SISTEMAS OPERACIONAIS

APOSTILA por Lucília Ribeiro

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

Versão 6 – ago/08

Índice

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

1 Visão Geral

"As coisas são sempre melhores no começo"
(Blaise Pascal)

1.1 INTRODUÇÃO

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

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

Sem o sistema operacional, um usuário para interagir com o computador deveria conhecer profundamente diversos detalhes sobre hardware do equipamento, o que tornaria seu trabalho lento e com grandes possibilidades de erros.

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

1.2 O SISTEMA OPERACIONAL - FUNCIONALIDADES

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

1.2.1 O SISTEMA OPERACIONAL VISTO COMO UMA MÁQUINA ESTENDIDA

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

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

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

1.2.2 O SISTEMA OPERACIONAL VISTO COMO UM GERENTE DE RECURSOS

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

tem a responsabilidade de permitir o acesso concorrente a esse e a outros recursos de forma organizada e protegida.

Não é apenas em sistemas multiusuário que o sistema operacional é importante. Se pensarmos que um computador pessoal nos permite executar diversas tarefas ao mesmo tempo, como imprimir um documento, copiar um arquivo pela Internet ou processar uma planilha, o sistema operacional deve ser capaz de controlar a execução concorrente de todas essas atividades.

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

1.3 SISTEMA OPERACIONAL NA VISÃO DO USUÁRIO

A arquitetura de um sistema operacional corresponde à imagem que o usuário tem do sistema, a forma como ele percebe o sistema. Essa imagem é definida pela interface através da qual o usuário acessa os serviços do sistema operacional. Essa interface, assim como a imagem, é formada pelas chamadas de sistema e pelos programas de sistema.

1.3.1 CHAMADAS DE SISTEMA

Os programas solicitam serviços ao sistema operacional através das **chamadas de sistema**. Elas são semelhantes às chamadas de sub-rotinas. Entretanto, enquanto as chamadas de sub-rotinas são transferências para procedimentos normais do programa, as chamadas de sistema transferem a execução para o sistema operacional. Através de parâmetros, o programa informa exatamente o que necessita. O retorno da chamada de sistema, assim como o retorno de uma sub-rotina, faz com que a execução do programa seja retomada a partir da instrução que segue a chamada. Para o programador *assembly*, as chamadas de sistema são bastante visíveis. Em uma linguagem de alto nível, elas ficam escondidas dentro da biblioteca utilizada pelo compilador. O programador chama sub-rotinas de uma biblioteca. São as sub-rotinas da biblioteca que chamam o sistema. Por exemplo, qualquer função da biblioteca que acesso o terminal (como printf na linguagem C) exige uma chamada de sistema. Como foi exposto antes, o acesso aos periféricos é feito, normalmente, pelo sistema operacional.

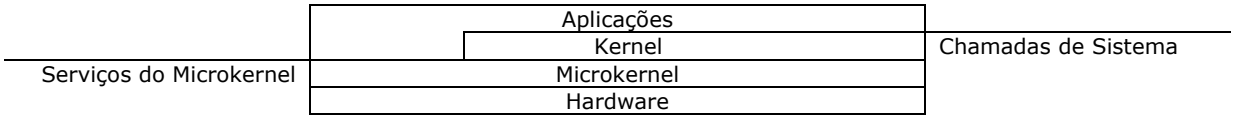
A lista de serviços do sistema operacional é agora transformada em uma lista de chamadas de sistema. A descrição dessas chamadas forma um dos mais importantes manuais de um sistema operacional. Por exemplo, considere um programa que lista o conteúdo de um arquivo texto na tela do terminal. Ele faz uma chamada de sistema para verificar se o arquivo a ser listado existe. Um dos parâmetros dessa chamada será provavelmente o nome do arquivo. O restante do programa é um laço no qual são feitas sucessivas leituras do arquivo e escritas no terminal. Essas duas operações também correspondem a chamadas de sistema.

A parte do sistema operacional responsável por implementar as chamadas de sistema é normalmente chamada de **núcleo** ou **kernel**. Os principais componentes do *kernel* de qualquer sistema operacional são a **gerência do processador**, a **gerência de memória**, o **sistema de arquivos** e a **gerência de entrada e saída**. Cada um desses componentes será visto detalhadamente.

Em função da complexidade interna de um *kernel* completo, muitos sistemas operacionais são implementados em camadas. Primeiro, um pequeno componente de software chamado **micronúcleo** ou **microkernel** implementa os serviços mais básicos associados com sistemas operacionais. Em cima do *microkernel*, usando os seus serviços, o *kernel* propriamente dito implementa os demais serviços. A figura abaixo ilustra um sistema no qual, acima do hardware, existe um *microkernel* que oferece serviços básicos tais como gerência do processador, alocação e liberação de memória física e instalação de novos tratadores de dispositivos. O *kernel* do sistema oferece serviços tais como sistema de arquivos, memória virtual e protocolos de comunicação.

Alguns sistemas permitem que as aplicações acessem tanto as chamadas de sistema suportadas pelo *kernel* quanto os serviços oferecidos pelo *microkernel*. Entretanto, na

maioria das vezes, apenas o código do *kernel* pode acessar os serviços do *microkernel*, enquanto aplicações ficam restritas às chamadas de sistema do *kernel*.



1.4 PROJETO DO SISTEMA OPERACIONAL

Na visão de projeto, o mais importante é como o sistema está organizado internamente. A organização de um sistema operacional corresponde à forma como ele implementa os vários serviços.

O sistema operacional não resolve os problemas do usuário final. Ele não serve para editar textos, nem faz a contabilidade de uma empresa. Entretanto, através dele, podemos obter uma maior eficiência e conveniência no uso do computador. A eficiência é obtida através do compartilhamento dos recursos. A conveniência é obtida através de uma interface mais confortável para a utilização dos recursos computacionais.

Normalmente o processador está executando programas de usuário. Para isso que o computador foi comprado. Somente quando ocorre algum evento especial, o sistema operacional é ativado. Dois tipos de eventos ativam o sistema operacional: uma chamada de sistema ou uma interrupção de periférico.

Uma chamada de sistema corresponde a uma **solicitação de serviço** por parte do programa em execução. Primeiramente, deve ser verificada a legalidade da solicitação. Por exemplo, um pedido para que arquivos de outros usuários sejam destruídos deverá ser recusado. No caso de uma solicitação legal, ela é realizada, e a resposta é devolvida ao programa. É possível que a chamada de sistema envolva o acesso a um periférico. Nesse caso, o programa deverá esperar até que o periférico conclua a operação solicitada.

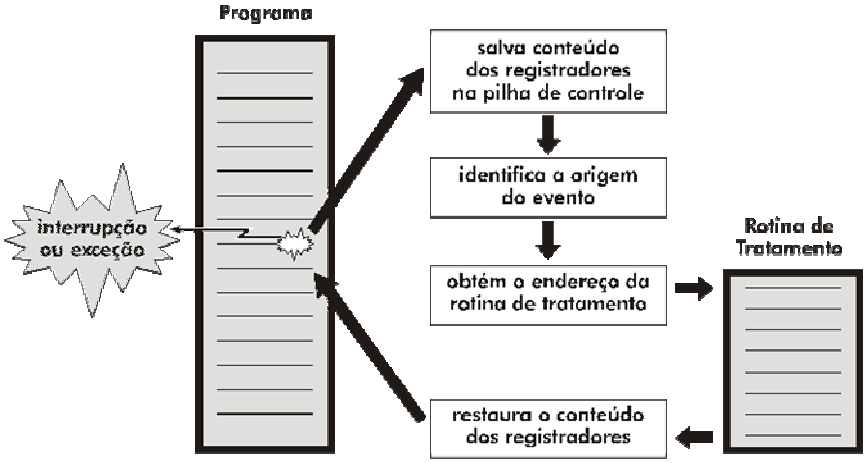
Em função das chamadas de sistema, o sistema operacional envia comandos para os controladores dos periféricos. O controlador deve informar ao sistema operacional quando a operação estiver concluída. Isso é feita através de uma interrupção. Quando a interrupção acontece, o processador pára o que está fazendo e passa a executar uma rotina específica do sistema operacional. Como a interrupção do periférico avisa o término de alguma operação de entrada e saída, possivelmente uma chamada de sistema foi concluída. Nesse caso, um programa à espera de resposta poderá ser liberado.

1.4.1 INTERRUPÇÃO x EXCEÇÃO

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

Ao final da execução de cada instrução, a unidade de controle verifica a ocorrência de algum tipo de interrupção. Neste caso, o programa em execução é interrompido e o controle desviado para uma rotina responsável por tratar o evento ocorrido, denominada **rotina de tratamento de interrupção**. Para que o programa possa posteriormente voltar a ser executado, é necessário que, no momento da interrupção, um conjunto de informações sobre a sua execução seja preservado.

Para cada tipo de interrupção existe uma rotina de tratamento associada, para qual o fluxo de execução deve ser desviado. A identificação do tipo de evento ocorrido é fundamental para determinar o endereço da rotina de tratamento.



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

As interrupções são decorrentes de eventos assíncronos, ou seja, não relacionadas à instrução do programa corrente, portanto são imprevisíveis e podem ocorrer múltiplas vezes de forma simultânea. Uma maneira de evitar esta situação é a rotina de tratamento inibir as demais interrupções.

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

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

A diferença fundamental entre exceção e interrupção é que a primeira é gerada por um evento síncrono, enquanto a segunda é gerada por eventos assíncronos. Um evento é **síncrono** quando é resultado direto da execução do programa corrente. Tais eventos são previsíveis e, por definição, só pode ocorrer um único de cada vez.

1.5 HISTÓRICO DOS SISTEMAS OPERACIONAIS

A evolução dos Sistemas Operacionais está intimamente relacionada com o desenvolvimento dos computadores. Objetivando permitir um conhecimento geral, o histórico dos Sistemas Operacionais aqui apresentados será dividido em décadas. Em cada uma das décadas serão discutidas as principais características do *hardware* e do Sistema Operacional da época.

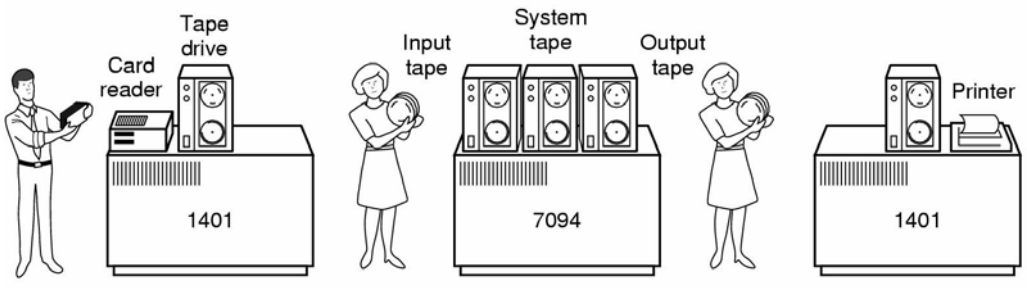
1.5.1 DÉCADA DE 1940 (Ausência de SOs)

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

1.5.2 DÉCADA DE 1950 (Processamento em *Batch*)

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

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



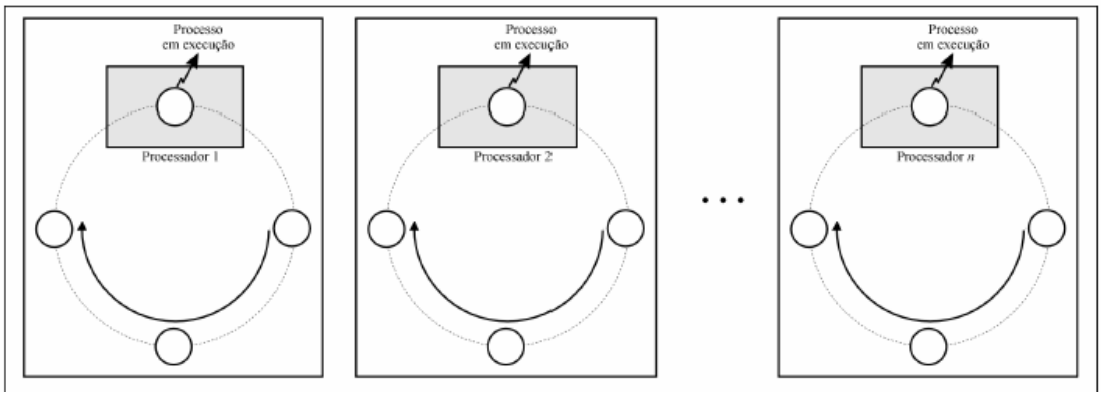
Pode não parecer um avanço, mas anteriormente os programas eram submetidos pelo operador, um a um, fazendo com que o processador ficasse ocioso entre a execução de um job e outro. Com o processamento batch, um conjunto de programas era submetido de uma vez só, o que diminuía o tempo de espera entre a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

1.5.3 DÉCADA DE 1960 (Processamento *Time-sharing*)

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

1.5.4 DÉCADA DE 1970 (Multiprocessamento)

Nesta década a integração em larga escala (Large Scale Integration – LSI) e a integração em muito grande escala (Very Large Scale Integration - VLSI) permitiram uma redução significativa no tamanho e no custo dos computadores. Com isso, os computadores com múltiplos processadores tornaram-se acessíveis e os processos passaram a ser executados em paralelo. Este tipo de processamento ficou conhecido como multiprocessamento, o qual é ilustrado pela figura abaixo:



1.5.5 DÉCADA DE 1980 (Computadores Pessoais):

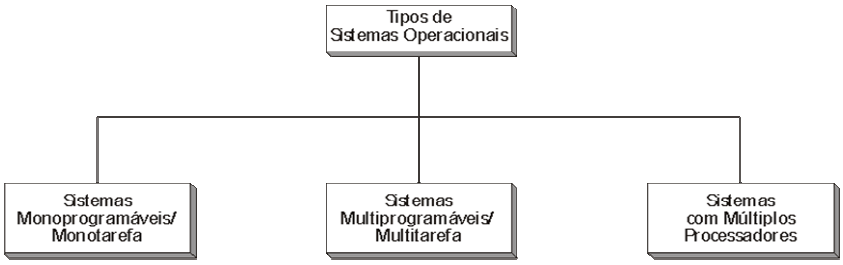
Os computadores pessoais tornam-se mais acessíveis comercialmente e a microcomputação se consolidou. Assim, surgiram os SOs voltados especificamente para os então chamados microcomputadores (ex.: CP/M6, MS-DOS7, OS/28 etc.). É interessante destacar que neste período a rápida evolução dos processadores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

1.5.6 DÉCADA DE 1990 (Sistemas Distribuídos):

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

1.6 CLASSIFICAÇÃO DOS SISTEMAS OPERACIONAIS

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



1.6.1 SISTEMAS MONOPROGRAMÁVEIS OU MONOTAREFAS

Os primeiros Sistemas Operacionais eram tipicamente voltados para a execução de um único programa. Qualquer outra aplicação, para ser executada, deveria aguardar o término do programa corrente. Os sistemas monoprogramáveis se caracterizam por permitir que o processador, a memória e os periféricos permaneçam exclusivamente dedicados à execução de um único programa.

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

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

1.6.2 SISTEMAS MULTIPROGRAMÁVEIS OU MULTITAREFAS

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

Neste caso, enquanto um programa espera por uma operação de leitura ou gravação em disco, outros programas podem estar sendo processados neste mesmo intervalo de tempo. Nesse caso, podemos observar o compartilhamento da memória e do processador. O Sistema Operacional se preocupa em gerenciar o acesso concorrente aos seus diversos recursos.

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

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

Os sistemas multiprogramáveis podem também ser classificados pela forma com que suas aplicações são gerenciadas, podendo ser divididos em sistemas batch, de tempo compartilhado ou de tempo real. Um Sistema Operacional pode suportar um ou mais desses tipos de processamento, dependendo de sua implementação.

a) Sistemas Batch

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

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

b) Sistemas de Tempo Compartilhado

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

Geralmente, sistemas de tempo compartilhado permitem a interação dos usuários com o sistema através de terminais que incluem vídeo, teclado e mouse. Esses sistemas possuem uma linguagem de controle que permite ao usuário comunicar-se diretamente com o Sistema Operacional através de comandos.

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

c) Sistemas de Tempo Real

Os sistemas de tempo real (*real-time*) diferem dos de tempo compartilhado no tempo exigido no processamento das aplicações.

Enquanto em sistemas de tempo compartilhado o tempo de processamento pode variar sem comprometer as aplicações em execução, nos sistemas de tempo real os tempos de processamento devem estar dentro de limites rígidos, que devem ser obedecidos, caso contrário poderão ocorrer problemas irreparáveis.

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

1.6.3 SISTEMAS COM MÚLTIPLOS PROCESSADORES

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

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

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

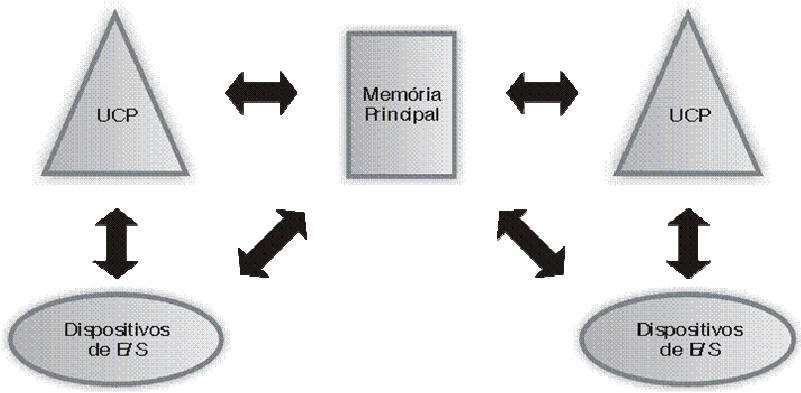
Disponibilidade é a capacidade de manter o sistema em operação mesmo em casos de falhas. Neste caso, se um dos processadores falhar, os demais podem assumir suas funções de maneira transparente aos usuários e suas aplicações, embora com menor capacidade de computação.

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

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

a) Sistemas Fortemente Acoplados

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



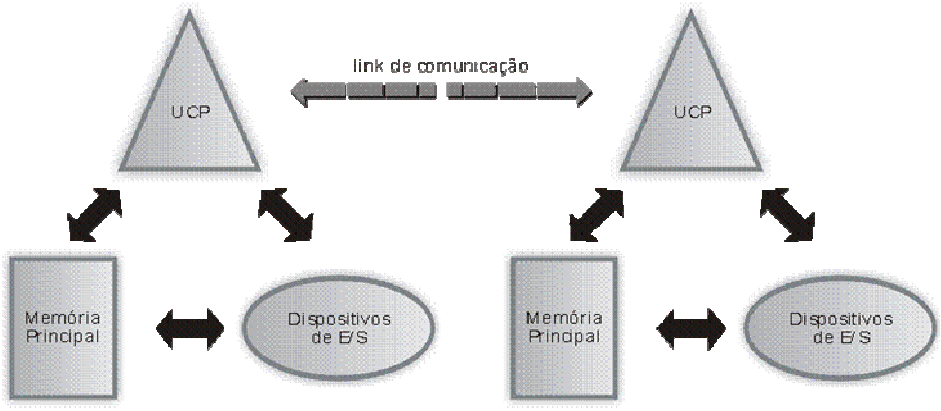
Os sistemas fortemente acoplados podem ser divididos em simétricos ou assimétricos. Os simétricos caracterizam-se pelo tempo uniforme de acesso à memória principal pelos diversos processadores.

Inicialmente, tais sistemas estavam limitados aos sistemas de grande porte, restritos ao ambiente universitário e às grandes corporações. Com a evolução dos computadores pessoais e das estações de trabalho, os sistemas multitarefa evoluíram para permitir a existência de vários processadores no modelo simétrico. Atualmente, a grande maioria dos Sistemas Operacionais, como o Unix e o Windows 2000, implementa esta funcionalidade.

b) Sistemas Fracamente Acoplados

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

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



Com base no grau de integração dos *hosts* da rede, podemos dividir os sistemas fracamente acoplados em Sistemas Operacionais de Rede e Sistemas Distribuídos. A grande diferença entre os dois modelos é a capacidade do Sistema Operacional em criar uma imagem única dos serviços disponibilizados pela rede.

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

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

1.7 EXERCÍCIOS

1) Como seria utilizar um computador sem um Sistema Operacional?

2) O que é um Sistema Operacional? Fale sobre suas principais funções.

3) Quais os tipos de Sistemas Operacionais existentes?

4) Por que dizemos que existe uma subutilização de recursos em sistemas monoprogramáveis?

5) Qual a grande diferença entre sistemas monoprogramáveis e multiprogramáveis?

6) Quais as vantagens dos sistemas multiprogramáveis?

7) Um sistema monousuário pode ser um sistema multiprogramável? Dê um exemplo.

8) Quais são os tipos de sistemas multiprogramáveis?

9) O que caracteriza um sistema *batch*? Quais aplicações podem ser processadas neste tipo de ambiente?

10) Como os processos são executados em um sistema *time-sharing*? Quais as vantagens em utilizá-los?

11) Qual a grande diferença entre sistemas de tempo compartilhado e tempo real? Quais aplicações são indicadas para sistemas de tempo real?

12) O que são sistemas com múltiplos processadores e quais as vantagens em utilizá-los?

13) Qual a grande diferença entre sistemas fortemente e fracamente acoplados?

14) O que é um sistema fracamente acoplado? Qual a diferença entre Sistemas Operacionais de rede e Sistemas Operacionais distribuídos?

15) Cite dois exemplos de Sistemas Operacionais de rede.

-X-

2 Multiprogramação

"Cada ferramenta carrega consigo o espírito com o qual foi criada."
(Werner Karl Heisenberg)

2.1 INTRODUÇÃO

Como visto no capítulo anterior, a multiprogramação torna mais eficiente o aproveitamento dos recursos do computador. Isso é conseguido através da execução simultânea de vários programas. Neste contexto, são exemplos de recursos o tempo do processador, o espaço na memória, o tempo de periférico, entre outros. Este capítulo trata dos principais conceitos associados com a multiprogramação.

2.2 MECANISMO BÁSICO

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

A tabela abaixo apresenta um exemplo de um programa que lê registros de um arquivo e executa, em média, 100 instruções por registro lido. Neste caso, o processador gasta aproximadamente 93% do tempo esperando o dispositivo de E/S concluir a operação para continuar o processamento.

Leitura de um registro	0,0015 s
Execução de 100 instruções	0,0001 s
Total	0,0016 s
% utilização da Cpu (0,0001 / 0,0016) = 0,066 = 6,6%	

Outro aspecto a ser considerado é a subutilização da memória principal. Um programa que não ocupe totalmente a memória ocasiona a existência de áreas livres sem utilização. Nos sistemas multiprogramáveis, vários programas podem estar residentes em memória, concorrendo pela utilização do processador.

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

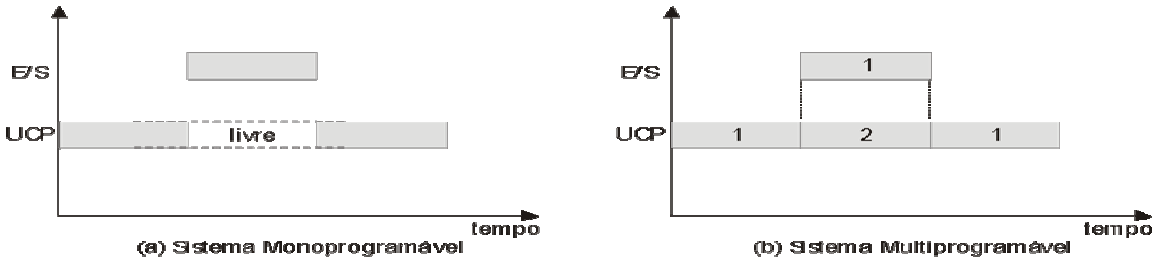
As vantagens proporcionadas pela multiprogramação podem ser percebidas onde existem um sistema computacional por exemplo, com um disco, um terminal e uma impressora, como mostra a tabela abaixo. Neste ambiente são executados três programas, que possuem características de processamento distintas.

Em um ambiente monoprogramável, os programas são executados seqüencialmente. Sendo assim, o Prog1 é processado em 5 minutos, enquanto o Prog2 espera para começar sua execução, que leva 15 minutos. Finalmente, o Prog3 inicia sua execução após 20 minutos e completa seu processamento em 10 minutos, totalizando 30 minutos na execução dos três programas. No caso de os programas serem executados concorrentemente, em um sistema multiprogramável, o ganho na utilização dos recursos e também no tempo de resposta é considerável.

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

Em um **sistema multiprogramado** diversos programas são mantidos na memória ao mesmo tempo. A figura abaixo mostra uma possível organização da memória. Nesse sistema existem 3 programas de usuário na memória principal, prontos para serem executados. Vamos supor que o sistema operacional inicia a execução do programa 1. Após algum tempo, da ordem de milissegundos, o programa 1 faz uma chamada de sistema. Ele solicita algum tipo de operação de entrada ou saída. Por exemplo, uma leitura do disco. Sem a multiprogramação, o processador ficaria parado durante a realização do acesso. Em um sistema multiprogramado, enquanto o periférico executa o comando enviado, o sistema operacional inicia a execução de um outro programa. Por exemplo, o programa 2. Dessa forma, processador e periférico trabalham juntos. Enquanto o processador executa o programa 2, o periférico realiza a operação solicitada pelo programa 1.

Memória Principal	Endereços
Sistema Operacional (256 Kbytes)	00000 H (0) 3FFFF H (262143)
Programa Usuário 1 (160 Kbytes)	40000 H (262144) 67FFF H (425983)
Programa Usuário 2 (64 Kbytes)	68000 H (425984) 77FFF H (491519)
Programa Usuário 3 (32 Kbytes)	78000 H (491520) 7FFFF H (524287)



2.3 O CONCEITO DE PROCESSO

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

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

Nos primeiros sistemas computacionais apenas um programa podia ser executado de cada vez. Tal programa tinha acesso a todos os recursos do sistema. Atualmente, diver-

sos programas executam simultaneamente num sistema computacional. Entretanto, para que isso seja possível é necessário um controle maior na divisão de tarefas entre os vários programas. Daí resultou o conceito de processo.

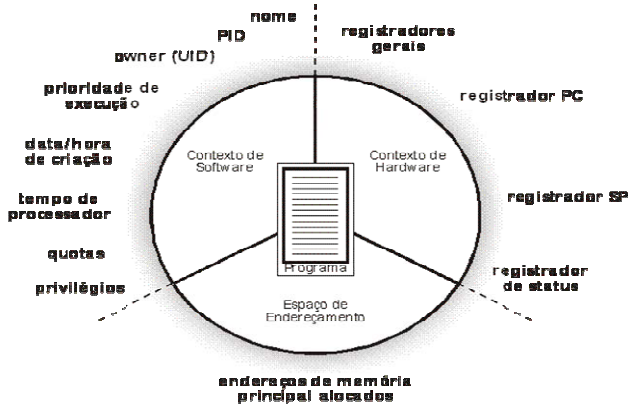
Um processo é um programa em execução, incluindo os valores correntes de todos os registradores do hardware, e das variáveis manipuladas por ele no curso de sua execução. Embora um processador execute vários processos ao mesmo tempo, dando ao usuário a impressão de paralelismo de execução, num dado instante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

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

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

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

2.4 ESTRUTURA DO PROCESSO

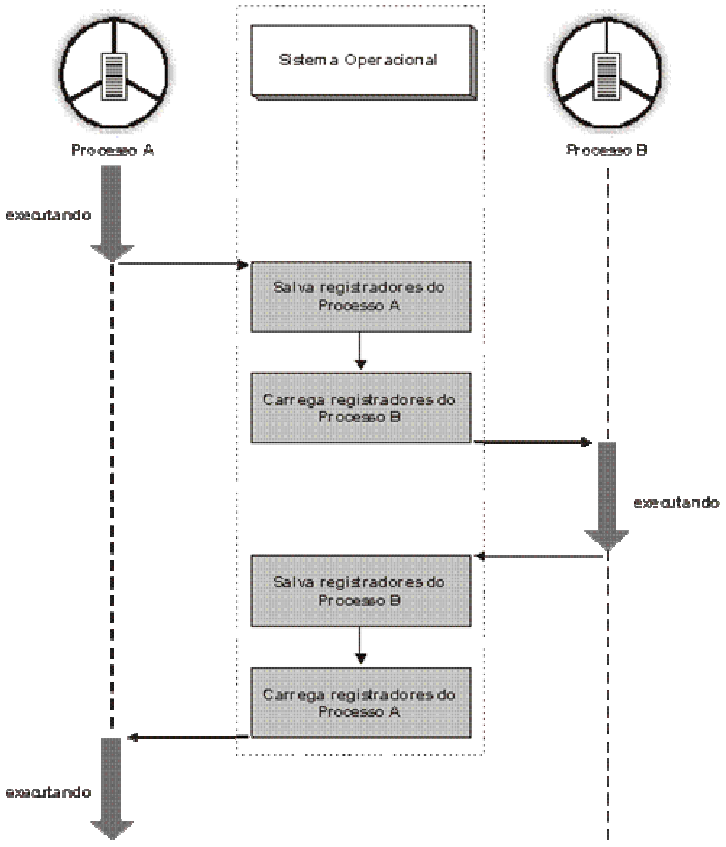


Um processo é formado por três partes, que juntas mantêm todas as informações necessárias à execução de um programa: Contexto de Hardware, Contexto de Software e Espaço de Endereçamento.

2.4.1 CONTEXTO DE HARDWARE

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

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



2.4.2 CONTEXTO DE SOFTWARE

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

A maior parte das informações do contexto de software do processo são provenientes de um arquivo do Sistema Operacional, conhecido como arquivo de contas. Neste arquivo, gerenciado pelo administrador do sistema, são especificados os limites dos recursos que cada processo pode alocar.

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

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

- **Quotas**: são os limites de cada recurso do sistema que um processo pode alocar. Caso uma quota seja insuficiente, o processo será executado lentamente, interrompido durante seu processamento ou mesmo não será executado. Exemplos:
  - Número máximo de arquivos abertos simultaneamente;
  - Tamanho máximo de memória principal e secundária que o processo pode alocar;
  - Número máximo de operações de E/S pendentes;
  - Tamanho máximo do buffer para operações de E/S;
  - Número máximo de processos e threads que podem ser criados.

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

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

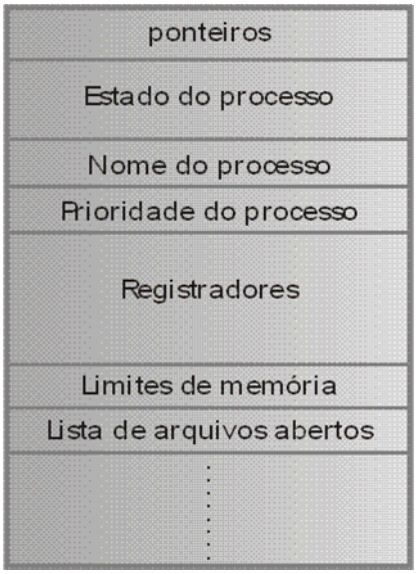
Privilégios que afetam o sistema são os mais amplos e poderosos, pois estão relacionados à operação e gerência do ambiente, como a desativação do sistema, alteração de regras de segurança, criação de outros processos privilegiados, modificação de parâmetros de configuração do sistema, dentro outros. A maioria dos Sistemas Operacionais possui uma conta de acesso com todos estes privilégios disponíveis, com o propósito de o administrador gerenciar o Sistema Operacional. No sistema Unix existe a conta "root", no Windows a conta "administrador" e no OpenVMS existe a conta "system" com este perfil.

2.4.3 ESPAÇO DE ENDEREÇAMENTO

É a área de memória pertencente ao processo onde as instruções e os dados do programa são armazenados para execução. Cada processo possui seu próprio espaço de endereçamento, que deve ser devidamente protegido do acesso dos demais processos.

