Introdução aos Sistemas Operacionais - ISO

Técnico em Informática com habilitação em programação e desenvolvimento de sistemas

Notas de aula

Prof. Dr. Marcelo de Paiva Guimarães

Salto, 2010

Prefácio

O objetivo deste guia é servir de referência inicial a alunos do curso de Introdução aos Sistemas Operacionais. Recomenda-se fortemente que os alunos busquem as informações complementares e detalhes nos livros citados na referência bibliográfica. Seu conteúdo é uma pesquisa de vários autores, sendo em partes transcrições e traduções dos mesmos. Esta apostila visa ser uma primeira leitura para os alunos e tenta sempre mostrar os temas abordados de forma simples e clara. Todas as referências bibliográficas utilizadas na construção desta apostila se encontram no final do texto.

Sumário

1 Introdução 1
1.1 Histórico dos Sistemas Operacionais 2
1.1.1 Válvulas e Painéis de Conexão (1945-1955) 2
1.1.2 Transistores e Sistemas de Lote (batch) (1955-1965) 3
1.1.3 Circuitos Integrados e Multiprogramação (1965-1980) 3
1.1.4 Computadores Pessoais e Redes (1980 - 1990) 4
1.2 Tipos de Sistemas Operacionais. 5
1.2.1 Sistema Operacional (Monotarefa) 5
1.2.2 Sistemas Operacionais Multiprogramados (Multitarefa) 5
1.2.3 Sistemas Operacionais Batch(lote) 5
1.2.4 Sistemas Operacionais de tempo compartilhado (time-sharing) 5
1.2.5 Sistemas Operacionais de Tempo Real (*real-time*) 5
1.2.6 Sistemas Operacionais Multiprocessados 6
1.3 Exercícios 6
2 Fundamentos 7
2.1 Sistema de computação 7
2.2 Buffering 7
2.3 Spooling 8
2.4 Interrupções 8
2.4.1 Interrupção de *hardware* 9
2.4.2 Interrupção de *software* 10
2.5 DMA (Acesso direto a memória) 11
2.6 Hierarquia de memória 11
2.7 Proteção de *hardware* 12
2.7.1 Operação em dois modos 13
2.7.2 Operação de E/S 13
2.7.3 Proteção de Memória 13
2.8 Exercícios 14
3 Estruturas dos sistemas operacionais 16
3.1 Componentes dos Sistemas Operacionais 16
3.1.1 Gerência de processos 16
3.1.2 Gerência de memória principal 16
3.1.3 Gerência de arquivos 16
3.1.4 Gerência do sistema de I/O 16
3.1.5 Gerência de armazenamento secundário 17
3.1.6 Proteção de sistema 17
3.1.7 Interpretador de comandos 17
3.2 Estrutura dos Sistemas Operacionais 17
3.2.1 Sistemas Monolíticos 17

3.2.2 Sistemas em Camada 19
3.2.3 Sistemas Cliente-Servidor 19
3.3 Sistemas Monolíticos versus Sistemas Cliente-Servidor 21
3.4 Exercícios 21
4 Processos 22
4.1 Fundamentos 22
4.2 O Núcleo do Sistema Operacional 26
4.3 Escalonamento de Processos 27
4.3.1 Escalonamento FCFS ou FIFO 28
4.3.2 Escalonamento Round Robin 28
4.3.3 Escalonamento com Prioridades 29
4.3.4 Filas Multi-nível com retorno 30
4.3.5 Escalonamento Menor tarefa Primeiro 31
4.4 Exercícios - Revisão 32
4.5 Comunicação e Sincronização entre Processos. 33
4.5.1 O Problema do Produtor e Consumidor sobre um Buffer Circular. 33
4.5.2 Soluções de *hardware*. 35
4.5.3 Soluções de *software* 36
4.5.4 Deadlock: Espera sem fim. 41
4.5.5 Problemas clássicos 41
5 Gerencia de memória 43
5.1 Alocação Contígua Simples 43
5.2 Alocação Particionada 44
5.2.1 Alocação Particionada Estática 44
5.2.2 Alocação Particionada Dinâmica 46
5.2.3 Estratégias para a escolha da Partição 47
5.3 Swapping 48
5.4 Memória Virtual 49
5.4.1 Paginação 50
5.4.2 Segmentação. 54
5.4.3 Segmentação com Paginação: 55
5.5 Exercícios 56
6 Sistemas de arquivos 57
6.2 Exercícios 65
Referências bibliográficas 66

1 INTRODUÇÃO

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

- Gerenciar os recursos de *hardware* de forma que sejam utilizados da melhor forma possível, ou seja, "tirar" o máximo proveito da máquina fazendo com que seus componentes estejam a maior parte do tempo ocupados com tarefas existentes; e
- 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.

É justamente neste segundo item que os sistemas operacionais podem ser bem sucedidos ou não, em despertar interesse para que a indústria de *software* e os programadores independentes construam programas para determinados sistemas operacionais. Isto justifica parte do sucesso do Microsoft Windows, pois, ao mesmo tempo que ele provê uma interface bastante amigável com o usuário, para o programador, não é tão difícil criar um programa com janelas, botões, listas, etc, como seria num sistema operacional como o MS-DOS. Além disso, os sistemas operacionais da Microsoft rodam no *hardware* mais "popular" hoje em dia: os computadores baseados em IBM PC.

Computadores modernos possuem um ou mais processadores, memória principal, dispositivos de entrada e saída como discos, fitas, 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 do sistema operacional: abstrair as particularidades do *hardware* dos programas, fornecendo a eles facilidades para sua operação, tais como: rotinas de acesso a dispositivos diversos; funções de armazenamento de dados como criação de arquivos, leitura e escrita de dados; e rotinas de acesso aos dispositivos de interação com a máquina, como teclado, *mouse*, monitor, etc.

Dada a existência de *softwares* como o sistema operacional, os programas normalmente são classificados como *software* básico (que inclui o sistema operacional), e *softwares* 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.

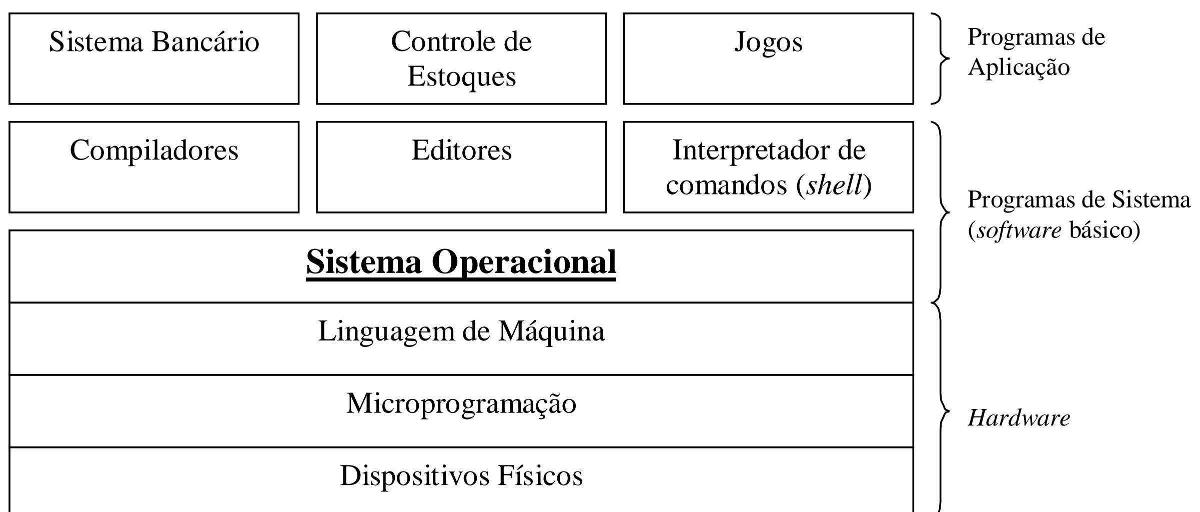


Figura 1 – Integração entre *hardware*, *software* básico e *software* aplicativo

Olhando para o diagrama, veremos que o que chamamos de "*hardware*" é na verdade composto de três camadas. Nem todas as máquinas seguem este esquema, algumas podem ter uma camada 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, e outros dispositivos, 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 formam uma instrução de processador completa, 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. Afinal, de nada adianta uma máquina maravilhosa, se não existir documentação alguma de como ela funciona. 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 sistema operacional.

Acima do sistema operacional estão os demais programas utilizados pelo usuário final, mas alguns deles ainda são considerados *software* básico, como o sistema operacional. Entre eles podemos citar o *shell*, que consiste do interpretador de comandos do usuário, ou seja, a interface com o usuário. 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. A maneira mais comuns de executar programas, copiar e mover arquivos, entre outras atividades mais comuns, é através do uso do *mouse*. Nos tempos do MS-DOS, o teclado era o dispositivo de entrada dominante, por onde o usuário entrava todos os comandos para realizar suas tarefas do dia a dia.

O que é muito importante observar quanto ao *software* básico é que, apesar de que editores (ex: bloco de notas do Windows), compiladores (ex: compilador C no Unix), e interpretadores de comando (ex: command.com ou explorer.exe no Windows) normalmente serem instalados junto como sistema operacional em um computador, eles não são o sistema operacional. Eles apenas utilizam o 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, mas com a finalidade de permitir que os usuários realizem suas tarefas mais freqüentes: executar programas e trabalhar com arquivos.

A grande diferença entre o sistema operacional, e os programas que rodam sobre ele, sejam *software* básico ou *software* aplicativo, é que o sistema operacional roda em modo kernel (ou supervisor), enquanto os demais programas rodam 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 os programas que rodam em modo usuário, tem acesso somente a determinadas regiões de memória, não podem acessar dispositivos diretamente, e precisam pedir para o 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. Isto também possibilita que programas de diferentes usuários estejam rodando na mesma máquina, de forma que um usuário não consiga interferir nos programas de outro.

1.1 Histórico dos Sistemas Operacionais

Para uma melhor idéia do curso de desenvolvimento dos sistemas operacionais atuais, a seguir é apresentada a esquematização da evolução histórica dos mesmos, enfatizando a relação entre a evolução dos S.O. e os avanços em *hardware*.

1.1.1 Válvulas e Painéis de Conexão (1945-1955)

Os primeiros computadores foram implementados através de válvulas a vácuo, consistindo em salas

inteiras de circuito, consumindo energia elétrica suficiente para cidades inteiras. A programação era realizada através de painéis onde as conexões realizadas representavam os 0 e 1 dos códigos binários da linguagem de máquina. Não existia o conceito de sistema operacional, sendo que cada usuário introduzia o seu programa por painéis e aguardava os resultados. A probabilidade de falha do sistema durante a execução de algum programa era altíssima, devido à baixa confiabilidade das válvulas a vácuo.

1.1.2 Transistores e Sistemas de Lote (batch) (1955-1965)

A introdução dos transistores, com a consequente redução de tamanho e consumo e o aumento da confiabilidade permitiu o desenvolvimento dos primeiros sistemas realmente utilizáveis fora dos círculos acadêmicos e governamentais, o que garantiu a a venda comercial dos mesmos. Nesta época sugiu a distinção entre projetistas, construtores, operadores, programadores e pessoal da manutenção.

Entretanto, os computadores ainda eram extremamente grandes e caros, devendo ser acondicionados em grandes salas com ar condicionado e operados por pessoal profissional. Então, o seu uso era restrito à entidades governamentais, grandes corporações e universidades. O processo de execução de uma tarefa (job) era, resumidamente:

i) perfuração de um conjunto de cartões com o programa a ser executado;
ii) o operador pega os cartões e os coloca na leitura. Se o compilador FORTRAN for necessário, ele é colocado (também como um conjunto de cartões) na leitora;
iii) o resultado sai na impressora e é levado pelo operador para um local onde o usuário o recolhe. Este processo, além de lento, desperdiça muito tempo de programação devido ao deslocamento do operador pela sala, buscando conjuntos de cartões a serem utilizados e pela lentidão dos dispositivos de entrada e saída (leitora de cartões e impressora).

Para maximizar a eficiência na utilização dos processadores, e devido ao surgimento das unidades de fita magnética, foi utilizada um novo procedimento:

i) perfuração dos cartões e envio ao operador;
ii) o operador junta os conjuntos de cartões e, com a utilização de um computador mais barato, grava-os em uma fita magnética;
iii) a fita magnética é levada ao processador principal e lida;
iv) os programas da fita são executados e o resultado é gravado em outra fita magnética;
v) esta fita de saída é levada ao computador secundário (mais barato), lida e seu conteúdo impresso em uma impressora comum;
vi) a saída da impressora é entregue aos usuários.

Este processo, denominado OFF-LINE, garantiu uma maior eficiência na utilização do processador principal. Porém aumentou o tempo de resposta do sistema para cada usuário. Este aumento do tempo de resposta do sistema se dá em função de se juntar uma quantidade razoável de conjuntos de cartões para se gravar uma fita. Desta forma, cada usuário, para obter a resposta a seu programa, deve aguardar a execução de diversos outros programas armazenados na mesma fita. Isto fica ainda mais crítico quando um dos programas de uma fita apresenta um tempo de execução muito elevado.

1.1.3 Circuitos Integrados e Multiprogramação (1965-1980)

Com a introdução de circuitos integrados, houve uma grande redução no tamanho e custo dos sistemas, bem com um aumento em sua complexidade e generalidade. Isto permitiu o desenvolvimento de dispositivos de entrada e saída inteligentes, de forma que os próprios se responsabilizam pelo controle da transferência de dados entre eles e a memória principal. Outro desenvolvimento importante foi a introdução dos discos, que permitem um acesso aleatório à informação contida nos mesmos, diferentemente das fitas magnéticas, que somente permitem um acesso aos dados na ordem em que os mesmos estão gravados (note que isto pode ficar transparente através de uma programação cuidadosa, entretanto com alto custo em tempo de execução). Estes foram fatores fundamentais para o sucesso do conceito de multiprogramação, apresentado a seguir. Simultaneamente com a utilização de circuitos integrados, surgiu o conceito de multiprogramação. A idéia provém dos seguintes fatos:

i) durante a execução de programas que realizam alta utilização de cálculos (ex: programas

científicos) todos os dispositivos de entrada e saída permanecem inativos;

ii) durante a execução de programas com alta utilização de entrada e saída (ex: programas comerciais com consultas à base de dados e impressão de relatórios) o processador permanece durante grande porcentagem do tempo aguardando os dispositivos de entrada/saída.

Desta forma, surgiu a idéia de se colocar diversas tarefas (jobs) dentro de alguma "partição" da memória principal e executando simultaneamente de forma que, se alguma tarefa precisa aguardar a transferência de dados para um dispositivo, outra tarefa pode utilizar o processador central neste período.

Outro conceito introduzido foi o de "SPOOL (de *Simultaneous Peripherical Operation On Line*) que corresponde à leitura imediata dos jobs para o disco no momento da sua chegada, sendo que ao terminar um dos jobs ativos, um novo job é imediatamente carregado do disco para a partição de memória vazia e executado (partição é um trecho de memória alocado a um **job**). Este processo tem a vantagem de que, com a leitura simultânea dos dados para um meio de armazenamento mais rápido e com a transferência de dados entre os meios realizada simultaneamente com a operação da unidade de processamento principal, desapareceu praticamente o tempo manual de montagem e desmontagem de fitas. Além disso, dispondo de diversos *jobs* a serem executados no disco, o sistema operacional podia escolher entre eles por prioridade, e não necessariamente por ordem de chegada.

Entretanto, até este ponto, o sistema continuava sendo um sistema de lotes, sendo o tempo entre a apresentação de um conjunto de cartões e a retirada do resultado extremamente alto, principalmente quando se está realizando a depuração de programas.

Para diminuir o tempo de resposta do sistema a um dado job foi introduzido o conceito de compartilhamento de tempo (*time-sharing*), no qual cada usuário possui um terminal ligado em linha com o computador, podendo ainda o computador rodar, no fundo, alguns lotes com a utilização do tempo disponível devido à lentidão de entrada de dados dos usuários. Nesta época também surgiram os minicomputadores, com uma menor capacidade de processamento numérico, mas também com um custo muito menor.

Obs.: Multiprogramação e multiprocessamento: Estes conceitos devem ser claramente distinguidos.

- Multiprogramação: corresponde a diversos programas distintos executando em um mesmo processador.
- Multiprocessamento: corresponde a diversos processadores, dentro de um mesmo sistema de computação, executando programas diversos ou cooperando na execução de um mesmo programa.

Note que foi a existência de multiprocessamento entre os dispositivos de entrada/saída e o processador central que tornou atrativa a introdução da multiprogramação, mas a relação para por aí.

1.1.4 Computadores Pessoais e Redes (1980 - 1990)

Com a integração em larga escala e o surgimento dos microcomputadores, surge também o conceito de *user-friendly* para Sistemas Operacionais, que corresponde ao desenvolvimento de sistemas operacionais para serem utilizados por pessoas sem nenhum conhecimento de computação e que, provavelmente, não têm nenhum interesse em vir a conhecer algo.

Um outro desenvolvimento interessante que foi bastante impulsionado pelos microcomputadores (apesar de não depender dos mesmos) é o de sistemas operacionais para redes de computadores, que consistem em computadores distintos interligados por elementos de comunicação. Os sistemas operacionais para redes são divididos em duas categorias:

- Sistemas operacionais de rede: no qual cada usuário tem conhecimento de seu próprio computador e pode acessar dados em outros computadores;
- Sistemas operacionais distribuídos: em que o sistema operacional faz com que todos os computadores da rede formem uma unidade, de forma que nenhum usuário tenha conhecimento de quantos computadores há na rede ou de em qual (ou quais) computador o seu específico programa está executando.

1.2 Tipos de Sistemas Operacionais.

Os diferentes tipos de sistemas operacionais são basicamente classificados de acordo com o número de processos do usuário que o SO pode executar ou de acordo com o número de processadores que o sistema possui.

1.2.1 Sistema Operacional (Monotarefa)

Possui as seguintes características:
- É executado por um único processador e é capaz de gerenciar a execução de um único programa (tarefa) do usuário por vez.
- Permite que o processador, a memória e os periféricos fiquem dedicados a um único usuário; são portanto monousuários (monoterminais).
- O processador fica ocioso quando o programa espera pela ocorrência de uma E/S.
- São sistemas de simples implementação.

1.2.2 Sistemas Operacionais Multiprogramados (Multitarefa)

Possui as seguintes características:

- É executado por um ou vários processadores. No caso de vários processadores, é classificado como "SO para Multiprocessadores" (discutido a seguir). No caso de apenas um processador, permite que vários programas disputem os recursos do sistema (paralelismo lógico ou virtual), e:

- Podem ser monousuário ou multiusuário:
  - ● Monousuário: um único usuário executando vários programas (monoterminal).
  - ● Multiusuário: vários usuários executando vários programas (multiterminais).
    - Divide o tempo da cpu entre os vários programas e entre os vários usuários.
    - Diminui a ociosidade, permitindo que durante o tempo de E/S outros processos sejam executados.

Inicialmente, os sistemas multiprogramados começaram com os sistemas de batch; depois com os sistemas time-sharing e finalmente com os sistemas real-time.

1.2.3 Sistemas Operacionais Batch(lote)

Os programas eram enfileirados em disco ou fita e aguardavam a execução, um por vez. Normalmente, os programas (jobs) não necessitavam de interação com o usuário. Embora sejam considerados como os precursores dos sistemas multiprogramados, pois aproveitavam os tempos de E/S para a execução de outros processos, o processamento era puramente sequêncial e ofereciam longos tempos de resposta.

1.2.4 Sistemas Operacionais de tempo compartilhado (time-sharing)

Os usuários interagem através de terminais e teclados on-line. São sistemas multiterminais, cujo processamento é controlado por um computador central. O sistema executa uma varredura (*polling*) nos terminais, compartilhando o tempo entre eles (time-sharing).

Neste sistema, o processador executa os programas de forma intercalada no tempo, alocando uma fatia de tempo (*time-slice*) para cada um, por vez, realizando assim a multiprogramação. Cada usuário tem a ilusão que todo o sistema está totalmente dedicado exclusivamente para ele.

1.2.5 Sistemas Operacionais de Tempo Real (*real-time*)

São semelhantes aos sistemas *time-sharing*, embora exijam tempo de resposta dentro de limites rígidos, na execução de tarefas. O conceito de *time-slice* é muito pouco utilizado e os processos executam o tempo necessário e conforme sua prioridade. São sistemas muito utilizados em controle de processos, onde o tempo é um fator crucial: refinaria de petróleo, automação industrial, controle de tráfego aéreo etc.. Neste sistema, os processos geralmente são ativados por sensores.

1.2.6 Sistemas Operacionais Multiprocessados

Possui as seguintes características:

- O sistema possui vários processadores, que podem estar confinados a um mesmo gabinete (centenas de processadores) ou espalhados fisicamente em forma de rede (dezenas de processadores).

- Executam várias tarefas simultaneamente e portanto são multitarefas.

- Cada processador pode operar monoprogramado ou multiprogramado.

- Ocorre paralelismo físico ou real, quando mais de um processador está sendo utilizado. Ocorre também paralelismo lógico, quando o número de tarefas é maior que o número de processadores disponíveis.

- Podem ser fracamente acoplados ou fortemente acoplados:

● Fracamente acoplados (*loosely coupled*): cada processador possui sua própria memória e executa seu próprio sistema operacional (Sistema Operacional de Rede) ou parte de um sistema operacional global (Sistema Operacional Distribuído).

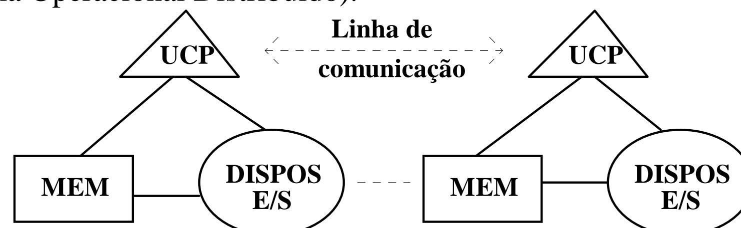


Figura 2 - Sistemas fracamente acoplados

● Fortemente acoplados (*tightly coupled*): todos os processadores compartilham uma única memória. Estes processadores geralmente são coordenados por um único SO localizado em um outro computador hospedeiro, que se encarrega de distribuir as tarefas entre os processadores e gerenciar a execução.

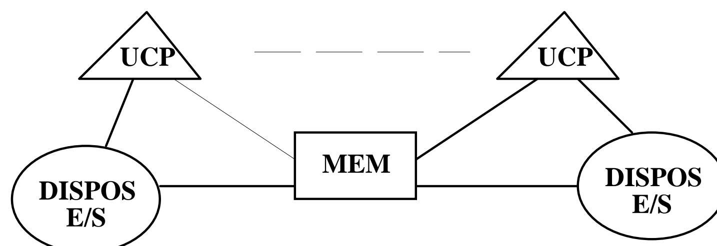


Figura 3- Sistemas fortemente acoplados

1.3 Exercícios

1. Quais os principais objetivos e as principais funções de um SO?
2. Qual a diferença entre monoprogramação e mutiprogramação. Exemplifique.
3. Defina as propriedades essenciais dos seguintes tipos de SO: Batch, Multiprogramados, Tempo Compartilhado, Tempo Real e Multiprocessados.
4. Se o computador possuir apenas um processador, é possível ocorrer um processamento paralelo? Justifique a resposta.
5. Quais são as diferenças entre um sistema fracamente e fortemente acoplado?
6. Explique a diferença entre processamento paralelo e concorrente.

2 FUNDAMENTOS

Este capítulo tem como objetivo introduzir diversos fundamentos que serão utilizados no decorrer do curso.

2.1 Sistema de computação

Um sistema de computação de uso geral moderno consiste de uma CPU e em uma série de controladoras de dispositivos que são conectadas através de um barramento comum que fornece acesso à memória compartilhada (Figura 4). Cada controladora de dispositivo está encarregada de um tipo específico de dispositivo (por exemplo, unidades de disco, dispositivos de áudio e monitores de vídeo). A CPU e as controladoras de dispositivo podem executar de modo concorrente, competindo pelos ciclos de memória. Para garantir acesso correto à memória compartilhada, uma controladora de memória é fornecida e sua função é sincronizar o acesso à memória.

