Gerência de Memória

Hermes Senger, Hélio Crestana Guardia

Para que os processos sejam executados, é preciso que tanto seus códigos (suas sequências de instruções) quanto os dados que eles manipulam **sejam carregados na memória** principal.

Uma vez residentes na memória principal, programas podem fazer referências a endereços de trechos de código, como funções chamadas durante a execução, e também a variáveis, que correspondem a posições de memória nas áreas de dados.

Na Figura 1, por exemplo, vemos dois trechos de código que fazem referência a posições de memória. Ambos os trechos possuem efeitos semelhantes, embora o primeiro deles seja escrito em uma linguagem de alto nível, enquanto no item (b) vemos os mnemônicos de um trecho de programa escrito em uma pseudo-linguagem de máquina. A primeira instrução, **MOV R1,[1000]**, reside no endereço 100 da memória e seu efeito será mover o valor armazenado na posição de memória de número 1000 (valor da variável X) para um registrador chamado R1.

... **x = x +1;** **...**	**100 MOV R1, [1000]; carrega o valor de X em R1** **103 INC R1 ; incrementa o valor em R1** **104 MOV [1000], R1; armazena o resultado em X** **...**
(a)	**(b)**

Figura 1 – Dois trechos de código: (a) instrução em linguagem de alto nível que incrementa a variável X; (b) mnemônicos de uma pseudo-linguagem de máquina que fazem função semelhante.

A memória é gerenciada por meio de software, o Sistema Operacional, e manipulada por elementos específicos do hardware. Neste texto trataremos principalmente as questões relativas ao gerenciamento de memória realizado pelo SO.

Conforme estudamos, a evolução dos SOs passa por inovações como a multiprogramação, o compartilhamento do tempo de uso do processador, o suporte a múltiplos usuários simultaneamente e o atendimento de questões de tempo-real. Em todos esses casos, o gerenciamento do uso da memória tem grande influência sobre a quantidade de processos que podem ser executados ao mesmo tempo. Diferentes estratégias podem ser empregadas pelo SO para esse gerenciamento, afetando também o aproveitamento da memória, o tempo de execução dos processos e até o tempo de resposta das aplicações interativas.

As estratégias de gerência de memória realizadas por um SO estão relacionadas a decisões sobre a forma como a memória será ocupada pelos processos, sobre o local onde cada processo deve ser carregado, por quanto tempo um processo deve permanecer carregado na memória, etc.

Outro aspecto fundamental para a presença de vários processos na memória é a existência de mecanismos para impedir a interferência, proposital ou involuntária, entre suas áreas.

Fundamentos de Gerência de Memória

Antes de estudarmos as principais estratégias de gerência de memória, é preciso entender alguns aspectos fundamentais sobre o funcionamento dos sistemas de memória. A arquitetura dos computadores modernos geralmente apresenta uma hierarquia de memória com pelo menos três níveis:

- Memória **cache**: normalmente implementada com circuitos de memória estática, com baixíssimo tempo de acesso (muito rápida), localiza-se no próprio **processador.** Sua velocidade é extremamente alta, mas é muito cara! Por isso, está disponível em menor

quantidade.Os dados mais comumente usados são copiados para o cache para que possam ser acessados mais rapidamente, e uma pequena quantidade de cache normalmente é suficiente para melhorar o desempenho dos programas, devido a um fenômeno chamado de **localidade temporal**. O gerenciamento do uso dessa memória cache é tratado pelo próprio processador, que decide quais dados manter ali, em função de suas utilizações. Em computadores com processadores de vários núcleos e/ou vários processadores, é comum haver uma hierarquia de memórias cache. Caches de nível 1 estão presentes em cada núcleo. Caches de nível 2 são normalmente compartilhados entre cada par de núcleos, e caches de nível 3 são compartilhados entre todos os núcleos/processadores.

- Memória **principal**: é a memória RAM, também chamada de memória física, real ou primária. É normalmente implementada com circuitos de memória dinâmica (RAM dinâmica) e é utilizada para armazenar instruções dos programas (código) e dados em uso.

- Memória **secundária**: é comumente implementada por discos rígidos e, atualmente, até como unidades de armazenamento em memórias de estado sólido (SSDs, em notebooks). É utilizada para armazenamento permanente, não volátil. Além dos arquivos comuns no sistema de arquivos, unidades de memória secundária podem conter uma área conhecida como área de *swap*, onde são mantidos dados e programas que não precisam, ou que não podem, estar na memória principal, por limitações de espaço. Este tipo de memória é bem mais barato, considerando o preço por unidade de armazenamento. Por ser barata, está disponível em maior quantidade num computador.

