SISTEMA OPERACIONAL I

Ismael H F Santos

Versão 2.0 – Novembro 2005

1 Bibliografia

1.1 Bibliografia Básica

- Sistemas Operacionais Modernos - Andrew S. Tanembaun, Prentice Hall 1992.
- Introdução à Arquitetura de Sistemas Operacionais- Francis B Machado e Luis Paulo Maia, LTC 1997 2ª Edição.
- Operating System Concepts: Internals and Design Principles – William c, Prentice Hall 1998 3ª Edição
- Operating System Concepts – James L. Peterson & Abraham Silberschatz, Addison-Wesley 1985, 2ª Edição
- Operating Systems - Harrey, M. Deitel, Addison-Wesley 1990, 2ª Edição

1.2 Bibliografia Adicional

- Computer Organization and Architecture: Principles Structure and Function – Williiam Stallings, Addison-Wesley 1990, 2ª Edição
- Organização Estruturada de Computadores - Andrew S. Tanembaun, Prentice Hall 1988.

2 Introdução

2.1 Conceitos Básicos

2.1.1 Conceito de Sistema Operacional

Software
- **System Software** (SW básico) - controlam a operação do computador.
- **Application Software** (SW aplicativo) - resolve os problemas do usuário.

O mais importante componente do software básico é o ***Sistema Operacional*** (SO), o qual pode ser entendido como um conjunto de rotinas executadas pelo processador com o objetivo de controlar o funcionamento e os recursos da máquina. O SO fornece o suporte necessário para que os programas aplicativos possam ser escritos mais facilmente.

Nos primeiros computadores a computação era realizada através de painéis, por meio de fios e conexões eletrônicas, o que exigia, por parte do programador, um grande conhecimento do HW e da sua linguagem de máquina. Isso era uma grande dificuldade para os programadores da época.

A solução para tal problema foi isolar o programador da complexidade do HW através da construção de uma "camada" de SW que manipulasse diretamente o HW e se apresentasse para o usuário como uma interface fácil de entender e programar. Dessa forma cria-se o conceito abstrato de ***máquina virtual***. Tal camada de SW é chamada comumente de ***Sistema Operacional.***

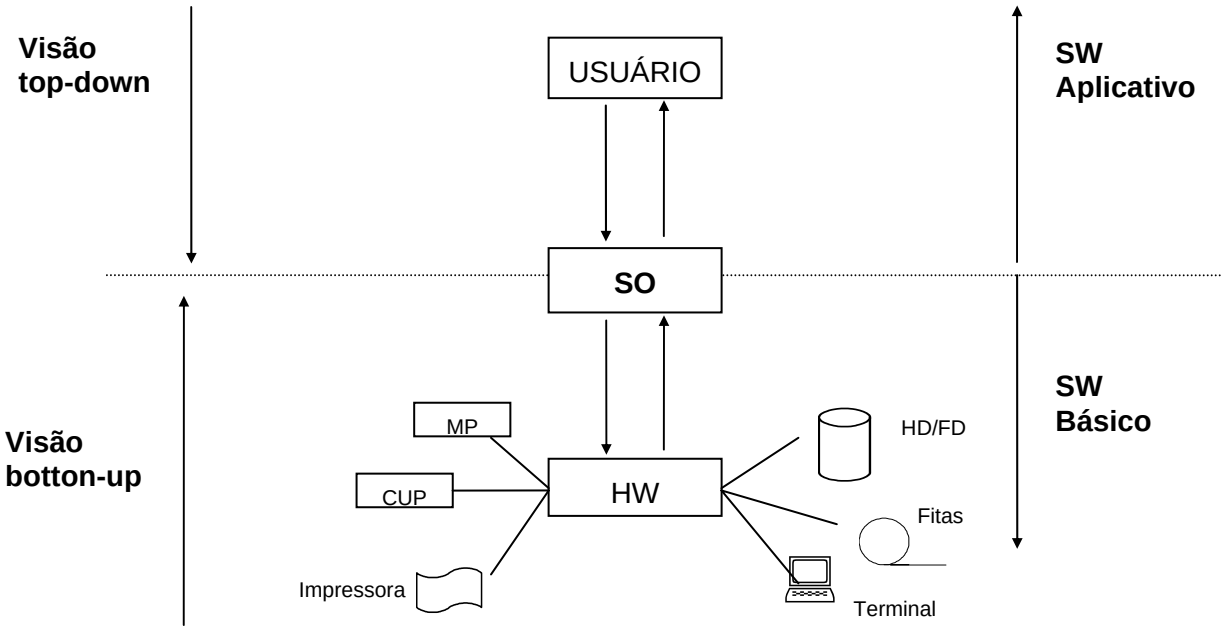


Fig 2.1 - Visão do Sistema Operacional

A partir da Figura 2.1 podemos observar duas visões distintas do SO, a saber:

- **Visão Top-down**: O SO deve *facilitar e padronizar o acesso aos recursos do sistema*, servindo de interface entre o usuário e os recursos do sistema computacional, tornando esta comunicação transparente, eficiente e menos suscetível a erros. Dessa forma, o SO permite ao usuário enxergar o computador como uma ***máquina virtual*** sobre a qual ele pode executar suas tarefas de forma conveniente e segura.

- **Visão Botton-up**: O SO deve *permitir o compartilhamento de recursos, entre os diversos usuários, de forma organizada e protegida*. Tal compartilhamento além de dar impressão ao usuário de ser o único a utilizar um determinado recurso permite a diminuição de custos, na medida que mais de um usuário passa a utilizar as mesmas facilidades concorrentemente e de forma segura.

É comum pensar-se que os compiladores, linkers, bibliotecas, depuradores e outras ferramentas fazem parte do SO, mas, na realidade, estas facilidades são apenas utilitários, destinados a ajudar na interação do usuário com o computador.

Não é apenas em sistemas multiusuário que o SO é importante. Também no caso dos computadores pessoais o SO nos permite executar várias tarefas, tais como: imprimir um documento, fazer um 'download' de um arquivo pela Internet ou editar um arquivo. O SO deve ser capaz de controlar a execução concorrente de todas essas tarefas.

2.1.2 Modelo do Computador em Camadas - Máquina de Níveis

Segundo Tanenbaum, a visão mais moderna para se entender o computador é encará-lo como uma máquina composta por várias camadas (ou níveis), sendo o SO um dessas camadas. Cada nível representa visões abstratas do computador com diferentes objetos e operações presentes. O conjunto formado pelos tipos de dados, operações e particularidades de um determinado nível é denominado a arquitetura deste nível. A arquitetura então se relaciona com os aspectos que são visíveis pelo usuário de cada nível, excetuando-se os aspectos de implementação física.

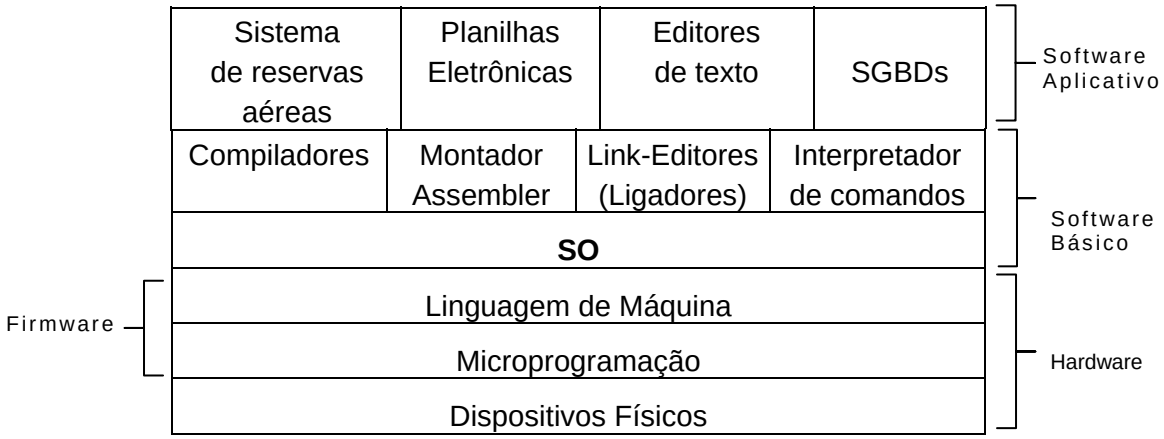


Fig 2.2 - Máquina de Níveis

O nível mais primitivo da máquina de Níveis proposta por Tanenbaum é o dos ***Dispositivos Físicos***, constituído pelos circuitos integrados, fonte de alimentação, memórias, periféricos e controladoras, etc. O modelo que adotaremos para um sistema de computação é aquele proposto por Von-Newman, apresentado a seguir.

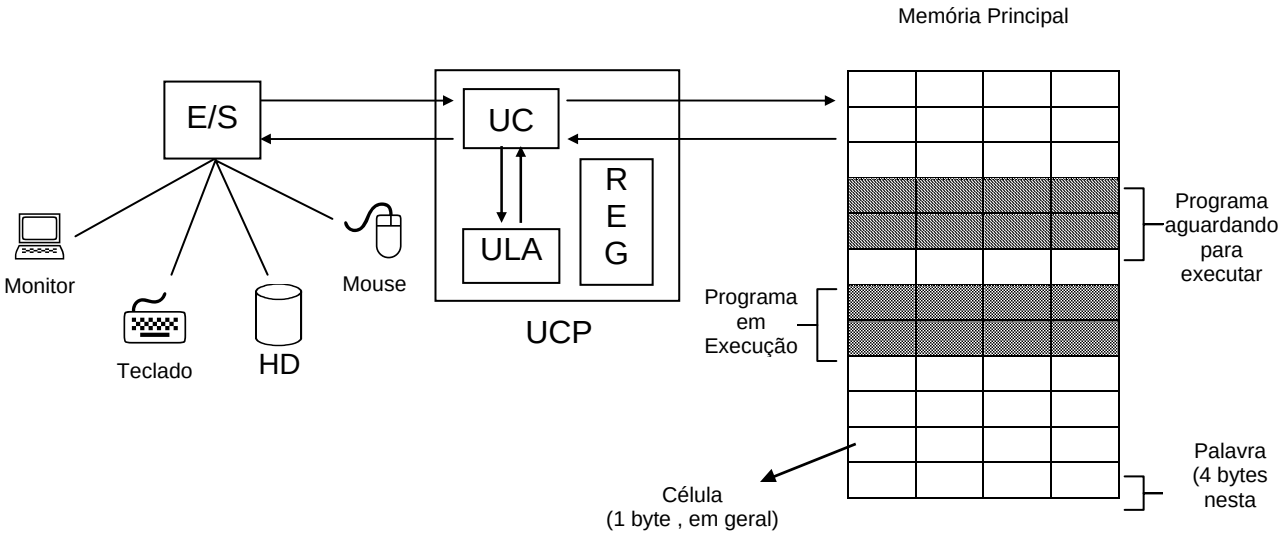


Fig 2.3 - Modelo de Máquina de Von-Newman

Acima do nível dos ***Dispositivos Físicos*** temos o nível de ***Microprogramação***, que representa uma camada de SW primitiva que controla diretamente os ***Dispositivos Físicos***, fornece uma interface clara para a camada superior e é usualmente gravada em memória ***ROM*** (Read Only Memory). Em algumas máquinas o microprograma é implementado em HW e em outras máquinas novos microprogramas podem ser escritos pelo usuário. Tal arquitetura é chamada de Máquina ***Microprogramável***.

O nível de ***Linguagem de Máquina*** é definido pelo conjunto de todas as instruções que serão executadas pelo nível de ***Microprogramação***. Cada instrução tem um formato definido, o qual é interpretado (e executado) pelo microprograma correspondente a instrução da ***Linguagem de Máquina***.

O ***Firmware*** é a designação que se dá a todo e qualquer SW que seja implementado em dispositivos semicondutores. Normalmente é utilizado quando o programa não deve ser alterado durante a vida útil do computador, ou quando não se deseja perdê-lo quando o computador é desligado. Geralmente o nível de ***Microprogramação*** é implementado com ***Firmware***. O programa de Boot (inicialização) do PC é um exemplo de ***Firmware***.

A seguir apresentamos o funcionamento da máquina de Von-Newman, através do ciclo de execução das instruções. Inicialmente o programa é carregado da memória secundária para a memória principal. A partir de então, quando o programa é iniciado, o PC é carregado com a primeira instrução executável e se inicia uma sucessão de ciclos de execução de instruções até o término do programa.

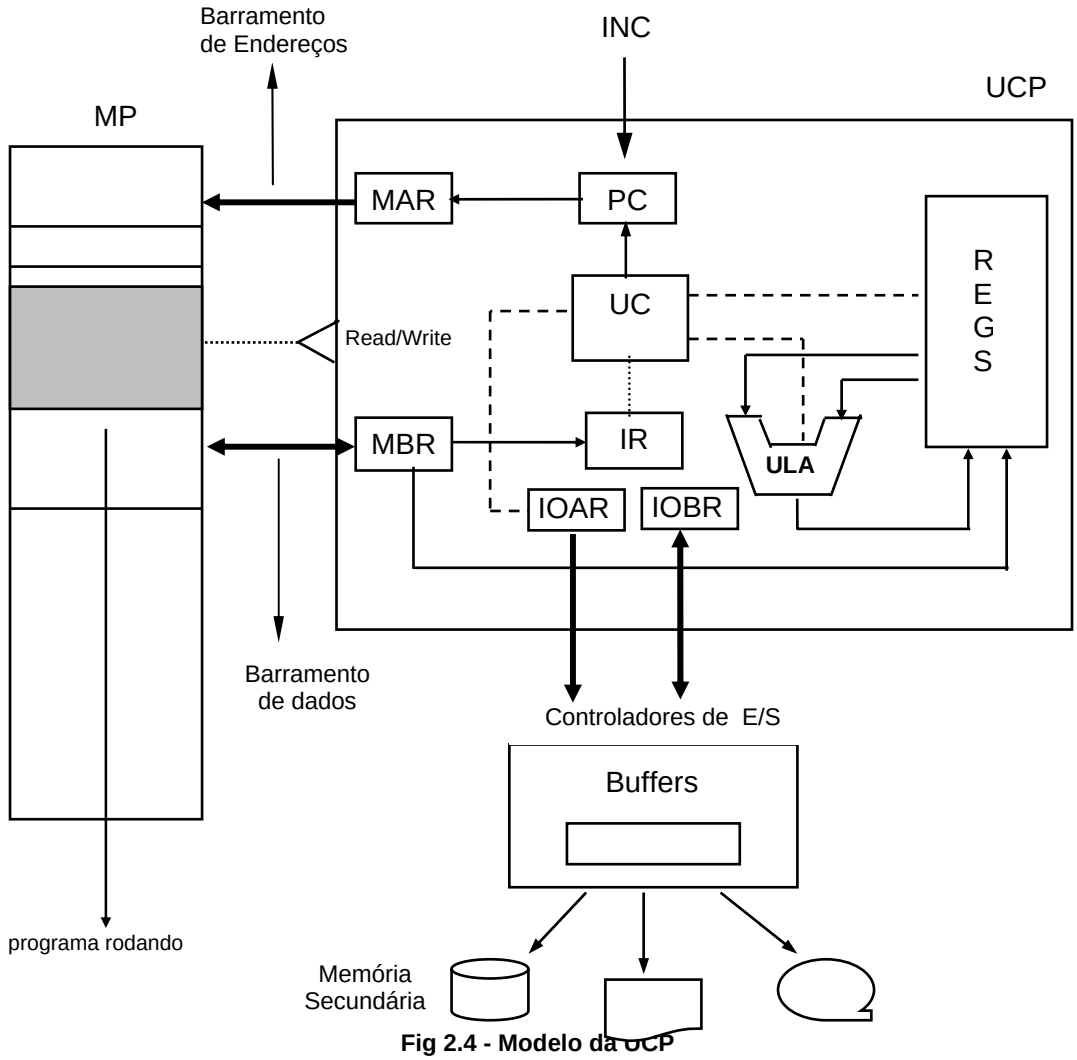


Fig 2.4 - Modelo da UCP

PC *(Program Counter)*
Indica o endereço da próxima instrução a ser executada.

MAR *(Memory Adress Register)*
Indica o endereço que será lido/gravado na MP.

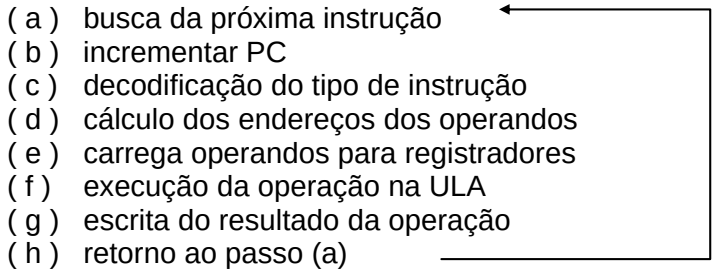
MBR *(Memory Buffer Register)*
Contém o conteúdo da posição de memória apontada por MAR.

IR *(Instruction Register)*
Contém a instrução vinda do MBR e que será executada pelo microprograma que executa a instrução na UC.

UC *(Unidade de Controle)*
Responsável pela busca das instruções da memória principal, identificação do tipo de instrução e execução do microprograma que vai executar a instrução.

ULA *(Unidade Lógica e Aritmética)*
Responsável pelas operações lógicas e aritméticas necessárias para a execução das instruções.

Ciclo de Execução das Instruções da Linguagem de Máquina



(a) busca da próxima instrução
(b) incrementar PC
(c) decodificação do tipo de instrução
(d) cálculo dos endereços dos operandos
(e) carrega operandos para registradores
(f) execução da operação na ULA
(g) escrita do resultado da operação
(h) retorno ao passo (a)

Cada operação de leitura é realizada pelos seguintes passos:

(a) UC carrega no MAR o endereço a ser lido (PC)
(b) UC gera sinal de controle Read para a MP
(c) o conteúdo do endereço indicado por MAR é transferido para o MBR

Cada operação de gravação é realizada pelos seguintes passos:

(a) UC carrega no MAR o endereço a ser atualizado
(b) UC carrega no MBR a informação a ser gravada
(c) UC gera sinal de controle write para a MP
(d) a informação contida no MBR é transferida para a MP

Por razões econômicas e por simplicidade de projeto, programas no nível de linguagem de máquina dos computadores mais modernos são executados por um interpretador rodando no nível de microprogramação. O conjunto de instruções disponíveis para o programador em cada nível é chamado de ***"Instruction Set"*** do nível. O número de instruções deste conjunto varia de máquina para máquina, sendo em geral de 20 a 300. Desta forma classificam-se, as máquinas em 3 categorias:

CISC *(Complex Instruction Set Computer)*
Muitas instruções complexas/específicas e não muito gerais, cada instrução é executada, em geral, em muitos ciclos de máquina.

RISC *(Reduced Instruction Set Computer)*
Poucas instruções, porém bastante simples e bem gerais, devido à sua simplicidade, quase todas executadas em 1 ciclo de máquina.

CRISC *(Complex but Reduced Instruction Set Computer)*
Junção dos dois tipos de arquiteturas, em geral usadas em máquinas especializadas em computação gráfica, processamento de imagens, processamento paralelo, inteligência artificial, etc.

Abaixo apresentamos, como exemplo, um pequeno simulador de Microprograma para podermos entender melhor a sua execução.

Simulador de Microprograma

```
TYPE
  word = .......
  adress = ......
  MP = array [0 ... 4095] of word
Procedure Interpreter (mem: MP; AC: word; StartingAdress: adress)
VAR
  PC, Datalocation: adress;
  IR, data: word;
  DataIsNeeded: boolean;
  InstrType: integer;
  Run-bit: 0 .. 1;
Begin
```

```
PC:=  StartingAdress;
Run-bit:= 1;
While Run-bit = 1 do
    Begin
        IR := mem [pc];
        PC:= PC + 1;
        DetermInstrType (InstrType, IR);
        FindData (InstrType, IR);
        If DataisNeeded then
            data:= mem [Datalocation];
        Execute (InstrType, data, mem, AC, PC, Run-bit);
    END;
END;
```

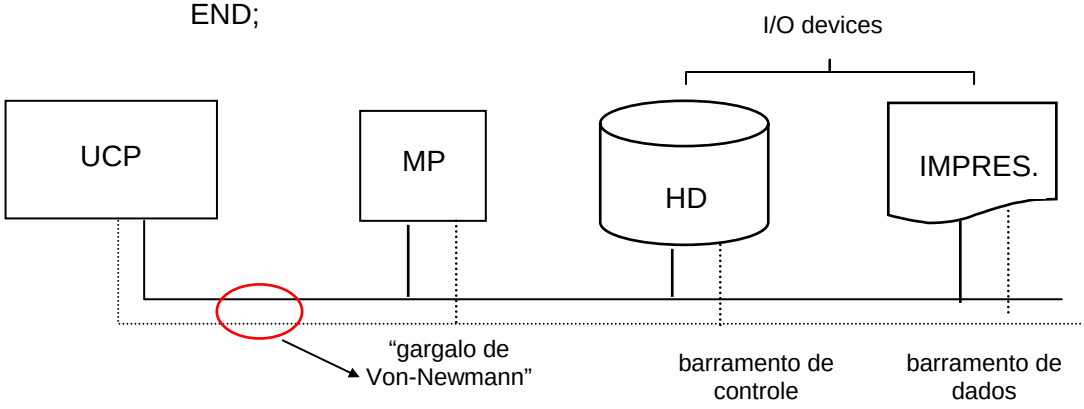


Fig 2.5 - O Gargalo de Von-Newmann

Para terminar, observamos que o ***Modelo de Von-Newman*** apresenta um problema estrutural chamado ***Gargalo de Von-Newman***. Este problema acontece devido ao elevado tráfego de dados e informações entre a UCP e a MP. Para amenizar este problema, criou-se a ***Memória Cache***, que é uma memória volátil de alta velocidade e cujo tempo de acesso a um dado nela contido é muito menor que o tempo de acesso caso o dado estivesse na memória principal.

Fig 2.6 - Hierarquia de Memória

Exercício: Supondo uma MP de 16 Mb e com tamanho de palavra de 32 bits, qual deve ser o tamanho do registrador de endereços MAR ? (1 byte = 8 bits)

16 Mbyte = $2^4 . 2^{20} = 2^{24}$ bytes
1 palavra = 32 bits = 2^5 bits = 4 bytes = 2^2 bytes

nº de endereços $= \frac{16Mbytes}{32bits} = \frac{2^{24}bytes}{2^2 bytes} = 2^{22} = 22$

2.2 Histórico dos Sistemas Operacionais

A evolução dos SOs está, em grande parte, relacionada ao desenvolvimento dos computadores (HW) que a cada vez se tornam mais velozes, compactos e baratos. Abaixo apresentamos a evolução do HW dividida em cinco fases com suas principais características e as respectivas gerações de SOs.

- **PRIMEIRA GERAÇÃO** - **Válvulas (tubos de vácuo)**
- **SEGUNDA GERAÇÃO - Transistores**
- **TERCEIRA GERAÇÃO - Circuitos Integrados**
- **QUARTA GERAÇÃO - Circuitos integrados em larga escala (LSI, VLSI)**
- **QUINTA GERAÇÃO - Circuitos integrados em ultra larga escala (ULSI)**

2.2.1 Gerações de Hardware

Antes do surgimento dos computadores de primeira geração, diversas iniciativas vinham sendo tomadas.

A **Máquina de Somar de Pascal**, foi o primeiro aparelho dotado de capacidade para processar dados; desenvolvido em 1642. Em 1670, **Leibniz** criou uma máquina semelhante à máquina de Pascal, mais com capacidade de multiplicar e de dividir.

Em 1804, Joseph Marie Jacquard usou **cartões perfurados** para controlar as máquinas de tecelagens. Em 1822, Charles Babbage projetou aquilo que seria o primeiro modelo de um computador. A **Máquina analítica de Babbage** era capaz de executar operações matemáticas em uma seqüência especificada. Em 1890, **Hollerith** desenvolveu um sistema de codificar dados em **cartões perfurados**, possibilitando o seu uso em máquinas de processar dados.

O **Computador Mark I** foi o primeiro computador moderno, construído em 1944, pelo prof. AIKEN, da Universidade de Harvard. O computador usava relés eletromecânicos e incorporava a idéia da máquina de BABBAGE.

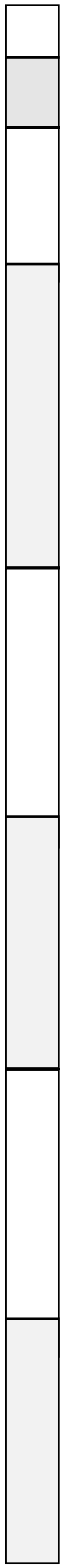
Em 1945, foi construído o primeiro computador, chamado **ENIAC** (Electronic Numerical Integrator and Computer)**,** que utilizava **válvulas eletrônicas**, na Universidade de Pensilvânia, pelos Drs. Eckert e Mauchley. Em 1949, Eckert e Mauchley construíram o **EDVAC** (Electronic Discrete Variable Computer)**,** o **primeiro computador que tinha um programa armazenado** para resolver problemas, uma vez que todos os computadores anteriores resolviam os problemas através de ligação externa de fios elétricos. O conceito de armazenamento de programas foi concebido pelo Dr. John Von-Neumann, da Universidade de Princeton.

Após o **ENIAC** e o **EDVAC**, o computador entrou na fase de produção industrial, motivado pêlos sucessos na resolução de cálculos complicados e pelo conseqüente aumento do interesse na sua utilização. Na fase industrial, os computadores começaram a ser classificados como de Primeira Geração, de acordo com a incorporação de processos tecnológicos.

Com o desenvolvimento da indústria, muitas empresas foram fundadas ou investiram no setor, como a Sperry e a IBM, o que levou á criação dos primeiros computadores para aplicações comerciais. A primeira máquina fabricada com esse propósito e bem-sucedida foi o **UNIVAC** (Universal Automatic Computer)**,** criado para ser usado no censo americano de 1950.

A seguir apresentamos um diagrama

ANO



1642 Máquina de somar de Pascal

1670 Máquina de Calcular de Leibniz

1804 Uso de cartões perfurador em teares por J. M. Jacquard
1822 Máquina Analítica de Charles Babbage

1890 Cartão Perfurado codificado em processamento de dados por H. Hollerith

1943 **PRIMEIRA GERAÇÃO DE COMPUTADORES** (a válvula eletrônica)
MARK I (réles), de Aiken da Universidade de Harvard
1944 ENIAC (válvula), de Mauchly e Eckert da Universidade da Pensylvania

1948 EDVAC (válvula e programa interno), de Eckert-Mauchley e Von-Neumann da Universidade de Princeton

1950 UNIVAC I criado especialmente para o censo americano

1953 **SEGUNDA GERAÇÃO DE COMPUTADORES** (transistores)
1954 TRADIC primeiro computador digital construído com transistores da AT&T
Linguagens de Programação ***Fortran*** (1954), ***Cobol*** (1959) e ***Algol*** (1960)

Processamento *Batch* através da execução de tarefas seqüenciais.
Independência de dispositivos através do *IOCS* e o conceito de canal

1963 **TERCEIRA GERAÇÃO DE COMPUTADORES** (com circuitos integrados)
1964 IBM Serie 360 e OS/360 introduzindo novos conceitos: *Multiprogramação,* Spooling e Buffering.

MULTICS do MIT e da DEC introduz o conceito de *time-sharing*
1969 Unix sucessor do MULTICS escrito em linguagem de alto nível ***C***

1981 **QUARTA GERAÇÃO DE COMPUTADORES** (integração LSI e VLSI)
Circuitos Integrados em larga escala (LSI) e muito grande escala (VLSI)
DOS do IBM/PC da Microsoft, SO para microprocessadores de 16 bits
VMS da DEC para minicomputadores da linha VAX 11/780
Multiprocessamento e *Arquiteturas Paralelas e Vetoriais*
Redes de Computadores – WANS e LANS
Banco de Dados;

1991 **QUINTA GERAÇÃO DE COMPUTADORES** (integração ULSI)
Estações de trabalho com SO do tipo UNIX like, com alto poder de Processamento Gráfico aplicadas a sistemas de CAD/CAM
Computação e Processamento distribuídos e Arquitetura Cliente-Servidor
Sistemas de Inteligência Artificial

Fig 2.7 - Cronologia dos Computadores

A seguir apresentamos as gerações de SOs e suas principais características.

PRIMEIRA GERAÇÃO - Início dos anos 40 a meados dos anos 50

- Não há sistema operacional;
- Necessidade de conhecimento profundo do HW, pois a programação era feita em painéis, através de fios, utilizando linguagem de máquina.

SEGUNDA GERAÇÃO - Início dos anos 50 a meados dos anos 60

- Surgimento das primeiras Linguagens de programação, como o **Assembly** e **Fortran**, os programas deixaram de ser feitos diretamente no HW, facilitando o processo de desenvolvimento de SW.
- Sistemas operacionais funcionavam com **processamento batch**. Programas passam a ser perfurados em cartões, que submetidos a uma leitora, os gravava em uma fita de entrada. Esta fita, pode então posteriormente ser lida pelo computador, que executava um programa de cada vez, gravando o resultado em do processamento em uma fita de saída. Veja a descrição do Sistema Batch Sofisticado (2.6.2).
- SOs passam a ter o seu próprio conjunto de rotinas de entrada/saída (**IOCS**- input/output control system). O IOCS eliminou a necessidade de os programadores terem que desenvolver suas próprias rotinas de leitura/gravação específicas para cada dispositivo periférico, criando o conceito de **independência de dispositivos E/S;** Avanços em nível de HW, principalmente na linha 7094 da IBM, permitiu a criação do conceito de **canal**, que veio permitir a transferência de dados entre dispositivos de entrada/saída e memória principal de forma independente da UCP. Nessa fase, destacam-se os sistemas **FMS** (Fortran Monitor System), IBSYS e o SO do computador do IBM 7094.

TERCEIRA GERAÇÃO - Meados dos anos 60 a final de 80

- Evolução dos processadores de entrada/saída permitiu que, enquanto um programa esperasse por uma operação de leitura/gravação, o processador executasse um outro programa, criando o conceito de **multiprogramação** (SO multiprogramado ou ***multitarefa***).
- Com o objetivo de reduzir o tempo de resposta no atendimento aos requisitos dos usuários (através de terminais on-line) a multiprogramação evoluiu até a criação dos sistemas **time-sharing** (tempo compartilhado).
- **Sistemas de propósito geral** (IBM/360, sistema operacional "OS" em 1964). Os sistemas de propósito geral são em geral associados a grande sobrecarga de utilização, de aprendizado demorado, grande tempo de depuração de erros e de difícil manutenção - grandes e caros ! Exceção - sistema UNIX, desenvolvido para máquinas de pequeno porte (minicomputador PDP-7 da Digital Equipment Corporation - DEC).
- **Sistemas de tempo real** foram criados para controle de processos.

QUARTA GERAÇÃO - Início dos anos 80 até início de 90.

- Surgimento dos microprocessadores e do sistema operacional DOS (Disk Operation System);
- No final dos anos 80, aplicações que exigiam um enorme volume de cálculos, puderam ser desenvolvidas com o suporte fornecido pelo SO ao **multiprocessamento.**. Assim, foi possível a execução de mais de um programa simultaneamente, ou até a execução de um mesmo programa por mais de um computador O surgimento de processadores vetoriais e técnicas de paralelismo em diferentes níveis tornou os computadores ainda mais poderosos.

QUINTA GERAÇÃO - 1991 até hoje

- Grandes evoluções em HW, SW e telecomunicações, vêm permitindo o desenvolvimento de sistemas multimídia, bancos de dados distribuídos, inteligência artificial (sistemas especialistas e redes neuronais) com cada vez maior capacidade.
- A forma de interação com os computadores sofrerá também modificações diversas: novas interfaces homem-máquina (MMI - Mem Machine Interface) serão utilizadas com o uso de linguagens naturais, reconhecimento de voz e de assinaturas.
- A evolução das telecomunicações, com a crescente capacidade de transmissão de dados, tornará a WEB (World Wide Web da Internet) uma realidade na interação entre usuários, permitindo inclusive a transmissão de shows ao vivo e interação entre ambientes 3D (VRML).

2.3 Classificação dos Sistemas Operacionais

2.3.1 Introdução

Os tipos de SOs e sua evolução estão intimamente relacionados com a evolução do HW e das aplicações por ele suportadas. Abaixo apresentamos uma classificação para os SOs, suas características, vantagens e desvantagens.

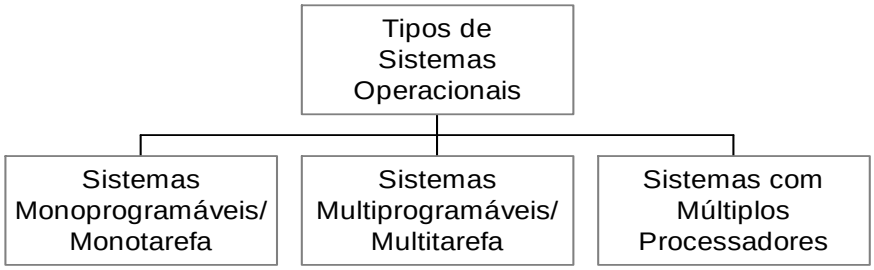


Fig 2.8 - Tipos de SOs

2.3.2 Sistemas Operacionais Monoprogramáveis ou Monotarefa

Os primeiros SOs eram tipicamente voltados para a execução de um único programa (job). Qualquer outro programa, para ser executado, deveria aguardar o término do programa corrente. Os SOs **Monoprogramáveis** se caracterizam por permitir que o processador, a memória e os periféricos permaneçam exclusivamente dedicados à execução de um único programa.

Os SOs Monoprogramáveis estão tipicamente relacionados aos computadores de grande porte (mainframes). Posteriormente com a introdução dos primeiros computadores pessoais e estações de trabalho, estes tipos de sistemas voltaram a ser desenvolvidos para atender máquinas que, na época, eram utilizados por apenas um usuário o que caracterizou o surgimento dos SOs **Monotarefa**.

As principais características são:

- processador dedicado a um único usuário;
- processador, memória e outros recursos do sistema são mal utilizados;
- SO de simples implementação.

2.3.3 Sistemas Operacionais Multiprogramáveis ou Multitarefa

Os SOs **Multiprogramáveis** vieram para substituir os Monoprogramáveis e são mais complexos porém mais eficientes. Enquanto nos SOs Monoprogramáveis existe um único programa utilizando os diversos recursos do sistema, nos Multiprogramáveis vários programas dividem esses mesmos recursos. As vantagens do uso destes sistemas são o aumento da produtividade dos seus usuários e a redução de custos, a partir do compartilhamento dos diversos recursos do sistema.

Os SOs **Multiprogramáveis** estão tipicamente associados aos computadores de grande porte (mainframes) e minicomputadores, onde existe a idéia do sistema sendo utilizado por vários usuários (**Multiusuário**). Já nos computadores pessoais e estações de trabalho, apesar de existir um único usuário interagindo com o sistema (que é dito **Monousuário**), é possível que ele execute diversas tarefas concorrentemente ou mesmo simultaneamente (**Multiprocessamento**) o que caracterizou o surgimento dos SOs **Multitarefa**.

As principais características dos SOs **Multiprogramáveis** são:

- vários usuários compartilham o processador e os recursos do sistema, diminuindo os custos de utilização do sistema;
- o SO se preocupa em gerenciar o acesso concorrente aos seus diversos recursos, como memória, processador e periféricos, de forma ordenada e protegida, entre os diversos usuários.

Os SOs Multiprogramáveis/Multitarefa podem ser classificados pela forma com que suas aplicações são gerenciadas, podendo ser divididos conforme mostra o gráfico abaixo.

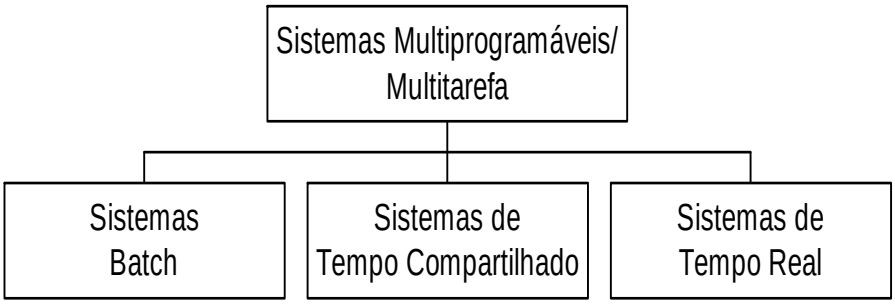


Fig 2.9 - Classificação dos SOs Multiprogramáveis

2.3.4 SO Multiprogramado Batch

Os SOs Batch (lote) foram os primeiros SOs Multiprogramáveis a serem implementados e caracterizam-se por terem seus programas, quando submetidos, armazenados em disco ou fita, onde esperam ser carregados para execução seqüencial pelo monitor (veremos adiante nos sistemas Batch Simplificado - 2.6.1 e Batch Sofisticado 2.6.2).

Esses sistemas, quando bem projetados, podem ser bastante eficientes, devido à melhor utilização do processador. Entretanto, podem oferecer tempos de resposta longos, em face do processamento puramente seqüencial e com uma variação alta dos seus **tempos de execução** (**tempo de parede** ou **elapsed time** ou **wall clock time,** e não **tempo de UCP**)

Hoje em dia, em alguns sistemas operacionais ainda se executam programas de forma Batch (não interativa) como: compilação, link-edições, sorts, balance-line, etc.

2.3.5 SO Multiprogramado de Tempo Compartilhado - Time-Sharing

Os SOs Multiprogramados de Tempo Compartilhado permitem a interação dos usuários com o sistema, basicamente através de terminais de vídeo e teclado (interação on-line). Dessa forma, o usuário pode interagir diretamente com o sistema em cada fase do desenvolvimento de suas aplicações.

