondente ao processo **B**, devemos substituir as ocorrências da variável **Avez** por **Bvez** e vice-versa e depois as ocorrências de **B** por **A** e vice-versa. O funcionamento correto desta solução implica na

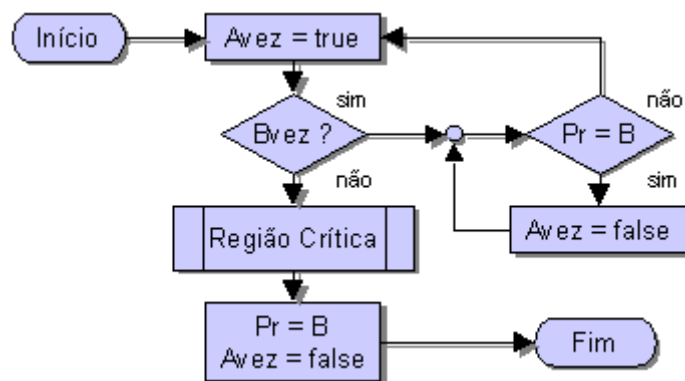


Figura 2.10: Algoritmo de Dekker para exclusão mútua de 2 processos

utilização destes dois algoritmos, cada um destinado a um dos processos envolvidos. Mesmo em situações de corrida, a solução garante a exclusão mútua, evitando o adiamento indefinido, a prioridade estática e o acesso simultâneo.

Para demonstrarmos uma aplicação da solução de Dekker, temos no Exemplo 2.6 uma sugestão para a resolução do problema de contagem das linhas enviadas pelos terminais usando a notação de paralelismo Pascal.

A solução proposta por Dekker apresenta as seguintes vantagens:

- é particularmente elegante, pois não necessita de instruções especiais no *hardware*;
- um processo fora de sua região crítica não impede (não bloqueia) outro processo de adentrá-la; e
- um processo que deseja entrar na região crítica o fará sem a possibilidade de adiamento infinito.

Em contrapartida esta solução possui algumas desvantagens que são:

- torna-se complexa para um número maior de processos pois exige a introdução de novas operações de teste a cada processo adicionado;
- é de difícil implementação pois necessita uma rotina diferente para cada processo; e
- impõe uma espera ativa ou espera ocupada, isto é, o processo bloqueado continua consumido tempo do processador na área de entrada do protocolo de acesso para verificar a possibilidade de entrada na região crítica.

```

program ExclusaoMutuaDekker;
  { Globais para contagem das linhas e prioridade }
  var linesEntered: integer;
      processo:      integer;

  procedure ProcessoUm; { Primeiro terminal }
  begin
    while true do
      begin
        LeProximaLinhaDoTerminal;
        while processo = 2 do; { espera RC livre }
          { Região Crítica }
          linesEntered := linesEntered + 1;
          { Fim da Região Crítica }
          processo := 2;
          ProcessaALinha;
        end;
      end;
  end;

  procedure ProcessoDois; { Segundo terminal }
  begin
    while true do
      begin
        LeProximaLinhaDoTerminal;
        while processo = 1 do; { espera RC livre }
          { Região Crítica }
          linesEntered := linesEntered + 1;
          { Fim da Região Crítica }
          processo := 1;
          ProcessaALinha;
        end;
      end;
  end;

  { Programa principal: ativação dos terminais }
  begin
    linesEntered := 0;
    processo := 1; { valor arbitrariamente escolhido }
    parbegin
      ProcessoUm;
      ProcessoDois;
    parend;
  end.

```

Exemplo 2.6 Solução do problema dos terminais através de Dekker

Dijkstra generalizou a solução de Dekker para n processos em 1965 e, como era de se esperar, a generalização mostrou-se bem mais complicada. Donald Knuth aperfeiçoou ainda mais a solução geral de Dijkstra, eliminando a possibilidade de adiamento infinito. Visto a introdução de instruções tipo TST o interesse nestas soluções é apenas histórico.

2.10 A solução de Peterson

Outras maneiras de implementar-se primitivas de exclusão mútua seria através da utilização do algoritmo de G. L. Peterson, publicado em 1981, que representa uma solução mais simples que a solução de Dekker.

A solução de Peterson, como colocado na Listagem 9, se baseia na definição de duas primitivas de exclusão mútua, utilizadas pelos processos que desejam utilizar a região crítica. Tais primitivas são as funções `enter_region()` e `leave_region()` que, respectivamente, devem ser utilizadas para sinalizar a entrada do processo na região crítica e sua saída da mesma.

```
#define FALSE 0
#define TRUE  1
#define N     2

int turn;
int interested[N];

void enter_region(int process) {
    int other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while (turn==process && interested[other]==TRUE);
}

void leave_region(int process) {
    interested[process] = FALSE;
}
```

Exemplo 2.7 Solução de Peterson para exclusão mútua de 2 processos

Note também o uso de um vetor contendo a sinalização de interesse dos processos em utilizar a região crítica. No exemplo é dado a solução para dois processos, que simplifica a determinação de interesse por parte dos demais processos.

Na solução proposta por Peterson temos que as funções `enter_region()` e `leave_region()` são equivalentes as primitivas `EnterMutualExclusion` e

ExitMutualExclusion respectivamente. Tal solução é bastante simples e pode ser facilmente implementada para n processos diferentes pois a mesma rotina serve a todos os processos. Sua única desvantagem é que impõe uma espera ocupada, tal como na solução de Dekker ou através do uso de instruções TST.

Os sistemas operacionais Microsoft Windows 9x/2000 oferecem um mecanismo eficiente e relativamente simples para o controle de acesso à regiões críticas, baseado numa estrutura especial designada `CriticalSection` e algumas primitivas de controle [CAL96, p. 224]:

- `InitializeCriticalSection`, que prepara o uso de uma região crítica;
- `EnterCriticalSection`, que corresponde a solicitação de uso da região crítica (região de entrada do protocolo de acesso);
- `LeaveCriticalSection`, que corresponde a finalização do uso da região crítica (região de saída do protocolo de acesso); e
- `DeleteCriticalSection`, que elimina as estruturas de controle declaradas para uma região crítica.

Tal mecanismo pode ser utilizado como sugerido no Exemplo 2.8, o qual está expresso na linguagem ObjectPascal do Borland Delphi.

```
{ declaração da variável especial de controle }
Section: TRTLCriticalSection;

{ inicialização da variável de controle }
InitializeCriticalSection(Section);

{ sinalização de entrada na região crítica }
EnterCriticalSection(Section);

{ código da região crítica é posicionado aqui }

{ sinalização de saída na região crítica }
LeaveCriticalSection(Section);

{ liberação da variável de controle }
DeleteCriticalSection(Section);
```

Exemplo 2.8 Mecanismo de exclusão mútua no MS-Windows 9x/2000

A declaração da variável especial de controle deve ser global, enquanto que a inicialização e liberação desta variável deve ocorrer respectivamente

antes de qualquer uso da região crítica e após a finalização de tal uso por diferentes processos da aplicação (no caso *threads* da aplicação). O código reconhecido como região crítica deve ser posicionado entre as primitivas de sinalização de entrada e saída da região crítica, sendo utilizado por todos as rotinas que possam estar em execução concomitante.

Esta solução resolve problemas de exclusão mútua originados por processos de uma mesma aplicação, ou seja, *threads* de um mesmo programa. Para sincronização de rotinas de programas diferentes deve ser utilizado outro mecanismo, denominado **Mutexes** no MS-Windows 9x, cujo funcionamento é semelhante aos semáforos (veja a seção 2.12.2).

Semáforos, Semáforos Contadores e Contadores de eventos são outros tipos de solução de podem solucionar problemas de exclusão mútua. Algumas destas alternativas serão vistas mais a frente, quando tratarmos da comunicação inter-processos (seção 2.12).

2.11 Deadlocks

Em sistema multiprogramado, o termo *deadlock*, ou seja, **bloqueio perpétuo** ou *impasse*, significa um evento que jamais irá ocorrer [DEI92, TAN92, SGG01]. Dizemos que um processo está em *deadlock* quando espera por um evento particular que jamais acontecerá. Igualmente dizemos que um sistema está em *deadlock* quando um ou mais processos estão nesta situação. Segundo Tanenbaum:

Um conjunto de processos está num bloqueio perpétuo quando cada processo do conjunto está esperando por um evento que apenas outro processo do conjunto pode causar. [TAN92, p. 242]

Observemos a Figura 2.11, onde temos ilustrado um *deadlock* que pode ocorrer em duas linhas de trens cujas interseções não são compartilháveis. Problemas semelhantes podem ocorrer no trânsito de uma metrópole.

Um bloqueio perpétuo pode ocorrer de diferentes maneiras:

- Quando um processo é colocado em espera por algo e o sistema operacional não inclui qualquer previsão para o atendimento desta espera dizemos que ocorreu o bloqueio perpétuo de um processo único (*one-process deadlock*).
- Quando se forma uma cadeia sucessiva de solicitações de recursos que culminam num arranjo circular, onde um processo A, que detêm um recurso R1, solicita um recurso R2 alocado para um processo B, que por sua vez está solicitando o recurso R1, em uso por A (veja a Figura 2.15). Como nenhum processo de dispõe a, voluntariamente, liberar o recurso que aloca, configura-se uma situação de bloqueio perpétuo.

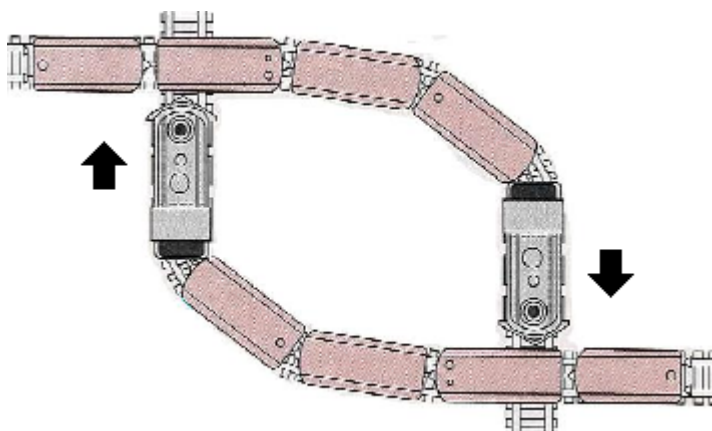


Figura 2.11: Representação de um *deadlock*

Independentemente do tipo, os *deadlocks* causam prejuízos sérios ao sistema, pois mesmo num *one-process deadlock*, recursos ficam alocados desnecessariamente, o que significa que restarão menos recursos para os demais processos. Nos *deadlocks* circulares, além da alocação desnecessária de recursos, podem ser formadas filas de esperas pelos recursos envolvidos, deteriorando o tempo de resposta do sistema, podendo até causar situações de instabilidade ou crash do sistema operacional.

Quando um processo é bloqueado indefinidamente, ficando em espera por um recurso, dizemos que está ocorrendo um adiamento indefinido ou bloqueio indefinido (respectivamente *indefinite postponement*, *indefinite blocking*). Como um processo nessa situação não pode prosseguir com sua execução devido a ausência de recursos, também dizemos que ele está em *starvation* (isto é, *estagnado*).

A maioria dos problemas que culminam com os *deadlocks* estão relacionados com recursos dedicados, isto é, com recursos que devem ser utilizados serialmente, ou seja, por um processo de cada vez [DEI92, p. 156].

2.11.1 Diagramas de processos e recursos

O estudo dos bloqueios perpétuos pode ser bastante facilitado através da utilização de diagramas especiais denominados diagramas de alocação de recursos ou diagramas de processo *versus* recursos ou ainda grafos de alocação de recursos [SGG01, p. 162].

Nestes diagramas existem apenas duas entidades, processos e recursos, interligadas por arcos direcionados. Os processos são representados através de quadrados ou retângulos. Os recursos são representados através de circunferências. Os arcos direcionados unindo processos e recursos são usados com dois significados distintos: requisição de recursos e alocação de recursos. Na Figura 2.12 temos os elementos construtivos possíveis deste tipo de

diagrama.

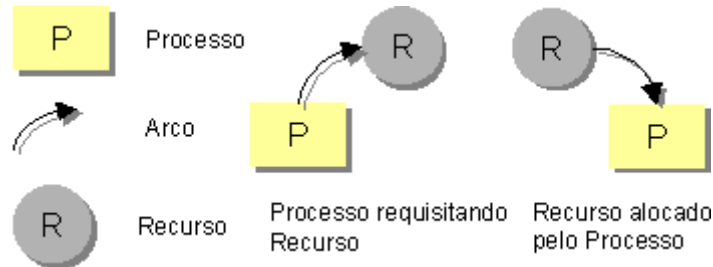


Figura 2.12: Elementos do Diagrama de Processos x Recursos

Recursos capazes de ser compartilhados podem ser representados por pequenas circunferências, dispostas dentro do recurso compartilhável, uma para cada unidade de sua capacidade de compartilhamento. Assim, através destes três elementos (retângulos, circunferências e arcos) podemos representar uma infinidade de diferentes situações envolvendo processos, seus pedidos de recursos (requests) e a alocação de recursos determinada pelo sistema operacional (grants).

Ao mesmo tempo, temos que são proibidas as construções ilustradas na Figura 2.13, isto é, processos requisitando ou alocando outros processos e, da mesma forma, recursos requisitando ou alocando outros recursos.

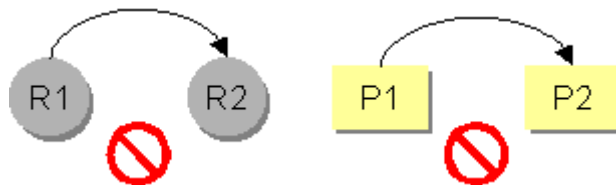


Figura 2.13: Situações proibidas no diagrama de processos e recursos

Usualmente não se ilustram o uso de memória e processador por parte dos processos, de modo que passam a não existir processos isolados, isto é, processos que não estejam utilizando algum outro tipo de recurso. Na Tabela 2.3 estão relacionados os recursos do sistema, os processos alocados e os processos requisitantes destes recursos.

Através da Tabela 2.3 podemos construir o diagrama de processos e recursos ilustrado na Figura 2.14.

Neste exemplo, embora exista disputa por recursos (processos B e E requisitam o uso do recurso W), não existe nenhum caminho fechado, ou seja, não existe qualquer *deadlock*, sendo assim este diagrama pode ser reduzido:

1. D finaliza o uso de Z.
2. Com Z livre, C libera W para alocar Z.

Tabela 2.3: Exemplo de alocação e requisição de recursos

Recurso	Processo Alocado	Processo(s) Requisitante
X	A	
Y	B	A
W	C	B, E
Z	D	C

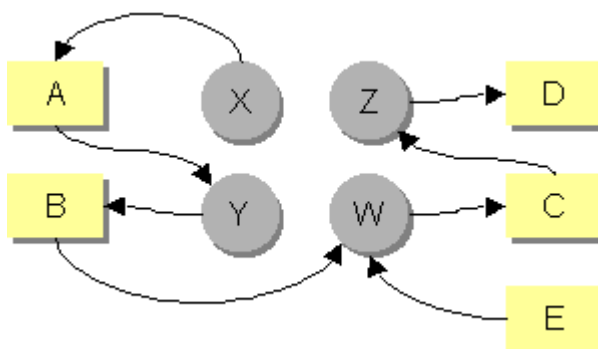


Figura 2.14: Diagrama de processos e recursos da Tabela 2.3

3. Com W livre, ou B ou E poderão alocá-lo. Se B for favorecido nesta disputa, alocando W, embora E permaneça em espera, Y será liberado.
4. Com Y livre, A libera X para alocar Y.
5. Não surgindo novos processos, B libera W.
6. Com W livre, E pode prosseguir sua execução.

A redução, independentemente do tempo que os processos utilizarão os recursos alocados, mostra que tais processos serão atendidos dentro de um intervalo de tempo menor ou maior. Apesar de questões relacionadas com desempenho, esta a situação desejável em um sistema.

Diferentemente, a alocação e requisição de recursos em um sistema pode configurar um *deadlock*. Utilizando o diagrama de alocação de recursos, poderíamos representar um *deadlock* envolvendo dois processos e dois recursos, como ilustra a Figura 2.15.

Através destes diagramas, fica clara a situação da formação de uma cadeia circular (um caminho fechado) ligando uma sequência de requisições e alocações de recursos do sistema por parte de um grupo de processos.

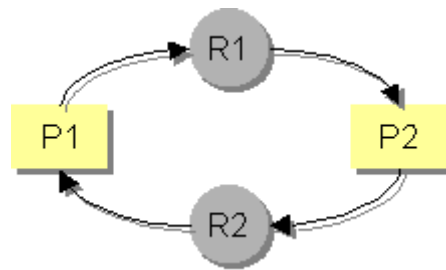


Figura 2.15: Representação de *deadlock* envolvendo 2 processos

2.11.2 Condições para ocorrência de *deadlocks*

Coffman, Elphick e Shoshani (1971) afirmam que existem quatro condições para a ocorrência de um *deadlock*:

1. Os processos exigem controle exclusivo dos recursos que solicitam (condição de exclusão mútua).
2. Os processos mantêm alocados recursos enquanto solicitam novos recursos (condição de espera por recurso)
3. Os recursos não podem ser retirados dos processos que os mantêm alocados enquanto estes processos não finalizam seu uso (condição de ausência de preemptividade).
4. Forma-se uma cadeia circular de processos, onde cada processo solicita um recurso alocado pelo próximo processo na cadeia (condição de espera circular).

Segundo Tanenbaum [TAN92, p. 245] existem basicamente quatro alternativas para o tratamento dos bloqueios perpétuos, ou seja, quatro estratégias básicas para resolvermos os problemas dos *deadlocks*:

1. Ignorar o problema (algoritmo da avestruz)
Apesar de não parecer uma solução para o problema, devem ser consideradas a possibilidade de ocorrência dos *deadlocks* e os custos computacionais associados ao seu tratamento. A alternativa mais simples realmente é ignorar o problema e conviver com a possibilidade de sua ocorrência. Os sistemas UNIX utilizam esta aproximação favorecendo aspectos de performance em situações mais comuns.
2. Detecção e recuperação dos *deadlocks*
Outra alternativa é permitir que os bloqueios ocorram, procurando-se detectá-los e recuperá-los. Deve-se utilizar algum algoritmo que produza um diagrama de alocação de recursos, analisando em busca

de caminhos fechados (*deadlocks*) utilizando alguma técnica de recuperação. Estes algoritmos especiais, que apesar da sobrecarga que provocam, podem evitar maiores transtornos no sistema possibilitando que na ocorrência dos *deadlocks* estes sejam descobertos e eliminados.

3. Prevenção dinâmica

Os *deadlocks* podem ser prevenido através de procedimentos cuidadosos de alocação que analisam constantemente a possibilidade de formação de cadeias circulares. Tais algoritmos também são complexos e acabam por onerar o sistema computacional.

4. Prevenção estrutural

Os *deadlocks* podem ser estruturalmente eliminados de um sistema através da negação de uma ou mais das quatro condições de ocorrência. Para isto devem ser tratadas as condições de Coffman *et al.*, o que é resumidamente apresentado na Tabela 2.4 abaixo.

Tabela 2.4: Condições de Coffman *et al.* para prevenção de *deadlocks*

Condição	Aproximação
Exclusão Mútua	Colocar todos os recursos do sistema em <i>spool</i> .
Retenção e Espera	Exigir a alocação inicial de todos os recursos necessários.
Ausência de Preemptividade	Retirada de recursos dos processos.
Espera Circular	Ordenação numérica dos pedidos de recursos.

2.11.3 Recuperação de *deadlocks*

Ainda que os *deadlocks* ocorram, dentro de certas circunstâncias é possível resolvê-los, isto é, recuperá-los ou eliminá-los, utilizando algumas técnicas:

1. Recuperação através de preempção

Retirando-se algum recurso envolvido no bloqueio perpétuo do processo que o aloca permite a quebra do caminho fechado e conseqüente solução do *deadlock*. O problema reside que nem sempre um recurso pode ser retirado de um processo sem efeitos colaterais prejudiciais a este processo.

2. Recuperação através de operações de *rollback*

Exige a implementação de *checkpoints*, isto é, um mecanismo de armazenamento de estados seguros do sistema através da cópia dos estados

individuais dos processos em arquivos especiais. Isto possibilita que tais estados sejam retomados a partir daquele ponto. Esta solução, além da difícil implementação, exige muitos recursos e tem elevado custo computacional, embora resolva bem o problema.

3. Recuperação através de eliminação de processos

Esta é maneira mais simples, embora também a mais drástica. Um ou mais dos processos identificados como envolvidos no bloqueio perpétuo podem ser sumariamente eliminados, de modo que o bloqueio seja resolvido. Enquanto alguns processos podem ser seguramente reiniciados (p.e., uma compilação), procedimentos de atualização em bancos de dados nem sempre podem ser interrompidos e reiniciados. A eliminação de processos que não podem ser simplesmente reiniciados pode provocar prejuízos ao sistema.

2.11.4 Prevenção de *deadlocks*

A prevenção de *deadlocks* é a estratégia preferencialmente adotada pelos projetistas de sistemas, adotando-se uma política que assume o custo da prevenção como alternativa aos prejuízos possíveis da ocorrência dos *deadlocks* e de sua eliminação. Para prevenir-se a ocorrência dos *deadlocks* podem ser adotadas uma ou mais das seguintes estratégias, tal como proposto por Havender (1968):

1. Um processo só pode solicitar um recurso se liberar o recurso que detêm.
2. Um processo que têm negado o pedido de recurso adicional deve liberar o recursos que atualmente detêm.
3. Se a solicitação de recursos ocorrer em ordem linear ascendente, a espera circular não consegue se formar.

Mesmo com as quatro condições estando presentes, é possível evitar-se a ocorrência dos *deadlocks* utilizando-se o algoritmo do banqueiro de Dijkstra (65). Antes disto devemos diferenciar estados seguros e inseguros.

2.11.5 Estados seguros e inseguros

A maioria dos algoritmos conhecidos para prevenção de *deadlocks* se baseia no conceito de estado seguro. Um estado seguro é aquele em que existe garantia que todos os processos poderão ser finalizados considerando (1) suas necessidades em termos de recursos e (2) os recursos efetivamente disponíveis no sistema [DEI92, p. 166] [TAN92, p. 254]. Desta forma os recursos cedidos aos processos serão devolvidos ao sistema para serem alocados para outros processos, numa seqüência de estados seguros (veja a Figura 2.16).

T1	Tem	Máx
A	3	9
B	2	4
C	2	7
Livre: 3		

T2	Tem	Máx
A	3	9
B	4	4
C	2	7
Livre: 1		

T3	Tem	Máx
A	3	9
B	0	-
C	2	7
Livre: 5		

T4	Tem	Máx
A	3	9
B	0	-
C	7	7
Livre: 0		

T5	Tem	Máx
A	3	9
B	0	0
C	0	0
Livre: 7		

Figura 2.16: Seqüência de estados seguros

Por outro lado, um estado considerado como inseguro é aquele em não existe a garantia de devolução dos recursos devidos pois o processo não recebe todos os recursos de que necessita não devolvendo ao sistema aqueles eventualmente já alocados [DEI92, p. 166] [TAN92, p. 254]. Esta situação é ilustrada na Figura 2.17, partindo da mesma situação inicial colocada na Figura 25. A maior consequência de um estado inseguro é que existem grandes chances de ocorrer um deadlock a partir desta situação sendo por isso necessário evitá-lo.

T1	Tem	Máx
A	3	9
B	2	4
C	2	7
Livre: 3		

T2	Tem	Máx
A	4	9
B	2	4
C	2	7
Livre: 2		

T3	Tem	Máx
A	4	9
B	4	4
C	2	7
Livre: 0		

T4	Tem	Máx
A	4	9
B	0	-
C	2	7
Livre: 4		

Figura 2.17: Seqüência de estados inseguros

A ocorrência de um estado inseguro, bem como de um estado seguro, depende estritamente da ordem com os recursos disponíveis são alocados e liberados pelos processos envolvidos. O sistema operacional, portanto, deve analisar se pode ou não atender plenamente as necessidades de um processo antes de ceder recursos que não poderão ser devolvidos.

2.11.6 Algoritmo do banqueiro

O algoritmo do banqueiro, proposto por Dijkstra em 1965, é uma solução clássica no estudo dos sistemas operacionais que visa ilustrar as questões associadas a concessão de recursos aos processos e as conseqüências possíveis destas concessões [DEI92, p. 167] [TAN92, p. 256]. Este algoritmo efetua um mapeamento dos recursos e processos de forma a considerar, a cada pedido de uso de um recurso, se tal alocação leva a um estado seguro ou não. Se o estado seguinte é seguro, o pedido é concedido, caso contrário tal solicitação é adiada até que possa conduzir a um estado seguro.

Na prática, o problema desta solução é que cada processo deve especificar, inicialmente, a quantidade máxima de cada recurso que pretenda utilizar. Além disso, a quantidade de processos varia a cada instante em sistemas reais. Se um novo processo conduzir a um estado inseguro, sua criação deverá ser adiada, o que também pode gerar um *deadlock*. Outro ponto é que a quantidade de recursos pode variar (geralmente diminuir com falhas no sistema) aumentando muito a complexidade desta solução e, portanto, sua aplicação prática.

2.12 Comunicação de processos

A comunicação entre processos ou comunicação inter-processo (IPC ou *Inter Process Communication*) é uma situação comum dentro dos sistemas computacionais que ocorre quando dois ou mais processos precisam se comunicar, isto é, quando os processos devem compartilhar ou trocar dados entre si. A comunicação entre processos pode ocorrer em várias situações diferentes tais como:

- redirecionamento da saída (dos resultados) de um comando para outro,
- envio de arquivos para impressão,
- transmissão de dados através da rede,
- transferência de dados entre periféricos, etc.

Tal comunicação se dá, geralmente, através da utilização de recursos comuns aos processos envolvidos na própria comunicação. Como não é razoável que tal comunicação envolva mecanismos de interrupção devido a sua complexidade e limitações de performance, as interrupções são reservadas para a administração do sistema em si. Para a comunicação inter-processo é necessário algum mecanismo bem estruturado. Veremos alguns mecanismos possíveis para a comunicação de processos destacando-se:

- Buffers

- Semáforos
- Memória Compartilhada

2.12.1 *Buffers* e operações de *sleep* e *wakeup*

Um problema típico é o do produtor-consumidor, onde dois processos distintos compartilham um *buffer*, uma área de dados de tamanho fixo que se comporta como um reservatório temporário [DEI92, p. 90] [TAN92, p. 39]. O processo produtor coloca informações no *buffer* enquanto o processo consumidor as retira de lá.

Se o produtor e o consumidor são processos seqüenciais, a solução do problema é simples, mas caso sejam processos paralelos passa a existir uma situação de concorrência. Mesmo nos casos onde existam múltiplos produtores ou múltiplos consumidores o problema encontrado é basicamente o mesmo. Este é um problema clássico de comunicação inter-processo, tais como os problemas do jantar dos filósofos e do barbeiro dorminhoco discutidos em detalhes por Tanenbaum [TAN92, p. 56].

Programas que desejam imprimir podem colocar suas entradas (nomes dos arquivos a serem impressos ou os arquivos de impressão propriamente ditos) em uma área de *spooling*, denominada de *printer spool*. Um outro processo (tipicamente um *daemon* de impressão) verifica continuamente a entrada de entradas no *spool*, direcionando-as para uma ou mais impressoras existentes quando estas se tornam ociosas, com isto retirando as entradas da área de *spool*. É claro que a área reservada para o *spool* é finita e que as velocidades dos diversos produtores (programas que desejam imprimir) pode ser substancialmente diferente das velocidades dos consumidores (das diferentes impressoras instaladas no sistema).

A mesma situação pode ocorrer quando diversos processos utilizam uma placa de rede para efetuar a transmissão de dados para outros computadores. Os processos são os produtores, enquanto o *hardware* da placa e seu código representam o consumidor. Em função do tipo de rede e do tráfego, temos uma forte limitação na forma que a placa consegue *consumir* (transmitir) os dados produzidos pelos programas e colocados no *buffer* de transmissão.

Existem outras situações semelhantes, o que torna este problema um clássico dentro do estudo dos sistemas operacionais. Na Figura 2.18 temos um esquema do problema produtor-consumidor.

Tanto o *buffer* como a variável que controla a quantidade de dados que o *buffer* contém são regiões críticas, portanto deveriam ter seu acesso limitado através de primitivas de exclusão mútua, desde que isto não impusesse esperas demasiadamente longas aos processos envolvidos.

Dado que o *buffer* tem um tamanho limitado e fixo podem ocorrer problemas tais como:

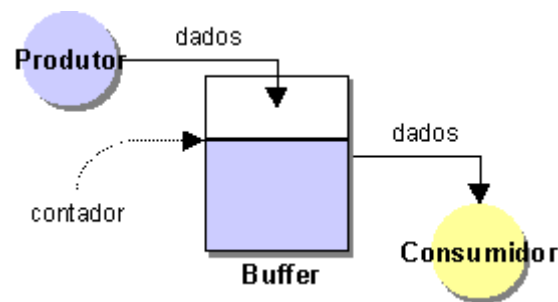


Figura 2.18: Problema do produtor-consumidor

- o produtor não pode colocar novas informações no *buffer* porque ele já está cheio; ou
- o consumidor não pode retirar informações do *buffer* porque ele está vazio.

Nestes casos tanto o produtor como o consumidor poderiam ser *adormecidos*, isto é, ter sua execução suspensa, até que existisse espaço no *buffer* para que o produtor coloque novos dados ou existam dados no *buffer* para o consumidor possa retirá-los. Uma tentativa de solução deste problema utilizando as primitivas *sleep* (semelhante a uma operação *suspend*) e *wakeup* (semelhante a uma operação *resume*) pode ser vista no Exemplo 2.9.

A solução dada é considerada parcial pois pode ocorrer que um sinal de *wakeup* seja enviado a um processo que não esteja logicamente *adormecido*, conduzindo os dois processos a um estado de suspensão que permanecerá indefinidamente, como indicado na seqüência de etapas a seguir.

1. O *buffer* se torna vazio.
2. O consumidor lê `contador=0`.
3. O escalonador interrompe o consumidor.
4. O produtor produz novo item e o coloca no *buffer*.
5. O produtor atualiza variável `contador=1`.
6. O produtor envia sinal *wakeup* para consumidor pois `contador=1`.
7. O sinal de *wakeup* é perdido pois o consumidor não está logicamente inativo.
8. O consumidor é ativado, continuando execução, pois considera que `contador=0`.
9. O consumidor se coloca como inativo através de um *sleep*.

```

#define FALSE 0
#define TRUE 1
#define N      100

int contador = 0;

void produtor(void) {
    int item;
    while (TRUE) {
        produzir_item(&item);
        if (contador == N) sleep();
        colocar_item(item);
        contador++;
        if (contador == 1) wakeup(consumidor);
    }
}

void consumidor (void) {
    int item;
    while (TRUE) {
        if (contador == 0) sleep();
        remover_item(&item);
        contador--;
        consumir_item(item);
        if (contador == N-1) wakeup(produtor);
    }
}

```

Exemplo 2.9 Solução parcial do problema produtor-consumidor

10. O produtor continuará produzindo.
11. O *buffer* ficará cheio pois consumidor está inativo, fazendo que o produtor se coloque como inativo com outro *sleep*.
12. Ambos os processos permanecerão para sempre inativos.

A perda de um sinal *wakeup* acontece devido ao uso irrestrito da variável *contador*. A primeira solução encontrada para este problema foi adicionar uma *flag* que sinalizasse a situação de envio de um sinal *wakeup* para um processo ativo. Tal *flag* se denomina (*wakeup waiting flag*). Em qualquer tentativa de adormecer, o processo deve antes verificar esta *flag*, ocorrendo o seguinte:

- se *flag*=0, o processo adormece; e

- se `flag=1`, o processo permanece ativo e faz `flag=0`.

Esta improvisada solução resolve o problema as custas de uma *flag* para processo envolvido no problema, mas é necessário destacar que o problema continua a existir, apenas de maneira mais sutil.

2.12.2 Semáforos

Para resolver o problema produtor-consumidor, Dijkstra propôs também em 1965 a utilização de variáveis inteiras para controlar o número de sinais *wakeup* para uso futuro [DEI92, p. 89]. Estas variáveis foram denominadas **semáforos**, e sobre elas estabeleceu-se duas diferentes operações: P (conhecida também como **Down**) e V (conhecida também como **Up**), que são generalizações das operações *sleep* e *wakeup* e funcionam como mostra a Tabela 2.5.

Tabela 2.5: Operações P() e V() sobre semáforos

Operação P(X) ou Down(X)
Verifica se o valor do semáforo X é positivo ($X > 0$).
Caso afirmativo decrementa X de uma unidade (ou seja, consome um sinal de <i>wakeup</i>).
Caso negativo envia sinal de sleep , fazendo inativo o processo.
Operação V(X) ou Up(X)
Incrementa o valor do semáforo X de uma unidade.
Existindo processos inativos, na ocorrência uma operação down , um deles é escolhido aleatoriamente pelo sistema para ser ativado (semáforo retornará a zero, i.e., $X=0$).

A implementação de semáforos em linguagem C poderia ser realizada como esquematizada no Exemplo 2.10.

Ambas as operações devem ser realizadas como transações atômicas, isto é, de forma indivisível, de forma que enquanto uma operação esteja em andamento com um dado semáforo, nenhuma outra seja efetuada, garantindo-se a consistência dos valores das variáveis semáforo.

A utilização de semáforos permite a sincronização de vários processos, ou seja, num ambiente onde existem diversos processos paralelos competindo por recursos, o uso de semáforos garante que um dado recurso seja utilizado de forma seqüencial, ou seja, de forma exclusiva.

Os semáforos e as operações sobre eles são usualmente implementadas como chamadas do sistema operacional, e representam uma solução para

```

typedef int semaforo;
void P(semaforo *s) {
    if(*s > 0) {
        s = *s - 1;
    } else {
        sleep();
    }
}

void V(semaforo *s) {
    if (ExisteProcessoEsperando(s)) {
        AtivaProcessoEsperando(s);
    } else {
        s = *s + 1;
    }
}

```

Exemplo 2.10 Implementação de operações P() e V() sobre semáforos

o problema de perda de sinais de wakeup visto no problema produtor-consumidor. Sendo regiões críticas, sua implementação utiliza instruções tipo TST e, embora provocando um espera ativa, são operações extremamente rápidas, muito mais eficientes que o uso de outras soluções que utilizem tais instruções ou as soluções de Dekker ou Peterson.

Semáforos iniciados com valor 1 são conhecidos como semáforos binários [TAN92, p. 42], que permitem que apenas um dentre n processos utilize um dado recurso, ou seja, garantem o uso individual deste recurso através da exclusão mútua.

Os semáforos são freqüentemente utilizados para sincronização de processos, ou seja, são utilizados para garantir a ocorrência de certas seqüências de eventos ou para impedir que outras seqüências nunca ocorram ou para que ocorram de uma forma específica.

No Exemplo 2.11 temos uma possível solução para o problema do produtor consumidor utilizando semáforos. Consideramos que as operações P() e V() foram implementadas como ilustrado no Exemplo 2.10 e incluídas no programa através do cabeçalho declarado.

2.12.3 Memória compartilhada

A **memória compartilhada** é um mecanismo freqüentemente utilizado para a comunicação entre processos diferentes onde uma região de memória é reservada para uso comum dos processos envolvidos na comunicação.

A área de memória reservada para os processo é semelhante a um *buffer*, mas nesta situação todos os processos envolvidos podem escrever e ler neste

```
#include "semforos.h"

#define FALSE 0
#define TRUE 1
#define N     100

typedef int semaforo;
semaforo mutex = 1;
semaforo vazio = N;
semaforo cheio = 0;

void produtor(void) {
    int item;
    while (TRUE) {
        produzir_item(&item);
        p(&vazio);
        p(&mutex);
        colocar_item(item);
        v(&mutex);
        v(&cheio);
    }
}

void consumidor(void) {
    int item;
    while (TRUE) {
        p(&cheio);
        p(&mutex);
        remover_item(&item);
        v(&mutex);
        v(&vazio);
        consumir_item(item);
    }
}
```

Exemplo 2.11 Solução do problema produtor-consumidor com semáforos

buffer. Como dois processos não podem acessar simultaneamente esta região de memória, embora compartilhada seu uso deve ser serializado, ou seja, um processo de cada vez deve utilizar a região de memória destinada a comunicação entre eles. Para obter-se isso, associa-se um semáforo com valor inicial 1 à região de memória. Todo os processos, antes de usarem a área de memória devem acionar uma operação $P()$. Ao finalizar o uso deverão acionar uma operação $V()$, garantindo seu uso exclusivo.

De forma geral, a solução é semelhante aquela do problema de produtores e consumidores, aplicada a n processos que podem tanto ler ou escrever numa região de memória específica, sendo frequentemente utilizada para passagem de mensagens entre processos.

2.12.4 Outros mecanismos de IPC

Contadores de eventos

Reed e Kanodia propuseram em 1979 uma outra solução para o problema de produtor-consumidor sem a necessidade de obter-se a exclusão mútua, tal como na solução que utiliza semáforos. Esta solução utiliza uma variável especial denominada **contador de eventos** (*event counters*) que possui três operações definidas como mostra a Tabela 2.6

Tabela 2.6: Operações sobre contadores de eventos

Operação	Descrição
<code>read(E)</code>	Retorna o valor corrente de E .
<code>advance(E)</code>	Incrementa, atômica, o valor de E de uma unidade.
<code>await(E, v)</code>	Espera que E tenha valor igual ou superior a v .

É possível solucionar-se o problema utilizando os contadores de eventos e suas operações, de forma semelhante ao uso de semáforos.

Monitores

Tanto os semáforos como os contadores de eventos podem resolver uma série de problemas, mas seu uso deve ser cuidadoso para que não provoque situações desastrosas. A inversão de dois semáforos, (por exemplo, `mutex` e `vazio` na solução do problema produtor-consumidor usando semáforos) pode provocar um bloqueio perpétuo, ou seja, faz com que uma dada tarefa pare de ser executada, degradando o sistema e podendo causar até mesmo sua instabilidade.

Para que tais problemas pudessem ser resolvidos mais facilmente Hoare (1974) e Hansem (1975) propuseram o conceito de **monitor**: uma coleção de procedimentos, variáveis e estruturas agrupados num módulo ou pacote especial. Segundo Guimarães:

Monitor é um conjunto de procedimentos que operam sobre variáveis comuns a vários processos. Um procedimento do monitor corresponde a uma região crítica. Um monitor corresponde, portanto, a um conjunto de regiões críticas operando sobre as mesmas variáveis comuns. [GUI86, p. 88]

Processos podem acessar os procedimentos e funções de um monitor embora não possam utilizar diretamente a estrutura interna de seus dados [TAN92, p. 45], num arranjo muito semelhante a utilização da interface de um objeto, sem acesso aos seus campos privados. No Exemplo 2.12, temos a declaração de um monitor numa notação semelhante ao Pascal.

```
Monitor Exemplo;
  { declaração de variáveis }
  emUso : boolean;
  livre : condition;

  procedure AlocaRecurso;
  begin
    if emUso then wait(livre);
    emUso := true;
  end;

  procedure LiberaRecurso;
  begin
    emUso := false;
    signal(livre);
  end;

  begin
    emUso := false; { inicialização }
  end;
end monitor.
```

Exemplo 2.12 Exemplo de monitor

Esta solução se baseia na introdução de variáveis de condição e de duas operações especiais sobre elas: `AlocaRecurso` e `LiberaRecurso`. Quando um monitor descobre que uma operação não é possível, ele efetua uma operação *wait* sobre uma certa variável de condição, provocando o bloqueio do processo que utilizou o monitor. Isto permite que outro processo, anteriormente bloqueado utilize o recurso. Este processo, ou ainda outro, pode efetuar uma operação de *signal* sobre aquela variável de condição, saindo imediatamente do monitor. Um dos processos em espera pela condição, ao ser ativado pelo escalonador poderá então prosseguir com o trabalho.

Apesar de serem semelhantes as operações de *sleep* e *wakeup*, vistas na seção 2.12.1, através dos monitores os problemas ocorridos em situações de corrida não se repetem dado a garantia de exclusão mútua sobre as operações nas variáveis de condição internas dos próprios monitores. Além disso, as operações *wait* e *signal* só podem ser utilizadas internamente aos procedimentos e funções dos monitores. Os monitores são um conceito de programação. Enquanto que os semáforos podem ser implementados diretamente em C ou Pascal, com uso de algumas rotinas *assembly*, a implementação de monitores é um pouco mais complexa, exigindo do compilador primitivas de exclusão mútua.

2.13 Threads

Como vimos, cada processo conta com uma estrutura de controle razoavelmente sofisticada. Nos casos onde se deseja realizar duas ou mais tarefas *simultaneamente*, a solução trivial a disposição do programador é dividir as tarefas a serem realizadas em dois ou mais processos. Isto implica na criação de manutenção de duas estruturas de controle distintas para tais processos, onerando o sistema, e complicando também o compartilhamento de recursos, dados serem processos distintos.

Uma alternativa ao uso de processos *comuns* é o emprego de *threads*. Enquanto cada processo tem um único fluxo de execução, ou seja, só recebe a atenção do processador de forma individual, quando um processo é dividido em *threads*, cada uma das *threads* componentes recebe a atenção do processador como um processo comum. No entanto, só existe uma estrutura de controle de processo para tal grupo, o espaço de memória é o mesmo e todos os recursos associados ao processo podem ser compartilhados de maneira bastante mais simples entre as suas *threads* componentes.

Segundo Tanebaum, "as threads foram inventadas para permitir a combinação de paralelismo com execução seqüencial e chamadas de sistema bloqueantes" [TAN92, p. 509]. Na Figura 2.19 representamos os fluxos de execução de um processo comum e de outro, dividido em *threads*.

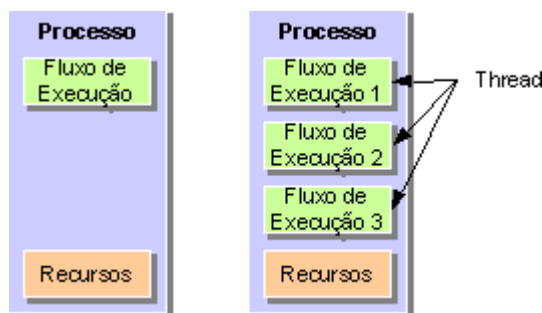


Figura 2.19: Processos e *threads*

Desta forma, as *threads* podem ser entendidas como fluxos independentes de execução pertencentes a um mesmo processo, que requerem menos recursos de controle por parte do sistema operacional. Assim, as *threads* são o que consideramos processos leves (*lightweight processes*) e constituem uma unidade básica de utilização do processador [TAN92, p. 508] [SGG01, p. 82].

Sistemas computacionais que oferecem suporte para as *threads* são usualmente conhecidos como sistemas *multithreading*. Como ilustrado na Figura 2.20, os sistemas *multithreading* podem suportar as *threads* segundo dois modelos diferentes:

User threads As *threads* de usuário são aquelas oferecidas através de bibliotecas específicas e adicionais ao sistema operacional, ou seja, são implementadas acima do *kernel* (núcleo do sistema) utilizando um modelo de controle que pode ser distinto do sistema operacional, pois não são nativas neste sistema.

