VII - Gerenciamento de Memória

Memória é o segundo recurso de maior importância dentro da arquitetura de um sistema computador. A disponibilidade e o gerenciamento deste recurso é vital para o desempenho do sistema. Assim, fatores como a quantidade de memória disponível, tempo de acesso e custo são capitais na sua especificação e aquisição.

Evidente que o desejado é poder-se ter memória de maior capacidade, que seja bastante rápida e cujo custo seja no mínimo compatível com o dos demais componentes do computador. Como era de esperar, existe uma relação de compromisso (*trade-off*) entre estes três fatores, na realidade o que ocorre é que:

- memórias mais rápidas apresentam maior custo por bit armazenado;
- memórias de maior capacidade apresentam menor custo por bit; e
- quanto maior a capacidade da memória mais lenta ela será.

Estes fatos fazem com que os projetistas busquem solucionar o problema, dentro das restrições inerentes a tecnologia atualmente disponível, adotando não uma mas sim um conjunto de memórias organizadas segundo uma hierarquia tal que possibilite adequar custo, capacidade de armazenamento e tempo de resposta.

VII.1 Níveis hierárquicos de Armazenamento

Seis níveis, conforme pode ser visto na figura VII.1 abaixo, formam uma estrutura de memória que opera integrada e ordenada segundo a capacidade de armazenamento, a velocidade de acesso que proporcionam ao sistema e ao custo por bit.

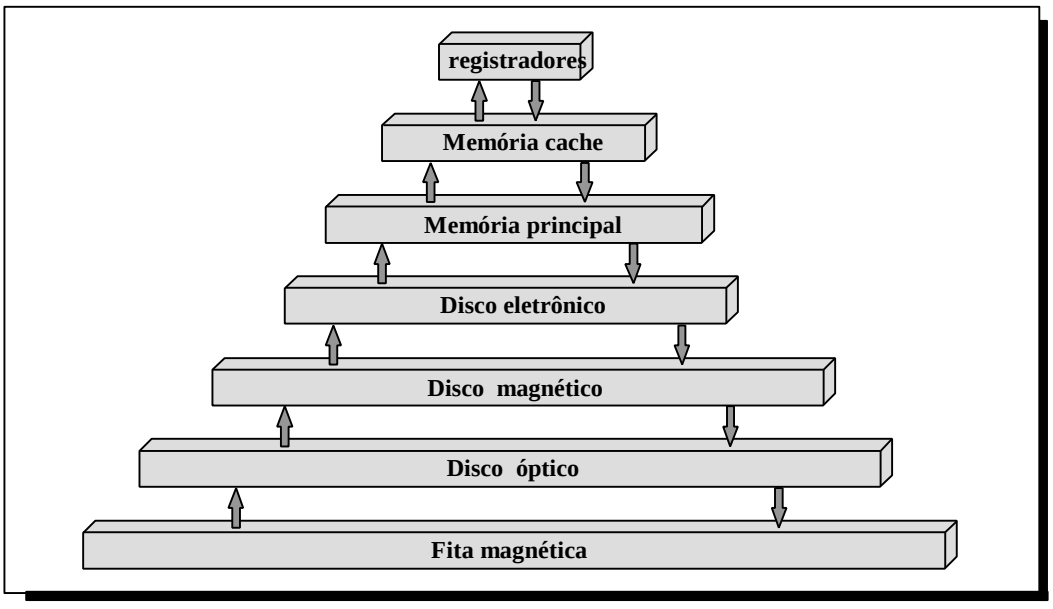


Figura VII.1 - Níveis Hierárquicos de Armazenamento

Quanto mais elevado o nível na pirâmide, mais rápida, de menor capacidade e de custo mais elevado é o dispositivo de memória. O intercâmbio de dados entre os diferentes dispositi-

vos que compõem o sistema hierárquico de memória deve ser suportado pelo sistema operacional da forma mais transparente e segura possível. Com esta estrutura é possível alcançar os seguintes marcos:

1. decréscimo no custo global do sistema por bit armazenado;
2. aumento da capacidade global de armazenamento;
3. melhoria do desempenho (tempo de acesso);
4. redução do número de acessos à memória principal.

a) Registradores

Localizados no interior da UCP, servem de memória (buffer) para as operações internas a UCP e são de acesso exclusivo ao processo corrente. Resumem-se a alguns poucos registradores, cujo número e tamanho depende da arquitetura interna de hardware da UCP. São de tamanho fixo e normalmente idêntico ao da palavra de memória.

b) Cache

É uma memória RAM de tamanho reduzido e com velocidade de acesso bem mais elevada do que a da memória principal. É utilizada com o objetivo de acelerar a execução dos processos possibilitando acesso mais rápido àquelas palavras mais freqüentemente referenciadas e, com isto, explorar o *princípio da localidade de referência* dos dados e dos programas.

A cache é geralmente localizada dentro da UCP, compartilhando da arquitetura interna e dos barramentos desta. Funcionalmente, a cache se situa entre os registradores e a memória principal e, assim, toda operação de leitura (ou escrita) na memória tem sua execução, primeiramente, tentada na memória cache (fig. VII.2).

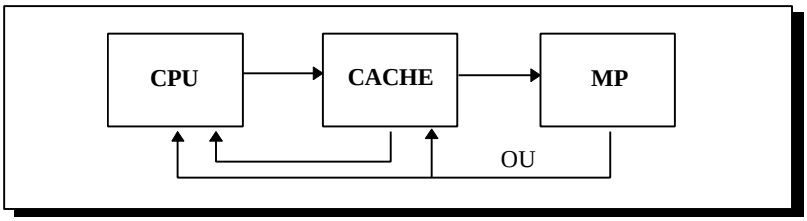


Figura VII.2 - Esquema de Acesso com CACHE Presente

Se o dado estiver na *cache*, ocorrerá um **hit**, e é esta então que fornece o dado à UCP; caso contrário ocorrerá um **miss**, e a memória principal fornecerá o dado à UCP e à memória *cache*. Os dados entre cache e UCP são transferidos com base na palavra (word) e entre a memória principal e a cache a transferência é feita em blocos, como mostra a figura VII.3.

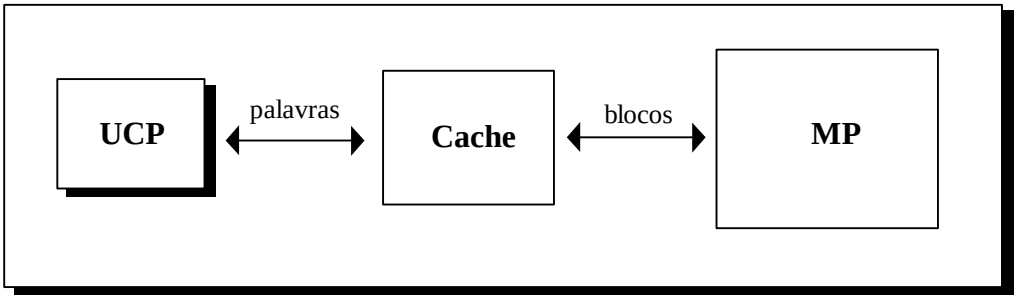


Figura VII.3 - Unidade de transferência para Cache e MP

O uso da cache otimiza o acesso à memória principal, geralmente muito mais lenta, por esta servir à UCP em grande parte dos acessos e evitar assim, a ida até a MP. O tempo de acesso nestes casos passa a ser dado pela seguinte expressão:

$$t_{ma} = t_c + (1\text{-hit}) * t_{mp}$$

onde,

t_{ma}	tempo médio de acesso
t_c	tempo de acesso à cache
t_{mp}	tempo de acesso à MP
hit	probabilidade do dado estar na cache

Assim, se por exemplo, um computador possui memória principal com tempo de acesso da ordem de 1.5μs e memória cache com tempo de acesso da ordem de 0.1μs. O tempo necessário para ler 1000 bytes com taxas de hit respectivamente de 0%, ou seja, 100% de acesso à memória principal, e de 85%, será:

a) $t_{ma1} = 1000 * (0.1 + 1.5) = 1.6\text{ms}$ e

b) $t_{ma2} = 1000 * (0.1 + 0.15 * 1.5) = 325\mu\text{s}$

Outras medidas de desempenho são eficiência de acesso (T_1/T_{ma}) e de custo (C_s). Em outras palavras:

$$\frac{T_1}{T_{ma}} = \frac{1}{hit + (1 - hit)\frac{T_2}{T_1}}$$

onde,

T_{ma}	tempo médio de acesso considerando-se 2 níveis de memória
T_1	tempo médio de acesso da memória de mais alto nível
T_2	tempo médio de acesso da memória de mais baixo nível
hit	taxa de localização do dado na memória de nível mais elevado

Observe que quanto mais T_1/T_{ma} se aproxima de 1, maior será o grau de eficiência do sistema.

e

$$C_s = \frac{C_1 S_1 + C_2 S_2}{S_1 + S_2}$$

onde,

C_s	custo médio por bit combinados dois níveis adjacentes de memória
C_1	custo médio por bit da memória de nível mais elevado
C_2	custo médio por bit da memória de nível mais baixo
S_1	tamanho da memória de nível mais elevado (M_1)
S_2	tamanho da memória de nível mais baixo (M_2)

A eficiência agora está direção da proximidade de C_s com C_2, uma vez que C_1 tende a ser muito maior que C_2. Isto geralmente requer que S_1 seja muito menor que S_2.

