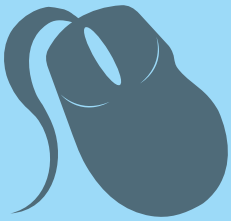


Computação

Sistemas Operacionais

Lorenna Maia Fernandes

EdUECE

Geografia

História

Educação Física

Química

Ciências Biológicas

Artes Plásticas

Computação

Física

Matemática

Pedagogia

Computação

Sistemas Operacionais

Lorenna Maia Fernandes

3ª edição
Fortaleza - Ceará

EdUECE

2015


Geografia


História


Educação Física


Química


Ciências Biológicas


Artes Plásticas


Computação


Física


Matemática


Pedagogia

Editora Filiada à

ASSOCIAÇÃO BRASILEIRA DAS EDITORAS UNIVERSITÁRIAS

Dados Internacionais de Catalogação na Publicação
Sistema de Bibliotecas
Luciana Oliveira – CRB-3 / 304
Bibliotecário

F363s Fernandes, Lorenna Maia.
Sistemas Operacionais / Lorenna Maia Fernandes. – 3. ed. – Fortaleza : EdUECE, 2015.
115 p. : il. ; 20,0cm x 25,5cm. (Computação)

Inclui bibliografia.

ISBN: 978-85-7826-458-1

1. Informática. 2. Sistemas Operacionais Computador. I. Título.

CDD 005.43

Editora da Universidade Estadual do Ceará – EdUECE
Av. Dr. Silas Munguba, 1700 – Campus do Itaperi – Reitoria – Fortaleza – Ceará
CEP: 60714-903 – Fone: (85) 3101-9893
Internet: www.uece.br – E-mail: eduece@uece.br

Secretaria de Apoio às Tecnologias Educacionais
Fone: (85) 3101-9962

Sumário

Apresentação

O texto existente neste material procura sintetizar a abordagem para sistemas operacionais de vários autores renomados nessa área, tais como Tanenbaum, Machado e Maia, Silberschatz, Galvin e Gagne, Oliveira, Toscani e Caríssimi entre outros.

Este material não se propõe a substituir a riqueza presente nos livros publicados pelos autores mencionados, mas sim, servir como apoio às aulas da disciplina de Sistemas Operacionais do Curso de Licenciatura em Computação da UAB/UECE.

Dada a grande quantidade de trechos extraídos de alguns livros, fica impraticável referenciar todos eles. Em compensação, os livros citados na bibliografia, ao final deste material, constituem as fontes principais do texto que a partir daqui se inicia.

Esperamos que este material possa ser útil ao aprendizado dos estudantes sobre os aspectos mais importantes dos Sistemas Operacionais.

A autora

Capítulo 1

Introdução e Histórico

Objetivos

- Definir sistemas operacionais.
- Apresentar um histórico sobre os sistemas operacionais discutindo a evolução das técnicas desenvolvidas.
- Apresentar os conceitos de interrupção, TRAP, bufferização, spooling, multiprogramação e tempo compartilhado.

1. Introdução

O *hardware* (parte física da máquina, composto por teclado, monitor, processador, memória etc) não executa nada se não for controlado/operado por um *software*. Sendo assim, um computador sem *software serve* apenas como peso para papel.

Todos os programas que um computador executa, como por exemplo, processadores de texto, navegadores Web, planilhas eletrônicas, etc, são gerenciados por um programa especial, chamado sistema operacional.

O que é, então, um sistema operacional? Um sistema operacional nada mais é do que um programa de computador que, após o processo de inicialização (*boot*) da máquina, é o primeiro a ser carregado e possui duas tarefas básicas:

- **Gerenciar os recursos de hardware** de modo que sejam utilizados da melhor forma possível, ou seja, de modo que se tire o máximo proveito da máquina fazendo com que seus componentes estejam a maior parte do tempo ocupados com tarefas existentes;
- **Prover funções básicas** para que programas de computador possam ser escritos com maior facilidade, de modo que os programas não precisem conhecer detalhes da máquina para poderem funcionar.

Computadores modernos possuem processador (um ou mais), memória principal e dispositivos de entrada e saída, como teclado, mouse, monitor, interface de rede, entre outros. Escrever programas que utilizem um computador com esta complexidade, de forma eficiente, é muito difícil e trabalhoso.

É exatamente neste ponto que entram as funções básicas providas pelo sistema operacional:

- Abstração das particularidades do *hardware* para os programas aplicativos;
- Fornecimento a esses programas de facilidades para sua operação, tais como:

  a) Funções de gerenciamento de dados como criação, leitura e escrita de arquivos;

  b) Rotinas de acesso aos dispositivos da máquina, como teclado, *mouse*, monitor, impressora, etc.

De acordo com o tipo de tarefa executada por cada programa, eles normalmente são classificados como *software* básico (que inclui o sistema operacional), ou *software* de aplicação, que são voltados a resolver problemas dos usuários.

Podemos visualizar através de um diagrama a integração entre *hardware*, *software* básico, e *softwares* aplicativos, como mostra a Figura 1.

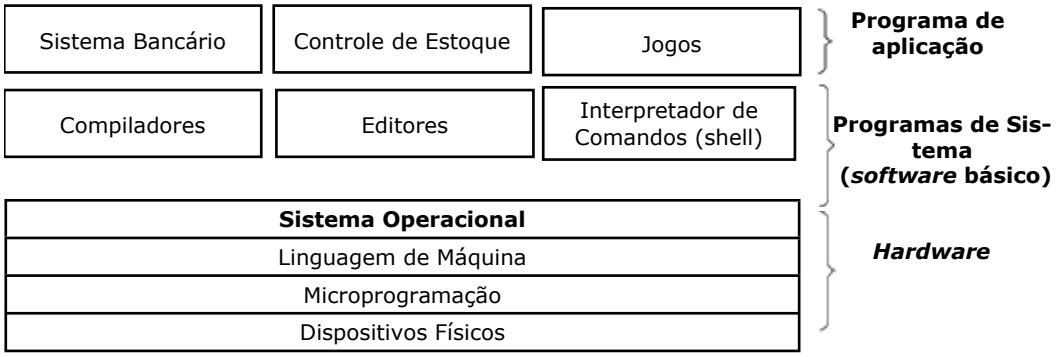


Figura 1 - Integração entre hardware, software básico e software aplicativo.

Fonte: Tanenbaum (2008, p. 22).

Olhando para o diagrama, vemos que o que chamamos de "hardware" é, na verdade, composto de três camadas. Nem todas as máquinas seguem este esquema; algumas podem ter camadas a menos, ou mesmo camadas adicionais, mas basicamente, os computadores seguem o esquema ilustrado na Figura 1.

No nível mais inferior, temos os dispositivos eletrônicos em si, como o processador, os *chips* de memória, controladores de disco, teclado, barramentos, e qualquer dispositivo adicional necessário para o funcionamento do computador.

Um nível acima, temos a camada de microprogramação, que de forma geral, são pequenos passos (chamados de **microoperações**) que

associadas formam uma instrução completa de processador, como ADD, MOV, JMP, etc.

O conjunto de instruções do computador é chamado de **linguagem de máquina**, e apesar de ser uma espécie de linguagem, podemos dizer que faz parte do *hardware* porque os fabricantes a incluem na especificação do processador, para que os programas possam ser escritos. Assim, **as instruções que a máquina entende são consideradas parte integrante do hardware.**

As instruções também incluem, geralmente, operações que permitem ao processador comunicar-se com o mundo externo, como controladores de disco, memória, teclado, etc. Como a complexidade para acesso a dispositivos é muito grande, é tarefa do Sistema Operacional "esconder" estes detalhes dos programas. Assim, o sistema operacional pode, por exemplo, oferecer aos programas uma função do tipo "LEIA UM BLOCO DE UM ARQUIVO", e os detalhes de como fazer isso ficam a cargo do próprio sistema operacional.

Acima do hardware encontram-se os programs de sistema, que são considerados software básico, como por exemplo, o sistema operacional. Dentre eles podemos citar também o **shell**, que consiste do interpretador de comandos do usuário, ou seja, a interface entre o usuário e a máquina. Nos sistemas operacionais mais recentes, freqüentemente o *shell* é uma interface gráfica (ou em inglês GUI – *Graphics User Interface*).

Raramente, numa interface gráfica bem elaborada, o usuário precisa digitar comandos para o computador. Nesse caso, a maneira mais comum de executar programas, copiar e mover arquivos, entre outras atividades, é através do uso do *mouse*. Nos tempos do MS-DOS, no entanto, o teclado era o dispositivo de entrada dominante, por onde o usuário entrava todos os comandos para realizar suas tarefas do dia a dia.

É muito importante observar que software básicos como compiladores (ex: compilador C no Unix), interpretadores de comando (ex: command.com ou explorer.exe no Windows) e *drivers* de dispositivos normalmente são instalados junto com o sistema operacional em um computador, **mas eles não são o sistema operacional**. Eles apenas fornecem facilidades ao usuário através da utilização de funções básicas providas pelo sistema operacional. Portanto, o *shell* que normalmente usamos em um sistema operacional nada mais é do que um programa que utiliza serviços do sistema operacional para permitir que os usuários realizem tarefas como executar programas e trabalhar com arquivos.

A grande diferença entre o sistema operacional e os programas que executam sobre ele, sejam software básicos ou software aplicativos, é que o sistema operacional executa em modo kernel (ou supervisor), enquanto os demais programas executam em modo usuário.

Estes dois modos de operação dos processadores dos computadores diferem no fato de que em modo supervisor, um programa tem acesso a todo o *hardware*, enquanto que em modo usuário, os programas têm acesso somente a determinadas regiões de memória e não podem acessar dispositivos diretamente, precisando fazer uma solicitação ao sistema operacional quando necessitam de alguma tarefa especial.

Isto garante que os programas dos usuários não acabem por invadir áreas de memória do sistema operacional, e acabem por "travar" o sistema. Como também possibilita que programas de diferentes usuários ou de um mesmo usuário executem em uma mesma máquina e um programa não interfira no outro.

2. Histórico

Os primeiros computadores eram máquinas fisicamente muito grandes que funcionavam a partir de um console, o qual consistia de um periférico ou terminal que podia ser usado para controlar a máquina por métodos manuais para, por exemplo, corrigir erros, determinar o estado dos circuitos internos e dos registradores e contadores, e examinar o conteúdo da memória.

O console era o meio de comunicação entre o homem e a máquina, ou seja, era o meio por onde o operador fornecia as entradas e por onde recebia as saídas. O console das primeiras máquinas consistia em chaves pelas quais o operador inseria informações, e por luzes indicativas das saídas, que podiam ser impressas ou perfuradas em uma fita de papel.

Com o passar do tempo, o uso de teclados para entrada de dados se tornou comum, e a saída passou a ser impressa em papel. Posteriormente, o console assumiu a forma de um terminal com teclado e vídeo.

No início da computação, os programadores que quisessem executar um programa, deveriam carregá-lo para a memória manualmente através de chaves no painel de controle, ou através de fita de papel ou cartões perfurados. Em seguida, botões especiais eram apertados para iniciar a execução do programa. Enquanto o programa rodava, o programador/operador podia monitorar a sua execução pelas luzes do console. Se erros eram descobertos, o programa precisava ser interrompido, e o programador podia examinar os conteúdos da memória e registradores, depurando-os diretamente do console. A saída era impressa diretamente, ou perfurada em fita ou cartão para impressão posterior.

As dificuldades eram evidentes. O programador era também o operador do sistema de computação. Como as máquinas nesta época custavam muito

dinheiro, pensou-se em algumas soluções para agilizar a tarefa de programação. Leitoras de cartões, impressoras de linha e fitas magnéticas tornaram-se equipamentos comuns. **Montadores** (*assemblers*), **carregadores** (*loaders*) e **ligadores** (*linkers*) foram projetados. Bibliotecas de funções comuns foram criadas para serem copiadas dentro de um novo programa sem a necessidade de serem reescritas.

Um bom exemplo do uso das bibliotecas de funções são as rotinas que executam operações de entrada e saída (E/S). Cada novo dispositivo tem suas próprias características, necessitando de cuidadosa programação. Subrotinas especiais foram então escritas para cada tipo de dispositivo de E/S. Essas subrotinas são chamadas de **controladores de dispositivos (device drivers)**, e sabem como "conversar" com o dispositivo para o qual foram escritas.

Uma tarefa simples como ler um caractere de um disco pode envolver seqüências complexas de operações específicas do dispositivo e, ao invés de escrever um código de acesso ao dispositivo a cada momento que for necessário realizar essa tarefa, o controlador do dispositivo é simplesmente utilizado a partir de uma biblioteca.

Mais tarde, compiladores para linguagens de alto nível, como FORTRAN e COBOL, surgiram, facilitando muito a tarefa de programação, que antes era feita diretamente na linguagem da máquina. Entretanto a operação do computador para executar um programa em uma linguagem de alto nível era bem mais complexa.

Para executar um programa FORTRAN, por exemplo, o programador deveria primeiramente carregar o compilador FORTRAN para a memória. O compilador normalmente era armazenado em fita magnética e, portanto, a fita correta deveria ser carregada para a unidade leitora de fitas magnéticas. Uma vez que o compilador estivesse pronto, o programa fonte em FORTRAN era lido através de uma leitora de cartões e escrito em outra fita.

O compilador FORTRAN produzia saída em linguagem *assembly* que precisava ser **montada** (*assembled*), isto é, convertida para código de máquina. A saída do montador era **ligada** (*linked*) para suportar rotinas de biblioteca. Finalmente, o código objeto do programa estava pronto para executar e era carregado na memória e depurado diretamente no console, como anteriormente.

Podemos perceber que poderia ser necessário um tempo significativo para a **preparação** da execução de uma tarefa (*job*). Vários passos deveriam ser seguidos, e em caso de erro em qualquer um deles, o processo deveria ser reiniciado após a solução do problema.

2.1. O Monitor residente

O tempo de preparação de um *job* era um problema real. Durante o tempo em que fitas eram montadas ou o programador estava operando o console, a UCP (Unidade Central de Processamento) ficava ociosa. Vale lembrar que no passado poucos computadores estavam disponíveis e eram muito caros (milhões de dólares).

Além disso, os custos operacionais com energia, refrigeração, programadores, tornava ainda mais cara sua manutenção. Por isso, tempo de processamento tinha muito valor, e os proprietários dos computadores os queriam ocupados o máximo do tempo possível. O computador precisava ter uma **alta utilização** para que o investimento fosse compensado.

Uma primeira solução foi contratar um profissional que operasse o computador. O programador não precisava mais operar a máquina, e assim que um *job* terminasse, o operador podia iniciar o próximo; já que o operador tinha mais experiência com a montagem de fitas, o tempo de preparação foi reduzido. O usuário apenas fornecia cartões ou fitas perfuradas contendo o programa e instruções necessárias para a execução de seus programas. Caso erros ocorressem durante a execução do programa, o operador emitia uma listagem dos conteúdos da memória e registradores para que o programador pudesse depurá-lo. Em seguida o próximo *job* era posto em execução, e assim por diante.

Para reduzir ainda mais o tempo de preparação, *jobs* com necessidades similares eram agrupados (*batched*) e executados em grupo pelo computador. Por exemplo, supondo que o operador tivesse recebido um *job* FORTRAN, um COBOL, e outro FORTRAN. Se ele os executasse nessa ordem, ele teria que preparar o *job* FORTRAN (carregar fitas de compilador, etc.), então o COBOL, e novamente o FORTRAN. Se ele executasse os dois *jobs* FORTRAN como um grupo ele prepararia o ambiente FORTRAN apenas uma vez, economizando tempo de preparação.

Esta abordagem marcou uma época, onde o **processamento em batch** (lotes) definiu uma forma de utilização do computador: os usuários preparavam seus programas e dados, entregavam-nos ao operador do computador, que os agrupava segundo suas necessidades e os executava, produzindo as saídas a serem devolvidas aos respectivos programadores.

Mesmo com esse avanço, quando um *job* parava, o operador ainda teria que notar o fato observando o console, determinar porque o programa havia parado (término normal ou anormal), listar conteúdos de memória se necessário e então carregar a leitora de cartões ou de fita de papel com o próximo *job*, e inicializar o computador novamente. Durante a transição entre os *jobs*, novamente a UCP ficava ociosa.

Para resolver este problema, foi desenvolvido um **seqüenciador automático de jobs**, que consistia em um **primeiro sistema operacional rudimentar**. Sua função era controlar a transferência automática de um *job* para outro. Este programa foi implementado sob a forma de um **monitor residente**, sempre presente na memória da máquina para este fim.

Assim que o computador era ligado, o monitor residente era chamado para a memória e executado. Para executar um programa de usuário, o monitor transferia o controle para o programa e, quando o programa terminava, ele retornava o controle para o monitor residente, que passava o controle para o próximo programa a ser executado. Assim, o monitor residente fornecia uma seqüência automática entre programas.

Para que o monitor residente pudesse saber qual programa deveria ser executado e de que forma, cartões de controle foram introduzidos. Esses cartões possuíam instruções de controle para a máquina, semelhantes às instruções que os operadores recebiam dos programadores para execução de seus programas. Assim, além do programa e dos dados para um *job*, cartões especiais de controle eram introduzidos entre os cartões de programa e dados do *job* a executar, como por exemplo:

$JOB - Primeiro cartão, indicando o início de um *job*;

$FTN - Executar o compilador FORTRAN;

$LOAD - Carregar o programa compilado;

$RUN - Executar o programa carregado;

$END - Fim do *job*.

Os cartões de início e fim de *job* eram geralmente utilizados para contabilizar o tempo de uso da máquina, para que seu tempo de processamento pudesse ser cobrado do usuário. Por isso, às vezes incluíam parâmetros indicando o usuário do *job*, nome do *job*, etc.

Para distinguir cartões de controle dos demais cartões era necessário identificá-los com um caractere ou um padrão especial no cartão. Em nosso exemplo, o símbolo do dólar ($) foi utilizado para este fim. A linguagem JCL (*Job Control Language*) da IBM usava duas barras (*//*) nas primeiras duas colunas. A Figura 2 ilustra este cenário.

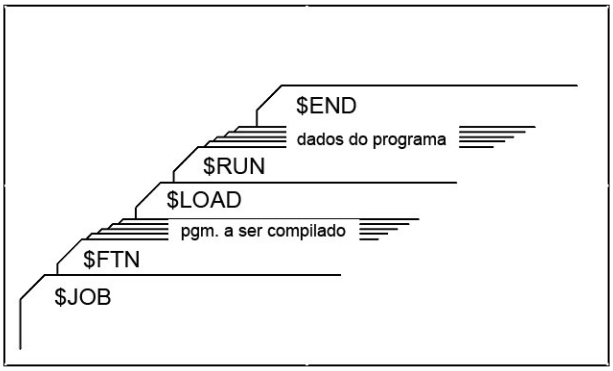


Figura 2 - Conjunto de cartões de um *job*. Fonte: Tanenbaum (2008, p. 28).

Um monitor residente tem várias partes identificáveis. Uma delas é o **interpretador de cartões de controle**, responsável pela leitura e extração das instruções dos cartões no instante da execução. O interpretador de cartões de controle chama um carregador para carregar programas do sistema e programas de aplicação para a memória. Dessa forma, um **carregador (loader)** também é uma parte do monitor residente.

O interpretador de cartões de controle e o carregador precisam realizar operações de Entrada/Saída. O monitor residente tem então um grupo de *drivers* de dispositivos para lidar com os dispositivos do sistema computacional em questão.

Frequentemente, o programa de aplicação e o programa do sistema estão ligados (*linked*) aos mesmos *drivers* de dispositivos, fornecendo continuidade na sua operação, bem como economizando espaço de memória e tempo de programação.

Um esquema de um monitor residente é mostrado na Figura 3.

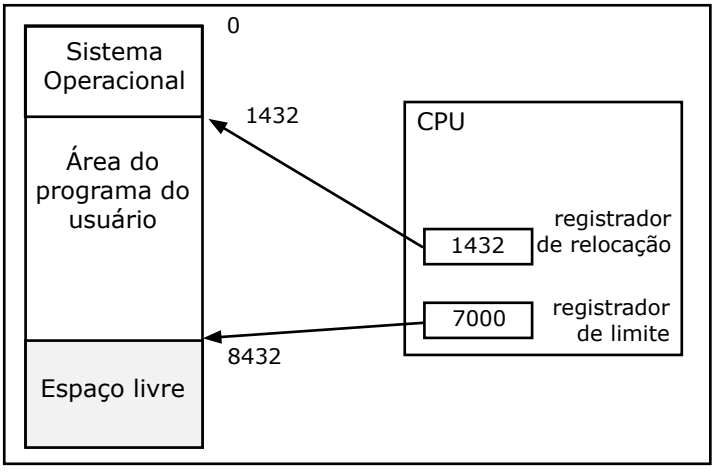


Figura 3 - Modelo de memória de um monitor residente.

Fonte: Oliveira,Carissimi e Toscani (2004, p. 8).

Sistemas *batch* utilizando este método funcionam razoavelmente bem. O monitor residente fornece seqüenciamento automático dos *jobs* conforme a indicação dos cartões de controle. Quando um cartão de controle indica a execução de um programa, o monitor carrega o programa para a memória e transfere o controle para o mesmo. Quando o programa termina, ele retorna o controle para o monitor, que lê o próximo cartão de controle, carrega o programa apropriado e assim por diante. Este ciclo é repetido até que todos os cartões de controle sejam interpretados para o *job*. Então o monitor continua automaticamente com o próximo *job*.

2.2. Operação off-line

O uso de sistemas *batch* com seqüenciamento automático de *jobs* aumentou o desempenho do sistema. Entretanto, ainda assim a UCP ficava freqüentemente ociosa, devido à baixíssima velocidade dos dispositivos mecânicos, como os discos, por exemplo, em relação aos eletrônicos.

Os dispositivos de E/S mais lentos podem significar que a UCP fica freqüentemente esperando por E/S. Por exemplo, um montador ou compilador pode ser capaz de processar 300 ou mais cartões por segundo. Uma leitora de cartões rápida, por outro lado, pode ser capaz de ler apenas 1200 cartões por minuto (20 cartões por segundo). Isto significa que montar um programa com 1200 cartões precisa de apenas 4 segundos de UCP, mas 60 segundos para ser lido pela leitora. Dessa forma, a UCP fica ociosa por 56 dos 60 segundos, ou 93,3% do tempo. A utilização de UCP resultante é de apenas 6,7%. O problema é que, enquanto uma operação de E/S está acontecendo, a UCP está ociosa, esperando que o dispositivo de E/S termine a transferência dos dados e, enquanto a UCP está executando, os dispositivos de E/S estão ociosos.

A maioria dos sistemas no final dos anos 50 e começo dos anos 60 eram sistemas *batch* cujos *jobs* eram lidos de leitoras de cartão e escritos em impressoras de linha ou perfuradoras de cartões. Uma solução simples era substituir as lentas leitoras de cartão (dispositivos de entrada) e impressoras de linha (dispositivos de saída) por unidades de fita magnética. Ao invés de a UCP ler diretamente os cartões, estes cartões eram primeiro copiados para uma fita magnética. Quando a fita estava suficientemente cheia ela era transportada para o computador.

Duas abordagens para esta solução (operação *off-line*) foram usadas:

- Desenvolver dispositivos específicos (leitoras de cartão, impressoras de linha) para provocar saída ou entrada direta de fitas magnéticas;
- Dedicar um pequeno computador para a tarefa de copiar de/para a fita magnética.

O pequeno computador era um "satélite" do computador principal. Processamento satélite foi um dos primeiros casos de múltiplos sistemas de computação trabalhando em conjunto para aumentar o desempenho.

A principal vantagem da operação *off-line* foi a de que o computador principal não estava mais restrito pela velocidade das leitoras de cartão e impressoras de linha, e sim pela velocidade das unidades de fita magnética, que eram mais rápidas. A técnica de usar fitas magnéticas para todo dispositivo de E/S podia ser aplicada a qualquer unidade de equipamento de registro (leitoras e perfuradoras de cartão, leitoras e perfuradoras de fitas de papel, impressoras). Além disso, nenhuma modificação precisava ser feita para adaptar programas de aplicação com operação direta para executar com operação *off-line*.

Considere um programa que executa em um sistema com uma leitora de cartões acoplada. Quando ele precisa de um cartão, ele chama o *driver* de dispositivo da leitora de cartão através do monitor residente. Se ao invés de cartão for fita magnética e estando a operação em modo *off--line*, apenas o *driver* de dispositivo deve ser modificado.

Assim, quando um programa precisa de um cartão de entrada, ele chama a mesma rotina de sistema como antes, entretanto, agora o código para aquela rotina não é o *driver* da leitora de cartões, mas uma chamada para o *driver* da fita magnética. O monitor residente é o responsável pelo gerenciamento dos drivers, enquanto que o programa de aplicação recebe os mesmos dados (a mesma imagem do cartão) em ambos os casos.

A habilidade de executar um programa com dispositivos de E/S diferentes é chamada **independência de dispositivo**. Essa habilidade torna--se possível pela existência de um sistema operacional que identifica qual o dispositivo que um determinado programa usa quando faz uma solicitação de E/S. **Conceito de dispositivo lógico/físico**: Programas são escritos para usar dispositivos de E/S lógicos. Cartões de controle indicam como os dispositivos lógicos deveriam ser mapeados em dispositivos físicos.

O ganho real da operação *off-line* vem da possibilidade de usar múltiplos sistemas leitora-para-fita e fita-para-impressora para uma mesma UCP. Se a UCP pode processar com o dobro da velocidade da leitora, então duas leitoras trabalhando simultaneamente podem produzir fita suficiente para manter a UCP ocupada. Por outro lado, há um atraso maior na execução de um *job* em particular. O job deve ser lido antes para a fita e existe o atraso até que uma quantidade suficiente de *jobs* seja transcrita para a fita até preenchê-la. A fita deve ser então rebobinada, descarregada, manualmente carregada para a UCP e montada em um *drive* de fita livre. Além disso, *jobs* similares podem ser agrupados em uma fita antes de serem levados para o computador, fazendo

com que às vezes um *job* tenha que "esperar" seu agrupamento com outros *jobs* similares em uma fita até que possa ser levado para a UCP.

2.3. Bufferização

Processamento *off-line* permite a sobreposição de operações de UCP e E/S através da execução dessas duas ações em máquinas independentes. Se, no entanto, desejarmos atingir tal sobreposição em uma única máquina, comandos de escalonamento devem ser executados para permitir uma separação similar de execução. Também, uma arquitetura adequada deve ser desenvolvida para permitir *bufferização*.

Na técnica de bufferização um espaço de armazenamento temporário é alocado, de forma que dados possam ser lidos a partir dele ou escritos para ele. Os dados lidos do dispositivo de entrada podem ser armazenados em um *buffer* a partir do qual serão lidos para processamento pela UCP. Esse método possibilita a sobreposição de tarefas de E/S de um *job* com a sua própria execução.

A idéia é muito simples. Os dados iniciais são lidos do dispositivo de entrada e armazenados em um *buffer*. Em seguida, se a UCP estiver pronta para iniciar o processamento desses dados, ela os lê do *buffer* e o dispositivo de entrada é instruído para iniciar a próxima entrada imediatamente, enquanto a UCP executa a entrada atual. Dessa forma, a UCP e o dispositivo de entrada de dados ficam ambos ocupados. Com sorte, no instante em que a UCP estiver pronta para processar o próximo item de dado (registro), o dispositivo de entrada terá terminado de lê-lo. A UCP pode então começar imediatamente o processamento dos novos dados lidos, enquanto o dispositivo de entrada começa a ler os dados seguintes. De forma semelhante isto pode ser feito para a saída. Nesse caso, a UCP cria dados que são colocados em um *buffer* até que o dispositivo de saída possa recebê-los.