Kernel threads As *threads* do sistema são aquelas suportadas diretamente pelo sistema operacional e, portanto, nativas.

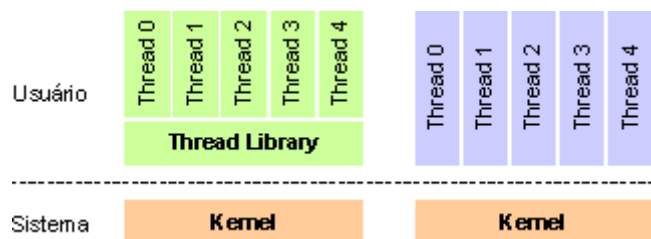


Figura 2.20: *Threads* de usuário e de *kernel*

Em sistemas não dotados de suporte a *threads* nativamente, o uso de bibliotecas de extensão permite a utilização de *pseudo-threads*. Exemplos de bibliotecas de suporte *threads* de usuário são o **PThreads** do POSIX ou o **C-threads** do sistema operacional Mach.

Através de tais biblioteca são oferecidos todos os recursos necessários para a criação e controle das *threads*. Usualmente os mecanismos de criação de *threads* de usuário são bastante rápidos e simples, mas existe uma desvantagem: quando uma *thread* é bloqueada (por exemplo, devido ao uso de recursos de I/O), as demais *threads* freqüentemente também são devido ao suporte não nativo. Quando o suporte é nativo, a criação das *threads* é usualmente mais demorada, mas não ocorrem os inconveniente decorrentes do bloqueio de uma ou mais *threads* em relação às demais.

2.13.1 Modelos de *multithreading*

A forma com que as *threads* são disponibilizadas para os usuários é o que denominamos modelos de *multithreading*. Como mostra a Figura 2.21, são comuns três modelos distintos:

- Modelo n para um .
Este modelo é empregado geralmente pelas bibliotecas de suporte de *threads* de usuário, onde as várias *threads* do usuário (n) são associadas a um único processo suportado diretamente pelo sistema operacional.
- Modelo um para um .
Modelo simplificado de *multithreading* verdadeiro, onde cada *threads* do usuário é associada a uma *thread* nativa do sistema. Este modelo é empregado em sistemas operacionais tais como o MS-Windows NT/2000 e no IBM OS/2.
- Modelo n para m .
Modelo mais sofisticado de *multithreading* verdadeiro, onde um conjunto de *threads* do usuário n é associado a um conjunto de *threads* nativas do sistema, não necessariamente do mesmo tamanho (m). Este modelo é empregado em sistemas operacionais tais como o Sun Solaris, Irix e Digital UNIX.

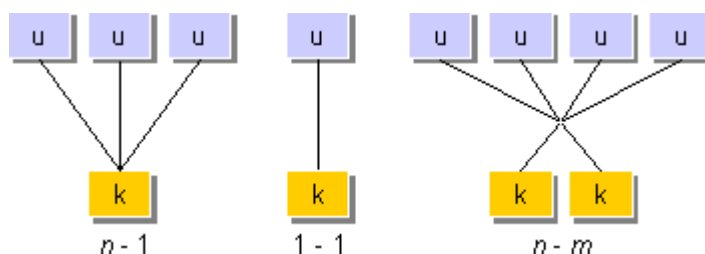


Figura 2.21: Modelos de *multithreading*

2.13.2 Benefícios do uso

A utilização das *threads* pode trazer diversos benefícios para os programas e para o sistema computacional em si [SGG01, p. 83]:

- Melhor capacidade de resposta, pois a criação de uma nova *thread* é substancialmente mais rápida do a criação de um novo processo.
- Compartilhamento de recursos simplificado entre as *threads* de um mesmo processo, que é a situação mais comum de compartilhamento e comunicação inter-processos.

- Economia, pois o uso de estruturas de controle reduzidas em comparação ao controle típico dos processos, desoneramos o sistema. Além disso o compartilhamento de recursos simplificado leva também a economia de outros recursos.
- Permitem explorar mais adequadamente as arquiteturas computacionais que dispõem de múltiplos processadores.

Capítulo 3

Escalonamento de Processos

O **escalonamento de processadores** é a forma com os processadores existentes num sistema computacional são utilizados para efetuar o processamento, isto é, é como os processos são distribuídos para execução nos processadores.

Tanenbaum propõe a seguinte definição:

Quando mais de um processo é executável, o sistema operacional deve decidir qual será executado primeiro. A parte do sistema operacional dedicada a esta decisão é chamada escalonador (scheduler) e o algoritmo utilizado é chamado algoritmo de escalonamento (scheduling algorithm). [TAN92, p. 62]

Por sua vez, Deitel coloca que:

A designação de processadores físicos para processos permite aos processos a realização de trabalho. Esta designação é uma tarefa complexa realizada pelo sistema operacional. Isto é chamado escalonamento do processador (processor scheduling). [DEI92, p. 287]

Simplificando, podemos afirmar que num sistema onde só exista um único processador, o escalonamento representa a ordem em que os processos serão executados.

A forma com que se dá o escalonamento é, em grande parte, responsável pela produtividade e eficiência atingidas por um sistema computacional. Mais do que um simples mecanismo, o escalonamento deve representar uma política de tratamento dos processos que permita obter os melhores resultados possíveis num sistema.

3.1 Objetivos do escalonamento

O projeto de um escalonador adequado deve levar em conta uma série de diferentes necessidades, ou seja, o projeto de uma política de escalonamento deve contemplar os seguintes objetivos:

- Ser justo: todos os processos devem ser tratados igualmente, tendo possibilidades idênticas de uso do processador, devendo ser evitado o adiamento indefinido.
- Maximizar a produtividade (*throughput*): procurar maximizar o número de tarefas processadas por unidade de tempo.
- Ser previsível: uma tarefa deveria ser sempre executada com aproximadamente o mesmo tempo e custo computacional.
- Minimizar o tempo de resposta para usuários interativos.
- Maximizar o número possível de usuários interativos.
- Minimizar a sobrecarga (*overhead*): recursos não devem ser desperdiçados embora algum investimento em termos de recursos para o sistema pode permitir maior eficiência.
- Favorecer processos *bem comportados*: processos que tenham comportamento adequado poderiam receber um serviço melhor.
- Balancear o uso de recursos: o escalonador deve manter todos os recursos ocupados, ou seja, processos que usam recursos sub-utilizados deveriam ser favorecidos.
- Exibir degradação previsível e progressiva em situações de intensa carga de trabalho.

Como pode ser visto facilmente, alguns destes objetivos são contraditórios, pois dado que a quantidade de tempo disponível de processamento (tempo do processador) é finita, assim como os demais recursos computacionais, para que um processo seja favorecido outro deve ser prejudicado.

O maior problema existente no projeto de algoritmos de escalonamento está associado a natureza imprevisível dos processos, pois não é possível prevermos se um dado processo utilizará intensamente o processador, ou se precisará grandes quantidades de memória ou se necessitará numerosos acessos aos dispositivos de E/S.

3.2 Níveis de escalonamento

Existem três níveis distintos de escalonamento em um sistema computacional quando se considera a frequência e complexidade das operações envolvidas.

- Escalonamento de **alto nível**
Chamado também de escalonamento de tarefas, corresponde a admissão de processos, isto é, a determinação de quais tarefas passarão a competir pelos recursos do sistema. Uma vez admitidas, as tarefas transformam-se em processos. Correspondem a rotinas de alto nível oferecidas pelas APIs do sistema operacional.
- Escalonamento de **nível intermediário**
Corresponde a determinação de quais processos existentes competirão pelo uso do processador (processos ativos). Este nível de escalonamento é responsável por administrar a carga do sistema, utilizando-se de primitivas de suspensão (*suspend*) e ativação (*resume* ou *activate*). Correspondem a rotinas internas do sistema operacional.
- Escalonamento de **baixo nível**
Rotinas que determinam qual processos, dentre os processos ativos, será o próximo processo que efetivamente utilizará o processador. Estas tarefas são executadas pelo *dispatcher*, usualmente uma rotina escrita diretamente em linguagem de máquina que se encontra permanentemente na memória principal.

Os níveis de escalonamento **alto**, **intermediário** e **baixo** também são conhecidos respectivamente como escalonamento de **longo prazo**, **médio prazo** e **curto prazo**. O escalonamento de alto nível ou de longo prazo ocorre menos frequentemente num sistema enquanto o escalonamento de baixo nível ou de curto prazo ocorre constantemente, dado que representa a troca de contexto e o chaveamento do processador entre os processos ativos.

Considerando assim os níveis de escalonamento e as operações de suspensão (*suspend* ou *sleep*) e ativação (*resume*, *wakeup* ou *activate*), o mapa de estados dos processos pode ser representado de maneira mais completa como ilustrado na Figura 3.2

3.3 Escalonamento preemptivo e não preemptivo

Um algoritmo de escalonamento é dito **não preemptivo** quando temos que o processador designado para um certo processo não pode ser retirado deste até que o processo seja finalizado (**completion**). Analogamente, um algoritmo de escalonamento é considerado **preemptivo** quando o processador designado para um processo pode ser retirado deste em favor de um outro processo.

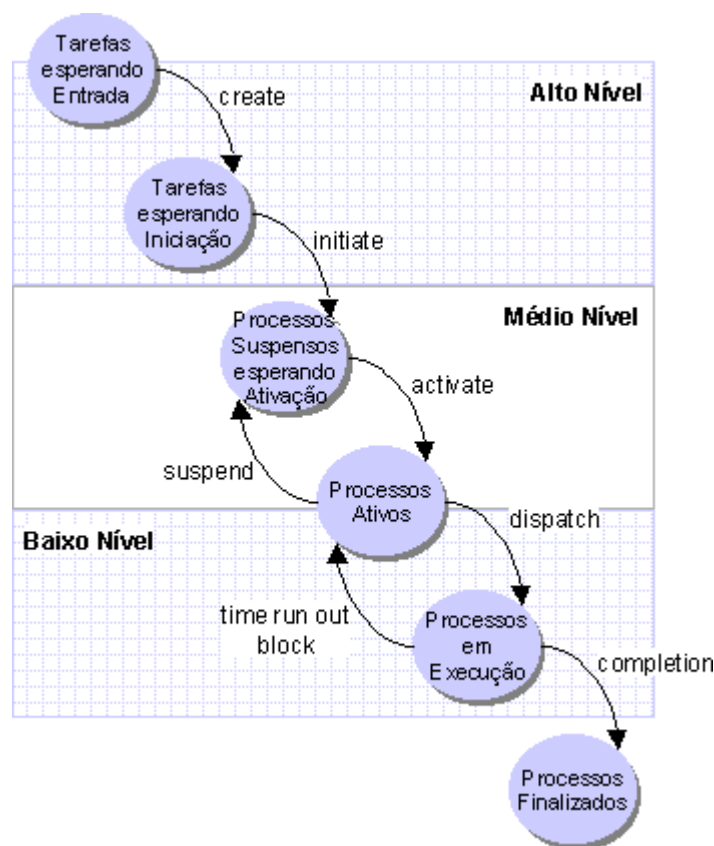


Figura 3.1: Níveis de escalonamento

Algoritmos preemptivos são mais adequados para sistemas em que múltiplos processos requerem atenção do sistema, ou seja, no caso de sistemas multiusuário interativos (sistemas em tempo repartido) ou em sistema de tempo real. Nestes casos, a preemptividade representa a troca do processo em execução, assim sendo, para que o processador seja retirado de um processo, interrompendo seu trabalho, e designado a outro processo, anteriormente interrompido, é fundamental que ocorra a troca de contexto dos processos. Tal troca exige que todo o estado de execução de um processo seja adequadamente armazenado para sua posterior recuperação, representando uma sobrecarga computacional para realização desta troca e armazenagem de tais dados. Usualmente os algoritmos preemptivos são mais complexos dada a natureza imprevisível dos processos.

Por sua vez, os algoritmos não preemptivos são mais simples e adequados para o processamento não interativo, semelhante aos esquemas de processamento em lote dos sistemas *batch*. Embora não proporcionando interatividade, são geralmente mais eficientes e previsíveis quanto ao tempo de entrega de suas tarefas.

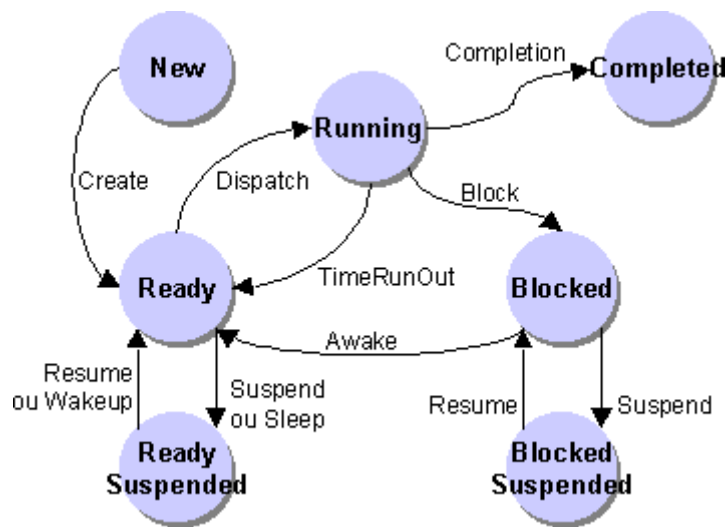


Figura 3.2: Mapa de estados dos processos

Existem também algoritmos de escalonamento cooperativo, onde os processos não são interrompidos, mas a preempção ocorre em duas situações bem definidas: quando o processo efetua uma operação de I/O e quando o processo é finalizado [SGG01, p. 97]. Também é possível que um processo ceda o processador, voluntariamente, em favor de outros processos. Neste caso, o processo *educado* poderia executar uma chamada a uma função *yield* (dar preferência), como no caso do MS-Windows 3.1. A preempção voluntária pode auxiliar numa distribuição mais equitativa da capacidade de processamento do sistema, mas conta com a *generosidade* do programador, nem sempre disponível.

A preemptividade de certos algoritmos se baseia no fato de que o processador é, naturalmente, um recurso preemptivo, ou seja, um recurso que pode ser retirado de um processo e posteriormente devolvido sem prejuízo. O mesmo acontece com a memória. Por outro lado, outros tipos de recursos não podem sofrer preempção, tais como impressoras e até mesmo arquivos, dado que muitas vezes não podem ser retirados de um processo sem que ocorra prejuízo para este.

3.4 Qualidade do escalonamento

Existem vários critérios que permitem a avaliação da qualidade do serviço oferecido por um algoritmo de escalonamento [SGG01, p. 98]: uso do processador, *throughput*, tempo de resposta e tempo de permanência.

O **tempo de permanência**, **tempo de retorno** ou *turnaround time*, é um critério simples dado pela soma do **tempo de espera** com o **tempo de serviço** ou **tempo de execução**:

$$t_p = t_{\text{permanencia}} = t_{\text{espera}} + t_{\text{servico}} \quad (3.1)$$

Em geral deseja-se que o tempo de permanência seja o menor possível. Na Figura 3.3 temos uma representação gráfica do tempo de permanência.

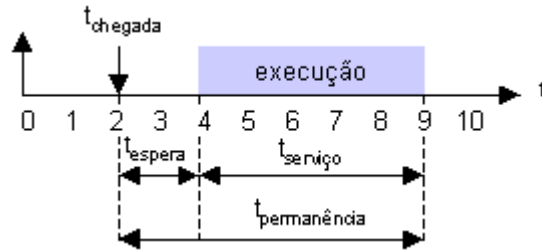


Figura 3.3: Representação gráfica do tempo de permanência

Uma outra forma de avaliar-se o escalonamento é utilizando-se o **tempo de permanência normalizado** (t_{pn}), ou seja, a razão entre o tempo de permanência (t_p) e o tempo de serviço (t_s):

$$t_{pn} = \frac{t_{\text{permanencia}}}{t_{\text{servico}}} = \frac{t_{\text{espera}} + t_{\text{servico}}}{t_{\text{servico}}} \quad (3.2)$$

Se a espera for zero, o que constitui o melhor caso possível, teremos que o tempo de permanência normalizado de um processo será 1 (um). Assim sendo, valores maiores do que este indicam um pior serviço oferecido pelo algoritmo de escalonamento.

3.5 Algoritmos de escalonamento

Existem vários algoritmos que são utilizados para a realização do escalonamento de baixo nível ou de curto prazo. Em todos eles, o principal objetivo é designar o processador para um certo processo dentre vários processos existentes, otimizando um ou mais aspectos do comportamento geral do sistema. Stallings categoriza os escalonadores como [STA92, p. 356]:

1. **Orientados ao usuário:** quando procuram otimizar os tempos de resposta e permanência além da previsibilidade.
2. **Orientados ao sistema:** quando enfatizam a produtividade, a taxa de utilização da processador, o tratamento justo e o balanceamento do uso de recursos.

Serão abordados os seguintes algoritmos de escalonamento:

- *First In First Out*

- *Highest Priority First*
- *Shortest Job First*
- *Highest Response-Ratio Next*
- *Shortest Remaining Time*
- *Round Robin*
- *Multilevel Queues*
- *Multilevel Feedback Queues*

3.5.1 Escalonamento FIFO (*First In First Out*)

É a forma mais simples de escalonamento também conhecido como FCFS (*First Come First Served*), ou *primeiro a chegar, primeiro a ser servido*. No escalonamento FIFO os processos prontos (ou *jobs*) são colocados numa fila organizada por ordem de chegada, o que corresponde dizer que sua **função de seleção** é $\min(t_{chegada})$, como mostra a Figura 3.4.

Com esta função de seleção, seleciona-se dentre os processos na fila de espera aquele com menor tempo de chegada. Tal processo recebe o uso do processador até que seja completado (*completion*), ou seja, o processo permanece em execução até que seja finalizado, de forma que os demais processos na fila fiquem esperando por sua oportunidade de processamento. Assim sendo, o escalonamento FIFO é um algoritmo não preemptivo, pois os processos em execução não são interrompidos.

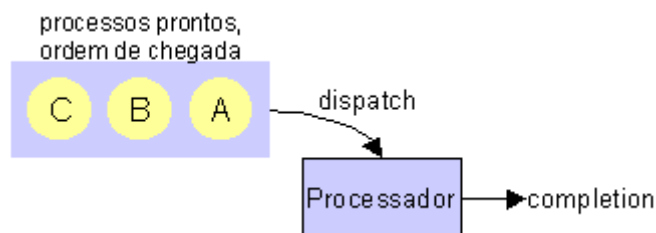


Figura 3.4: Escalonamento FIFO (*First In First Out*)

Embora dê igual tratamento a todos os processos, ocorre que processos de pequena duração não são favorecidos pois tem seu tempo de resposta fortemente influenciado pelos processos a serem processados primeiramente, ou seja, o tempo de resposta aumenta consideravelmente em função da quantidade de processos posicionados a frente e também pela duração destes.

Outro ponto é que processos importantes podem ficar a espera devido à execução de outros processos menos importantes dado que o escalonamento

FIFO não concebe qualquer mecanismo de distinção entre processos (por exemplo, processos com diferentes níveis de prioridade).

Avaliemos o exemplo ilustrado pela Tabela 3.1, onde quatro processos A, B, C e D com tempos de processamento distintos, respectivamente 3, 35, 12 e 4 segundos, são escalonados conforme sua chegada pelo sistema operacional.

Tabela 3.1: Exemplo de fila de processos sob escalonamento FIFO

Processo	$t_{chegada}$	$t_{servico}$	t_{espera}	t_{perm}	t_{pn}
A	0	3	0	3	1.00
B	1	35	2	37	1.06
C	2	12	36	48	4.00
D	3	4	47	51	12.75
		Médias	21.25	34.75	4.7

Podemos perceber que, em função de como ocorreu o escalonamento, o menor processo obteve o pior serviço, num esquema de escalonamento que tende a ter tempos médios de resposta maiores quanto maior o número de processos em espera. Esta forma de escalonamento, semelhante aos sistemas de processamento em lote (*batch systems*) não é utilizada como esquema principal de escalonamento, mas como forma auxiliar de escalonamento para processamento de filas *batch*.

3.5.2 Escalonamento HPF (*Highest Priority First*)

O escalonamento HPF (*Highest Priority First*) ou escalonamento por prioridades é uma variante do escalonamento FIFO onde os processos em espera pelo processador são organizados numa fila segundo sua prioridade, sendo colocados a frente os processos *jobs* de maior prioridade, isto é, sua **função de seleção** é $\max(\text{prioridade})$, favorecendo os processos considerados mais importantes [TAN92, p. 64] [SGG01, p. 105].

Após iniciados, os processos não são interrompidos, ou seja, é uma forma de escalonamento não preemptivo, oferecendo como a vantagem de proporcionar tempos médios de espera menores para processos prioritário. Na Figura 3.5 temos uma representação deste tipo de escalonamento.

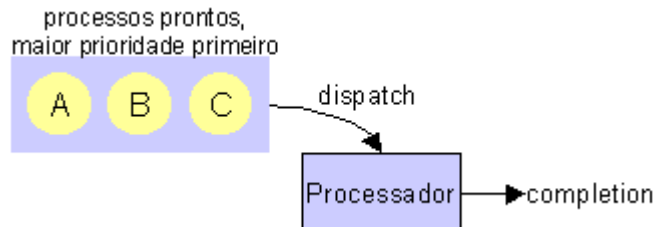


Figura 3.5: Escalonamento HPF (*Highest Priority First*)

A prioridade é, em geral, representada por um número inteiro, no entanto não existe consenso em definir prioridade maiores como números maiores (indicando sua maior importância) ou o inverso (indicando sua ordem preferencial). A prioridade pode ser definida interna ou externamente ao sistema. Quando determinada internamente, o sistema pode utilizar diversos fatores, tais como quantidade de memória necessária, estimativas do tempo de serviço e outros elementos para calcular a prioridade. Notemos que o tempo de processamento de um *job* não pode ser determinado antes de seu processamento, sendo necessário o uso de estimativas feitas pelo usuário ou pelo programador, as quais são com frequência pouco precisas. No caso de prioridade externamente, a mesma é simplesmente atribuída ao processo pelo operador ou pelo programador, de maneira empírica.

Avaliemos o mesmo exemplo utilizado para o escalonamento FIFO, onde quatro processos A, B, C e D com tempos de processamento distintos, respectivamente 3, 35, 12 e 4 segundos, são escalonados conforme suas prioridades, definidas externamente, como mostra a Tabela 3.2.

Tabela 3.2: Exemplo de fila de processos sob escalonamento HPF

Processo	$t_{chegada}$	prio	$t_{servico}$	t_{espera}	t_{perm}	t_{pn}
A	0	4	3	0	3	1.00
C	2	1	12	1	13	1.08
B	1	2	35	14	49	1.4
D	3	3	4	47	51	12.75
			Médias	15.5	29.00	4.06

Podemos perceber que, dado o escalonamento ordenar os processos segundo sua prioridade, alguns processos podem ser prejudicados em função de outros de maior prioridade, pois se processos grande são executados primeiro ocorre uma degradação significativa para os demais processos.

Observe também que o processo A, embora tenha prioridade mais baixa que os demais, começou a ser executado antes devido ao seu tempo de chegada, não sendo interrompido com a chegada de processos de maior prioridade, pois este algoritmo não é preemptivo. Por outro lado, devemos considerar também que um processo de prioridade relativamente baixa pode ter sua execução adiada indefinidamente pela chegada contínua de processos de maior prioridade (*indefinite postponement*), ou seja pode sofrer de estagnação (*starvation*).

No caso do HPF, existe uma solução, relativamente simples, para a solução do problema da estagnação denominada *aging* (envelhecimento). Tal solução consistem em progressivamente aumentar a prioridade dos processos que aguardam na fila a medida que o tempo passa, ou seja, como forma de compensar a espera e evitar a estagnação, processos poderiam ter sua prioridade incrementada até que sejam executados [SGG01, p. 103].

Conforme as prioridades e os processos, os resultados apresentados por este algoritmo poderiam tanto aproximá-lo do escalonamento FIFO como apresentar resultados de qualidade melhores. No caso, os resultados exibidos foram um pouco melhores. Na verdade sua maior qualidade está na execução seletiva de *jobs* conforme sua prioridade calculada ou atribuída.

Da mesma forma que no escalonamento FIFO, o escalonamento HPF não é utilizado como esquema principal de escalonamento, mas como forma auxiliar de escalonamento para processamento de filas batch.

3.5.3 Escalonamento SJF (*Shortest Job First*)

O escalonamento SJF (*Shortest Job First*) ou menor *job* primeiro, também é conhecido como SPF (*Shortest Process First*) ou menor processo primeiro. É um caso especial do HPF, onde o tempo de serviço é tomado como prioridade, ou seja, os processos em espera pelo processador são organizados numa fila segundo seu tempo de execução, sendo colocados a frente os menores processos *jobs*, isto é, sua **função de seleção** é $\min(t_{servico})$, favorecendo os processos que podem ser finalizados intervalos de tempo menores.

Após iniciados, os processos não são interrompidos, ou seja, é uma forma de escalonamento não preemptivo, mas mesmo assim oferece a vantagem de proporcionar tempos médios de espera menores do aqueles obtidos num esquema FIFO. Na Figura 3.6 temos uma representação deste tipo de escalonamento.

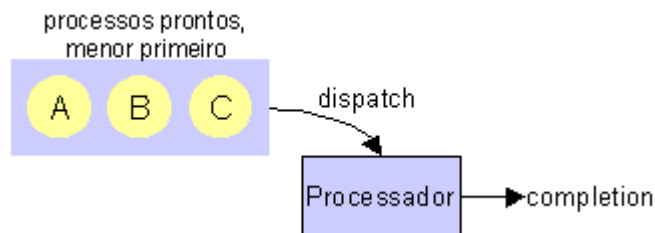


Figura 3.6: Escalonamento SJF (*Shortest Job First*)

O grande problema deste esquema de escalonamento é que o tempo de processamento de um *job* não pode ser determinado antes de seu processamento, sendo necessário o uso de estimativas feitas pelo usuário ou pelo programador, as quais são com freqüência pouco precisas. Para evitar abusos por parte dos usuários (que podem indicar estimativas mais baixas que os tempos esperados), podem ser adotadas políticas de premiação em função do acerto das estimativas, mas isto não resolve uma série de problemas, entre os quais o incorreto posicionamento de um *job* pelo sistema devido a uma estimativa incorreta, prejudicando outros *jobs*.

Uma outra questão é a possibilidade deste algoritmos provocar a estagnação (*starvation*) de processos grandes os quais podem permanecer in-

definidamente na fila de espera caso processos de menor duração cheguem continuamente ao sistema. Uma solução para este problema seria impor um tempo máximo de espera fixo ou proporcional ao tempo de serviço, a partir do qual tais processos seriam executados, para evitar a estagnação.

Avaliemos o mesmo exemplo utilizado para o escalonamento FIFO, onde quatro processos A, B, C e D com tempos de processamento distintos, respectivamente 3, 35, 12 e 4 segundos, são escalonados em conforme suas durações e tempo de chegada pelo sistema operacional, como mostra a Tabela 3.3.

Tabela 3.3: Exemplo de fila de processos sob escalonamento SJF

Processo	$t_{chegada}$	$t_{servico}$	t_{espera}	t_{perm}	t_{pn}
A	0	3	0	3	1.00
D	3	4	0	4	1.00
C	2	12	5	17	1.42
B	1	35	18	51	1.46
		Médias	5.75	18.75	1.22

Podemos perceber que neste caso, em função do escalonamento ordenar os processos segundo sua duração, os menores processos obtiveram o melhor serviço sem que isso resultasse numa degradação significativa para os processos maiores. Tanto o tempo médio de espera como os tempos médios de permanência e de permanência normalizado apresentam valores bastante inferiores aos obtidos com o escalonamento FIFO (Tabela 3.1) ou mesmo o HPF (Tabela 3.2), evidenciando as qualidades do escalonamento SJF.

Da mesma forma que os demais esquemas de escalonamento vistos, o escalonamento SJF não é utilizado como esquema principal de escalonamento, mas como forma auxiliar de escalonamento para processamento de filas batch.

3.5.4 Escalonamento HRN (*Highest Response-Ratio Next*)

Para corrigir algumas das deficiências do escalonamento SJF, Hansen (1971) propôs um balanceamento entre a duração do *job* e seu tempo de espera, de forma a compensar a espera excessiva de tarefas de maior duração. Para tanto idealizou uma forma dinâmica de organização da fila de processos através do cálculo de suas taxas de resposta (*response ratio*) ou prioridades como dado a seguir:

$$prioridade = \frac{t_{espera} + t_{servico}}{t_{servico}} \quad (3.3)$$

A **função de seleção** escolhe o *job* de maior a prioridade para fazer uso do processador, ou seja, $max(prioridade)$. Observe atentamente a relação que determina a prioridade, podemos notar que, na verdade, este valor corresponde ao tempo de permanência normalizado, tal como mostra

a Equação 3.2. Temos assim que, num primeiro momento, os *jobs* de curta duração são favorecidos, pois o tempo de permanência figura no numerador da fração enquanto apenas o tempo de serviço aparece no denominador. A medida que a espera vai se tornando maior, até mesmo os *jobs* de maior duração vão tendo suas prioridades aumentadas até que estas superem as prioridades de processos curtos recém-chegados, proporcionando um equilíbrio bastante positivo a medida que os processos vão sendo selecionados e executados.

Na Figura 3.7 temos uma ilustração deste esquema de escalonamento.

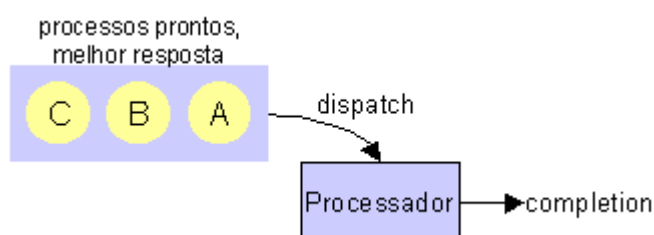


Figura 3.7: Escalonamento HRN (*Highest Response-Ratio Next*)

Uma vez encaminhados a CPU, os *jobs* são processados até sua finalização, sendo assim, este um algoritmo não preemptivo de escalonamento apesar da forma dinâmica com que a fila de processos em espera é administrada.

Da mesma forma que no escalonamento SJF e SRT, o escalonamento HRN pressupõe a disponibilidade dos tempos de execução de cada *job*, o que pode inviabilizar seu uso prático, tal como o de outros algoritmos de escalonamento baseados em estimativas do tempo de execução.

Novamente tomemos o exemplo utilizado para o escalonamento FIFO e SJF, onde quatro processos A, B, C e D com tempos de processamento distintos, respectivamente 3, 35, 12 e 4 segundos, são escalonados em conforme suas prioridades pelo sistema operacional. Note que as prioridades são reavaliadas ao final do término de cada *job*, determinando a próxima tarefa a ser processada, como mostra a Tabela 3.4.

Tabela 3.4: Exemplo de fila de processos sob escalonamento HRN

Processo	$t_{chegada}$	$t_{servico}$	t_{espera}	t_{perm}	t_{pn}
A	0	3	0	3	1.00
C	2	12	1	13	1.08
D	3	4	12	16	4.00
B	1	35	18	53	1.51
		Médias	7.75	21.25	1.90

Podemos observar que os resultados apresentados pelo HRN foram bas-

tante superiores ao FIFO (Tabela 3.1), mas um pouco inferiores ao algoritmo SJF (Tabela 3.3). Por outro lado, o HRN não apresenta o risco do adiamento infinito, sendo mais equilibrado para o processamento de *jobs* de tamanhos diversos.

3.5.5 Escalonamento SRT (*Shortest Remaining Time*)

O algoritmo de escalonamento SRT (*Shortest Remaining Time*) ou SRF (*Shortest Remaining First*) é a variante preemptiva do escalonamento SJF. A fila de processos a serem executados pelo SRT é organizada conforme o tempo estimado de execução, ou seja, de forma semelhante ao SJF, sendo processados primeiro os menores *jobs*. Na entrada de um novo processo, o algoritmo de escalonamento avalia seu tempo de execução incluindo o *job* em execução, caso a estimativa de seu tempo de execução seja menor que o do processo correntemente em execução, ocorre a substituição do processo em execução pelo recém chegado, de duração mais curta, ou seja, ocorre a preempção do processo em execução. Assim, a **função de seleção** do SRT corresponde à $\min(t_{servicorestante})$.

Cada processo suspenso pelo SRT deverá ser recolocado na fila em uma posição correspondente apenas ao seu tempo restante de execução e não mais o tempo de execução total, tornando-se necessário registrar os tempos decorridos de execução de cada processo suspenso. A Figura 3.8 esquematiza o funcionamento de algoritmo.

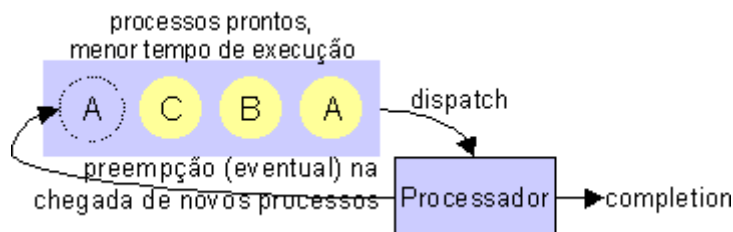


Figura 3.8: Escalonamento SRT (*Shortest Remaining Time*)

A sobrecarga imposta pelo registro dos tempos decorridos de execução e pela troca de contexto é justificada pelo fato de pequenos processos serem executados praticamente de imediato, permitindo oferecer tempos médios de espera baixos, tornando este algoritmo útil para sistemas em tempo repartido.

Por outro lado, este algoritmo também se baseia nas estimativas de tempo de execução dos processos, ou seja, possui as mesmas deficiências dos escalonamentos SJF e HRN quanto à precisão das estimativas e abusos por parte dos usuários. Sendo assim, devem também ser consideradas as seguintes questões:

1. *Jobs* de maior duração tem seus tempos de espera variáveis em função de *jobs* menores que venham a ser executados primeiramente.
2. Existe um risco potencial de processos grandes acabarem sendo adiados por um tempo indeterminado (*starvation*) devido ao excessivo favorecimento de *jobs* de curta duração.
3. Dado que a preempção poderia, teoricamente, ocorrer quando um processo esta prestes a ser finalizado, algum mecanismo extra deve ser adicionado para evitar que esta preempção inoportuna acabe impondo um serviço pior.

Teoricamente, o escalonamento SRT deveria oferecer um melhor situação de performance do que o escalonamento SJF, embora a sobrecarga existente possa equiparar os resultados obtidos em situações particulares.

Podemos aplicar o exemplo utilizado anteriormente para os algoritmos de escalonamento estudados ao SRT, como mostra a Tabela 3.5.

Tabela 3.5: Exemplo de fila de processos sob escalonamento SRT

Processo	$t_{chegada}$	$t_{servico}$	t_{espera}	t_{perm}	t_{pn}
A	0	3	0	3	1.00
D	3	4	0	4	1.00
C	2	12	5	17	1.42
B	1	35	18	51	1.46
		Médias	5.75	18.75	1.22

Dada a ordem de chegada particular deste caso, podemos perceber que os resultados obtidos pelo SRT são exatamente os mesmos do algoritmos SJF (Tabela 3.3), como teoricamente previsto. Mas devemos observar que para outros conjuntos de processos, os resultados deveriam ser ligeiramente melhores que o SJF.

3.5.6 Escalonamento RR (*Round-Robin*)

No escalonamento RR (*Round Robin*) ou circular os processos também são organizados numa fila segundo sua ordem de chegada, sendo então despachados para execução. No entanto, ao invés de serem executados até o fim (*completion*), a cada processo é concedido apenas um pequeno intervalo de tempo (*time slice* ou *quantum*). Caso o processo não seja finalizado neste intervalo de tempo, ocorre sua substituição pelo próximo processo na fila de processos ativos, sendo o processo em execução interrompido e novamente colocado na fila de processos prontos, mas em seu fim. Isto significa que ao final de seu intervalo de tempo, isto é, de seu *quantum*, ocorre a preempção do processador, ou seja, o processador é designado para outro processo, sendo

salvo o contexto do processo interrompido para permitir a continuidade da sua execução quando sua vez chegar novamente. Tal situação é ilustrada na Figura 3.9.

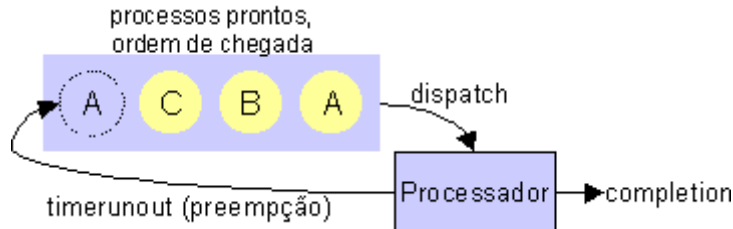


Figura 3.9: Escalonamento RR (*Round Robin*)

O escalonamento RR se baseia na utilização de temporizadores, constituindo um algoritmo preemptivo bastante adequado para ambiente interativos, ou seja, em sistemas em tempo repartido onde coexistem múltiplos usuários simultâneos sendo, portanto, necessário garantir-se tempos de resposta razoáveis. A sobrecarga (*overhead*) imposta pela troca de contexto representa um investimento para atingir-se um bom nível de eficiência, pois com diversos processos em execução simultânea (pseudoparalelismo) é possível manter ocupados todos os recursos do sistema.

A determinação do tamanho do intervalo de tempo (*quantum*) é extremamente importante, pois relaciona-se com a sobrecarga imposta ao sistema pelas trocas de contexto dos processos ativos. Na Figura 2.5, onde ilustramos o escalonamento de processos, podemos observar o *quantum* de processamento concedido para cada processo e os tempos de preservação de recuperação de contexto a cada preempção. Para cada processo despachado para execução ocorre:

1. a recuperação do contexto do processo, que toma um tempo que denominaremos (t_{rc}),
2. a execução do processo pela duração do *quantum* e
3. a preservação do processo após o término de seu *quantum*, a qual também toma um intervalo de tempo denotado por (t_{pc}).

Como o tempo tomado para a troca de contexto (t_{tc}) não é útil do ponto de vista de processamento de processos dos usuários, temos que para cada janela de tempo concedida aos processos a troca de contexto representa uma sobrecarga, pois somente o *quantum* de processamento é efetivamente útil.

Dado que a troca de contexto toma um tempo aproximadamente constante temos que a sobrecarga pode ser calculada através da relação a seguir:

$$t_{tc} = t_{rc} + t_{pc} \quad (3.4)$$

$$\text{sobrecarga} = \frac{t_{tc}}{t_{tc} + \text{quantum}} \quad (3.5)$$

Por exemplo, se o tempo para troca de contexto (t_{tc}) toma 2 ms e o quantum é de 8 ms, temos que apenas 80% do tempo de processamento é útil, ou seja, a sobrecarga imposta pela troca de contexto representa 20% do processamento.

Podemos também medir o rendimento proporcionado pelo escalonamento RR considerando quanto do tempo alocado para cada processo é efetivamente usado para o processamento, ou seja, a relação entre o quantum (usado para o processamento) e a soma deste com o tempo para troca de contexto (tomada para cada processo), como na relação que segue:

$$\text{rendimento} = \frac{\text{quantum}}{\text{quantum} + t_{tc}} = 1 - \text{sobrecarga} = 1 - \frac{t_{tc}}{t_{tc} + \text{quantum}} \quad (3.6)$$

Ao aumentarmos o quantum diminuimos a sobrecarga percentual da troca de contexto, mas um número menor de usuários (nu) será necessário para que os tempos de resposta (t_r) se tornem maiores e perceptíveis. Diminuindo o quantum temos uma situação inversa de maior sobrecarga e também de um maior número possível de usuários sem degradação sensível dos tempos de resposta.

$$t_r = nu * (\text{quantum} + t_{tc}) \quad (3.7)$$

ou

$$nu = \frac{t_r}{\text{quantum} + t_{tc}} \quad (3.8)$$

Usualmente o tamanho do quantum utilizado é tipicamente algo em torno de 20ms. Com ou aumento da velocidade dos processadores, a troca de contexto se dá mais rapidamente, diminuindo a sobrecarga e aumentando ligeiramente a quantidade de usuários possíveis para um mesmo limite de tempo de resposta.

Na Tabela 3.6 temos um exemplo hipotético do comportamento possível do rendimento e do número de usuários em função da variação do quantum usado pelo sistema. Nestes cálculos consideramos um tempo de resposta fixo de $t_r = 1s$ e também um tempo troca de contexto constante $t_{tc} = 2ms$.

Na Figura 3.10 podemos ver o comportamento do rendimento e número de usuários do sistema em função do quantum para um tempo de resposta fixo, conforme a Tabela 3.6

Como antes, podemos verificar o comportamento do algoritmo de escalonamento RR para uma seqüência de quatro processos A, B, C e D, com os mesmos tempos de chegada e serviço. Tomaremos como quantum um valor de 100 ms. Assim teremos os resultados apontados pela Tabela 3.7.

Tabela 3.6: Rendimento e número de usuários em função do *quantum*

quantum (ms)	número de usuários	rend. (%)
1	333	33.3
2	250	50.0
5	143	71.4
10	83	83.3
20	45	90.9
50	19	96.2
100	10	98.0
200	5	99,0
500	2	99.6
1000	1	99.8

Tabela 3.7: Exemplo de fila de processos sob escalonamento RR

Processo	$t_{chegada}$	$t_{servico}$	t_{espera}	t_{perm}	t_{pn}
A	0	3	4.6	7.6	2.53
B	1	35	18.0	53.0	1.51
C	2	12	17.5	29.5	2.46
D	3	4	9.3	13.3	3.32
		Médias	12.35	25.85	2.46

Não surpreendentemente, o algoritmo RR apresenta resultados bastante melhores que o escalonamento FIFO e pouco inferiores aos algoritmos SJF, HRN e SRT. Por outro lado, apresenta um grau de interatividade muito superior a todos os algoritmos citados, compensando largamente seu emprego.

3.5.7 Escalonamento MQ (*Multilevel Queues*)

Quando é possível dividir os processos em diferentes categorias, conforme seu tipo, prioridade e consumo de recursos, pode-se empregar o escalonamento em múltiplas filas [TAN92, p. 64] [SGG01, p. 105]. Deste modo, a fila de processos prontos seria separada em várias filas, uma para cada tipo de processo, tais como processos do sistema, processos interativos, processos de edição interativa, processos *batch* e processos secundários, como ilustrado na Figura 3.11, onde a fila de processos do sistema teria a maior prioridade enquanto a fila de processos secundários teria a menor prioridade.

