A técnica particular que determinado sistema operacional emprega depende, entre outras coisas, de o que a arquitetura do computador em questão suporta. As técnicas de gerência de memória estão intimamente ligadas ao *hardware* do computador. Em Bach (1986), Comer (1984), Tanebaum e Woodhull (2008) e Vahalia (1996), podem ser encontradas as descrições de algumas soluções empregadas em sistemas operacionais específicos. Essas soluções variam com relação à funcionalidade oferecida e à complexidade dos mecanismos empregados. Na prática, as arquiteturas existentes no mercado possuem uma série de detalhes que tornam complexa a implementação dos mecanismos de gerência de memória. Uma excelente descrição de arquiteturas contemporâneas pode ser encontrada em Jacob e Mudge (1998a, 1998b).

6.1 → memória lógica e memória física

A **memória lógica** de um processo é aquela que o processo enxerga, ou seja, aquela que ele é capaz de endereçar e acessar usando as suas instruções. Os endereços manipulados pelo processo são endereços lógicos. Em outras palavras, as instruções de máquina de um processo especificam endereços lógicos. Por exemplo, um processo executando um programa escrito na linguagem C manipula variáveis tipo *pointer*. Essas variáveis contêm endereços lógicos. Em geral, cada processo possui a sua memória lógica, que é independente da memória lógica dos outros processos.

A **memória física** é aquela implementada pelos circuitos integrados de memória, pela eletrônica do computador. O endereço físico é aquele que vai para a memória física, ou seja, é usado para endereçar os circuitos integrados de memória.

O **espaço de endereçamento lógico** de um processo é formado por todos os endereços lógicos que esse processo pode gerar. Existe um espaço de endereçamento lógico por processo. Já o **espaço de endereçamento físico** é formado por todos os endereços aceitos pelos circuitos integrados de memória.

A **unidade de gerência de memória** (*memory management unit*, MMU) é o componente do *hardware* responsável por prover os mecanismos básicos que serão usados pelo sistema operacional para gerenciar a memória. Entre outras coisas, é a MMU que vai mapear os endereços lógicos gerados pelos processos nos correspondentes endereços físicos que serão enviados para a memória. A figura 6.1 ilustra o papel da MMU entre o processador e a memória. Na verdade, o processador e a MMU formam, na maioria das vezes, um único circuito integrado. É interessante observar que o processador trabalha sempre com endereços lógicos.

No capítulo sobre multiprogramação, foi mostrado um exemplo de *hardware* de proteção para a memória. A figura 6.2 reproduz aquele exemplo. Podemos considerar os dois registradores de limite como uma MMU muito simples. Nesse caso, considera-se ainda que os endereços lógicos e físicos possuem valores idênticos. Isso é, quando o

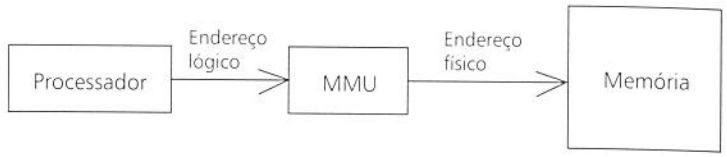

figura 6.1 Diagrama incluindo a MMU entre o processador e a memória.

processo gera o endereço lógico 123, a memória recebe o endereço físico 123. Entretanto, o espaço de endereçamento lógico de um processo de usuário é limitado. O conteúdo dos registradores de limite em um dado momento definem o espaço de endereçamento lógico. No exemplo da figura 6.2, o espaço de endereçamento lógico do processo em execução vai de 100 a 799. Qualquer endereço lógico fora desse intervalo será considerado ilegal. Já o espaço de endereçamento físico é sempre toda a memória principal.

Uma outra forma de MMU simples é mostrada na figura 6.3. Agora o endereço lógico gerado pelo processo é primeiro comparado com um limite superior. Caso seja menor ou igual, ele então é somado ao valor do registrador de base. O resultado da soma é o endereço físico que vai para a memória. Nesse esquema, o endereço lógico é transformado em endereço físico por meio da soma do valor da base. Temos então o endereço lógico diferente do respectivo endereço físico.