2.4.4 BLOCO DE CONTROLE DE PROCESSO

O processo é implementado pelo Sistema Operacional através de uma estrutura de dados chamada Bloco de Controle de Processo (Process Control Block – PCB). A partir do PCB, o Sistema Operacional mantém todas as informações sobre o contexto de hardware, contexto de software e espaço de endereçamento de cada processo.



2.5 ESTADOS DO PROCESSO

Em um sistema multiprogramável, um processo não deve alocar a CPU com exclusividade, de formas que possa existir um compartilhamento no uso do processador. Os processos passam por diferentes estados ao longo do seu processamento, em função de

eventos gerados pelo Sistema Operacional ou pelo próprio processo. Um processo ativo pode encontrar-se em três estados diferentes:

- **Rodando** (Execução ou running): nesse estado, o processo está sendo processado pela CPU. Somente um processo pode estar sendo executado em um dado instante. Os processos se alternam na utilização do processador.

- **Pronto** (ready): um processo está no estado de pronto quando aguarda apenas para ser executado. O Sistema Operacional é responsável por determinar a ordem e os critérios pelos quais os processos em estado de pronto devem fazer uso do processador. Em geral existem vários processos no sistema no estado de pronto organizados em listas encadeadas.

- **Bloqueado** (Espera ou wait): nesse estado, um processo aguarda por algum evento externo ou por algum recurso para prosseguir seu processamento. Como exemplo, podemos citar o término de uma operação de E/S ou a espera de uma determinada entrada de dados para continuar sua execução. O sistema organiza os vários processos no estado de espera também em listas encadeadas, separados por tipo de evento. Neste caso, quando um evento acontece, todos os processos da lista associada ao evento são transferidos para o estado de pronto.

2.6 MUDANÇA DE ESTADO DO PROCESSO

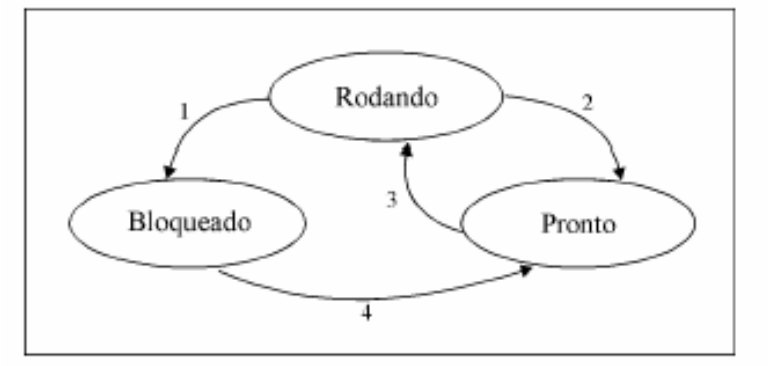
Um processo muda de estado durante seu processamento em função de eventos originados por ele próprio (*eventos voluntários*) ou pelo Sistema Operacional (*eventos involuntários*). Basicamente, existem quatro mudanças de estado que podem ocorrer a um processo:

1) **Rodando – Bloqueado**: essa transição ocorre por eventos gerados pelo próprio processo, como uma operação de E/S.

2) **Rodando – Pronto**: um processo em execução passa para o estado de pronto por eventos gerados pelo sistema, como o término da fatia de tempo que o processo possui para sua execução. Nesse caso, o processo volta para a fila de pronto, onde aguarda por uma nova oportunidade para continuar seu processamento.

3) **Pronto – Rodando**: o Sistema Operacional seleciona um processo da fila de prontos para executar.

4) **Bloqueado** – **Pronto**: um processo passa de bloqueado para pronto quando a operação solicitada é atendida ou o recurso esperado é concedido.



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

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

As transições 2 e 3 são causadas pelo escalonador de processos sem que o processo saiba disso. A transição 2 ocorre quando o escalonador decide que o processo em execu-

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

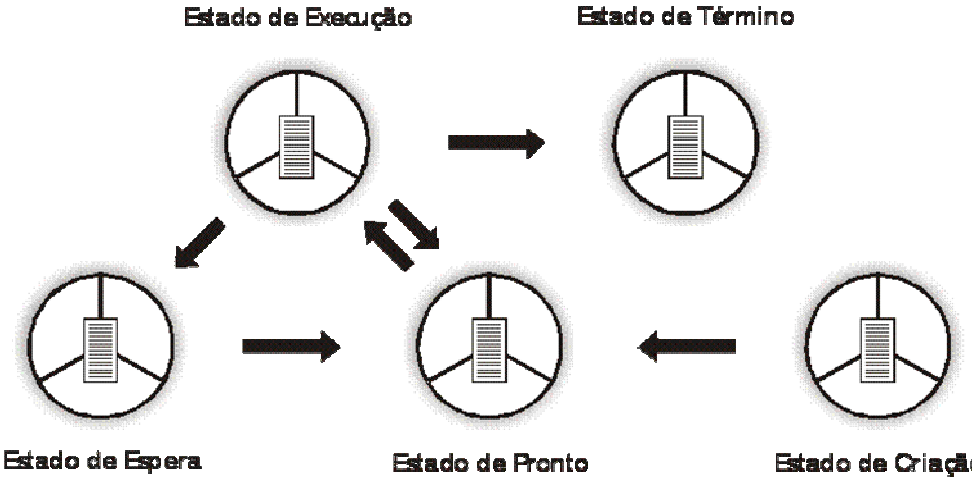
A transição 4 ocorre quando acontece um evento externo pelo qual um processo estava aguardando (como a chegada de alguma entrada).

2.7 CRIAÇÃO E ELIMINAÇÃO DE PROCESSOS

A criação de um processo ocorre a partir do momento em que o Sistema Operacional adiciona um novo PCB à sua estrutura e aloca um espaço de endereçamento na memória para uso. No caso da eliminação de um processo, todos os recursos associados ao processo são desalocados e o PCB eliminado pelo Sistema Operacional.

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

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



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

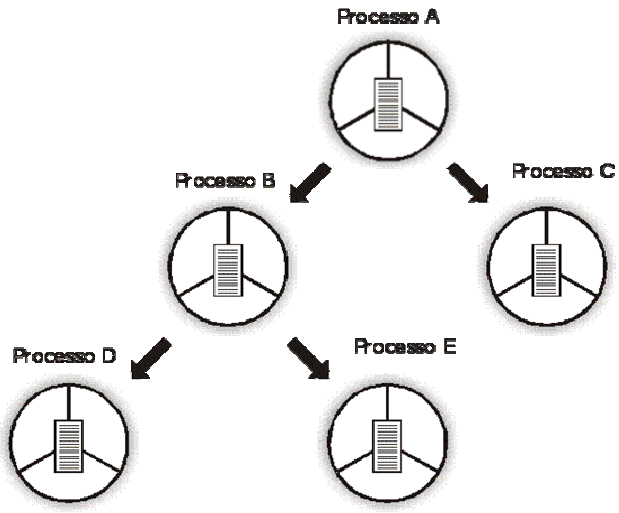
O segundo motivo para término é que o processo descobre um erro. Por exemplo, se um usuário digita um comando para compilar um arquivo que não existe, o compilador simplesmente emite uma chamada de saída ao sistema. Processos interativos com base na tela geralmente não fecham quando parâmetros errados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

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

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

2.8 PROCESSOS INDEPENDENTES, SUBPROCESSOS E THREADS

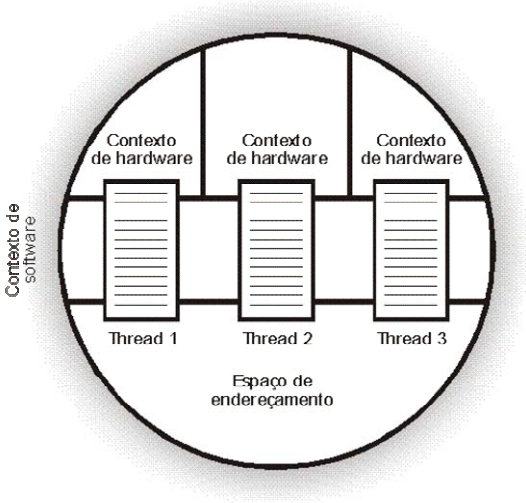
Processos independentes, subprocessos e threads são maneiras diferentes de implementar a concorrência dentro de uma aplicação. Neste caso, busca-se subdividir o código em partes para trabalharem de forma cooperativa. Considere um banco de dados com produtos de uma grande loja onde vendedores fazem freqüentes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



O uso de **processos independentes** é a maneira mais simples de implementar a concorrência. Neste caso não existe vínculo do processo criado com seu criador. A criação de um processo independente exige a alocação de um PCB próprio.

Subprocessos são processos criados dentro de uma estrutura hierárquica. Neste modo, o processo criador é denominado **processo pai** enquanto o novo processo é chamado de **subprocesso** ou **processo filho**. O subprocesso, por sua vez, pode criar outras estruturas de subprocessos. Uma característica desta implementação é a dependência existente entre o processo criador e o subprocesso. Caso um processo pai deixe de existir, os subprocessos subordinados são automaticamente eliminados.

Uma outra característica neste tipo de implementação é que subprocessos podem compartilhar quotas com o processo pai. O uso de processos independentes e subprocessos no desenvolvimento de aplicações concorrentes demanda consumo de diversos recursos do sistema. Sempre que um novo processo é criado, o sistema deve alocar recursos (contexto de hardware, de software e espaço de endereçamento), consumindo tempo de CPU. No momento do término dos processos, o Sistema Operacional também dispensa tempo para desalocar recursos previamente alocados. Outro problema é a comunicação e sincronização entre processos, considerada pouco eficiente, visto que cada processo possui seu próprio espaço de endereçamento.



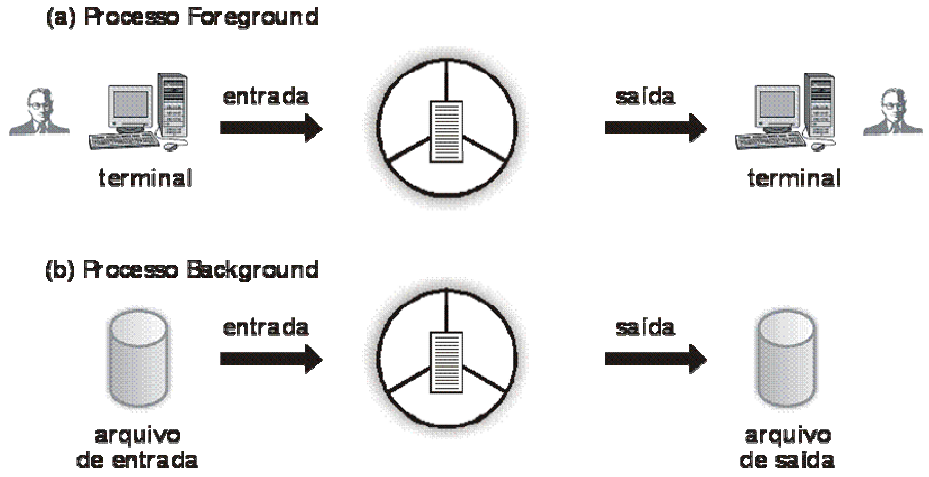
O conceito de thread foi introduzido na tentativa de reduzir o tempo gasto na criação, eliminação e troca de contexto de processos nas aplicações concorrentes, bem como economizar recursos do sistema como um todo. Em um ambiente multithread, um único processo pode suportar múltiplos threads, cada qual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

Cada thread possui seu próprio contexto de hardware, porém compartilha o mesmo contexto de software e espaço de endereçamento com os demais threads do processo. O

compartilhamento do espaço de endereçamento permite que a comunicação de threads dentro de um mesmo processo seja realizada de forma simples e rápida. Este assunto é melhor detalhado no próximo capítulo.

2.9 PROCESSOS PRIMEIRO (*FOREGROUND*) E SEGUNDO PLANO (*BACKGROUND*)

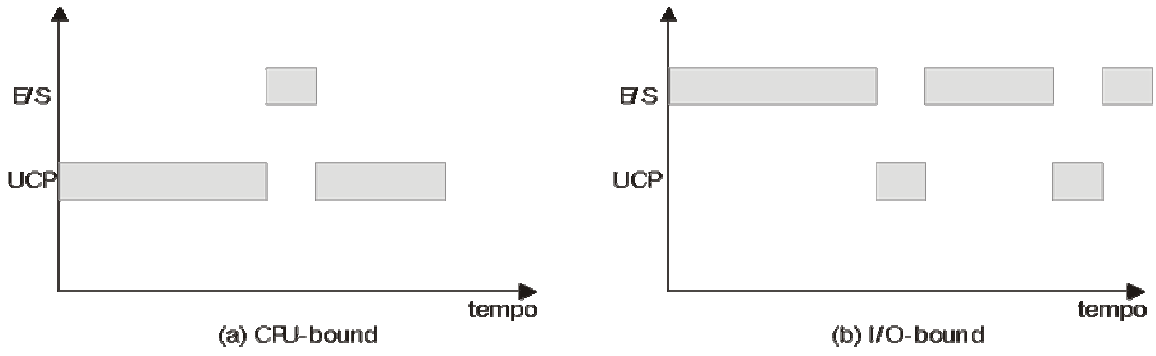
Um **processo de primeiro plano** (interativo) é aquele que permite a comunicação direta do usuário com o processo durante o seu processamento. Um **processo de segundo plano** (*batch*) *é* aquele onde não existe a comunicação com o usuário durante o seu processamento. Neste caso, os canais de E/S não estão associados a nenhum dispositivo de E/S interativo, mas em geral a arquivos de E/S.



2.10. PROCESSOS LIMITADOS POR CPU (CPU-*BOUND*) E POR E/S (*I/O-BOUND*)

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

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



2.11 SELEÇÃO DE PROCESSOS

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

2.11.1 FILAS PARA SELEÇÃO DE PROCESSOS

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

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

2.11.2 ESCALONADORES

Como foi dito anteriormente, um processo passa por várias filas de seleção durante sua execução. Para realizar a alocação de recursos, o sistema operacional deve selecionar os processos dessas filas de alguma maneira. Um escalonador é um programa responsável pelo trabalho de escolher processos e de escalar processos para execução. O escalonador de processos escolhe processos guardados em disco e os carrega na memória para execução. Pode-se dizer que ele seleciona processos que passaram a poder competir pela CPU. Já o escalonador da CPU escolhe dentre os processos prontos aquele que será executado, alocando a CPU a ele.

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

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

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

2.12 EXERCÍCIOS

16) O que é um processo? Diferencie processo de programa.

17) Por que o conceito de processo é tão importante no projeto de sistemas multiprogramáveis?

18) É possível que um programa execute no contexto de um processo e não execute no contexto de um outro? Por quê?

19) Quais partes compõem um processo?

20) O que é contexto de hardware de um processo e como é a implementação da troca de contexto?

21) Qual a função do contexto de software? Exemplifique cada grupo de informação.

22) O que é o espaço de endereçamento de um processo?

23) Como o Sistema Operacional implementa o conceito de processo? Qual a estrutura de dados indicada para organizar os diversos processos na memória principal?

24) Cada processo é descrito por um bloco de controle de processos. Quais são as informações contidas no BCP?

25) Defina os três estados possíveis de um processo.

26) Faça o diagrama de estados de um processo demonstrando e explicando as transições de um estado para outro.

27) Dê um exemplo que apresente todas as mudanças de estado de um processo, juntamente com o evento associado a cada mudança.

28) Na teoria, com três estados poderia haver seis transições, duas para cada estado. Contudo, somente quatro transições são mostradas. Por que?

29) Diferencie processos multithreads, subprocessos e processos independentes.

30) Explique a diferença entre processos de primeiro plano e de segundo plano.

31) Dê exemplos de aplicação limitada por CPU e limitada por E/S.

32) Trace um paralelo entre o Escalonador de Processos e o Escalonador de CPU.

-X-

3 Programação Concorrente

"Não se pode conceber a maioria sem a unidade."
(Platão)

3.1 INTRODUÇÃO

Programação concorrente tem sido usada frequentemente na construção de sistemas operacionais e em aplicações nas áreas de comunicação de dados e controle industrial. Esse tipo de programação torna-se ainda mais importante com o advento dos sistemas distribuídos e das máquinas com arquitetura paralela.

3.2 DEFINIÇÃO

Um programa que é executado por apenas um processo é chamado de **programa seqüencial**. A grande maioria dos programas escritos são assim. Nesse caso, existe somente um fluxo de controle durante a execução. Isso permite, por exemplo, que o programador realize uma "execução imaginária" de seu programa apontando com o dedo, a cada instante, a linha do programa que está sendo executada no momento.

Um **programa concorrente** é executado simultaneamente por diversos processos que cooperam entre si, isto é, trocam informações. Para o programador realizar agora uma "execução imaginária", ele vai necessitar de vários dedos, um para cada processo que faz parte do programa. Nesse contexto, trocar informações significa trocar dados ou realizar algum tipo de sincronização. É necessária a existência de interação entre os processos para que o programa seja considerado concorrente. Embora a interação entre processos possa ocorrer através de acesso a arquivos comuns, esse tipo de concorrência é tratada na disciplina de Banco de Dados. A concorrência tratada aqui, utiliza mecanismos rápidos para interação entre processos: variáveis compartilhadas e trocas de mensagens.

O verbo "concorrer" admite em português vários sentidos. Pode ser usado no sentido de cooperar, como em "tudo coopera para o bom êxito da operação". Também pode ser usado com o significado de disputa ou competição, como em "ele concorreu a uma vaga na universidade". Em uma forma menos comum ele significa também existir simultaneamente. De certa forma, todos os sentidos são aplicáveis aqui na programação concorrente. Em geral, processos concorrem (disputam) pelos mesmos recursos do hardware e do sistema operacional. Por exemplo, processador, memória, periféricos, estruturas de dados, etc. Ao mesmo tempo, pela própria definição de programa concorrente, eles concorrem (cooperam) para o êxito do programa como um todo. Certamente, vários processos concorrem (existem simultaneamente) em um programa concorrente. Logo, programação concorrente é um bom nome para esse assunto.

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

3.3 SINCRONIZAÇÃO

O problema apresentado a seguir não é um dos vários problemas clássicos apresentados na disciplina sistemas operacionais. É utilizado somente como um exemplo figurado para fixar os conceitos desse tópico.

Considere que exista uma casa onde moram duas pessoas que raramente se encontram. Quando uma chega a outra sai e vice-versa. A melhor forma de comunicação é através de bilhetes.

Nessa casa, consome-se sempre um litro de leite por dia. Não pode ocorrer a falta do leite nem tão pouco o excesso do mesmo.

Tentaremos resolver o "Problema do Leite na Geladeira" utilizando algoritmos com funções genéricas.

O Problema do Leite na Geladeira

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

Primeira tentativa de resolver o Problema do Leite na Geladeira:

Processos A e B

```
if (SemLeite) {
    if (SemAviso) {
        Deixa Aviso;
        Compra Leite;
        Remove Aviso;
    }
}
```

E se A for atropelada quando for comprar o leite?

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

Processo A

```
if (SemAviso) {
    if (SemLeite) {
        Compra Leite;
    }
    Deixa Aviso;
}
```

Processo B

```
if (Aviso) {
    if (SemLeite) {
        Compra Leite;
    }
    Remove Aviso;
}
```

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

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

Terceira tentativa de resolver o Problema do Leite na Geladeira: utilizar dois avisos diferentes.

Processo A	Processo B

Deixa AvisoA;
if (SemAvisoB) {
  if (SemLeite) {
  Compra Leite;
  }
}
Remove AvisoA;
	Deixa AvisoB;
	if (SemAvisoA) {
	  if (SemLeite) {
	  Compra Leite;
	  }
	}
	Remove AvisoB;

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

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

Processo A	Processo B

Deixa AvisoA;
if (SemAvisoB) {
  if (SemLeite) {
  Compra Leite;
  }
} else {
  while (AvisoB) {
  EsperaAToa;
  /* pode sentar */
  }
  if (SemLeite) {
  CompraLeite;
  }
}
Remove AvisoA;
	Deixa AvisoB;
	if (SemAvisoA) {
	  if (SemLeite) {
	  Compra Leite;
	  }
	}
	Remove AvisoB;

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

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

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

3.4 ESPECIFICAÇÃO DE CONCORRÊNCIA EM PROGRAMAS

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

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

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

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

Uma das implementações mais claras e simples de expressar concorrência em um programa é a utilização dos comandos PARBEGIN e PAREND (Dijkstra, 1965). O comando PARBEGIN especifica que a seqüência de comandos seja executada concorrentemente em uma ordem imprevisível, através da criação de um processo (P1, P2, Pn) para cada comando (C1, C2, Cn). O comando PAREND define um ponto de sincronização, onde o processamento só continuará quando todos os processos ou threads criados já tiverem terminado suas execuções.

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

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

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

3.5 PROBLEMAS DE COMPARTILHAMENTO DE RECURSOS

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

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

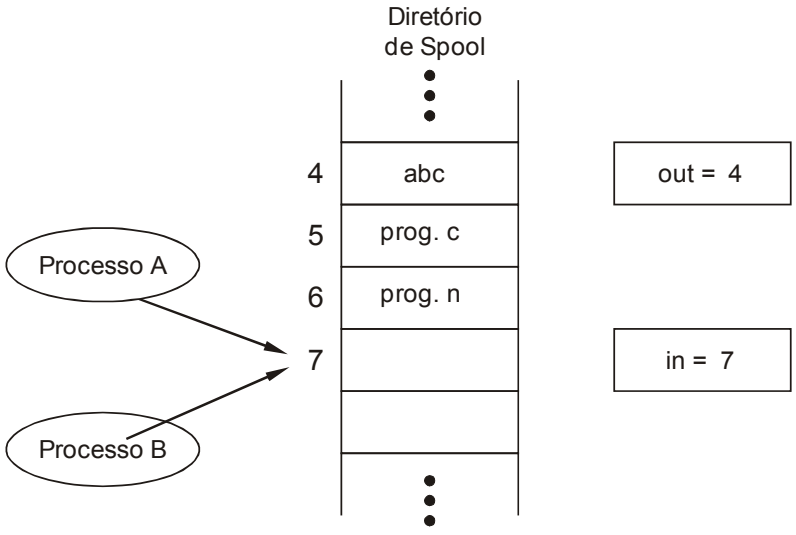
Considerando processos concorrentes pertencentes a dois funcionários do banco que atualizam o saldo de um mesmo cliente simultaneamente, a situação de compartilhamento do recurso pode ser analisada. O processo do primeiro funcionário (Caixa 1) lê o registro do cliente e soma ao campo Saldo o valor do lançamento de débito. Antes de gravar o novo saldo no arquivo, o processo do segundo funcionário (Caixa 2) lê o registro do mesmo cliente, que está sendo atualizado, para realizar outro lançamento, desta vez de crédito. Independentemente de qual dos processos atualize primeiro o saldo no arquivo, o dado gravado estará inconsistente.

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

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

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

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



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

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

Analisando os dois exemplos apresentados, é possível concluir que em qualquer situação, onde dois ou mais processos compartilham um mesmo recurso, devem existir mecanismos de controle para evitar esses tipos de problemas.

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

3.6 EXCLUSÃO MÚTUA

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

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

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

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

1. Nenhum processo que esteja fora de sua região crítica pode bloquear a execução de outro processo

2. Nenhum processo pode esperar indefinidamente para entrar em sua região crítica (*starvation* ou espera indefinida)

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

3.7 SOLUÇÕES DE HARDWARE

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

3.7.1 DESABILITAÇÃO DAS INTERRUPÇÕES

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

Em sistemas com múltiplos processadores, essa solução torna-se ineficiente devido ao tempo de propagação quando um processador sinaliza aos demais que as interrupções devem ser habilitadas ou desabilitadas.

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

3.7.2 INSTRUÇÃO TEST-AND-SET LOCKED (TSL)

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

A instrução TSL possui o formato a seguir, e quando executada o valor lógico da variável Y é copiado para X, sendo atribuído à variável Y o valor lógico verdadeiro:

```
TSL (X,Y);
```

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

```
PROGRAM Test_And_Set;
   VAR Bloqueio : BOOLEAN;
BEGIN
   Bloqueio := False;
   PARBEGIN
      Processo_A;
      Processo_B;
   PAREND;
END.
```

```
PROCEDURE Processo_A;
  VAR Pode_A : BOOLEAN;
BEGIN
    Pode_A := True;
    WHILE (Pode_A) DO
      TSL(Pode_A, Bloqueio);
    Regiao_Critica_A;
    Bloqueio := False;
END;
```

```
PROCEDURE Processo_B;
  VAR Pode_B : BOOLEAN;
BEGIN
    Pode_B := True;
    WHILE (Pode_B) DO
       TSL(Pode_B, Bloqueio);
    Regiao_Critica_B;
    Bloqueio := False;
END;
```

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

3.8 SOLUÇÕES DE SOFTWARE

Diversos algoritmos foram propostos na tentativa de implementar a exclusão mútua através de soluções de software. As primeiras soluções tratavam apenas da exclusão mútua para dois processos e, inicialmente, apresentavam alguns problemas. Veremos os algoritmos de forma evolutiva até alcançar uma solução definitiva para a exclusão mútua entre N processos.

3.8.1 PRIMEIRO ALGORITMO: ESTRITA ALTERNÂNCIA

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

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

```
PROCEDURE Processo_A;
BEGIN
    WHILE (Vez = 'B') DO;
    Regiao_Critica_A;
    Vez := 'B';
    Processamento_A;
END;
```

```
PROCEDURE Processo_B;
BEGIN
    WHILE (Vez = 'A') DO;
    Regiao_Critica_B;
    Vez := 'A';
    Processamento_B;
END;
```

Este algoritmo apresenta duas limitações: 1) a primeira surge do próprio mecanismo de controle. O acesso ao recurso compartilhado só pode ser realizado por dois processos e sempre de maneira alternada. Caso o Processo A permaneça muito tempo na rotina Processamento_A, é possível que o Processo B queira executar sua região crítica e não consiga, mesmo que o Processo A não esteja mais utilizando o recurso. Como o Processo B pode executar seu loop mais rapidamente que o Processo A, a possibilidade de executar sua região crítica fica limitada pela velocidade de processamento do Processo A. 2) no caso da ocorrência de algum problema com um dos processos, de forma que a variável de bloqueio não seja alterada, o outro processo permanecerá indefinidamente bloqueado.

3.8.2 SEGUNDO ALGORITMO:

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

```
PROGRAM Algoritmo2;
   VAR CA, CB : BOOLEAN;
BEGIN
   CA := false;
   CB := false;
   PARBEGIN
      Processo_A;
      Processo_B;
   PAREND;
END.
```

```
PROCEDURE Processo_A;
BEGIN
    WHILE (CB) DO;
    CA := true;
    Regiao_Critica_A;
    CA := false;
    Processamento_A;
END;
```

```
PROCEDURE Processo_B;
BEGIN
    WHILE (CA) DO;
    CB := true;
    Regiao_Critica_B;
    CB := false;
    Processamento_B;
END;
```

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

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

3.8.3 TERCEIRO ALGORITMO:

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

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

```
PROCEDURE Processo_A;
BEGIN
    CA := true;
    WHILE (CB) DO;
    CA := true;
    Regiao_Critica_A;
    CA := false;
    Processamento_A;
END;
```

```
PROCEDURE Processo_B;
BEGIN
    CB := true;
    WHILE (CA) DO;
    CB := true;
    Regiao_Critica_B;
    CB := false;
    Processamento_B;
END;
```

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

3.8.4 QUARTO ALGORITMO:

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

```
PROGRAM Algoritmo4;
   VAR CA, CB : BOOLEAN;
BEGIN
   CA := false;
   CB := false;
   PARBEGIN
      Processo_A;
      Processo_B;
   PAREND;
END.
```

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

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

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

3.8.5 ALGORITMO DE PETERSON

A primeira solução de software que garantiu a exclusão mútua entre dois processos sem a incorrência de outros problemas foi proposta pelo matemático holandês T. Dekker com base no primeiro e quarto algoritmos. Esse algoritmo possui uma lógica bastante complexa. Posteriormente, G. L. Peterson propôs outra solução mais simples.

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

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

```
PROGRAM Algoritmo_Peterson;
   VAR CA, CB : BOOLEAN;
       Vez : CHAR;
BEGIN
   CA := false;
   CB := false;
   PARBEGIN
      Processo_A;
      Processo_B;
   PAREND;
END.
```

```
PROCEDURE Processo_A;
BEGIN
    CA := true;
    Vez := 'B';
    WHILE (CB and Vez = 'B') DO;
    Regiao_Critica_A;
    CA := false;
    Processamento_A;
END;
```

```
PROCEDURE Processo_B;
BEGIN
    CB := true;
    Vez := 'A';
    WHILE (CA and Vez = 'A') DO;
    Regiao_Critica_B;
    CB := false;
    Processamento_B;
END;
```

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

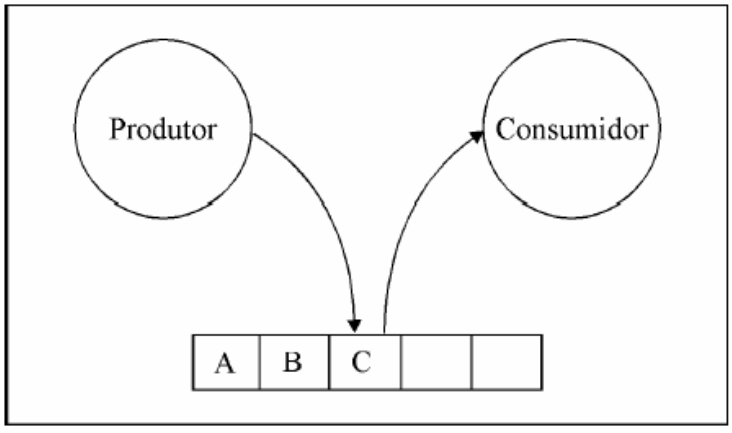
a) **Perda de tempo de processamento**: todas as soluções utilizam-se de um laço de espera ocupada para impedir que dois processos entrem ao mesmo tempo em suas regiões críticas e tais laços consomem, desnecessariamente, o tempo do processador;

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

3.9 PROBLEMA DO PRODUTOR-CONSUMIDOR ou SINCRONIZAÇÃO CONDICIONAL

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

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



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

O programa a seguir exemplifica o problema. O recurso compartilhado é um buffer, definido no algoritmo com o tamanho TamBuf e sendo controlado pela variável Cont. Sempre que a variável Cont for igual a 0, significa que o buffer está vazio e o processo

Consumidor deve permanecer aguardando até que se grave um dado. Da mesma forma, quando a variável Cont for igual a TamBuf, significa que o buffer está cheio e o processo Produtor deve aguardar a leitura de um novo dado. Nessa solução, a tarefa de colocar e retirar os dados do buffer é realizada, respectivamente, pelos procedimentos Grava_Buffer e Le_Buffer, executados de forma mutuamente exclusiva. Não está sendo considerada, neste algoritmo, a implementação da exclusão mútua na variável compartilhada Cont.

```
PROGRAM Produtor_Consumidor_1;
    CONST TamBuf    = (* Tamanho qualquer *);
    TYPE Tipo_Dado  = (* Tipo qualquer *);
    VAR Buffer : ARRAY [1..TamBuf] OF Tipo_Dado;
        Dado   : Tipo_Dado;
        Cont   : 0..TamBuf;
BEGIN
    Cont := 0;
    PARBEGIN
         Produtor;
         Consumidor;
    PAREND;
END.
```

```
PROCEDURE Produtor;
BEGIN
    Produz_Dado (Dado);
    WHILE (Cont = TamBuf) DO;
    Grava_Buffer (Dado, Cont);
    Cont++;
END;
```

```
PROCEDURE Consumidor;
BEGIN
    WHILE (Cont = 0) DO;
    Le_Buffer (Dado);
    Consome_Dado (Dado, Cont);
    Cont--;
END;
```

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

3.10 SEMÁFOROS

Um semáforo é uma variável inteira, não-negativa, que só pode ser manipulada por duas instruções atômicas: DOWN e UP. A instrução UP incrementa uma unidade ao valor do semáforo, enquanto DOWN decrementa a variável. Como, por definição, valores negativos não podem ser atribuídos a um semáforo, a instrução DOWN executada em um semáforo com valor 0, faz com que o processo entre no estado de espera. Em geral, essas instruções são implementadas no processador, que deve garantir todas essas condições.