Na prática, é raro UCP e dispositivos de E/S estarem ocupados o tempo todo, já que geralmente um dos dois termina de executar primeiro. Se a UCP termina primeiro, ela deve esperar até que o próximo registro seja lido para a memória. É importante observar que a UCP não fica o tempo todo ociosa, no máximo fica o tempo que ficaria se não estivesse sendo utilizada bufferização.

Por outro lado, se o dispositivo de entrada de dados termina primeiro, ele pode tanto esperar quanto continuar com a leitura do próximo registro. Neste caso, ele só deverá parar quando os *buffers* estiverem cheios. Para que o dispositivo de entrada continue sempre trabalhando, normalmente os *buffers* costumam ter tamanho suficiente para mantê-lo sempre ocupado.

A bufferização é geralmente uma função do sistema operacional. O monitor residente ou os *drivers* de dispositivo incluem *buffers* para cada dispositivo de E/S. Assim, chamadas ao *driver* de dispositivo pelos programas de aplicação normalmente causam apenas uma transferência do *buffer* para o sistema.

Apesar da bufferização ser de alguma ajuda, ela raramente é suficiente para manter a UCP sempre ocupada, já que os dispositivos de E/S costumam ser muito lentos em relação à UCP.

2.4. Spooling

Com o passar do tempo, dispositivos baseados em discos tornaram-se comuns e facilitaram muito a operação *off-line* dos sistemas. A vantagem é que em um disco era possível escrever e ler a qualquer momento, enquanto que uma fita precisava ser escrita até o fim para então ser rebobinada e lida.

Em um sistema de disco, cartões são diretamente lidos da leitora de cartões para o disco. Quando um *job* é executado, o sistema operacional atende as suas solicitações de entrada de dados através da leitura do disco. Da mesma forma, quando um *job* pede a impressão de uma linha para a impressora, esta é copiada em um *buffer* do sistema que é escrito para o disco. Quando a impressora fica disponível, a saída é realmente impressa.

Esta forma de processamento é chamada de **spooling (spool = Simultaneous Peripheral Operation On-Line)**. *Spooling* utiliza um disco como um *buffer* muito grande para ler tanto quanto possa dos dispositivos de entrada e para armazenar arquivos de saída até que os dispositivos de saída estejam aptos para recebê-los.

Esta técnica de spooling é também muito utilizada para comunicação com dispositivos remotos. A UCP pode, por exemplo, enviar os dados através dos canais de comunicação para uma impressora remota ou aceitar um *job* completo de entrada de uma leitora de cartões remota. O processamento remoto é feito em sua própria velocidade sem a intervenção da UCP, que apenas precisa ser notificada quando o processamento termina, para que possa passar para o próximo conjunto de dados do *spool*.

A diferença entre **bufferização** e **spooling** é que enquanto a bufferização sobrepõe o processamento de um job com suas próprias funções E/S, o spooling sobrepõe as E/S de um job com o processamento de outros jobs. Assim, a técnica spooling é mais vantajosa do que a bufferização. O único efeito colateral é a necessidade de algum espaço em disco para o spool, além de algumas tabelas em memória.

Outra vantagem do *spooling* é a estruturação dos dados através da lista de *jobs*. Dessa forma, os vários *jobs* armazenados no disco podem ser processados em qualquer ordem que o sistema operacional decidir, buscando sempre o aumento de utilização da UCP. Quando *jobs* são lidos diretamente de cartões ou fita magnética, não é possível executar os *jobs* fora de ordem.

2.5. Multiprogramação

O aspecto mais importante do escalonamento de *jobs* é a habilidade de **multiprogramação**. As técnicas de operação *off-line*, bufferização e *spooling* têm suas limitações em relação à realização de operações de E/S em paralelo com operações da UCP. Em geral, um único usuário não pode manter tanto UCP quanto dispositivos de E/S ocupados o tempo todo. A multiprogramação, no entanto, aumenta a utilização da UCP organizando os vários *jobs* de forma que a UCP sempre tenha algo para processar.

A multiprogramação funciona da seguinte maneira: inicialmente o sistema operacional escolhe um dos *jobs* da lista de *jobs* a serem executados e começa a executá-lo. Eventualmente, o *job* deve esperar por alguma tarefa, como a montagem de uma fita, um comando digitado pelo teclado, ou mesmo o término de uma operação de E/S.

Em um sistema **monoprogramado**, em qualquer um desses casos a UCP permaneceria ociosa. Por outro lado, em um sistema **multiprogramado**, o sistema operacional simplesmente seleciona outro job da lista e passa a executá-lo. Quando este novo *job* precisa esperar, a UCP troca para outro *job* e assim por diante. Em um dado momento, o primeiro *job* não precisa mais esperar e ganha a UCP novamente. Assim, sempre que existirem *jobs* a serem processados, a UCP não ficará ociosa.

Sistemas operacionais multiprogramados são bastante sofisticados. Para que vários *jobs* estejam prontos para executar, é necessário que todos estejam presentes na memória principal da máquina simultaneamente. Isto requer gerenciamento de memória para os vários *jobs*. Além disso, se vários *jobs* estão prontos para executar ao mesmo tempo, o sistema deve escolher qual deles deve ser executado primeiro. **A política de decisão de qual *job* será executado é chamada de escalonamento de UCP.** Por fim, o sistema operacional deve garantir que vários *jobs* executando concorrentemente não afetem uns aos outros, nem o próprio sistema operacional.

2.6. Tempo compartilhado

O conceito de sistemas de **tempo compartilhado**, também chamados de **multitarefa**, é uma extensão lógica da multiprogramação. Neste ambiente,

múltiplos *jobs* são executados simultaneamente, sendo que a UCP atende cada *job* por um pequeno tempo, passando em seguida a execução para o próximo job na fila. Os tempos dedicados para cada *job* são pequenos o suficiente para que os usuários consigam interagir com cada programa sem que percebam que existem outros programas executando. Quando muitos programas estão sendo executados, a impressão que o usuário tem é de que o computador está lento. Isso acontece porque a UCP tem mais *jobs* para atender e, portanto, aumenta o tempo entre os sucessivos atendimentos para um determinado *job*.

É fácil de entender como funcionam sistemas de tempo compartilhado interativos quando comparados com sistemas *batch*. Nos sistemas batch, um fluxo de *jobs* separados é lido (de uma leitora de cartões, por exemplo), incluindo seus cartões de controle que predefinem o que faz o *job*. Quando o *job* termina, seu resultado normalmente é impresso, e o próximo *job* é posto em execução.

A principal característica (e desvantagem) deste sistema é a **falta de interação** entre o usuário e o programa em execução no *job*. O usuário precisa entregar ao operador o programa que ele deseja executar, incluindo seus dados de entrada. Algum tempo depois (podendo demorar minutos, horas ou mesmo dias), a saída do *job* é retornada. Este tempo entre a submissão do *job* e seu término, chamado de tempo de *turnaround*, vai depender da quantidade de processamento necessária, tempo de preparação necessário, e da quantidade de *jobs* que estavam na fila antes dele ser submetido ao processamento.

Do ponto de vista do programador ou do usuário, existem algumas dificuldades com o sistema *batch*. O usuário não pode interagir com o *job* que está executando, e deve indicar através dos cartões de controle o tratamento de todos os resultados possíveis. Em um *job* de múltiplos passos, determinados passos podem depender do resultado dos anteriores. A execução de um programa, por exemplo, pode depender do sucesso da compilação. Pode ser difícil definir completamente o que fazer em todos os casos.

Outra dificuldade em um sistema *batch* é que programas devem ser depurados estaticamente, a partir de uma listagem. Um programador não pode modificar um programa quando ele está sendo executado para estudar o seu comportamento, como hoje é possível na maioria dos ambientes de programação.

Um sistema de computação **interativo** (chamado de *hands-on*), por outro lado, fornece comunicação *on-line* entre o usuário e o sistema. O usuário dá instruções ao sistema operacional ou a um programa diretamente, e recebe uma resposta imediata. Quando o sistema operacional termina a execução de um comando, ele passa a aceitar outros comandos do usuário através do teclado, e não mais de uma leitora de cartões. Assim, o usuário fornece o comando,

espera pela resposta e decide o próximo comando a ser executado, com base no resultado do comando anterior. O usuário pode fazer experimentos com facilidade e pode ver resultados imediatamente. Usualmente, um teclado é usado para a entrada de dados e uma impressora ou monitor de vídeo para a saída de dados. Este tipo de **terminal**, no entanto, só apareceu algum tempo depois, com o barateamento de componentes eletrônicos neles utilizados.

Sistemas *batch* são bastante apropriados para executar *jobs* grandes que precisam de pouca interação. O usuário pode submeter *jobs* e retornar mais tarde para buscar os resultados; não é necessário esperar seu processamento. Por outro lado, *jobs* interativos costumam ser compostos por várias ações pequenas, onde os resultados de cada ação podem ser imprevisíveis. O usuário submete o comando e espera pelos resultados. O **tempo de resposta** deve ser pequeno — da ordem de segundos no máximo. Um sistema interativo é usado quando é necessário um tempo de resposta pequeno.

Conforme já vimos, no início dos tempos da computação, apesar de primitivos, os sistemas eram interativos. Um novo processamento só começava após o operador analisar os resultados do *job* anterior e decidir que ação tomar. Para aumentar o uso de UCP, sistemas *batch* foram introduzidos, o que realmente fez com que os computadores ficassem menos tempo ociosos. Entretanto, não havia interatividade alguma do usuário ou programador com o sistema.

Sistemas de tempo compartilhado foram desenvolvidos para permitir o uso interativo de um sistema de computação a custos razoáveis. Um **sistema operacional de tempo compartilhado** *(time-sharing*) usa **escalonamento de UCP** e **multiprogramação** para fornecer a cada programa de usuário uma pequena porção de tempo do processador.

Um sistema operacional de tempo compartilhado permite que muitos usuários **compartilhem** o computador simultaneamente. Já que cada ação ou comando em um sistema de tempo compartilhado tende a ser pequeno, apenas uma pequena quantidade de tempo de UCP é necessária para cada usuário. Conforme o sistema troca de um usuário para outro, cada usuário tem a impressão de ter seu próprio computador, enquanto na realidade um computador está sendo compartilhado entre muitos usuários.

A idéia de tempo compartilhado foi demonstrada no início de 1960, mas já que sistemas de tempo compartilhado são mais difíceis e custosos para construir (devido aos numerosos dispositivos de E/S necessários), eles somente tornaram-se comuns no início dos anos 70. Conforme a popularidade destes sistemas cresceu, pesquisadores tentaram combinar os recursos de sistemas *batch* e de tempo compartilhado em um único sistema operacional. Muitos sistemas que foram inicialmente projetados como sistemas *batch*

foram modificados para criar um subsistema de tempo compartilhado. Por exemplo, o sistema *batch* OS/360 da IBM foi modificado para suportar a opção de tempo compartilhado (*Time Sharing Option* - TSO). Ao mesmo tempo, à maioria dos sistemas de tempo compartilhado foi adicionado um subsistema *batch*. Hoje em dia, a maioria dos sistemas fornece ambos os processamentos *batch* e de tempo compartilhado, embora seu projeto básico e uso sejam preponderantes de um ou de outro tipo.

Sistemas operacionais de tempo compartilhado são sofisticados. Eles fornecem um mecanismo para execução concorrente. Como na multiprogramação, vários *jobs* devem ser mantidos simultaneamente na memória, o que requer alguma forma de gerenciamento e proteção de memória, além de escalonamento de UCP. Como a memória tem tamanho limitado, em dadas situações alguns *jobs* terão que ser retirados da memória e gravados temporariamente em disco, para que outros programas possam ser lidos do disco e postos em execução na memória. Quando aquele *job* novamente precisar de continuação em sua execução, ele será trazido de volta para a memória.

Os primeiros sistemas operacionais para microcomputadores eram muito simples, pois o poder computacional dos primeiros "micros" era suficiente para atender somente um programa de um único usuário por vez. Com o passar dos anos, os microcomputadores ganharam poder de processamento equivalente a computadores que ocupavam salas inteiras no passado. Para aproveitar este potencial, os microcomputadores ganharam sistemas operacionais multitarefa, permitindo ao usuário executar mais de uma aplicação por vez, além de permitir situações desejáveis como imprimir um documento enquanto utilizava o editor de textos.

Hoje, multiprogramação e sistema compartilhado são os temas centrais dos sistemas operacionais modernos. Os sistemas operacionais mais recentes para microcomputadores suportam múltiplos usuários e múltiplos programas executando concorrentemente em tempo compartilhado.

3. Os conceitos de interrupção e trap

Pode-se dizer que interrupções e *traps* são as forças que movimentam e dirigem os sistemas operacionais, pois um sistema operacional só recebe o controle da máquina quando ocorre alguma interrupção ou *trap*.

Uma **interrupção** é um sinal de *hardware* que faz com que o processador sinalizado interrompa a execução do programa que vinha executando (guardando informações para poder continuar, mais tarde, a execução desse programa do ponto onde parou) e passe a executar uma rotina específica que trata a interrupção gerada.

Um *trap* é uma instrução especial que, quando executada pelo processador, origina as mesmas ações ocasionadas por uma interrupção (salvamento de informações para poder continuar, mais tarde, a execução do programa e desvio para uma rotina específica que trata do *trap*). **Pode-se dizer que um trap é uma interrupção ocasionada por software.**

Interrupções podem ser originadas pelos vários dispositivos periféricos (terminais, discos, impressoras, etc.), pelo operador (através das teclas do console de operação) ou pelo relógio do sistema. O **relógio (timer)** é um dispositivo de *hardware* que decrementa automaticamente o conteúdo de um registrador ou posição de memória, com uma freqüência constante, e interrompe a UCP quando o valor decrementado atinge zero.

O sistema operacional garante que ocorrerá pelo menos uma interrupção (e ele voltará a ter o controle) dentro de um intervalo de tempo de **t** unidades. Imediatamente antes do sistema operacional entregar a UCP para um programa de usuário, ele atribui um valor **t** ao relógio e esse valor é decrementado até zero, fazendo com que o relógio, e consequentemente o tempo de execução do programa de usuário, demore **t** unidades de tempo.

Uma interrupção não afeta a instrução que está sendo executada pela UCP no momento em que ela ocorre: a UCP detecta interrupções apenas após o término da execução de uma instrução e antes do início da execução da instrução seguinte.

Os computadores possuem instruções para **mascarar** (desabilitar, inibir) o sistema de interrupções. Enquanto as interrupções estão mascaradas elas podem ocorrer, mas não são percebidas pelo processador. Neste caso, as interrupções ficam pendentes (enfileiradas) e somente serão percebidas quando for executada uma instrução que as desmascare.

Conforme já foi dito, os *traps* são instruções especiais que, quando executadas, originam ações idênticas às que ocorrem por ocasião de uma interrupção. Pode-se dizer que um ***trap* é uma interrupção prevista**, programada no sistema pelo próprio programador. Uma interrupção, por outro lado, é completamente imprevisível, ocorrendo em pontos que não podem ser pré-determinados.

Os *traps* têm a finalidade de permitir aos programas dos usuários a passagem do controle da execução para o sistema operacional. Por esse motivo também são denominadas **"chamadas do supervisor" ou "chamadas do sistema" (*supervisor call* ou *system call*)**. Os *traps* são necessários principalmente nos computadores que possuem **instruções protegidas** (privilegiadas). Nesses computadores o registrador de estado do processador possui um *bit* para indicar se a UCP está em estado privilegiado (estado de

sistema, estado de supervisor, estado mestre) ou não privilegiado (estado de usuário, estado de programa, estado escravo). Sempre que ocorre uma **interrupção ou trap**, o novo valor carregado no registrador de estado do processador indica **estado privilegiado de execução**.

No estado de supervisor, qualquer instrução pode ser executada e no estado de usuário apenas as instruções não protegidas podem ser executadas. Exemplos de instruções protegidas são instruções para desabilitar e habilitar interrupções e instruções para realizar operações de E/S. Operações que envolvam o uso de instruções protegidas só podem ser executadas pelo sistema operacional. Dessa forma, quando um programa de usuário necessita executar alguma dessas operações, o mesmo deve executar um *trap*, passando como argumento ao sistema operacional o número que identifica a operação que está sendo requerida.

Atividades de avaliação

1. Quais são as finalidades principais de um sistema operacional?
2. Qual era a utilidade do monitor residente?
3. Defina processamento batch?
4. O que é bufferização?
5. O que é *Spooling?*
6. Cite as vantagens de sistemas multiprogramáveis.
7. Considere as diversas definições de sistema operacional. Considere se o sistema operacional deve incluir aplicações como navegadores da Web e programas de e-mail. Defenda tanto que ele deve quanto que ele não deve fazer isso e justifique suas respostas.
8. Por que alguns sistemas armazenam o sistema operacional em firmware enquanto outros o armazenam em disco?

Capítulo 2

Processos

Objetivos

- Definir processos e seus estados.
- Definir escalonamento de processos e apresentar alguns algorítmos de escalonamento.
- Apresentar comunicação entre processos e mecanismos de sincronização de processos.
- Discutir *deadlock*.

1. Introdução

O conceito de processo é, certamente, o conceito mais importante no estudo de sistemas operacionais. Para facilitar o entendimento deste conceito, consideremos um computador funcionando em multiprogramação (isto é, tendo vários programas executando simultaneamente e ativos na memória). Cada programa em execução corresponde a um procedimento (seqüência de instruções) e um conjunto de dados (variáveis utilizadas pelo programa).

Além das instruções e dados, cada programa em execução possui também uma área de memória correspondente para armazenar os valores dos registradores da UCP. Essa área de memória é conhecida como registro descritor (ou bloco descritor, bloco de contexto, registro de estado, vetor de estado) que, além dos valores dos registradores da UCP, contém também outras informações.

Assim, em um determinado sistema, cada programa em execução constitui um processo. Portanto, podemos definir **processo** como sendo um **programa em execução, o qual é constituído por uma seqüência de instruções, um conjunto de dados e um registro descritor.**

2. Multiprogramação

Num ambiente de multiprogramação, quando existe apenas um processador, cada processo é executado um pouco de cada vez, de forma intercalada. O sistema operacional aloca a UCP um pouco para cada processo, em uma ordem que não é previsível, em geral, pois depende de fatores externos aos processos, que variam no tempo (carga do sistema, prioridade dos processos, por exemplo). Um processo após receber a UCP, só perde o controle da

execução quando ocorre uma interrupção ou quando ele executa um *trap*, requerendo algum serviço do sistema operacional.

As interrupções são transparentes aos processos, pois os efeitos das mesmas é apenas parar, temporariamente, a execução de um processo, o qual continuará sendo executado, mais tarde, como se nada tivesse acontecido. Um *trap*, por outro lado, é completamente diferente, pois bloqueia o processo até que seja realizado o serviço solicitado por ele ao sistema operacional.

Em um sistema com multiprocessamento (com mais de uma UCP), a única diferença em relação ao ambiente monoprocessado é que o sistema operacional passa a dispor de mais processadores para alocar os processos, e neste caso tem-se realmente a execução simultânea de vários processos.

Um sistema monoprocessado executando N processos de forma intercalada pode ser visto como se possuísse N **processadores virtuais**, um para cada processo em execução. Cada processador virtual teria 1/N da velocidade do processador real (desprezando-se o *overhead* existente na implementação da multiprogramação). O ***overhead* de um sistema operacional é o tempo que o mesmo gasta na execução de suas próprias funções**, como por exemplo, o tempo gasto para fazer a distribuição da UCP entre os processos. É o tempo durante o qual o sistema não está produzindo "trabalho útil" para qualquer usuário.

Os processos durante suas execuções solicitam operações de E/S que são executadas em dispositivos mais lentos que a UCP, pois os dispositivos periféricos possuem componentes mecânicos, que funcionam a velocidades muito inferiores às dos dispositivos eletrônicos.

Durante o tempo em que um processo deve ficar esperando a realização de uma operação de E/S, a UCP pode ser entregue a outro processo. Dessa forma, a utilização dos recursos será mais eficiente. Além de uma melhor utilização dos recursos, a multiprogramação permite que as requisições de serviço dos usuários sejam atendidas com menores tempos de resposta. Por exemplo, na situação em que um *job* pequeno e prioritário seja submetido ao sistema após um *job* demorado já ter iniciado a execução, a multiprogramação permite que o *job* pequeno seja executado em paralelo e termine muito antes do término do *job* longo.

Durante suas execuções, os processos dos usuários, ocasionalmente, através de *traps*, fazem requisições ao sistema operacional (para gravar um arquivo no disco, por exemplo). Recebendo a requisição, o sistema operacional bloqueia o processo (deixa de dar tempo de UCP a ele) até que a operação solicitada seja completada. Quando isto acontece o processo é desbloqueado e volta a competir pela UCP com os demais processos.

Um processo pode assumir três estados em um sistema, são eles: executando, bloqueado e pronto. Quando um processo está realmente usando a UCP, diz-se que o mesmo está no estado **executando**. Quando está esperando pelo término de um serviço que solicitou, diz-se que está no estado **bloqueado**. Quando o processo tem todas as condições para ser executado e só não está em execução porque a UCP está alocada para outro processo, diz-se que o mesmo está no estado **pronto**.

O sistema operacional mantém uma lista (fila) dos processos que estão prontos, a chamada lista de processos prontos, e é essa lista que servirá de base para o algoritmo de escalonamento executado pelo escalonador. O diagrama da Figura 4 mostra como os estados de um processo podem mudar durante a execução.

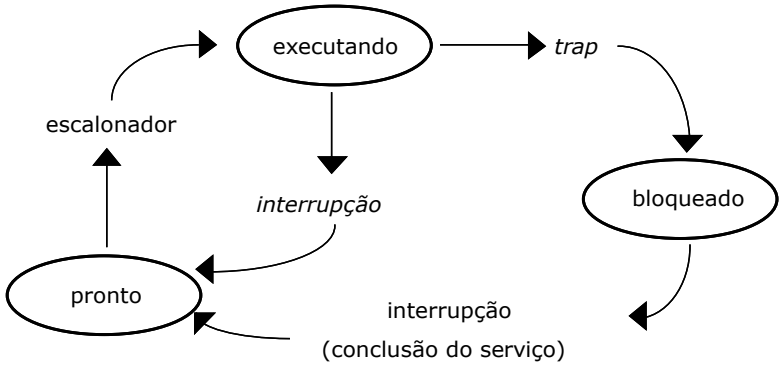


Figura 4 - Estados sucessivos de um processo no sistema computacional.

Fonte: Tanenbaum (2008 p. 74).

O componente do sistema operacional que, após o atendimento de uma interrupção ou trap, escolhe o próximo processo a ser executado é denominado **escalonador de processos** ou despachador de processos.

Em geral, um trap faz com que o processo fique bloqueado. Entretanto, em algumas ocasiões especiais, quando o sistema operacional pode atender imediatamente a requisição de serviço, o processo pode ser novamente despachado (escalonado), não ocorrendo o bloqueio.

Quando um job é admitido no sistema computacional, um processo correspondente é criado e normalmente inserido no final da lista de prontos. O processo se move gradualmente para a cabeça dessa lista conforme os processos anteriores a ele forem sendo executados pela UCP. Quando o processo alcança a cabeça da lista, e quando a UCP torna-se disponível, o processo é alocado à UCP e diz-se que foi feito uma transição do estado "pronto" para o estado "executando". A transferência da UCP para o primeiro processo da lista de pronto é chamada **escalonamento**, e é executada pelo escalonador.

Para prevenir que um processo monopolize o sistema computacional acidentalmente ou propositadamente, o SO (Sistema Operacional) tem um **relógio interno (relógio de interrupção ou temporizador de intervalo)** o qual faz um processo ser executado somente por um **intervalo de tempo específico ou *quantum***. Se o processo voluntariamente não libera a UCP antes de expirar seu intervalo de tempo, esse relógio gera uma interrupção, devolvendo ao SO o controle do sistema. O SO altera o estado do processo de "executando" para "pronto" e coloca o primeiro processo da lista de prontos em execução. Estas transições de estado são indicadas como:

Temporizador_Esgotou (nome_do_processo): executando → pronto

Escalona (nome_do_processo): pronto → executando

Se o processo atualmente na UCP iniciar uma operação de E/S antes de expirar o seu *quantum*, o processo corrente voluntariamente libera a UCP (isto é, ele se bloqueia, ficando pendente até completar a operação de E/S). Esta transição de estado é:

Bloqueia (nome_do_processo): executando → bloqueado

Quando termina a operação que tornou o processo bloqueado, este passa para o estado pronto. A transição que faz tal operação é definida como:

Acorda (nome_do_processo): bloqueado è pronto

> Note que somente um estado de transição é inicializado pelo próprio processo — a transição Bloqueado — os outros três estados de transição são inicializados por entidades externas ao processo.

3. O núcleo do Sistema Operacional

Todas as operações envolvendo processos são controladas por uma porção do sistema operacional chamada de **núcleo**, **core**, ou **kernel**. O núcleo normalmente representa somente uma pequena porção do código que em geral é tratado como sendo todo o sistema operacional, mas é a parte de código mais intensivamente utilizada. Por essa razão, o núcleo comumentemente reside em armazenamento primário (memória RAM) enquanto outras porções do sistema operacional são chamadas da memória secundária (disco, por exemplo) quando necessário.

Uma das funções mais importantes incluídas no núcleo é o processamento de interrupções. Em grandes sistemas multiusuários, uma constante rajada de interrupções é direcionada ao processador. Respostas rápidas a essas interrupções são essenciais para manter o bom desempenho na utilização dos recursos do sistema, e para prover tempos de resposta aceitáveis pelos usuários.

Enquanto o núcleo responde a uma interrupção, ele desabilita as outras interrupções. Após a finalização do processamento da interrupção atual, as interrupções são novamente habilitadas. Com um fluxo permanente de interrupções, é possível que o núcleo mantenha interrupções desabilitadas por um grande período de tempo; isto pode resultar em respostas insatisfatórias para interrupções.

Entretanto, núcleos são projetados para fazer o "mínimo" processamento possível para cada interrupção, e então passar o restante do processamento de uma interrupção para um processo apropriado do sistema operacional, que pode terminar de tratá-las enquanto o núcleo continua apto a receber novas interrupções. Isto significa que as interrupções podem ficar habilitadas durante uma porcentagem muito maior do tempo, e o sistema operacional torna-se mais eficiente em responder às requisições das aplicações dos usuários.

3.1. Um resumo das funções do núcleo

Um sistema operacional normalmente executa as seguintes funções:

- Manipulação de interrupções;
- Criação e destruição de processos;
- Troca de contexto de processos;
- Desabilitação de processos;
- Suspensão e reativação de processos;
- Sincronização de processos;
- Intercomunicação entre processos;
- Suporte a atividades de E/S;
- Suporte à alocação e desalocação de armazenamento;
- Suporte ao sistema de arquivos;
- Suporte a um mecanismo de chamada/retorno de procedimentos;
- Suporte a certas funções do sistema de contabilização.