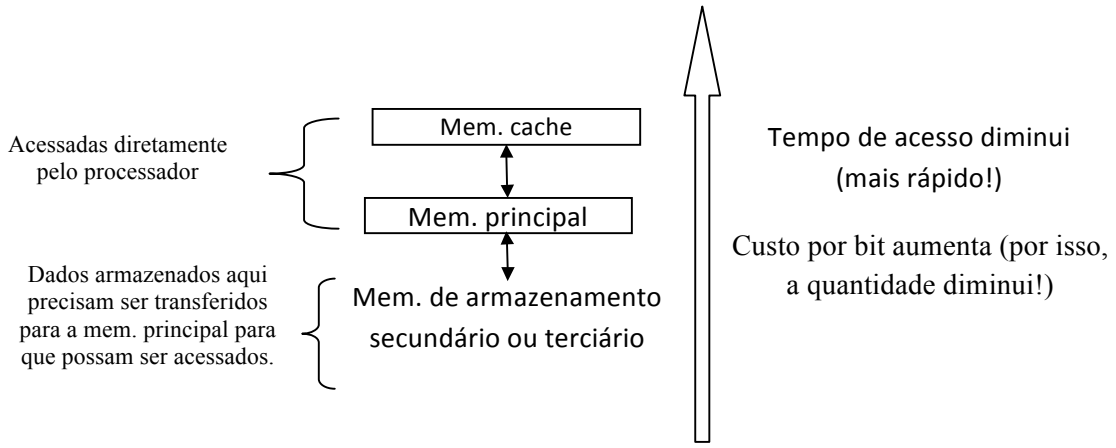


Figura 2 – Hierarquia de memória

Considerando o armazenamento de programas na memória principal, atualmente a maioria dos computadores e sistemas operacionais possuem mecanismos de gerenciamento de memória virtual. Pensando em sistemas computacionais dedicados e visando a entender as estratégias em uso pelos SOs atuais, contudo, apresenta-se a seguir uma descrição evolutiva do uso e do gerenciamento de memória.

Sistemas Monotarefa e Monousuário

Os primeiros sistemas operacionais eram monotarefa e monousuário e, portanto, carregavam na memória um único processo por vez. Isso podia ser feito de várias maneiras, tal como ilustrado na Figura 3. Na Figura 3(a) vemos o sistema operacional ocupando a parte alta da

memória, enquanto a parte baixa é ocupada pelo processo usuário. Já no item (b) vemos a parte baixa da memória RAM (*Random Access Memory*) sendo ocupada pelo sistema operacional, enquanto alguns *drivers* de dispositivos de E/S são armazenados em memória ROM (*Read Only Mempory*), que corresponde aos endereços mais altos da memória. O processo usuário fica armazenado na parte intermediária. Essa organização de memória era utilizada pelo SO MS-DOS, por exemplo.

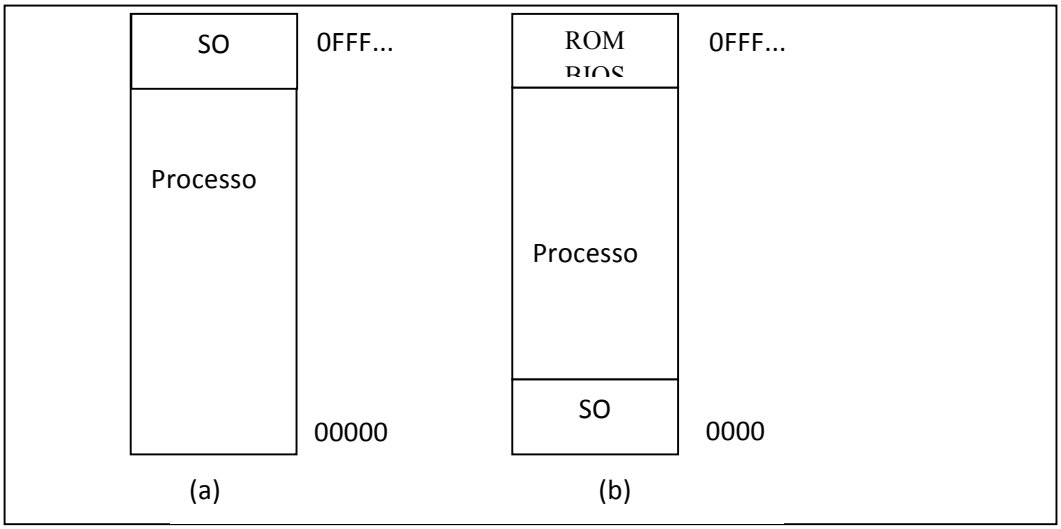


Figura 3 – Formas para carregamento de um único processo na memória.

Nesses casos, se um processo não puder ser carregado na memória por não haver espaço suficiente, ele simplesmente não será executado. Para amenizar o problema, foi desenvolvida uma técnica de sobreposição de partes denominada ***overlay***. Em geral, essa técnica era implementada por compiladores de linguagens de programação, que dividiam os programas em partes fixas, que permaneciam carregadas na memória durante toda a execução do processo, e outras partes (os *overlays*) que eram carregadas uma após a outra em uma mesma região da memória. Dividir o programa em *overlays* era uma tarefa do programador, que determinava ao compilador onde começava e onde terminava cada *overlay*.

Relocação de endereços

Nos primeiros sistemas, cada programa só podia ser carregado e executado a partir de um endereço de memória fixo, determinado no momento de sua compilação. Isso era ruim, pois implicava que o programa só podia ser carregado nesse endereço de memória. Uma importante técnica desenvolvida para tratar esse problema consistiu no uso de **registradores** especiais no hardware, denominados **registrador base** e **limite**. Assim, cada processo possuía um par de registradores base e limite, e o hardware cuida para que um processo só pudesse acessar endereços que estivessem dentro desse intervalo. Deste modo, provia-se proteção de memória para que diversos processos coexistissem na memória principal. Além disso, a técnica permitia que programas fossem carregados em qualquer endereço livre da memória, bastando ajustar o valor desses registradores.