Esses sistemas possuem uma **Linguagem de Controle** que permite ao usuário comunicar-se diretamente com o SO para obter e dar informações diversas.

O SO aloca para cada usuário uma fatia de tempo de execução do processador o qual é chamada de **time-slice** (ou quantum de tempo). A execução do programa do usuário é interrompida após este tempo ou caso o programa peça a execução de uma SVC (chamada ao supervisor - *supervisor call*) para a leitura/gravação em algum periférico de E/S.

O **processo** (programa do usuário executando) cuja execução foi suspensa por fim do time-slice entra numa "**fila de execução de processos prontos**" para ser processado mais tarde. O processo que pediu uma **SVC** entra em **estado de espera** (até que a SVC tenha sido executada). Poderíamos traçar então o seguinte diagrama de estados dos processos executando em um SO Multiprogramado.

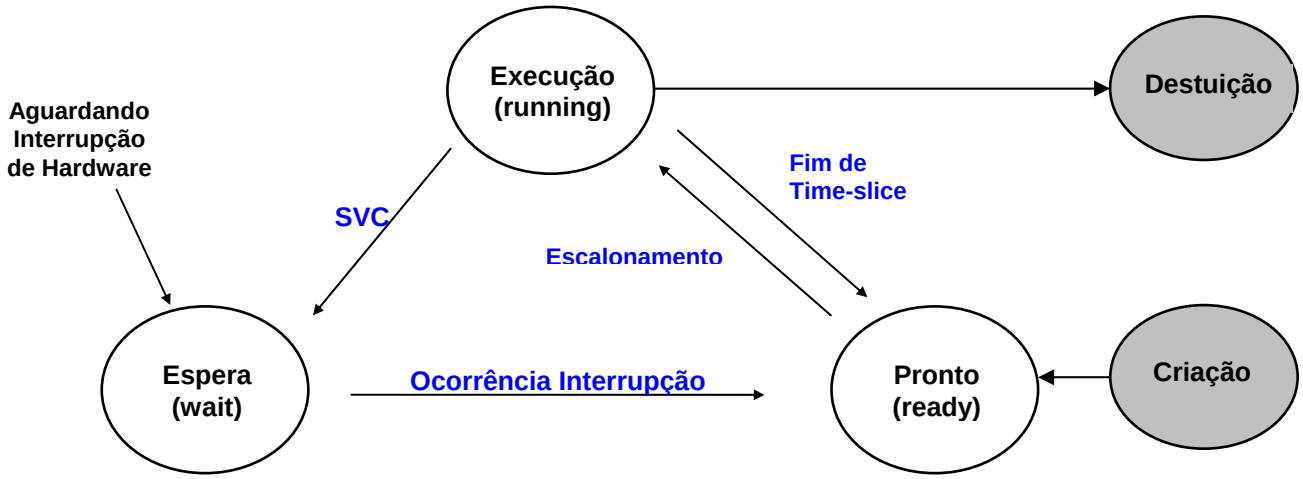


Fig 2.10 - Diagrama de transição de estados de um processo

2.3.6 SO Multiprogramado de Tempo Real - Real-Time

Os SOs de Tempo Real são semelhantes aos sistemas Time-Sharing. A maior diferença é o tempo de resposta, exigido no processamento das aplicações, que não pode variar para não comprometer a segurança do sistema.

Neste sistema não existe o conceito de time-slice. Um programa executa o tempo que for necessário, ou até que outro programa com maior prioridade tome o seu lugar. Esta importância ou prioridade de execução é controlada pela própria aplicação e não pelo sistema operacional. Exemplos do uso de SOs de Tempo Real:

- Sistema de monitoração de refinarias de petróleo;
- Sistemas de posicionamento dinâmica de plataformas de petróleo.

2.3.7 Sistemas Operacionais com Múltiplos Processadores

Os SOs com Múltiplos Processadores caracterizam-se por possuírem duas ou mais UCPs interligadas, trabalhando em conjunto. Um fator chave na classificação de SOs com Múltiplos Processadores é a forma de comunicação entre as UCPs e o grau de compartilhamento da memória e dos dispositivos de entrada e saída. Em função destes fatores, podemos classificar os SOs com **Fortemente** ou **Fracamente Acoplados** conforme a figura abaixo:

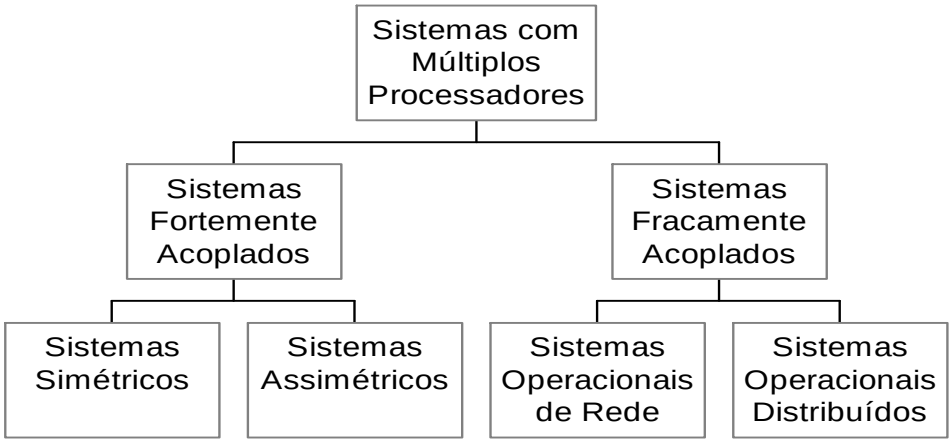


Fig 2.11 - SOs com Múltiplos Processadores

SOs Fracamente Acoplados permitem que as máquinas e os usuários de um sistema distribuído sejam completamente independentes. Este é o caso, por exemplo, de uma rede de PCs que compartilham alguns recursos como impressoras laser, servidores de Banco de Dados, via uma rede local (LAN).

SOs Fortemente Acoplados permitem que os programas dos usuários sejam divididos em sub-programas para execução simultânea em mais de um processador.

2.3.7.1 Sistemas Fortemente Acoplados

2.3.7.1.1 Sistemas Assimétricos

2.3.7.1.2 Sistemas Simétricos

2.3.7.1.3 Multiprocessamento

2.3.7.2 Sistemas Fracamente Acoplados

2.3.7.2.1 Sistemas Operacionais de Rede

2.3.7.2.2 Sistemas Operacionais Distribuídos

2.4 Sistemas Operacionais Multiprogramáveis

Os SOs Multiprogramáveis surgiram devido a baixa utilização dos recursos do sistema presente nos SOs Monoprogramáveis. Estudos mostraram que em SOs Monoprogramáveis, a UCP é utilizada em aproximadamente 30% do tempo, enquanto em SOs Multiprogramáveis o tempo de utilização sobe para até 90%. Outro aspecto a considerar é a subutilização da memória, pois um programa que não ocupe totalmente a memória principal ocasiona a existência de áreas livre que representa um desperdício de um recurso tão caro.

As principais funções desejadas para SOs Multiprogramáveis podem ser colocadas em quatro categorias abaixo relacionadas:

Flexibilidade

- facilidade para carregar programas para execução;
- interpretar linguagem de comandos e de controle - Shell;
- suportar interação com usuário através de terminais - “Time-Sharing”;
- facilidade para controle, administração, contabilização do uso dos recursos do HW;

Concorrência

- compartilhar memória entre os diversos programas dos usuários;
- compartilhar o uso da UCP para a realização das tarefas dos usuários;
- sobrepor E/S e processamento;

Compartilhamento

- compartilhar a UCP selecionando o próximo programa a executar - “Scheduling”;
- compartilhamento de dados;
- reutilização de programas criados por outros - “código reentrante”;
- eliminação de redundâncias para tratamento de E/S;

Proteção

- tratamento de erros e exceções de HW;
- tratamento de interrupções;
- proteger o programa contra violações de outros programas - “Memory Protection”;

Os principais problemas a serem resolvidos são :

- Como revezar a UCP entre os programas dos usuários ?
- Como proteger os programas entre si ?
- Como sincronizar atividades que são mutuamente exclusivas (Ex.: impressão, leitura e/ou gravação em dispositivos de acesso seqüencial)

A concorrência é fundamental para entendermos o funcionamento de um sistema operacional Multiprogramável. A possibilidade de periféricos e dispositivos funcionarem simultaneamente com a UCP, permitiu a execução de tarefas concorrentes.

Enquanto nos SOs Monoprogramáveis, somente um programa pode estar residente na memória (com conseqüente dedicação exclusiva da UCP à execução desse programa), nos SOs Multiprogramados, vários programas podem estar residentes em memória, concorrendo pela utilização da UCP.

Dessa forma quando um programa solicita uma operação de entrada/saída, outros programas poderão estar disponíveis para utilizar o processador. Nesse caso a UCP permanece menos tempo ociosa e a memória principal é utilizada de forma mais eficiente, pois existem vários programas residentes se revezando na utilização da UCP.

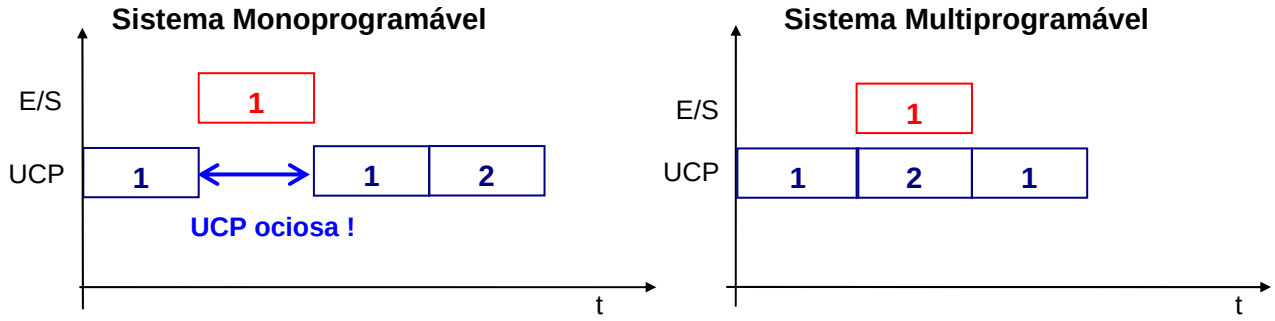


Fig 2.12 - Comparação entre SOs Monoprogramável x Multiprogramável

2.5 Conceito de Interrupção e Exceção

Durante a execução de um programa, alguns eventos podem ocorrer durante o seu processamento, obrigando a intervenção do SO. Este tipo de intervenção é chamada de **Interrupção** ou **Exceção**, e pode ser resultado da execução de instruções do próprio programa, gerado pelo SO ou por algum dispositivo de HW. Nestas situações o fluxo de execução do programa é desviado para uma rotina especial de tratamento. O que diferencia uma **Interrupção** de uma **Exceção** é o tipo de evento que gera esta condição.

Uma **Interrupção** é gerada pelo SO ou por algum dispositivo e, neste caso, independe do programa que está sendo executado. Um exemplo é quando um periférico avisa à UCP que está pronto para transmitir algum dado. Neste caso, a UCP deve interromper o programa para atender a solicitação do dispositivo. Este tratamento é feito pelo mecanismo definido a seguir.

2.5.1 Tratamento de Interrupção

No momento em que a unidade de controle detecta a ocorrência de algum tipo de interrupção, ela interrompe a execução do programa e salva o PC e a PSW do processo corrente na sua área de STACK, a seguir o controle da execução é passado para o sistema operacional que salva o restante das informações do contexto do processo que estava executando. Feito isto o controle é então desviado para a rotina do SO responsável pelo tratamento da interrupção.

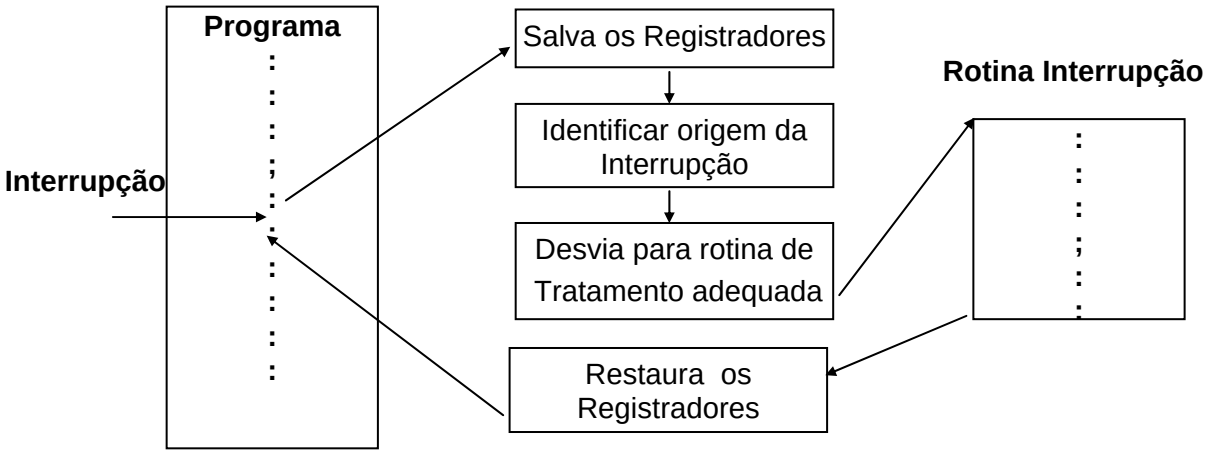


Fig 2.13 - Mecanismo de Interrupção

Não existe apenas um único tipo de interrupção e sim diferentes tipos que devem ser atendidos por diversas rotinas de tratamento. No momento que uma interrupção acontece, a UCP deve saber para qual rotina de tratamento deverá ser desviado o fluxo de execução. Essa informação está em uma estrutura do SO chamado **Vetor de Interrupção**, que contém a relação de todas as rotinas de tratamento existentes, associadas a cada interrupção.

Todo o procedimento para detectar a interrupção, salvar o contexto do programa e desviar para a rotina de tratamento é denominado **Mecanismo de Interrupção.** Este mecanismo é realizado, na maioria das vezes, pelo HW dos computadores, e foi implementado pelos projetistas para criar uma maneira de sinalizar ao processador eventos assíncronos que possam ocorrer no sistema.

No caso de múltiplas interrupções ocorrerem, o processador deve saber qual interrupção será tratada primeiro, o que é feito através da prioridade que é atribuída pelo sistema operacional a cada interrupção. Normalmente o HW possui um dispositivo denominado **Controlador de Pedidos de Interrupção** que avalia e ordena os pedidos.

A interrupção é o mecanismo que torna possível a implementação da concorrência nos computadores, sendo o fundamento básico dos sistemas Multiprogramáveis.

Inicialmente os SOs apenas implementavam o Mecanismo de Interrupção. Com a evolução dos sistemas foi introduzido o conceito de **Exceção**. Uma exceção é resultado direto da execução de uma instrução do próprio programa. Situações como a divisão por zero ou a ocorrência de um overflow caracterizam essa situação.

O **Mecanismo de Tratamento de Exceção** é semelhante ao Mecanismo de Interrupção, porém em muitos casos, as rotinas de tratamento podem ser escritas pelo próprio programador (linguagem C, C++, JAVA, etc).

A principal diferença entre **Exceção** e **Interrupção** é que a primeira é gerada por um evento síncrono, enquanto que a segunda é gerada por eventos assíncronos.

2.6 Evolução das Funções do Sistema Operacional

Durante a fase dos computadores de segunda geração diversos fatores permitiram o desenvolvimento do embrião do primeiro SO com o objetivo de aumentar a eficiência dos programadores/operadores no uso do sistema computacional.

O surgimento do transistor proporcionou um aumento da velocidade e da confiabilidade do processamento, ao passo que o aparecimento da linguagem Assembly permitiu a criação de bibliotecas de rotinas comuns escritas em Assembly para tratamento de operações de E/S (**Input/Output Control System – IOCS**) tratando das diferentes características físicas de cada periférico. Assim o trabalho de programação foi bastante facilitado pois o **IOCS** eliminou a necessidade de os programadores desenvolverem suas próprias rotinas de leitura/gravação específicas para cada dispositivo periférico. Esta facilidade criou o conceito de **Independência de Dispositivos.**

A seguir apresentamos as principais características do primeiro Sistema Operacional criado que se baseava no processamento em lote (Batch) de uma tarefa por vez (Monotarefa). Mais tarde, como uma evolução natural surgiu o SO Batch (Multitarefa)

2.7 Sistema Batch Simplificado - Monotarefa

2.7.1 Monitor e Linguagem de Controle

Este SO tem sua execução baseada em fitas magnéticas. Vários passos eram necessários para executar os programas. Assim para facilitar as atividades do programador/operador foi criada a **Linguagem de Controle de Programas** (**JCL - Job Control Language**) para automatizar as etapas necessárias para a execução de um programa.

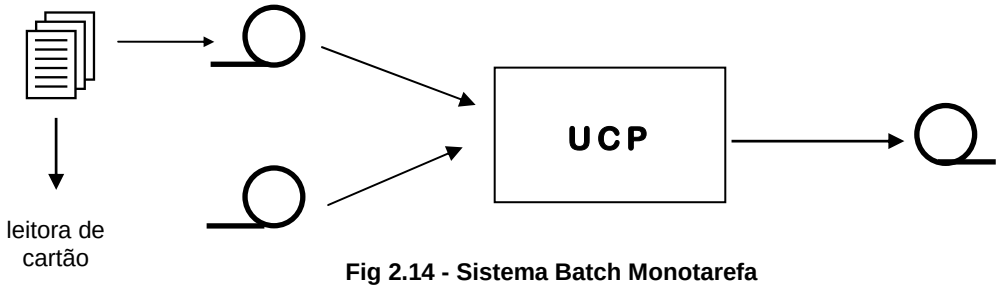


Fig 2.14 - Sistema Batch Monotarefa

As principais características desse sistema são:

- Mais de um lote (batch) para execução na UCP, acelera o ciclo de execução, cria a figura do operador de computador e libera o programador.
- UCP ociosa entre a carga de diferentes lotes.
- **Monitor** residente: programa para carregar os programas a serem executados a partir das unidades de fita.
- Cartões de controle: servem para instruir o monitor na carga dos programas. Ex.: $JOB, $FTN, $END.

2.7.2 Modos Monitor/Usuário

- Melhoria em HW possibilitando maior robustez ao SO.
- Uso opcional das rotinas de E/S para manipulação de periféricos, com isso obtêm-se maior eficiência e segurança na execução dos programas - estas rotinas fazem parte do monitor. Portanto agora parte do programa é executada pelo monitor (**Modo Monitor**) e parte pelo programa propriamente dito (**Modo Usuário**). Funcionando desta forma, a interceptação de erros de execução (através do "trapeamento de erros") podia ser identificada pelo monitor, o que aumentava a segurança do sistema.

2.7.3 Instruções Privilegiadas e Modos do Processador

Apenas em **modo monitor** o HW permite executar determinadas instruções, as chamadas **instruções privilegiadas**. Então surge o problema de como se passar do estado usuário para o estado monitor ? Por exemplo: como se consegue executar uma instrução privilegiada de E/S ?

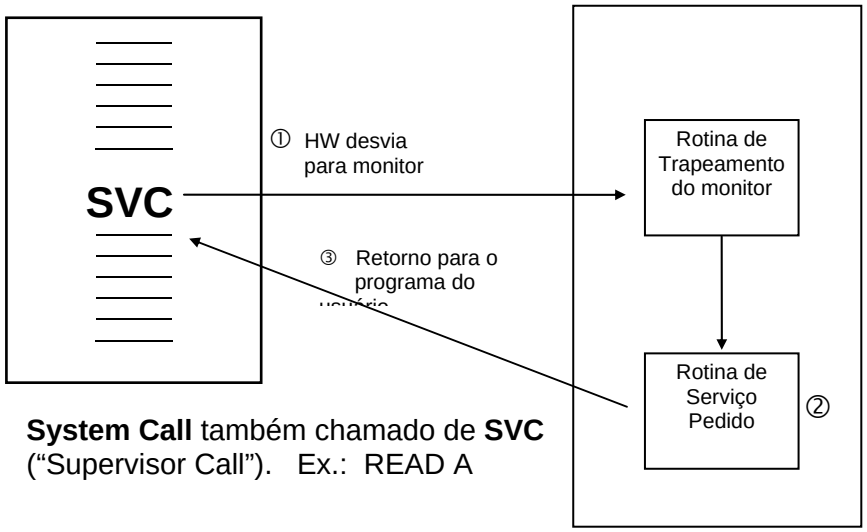


Fig 2.15 - Execução de uma SVC

1. Programa do usuário executa uma SVC para executar uma rotina de monitor. (Ex.: READ).

2. Parâmetros na chamada indicam o serviço pedido. No momento do desvio do programa do usuário para Rotina de Trapeamento o estado do processo (programa executando) passa para monitor (HW).

3. Na volta da Rotina de Serviço o modo passa de monitor para usuário (HW) Exemplo de Instruções Privilegiadas: Instruções de E/S; Parada do processador (HALT); Mudança no modo do processador para monitor.

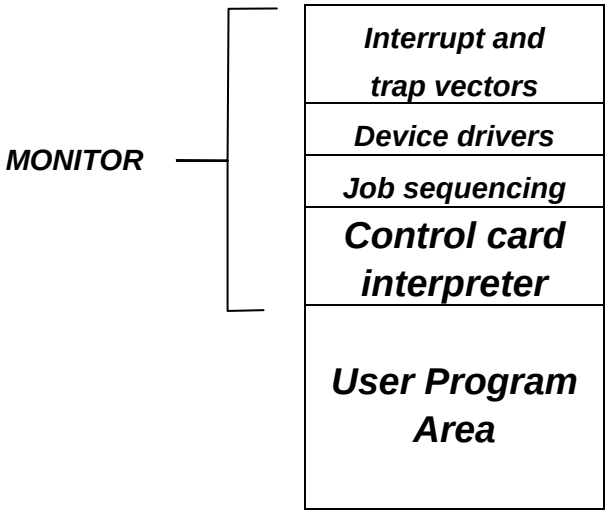


Fig 2.16 - Estrutura do Monitor no Sistema Batch Monotarefa

2.8 Sistema Batch Sofisticado – Multitarefa

Os periféricos de E/S são dispositivos eletromecânicos e portanto lentos em relação a UCP e memória, que são dispositivos eletrônicos. O tempo de switching (chaveamento - ciclo de máquina) da UCP é da ordem de microssegundos enquanto o tempo médio de acesso a disco é da ordem de dezenas de milissegundos. Para resolver este problema de diferença de velocidades de acesso diversas técnicas foram sendo introduzidas para reduzir o **Blocked-Time**.

2.8.1 Operações "Off-line"

- cartões de entrada e imagem de relatórios gravados em fita magnética;
- independência de dispositivos (E/S lógica). O sistema operacional torna transparente ao programa do usuário o acesso à fita e entrega/recebe um registro por vez com a imagem do cartão ou linha do relatório de saída;
- cartões de controle mapeiam dispositivos lógicos em físicos; mais de um sistema leitora "cartões-fita" ou "fita-impressora" otimizava a operação, porém apenas quando a fita estivesse cheia é que então a UCP pode continuar o processamento.

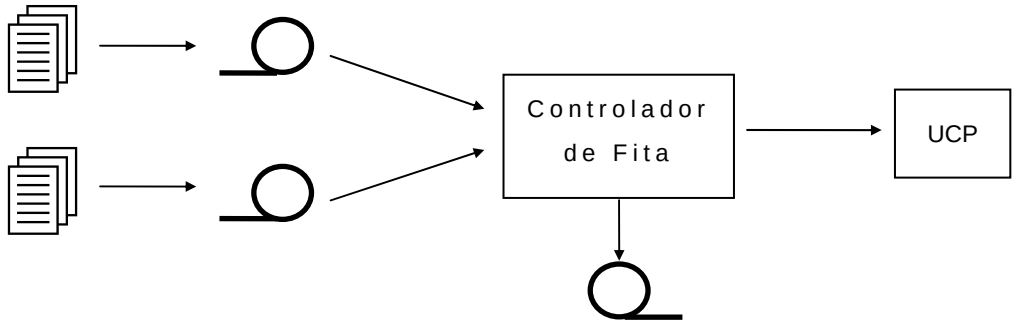


Fig 2.17 - Sistema Batch Multitarefa

2.8.2 Operações de Entrada e Saída

Nos primeiros SOs, a comunicação entre a UCP e os periféricos era controlada por um conjunto de instruções especiais, denominadas **instruções de entrada/saída**, executadas pela própria UCP. Essas instruções continham detalhes específicos de cada periférico, como quais trilhas e setores de um disco deveriam ser lidos ou gravados em determinado bloco de dados. Esse tipo de instrução limitava a comunicação do processador a um grupo de dispositivos.

A implementação de um dispositivo chamado **controlador ou interface** permitiu à UCP agir de maneira independente dos dispositivos de E/S. Com esse novo elemento a UCP não se comunicava mais diretamente com os periféricos, mas sim através do **controlador** (Fig 2.18). Isso simplificou as instruções de E/S, por não ser mais preciso especificar detalhes de operação dos periféricos, tarefa essa realizada pelo controlador.

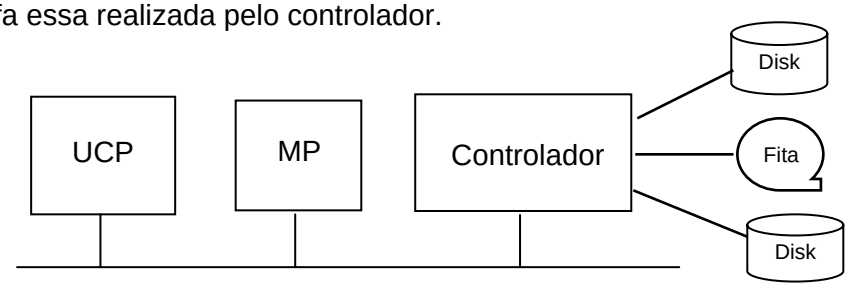


Fig. 2.18 - Controlador de E/S

Com a nova implementação, existiam duas maneiras básicas pelas quais o processador podia controlar as operações de E/S. Na primeira, a UCP sincronizava-se com o periférico para o início da transferência de dados e, após iniciada a transferência, o sistema ficava permanentemente testando o estado do periférico para saber quando a operação tinha chegado ao seu final. Este controle, chamado de **E/S controlada por programa** mantinha a UCP ocupada até o término da operação de E/S, o que caracterizava uma situação de **busy-waiting**, ocasionando um desperdício do tempo de UCP (Explique o porquê !).

Outra forma derivada da anterior, porém mais inteligente, era permitir que, após o início da transferência dos dados, a UCP ficasse livre para se ocupar de outras tarefas. Assim, em determinados intervalos de tempo, o SO deveria realizar um teste para saber do término ou não da operação de E/S em cada dispositivo, o que caracterizou o que chamamos de **polling**. Esse tipo de operação introduziu um certo grau de paralelismo de operações, visto que um programa poderia ser processado, enquanto outro esperava pelo término de uma operação de E/S. O problema desta implementação é que, no caso de existir um grande número de periféricos, o sistema tem que freqüentemente interromper o processamento dos programas para testar os diversos periféricos, já que é difícil determinar o momento exato do término das operações de E/S em andamento.

Com a implementação do **Mecanismo de Interrupção** no hardware dos computadores, as operações de *E/S* puderam ser realizadas de uma forma mais eficiente. Em vez de o sistema periodicamente verificar o estado de uma operação pendente, *o próprio controlador interrompia a UCP para avisar do término da operação*. Com esse mecanismo, denominado **E/S controlada por interrupção***,* a UCP, após a execução de um comando de leitura ou gravação, fica livre para o processamento de outras tarefas. O controlador por sua vez, ao receber, por exemplo, um sinal de leitura, fica encarregado de ler os blocos do disco e armazená-los em memória ou registradores próprios. Em seguida, o controlador, através de uma linha de controle, sinaliza uma interrupção ao processador. Quando a UCP atende a interrupção, a rotina responsável pelo tratamento desse tipo de interrupção transfere os dados dos registradores do controlador para a memória principal. Ao término da transferência, a UCP volta a executar o programa interrompido e o controlador fica novamente disponível para outra operação.

A operação de ***EIS* controlada por interrupção** é muito mais eficiente que a operação de ***EIS* controlada por programa**, já que elimina a necessidade de a UCP esperar pelo término da operação, além de permitir que várias operações de *E/S* sejam executadas simultaneamente. Apesar disso, essa implementação ainda sobrecarregava a UCP, uma vez que toda transferência de dados entre memória e periféricos exigia a intervenção da UCP. A solução para esse problema foi a implementação, por parte do controlador, de uma técnica de transferência de dados denominada **DMA - Direct Memory Access.**

A **técnica de DMA** permite que um bloco de dados seja transferido entre memória e periféricos, sem a intervenção da UCP, exceto no início e no final da transferência. Quando o sistema deseja ler ou gravar um bloco de dados, são passadas da UCP para o controlador informações como: onde o dado está localizado, qual o dispositivo de *E/S* envolvido na operação, posição inicial da memória de onde os dados serão lidos ou gravados e o tamanho do bloco de dados. Com estas informações, o controlador realiza a transferência entre o periférico e a memória principal, e a UCP é somente interrompida no final da operação. A área de memória utilizada pelo controlador na técnica de DMA é chamada **buffer de E/S***,* sendo reservada exclusivamente para este propósito**.**

No momento em que a transferência de **DMA** é realizada, o controlador deve assumir momentaneamente o controle do barramento. Como a utilização do barramento é exclusiva de um dispositivo, a UCP deve suspender o acesso ao barramento temporariamente durante a operação de transferência. Este procedimento não gera uma interrupção, e a UCP pode realizar tarefas desde que sem a utilização do barramento, como, por exemplo, um acesso à memória cache.

A extensão do conceito de **DMA** possibilitou o surgimento dos **canais de E/S**, ou somente canais, introduzidos pela IBM no Sistema 7094. O ***canal de E/S*** é um processador com capacidade de executar programas de E/S, permitindo o controle total sobre operações de entrada e saída. As instruções de E/S são armazenadas na memória principal pela UCP, porém o canal é responsável pela sua execução. Assim, a UCP realiza uma operação de *E/S,* instruindo o canal para executar um programa localizado na memória (programa de canal). Este programa especifica os dispositivos para transferência, buffers e ações a serem tomadas em caso de erros. O **canal de E/S** realiza a transferência e, ao final, gera uma interrupção, avisando do término da operação.

Um **canal de E/S** pode controlar múltiplos dispositivos através de diversos controladores (Fig. 2.19). Cada dispositivo, ou conjunto de dispositivos, é manipulado por um único controlador. O canal atua como um elo de ligação entre a UCP e o controlador.

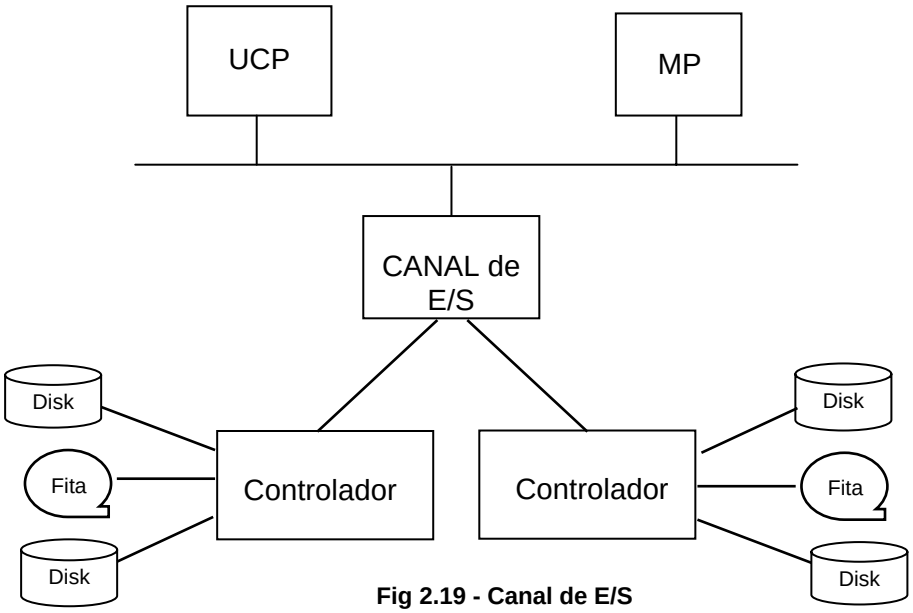


Fig 2.19 - Canal de E/S

A evolução do canal permitiu que este possuísse sua própria memória eliminando a necessidade de os programas de E/S serem carregados para a memória principal. Com essa nova arquitetura, várias funções de E/S puderam ser controladas com mínima intervenção da UCP**.** Este último estágio do canal é também denominado *de* **processador de E/S***,* embora seja comum encontrarmos os dois termos empregados indistintamente [Stalling].

2.8.3 Buffering

A técnica de **Buffering** consiste na utilização de uma área de memória para a transferência de dados entre os periféricos e a memória principal denominada Buffer de E/S. 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 UCP 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 UCP coloca o dado no buffer para um dispositivo de saída manipular (Fig. 2.20). A finalidade da técnica de Buffering é minimizar o problema da disparidade da velocidade de processamento existente entre a UCP e os dispositivos de E/S. mantendo na maior parte do tempo UCP e dispositivos de E/S ocupados.

A unidade de transferência usada no mecanismo de Buffering é o ***registro***. O tamanho do registro pode ser especificado em função da natureza do dispositivo (como uma linha gerada por uma impressora ou um caracter de um teclado) ou da aplicação (como um registro lógico definido em um arquivo).

O buffer deve possuir a capacidade de armazenar diversos registros de forma a permitir que existam dados lidos no buffer, mas ainda não processados (operação de leitura) ou processados, mas ainda não gravados (operação de gravação). Desta forma o dispositivo de entrada poderá ler diversos registros antes que a UCP os processe, ou a UCP poderá processar diversos registros antes de o dispositivo de saída realizar a gravação. Isso é extremamente eficiente, pois, dessa maneira, é possível compatibilizar a diferença existente entre o tempo em que a UCP processa os dados e o tempo em que o dispositivo de E/S realiza as operações de leitura e gravação.

As principais características da técnica de **Buffering** são:

- paralelismo entre UCP e E/S, já que a UCP é muito mais rápida que E/S
- leitura antecipadada de vários registros;
- registros colocados no buffer para posterior gravação;
- detecção do fim de E/S: via interrupção de HW;
- o sistema operacional gerencia buffers de cada periférico através de device drivers;
- o programa do usuário recebe/envia um registro a cada "chamada ao sistema" (SVC) para E/S;

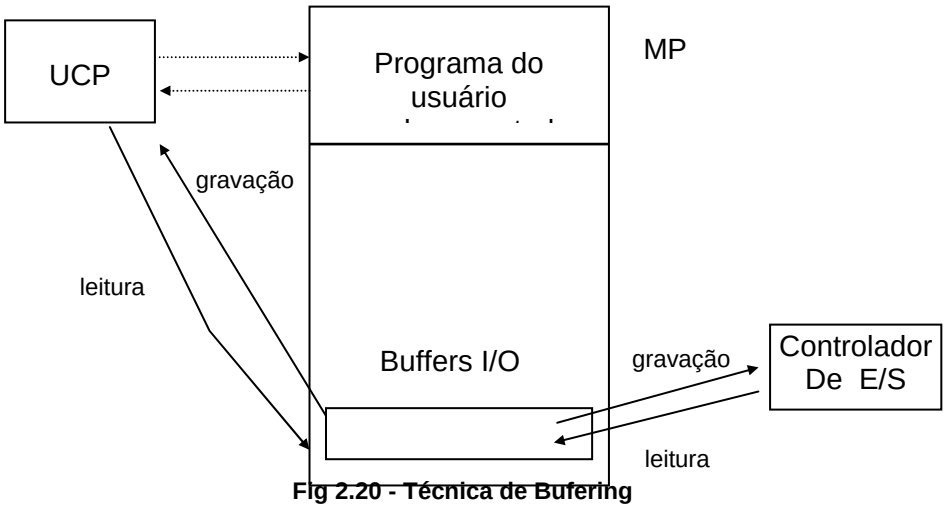


Fig 2.20 - Técnica de Bufering

2.8.4 Spooling ou Operações On-line

A técnica de **Spooling** (Simultaneous Peripheral Operation On-Line) foi introduzida no final dos anos 50 para aumentar a produtividade e a eficiência dos sistemas operacionais.

Naquela época, os programas dos usuários eram submetidos um a um para processamento pelo operador. Como a velocidade de operação dos dispositivos de entrada/saída é muito lenta se comparada à do processador, era comum que a IJCP ficasse ociosa à espera de programas e dados de entrada ou 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. Desta forma, a UCP poderia processar seqüencialmente cada job, 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 diretamente na impressora, poderia direcioná-las para uma outra fita, que depois seria impressa integralmente. Esta forma de processamento é chamada de **Spooling** e foi a base dos sistemas Batch.

A utilização de fitas magnéticas obrigava o processamento a ser estritamente seqüencial, ou seja, o primeiro job a ser gravado na fita era o primeiro a ser processado. Assim, se um job que levasse várias horas antecedesse pequenos jobs, seus tempos de resposta ficariam seriamente comprometidos. Com o surgimento de **dispositivos de acesso direto**, como os discos magnéticos, foi possível tornar o Spooling muito mais eficiente e, principalmente, permitir a eliminação do processamento estritamente seqüencial, com a atribuição de prioridades aos jobs.

A técnica de **Buffering**, como já apresentamos, permite que um job utilize um buffer concorrentemente com um dispositivo de E/S. O Spooling, basicamente, utiliza o disco como um grande buffer, permitindo que dados sejam lidos e gravados em disco, enquanto outros jobs são processados.