4. Escalonamento de processos

Em um sistema de multiprogramação existem vários processos querendo executar e, para isso, devem ser alocados à UCP. No entanto, apenas um processo pode ocupar a UCP por vez. Portanto, quando um ou mais processos estão **prontos** para serem executados, o sistema operacional deve decidir qual deles vai ser executado primeiro, ou seja, qual deles ganhará o controle da UCP em determinado instante.

A parte do sistema operacional responsável por essa decisão é chamada **escalonador**, e o algoritmo usado pelo escalonador é chamado de **algoritmo de escalonamento**. Os algoritmos de escalonamento dos primeiros sistemas operacionais, baseados em cartões perfurados e unidades de fita, eram simples: eles simplesmente deveriam executar o próximo *job* na fita ou leitora de cartões. Em sistemas multiusuário e de tempo compartilhado, muitas vezes combinados com *jobs batch* em *background*, no entanto, o algoritmo de escalonamento é mais complexo.

Antes de vermos os algoritmos de escalonamento, vejamos os critérios que devem ser considerados por tais algoritmos:

1. **Justiça**: fazer com que cada processo ganhe seu tempo justo de UCP;
2. **Eficiência**: manter a UCP ocupada 100% do tempo (se houver demanda);
3. **Tempo de Reposta**: minimizar o tempo de resposta para os usuários interativos;
4. **Tempo de Turnaround**: minimizar o tempo que usuários *batch* devem esperar pelo resultado;
5. **Throughput** maximizar o número de *jobs* processados por unidade de tempo.

Um pouco de análise mostrará que alguns desses objetivos são contraditórios, tornando mais difícil a tarefa do escalonador. Para minimizar o tempo de resposta para usuários interativos, por exemplo, o escalonador não deveria rodar nenhum *job batch* (exceto entre 3 e 6 da manhã, quando os usuários interativos estão dormindo). Usuários *batch*, contudo, não gostarão deste algoritmo, porque ele viola a regra 4.

Tendo em mente o requisito de alocação justa da UCP entre os processos, um único processo não deve monopolizá-la. Para isso, praticamente todos os computadores possuem um **mecanismo de relógio (clock)** que causa periodicamente uma interrupção. A cada interrupção de relógio, o sistema operacional assume o controle e decide se o processo que está atualmente na UCP pode continuar executando ou se já ganhou tempo de UCP suficiente. Neste último caso, o processo é suspenso e a UCP é dada a outro processo.

A estratégia de permitir ao SO temporariamente suspender a execução de processos que queiram continuar a executar é chamada de **escalonamento preemptivo,** em contraste com o método **execute até o fim** dos antigos sistemas *batch* (escalonamento não-preemptivo). Como vimos até agora, em sistemas preemptivos um processo pode perder a UCP a qualquer momento para outro processo, sem qualquer aviso. Isto gera condições de corrida e a necessidade de mecanismos de controle, tais como semáforos, contadores

de eventos, monitores, ou outros, que possibilitem a comunicação entre os processos. Esses mecanismos são discutidos mais adiante.

Existem vários tipos de algoritmos de escalonamento, cada um com características específicas, sendo importante conhecê-los para decidir qual o mais apropriado para determinado sistema.

4.1. Escalonamento FIFO ou FCFS

Talvez a disciplina de escalonamento mais simples que exista seja a *First-In-First-Out* - FIFO (o primeiro a entrar é o primeiro a sair). Vários autores referem-se a este algoritmo como FCFS - *First-Come-First-Served* (o primeiro a chegar é o primeiro a ser servido). Processos prontos são inseridos em uma fila e são alocados à UCP de acordo com sua ordem de chegada nessa fila. Uma vez que um processo ganhe a UCP, ele executa até terminar. FIFO é uma disciplina **não preemptiva.**

Ela é justa no sentido de que todos os *jobs* são executados, e na ordem de chegada, mas é injusta no sentido que grandes *jobs* podem fazer pequenos *jobs* esperarem, e *jobs* sem grande importância podem fazer com que *jobs* importantes esperem. FIFO oferece uma menor variação nos tempos de resposta e é, portanto, mais previsível do que outros esquemas. Ele não é útil no escalonamento de usuários interativos porque não pode garantir bons tempos de resposta. **Sua natureza é essencialmente a de um sistema *batch*.**

4.2. Escalonamento *round robin* (RR)

Um dos mais antigos, simples, justos, e mais largamente utilizados algoritmo de escalonamento é o **Round Robin**. Cada processo recebe um intervalo de tempo, chamado ***quantum***, durante o qual ele pode executar. Se o processo ainda estiver executando ao final do *quantum*, o sistema operacional interrompe a sua execução e passa a UCP a outro processo. Se um processo bloqueou ou terminou antes do final do *quantum*, a troca da UCP para outro processo é obviamente feita assim que o processo bloqueia ou termina. *Round Robin* é fácil de implementar. Tudo que o escalonador tem a fazer é manter uma lista de processos que desejam executar, conforme aFigura 5(a). Quando o *quantum* de um processo acaba, ele é colocado no final da lista, conforme a Figura 5(b).

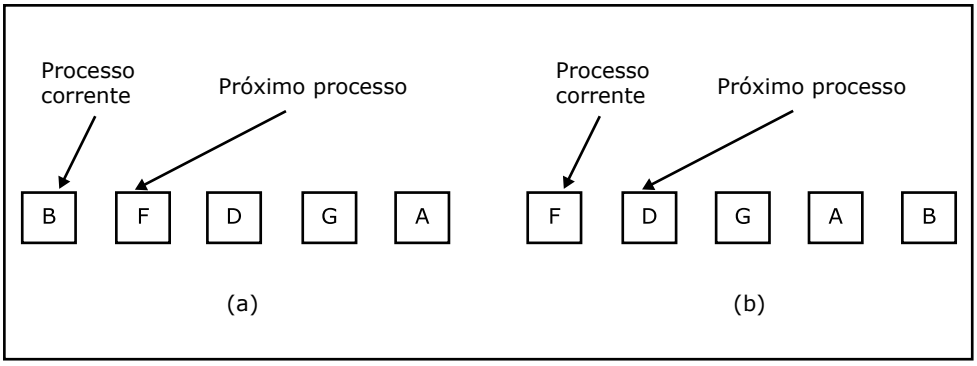


Figura 5 - Escalonamento Round Robin. (a) Lista de processos a executar. (b) Lista de processos a executar depois de terminado o *quantum* de 'B'.
Fonte: Tanenbaum (2008 p. 111).

Assim, o algoritmo Round Robin é semelhante ao FIFO, mas com a diferença de que é **preemptivo**: os processos não executam até o seu final, mas sim durante um certo tempo (quantum), um por vez. Executando sucessivamente em intervalos de tempo o *job* acaba por terminar sua execução em algum momento.

Um aspecto importante do algoritmo R*ound Robin* é a duração do *quantum*. O que é melhor para o bom desempenho do sistema, um *quantum* grande ou um *quantum* pequeno?

Mudar de um processo para outro requer certo tempo para a administração do sistema — salvar e carregar registradores e mapas de memória, atualizar tabelas e listas do SO, etc. Suponha que esta **troca de processos** ou **troca de contexto**, como às vezes é chamada, dure 5 ms. Suponha também que o *quantum* esteja ajustado em 20 ms. Com esses parâmetros, após fazer 20 ms de trabalho útil, a UCP terá que gastar 5 ms com troca de contexto. Assim, 20% do tempo de UCP é gasto com o ***overhead*** administrativo.

Para melhorar a eficiência da UCP, poderíamos ajustar o *quantum* para digamos, 500 ms. Agora o tempo gasto com troca de contexto é menos do que 1%. Mas considere o que aconteceria se dez usuários apertassem a tecla <ENTER> exatamente ao mesmo tempo, cada um disparando um processo. Dez processos serão colocados na lista de processos aptos a executar. Se a UCP estiver ociosa, o primeiro começará imediatamente, o segundo não começará antes de ½ segundo depois, e assim por diante. O azarado do último processo somente começará a executar 5 segundos depois de o usuário ter apertado <ENTER>, isto se todos os outros processos tiverem utilizado todo o seu *quantum*. Muitos usuários vão achar que o tempo de resposta de 5 segundos para um comando simples é "muito" grande.

Conclusão: ajustar um quantum muito pequeno causa muitas trocas de contexto e diminui a eficiência da UCP, mas ajustá-lo para um valor muito alto causa um tempo de resposta inaceitável para pequenas tarefas interativas. Um quantum em torno de 100 ms freqüentemente é um valor razoável.

4.3. Escalonamento com prioridades

O algoritmo R*ound Robin* assume que todos os processos são igualmente importantes. Freqüentemente, as pessoas que possuem e operam centros de computação possuem um pensamento diferente sobre este assunto. Em uma Universidade, por exemplo, as prioridades de processamento normalmente são para a administração em primeiro lugar, seguida de professores, secretárias e finalmente estudantes.

A necessidade de se levar em conta fatores externos que interferem na definição da "importância" dos processos, nos leva ao **escalonamento com prioridades**. A idéia básica é a seguinte: **cada processo possui uma prioridade associada**, **e o processo pronto para executar com a maior prioridade é quem ganha o processador**.

Para evitar que processos com alta prioridade executem indefinidamente, o escalonador pode decrementar, a cada *tick* de relógio (isto é, a cada interrupção de relógio), a prioridade do processo em execução. Se esta ação fizer com que a prioridade do processo atual se torne menor do que a prioridade do processo que possuía a segunda mais alta prioridade, então uma troca de processos ocorre.

Prioridades podem ser associadas aos processos estaticamente ou dinamicamente. Em um computador militar, por exemplo, processos iniciados por generais deveriam começar com a prioridade 100, processos de coronéis com 90, de majores com 80, de capitães com 70, de tenentes com 60, e assim por diante. Alternativamente, em um centro de computação comercial (bancos), *jobs* de alta prioridade poderiam custar 100 dólares por hora, os de média prioridade 75 dólares por hora, e os de baixa prioridade 50 dólares por hora.

O sistema operacional UNIX possui um comando, **nice**, que permite a um usuário voluntariamente reduzir a prioridade de um processo seu, de modo a ser gentil (*nice*) com os outros usuários. Na prática, ninguém utiliza este comando, pois ele somente permite baixar a prioridade de um processo. Entretanto, o superusuário UNIX pode aumentar a prioridade dos processos.

Prioridades podem também ser atribuídas dinamicamente pelo sistema para atingir certos objetivos de desempenho. Por exemplo, alguns processos

são altamente limitados por E/S, e passam a maior parte do tempo esperando por operações desse tipo. Sempre que um desses processos quiser a UCP, ele deve obtê-la imediatamente, para que possa usá-la até iniciar sua próxima requisição de E/S, de forma que ele libere a UCP para outro processo e passe a executar a E/S enquanto outro processo utiliza a UCP.

Fazer com que processos limitados por E/S esperem um bom tempo pela UCP significa deixá-los um tempo demasiado ocupando memória. Um algoritmo simples para prover um bom serviço a um processo limitado por E/S é ajustar a sua prioridade para 1/*f*, onde *f* é a fração do último *quantum* de processador que o processo utilizou. Um processo que utilizou todo o *quantum* ganharia uma prioridade 1, outro processo que executou durante 50 ms antes de bloquear ganharia prioridade 2, enquanto um processo que utilizou somente 2 ms do seu *quantum* de 100 ms ganharia uma prioridade 50.

É frequentemente conveniente agrupar processos em classes de prioridades e utilizar escalonamento com prioridades entre as classes junto com o escalonamento R*ound Robin* dentro de cada classe. Por exemplo, em um sistema com quatro classes de prioridade, o escalonador executa os processos na classe 4 segundo a política *Round Robin* até que não haja mais processos na classe 4. Então ele passa a executar os processos de classe 3 também segundo a política *Round Robin*, enquanto houver processos nesta classe. Então executa processos da classe 2 e assim por diante. Se as prioridades não forem ajustadas de tempos em tempos, os processos nas classes de prioridades mais baixas podem sofrer o fenômeno que chamamos **starvation** (o processo nunca recebe o processador, pois sua vez nunca chega).

4.4. Escalonamento com prazos

No **escalonamento com prazos** certos *jobs* são escalonados para serem completados até uma certa data ou hora, ou dentro de um prazo. Esses *jobs* podem ter alta importância se entregues dentro do tempo, ou podem não ter utilidade alguma se terminarem de ser processados além do tempo previsto no prazo. O usuário normalmente paga um "prêmio" para ter seu *job* completado em tempo.

Este tipo de escalonamento é complexo por muitas razões:

- O usuário deve fornecer previamente as necessidades do *job* por recursos do sistema. Tal informação raramente está disponível;
- O sistema deve executar o *job* com prazo sem degradar o serviço para os outros usuários;

- O sistema deve cuidadosamente planejar suas necessidades por recursos durante o prazo. Isto pode ser difícil porque novos *jobs* podem chegar e adicionar uma demanda imprevisível ao sistema;
- Se muitos *jobs* com prazos existirem ao mesmo tempo, o escalonamento com prazos poderia se tornar tão complexo que métodos sofisticados de otimização poderiam ser necessários para garantir que as metas fossem atingidas;
- O intensivo gerenciamento de recursos necessário ao escalonamento com prazos pode gerar um *overhead* substancial. Mesmo que os usuários dos *jobs* com prazos estejam aptos a pagar uma taxa suficientemente alta pelos serviços recebidos, o consumo total de recursos do sistema pode se tornar tão alto que o resto da comunidade de usuários poderia ter um serviço degradado. Tais conflitos devem ser considerados cuidadosamente por projetistas de sistemas operacionais.

4.5. Escalonamento *shortest-job-first* (SJF)

***Shortest-job-first* (o menor *job* primeiro) é um algoritmo não preemptivo no qual o *job* na fila de espera com o menor tempo total estimado de processamento é executado em seguida.** SJF reduz o tempo médio de espera sobre o algoritmo FIFO. Entretanto, os tempos de espera têm uma variação muito maior (são mais imprevisíveis) do que no algoritmo FIFO, especialmente para grandes *jobs*.

SJF favorece *jobs* pequenos em prejuízo dos *jobs* maiores. Muitos projetistas acreditam que quanto mais curto o *job*, melhor serviço ele deveria receber. Não há um consenso universal quanto a isso, especialmente quando prioridades de *jobs* devem ser consideradas.

SJF seleciona *jobs* para serviço de uma maneira que garante que o próximo *job* irá completar e deixar o sistema o mais cedo possível. Isto tende a reduzir o número de *jobs* esperando. **Como resultado, SJF pode minimizar o tempo médio de espera conforme os *jobs* passam pelo sistema.**

O problema óbvio com SJF é que ele requer um conhecimento preciso de quanto tempo um *job* demorará para executar, e esta informação não está usualmente disponível. O melhor que SJF pode fazer é tomar como basea estimativa do usuário para o tempo de execução de cada *job*. Em ambientes de produção onde os mesmos *jobs* executam regularmente, pode ser possível prover estimativas razoáveis. Mas em ambientes de desenvolvimento, os usuários raramente sabem durante quanto tempo seus programas executarão.

Basear-se nas estimativas dos usuários possui uma ramificação interessante. Se os usuários sabem que o sistema está projetado para favorecer *jobs* com tempos estimados de execução pequenos, eles podem fornecer estimativas com valores menores que os reais. O escalonador pode ser projetado, entretanto, para remover esta tentação. O usuário pode ser avisado previamente que se o *job* executar por um tempo maior do que o estimado, ele será abortado e o usuário terá que ser cobrado pelo trabalho.

Uma segunda opção é executar o *job* pelo tempo estimado mais uma pequena porcentagem extra, e então salvá-lo no seu estado corrente de forma que possa ser continuado mais tarde. O usuário, é claro, teria que pagar por este serviço, e ainda sofreria um atraso na finalização de seu *job*. Outra solução é executr o *job* durante o tempo estimado a taxas de serviços normais, e então cobrar uma taxa diferenciada (mais cara) durante o tempo que executar além do previsto. Dessa forma, o usuário que fornecer tempos de execução subestimados pode pagar um preço alto por isso.

SJF, assim como FIFO, é não preemptivo e, portanto, não é útil para sistemas de tempo compartilhado nos quais tempos razoáveis de resposta devem ser garantidos.

4.6. Comunicação entre Processos (IPC)

Tanto no paralelismo físico (real, com várias UCP) como no lógico (virtual, uma UCP compartilhada por vários processos), existe a possibilidade de que processos acessem dados compartilhados ou áreas comuns de memória. Quando dois processos executam ao mesmo tempo e tentam acessar os mesmos dispositivos, acontecem as condições de corrida. Ou seja, se o processo A e o processo B tentam acessar o mesmo disco ao mesmo tempo, acontece uma condição de corrida que o sistema tem que tratar, pois se os dois processos escreverem no disco ao mesmo tempo, pode acontecer erro nessa escrita. Isto implica, portanto, na necessidade de desenvolvimento de mecanismos que realizem a sincronização entre os processos.

Processos são concorrentes se eles executam no mesmo instante de tempo e implicam em compartilhamento dos mesmos recursos do sistema, tais como mesmos arquivos, registros, dispositivos de E/S e áreas de memória. Um sistema operacional multiprogramável deve fornecer mecanismos de **sincronização de processos**, para garantir a **comunicação interprocessos** (ou seja, entre os processos) e o acesso aos recursos compartilhados de forma organizada.

4.7. Exclusão mútua

Considere um sistema com muitos terminais em tempo compartilhado. Assuma que os usuários de cada terminal digitam uma linha e pressionam a tecla <ENTER>. Suponha então que o sistema quer monitorar o número total de linhas que os usuários entraram no sistema desde o início do dia. Se cada terminal for monitorado por um processo, toda vez que um destes processos recebe uma linha do terminal, ele incrementa de 1 uma variável global compartilhada do sistema, chamada LINHAS. Considere o que aconteceria se dois processos tentassem incrementar a variável LINHAS ao mesmo tempo. A Figura 6 apresenta o código que é executado por cada um dos processos.

```
LOAD LINHAS ; Lê a variável linhas no registrador acumulador
ADD  1;                Incrementa o registrador acumulador
STORE        LINHAS ; Armazena o conteúdo do acumulador na variável
```

Fig. 6 - Código para contar o número de linhas que os usuários entram no sistema.
Fonte: Oliveira,Carissimi e Toscani (2004, p. 39).

Suponhamos agora a seguinte situação:

- LINHAS tem o valor atual 21687;
- O processo A executa as instruções LOAD e ADD, deixando então o valor 21688 no acumulador;
- O processo A perde o processador (após o término de seu *quantum*) para o processo B;
- O processo B então executa as três instruções, fazendo com que a variável linhas tenha o valor 21688;
- O processo B perde o processador para o processo A, que continua executando de onde tinha parado e, portanto, executa a instrução STORE e armazena 21688 (esse valor tinha sido aramzenado no acumulador, mas ainda não tinha sido gravado na memória, pois não tinha sido executada a instrução STORE) na variável LINHAS.

Devido à falta de comunicação entre os processos, o sistema deixou de contar uma linha — o valor correto seria 21689.

Qual o problema que realmente aconteceu? O problema está no fato de dois ou mais processos escreverem em uma variável compartilhada (ou seja, escreverem na mesma variável, que no caso é a variável LINHAS). Vários processos podem ler uma variável compartilhada sem problemas. Entretanto, quando um processo lê uma variável que outro processo está escrevendo,

ou quando um processo tenta escrever em uma variável que outro processo também esteja escrevendo, resultados inesperados podem acontecer.

O problema pode ser resolvido dando a cada processo **acesso exclusivo** à variável LINHAS. Enquanto um processo incrementa a variável, todos os outros processos que querem fazer a mesma coisa no mesmo instante de tempo deverão esperar; quando o primeiro processo terminar o acesso à variável compartilhada, um dos outros processos passa a ter acesso à variável. Assim, cada processo acessando a variável exclui todos outros de fazê-lo ao mesmo tempo. Isto é chamado de **exclusão mútua**.

4.8. Regiões críticas

A exclusão mútua somente precisa estar presente nos instantes em que os processos acessam dados compartilhados que estejam possam ser modificados — quando os processos estão executando operações que não geram conflito de um processo com o outro, eles podem ser liberados para processarem concorrentemente. Quando um processo está acessando dados compartilhados modificáveis, é dito que o processo está em sua **região crítica** ou seção crítica.

Para evitarmos problemas como o mostrando acima, é necessário garantir que quando um processo está em sua região crítica, todos os outros processos (pelo menos aqueles que acessam a mesma variável compartilhadas) sejam proibidos de executar as suas respectivas regiões críticas.

Enquanto um processo está em sua região crítica, outros processos podem certamente continuar sua execução fora de suas regiões críticas. Quando um processo deixa sua região crítica, um dos processos esperando para entrar em sua própria região crítica pode prosseguir. Garantir a exclusão mútua é um dos principais problemas em programação concorrente. Muitas soluções foram propostas: algumas baseadas em *software* e outras baseadas em *hardware*; algumas em baixo nível, outras em alto nível; algumas requerendo cooperação voluntária entre processos, outras exigindo uma rígida aderência a protocolos estritos.

4.9. Primitivas de exclusão mútua

O programa concorrente apresentado a seguir, implementa o mecanismo de contagem de linhas de texto digitadas pelos usuários, descrito anteriormente. A linguagem utilizada é uma pseudo-linguagem com sintaxe semelhante à linguagem Pascal, mas com mecanismos de paralelismo. Para maior simplicidade, vamos assumir que existam somente dois processos concorrentes nos programas apresentados nesta e nas seções seguintes. Manipular *n* processos concorrentes é consideravelmente mais complexo.

```
program exclusão_mútua;
var linhas_digitadas: integer;
procedure processo_um;
  while true do
    begin
      leia_próxima_linha_do_terminal;
      entermutex;
        linhas_digitadas := linhas_digitadas + 1;
      exitmutex;
      processe_a_linha;
    end;
procedure processo_dois;
  while true do
    begin
      leia_próxima_linha_do_terminal;
      entermutex;
        linhas_digitadas := linhas_digitadas + 1;
      exitmutex;
      processe_a_linha;
    end;
begin
  linhas_digitadas := 0;
  parbegin
    processo_um;
    processo_dois;
  parend;
end.
```

Fig. 7 - Programa concorrente para contar linhas de texto.

Fonte: Oliveira,Carissimi e Toscani (2004, p. 42).

O par de construções **entermutex/exitmutex** introduzidos no exemplo delimitam o código da região crítica de cada processo. Essas operações são normalmente chamadas de **primitivas de exclusão mútua**, e cada linguagem que as suporte pode implementá-las à sua maneira.

No caso de dois processos, como no exemplo, as primitivas operam da seguinte maneira. Quando o **processo_um** executa **entermutex**, se o **processo_dois** não está em sua região crítica, então o **processo _um** entra em sua região crítica, acessa a variável e então executa **exitmutex** para indicar que ele deixou sua região crítica.

Se o **processo_dois** está em sua região crítica quando o **processo_um** executa **entermutex**, então o **processo_um** espera até que o **processo_dois** execute **exitmutex**. Então o **processo_um** pode prosseguir e entrar em sua região crítica.

Se tanto o **processo_um** como o **processo_dois** executam simultaneamente **entermutex**, então somente um é liberado para prosseguir, enquanto o outro permanece esperando.

Para que essa solução funcione, cada instrução da máquina é executada indivisivelmente, isto é, uma vez que uma instrução começa a ser executada, ela termina sem nenhuma interrupção.

Vários métodos foram desenvolvidos para implementar a exclusão mútua. Alguns desses métodos são semáforos, monitores e passagem de mensagens, os quais são apresentados a seguir.

4.10. Semáforos

Um semáforo é uma **variável protegida** cujo valor somente pode ser acessado e alterado pelas **operações P e V**, e por uma operação de inicialização que chamaremos inicializa_semáforo. Semáforos binários podem assumir somente os valores 0 ou 1. Semáforos contadores podem assumir somente valores inteiros não negativos.

A operação **P** no semáforo **S**, denotada por **P(S)**, funciona da seguinte maneira:

```
if S > 0 then
  S := S - 1
else
  (espera no semáforo S)
```

A operação **V** no semáforo **S**, denotada por **V(S)**, funciona da seguinte maneira:

```
if (um ou mais processos estão esperando em S) then
  (deixe um desses processos continuar)
else
  S := S + 1;
```

As operações P e V são **indivisíveis**. A exclusão mútua no semáforo S é garantida pelas operações P(S) e V(S). Se vários processos tentam executar P(S) simultaneamente, somente um deles poderá prosseguir. Os outros ficarão esperando. Semáforos e operações com semáforos são comumente implementados no núclo do sistema operacional.

O exemplo a seguir ilustra o uso de semáforos para implementar exclusão mútua. Neste exemplo, **P(ativo)** é equivalente a **entermutex** e **V(ativo)** é equivalente a **exitmutex**.

```
program exemplo_semáforo;
var ativo: semaphore;

procedure processo_um;
begin
  while true do
  begin
    algumas_funcoes_um;
    P(ativo);
    regiao_critica_um;
    V(ativo);
    outras_funcoes_um;
  end
end;

procedure processo_dois;
begin
  while true do
  begin
    algumas_funcoes_dois;
    P(ativo);
    regiao_critica_dois;
    V(ativo);
    outras_funcoes_dois;
  end
end;
```

```
begin
  inicializa_semaforo(ativo, 1);
  parbegin
    processo_um;
    processo_dois
  parend
end.
```

Fig. 8 - Exclusão mútua utilizando semáforos.

Fonte: Oliveira,Carissimi e Toscani (2004, p. 47).

Em sistemas com um único processador, a indivisibilidade de P e V pode ser garantida simplesmente desabilitando interrupções enquanto operações P e V estão manipulando o semáforo. Isto impede que o processador seja "roubado" até que a manipulação do semáforo esteja completa. Em seguida as interrupções poderão ser novamente habilitadas.

4.11. A Relação produtor-consumidor

Em um programa seqüencial, quando um procedimento chama outro e lhe passa dados como parâmetros, os procedimentos são partes de um único processo e não operam concorrentemente. Mas quando um processo concorrente passa dados para outro processo concorrente, os problemas são mais complexos.

Considere a seguinte relação produtor-consumidor. Suponha que um processo, um **produtor**, esteja gerando informação que um segundo processo, o **consumidor**, esteja utilizando. Suponha que eles se comunicam por uma única variável inteira compartilhada, chamada **numero**. O produtor faz alguns cálculos e então escreve o resultado em **numero**; o consumidor lê o dado de **numero** e o imprime.

É possível que os processos produtor e consumidor executem sem problemas, e que suas velocidades sejam compatíveis. Se cada vez que o produtor depositar um resultado em **numero** o consumidor puder lê-lo e imprimí-lo, então o resultado impresso certamente representará a seqüência de números gerados pelo produtor. Mas suponha que as velocidades dos processos sejam incompatíveis. Se o consumidor estiver operando mais rápido que o produtor, o consumidor poderia ler e imprimir o mesmo número duas vezes (ou talvez várias vezes) antes que o produtor depositasse o próximo número.

Se o produtor estiver operando mais rápido que o consumidor, o produtor poderia sobrescrever seu resultado prévio sem que o consumidor tivesse

a chance de lê-lo e imprimí-lo; um produtor rápido poderia de fato fazer isto várias vezes de forma que vários resultados poderiam ser perdidos.