Cada fila pode possuir seu próprio algoritmo de escalonamento, ou seja, algumas filas pode ser preemptivas enquanto outras não, sendo que os critérios de seleção dos processos para execução e da preempção, quando possível, podem ser distintos de uma fila para outra. É um arranjo comum que a fila de processos do sistema e *batch* usem um algoritmo tal como o FIFO

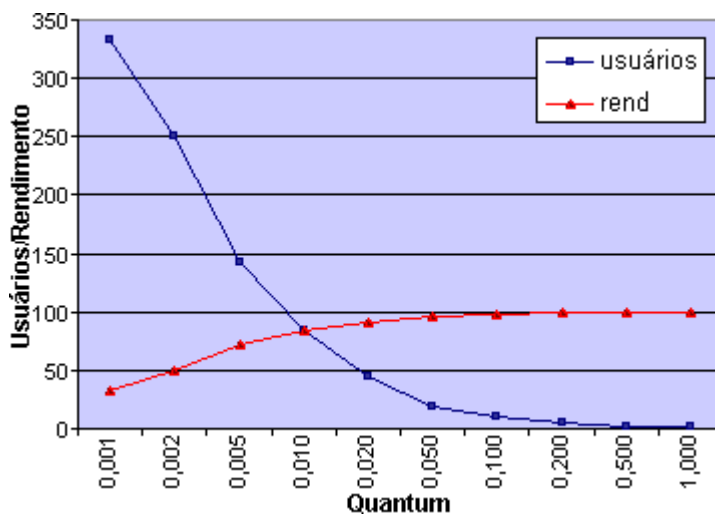


Figura 3.10: Comportamento do rendimento e número de usuários em função do *quantum*

enquanto que as demais utilizem algoritmos semelhantes ao *round robin*.

Entre as diferentes filas a divisão do processamento também pode ocorrer de diversas maneiras. É possível realizar tal divisão considerando um algoritmo de prioridade simples, onde a fila de maior prioridade tem seus processos executados até que esteja vazia. Isto pode comprometer o nível de resposta dos processos nas demais filas, possibilitando até mesmo a estagnação de processos nas filas de prioridades inferiores. Outra solução seria o emprego de um algoritmo *round robin* assimétrico, onde, por exemplo, a fila de processos de sistema poderia receber uma janela de processamento de 50% enquanto as demais receberiam janelas progressivamente menores, tais como 25%, 15%, 10% e 5%.

Uma alternativa simplificada de emprego deste algoritmos é o uso de apenas duas filas, uma para processos em primeiro plano (*foreground*) e outra para processos em segundo plano (*background*), de modo que as duas filas utilizam um algoritmo *round robin* convencional e a divisão do processamento entre as filas utilizasse um *round robin* assimétrico que dedicasse 80% do processamento para a fila de primeiro plano e os 20% restantes para fila de segundo plano.

3.5.8 Escalonamento MFQ (*Multilevel Feedback Queues*)

O escalonamento MFQ (*Multilevel Feedback Queues*) ou filas multinível realimentadas é um interessante esquema de divisão de trabalho do processador que é baseado em várias filas encadeadas, como mostra a Figura 3.12.

Diferentemente do algoritmo MQ, que dispõe de filas distintas para processos de tipos diferentes, todos os novos processos são colocados inicial-

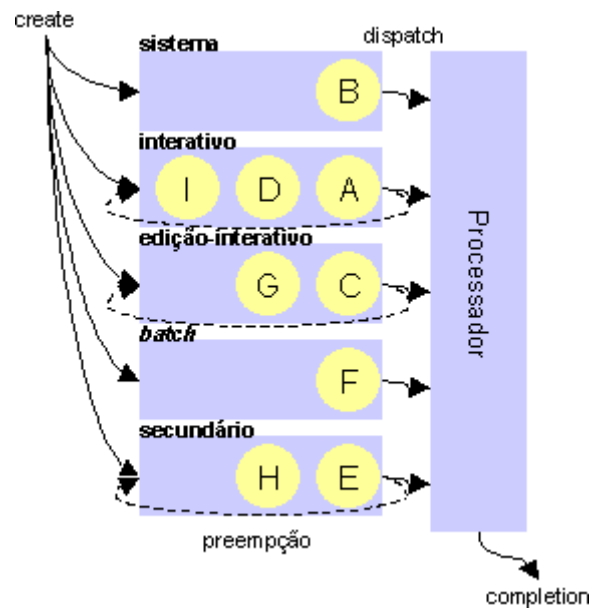


Figura 3.11: Escalonamento MQ (*Multiple queues*)

mente na fila de nível 1, que tem um comportamento FIFO, ou seja, o processo aguarda sua vez para execução conforme sua ordem de chegada. Ao utilizar o processador podem ocorrer três situações:

1. o processo é finalizado e então retirado das filas;
2. o processo solicita o uso de dispositivos de E/S, sendo bloqueado até que o pedido de E/S seja atendido, voltando para a mesma fila até que seu quantum de tempo se esgote; e
3. tendo esgotado seu quantum inicial de tempo, o processo é colocado no final da fila de nível 2.

Nas filas seguintes o mecanismo é o mesmo, embora os processos só utilizem o processador na sua vez e na ausência de processos nas filas de nível superior. Quando novos processos aparecem em filas superiores ocorre a preempção dos processos nas filas nível inferior, de forma a atender-se os processos existentes nas filas superiores.

A última fila apresenta um comportamento um pouco diferente: ela não é uma FIFO e sim uma fila circular, ou seja, de escalonamento *round robin*, onde os processos permanecem até que seja finalizados.

Neste esquema de escalonamento temos que:

- processos curtos são favorecidos pois recebem tratamento prioritário enquanto permanecem nas filas de nível superior;

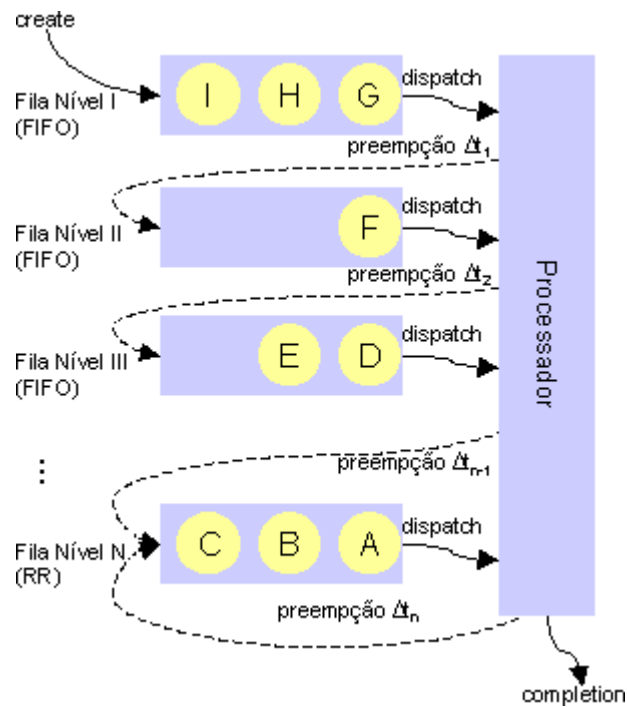


Figura 3.12: Escalonamento MFQ (*Multiple feedback queues*)

- processos com utilização intensa de E/S são favorecidos pois o uso de E/S não os desloca para filas inferiores; e
- processos de processamento maior também são favorecidos pois os *quanta* de tempo são progressivamente maiores nas filas de nível inferior.

O esquema de múltiplas filas encadeadas tem comportamento semelhante à um esquema de prioridades, onde as filas de nível superior equivalem aos níveis de prioridade mais altos do sistema e a última fila, de escalonamento *round robin* corresponde ao nível de prioridade mais baixo.

Considerando que um algoritmo de escalonamento deveria avaliar a natureza dos processos em execução promovendo um escalonamento adequado de modo que, minimamente, fossem favorecidos:

- processos de curta duração de forma a minimizar os tempos médios de resposta; e
- processos que demandem dispositivos de E/S para obter adequada utilização dos periféricos do sistema.

Notamos que o MFQ constitui uma alternativa bastante atraente, pois pode ser considerado um algoritmo adaptativo, ou seja, é capaz de perceber

o comportamento de um processo, favorecendo-o através da recolocação em uma fila de nível adequado. O critério de recolocação de processos nas mesmas filas após uso de dispositivos de E/S influencia grandemente no quanto este esquema de escalonamento é adaptativo, isto é, no quanto ele é capaz de atender aos diferentes padrões de comportamento dos processos. Embora a sobrecarga para administração deste esquema de escalonamento seja maior, o sistema torna-se mais sensível ao comportamento dos processos, separando-os em categorias (os vários níveis das filas), possibilitando ganho de eficiência.

3.6 Comparação dos algoritmos de escalonamento

Nas Tabelas 3.8 e 3.9 agrupamos as principais características dos algoritmos de escalonamento estudados nas seções anteriores, permitindo a avaliação comparativa em termos de sua função de seleção, preemptividade, *throughput* (produtividade), tempo de resposta e sobrecarga apresentados.

Tabela 3.8: Algoritmos de escalonamento não preemptivos

Característica	FIFO	HPF	SJF	HRN
Função de Seleção	$\min(t_{chegada})$	$\max(prio)$	$\min(t_{servico})$	$\max(t_{pn})$
Preemptividade	não	não	não	não
<i>Throughput</i>	média	média	alta	alta
$T_{resposta}$	alto	baixo:médio	baixo:médio	baixo:médio
Sobrecarga	baixa	baixa:alta	baixa:alta	baixa:alta
Adiamento Indefinido	não	sim	sim	não

Dentre os algoritmos não preemptivos (Tabela 3.8) temos que apresentam o seguinte comportamento geral:

- O algoritmo FIFO é o mais simples e também o mais ineficiente, pois acaba penalizando processos pequenos que ocorram entre processos maiores e também não trata o uso de dispositivos de E/S da maneira adequada, penalizando o sistema como um todo.
- O algoritmo HPF é relativamente simples e permite implantar uma política diferenciada de escalonamento através da definição de prioridades. No entanto os resultados do escalonamento tendem a se aproximar do FIFO quando ocorrem vários processos com o mesmo nível de prioridade.
- O escalonamento SJF penaliza os processos longos, sendo possível o adiamento indefinido destes. Por outro lado, proporciona uma alta

produtividade as custas de uma maior sobrecarga e tempos de resposta médios.

- O HRN é um dos melhores algoritmos de escalonamento não preemptivos, pois apresenta alta produtividade, tempos de resposta adequados e bom equilíbrio entre atendimento de processos grandes e pequenos. Além disso não possibilita a ocorrência de adiamento indefinido.

Tabela 3.9: Algoritmos de escalonamento preemptivos

Característica	RR	SRT	MQ	MFQ
Função de Seleção	constante	$\min(t_{servrest})$	complexa	complexa
Preemptividade	sim (<i>quantum</i>)	sim (chegada)	sim (<i>quantum</i>)	sim (<i>quantum</i>)
<i>Throughput</i>	média	alta	média	média
$T_{resposta}$	baixo:médio	baixo:médio	baixo:médio	baixo:médio
Sobrecarga	baixa	média:alta	média:alta	média:alta
Adiamento Indefinido	não	sim	sim	sim

Analisando agora os algoritmos preemptivos (Tabela 3.9), todos possuem implementação mais sofisticada que os algoritmos não preemptivos

- A solução *round robin* é uma alternativa geral simples, que proporciona tratamento justo e é ainda razoavelmente produtivo, pois sua eficiência se acha bastante associada ao tamanho do *quantum*.
- O escalonamento SRT apresenta produtividade alta e tempos de resposta adequado, mas desfavorece processos longos além de possibilitar o adiamento indefinido. A sobrecarga é um de seus aspectos negativos.
- O algoritmo MQ é uma solução que pode ser ligeiramente mais complexa que o *round robin* ou tão complexa como o MFQ. Permite obter bons resultados quando a categorização dos processos é possível.
- O algoritmo MFQ é razoavelmente mais complexo, implicando em maior sobrecarga, oferecendo o mesmo nível de produtividade do *round robin*. Por outro lado se mostra mais eficiente em situações que os processos exibem comportamento *I/O bounded*, típico de aplicações comerciais com processamento simples de grandes quantidades de dados.

Capítulo 4

Gerenciamento de Memória

Neste capítulo vamos tratar do gerenciamento de memória e de suas implicações para com as funcionalidades oferecidas pelos sistemas operacionais. Para isto veremos como as diferentes formas de endereçamento influenciam as capacidades de um sistema e as diferentes estratégias de organização e controle da memória.

4.1 Primeiras considerações

Antes de definirmos o que é o gerenciamento de memória, devemos primeiro considerar a estrutura dos computadores, tal como proposta no final da década de 1930 por John Von Neumann, matemático húngaro radicando nos EUA, que trabalhava na Universidade de Princeton:

Um computador é composto de três partes fundamentais: o processador, os dispositivos de entrada/saída e a memória.

Através desta estrutura, ilustrada na Figura 4.1, Von Neumann mudou radicalmente o conceito do computador, afirmando que o programa deveria ser armazenado na memória junto com os dados, transformando as sofisticadas máquinas de calcular existentes em uma nova espécie de máquina, completamente diferente, mais genérica e capaz de lembrar seqüências de comandos previamente fornecidas, executando-as fielmente.

Este novo conceito permitiu construir-se computadores capazes de executar diferentes seqüências de instruções sem a alteração de seu *hardware*, o que não acontecia com as antigas *máquinas de calcular gigantes*. Mas duas vertentes surgiram praticamente na mesma época, direcionando diferentemente a arquitetura de sistemas computacionais baseados em memória. Conforme forma organizados, receberam o nome de **arquitetura de Von Neumann** ou **arquitetura Princeton**.

Os pesquisadores da Universidade de Harvard propuseram uma arquitetura ligeiramente diferente da proposta inicial de Von Neumann, onde

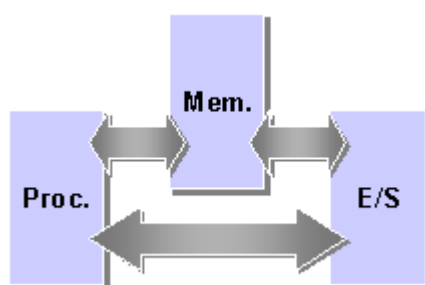


Figura 4.1: Arquitetura Princeton

existiam dispositivos de memória distintos para o armazenamento de dados e programas (veja a Figura 4.2). O grande diferencial é que a arquitetura de Princeton possui um único barramento para transferência de dados e instruções entre memória e processador enquanto que a arquitetura de Harvard possui dois barramentos, cada um ligando o processador aos dispositivos de memória de dados e programas respectivamente. Apesar da arquitetura de Harvard ser potencialmente mais eficiente, a simplicidade da arquitetura Princeton predominou com o passar dos anos.

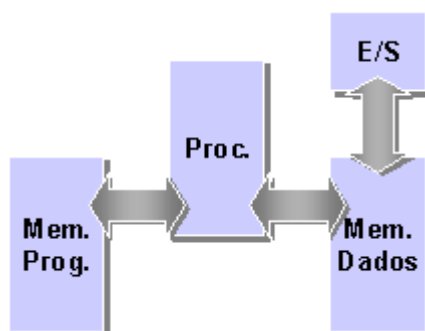


Figura 4.2: Arquitetura Harvard

Apesar destas duas arquiteturas serem conceitos revolucionários para a época, existia um grave problema: não se dispunha de tecnologia suficiente para construção de dispositivos eletro-eletrônicos que funcionassem como memórias.

Quando Von Neumann propôs o computador como uma máquina capaz de memorizar e executar seqüências de comandos, já existia tecnologia para implementar-se os primeiros circuitos processadores e também os dispositivos básicos de entrada e saída, mas não se sabia como construir circuitos de memória. Se passaram vários anos até que uma primeira implementação satisfatória de um circuito de memória permitisse construir um computador conforme sugerido por Von Neumann. Isto explica a origem de uma defasagem tecnológica, até hoje não superada por completo, entre memórias e

processadores.

O próprio termo *core memory*, utilizado para designar a memória primária ou principal, tem sua origem nos primeiros dispositivos de memória construídos com pequenos núcleos magnetizáveis de ferrite interligados matricialmente por uma delicada fiação de cobre, onde cada núcleo armazenava um único *bit*.

Tipicamente tem-se que os processadores são capazes de ler e principalmente escrever mais rápido do que os circuitos de memória do mesmo nível tecnológico. Isto usualmente força uma determinada espera por parte dos processadores quando ocorre a troca de informações com a memória. Estes tempos de espera são conhecidos como *wait states*. Este problema não seria tão grave se a interação entre processador e memória não constituísse a própria essência do funcionamento dos computadores baseados na arquitetura de Von Neumann.

Na arquitetura de Von Neumann o processador exibe o seguinte comportamento (como ilustrado na Figura 1.3) enquanto estiver ativo são repetidas as ações seguintes:

- busca de uma instrução (ciclo de *fetch* ou *opcode fetch*);
- decodificação da instrução (ciclo de *instruction decode*); e
- execução da instrução (ciclo de *instruction execution*).

Notamos que o ciclo de *fetch* é basicamente uma operação de leitura da memória, sendo o ciclo de decodificação interno ao processador, enquanto o ciclo de execução pode realizar tanto operações internas ao processador como também operações de leitura ou escrita na memória.

Apesar das dificuldades encontradas, a arquitetura computacional baseada nos três elementos básicos (processador, memória e dispositivos de entrada/saída) ainda se mostra a melhor que existe, principalmente quando se inclui nela todos os avanços tecnológicos destes últimos 1950 anos.

Hoje temos que, em consequência da mencionada defasagem tecnológica e dos problemas inerentes a construção de dispositivos de memória, o custo relativo entre memória e principalmente o processador faz com que a memória represente uma parcela bastante significativa de um sistema computacional, parcela esta que se torna ainda mais significativa quanto mais sofisticado é tal sistema (usualmente são sistemas de alto desempenho).

É muito importante ressaltarmos que, quando se fala em memória, estamos nos referindo aos circuitos que trocam dados diretamente com o processador durante o ciclo de execução de seus comandos mais básicos, ou seja, a memória primária ou armazenamento primário. Os dispositivos de armazenamento secundário, isto é, os dispositivos de memória de massa tal como fitas, cartuchos e disco magnéticos (fixos ou removíveis) não devem ser confundidos com a memória básica do computador.

A memória dita primária possui interação direta com o processador, isto é, fica diretamente conectada aos seus barramentos. Já os dispositivos chamados de memória secundária necessitam de alguma circuitaria ou memos mecânica extra para que os dados presentes no barramento do processador sejam gravados em suas mídias e vice-versa. Como veremos mais tarde, a memória secundária terá papel fundamental dentro do gerenciamento da memória primária, mas isto deve ser tratado com cautela, pois apesar de seu custo ser bastante inferior ao da memória primária básica, sua velocidade é milhares de vezes menor, como indicado na Tabela 4.1.

Tabela 4.1: Valores Médios de Tempo de Acesso e Preço por MByte para Dispositivos de Microcomputadores PC Compatíveis em 1998

Dispositivo	Tempo de Acesso (ms)	Preço/MByte (US\$)
Cartuchos	200	0.05
Disco Rígido	13	0.22
SIMM	45	12.00
DIM	25	16.00

Sendo assim, somente uma cuidadosa análise de custo versus benefício, que leve em conta outros fatores inerentes ao gerenciamento de memória, poderá diagnosticar com precisão como e quando utilizar-se da memória secundária ao invés da memória primária.

Resumidamente, justificamos a necessidade do gerenciamento de memória pelos três fatores relacionados a seguir:

- A memória primária é um dos elementos básicos da arquitetura computacional atual.
- Dado que a velocidade de suas operações de leitura e escrita serem mais baixas que a correspondentes velocidades dos processadores e ainda seu custo ser mais elevado, exige-se seu uso cuidadoso.
- Como todo processo utiliza memória primária, ao gerenciarmos memória estamos indiretamente gerenciando os processos.

4.2 Multiprogramação

Como já vimos, a **multiprogramação** é uma importante técnica utilizada para o projeto e construção de sistemas operacionais. Segundo Guimarães:

A maioria dos computadores modernos opera em regime de multiprogramação, isto é, mais de um programa em execução "simultânea" na memória. A existência nesses sistemas de pe-

reféricos assíncronos e com velocidades de operação as mais diversas tornou economicamente necessário introduzir a multiprogramação a fim de utilizar de maneira mais eficiente os recursos do sistema. [GUI86, p. 71] [grifos do autor]

Da mesma forma, isto também permite que uma tarefa seja dividida em partes, as quais podem ser executadas paralelamente, reduzindo o tempo total de processamento. Em suma, a multiprogramação visa alcançar melhores índices de produtividade num sistema computacional.

Em computadores de pequeno porte ou destinados ao uso pessoal é tolerável uma situação de baixa eficiência ou baixa produtividade, onde tanto o processador como os dispositivos periféricos permanecem ociosos por boa parte do tempo de funcionamento do equipamento.

Tomemos como exemplo um programa (ou rotina) conversor de imagens de formato BMP para GIF que, num sistema monoprogramado, é executado utilizando 8 segundos da atividade do processador (a conversão da imagem propriamente dita) e 24 segundos de atividade de dispositivos de E/S (gastos para carregamento do arquivo de entrada e armazenamento do arquivo de saída produzido pela rotina. A execução deste programa impõe a situação relacionada na Tabela 4.2.

Tabela 4.2: Análise de ociosidade

Tempo Total	32 s
Tempo de Proc	8 s
Tempo de I/O	24 s
Ociosidade do Proc.	75 %
Ociosidade do Disco	25 %
Ociosidade de Outros Disp.	100 %

A ociosidade do processador exibida neste exemplo representa um desperdício de sua capacidade equivalente a execução de outros três programas idênticos. Constatação análoga pode ser feita para os dispositivos de E/S deste sistema.

Tal situação não é admissível em sistemas de maior porte e, portanto, de maior custo de aquisição e propriedade. As possibilidades oferecidas por sistemas monoprogramados impedem que melhores índices de produtividade e eficiência sejam atingidos, neste sentido, a multiprogramação é a alternativa que permite, as custas de um sistema de maior complexidade, obter índices adequados de produtividade e eficiência.

Basicamente, o objetivo da **multiprogramação** é maximizar a produtividade (*throughput*) e minimizar os tempos de resposta. Para isto é necessário a presença de vários programas ativos, simultaneamente na memória principal do sistema de maneira a obter-se a melhor utilização possível dos recursos do sistema. O maior problema da multiprogramação é, portanto,

tornar compatíveis os tempos de execução dos programas e o atendimento dos usuários junto das diferentes velocidades de operação dos periféricos do sistema.

Se partirmos de um modelo bastante simplificado de sistema onde posamos supor verdadeiras as seguintes condições:

- a sobrecarga imposta pelos mecanismos de administração do sistema operacional é desprezível;
- os processos sejam totalmente independentes; e
- existe capacidade de efetuar-se processamento verdadeiramente paralelo.

Nestes casos a taxa de ociosidade e utilização do processador podem ser dadas respectivamente pelas seguintes relações:

$$ociosidade = t_{io}^p \quad (4.1)$$

$$utilizacao = 1 - ociosidade = 1 - t_{io}^p \quad (4.2)$$

Nestas equações p é o número de processos ativos existentes no sistema e t_{io} representa a taxa média de utilização de dispositivos de E/S por parte destes processos. Através desta relação, podemos obter as seguintes curvas ilustrando o comportamento da utilização do processador para um número variável de processos, considerando-se diferentes taxas médias de utilização de E/S.

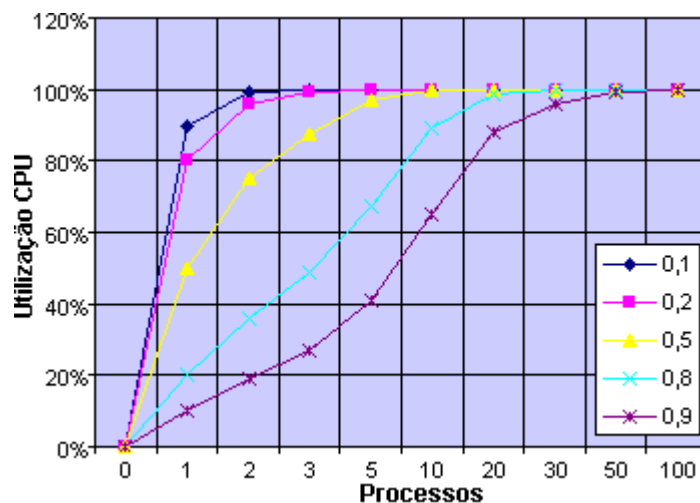


Figura 4.3: Comportamento da taxa de utilização do processador

Notamos que conforme aumenta a taxa média de utilização de dispositivos de E/S (t_{io}) por parte dos processos, maior é o número p de processos necessário para que a utilização do processador se mantenha em níveis adequados, ou seja, $utilizacao > 85\%$. Isto pode ser entendido de outra forma: quanto maior a taxa média de utilização de dispositivos de E/S, maior é o número de usuário suportado pelo sistema.

Apesar de ser uma simplificação, tais valores tem valor indicativo, ou seja, o comportamento esperado em sistemas reais é o mesmo a despeito dos valores absolutos obtidos. Com isto justifica-se a necessidade de ambientes multiprogramados como única forma de obter-se sistemas de alta produtividade e eficiência.

4.3 Organização da memória

Num sistema computacional o armazenamento de dados ocorre hierarquicamente, ou seja, em diversos níveis dado que é realizado em diferentes tipos de dispositivos devido à quatro fatores básicos:

- tempo de acesso
- velocidade de operação
- custo por unidade de armazenamento
- capacidade de armazenamento

Com isto em mente, o projetista de um sistema operacional determina quanto de cada tipo de memória será necessário para que o sistema seja ao mesmo tempo eficiente e economicamente viável.

Em virtude das dificuldades tecnológicas associadas a construção de dispositivos eficientes de memória e seu custo, o armazenamento de dados assumiu historicamente a seguinte organização:

- **Armazenamento interno**
São posições de memória disponíveis internamente ao processador para permitir ou agilizar sua operação. Constitui-se dos registradores do processador e de seu *cache* interno.
- **Armazenamento primário**
São as posições de memória externa, diretamente acessíveis pelo processador. Tipicamente são circuitos eletrônicos integrados do tipo RAM, EEPROM, EPROM, PROM ou ROM.
- **Armazenamento secundário**
São as posições de memória externa que não podem ser acessadas

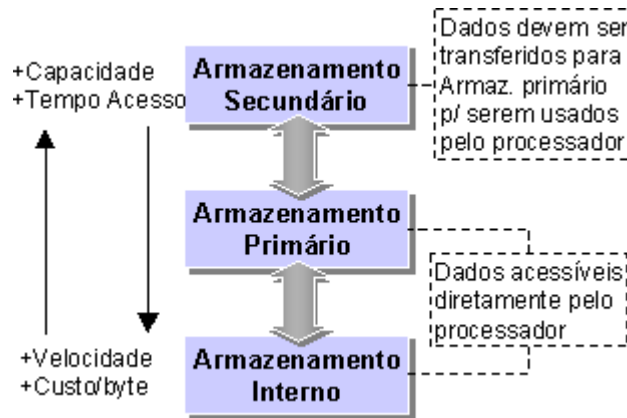


Figura 4.4: Organização da memória em níveis

diretamente pelo processador, devendo ser movidas para o armazenamento primário antes de sua utilização. Tipicamente dispositivos de armazenamento de massa tais como unidades de disco e fita.

Note que o armazenamento interno é aquele que possui as maiores velocidades de acesso, ou seja, os menores tempos de acesso representando os melhores dispositivos em termos de performance, embora sendo os mais caros. Disto decorre sua implementação em quantidades menores. Em contrapartida, os dispositivos de armazenamento secundário são os de maior capacidade e de melhor relação custo por *byte*, mas significativamente mais lentos. A memória primária representa um caso intermediário, onde a velocidade e tempo de acesso são adequadas à operação direta com o processador, mas cujo custo ainda assim é elevado.

Com a evolução dos computadores, a atual organização conta com outros elementos adicionados para otimizar a performance do sistema e ainda assim reduzir seu custo, conforme a figura a seguir:



Figura 4.5: Organização típica de armazenamento

Os registradores, implementados em número limitado devido ao seu custo, são geralmente utilizados para manter dentro do processador dados freqüentemente utilizados. Os *cache* interno e externo, devido sua maior velocidade, são usados para manter uma porção do programa (que pode assim ser executada mais rapidamente do que na memória principal), ou uma porção de dados (evitando-se uso da memória principal) e com isto aumentando o desempenho do sistema [DEI92, p. 30].

A memória primária armazena os programas e dados em execução no sistema. Os dispositivos de armazenamento secundário são usados para preservação dos dados de forma perene, embora também possam ser usados para expandir as capacidades da memória primária. O cache de disco é utilizado para acelerar a operação das unidades de disco, podendo esta técnica ser utilizada para outros tipos de periféricos.

4.4 Definição de gerenciamento de memória

A necessidade de manter múltiplos programas ativos na memória do sistema impõe outra, a necessidade de controlarmos como esta memória é utilizada por estes vários programas. O **gerenciamento de memória** é, portanto, o resultado da aplicação de duas práticas distintas dentro de um sistema computacional:

1. Como a memória principal é vista, isto é, como pode ser utilizada pelos processos existentes neste sistema.
2. Como os processos são tratados pelo sistema operacional quanto às suas necessidades de uso de memória.

Como a memória é um recurso caro, cuja administração influencia profundamente na eficiência e performance de um sistema computacional, é necessário considerar-se três estratégias para sua utilização:

1. Estratégias de busca

As estratégias de busca (*fetch strategies*) preocupam-se em determinar qual o próximo bloco de programa ou dados que deve ser transferido da memória secundária para a memória primária. Usualmente se utilizam estratégias de demanda, ou seja, são transferidos os blocos determinados como necessários para a continuação do processamento.

2. Estratégias de posicionamento

São as estratégias relacionadas com a determinação das regiões da memória primária (física) que serão efetivamente utilizados pelos programas e dados, ou seja, pela determinação do espaço de endereçamento utilizado (*placement strategies*).

3. Estratégias de reposição ou substituição

São as estratégias preocupadas em determinar qual bloco será enviado a memória secundária para disponibilização de espaço na memória principal para execução de outros programas, ou seja, determinam quais blocos de memória serão substituídos por outros (*replacement strategies*).

Minimamente, todo sistema computacional possui alguma estratégia de busca e alguma estratégia básica de posicionamento. O aumento da sofisticação dos sistemas computacionais exige a utilização de estratégias de busca e posicionamento mais sofisticadas. Para maximizar-se as capacidades dos sistemas computacionais são necessárias as estratégias de reposição.

Historicamente, o desenvolvimento da organização e gerenciamento de memória foi grandemente afetado pelo próprio desenvolvimento dos computadores e evolução dos sistemas operacionais. Os modos básicos de organização da memória dos sistemas são:

- monoprogramado
- multiprogramados com armazenamento real, particionamento fixo e endereçamento absoluto
- multiprogramados com armazenamento real, particionamento fixo e endereçamento relocável
- multiprogramados com armazenamento real, de particionamento variável
- multiprogramados com armazenamento virtual paginado
- multiprogramados com armazenamento virtual segmentado
- multiprogramados com armazenamento virtual combinado

Na Figura 4.6 temos um quadro onde se ilustra o relacionamento dos modelos básicos de organização da memória e, de certa forma, sua evolução.

Com relação ao primeiro aspecto básico da gerência de memória, para entendermos como os processos *exercitam* a memória, é necessário conhecer em detalhe como os programas se comportam durante sua execução.

O comportamento exibido pelos programas durante sua execução cria determinadas limitações que devem ser observadas cuidadosamente pelo sistema operacional através de seu gerenciamento de memória. Por outro lado, os programas também devem se comportar dentro de regras estabelecidas pelo próprio sistema operacional, as quais compõem o modelo de administração de memória empregado pelo sistema.

Para sabermos como se comporta um programa durante sua execução e quais são suas limitações quanto a utilização da memória, devemos analisar todo o processo de criação dos programas.

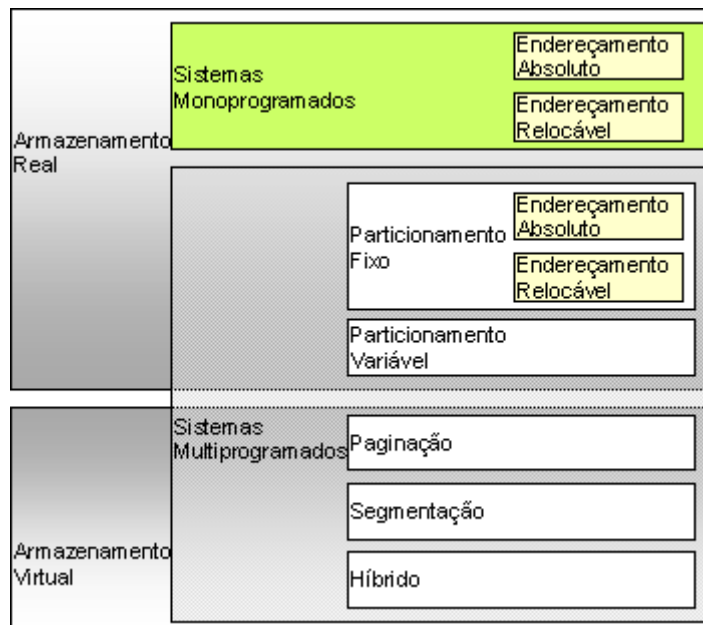


Figura 4.6: Evolução da organização da memória

4.5 Criação de programas

Os programas são criados a partir de **arquivos-texto**, que contêm um roteiro estruturado de passos e ações a serem executadas pelo programa que se deseja., ou seja, estes arquivos-texto são uma representação dos algoritmos que se desejam programar. Estes passos e ações estão descritos dentro do arquivo-texto através de uma **linguagem de programação** e por isso são usualmente chamados de **arquivo-fonte do programa** (resumidamente **arquivo-fonte** ou **fonte**). As linguagens de programação utilizadas podem ser de alto, médio ou baixo nível, mas qualquer que seja a linguagem, seu tipo e a forma de estruturação do programa, o arquivo-fonte continua a ser simplesmente um texto, análogo à uma redação, sujeito à regras de sintaxe e de contexto.

Da mesma forma que os computadores não entendem a nossa linguagem, ou seja a linguagem que naturalmente utilizamos para nossa comunicação, estas máquina tão pouco entendem as linguagens de programação diretamente. Existem *entidades* especiais responsáveis pela transformação do arquivo-fonte do programa em uma forma passível de execução pelo computador. Estas entidades estão ilustradas na Figura 4.7.

O **compilador** (*compiler*) é um programa especial que traduz o arquivo fonte em um arquivo binário que contêm instruções, dados e endereços (representados binariamente) que permitem executar as ações necessárias através das instruções em linguagem de máquina do processador existente

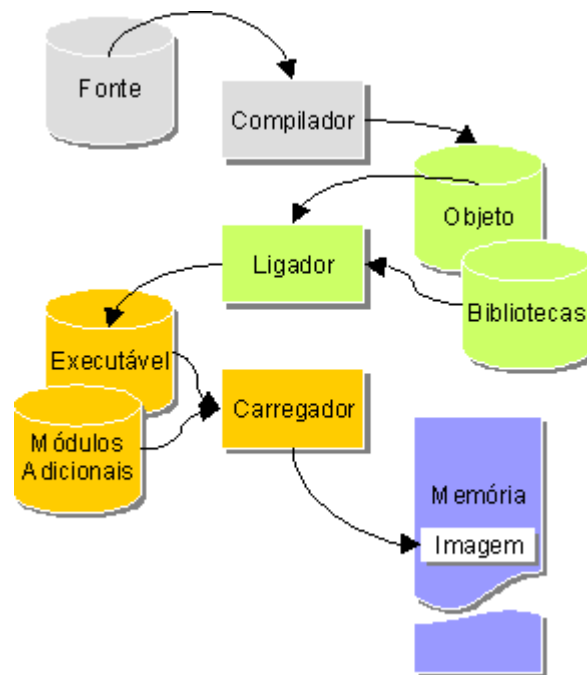


Figura 4.7: Esquema de criação de programas

no computador em questão. Os arquivos binários produzidos pelo compilador são os **arquivos-objeto** ou resumidamente **objeto**. Note que cada compilador é apropriado para uma única linguagem de programação.

O **ligador** (*linker*), quando necessário, apenas encadeia dois ou mais arquivos objeto sob a forma de um único arquivo de programa executável ou **arquivo-executável**. O arquivo-executável é aquele que pode ser transferido para a memória do computador possibilitando a execução do programa. Assim como os compiladores, o ligador também é uma entidade deste processo de geração de programas e também está sujeito a operar com arquivos objeto produzidos apenas por determinados compiladores.

Devemos ressaltar que até agora os arquivos fonte, objeto e executável constituem arquivos, ou seja, estão armazenados nas estruturas de memória secundária (unidades de disco rígido, discos flexíveis, fitas, cartuchos ou discos ópticos).

Existe uma outra entidade especial, chamada **carregador** (*loader*), que é parte integrante do sistema operacional, responsável por transportar os arquivos de programa executável da memória secundária para a memória principal, onde se dará a execução do programa carregado.

Os carregadores constituem uma parte do sistema operacional porque a colocação de programas na memória e a execução dos mesmos são funções deste, responsável por controlar eficientemente as estruturas de memória primária, de armazenamento secundário e o processamento do sistema com-

putacional.

Após o transporte do arquivo executável para a memória principal é possível iniciar sua execução, onde ele mesmo se transforma numa **imagem executável**, que representa a expansão do código de programa contido no arquivo executável em código executável, áreas de memória reservadas para variáveis do programa, pilha retorno e área extra para alocação dinâmica por parte do programa. A bem da verdade, o sistema operacional, antes da carga do módulo de código, deve conhecer de antemão seu tamanho total e a quantidade mínima de memória extra necessária. Tais informações residem geralmente num cabeçalho (*header*) localizado no início do arquivo de programa executável, que não é copiado para memória, mas apenas lido pelo sistema operacional.

4.5.1 Espaços lógicos e físicos

Retomemos os conceitos envolvidos com os arquivos de programa fonte. Qual é o objetivo básico de um programa? A resposta é: ensinar o computador a executar um seqüência de passos, manuseando dados de forma interativa ou não, com o objetivo final de realizar cálculos ou transformações com os dados fornecidos durante a execução do programa.

Para isto, após o entendimento do problema, idealiza-se conceitualmente uma forma de representação do dados a serem manipulados e depois disso um conjunto de operações especiais que manipularão as estruturas criadas possibilitando a obtenção dos resultados esperados do programa.

Notem que ao projetar-se um programa, por mais simples ou complexo que ele seja, define-se um **espaço lógico** que reúne todas as abstrações feitas para criar-se o programa, sejam elas de natureza estrutural ou procedural/funcional. Tais abstrações são os objetos lógicos do programa. O espaço lógico contém todas as definições necessárias para o programa, mas sem qualquer vínculo com as linguagens de programação ou com os processadores e computadores que executarão os programas criados a partir desta concepção. O espaço lógico é a representação abstrata da solução do problema, também abstrata.

Durante a implementação dos programas utilizam-se, como meios de expressão, as linguagens de programação que possibilitam expressar de maneira concreta (apesar das limitações impostas por qualquer linguagem de programação) as formulações contidas no espaço lógico. Pode-se dizer assim que os programas fonte representam, numa dada linguagem de programação, o espaço lógico do programa. Num outro extremo, dentro do computador a execução do programa tem que ocorrer dentro da memória principal, como consequência e limitação da arquitetura de Von Neumann.

Seja qual for o computador e a particularização da arquitetura de seu *hardware*, a memória principal pode sempre ser expressa como um vetor, unidimensional, de posições de memória que se iniciam num determinado

ponto, usualmente o zero, e terminam em outro, 536.870.912 para um computador com 512 Mbytes de memória, por exemplo. Cada posição desta estrutura de memória é idêntica, podendo armazenar o que se chama de palavra de dados do computador, na prática um conjunto de *bits*.

Se a palavra de dados tem 4 *bits*, a memória está organizada em *nibbles*. Palavras de dados de 8, 16 e 32 *bits* representam, respectivamente, organizações de memória em *bytes*, *words* e *double words*. Tipicamente se organizam as memórias dos microcomputadores em *bytes*, assim cada posição de memória poderia armazenar um *byte* de informação, sendo que nos referenciamos as posições de memória pelos seus números de posição, os quais são chamados de **endereços**. Como a memória de um computador é um componente eletrônico, fisicamente palpável, dizemos que os endereços de memória representam fisicamente a organização de memória de um computador.

Sabemos que um programa quando em execução na memória principal de um computador se chama imagem executável e que esta imagem ocupa um região de memória finita e bem determinada. Ao conjunto de posições de memória utilizado por uma imagem se dá o nome de **espaço físico** desta imagem.

De um lado a concepção do programa, representada e contida pelo seu espaço lógico. Do outro lado, durante a execução da imagem temos que as posições de memória usadas constituem o espaço físico deste mesmo programa. De alguma forma, em algum instante o espaço lógico do programa foi transformado e ligado a organização de memória do sistema computacional em uso, constituindo o espaço físico deste mesmo programa.

A ligação entre o espaço lógico e físico representa, na verdade, a um processo de mapeamento, onde cada elemento do espaço lógico é unido de forma única a uma posição de memória do computador, acabando por definir um espaço físico. A este processo de ligação se dá o nome de **mapeamento** ou *binding* (amarração), como representado na Figura 4.8. No nosso contexto *binding* significa o mapeamento do espaço lógico de um programa no espaço físico que possibilita sua execução dentro do sistema computacional em uso.

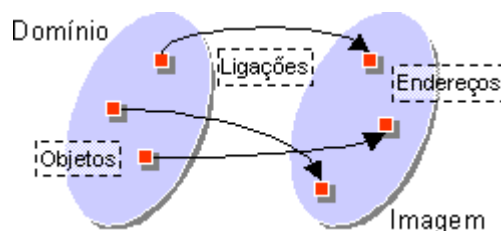


Figura 4.8: Representação do *binding*

Veremos a seguir que o *binding* tem que ser realizado por uma das entidades envolvidas no processo de criação de programas, ou seja, em alguns ins-

tante da compilação, ligação, carregamento ou mesmo execução. É possível também que o *binding* seja realizado por mais de uma destas entidades, onde cada uma realiza uma parcela deste processo de mapeamento.

4.5.2 Compiladores (*compilers*)

Como já foi definido, um **compilador** é um programa especial capaz de traduzir um arquivo escrito em uma linguagem de programação específica em um outro arquivo contendo as instruções, dados e endereços que possibilitam a execução do programa por um processador particular. O compilador é, portanto, capaz de *entender* um algoritmo expresso em termos de uma linguagem de programação, convertendo-o nas instruções necessárias para sua execução por um processador particular (vide Figura 4.9).

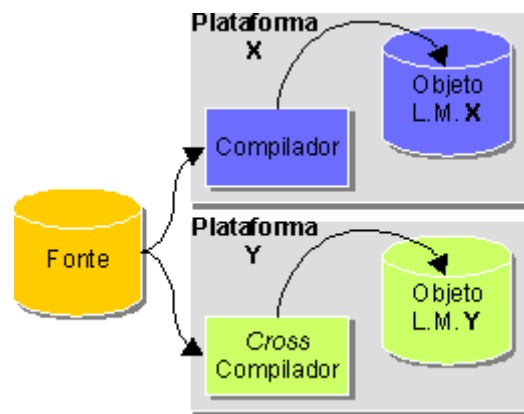


Figura 4.9: Compilador e *cross*-compilador

Todas as definições internas são transformadas em código. Funções, estruturas de dados e variáveis externas, tem apenas o local de chamada marcado em tabelas de símbolos externos para ligação posterior com bibliotecas ou outros módulos de código. Além de considerar o processador que executará tais instruções, alguns aspectos da arquitetura e do sistema operacional devem ser observados pelos compiladores como forma de produzir código verdadeiramente útil para uma dada arquitetura computacional.

No Exemplo 4.1 temos um trecho de código escrito em linguagem de alto nível e um possível resultado de compilação.