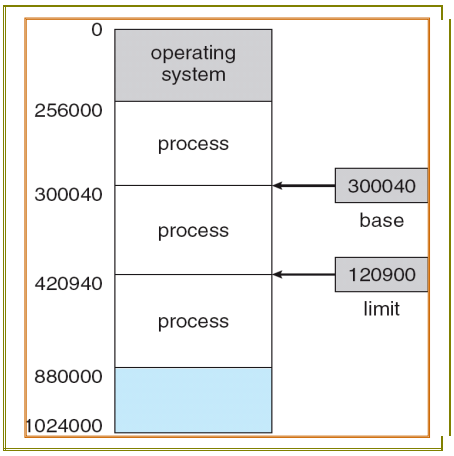


Figura 4 – Relocação de código com registradores de base e limite (Fonte:Siberschatz, 2008)

Para que um processo pudesse ser executado, tanto suas as instruções quanto os dados deveriam ser carregados em algum endereço de memória. Portanto, era preciso **atribuir** endereços de memória para todas as instruções e todos os dados de um processo. Essa **atribuição** (*address binding*) pode ser feita em três momentos distintos:

- Em tempo de compilação: quando o endereço de memória for conhecido previamente, o próprio compilador pode fazer a atribuição de endereços absolutos. O resultado é o processo somente poderá ser carregado a partir de um mesmo endereço fixo, e isso só poderá ser modificado se o programa for recompilado.

- Em tempo de carga: é útil nos casos em que não se conhece previamente o endereço onde o programa será carregado. Nesse caso, o compilador deve gerar um código relocável, que possa ser carregado a qualquer endereço que esteja disponível no momento em que ocorrer a carga na memória.

- Em tempo de execução: quando um processo pode ser deslocado de um segmento de memória para outro durante a sua execução, então a atribuição de endereço pode ser atrasada até o momento de sua execução. Essa técnica requer o uso de registradores de base e limite.

Sistemas com multiprogramação

A técnica de multiprogramação permite um melhor aproveitamento do uso dos recursos do sistema, principalmente do processador, alocando-o para outro processo quando o processo que executava precisa esperar por operações de entrada e saída de dados. Para tanto, contudo, a multiprogramação implica em manter múltiplos processos (inteiramente, ou parcialmente) carregados na memória durante o tempo em que estes estiverem sendo executados.

Multiprogramação com Partições Fixas

Uma das primeiras técnicas utilizadas por sistemas multiprogramáveis consistia em dividir a memória em um conjunto de partições fixas, tal como na Figura 5. Nesse esquema, cada partição possuía um endereço inicial e endereço final, e cada programa era compilado para ser carregado e executado exclusivamente em uma partição. Se uma partição estivesse ocupada no momento, por exemplo, a partição 1, outros processos designados para essa mesma partição deveriam aguardar na fila. Assim, havia uma fila para cada partição disponível.

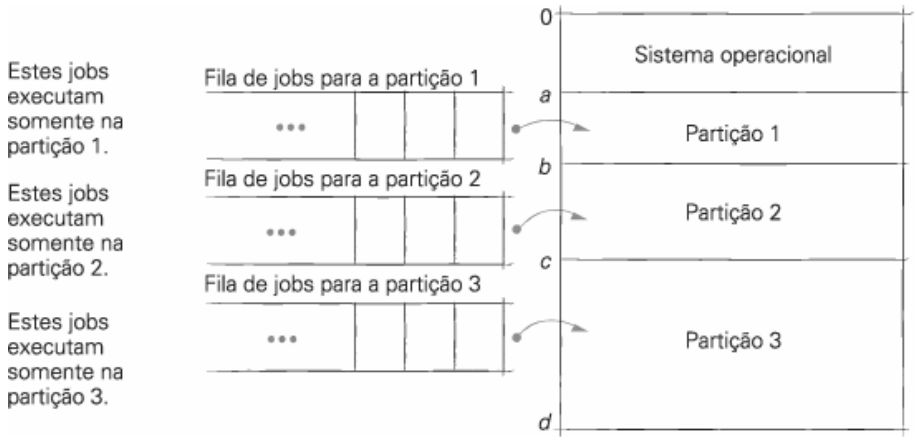


Figura 5 – Multiprogramação com partições fixas. (**Fonte: Deitel, 2005)**

Obviamente, esse esquema tinha limitações em termos da utilização da memória pois, se uma partição estivesse ocupada, alguns processos poderiam ficar muito tempo aguardando na fila, mesmo que outras partições estivessem livres. O sistema OS/360 da IBM foi um exemplo de sistema que implementava essa técnica de multiprogramação com partições fixas.

Parte do problema das partições fixas foi resolvido pelo uso de **partições fixas com código relocável**. Nesse esquema, o uso de código relocável permitiu que cada processo fosse carregado em qualquer partição livre que tivesse tamanho suficiente. O esquema é ilustrado na Figura 6.