Um exemplo dessa técnica está presente quando impressoras são utilizadas. No momento em que um comando de impressão é executado por um programa, as informações que serão impressas são gravadas em um arquivo em disco (arquivo de spool), para ser impresso posteriormente pelo sistema (Fig. 2.21). Dessa forma, situações como *a* de um programa reservar a impressora, imprimir uma linha e ficar horas para continuar a impressão não acontecerão. Essa implementação permite maior grau de compartilhamento na utilização de impressoras.

Atualmente a técnica de **Spooling** é implementada na maioria dos SOs, fazendo com que tanto a UCP quanto os dispositivos de E/S sejam aproveitados de forma mais eficiente.

As principais características da técnica de **Spooling** são portanto:

- "spool" = carretel; "spooling" uso do disco para enfileirar imagens de cartões e relatórios para serem gravados/lidos posteriormente;
- o sistema operacional controla a localização de cada "job" e cada "relatório" no disco através de uma estrutura de dados chamada "tabela de spooling";
- há concorrência entre submissão, execução e impressão de diferentes serviços;
- Job-spool - grupo de serviços aguardando execução seguindo algum critério de escalonamento ("Scheduling").

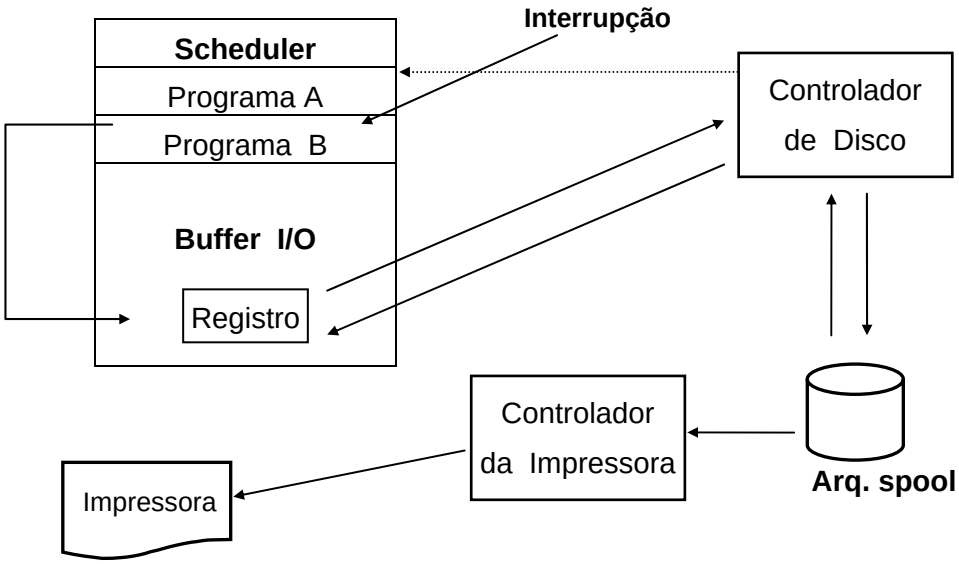


Fig 2.21 - Técnica de Spooling

3 Estrutura do Sistema Operacional

3.1 Introdução

Um Sistema Operacional é formado por um conjunto de rotinas (procedimentos) que oferecem serviços aos usuários do sistema e suas aplicações, bem como a outras rotinas do próprio sistema. Este conjunto de rotinas é chamado **Núcleo do Sistema ou Kernel**. As principais funções do **Kernel** são:

- **Tratamento de Interrupções;**
- **Criação e Eliminação de Processos;**
- **Sincronização e Comunicação entre Processos;**
- **Escalonamento e controle dos Processos (Gerência de Processos)**
- **Gerência de Memória;**
- **Gerência do Sistema de Arquivos;**
- **Operações de E/S (Gerência de Periféricos);**
- **Contabilização e Segurança do Sistema;**

A estrutura do Kernel do SO, isto é, a maneira como o código do SO é organizado e o inter-relacionamento entre os seus diversos componentes, pode variar conforme a concepção de projeto do SO. Existem basicamente três abordagens que estudaremos nesse capítulo. São elas: **Monolítico**, **Em Camadas** e **Micro-Núcleo** (**Micro-Kernel**). Antes porém façamos uma rápida descrição das funções descritas anteriormente.

Núcleo ou Kernel

Parte central do Sistema Operacional. É quem controla os processos gerentes de recursos e transfere mensagens entre eles. É a camada do Sistema Operacional que se comunica com o nível de linguagem de máquina (veja início do curso).

O núcleo roda com interrupções inibidas (desabilitadas) portanto, deve limitar-se às funções essenciais para minimizar o tempo em que a máquina fica insensível as modificações do ambiente. Os demais gerentes do Sistema Operacional são processos interrompíveis, assim como os processos dos usuários. Todos os gerentes e o núcleo rodam em modo **SUPERVISOR**. As rotinas de tratamento de interrupção e SVC residem no núcleo.

O Sistema Operacional mantém a estrutura de dados que reflete “o estado” dos recursos do Sistema Operacional em um dado momento.

Esta estrutura é composta por:

- descritores de processos - *recursos lógicos*
- descritores de memória - *recursos físicos*
- descritores de arquivo - *recursos lógicos*
- descritores de periféricos - *recursos físicos*

Quando um gerente se comunica com outro gerente a comunicação passa pelo núcleo. Os descritores são implementados como listas encadeadas em memória.

Descritores de Processo

- identificação do processo;
- limites de memória do processo;
- registradores utilizados pelos processos;
- lista de arquivos abertos.

Descritores de Memória

- endereços das áreas de memória;
- tamanho de cada área;
- informação sobre proteção;
- contador de acesso.

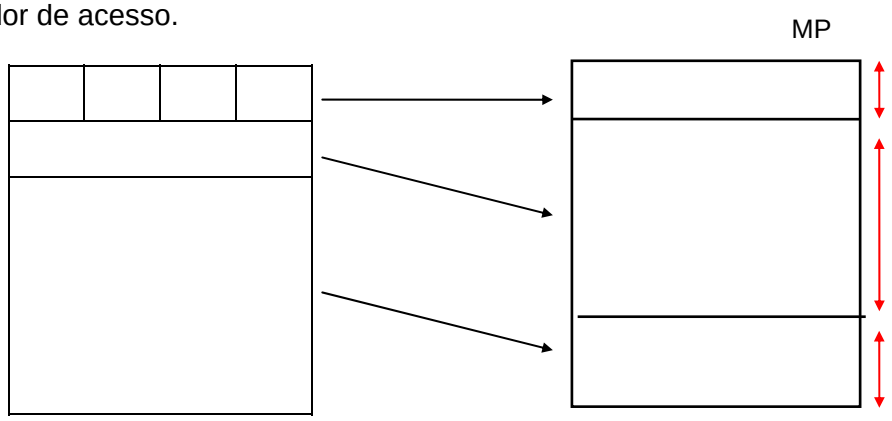


Fig 3.1 - Descritores de Memória

Descritores de Periférico

- código de identificação do periférico;
- endereço físico (endereço da porta do processador onde o controlador de processo esta ligado);
- modelo e tipo do periférico.
- lista de pedidos pendentes de periféricos

Descritores de Arquivos

- nome do arquivo;
- tipo de acesso permitido (leitura, escrita, leitura/escrita);
- periférico associado (endereço do descritor de periférico);
- identificação do volume (em caso de disco ou fita);
- tamanho do registro lógico e do registro físico;
- tipo de organização de dados;
- endereço na memória do registro lógico lido pelo controlador do periférico;
- endereço na memória da área de Buffering que contém o registro físico lido pelo controlador do periférico.

3.2 Proteção do Sistema

Nos SOs Multiprogramados, onde diversos usuários compartilham os mesmos recursos, deve existir uma preocupação por parte do sistema operacional a fim de garantir a integridade dos dados pertencentes a cada usuário.

Como vários programas ocupam a memória simultaneamente e cada usuário possui uma área onde dados e código são armazenados (code area, stack e data) o SO deve possuir mecanismos de proteção à memória de forma a preservar as informações contra acessos indevidos (acidentais ou propositais) de forma a torná-lo confiável.

Para isso todo sistema implementa algum tipo de proteção aos diversos recursos que são compartilhados no sistema, como memória, dispositivos de E/S e UCP.

O mecanismo de proteção a memória funciona de tal forma que quando o programa tenta acessar uma posição de memória fora de sua área endereçável, um erro do tipo violação de acesso ocorre e o programa é encerrado. O mecanismo para controle de acesso à memória varia em função do tipo de Gerência de Memória implementada pelo SO e será discutido mais adiante.

Um problema relacionado à proteção é quando um programa reserva um periférico para realizar alguma operação (Exemplo: na utilização de uma impressora nenhum outro processo deve interferir até que o processo a libere). O compartilhamento de dispositivos de E/S deve ser controlado de forma centralizada pelo SO.

O compartilhamento de arquivos em disco permite que dois ou mais usuários acessem o mesmo arquivo simultaneamente. Caso o acesso concorrente não seja controlado pelo SO, problema de inconsistência nas informações obtidas dos arquivos podem ocorrer. Em geral o SO disponibiliza rotinas para trancamento (lock) de arquivos e/ou registros de arquivos para permitir o acesso exclusivo ou compartilhado por diversos usuários. No UNIX temos a system call **flock** (file lock) que realiza esta tarefa.

Para evitar que um programa em loop aloque o processador por tempo indeterminado, a UCP possui um mecanismo denominado timer (relógio de HW) que gera interrupções periódicas, geralmente chamadas de relógio de tempo real. Estas interrupções provocam a execução de uma rotina crítica dos SOs, a rotina de tratamento de interrupções de relógio, que entre outras funções, tem a responsabilidade de suspender o processamento de qualquer processo que tenha estado utilizando a UCP por tempo superior a uma variável global do sistema operacional denominada time-slice. Este mecanismo garante o rodízio do processador pelos vários processos ativos na máquina.

O **relógio de tempo real** permite, assim, a figura conhecida como preempção de processos, que pode ser descrita como a "perda da vez de usar o processador em decorrência do surgimento de processo de maior prioridade, ao ocorrer uma interrupção de relógio e o esgotamento do time-slice, ou outra interrupção qualquer".

Para solucionar estes diversos problemas, o SO deve implementar mecanismos que controlem o acesso concorrente dos vários usuários aos diversos recursos do sistema. Toda vez que um usuário desejar utilizar um recurso, ele terá que solicitá-lo ao SO. O pedido é realizado através de chamadas a rotinas especiais denominadas **rotinas do sistema (system calls).** Estas rotinas permitem acessar recursos do SO, devendo, por isso, possuir um mecanismo de proteção para poderem ser executadas. O mecanismo de proteção, implementado na maioria dos SOs Multiprogramados é denominado **estado de execução**. O estado de execução é uma característica associada ao programa em execução, que determina se ele pode ou não executar outras instruções ou rotinas.

No **estado usuário**, um programa só pode executar instruções que não afetam diretamente outros programas, denominadas **instruções não-privilegiadas**. No **estado supervisor** (ou monitor ou Kernel), qualquer instrução pode ser executada e, conseqüentemente, todas as rotinas do sistema. As instruções que podem ser executadas por programas no estado supervisor são denominadas **instruções privilegiadas**.

O **estado de execução** de um processo é determinado por um conjunto de bits, localizado em um registrador especial da UCP que indica o estado corrente (PSW process status word) o qual o HW do sistema acessa para verificar o estado do processo, e se a instrução pode ou não ser executada. Quando um programa que está sendo processado no estado usuário executa uma system call o seu estado é alterado pela própria rotina do sistema que se encarrega, ao seu término, de restaurar o estado de execução anterior do processo. Caso um programa tente executar uma **instrução privilegiada**, sem estar no estado supervisor, uma interrupção é gerada e o programa é encerrado.

Tipicamente, o SO divide sua área de memória em duas partes: a parte de endereços mais baixos, reservada para o próprio SO, chamada de ***Espaço de Endereçamento do Sistema*** (**EES**); a parte dos endereços mais altos, reservada para os programas dos usuários, chamada de ***Espaço de Endereçamento do Usuário*** (**EEU**). Programas dos usuários só podem ter acesso ao ***Espaço de Endereçamento do Sistema*** (**EES**) via mecanismos de "chamada ao sistema" (***system call***), que é invocado sempre que um processo precisa ter acesso a algum serviço disponível.

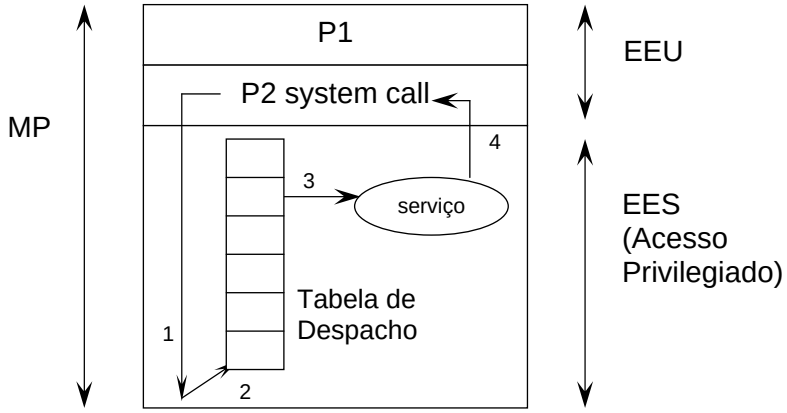


Fig 3.2 - Proteção de Sistema

Como exemplo, suponha que um programa deseje ler um registro em um arquivo localizado em disco. O programa, por si só, não pode especificar diretamente as instruções que acessam o bloco de dados que contém o registro. Isso acontece porque o disco é um recurso compartilhado por vários usuários, por isso sua utilização deverá ser realizada pelo SO. Essa implementação evita que um programa possa acessar qualquer arquivo em disco indiscriminadamente. Para o programa conseguir ler um registro, deve solicitar a operação ao SO através de uma system call. **Na execução da rotina o sistema verificará se o programa pode ou não acessar o arquivo. Em caso afirmativo, mudará o estado de execução do programa para supervisor e transfere o controle da execução para o Kernel (1)**. **O SO examina os parametros da chamada para determinar qual system call deverá ser executada (2). A system call é então realizada (3)**, no caso uma operação de leitura e, a seguir, **após o término a system call retorna o estado de execução para o estado usuário e retorna para a fun'ção chamadora (4)** para continuar o processamento.

As ***system calls*** podem ser divididas em 3 categorias principais:

(a) Controle de processos

- Terminar, abortar processo
- Carregar, executar processo
- Obter, atualizar atributos de processos
- Esperar ou sinalizar algum evento

(b) Manipulação de arquivos

- Criar, remover arquivos
- Ler, gravar e posicionar em arquivos
- Abrir, fechar arquivos
- obter, atualizar atributos de arquivos e periféricos
- requisitar, liberar dispositivo
- ler, gravar e posicionar em dispositivo
- obter, atualizar atributos de dispositivos

(c) Informações gerais

- obter, atualizar data e hora do sistema
- obter informações sobre contabilização de recursos utilizados pelos processos e usuários (ex: tempo UCP, elapsed time)

Os sistemas operacionais modernos oferecem uma larga coleção de programas (***system programs***) para resolver tarefas comuns de forma a proporcionar um ambiente mais conveniente para o usuário. Podemos dividir os programas do sistema nas seguintes categorias:

(a) Manipulação de arquivos - Modificação

- criação, cópia, remoção, renomeação
- impressão, visualização de arquivos e diretórios, edição de arquivos

(b) Informação de Status

- data, hora, nível de utilização de memória e disco, número de usuários
- quais usuários estão logados (conectados ao sistema)

(c) Suporte às linguagens de programação

- compiladores, assemblers, interpretadores

(d) Carregamento e execução de programas

- absolute loaders
- relocatable loaders
- linkage editors e overlay loader
- depuradores de código (debuggers)

(e) Programas de Aplicação

- formatadores de texto
- database systems
- pacotes de análise estatística, etc.

Conclusão:

O SO é um conjunto de programas direcionados por eventos (event-driven programs): se não há jobs para executar, nenhum dispositivo de E/S para atender e nenhum usuário para atender o SO permanecerá parado esperando até que algum evento ocorra para ser atendido. Como os eventos são normalmente sinalizados por interrupção (que é um mecanismo mais eficiente que o polling ou o busy wait) podemos concluir que o SO é interrupt-driven. Esta natureza do SO é que define a estrutura geral do SO. Quando uma interrupção (ou trap) ocorre, o HW transfere o controle para o SO. Primeiro o SO "salva o contexto" do processo ora em execução, determina qual o tipo de interrupção ocorreu (consultando vetor de interrupções do sistema) e chama a rotina de serviço de interrupção adequada. Dependendo do tipo de interrupção o processo anterior será restaurado para execução ou não. Genericamente podemos ter 3 tipos de **interrupção**:

- **System Call**
- **Interrupção de dispositivo de E/S**
- **Program Error**

O Fluxo de Controle do SO quando da ocorrência de uma interrupção é exemplificado abaixo:

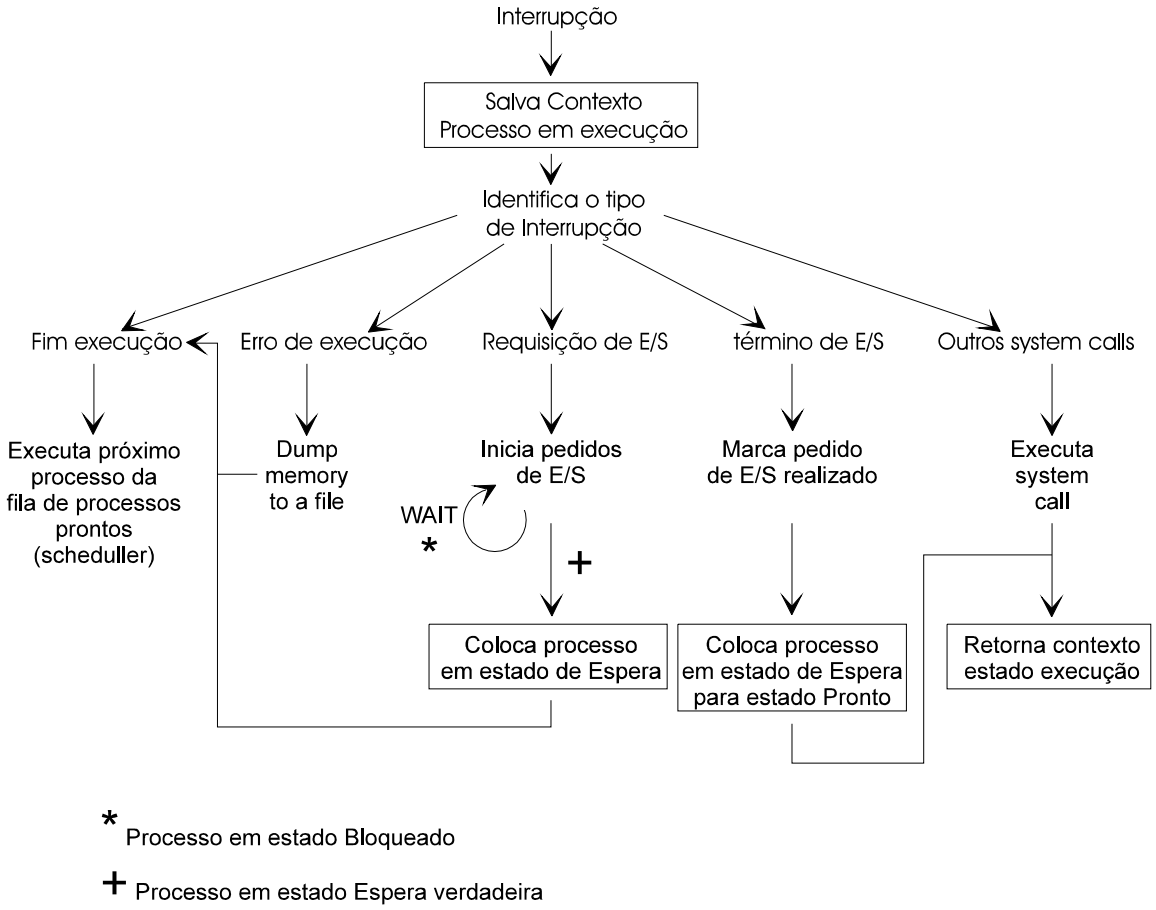


Fig 3.3 - Fluxo de Controle do SO para tratamento de interrupção

Para finalizar esta parte vamos entender como o SO gerencia o tratamento de interrupções de dispositivos de E/S. O dispositivo de E/S interromperá a UCP assim que terminar de atender um pedido de E/S (I/O request). Quando um pedido de E/S é iniciado o SO varre a tabela de status dos dispositivos (device status table) para verificar o status do dispositivo (ocupado, livre, não-disponível). No caso do dispositivo estar ocupado o SO adiciona o pedido na lista de pedidos pendentes do dispositivo e coloca o processo no estado "bloqueado".

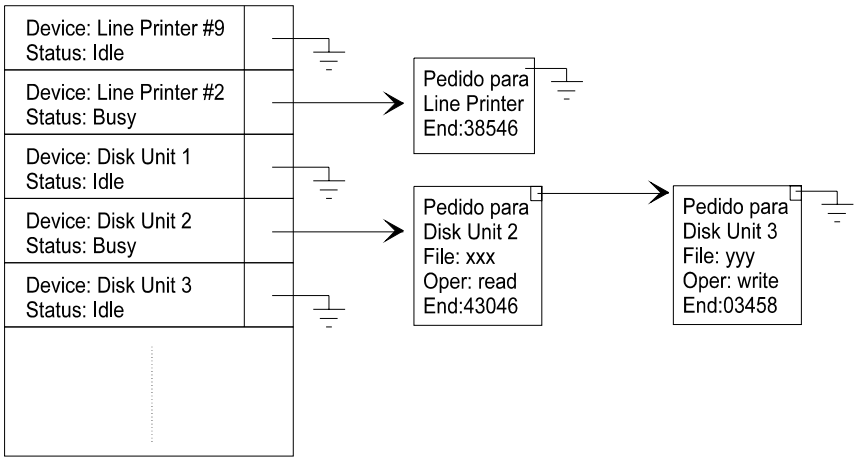


Fig 3.4 - Tabela de Status de dispositivo

Quando um dispositivo acaba a sua tarefa ele interrompe a UCP avisando o término do tratamento de E/S. O SO identifica qual dispositivo fez a interrupção, consulta a lista de pedidos pendentes e trata o próximo pedido mudando o estado de processo de Bloqueado → Espera.

Adicionalmente o processo que estava em estado de Espera aguardando o término da E/S é colocado na fila de processos prontos. Dependendo do mecanismo de E/S do dispositivo utilizar ou não **DMA** (direct memory access - capacidade de periférico endereçar diretamente o espaço utilizado pelo processo) será necessário carregar o contexto do processo que aguardará o término da E/S para copiar as informações do Buffer E/S para a área de dados do processo para depois então trocar o estado de Espera para Pronto.

3.3 Sistemas Monolíticos, Em Camadas e Micro-Núcleo

Nos SOs **monolíticos**, o código do SO reside no **EES**, que é protegido, e os programas dos usuários residem no **EEU**. A estrutura monolítica consiste de um conjunto de rotinas, organizadas na forma de um grande módulo objeto, que podem interagir livremente umas com as outras. A ausência de uma estrutura hierárquica no Kernel é uma característica deste sistema.

Sempre que um processo necessitar de um serviço, ele o requisita através de uma chamada ao sistema (*system call*). Para tratar essa chamada, a rotina de tratamento de interrupções do SO interpreta o pedido através de uma tabela de despacho (***Dispatch Table***). Esta tabela fornece o endereço para a rotina de serviço pedido e o SO passa o controle da execução para a mesma que, após o seu final, retorna o controle para o programa do usuário. O núcleo **Monolítico** é bastante popular e foi o primeiro modelo a ser implementado. O SO fornece todos os serviços que os programas dos usuários necessitam, incluindo sistemas de arquivos, serviços de diretórios, gerencia de processos e de memória bem como o mecanismo para tratamento das chamadas ao sistema. Normalmente, os serviços de rede e serviços remotos também estão integrados ao sistema.

O acesso ao núcleo do SO a partir dos programas dos usuários pode causar problemas, particularmente em sistemas onde os processos individuais podem comunicar-se uns com os outros. Um conflito entre processos, que concorrem pelo mesmo serviço, pode causar a queda da máquina hospedeira devido a ocorrência de **deadlocks**.

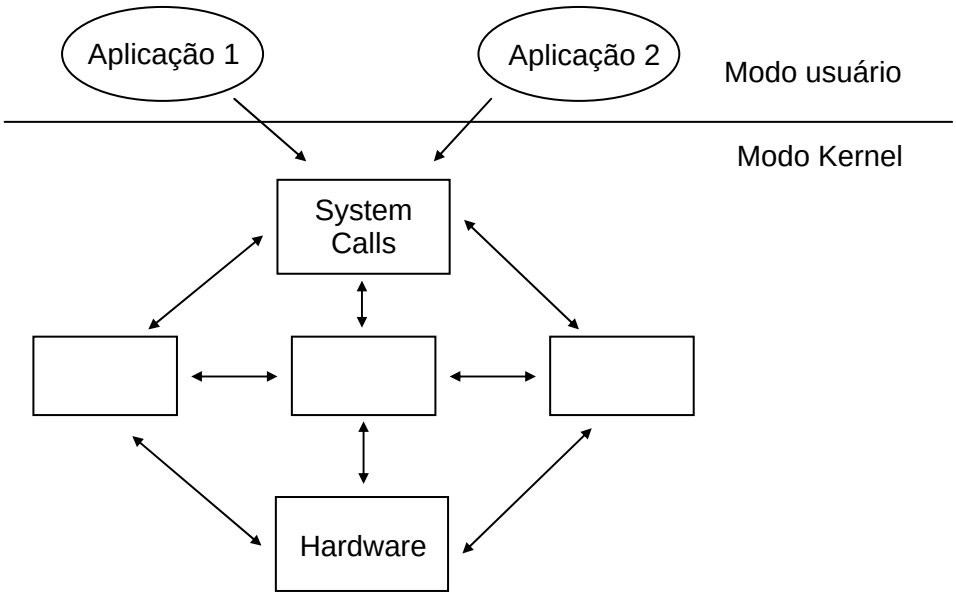
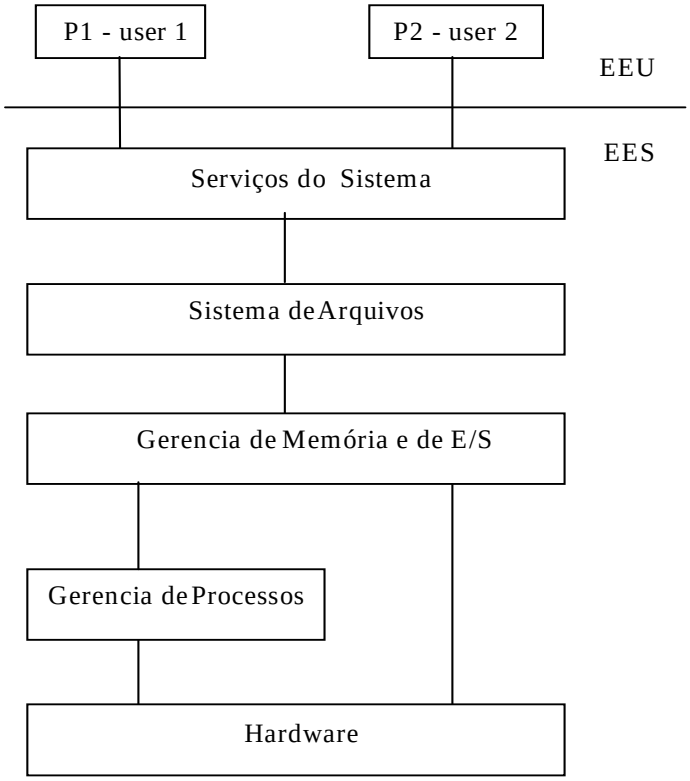


Fig 3.5 - Sistemas Monolíticos

Para resolver este problema, foram projetados os **SOs Em camadas**, que fornecem proteção aos endereços críticos de serviços pelo estabelecimento de uma hierarquia como a estabelecida na figura abaixo:



Nesta estrutura, os programas dos usuários fazem chamadas às camadas mais altas do núcleo, as quais por sua vez, passam adiante o pedido para o serviço apropriado. Um usuário não pode, desta forma, ter acesso a uma posição arbitrária de memória ou a uma interface de hardware. A vantagem deste sistema é isolar as funções do SO facilitando sua alteração e depuração, além de criar uma hierarquia de níveis de modos de acesso, protegendo as camadas mais internas.

Exemplos deste sistema são o ***Multics***, precursor do ***UNIX*** e o ***VMS*** da ***DEC***. Apesar de uma perda de desempenho do SO como um todo, devido a estrutura em camadas, garante-se um processamento seguro e robusto.

Fig 3.6 - Sistemas Em Camadas

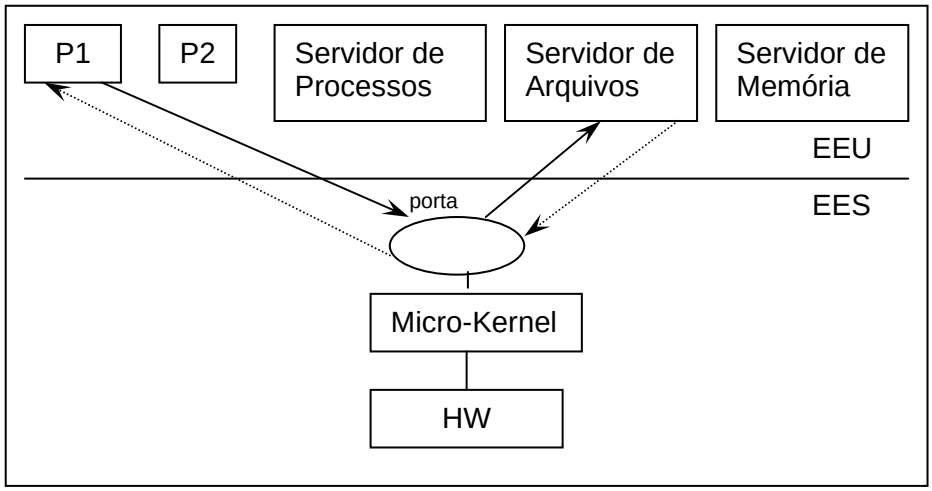


Fig 3.7 - Sistemas Micro-Kernel

A Estrutura baseada em **Micro-Núcleos** representa o que há de mais adequado para as plataformas de computação distribuída, devido à sua modularidade e rapidez de processamento. Sua principal função é fornecer uma interface para todo o hardware e gerenciar toda a comunicação entre os serviços que agora residem no **EEU**, e não mais no **EES** como nos outros casos.

Um processo cliente obtém um serviço pela troca de mensagens com processos servidores através de portas de comunicação (mailbox), mantidos no EES. Quando um processo cliente faz um pedido por um serviço, ele envia a mensagem para o Kernel, que a armazena em uma porta específica para aquele processo cliente. A seguir, o núcleo notifica o processo servidor requisitado para que o mesmo estabeleça uma comunicação com a porta do cliente, de modo que se possa atender ao pedido.

Os **Micro-Núcleos** disponibilizam uma quantidade mínima de serviços chamados serviços essenciais, que incluem, dentre outras, a comunicação entre processos; a gerência de memória básica; gerencia de processos de baixo nível e escalonamento; e entrada/saída de baixo nível, incluindo serviços para comunicação remota. Geralmente, não estão incluídos no Micro-Núcleo os chamados serviços não essenciais, como, por exemplo, os sistemas de arquivos, os serviços de diretórios, a gerência completa de processo, etc. Esses serviços não essenciais são implementados como processos em nível de EEU. Tal enfoque conduz ao projeto de um núcleo menor, mais confiável, que fornece uma maior facilidade para a adição, alteração e teste de novos serviços. Além disso, a instalação, implementação e a depuração de novos serviços não essenciais são muito mais simples nas arquiteturas em **Micro-Núcleos**, pois, para adicionar ou modificar um desses serviços não é necessário que se pare o sistema, ou que se reinicie o núcleo, como ocorre nas arquiteturas monolíticas.

Esta é a grande ***flexibilidade*** apresentada por estes sistemas. Além do mais, usuários insatisfeitos com serviços fornecidos podem implementar seus próprios serviços da maneira que considerarem mais apropriadas.

A única vantagem potencial dos sistemas monolíticos é em relação a performance, já que a execução de uma *system call* (que envolve o desvio para a área do sistema operacional e execução da rotina de serviço no *modo kernel*) é mais rápida que enviar mensagens para servidores remotos.

4 Processos e Threads

4.1 Introdução

O processo pode ser entendido como um programa em execução. Só que seu conceito é mais abrangente. Este conceito toma-se mais claro quando pensamos de que forma os sistemas multiprogramáveis (multitarefa) atendem os diversos usuários (tarefas) e mantém informações a respeito dos vários programas que estão sendo executados concorrentemente.

Como sabemos, um sistema multiprogramável simula um ambiente de monoprogramação para cada usuário, isto é, cada usuário do sistema tem a impressão de possuir o processador exclusivamente para ele. Nesses sistemas, o processador executa a tarefa de um usuário durante um intervalo de tempo (time-slice) e, no instante seguinte, está processando outra tarefa. A cada troca, é necessário que o sistema preserve todas as informações da tarefa que foi interrompida, para quando voltar a ser executada não lhe faltar nenhuma informação para a continuação do processamento. A estrutura responsável pela manutenção de todas as informações necessárias à execução de um programa, como conteúdo de registradores e espaço de memória, chama-se *processo.*

Neste capítulo, entenderemos melhor o que é um processo, seus estados, transições e tipos, além dos conceitos de subprocesso e thread.

4.2 Modelo de Processo

O conceito de *processo* pode ser definido como sendo o ambiente onde se executa um programa. Um mesmo programa pode produzir resultados diferentes, em função do Processo no qual ele é executado. Por exemplo, se um programa necessitar abrir cinco arquivos simultaneamente, e o processo onde será executado só permitir que se abram quatro, o programa será interrompido durante a sua execução.

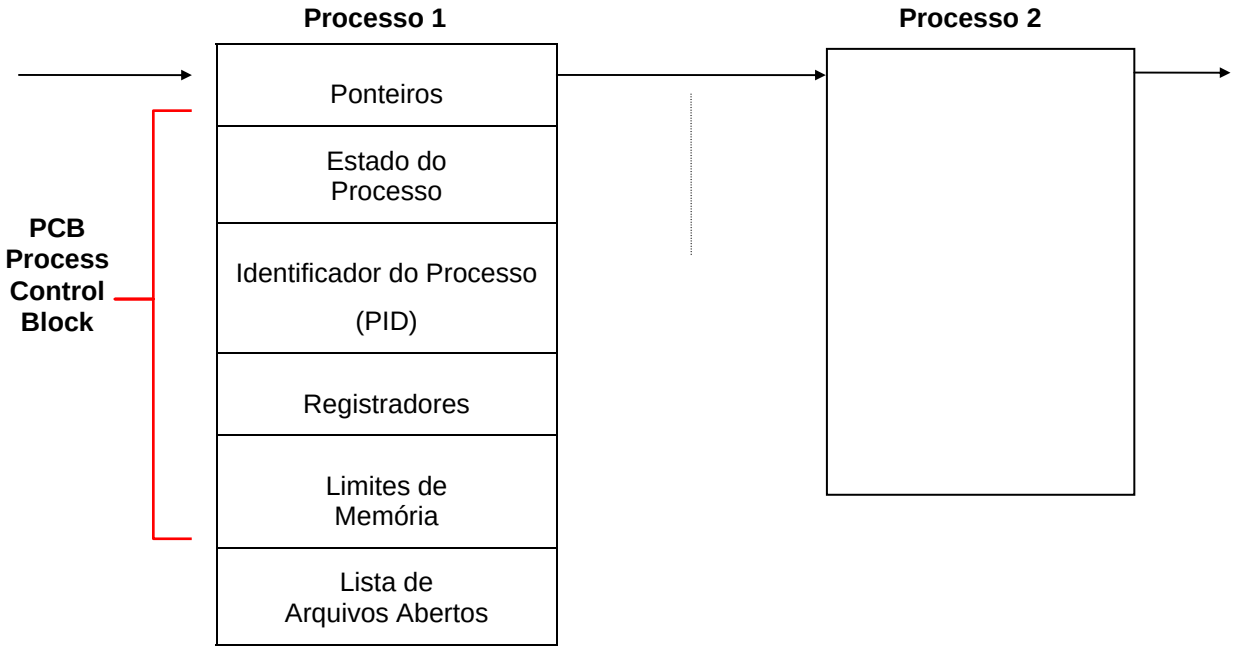


Fig 4.1 - Descritores de Processo

O sistema operacional materializa o processo através de uma estrutura chamada **bloco de controle do processo (Process Control Block PCB).** A partir do **PCB**, o sistema operacional mantém todas as informações sobre o processo, como sua identificação, prioridade, estado corrente, recursos alocados por ele e informações sobre o programa em execução (Fig. 4.1). O sistema operacional gerencia os processos através de system calls, que realizam operações como criação, eliminação, sincronização, suspensão de processos, dentre outras.

O processo pode ser dividido em três elementos básicos: **contexto de hardware**, **contexto de software** e **espaço de endereçamento**, que juntos mantêm todas as informações necessárias à execução do programa.

4.2.1 Contexto de Hardware

O **Contexto de Hardware** constitui-se, basicamente, do conteúdo de registradores: program counter (PC), stack pointer (SP) e bits de estado. Quando um processo está em execução, o seu contexto de hardware está armazenado nos registradores do processador. No momento em que o processo perde a utilização da UCP, o sistema salva suas informações no seu Contexto de Hardware.

