Sistemas Operacionais

2015

Flávio Augusto de Freitas
Professor de Informática

Impresso em outubro de 15

Índice Analítico

Capítulo 1
Introdução

1.1 - O que é um Sistema Operacional?

Mesmo que os usuários já tenham uma certa experiência em Sistemas Operacionais, é difícil dar-lhes uma idéia precisa do que é um Sistema Operacional (SO). Parte do problema vem do fato do Sistema Operacional realizar duas funções que não possuem nenhuma relação uma com a outra, e dependendo de quem está tentando passar a idéia, poderá dar ênfase maior a uma ou outra função. Essas funções estão descritas a seguir.

1.2 - O Sistema Operacional como uma Máquina Virtual

A arquitetura de um computador (conjunto de instruções, organização da memória, estrutura de Entrada/Saída (E/S) e estrutura de barramento) é bastante primitiva e difícil de programar, em especial a parte de entrada e saída. Fica claro que a maioria dos programadores não quer ou não pode envolver-se com detalhes da programação. Torna-se evidente, então, que o programador deseja lidar com uma abstração de alto nível e, consequentemente, bastante simples. No caso dos discos, uma abstração típica poderia fazer com que o disco fosse visto como uma coleção de arquivos identificados por nomes. Cada arquivo deve ser aberto para leitura ou escrita, em seguida deve ser lido ou escrito, e finalmente deve ser fechado. Detalhes a respeito do processo de gravação ou a respeito do estado da corrente do motor da controladora, não devem aparecer na abstração apresentada ao usuário.

O programa que esconde o verdadeiro hardware do usuário e apresenta-lhe um esquema simples de arquivos identificados que podem ser lidos ou escritos é, naturalmente, o Sistema Operacional. Da mesma forma que o Sistema Operacional isola o usuário dos detalhes da operação do disco, fornecendo-lhe uma interface bastante simples, ele também trata de uma série de outras questões de nível bastante baixo, tais como interrupções, os temporizadores, a gerência da memória e CPU, além de várias outras questões relacionadas ao uso do hardware. Em cada caso, a abstração apresentada ao usuário do SO é mais simples e mais fácil de utilizar que o próprio hardware.

O SO tornou a interação entre usuários e computador mais simples, confiável e eficiente, eliminando a necessidade do programador se envolver com a complexidade do hardware para poder trabalhar, ou seja, a parte física do computador tornou-se transparente para o usuário.

Partindo desse princípio, pode-se considerar o computador como uma máquina de níveis ou camadas onde, inicialmente, existem dois níveis: o nível 0 (hardware) e o nível 1 (Sistema Operacional).

De forma simplificada, uma boa parte dos computadores possui a estrutura mostrada na figura abaixo, podendo conter mais ou menos camadas, para adequar o usuário às suas diversas aplicações.

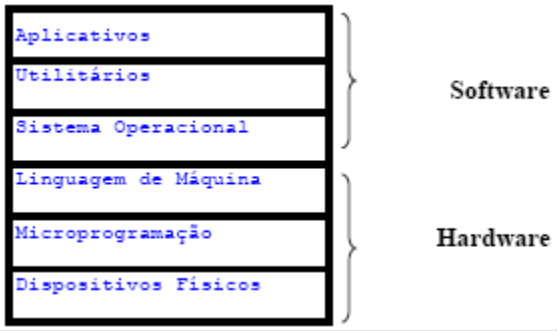


Figura 1.2 – Estrutura geral de um computador

Neste aspecto, a função do Sistema Operacional é a de apresentar ao usuário uma máquina virtual equivalente ao hardware, porém muito mais simples de programar.

1.3 - O Sistema Operacional Visto como um Gerente de Recursos

O conceito de Sistema Operacional como fornecedor de uma interface conveniente a seus usuários é uma visão top-down. Uma visão alternativa, bottom-up, mostra o SO como um gerente dos recursos de hardware disponíveis na máquina. Os computadores modernos são compostos de processadores, memórias, temporizadores, discos, dispositivos de fita magnética, interface de rede e dispositivos de E/S, tais como scanners, câmeras digitais e impressoras. Na visão alternativa, a função do Sistema Operacional é a de fornecer um esquema de alocação dos processadores, das memórias e dos dispositivos de Entrada/Saída entre os vários processos que competem pela utilização de tais recursos.

Imagine o que poderia acontecer se três processos, executando em um dado computador, resolvessem imprimir suas saídas simultaneamente na mesma impressora. As três primeiras linhas da listagem poderiam ser do processo 1, as seguintes do processo 2, e assim por diante, alternadamente, até que os três terminassem a impressão. Fica claro que tal situação não é admissível em nenhum sistema computacional. O Sistema Operacional tem por função colocar ordem nestes casos, armazenando em disco todas as saídas destinadas à impressora, durante a execução dos processos. Quando um dos processos terminar sua execução, o SO copia sua saída do disco para a impressora, enquanto os demais continuam a executar e, eventualmente, a gerar saída no arquivo em disco.

No caso do computador possuir múltiplos usuários, a necessidade de gerência e proteção da memória, dos dispositivos de Entrada/Saída e dos demais recursos do sistema fica ainda mais aparente. Tal necessidade vem do fato de ser frequentemente necessário aos usuários fazer o compartilhamento de recursos relativamente caros, como é o caso das impressoras a laser. Além das questões econômicas, é comum, entre usuários que trabalham juntos no mesmo sistema, a necessidade de compartilhar informações.

Esta outra visão da função de um Sistema Operacional mostra que sua tarefa principal é a de gerenciar os usuários no uso de recursos da máquina, contabilizando o tempo de uso de cada um e garantindo o acesso ordenado dos usuários a recursos, por meio da mediação dos conflitos entre as requisições dos diversos processos do sistema.

Capítulo 2
Evolução dos Sistemas Operacionais

2.1 - Introdução

Uma boa maneira de se compreender um SO é acompanhando a sua evolução através dos últimos anos. Assim, pode-se entender o porquê de determinadas características, como e quando estas foram incorporadas.

Sistema Operacional e arquitetura de computadores influenciaram-se mutuamente. O SO surgiu da necessidade de aproveitamento do hardware. Em determinados momentos, alterações no hardware foram necessárias para facilitar o projeto de um determinado SO. O surgimento de novas facilidades de hardware propiciou melhor desenvolvimento de SOs, mais complexos e eficientes. A Figura 2.1 mostra a relação da evolução de software e hardware.

Figura 2.1 – Relação da Evolução do Software e Hardware.

2.2 - Os primeiros Sistemas Operacionais

A evolução dos SOs está, em grande parte, relacionada ao desenvolvimento de equipamentos cada vez mais velozes, compactos e de custos baixos, e à necessidade de aproveitamento e controle desses recursos. Neste histórico a evolução é dividida em fases, onde são destacadas, em cada uma, suas principais características de hardware e de software.

2.2.1 - Primeira Fase (1945-1955)

Os primeiros computadores digitais surgiram na II Guerra Mundial. Eles eram formados por milhares de válvulas e ocupavam áreas enormes, sendo de funcionamento lento e não confiável.

O primeiro computador digital de propósito geral foi o ENIAC (*Eletronic Numerical Integration and Computer*). Sua estrutura possuía válvulas, capacitores, resistores, pesava 30 toneladas, e ele realizava 5 mil adições por segundo.

Para trabalhar nessas máquinas, era necessário conhecer profundamente o funcionamento do hardware, pois a programação era feita em painéis, por meio de fios, e em linguagem de máquina.

Existia um grupo de pessoas que projetava, construía, programava, operava e realiza a manutenção nestes computadores. Nesta fase não existia o conceito de Sistema Operacional e nem de linguagem de programação.

2.2.2 - Segunda Fase (1956-1965)

A criação do transistor e das memórias magnéticas contribui para o enorme avanço dos computadores da época. O transistor permitiu o aumento da velocidade e da confiabilidade do processamento, e as memórias magnéticas permitiram o acesso mais rápido aos dados, maior capacidade de armazenamento e computadores menores.

Com o surgimento das primeiras linguagens de programação, como Assembly e Fortran, os programas deixaram de ser feitos diretamente no hardware, o que facilitou o processo de desenvolvimento de programas. Os primeiros Sistemas Operacionais surgiram, justamente, para tentar automatizar as tarefas manuais até então utilizadas.

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

Para otimizar a entrada de programas, surgiram as máquinas leitoras de cartão perfurado que aceleravam muito a entrada de dados. Os programadores deveriam então escrever seus programas e transcrevê-los em cartão perfurados. Cada programa e seus respectivos dados eram organizados em conjuntos denominados *jobs* que poderiam ser processados da seguinte forma. Os vários *jobs* de diferentes usuários eram lidos por um computador, como o IBM 1401, que gravava os dados em uma fita magnética. Depois de aproximadamente 1 hora de coleta do lote de *jobs*, a fita era rebobinada e levada por um operador (pessoa) para a sala de máquina. Na sala de máquina, estava o computador que era adequado para cálculos numéricos, como o IBM 7094. Este computador era mais caro do que o 1401. O operador, então, montava a fita magnética em um drive de fita de entrada e carregava um programa (o ancestral dos SOs de hoje) no 7094, o qual lia o primeiro *job* da fita, o executava e a saída era escrita em uma segunda fita (drive de saída). Depois de cada *job* ter terminado, o SO lia o próximo *job* e o executava. Depois de todo o lote de *jobs* ter sido lido e executado, o operador levava a fita de saída para um outro computador 1401 o qual imprimia as saídas dos *jobs*. A Figura 2.2 ilustra o processo descrito.

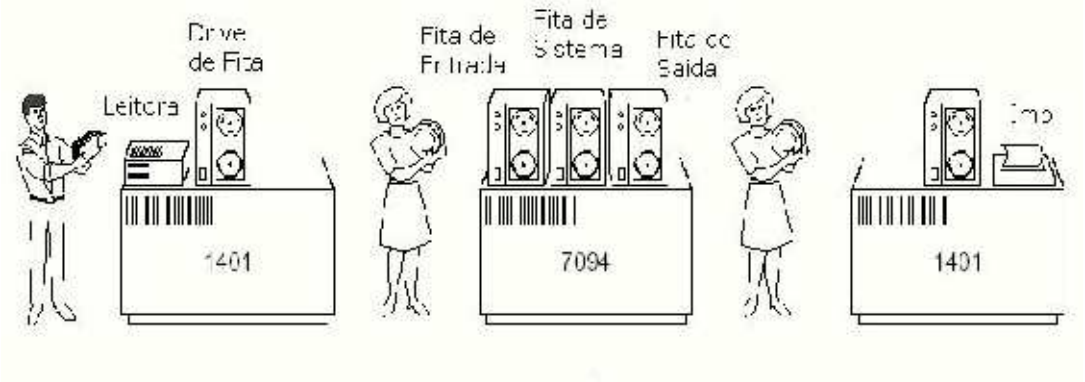


Figura 2.2 – Um Sistema de Processamento em Lote (Batch). Imp = Impressora.

Apesar da natureza sequencial do processamento, para os usuários era como se um lote de *jobs* fosse processado a cada vez, originando o termo Processamento em Lote (*Batch Processing*). Com o Processamento Batch, um grupo de programas era submetido de uma só vez, o que diminuía o tempo existente entre a execução dos programas, permitindo o melhor uso do processador.

Os SOs passaram e ter seu próprio conjunto de rotinas para operações de Entrada/Saída (*Input/Output Control System* – IOCS), o que veio a facilitar bastante o processo de programação. O IOCS eliminou a necessidade de os programadores desenvolverem suas próprias rotinas de leitura/gravação para cada dispositivo periférico. Importantes avanços, em

nível de hardware, foram implementados no final dessa fase, principalmente na linha 7094 da IBM. Entre eles, destaca-se o conceito de canal, que veio permitir a transferência de dados entre dispositivos de E/S e memória principal de forma independente da CPU.

2.2.3 - Terceira Fase (1966-1980)

Com o uso de circuitos integrados (CIs) e, posteriormente, dos microprocessadores, foi possível viabilizar e difundir o uso de sistemas computacionais por empresas. Em 1964, a IBM lançou a Série 360. Esse lançamento introduziu umA linha de computadores pequena, poderosa e compatível. Para essa série, foi desenvolvido o sistema operacional OS/360, que introduziu novas técnicas, utilizadas até hoje.

Um dos principais avanços desta fase foi a utilização da multiprogramação. Segundo Deitel, "multiprogramação é quando vários *jobs* estão na memória principal simultaneamente, enquanto o processador é chaveado de um *job* para outro *job*, fazendo-os avançarem enquanto os dispositivos periféricos são mantidos em uso quase constante".

Enquanto o processamento chamado científico era muito bem atendido pelo Processamento em Lote comum, o mesmo não acontecia com o processamento dito comercial. No processamento científico, ocorre a execução de grande quantidade de cálculos com quantidade relativamente pequena de dados, mantendo o processador ocupado na maior parte do tempo. O tempo gasto com operações de Entrada/Saída era insignificante; eis então o fato deste comportamento ser chamado CPU *Bounded*. Já no processamento comercial, o processador permanece bastante ocioso dado que os cálculos são relativamente simples, em comparação com o processamento científico, e o uso de operações de Entrada/Saída é frequente, devido a quantidade de dados a ser processada. Este comportamento é conhecido como I/O (E/S) *Bounded*.

A multiprogramação permitiu uma solução para o problema da ociosidade do processador por meio da divisão da memória em partes, denominadas partições, onde em cada divisão um *job* poderia ser mantido, conforme mostrado na Figura 2.3.

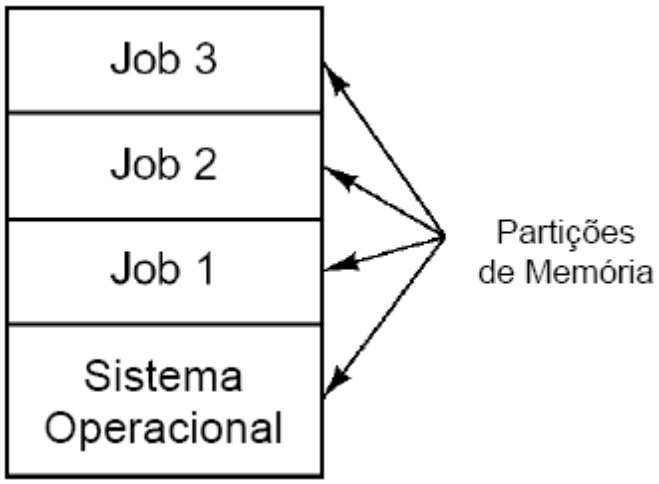


Figura 2.3 – Um sistema multiprogramado com 3 jobs na memória.

Na Figura 2.3, se o Job 1 solicitasse uma operação de E/S e estivesse esperando-a terminar, o processador poderia então ser chaveado para executar o Job 2, por exemplo. Se vários jobs fossem mantidos na memória de uma vez, a CPU permaneceria ocupada o suficiente para compensar o tempo das operações mais lentas de Entrada/Saída.

A utilização de fitas magnéticas obrigava o processamento a ser estritamente sequencial, ou seja, o primeiro programa a ser gravado na fita era o primeiro a ser processado. Assim, se um programa que levasse várias horas antecedesse pequenos programas seus tempos de resposta

ficariam comprometidos. Com o surgimento de dispositivos de acesso direto, como os discos, foi possível escolher os programas a serem executados, realizando o escalonamento de tarefas. Isto permitia a alteração na ordem de execução das tarefas, tornando o Processamento em Lote mais eficiente.

Outra técnica presente nesta geração foi a técnica de spooling (simultaneous peripheral operation on-line), isto é, a habilidade de certos SOs em ler novos jobs de cartões ou fitas armazenando-os em uma área temporária do disco rígido interno, para uso posterior quando uma partição de memória fosse liberada.

Um exemplo dessa técnica nos dias atuais ocorre 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. Essa técnica permite um maior grau de compartilhamento na utilização de impressora.

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

Associem-se a isto a limitação que ainda existia na comunicação de usuários com a máquina e, também, o problema de monopólio de CPU por parte de programas que tomam muito tempo de processamento e realizam relativamente poucas operações de E/S como, por exemplo, os programas científicos.

Tais problemas motivaram o surgimento de uma variação de multiprogramação chamada sistemas de tempo compartilhado (time-sharing systems). Os sistemas de tempo compartilhado permitiam a interação de vários usuários com o sistema, basicamente por meio de terminais de vídeo. Devido a essa característica, eles ficaram também conhecidos como sistemas on-line.

O computador poderia, então, ser usado por vários usuários ao mesmo tempo, por meio de pseudoparalelismo. O pseudoparalelismo poderia ser obtido com o chaveamento do processador entre vários processos que poderiam atender aos usuários. A idéia central era dividir o poder computacional do computador entre seus diversos usuários, passando a impressão de que o computador estava totalmente disponível para cada usuário, embora isto não fosse verdade.

Nos sistemas de tempo compartilhado, o tempo do processador era dividido em pequenos intervalos denominados quanta de tempo ou fatia de tempo (time-slice). A idéia consiste em dar uma fatia de tempo para cada programa em execução, tirando o direito de uso da CPU obedecendo as duas condições básicas abaixo:

1– se esgotar sua fatia de tempo, ele será transferido para uma Fila de Prontos;

2– se o programa necessitar de um recurso externo durante a execução, ele será transferido para uma Fila de Suspensos.

A Figura 2.4 mostra uma possível situação em um sistema de tempo compartilhado com Pn processos.

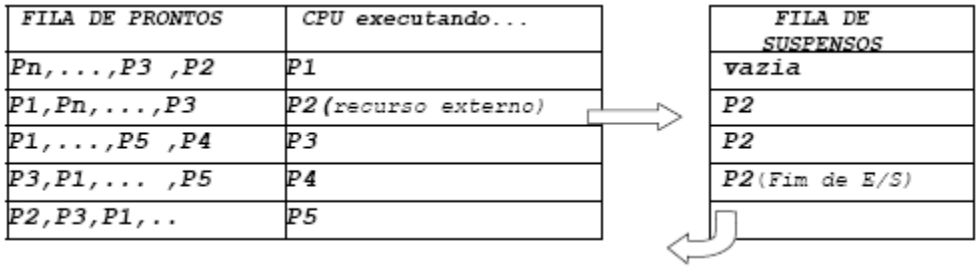


Figura 2.4 – Exemplo de sistema de tempo compartilhado.

Nos sistemas de tempo compartilhado, o controle da execução dos programas é feita interativamente e ocorre a eliminação do monopólio sobre a CPU da multiprogramação, realizando uma distribuição mais justa do tempo de uso do processador.

Outro fato importante nessa fase foi o surgimento do sistema operacional UNIX (1969). Concebido inicialmente em um minicomputador PDP-7, baseado no sistema MULTICS (Multiplexed Information and Computing Service), o Unix foi depois escrito em uma linguagem de alto nível (Linguagem C). O Unix tornou-se conhecido por sua portabilidade, pois era mais fácil modificar o código do sistema Unix para implementá-lo em um novo computador do que escrever um novo Sistema Operacional.

No final dessa fase, com a evolução dos microprocessadores, surgiram os primeiros microcomputadores, muito mais baratos que qualquer um dos computadores até então comercializados. Entre eles, destaca-se os micros de 8 bits da Apple e o Sistema Operacional CP/M (Control Program Monitor).

2.2.4 - Quarta Fase (1981-1990)

A integração em larga escala (Large Scale Integration – LSI) e integração em muito larga escala (Very Large Scale Integration – VLSI) levaram adiante o projeto de miniaturização e barateamento dos equipamentos. Os mini e superminicomputadores se firmaram no mercado e os microcomputadores ganharam um grande impulso.

Nesse quadro, surgiram os microcomputadores (Personal Computer - PC) de 16 bits da IBM e o Sistema Operacional da Microsoft MS-DOS, criando a filosofia dos computadores pessoais.

Na área dos minis e superminis, ganharam impulsos os sistemas multiusuários, destacando-se sistemas compatíveis com o Unix (Unix-type). O chip de microprocessador tornou possível para uma pessoa ter o seu próprio computador pessoal. O mais poderoso dos PCs usados por empresas, universidades e instalações governamentais eram usualmente denominadas de estações de trabalho (workstations) que, de fato, eram realmente PCs com maior capacidade de processamento. Apesar de monousuárias, elas permitem que se executem diversas tarefas concorrentemente, criando o conceito de multitarefa.

Neste ponto, é importante fazer uma observação sobre os termos multitarefa e sistemas de tempo compartilhado. O termo tempo compartilhado (time-sharing) está associado aos Mainframes, computadores grandes, potentes e caros, que possuíam terminais de vídeo, com quase nenhum processamento, ligados a eles. A idéia era permitir que múltiplos usuários compartilhassem um único Mainframe, conforme mencionado na seção 2.2.3. Atualmente, existe um conceito mais amplo que veio a substituir o conceito de Sistemas de Tempo Compartilhado, que é justamente Sistemas Multitarefa. Uma definição de Multitarefa é:

> *Método em que múltiplas tarefas podem ser executadas, aparentemente de forma simultânea, em um computador com uma única CPU.*

Pode-se dividir os Sistemas Multitarefas em dois tipos: Sistemas Preemptivos e Sistemas Cooperativos. Um Sistema Operacional Multitarefa Preemptivo contempla justamente as

mesmas características dos chamados sistemas de tempo compartilhado, conforme descrito na seção 2.3.3. deste Capítulo. Exemplos de Sistemas Multitarefa são o Windows NT, Unix e Linux. Um Sistema Operacional Multitarefa Cooperativo se caracteriza pelo fato de que o processo que está atualmente de posse da CPU, ou seja, o que está sendo executado, é quem passa o controle da CPU para outros processos. Um exemplo deste tipo de Sistema Operacional é o Windows 3.1.

É importante perceber que um Sistema Multitarefa Cooperativo não deixa de ser um Sistema de Tempo Compartilhado. Porém, será adotada a terminologia Sistemas de Tempo Compartilhado para referenciar os Sistemas Multitarefa Preemptivos.

As redes distribuídas (Wide Area Networks – WANs) se difundiram por todo o mundo, permitindo o acesso a outros sistemas de computação, em locais distantes. Também foram desenvolvidos inúmeros protocolos de rede como o DECnet da Digital E.C., SNA (System Network Architecture) da IBM, e outros de domínio público, como o TCP/IP e o X25. Surgem as primeiras redes locais (Local Area Networks – LANs) interligando pequenas áreas. Os softwares de rede passaram a estar intimamente relacionados ao Sistema Operacional e surgem os Sistemas Operacionais de Rede.

2.2.5 - Quinta Fase (1991-Atualmente)

Os grandes avanços em termos de hardware, software e telecomunicações no final de século passado, são consequência da evolução das aplicações, que necessitam cada vez mais de capacidade de processamento e armazenamento de dados. Sistemas especialistas, sistemas multimídia, banco de dados distribuídos, inteligência artificial e redes neurais são apenas alguns exemplos da necessidade cada vez maior de capacidade de processamento.

A evolução da microeletrônica permitirá o desenvolvimento de processadores e memórias cada vez mais velozes e baratos, além de dispositivos menores, mais rápidos e com maior capacidade de armazenamento. Os componentes baseados em tecnologia VLSI evoluem rapidamente para o ULSI (Ultra Large scale Integration).

Tópicos importantes:

- Modificações profundas nas disciplinas de programação no uso de arquiteturas paralelas;
- Processamento distribuído explorado nos Sistemas Operacionais, espalhando funções por vários processadores (redes);
- Arquitetura cliente-servidor em redes locais passa a ser oferecida em redes distribuídas, permitindo o acesso a todo tipo de informação, independentemente de onde esteja armazenada.
- Consolidação dos SOs baseados em interfaces gráficas. Novas interfaces homem-máquina serão utilizadas, como linguagem naturais, sons e imagens.
- Conceitos e implementações só vistos em sistemas considerados de grande porte estão sendo introduzidos na maioria dos sistemas desktop (Windows, Unix, Linux, etc.).
- Durante o ano de 1991, as primeiras versões (releases) do Sistema Operacional Linux começaram a ser desenvolvidas por Linus Torvalds, com ajuda de outros desenvolvedores.

Desenvolvido para os clones AT 386, 486, o Linux era (é) um sistema tipo Unix (Unix-type) para computadores pessoais e tinha como grande atrativo, além do fato de ser parecido com o Unix, o fato de ser um Sistema Operacional gratuito. Estima-se hoje que existam 18 milhões de usuários no Mundo usando Linux.

Capítulo 3
Estrutura de Sistemas Operacionais

A estrutura e funcionamento de um SO são tópicos de difícil compreensão. Um SO não é executado como uma aplicação sequencial, com início, meio e fim. As rotinas do SO são executadas sem uma ordem predefinida. A execução é baseada em eventos assíncronos.
Para tentar entender o conceito de estrutura de um SO, primeiro é necessário definir o que seja Núcleo (Kernel) de um Sistema Operacional, as Chamadas ao Sistema e Modos de Acesso.

3.1 Núcleo de um Sistema Operacional

O Núcleo (Kernel) é o software que fornece serviços básicos para todas as outras partes de um SO. De forma mais detalhada, o Núcleo é um conjunto de rotinas que oferecem serviços aos usuários do sistema e suas aplicações, bem como a outras rotinas do próprio SO.

Um núcleo pode ser contrastado com um interpretador de comandos, conhecido como shell nos sistemas UNIX, o qual não é parte do SO mas que desempenha um relevante papel interagindo com comandos do usuário. O shell será estudado adiante.