Nesse esquema de MMU, o espaço de endereçamento lógico vai de zero até o valor limite. Esses são os endereços de memória manipulados pelo processo. No caso do exemplo da figura 6.3, o processo pode gerar endereços lógicos entre zero e 200 (qualquer valor fora desse intervalo será considerado ilegal). Os endereços lógicos são mapeados pela MMU para uma área do espaço de endereçamento físico. Essa área da memória física inicia no valor indicado pelo registrador de base

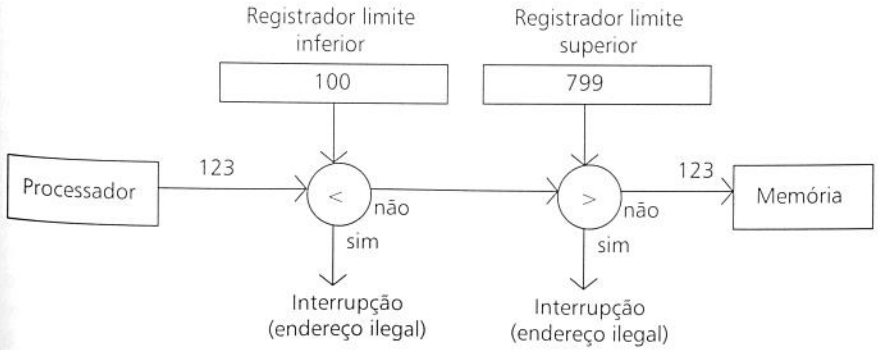

figura 6.2 Mecanismo de proteção para a memória com registradores de limite.

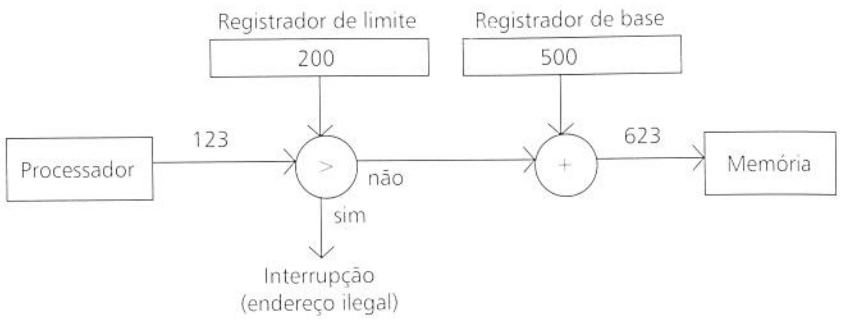

figura 6.3 Mecanismo de proteção para a memória com registradores de base e limite.

e tem o mesmo tamanho da memória lógica do processo. Observe que a proteção de memória é conseguida, pois o processo de usuário está restrito a essa área da memória física.

Tanto os registradores de limite inferior e superior quanto os registradores de base e limite devem ser protegidos, eles não podem ser acessados em modo usuário. Obviamente, eles devem poder ser acessados em modo supervisor. O conteúdo desses registradores passa a fazer parte do contexto de execução dos processos. Podem ser mantidos no descritor de processo (DP) juntamente com as demais informações do seu contexto de execução. Quando ocorre um chaveamento de processo, os valores são copiados do DP para os registradores da MMU, limitando assim a região de memória a qual o processo que recebe o processador tem acesso.

Uma diferença básica entre as soluções das figuras 6.2 e 6.3 está na carga dos programas. No esquema que emprega apenas registradores de limite, os programas são gerados para o endereço zero de memória. Entretanto, eles serão provavelmente carregados em um outro endereço da memória física. O endereço inicial do programa na memória física somente é conhecido no momento da carga, pois depende de quais outros programas estão sendo executados e de quanta memória eles ocupam. Dessa forma, no momento da carga, os endereços do programa devem ser corrigidos para que o programa execute corretamente no lugar onde foi colocado. Esse processo de correção de endereços é chamado de **relocação**. Um carregador que efetua uma relocação do programa em tempo de carga é chamado de **carregador relocador**.

No esquema que emprega registradores de base e limite, todos os programas são também gerados para o endereço zero de memória. Entretanto, eles podem ser carregados em qualquer lugar da memória física. Com o registrador de base contendo o endereço físico inicial, o programa funciona sem alterações em qualquer lugar da memória. Um carregador de programas que não precisa corrigir os endereços durante a carga é chamado de **carregador absoluto**. Nesse esquema, podemos considerar que

ocorre uma relocação em tempo de execução, pois cada endereço sofre uma correção automática ao ser somado com o conteúdo do registrador de base.

A questão da relocação em tempo de carga ilustra a importância do suporte do *hardware* para o sistema operacional, nesse caso, o suporte que a MMU oferece para a gerência de memória. Ao longo deste capítulo, será visto que MMU mais sofisticadas permitem soluções mais eficientes para o problema da gerência da memória. O anexo A descreve as operações básicas realizadas por montadores, ligadores e carregadores.

