GERÊNCIA DE MEMÓRIA VIRTUAL

1. INTRODUÇÃO

Memória virtual é uma técnica sofisticada e poderosa de gerência de memória, onde as memórias principal e secundária são combinadas, dando ao usuário a ilusão de existir uma memória muito maior que a capacidade real da memória principal. O conceito de memória virtual fundamenta-se em não vincular o endereçamento feito pelo programa aos endereços físicos da memória principal. Desta forma, programas e suas estruturas de dados deixam de estar limitados ao tamanho da memória física disponível, pois podem possuir endereços associados à memória secundária.

Outra vantagem da técnica de memória virtual é permitir um número maior de processos compartilhando a memória principal, já que apenas partes de cada processo estarão residentes. Isto leva a uma utilização mais eficiente também do processador. Além disso, essa técnica possibilita minimizar o problema da fragmentação da memória principal.

A primeira implementação de memória virtual foi realizada no início da década de 1960, no sistema Atlas, desenvolvido na Universidade de Manchester. Posteriormente, a IBM introduziria este conceito comercialmente na família System/370 em 1972. Atualmente, a maioria dos sistemas implementa memória virtual, com exceção de alguns sistemas operacionais de supercomputadores.

Existe um forte relacionamento entre a gerência da memória virtual e a arquitetura de hardware do sistema computacional. Por motivos de desempenho, é comum que algumas funções da gerência de memória virtual sejam implementadas diretamente no hardware. Além disso, o código do sistema operacional deve levar em consideração várias características específicas da arquitetura, especialmente o esquema de endereçamento do processador.

2. ESPAÇO DE ENDEREÇAMENTO VIRTUAL

O conceito de memória virtual se aproxima muito da idéia de um vetor, existente nas linguagens de alto nível. Quando um programa faz referência a um elemento do vetor, não há preocupação em saber a posição de memória daquele dado. O compilador se encarrega de gerar instruções que implementam esse mecanismo, tornando-o totalmente transparente ao programador.

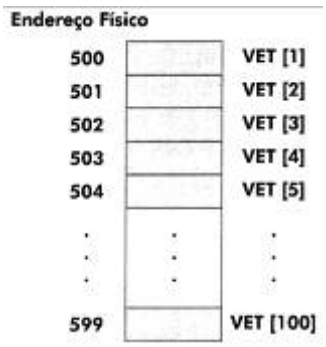


Figura 1: Vetor de 100 posições

A memória virtual utiliza abstração semelhante, só que em relação aos endereços dos programas e dados. Um programa no ambiente de memória virtual não faz referência a endereços físicos de memória (endereços reais), mas apenas a endereços virtuais. No momento da execução de uma instrução, o endereço virtual referenciado é traduzido para um endereço físico, pois o processador manipula apenas posições da memória principal. O mecanismo de tradução do endereço virtual para endereço físico é denominado mapeamento.

Em ambientes que implementam memória virtual, o espaço de endereçamento do processo é conhecido como espaço de endereçamento virtual e representa o conjunto de endereços virtuais que o processo pode endereçar. Analogamente, o conjunto de endereços reais que o processador pode referenciar é chamado de espaço de endereçamento real.

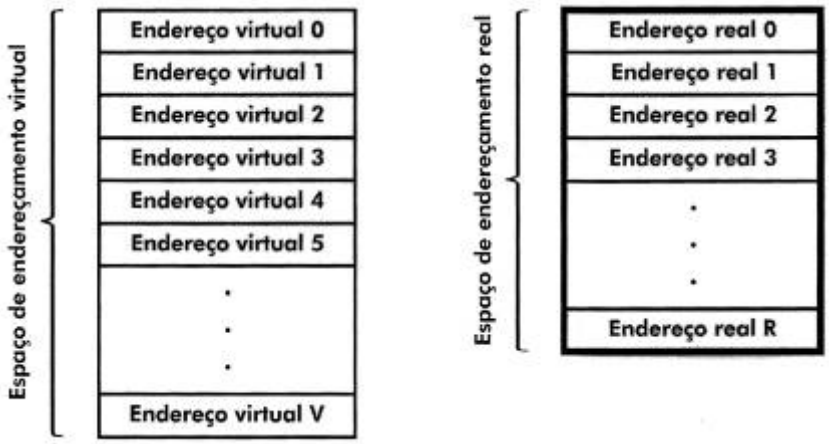


Figura 2: Espaço de endereçamentos virtual e real

Como o espaço de endereçamento virtual não tem nenhuma relação direta com os endereços no espaço real, um programa pode fazer referência a endereços virtuais que estejam fora dos limites da memória principal, ou seja, os programas e suas estruturas de dados não estão mais limitados ao tamanho da memória física disponível. Para que isso seja possível, o sistema operacional utiliza a memória secundária como extensão da

memória principal. Quando um programa é executado, somente uma parte do seu código fica residente na memória principal, permanecendo o restante na memória secundária até o momento de ser referenciado. Esta condição permite aumentar o compartilhamento da memória principal entre muitos processos.

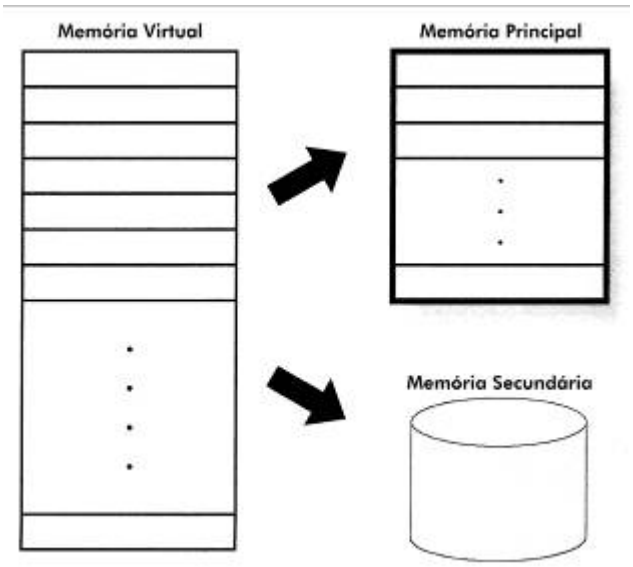


Figura 3: Espaço de endereçamento virtual

No desenvolvimento de aplicações, a existência dos endereços virtuais é ignorada pelo programador. Os compiladores e linkers se encarregam de gerar o código executável em função do espaço de endereçamento virtual, e o sistema operacional cuida dos detalhes durante sua execução.

3. MAPEAMENTO

O processador apenas executa instruções e referencia dados residentes no espaço de endereçamento real; portanto, deve existir um mecanismo que transforme os endereços virtuais em endereços reais. Esse mecanismo, conhecido por *mapeamento*, permite traduzir um endereço localizado no espaço virtual para um associado no espaço real. Como consequência do mapeamento, um programa não mais precisa estar necessariamente em endereços contíguos na memória principal para ser executado.

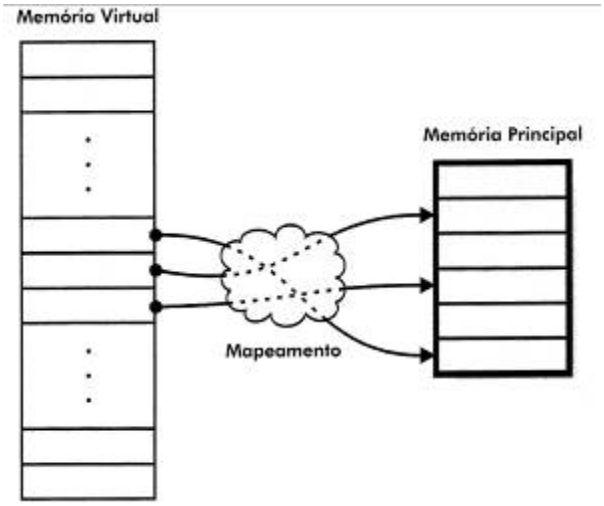


Figura 4: Mapeamento

Nos sistemas modernos, a tarefa de tradução de endereços virtuais é realizada por hardware juntamente com o sistema operacional, de forma a não comprometer seu desempenho e torná-lo transparente a usuários e suas aplicações. O dispositivo de hardware responsável por esta tradução é conhecido como unidade de gerência de memória (Memory Management Unit – MMU), sendo acionado sempre que se faz referência a um endereço virtual. Depois de traduzido, o endereço real pode ser utilizado pelo processador para o acesso à memória principal.

Cada processo tem o seu espaço de endereçamento virtual como se possuísse sua própria memória. O mecanismo de tradução se encarrega, então, de manter tabelas de mapeamento exclusivas para cada processo, relacionando os endereços virtuais do processo às suas posições na memória real.

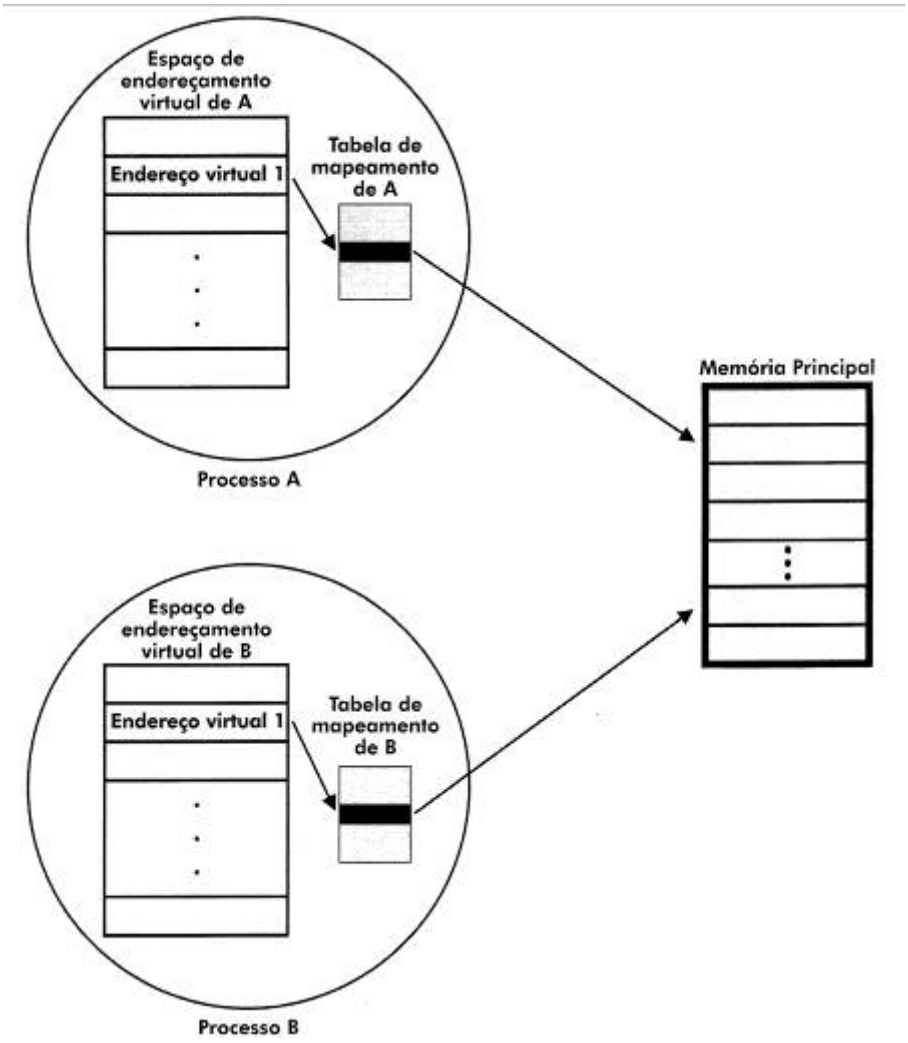


Figura 5: Tabela de mapeamento

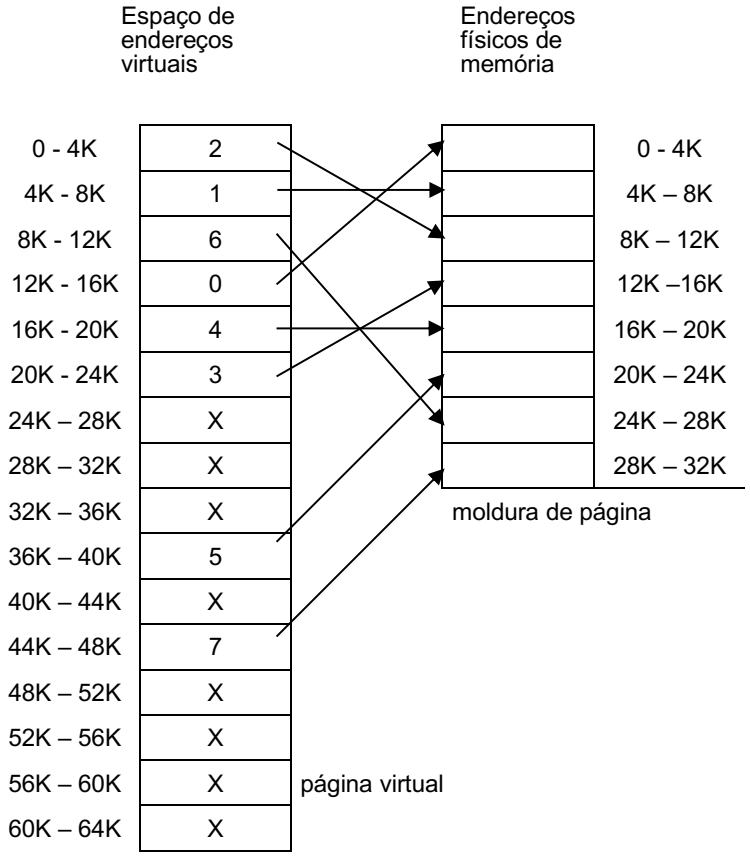
A tabela de mapeamento é uma estrutura de dados existente para cada processo. Quando um determinado processo está sendo executado, o sistema utiliza a tabela de mapeamento do processo em execução para realizar a tradução de seus endereços virtuais. Se um outro processo vai ser executado, o sistema deve passar a referenciar a tabela de mapeamento do novo processo. A troca de tabelas de mapeamento é realizada através de um registrador, que indica a posição inicial da tabela corrente, onde, toda vez que há mudança de contexto, o registrador é atualizado com o endereço da nova tabela.