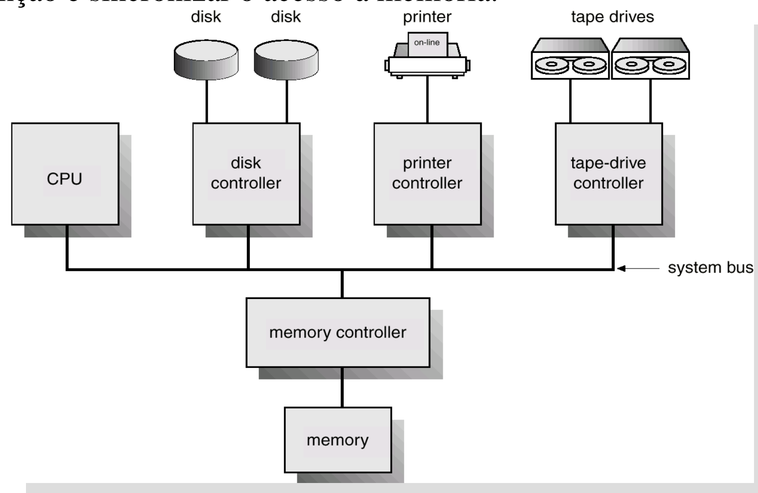
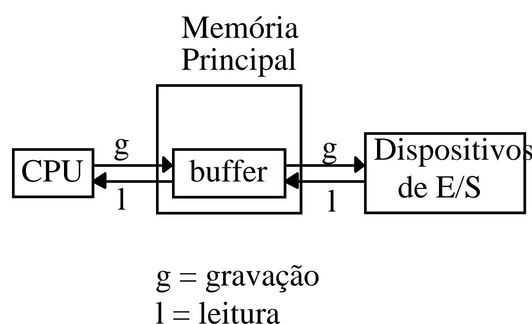


Figura 4 - Sistema de computação moderno

2.2 Buffering

A técnica de *buffering* consiste na utilização de uma área de memória (buffer) para a transferência de dados entre os periféricos e a memória principal. O *buffering* veio permitir que, quando um dado fosse transferido para o *buffer* após uma operação de leitura, o dispositivo de entrada pudesse iniciar uma nova leitura. Neste caso, enquanto a CPU manipula o dado localizado no *buffer*, o dispositivo realiza outra operação de leitura no mesmo instante. O mesmo raciocínio pode ser aplicado para operações de gravação, onde a CPU coloca o dado no buffer para um dispositivo de saída manipular, como mostra a Figura 5.



g = gravação
l = leitura

Figura 5- Buffer

O *buffering* é uma técnica utilizada para minimizar o problema da disparidade da velocidade de processamento existente entre a CPU e os dispositivos de E/S. O objetivo do *buffering* é manter, na maior parte do tempo, a UCP e dispositivos de E/S ocupados.

2.3 Spooling

A técnica de *spooling* foi introduzida nos anos 50 com o processamento *batch* e hoje é utilizada na maioria dos sistemas operacionais. Naquela época, os programas eram submetidos um a um para o processamento, e como a velocidade de operação dos dispositivos de E/S, é muito lenta, a CPU ficava ociosa esperando o carregamento de um programa e dados para a memória, ou esperando pelo término de uma impressão.

A solução foi armazenar os vários programas e seus dados, também chamados de *jobs*, em uma fita magnética e, em seguida, submetê-los a processamento. Dessa forma, a CPU poderia processar os *jobs*, diminuindo o tempo de execução dos jobs e o tempo de transição entre eles. Da mesma forma, em vez de um *job* gravar suas saídas na impressora, poderia direcioná-las para uma fita, que depois seria impressa integralmente. Essa forma de processamento é chamado de *spooling*, e é mostrada na Figura 6.

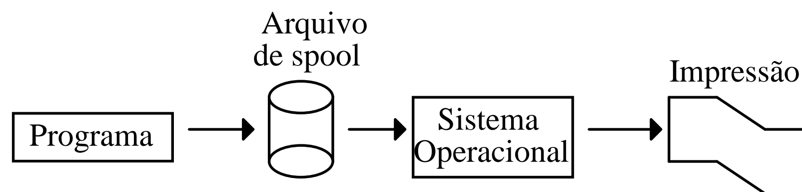


Figura 6- Spool

A técnica de *buffering* permite que um job utilize um *buffer* concorrentemente com um dispositivo de entrada e saída. O *spooling*, basicamente, utiliza o disco como um grande buffer, permitindo que dados sejam lidos e gravados em disco, enquanto outros *jobs* são processados.

2.4 Interrupções

É um sinal informando a um programa que um evento ocorreu. Quando um programa recebe um sinal de interrupção, ele deve tomar uma ação. Existem muito tipos diferentes de sinais que podem disparar uma interrupção, por exemplo, a conclusão de uma opração de I/O, divisão por zero, acesso inválido à memória e um pedido por algum serviço do sistema operacional. Para cada interrupção, uma rotina de serviço é designada para tratar a interrupção.

Quando a CPU é interrompida, ela pára o que está fazendo e imediatamente transfere a execução para um local fixo. Esse local fixo geralmente contém o endereço de início onde está localizada a rotina de serviço para a interrupção. Um diagrama de tempo dessa operação é apresentada na Figura 7.

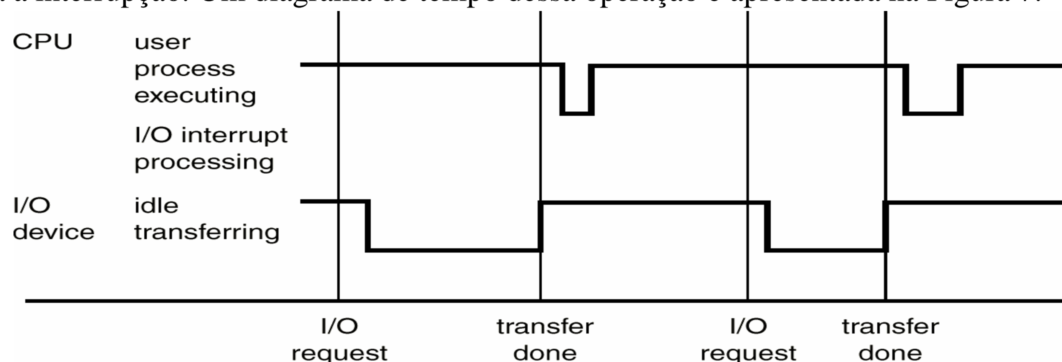


Figura 7 - Diagrama de tempo de interrupção para um único processo gerando saída.

Para começar uma operação de I/O, como mostrada na Figura 7, a CPU carrega os registradores adequados dentro da controladora de dispositivo. A controladora de dispositivo, por usa vez, examina o conteúdo desses registradores para determinar que ação deve ser tomada. Por exemplo, se encontrar um pedido de leitura, a controladora começará a transferir dados do dispositivo para o seu *buffer* local. Uma vez concluída a transferência, a controladora de dispositivo informa a CPU que terminou a operação. Essa comunicação é feita disparando uma interrupção.

Essa situação ocorrerá, em geral, como resultado de um processo de usuário que solicita I/O. Uma vez iniciada a operação de I/O, dois roteiros são possíveis. No caso mais simples, a I/O é iniciada; em seguida, quando tiver sido concluída, o controle é devolvido para o processo do usuário. Esse caso é denominado I/O síncrono. A outra possibilidade, chamada I/O assíncrona, devolve o controle ao programa de usuário sem esperar que a I/O termine. A I/O continua enquanto outras operações de sistema ocorrem Figura 8.

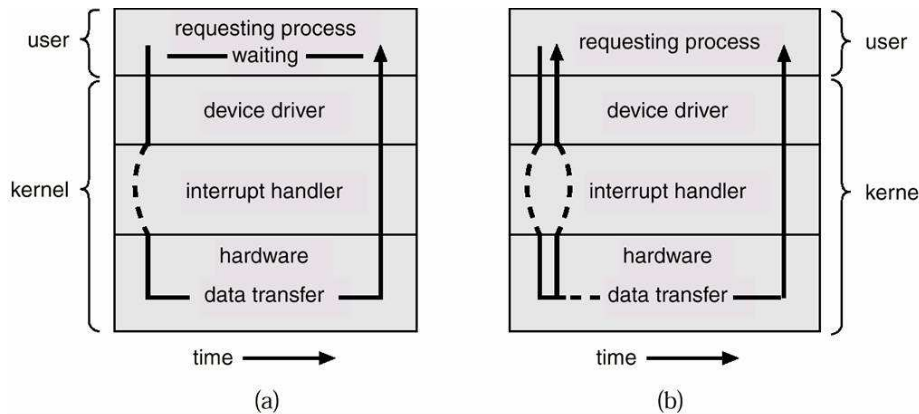


Figura 8 - Dois métodos de I/O: (a) síncrona e (b) assíncrona

Existem dois tipos de interrupções: a de *hardware* – que é um sinal originado em algum dispositivo físico; e a de *software* – que é um sinal originado por alguma aplicação.

2.4.1 Interrupção de *hardware*

O *hardware* da UCP (Unidade Central de Processamento) possui uma linha chamada linha de solicitação de interrupção que a UCP verifica depois de executar cada instrução. Quando a UCP detecta que uma controladora emitiu um sinal na linha de solicitação de interrupção, a UCP salva uma pequena quantidade de informações de estado, como o valor atual do ponteiro de instruções, e passa para a rotina de tratamento de interrupção em um endereço físico na memória. A rotina de tratamento de interrupção determina a causa da interrupção, realiza o processamento necessário e executa uma instrução *return from interrupt* para retornar a UCP ao estado de execução antes da interrupção. A Figura 9 resume o ciclo de I/O baseado em interrupções de *hardware*.

Interrupções de *hardware* podem ser originadas pelos vários dispositivos periféricos (discos, impressora, teclado e outros) ou pelo relógio. O relógio é um dispositivo que decrementa automaticamente o conteúdo de um registrador, com uma freqüência constante, e interrompe a UCP quando o valor do registrador atinge zero. Os computadores possuem instruções especiais para desabilitar (mascarar, inibir) o sistema de interrupção. Enquanto as interrupções estão desabilitadas elas podem ocorrer, mas não são atendidas pelo processador. Nesse caso, elas ficam pendentes (enfileiradas) e só voltam à serem atendidas quando a UCP executa uma instrução especial que habilita (desmascara, desinibe) as mesmas.

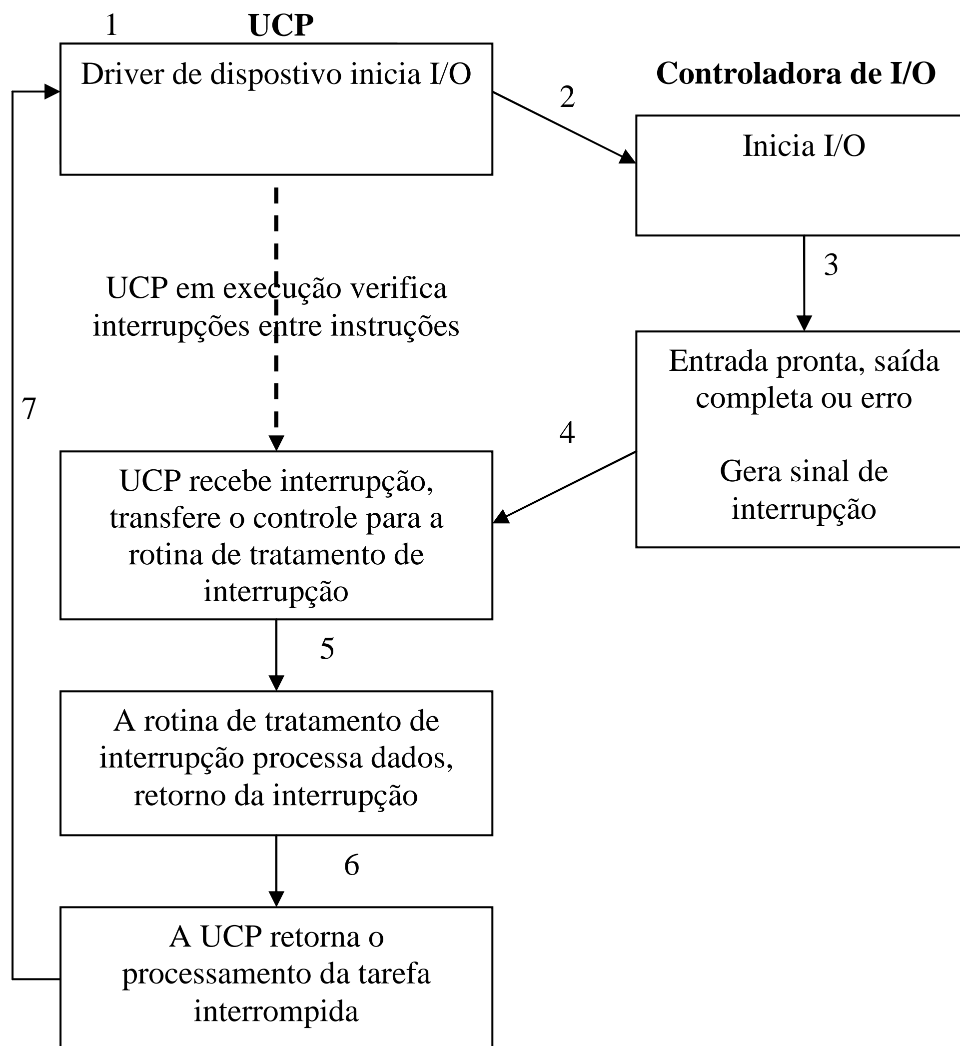


Figura 9 – Ciclo de E/S baseado em interrupções

2.4.2 Interrupção de *software*

Uma interrupção de *software* é um sinal gerado por uma instrução especial (portanto, por *software*) denominada *trap*, *system call* ou supervisor *call*. Quando uma instrução desse tipo é executada pela UCP, o computador desencadeia as mesmas ações desencadeadas por uma interrupção de *hardware*, isto é, o programa em execução é suspenso, informações são salva a rotina específica do SO é executada.

Os *traps* são necessários principalmente nos computadores que possuem **instruções protegidas** (privilegiadas). Nesses computadores o registrador (palavra) 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 do 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, portanto. Quando um programa de usuário necessita executar alguma dessas operações, o mesmo deve executar um *trap*, passando como argumento o número que identifica a operação que está sendo requerida.

A diferença fundamental dos *traps* para as interrupções de *hardware* é que eles são eventos previsíveis. Isto é, dado um programa e seus dados, é possível determinar os pontos em que os *traps* irão ocorrer. Por outro lado, as interrupções de *hardware* são geradas por dispositivos periféricos em pontos imprevisíveis. As interrupções de software têm prioridade baixa quando comparadas com as de *hardware*,

pois uma interrupção de *software* é menos urgente do que uma realizada por uma controladora de dispositivo, pois se a fila (FIFO) da controladora transbordar, pode ocorrer perda de dados.

2.5 DMA (Acesso direto a memória)

O acesso direto a memória (DMA) apresenta o seguinte esquema:

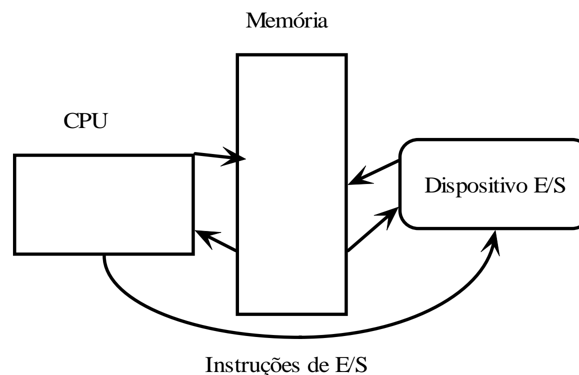


Figura 10- DMA

- Utilizado para dispositivos de E/S de alta velocidade capazes de transmitir informação com velocidades próximas da memória;
- O controlador do dispositivo transfere blocos de dados do *buffer* diretamente para a memória principal sem a intervenção da CPU;
- Somente uma interrupção é gerada por bloco, ao contrário de uma interrupção por *byte*.

2.6 Hierarquia de memória

Para o correto e eficaz funcionamento da manipulação das informações (instruções de um programa e dados) para a memória de um computador, verifica-se a necessidade de se ter, em um mesmo computador, diferentes tipos de memória. Para certas atividades, por exemplo, é fundamental que a transferência de informações seja a mais rápida possível. É o caso das atividades realizadas internamente no processador central, onde a velocidade é primordial, porém a quantidade de *bits* a ser manipulada é muito pequena (em geral, corresponde à quantidade de *bits* necessária para representar um único valor - um único dado).

Isso caracteriza um tipo de memória diferente, por exemplo, daquele em que a capacidade da memória (disponibilidade de espaço para guardar informações) é mais importante que a sua velocidade de transferência.

Ainda em relação ao tipo de alta velocidade e pequena quantidade de *bits* armazenáveis, que se usa na UCP, existem variações decorrentes do tipo de tecnologia utilizada na fabricação da memória.

Devido a essa grande variedade de tipos de memória, não é possível implementar um sistema de computação com uma única memória. Na realidade, há muitas memórias no computador, as quais se interligam de forma bem estruturada, constituindo um sistema em si, parte do sistema global de computação, podendo ser denominado subsistema de memória.

Esse subsistema é projetado de modo que seus componentes sejam organizados hierarquicamente, conforme mostrado na estrutura em forma de pirâmide da Figura 11.

A pirâmide em questão é projetada com uma base larga, que simboliza a elevada capacidade, o tempo de uso e o custo do componente que a representa.

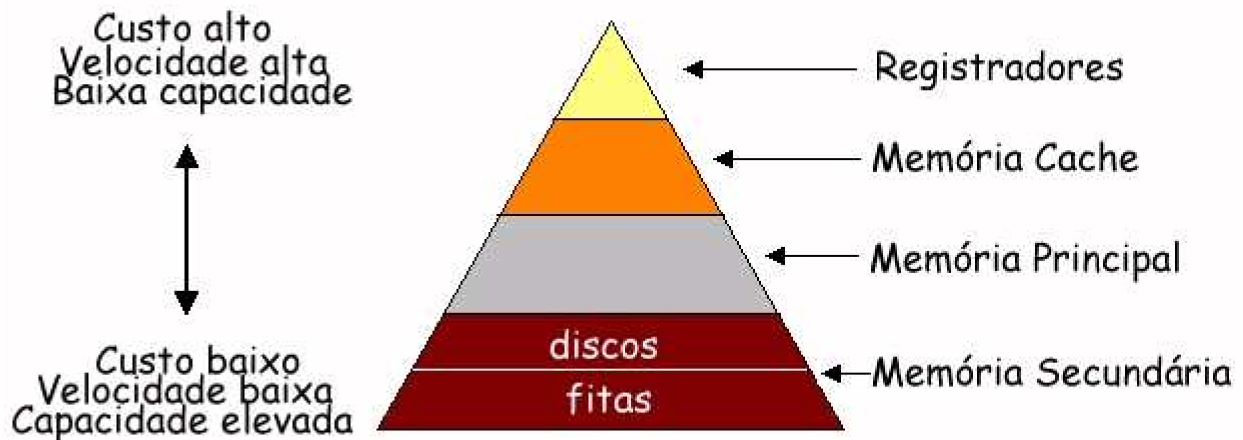


Figura 11 - Hierarquia de memórias.

A seguir serão definidos os principais parâmetros para análise das características de cada tipo de memória componente da hierarquia apresentada na Figura 11. O valor maior (base) ou menor (pico) de algum parâmetro foi a causa da utilização de uma pirâmide para representar a hierarquia do sistema de memória de um computador.

● Tempo de acesso - indica quanto tempo a memória gasta para colocar uma informação no barramento de dados após uma determinada posição ter sido endereçada. Isto é, o período de tempo decorrido desde o instante em que foi iniciada a operação de acesso (quando a origem - em geral é a CPU - passa o endereço de acesso para o sistema de memória) até que a informação requerida (instrução ou dado) tenha sido efetivamente transferida. É um dos parâmetros que pode medir o desempenho da memória. Pode ser chamado de tempo de acesso para leitura ou simplesmente tempo de leitura.

●Capacidade - é a quantidade de informação que pode ser armazenada em uma memória; a unidade de medida mais comum é o byte, embora também possam ser usadas outras unidades como células (no caso de memória principal ou cache), setores (no caso de discos) e bits (no caso de registradores). Dependendo do tamanho da memória, isto é, de sua capacidade, indica-se o valor numérico total de elementos de forma simplificada, através da inclusão de K (kilo), M (mega), G (giga) ou T (tera).

● Volatilidade - memórias podem ser do tipo volátil ou não volátil. Uma memória não volátil é a que retém a informação armazenada quando a energia elétrica é desligada. Memória volátil é aquela que perde a informação armazenada quando a energia elétrica desaparece (interrupção de alimentação elétrica ou desligamento da chave ON/OFF do equipamento).

● Custo - o custo de fabricação de uma memória é bastante variado em função de diversos fatores, entre os quais se pode mencionar principalmente a tecnologia de fabricação, que redunda em maior ou menor tempo de acesso, ciclo de memória, quantidade de bits em certo espaço físico e outros. Uma boa unidade de medida de custo é o preço por byte armazenado, em vez do custo total da memória em si. Isso porque, devido às diferentes capacidades, seria irreal considerar, para comparação, o custo pelo preço da memória em si, naturalmente diferente, e não da unidade de armazenamento (o *byte*), igual para todos os tipos.

2.7 Proteção de *hardware*

Em relação a proteção de *hardware*, tem-se os seguintes focos:

- Operação em dois modos (*dual-mode operation*);
- Proteção de E/S;
- Proteção de memória;
- Proteção da CPU;

2.7.1 Operação em dois modos

O compartilhamento de recursos de sistema requer que o sistema operacional garanta que um programa incorreto não possa causar danos à execução de outros programas. O *hardware* deve suportar a diferenciação entre pelo menos dois modos de operação:

1) Modo usuário: quando a execução está sendo feita em “nome” de um usuário;
2) Modo monitor (privilegiado): também conhecido como modo supervisor ou modo de sistema; ativado quando da execução do próprio sistema operacional.

O *hardware* utiliza um *bit* (*mode bi*t) que indica qual o modo corrente: monitor (0) ou usuário (1). Quando uma interrupção ou falha ocorre o *hardware* passa para o modo monitor.

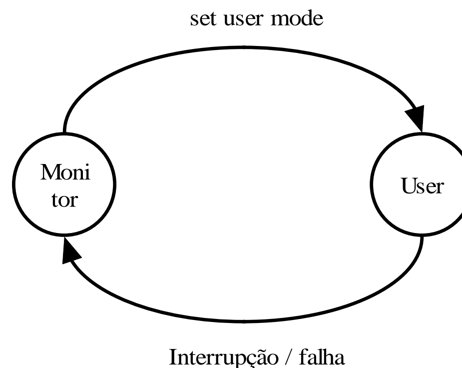


Figura 12 - Operações de dois modos

As instruções privilegiadas podem ser executadas somente no modo monitor. Dessa forma se garante que somente o sistema operacional pode executá-las; ou seja, um usuário não poderá alterar o sistema operacional acidentalmente ou propositalmente.

2.7.2 Operação de E/S

Todas as instruções de E/S são instruções privilegiadas (só podem ser executadas no modo monitor). Deve-se garantir que um programa de usuário nunca poderá ganhar controle do computador em modo monitor (por exemplo, um programa de usuário que durante a sua execução armazena um novo endereço no vetor de interrupções).

2.7.3 Proteção de Memória

Deve-se ter proteção de memória pelo menos para o vetor de interrupções e as rotinas de tratamento de interrupções. A fim de que se tenha proteção de memória, deve-se ter dois registradores extras que determinam os limites de espaço de endereçamento da memória que um programa pode acessar:

- *base register*: retém o menor endereço físico legal de memória;
- *limit register*: contém o tamanho do espaço acessível.

A memória fora da faixa estabelecida é protegida.