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

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

3.10.1 EXCLUSÃO MÚTUA UTILIZANDO SEMÁFOROS

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

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

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

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

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

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

```
PROCEDURE DOWN (Var S : Semaforo);
BEGIN
  IF (S = 0) THEN Coloca_Proc_Fila
  ELSE
    S := S - 1;
END;
```

```
PROCEDURE UP (Var S : Semaforo);
BEGIN
  S := S + 1;
 IF(Tem_Proc_Espera)THEN(Tira_Proc);
END;
```

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

```
PROGRAM Semaforo_1;
   VAR S : Semaforo := 1;
BEGIN
   PARBEGIN
      Processo_A;
      Processo_B;
   PAREND;
END.
```

```
PROCEDURE Processo_A;
BEGIN
    DOWN (S);
    Regiao_Critica_A;
    UP (S);
    Processamento_A;
END;
```

```
PROCEDURE Processo_B;
BEGIN
    DOWN (S);
    Regiao_Critica_B;
    UP (S);
    Processamento_B;
END;
```

O Processo A executa a instrução DOWN, fazendo com que o semáforo seja decrementado de 1 e passe a ter o valor 0. Em seguida, o Processo A ganha o acesso à sua região crítica. O Processo B também executa a instrução DOWN, mas como seu valor é igual a 0, ficará aguardando até que o Processo A execute a instrução UP, ou seja, volte o valor semáforo para 1.

3.10.2 SINCRONIZAÇÃO CONDICIONAL UTILIZANDO SEMÁFOROS

O problema do produtor/consumidor é um exemplo de como a exclusão mútua e a sincronização condicional podem ser implementadas com o uso de semáforos. O programa ProdutorConsumidor2 apresenta uma solução para esse problema, utilizando um buf-

fer de apenas duas posições. O programa utiliza três semáforos, sendo um do tipo binário, utilizado para implementar a exclusão mútua e dois semáforos contadores para a sincronização condicional. O semáforo binário Mutex permite a execução das regiões críticas Grava_Buffer e Le_Buffer de forma mutuamente exclusiva. Os semáforos contadores Vazio e Cheio representam, respectivamente, se há posições livres no buffer para serem gravadas e posições ocupadas a serem lidas. Quando o semáforo Vazio for igual a 0, significa que o buffer está cheio e o processo produtor deverá aguardar até que o consumidor leia algum dado. Da mesma forma, quando o semáforo Cheio for igual a 0, significa que o buffer está vazio e o consumidor deverá aguardar até que o produtor grave algum dado. Em ambas as situações, o semáforo sinaliza a impossibilidade de acesso ao recurso.

```
PROGRAM Produtor_Consumidor_2;
    CONST TamBuf    = 2;
    VAR  Vazio : Semaforo := TamBuf;
         Cheio : Semaforo := 0;
         Mutex : Semaforo := 1;
BEGIN
    PARBEGIN
         Produtor;
         Consumidor;
    PAREND;
END.
```

```
PROCEDURE Produtor;
BEGIN
    Produz_Dado;
    DOWN (Vazio);
    DOWN (Mutex);
    Grava_Buffer;
    ÚP (Mutex);
    UP (Cheio);
END;
```

```
PROCEDURE Consumidor;
BEGIN
    DOWN (Cheio);
    DOWN (Mutex);
    Le_Buffer;
    ÚP (Mutex);
    UP (Vazio);
    Consome_Dado;
END;
```

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

3.11 TROCA DE MENSAGENS

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

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

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

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

```
PROCEDURE Produtor;
```

```
PROCEDURE Consumidor;
```

```
VAR Msg : Tipo_Msg;
BEGIN
    Produz_Mensagem (Msg);
    SEND (Msg);
END;
```

```
VAR Msg : Tipo_Msg;
BEGIN
    RECEIVE (Msg);
    Consome_Mensagem (Msg);
END;
```

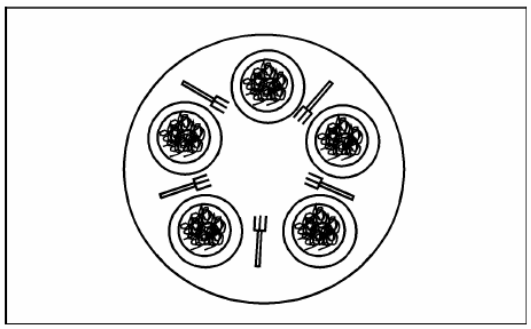
3.12 PROBLEMAS CLÁSSICOS DE SINCRONIZAÇÃO

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

3.12.1 PROBLEMA DOS FILÓSOFOS GLUTÕES

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

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



3.12.2 PROBLEMA DO BARBEIRO DORMINHOCO

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

3.13 EXERCÍCIOS

33) Defina o que é uma aplicação concorrente e dê um exemplo de sua utilização.

34) Relacione Região Crítica com Condição de Corrida.

35) Considere uma aplicação que utilize uma matriz na memória principal para a comunicação entre vários processos concorrentes. Que tipo de problema pode ocorrer quando dois ou mais processos acessam uma mesma posição da matriz?

36) O que é exclusão mútua e como é implementada?

37) Como seria possível resolver os problemas decorrentes do compartilhamento da matriz, apresentado anteriormente, utilizando o conceito de exclusão mútua?

38) O que é *starvation* e como podemos solucionar esse problema?

39) Qual o problema com a solução que desabilita as interrupções para implementar a exclusão mútua?

40) O que é espera ocupada?

41) Quais são os problemas comuns a todas as soluções que se utilizam da espera ocupada para garantir a exclusão mútua de execução?

42) Explique o que é sincronização condicional e dê um exemplo de sua utilização.

43) Explique o que são semáforos e dê dois exemplos de sua utilização: um para a solução da exclusão mútua e outro para sincronização condicional.

44) Em uma aplicação concorrente que controla saldo bancário em contas-correntes, dois processos compartilham uma região de memória onde estão armazenados os saldos dos clientes A e B. Os processos executam concorrentemente os seguintes passos:

Processo 1 (Cliente A)	**Processo 2 (Cliente B)**
`/* saque em A */`	`/* saque em A */`
`1a. x := saldo_cliente_A;`	`2a. y := saldo_cliente_A;`
`1b. x := x - 200;`	`2b. y := y - 100;`
`1c. saldo_cliente_A := x;`	`2c. saldo_cliente_A := y;`
`/* deposito em B */`	`/* deposito em B */`
`1d. x := saldo_cliente_B;`	`2d. y := saldo_cliente_B;`
`1e. x := x + 100;`	`2e. y := y + 200;`
`1f. saldo_cliente_B := x;`	`2f. saldo_cliente_B := y;`

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

a) Quais os valores corretos esperados para os saldos dos clientes A e B após o término da execução dos processos?
b) Quais os valores finais dos saldos dos clientes se a seqüência temporal de execução das operações for: 1a, 2a, 1b, 2b, 1c, 2c, 1d, 2d, 1e, 2e, 1f, 2f?
c) Utilizando semáforos, proponha uma solução que garanta a integridade dos saldos e permita o maior compartilhamento possível dos recursos entre os processos, não esquecendo a especificação da inicialização dos semáforos.

45) O problema dos leitores/escritores apresentado a seguir, consiste em sincronizar processos que consultam/atualizam dados em uma base comum. Pode haver mais de um leitor lendo ao mesmo tempo; no entanto, enquanto um escritor está atua-

lizando a base, nenhum outro processo pode ter acesso a ela (nem mesmo leitores).

```
VAR Acesso : Semaforo := 1;
    Exclusao : Semaforo := 1;
    Nleitores : INTEGER := 0;
```

```
PROCEDURE Escritor;
BEGIN
  ProduzDado;
  DOWN (Acesso);
  Escreve;
  UP (Acesso);
END;
```

```
PROCEDURE Leitor;
BEGIN
  DOWN (Exclusao);
  Nleitores := Nleitores + 1;
  IF (Nleitores = 1) THEN DOWN (Acesso);
  UP (Exclusao);
  Leitura;
  DOWN (Exclusao);
  Nleitores := Nleitores – 1;
  IF (Nleitores = 0) THEN UP(Acesso);
  UP (Exclusao);
  ProcessaDado;
END;
```

a) Suponha que exista apenas um leitor fazendo acesso à base. Enquanto este processo realiza a leitura, quais os valores das três variáveis?
b) Chega um escritor enquanto o leitor ainda está lendo. Quais os valores das três variáveis após o bloqueio do escritor? Sobre qual(is) semáforo(s) se dá o bloqueio?
c) Chega mais um leitor enquanto o primeiro ainda não acabou de ler e o escritor está bloqueado. Descreva os valores das três variáveis enquanto o segundo leitor inicia a leitura.
d) Os dois leitores terminam simultaneamente a leitura. É possível haver problemas quanto à integridade do valor da variável Nleitores? Justifique.
e) Descreva o que acontece com o escritor quando os dois leitores terminam suas leituras. Descreva os valores das três variáveis quando o escritor inicia a escrita.
f) Enquanto o escritor está atualizando a base, chegam mais um escritor e mais um leitor. Sobre qual(is) semáforo(s) eles ficam bloqueados? Descreva os valores das três variáveis após o bloqueio dos recém-chegados.
g) Quando o escritor houver terminado a atualização, é possível prever qual dos processos bloqueados (leitor ou escritor) terá acesso primeiro à base?
h) Descreva uma situação onde os escritores sofram (adiamento indefinido) *starvation*.

-X-

4 Deadlock

"Para todo problema há uma solução que é simples, elegante e errada."
(H. L. Mencken)

4.1 INTRODUÇÃO

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

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

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

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

4.2 EXEMPLOS DE DEADLOCKS

4.2.1 DEADLOCK DE TRÁFEGO

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

4.2.2 DEADLOCK SIMPLES DE RECURSOS

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

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

4.2.3 DEADLOCK EM SISTEMAS DE SPOOLING

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

Em alguns sistemas de *spool*, todo o *job* de impressão deve ser gerado antes do início da impressão. Isto pode gerar uma situação de *deadlock*, uma vez que o espaço disponível em disco para a área de *spooling* é limitado. Se vários processos começarem a gerar seus dados para o *spool*, é possível que o espaço disponível para o *spool* fique cheio an-

tes mesmo de um dos *jobs* de impressão tiver terminado de ser gerado. Neste caso, todos os processos ficarão esperando pela liberação do espaço em disco, o que jamais vai acontecer. A solução nesse caso seria o operador do sistema cancelar um dos *jobs* parcialmente gerados.

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

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

4.2.4 ADIAMENTO INDEFINIDO

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

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

4.3 RECURSOS

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

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

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

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

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

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

Um sistema possui um número finito de recursos para serem distribuídos entre processos concorrentes. Os recursos são classificados segundo vários tipos, sendo que cada tipo pode consistir de uma quantidade de instâncias idênticas. Por exemplo, se considerarmos o tipo de recurso CPU, em uma máquina com dois processadores, temos duas instâncias do recurso CPU.

Se um processo requisita uma instância de um tipo de recurso, a alocação de qualquer instância daquele tipo irá satisfazer a requisição. Se em um determinado sistema esta satisfação não ocorrer, isto significa que as instâncias não são idênticas, e que as classes de tipos de recursos não estão definidas corretamente. Por exemplo, suponha que um sistema possui duas impressoras. Elas poderiam ser definidas como instâncias de um mesmo tipo de recurso. Entretanto se uma estivesse instalada no quarto andar, e

outra no térreo do prédio, os usuários do andar térreo podem não enxergar as duas impressoras como sendo equivalentes. Neste caso, classes de recursos separadas precisariam ser definidas para cada impressora.

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

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

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

4.4 QUATRO CONDIÇÕES NECESSÁRIAS PARA DEADLOCKS

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

1) **Condição de exclusão mútua**: cada recurso ou está alocado a exatamente um processo ou está disponível

2) **Condição de posse e de espera**: processos que estejam de posse de recursos obtidos anteriormente podem solicitar novos recursos

3) **Condição de não-preempção**: recursos já alocados a processos não podem ser tomados à força. Eles precisam ser liberados explicitamente pelo processo que detém sua posse.

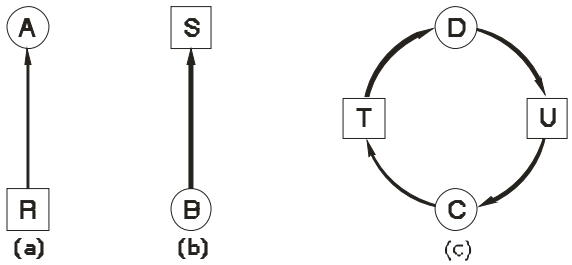
4) **Condição de espera circular**: deve existir uma cadeia circular de dois ou mais processos, cada um dos quais esperando por um recurso que está com o próximo membro da cadeia

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

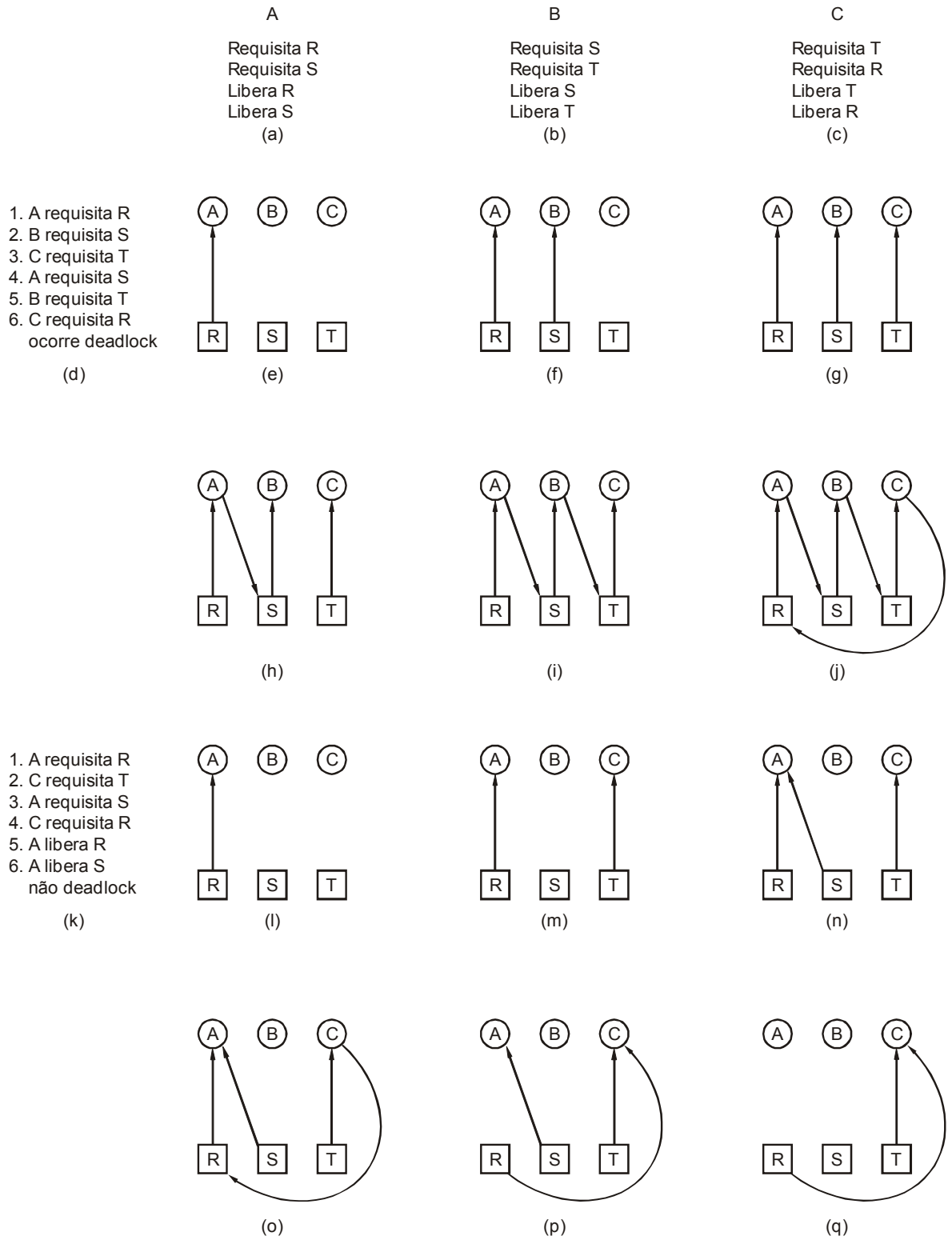
4.5 O MODELO DO DEADLOCK

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

A figura abaixo exibe um grafo de alocação de recursos. (a) Um processo de posse de um recurso. (b) Um processo solicitando um recurso. (c) Deadlock



A seguir, um exemplo de como o deadlock pode ocorrer e de como ele pode ser evitado.



4.6 MÉTODOS PARA LIDAR COM DEADLOCKS

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

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

Detectar e recuperar uma situação de *deadlock*. Se não são usadas estratégias de prevenção ou para evitar *deadlocks*, existe a possibilidade de ocorrência destes. Neste

ambiente, o sistema operacional pode possuir um algoritmo que consiga determinar se ocorreu um *deadlock*, além de um algoritmo que faça a recuperação da situação.

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

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

4.7 O ALGORITMO DO AVESTRUZ

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

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

4.8 DETECÇÃO DE DEADLOCKS

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

4.8.1 DETECÇÃO DO *DEADLOCK* COM UM RECURSO DE CADA TIPO

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

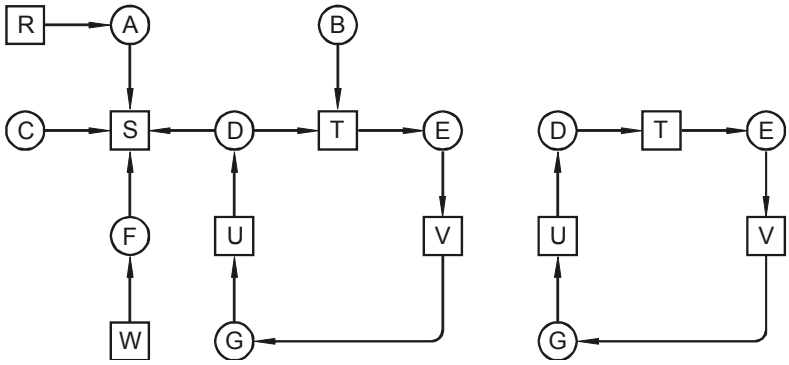
Para esse método, construímos um grafo de recursos conforme já foi visto. Se tal grafo contiver um ou mais ciclos, estará garantida pelo menos uma situação de *deadlock*. Se não houver ciclos, o sistema não estará em *deadlock*.

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

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

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

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



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

4.8.2 DETECÇÃO DO *DEADLOCK* COM VÁRIOS RECURSOS DE CADA TIPO

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

Vamos denominar **E** de vetor de recursos existentes. Ele fornece o número total de instâncias de cada recurso existente. Por exemplo, se a classe 1 refere-se à unidade de fita, E1 = 2, significa que há duas unidades de fitas no sistema

Seja **D** o vetor de recursos disponíveis, com Di fornecendo o número de instâncias do recurso i atualmente disponíveis. Se ambas as unidades de fita estiverem alocadas, D1 = 0.

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

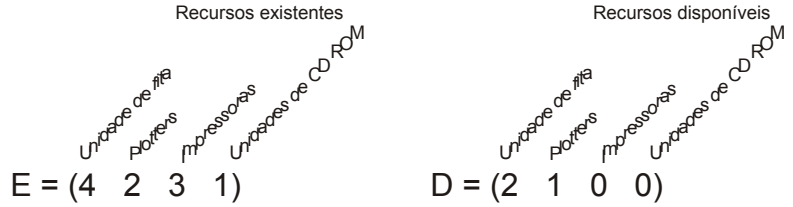
Como exemplo, vamos considerar a figura abaixo, onde existem três processos e quatro classes de recursos, as quais chamamos arbitrariamente de unidades de fita, *plotters*, impressoras e unidades de CD-ROM.

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

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

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

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



Matriz de alocação corrente

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Matriz de requisições

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

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

4.9 RECUPERAÇÃO DE DEADLOCKS

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

4.9.1 RECUPERAÇÃO ATRAVÉS DA PREEMPÇÃO

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

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

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

4.9.2 RECUPERAÇÃO ATRAVÉS DE VOLTA AO PASSADO

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

4.9.3 RECUPERAÇÃO ATRAVÉS DE ELIMINAÇÃO DE PROCESSOS

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

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

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

4.10 TENTATIVAS DE EVITAR O DEADLOCK

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

4.10.1 ESTADOS SEGUROS E INSEGUROS

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

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

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

	Possui	Máximo
A	3	9
B	2	4
C	2	7

Livre: 3

(a)

	Possui	Máximo
A	3	9
B	4	4
C	2	7

Livre: 1

(b)

	Possui	Máximo
A	3	9
B	0	-
C	2	7

Livre: 5

(c)

	Possui	Máximo
A	3	9
B	0	-
C	7	7

Livre: 0

(d)

	Possui	Máximo
A	3	9
B	0	-
C	0	-

Livre: 7

(e)

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

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

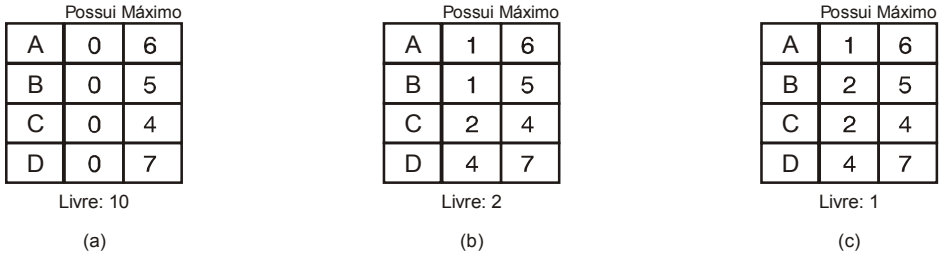
Um estado inseguro não leva necessariamente a um *deadlock*. A diferença entre um estado seguro e um inseguro é que, de um estado seguro, o sistema pode garantir que todos os processos vão terminar, enquanto que a partir de um inseguro esta garantia não pode ser dada.

4.10.2 ALGORITMO DO BANQUEIRO PARA UM ÚNICO TIPO DE RECURSO

Dijkstra (1965) idealizou um algoritmo de escalonamento que pode evitar a ocorrência de *deadlocks*, conhecido como algoritmo do banqueiro. Ele baseia-se nas mesmas premissas adotadas por um banqueiro de um pequeno banco para garantir ou não crédito a seus correntistas.

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

Na figura a seguir, temos três cenários de alocação de recursos: (a) Seguro. (b) Seguro. (c) Inseguro.



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

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

4.11 PREVENÇÃO DE DEADLOCKS

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

4.11.1 ATACANDO O PROBLEMA DA EXCLUSÃO MÚTUA

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

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

4.11.2 ATACANDO O PROBLEMA DA POSSE E DA ESPERA

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

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

4.11.3 ATACANDO O PROBLEMA DA CONDIÇÃO DE NÃO-PREEMPÇÃO

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

4.11.4 ATACANDO O PROBLEMA DA ESPERA CIRCULAR

Resta-nos somente uma condição para tentar resolver a questão dos *deadlocks*. A condição de espera circular pode ser eliminada de diversas maneiras. Uma delas é simplesmente seguindo a regra de que um processo só está autorizado a usar apenas um recurso por vez. Se ele precisar de um segundo recurso, deve liberar o primeiro.

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

Resumindo:

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

4.12 EXERCÍCIOS

46) Quando é que um processo está em *deadlock*?

47) Descreva o *deadlock* em sistemas de *spooling*.

48) Dentro do contexto de *deadlock*, o que é a espera circular?

49) Um sistema operacional pode ser visto de forma mais ampla como um gerenciador de recursos. Ele é responsável pela alocação de vários recursos de diversos tipos. O que seriam recursos nesse contexto?

50) Diferencie recursos preemptíveis e não-preemptíveis.

51) Em sistemas onde processos ficam esperando pela alocação de recursos ou pelas decisões de escalonamento, é possível que ocorra adiamento indefinido (ou *starvation*) e conseqüentemente um *deadlock*. O que vem a ser isso?

52) Existem quatro condições necessárias que devem estar presentes para que possa ocorrer deadlock. Se uma delas estiver ausente, não há a menor possibilidade de ocorrer uma situação de impasse. Quais são elas? Cite e explique cada uma.

53) De acordo com Havender, 1968, se pudermos garantir que pelo menos uma das quatro condições necessárias para que um deadlock ocorra (exclusão mútua, posse e espera, não-preempção e espera circular) nunca sejam satisfeitas, poderemos garantir que será estruturalmente impossível a ocorrência dos mesmos. Fale como isso poderia ser feito considerando o contexto descrito.

54) Existem três formas de recuperar o sistema depois da detecção de um deadlock: recuperação através da preempção, através da volta ao passado e através da eliminação de processos. Fale sobre cada uma delas.

55) Considerando a seguinte matriz de alocação de recursos, onde p é o número de recursos que determinado processo possui, m é o número máximo de recursos que esse processo pode requerer e L o número de recursos Livres no sistema computacional diga se o estado é seguro ou não e demonstre isso através das matrizes. Considere que o sistema computacional tenha 10 recursos disponíveis:

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

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

E = (5 3 4 2) D = ( )

$$C = \begin{pmatrix} 2 & 1 & 0 & 1 \\ 1 & 0 & 2 & 0 \\ 0 & 2 & 1 & 0 \end{pmatrix} \qquad R = \begin{pmatrix} 3 & 0 & 2 & 1 \\ 1 & 0 & 0 & 1 \\ 4 & 1 & 1 & 2 \end{pmatrix}$$

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

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

-X-

5 Gerência do Processador

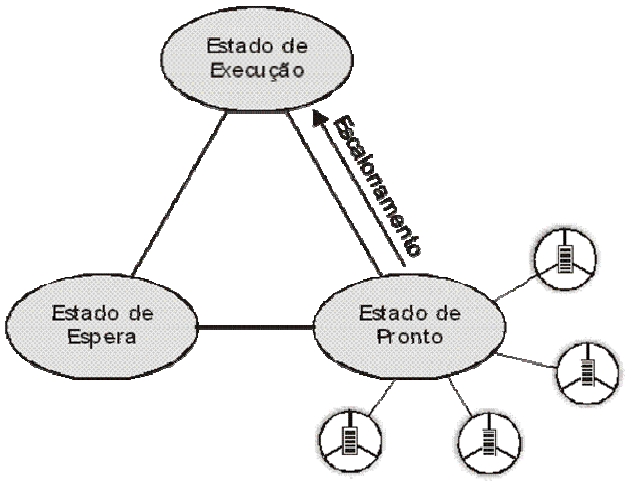
"Mesmo os céus, os planetas e este centro observam grau, prioridade e lugar"
(William Shakespeare)

5.1 INTRODUÇÃO

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

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

5.2 FUNÇÕES BÁSICAS



A política de escalonamento de um Sistema Operacional possui diversas funções básicas, como a de manter o processador ocupado a maior parte do tempo, balancear o uso da CPU entre processos, privilegiar a execução de aplicações críticas, maximizar o *throughput* do sistema e oferecer tempos de resposta razoáveis para usuários interativos. Cada Sistema Operacional possui sua política de escalonamento adequada ao seu propósito e às suas características. Sistemas de tempo compartilhado, por exemplo, possuem requisitos de escalonamento distintos dos sistemas de tempo real.

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

Outra rotina importante na gerência do processador é conhecida como **despachante** (*dispatcher*), responsável pela troca de contexto dos processos após o escalonador determinar qual processo deve fazer uso do processador. O período de tempo gasto na substituição de um processo em execução por outro é denominado **latência do despachante**.

5.3 CRITÉRIOS DE ESCALONAMENTO

As características de cada Sistema Operacional determinam quais são os principais aspectos para a implementação de uma política de escalonamento adequada. Por exemplo, sistemas de tempo compartilhado exigem que o escalonamento trate todos os processos de forma igual, evitando assim, a ocorrência de *starvation*. Já em sistemas de tempo real, o escalonamento deve priorizar a execução de processos críticos em detrimento da execução de outros processos.

a) **Utilização do processador**: o processador deve permanecer ocupado o máximo de tempo possível. Deve ser frisado que quando o sistema computacional trabalha com uma taxa de utilização do processador por volta de 30%, ele é considerado um sistema

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

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

c) **Tempo de processador ou Tempo de CPU**: é o tempo que o processo leva no estado de execução durante seu processamento. As políticas de escalonamento não influenciam o tempo de processador de um processo, sendo este tempo função apenas do código da aplicação e da entrada de dados.

c) **Tempo de espera**: é o tempo total que um processo permanece na fila de pronto durante seu processamento, aguardando para ser executado. A redução do tempo de espera dos processos é desejada pela maioria das políticas de escalonamento.

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

e) **Tempo de resposta**: é o tempo decorrido entre uma requisição ao sistema ou à aplicação e o instante em que a resposta é exibida. Em sistemas interativos, podemos entender como o tempo decorrido entre a última tecla digitada pelo usuário e o início da exibição do resultado no monitor. Em geral, o tempo de resposta não é limitado pela capacidade de processamento do sistema computacional, mas pela velocidade dos dispositivos de E/S.

De uma maneira geral, qualquer política de escalonamento busca otimizar a utilização do processador e a vazão, enquanto tenta diminuir os tempos de retorno, espera e resposta. Apesar disso, as funções que uma política de escalonamento deve possuir são muitas vezes conflitantes. Dependendo do tipo de Sistema Operacional, um critério pode ter maior importância do que outros, como nos sistemas interativos onde o tempo de resposta tem grande relevância.

5.4 ESTRATÉGIAS DE ESCALONAMENTO

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

As estratégias de escalonamento são duas:

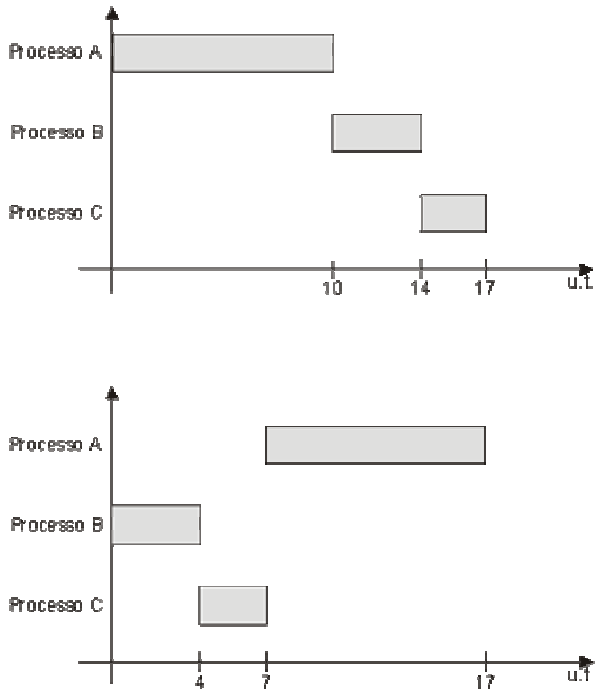
a) **Escalonamento não-preemptivo**: com esta estratégia um processo que entra no processador roda até terminar, sem jamais ser interrompido. Este foi o primeiro tipo de escalonamento desenvolvido e foi utilizado nos SOs de processamento em batch;

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

5.5 ESCALONAMENTO FIRST COME FIRST SERVED (FCFS) ou FIFO

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

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



Processo	Tempo de processador (u.t.)
A	10
B	4
C	3

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

5.6 ESCALONAMENTO MENOR JOB PRIMEIRO ou SJF (Shortest Job First)

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

Na sua concepção inicial, o escalonamento SJF é não preemptivo. Sua vantagem sobre o escalonamento FIFO está na redução do tempo médio de retorno dos processos, porém no SJF é possível haver *starvation* para processos com tempo de processador muito longo ou do tipo CPU-*bound*.