Os compiladores podem gerar código de duas maneiras básicas, isto é, empregando dois modos de endereçamento: o absoluto e o relocável.

Quando no modo de **endereçamento absoluto**, o compilador *imagina* que o programa será sempre executado numa única e bem determinada região de memória. Sendo assim, durante a compilação, o compilador associa diretamente posições de memória a estruturas de dados, variáveis, endereços de rotinas e funções do programa. Em outras palavras, o compilador fixa

<pre>// trecho de código fonte; while (...) { : a = a + 1; : printf("%d\n", a); : }</pre>	<pre>trecho de código compilado 0200: : LOAD 500 : CALL printf : JNZ 0200</pre>
---	---

Exemplo 4.1 Resultado da compilação

os endereços de execução do programa, realizando por completo a *binding*, tornando-se assim **compiladores absolutos**.

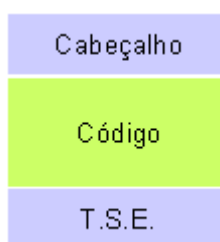


Figura 4.10: Arquivo objeto gerado através de compilação absoluta

Na Figura 4.10, onde se apresenta a estrutura interna de arquivos gerados através de compilação absoluta, temos os elementos seguintes:

Cabeçalho Região onde são colocadas informações gerais sobre o arquivo objeto e suas partes. Também conhecido como *header*.

Código Segmento onde reside o código do programa, propriamente dito. É gerado da mesma forma que na compilação absoluta.

TSE A **tabela de símbolos externos** é o local onde são listadas as posições de chamada de símbolos externos (variáveis, estruturas ou funções).

Os arquivos objeto produzidos tem seus endereços calculados a partir de um endereço de origem padronizado ou informado antes da compilação. Este endereço de origem, a partir do qual os demais são definidos, é chamado de **endereço base de compilação** ou apenas de **endereço base**. Desta maneira a compilação se torna mais simples, mas como consequência direta disto temos que:

- um arquivo de programa executável só pode ser executado numa região fixa de memória;
- não podem existir duas ou mais instâncias do mesmo programa executável na memória, a não ser que se realizem compilações adicionais forçando a geração do código para uso em diferentes regiões de memória;
- dois ou mais diferentes programas executáveis não podem ser carregados na memória a não ser que tenham sido compilados prevendo exatamente a ordem de carregamento e as áreas adicionais de memória que venham a utilizar;
- duas ou mais imagens executáveis não podem se sobrepor na memória (ocupar os mesmos endereços de memória) total ou parcialmente;
- uma imagem executável deve sempre ocupar uma região contínua de memória;
- a soma das imagens possíveis de serem carregadas em memória para execução paralela tem que ser menor ou igual a quantidade total de memória disponível.

As razões para estas limitações são simples e todas baseadas no fato de que dentro do arquivo objeto só existem números binários. Tais números representam tanto os códigos das instruções do processador como os dados constantes do programa e também os endereços determinados pelo compilador. Como existem apenas números binários em todo o arquivo objeto, não é trivial a distinção entre instruções, dados e endereços, tornando praticamente impossível:

- reverter a compilação, pois o *binding* se tornou irreversível, não sendo possível reconstituir-se o espaço lógico que originou o programa;
- modificar o endereçamento do programa pois não se pode distinguir que são os endereços dentro dos arquivos objeto gerados pelos compiladores absolutos.

Quando no modo de **endereçamento relocável**, o compilador continua realizando todas as suas tarefas, como no caso da compilação absoluta, fixando os endereços de execução durante a compilação, mas além de gerar o código o **compilador relocável** monta o arquivo objeto da seguinte forma:

O único elemento novo no arquivo objeto é a TER (**tabela de endereços relocáveis**) onde são relacionadas as posições de todos os endereços existentes dentro do bloco de código cujo valor depende da posição inicial do código, ou seja, lista todos os endereços relativos existentes.



Figura 4.11: Arquivo objeto gerado através de compilação relocável

Desta forma, a TER relaciona onde estão os endereços que deveriam ser modificados para permitir a transposição da imagem executável de uma região de memória para outra, constituindo o *segredo* dos compiladores relocáveis, pois é através desta tabela que o *binding* torna-se reversível, ou melhor, alterável, o que corresponde dizer que o *binding* não se realizou por completo. Portanto temos as seguintes implicações:

- ainda não é possível reverter-se a compilação em si pois apesar do *binding* ser alterável, ainda é impossível reconstituir-se o espaço lógico original do programa;
- é possível que outra entidade venha a modificar o atual endereçamento do programa, pois os endereços estão evidenciados na TER, sendo que tal modificação possibilita determinar o espaço físico do programa em função da disponibilidade de memória do sistema.

De qualquer forma o uso de compiladores relocáveis proporcionará as seguintes situações:

- um arquivo de programa executável poderá ser executado em diversas regiões de memória, a serem determinadas pelo sistema operacional;
- poderão existir duas ou mais instâncias do mesmo programa executável na memória, sem a necessidade da realização de compilações adicionais para forçar a geração do código para uso em diferentes regiões de memória;
- dois ou mais diferentes programas executáveis poderão ser carregados na memória sem que seja necessária a previsão da ordem exata de carregamento e as áreas adicionais de memória que venham a ser utilizadas;
- duas ou mais imagens executáveis não podem se sobrepor na memória (ocupar os mesmos endereços de memória) total ou parcialmente;

- uma imagem executável deve sempre ocupar uma região contínua de memória;
- a soma das imagens possíveis de serem carregadas em memória para execução paralela tem que ser menor ou igual a quantidade total de memória disponível.

Vemos que uma parte das limitações provocadas pelo uso de compiladores absolutos podem ser contornadas com o uso do modelo de compilação relocável. Como os ligadores não exercem participação no *binding* no tocante a modificação ou finalização do *binding*, temos que os carregadores são os candidatos naturais a finalização do *binding* no caso da compilação relocável.

4.5.3 Ligadores (*linkers*)

Programas capazes de unir parcelas de código, compiladas separadamente, em um único arquivo de programa executável. Através de símbolos e posições relacionados em tabelas de símbolos geradas pelos compiladores, os **ligadores** são capazes de unir trechos de código existentes em diferentes arquivos objeto em um único arquivo executável.

Os símbolos destas tabelas representam funções ou estruturas de dados que podem, dentro de certas regras, ser definidas e criadas em certos arquivos. Segundo estas mesmas regras, outros arquivos de programa fonte podem utilizar-se destas funções e estruturas sem a necessidade de redefini-las, bastando a indicação adequada de sua existência no exterior destes arquivos fontes. Assim sendo temos:

- **Módulos exportadores**

Aqueles onde são indicadas funções, estruturas de dados e variáveis que serão utilizadas por módulos externos. Utilizam variações de cláusulas *extern* ou *export*.

```
// Exportação de estruturas e variáveis em linguagem C
// estrutura de dados
typedef struct {
    char FuncName[ID_LEN];
    int Loc;
} FuncType;
// vetor de estruturas exportado
extern FuncType FuncTable[];
// variável inteira exportada
extern int CallStack[NUM_FUNC];
```

Exemplo 4.2 Declarações de exportação

- **Módulos importadores**

Aqueles onde são indicadas quais funções, estruturas de dados e variáveis encontram-se declaradas e implementadas em módulos externos. Utilizam variações de cláusulas *import* ou *include*.

```
// Importação de módulos em linguagem C
#include <vcl\Forms.hpp>
#include <vcl\Classes.hpp>
#include <vcl\ Windows.hpp>
#include <dos.h>
```

Exemplo 4.3 Declarações de importação

Cabe ao ligador a tarefa de unir os arquivos que contém estas definições aos arquivos que as utilizam, gerando disto um único arquivo de programa, como ilustrado na Figura 4.12. A ligação nunca afeta a maneira com que o *binding* foi ou será realizado, constituindo um elemento neutro dentro da criação dos programas quando analisada sob o aspecto de modelo de endereçamento.

Os ligadores participam do *binding* efetuando a união dos espaços físicos dos módulos a serem ligados como o programa executável, que determina o espaço físico definitivo. Os ligadores não exercem papel de modificação ou finalização do *binding*, tarefa que fica a cargo das entidades anteriores (os compiladores) ou posteriores (os carregadores e relocadores).

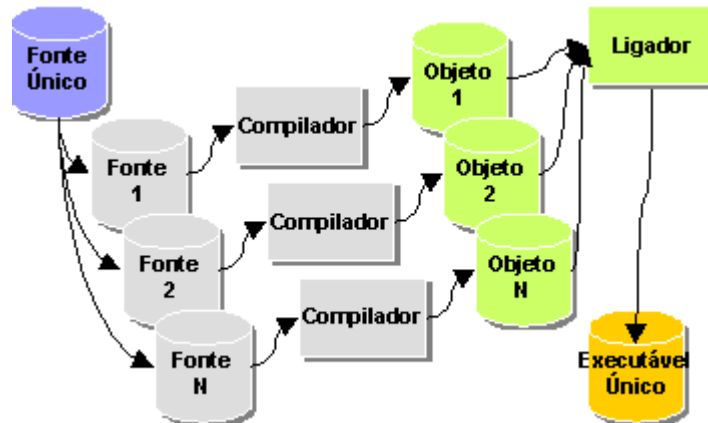


Figura 4.12: Esquema de compilação em separado e uso de ligador

A compilação em separado, com a conseqüente união posterior dos módulos produzidos através de um ligador, tem os seguintes objetivos:

1. Reduzir o tempo de desenvolvimento diminuindo os tempos consumidos durante a compilação através da partição do programa fonte em

pedaços (logicamente divididos e encadeados). Pode-se a partir desta divisão concentrar-se o trabalho em uma das partes de cada vez, que por ser menor toma um menor tempo de compilação. Quando se considera o resultado final, as diversas compilações intermediárias durante o desenvolvimento totalizam um menor tempo quando feita por partes do que quando o programa era manuseado por inteiro.

2. Permitir a divisão do trabalho dado que o programa pode ser dividido em partes. Seguindo o mesmo princípio que o da redução do tempo de desenvolvimento, as diversas partes podem ser implementadas paralelamente por equipes de 2 ou mais programadores. Assim os totais gastos são os mesmos quando se consideram a soma dos tempos gastos por cada elemento da equipe ou o custo de tal trabalho, mas tem-se a indiscutível vantagem de que o desenvolvimento pode ser realizado num prazo bastante inferior dado que as partes podem ser desenvolvidas em paralelo. Tal divisão requer cuidadoso projeto e especificação detalhada e consistente das partes do programa.
3. Permitir a padronização de código e a construção de bibliotecas de funções e estruturas de dados. Dado que é possível a compilação de uma parte de um programa contendo apenas funções (de uso geral ou específico) e estruturas de dados associadas, pode-se com isto distribuir-se estas funções e estruturas sob a forma compilada, ou seja um módulo objeto. Outros programadores poderão utilizar este módulo em seus programas, mas não poderão alterá-lo, daí obtém-se a padronização segura de funções e estruturas de dados. Para que isto seja possível basta seguir as regras de compilação por partes e acompanhar o módulo objeto de uma descrição das suas funções (parâmetros de entrada, resultados retornados) criando-se assim bibliotecas.
4. Permitir o uso de diferentes linguagens de programação dentro de um mesmo programa. Considerando a capacidade limitada dos Ligadores em interpretar as tabelas de símbolos geradas pelos compiladores, se vários compiladores, mesmo que de diferentes linguagens de programação, são capazes de gerar um formato compatível de informação simbólica, então um ligador apropriado será capaz de unir estes diferentes módulos num único arquivo de programa executável, mesmo que os módulos tenham sido escrito em diferentes linguagens. Na verdade, durante a implementação destes módulos, devem ser observadas as convenções de chamada para rotinas externas escritas em outras linguagens específicas para cada linguagem. Com isto podem ser aproveitadas bibliotecas escritas numa certa linguagem (as bibliotecas matemáticas do FORTRAN, por exemplo) em programas escritos em outras linguagens (C ou PASCAL). Através destas técnicas, podem ser melhor exploradas certas características das linguagens em programas

envolvendo várias linguagens (C e Clipper, C e DBase, C e SQL, C e PASCAL, VisualBasic e C, etc).

4.5.4 Carregadores (*loaders*)

Os **carregadores** são os programas responsáveis pelo transporte dos arquivos de programa executáveis das estruturas de armazenamento secundário (unidades de disco ou fita) para a memória principal. Os carregadores tem suas ações dirigidas pelo sistema operacional, que determina qual módulo executável deve ser carregado e em que região de memória isto deve ocorrer (endereço base de execução). Após o término do processo de carregamento, o carregador sinaliza ao sistema operacional que o programa foi carregado, neste ponto o sistema operacional determinará quando se iniciará a execução do programa, que se transforma em uma imagem executável ao iniciar sua execução efetivamente.

Quando o arquivo objeto foi gerado em **modo absoluto**, os carregadores apropriados para esta situação apenas realizam uma cópia do arquivo de programa executável, transferindo dados do dispositivo de armazenamento secundário (unidades de disco ou fita) para a memória principal. Nesta situação, tais carregadores são chamados de **carregadores absolutos** e recebem do sistema operacional apenas as informações do nome do arquivo executável a ser carregado e o endereço de carga (endereço a partir de onde se iniciará a cópia do código na memória principal).

Se o endereço de carga for o mesmo que o endereço base da compilação, a imagem resultante será executada sem problemas, de acordo com o que foi programado no fonte. Caso contrário as conseqüências são imprevisíveis, resultando geralmente na interrupção abrupta do programa pelo sistema operacional, na invasão da área de dados/código de outros programas, no cálculo impróprio dos resultados ou na perda de controle do sistema.

Quando temos que o arquivo objeto foi gerado em **modo relocável**, devem ser utilizados **carregadores relocáveis**, ou seja, carregadores capazes de interpretar o conteúdo da TER (tabela de endereços relocáveis) de forma a transpor a imagem executável da área original (iniciada/definida pelo endereço base de compilação) para uma outra área de memória.

A transposição da área de memória se baseia no fato de que durante a transferência do programa executável para a memória principal o carregador relocável, através da TER identifica quem são os endereços componente do código. Cada vez que o carregador relocável lê um endereço dentro do código, ele soma ao endereço lido (endereço original da compilação) o valor do endereço de carga fornecido pelo sistema operacional e com isto se realiza o modificação do *binding* (iniciado pelo compilador relocável) finalizando-se o mapeamento com a transposição da imagem para uma nova região de memória, determinada pelo sistema operacional e não pelo compilador.

$$End_{novo} = End_{original} + (End_{basecompilacao} - End_{basecarregamento}) \quad (4.3)$$

A TSE (tabela de símbolos externos) é lida pelo sistema operacional para que os módulos externos necessários ao programa sejam carregados previamente. Caso tais módulos sejam usados globalmente, isto é, compartilhados por vários programas (como as DLLs dos sistemas MS-Windows 95/98), o sistema operacional só realiza o carregamento quando tais módulos não estão presentes na memória.

Finalmente devemos observar que tanto o cabeçalho existente no arquivo de programa executável como a TSE não são transferidas para memória principal. No entanto a área ocupada pelo programa não corresponde apenas ao segmento de código contido no arquivo executável, sendo substancialmente maior, como ilustrado na Figura 4.13

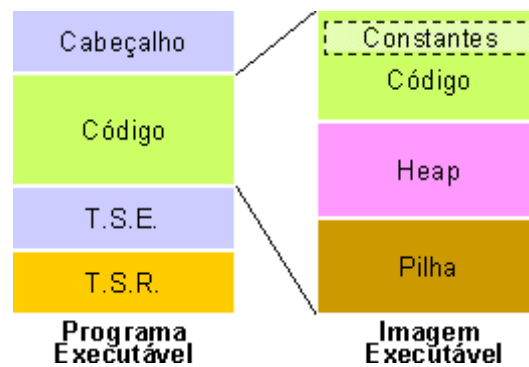


Figura 4.13: Estrutura típica de imagem executável

Tal *expansão* do código ocorre devido as necessidades dos programas de possuírem uma área adicional de memória para armazenamento da pilha dos endereços de retorno (*stack*) e outra área extra para armazenamento de conteúdo dinâmico do programa (*heap*), tais como variáveis locais e blocos de memória alocados dinamicamente. Por essa razão, após a transferência do código do armazenamento secundário para memória, chamamos tal conteúdo de memória de **imagem executável** ou simplesmente **imagem**.

4.5.5 Relocadores (*swappers*)

Os **relocadores** são rotinas especiais do sistema operacional responsáveis pela movimentação do conteúdo de certas áreas de memória primária para memória secundária (especificamente dispositivos de armazenamento como unidades de disco) e vice-versa, como ilustrado na Figura 4.14. A existência de relocadores num sistema depende do tipo de gerenciamento de memória oferecido pelo sistema operacional. Antes de verificarmos quais modelos

de gerenciamento de memória podem fazer uso dos relocadores, devemos compreender melhor a natureza de seu trabalho.

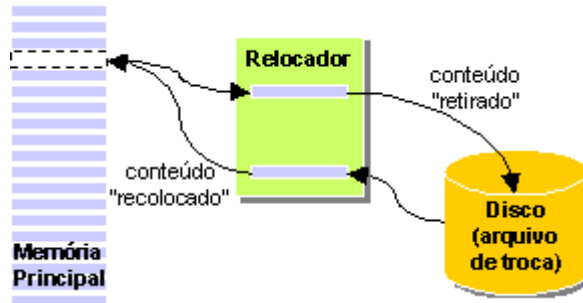


Figura 4.14: Conceito de relocação

Considerando que num sistema multiprogramado existem vários processos ativos (em estado *ready*), bloqueados (em estado *blocked*) e suspensos (em estado *suspended*), sabemos que apenas um processo está efetivamente em execução, isto é, utilizando o processador (estado *running*), então, a medida de suas necessidades, este processo em execução poderá solicitar áreas adicionais de memória. Estes pedidos de alocação de memória podem ser atendidos de duas formas: áreas efetivamente livres são cedidas ao processo ou aciona-se o relocador para liberação de áreas de memória pertencentes aos demais processos ativos e inativos através da remoção de seu conteúdo para os arquivos de troca, no que se denomina operação de troca ou *swapping*.

Seguindo instruções do sistema operacional, que detém o gerenciamento da memória e dos processos, um relocador pode ser comandado para *retirar* o conteúdo de uma área de memória armazenado-a em disco. O espaço em memória disponibilizado através desta operação é pode ser usado para atender pedidos de alocação de memória do processo correntemente em execução. Com isto, partes de alguns processos serão transferidas para o disco. Quando estes processos, cuja algumas de suas partes estão armazenadas no disco, necessitarem de tais conteúdos, uma nova operação de relocação pode ser efetuada para, novamente disponibilizar espaço, de modo que este seja agora usado para que o conteúdo das partes anteriormente *copiadas* em disco, seja recolocada na memória. Todas estas operações ocorrem sob orientação do sistema operacional.

O que geralmente ocorre é que o relocador realiza uma cópia das área de memória movimentadas para o disco em um arquivo especial denominado **arquivo de troca** ou *swap file*. Ao copiar tais áreas de memória para o disco, estas são assinaladas como livres, tornando-se disponíveis para outros processos. Também se efetua um registro do que foi copiado para memória possibilitando recuperar este conteúdo quando necessário.

Nesta situação, se avaliarmos a soma total de memória utilizada por todos os processos ativos, isto é, processos nos estados de *running* e também

em *ready*, temos que a quantidade de memória utilizada por eles pode ser significativamente maior do que a memória física instalada no sistema, pois alguns destes processos tiveram suas área de memória transferidas para a unidade de disco quando não estavam em execução. Este é o princípio básico que possibilita a implementação de memória virtual como será tratado a seguir (seção 4.6).

4.6 Memória virtual

O conceito de relocação de memória possibilitou o desenvolvimento de um mecanismo mais sofisticado de utilização de memória que se denominou **memória virtual** ou *virtual memory*. Segundo Deitel:

O termo memória virtual é normalmente associado com a habilidade de um sistema endereçar muito mais memória do que a fisicamente disponível [DEI92, p. 215].

Este conceito é antigo: surgiu em 1960 no computador Atlas, construído pela Universidade de Manchester (Inglaterra), embora sua utilização mais ampla só tenha acontecido muitos anos depois. Tanenbaum simplifica a definição do termo:

A idéia básica da memória virtual é que o tamanho combinado do programa, dados e pilha podem exceder a quantidade de memória física disponível para o mesmo [TAN92, p. 89].

Outra definição possível de memória virtual é a quantidade de memória excedente a memória física instalada em um sistema computacional que esta aparentemente em uso quando se consideram a soma das quantidades totais de memória utilizadas por todos os processos existentes num dado momento dentro deste sistema. Na Figura 4.15 temos uma representação da memória real e virtual.



Figura 4.15: Representação da memória real e virtual

Por sua vez, Silberschatz e Galvin propõem as seguintes definições:

Memória Virtual é uma técnica que permite a execução de processos que podem não estar completamente na memória [SG94, p. 301].

Memória Virtual é a separação da memória lógica vista pelo usuário da memória física [SG94, p. 302].

De qualquer forma, o termo **memória virtual** indica que o sistema computacional possui a capacidade de oferecer mais memória do que a fisicamente instalada, ou seja, é capaz de disponibilizar uma quantidade *aparente* de memória maior do que a memória *de fato* (real) existente do sistema.

Os maiores benefícios obtidos através da utilização de sistemas que empregam mecanismos de memória virtual são:

- Percepção por parte de programadores e usuários de que a quantidade de memória potencialmente disponível é maior do que a realmente existente no sistema.
- Abstração de que a memória é um vetor unidimensional, contínuo, dotado de endereçamento linear iniciado na posição zero.
- Maior eficiência do sistema devido à presença de um número maior de processos, permitindo uso equilibrado e sustentado dos recursos disponíveis.

A memória física tem seu tamanho usualmente limitado pela arquitetura do processador ou do sistema, ou seja, possui um tamanho máximo que é fixo e conhecido. Já a memória virtual tem seu tamanho limitado, freqüentemente, pela quantidade de espaço livre existente nas unidades de disco do sistema possuindo, portanto, um valor variável e mais flexível (pois é mais fácil acrescentar novas unidades de disco a um sistema ou substituir as unidades do sistema por outras de maior capacidade).

Do ponto de vista de velocidade, a velocidade de acesso da memória física é substancialmente maior do que da memória virtual, mas a velocidade da memória total do sistema tende a ser uma média ponderada das velocidades de acesso da memória física e virtual cujos pesos são as quantidades envolvidas. De qualquer modo, a velocidade média de acesso a memória do sistema torna-se um valor intermediário entre as velocidades de acesso da memória física e virtual sendo que quanto maior a quantidade de memória virtual utilizada menor será a velocidade média de acesso a memória.

$$VelMemFisica > VelMemTotal > VelMemVirtual$$

A memória virtual pode ser implementada basicamente através de mecanismos de:

- **Paginação**

Técnica em que o espaço de endereçamento virtual é dividido em blocos, denominados unidades de alocação, de tamanho e posição fixas,

geralmente de pequeno tamanho, os quais se associa um número. O sistema operacional efetua um mapeamento das unidades de alocação em endereços de memória, determinando também quais estão presentes na memória física e quais estão nos arquivos de troca.

- **Segmentação**

Técnica em que o espaço de endereçamento virtual é dividido em blocos de tamanho fixo ou variável, definidos por um início e um tamanho, cuja posição também pode ser fixa ou variável, mas identificados univocamente. O sistema operacional mapeia estes blocos em endereços de memória, efetuando um controle de quais blocos estão presentes na memória física e quais estão nos arquivos de troca.

Atualmente, os mecanismos mais populares de implementação de memória virtual são através da paginação. A segmentação é uma alternativa menos utilizada, embora mais adequada do ponto de vista de programação, de forma que em alguns poucos sistemas se usam ambas as técnicas.

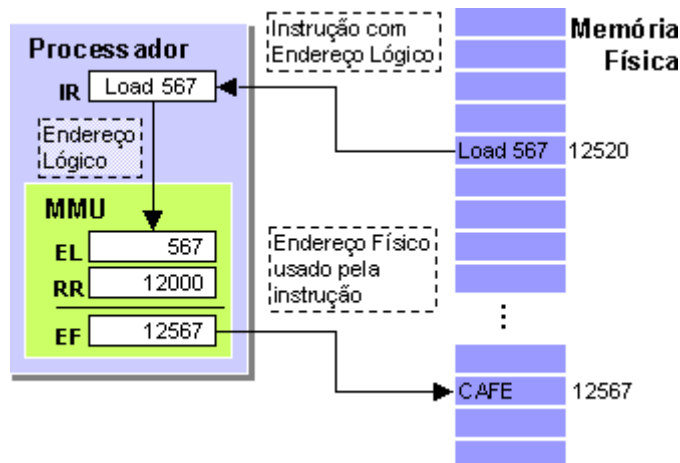


Figura 4.16: MMU e Relocação dinâmica

Estas duas técnicas só podem ser implementadas se for possível a desassociação dos endereços referenciados pelos processos em execução dos efetivamente utilizados na memória física do sistema. Isto equivale a dizer que o *binding* deve se completar no momento da execução de cada instrução, permitindo adiar até o último momento o mapeamento do espaço lógico de um programa em seu espaço físico definitivo de execução, isto é o que chamamos de relocação dinâmica. Para isto o processador deve dispor de mecanismos de deslocamento dos endereços referenciados pelo programa para as regiões de memória que efetivamente serão usadas. É óbvio que tal tarefa só pode ser completada com um sofisticado mecanismo de endereçamento de memória, mantido pelo sistema operacional. Tais mecanismos são geralmente imple-

mentados como uma **unidade de gerenciamento de memória** ou *memory management unit* (MMU) esquematizada na Figura 53.

Além dos mecanismos de paginação ou segmentação, a memória virtual exige a disponibilização de espaço nos dispositivos de armazenamento secundário para a criação de um (ou mais) arquivos de troca, os *swap files*. Em função da velocidade dos dispositivos de E/S, as unidades de disco são quase sempre utilizadas, minimizando o impacto das transferências entre memória primária (memória física do sistema) e memória secundária (unidades de disco).

A memória total de um sistema é, portanto, a soma de sua memória física (de tamanho fixo) com a memória virtual do sistema. O tamanho da memória virtual do sistema é definida por, basicamente, o menor valor dentre os seguintes:

- capacidade de endereçamento do processador,
- capacidade de administração de endereços do sistema operacional e
- capacidade de armazenamento dos dispositivos de armazenamento secundário (unidades de disco).

Nos sistemas Win32 (Windows 95/98 e Windows NT) são oferecidas funções específicas para o gerenciamento de memória virtual. Suas API (*Application Program Interface*) oferecem, dentre outras, as importantes funções relacionadas na Tabela 4.3 [CAL96, p. 252-262].

Tabela 4.3: Funções de gerenciamento de memória virtual da API Win32

Função	Utilização
VirtualAlloc	Permite reservar ou alocar memória para um programa.
VirtualFree	Libera memória alocada ou reservada através de VirtualAlloc.
GetProcessHeap	Obtêm um <i>handler</i> para o <i>heap</i> atual.
CreateHeap	Cria um novo <i>heap</i> .
HeapAlloc	Efetua uma alocação parcial no <i>heap</i> .
HeapReAlloc	Modifica o tamanho de uma alocação parcial do <i>heap</i> .
HeapSize	Retorna o tamanho corrente do <i>heap</i> .
HeapFree	Libera uma alocação parcial do <i>heap</i> .
HeapDestroy	Destroy a área de <i>heap</i> .
GlobalMemoryStatus	Retorna informações sobre utilização da memória do sistema.

Através destas funções o usuário pode administrar o uso da memória virtual do sistema, determinando a utilização da memória física, tamanho

e utilização do arquivo de troca, tamanho do *heap* etc. Além disso pode efetuar a alocação de novas áreas de memória para sua aplicação, o que permite a criação de um mecanismo particular de utilização e controle do espaço de endereçamento virtual que opera de forma transparente com relação aos mecanismos de memória virtual implementados pelo sistema operacional.

Nos Exemplos 4.4 e 4.5 temos exemplos de utilização de algumas destas funções. O Exemplo 4.4 utiliza a função da API `GlobalMemoryStatus` para determinar o tamanho da memória física instalada, a quantidade de memória física disponível, o tamanho do arquivo de troca e sua utilização.

```
{ Para Borland Delphi 2.0 ou superior. }
procedure TForm1.UpdateMemStatus;
var
    Status: TMemoryStatus;
function ToKb(Value: DWORD): DWORD;
begin
    result := Value div 1024;
end;

begin
    { Obtêm status da memória }
    Status.dwLength := sizeof(TMemoryStatus);
    GlobalMemoryStatus(Status);
    with Status do { Atualiza labels e gauges }
    begin
        Label1.Caption:=IntToStr(ToKb(dwTotalPhys))+ ' Kb';
        Label2.Caption:=IntToStr(ToKb(dwTotalPhys -
            dwAvailPhys))+ ' Kb';
        Label3.Caption:=IntToStr(ToKb(dwTotalPageFile))+ ' Kb';
        Label4.Caption:=IntToStr(ToKb(dwTotalPageFile -
            dwAvailPageFile))+ ' Kb';
        Gauge1.MaxValue:=dwTotalPhys;
        Gauge1.Progress:=dwTotalPhys - dwAvailPhys;
        Gauge2.MaxValue:=dwTotalPageFile;
        Gauge2.Progress:=dwTotalPageFile - dwAvailPageFile;
    end;
end;
```

Exemplo 4.4 Uso de `GlobalMemoryStatus`

Já no Exemplos 4.5 que aloca um bloco de memória de tamanho *Size*, permitindo seu uso através de um ponteiro para a área alocada, efetuando sua liberação após o uso. A rotina possui um tratamento mínimo de erros.

```
{ Para Borland Delphi 2.0 ou superior. }
P := VirtualAlloc(nil, Size,
                 memCommit or mem.Reserve,
                 Page.ReadWrite);
if P = nil then
    ShowMessage("Alocação não foi possível")
else
begin
    { Uso da área alocada através do ponteiro P. }
    :
    { Liberação da área alocada após uso. }
    if not VirtualFree(P, 0, mem.Release) then
        ShowMessage("Erro liberando memória.");
end;
```

Exemplo 4.5 Alocação de bloco de memória com `VirtualAlloc`

4.7 Modelos de gerenciamento de memória

Como ilustrado na Figura 44, existem vários diferentes modelos para a organização e o gerenciamento de memória os quais trataremos brevemente:

- Monoprogramado com armazenamento real
- Multiprogramado com partições fixas sem armazenamento virtual
- Multiprogramado com partições variáveis sem armazenamento virtual
- Multiprogramado com armazenamento virtual através de paginação
- Multiprogramado com armazenamento virtual através de segmentação
- Multiprogramado com armazenamento virtual através de paginação e segmentação combinadas

4.7.1 Monoprogramado com armazenamento real

Neste modelo de gerenciamento a memória é dividida em duas partições distintas, de tamanhos diferentes, onde uma é utilizada pelo sistema operacional e a outra é utilizada pelo processo do usuário conforme ilustrado na Figura 4.17. Este modelo, também chamado de modelo de alocação contínua, armazenamento direto ou monoprogramado com armazenamento real, era a forma mais comum de gerenciamento de memória até meados da década de 1960. Também era a técnica mais comum usada pelos sistemas operacionais das primeiras gerações de microcomputadores.

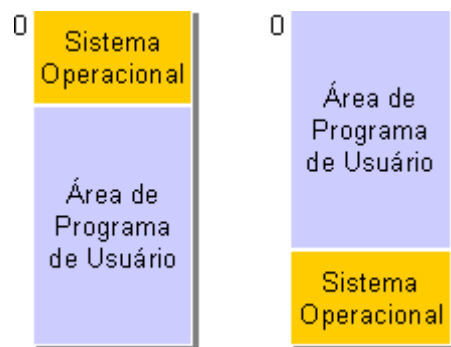


Figura 4.17: Organização da memória em modo monoprogramado real

Esta forma de gerenciamento de memória é bastante simples e permite que apenas um processo seja executado de cada vez, o que limita a programação a construção de programas estritamente seqüenciais. Na prática este esquema de gerenciamento só está preparado para a execução de um programa de cada vez, sendo que raramente a memória será inteiramente utilizada, sendo freqüente a existência de uma área livre ao final da área de programa destinada ao usuário.

Dado que o espaço de endereçamento corresponde a quantidade de memória primária fisicamente instalada no sistema, que não são utilizados mecanismos de memória virtual e que usualmente apenas um processo (programa) era executado de cada vez, este modelo de organização também é conhecido como organização monoprogramada real.

Como exemplo o PC-DOS/MS-DOS (*Disk Operating System*), sistema operacionais dos microcomputadores IBM e seus compatíveis, utiliza um esquema semelhante, onde o sistema operacional ficava residente na primeira parte da memória e a área de programa destinada aos usuários utilizava o espaço restante dos 640 Kbytes de espaço de endereçamento disponíveis. O CP/M (*Control Program/Monitor*), dos microcomputadores Apple e compatíveis utilizava esquema semelhante. No caso do DOS, vários outros esquemas adicionais foram criados para estender as capacidades básicas (e bastante limitadas) de endereçamento do sistema operacional, entre elas os mecanismos de extensão de memória e os *overlays*.

Os *overlays* (do termo recobrimento), são o resultado da estruturação dos procedimentos de um programa em forma de árvore, onde no topo estão os procedimentos mais usados e nos extremos os menos utilizados. Esta estruturação deve ser feita pelo usuário, satisfazendo as restrições do programa a ser desenvolvido e da memória disponível no sistema. Uma biblioteca de controle dos *overlays*, que funcionava como um sistema de gerenciamento de memória virtual, deve ser adicionada ao programa e mantida na memória todo o tempo, procura manter apenas os procedimentos de uma seção vertical da árvore, minimizando a quantidade necessária de memória física e

assim superando as limitações do DOS [GUI86, p. 184].

4.7.2 Particionamento fixo

Dada as vantagens dos sistemas multiprogramados sobre os monoprogramados, é necessário que a memória seja dividida de forma tal a possibilitar a presença de vários processos simultaneamente. A maneira mais simples de realizar-se esta tarefa é efetuar a divisão da memória primária do sistema em grandes blocos os quais são denominados partições. As partições, embora de tamanho fixo, não são necessariamente iguais, possibilitando diferentes configurações para sua utilização, como ilustrado na Figura 4.18.

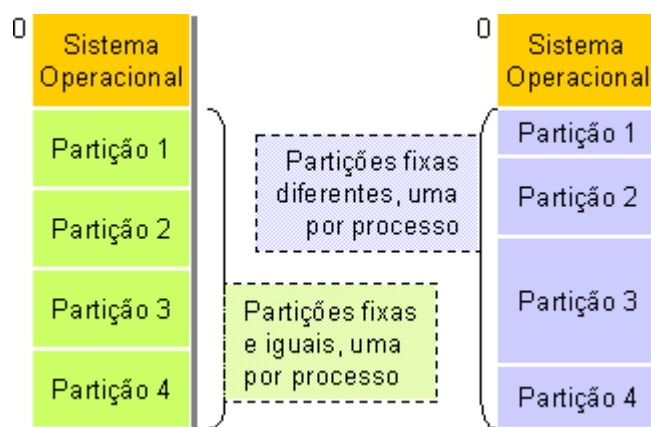


Figura 4.18: Organização da memória em modo multiprogramado com partições fixas

Enquanto o sistema operacional utiliza permanentemente uma destas partições, usualmente a primeira ou a última, os processos dos usuários podem ocupar as demais partições, cujo número dependerá do tamanho total da memória do sistema e dos tamanhos das partições realizadas.

Geralmente as partições eram determinadas através da configuração do sistema operacional, o que poderia ser feito de tempos em tempos ou até mesmo diariamente pelo operador do sistema. Até uma nova definição dos tamanhos das partições, os tamanhos e posições anteriormente definidos eram fixos.

Os processos poderão então ocupar as partições de memória a partir de uma fila única de processos que encaminhará o processo para a partição disponível. Tanto o modelo de endereçamento absoluto como relocável podem ser utilizados pois:

- Nos sistemas batch os programas eram compilados no instante da execução possibilitando o uso de compiladores absolutos, dado que

a posição que o programa utilizaria na memória (partição) era conhecida;

- Se utilizado compiladores relocáveis, um carregador relocável poderia transpor o código corretamente para a partição escolhida.

Quando do uso de partições iguais, uma única fila de processos poderia atender a contento a tarefa de definir qual processo ocuparia uma certa partição, embora ocorresse perda significativa de memória pela não utilização integral das partições. O uso de partições fixas de diferentes tamanhos permitia fazer melhor uso da memória, pois nesta situação poderiam ser utilizadas filas diferentes de processos para cada partição, baseadas no tamanho do processo/partição. Ainda assim poderíamos ter uma situação de partições livres e uma, em especial, com uma fila de processos. A melhor solução encontrada foi adotar uma única fila de processos e critérios de elegibilidade para designação de partições para processos visando bom uso da memória e um *throughput* adequado.

Torna-se evidente que a determinação da partição para a execução de um dado processo influencia no desempenho do sistema. Para esta tarefa podemos utilizar um dos seguintes critérios, que correspondem a estratégias de posicionamento (*placement strategies*):

- *First fit*: Aloca-se o processo para a primeira partição encontrada que comporte o processo, minimizando o trabalho de procura.
- *Best fit*: O processo é alocado para a menor partição que o comporte, produzindo o menor desperdício de áreas de memória, exige pesquisa em todas as partições livres.
- *Worst fit*: O processo é alocado para a maior partição que o comporte, produzindo o maior desperdício de áreas de memória, exige pesquisa em todas as partições livres.

Langsam *et al.* [LAT96, p. 625] sugerem alguns algoritmos em linguagem C para a alocação de blocos de memória utilizando o *first fit* e *best fit*, bem como para seleção da melhor partição a ser liberada.

De qualquer forma, o espaço de endereçamento corresponde ao tamanho da memória primária do sistema, ou seja, a soma dos tamanhos das partições e, portanto, do tamanho máximo dos processos em execução, é igual a memória física instalada no sistema. Assim, o particionamento fixo é um esquema de organização de memória que não utiliza memória virtual.

Vários sistemas comerciais de grande porte utilizavam este esquema de gerenciamento de memória, onde o operador ou o administrador do sistema definia o número e o tamanho das partições da memória principal.

4.7.3 Particionamento variável

O **particionamento variável** é bastante semelhante à organização de memória em partições fixas, exceto pelo fato de que agora é o sistema operacional efetua o particionamento da memória. A cada novo processo, a memória é dividida, de forma que partições de diferentes tamanhos sejam posicionadas na memória do sistema. A medida que os processos sejam finalizados, suas partições tornam-se livres, podendo ser ocupadas no todo ou em parte por novos processos como esquematizado na Figura 4.19. Este esquema de organização de memória também é denominado de particionamento por demanda.

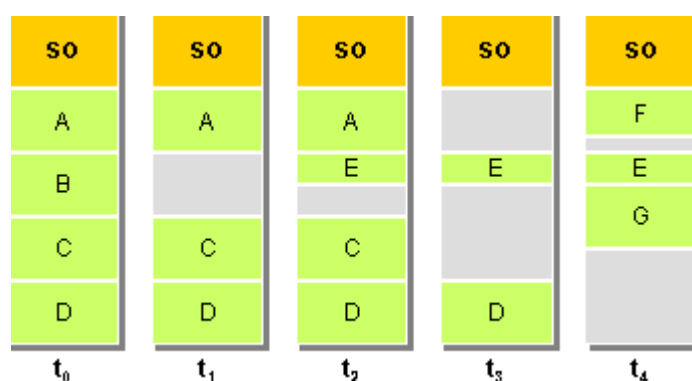


Figura 4.19: Organização da memória em modo multiprogramado com partições variáveis

Neste tipo de sistema, a estratégia de posicionamento *worst fit* é bastante útil pois permite maximizar o tamanho das áreas livres (buracos) obtidas a cada alocação, aumentando as possibilidades de sucesso de transformação da área desocupada em uma nova partição livre para um novo processo.

Mesmo utilizando-se o algoritmo *worst fit* ainda é possível que existam regiões livres de memória entre as partições efetivamente alocadas. Este fenômeno, que tende a aumentar conforme a utilização do sistema e número de processos presentes na memória, é denominado fragmentação interna. Desta forma, uma certa porção da memória total do sistema pode continuar permanecendo sem uso, anulando alguns dos benefícios do particionamento variável e do algoritmo *worst fit*.

Uma estratégia possível para eliminar a fragmentação interna é a da compactação de memória, onde todas as partições ocupadas são deslocadas em direção ao início da memória, de forma que todas as pequenas áreas livres componham uma única área livre maior no final da memória, como indicado na Figura 4.20.

A compactação de memória é uma técnica raramente utilizada devido ao alto consumo de processador para o deslocamento de todas as partições e

manutenção das estruturas de controle da memória. O trabalho despendido no reposicionamento de uma partição pode ser suficiente para finalizar o processo que a ocupa ou outro presente na memória, tornando a movimentação de partições um ônus para os processos em execução.

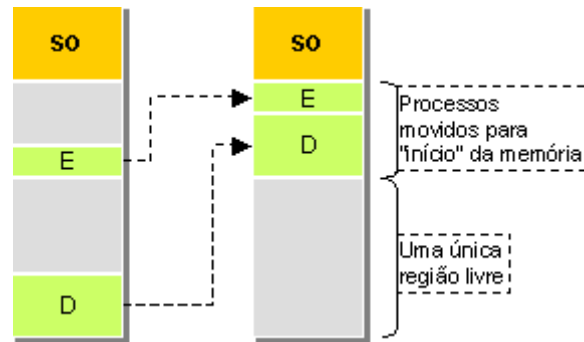


Figura 4.20: Compactação de memória

Da mesma forma que na organização da memória através de partições fixas, no particionamento variável o espaço de endereçamento é igual ao tamanho da memória primária existente no sistema e, portanto, um esquema de organização de memória que também não utiliza memória virtual.

4.7.4 Paginação

A **paginação** é um esquema de organização de memória que faz uso da memória virtual, ou seja, o espaço de endereçamento é maior que o tamanho da memória fisicamente presente no sistema, como representado na Figura 4.21.

O espaço de endereçamento total do sistema, denominado de espaço de endereçamento virtual é dividido em pequenos blocos de igual tamanho chamados **páginas virtuais** (*virtual pages*) ou apenas **páginas** (*pages*). Cada página é identificada por um número próprio. Da mesma forma a memória física é dividida em blocos iguais, do mesmo tamanho das páginas, denominados **molduras de páginas** (*page frames*). Cada moldura de página também é identificada por um número, sendo que para cada uma destas molduras de página corresponde uma certa região da memória física do sistema, como mostra a Figura 4.22.