Obviamente, o comportamento que desejamos aqui é que o produtor e o consumidor operem de tal forma que dados escritos para **numero** nunca sejam perdidos ou duplicados. A garantia de tal comportamento é um exemplo de **sincronização de processos**.

A seguir temos um exemplo de um programa concorrente que utiliza operações com semáforos para implementar uma relação produtor-consumidor.

```
program relacao_produtor_consumidor;
var numero: integer;
   numero_depositado: semaphore;
   numero_retirado: semaphore;

procedure processo_produtor;
var proximo_resultado: integer;
begin
  while true do
  begin
    calcule_proximo_resultado;
    P(numero_retirado);
    numero := proximo_resultado;
    V(numero_depositado)
  end
end;

procedure processo_consumidor;
var proximo_resultado: integer;
begin
  while true do
  begin
    calcule_proximo_resultado;
    P(numero_depositado);
    proximo_resultado := numero;
    V(numero_retirado);
```

```
    write(proximo_resultado)
  end
end;

begin
  inicializa_semaforo(numero_depositado, 0);
  inicializa_semaforo(numero_retirado, 1);
  parbegin
    processo_produtor;
    processo_consumidor
  parend
end.
```

Fig. 9 - Exemplo do problema produtor-consumidor utilizando semáforos.
Fonte: Oliveira,Carissimi e Toscani (2004, p. 51).

Dois semáforos contadores foram utilizados: **numero_depositado** é sinalizado (operação V) pelo produtor e testado (operação P) pelo consumidor; o consumidor não pode prosseguir até que um número seja depositado em **número**. O processo consumidor sinaliza (operação V) **numero_retirado** e o produtor o testa (operação P); o produtor não pode prosseguir até que um resultado existente em numero seja retirado. Os ajustes iniciais dos semáforos forçam o produtor a depositar um valor em **número** antes que o consumidor possa prosseguir.

Note que o uso de semáforos neste programa força a sincronização passo a passo; isto é aceitável porque só existe uma única variável compartilhada. É comum em relações produtor-consumidor que se tenha um buffer com espaço para várias variáveis. Com tal arranjo, o produtor e o consumidor não precisam executar à mesma velocidade na sincronização passo a passo. Ao contrário, um produtor rápido pode depositar diversos valores enquanto o consumidor está inativo, e um consumidor rápido pode retirar diversos valores enquanto o produtor está inativo.

4.12. Monitores

A comunicação interprocessos utilizando exclusão mútua e semáforos de eventos parece ser a solução definitiva. Entretanto, se analisarmos mais diretamente estas técnicas, veremos que elas possuem alguns problemas:

- São tão primitivos que é difícil expressar soluções para problemas de concorrência mais complexos;
- Seu uso em programas concorrentes aumenta a já difícil tarefa de provar a corretude de programas;
- O mau uso, tanto de forma acidental como maliciosa, poderia corromper a operação do sistema concorrente.

Particularmente com semáforos, alguns outros problemas podem ocorrer:

- Se a operação P for omitida, não é garantida a exclusão mútua;
- Se a operação V for omitida, tarefas esperando em uma operação P nunca executariam;
- Uma vez que a operação P é usada e o processo fica nela bloqueado, ele não pode desistir e tomar um curso de ação alternativo enquanto o semáforo estiver em uso;
- Um processo só pode esperar em um semáforo por vez, o que pode levá-lo a *deadlock* em algumas situações de alocação de recursos (o conceito de deadlock será apresentado mais adiante).

Assim, podemos perceber que o programador deve ser extremamente cuidadoso ao utilizar semáforos. Um súbito erro e tudo poderá parar de funcionar sem nenhuma explicação. Para tornar mais fácil a escrita de programas corretos, foi proposta uma primitiva de sincronização de alto nível chamada de **monitor**.

Um **monitor** é uma coleção de procedimentos, variáveis, e estruturas de dados que são todos agrupados em um tipo especial de módulo ou pacote. Processos podem chamar os procedimentos em um monitor sempre que o desejarem, mas eles não podem acessar diretamente as estruturas de dados internas do monitor através de procedimentos declarados fora do monitor. O exemplo abaixo ilustra um monitor escrito em nossa linguagem imaginária, semelhante à Pascal.

```
monitor exemplo;
  var i: integer;
     c: condition; {variável de condição}

  procedure produtor(x: integer);
  begin
  .
  .
  .
  end;

  procedure consumidor(x: integer);
  begin
  .
  .
  .
  end;

end monitor;
```

Fig. 10 - Exemplo de um monitor. Fonte: Oliveira,Carissimi e Toscani (2004, p. 52).

Monitores possuem uma importante propriedade que os torna útil para atingir exclusão mútua: **somente um processo por vez pode estar ativo em um monitor a qualquer momento.** Monitores são uma construção da própria linguagem de programação utilizada, de forma que o compilador sabe que eles são especiais, e pode manipular chamadas a procedimentos dos monitores de forma diferente da qual manipula outras chamadas de procedimentos.

Tipicamente, quando um processo chama um procedimento de um monitor, as primeiras instruções do procedimento irão checar se algum outro processo está ativo dentro do monitor. Se isto acontecer, o processo que chamou o procedimento será suspenso até que o outro processo tenha deixado o monitor. Se nenhum outro processo estiver usando o monitor, o processo que chamou o procedimento poderá entrar.

Uma vez que o compilador, e não o programador, é quem faz os arranjos para a exclusão mútua, é muito menos provável que alguma coisa dê errado. Em qualquer situação, a pessoa escrevendo o monitor não precisa saber como o compilador faz os arranjos para exclusão mútua. É

suficiente saber que ao transformar todas as regiões críticas em procedimentos de monitor, dois processos jamais executarão suas regiões críticas simultaneamente.

O problema produtor-consumidor pode ser implementado utilizando monitores e colocando os testes para *buffer* cheio e *buffer* vazio dentro de procedimentos de monitor, mas como o produtor poderia se bloquear se encontrasse o *buffer* cheio?

A Figura 11 apresenta uma solução para o problema Produtor-Consumidor utilizando monitor. A solução recai na introdução do conceito de **variáveis de condição**, juntamente com duas operações sobre elas, **wait** e **signal**. Quando um procedimento de monitor descobre que ele não pode continuar (por exemplo, o produtor encontra o *buffer* cheio), ele executa um **wait** em alguma variável de condição, digamos, **cheio**. Esta ação faz com que o processo que chamou o procedimento bloqueie.

Nesse caso, outro processo, como por exemplo, o consumidor, pode acordar seu parceiro suspenso executando um **signal** na variável de condição na qual seu parceiro está esperando. Para evitar que dois processos estejam ativos no monitor ao mesmo tempo, é preciso uma regra que diga o que acontece após um signal, quando existe mais de um processo bloqueado.

```
monitor ProdutorConsumidor;
 var cheio, vazio: condition; { variáveis de condição }
     cont: integer;

 procedure colocar;
 begin
  if cont = N then
    wait(cheio);
  entra_item;
  cont := cont + 1;
  if cont = 1 then
    signal(vazio);
 end;
 procedure remover;
 begin
  if cont = 0 then
    wait(vazio);
```

```
    remove_item;
    cont := cont - 1;
    if cont = N - 1 then
      signal(cheio);
  end;

  count := 0;
end monitor;

procedure produtor;
begin
  while true do
   begin
    produz_item;
    ProdutorConsumidor.colocar;
   end
end;

procedure consumidor;
begin
  while true do
   begin
    ProdutorConsumidor.remover;
    consome_item;
   end
end;
```

Figura 11 - Uma solução para o problema Produtor-Consumidor utilizando Monitor.
Fonte: Oliveira,Carissimi e Toscani (2004, p. 54).

Uma desvantagem dos monitores é que para utilizá-los, é necessária uma linguagem que os tenha por natureza. Muito poucas linguagens, como **Concurrent Euclid** os possuem, e compiladores para elas são raros.

4.13. Troca de mensagens

No caso de sistemas multiprocessados que não compartilham a memória, o método de comunicação entre processos utilizado é a troca de mensagens através das primitivas ***send*** e ***receive***. Assim como os semáforos, mas diferentemente dos monitores, as primitivas são chamadas de sistema e não construções de linguagem. Dessa maneira, elas podem facilmente ser colocadas em rotinas de biblioteca, como:

```
send(destino, & mensagem);

e

receive(origem, & mensagem);
```

A primeira chamada envia uma mensagem para um dado destino; a segunda recebe uma mensagem de uma dada origem. Se nenhuma mensagem estiver disponível, o receptor poderá ficar bloqueado até que alguma mensagem chegue. Como alternativa, ele pode retornar uma mensagem imediatamente acompanhada de um código de erro.

4.14. Threads

Threads são como miniprocessos dentro de processos maiores. A principal razão para existirem threads é que em muitas aplicações ocorrem múltiplas atividades ao mesmo tempo. Algumas dessas atividades podem ser bloqueadas de tempos em tempos. O modelo de programação se torna mais simples quando decompomos uma aplicação em múltiplos threads sequenciais, que executam quase em paralelo. Esse argumento é o mesmo para a existência dos processos[1] só que, com os threads, adicionamos a capacidade de entidades paralelas compartilharem um mesmo espaço de endereçamento e todos os seus dados presentes nesse espaço. Isso é essencial para certas aplicações, nas quais múltiplos processos (com seus espaços de endereçamento separados) não funcionarão.

[1] Os mesmos problemas e soluções para a comunicação entre processos (*interprocess communication* – IPC) também se aplicam a comunicações entre threads.

Um segundo argumento para a existência de threads é que eles são mais fáceis (isto é, mais rápidos) de criar e destruir que os processos, pois não têm quaisquer recursos associados a eles. Em muitos sistemas, criar um thread é cem vezes mais rápido do que criar um processo. É útil ter essa propriedade quando o número de threads necessários se altera dinâmica e rapidamente.

Uma terceira razão é também um argumento de desempenho. O uso de threads não resulta em ganho de desempenho quando todos eles são UCP-bound (limitados pela UCP, isto é, muito processamento com pouca E/S). No entanto, quando há grande quantidade de computação e de E/S,

os threads permitem que essas atividades se sobreponham e, desse modo, aceleram a aplicação.

5. *Deadlocks* e adiamento indefinido

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 sistemas multiprogramados, o compartilhamento de recursos é uma das principais metas dos sistemas operacionais. Quando recursos, no entanto, 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.

O estudo de *deadlocks* envolve quatro áreas:

1. Prevenir
2. Evitar
3. Detecta
4. Recuperar

5.1. Exemplos de *deadlocks*

Deadlocks podem ocorrer de várias maneiras. Se um processo recebe a tarefa de esperar pela ocorrência de um determinado evento, e o sistema não inclui condições para que este evento ocorra, então tem-se o *deadlock* de um processo. *Deadlocks* desta natureza são extremamente difíceis de detectar, pois estão intimamente associados aos códigos dos processos que, nesses casos, provavelmente contêm erros.

A maioria dos *deadlocks* em sistemas reais geralmente envolve múltiplos processos competindo por múltiplos recursos. Vejamos alguns exemplos comuns.

5.2. Um *deadlock* de tráfego

A Figura 12 ilustra um tipo de *deadlock* que ocasionalmente ocorre em cidades. Um certo número de automóveis está tentando atravessar uma parte da cidade bastante movimentada, mas o tráfego ficou completamente paralisado, ou seja, os carros estão esperando por um evento que não vai acontecer, que é andar. O tráfego chegou numa situação onde somente a polícia pode resolver a questão, fazendo com que alguns carros recuem na área congestionada, de forma que o tráfego volte a fluir normalmente. A essa altura, contudo, os motoristas já se aborreceram e perderam tempo considerável.

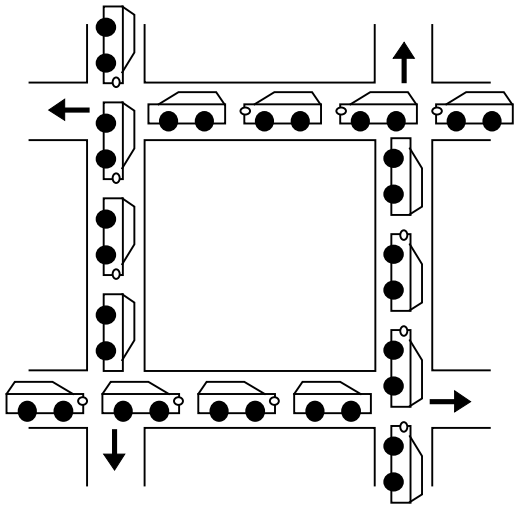


Figura 12 - Um *deadlock* de tráfego. Fig. 2.9
Fonte: Silberschatz, Galvin, Gagne (2010, p. 160).

5.3. Um *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, algumas vezes sendo chamados de recursos serialmente reusáveis).

Suponha que em um sistema o processo A detenha o recurso 1, e precise 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, pois o processo A só vai liberar o recurso 1 quando finalizar sua execução, mas para finalizar a execução precisa do recurso 2, que só será liberado por B quando B terminar sua execução, mas para isso, ele precisa que A libere o recurso 1. Esta situação de espera mútua é chamada muitas vezes de **espera circular.**

5.4. *Deadlock* em sistemas de *spooling*

Sistemas de *spooling* costumam estar sujeitos a *deadlocks*. Um sistema de *spooling* serve, por exemplo, para agilizar as tarefas de impressão do sistema. Ao invés do aplicativo mandar linhas a serem impressas 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. Vários *jobs* de impressão podem ser enfileirados e serão gerenciados pelo sistema de *spooling*.

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

Neste caso, todos os processos ficarão esperando pela liberação de espaço em disco, o que jamais vai acontecer, gerando portanto uma situação de deadlock. A solução neste caso seria o operador do sistema cancelar um dos *jobs* que parcialmente gerou dados para o spool, liberando espaço e, dessa forma, permitindo que os outros *jobs* continuem.

Para resolver o problema sem a intervenção do operador, o SO poderia alocar uma área maior de *spooling*, ou o tamanho da área de *spooling* poderia variar dinamicamente. Entretanto, pode acontecer de mesmo assim 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 esperar pelo término de toda a impressão solicitada pelo *job*. Isso faria com que o espaço fosse sendo liberado gradualmente, na velocidade em que a impressora conseguisse consumir os dados.

Também relacionado com *deadlocks* está o conceito de adiamento indefinido (starvation).

5.5. Adiamento indefinido

Em sistemas onde processos ficam esperando pela alocação de recursos ou pelas decisões de escalonamento, é possível que ocorra adiamento indefinido da execução do processo, também chamado de **bloqueamento indefinido** ou ***starvation*.**

Adiamento indefinido pode ocorrer devido às políticas de escalonamento de recursos do sistema. Quando recursos são alocados segundo um esquema de prioridades, por exemplo, é possível que um determinado processo espere indefinidamente por um recurso conforme processos com prioridades mais altas venham chegando.

Os sistemas operacionais devem ser justos com processos em espera, bem como devem considerar a eficiência de todo o sistema. Em alguns sistemas, o adiamento indefinido pode ser evitado permitindo que a prioridade de um processo em espera cresça conforme ele espera por um recurso. Isto é chamado de envelhecimento (*aging*).

5.6. Quatro condições necessárias para *deadlock*

Coffman, Elphick, e Shosani (1971) enumeraram as seguintes quatro **condições necessárias** que devem estar ocorrer para que um *deadlock* exista.

- Processos requisitam controle exclusivo dos recursos que eles necessitam (condição **"exclusão mútua"**);
- Processos **detêm** para si recursos já alocados enquanto estão esperando pela alocação de recursos adicionais (condição **"pega e espera"**, ou **"espera por"**);
- Recursos não podem ser removidos dos processos que os detêm até que os recursos sejam utilizados por completo (condição **"não preempção"**);
- Uma cadeia circular de processos existe de forma que cada processo detém um ou mais recursos que estão sendo requisitados pelo próximo processo na cadeia (condição **"espera circular"**).

Para que um *deadlock* ocorra, essas quatro condições necessárias devem ocorrer. Isto quer dizer que, se garantirmos que somente uma delas não ocorra, estaremos **prevenindo** a ocorrência de *deadlocks* em um determinado sistema.

5.7. Métodos para lidar com deadlocks

Basicamente, há três maneiras diferentes de lidar com *deadlocks*:

- Pode ser usado um protocolo para garantir que em um determinado sistema deadlocks jamais ocorrerão;
- Pode-se deixar o sistema entrar em um estado de deadlock e então tratar da sua recuperação;
- Pode-se simplesmente ignorar o problema, e fingir que deadlocks nunca ocorrem. Esta solução é usada pela maioria dos sistemas operacionais, inclusive o UNIX.

Para garantir que *deadlocks* nunca ocorram, o sistema pode usar um esquema de **prevenir deadlocks**, ou **evitar deadlocks**. A **prevenção** de *deadlocks* é um conjunto de regras de requisição de recursos que garantem que pelo menos uma das quatro condições necessárias para a ocorrência de *deadlocks* não ocorra. **Evitar** *deadlocks*, por outro lado, requer que seja fornecida ao sistema operacional informação adicional sobre quais recursos um processo irá requisitar e usar durante sua execução. Com o conhecimento dessa informação, é possível decidir, a cada requisição, se o processo pode prosseguir, recebendo o recurso solicitado, ou se deve esperar. Cada requisi-

ção requer que o sistema operacional considere os recursos atualmente disponíveis, os recursos alocados a cada processo, e as futuras requisições e liberações de cada processo, para que possa decidir se a requisição corrente pode ser satisfeita ou se deve ser adiada.

Se não são usadas estratégias para prevenir ou para evitar *deadlocks*, existe a possibilidade de ocorrência destes. Neste ambiente, o sistema operacional pode possuir um **algoritmo que consiga detectar** se ocorreu um *deadlock* no sistema, além de um **algoritmo que faça a recuperação** da situação de *deadlock*.

Em um sistema que **não previne**, **não evita** ou **não recupera** situações de *deadlock*, se um *deadlock* 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* (algoritmo do avestruz) não parecer uma abordagem viável para o problema da ocorrência de *deadlocks*, ele é utilizado em vários sistemas operacionais. Em muitos sistemas, ***deadlocks* ocorrem de forma não freqüente**, como por exemplo, uma vez por ano. Assim, é muito mais simples e "barato" usar este método do que os dispendiosos métodos de prevenir, **evitar, detectar e recuperar situações de *deadlock*.**

Além disso, podem existir situações em que um sistema fica aparentemente **"congelado"** sem estar, no entanto, em situação de *deadlock*. Exemplos dessa situação podem ser um sistema executando um processo em tempo real com a máxima prioridade (o processo de alta prioridade nunca deixará que outros processos assumam o controle do processador), ou ainda, um processo executando em um sistema com escalonador não preemptivo (o processo que assume o controle do processador só liberará o processador quando terminar sua execução; se esse processo for muito grande, os outros processos esperarão por um bom tempo e o sistema parecerá "travado").

Atividades de avaliação

1. O que é um processo?
2. Por que processos são fundamentais para sistemas operacionais multiprogramáveis?

3. O processo é composto por quais partes?

4. Descreva os estados de um processo.

5. O que é um ambiente monothread e ambiente multithread?

6. Quais as vantagens do ambiente multithread?

7. Explique como em ambientes com múltiplos processadores o uso de threads melhora o desempenho de aplicações paralelas.

8. Como a diferença entre a modalidade de kernel e a modalidade de usuário funciona como um tipo rudimentar de sistema de proteção (segurança)?

9. Alguns computadores antigos protegiam o sistema operacional inserindo-o em uma partição da memória que não podia ser modificada pelo job (tarefa) do usuário ou pelo próprio sistema operacional. Descreva duas dificuldades que você acha que poderiam surgir nesse esquema.

10. Qual é a finalidade das chamadas de sistema?

11. Quais são as cinco principais atividades de um sistema operacional relacionadas ao gerenciamento de processos?

12. Liste cinco serviços fornecidos por um sistema operacional e explique por que cada um deles é conveniente para os usuários. Em que casos seria impossível programas de nível de usuário fornecerem esses serviços? Explique sua resposta.

16. Explique a diferença entre escalonamento com e sem preempção.

17. Suponhamos que os processos a seguir chegassem para execução nos momentos indicados. Cada processo será executado durante o período de tempo listado. Ao responder às perguntas, use o escalonamento sem preempção e baseie todas as decisões nas informações disponíveis no momento em que a decisão tiver de ser tomada.

Processo	Tempo de Chegada	Duração do Pico
P1	0,0	8
P2	0,4	4
P3	1,0	1

Qual é o tempo médio de turnaround desses processos com o algoritmo de scheduling FCFS?

a) Qual é o tempo médio de turnaround desses processos com o algoritmo de escalonamento SJF?

b) O algoritmo SJF deveria melhorar o desempenho, mas observe que optamos por executar o processo *P*1 no momento 0 porque não sabíamos que dois processos mais curtos estavam para chegar. Calcule qual será o tempo médio de turnaround se a UCP for deixada ociosa durante a primeira unidade de tempo 1 para então o scheduling SJF ser usado. Lembre que os processos *P*1 e *P*2 estão esperando durante esse tempo ocioso e, portanto, seu tempo de espera pode aumentar. Esse algoritmo poderia ser chamado de scheduling de conhecimento futuro.

18. Qual a vantagem de termos tamanhos diferentes para o *quantum* de tempo em níveis distintos de um sistema de enfileiramento em vários níveis?

19. Que relação existe (se existir alguma) entre os conjuntos de pares de algoritmos a seguir?

a) Por prioridades e SJF

b) Filas com retroalimentação em vários níveis e FCFS

c) Por prioridades e FCFS

d) RR e SJF

20. Liste três exemplos de deadlocks que não estejam relacionados a um ambiente de sistema de computação.

22. É possível ocorrer um deadlock envolvendo apenas um processo? Explique sua resposta.

Capítulo 3

Gerenciamento de Memória

Objetivos

- Definir conceitos básicos sobre gerenciamento de memória.
- Apresentar e discutir a técnica de swapping.
- Discutir formas de alocação de processos na memória principal.
- Apresentar e discutir a técnica de paginação.

1. Introdução

Para que um programa execute em um computador é necessário que, no momento da execução, ele esteja alocado na memória principal. A memória[2], no entanto, é um recurso limitado e não suporta a alocação, ao mesmo tempo, de todos os programas existentes no computador. Dessa forma, a memória é um importante recurso que deve ser cuidadosamente gerenciado e o responsável por este gerenciamento é o sistema operacional.

Apesar de um computador doméstico atual possuir dezenas ou até mesmo centenas de vezes mais memória que um computador IBM 7094 (o maior computador existente no início dos anos 1960), o tamanho dos programas de computador têm crescido em uma escala tão grande quanto à da quantidade de memória dos computadores.

Além da importância da memória principal[3] para a execução dos programas, outro fator relevante para o correto gerenciamento da memória é que este é um recurso que, apesar da queda vertiginosa do seu preço, ainda é muitas vezes mais caro do que a memória secundária (discos, fitas, etc.).

O componente do sistema operacional responsável pelo gerenciamento da memória é chamado de **gerenciador de memória**. Seu papel consiste em:

- Saber quais partes da memória estão ou não em uso;
- Alocar memória para os processos quando esses necessitam dela para executar e desalocá-la quando deixam de usá-la (por exemplo, ao liberarem uma determinada área de dados ou quando terminam sua execução);

[2] **Memórias voláteis**: são as que requerem energia (computador ligado) para manter a informação armazenada.
Memórias não voláteis: são aquelas que guardam todas as informações mesmo quando não estiverem recebendo energia (computador desligado).

[3] **Memória principal:** Também chamadas de memória real, são memórias que o processador pode endereçar diretamente, sem as quais o computador não pode funcionar. Estas fornecem geralmente uma ponte para as secundárias, mas a sua função principal é a de conter a informação necessária para o processador num determinado momento; esta informação pode ser, por exemplo, os programas em execução. Nesta categoria insere-se a memória RAM (volátil), memória ROM (não volátil), registradores e memórias cache.

- Gerenciar as trocas entre a memória principal e a memória secundária quando a memória principal não é grande o suficiente para conter todos os processos.

Sistemas de gerenciamento de memória podem ser divididos em dois grupos: **aqueles que movem processos entre memória principal e memória secundária durante sua execução (*paginação e swapping*), e aqueles que não o fazem.** Cada abordagem possui vantagens e desvantagens. A escolha do melhor esquema de gerenciamento de memória depende de vários fatores, especialmente do projeto do *hardware* do sistema. Como veremos mais adiante, vários algoritmos para gerenciamento de memória requerem algum suporte do *hardware*.

Saiba Mais

RAM (Randomic Acess Memory - memória de acesso aleatório): Os dados nela armazenados podem ser acessados a partir de qualquer endereço. É utilizada para armazenar os programas e dados no momento da execução;
ROM (Read Only Memory – memória somente de leitura): Permitem o acesso aleatório e são conhecidas pelo fato de o usuário não poder alterar o seu conteúdo.
Registrador: mecanismo, circuito ou dispositivo que efetua o registro de (ou que guarda, registra) um dado ou um evento.
Memória cache: área de armazenamento temporária onde os dados mais frequentemente acessados são armazenados para acesso rápido.
Memória secundária: Memórias que não podem ser endereçadas diretamente, e cuja informação precisa ser carregada em memória principal antes de poder ser tratada pelo processador. Não são estritamente necessárias para a operação do computador. São geralmente não-voláteis, permitindo guardar os dados permanentemente. Incluem-se, nesta categoria, os discos rígidos, CDs, DVDs e disquetes.

2. Conceitos básicos

Como já falado anteriormente, memória é um componente essencial para a operação de sistemas de computação modernos. A memória é um grande vetor de palavras ou *bytes* (o tamanho de uma palavra depende de cada máquina), cada qual com seu próprio endereço. Um programa para ser executado pela UCP deve estar na memória principal, de forma que ela busque na memória as instruções que devem ser executadas, uma a uma.

Quando um programa está em execução várias informações relacionadas a ele ficam armazenadas na UCP em registradores especiais. Um destes registradores é o contador de programa (*program counter*, que no caso dos microprocesadores Intel x86, é chamado de IP – *Instruction Pointer*, ou seja,

ponteiro de instruções), que informa à UCP qual a próxima instrução a ser executada. Um ciclo típico de execução de uma instrução no processador consiste nos seguintes passos:

1. Uma instrução é carregada da memória para o processador;
2. A instrução é decodificada. Dependendo da instrução, pode ser necessário carregar da memória para o processador também alguns operandos. Por exemplo, se a instrução for para somar dois números, os valores dos números devem ser levados da memória para o processador, pois a execução da instrução se dará no processador;
3. A instrução é executada;
4. Em alguns casos pode ser necessário o armazenamento em memória do resultado da execução da instrução. Voltando ao exemplo da soma, ao final da execução desta instrução, deve-se armazenar na memória o resultado do cálculo.

A unidade de memória da máquina **"enxerga" apenas seqüências de endereços de memória.** Ou seja, ela não conhece o conteúdo que está armazenado em cada posição da memória, se são instruções ou dados, pois para ela o que importa é apenas o gerenciamento daquele espaço de memória, que é feito através do seu endereço.

2.1. Ligação de endereços (*address binding*)

Para que um programa possa ser executado ele é trazido para a memória e passa a ser tratado como processo. Normalmente existirá um grupo de processos em disco esperando para serem trazidos para a memória para execução. Esse grupo é chamado fila de entrada.