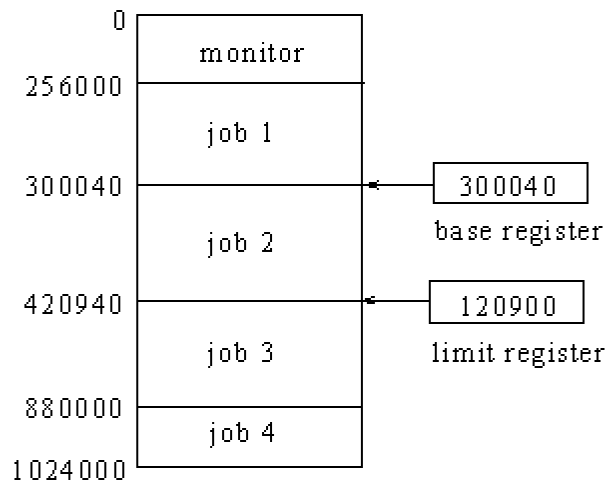


Figura 13 - A memória

Quando está executando no modo monitor, o sistema operacional tem acesso irrestrito a todo o espaço de endereçamento. As instruções de carga dos registradores “base” e “limit” são instruções privilegiadas. O *hardware* de proteção atua conforme apresentado na Figura 14:

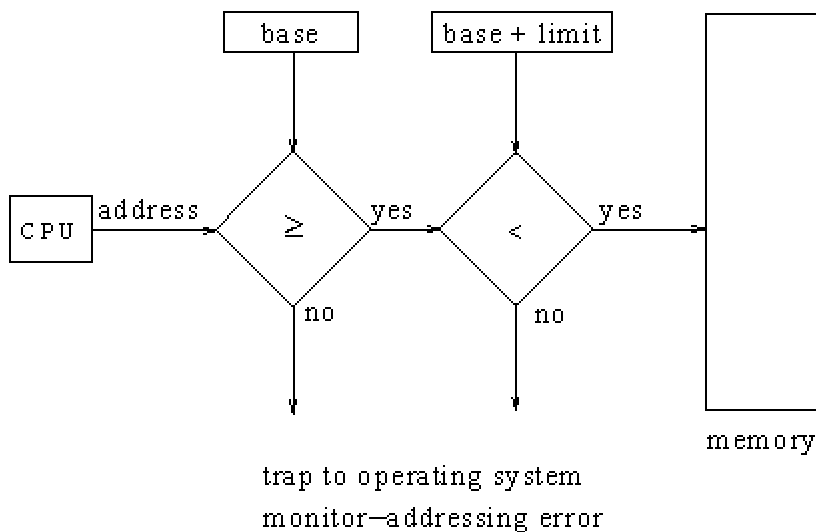


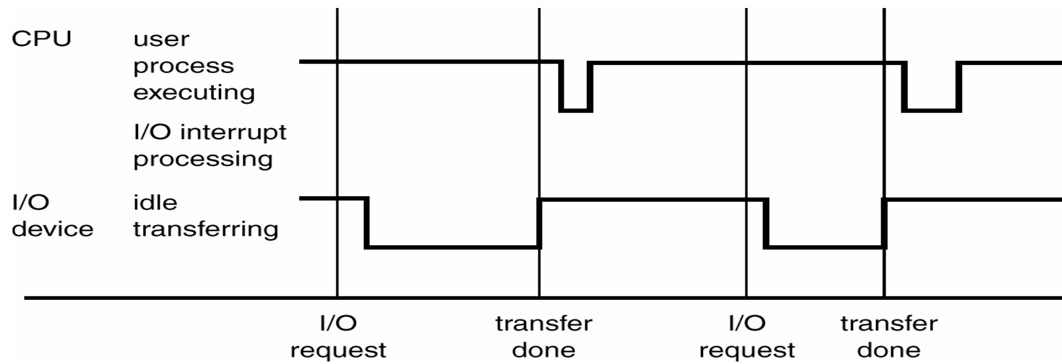
Figura 14 -A proteção da memória

2.7 Proteção de CPU

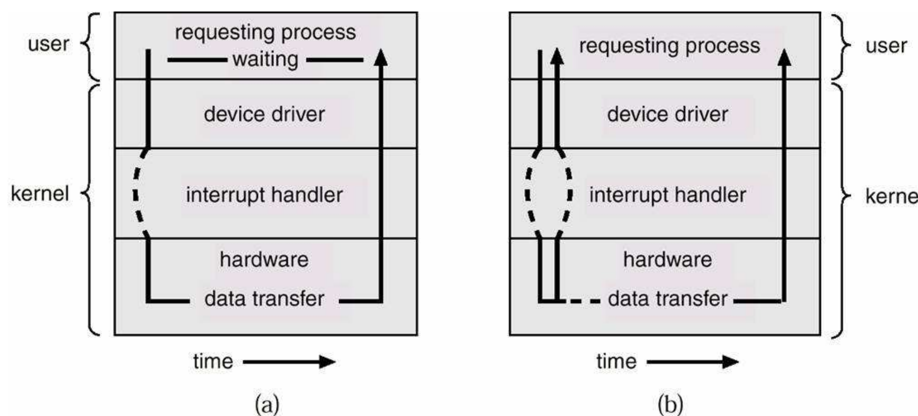
- ***Timer*:** interrompe o computador após um período especificado de tempo. O *timer* é decrementado a todo *tick* de relógio; quando alcança o valor zero (0), uma interrupção ocorre;
- O *timer* é comumente utilizado para implementar “time sharing”;
- O timer também é utilizado para calcular o tempo corrente;
- A instrução de *load* do timer é privilegiada.

2.8 Exercícios

1.Qual a função dos controladores de dispositivos? Por que eles possuem um buffer local?
2.O que é uma interrupção?
3.Explique o gráfico abaixo (Linha do Tempo de uma Interrupção para um Único Processo Gerando Saídas)4.Um programa de usuário pode alterar a rotina de tratamento de uma interrupção?



5.O gráfico abaixo apresenta interrupções de I/O síncrona/assíncrona, em quais situações é utilizado uma ou outra? Justifique sua resposta.



6.O que é DMA? Quando é utilizado?

7. Compare os dispositivos de armazenamento quanto a custo, velocidade e quantidade de informações armazenadas.

8. Como funciona a proteção de hardware para acesso à memória.

9. Se o sistema de I/O for lento, qual a conseqüência que isto acarretará para as outras aplicações?

3 ESTRUTURAS DOS SISTEMAS OPERACIONAIS

Os sistemas operacionais variam internamente em sua constituição, sendo organizados em muitas linhas diferentes. O projeto de um novo sistema operacional é uma tarefa importante. Portanto, é fundamental que os objetivos do sistema sejam bem definidos antes do início do projeto,

3.1 Componentes dos Sistemas Operacionais

É possível criar um sistema tão grande e complexo quanto um sistema operacional simplesmente dividindo-o em partes menores. Cada uma dessas partes deve ser uma porção bem delineada do sistema, com entradas, saídas e funções cuidadosamente definidas. Obviamente, nem todos os sistemas têm a mesma estrutura. A seguir são descritos os componentes gerais dos sistemas operacionais.

3.1.1 Gerência de processos

Um processo é um programa em execução. Um processo precisa de certos recursos, incluindo tempo de CPU, memória, arquivos, dispositivos de E/S, para realizar suas tarefas.

O SO é responsável pelas seguintes atividades com relação a gerência de processos:

- Criação e eliminação de Processos.
- Suspensão e retomada de processos.

Fornecimento de mecanismos para:

- Sincronização de processo
- Comunicação de processo

3.1.2 Gerência de memória principal

Memória é um grande *array* de *bytes*, cada um com o seu endereço. É um depósito de acesso rápido de dados compartilhado pela CPU e E/S. É um dispositivo de armazenamento volátil. O SO é responsável pelas seguintes atividades com relação a gerência de memória:

● Manter informações de que partes da memória estão em uso e por quem.
● Decidir que processos carregar quando espaços de memória estão disponíveis.
● Alocar e liberar espaço de memória quando necessário.

3.1.3 Gerência de arquivos

Um arquivo é uma coleção de informações relacionadas definidas pelo seu criador. Arquivos representam programas e dados. O SO é responsável pelas seguintes atividades com relação a gerência de arquivos:

● Criação e deleção de arquivo.
● Criação e deleção de diretório.
● Suporte de primitivas para manipular arquivos e diretórios.
● Mapeamento de arquivos na memória secundária.
● Backup de arquivos em meios de armazenagem estáveis (não volátil).

3.1.4 Gerência do sistema de I/O

Um dos objetivos do SO é ocultar as peculiaridades de hardware específicos dos usuários. O sistema de E/S consiste de:

● Um sistema de memória intermediária (*buffer*, *caching* e *spooling*)
● Uma interface geral para *drivers* de dispositivos
● *Drivers* para os dispositivos de hardware específicos

Um fato importante em relação ao sistema de é que apenas o *driver* de dispositivo conhece as peculiaridades do dispositivo ao qual foi atribuído.

3.1.5 Gerência de armazenamento secundário

Memória principal é volátil e pequena, sistema deve fornecer memória secundária para funcionar como backup da principal. Maioria dos sistemas modernos usam discos. O SO é responsável pelas seguintes atividades com relação a gerência de disco:

- Gerência de espaço livre
- Alocação de memória
- Escalonamento de disco

3.1.6 Proteção de sistema

Proteção refere-se a um mecanismo para controle de acesso de programas, processos, ou usuários a recursos do sistema e usuário. O mecanismo de proteção deve:

- Distinguir entre uso autorizado e não autorizado.
- Especificar os controles a serem impostos.
- Fornecer um meio de forçar os mesmos.

3.1.7 Interpretador de comandos

Um dos programas de sistema mais importantes para um sistema operacional é o interpretador de comandos, que é a interface entre o usuário e sistema operacional. Alguns sistemas operacionais incluem o interpretador de comandos no *kernel*. Outros, tais como MS-DOS e o UNIX, tratam o ele como um programa especial que fica executando quando um *job* é iniciado ou quando um usuário entra no sistema. Por exemplo, os comandos para as seguintes atividades podem ser dados ao SO no interpretador de comandos:

- criação de processos e gerência
- tratamento de E/S
- gerência de memória secundária
- gerência de memória principal
- acesso ao sistema de arquivos
- proteção
- rede

O programa que lê e interpreta comandos de controle é chamado interpretador de linha de comando. No UNIX é o shell. Sua função é capturar e executar o próximo comando de controle.

3.2 Estrutura dos Sistemas Operacionais

A estrutura de um Sistema Operacional está relacionada ao desenho (*design*) interno do sistema. Os seguintes tipos de estrutura serão examinados nas próximas seções: Sistemas Monolíticos, Sistemas em Camada e Sistemas Cliente-Servidor.

3.2.1 Sistemas Monolíticos

Neste tipo de estrutura (Figura 15), o Sistema Operacional é escrito como uma coleção de rotinas, onde cada uma pode chamar qualquer outra rotina, sempre que for necessário. Portanto, o sistema é estruturado de forma que as rotinas podem interagir livremente umas com as outras. Quando esta técnica é usada, cada rotina no sistema possui uma interface bem definida em termos de parâmetros e resultados.

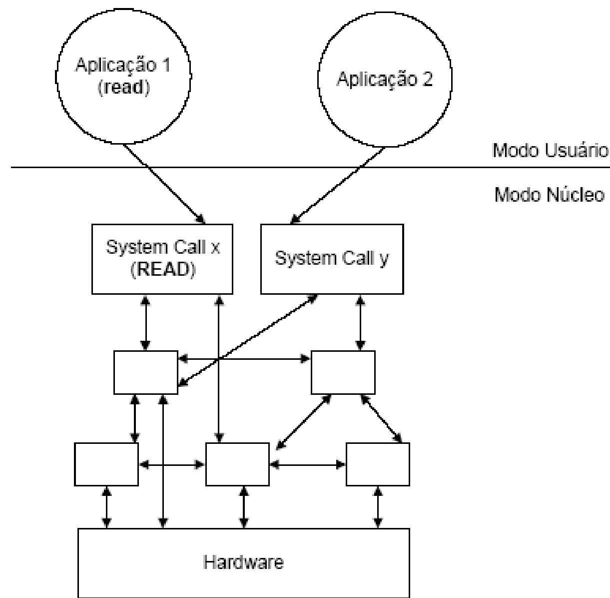


Figura 15- Sistema Operacional monolítico

Os serviços do Sistema Operacional (gerenciamento de processos, gerenciamento de memória, gerenciamento do sistema de arquivos, ...) são implementados por meio *de System Calls* em diversos módulos, executando em Modo Núcleo. Mesmo que as diversas rotinas que fornecem serviços estejam separadas umas das outras, a integração de código é muito grande e é difícil desenvolver o sistema corretamente. Como todos os módulos (rotinas) executam no mesmo espaço de endereçamento, um *bug* em um dos módulos pode derrubar o sistema inteiro. Evidentemente que esta é uma situação indesejável.

Para construir um código executável de um SO desta natureza, todas as rotinas (ou arquivos que possuem as rotinas) são compiladas individualmente e unidas pelo *linker* em um código executável único. Tal código executa em Modo Núcleo. Não existe ocultação de informação, o que também é indesejável, pois cada rotina é visível a qualquer outra. A Figura 16 mostra o processo de criação de um código executável de um SO Monolítico.

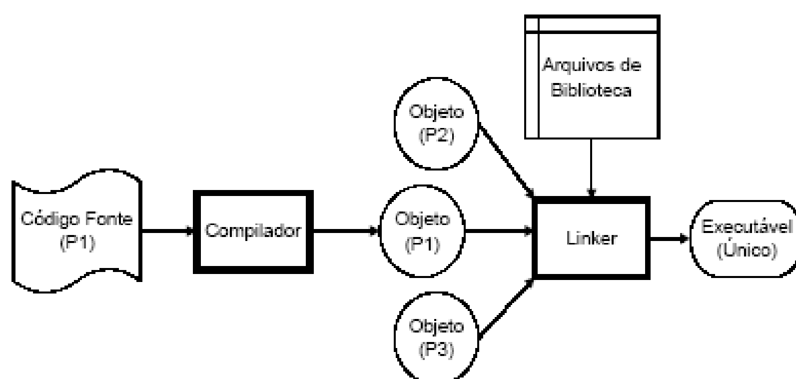


Figura 16- Geração de Código Executável de um SO Monolítico. As caixas com borda mais grossa indicam ferramentas do ambiente de desenvolvimento.

Embora possa parecer que não há quase estruturação em um SO Monolítico, existe um pouco de estruturação quando os serviços do são solicitados por meio das *System Calls*.

Como vantagem dos SOs Monolíticos pode-se afirmar que, se a implementação do sistema está completa e confiável, após um processo de desenvolvimento em que se supõe que técnicas consagradas de

Engenharia de Software tenham sido aplicadas, a forte integração interna dos componentes permite que detalhes de baixo nível do hardware sejam efetivamente explorados, fazendo com um bom SO Monolítico seja altamente eficiente. Entre os SOs Monolíticos estão as versões tradicionais do **UNIX**, incluindo o **Linux**, e **MS-DOS**.

3.2.2 Sistemas em Camada

A idéia por trás deste tipo de SO é fazer a organização por meio de hierarquia de camadas. O SO é dividido em camadas sobrepostas, onde cada módulo oferece um conjunto de funções que podem ser utilizadas por outros módulos. Módulos de uma camada podem fazer referência apenas a módulos das camadas inferiores. O primeiro SO construído de acordo com esta concepção foi o **THE**, que foi desenvolvido na Technische Hogeschool Eindhoven na Holanda por E. W. Dijkstra (1968) e seus estudantes. O computador que executava o THE possuía Memória Principal com capacidade de 32K palavras de 27 bits cada. A estrutura do THE pode ser vista na Figura 17.

5	Operador
4	Programas de Usuário
3	Entrada/Saída (E/S)
2	Comunicação
1	Gerenciamento de Memória
0	Multiprogramação

Figura 17 - Estrutura do Sistema Operacional THE.

A camada 0 era responsável pela alocação do processador entre os processos, chaveamento entre processos quando ocorria interrupções ou quando os temporizadores expiravam. Resumindo, a camada 0 fornecia a multiprogramação básica da CPU. Acima da camada 0, o sistema consistia de processos seqüenciais que podiam ser programados sem se preocupar se havia múltiplos processos executando na CPU.

A camada 1 realizava o gerenciamento de memória. Ela alocava espaço para os processos na Memória Principal do sistema e também em um Tambor (dispositivo de armazenamento magnético usado nos computadores antigamente) de 512K palavras, usado para armazenar partes de processos (páginas) para as quais não havia espaço na Memória Principal. Acima da camada 1, os processos não tinham que se preocupar se eles estavam na Memória Principal ou no Tambor; a camada 1 do SO era quem tratava deste tipo de situação, trazendo as partes do software para a Memória Principal sempre quando necessário.

A camada 2 manipulava a comunicação entre cada processo e o operador do *console*. Um *console* consistia de um dispositivo de entrada (teclado) e um de saída (monitor ou impressora). A camada 3 era responsável pelo gerenciamento dos dispositivos de E/S. Acima da camada 3, cada processo podia lidar com dispositivos de E/S abstratos, com propriedades mais agradáveis, e não com os dispositivos reais em si. Na camada 4 havia os programas do usuário, e na camada 5 havia o processo do operador do sistema.

O esquema de camadas do THE era, de fato, apenas um auxílio de desenho (*design*), pois todas as partes do sistema eram ultimamente unidas em um único código executável.

3.2.3 Sistemas Cliente-Servidor

Os Sistemas Operacionais com estrutura Cliente-Servidor são baseados em Micronúcleo (Microkernel). A idéia neste tipo de sistema é tornar o núcleo do SO o menor e o mais simples possível (Micronúcleo), movendo código para camadas superiores. A abordagem usual é implementar a maior parte dos serviços do SO em processos de usuário. Em tal implementação, o SO é dividido em processos, sendo cada um responsável por oferecer um conjunto de serviços tais como:

- serviços de arquivo (servidor de arquivos);

● serviços de criação de processos (servidor de processos);
● serviços de memória (servidor de memória), etc...

Para requisitar um serviço, tal como ler um bloco de dados de um arquivo, um processo de usuário, conhecido como processo cliente, envia uma solicitação a um processo servidor (servidor de arquivos, neste caso), que realiza o trabalho e envia a resposta de volta ao processo cliente. A Figura 18 mostra o modelo Cliente-Servidor.

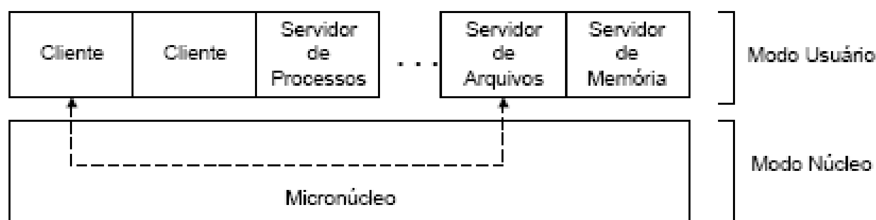


Figura 18- Modelo Cliente-Servidor

Observe na Figura 17 que diversas funções do SO estão agora sendo executadas no Modo Usuário, e não mais no Modo Núcleo, como era no caso dos Sistemas Monolíticos. Somente o Micronúcleo do SO executa no Modo Núcleo. Ao Micronúcleo cabe, basicamente, permitir a comunicação entre processos clientes e servidores. Entre as vantagens apresentadas pelos Sistemas Cliente-Servidor estão o fato de, ao dividir o SO em partes onde cada parte manipula um aspecto do sistema tais como serviço de arquivos, serviço de processo, serviço de memória entre outros, cada parte se torna menor e mais fácil de gerenciar. Além disto, devido ao fato de que todos os processos servidores executarem em Modo Usuário, eles não têm acesso direto ao *hardware* da máquina. Como conseqüência, se houver um *bug* no processo servidor de arquivos, este serviço pode deixar de funcionar, mas isto usualmente não derrubará (*crash*) o sistema inteiro. Se esta mesma situação ocorresse em um SO Monolítico, e possível que o sistema sofresse conseqüências mais sérias do que em um Sistema Cliente-Servidor. Uma outra vantagem do modelo Cliente-Servidor é a sua adaptabilidade para usar em sistemas com processamento paralelo/distribuído. Se um processo cliente se comunica com um servidor pelo envio de mensagens, o cliente não necessita saber se a mensagem é tratada localmente na sua máquina, ou se ela foi enviada por meio de uma rede para um processo servidor executando em uma máquina remota. Do ponto de vista do cliente, o mesmo comportamento ocorreu: um pedido foi requisitado e houve uma resposta como retorno. A Figura 19 mostra esta situação.

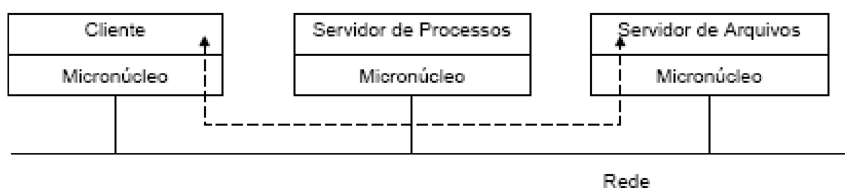


Figura 19 - Uso do Modelo Cliente-Servidor em um Sistema Paralelo ou Distribuído

Como exemplos de SO Cliente-Servidor pode-se citar o Minix, o Windows NT e o QNX.

3.3 Sistemas Monolíticos versus Sistemas Cliente-Servidor

Em uma primeira análise, uma estrutura de SO Cliente-Servidor parece ser bem melhor do que um SO Monolítico. Porém, em termos práticos, a implementação de uma estrutura Cliente-Servidor é bastante complicada devido a certas funções do SO exigirem acesso direto ao hardware, como operações de E/S. Um núcleo Monolítico, por outro lado, possui uma complexidade menor, pois todo código de controle do sistema reside em um espaço de endereçamento com as mesmas características (Modo Núcleo). Usualmente, SOs Monolíticos tendem a ser mais fáceis de desenhar corretamente e, portanto, podem crescer mais rapidamente do que SOs baseados em Micronúcleo. Existem casos de sucesso em ambas as estruturas. Um aspecto interessante sobre qual a melhor estrutura de SO foi a discussão entre Linus Torvalds, o criador do SO Linux, e Andrew Tanenbaum, um dos principais pesquisadores na área de SOs e criador do SO Minix. Em 1992, quando o Linux estava no seu início, Tanenbaum decidiu escrever uma mensagem para o Newsgroup comp.os.minix, acusando justamente o Linux de ser um SO obsoleto. O ponto principal do argumento de Tanenbaum era justamente a estrutura Monolítica, considerada ultrapassada por ele, do Linux. Ele não concebia que um SO, em meados dos anos 90, fosse concebido com um tipo de estrutura que remonta a década de 70 (época em que o Unix foi desenvolvido; o Unix também é um SO Monolítico). O SO desenvolvido por Tanenbaum, Minix, apresenta estrutura baseada em Micronúcleo (Cliente-Servidor). Em sua primeira resposta Torvalds argumentou, entre vários pontos, um aspecto não muito técnico: o Minix não era gratuito, enquanto o Linux sim. Do ponto de vista técnico, Torvalds concordava que um sistema com Micronúcleo era mais agradável. Porém, ele acusava o Minix de não realizar corretamente o papel do Micronúcelo, de forma que havia problemas no que se refere a parte de multitarefa no núcleo do sistema. A discussão continuou entre ambos sobre outros conceitos associados a SOs. Para saber mais sobre este assunto, vide o link http://www.dina.dk/~abraham/Linus_vs_Tanenbaum.html#liu. Vários anos após tal discussão, o que pode ser afirmado é que SOs Monolíticos ainda conseguem atrair a atenção de desenvolvedores devido a uma complexidade menor do que Sistemas Cliente-Servidor. Tanto que o Linux hoje é uma realidade, sendo um SO bastante usado em servidores em empresas e ambientes acadêmicos. Os Sistemas Cliente-Servidor, porém, possuem casos de sucesso como é o exemplo do sistema QNX, usado em sistemas de braços de robôs nos Ônibus Espaciais.

3.4 Exercícios

1.Quais são os componentes de um SO ?
2.Qual a responsabilidade do SO em relação à Gerência de Processos ?
3.O que é um processo ?
4.Qual a responsabilidade do SO em relação à Gerência de Memória ?
5.Qual a responsabilidade do SO em relação à Gerência de Arquivos ?
6.Qual a responsabilidade do SO em relação à Gerência de I/O ?
7.Qual a responsabilidade do componente de Proteção de Sistema ?
8.Se ocorrer uma divisão por zero, o que ocorre ? quem trata o erro ?
9.Qual a vantagem/desvantagem de desenvolver um SO na linguagem C ?
10.Qual a vantagem/desvantagem de desenvolver um SO na linguagem *Assembly* ?
11. Compare as abordagens em camadas e *microkernel* no projeto do SO?

4 PROCESSOS

Fazendo uma retrospectiva das últimas décadas do mercado de computação de alto desempenho, *High Performance Computing* (HPC), verifica-se que o fator mais constante foi a evolução. Este mercado tem sido marcado pela rápida mudança de fornecedores, arquiteturas, tecnologias e aplicações. Apesar desta instabilidade, a evolução do desempenho em larga escala tem-se mostrado um processo contínuo e constante. A convergência de desempenho entre os microcomputadores e os supercomputadores com um só processador motivou o aparecimento do processamento paralelo, que vem ultrapassar alguns limites do uniprocessamento, reduzir alguns custos (podem construir-se supercomputadores recorrendo a um conjunto de processadores normais), e também para facilitar a expansibilidade de um computador (bastando para isso adicionar mais processadores).

4.1 Fundamentos

O conceito de processo é, certamente, o conceito mais importante no estudo de sistemas operacionais. Para facilitar o entendimento deste conceito, considere-se um computador funcionando em multiprogramação (isto é, tendo vários programas simultaneamente ativos na memória). Cada programa em execução corresponde a um procedimento (seqüência de instruções) e um conjunto de dados (variáveis utilizadas pelo programa). É conveniente ter as instruções separadas dos dados, pois isso possibilita o compartilhamento do código do procedimento por vários programas em execução (neste caso diz-se que o procedimento e **reentrante** ou puro). Se cada programa em execução possui uma pilha própria, então os dados podem ser criados (alocados) na própria pilha do programa.

Além das instruções e dados, cada programa em execução possui uma área de memória correspondente para armazenar os valores dos registradores da UCP, quando o programa, por algum motivo, não estiver sendo executado. Essa área de memória é conhecida como **bloco de controle de processo -BCP** (ou bloco descritor, bloco de contexto, registro de estado, vetor de estado) e, além dos valores dos registradores da UCP, contém outras informações. A Figura 20 mostra um BCP. Ele contém muitas informações associadas a um processo específico:

- Estado do processo: o estado pode ser pronto, execução ou bloqueado (espera).
- Contador do programa: o contador indica o endereço da próxima instrução a ser executada para esse processo.
- Registradores de UCP: os registradores variam em número e tipo, dependendo da arquitetura do computador. Incluem acumuladores, registradores, ponteiros de pilha e registradores de uso geral, além de informações de código de condição. Juntamente com o contador do programa, essas informações de estado devem ser salvas quando ocorre uma interrupção, para permitir que o processo continue corretamente depois.
- Informações de escalonamento de UCP: essas informações incluem prioridade de processo, ponteiros para filas de escalonamento e quaisquer outros parâmetros de escalonamento
- Informações de gerência de memória: essas informações podem incluir dados como o valor dos registradores de base e limite, as tabelas de páginas ou tabelas de segmentos, dependendo do sistema de memória usado pelo sistemas operacional
- Informações de contabilização: essas informações incluem a quantidade de UCP e tempo real usados, limites de tempo, números de contas, números de *jobs* ou processos, etc.
- Informações de status de E/S: as informações incluem a lista de dispositivos de E/S alocados para este processo, uma lista de arquivos abertos e outras informações.

ponteiro	estado do processo
Número do processo	
Contador de programa	
registradores	
limites de memória	
limites de arquivos abertos	
...	