O projeto de uma memória cache envolve os seguintes aspectos:

1. Tamanho da cache
2. Unidade de transferência com a MP
3. Função de alocação
4. Algoritmo de substituição
5. estratégia para operações de escrita

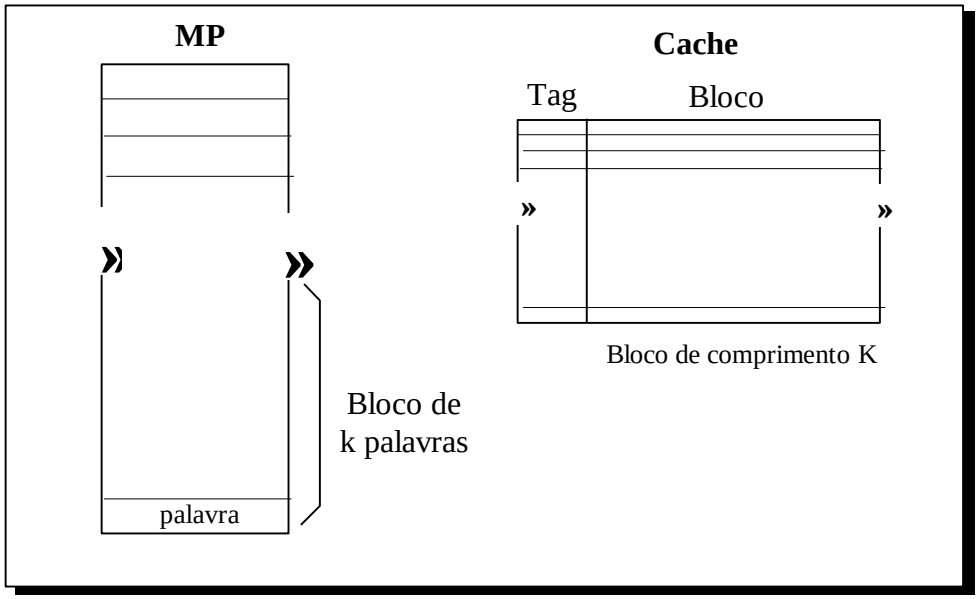


Figura VII.4 - Organização física da MP e da Cache

Mesmo um tamanho pequeno de cache introduz uma melhora sensível no desempenho do sistema. O tamanho do bloco, por outro lado, está ligado à taxa de **hit**. Ao ser elevado tende a aumentar o hit porém, a partir de um certo tamanho passa a provocar um reflexo contrário, em virtude da intensificação da troca de blocos por substituição.

A Função de Alocação se refere ao processo de escolha dos possíveis locais de armazenamento para um bloco a ser trazido da MP e o Algoritmo de Substituição se refere, em caso de necessidade, à estratégia de escolha, dentre os locais selecionados pela função de alocação, daquele bloco de dados a ser retirado da cache para abrir espaço para o novo que entra.

As operações de escrita num sistema dotado de memória cache, precisam de uma atenção especial, com vistas a manter a coerência do sistema (*cache coherency*). Num extremo temos a estratégia chamada *write-through*, onde tanto a cache como a MP são atualizadas a cada operação de escrita corresponde e no outro a *write-back*, onde a MP somente é atualizada ao final de toda uma seqüência de operações ou quando o dado alterado precisar deixar a cache em virtude do algoritmo de substituição.

Na técnica ***write-through***, caso a memória cache retorne um *hit* ou um *miss* a ação do sistema será a seguinte:

- no caso de um hit (ou seja, o dado se encontra na cache)

1. escreve o novo valor do dado na cache, e
2. escreve o novo valor do dado também na MP (fig VII.5).

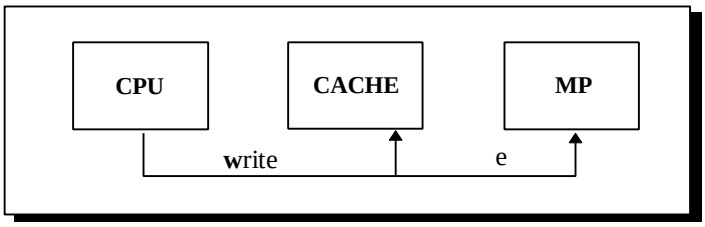


Figura VII.5 - Esquema de write-through com hit

- no caso de um miss (o dado não se encontra na cache)

1. lê o dado na MP e o grava na cache,
2. escreve o novo valor do dado na cache, e
3. escreve o novo valor do dado também na MP (fig VII.6)

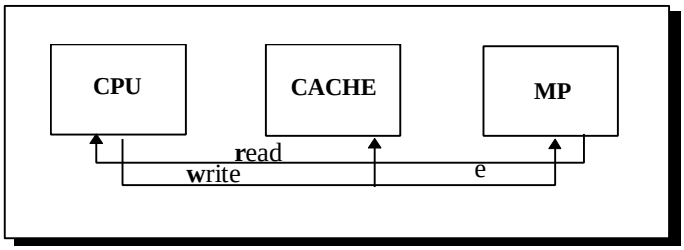
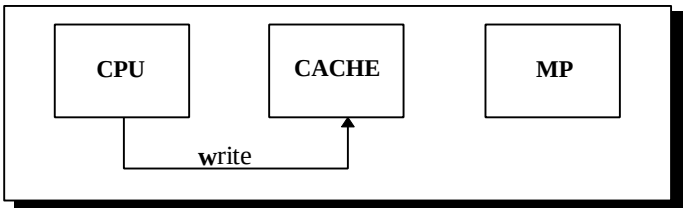


Figura VII.2b - Esquema de write-through com miss

Na técnica ***write-back***, caso a memória cache retorne um hit a ação do sistema será a seguinte:

1. escreve o novo valor do dado apenas na cache tantas vezes quanto puder, e



2. lê da cache e escreve o valor na MP somente quando precisar remover o dado da cache por falta de espaço ou no final do processamento do programa (fig VII.7).

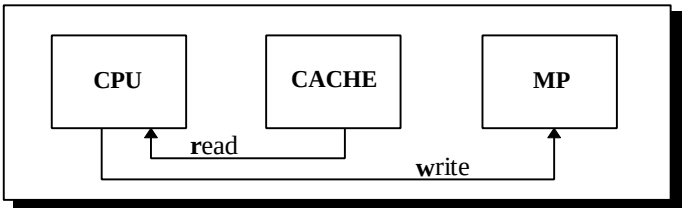


Figura VII.7 - Esquema de write-back com hit, (a) escreve apenas na CACHE tantas vezes quantas necessárias, (b) escreve (copia) o dado na MP uma única vez ao final do processamento ou quando necessário para abrir espaço na Cache

No caso de um miss (o dado não se encontra na cache)

- escreve o novo valor do dado apenas na MP (fig. VII.8).

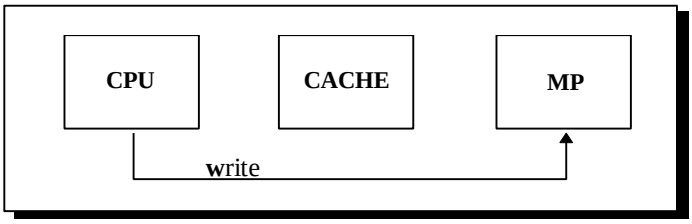


Figura VII.8 - Esquema de write-back com miss

c) Memória Principal

É constituída por um determinado número de células, cada uma podendo armazenar uma quantidade fixa de informação e sendo individualizada por um endereço único. A célula representa a unidade de acesso à memória, isto é, a menor quantidade de bits que pode ser individualmente endereçada e acessada. O tamanho (quantidade de bits) de cada célula depende da arquitetura física do computador. A célula de memória muitas vezes é referenciada como byte por geralmente coincidir com o tamanho deste. Na figura VII.9 abaixo são apresentados os tamanhos de célula para alguns computadores conhecidos.

ex.

Máquina	bits/célula
Borroughs B1700	1
IBM PC	8
PDP8	12
IBM 1130	16
XDS 940	24
XDS Sigma 9	32
CDC 3600	48
CDC Cyber	60

Figura VII.8 - Tamanhos de Células para alguns computadores conhecidos

d) Memória Virtual

É uma estratégia de utilização combinada da memória principal com a memória secundária (disco magnético) tal que, de forma transparente ao usuário, dá a este e aos processos a impressão de estarem fazendo uso de uma memória principal muito maior do que a real.