O Contexto de Hardware é fundamental para a implementação dos sistemas de tempo compartilhado (time-sharing), onde os processos se revezam na utilização do processador, podendo ser interrompidos e, posteriormente, restaurados como se nada tivesse acontecido. A troca de um processo por outro na UCP, realizada pelo sistema operacional, é notificada a **mudança de contexto** (**context switching**). A mudança de contexto consiste em salvar o conteúdo dos registradores da UCP e carregá-los com os valores referentes ao do processo que esteja ganhando a utilização do processador. Essa operação resume-se, então, em substituir o contexto de hardware de um processo pelo de outro (Fig. 4.2).

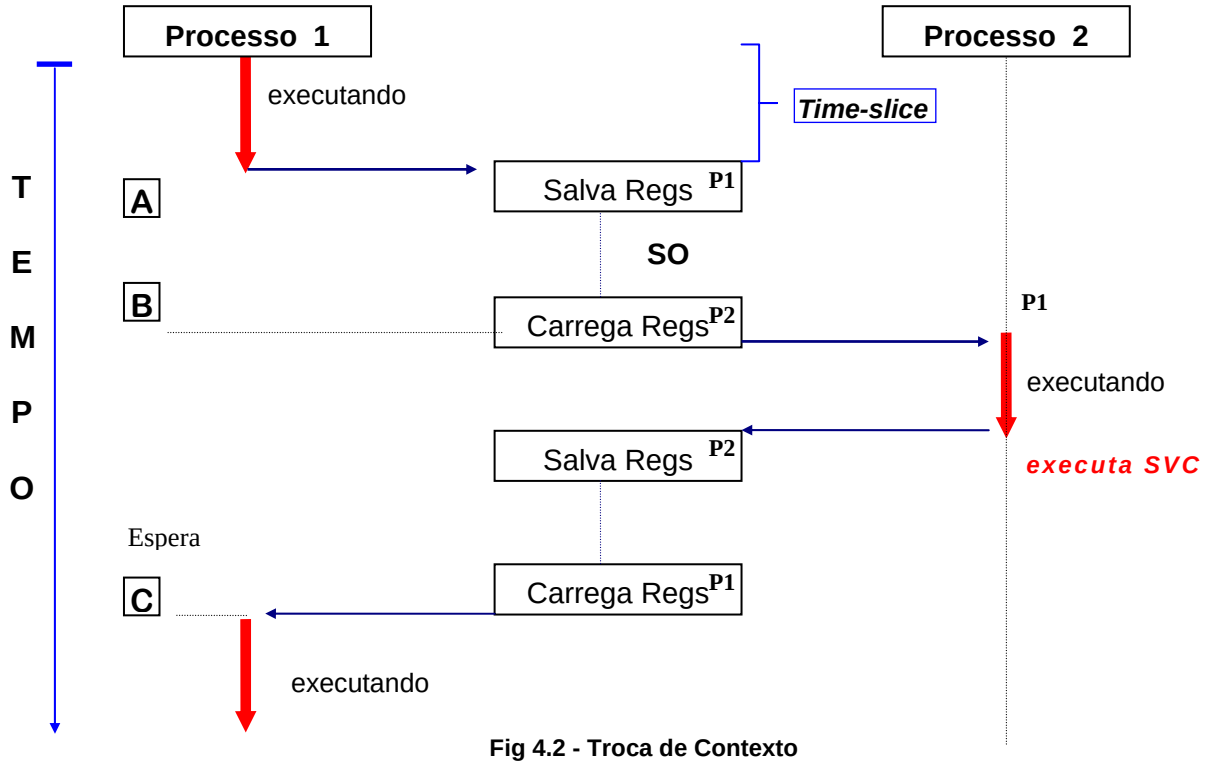


Fig 4.2 - Troca de Contexto

A utilização concorrente da UCP deve ser implementada de maneira que, quando um programa perde o uso do processador e depois retorna para continuar o seu processamento, seu estado (valor das variáveis e dos registradores da máquina) deve ser idêntico ao do momento em que foi interrompido. O programa deverá continuar sua execução exatamente na instrução seguinte àquela em que havia parado, aparentando ao usuário que nada aconteceu. Este mecanismo de salvamento do estado do processo chama-se de "Salvamento de Contexto".

4.2.2 Contexto de Software

O **Contexto de Software** especifica características do processo que vão influir na execução de um programa. Como, por exemplo, o número máximo de arquivos abertos simultaneamente ou o tamanho do buffer para operações de *E/S.* Essas características são determinadas no momento da criação do processo, podendo algumas ser alteradas durante sua existência.

O contexto de software define basicamente três grupos de informações sobre um processo: sua identificação, suas quotas e seus privilégios.

1. **Identificação -** Cada processo criado pelo sistema recebe uma identificação única (**PID - process identification**), representada por um número. Alguns sistemas, além do PID, identificam o processo através de um nome. Através do PID, o sistema operacional e outros processos podem fazer referência a um determinado processo e, por exemplo, alterar uma de suas características. O processo também possui a identificação do usuário ou processo que o criou (owner). Cada usuário possui uma identificação única no sistema (**UID - user identification**), atribuída ao processo no momento de sua criação. A UID permite implementar um modelo de segurança, onde apenas os objetos (processos, arquivos, áreas de memória etc.) que possuem a mesma UID do usuário (processo) podem ser acessados.

2. **Quotas -** As quotas são os limites de cada recurso do sistema que um processo pode alocar. Caso uma quota seja insuficiente, o processo poderá ser executado lentamente ou mesmo não ser executado. Alguns exemplos de quotas que aparecem na maioria dos sistemas operacionais são:

    - número máximo de arquivos abertos simultaneamente:
    - tamanho máximo de memória que o processo pode alocar:
    - número máximo de operações de *E/S* pendentes:
    - tamanho máximo do buffer para operações de E/S:
    - número máximo de processos e subprocessos que podem ser criados.

3. **Privilégios -** Os privilégios definem o que o processo pode ou não fazer em relação ao sistema e aos outros processos. Existem privilégios associados à segurança que permitem a um usuário eliminar processos de outros usuários e ter acesso a arquivos que não lhe pertencem. Existem outros privilégios associados à operação e à gerência do sistema, dentre outros.

4.2.3 Espaço de Endereçamento

O espaço de endereçamento é a área de memória do processo onde o programa será executado, além do espaço para os dados utilizados por ele. Cada processo possui seu próprio espaço de endereçamento, que deve ser protegido do acesso dos demais processos.

No capítulo Gerência de Memória estudaremos diversos mecanismos de implementação e gerência do espaço de endereçamento, principalmente a técnica de memória virtual.

Atualmente o modelo mais geral para o processo executando na memória principal é o mostrado a seguir na figura 4.3.

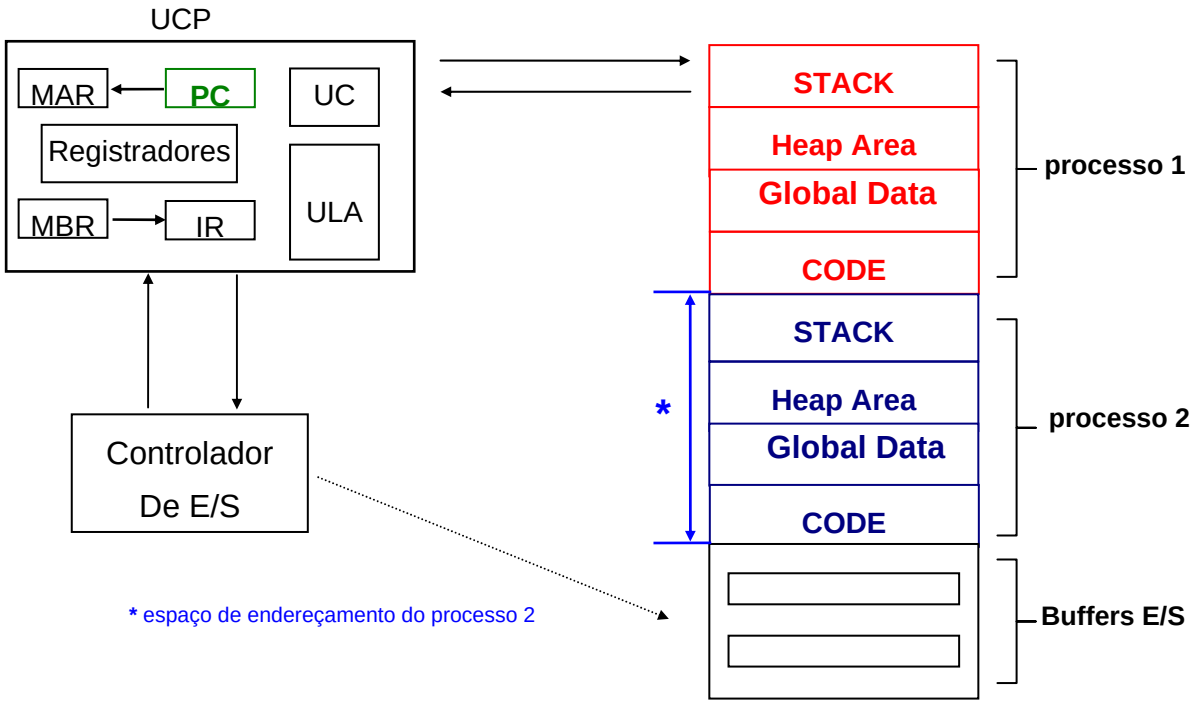
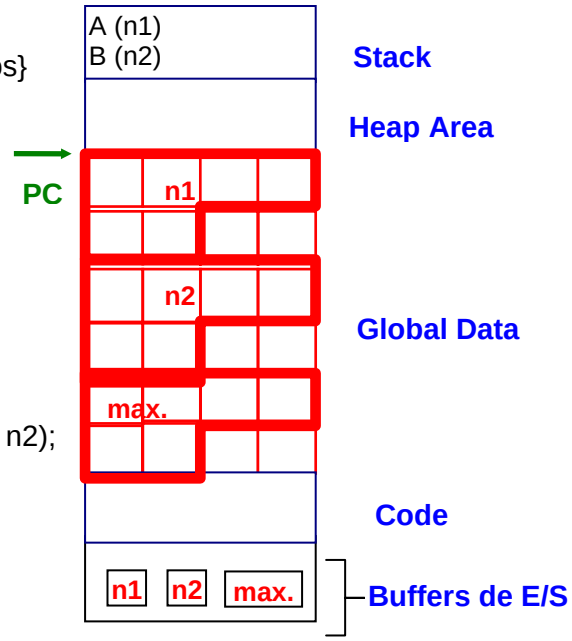


Fig 4.3 - Processo e seu Espaço de Endereçamento

Para exemplificar a função de cada uma dessas áreas vejamos como é feita a execução de um programa que irá imprimir o maior de dois números informados pelo usuário

```
Programa MAIOR; {Imprimir o maior de dois números}
VAR {global}
    n1, n2, max: integer;
Function maior (A, B: integer): integer;
BEGIN
    IF A > B then
        maior:= A;
    ELSE
        maior:= B;
END;
BEGIN {pgm principal}
    READLN (n1); READLN (n2);    max:= maior (n1, n2);
    WRITELN ('maximo entre', n1, n2, 'é', max);
END
```



4.3 Estados de um Processo

Um processo, em um sistema multiprogramável, não é executado todo o tempo pelo processador. Durante a sua existência, ele passa por uma série de estados. Basicamente, existem três estados em que um processo pode se encontrar no sistema:

1. **Execução (running) -** Um processo é dito no **estado de Execução** quando está sendo processado pela UCP. Em sistemas com apenas um processador, somente um processo pode estar sendo executado num dado instante de tempo. Os processos se revezam na utilização do processador segundo uma política estabelecida pelo sistema operacional. *Já* em sistemas com múltiplos processadores vários processos podem estar sendo executado ao mesmo tempo, dependendo do número de processadores. Existe também a possibilidade de um mesmo processo ser executado por mais de um processador (processamento paralelo).

2. **Pronto (ready) -** Um processo está no **estado de Pronto** quando apenas aguarda uma oportunidade para executar, ou seja, espera que o sistema operacional aloque a UCP para sua execução. O sistema operacional é responsável por determinar a ordem pela qual os processos em estado de pronto devem ganhar a UCP. Normalmente existem vários processos no sistema no estado de pronto.

3. **Espera (wait) -** Um processo está no **estado de Espera** quando aguarda algum evento externo ou por algum recurso para poder prosseguir seu processamento. Como exemplo podemos citar o término de uma operação de entrada/saída ou a espera de uma determinada data e/ou hora para poder continuar sua execução.

Em alguns sistemas, o estado de espera pode possuir uma subdivisão, em função do tipo de evento que o processo aguarda. Quando um processo espera por um recurso do sistema que não se encontra disponível, é dito que o processo está no **estado de bloqueado** (blocked). A diferença básica entre o estado de bloqueado e o de espera é que um processo em estado de bloqueado espera ser autorizado a utilizar um recurso, enquanto o processo em estado de espera aguarda pela conclusão de uma operação em um recurso que já foi garantido.

Um processo em estado de pronto ou de espera pode não se encontrar, momentaneamente na memória principal, ou seja, pode estar armazenado em memória secundária. Uma técnica denominada **Swapping** pode retirar processos da memória principal quando não existe espaço suficiente para todos os processos. No capítulo Gerência de Memória, este tópico será abordado com maior detalhamento.

O Sistema Operacional gerencia os processos através de listas encadeadas, onde cada PCB tem um ponteiro para seu sucessor. Como podem existir vários processos nos estados de pronto ou de espera, o sistema implementa listas, onde os processos aguardam seu processamento (**listas de processos no estado de pronto**) ou esperam por algum evento (**listas de processos no estado de espera**).

4.4 Diagrama de Transições de Estado

Um processo muda de estado diversas vezes, durante seu processamento. Em função de eventos originados por ele próprio *(eventos voluntários)* ou pelo sistema operacional *(eventos involuntários).*

Como eventos originados pelo próprio processo, podemos exemplificar uma operação de entrada/saída (SVC) ou qualquer chamada a uma rotina do sistema, requisitando algum tipo de serviço. Eventos que se originem do sistema operacional têm a intenção de permitir maior compartilhamento dos recursos do sistema. Por exemplo, se um programa está em looping, o sistema deve intervir para que o processador não fique dedicado eternamente ao processo onde o programa está sendo executado.

Basicamente, existem cinco mudanças de estado que podem ocorrer a um processo:

1. **Pronto - Execução :** Quando um processo é criado, o sistema o coloca em uma lista de processos no estado de pronto, onde aguarda uma oportunidade para ser executado (Fig. 4.4a). Cada sistema operacional tem seus próprios critérios e algoritmos para a escolha da ordem em que os processos serão executados (escalonamento ou schedulling). No capítulo Gerência do Processador, esses critérios e seus algoritmos serão analisados profundamente.

2. **Execução - Espera :** Um processo em execução passa para o estado de espera por eventos gerados pelo próprio processo, como, por exemplo, uma operação de entrada/saída (Fig. 4.4b). Nesse caso, o processo ficará neste estado esperando pela conclusão do evento solicitado.

3. **Espera - Pronto :** Um processo no estado de espera passa para o estado de pronto quando a operação solicitada é atendida ou o recurso esperado é concedido. Um processo no estado de espera sempre terá de passar pelo estado de pronto antes de poder ser novamente selecionado para execução. Não existe a mudança do estado de espera para o estado de execução diretamente (Fig. 4. 4c).

4. **Execução - Pronto :** Um processo em execução passa para o estado de pronto por eventos gerados pelo sistema. Como por exemplo, o fim da fatia de tempo que o processo possui para sua execução (Fig. 4.4d). Nesse caso, o processo volta para a fila de processos prontos onde aguarda por uma nova oportunidade para continuar seu processamento.

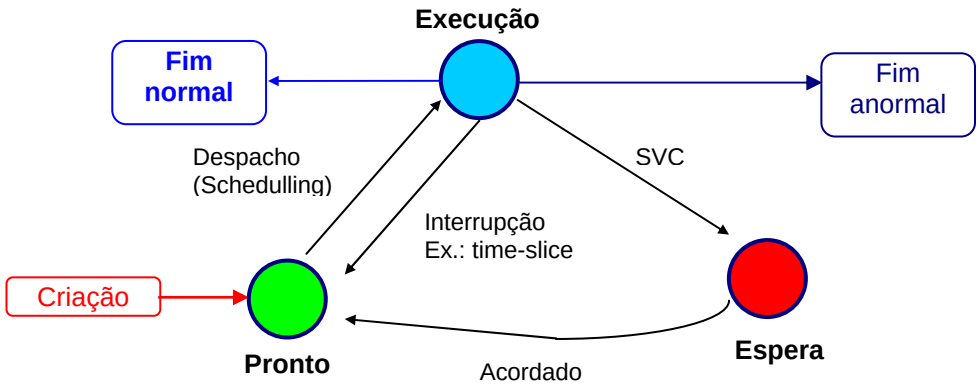


Fig 4.4 - Transições de Estado de um Processo

4.5 Subprocesso e Thread

Um processo pode criar outros processos de maneira hierárquica. Quando um processo (**processo pai***)* cria um outro, chamamos o processo criado de **subprocesso ou processo filho**. O subprocesso, por sua vez, pode criar outros subprocessos (Fig. 6.5). Como conseqüência dessa estrutura, caso um processo deixe de existir, os subprocessos subordinados são eliminados.

A utilização de subprocessos permite dividir uma aplicação em partes que podem trabalhar de forma concorrente. Por exemplo, suponha que um processo seja responsável pelo acesso a um banco de dados e existam vários usuários solicitando consultas sobre esta base. Caso um usuário solicite um relatório impresso de todos os registros, os demais usuários terão de aguardar até que a operação termine.

Com o uso de subprocessos, cada solicitação implicaria a criação de um novo processo para atendê-la, aumentando o **throughput** da aplicação e, conseqüentemente, melhorando seu desempenho.

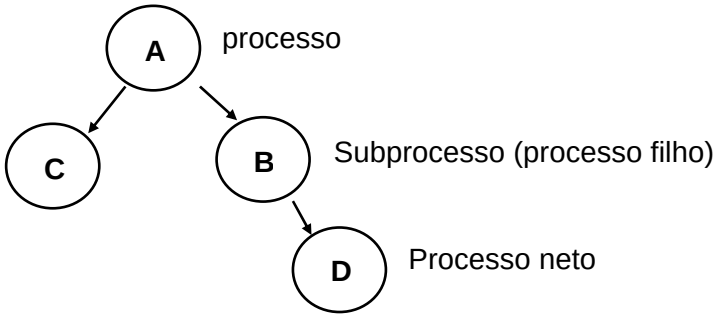
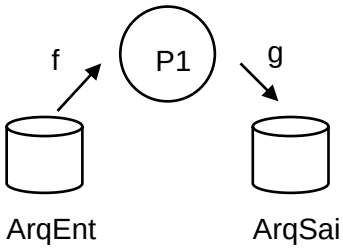


Fig 4.5 - Estrutura de Processo e seus Subprocessos

Subprocessos são criados, em geral, para auxiliar a computação, conforme mostra o exemplo a seguir do problema **produtor x consumidor.**

Programa de Cópia de Arquivos

```
Assign(f, ´ArqEnt´);
Assign(g, ´ArqSai´);
Reset (f, ArqEnt);
Rewrite (g, ArqSai);
 Read (f, reg);
While not eof(f) do
    Begin
        Write (g, reg);
        Read (f, reg);
    End/
```



Será que podemos melhorar este programa ?

Uma solução possível seria a criação de dois processos, um processo chamado **Produtor** que se encarregará de ler o arquivo de entrada e carregar um **Buffer** intermediário e um segundo subprocesso chamado **Consumidor** que irá ler as informações do **Buffer** e gravará os dados no arquivo de saída. A função do **Buffer** é a de prover um meio de armazenamento para os dois processos, de tal forma que se o **Buffer** for infinito ambos nunca ficarão bloqueados (Explique o porquê).

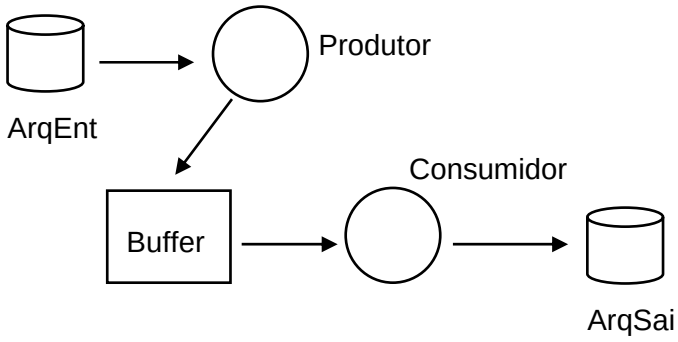


Fig 4.6 - Problema Produtor x Consumidor

Neste caso conseguiremos melhorar a performance de execução do algoritmo do programa de cópia se soubermos como criar subprocessos e fazer a comunicação entre eles.

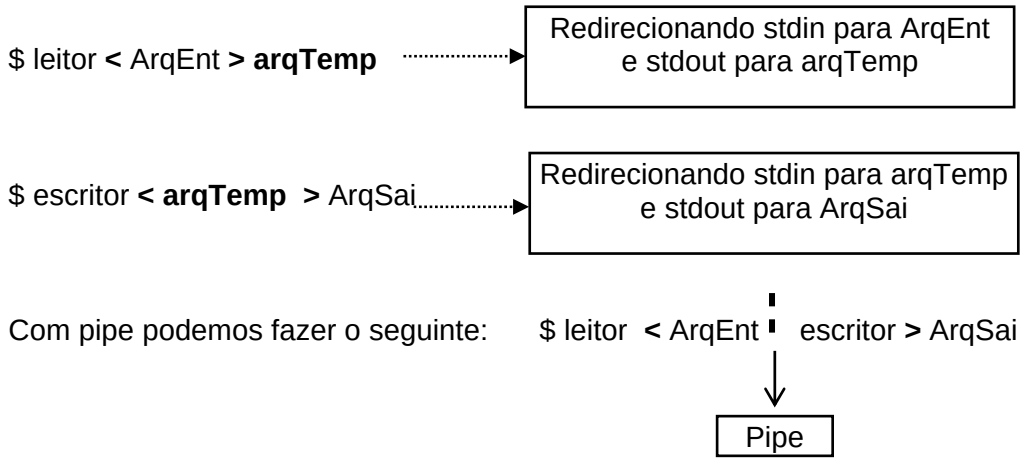
No caso do SO **Unix** existe o mecanismo de **Pipe** que permite a execução de dois processos de forma sincronizada. Sabemos que no Unix todo processo tem previamente abertos três arquivos, que são: **stdin** – entrada padrão (normalmente associada ao teclado); **stdout** – saída padrão (normalmente associada ao terminal); **stderr** – saída padrão de erro (normalmente associada também ao terminal).

O uso dos comandos de redirecionamento de stdin (**<**), stdout (**>**), stderr (**2>**), permitem que programas que leiam de stdin e gravem em stdout possam ter as suas entradas e/ou saídas redirecionadas para a leitura e gravação de arquivos quaisquer.

O mecanismo de **Pipe** é implementado como um arquivo FIFO em memória que é compartilhado pelos dois processos do **Pipe**. Portanto para permitir a sincronização todas as system calls de read e/ou write no pipe devem ser *bloqueantes.* **(para maiores detalhes veja Apostila de Unix do mesmo autor).**

Se os códigos dos programas forem escritos utilizando-se **stdin** e **stdout** como abaixo, o mecanismo de **Pipe** poderia ser utilizado para permitir a execução do programa concorrente. Vejamos o exemplo.

Processo Produtor	***Processo Consumidor***
Reset (f, ArqEnt);	*Rewrite (g, ArqSai);*
Read (f, reg);	*Read (Stdin, reg);*
While not eof (f)	*While not eof (Stdin)*
Begin	*Begin*
*Write (**stdout**, reg);*	*Write (g, reg);*
Read (f, reg);	*Read (**stdin**, reg);*
End;	*End;*



O uso de subprocessos no desenvolvimento de aplicações concorrentes demanda consumo de diversos recursos do sistema. Sempre que um novo processo é criado (no Unix isto é feito pela system call **fork**), o SO deve alocar recursos (contexto de HW, contexto de SW e espaço de endereçamento) para cada processo além de consumir tempo de UCP neste trabalho. No caso de término do processo, o sistema desperdiça tempo para desalocar recursos previamente alocados.

Na tentativa de diminuir o tempo gasto na criação/eliminação de processos, bem como economizar recursos do sistema como um todo, foi introduzido o conceito de **thread** (ou processo leve ou linha de controle ou linha de execução). Em um SO com Kernel com capacidade para criar múltiplas threads (**multithreaded Kernel**) não é necessária a criação de vários processos para se implementar aplicações concorrentes. Em um **SO Multithread** cada processo pode responder a várias solicitações concorrentes.

Na Fig 4.7a existem quatro processos, cada um com seu próprio contexto de HW, contexto de SW e espaço de endereçamento. Enquanto que na 4.7b existe um único processo com três threads de execução, cada uma com seu próprio contexto de HW (com diferentes valores para o PC - program counter) e contexto de SW.

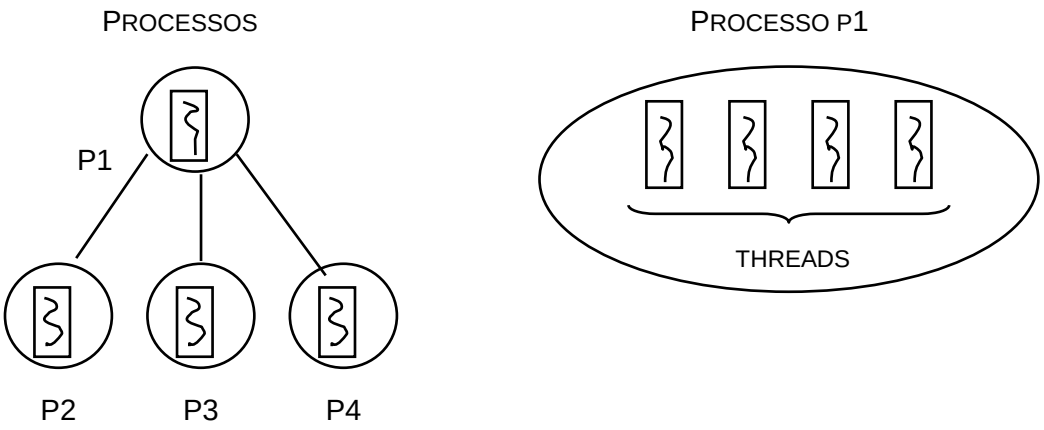


Fig 4.7 - Relação Processo x Subprocessos e Processo x Threads

Threads compartilham o processador da mesma maneira que processos. Por exemplo, enquanto uma thread espera por uma operação de E/S, outra thread pode estar executando. Cada thread possui seu próprio conjunto de registradores (contexto de HW), e contexto de SW, porém compartilha o mesmo espaço de endereçamento com as demais threads do processo.

O **mecanismo de runtime** é responsável pelo despacho para execução da thread de maior prioridade, ou da thread que estava esperando o fim de alguma operação de E/S. Dessa forma dizemos que o controle de execução de threads é feito de forma cooperativa.

Quando uma thread está sendo executada o contexto de HW da respectiva thread é carregado no processador. No momento em que uma thread perde (fim de time-slice) ou libera (yield) a UCP, o SO salva informações. Threads passam pelos mesmos estados que passam os processos.

A grande diferença entre subprocesso e thread é em relação ao espaço de endereçamento. Enquanto subprocessos possuem, cada um, espaços independentes e protegidos, threads compartilham o mesmo espaço de endereçamento do processo, sem nenhuma proteção, permitindo assim que uma thread possa alterar dados de outra thread. Apesar disto, threads são desenvolvidas para trabalhar de forma cooperativa na realização de uma computação.

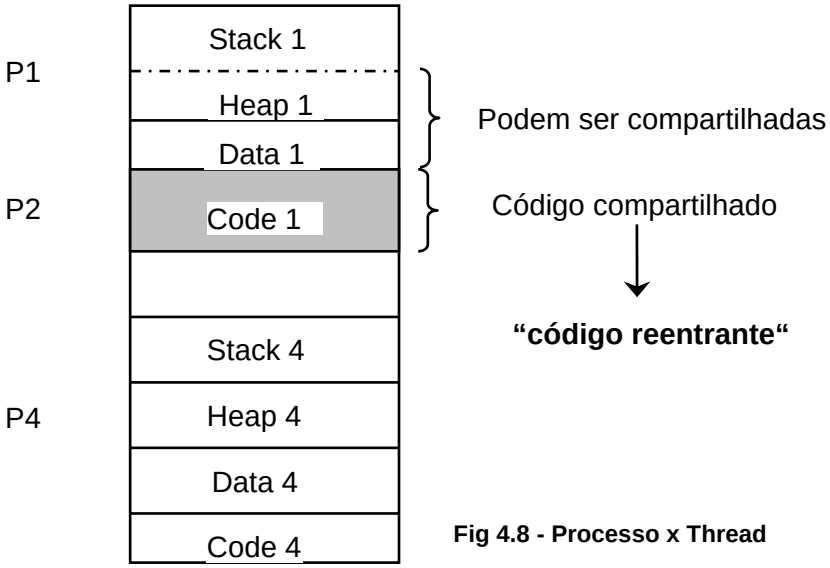


Fig 4.8 - Processo x Thread

5 Gerência do Processador

5.1 Introdução

Conforme já vimos anteriormente durante a execução de um processo no computador este passa por diversos estados, a saber:

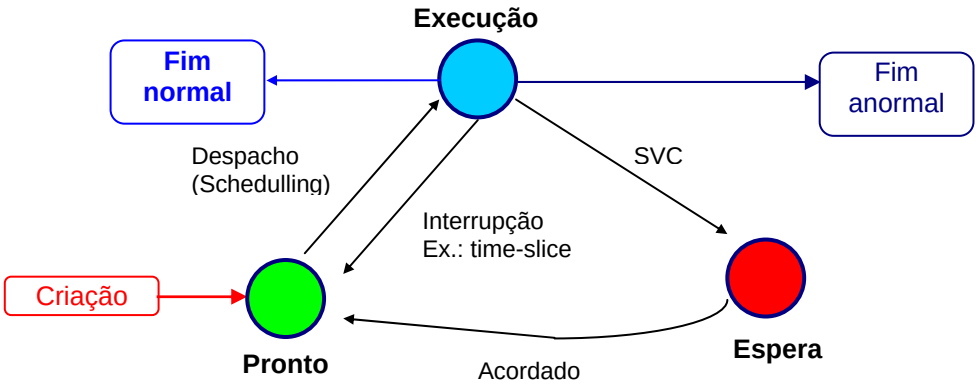


Fig 5.1 - Transições de Estado de um Processo

O conceito básico que gerou a implementação de sistemas multiprogramáveis foi a necessidade da UCP ser compartilhada entre os diversos processos.

Para isso tornou-se necessário a adoção de um critério para determinar qual a ordem na escolha dos processos para execução dentre os vários que concorrem pela utilização do processador. A este critério denominamos Escalonamento (Schedulling).

5.2 Escalonamento – Schedulling

Os principais objetivos do escalonamento são:

- manter a UCP ocupada a maior parte do tempo;
- balancear a utilização da UCP entre os diversos processos;
- maximizar o throughput;
- oferecer tempos de respostas aceitáveis para usuários interativos.

Para atender a estes objetivos, muitas vezes conflitantes, os Sistemas Operacionais devem levar em consideração as características dos processos, ou seja, se um processo é do tipo batch (não tem nenhuma interação com o usuário), interativo, UCP-Bound ou I\O-Bound. Sistemas Operacionais de tempo real ou de tempo compartilhado também são aspectos fundamentais para a implementação de uma política adequada de escalonamento.

5.2.1 Critérios de Escalonamento

A principal função do escalonamento é decidir qual dos processos prontos para execução deve ser alocado a UCP. Cada SO necessita de um algoritmo de escalonamento adequado a seu tipo de processamento. Os principais critérios que o escalonamento tenta otimizar são:

- **Utilização da UCP:** deseja-se que a CPU fique a maior parte do tempo ocupada, sendo dividida de forma imparcial entre os processos. Uma faixa de utilização de 90% indica uma um sistema bastante ocupado, próximo de sua capacidade ideal.

- **Throughput:** representa o número de processos (tarefas) executados em um determinado intervalo de tempo. Quanto maior o **Throughput,** maior o número de tarefas executadas em função do tempo.

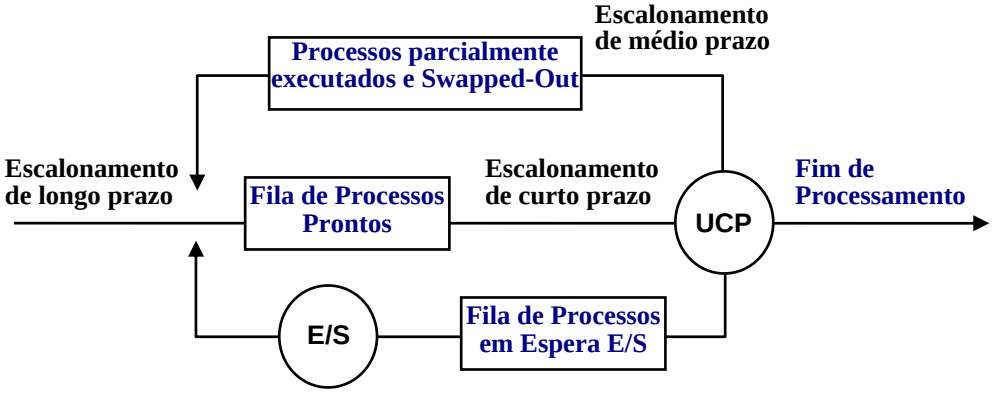
- **Tempo de turnaround:** é o tempo que um processo leva desde sua admissão até o seu término, levando-se em consideração o tempo de espera para alocação de memória, espera na fila de processos prontos, processamento na UCP e operações de E/S.

- **Tempo de resposta:** em sistemas interativos, este tempo é o tempo decorrido do momento da submissão do pedido até a primeira resposta produzida. O tempo de resposta não é o tempo utilizado no processamento total de uma tarefa, e sim o tempo decorrido até que uma resposta seja apresentada.

De uma maneira geral, qualquer algoritmo de escalonamento busca otimizar a **utilização da UCP** e o **Throughput**, enquanto tenta diminuir os **tempos de turnaround** e **de resposta.** O algoritmo de escalonamento não é o único responsável pelo tempo de execução de um processo. Outros fatores, como o **tempo de processamento (tempo de UCP)** e de **espera em operações de E/S**. devem ser considerados no tempo total da execução (**tempo de parede** ou **elapsed time** ou **wall clock time**). O escalonamento somente afeta o tempo de espera de processos na fila de pronto.

Os principais tipos de escalonamento da UCP são o **Escalonamento de Longo Prazo** o de **Médio Prazo** e o **de Curto Prazo**. O primeiro determina quais os jobs serão admitidos pelo SO para processamento. O de **Curto Prazo** seleciona o próximo processo a executar, dentre todos os processos da fila de processos prontos que estejam residentes em memória. O de **Médio Prazo**, mais comum em sistemas com **Memória Virtual** (capítulo 6), tem objetivo de liberar espaço na memória, removendo processos que estejam esperando algum evento externo (por exemplo, o fim de uma operação de E/S) para area de **Swap**, para permitir que novos processos sejam executados.

A diferença entre o escalonamento de **Longo Prazo** e o de **Curto Prazo** está na freqüência de execução de cada um. O de **Curto Prazo** é bem mais frequente que o de **Longo Prazo** pois ele deve executar sempre que um ciclo de execução do processo na UCP tenha terminado, ao passo que o de **Longo Prazo** tem o objetivo de garantir uma entrada contínua de novos processos para execução de forma a garantir um **Throughput** elevado. Em sistemas de **Tempo Compartilhado** o escalonamento de **Longo Prazo** não existe porque todo o processo que necessitar ser executado é imediatamente colocado para execução, já em sistemas **Multiprogramáveis Batch** cabe ao **Monitor** carregar os processos para execução na UCP a partir das unidades de fita ou disco do sistema conforme vimos no capítulo 2.8.



5.3 Escalonamentos Não Preemptivos

Nos primeiros sistemas operacionais multiprogramáveis, onde predominava tipicamente o processamento batch, o escalonamento implementado era do tipo **Não Preemptivo**. Neste tipo de escalonamento, quando um processo ganha o direito de utilizar a UCP nenhum outro processo pode lhe tirar este recurso.

5.3.1 Escalonamento Circular Simples - FIFO

Todos os processos começam a executar segundo a ordem que são chamados para execução. Quando um processo ganha o processador, ele utilizará o processador até o seu final sem ser interrompido.

No caso de ser executada uma SVC, o processo, após ter sido atendida a SVC, voltará para o final da fila de processos prontos.

Problemas do escalonamento FIFO é a impossibilidade de se prever quando um processo terá sua execução iniciada, já que isso varia em função do tempo de execução dos processos que se encontram na sua frente.

Observe que a UCP não é compartilhada de forma igual entre os processos em execução, portanto processos UCP-Bound de menor importância prejudicam processos IO-Bound mais prioritários.