Figura 20 – Bloco de controle de processo (BCP)

Cada programa em execução constitui um **processo**. Portanto, pode-se definir processo como sendo um programa em execução, o qual é constituído por uma seqüência de instruções, um conjunto de dados e um bloco de controle de processo.

Num ambiente de multiprogramação, quando existe apenas um processador na instalação, 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, 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 o efeito 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 o serviço requerido pelo mesmo, ao sistema operacional, seja realizado.

Deve ser observado que um processo é uma entidade completamente definida por si só, cujas operações (instruções executadas) se desenvolvem no tempo, em uma ordem que é função exclusiva dos valores iniciais de suas variáveis e dos dados lidos durante a execução.

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 de forma intercalada N processos 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 perde na execução de suas próprias funções, como por exemplo o tempo perdido para fazer a multiplexação da UCP entre os processos. É o tempo durante o qual o sistema não está produzindo trabalho útil para qualquer usuário. Quando a UCP altera a execução para outro processo, o sistema salva o estado do processo antigo e carregar o estado salvo do novo processo (Troca de contexto). A Figura 21 exemplifica a troca de contexto.

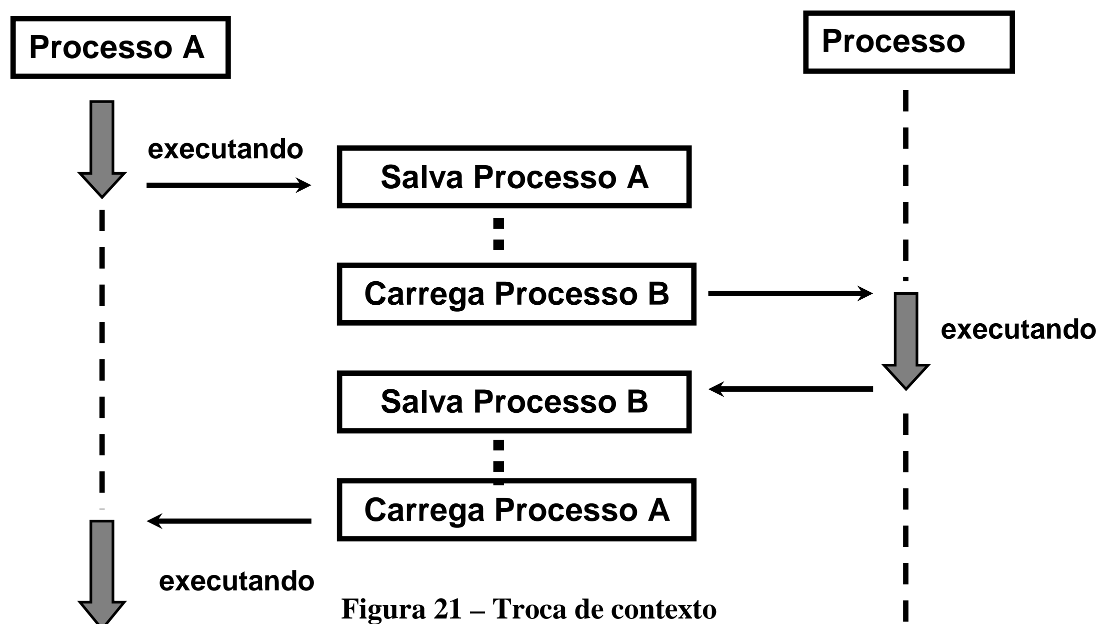


Figura 21 – Troca de contexto

Tanto no paralelismo físico (real, com várias UCP) como no lógico (virtual, uma UCP compartilhada), as velocidades relativas com que os processos acessarão dados compartilhados não podem ser previstas.

Quando os processos são denominados paralelos quando eles realmente (de maneira física) são executados simultaneamente e são denominados concorrentes quando são disputam para serem executados no mesmo período de tempo em uma UCP. De acordo com o tipo de interação existente entre eles, podem ser classificados como **disjuntos** (não interativos), quando operam sobre conjuntos distintos de dados, ou **interativos**, quando têm acesso a dados comuns. Processos interativos podem ser **competitivos**, se competirem por recursos, e/ou **cooperantes**, se trocarem informações entre si.

No caso de computações realizadas por processos interativos, como a ordem das operações sobre as variáveis compartilhadas pode variar no tempo (pois as velocidades relativas dos processos dependem de fatores externos que variam no tempo), o resultado da computação pode não depender somente dos valores iniciais das variáveis e dos dados de entrada. Quando o resultado de uma computação varia de acordo com as velocidades relativas dos processos diz-se que existe uma **condição de corrida** (*race condition*). É necessário evitar condições de corrida para garantir que o resultado de uma computação não varie entre uma execução e outra. Condições de corrida resultam em computações paralelas errôneas, pois cada vez que o programa for executado (com os mesmos dados) resultados diferentes poderão ser obtidos. A programação de computações paralelas exige mecanismos de sincronização entre processos, e por isso a sua programação e depuração são bem mais difíceis do que em programas tradicionais.

A maioria das linguagens de programação existentes não permite a programação de computações paralelas, pois cada programa gera um único processo durante a sua execução. Tais linguagens são denominadas seqüenciais. Linguagens que permitem a construção de programas que originam vários processos para serem executados em paralelo são denominadas **linguagens de programação concorrente**. Exemplos deste tipo de linguagem são: Pascal Concorrente, Modula 2, Ada, Java e algumas extensões de linguagem C.

A programação concorrente, além de ser essencial ao projeto de sistemas operacionais, também tem aplicações na construção de diversos outros tipos de sistema importantes. Qualquer sistema que deva atender a requisições de serviço que possam ocorrer de forma imprevisível pode ser organizado, convenientemente, para permitir que cada tipo de serviço seja realizado por um dos processos do sistema. Dessa maneira, diversos serviços poderão ser executados simultaneamente e a utilização dos recursos computacionais será, certamente, mais econômica e eficiente. Exemplos de aplicações deste tipo são sistemas para controle *on-line* de informações (contas bancárias, estoques, etc) e controle de processos externos (processos industriais, processos químicos, rotas de foguetes, etc).

Os processos durante suas execuções requerem operações de E/S que são executadas em dispositivos

muito lentos que a UCP, pois os dispositivos periféricos possuem componentes mecânicos, que funcionam a velocidades muito inferiores à dos dispositivos eletrônicos — que funcionam à velocidade da luz.

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 completa e, portanto, mais econômica e mais eficiente. Se um processo passa a maior parte do tempo esperando por dispositivos de E/S, diz-se que o processo é **ligado a E/S** (limitado por E/S ou *I/O-bound*). Se, ao contrário, o processo gasta a maior parte do seu tempo usando a UCP ele é dito **ligado a CPU** (limitado por computação ou *compute-bound* ou *UCP-bound*). Obviamente, processos *I/O-bound* devem ter prioridade sobre processos *UCP-bound.*

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 de um *job* pequeno e prioritário ser submetido após um *job* demorado já ter iniciado a execução, a multiprogramação fará com que o *job* pequeno seja executado em paralelo e termine muito antes do término do *job* longo.

Os sistemas operacionais acionam os dispositivos de E/S através de instruções do tipo *Start I/O* (Iniciar E/S). Se o dispositivo é uma unidade de disco, por exemplo, a instrução faz com que um bloco de setores do disco seja lido para a memória principal. Quando o dispositivo termina a operação, ele manda um sinal de interrupção para a UCP, indicando que está livre para realizar outra operação. Este sinal faz com que o controle da execução vá para o sistema operacional, o qual pode acionar o dispositivo para executar outra operação, antes de devolver a UCP para um processo de usuário.

Durante suas execuções os processos dos usuários, ocasionalmente, através de *traps*, fazem requisições ao sistema operacional (para gravar um setor de disco, por exemplo). Recebendo a requisição, o sistema operacional bloqueia o processo (deixa de dar tempo de UCP a ele) até que a operação requerida seja completada. Quando isto acontece o processo é desbloqueado e volta a competir pela UCP com os demais processos.

Quando um processo está realmente usando a UCP, diz-se que o mesmo está no estado **executando** (*running*). Quando está esperando pelo término de um serviço que requereu, diz-se que está no estado **bloqueado** (*blocked*). 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** (*ready*). O sistema operacional mantém uma lista (fila) dos processos que estão prontos, a chamada **lista de processos prontos** (*ready list* ou *ready queue*). O diagrama da Figura 22 mostra como os estados de um processo podem mudar durante a execução.

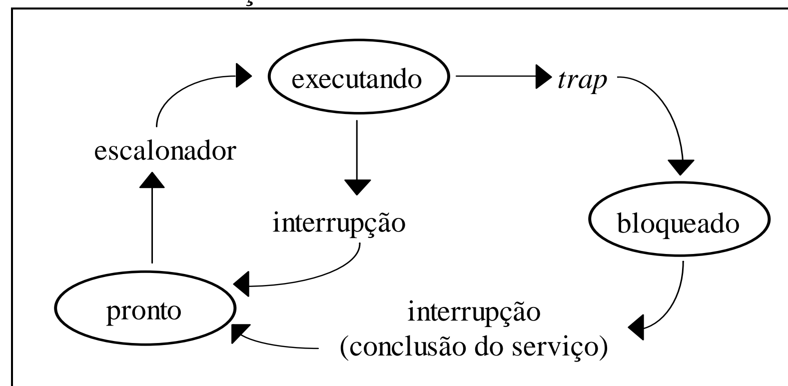


Figura 22 - Estados sucessivos de um processo no sistema

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 (*scheduler*) ou **despachador** de processos (*dispatcher*).

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, não ocorrendo o bloqueio.

Quando um *job* é admitido no sistema, um processo correspondente é criado e normalmente inserido

no final da fila de prontos (*ready list*). O processo se move gradualmente para a cabeça da fila de prontos, conforme os processos anteriores a ele forem sendo usados pela UCP.

Quando o processo alcança a cabeça da lista, e quando a UCP torna-se disponível, o processo é dado à UCP e diz-se que foi feita uma transição do estado pronto para o estado executando. A transferência da UCP para o primeiro processo da fila de prontos é chamado de escalonamento (*dispatching*), e é executada pelo escalonador. Este transição de estado pode ser ilustrada da seguinte forma:

Escalone(nomedoprocesso): pronto ➔ execução

Para prevenir que um processo monopolize o sistema acidentalmente ou propositadamente, o sistema operacional tem um relógio interno (*interrupting clock* ou *interval timer*) que faz com que o processo execute somente por um intervalo de tempo específico ou *quantum*. Se o processo voluntariamente não libera a UCP antes de expirar seu intervalo de tempo, o *interrupting clock* gera uma interrupção, dando ao sistema operacional o controle novamente. O sistema operacional torna o processo corrente (execução) em pronto e torna o primeiro processo da fila de pronto em corrente. Estas transições de estado são indicadas como:

Tempoexpirou(nomedoprocesso): execução ➔ pronto
Escalone(nomedoprocesso): pronto ➔ execução

Se um processo corrente 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(nomedoprocesso): execução ➔ bloqueado

Quando é terminada a operação que fez com que o estado fique bloqueado, este passa para o estado pronto. A transição que faz tal operação é definida como:

Acorde(nomedoprocesso): bloqueado ➔ pronto

Deste modo podemos definir quatro possíveis estados de transição:

- Escalone(nomedoprocesso): pronto ➔ execução
- Tempoexpirou(nomedoprocesso): execução ➔ pronto
- Bloqueia(nomedoprocesso): execução ➔ bloqueado
- Acorde(nomedoprocesso): bloqueado ➔ pronto

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

4.2 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 ordinariamente reside em armazenamento primário (memória RAM) enquanto outras porções do sistema operacional são chamadas da memória secundária quando necessário.

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

O núcleo desabilita interrupções enquanto ele responde a uma interrupção; interrupções são novamente habilitadas após o processamento de uma interrupção estar completo. Com um fluxo permanente de interrupções, é possível que o núcleo mantenha interrupções desabilitadas por uma grande porção 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 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 torna-se mais eficiente em responder a requisições das aplicações dos usuários.

Um sistema operacional normalmente possui código para executar as seguintes funções:

- Manipulação de interrupções;
- Criação e destruição de processos;
- Troca de contexto de processos;
- Suspensão e reanimação de processos;
- Sincronização de processos;
- Intercomunicação entre processos;
- Manipulação de BCPs;
- 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.

4.3 Escalonamento de Processos

Os processos podem estar executando, bloqueados, ou prontos para serem executados. Quando um ou mais processos estão **prontos** para serem executados, o sistema operacional deve decidir qual deles vai ser executado primeiro. A parte do sistema operacional responsável por essa decisão é chamada **escalonador**, e o algoritmo usado para tal é chamado de **algoritmo de escalonamento**. Os algoritmos de escalonamento dos primeiros sistemas, baseados em cartões perfurados e unidades de fita, era simples: ele simplesmente deveria executar o próximo *job* na fita ou leitora de cartões. Em sistemas multi-usuário e de tempo compartilhado, muitas vezes combinados com *jobs batch* em *background*, o algoritmo de escalonamento é mais complexo.

Antes de vermos os algoritmos de escalonamento, vejamos os critérios com os quais eles devem se preocupar:

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. Para minimizar o tempo de resposta para usuários interativos, o escalonador não deveria rodar nenhum *job batch* (exceto entre 3 e 6 da manhã, quando os usuários interativos estão dormindo). Usuários *batch* não gostarão deste algoritmo, porque ele viola a regra 4.

Uma complicação que os escalonadores devem levar em consideração é que cada processo é único e imprevisível. Alguns passam a maior parte do tempo esperando por E/S de arquivos, enquanto outros utilizam a UCP por horas se tiverem chance. Quando o escalonador inicia a execução de um processo, ele nunca sabe com certeza quanto tempo vai demorar até que o processo bloqueie, seja por E/S, seja em um semáforo, seja por outro motivo. Para que um processo não execute tempo demais, praticamente todos os computadores possuem um mecanismo de relógio (**clock**) que causa uma interrupção periodicamente. Freqüências de 50 ou 60 Hz são comuns, mas muitas máquinas permitem que o sistema operacional especifique esta freqüência. A cada interrupção de relógio, o sistema operacional assume o controle e decide se o processo 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 estejam querendo executar é chamada de **escalonamento preemptivo**, em contraste com o método **execute até o**

fim dos antigos sistemas *batch*. 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 semáforos, contadores de eventos, monitores, ou algum outro método de comunicação interprocessos. Por outro lado, uma política de deixar um processo rodar enquanto desejar pode fazer com que um processo que demore uma semana para executar deixe o computador ocupado para os outros usuários durante este tempo.

4.3.1 Escalonamento FCFS ou FIFO

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 são escalonados de acordo com sua ordem de chegada na fila de processos prontos do sistema. Uma vez que um processo ganhe a UCP, ele roda até terminar. FIFO é uma abordagem 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 fazem *jobs* importantes esperar. FIFO oferece uma menor variância 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*.

4.3.2 Escalonamento Round Robin

Um dos mais antigos, simples, justos, e mais largamente utilizados dos algoritmos 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*, a UCP é dada a outro processo. Se um processo bloqueou ou terminou antes do final do *quantum*, a troca de 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 *runnable* (que desejam executar), conforme a Figura 23(a). Quando o *quantum* de um processo acaba, ele é colocado no final da lista, conforme a Figura 23(b).

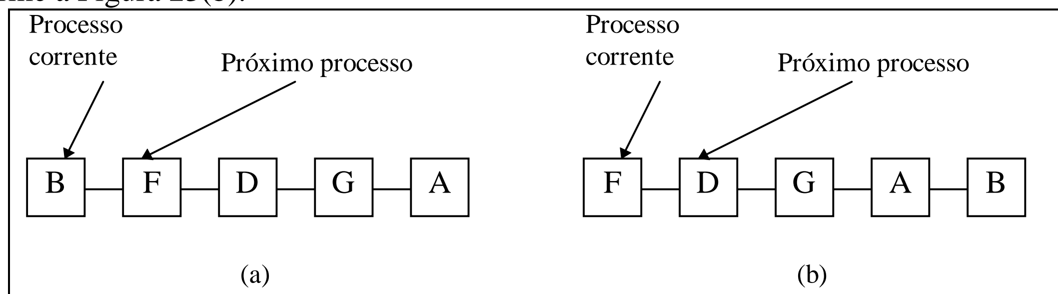


Figura 23 - Escalonamento Round Robin.

(a) Lista de processos a executar.
(b) Lista de processos a executar depois de terminado o *quantum* de 'B'

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, um por vez. Executando sucessivamente em intervalos de tempo o *job* acaba por terminar sua execução em algum momento.

O único aspecto interessante sobre o algoritmo *round robin* é a duração do *quantum*. Mudar de um processo para outro requer um certo tempo para a administração — salvar e carregar registradores e mapas de memória, atualizar tabelas e listas do Sistema Operacional, etc. Suponha esta **troca de contexto** dure 5 ms. Suponha também que o *quantum* está 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, disparando cada um processo. Dez processos serão colocados na lista de processo 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 do 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 "muita" coisa.

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.

4.3.3 Escalonamento com Prioridades

O algoritmo *round 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 nos leva ao **escalonamento com prioridades**. A idéia básica é direta: 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 prioridade do processo atualmente executando a cada *tick* de relógio (isto é, a cada interrupção de relógio). Se esta ação fizer com que a prioridade do processo 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 a 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 (incomum hoje em dia), *jobs* de alta prioridade poderiam custar 100 dólares por hora, os de média prioridade a 75 por hora, e os de baixa prioridade a 50 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 do processo. Entretanto, o superusuário UNIX pode aumentar a prioridade de processos.

Prioridades podem também ser atribuídas dinamicamente pelo sistema para atingir certos objetivos do sistema. Por exemplo, alguns processos são altamente limitados por E/S, e passam a maior parte do tempo esperando por operações de E/S. Sempre que um desses processos quiser a UCP, ele deve obtê-la imediatamente, para que possa iniciar sua próxima requisição de E/S, e deixá-la sendo feita em paralelo com outro processo realmente processando. 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 somente 2 ms do seu *quantum* de 100 ms ganharia uma prioridade 50, enquanto um processo que executou durante 50 ms antes de bloquear ganharia prioridade 2, e um processo que utilizou todo o *quantum* ganharia uma prioridade 1.

É freqüentemente conveniente agrupar processos em classes de prioridade e utilizar escalonamento com prioridades entre as classes, mas *round robin* dentro de cada classe. Por exemplo, em um sistema com quatro classes de prioridade, o escalonador executa os processos na classe 4 segundos 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 houverem 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).

4.3.4 Filas Multi-nível com retorno

Quando um processo ganha a UCP, especialmente quando ele ainda não pôde estabelecer um padrão de comportamento, o escalonador não tem idéia da quantidade de tempo de UCP que ele precisa e que precisará. Processos I/O bound geralmente usam a UCP brevemente antes de gerar em pedido de I/O. Processos CPU bound poderiam utilizar a UCP por horas se ela estivesse disponível para eles em um ambiente não preemptivo.

Um mecanismo de escalonamento deveria:

- favorecer pequenos *jobs*;
- favorecer *jobs* I/O bounds para atingir uma boa utilização dos dispositivos de E/S; e
- determinar a natureza de um *job* tão rápido quanto possível e escalonar o *job* de acordo.

Filas multi-nível com retorno (*Multilevel feedback queues*) fornecem uma estrutura que atinge esses objetivos. O esquema é ilustrado na Figura 24:

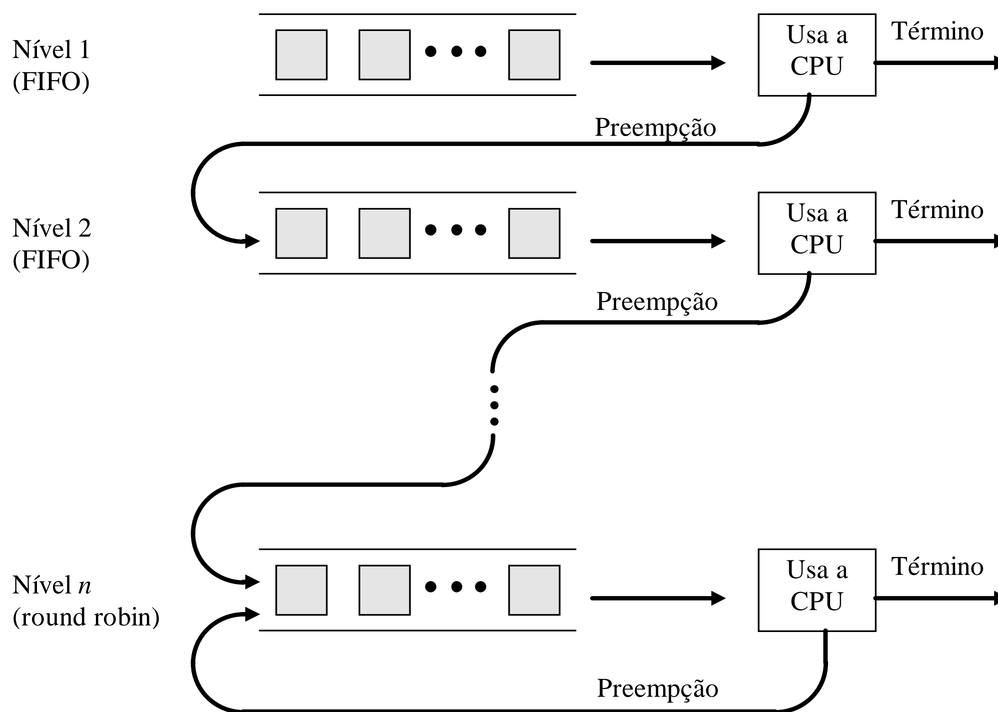


Figura 24 - Filas Multinível com Retorno

Um novo processo entra na rede de filas ao final da fila do topo. Ele se move através desta fila segundo uma política FIFO até que ganhe a UCP. Se o *job* termina ou desiste da UCP para esperar um término de E/S ou outro evento, ele deixa a rede de filas. Se o *quantum* expira antes do processo voluntariamente desistir da UCP, o processo é colocado de volta no final da fila um nível abaixo. O processo avança nesta fila, e em algum momento atinge a cabeça da fila. No momento em que não houver processos na primeira fila, ele ganha a UCP novamente. Se ele ainda utiliza todo o *quantum*, ele vai descendo para as filas de níveis inferiores. Normalmente, a fila de nível mais baixo possui uma política *round robin* para que todos os processos terminem de executar de uma maneira ou outra.

Em muitos sistemas de filas multi-nível, o *quantum* dado ao processo conforme ele se move para as filas de níveis inferiores é aumentado. Assim, quanto mais um processo permanece no sistema de filas, maior o seu *quantum*. Entretanto ele passa a não ganhar a UCP com tanta freqüência, porque as filas superiores possuem prioridade maior. Um processo em uma dada fila não pode executar até que as filas superiores estejam vazias. Um processo em execução é suspenso em favor de um processo que chegue em

uma fila superior.

Considere como tal mecanismo responde a diferentes tipos de processos. O mecanismo deveria favorecer processos limitados por E/S para atingir boa utilização dos dispositivos e bons tempos de resposta aos usuários interativos. Realmente isso funciona porque um processo limitado por E/S vai entrar na primeira fila e rapidamente ganhar a UCP. O *quantum* da primeira fila é ajustado para que a maioria dos *jobs* limitados por E/S tenham tempo de fazer sua requisição de E/S. Quando o processo faz a requisição de E/S, ele deixa a rede de filas, tendo recebido o tratamento desejado.

Agora considere um processo limitado por UCP que necessita de um grande tempo de UCP. Ele entra a rede de filas no nível mais alto, recebendo rapidamente seu primeiro *quantum* de UCP, mas quando ele expira, o processo é movido para a fila inferior. Agora o processo tem prioridade inferior aos da fila superior, mas eventualmente ele recebe a UCP, ganhando um *quantum* maior do que o anterior. Conforme o processo ainda precise de UCP, ele vai caminhando pelas filas, até chegar à fila de mais baixo nível, onde ele circula por uma fila *round robin* até que tenha terminado.

Filas Multi-nível com retorno são ideais para separar processos em categorias baseadas na sua necessidade por UCP. Em um sistema de tempo compartilhado, cada vez que o processo deixe a rede de filas, ele pode ser marcado com a identificação do nível da fila onde ele esteve pela última vez. Quando o processo reentra no sistema de filas, ele pode ser enviado diretamente para a fila onde ele anteriormente completou sua execução, de forma que um processo retornando para as filas não interfira no desempenho dos processos das filas de níveis mais altos.

Se processos são sempre colocados de volta na rede de filas no nível mais alto que eles ocuparam da última vez que estiveram no sistema de filas, será impossível para o sistema responder a mudanças no processo, como por exemplo, deixando de ser limitado por UCP para ser limitado por E/S. Este problema pode ser resolvido marcando também o processo com o seu tempo de permanência na rede de filas na última vez em que lá esteve. Assim, quando o processo reentra no sistema de filas, ele pode ser colocado no lugar correto. Dessa forma, se o processo está entrando em uma nova fase na qual ela deixa de ser limitado por UCP para ser limitado por E/S, inicialmente ele vai sofrer uma penalidade pelo sistema, mas da próxima vez o algoritmo perceberá a mudança de comportamento do processo. Uma outra maneira de responder a mudanças no comportamento de um processo é colocá-lo em um nível de filas cima do qual esteve se ele voluntariamente desistir da UCP antes do término do seu *quantum*.