Dentre as principais funções do Núcleo estão:

- tratamento de interrupções;
- gerenciamento de processos (criação e destruição de processos; sincronização e comunicação entre processos, ...);
- gerenciamento de memória;
- gerenciamento do sistema de arquivos;
- operações de E/S.

Os serviços do Núcleo são solicitados por outras partes do SO ou por aplicações de usuários, por meio de um conjunto especificado de interfaces de programa (rotinas) conhecidas como Chamadas ao Sistema (System Calls).

3.2 Chamadas ao Sistema (System Calls)

As Chamadas ao Sistema são um mecanismo de proteção ao Núcleo do sistema e de acesso aos seus serviços. O usuário (ou aplicação), quando deseja solicitar algum serviço do SO, realiza uma chamada a um de seus serviços por meio de uma rotina (procedimento de biblioteca) que está diretamente associada às System Calls. A Figura 3.1 mostra a relação entre a rotina de biblioteca, usada pelos programas dos usuários, e as System Calls.

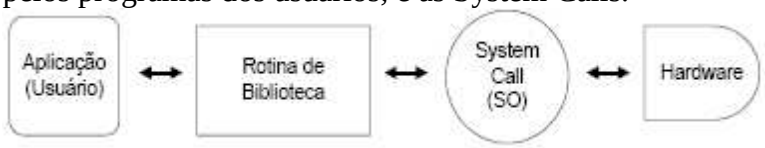


Figura 3.1 – Relação entre a System Call e a respectiva rotina de biblioteca.

Para tentar tornar claro o mecanismo de chamada ao sistema, seja o caso da System Call READ do SO UNIX. A System Call READ possui três parâmetros: o nome do arquivo que deverá ser lido, o buffer para onde devem ir os dados do arquivo e a quantidade de Bytes para serem lidos. Uma chamada a READ de um programa de usuário escrito em C poderia ser a seguinte:

```
contador = read (arquivo, buffer, nBytes);
```

É importante perceber que existe uma diferença entre a rotina de biblioteca usada no programa do usuário, read, e a System Call READ, a qual é invocada por read. O efeito de

executar read, e portanto chamar a System Call READ, é copiar os dados do respectivo arquivo para o buffer, onde o programa pode acessá-lo.

A rotina read retorna o número de Bytes atualmente lidos em contador. Este valor é normalmente o mesmo do que nBytes, mas pode ser menor se, por exemplo, for encontrado o final de arquivo (EOF) no processo de leitura.

OBS: embora tenha sido feita a distinção entre a rotina usada pelo programa do usuário (read, no exemplo), e a System Call (READ, no exemplo), alguns autores consideram a rotina usada pelo usuário (read) como, de fato, a System Call. Para este curso, o termo System Call será usado para se referir à rotina do SO (READ).

3.3 Modos de Acesso

As Chamadas ao Sistema estão relacionadas aos Modos de Acesso do processador. O Modo de Acesso é um mecanismo para impedir a ocorrência de problemas de segurança e mesmo violação do sistema. Existem as instruções privilegiadas as quais possuem o poder de comprometer o sistema, pois atuam diretamente no hardware da máquina. As instruções não privilegiadas não fornecem qualquer perigo ao sistema. Os Modos de Acesso podem ser divididos em dois nos SOs mais recentes:

1.) Modo Núcleo (Modo Supervisor, Modo Kernel): a aplicação pode ter acesso ao conjunto total de instruções do processador. O Sistema Operacional executa no Modo Núcleo, de modo que somente ele, SO, tem acesso às instruções privilegiadas.

2.) Modo Usuário: a aplicação pode executar somente instruções não privilegiadas (acesso a um número reduzido de instruções). Alguns dos programas de sistema, tais como editores e compiladores, também executam em Modo Usuário. É importante mencionar que editores e compiladores não fazem parte do SO, embora eles sejam costumeiramente fornecidos juntos com o SO.

A Figura 3.2 mostra a relação entre os Modos de Acesso do processador.

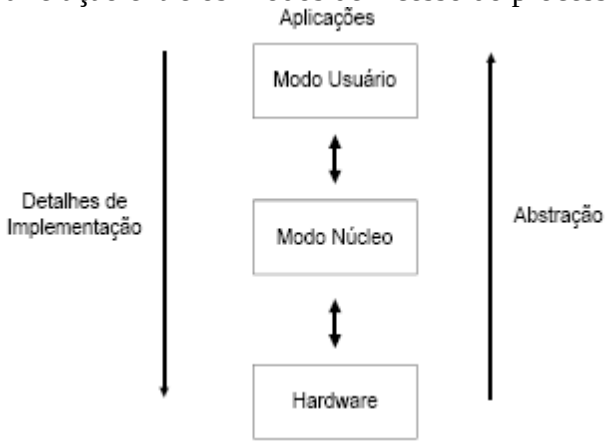


Figura 3.2 – Modos de Acesso.

Os processadores mais modernos implementam em hardware no mínimo 2 diferentes estados. Por exemplo, considerando a linha Intel, 4 estados determinam o nível de privilégio de execução dos programas. É possível ter os estados 0, 1, 2 e 3, sendo o estado 0 correspondendo ao Modo Núcleo. Sistemas Unix, o que inclui o Linux, requerem somente dois estados de privilégio, conforme mostrado na Figura 3.2.

Objetivando esclarecer como uma System Call pode ser realizada, seja o exemplo anterior onde o programa do usuário chamava a rotina read no UNIX. A rotina read coloca os parâmetros da System Call READ em um lugar especificado, como os registros de máquina do processador, e então executa uma instrução de controle (trap) especial. Esta instrução faz com que haja um

salto para uma posição definida no Núcleo do SO. Em processadores Intel, isto é feito por meio da interrupção 0x80. Desta forma, o hardware sabe que agora o programa está sendo executado no Modo Núcleo e que, portanto, o que está sendo executado é justamente o SO.

Após isto, o SO examina os parâmetros da chamada para determinar qual System Call (número da System Call) deve ser invocada. Sendo esse número k, o SO procura no slot k da tabela de System Calls o endereço da System Call k, e chama tal procedimento. Após terminada a System Call, o controle retorna para o programa do usuário. A Figura 3.3 ilustra o processo de uma Chamada ao Sistema.

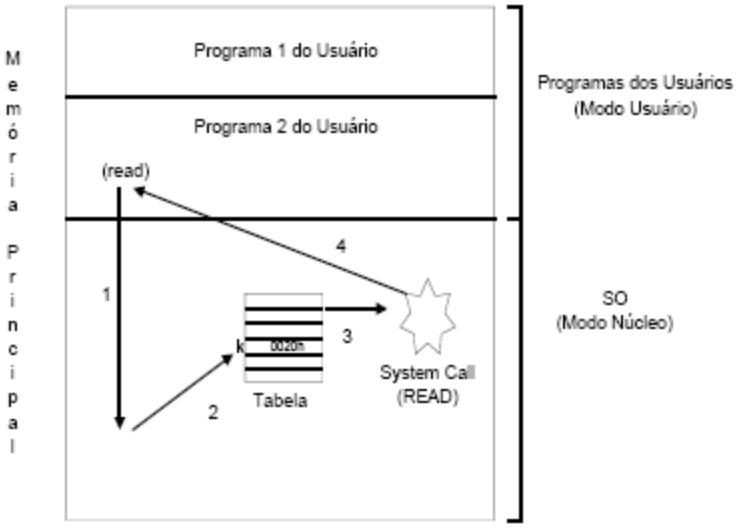


Figura 3.3 – Realização de uma System Call: (1) Programa do usuário (rotina de biblioteca) salta para o Núcleo. (2) SO determina o número do serviço requisitado. (3) SO localiza e chama a System Call. (4) Controle é retornado para o programa do usuário.

Na Figura 3.3, a título de exemplificação, percebe-se que o endereço onde está armazenada a System Call READ, chamada pelo rotina de read, é 0020h (base hexadecimal).

3.4 O INTERPRETADOR DE COMANDOS

Um Interpretador de Comandos não faz parte de um SO, embora seja extremamente útil. No UNIX, o interpretador de comandos é chamado de shell e ele é a interface primária entre um usuário sentado em frente ao computador e o SO. O shell e o Núcleo do SO são programas separados que se comunicam por meio de um conjunto de System Calls.

No caso do UNIX, considerando a interface de linha de comando, o shell mostrará um prompt após o usuário ter entrado no sistema (login). Supondo que seja o Bourne Shell, este prompt será $. Então, seja o caso de o usuário digitar na linha de comando o seguinte:

```
$ ls (Enter)
```

O comando ls lista arquivos e diretórios. O shell lerá, então, o comando digitado no terminal e entenderá que o usuário deseja executar o comando no arquivo /bin/ls. Então, o shell solicita ao Núcleo que crie um novo processo filho (rotina: fork; System Call: FORK). O processo filho executará o comando (ls) digitado pelo usuário por meio de outra System Call: EXEC (rotina execve, por exemplo). O shell se suspenderá, esperando que o processo filho termine de executar. Quando o processo filho terminar de executar, ele avisa ao Núcleo, por meio da System Call EXIT (rotina: exit) e o núcleo, por sua vez, acorda o shell e avisa-o que ele pode continuar a executar. O shell mostra o prompt novamente e espera por um novo comando. O que é importante é perceber que o shell é um programa comum de usuário. Tudo o que ele

precisa é a habilidade de ler ou escrever para o terminal, e a capacidade de interagir com o Núcleo do SO.

No Unix, outro shell bastante conhecido é o C Shell (%). No Linux, dois shells usados são o Bash (Bourne Again Shell) e o Csh.

3.5 Tipos de Estrutura de Sistemas Operacionais

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

3.5.1 Sistemas Monolíticos

Neste tipo de estrutura, o SO é escrito como uma coleção de rotinas, onde cada uma pode chamar qualquer outra rotina, sempre que for necessário. Portanto, o sistema é estruturado de forma que as rotinas podem interagir livremente umas com as outras. Quando esta técnica é usada, cada rotina no sistema possui uma interface bem definida em termos de parâmetros e resultados. A Figura 3.4 mostra a estrutura de um SO Monolítico.

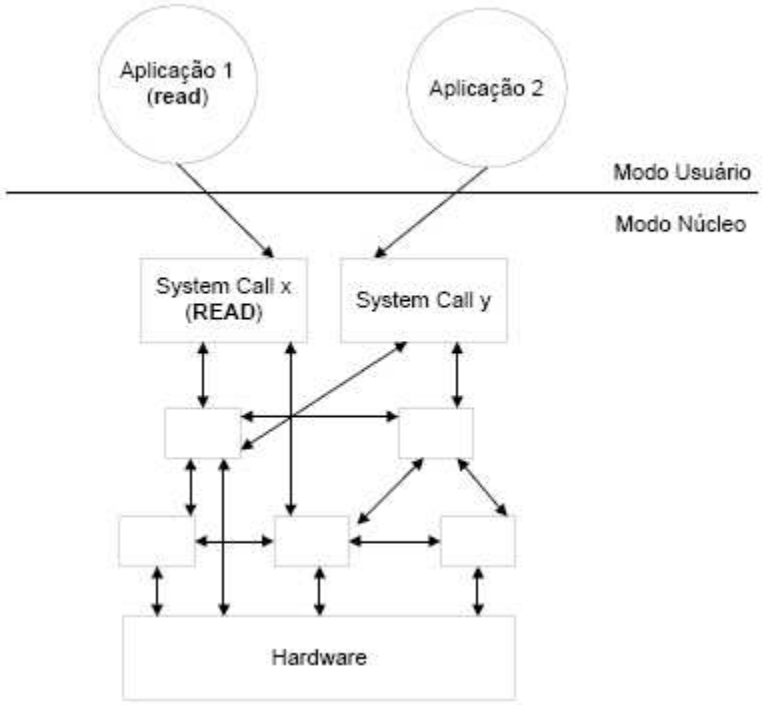


Figura 3.4 – Estrutura de um SO Monolítico.

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

Para construir um código executável de um SO desta natureza, todas as rotinas (ou arquivos que possuem as rotinas) são compiladas individualmente e unidas pelo linker em um código executável único. Tal código executa em Modo Núcleo. Não existe ocultação de informação, o

que também é indesejável, pois cada rotina é visível a qualquer outra. A Figura 3.5 mostra o processo de criação de um código executável de um SO Monolítico.

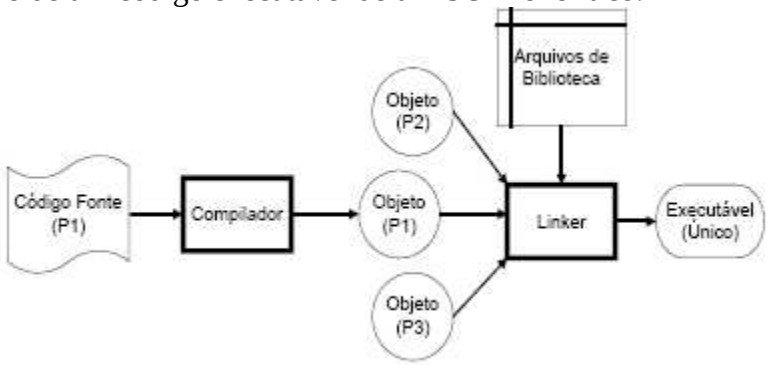


Figura 3.5 – Geração de Código Executável de um SO Monolítico. As caixas com borda mais grossa indicam ferramentas do ambiente de desenvolvimento.

Embora possa parecer que não há quase estruturação em um SO Monolítico, existe um pouco de estruturação quando os serviços do SO são solicitados por meio das System Calls. Este processo é exatamente o que foi descrito na seção 3.3 e sumarizado na Figura 3.3.

Como vantagem dos SOs Monolíticos pode-se afirmar que, se a implementação do sistema está completa e confiável, após um processo de desenvolvimento em que se supõe que técnicas consagradas de Engenharia de Software tenham sido aplicadas, a forte integração interna dos componentes permite que detalhes de baixo nível do hardware sejam efetivamente explorados, fazendo com que um bom SO Monolítico seja altamente eficiente. Entre os SOs Monolíticos estão as versões tradicionais do UNIX, incluindo o Linux, e MS-DOS.

3.5.2 Sistemas em Camada

A idéia por trás deste tipo de SO é fazer a organização por meio de hierarquia de camadas. O SO é dividido em camadas sobrepostas, onde cada módulo oferece um conjunto de funções que podem ser utilizadas por outros módulos. Módulos de uma camada podem fazer referência apenas a módulos das camadas inferiores.

O primeiro SO construído de acordo com esta concepção foi o THE, que foi desenvolvido na Technische Hogeschool Eindhoven na Holanda por E. W. Dijkstra (1968) e seus estudantes. O computador que executava o THE possuía Memória Principal com capacidade de 32K palavras de 27 bits cada. A estrutura do THE pode ser vista na Figura 3.6.

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

Figura 3.6 – Estrutura do Sistema Operacional THE.

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

A camada 1 realizava o gerenciamento de memória. Ela alocava espaço para os processos na Memória Principal do sistema e também em um Tambor (dispositivo de armazenamento magnético usado nos computadores antigamente) de 512K palavras, usado para armazenar partes de processos (páginas) para as quais não havia espaço na Memória Principal. Acima da camada 1, os processos não tinham que se preocupar se eles estavam na Memória Principal ou

no Tambor; a camada 1 do SO era quem tratava deste tipo de situação, trazendo as partes do software para a Memória Principal sempre quando necessário.

A camada 2 manipulava a comunicação entre cada processo e o operador do console. Um console consistia de um dispositivo de entrada (teclado) e um de saída (monitor ou impressora). A camada 3 era responsável pelo gerenciamento dos dispositivos de E/S.

Acima da camada 3, cada processo podia lidar com dispositivos de E/S abstratos, com propriedades mais agradáveis, e não com os dispositivos reais em si. Na camada 4 havia os programas do usuário, e na camada 5 havia o processo do operador do sistema.

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

3.5.3 Sistemas Cliente-Servidor

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

- serviços de arquivo (servidor de arquivos);
- serviços de criação de processos (servidor de processos);
- serviços de memória (servidor de memória), etc.

Para requisitar um serviço, tal como ler um bloco de dados de um arquivo, um processo de usuário, conhecido como processo cliente, envia uma solicitação a um processo servidor (servidor de arquivos, neste caso), que realiza o trabalho e envia a resposta de volta ao processo cliente. A Figura 3.7 mostra o modelo Cliente-Servidor.

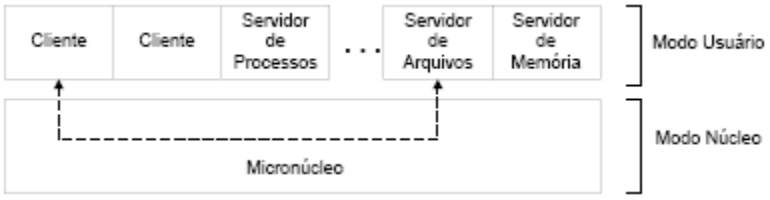


Figura 3.7 – Estrutura Cliente-Servidor.

Observe na Figura 3.7 que diversas funções do SO estão agora sendo executadas no Modo Usuário, e não mais no Modo Núcleo, como era no caso dos Sistemas Monolíticos. Somente o Micronúcleo do SO executa no Modo Núcleo. Ao Micronúcleo cabe, basicamente, permitir a comunicação entre processos clientes e servidores.

Entre as vantagens apresentadas pelos Sistemas Cliente-Servidor estão o fato de, ao dividir o SO em partes onde cada parte manipula um aspecto do sistema tais como serviço de arquivos, serviço de processo, serviço de memória entre outros, cada parte se torna menor e mais fácil de gerenciar. Além disto, devido ao fato de que todos os processos servidores executarem em Modo Usuário, eles não têm acesso direto ao hardware da máquina.

Como consequência, se houver um bug no processo servidor de arquivos, este serviço pode deixar de funcionar, mas isto usualmente não derrubará (crash) o sistema inteiro. Se esta mesma situação ocorresse em um SO Monolítico, e possível que o sistema sofresse consequências mais sérias do que em um Sistema Cliente-Servidor.

Uma outra vantagem do modelo Cliente-Servidor é a sua adaptabilidade para usar em sistemas com processamento paralelo/distribuído. Se um processo cliente se comunica com um servidor pelo envio de mensagens, o cliente não necessita saber se a mensagem é tratada localmente na sua máquina, ou se ela foi enviada por meio de uma rede para um processo

servidor executando em uma máquina remota. Do ponto de vista do cliente, o mesmo comportamento ocorreu: um pedido foi requisitado e houve uma resposta como retorno. A Figura 3.8 mostra esta situação.

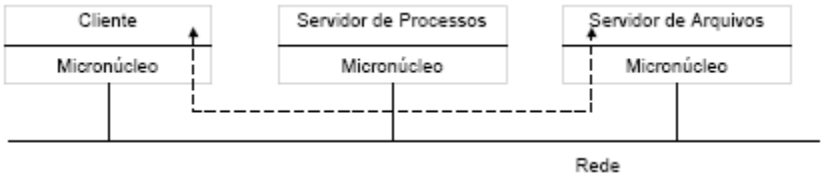


Figura 3.8 – Uso do Modelo Cliente-Servidor em um Sistema Paralelo ou Distribuído.

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

3.5.4 Sistemas Monolíticos versus Sistemas Cliente-Servidor

Em uma primeira análise, uma estrutura de SO Cliente-Servidor parece ser bem melhor do que um SO Monolítico. Porém, em termos práticos, a implementação de uma estrutura Cliente-Servidor é bastante complicada devido a certas funções do SO exigirem acesso direto ao hardware, como operações de E/S. Um núcleo Monolítico, por outro lado, possui uma complexidade menor, pois todo código de controle do sistema reside em um espaço de endereçamento com as mesmas características (Modo Núcleo).

Usualmente, SOs Monolíticos tendem a ser mais fáceis de desenhar corretamente e, portanto, podem crescer mais rapidamente do que SOs baseados em Micronúcleo. Existem casos de sucesso em ambas as estruturas.

Um aspecto interessante sobre qual a melhor estrutura de SO foi a discussão entre Linus Torvalds, o criador do SO Linux, e Andrew Tanenbaum, um dos principais pesquisadores na área de SOs e criador do SO Minix. Em 1992, quando o Linux estava no seu início, Tanenbaum decidiu escrever uma mensagem para o Newsgroup comp.os.minix, acusando justamente o Linux de ser um SO obsoleto.

O ponto principal do argumento de Tanenbaum era justamente a estrutura Monolítica, considerada ultrapassada por ele, do Linux.

Ele não concebia que um SO, em meados dos anos 90, fosse concebido com um tipo de estrutura que remonta a década de 70 (época em que o Unix foi desenvolvido; o Unix também é um SO Monolítico). O SO desenvolvido por Tanenbaum, Minix, apresenta estrutura baseada em Micronúcleo (Cliente-Servidor).

Em sua primeira resposta Torvalds argumentou, entre vários pontos, um aspecto não muito técnico: o Minix não era gratuito, enquanto o Linux sim. Do ponto de vista técnico, Torvalds concordava que um sistema com Micronúcleo era mais agradável.

Porém, ele acusava o Minix de não realizar corretamente o papel do Micronúcelo, de forma que havia problemas no que se refere a parte de multitarefa no núcleo do sistema.

A discussão continuou entre ambos sobre outros conceitos associados a SOs. Para saber mais sobre este assunto, vide o link `http://www.dina.dk/~abraham/Linus_vs_Tanenbaum.html#liu`. Dez anos após tal discussão, o que pode ser afirmado é que SOs Monolíticos ainda conseguem atrair a atenção de desenvolvedores devido a uma complexidade menor do que Sistemas Cliente-Servidor. Tanto que o Linux hoje é uma realidade, sendo um SO bastante usado em servidores em empresas e ambientes acadêmicos. Os Sistemas Cliente-Servidor, porém, possuem casos de sucesso como é o exemplo do sistema QNX, usado em sistemas de braços de robôs nos Ônibus Espaciais.

Capítulo 4
Processos

Um conceito chave em todos os Sistemas Operacionais é justamente o processo. Um processo é basicamente um programa em execução. Ele consiste do programa executável, dos dados do programa, do seu contador de programa (PC – Program Counter), de diversos registros e de toda a informação necessária para executar o programa.

Em um sistema multitarefa, a CPU permuta (switch) de programa para programa, executando cada um por dezenas ou centenas de milisegundos. Em cada instante de tempo, uma CPU pode executar somente um programa, mas, no curso de 1 segundo, ela pode executar diversos programas, dando a ilusão para os usuários de paralelismo. Isto é o que se pode denominar pseudoparalelismo, fazendo referência a esta rápida permutação de programas realizada pela CPU, em contraste com o verdadeiro paralelismo de hardware obtido, por exemplo, em um multiprocessador (1 computador com vários processadores internos os quais se comunicam por meio de memória compartilhada).

O fato de tratar atividades paralelas, múltiplas é algo difícil de ser realizado. Os desenhistas de SOs têm evoluído em um modelo que torna o paralelismo mais fácil de ser tratado. Este é o Modelo de Processo.

4.1 O Modelo de Processo

Neste modelo, todo o software executável no computador, frequentemente incluindo o SO, é organizado em um número de processos sequenciais, ou simplesmente processos. Conceitualmente, cada processo possui seu próprio processador virtual, mas, de fato, a CPU permuta de processo para processo, como já mencionado anteriormente. A Figura 4.1, mostra 4 programas na memória de um sistema multitarefa e a abstração em processos.

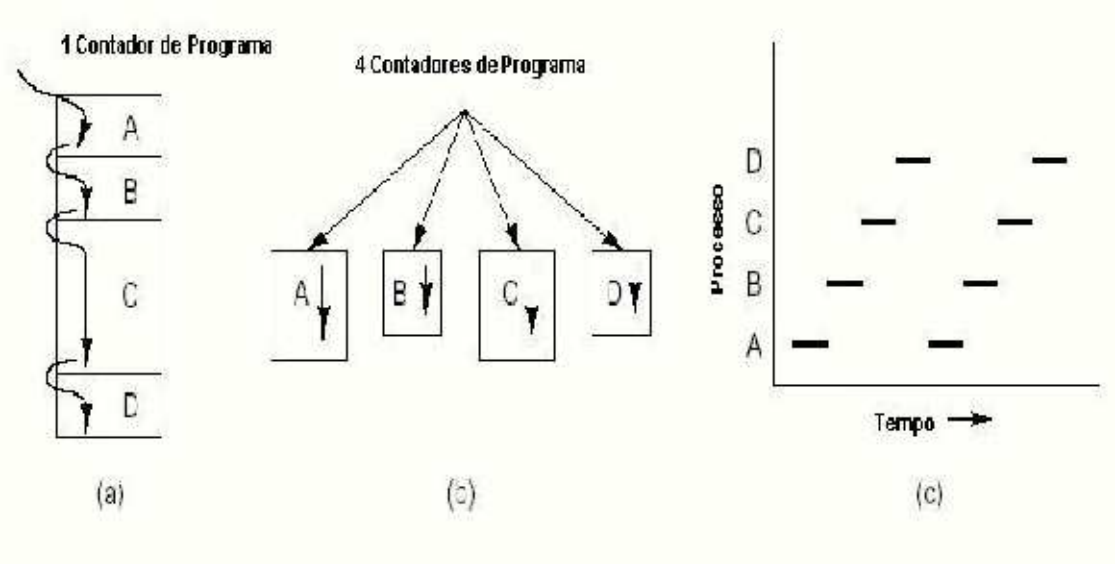


Figura 4.1 – Processos. (a) Quatro programas na memória principal compartilhando o tempo da CPU. (b) Modelo conceitual de quatro processos sequenciais e independentes. (c) Somente um processo está executando em cada instante.