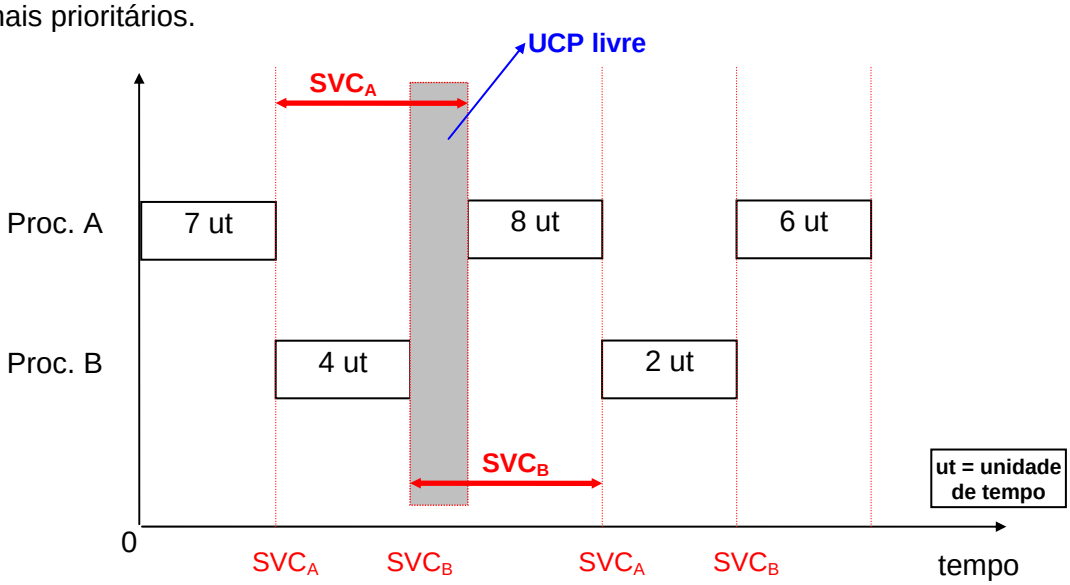


Fig 5.2 - Escalonamento FIFO

5.3.2 Escalonamento Shortest Job First - SJF

Neste escalonamento cada processo tem associado o seu tempo de execução. Desta forma quando a UCP está livre o processo em estado de pronto que tiver menor tempo de execução será selecionado para execução.

O escalonamento SJF favorece os processos que executam programas menores, além de reduzir o tempo médio de espera (na fila de processos prontos) em relação ao escalonamento FIFO. A dificuldade é determinar, exatamente, quanto tempo de UCP cada processo necessita para terminar seu processamento.

Tanto o SJF quanto o FIFO não são algoritmos de escalonamento aplicados a sistemas de tempo compartilhado, onde um tempo de resposta razoável deve ser garantido ao usuário interativo.

5.3.3 Escalonamento Cooperativo

No escalonamento cooperativo alguma política não-preemptiva deve ser adotada. A partir do momento que um processo está em execução, este voluntariamente libera o processador, retornando para a fila de pronto.

Este tipo de escalonamento permite a implementação de sistemas multiprogramáveis com uma melhor distribuição do uso do processador entre os processos. Sua principal característica está no fato de a liberação do processador ser uma tarefa realizada exclusivamente pelo processo em execução, que de uma maneira cooperativa libera a UCP para um outro processo.

No escalonamento cooperativo, não existe nenhuma intervenção do sistema operacional na execução do processo. Isto pode ocasionar sérios problemas na medida em que um programa pode não liberar o processador ou um programa mal escrito pode entrar em looping, monopolizando desta forma a UCP.

Um exemplo deste tipo de escalonamento pode ser encontrado nos sistemas Windows 3.1 e Windows 3.11 (ambos da Microsoft), sendo conhecido como **Multitarefa Cooperativa**. Nestes sistemas, as aplicações verificam uma fila de mensagens periodicamente para determinar se existem outras aplicações que desejam usar a UCP. Caso uma aplicação não verifique a fila de mensagens, as outras tarefas não terão chance de ser executadas até o término do programa.

5.4 Escalonamentos Preemptivos

Um algoritmo de escalonamento é dito preemptivo quando o sistema pode interromper um processo em execução, para que outro utilize o processador. Em sistemas que não implementam **preempção** um processo pode utilizar o processador enquanto for necessário.

O **escalonamento preemptivo** permite que o sistema dê atenção imediata a processos mais prioritários, como no caso de sistemas de tempo real, além de proporcionar melhores tempos de resposta em sistemas de tempo compartilhado. Outro benefício decorrente deste tipo de escalonamento é o compartilhamento do processador de uma maneira mais uniforme entre os processos.

A troca de um processo por outro na UCP (mudança de contexto), causada pela preempção, gera um overhead ao sistema. Para isto não se tomar crítico, o sistema deve estabelecer corretamente os critérios de preempção. Veremos a seguir alguns dos principais algoritmos de escalonamento preemptivos.

5.4.1 Escalonamento Circular - Round-Robin

É implementado através de um algoritmo projetado especialmente para sistemas de tempo compartilhado.

O algoritmo é semelhante ao FIFO, porém, quando um processo passa para o estado de execução, existe um tempo limite para a sua utilização de forma contínua. Quando este tempo, denominado time-slice ou quantum, expira sem que antes a UCP seja liberada pelo processo, este volta ao estado de pronto (**preempção por tempo**), dando a vez a outro processo.

A fila de processos prontos é tratada como uma fila circular.

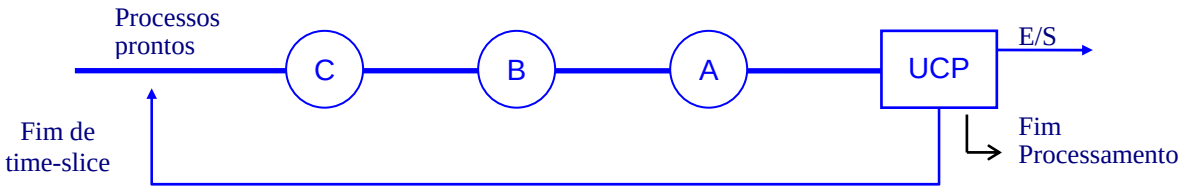
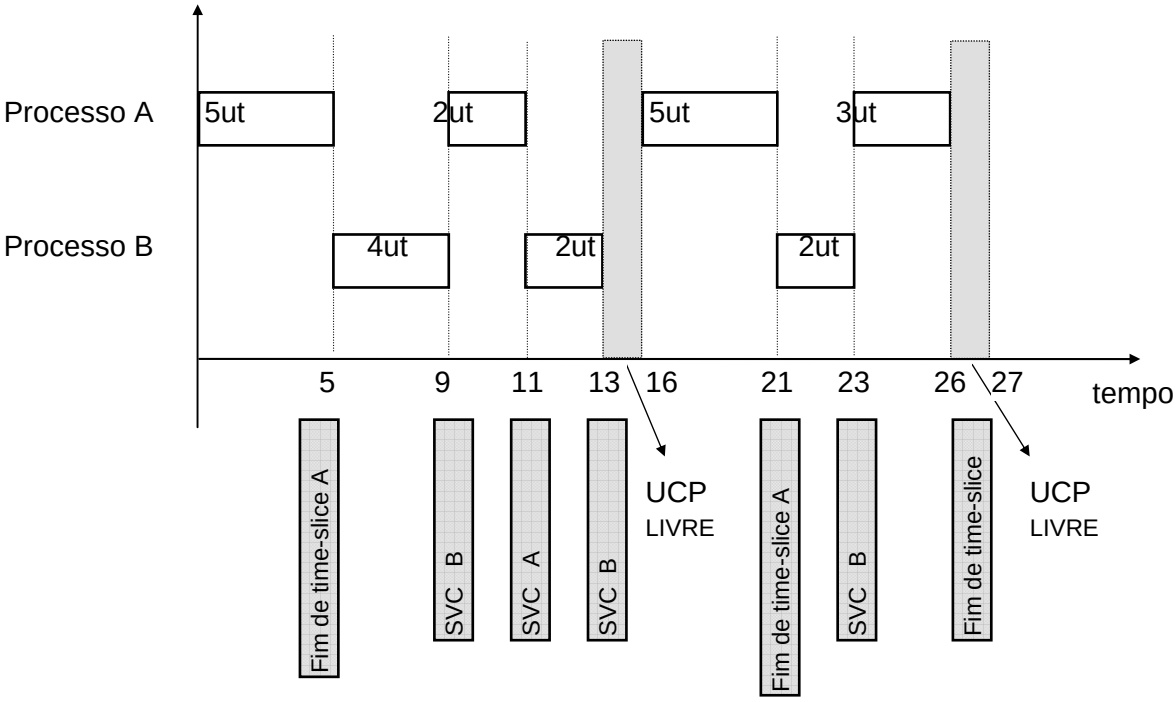
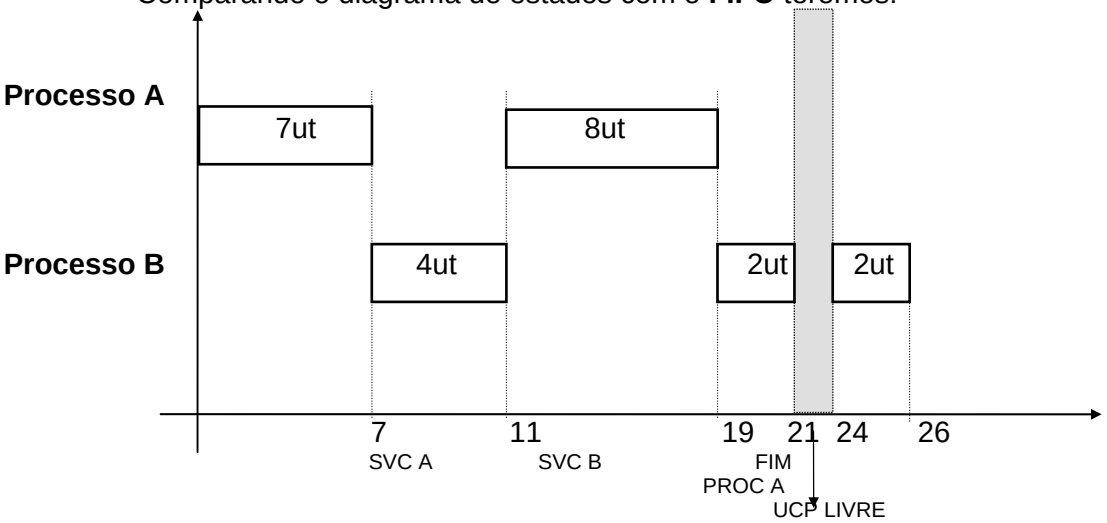


Diagrama de estados com **Round-Robin** seria:

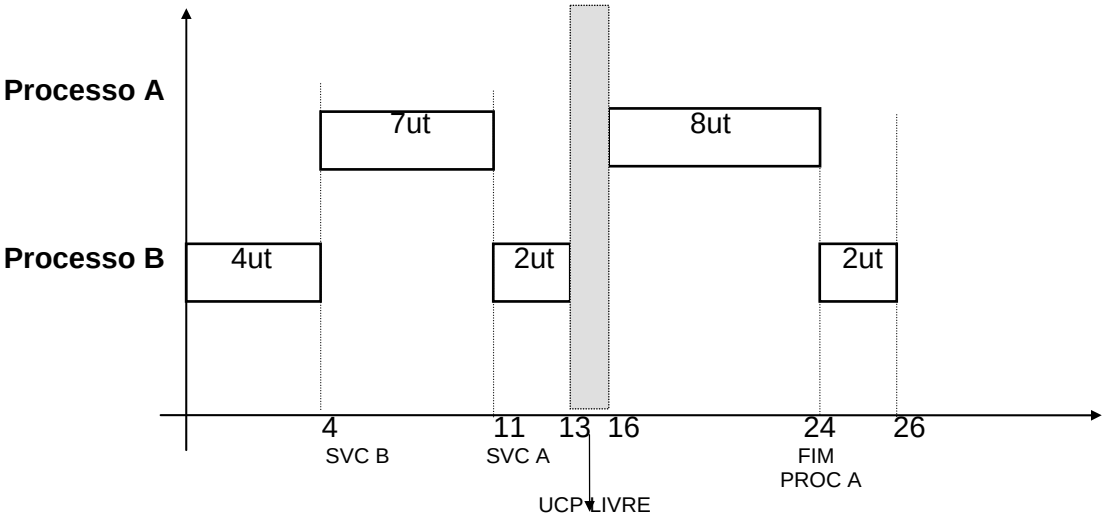


	Tempo UCP	Tipo Processo
Processo A	15	***UCP-Bound***
Processo B	08	***IO-Bound***

Comparando o diagrama de estados com o **FIFO** teremos:



Comparando o diagrama de estados com o **SJF** teremos:



5.4.2 Escalonamento por Prioridades

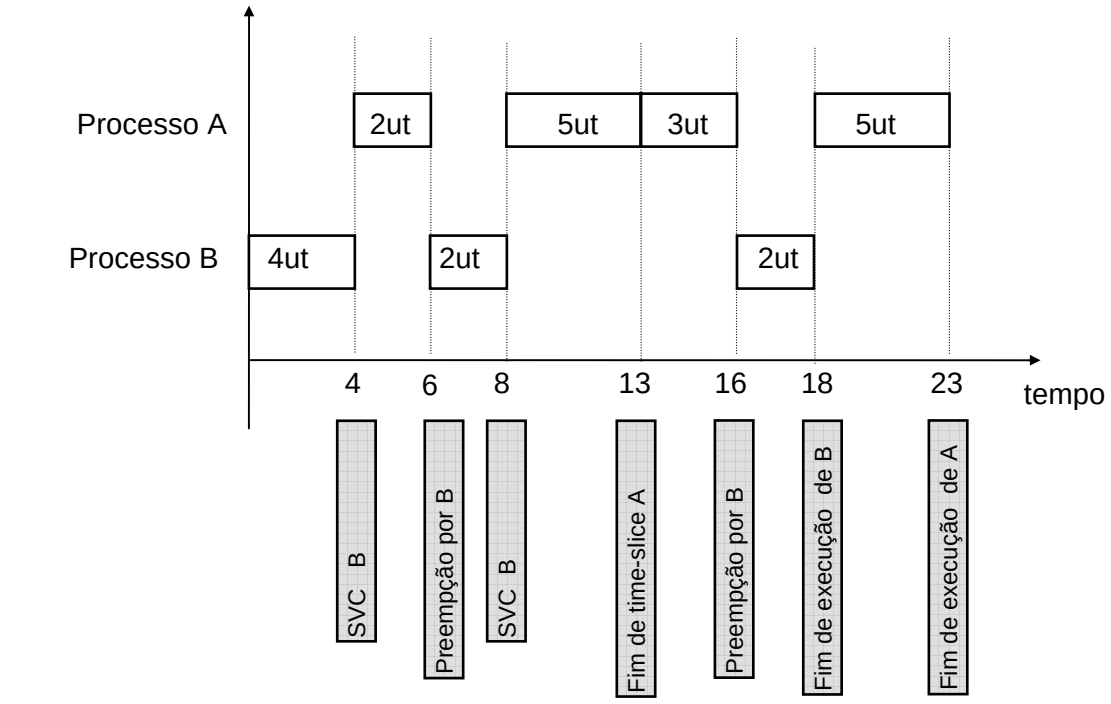
O escalonamento Round-Robin (RR) consegue melhorar a distribuição de tempo de UCP em relação aos escalonamentos não preemptivos, porém não consegue implementar um compartilhamento eqüitativo entre os diferentes tipos de processos. Isso acontece em razão do escalonamento circular tratar os processos igualmente.

No escalonamento RR os processos IO-Bound são prejudicados em relação aos processos UCP-Bound. Para compensar o excessivo tempo gasto no estado de espera, devemos atribuir alguma compensação aos processos IO-Bound. Isto pode ser feito através da variação da prioridade de execução associada a cada processo. Nesse esquema, processos de maior prioridade são escalonados preferencialmente. Toda vez que um processo for para a fila de prontos com prioridade superior a do processo em execução, o SO deverá interromper o processo corrente, colocá-lo no estado de pronto e escalonar o processo de maior prioridade para execução. Esse mecanismo é definido como **preempção por prioridade.**

Assim como na **preempção por tempo,** a **preempção por prioridade** é implementada mediante um clock, que interrompe o processador em determinados intervalos de tempo, para que a rotina de **Escalonamento de Curto Prazo** (**Escalonador ou Dispatcher**) reavalie as prioridades e, possivelmente, escalone outro processo.

A prioridade é uma característica do **contexto de SW** do processo, podendo ser **estática** ou **dinâmica.** A prioridade é dita **estática** quando não é modificada durante a existência do processo. Na prioridade **dinâmica** a prioridade do processo pode ser ajustada de acordo com o tipo de processamento realizado pelo processo e/ou pela carga do sistema. Todo processo, ao sair do **estado de espera**, recebe um acréscimo à sua prioridade. Dessa forma, os processos **I/O Bound** terão mais chance de ser escalonados e, assim, compensar o tempo que passam no **estado de espera**. Observe que este procedimento não prejudica os processos **CPU Bound**, pois estes podem ser executados enquanto os processos **I/O Bound** esperam por algum evento.

Um problema relacionado com este tipo de escalonamento é que um processo pode sofrer um **adiamento indefinido** ou **starvation** quando sempre que ele estiver na fila de processos prontos aparecer outro processo de maior prioridade. A utilização de prioridade **dinâmica** tende a diminuir este problema.

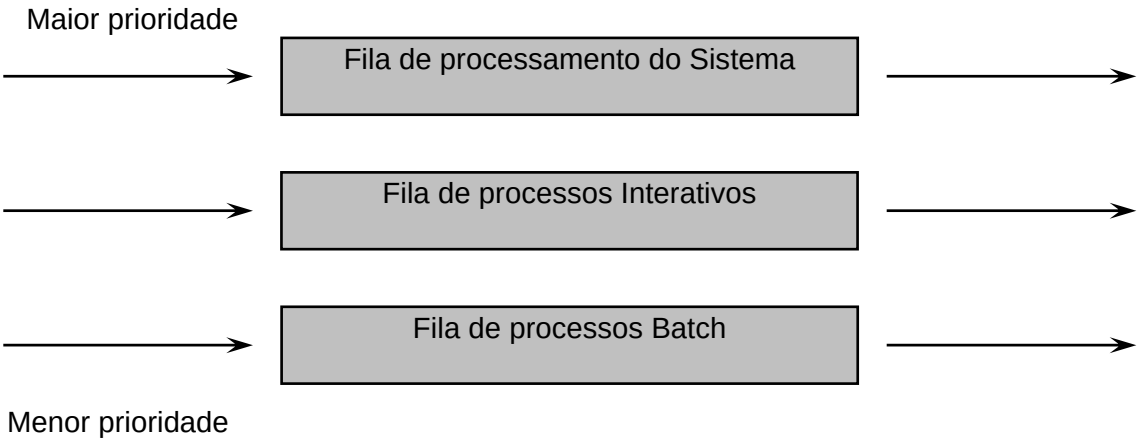


	Tempo UCP	Tipo Processo	Prioridade
Processo A	15	***UCP-Bound***	***4***
Processo B	08	***IO-Bound***	***7***

5.4.3 Escalonamento por Múltiplas Filas - Multi-Level Queues

Como os diversos processos do sistema possuem características de processamento diferentes, é difícil que um único mecanismo de escalonamento seja adequado a todos os tipos de processo. Uma boa política seria classificar os processos em função do tipo de processamento realizado e aplicar a cada grupo mecanismos de escalonamentos distintos.

Assim o **Escalonamento por Múltiplas Filas** implementa diversas filas de processo no estado de pronto, onde cada processo é associado exclusivamente a uma delas conforme figura abaixo. Cada fila possui um mecanismo próprio de escalonamento, em função das características do processo. Cada fila possui uma prioridade associada, que estabelece quais filas são prioritárias em relação às outras. O sistema só irá escalonar processos de uma fila se todas as outras filas de prioridade maior estiverem vazias.

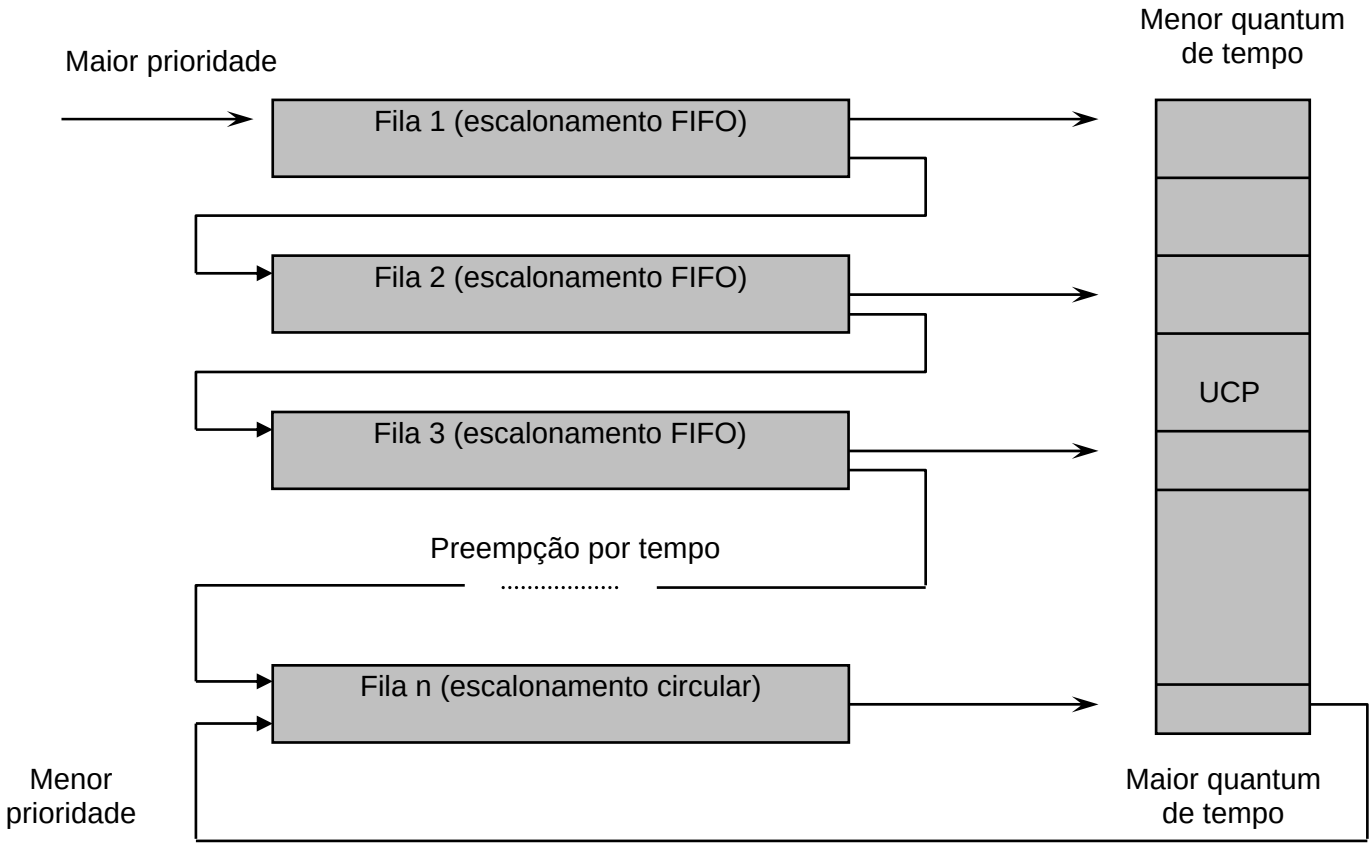


5.4.4 Escalonamento por Múltiplas Filas com Realimentação - Feedback Multi-Level Queues

No **Escalonamento por Múltiplas Filas**, os processos são previamente classificados para ser associados a uma determinada fila. No caso de um processo alterar o seu comportamento no decorrer do tempo, esse esquema é falho, pois o processo não poderá ser redirecionado para uma outra fila mais adequada. Um mecanismo ideal seria aquele em que o sistema conheça como os diversos processos e o próprio sistema se comportam ao longo do tempo, ajustando dinamicamente seus tipos de escalonamento.

O **Escalonamento por Múltiplas Filas com Realimentação**, assim como no escalonamento anterior, implementa diversas filas, onde cada fila tem associada uma prioridade de execução, porém os processos não permanecem em uma mesma fila até o término do processamento. Neste escalonamento, o sistema tenta identificar dinamicamente o comportamento de cada processo, ajustando assim as suas prioridades de execução e mecanismos de escalonamento.

Esse ajuste dinâmico permite que os processos sejam redirecionados entre as filas do sistema, fazendo com que o SO implemente um mecanismo de ajuste dinâmico, denominado **mecanismo adaptativo**, que tem como objetivo ajustar os processos em função do comportamento do sistema. Os processos não são previamente associados às filas de pronto, e sim direcionados pelo sistema entre as diversas filas com base no seu comportamento.



5.4.5 Escalonamento de Sistemas de Tempo Real

5.5 Escalonamento com Múltiplos Processadores

6 Gerência de Memória

Objetivos

i. Tornar o mais eficiente possível o compartilhamento de memória entre os processos.

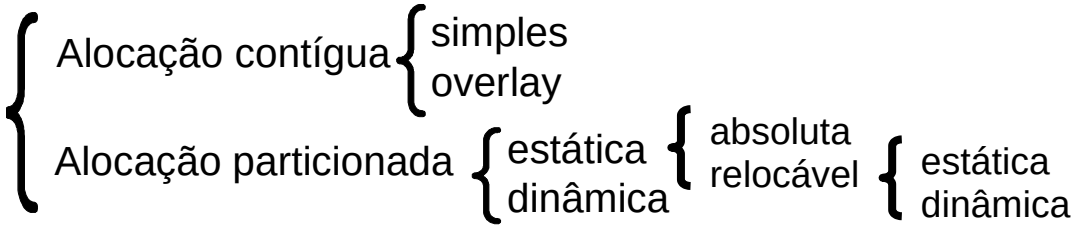
ii. Impedir que um processo acesse área de memória que não lhe pertence.

iii. Facilitar alocação de memória.

iv. Recuperar a memória liberada pelos processos.

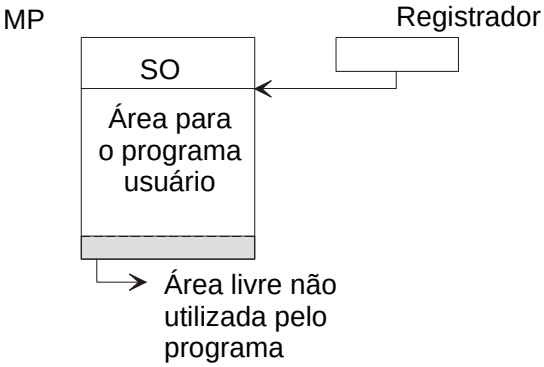
Nos sistemas monoprogramáveis a gerência da memória não é muito complexa, nos sistemas multiprogramáveis ela se torna crítica.

Temos os seguintes esquemas de alocação de memória:



6.1 Alocação Contígua

6.1.1 Alocação Contígua Simples



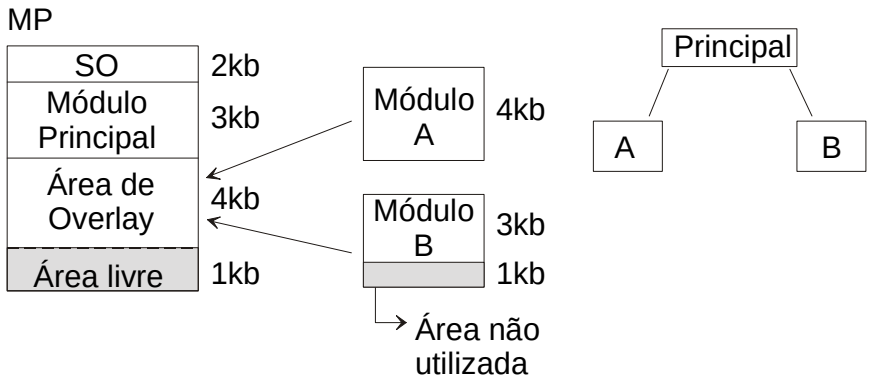
A MP é dividida em 2 partes:
uma para o SO e outra para o programa do usuário.
Nesse esquema, o usuário tem controle sobre toda a MP, podendo acessar qualquer posição de memória, inclusive para alterar e destruir o SO. Para proteção em geral se utiliza um registrador que delimita a área do SO e do usuário.

Problemas:

- não permite a utilização eficiente da UCP e da MP, pois apenas um processo pode utilizar esses recursos.
- a princípio os programas estavam limitados ao tamanho da MP disponível.

6.1.2 Alocação Contígua Overlay

A técnica de overlay divide o programa em módulos, de forma que cada parte possa executar independentemente uma da outra, utilizando uma mesma área de memória:



A definição das áreas de overlay é função do programador, através de comandos específicos da linguagem utilizada. O tamanho da área de overlay será estabelecido a partir do tamanho do maior módulo e limitado a área disponível para o programa para executar. Com esta técnica tem a vantagem de permitir ao programador expandir os limites da MP.

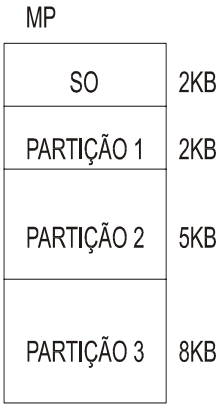
6.2 Alocação Particionada

Os SOs monoprogramáveis subutilizam UCP e MP pois não conseguem sobrepor processamento e E/S para isso foram criados os SOs multiprogramados que precisam ter vários programas de usuário carregados na máquina ao mesmo tempo e em partições distintas da MP.

6.2.1 Alocação Particionada Estática

Tabela de Partições

PART	INÍCIO	TAM
1	2KB	2KB
2	4KB	5KB
3	9KB	8KB



Partições de tamanho fixo estabelecendo na fase de inicialização do sistema (BOOT) em função dos programas que normalmente executam no ambiente.

A princípio, os programas só podiam executar em uma das partições, mesmo estando as outras livres. Essa limitação se devia aos compiladores e montadores que geravam apenas código absoluto o que gerou a **Alocação Particionada Estática Absoluta.**

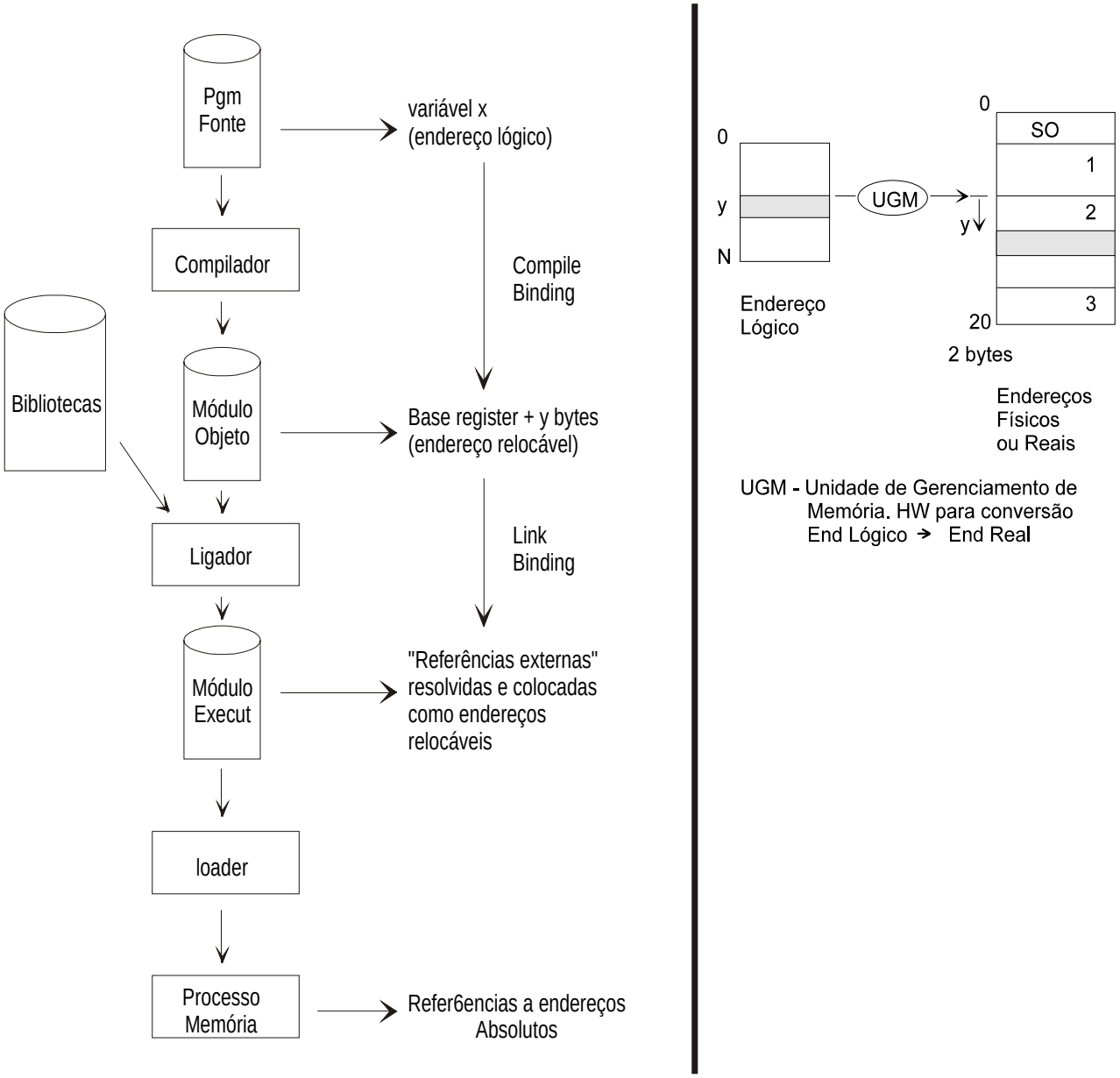
O problema da alocação particionada estática absoluta é que os programas só podem rodar na partição para qual foram compilados mesmo que existam outras partições livres.

6.2.1.1 Relocação

O problema de determinados programas só rodarem em uma partição passou a ser crítico, pois trazia uma grande ineficiência para o sistema. Somente com a evolução dos compiladores, ligadores e loaders, a geração de **código relocável** foi possível dando origem a um novo tipo de alocação chamada **Alocação Particionada Estática Relocável**.

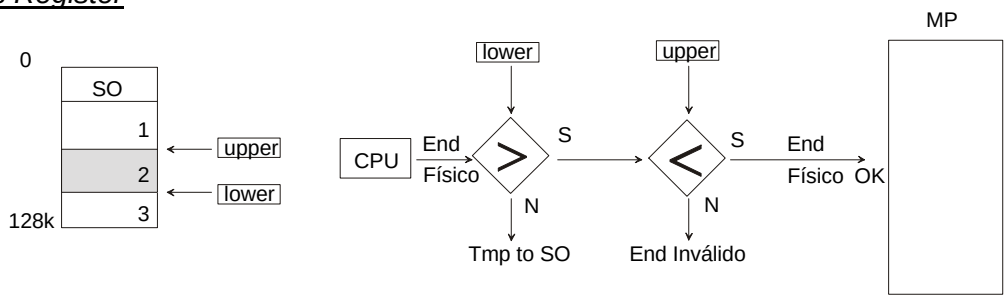
Até ser executado um programa do usuário passa por diversas fases. Endereços são representados de formas diferentes em cada fase.

A técnica de relocação consiste na utilização de um registrador (fence register) como endereço base a partir do qual todos os endereços são referenciados. O valor desse registrador é somado a todo endereço gerado pelo processo do usuário em execução.

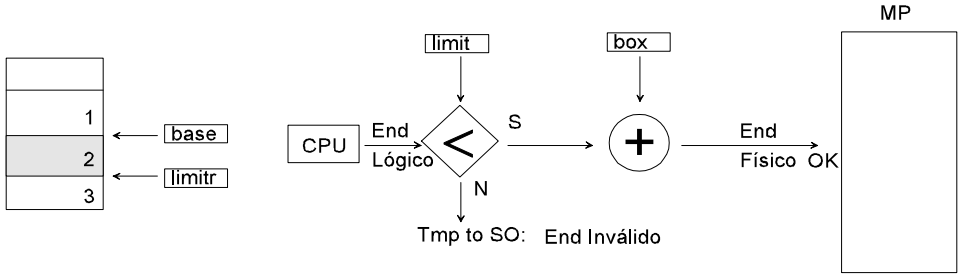


Neste novo esquema a tabela de partições contém um bit de status informando se a partição está ou não livre para ser utilizada. A proteção, nesse esquema de alocação de memória, é feita com o uso de dois registradores e pode ser implementada de duas maneiras:

i. *Bounds Register*



ii. *Base/Limits Register*

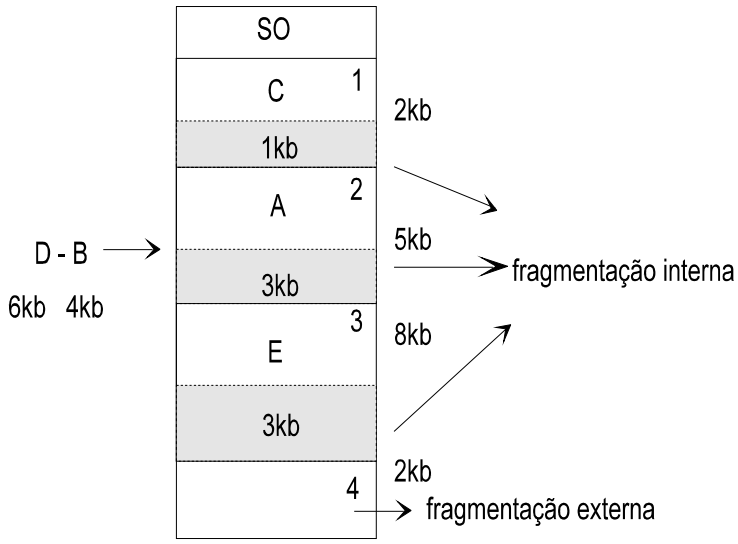


Observe que o HW da UGM é diferente para os dois casos. No caso i nós temos a chamada relocação estática (gerada em tempo de carga do programa para execução). No caso ii nós temos a relocação dinâmica (a qual exige a soma do registrador base a cada endereço referenciado o que representa um custo adicional a execução que pode ser compensado com técnicas de overlapping de instruções no processador).