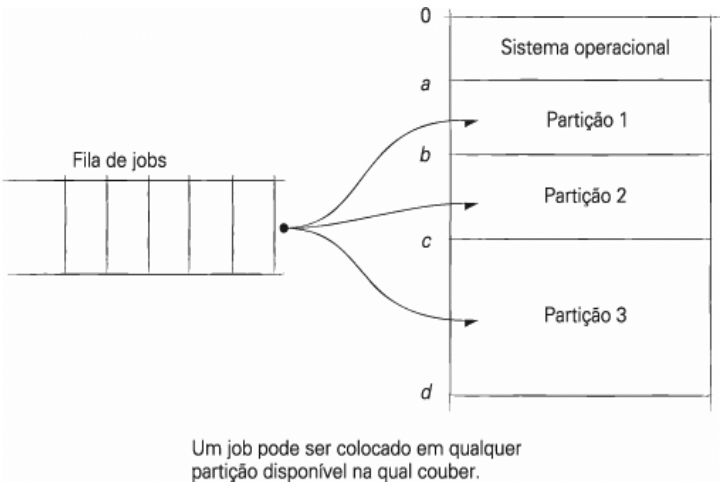


Figura 6 – Multiprogramação com partições fixas e código relocável. (Fonte: **Deitel, 2005)**

Apesar da melhoria, restaram alguns problemas. Os processos carregados nem sempre utilizavam a partição inteira, ocasionando um problema conhecido como **fragmentação interna**. Além disso, poderia ocorrer de nenhuma partição ser grande o bastante para um dado processo.

Multiprogramação com Partições Variáveis

O uso de partições fixas em geral foi suficiente para sistemas de processamento em *batch*, pois era simples e bastante efetivo. Entretanto, com a evolução e o surgimento dos sistemas de tempo compartilhado (*time-sharing*) isso mudou. O número de processos a serem mantidos na memória

aumentou, principalmente em função do aumento do número de processos interativos. Nesse caso, o desperdício de memória tornou-se um problema mais grave.

A estratégia de utilizar **partições variáveis**, então, entrou em cena. Segundo essa estratégia, no início não haviam partições demarcadas, mas sim uma única partição livre. À medida que os processos tivessem de ser carregados, o sistema operacional alocava novas partições com o tamanho exato requisitado pelo novo processo. A Figura 7 ilustra o funcionamento desse esquema.

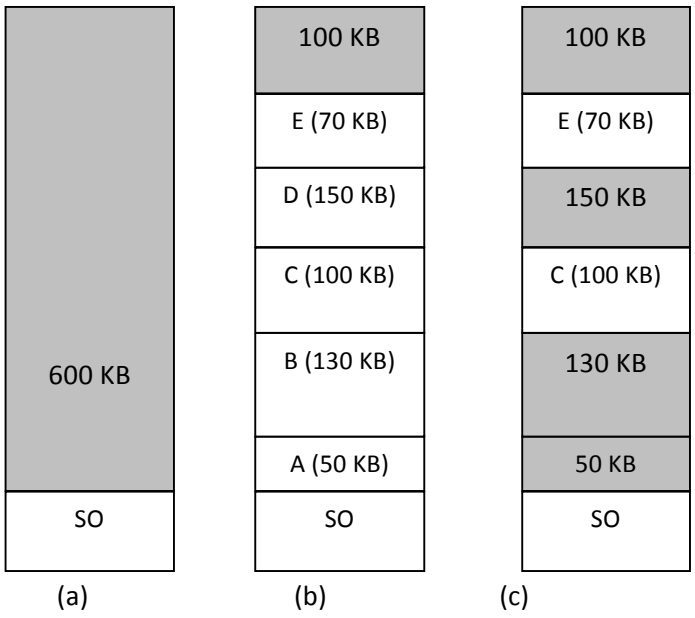


Figura 7 – Multiprogramação com partições variáveis.

Apesar da maior flexibilidade, esse esquema pode provocar o que é chamado de fragmentação externa, quando algum processo é removido. Fragmentação externa está relacionada com possíveis sobras de espaço dentro de uma partição, quando essa é maior do que o programa que ela contém. Por exemplo, na Figura 7-a) vemos uma situação inicial com toda a memória livre. Na Figura 7-b) vemos a memória após o início dos processos A, B, C, D e E. Suponha que os processos A, B e D terminem, tal como na Figura 7-c) e um novo processo F necessite de 200 KB para executar. Ele não terá uma partição livre (contígua) grande o suficiente para carregá-lo.

Existem algumas formas de combater a **fragmentação externa**. A primeira delas é chamada de **coalescência**, que consiste em reunir blocos livres que sejam vizinhos (para formar um único bloco grande). Por exemplo, na **Figura 8** a técnica de coalescência poderia agrupar as partições adjacentes de 50 KB e de 130 KB (onde estiveram os processos A e B), para formarem uma única partição livre de 180 KB.