Na Figura 4.1(b), pode-se perceber a multitarefa realizada pelo computador sendo abstraída em quatro processos, cada qual com o seu próprio fluxo de controle (ou seja, seu próprio

Contador de Programa) e executando independentemente uns dos outros. Na Figura 4.1(c), percebe-se que todos os processos com o decorrer do tempo fizeram evolução em relação à execução, mas somente um processo é executado em cada instante.
A diferença entre programa e processo é sutil, mas crucial.

Um programa é uma entidade passiva, enquanto um processo é uma entidade (unidade de trabalho) ativa. Neste contexto, um programa seria o conjunto de instruções necessárias (um algoritmo expresso em uma notação adequada) à execução das operações desejadas, enquanto que um processo seria o programa associado ao seu conjunto de dados e variáveis em um determinado instante de sua execução.

Em um Sistema Operacional existem basicamente dois tipos de processos: processos do SO e processos do usuário. Os processos do sistema executam funções de gerenciamento de recursos e gerenciamento de processos do usuário, enquanto que os processos do usuário executam as tarefas programadas pelo usuário. Todos estes processos podem executar concorrentemente. Para garantir a ordem desta execução, o SO deve fornecer um mecanismo de comunicação e sincronização entre processos.

Os processos podem ser classificados como independentes ou cooperativos. Um processo é dito independente no caso em que a sua execução não afete e não seja afetada pela execução de um outro processo. Um processo é dito cooperativo quando interage (troca informações) com outros processos presentes no sistema.

4.1.1 Estados dos Processos

Embora cada processo seja uma entidade independente, com seu próprio PC e estado interno, processos frequentemente precisam interagir (cooperar) com outros processos. Um processo pode gerar saída da qual outro processo necessite como entrada. Seja o caso de digitar no shell do Unix o seguinte comando:

```
cat cap1 cap2 cap3 | grep linguagem.
```

O comando cat do Unix é extremamente flexível. Ele pode ser usado para criar, ver e concatenar arquivos. No exemplo acima, cat está sendo usado para concatenar 3 arquivos (cap1, cap2 e cap3) e gerar a saída para a tela do computador como resultado. O símbolo | é denominado de pipe (tubo). O que pipe faz, neste caso, é encaminhar a saída do comando cat para a entrada do comando grep.

O comando grep serve para procurar informação em um arquivo ou em vários arquivos. No exemplo, grep mostrará todas as linhas da saída de cat que possuem a palavra linguagem. Dependendo das velocidades relativas dos dois processos, um executando cat e outro grep, o que deriva tanto da complexidade de cada programa como de quanto tempo de CPU cada um tem tido, pode ser que grep esteja pronto para executar, mas não existe entrada disponível para ele. Então grep deve-se bloquear até que a entrada esteja disponível.

Na medida em que um processo vai sendo executado, o estado em que ele se encontra é alterado de acordo com a atividade que está sendo efetuada. Os estados em que um processo pode se encontrar variam de sistema para sistema mas, de uma maneira geral, pode-se citar: executando, pronto e bloqueado. A explicação destes estados é dada a seguir:

- executando (running): também chamado de estado ativo. Um processo está no estado executando, como o próprio nome diz, quando ele está sendo executado pelo processador. Em ambientes com um único processador, somente um processo pode estar sendo executado em um certo instante;

- pronto (ready): temporariamente parado para que outro processo possa ser executado. O processo encontra-se pronto para a execução, aguardando apenas a liberação do processador para que ele seja executado;

- bloqueado (blocked): um processo está no estado bloqueado quando aguarda a ocorrência de algum evento externo para poder prosseguir, como o término de uma operação de E/S.

A Figura 4.2 mostra os estados possíveis de um processo e as transições que podem ocorrer.

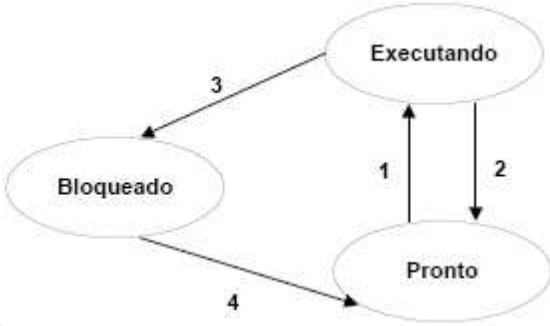


Figura 4.2 – Estados dos processos e transições associadas.

A mudança de estados (transições) de um processo durante o seu processamento ocorre em função dos eventos originados por ele próprio (evento voluntário) ou pelo Sistema Operacional (evento involuntário). Basicamente, existem 4 mudanças de estado que podem ocorrer a um processo, conforme detalhado a seguir e mostrado na Figura 4.2:

1. pronto/executando: causada pelo escalonador de processos (parte do SO) de forma involuntária ao processo. Quando um processo é criado, o sistema o coloca em uma fila de processos prontos, onde aguardará uma oportunidade para ser executado. Cada SO tem seus próprios critérios e algoritmos para a escolha da ordem em que os processos serão executados (escalonamento). Esta transição também ocorre quando todos os processos já tiveram o seu tempo de execução (fatia de tempo) e agora é novamente a vez do primeiro processo da fila de prontos ser executado pela CPU;

2. executando/pronto: causada pelo escalonador de processos (parte do SO) de forma involuntária ao processo. Um processo em execução passa para o estado de pronto quando ocorrer o término da sua fatia de tempo, por exemplo. Nesse caso, o processo volta para a fila de processos prontos, onde aguarda uma nova fatia de tempo;

3. executando/bloqueado: Um processo executando passa para o estado bloqueado por meio de eventos gerados pelo próprio processo como, por exemplo, uma operação de E/S.
Em alguns sistemas, o próprio processo, de maneira voluntária, deve executar uma system call, BLOCK, para que ele passe para o estado bloqueado;

4. bloqueado/pronto: quando ocorrer a conclusão do evento solicitado, um processo no estado bloqueado passará para o estado de pronto (fila de prontos), para que possa ser novamente escalonado.

Objetivando detalhar como ocorrem as transições associadas aos processos, seja o caso do usuário solicitar ao SO que execute um programa. O SO então cria um processo e associa a este um número de identificação, pid (process identifier), colocando-o no final da fila de processos prontos. Em outras palavras, o estado inicial de um processo é o estado pronto. Suponha que o pid deste processo é 5 (P5). Em um determinado tempo de execução, a fila de prontos e o processo sendo executado pela CPU pode ser como mostrado na Figura 4.3. Perceba que o processo P5 é o próximo processo a ser executado.

FILA DE PRONTOS	*CPU executando ...*
..., P4, P3, P1, P5	P2

Figura 4.3 – Exemplo de processos em um sistema de tempo compartilhado.

Em um sistema de tempo compartilhado, a entidade que coordena a utilização do processador por parte dos processos é o escalonador de processos (scheduler). O escalonador é uma função de baixo nível, que utiliza de um temporizador (timer) do sistema para efetuar a divisão do processamento e, portanto, está intimamente ligado ao hardware do computador.

Regularmente, a cada fatia de tempo (time-slice) do sistema, este temporizador dispara uma interrupção a qual, em última instância, ativará uma rotina que corresponde ao escalonador do sistema. Tal rotina realiza algumas operações com os registradores do processador, de forma que o resultado é o chaveamento do processador para o próximo processo na fila de prontos. Ao terminar esta interrupção, o novo processo em execução é aquele designado pelo escalonador. Na seção 4.1.2, será abordado com mais profundidade o papel do escalonador de processos.

No exemplo mostrado na Figura 4.3, esgotando a fatia de tempo de P2, a interrupção será gerada pelo temporizador e ocorrerá o chaveamento, de forma que P5 estará sendo executado após o término da interrupção. Do ponto de vista de P5, esta mudança é a transição 1 da Figura 4.2. A Figura 4.4 mostra a nova situação após o chaveamento.

FILA DE PRONTOS	*CPU executando ...*
P2, ..., P4, P3, P1	P5

Figura 4.4 – Processo P5 sendo executado após o chaveamento.

A Figura 4.5 mostra o mecanismo de chaveamento de processos para este caso.

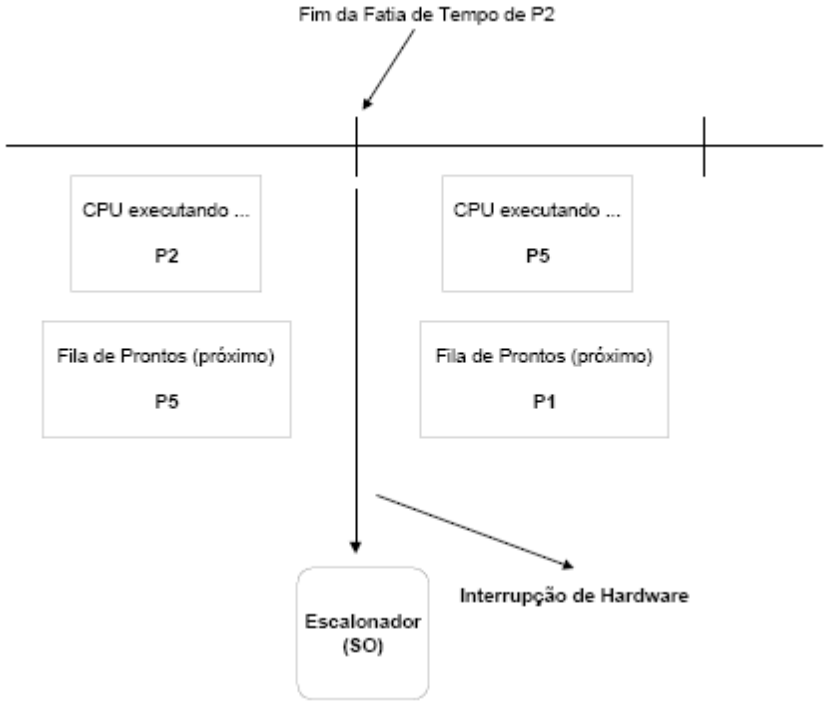


Figura 4.5 – O escalonador e o mecanismo de chaveamento de processos.

Supondo que P5 começou a ser executado e terminou a sua fatia de tempo. Então, uma interrupção será gerada e o escalonador colocará P5 na fila de prontos, conforme mostra a Figura 4.6. Isto corresponde a transição 2 da Figura 4.2.

FILA DE PRONTOS	*CPU executando ...*
P5,P2 ..., P4, P3	P1

Figura 4.6 – Término da Fatia de tempo de P5.

Quando P5 voltar a ser escalonado para usar a CPU, suponha que, antes de terminar a sua fatia de tempo, ele realize uma operação de E/S. O próprio processo P5 se suspenderá para utilizar ou esperar pela disponibilidade do recurso solicitado. Esta situação corresponde a transição 3 da Figura 4.2. Ao finalizar o uso do recurso, o SO recolocará P5 na fila de processos prontos.

Esta situação corresponde a transição 4 da Figura 4.2.

4.1.2 Implementação de Processos

Para implementar o modelo de processo, é comum a criação e manutenção de uma tabela que organize as informações relativas aos processos. Esta tabela é chamada de tabela de processos e é usualmente implementada sob a forma de um vetor de estruturas ou uma lista ligada de estruturas. Cada processo existente corresponde a uma entrada nesta tabela, e cada entrada da tabela é denominada PCB (Process Control Block ou Bloco de Controle de Processo). O PCB possui todas as informações necessárias para que a execução do processo possa ser iniciada, interrompida e retomada conforme determinação do SO, sem prejuízo para o processo. A Figura 4.7 mostra a tabela de processos, implementada como um vetor de estruturas, e os PCBs.

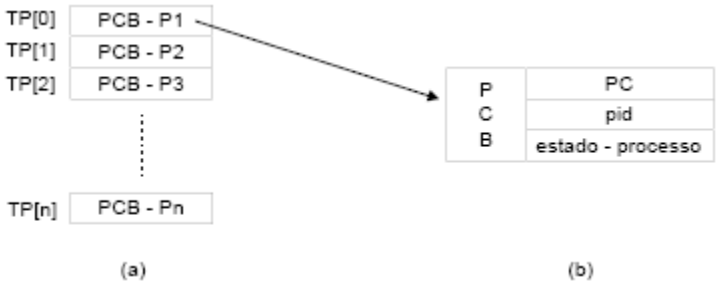


Figura 4.7 – Tabela de processos e PCBs. Em (a), a tabela de processos implementada como um vetor de n posições; em (b), um exemplo de PCB.

Os campos exatos das entradas na tabela de processos, PCBs, variam de sistema para sistema mas, em geral, uns lidam com gerenciamento de processos, outros com gerenciamento de memória e outros com o sistema de arquivo. Algumas informações típicas que o PCB possui são:

- identificador de processo (pid);
- estado atual do processo;
- cópia do conteúdo do registrador contador de programa (PC – Program Counter);
- cópia do conteúdo dos demais registradores do processador;
- pid do processo pai (parent process);
- ponteiro para a pilha;
- tempo em que o processo iniciou;
- tempo utilizado do processador;
- informações sobre diretório raiz e de trabalho.

A troca de um processo por outro para ser executado pela CPU, realizada pelo SO com o auxílio do hardware, é denominada mudança de contexto. No fundo, a mudança de contexto consiste em salvar as informações PCB do processo que está deixando de ser executado e carregar os valores referentes ao do processo que esteja ganhando a utilização do processador. Como exemplo, suponha, por simplificação, que cada PCB possua as seguintes informações: PC, pid, estado do processo.

Considerando a Figura 4.5, o processo P2 está atualmente sendo executado pela CPU e P5 é o próximo na fila de prontos. Ao terminar a fatia de tempo de P2, uma interrupção é gerada e o SO, com auxílio do hardware, salva as informações relevantes de P2 na sua entrada na tabela de processos. Supondo que no instante imediatamente anterior à interrupção, o valor do registrador PC seja 0B30h (processador com endereços de 16 bits) e o PC do processo P5, armazenado no PCB de P5, seja 0780h, as informações na tabela de processo ficariam como mostra a Figura 4.8.

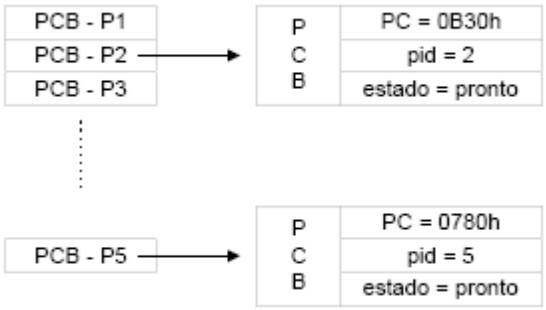


Figura 4.8 – Tabela de processos e PCBs de P2 e P5 após as informações de P2 terem sido salvas.

Após isto, o escalonador é chamado para decidir qual novo processo será o eleito para ser executado pela CPU. No exemplo, o escalonador escolhe P5 para ser o próximo a ser executado. Desta forma, o SO copia o conteúdo dos registradores no PCB de P5 para os registradores do processador e começa a executar P5. A Figura 4.9 ilustra a tabela de processos e o registrador PC logo após o escalonamento de P5.

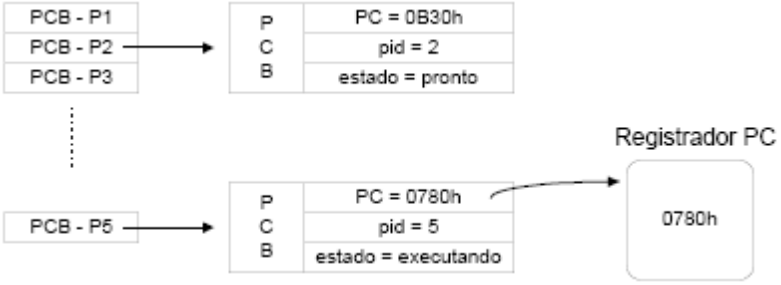


Figura 4.9 – Situação após a mudança de contexto.

Perceba que agora o registrador PC possui o valor que estava no PCB de P5 e o estado de P5 está como executando.

OBS: Ao comentar a Figura 4.2, foi mencionado que as mudanças de estado pronto/executando e executando/pronto eram tarefas do escalonador de processos. De fato, o escalonador é quem decide qual novo processo deve ser escolhido, baseando-se nas informações da tabela de processos. Porém, existem outros processos do SO associados com a mudança de contexto. Tais processos são responsáveis, por exemplo, pela atualização dos PCBs.

4.2 Comunicação entre Processos

A comunicação entre processos (IPC – Interprocess Communication) é uma situação comum em sistemas de computação, que ocorre quando dois ou mais processos precisam se comunicar, ou seja, quando os processos precisar compartilhar ou trocar dados entre si. A comunicação entre processos pode ocorrer em várias situações diferentes, tais como:

- redirecionamento de saída (resultados) de um comando para outro;
- envio de arquivos para impressão;
- transmissão de dados pela rede, entre outras.

Tal comunicação ocorre, geralmente, por meio da utilização de recursos comuns, como a memória do sistema, aos processos envolvidos na comunicação. Devido à complexidade e limitações de desempenho, as interrupções não são usadas para este tipo de comunicação. A seguir serão abordados alguns tópicos associados à comunicação entre processos.

4.2.1 Condições de Corrida

Em alguns Sistemas Operacionais, processos que estão trabalhando conjuntamente frequentemente compartilham algum recurso comum de armazenamento onde cada processo pode ler ou escrever em tal recurso. O armazenamento compartilhado pode estar na memória principal ou pode ser um arquivo compartilhado. Para entender a natureza da comunicação entre processos, considere um exemplo simples mas comum: um spooler de impressão. Quando um processo deseja imprimir um arquivo, ele coloca o nome do arquivo em um diretório especial chamado de diretório de spooler. Um processo responsável pelo gerenciamento de impressão periodicamente verifica se existem arquivos para serem impressos e, caso haja, ele os imprime e remove os nomes dos arquivos do diretório.

Considere que o diretório de spooler possui um número grande de slots (0, 1, 2, ...) cada qual sendo capaz de armazenar um nome de arquivo que deverá ser impresso. Também considere que existem duas variáveis compartilhadas, na memória principal, tais que:

- **prox**: aponta para o próximo arquivo para ser impresso;
- **livre**: aponta para o próximo slot livre no diretório.

Em um certo instante, os slots de 0 a 3 estão vazios, os arquivos já forma impressos, e os slots de 4 a 6 estão cheios, com os nomes dos arquivos para serem impressos. Quase que simultaneamente, os processos A e B decidem que eles desejam imprimir um arquivo. A Figura 4.10 mostra esta situação.

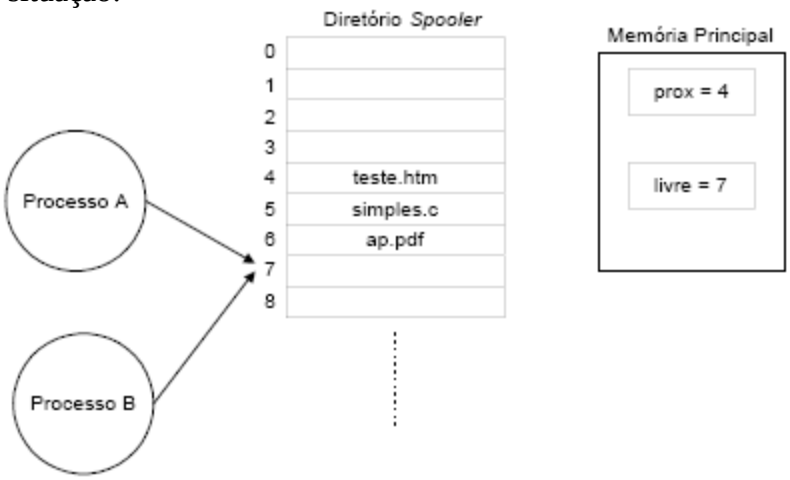


Figura 4.10 – Dois processos decidem acessar memória compartilhada ao mesmo tempo.

Seja o seguinte caso. O processo A está executando atualmente e ele lê a variável livre, cujo valor atual é 7 pois este é o próximo slot livre no diretório, e armazena este valor em uma variável local sua, chamada slot_livre. Logo após realizar isto, a fatia de tempo de A terminou e este é posto na fila de prontos. O processo B é o novo processo a executar e, assim como A, B lê a variável livre e obtém o valor 7. Então, B escreve o nome do arquivo que ele deseja imprimir (outro.c, por exemplo) no slot 7 e incrementa a variável compartilhada livre (livre = 8, agora).

O processo B tinha sido o último a enviar arquivo para o diretório spooler, até que o processo A, depois de um certo tempo, volta a ter a posse da CPU. Então, A será executado do ponto em que parou e, ao verificar a sua variável local slot_livre, A encontra o valor 7. Portanto, ele escreve o nome do arquivo que deseja imprimir (isto.pdf, por exemplo) no slot 7 do diretório spooler, sobrescrevendo o nome do arquivo que B tinha posto no referido slot. Então, A incrementa a variável slot_livre e atualiza a variável livre para 8. A Figura 4.11 resume a sequência de ações dos dois processos.

```
Processo A → slot_livre = livre (7)
Processo B → lê livre (7)
Processo B → escreve o nome do arquivo (outro.c) no slot 7
Processo B → livre = livre + 1 (8)
.
.
.
Processo A → verifica slot_livre (7)
Processo A → escreve o nome do arquivo (isto.pdf) no slot 7
Processo A → slot_livre = slot_livre + 1 (8)
Processo A → livre = 8
```

Figura 4.11 – Sequência de ações dos processos A e B. A ação em negrito indica o momento em que o arquivo de B foi sobrescrito.

O diretório spooler está internamente consistente e o processo que gerencia a impressão nada encontrará de errado nesta situação. Entretanto, o processo B jamais conseguirá ter o seu arquivo, outro.c, impresso. Situações como esta, onde dois ou mais processos estão lendo ou escrevendo dados compartilhados e o resultado final depende de qual processo executa e quando executa são chamadas condições de corrida. Evidentemente, esta é uma situação indesejável e que deve ser contornada de alguma forma.

4.2.2 Seções (Regiões) Críticas

A idéia principal para prevenir as condições de corrida, em diversas situações envolvendo compartilhamento de memória principal, compartilhamento de arquivos e etc..., é encontrar uma forma que proíba que mais de um processo leia e/ou escreva ao mesmo tempo os dados compartilhados. Em outras palavras, o que é necessário é a chamada exclusão mútua – uma forma de assegurar que, se um processo está usando uma variável compartilhada ou arquivo, os outros processos estarão excluídos de fazer o mesmo.

O problema de evitar condições de corrida pode também ser formulado da seguinte maneira. Parte do tempo, um processo está ocupado realizado computações internas e outras tarefas que não conduzem às condições de corrida. Porém, em algumas oportunidades, o processo pode estar acessando memória e arquivos compartilhados e isto pode resultar em condições de corrida. A parte do programa onde a memória compartilhada é acessada é denominada de seção ou região crítica. Se for assegurado que dois processos jamais entrem em suas seções críticas ao mesmo tempo, as condições de corrida podem ser evitadas. Além de uma parte de software, uma seção crítica pode ser, também, um dispositivo de hardware ou uma rotina de acesso para um certo dispositivo de hardware.

Embora o requisito de assegurar que somente um processo esteja dentro de sua seção crítica por vez evite condições de corrida, isto não é suficiente para o caso de existirem processos paralelos cooperando correta e eficientemente usando dados compartilhados. Os requisitos para ter uma boa solução são:

1.) Dois processos não podem estar simultaneamente dentro das suas seções críticas;

2.) Nenhuma consideração deve ser feita a respeito da velocidade e do número de processadores;

3.) Nenhum processo executando fora da sua seção crítica deve bloquear outros processos;

4.) Nenhum processo deve ter de esperar eternamente para entrar na sua seção crítica.

4.2.3 Propostas Para Obtenção de Exclusão Mútua

Algumas propostas para que a exclusão mútua seja obtida serão analisadas nesta seção. A primeira delas está associada a um buffer compartilhado por dois processos.

4.2.3.1 Buffers e Operações de Sleep e Wakeup

Existem algumas primitivas (rotinas) de comunicação entre processos que se caracterizam por bloquear o processo quando a este não é permitido entrar na sua seção crítica. Umas das mais simples são sleep (system call SLEEP) e wakeup (system call WAKEUP). SLEEP faz com que o processo que a chama seja bloqueado, ou seja, seja suspenso até que outro processo o acorde. WAKEUP é uma system call que possui um parâmetro: o processo que deve ser acordado. Para exemplificar como estas primitivas são usadas, será abordado o problema do Produtor-Consumidor.

No problema do Produtor-Consumidor, existem dois processos que compartilham um buffer, uma área de dados de tamanho fixo que se comporta como um reservatório temporário. Se Produtor e Consumidor são processos que executam em sequência, a solução do problema é simples, mas se estes são processos que executam paralelamente passa existir a situação de concorrência. O processo Produtor coloca informações no buffer enquanto o Consumidor as retira de lá. Existem casos onde podem existir múltiplos Produtores ou múltiplos Consumidores, mas basicamente o problema encontrado é o mesmo. Estes são problemas clássicos de comunicação entre processos, tais como os problemas do jantar dos filósofos e do barbeiro dorminhoco, discutidos em detalhes por Tanenbaum no livro-texto desta disciplina.