Requer a existência de componentes específicos no sistema operacional, como será visto mais a frente, para prover e gerenciar o uso da memória virtual de maneira transparente aos demais componentes do próprio sistema operacional, dos aplicativos, dos programas elaborados pelo usuário e do próprio usuário. Pela importância deste tópico ele será visto em maiores detalhes mais a frente ainda neste capítulo.

e) Memória Secundária

Provê flexibilidade (*memória virtual*), segurança (*backup*) e maior capacidade de armazenamento ao sistema. Inclui os seguintes dispositivos:

- discos magnéticos - fixos (HDs) e removíveis (disquetes);
- fitas magnéticas - cartucho, cassete e rolo;
- discos ópticos - apenas para leitura (CDROMs) e para leitura e gravação (WORMs e magneto-ópticos).

VII.2 Alocação de Espaço na Memória Principal

Consiste nos procedimentos do sistema operacional quanto a proteção e a alocação de espaços em memória principal para execução dos diferentes processos que compartilham no tempo o uso da UCP. Com a evolução do hardware e dos sistemas, vários métodos de alocação foram surgindo e sendo adotados pelos sistemas operacionais das respectivas épocas. Dentre eles temos:

- alocação contígua simples,
- partições estáticas,
- partições estáticas relocáveis, e
- partições dinâmicas

a) Alocação Contígua Simples

Adotada nos sistemas operacionais mais antigos do tipo *monoprogramáveis*, consiste na alocação contínua dos endereços de memória, onde os mais baixos são utilizados pelo SO e o restantes são disponibilizados para o usuário (fig. VII.9).

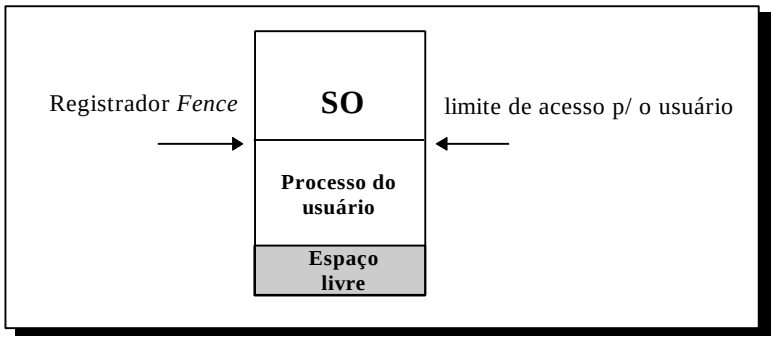


Figura VII.9 - Memória com Alocação Contígua Simples

Neste esquema não há proteção, ou seja, o usuário tem controle total sobre todo o espaço de memória, podendo acessar qualquer posição, inclusive alterar e destruir o sistema operacional. Para proteção alguns sistemas incluem um registrador, chamado *registrador fence*, para delimitar as áreas do SO e do usuário.

Embora de implementação simples e código reduzido, a *alocação contígua simples* não permite uma utilização eficiente do processador e da memória, pois apenas um usuário pode

usar os recursos existentes. Os problemas de proteção e de realocação são resolvidos da seguinte maneira:

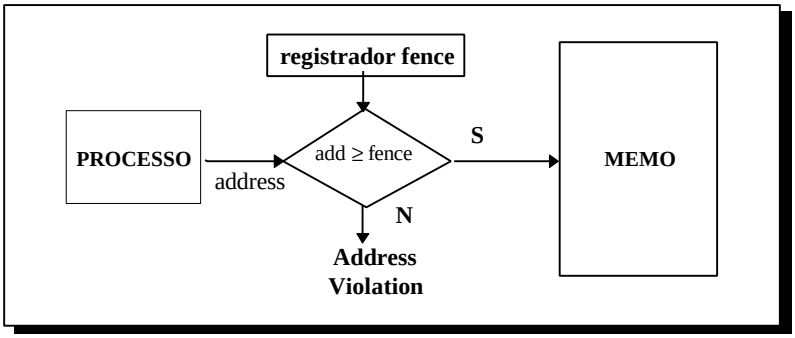


Figura VII.10 - Esquema de Proteção

a. proteção - a cada tentativa de acesso do usuário à memória, o SO realiza um teste com o registrador fence, permitindo ou não o acesso.

b. realocação - da forma como está configurada a MP (fig. VII.9), se o SO precisar crescer será necessário deslocar o registrador fence e, por via de conseqüência, recompilar todos os programas, uma vez que os endereços são absolutos. Uma primeira solução para este problema surgiu com a adoção do endereçamento decrescente para o processo, isto é a partir do final da MP (fig. VII.11).

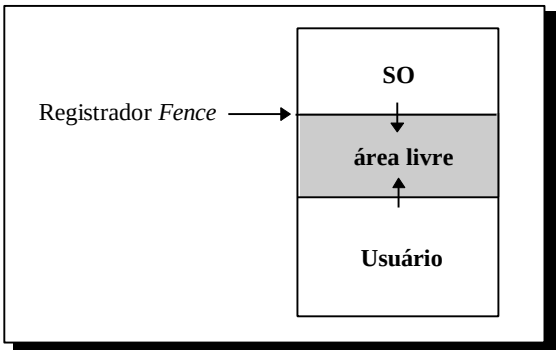


Figura VII.11 - Alocação com Endereçamento Decrescente

Desta forma, tanto o SO como o processo do usuário podem crescer até o limite físico da memória, sem que um gere problemas, *"overhead"*, para o outro.

Os programas do usuário nesta época sofriam fortemente da limitação física do espaço de memória que, por força da tecnologia existente, já era reduzido por natureza. A solução encontrada na época foi a de dividir o programa em módulos independentes, que pudessem ser executados separadamente reutilizando, de forma sucessiva, uma mesma área de memória. Esta técnica é chamada de ***overlay***, e nela cabe ao programador a identificação dos trechos do programa que podem ser colocados em diferentes módulos.

Na figura VII.12 abaixo é exemplificado um caso de *overlay* onde um programa de 940K é subdividido em três partes (módulos), um principal de 510K, que permanece em memória durante todo o período de execução e chama, de forma alternada, as outras duas rotinas, uma de 200K e outra de 230K.

Observe que o espaço de memória disponível para o usuário é de aproximadamente 750K e que o maior comprimento do programa em execução será de 740K, quando as rotinas principal e B estiverem juntas na memória.

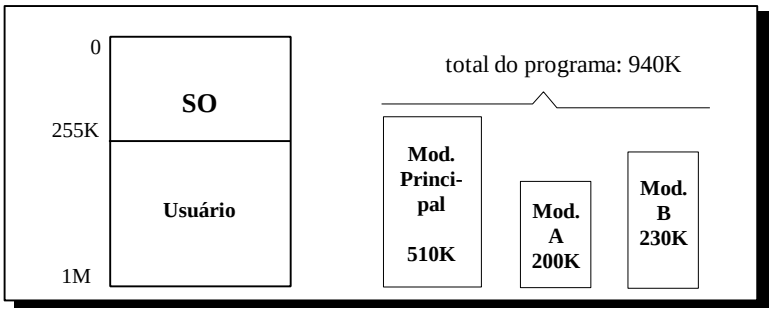


Figura VII.12 - Técnica de Overlay

b) Múltiplas Partições

Para a multiprogramação ser possível, é necessário que vários programas estejam na memória ao mesmo tempo para que possam se revezar rapidamente no uso e controle da UCP, daí a necessidade de uma nova forma de organização da memória.

b.1) Alocação Particionada Estática

Nos primeiros sistemas operacionais multiprogramados, a MP foi dividida em pedaços de tamanho fixo, chamados ***partições***. O tamanho das partições era variável e estabelecido na fase de ***boot*** do sistema, sendo definido pelo operador em função do tamanho dos programas a serem executados.

A proteção das partições era garantida através dos registradores de limite superior e inferior (*fence*). A princípio, os programas eram previamente alocados e só podiam ser executados em uma das partições, mesmo que outras estivessem disponíveis. Essa limitação se devia aos compiladores que eram capazes apenas de gerar endereçamento absoluto.

Os processos para execução eram selecionados e organizados pelo operador formando uma fila para cada partição (fig. VII.13).

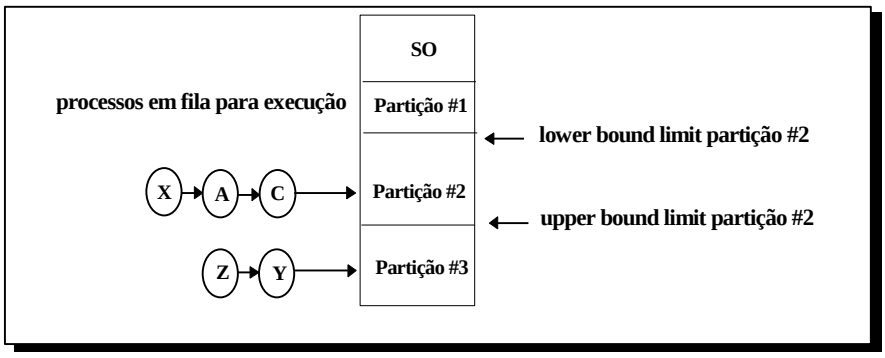


Figura VII.13 - Alocação Particionada Estática

b.2) Alocação Particionada Estática Relocável

Com a evolução dos compiladores, linkers e loaders, foi possível a geração de códigos relocáveis a nível de execução, isto é, os endereços dentro de um programa eram relativos ao início do mesmo e, assim, se tornando independente da posição absoluta de carga do mesmo na memória.