Caso o mapeamento fosse realizado para cada célula na memória principal, o espaço ocupado pelas tabelas seria tão grande quanto o espaço de endereçamento virtual de cada processo, o que inviabilizaria a implementação do mecanismo. Em função disso, as tabelas mapeiam blocos de dados, cujo tamanho determina o número de entradas existentes nas tabelas de mapeamento. Quanto maior o bloco, menos entradas nas tabelas de mapeamento e, conseqüentemente, tabelas de mapeamento que ocupam um menor espaço de memória.

Espaço de endereçamento virtual	*Tamanho do bloco*	*Número de blocos*	*Número de entradas na tabela de mapeamento*
2^{32} *endereços*	*512 endereços*	2^{23}	2^{23}
2^{32} *endereços*	*4k endereços*	2^{20}	2^{20}
2^{64} *endereços*	*4k endereços*	2^{52}	2^{52}
2^{64} *endereços*	*64k endereços*	2^{48}	2^{48}

3.1. Exemplo de Mapeamento

Um exemplo de como o mapeamento trabalha é mostrado abaixo:



Neste exemplo, temos um computador que pode gerar endereços de 16 bits, de 0 até 64K, correspondentes ao espaço de endereçamento virtual da máquina. No entanto, este computador tem apenas 32K de memória física, de forma que apesar de podermos escrever programas com até 64K, estes não podem ser totalmente carregados na memória e executados. Uma cópia completa do programa deve sempre ser mantida no disco, de forma que pedaços dele possam ser colocados na memória quando necessário.

Quando o programa tenta acessar o endereço 0, por exemplo, usando a instrução

MOVE REG, 0

o endereço virtual 0 é enviado à MMU. Esta conclui que este endereço virtual cai na página 0 (endereços virtuais de 0 a 4.095), que de acordo com seu processo de

mapeamento está na moldura de página 2 (endereços reais de 8.192 a 12.287). Ela então transforma o endereço que lhe foi entregue para 8.192, e coloca este valor no barramento da memória. A placa de memória desconhece a existência da MMU, e simplesmente enxerga uma requisição para ler ou escrever no endereço 8.192, realizando tal acesso. Então, a MMU efetivamente mapeia todos os endereços virtuais de 0 a 4.095 nos endereços reais de 8.192 a 12.287.

Da mesma forma, a instrução

MOVE REG, 8192

é executada como

MOVE REG, 24.576

em virtude do endereço virtual 8.192 encontrar-se na página 2, e esta página ser mapeada na moldura 6 (endereços físicos de 24.576 a 28.671). Em outro exemplo, o endereço virtual 20.500 está a 20 bytes do início da página virtual 5 (endereços virtuais de 20.480 a 24.575) e é mapeado no endereço real 12.288 + 20 = 12.308.

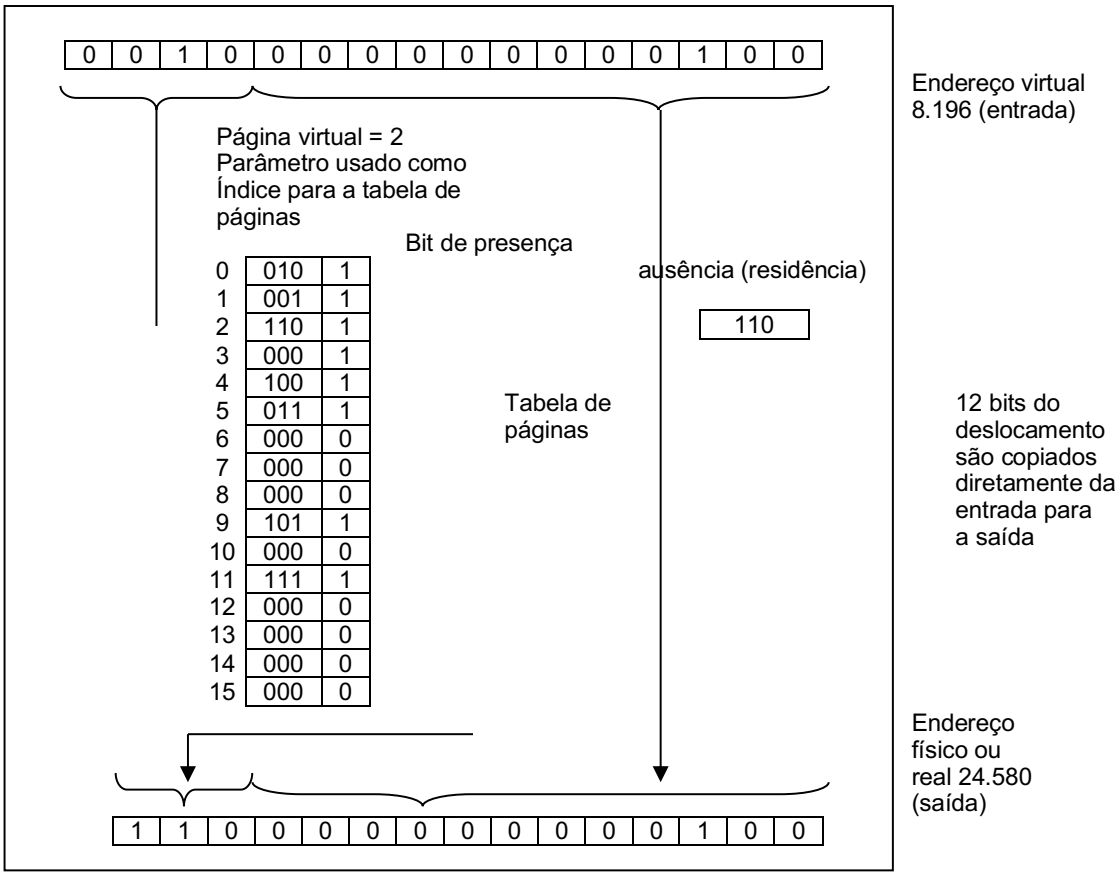
O que ocorre se um programa tenta usar uma página não mapeada, tentando, por exemplo, executar a instrução

MOVE REG, 32780

cujo endereço virtual gerado está 12 bits após a origem da página 8, que tem início em 32.768? A MMU observa que se trata de uma página não-mapeável e força o processador a executar um trap para o sistema operacional. Este trap é denominado falta de página. Como resposta ao trap, o sistema operacional pega a moldura de página menos usada e a copia de volta para o disco. Então ele busca a página referenciada e a copia na moldura que acabou de ser liberada, modifica o mapa e reexecuta a instrução causadora do trap.

Por exemplo, se o sistema operacional resolver substituir a página na moldura 1, ele deve carregar a página virtual 8 no endereço 4K e fazer duas mudanças no mapeamento da MMU. Primeiro, ele deve marcar a entrada correspondente à página virtual 1 como não-mapeável, para que um trap seja gerado quando houver acesso futuro a um endereço virtual desta página. Depois, ele deve substituir o X na entrada correspondente à página virtual 8 por um 1, de maneira que quando a instrução que gerou o trap for reexecutada, deve haver o mapeamento do endereço virtual 32.780 no endereço físico 4.108.

Vamos verificar dentro da MMU, para ver como ela funciona. Na figura abaixo, vemos um exemplo do endereço virtual 8.196 (0010000000000100 em binário), sendo mapeado usando o mapa da MMU da figura anterior.



O endereço virtual de 16 bits que chega à MMU é dividido em um número de página de quatro bits, e em um deslocamento de 12 bits. Com os quatro bits referentes ao número da página, podemos representar 16 páginas, e com os 12 do deslocamento, podemos endereçar todos os 4.096 bytes que compõem uma determinada página.

4. MEMÓRIA VIRTUAL POR PAGINAÇÃO

A memória virtual por paginação é a técnica de gerência de memória onde o espaço de endereçamento virtual e o espaço de endereçamento real são divididos em blocos do mesmo tamanho chamados páginas. As páginas no espaço virtual são denominadas páginas virtuais, enquanto as páginas no espaço real são chamadas de páginas reais ou frames.

Todo o mapeamento de endereço virtual em real é realizado através de tabelas de páginas. Cada processo possui sua própria tabela de páginas e cada página virtual do processo possui uma entrada na tabela (entrada na tabela de páginas – ETP), com informações de mapeamento que permitem ao sistema localizar a página real correspondente.

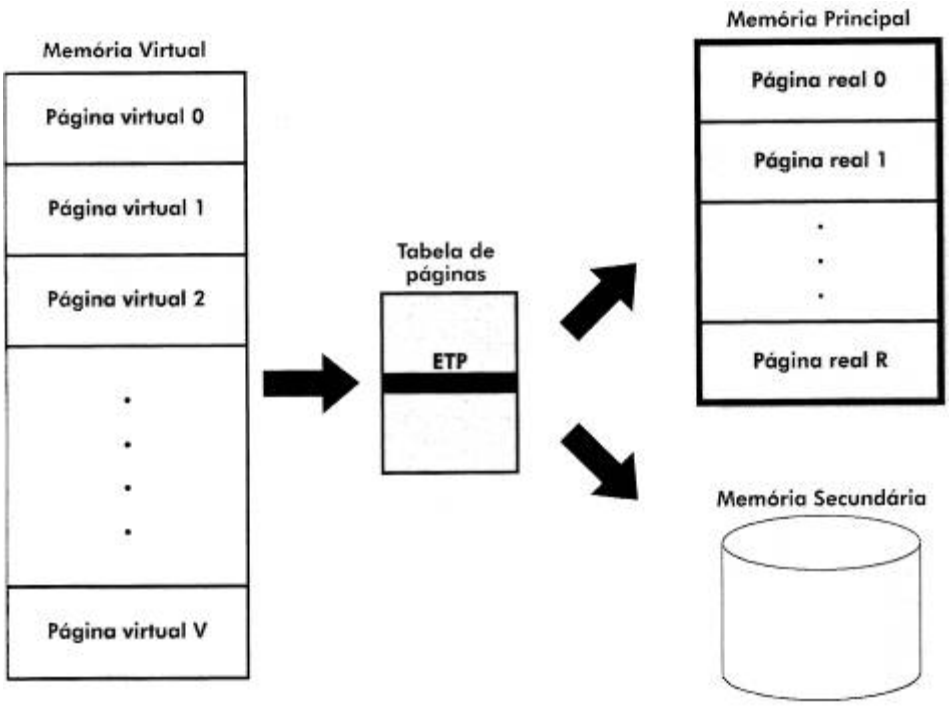


Figura 6: Tabela de páginas

Quando um programa é executado, as páginas virtuais são transferidas da memória secundária para a memória principal e colocadas nos frames. Sempre que um programa fizer referência a um endereço virtual, o mecanismo de mapeamento localizará na ETP da tabela do processo o endereço físico do frame no qual se encontra o endereço real correspondente.

Nessa técnica, o endereço virtual é formado pelo número da página virtual (NPV) e por um deslocamento. O NPV identifica unicamente a página virtual que contém o endereço, funcionando como um índice na tabela de páginas. O deslocamento indica a posição do endereço virtual em relação ao início da página na qual se encontra. O endereço físico é obtido, então, combinando-se o endereço do frame, localizado na tabela de páginas, com o deslocamento, contido no endereço virtual.

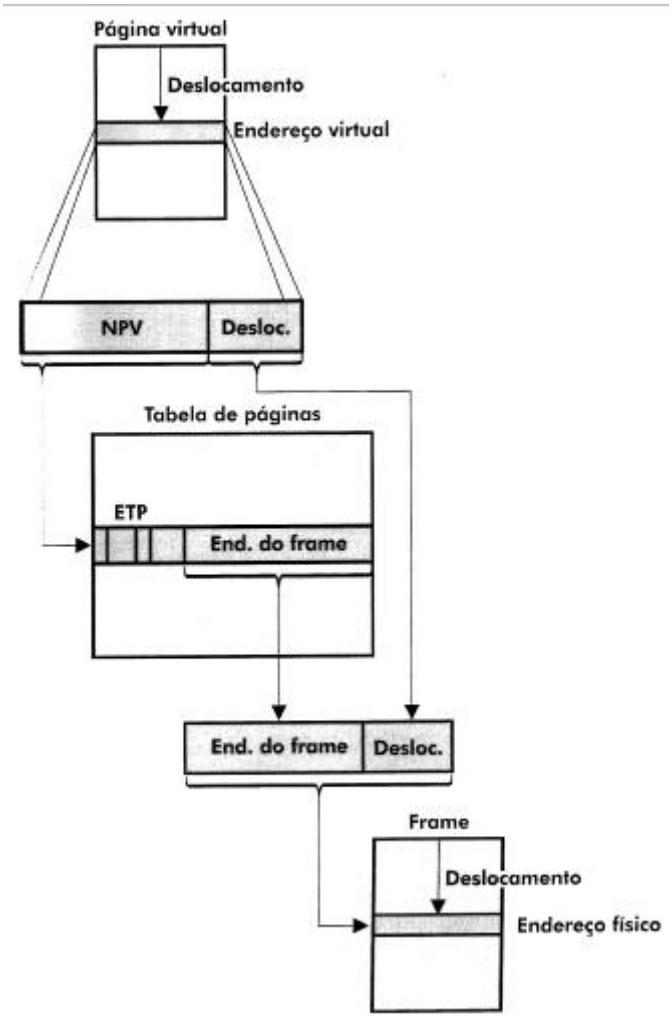


Figura 7: Tradução do endereço virtual

Além da informação sobre a localização da página virtual, a ETP possui outras informações, como o bit de validade (valid bit) que indica se uma página está ou não na memória principal. Se o bit tem valor 0, isto indica que a página virtual não está na memória principal, mas se é igual a 1, a página está localizada na memória.