Pergunta - Qual das duas técnicas deve ser usada quando existe o swapping de processos?

Resposta - Relocação Estática - só permite swapping para jobs na mesma partição o que tende a crescer o número de jobs swapped out ⇒ ineficiência.
Relocação Dinâmica - um processo pode ser swapped in para qualquer partição.

O problema básico da **Alocação Particionada Estática**, inicialmente adotada pelo OS/MFT (multiprogramming with a fixed number of tasks) da IBM é quanto à escolha do tamanho das partições para melhor atender os requerimentos de memória dos jobs atualmente processados. O **throughput** *do sistema de computação é em geral proporcional ao nível de multiprogramação*, que é afetado pela maneira como gerenciamos o uso da memória. Portanto uma boa gerência de memória (que maximize a utilização da mesma) colabora de forma decisiva para o aumento do **throughput**.



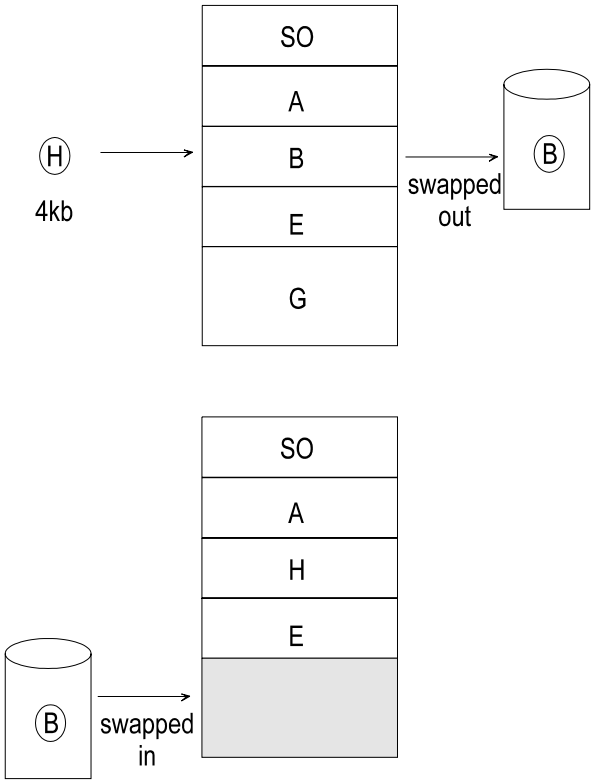
Outro problema importante é a **Fragmentação**:

i. ***interna*** - memória requerida pelo processo é menor que a disponível na partição.
ii. ***externa*** - partição livre mas não utilizada por falta de espaço.

Fragentação-Total = ∑ FragInt + ∑FragExt = (1+3+3) + 2 = 9Kb

%Memória perdida = FragTot/MemTot = 9/(2+5+8+2) = 50 %

6.2.2 Swapping



O swapping é uma técnica aplicada à gerência de memória para permitir que programas que esperam por memória livre possam ser processados. Nesta situação, o SO escolhe um programa residente que é levado da MP para o disco (swapped out) retornando posteriormente para a MP (swapped in) como se nada tivesse ocorrido.

O swapping requer o uso de um device especial chamado swap device (ou backing store), o qual deve ser grande o suficiente para acomodar os processos dos usuários que estejam na fila de processos prontos. Sempre que o escalonador (UCP scheduler) decide executar um processo ele chama o Dispatcher.

O Dispatcher verifica se o processo a ser executado está residente em memória, se não, ele remove (swap out) algum processo que esteja correntemente na MP e carrega (swap in) o processo escolhido para ser executado. Após isto o contexto do processo a ser executado é restaurado e o controle é passado ao processo que passa então a executar.

Deve ficar claro que a troca de contexto quando é feito o swapping é extremamente elevado.

Vejamos um exemplo: Tamanho do processo = 1 Mb
Taxa de transferência = 100 Mbits/s
Tempo de seek= 10 msec
Disco = 7200 rpm

Δ Tswap = ?

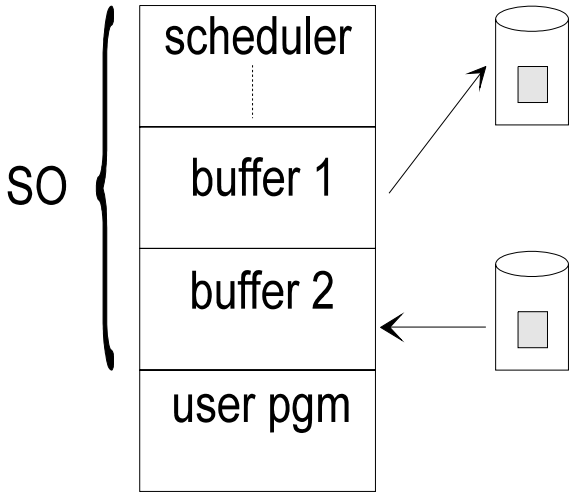
7200 voltas → 1 minuto = 60 s
t 1volta → 60/7200 s Assumimos **Tlatência = ½ t 1volta** = ½ (60/7200)

Ttransf disco-memória = Tseek + Tlatência + Ttransf =
= 10 x 10^{-3} + ½ (60/7200) + (1Mb x 8)/100 Mbits /s =
= 0,01 + 1/240 + 8/100 = 0,01 + 0,04 + 0,08 = 0,13 s

Δ Tswap = 2 * Transfdisco_memória = 0,26 seg
↓
Swap in
Swap out

Para uma utilização eficiente da UCP, nós queremos que o tempo de execução de cada processo seja longo comparado com o tempo requerido para o swap(△Tswap). Portanto no algoritmo de escalonamento Round-Robin o time-slice (time-quantum) deve ser substancialmente maior do que 0,26 segundos.

É importante que o SO tenha conhecimento do tamanho exato do processo em memória para não ter que "swappar" área desnecessária. Por esta razão programas que alocam memória dinamicamente devem fazê-lo através de system calls (request memory / release memory - malloc / free (no C)) de forma a avisar o SO estas mudanças.



O efeito do swapping na troca de contexto pode ser minimizado sobrepondo o swapping com a execução de outro programa (overlapped swapping). Dessa forma a UCP não ficará parada enquanto o swap é realizado. Enquanto o programa do usuário está executando, o processo do usuário anterior está sendo swapped out (buffer 1) e o próximo processo do usuário a ser executado está sendo swapped in (buffer2). Observe que esta estratégia requer memory-to-memory swap (Porque?) Existem restrições do swapping com relação a operações de E/S. Se um processo está esperando por uma operação de E/S que está sendo executada assincronamente pelo controlador de dispositivo de E/S acessando a área de dados do processo então o processo não pode ser swapped out.

Se o processo está em estado bloqueado (operação de E/S, está na fila de pedidos do dispositivo, que se encontra ocupado neste instante) e é swapped out então quando a operação de E/S for tratada pelo dispositivo o processo que requisitou a operação de E/S pode não estar mais na posição de memória ocupada quando da realização do pedido. As duas soluções possíveis são:

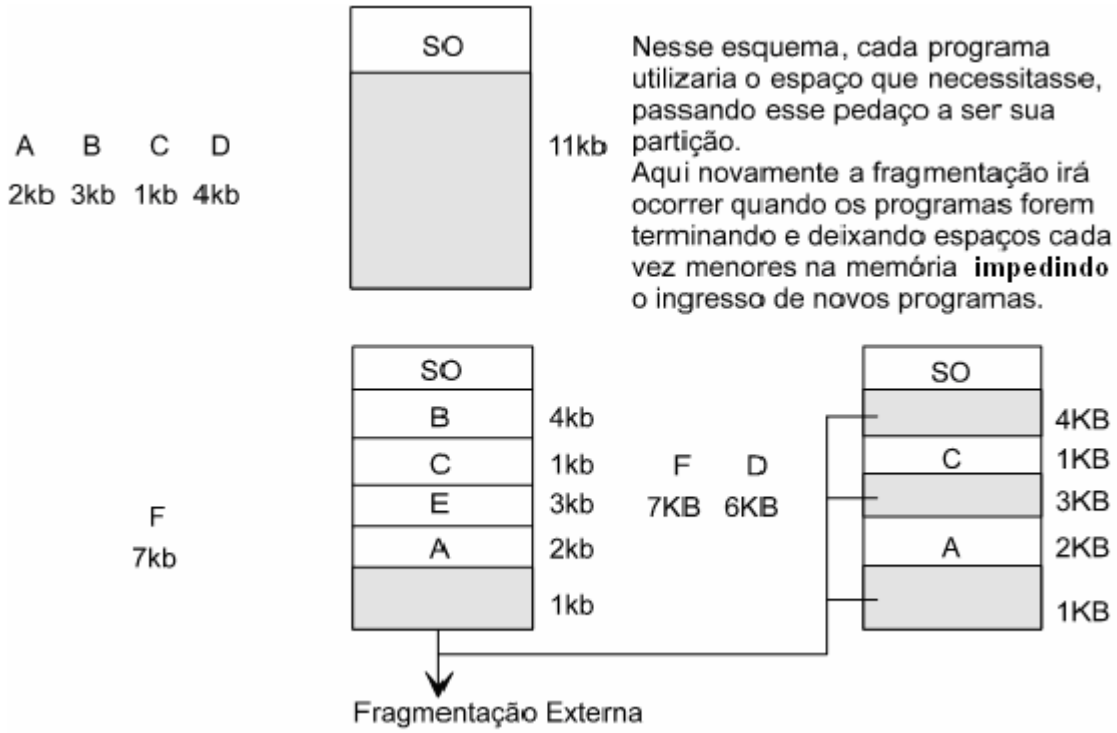
i. nunca "swappar" um processo com operações de E/S pendentes.

ii. execução de operações de E/S sempre em buffers do SO. As transferências entre buffers E/S e área do usuário acontece somente quando o processo é swapped in.

O conceito de **Swapping** permitiu um maior compartilhamento da memória e, consequentemente, um maior **throughput**. A técnica se mostrou eficiente para sistemas onde existiam poucos usuários competindo por memória e em ambientes que trabalhavam com aplicações pequenas. Seu maior problema é o elevado custo da operações de E/S (swap in/out).

6.2.3 Alocação Particionada Dinâmica

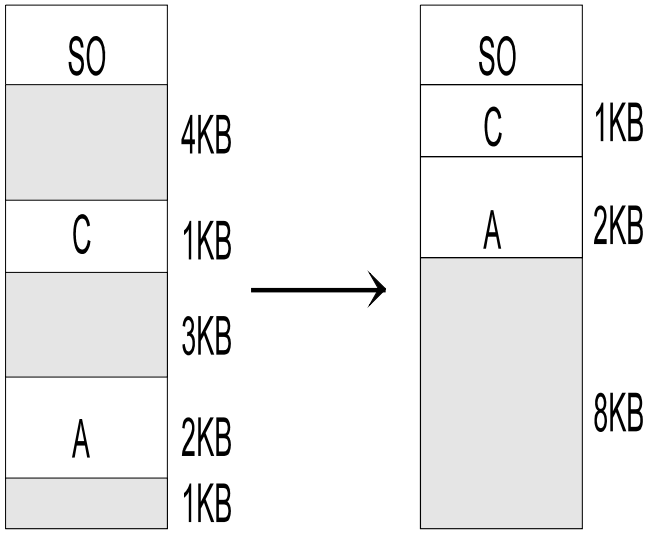
O problema principal da **Alocação Particionada Estática** era a escolha do número e tamanho das regiões de forma a diminuir a fragmentação (interna e externa). É praticamente impossível definir estes valores devido a diversidade dos jobs presentes em um Sistema de Computação. A solução é permitir que as regiões (partições) variem seu tamanho de forma dinâmica que é a chamada **Alocação Particionada Dinâmica** (ou variável) adotada pelo OS/MVT (multiprograming with a variable number of tasks) da IBM.



Apesar do esquema MVT ser melhor que o MFT, apresentando em geral uma menor fragmentação, o problema de **Fragmentação Externa** no MVT pode se tornar crítico. No pior caso podemos ter um bloco livre (não utilizado) entre cada dois processos. Se toda essa memória fosse reunida em um único bloco livre mais processos poderiam rodar nesta nova partição. A política de escolha da partição (que veremos adiante) também afeta a fragmentação, não existindo uma política melhor - FIRST-FIT é melhor em alguns sistemas e BEST-FIT é melhor em outros.

6.2.4 Compactação

Uma solução para o problema da fragmentação é o uso da técnica de compactação. O objetivo desta técnica é reunir toda memória livre num único bloco de memória, movendo de lugar as partições ocupadas.



Cabe ressaltar que a compactação só pode ser usada se o código gerado utilizar relocação dinâmica. (Explique o porque).

Existem vários algoritmos de compactação, o mais simples seria aquele que move todos processos para o topo (ou base) da memória enquanto os blocos livres são movidos em direção contrária. Este algoritmo normalmente tem um custo proibitivo devido a grande movimentação de memória requerida.

Swapping pode ser combinado com MVT. Se existe relocação estática todo job que é swapped in precisa ser executado na mesma região de memória que ocupava anteriormente, o que força a necessidade de se remover da memória (swap out) os processos que por ventura estejam utilizando a sua região. Se existe relocação dinâmica então o job pode ser swapped in em uma posição diferente.

Um algoritmo alternativo para a compactação é remover processos da memória (swap out) e recarregá-los (swap in), quando forem ser re-executados, em posições diferentes.
Outra forma de se reduzir a fragmentação externa é reduzir o tamanho médio dos processos sendo executados. A título de exemplo observamos que o PDP-I0 (1978) já dividia o código em dois segmentos:

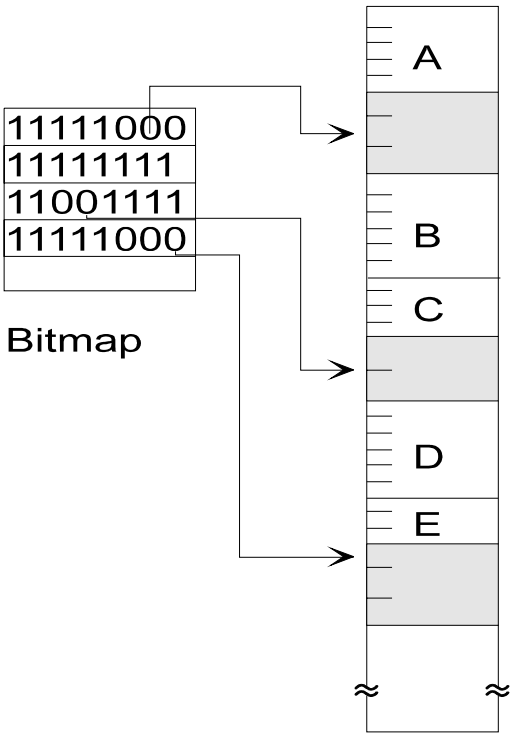
a) Instruções (code segment) + dados constantes - high memory
b) Variáveis (data e stack segment) - low memory

Este mecanismo permite o compartilhamento entre diversos programas que utilizem o mesmo código sobre diferentes dados. Estratégia esta utilizada principalmente para compiladores, ligadores, editores de texto.

6.2.5 Estratégias para Escolha e Gerenciamento de Partições

Os SOs implementam estratégias para determinar qual partição livre um programa será carregado para execução. O objetivo destas estratégias e tentar evitar, ou diminuir, o problema da fragmentação antes que ela ocorra. Estas estratégias dependem de vários fatores tais como: tamanho médio dos processos e gerenciamento das áreas livres pelo SO. Portanto a escolha da estratégia ótima é uma tarefa complicada. Em termos gerais existem 3 maneiras para o SO controlar (gerenciar) o uso da Memória Principal: Bitmaps, Listas Encadeadas, Buddy Systems.

6.2.5.1 Bitmaps

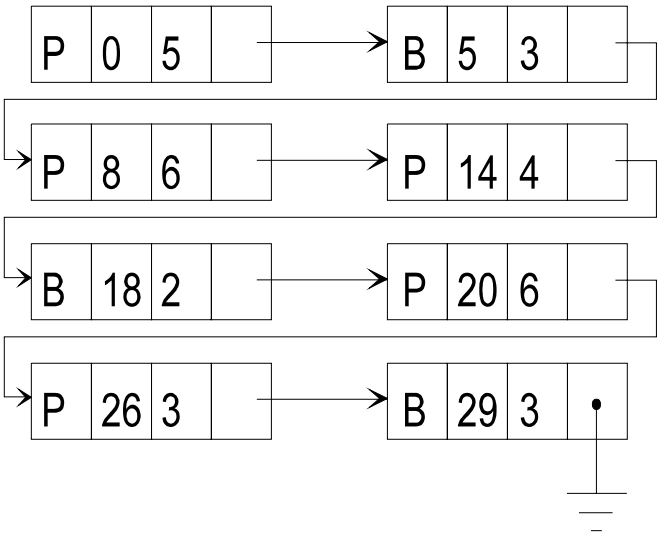


Neste caso a MP é dividida em unidades de alocação (clusters). A cada unidade de alocação se associa um bit no Bitmap que tem o valor 0 (zero) quando o cluster está livre e 1 (um) em caso contrário. O número de entradas na tabela (Bitmap) é proporcional ao tamanho do cluster. O tamanho do cluster define o percentual de fragmentação interna.

O principal problema desta organização é que toda vez que se deseja carregar um processo que ocupe k clusters devemos varrer a tabela para procurar k bits zero consecutivos o que é extremamente lento.

(comparação de strings)

6.2.5.2 Listas Encadeadas

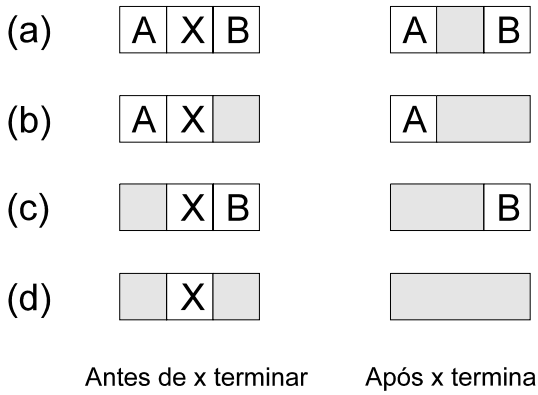


Neste caso é mantida uma lista encadeada com os segmentos alocados (P-Processo) e os livres (B-Buraco). Cada entrada na lista contém informações de onde começa e o tamanho de cada processo, ou buraco.

Em geral a lista é ordenada por endereço desta forma quando um processo termina ou é swapped out a atualização da lista é imediata.

(Comparação lógica)

Existem as seguintes possibilidades para remoção de processo:



Exercício:

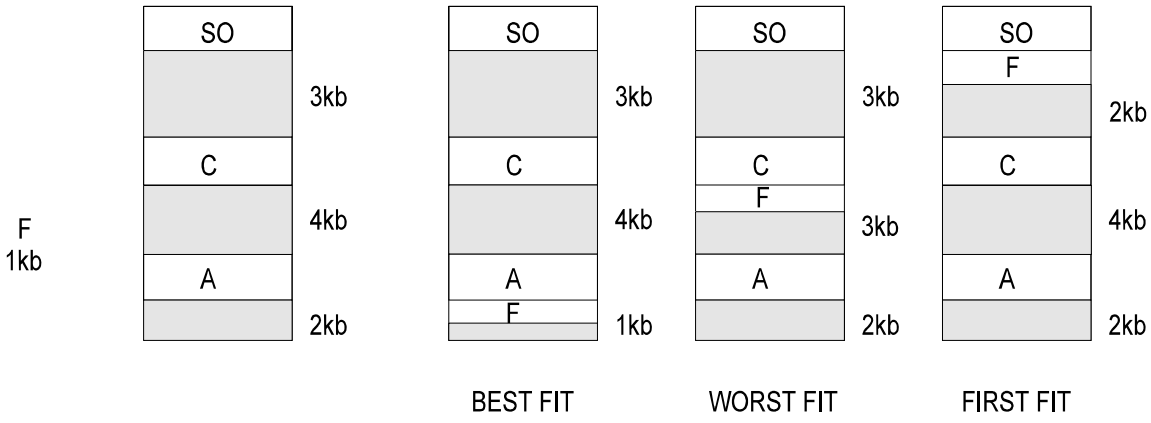
Diga o que deve ser feito em cada situação com a lista.

Diversos algoritmos podem ser utilizados para alocar memória para o novo processo a ser carregado:

- **First-Fit** - escolha a primeira partição livre, de tamanho suficiente para carregar o processo. O Buraco é dividido em duas partes, um para o processo e outro para o novo buraco. É um algoritmo rápido pois ele não faz muitas pesquisas na lista. Existe uma variação do FIRST FIT chamada a NEXT FIT que começa a pesquisa na lista a partir da última posição encontrada ao invés de começar desde o princípio.

- **Best-Fit** - escolhe a melhor partição, i.e., aquela em que o programa deixa o menor espaço sem utilização. Nesse algoritmo, a lista está ordenada por tamanho, diminuindo o tempo de busca por uma área desocupada. A desvantagem é que cada vez mais a memória fica com pequenas áreas não contíguas, aumentando o problema de fragmentação externa.

- **Worst-Fit** - a idéia deste algoritmo é tentar diminuir o problema da fragmentação externa gerada pelo BEST FIT escolhendo a partição que deixa o maior espaço sem utilização.



6.2.5.3 Buddy System

Observa-se que no esquema anterior criando-se listas encadeadas (ou duplamente encadeadas) com áreas livres (buracos) e os processos (ou mesmo listas diferentes para cada um) torna o processo de alocação de memória extremamente rápido, entretanto o processo de liberação de memória quase sempre é mais demorado (veja casos b, c e d de listas encadeadas).

O buddy system (Knuth, 1973) é um algoritmo para gerenciamento de memória que tira vantagem do fato da aritmética nos computadores ser binária para acelerar o merge de buracos adjacentes quando um processo é terminado ou é swapped out.

O algoritmo mantém uma lista de blocos livres de tamanho: 1, 2, 4, 8, 16, 32, etc bytes (potências crescentes de 2). Para uma memória de 1 MB teríamos 21 listas (Porque?). No início toda MP está livre e a lista de blocos de 1 MB tem um único elemento (buraco de 1 MB) enquanto as outras listas estão vazias. A medida que são requisitado blocos de tamanho x encontra-se o bloco com tamanho potência de 2 imediatamente superior a x (Exemplo: x=70k ⇒ bloco 128 k) caso não exista bloco livre na lista de 128 kb, o algoritmo faz a divisão de blocos maiores para se criar um bloco de 128k para armazenar o processo. Vejamos um exemplo:

MP

	0	128	256	384	512	640	768	896	1MB	
Inicialmente										1
Pedido 70k	A		128		256			512		3
Pedido 35k	A	B	64		256			512		3
Pedido 80k	A	B	64	C		128		512		3
Liberação A	128	B	64	C		128		512		4
Pedido 60k	128	B	D	C		128		512		4
Liberação B	128	64	D	C		128		512		4
Liberação D		256		C		128		512		3
Liberação C					1024					1
										Buracos

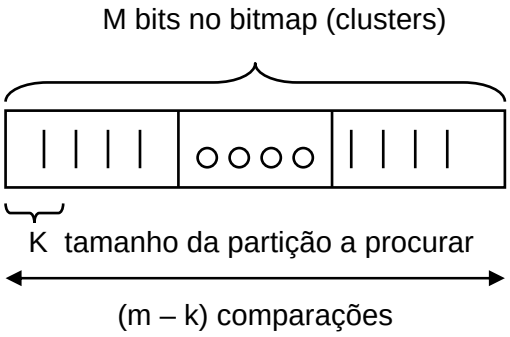
Analisando a figura vemos que quando um pedido de 70k é feito o bloco de 1MB é dividido em 2 de 512bytes (buddies). Posteriormente o bloco de 512k inicial (que começa no endereço 0) é dividido em 2 blocos de 256kbytes e o bloco que começa no endereço 0 é subdividido novamente em 2 buddy blocos de 128kb e o primeiro é alocado ao processo. Observe que a lista de blocos de 1MB agora está vazia, as listas de 512kb e 256kb tem 1 bloco cada uma e a lista de 128kb tem 2 blocos (sendo 1 livre).

A vantagem deste algoritmo é que quando um bloco de tamanho 2kbytes é liberado, somente os blocos buddy (lista de 2kbytes) devem ser pesquisados para um possível merge. No caso do algoritmo de listas encadeadas todos os buracos devem ser pesquisados para a verificação da possibilidade de merge o que torna o algoritmo mais rápido na liberação de memória.

O problema deste algoritmo é uma ineficiente utilização de memória devido ao problema de fragmentação interna (Porque?)

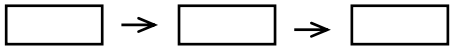
Comparação entre as 3 estratégias:

BITMAP



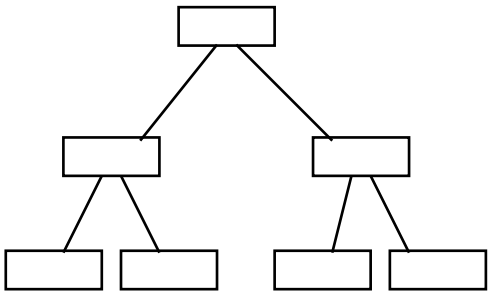
$(m-k)\ k \cong mk - k^2$

LISTAS ENCADEADAS



N comparações
N – no. De processos ou buracos
Em geral n <<< m

BUDDY SYSTEM



Log_2 n comparações

$N/2^0$ n → nós → 1 comparação
↓
$N/2^1$ n/2 → nós → 1 comparação
↓
$N/2^2$ n/4 → nós → 1 comparação
↓
$N/2^3$ n/8 → nós → 1 comparação
⋮
$N/2^{k-1}$ 1 → nós → k comparação OK

$N/2^{k-1} = 1 \Rightarrow n = 2^{k-1}$
$\log_2 n = \log_2 2^{k-1}$
$\log_2 n = \log_2 2 \Rightarrow k = \log_2 n + 1$

6.3 Memória Virtual

Memória Virtual (virtual memory) é uma técnica sofisticada e poderosa de gerência de memória, onde as memórias principal e secundária são combinadas, dando ao usuário a ilusão de existir uma memória muito maior que a memória principal.

O conceito de **Memória Virtual** está baseado em desvincular o endereçamento feito pelo programa dos endereços físicos da memória principal. Assim, os programas e suas estruturas de dados deixam de estar limitados ao tamanho da memória física disponível o que facilita a tarefa de programação, já que o programador não precisa mais criar overlays.

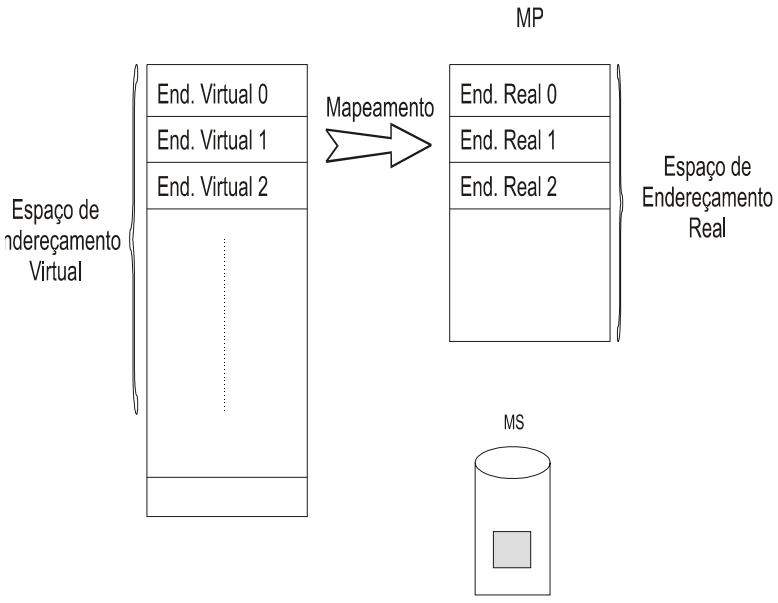
O mecanismo MVT, apesar de ser uma evolução em relação ao MFT, ainda sofre o problema de fragmentação externa. Geralmente esta situação acontece quando a memória disponível não está contígua mas dividida entre áreas alocadas a outros processos. Uma vez que a área ***alocada a um processo deve ser contígua*** duas coisas podem ser feitas para reunir estas áreas não contíguas:

(i) **Compactação** - movimentação de memória livre para gerar uma região contígua através de movimentação dos programas em memória ou via **Swapping**.

(ii) **Paginação** - permitir que a memória utilizada pelo programa seja <u>não contígua</u> ou ao menos <u>contígua por partes</u>.

6.3.1 Mapeamento

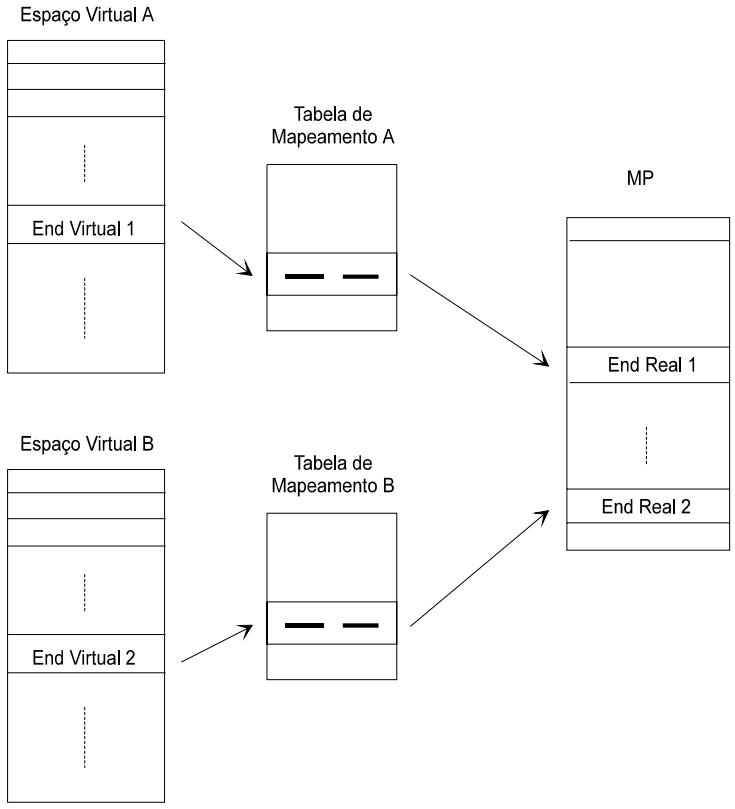
Um programa executando em ambiente de memória virtual não faz referência a endereços físicos de memória (endereços reais), mas apenas a endereços virtuais. No momento da execução de uma instrução, o endereço virtual é traduzido para um endereço físico, pois o processador acessa apenas posições da MP. O mecanismo de tradução do endereço virtual para endereço físico é denominado **<u>mapeamento</u> (maping).**



O **EEV (espaço de endereçamento virtual)** não tem nenhuma relação direta com os endereços do **EER (espaço de endereçamento real)**. Um programa pode fazer referência a endereços virtuais que estejam fora dos limites do espaço real, ou seja, os programas e suas estruturas de dados não estão limitados ao tamanho da memória física disponível. Como os programas podem ser maiores que a memória física, somente parte deles pode estar residente na MP em um determinado instante. O SO utiliza a MS como extensão da MP. Quando um programa é executado, somente uma parte do código fica residente na MP, o restante permanece na MS até o momento de ser referenciado.

Quando o usuário desenvolve suas aplicações, ele ignora a existência dos endereços virtuais. Os **compiladores** e **linkers** se encarregam de gerar código executável em função desses endereços, e o SO cuida dos detalhes de sua execução.

Como conseqüência do mapeamento, um programa não precisa estar necessariamente contíguo na MP para ser executado. Nos SOs atuais a tarefa de tradução é realizada por um HW específico dentro da UCP (**UGM - unidade de gerenciamento de memória**), juntamente com o SO, de forma a não comprometer seu desempenho e torná-lo transparente para a aplicação. Como a maioria das aplicações tende a fazer referência a um número reduzido de endereços virtuais (**princípio da localidade**), somente uma pequena fração da tabela de mapeamento é realmente necessária, por isso criou-se um HW especial para mapear **endereços virtuais** para **endereços físicos** sem a necessidade de se fazer um acesso a tabela de mapeamento. Este HW é chamado de **TLB - translation lookaside buffer** e tem o mesmo princípio de funcionamento da cache de memória.



Cada processo tem o mesmo espaço de endereçamento virtual como se possuísse a sua própria MV.

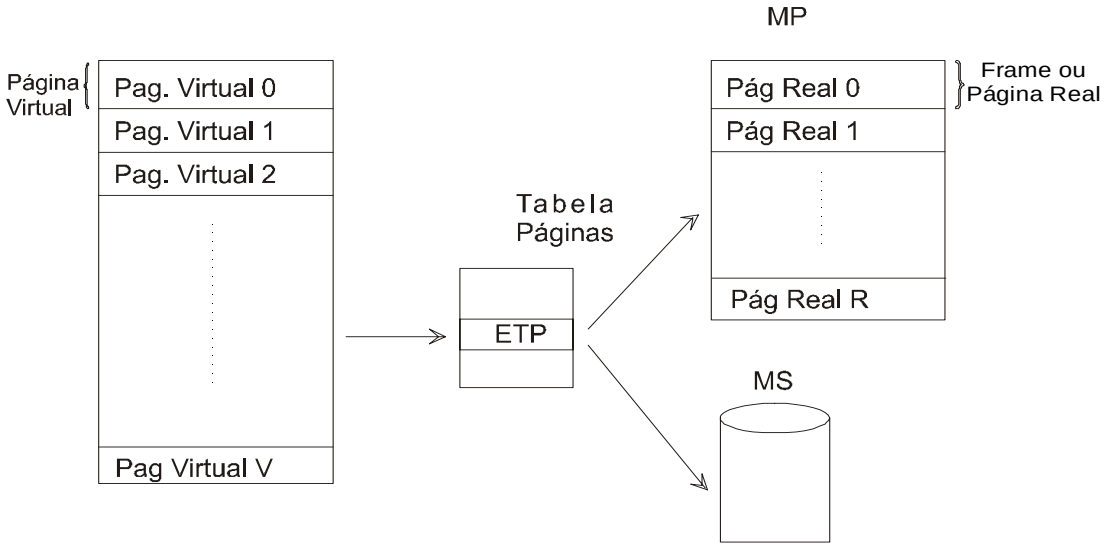
O mecanismo de tradução se encarrega, então, de manter tabelas de mapeamento exclusivas para cada processo. Quando um programa está sendo executado o sistema utiliza a tabela de mapeamento do processo, no qual o programa executa, para realizar a tradução.

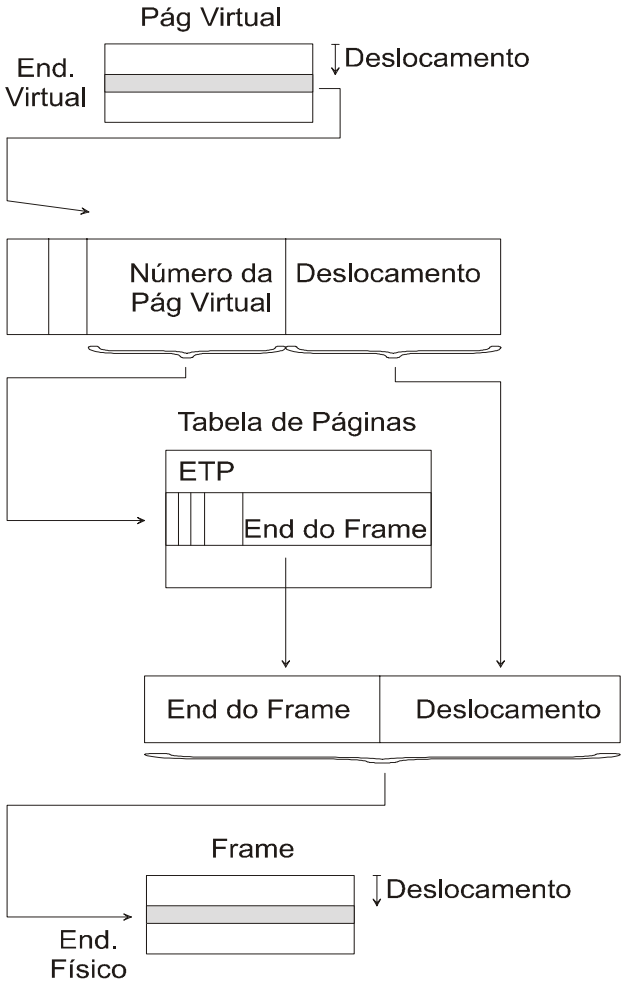
A informação correspondente à posição inicial da tabela de mapeamento é em geral indicada por um registrador, chamado **PTBR** (**page table base register**) e faz parte do contexto do HW do processo.

A **tabela de mapeamento** mapeia **blocos de informação** cujo tamanho determina o número de entradas necessário na tabela. *Quanto maior o bloco, menor número de entradas na tabela e, conseqüentemente, tabelas menores; entretanto blocos maiores aumentam o tempo de transferência do bloco entre **MS** e **MP***. Veremos adiante que existem SOs que trabalham apenas com blocos de mesmo tamanho (**páginas**), outros que utilizam blocos de tamanho diferente (**segmentos**) e, ainda há SOs que trabalham com os dois sistemas.

6.4 Paginação

Espaço de endereçamento virtual e o espaço de endereçamento real são divididos em blocos de mesmo tamanho (páginas). Todo mapeamento é feito em nível de página, através da ETP (entrada na tabela de página), com informações que permitem ao SO encontrar a página real correspondente.