No esquema de endereçamento relocável, o endereço absoluto de cada instrução do programa passa a ser calculado através do seu endereço relativo (gerado pelo compilador) e o endereço de carga na memória (fornecido pelo carregador - loader, e armazenado num registrador especial chamado *registrador base*).

Na partição relocável uma única fila de processos é formada, e o carregador (loader) escolhe a partição a ser usada dentre as disponíveis no momento da carga de cada processo (fig. VII.14).

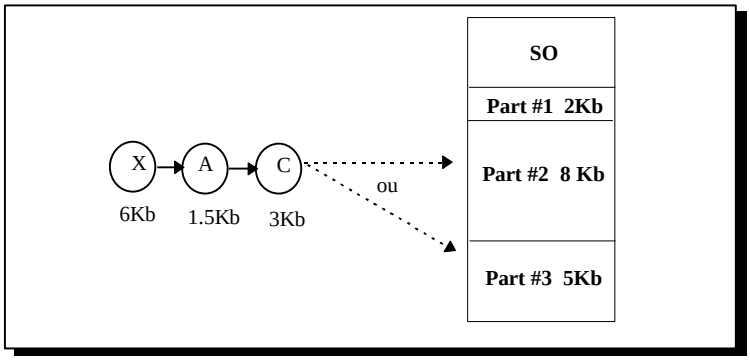


Figura VII.14 - Alocação Particionada Relocável

Este tipo de partição tem o potencial de gerar **fragmentação interna de memória**, i.e. memória alocada e não utilizada pelo programa. Este tipo de fragmentação ocorre sempre que o programa não utilizar toda a memória disponível na partição para a qual foi alocado.

A fila única também gerava problemas, uma vez que processos com partição disponível poderiam ter que ficar aguardando na fila o atendimento de outros processos que estivessem na frente e que necessitassem alguma partição de tamanho não disponível.

b.3) Alocação Particionada Dinâmica

A solução anterior, particionada estática relocável, deixou claro a necessidade de se implementar uma nova forma de organização da MP que reduzisse o problema de fragmentação e aumentasse o grau de compartilhamento da mesma.

Eliminou-se então, o conceito de partições de tamanho fixo definidas no momento do boot. O tamanho da partição passou a ser dinâmico e definido automaticamente pelo próprio SO, de acordo com cada programa a ser executado.

A princípio o problema da fragmentação parecia resolvido porém, observou-se que a mesma passaria agora a ocorrer na medida que os programas fossem terminando e deixando fragmentos, chamados de fragmentos externos, cada vez menores e dispersos pela memória. Fenômeno este vulgarmente conhecido como *colcha-de-retalhos*. Desta forma, como exempli-

ficado na figura VII.15 abaixo, os espaços iam se tornando cada vez menores e, por não serem contíguos, não podiam ser reagrupados para execução de novos processos.

O tratamento deste problema veio com a introdução da técnica de ***compactação***. Porém com a restrição de um custo computacional mais elevado.

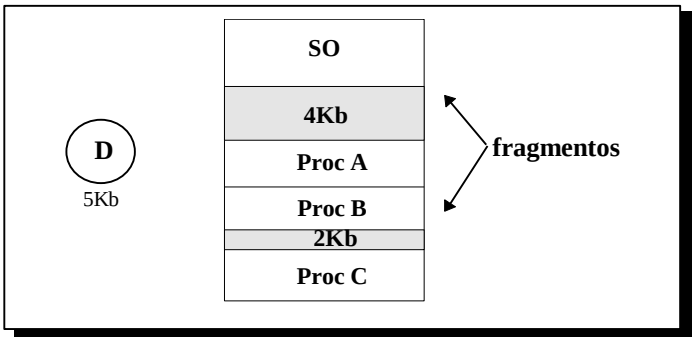


Figura VII.15 - Alocação Particionada Dinâmica

1^{0}) Compactação Simples

Consistia em reunir espaços adjacentes não utilizados produzindo um único espaço de tamanho maior (fig. VII.16).

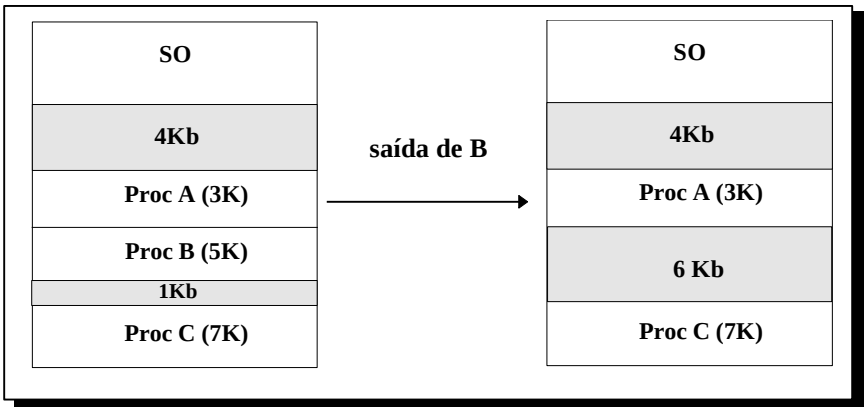


Figura VII.16 - Compactação Simples

2^{0}) Compactação com Relocação

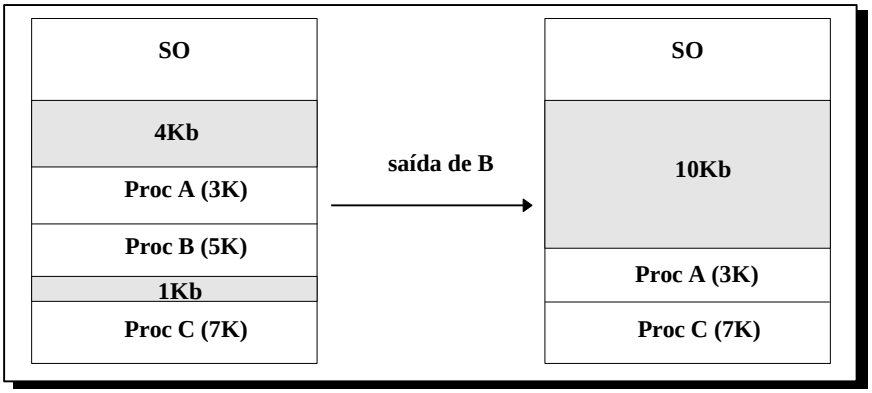


Figura VII.17 - Compactação com Relocação

Envolve a relocação de todas as partições ocupadas, eliminando assim todos os espaços vazios entre elas e criando uma única área livre e contígua de memória. Esta solução reduz o problema da fragmentação, porém a complexidade do seu algoritmo e o custo computacional para sua operacionalização podem torná-lo inviável (fig. VII.17).

b.3.1) Estratégias para Escolha da Partição

A escolha da partição pelo SO merece cuidado especial uma vez que reflete no nível de fragmentação gerado e no desempenho global do sistema. Três estratégias básicas são comumente adotadas: *best-fit*; *worst-fit* e *first-fit*. O SO mantém uma lista de áreas livres, com o endereço e o tamanho de cada uma.

- Best-fit - é alocada a partição que deixa a menor área livre (fragmentação). A tendência é da memória ficar cada vez mais com pequenas áreas não contíguas, tornando mais crítico o problema da fragmentação.

- Worst-fit - é alocada a partição que deixa a maior área livre. Tem a tendência de deixar espaços livres maiores e não provocar fragmentações tão pequenas como no caso do best-fit.

- First-fit - é alocada a partição de endereço mais baixo. Apresenta grandes chances de permitir a existência de partições maiores nos endereços mais elevados.

VII.3 Swapping

A técnica de *swapping* veio para tentar minorar o problema da insuficiência de memória para a execução dos processos em ambiente multiprogramado. Consiste na transferência automática de todo um processo da MP para o disco (*swap out*) e vice-versa, do disco para a MP (*swap in*), conforme exemplificado na figura VII.18 abaixo.