O mecanismo de filas multi-nível com retorno é um bom exemplo de um **mecanismo adaptativo**, que responde a mudanças de comportamento do sistema que ele controla. Mecanismos adaptativos normalmente requerem um maior *overhead* do que os não adaptativos, mas a sensibilidade a mudanças torna o sistema mais ágil e justifica o *overhead* adicional.

Uma variação comum deste algoritmo é ter os processos circulando em várias filas *round robin*. O processo circula pela primeira fila um certo número de vezes, depois desce um nível, circulando um número maior de vezes, e assim por diante.

4.3.5 Escalonamento Menor tarefa Primeiro

Menor tarefa primeiro (*Shortest-job-first*) 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. O escalonamento de menor tarefa primeiro reduz o tempo médio de espera sobre o algoritmo FIFO. Entretanto, os tempos de espera têm uma variância muito grande (são mais imprevisíveis) do que no algoritmo FIFO, especialmente para grandes tarefas.

O escalonamento de menor tarefa primeiro favorece as tarefas pequenas em prejuízo dos *jobs* maiores. Muitos projetistas acreditam que quanto mais curta a tarefa, melhor serviço ele deveria receber. Não há um consenso universal quanto a isso, especialmente quando prioridades de tarefas devem ser consideradas.

O escalonamento de menor tarefa primeiro seleciona a tarefa para serviço de uma maneira que garante que a próxima tarefa irá completar e deixar o sistema o mais cedo possível. Isto tende a reduzir o número de tarefas esperando, e também reduz o número de tarefas esperando atrás de grandes tarefas.

Como resultado, o escalonamento de menor tarefa primeiro pode minimizar o tempo médio de espera conforme eles passam pelo sistema.

O problema óbvio com o escalonamento de menor tarefa primeiro é que ele requer um conhecimento preciso de quanto tempo uma tarefa demorará para executar, e esta informação não está usualmente disponível. O melhor que o escalonamento de menor tarefa primeiro pode fazer é se basear na estimativa do usuário de tempo de execução. Em ambientes de produção onde as mesmas tarefas rodam regularmente, pode ser possível prover estimativas razoáveis. Mas em ambientes de desenvolvimento, os usuários raramente sabem durante quanto tempo seus programas vão executar.

Basear-se nas estimativas dos usuários possui uma ramificação interessante. Se os usuários sabem que o sistema está projetado para favorecer tarefas 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 rodar a tarefa pelo tempo estimado mais uma pequena percentagem 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 completude de sua tarefa. Outra solução é rodar 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 sub-estimados pode pagar um preço alto por isso.

O escalonamento de menor tarefa primeiro , 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. A Figura 25mostra exemplifica dois escalonamentos quando há vários *jobs* com prioridades iguais e com tempo de processamento conhecido inicialmente (processo A = 8 minutos; processo B = 4 minutos; processo C = 4 minutos; processo D = 4 minutos). Inicialmente eles são escalonados para serem executados seqüencialmente - FIFO (tempo médio = 14 minutos); e em seguida os menores são colocados serem executados primeiro (tempo médio = 11 minutos). Assim, o escalonamento de menor tarefa primeiro conduz a resultados ótimos quando todos os *jobs* estiverem disponíveis ao mesmo tempo.

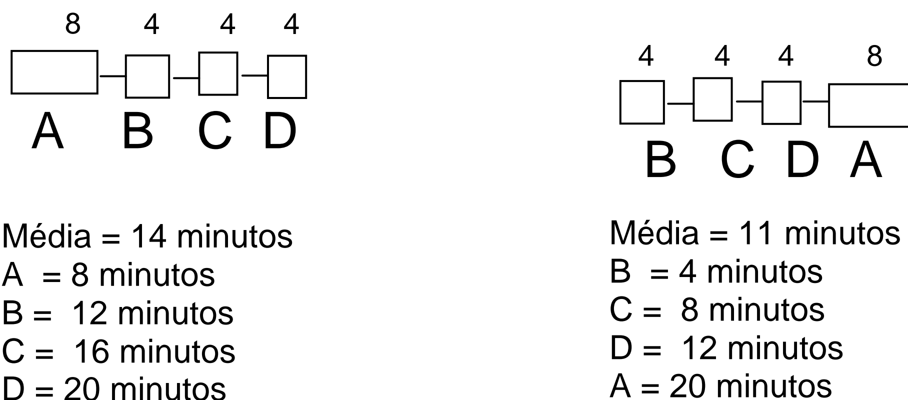


Figura 25- ***Shortest-job-first***

4.4 Exercícios - Revisão

1. Quais as diferenças entre interrupções de *hardware* e de *software*?
2. Explique todos os passos que ocorrem quando há uma interrupção de *hardware*.
3. Cada processo possui o seu próprio bloco de controle de processo ? Se sim, explique as informações contidas nele.
4. Explique os passos que ocorrem em uma troca de contexto.
5. Explique o que é um processo limitado por E/S (I/O bound) de processo limitado por computação (UCP-bound). Qual dos dois deve receber prioridade de escalonamento ? Explique.
6. Quais são os possíveis estados de um processo ? Explique as mudanças de estado.

7. Qual a função do escalonador de processos ?
8. Explique os seguintes algoritmos de escalonamento e as suas vantagens/desvantagens : FIFO, Round Robin, Prioridades, Filas-múlti-nível com retorno, Escalonamento com prazos e o de Menor tarefa primeiro.
9. Explique os critérios que devem ser considerados na escolha de um algoritmo de escalonamento.
10. Como o sistema operacional utiliza o relógio interno para prevenir que um processo monopolize o sistema?

4.5 Comunicação e Sincronização entre Processos.

É comum processos que executam concorrentemente (em sistemas multiprogramado ou multiprocessador) compartilharem recursos do sistema, como arquivos, registradores, memória, processadores e outros dispositivos. Esses recursos são compartilhados por dois motivos: ou por disputa/competição ou por que são utilizados como meio comum para comunicação (*buffers*) entre processos cooperantes.

Seja qual for o motivo pelo qual o recurso está sendo compartilhado, deverá existir uma maneira de sincronização dos processos concorrentes para garantir a integridade dos dados durante o acesso simultâneo. Para realizar esta sincronização, os processos terão que de certa forma trocarem algumas informações.

Uma maneira simples e geral de realizar esta sincronização está na restrição de permitir, somente a um processo por vez, obter ou utilizar o recurso compartilhado. Esta restrição é conhecida como exclusão mútua, e os trechos do código de cada processo que usam o recurso compartilhado e que são executados um por vez são denominados seções críticas ou regiões críticas.

Uma vez que existam métodos de sincronização que garantam a exclusão mútua sobre recursos compartilhados, pode-se então implementar mecanismos de comunicação, utilizando *buffer*(s) para a troca de informações. De certa forma, esta comunicação entre os processos é também uma forma de sincronização, em um nível de abstração maior, relacionada com o andamento dos processos.

Pode-se então generalizar e dizer que a comunicação entre processos está associada a duas formas de sincronização: uma relacionada com o acesso controlado às variáveis compartilhadas (disputa/competição), e outra, mais geral, relacionada como o andamento dos processos (cooperação). Para todas as formas de mecanismos que possam ser utilizados para qualquer tipo de sincronização ou para qualquer tipo de comunicação são denominadas de: mecanismos de comunicação e sincronização entre processos.

Questões

1. O que é uma variável compartilhada, por que ela existe e qual o problema gerado em torna dela?
2. Qual a relação que existe entre processos cooperantes e exclusão mútua?

4.5.1 O Problema do Produtor e Consumidor sobre um Buffer Circular.

Para exemplificar as implicações do compartilhamento de recursos para processos concorrentes, será apresentado o problema do produtor e consumidor. Neste problema existe um processo produtor e de um processo consumidor, ambos atuando concorrentemente sobre um buffer compartilhado (Figura 26).

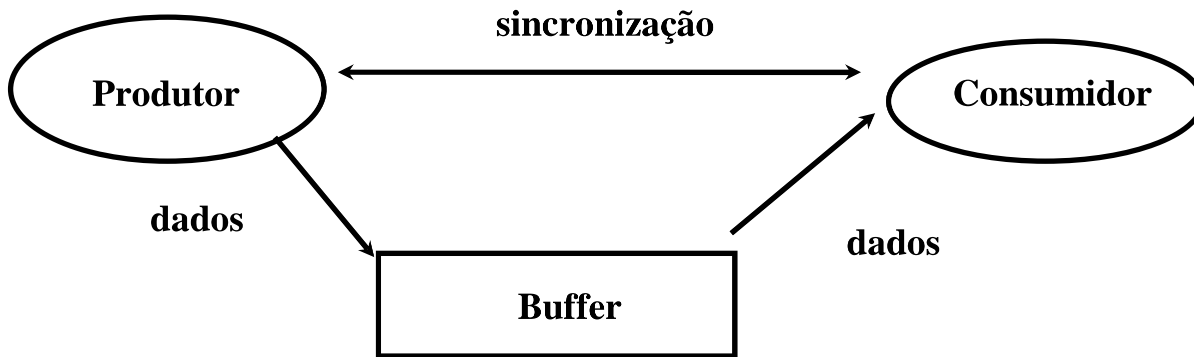


Figura 26 -Produtor x Consumidor

Como o produtor e o consumidor poderão ser executados concorrentemente, então existe a possibilidade de os dois tentarem realizar simultaneamente depósito e retirada de mensagens no buffer, assim algumas restrições devem ser impostas:

1. O produtor não poderá exceder a capacidade finita do buffer;
2. Se o produtor tentar depositar uma mensagem no buffer cheio, ele será suspenso até que o consumidor retire pelo menos uma mensagem do buffer;
3. Se o consumidor tentar retirar uma mensagem do buffer vazio, ele será suspenso até que o produtor deposite pelo menos uma mensagem no buffer.

Desta maneira, o acesso ao *buffer* (recurso compartilhado) será feito pelas operações depositar e retirar colocadas respectivamente nos processos produtor e consumidor. Essas operações deverão estar associadas com mecanismos de comunicação e sincronização de forma tal que as restrições estabelecidas pelo problema sejam plenamente satisfeitas.

4.5.1.1 Problemas de Compartilhamento de Recursos

4.5.1.1.1 Compartilhamento de um arquivo em disco

Este programa lê o registro do cliente no arquivo (Reg_Cliente), lê o valor a ser depositado ou retirado e atualiza o saldo

```
Read (Arq_contas, Reg_cliente);
Readln (valor_Dep_Ret);
Reg_cliente.Saldo:= Reg_cliente.Saldo+Valor_Dep_Ret;
Write (Arq_Contas, Reg_Cliente);
```

Agora, imaginem dois caixas diferentes atualizando o saldo de um mesmo cliente simultaneamente.

Caixa	comando	saldo arquivo	valor dep/ret	saldo memória
1	read	1.000	*	1.000
1	readln	1.000	-200	1.000
1	:=	1.000	-200	800
2	read	1.000	*	1.000
2	readln	1.000	300	1.000
2	:=	1.000	300	1.300
1	write	800	-200	800
2	write	1.300	300	1.300

4.5.1.1.2 Compartilhamento de uma variável em memória

Suponha dois processos A e B estejam executando um comando de atribuição, onde o processo A some 1 à variável X, e o processo B diminua 1 da mesma variável que está sendo compartilhada. Considere inicialmente X=2

```
Processo A        Processo B
```

```
x:= x+1;              x:= x-1
load x, r1            load x, r2
add 1, r1             sub 1, r2
store r1, x           store r2, x
```

```
Processo        Comando         x     r1    r2
A               load x, r1      2     2     *
A               add 1, r1       2     3     *
B               load x, r2      2     *     2
B               sub 1, r2       2     *     1
A               store r1, x           3     3     *
B               store r2, x           1     *     1
```

Em qualquer situação, onde dois ou mais processos tenham acesso a um mesmo recurso compartilhado, devam existir mecanismos de controle que evitem esses tipos de problemas. A parte do programa onde é necessária a exclusão mútua, ou seja, onde o dado compartilhado é acessado é denominada região crítica. A exclusão mútua consiste em impedir que dois ou mais processos acessem algum dado compartilhado simultaneamente

4.5.2 Soluções de *hardware*.

Antes das soluções de *software* para o problema da exclusão mútua serem examinadas, serão apresentadas duas soluções de *hardware* para o mesmo problema. As soluções de *hardware* são importantes porque criam mecanismos que permitem a implementação das soluções de *software*.

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

```
BEGIN
  Desabilita_Interrupções;
  Região_Crítica;
  Habilita_Interrupções;
END;
```

Esse mecanismo é inconveniente por vários motivos. O maior deles acontece, quando o processo que desabilitou as interrupções não torna a habilitá-las. Nesse caso, o sistema provavelmente terá seu funcionamento seriamente comprometido. No entanto, pode ser útil ao sistema operacional, quando ele necessita manipular estruturas de dados compartilhadas do sistema, como listas de processos. Dessa forma, o sistema garante que não ocorrerão problemas de inconsistência em seus dados.

Uma outra opção, é a existência de uma instrução especial, que permite ler uma variável, armazenar seu conteúdo em uma outra área e atribuir um novo valor a essa variável. Esse tipo de instrução é chamada ***test-and-set***, e é caracterizada pela execução sem interrupção, ou seja, é uma instrução indivisível ou atômica. Assim, não existe a possibilidade de dois processos estarem manipulando uma variável compartilhada ao mesmo tempo, possibilitando a implementação da exclusão mútua.

Na execução da instrução ***test-and-set*** o valor lógico da variável **Y** é copiado para **X**, e é atribuído para à variável **Y** o valor lógico ***true*** (verdadeiro). Para coordenar o acesso concorrente a um recurso, esta instrução utiliza uma variável lógica global denominada Bloqueio. Quando a variável Bloqueio for ***false*** (falsa), qualquer processo pode alterar seu valor para verdadeiro, através da instrução ***test-and-set*** e assim, acessar ao recurso de forma exclusiva. Ao terminar o acesso, o processo deve simplesmente retornar o valor da variável para falso, liberando o acesso ao recurso.

```
Program Programa_Test_and_set;
Var  Bloqueio: Boolean;
     Procedure Processo_A;
```

```
                    Var Pode_A: Boolean;
                             Begin
                        Repeat
                            Pode_A:=true;
                            While (Pode_A) Do
                          test_and_set (Pode_A,Bloqueio);
                            Regiao_Critica_A;
                            Bloqueio:=False;
                        Until True;
               End;

                    Procedure Processo_B;
                    Var  Pode_B; Boolean;
                    Begin
                     Repeat
                            Pode_B:=true;
                            While (Pode_B) Do
                          test_and_set (Pode_B,Bloqueio);
                            Regiao_Critica_B;
                            Bloqueio:=False;
                        Until True;
                    End;


               Begin
                   Bloqueio:= False;
                   Cobegin
                       Processo_A;
                       Processo_B
                   Coend;
               End.
```

Num ambiente mutiprogramado, a inibição de interrupção ao longo de um trecho de programa é suficiente para implementar exclusão mútua no acesso à memória compartilhada, eliminando-se temporariamente a possibilidade de comutação de processos. Já num ambiente multiprocessador, torna-se necessário o uso de um mecanismo do tipo ***test_and_set***, uma vez que as interrupções, atuando individualmente em cada processador, não conseguem impor uma limitação afetiva no acesso à memória compartilhada.

4.5.3 Soluções de *software*

Além da exclusão mútua, que soluciona os problemas do compartilhamento de recursos, existem alguns fatores fundamentais que deverão ser atendidos:

- Dois ou mais processos não podem estar simultaneamente dentro de suas regiões críticas correspondentes
- O número de processadores e o tempo de execução dos processos devem ser irrelevantes.
- Nenhum processo que esteja rodando fora de sua região crítica pode bloquear a execução de outro processo
- Nenhum processo pode ser obrigado a esperar indefinidamente para entrar na região crítica

Um dos principais problemas existentes nas soluções já apresentadas é o problema da **espera ocupada** (*busy wait*). Na espera ocupada, toda vez que um processo tenta entrar em sua região crítica e é impedido, por já existir outro processo acessando o recurso, ele fica em ***looping***, testando uma condição, até que lhe seja permitido o acesso. Dessa forma, o processo bloqueado consome tempo do processador desnecessariamente.

A solução para o problema da espera ocupada foi a introdução de comandos (primitivas) que permitissem que um processo, quando não pudesse entrar em sua região crítica, fosse colocado no estado

de espera, sem consumir processador, até que outro processado o liberasse. Esse mecanismo passou a ser adotado nas soluções seguintes.

4.5.3.1 Semáforos

Semáforo é um mecanismo de comunicação e sincronização que envolve a utilização de uma variável compartilhada, denominada semáforo (**S**), e de duas operações primitivas indivisíveis que atuam sobre ela (**DOWN** e **UP**). A variável **S** poderá assumir valores inteiros não negativos. Essa variável deverá ter um valor inicial maior ou igual a zero e não poderá ser manipulada a não ser por meio de suas primitivas.

Essas primitivas poderão ser implementadas de duas formas: utilizando espera ocupada (***busy wait***) sobre a primitiva **DOWN** ou utilizando uma fila de espera (**Q**) associada ao semáforo **S**. No caso de haver mais de um semáforo, cada semáforo $\mathbf{S_i}$ teria uma fila $\mathbf{Q_i}$ associada a ele.

No primeiro caso, as primitivas seriam definidas do seguinte modo:

```
DOWN(Var Si: Int)
Begin
   While Si<=0 Do ;   (* não faz nada *)
   Si:=Si-1;
End;

UP(Var Si: Int)
Begin
   Si:=Si+1;
end;
```

No segundo caso, as primitivas seriam definidas do seguinte modo:

```
DOWN(Var Si: Int)
Begin
   If Si>0
      Then Si:=Si-1                  (* E o processo continua. *)
      Else  Suspende_Processo(Qi);  (* suspende o processo que executou *)
End;                                (* esta primitiva,  colocando  em  uma *)
                                    (* fila de espera Qi                   *)
UP(Var Si: Int)
Begin
   If Fila_Vazia(Qi)
      Then Si:=Si+1
      Else Acorda_Processo(Qi);     (* retira o processo que estava *)
End;                                (* suspenso na fila Qi e coloca na *)
                                    (* fila dos prontos *)
```

Exemplo 1: Este exemplo mostra uma solução para o problema da exclusão mútua entre dois processos através do uso de semáforos. Inicialmente o semáforo está com o valor **1**, indicando que nenhum processo está executando sua região crítica. Se o **Processo_A** executar a instrução **DOWN**, faz com que o semáforo seja decrementado de **1**, passando a ter o valor **0**. Em seguida, o **Processo_A** ganha o acesso a região crítica. O **Processo_B** também executará a instrução **DOWN**, mas como seu valor é igual a zero, ficará aguardando até que o **Processo_A** execute a instrução **UP**, ou seja, volte o valor do semáforo

para **1**. Seja qual for o processo que entrar primeiro na região crítica, o problema da exclusão será resolvido.

```
Program Testa_Semáforo_1;
Var S: Semáforo := 1;          (* inicializando o semáforo *)
   Procedure Processo_A;
   Begin
      Repeat
            DOWNt(s);
            Região_Crítica_A;
            UP(s);
       Until False;
   End;

   Procedure Processo_B;
   Begin
      Repeat
          DOWN(s);
          Região_Crítica_B;
          UP(s);
      Until False;
   End;

Begin
   Cobegin
      Processo_A;
      Processo_B;
   Coend;
End.
```

Exemplo 2: Aqui será apresentada uma solução para o problema do Produtor e Consumidor. O Programa utiliza 3 semáforos, sendo que um é utilizado para implementar a exclusão mútua e 2 para a sincronização condicional. O semáfaro **Mutex** permite a execução das regiões críticas Depositar e Retirar de forma mutuamente exclusiva. Os semáforos **Vazio** e **Cheio** representam, respectivamente, se há posições livres no *buffer* para serem depositadas e posições ocupadas a serem retiradas.

Quando o semáforo **Vazio** for igual a 0 (zero), significa que o *buffer* está cheio e o processo produtor deverá aguardar até que o consumidor retire algum dado. Da mesma forma, quando o semáforo **Cheio** for igual a 0 (zero), significa que o buffer está vazio e o consumidor deverá aguardar até que o produtor grave algum dado. Em ambas as situações, o semáforo demonstra a impossibilidade de acesso ao recurso.

```
Program Produtor_Consumidor;
Const TamBuf = 2;
Type        Tipo_Dado = (* tipo qualquer *);
Var         Vazio : semáforo := TamBuf;
            Cheio : semáforo := 0;  (* Inicialização do semáforo *)
            Mutex : Semáforo := 1;
                  Buffer : Array[1..TamBuf] of Tipo_Dado;

Procedure Produtor;
      Var Dado : Tipo_Dado;
      Begin
          Repeat
            :    :
            Produz_Dado(Dado);
```

```
            DOWN(Vazio);
            DOWN(Mutex);
            Depositar(Dado,Buffer);
            UP(Mutex);
            UP(Cheio);
          Until False;
      End;

      Procedure Consumidor;
      Var Dado : Tipo_Dado;
      Begin
          Repeat
            DOWN(Cheio);
            DOWN(Mutex);
            Retirar(Dado,Buffer);
            UP(Mutex);
            UP(Vazio);
            Consome_Dado(Dado);
            :    :
          Until False;
      End;

Cobegin
    Produtor;
          Consumidor;
Coend;
```

4.5.3.2 Monitores

São mecanismos de sincronização compostos de um conjunto de procedimentos, variáveis e estrutura de dados definidos dentro de um módulo cuja finalidade é a implementação automática da exclusão mútua entre seus procedimentos. Somente um processo pode estar executando um dos procedimentos do monitor em um determinado instante. Toda vez que um processo chamar um destes procedimentos, o monitor verifica se já existe outro processo executando algum procedimento do monitor. Caso exista, o processo fica aguardando a sua vez até que tenha permissão para executá-lo.

A implementação da exclusão mútua nos monitores é realizada pelo compilador do programa e não mais pelo programador. Para isto ele irá colocar todas as regiões críticas do programa em forma de procedimentos no monitor e o compilador se encarregará de garantir a exclusão mútua destes procedimentos. A comunicação do processo com o monitor passa a ser feita através de chamadas a seus procedimentos e dos parâmetros passados para eles.

Outra característica do monitor é que os processos, quando não puderem acessar estes procedimentos, ficarão aguardando em uma fila de espera e enquanto isto, eles poderão executar outros procedimentos. Como mostrado no código a seguir o monitor não pode ser executado como uma subrotina comum. A comunicação do processo com o monitor é feita unicamente através de chamadas aos seus procedimentos e dos parâmetros passados para eles. Neste exemplo, a inicialização da variável compartilhada X com o valor zero só acontecerá uma vez (no momento da primeira ativação do monitor Região_Crítica). O valor final de X será 0.

```
PROGRAM Exemplo;
        MONITOR Região_Crítica;
        VAR X: INTEGER;
 PROCEDURE Soma;
            BEGIN
                X:=X+1;
             END;
 PROCEDURE Diminui;
            BEGIN
                X:=X-1;
```

```
             END;
BEGIN
    x:=0;
     PARABEGIN
            Região_Critica.Soma;
            Região_Crítica.Diminui;
    PARAEND;
END.
```

4.5.3.3 Troca de Mensagens.

A troca de mensagens é um mecanismo de comunicação e sincronização que exige do sistema operacional tanto a sincronização quanto a comunicação entre processos, ao contrário do que acontece com os mecanismos já considerados, que exigem do sistema operacional somente as primitivas de sincronização, deixando com o programador a sua utilização na sincronização e a comunicação de mensagens através de memória compartilhada.

Este método de comunicação entre processos usa as primitivas *send (envia)* e *receive (recebe).* As chamadas send e receive são normalmente implementadas como chamadas ao sistema operacional. Estas primitivas são representadas da seguinte forma:

```
send ( Receptor , Mensagem );
receive ( Transmissor , Mensagem );
```

A solução do problema do produtor/consumidor utilizando troca de mensagens poderá ser visualizado no algoritmo abaixo:

```
program Produtor_Consumidor;
      procedure Produtor;
      var Msg: Tipo_Msg;
      begin
          repeat
                  :
              Produz_Mensagem(Msg);
              send(Msg);
          until false;
      end;

      procedure Consumidor;
      var Msg: Tipo_Msg;
      begin
          repeat
              receive(Msg);
              Consome_Mensagem(Msg);
                  :
          until false;
      end;

Cobegin
     Produtor;
```

```
        Consumidor;
    Coend.
```

4.5.4 Deadlock: Espera sem fim.

Um processo está em deadlock quando ele está suspenso à espera de uma condição que jamais irá acontecer. Por exemplo: Espera circular (Figura 27)

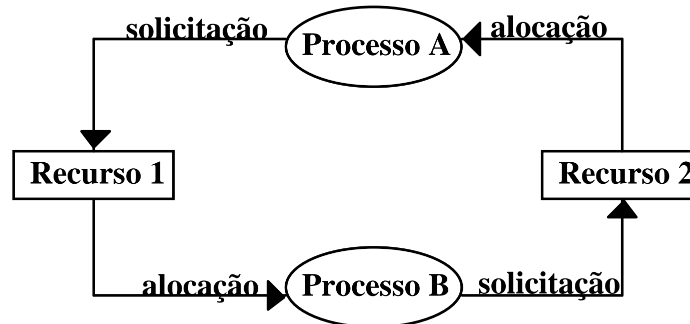


Figura 27 -Deadlock

4.5.5 Problemas clássicos

4.5.5.1 Problema dos Filósofos

Em uma mesa redonda estão sentados 5 filósofos. Para cada filósofo existe um prato. Entre cada 2 pratos existe um palito chinês, como mostra a Figura 28. Cada filósofo gasta seu tempo pensando ou comendo. Para comer cada filósofo necessita dos 2 palitos que cercam seu prato, o da esquerda e o da direita. Cada palito é compartilhado entre 2 filósofos. Faça um código genérico para o filósofo(i), i=1,2..5, de forma que eles possam executar concorrentemente, de forma sincronizada, que permita a todos pensar e comer, utilizando semáforo.