O exemplo mostrado na seção 4.2.1, onde processos colocavam os nomes dos arquivos que eles desejavam que fossem impressos em um diretório (área) de spooler e um processo de gerenciamento de impressão imprimia os arquivos e os removia do diretório, é um exemplo do problema do Produtor-Consumidor. No caso mais geral, tendo em mente que o diretório de spooler é finito, onde existem vários processos desejando imprimir (Produtores) e várias impressoras (Consumidores), as velocidades dos diversos processos que desejam imprimir podem ser consideravelmente diferentes das velocidades das impressoras.

Um outro exemplo pode ocorrer quando diversos processos utilizam uma placa de rede para realizar a transmissão de dados para outros computadores. Os vários processos aparecem como Produtores, enquanto o hardware da placa e seu código representam o Consumidor. Em função do tipo de rede e do tráfego, existe uma forte limitação com que a placa consegue consumir (transmitir) os dados produzidos pelos programas e colocados no buffer de transmissão. A Figura 4.12 mostra a situação do problema do Produtor-Consumidor.

Produtor

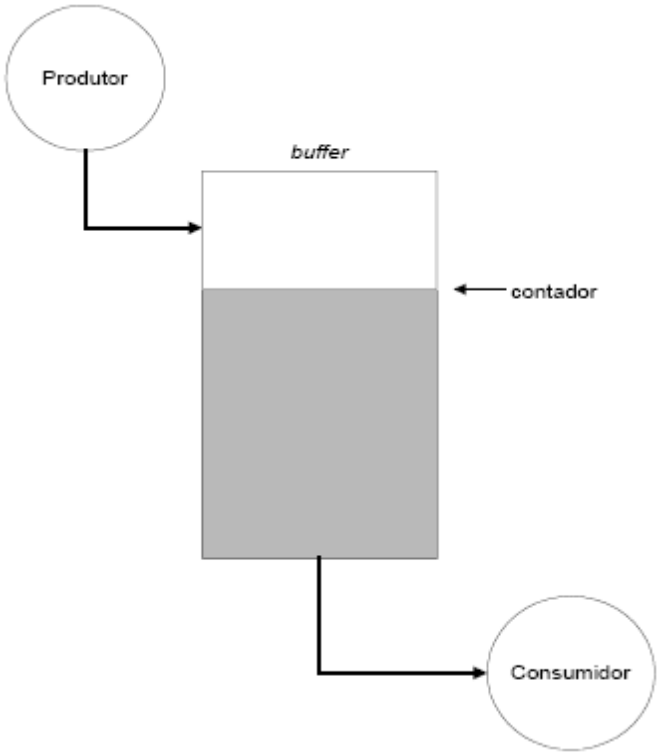


Figura 4.12 – Problema do Produtor-Consumidor.

Tanto o buffer como a variável que controla a quantidade de dados do buffer, contador, são seções críticas e, portanto, deveriam ter seu acesso limitado por meio de primitivas de exclusão mútua, desde que isso não impusesse esperas demasiadas aos processos envolvidos.

Problemas ocorrem quando o Produtor deseja colocar uma nova informação (item) no buffer, mas este está cheio. A solução, do ponto de vista do Produtor, é este adormecer (rotina sleep system call SLEEP) e ser acordado pelo Consumidor (rotina wakeup system call WAKEUP) quando ele, Consumidor, remover um ou mais itens do buffer. De forma similar, se o Consumidor deseja remover um item do buffer e observa que o buffer está vazio, o Consumidor adormece até que o produtor coloque um ou mais itens no buffer e o acorde.

Se a variável contador informa quantos itens existem atualmente no buffer, o Produtor deve verificar se contador é igual a N, onde N é o valor máximo de itens que o buffer pode comportar. Caso isto seja verdadeiro, o Produtor deve adormecer; caso contrário, o Produtor adicionará um item no buffer e incrementará contador. Do ponto de vista do Consumidor, primeiramente verifica-se se contador é igual a 0. Se isto for verdadeiro, então o Consumidor deve adormecer; senão, o Consumidor deve remover um item e decrementar contador. A Figura 4.13 mostra os códigos-fonte do Produtor e Consumidor escritos em linguagem C.

```
#include "prototypes.h"

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


int contador = 0;


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


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

Figura 4.13 – Solução parcial do problema do Produtor-Consumidor.

A solução mostrada na figura 4.13 é dita parcial, pois pode ocorrer condição de corrida. A sequência de eventos mostrada na Figura 4.14 ilustra a condição de corrida que pode acontecer.

1. *buffer* está vazio.
2. Consumidor lê contador = 0.
3. SO (escalonador) interrompe o Consumidor.
4. Produtor (processo sendo executado) produz novo item.
5. Produtor lê contador = 0 e adiciona o item no *buffer*.
6. Produtor incrementa contador de forma que contador = 1.
7. Produtor envia sinal wakeup para Consumidor (contador = 1).
8. Sinal wakeup é perdido pois Consumidor não está logicamente inativo (não bloqueado).
9. Consumidor volta a ser executado e testa o valor de contador que ele leu anteriormente. Como contador = 0, Consumidor adormece (sleep → bloqueado).
10. Produtor continuará sendo executado e produzindo itens.
11. Em certo ponto, *buffer* ficará cheio, pois Consumidor está bloqueado, fazendo com que o Produtor se coloque como inativo (contador = N; sleep → bloqueado).
12. Ambos os processos permanecerão para sempre inativos (bloqueados).

Figura 4.14 – Eventos que geram condição de corrida com a solução parcial do problema do Produtor-Consumidor.

A perda de um sinal wakeup deriva do fato do acesso à variável contador ser irrestrito. Uma primeira solução para este problema foi adicionar um flag (wakeup waiting bit) que sinalizasse

a situação de envio de um sinal wakeup para um processo ativo. Na próxima tentativa de adormecer, o processo verifica este flag ocorrendo o seguinte:

- se flag = 0 processo adormece;
- se flag = 1 processo permanece ativo e faz flag = 0.

O flag mencionado funciona para os casos mais simples, porém é insuficiente para os casos onde existem mais processos em operação.

4.2.3.2 Semáforos

Dentro do contexto apresentado no tópico anterior, Dijkstra propôs em 1965 a utilização de variáveis inteiras para controlar o número de sinais wakeup para uso futuro. Na proposta apresentada por ele, um novo tipo de variável, denominada semáforo, foi introduzido. Um semáforo poderia ter um valor 0, indicando que nenhum wakeup foi salvo, ou algum valor positivo se um ou mais sinais wakeup estão pendentes.

Duas operações foram estabelecidas para atuar sobre as variáveis do tipo semáforo: P (conhecida também como down system call DOWN) e V (conhecida também como up system call UP), que são generalizações de sleep e wakeup, respectivamente. O funcionamento de DOWN e UP é descrito na Figura 4.15.

Operação P(x) ou DOWN(x)	Operação V(x) ou UP(x)
• Verifica se o valor do semáforo x é positivo (x > 0) • Caso afirmativo, decrementa x de uma unidade (ou seja, consome um sinal wakeup) • Caso negativo, envia sinal sleep, fazendo com que o processo fique inativo	• Incrementa o valor do semáforo x de uma unidade • Existindo processos inativos (bloqueados), esperando a finalização de uma operação down, um deles é escolhido aleatoriamente pelo sistema para ser ativado (semáforo retornará para zero)

Figura 4.15 – Funcionamento de down e up.

Ambas as operações devem ser realizadas como ações atômicas, ou seja, de forma indivisível. Deste modo, garante-se que, uma vez que uma operação de semáforo foi iniciada, nenhum outro processo pode acessar o semáforo até que a operação tenha sido completada ou bloqueada. A maneira usual de fazer isto é implementar DOWN e UP como chamadas ao sistema, com o SO desabilitando todas as interrupções enquanto está testando o semáforo, atualizando-o e colocando o processo para dormir, se for necessário. Todas estas ações consomem apenas umas poucas instruções, não ocasionando maiores problemas ao desabilitar as interrupções. Portanto, para o caso de DOWN, verificar o valor do semáforo, modificá-lo e possivelmente colocar o processo para dormir é feito de forma indivisível. Para o caso de UP, incrementar o semáforo e, possivelmente, acordar um processo é feito de maneira indivisível.

Nenhum processo se bloqueia realizando um UP, assim como nenhum processo se bloqueia realizando um WAKEUP no modelo anterior.

A solução do problema do Produtor-Consumidor usando semáforos pode ser vista na Figura 4.16.

```
#include "prototypes.h"

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

typedef int semaforo;

semaforo mutex = 1; /* controla acesso à região crítica */
semaforo vazio = N; /*contador de slots vazios do buffeer */
semaforo cheio = 0; /* contador de slots cheios do buffer */

void produtor (void) {
  int item;

  while (TRUE) {
     produzir_item(&item);
     down(&vazio);           /* decrementa vazio */
     down(&mutex);           /* entra na região crítica */
     colocar_item(item);
     up(&mutex);             /* sai da região crítica */
     up(&cheio);             /* incrementa cheio */
  }
}

void consumidor (void) {
  int item;

  while (TRUE) {
     down(&cheio);          /* decrementa cheio */
     down(&mutex);          /* entra na região crítica */
     remover_item(&item);
     up(&mutex);            /* sai da região crítica */
     up(&vazio);            /* incrementa vazio */
     consumir_item(item);
  }
}
```

Figura 4.16 – O problema do Produtor-Consumidor usando semáforos.

A solução apresentada usa três semáforos. O semáforo cheio conta o número de slots que estão cheios no buffer, o semáforo vazio conta o número de slots vazios no buffer enquanto o semáforo mutex assegura que o Produtor e o Consumidor não acessam o buffer ao mesmo tempo. O semáforo cheio está 0 inicialmente, enquanto vazio inicia com valor igual ao número de slots no buffer e mutex inicialmente vale 1. Semáforos que são iniciados com 1 e que são usados por dois ou mais processos para assegurar que somente um deles possa entrar na sua seção crítica por vez são chamados de semáforos binários. Se cada processo realizar um DOWN imediatamente antes de entrar na sua seção crítica e um UP logo após sair dela, a exclusão mútua está garantida.

Na Figura 4.16, semáforos foram usados de duas maneiras distintas. O semáforo mutex foi usado para exclusão mútua. O objetivo dele é garantir que somente um processo por vez possa ler do ou escrever no buffer e variáveis associadas. O outro uso de semáforos é para sincronização. Os semáforos cheio e vazio são necessários para garantir que certas sequências de eventos possam ou não ocorrer. Neste caso, eles asseguram que o Produtor pára de executar se o buffer estiver cheio, e o Consumidor pára de executar quando o buffer estiver vazio. Este uso é diferente do caso de exclusão mútua. A utilização de semáforos permite, portanto, a sincronização de vários processos. Em outras palavras, em um ambiente onde existem vários processos concorrendo por recursos, semáforos podem ser usados para garantir o uso exclusivo de um recurso por um processo.

4.2.3.3 Monitores

Com semáforos, a comunicação entre processos parece fácil.

Mas, suponha que dois DOWNs no código do Produtor fossem invertidos de ordem no código, de forma que mutex fosse decrementado antes de vazio ao invés de depois de vazio. Se o buffer estivesse completamente cheio (semáforos cheio = N e vazio = 0, no exemplo) o produtor se bloquearia, pois vazio igual a 0, com mutex igual a 0. Consequentemente, na próxima vez que o Consumidor tentasse acessar o buffer, ele realizaria um DOWN em mutex e, como mutex está igual a 0, ele se bloquearia também.

Ambos os processos ficariam bloqueados indefinidamente e nenhum trabalho seria realizado a partir deste ponto. Esta situação indesejada é chamada **deadlock**. Uma definição formal de deadlock é a seguinte: "*Um conjunto de processos está em deadlock se cada processo no conjunto está esperando por um evento que somente outro processo no conjunto pode causar*." Este problema é mencionado para demonstrar o quão cuidadoso o programador deve ser ao usar semáforos. Os erros gerados são condições de corrida, deadlocks e outras formas de comportamento imprevisíveis e não reprodutíveis.

Para que tais problemas pudessem ser resolvidos mais facilmente, Hoare (1974) e Hansen (1975) propuseram o conceito de monitores. Um monitor é uma coleção de procedimentos, variáveis, e estruturas de dados que são todos agrupados conjuntamente em um tipo especial de módulo ou pacote. Processos podem chamar os procedimentos em um monitor sempre que eles precisarem, mas eles não podem acessar diretamente as estruturas de dados internas do monitor, por meio de procedimentos declarados fora do monitor.

Monitores possuem uma propriedade importante que os torna úteis para garantir exclusão mútua: somente um processo pode estar ativo em um monitor em qualquer instante. Monitores são uma construção de linguagens de programação, de forma que o compilador sabe que eles são especiais, e que pode manipular chamadas a procedimentos de monitor de forma diferente de outras chamadas a procedimentos.

É responsabilidade do compilador implementar exclusão mútua em monitores, mas uma maneira usual de se fazer isto é usando semáforos binários, como o semáforo mutex da Figura 4.16. Devido ao fato do compilador, e não o programador, está encarregado da exclusão mútua, é bem menos provável que algo saia errado. Para um programador, é suficiente saber que ao estarem todas as seções críticas em procedimentos de monitores, dois processos jamais executarão suas seções críticas ao mesmo tempo.

Embora monitores forneçam uma maneira fácil de conseguir exclusão mútua, é necessário haver um modo dos processos bloquearem quando eles não podem prosseguir. No problema do Produtor-Consumidor é fácil colocar todos os testes de "buffer cheio?" e "buffer vazio?" em procedimentos de monitor, mas como deve o Produtor se bloquear quando ele encontrar o buffer cheio ou Consumidor quando ele encontrar o buffer vazio? A solução está no uso de variáveis de condição e com duas operações nela: WAIT e SIGNAL. Quando um procedimento de monitor descobre que ele não pode continuar (por exemplo, o Produtor encontra o buffer cheio), ele realiza um WAIT em uma variável de condição, que poderia ser denominada cheio. Esta ação causa o processo em questão, o Produtor, se bloquear e permite a outro processo, que não foi permitido anteriormente entrar no monitor, entrar agora.

Este outro processo, que poderia ser o Consumidor, poderia acordar o seu parceiro adormecido por meio de um SIGNAL na variável de condição na qual o seu parceiro está esperando (variável de condição cheio, no exemplo). Após a realização de um SIGNAL, o processo que o fez deve sair imediatamente do monitor.

Isto evita ter dois processos ativos no monitor ao mesmo tempo. Em outras palavras, um SIGNAL deve aparecer somente como uma instrução final no procedimento do monitor. Se um SIGNAL é realizado em uma variável de condição onde vários processos estão esperando, somente um deles, determinado pelo escalonador, é reativado.

Variáveis de condição não são contadores. Eles não acumulam sinais para uso posterior como os semáforos fazem. Então, se um SIGNAL é realizado em uma variável de condição com nenhum processo esperando nesta variável, então o sinal é perdido. Portanto, o WAIT deve vir antes do SIGNAL. Um esqueleto do problema do Produtor-Consumidor com monitores é dado na Figura 4.17, em uma linguagem semelhante ao Pascal.

```
monitor ProdutorConsumidor
  condition cheio, vazio;
  integer contador;

  procedure entrar;
  begin
    if contador = N then wait(cheio);
    entrar_item;
    contador := contador + 1;
    if contador = 1 then signal(vazio);
  end;

  procedure remover;
  begin
    if contador = 0 then wait(vazio);
    remover_item;
    contador := contador - 1;
    if contador = N - 1 then signal(cheio);
  end;

  contador:=0;
end monitor;
```

```
procedure produtor;
begin
  while true do
  begin
    produzir_item;
    ProdutorConsumidor.entrar;
  end;
end;

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

Figura 4.17 – O problema do Produtor-Consumidor com monitores. O buffer possui N slots.

As operações WAIT e SIGNAL são bastante similares às operações SLEEP e WAKEUP, respectivamente, com uma diferença crucial: SLEEP e WAKEUP falham pois é possível que, enquanto um processo está tentando adormecer, outro está tentando acordá-lo. A exclusão mútua assegurada nos procedimentos do monitor evita este problema pois, por exemplo, se um processo Produtor dentro de um procedimento de monitor encontrar um buffer cheio, será possível a ele completar a operação WAIT sem que seja necessário ele, Produtor, se preocupar com a possibilidade do escalonador chavear para o Consumidor imediatamente antes de WAIT ser completada.

A despeito dos aspectos positivos mencionados, monitores também têm os seus problemas. Não é sem razão que o exemplo mostrado na Figura 4.18 foi descrito em uma linguagem estranha, semelhante ao Pascal, ao invés de C. Monitores são um conceito de linguagens de programação. O compilador deve reconhecê-los e deve lidar com a exclusão mútua de alguma forma. C, Pascal e a maioria de outras linguagens de programação não possui monitores, então

não é razoável esperar que os seus respectivos compiladores reforcem quaisquer regras de exclusão mútua. Portanto, com monitores é necessária uma linguagem que os tenha embutidos.

Algumas poucas linguagens, como Concurrent Euclid, possuem monitores, mas elas são raras.

Outro problema com monitores, e também com semáforos, é que eles foram projetados para resolver o problema da exclusão mútua em uma ou mais CPUs, onde todas possuem acesso a uma memória comum. Porém, em um sistema distribuído consistindo de múltiplos processadores, cada qual com a sua memória privada, conectados por uma rede local, estas primitivas se tornam não aplicáveis. A conclusão é que semáforos são de muito baixo nível e que monitores existem apenas em poucas linguagens de programação.

4.2.3.4 Passagem de Mensagens

Este método de comunicação entre processos usa duas primitivas SEND e RECEIVE as quais, da mesma forma como semáforos e diferentemente de monitores, são system calls ao invés de construções de linguagem. Deste modo, elas podem ser facilmente colocadas em procedimentos de biblioteca. Um possível formato para estas primitivas é mostrada na Figura 4.18.

```
send(destino, &mensagem);

receive(fonte, &mensagem);
```

Figura 4.18 – Primitivas de comunicação usadas em passagem de mensagens.

Na Figura 4.18, destino e fonte são identificadores de processos (pid) e mensagem corresponde à informação que deve ser transmitida (enviada pelo processo fonte e recebida pelo processo destino). Existem outras assinaturas de send e receive diferentes da mostrada na Figura 4.18, onde outros parâmetros são passados para as primitivas, como o tamanho da mensagem que está sendo transmitida.

Sistemas que usam passagem de mensagens possuem muitos problemas desafiadores que não ocorrem com semáforos ou monitores, especialmente se os processos comunicantes estão em computadores diferentes conectados por uma rede. Embora passagem de mensagens também possa ser aplicada a processos executando na mesma máquina, no caso dos processos estarem em computadores diferentes, onde cada computador possui a sua memória privada, problemas como a mensagem ser perdida pela rede pode ocorrer. Para contornar este problema, os processos fonte e destino podem concordar que, logo que uma mensagem tenha sido recebida, o processo destinatário enviará uma mensagem especial de reconhecimento (acknowledgement).

Se o processo fonte não receber o acknowledgement em um certo intervalo de tempo (timeout), o fonte enviará a mensagem novamente.

Considere, então, o que ocorre se a mensagem propriamente dita é recebida, mas o reconhecimento enviado é perdido. O processo fonte retransmitirá a mensagem e o processo destinatário receberá a mesma mensagem duas vezes. Usualmente este problema pode ser resolvido colocando números de sequências consecutivos em cada mensagem original. Se o processo destino receber uma mensagem cujo número de sequência é o mesmo da mensagem anterior, ele saberá que é uma mensagem duplicada e ignorará a mensagem.

4.2.3.4.1 Classificação da Comunicação em Sistemas com Várias CPUs

A classificação da comunicação em sistemas que possuem vários processadores interligados e trabalhando em conjunto pode ser analisada sobre diversos aspectos. Entre estes, pode-se citar: sincronismo e bloqueamento.

Em termos de sincronismo, a comunicação pode ser síncrona ou assíncrona. Na comunicação síncrona, um processo fonte, ao invocar uma primitiva send, fica bloqueado enquanto a mensagem está sendo enviada. A instrução seguinte a send não é executada até que a mensagem seja completamente enviada. Similarmente, uma chamada a receive não permite a execução da próxima instrução, até que a mensagem seja recebida e armazenada na área alocada para isto. Em outras palavras, o processo que chamar a primitiva de comunicação primeiro "espera" pelo outro.

Na comunicação assíncrona, o processo fonte não espera pelo respectivo destinatário. Uma vez invocada send, ocorre o retorno ao programa principal sem esperar o término da mensagem. Os dados enviados devem ser armazenados de alguma forma, neste caso.

Em relação a bloqueamento, a comunicação pode ser com bloqueio ou sem bloqueio. Na comunicação com bloqueio, só existe um retorno das rotinas send/receive, e isto ocorre quando a comunicação for finalizada. No caso da comunicação sem bloqueio, pode haver um retorno das rotinas send/receive sem que a comunicação tenha sido completada. Então, em uma comunicação síncrona, ambas as primitivas, send e receive, são com bloqueio.

No caso de uma comunicação assíncrona, podem existir dois casos:

• send sem bloqueio, receive com bloqueio: embora o processo fonte possa seguir com o seu processamento, após chamar a rotina send, sem que a comunicação tenha sido terminada, um processo destinatário fica bloqueado, caso seja chamado um receive cujo respectivo send do processo fonte não tenha ocorrido;

• send sem bloqueio, receive sem bloqueio: ao invocar um receive, um processo destinatário segue o seu processamento, da mesma forma que o fonte.

Nos casos de comunicação sem bloqueio, buffers são necessários para o armazenamento da mensagem. Podem ocorrer problemas, caso a quantidade de buffers se torne muito grande durante as comunicações.

4.2.3.4.2 Problema do Produtor-Consumidor com Passagem de Mensagens

O problema do Produtor-Consumidor será analisado com a solução usando passagem de mensagens e sem compartilhamento de memória. Assume-se que todas as mensagens possuem o mesmo tamanho e que as mensagens enviadas mas não ainda recebidas são armazenadas automaticamente pelo SO. Além disso, o Produtor e o Consumidor estão em computadores diferentes. Portanto, a comunicação é assíncrona, send sem bloqueio, receive com bloqueio.

Nesta solução, um total de N mensagens é usado, analogamente aos N slots em um buffer de memória compartilhado. A Figura 4.19 mostra a solução do problema do Produtor-Consumidor com passagem de mensagens.

```
#include "prototypes.h"

#define N 5             /*número de "slots" no buffer*/
#define TamanhoMens 4   /*tamanho da mensagem*/

typedef int mensagem[TamanhoMens];

/* O código do Produtor ou do Consumidor será executado de
acordo com o número do processo na rede:
0 = Consumidor
1 = Produtor */

void produtor(void)
{
  int item, i, consumidor;
  mensagem m;

  i = obter_numero_processo();
  if (i == 1){
    consumidor = 0;
    while(TRUE) {
      produzir_item(&item);       /*gera algo para o buffer*/
      receive(consumidor, &m);    /*espera uma mens vazia*/
      montar_mensagem(&m, item); /*constrói mens p/enviar*/
      send(consumidor, &m);       /*envia item p/consumidor*/
    }
  }
}

void consumidor(void)
{
  int item, i, j, produtor;
  mensagem m;

  i = obter_numero_processo();
  if (i == 0) {
    produtor = 1;
    for (j=0; j<N; j++)
       send(produtor, &m); /*envia N mensagens vazias*/
    while(TRUE) {
      receive(produtor, &m);      /*recebe mens com item*/
      extrair_item(&m, &item);    /*extrai o item da mens*/
      send(produtor, &m);         /*envia mensagem vazia*/
      consumir_item(item);        /*faz algo com o item*/
    }
  }
}
```

Figura 4.19 – O problema do Produtor-Consumidor com passagem de mensagens.

Na Figura 4.19, assume-se que o mesmo código estará presente em dois computadores em uma rede. De acordo com o identificador do processo, valor que pode ser obtido pela rotina obter_numero_processo(), o código do Consumidor (identificador = 0) ou do Produtor (identificador = 1) será executado. Este paradigma é chamado Single Program Multiple Data (SPMD), onde cada processador da rede possui uma cópia do programa executável, mas processos diferentes executarão instruções diferentes dependendo da identificação do processo.

O Consumidor inicia enviando N mensagens vazias para o Produtor. Sempre que o Produtor possui um item para dar para o Consumidor, ele toma uma mensagem vazia e envia de volta para o Consumidor uma mensagem preenchida. Desta forma, o número total de mensagens permanece constante no tempo, de forma que elas podem ser armazenadas em uma determinada quantidade de memória.

Se o Produtor trabalha mais rápido do que o Consumidor, todas as mensagens terminarão preenchidas, esperando pelo Consumidor. O Produtor, então, será bloqueado e esperará uma nova mensagem vazia do Consumidor. Se o Consumidor trabalha mais rápido, ocorre o contrário: todas as mensagens estarão vazias esperando para que o Produtor as preencha. O Consumidor ficará bloqueado até que o Produtor envie uma mensagem preenchida.