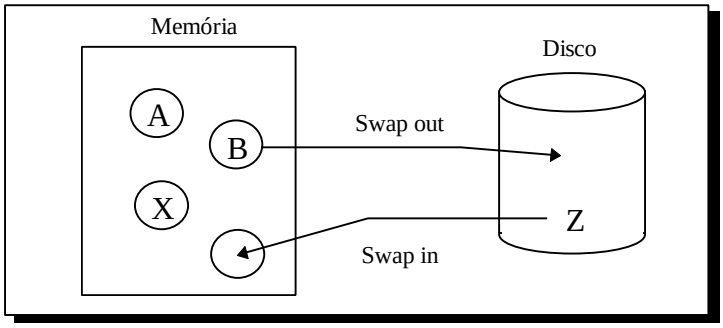


Figura VII.18 - Esquema de Swapping

Um dos problemas gerados pelo swapping é a relocação dos programas. No caso de um programa que saia e volte muitas vezes para a memória, é necessário que a relocação do mesmo seja realizada automaticamente pelo **loader** a cada operação de carregamento.

A melhor solução para esta relocação é a realizada através do registrador base ou registrador de relocação. Toda vez que um programa é carregado na memória, o seu registrador base é carregado com o endereço inicial da região de memória onde o programa será carregado.

O conceito de swapping permitiu um maior compartilhamento da MP e, consequentemente, um maior throughput. Seu maior problema é o elevado custo das operações de E/S (swap in/out).

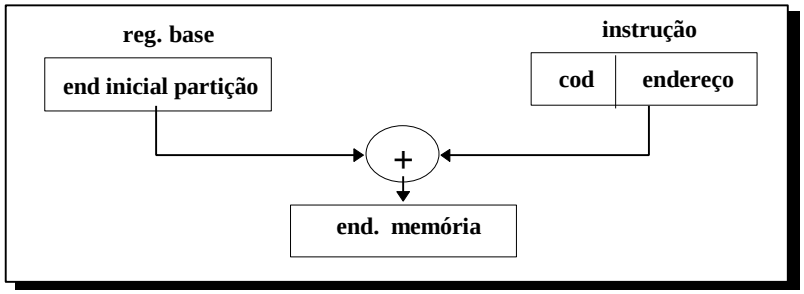


Figura VII.19 - Cálculo do Endereço Absoluto de Memória

VII.4 Memória Virtual

É uma técnica sofisticada e eficiente de gerenciamento de memória, incorporada aos SOs mais recentes. Ela combina, de forma transparente aos usuários e ao restante do próprio sistema operacional, as memórias principal e secundária (disco) afim de proporcionar a ilusão de existir uma memória principal muito maior do que a real.

O conceito de memória virtual se baseia na separação entre o endereçamento usado pelo programa e os endereços físicos da memória principal. Agora, o espaço que um programa pode endereçar é chamado de *espaço virtual*, que pode ser bem maior do que o espaço físico existente na memória principal.

O Endereçamento Virtual possui as seguintes características:

- é o conjunto de endereços que podem ser utilizados pelos processos em execução.
- o espaço virtual pode ser muito maior do que o espaço físico.
- o compilador se encarrega de implementar automaticamente o endereçamento virtual.
- um programa, em um ambiente de memória virtual, não faz referência a endereços físicos, apenas a endereços virtuais. O software de gerenciamento da memória virtual se encarrega de mapear cada endereço virtual para um endereço físico, de forma automática e transparente ao processo e ao usuário.
- o sistema operacional faz uso da memória secundária como extensão da memória principal, de forma transparente.
- quando um programa é executado, apenas uma parte do código executável fica na memória principal, o restante permanece em memória secundária e só é carregado quando referenciado para uso.
- cada processo tem a seu dispor o mesmo espaço de endereçamento virtual, como se toda a memória virtual lhe pertencesse. Cabe ao software de gerenciamento controlar e impedir qualquer violação de acesso entre dois ou mais processos concorrentes.

a) Mapeamento

Nos sistemas atuais, para agilizar o processo, o mapeamento é executado por um hardware específico que trabalha junto com o sistema operacional. Para garantir segurança, o me-

canismo de mapeamento se encarrega de manter tabelas de mapeamento exclusivas para cada processo (*page tables*).

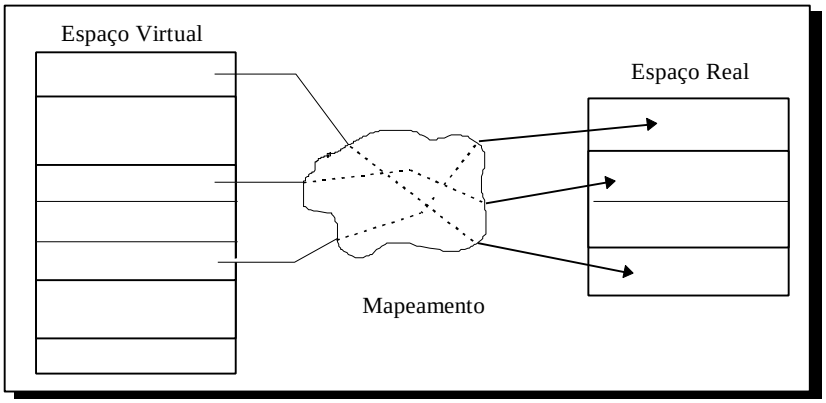


Figura VII.20 - Mapeamento do Endereço Virtual x Real

Caso a granularidade do mapeamento fosse o *byte*, o espaço físico requerido pelas tabelas de mapeamento seria tão grande quanto o espaço virtual, o que inviabilizaria o uso e a implementação do mecanismo de memória virtual. Em virtude disto, as tabelas *mapeiam blocos* de informações chamadas *páginas ou segmentos*. Observe que quanto maior o tamanho do bloco, menor a tabela de mapeamento, porém maior o tempo de transferência do mesmo de / ou para a MP (fig. VII.20).

obs: Com a introdução do mapeamento, os programas deixaram de ter, necessariamente, de ocupar um espaço contígüo na memória física.

b) Paginação

É a técnica virtual que faz uso de blocos de tamanho fixo tanto no espaço virtual (*páginas*) como no espaço real (*frames*). Nesta técnica, o endereço virtual é formado pelo número da página virtual e pelo deslocamento dentro desta mesma página. O número (endereço) da página virtual é único dentro da *page table*.

O endereço físico é calculado (*mapeado*) pela composição do endereço do *frame* fornecido pela *page table* com o deslocamento fornecido pela instrução virtual, conforme pode ser visto na figura VII.21 a seguir.

Além da informação sobre a localização física da página virtual (*frame*), a *page table* possui outras informações como o nível de proteção da página, o bit de validade (*validity bit*), que indica se a página está na memória física ou não, e o bit de modificação (*modified bit*), que indica se a página na memória física foi modificada ou não. Esta informação é importante pois se o sistema precisar remover a página da memória física por uma razão de necessidade de espaço, terá ou não que salvá-la em disco antes de alterar o bit de validade na *page table*.

Toda vez que uma página é referenciada e não está na memória física ocorre um *page fault*, que dispara um processo de *alocação de memória física*, que pode, por sua vez, requerer a retirada de uma página que esteja residindo na memória física antes de realizar a carga da página referenciada.

Observe que neste esquema de memória virtual, cada referência a memória se converte, na melhor das hipóteses, em dois acessos a memória principal, um para leitura da *page table* e outra para acesso ao dado propriamente dito. Na ocorrência de um *page fault*, além dos acessos acima referenciados, tem-se o acesso ao disco para carga dos dados e, antes disso, se necessário, o salvamento de algum dado em memória para abertura de espaço.