Quando o processo selecionado na fila entra em execução, ele acessa instruções e dados da memória e, ao terminar, seu espaço de memória é declarado como disponível.

RAM (Randomic Acess Memory - memória de acesso aleatório): Os dados nela armazenados podem ser acessados a partir de qualquer endereço. É utilizada para armazenar os programas e dados no momento da execução;
ROM (Read Only Memory – memória somente de leitura): Permitem o acesso aleatório e são conhecidas pelo fato de o usuário não poder alterar o seu conteúdo.
Registrador: mecanismo, circuito ou dispositivo que efetua o registro de (ou que guarda, registra) um dado ou um evento.
Memória cache: área de armazenamento temporária onde os dados mais frequentemente acessados são armazenados para acesso rápido.
Memória secundária: Memórias que não podem ser endereçadas diretamente, e cuja informação precisa ser carregada em memória principal antes de poder ser tratada pelo processador. Não são estritamente necessárias para a operação do computador. São geralmente não-voláteis, permitindo guardar os dados permanentemente. Incluem-se, nesta categoria, os discos rígidos, CDs, DVDs e disquetes.

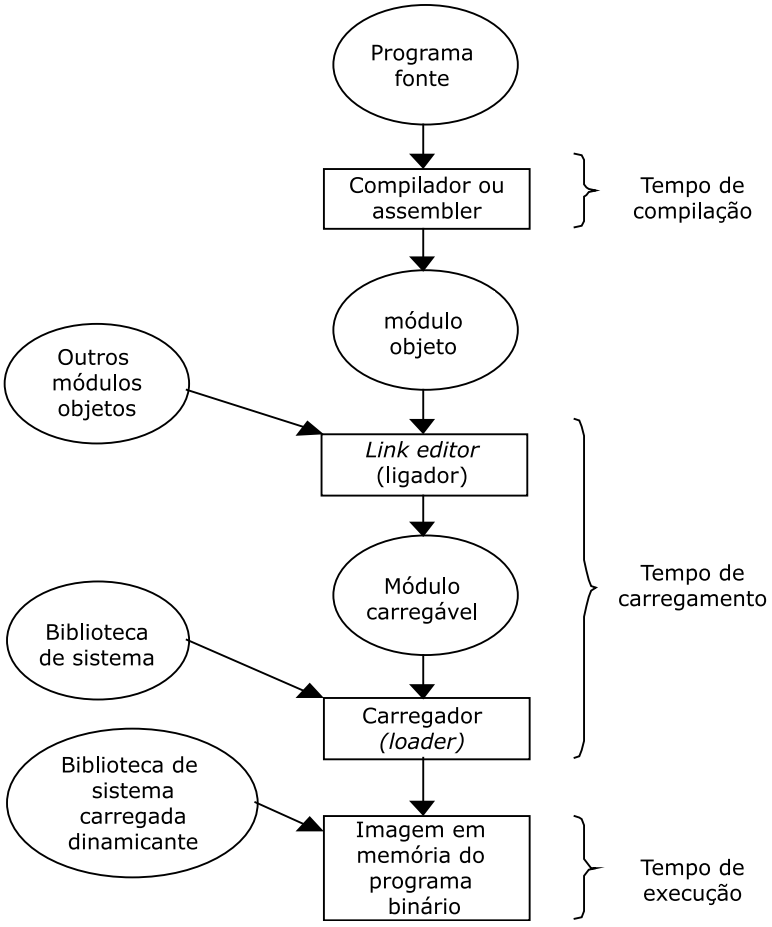


Figura 13 - Passos no processamento de um programa de usuário.

Fonte: Silberschatz, Galvin, Gagne (2010, p. 167).

Na maioria dos casos um programa de usuário passará por várias etapas antes de ser propriamente executado. A Fig. 13 apresenta essas etapas, as quais são descritas abaixo:

- **Tempo de Compilação:** Se, em tempo de compilação, é possível saber onde o programa residirá em memória, então o **código absoluto** pode ser gerado. Por exemplo, se é sabido *a priori* que um processo de usuário reside começando na localização R, então o código gerado pelo compilador irá iniciar naquela localização e continuar a partir dali. Se, mais tarde, a localização de início mudar, então será necessário recompilar este código. Os programas em formato .COM do MS-DOS são códigos ligados absolutamente a endereços de memória em tempo de compilação;
- **Tempo de Carregamento:** Se, em tempo de compilação, não é sabido onde o processo residirá em memória, então o compilador deve gerar **código relocável.** Neste caso, a ligação final a endereços de memória é adiada até o momento de carga do programa. Se o endereço de início

mudar, será necessário somente recarregar o código para refletir a mudança do endereço inicial;

- **Tempo de Execução:** Se o processo pode ser movido durante sua execução de um segmento de memória para outro, então a ligação a endereços de memória deve ser adiada até o momento da execução do programa.

2.2. Carregamento dinâmico (*dynamic loading*)

No carregamento dinâmico, uma rotina não é carregada em memória até que seja chamada. Isso otimiza o uso do espaço da memória na UCP. Esse esquema ocorre da seguinte forma:

1. O programa principal é carregado em memória e executado;
2. Quando a rotina A precisa chamar a rotina B, a rotina A primeiro verifica se a rotina B já está carregada;
3. Se não está, o carregador-ligador relocável é chamado para carregar a rotina B em memória, e para atualizar as tabelas de endereços do programa para refletir esta mudança;
4. O controle é passado para a rotina B recém carregada.

A vantagem do carregamento dinâmico é que uma rotina não usada jamais é carregada em memória. Este esquema é particularmente útil quando grandes quantidades de código são necessárias para manipular casos que ocorrem raramente, como rotinas de tratamento de erros, por exemplo. Neste caso, apesar do tamanho total do programa ser grande, a porção efetivamente usada e, portanto carregada pode ser muito menor.

Carregamento dinâmico não requer suporte especial do sistema operacional. É responsabilidade dos usuários projetarem seus programas para tirar vantagem deste esquema. Sistemas operacionais, entretanto, podem ajudar o programador provendo rotinas de bilbioteca que implementam carregamento dinâmico.

Sem carregamento dinâmico, seria impossível a uma aplicação atual razoavelmente pesada, como o Microsoft Word, ser carregado rapidamente. Em algumas versões, quando o Microsoft Word é invocado, uma rotina principal de tamanho pequeno rapidamente é carregada, mostrando uma mensagem do aplicativo para o usuário, enquanto as outras rotinas usadas inicialmente terminam de ser carregadas.

2.3. Ligação dinâmica

A Figura 3.1 também ilustra bibliotecas ligadas dinamicamente. O conceito de ligação é similar ao do carregamento dinâmico. Nesse procedimento a ligação das rotinas é adiada até o tempo de execução das mesmas.

Na ligação dinâmica, um *stub* é incluído na imagem binária do programa para cada referência a uma rotina de biblioteca. Este *stub* é um pequeno código que indica como localizar a rotina de biblioteca apropriada ou como carregar a biblioteca se a rotina ainda não está presente em memória.

O *stub* funciona da seguinte maneira:

1. Quando o *stub* é executado, ele verifica se a rotina A já está em memória;
2. Se a rotina A não está em memória, o programa a carrega;
3. O *stub* substitui a si mesmo pelo endereço da rotina A e em seguida a executa.

Dessa forma, da próxima vez que o trecho de código que referencia a rotina A da biblioteca é executada diretamente, sem custo novamente da ligação dinâmica. Com este esquema, todos os processos que usam uma mesma biblioteca de linguagem executam somente sobre uma cópia do código da biblioteca.

Este recurso pode ser útil também para atualizações de versões de bibliotecas, como o conserto de *bugs* (defeitos). Uma biblioteca pode ser substituída pela nova versão. Sem ligação dinâmica, todos os programas que usassem tais bibliotecas precisariam ser religados para funcionarem com a nova versão da biblioteca.

Para que programas não executem acidentalmente versões mais recentes e incompatíveis de bibliotecas, uma informação sobre versão da biblioteca é incluída tanto no programa quanto na biblioteca.

A ligação dinâmica precisa da ajuda do sistema operacional para funcionar. Se os processos em memória são protegidos uns dos outros, então o sistema operacional é a única entidade que pode verificar se a rotina necessária está no espaço de memória de outro processo, e pode permitir que múltiplos processos acessem os mesmos endereços de memória.

3. Endereçamento lógico e endereçamento físico

Um endereço gerado pela UCP é normalmente referido como sendo um endereço lógico, enquanto que um endereço visto pela unidade de memória do computador (isto é, aquele carregado no registrador de endereço de memória do controlador de memória) é normalmente referido como sendo um endereço físico.

Os esquemas de ligação de endereços em tempo de compilação e em tempo de carregamento resultam em um ambiente onde os endereços lógicos e físicos são os mesmos. Entretanto, a ligação de endereços em tempo de execução faz com que endereços lógicos e físicos sejam diferentes. Neste caso, nos referimos normalmente ao endereço lógico como um endereço virtual.

Os termos endereço lógico e endereço virtual são, na verdade, a mesma coisa. O conjunto de todos os endereços lógicos gerados por um programa é chamado de espaço de endereços lógicos; o conjunto dos endereços físicos correspondentes a estes endereços lógicos é chamado de espaço de endereços físicos. Assim, no esquema de ligação de endereços em tempo de execução, os espaços de endereços lógicos e físicos diferem.

O mapeamento em tempo de execução de endereços virtuais para endereços físicos é feito pela unidade de gerenciamento de memória (MMU – Memory Management Unity), que é um dispositivo de hardware. Existem diversas maneiras de fazer este mapeamento, conforme discutiremos mais adiante. Vejamos um esquema de MMU simples (Figura 14).

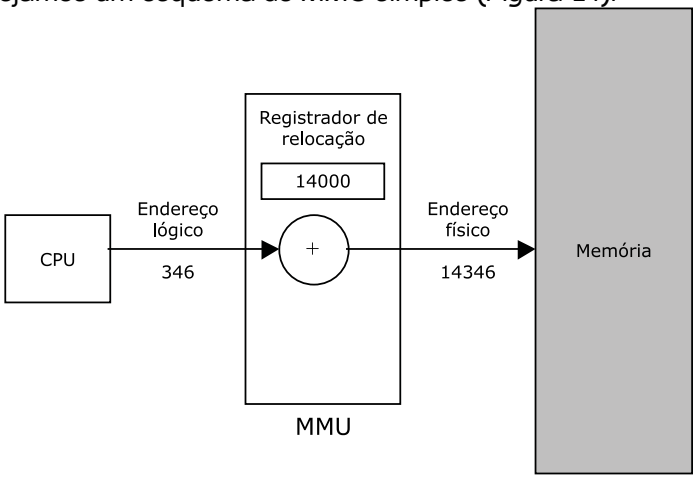


Figura 14 - Relocação dinâmica usando um registrador de relocação.
Fonte: Silberschatz, Galvin, Gagne (2010, p. 167).

Conforme ilustrado na figura, este esquema requer um suporte do *hardware*, particularmente, na MMU. Um registrador de relocação mantém um valor que deve ser somado a qualquer endereço requisitado à memória, gerados por um processo de usuário em execução. Conforme o exemplo, se o endereço base para o processo é 14000, então uma tentativa do processo do usuário acessar o endereço de memória 0 é dinamicamente relocada para o endereço físico 14000, enquanto uma tentativa de acesso ao endereço lógico (virtual) 346 é mapeado para o endereço físico 14346. O sistema operacional MS-DOS, quando executado em um processador Intel 80x86 (uma vez que existem emuladores de MS-DOS para outras máquinas), utiliza quatro registradores de relocação enquanto carrega e executa processos.

Vale observar que o programa do usuário nunca enxerga os reais endereços físicos de memória. O programa em C, por exemplo, pode criar um ponteiro para o endereço 346, armazená-lo em memória, manipulá-lo, compará-lo com outros endereços – sempre como o endereço 346. Somente quando

este valor é usado como um endereço de memória, é que ele será mapeado com relação ao endereço base de memória. O programa do usuário lida com endereços lógicos. O hardware de mapeamento de memória (MMU) converte endereços lógicos em endereços físicos. Isto forma a ligação de endereços em tempo de execução que discutimos nos itens anteriores. A localização final de uma referência a um endereço de memória não é determinada até que a referência seja efetivamente feita.

4. Swapping

Para que um programa seja executado já se sabe que ele deve estar na memória principal do computador e, ao executar, é visto como um processo. Devido à limitação no tamanho da memória e a necessidade eventual de alocar novos processos, no entanto, pode ser que em alguns casos seja necessário retirar um processo da memória e levá-lo para uma área de armazenamento de trocas. Essa operação é conhecida como *swapping* e a área de armazenamento de trocas é denominada área de *swapping*. Quando o processo retirado precisar novamente ser executado, ele é trazido de volta da área de *swapping* para a memória principal.

Suponha, por exemplo, um ambiente multiprogramado com um algoritmo de escalonamento de UCP do tipo *round-robin*. Quanto o *quantum* de determinado processo expira, o gerenciador de memória do SO pode retirar (*swap out*) o processo recém interrompido, e realocar (*swap in*) outro processo no espaço de memória que foi liberado. Enquanto isso, o escalonador de UCP irá alocar uma fatia de tempo para outro processo que esteja na memória. Conforme cada processo tem seu quantum expirado, ele será trocado (swapped) por outro processo que esteja na área de swapping. Em uma situação ideal, o gerenciador de memória conseguirá trocar processos em uma velocidade tal que sempre haja processos em memória, prontos para executar, sempre que o escalonador de UCP decidir colocar outro processo em execução. Para melhorar o desempenho do sistema, o *quantum* dos processos deve ser suficientemente grande para que os processos consigam executar por um tempo razoável antes de serem retirados (*swapped out*) da memória.

Uma variação desta política de *swapping* poderia ser usada para algortimos de escalonamento baseados em prioridades. Se um processo com maior prioridade chega ao sistema e requisita a UCP, então o gerenciador de memória poderia retirar um ou mais processos com prioridades mais baixas de forma que possa ser carregado e executado o processo de prioridade mais alta. Quando o processo de alta prioridade terminasse, os processos de

baixa prioridade poderiam ser trazidos de volta para a memória e continuariam executando. Esta variante de *swapping* é às vezes chamada de roll out, roll in.

Normalmente, um processo que foi retirado da memória será trazido de volta no mesmo espaço de memória que ocupava anteriormente. Esta restrição pode existir ou não conforme o método de ligação de endereços de memória. Se a ligação de endereços de memória é feita em tempo de compilação ou de carregamento, então o processo não pode ser movido para localizações diferentes de memória. **Se a ligação em tempo de execução é usada, então é possível retirar um processo da memória e recolocá-lo em um espaço de memória diferente, uma vez que os endereços físicos são calculados em tempo de execução.**

A técnica de *swapping* requer uma área de armazenamento. Normalmente, este espaço para armazenamento é um disco (ou uma partição de um disco) de alta velocidade. Ele deve ser grande o suficiente para acomodar cópias de todas as imagens de memória (como por exemplo, instruções e dados de processos) para todos os usuários, e deve prover acesso direto a essas imagens de memória. O sistema mantém uma **fila de pronto** consistindo de todos os processos cujas imagens de memória estão na área de armazenamento ou em memória real, e que estão prontos para executar.

Toda vez que o escalonador de UCP decide executar um processo da fila de prontos, ele chama o **despachador**, que verifica se o próximo processo da fila está ou não em memória. Se o processo não está em memória, ele deve primeiramente ser carregado para que possa executar. Se, no entanto, não há uma área de memória livre para carregá-lo, o despachador retira (*swap out*) um processo atualmente em memória e aloca (*swap in*) o processo desejado no espaço liberado. O despachador então restaura o contexto do processo (informações sobre o processo, como conteúdos de registradores, etc.) que será executado, muda seu estado para executando, e transfere o controle da UCP para este processo.

Um fato a ser considerado em um sistema com *swapping* é que o tempo de troca de contexto é razoavelmente alto. Para se ter uma idéia desse tempo, suponhamos que um processo de usuário tenha tamanho de 100KB e que o dispositivo de armazenamento de *swapping* seja um disco rígido padrão com taxa de transferência de 1 MB por segundo. A transferência real dos 100KB do processo da memória para a área de *swapping* ou da área de *swapping* para a memória leva:

100 KB / 1000 KB por segundo = 1/10 segundo = 100 milisegundos

Assumindo que não haja tempo de posicionamento dos cabeçotes do disco rígido, e que haja um tempo médio de latência (tempo até o disco girar

para a posição desejada) de 8 milisegundos, o tempo de troca (*swap*) do processo leva 108 milisegundos. Como é necessária a retirada de um processo para a realocação de outro, e considerando que os dois processos tenham aproximadamente o mesmo tamanho, este tempo deve ser contabilizado em dobro, ficando em torno de 216 milisegundos.

Para o uso eficiente de UCP, é desejável que o tempo de execução para cada processo seja longo em relação ao tempo de *swap*. Dessa forma, em um algoritmo de escalonamento *round-robin*, por exemplo, o tempo do *quantum* deveria ser substancialmente maior do que 0.216 segundos.

Note que a maior parte do tempo de *swap* é o tempo de transferência, que é diretamente proporcional à quantidade de memória trocada (*swapped*). Se um determinado sistema de computação tiver 1 MB de memória principal e um sistema operacional residente de 100 KB, o tamanho máximo para um processo de usuário é de 900 KB. Entretanto, muitos processos de usuários podem ser muito menores que este tamanho, por exemplo, 100 KB cada. Um processo de 100 KB, como já vimos, poderia ser retirado da memória em 108 milisegundos, enquanto que um processo de 900KB necessitaria de 908 milisegundos.

Portanto, seria útil saber exatamente quanta memória um processo de usuário está usando e não simplesmente quanto ele poderia usar. Ou seja, um processo de 900KB que precisa ser alocado na memória, mas que efetivamente usará apenas 200KB, pode solicitar a alocação de espaço de memória de apenas 200KB, excluindo a necessidade de alocação dos 900KB completos, reduzindo assim o tempo de *swap* deste processo. Para que este esquema seja efetivo, o usuário deve manter o sistema informado de quaisquer mudanças relativas às suas necessidades de memória. Dessa forma, um processo com necessidades dinâmicas de memória deverá fazer chamadas de sistema (solicita_memória e libera_memória) para informar o sistema operacional sobre suas mudanças.

Há outras limitações com o método de *swapping* que não serão detalhadas neste material, sendo uma delas relacionada ao estado de um processo que será retirado da memória para a área de *swapping*. Para executar essa operação, deve-se estar certo de que o processo está completamente ocioso, pois uma preocupação em especial é em relação a solicitações de E/S pendentes.

Atualmente, o *swapping* tradicional é usado em poucos sistemas. Ele requer muito tempo para ser executado, prejudicando consideravelmente o tempo disponível para a execução real dos processos, não sendo, dessa forma, uma solução razoável de gerenciamento de memória. Versões modificadas de *swapping*, entretanto, são encontradas em muitos sistemas.

Uma versão modificada de *swapping* é encontrada em muitas versões de UNIX (mais antigas). Normalmente, o swapping é desabilitado, sendo inicializado apenas no momento em que existem muitos processos em execução, e é atingido um certo limiar de quantidade de memória em uso. Se a carga do sistema reduzir, ou seja, se diminuir a quantidade de processos em execução, o swapping é novamente desabilitado.

5. Alocação contígua de memória

A memória principal deve acomodar tanto o sistema operacional como os processos dos usuários. Dessa forma, a memória é usualmente dividida em duas partições, uma para o sistema operacional residente, e outra para os processos dos usuários. É possível que o sistema operacional fique alocado tanto no início da memória, quanto no fim.

O principal fator que afeta esta decisão é a localização do vetor de interrupções, que armazena os endereços das rotinas de tratamento das interrupções. Como em geral o vetor de interrupções está nas posições de memória mais baixas, é comum colocar o SO no início da memória. A figura 15 ilustra possíveis configurações de posicionamento do SO na memória principal.

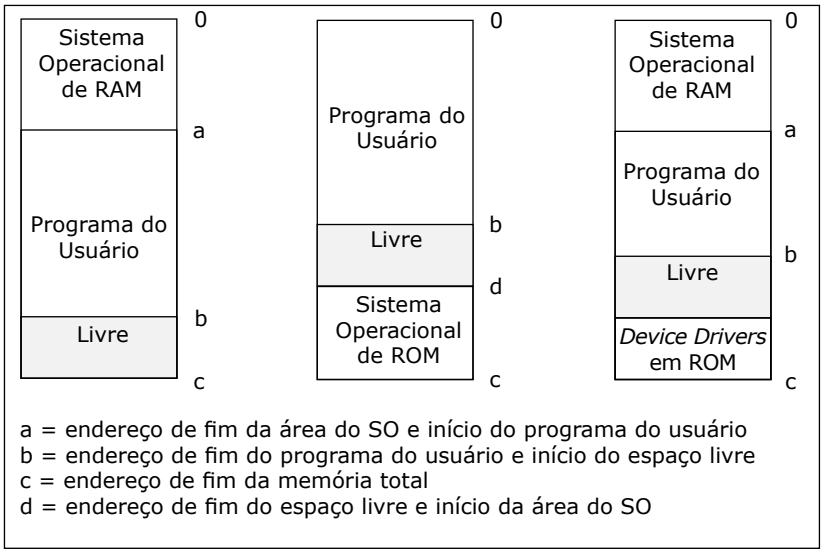


Figura 15 - Três maneiras de organizar a memória.

Fonte: Tanenbaum (2008 p. 350).

O segundo e terceiro esquemas da Figura 15 apresentam algumas rotinas em ROM. Esta é a configuração básica do IBM-PC. Alguns sistemas operacionais mais modernos podem ignorar completamente as rotinas em ROM (BIOS – *Basic Input Output System* – Sistema Básico de Entrada e Saída)

do PC, que são usadas apenas durante o processo de inicialização (*boot)* do sistema operacional.

6. Alocação com partição única

O esquema mais simples de gerenciamento de memória é ter apenas um processo em memória a qualquer momento, e permitir que ele use toda a memória disponível. Os sistemas mais antigos da década de 1960 deixavam a máquina sob controle exclusivo do programa do usuário, que era carregado de uma fita ou de um conjunto de cartões (mais tarde de um disco). Esta abordagem, no entanto, não é mais usada, mesmo nos computadores pessoais mais baratos, porque desse modo apenas um porgrama pode ser executado por vez, não sendo possível a multiprogramação.

Apesar de aparentemente não existirem problemas em sistemas monoprogramados, não se deve esquecer que o usuário tem controle total da memória principal. Devemos lembrar que a memória é dividida entre uma porção contendo as rotinas do SO, outra com o programa do usuário, e outra porção não usada. Dessa forma, se o usuário possui controle total sobre a memória, existe a possibilidade do seu programa escrever sobre as regiões de memória do SO, podendo destruir o SO. Se isso comprometer a execução do programa e fizer com que ele pare, então o usuário perceberá algo errado, corrigirá o programa e tentará novamente. Nessas circunstâncias, a necessidade de proteção para o SO não é tão aparente.

Mas se o programa do usuário destruir o SO de forma que interfira na sua execução, como por exemplo, a mudança de certas rotinas de E/S, então o funcionamento dessas rotinas pode ficar seriamente comprometido. O processo continuará executando, e como algumas vezes o funcionamento errado não é facilmente perceptível, o usuário pode ser levado a crer que a execução do processo esteja correta.

Uma situação ainda mais crítica seria o SO ser modificado de forma que suas rotinas acabem por destruir o sistema como um todo, como no caso de uma rotina de acesso a disco acabar escrevendo em regiões erradas do disco, escrevendo sobre o próprio código do SO no disco, ou ainda destruir informações vitais do disco, como tabelas de alocação, partições, etc.

Está claro, portanto, que o SO deve ser protegido do usuário. Uma forma de realizar essa proteção pode ser simplesmente através de um recurso chamado ***registrador de limite*****,** existente em algumas UCPs, conforme ilustrado na Figura 16.

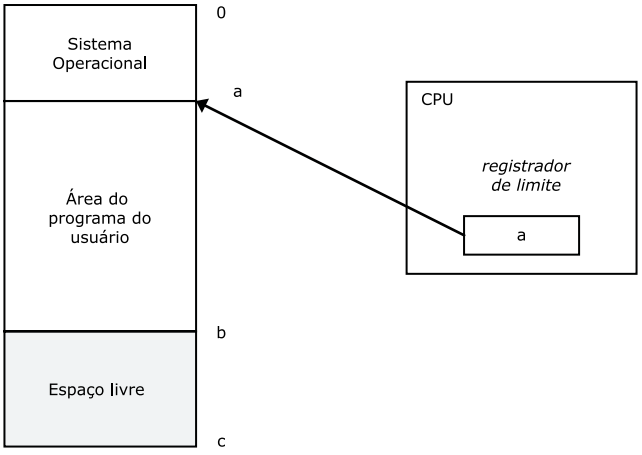


Figura 16 - Proteção de memória em sistemas monousuário usando registrador de limite.
Fonte: Oliveira, Carissimi e Toscani (2004, p. 100).

O registrador de limite contém o endereço da memória mais alta usada pelo SO. Cada vez que um programa do usuário faz referência a um endereço de memória, o registrador de limite é verificado para certificar que o usuário não está prestes a escrever sobre a área de memória do SO. Se o usuário tentar entrar na área de código do SO, a instrução é interceptada e o processo é finalizado com uma mensagem de erro apropriada.

Entretanto, é claro que o programa do usuário eventualmente precisa chamar certas funções que estão no código do SO. Para isso, o usuário usa instruções específicas com a quais ele requisita serviços do SO, que são as chamadas de sistema. Por exemplo, o usuário que deseje ler uma informação do disco deve disparar uma chamada de sistema requisitando que o SO realize essa operação. O SO por sua vez, executará a função desejada, disponibilizando a informação solicitada, e retornará o controle ao programa do usuário.

Conforme os SOs foram ficando mais complexos e tornaram-se multiprogramados, foi necessário implementar mecanismos mais sofisticados de proteção das regiões de memória, de forma que os diversos programas de usuários não interferissem na execução do sistema operacional nem nas execuções uns dos outros.

Esta proteção pode ser implementada através do registrador de limite juntamente com um outro registrador, o registrador de relocação. Assim, o registrador de relocação contém o menor endereço de memória que o processo pode acessar, e o registrador de limite contém a faixa de endereços (quantidade) que o processo pode endereçar a partir do registrador de relocação. Essa situação é mostrada na Figura 17.

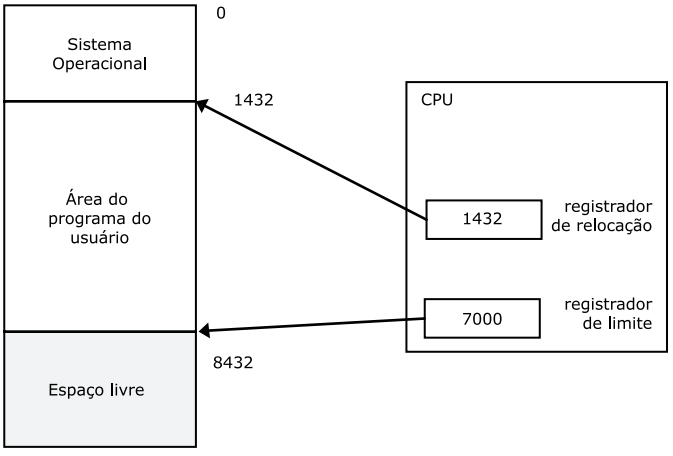


Figura 17 - Suporte do ***hardware*** para registradores de relocação e limite.
Fonte: Oliveira, Carissimi e Toscani (2004, p. 101).