Capítulo 5
Escalonamento de Processos

O escalonamento de processos se refere a como os processos são distribuídos para execução nos processadores em um Sistema de Computação. De acordo com Tanenbaum: "Quando mais de um processo é executável, o Sistema Operacional deve decidir qual será executado primeiro. A parte do Sistema Operacional dedicada a esta decisão é chamada escalonador (scheduler) e o algoritmo utilizado é chamado algoritmo de escalonamento (scheduling algorithm)." A forma com que ocorre o escalonamento é, em grande parte, responsável pela produtividade e eficiência atingidas por um Sistema de Computação. Mais do que um simples mecanismo, o escalonamento deve representar uma política de tratamento dos processos que permita obter os melhores resultados possíveis em um sistema.

5.1 Escalonamento Preemptivo versus Não Preemptivo

Um algoritmo de escalonamento é dito não preemptivo quando o processador designado para um certo processo não pode ser retirado deste até que o processo seja finalizado. Por outro lado, um algoritmo de escalonamento é considerado preemptivo quando a CPU designada para um processo pode ser retirada deste em favor de um outro processo.

Algoritmos preemptivos são mais adequados para sistemas em que múltiplos processos requerem atenção do sistema, ou seja, no caso de sistemas multiusuário interativos (sistemas de tempo compartilhado) ou em sistema de tempo real. Nestes casos, a preempção representa a mudança do processo em execução. Deste modo, para que a CPU seja retirada de um processo, interrompendo a execução deste, e designada a outro processo, anteriormente interrompido, é fundamental que ocorra a mudança de contexto dos processos, previamente mencionada no Capítulo 4. Tal mudança exige que todo o estado de execução de um processo seja adequadamente armazenado para sua posterior recuperação, representando uma sobrecarga computacional para a realização desta mudança e armazenamento de tais dados. Usualmente, os algoritmos preemptivos são mais complexos, dada a natureza imprevisível dos processos.

Por sua vez, os algoritmos não preemptivos são mais simples, se comparados aos preemptivos, e adequados para o processamento não interativo, como no caso do Processamento em Lote. Embora não proporcionem interatividade, são geralmente mais eficientes e previsíveis quanto ao tempo de entrega de suas tarefas.

5.2 Qualidade do Escalonamento

A qualidade do serviço oferecido por um algoritmo de escalonamento pode ser avaliada por meio de um critério simples: o tempo de permanência (tp), dado pela soma do tempo de espera (te) com o tempo de serviço ou execução (ts), conforme mostrado a seguir.

t +

$$t_{permanência} = t_{espera} + t_{serviço} \Rightarrow t_p = t_e + t_s$$

De um modo geral, deseja-se que o tempo de permanência seja o menor possível. A Figura 5.1 mostra uma representação gráfica do tempo de permanência.

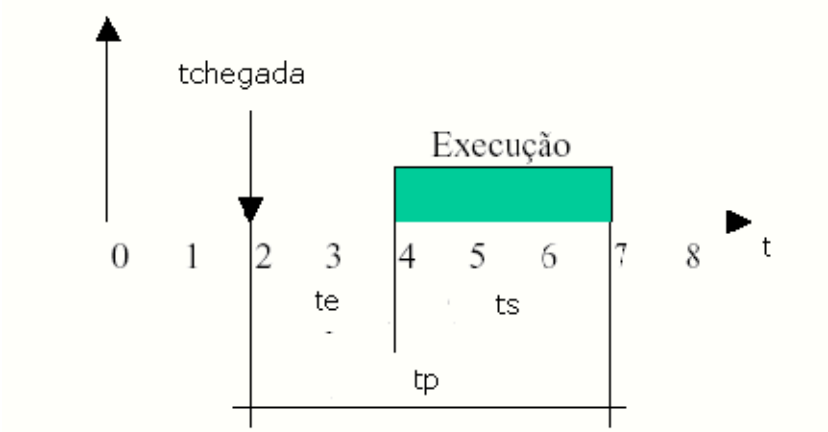


Figura 5.1 – Representação gráfica do tempo de permanência.

Uma outra forma de avaliar o escalonamento é utilizando o tempo de permanência normalizado (tpn), ou seja, a razão entre o tempo de permanência e o tempo de serviço conforme mostrado a seguir.

$$t_{pn} = \frac{t_p}{t_s} = \frac{t_e + t_s}{t_s}.$$

Se a espera for zero (melhor situação possível, pois indica que o processo ao chegar logo foi atendido), ter-se-á que o tempo de permanência normalizado de um processo será 1. Assim sendo, valores maiores indicam um pior serviço oferecido pelo algoritmo de escalonamento.

5.2 Algoritmos de Escalonamento

Existem vários algoritmos de escalonamento cujo objetivo principal é alocar o processador para um certo processo, dentre vários processos existentes, otimizando um ou mais aspectos do comportamento geral do sistema. Quatro algoritmos de escalonamento serão abordados neste capítulo: FIFO (First In, First Out), Round Robin, Shortest Job First e Multilevel Feedback Queues (Filas Multíniveis Realimentadas).

5.2.1 Escalonamento First In, First Out

Esta é a forma mais simples de escalonamento. No escalonamento FIFO (First In, First Out – Primeiro a entrar, Primeiro a sair), os processos prontos são colocados em uma fila organizada por ordem de chegada. No seu momento, cada processo recebe o uso da CPU até que a sua execução total seja completada, ou seja, o processo permanece em execução até que seja finalizado.

Os demais processos na fila de prontos ficam esperando o seu momento de ser executado pelo processador. Desta maneira, o escalonamento FIFO é um algoritmo não preemptivo. A Figura 5.2 mostra o esquema do escalonamento FIFO.

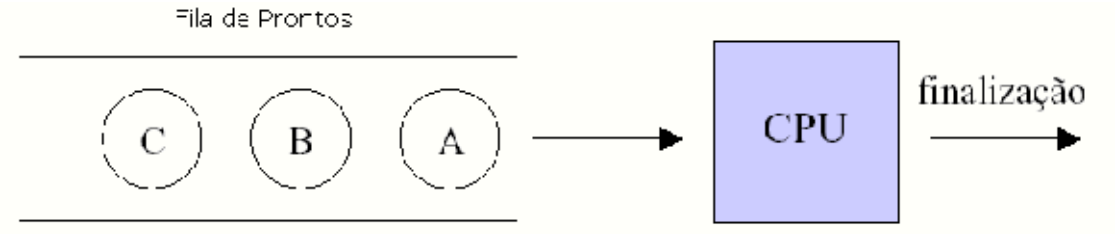


Figura 5.2 – Escalonamento FIFO.

Embora haja igual tratamento para todos os processos, ocorre que processos de pequena duração não são favorecidos pois, dependendo da quantidade de processos que existem para serem processados antes deles e da duração de cada um destes processos, o serviço oferecido para estes processos menores pode ser consideravelmente ruim.

Outro ponto é que processos importantes podem ficar esperando devido à execução de outros processos menos importantes, dado que o escalonamento FIFO não possui qualquer mecanismo de distinção entre processos (por exemplo, processos com diferentes níveis de prioridade).

A título de exemplificação, seja o caso de quatro processos A, B, C e D com tempos de serviço distintos, 3, 35, 12 e 4 segundos respectivamente, os quais são escalonados conforme a sua chegada pelo SO. A Figura 5.3 mostra os valores dos parâmetros desta situação considerando o escalonamento FIFO.

Processo	Tempo de Chegada	ts	te	tp	tpn
A	0	3	0	3	1,00
B	1	35	2	37	1,06
C	2	12	36	48	4,00
D	3	4	47	51	12,75

Figura 5.3 – Exemplo considerando o escalonamento FIFO.

Na Figura 5.3, percebe-se que, em função de como ocorreu o escalonamento, um processo pequeno, como o processo D, obteve o pior serviço.

5.2.2 Escalonamento Round Robin

No escalonamento Round Robin ou circular, os processos também são organizados em uma fila segundo a ordem de chegada e então são despachados para execução pelo processador. Entretanto, ao invés de serem executados até o final, a cada processo é concedido apenas um pequeno intervalo de tempo, denominado fatia de tempo, time-slice, quantum de tempo ou quantum. Caso o processo não seja finalizado neste intervalo de tempo, ocorre sua substituição pelo próximo processo na fila de processos prontos, sendo o processo interrompido colocado no final da fila. Isto significa que, ao final da fatia de tempo do processo, ocorre a preempção do processador, ou seja, o processador é designado a outro processo, sendo salvo o contexto do processo interrompido para permitir a continuidade da execução deste processo quando chegar o seu momento novamente.

O escalonamento Round Robin se baseia na utilização de temporizadores, sendo um algoritmo preemptivo e bastante adequado para ambientes interativos, ou seja, em sistemas de tempo compartilhado onde existem múltiplos usuários simultâneos sendo, portanto, necessário garantir tempos de resposta razoáveis. A Figura 5.4 mostra o esquema do escalonamento Round Robin.

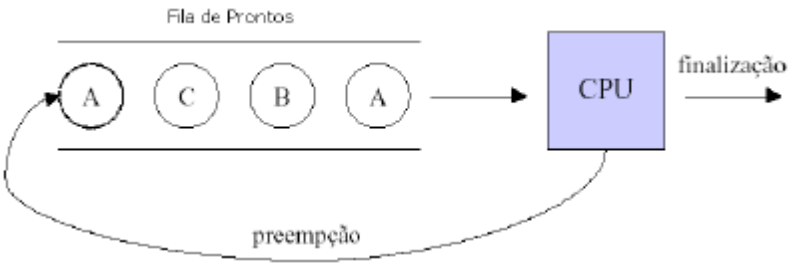


Figura 5.4 – Escalonamento Round Robin.

O tempo de resposta (tr) pode ser visto como o tempo que leva para que um certo processo possa ser atendido pelo processador. A Figura 5.5 mostra a ilustração deste conceito, considerando que existam apenas 3 processos no sistema e que a janela de tempo de cada

processo seja de 10 ms. O conceito de janela de tempo do processo será explicado logo adiante nesta seção.

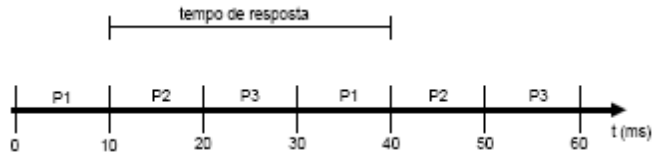


Figura 5.5 – Tempo de resposta.

Considerando o escalonamento Round Robin, na Figura 5.5 percebe-se que o processo P2 tomou posse da CPU em t = 10 ms, foi desescalonado em t = 20 ms e voltou a ter a posse da CPU em t = 40 ms. Para esta situação, o tempo de resposta vale 30 ms (40 ms – 10 ms).
A sobrecarga imposta pela mudança de contexto representa um investimento para que seja conseguido um bom nível de eficiência.

É importante perceber que o escalonamento mostrado no Capítulo 2 é justamente um escalonamento Round Robin, mas desconsiderou-se, até aquele momento, a sobrecarga devido à mudança de contexto.

A determinação do tamanho da fatia de tempo (quantum) é extremamente importante pois relaciona-se com a sobrecarga imposta ao sistema pelas mudanças de contexto dos processos em operação.

Para cada processo, existe uma janela de tempo onde ocorrem:
1.) a recuperação do contexto do processo (trc);
2.) a execução do processo pela duração do quantum (q);
3.) a preservação do contexto do processo após o término do seu quantum (tpc).
A Figura 5.6 mostra os tempos associados à janela de tempo do processo.

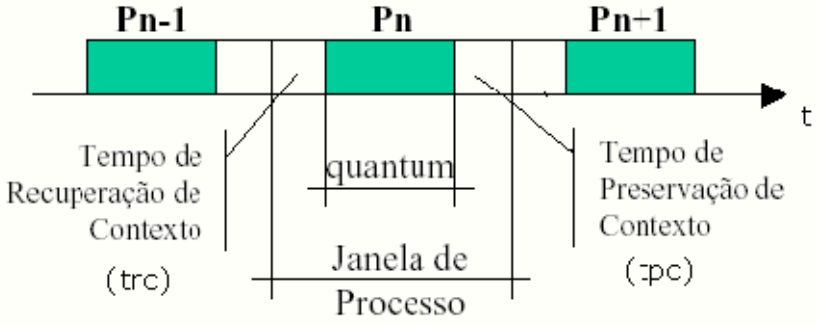


Figura 5.6 – Janela de tempo do processo.

Como o tempo gasto para a mudança de contexto (tc) não é útil do ponto de vista de processamento de processos dos usuários tem-se que, para cada janela de tempo concedida aos processos, existe um desperdício de tempo de processamento devido a sobrecarga, identificada pelos tempos trc e tpc na Figura 5.6.
Dado que a mudança de contexto toma um tempo aproximadamente constante, tem-se que a sobrecarga (s) pode ser calculada por meio da seguinte relação:

$$s = \frac{t_c}{t_c + q}, \text{ onde:}$$

tc = trc + tpc, é o tempo associado à mudança de contexto; q = tamanho do quantum.

Como exemplo, se tc = 2 ms e q = 8 ms, então apenas 80% do tempo de processamento é útil, sendo a sobrecarga imposta pela mudança de contexto de 20% do processamento útil.

Se o quantum for aumentado, então a sobrecarga da mudança de contexto é diminuída, mas um número de usuários (nu) menor será necessário para que os tempos de resposta se tornem maiores e mais perceptíveis. Diminuindo o quantum, existe uma situação inversa de maior

sobrecarga e também de um maior número possíveis de usuários sem degradação sensível dos tempos de resposta. As equações a seguir mostram a relação existente entre o tempo de resposta (tr), o número de usuários (nu) e o quantum do sistema (q).

$$t_r = nu * (q + t_c) \Leftrightarrow nu = \frac{t_r}{q + t_c}.$$

A Figura 5.7 mostra um exemplo da sobrecarga e do número de usuários possível para um tempo de resposta (tr) de 1 s e um tempo de mudança de contexto (tc) de 2 ms, considerando diferentes valores de quantum.

quantum (q)	sobrecarga (s)	número de usuários (nu)
2 ms	50%	250
8 ms	20%	100
18 ms	10%	50
98 ms	2%	10

Figura 5.7 – Comporatamento da sobrecarga e do número de usuários para diferentes valores de quantum.

Valores usuais de quantum são 10 ms, 20 ms. Com o aumento da velocidade dos processadores, a mudança de contexto ocorre mais rapidamente diminuindo a sobrecarga e aumentando ligeiramente a quantidade de usuários possíveis para um mesmo limite de tempo de resposta.

5.2.3 Escalonamento Shortest Job First

O escalonamento SJF (Shortest Job First – Menor Job Primeiro) é uma variante do escalonamento FIFO onde os processos em espera pelo processador são organizados em um fila segundo seu tempo de serviço (ts), sendo colocados à frente os menores jobs, isto é, os que serão processados em intervalos de tempo menores.

Mesmo sendo uma forma de escalonamento não preemptivo, oferece a vantagem de proporcionar tempos médios de espera menores do que aqueles obtidos em um esquema FIFO. O grande problema deste esquema de escalonamento é que o tempo de processamento de um job não pode ser determinado antes de seu processamento, sendo necessário o uso de estimativas feitas pelo usuário ou programador, ainda assim, pouco precisas. A Figura 5.8 mostra o esquema do escalonamento SJF.

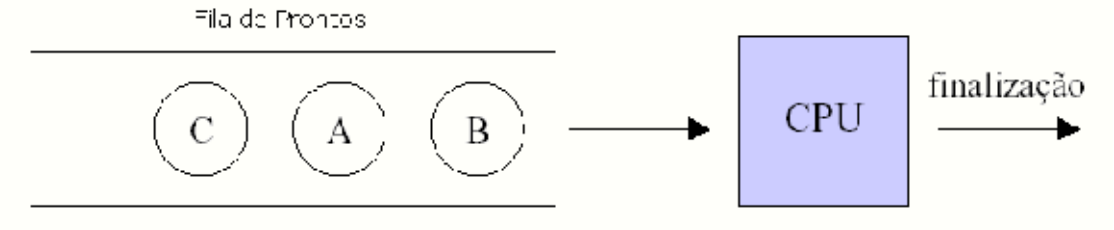


Figura 5.8 – Escalonamento SJF. No caso, o processo B é menor de todos.

Seja o mesmo caso usado para o escalonamento FIFO, onde quarto processos A, B, C e D com tempos de processamento distintos, respectivamente 3, 35, 12 e 4 segundos, são escalonados conforme sua duração pelo SO. A Figura 5.9 mostra esta situação.

Processo	ts	Tempo de Chegada	te	tp	tpn
A	3	0	0	3	1,00
D	4	3	0	4	1,00
C	12	2	5	17	1,42
B	35	1	18	53	1,51

Figura 5.9 – Exemplo considerando o escalonamento SJF.

Percebe-se na Figura 5.9 que, em função do escalonamento ordenar os processos segundo sua duração, os menores processos obtiveram o melhor serviço sem que isso resultasse em uma

degradação significativa para os processos maiores. O tempo de permanência normalizado possui valores bem inferiores aos obtidos com o escalonamento FIFO, evidenciando as qualidades do escalonamento SJF.

5.2.4 Escalonamento Multilevel Feedback Queues

O escalonamento MFQ (Multilevel Feedback Queues – Filas Multinível Realimentadas) é um interessante esquema de escalonamento baseado em várias filas encadeadas como mostra a Figura 5.10.

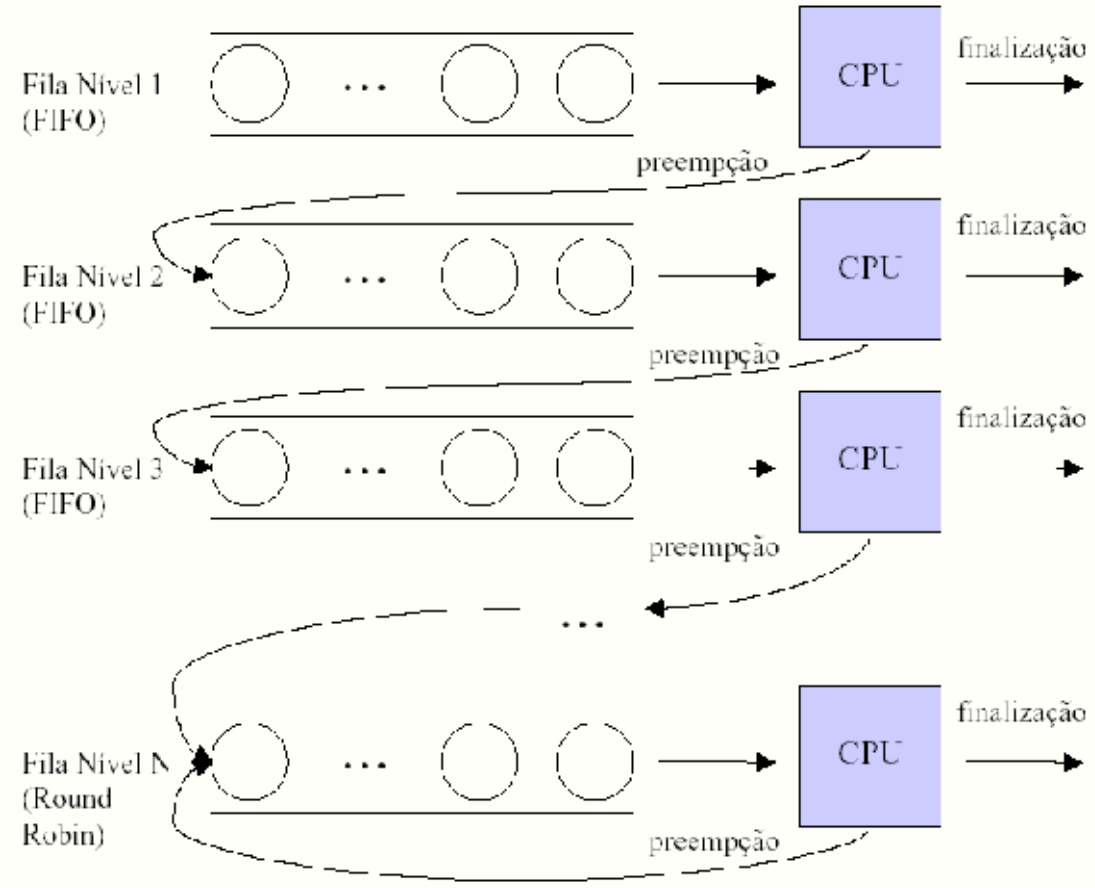


Figura 5.10 – Escalonamento MFQ.

Todos os novos processos são colocados inicialmente na Fila de Nível 1, a qual possui um comportamento FIFO. Ao utilizar a CPU podem ocorrer três situações:

1.) o processo é finalizado e então é retirado das filas;

2.) o processo solicita o uso de dispositivos de E/S, ficando bloqueado até que o pedido de E/S seja atendido, voltando para a mesma fila até que seu quantum se esgote;

3.) tendo esgotado seu quantum inicial de tempo, o processo é colocado no final da fila de Nível 2.

Nas filas seguintes, o mecanismo é o mesmo, embora os processos somente utilizem a CPU na sua vez e na ausência de processos nas filas de nível superior. Quando novos processos aparecem nas filas de nível superior ocorre a preempção dos processos nas filas de nível inferior, de forma a atender os processos existentes nas filas superiores. Portanto, o escalonamento MFQ possui níveis de prioridades sendo os processos que estão atualmente na Fila 1 os de mais alta prioridade.

A última fila apresenta um comportamento um pouco diferente: ela possui escalonamento Round Robin, onde os processos permanecem até que sejam finalizados.

As seguintes observações são relevantes neste tipo de escalonamento:

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

b.) processos com uso intenso de E/S são favorecidos, pois a utilização de E/S não os desloca para filas inferiores, até que o seu quantum termine;

c.) processos de duração maior também são favorecidos pois os quanta de tempo são progressivamente maiores nas filas de nível inferior. Portanto, se o quantum da Fila de Nível 1 for 10 ms, por exemplo, então o quantum da Fila 2 poderia ser 20 ms, o da Fila 3 poderia ser 40 ms e assim sucessivamente.

Um algoritmo de escalonamento deveria, no mínimo, favorecer os processos de curta duração, de forma a minimizar os tempos médios de resposta, e os processos que demandam por dispositivos de E/S para obter adequada utilização dos periféricos do sistema.

Deste modo, é importante que tais algoritmos avaliem a natureza dos processos em execução promovendo um escalonamento adequado.

Neste sentido, o escalonamento MFQ pode ser considerado adaptativo, ou seja, ele é capaz de perceber o comportamento de um processo, favorecendo-o por meio da recolocação em uma fila de nível adequado. O critério de recolocação de processos nas filas após a solicitação de uso de dispositivos de E/S influencia de forma considerável no quanto este esquema de escalonamento é adaptativo, isto é, o quanto ele é capaz de atender aos diferentes padrões de comportamento dos processos. Embora a sobrecarga para a administração deste esquema de escalonamento seja maior, o sistema torna-se mais sensível ao comportamento dos processos, separando-os em categorias (níveis de filas), possibilitando ganho de eficiência.

Capítulo 6
Sistemas de Arquivos

O armazenamento e a recuperação de informações são atividades essenciais para qualquer tipo de aplicação. Um processo deve ser capaz de ler e gravar grande volume de dados em dispositivos como fitas e discos de forma permanente, além de poder compartilhá-los com outros processos. É mediante a implementação de arquivos que o Sistema Operacional estrutura e organiza estas informações.

O Sistema Operacional controla as operações sobre os arquivos, organizando seu armazenamento no que é denominado sistema de arquivos. De acordo com Deitel, um sistema de arquivos geralmente possui:

- **método de acesso**: maneira como os dados são acessados nos arquivos;
- **gerenciamento de arquivos**: conjunto de mecanismos de armazenamento, referência, compartilhamento e segurança;
- **mecanismos de integridade**: objetivam assegurar que os dados de um arquivo permanecem íntegros.

Ainda segundo Deitel, um sistema de arquivos deve permitir, em termos funcionais, que:

- os usuários possam criar, modificar e eliminar arquivos, assim como duplicá-los ou transferir dados entre arquivos;
- operações de backup e sua recuperação sejam realizadas;
- seja possível a adoção ou implementação de procedimentos de proteção e segurança;
- a interface seja amigável e consistente, admitindo a realização de operações apenas por meio dos nomes simbólicos dos arquivos, devendo estas operações sempre ocorrer de maneira uniforme, independentemente dos diferentes dispositivos de armazenamento.

Percebe-se que os sistemas de arquivos se preocupam com a organização e controle do armazenamento secundário de um sistema de computação. É importante mencionar que um SO pode suportar diversos sistemas de arquivos, isto é, o SO pode operar usando diferentes formas de administração dos dispositivos de armazenamento secundário. A Figura 6.1 mostra exemplos de SOs e de sistemas de arquivo suportados.

Sistema Operacional	Sistemas de Arquivos Suportados
MS-DOS	FAT (File Allocation Table) 12 e 16 bits
MS-Windows 95/98	FAT (File Allocation Table) 12 e 16 bits
	VFAT (File Allocation Table) 32 bits
MS-Windows NT	FAT (File Allocation Table) 12 e 16 bits
	VFAT (File Allocation Table) 32 bits
	NTFS (New Technology File System)
IBM OS/2	FAT (File Allocation Table) 12 e 16 bits
	HPFS (High Performance File System)
Linux	FAT (File Allocation Table) 12, 16 e 32 bits
	VFAT (File Allocation Table) 32 bits
	Minix (Mini Unix)
	Extended FS
	Ext2
	Ext3

Figura 6.1 – Exemplos de SOs e sistemas de arquivos suportados.