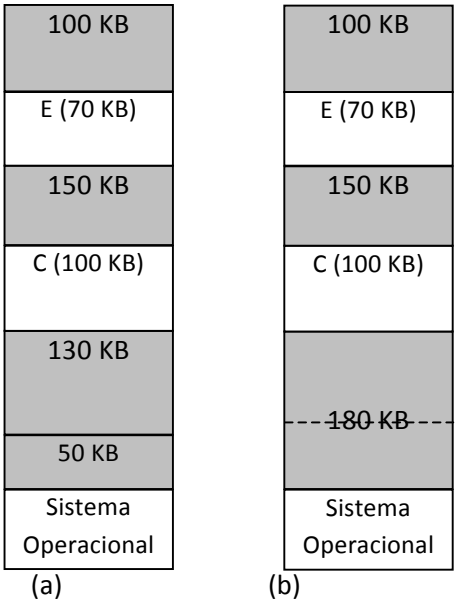


Figura 8 – Coalescência.

A coalescência reduz o problema da fragmentação externa, mas geralmente isso não é suficiente para obter uma quantidade significativa de memória. Outra técnica empregada foi a **compactação** das áreas livres, que consiste em reorganizar a memória de modo a reunir todos os blocos livres para formarem um único bloco contíguo livre. Com isso, é possível carregar novos processos que caibam nessa partição livre. Evidentemente, relocar processos que já estejam em execução traz uma sobrecarga significativa, pois o sistema fica lento durante essa reorganização.

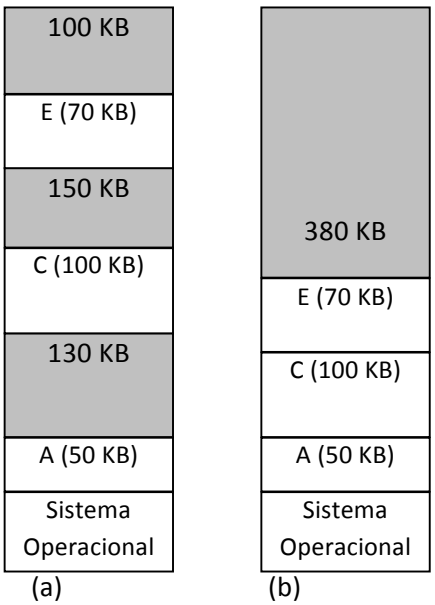


Figura 9 – Estratégia de compactação das áreas livres.

Para tentar evitar o problema da fragmentação das áreas livres, uma estratégia utilizada consistia em "escolher" uma partição livre para alocar o novo processo. Três estratégias mais comuns são comumente empregadas nesse caso:

- Estratégia do **primeiro que couber** (*first-fit*): nesse caso, o SO escolhe a a primeira lacuna de tamanho suficiente que for encontrada e aloca para o carregamento do processo. Essa estratégia apresenta uma sobrecarga baixa e elementar em tempo de execução.

- Estratégia o que **melhor couber** (*best-fit*): escolhe a lacuna que deixar o menor espaço livre ao seu redor, para lá posicionar o processo. A ideia nesse caso é tentar minimizar os tamanhos dos fragmentos resultantes. Essa estratégia causa alguma sobrecarga em tempo de execução.

- Estratégia o que **pior couber** (*worst-fit*): o processo é posicionado na lacuna que deixar o maior espaço não utilizado ao seu redor, procurando aumentar a chance desse espaço restante servir para o carregamento de outro processo.

Multiprogramação com Transição da Memória: *Swapping*

O uso de partições fixas em geral foi suficiente para sistemas de processamento em *batch*, pois era simples e bastante efetivo para tratar poucos processos simultâneos. Entretanto, com a evolução e o surgimento dos sistemas de tempo compartilhado (*time-sharing*) isso mudou, pois o número de processos a serem mantidos na memória aumentou.

Por outro lado, observou-se que alguns desses processos ficavam inativos, por exemplo, por estarem aguardando interação de usuário. Dentre os processos criados, também era comum haver processos em *batch* que foram temporariamente bloqueados pelo escalonador, ou cuja prioridade foi diminuída. Procurando maximizar o uso da memória principal, percebeu-se que um processo não precisaria permanecer na memória enquanto estivesse bloqueado, e portanto, poderia ser temporariamente enviado ao disco, e mais tarde ser recarregado para ter sua execução continuada. Assim, surgiu uma estratégia denominada ***swapping***, que envolve essa transição entre as áreas de memória de um processo entre a memória principal e um espaço reservado numa unidade de armazenamento secundário.

Com o uso dessa técnica, amplia-se o uso da memória principal disponível, que passa a ser utilizada por um conjunto de processos maior do que efetivamente pode ser ali armazenado. O funcionamento da estratégia é ilustrado na Figura 10.