Sempre que o processo referencia um endereço virtual, a unidade de gerência de memória verifica, através do bit de validade, se a página que contém o endereço referenciado está ou não na memória principal. Caso a página não esteja na memória, dizemos que ocorreu uma falta de página (page fault). Neste caso, o sistema transfere a página da memória secundária para a memória principal, realizando uma operação de E/S conhecida como page in ou paginação. O número de page faults gerado por um processo depende de como o programa foi desenvolvido, além da política de gerência de memória implementada pelo sistema operacional.

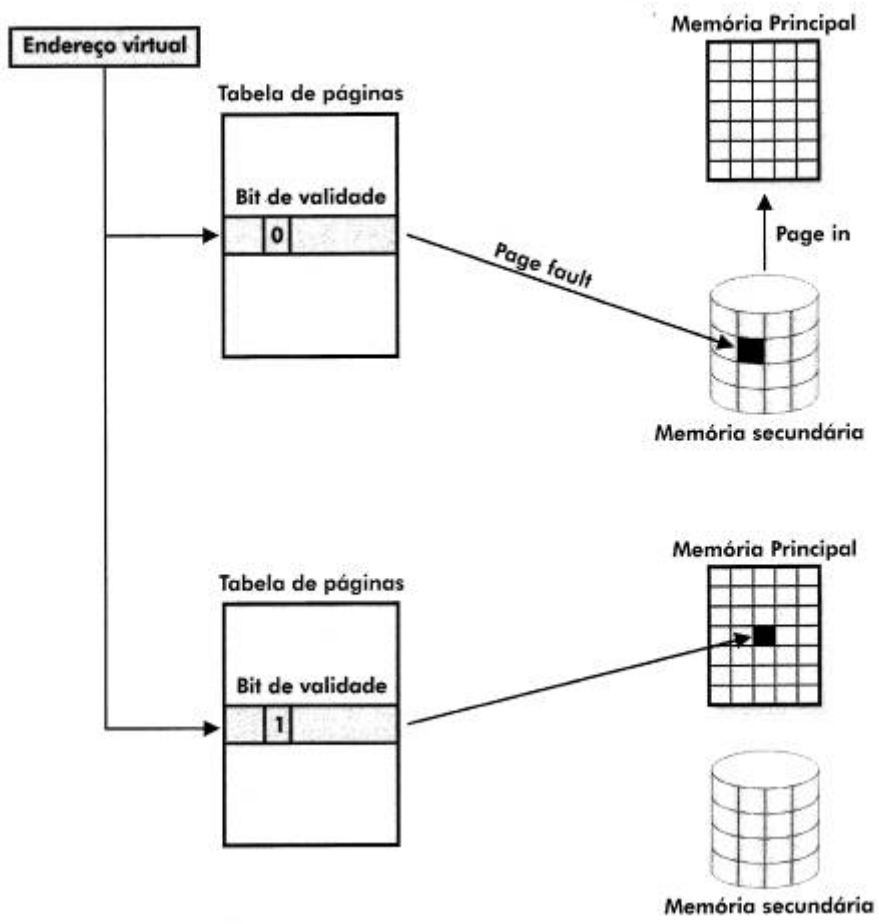


Figura 8: Mecanismo de tradução

O número de page faults gerado por um processo em um determinado intervalo de tempo é definido como taxa de paginação do processo. O overhead gerado pelo mecanismo de paginação é inerente da gerência de memória virtual, porém se a taxa de paginação dos processos atingir valores elevados, o excesso de operações de E/S poderá comprometer o desempenho do sistema.

Quando um processo referencia um endereço e ocorre um page fault, o processo em questão passa do estado de execução para o estado de espera, até que a página seja transferida do disco para a memória principal. Na troca de contexto, as informações sobre a tabela de mapeamento são salvas e as informações do novo processo escalonado são restauradas. Após a transferência da página para a memória principal, o processo é recolocado na fila de processos no estado de pronto, e quando for reescalonado poderá continuar sua execução.

4.1. Políticas de Busca de Páginas

O mecanismo de memória virtual permite a execução de um programa sem que seu código esteja completamente residente na memória principal. A política de busca de

páginas determina quando uma página deve ser carregada para a memória. Basicamente, existem duas estratégias para este propósito: paginação por demanda e paginação antecipada.

Na paginação por demanda (demand paging), as páginas dos processos são transferidas da memória secundária para a principal apenas quando são referenciadas. Este mecanismo é conveniente, na medida em que leva para a memória principal apenas as páginas realmente necessárias à execução do programa. Desse modo, é possível que partes não executadas do programa, como rotinas de tratamento de erros, nunca sejam carregadas para a memória.

Na paginação antecipada (anticipatory paging ou prepaging), o sistema carrega para a memória principal, além da página referenciada, outras páginas que podem ou não ser necessárias ao processo ao longo do seu processamento. Se imaginarmos que o programa está armazenado seqüencialmente no disco, existe uma grande economia de tempo em levar um conjunto de páginas da memória secundária, ao contrário de carregar uma de cada vez. Por outro lado, caso o processo não precise das páginas carregadas antecipadamente, o sistema terá perdido tempo e ocupado memória principal desnecessariamente.

A técnica de paginação antecipada pode ser empregada no momento da criação de um processo ou na ocorrência de um page fault. Quando um processo é criado, diversas páginas do programa na memória secundária devem ser carregadas para a memória principal, gerando um elevado número de page faults e várias operações de leitura em disco. Na medida em que as páginas são carregadas para a memória, a taxa de paginação tende a diminuir. Se o sistema carregar não apenas uma, mas um conjunto de páginas, a taxa de paginação do processo deverá cair imediatamente e estabilizar-se durante um certo período de tempo. Seguindo o mesmo raciocínio, sempre que houver um page fault, o sistema poderá carregar para a memória, além da página referenciada, páginas adicionais, na tentativa de evitar novos page faults e sucessivas operações de leitura em disco.

4.2. Política de Alocação de Páginas

A política de alocação de páginas determina quantos frames cada processo pode manter na memória principal. Existem, basicamente, duas alternativas: alocação fixa e alocação variável.

Na política de alocação fixa, cada processo tem um número máximo de frames que pode ser utilizado durante a execução do programa. Caso o número de páginas reais seja insuficiente, uma página do processo deve ser descartada para que uma nova seja carregada. O limite de páginas reais pode ser igual para todos os processos ou definido individualmente. Apesar de parecer justo alocar o mesmo número de páginas para todos os processos, pode não ser uma boa opção, pois a necessidade de memória de cada processo

raramente é a mesma. O limite de páginas deve ser definido no momento da criação do processo, com base no tipo de aplicação que será executada. Essa informação faz parte do contexto de software do processo.

Apesar de sua simplicidade, a política de alocação fixa de páginas apresenta dois problemas. Se o número máximo de páginas alocadas for muito pequeno, o processo tenderá a ter um elevado número de page faults, o que pode impactar no desempenho de todo o sistema. Por outro lado, caso o número de páginas seja muito grande, cada processo irá ocupar na memória principal um espaço maior do que o necessário, reduzindo o número de processos residentes e o grau de multiprogramação. Nesse caso, o sistema pode implementar a técnica de swapping, retirando e carregando processos da/para a memória principal.

Na política de alocação variável, o número máximo de páginas alocadas ao processo pode variar durante sua execução em função de sua taxa de paginação e da ocupação da memória principal. Nesse modelo, processos com elevadas taxas de paginação podem ampliar o limite máximo de frames, a fim de reduzir o número de page faults. Da mesma forma, processos com baixas taxas de paginação podem ter páginas realocadas para outros processos. Este mecanismo, apesar de mais flexível, exige que o sistema operacional monitore constantemente o comportamento dos processos, gerando maior overhead.

4.3. Políticas de Substituição de Páginas

Em algumas situações, quando um processo atinge o seu limite de alocação de frames e necessita alocar novas páginas na memória principal, o sistema operacional deve selecionar, dentre as diversas páginas alocadas, qual deverá ser liberada. Este mecanismo é chamado de política de substituição de páginas. Uma página real, quando liberada por um processo, está livre para ser utilizada por qualquer outro processo. A partir dessa situação, qualquer estratégia de substituição de páginas deve considerar se uma página foi ou não modificada antes de liberá-la; caso contrário, os dados armazenados na página podem ser perdidos. No caso de páginas contendo código executável, que não sofrem alterações, não existe essa preocupação, pois existe uma cópia do código no arquivo executável em disco. As páginas modificáveis, que armazenam variáveis e estruturas de dados, podem sofrer alterações. Neste caso, o sistema deverá gravá-la na memória secundária antes do descarte, preservando seu conteúdo para uso em futuras referências. Este mecanismo é conhecido como page out. Com este propósito, o sistema mantém um arquivo de paginação (page file) onde todas as páginas modificadas e descartadas são armazenadas. Sempre que uma página modificada for novamente referenciada, ocorrerá um page in, carregando-a para a memória principal a partir do arquivo de paginação.

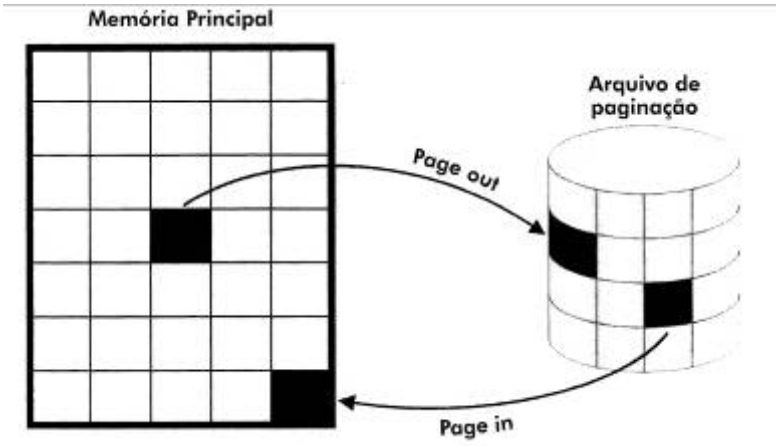


Figura 9: Substituição de página

O sistema operacional consegue identificar as páginas modificadas através de um bit que existe em cada entrada da tabela de páginas, chamado bit de modificação (dirty bit ou modify bit). Sempre que uma página sofre uma alteração, o valor do bit de modificação é alterado, indicando que a página foi modificada.

A política de substituição de páginas pode ser classificada conforme seu escopo, ou seja, dentre os processos residentes na memória principal quais são candidatos a ter páginas realocadas. Em função desse escopo, a política de substituição pode ser definida como local ou global.

Na política de substituição local, apenas as páginas do processo que gerou o page fault são candidatas a realocação. Nesse modelo, sempre que um processo precisar de uma nova página, o sistema deverá selecionar, dentre os frames alocados pelo processo, a página a ser substituída. Os frames dos demais processos não são avaliados para substituição.

Já na política de alocação global, todas as páginas alocadas na memória principal são candidatas a substituição, independente do processo que gerou o page fault. Como qualquer processo pode ser escolhido, é possível que o processo selecionado sofra um aumento na sua taxa de paginação, em função da redução do número de páginas alocadas na memória. Na verdade, nem todas as páginas podem ser candidatas a substituição. Algumas páginas, como as do núcleo do sistema, são marcadas como bloqueadas e não podem ser realocadas.

Existe uma relação entre o escopo da política de substituição e a política de alocação de páginas. A política de alocação fixa permite apenas a utilização de uma política de substituição local. Nesse caso, sempre que um processo necessita de uma nova página, o sistema deverá selecionar um frame do próprio processo para ser realocado, mantendo assim o seu limite de páginas.

A política de alocação variável permite uma política de substituição tanto local quanto global. Na política de alocação variável com substituição global, quando um processo

necessita de uma nova página, o sistema poderá selecionar um frame dentre todas as páginas na memória principal, independente do processo. Nesse caso, o processo escolhido perde uma de suas páginas, reduzindo assim o número de frames alocados na memória pelo processo. Por outro lado, o processo que gerou o page fault recebe um novo frame e tem seu número de páginas aumentado.

Na política de alocação variável com substituição local, quando um processo necessita de nova página o sistema deverá selecionar uma página do próprio processo para substituição. Em função do comportamento do processo e do nível de utilização do sistema, o número de páginas alocadas ao processo pode ser aumentado ou diminuído, a fim de melhorar o desempenho do sistema.

4.4. Conjunto de Trabalho (Working Set)

Apesar de suas diversas vantagens, o mecanismo de memória virtual introduz um sério problema. Como cada processo possui na memória principal apenas algumas páginas alocadas, o sistema deve manter um conjunto mínimo de frames buscando uma baixa taxa de paginação. Ao mesmo tempo, o sistema operacional deve impedir que os processos tenham um número excessivo de páginas na memória, de forma a aumentar o grau de compartilhamento da memória principal.

Caso os processos tenham na memória principal um número insuficiente de páginas para a execução do programa, é provável que diversos frames referenciados ao longo do seu processamento não estejam na memória. Esta situação provoca a ocorrência de um número elevado de page faults e, conseqüentemente, inúmeras operações de E/S. Neste caso, ocorre um problema conhecido como trashing, provocando sérias conseqüências ao desempenho do sistema.