Com este esquema, toda vez que um processo solicita acesso à memória, ele deve passar pela verificação dos dois registradores, conforme a Figura 18.

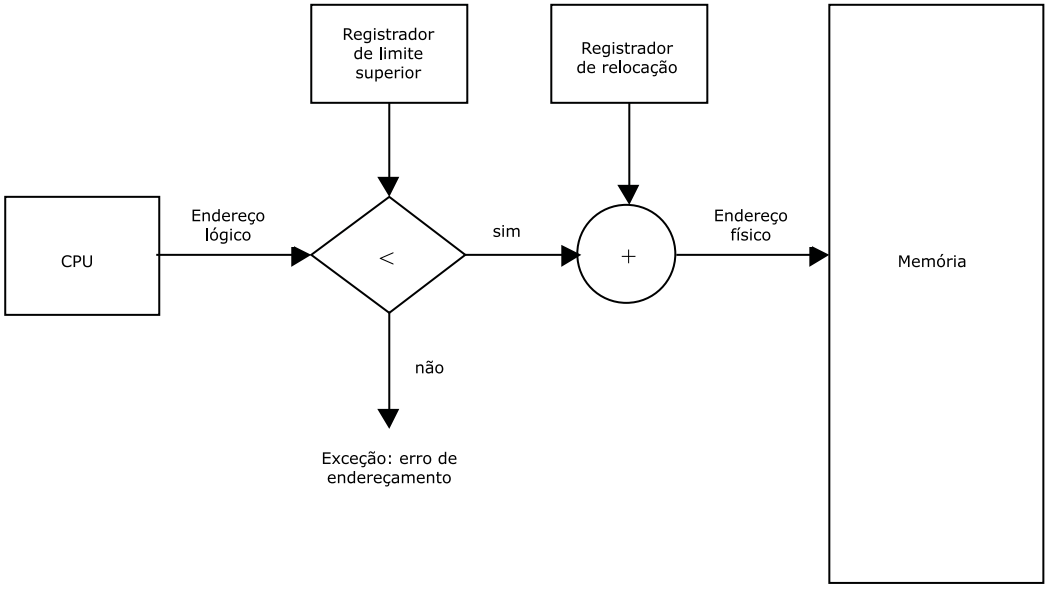


Figura 18 - Funcionameto do ***hardware*** para os registradores de relocação e limite.
Fonte: Silberschatz, Galvin, Gagne (2010, página 166).

Quando o escalonador de UCP seleciona outro processo para execução na fila de prontos, o despachador carrega os registradores de limite superior e de relocação com os valores corretos, da mesma forma que ele o faz

com os demais registradores durante a troca de contexto. Como todo endereço de memória gerado pela UCP é checado com os valores dos registradores, tanto o sistema operacional quanto os processos dos outros usuários estão protegidos das tentativas do processo atualmente em execução de tentar modificar suas respectivas áreas de memória.

Vale notar que o registrador de relocação fornece um esquema eficiente para permitir que o tamanho do SO em memória cresça ou diminua dinamicamente. Esta flexibilidade é desejável em muitas situações. Por exemplo, o SO contém código e *buffers* para os *device drivers*. Se um *device driver* (ou um outro serviço do SO) não é usado com freqüência, é interesante que suas áreas de código e dados sejam liberadas para uso pelos programas dos usuários. Este tipo de código é chamado de **código transiente** (ao contrário de residente), pois ele é carregado e descarregado da memória conforme sua necessidade. Dessa forma, ao carregar um *driver* deste tipo, o sistema operacional muda de tamanho em memória, e o registrador de relocação resolve o problema.

Um exemplo de sistema operacional que possui *drivers* carregados e descarregados da memória dinamicamente é o **Sun Solaris 2.5**. Toda vez que um determinado *driver* (ou até mesmo partes importante do SO, como os protocolos de comunicação de rede) fica certo tempo sem ser usado por nenhum processo de usuário, ele é automaticamente descarregado da memória. Assim que um processo faz alguma chamada que utiliza este *driver*, o SO automaticamente o carrega. Outro sistema operacional que suporta este tipo de *driver* é o **Linux**, que incluiu este recurso a partir da versão de **kernel 2.0.0**.

7. Memória virtual

Nas seções anteriores, foram apresentadas algumas técnicas de gerenciamento de memória que evoluíram no sentido de maximizar o número de processos residentes na memória principal e reduzir o problema da fragmentação[4], porém os esquemas vistos se mostraram muitas vezes ineficientes. Além disso, os tamanhos dos programas e de suas estruturas de dados estavam limitados ao tamanho da memória disponível. Por isso foi desenvolvido a técnica de memória virtual. Na memória virtual a memória principal e secundária são combinadas, dando ao usuário a impressão de que existe muito mais memória do que a capacidade real da memória principal.

O conceito de memória virtual baseia-se em não vincular o endereçamento gerado pelo programa aos endereços físicos da memória principal. Desta forma, o programa e suas estruturas de dados deixam de estar limitados ao tamanho da memória principal física disponível, pois podem

[4] **Fragmentação:** é o desperdício de espaço disponível em memória. Ocorre quando um arquivo ou *fragmento* de arquivo não ocupa completamente o espaço da unidade de alocação destinado a ele, causando desperdício de espaço seja na memória princial seja na memória secundária.

possuir endereços vinculados à memória secundária, que funciona como uma extensão da memória principal. Nessa técnica os programas podem ser divididos em "pedaços" e em vez de levar um programa todo para a memória a cada vez que ele for referenciado, pode-se levar apenas o(s) pedaço(s) que será(ão) utilizados na execução naquele momento.

Uma vantagem da memória virtual é permitir um número maior de processos compartilhando a memória principal, já que apenas partes de cada processo estarão residentes (alocadas na memória principal). Isto leva a uma utilização mais eficiente do processador, além de minimizar (ou quase eliminar) o problema da fragmentação.

Um conceito importante em gerenciamento de memória virtual é o de endereçamento. O conjunto de endereços virtuais que um processo pode referenciar é conhecido como **espaço de endereçamento virtual**, enquanto que o conjunto de endereços reais que um processo pode referenciar é conhecido como **espaço de enredeçamento real**.

O espaço de endereçamento virtual não tem relação alguma com o espaço de endereçamento real. Dessa forma, um programa pode fazer referência a um endereço virtual que esteja fora dos limites da memória principal (real), ou seja, os programas e suas estruturas de dados não estão mais limitados ao tamanho da memória principal física disponível. O processador, no entanto, executa apenas instruções e faz referência a dados que estejam residentes no espaço de endereçamento real. Portanto, deve existir um mecanismo que transforme os endereços virtuais em endereços reais. Este mecanismo é o que chamamos de **mapeamento**, e consiste em permitir a tradução do endereço virtual em endereço real e vice versa.

7.1. Paginação

Uma forma de implementar memória virtual é através da técnica de paginação, onde o espaço de endereçamento virtual e o espaço de endereçamento real são divididos em blocos de mesmo tamanho chamados *páginas*. As páginas do espaço virtual são chamadas *páginas virtuais*, enquanto as páginas do espaço real são chamadas *páginas reais* ou *frames*.

As páginas são as unidades trocadas entre memória principal e memória secundária. Um programa com 20KB de tamanho, por exemplo, pode ser dividido em 5 páginas de 4KB e, em determinado instante, apenas 2 destas páginas estarem alocadas na memória principal.

Um aspecto importante é a definição do tamanho de uma página. Geralmente este tamanho deve ser definido entre 512 bytes e 128KB, aproximadamente. Páginas menores promovem maior compartilhamento da memó-

ria, permitindo que mais programas possam ser executados. Páginas maiores diminuem o grau de compartilhamento da memória, com menos programas disputando o processador. **Assim conclui-se que quanto menor o tamanho da página, maior é o grau de compartilhamento da memória e da UCP.**

O sistema operacional mantém uma estrutura para armazenar, entre outras informações, o mapeamento. Essa estrutura é a tabela de endereçamento de páginas, que relaciona os endereços virtuais do processo às suas posições na memória real, e é única e exclusiva para cada processo,

*S*empre que um processo referencia um endereço virtual, o sistema verifica se a página correspondente já está carregada na memória principal. Se estiver, ele continua sua execução normalmente. Se não, acontece um *page fault* ou falha de página, que deve ser tratado pelo sistema antes que o processo possa continuar sua execução.

No caso de uma falha de página, o sistema deve transferir a página virtual solicitada pelo processo para um endereço na memória principal. Esta transferência é chamada de paginação. O número de falhas de páginas gerados por um processo em um determinado intervalo de tempo é chamado de taxa de paginação do processo. Se esta taxa atingir valores elevados, pode haver um comprometimento do desempenho do sistema. Uma falha de página provoca uma interrupção no processo, pois para buscar a página na memória secundária (disco), há a necessidade de acessar operações de E/S.

Assim, sempre que acontece a paginação, uma interrupção de E/S faz com que o processo em execução seja interrompido e colocado em estado de espera até que sua intervenção de E/S seja realizada, quando então o processo volta à fila de pronto e entra em execução de acordo com o escalonamento realizado pelo sistema operacional. Enquanto o sistema trata a interrupção de falha de página gerada por um processo, outro processo ocupa a UCP.

Políticas de buscas de páginas definem como as páginas serão carregadas da memória virtual para a memória real. A política **por demanda** estabelece que uma página somente seja carregada quando for referenciada. Este mecanismo é conveniente, pois leva para a memória principal somente as páginas realmente necessárias à execução do programa, ficando as outras na memória virtual. A outra política, chamada **paginação antecipada,** funciona carregando antecipadamente várias páginas da memória virtual para a principal, na tentativa de economizar tempo de E/S. Nem sempre o sistema acerta na antecipação, mas o índice de acertos é quase sempre maior que o de erros.

Políticas de alocação de páginas determinam quantos frames cada processo pode manter na memória principal. A política de alocação fixa determina um limite igual para todos os processos, e pode ser vista como uma política

injusta, na medida em que processos maiores normalmente necessitam de um limite maior. A outra política é a variável, que define um limite diferente e variável para cada processo, em função de seu tamanho, taxa de paginação ou até mesmo da taxa de ocupação da memória principal.

O maior problema na gerência de memória virtual por paginação não é decidir quais páginas carregar para a memória real, mas sim quais páginas liberar. Quando há a necessidade de carregar uma nova página e não há mais espaço, o sistema deve selecionar dentre as diversas páginas alocadas na memória principal qual delas deverá ser liberada para dar espaço às páginas solicitadas por um processo e que geraram uma falha de páginas. Para isso, são definidas as políticas de substituição de p*áginas*.

Na política local, somente as páginas do processo que gerou a falha de página são candidatas a serem substituídas. Já na política global, todas as páginas alocadas na memória principal são candidatas à substituição, independente do processo que gerou a falha de página. Na política global, como uma página de qualquer processo pode ser escolhida, pode ser que o processo cujas páginas foram escolhidas para sair da memória sofra um aumento temporário da taxa de paginação em função da diminuição das suas páginas alocadas em memória.

a) Algoritmos de substituição de páginas

Os algoritmos de substituição de páginas têm o objetivo de selecionar os frames (páginas da memória principal) que tenham as menores chances de serem referenciados num futuro próximo, pois se a página que acabou de ser retirada for solicitada logo em seguida será gerada uma nova falha de página, podendo provocar um excesso destas falhas. A seguir são definidos alguns algoritmos de substituição de páginas.

- **Algoritmo Ótimo:** Seleciona a página que não será mais referenciada durante a execução do processo. É impossível de ser implementado, pois o processador não consegue prever com certeza quais são essas páginas, já que o processador desconhece a lógica do programa e os dados que ele manipula;
- **Algoritmo Aleatório:** Escolhe qualquer página, dentre as alocadas na memória principal. Em função de sua baixa eficiência, este algoritmo não é muito utilizado, embora consuma poucos recursos do sistema;
- **Algoritmo FIFO (first in, first out):** Escolhe a página que está há mais tempo na memória principal. É um algoritmo de simples implementação, mas corre o risco de retirar uma página que, embora tenha sido carregada há mais tempo, esteja sendo muito utilizada. Por essa razão não é muito usado;

- **Algoritmo LFU (least frequently used):** Elege a página menos frequentemente usada para efetuar a troca. Através de um contador, armazenado na tabela de endereçamento de páginas, o sistema identifica quantas referências cada página teve e utiliza esta informação para escolher a página;
- **Algoritmo LRU (least recently used):** Elege a página menos recentemente usada para fazer a troca. O sistema mantém na tabela de endereçamento de páginas um campo onde é armazenada a data e a hora da última referência de cada página, e com base nestas informações faz a seleção;
- **Algoritmo NRU (not recently used):** Elege as páginas não recentemente usadas para efetuar a troca. O sistema exclui da decisão a página mais recente e escolhe entre as outras, pelo método FIFO, qual página deve sair.

Atividades de avaliação

1. Descreva as funções básicas da gerência de memória.
2. Descreva os passos de um ciclo típico de execução de uma instrução no processador.
3. Em relação a ligação de endereços o que ocorre no:

a) Tempo de compilação;

b) Tempo de carga;

c) Tempo de execução.

4. Para que servem os registradores de limite e de relocação.
5. Descreva a técnica de *swapping* e explique como essa técnica melhora a eficiência da UCP.
6. Quais as vantagens da memória virtual?
7. Qual a utilidade de uma política de substituição de páginas?
8. Cite duas diferenças entre endereços lógicos e físicos
9. Por que os tamanhos de página são sempre potências de 2?

Capítulo 4

Gerência de Sistemas de Arquivos

Objetivos

- Definir arquivos diretórios e sistemas de alocação de arquivos.
- Apresentar métodos de gerência de espaço livre.
- Apresentar métodos de gerência de alocação de espaço em disco.
- Discutir mecanismos de proteção de acesso.

1. Introdução

O sistema de arquivos é o aspecto mais visível de um sistema operacional. Ele fornece os mecanismos para armazenamento e acesso a dados e programas que pertençam tanto ao sistema operacional quanto aos usuários do sistema de computação. Consiste em duas partes distintas:

- Uma coleção de arquivos, onde cada arquivo armazena dados correlatos;
- Uma estrutura de diretórios (pastas), que organiza e fornece informações sobre todos os arquivos no sistema.

2. Estrutura de diretórios

O Sistema Operacional organiza logicamente os arquivos em diretórios. Cada diretório contém entradas associadas aos arquivos, como as informações de localização, nome, organização e outros atributos.

Em relação à forma como um diretório armazena os arquivos, ele pode ser organizado em um, dois ou mais níveis, conforme apresentado a seguir.

- **Nível Único:** É a implementação mais simples de uma estrutura de diretórios, onde existe um único diretório contendo todos os arquivos do disco. É muito limitado, não permitindo, por exemplo, a criação de arquivos com o mesmo nome (Figura 19).

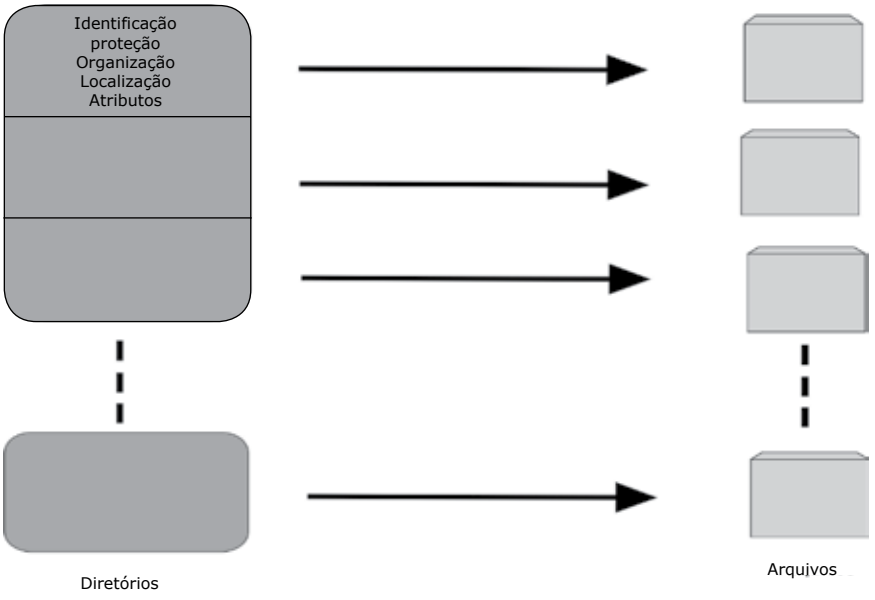


Fig. 19 - Estrutura de diretórios de nível único.

Fonte: Machado e Maia (2007, p. 219).

- **Diretório Pessoal:** Evolução do modelo anterior. Permite a cada usuário ter um "diretório" particular, sem a preocupação de conhecer os outros arquivos do disco. Neste modelo há um diretório "master" que indexa todos os diretórios particulares dos usuários, provendo o acesso a cada um (Figura 20).

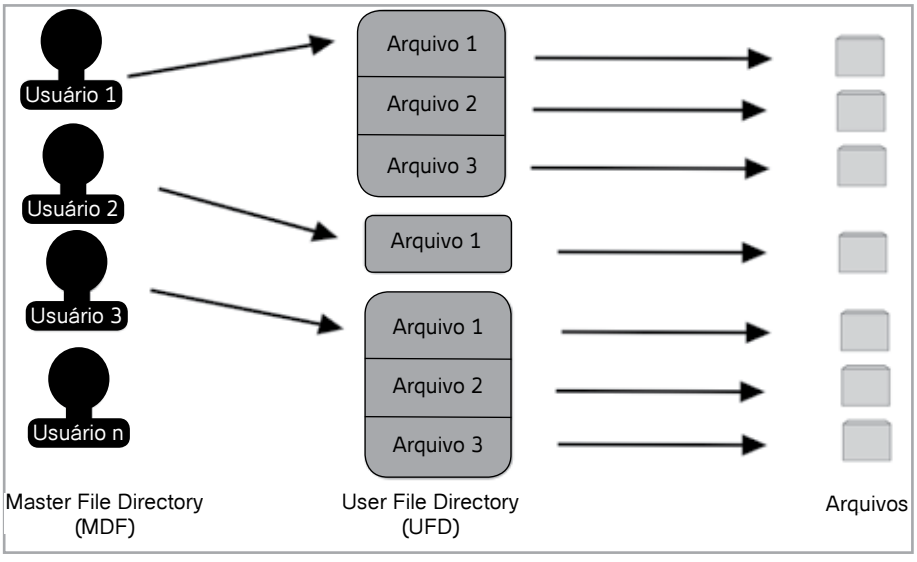


Fig. 20 - Estrutura de diretórios com dois níveis.

Fonte: Machado e Maia (2007, p. 220).

- **Múltiplos Níveis** (ÁRVORE): É o modelo utilizado hoje em dia em quase todos os Sistemas Operacionais. Nesta modalidade cada usuário pode criar vários níveis de diretórios (ou subdiretórios), sendo que cada diretório pode conter arquivos e subdiretórios. O número de níveis possíveis depende do Sistema Operacional (Figura 21).

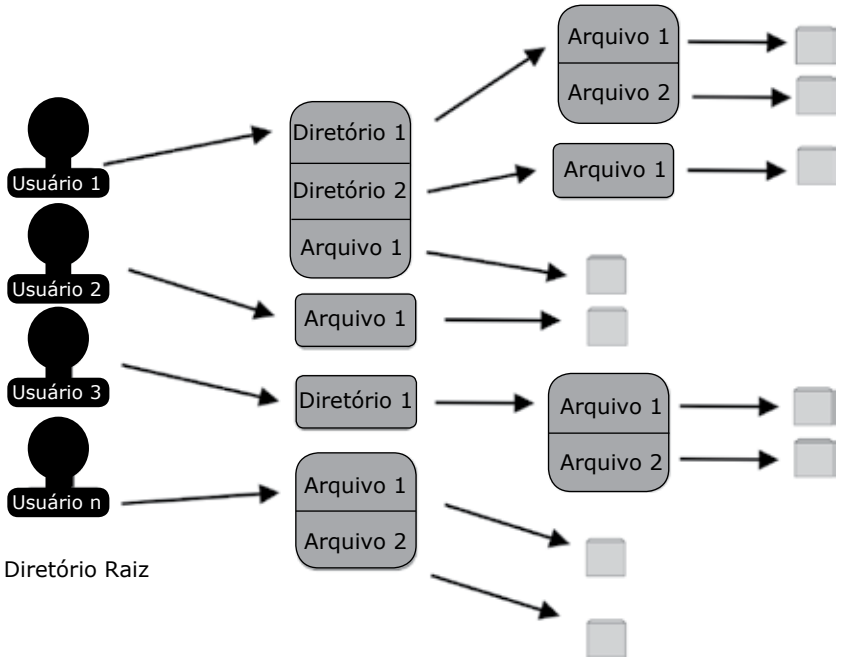


Fig. 21 - Estrutura de diretórios em árvore.
Fonte: Machado e Maia (2007, p. 221).

3. Sistemas de alocação de arquivos

O sistema de alocação de arquivos gerencia o espaço em disco de forma que seja utilizado com eficácia e os arquivos sejam acessados rapidamente. Abaixo temos a descrição de alguns sistemas de alocação de arquivos.

- **FAT:** Sistema criado no MS-DOS e depois utilizado no Windows. Usa listas encadeadas, tem um limite de área utilizável em partições de 2 GB, e caracteriza-se por um baixo desempenho no acesso e armazenamento;
- **FAT32:** Igual ao FAT no que diz respeito à organização e desempenho, mas pode trabalhar com partições de até 2TB;

- **NTFS:** NT *File System*, original da plataforma Windows NT/2000/XP. Opera com uma estrutura em árvore binária, oferecendo alto grau de segurança e desempenho:
  - Nomes de arquivo com até 255 caracteres, podendo conter maiúsculas, minúsculas e espaços em branco;
  - Dispensa ferramentas de recuperação de erros;
  - Bom sistema de proteção de arquivos;
  - Criptografia;
  - Suporta discos de até 2^{64} bytes.
- **UNIX:** Usa um esquema de diretórios hierárquicos, com um **raiz** ao qual outros diretórios estão subordinados. Neste Sistema Operacional todos os arquivos são considerados apenas uma "seqüência" de bytes, sem significado para o Sistema. É responsabilidade da aplicação controlar os métodos de acesso aos arquivos.

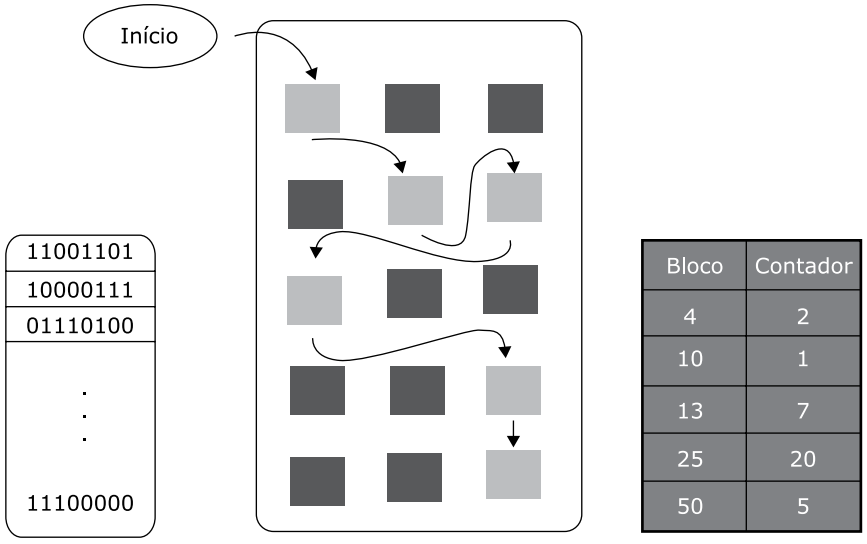
4. Gerência de espaço livre

Uma das tarefas de um Sistema Operacional é armazenar dados em um disco. Para executar essa tarefa, o Sistema Operacional deve gerenciar o espaço livre no disco, de forma que ele saiba quanto de espaço livre existe e quais áreas do disco compõem esse espaço. Existem basicamente três formas principais de executar esse gerenciamento.

A primeira é implementada através de uma tabela denominada **mapa de bits**, onde cada entrada da tabela é associada a um bloco do disco representado por um bit, que estando com valor 0 indica que o espaço está livre, e com valor 1 representa um espaço ocupado**.** Essa técnica gasta muita memória, pois para cada bloco do disco há uma entrada na tabela (Figura 22a)**.**

A segunda forma utiliza uma **lista encadeada dos blocos livres** do disco. Nesse modo, cada bloco possui uma área reservada para armazenar o endereço do próximo bloco livre. Para acessar um determinado bloco é preciso ir para o início da lista e percorrer os blocos, seguindo os endereços (ponteiros), até chegar ao bloco desejado. Apresenta problemas de lentidão no acesso, devido às constantes buscas seqüenciais na lista (Figura 22b).

A terceira forma é denominada tabela de blocos livres. Esta técnica leva em consideração que blocos contíguos de dados geralmente são alocados/liberados simultaneamente. Desta forma, pode-se enxergar o disco como um conjunto de segmentos de blocos livres. Assim, pode-se manter uma tabela com o endereço do primeiro bloco de cada segmento e o número de blocos contíguos que se seguem (Figura 22c).



(a) Mapa de bits (b) Lista encadeada (c) Tabela de blocos livres

Fig. 22 - Alocação de espaço em disco.

Fonte: Machado e Maia (2007, p. 223).

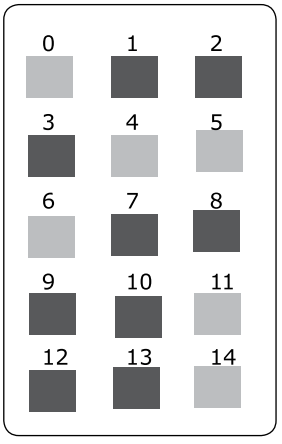
5. Gerência de alocação de espaço em disco

Em relação ao gerenciamento de arquivos, uma outra tarefa do Sistema Operacional é gerenciar o espaço alocado em um disco.

Esse gerenciamento deve levar em consideração a questão da fragmentação do disco. Fragmentos são pequenos espaços livres existentes em várias áreas do disco decorrentes das constantes atividades de inserção e remoção de arquivos. A fragmentação é um tema importante, pois pode afetar o desempenho do sistema de arquivos. Os arquivos podem ser alocados em um disco de três formas: Alocação Contínua, Alocação Encadeada e Alocação Indexada.

- **Alocação Contígua:** Armazena o arquivo em blocos sequenciais. O arquivo é localizado através do endereço do primeiro bloco e a partir desse bloco são localizados os próximos blocos de forma contígua (Figura 23). O principal problema neste tipo de alocação é que para armazenar novos arquivos é necessária a existência de espaço livre contíguo, que nem sempre é possível, pois muitas vezes um disco tem espaço livre que, no entanto, não é contíguo. Por outro lado, havendo espaço contiguo livre, mais de um segmento pode comportar o novo arquivo, sendo necessário decidir em qual segmento o arquivo será armazenado. Para tanto, existem três estratégias de alocação: *first-fit*, *best-fit* e *worst-fit*.