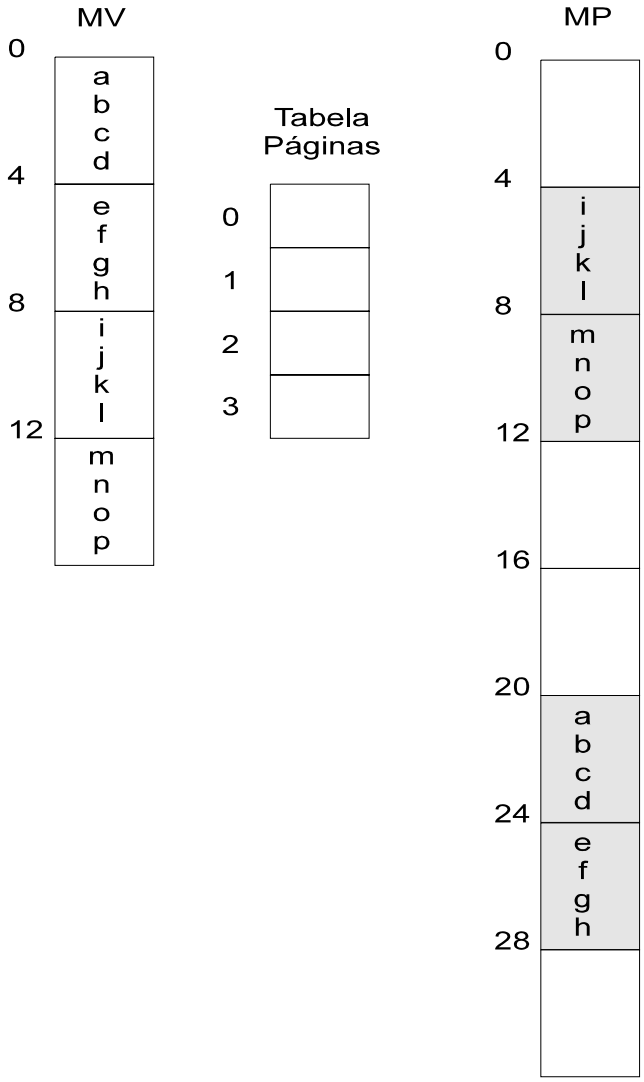
Quando um programa é executado, as páginas virtuais são carregadas da MS para a MP e colocadas nos frames disponíveis e a tabela de páginas é atualizada com os frames utilizados. Sempre que o programa fizer referência a um endereço virtual, o mecanismo de mapeamento localiza na ETP da tabela do processo o endereço físico do frame para ser acessado.

O tamanho da página (e do frame) é definido pelo HW, sendo tipicamente uma potência de 2. Por exemplo, o IBM 370 usa 2k ou 4k bytes/pág. Em geral, se o **tamanho da página** é **P** então um **endereço virtual V** produz um **número de página virtual NPV** e um **deslocamento d** dentro página relacionados da seguinte forma:

NPV = V div P
d = V mod P

Onde div e mod são respectivamente divisão inteira e resto da divisão. (Explique porque).

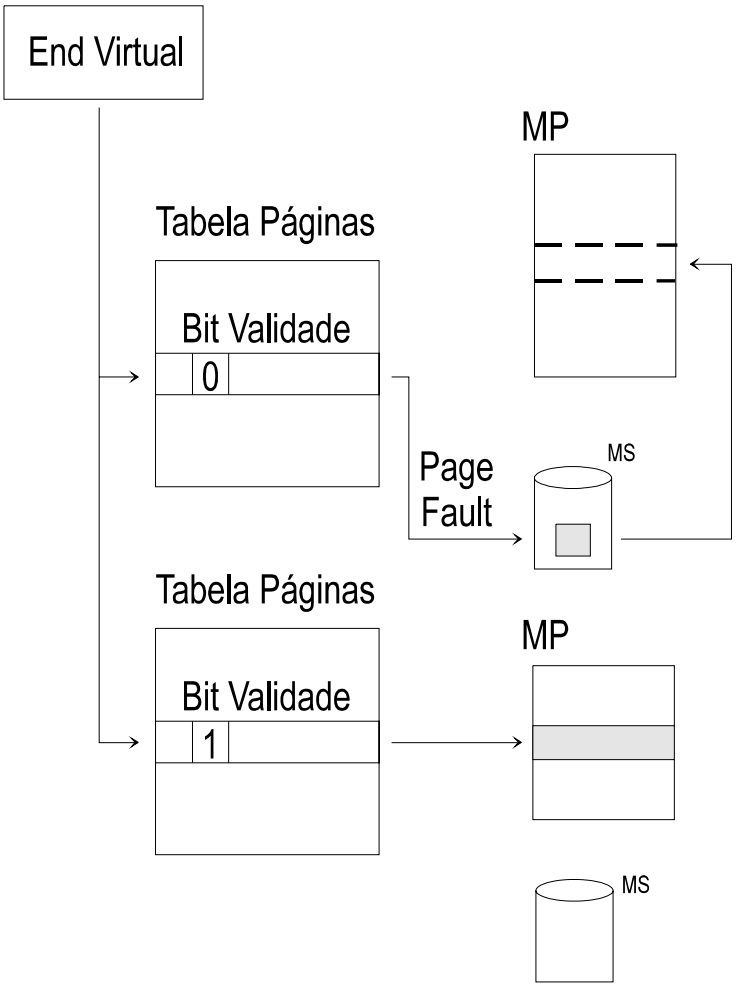
Exemplo: Para o caso em que o tamanho Pag=4 palavras; MP=32 palavras, MV=16 páginas. Preencha a tabela de páginas desse processo.



Outra observação importante é que se o tamanho da página é uma potência de 2, e é 2^n, então os n bits menos significativos de um endereço virtual designam o deslocamento (offset) dentro de uma página, enquanto os bits restantes mais significativos designam o número da página virtual (NPV). Desta forma sendo o tamanho da página um múltiplo de 2 podemos evitar o processo de divisão (fórmula anterior).

É importante ressaltar que o mecanismo de paginação é uma forma de relocação dinâmica. Todo endereçamento virtual é mapeado pelo mecanismo de paginação para um endereço real.

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



Sempre que o processo faz referência a um endereço virtual o SO verifica se a página que contém o endereço referenciado está ou não na MP. Caso não esteja, o SO acusa a ocorrência de um **page-fault** (interrupção) e uma página é então transferida da MS para a MP.

As páginas dos processos são transferidas da MS para a MP, apenas quando são referenciadas. Este mecanismo, é chamado **paginação por demanda (demand paging)** e é conveniente, na medida em que leva para a MP apenas as páginas realmente necessárias para a execução do programa. Existe uma tendência de os SO's modernos passarem a utilizar a técnica de **paginação antecipada (antecipatory paging).**

Os problemas dessa técnica é que quando o SO erra na previsão das próximas páginas a serem referenciadas terá sido perdido tempo de processador e ocupado memória desnecessariamente.

Fragmentação também está presente em sistemas paginados, só que em menor escala, se comparada com a de outras organizações já vistas. A fragmentação só é encontrada, realmente, na última página, quando o código não a ocupa por completo. Maior ou menor fragmentação depende do tamanho da página. Porém o tamanho da página influencia outros fatores tais como:

- **tamanho da tabela de mapeamento**;
- **taxa de paginação**, expressa pelo número de page-faults do sistema por unidade de tempo;
- **fragmentação** - percentual de utilização da MP

6.4.1 Proteção em Sistemas Paginados

Em qualquer sistema multiprogramável, onde vários processos compartilham a memória principal, deve existir um mecanismo que proteja o espaço de memória real de cada processo e, principalmente a área do SO.

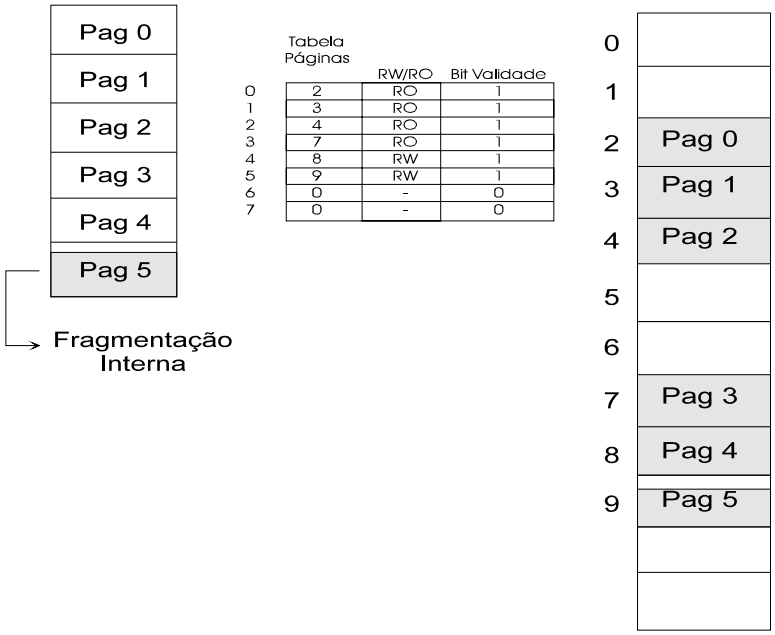
É importante observar que como no esquema de memória virtual, cada processo tem a sua própria tabela de mapeamento, e a tradução dos endereços é realizada pelo SO, torna-se ***impossível*** *que um processo tenha acesso a áreas de memória de outros processos*, a menos que haja compartilhamento explícito de páginas (ou segmentos) conforme veremos a seguir.

A proteção é necessária para impedir que um processo, ao acessar uma página (ou segmento) do SO, a modifique ou mesmo tenha acesso a ela. Mesmo as páginas do processo podem estar definidas, por exemplo, com uma proteção que impeça a gravação, como é o caso das páginas de código executável.

O mecanismo de proteção para sistemas paginados é feito através de **bits de proteção** associados a cada página (na ETP). Em geral **dois bits** são reservados para identificar se a página é **read/write**, **read only** ou **sem acesso**. Uma tentativa de se escrever em uma página RO ocasiona uma interrupção chamada violação de memória (memory protection violation).

A arquitetura da máquina define o range de endereços válidos para um programa. Por exemplo uma máquina com MAR de 16 bits pode gerar endereços virtuais entre 0 e 65535. Com os registradores de fronteira (barrier registers) ou base e limite (base limit register) nós podemos trapear endereços gerados erroneamente por programas.

Um problema colateral gerado pela fragmentação interna é que referências a pág 5 (conforme figura acima) mesmo que além da área definida pelo programa não gerarão violação de endereço !



6.4.2 Working Set

O mecanismo de memória virtual apesar de suas vantagens, introduz um grande problema. Sempre que um processo faz referência a uma de suas páginas e esta não se encontra na memória ocorre um **page-fault**, que exige do SO pelo menos uma operação de E/S, que, quando possível, deve ser evitada. A taxa de **page-faults** gerada por um programa depende de como a aplicação foi desenvolvida, além da política de gerência de memória implementada pelo SO.

Qualquer sistema que implementa **paginação** deve se preocupar em manter na memória principal um certo número de páginas que reduza ao máximo a taxa de **paginação** dos processos, ao mesmo tempo que não prejudique os demais processos que desejam ter acesso à memória.

O conceito de **working set** surgiu a partir da análise da taxa de paginação dos processos. Quando um programa iniciava a sua execução, percebia-se uma elevada taxa de **page-faults**, que se estabilizava com o decorrer de sua execução. Este fato está ligado diretamente a um outro conceito chave na estratégia de gerência de memória chamado **princípio de localidade**, que foi observado experimentalmente por ***Donald Knuth :***

"Programas tendem a reutilizar dados e instruções que foram utilizados recentemente. Uma heurística aplicada a quase todos os programas é que geralmente um programa gaste 90% do seu tempo de execução em somente 10% do código".

Uma implicação deste princípio é que baseado no passado recente da execução de um programa, alguém pode predizer com uma precisão razoável quais instruções e dados serão referenciados em um futuro próximo. É nesta informação que se baseiam as políticas de gerenciamento de cache e gerenciamento de política de alocação de páginas.

Existem dois tipos de localidade:

- **Localidade temporal** - itens recentemente acessados têm alta probabilidade de serem acessados em um futuro próximo.

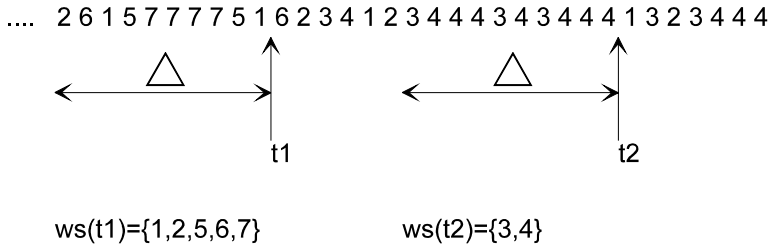
- **Localidade espacial** - itens cujos endereços são próximos tendem a ser acessados em um futuro próximo.

A localidade tem muito a ver com a forma que a aplicação foi escrita. Normalmente, se um programa foi desenvolvido utilizando técnicas estruturadas, o conceito de localidade quase sempre é válido.

A partir da observação do **princípio da localidade**, formulou-se a teoria do **working set (ws)**. O ws é o conjunto de páginas que um processo referencia constantemente, e, por isso mesmo, deve permanecer na MP. Caso contrário, o sistema poderá sofrer com a elevada taxa de paginação, comprometendo sua performance.

Quando um processo é criado, todas as suas páginas estão na MS. À medida que acontecem referências às páginas virtuais, elas são transferidas para o ws do processo na MP (page in). O **ws** usa um parâmetro, Δ, para definir a **janela de working set** (**ws window**). O conjunto das últimas Δ páginas referenciadas forma o **ws**. Se uma página está sendo acessada, então ela está no working set. Se a página não está sendo acessada ela deixará o **ws** após Δ referências a outras páginas distintas. Portanto o **ws** é uma aproximação da localidade de um programa. A acurácia do **ws** depende da seleção do parâmetro **Δ.** Se Δ for muito pequeno, ele não abrigará todo **ws**; se for muito grande, ele pode conter diversas localidades de um programa.

Páginas referenciadas pelo programa com Δ=10:



ws(t1)={1,2,5,6,7} ws(t2)={3,4}

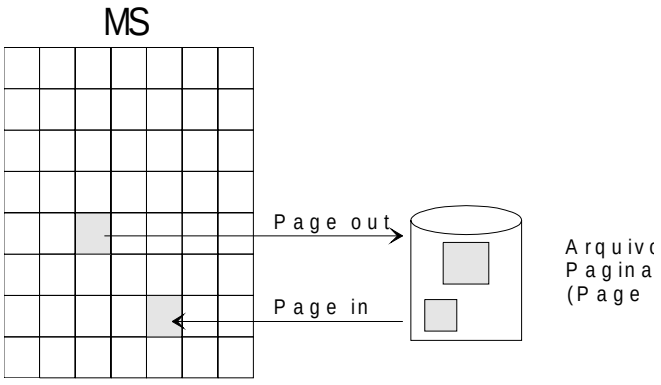
A propriedade mais importante no **ws** é o seu tamanho. Se nós computarmos o tamanho do working set de cada processo (**wss**$_i$) no sistema então **D =** $\sum$ **wss**$_i$ é o número total de páginas sendo requeridas pelos processos. Se **D > total-de-frames-disponíveis** nós começaremos a ter problemas (**trashing**) uma vez que alguns processos não poderão ser executados por falta de frames suficientes.

O SO monitora o **ws** de cada processo e aloca para ele o número de páginas necessárias (**wss**$_i$) para completar o **ws** do processo. Se existem frames suficientes um novo processo pode ser iniciado. Se a soma dos **ws's** dos processos (**D**) aumenta ultrapassando o número de frames disponíveis, o SO escolhe um processo para suspender a sua execução. As suas páginas são gravadas em disco (**swapped out**) e os seus frames são alocados para novos processos. O processo suspenso pode ser reescalonado mais tarde.

A estratégia do **ws** serve para prevenir o **trashing** e elevar ao máximo possível o **grau de multiprogramação**, o que acarreta uma otimização no uso da UCP.

6.4.3 Políticas de Realocação de Páginas

O maior problema na gerência de memória virtual por paginação não é decidir que página carregar para a MP, mas quais páginas remover. Quando o limite do **ws** do processo (**wss**) é alocado, e este necessita de frames, o SO deve intervir e escolher, dentre as diversas páginas do seu **ws**, quais que devem ser liberadas. Sempre que o SO libera uma página, cujo seu conteúdo tenha sido alterado, ele antes deverá gravá-la na MS (**page out**) no arquivo de **paginação** (**page file**) onde as páginas alteradas são armazenadas. Sempre que uma dessas páginas for novamente referenciada, ela será trazida novamente para o **ws** do processo (**page in**). O SO controla o salvamento de páginas através do **bit de modificação** (**dirty/modify bit**), que existe na entrada de cada tabela de páginas.



A melhor estratégia de relocação de páginas seria, certamente, aquela que escolhesse uma página que não fosse referenciada num futuro próximo, porém, o SO não tem como prever se uma página será ou não utilizada novamente.

Os principais algoritmos para relocação de páginas são:

- **Aleatória** – Não utiliza critério algum de seleção. Todas as páginas do **ws** tem igual chance de ser selecionadas inclusive as páginas que são frequentemente referenciadas. Apesar de ser implementada facilmente é muito ineficiente;

- **FIFO** – A página que primeiro foi utilizada será a primeira a ser escolhida para ser substituída. Sua implementação é simples, sendo necessária apenas o uso de uma fila onde as páginas mais antigas estão no início e as mais recentes no final.

  O problema acontece quando páginas que são constantemente referenciadas, como é o caso das páginas de código dos utilitários do sistema, são substituídas devido ao fator tempo e o SO tem que fazer retorná-las novamente para a memória várias vezes;

- **Least Recently Used (LRU)** – Esta estratégia seleciona a página menos recentemente utilizada. Apesar de ser uma boa estratégia é difícil de ser implementada devido ao grande overhead causado pela atualização, em cada página referenciada, do momento do ultimo acesso, além do algoritmo de busca dessas páginas;

- **Not recently used (NUR) –** Escolhe-se a página que não foi recentemente utilizada. Nessa estratégia existe um bit, que permite ao SO a implementação do algoritmo. O **bit de referência** indica quando a página foi referenciada ou não, e está associado a cada entrada da tabela de páginas.

  Inicialmente todas as páginas estão com o bit zerado indicando que não foram referenciadas. À medida que as páginas são referenciadas, o flag associado a cada página é modificado pelo HW. Depois de um certo tempo, é possível saber quais páginas foram referenciadas ou não;

- **Least Frequently Used (LFU) –** Neste esquema, a página menos freqüentemente utilizada será escolhida. Para isso é mantido um contador do número de referências feitas às páginas. A página que tiver o contador com o menor número de referências será a página escolhida. O algoritmo privilegia as páginas que são bastante utilizadas.

  Essa parecer uma boa estratégia, porém, as páginas que entrarem mais recentemente no **ws** serão, justamente, aquelas que estarão com os contadores com o menor valor;

A avaliação de um algoritmo para troca de páginas é feita rodando-o com uma cadeia particular de referências de memória e computando número de **page-faults** que ocorrem durante a execução. A **cadeia de referências** de memória pode ser gerada artificialmente (por um gerador de número aleatórios) ou acompanhando-se a execução de um programa e gravando-se os endereços de cada referência de memória. A ultima opção produz um grande número de dados (um programa referencia milhões de endereços por seg). Para reduzir a quantidade de dados, notamos duas coisas:

- Primeiro, para um tamanho de página dado (que é geralmente fixado pelo hardware do sistema), só precisamos considerar o número da página não o endereço inteiro.

- Segundo, se temos uma referência a página p, então qualquer referencia, que seja imediatamente seguinte, à mesma página p jamais causará page-fault, já que ela já estará na memória.

Para determinar o número de **page-faults** para uma dada **cadeia de referências** e um determinado algoritmo de troca de páginas precisamos também saber o número de frames disponíveis na MP. À medida que este número aumenta, o número de page-faults diminui. No caso de termos uma **cadeia de referências: 1,4,1,6,1,6,1,6,1,6,1**, e existirem 3 ou mais frames, teríamos a ocorrência de 3 page-faults, um para a primeira referência a cada página. Por outro lado, se só houvesse um frame disponível, teríamos uma troca a cada referencia, resultando um total de 11 page-faults. (Explique !)

Para ilustrar os algoritmos, usaremos a seguinte **cadeia de referencias**:

7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 com **3 frames** disponíveis na MP.

6.4.3.1 FIFO

O algoritmo de troca de página mais simples é o **First-In First-Out** (primeiro a entrar, primeiro a sair). Ele associa a cada página o tempo em que ele foi trazido a memória. Quando uma página precisa ser trocada, a mais velha é escolhida. Note que não é estritamente necessário armazenar o tempo em que a página foi trazida, basta organizar uma fila, retirando sempre a página do inicio da fila e inserindo as novas no final da fila. Para a **cadeia de referências** adotada teremos **15 page-faults**. (Mostre !)

O algoritmo FIFO é fácil de implementar, mas a performance deixa a desejar pois a página trocada pode ser a de um modulo de inicialização que não será mais utilizado, ou pode conter uma variável bastante utilizada que foi inicializada há muito tempo, e nesse caso a remoção da página é indesejada.

Note que mesmo que uma página em uso seja selecionada para troca, tudo funciona corretamente. Apenas teremos uma **page-fault** em seguida, para traze-la de volta, então, uma má troca aumenta o número de **page-faults** e aumenta o tempo de execução do programa, mas não causa execução incorreta.

Para ilustrar os problemas possíveis com o FIFO, considere a **cadeia de referencias: 1,2,3,4,1,2,5,1,2,3,4,5** observamos que para o caso em que temos 4 frames disponíveis temos mais **page-faults** (10) que para o caso em que temos 3 frames (9) o que contraria as nossas expectativas. Este resultado é conhecido como **Belady's anomaly** (anomalia de Belady). Esta anomalia reflete o fato de que para alguns algoritmos de troca de páginas o número de **page-faults** pode crescer com o aumento de **frames** disponíveis.

6.4.3.2 Troca Ótima

Um resultado da descoberta da anomalia de **Belady** foi a procura por um algoritmo de **troca de página ótimo**. Tal algoritmo tem a menor taxa de **page-faults** e nunca apresentaria a anomalia de Belady. Este algoritmo existe, e é chamado **OPT** ou **MIN**. Ele diz:

"troque a página que não será usada pelo maior período de tempo".

Esse algoritmo garante a menor taxa possível de page-faults para um número fixo de frames disponíveis. Infelizmente, este algoritmo é difícil de implementar, já que requer um conhecimento futuro da **cadeia de referências** (encontramos situação similar com o algoritmo de escalonamento SJF - "job mais curto primeiro", na seção 5.3.2). Assim, o algoritmo ótimo é usado mais para estudos de comparação.

6.4.3.3 LRU - Least Recently Used

A principal distinção entre o **FIFO** e o **OPT,** além de um olhar para traz e outro para frente no tempo, é que o **FIFO** usa o tempo em que a página foi trazida, **OPT** usa o tempo em que ela será usada. Se usarmos o passado recente como uma aproximação do futuro próximo, trocaríamos a página que não foi usada há mais tempo. Este é o LRU.

A troca LRU associa a cada página o tempo de sua ultima utilização. Quando alguma página precisa ser trocada, **LRU escolhe aquela não usada há mais tempo**, este é o algoritmo OPT olhando para traz e não para frente (de fato, uma técnica comum para gerar o número de page-faults do OPT para uma determinada seqüência de referencia é inverter a cadeia inteira e usar o LRU na cadeia invertida).

O resultado de aplicar LRU na **cadeia de referencia** apresentará um total de **12 page-faults**, contra 15 de FIFO. LRU é bastante usado e é considerado bastante bom, o maior problema é como implementá-lo. Ele pode requerer uma assistência substancial do hardware. o problema é determinar a ordem dos frames definida pelo ultimo uso. Há duas implementações:

Contadores: no caso mais simples, associamos a cada entrada na tabela de páginas um registrador de tempo de uso e adicionamos a UCP um clock lógico ou contador. O clock é incrementado a cada referência de memória. Sempre que uma referencia a uma página é feita, o conteúdo do clock é copiado para o registrador da tabela de página para essa página. Assim, sempre temos o tempo da ultima referencia para a página. Esse esquema requer a procura na tabela de páginas pela página não usada há mais tempo. Os tempos também precisam ser mantidos quando as páginas são trocadas (devido a scheduling da UCP). O overflow do clock também precisa ser considerado.

Pilha: outra aproximação para implementar LRU é manter uma pilha de números de página. Sempre que uma página é referenciada, ela é removida da pilha e colocada no topo. Assim, a parte de baixo da pilha é a página não usada há mais tempo. Como entradas precisam ser retiradas do meio da pilha, a política LRU é mais bem implementada por uma lista duplamente encadeada, com um apontador para o inicio e o fim. Remover uma página e colocá-la no topo requer mudar 6 ponteiros no máximo cada atualização é um pouco mais cara, mas não existe necessidade de uma pesquisa para a troca. Esse modo é particularmente apropriado para implementações por software ou por microcódigo.

Nem OPT nem LRU sofrem a anomalia de Belady. Existe uma classe de algoritmos de troca, chamada de algoritmos de pilha, que não podem exibir a anomalia. Um algoritmo de pilha é aquele em que se pode mostrar que o conjunto de páginas na memória para n frames é sempre o subconjunto das páginas que existiriam se houvessem n+1 frames. Note que nenhuma das implementações LRU seria possível sem assistência de **hardware**.

6.4.3.4 Aproximação LRU

Poucos sistemas têm suporte suficiente de hardware para a verdadeira troca LRU. Alguns não têm nenhum suporte e um outro algoritmo (como FIFO) tem que ser usado. Alguns têm alguma ajuda, na forma de um **bit de referencia**. Este bit é setado pelo hardware, sempre que a uma página é referenciada (leitura ou escrita). Bits de referencia são associados com cada entrada na tabela de páginas, ou como um registrador separado com um bit por quadro. instruções especiais são fornecidas para ler e limpar esses bits.

Inicialmente, todos os bits estão zerados pelo SO. À medida que o programa é executado, o bit associado a cada página que é referenciada é ligado pelo hardware. Depois de um tempo podemos determinar quais páginas foram usadas e quais não foram referenciadas examinando os bits. Não sabemos a ordem de uso, só sabemos que foram usados. Essa informação de ordenação parcial leva a vários algoritmos de troca que tentam aproximar a troca LRU.

6.4.3.4.1 Bits de referencia adicionais

Informações de ordem adicionais podem ser tiradas gravando-se bits de referencia em intervalos regulares. Podemos guardar um byte de 8 bits para cada página em uma tabela na memória. Em intervalos regulares (p ex. 100 ms), uma interrupção do timer transfere controle para o SO, que desloca o bit de referencia de cada página para o bit de maior ordem do byte, deslocando os outros bits p/ direita e descartando o de menor ordem. Esses registradores de deslocamento de 8 bits conterão a historia de utilização das páginas para os últimos 8 períodos.

Uma página com valor no registro de historia de 11000100 foi usada mais recentemente que uma com 01110111. Se interpretarmos o byte como inteiros sem sinal, a página com menor número é a que não é usada há mais tempo, e pode ser trocada. Note que não necessariamente esses números serão únicos, podemos ou trocar todas as páginas com o menor valor ou usar uma seleção FIFO entre elas.

Claro que o número de bits do registrador de historia pode variar, e seria selecionado para fazer a atualização o mais rápido possível. No caso extremo, poderia ser reduzido a zero, restando somente o bit de referencia. Este algoritmo é chamado de algoritmo **segunda chance de troca de página.**

6.4.3.4.2 Troca de segunda chance

O algoritmo básico de troca de segunda chance é o FIFO. Quando uma página é selecionada, no entanto, inspecionamos o bit de referencia. Se for zero, trocamos a página. Se for um, damos a ela uma segunda chance e movemos a seleção para a próxima página FIFO. Quando uma página tem uma segunda chance, seu bit de referencia é zerado e seu tempo de chegada é colocado como o tempo atual. Então, uma página a quem é dada 2a. chance não será trocada ate que todas as demais sejam trocadas (ou ganhem 2a. chance). Alem disso, se uma página é usada o suficiente para manter seu bit como 1, nunca será trocada.

Um modo de ver esse algoritmo é como uma fila circular. Um ponteiro indica qual página deve ser trocada em seguida. Quando um quadro é necessário, o ponteiro avança ate achar uma página com bit de referência zerado. À medida que avança, vai zerando os bits. No pior caso, quando todos estão setados, ele dá a volta completa. Limpa todos os bits antes de selecionar a próxima página para troca. 2a. chance degenera em FIFO quando todos os bits estão setados.

6.4.3.4.3 LFU - Least Frequently Used

Mantém um contador com o número de referencias feitos à página. Aquela com o menor contador é trocada. A motivação é que uma página usada ativamente terá um contador alto. Ele sofre na situação em que uma página é usada intensivamente no inicio do programa e então não é mais usada, já que seu contador permanecera alto e ela ficara um tempo na memória. Uma solução é deslocar os contadores para a direita 1 bit a intervalos regulares, formando um contador de usos com média de decaimento exponencial.

6.4.3.4.4 MFU - Most Frequently Used

Argumenta que a página com o menor contador foi provavelmente recém trazida e ainda será usada. Nem MFU nem LFU são muito usadas. Suas implementações são bastante caras.

6.4.3.4.5 Classes de páginas

Há vários outros algoritmos de troca. Por ex., se consideramos o bit de referencia e o bit de alteração (dirty bit) como um par ordenado, teremos as seguintes classes:

0. (0,0) não usado nem sujo;
1. (0,1) não usado recentemente mas sujo;
2. (1,0) usado mas limpo;
3. (1,1) usado e sujo;

Quando uma troca é necessária, cada página esta em uma dessas classes. Trocamos qualquer página na classe não vazia mais baixa. Se houverem varias, usamos FIFO ou escolha aleatória para escolher uma entre elas.

6.4.3.5 Algoritmos Ad Hoc

Outros procedimentos são usados freqüentemente em adição a um algoritmo de troca especifico. p ex., SOs mantém um pool de frames livres. quando há page-fault, um quadro vitima é escolhido como antes. No entanto, a página desejada é lida para um frame livre do pool antes da vitima sair. Isso faz com que o programa recomece mais rápido, sem esperar a página vitima ser retirada. Quando ela é retirada posteriormente, seu quadro vai para o pool de livres.

Uma expansão dessa idéia mantém uma lista de páginas sujas. Quando o dispositivo de paginação esta disponível, a página suja é selecionada e escrita na área de copia, seu bit sujo é então resetado. Esse esquema aumenta a probabilidade de uma página ser limpa ao ser selecionada, e não será necessário retira-la.

Outra modificação é manter o pool de quadros livres mas manter a informação de que página estava no quadro. Como o conteúdo do quadro não é modificado copiando-se o quadro para a área de copia, a página antiga pode ser reusada diretamente do quadro do pool se for necessária antes do quadro ser reutilizado. Nenhum I/O é necessário nesse caso. Quando há page-fault, primeiro checamos se a página desejada esta no pool. Se não, selecionamos um frame livre e escrevemos nele.

Esta técnica é usada no sistema VAX/VMS, junto com o algoritmo FIFO. Quando o FIFO troca por engano uma página que ainda esta em uso, ela é rapidamente recuperada do buffer de quadros livres e não é necessário I/O. o buffer de frames livres oferece.

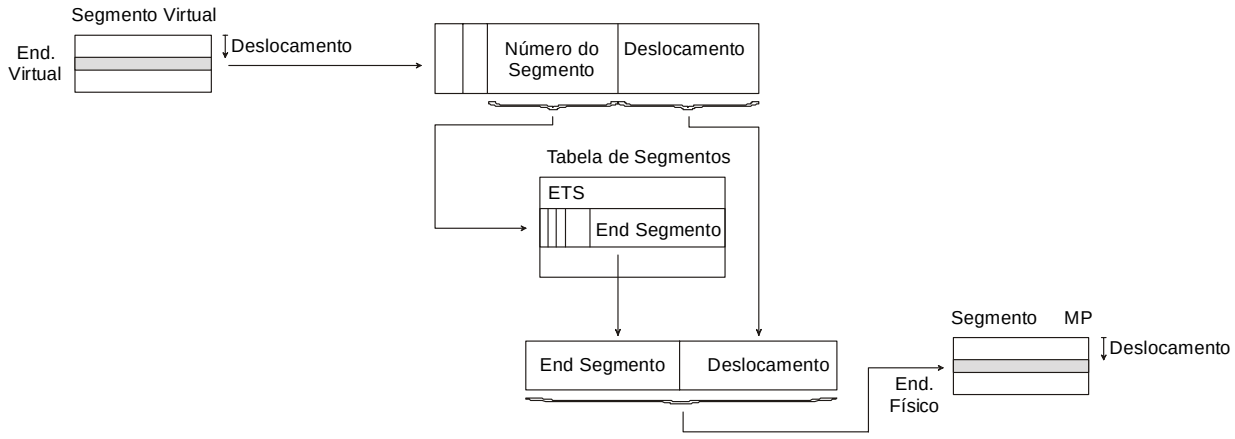
6.5 Segmentação

Memória virtual também pode ser implementada em sistemas segmentados. Muitos sistemas como o Multics (precursor do UNIX) implementam um mecanismo de segmentação com paginação, onde os segmentos são divididos em páginas.

Segmentação é a técnica de gerência de MP, onde os programas são divididos logicamente em blocos (segmento de dados, código, stack). Estes blocos têm tamanhos diferentes e são chamados segmentos, cada um com seu próprio espaço de endereçamento.

A diferença em relação à paginação é que a segmentação permite uma relação entre a lógica do programa e sua divisão na MP (porque?)

O mecanismo de mapeamento é semelhante ao de paginação. Os segmentos são mapeados através de tabelas de mapeamento e os endereços são compostos pelo número do segmento e um deslocamento dentro do segmento.



O SO mantém uma tabela com as áreas livres e ocupadas da MP. Quando um processo é carregado para MP, o SO localiza um espaço livre que o acomode. As estratégias para escolha da área livre podem ser as mesmas do **MVT (Alocação Particionada Dinâmica)**, ou seja, **BEST-FIT, WORST-FIT, FIRST-FIT**.

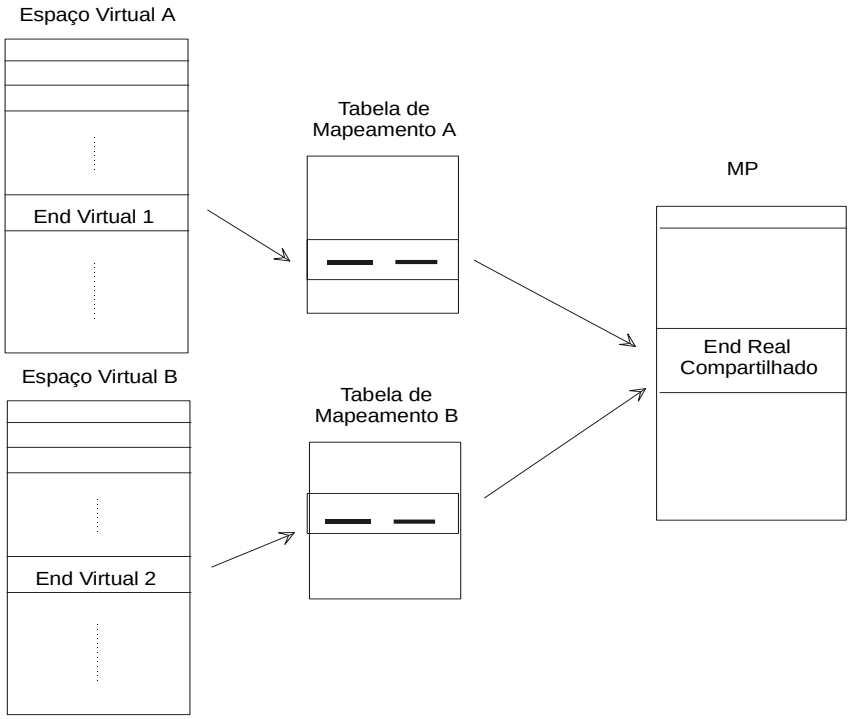
Na segmentação, apenas os segmentos referenciados são transferidos da MS para MP. Logo, para ser mais eficiente, os programas devem estar bem modularizados.

O problema de fragmentação também ocorre nesse modelo, quando as áreas livres são tão pequenas, que não acomodam nenhum segmento que necessite ser carregado.

6.6 Compartilhamento de Memória

Em sistemas multiprogramáveis, é comum usuários utilizarem certos programas simultaneamente (**código reentrante**), o que evita que várias cópias de um mesmo programa ocupem a memória desnecessariamente. Um bom exemplo desse tipo de aplicação são os utilitários do sistema, como compiladores, editores de texto ou, mesmo, algumas aplicações de usuários.

Em sistemas que implementam **memória virtual**, é bastante simples o **compartilhamento de código** e dados entre vários processos. Para isso, basta que as entradas das páginas/segmentos apontem para os mesmos frames/segmentos na memória principal. Dessa forma, é possível reduzir-se o número de programas na memória e aumentar o número de usuários compartilhando o mesmo recurso conforme mostra a figura abaixo.



A vantagem da **segmentação** em relação à **paginação**, no aspecto do compartilhamento, baseia-se na forma em que os programas são divididos. Como as tabelas de segmentos mapeiam estruturas lógicas (área de códigos, área de dados, etc) o compartilhamento de segmentos é mais simples de implementar do que o de páginas.