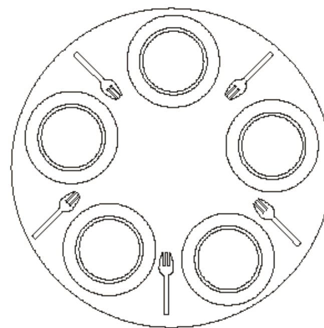


Figura 28 - Filósofos de glutões

4.5.5.1.1 Solução 1 (pode gerar deadlock) – errada !!!!!!

O procedimento pegar_garfo espera até que o garfo especificado esteja livre, e então aloca-o ao filósofo. Se os cincos filósofos resolverem comer ao mesmo tempo e se conseguirem o garfo a esquerda, quando forem pegar o garfo a direita acarretará uma situação de deadlock

4.5.5.1.2 Solução 2 (pode gerar starvation) – errada !!!!!!

Antes de pegar o garfo da esquerda, verificamos se o garfo da direita está livre. Se não estiver, ele deve liberar o garfo da esquerda, esperar um tempo e repetir todo o processo
Se todos os filósofos começarem a mesmo tempo, e notarem que o garfo a direita está ocupado, esperarão e retornarão o processo, e assim sucessivamente (*starvation*)

4.5.5.1.3 Solução 3 (livre de deadlock e starvation)

Utiliza-se um semáforo DOWN/UP, para proteger os cincos comandos que seguem o comando pensar. Antes de iniciar a aquisição dos garfos o filósofo precisa executar um DOWN. Após a devolução

de um garfo, o filósofo deve executar um UP. Do ponto de vista teórico é adequada, mas tem um *bug*, somente um filósofo pode comer por vez.

4.5.5.2 Problema dos Escritores e Leitores

Este problema modela acessos a uma base de dados. Suponha-ses que vários processos concorrentes possam ler ou escrever sobre um recurso compartilhado, respeitando as seguintes restrições:

- Quando um processo quiser escrever, somente 1 poderá fazê-lo e nenhum outro poderá ler ou escrever.
- Quando um processo quiser ler, qualquer outro poderá ler também, menos escrever.

Escreva o código para o processo leitor e o código para o processo escritor, usando semáforo, de modo que as restrições sejam satisfeitas e possam executar concorrentemente.

4.5.5.2.1 Solução

Quando o primeiro leitor tiver acesso à base ele executa um DOWN no semáforo do banco. Os leitores subseqüentes só incrementam o contador e os que deixarem decrementam o contador. O último que sair executa um UP. Assim, o escritor bloqueado, se houver algum, terá acesso a base. Pode-se implementar um esquema de prioridade para os Leitor (maior prioridade) para que eles não sofram *starvation*.

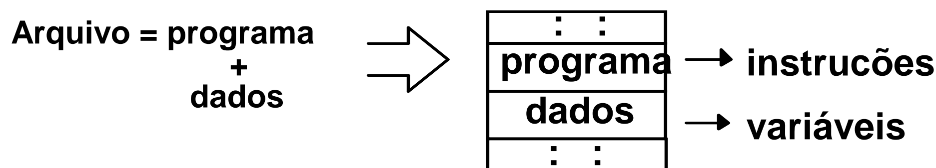
5 GERENCIA DE MEMÓRIA

A organização e o gerenciamento da memória real (também denominada principal, memória física ou memória primária) de um sistema de computador tem sido a principal influência sobre o projeto de sistemas operacionais. Armazenamento secundário – mais comumente disco e fita – fornece capacidade maciça e barata para a grande quantidade de programas e dados que devem ser mantidos prontamente disponíveis pra o processamento, porém é lento e não diretamente acessível por processadores. Para que sema executados e referenciados diretamente, programas e dados devem estar na memória principal.

Memória: - principal: volátil, pouca capacidade e muito rápida
- secundária: permanente, muita capacidade e lenta

Principal função: armazenar programas para a execução.

- o arquivo que está na memória secundária deve ser carregado para a memória principal para ser executado pelo processador.

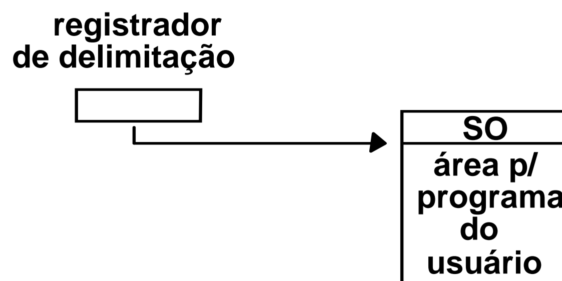


Tipos de Gerenciamento de Memória

- alocação contígua simples
- alocação particionada (estática e dinâmica)
- swapping
- paginação
- segmentação

5.1 Alocação Contígua Simples

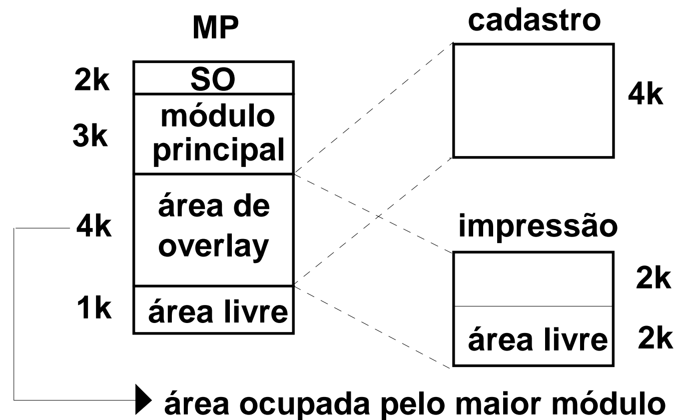
- para sistemas monoprogramáveis
- memória principal dividida em 2: SO + usuário
- usuário tem controle de quase toda a memória
- controle de acesso fora dos limites do usuário: registrador de delimitação



- caso o usuário não utilize toda a memória: espaço inutilizado
- quando o programa não cabe na memória principal
  -> dividir em módulos independentes: overlay (sobreposição)

Ex: -> Programa todo: - módulo principal - 3k
- módulo cadastro - 4k
- módulo impressão - 2k

-> memória - 10k
-> SO - 2k



- o programador especifica que módulos serão de overlay
- overlay: permite a expansão dos limites da memória principal

Pergunta: Em um sistema monoprogramado, dispondo de uma memória física de 10k, como pode ser executado um programa de 20k? Explique o método.

- somente se for utilizado técnica de overlay, de forma que o módulo principal mais o maior módulo de overlay não devem ultrapassar 10k de memória.

5.2 Alocação Particionada

- para sistemas multiprogramáveis
- memória é dividida em partições

5.2.1 Alocação Particionada Estática

- as partições são fixadas durante o boot, em função do número e tamanho dos programas
- controlado por uma tabela de partições

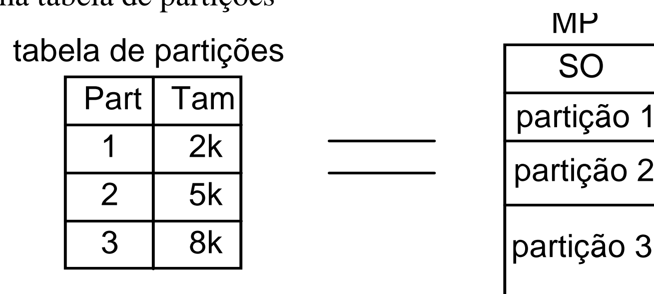
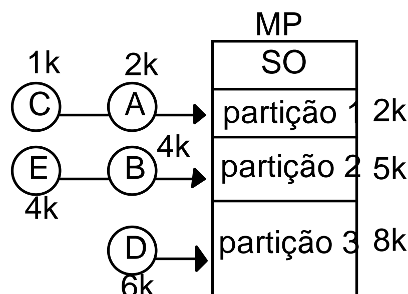


tabela de partições

Part	Tam
1	2k
2	5k
3	8k

- quando os compiladores só geravam código absoluto: A.P.E.Absoluta
  - -> um programa só poderia executar em uma partição pré-estabelecida

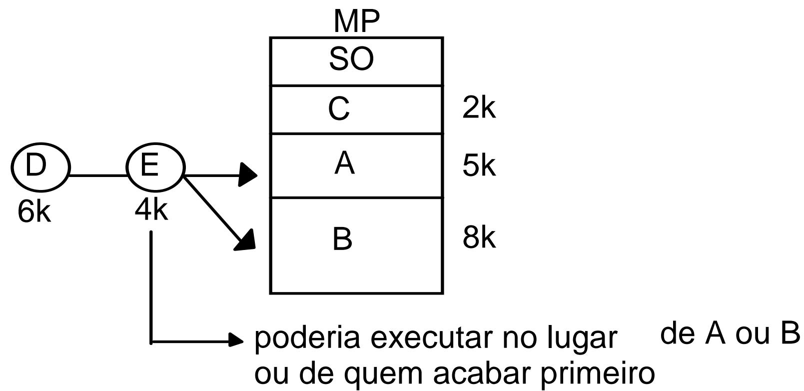
Ex:



- se por acaso os programas A e B estivessem executando e a 3o. partição estivesse livre, C e E não poderiam executar

- com a geração do código relocável: A.P.E.Relocável
  - -> os programas poderiam ser carregados em qualquer posição

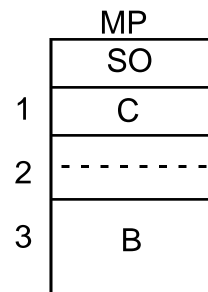
Ex:



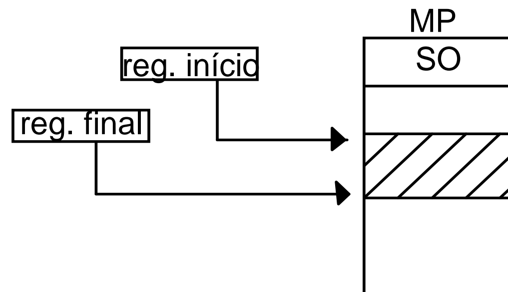
- uso de uma tabela de ocupação:

tabela de ocupação

Part	Tam	Ocp
1	2k	N
2	5k	S
3	8k	N

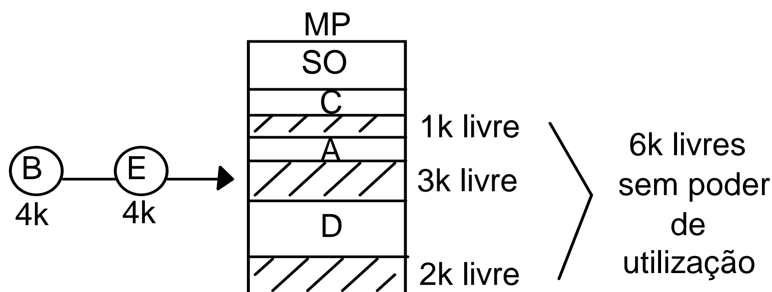


- utiliza 2 registradores delimitadores
  - -> para impedir acesso fora dos limites de cada usuário



- tanto a alocação absoluta quanto a relocável: os programas não preenchiam todos os espaços das partições -> áreas livres inutilizadas -> "fragmentação"

Ex:

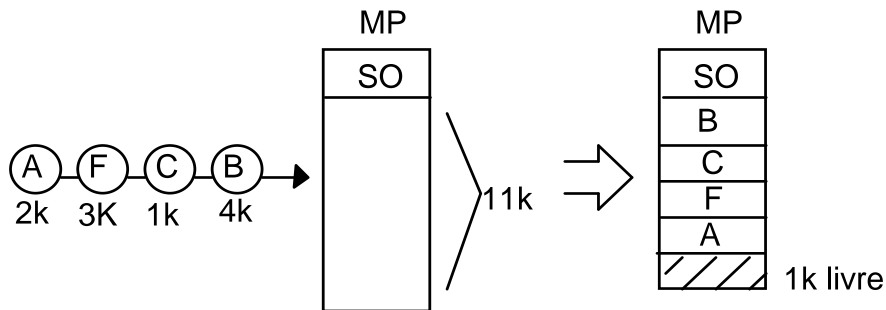


Pergunta: De que forma uma memória pode ficar totalmente preenchida (sem fragmentação) por processos, utilizando o método de Alocação Particionada Estática?

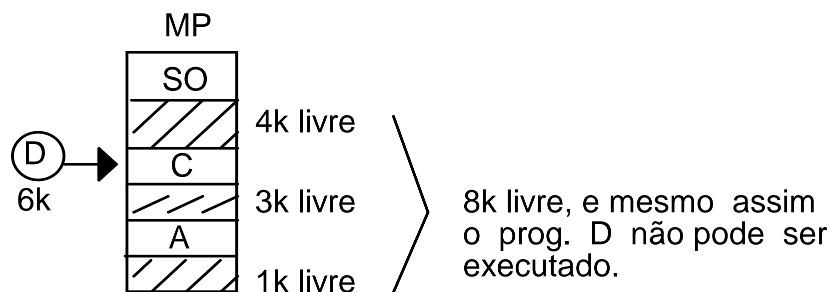
- Somente se por coincidência, os programas preencherem a memória.

5.2.2 Alocação Particionada Dinâmica

- sem partições fixas
- cada programa utiliza o espaço que precisa, se couber
- a partição seria do tamanho do próprio programa
- Ex:

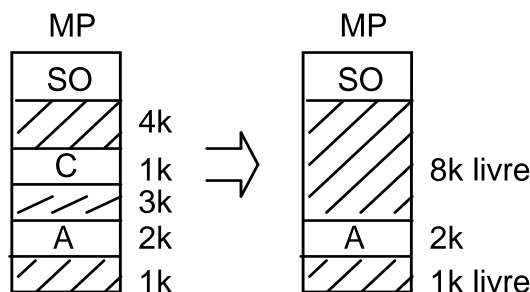


- quando os programas forem acabando aparecem a fragmentação
Ex:

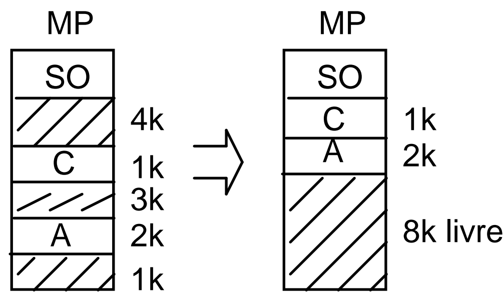


Soluções para Diminuir a Fragmentação

1) unir partições adjacentes



2) relocar as partes ocupadas, criando uma única área livre.



- possui algoritmos complexos, consumindo muito processador e disco, tornando-se inviável.

5.2.3 Estratégias para a escolha da Partição

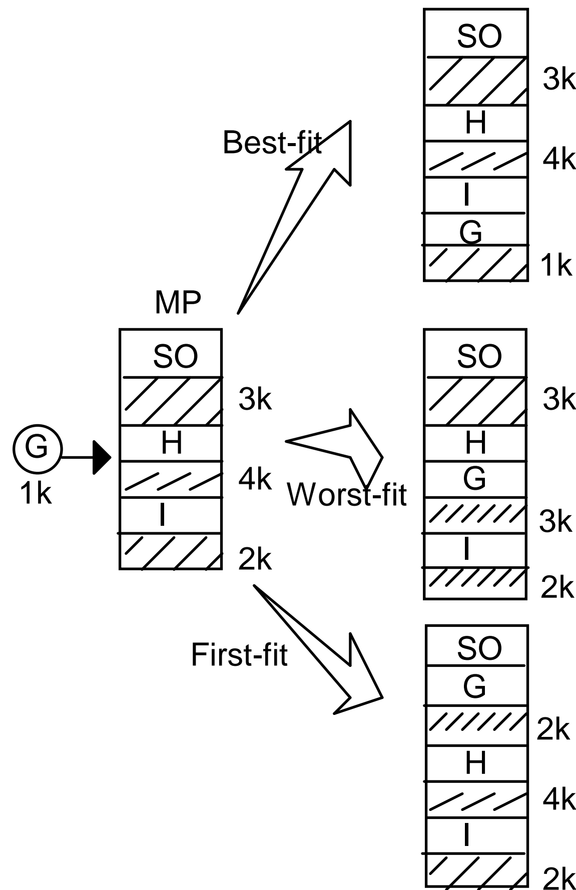
- tentar evitar ou diminuir o problema da "fragmentação"
- o SO deve possuir uma lista de áreas livres ou "free-list"
- 3 técnicas principais: best-fit, worst-fit e first-fit

1) Best-fit: escolhe a melhor partição, ou seja, aquela em que o programa deixa o menor espaço sem utilização.
- a tendência é que a memória fique cada vez mais com pequenas áreas livres não contíguas, aumentando o problema da partição

2) Worst-fit: escolhe a pior partição, ou seja, aquela em que o programa deixa o maior espaço sem utilização.
- deixando espaços maiores, a tendência é permitir que um maior número de programas utilize a memória, diminuindo o problema da fragmentação.

3) First-fit: escolhe a primeira partição livre que seja suficiente para carregar o programa. É a estratégia mais rápida entre elas.

Ex:



Pergunta: Supondo a situação abaixo, responda:

Alocação Particionada Dinâmica

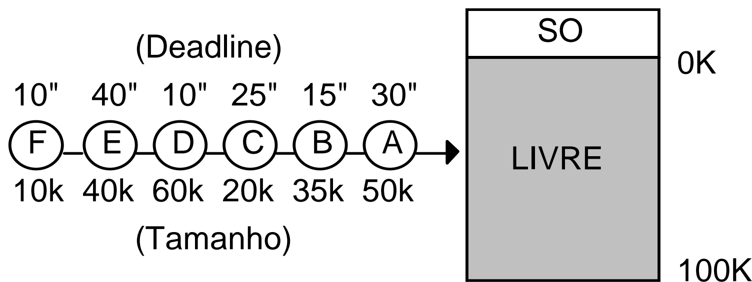


Tabela de Partições

Part	Inic	Fim
1		
2		
:	:	:
N		

time-slice = 5"
escalonamento a longo prazo = FIFO
estratégia p/ escolha da partição = worst-fit
escalonamento a curto prazo = round-robin
redução da fragmentação = união das partições adjacentes
obs: considere apenas o tempo de processamento

a) quanto tempo levará para executar todos os programas ?
b) após 65", qual a situação da memória e da tabela de partição ?
c) após 95", qual a situação da memória e da tabela de partição ?

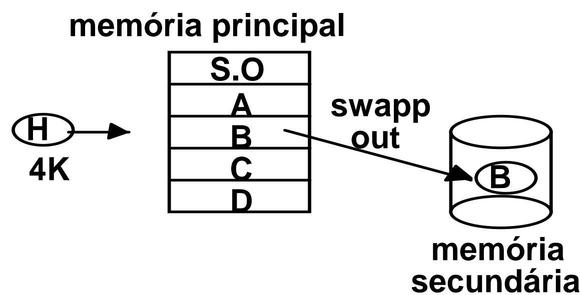
5.3 Swapping

-> Técnica para resolver o problema da insuficiência de memória

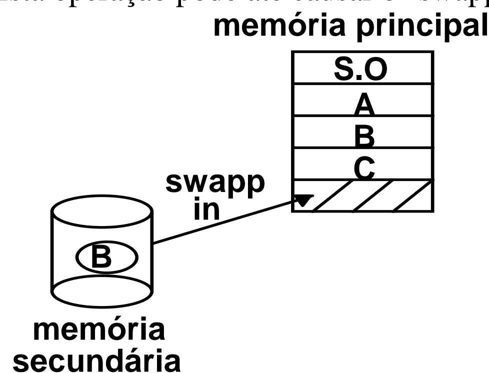
antes: O programa ficava na memória até o fim da sua execução, enquanto os outros esperavam por memória livre.

com swapping: O sistema retira temporariamente um programa da memória, coloca-o no disco (swapp out), para a entrada de outro.

obs: A escolha do programa a ser retirado depende da implementação. Pode-se por exemplo, retirar aquele que está bloqueado por algum motivo ou aquele que levará mais tempo para ser executado (último da lista dos prontos)



Quando o programa tiver que ser executado novamente, então ele é novamente carregado para a memória principal (swapp in). Esta operação pode até causar o "swapp out" de um outro programa.



Problema: A relocação (feito pelo loader) no swapp in torna o sistema ineficiente quando existe uma troca intensa de programas.

Solução: Implementação via hardware para a relocação automática em tempo de execução
-> "relocação dinâmica", através do registrador de relocação

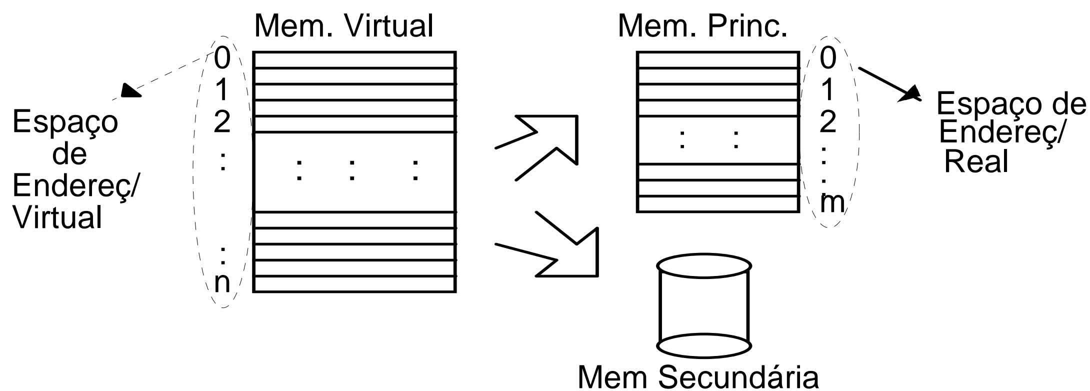
5.4 Memória Virtual

Combinação entre a memória principal e a memória secundária (disco) para dar ao usuário a ilusão de existir uma memória principal bem maior que na realidade.

Espaço de Endereçamento Virtual: Um programa no ambiente de memória virtual não faz referência a endereços físicos de memória (endereços reais), mas apenas a endereços virtuais (imaginários).

Durante a execução, o endereço virtual é mapeado para o endereço físico real da memória principal -> "mapeamento"

O conjunto de endereços virtuais que os processos podem endereçar é chamado de "espaço de endereçamento virtual", e o conjunto de endereços reais é chamado "espaço de endereçamento real".

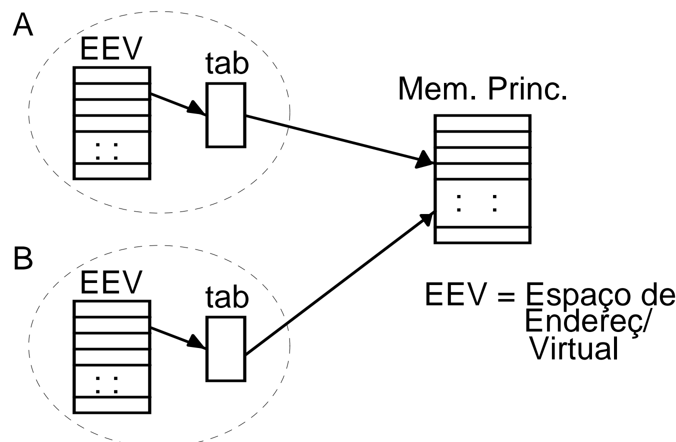


Obs: Como os programas podem ser muito maiores que a memória física, somente parte deles devem estar na memória física em um dado instante.

Obs: O usuário não se preocupa com o endereçamento, pois para ele, é como se a memória fosse contínua e inacabável. O SO + Compilador é que se preocupam com o mapeamento.

Mapeamento: Nos sistemas atuais, a tarefa de tradução é realizada por um hardware específico controlado pelo sistema operacional, de forma transparente para o usuário.

-> através de tabelas de mapeamento para cada processo.



Controle: Registrador que indica a posição inicial da tabela relativa ao processo que será executado.

Obs: As tabelas mapeiam blocos de informações, ou seja, cada bloco da memória virtual possui um bloco correspondentes na memória física.