O conceito de working set surgiu com o objetivo de reduzir o problema do trashing e está relacionado ao princípio da localidade. Existem dois tipos de localidades que são observados durante a execução da maioria dos programas. A localidade espacial é a tendência de que após uma referência a uma posição de memória sejam realizadas novas referências a endereços próximos. A localidade temporal é a tendência de que após a referência a uma posição de memória esta mesma posição seja novamente referenciada em um curto intervalo de tempo.

O princípio da localidade significa que o processador tenderá a concentrar suas referências a um conjunto de páginas do processo durante um determinado período de tempo. Imaginando um loop, cujo código ocupe três páginas, a tendência de essas três páginas serem referenciadas diversas vezes é muito alta.

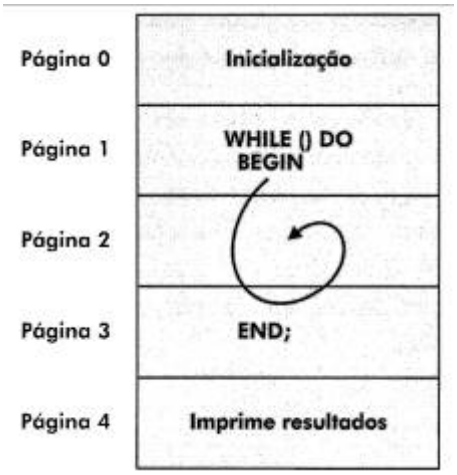


Figura 10: Conceito de localidade

No início da execução de um programa, observa-se um elevado número de page faults, pois não existe nenhum frame do processo na memória principal. Com o decorrer da sua execução, as páginas são carregadas para a memória e o número de page faults diminui. Após um período de estabilidade, o programa gera novamente uma elevada taxa de paginação, que depois de algum tempo volta a se estabilizar. Esse fenômeno pode repetir-se inúmeras vezes durante a execução de um processo e está relacionado com a forma com que a aplicação foi escrita. Normalmente, se um programa foi desenvolvido utilizando técnicas estruturadas, o conceito de localidade quase sempre é válido. Nesse caso, a localidade será percebida, por exemplo, durante a execução de repetições e sub-rotinas.

O princípio da localidade é indispensável para que a gerência de memória virtual funcione eficientemente. Como as referências aos endereços de um processo concentram-se em um determinado conjunto de páginas, é possível manter apenas parte do código de cada um dos diversos programas na memória principal, sem prejudicar a execução dos processos. Caso contrário, o sistema teria que manter integralmente o código de todos os programas na memória para evitar o problema do trashing. Considerando um programa com rotinas de inicialização, um loop principal e rotinas de finalização, manter o programa inteiro na memória principal seria ineficiente. A má utilização da memória fica mais clara quando o programa possui rotinas de tratamento de erros na memória que, por muitas vezes, nunca serão executadas.

A partir da observação do princípio da localidade, Peter Denning formulou o modelo de working set. O conceito de working set é definido como sendo o conjunto das páginas referenciadas por um processo durante determinado intervalo de tempo. A figura 11 ilustra que no tempo t_2, o working set do processo, $W(t_2, t)$, são as páginas referenciadas no intervalo t ($t_2 - t_1$), isto é, as páginas P2, P3 e P8. O intervalo de tempo t é denominado janela do working set. Podemos observar, então, que o working set de um processo é uma função do tempo e do tamanho da janela do working set.

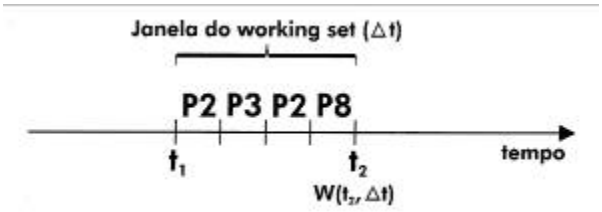


Figura 11: Modelo de working set

Dentro da janela do working set, o número de páginas distintas referenciadas é conhecido como tamanho do working set. Na figura 12, são apresentadas as referências às páginas de um processo nas janelas t_a ($t_2 - t_1$) e t_b ($t_3 - t_2$). O working set do processo no instante t_2, com a janela t_a, corresponde às páginas P2, P3, P4 e P5, e o tamanho do working set é igual a quatro páginas. No instante t_3, com a janela t_b, o working set corresponde às páginas P5 e P6, e o tamanho do working set é igual a duas páginas.

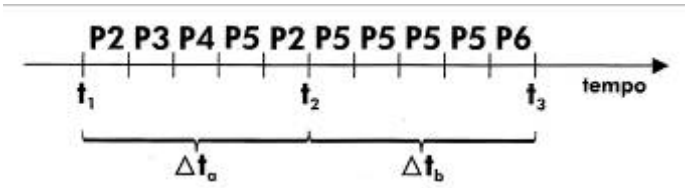


Figura 12: Tamanho do working set

O modelo de working set proposto por Denning possibilita prever quais páginas são necessárias à execução de um programa de forma eficiente. Caso a janela do working set seja apropriadamente selecionada, em função da localidade do programa, o sistema operacional deverá manter as páginas do working set de cada processo residentes na memória principal. Considerando que a localidade de um programa varia ao longo da sua execução, o tamanho do working set do processo também varia, ou seja, o seu limite de páginas reais deve acompanhar esta variação. O working set refletirá a localidade do programa reduzindo a taxa de paginação dos processos e evitando, consequentemente, o trashing.

Caso o limite de páginas reais de um processo seja maior do que o tamanho do working set, menor será a chance de ocorrer uma referência a uma página que não esteja na memória principal. Por outro lado, as páginas dos processos ocuparão excessivo espaço, reduzindo o grau de compartilhamento da memória. No caso do limite de páginas reais ser menor, a taxa de paginação será alta, pois parte do working set não estará residente na memória principal. Outro fato que pode ser observado é a existência de um ponto onde o aumento do limite de páginas reais do processo não implica a diminuição significativa da taxa de paginação, sendo este ponto alcançado muito antes do programa ser totalmente carregado para a memória.

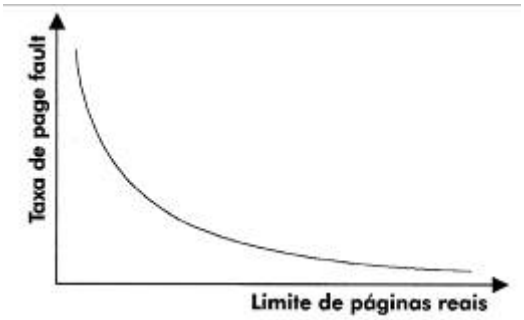


Figura 13: taxa de page fault x limite de páginas reais

Apesar do conceito de working set ser bastante intuitivo, sua implementação não é simples, por questões de desempenho. Para implementar esse modelo, o sistema operacional deve garantir que o working set de cada processo permaneça na memória principal, determinando quais páginas devem ser mantidas e retiradas em função da última janela de tempo. Em função disso, o modelo de working set deve ser implementado somente em sistemas que utilizam a política de alocação de páginas variável, onde o limite de páginas reais não é fixo.

Uma maneira de implementar o modelo de working set é analisar a taxa de paginação de cada processo, conhecida como estratégia de freqüência de page fault. Caso um processo tenha uma taxa de paginação acima de um limite definido pelo sistema, o processo deverá aumentar o seu limite de páginas reais na tentativa de alcançar o seu working set. Por outro lado, se o processo tem uma taxa de paginação abaixo de um certo limite, o sistema poderá reduzir o limite de páginas sem comprometer seu desempenho. O sistema operacional OpenVMS implementa uma estratégia semelhante a esta. Na prática, o modelo de working set serve como base para inúmeros algoritmos de substituição de páginas.

4.5. Algoritmos de Substituição de Páginas

O maior problema na gerência de memória virtual por paginação não é decidir quais páginas carregar para a memória principal, mas quais liberar. Quando um processo necessita de uma nova página e não existem frames disponíveis, o sistema deverá selecionar, dentre as diversas páginas alocadas na memória, qual deverá ser liberada pelo processo.

Os algoritmos de substituição de páginas têm o objetivo de selecionar os frames que tenham as menores chances de serem referenciados em um futuro próximo; caso contrário, o frame poderia retornar diversas vezes para a memória principal, gerando vários page faults e acessos à memória secundária. A partir do princípio da localidade, a maioria dos algoritmos tenta prever o comportamento futuro das aplicações em função do comportamento passado, avaliando o número de vezes que uma página foi referenciada, o momento em que foi carregada para a memória principal e o intervalo de tempo da última referência.

A melhor estratégia de substituição de páginas seria aquela que escolhesse um frame que não fosse mais utilizado no futuro ou levasse mais tempo para ser novamente referenciado. Porém, quanto mais sofisticado o algoritmo de substituição deve tentar manter o working set dos processos na memória principal e, ao mesmo tempo, não comprometer o desempenho do sistema.

a. Ótimo

O *algoritmo ótimo* seleciona para substituição uma página que não será mais referenciada no futuro ou aquela que levará o maior intervalo de tempo para ser novamente utilizada. Apesar deste algoritmo garantir as menores taxas de paginação para os processos, na prática é impossível de ser implementado, pois o sistema operacional não tem como conhecer o comportamento futuro das aplicações. Essa estratégia é utilizada apenas como modelo comparativo na análise de outros algoritmos de substituição.

b. Aleatório

O *algoritmo aleatório* não utiliza critério algum de seleção. Todas as páginas alocadas na memória principal têm a mesma chance de serem selecionadas, inclusive os frames que são frequentemente referenciados. Apesar de ser uma estratégia que consome poucos recursos do sistema, é raramente implementada, em função de sua baixa eficiência.

c. FIFO (First-in-first-out)

No *algoritmo FIFO*, a página que primeiro foi utilizada será a primeira a ser escolhida, ou seja, o algoritmo seleciona a página que está a mais tempo na memória principal. O algoritmo pode ser implementado associando-se a cada página o momento em que foi carregada para a memória ou utilizando-se uma estrutura de fila, onde as páginas antigas estão no início e as mais recentes no final.

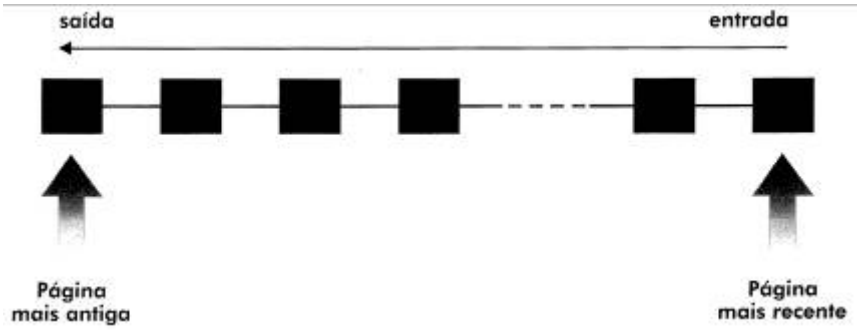


Figura 14: FIFO

Parece razoável pensar que uma página que esteja a mais tempo na memória seja justamente aquela que deve ser selecionada, porém isto nem sempre é verdade. No

caso de uma página que seja constantemente referenciada, como é o caso de páginas que contêm dados, o fator tempo torna-se irrelevante e o sistema tem que referenciar a mesma página diversas vezes ao longo do processamento.

O algoritmo FIFO é raramente implementado sem algum outro mecanismo que minimize o problema da seleção de páginas antigas que são constantemente referenciadas.

d. LFU (Least-frequently-used)

O *algoritmo LFU* seleciona a página menos referenciada, ou seja, o frame menos utilizado. Para isso, é mantido um contador com o número de referências para cada página na memória principal. A página que possuir o contador com o menor número de referências será a escolhida, ou seja, o algoritmo evita selecionar páginas que são bastante utilizadas.

Inicialmente, esta parece ser uma boa estratégia, porém as páginas que estão há pouco tempo na memória principal podem ser, justamente aquelas selecionadas, pois seus contadores estarão com o menor número de referências. É possível também que uma página muito utilizada no passado não seja mais referenciada no futuro. Nesse caso, como o contador possuiria um número elevado de referências, a página não seria selecionada para substituição. Este esquema, como apresentado, é raramente implementado, servindo apenas de base para outros algoritmos de substituição.

e. LRU (Least-recently-used)

O *algoritmo LRU* seleciona a página na memória principal que está há mais tempo sem ser referenciada. Se considerarmos o princípio da localidade, uma página que não foi utilizada recentemente provavelmente não será referenciada novamente em um futuro próximo.

Para implementar esse algoritmo, é necessário que cada página tenha associado o momento do último acesso, que deve ser atualizado a cada referência a um frame. Quando for necessário substituir uma página, o sistema fará uma busca por um frame que esteja há mais tempo sem ser referenciado. Outra maneira de implementar o LRU seria através de uma lista encadeada, onde todas as páginas estariam ordenadas pelo momento da última referência. Neste caso, cada acesso à memória exigiria um acesso à lista.

Apesar de ser uma estratégia com uma eficiência comparável ao algoritmo ótimo, é pouco empregada na prática, devido ao seu elevado custo de implementação.

f. NRU (Not-recently-used)

O *algoritmo NRU* é bastante semelhante ao LRU, porém com menor sofisticação. Para a implementação deste algoritmo é necessário um bit adicional, conhecido como *bit de*

referência (BR). O bit indica se a página foi utilizada recentemente e está presente em cada entrada da tabela de páginas.

Quando uma página é carregada para a memória principal, o bit BR referência é alterado pelo hardware, indicando que a página foi referenciada (BR = 1). Periodicamente, o sistema altera o valor do bit de referência (BR = 0), e à medida que as páginas são utilizadas, o bit associado a cada frame retorna para 1. Desta forma, é possível distinguir quais frames foram recentemente referenciados. No momento da substituição de uma página, o sistema seleciona um dos frames que não tenha sido utilizado recentemente, ou seja, com o bit de referência igual a zero.