Outra vantagem da **segmentação** está no compartilhamento de estruturas de dados dinâmicas, ou seja, estruturas cujo tamanho varia durante a execução do programa (através do uso das rotinas de alocação dinâmica das linguagens). Enquanto na **paginação** o crescimento de um vetor, por exemplo, implica a alocação de novas páginas e, consequentemente, o ajuste das tabelas de mapeamento, na **segmentação**, as tabelas devem ter ajustado apenas o tamanho do segmento.

6.7 Trashing

Trashing pode ser definido como sendo a excessiva transferência de páginas/segmentos entre a MP e a MS, presentes em **sistemas paginados** ou **segmentados**. Na **paginação**, o **Trashing** pode ocorrer em dois níveis: processo e sistema.

- **processo** - elevado número de page-faults gerado pelo programa em execução. Os principais motivos para tal fato são:

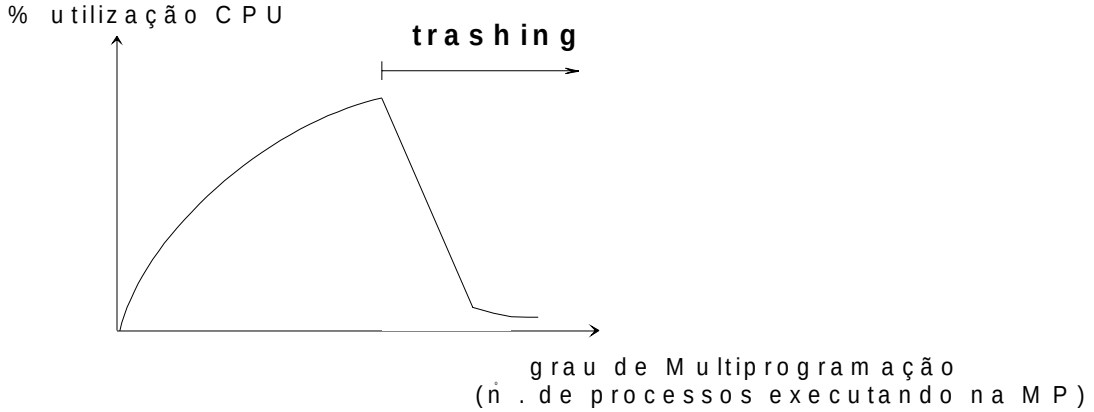
  (a) mau dimensionamento do tamanho do **ws** (**wss**) sendo pequeno demais para acomodar as páginas constantemente referenciadas pelo programa. (b) programas mal-estruturados onde o princípio da localidade não se aplica.

- **sistema** - quando existem mais processos competindo pela MP que espaço disponível. Nesse caso o SO tenta administrar a memória de forma que todos os processos sejam atendidos. O primeiro passo é a redução do **ws** dos processos, o que acaba acarretando o problema de **trashing** em nível de processo. Caso a redução do **ws** não resolva, o SO começa o trabalho de **swapping** para liberar espaço na MP. Se esse mecanismo for levado ao extremo, o SO passará mais tempo fazendo swapping que executando os processos.

O **Trashing** em sistemas com **segmentação** também ocorre em dois níveis. Em nível do processo, a transferência de segmentos é excessiva devido à modularização extrema do programa, não seguindo o conceito da localidade. Em nível do sistema, o **Trashing** em **segmentação** é bastante semelhante ao da **paginação** com a ocorrência de **swapping.** Nos dois caso, o problema da transferência de páginas ou segmentos dos processos entre a MP e a MS pode se tornar crítico.

De qualquer forma, se existem mais processos para serem executados que a memória real disponível, a única solução é expandir a MP. É importante ressaltar que este problema não ocorre apenas em sistemas que implementam memória virtual, mas também em sistemas com outros mecanismos de gerência de memória.

O fenômeno de Trashing pode ser visualizado em um gráfico que indique o comportamento da UCP (%utilização de UCP) contra o número de processos executando em memória (Grau de Multiprogramação).



Exercício: Explique o gráfico acima, sabendo-se que:

grau de Multiprogramação = 1 - p^n

onde: p = probabilidade média de que um determinado processo execute alguma operação de IO

n = número de processos residentes na MP

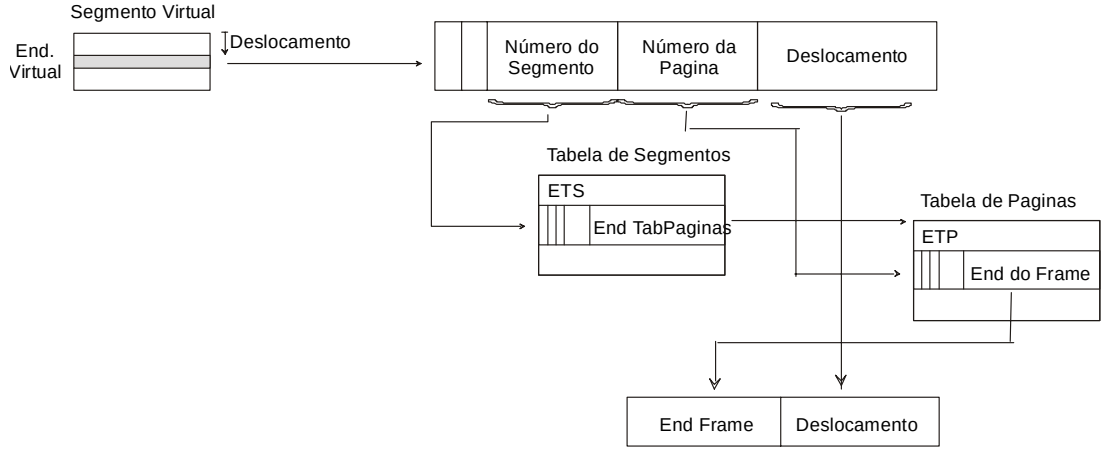
(a) Explique o porque da fórmula
(b) Explique o comportamento do gráfico acima

6.8 Sistemas Combinados

Paginação e segmentação têm suas vantagens e desvantagens. Também é possível combinar esses dois esquemas para melhorar os dois. Isso é mais bem ilustrado por dois sistemas específicos: o IBM 360/67(e posteriormente IBM 370) e o GE 645 para Multics.

6.8.1 Segmentação com Paginação

Sistemas que implementam **Segmentação com Paginação** permitem a divisão lógica dos programas em segmentos e, por sua vez, cada segmento é dividido, fisicamente, em páginas. Neste sistema, um endereço é formado pelo **número do segmento**, um número de página dentro do segmento e um deslocamento e dentro de uma página.



O sistema Multics enfrentava um problema diferente. Endereços lógicos eram formados de um número de segmento de 18 bits e 16 de deslocamento. Embora o esquema criasse um espaço de endereçamento de 34 bits, o trabalho adicional é bem tolerável já que o número variável de segmentos naturalmente implica em um Registrador de Comprimento da Tabela de Segmentos. Só precisamos de tantas entradas na tabela de segmentos quantos forem os nossos segmentos. Não teremos entradas vazias.

No entanto, com segmentos de palavras de 64K, o tamanho médio do segmento poderia ser bastante grande e fragmentação externa seria um problema. Mesmo que fragmentação externa não seja problema, o tempo de pesquisa para alocar um segmento, usando first-fit (primeiro a servir) ou best-fit (que serve melhor), poderia ser longo. Poderíamos gastar memória por fragmentação externa ou tempo devido a pesquisas longas, ou ambos.

A solução adotada foi paginar os segmentos. Paginação elimina fragmentação externa e torna trivial o problema de alocução: quadro vazio pode ser usado para a página desejada. O resultado é mostrado em 5.33. Note que a diferença entre essa solução e segmentação pura é que a entrada da tabela de segmentos não contém o endereço de base do segmento, mas o endereço base da tabela de página para o segmento. O deslocamento do segmento é então quebrado em um número de página de 6 bits e deslocamento de 16. O número de página serve de índice na tabela de página para dar o número do quadro. finalmente, o número de quadro é combinado com o deslocamento da página para formar o endereço físico.

Devemos ter uma tabela de página separada para cada segmento. Porem, como cada segmentação e limitado em comprimento por sua entrada na tabela de segmentos, a tabela de página não precisa ter todo o tamanho. Só terá as entradas realmente desejadas. Também, a ultima página de cada segmentação em geral não estará completa. Teremos, em média, metade de uma página de fragmentação interna por segmento. Conseqüentemente, embora tenhamos eliminado fragmentação externa, introduzimos fragmentação interna e aumentamos o custo adicional do espaço da tabela.

Outros sistemas usam segmentação paginada. Prime 500 usa um número de segmentação de 12 bits e uma palavra de deslocamento dividida em número de página de 6 bits e 10 de deslocamento. O RCA Spectra 70/46 tinha número de segmentação de 5 bits e 18 de deslocamento, composto de número de página de 6 bits e 12 de deslocamento.

A verdade é que mesmo a segmentação paginada da multics apresentada é simplista. Como o número de segmento é de 18 bits, teríamos ate 262144 segmentos, requerendo uma tabela de segmentos muito grande. Para solucionar, a multics pagina a tabela de segmentos. em geral, o endereço da multics usa o número de segmentação p/ definir um índice na tabela de página para a tabela de segmentos. Desta entrada, ela localiza a parte da tabela de seguimentos com a entrada para este segmento. A entrada da tabela de página aponta para a tabela de página para este segmento, que especifica o quadro contendo a palavra desejada.

6.8.2 Paginação segmentada

O IBM 360/67 era um sistema paginado. O endereço básico de um byte de 24 bits era dividido em um número de página de 12 bits e um deslocamento de 12 bits. Cada entrada da tabela de páginas era de 2 bytes (meia palavra) contendo um número de quadro de 12 bits e um bit válido/inválido. O problema era que um número de página de 12 bits permitia 4096 páginas, precisando de 8K bytes para cada tabela de pág. Também era desejado expandir o endereço lógico para 32 bits com um endereço de página de 20 bits. Isso requereria uma tabela de página com 1048576 entradas, ou 2097152 bytes.

Com o espaço de endereçamento maior, a maior parte da tabela de página iria ficar vazia, já que muitos programas usariam apenas uma fração do total do espaço do endereçamento possível. Então, a tabela de página era segmentada. Os 4 bits superiores do número de página são considerados o número de segmento, que é usado para selecionar uma de 16 entradas da tabela de segmentos. Cada segmento pode ter até 268435456 bytes de comprimento, sendo múltiplo de 4096 bytes. A entrada da tabela de segmentos aponta para a base da tabela de páginas para este segmento, e também indica o comprimento da tabela de página. Assim, seções grandes da tabela de páginas que estavam zeradas poderiam ser agrupadas setando o endereço da tabela de páginas para zero.

Este esquema agora requer, no pior caso, 3 acessos de memória por referencia de memória desejada. Um conjunto de 8 registradores associativo reduz o trabalho adicional para apenas 150 nanossegundos mais lento que uma referencia não mapeada quando um acerto (hit) acontece.

O conceito importante aqui é que o usuário ainda vê um espaço linear de endereço paginado. Um resultado é que é possível incrementar um índice e produzir primeiro a ultima posição do segmento i e depois a primeira posição do segmento i+1; a memória é ainda basicamente linear.

7 Entrada e Saída

7.1 Conceitos de Operação de Entrada e Saída

Nas primeiras gerações de computadores o programador necessitava efetuar todos os procedimentos para o controle físico das operações de entrada e saída. Para isto precisava conhecer cada detalhe do hardware associado à estas operações. Este tipo de atividade tomava grande parte do esforço necessário ao desenvolvimento dos programas.

Se o programa fosse executar em outro sistema, que quase sempre tinha endereços de dispositivos diferentes ou mesmo com ligeiras alterações no funcionamento físico, ele precisava reescrever todo o trecho do programa que efetuava as operações de entrada e saída.

Sendo estas operações fundamentais, a evolução dos Sistemas Operacionais passou a incorporá-las como funções próprias destes sistemas.

Assim sendo o controle dos dispositivos de entrada e saída passou a ser um dos componentes principais de um Sistema Operacional. Quando um dispositivo emite uma interrupção o seu manuseio tem de ser feito pelo S.O.. Igualmente, quando um processo inicia uma operação de E/S, normalmente via um SVC, quem deve emitir o comando para o dispositivo adequado é o Sistema Operacional e, ele também, deve controlar o sucesso ou não da operação, algumas vezes tentando a repetição da operação em caso de erro, retornando a informação ao processo solicitante.

Então a parte do S.O. responsável por esta função tem que fornecer uma interface entre os diferentes dispositivos e o resto do sistema de modo que seja simples e fácil o seu uso.

Esta interface tem que funcionar de forma que um determinado comando seja executado do mesmo modo, independente do dispositivo sendo endereçado. A esta característica dá-se o nome de independência do dispositivo (device independence).

Os componentes físicos dos dispositivos podem até variar (e, certamente, variam) porém o programador está convenientemente isolado desta variação pelas funções de entrada e saída do S.O.. Não importa assim o tipo de circuitos eletrônicos, cabeças de leitura e gravação, cabeamentos fontes de energia, meios físicos de representação dos dados, motores e todo o resto da parafernália que compõem o dispositivo fisicamente.

7.2 Categorias de Dispositivos de E/S

De um modo geral podemos distinguir duas categorias básicas de dispositivos em função do modo como eles operam:

- dispositivos em blocos ou rajadas (block devices)
- dispositivos em modalidade caracter (character devices)

Os dispositivos em modalidade caracter operam com uma cadeia de dados, passando-os ou recebendo-os, byte a byte, sem se preocupar com uma estrutura organizada destes dados. Normalmente esta categoria de dispositivos utiliza meios não magnéticos para o registro das informações. Adicionalmente não permitem o endereçamento dos dados e também não permitem operações de busca (seek). Grande parte destes dispositivos caíram em desuso atualmente. Como exemplo de dispositivos orientados a caracter podemos citar: leitora e perfuradora de cartões, leitora e perfuradora de fita de papel, impressoras, terminais, mouse, teclados e plotadores.

Os dispositivos em bloco, como o nome indica, agrupam os dados em uma estrutura denominada bloco de dados. Estes blocos tem, normalmente, um tamanho equivalente a uma potência na base binária: 64, 128, 256, 512, ... bytes. Dentro do bloco são colocados os registros lógicos de dados e por isso o bloco é também chamado de registro físico.

Uma característica fundamental do dispositivo em bloco é ser permitido acessar um bloco, para leitura ou gravação, independentemente dos demais. Exemplos típicos dos dispositivos em bloco são os discos e as fitas magnéticas.

Um dispositivo fora destes dois esquemas é o relógio (clock) pois ele não é endereçável a bloco, não aceita cadeia de dados e apenas pode ser ativado ou desativado, gerando como saída interrupções. Os discos e o relógio serão objetos de estudo mais detalhado no Capítulo 7.

7.3 Controladores de Dispositivos

Os dispositivos podem ser vistos como compostos de duas partes principais, a saber: componentes eletrônicos e componentes mecânicos. Os componentes eletrônicos tem como função emitir os sinais elétricos que irão ativar os componentes mecânicos (motores, cabeçotes, braços e outros).

Os componentes eletrônicos podem se apresentar de diversas formas, isolados ou integrados no computador. São conhecidos por controladoras de dispositivos ou adaptadores.

Em computadores de pequeno porte ou microcomputadores as controladoras são placas que se encaixam nos conectores de ranhuras da placa-mãe (slots), como a fig. 5.1A mostra. Elas são de diversos tipos, como controladoras de vídeo (CGA, EGA, VGA, SVGA), de discos rígidos, flexíveis ou CD-ROM, controladoras de rede e outras.

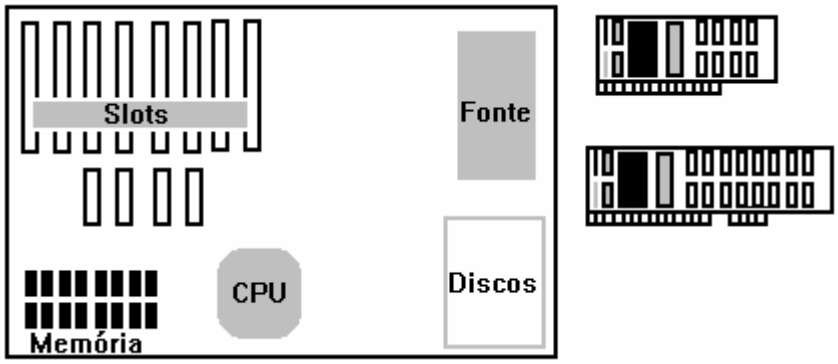


Fig. 7.1A - Exemplo de placa de CPU com os slots e placas controladoras.

Os computadores de grande porte costumam ter controladoras agrupando dispositivos de um mesmo tipo, os quais conectam-se a elas por cabos. Algumas destas são: controladoras de unidades de discos, controladoras de unidades de fitas magnéticas e controladoras de telecomunicações. A fig. 7.1B mostra esta ligação.

Nos dois casos existem conexões entre a CPU e as controladoras e entre estas e os dispositivos sobre seu controle. Esta conexão é feita por cabos especiais com diversos fios, por onde são transmitidos impulsos elétricos que configuram os dados, endereços e sinais de controle necessários.

Nas primeiras gerações cada fabricante estabelecia o padrão de ligação para seus equipamentos o que impedia que controladoras ou periféricos de um fabricante pudessem ser utilizados por equipamentos de outra marca.

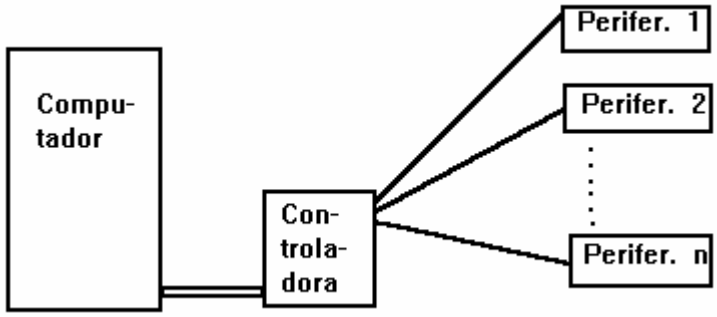


Fig. 7.1B - Ligação da controladora externa ao computador e aos periféricos.

A partir da terceira geração começou a haver um entendimento no mercado da necessidade de serem estabelecidos padrões mais genéricos de forma que permitissem uma maior flexibilidade de uso dos equipamentos.

Começaram a surgir normas, definidas por entidades oficiais, como a ISO (International Standards Organization), IEEE (Institute of Eletrical and Eletronic Enginnering), ANSI (American National Standards Institute), apesar da reação de alguns grandes fabricantes, principalmente a IBM, que temiam perder sua fatia de mercado. Apesar desta reação a realidade foi que os consumidores e pequenos fabricantes (na época) começaram a adotar tais padrões.

Outros fabricantes mais audaciosos começaram a produzir periféricos e controladoras que se conectavam aos grandes fabricantes da época (IBM, Burroughs, Univac e Digital) com custo muito inferior ao cobrado por estes e apesar dos diversos problemas enfrentados conseguiram romper o clube fechado dos grandes, possibilitando ao usuário ter novas alternativas para expansão de seus sistemas.

7.4 Comunicação entre o Sistema Operacional e Controladores de Dispositivos

Com o uso das controladoras a comunicação do CPU não era mais feita com o periférico diretamente, e sim com estas unidades. Para isto o sistema operacional enviava um comando para a controladora e esta se encarregava de executar o controle do periférico, liberando a CPU para as atividades de processamento.

A partir do comando recebido do sistema operacional a controladora seleciona o periférico adequado, comandando e controlando a realização física da operação. Ao terminar a operação, quando necessitar da atenção do S.O. ela emite um sinal de interrupção para o S.O. , que irá atender tão logo seja possível.

Para receber comandos e dados da CPU as controladoras possuem registradores cujos endereços são selecionados quando o S.O. deseja efetuar operações de entrada ou saída de dados. A seqüência habitual é a CPU endereçar um ou mais registradores de controle para:

- conhecer o estado (status) da controladora e dos dispositivos ligados à ela
- enviar um ou mais comandos operacionais

A seguir é feito um acesso aos registradores de dados ou buffers da controladora para escrever ou ler dados. A figura 7.2 exemplifica esta ligação.

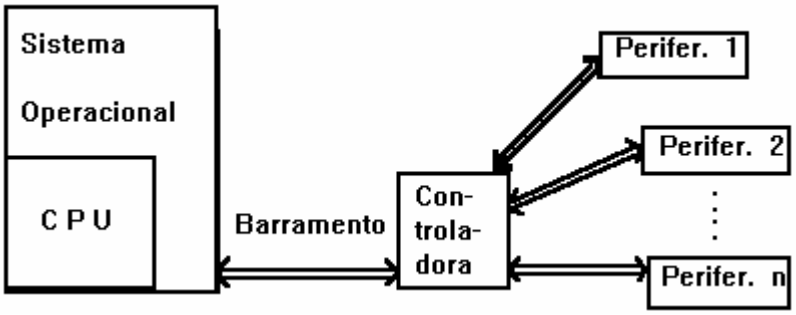


Fig. 7.2 - Comunicação entre o Sistema Operacional e a controladora.

Quando a controladora precisa da atenção da CPU ela vai enviar uma interrupção para a sua entrada específica no Vetor de Interrupções do Sistema Operacional. Assim pode-se resumir toda a comunicação entre S.O e controladora, nas seguintes etapas:

a) Operação de Saída de Dados

Quando o S.O. possui dados para enviar ele lê no registrador de controle o status da controladora. Estando ela ready (pronta) ele envia um ou mais comandos e a seguir os dados, se liberando para realizar outras atividades. Se a controladora não estiver disponível ele continua suas operações testando a sua disponibilidade após um período de tempo determinado.

b) Operação de Entrada de Dados

Ao receber uma interrupção, que fica indicada no vetor de interrupções, o S.O. envia um comando para a controladora avisando que está pronto para receber os dados. Dependendo do tipo da controladora ele irá ler estes dados ou eles serão transferidos diretamente para a memória principal pelo DMA (Direct Memory Access), sendo esta modalidade usada principalmente pelos dispositivos que trabalham com blocos de dados.

O quadro a seguir mostra o endereço dos registradores e das entradas no vetor de interrupções para os microcomputadores da linha IBM-PC.

Dispositivo	End. Reg. E/S	Ent. Vet. Int.
Porta Serial 1 (COM1)	3F8-3FF	12
Impressora Paralela 1 (LPT1)	378-37F	15
Teclado (CON:)	060-063	9
Disco Rígido (C:)	320-32F	13

7.5 Características Básicas do Software de Entrada e Saída

Os softwares de E/S para operações com os dispositivos periféricos devem ser projetados de forma que a independência de dispositivo seja um ponto básico, garantindo a flexibilidade de sua adaptação, ainda que se mude o dispositivo. Como exemplo se o usuário emitir o comando:

sort < entrada > saída

Será função do S.O. descobrir quais gerenciadores de dispositivo irá utilizar. O comando sort quer receber dados de um arquivo associado a um dispositivo de entrada e enviar o seu resultado (dados de saída) para outro arquivo. Tanto faz que estes dispositivos sejam discos rígidos, disquetes, fitas magnéticas ou qualquer outro.

Para possibilitar esta independência de dispositivo os nomes dos arquivos devem ter uma denominação uniforme, que pode ser uma cadeia de caracteres ou um número inteiro numa faixa determinada, não sendo de alguma forma dependente do dispositivo onde ele está contido.

Adicionalmente o software de E/S deve tentar efetuar a correção de erros (error handling) e esta tarefa deve ser realizada no nível mais baixo possível, ou seja, mais próximo do hardware. Outras características devem ser levadas em conta, como:

- tipo de transferência de dados: síncrona (blocados) ou assíncrona (por interrupções);

- modalidade de operação do dispositivo: dedicado ou compartilhado. O uso de dispositivos dedicados podem gerar diversos problemas, como, por exemplo, os deadlocks.

7.6 Estrutura de Camadas do Software de Entrada e Saída

Uma preocupação fundamental na escrita de um software de entrada ou saída diz respeito aos aspectos citados e uma maneira eficiente de desenvolvê-lo e escrever o software em camadas cada uma com a sua função específica, o que adotado pela maioria dos S.Os. atuais. Este enfoque tanto facilita o seu desenvolvimento como a sua depuração (debug) e manutenção. Em princípio podemos dividir estas camadas em quatro, a saber:

a) Manipuladora de Interrupções (Interruption Handlers)
b) Gerenciadora de Dispositivos (Device Drivers)
c) Software de S.O. Independente de Dispositivo (Device Independent Operating System Software)
d) Software em Nível de Usuário (Application Software)

A seguir vamos analisar cada uma das camadas identificando as atividades pertinentes a elas. A figura 7.3 fornece uma visão da estrutura em camadas do UNIX.

7.6.1 Camada Manipuladora de Interrupções

Uma interrupção ocorre, tipicamente, após ser emitido um comando de E/S, causando um bloqueio do dispositivo associado ao comando e em resposta a este comando.

Ao acontecer a interrupção a rotina de manuseio irá efetuar os procedimentos necessários, de forma a remover o bloqueio do dispositivo. O modo como isto é feito varia de um sistema para outro, podendo ser um SIGNAL em uma variável de condição de um monitor, um UP em um semáforo ou mesmo um send de uma mensagem para o processo bloqueado. O efeito é fazer com que o processo bloqueado fique novamente pronto para executar.

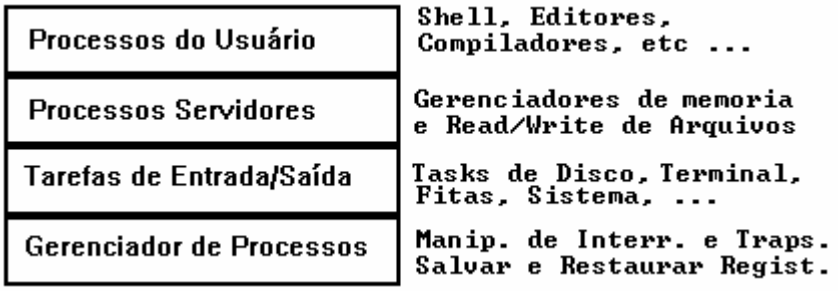


Fig. 7.3 - Estrutura em camadas do software em um sistema UNIX.

7.6.2 Gerenciadores de Dispositivos

Este componente tem como função conter a programação dependente de dispositivo, existindo um gerenciador para cada dispositivo ou grupo de dispositivos semelhantes. Ele envia comandos para os registradores do controlador de dispositivo e testa se a sua execução foi efetuada corretamente.

O Gerenciador de Dispositivo completa a ponte que liga a abstração, ou seja, a emissão de um comando por uma aplicação em nível mais alto, passando pelo software independente de dispositivo, à realização física da operação, que é o comando enviado ao controlador do periférico. Com isto, apenas, ele como componente do S.O., precisa conhecer a estrutura física do controlador, isto é, os registradores e o seu uso e características, como: se possuem memória RAM para buferização ou não, o tipo de sinal usado para atualizar o seu estado e as demais informações necessárias ao funcionamento do controlador.

Depois de determinar quais comandos irá emitir para o controlador o gerenciador de dispositivo envia-os para os registradores de controle do dispositivo. Note-se que enquanto alguns controladores só aceitam um comando de cada vez outros aceitam uma lista de comandos (que são executados por ele sem ajuda do S.O.).

Como conseqüência do(s) comando(s) emitidos podem ocorrer duas situações:

a) Como o gerenciador de dispositivo deve esperar até que o controlador realize algum trabalho para ele, então ele se bloqueia até a chegada de uma interrupção que o desbloqueie.

b) A operação termina sem demora, de modo que o dispositivo não será bloqueado, como, por exemplo no caso de uma exibição no vídeo.

Em qualquer caso ele deve verificar se a operação terminou sem erro e, em caso positivo, ele irá passar os dados (se a operação for de entrada) e as informações de status. Havendo erro passará só informações de status. Havendo operações pendentes e enfileiradas o dispositivo irá executá-las e em caso contrário ele entra em estado de espera.

7.6.3 Software de Entrada e Saída Independente de Dispositivo

Algumas atividades realizadas pelo software independente de dispositivo são:

Interfaceamento uniforme para os device drivers
Denominação dos dispositivos
Proteção dos dispositivos
Fornecimento, quando for o caso, de um tamanho de bloco independente do dispositivo
Buferização dos dados
Reserva de memória nos dispositivos operando com blocos de dados
Reserva e liberação dos dispositivos dedicados
Indicação de erros

Cada sistema operacional, em função de suas características peculiares, irá determinar qual a parte do software específica e qual a parte do software independente do dispositivo. Apesar do ideal ser uma total independência, uma pequena quantidade de funções, principalmente por razões de eficiência, são realizadas fisicamente nos dispositivos.

A independência permite a uniformidade, tanto para as funções de entrada e saída, como em relação ao interfaceamento com os softwares em nível de aplicação.

A denominação permite um mapeamento de nomes simbólicos para os dispositivos apropriados.

A proteção significa não permitir o acesso aos dispositivos para a realização de operações que não sejam permitidas. Ela está fortemente associada à denominação do dispositivo. Para isso existem indicadores das operações possíveis acrescentados ao dispositivo e adicionalmente, podem existir outros esquemas de proteção, por software, restringindo o acesso, por exemplo, a determinados usuários ou grupos de usuários.

O tamanho do bloco deve ser uma das preocupações, principalmente no caso dos discos. Isto porque, mesmo trabalhando com discos com registros físicos de diferentes tamanhos, o usuário não deve precisar de se preocupar com este detalhe. Esta tarefa cabe ao software de E/S, que deverá proceder a uniformização, que separando, quer grupando, os blocos de dados recebidos ou enviados, ao tamanho adequado ao dispositivo usado.

A buferização irá permitir grupar os dados de forma a permitir um funcionamento "suave" do sistema. No caso de saída para dispositivos em bloco irá esperar até que os dados formem um bloco completo para gravá-los ou quando operando com dispositivos caracter esperar até que eles tenham capacidade para dar vazão à saída. Em caso de entrada de dados o procedimento é inverso, ou seja, quando bloco, após lido, irá passar os dados em uma seqüência de bytes para o processo requisitante ou, quando caracter, irá juntando bytes para permitir ao processo solicitante pegá-los à sua conveniência (por exemplo, no caso do teclado).

A reserva de memória nos dispositivos em bloco tem a ver com o fato de que ao ser iniciada uma operação de saída o sistema muitas vezes já sabe quantos blocos irá precisar no disco e, assim, pode reservar o seu espaço. Isto evita, por exemplo, que ele comece uma operação de cópia quando sabe que o espaço não será suficiente. Em outros casos quando não é possível prever a necessidade poderá reservar um espaço padrão definido "a priori".

A reserva de dispositivos e a sua liberação tem relação com o fato que muitos dispositivos não permitem o compartilhamemento (por exemplo: unidades de fita e impressora) e, por isso, ao ser definido que eles serão usados, o software de E/S deve efetuar a sua reserva para o processo solicitante, só os liberando quando o solicitante indicar, de algum modo, que não irá mais usá-los. Um modo simples de manipular tais requisitos é fazer com que sejam realizados OPENs em arquivos especiais para os dispositivos acessados diretamente. Se o OPEN falhar isto indica que o dispositivo não está disponível. Para liberar estes dispositivos dedicados efetua-se um CLOSE.

A manipulação de erros é, fundamentalmente, feita pelos gerenciadores de dispositivos, porque estando a maioria dos erros relacionadas ao funcionamento do dispositivo é função do seu gerenciador saber o que fazer. Os erros podem ser provocados por uma série de causas, tanto as que são realmente problemas físicos como os problemas operacionais (ligados ao uso dos dispositivos pelos seus operadores) ou os relacionados com a estrutura do software usada no seu controle.

Como exemplo de um problema físico podemos citar um bloco de dados no disco danificado. Neste caso o gerenciador de dispositivo irá tentar refazer e leitura um determinado número de vezes e caso não consiga irá informar ao software independente de dispositivo. Por outro lado a falta de disco na unidade, a tentativa de gravar um disco protegido ou a falta de papel na impressora são problemas operacionais. Em qualquer caso será enviada uma mensagem as camadas superiores de software que se encarregarão de indicar o problema ao usuário.

Quando o problema está relacionado à estrutura do software de controle a situação torna-se mais crítica, podendo provocar a queda ou inoperância do sistema operacional. Como exemplos, um erro no registro de boot ou na tabela de alocação de páginas do disco (se for o disco principal do sistema este irá exibir uma mensagem de erro e parar).

7.6.4 Software de Entrada e Saída em Nível de Usuário

Este componente abrange duas classes distintas, que são:

- Bibliotecas de Procedimentos de Entrada e Saída (Library Procedures)
- Sistema de Armazenamento de Dados de Entrada ou Saída (Spooling)

Os procedimentos de Entrada e Saída são usados para funções especiais como a formatação dos dados lidos ou gravados ou para conversão de tipos de dados. Para isto sistemas como o Unix possuem uma biblioteca de procedimentos padrões de E/S (Standard I/O Library). Esta rotinas são ligadas (LINK) aos programas dos usuários para possibilitar sua execução. Como exemplo, no C, temos a stdio.h com função com a printf e atoi.

O outro componente, spooling, não é usado apenas para as saídas impressas como concebido originalmente porém como uma forma de permitir ao sistema em multiprocessamento trabalhar com dispositivos dedicados. A intenção é evitar que um processo tome conta de um dispositivo que ele abriria e, em muitos casos, ficaria longos períodos sem usar.

Para implementar a técnica de spooling existem dois componentes principais que são uma área de spooling, que é uma estrutura em forma de diretório (spooling directory) e um processo especial chamado de daemon que irá acessar os arquivos no diretório de spool.

Na impressão, que é o exemplo clássico, quando algum processo quer efetuar uma saída ele faz o acesso no diretório de spooling de impressão onde é criada uma entrada associada aos dados que o processo quer enviar para a impressora. Assim que o processo indicar o fim de arquivo de impressão este é fechado ficando habilitada a sua saída para a impressora. O daemon de impressão, por sua vez, irá acessar o diretório de spooling e, entre as entradas fechadas, escolherá de algum modo a que ele irá dar saída. Após imprimir todos os dados relativos àquela entrada no diretório ele marca a entrada como livre. Note-se que apenas o daemon terá acesso para leitura do diretório de spool e só ele bloqueia a impressora.

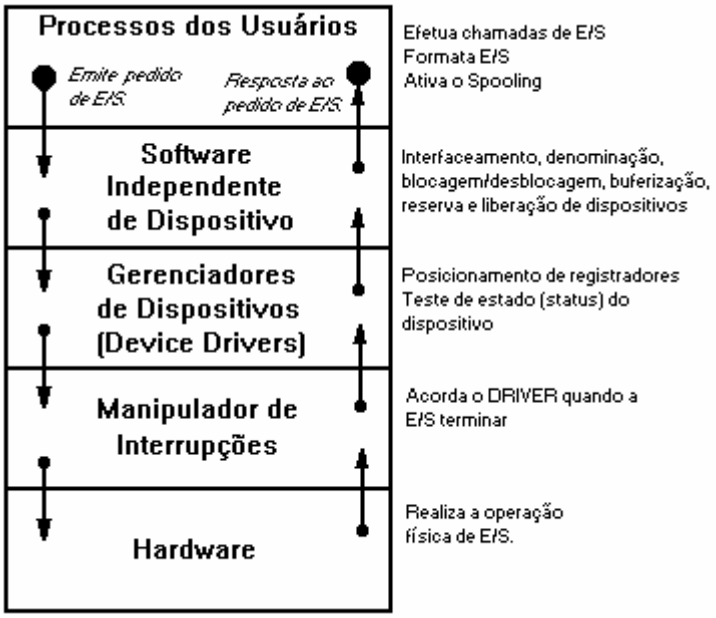


Fig. 7.4 - Camadas do sistema de entrada e saída, suas ligações e resumo das principais funções de cada camada.

Como mencionado a técnica de spooling é usada nos sistemas atuais para outras finalidades além da impressão, como a transferência de arquivos através de redes de computadores ou no correio eletrônico. Note-se que a maioria das máquinas só tem uma porta de comunicação com a rede, a qual pode ser vista como um dispositivo dedicado.

As camadas do sistema de entrada e saída, suas ligações e um resumo das principais funções de cada camada são mostradas na figura 7.4.

8 Gerência do Sistema de Arquivos

8.1 Arquivos

8.1.1 Organização de Arquivos

8.1.2 Métodos de Acesso

8.1.3 Operações de E/S

8.1.4 Atributos

8.2 Diretórios

8.3 Alocação de Espaço em Disco

8.3.1 Alocação Contígua

8.3.2 Alocação Encadeada

8.3.3 Alocação Indexada

8.4 Mecanismos de Segurança

8.5 Senha de Acesso

8.5.1 Grupos de Usuários

8.5.2 Lista de Controle de Acesso

9 Gerência de Dispositivos

9.1 Subsistema de Entrada/Saída

9.2 Devices Drivers

9.2.1 Controladores

9.2.2 Dispositivos de E/S

9.2.3 Acesso Direto a Memória

9.3 Discos Magnéticos

9.3.1 Algoritmos para Otimização do Acesso

9.3.1.1 FCFS – First Come First Served

9.3.1.2 SSF – Shortest Seek-Time First

9.3.1.3 SCAN e Circular Scan – Varredura e Varredura Circular

9.4 Terminais

10 Estudos de Caso : UNIX

10.1 Histórico

10.2 Características

10.3 Estrutura do Sistema

10.3.1 Processo

10.4 Gerência do Processador

10.5 Gerência de Memória

10.6 Sistema de Arquivos

10.7 Gerência de Entrada/Saída

11 Estudo de Caso : Windows NT

11.1 Características

11.2 Estrutura do Sistema

11.2.1 Processo

11.3 Gerência do Processador

11.4 Gerência de Memória

11.5 Sistema de Arquivos

11.6 Gerência de Entrada/Saída