1. ***First-fit***: O primeiro segmento livre com tamanho suficiente para alocar o arquivo é selecionado. A busca na lista de segmentos livres é sequencial, sendo interrompida tão logo se localize um segmento com tamanho adequado para o arquivo;
2. ***Best-fit***: Seleciona o menor segmento livre disponível com tamanho suficiente para armazenar o arquivo. É preciso fazer uma busca em toda a lista de segmentos livres para decidir qual segmento é o melhor, a não ser que a lista esteja ordenada por tamanho. O melhor segmento nesse caso é aquele que deixa menor fragmentação, ou seja, o que melhor se ajusta ao tamanho do arquivo, de forma que depois que o arquivo seja alocado reste o menor espaço possível do segmento (onde o melhor caso é aquele onde o segmento tem o mesmo tamanho do arquivo e não deixa espaço remanescente algum);
3. ***Worst-fit***: Nessa estratégia, o maior segmento é alocado. O objetivo é fazer com que após a alocação do arquivo, restem espaços livres grandes, de forma que facilite encontrar espaços contíguos disponíveis para novos arquivos. Mais uma vez a busca em toda a lista se faz necessária, a menos que exista uma ordenação por tamanho.



Arquivo	Bloco	Extenção
A. TXT B. TXT	4 10	3 1
C. TXT	13	2

Fig. 23 - Alocação contígua.
Fonte: Machado e Maia (2007, p. 224).

- **Alocação Encadeada:** Nesta modalidade o arquivo é organizado como um conjunto de blocos ligados logicamente no disco, independente de sua localização física. Cada bloco possui um espaço reservado para armazenar um ponteiro para o bloco seguinte (Figura 24). Para localizar o arquivo no disco é necessário saber apenas a localização de seu primeiro bloco, pois a partir dele é possível seguir os ponteiros e encontrar os blocos restantes. A fragmentação do disco (pequenos espaços livres

existentes em várias áreas do disco) não representa problemas na alocação encadeada, pois os blocos livres para alocação do arquivo não necessariamente precisam estar contíguos. Um problema que acontece, no entanto, é a quebra do arquivo em vários pedaços, o que aumenta o tempo de acesso a um arquivo. Uma desvantagem desse tipo de alocação é que só é permitido acesso seqüencial aos blocos do arquivo, ou seja, para acessar o bloco 10, por exemplo, é necessário ter acessado antes os blocos de 1 a 9. Outra desvantagem é a perda de espaço nos blocos com o armazenamento dos ponteiros.

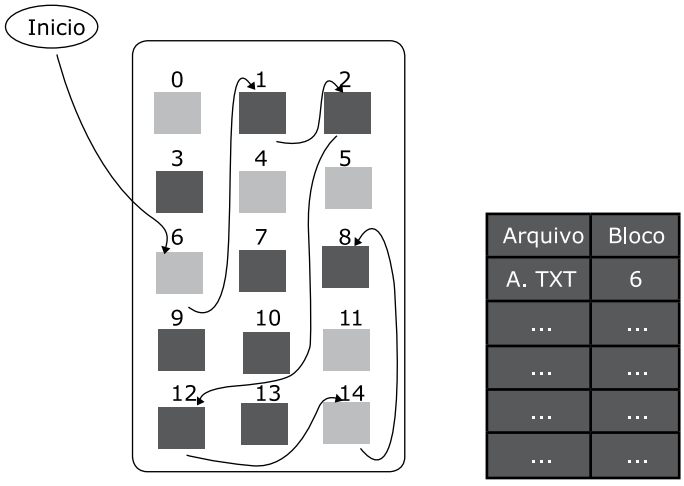


Fig. 24 - Alocação encadeada.
Fonte: Machado e Maia (2007, p. 225).

- **Alocação Indexada:** Esta técnica soluciona a limitação da alocação encadeada no que diz respeito ao acesso, pois permite acesso direto aos blocos de um arquivo. Isso é conseguido mantendo-se os ponteiros de todos os blocos de cada arquivo em uma única estrutura chamada bloco de índices (Figura 25). Este tipo de alocação, além de permitir acesso direto aos blocos, não utiliza informações de controle nos blocos de dados, como os ponteiros existentes na alocação encadeada.

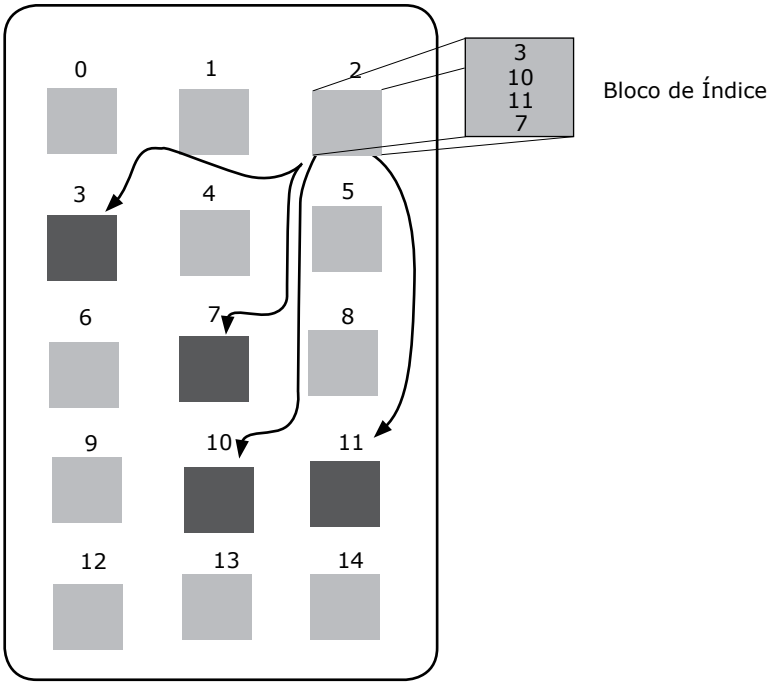


Fig. 25 - Alocação indexada.
Fonte: Machado e Maia (2007, p. 226).

6. Proteção de acesso

Considerando-se que os meios de armazenamento são compartilhados por vários usuários, é fundamental que mecanismos de proteção sejam implementados para garantir a integridade e proteção individual dos arquivos e diretórios. Alguns mecanismos de proteção são apresentados a seguir.

- **Senha de Acesso:** A cada arquivo são associadas senhas de acesso. Esse é um mecanismo de simples implementação, mas que apresenta duas desvantagens. Primeiro, não é possível determinar quais os tipos de operação podem ser efetuadas no arquivo e, segundo, se o arquivo for compartilhado, todos os usuários que o utilizam devem conhecer a senha de acesso;
- **Grupos de Usuários:** Consiste em associar cada usuário a um grupo. Os grupos são organizados logicamente com o objetivo de compartilhar arquivos e diretórios no disco. Este mecanismo implementa três níveis de proteção, OWNER (dono), GROUP (grupo) e ALL (todos). Na criação do arquivo o usuário especifica se o arquivo pode ser acessado somente pelo seu criador, pelo grupo ou por todos os usuários, além de definir que tipos de acesso podem ser realizados (leitura, escrita, execução e eliminação);
- **Lista de Controle de Acesso:** Nesse mecanismo, cada arquivo possui

uma lista associada a ele. Nessa lista são especificados quais os usuários e os tipos de acesso são permitidos ao arquivo. Essa estrutura pode ser bastante extensa se considerarmos que um arquivo pode ser compartilhado por vários usuários. Além deste problema, há o inconveniente de ser necessário fazer acesso seqüencial à lista toda vez que é solicitado o acesso a um arquivo. Em determinados sistemas de arquivos pode-se utilizar uma combinação de proteção por grupos de usuários ou por listas de acesso, oferecendo assim maior flexibilidade ao mecanismo de proteção de arquivos e diretórios.

Atividades de avaliação

1. O que é um arquivo?
2. Descreva as diferentes formas de implementação de uma estrutura de diretórios?
3. Em relação às técnicas para gerência de espaços livres descreva as vantagens e desvantagens de cada uma.
4. O que é alocação contígua de blocos?
5. Em relação às técnicas de alocação encadeada e indexada na gerência de alocação de espaço em disco quais são as vantagens e desvantagens de cada uma.
6. Descreva os tipos de proteção de acesso a arquivos e cite suas principais vantagens.
7. Alguns sistemas suportam muitos tipos de estruturas para os dados de um arquivo enquanto outros suportam simplesmente uma cadeia de bytes. Quais são as vantagens e desvantagens de cada abordagem?
8. Em alguns sistemas, um subdiretório pode ser lido e gravado por um usuário autorizado, do mesmo modo que os arquivos comuns.

a) Descreva os problemas de proteção que podem surgir.

b) Sugira um esquema para lidar com cada um desses problemas de proteção.

9. Pesquisadores têm sugerido que, em vez de termos uma lista de acesso associada a cada arquivo (especificando quais usuários podem acessar o arquivo e como), poderíamos ter uma lista de controle de usuários associada a cada usuário (especificando quais arquivos um usuário pode acessar e como). Discuta os méritos relativos destes dois esquemas.

10. Por que o mapa de bits para alocação de arquivos deve ser mantido em memória de disco e não na memória principal?

11. Considere um sistema que suporte as estratégias de alocação contígua, encadeada e indexada. Que critérios devem ser levados em conta na decisão de qual estratégia é a melhor para um arquivo em particular?

12. Um problema com a alocação contígua é que o usuário deve pré-alocar espaço suficiente para cada arquivo. Se o arquivo crescer a ponto de ficar maior do que o espaço a ele alocado, ações especiais devem ser executadas. Uma solução para este problema é definir uma estrutura de arquivos consistindo de uma área contígua inicial (de um tamanho especificado). Se essa área for preenchida, o sistema operacional definirá automaticamente uma área excedente que será encadeada na área contígua inicial. Se a área excedente for preenchida, outra área excedente será alocada. Compare esta implementação com as implementações contígua e encadeada.

13. Por que é vantajoso para o usuário que um sistema operacional aloque dinamicamente suas tabelas internas? Que desvantagens isso traz para o sistema operacional?

Capítulo 5

Gerência de Dispositivos

Objetivos

- Definir subsistema de entrada e saída e discutir suas funções.
- Apresentar drivers de dispositivos, controladores de entrada/saída e dispositivos de entrada/saída.
- Discutir discos magnéticos e técnicas de redundância e proteção de dados.

1. Introdução

A gerência de dispositivo permite que inúmeros dispositivos físicos com diferentes especificações consigam interagir com o sistema operacional, sem que esse precise ser alterado para cada dispositivo. Isso se dá através do modelo de camadas onde as de nível mais baixo omitem suas características das camadas superiores, tal modelo viabiliza uma interface simplificada e confiável para as aplicações dos usuários (Fig. 26).

Quando um usuário deseja acessar um dispositivo, como por exemplo, um disco, o usuário solicita o acesso ao sistema operacional e o sistema operacional faz o acesso direto ao disco, retornando para o usuário o resultado da solicitação. Por exemplo, se o usuário solicita a leitura de um arquivo em disco, o SO comunica-se com o disco, acessa os dados solicitados e retorna o resultado para o usuário. Devido à interface simples e confiável oferecida pelo SO, o usuário não precisa conhecer detalhes sobre o dispositivo, como por exemplo, no caso do disco, ele não precisa saber informações sobre endereço da trilha e do setor onde estão armazenados os dados que ele quer ler.

O acesso a qualquer dispositivo do sistema é semelhante ao acesso ao disco apresentado. A comunicação do usuário com dispositivos como mouse, impressora, escanner, etc, se dá da mesma forma, através do sistema operacional.

Cada dispositivo, no entanto, tem características particulares. Cada tipo de impressora, por exemplo, tem especificações distintas de funcionamento, bem como uma forma específica de comunicar-se com o SO. É impossível para o SO, no entanto, conhecer as especificações de todas as impressoras existentes no mundo. Dessa forma, cada dispositivo tem associado a ele um pequeno programa desenvolvido pelo fabricante que informa ao SO os detalhes sobre aquele dispositivo, permitindo a comunicação com ele. O nome desse programa é driver de dispositivo ou device driver.

O sistema operacional isola a complexidade dos divesos dispositivos de E/S através da implementação de uma camada, chamada de subsistema de E/S. Isto permite ao SO flexibilidade na comunicação das aplicações com qualquer tipo de periférico, pois a camada de driver de dispositivo ou device driver que trata de parâmetros como velocidade de operação, unidade de transferência, representação dos dados, tipos de operações e demais detalhes de cada periférico oferece uma interface uniforme entre o subsistema de E/S e todos os dispositivos físicos de E/S.

O sistema de gerência de dispositivos divide-se em dois grupos, o primeiro grupo independe do dispositivo e trata os diversos tipos de dispositivos do sistema de um mesmo modo (Fig. 26 a), o segundo é específico para cada dispositivo (Fig. 26 b).

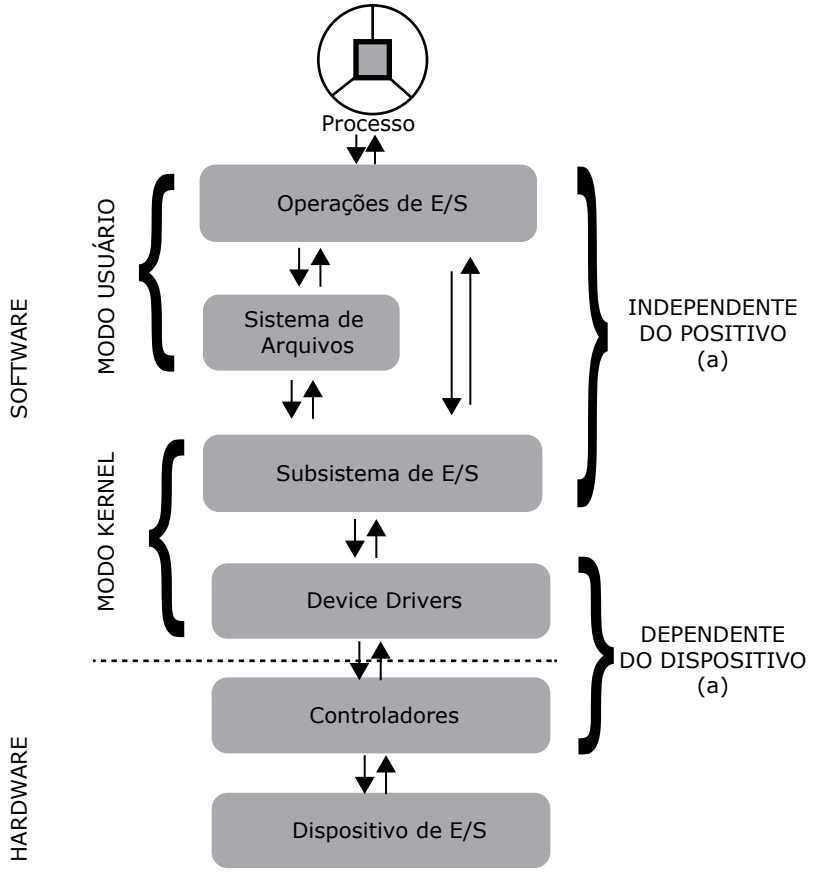


Fig. 26 – Arquitetura de camadas da gerência de dispositivos.

Fonte: Machado e Maia (2004, p. 231).

2. Subsistema de entrada e saída

No intuito de simplificar as operações de E/S para as aplicações do usuário a complexidade de operações específicas de cada dispositivo é isolada pelo subsistema de E/S, isto permite a comunicação com qualquer dispositivo conectado ao computador e as aplicações poderão trabalhar com qualquer periférico de forma mais simples.

Quando se deseja trabalhar com dados de diferentes fontes (CDs, HDs ou fita magnéticas) realizam-se operações de E/S através de chamadas às rotinas de E/S. Isso permite que a aplicação independa das características específicas de cada dispositivo de armanazenamento dos dados.

As diferentes interações da aplicação com o subsistema de E/S é apresentada na Fig. 26. Na forma **explicita** uma chamada direta à rotina de E/S do SO é feita a partir de um programa de usuário. Na foram **implícita** a comunicação entre os comandos de E/S das linguagens de programação e as rotinas de E/S se dá através da passagem de parâmetros. Sendo assim a relação entre o comando e a rotina de E/S é estabelecida na geração do código executável do programa.

Classificação das Operações de E/S:

- **Operação Síncrona:** Quando uma operação é realizada e o processo responsável fica aguardando no estado de espera até o fim dela;
- **Operação Assíncrona:** Quando uma operação é realizada e o processo responsável não aguarda a conclusão dela e contínua pronto para ser executado.

Quando uma operação assíncrona é concluída o sistema avisa ao processo esse estado através de algum mecanismo de sinalização.

No SO o subsistema de E/S é responsável por disponibilizar uma interface uniforme com as camadas superiores, pois os aspectos específicos de cada periférico são tratados pelos drives de dispositivos e os aspectos que são comuns a todos os tipos de periféricos são de responsabilidade do subsistema de E/S. Sendo assim à medida que novos dispositivos de E/S forem instalados o driver referente a esse dispositivo precisa ser adicionado e para isso ser possível o subsistema de E/S precisa ter uma interface padronizada que possibilite a inclusão de novos drivers sem a necessidade de alterar a camada de subsistema de E/S.

Uma unidade lógica de transferência que independe do dispositivo de E/S é criada pelo subsistema de E/S para trasmitir informações de tamanhos diferentes (caracteres ou blocos) as camadas superiores do SO.

Quando ocorre erro nas operações de E/S os mais comuns são tratados nas camadas inferiores do SO, alguns tipos de erros podem ser tratados pelo sistema de arquivos e independem do tipo de dispositivo de E/S. Como exemplo cita-se os erros de gravação em dispositivos de entrada e erros de leitura em dispositivos de saída.

Quando dispositivos de E/S podem ser compartilhados simultaneamente (HD) entre vários usuários o SO precisa garantir a integridade dos dados e quando o dispositivo só pode ter acesso exclusivo (impressora) de um usuário por vez o SO precisa controlar seu compartilhamento.

Aspectos de segurança também são implementados pelo subsistema de E/S, através de um mecanismo de proteção de acesso aos dispositivos de E/S que verifica se o processo solicitante da operação de E/S tem permissão para tal.

Para reduzir o número de operações de E/S o subsistema de E/S usa bufferização, que consiste em carregar do disco para o buffer além do dado solicitado todo o bloco de dados, assim quando uma outra solicitação de leitura de um novo dado pertencente ao bloco anteriormente carregado ocorrer não será necessário uma nova operação de E/S.

3. Drivers de dispositivos

Os drives de dispositivos de E/S juntamente com os controladores viabilizam a comunicação do subsistema de E/S com os dispositvos da seguinte forma:

- Os *drives* traduzem comandos gerais de acesso aos dispositivos de E/S para comandos específicos;
- Os controladores executarão os comandos específicos.

A Fig. 27 mostra que para cada tipo de dispositivo ou grupo de dispositivos semelhantes deve existir um *drive* correspondente para reconhecer as características particulares do funcionamento de cada dispositivo de E/S e traduzi-las para comandos que o controlador possa entender e executar. Funções como a inicialização do dispositivo e seu gerenciamento também são executadas pelos *drives*.

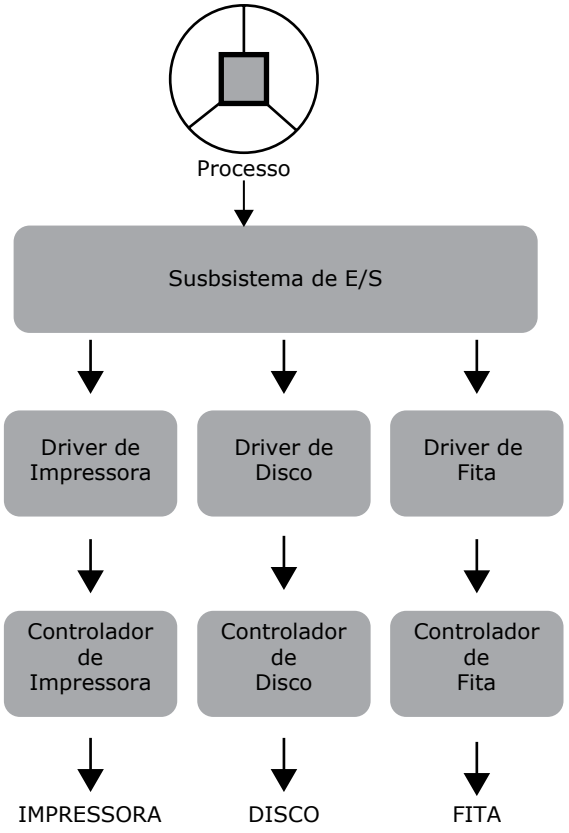


Fig. 27 – Device drivers.

Fonte: Machado e Maia (2007, p. 234).

O processo de leitura síncrona de um dado em disco é apresentado na Fig. 28.

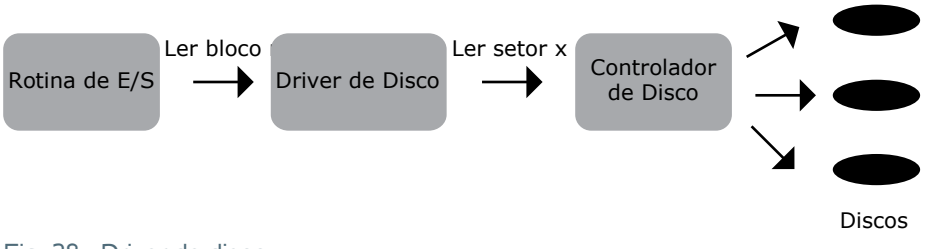


Fig. 28 - Driver de disco.

Fonte: Machado e Maia (2007, p. 234).

1. O *driver* recebe a solicitação de leitura de um bloco;
2. O *driver* informa ao controlador o disco, o cilindro, a trilha e o setor no qual o bloco se encontra, iniciando a operação de leitura;
3. O processo solicitante da operação fica em estado de espera enquanto durar a leitura;
4. Uma interrupção (trap) do controlador avisa a UCP que a operação de leitura foi concluída;
5. A interrupção do controlador também reativa o *drive*;
6. O *drive* verifica se há erros, em caso negativo as informações são trasmitidas a camada superior;
7. Estando os dados disponíveis o processo é retirado do estado de espera retornando ao estado de pronto e poderá proseguir com seu processamento.

Se um drive não foi corretamente desenvolvido o funcionamento do SO pode ser totalmente afetado, pois o drive é um programa de códigos reentrantes executados em modo kernel. Diferentes sistemas operacionais terão diferentes drives associados para o mesmo dispositivo.

4. Controlador de entrada e saída

A Fig. 29 mostra que a comunicação entre o SO e os dispositivos de E/S é feita através dos controladores, os quais possuem memória e registradores próprios que podem estar implementados na mesma placa do processador ou estar em uma placa independente conectada a um slot do computador.

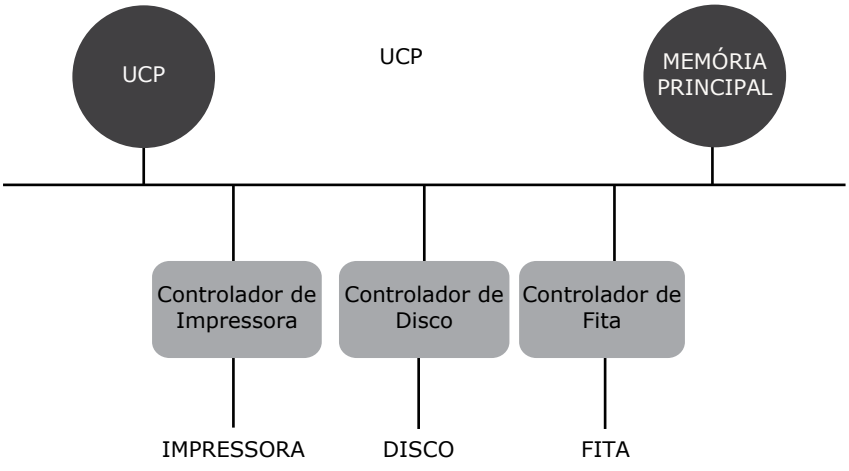


Fig. 29 - UCP, memória e controladores.

Fonte: Machado e Maia (2007, p. 235).

Nas operações de leitura os bits do dispositivos são primeiramente armazenados no buffer interno dos controladores quando formam um bloco são

transferidos para o buffer de E/S da memória principal. Tal transferência pode ser realizada pela UCP ou por um controlador de Acesso Direto à Memória (DMA – Direct Memory Access), o qual pode ser um dispositivo de hardware independente ou que faça parte do controlador.

Com o DMA a operação de leitura em disco seria como descrita na Fig. 30.

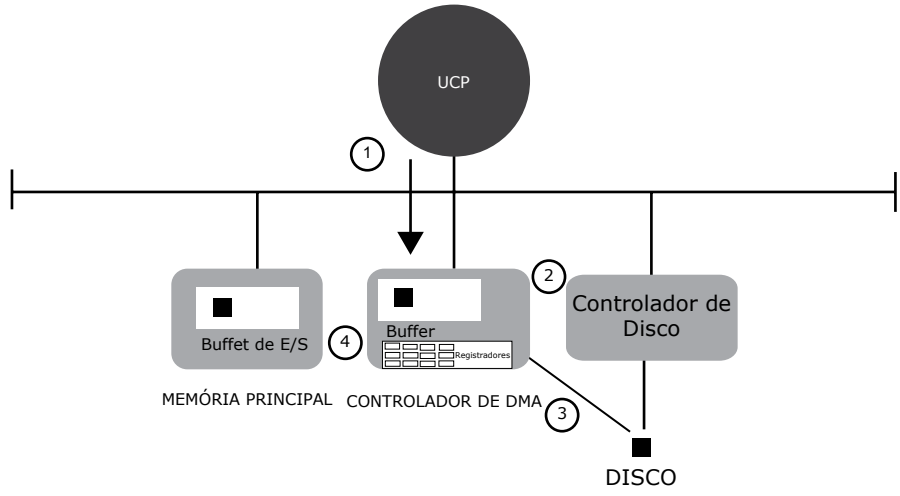


Fig. 30 - Técnica de DMA.

Fonte: Machado e Maia (2007, p. 236).

1. Os registradores do controlador de DMA são inicializados pela UCP através dos *drives*;
2. A UCP é liberada para outras atividades;
3. O controlador de disco atende a solicitação do controlador de DMA e transfere o bloco de bits do disco para o buffer interno do controlador de DMA;
4. Finalizada a transferência acima é feita uma verificação de erros pelo controlador de disco, não havendo erros o controlador de DMA transfere o bloco de bits para o buffer de E/S da memória principal;
5. Para avisar ao processador que o bloco de bits está na memória principal o controlador de DMA sinaliza a este com uma interrupção.

Técnicas de cache como as usadas pelos sistemas de arquivos também são usadas por certos controladores (contoladores de disco) para melhorar o desempenho das operações de E/S. Quando uma operação de gravação em disco é finalizada o controlado avisa ao SO, no entanto para aumentar o desempenho o controlador pode ser configurado para avisar do fim da gravação mesmo quando os dados ainda estão no buffer do controlador e a gravação no disco ainda não foi feita.