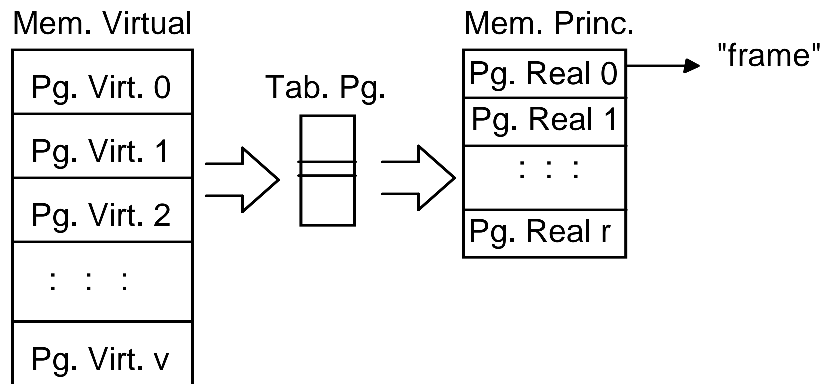
blocos menores -> - tabelas maiores
- maior tempo de acesso ao disco
- proporciona um grau maior de multiprogramação

blocos maiores -> - tabelas menores
- menor tempo de acesso ao disco
- proporciona um grau menor de multiprogramação

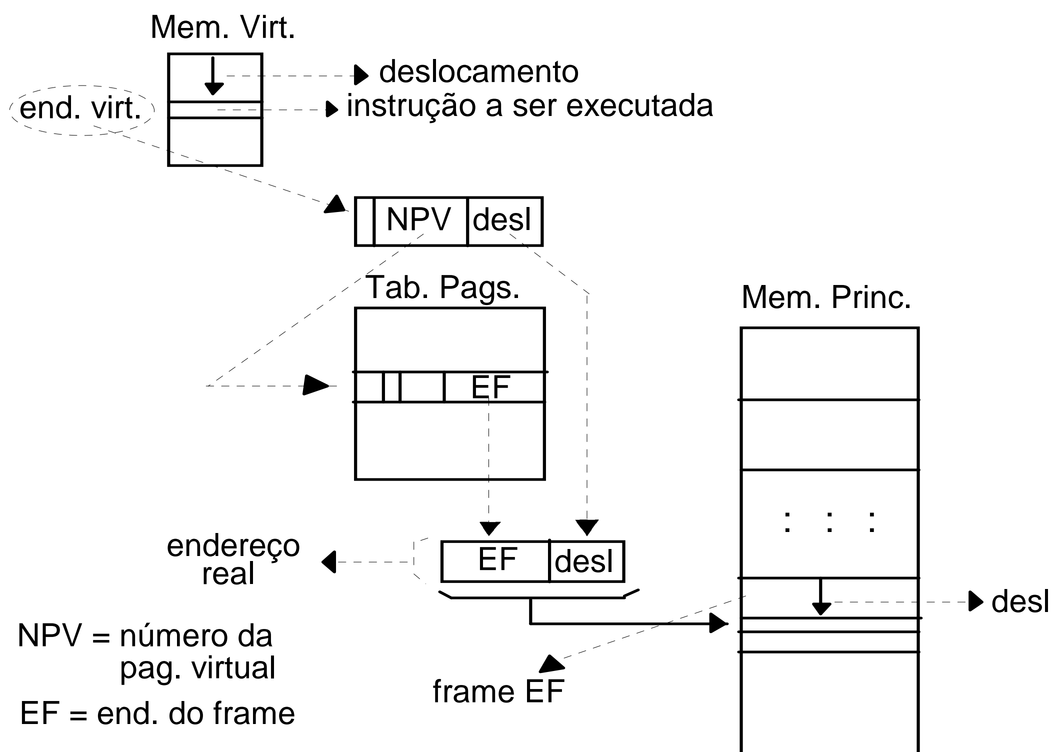
- blocos de tamanhos iguais -> Paginação
- blocos de tamanhos diferentes -> Segmentação

5.4.1 Paginação

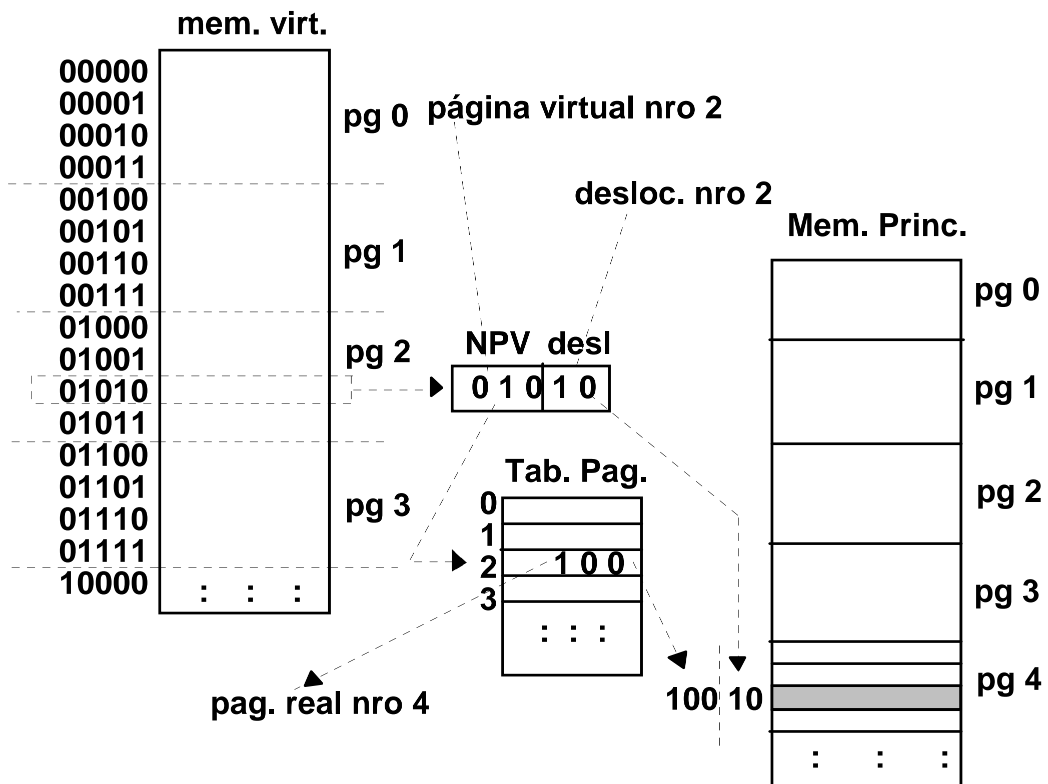
O espaço de endereçamento virtual e o espaço de endereçamento real são divididos em blocos de mesmo tamanho chamados "páginas".



Execução de Programas: Quando o processador tentar executar uma determinada instrução do programa do usuário (memória virtual), antes, ele deverá consultar a tabela de páginas para saber se a página que contém aquela instrução já está ou não na memória principal, e se estiver, em que posição real ela está. Se não estiver (page fault), a página deverá ser carregada para a memória principal.
Endereçamento: O processador tenta sempre executar um endereço virtual. Este endereço virtual contém duas partes: um número da página e um deslocamento dentro da página. O número da página serve para o processador consultar a tabela de páginas para saber qual a página real (endereço do frame) que a instrução desejada está. Então o endereço real definitivo é obtido concatenando o número da página real (endereço do frame) com o deslocamento. Note que o deslocamento é sempre o mesmo, tanto para páginas virtuais como para páginas reais, já que os tamanhos destas são iguais.

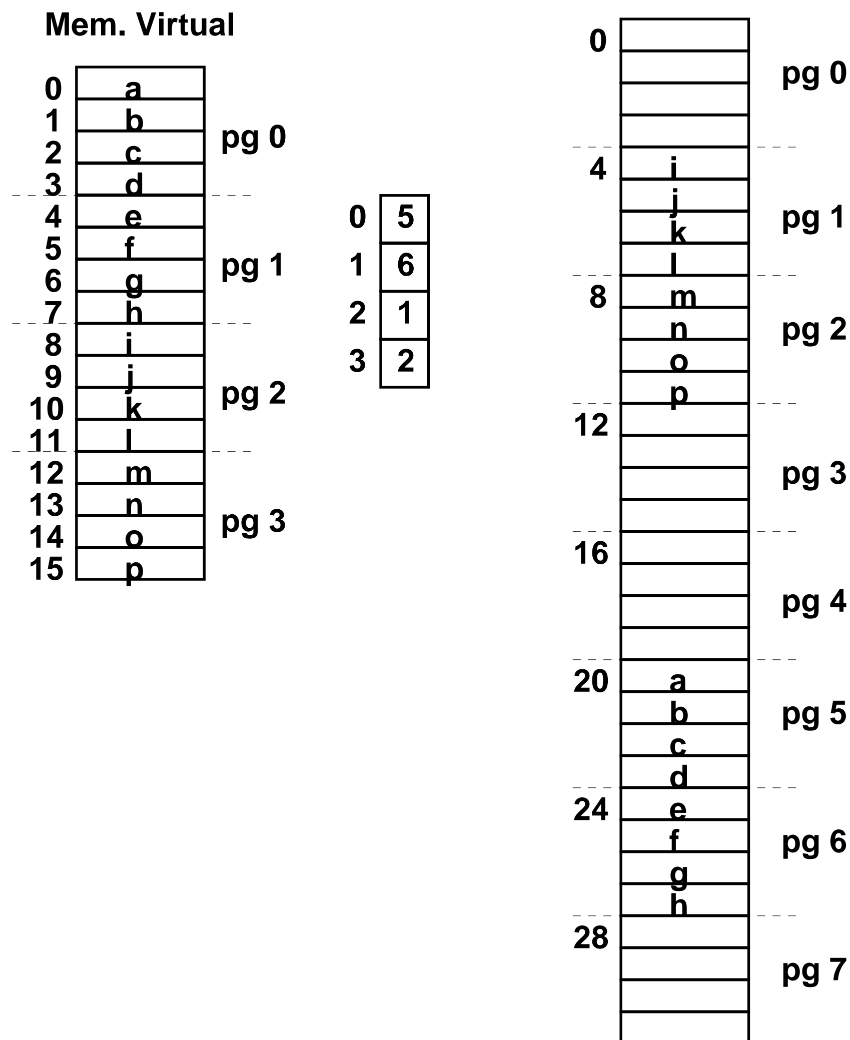


Ex: páginas de 4 bytes : então, são necessários 2 bits para o deslocamento.



Exemplo genérico:

Usando um comprimento de páginas de 4 *words* e uma memória física de 32 words (8 páginas). Obs: offset = deslocamento.



Para localizar o endereço físico onde se encontra a instrução "a", faz 5x4+0=20; "f" : 6X4+1=25 e "d" : 5x4+3=23.

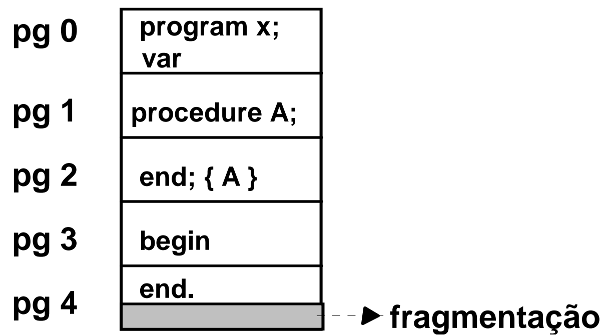
Além da informação sobre a localização da página virtual, uma entrada na tabela de páginas possui um bit que indica se uma página está ou não na memória principal (bit de validade). Se o bit for 0, indica que a página virtual não está na memória (page fault) e se for 1, indica que está.

Quanto a busca da página "faltante" no disco, existem 2 técnicas de paginação:

- paginação por demanda: o sistema só busca a página quando ela for referenciada (é a técnica mais utilizada).

- paginação antecipada: o sistema tenta prevê a utilização da página e faz uma busca antecipada (antes dela ser referenciada).

Fragmentação: Mesmo sendo em menor escala que as técnicas anteriores vistas, ainda existe o problema da fragmentação, que ocorre sempre na última página, pelo fato do programa não ocupar todas as páginas por completo.



Working Set: É o conjunto de páginas que um processo referencia constantemente, e por isso, deve permanecer na memória, na medida do possível. Sempre que um processo referencia uma nova página, ela entra no working set do processo.

O *working set* deve ter um limite máximo de páginas, assim, quando o sistema estiver com o seu *working set* no limite e precisar acessar outra página (*page in*), alguma das páginas do *working set* deve ser liberada (descartada) para permitir o carregamento da página nova.

Antes da página ser descartada, deve ser verificado se ela não foi alterada. Se foi, ela deve ser atualizada (gravada) no disco (*page out*). Para isso, basta colocar um campo "bit de modificação" na tabela de páginas, que indica se a página foi ou não modificada. Sempre que uma página for modificada, o SO seta este bit com o valor 1.

Existem várias técnicas para selecionar a página que será descartada:

1) Aleatória

2) FIFO: escolhe-se a página mais antiga do *working set*

3) LRU (*Least-recently used*): menos recentemente usada, ou seja, aquela que está a mais tempo sem ser referenciada

4) NUR (*Not used recently*): não usada recentemente, ou seja, que não seja a última que foi referenciada

5) LFU (*Least frequently used*): menos frequentemente usada, ou seja, menas vezes referenciada.

Pergunta: Na paginação, analisando a tabela de páginas, o que acontece se:

a) Se o *working set* de um processo for maior que a tabela de páginas?

- Haverá sobra de páginas de memória, de modo que o processo não terá necessidade de realizar paginação, pois todo o programa caberá na memória.

b) Se o working set de um processo for menor que a tabela de páginas?

- Neste caso, haverá necessariamente paginação, podendo até ser excessiva, conforme o tamanho do working-set.

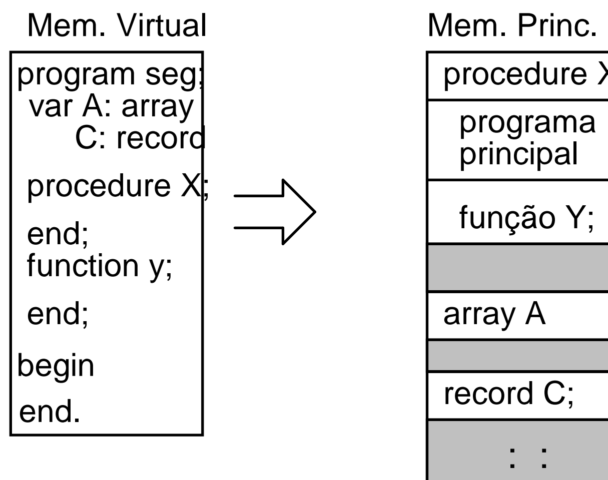
Pergunta: No mecanismo de Paginação, a memória física pode ficar fragmentada em diversas posições? Explique?

- Sim, pois os programas dificilmente totalizam páginas completas, ficando sempre a última página com fragmentação. Assim, sempre que as últimas páginas dos códigos dos programas estiverem na memória, haverá fragmentação.

5.4.2 Segmentação.

Aproveita a modularidade do programa, ou seja, a memória não é dividida em tamanhos fixos e sim conforme a estruturação do programa. Assim, os programas são divididos logicamente em sub-rotinas e estruturas de dados, e colocados em blocos de informações na memória.

Os blocos têm tamanhos diferentes e são chamados de segmentos, cada um com seu próprio espaço de endereçamento.



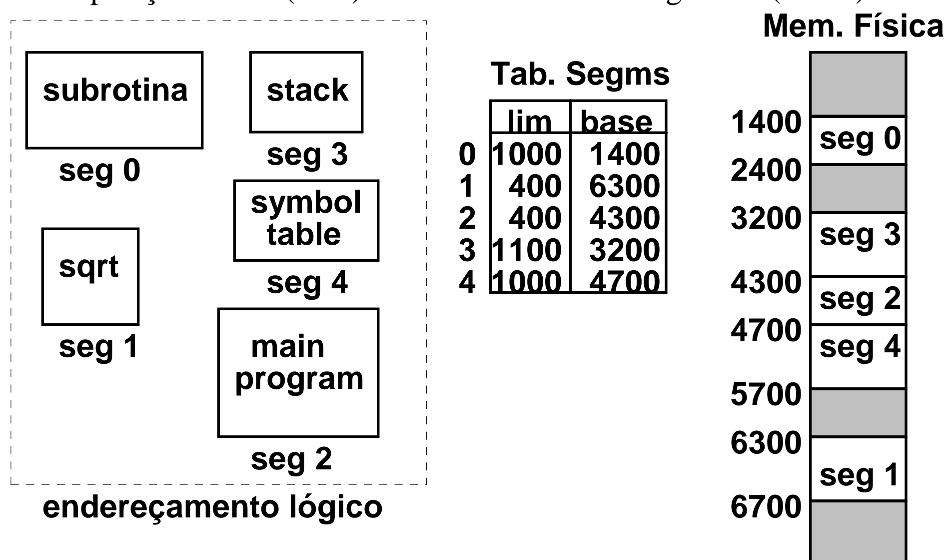
O mecanismo de mapeamento é semelhante ao da paginação. Os segmentos são mapeados por uma tabela de mapeamento de segmentos TMS, e os endereços são compostos pelo número do segmento e um deslocamento dentro do segmento.

Além do endereço do segmento na memória física, cada entrada na tabela de segmentos possui informações sobre o tamanho do segmento, se ele está ou não na memória etc. O sistema mantém uma lista de áreas livres e ocupadas da memória.

Na segmentação, apenas os segmentos referenciados são transferidos da memória secundária para a memória principal, e as técnicas para a escolha da área livre podem ser as mesmas da Alocação Particionada Dinâmica (*best-fit*, *worst-fit* ou *first-fit*).

Exemplo genérico:

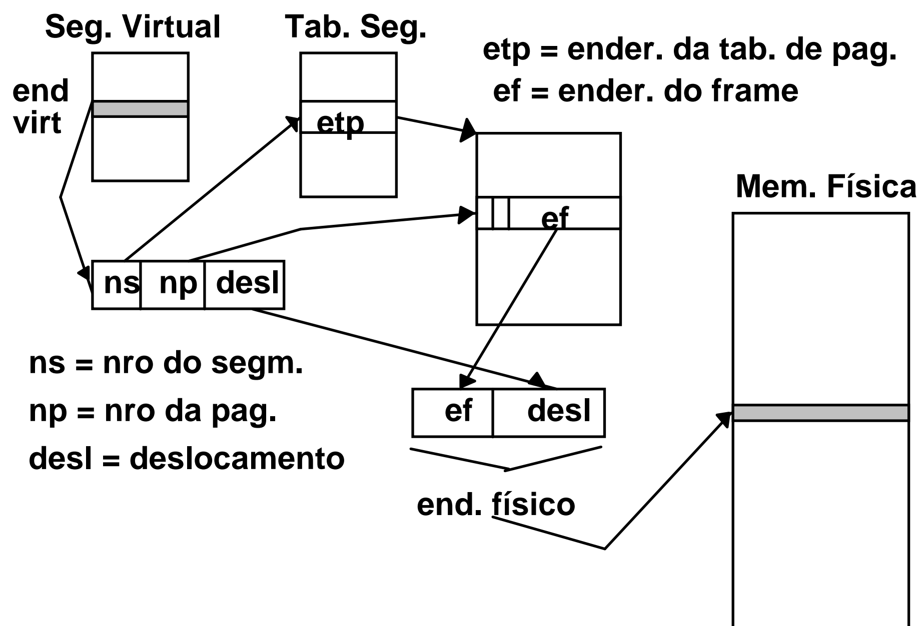
O programa possui 5 segmentos (0-4). A tabela de segmentos possui uma entrada separada para cada segmento, com a posição inicial (base) e o tamanho de cada segmento (limite).



Para localizar o endereço físico onde se encontra a instrução nro 53 do seg 2: 4300+53=4353; a instrução número 852 do segmento 3: 3200+852=4052.

5.4.3 Segmentação com Paginação:

Permite a divisão lógica dos programas em segmentos e, por sua vez, cada segmento é dividido fisicamente em páginas. Nesse sistema, um endereço é formado pelo número do segmento, um número da página dentro deste segmento e um deslocamento dentro dessa página. O nro do segmento aponta para uma entrada na tabela de segmentos, que por sua vez aponta para uma tabela de páginas.



Thrashing: Excessiva transferência de páginas e/ou segmentos entre a memória principal e a secundária. Ocorre basicamente em 2 níveis: do processo e do sistema.

do processo:

- elevado número de "page faults"
- o programa fica mais tempo esperando por páginas do que executando
- causa: - *working set* muito pequeno -> "aumentar"
  - mal programação: não respeito da "localidade" -> "reescrever"

do sistema:

- mais processos concorrendo pela memória do que espaço disponível (atender todos os processos) -> "expandir a memória"

Pergunta: Qual a relação do princípio da localidade e overhead no barramento?

- Se o programa não respeita o princípio da localidade, o sistema poderá acusar muito *page-fault* e efetuar muita paginação e consequentemente haverá muito overhead.

5.5 Exercícios

1. Explique o que é alocação contígua simples.
2. Explique o que é alocação particionada (estática e dinâmica).
3. Explique o que é *swapping*.
4. Explique o que o esquema de paginação.
5. Explique o que o esquema de segmentação.
6. Explique as seguintes técnicas para alocação de partições: best-fit, worst-fit e first-fit.
7. Explique as seguintes técnicas para a escolha da página a ser descartada: Aleatória, FIFO, LRU, NUR e a LFU
8. Explique o problema de fragmentação
9. Explique Memória Virtual
10. Explique Paginação sob-demanda
11. Explique a substituição de página quando ocorre Page-fault
12. Explique três algoritmos de substituição de página
13. Explique Thrashing

6 SISTEMAS DE ARQUIVOS

Discos

Os discos possuem um formato circular. Suas duas superfícies são cobertas com um material magnético. Durante a formatação, cada superfície é dividida em trilhas e cada trilha é dividida em setores (também chamado de bloco do disco), onde são armazenadas as informações. As informações são lidas ou escritas através de uma cabeça de leitura/gravação.

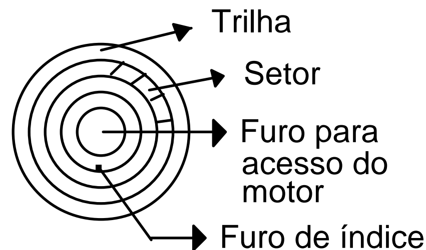


Figura 29 - Disco

Um setor é a menor unidade de informação que pode ser lida ou escrita no disco. Todas as operações de entrada e saída no disco, são feitas em uma certa quantidade de setores (isto pode gerar fragmentação, se o setor não for totalmente preenchido).

O *hardware* para um sistema de disco é basicamente gerenciado pelo **controlador de disco,** que determina a interação lógica com o computador. O controlador traduz instruções enviadas pela CPU, em um conjunto de sinais que serão utilizados para o controle do disco. O controlador de disco é ativado pela CPU através de seus registradores de entrada e saída.

O sistema operacional implementa várias camadas de *software*, para que seja facilitada a gerência dos discos. As camadas de mais baixo nível, tentam esconder as características do *hardware* dos dispositivos (são camadas vinculadas ao dispositivo) das camadas mais superiores, dando maior portabilidade para estas, permitindo ao sistema operacional trabalhar de forma independente do *hardware*.

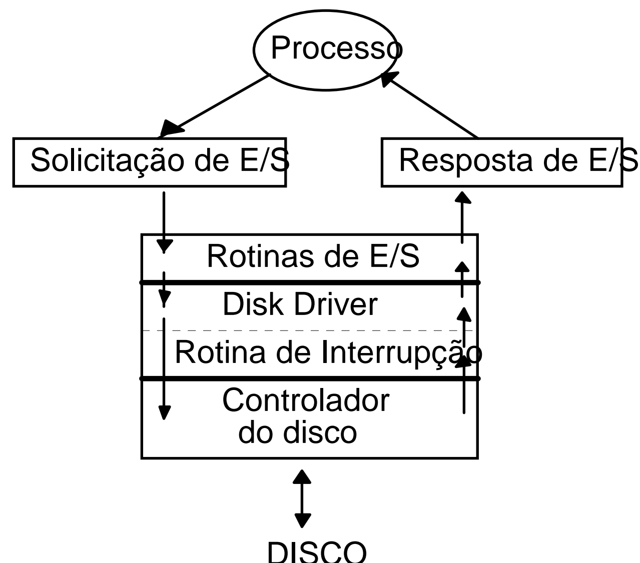


Figura 30 - Portabilidade

O usuário acessa o disco através das rotinas de E/S fornecidas pelo SO. Estas rotinas permitem ao usuário realizar operações sobre os discos, sem se preocupar com detalhes tais como: como o disco está formatado, qual a trilha ou setor que o arquivo será gravado, etc.. Estas rotinas também são as responsáveis pelo controle de permissões do usuário.

Os *disk drivers*, ou somente *drivers*, são todas as rotinas do SO (incluindo as rotinas de interrupção) que devem se comunicar com o disco, em nível de hardware, através dos controladores de discos. Os *disk drivers* sabem quantos registradores um controlador possui, para que eles servem, como são utilizados e quais são seus comandos. O controlador do disco geralmente possui memória e registradores próprios, para poder executar as instruções enviadas pelo *disk driver*.

Arquivos

Os computadores podem armazenar informações fisicamente em diferentes formas: discos e fitas magnéticas são as mais comuns. Cada um destes dispositivos possui organização e características próprias. Mas o sistema operacional deve se abstrair das propriedades físicas e fornecer uma visão lógica para o usuário, definindo uma unidade lógica de armazenamento, o arquivo ou ***file.*** Desta forma, os *files* são mapeados pelo sistema operacional sobre os dispositivos físicos reais.

Um arquivo é um conjunto de informações relacionadas, definidas pelo seu criador. Geralmente, arquivos representam programas (fontes ou objetos) ou dados, que podem ser numéricos, alfabéticos ou alfanuméricos. Podem ser compostos por uma sequência de bits, bytes, linhas ou registros, mas seu significado é definido pelo seu criador e usuário. Arquivos possuem nomes e algumas propriedades, tais como: tipo, data de criação, nome do proprietário, tamanho etc.., que são armazenadas em seu Descritor de Arquivos.