Para que este esquema de divisão seja útil, o sistema operacional deve realizar um mapeamento de forma a identificar quais páginas estão presentes na memória física, isto é, deve determinar quais os *page frames* que estão ocupados e quais páginas virtuais (*virtual pages*) estão nele armazenados. O sistema operacional também deve controlar quais páginas virtuais estão localizadas nos arquivos de troca (*swap files*), que em conjunto com a memória física do sistema representam a memória virtual do sistema (vide

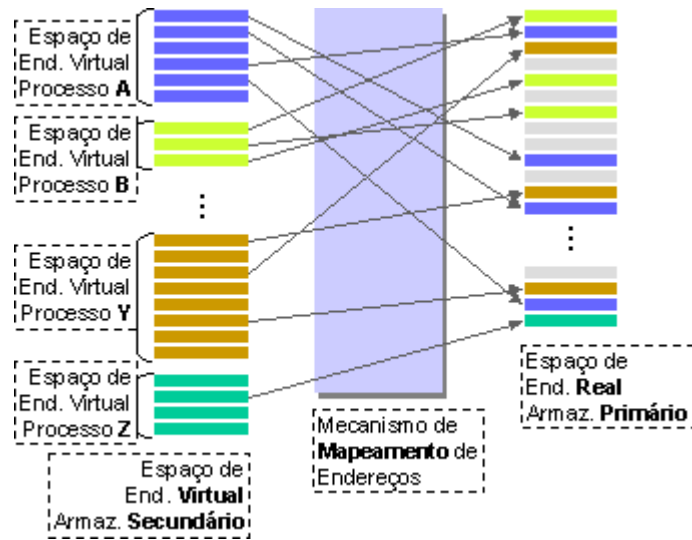


Figura 4.21: Espaços de endereçamento virtual e real na paginação

Figura 4.15).

A medida que os programas vão sendo executados, o sistema operacional vai relacionando quais páginas virtuais estão sendo alocadas para cada um destes programas, sem se preocupar com o posicionamento contíguo de partes de um mesmo programa. No instante efetivo da execução a MMU (*memory management unit*) converte os endereços virtuais em endereços físicos utilizando as tabelas de páginas, como esquematizado na Figura 61. Neste mesmo momento a MMU detecta se uma dada página está ou não presente na memória física, realizando uma operação de *page fault* (falta de página) caso não esteja presente.

Quando ocorre um *page fault* é acionada uma rotina do sistema operacional que busca na memória virtual (nos arquivos de troca) a página necessária, trazendo-a para a memória física. Esta operação é particularmente complexa quando já não existe espaço livre na memória física, sendo necessária a utilização de um algoritmo de troca de páginas para proceder-se a substituição de páginas.

Os *page faults* são um decorrência da existência de um mecanismo de memória virtual e embora sejam operações relativamente lentas quando comparadas ao processamento, propiciam grande flexibilidade ao sistema. É comum a implementação de mecanismos de contabilização dos *page faults* em sistemas de maior porte, onde pode existir até mesmo um limite, configurado pelo administrador do sistema, para a ocorrência de troca de páginas.

A conversão de endereços por parte da MMU é necessária porque cada programa *imagina* possuir um espaço de endereçamento linear originado no zero quando na verdade compartilha blocos isolados da memória física com

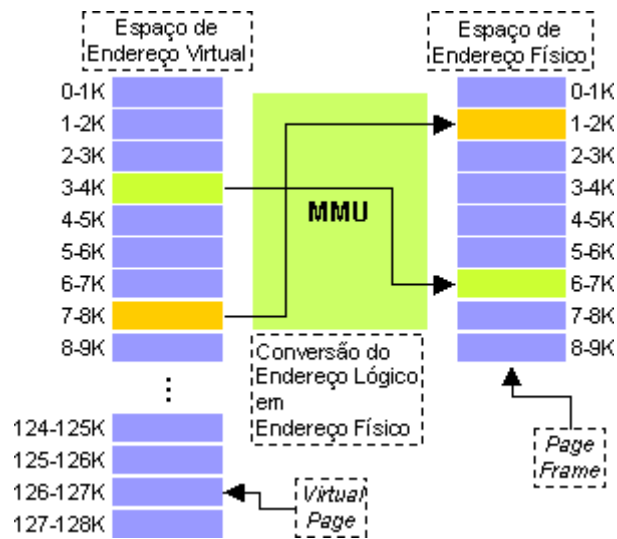


Figura 4.22: Endereçamento Virtual e Real na Paginação

outros programas que também estão em execução.

Outra implicação deste mecanismo é que os blocos fisicamente ocupados na memória principal não necessitam estar continuamente nem ordenadamente posicionados. Isto permite tanto a execução de um processo com apenas uma de suas páginas presente na memória física, como a execução de um processo cujo tamanho total é maior que o armazenamento primário do sistema. Sendo assim, a paginação é um esquema extremamente flexível.

O mapeamento das páginas virtuais nos efetivos endereços de memória é realizado pela MMU com o auxílio de **tabelas de páginas**, que determinam a relação entre as páginas do espaço de endereçamento virtual e as molduras de páginas do espaço de endereçamento físico, ou seja, oferecendo suporte para as operações de conversão de endereços necessárias ao uso deste esquema de organização de memória.

Num sistema de paginação pura, os endereços virtuais (veja a Figura 4.23) são denominados v , tomando a forma de pares ordenados (p, d) , onde p representa o número da página virtual e d a posição desejada, ou seja, o deslocamento (*displacement* ou *offset*) a partir da origem desta página.

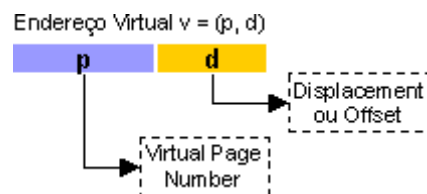


Figura 4.23: Formato do endereço virtual para sistema de paginação pura

Já as posições das molduras de páginas (*page frames*), isto é, seus endereços iniciais são determinados da seguinte forma: como as molduras de páginas possuem o mesmo tamanho das páginas virtuais, os endereços iniciais dos *page frames* são múltiplos integrais do tamanho das páginas, não podendo ser designadas de outra forma. A Tabela 4.4 exibe a relação entre as molduras de páginas e seu endereçamento na memória física.

Tabela 4.4: Endereçamento das Molduras de Páginas

Número do Page Frame	Tamanho do Page Frame	Endereçamento Real das Páginas
0	p	$0 : p - 1$
1	p	$p : 2p - 1$
2	p	$2p : 3p - 1$
3	p	$3p : 4p - 1$
4	p	$4p : 5p - 1$
\vdots	\vdots	\vdots
$n - 1$	p	$(n - 1)p : np - 1$

O funcionamento da MMU, conforme esquematizado na Figura 4.24, pode ser descrito resumidamente nos passos relacionados abaixo:

1. MMU recebe o endereço virtual contido nas instruções do programa.
2. O número de página virtual é usado como índice na tabela de páginas.
3. Obtêm-se o endereço físico da moldura de página que contém o endereço virtual solicitado ou ocorre um *page fault*.
4. MMU compõe o endereço final usando o endereço da moldura de página e uma parte do endereço virtual (*displacement*).

Para o funcionamento apropriado da MMU é necessária a existência de tabelas de páginas, mantidas total ou parcialmente na memória primária pelo sistema operacional. Cada entrada da tabela de páginas contém, geralmente:

- um *bit* indicando presença ou ausência da página na memória principal;
- o número da moldura de página (*page frame number*); e
- dependendo da implementação, o endereço da página no armazenamento secundário (*swap files*) quando ausente da memória principal.

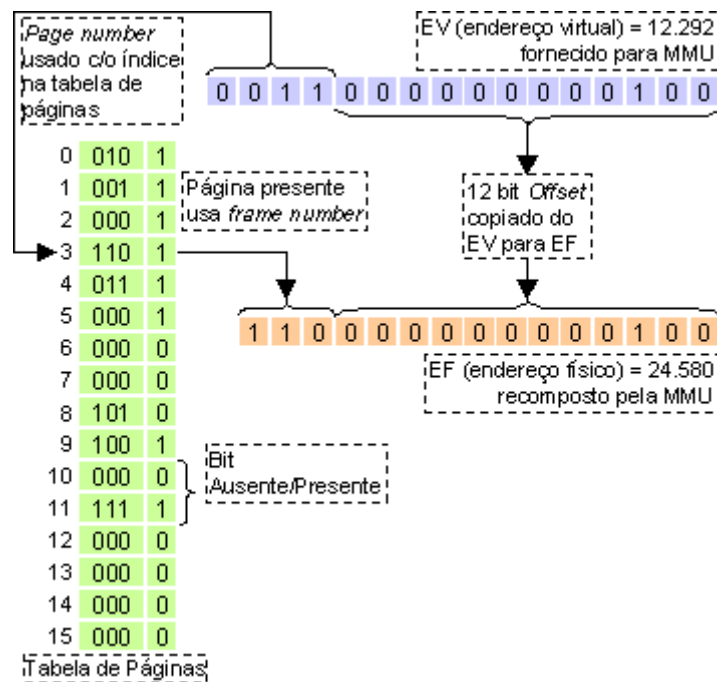


Figura 4.24: Conversão de endereços pela MMU

O mecanismo de conversão de endereços depende da organização das tabelas de páginas (um, dois ou múltiplos níveis) e da forma do mapeamento (mapeamento direto, mapeamento associativo e mapeamento combinado associativo/direto). Em função do tamanho do espaço de endereçamento virtual, do tamanho da página e do tamanho da memória real, os arranjos das tabelas de páginas podem se tornar grandes e complexos. Diversos estudos e estratégias já foram realizados para sugerir organizações mais eficientes para o mapeamento e a conversão de endereços.

Temos assim que a paginação permite a execução de programas individualmente maiores que a memória física do sistema ou, no caso mais comum, a execução de diversos programas cuja soma dos seus tamanhos exceda o tamanho da memória física. Graças a MMU, implementada no *hardware* do processador, as aplicações podem ser desenvolvidas imaginando um espaço de endereçamento linear, contínuo e de grande tamanho, simplificando bastante o trabalho de programação. A paginação é o sistema de organização de memória mais utilizado atualmente.

Exemplos de sistemas computacionais que utilizam a paginação pura são:

- DEC PDP-11, minicomputador de 16 *bits* popular da década de 1970, contando com um espaço de endereçamento virtual de 16 *bits*, páginas de 8 KBytes e até 4 MBytes de memória física, utilizando tabelas de páginas de um único nível [TAN92, p. 97].

- DEC VAX (Virtual Addresses eXtensions), sucessor do DEC PDP-11, minicomputador de 32 *bits*, possuindo um espaço de endereçamento virtual de 32 *bits* e pequenas páginas de 512 bytes. Os modelos de sua família contavam com no mínimo 2 MBytes de memória física até 512 MBytes. As tabelas de páginas possuíam dois níveis [TAN92, p. 98].
- IBM OS/2 2.0 (Operating System/2), operando em plataforma Intel 80386 ou 80486, oferecia até 512 MBytes de espaço lógico linear por processo num esquema de endereçamento de 32 *bits*, tamanho de página de 4 KBytes com paginação por demanda [IBM92b, p. 11].
- IBM AS/400, minicomputador de 64 *bits* que utiliza um esquema de tabela de páginas invertidas (*inverted page table*) [STA96, p. 248].
- Microsoft Windows 95, dirigido para processadores 80386 ou superior, oferece espaço de endereçamento linear virtual de 2 GBytes (endereços de 32 *bits*), páginas de 4 KBytes [PET96, p. 293].

4.7.5 Segmentação

Enquanto que a organização da memória através da paginação é um modelo puramente unidimensional, isto é, o espaço de endereçamento virtual oferecido a cada um dos diversos processos é único e linear, a **segmentação** propõe um modelo bidimensional, onde cada processo pode utilizar-se de diversos espaços de endereçamento virtuais independentes. Este conceito foi introduzido nos sistemas Burroughs e Multics [GUI86, p. 137].

Num esquema de memória segmentada, o espaço de endereçamento virtual é dividido em blocos de tamanho variável, onde cada bloco pode assumir também um posicionamento variável, isto é, para um dado processo, enquanto cada segmento deve ocupar um espaço de endereçamento contínuo na memória física, não existe necessidade dos diversos segmentos deste processo estarem alocados de forma contígua ou sequer ordenada. Estes blocos são denominados **segmentos de memória** ou simplesmente **segmentos**, como ilustrado na Figura 4.25.

É comum que os segmentos possuam um tamanho máximo, limitado ao tamanho da memória física do sistema e um número máximo de segmentos distintos. Cada segmento representa um espaço de endereçamento linear independente dos demais segmentos, isto permite que cada segmento possa crescer ou diminuir conforme suas necessidades e livremente de outros segmentos. Uma situação possível e comum é a de um processo que possui um segmento de código (o programa em si), um ou mais segmentos de dados e um segmento para sua pilha (*stack*), todos com diferentes tamanhos.

Dado que um segmento é uma unidade lógica, o programador deve explicitamente determinar sua utilização. Embora seja possível ter-se código,

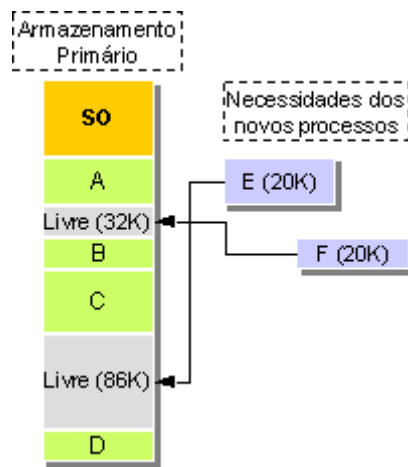


Figura 4.25: Armazenamento primário na segmentação

dados e pilha num único segmento, isto representa uma má utilização desta estrutura de organização da memória, invalidando seus benefícios.

A organização da memória em segmentos favorece e simplifica a organização de estruturas de dados, principalmente as de tamanho variável em tempo de execução. Além disto oferece importantes facilidades do ponto de vista de compartilhamento e proteção. Por exemplo, uma biblioteca de funções pode ser colocada num segmento e compartilhada pelas diversas aplicações que as necessitem. A cada segmento podem ser associados tipos de acesso que especifiquem as operações que podem ser executadas no segmento, tais como leitura (*read*), escrita (*write*), execução (*execute*) e anexação (*append*). Tais operações podem ainda ser associadas a modos de acesso específicos, criando um amplo conjunto de possibilidades úteis para implantação de esquemas de segurança [DEI92, p. 233].

Num sistema de segmentação pura, os endereços virtuais, cuja estrutura se indica na Figura 4.26, são denominados v e tomam a forma de pares ordenados (s, d) , onde s representa o número do segmento e d a posição desejada, ou seja, o deslocamento (*displacement* ou *offset*) a partir da origem deste segmento. Notamos que a formação dos endereços na segmentação é semelhante a existente na paginação.

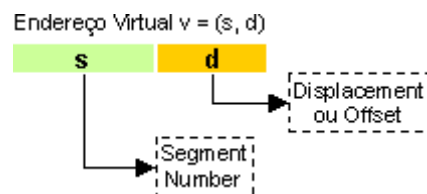


Figura 4.26: Formato do endereço virtual para sistema de segmentação

Um processo somente pode ser executado se ao menos um de seus segmentos contendo código estiver presente na memória física. Para isto segmentos devem ser transferidos da memória secundária para a memória primária da mesma forma que as páginas no esquema de paginação, ou seja, cada segmento deve ser transferido inteiramente e posicionado numa região contínua de memória.

Isto indica que os *segment faults* são operações mais lentas, dado que os segmentos são usualmente maiores do que as páginas, e também menos freqüentes, pois o número de segmentos de um processo é tipicamente menor que o número de páginas equivalente. Outro ponto é que deve ser determinada qual região de memória permite a colocação do novo segmento, operação que pode ser realizada através de algoritmos que apliquem as estratégias de posicionamento (*placement strategies*). O segmento que deve ser substituído, em caso de um *segment fault*, deve ser obtido através de algoritmos que implementem as estratégias de substituição (*replacement strategies*).

O mapeamento dos endereços virtuais em endereços reais pertencentes aos segmentos corretos se faz de maneira idêntica à paginação, ou seja, utiliza um esquema de mapeamento e tabelas de mapeamento de segmentos (*segment map tables*):

1. MMU recebe o endereço virtual contido nas instruções do programa.
2. O número de segmento virtual é usado como índice na tabela de segmentos.
3. Obtêm-se o endereço físico de início do segmento ou ocorre um *segment fault*.
4. MMU compõe o endereço final usando o endereço de início do segmento e uma parte do endereço virtual (*displacement*).

O mecanismo de conversão de endereços depende da organização das tabelas de segmentos e da forma do mapeamento (mapeamento direto ou mapeamento associativo).

Exemplos de sistemas computacionais que utilizaram a segmentação pura são:

- Burroughs B6700, computador do início da década de 60, com arquitetura tipo pilha [GUI86, p.157].
- HP 3000, minicomputador tipo pilha cujo espaço lógico consistia de até 255 segmentos de código executável de 16 KBytes cada e um único segmento de dados de 64 KBytes manipulado por *hardware* [GUI86, p.172].

- Intel 8086/8088, microprocessador de 8 bits, oferecia um espaço de endereçamento lógico de 1 MByte, podendo efetuar o endereçamento físico de no máximo 64 KBytes, tamanho máximo dos segmentos que administrava [BOR92, p. 342].
- IBM OS/2 1.x (Operating System/2), voltado para o microprocessador Intel 80286, utilizava segmentação pura, onde o tamanho máximo dos segmentos era 64 Kbytes, espaço de endereçamento virtual de 512 MBytes por aplicação e memória física máxima de 16 MBytes [IBM92b, p. 11][LET89, p. 142].
- Microsoft Windows 3.x, também dirigido para o microprocessador Intel 80286, usava segmentação pura, segmentos de no máximo 64 KBytes, 48MBytes de espaço de endereçamento virtual e memória física máxima de 16 MBytes [IBM92b, p. 14].

Apesar de ser um esquema de organização de memória que oferece uma série de vantagens, a segmentação apresenta uma grande desvantagem: conforme os segmentos se tornam grandes, as operações de posicionamento e substituição tendem a se tornar lentas conduzindo o sistema a uma situação de ineficiência. Existe ainda o problema maior de um segmento individual se tornar maior que a memória física disponível no sistema. A solução desta desvantagem se dá na utilização conjunta dos esquemas de segmentação e paginação, como veremos mais a frente.

4.7.6 Paginação *versus* Segmentação

Como visto, os esquemas de organização de memória através de paginação e segmentação possuem vantagens e desvantagens. Na Tabela 4.5 temos um quadro comparativo, tal como proposto por Deitel [DEI92, p. 131], onde se avaliam estas formas de organização do armazenamento primário.

Podemos notar que a paginação é um esquema de organização de memória mais simples, principalmente para o programador, enquanto que a segmentação, a custo de uma maior complexidade, oferece mecanismos mais sofisticados para organização e compartilhamento de dados ou procedimentos. A razão para isto se encontra no porque destes esquemas terem sido inventados.

Enquanto a paginação foi desenvolvida para ser um esquema de organização invisível ao programador, proporcionando um grande espaço de endereçamento linear, maior que a memória física e de uso simples, o propósito da segmentação foi permitir que programas e dados pudessem ser logicamente divididos em espaços de endereçamento independentes facilitando o compartilhamento e proteção [STA96, p. 249].

Enquanto o grande inconveniente da paginação pura é sua excessiva simplicidade como modelo de programação, a segmentação pura impõe dificul-

Tabela 4.5: Quadro comparativo paginação *versus* segmentação

Consideração	Pag.	Seg.
Programador precisa saber que esta técnica é utilizada?	Não	Sim
Quantos espaços de endereçamento linear existem?	1	Muitos
O espaço de endereçamento virtual pode exceder o tamanho da memória física?	Sim	Sim
Dados e procedimentos podem ser distinguidos?	Não	Sim
Tabelas cujo tamanho é variável podem ser acomodadas facilmente?	Não	Sim
O compartilhamento de procedimentos ou dados entre usuários é facilitado?	Não	Sim

dades no gerenciamento da memória virtual, pois a troca de segmentos entre o armazenamento primário e secundário se torna lento para segmentos de grande tamanho, penalizando o desempenho do sistema.

4.7.7 Paginação e segmentação combinadas

De forma que possam ser obtidas as maiores vantagens dos esquemas de paginação e segmentação, desenvolveu-se o uso combinado destas duas técnicas em sistemas com esquemas híbridos de gerenciamento de memória, mais conhecidos como sistemas multiprogramados com paginação e segmentação combinadas.

A paginação proporciona grande espaço de endereçamento linear e facilidade para o desenvolvimento embora não ofereça mecanismos mais sofisticados de organização de código e dados bem como de compartilhamento, segurança e proteção. Por sua vez, a segmentação oferece tais mecanismos de organização, compartilhamento e proteção, mas deixa de ser conveniente quando os segmentos tornam-se grandes além de impor um modelo de desenvolvimento de *software* um pouco mais complexo. Combinando-se paginação e segmentação de forma que os segmentos tornem-se paginados, associam-se as vantagens de cada um destes esquemas eliminando suas maiores deficiências as custas de uma organização um pouco mais complexa mas transparente para o desenvolvedor.

Num sistema com paginação/segmentação combinadas, os segmentos devem necessariamente ter tamanho múltiplo do tamanho das páginas, não mais necessitando ser armazenado inteiramente na memória e tão pouco de forma contígua e ordenada. Todos os benefícios da segmentação são mantidos, ou seja, os programas podem ser divididos em múltiplos espaços de endereçamento virtuais que, ao serem paginados, não necessitam de armazenamento contínuo na memória real. Se desejado, todo programa e dados podem ser concentrados num único segmento, fazendo que o resultado sejam

semelhante a um sistema paginado puro.

Desta forma, num sistema de paginação/segmentação combinadas, os endereços virtuais, como indicado na Figura 4.27, denominados v , tomam a forma de triplas ordenadas (s, p, d) , onde s representa o número do segmento, p representa o número da página virtual e d a posição desejada, ou seja, o deslocamento (*displacement* ou *offset*) a partir da origem da página indicada dentro deste segmento.

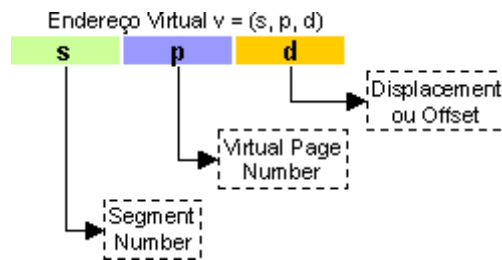


Figura 4.27: Formato do endereço virtual para sistema de paginação e segmentação combinadas

Notamos que o espaço de endereçamento virtual oferecido é tridimensional, tornando-se necessário a existência de uma estrutura de controle mais sofisticada nestes sistemas. Geralmente o sistema operacional mantém uma tabela de mapa de segmentos por processo, cuja indicação figura no PCB (*Process Control Block* abordado na seção 2.4.1), e uma tabela de páginas para cada segmento individual. Para se resolver um endereço virtual determinando-se o endereço real torna-se necessária a utilização de informações em três tabelas diferentes.

O funcionamento da MMU nestes sistemas, se encontra esquematizado na Figura 4.28 e pode ser descrito, resumidamente, como segue:

1. MMU recebe o endereço virtual contido nas instruções do programa.
2. A partir da tabela de controle dos processos (tabela de PCB), é selecionada a tabela de mapa de segmentos pertencente ao processo.
3. O número de segmento virtual é usado como índice na tabela de segmentos obtendo-se o número de página virtual.
4. É utilizada a tabela de páginas relativa ao segmento em uso.
5. O número de página virtual é usado como índice na tabela de páginas.
6. Obtêm-se o endereço físico da moldura de página que contém o endereço virtual solicitado ou ocorre um *page fault*.
7. MMU compõe o endereço final usando o endereço da moldura de página e uma parte do endereço virtual (*displacement*).

A manutenção desta estrutura complexa requer cuidadoso projeto para que não consuma recursos excessivos e processamento significativo nos sistemas que as utilizam.

Exemplos de sistemas computacionais que utilizam a paginação e segmentação combinadas são:

- Honeywell 6000, computadores das décadas de 1960 e 1970, operando com sistema operacional MULTICS suportando processos com até 2^{18} (262.144) segmentos cada um com até 64 KBytes de tamanho [TAN92, p. 132].
- IBM System/360, computador do final da década de 1960, com espaço lógico de 16 MBytes divididos em 16 segmentos de 1 MByte [GUI86, p.154].
- IBM MVS (*Multiple Virtual Storage System*), operando na arquitetura ESA/370, provê cada processo com até 2 GBytes de, nos quais poderiam existir 2048 segmentos de 256 páginas de 4096 bytes [DEI92, p. 677].
- Família Intel P6, suportando até 64 TBytes de endereçamento virtual e um máximo de 4 GBytes de memória física, oferecendo até 8192 segmentos de até 4 GBytes cada um, compostos de páginas de 4096 bytes [STA96, p. 252].

4.7.8 Tabelas de páginas

Como visto, tanto a organização de memória através de paginação como de segmentação e os sistemas híbridos que utilizam a paginação combinada com segmentação, são implementadas tabelas para realizar a conversão de endereços virtuais em endereços físicos. Estas tabelas, suportadas diretamente pelo *hardware* do sistema e mantidas pelo sistema operacional são, juntamente com os mecanismos de conversão de endereços, o ponto central destes esquemas de organização de memória.

A idéia básica é que o endereço virtual é composto de duas partes, um número da página virtual e um deslocamento dentro da página. O número da página virtual é usado como índice numa tabela de páginas, ou seja, é somado ao endereço de base da tabela de páginas, mantido num registrador qualquer do processador, obtendo-se uma referência para uma entrada da tabela que contém o endereço real da moldura de página desejada. Somando-se o deslocamento contido no endereço virtual ao endereço da moldura de página obtido da tabela de páginas obtêm-se o endereço real completo.

Na Figura 4.29 temos uma ilustração que esquematiza as operações realizadas na conversão de um endereço virtual para um outro real. Este esquema de conversão de endereços é denominado conversão ou tradução

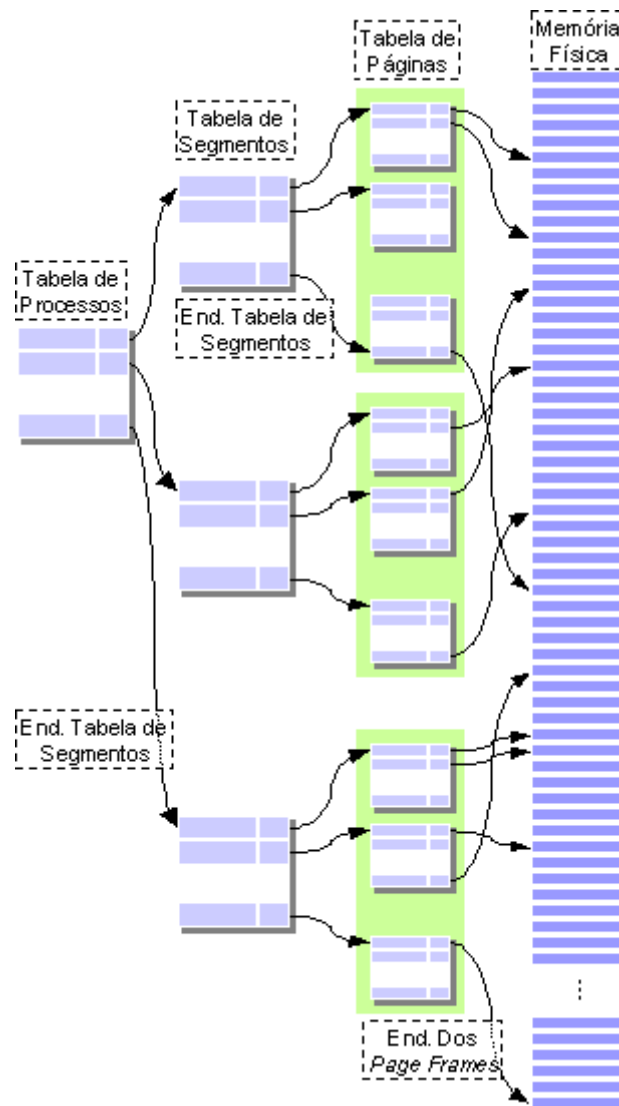


Figura 4.28: Estrutura de tabelas para sistemas com paginação e segmentação combinadas

de endereços por mapeamento direto, ou ainda, paginação com um nível de tabelas.

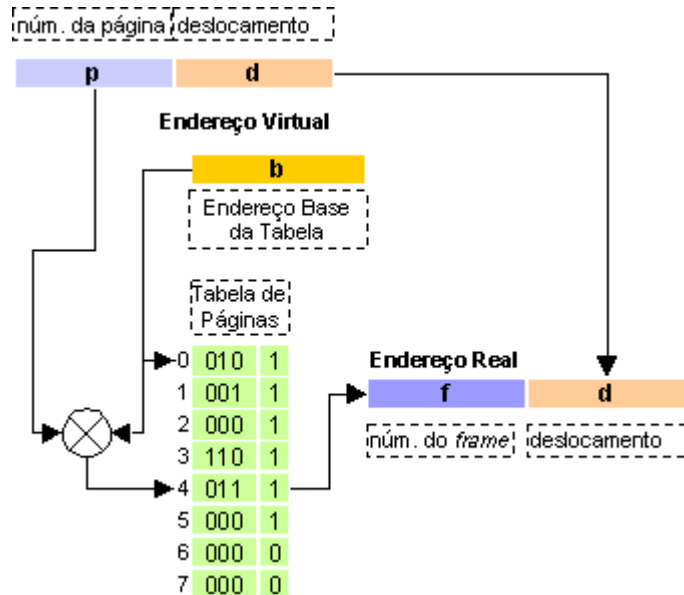


Figura 4.29: Conversão de endereços por mapeamento direto

Embora de relativa simplicidade e eficiência, o mapeamento indireto pode apresentar dois problemas importantes a medida que o espaço de endereçamento virtual se torna relativamente muito maior que a memória física disponível ou possível de ser implementada no sistema [TAN92, p. 93].

Os problemas identificados são:

1. A tabela de páginas pode se tornar extremamente grande. Grandes espaços virtuais de endereçamento requerem tabelas com muitas entradas. Se as tabelas são grandes, uma porção preciosa da memória pode ser consumida para este fim, reduzindo a memória disponível para os processos que podem ser executados pelo sistema.
2. Tabelas de páginas em memória versus troca de tabela de páginas. Como cada processo possui suas tabelas de páginas, ao esgotar-se o seu *quantum*, a sua execução é interrompida sendo substituída por outro processo. Se for realizada a troca das tabelas durante o chaveamento de processos, economiza-se memória primária embora tornando a operação de troca de contexto lenta. Se forem mantidas todas as tabelas de páginas em memória primária, a troca de contexto torna-se rápida, mas isto pode exaurir a memória primária do sistema.
3. O mapeamento pode se tornar lento para tabelas grandes ou complexos. Como cada referência a memória deve necessariamente ter

seu endereço convertido, quanto maiores ou mais complexas as tabelas, mais numerosas e complicadas serão as operações de conversão e, assim, será maior o *overhead* imposto pela conversão dos endereços, fazendo que o mapeamento se torne inconvenientemente lento, afetando de forma significativa a performance do sistema.

Para ilustrar esta situação, analisemos a seguinte situação: na arquitetura DEC VAX, cada processo pode possuir até 2 GBytes (2^{31} bytes) de espaço de endereçamento. Como as páginas deste sistema possuem apenas 512 bytes (2^9 bytes), então é necessário uma tabela de páginas contendo 2^{22} entradas para cada processo existente no sistema, ou seja, 4.194.304 entradas. Se uma tabela desta magnitude já é indesejável, que tal um sistema Pentium, que no modo segmentado/paginado oferece 64 TBytes (2^{46} bytes) de memória virtual? Com páginas de 4096 bytes (2^{12} bytes) e sabendo que metade do endereçamento virtual é oferecido individualmente para cada processo, uma tabela simples por processo deveria conter $\frac{2^{34}}{2}$ entradas, ou seja, 8.589.934.592 de entradas!

Uma primeira solução seria submeter a tabela de páginas à paginação, como qualquer outra área de memória, armazenando-a na memória virtual, fazendo que apenas uma parte dela esteja necessariamente presente na memória primária.

Outra solução para evitar a presença de enormes tabelas na memória, conforme indicado por Tanenbaum [TAN92, p. 94], é divisão destas numa estrutura de múltiplos níveis, como indicado na Figura 68. Um espaço virtual de 32 *bits* poderia ter seu endereço dividido em três partes: (1) um número de tabela de páginas TP_1 de 10 *bits*, (2) um número de página virtual TP_2 de 10 *bits* e (3) um deslocamento (*offset*) de 12 *bits*.

O valor TP_1 atua como índice para a tabela de páginas de primeiro nível, selecionando uma das tabelas do segundo nível. Na tabela de segundo nível selecionada, o valor TP_2 atua como outro índice, obtendo-se assim o endereço real da moldura de página (*page frame*). A MMU compõe o endereço real final a partir do endereço da moldura de página, obtido na tabela de segundo nível, e do deslocamento (*offset*), retirado do endereço virtual.

Com a divisão de uma única tabela (de primeiro nível) em uma tabela de entrada (de primeiro nível) e tabelas de páginas auxiliares (de segundo nível) passa a ser possível administrar-se a paginação das tabelas de páginas de maneira mais flexível. Como regra geral, cada tabela de páginas nunca é maior que o tamanho de uma página [STA96, p. 248].

De forma análoga, um sistema de tabela de páginas de dois níveis pode ser expandido para três, quatro ou mais nível, embora a flexibilidade adicionada torna-se menos valiosa do que a complexidade inerente ao maior número de níveis. Cada *hardware* específico possui uma estrutura de tabelas particular, que leva em conta as peculiaridades da implementação tais

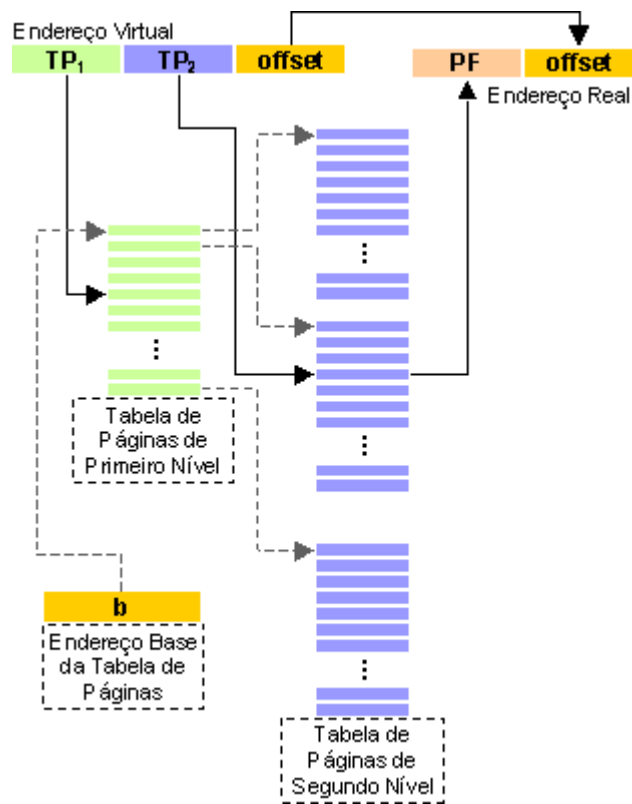


Figura 4.30: Estrutura multinível para tabelas de páginas

como o tamanho do espaço de endereçamento virtual, a máxima quantidade física de memória endereçável e o tamanho da página.

Independentemente do número de níveis e do layout das tabelas, as entradas típicas das tabelas de páginas possuem vários campos utilizados para o controle adequado da memória: o número da moldura de página (*page frame number*) que indica o endereço real da página, o *bit* presente/ausente (*present/absent*) que sinaliza se a página está ou não na memória primária, os *bits* de proteção (*protection*) que especificam as operações que podem ser realizadas na página, um *bit* de habilitação do cache (*caching disabled*) usado para evitar que a página seja colocada no cache e os *bits* de referência (*referenced*) e modificação (*modified*) utilizados para controlar o uso e alteração do conteúdo da página.



Figura 4.31: Entrada Típica de uma Tabela de Páginas

Exemplos de sistemas que se utilizam de diferentes formas para a implementação e administração de suas tabelas de páginas são:

- Paginação em um nível: DEC PDP-11.
- Paginação em dois níveis: DEC VAX.
- Paginação em três níveis: Sun Spark.
- Paginação em quatro níveis: Motorola 68030.
- Paginação via memória associativa (nível zero): MIPS R2000.
- Paginação via tabelas invertidas: IBM RS6000 (sistemas RISC), IBM PowerPC, IBM AS/400.

4.7.9 Algoritmos de troca de páginas

Os mecanismos de memória virtual se baseiam no fato de que porções dos processos são armazenadas em arquivos especiais denominados arquivos de troca. Quando um processo necessita acessar uma porção de seu código contida fora do espaço de endereçamento real, ocorre um *page fault*, ou seja, é detectada a falta de uma página que deverá ser trazida novamente para a memória principal.

As operações de substituição de páginas são lentas pois envolvem o acesso à memória secundário, ou seja, necessitam acessar dispositivos de entrada e saída, muito mais lentos do que a memória primária. É óbvio que se a página substituída for necessária em breve ou for preciso um número muito grande de substituições para execução dos programas ativos, ocorrerá a hiperpaginação (*hyperpaging* ou *thrashing*) ou seja, uma degradação significativa da performance do sistema devido ao excesso de operações de troca de páginas. Sendo assim estes algoritmos devem procurar substituir páginas pouco utilizadas ou não utilizadas por aquelas que são freqüentemente utilizadas pelos processos ativos, minimizando as operações de substituição de páginas.

O grande problema consiste então em determinar qual página será substituída ou copiada para os arquivos de troca como forma de liberar espaço para aquela página que se tornou necessária. Um algoritmo ótimo de troca de páginas deveria ser capaz de identificar qual página não mais será utilizada ou estabelecer aquela que demorará mais a ser novamente utilizada, minimizando a ocorrência de *page faults* e com isto as operações de troca de páginas. Como não é possível realizar tal previsão, outras formas de se definir qual página será substituída são empregadas [DEI92, p. 254].

Note que se uma página não tiver sido modificada então não necessita ser copiada para a memória virtual (armazenamento secundário), podendo ser simplesmente sobrescrita pela página que tomará seu lugar na memória primária, ou seja, uma operação de substituição simples. Se a página foi modificada então deverá ser copiada para o armazenamento secundário antes da substituição pela nova página, numa operação mais lenta do que uma substituição simples.

Os algoritmos que tratam deste problema são aqueles que implementam as estratégias de substituição (*replacement strategies*) e são denominados algoritmos de troca ou algoritmos de substituição de páginas. Os algoritmos de substituição de páginas mais comuns são:

- Random
- First In First Out (FIFO)
- Second Chance
- Clock
- Last Recently Used (LRU)
- Last Frequently Used (LFU)
- Not Recently Used (NRU)

Troca de páginas aleatória

Algoritmo de baixa sobrecarga que seleciona aleatoriamente qual página deverá ser substituída. Quanto maior o número de páginas existentes, maior são as chances de sucesso imediato deste algoritmo. Embora seja rápido e de implementação simples, é raramente utilizado dado que a página substituída pode ser a próxima a ser necessária. Também é chamado de *random page replacement*.

Troca de páginas FIFO

A idéia central deste algoritmo é que as páginas que estão a mais tempo na memória podem ser substituídas, ou seja, as primeiras que entram são as primeiras que saem (FIFO ou *First In First Out*). Para isto associa-se um marca de tempo (*timestamp*) para cada página, criando-se uma lista de páginas por idade, permitindo a identificação das mais antigas.

Este mecanismo de substituição, embora provável e lógico, não necessariamente se traduz em verdade, pois processos de longa duração pode continuar necessitando de suas páginas mais do que processos de curta duração que entram e saem rapidamente enquanto os outros permanecem. Dada esta razão não é utilizado na forma pura, mas sim variações deste algoritmo.

Troca de páginas segunda chance

O algoritmo de troca de páginas **segunda chance** (*second chance*) é uma variação da estratégia FIFO. Como visto, a deficiência do algoritmo de troca de páginas FIFO é que uma página de uso intenso, presente a muito tempo na memória, pode ser indevidamente substituída.

No algoritmo de troca de páginas Segunda Chance a seleção primária da página a ser substituída e semelhante ao FIFO, ou seja, escolhe-se a página mais antiga. Antes de proceder-se a substituição propriamente dita, verifica-se o *bit* de referência da página. Se o *bit* estiver em 1, significa que a página foi usada, daí o algoritmo troca sua marca de tempo por uma nova e ajusta o *bit* de referência para zero, *simulando* uma nova página na memória, ou seja, uma segunda chance de permanência na memória primária. Nesta situação outra página deve ser escolhida para substituição. Se o *bit* de referência estivesse em 0 a página seria substituída. Com este comportamento, se uma página antiga é utilizada, seu *bit* de referência sempre será 1, fazendo com que permaneça na memória primária a despeito de sua idade real.

Troca de páginas relógio

O algoritmo de troca de páginas **relógio** (*clock*) é uma outra variação da estratégia FIFO. Para superar a deficiência do algoritmo de troca de páginas FIFO, que é a substituição de uma página de uso intenso em função de sua idade na memória, o algoritmo **segunda chance** verifica o *bit* de referência mantendo na memória páginas em uso através da renovação de sua marca de tempo. Tal comportamento equivale a dizer que as páginas no início da lista (mais velhas) são reposicionadas no fim da lista (mais novas).

A estratégia do **relógio** é manter uma lista circular, onde se o *bit* de referência é 1, seu valor é trocado por 0 e a referência da lista movida conforme os ponteiros de um relógio. Caso contrário a página é substituída.

Troca de páginas LRU

A atuação do algoritmo de troca de páginas LRU (*least recently used* ou menos usada recentemente) se baseia em identificar a página que não foi utilizada pelo maior período de tempo, assumindo que o passado é um bom indicativo do futuro.

Para isto é necessário que cada página possua uma marca de tempo (*timestamp*) atualizada a cada referência feita à página, o que significa uma sobrecarga substancial. A implementação pode ser feita através de listas contendo uma entrada para cada page frame, sendo o elemento da lista correspondente a uma página utilizada sendo posicionado no final da lista.

Este algoritmo não costuma ser usado sem otimizações devido à sobrecarga que impõe. Além disso, em laços longos ou chamadas com muitos níveis de profundidade, a próxima página a ser usada pode ser exatamente uma das menos usadas recentemente, colocando o sistema numa situação de operações desnecessárias devido a page faults.

O sistema operacional MS Windows 95 utiliza esta estratégia de substituição de páginas [PET96, p. 725].

Troca de páginas LFU

Uma variante do algoritmo LRU é a estratégia conhecida como LFU (*least frequently used* ou menos freqüentemente usada) ou ainda NFU (*not frequently used* ou não usada freqüentemente). Neste algoritmo pretende-se calcular a freqüência de uso das páginas, de forma a se identificar qual página foi menos intensivamente utilizada.

Apesar de melhorar o desempenho do algoritmo LRU, ainda é possível que páginas pesadamente utilizadas durante uma certa etapa do processamento permaneçam desnecessariamente na memória primária em relação a outras etapas mais curtas cujas páginas não terão uso tão intenso, levando a substituições inúteis.

Troca de páginas NRU

As entradas de cada página na tabela de páginas possuem geralmente bits de referência e modificação (*bits referenced* e *modified*, conforme Figura 69) cujos valores combinados definem quatro situações ou grupos de páginas, como relacionado na Tabela 4.6.