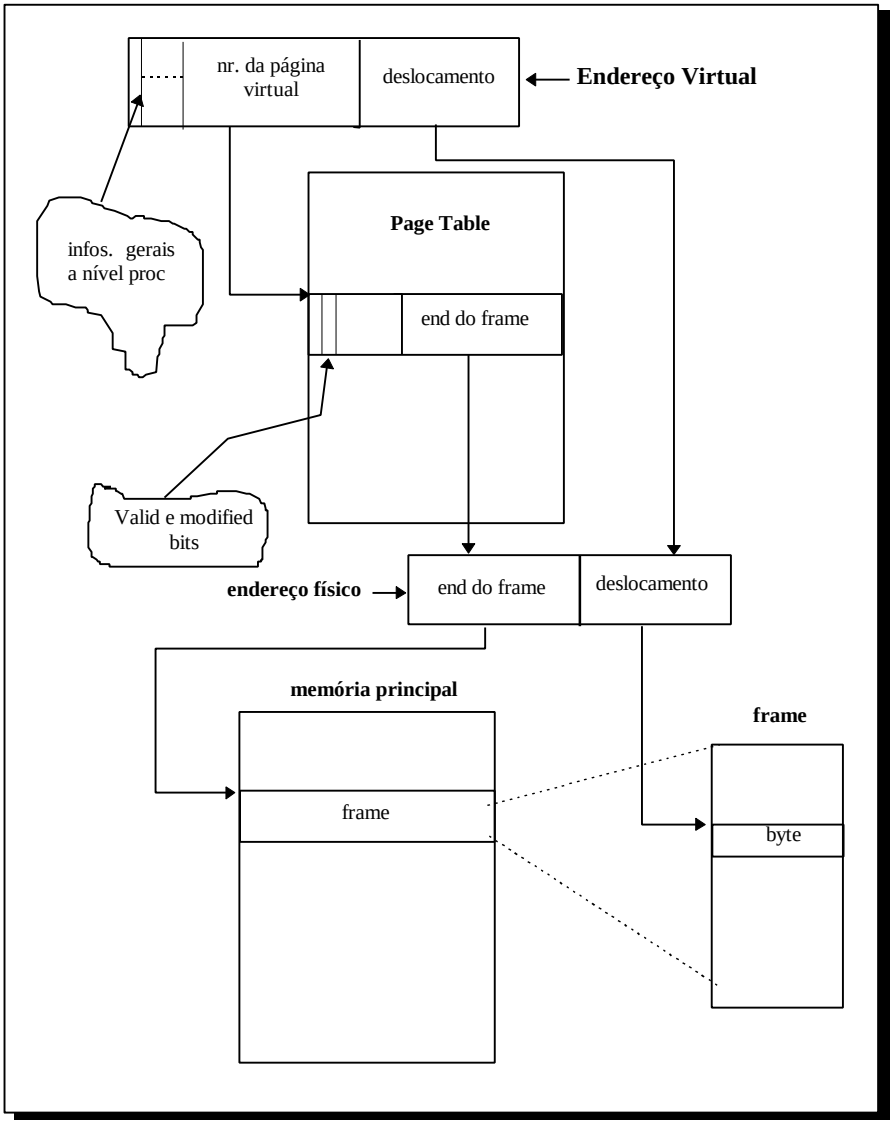


Figura VII.21 - Esquema de Mapeamento com Paginação

Quando a página não está na memória principal, a *page table* contém o endereço da mesma no disco magnético. Abaixo vê-se alguns conceitos relacionados com esta estratégia de gerenciamento:

- *Demand Paging* - é a transferência sob demanda para a memória principal, ou seja apenas quando a página é referenciada;
- *Antecipatory Paging* - tenta antecipar a demanda por uma página carregando-a na MP antes dela ser referenciada;
- *Working Set* - é o espaço de memória (número de páginas) que é permitido a um processo manter simultaneamente na MP;
- *Page Size* - é o tamanho adotado para as página, normalmente entre 512 e 4Kb (512 é o mais comum);

- *Fragmentação* - é o espaço de MP alocado a um processo e não utilizado pelo mesmo que, no caso de paginação, é interna a cada página e bem mais reduzida do que nos demais casos já visto até aqui;
- *Swapping* - também é válido para memória virtual (neste caso a page table do processo é transferida da MP para a MS e, portanto, precisa ser carregada para MP antes de ser iniciado o processo de mapeamento do endereço virtual;
- *Thrashing* - é definido como a excessiva transferência de dados entre MP e MS (paginação ou segmentação). As causas mais prováveis são:
    a) a nível do próprio processo - mau dimensionamento do working set, ou não observação do princípio da localidade pelo programador;
    b) a nível do sistema - pouca memória ou número excessivo de processos concorrentes.

c) Working Set

O conceito de *working set* surgiu da observação e análise da taxa de paginação dos vários processos em execução. Percebeu-se a ocorrência de uma elevada taxa de paginação durante a fase inicial de execução (carga) do processo e, geralmente, uma estabilização desta taxa no decorrer do restante do processamento. Este comportamento deve-se ao fato de que antes de iniciada a execução o processo reside como um todo na memória secundária (MS), sendo então necessário carregar para a memória as páginas a medida que estas vão sendo referenciadas pela primeira vez, mas depois de um certo tempo de processamento a demanda por novas páginas tende a reduzir (princípio da localidade) e a taxa de paginação se estabiliza.

O tamanho do *working set* é definido pelo gerente do sistema (system manager), e é de fundamental importância para garantir um fluxo adequado de execução dos processos (throughput). Um *working set* muito grande reduz o número de processos que podem compartilhar simultâneamente a memória, além de geralmente ser sub-utilizado pela maioria dos processos existentes. Alguns sistemas operacionais adotam a estratégia de estabelecimento de um working set dinâmico, observado um tamanho máximo e mínimo pré-definidos.

d) Relocação de Páginas

Um dos maiores problemas na implementação de um software de gerenciamento para paginação é o tratamento dos *page faults*, pois isto demanda a escolha de uma estratégia para selecionar as páginas a serem removidas da MP, afim de que seja aberto espaço para o carregamento de páginas residentes na MS.

Qualquer que seja a estratégia adotada de projeto, ela deve sempre considerar o bit de modificação (*modified bit*) para determinar se a página a ser retirada da MP sofreu ou não alguma alteração durante sua estada na memória. Caso positivo a página precisa ser salva na MS antes que a outra seja carregada, caso contrário basta carregar a outra por cima.

A melhor estratégia possível a adotar seria a de relocar a página que não viesse a ser referenciada em futuro próximo. Isto seria o ideal, porém é impraticável pois o sistema operacional (SO) não possui bola de cristal e, portanto, não tem como adivinhar o futuro. As estratégias mais conhecidas são:

- escolha aleatória (raramente adotada);
- FIFO (*first_in_first_out*, simples de ser implementada);

- LRU (*least_recently_used*, gera um *overhead* razoável);
- NUR (*not_used_recently*, usa um flag inicializado com 0);
- LFU (*least_frequently_used*, mantém um contador de referências p/ cada página).

- **Escolha Aleatória**

Não faz uso de qualquer critério para a seleção. Todas as páginas do working set têm a mesma chance de ser selecionada. Embora seja a estratégia mais simples de ser implementada e consumir poucos recursos de processamento, é raramente utilizada pela sua baixa eficiência.

- **FIFO**

Remove a página carregada menos recentemente, independente de quando a página foi referenciada pela última vez. Sua implementação é bastante simples, por exemplo, através de uma fila FIFO para cada working set.

- **LRU**

Remove a página utilizada menos recentemente, isto é, aquela que não é referenciada há mais tempo. Sua implementação é mais complexa que a dos algoritmos anteriores e, embora o algoritmo funcione razoavelmente bem, existe algumas situações em que ele falha grosseiramente, como no caso extremo de um loop que referencia em seqüência um número de páginas que exceda em um o tamanho máximo do working set.

Uma possível forma de implementação deste algoritmo é realizada através do uso de uma matriz quadrada conforme mostra a figura VII.22. A matriz tem dimensão igual ao número de páginas do working set. Cada linha e coluna de mesma ordem se referencia a uma mesma página. Ao ser carregada a página, a linha correspondente da matriz é zerada. Quando a página é referenciada, a linha correspondente da matriz é carregada com "1s" e a coluna correspondente com "0s". A qualquer instante, a página menos recentemente referenciada é aquela cuja linha apresenta o menor somatório.

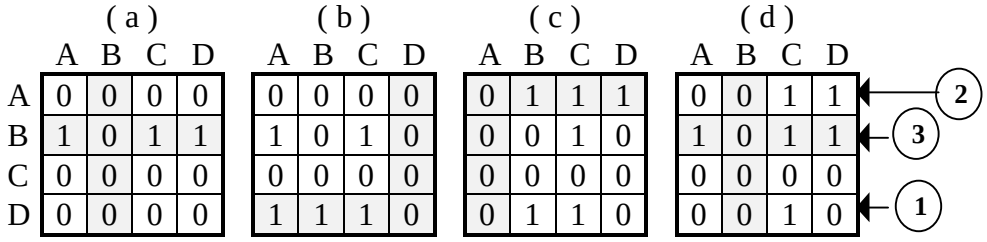


Figura VII.22 - Implementação do esquema LRU para a referência de páginas: B, D, A, B. Observe que a qualquer instante a linha de maior valor corresponde a página mais recentemente referenciada, e assim por diante

- **NRU**

Escolhe a página que não tenha sido recentemente referenciada. O algoritmo faz uso de um bit extra: flag "R", para registrar a utilização da página. O bit é setado para "1" quando a página é referenciada e, em períodos de tempo pré-determinados, são todos ressetados para "0". Toda vez que ocorrer um page fault e uma relocação de página for necessária, o algoritmo

analisa os bist "R" e "M" (modified bit) de todas as páginas do working set e as classifica em 04 grupos:

classe 1:	não referenciadas (R = 0) e não modificadas (M = 0),
classe 2:	não referenciadas (R = 0) e modificadas (M = 1),
classe 3:	referenciadas (R = 1) e não modificadas (M = 0), e
classe 4:	referenciadas (R = 1) e modificadas (M = 1).

O algoritmo remove uma página qualquer, escolhida aleatoriamente, de uma classe não vazia a partir da classe 1. Observe que o algoritmo dá preferência para remover uma página que tenha sido modificada porém não recentemente referenciada, a uma página não modificada mas recentemente referenciada.