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

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

5.7 ESCALONAMENTO PRÓXIMA TAXA DE RESPOSTA MAIS ALTA ou HRRN

Esse algoritmo corrige algumas das deficiências do SJF, particularmente o adiamento indefinido contra processos mais longos e o favoritismo excessivo em relação a processos curtos. O HRRN (*Highest Response Ratio Next*) usa uma disciplina de escalonamento não preemptiva na qual a prioridade de cada processo é uma função não apenas do seu tempo de execução, mas também do seu tempo de espera. HRRN calcula prioridades dinâmicas segundo a fórmula apresentada posteriormente.

Como o Tempo de CPU aparece no denominador, processos mais curtos recebem preferência. Entretanto, porque o Tempo de Espera aparece no numerador, processos mais

longos que estão à espera também receberão tratamento favorável. Quanto maior a razão, maior a prioridade.

$$\text{Prioridade} = \frac{\text{Tempo de Espera} + \text{Tempo de CPU}}{\text{Tempo de CPU}}$$

5.8 ESCALONAMENTO MENOR TEMPO REMANESCENTE ou SRT

Uma implementação do escalonamento SJF com preempção é conhecida **escalonamento de menor tempo remanescente** ou **SRT** (*Shortest Remaining Time*). Nessa política, toda vez que um processo no estado de pronto tem um tempo de processador estimado menor do que o processo em execução, o Sistema Operacional realiza uma preempção, substituindo-o pelo novo processo.

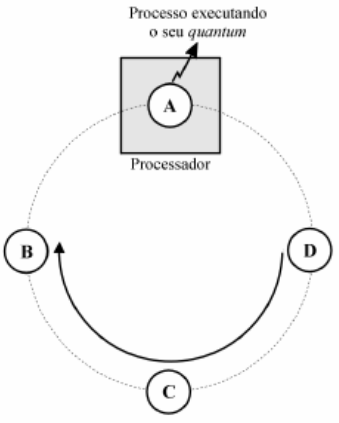
O algoritmo deve manter informações sobre o tempo que está sendo gasto no serviço do processo em execução e realizar preempções ocasionais. Processos que acabaram de chegar cujos tempos de execução são curtos executam quase imediatamente. Todavia, o tempo médio de espera e a variância dos tempos de espera dos processos mais longos são ainda maiores do que no SJF.

5.9 ESCALONAMENTO CIRCULAR ou ROUND ROBIN (RR)

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

No escalonamento circular, toda vez que um processo é escalonado para execução, uma nova fatia de tempo é concedida. Caso a fatia de tempo expire, o Sistema Operacional interrompe o processo em execução, salva seu contexto e direciona-o para o final da fila de pronto. Este mecanismo é conhecido como **preempção por tempo**.

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



Processo	Tempo Estimado de Execução
A	3min.
B	5min.
C	2min.
D	1min.

Simulação do *Round-Robin* com *quantum* de 2min:

Processo	Tempo Executado	Tempo Acumulado	
A	2min.	2min.	
B	2min.	2min.	
C	2min.	2min.	← C terminou!
D	1min.	1min.	← D terminou!
A	1min.	3min.	← A terminou!
B	2min.	4min.	
B	1min.	5min.	← B terminou!

Processo	Tempo Final de Execução
A	2 + 2 + 2 + 1 + 1 = 8min
B	2 + 2 + 2 + 1 + 1 + 2 + 1 = 11min
C	2 + 2 + 2 = 6min.
D	2 + 2 + 2 + 1 = 7min.
Tempo Médio de Execução	**(8 + 11 + 6 + 7)/4 = 8min.**

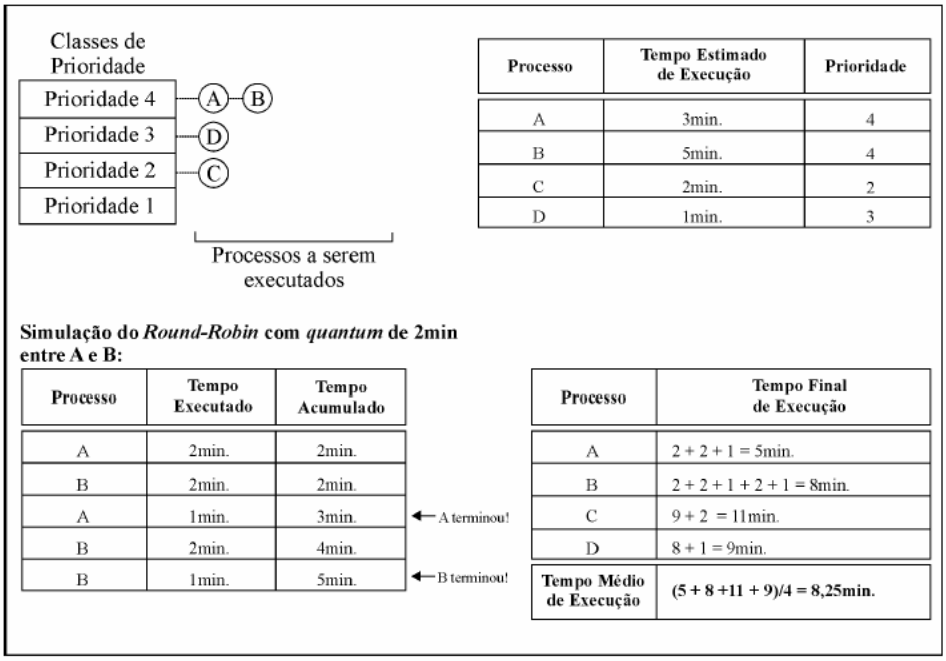
O *Round-Robin* apresenta como vantagens principais a sua simplicidade de implementação e, ainda, o fato de não permitir que processos monopolizem o processador. Já a sua desvantagem mais evidente é que ele não faz distinção entre a prioridade dos processos. Assim sendo, um processo de tempo-real vai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

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

5.10 ESCALONAMENTO POR PRIORIDADES

O algoritmo em questão é preemptivo e considera fatores externos para escolher qual processo irá rodar em um dado momento. Assim, cada processo deve possuir uma prioridade de execução a fim de que e o Gerente de Processos defina qual processo irá rodar. Normalmente, o processo com maior prioridade é aquele que deve ser executado primeiro. Contudo, existindo processos com prioridades iguais, os mesmos devem ser agrupados em classes e o *Round-Robin* é, geralmente, aplicado nas mesmas para determinar qual processo de uma classe irá rodar.

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



Processo	Tempo Estimado de Execução	Prioridade
A	3min.	4
B	5min.	4
C	2min.	2
D	1min.	3

Simulação do *Round-Robin* com *quantum* de 2min entre A e B:

Processo	Tempo Executado	Tempo Acumulado	
A	2min.	2min.	
B	2min.	2min.	
A	1min.	3min.	← A terminou!
B	2min.	4min.	
B	1min.	5min.	← B terminou!

Processo	Tempo Final de Execução
A	2 + 2 + 1 = 5min.
B	2 + 2 + 1 + 2 + 1 = 8min.
C	9 + 2 = 11min.
D	8 + 1 = 9min.
Tempo Médio de Execução	**(5 + 8 +11 + 9)/4 = 8,25min.**

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

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

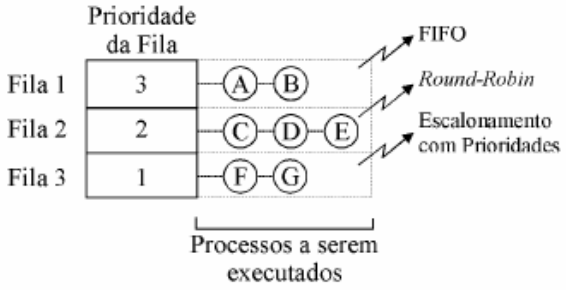
5.11 ESCALONAMENTO POR MÚLTIPLAS FILAS

Este algoritmo preemptivo, como o próprio nome sugere, considera a existência de várias filas de processos no estado de pronto, cada uma delas com sua prioridade de execução e seu algoritmo de escalonamento. Os processos são associados às filas em

função de características particulares como, por exemplo, tipo de processo, área de memória ocupada etc.

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

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



Processo	Tempo Estimado de Execução	Prioridade
A	3min.	-
B	5min.	-
C	2min.	-
D	1min.	-
E	4min.	-
F	6min.	3
G	7min.	1

Simulação do *Round-Robin* com *quantum* de 2min entre C,D e E:

Processo	Tempo Executado	Tempo Acumulado	
C	2min.	2min.	← C terminou!
D	1min.	1min.	← D terminou!
E	2min.	2min.	
E	2min.	4min.	← E terminou!

Processo	Tempo Final de Execução
A	3min.
B	3 + 5 = 8min.
C	8 + 2 = 10min.
D	8 + 2 + 1 = 11min.
E	8 + 2 + 1 + 2 + 2 = 15min.
F	15 + 6 = 21min.
G	15 + 6 + 7 = 28min.
Tempo Médio de Execução	**(3 + 8 +10 + 11 + 15 + 21 + 28)/7 = 13,71min.**

5.12 ESCALONAMENTO DE TEMPO REAL

Um objetivo primário dos algoritmos apresentados anteriormente, era garantir alta utilização de recursos. Processos que devem executar periodicamente (como uma vez por minuto) requerem algoritmos de escalonamento diferentes. Por exemplo, os tempos de espera ilimitados no SJF podem ser catastróficos para um processo que verifique a temperatura de um reator nuclear. Similarmente, um sistema que use SRT para escalonar um processo que reproduza um videoclipe produziria uma reprodução entrecortada. **Escalonamento de Tempo Real** atende às necessidades de processos que devem produzir saídas corretas em determinado momento (ou seja, que tem uma **restrição de tempo**). Um processo de tempo real pode dividir suas instruções em tarefas isoladas, cada qual deve concluir em determinado prazo. Outros processos de tempo real podem executar certa tarefa periodicamente, como atualizar as localizações de aviões em um sistema de controle aéreo. Escalonadores de tempo real devem garantir que as restrições de tempo sejam cumpridas.

Estratégias de escalonamento de tempo real são classificadas conforme quão bem cumprem os prazos de um processo. **Escalonamento de tempo real não crítico** garante que processos de tempo real sejam despachados antes de outros processos do sistema, mas não garante qual processo, se é que algum deles, cumprirá suas restrições de tempo.

O **Escalonamento de tempo real crítico** garante que as restrições de prazo de um processo sejam sempre atendidas. Cada tarefa especificada por um processo de tempo

real crítico deve concluir antes de seu prazo final; caso isso não aconteça, os resultados poderão ser catastróficos, entre eles trabalho inválido, falha de sistema ou até danos aos usuários do sistema. Sistemas de tempo real crítico podem conter **processos periódicos** que realizam suas computações em intervalos de tempo regulares (por exemplo, coletar dados de controle de tráfego aéreo a cada segundo) e **processos assíncronos** que executam em respostas a eventos (por exemplo, responder a altas temperaturas no núcleo de uma usina nuclear).

5.12.1 ESCALONAMENTO DE TEMPO REAL ESTÁTICOS

Esses algoritmos não ajustam a prioridade do processo ao longo do tempo. Porque as prioridades são calculadas somente uma vez, esses algoritmos tendem a ser simples e a incorrer em pouca sobrecarga. Eles são limitados, pois não podem se ajustar ao comportamento variável do processo e dependem de os recursos estarem funcionando e à disposição para garantir que sejam cumpridas as restrições de tempo.

Sistemas de tempo real críticos tendem a usar algoritmos de escalonamento estáticos, porque incorrem em baixa sobrecarga e é relativamente fácil de provar que as restrições de prazo de cada processo sejam atendidas. O algoritmo de escalonamento por **taxa monotônica** ou **RM** (Rate Monotonic), por exemplo, é um algoritmo de alternância circular, preemptivo, por prioridade, que eleva a prioridade de um processo linearmente (monotonicamente) com a freqüência (taxa) com a qual ele deve executar. Esse algoritmo de escalonamento estático favorece processos periódicos que executam frequentemente. O algoritmo por **taxa monotônica com prazo** pode ser usado quando um processo periódico especifica um prazo que não seja igual ao seu período.

5.12.2 ESCALONAMENTO DE TEMPO REAL DINÂMICOS

Escalonam processos ajustando suas prioridades durante a execução, o que pode causar sobrecarga significativa. Alguns algoritmos tentam minimizar a sobrecarga de escalonamento designando prioridades estáticas a alguns processos, e prioridades dinâmicas a outros.

O algoritmo de escalonamento por **prazo mais curto primeiro** (Earliest Deadline First – EDF) é do tipo preemptivo que despacha primeiro o processo com prazo mais curto. Se o processo que chegar tiver um prazo mais curto do que o processo em execução, o sistema provocará a preempção do processo em execução e despachará o que acabou de chegar. O objetivo é minimizar o rendimento cumprindo os prazos do maior número de processos por unidade de tempo (análogo ao SRT) e minimizando o tempo médio de espera (o que evita que processos curtos percam seus prazos enquanto processos longos executam). Estudos provam que, se um sistema fornecer preempção por hardware (temporizadores de interrupção) e os processos de tempo real que estão em escalonamento não forem interdependentes, o EDF minimiza a quantidade de tempo pelo qual o projeto "mais atrasado" perde seu prazo. Todavia, muitos sistemas de tempo real não fornecem preempção por hardware, portanto outros algoritmos devem ser empregados.

5.13 EXERCÍCIOS

58) O que é política de escalonamento de um Sistema Operacional?

59) Qual é a função do escalonador e do despachante?

60) Quais os principais critérios utilizados em uma política de escalonamento?

61) Diferencie os tempos de processador, espera, retorno e resposta.

62) Qual é a diferença entre escalonamento preemptivo e não-preemptivo?

63) Qual a diferença entre os escalonamentos FIFO e RR?

64) O que é fatia de tempo?

65) Descreva o escalonamento SJF e o escalonamento por prioridades.

66) Considere um Sistema Operacional com escalonamento por prioridades onde a avaliação do escalonamento é realizada em um intervalo mínimo de 5 ms. Neste sistema, os processos A e B competem por uma única CPU. Desprezando os tempos de processamento relativo às funções do Sistema Operacional, a tabela a seguir fornece os estados dos processos A e B ao longo do tempo, medido em intervalos de 5 ms (E=execução, P=pronto e B=bloqueado). O processo A tem menor prioridade que o processo B

	00-04	05-09	10-14	15-19	20-24	25-29	30-34	35-39	40-44	45-49
procA	P	P	E	E	E	P	P	P	E	B
procB	E	E	B	B	P	E	E	E	B	B

	50-54	55-59	60-64	65-69	70-74	75-79	80-84	85-89	90-94	95-99
procA	P	E	P	P	E	E	B	B	P	E
procB	B	P	E	E	B	B	P	E	E	-

a) Em que tempos A sofre preempção?
b) Em que tempos B sofre preempção?
c) Refaça a tabela anterior supondo que o processo A é mais prioritário que o processo B.

67) Tomando como base os dados da tabela apresentada a seguir, ignorando a perda de tempo causada pelas trocas de contexto e considerando que nenhum processo realiza entrada/saída, informe o tempo que cada um dos processos efetivamente levará para executar se os seguintes algoritmos de escalonamento forem utilizados: a) Round-Robin com quantum de 2 minutos; b) SJF; c) Escalonamento com Prioridade.

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

68) Explique o funcionamento da técnica de envelhecimento (*aging*) e quando ela pode ser utilizada.

69) Verificamos uma situação na qual um processo A, de alta prioridade, e um processo B de baixa prioridade interagem de forma a levar A a um loop eterno. Tal situação persistiria se utilizássemos o escalonamento round robin em vez do escalonamento com prioridade?

70) Os escalonadores round robin normalmente mantêm uma fila com todos os processos prontos, com cada processo aparecendo uma vez nesta fila. O que pode acontecer se determinado processo aparecer duas vezes na fila de prontos? Você pode encontrar alguma razão para que isto seja permitido?

71) Que tipos de critérios devem ser utilizados no momento da definição da fatia de tempo a ser empregada em um determinado sistema?

72) Em um sistema operacional, o escalonador de curto prazo utiliza duas filas. A fila "A" contém os processos do pessoal do CPD e a fila "B" contém os processos dos alunos. O algoritmo entre filas é fatia de tempo. De cada 11 unidades de tempo de processador, 7 são fornecidas para os processos da fila "A", e 4 para os processos da fila "B". O tempo de cada fila é dividido entre os processos também por fatias de tempo, com fatias de 2 unidades para todos. A tabela abaixo mostra o conteúdo das duas filas no instante zero. Considere que está iniciando um ciclo de

11 unidades, e agora a fila "A" vai receber as suas 7 unidades de tempo. Mostre a sequência de execução dos processos, com os momentos em que é feita a troca.

OBS: Se terminar a fatia de tempo da fila "X" no meio da fatia de tempo de um dos processos, o processador passa para a outra fila. Entretanto, esse processo permanece como primeiro da fila "X", até que toda sua fatia de tempo seja consumida.

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

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

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

74) Um algoritmo de escalonamento de CPU determina a ordem para execução dos seus processos escalonados. Considerando n processos a serem escalonados em um processador, quantos escalonamentos diferentes são possíveis? Apresente uma fórmula em termos de n.

75) Considere um Sistema Operacional que implemente escalonamento circular com quantum igual a 10 ut. Em um determinado instante de tempo, existem apenas três processos (P1, P2 e P3) na fila de pronto, e o tempo de CPU de cada processo é 18, 4 e 13 ut, respectivamente. Qual o estado de cada processo no instante de tempo T, considerando a execução dos processos P1, P2 e P3, nesta ordem, e que nenhuma operação de E/S é realizada?
a) T = 8 ut b) T = 11 ut c) T = 33 ut

76) Considere um Sistema Operacional que implemente escalonamento circular com quantum igual a 10 ut. Em um determinado instante de tempo, existem apenas três processos (P1, P2 e P3) na fila de pronto, e o tempo de CPU de cada processo é 14, 4 e 12 ut, respectivamente. Qual o estado de cada processo no instante de tempo T, considerando a execução dos processos P1, P2 e P3, nesta ordem, e que apenas o processo P1 realiza operações de E/S? Cada operação de E/S é executada após 5 ut e consome 10 ut.
a) T = 8 ut b)T = 18 ut c)T = 28 ut

77) Considere as seguintes tabelas. Elabore o gráfico de Gantt para os algoritmos SJF preemptivo, RR e prioridade. Calcule os tempos médios de retorno, resposta e espera para cada uma das situações. Considere uma fatia de tempo de 2 instantes.

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

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

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

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

79) Quando o algoritmo RR se degenera para o FIFO?

80) Quando o algoritmo MLF se degenera para o EDF? Isso pode ocorrer?

81) Com o escalonamento HRRN, processos curtos são sempre escalonados antes dos longos? Justifique sua resposta.

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

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

a) Algoritmos: Fila 1 – FCFS; Fila 2 – Prioridade; Fila 3 – RR, com quantum = 3
Algoritmo entre filas: RR com quantum = 4

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

b) Algoritmos: Fila 1 – SJF; Fila 2 – FIFO; Fila 3 – RR, com quantum = 2

Algoritmo entre filas: prioridade. Fila 1 = 2, Fila 2 = 3 e Fila 3 = 1

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

84) O processo P1 possui um tempo de CPU de 5 segundos e está esperando há 20 segundos. O processo P2 possui um tempo de CPU de 3 segundos e está esperando há 9. Se o sistema usar HRRN, qual processo executará primeiro?

85) Elabore o Gráfico de Gantt utilizando os algoritmos RMS e EDF, considerando os processos em tempo real com as seguintes características e que todos chegaram no instante zero.

PROC	Tcpu	Prazo (D)
A	10	20
B	8	10
C	4	25
D	7	15

-X-

6 Gerência de Memória

"Uma grande memória não faz um filósofo, bem como não pode um dicionário ser chamado de gramática."
(John Henry, cardeal Newman)

6.1 INTRODUÇÃO

Historicamente, a memória principal sempre foi vista como um recurso escasso e caro. Uma das maiores preocupações dos projetistas foi desenvolver Sistemas Operacionais que não ocupassem muito espaço de memória e, ao mesmo tempo, otimizassem a utilização dos recursos computacionais. Mesmo com a redução do custo e conseqüente aumento da capacidade, seu gerenciamento é um dos fatores mais importantes no projeto de Sistemas Operacionais.

Nós (na qualidade de projetistas de sistemas) consideramos a memória principal em termos de organização de memória. Colocamos apenas um único processo na memória principal ou incluímos vários processos ao mesmo tempo (ou seja, implementamos a multiprogramação)? Se a memória principal contiver diversos processos simultaneamente, damos a cada um a mesma quantidade de espaço ou dividimos a memória principal em porções de tamanhos diferentes? Definimos partições rigidamente por períodos estendidos ou dinamicamente, permitindo que o sistema se adapte rapidamente às mudanças das necessidades dos processos? Exigimos que processos executem em uma partição específica ou em qualquer lugar onde couberem? Exigimos que um sistema coloque cada processo em um bloco contíguo de localizações de memória ou permitimos que divida processos em blocos separados e os coloque em quaisquer molduras disponíveis na memória principal? Vamos falar sobre tudo isso.

6.2 FUNÇÕES BÁSICAS

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

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

6.3 ALOCAÇÃO CONTÍGUA SIMPLES

Foi implementada nos primeiros Sistemas Operacionais, porém ainda está presente em alguns sistemas monoprogramáveis. Nesse tipo de organização, a memória principal é subdividida em duas áreas: uma para o Sistema Operacional e outra para o programa do usuário. Dessa forma, o programador deve desenvolver suas aplicações preocupado apenas, em não ultrapassar o espaço de memória disponível.

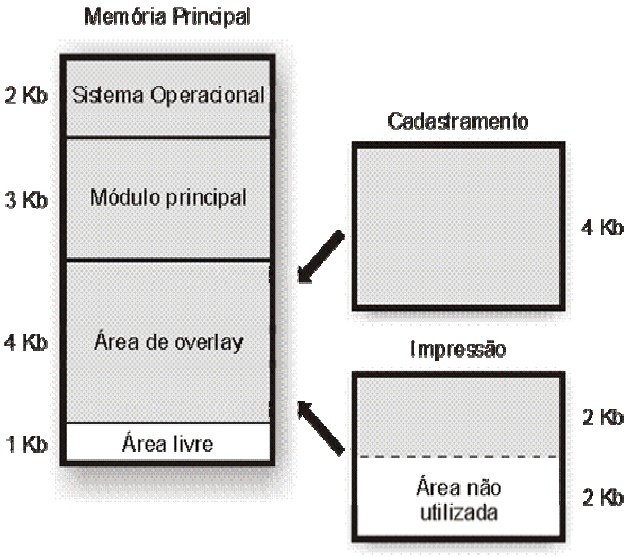
Nesse esquema, o usuário tem controle sobre toda a memória principal, podendo ter acesso a qualquer posição de memória, inclusive a área do Sistema Operacional. Para proteger o sistema desse tipo de acesso, que pode ser intencional ou não, alguns sistemas implementam proteção através de um registrador que delimita as áreas do Sistema Operacional e do usuário. Dessa forma, sempre que um programa faz referência a um endereço na memória, o sistema verifica se o endereço está dentro dos limites permiti-

dos. Caso não esteja, o programa é cancelado e uma mensagem de erro é gerada, indicando que houve uma violação no acesso à memória principal.

6.4 TÉCNICA DE *OVERLAY*

Na alocação contígua simples, todos os programas estão limitados ao tamanho da área de memória principal disponível para o usuário. Uma solução encontrada é dividir o programa em módulos, de forma que seja possível a execução independente de cada módulo, utilizando uma mesma área de memória. Essa técnica é chamada de ***overlay***.

Considere um programa que tenha três módulos: um principal, um de cadastramento e outro de impressão, sendo os módulos de cadastramento e de impressão independentes. A independência do código significa que quando um módulo estiver na memória para execução, o outro não precisa necessariamente estar presente. O módulo principal é comum aos dois módulos; logo, deve permanecer na memória durante todo o tempo da execução do programa.



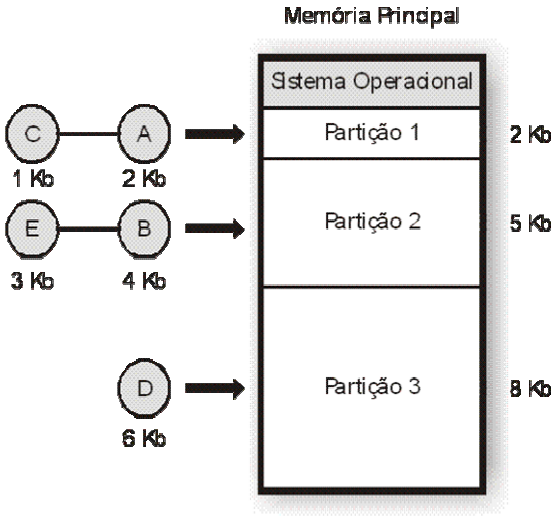
Como podemos verificar na figura, a memória é insuficiente para armazenar todo o programa, que totaliza 9KB. A técnica de *overlay* utiliza uma área de memória comum, onde os módulos de cadastramento e de impressão poderão compartilhar a mesma área de memória. Sempre que um dos dois módulos for referenciado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

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

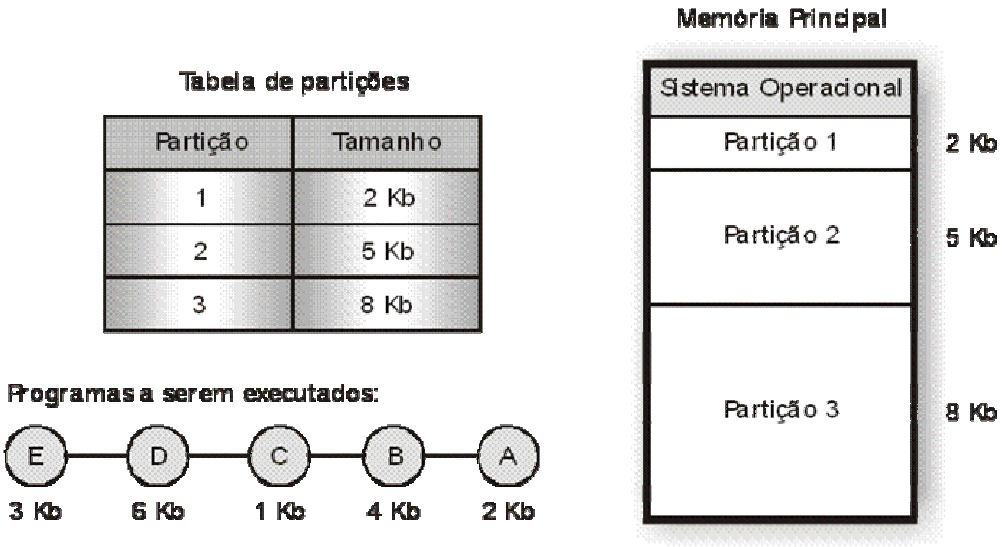
6.5 ALOCAÇÃO PARTICIONADA

Os sistemas multiprogramáveis são muito mais eficientes no uso do processador, necessitando assim, que diversos programas estejam na memória principal ao mesmo tempo e que novas formas de gerência de memória sejam implementadas.

6.5.1 ALOCAÇÃO PARTICIONADA ESTÁTICA

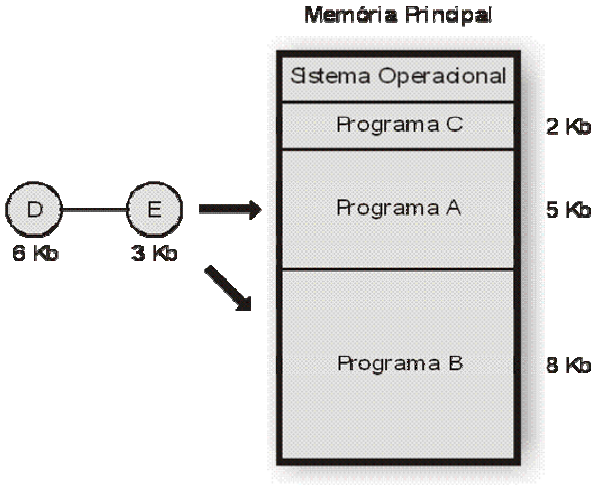


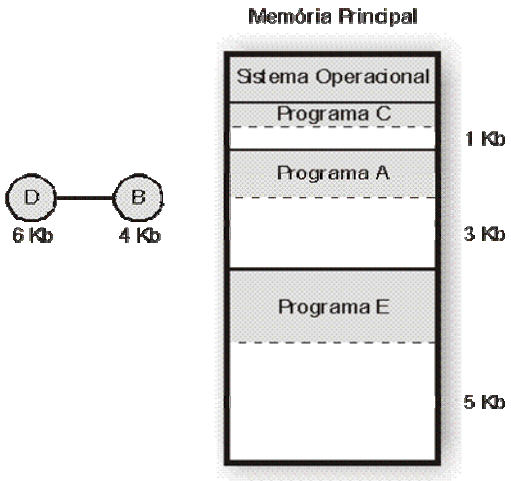
Nos primeiros sistemas multiprogramáveis, a memória era dividida em pedaços de tamanho fixo, chamados **partições**. O tamanho das partições, estabelecido na fase de inicialização do sistema, era definido em função do tamanho dos programas que executariam no ambiente. Sempre que fosse necessária a alteração do tamanho de uma partição, o sistema deveria ser desativado e reinicializado com uma nova configuração. Esse tipo de gerência de memória é conhecido como **alocação particionada estática** ou **alocação fixa**.



Inicialmente, os programas só podiam ser carregados em apenas uma partição específica, mesmo se outras estivessem disponíveis. Essa limitação se devia aos compiladores e montadores, que geravam apenas código absoluto. No **código absoluto**, todas as referências a endereços no programa são posições físicas na memória principal, ou seja, o programa só poderia ser carregado a partir do endereço de memória especificado no seu próprio código. Se, por exemplo, os programas A e B estivessem sendo executados, e a terceira partição estivesse livre, os programas C e E não poderiam ser processados. A esse tipo de gerência de memória chamou-se **alocação particionada estática absoluta**.

Com a evolução dos compiladores, o código gerado deixou de ser absoluto e passa a ser relocável. No **código relocável**, todas as referências a endereços no programa são relativas ao início do código e não a endereços físicos de memória. Desta forma, os programas puderam ser executados a partir de qualquer partição. Quando o programa é carregado, o *loader* calcula todos os endereços a partir da posição inicial onde o programa foi alocado. Caso os programas A e B terminassem, o programa E poderia ser executado em qualquer uma das partições. Esse tipo de gerência é denominado **alocação particionada estática relocável**.





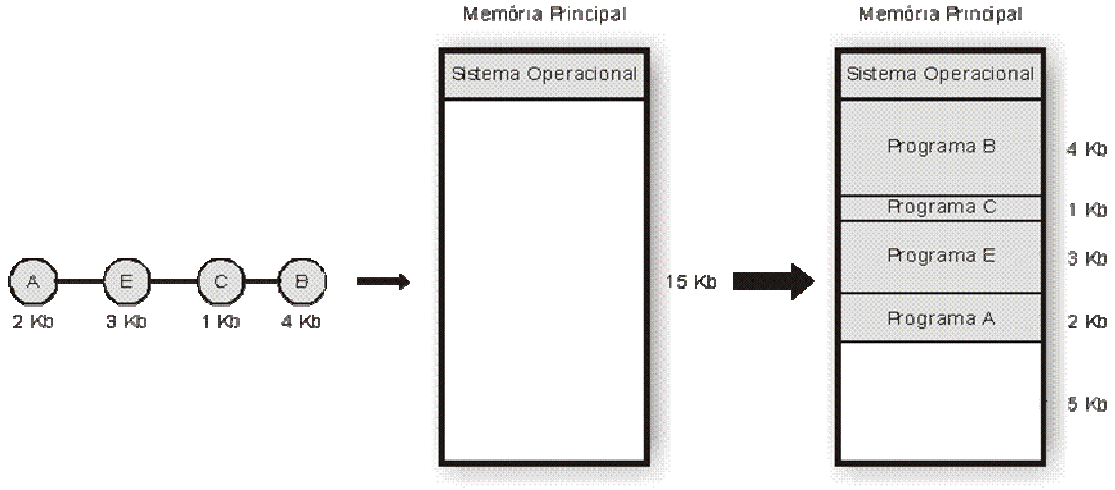
Para manter controle sobre quais partições estão alocadas, a gerência de memória mantém uma tabela com o endereço inicial de cada partição, seu tamanho, e se está em uso. Sempre que um programa é carregado para a memória, o sistema percorre a tabela, na tentativa de localizar uma partição livre, onde o programa possa ser carregado.