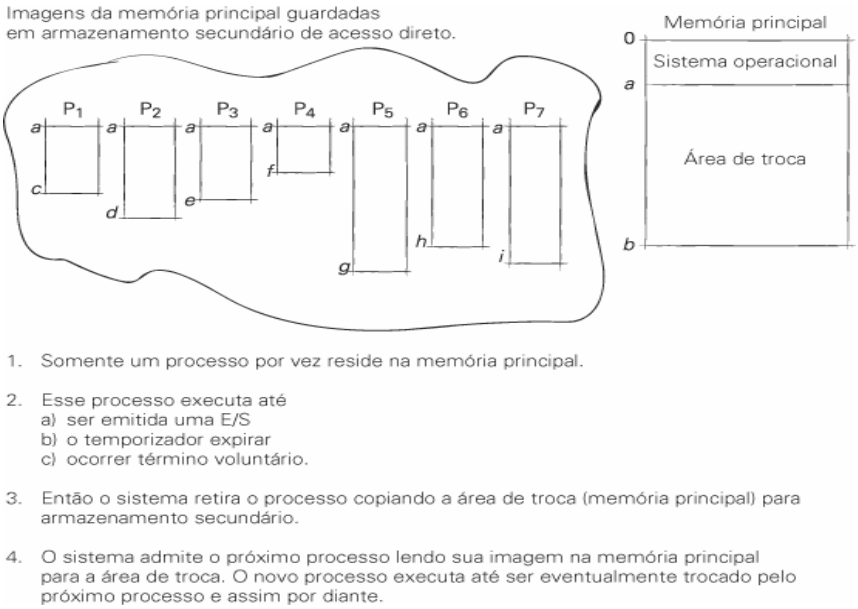


Figura 10 – *Swapping* (Fonte: **Deitel 2005)**

Com essa técnica, contudo, há uma sobrecarga significativa no momento de fazer o chaveamento de processos, pois é preciso enviar um processo inteiro da memória para o disco (*swap out*) e carregar outro processo (*swap in*) do disco para a memória. Atualmente, essa ideia é empregada nos sistemas operacionais com memória virtual, conforme veremos adiante, mas com a diferença de que não são trocados processos inteiros, mas apenas parte deles (páginas e/ou segmentos).

Sistemas com Memória Virtual

Memória virtual é o nome que se dá a uma abstração que faz com que as referências a posições de memória feitas pelos processos sejam relativas a um espaço lógico, e não diretamente à memória física. Um mecanismo de tradução é realizado pelo hardware em tempo de execução.

Com essa técnica, há várias vantagens. Primeiro, os programas podem ser posicionados em qualquer parte da memória. Além disso, os programas em execução e suas áreas de dados não precisam estar completamente na memória o tempo todo, permitindo que a memória seja usada por mais programas do que caberia de uma vez.

Antes de estudarmos o funcionamento da memória virtual, é preciso compreendermos alguns conceitos importantes. O primeiro conceito é diferenciarmos o que é **endereço real** de **endereço virtual**. Para exemplificar, considere a instrução:

MOV R1, [1000]

Na maioria das vezes, o endereço que o programador vê (como o endereço 1000 dessa instrução) é o endereço lógico, e não o endereço físico ou endereço real. O sistema (*hardware* + sistema operacional) faz o **mapeamento** ou **tradução de endereço** de endereço lógico para endereço físico, de forma transparente ao usuário. Assim, o **espaço de endereço virtual** V de um sistema é a gama de endereços virtuais que um processo pode referenciar, enquanto o **espaço de endereço real** R corresponde à gama de endereços físicos disponíveis em um sistema de

computador específico. Uma unidade de gerenciamento de memória (*memory management unit* - MMU) presente no processador possui um mecanismo de tradução dinâmica de endereço que converte endereços virtuais em endereços físicos durante a execução de um programa. Tudo isso de forma totalmente transparente ao usuário! Nesse processo, cabe ao SO manter uma tabela de mapeamento de endereços lógicos em físicos para cada processo. A tradução desses endereços, usando a tabela, contudo, é feita pelo processador.

Três estratégias de tradução de endereços virtuais em endereços físicos são comumente utilizadas, em função dos recursos presentes no hardware. São elas: paginação, segmentação e segmentação paginada.

Paginação

Vimos que a fragmentação das áreas livres de memória é um dos maiores problemas no gerenciamento de memória. Na verdade, esse problema é chamado de **fragmentação externa** de memória, pois um processo pode ter seu carregamento para a memória negado, mesmo havendo memória livre disponível, se não houver um espaço contíguo suficientemente grande para contê-lo. Uma solução para esse problema é **separar** o espaço de endereçamento **lógico** referenciado pelos processos, do espaço de endereçamento real (ou físico) em que os programas e dados efetivamente estão armazenados.

Quando a tradução desses endereços é feita usando blocos de tamanho fixo, tem-se a técnica denominada **paginação**. A paginação consiste em dividir o espaço de endereçamento de um processo em **blocos de tamanho fixo**, que são chamados de **páginas**. Dessa forma, todo endereço passa a ter 2 dimensões: uma indicação da **página** lógica, que deve ser mapeada para uma página física, e o **deslocamento** dentro dessa página. Dado o número de bits de um endereço no processador alvo (32, por exemplo), considera-se que os N bits menos significativos desse endereço estão associados ao deslocamento dentro de uma página, e os restantes indicam a qual página o endereço se refere. Na arquitetura x86 (PCs) as páginas tipicamente têm tamanho de 4096 (4k) bytes. Assim, os 12 bits menos significativos indicam o deslocamento, ou a posição de um endereço na página física que contém a página lógica referenciada.

A memória física é, então, subdividida em **quadros** (***frames***) do mesmo tamanho que as páginas lógicas (e.g. 4K). Assim, páginas do espaço de endereçamento virtual de um processo são **mapeadas** em quadros do espaço de memória física. Esse mapeamento normalmente é feito com o auxílio de uma **tabela de páginas**, que contém o endereço físico (número do *frame*) de cada página da memória de um processo. Por exemplo, um processo de 20 KB pode ter o seu espaço de endereçamento subdividido em 5 páginas de 4 KB cada, como ilustra a Figura 11. Repare que as páginas não são armazenadas de forma contígua na memória física, e para executar um novo programa de tamanho igual a P páginas seria preciso apenas encontrar P **quadros** livres (não contíguos) na memória física em qualquer posição onde houver memória livre.