O padrão Small Computer Systems Interface (SCSI) de conexão de dispositivos a um computador está presente atualmente em sistemas de computação de todos os portes.

5. Dispositivos de entrada e saída

A comunicação entre o SO e os dados externos a UCP é feita pelos dispositivos de entrada e saída. Tais dispositivos podem ser divididos em:

a) Entrada de Dados: CD-ROM, teclado, mouse, etc...

b) Saída de Dados: Impressoras, caixas de som, monitor, etc...

c) Entrada e Saída de Dados: Modems, discos, CD-RW, etc...

Os dispositivos de entrada e saída classificam-se em:

1. **Dispositivos Estruturados**: As informações são armazenadas em blocos de tamanho fixo. E cada bloco tem um endereço que permite a leitura e gravação totalmente independente dos demais blocos. Os dispositivos estruturados podem ser divididos em :
   - **Dispositivos Estruturados de Acesso Direto**: Quando se recupera um bloco diretamente através de um endereço. Ex: Disco magnético
   - **Dispositivos Estruturados de Acesso Sequêncial**: Quando deve-se percorrer sequencialmente os demais blocos até encontra o bloco desejado. Ex: Fita magnética
2. **Dispositivos Não-estruturados**: Os caracteres enviados ou recebidos não estão organizados em forma de bloco. Isto torna a sequência de caracteres não endereçável e impossibilitando o acesso direto aos dados. Ex: Terminais e impressoras

6. Discos magnéticos

Os discos magnéticos são o repositório principal de dados. Por isso a segurança e o desempenho desse dispositivo são de grande relevância para o sistema computacional como um todo. Eles podem ser enquadrados em duas áreas distintas:

- Memória secundária ou memória de massa;
- Dispositivos periféricos de E/S. Ex.: Hard Disk (HD), disquetes, fitas magnéticas.

Os HDs são constituídos de um prato circular de metal ou plástico, coberto com material (em geral óxido de ferro) que pode ser magnetizado para

representar dados. Os dados são gravados e posteriormente lidos do disco por meio de uma bobina condutora chamada de cabeçote ("head"), conhecida também como cabeça de leitura e gravação. Durante uma operação de leitura ou de escrita, o cabeçote permanece parado enquanto o prato gira embaixo dele.

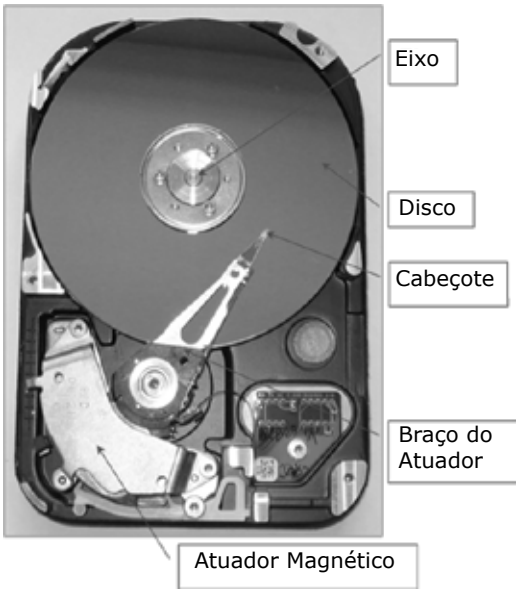


Fig. 31 - HD com um disco.

Fonte: Wikipédia

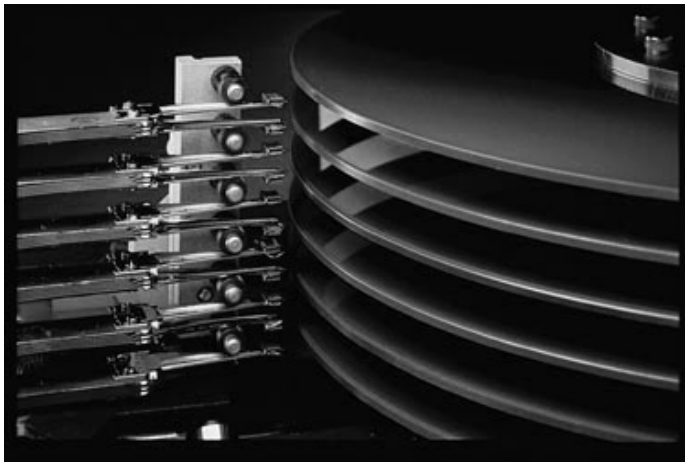


Fig. 32 - HD com vários discos.

Fonte: Wikipédia

Geometria do disco magnético:

- **Faces:** São as superfícies dos pratos. Cada superfície é circular, fina e coberta com uma camada de material magnetizável. Pode haver gravação nas duas faces (superior e inferior) ou só em uma dela, depende do tipo de disco;
- **Trilhas:** São áreas circulares concêntricos onde os dados são gravados. Os dados seqüenciais são gravados em uma mesma trilha. Essas trilhas são numeradas de 0 até N-1;
- **Cilindro:** As trilhas dos diferentes discos que ocupam a mesma posição vertical formam um cilindro. O conceito de cilindro é para minimizar a movimentação da cabeça de leitura/gravação, pois os dados de um arquivo podem ser gravados em um mesmo cilindro, ou seja, em trilhas concêntricas de discos diferentes;
- **Setor ou registro físico:** Cada trilha é dividida em setores, pedaços de trilha. Um setor é também conhecido como registro físico, pois são acessados individualmente nas operações de leitura e gravação, ou seja, a unidade mínima de acesso ao disco é sempre um setor.

> **Nota**: Um único setor de 512 bytes pode parecer pouco, mas é suficiente para armazenar o registro de boot devido ao seu pequeno tamanho. O setor de boot também é conhecido como "trilha MBR", "trilha 0".

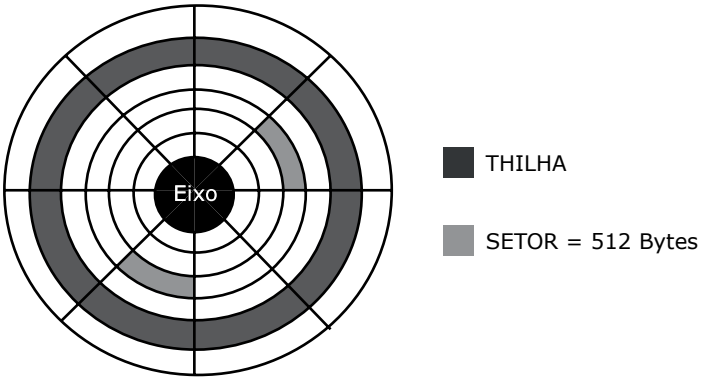


Fig. 5.8. Trilhas e Setores no HD.

Fonte: Wikipédia

Capacidade total do disco

Capacidade = N° de faces x Nº de trilhas x Nº de setores por trilha x 512 bytes

ZBR - Zone bit recording (setorização multizona)

Até o início da década de 90, cada trilha era dividida num mesmo número de setores. Isto, de certa forma, subutilizava a capacidade de armazenamento de um HD, pois o comprimento das trilhas mais externas é maior que as mais internas. Hoje, os discos rígidos usam um esquema chamado setorização multizona, onde os cilindros mais externos possuem mais setores do que os cilindros mais internos do disco, por questões óbvias de diferença no espaço físico disponível. Por isto, um disco rígido que informa ter 63 setores por trilha na realidade não possui esta quantidade, já que a quantidade de setores por trilha é variável.

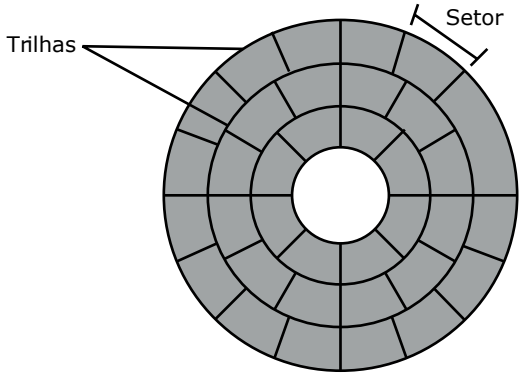


Fig. 34 - Trilhas e Setores ZBR no HD

Fonte: Wikipédia

Como os dados são gravados

Quando estão sendo gravados dados no disco, o cabeçote utiliza seu campo magnético para organizar as moléculas de óxido de ferro da superfície de gravação, fazendo com que os pólos positivos das moléculas fiquem alinhados com o pólo negativo do cabeçote, ou vice-versa.

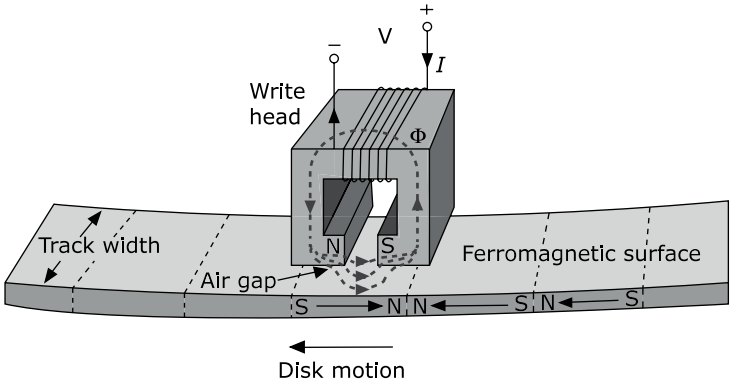


Fig. 35 - Gravação dos dados no HD.

Fonte: Wikipédia

Como o cabeçote do HD é um eletroímã, sua polaridade pode ser alternada constantemente. Com o disco girando continuamente, variando a polaridade do cabeçote, varia-se também a direção dos pólos positivos e negativos das moléculas da superfície magnética. De acordo com a direção dos pólos tem-se bit 1 ou bit 0.

Para gravar as seqüências de bits 1 e 0 que formam os dados, a polaridade da cabeça magnética é mudada alguns milhões de vezes por segundo, sempre seguindo ciclos bem determinados. Quanto maior for a densidade do disco menos moléculas ele utiliza, entretanto, o cabeçote tem que ser mais poderoso.

Tempo de acesso de dados em um HD

O tempo de acesso em um HD é o período gasto entre a ordem de acesso (com o respectivo endereço) e o final da transferência dos dados. Ele se divide em 4 tempos menores:

1. **Tempo de interpretação do comando**: Período gasto para o SO interpretar o comando, passar a ordem para o controlador do disco e este converter no endereço físico;
2. **Tempo de busca (seek):** Tempo necessário para a decodificação do endereço físico (processamento) e o movimento mecânico do braço (posicionamento) para cima da trilha desejada. Por envolver partes mecânicas, é o maior dos tempos que compõem o tempo de acesso de um HD, normalmente gasta em torno de 5 a 10ms. Para a maioria dos discos magnéticos, o tempo de busca é o fator de maior impacto no acesso a seus dados;
3. **Tempo de latência rotacional:** Período entre a chegada do cabeçote sobre a trilha e a passagem do SETOR desejado sobre ele. Depende da velocidade de rotação do disco;
4. **Tempo de transferência:** Nesse período se dá a transferência do bloco de dados entre memória principal e o setor do disco. É quando ocorre a gravação ou leitura dos bits propriamente dito.

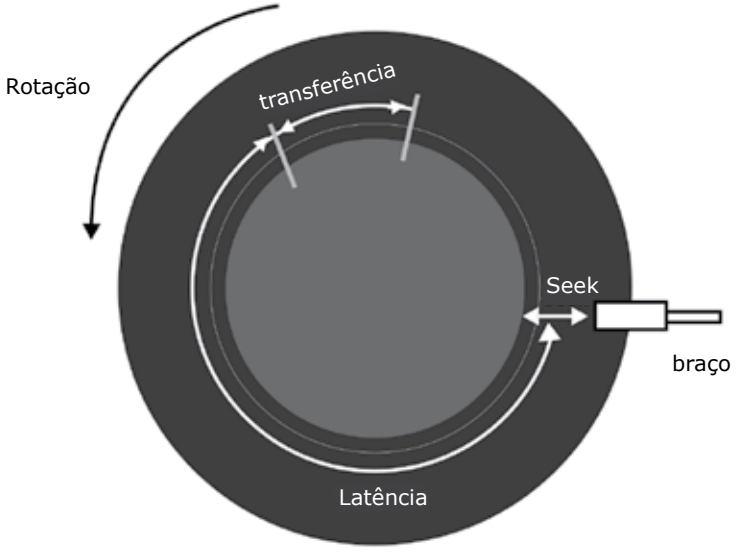


Fig. 36 - Tempo de acesso.
Fonte: Machado e Maia (2007, p. 239).

Nota: Para minimizar os tempos de busca e latência parte dos dados do disco são copiados para a RAM da memória principal.

6.1. Desempenho, redundância e proteção de dados

A tecnologia RAID (Redundant Arrays of Inexpensive Disk – Arranjo Redundante de Discos Independentes) é utilizada para combinar dois ou mais discos (HDs) ou partições em um arranjo formando uma única unidade lógica (matriz) para armazenamento de dados. A idéia básica do RAID consiste em dividir a informação em unidades e, em caso de falha de uma destas, outra unidade assume a que falhou. A forma como os dados são divididos defini o nível do RAID.

Quando se armazena uma informação no RAID ela é dividida em pedaços (stripe) e estes são distribuidos pelos discos que compoem o arranjo. O conjunto dos stripes dos dispositivos forma o chunk do RAID. Quando não é definido o tamanho do chunk, o sistema assume como padrão o valor de 64KB. O chunk para o dispositivo RAID é o mesmo que o bloco para o sistema de arquivo.

Existem dois tipos de RAID:

- **RAID controlado por hardware**: Este tipo de RAID é implementado, principalmente nas controladoras SCSI de disco e são conhecidos como subsistema RAID externo.
- **RAID controlado por software**: Neste tipo de RAID, o arranjo é controlado pelo kernel do sistema operacional ou por um produto gerenciador de discos chamadado de MD (multiple device – dispositivo múltiplo)

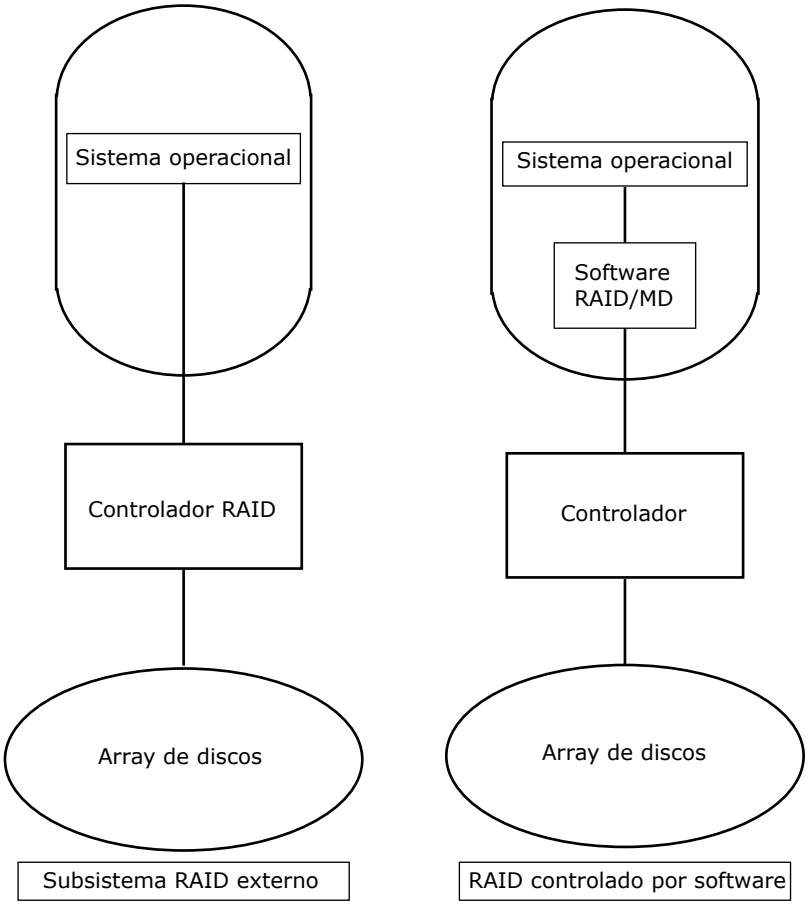


Fig. 37 - Subsistema de discos.
Fonte: Machado e Maia (2007, p. 240).

Nível RAID 0 - Data Striping

No RAID 0, a informação é segmentada e, cada segmento, armazenado em cada unidade do arranjo. Neste nível, não há redundância, pois há somente uma divisão (stripping) da informação. A única vantagem do RAID 0 é o ganho de velocidade no acesso a informação.

Essa técnica é muito usada em aplicações multimídia, pois tais aplicações necessitam de alto desempenho nas operações com discos.

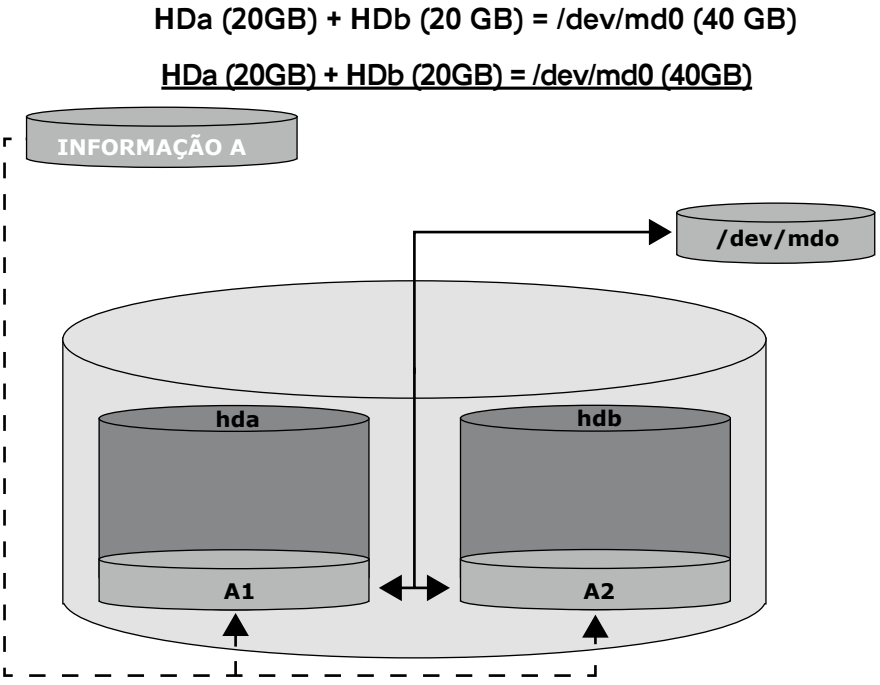


Fig. 38 - RAID 0.

Fonte: Wikipédia

Nível RAID 1 - DATA MIRRORING (Espelhamento de dados)

No RAID 1, a informação é gravada igualmente em todas as unidades (mirror – espelho). À partir deste nível, existe redundância de dados, pois com a duplicidade da informação, quando o disco principal falha, o seu disco-espelho (mirror) assume. A capacidade útil do subsistema de discos com a implementação do RAID 1 é de apenas 50%

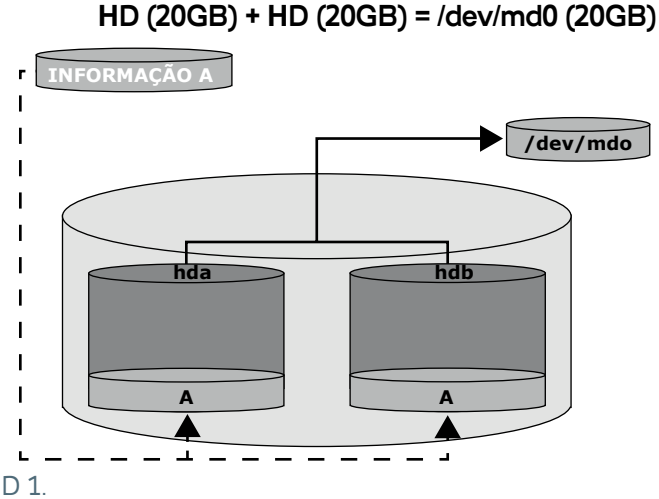


Fig. 39 - RAID 1.

Fonte: Wikipédia

Nível RAID 5 - Acesso Independente com Paridade Distribuída:

No RAID 5, surge uma nova implementação da segmentação da informação e do uso (armazenamento) da paridade. Neste nível, a paridade não é mais armazenada em um único disco dedicado. Um algorítimo é utilizado para segmentar a informação e calcular a paridade. Se um arranjo RAID tem 4 dispositivos, o primeiro bloco da informação será segmentado pelos três primeiros dispositivos e a paridade armazenada no quarto.

No segundo bloco de dados, os dois primeiros segmentos de dados serão armazenados nos dois primeiros dispositivo, o segmento de paridade será armazenado no terceiro dispositivo e o terceiro segmento de dados será gravado no quarto dispositivo. O terceiro bloco de dados segue o mesmo algorítimo (raciocínio). O algorítimo utilizado para a paridade utiliza cerca de 30% de espaço em disco para armazenar a paridade.

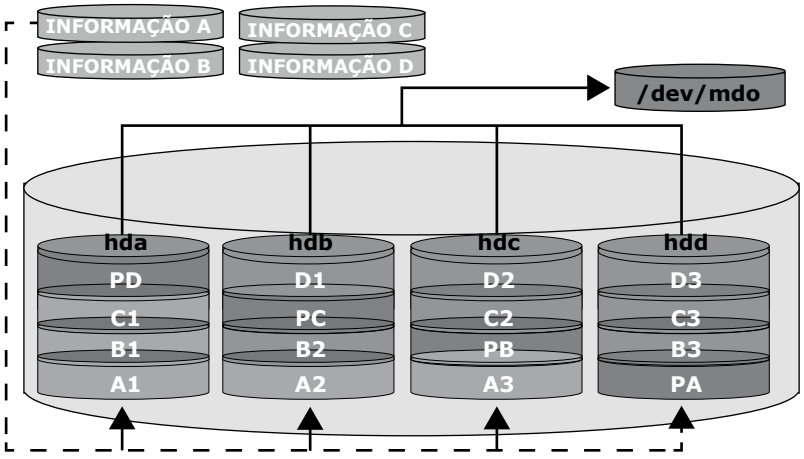


Fig. 40 - RAID 5.

Fonte: Wikipédia

Quando ocorre uma falha em algum dos discos do array, um algoritmo de reconstrução recupera os dados perdidos a partir dos dados de paridade. Essa técnica tem a principal vantagem de usar menos espaço para armazenar inofrmações de controle que a técnica RAID 1. A capacidade útil do subsistema RAID 5 é de aproximadamente 80%.

Atividades de avaliação

1. Para que o sistema operacional implementa a camada de subsistema de E/S?
2. Quais as diferentes maneiras de uma aplicação interagir com o subsistema de E/S?
3. Qual a função do driver de dispositivo?
4. Qual a função dos controladores de entrada e saída?
5. Quais os passos da operação de leitura em disco utilizando o DMA?
6. O que é SCSI (*Small Computer Systems Interface*)?
7. Quais as diferenças entre dispositivos estruturados e dispositivos não estruturados?
8. Descreva quais os fatores que influenciam o tempo de leitura e gravação de um bloco de dados em um disco.
9. Por que a velocidade das operações de E/S em fitas e discos magnéticos são mais lentas que a velocidade de execução de instruções no processador?
10. Descreva as vantagens e desvantagens das técnicas RAID 0, RAID 1, RAID 5.

Referências

MACHADO e MAIA. **Arquitetura de Sistemas Operacionais**, 4ª Edição. Editora: LTC

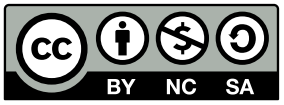
SILBERSCHATZ. GALVIN e GAGNE. **Fundamentos de Sistemas Operacionais** - 8ª Edição. Editora: LTC

TANENBAUM. **Sistemas Operacionais Modernos** – 3ª Edição. Editora: Pearson – Prentice Hall

TANENBAUM e WOODHULL. **Sistemas Operacionais – Projeto e Implementação** – 3ª Edição. Editora: Bookman.

Sobre a autora

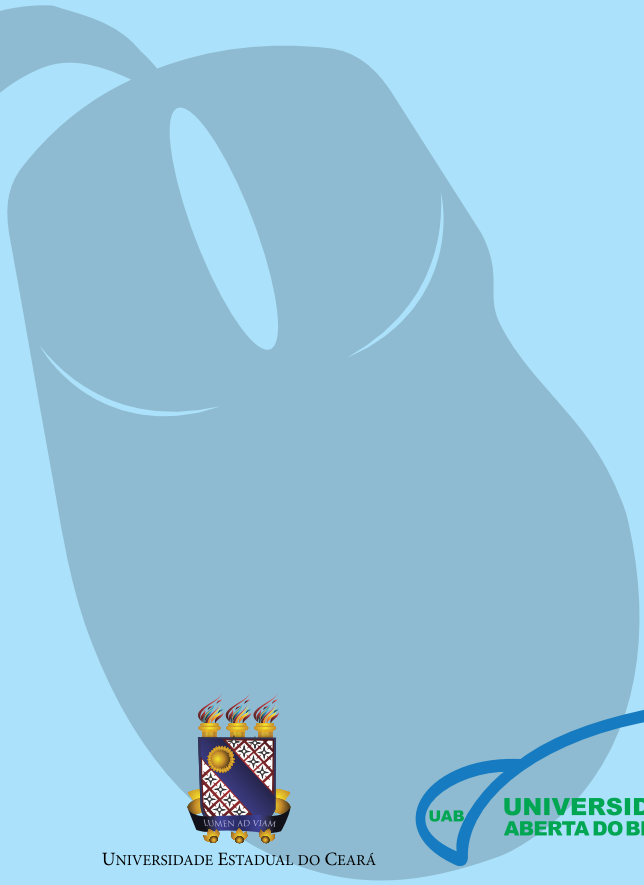
Lorenna Maia Fernandes: Graduada em Engenharia Elétrica, com enfase em Eletrônica, pela Universidade Federal de Campina Grande – PB, Mestre em Engenharia de Teleinformática pela Universidade Federal do Ceará. Atualmente é Doutoranda em Engenharia de Teleinformática na UFC e exerce a docência na Universidade Estadual do Ceará ministrando disciplinas no curso de Ciências da Computação. Já ministrou aulas no Instituto Federal do Ceará e na UFC Virtual.



Computação

Fiel a sua missão de interiorizar o ensino superior no estado Ceará, a UECE, como uma instituição que participa do Sistema Universidade Aberta do Brasil, vem ampliando a oferta de cursos de graduação e pós-graduação na modalidade de educação a distância, e gerando experiências e possibilidades inovadoras com uso das novas plataformas tecnológicas decorrentes da popularização da internet, funcionamento do cinturão digital e massificação dos computadores pessoais.

Comprometida com a formação de professores em todos os níveis e a qualificação dos servidores públicos para bem servir ao Estado, os cursos da UAB/UECE atendem aos padrões de qualidade estabelecidos pelos normativos legais do Governo Federal e se articulam com as demandas de desenvolvimento das regiões do Ceará.



ISBN 978-85-78264-58-1

Universidade Estadual do Ceará

UAB UNIVERSIDADE ABERTA DO BRASIL

CAPES