- **LFU**

Nesta estratégia o algoritmo remove a página que tenha sido menos referenciada. Para isto ele precisa manter um contador para cada página, que é zerado por ocasião da carga da página e acrescido de uma unidade cada vez que a mesma é referenciada. O problema desta estratégia é que páginas mais recentemente carregadas tendem a ter menores valores nos seus contadores e páginas fortemente utilizadas durante um período do processamento tendem a se perpetuar na memória, mesmo não sendo mais referenciadas.

e) Segmentação

É a técnica de gerenciamento que possibilita os programas serem subdivididos logicamente em blocos distintos como, por exemplo, área de código e área de dados, e o carregamento de cada um destes blocos em uma área (segmento) distinto da MP. Os blocos, diferentemente das páginas, podem ter tamanhos distintos e são chamados *segmentos*, cada um possuindo seu espaço de endereçamento próprio.

A grande diferença entre paginação e segmentação está no fato de que a primeira divide o programa em partes iguais e de tamanho fixo, sem qualquer relação lógica com a estrutura do mesmo, e a segunda, não fazendo uso de blocos de tamanho fixo, permite a adoção de uma relação entre os blocos e a lógica do programa. Na paginação tem-se um esquema de memória virtual unidimensional, ao passo que na segmentação pode-se adotar um esquema multidimensional. Este fato pode facilitar ao programador sua tarefa de otimizar o desempenho operacional do programa.

Por exemplo, um compilador precisa lidar com muitas tabelas dinâmicas que são construídas, expandidas e encolhidas a medida que a compilação procede, incluindo:

1. *Texto Fonte*, que vai sendo salvo para a impressão da listagem;
2. *Tabela de símbolos*, contendo os nomes e atributos das variáveis;
3. *Tabela de Constantes*, contendo todas as constantes inteiras e de ponto-flutuante utilizadas;
4. *Árvore de parsing*, contendo a análise sintática do programa; e
5. *Pilha*, usada para chamadas de procedimentos dentro do compilador.

As quatro primeiras tabelas crescem continuamente a medida que a compilação procede, já a última pode crescer e diminuir de forma imprevisível.

Num esquema unidimensional as tabelas precisariam ser alocadas em porções contíguas do espaço de endereçamento virtual, como pode ser visto na figura VII.23 abaixo, e, mesmo com a previsão de espaço livre para crescimento, problemas podem ocorrer. Num esquema multidimensional por outro lado, os espaços são separados, independentes e não necessariamente contíguos.

O mecanismo de mapeamento é muito semelhante ao da paginação. Uma *segment table* faz agora as vezes da *page table*, sendo que neste caso cada entrada na tabela possui também a informação sobre o tamanho do segmento. O endereço virtual é formado por um número de segmento e um deslocamento (*offset*) dentro do segmento.

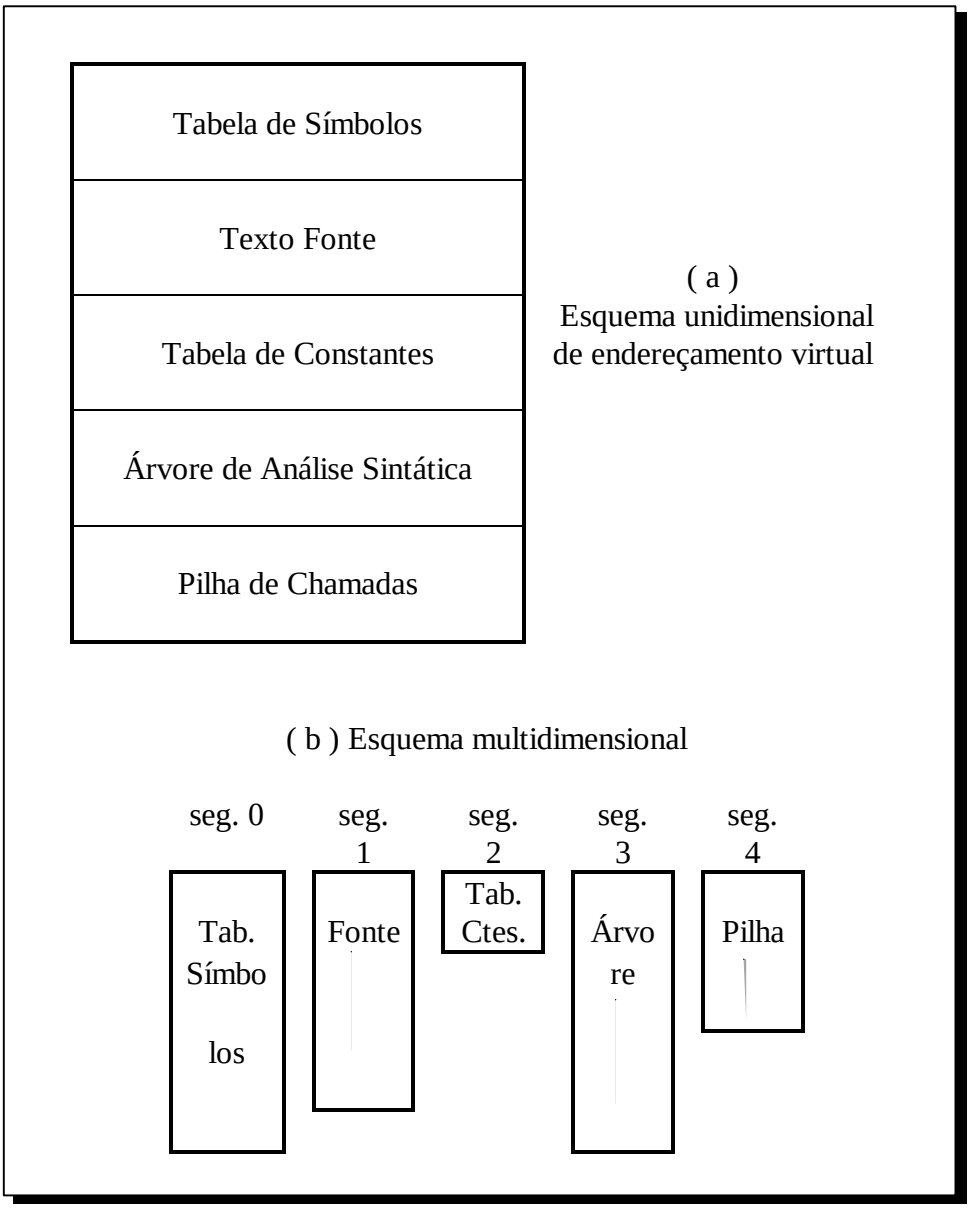


Figura VII.23 - Espaço uni e multidimensional de endereçamento da memória virtual

A técnica de segmentação elimina a fragmentação interna característica da técnica de paginação, porém volta a introduzir a possibilidade de ocorrência de segmentação externa, isto é, entre segmentos.

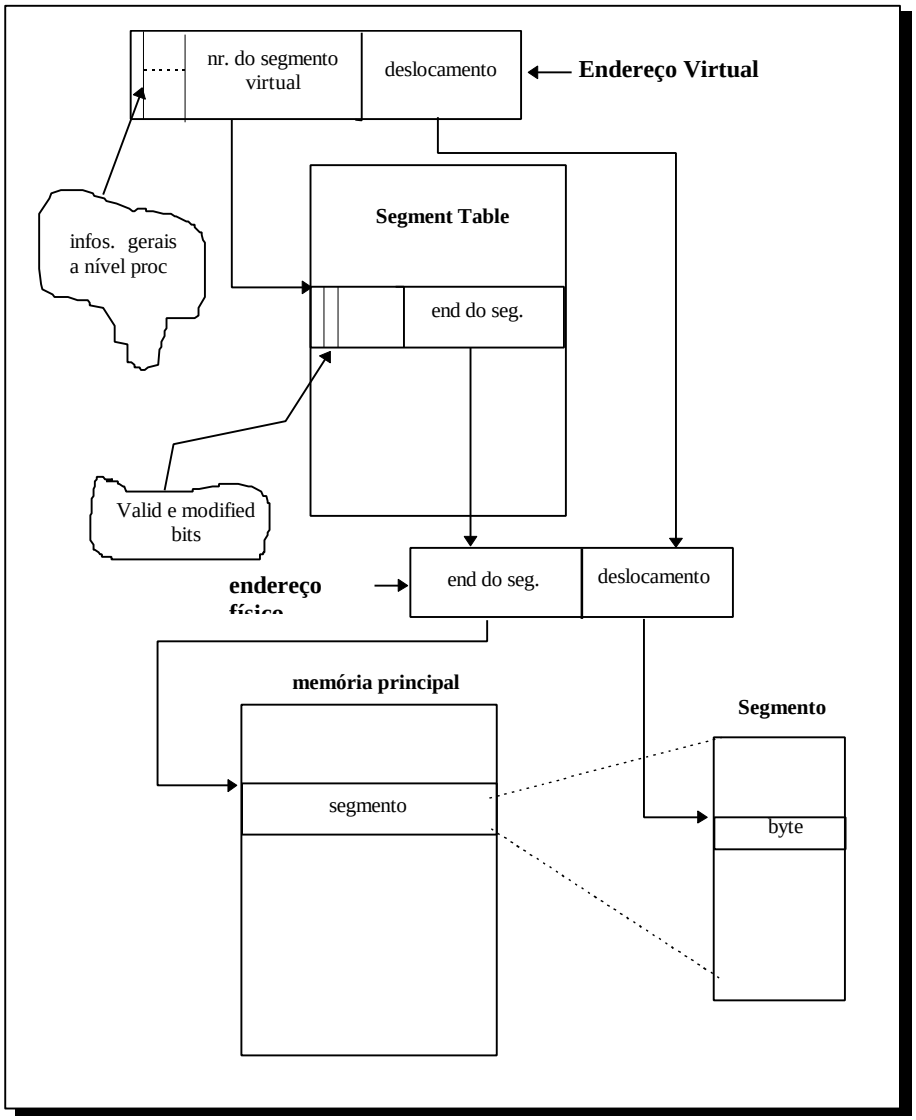


Figura VII.24 - Esquema de Mapeamento com Segmentação

f) Segmentação_com_Paginação

Conceitualmente é a reunião das técnicas de segmentação e de paginação. O processo é logicamente segmentado (como na técnica de segmentação) e agora, dentro de cada segmento ele é subdividido em páginas de tamanho fixo.