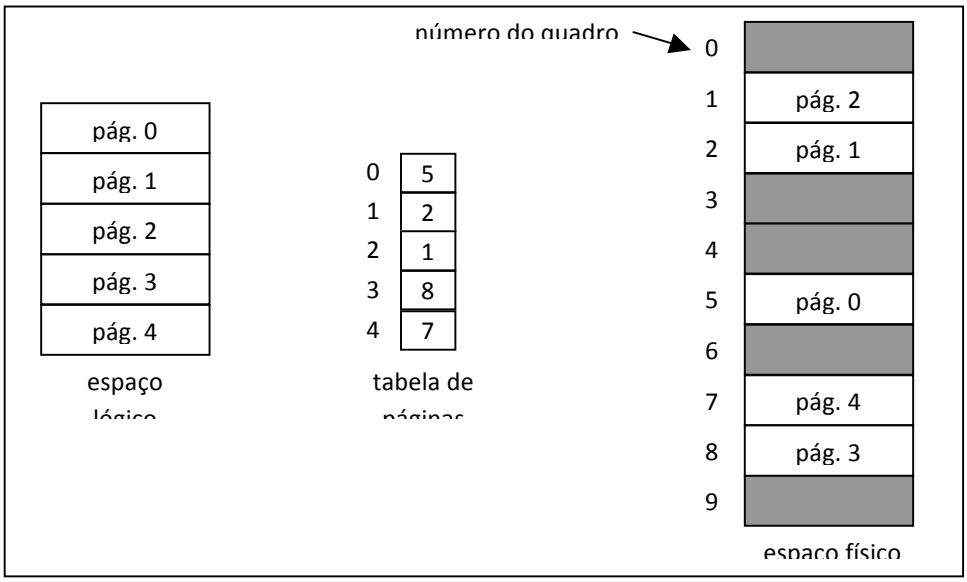


Figura 11- Visão geral da paginação.

A **tradução** de um endereço **lógico**, referenciado por um processo, para o endereço **físico** correspondente ocorre com o auxílio da **tabela de páginas**. Para tanto, substitui-se os bits que correspondem ao **número da página** pelo **número do *frame*** que a contém. É responsabilidade do SO criar e manter uma **tabela de páginas** para cada processo ativo. É o SO que aloca (escolhe entre as disponíveis) páginas físicas (*frames*) para conter as páginas lógicas dos processos. A tradução dos endereços, contudo, ocorre durante a execução das instruções e é feita de maneira eficiente por um componente do processador, chamado de unidade de gerenciamento de memória (*Memory Management Unit* – MMU). [1]

Tabela de páginas

15	000	0
14	000	0
13	000	0
12	000	0
11	111	1
10	000	0
9	101	1
8	000	0
7	000	0
6	000	0
5	011	1
4	100	1
3	000	1
2	110	1
1	001	1

Figura 12- Tabela de páginas.

[1] É importante observar que **não** é o SO que faz as trduções de endereço. Ao restaurar o contexto de um processo, o SO ajusta também o endereço da tabela de páginas deste processo, para que o **processador** faça as traduções de maneira apropriada.

Segmentação

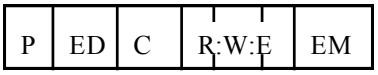
Outra forma de fazer a tradução de endereços virtuais em endereços lógicos consiste em usar a técnica de **segmentação.** De maneira semelhante à paginação, com a segmentação, blocos de endereços lógicos, de tamanhos variados, são mapeados para endereços físicos. Novamente, tem-se endereços bidimensionais, agora com uma parte indicando o segmento lógico a que pertencem, e outra parte indicando o deslocamento dentro desse segmento. Uma tabela de mapeamento de segmentos é mantida pelo SO nesse caso também e, da mesma forma, cabe à MMU no processador usar essa tabela para traduzir os endereços lógicos (virtuais) em endereços físicos.

- Programa particionado, pelo programador ou compilador, em blocos de tamanhos variados
- Tentativa de posicionar objetos diferentes em segmentos distintos, desobrigando o gerente de memória da necessidade de alocar um espaço contíguo muito grande e facilitando o compartilhamento de objetos entre processos.
- Cada segmento contém um programa ou módulos de dados, sem particioná-los.
- Ex: procedimentos, estruturas de dados distintas e pilha poderiam ser tratados como segmentos distintos.
- Mapeamento dos endereços dos segmentos;
  - Mapeamento Direto na Memória
  - *Cache*, Memória Associativa

Compartilhamento: ponteiros para segmentos comuns.

- Compartilhamento: ponteiros para segmentos comuns.