O algoritmo NRU torna-se mais eficiente se o bit de modificação for utilizado em conjunto com o bit de referência. Neste caso, é possível classificar as páginas em quatro categorias, conforme a tabela abaixo:

Categorias	***Bits avaliados***	***Resultado***
1	*BR = 0 e BM = 0*	*Página não referenciada e não modificada*
2	*BR = 0 e BM = 1*	*Página não referenciada e modificada*
3	*BR = 1 e BM = 0*	*Página referenciada e não modificada*
4	*BR = 1 e BM = 1*	*Página referenciada e modificada*

O algoritmo, inicialmente, seleciona as páginas que não foram utilizadas recentemente e não foram modificadas, evitando assim um page out. O próximo passo é substituir as páginas que não tenham sido referenciadas recentemente, porém modificadas. Neste caso, apesar de existir um acesso à memória secundária para a gravação da página modificada, seguindo o princípio da localidade, há pouca chance dessa página ser novamente referenciada.

g. FIFO com buffer de páginas

O *algoritmo FIFO com buffer de páginas* combina uma lista de páginas alocadas (LPA) com uma lista de páginas livres (LPL). A LPA organiza todas as páginas que estão sendo utilizadas na memória principal, podendo ser implementada como uma lista única para todos os processos ou uma lista individual para cada processo. Independente da política utilizada, a LPA organiza as páginas alocadas há mais tempo na memória no início da lista, enquanto as páginas mais recentes no seu final. Da mesma forma, a LPL organiza todos os frames livres da memória principal, sendo que as páginas livres há mais tempo estão no início e as mais recentes no final. Sempre que um processo necessita alocar uma nova página, o sistema utiliza a primeira página da LPL, colocando-a no final da LPA (figura 15a). Caso o processo tenha que liberar uma página,

o mecanismo de substituição seleciona o frame em uso há mais tempo na memória, isto é, o primeiro da LPA, colocando-o no final da LPL (figura 15b).

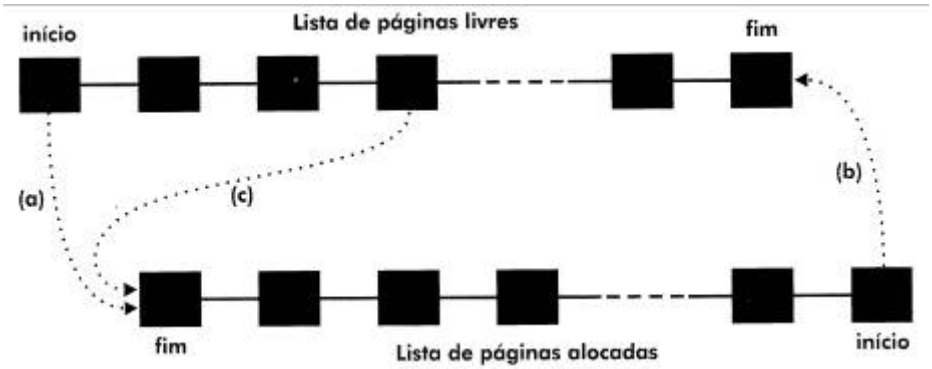


Figura 15: FIFO com buffer de páginas

É importante notar que a página selecionada e que entrou na LPL continua disponível na memória principal por um determinado intervalo de tempo. Caso esta página seja novamente referenciada e ainda não tenha sido alocada, basta retirá-la da LPL e devolvê-la ao processo (figura 15c). Nesse caso, a LPL funciona como um buffer de páginas, evitando o acesso à memória secundária. Por outro lado, se a página não for mais referenciada, com o passar do tempo irá chegar ao início da LPL, quando será utilizada para um outro processo. Caso a página seja posteriormente referenciada, o sistema terá que carregá-la novamente da memória secundária.

O buffer de páginas permite criar um algoritmo de substituição de páginas simples e eficiente, sem o custo de outras implementações. O sistema operacional Mach utiliza o esquema de buffer de páginas único, enquanto o OpenVMS e o Windows 2000 utilizam dois buffers: um para páginas livres (free page list) e outro para páginas modificadas (modified page list).

h. FIFO circular (clock)

O algoritmo FIFO circular utiliza como base o FIFO, porém as páginas alocadas na memória estão em uma estrutura de lista circular, semelhante a um relógio. Este algoritmo é implementado, com pequenas variações na maioria dos sistemas Unix.

Para a implementação do algoritmo existe um ponteiro que guarda a posição da página mais antiga na lista (figura 16a). Cada página possui associado um bit de referência, indicando se a página foi recentemente referenciada. Quando é necessário substituir uma página, o sistema verifica se o frame apontado tem o bit de referência desligado (BR = 0). Nesse caso, a página é selecionada para descarte, pois, além de ser a mais antiga, não foi utilizada recentemente. Por outro lado, se a página apontada tem o bit de referência ligado (BR = 1), o bit é desligado e o ponteiro incrementado, pois, apesar de

ser a página mais antiga, foi utilizada recentemente. O processo se repete até ser encontrada uma página com bit de referência igual a zero (figura 16b).

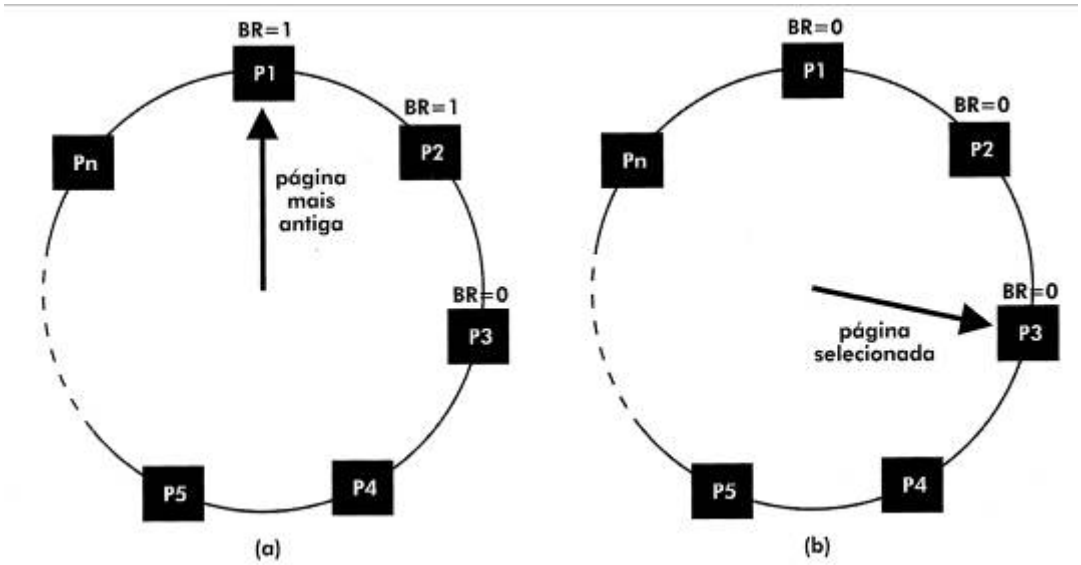


Figura 16: FIFO circular

Neste algoritmo, existe a possibilidade de todos os frames possuírem o bit de referência ligado. Nesse caso, o ponteiro percorrerá toda a lista, desligando o bit de referência de cada página. Ao final, a página mais antiga é selecionada. A utilização do bit de referência permite conceder a cada página uma segunda chance antes de ser substituída. É possível melhorar a eficiência do algoritmo utilizando o bit de modificação, juntamente com o bit de referência, como apresentado no esquema NRU.

4.6. Tamanho de Páginas

A definição do tamanho de página é um fator importante no projeto de sistemas que implementam memória virtual por paginação. O tamanho da página está associado à arquitetura do hardware e varia de acordo com o processador, mas normalmente está entre 512 e 16 M endereços. Algumas arquiteturas permitem a configuração do tamanho de página, oferecendo assim maior flexibilidade.

O tamanho da página tem impacto direto sobre o número de entradas na tabela de páginas e, conseqüentemente, no tamanho da tabela e no espaço ocupado na memória principal. Por exemplo, em uma arquitetura de 32 bits para endereçamento e páginas de 4k endereços, teríamos tabelas de páginas de até 220 entradas. Se cada entrada ocupasse 4 bytes, poderíamos ter tabelas de páginas de 4Mb por processo. Logo, páginas pequenas necessitam de tabelas de mapeamento maiores, provocam maior taxa de paginação e aumentam o número de acessos à memória secundária.

Apesar de páginas grandes tornarem menor o tamanho das tabelas de páginas, ocorre o problema da fragmentação interna. Como podemos observar na figura 17, o programa

ocupa quase que integralmente todas as páginas. A fragmentação só é encontrada, realmente, na última página, quando o código não ocupa o frame por completo.

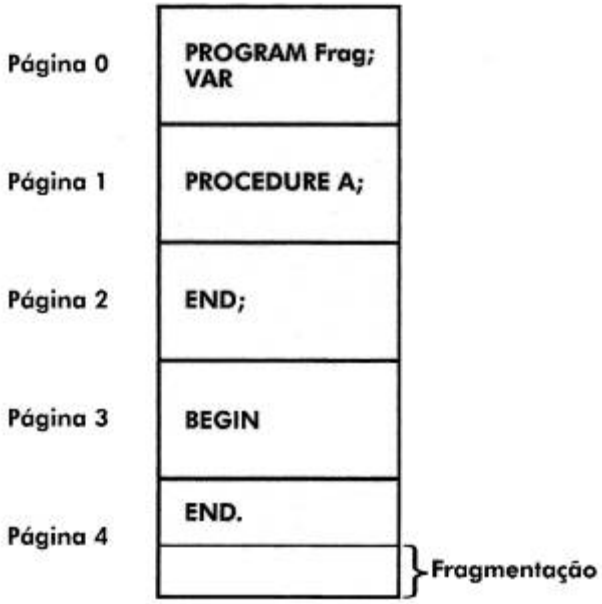


Figura 17: Fragmentação interna

O principal argumento a favor do uso de páginas pequenas é a melhor utilização da memória principal. A partir do princípio da localidade, com páginas pequenas teríamos na memória apenas as partes dos programas com maiores chances de serem executadas. Quanto maior o tamanho de página, maiores as chances de ter na memória código pouco referenciado, ocupando espaço desnecessariamente. Além disso, páginas pequenas reduzem o problema da fragmentação interna.

Outra preocupação quanto ao tamanho da página é a relacionada aos tempos de leitura e gravação na memória secundária. Devido ao modo de funcionamento dos discos, o tempo de operação de E/S com duas páginas de 512 bytes é muito maior do que uma página de 1024 bytes.

Com o aumento do espaço de endereçamento e da velocidade de acesso à memória principal, a tendência no projeto de sistemas operacionais com memória virtual por paginação é a adoção de páginas maiores, apesar dos problemas citados.

4.7. Paginação em Múltiplos Níveis

Em sistemas que implementam apenas um nível de paginação, o tamanho das tabelas de páginas pode ser um problema. Como já visto, em uma arquitetura de 32 bits para endereçamento e páginas com 4k endereços por processo, onde cada entrada na tabela de páginas ocupe 4 bytes, a tabela de páginas poderia ter mais de um milhão de entradas e ocuparia 4 Mb de espaço. Imaginando vários processos residentes na memória principal,

manter tabelas desse tamanho para cada processo certamente seria de difícil gerenciamento.

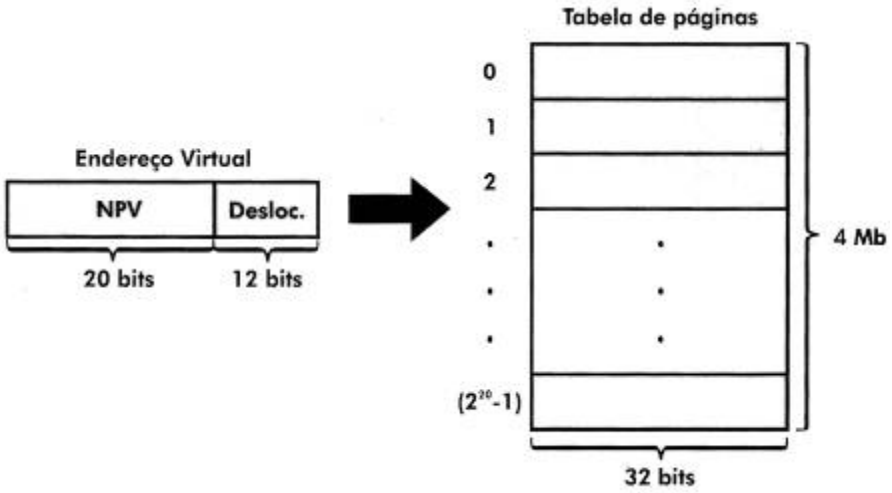


Figura 18: Paginação em um nível

Uma boa solução para contornar o problema apresentado é a utilização de tabelas de páginas em múltiplos níveis. A idéia é que o princípio da localidade seja aplicado também às tabelas de mapeamento – apenas as informações sobre páginas realmente necessárias aos processos estariam residentes na memória principal.

No esquema de paginação em dois níveis existe uma tabela diretório, onde cada entrada aponta para uma tabela de página. A partir do exemplo anterior, podemos dividir o campo NPV em duas partes: número da página virtual de nível 1 (NPV1) e número da página virtual de nível 2 (NPV2), cada um com 10 bits. O NPV1 permite localizar a tabela de páginas na tabela diretório; por sua vez, o NPV2 permite localizar o frame desejado na tabela de páginas.