6.1 Conceitos de Arquivos

Entre as várias definições possíveis, Tanenbaum afirma, de forma simplificada, que os arquivos podem ser entendidos como sequências de Bytes não interpretadas pelo sistema, dependendo de aplicações apropriadas para sua correta utilização.

Os arquivos, dependendo do SO e do sistema de arquivos em uso, podem possuir uma identificação, atributos, lista de controle de acesso e uma organização ou tipo.

Para que os dados possam ser corretamente identificados e utilizados, o sistema de arquivos deve prover um mecanismo de identificação dos arquivos, isto é, uma forma de distinção simples dos arquivos, que permita sua adequada operação. Isto é feito por meio da associação de um nome ao arquivo propriamente dito. Deste modo, muitas das operações que podem ser feitas sobre os arquivos são especificadas em termos de seus nomes, simplificando muitas tarefas. A Figura 6.2 mostra formas de denominação diferentes em alguns Sistemas Operacionais.

Sistema Operacional	Composição do Nome	Composição da Extensão
MS-DOS (FAT12 e FAT16)	1 a 8 caracteres	1 a 3 caracteres
MS-Windows 95/98 (VFAT)	1 a 255 caracteres	1 a 255 caracteres
MS-Windows NT (NTFS)	1 a 255 caracteres	1 a 255 caracteres
IBM OS/2 (HPFS)	1 a 255 caracteres	1 a 255 caracteres

Figura 6.2 – Denominação de arquivos em diversos SOs.

Além dos nomes, os arquivos podem possuir atributos, isto é, informações de controle que não fazem parte dos arquivos, embora associadas aos mesmos. Dependendo do sistema de arquivos, os atributos variam, porém alguns, como tamanho do arquivo, proteção, identificação do criador e data de criação, estão presentes em quase todos os sistemas.

Alguns atributos especificados na criação dos arquivos não podem ser modificados em função de sua própria natureza, como organização e data/hora de criação. Outros são alterados pelo próprio SO, como tamanho e data/hora do último backup realizado.

Existem ainda atributos que podem ser modificados pelo próprio usuário, como proteção do arquivo, tamanho máximo e senha de acesso. A Figura 6.3 apresenta os principais atributos presentes nos sistemas de arquivos de uma maneira geral.

Atributos	Descrição
Tamanho	Especifica o tamanho em Bytes
Proteção	Código de proteção de acesso
Proprietário	Identifica o criador do arquivo
Criação	Data e hora de criação do arquivo
Backup	Data e hora do último backup
Organização	Organização lógica dos registros
Senha	Senha para acessar o arquivo

Figura 6.3 – Principais atributos de arquivos.

As listas de controle de acesso são relações de usuário que podem realizar operações específicas sobre um dado arquivo, tal como execução ou escrita, e estão associadas diretamente aos arquivos. Por exemplo, os sistemas Unix utilizam-se de uma lista de acesso simplificada baseada no conceito de grupos. Permissões são associadas a todos os arquivos e podem ser diferenciadas em 3 níveis: o do proprietário, o do grupo ao qual pertence o seu proprietário e dos demais usuários. Para cada nível, podem ser especificadas separadamente permissões para leitura, escrita e execução, que quando combinadas resultam em diferentes modos de acessar os arquivos.

Existem diferentes organizações ou estruturas possíveis para os arquivos dentre as quais pode-se citar: sequencial e de registro. A Figura 6.4 mostra tais estruturas.

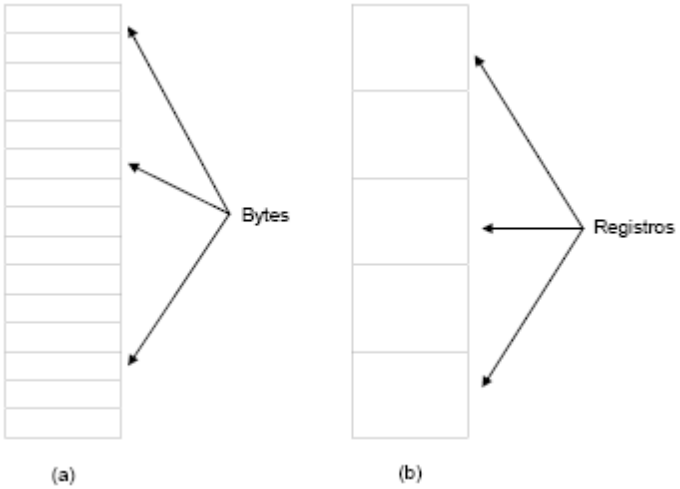


Figura 6.4 – Estruturas de arquivos. (a) Estrutura Sequencial. (b) Estrutura de Registro.

Na estrutura sequencial, um arquivo é uma sequência de Bytes, garantindo a máxima flexibilidade ao usuário do SO o qual oferece operações bastante simples. Na estrutura de registro, um arquivo é uma sequência de registros, os quais podem ter diversos formatos.

Tal estrutura é menos flexível do que a sequencial e depende de operações um pouco mais sofisticadas do SO. Existem diversos arranjos para arquivos organizados como sequências de registros, podendo haver registros de tamanho fixo e registros de tamanho variável.

Sistemas Operacionais como o DOS, Windows 95/98/NT e o Unix estruturam seus arquivos como uma sequência de Bytes. Computadores de grande porte, como os sistemas IBM, utilizam sequências de registros como organização comum.

Dependendo do conteúdo do arquivo, este pode ser entendido como de um certo tipo. Muitos SOs utilizam a extensão do nome do arquivo como uma referência para o tipo de seu conteúdo. Nos SOs Windows 98/NT/2000/XP podem ser associadas aplicações a determinadas extensões, auxiliando o usuário. É comum o usuário selecionar 2 vezes em um arquivo com extensão .doc, no Windows XP por exemplo, e automaticamente é invocado o Microsoft Word abrindo o arquivo selecionado pelo usuário. De uma maneira geral, pode-se dividir os tipos de arquivos em dois: arquivos alfanuméricos e arquivos binários. Um arquivo alfanumérico possui apenas caracteres definidos por algum código como o código ASCII. Um arquivo tipo .txt, gerado pelo Bloco de Notas (Notepad), é um exemplo de um arquivo alfanumérico. Um arquivo binário pode conter qualquer tipo de caractere. O conteúdo de um arquivo binário precisa ser interpretado por uma aplicação para poder ser visualizado. Os arquivos executáveis são também arquivos binários, cuja execução é realizada diretamente pelo SO.

O sistema de arquivos oferece um conjunto de System Calls que permite às aplicações realizar operações de E/S, como tradução de nomes em endereços, leitura e gravação de dados e criação/eliminação de arquivos. A Figura 6.5 apresenta algumas System Calls encontradas na maioria das implementações para gerenciamento de arquivos.

System Call	Descrição
CREATE	Criação de um arquivo
OPEN	Abertura de um arquivo
READ	Leitura de dados de um arquivo
WRITE	Gravação de dados em um arquivo
CLOSE	Fechamento de um arquivo

RENAME	Alteração do nome de um arquivo
DELETE	Eliminação de um arquivo

Figura 6.5 – System Calls para gerenciamento de arquivos.

6.2 Organização de Diretórios

A estrutura de diretórios é o modo como o sistema organiza logicamente os diversos arquivos contidos em um disco. O diretório é uma estrutura de dados que contém entradas associadas aos arquivos onde são armazenadas informações como localização física, nome, organização e demais atributos.

Quando um arquivo é aberto, o SO procura a sua entrada na estrutura de diretórios, armazenando as informações sobre atributos e localização do arquivo em uma tabela mantida na memória principal. Esta tabela contém todos os arquivos abertos, sendo fundamental para aumentar o desempenho das operações com arquivos. É importante que ao termino do uso do arquivo, este seja fechado, ou seja, que se libere o espaço na tabela de arquivos abertos.

6.2.1 Organização em Nível Único

A implementação mais simples de uma estrutura de diretórios é chamada de nível único. Neste caso, existe apenas um único diretório contendo todos os arquivos do disco. Este modelo é bastante limitado, já que não permite que usuários criem arquivos com o mesmo nome, o que ocasionaria um conflito no acesso aos arquivos. Uma vantagem deste tipo de organização é a simplicidade de implementação e como desvantagens pode-se citar a ineficiência quando existir mais de um usuário e a preocupação com arquivos de outros usuários. A Figura 6.6 mostra a organização em nível único.

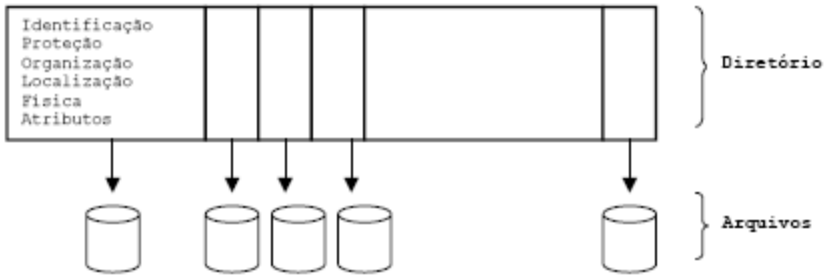


Figura 6.6 – Organização de diretório em nível único.

Esta estrutura de diretório em nível único foi usada nos primeiros computadores.

6.2.2 Organização em Dois Níveis

Como a organização de nível único é bastante limitada, foi implementada uma estrutura onde, para cada usuário, existia um diretório particular denominado User File Directory (UFD) . Com esta implementação, cada usuário poderia criar arquivos com qualquer nome, sem a preocupação de conhecer os demais arquivos do disco.

Para que o sistema possa localizar arquivos nesta estrutura, deve haver um nível de diretório adicional para controlar os diretórios individuais. Este nível, denominado Master File Directory (MFD) , é indexado pelo nome do usuário e nele cada entrada aponta para o diretório pessoal. A Figura 6.7 mostra a Organização em dois níveis.

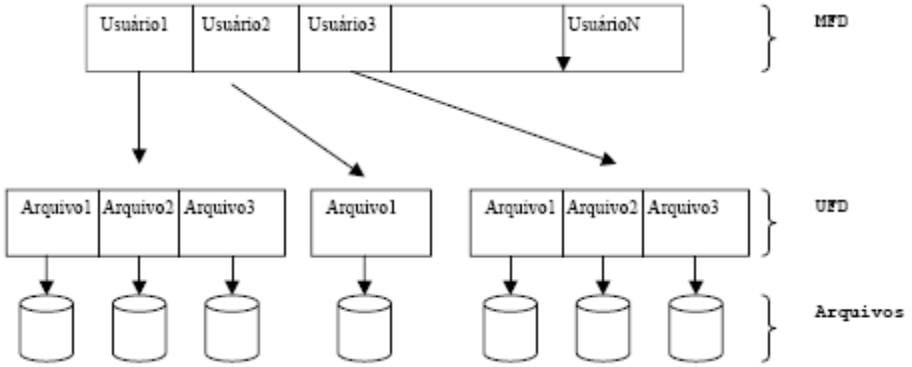


Figura 6.7 – Organização de diretório em dois níveis.

A estrutura de diretórios em dois níveis é análoga a uma estrutura de dados em árvore, onde o MFD é a raiz, os galhos são os UFD e os arquivos são as folhas. Neste tipo de estrutura, quando se referencia um arquivo, é necessário especificar o seu nome, bem como o diretório onde ele se encontra. Esta referência é chamada de PATH (caminho). Como exemplo, caso o usuário CARLOS necessite acessar seu próprio arquivo denominado DOCUMENTO.TXT, ele pode referenciá-lo como /CARLOS/DOCUMENTO.TXT. Cada sistema de arquivos possui sua própria sintaxe para especificar diretórios e arquivos.

Como vantagens deste sistema, pode-se mencionar que o usuário tem melhor controle sobre seus arquivos, existem poucas chances de um usuário interferir no outro e usuários distintos podem ter arquivos de mesmo nome. Como desvantagens, ocorre um isolamento de um usuário de outros, dificultando o trabalho conjunto. Além disto para acessar arquivos de outro usuário, deve-se indicar, além do nome do arquivo, o usuário que o contém.

Para evitar a geração de cópias de arquivos que são de uso comum entre usuários, tais como utilitários, editores, compiladores etc., utiliza-se do seguinte artifício:

- é criado e informado ao sistema um subdiretório especial;
- quando o sistema vai buscar um arquivo, ele procura primeiro no diretório atual; se não achar, ele busca no subdiretório indicado como especial.

6.2.3 Organização em Árvores

Pelo ponto de vista do usuário, a organização dos seus arquivos em único diretório não permite uma organização adequada.

A extensão do modelo de dois níveis para um de múltiplos níveis permitiu que os arquivos fossem logicamente melhor organizados.

Este novo modelo, chamado organização de diretórios em árvores, é atualmente adotado por um grande número de SOs.

Na estrutura em árvore, é possível ao usuário criar quantos diretórios desejar, podendo um diretório conter arquivos ou outros diretórios (chamados subdiretórios). Cada arquivo nesta estrutura possui um PATH único, que descreve todos os diretórios desde a raiz (MFD) até o diretório no qual o arquivo está ligado. A Figura 6.8 mostra este tipo de organização.

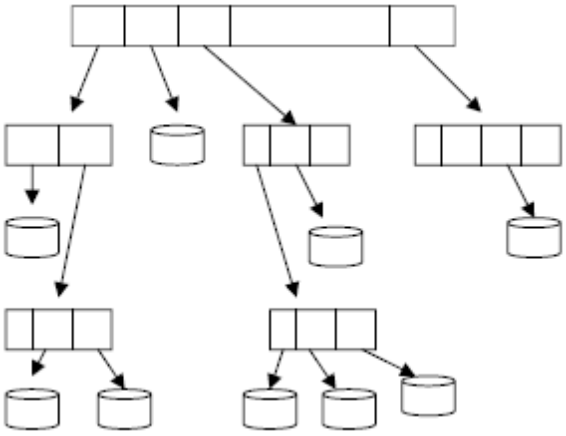


Figura 6.8 – Organização de diretórios em árvores.

Como vantagens, pode-se citar o fato de poder-se organizar os arquivos não apenas por usuários, mas por projetos e, também, de cada usuário poder organizar os seus arquivos em subdiretórios.

Na maioria dos sistemas, os diretórios também são tratados como arquivos, possuindo identificação e atributos, como proteção, identificação do criador e data de criação.

6.3 Sistemas Baseados em Disco

A grande vantagem do disco está na possibilidade de se acessar a informação diretamente. As unidades de leitura ou gravação de discos magnéticos constam dos seguintes elementos:

• Um dispositivo de tração que mantém os discos em contínuo movimento e a velocidade constante;

• um pente de cabeças de leitura/gravação (uma por face magnetizável) capaz de mover-se radialmente sobre a superfície dos discos. Em alguns casos de unidades rápidas de discos, existem mais de uma cabeça por face magnetizável para que o tempo de acesso à informação seja menor. Estes casos são típicos do uso do disco como memória virtual. A Figura 6.9 mostra a estrutura dos sistemas baseados em disco.

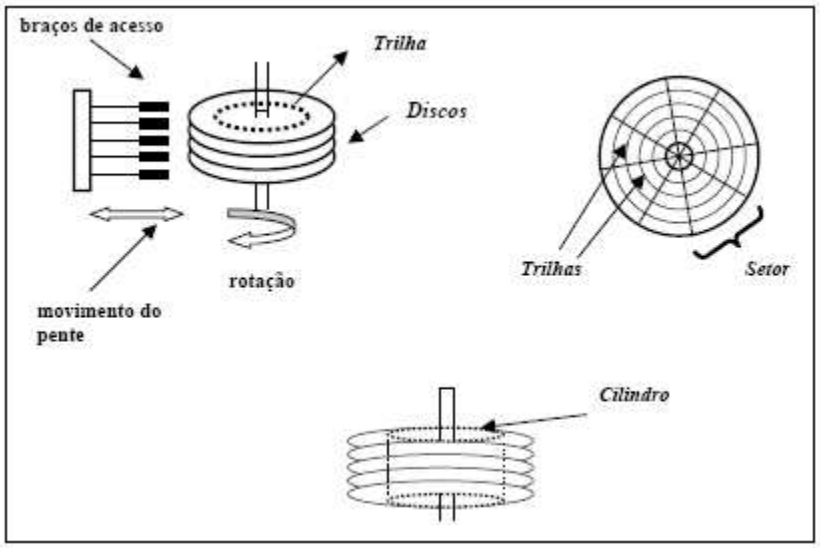


Figura 6.9 – Estrutura de sistemas baseados em disco.

Fisicamente, para a controladora, a unidade de armazenamento no disco é o setor. Um setor possui 512 Bytes. Para se acessar uma informação no disco, é preciso fornecer à controladora o seguinte:

• face;
• trilha ou cilindro;
• setor dentro da trilha.

A controladora enxerga o disco por meio dos cilindros que é o conjunto de trilhas sobrepostas, faces e setores. O SO, por sua vez, enxerga os setores de forma contínua, ignorando as trilhas e faces. A esses setores contínuos dá-se o nome de blocos contínuos ou registros físicos.

6.3.1 Conversão de Trilha/Setor em Bloco Contínuo

O SO estabelece uma relação da parte física do disco com a sua parte lógica, por meio do cálculo de trilha/setor em bloco contínuo.

Como exemplo, considere um disco contendo duas faces e 16 setores por trilha, sendo 8 setores de cada lado. Qual o bloco correspondente ao setor de número 4 da terceira trilha na segunda face do disco? Neste exemplo, existem duas trilhas anteriores àquela que contém o bloco procurado; portanto são 16 x 2 = 32 setores contidos nestas duas trilhas. O bloco está na terceira trilha da segunda face e, portanto, existem 8 setores na primeira face pertencente a terceira trilha. Tem-se, desta forma, 32 + 8 = 40 setores que antecedem ao bloco desejado. Como o bloco corresponde ao quarto setor da trilha e, supondo que a numeração dos blocos começa em 1, o número do bloco desejado é:

```
40 + 4 = 44.
```

Para o SO, muitas vezes é conveniente trabalhar com unidades de alocação de tamanho diferente de 1 setor. Por exemplo, discos com alta capacidade podem trabalhar, segundo o SO, com unidades maiores, para aumentar a velocidade de trabalho. Existe, então, o conceito de registros lógicos em contraposição aos registros físicos (blocos contínuos). Um registro lógico é composto de vários registros físicos conforme mostra a Figura 6.10.

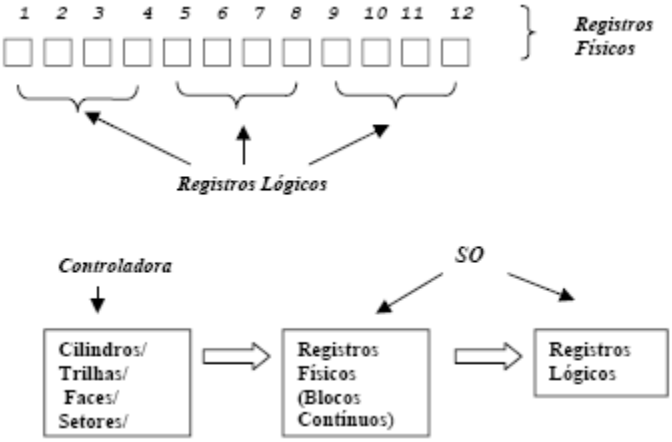


Figura 6.10 – Registros Lógicos e Físicos e como a controladora e o SO visualizam o disco.

A Figura 6.11 apresenta uma relação da capacidade de disco e do tamanho de registro lógico.

Capacidade de Disco	Tamanho do Registro Lógico
Até 1 MByte	2 setores contíguos
Entre 1 e 16 MBytes	4 setores contíguos
Entre 16 e 32 MBytes	8 setores contíguos
Acima de 32 MBytes	16 setores contíguos

Figura 6.11 – Capacidade de disco x tamanho do registro lógico.

Um registro lógico grande tem como vantagem o aumento da velocidade de uso do disco e, como desvantagem, o aumento de espaço perdido por registro.

6.4 Métodos de Acesso

Como as memórias secundárias são vistas pelo SO como um conjunto de unidades de alocação (registros lógicos), existem mecanismos para gerenciamento de tais unidades com o objetivo de melhorar o desempenho do SO (velocidade, aproveitamento e confiabilidade). Os pontos a seguir devem ser considerados na análise dos diferentes casos de alocação de espaço em disco:

a) os métodos são basicamente para aplicação em discos;

b) os registros lógicos são chamados simplesmente de blocos;

c) um disco é dividido em n blocos, começando do bloco número 1;

d) Um arquivo também é dividido em blocos, iniciando pelo bloco de número 1. Entretanto, o primeiro bloco do arquivo não corresponde necessariamente ao primeiro bloco do disco.

A observação *d* pode ser mais bem entendida por meio da Figura 6.12 a seguir.

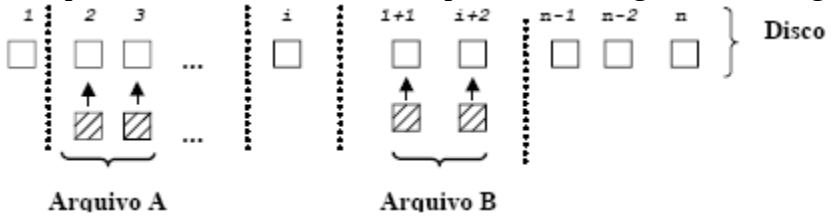


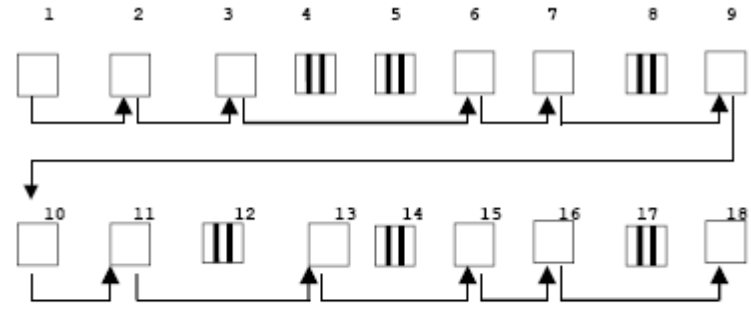
Figura 6.12 – Blocos em arquivos e em disco.

6.4.1 Gerenciamento de Espaço Livre em Disco

Como os discos têm a propriedade de poderem ser reutilizados, os arquivos podem ser criados e apagados com frequência. Portanto, para melhor reutilização de espaço em disco, faz-se necessário um controle sobre seu espaço livre. A seguir, serão apresentadas algumas técnicas para gerenciamento de espaço livre em disco.

6.4.1.1 Lista Ligada ou Encadeada

A idéia para realizar este controle é por meio da ligação encadeada de todos os blocos livres do disco. Para que isto seja possível, cada bloco possui uma área reservada para armazenamento do endereço do próximo bloco. A partir do primeiro bloco livre, pode-se ter acesso sequencial aos demais de forma encadeada. Este esquema apresenta algumas restrições se for considerado que, além do espaço utilizado no bloco com informação de controle, o algoritmo de busca de espaço livre sempre deve realizar uma pesquisa sequencial na lista. A Figura 6.13 mostra um exemplo usando esta técnica.



(a)

LISTA LIGADA

01	02	03	04	05	06	07	08	09
2	3	6	-1	-1	7	9	-1	10

10	11	12	13	14	15	16	17	18
11	13	-1	15	-1	16	18	-1	0

Figura 6.13 – Um exemplo usando a técnica de lista ligada.

(a): situação em termos de blocos ocupados e livres. (b): lista ligada.

Na Figura 6.13, o valor –1 indica que o bloco está ocupado e o valor 0 significa que este é o último bloco livre do disco.

6.4.1.2 Lista de Blocos Contíguos

A idéia por trás desta técnica é guardar o endereço do primeiro bloco livre de uma série de blocos contíguos (adjacentes). Considere o exemplo mostrado no tópico anterior (6.4.1.1). A lista de blocos contíguos é mostrada na Figura 6.14.

Primeiro Bloco Livre	Número de Blocos Contíguos
1	3
6	2
9	3
13	1
15	2
18	1

Figura 6.14 – Lista de blocos contíguos para o exemplo do tópico anterior.

6.4.2 Alocação de Espaço Para Arquivos

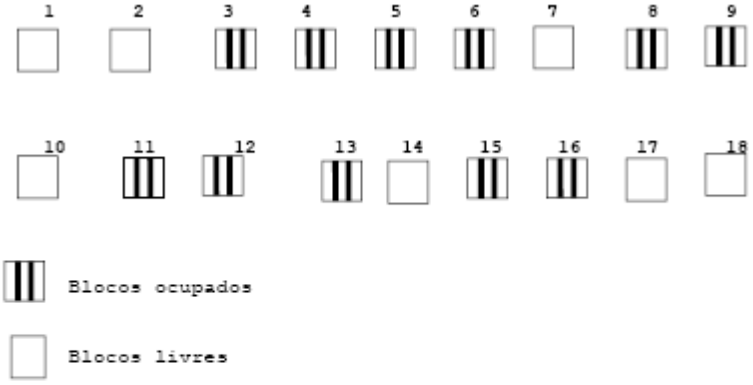
Uma outra tarefa do sistema de arquivos é alocar espaço para novos arquivos no disco. Existem formas diferentes desta tarefa ser realizada. A seguir serão descritos alguns métodos associados a este assunto.