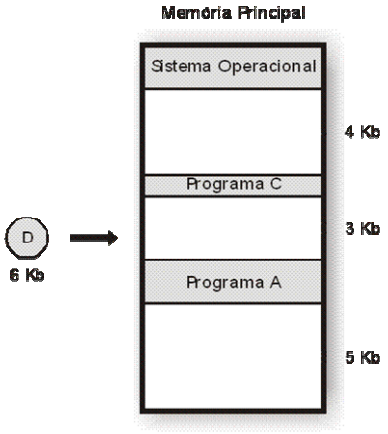
Nesse esquema de alocação de memória, a proteção baseia-se em dois registradores, que indicam os limites inferior e superior da partição onde o programa está sendo executado.

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

6.5.2 ALOCAÇÃO PARTICIONADA DINÂMICA

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





Nesse esquema, o programa utilizaria o espaço necessário, tornando essa área sua partição. Como os programas utilizam apenas o espaço que necessitam, o problema da fragmentação interna não ocorre. Porém, nesse caso, existe um problema que não é tão óbvio quanto no esquema anterior. Um tipo diferente de fragmentação começará a ocorrer, quando os programas forem terminando e deixando espaços cada vez menores na memória, não permitindo o ingresso de novos programas. Na figura, mesmo existindo 12 KB livres de memória principal, o programa D, que necessita de 6 KB não poderá ser carregado para execução, pois este espaço não está disposto contiguamente. Esse tipo de problema é chamado de **fragmentação externa**.

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

6.5.3 ESTRATÉGIAS DE ALOCAÇÃO DE PARTIÇÃO

Para que diversos programas estejam na memória principal ao mesmo tempo, novas formas de gerência de memória devem ser implementadas. Os Sistemas Operacionais implementam algumas estratégias para determinar em qual área livre um programa será carregado para execução. Essas estratégias visam evitar ou diminuir o problema da fragmentação externa.

A melhor estratégia a ser adotada por um sistema depende de uma série de fatores, sendo o mais importante o tamanho dos programas acessados no ambiente. Indepen-

dentemente do algoritmo utilizado, o sistema deve possuir formas de saber quais áreas estão livres e quais não estão. Existem duas estratégias principais: Mapa de Bits e Lista Encadeada.

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

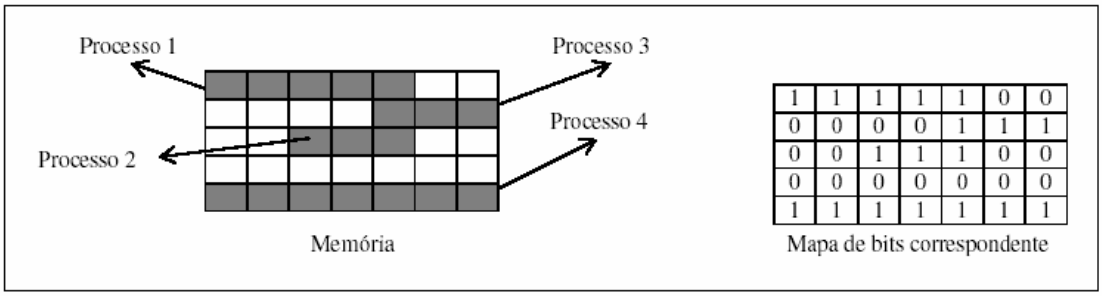
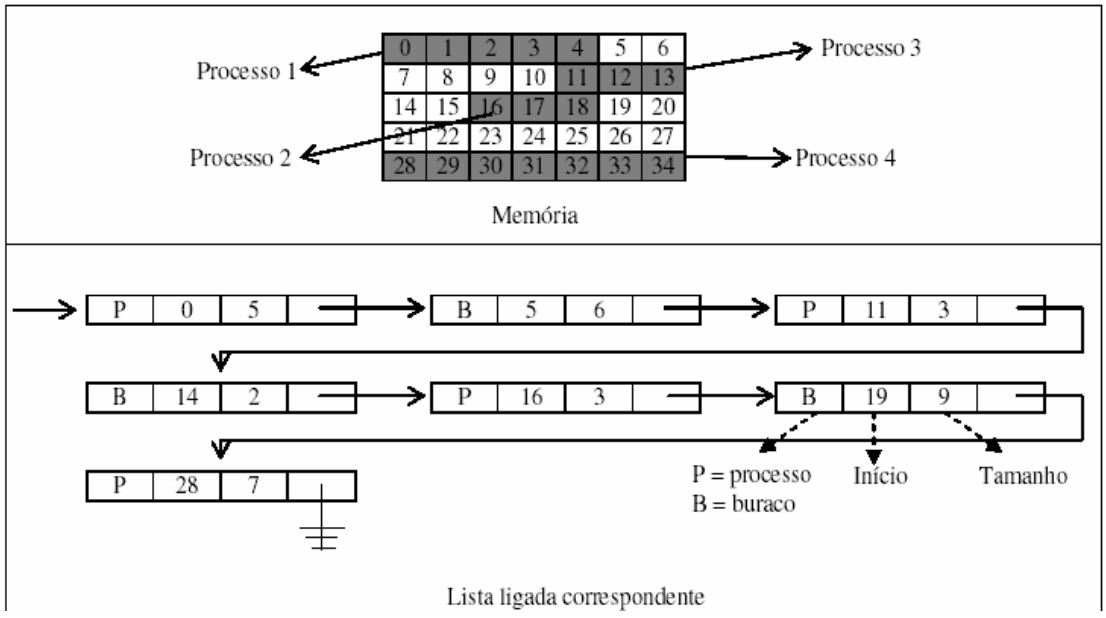


Figura 3.5: Estratégia de alocação/liberação de memória que utiliza mapa de bits.

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

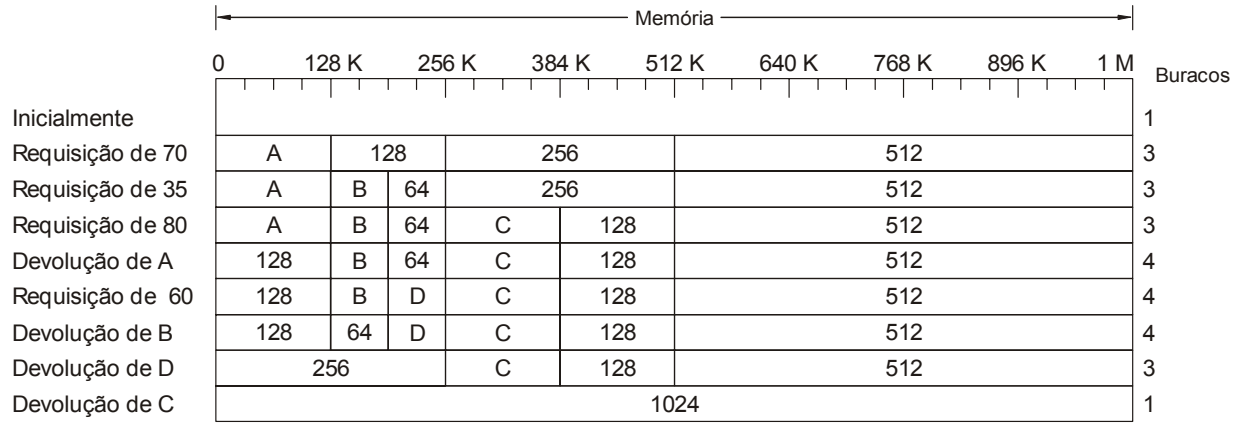


No esquema de listas ligadas podem ser utilizados os algoritmos de alocação de memória apresentados na Tabela a seguir:

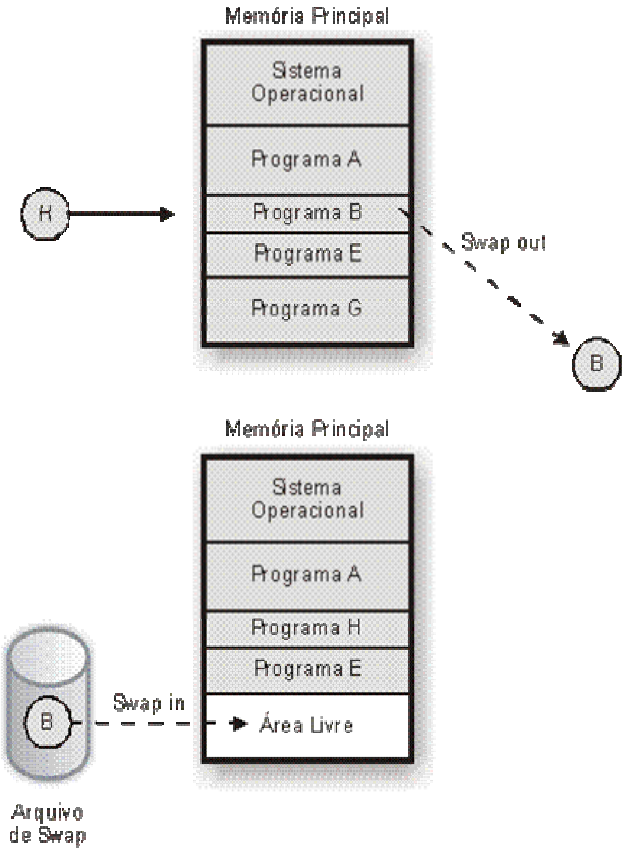
ALGORITMO	DESCRIÇÃO
Primeira Alocação	O Gerente de Memória procura ao longo da lista de segmentos até encontrar um buraco que seja suficientemente grande para abrigar o processo. O buraco é então quebrado em 2 pedaços, um para o processo e outro para o espaço não-utilizado, exceto no caso, altamente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

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

Apesar de extremamente eficiente sob o aspecto da velocidade, o sistema *buddy* não é eficiente em termos de utilização da memória. O problema decorre, obviamente, da necessidade de se arredondar a requisição feita pelo processo para a próxima potência inteira de 2. Assim, a um processo de 35K deve ser alocado 54K. Os 29K excedentes serão perdidos, ocasionando a fragmentação interna.



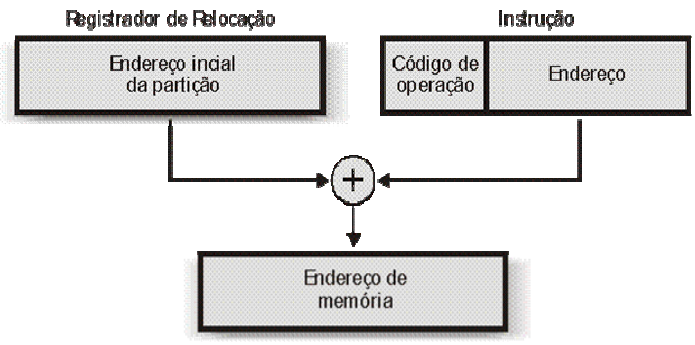
6.6 SWAPPING



Mesmo com o aumento da eficiência da multiprogramação e, particularmente, da gerência de memória, muitas vezes um programa não podia ser executado por falta de uma partição livre disponível. A técnica de ***swapping*** foi introduzida para contornar o problema da insuficiência de memória principal.

Em todos os esquemas apresentados anteriormente, um processo permanecia na memória principal até o final da sua execução, inclusive nos momentos em que esperava por um evento, como uma operação de E/S. O *swapping* é uma técnica aplicada à gerência de memória para programas que esperam por memória livre para serem executados. Nesta situação, o sistema escolhe um processo residente, que é transferido da memória principal para a memória secundária (*swap out*), geralmente disco. Posteriormente, o processo é carregado de volta da memória secundária para a memória principal (*swap in*) e pode continuar sua execução como se nada tivesse ocorrido.

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



No momento em que o programa é carregado na memória, o registrador recebe o endereço inicial da posição de memória que o programa irá ocupar. Toda vez que ocorrer uma referência a algum endereço, o endereço contido na instrução será somado ao conteúdo do registrador, gerando, assim, o endereço físico. Dessa forma, um programa pode ser carregado em qualquer posição de memória.

6.7 EXERCÍCIOS

86) Quais as funções básicas da gerência de memória?

87) Considere um sistema computacional com 40 KB de memória principal e que utilize um Sistema Operacional de 10 KB que implemente alocação contígua de memória. Qual a taxa de subutilização da memória principal para um programa que ocupe 20 KB de memória?

88) Suponha que um sistema computacional de 64 KB de memória principal e que utilize um Sistema Operacional de 14 KB que implemente alocação contígua de memória. Considere também um programa de 90 KB, formado por um módulo principal de 20 KB e três módulos independentes, cada um com 10 KB, 20 KB e 30 KB. Como o programa poderia ser executado utilizando-se apenas a técnica de overlay?

89) Considerando o exercício anterior, se o módulo de 30 KB tivesse seu tamanho aumentado para 40 KB, seria possível executar o programa? Caso não possa, como o problema poderia ser contornado?

90) Qual a diferença entre fragmentação interna e externa da memória principal?

91) Suponha um sistema computacional com 128 KB de memória principal e que utilize um Sistema Operacional de 64 KB que implemente alocação particionada estática relocável. Considere também que o sistema foi inicializado com três partições: P1 (8 KB), P2 (24 KB) e P3 (32 KB). Calcule a fragmentação interna da memória principal após a carga de três programas: PA, PB e PC.
    a) P1 <- PA (6 KB); P2 <- PB (20 KB); P3 <- PC (28 KB)
    b) P1 <- PA (4 KB); P2 <- PB (16 KB); P3 <- PC (26 KB)
    c) P1 <- PA (8 KB); P2 <- PB (24 KB); P3 <- PC (32 KB)

92) Considerando o exercício anterior, seria possível executar quatro programas concorrentemente utilizando apenas a técnica de alocação particionada estática relocável? Se for possível, como? Considerando ainda o mesmo exercício, seria possível executar um programa de 32 KB? Se for possível, como?

93) Qual a limitação da alocação particionada estática absoluta em relação à alocação estática relocável?

94) Considere que os processos da tabela a seguir estão aguardando para serem executados e que cada um permanecerá na memória durante o tempo especificado. O Sistema Operacional ocupa uma área de 20 KB no início da memória e gerencia a memória utilizando um algoritmo de particionamento dinâmico modificado. A memória total disponível no sistema é de 64 KB e é alocada em blocos múltiplos de 4 KB. Os processos são alocados de acordo com sua identificação (em ordem crescente) e irão aguardar até obter a memória de que necessitam. Calcule a perda de memória por fragmentação interna e externa sempre que um processo é colocado ou retirado da memória. O Sistema Operacional compacta a memória apenas quando existem duas ou mais partições livres adjacentes.

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

95) Considerando os algoritmos para escolha da partição dinamicamente, conceitue as estratégias especificando prós e contras de cada uma.

96) Considere um sistema que possua as seguintes áreas livres na memória principal, ordenadas crescentemente: 10 KB, 4 KB, 20 KB, 18 KB, 7 KB, 9 KB, 12 KB e 15 KB. Para cada programa abaixo, qual seria a partição alocada utilizando-se os algoritmos "Primeira Alocação", "Próxima Alocação", "Melhor Alocação" e "Pior Alocação"?

a) 12 KB b) 10 KB c) 9 KB

97) Um sistema utiliza alocação particionada dinâmica como mecanismo de gerência de memória. O Sistema Operacional aloca uma área de memória total de 50 KB e possui, inicialmente, os programas da tabela a seguir:

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

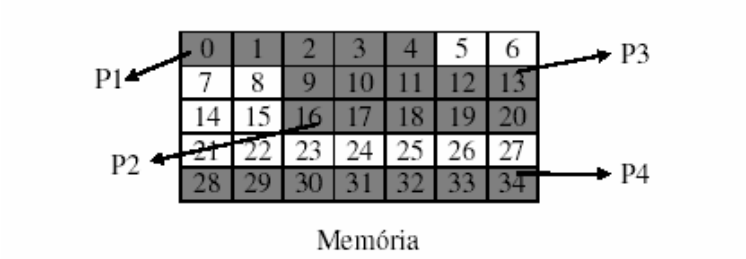
Realize as operações abaixo seqüencialmente, mostrando o estado da memória após cada uma delas. Resolva a questão utilizando as estratégias "Melhor Alocação", "Pior Alocação", "Primeira Alocação" e "Próxima Alocação".

a) alocar uma área para o programa D que possui 6 KB;
b) liberar a área do programa A;
c) alocar uma área para o programa E que possui 4 KB.

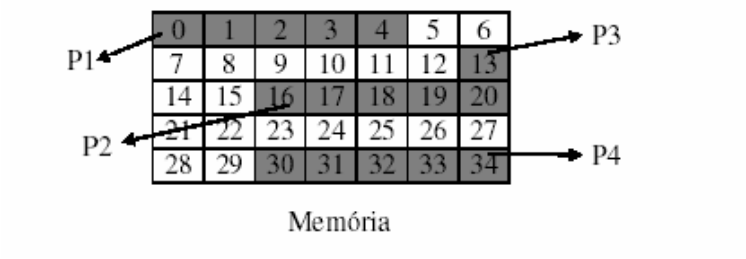
98) O que é swapping e para que é utilizada essa técnica?

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

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



Memória



Memória

101) Considere um sistema com swapping no qual os seguintes buracos estão na memória, na ordem apresentada: 17K, 4K, 20K, 18K, 7K, 9K, 11K e 15K. Consi-

dere, ainda, que foram sucessivamente carregados para a memória os processos A, B e C de tamanhos 18K, 9K e 14K, respectivamente. Assim sendo, aplique os algoritmos da Primeira Alocação e o da Próxima Alocação.

102) Um minicomputador usa o sistema buddy para gerenciar sua memória. Inicialmente, ele tem um bloco de 512K no endereço 0. Demonstre graficamente a situação da memória após cada passo e responda ao final de todos os passos: a) Quantos blocos livres restaram, b) quais os seus tamanhos e c) quais seus endereços.
   a) Alocação do Processo A de 12 KB
   b) Alocação do Processo B de 74 KB
   c) Alocação do Processo C de 30 KB
   d) Finalização do Processo B
   e) Alocação do Processo D de 200 KB
   f) Finalização do Processo A
   g) Alocação do Processo E de 7 KB

-X-

7 Memória Virtual

"Endereços servem para que escondamos nossa localização."
(Saki (H. H. Munro))

7.1 INTRODUÇÃO

As implementações vistas anteriormente no gerenciamento de memória se mostraram muitas vezes ineficientes. Além disso, o tamanho de um programa e de suas estruturas de dados estava limitado ao tamanho da memória disponível. A utilização da técnica de *overlay* para contornar este problema é de difícil implementação na prática e nem sempre uma solução garantida.

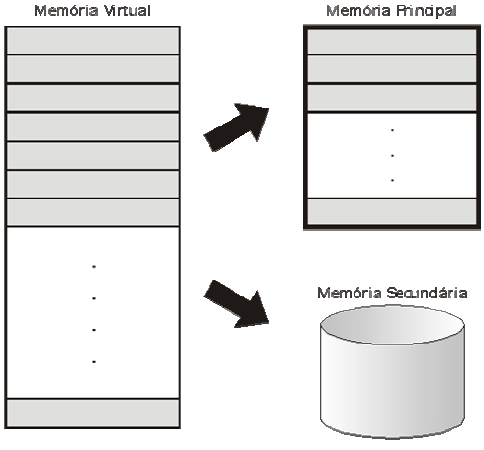
Memória Virtual é uma técnica sofisticada e poderosa de gerência de memória, onde as memórias principal e secundária são combinadas, dando ao usuário a ilusão de existir uma memória muito maior que a capacidade real da memória principal. O conceito de memória virtual fundamenta-se em não vincular o endereçamento feito pelo programa dos endereços físicos da memória principal. Desta forma, programas e suas estruturas de dados deixam de estar limitados ao tamanho da memória física disponível, pois podem possuir endereços associados à memória secundária.

Outra vantagem da técnica de memória virtual é permitir um número maior de processos compartilhando a memória principal, já que apenas partes de cada processo estarão residentes. Isto leva a uma utilização mais eficiente também do processador. Além disso, essa técnica possibilita minimizar o problema da fragmentação da memória principal.

7.2 ESPAÇO DE ENDEREÇAMENTO VIRTUAL

O conceito de memória virtual se aproxima muito da idéia de um vetor existente nas linguagens de alto nível. Quando um programa faz referência a um elemento do vetor, não há preocupação em saber a posição de memória daquele dado. O compilador se encarrega de gerar instruções que implementem esse mecanismo, tornando-o totalmente transparente ao programador.

A memória virtual utiliza abstração semelhante, só que em relação aos endereços dos programas e dados. Um programa no ambiente de memória virtual não faz referência a endereços físicos de memória (**endereços reais**), mas apenas a **endereços virtuais**. No momento da execução de uma instrução, o endereço virtual referenciado é traduzido para um endereço físico, pois o processador manipula apenas posições da memória principal. O mecanismo de tradução do endereço virtual para o endereço físico é chamado de **mapeamento**.



Como o espaço de endereçamento virtual não tem nenhuma relação direta com os endereços no espaço real, um programa pode fazer referência a endereços virtuais que estejam fora dos limites da memória principal, ou seja, os programas e suas estruturas de dados não estão mais limitados ao tamanho da memória física disponível. Para que isso seja possível, o Sistema Operacional utiliza a memória secundária como extensão da memória principal. Quando um programa é executado, somente uma parte do seu código fica residente na memória principal, permanecendo o restante na memória secundária até o momento de ser referenciado. Esta condição permite aumentar o comparti-

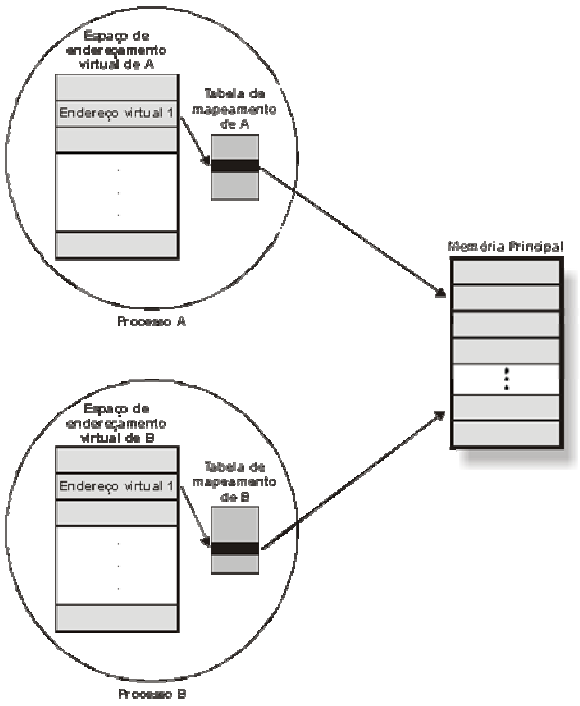
lhamento da memória principal entre muitos processos.

No desenvolvimento de aplicações, a existência dos endereços virtuais é ignorada pelo programador. Os compiladores e *linkers* se encarregam de gerar o código executável em função do espaço de endereçamento virtual, e o Sistema Operacional cuida dos detalhes durante sua execução.

7.3 MAPEAMENTO

O processador apenas executa instruções e referencia dados residentes no espaço de endereçamento real; portanto, deve existir um mecanismo que transforme os endereços virtuais em reais. Esse mecanismo é chamado **mapeamento**.

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



Cada processo tem o seu espaço de endereçamento virtual como se possuísse sua própria memória. O mecanismo de tradução se encarrega, então, de manter **tabelas de mapeamento** exclusivas para cada processo, relacionando os endereços virtuais do processo às suas posições na memória real.

Caso o mapeamento fosse realizado para cada célula na memória principal, o espaço ocupado pelas tabelas seria tão grande quanto o espaço de endereçamento virtual de cada processo, o que inviabilizaria a implementação do mecanismo de memória virtual. Em função disso, as tabelas mapeiam blocos de dados, cujo tamanho determina o número de entradas existentes nas tabelas de mapeamento. Quanto maior o bloco, menos entradas nas tabelas de mapeamento e, conseqüentemente, tabelas de mapeamento que ocupam um menor espaço na memória.

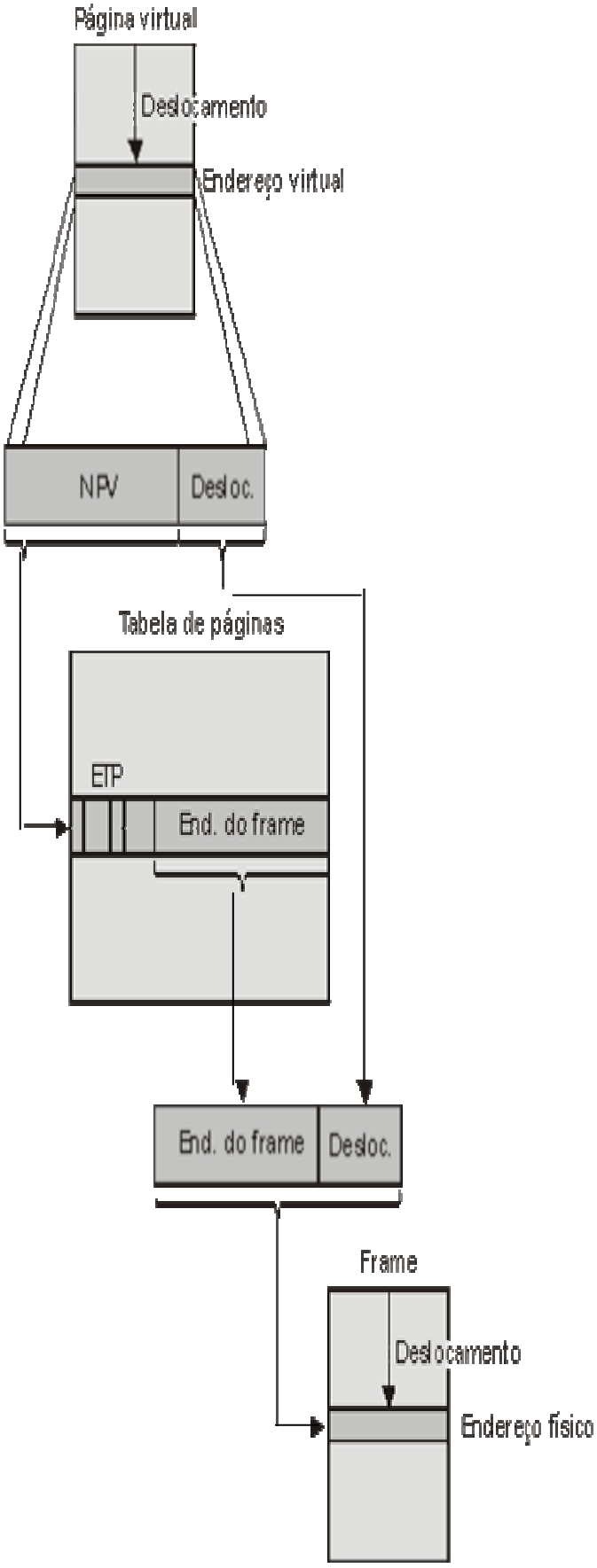
Espaço de Endereçamento Virtual	Tamanho do Bloco	Número de Blocos	Número de entradas na tabela de mapeamento
2^{32} endereços	512 bytes	2^{23}	2^{23}
2^{32} endereços	4 KB	2^{20}	2^{20}
2^{64} endereços	4 KB	2^{52}	2^{52}
2^{64} endereços	64 KB	2^{48}	2^{48}

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

7.4 PAGINAÇÃO

É a técnica de gerência de memória onde o espaço de endereçamento virtual e o real são divididos em blocos do mesmo tamanho chamados **páginas**. A definição do **tamanho da página** é um fator importante no projeto de sistemas que implementam memória virtual por paginação. O tamanho da página está associado à arquitetura do hardware e varia de acordo com o processador, mas normalmente está entre 512 e 16MB. Páginas

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

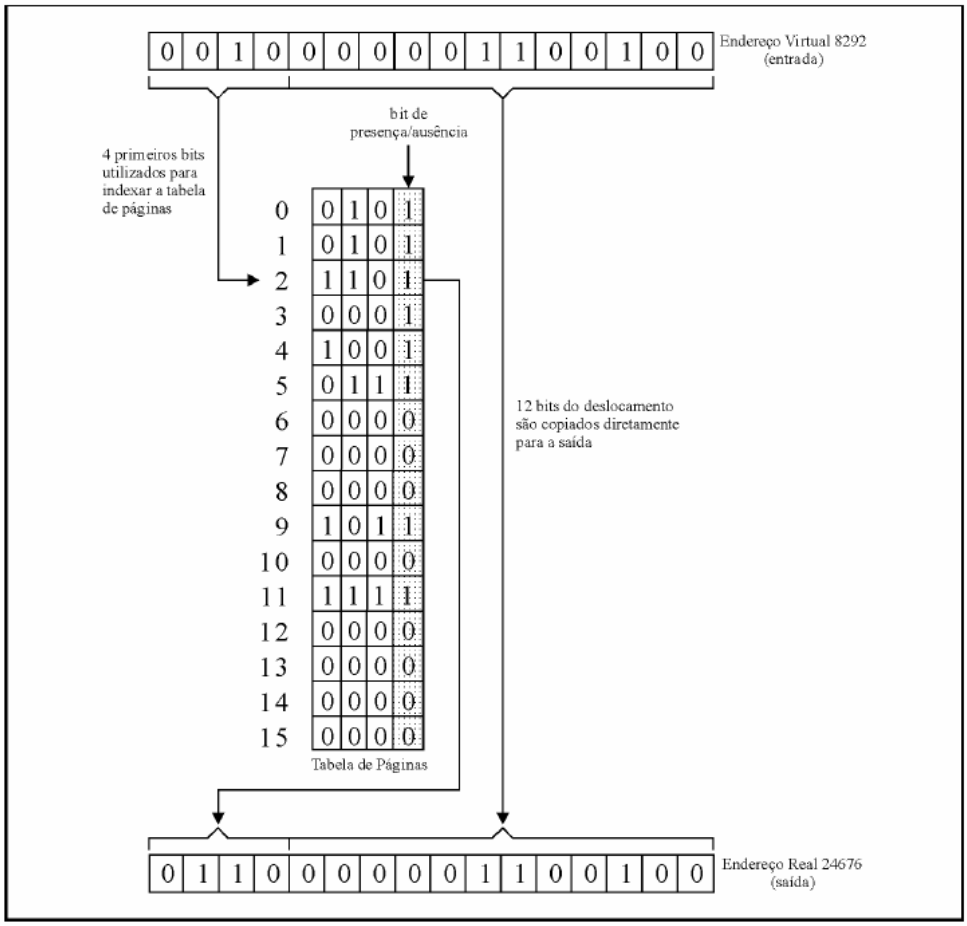


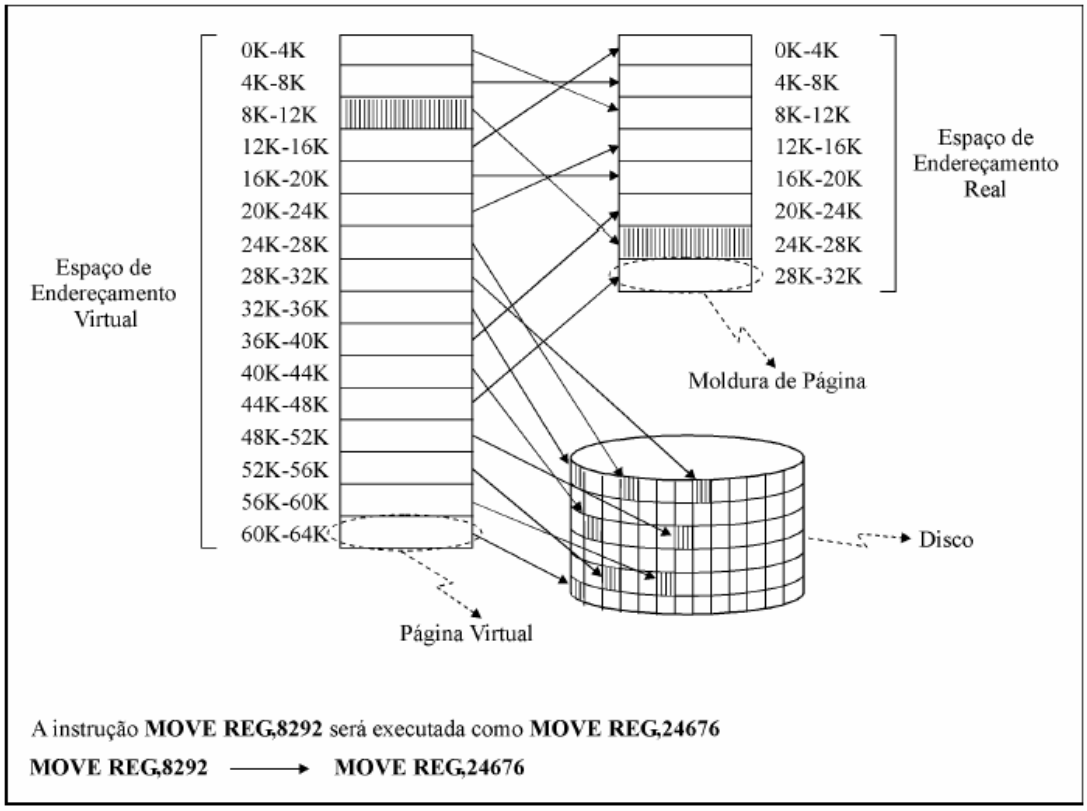
Todo o mapeamento de endereço virtual em real é realizado através de **tabelas de páginas**. Cada processo possui sua própria tabela e cada página virtual do processo possui uma **entrada na tabela de páginas** (ETP), com informações de mapeamento que permitem ao sistema localizar a página real correspondente.