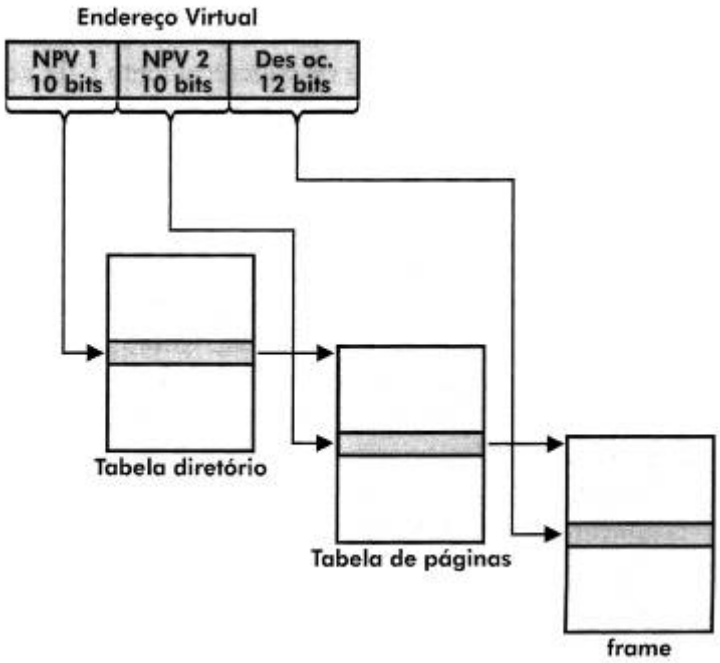


Figura 19: Endereço virtual em dois níveis

Utilizando-se o exemplo anterior, é possível que existam 1024 tabelas de páginas para cada processo. A grande vantagem da paginação em múltiplos níveis é que apenas estarão residentes na memória principal as tabelas realmente necessárias aos processos, reduzindo, dessa forma, o espaço ocupado na memória. A arquitetura VAX da Compaq/Digital é um exemplo de implementação de paginação em dois níveis.

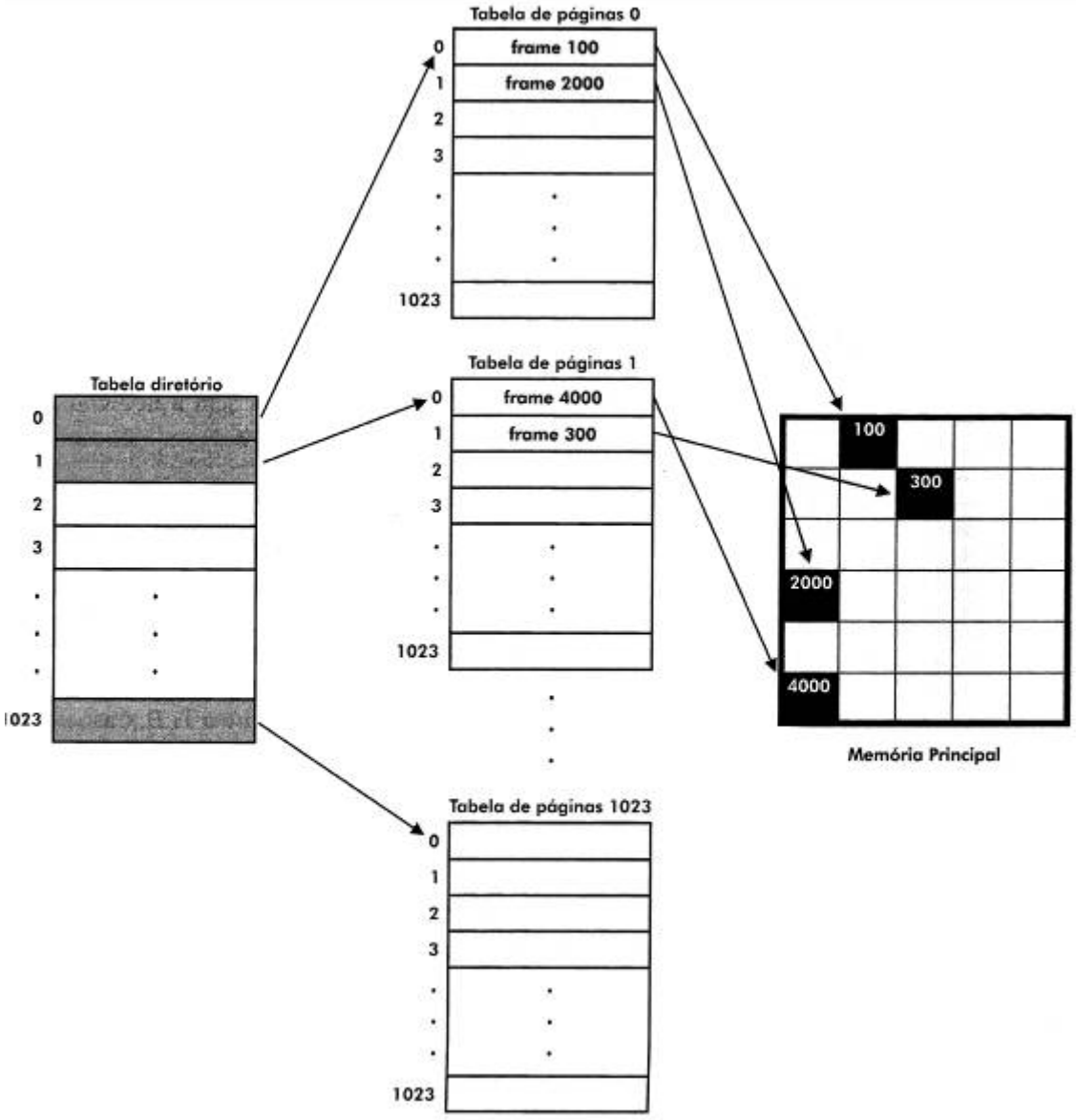


Figura 20: Paginação em dois níveis

Em uma arquitetura de 64 bits, a estrutura em dois níveis já não é mais adequada devido ao espaço de endereçamento. Novamente a solução passa por dividir a tabela diretório, criando uma estrutura em três níveis. Por exemplo, NPV1 com 32 bits, NPV2 com 10 bits e NPV3 com 10 bits. Esse tipo de implementação pode ser encontrado em alguns processadores da família SPARC, da Sun Microsystems.

A técnica de paginação em múltiplos níveis pode ser estendida para quatro níveis, como nos processadores Motorola 68030, cinco ou mais níveis. A cada nível introduzido há, pelo menos, mais um acesso à memória principal, o que gera problemas de desempenho. Tais problemas podem ser solucionados utilizando-se cachês.

4.8. Translation Lookaside Buffer

A gerência de memória virtual utiliza a técnica de mapeamento para traduzir endereços virtuais em endereços reais, porém, o mapeamento implica pelo menos dois acessos à memória principal: o primeiro à tabela de páginas e o outro à própria página. Sempre que um endereço virtual precisa ser traduzido, a tabela de mapeamento deve ser consultada para se obter o endereço do frame e, posteriormente, acessar o dado na memória principal.

Como a maioria das aplicações referencia um número reduzido de frames na memória principal, seguindo o princípio da localidade, somente uma pequena fração da tabela de mapeamento é realmente necessária. Com base neste princípio, foi introduzida uma memória especial chamada translation lookaside buffer (TLB), com o intuito de mapear endereços virtuais em endereços físicos sem a necessidade do acesso à tabela de páginas.

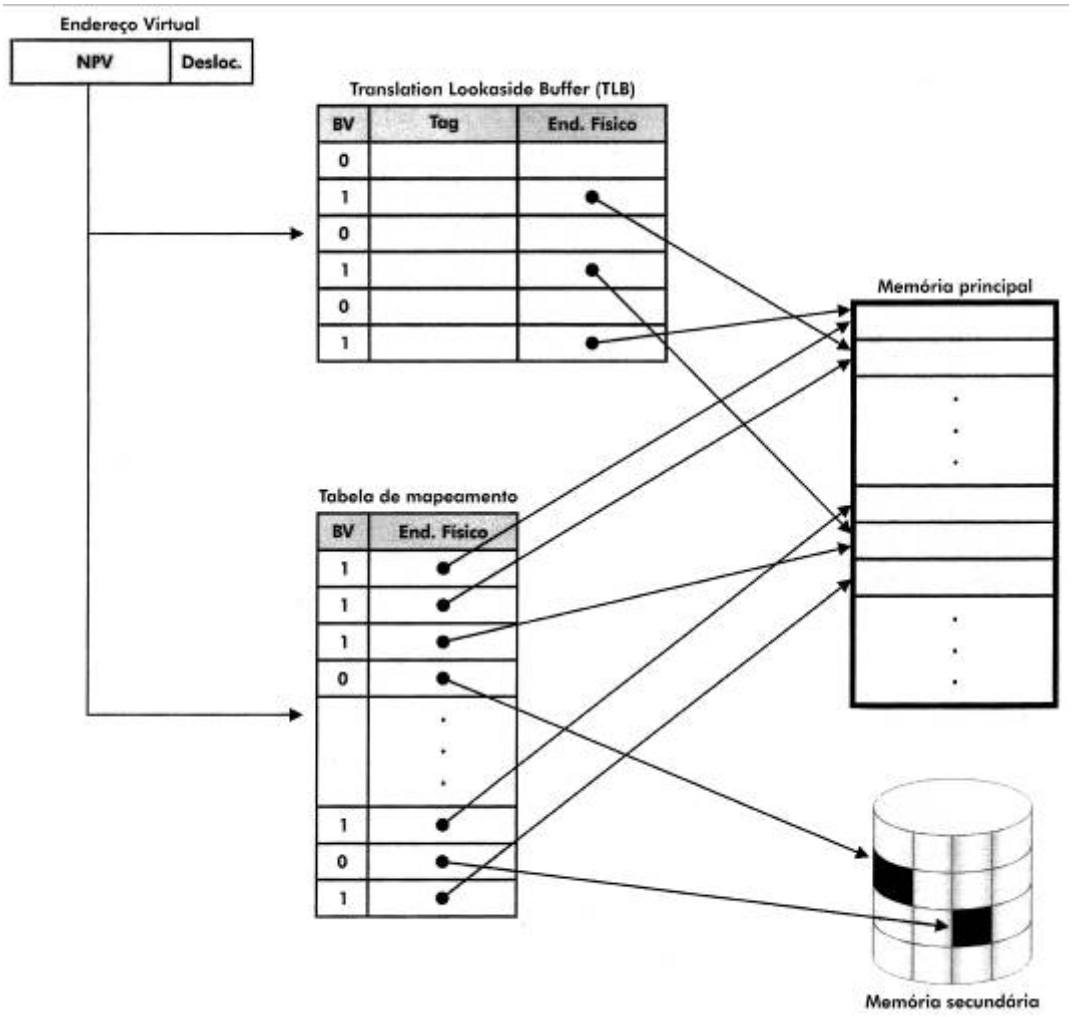


Figura 21: Translation lookaside buffer (TLB)

O TLB funciona como uma memória cache, mantendo apenas as traduções dos endereços virtuais das páginas mais recentemente referenciadas. Em geral, o TLB utiliza o esquema de mapeamento associativo, que permite verificar simultaneamente em todas as

suas entradas a presença do endereço virtual. Dessa forma, para localizar uma entrada, não é necessário realizar uma pesquisa em todo o TLB. Além disso, as traduções dos endereços virtuais podem ser armazenadas em qualquer posição da cache.

Na tradução de um endereço virtual, o sistema verifica primeiro o TLB. Caso o endereço virtual (tag) esteja na cache, o endereço físico é utilizado, eliminando o acesso à tabela de mapeamento (TLB hit). Caso o endereço não esteja na cache, a tabela de mapeamento deve ser consultada (TLB miss). Se a página estiver na memória principal, a tradução do endereço virtual é colocada no TLB e o endereço é traduzido. Caso contrário, ocorre um page fault, a página é carregada para a memória, a tabela de mapeamento é atualizada e a informação é carregada para a TLB.

Como a TLB pode eliminar o acesso à tabela de mapeamento, as informações de um endereço virtual, contidas na entrada da tabela de páginas, devem também estar na cache. A tabela a seguir descreve os campos de uma entrada típica de uma TLB.

Campo	***Descrição***
Tag	*Endereço virtual sem o deslocamento*
Modificação	*Bit que indica se a página foi alterada*
Referência	*Bit que indica se a página foi recentemente referenciada, sendo utilizada para a realocação de entradas na TLB*
Proteção	*Define a permissão de acesso à página*
Endereço físico	*Posição do frame na memória principal*

Geralmente, a TLB contém apenas informações sobre endereços do processo em execução, ou seja, quando há uma mudança de contexto, a TLB deve ser reinicializada com informações da tabela de mapeamento do novo processo. Caso a TLB possa armazenar endereços de vários processos, deve existir um campo adicional em cada entrada da TLB para a identificação do processo.

Sempre que a TLB fica com todas as suas entradas ocupadas e um novo endereço precisa ser armazenado, uma entrada da cache deve ser liberada. Geralmente, a política de realocação de entradas da TLB é aleatória e, mais raramente, utiliza-se a política NRU.

A TLB é essencial para reduzir o número de operações de acesso à memória principal em sistemas que implementam memória virtual. Devido ao conceito de localidade, a TLB pode ser implementada com poucas entradas, mapeando de 8 a 2048 endereços. Mesmo pequena, a taxa de TLB hits é muito alta, reduzindo significativamente o impacto da gerência de memória virtual no desempenho do sistema.

4.9. Proteção de Memória

Em qualquer sistema multiprogramável, onde diversas aplicações compartilham a memória principal, devem existir mecanismos para preservar as áreas de memória do sistema operacional e dos diversos processos dos usuários. O sistema operacional deve impedir que um processo tenha acesso ou modifique uma página do sistema sem autorização. Caso uma página do sistema operacional seja indevidamente alterada, é possível que, como conseqüência, haja uma instabilidade no funcionamento do sistema ou sua parada completa. Até mesmo páginas dos processos de usuários necessitam serem protegidas contra alteração, como no caso de frames contendo código executável.