6.4.2.1 Alocação Contígua

A alocação contígua consiste em armazenar um arquivo em blocos sequencialmente dispostos no disco. Neste tipo de alocação, o sistema localiza um arquivo por meio do endereço do primeiro bloco e da sua extensão em bloco.

O acesso a arquivos dispostos contiguamente no disco é bastante simples, tanto para a forma sequencial quanto para a forma direta. Seu principal problema é a alocação de espaço livre para

novos arquivos. Caso um arquivo deva ser criado com n blocos, é necessário que exista uma cadeia de n blocos dispostos sequencialmente no disco. A Figura 6.15 mostra um exemplo do uso desta técnica.



(a)

TABELA DE BLOCOS CONTÍGUOS

Arquivo	Bloco	Extensão
Arq1.txt	3	4
Arq2.txt	8	2
Arq3.txt	11	3
Arq4.txt	15	2

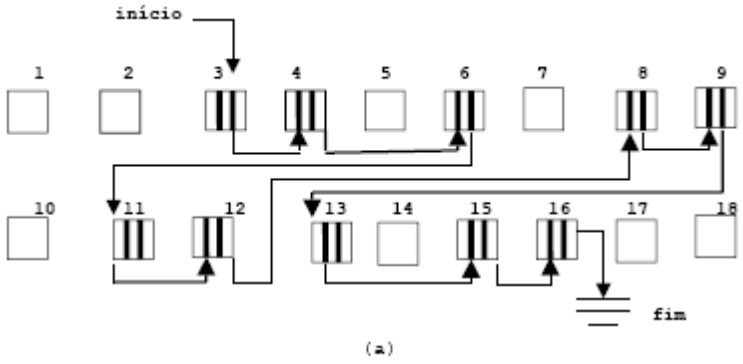
Figura 6.15 - Um exemplo usando a técnica de alocação contígua. (a): situação em termos de blocos ocupados e livres. (b): tabela de blocos contíguos.

A alocação contígua apresenta um problema chamado fragmentação dos espaços livres. Como os arquivos são criados e eliminados frequentemente, os segmentos livres acabam se fragmentando em pequenos pedaços por todo o disco. O problema pode tornar-se crítico quando um disco possui blocos livres disponíveis, porém não existe um segmento contíguo em que o arquivo possa ser alocado. No exemplo mostrado na Figura 6.15, suponha que seja desejado alocar espaço para um arquivo que ocupa 3 blocos. Embora haja 7 blocos livres, não seria possível alocar o arquivo pois não existem 3 blocos contíguos.

O problema da fragmentação pode ser contornado por meio de rotinas que reorganizem todos os arquivos no disco de maneira que só exista um único segmento de blocos livres. Este procedimento, denominado desfragmentação, geralmente utiliza uma área de trabalho no próprio disco ou uma fita magnética. Existe um grande consumo de tempo nesse tipo de operação, sendo importante ressaltar que a desfragmentação é um procedimento com efeito temporário e deve, portanto, ser realizada periodicamente.

6.4.2.2 Alocação Encadeada

Nesta alocação, um arquivo pode ser organizado como um conjunto de blocos ligados logicamente no disco, independentemente da sua localização física. Cada bloco deve possuir um ponteiro para o bloco seguinte do arquivo. A Figura 6.16 mostra um exemplo usando esta técnica de alocação.



LISTA ENCADEADA

01	02	03	04	05	06	07	08	09
-1	-1	4	6	-1	11	-1	9	13

10	11	12	13	14	15	16	17	18
-1	12	8	15	-1	16	0	-1	-1

Figura 6.16 – Alocação Encadeada. (a): situação em termos de blocos ocupados e livres. (b): lista encadeada.

Na Figura 6.16, o valor –1 agora indica que o bloco está livre e o valor 0 indica que este é o último bloco do arquivo.

A fragmentação dos espaços livres, apresentada no método anterior, não ocasiona nenhum problema na alocação encadeada, pois os blocos livres alocados para um arquivo não precisam necessariamente estar contíguos. O que ocorre neste método é a fragmentação de arquivos, que é a quebra do arquivo em diversos pedaços denominados extents.

A fragmentação resulta no aumento do tempo de acesso aos arquivos, pois o mecanismo de leitura/gravação dos discos deve se deslocar diversas vezes sobre sua superfície para acessar cada extent, gerando um excessivo tempo de busca (seek). Para otimizar o tempo das operações de E/S nesse tipo de sistema, é importante que o disco seja periodicamente desfragmentado. Apesar de ter propósitos diferentes, o procedimento de desfragmentação é idêntico ao já apresentado na alocação contígua.

A alocação encadeada só permite que se realize acesso sequencial aos blocos dos arquivos. Isto constitui uma das principais desvantagens dessa técnica, já que não é possível o acesso direto aos blocos. Por exemplo, se fosse necessário acessar o bloco de número 13, então a seguinte sequência de acessos seria realizada até que fosse conseguido chegar ao bloco 13: 3, 4, 6, 11, 12, 8 e 9. Além disso, essa técnica desperdiça espaço nos blocos com o armazenamento de ponteiros.

6.4.2.3 Alocação Indexada

A alocação indexada soluciona uma das principais limitações da alocação encadeada que é a impossibilidade do acesso direto aos blocos dos arquivos. O principio dessa técnica é manter os ponteiros dos endereços de todos os blocos do arquivo em uma única estrutura denominada bloco de índice. A Figura 6.17 ilustra esta técnica.

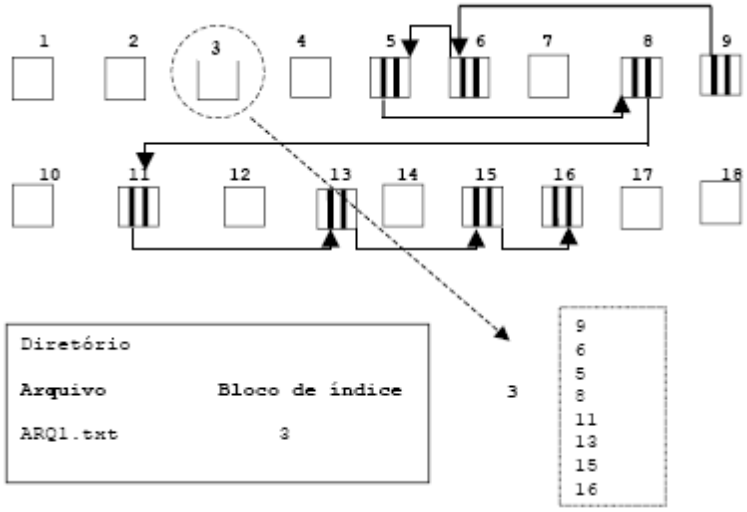


Figura 6.17 – Exemplo usando alocação indexada.

No caso de arquivos grandes, pode-se fazer um encadeamento de blocos de índices, ou seja, o ultimo valor de um bloco de índice aponta para o próximo bloco de índice. Portanto, é possível existir mais de um bloco para armazenar a informação de ponteiros dos endereços dos blocos do arquivo.

Capítulo 7
Gerenciamento de Memória

Memória é um importante recurso que deve ser gerenciado cuidadosamente. É senso comum perceber que, embora a capacidade de memória disponível nos sistemas de computação cada vez aumente mais, os desenvolvedores de software demandam por mais memória para que seus programas possam ser armazenados e executados.

Como o próprio nome sugere, gerenciamento de memória é a tarefa desempenhada pela parte do SO que gerencia a memória. É função desta parte do SO conhecer quais regiões da memória estão em uso e quais não estão sendo usadas, alocar memória para processos quando eles necessitam dela e desalocá-la quando os processos terminarem de ser executados, e gerenciar o swapping entre a memória principal e o disco, quando a memória principal não for grande o suficiente para comportar todos os processos.

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

• como a memória é vista, isto é, como pode ser utilizada pelos processos existentes neste sistema;

• como os processos são tratados pelo SO quanto às suas necessidades de uso de memória.

Este capítulo tratará de alguns esquemas de gerenciamento de memória. Porém, antes será abordada a organização hierárquica de memória.

7.1 Organização Hierárquica de Memória

Em um sistema de computação, o armazenamento de dados ocorre em diversos níveis. Em outras palavras, o armazenamento é realizado em diferentes tipos de dispositivos devido à quatro fatores básicos:

1.) tempo de acesso;
2.) velocidade de operação;
3.) custo por unidade de armazenamento;
4.) capacidade de armazenamento.

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

Em virtude das dificuldades tecnológicas associadas à construção de circuitos de memória e seu custo, o armazenamento de dados assumiu historicamente a organização mostrada na Figura 7.1.

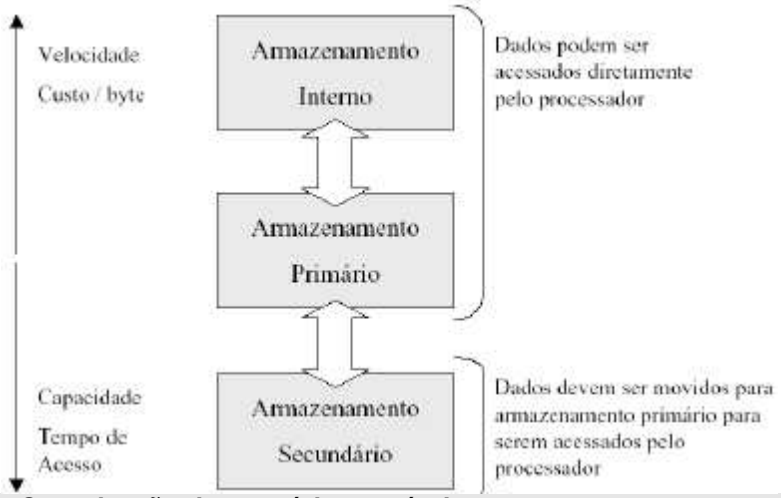


Figura 7.1 – Organização da memória em níveis.

Na Figura 7.1, **Armazenamento Interno** são posições de memória disponíveis internamente ao processador para permitir ou agilizar sua operação. É constituído dos registradores do processador e de seu cache interno. O **Armazenamento Primário** são as posições de memória externa diretamente acessíveis pelo processador.

Tipicamente, são CIs de memória SRAM, DRAM, EPROM, PROM, entre outras. Por seu lado, o **Armazenamento Secundário** são as posições de memória externa que não podem ser acessadas diretamente pelo processador, devendo ser movidas para o Armazenamento Primário antes da sua utilização. Tipicamente são os dispositivos de armazenamento de massa tal como o disco rígido.

Perceba que o Armazenamento Interno possui as maiores velocidades de acesso, ou seja os menores tempos de acesso, representando os melhores dispositivos em termos de desempenho, embora sejam os mais caros. Por outro lado, os dispositivos de Armazenamento Secundário são os de maior capacidade e os de melhor relação custo/Byte, mas consideravelmente mais lentos. O Armazenamento Primário representa um caso intermediário, onde a velocidade e o tempo de acesso são adequadas a operação direta com o processador, mas cujo custo ainda assim é alto.

Com a evolução dos computadores, a atual organização conta com outros elementos adicionados para otimizar o desempenho do sistema e, ainda assim, reduzir seu custo. A Figura 7.2 mostra a organização atual típica de armazenamento.

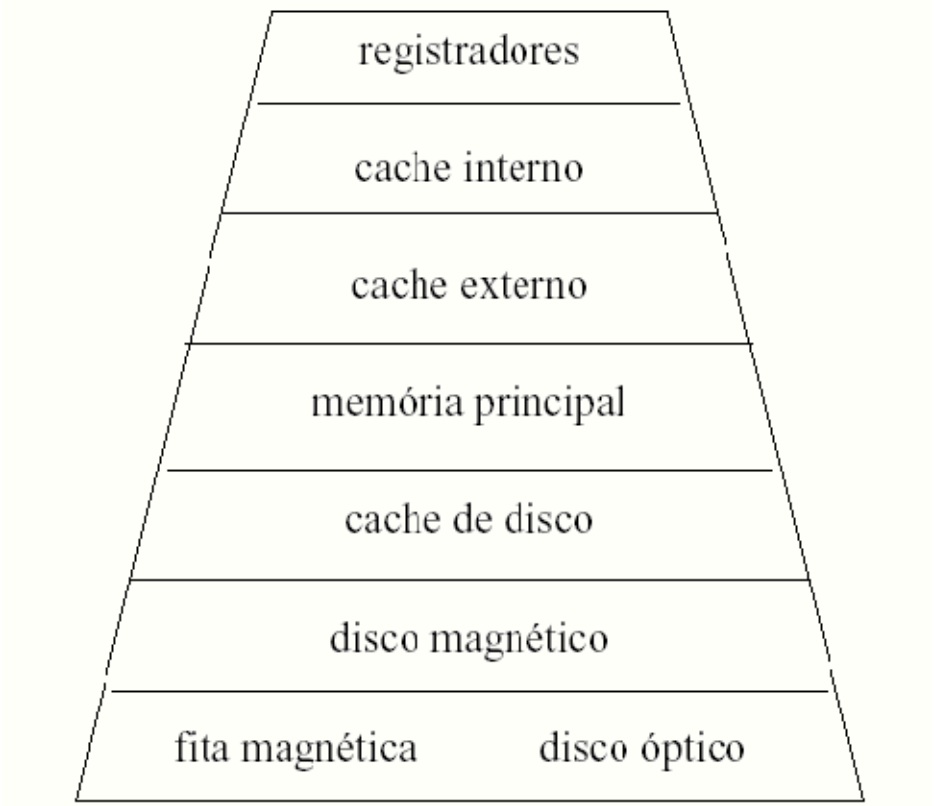


Figura 7.2 – Organização atual típica de armazenamento.

Os registradores, implementados em número limitado devido ao seu custo, são geralmente usados para manter dentro do processador dados frequentemente utilizados. Os caches interno e externo devido a sua maior velocidade, são usados para manter uma porção do programa que pode ser executada mais rapidamente do que na memória principal, aumentando o desempenho do sistema. A memória primária armazena os programas e dados em execução no sistema. Os dispositivos de Armazenamento Secundário são usados para preservação dos dados de forma perene. O cache de disco é utilizado para acelerar a operação das unidades de disco, podendo esta técnica ser utilizada para outros tipos de periféricos.

7.2 Tipos de Gerenciamento de Memória

De uma maneira geral, sistemas de gerenciamento de memória podem ser divididos em duas classes: aqueles que movem processos (programas) do disco para a memória principal e vice-versa, e aqueles que não realizam isto. Nesta seção, alguns tipos e conceitos relativos ao gerenciamento de memória serão abordados.

Os esquemas de alocações particionadas estática e dinâmica serão comentados primeiramente.

7.2.1 Alocações Particionadas Estática e Dinâmica

Nos sistemas multiprogramados, a memória primária foi dividida em blocos chamados de partições. Inicialmente, as partições, embora de tamanho fixo, não tinham necessariamente o mesmo tamanho, possibilitando diferentes configurações para sua utilização. Este esquema era conhecido como alocação particionada estática e tinha como grandes problemas:

- o fato dos programas, normalmente, não preencherem totalmente as partições onde eram carregados, desperdiçando espaço;

- se um programa fosse maior do que qualquer partição livre, ele ficaria aguardando uma que o acomodasse, mesmo se existisse duas ou mais partições adjacentes que, somadas, totalizassem o tamanho do programa. Este tipo de problema, onde pedaços de memória ficam impedidos de serem usados por outros programas, é chamado de fragmentação.

A Figura 7.3 mostra o problema da fragmentação associado à alocação particionada estática.

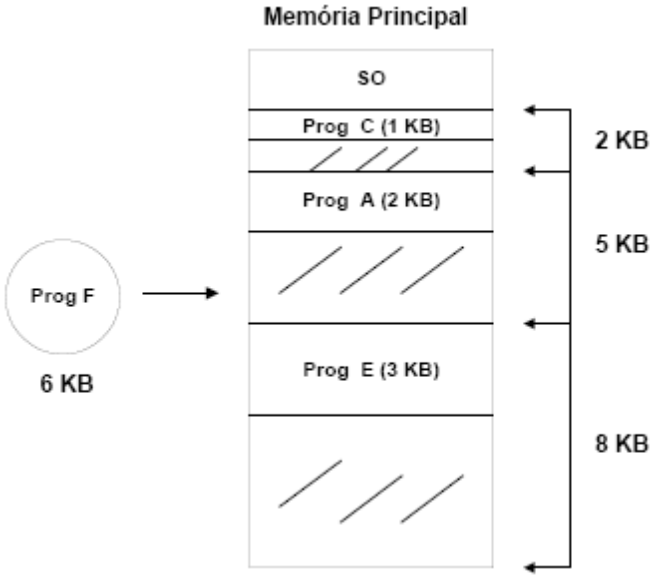


Figura 7.3 – Problema da fragmentação associado à alocação particionada estática.

Dado o problema da fragmentação na alocação particionada estática, foi necessário um outro tipo de alocação como solução e, consequentemente, o aumento do compartilhamento da memória. Na alocação particionada dinâmica, foi eliminado o conceito de partições de tamanho fixo. Nesse esquema, cada programa utilizaria o espaço que necessitasse, passando esse pedaço a ser sua partição. A Figura 7.4 mostra uma situação supondo alocação particionada dinâmica.

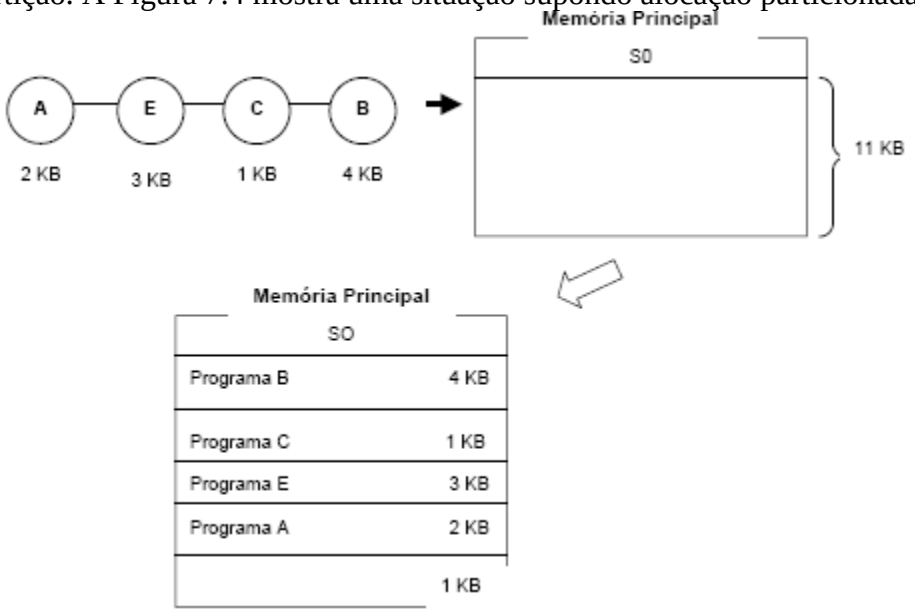


Figura 7.4 – Exemplo com alocação particionada dinâmica.

A princípio, o problema da fragmentação pareceu estar resolvido, porém, neste caso, a fragmentação começará a ocorrer, realmente, quando os programas forem terminando e deixando espaços cada vez menores na memória, não permitindo o ingresso de novos

programas. Para exemplificar este fato, seja a Figura 7.5. Neste caso, com o término de B e E e mesmo existindo 8 KB livres de memória, o programa D, de 6 KB, não poderá ser carregado.

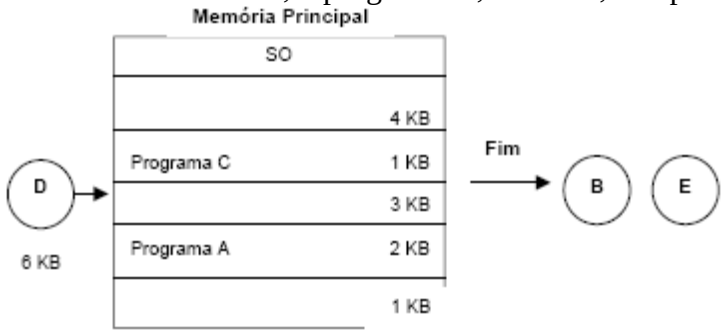


Figura 7.5 – Problema da fragmentação, a princípio, persiste na alocação particionada dinâmica.

Depois de já ter sido detectada a fragmentação da memória, existem duas soluções para o problema:

a) caso o programa C termine, o sistema pode reunir apenas os espaços adjacentes, produzindo espaço de tamanho 8 KB;

b) caso o programa C continue executando, o sistema pode realocar todas as partições ocupadas, eliminando todos os espaços entre elas, e criando uma única área livre contígua.

A Figura 7.6 mostra estas duas soluções para o problema da fragmentação.

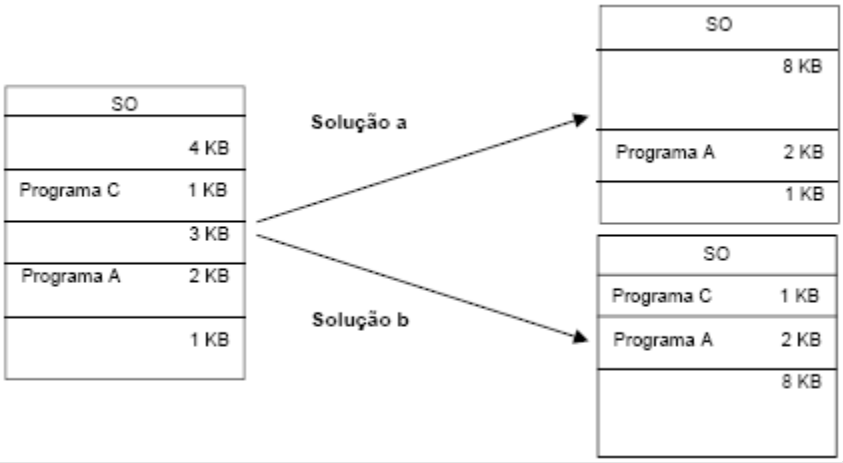


Figura 7.6 – Soluções para o problema da fragmentação.

A complexidade do algoritmo de desfragmentação e o consumo de recursos do sistema, como processador e área em disco, podem tornar este processo inviável.

É importante perceber que nestes dois tipos de gerenciamento de memória apresentados, o espaço de endereçamento é igual ao tamanho da memória primária existente no sistema.

7.2.2 Swapping

Em um sistema de Processamento em Lotes, a organização de memória em partições fixas é simples e eficiente. Desde que jobs suficientes possam ser mantidos na memória de modo que a CPU fique ocupada todo o tempo, não existe razão para usar outra organização mais complexa. Em sistemas de tempo compartilhado, a situação é diferente: normalmente existem mais usuários do que memória para manter todos os seus processos (programas), de modo que é necessário manter os processos em excesso no disco. Para executar tais processos, é necessário que eles sejam trazidos para a memória principal. O movimento de processos da memória principal para o disco e vice-versa é denominado de swapping.

Nas alocações particionadas estática e dinâmica, um programa permanecia na memória principal até o final da sua execução, inclusive nos momentos em que esperava um evento, como uma operação de leitura ou gravação em periféricos. Em outras palavras, o programa somente sairia da memória principal quando tivesse terminado sua execução.

Swapping pode ser usado em sistemas multiprogramados com partições de tamanho variável. Desta forma, de acordo com algum critério, um programa pode ser movido da memória principal para o disco (swap out) e, este mesmo programa, pode voltar do disco para a memória principal (swap in), como se nada tivesse acontecido. A Figura 7.7 ilustra esta situação.

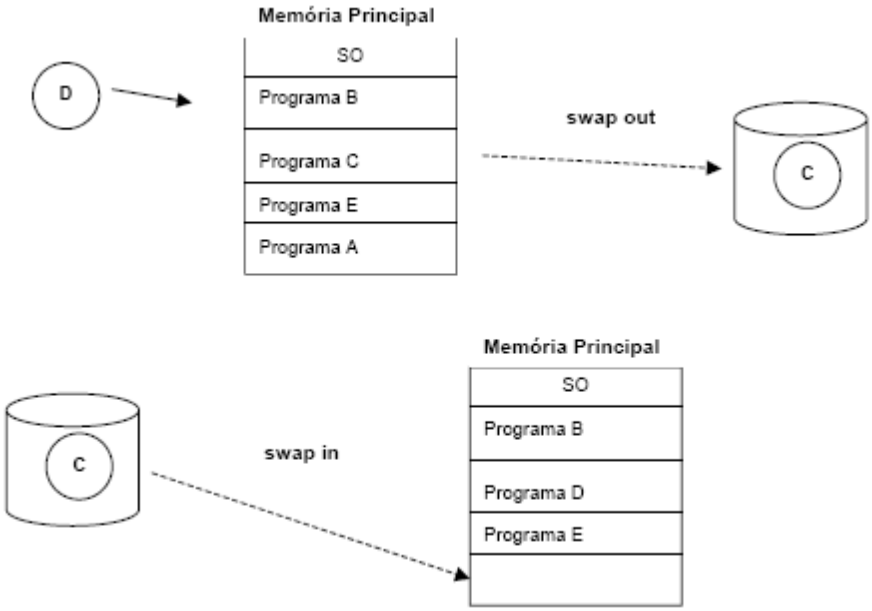


Figura 7.7 – Técnica de swapping.