Tabela de Segmentos:



P : Presença do segmento na memória

ED : Endereço do segmento no disco

C : Comprimento do segmento

RWE : Bits de proteção - R : Leitura
W : Escrita
E : Execução

Segmentação Paginada

- Combinação das vantagens de segmentação e paginação
- Divisão lógica dos programas em segmentos que, por sua vez, são divididos em páginas
- Tamanho dos segmentos: múltiplos do tamanho das páginas
- Localização das páginas mais utilizadas é mantida na memória associativa
- Endereçamento Tri-dimensional: **v (s,p,d)**

A figura 13 ilustra a tradução de endereços usando memória virtual com segmentação paginada.

s: # do segmento
p: # da página
d: deslocamento dentro da página

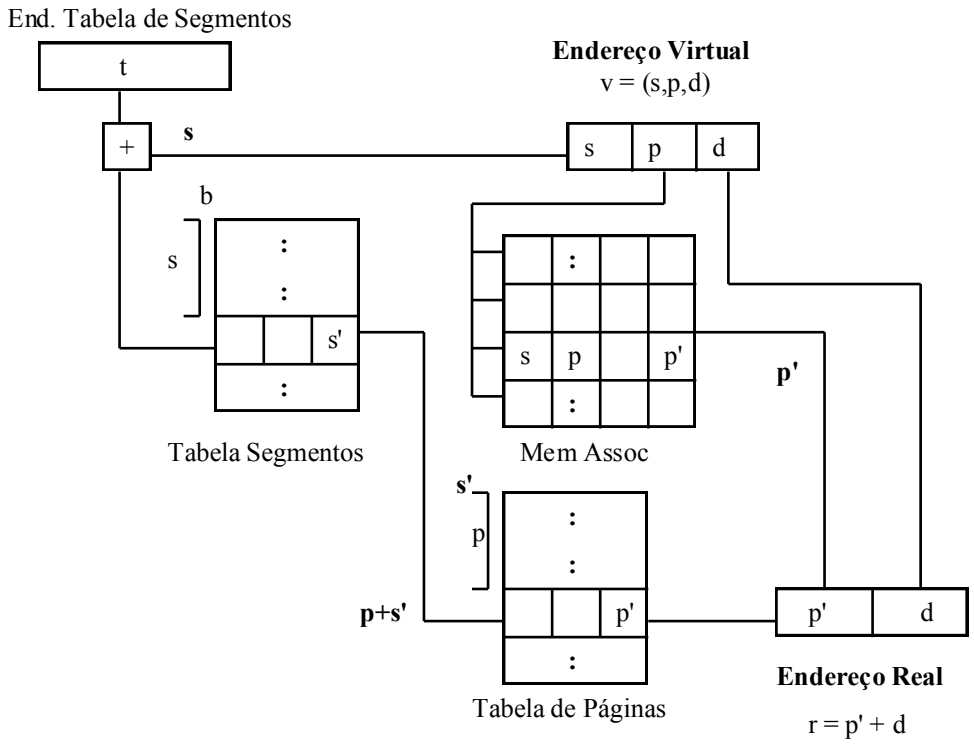


Figura 13- Mapeamento de endereços.

Gerenciamento de memória virtual

Outros aspectos de interesse na organização de memória virtual com tradução de endereços, incluem:

- Quando são carregadas as páginas lógicas dos processos em páginas físicas?
  - O SO pode fazer o carregamento todo dessas páginas *a priori*, pode tentar antecipar o carregamento das páginas que considera que serão usadas em breve, ou pode ir carregando-as por demanda, à medida que são necessárias.
  - Usando as informações na tabela de páginas, a MMU consegue identificar quando uma página não está presente na memória. Quando isso acontece, o processador gera um *exceção* de falta de página, que é tratada pelo SO, alocando uma página física para conter a página lógica em falta.

- Onde ficam armazenadas as páginas lógicas de um processo que não estão residentes em memória?
  - Inicialmente, elas estão associadas ao arquivo executável que deu origem à criação do processo. Uma vez carregada para a memória, se o espaço (*page frame*) usado por uma

página for necessário para outro processo, o SO pode gravar essa página lógica numa área de *swap* em disco, de onde é recuperada posteriormente, quando for novamente necessária.

- Como alocar páginas físicas quando não há páginas (*frames*) disponíveis?
  - O SO pode ter uma política de substituição de páginas. Essa política busca identificar páginas lógicas de processos que não serão usadas (ao menos num curto período de tempo). Identificada uma página física candidata, o SO verifica se há uma cópia atualizada de seu conteúdo na área de *swap*, ou se ela pode ser recuperada a partir do arquivo executável. Se for esse o caso, o SO apenas ajusta a tabela de páginas do processo que está cedendo a página física, marcando que a página lógica correspondente não está mais presente na memória. Se não houver uma cópia em disco da página sendo substituída, o SO a copia para uma área de *swap* e atualiza a informação de sua ausência na tabela de páginas.
  - Políticas para a escolha da página a substituir incluem: FIFO, selecionar entre aquelas que não foram usadas recentemente, selecionar a página que foi usada pela última vez há mais tempo, e variações dessas estratégias.

Referêncis adicionais:

Para o estudo desse assunto é relevante ler o conteúdo dos capítulos 9, 10 e 11 do livro texto [1] e do capítulo 4 do livro [2].

[1] Deitel, H.M.; Deitel, P.J. and Choffnes, D.R. *Sistemas Operacionais*. 3a edição. Pearson, 2005.

[2] Tanembaum, A. S. *Sistemas Operacionais Modernos*. Pentice Hall Brasil, 2003.