Um primeiro nível de proteção é inerente ao próprio mecanismo de memória virtual por paginação. Neste esquema, cada processo tem a sua própria tabela de mapeamento e a tradução dos endereços é realizada pelo sistema. Desta forma, não é possível a um processo acessar áreas de memória de outros processos, a menos que haja compartilhamento explícito de páginas entre processos. A proteção de acesso é realizada individualmente em cada página da memória principal, utilizando-se as entradas das tabelas de mapeamento, onde alguns bits especificam os acessos permitidos.

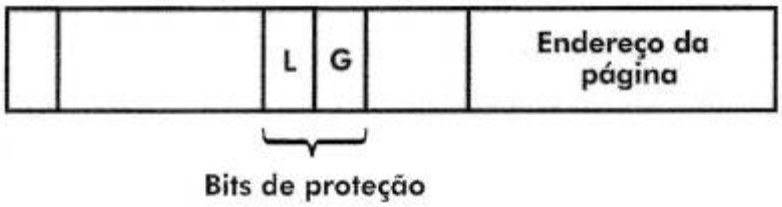


Figura 22: Proteção para páginas

De forma resumida, definiremos dois tipos de acessos básicos realizados em uma página: leitura e gravação. O acesso de leitura permite a leitura da página, enquanto o de gravação a sua alteração. Os dois tipos de acessos combinados produzem um mecanismo de proteção simples e eficiente, permitindo desde o total acesso à página, passando por acessos intermediários, até a falta completa de acesso.

LG	Descrição
00	Sem acesso
10	Acesso de leitura
11	Acesso para leitura/gravação

Figura 23: Mecanismo de proteção

Sempre que uma página é referenciada, o sistema operacional verifica na tabela de mapeamento do processo a proteção do frame e determina se a operação é permitida. Caso a tentativa de uma operação de gravação seja realizada em uma página com proteção apenas de leitura, o sistema gera um erro indicando a ocorrência do problema.

4.10. Compartilhamento de Memória

Em sistemas que implementam memória virtual, é bastante simples a implementação da reentrância, possibilitando compartilhamento de código entre os diversos processos. Para isso, basta que as entradas das tabelas de mapeamento dos processos apontem para os mesmos frames na memória principal, evitando, assim, várias cópias de um mesmo programa na memória. Apesar de os processos compartilharem as mesmas páginas de código, cada um possui sua própria área de dados em páginas independentes.

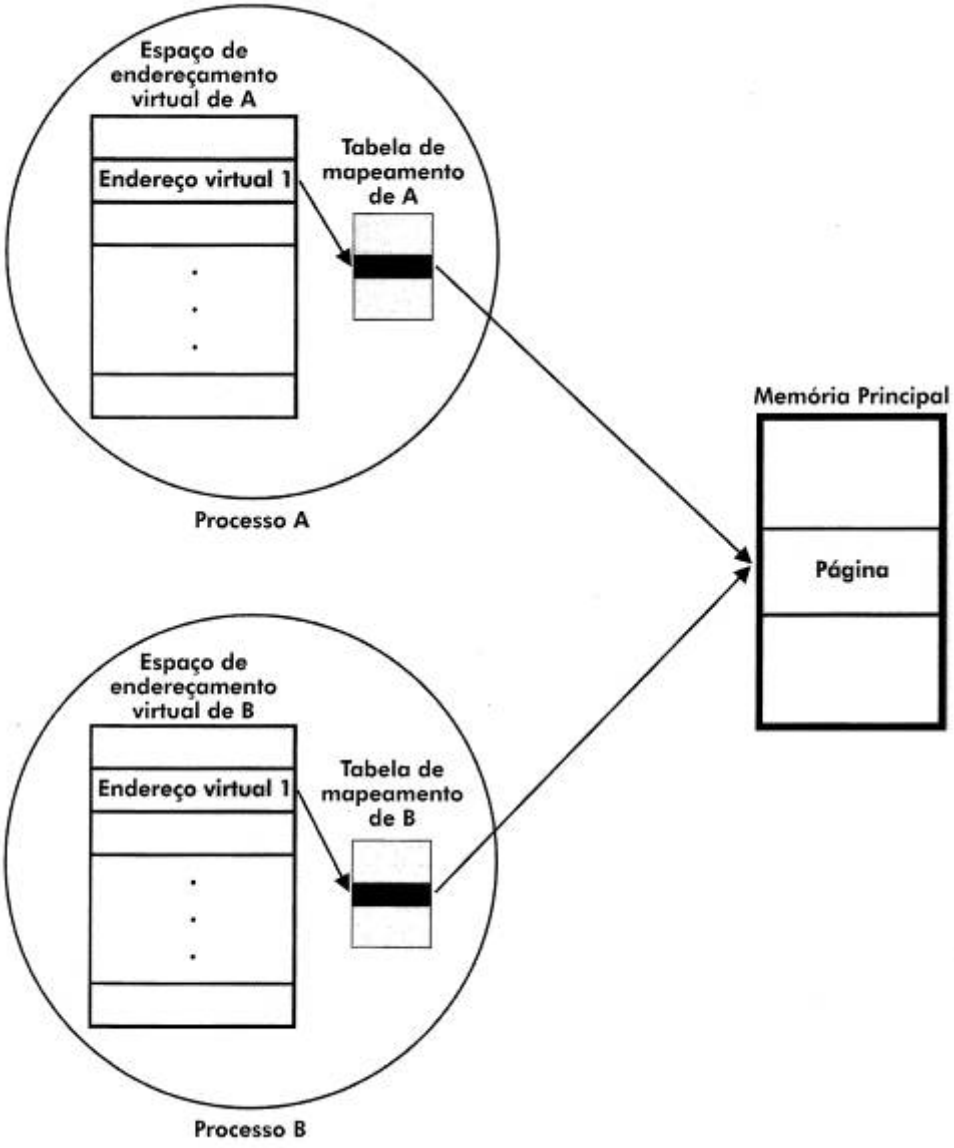


Figura 24: Compartilhamento de memória

O compartilhamento de memória também é extremamente importante em aplicações que precisam compartilhar dados na memória principal. Similar ao compartilhamento de código, o mecanismo de paginação permite que processos façam o mapeamento de uma mesma área na memória e, conseqüentemente, tenham acesso compartilhado de leitura e gravação. A única preocupação da aplicação é garantir o sincronismo no acesso à região compartilhada, evitando problemas de inconsistência.

5. MEMÓRIA VIRTUAL POR SEGMENTAÇÃO

Memória virtual por segmentação é a técnica de gerência de memória onde o espaço de endereçamento virtual é dividido em blocos de tamanhos diferentes chamados segmentos. Na técnica de segmentação, um programa é dividido logicamente em sub-rotinas e estruturas de dados, que são alocadas em segmentos na memória principal.

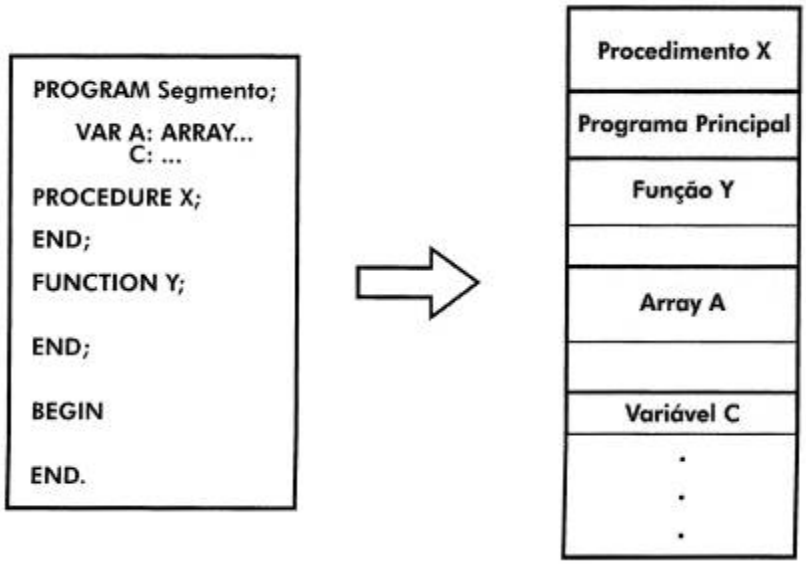


Figura 25: Segmentação

Enquanto na técnica de paginação o programa é dividido em páginas de tamanho fixo, sem qualquer ligação com sua estrutura, na segmentação existe uma relação entre a lógica do programa e sua alocação na memória principal. Normalmente, a definição dos segmentos é realizada pelo compilador, a partir do código fonte do programa, e cada segmento pode representar um procedimento, função, vetor ou pilha.

O espaço de endereçamento virtual de um processo possui um número máximo de segmentos que podem existir, onde cada segmento pode variar de tamanho dentro de um limite. O tamanho do segmento pode ser alterado durante a execução do programa, facilitando a implementação de estruturas de dados dinâmicas. Espaços de endereçamento independentes permitem que uma sub-rotina seja alterada sem a necessidade do programa

principal e todas as suas sub-rotinas serem recompiladas e religadas. Em sistemas que implementam paginação, a alteração de uma sub-rotina do programa implica recompilar e religar a aplicação por completo.

O mecanismo de mapeamento é muito semelhante ao da paginação. Os segmentos são mapeados através de tabelas de mapeamento de segmentos (TMS), e os endereços são compostos pelo número do segmento virtual (NSV) e por um deslocamento. O NSV identifica unicamente o segmento virtual que contém o endereço, funcionando como um índice na TMS. O deslocamento indica a posição do endereço virtual em relação ao início do segmento no qual se encontra. O endereço físico é obtido, então, combinando-se o endereço do segmento, localizado na TMS, com o deslocamento, contido no endereço virtual.

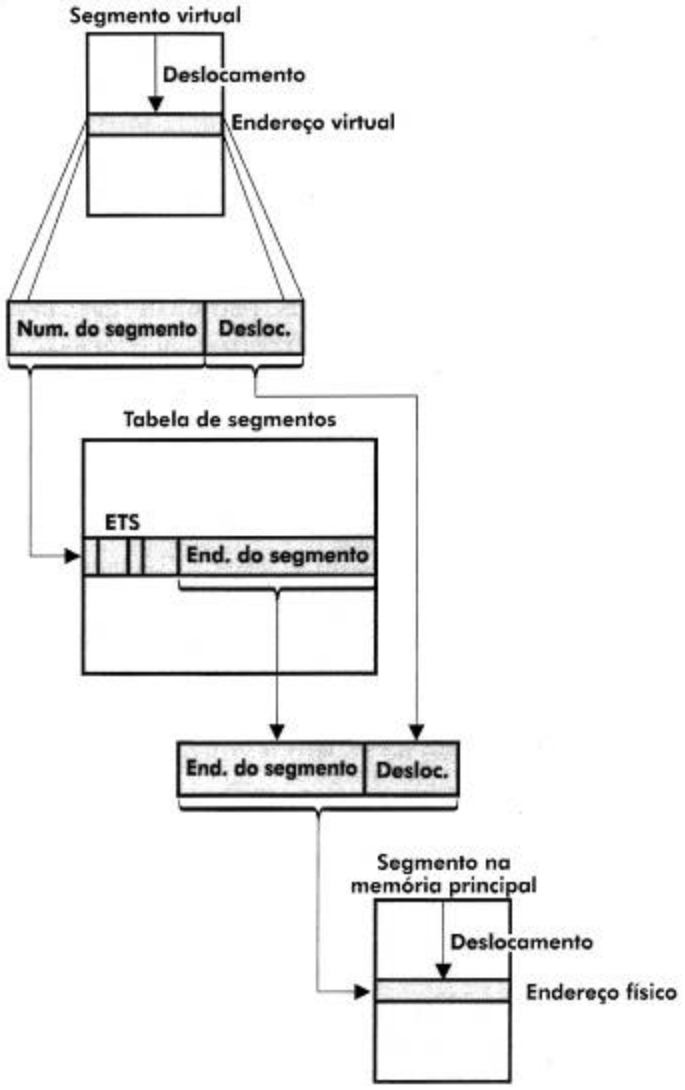


Figura 26: Tradução do endereço virtual

Cada ETS possui, além do endereço do segmento na memória principal, informações adicionais, conforme a tabela abaixo:

Campo	*Descrição*
Tamanho	*Especifica o tamanho do segmento*
Bit de validade	*Indica se o segmento está na memória principal*
Bit de modificação	*Indica se o segmento foi alterado*
Bit de referência	*Indica se o segmento foi recentemente referenciado, sendo utilizado pelo algoritmo de substituição*
Proteção	*Indica a proteção do segmento*

Uma grande vantagem da segmentação em relação à paginação é a sua facilidade em lidar com estruturas de dados dinâmicas. Como o tamanho do segmento pode ser facilmente alterado na ETS, estruturas de dados, como pilhas e listas encadeadas, podem aumentar e diminuir dinamicamente, oferecendo grande flexibilidade ao desenvolvedor. Enquanto na paginação a expansão de um vetor implica a alocação de novas páginas e, conseqüentemente, o ajuste da tabela de paginação, na segmentação deve ser alterado apenas o tamanho do segmento.

Na técnica de segmentação, apenas os segmentos referenciados são transferidos da memória secundária para a memória principal. Se as aplicações não forem desenvolvidas em módulos, grandes segmentos estarão na memória desnecessariamente, reduzindo o compartilhamento da memória e o grau de multiprogramação. Logo, para que a segmentação funcione de forma eficiente, os programas devem estar bem modularizados.

Para alocar os segmentos na memória principal, o sistema operacional mantém uma tabela com as áreas livres e ocupadas da memória. Quando um novo segmento é referenciado, o sistema seleciona um espaço livre suficiente para que o segmento seja carregado na memória. A política de alocação de páginas pode ser a mesma utilizada na alocação particionada dinâmica (best-fit, worst-fit ou first-fit) apresentada no capítulo Gerência de Memória.