Tabela 4.6: Grupos de páginas

Bit Referenced	Bit Modified	Situação
0	0	Não referenciado e não modificado
0	1	Não referenciado e modificado
1	0	Referenciado e não modificado
1	1	Referenciado e modificado

A atuação do algoritmo de troca de páginas NRU (*not recently used* ou não recentemente usada) ou NUR (*not used recently* ou não usada recentemente) se baseia em remover uma página, aleatoriamente escolhida, do grupo de menor utilização que contiver páginas nesta situação. Neste algoritmo se dá preferência a remoção de uma página modificada sem uso no último ciclo do que uma sem modificação que tenha sido utilizada.

Este algoritmo é utilizado em vários sistemas devido aos seus pontos fortes: simplicidade, fácil implementação e performance adequada.

Capítulo 5

Gerenciamento de I/O

O acrônimo I/O (*Input/Output*) ou E/S (Entrada/Saída) representa toda a sorte de dispositivos eletrônicos, eletromecânicos e ópticos que são integrados a um sistema computacional com o propósito de realizar a comunicação do processador e memória com o meio externo. De certa forma, o computador seria uma máquina inútil caso não existissem meios de realizar-se as operações de entrada e saída.

Os dispositivos de I/O, dispositivos periféricos ou apenas periféricos podem ser classificados de forma ampla em três categorias [STA96, p. 179]:

Human-Readable Dispositivos apropriados para serem utilizados por usuários do computador tais como o teclado, o *mouse* ou o monitor de vídeo.

Machine-Readable São aqueles projetados para interligação do computador com outros equipamentos, tais como unidades de disco, CD-ROM ou fita magnética.

Communication Destinados a realizar a comunicação do computador com outros dispositivos remotos, tais como placas de rede ou *modems*.

5.1 Módulos de I/O

A arquitetura de Von Neumann (Figura 1.2) define o computador como o conjunto de processador, memória e dispositivos de entrada e saída interconectados através vários barramentos (*buses*) especializados: o barramento de dados (*data bus*), o barramento de endereços (*address bus*) e o barramento de controle (*control bus*).

Mas diferentemente do que poderíamos pensar, os dispositivos periféricos em si não são conectados diretamente à tais barramentos, mas, ao invés disso, os periféricos são conectados a **módulos de I/O** que por sua vez são ligados aos barramentos do sistema. As razões para isto são várias:

- Existem inúmeros tipos de periféricos, com diferentes formas de operação, sendo impraticável implantar no computador uma lógica que permitisse a operação com todos ou mesmo uma grande parte destes dispositivos.
- Como visto, a velocidade de operação dos dispositivos periféricos é muito menor que a da memória ou do processador.

Desta forma é muito mais conveniente implementar-se **módulos de I/O** que atuem como conexões mais genéricas para os diferentes periféricos, possibilitando o uso de estruturas padronizadas para ligação com a memória e processador. Por essa razão, os módulos de IO são freqüentemente chamados de **interfaces**.

Um módulo de I/O geralmente possui algumas linhas de controle internas que servem para determinar as ações que devem ser executadas pelo módulo, linhas de *status* que servem para indicar o estado de funcionamento do módulo ou resultado de alguma operação, linhas de dados para conexão do módulo de I/O com o barramento de dados do sistema e um conjunto particular de linhas para conexão com o periférico que efetivamente será controlado pelo módulo de I/O. Na Figura 5.1 temos um representação genérica do que pode ser um módulo de I/O.

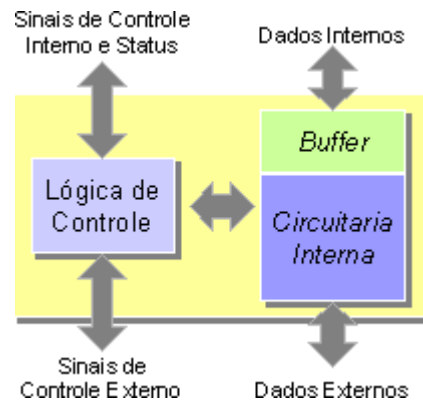


Figura 5.1: Estrutura genérica de módulo de I/O

A conexão com o barramento do processador geralmente tende a ser uma interface de alto nível, isto é, uma conexão orientada à comandos, mais adaptada à operação com um processador e, portanto, dirigida ao trabalho de programação de alto nível. Já a conexão com o periférico propriamente dito é usualmente uma interface de baixo nível, contendo inúmeros sinais elétricos e um protocolo dedicado próprio, que exige tratamento mais especializado. Desta forma, as particularidades de cada tipo de periférico ficam isoladas do sistema principal, facilitando o desenvolvimento dos programas que utilizarão estes dispositivos.

Esta estrutura permite que um único módulo de I/O controle mais de um periférico, geralmente do mesmo tipo, tal como nos controladores de unidades de disco IDE¹ que podem administrar de uma até quatro unidades de disco deste padrão.

Os módulos de I/O geralmente executam algumas das seguintes funções: controle e temporização, comunicação com o processador, comunicação com periférico, armazenamento temporário de dados e detecção de erros.

Uma outra estrutura possível para os módulos de I/O são os **canais de I/O** (*I/O channels*). Os canais de I/O são sistemas computacionais de propósito especial destinados ao tratamento de entrada e saída de forma independente do processador do sistema computacional [DEI92, p. 27]. Esta alternativa estrutural, usada tipicamente em computadores de grande porte (*mainframes*), opera com múltiplos barramentos de alta velocidade, podendo acessar o armazenamento primário de forma independente, proporcionando grande desempenho, pois as operações de I/O são realizadas paralelamente ao processamento.

Micro e minicomputadores utilizam geralmente um modelo de barramento interno simples para comunicação entre processador, memória e os demais dispositivos do sistema. Compartilhando este barramento encontram-se dispositivos especializados nas em funções mais importantes (unidades de disco e monitor de vídeo) chamados controladores, proporcionando considerável ganho de performance e ainda assim utilizando uma arquitetura mais simples que os canais de I/O [TAN92, p. 207].

5.2 Operação de Módulos de I/O

Os módulos de I/O podem ser operados de três maneiras básicas:

- I/O Programado,
- I/O com Interrupções e
- I/O com Acesso Direto à Memória (DMA)

De forma geral, o que distingue estas três formas de operação é a participação do processador e a utilização do mecanismo de interrupções, conduzindo a resultados bastante distintos.

5.2.1 I/O Programado

Com o I/O programado (*programmed I/O*) os dados são trocados diretamente entre o processador e o módulo de I/O, ou seja, o processador deve

¹O acrônimo IDE significa *integrated device electronics* ou dispositivo eletrônico integrado.

executar um programa que verifique o estado do módulo de I/O, preparando-o para a operação se necessário, enviando o comando que deve ser executado e aguardando o resultado do comando para então efetuar a transferência entre o módulo de I/O e algum registrador do processador.

Portanto é responsabilidade do processador verificar o estado do módulo de I/O em todas as situações, inclusive quando aguarda dados. Isto significa que o processador não pode executar outras tarefas enquanto aguarda a operação do módulo de I/O. Veja na Figura 5.2 um esquema indicando o funcionamento das operações de I/O programadas.

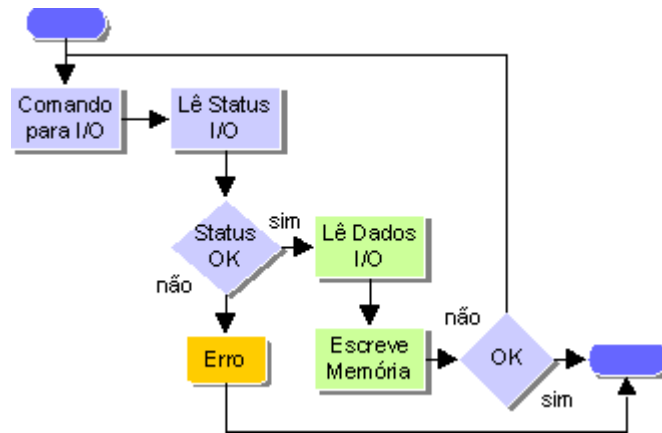


Figura 5.2: Funcionamento do I/O programado

Como os comandos enviados ao módulo de I/O podem significar uma operação com um dispositivo periférico lento, digamos uma unidade de disco, então o processador deverá permanecer em espera durante toda a operação executada no periférico, por mais lento que possa ser, significando um sério comprometimento da performance geral do sistema, dado que o processador fica ocupado com a monitoração da operação em andamento (veja Figura 5.3).

O módulo de I/O se relaciona com o processador através de comandos, ou seja, códigos de controle transformados em sinais enviados ao módulo que indica qual operação deverá ser realizada, tais como controle (*control*), teste (*test*), escrita (*write*) ou leitura (*read*). Geralmente os comandos do módulo de I/O tem equivalência direta com as instruções do processador, facilitando sua operação integrada.

Dado que é possível para um módulo de I/O controlar mais de um dispositivo periférico, então é necessário que sejam associados endereços a cada um destes periféricos, de forma a permitir sua operação individualizada. Existem duas formas para a interpretação destes endereços por parte dos módulos de I/O quando existe o compartilhamento de barramentos do sistema:

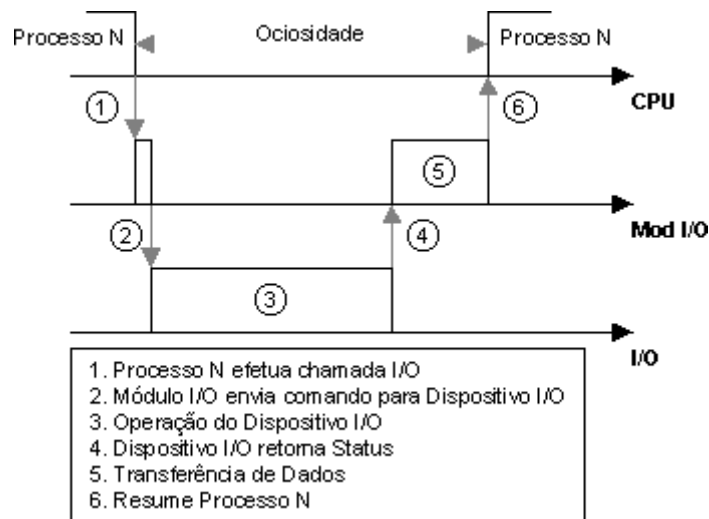


Figura 5.3: Temporização do I/O programado

- **Mapeada em Memória** (*memory-mapped*)

Onde o módulo de I/O opera dentro do espaço de endereçamento da memória, usando um conjunto de endereços reservados. Desta forma o processador trata os registradores de *status* e dados do módulo de I/O como posições ordinárias de memória utilizando operações comuns de leitura e escrita. Para o funcionamento neste modo o processador deve dispor de uma linha individual de leitura e outra para escrita.

- **Mapeada em I/O** (*I/O mapped*)

Também chamada de I/O isolado (*isolated I/O*), onde existe um espaço de endereçamento independente para os dispositivos de I/O. Para tanto o processador deve dispor de uma linha de leitura/escrita e outra de entrada/saída. As portas de I/O passam a ser acessíveis apenas por operações especiais de I/O.

Utilizando a forma de I/O mapeado em memória temos uma maior simplicidade e a disponibilidade de um maior conjunto de instruções embora reduzido espaço em memória devido a reserva de endereços para portas de I/O. Os computadores baseados nos processadores Motorola 680x0 utilizam este método.

Com o I/O isolado temos maior segurança nas operações envolvendo memória ou I/O e um maior espaço de endereçamento as custas de uma organização ligeiramente mais complexa e um reduzido número de instruções dedicadas. Os microcomputadores IBM PC compatíveis são um exemplo desta utilização. No caso, para enviar-se dados ao monitor de vídeo padrão do sistema devem ser utilizados os endereços de I/O na faixa de 03D0h a 03DFh, enquanto que o acesso às placas de som tipo SoundBlaster devem

usar o endereço 0220h.

5.2.2 I/O com interrupções

Para superar o problema da espera do processador por operações nos dispositivos periféricos, pode ser utilizado o mecanismo das interrupções, ou seja o I/O através de interrupções (*interrupt driven I/O*). Tecnicamente falando, uma interrupção permite que uma unidade ganhe a atenção imediata de outra, de forma que a primeira unidade possa finalizar sua tarefa [DEI92, p. 25].

Assim sendo, quando o processador envia um comando para o módulo de I/O, o mesmo pode passar executar uma outra tarefa, sem a necessidade de monitorar o módulo acionado. Quando a operação for concluída, o módulo de I/O interrompe o processador, isto é, aciona uma interrupção para requisitar o processamento dos dados (a troca de dados com o processador). O processador executa a troca de dados, liberando o módulo de I/O e retomando o processamento anterior.

Conforme podemos observar na Figura 5.4, a operação de I/O com interrupções é a seguinte:

1. O processador envia um comando ao módulo de I/O (por exemplo, uma operação *read*), que a realiza de modo independente (i.e., em paralelo a atividade do processador).
2. O processador passa a executar outra tarefa, ou seja, um outro processo.
3. Ao finalizar a operação requisitada, o módulo de I/O sinaliza uma interrupção para o processador.
4. Ao término da instrução corrente, o processador verifica a ocorrência de uma interrupção, determinando qual dispositivo a originou e então sinalizando conhecimento (*acknowledgment signal*).
5. O processador salva o contexto da tarefa atual na pilha (*stack*), preservando assim o conteúdo do contador de programa e demais registradores.
6. O processador carrega o endereço da rotina de serviço no contador de programa para poder iniciar a execução da rotina de tratamento da interrupção detectada. Tais endereços são armazenados numa região pré-determinada da memória, denominada vetor de interrupções.
7. A rotina de tratamento da interrupção é executada, ou seja, os dados solicitados são lidos do módulo de I/O para um registrador do processador e depois transferidos para uma área de memória apropriada.

8. Finalizada a rotina de tratamento da interrupção, o processador restaura o contexto da tarefa interrompida, lendo o conteúdo do contador de programa e demais registradores da pilha.
9. O processador retorna ao processamento da tarefa no ponto em que foi interrompida.
10. Mais um momento mais a frente, em seu *quantum*, o processo que solicitou a operação de I/O será executado com os dados obtidos a sua disposição.

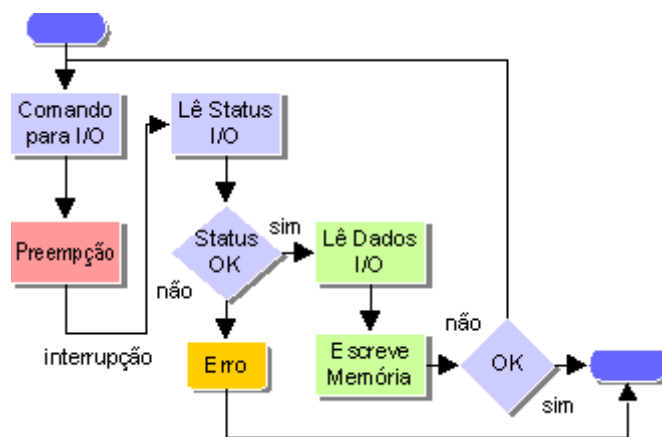


Figura 5.4: Funcionamento do I/O com interrupções

A operação de dispositivos de I/O utilizando interrupções permite que o processador permaneça trabalhando enquanto o módulo de I/O realiza a operação solicitada, melhorando o desempenho do sistema pois duas atividades são *paralelizadas*, embora os dados da operação continuem a ser manipulados pelo processador, como mostra também a Figura 5.5.

O maior problema relacionado com o uso das interrupções é que, usualmente, o processador dispõe de poucas linhas de interrupção. Desta forma surgem as seguintes questões: como o processador identificará o módulo que sinalizou uma interrupção e como serão tratadas múltiplas interrupções simultâneas?

Para resolver-se esta questões, podem ser empregadas várias diferentes técnicas [STA96, p. 194]: múltiplas linhas de interrupção, *software poll*, *hardware poll* vetorizado e *bus arbitration* vetorizada. Usualmente são assinalados números para as interrupções, onde as de menor número tem prioridade sobre as número maior. Isto significa que uma interrupção de número 4 será processada primeiro do que uma interrupção de número maior (≥ 5), sem ser interrompida por estas, mas podendo ser interrompida por uma interrupção de número menor (< 4).

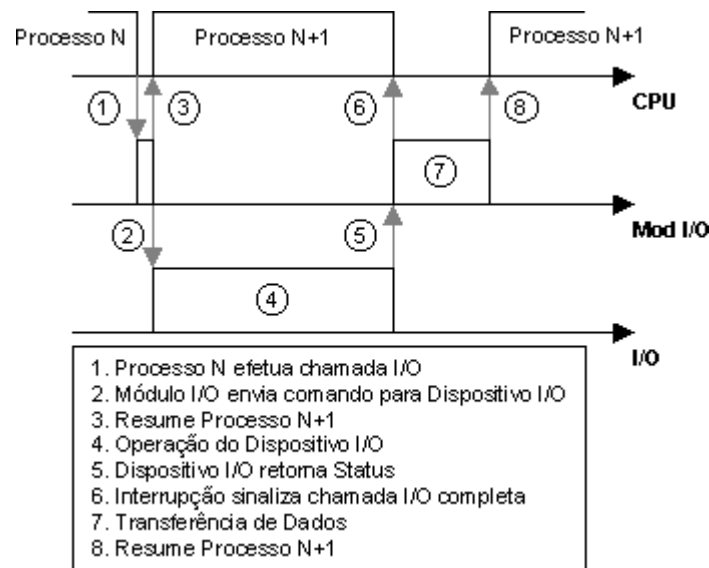


Figura 5.5: Temporização do I/O com interrupções

Como exemplo, apresentamos a Tabela 5.1 que contém o mapeamento das interrupções num sistema IBM-PC compatível. Alguns dos valores são padronizados enquanto outros são particulares do sistema utilizado como exemplo.

Tabela 5.1: Mapa de interrupções de um IBM-PC compatível

Int	Dispositivo	Int	Dispositivo
0	Cronômetro do sistema	8	CMOS/Relógio do sistema
1	Teclado	9	Porta de comunicação COM3
2	Controlador de interrupção	10	Porta de comunicação COM2
3	Placa de rede (*)	11	Ponte PCI (*)
4	Porta de comunicação COM1	12	Mouse porta PS/2 (*)
5	Placa de som (*)	13	Coprocessador numérico
6	Controlador de disco flexível	14	Controlador IDE/ESDI
7	Porta de Impressora LPT1	15	Controlador IDE/ESDI

(*) Opções não padronizadas

5.2.3 I/O com Acesso Direto à Memória (DMA)

As técnicas de I/O programado e I/O com interrupções possuem um grande inconveniente que é a limitação da velocidade de transferência de dados a capacidade do processador em movimentar tais dados a partir do módulo de I/O para o armazenamento primário, pois isso envolve a execução repetida de várias instruções. Além disso o processador fica comprometido não

apenas com a transferência dos dados, mas com a monitoração do módulo de I/O, no caso de I/O programado, ou com a sobrecarga imposta pelas operações de interrupção, no caso de I/O via interrupção. Se um módulo de I/O for utilizado para a movimentação de uma grande quantidade de dados, ambas as formas comprometerão a performance do sistema.

Para solucionar-se este problema pode ser utilizada uma outra técnica denominada I/O através de acesso direto à memória ou DMA (*direct memory access*).

A técnica de DMA propõe utilizar uma única interrupção para efetuar a transferência de um bloco de dados diretamente entre o periférico e a memória primária, sem o envolvimento do processador e com isso reduzindo o número de operações necessárias e assim acelerando o processo.

Para tanto, torna-se necessária a existência de um módulo adicional, chamado de controlador de DMA, cuja operação, ilustrada na Figura 5.6, é descrita a seguir [STA96, p. 199]:

1. O processador envia comando (leitura ou escrita) para controlador de DMA.
2. O processador continua seu trabalho enquanto DMA efetua a transferência com o dispositivo de I/O.
3. Para acessar a memória o controlador de DMA *rouba* ciclos do processador para acessar a memória principal, atrasando-o ligeiramente.
4. Ao final da operação o controlador de DMA aciona uma interrupção para sinalizar o término da operação.
5. O processador pode executar a rotina de tratamento da interrupção processando os dados lidos ou produzindo novos dados para serem escritos.

Este método é significativamente mais rápido do que o I/O programado ou I/O via interrupções pois utiliza apenas uma única interrupção, o processador fica liberada para executar outras tarefas e a transferência dos dados ocorre em bloco (e não *byte a byte*) diretamente entre o periférico e a memória.

O controlador de DMA é um dispositivo especializado nesta operação, suportando tipicamente o trabalho com vários periféricos diferentes, cada um utilizando um canal de DMA (*DMA channel*).

Outra grande vantagem da técnica de DMA é que ela pode ser implementada no *hardware* de diversas formas diferentes, conforme a quantidade de dispositivos de I/O e performance pretendida, como ilustrado na Figura 5.7.

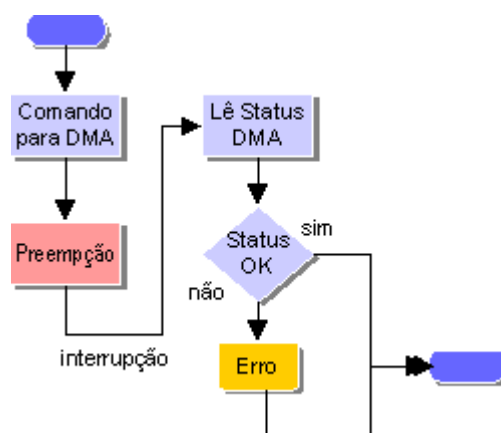


Figura 5.6: Funcionamento do I/O com DMA

5.3 Tipos de dispositivos de E/S

Os dispositivos de I/O e suas interfaces podem ser classificados de forma ampla quanto ao tipo de conexão e tipo de transferência de dados.

5.3.1 Conexão de Dados dos I/O

Conforme natureza do periférico que será conectado ao sistema e também as condições desta ligação, as conexões dos dispositivos de I/O, do ponto de vista dos dados, são projetadas para operação **serial** ou **paralela**.

Numa conexão **serial**, uma única linha de sinal é utilizada para o estabelecimento de toda a conexão, protocolo e transferência de dados entre o módulo de I/O e o periférico, ou seja, todos os *bits*, sejam de dados ou controle, são transferidos um a um entre módulo de I/O e periférico.

Numa conexão **paralela**, várias linhas de sinal são utilizadas de modo que vários *bits* de dados (*bytes* ou *words* tipicamente) sejam transferidos em paralelo, ou seja, ao mesmo tempo, acelerando as transferências, pois se comportam como várias linhas seriais atuando ao mesmo tempo. Também é comum que existam linhas independentes para o tráfego de sinais de controle.

As conexões seriais são baratas, relativamente confiáveis, embora nominalmente mais lentas que as conexões paralelas, sendo usualmente utilizadas para dispositivos baratos e lentos tais como impressoras e terminais. As conexões paralelas, devido a interface mais complexa, são mais caras, bastante confiáveis e de melhor desempenho, sendo utilizadas para conexão com dispositivos mais rápidos, tais como unidades de disco, unidades de fita ou mesmo impressoras rápidas.

Em ambas os tipos de conexão, o módulo de I/O e o periférico trocam sinais de controle garantindo a permissão para o envio ou recebimento de dados (protocolo de conexão ou *handshaking*). A transferência dos dados

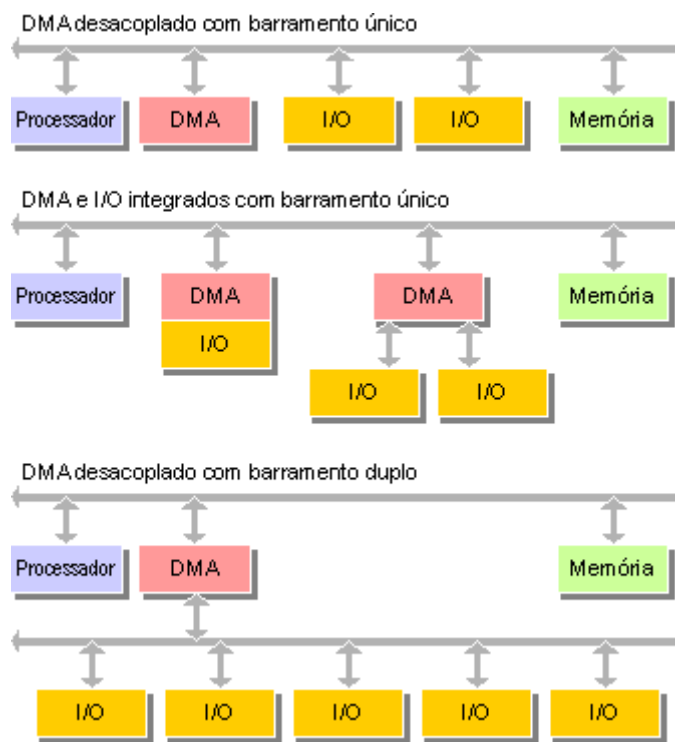


Figura 5.7: Configurações de DMA

é feita, exigindo o envio de sinais de confirmação a cada *byte* ou bloco dependendo do dispositivo, tipo de conexão e do protocolo de transferência adotado.

5.3.2 Tipos de Transferência de I/O

Os dispositivos de I/O atuam usualmente como dispositivos **orientados à caractere** (*character devices*) e dispositivos **orientados à blocos** (*block devices*).

Nos primeiros, orientados à caractere, a transferência de dados é feita *byte a byte*, sem a necessidade de alguma forma de estruturação dos dados por parte do módulo de I/O e do periférico, ou seja, o formato dos dados recebidos e transmitidos é responsabilidade da aplicação que utiliza o dispositivo.

Nos dispositivos de transferência orientados à blocos, a troca de dados é realizada em blocos de tamanho fixo, cujo tamanho depende do dispositivo, usualmente entre 128 e 1024 bytes. Os blocos também possuem um formato particular, exigindo que a aplicação conheça tal formato tanto para a construção de tais blocos destinados à transmissão como para sua adequada recepção.

Temos portanto que a operação de dispositivos orientados à caractere e à blocos é bastante diferente. Unidades de disco e fita são dispositivos orientados à blocos enquanto que impressoras, terminais, teclados e portas seriais são orientados à caractere [PIT98, p. 68].

Nem todos os dispositivos se ajustam a esta classificação, tais como os temporizadores (*timers*) do sistema ou monitores de vídeo de memória [TAN92, p. 206].

Nos sistemas Unix esta distinção é bastante aparente, principalmente durante os procedimento de instalação e configuração do sistema operacional.

5.3.3 Conexões ponto a ponto e multiponto com I/Os

A conexão mais simples entre um dispositivo periférico e seu módulo de I/O é do tipo , ou seja, as linhas de sinais existentes para a comunicação entre estas unidades são dedicadas a este fim. Desta forma, um módulo de I/O deveria dispor de um conjunto de linhas para dispositivo de I/O com o qual pretende se comunicar.

Por outro lado, é possível que um módulo de I/O compartilhe um conjunto de linhas de sinais entre diversos dispositivos periféricos, desde que dentre estas linhas existam algumas para realizar o endereçamento ou seleção do dispositivo com o qual deseja-se realizar a operação. A conexão **multi-ponto** é como um conjunto de barramentos dedicado a comunicação entre um módulo de I/O e vários dispositivos semelhantes. Uma representação das conexões ponto a ponto e multiponto se encontra na Figura 5.8.

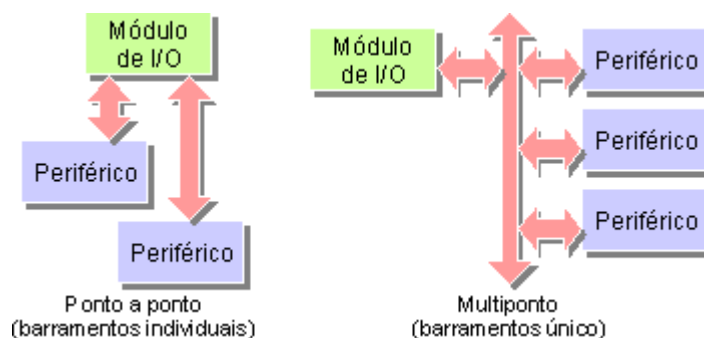


Figura 5.8: Conexões ponto-a-ponto e multiponto

A conexão ponto-a-ponto oferece melhor confiabilidade, permite a operação simultânea de diversos periféricos simultaneamente (dependendo apenas das capacidades do módulo de I/O) embora exigindo um maior número de conexões e, portanto linhas de sinal. É geralmente utilizada para a conexão de dispositivos mais simples, tais como modems, teclado e impressoras.

Exemplos de conexões ponto-a-ponto padronizadas são os protocolos RTS/CTS (*Request to Send* e *Clear to Send*) e Xon/Xoff (*Transmission*

On e *Transmisson Off*). O RTS/CTS e Xon/XOff são considerados protocolos de baixo nível simples, bastante utilizados em comunicação de curta distância entre computadores e periféricos de baixa velocidade, usualmente utilizando a interface padrão RS-232C (equivalente à *standard CCITT V.24*) [BLA87, p. 53].

Veja uma representação do funcionamento destes protocolos entre dois equipamentos DTE (*data terminal equipment*) na Figura 5.9.

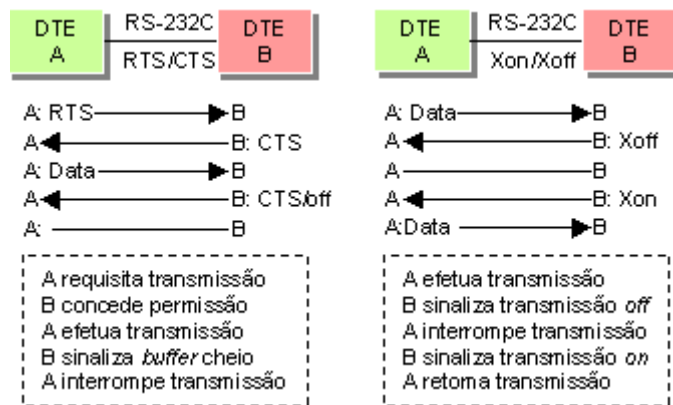


Figura 5.9: Protocolos RTS/CTS e Xon/Xoff

Dado que são protocolos simples, sua implementação é fácil, constituindo uma alternativa flexível e de baixo custo para interligação de equipamentos tais como multiplexadores, demultiplexadores, *modems*, impressoras, terminais de vídeo, *plotters*, mesas digitalizadoras etc.

A conexão multiponto é bastante mais flexível do que a conexão ponto-a-ponto pois permite maior escalabilidade, utilizando reduzido número total de linhas, mas por outro lado não permite a operação simultânea dos periféricos conectados. Tal conexão é tipicamente utilizada para dispositivos de armazenamento, tais como unidades de disco, fita, cartucho, CD-ROM, etc. Existem vários padrões para estas conexões, onde são exemplos:

- IDE (*integrated device electronics*),
- EIDE (*extended IDE*),
- SCSI (*small computer system interface*),
- USB (*universal serial bus*)

5.4 Dispositivos periféricos típicos

Os dispositivos periféricos tem papel fundamental dentro de um sistema computacional, pois como colocado anteriormente, o computador seria inútil

se fosse apenas composto de processador e memória. Existem muitos tipos de dispositivos periféricos, dentre os mais comuns podemos citar:

Unidades de Disco Rígido	Unidades de Disco Flexível
Unidades de Fitas Magnética	Unidades de CD-R/RW
Unidades de DVD-R/RW	<i>Mouse</i>
<i>Trackball</i>	<i>Mousepad</i>
Teclados	<i>Scanners</i>
Mesas digitalizadoras	Impressoras
<i>Modems</i>	Portas de comunicação serial
Portas de comunicação paralela	Portas de comunicação USB
Placas de Rede	Monitores de vídeo
Portas de jogos	Placas de som
Placas de captura de vídeo	etc.

Além destes, a maioria voltados para uso em microcomputadores, existem dispositivos apropriados para sistemas computacionais de grande porte, tais como controladoras de terminais, terminais de vídeo, subsistemas de armazenamento secundário, subsistemas de comunicação etc.

Dentre todos estes dispositivos daremos destaque as unidades de disco e fita por representarem os periféricos mais importantes do ponto de vista de armazenamento secundário. Além destes, faremos alguns comentários sobre os terminais de vídeo, essenciais em sistemas multiusuário.

5.4.1 Unidades de disco

Atualmente, praticamente todos os tipos de computadores dispõem de unidades de disco magnético. Estas unidades são compostas de um ou mais discos metálicos de aço ou alumínio recobertos de uma fina película magnetizável. Estes discos, montados verticalmente num mesmo eixo, giram em velocidade constante (2400 ou 3600 rpm, por exemplo). As unidades podem possuir cabeças de leitura e gravação fixas (uma para cada trilha de cada disco) ou cabeças móveis (uma para cada disco). Neste caso braços mecânicos, dotados de dispositivos de acionamento, são responsáveis pela movimentação rápida e precisa de cabeças por praticamente toda a superfície dos discos. Estas cabeças são capazes de gravar ou ler dados magneticamente armazenados na película que recobre os discos, dados estes que permanecem gravados mesmo que as unidades de disco permaneçam desligadas por um razoável período de tempo.

As tecnologias envolvidas no desenvolvimento e fabricação das unidades de disco vem sendo aperfeiçoadas desde as primeiras gerações de computadores e com isto, as unidades de disco magnético comuns, isto é, instaladas em computadores destinados ao segmento SOHO², exibem as seguintes características:

²SOHO significa *small office or home office*, ou seja, micro e pequenas empresas além de escritórios domésticos caracterizando um grande segmento de mercado que utiliza mi-

- Grandes capacidades de armazenamento, tipicamente maiores que 1 GBytes (2^{30}),
- Dimensões reduzidas (discos de 3.5" ou menores),
- Baixo consumo (apropriados para equipamentos portáteis),
- Tempos de acesso inferiores a 15 ms e
- Baixo custo por MByte.

As unidades de disco são construídas de forma modular por questões de economia e modularidade, permitindo que várias unidades possam ser controladas por um mesmo módulo de I/O, mais conhecido como controladora de disco, em arranjos ponto a ponto ou multiponto, como mostra a Figura 5.8. A configuração multiponto é mais comum pois simplifica o projeto das controladoras de disco dada a redução do número de conexões necessárias. Isto permite grande flexibilidade aos sistemas computacionais pois a capacidade do armazenamento secundário pode ser aumentada pela simples adição de novas unidades de disco.

Outros sistemas tem duplicadas suas unidades de disco, utilizando técnicas de espelhamento (*mirroring*), onde os dados são gravados de forma idêntica nas unidade espelhadas, permitindo a rápida recuperação do sistema em caso de falhas. Uma outra estratégia de alta confiabilidade e disponibilidade é a utilização de múltiplas unidades de disco num arranjo conhecido como RAID (*redundant array of inexpensive disks*), onde os dados são gravados de forma distribuída num grupo de unidades permitindo até mesmo a substituição de uma unidade de disco com o equipamento em funcionamento. Stallings traz maiores detalhes sobre as técnicas de RAID [STA96, p. 161].

As unidade de disco, que são dispositivos de **acesso direto**, isto é, qualquer setor contendo informação pode ser acessado através de uma simples operação de pesquisa (*seek*) sem necessidade da leitura de setores adicionais. Dispositivos desta natureza também são chamados de dispositivos de acesso aleatório. A IBM tradicionalmente denomina suas unidades de disco de DASD (*direct access storage devices*) numa clara alusão a forma com que os dados pode serem lidos e gravados.

Com tais características, podemos perceber sua importância para os computadores. Segundo Deitel [DEI92, p. 26], as unidades de disco magnético são talvez o mais importante periférico dentro de um sistema computacional.

Organização dos discos

Do ponto de vista de organização, uma unidade de disco pode possuir um ou vários **discos** (*disks*), às vezes chamados de pratos (*platters*). Todo o crocomputadores e equipamentos de pequeno porte.

conjunto de discos é dividido em circunferências concêntricas denominadas **cilindros** (*cylinders*). Para cada superfície de disco equipada com cabeça de leitura (*head*) se define uma **trilha** (*track*), que também é dividida radialmente em **setores** (*sectors*), tal como fatias de uma *pizza*. Entre as trilhas existe um espaço livre (*inter-track gap*) tal como entre os setores (*inter-sector gap*). Todo o espaço livre existente entre trilhas e setores não é utilizado por estes dispositivos. Na Figura 5.10 temos a estrutura de uma unidade de disco magnético.

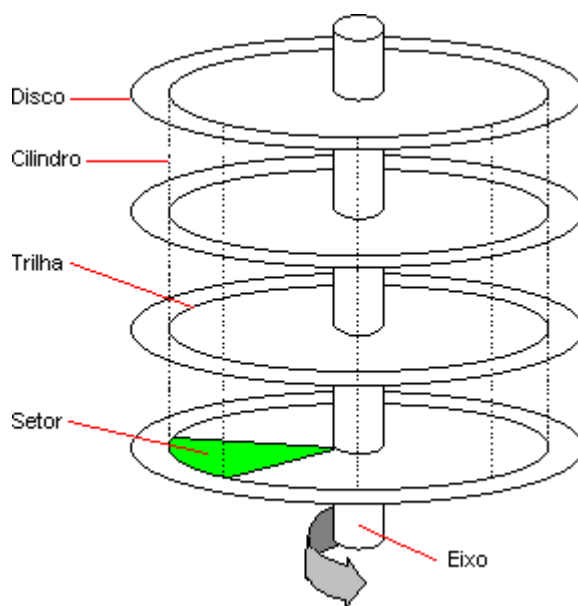


Figura 5.10: Estrutura esquemática de uma unidade de disco magnético

Como forma de simplificação, todas as trilhas armazenam a mesma quantidade de dados, portanto a densidade de gravação é maior nas trilhas interiores dos discos. O *hardware* da unidade de disco dispõe de meios para identificar o setor inicial sendo que os demais setores são identificados conforme o disco e trilha ao quais pertencem, recebendo uma numeração de referência. O processo de divisão das superfícies em trilhas e setores é o que se denomina **formatação física** da unidade enquanto que sua adequação ao sistema de arquivos do sistema operacional é chamada de **formatação lógica**. Os dados são gravados nos setores de cada trilha podendo ser recuperados posteriormente se for conhecido o número do setor desejado.

Uma outra característica fundamental das unidades de disco é a possibilidade de serem divididas em **partições**. Uma partição é um conjunto de cilindros consecutivos, cujo tamanho é determinado pelo usuário [NOR89, p. 103], permitindo que:

- uma unidade de disco física seja vista e tratada como várias unida-

des de disco lógicas distintas, facilitando a organização dos dados e instalação de programas; e

- vários sistemas operacionais diferentes sejam instalados nas diversas partições, ampliando significativamente as possibilidades de uso da máquina.

No sistema operacional multiusuário IBM VM/SP (*Virtual Machine/System Product*), usado em computadores IBM de grande porte, um procedimento comum para alocação de espaço em disco para os usuários era a criação de **mini-discos**, na verdade, pequenas partições de uma das unidades de disco do sistema. Através de uma solicitação aos operadores do sistema, o usuário recebia direitos de acesso à um novo mini-disco, criado para seu uso particular. Para o usuário, cada mini-disco aparentava ser uma unidade de disco de uso privativa e isolada das demais, onde os arquivos e programas eram criados e modificados, podendo o usuário dar direitos de acesso de seu mini-disco para outros usuários do sistema. Quando necessário, o usuário podia pedir uma ampliação de seu mini-disco ou requerer um novo mini-disco, de forma a possuir várias mini-partições diferentes do sistema.

Resumidamente, características importantes que descrevem uma unidade de disco são: número de discos, número de superfícies, número de cilindros, número de setores, movimentação das cabeças (fixas ou móveis), tipo de cabeças (de contato, de espaçamento fixo ou aerodinâmicas), tempo médio de acesso, capacidade de transferência da controladora e MTBF (*Medium Time Between Failures*).

A maioria dos detalhes de operação das unidades de disco magnética são tratadas pelas controladoras de disco, cujas interfaces padronizadas são de fácil integração com a maioria dos sistemas computacionais. Estas interfaces padronizadas (por exemplo IDE, EIDE e SCSI) trabalham com comandos de alto nível, facilitando o trabalho de programação. As controladoras geralmente possuem capacidade para tratar vários discos simultaneamente, embora a operação paralela se resuma ao acionamento das unidade de discos em busca de setores (*overlapped seeks*), pois a transferência de dados de uma única unidade geralmente consome toda capacidade de processamento destas controladoras. Ainda assim, a habilidade de realizar acessos (*seeks*) aos discos melhora consideravelmente a performance destas unidades.

Tempo de acesso

Quando é solicitada uma operação de leitura ou escrita numa unidade de disco, é necessário mover-se a cabeça de leitura e escrita até o setor desejado para o início da operação. Tal tempo é determinado por três fatores [TAN92, p. 217] [STA96, p. 160]:

1. o tempo necessário para mover-se até o cilindro correto, ou seja, o **tempo de acesso** à trilha ou tempo de pesquisa da trilha (*seek time*);
2. o tempo necessário para a cabeça ser posicionada no início do setor desejado, chamado de **atraso rotacional** (*rotational delay*) e
3. o **tempo de transferência dos dados**, isto é, a leitura ou escrita dos dados (*transfer time* ou *transmission time*).

A soma destes três componentes de tempo é o que se denomina tempo de acesso (*access time*), ou seja, o tempo necessário para a leitura ou escrita de um determinado setor, como indicado na Equação 5.1. Estes componentes do tempo de acesso também estão representados na Figura 5.11.

$$t_{access} = t_{seek} + t_{rotationaldelay} + t_{transfer} \quad (5.1)$$

Dado que, para a maioria das unidades de disco, o tempo de movimentação entre trilhas (*seek time*) é o maior dentro desta composição, sua redução colabora substancialmente com a performance destes dispositivos (esta é razão pela qual existem unidades de disco com cabeças fixas, pois nelas o *seek time* é zero).

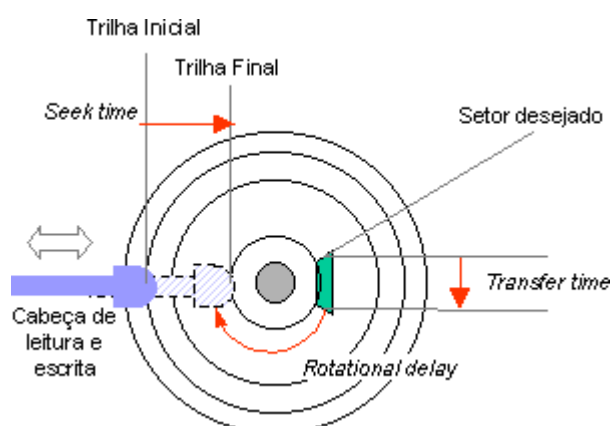


Figura 5.11: Componentes do tempo de acesso de uma unidade de disco

5.4.2 Escalonamento de disco

Uma das maiores questões relacionadas ao desempenho das unidades de disco se relaciona a forma com que as cabeças de leitura são posicionadas em função dos pedidos de leitura e escrita realizados. O controle deste atendimento é feito por algoritmos de escalonamento de disco (*disk scheduling algorithms* ou *disk arm scheduling algorithms*).

Considerando um sistema multiprogramado onde existam inúmeros processos efetuando pedidos de leitura e escrita, em função da maior velocidade

de processamento em relação a capacidade de realizar a leitura e escrita dos dados por parte da unidade de disco, é razoável considerar que os pedidos deverão esperar para poderem ser atendidos, portanto os processos ficarão bloqueados até suas respectivas solicitações serem completas.