Nessa técnica, o endereço virtual é formado pelo **número da página virtual** (NPV) e por um **deslocamento**. O NPV identifica unicamente a página virtual que contém o endereço, funcionando como um índice na tabela de paginas. O deslocamento indica a posição do endereço virtual em relação ao início da página na qual se encontra. O endereço físico é obtido, então, combinando-se o endereço do *frame*, localizado na tabela de páginas, com o deslocamento, contido no endereço virtual.

Além da informação sobre a localização da página virtual, a ETP possui outras informações, como o **bit de validade** (*valid bit*) que indica se uma página está ou não na memória principal.

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

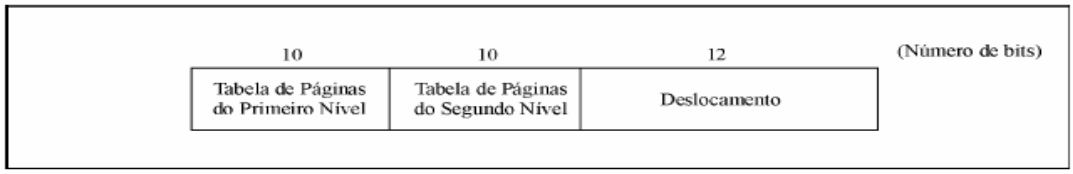

0 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0
Endereço Virtual 8292
(entrada)
bit de
presença/ausência
4 primeiros bits
utilizados para
indexar a tabela
de páginas
0 0 1 0 1
1 0 1 0 1
2 1 1 0 1
3 0 0 0 1
4 1 0 0 1
5 0 1 1 1
6 0 0 0 0
7 0 0 0 0
8 0 0 0 0
9 1 0 1 1
10 0 0 0 0
11 1 1 1 1
12 0 0 0 0
13 0 0 0 0
14 0 0 0 0
15 0 0 0 0
Tabela de Páginas
12 bits do deslocamento
são copiados diretamente
para a saída
0 1 1 0 0 0 0 0 0 1 1 0 0 1 0 0
Endereço Real 24676
(saída)


Espaço de
Endereçamento
Virtual
0K-4K
4K-8K
8K-12K
12K-16K
16K-20K
20K-24K
24K-28K
28K-32K
32K-36K
36K-40K
40K-44K
44K-48K
48K-52K
52K-56K
56K-60K
60K-64K
Página Virtual
0K-4K
4K-8K
8K-12K
12K-16K
16K-20K
20K-24K
24K-28K
28K-32K
Espaço de
Endereçamento
Real
Moldura de Página
Disco
A instrução MOVE REG,8292 será executada como MOVE REG,24676
MOVE REG,8292 ⟶ MOVE REG,24676

7.4.1 PAGINAÇÃO MULTINÍVEL

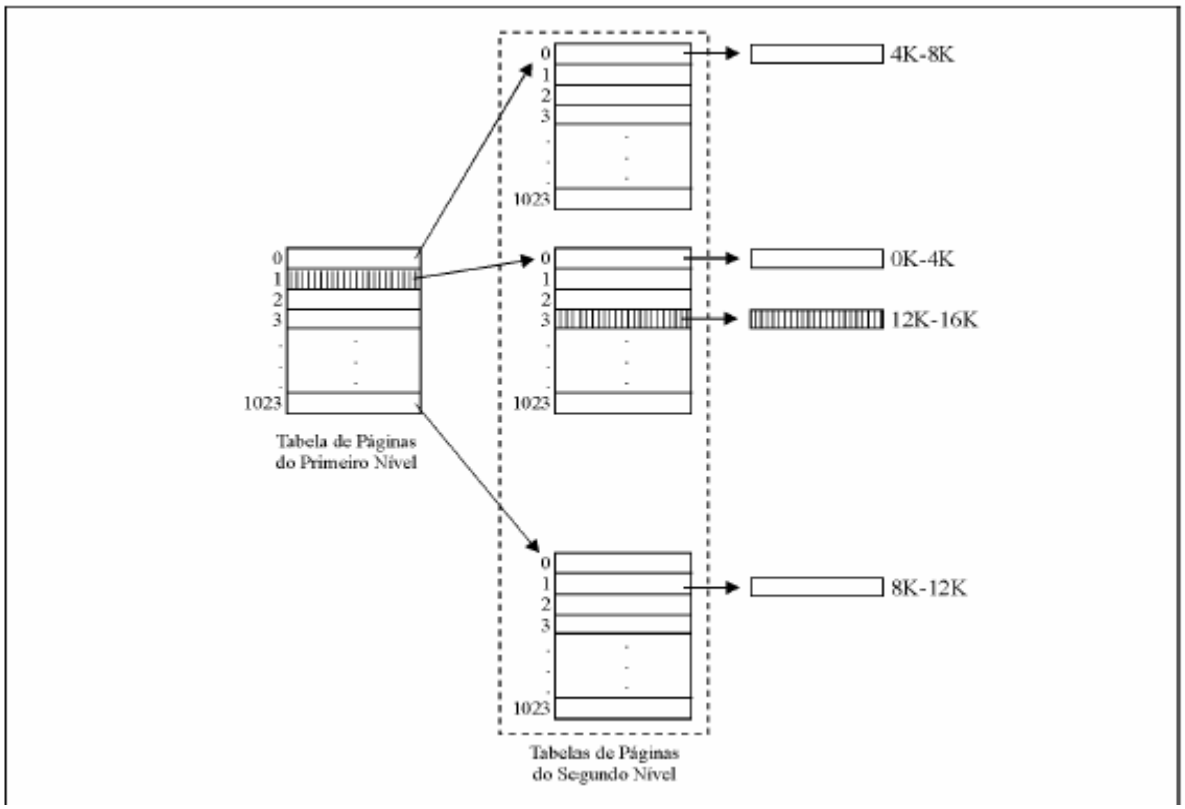
Em sistemas que implementam apenas um nível de paginação, o tamanho das tabelas de páginas pode ser um problema. Em uma arquitetura de 32 bits para endereçamento e páginas com 4KB por processo, onde cada entrada na tabela de páginas ocupe 4 bytes, a tabela de páginas poderia ter mais de um milhão de entradas e ocuparia 4 MB de espaço. Imaginando vários processos residentes na memória principal, manter tabelas desse tamanho para cada processo certamente seria de difícil gerenciamento.

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



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

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



1º) MOVE REG 4206600 → MOVE REG?

2º) 4206600 → 0000000001 | 0000000011 | 000000001000

3º) 0000000001 = 1 (TP1) – Endereços de 4Mb a 8 Mb de TP1

4º) 0000000011 = 3 (TP2) – Endereço da Base = 12.288

5º) 000000001000 = 8 (Deslocamento)

6º) Endereço Real=Base + Deslocamento = 12.288 + 8 =12.296 → MOVE REG 12296

7.4.2 POLÍTICAS DE BUSCA DE PÁGINAS

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

Na **paginação sob demanda**, as páginas dos processos são transferidas da memória secundária para a principal apenas quando são referenciadas. Este mecanismo é conveniente, na medida em que leva para a memória principal apenas as páginas realmente necessárias à execução do programa. Desse modo, é possível que partes não executadas do programa, como rotinas de tratamento de erros, nunca sejam carregadas para a memória.

Na **pré-paginação**, o sistema carrega para a memória principal, além da página referenciada, outras páginas que podem ou não ser necessárias ao processo ao longo do processamento. Se imaginarmos que o programa está armazenado seqüencialmente no disco, existe uma grande economia de tempo em levar um conjunto de páginas da memória secundária, ao contrário de carregar uma de cada vez. Por outro lado, caso o processo não precise das páginas carregadas antecipadamente, o sistema terá perdido tempo e ocupado memória principal desnecessariamente.

7.4.3 POLÍTICAS DE ALOCAÇÃO DE PÁGINAS

Determina quantas molduras (*frames*) cada processo pode manter na memória principal. Existem, basicamente, duas alternativas: alocação fixa e alocação variável.

Na **política de alocação fixa**, cada processo tem um número máximo de molduras que pode ser utilizado durante a execução do programa. Caso o número de páginas reais seja insuficiente, uma página do processo deve ser descartada para que uma nova seja carregada. O limite de páginas deve ser definido no momento da criação do processo, com base no tipo da aplicação que será executada. Essa informação faz parte do contexto de software do processo.

Apesar de sua simplicidade, a política de alocação fixa de página apresenta dois problemas. Se o número máximo de páginas alocadas for muito pequeno, o processo tenderá a ter um elevado número de falta de página, o que pode impactar no desempenho de todo o sistema. Por outro lado, caso o número de páginas seja muito grande, cada processo irá ocupar na memória principal um espaço maior do que o necessário, reduzindo o número de processos residentes e o grau de multiprogramação.

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

7.4.4 POLÍTICAS DE SUBSTITUIÇÃO DE PÁGINAS

Em algumas situações, quando um processo atinge o seu limite de alocação de molduras e necessita alocar novas páginas na memória principal, o Sistema Operacional deve selecionar, dentre as diversas páginas alocadas, qual deverá ser liberada. Este mecanismo é chamado de **política de substituição de páginas**. Uma página real, quando liberada por um processo, está livre para ser utilizada por qualquer outro. A partir dessa situação, qualquer estratégia de substituição de páginas deve considerar se uma página foi ou não modificada antes de liberá-la. Se a página tiver sido modificada, o sistema deverá gravá-la na memória secundária antes do descarte, preservando seu conteúdo para uso em futuras referências. Este mecanismo é conhecido como ***page out***.

O Sistema Operacional consegue identificar as páginas modificadas através de um bit que existe em cada entrada da tabela de páginas, chamado **bit de modificação**. Sempre que uma página sofre uma alteração, o valor do bit de modificação é alterado, indicando que a página foi modificada.

A política de substituição de páginas pode ser classificada conforme seu escopo, ou seja, dentre os processos residentes na memória principal quais são candidatos a ter páginas realocadas. Em função deste escopo, pode ser definida como local ou global.

Na **política de substituição local**, apenas as páginas do processo que gerou a falta de página são candidatas a realocação. Os *frames* dos demais processos não são avaliados para substituição.

Já na **política de substituição global**, todas as páginas alocadas na memória principal são candidatas a substituição, independente do processo que gerou a falta de página. Na verdade, nem todas as páginas podem ser candidatas a substituição. Algumas páginas, como as do núcleo do sistema, são marcadas como bloqueadas e não podem ser realocadas.

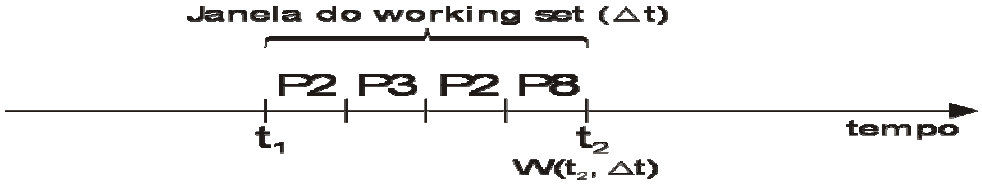
7.4.5 WORKING SET

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

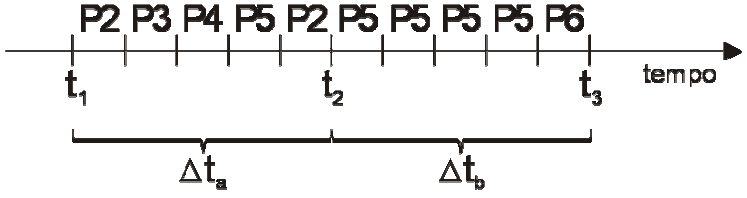
O conceito de *working set* surgiu com o objetivo de reduzir o problema do *trashing* e está relacionado ao **princípio da localidade**. Existem dois tipos de localidade que são observados durante a execução da maioria dos programas. A **localidade espacial** é a tendência de que após uma referência a uma posição de memória sejam realizadas novas referências a endereços próximos. A **localidade temporal** é a tendência de que após a referência a uma posição de memória esta mesma posição seja novamente referenciada em um curto intervalo de tempo.

O princípio da localidade significa, na prática, que o processador tenderá a concentrar suas referências a um conjunto de páginas do processo durante um determinado período de tempo. Imaginando um *loop*, cujo código ocupe três páginas, a tendência de essas três páginas serem referenciadas diversas vezes é muito alta.

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



Dentro da janela do *working set,* o número de páginas distintas referenciadas é conhecido como **tamanho do *working set***. Na figura são apresentadas as referências às páginas de um processo nas janelas Δt_a $(t_2 - t_1)$ e Δt_b $(t_3 - t_2)$. O *working set* do processo no instante t_2, com a janela Δt_a, corresponde às páginas P2, P3, P4 e P5, e o tamanho do *working set* é igual a quatro páginas. No instante t_3, com a janela Δt_b, o *working set* corresponde às páginas P5 e P6, e o tamanho é igual a duas páginas.



O modelo de *working set* proposto por Denning possibilita prever quais páginas são necessárias à execução de um programa de forma eficiente. Caso a janela do *working set* seja apropriadamente selecionada, em função da localidade do programa, o Sistema Operacional deverá manter as páginas do *working set* de cada processo residente na memória principal. Considerando que a localidade de um programa varia ao longo da sua execução, o tamanho do *working set* do processo também varia, ou seja, o seu limite de páginas reais deve acompanhar esta variação. O *working set* refletirá a localidade do programa, reduzindo a taxa de paginação dos processos e evitando, conseqüentemente, o *trashing*. Na prática, o modelo de *working set* serve como base para inúmeros algoritmos de substituição de páginas, como os apresentados a seguir.

7.4.6 ALGORITMOS DE SUBSTITUIÇÃO DE PÁGINAS

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

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

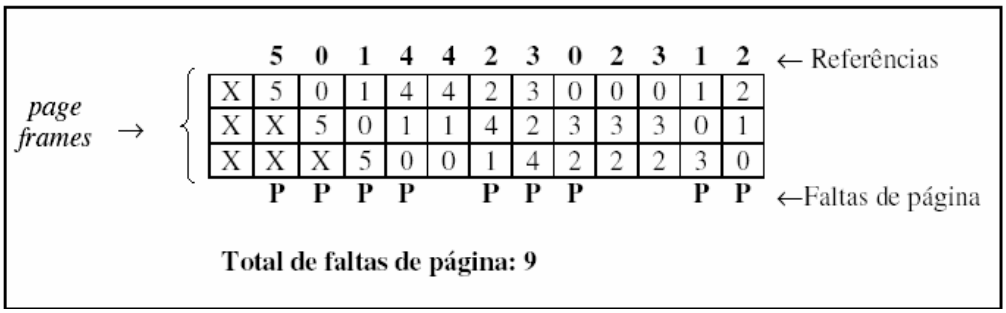
O algoritmo ótimo simplesmente diz que a página com o maior rótulo deve ser removida, adiando-se o máximo possível a próxima falta de página. (A exemplo das pessoas, os computadores também tendem a adiar o quanto possível a ocorrência de eventos desagradáveis).

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

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

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

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



c) **Segunda Chance**:

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

d) **Relógio**:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

		5	**0**	**1**	**4**	**4**	**2**	**3**	**0**	**2**	**3**	**1**	**2**	← Referências
page frames	→	X	5	0	1	4	4	2	3	0	2	3	1	2	
		X	X	5	0	1	1	4	2	3	0	2	3	1	
		X	X	X	5	0	0	1	4	2	3	0	2	3	
			P	**P**	**P**	**P**		**P**	**P**	**P**			**P**		←Faltas de página

Total de faltas de página: 8

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

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

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

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

		0	1	2	3	0	1	4	0	1	2	3	4	← Referências
page frames →	X	0	1	2	3	0	1	4	4	4	2	3	3	
	X	X	0	1	2	3	0	1	1	1	4	2	2	
	X	X	X	0	1	2	3	0	0	0	1	4	4	
		P	P	P	P	P	P	P			P	P		←Faltas de página

Total de faltas de página: 9

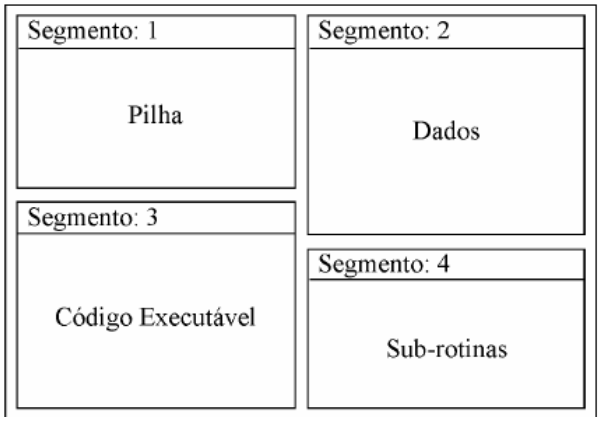
		0	1	2	3	0	1	4	0	1	2	3	4	← Referências
page frames →	X	0	1	2	3	3	3	4	0	1	2	3	4	
	X	X	0	1	2	2	2	3	4	0	1	2	3	
	X	X	X	0	1	1	1	2	3	4	0	1	2	
	X	X	X	X	0	0	0	1	2	3	4	0	1	
		P	P	P	P			P	P	P	P	P	P	←Faltas de página

Total de faltas de página: 10

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

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

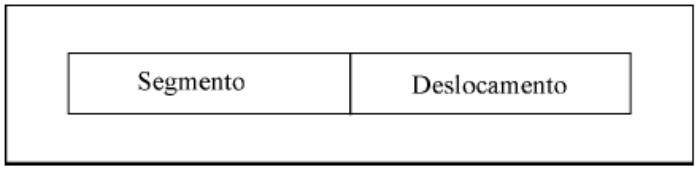
7.5 SEGMENTAÇÃO



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

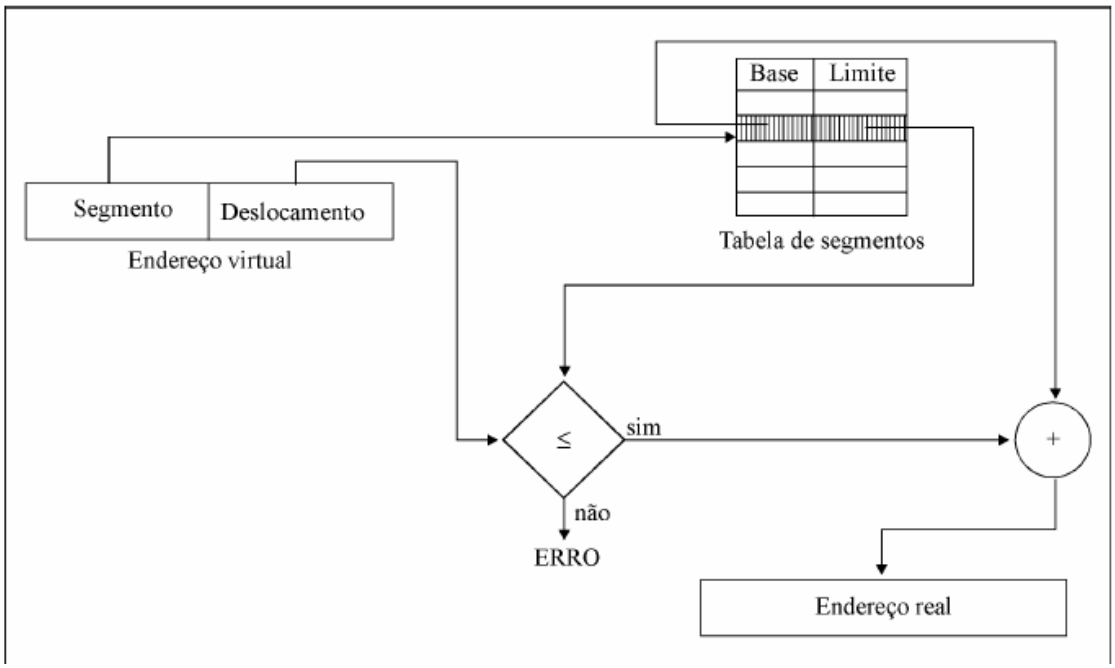
Enquanto na técnica de paginação o programa é dividido em páginas de tamanho fixo, sem qualquer ligação com sua estrutura, na segmentação existe uma relação entre a lógica do programa e sua alocação na memória principal. Normalmente, a definição dos segmentos é realizada pelo compilador, a partir do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

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



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

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



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

7.6 SEGMENTAÇÃO PAGINADA

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

Na visão do programador, sua aplicação continua sendo mapeada em segmentos de tamanhos diferentes, em função das sub-rotinas e estruturas de dados definidas no programa. Por outro lado, o sistema trata cada segmento como um conjunto de páginas do mesmo tamanho, mapeadas por uma tabela de páginas associada ao segmento. Desta forma, um segmento não precisa estar contíguo na memória principal, eliminando o problema da fragmentação externa encontrado na segmentação pura.

7.7 EXERCÍCIOS

103) Quais os benefícios oferecidos pela técnica de memória virtual? Como este conceito permite que um programa e seus dados ultrapassem os limites da memória principal?

104) Explique como um endereço virtual de um processo é traduzido para um endereço real na memória principal?

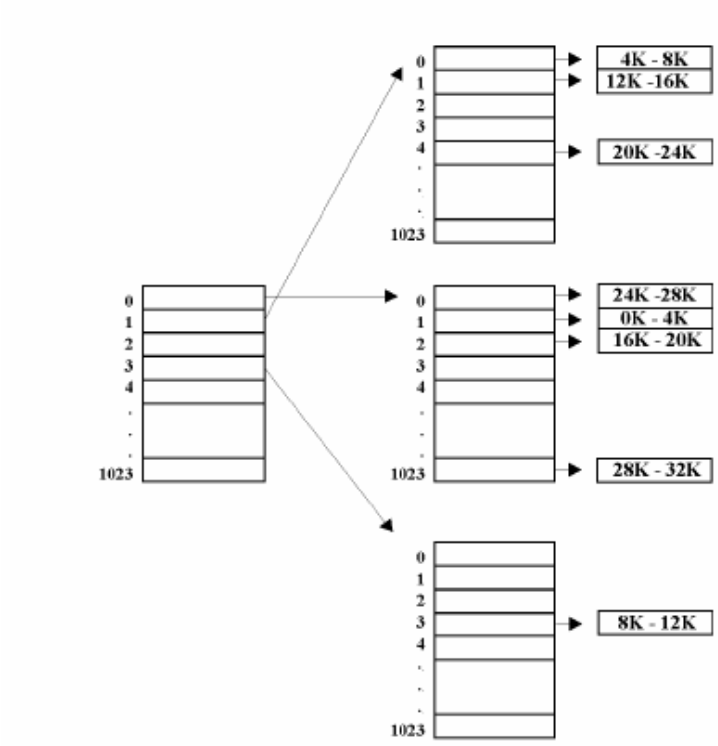
105) Por que o mapeamento deve ser feito em blocos e não sobre células individuais?

106) Qual a principal diferença entre os sistemas que implementam paginação e os que implementam segmentação?

107) Diferencie página virtual de página real.

108) Para que serve o bit de validade nas tabelas de página?

109) Apresente o funcionamento binário da MMU de uma máquina hipotética para encontrar a instrução correspondente de: MOVE REG, 700



110) O que é page fault , quando ocorre e quem controla a sua ocorrência? Como uma elevada taxa de falta de página pode comprometer o Sistema Operacional?

111) Descreva como ocorre a fragmentação interna em sistemas que implementam a paginação.

112) Compare as políticas de busca de páginas apresentadas.

113) Quais as vantagens e desvantagens da política de alocação de páginas variável comparada à alocação fixa?

114) Um sistema com gerência de memória virtual por paginação possui tamanho de página com 512 posições, espaço de endereçamento virtual com 512 páginas endereçadas de 0 a 511 e memória real com 10 páginas numeradas de 0 a 9. o conteúdo atual da memória real contém apenas informações de um único processo e é descrito resumidamente na tabela abaixo:

Endereço Fixo	Conteúdo
1536	Página Virtual 34
2048	Página Virtual 9
3072	Tabela de Páginas
3584	Página Virtual 65
4608	Página Virtual 10

a) Considere que a entrada da tabela de páginas contém, além do endereço do frame, o número da página virtual. Mostre o conteúdo da tabela de páginas desse processo.

b) Mostre o conteúdo da tabela de páginas após a página virtual 49 ser carregada na memória a partir do endereço real 0 e a página virtual 34 ser substituída pela página virtual 12.

c) Como é o formato do endereço virtual deste sistema?

d) Qual endereço físico está associado ao endereço virtual 4613?

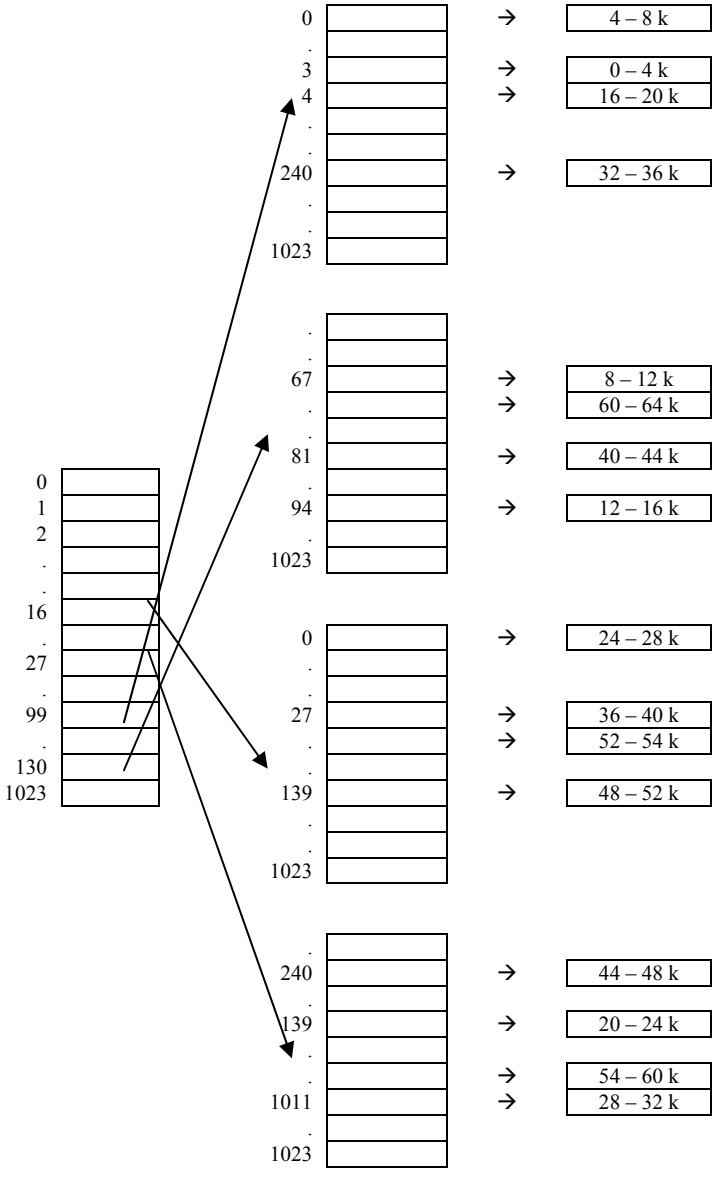
115) Encontre o endereço físico correspondente aos seguintes endereços virtuais:

a) 67.219.712 = 0000010000.0000011011.000100000000

b) 113.819.552 = 0000011011.0010001011.111110100000

c) 545.591.437 = 0010000010.0001010001.000010001101

d) 416.219.160 = 0001100011.0011110000.000000011000



116) Um Sistema Operacional implementa gerência de memória virtual por paginação, com frames de 2KB. A partir da tabela abaixo, que representa o mapeamento de páginas de um processo em um determinado instante de tempo, responda:

Página	Residente	Frame
0	Sim	20
1	Sim	40
2	Sim	100
3	Sim	10
4	Não	50
5	Não	70
6	Sim	1000

a) Qual o endereço físico de uma variável que ocupa o último byte da página 3?
b) Qual o endereço físico de uma variável que ocupa o primeiro byte da página 2?
c) Qual o endereço físico de uma variável que tem deslocamento 10 na página 3?
d) Quais páginas do processo estão na memória?

117) Um Sistema Operacional implementa gerência de memória virtual por paginação. Considere endereços virtuais com 16 bits, referenciados por um mesmo processo durante sua execução e sua tabela de páginas abaixo com no máximo 256 entradas. Estão representadas apenas as páginas presentes na memória real. Indique para cada endereço virtual a seguir a página virtual em que o endereço se encontra, o respectivo deslocamento e se a página se encontra na memória principal nesse momento.

a) $(307)_{10}$
b) $(2049)_{10}$
c) $(2304)_{10}$

Página	Endereço Físico
0	8 K
1	4 K
2	24 K
3	0 K
4	16 K
5	12 K
9	20 K
11	28 K

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

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

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

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

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

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

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

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

124) Explique o funcionamento do algoritmo de substituição de páginas NUR.

125) Explique o funcionamento do algoritmo de substituição de páginas da segunda chance.

126) Informe a principal desvantagem de se utilizar o algoritmo de substituição de páginas da segunda chance.

127) Informe a principal vantagem que o algoritmo do relógio possui sobre o da segunda chance.

128) Considere uma máquina com três molduras de página (page frames), as quais estão inicialmente vazias. Assim sendo, informe quantas faltas de páginas serão geradas com a utilização do algoritmo de substituição de páginas FIFO se a seqüência de referência às páginas for: 0, 1, 2, 3, 0, 1, 4, 1, 2, 3, 1 e 4.

129) Considere uma máquina com quatro molduras de página (page frames), as quais estão inicialmente vazias. Assim sendo, informe quantas faltas de páginas serão geradas com a utilização do algoritmo de substituição de páginas LRU se a seqüência de referência às páginas for: 0, 1, 2, 3, 0, 1, 4, 1, 2, 3, 1, 4 e 5.