O endereço virtual agora é formado por um número de segmento, um número de página e um deslocamento (*offset*) dentro da página. O mapeamento requer uma etapa a mais do que o necessário para as duas técnicas anteriores, conforme pode ser visto na figura VII.25 a seguir.

Fazendo-se uma análise das 3 estratégias de organização do esquema virtual, pode-se constatar que:

a) na paginação a memória é plana e contínua (unidimensional) e pode ocorrer o problema da segmentação interna, ou seja, o processo não utilizar completamente a última página a ele alocada;
b) na segmentação a memória pode ser alocada por partes e não contínua (multidimensional) e desaparece o problema da fragmentação interna, mas volta a aparecer o da segmentação externa;
c) na segmentação com paginação tem-se a forma de alocação segmentada e a fragmentação interna, bem emnor e mais simples de ser gerenciada.

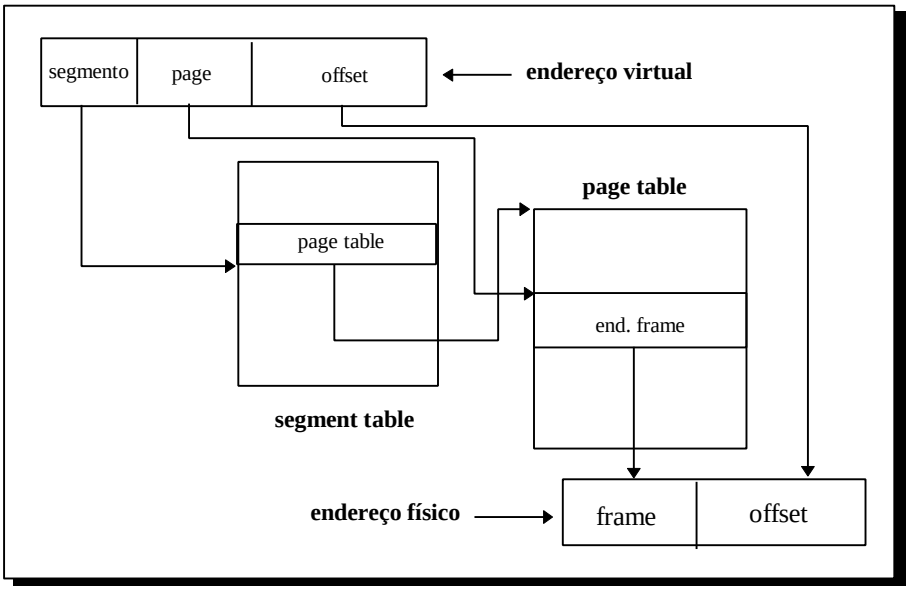


Figura VII.25- Esquema de Segmentação com Paginação

g) Proteção

É realizada a nível de página ou segmento (dependendo da técnica adotada) e vem especificada nos primeiros bits de cada entrada na *page* ou *segment table*.

exemplo:

R	W	descrição
0	0	sem acesso
1	0	apenas leitura
1	1	leitura e gravação

h) Compartilhamento de Memória (código reentrante)

O código executável de alguns programas, principalmente utilitários e programas de sistema, podem ser compartilhados afim de evitar o consumo desnecessário de MP. Para que isto ocorra o programa reentrante precisa ser especificamente caracterizado como tal pelo gerente do sistema e, assim, automaticamente, o sistema operacional se encarrega de fazer com que os processos que estiverem usando um determinado programa reentrante estejam com as entradas das suas respectivas *page* ou *segment tables* apontando para os mesmos frames na MP (frames estes ocupados pelo código executável do programa reentrante). Na figura VII.26 abaixo pode ser observado o caso em que dois processos A e B, compartilham o mesmo código de um terceiro processo, reentrante, que reside em MP.

A utilização de código reentrante não só otimiza a ocupação da MP, uma vez que evita a coexistência de cópias idênticas de um mesmo trecho de programa na memória, mas também melhora o desempenho dos programas, uma vez que reduz a paginação ou segmentação destes pelo fato do programa reentrante permanecer residente em memória.

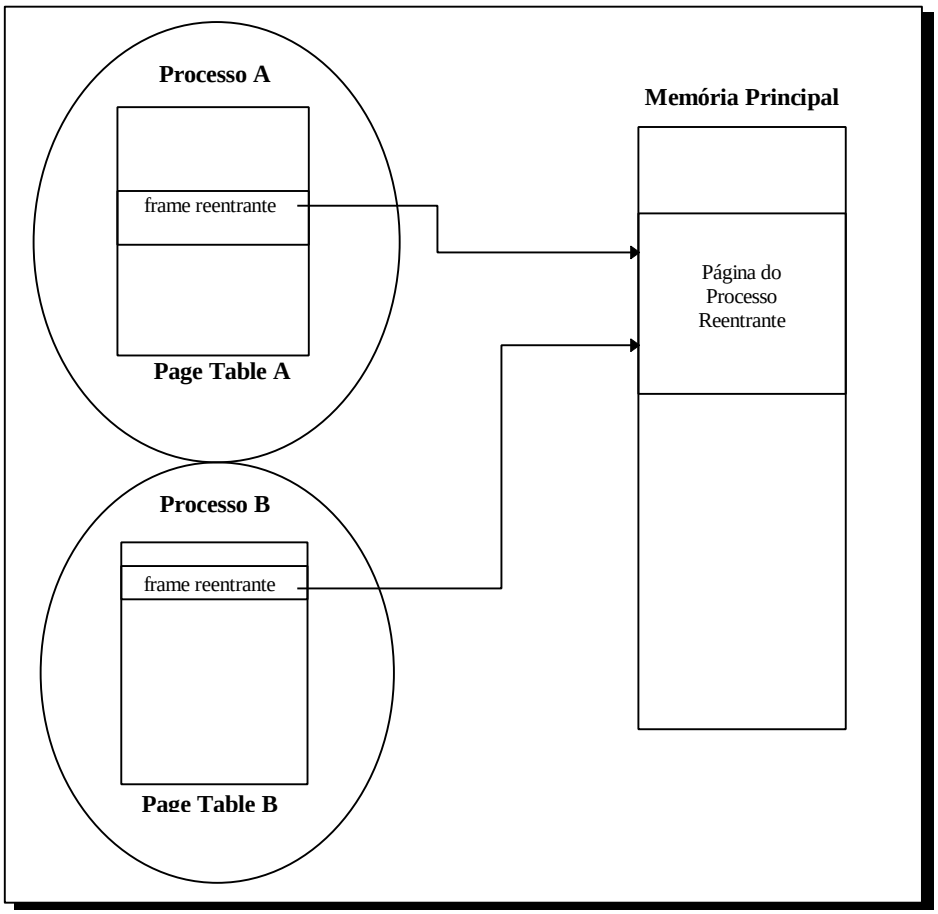


Figura VII.26 - Processo Reentrante

VII.5 Referências Bibliográficas

- Davis, William S., *Sistemas Operacionais - Uma Visão Sistemática*, Ed. Campus, 1990.
- Machado, Francis B. e Maia, Luiz P., *Introdução à Arquitetura de Sistemas Operacionais*, Ed. LTC, 1994.
- Silberschatz, Abraham e Galvin, Peter B., *Operating System Concepts*, 4ª edição, Ed. Addison-Wesley Publishing Company, 1994.
- Stallings, William, *Operating Systems*, Ed. Macmillan Publishing Company, 1992.
- Tanenbaum, Andrew S., *Organização Estruturada de Computadores*, Ed. Prentice-Hall do Brasil, 1990.
- Tanenbaum, Andrew S., *Sistemas Operacionais - Projeto e Implementação*, Ed. Prentice-Hall do Brasil, 1987.
- Tanenbaum, Andrew S., *Sistemas Operacionais Modernos*, Ed. Campus, 1995.