Quanto mais rápido a unidade de disco puder completar as solicitações, menor o tempo de espera dos processos, beneficiando o sistema. Como as operações solicitadas provavelmente utilizarão setores distintos, a unidade deverá efetuar uma série de movimentações da cabeça de leitura para realizar o trabalho necessário, assim sendo, quanto menor o tempo despendido na movimentação da cabeça de leitura melhor o desempenho da unidade. Esta é a justificativa da preocupação com o escalonamento das tarefas de movimentação das cabeças de leitura [DEI92, p. 363].

Um algoritmo para escalonamento do disco deve então proporcionar boa produtividade (*throughput*), oferecer baixo tempo de resposta e apresentar razoável previsibilidade (comportamento previsível nas diversas situações de carga). Existem vários algoritmos para escalonamento do disco, onde alguns preocupam-se em otimizar a movimentação entre trilhas e outros em aproveitar o percurso rotacional das cabeças de leitura:

- FCFS (*first come first served*)
Neste algoritmo, a fila de pedidos é executada na ordem em que aparecem, sem qualquer reordenação ou otimização de movimentação. Esta forma de escalonamento pode resultar em longos tempos de espera em situações de alta carga de trabalho, embora razoável para situações de baixo carregamento [DEI92, p. 366] [TAN92, p. 217].
- SSTF (*shortest seek time first*)
A fila de pedidos é executada de forma que sejam atendidos primeiro os pedidos que exigem a menor movimentação possível entre trilhas, qualquer que seja o sentido da movimentação (setores internos para centro ou setores externos para bordas). Pedidos destinados aos setores extremos geralmente recebem baixa qualidade de serviço, podendo ocorrer um adiamento indefinido (*starvation*), além disso proporciona grande variância em termos de desempenho conforme a seqüência de pedidos. É um algoritmos orientado à cilindros [DEI92, p. 366] [TAN92, p. 218].
- SCAN
É uma variação do SSTF, desenvolvida por Denning em 1967, que pretendia corrigir sua variância e a possibilidade de adiamento indefinido. O SCAN, tal como o SSTF, também trabalha escolhendo os pedidos que exigem menor movimentação, mas apenas numa direção preferencial, ou seja, ele primeiro realiza os pedidos numa direção (p.e., do cilindro mais externo para o mais interno) para depois realizar uma mudança de direção (do cilindro mais interno para o mais externo)

completando as tarefas. Novos pedidos que vão surgindo durante a varredura são atendidos se possível durante a varredura em andamento. Por isso também é conhecido como algoritmo do elevador (*elevator algorithm*). Embora os setores externos continuem a ser menos visitados que os setores intermediários, não existe a possibilidade de um adiamento indefinido e a qualidade do serviço é mais regular. Para baixa carga de trabalho este é melhor algoritmo de escalonamento para disco conhecido. Também é um algoritmos orientado à cilindros [DEI92, p. 366].

- C-SCAN

O algoritmo C-SCAN (*circular SCAN*) é uma variação do algoritmo SCAN que elimina a questão de atendimento diferenciado para os cilindros extremos. O C-SCAN atende os pedidos, na ordem em que exigem menor movimentação, seguindo uma direção pré-definida: do cilindro mais externo para o mais interno. Ao finalizar os pedidos nesta direção, o braço é deslocado para o pedido que necessita o setor mais externo sendo reiniciada a varredura. Para uma carga de trabalho média este algoritmo proporciona os melhores resultados. Se também otimizado para minimizar o atraso rotacional, torna-se o melhor algoritmo, inclusive para alta carga de trabalho [DEI92, p. 369].

- N-SCAN

O algoritmo N-SCAN (*n step SCAN*) é uma outra variação do SCAN. Sua movimentação é semelhante ao SCAN, exceto pelo fato que apenas os pedidos pendentes são atendidos à cada varredura. Os pedidos que chegam durante uma varredura são agrupados e ordenados para serem atendidos no retorno da varredura. Proporciona boa performance e bom tempo de resposta médio com pequena variância, não existindo a possibilidade de adiamento infinito para quaisquer pedidos [DEI92, p. 368].

- Eschenbach

A movimentação é semelhante ao C-SCAN, mas cada cilindro tem sua trilha percorrida por completo, sendo os pedidos reordenados durante este percurso. Para pedidos em setores sobrepostos apenas um é atendido na varredura corrente. É uma estratégia para otimização do atraso rotacional, embora o C-SCAN se prove melhor [DEI92, p. 365].

Em situações de grande carga de trabalho, deve ser considerada a otimização rotacional, ou seja, dado que aumentam as possibilidades de dois ou mais pedidos se referenciam a mesma trilha. Uma estratégia semelhante ao SSTF para a otimização rotacional é o algoritmo SLTF (*shortest latency time first*) que procura atender os pedidos com menor atraso rotacional dentro da trilha corrente [DEI92, p. 370].

Veja na Figura 5.12 uma comparação destes algoritmos de escalonamento de disco supondo um disco de 40 cilindros, com a cabeça inicialmente sobre o cilindro número 11, cujos pedidos indicam cilindros: 2, 36, 16, 34, 9 e 12. Os gráficos representam a movimentação da cabeça de leitura segundo um algoritmo específico para atender a série dada de pedidos.

Se considerarmos que entre duas trilhas distintas existe um percurso que pode ser calculado por:

$$Percurso = |trilha_{final} - trilha_{inicial}| \quad (5.2)$$

Então o percurso total realizado para atendimento de n pedidos é a somatória dos percursos individuais, ou seja:

$$Percurso_{total} = \sum Percurso_i \quad (5.3)$$

As unidades de discos flexíveis (*floppies* comuns e ZIP disks) também funcionam segundo os mesmos princípios e possuem a mesma organização, exceto pelo fato de possuírem cabeças de leitura/escrita de contato e que utilizam o algoritmo de escalonamento de disco FCFS. Na Tabela 5.2 temos a organização de um disco flexível de 3.5" com quádrupla densidade [NOR89, p. 100].

Tabela 5.2: Organização de um *floppy* de 1.44 Mbytes

Número de Superfícies	2
Número de Cilindros	80
Número de Trilhas por Cilindro	2
Número de Setores por Trilha	9
Tamanho do Setor (<i>bytes</i>)	512
Capacidade Total (<i>bytes</i>)	1.474.560

Na trilha 0 do lado 0 de um disquete de quádrupla densidade temos 9 setores ocupados respectivamente por: um setor de boot, quatro setores destinados à tabela de alocação de arquivos (FAT propriamente dita, como veremos na seção ???) e quatro setores para a organização do diretório do disquete. Na trilha 0 do lado 1 temos os três últimos setores destinados à organização do diretório e seis setores destinados aos dados. Nas demais trilhas do disquete teremos nove setores de dados cada.

O registro de *boot* sempre está localizado na trilha 0 e setor 0 de disquetes. Nele pode ser colocado um pequeno programa destinado a iniciar o processo de carregamento do sistema operacional, possibilitando que este disquete possa dar a *partida* do sistema. Cada sistema operacional possui um formato diferente para o conteúdo do registro de *boot*, mas a idéia central é sempre a mesma.

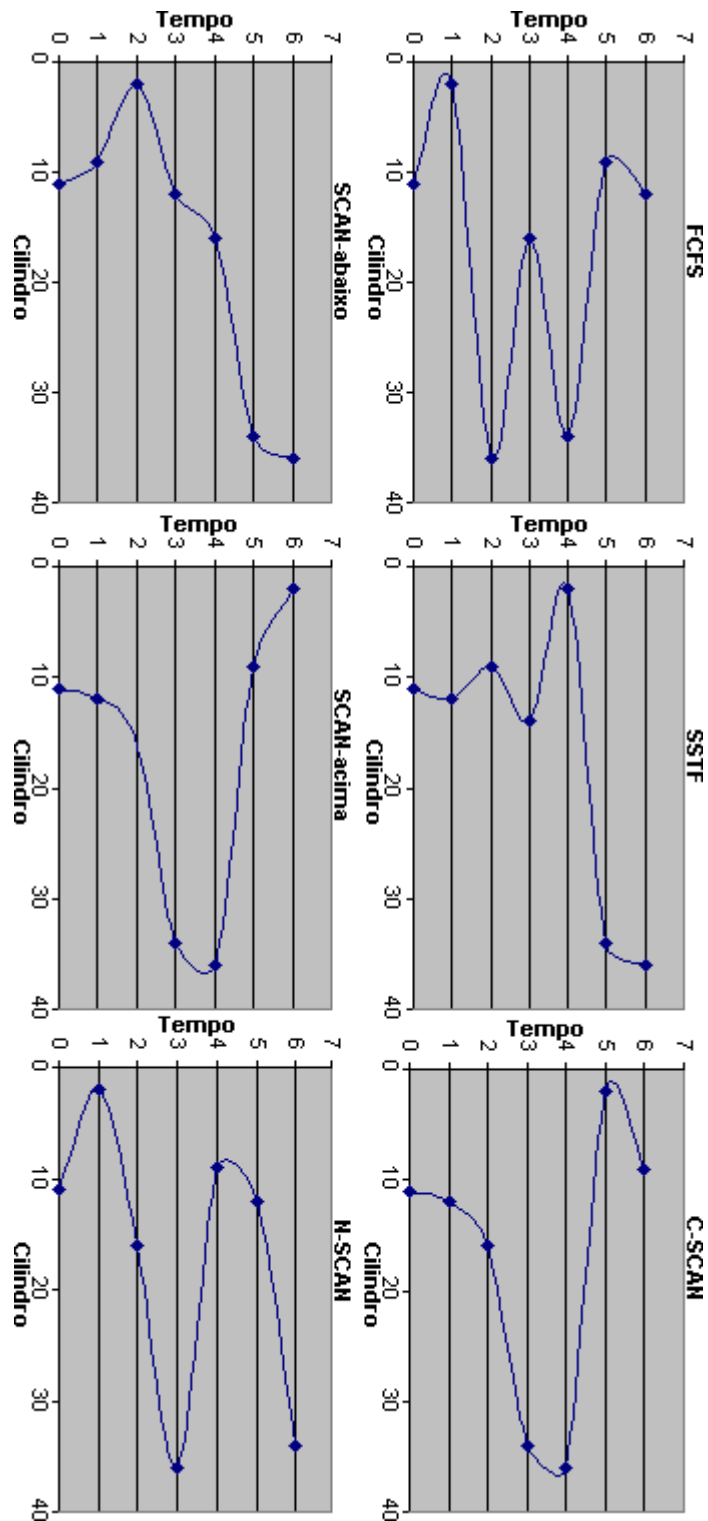


Figura 5.12: Comparação dos algoritmos de escalonamento de disco

5.4.3 Unidades de fita

As unidades de fita magnética são outro importante dispositivo de I/O tendo sido os primeiros periféricos utilizados para o armazenamento secundário nos sistemas computacionais. Utiliza os mesmos princípios físicos para o armazenamento perene de dados: uma fina fita plástica (de Mylar ou material semelhante) é recoberta com algum óxido metálico cujas propriedades magnéticas possibilitam que através de uma cabeça leitura se realize a gravação ou leitura de dados na fita.

As fitas, acondicionadas em rolos ou cartuchos de vários tipos e tamanhos, são movimentadas tal como gravadores de áudio de fita em rolo ou cassete, embora tipicamente com maior velocidade. A cabeça leitura, que trabalha em contato direto com a fita plástica, grava simultaneamente vários *bits* de informação na fita, paralelamente ao sentido do comprimento da fita, de forma que nesta sejam identificadas **trilhas de gravação** (*tracks*). O número de trilhas existente é variável, dependendo do tipo de unidade de fita e da fita magnética que utiliza, mas valores usuais são 9, 18 ou 36 trilhas, para a gravação paralela de um *byte*, *word* ou *double word* por vez respectivamente, juntamente com alguns *bits* extras utilizados para o controle de paridade [STA96, p. 174].

Cada bloco de informação gravada, 128, 256 ou 512 bytes, é identificado como um **registro físico** (*physical record*), separado do registro anterior e posterior por espaços denominados espaços **inter-registro** (*inter-record gaps*). Veja na Figura 5.13 a representação esquemática da organização de uma fita magnética contendo 9 trilhas.

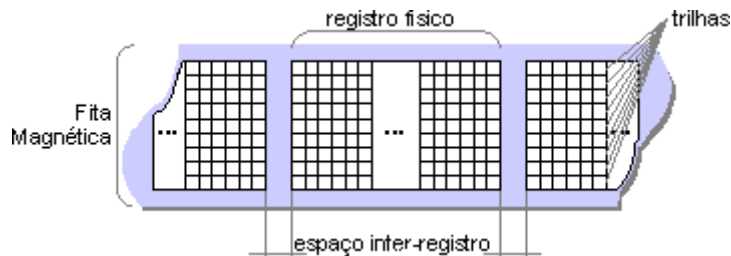


Figura 5.13: Formato esquemático de fita magnética

Diferentemente das unidades de disco, que são dispositivos de acesso direto, as unidades de fita magnética são conhecidas como dispositivos de **acesso seqüencial**, pois para efetuar-se a leitura de um determinado registro físico, todos os registros existentes entre a posição inicial da cabeça de leitura e o registro desejado devem ser lidos. Se o registro desejado estiver localizado numa posição anterior a posição corrente, então a fita ou cartucho deverão ser rebobinados até o seu início ou posição anterior adequada para que a pesquisa pelo registro desejado possa ser iniciada. Claramente temos que as unidades de fita são dispositivos mais lentos que as unidades de disco.

O **tempo de acesso** (*access time*) para uma unidade de fita magnética também é dado pela soma de três componentes de tempo, representados na Figura 5.14:

1. o tempo necessário para a cabeça de leitura começar a ser movimentada, chamado de **tempo de atraso de movimentação** (*move delay*);
2. o tempo necessário para mover-se até o registro físico, ou seja, o **tempo de acesso ao registro** (*seek time*) e
3. o **tempo de transferência dos dados**, isto é, o tempo decorrido nas operações de leitura ou escrita dos dados no registro (*transfer time* ou *transmission time*).

Sendo assim, o tempo de acesso pode ser determinado como indicado na Equação 5.4.

$$t_{access} = t_{movedelay} + t_{seek} + t_{transfer} \quad (5.4)$$

Devido ao fato das unidades de fita magnética terem cabeças de leitura de contato, isto é, que para lerem os dados são mantidas em contato com a fita, a movimentação da fita só ocorre durante as operações de leitura ou escrita minimizando tanto o desgaste da fita magnética como da cabeça de leitura. Daí a existência do atraso de movimentação que corresponde ao tempo necessário para o acionamento do mecanismo de movimentação da fita. Veja na Figura 5.14 uma representação esquemática dos componentes do tempo de acesso de uma unidade de fita.

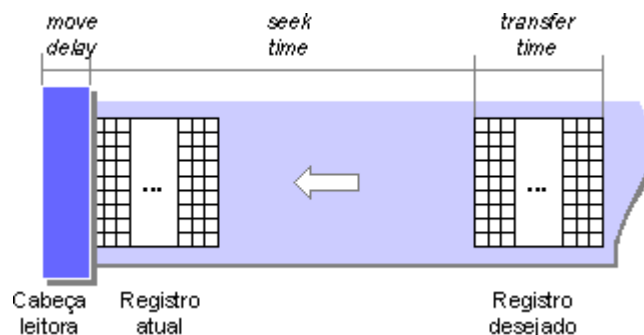


Figura 5.14: Componentes do tempo de acesso de uma unidade de fita

Nos discos flexíveis ocorre o mesmo que nas unidades de fita: a movimentação só ocorre durante as operações de leitura e escrita, pois a cabeça de leitura também é do tipo de contato, sendo que depois da operação a

movimentação é interrompida. Nos discos rígidos, o mecanismo de movimentação dos discos permanece ativo durante todo o tempo, exceto nos casos de economia de energia, onde o mecanismo de acionamento é desativado para redução do consumo, típico em computadores portáteis ou pessoais.

Em sistemas computacionais de grande porte, podem existir procedimentos especiais para que unidades de fita magnética, geralmente próximas ao computador central, possam ser utilizadas pelos usuários do sistema.

Apesar da velocidade bastante mais baixa, as unidades de fita magnética ainda são largamente utilizadas, pois a mídia utilizada é removível e barata. Atualmente os formatos de fita em rolo estão caindo em desuso, sendo preferidos os formatos tipo cartucho, cujas capacidades de gravação podem ser superiores a 1 GByte por cartucho, configurando um dispositivo de armazenamento secundário compacto e de baixíssimo custo por MByte. Devido à velocidade e custo, as aplicações mais comuns das fitas magnéticas são a realização de cópias de segurança (*backups*) e o transporte manual de grandes quantidades de dados (por exemplo, RAIS, IR, folha de pagamento, relação de recebimento para concessionárias de serviços públicos etc).

5.4.4 Terminais

Todos os sistemas computacionais, de qualquer porte, que oferecem serviços interativos para vários usuários simultâneos, utilizam terminais para realizar a comunicação entre os usuários e os computadores. Os terminais também são denominados TTY (*teletype*), devido a uma empresa de mesmo nome que tornou-se conhecida pelo seus equipamentos. Existem vários tipos diferentes de terminais os quais empregam diferentes tecnologias e estratégias para seu funcionamento. Na Figura 5.15 temos uma classificação dos terminais, semelhante a sugerida por Tanenbaum [TAN92, p. 227].

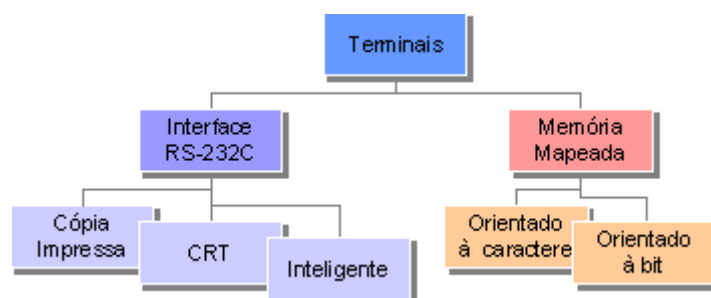


Figura 5.15: Tipos de terminais

Os terminais baseados na interface RS-232 utilizam um protocolo de comunicação serial para a transmissão de dados entre o computador e o terminal, onde um único caractere é transmitido por vez, a taxa de 1200 a 9600 bps (*bits* por segundo), como ilustrado na Figura 5.16. Esta forma

de comunicação se tornou tão comum que vários fabricantes desenvolveram circuitos integrados (chips) especializados nesta tarefa, chamados de UARTs (Universal Asynchronous Receiver Transmitter), entre os quais citamos o popular Intel 8255. Apenas a transmissão de um único caractere a 9600 bps toma aproximadamente 1 ms, portanto uma tela de 25 linhas por 80 colunas (2000 caracteres), tomaria 2 segundos para ser preenchida completamente.

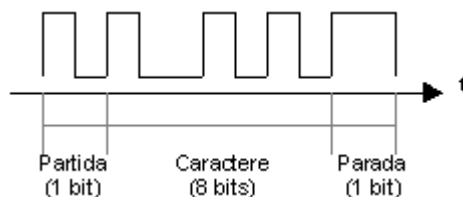


Figura 5.16: Protocolo de comunicação serial

Os terminais que utilizam este protocolo são compostos por um teclado, um monitor de vídeo e alguns circuitos eletrônicos que implementam a lógica de controle necessária ao seu funcionamento, sendo portanto equipamentos de custo relativamente baixo. A UART interna do terminal é conectada à interface RS-232 do computador via um cabo de três fios (terra, transmissão e recepção). A interface RS-232 do computador deve possuir tantas conexões e UARTs quantos terminais a serem conectados, como ilustrado na Figura 5.17. Tal forma de conexão, com o passar do anos, começou a se tornar um inconveniente dado o número crescente de terminais de um sistema computacional. O funcionamento desta conexão pode ser descrito da seguinte maneira:

1. Quando o terminal deseja enviar um caractere ao computador (para estabelecer uma conexão com o sistema ou para enviar dados e comandos do usuário), ele o envia para sua UART.
2. Quando a UART efetua a transmissão do caractere, gera uma interrupção indicando que pode transmitir outro caractere.
3. A UART do computador recebe o caractere, processando-o, enviando uma resposta quando devido.
4. A UART recebe o caractere, gerando uma interrupção para o processamento do caractere no terminal.

Os terminais de cópia impressa (*hardcopy*), percursos dos terminais de vídeo, imprimem em papel todas as mensagens recebidas do computador. Através de seu teclado as mensagens são enviadas do terminal para o computador. Os terminais CRT (*catodic ray tube*) são os terminais que utilizam monitores de vídeo para exibição das mensagens, operando da mesma forma que os terminais *hardcopy*.

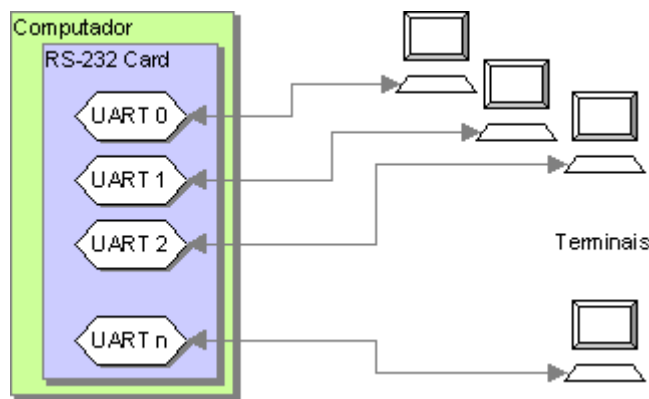


Figura 5.17: Conexão computador e terminais RS-232

Os terminais inteligentes são pequenos computadores especializados na transmissão e recepção de mensagens, possuindo inúmeras funções de configuração, possibilitando sua configuração para o trabalho com diferentes computadores em situações diversas. Dada sua capacidade interna de processamento, podem armazenar seqüências de dados ou realizar tarefas mais complexas.

Outra categoria dos terminais são aqueles que possuem memória mapeada, isto é, cuja interface com o computador principal se dá através de sua memória de vídeo (vídeo RAM) e um controlador especial denominado controlador de vídeo (vídeo controller). Nesta situação, o terminal deve possuir capacidade interna de processamento, podendo se comunicar com o computador central através de outras formas mais eficientes tais como os protocolos BSC (*binary synchronous control*) introduzido pela IBM em 1961, o padrão ISO HDLC (*high-level data link control*) ou o SDLC (*synchronous data link control*), versão IBM do protocolo HDLC. Black [BLA87] traz detalhes destes protocolos, comparando-os e mostrando suas aplicações.

A família IBM 3270 é um conjunto de terminais, impressoras e controladoras de terminais tipicamente utilizados em sistemas de grande porte IBM, sendo bastante conhecidas em diversos segmentos corporativos. Os terminais de vídeo IBM 3178 e 3278 são seus integrantes os mais conhecidos, com alguns milhões de unidades vendidas [BLA87, p. 83]. Estes terminais, conectados através de cabos coaxiais, utilizam um protocolo proprietário da IBM para comunicação com controladoras de terminais (IBM 3172 e 3174), que concentravam tal comunicação para um computador central, possibilitando inúmeras configurações, como ilustrado na Figura 5.18.

O processador do terminal, tendo recebido as informações do computador central através de sua conexão dedicada, coloca as informações que devem ser exibidas na memória de vídeo enquanto o controlador de vídeo gera o sinal correspondente ao conteúdo da memória de vídeo, produzindo a imagem

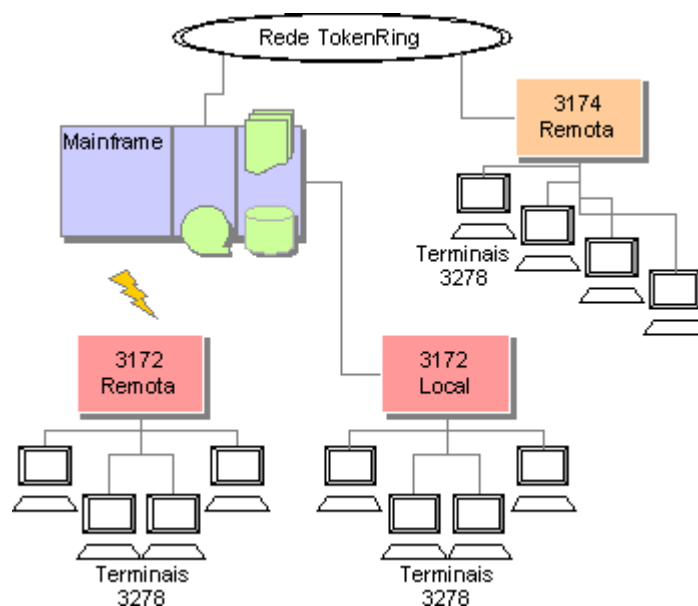


Figura 5.18: Família IBM 3270

adequada, liberando o processador de tal tarefa. O princípio é o mesmo para os terminais orientados à caractere, onde a memória de vídeo é preenchida com os caracteres e atributos para sua exibição; ou orientados à *bit*, onde a memória de vídeo é preenchida com os pontos (*pixels*) que devem ser renderizados.

Nos terminais orientados à caractere apenas telas constituídas de caracteres e uma limitada quantidade de símbolos gráficos podem ser compostas e exibidas, restringindo as suas possibilidades de utilização. Já os terminais orientados à *mphbit* permitem a exibição de gráficos e imagens, caracteres usando múltiplos fontes, sistemas de janelas, ícones, cuja **resolução** (quantidade total de pixels que podem ser apresentados numa única imagem) depende da quantidade de memória de vídeo, das capacidades do monitor de vídeo e do meio de transmissão. As telas são reproduzidas em ciclos de 20 ms (terminais monocromáticos) e 17 ms (terminais coloridos) permitindo a rápida exibição de caracteres. O teclado, nesta configuração também torna-se completamente independente, interrompendo o processador do terminal para enviar seqüências de caracteres.

Os microcomputadores IBM PC e compatíveis utilizam esta estratégia para exibição de seus vídeos, podendo possuir controladores de vídeo bastante sofisticados, destinados a reprodução de imagens em alta velocidade (vídeo) ou renderização em 2D e 3D.

5.5 Sistemas de arquivos

Como vimos na seção 4.3, os dispositivos de armazenamento secundário são utilizados para armazenar dados de forma *perene*, isto é, o armazenamento é feito de forma confiável e íntegra, mesmo quando o sistema computacional permanece desligado por longos períodos de tempo. O armazenamento *perene* de informações é desejado pelas seguintes razões [TAN92, p. 145]:

- Existem inúmeros problemas práticos que requerem o armazenamento de grandes quantidades de informações, as quais não serão utilizadas totalmente por um certo processo, por exemplo, consultas a uma lista telefônica, movimentação de uma conta corrente, recuperação de um histórico escolar, alteração de um cadastro médico, atualização de uma ficha de estoque etc. É inadmissível que tais informações sejam re-introduzidas no computador a cada uso.
- Existem uma outra infinidade de problemas práticos que produzem dados durante sua execução, razão de sua existência, que não podem ser descartados pois serão utilizados por outros processos, tal como o cruzamento de dados informados em declarações de imposto de renda, o resultado de simulações físicas/químicas/matemáticas, a solução de problemas diversos etc.
- Certos conjuntos de dados não podem pertencer a um único processo, isto é, serem dispostas no espaço de endereçamento de um simples programa, devendo ser compartilhada por muitos outros processos, tal como os dados de lotação de aviões de uma companhia aérea, as informações de tarifação de uma companhia telefônica etc.
- Os sistemas computacionais estão sujeitos à falhas ou situações de contingência, onde é importante que seus dados estejam preservados (duplicados) de forma segura, permitindo sua recuperação e uso em um outro sistema computacional.

Todas estas razões justificam a necessidade do armazenamento de informações em meios distintos da memória primária, que tem capacidade limitada, e de forma independente dos processos, dada as questões de persistência e compartilhamento, sem considerarmos as questões relacionadas a integridade e segurança dos dados.

Para que o armazenamento possa ser realizado de maneira tal a atender as razões enunciadas, é necessária uma adequada organização destes dados nos dispositivos destinados ao armazenamento secundário, tal como unidades de disco, unidades de fita magnética, CD-ROMs etc.

Dado que podem existir muitos processos num sistema computacional, cada um utilizando e produzindo diferentes conjuntos de dados, torna-se

necessário distinguir tais conjuntos de dados. Os **arquivos** (*files*) são as unidades que contêm estes conjuntos distintos de dados, de forma que estes possam ser utilizados pelos processos.

Como tudo mais num sistema computacional, o sistema operacional controla as operações sobre os arquivos, organizando seu armazenamento no que chamamos de **sistema de arquivos** (*file system*). Um sistema de arquivos geralmente contém [DEI92, p. 389]:

- Métodos de acesso: forma com que os dados são armazenados nos arquivos;
- Gerenciamento de arquivos: conjunto de mecanismos de armazenamento, referência, compartilhamento e segurança;
- Mecanismos de integridade: que asseguram que os dados de um arquivo permanecem íntegros.

Ainda segundo Deitel [DEI92, p. 391], um sistema de arquivos deve permitir, funcionalmente falando, que:

- os usuários possam criar, modificar e eliminar arquivos, bem como realizar sua duplicação ou a transferência de dados entre arquivos;
- seja possível o compartilhamento de arquivos através de mecanismos controlados e bem definidos;
- as operações de *backup* e sua restauração sejam suportadas;
- seja possível a adoção ou implementação de procedimentos de proteção e segurança; e
- que a interface seja amigável e consistente, admitindo a realização de operações apenas através dos nomes simbólicos dos arquivos, garantindo independência do dispositivo utilizado.

Com isto percebemos que os sistemas de arquivos preocupam-se com a organização e controle do armazenamento secundário de um sistema computacional. Também é necessário destacar que um sistema operacional pode suportar diversos sistemas de arquivos, isto é, podem operar utilizando diferentes formas de administração dos dispositivos de armazenamento secundário. A seguir, na Tabela 5.3, alguns exemplos de sistemas operacionais e dos sistemas de arquivos suportados.

Tabela 5.3: Sistemas de arquivos de alguns sistemas operacionais

Sistema operacional	Sistemas de arquivos suportados
MS-DOS	FAT (<i>file allocation table</i>) 12 e 16 bits
MS-Windows 95/98	FAT (<i>file allocation table</i>) 12 e 16 bits VFAT (<i>virtual file allocation table</i>) 32 bits
MS-Windows NT MS-Windows 2000 MS-Windows XP	FAT (<i>file allocation table</i>) 12 e 16 bits VFAT (<i>virtual file allocation table</i>) 32 bits NTFS (<i>new technology file system</i>)
IBM OS/2	FAT (<i>file allocation table</i>) 12 e 16 bits HPFS (<i>high performance file system</i>)
Linux	FAT (<i>file allocation table</i>) 12 e 16 bits VFAT (<i>virtual file allocation table</i>) 32 bits Minix (Mini Unix) Extended File System Ext2 (second extended file system - Nativo) Sun Solaris UFS

5.5.1 Arquivos

Existem várias definições possíveis para o conceito de arquivos. Tanenbaum afirma que, de forma simplificada, os **arquivos** podem ser entendidos como seqüências de *bytes* não interpretadas pelo sistema, dependendo-se de aplicações apropriadas para sua correta utilização [TAN95, p. 246]. Deitel coloca que arquivos são uma coleção identificada de dados [DEI92, p. 389], enquanto Guimarães explica:

Um arquivo é um conjunto de informações relacionadas entre si e residentes no sistema de memória secundária: discos, fitas, cartões, etc [GUI86, p. 211].

Outras definições poderiam ser citadas, mas o que é mais importante é o conceito inerente à utilização dos arquivos. Arquivos são um poderoso mecanismo de abstração que permite ao usuário (e seus programas) utilizar dados armazenados dentro do sistema computacional, ou seja, através da manipulação dos arquivos são realizadas as operações de escrita e leitura de dados, de forma transparente, evitando que sejam conhecidos detalhes do funcionamento com que estas operações tratam e armazenam a informação [TAN92, p. 146].

Os arquivos, dependendo do sistema operacional e do sistema de arquivos em uso, podem possuir uma identificação (*file naming*), atributos (*attributes*), capacidades (*capacities*), lista de controle de acesso (*control access list*) e uma organização ou tipo.

Para que os dados possam ser corretamente identificados e utilizados, o sistema de arquivos deve prover um mecanismo de identificação dos arquivos, isto é, uma forma de distinção simples dos arquivos, que permita

sua adequada operação. Isto é feito através da associação de um nome ao arquivo propriamente dito. Desta forma muitas das operações que podem ser realizadas sobre os arquivos são especificadas em termos de seus nomes, simplificando muito tais tarefas.

Diferentes sistemas operacionais usam formas de denominação distintas para a identificação de seus arquivos, como exemplificado na Tabela 5.4.

Tabela 5.4: Denominação de arquivos em vários sistemas

Sistema operacional	Composição do nome	Composição da extensão	Outras características
MS-DOS (FAT12 e FAT16)	1-8 char	1-3 char	Extensão opcional Insensível ao caixa
MS-Windows 95, 98, 2000, XP (VFAT)	1-255 char	1-255 char	Admite múltiplas exts Comprimento total < 256 Insensível ao caixa
MS-Windows NT (NTFS)	1-255 char	1-255 char	Admite múltiplas exts Comprimento total < 256 Insensível ao caixa
IBM OS/2 (HPFS)	1-255 char	1-255 char	Extensão opcional Comprimento total < 256 Insensível ao caixa
UNIX (genérico)	1-255 char	1-255 char	Não usa extensões Comprimento total < 256 Sensível ao caixa

Note que o UNIX, dado possuir um sistema de arquivos com denominação sensível ao caixa, trataria como sendo diferentes arquivos com os nomes `ab`, `AB`, `Ab` e `aB`. Tal característica, embora útil, pode se tornar um pesadelo em certas situações quando não se percebe a sutileza de um único caractere com o caixa errado. Nos sistemas que não utilizam extensões ou admitem múltiplas extensões, podem surgir denominações tais como `names.dat.old` ou `source.c.zip`, indicando que diferentes ações foram tomadas com os arquivos.

Além dos nomes os arquivos podem possuir atributos, isto é, informações que não fazem parte dos arquivos, embora associadas aos mesmos, que podem ser utilizadas para indicar: criador (*creator*), proprietário (*owner*), tamanho (*size*), data de criação (*creation date*), data do último acesso (*last access date*), data da última alteração (*last modification date*), flags diversas (de sistema, ocultação, de arquivo, de leitura etc) permissões de acesso, tamanho do registro, tamanho máximo, tipo etc. As permissões de acesso são usadas em conjunto com as capacidades ou listas de controle de acesso.

As capacidades são informações que autorizam certos usuários ou processos a realizarem certas operações sobre os arquivos, tal como leitura e escrita. As listas de controle de acesso são relações de usuário que podem realizar operações específicas sobre um dado arquivo, tal como execução ou

escrita. Enquanto as capacidades são associadas aos usuários ou processos as listas de acesso são associadas diretamente aos arquivos.

Os sistemas UNIX utilizam-se de uma lista de acesso simplificada baseada no conceito de grupos. Permissões são associadas a todos os arquivos e podem ser diferenciadas em três níveis: o do proprietário, o do grupo ao qual pertence seu proprietário e dos demais usuários. Para cada nível podem ser especificadas separadamente permissões para leitura, escrita e execução, que quando combinadas resultam em diferentes modos de acesso. Já o MS-DOS associa apenas um mbit de permissão de escrita (*Read-Only*), ao qual se associam os atributos Oculto (*Hidden*), Sistema (*System*) e Arquivo (*Archive*), que na prática representam um precário esquema de proteção contra eliminação indevida.

Existem diversas estruturas possíveis para os arquivos tais como seqüencial, por registros e em árvore [TAN92, p. 148], tal como ilustrado na Figura 5.19.

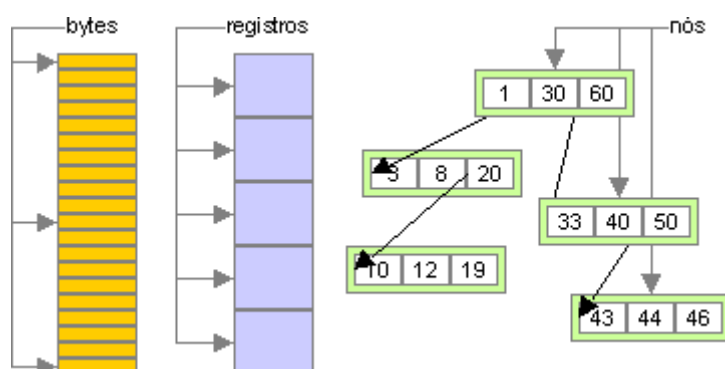


Figura 5.19: Estruturas possíveis de arquivos

Na estrutura seqüencial, um arquivo é uma seqüência de bytes, garantindo a máxima flexibilidade ao usuário do sistema operacional que oferece operações bastante simples. Na estrutura de registro, um arquivo é uma seqüência de registros, os quais podem ter diversos formatos. Esta estrutura é um pouco menos flexível e depende de operações um pouco mais sofisticadas do SO. Existem diversos arranjos para arquivos organizados como seqüências de registros: registro de tamanho fixo desbloqueados, registro de tamanho fixo bloqueado, registro de tamanho variável desbloqueados e registro de tamanho variável bloqueado [DEI92, p. 393].

Sistemas operacionais como DOS, MS-Windows 95, 98 e NT e Unix estruturam seus arquivos como seqüências de *bytes*. Computadores de grande porte, tipicamente os sistemas IBM utilizam, como organização mais comum, as seqüências de registro. A organização em árvore é raramente implementada.

Dependendo do conteúdo do arquivo este pode ser entendido como de um certo tipo. Muitos sistemas operacionais utilizam a extensão do nome do arquivo como uma referência para o tipo de seu conteúdo. Particularmente nos sistemas operacionais MS-Windows 95, 98, NT, 2000 ou XP podem ser associadas aplicações a determinadas extensões, auxiliando o usuário. Genericamente arquivos comuns, isto é, aqueles utilizados pelos usuário para o armazenamento de informações, podem ser considerados como do tipo **texto**, quando seu conteúdo pode ser visualizado diretamente (via comandos DOS `type` [IBM92a] ou Unix `more` [CON99, p. 132]); ou sendo **binário**, quando seu conteúdo precisa ser *interpretado* por uma aplicação para poder ser visualizado.

Os arquivos **executáveis** também são arquivos **binários**, cuja execução é realizada diretamente pelo sistema operacional. Vale ressaltar que um arquivo binário não é simplesmente uma seqüência de instruções que podem ser executadas pelo processador, além do código executável propriamente dito o sistema operacional necessita conhecer informações adicionais sobre o tamanho do programa, sua área de dados, o tamanho necessário para sua pilha de retorno, etc.

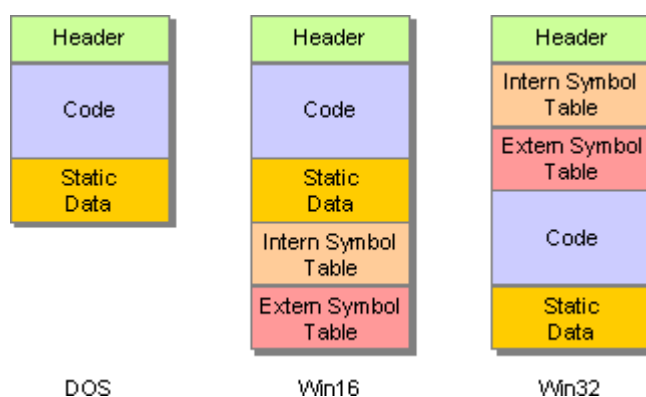


Figura 5.20: Formato de arquivos executáveis no Win-32

Geralmente estas informações são posicionadas num cabeçalho (*header*) do arquivo sendo que cada sistema operacional possui um formato próprio de arquivo identificado como executável. Na Figura 5.20 encontramos os formatos de arquivos executáveis dos sistemas operacionais DOS, Windows 3.x e Windows 95 enquanto que Figura 5.22 mostra a estrutura de um arquivo executável no sistema Unix com detalhes de seu cabeçalho. Especificamente no sistema operacional Windows 95/98 e Windows NT é possível visualizar-se detalhes internos de arquivos executáveis e bibliotecas de vínculo dinâmico (DLLs ou *dynamic link libraries*) tais como cabeçalho da imagem, cabeçalho opcional, tabela de importação, tabela de seções e informações do cabeçalho. Isto é possível através da opção de *visualização rápida* oferecida pelo gerenciador de arquivos nativo, como ilustrado na Figura 5.21.

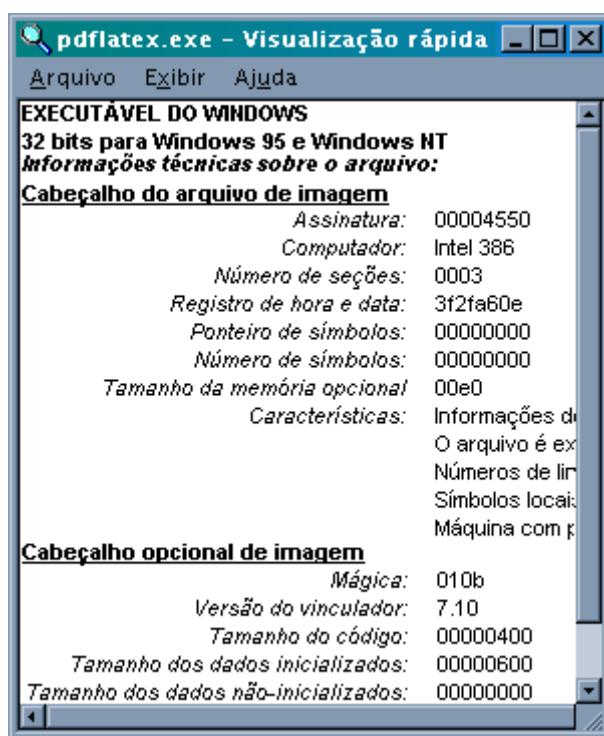


Figura 5.21: Visualização rápida de executáveis no MS-Windows 98

Além dos arquivos comuns, os sistemas de arquivos geralmente mantêm arquivos especiais denominados **diretórios** (*directories*) que contêm partes da estrutura do sistema de arquivos. Em sistemas Unix ainda existem arquivos especiais utilizados para modelar certos dispositivos periféricos, podendo ser arquivos especiais de caracteres (*character special files*) ou arquivos especiais de blocos (*block special files*) que permitem o acesso terminais, impressoras e rede no primeiro caso, discos, fitas e outros dispositivos de armazenamento secundário no segundo.

Finalmente, do ponto de vista de armazenamento e acesso, os arquivos podem ser organizados das seguintes maneiras [DEI92, p. 392]:

Seqüencial Quando os registros ou bytes são posicionados em sua ordem física. Numa fita isto representa armazenamento contíguo, embora em discos magnéticos isto não seja necessariamente verdade.

Direto Quando os registro ou bytes são diretamente acessados no meio em que são armazenados, usualmente um DASD. A aplicação deve conhecer a localização dos dados no dispositivo, sendo familiar com sua organização. É um método extremamente rápido, embora complexo.