130) Considere uma máquina com quatro molduras de página (page frames), as quais estão inicialmente vazias, e uma seqüência de referência às páginas igual a: 0, 1, 2, 3, 0, 1, 4, 1, 2, 3, 1, 4, 5, 4 e 1. Assim sendo, informe, justificando sua resposta, quantas faltas de páginas serão geradas com a utilização de cada um dos algoritmos de substituição de páginas abaixo relacionados:
a) FIFO b) LRU c) Segunda Chance

131) Qual é a principal desvantagem de se utilizar no projeto de um sistema paginado um tamanho de página considerado muito grande? E muito pequeno?

132) O que é a anomalia de Belady?

133) Considere um sistema com memória virtual por paginação com endereço virtual com 24 bits e página com 2048 endereços. Na tabela de páginas a seguir, de um processo em determinado instante, o bit de validade 1 indica página na memória principal e o bit de modificação 1 indica que a página sofreu alteração.

Página	BV	BM	End. do *Frame*
0	1	1	30.720
1	1	0	0
2	1	1	10.240
3	0	1	*
4	0	0	*
5	1	0	6.144

a) Quantos bits possui o campo deslocamento do endereço virtual?
b) Qual o número máximo de entradas que a tabela de páginas pode ter?
c) Qual o endereço físico que ocupa o último endereço da página 2?
d) Qual o endereço físico traduzido do endereço virtual $(00080A)_{16}$?
e) Caso ocorra uma falta de página e uma das páginas do processo deva ser descartada, quais páginas poderiam sofrer *page out*?

134) Considere um sistema de memória virtual que implemente paginação, onde o limite de frames por processo é igual a três. Descreva para os itens abaixo, onde é apresentada uma seqüência de referências a páginas pelo processo, o número total de faltas de páginas para as estratégias de realocação de páginas FIFO e LRU. Indique qual a mais eficaz para cada item:
a) 1 / 2 / 3 / 1 / 4 / 2 / 5 / 3 / 4 / 3
b) 1 / 2 / 3 / 1 / 4 / 1 / 3 / 2 / 3 / 3

135) Em um sistema de memória virtual que implementa paginação, as páginas têm 4 K endereços, a memória principal possui 32 KB e o limite de páginas na memória principal é de 8 páginas. Um programa faz referência a endereços virtuais situados nas páginas 0, 2, 1, 9, 11, 4, 5, 2, 3, 1 nesta ordem. Após essa seqüência de acessos, a tabela de páginas completa desse programa tem a configuração abaixo. As entradas em branco correspondem a páginas ausentes.

Página	End. Físico
0	8 K
1	4 K
2	24 K
3	0 K
4	16 K
5	12 K
6	*
7	*
8	*
9	20 K
10	*
11	28 K
12	*
13	*
14	*
15	*

a) Qual o tamanho (em bits) e o formato do endereço virtual?

b) O processo faz novas referências a endereços virtuais situados nas páginas 5, 15, 12, 8 e 0 nesta ordem. Complete o quadro a seguir, que ilustra o processamento dessa seqüência de acessos utilizando a estratégia de remoção FIFO. Mostre o estado final da tabela de páginas.

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

136) Em um computador, o endereço virtual é de 16 bits e as páginas têm tamanho de 2K. O limite de páginas reais de um processo qualquer é de quatro páginas. Inicialmente, nenhuma página está na memória principal. Um programa faz referência a endereços virtuais situados nas páginas 0, 7, 2, 7, 5, 8, 9, 2 e 4, nesta ordem.

a) Quantos bits do endereço virtual destinam-se ao número da página? E ao deslocamento?

b) Ilustre o comportamento da política de substituição LRU mostrando, a cada referência, quais páginas estão em memória, as faltas de páginas e as páginas escolhidas para descarte.

137) Um sistema trabalha com gerência de memória virtual por paginação. Para todos os processos do sistema, o limite de páginas na memória principal é igual a 10. Considere um processo que esteja executando um programa e em um determinado instante de tempo (T) a sua tabela de páginas possui o conteúdo a seguir. O bit de validade igual a 1 indica página na memória principal e o bit de modificação igual a 1 indica que a página sofreu alteração.

Número da Página	BV	BM	Endereço do Frame (Hexa)
0	1	0	3303A5
1	1	0	AA3200
2	1	0	111111
3	1	1	BFDCCA
4	1	0	765BFC
5	1	0	654546
6	1	1	B6B7B0
7	1	1	999950
8	1	0	888BB8
9	0	0	N/A
10	0	0	N/A

Responda às perguntas abaixo, considerando que os seguintes eventos ocorrerão nos instantes de tempo indicados:
(T + 1): O processo referencia um endereço na página 9 com *page fault*.
(T + 2): O processo referencia um endereço na página 1.
(T + 3): O processo referencia um endereço na página 10 com *page fault*.
(T + 4): O processo referencia um endereço na página 3 com *page*

a) Em quais instantes de tempo ocorrem um *page out*?

b) Em que instantes de tempo o limite de páginas do processo na memória principal é atingido?

c) Caso a política de realocação de páginas utilizada seja o FIFO, no instante (T + 1), qual a página que está há mais tempo na memória principal?

d) Como o sistema identifica que no instante de tempo (T + 2) não há ocorrência de falta de página?

138) Um sistema possui quatro frames. A tabela abaixo apresenta para cada página o momento da carga, o momento do último acesso, o bit de referência e o bit de modificação. Responda:

Frame	Carga	Referência	BR	BM
0	126	279	0	0
1	230	260	1	0
2	120	272	1	1
3	160	280	1	1

a) Qual pagina será substituída utilizando o algoritmo NRU?
b) Qual pagina será substituída utilizando o algoritmo FIFO?
c) Qual pagina será substituída utilizando o algoritmo LRU?

139) Considere um processo com limite de páginas reais igual a quatro e um sistema que implemente a política de substituição de páginas FIFO. Quantos *page faults* ocorrerão considerando que as páginas virtuais são referenciadas na seguinte ordem: 0172327103. Repita o problema utilizando a política LRU e Segunda Chance

140) O que é *trashing*?

-X-

8 Sistema de Arquivos

"Está na minha memória trancado e você mesma guardará a chave."
(William Shakespeare)

8.1 INTRODUÇÃO

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

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

8.2 HIERARQUIA DE DADOS

Informações são armazenadas em computadores segundo uma hierarquia de dados. O nível, mais baixo é composto de bits. Bits são agrupados em **padrões de bits** para representar itens de dados de interesse em sistemas de computador. Há 2^n padrões de bits possíveis para uma série de *n* bits.

O nível seguinte da hierarquia de dados representa padrões de bits de tamanho fixo, como bytes, caracteres e palavras. Quando se refere a armazenamento, um **byte** normalmente são 8 bits. Uma **palavra** representa o número de bits sobre os quais um processador pode operar de uma só vez. Assim, uma palavra são 4 bytes em processadores de 32 bits e 8 bytes em processador de 64 bits.

Caracteres mapeiam bytes (ou grupos de bytes) para símbolos como letras, números, pontuação e novas linhas. Muitos sistemas usam caracteres de 8 bits e, portanto, podem ter 2^8, ou 256 caracteres possíveis em seus **conjuntos de caracteres**. Os três conjuntos de caracteres mais populares em uso hoje são o ASCII (*American Standard Code for Information Interchange*), o EBCDIC (*Extended Binary Coded Decimal Interchange Code*) e o Unicode.

O ASCII armazena caracteres como bytes de 8 bits e, assim, pode ter 256 caracteres possíveis em seu conjunto de caracteres. Devido ao pequeno tamanho de caracteres do ASCII, ele não suporta conjuntos de caracteres internacionais. O EBCDIC é frequentemente usado para representar dados em sistemas de computadores de grande porte (mainframes), particularmente nos sistemas desenvolvidos pela IBM; também armazena caracteres de 8 bits.

Unicode é um padrão reconhecido internacionalmente, bastante usado em aplicações da Internet e multilíngües. Seu objetivo é utilizar um único número para representar cada caractere em todos os idiomas do mundo. O Unicode fornece representações de 8, 16 e 32 bits do seu conjunto de caracteres. Para simplificar a conversão de caracteres ASCII em Unicode, a representação de 8 bits do Unicode, denominada UTF-8 (Unicode Character Set Translation Format-8 bits), corresponde diretamente ao conjunto de caracteres ASCII. Arquivos HTML normalmente são codificados usando UTF-8. UTF-16 e UTF-32 oferecem conjuntos de caracteres maiores, que habilitam aplicações a armazenar informações que contenham caracteres de vários alfabetos como o grego, o cirílico, o chinês e muitos outros. Todavia, exigem arquivos maiores para armazenar o mesmo número de

caracteres em comparação com o UTF-8. Por exemplo, a série de 12 caracteres "Hello Word" requer 12 bytes de armazenamento usando caracteres de 8 bits, 24 bytes usando caracteres de 16 bits e 48 bytes usando caracteres de 32 bits.

Um **campo** é um grupo de caracteres (por exemplo, o nome, o endereço ou o número de telefone de uma pessoa). Um **registro** é um conjunto de campos. Um **arquivo** é um grupo de registros relacionados. O nível mais alto da hierarquia de dados é um sistema de arquivo ou **banco de dados**. Sistemas de arquivos são coleções de arquivos, ou coleções de dados.

O termo **volume** representa uma unidade de armazenamento de dados que pode conter vários arquivos. Um volume físico é limitado a um único dispositivo de armazenamento; um volume lógico – como um volume que poderia ser utilizado em uma máquina virtual – pode estar disperso por muitos dispositivos. Exemplos de volumes são CDs, DVDs, fitas e discos rígidos.

8.3 ARQUIVOS

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

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

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

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

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

Arquivos podem consistir em um ou mais registros. Um registro físico (ou bloco físico) é a unidade de informação realmente lida de ou escrita para um dispositivo de armazenamento. Um registro lógico (ou bloco lógico) é uma coleção de dados tratada como uma unidade pelo software. Quando cada registro físico contém exatamente um registro lógico, diz-se que o arquivo consiste em registros não blocados. Em um arquivo com registros de tamanho fixo, todos os registros têm o mesmo tamanho; o tamanho do bloco é, ordinariamente, um múltiplo inteiro do tamanho do registro. Em um arquivo com registros de tamanhos variáveis, os tamanhos dos registros podem variar até o tamanho do bloco.

8.4 SISTEMAS DE ARQUIVOS

Um sistema de arquivo organiza arquivos e gerencia o acesso aos dados. Eles são responsáveis por:

- Gerenciamento de arquivos – fornecer os mecanismos para que os arquivos sejam armazenados, referidos, compartilhados e fiquem em segurança.
- Gerenciamento de armazenamento auxiliar – alocar espaço para arquivos em dispositivos de armazenamento secundário.
- Mecanismos de integridade do arquivo – garantir que as informações armazenadas em um arquivo não sejam corrompidas. Quando a integridade do arquivo é assegurada, os arquivos contêm somente as informações que devem conter.
- Métodos de acesso – como os dados armazenados podem ser acessados.

8.4.1 DIRETÓRIOS

Considere um sistema de compartilhamento de grande escala que suporte uma grande comunidade de usuários. Cada usuário pode conter várias contas; cada conta pode ter muitos arquivos. Alguns arquivos podem ser pequenos, como mensagens de e-mail; outros podem ser grandes, como listas de um controle de estoque.

É comum que contas de usuários contenham centenas e até milhares de arquivos. Com uma comunidade de vários milhares de usuários, os discos de um sistema poderiam facilmente conter milhões de arquivos. Esses arquivos precisam ser acessados rapidamente para limitar os tempos de resposta.

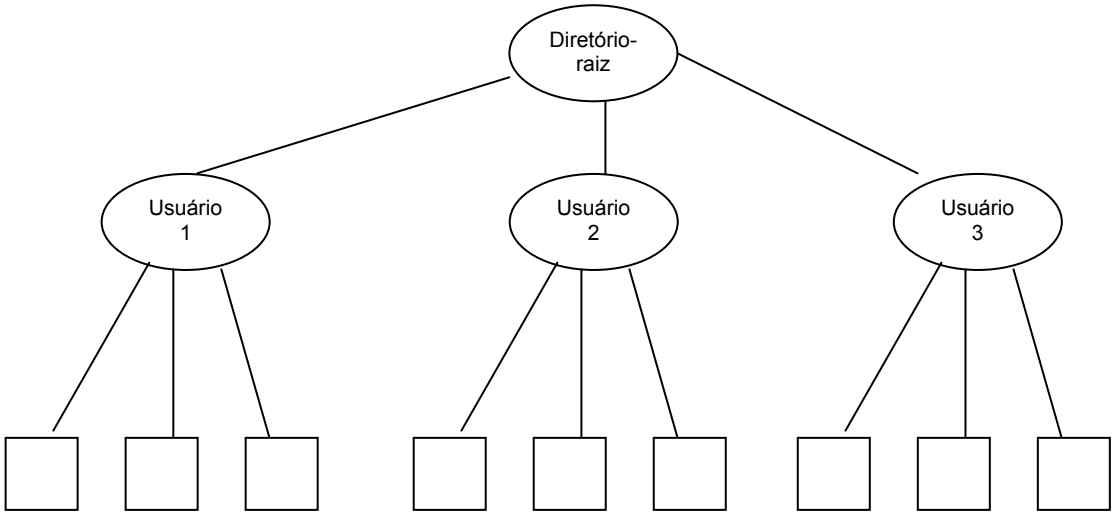
Para organizar e localizar arquivos rapidamente, sistemas de arquivos usam diretórios, que são arquivos que contêm os nomes e as localizações de outros arquivos do sistema de arquivos. Diferentemente de outros arquivos, um **diretório** não armazena dados de usuários, mas campos como nome, localização, tamanho, tipo, horário de acesso, de modificação e de criação do arquivo.

Sistema de Arquivo de nível único

A organização mais simples de sistemas de arquivo é uma estrutura de diretório de nível único (ou plana). Nessa implementação, o sistema de arquivo armazena todos os seus arquivos utilizando um só diretório. Em um sistema de arquivo de nível único, dois arquivos não podem ter o mesmo nome. Porque a maioria dos ambientes contém um grande número de arquivos, muitos dos quais usam o mesmo nome, tais sistemas de arquivos são raramente implementados.

Sistema de Arquivo estruturado hierarquicamente

Um sistema de arquivo mais apropriado para a maioria dos ambientes pode ser organizado da maneira como apresentado na figura a seguir. Uma **raiz** indica em que lugar do dispositivo de armazenamento começa o **diretório-raiz**.



O nome de um arquivo usualmente é formado como o nome de caminho desde o diretório-raiz até o arquivo. Por exemplo, em um sistema de arquivo de dois níveis com os usuários BRUNA, ANDRE e SOFIA, no qual ANDRE tem os arquivos MESADA e ESTUDOS, o **nome do caminho** para o arquivo MESADA poderia ser formado como RAIZ:ANDRE:MESADA. Nesse exemplo, RAIZ indica o diretório-raiz.

Sistemas hierárquicos de arquivo são implementados pela maioria dos sistemas de arquivos de propósito geral, mas o nome do diretório-raiz pode variar entre sistemas de arquivos. O diretório-raiz de um sistema de arquivo do Windows é especificado por uma letra seguida de dois pontos (por exemplo, C:) e sistema de arquivo baseados no Unix usam uma barra inclinada (/). No Windows o exemplo utilizado acima seria C:\ANDRE\MESADA e no Unix seria /ANDRE/MESADA.

Nomes de caminhos relativos

Muitos sistemas de arquivos suportam a noção de um **diretório de trabalho** para simplificar a navegação usando nomes de caminhos. O diretório de trabalho (representado pela entrada de diretório '.' nos sistemas de arquivo do Windows e baseados no Unix) habilita usuários a especificar um nome de caminho que não comece no diretório-raiz. Por exemplo, suponha que o diretório de trabalho corrente tenha sido designado /home/hmd/ em um sistema de arquivos. O **nome de caminho relativo** para /home/hmd/lucilia/cap8 seria ./lucilia/cap8. Essa característica reduz o tamanho do nome de caminho para acessar arquivos. Quando um sistema de arquivos encontra um nome de caminho relativo, forma um **nome de caminho absoluto** (ou seja, o caminho que começa na raiz) concatenando o diretório de trabalho e o caminho relativo. O sistema de arquivo então percorre a estrutura de diretório para localizar o arquivo requisitado.

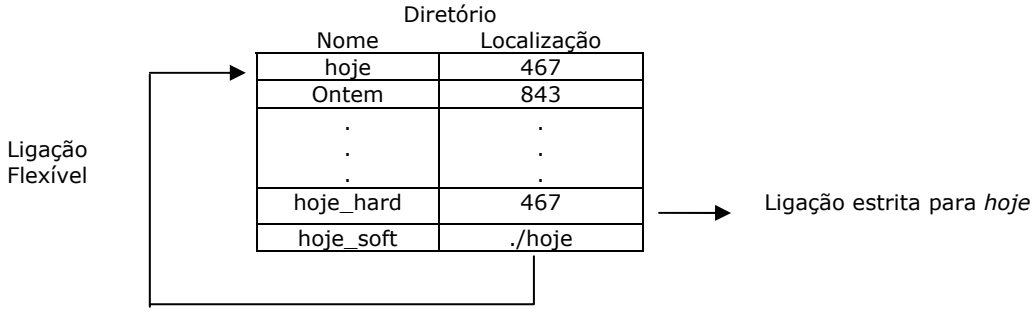
O sistema de arquivo normalmente mantém uma referência ao diretório-pai do diretório de trabalho, ou seja, o diretório que está um nível acima na hierarquia do sistema de arquivo. No Windows e nos sistemas baseados em Unix, '..' é uma referência ao diretório-pai.

Ligação (link) é uma entrada de diretório que se refere a um arquivo de dados ou diretório que normalmente está localizado em um diretório diferente. Usuários comumente empregam ligações para simplificar a navegação do sistema de arquivos e compartilhar arquivos. Por exemplo, suponha que o diretório de trabalho de um usuário seja /home/drc/ e o usuário compartilhe os arquivos de um outro usuário localizado no diretório /home/hmd/lucilia. Criando uma ligação para /home/hmd/lucilia no diretório de trabalho do usuário, esse pode acessar os arquivos compartilhados simplesmente por meio do nome de caminho relativo lucilia/.

Ligação flexível (também chamada de ligação simbólica, atalho ou apelido), é uma entrada de diretório que contém o nome de caminho para um outro arquivo. O sistema de arquivos localiza o alvo da ligação flexível percorrendo a estrutura do diretório por meio do nome de caminho especificado.

Ligação estrita é uma entrada de diretório que especifica a localização do arquivo (normalmente um número de bloco) no dispositivo de armazenamento. O sistema de arquivo localiza os dados do arquivo da ligação estrita acessando diretamente o bloco físico a que ela se refere.

A figura abaixo ilustra a diferença entre ligações flexíveis e ligações estritas. Os arquivos *hoje* e *ontem* estão localizados nos blocos número 467 e 843, respectivamente. A entrada de diretório *hoje_hard* é uma ligação estrita, porque especifica o mesmo número de bloco (467) da entrada do diretório *hoje*. A entrada de diretório *hoje_soft* é uma ligação flexível, porque especifica o nome do caminho para *hoje* (neste caso, *./hoje*)



8.4.2 METADADOS

A maioria dos sistemas de arquivo armazena outros dados além dos dados de usuário e diretório, como a localização dos blocos livres de um dispositivo de armazenamento (para garantir que um novo arquivo não sobrescreva blocos que estão sendo usados) e o horário em que um arquivo foi modificado. Essas informações, denominadas **metada-**

dos, protegem a integridade dos dados e não podem ser modificadas diretamente por usuários.

Antes de um sistema de arquivo poder acessar dados, seu dispositivo de armazenamento precisa ser **formatado**. Formatar é uma operação dependente do sistema, mas normalmente implica inspecionar o dispositivo de armazenamento em busca de quaisquer setores não utilizáveis, apagando quaisquer dados que estejam no dispositivo e criando o diretório-raiz do sistema de arquivo.

Descritores de arquivo

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

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

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

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

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

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

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

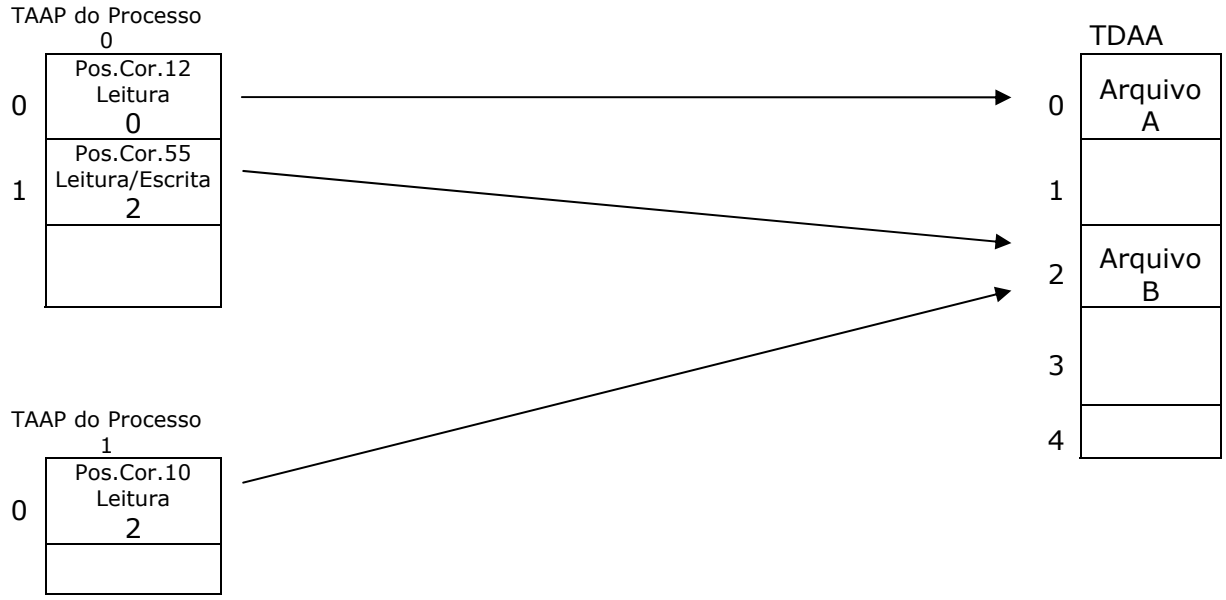
Uma vez que o descritor do arquivo foi copiado para a memória, verifica se o processo em questão tem o direito de abrir o arquivo conforme solicitado. Se não tiver, a entrada na TDAA é liberada e o open retorna um código de erro.

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

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

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

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



8.4.3 MONTAGEM

Usuários muitas vezes requerem acesso a informações que não fazem parte do sistema de arquivo nativo (o sistema de arquivo que está permanentemente montado em um sistema particular e cuja raiz é referida pelo diretório-raiz). Por exemplo, muitos usuários armazenam dados em um segundo disco rígido, em DVD, ou em uma outra estação de trabalho. Por essa razão, sistemas operacionais fornecem a capacidade de **montar** vários sistemas de arquivos. A montagem combina vários sistemas de arquivos em um único **espaço de nomes** – um conjunto de arquivos que pode ser identificado por um único sistema de arquivo. O espaço de nomes unificado permite que usuários acessem dados de diferentes localizações como se todos os arquivos estivessem posicionados dentro do sistema de arquivos nativo.

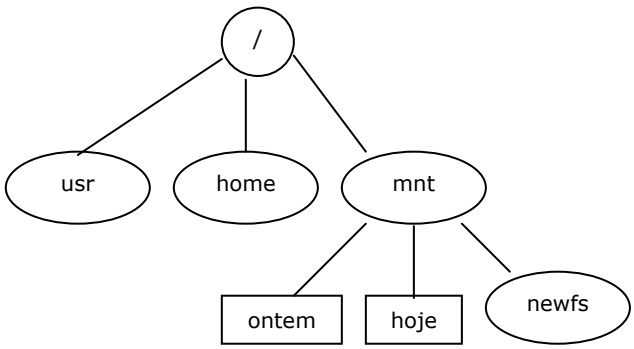
O comando *montar* designa um diretório do sistema de arquivo nativo, denominado **ponto de montagem**, para a raiz do sistema de arquivos montado. Os primeiros sistemas de arquivo windows apresentavam uma estrutura de montagem achatada; cada sistema de arquivo montado era designado por uma letra e localizado no mesmo nível da estrutura do diretório. Por exemplo, normalmente o sistema de arquivo que continha o sistema operacional era montado em C: e o sistema de arquivo seguinte, em D:.

Sistemas de arquivos compatíveis com Unix, apresentam pontos de montagem que podem ser localizados em qualquer lugar do sistema de arquivos. O conteúdo do diretório do sistema de arquivo nativo no ponto de montagem fica temporariamente oculto enquanto um outro sistema de arquivo é montado naquele diretório.

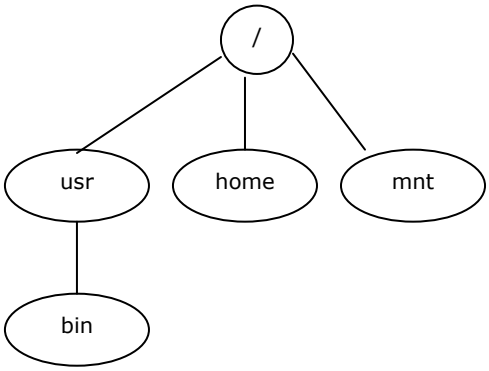
Em sistemas unix, alguns sistemas de arquivo são montados em um dos diretórios localizados em /mnt. Considere a figura a seguir. O comando de montagem coloca a raiz do sistema de arquivo B sobre o diretório /mnt/newfs/ do sistema de arquivo A. Após a operação de montagem, os arquivos do sistema B podem ser acessador por meio de /mnt/newfs/ do sistema de arquivo A. Quando um sistema de arquivo é montado, o conteúdo do diretório no ponto de montagem fica disponível para usuários até que o sistema de arquivo seja desmontado.

A maioria dos sistemas operacionais suporta vários sistemas de arquivos para armazenamento removível, como o *Universal Disk Format* (UDF) para DVDs e o sistema de arquivo ISO 9660 para CDs. Uma vez determinado o tipo de sistema de arquivo do dispositivo, o sistema operacional usa o sistema de arquivo apropriado para acessar o arquivo no dispositivo especificado. O comando desmontar permite que o usuário desconecte sistemas de arquivos montados. Essa chamada atualiza a tabela de montagem e habilita o usuário a acessar quaisquer arquivos que estejam ocultos pelo sistema montado.

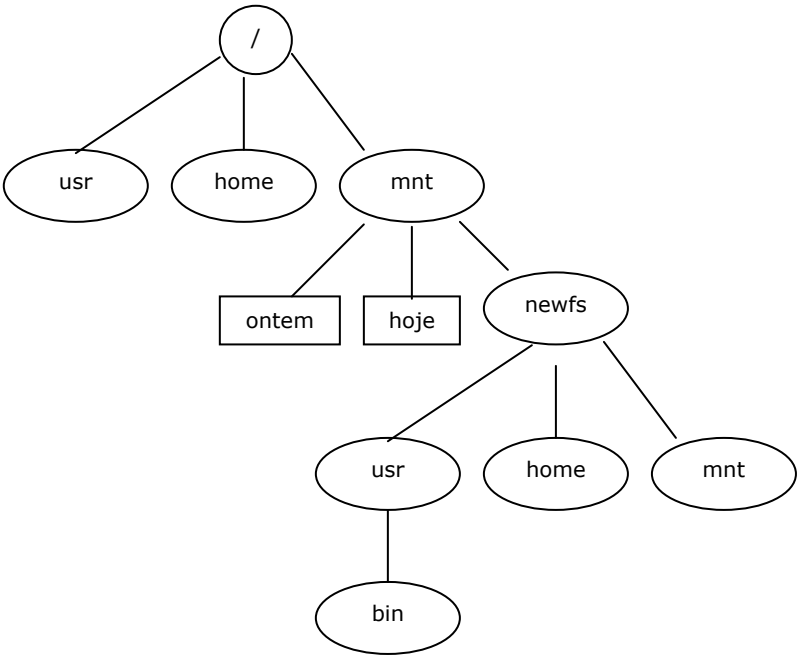
Sistema de Arquivo A



Sistema de Arquivo B



Sistema de arquivo B montado no diretório /mnt/newfs do sistema de arquivo A



8.5 ORGANIZAÇÃO DE ARQUIVO

Organização de arquivo refere-se à maneira como os registros de um arquivo são organizados no armazenamento secundário. Diversos esquemas de organização de arquivo foram implementados:

Seqüencial: Registros são colocados em ordem física. O registro 'seguinte' é aquele que vem depois, fisicamente, do registro anterior. Essa organização é natural para arquivos armazenados em fita magnética, um meio inerentemente seqüencial. Arquivos de disco também podem ser organizados sequencialmente mas, por várias razões, os registros de um arquivo seqüencial não são necessariamente armazenados contiguamente.

Direto: Registros são acessados diretamente (aleatoriamente) por seus endereços físicos em um dispositivo de armazenamento de acesso direto. O usuário da aplicação coloca os registros em qualquer ordem adequada para uma aplicação particular.

Seqüencial indexado: Registros em disco são organizados em seqüência lógica de acordo com uma chave contida em cada registro. O sistema mantém um índice contendo os endereços físicos de certos registros principais. Registros seqüenciais indexados po-

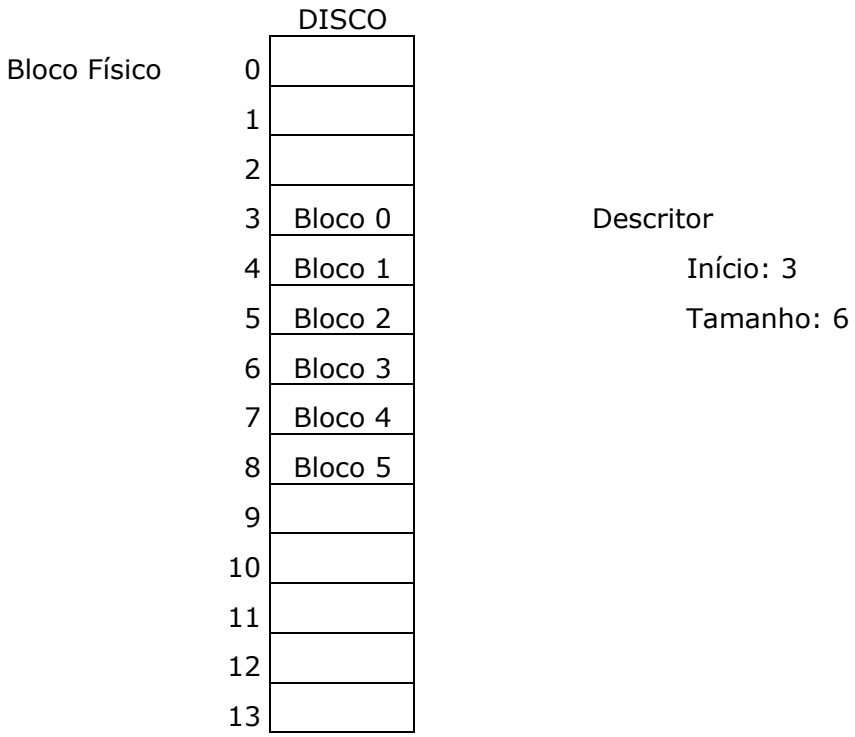
dem ser acessados sequencialmente pela ordem das chaves, ou podem ser acessados diretamente pesquisando-se o índice criado pelo sistema.

Particionado: É, essencialmente, um arquivo de subarquivos seqüenciais. Cada subarquivo seqüencial é denominado membro. O endereço de início de cada **membro** é armazenado no diretório do arquivo. Arquivos particionados têm sido usados para armazenar bibliotecas de programas.