Enquanto na paginação existe o problema da fragmentação interna, na segmentação surge o problema da fragmentação externa. Este problema ocorre sempre que há diversas áreas na memória principal, mas nenhuma é grande o suficiente para alocar um novo segmento. Neste caso, é necessário que os segmentos sejam realocados na memória de forma que os espaços livres sejam agrupados em uma única área maior.

Em sistemas com segmentação, a proteção de memória é mais simples de ser implementada do que em sistemas com paginação. Como cada segmento possui um conteúdo bem definido, ou seja, instruções ou dados, basta especificar a proteção do segmento na ETS, onde alguns bits podem especificar os tipos de acesso ao segmento.

Na segmentação é mais simples o compartilhamento de memória do que na paginação, pois a tabela de segmentos mapeia estruturas lógicas e não páginas. Para compartilhar um segmento, basta que as ETS dos diversos processos apontem para o mesmo segmento na memória principal. Por exemplo, enquanto o mapeamento de um vetor pode necessitar de várias entradas na tabela de páginas, na tabela de segmentos é necessária apenas uma única entrada.

A tabela abaixo compara as técnicas de paginação e segmentação em função de suas principais características:

Característica	*Paginação*	*Segmentação*
Tamanho dos blocos de memória	*Iguais*	*Diferentes*
Proteção	*Complexa*	*Mais simples*
Compartilhamento	*Complexa*	*Mais simples*
Estruturas de dados dinâmicas	*Complexa*	*Mais simples*
Fragmentação interna	*Pode existir*	*Não existe*
Fragmentação externa	*Não existe*	*Pode existir*
Programação modular	*Dispensável*	*Indispensável*
Alteração do programa	*Mais trabalhosa*	*Mais simples*

6. MEMÓRIA VIRTUAL POR SEGMENTAÇÃO COM PAGINAÇÃO

Memória virtual por segmentação com paginação é a técnica de gerência de memória onde o espaço de endereçamento é dividido em segmentos e, por sua vez, cada segmento dividido em páginas. Esse esquema de gerência de memória tem o objetivo de oferecer as vantagens tanto da técnica de paginação quanto da segmentação.

Nessa técnica, um endereço virtual é formado pelo número do segmento virtual (NSV), um número de página virtual (NPV) e um deslocamento. Através do NSV, obtém-se uma entrada na tabela de segmentos, que contém informações da tabela de páginas do segmento. O NPV identifica unicamente a página virtual que contém o endereço, funcionando como um índice na tabela de páginas. O deslocamento indica a posição do endereço virtual em relação ao início da página na qual se encontra. O endereço físico é obtido, então, combinando-se o endereço do frame, localizado na tabela de páginas, com o deslocamento, contido no endereço virtual.

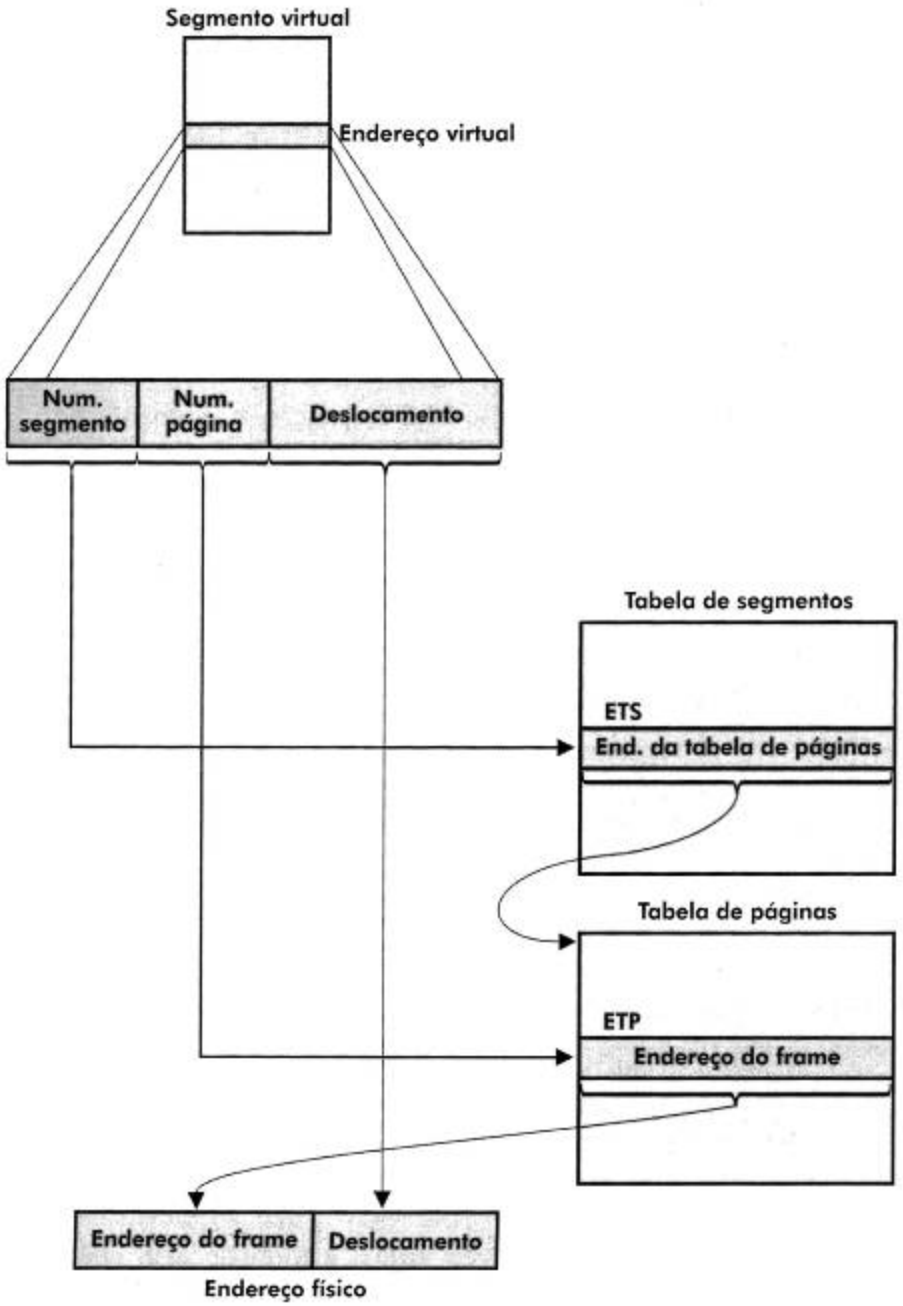


Figura 27: Segmentação com paginação

Na visão do programador, sua aplicação continua sendo mapeada em segmentos de tamanhos diferentes, em função das sub-rotinas e estruturas de dados definidas no programa. Por outro lado, o sistema trata cada segmento como um conjunto de páginas de mesmo tamanho, mapeadas por uma tabela de páginas associada ao segmento. Dessa forma, um segmento não precisa estar contíguo na memória principal, eliminando o problema da fragmentação externa encontrado na segmentação pura.

7. SWAPPING EM MEMÓRIA VIRTUAL

A técnica de swapping também pode ser aplicada em sistemas com memória virtual, permitindo aumentar o número de processos que compartilham a memória principal e, conseqüentemente, o grau de multiprogramação do sistema.

Quando existem novos processos para serem executados e não há memória principal livre suficiente para alocação, o sistema utiliza o swapping, selecionando um ou mais processos para saírem da memória e oferecer espaço para novos processos. Depois de escolhidos, o sistema retira os processos da memória principal para a memória secundária (swap out), onde as páginas ou segmentos são gravados em um arquivo de swap (swap file). Com os processos salvos na memória secundária, os frames ou segmentos alocados são liberados para novos processos. Posteriormente, os processos que foram retirados da memória devem retornar para a memória principal (swap in) para serem novamente executados.

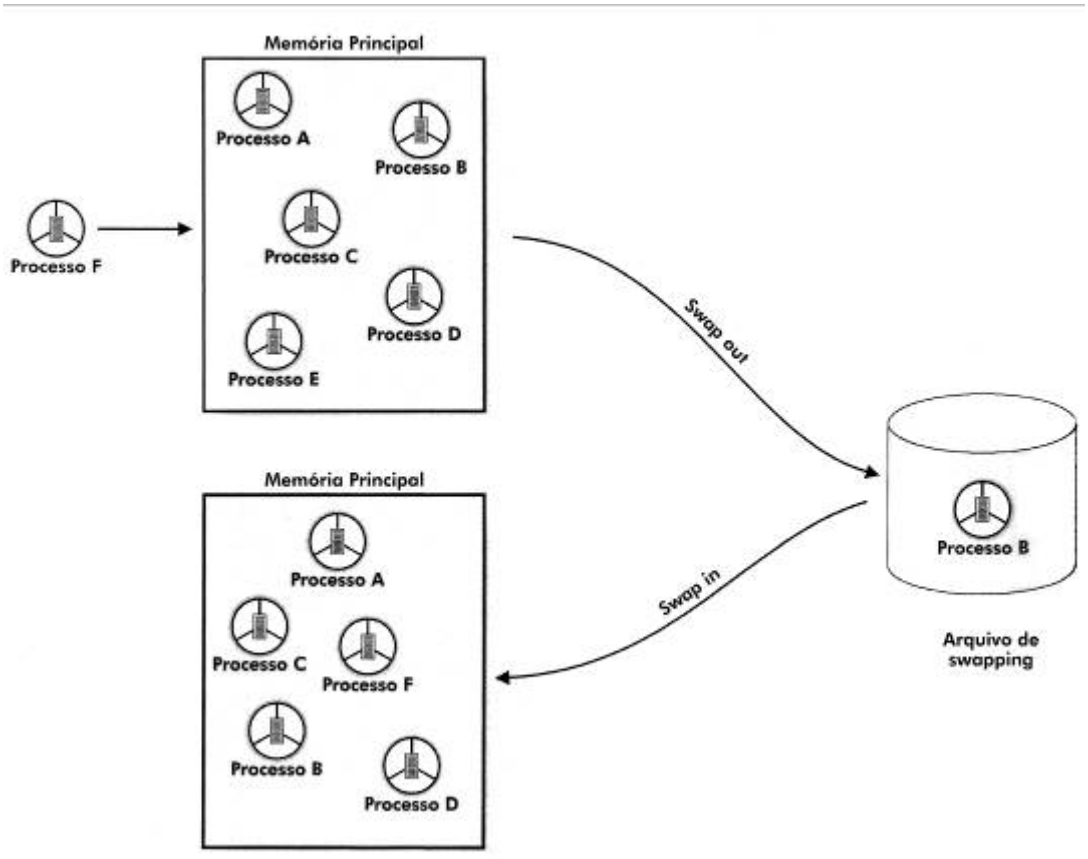


Figura 28: Swapping em memória virtual

Há várias políticas que podem ser aplicadas na escolha dos processos que devem ser retirados da memória principal. Independente do algoritmo utilizado, o sistema tenta selecionar os processos com as menores chances de serem executados em um futuro próximo. Na maioria das políticas, o critério de escolha considera o estado do processo e sua prioridade.

O swapping com base no estado dos processos seleciona, inicialmente, os processos que estão no estado de espera. A seleção pode ser refinada em função do tipo de espera de cada processo. É possível que não existam processos suficientes no estado de espera para atender as necessidades de memória do sistema. Nesse caso, os processos no estado de pronto com menor prioridade deverão ser selecionados.

O arquivo de swap é compartilhado por todos os processos que estão sendo executados no ambiente. Quando um processo é criado, o sistema reserva um espaço no arquivo de swap para o processo. Da mesma forma, quando um processo é eliminado o sistema libera a área alocada. Em alguns sistemas operacionais, o arquivo de swap é, na verdade, uma área em disco reservada exclusivamente para esta função. Independentemente da implementação, o arquivo de swap deve oferecer o melhor desempenho possível para as operações de swapping.

8. TRASHING

Trashing pode ser definido como sendo a excessiva transferência de páginas/segmentos entre a memória principal e a memória secundária. Esse problema está presente em sistemas que implementam tanto paginação como segmentação.

Na memória virtual por paginação, o trashing ocorre em dois níveis: no do próprio processo e no do sistema. No nível do processo, a excessiva paginação ocorre devido ao elevado número de page faults gerado pelo programa em execução. Esse problema faz com que o processo passe mais tempo esperando por páginas que realmente sendo executado. Existem dois motivos que levam um processo a sofrer esse tipo de trashing. O primeiro é o mau dimensionamento do limite máximo de páginas do processo, pequeno demais para acomodar o working set. O segundo é a ausência do princípio da localidade.

O trashing no sistema ocorre quando existem mais processos competindo por memória principal que espaço disponível. Nesse caso, o primeiro passo é a redução do número de páginas de cada processo na memória; porém, como já analisamos, esse mecanismo leva ao trashing do processo. Caso a redução não seja suficiente, o sistema inicia o swapping, retirando processos da memória principal para a memória secundária. Se esse mecanismo for levado ao extremo, o sistema passará mais tempo realizando swapping que atendendo aos processos.

O trashing em sistemas que implementam segmentação também ocorre em dois níveis. No nível do processo, a transferência excessiva de segmentos é devida à modularização extrema do programa. O trashing no sistema é semelhante ao da paginação, com a ocorrência de swapping de processos para liberar memória para os demais.

Independentemente das soluções apresentadas, se existirem mais processos para serem executados que memória real disponível, a única solução é a expansão da memória principal. É importante ressaltar que este problema não ocorre apenas em sistemas que implementam memória virtual, mas também em sistemas com outros mecanismos de gerência de memória.