Seqüencial Indexado Os registros bytes são organizados numa seqüência lógica conforme uma chave e o sistema mantêm um índice para acelerar

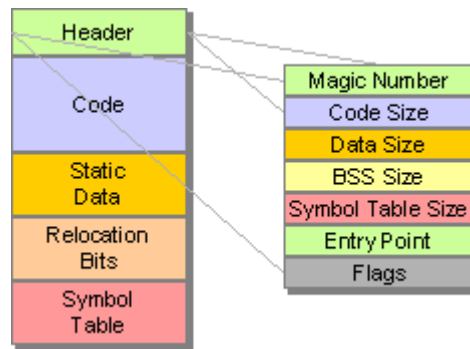


Figura 5.22: Formato de arquivos executáveis (*binaries*) no Unix

o acesso a determinadas partes de um arquivo. Esta é uma forma de organização comum em tabelas de bancos de dados.

Particionado Quando o arquivo é composto de subarquivos denominados membros. Para acessar seus membros (*members*), o arquivo particionado possui um diretório, que funciona como seu índice. São utilizados para armazenar bibliotecas ou bancos de dados.

5.5.2 Diretórios

Um **diretório** (*directory*) nada mais é que um arquivo mantido pelo sistema de arquivos que contém uma lista de arquivos e possivelmente outros diretórios. Desta forma é possível criar-se estruturas hierárquicas de arquivos, onde os diretórios existentes dentro de outros são denominados **subdiretórios**, tal como nos sistemas Unix, OS/2, DOS, Windows e outros.

O que motivou a criação dos diretórios é o fato de que torna-se difícil a organização de um número grande de arquivos armazenados num determinado dispositivo sem qualquer forma de divisão. Os primeiros dispositivos de armazenamento secundário tinham pequena capacidade o que resultava no armazenamento de um pequeno número de arquivos, catalogados num diretório único associado ao próprio dispositivo. A medida que a capacidade de armazenamento destes dispositivos cresceu, as unidades de disco passaram a armazenar vários milhares de arquivos, assim a apresentação de uma simples listagem dos arquivos existentes significaria para o usuário visualizar muitas telas repletas de nomes.

O sistema operacional DOS, em sua versão 1.0 lançada em 1981, suportava apenas um diretório único por dispositivo, usualmente discos flexíveis de limitada capacidade. Com a introdução de suporte para discos rígidos de maior capacidade na versão 2.0 liberada em 1983, foram adicionadas capacidades para criação e manutenção de diretórios e subdiretórios [JAM87, p. 25].

A criação de diretórios permite que os arquivos sejam divididos e organizados conforme seu tipo, proprietário, utilização ou qualquer outro critério, possibilitando seu acesso de forma mais simples. Como mostra a Figura 5.23, existem várias organizações possíveis de diretórios, tais como em um nível, em dois níveis ou em múltiplos níveis (ou em árvores).

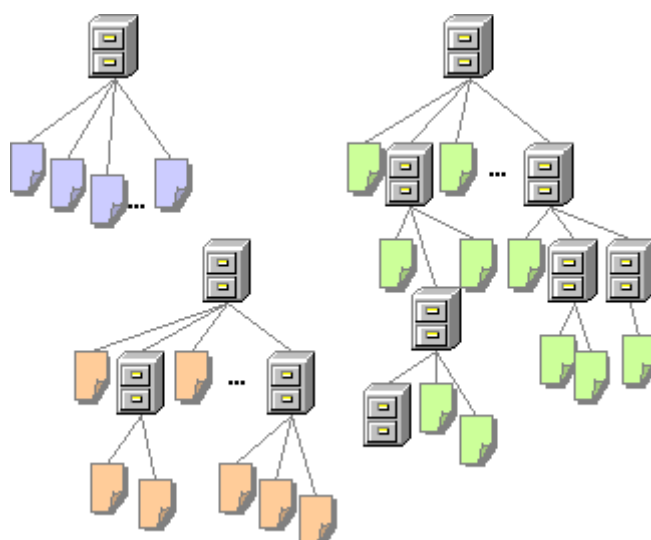


Figura 5.23: Organizações Possíveis de Diretórios

Também é possível organizar-se os diretórios como grafos acíclicos (*acyclic graph directories*) que permite que um arquivo ou mesmo um diretório seja apontado por múltiplas entradas presentes em diferentes diretórios, ou seja, que tal arquivo ou diretório *simultaneamente* presentes como conteúdo de vários diretórios ao mesmo tempo, permitindo seu compartilhamento [SG00, p. 356]. Isto é comum em sistemas UNIX, onde podem ser criados *links* apontando os arquivos ou diretórios compartilhados.

Internamente os diretórios são organizados como um conjunto de entradas, uma para cada arquivos ou subdiretório que contêm. A cada uma destas entradas são associados atributos que descrevem as permissões associadas aos arquivos, os endereços onde o arquivo está fisicamente armazenado no dispositivo e outras informações tais como tamanho do arquivo, data de criação, etc. como visto na seção 5.5.1.

No sistema de arquivos FAT (*file allocation table*) do MS-DOS e outros sistema operacionais, uma entrada típica de diretório possui a estrutura ilustrada na Tabela 5.5 [JAM87, p. 266] [NOR89, p. 106] [TAN92, p. 167]. Nos sistemas Unix as entradas de diretórios são bem mais simples, como mostra a Tabela 5.6.

Quando são solicitadas operações sobre arquivos, tais como abertura ou criação de um arquivo, o sistema operacional consulta o diretório corrente

Tabela 5.5: Entrada de diretório do sistema de arquivos FAT

Campo	Tamanho (bytes)
Nome do arquivo	8
Extensão	3
Atributos	1
Reservado para DOS	10
Hora	2
Data	2
Número do Setor Inicial	2
Tamanho do Arquivo	4
Total	32

Tabela 5.6: Entrada de diretório de sistemas Unix

Campo	Tamanho (bytes)
Número do I-Node	2
Nome do Arquivo	14
Total	16

(*current directory*), isto é, o diretório que está sendo consultado no momento, verificando a existência do arquivo desejado para abertura ou a possibilidade de criação de um novo arquivo segundo as regras de identificação do sistema.

A existência de uma estrutura de diretórios traz como implicação direta a necessidade de uma forma de denominação única dos arquivos considerando-se o sistema como um todo. Da mesma forma que não é possível a existência de dois arquivos de mesmo nome em um mesmo diretório, dois arquivos ou diretórios quaisquer não podem ter a mesma denominação do ponto de vista do sistema. Esta denominação sistêmica dos arquivos é chamada de especificação completa do caminho do arquivo ou apenas caminho do arquivo (*pathname*) e deve permitir a identificação única de qualquer arquivo ou diretório do sistema.

A especificação de caminho de arquivo pode ser absoluta (*absolute pathname*) ou relativa (*relative pathname*) quando se refere a diretório corrente. Muitas vezes o diretório corrente é tratado como diretório de trabalho (*working directory*).

Os caminhos de arquivos para arquivos ou diretórios são formados pelo nome dos diretórios nos quais estão contidos, sendo que utiliza-se um caractere especial, que não pode ser utilizado na denominação de arquivos e diretórios, como ”

O sistemas operacionais DOS e Windows utilizam-se de uma estrutura de diretórios hierarquicamente organizada, tal como nos sistemas Unix,

diferenciando-se pelo caractere de separação de nomes e principalmente pela forma de denominação de dispositivos. Enquanto que no Unix os dispositivos são tratados como arquivos especiais e mapeados diretamente no sistema de arquivos a partir de diretórios arbitrários, isto é, especificados pelo usuário durante a configuração ou operação do sistema através do comando `mount` [CON99, p. 224], tanto o DOS como o Windows dão uma denominação específica para cada dispositivo físico, chamando-os de **unidades**. Sendo assim, qualquer que seja o dispositivo de armazenamento, ou seja, para *floppies*, discos, CD-R/RWs etc., sempre temos um diretório raiz para cada unidade. Daí a necessidade de adicionar-se a especificação da unidade ao caminho absoluto e também a impossibilidade de especificar-se caminhos relativos envolvendo unidades distintas.

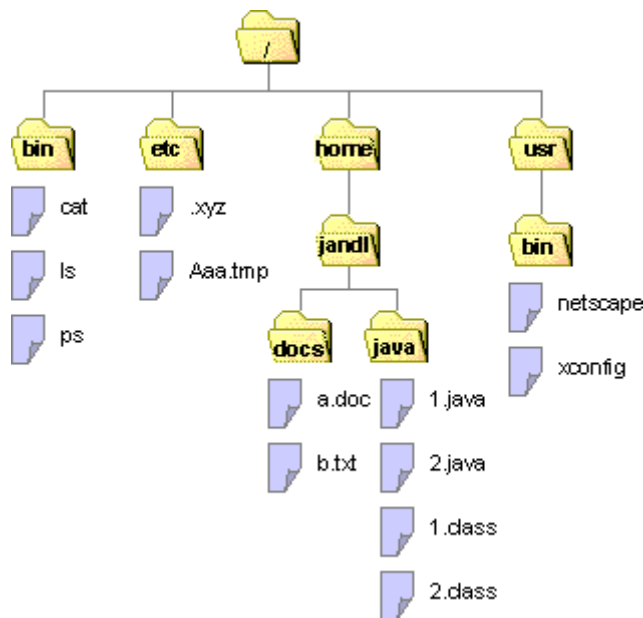


Figura 5.24: Estrutura de diretório de um sistema Unix

Na maioria dos casos, os sistemas operacionais utilizam-se de diretórios particulares para armazenamento de arquivos específicos, como mostra a Tabela 5.7.

5.5.3 Serviços do sistema operacional

O sistema operacional, através de seu sistema de arquivo, deve prover um conjunto de operações para manipulação dos arquivos e de seu conteúdo e também dos diretórios. Do ponto de vista dos arquivos, visto como unidades, devem ser oferecidas as seguintes operações [DEI92, p. 389] [TAN92, p. 153]:

- **Abertura** (*open*) prepara arquivo para uso.

Tabela 5.7: Diretórios específicos

Sistema operacional	Diretório	Propósito
DOS	DOS TMP ou TEMP	Arquivos do sistema Arquivos temporários
Windows 95/98	Windows Windows\System Windows\Temp Windows\Command Arquivos de Programa Meus Documentos	Utilitários do sistema Arquivos do sistema Arquivos temporários Arquivos do DOS Produtos instalados Arquivos do usuário
Unix	usr home bin etc	Programas de usuários Diretórios de usuários Arquivos binários do sistema Arquivos diversos de config.
OS/2	OS2 OS2\System OS2\MDOS	Utilitários do sistema Arquivos do sistema Arquivos do DOS

- **Fechamento** (*close*) encerra o uso do arquivo evitando sua alteração.
- **Criação** (*create*) cria um novo arquivo.
- **Eliminação** (*erase*, *delete* ou *destroy*) remove um arquivo.
- **Renomeação** (*rename*) troca o nome de um arquivo.
- **Cópia** (*copy*) copia o conteúdo de um arquivo para outro.
- **Exibição** (*type* ou *list*) exibe o conteúdo de um arquivo.
- **Catálogo** (*cat* ou *dir*) lista os arquivos existentes num determinado diretório ou unidade.
- **Modificação de atributos** (*get* ou *set*) obtêm ou modifica os atributos de um arquivo.

Do ponto de vista do conteúdo dos arquivos, isto é, considerando a manipulação dos dados armazenados nos arquivos, devem também existir operações para:

- **Leitura** (*read*) possibilita a leitura de dados contidos nos arquivos.
- **Escrita** (*write*) efetua a gravação de dados em um arquivo.
- **Pesquisa** (*seek* ou *find*) para determinar uma posição para escrita ou leitura em um arquivo.

- **Anexação** (*append*) para adição de novos dados num arquivo existente.
- **Inserção** (*insert*) para inclusão de um dado em um arquivo.
- **Atualização** (*update*) para modificação de um dado existente em um arquivo.
- **Eliminação** (*delete*) para remoção de um dado existente em um arquivo.

As operações de inserção, atualização e eliminação de dados em arquivos não são comuns, existindo em sistemas de grande porte onde os arquivos são usualmente organizados em blocos ou registros formatados.

Como muitos sistemas de arquivos suportam diretórios, operações específicas deve ser supridas para sua utilização [TAN92][p. 161] [SG00, p. 350]

- **Criação** (*create*) efetua a criação e preparo de um novo diretório.
- **Remoção** (*delete* ou *remove*) elimina um diretório e opcionalmente seu conteúdo.
- **Abertura** (*open*) operação que permite a leitura de um diretório.
- **Fechamento** (*close*) operação que encerra o uso de um dado diretório.
- **Leitura** (*read*) permite a leitura do conteúdo de um diretório, ou seja, sua catalogação ou listagem.
- **Pesquisa** (*search*) para possibilitar a localização de determinados arquivos em um diretório.
- **Renomeação** (*rename*) troca o nome de um diretório.
- **Navegação** (*traverse*) permite a navegação entre diretórios do sistema.

Ainda podem ser possíveis outras operações sobre arquivos e diretórios, tais como sua reorganização sob algum critério (ordenação), inclusão ou remoção de ligações (*links* no sistema Unix, *atalhos* nos sistemas Windows ou *shadows* nos sistemas OS/2), obtenção de informações sobre arquivos ou diretórios, associação com aplicativos etc.

Todas as operações suportadas pelo sistema operacional para manipulação de arquivos e diretórios estão disponíveis para utilização em programas através das chamadas do sistema (*system calls*), como no exemplo dado no Exemplo 5.1 de algumas chamadas para um programa simples destinado ao sistema Unix.

```
// variável para controle do arquivo
int arquivo;

// criação de arquivo
arquivo = create("nome_do_arquivo", modo);

// abertura de arquivo apenas para leitura
arquivo = open("nome_do_arquivo", O_RDONLY);

// leitura de arquivo
read(arquivo, buffer, tamanho_do_buffer);

// escrita em arquivo
write(arquivo, buffer, tamanho_do_buffer);

// fecha arquivo
close(arquivo);
```

Exemplo 5.1 Fragmentos de programa para sistema Unix

5.5.4 Implementação Lógica

Como implementação lógica de um sistema de arquivos entende-se a metáfora que é apresentada ao usuário do sistema para que o sistema de arquivos se torne compreensível da maneira mais simples possível, isto é, de modo que os detalhes da implementação e funcionamento reais do sistema de arquivos sejam completamente ocultos do usuário do sistema.

Nos sistemas DOS, Windows e Unix, o usuário entende o sistema de arquivos como uma estrutura hierárquica de diretórios que admite operações sobre arquivos e diretórios. Utilizando uma interface de modo texto que oferece uma linha de comando, isto é, uma interface tipo linha de comando ou *prompt* do sistema, o usuário pode digitar comandos para realizar as operações desejadas *imaginando* a organização hierárquica de diretórios e arquivos que não é diretamente visível através destas interfaces simples. Na Tabela 5.8 a seguir temos alguns exemplos de comandos DOS, Windows e Unix para manipulação de arquivos e diretórios.

As interfaces em modo texto exigem maior abstração do usuário, que deve memorizar a sintaxe dos comandos que utiliza mais frequentemente, o que evidentemente significa que tais usuários devem ter maior proficiência técnica para utilização destes sistemas.

Esta dificuldade motivou o desenvolvimento de interfaces gráficas que, entre outros benefícios, possibilitassem:

- uma abstração mais simples do sistema de arquivos,

Tabela 5.8: Comandos DOS, Windows e Unix para manipulação de arquivos

Propósito	DOS	Windows	Unix
Copiar arquivo	copy	copy	cp
Renomear arquivo	ren	ren	mv
Apagar arquivo	del	del	rm
Mover arquivo	N/D	move	mv
Exibir arquivo	type	type	more
Listar arquivos	dir	dir	ls
Criar diretório	md	md	mkdir
Remover diretório	rd	rd	rmdir
Troca de diretório	cd	cd	cd

- uma melhor visualização da distribuição de arquivos através da estrutura de diretórios e unidades de armazenamento,
- maior produtividade do usuário e
- criação de um modelo de interface mais consistente que pudesse ser disponibilizada para outras aplicações.

Para os sistemas Unix foi criado o padrão gráfico de janelas Xwindows, que pode ser utilizado para desenvolvimento de aplicações gráficas, e também o CDE (*Common Desktop Environment*), conduzido por um consórcio dos maiores fabricantes de sistemas Unix num esforço de padronização da interface gráfica de seus sistemas. Outros fabricantes, tais como IBM, Microsoft e Apple oferecem seus sistemas operacionais com interfaces gráficas proprietárias, todas baseadas em janelas.

O tratamento gráfico que especificamente os sistemas MS-Windows dão a estrutura de diretórios, isto é, uma organização hierárquica de *pastas* nas quais podem existir arquivos e outras pastas é a idéia central da metáfora apresentada. Mesmo sem o conhecimento de comandos de navegação na estrutura de diretórios ou de operações sobre arquivos, o usuário dos sistemas MS-Windows pode se movimentar pela estrutura de diretórios, pode copiar, renomear, eliminar e criar arquivos ou diretórios apenas através da utilização do sistema de menus oferecidos ou das operações que podem ser realizadas através do teclado ou mouse. Também é possível executar-se aplicações e criar-se *atalhos* (*links*).

As operações realizadas através dos menus, teclado e *mouse* são transformadas pelo sistema operacional em chamadas do sistema que realizam as tarefas especificadas, evitando tanto o conhecimento da forma com que tais operações são verdadeiramente realizadas como a memorização dos comandos de manipulação de arquivos e diretórios.

Nas Figuras 5.25 e 5.26 temos exemplos dos utilitários de gerenciamento

de arquivos que acompanham os sistemas MS-Windows 95/98/NT e XP respectivamente.

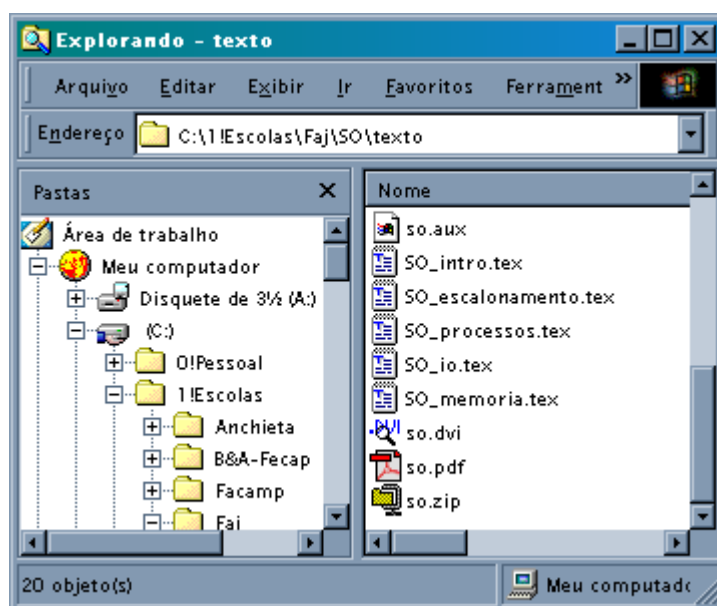


Figura 5.25: Gerenciador de arquivos do MS-Windows 95/98/NT

Estas interfaces gráficas facilitam muito a operação do sistema principalmente para usuários menos experientes, além disso proporcionam melhor visualização da estrutura de diretórios e da distribuição de arquivos pelos diretórios e unidades de armazenamento do sistema.

5.5.5 Implementação Física

A implementação física de um sistema de arquivos representa a forma com que efetivamente os dados são tratados nos dispositivos de armazenamento, isto é, corresponde a organização física dos dados nestes dispositivos e como se dão as operações de armazenamento e recuperação destes dados. Conforme Tanenbaum:

Usuários estão preocupados em como os arquivos são denominados, quais operações são permitidas, a aparência da estrutura de diretórios e questões relacionadas a interface. Os implementadores estão interessados em como arquivos e diretórios são armazenados, como o espaço em disco é gerenciado e como fazer que tudo funcione de forma eficiente e confiável [TAN92, p. 162].

Como indicado por Deitel [DEI92, p. 394] e Tanenbaum [DEI92, p. 162], os arquivos podem ser armazenados basicamente de duas formas, ou seja, através da:

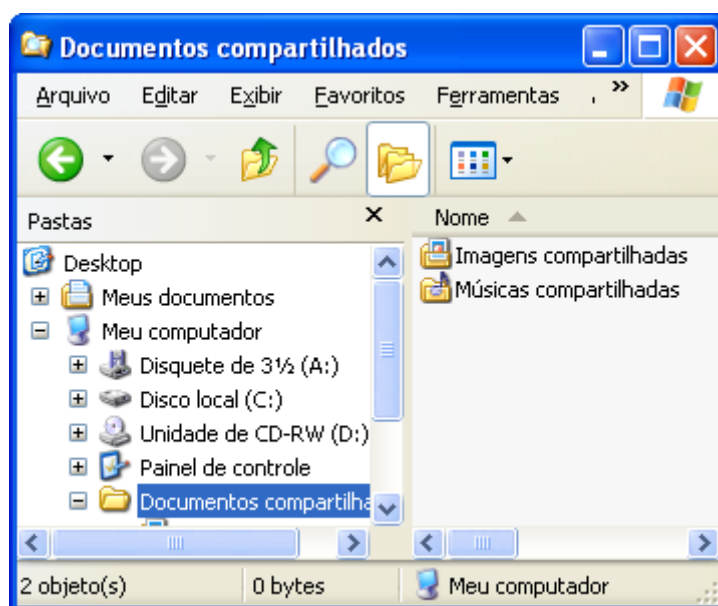


Figura 5.26: Gerenciador de arquivos do MS-Windows XP

- Alocação contígua e
- Alocação não contígua.

O projeto de um sistema de arquivos exige o conhecimento de várias informações, entre elas: o tipo e número de usuários; características das aplicações que serão utilizadas; quantidade, tamanho e operações sobre arquivos. A consideração destes fatores permite determinar qual a melhor forma de organização de seus arquivos e diretórios.

Alocação contígua

Uma forma de organizar-se os arquivos fisicamente é através da armazenagem dos dados em áreas adjacentes dos dispositivos físicos, isto é, em setores consecutivos das unidades de disco [SG00, p. 373]. Sistemas de arquivos implementados desta forma são denominados de alocação contígua ou contínua e neles o armazenamento físico corresponde à organização lógica do arquivo, ou seja, o primeiro bloco de dados ocupa o primeiro setor alocado e assim sucessivamente (vide Figura 5.27).

Este é o esquema mais simples de organização física de arquivos que exhibe algumas vantagens e desvantagens [DEI92, p. 395] [DEI92, p. 163]. As vantagens são:

- o controle de armazenamento do arquivo, isto é, a manutenção de diretórios, se reduz ao tamanho do arquivo e ao setor inicial utilizado e

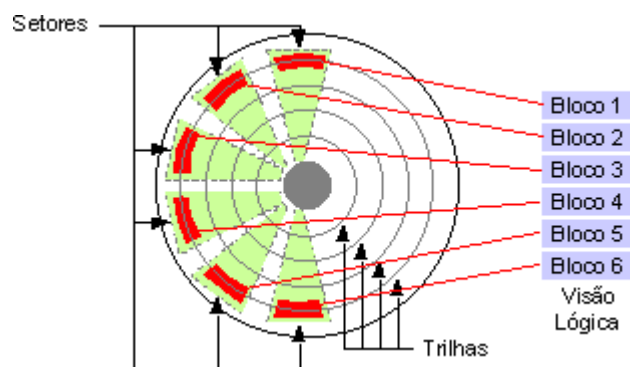


Figura 5.27: Organização física na alocação contígua

- as operações de leitura e escrita são as mais eficientes possíveis para qualquer tipo de dispositivo.

Enquanto que as desvantagens são:

- exige que o tamanho do arquivo seja conhecido no instante de sua criação e
- a ocorrência de fragmentação (veja discussão na seção 5.5.6) reduz a capacidade efetiva de armazenamento, pois novos arquivos só podem ser criados em áreas contíguas.

Sistemas que exigem maior performance nas operações de leitura e escrita e que não necessitem de modificações freqüentes no tamanho de seus arquivos podem utilizar eficientemente este esquema de alocação, tal como o sistema operacional IBM Vm/CMS.

Alocação não contígua

A outra forma possível de organizar-se fisicamente o armazenamento de arquivos é através da alocação não contígua. Neste esquema cada bloco do arquivo pode estar armazenado num setor distinto da unidade de disco, de forma que o armazenamento físico não corresponde à organização lógica do arquivo, como mostra a Figura 5.28.

O principal objetivo da alocação não-contígua é proporcionar um mecanismo mais apropriado para o armazenamento de arquivos que tendem a ter seus tamanhos aumentados ou diminuídos conforme são utilizados.

A alocação não-contígua pode ser implementada de várias formas e dentre estas as estratégias de alocação orientadas à setores:

- Lista ligada de setores
- Lista ligada de setores indexada

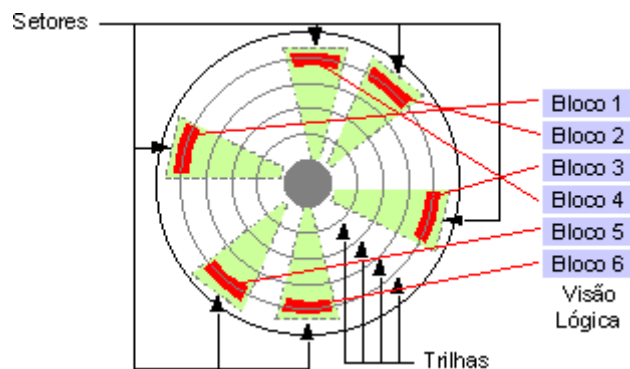


Figura 5.28: Organização física na alocação não contígua

- Indexação de nós (*i-nodes*)

Na **lista ligada de setores** (*linked list allocation*) cada setor do disco contém um ponteiro que pode ser utilizado para indicar um outro setor [SG00, p. 376]. Desta forma um arquivo pode ser armazenado através de uma entrada simples no diretório que indica o primeiro de uma seqüência de setores, onde cada um destes setores aponta para o próximo. Um ponteiro com valor nulo indica que o arquivo terminou (vide Figura 5.29). Não existe necessidade dos setores alocados estarem em posições adjacentes tal como na alocação contígua. O sistema pode manter uma lista de setores livres, que podem ser retirados para a criação e aumento de arquivos ou recuperados quando diminuem ou são eliminados. Isto permite um mecanismo de armazenamento que acomoda facilmente as variações de tamanho dos arquivos, usando integralmente a capacidade do disco, eliminando a necessidade de mecanismos de compactação embora promova a fragmentação da unidade de disco.

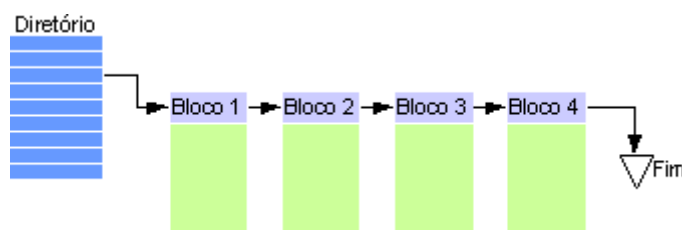


Figura 5.29: Alocação não contígua com lista ligada

Conforme Tanenbaum [TAN92, p. 163] os maiores problemas encontrados no uso do mecanismo de uma lista ligada de setores são:

1. As operações de leitura e escrita tendem a ser ineficientes devido a fragmentação inerente deste método.

2. O acesso randômico deixa de existir pois torna-se necessário ler cada setor alocado para se determinar o próximo até a posição desejada.
3. O posicionamento do ponteiro dentro de cada setor faz com que o bloco de dados deixe de ser uma potência de 2, criando alguns inconvenientes do ponto de vista de programação.

A **lista ligada de setores indexada** utiliza o mesmo princípio de armazenamento de setores interligados eliminando o inconveniente associado ao ponteiro existente em cada setor. É criada uma tabela contendo a relação de todos os setores do dispositivos sendo que para cada entrada se associa um ponteiro (retirado dos setores de dados). A entrada do diretório correspondente ao arquivo aponta para um setor desta tabela (o primeiro setor do arquivo) que tem associado um ponteiro para a próxima entrada e assim sucessivamente. Um ponteiro nulo indica o fim do arquivo na tabela. Isto permite o acesso randômico do arquivo e mantém o bloco de dados do setor num tamanho que é potência de 2, embora a ineficiência devido a fragmentação permaneça. Temos uma representação deste esquema de organização na Figura 5.30.

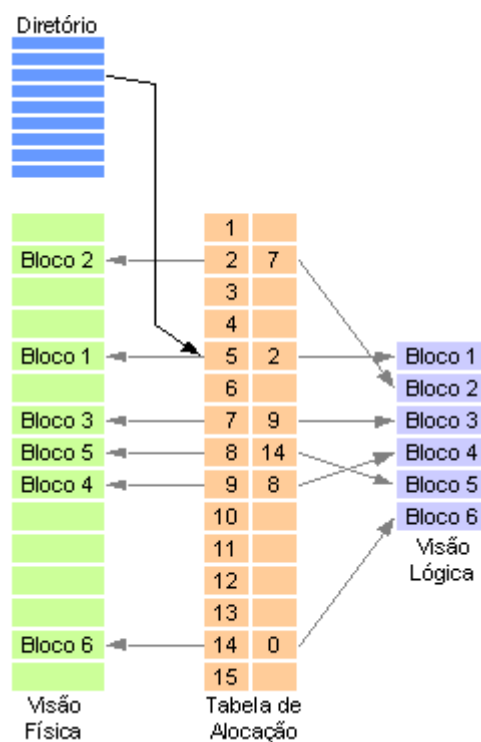


Figura 5.30: Alocação não contígua com lista ligada indexada

A principal desvantagem deste método é o tamanho que da tabela de alocação pode ter em unidades de grande capacidade, por exemplo, uma

unidade de 1.2 Gbytes dividida em setores de 1 Kbytes possui uma tabela de alocação com mais de um milhão de entradas, que pode consumir espaço precioso para ser mantida integralmente em memória, pois cada entrada da tabela tipicamente utiliza 3 ou 4 bytes dependendo de como se dá a otimização do sistema [TAN92, p. 164].

O DOS e Windows utilizam esta estratégia para a organização física de *floppies* e unidades de disco. A tabela de setores é chamada de **Tabela de Alocação de Arquivos** (*file allocation table*) que dá origem ao nome do sistema de arquivos FAT. Na FAT o registro inicial armazena informações sobre a própria unidade, existindo valores especiais para designação de fim de arquivo e setores defeituosos [NOR89, p. 112].

A numeração associada ao nome FAT, como FAT12, FAT16 e FAT32, indica o número de *bits* utilizado para numeração dos setores, representando assim a quantidade máxima de setores que pode ser controlada, ou seja, 12 *bits* permitem endereçar 4.096 setores, 16 *bits* endereçam 64K setores e 32 *bits* possibilitam 4G setores distintos. Isto exhibe outra fraqueza desta estratégia, uma unidade de 1.2 GBytes de capacidade possuirá setores de 512K (307,2K) e 32K (19,2K) com FAT de 12 e 16 *bits* respectivamente, o que é um inconveniente devido a granularidade excessivamente grossa (um arquivo de 1 *byte* ocupa sempre um setor).

Outra forma comum de organização de arquivos é através da indexação de nós (*index nodes* ou *i-nodes*). Um nó indexado é uma pequena estrutura de dados que contém um conjunto de atributos e umas poucas centenas de entradas onde cada entrada é um endereço de um bloco de dados na unidade de disco. Desta forma, um pequeno arquivo pode ser mapeado com um único *i-node*, otimizando todas as operações realizadas sobre ele. Se o arquivo não puder ser armazenado num único *i-node*, alguns dos endereços de blocos de dados são substituídos por endereços de outros *i-nodes* denominados de bloco indireto simples (*single indirect block*). Para arquivos ainda maiores podem ser usados blocos indiretos duplos ou triplos (*double* ou *triple indirect block*). Este é esquema tipicamente utilizado pelos sistemas Unix, como também esquematizado na Figura 5.31.

Uma variação destas estratégias é alocar grupos de setores, denominados blocos (*blocks* ou *extents*) ao invés de setores individuais, no que se denomina estratégias de alocação orientadas à blocos. Estas estratégias visam combinar algumas das vantagens da alocação contígua e da alocação não contígua através da alocação de blocos ao invés de setores individuais, o que elimina parcialmente o problema da fragmentação além de permitir a otimização da leitura ou escrita através de operações com blocos inteiros (*read ahead* ou *lazy write*). Como indica Deitel [DEI92, p. 397], existem várias maneiras de se implementar sistemas de alocação orientados à blocos, semelhantes as existentes para setores:

- Lista ligada de blocos (*block chaining*)

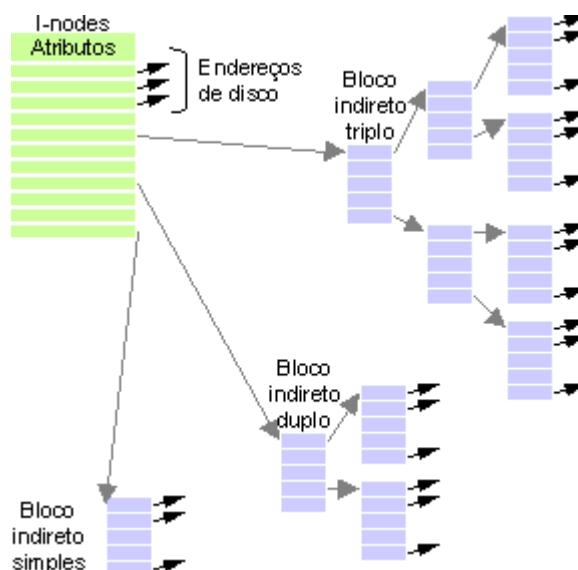


Figura 5.31: Alocação não contígua com I-Nodes

- Lista ligada de blocos indexada (*index block chaining*)
- Mapeamento orientado à blocos (*block oriented file mapping*)

Na **lista ligada de blocos** um número fixo de setores é alocado de cada vez, isto é, os blocos possuem o mesmo tamanho, de modo que cada bloco contenha um ponteiro para o próximo bloco tal como na **lista ligada de setores**, apresentando vantagens e desvantagens idênticas.

Na **lista ligada de blocos indexada**, cujo princípio também é o mesmo da **lista ligada de setores indexada**, é possível termos blocos de tamanho fixo ou variável, num mecanismo mais flexível para o armazenamento de dados.

No **mapeamento orientado à blocos**, os ponteiros são substituídos por um esquema de numeração que pode ser facilmente convertido para a numeração de setores de uma unidade de disco. As operações de modificação de tamanho de arquivos se tornam bastante ágeis neste esquema [DEI92, p. 400].

5.5.6 Fragmentação

Sob certos aspectos, os problemas de que devem ser resolvidos para organização do armazenamento secundário são semelhantes aos encontrados no gerenciamento de memória. A forma de armazenamento mais simples é a disposição dos dados em setores adjacentes das unidades de disco, mas os arquivos são freqüentemente modificados e eliminados e com isto seus tama-

nhos são variáveis. Este fato provoca o que se chama de fragmentação, isto é, começam a surgir setores livres entre setores ocupados.

Se o armazenamento de arquivos é feito através da alocação contígua temos que uma alteração em seu tamanho impõe algum grau de fragmentação, pois se seu tamanho aumenta e a região de armazenamento atual não pode ser expandida, a área anteriormente utilizada provavelmente ficará livre enquanto o arquivo será rearranjado numa outra área. Na situação em que o arquivo tem seu tamanho reduzido, sobrarão alguns setores livres no final de sua área original que só poderão ser ocupados por pequenos arquivos.

Na Figura 5.32 temos uma esquematização da ocorrência da fragmentação em sistemas de arquivos com alocação contígua e também com alocação não contígua.

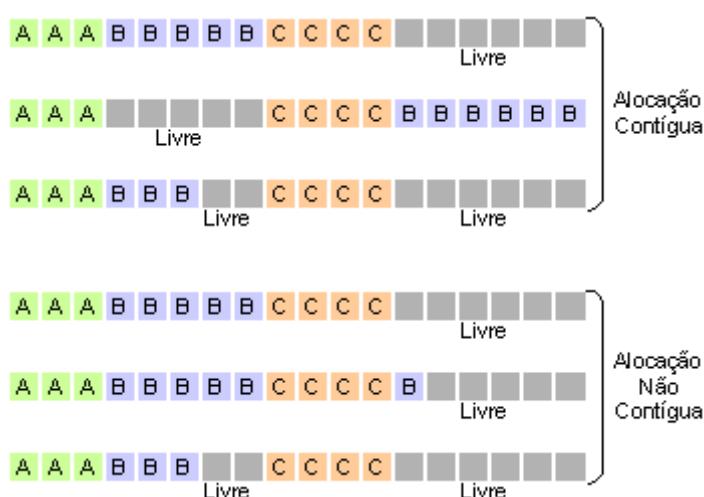


Figura 5.32: Ocorrência de fragmentação com alocação contígua

Na alocação contígua temos que os arquivos são armazenados sempre em setores consecutivos, assim o desempenho de operações de leitura e escrita não é comprometido pela fragmentação, por outro lado a utilização do disco será comprometida pois setores fragmentados do disco poderão permanecer livres não possibilitando o armazenamento de arquivos maiores.

Caso o armazenamento de arquivos seja feito através de alocação não contígua, setores livres passam a ser ocupados de forma descontínua pelos arquivos a medida que seus tamanhos aumentam ou diminuem. Embora o aproveitamento do espaço em disco seja integral, as operações de leitura e escrita terão seu desempenho comprometido tanto mais fragmentado esteja o arquivo sob uso.

Notamos que em ambos os casos ocorre a fragmentação. Satyanarayanan (1981) efetuou um estudo sobre o tamanho dos arquivos e as operações realizadas sobre eles concluindo que:

- a maioria dos arquivos é de pequeno tamanho,
- as operações de leitura são mais frequentes que as operações de escrita,
- a maioria das operações de leitura e escrita são seqüenciais e
- que grande parte dos arquivos tem vida curta.

Isto significa que a fragmentação irá ocorrer qualquer que seja a forma de alocação. Como isto pode não ser admissível em certos sistemas, podem ser implementados mecanismos de compactação ou desfragmentação que reorganizam o armazenamento nos dispositivos de armazenamento de forma a possibilitar a utilização integral do espaço disponível no caso da alocação contígua ou de otimizar as operações de leitura e escrita quando a alocação é não-contígua.

Outros sistemas oferecem utilitários que podem realizar esta operação, permitindo ao usuário ou administrador maior controle sobre o armazenamento, como ilustrado na Figura 5.33.

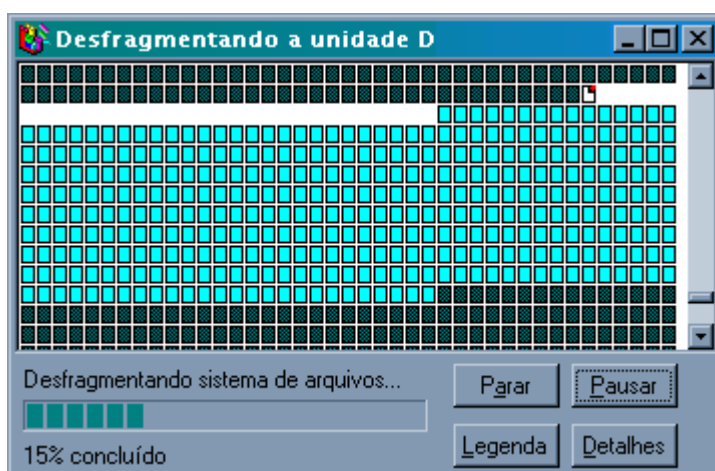


Figura 5.33: Utilitário de desfragmentação de disco do MS-Windows 98

Referências Bibliográficas

- [BLA87] Uyles BLACK. *Computer Networks: Protocols, Standards and Interfaces*. Prentice Hall, Englewood Cliffs, NJ, 1987.
- [BOR92] BORLAND. *Borland C++ 3.1 Programmer's Guide*. Borland International, Scotts Valey, CA, 1992.
- [CAL96] Charles CALVERT. *Delphi 2 Unleashed, 2nd Edition*. Sams Publishing, Indianapolis, IN, 1996.
- [CON99] CONECTIVA. *Manual do Usuário do Conectiva Linux Guarani*. Conectiva, Curitiba, PR, 1999.
- [CRA04] CRAY. *Cray X1 Product Overview*. Cray Research Inc., Internet: <http://www.cray.com/>, recuperado em 01/2004, 2004.
- [DAV91] William S. DAVIS. *Sistemas Operacionais: uma visão sistêmica*. Câmpus, Rio de Janeiro, RJ, 1991.
- [DEI92] Harvey M. DEITEL. *An Introduction to Operating Systems, 2nd Edition*. Addison-Wesley, Reading, MA, 1992.
- [GUI86] Célio Cardoso GUIMARÃES. *Princípios de Sistemas Operacionais, 5ª Edição*. Câmpus, Rio de Janeiro, RJ, 1986.
- [IBM92a] IBM. *IBM DOS 5.02 Technical Reference*. IBM, BocaRaton, FL, 1992.
- [IBM92b] IBM. *A Technical Guide to OS/2 2.0*. IBM, BocaRaton, FL, 1992.
- [JAM87] Kris JAMSA. *DOS: The Complete Reference*. Osborne McGraw-Hill, Berkeley, CA, 1987.
- [LAT96] Y. LANGSAM, M. J. AUGENSTAIN, and A. M. TENENBAUM. *Data Structures Using C and C++, 2nd Edition*. Prentice Hall, Upper Saddle River, NJ, 1996.
- [LET89] Gordon LETWIN. *Explorando o OS/2*. Câmpus, Rio de Janeiro, RJ, 1989.

- [NCA04] NCAR. *NCAR/SCD Supercomputer Gallery*. National Center for Atmospheric Research - Scientific Computer Division, Internet: <http://www.scd.ucar.edu/computers/gallery/>, recuperado em 01/2004, 2004.
- [NOR89] Peter NORTON. *Guia do Programador para IBM PC*. Câmpus, Rio de Janeiro, RJ, 1989.
- [PET96] Charles PETZOLD. *Programming Windows 95*. Microsoft Press, Redmond, WA, 1996.
- [PIT98] David PITTS. *Red Hat Linux Unleashed*. Sams Publishing, Indianapolis, IN, 1998.
- [SG94] Abraham SILBERSCHATZ and Peter Baer GALVIN. *Operating System Concepts, 4th Edition*. Addison-Wesley, Reading, MA, 1994.
- [SG00] Abraham SILBERSCHATZ and Peter Baer GALVIN. *Sistemas Operacionais: Conceitos, 5a Edição*. Prentice Hall, São Paulo, SP, 2000.
- [SGG01] Abraham SILBERSCHATZ, Peter Baer GALVIN, and Greg GAGNE. *Sistemas Operacionais: Conceitos e Aplicações*. Câmpus, Rio de Janeiro, RJ, 2001.
- [SHA96] William A. SHAY. *Sistemas Operacionais*. Makron Books, São Paulo, SP, 1996.
- [STA92] William STALLINGS. *Operating Systems*. Macmillan, New York, NY, 1992.
- [STA96] William STALLINGS. *Computer Organization and Architecture: Designing for Performance, 4th Edition*. Prentice Hall, Upper Saddle River, NJ, 1996.
- [TAN92] Andrew S. TANENBAUM. *Modern Operating Systems*. Prentice Hall, Upper Saddle River, NJ, 1992.
- [TAN95] Andrew S. TANENBAUM. *Distributed Operating Systems*. Prentice Hall, Upper Saddle River, NJ, 1995.