As rotinas de entrada e saída implementam operações sobre os arquivos, entre elas temos: criação, deleção, leitura, gravação, obtenção do tamanho e truncamento.

6.1.1.1 Diretórios

Um disco normalmente tem um diretório indicando quais arquivos estão no disco. Este diretório é uma lista de todos os arquivos por nome. O diretório também contém os endereços de todos os descritores de todos os arquivos, como mostra o exemplo da figura abaixo, que possibilitam qualquer operação sobre eles. Estes diretórios são colocados em um ou mais arquivos especiais, ou em áreas de armazenamento.

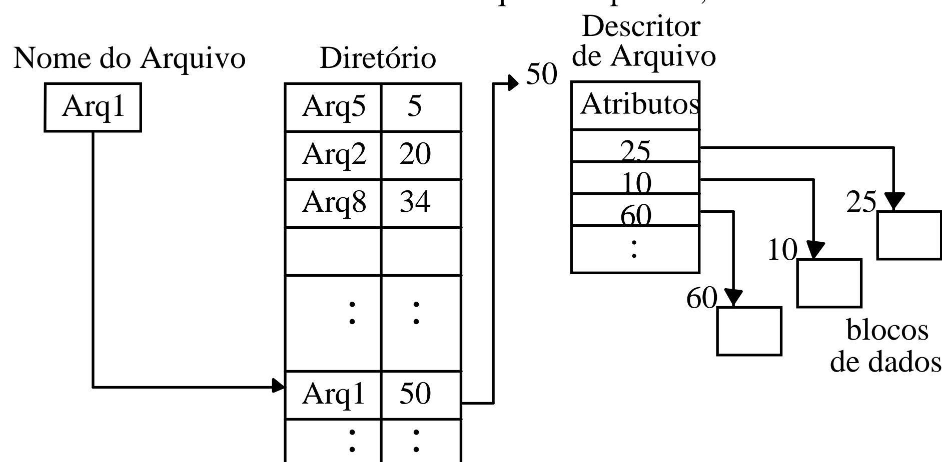


Figura 31 - Diretórios

Para facilitar a manipulação dos arquivos no disco pelo usuário, o Sistema de Arquivos fornece uma estruturação o diretório da forma mais conveniente. Esta estruturação deverá fornece mecanismos para a organização de muitos arquivos, que podem estar armazenados fisicamente em diferentes discos. O usuário só se preocupa com o diretório lógico e ignora os problemas da alocação física.

Existem muitas estruturas propostas para diretórios, mas os tipos principais são: Diretório de Nível Simples, Diretório de Dois Níveis, Diretório em Árvores e Diretório em Grafos Acíclicos.

6.1.1.1.1 Diretório de Nível Simples

É a estrutura de diretório mais simples que contém um único nível. Todos os arquivos estão contidos no mesmo diretório, como mostra a figura abaixo.

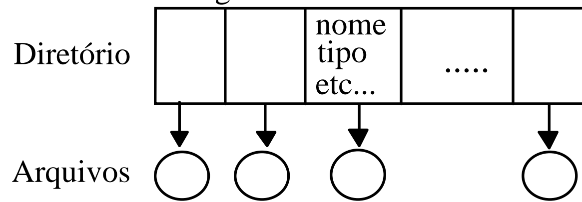


Figura 32 - Nível simples

Este tipo de estrutura possui limitações significantes, quando o número de arquivos cresce ou quando existe mais do que um usuário, como por exemplo: se todos os arquivos estão em um mesmo diretório, eles devem possuir nomes diferentes. Como controlar isto se houver mais que um usuário? Mesmo com um único usuário, se a quantidade de arquivos for grande, como controlar a utilização sempre de nomes diferentes para os arquivos?

6.1.1.1.2 Diretório de Dois Níveis

Neste tipo de estrutura, cada usuário possui seu próprio diretório de nível simples, permitindo que usuários diferentes possam ter arquivos com os mesmos nomes. Assim, o sistema possui um diretório mestre que indexa todos os diretórios de todos os usuários, como mostra a figura abaixo.

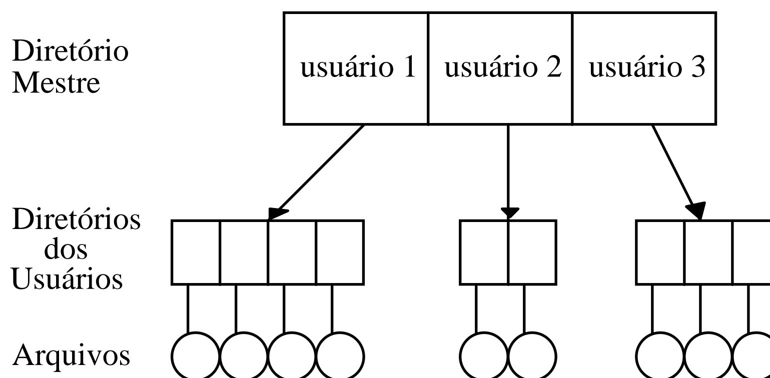


Figura 33 - Dois níveis

Este tipo de estrutura é vantajosa quando os usuários são completamente independentes, mas é desvantajosa, quando os usuários querem cooperar sobre algumas tarefas e acessar os arquivos de outros. Alguns sistemas simplesmente não permitem que os usuários acessem outros diretórios. Mas se o acesso for permitido, o usuário deve ser hábil em nomear um arquivo em um diretório de um outro usuário.

6.1.1.1.3 Diretórios em Árvores

Um diretório de dois níveis pode ser visto como uma árvore de dois níveis. A generalização natural é extender a estrutura de diretório para uma árvore árbitrária, como mostra a figura abaixo. Isto permite que os usuários criem seus próprios sub-diretórios. O sistema UNIX por exemplo, utiliza este tipo de estruturação.

A árvore possui um diretório raíz. Cada arquivo neste sistema possui um único caminho ("pathname"). Este caminho começa da raíz, percorre todas as subárvores, até o arquivo especificado, como mostra a figura abaixo.

Cada diretório possui um conjunto de arquivos e/ou subdiretórios. Todos os diretórios possui um formato interno. Um *bit* em cada entrada do diretório define a entrada como um arquivo (0) ou subdiretório (1).

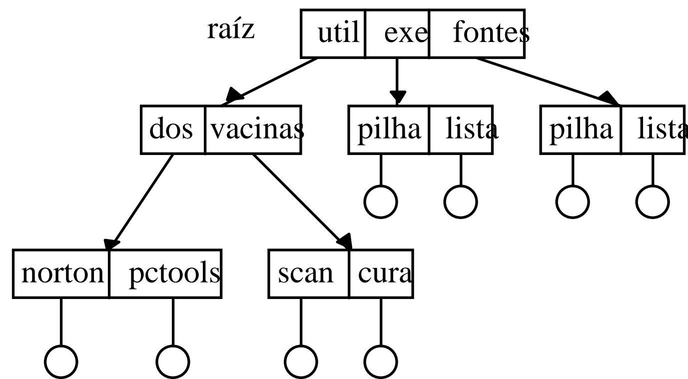


Figura 34 - Diretório em árvore

O sistema mantém um arquivo de usuários cadastrados, e para cada entrada neste arquivo existe um nome do usuário e um ponteiro para o seu diretório inicial. Cada usuário possui um diretório corrente. Quando é feita uma referência a um arquivo, o diretório corrente é pesquisado. Se o arquivo não está no diretório corrente, então o usuário deve especificar o "caminho" (*pathname*) ou trocar o diretório corrente.

6.1.1.1.4 Diretórios em Grafos Acíclicos

Este tipo de estruturação permite o compartilhamento de arquivos e/ou diretórios por vários usuários, evitando a manipulação de várias cópias do mesmo arquivo. Quando o arquivo compartilhado por um usuário que possui o direito de acesso, todas as alterações são efetuadas neste arquivo e imediatamente se tornam visíveis aos outros usuários.

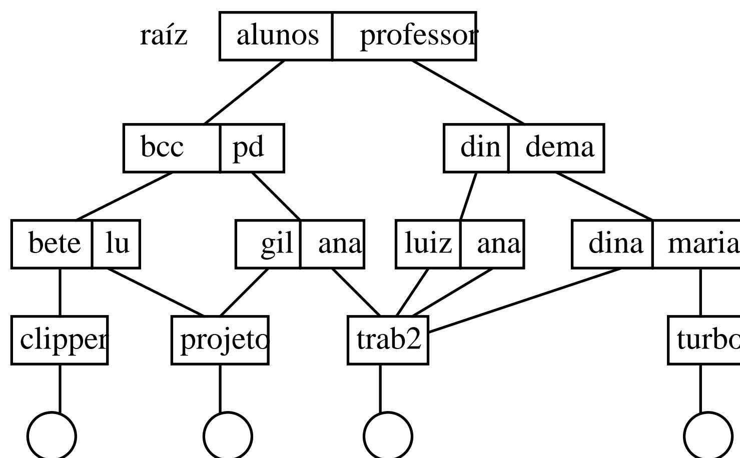


Figura 35 - Grafos Acíclicos

Uma maneira usual para implementar este tipo de estrutura, é utilizando um *link*, que funciona da seguinte forma: quando se deseja ter acesso a um arquivo (ou diretório) através de um diretório, adiciona-se uma entrada neste diretório com o nome do arquivo (ou diretório) desejado e marca-o como sendo do tipo *link.* Em compensação, adiciona-se nesta entrada o *pathname* definitivo do arquivo.

Este tipo de estruturação apresenta uma série de incovenientes, entre eles podemos destacar a "deleção". Neste caso teremos que eliminar todos os *links* para este arquivo.

6.1.1.2 Gerenciamento de Espaço Livre em Disco

Na criação do arquivo, é necessário que o sistema operacional tenha condições de verificar se cada bloco do disco está livre ou ocupado. Torna-se necessário, portanto, que sejam mantidas estruturas de

dados que armazenem as informações necessárias, de modo a possibilitar ao sistema operacional, o gerenciamento do espaço livre presente no disco.

Existem várias formas para o armazenamento das informações necessárias, entre elas: Mapa de Bit, Lista de Blocos Livres, Bloco de Endereços e Tabela de Blocos Contíguos.

6.1.1.2.1 Mapa de Bits

Cada bloco é representado por 1 bit; se o bloco está livre, o bit é "0", senão é "1". Por ex: considere um disco onde os blocos 2,3,4,5,8,9,10 estão livres, e o resto dos blocos estão ocupados.

11000011000....

- fácil procura de n blocos consecutivos
- gasto de disco para armazenar o mapa de bit
- Em uma única leitura no disco (leitura do mapa de bits), é possível saber a situação de um grande número de setores.

6.1.1.2.2 Lista de Blocos Livres

Manter todos os blocos livres ligados por uma lista, e guardar a cabeça (endereço inicial da lista) da lista. Aproveita os próprios blocos livres para armazenar os endereços da lista. Não é possível ter uma visão geral sobre a situação de um conjunto de setores; deve-se ler uma a um, para saber se o setor está disponível ou não.

Figura 36 - Lista de blocos livres

6.1.1.2.3 Bloco de Endereços

Utiliza-se o primeiro bloco livre, como bloco de endereços, para armazenar o endereço dos próximos "n" blocos livres, sendo que o bloco número "n", é um outro bloco de endereços. Guarda-se o endereço do primeiro bloco de endereços. Este método também permite, em uma única leitura (leitura do bloco de endereços), saber sobre a situação de vários blocos, incluindo os endereços.

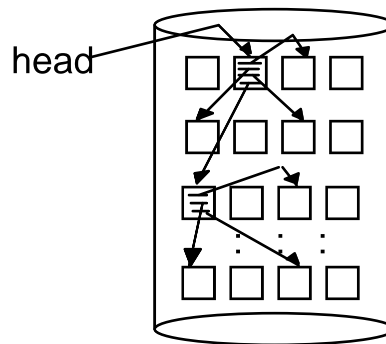
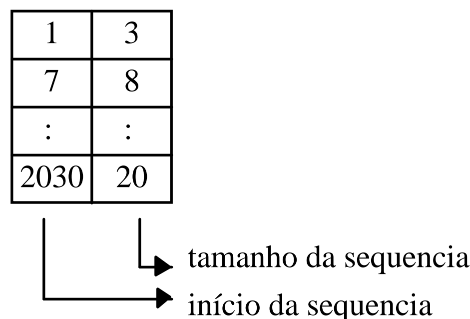


Figura 37 - Bloco de endereço

6.1.1.2.4 Tabela de Blocos Contíguos

Aproveitar a idéia de que sempre existe vários blocos livres contíguos. Mantém uma tabela de blocos livres, onde cada entrada na tabela contém o endereço do 1° bloco de um grupo de blocos contíguos e o tamanho do grupo. Guarda-se o cabeça da lista.



1	3
7	8
:	:
2030	20

Figura 38 - Blocos contíguos

Pergunta: Na figura acima, como poderia ser atualizada a tabela, se o SO alocasse 2 setores da segunda sequência?

- O primeiro campo (início) seria incrementado de 2 e o segundo campo (quantidade) seria decrementado de 2.

6.1.1.3 Alocação de Espaço em Disco

Uma das preocupações do sistema de arquivos, é como alocar os espaços livres do disco para os arquivos, de forma que o disco seja melhor utilizado e os arquivos possam ser acessados mais rapidamente. Os principais métodos de alocação de espaço do disco são: alocação contígua, alocação ligada e alocação indexada.

6.1.1.3.1 Alocação Contígua

Este método procura identificar uma sequência contínua de blocos que apresente um tamanho suficiente para armazenar todas as informações relativas ao arquivo, ou seja, caso um arquivo precise de N blocos para o seu armazenamento, o sistema operacional procurará identificar uma sequência contínua de N blocos livres do disco.

Nesta técnica, o espaço só será alocado se for contíguo, caso contrário, o arquivo não poderá ser armazenado ou extendido. Se por acaso existir mais de uma possibilidade, então poderá ser utilizado uma técnica para selecionar uma opção. Pode se utilizado as técnicas Worst-Fit, Best-Fit e First-Fit.

Note que com esse método, para acessar um bloco *b+1* após ter acessado o bloco *b*, geralmente não haverá necessidade de nenhum movimento da cabeça de leitura/gravação do disco, a menos que o bloco se encontre em outra trilha. Além disso, o arquivo que foi alocado com esta técnica, pode ser acessado tanto sequencialmente quanto diretamente.

A alocação contígua de um arquivo é definida pelo endereço do primeiro bloco e seu comprimento, e dessa forma é colocado no diretório, como mostra a figura abaixo.

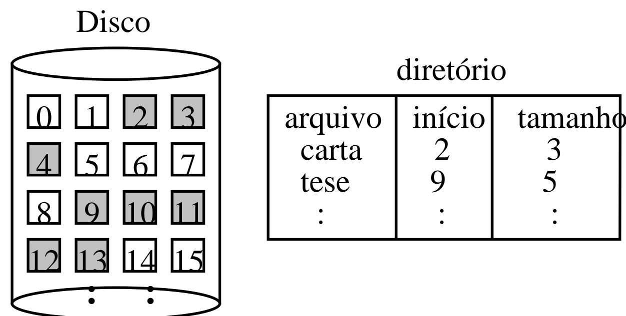


Figura 39 - Alocação contígua

Este método encontra duas principais problemas:

1. Encontra o espaço contíguo suficiente para o arquivo. Isto pode causar fragmentação externa, isto é, existe espaço no disco mas não é contíguo.
2. Determinar quanto de espaço uma arquivo que está sendo criado irá precisar. Deve-se portanto estimar uma quantidade de blocos. Se for pouco, o arquivo não poderá crescer, e se for muito, o disco poderá estar sendo sub-utilizado caso o arquivo for pequeno.

6.1.1.3.2 Alocação Ligada

Nesta técnica, os blocos alocados para um arquivo são encadeados em uma lista, não importanto a contiguidade. Desta forma, seria armazenado no diretório o nome do arquivo e duas variáveis, que seriam os identificadores do primeiro e do último bloco do arquivo. Em cada um dos blocos utilizados pelo arquivo, seria reservado um espaço para conter o endereço do próximo bloco da lista.

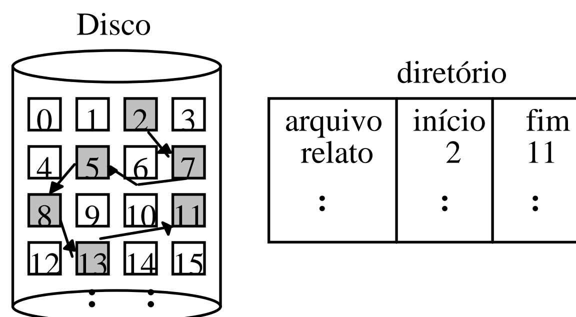


Figura 40 - Alocação ligada

Apesar de qualquer bloco poder ser utilizado na alocaçãò, este método apresenta 2 principais desvantagens: o gasto de disco decorrente da necessidade de armazenamento dos ponteiros usados para implementar a lista, e a necessidade de acesso sequencial as informações armazenadas, que podem causar vários movimentos da cabeça de leitura/gravação.

Este método resolve o problema da fragmentação externa e o problema do crescimento do arquivo, não existindo a necessidade de alocação prévia.

6.1.1.3.3 Alocação Indexada

Neste método, cada arquivo possui seu próprio bloco de índice, que é um vetor de endereços. A i-ésima entrada no bloco de índice aponta para o i-ésimo bloco do arquivo, como mostra afigura abaixo:

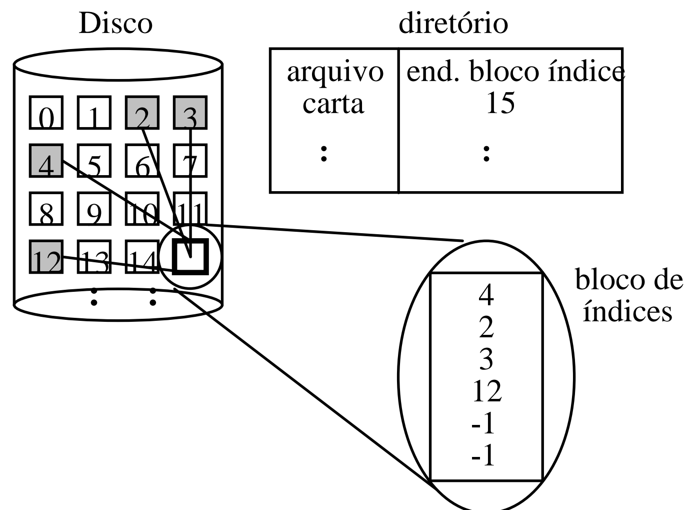


Figura 41 - Alocação indexada

Quando o arquivo é criado, todos os ponteiros do bloco de índice são inicializados com *nil*. Quando o i-ésimo bloco é escrito, um bloco é removido do espaço livre e seu endereço é colocado na i-ésima entrada do bloco de índices do arquivo.

mesmo se o arquivo é pequeno (1 ou 2 blocos), esta técnica irá consumir um bloco de índices, o que na alocação ligada seria utilizado apenas 2 ponteiros. O objetivo é ter um bloco de índices tão pequeno quanto possível, sempre levando em consideração que o tamanho do bloco de índices limita o tamanho do arquivo, assim, algumas técnicas podem ser utilizadas:

1. ligar vários blocos de índices, de forma que a última posição do bloco de índices contenha o endereço de um outro bloco de índices ou contenha o valor *nil*.

2. utilizar uma indexação de 2 níveis, usando um bloco de índices para conter os endereços de outros blocos de índices, que por sua vez conteriam os endereços dos blocos de dados do arquivo. Isto pode ser estendido para mais de 2 níveis.

3. usar um método misto (utilizado no sistema UNIX), como por exemplo: o diretório do disco conteria para cada arquivo 15 ponteiros, que seriam utilizados da seguinte maneira:

- 12 apontariam diretamente para os blocos de dados do arquivo e seriam suficientes para arquivos pequenos. Se o bloco = 4096, seriam suficientes para arquivos até 48K.
- 3 ponteiros seriam usados para apontar blocos indiretos, onde o primeiro apontaria para um bloco indireto simples, o segundo apontaria para um bloco indireto duplo (índice de 2 níveis) e o terceiro apontaria para um bloco indireto triplo (índice de 3 níveis).

6.1.1.4 Escalonamento do Disco

Para que o sistema operacional atenda uma solicitação de acesso ao disco por uma aplicação, ele gasta um certo tempo. Este tempo é equivalente a somatória do tempos gastos em 3 etapas:

1. Tempo de Seek: tempo gasto para locomover a cabeça de leitura/gravação da trilha atual para a trilha desejada.
2. Tempo de Latência: tempo gasto para esperar que o disco rotacione até que o bloco desejado esteja sob a cabeça de leitura/gravação.
3. Tempo de Transferência: tempo gasto para transferir os dados do bloco do disco para a memória principal.

Em um sistema multiprogramado, várias solicitações de acesso ao dico são feitas pelas diferentes aplicações, de forma que o sistema operacional deve ser hábil em escalonar qual solicitação atender, visando obter o menor tempo médio nos acessos ao disco, para obetr o melhor desempenho do sistema.

As principais políticas de escalonamento do disco são: *First-Come-First-Served*, *Shortest-Seek-Time-Firs*t e *SCAN*.

1. FCFS: As solicitações são atendidas por ordem de chegada (FIFO)
2. SSTF: A solicitação cuja trilha estiver mais próxima da trilha atual da cabeça de leitura/gravação do disco é atendida primeiro.
3. SCAN: O sistema atende as solicitações que estiverem em um determinado sentido (da trilha 0 em direção a última trilha ou vice-versa), e quando não houver mais solicitação naquele sentido, o sistema inverte o sentido e repete o processo. Desta forma, a cabeça de leitura/gravação executa uma varredura de um lado para o outro do disco.

Para exemplificarmos as três técnicas, vamos supor que existe uma fila de requisição de acesso ao disco, onde as trilhas que devam ser acessadas estão na seguinte sequência 98, 183, 37, 122, 14, 124, 65 e 67, de forma que a primeira solicitação que foi feita é a trilha 98 e a ultima é a 67. Supomos que a cabeça de leitura/gravação do disco esteja na posição 53. O gráfico abaixo mostra o caminho percorrido utilizando-se as 3 técnicas.

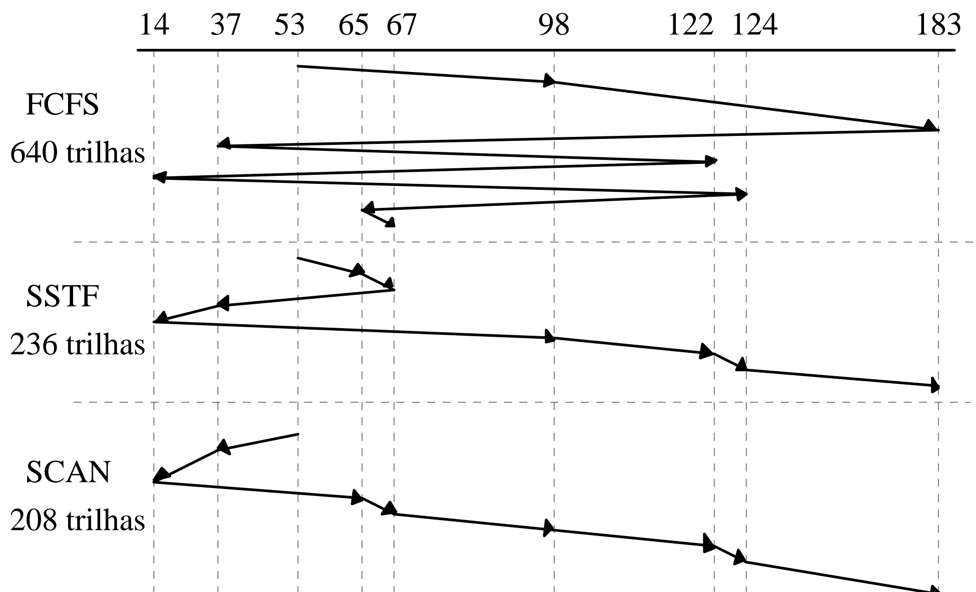


Figura 42 - varredura

6.2 Exercícios

1) Qual é o motivo do SO ter um Sistema de Arquivos ?
2) Explique os métodos de acesso a arquivos
3) Explique as formas de estruturar diretórios
4) Explique a proteção que o Sistemas de Arquivos devem oferecer
5) Qual é o motivo do SO ter um Sistema de Arquivos ?
6) Explique os métodos de acesso a arquivos
7) Explique as formas de estruturar diretórios
8) Explique a proteção que o Sistemas de Arquivos devem oferecer
9) Explique os algoritmos FCFS, SSTF, SCAN, C-SCAN
10) Qual a importância do backup ? Como pode ser feito ? (diário, mensal)

REFERÊNCIAS BIBLIOGRÁFICAS

MACHADO,F.B. & MAIA,L.P. **Introdução à arquitetura de Sistemas Operacionais**. Rio de Janeiro.Editora LTC- Livros Técnicos e Científicos.1992.

SILBERSCHATZ, A. & GALVIN, P. B. & GAGNE, G. **Fundamentos de Sistemas Operacionais**, 6ª Edição, 2004, LTC.

TANENBAUM,A. S. **Sistemas Operacionais Modernos**. 2ºEdição. São Paulo. Prentice Hall.2003.

TOSCANI, S., OLIVEIRA, R., CARISSIMI, A. **Sistemas Operacionais e Programação Concorrente**. Série Didática do II UFRGS. Ed. Sagra-Luzzato, 2003.