8.6 MÉTODOS DE ALOCAÇÃO

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

8.6.1 ALOCAÇÃO CONTÍGUA



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

Vantagens:

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

Desvantagens:

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

8.6.2 ALOCAÇÃO COM LISTA LIGADA

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

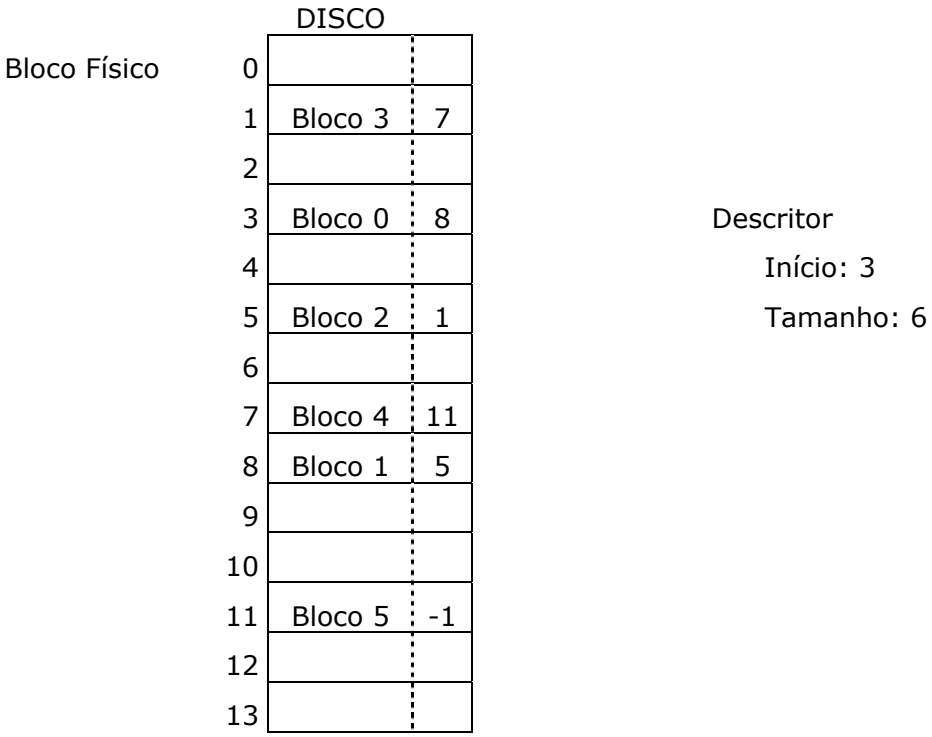
Vantagens:

• Evita-se a perda de espaço em disco ocasionado pela fragmentação;

• Para se acessar todo o arquivo, basta que seja conhecido o endereço de seu primeiro bloco.

Desvantagens:

• Acesso randômico lento e difícil de ser implementado



8.6.3 ALOCAÇÃO COM LISTA LIGADA USANDO UM ÍNDICE

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

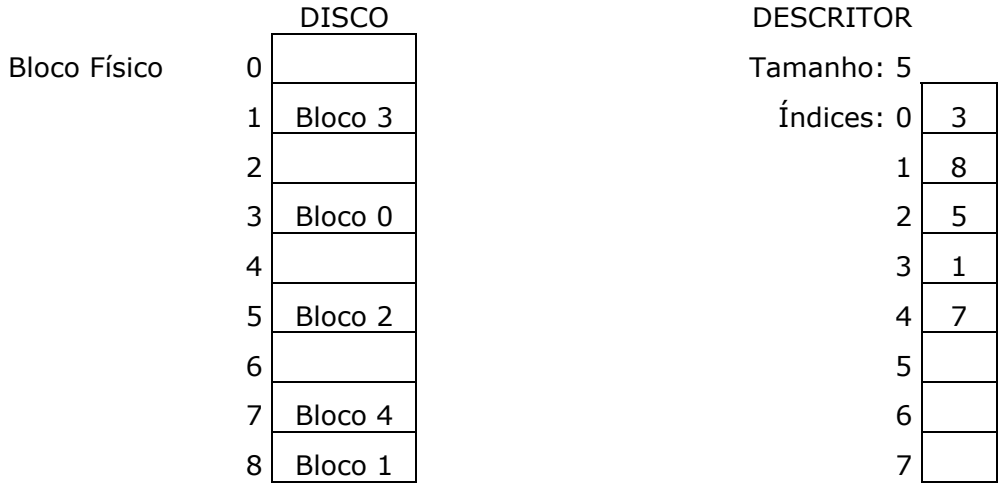
Blocos de índices são denominados **inodes** (*index nodes* – nós de índice) em sistemas operacionais baseados no unix. O inode de um arquivo armazena os atributos do arquivo, como seu proprietário, tamanho, data e horário de criação. Armazena também os endereços de alguns dos blocos de dados do arquivo e ponteiros para blocos de continuação de índices, denominados **blocos indiretos**. Estruturas de inodes suportam até três níveis de blocos indiretos. O primeiro bloco aponta para blocos de dados; esses blocos de dados são unicamente indiretos. O segundo bloco indireto contém ponteiros que se referem somente a outros blocos indiretos; esses blocos indiretos apontam para blocos de dados duplamente indiretos. O terceiro bloco indireto aponta somente para outros blocos indiretos que apontam somente para mais blocos indiretos, que apontam para blocos de dados; esses blocos de dados são triplamente indiretos. O poder dessa estrutura hierárquica é que ela estabelece um limite relativamente baixo ao número máximo de ponteiros que devem ser seguidos para localizar dados de arquivo – ela habilita os inodes a localizar qualquer bloco de dados seguindo no máximo quatro ponteiros (o inode e até três níveis indiretos).

Vantagens:

• Facilidade para se implementar um acesso randômico rápido, pois as entradas a serem seguidas encontram-se na memória

• Para se acessar todo o arquivo basta que a entrada de diretório contenha o endereço de seu primeiro bloco.

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



8.7 GERÊNCIA DE ESPAÇO EM DISCO

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

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

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

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

8.8 CONTROLE DE BLOCOS LIVRES

Existem quatro maneiras mais usuais de se realizar o controle dos blocos não utilizados do disco. Para exemplificar, tomaremos a seguinte memória, considerando blocos preenchidos como ocupados e em branco como livres:

00	**01**	**02**	**03**	**04**	**05**	**06**	**07**
08	**09**	**10**	**11**	**12**	**13**	**14**	**15**
16	**17**	**18**	**19**	**20**	**21**	**22**	**23**
24	**25**	**26**	**27**	**28**	**29**	**30**	**31**
32	**33**	**34**	**35**	**36**	**37**	**38**	**39**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**

8.8.1 MAPA DE BITS

Cada bloco é representado por 1 bit. Se o bloco estiver livre, o bit será 1; se ele estiver alocado, o bit será 0. O mapa de bits para a memória acima seria:

11110000 00001111 11111110 00000000 11111111 00000000

8.8.2 LISTA ENCADEADA

Outra abordagem é encadear os blocos de discos livres, mantendo um ponteiro ao primeiro bloco livre. Tal bloco contém um ponteiro ao próximo bloco livre, e assim por diante. A lista encadeada para o mesmo exemplo acima:

4 → 5, 5 → 6, 6 → 7, 7 → 8, 8 → 9, 9→ 10, 10 → 11, 11 → 23, 23 → 24, 24 → 25, 25 → 26, 26 → 27, 27 → 28, 28 → 29, 29 → 30, 30 → 31, 31 → 40, 40 → 41, 41 → 42...

8.8.3 AGRUPAMENTO

Uma modificação da abordagem de lista livre é armazenar os endereços de n blocos livres no primeiro bloco livre. Os primeiros n-1 desses blocos estão realmente livres. O bloco final contém os endereços de outros n blocos livres, e assim por diante. A lista de agrupamento para a memória acima, considerando um *n* = 4, seria:

4 → 5, 6, 7, 8
8 → 9, 10, 11, 23
23 → 24, 25, 26, 27
27 → 28, 29, 30, 31
31 → 40, 41, 42, 43
43 → 44, 45, 46, 47

8.8.4 CONTADORES

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

Bloco Inicial	Quantidade de Blocos
4	8
23	9
40	8

8.9 CONSISTÊNCIA EM BLOCOS

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

Situações indesejadas possíveis:

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

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

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

Solução: reconstruir a lista de blocos livres.

- Um bloco está presente em mais de um arquivo.

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

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

Solução: remover o bloco da lista de blocos livres.

8.10 EXERCÍCIOS

141) Considerando que a unidade de alocação física é de 2k, o tamanho da memória secundária é de 64k e a primeira unidade de alocação está ocupada com a tabela de arquivos, faça o mapa de bits final depois de todos os passos a seguir. Em seguida, faça a tabela de contadores, de agrupamento e a lista ligada mostrando os espaços livres. Considere n = 4.
   a) Escrever o arquivo A de 11k
   b) Escrever o arquivo B de 6k
   c) Remover o arquivo A
   d) Alocar o arquivo C seqüencial, de 15k.
   e) Escrever o arquivo D com 31K, não seqüencial.

f) Remover o arquivo B

142) Qual a diferença entre fragmentação externa e fragmentação interna? Demonstre graficamente.

143) Relacione a Tabela de Descritores de Arquivos Abertos (TDAA) e a Tabela de Arquivos Abertos por Processo (TAAP), ilustrando e citando exemplos. Simule situações de processos sendo executados e abrindo ou criando arquivos.

144) Cite as vantagens e desvantagens dos métodos de alocação: contígua, encadeada e indexada. Demonstre através de figuras.

145) Existem alguns métodos que objetivam associar blocos de disco a arquivos existentes. No entanto, algumas considerações necessitam ser feitas com relação ao tamanho do bloco. Porque não se deve ter unidades de alocação nem tão grandes nem tão pequenas?

146) Considere que o sistema operacional XYZ utiliza contadores para o controle de seus blocos livres. A lista está reproduzida abaixo. Monte o mapa de bits, agrupamento e a lista encadeada partindo dos contadores observados. Considere n=3.

26	4
10	7
02	3

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

148) Descreva as vantagens e desvantagens de armazenar dados de controle de acesso como parte do bloco de controle de arquivo

-X-

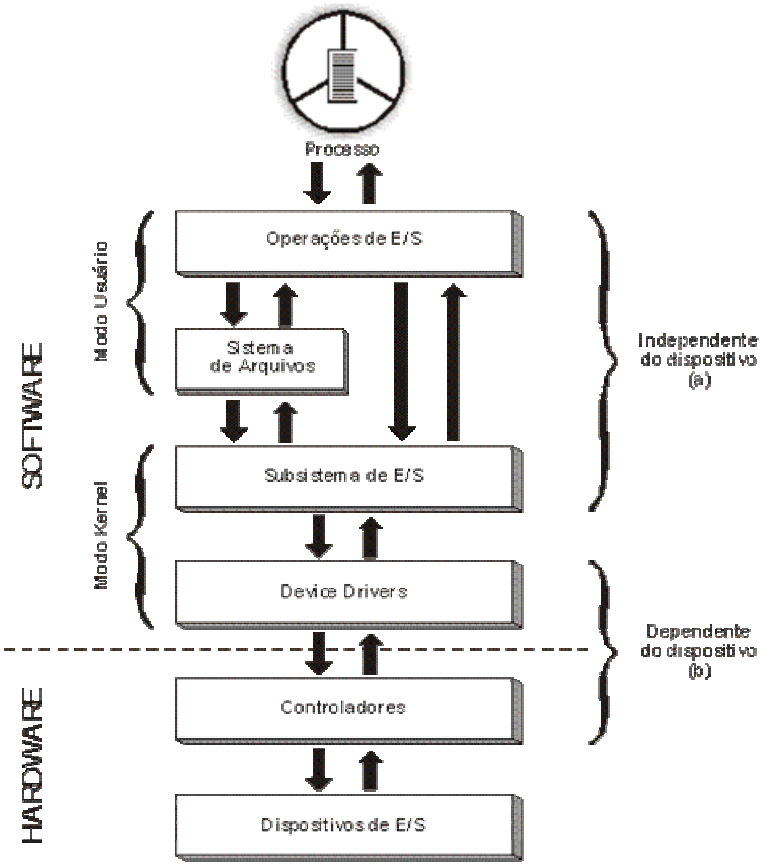
9 Gerência de Dispositivos

"O caminho do dever encontra-se no que está perto,
e o homem o procura no que está distante."
(Mêncio)

9.1 INTRODUÇÃO

A Gerência de Dispositivos de entrada/saída é uma das principais e mais complexas funções de um Sistema Operacional. Sua implementação é estruturada através de camadas. As camadas de mais baixo nível escondem características dos dispositivos das camadas superiores, oferecendo uma interface simples e confiável ao usuário e suas aplicações.

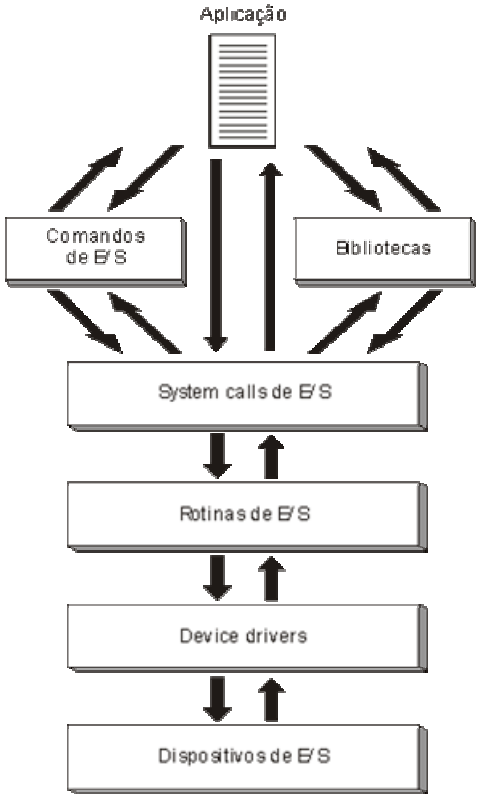
As camadas são divididas em dois grupos, onde o primeiro grupo visualiza os diversos tipos de dispositivos do sistema de um modo único (Fig. a), enquanto o segundo é específico para cada dispositivo (Fig. b). A maior parte das camadas trabalha de forma independente do dispositivo.



9.2 ACESSO AO SUBSISTEMA DE ENTRADA E SAÍDA

O Sistema Operacional deve tornar as operações de E/S o mais simples possível para o usuário e suas aplicações. Para isso, o sistema possui um conjunto de rotinas, denominado *rotinas de entrada/saída*, que faz parte do subsistema de E/S e permite ao usuário realizar operações de E/S sem se preocupar com detalhes do dispositivo que está sendo acessado. Nesse caso, quando um usuário cria um arquivo em disco, não lhe interessa como é a formatação do disco, nem em que trilha ou setor o arquivo será gravado.

As operações de E/S devem ser realizadas através de chamadas de sistemas que chamam as rotinas de E/S do núcleo do Sistema Operacional. Dessa forma, é possível escrever um programa que manipule arquivos, estejam eles em disquetes, discos rígidos ou CD-Rom, sem ter que alterar o código para cada tipo de dispositivo.



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

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

As operações de E/S podem ser classificadas conforme o seu sincronismo. Uma operação é dita síncrona quando o processo que realizou a operação fica aguardando no estado de espera pelo seu término. Assíncrona é quando o processo que realizou a operação não aguarda pelo seu término e continua pronto para ser executado. Neste caso, o sistema deve oferecer algum mecanismo de sinalização que avise ao processo que a operação foi terminada.

9.3 SUBSISTEMA DE ENTRADA E SAÍDA

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

Independência de Dispositivos: Cada dispositivo trabalha com unidades de informação de tamanhos diferentes, como caracteres ou blocos. O subsistema de E/S é responsável por criar uma unidade lógica de transferência do dispositivo e repassá-la para os níveis superiores, sem o conhecimento do conteúdo da informação.

Tratamento de Erros: Normalmente, o tratamento de erros nas operações de E/S é realizado pelas camadas mais próximas ao hardware. Existem, porém, certos erros que podem ser tratados e reportados de maneira uniforme pelo sistema de arquivos, independentemente do dispositivo. Erros como a gravação em dispositivos de entrada, leitura em dispositivos de saída e operações em dispositivos inexistentes podem ser tratados nesse nível.

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

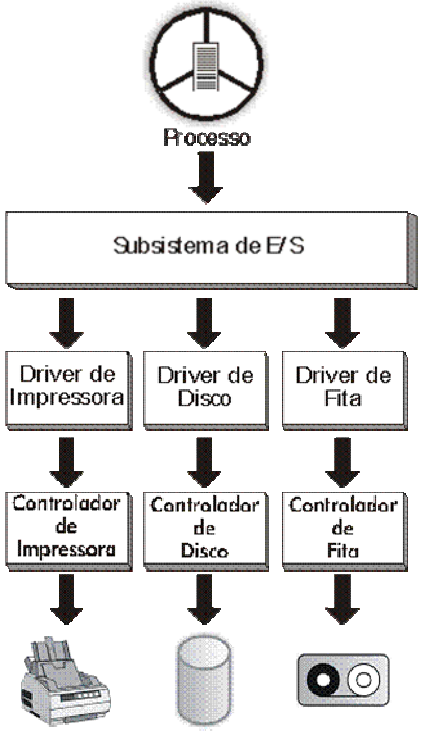
Bufferização: Essa técnica permite reduzir o número de operações de E/S, utilizando uma área de memória intermediária chamada de *buffer*. Por exemplo, quando um dado é lido do disco, o sistema traz para a área de buffer não só o dado solicitado, mas um bloco de dados. Caso haja uma solicitação de leitura de um novo dado que pertença ao blo-

co anteriormente lido, não existe a necessidade de uma nova operação de E/S, melhorando desta forma a eficiência do sistema.

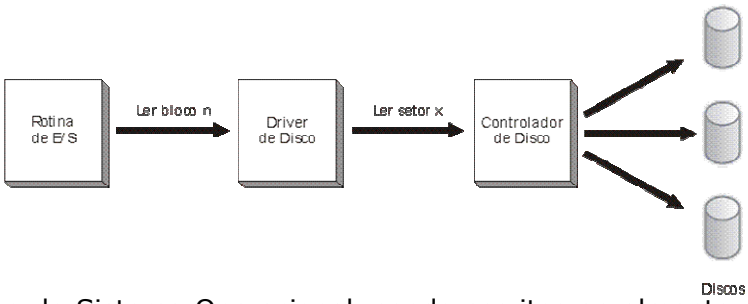
Interface Padronizada: O subsistema de E/S deve oferecer uma interface padronizada que permita a inclusão de novos drivers sem a necessidade de alteração da camada de subsistema de E/S.

9.4 DEVICE DRIVERS

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

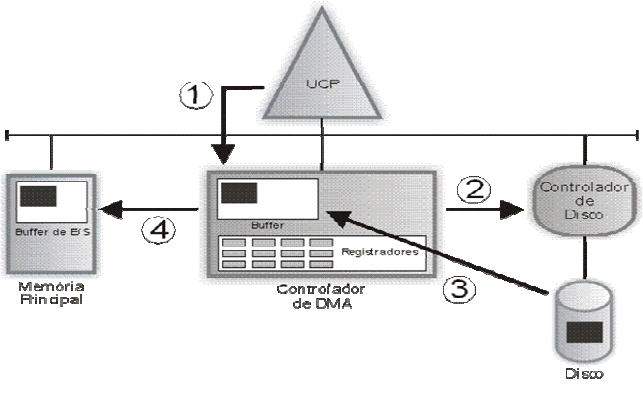


Por exemplo, na leitura síncrona de um dado em disco, o driver recebe a solicitação de um determinado bloco e informa ao controlador o disco, cilindro, trilha e setor que o bloco se localiza, iniciando, dessa forma, a operação. Enquanto se realiza a leitura, o processo que solicitou a operação é colocado no estado de espera até que o controlador avise a UCP do término da operação através de uma interrupção que, por sua vez, ativa novamente o driver. Após verificar a inexistência de erros, o driver transfere as informações para a camada superior. Com os dados disponíveis, o processo pode ser retirado do estado de espera e retornar ao estado de pronto para continuar seu processamento.



Os drivers fazem parte do núcleo do Sistema Operacional, sendo escritos geralmente em *assembly*. Devido ao alto grau de dependência entre os drivers e o restante do núcleo do SO, os fabricantes desenvolvem, para um mesmo dispositivo, diferentes drivers, um para cada Sistema Operacional. Sempre que um novo dispositivo é instalado, o driver deve ser adicionado ao núcleo do sistema. Nos sistemas mais antigos, a inclusão de um novo driver significava a recompilação do *kernel*, uma operação complexa e que exigia a reinicialização do sistema. Atualmente, alguns sistemas permitem a fácil instalação de novos drivers sem a necessidade de reinicialização.

9.5 CONTROLADORES



São componentes de hardware responsáveis por manipular diretamente os dispositivos de E/S. Possuem memória e registradores próprios utilizados na execução de instruções enviadas pelo driver. Em operações de leitura, o controlador deve armazenar em seu *buffer* interno uma seqüência de bits proveniente do dispositivo até formar um bloco. Após verificar a ocorrência de erros, o bloco pode ser transferido para um *buffer* de E/S na memória principal. A transferência do bloco pode

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

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

9.6 DISPOSITIVOS DE ENTRADA E SAÍDA

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

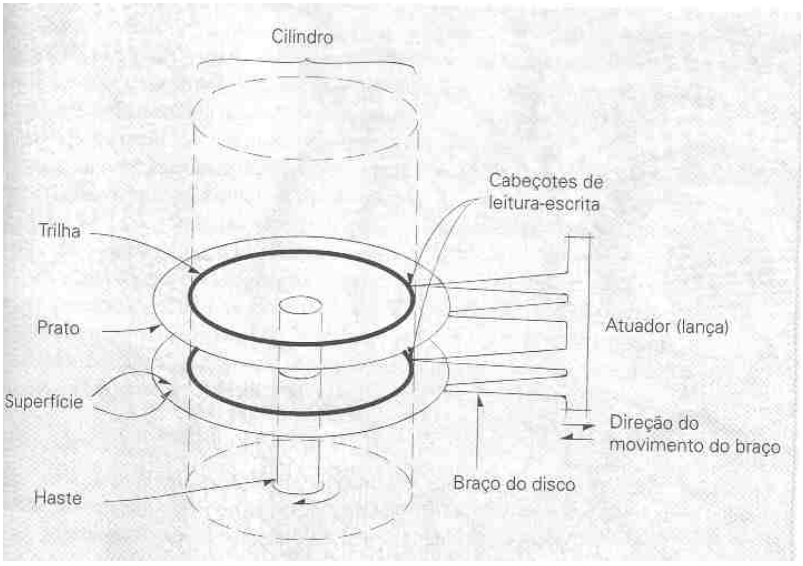
A transferência de dados pode ocorrer através de blocos de informação ou caracteres, por meio dos controladores dos dispositivos. Em função da forma com que os dados são armazenados, os dispositivos de E/S podem ser classificados em duas categorias: ***estruturados*** e ***não-estruturados***. Os dispositivos estruturados (*block devices) caracterizam*-se por armazenar informações em blocos de tamanho fixo, possuindo cada qual um endereço que pode ser lido ou gravado de forma independente dos demais. Discos magnéticos e ópticos são exemplos.

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

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

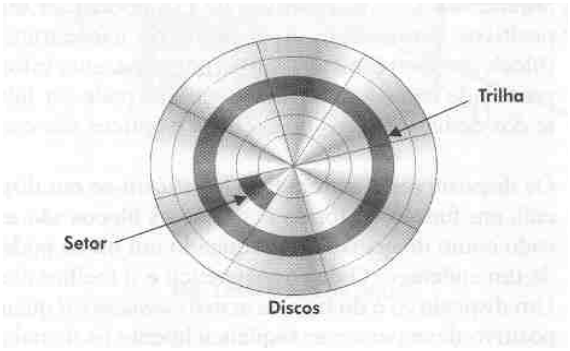
9.7 DISCOS

Diferentemente da memória principal, que fornece velocidade de acesso (quase) uniforme a todos os seus conteúdos, o armazenamento em disco de cabeçote móvel exibe velocidade de acesso variável que depende das posições relativas entre o cabeçote de leitura-escrita e o dado requisitado. A figura a seguir mostra uma vista simplificada de um disco rígido.



Dados são gravados em uma série de discos magnéticos, ou **pratos** (*platters*), conectados a uma haste (*spindle*) que gira em alta velocidade (comumente, milhares de rotações por minuto).

Os dados de cada superfície do disco são acessados por um cabeçote de leitura-escrita que paira a uma pequena distância da superfície do disco (muito menor do que uma partícula de fumaça). Por exemplo, o disco da figura anterior contém dois pratos, cada um com duas superfícies (superior e inferior) e quatro cabeçotes de leitura-escrita, um para cada superfície. Um cabeçote de leitura-escrita pode acessar dados imediatamente abaixo (ou acima) dele. Portanto, antes que os dados possam ser acessados, a porção da superfície do disco na qual eles devem ser lidos (ou escritos) deve girar até ficar imediatamente abaixo (ou acima) do cabeçote de leitura-escrita. O tempo que leva para o dado girar da posição em que está até a extremidade do cabeçote de leitura-escrita é denominado **tempo de latência rotacional**. A latência rotacional média de um disco é simplesmente a metade do tempo que ele leva para completar uma rotação. A maioria dos discos rígidos exibe latência rotacional média da ordem de vários milissegundos.



Enquanto os pratos giram, cada cabeçote de leitura-escrita desenha uma **trilha** circular de dados sobre a superfície de um disco. Cada cabeçote de leitura-escrita é posicionado na extremidade de um **braço de disco**, ligado a um **atuador** (também denominado **lança** ou **unidade de braço móvel**). O braço move-se paralelamente à superfície do disco. Quando o braço do disco movimenta os cabeçotes de leitura-escrita para uma nova posição, um conjunto vertical diferente de trilhas circulares, ou **cilindro**, torna-se acessível. O processo de levar o braço do disco até um novo cilindro é denominado **operação de busca** (*seek*). Para localizar pequenas unidades de dados, o disco divide as trilhas em diversos **setores**, quase sempre de 512 bytes (veja a figura). Portanto, um sistema operacional pode localizar determinado dado especificando o cabeçote (que indica qual superfície do disco deve ser lida), o cilindro (que indica qual trilha a ser lida) e o setor no qual o dado está localizado.

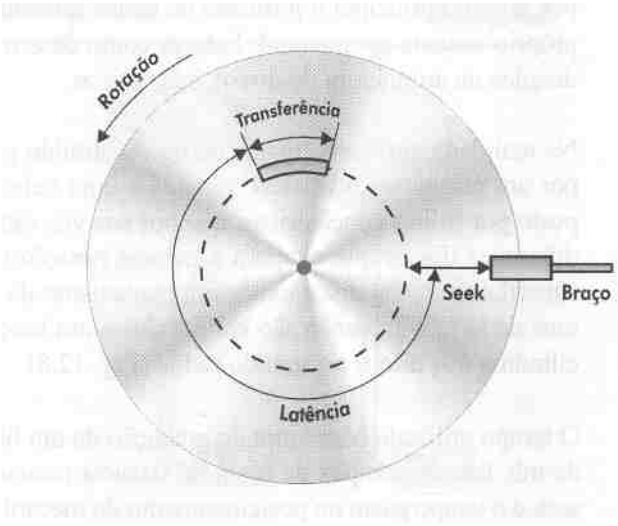
Para realizar um acesso a um disco rígido, é necessário posicionar o cabeçote de leitura e escrita sob um determinado setor e trilha onde o dado será lido ou escrito. Considerando-se a organização de um disco, esse procedimento de posicionamento implica um certo tempo: o tempo de acesso. O tempo de acesso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

$$T_{acesso} = t_{seek} + t_{latência} + t_{transferência}$$

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

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

Tempo de Transferência: tempo de transferência propriamente dito.



9.8 ESCALONAMENTO DO BRAÇO DO DISCO

Muitos processos podem gerar requisições para ler e escrever dados em um disco simultaneamente. Pelo fato desses processos às vezes fazerem requisições mais rapidamente do que podem ser atendidas pelo disco, formam-se filas de espera para reter as requisições de disco. Alguns sistemas de computador mais antigos simplesmente atendiam a essas requisições segundo o algoritmo **primeira a chegar, primeira a ser atendida** (*First-Come-First-Served* -FCFS), pelo qual a requisição que chegou mais cedo é atendida em primeiro lugar. O FCFS é um método justo de alocar serviço, mas, quando a **taxa de requisição** (a carga) fica pesada, pode haver longos tempos de espera.

O FCFS exibe um **padrão de busca aleatório** no qual requisições sucessivas podem causar buscas demoradas partindo dos cilindros mais internos para os mais externos. Para reduzir o tempo gasto na busca de registros, nada mais razoável do que ordenar a fila de requisições de alguma outra maneira que não seja o FCFS. Esse processo, denominado **escalonamento de disco**, pode melhorar significativamente o rendimento.

Escalonamento de disco envolve um exame cuidadoso de requisições pendentes para determinar o modo mais eficiente de atendê-las. Um escalonador de disco examina as relações posicionais entre as requisições que estão à espera e, então, reorganiza a fila de modo que as requisições sejam atendidas com o mínimo de movimento mecânico.

Pelo fato de o FCFS não reordenar requisições, ele é considerado por muitos como o mais simples algoritmo de escalonarnento de disco. Porque os tempos de busca tendem a ser maiores do que os tempos de latência, a maioria dos algoritmos de escalonamento concentra-se em minimizar o tempo total de busca para um conjunto de requisições. A medida que diminui a lacuna entre os tempos de latência rotacional e de busca, minimizar a latência rotacional para um conjunto de requisições também pode melhorar o desempenho geral do sistema, especialmente sob cargas pesadas.

9.8.1 PRIMEIRO A CHEGAR, PRIMEIRO A SER ANTENDIDO (FCFS)

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

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

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

Total Percorrido: 111

9.8.2. MENOR *SEEK* PRIMEIRO (*Shortest Seek First* - SSF)

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

Requisição: 11, 1, 36, 16, 34, 9 e 12

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

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

Total Percorrido: 61

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

9.8.3. ALGORITMO DO ELEVADOR (Scan)

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

Requisição: 11, 1, 36, 16, 34, 9 e 12

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

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

Total Percorrido: 60

9.8.4. ALGORITMO CIRCULAR (C-Scan)

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

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

Requisição: 11, 1, 36, 16, 34, 9 e 12

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

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

Total Percorrido: 68

9.9 EXERCÍCIOS

149) Explique o modelo de camadas aplicado na gerência de dispositivos

150) Qual a principal finalidade das rotinas de E/S?

151) Quais as diferentes formas de um programa chamar rotinas de E/S?

152) Quais as principais funções do subsistema de E/S?

153) Qual a principal função de um driver?

154) Por que o sistema de E/S deve criar uma interface padronizada com os device drivers?

155) Explique o funcionamento da técnica de DMA e sua principal vantagem.

156) Diferencie os dispositivos de E/S estruturados dos não-estruturados.

157) Desenhe um disco com 21 setores e um fator 3 de entrelaçamento. Mostre agora porque há uma minimização no tempo de acesso utilizando esse artifício.

158) As requisições do disco chegam ao driver do disco na seguinte ordem dos cilindros: 10, 22, 20, 2, 40, 6 e 38. Um posicionamento leva 6ms por cilindro movi-

do. Quanto tempo é necessário para os algoritmos FCFS, SSF, Scan e C-Scan, considerando que o braço está inicialmente no cilindro 20? Considere o movimento descendo.

159) Monte um quadro mostrando a seqüência do braço, o deslocamento a cada instante e o deslocamento total percorrido pelo braço utilizando os algoritmos FCFS, SSF, Scan e C-Scan. Considere as seguintes requisições e que o braço estava em repouso. Considere o sentido subindo.

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

-X-

Bibliografia

"Só sei que nada sei" (Sócrates)

Esta apostila foi totalmente baseada nos livros:

DEITEL H. M., DEITEL P. J. e CHOFFNES D.R. *Sistemas Operacionais*. São Paulo, 2005. Prentice Hall.

TANEMBAUM, Andrew S. *Sistemas Operacionais Modernos*. São Paulo, 2003. Prentice Hall.

MAIA, Luiz Paulo e MACHADO, Francis Berenger. *Arquitetura de Sistemas Operacionais*. Rio de Janeiro, 2002. LTC.