Para um maior aprofundamento no assunto, é relevante mencionar que o swapping é realizado por rotinas especiais do SO chamadas de relocadores ou swappers. A existência de relocadores em um sistema depende do tipo de gerenciamento de memória oferecido pelo SO. Uma explicação simplificada do trabalho realizado pelos relocadores é dada a seguir.

Seguindo instruções do SO, que detêm o gerenciamento de memória e dos processos, um relocador pode ser comandado para retirar o conteúdo de uma área de memória, armazenando-a em disco.

O que geralmente ocorre é que o relocador realiza uma cópia desta área de memória em um arquivo especial denominado arquivo de troca ou swap file. Ao copiar a área de memória para o disco, tal área é assinalada como livre, tornando disponível para outros processos.
Também, é efetuado um registro do que foi copiado para a memória possibilitando a recuperação deste conteúdo. A Figura 7.8 mostra o relocador realizando swapping.

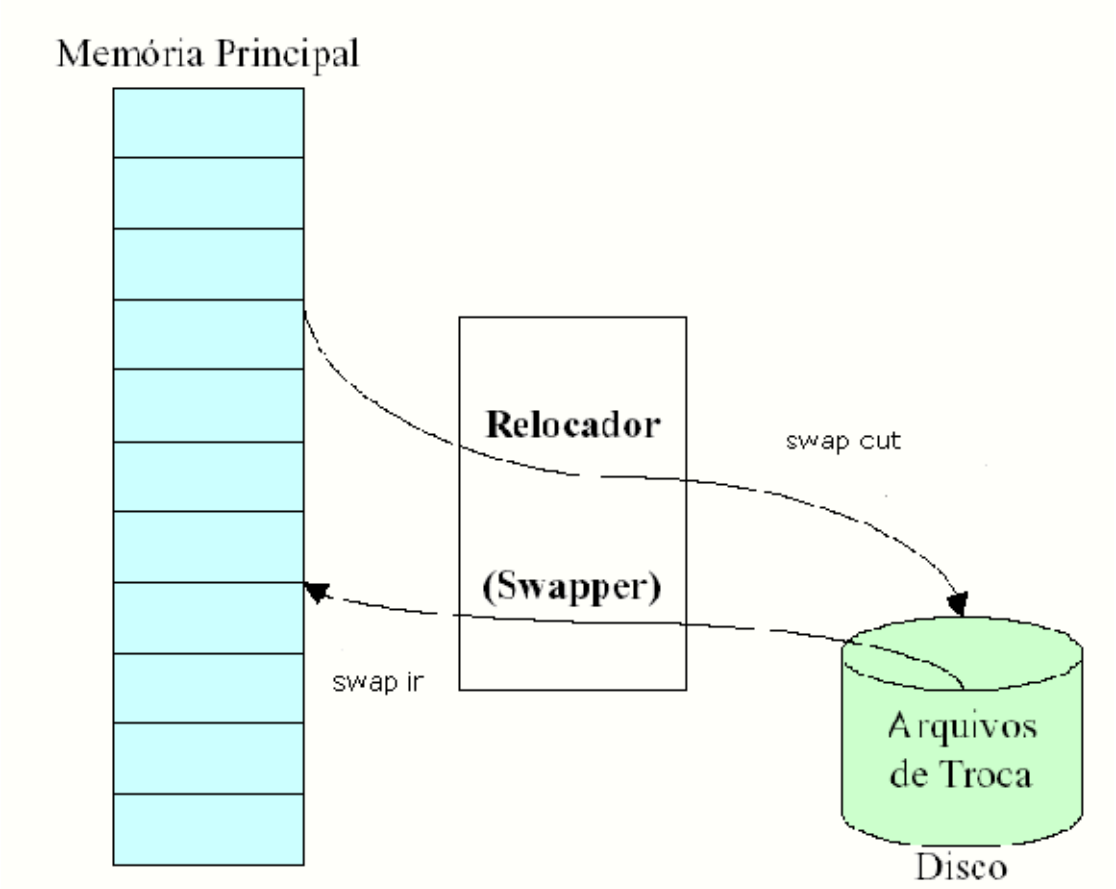


Figura 7.8 – Relocador em ação.

Em um sistema multiprogramado podem existir vários processos em estado pronto e vários bloqueados, e apenas um processo efetivamente em execução em um determinado instante. Tal processo em execução pode solicitar áreas adicionais de memória. Estes novos pedidos de alocação de memória podem ser atendidos de duas formas: áreas efetivamente livres são cedidas ao processo ou aciona-se o relocador para liberação de áreas de memória pertencentes aos demais processos por meio da remoção de seu conteúdo para os arquivos de troca.

7.2.3 Memória Virtual

O conceito de relocação de memória possibilitou o desenvolvimento de um mecanismo mais sofisticado de utilização de memória denominado memória virtual.

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.
Segundo Deitel, "o termo memória virtual é normalmente associado com a habilidade de um sistema endereçar muito mais memória do que a fisicamente disponível". Este conceito surgiu em 1960 no computador Atlas, construído pela Universidade de Manchester (Inglaterra), embora sua utilização mais ampla tenha acontecido recentemente. A Figura 7.9 mostra a representação da memória virtual e real.

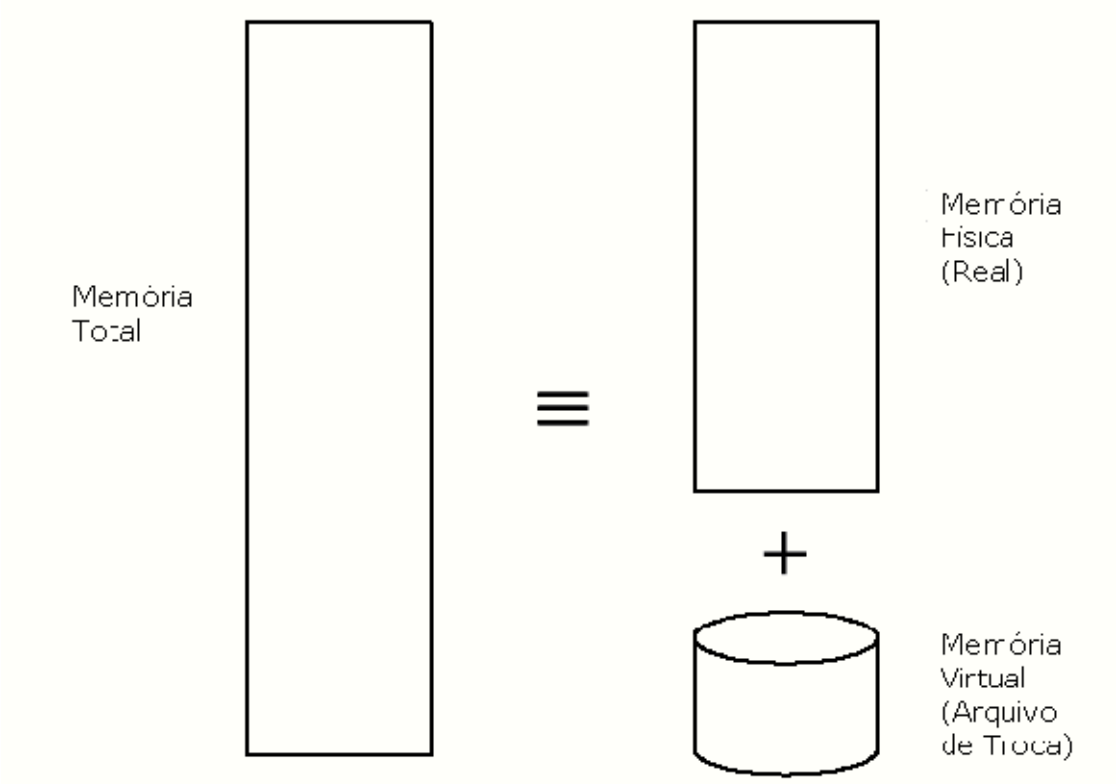


Figura 7.9 – Representação da memória virtual e real.

Conforme pode ser visto na Figura 7.9, a memória virtual de um sistema é, de fato, o(s) arquivo(s) de troca ou swap file(s) gravado(s) no disco rígido. Portanto a memória total de um sistema, que possui memória virtual, é a soma de sua memória física, de tamanho fixo, com a memória virtual. O tamanho da memória virtual, chamado de arquivo de paginação no Windows XP, é definido, basicamente, pelo menor valor dentre os seguintes:

• capacidade de endereçamento do processador;

• capacidade de administração de endereços do SO;

• capacidade de armazenamento dos dispositivos de armazenamento secundário (unidades de disco).

Um programa no ambiente de memória virtual não faz referência a endereços físicos de memória (endereço reais), mas apenas a endereço virtuais. No momento da execução de uma instrução, o endereço virtual é traduzido para um endereço físico, pois o processador acessa apenas posições da memória principal.

O mecanismo de tradução do endereço virtual para endereço físico é denominado mapeamento. Este mecanismo, nos sistemas atuais, é feito pelo hardware, com o auxílio do SO, e ele traduz um endereço localizado no espaço de endereçamento virtual para um endereço físico de memória, pois o programa executado em seu contexto precisa estar no espaço de endereçamento real para poder ser referenciado ou executado. Portanto um programa não precisa estar necessariamente contíguo na memória real para ser executado.

O mecanismo de tradução se encarrega de manter tabelas de mapeamento exclusivas para cada processo, relacionando os endereços virtuais do processo às suas posições na memória física.

A Figura 7.10 mostra o mapeamento para o caso do processo A.

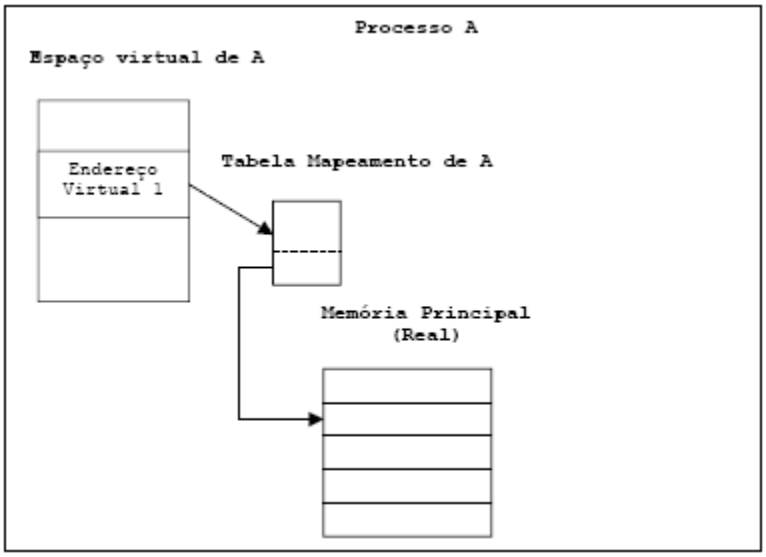


Figura 7.10 – Mecanismo de Mapeamento.

A memória virtual pode ser implementada basicamente por meio dos mecanismos de paginação e segmentação. Atualmente, paginação é o mecanismo mais popular de implementação de memória virtual. A segmentação é uma alternativa menos utilizada, embora mais adequada do ponto de vista de programação. Alguns sistemas usam ambas as técnicas. Paginação e segmentação serão abordadas nas próximas seções.

7.2.3.1 Paginação

A paginação é a técnica de gerenciamento de memória que faz uso da memória virtual, ou seja, o espaço de endereçamento é maior do que o tamanho da memória fisicamente presente no sistema. Nesta técnica, o espaço de endereçamento total do sistema, denominado de espaço de endereçamento virtual, é dividido em pequenos blocos de igual tamanho chamados páginas virtuais ou simplesmente páginas.

Cada página possui um número que a identifica. Da mesma forma, a memória física é dividida em blocos iguais, do mesmo tamanho das páginas virtuais, denominados de molduras de páginas. Cada moldura de página também é identificada por um número, e corresponde uma certa região da memória física. Tanto as páginas virtuais como as molduras de páginas começam a ser identificadas a partir do número zero.

Os endereços gerados por um programa em execução são chamados de endereços virtuais e formam o já mencionado espaço de endereçamento virtual. Seja o caso de um programa executar a seguinte instrução de uma certa linguagem de programação: MOV REG, 2060. Esta instrução, desta linguagem, diz que o conteúdo do endereço de memória 2060 deve ser copiado para o registrador REG.

O endereço 2060, gerado pelo programa é, portanto, um endereço virtual.

Em computadores sem o mecanismo de memória virtual, o endereço virtual é colocado diretamente no barramento da memória e isto causará o acesso (leitura ou escrita) da palavra de memória física com o mesmo endereço. No caso de existir memória virtual, o endereço virtual vai para a MMU (Memory Management Unit – Unidade de Gerenciamento de Memória), um CI ou uma coleção de CIs (hardware) que faz o mapeamento dos endereços virtuais em endereços físicos. A Figura 7.11 ilustra esta situação.

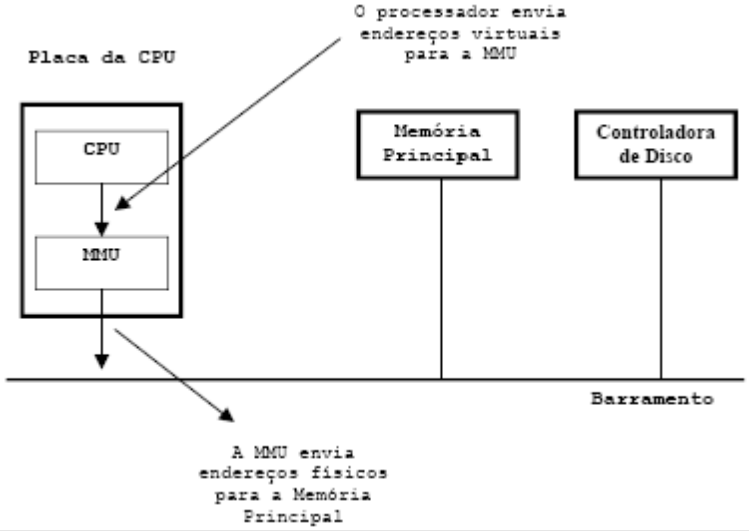


Figura 7.11 – MMU e a tradução de endereços virtuais em físicos.

Para que o esquema de divisão proposto pela paginação seja eficaz, o SO deve realizar o mapeamento de forma a identificar quais páginas virtuais estão presentes na memória física (em molduras de páginas) e quais estão localizadas na memória virtual do sistema (arquivo de troca). Tal mapeamento é feito por meio de tabelas de páginas que determinam a relação entre as páginas virtuais, do espaço de endereçamento virtual, e as molduras de páginas do espaço de endereçamento físico (real). A Figura 7.12 mostra os espaços de endereçamento virtual e real na paginação.

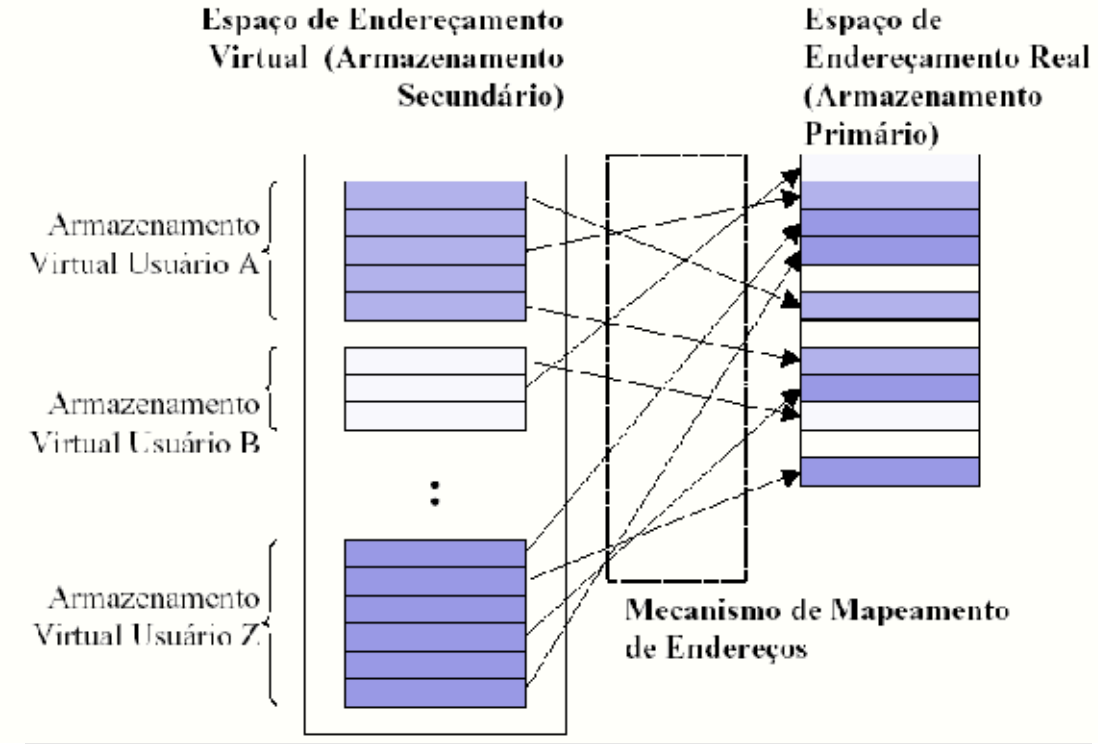


Figura 7.12 – Espaços de endereçamento virtual e real na paginação.

Quando programas são executados, as páginas virtuais são transferidas da memória secundária para a memória principal e colocadas nas molduras de páginas. Com o decorrer da execução dos programas, o SO relaciona quais páginas virtuais estão sendo alocadas para cada um destes programas, sem se preocupar com o posicionamento contíguo de partes de um mesmo

programa. No instante efetivo da execução, a MMU converte os endereços virtuais em endereços físicos, consultando as tabelas de páginas, conforme mostra a Figura 7.13.

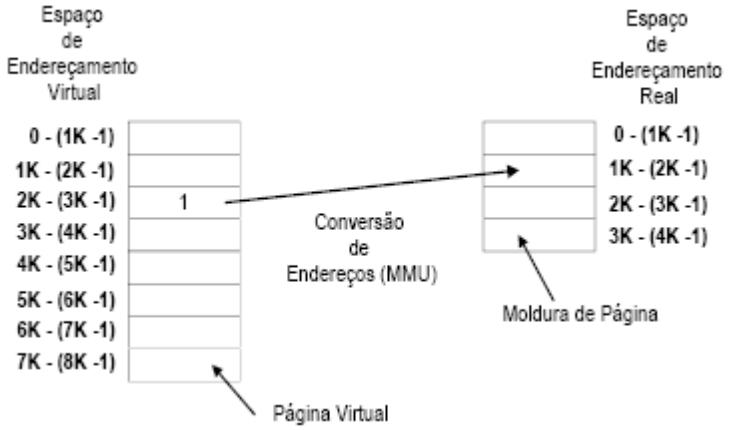


Figura 7.13 – Relacionamento entre endereços virtuais e físicos.

Na Figura 7.13, o tamanho das páginas é de 1 KB. Perceba que o espaço de endereçamento virtual é o dobro do espaço real. Então, supondo a instrução mencionada anteriormente, o endereço virtual gerado pelo programa é 2060. Este endereço está na página virtual 2, conforme mostra a Figura 7.13. A MMU, ao consultar a tabela de páginas, verifica que a página virtual 2 está mapeada na moldura de página 1. Deste modo, a MMU consegue fazer a conversão do endereço virtual para o endereço físico.

O endereço virtual é formado pelo número da página virtual e um deslocamento dentro da página. Para obter o deslocamento, basta subtrair o endereço virtual da instrução do endereço virtual inicial da página virtual em questão. No exemplo da instrução MOV REG, 2060, tem-se:

```
Endereço virtual inicial da página virtual 2 = 2048 (2K);
Endereço virtual da instrução = 2060;
Deslocamento = 2060 – 2048 = 12.
```

O endereço físico é calculado somando-se o endereço físico inicial da moldura de página, que está mapeada com a página virtual endereçada, com o deslocamento contido no endereço virtual. Então, no exemplo:

```
Endereço físico inicial da moldura de página 1 = 1024 (1K);
Deslocamento = 12;
Endereço físico = 1024 + 12 = 1036.
```

Além da informação sobre a localização da página virtual, a entrada na tabela de páginas possui a informação se a página que contém o endereço referenciado está ou não na memória principal, por meio de um bit de validade. Se o bit tem o valor 0, isto indica que a página virtual não está na memória principal, enquanto se for igual a 1, a página virtual está localizada na memória principal. A Figura 7.14 mostra o conceito de bit de validade.

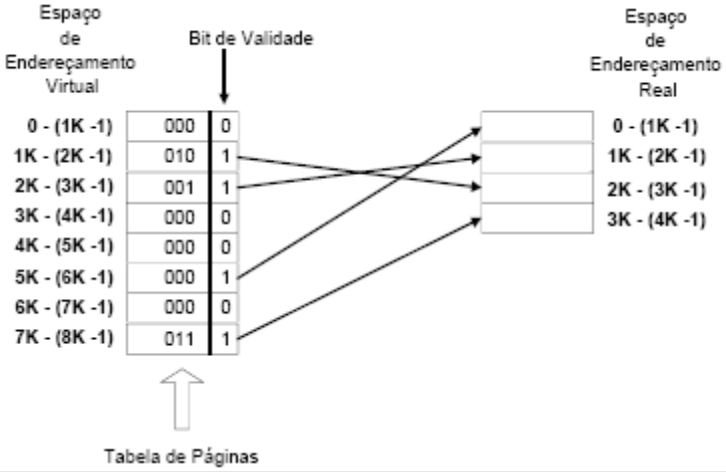


Figura 7.14 Tabela de páginas e bit de validade.

Na Figura 7.14, os valores das entradas na tabela de páginas estão expressos em binário. O bit destacado dos demais é o bit de validade.

Se a MMU identificar que uma página virtual não está mapeada na memória física (principal), como a página virtual 6 da Figura 7.14, ela realiza uma operação de falha de página (page fault).

Quando ocorre isto, uma rotina do SO busca na memória virtual (arquivo de troca) a página necessária, trazendo-a para a memória física (swap in). Esta operação é particularmente complexa quando não existe espaço livre na memória física. Por exemplo, na Figura 7.14, dado que um programa endereçou a página virtual 6, como esta página não está mapeada na memória física e a memória está cheia, qual das páginas atualmente na memória deve ser removida para o disco (swap out)? Um grande problema da técnica de paginação é justamente este: a escolha da página atualmente na memória física que deve ser removida. Existem diversos algoritmos adotados pelos SOs para a realocação de páginas, entre eles pode-se citar:

• First-In, First-Out (FIFO): a página que primeiro foi utilizada será a primeira a ser selecionada;

• Least-Recently-Used (LRU): a página selecionada é a que está há mais tempo sem ser referenciada;

• Not-Recently-Used (NRU): a página selecionada é a que não foi recentemente utilizada.

7.2.3.2 Segmentação

Segmentação é a técnica de gerenciamento de memória onde os programas são divididos logicamente em sub-rotinas e estruturas de dados e colocados em blocos de informações na memória. Os blocos têm tamanhos variáveis e são chamados segmentos, cada um com seu próprio espaço de endereçamento. A Figura 7.15 ilustra a técnica de segmentação.

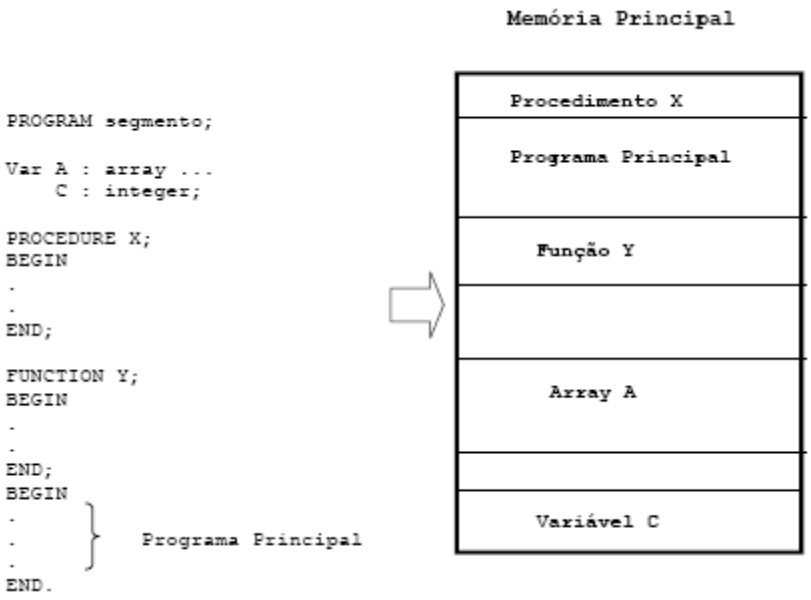


Figura 7.15 – Técnica de segmentação.

A grande diferença entre a paginação e a segmentação é que, enquanto a primeira divide o programa em partes de tamanho fixo, sem qualquer ligação com a estrutura do programa, a segmentação permite uma relação entre a lógica do programa e sua divisão na memória.
O mecanismo de mapeamento é muito semelhante ao da paginação. Além do endereço do segmento na memória física, cada entrada na tabela de segmentos possui informações sobre o tamanho do segmento e se ele está ou não na memória.
Se as aplicações não estiverem divididas em módulos, grandes pedaços de código estarão na memória desnecessariamente, não permitindo que outros usuários, também utilizem a memória.

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

Índice Remissivo