6.2 → partições fixas

Partições fixas são a forma mais simples de gerência de memória para multiprogramação. A memória é primeiramente dividida em uma parte para uso do sistema operacional e uma parte para uso dos processos de usuários. A seguir, a parte dos usuários é dividida em várias partições de tamanhos diferentes porém fixos. Quando um programa deve ser carregado, é escolhida uma partição ainda livre. Obviamente, a partição deve ter um tamanho igual ou maior que o programa. Caso não exista uma partição livre que seja grande o bastante para conter o programa, ele não poderá ser executado no momento. Deverá esperar até que um dos processos em execução termine, liberando uma partição de tamanho suficiente. A figura 6.4 mostra a memória com quatro partições de diferentes tamanhos, além da área reservada para o sistema operacional. Partições fixas podem ser implementadas tanto com registradores de limite superior e inferior quanto com registradores de base e limite.

Existem dois problemas com esse tipo de gerência de memória. Dificilmente o programa a ser carregado terá o tamanho exato de uma partição. Ele será carregado em uma

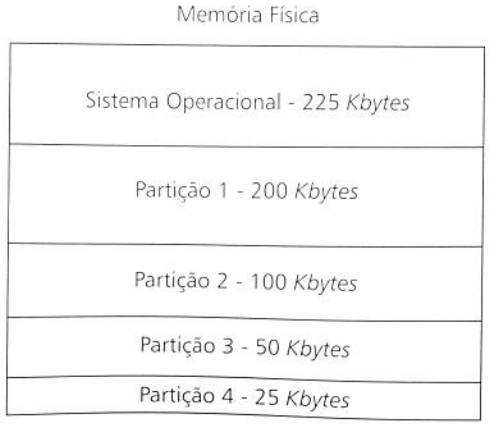

figura 6.4 Memória física dividida em partições fixas.

partição que é um pouco maior que o necessário. Isso resulta em um desperdício de memória que é chamado de **fragmentação interna**, isso é, memória perdida dentro da área alocada para um processo. Outra possibilidade é termos duas partições livres, digamos, de 25 e 100 *Kbytes*. Nesse momento é criado um processo para executar um programa de 110 *Kbytes*. Observe que a memória total livre no momento é de 125 *Kbytes*, mas ela não é contígua. O programa não pode ser executado devido à forma como a memória é gerenciada. Esse tipo de problema é chamado de **fragmentação externa**, isso é, memória perdida fora da área ocupada por um processo.

6.3 → partições variáveis

Quando **partições variáveis** são empregadas, o tamanho das partições é ajustado dinamicamente às necessidades exatas dos processos. Essa é uma técnica de gerência de memória mais flexível que partições fixas.

O sistema operacional mantém uma **lista de lacunas**, ou seja, de espaços livres na memória física. Quando um processo é criado, a lista de lacunas é percorrida. Será usada uma lacuna de tamanho maior ou igual ao tamanho do programa em questão. Entretanto, o que a lacuna original tiver a mais que o necessário para executar o programa será transformado em uma nova lacuna, apenas menor que a original. Dessa forma, o programa vai receber o tamanho exato de memória que necessita.

Existem quatro formas básicas de percorrer a lista de lacunas atrás de uma lacuna de tamanho suficiente. Os algoritmos ***first-fit*** e ***circular-fit*** são os mais utilizados. São eles:

- *First-fit*: utiliza a primeira lacuna que encontrar com tamanho suficiente;
- *Best-fit*: utiliza a lacuna que resultar na menor sobra;
- *Worst-fit*: utiliza a lacuna que resultar na maior sobra;
- *Circular-fit*: como *first-fit*, mas inicia a procura na lacuna seguinte à última sobra.

Quando um processo termina, a memória que ele ocupava é liberada. Isso corresponde à criação de uma nova lacuna. Caso a nova lacuna criada seja adjacente a outras lacunas, elas são unificadas. A figura 6.5a mostra uma memória com três processos e uma lacuna. Existe um processo esperando para ser executado, mas não existe uma lacuna com tamanho suficiente. Suponha que o processo 2 termine. Sua área de memória é transformada em uma lacuna, o que permite que o programa do processo 4 seja carregado. A figura 6.5b mostra a situação final.

Se o tamanho exato de cada programa é sempre alocado, não ocorre fragmentação interna. Entretanto, alocar sempre o tamanho exato pode gerar lacunas de alguns poucos *bytes*. Não é interessante arcar com o custo, em termos de memória e tempo de processamento, para manter lacunas de poucos *bytes*. Alguns sistemas organizam a memória em blocos de, por exemplo, 32 *bytes*. Esses blocos são muitas vezes chamados de **parágrafos**. A unidade de alocação passa a ser o parágrafo e o tamanho da área alocada por um processo deve ser um número inteiro de parágrafos, ou seja, um múltiplo exato de 32 *bytes*. Dessa forma, a menor lacuna possível terá o tamanho de um parágrafo ou 32 *bytes*. Nesse caso, poderemos ter uma fragmentação interna de até 31 *bytes* por processo.

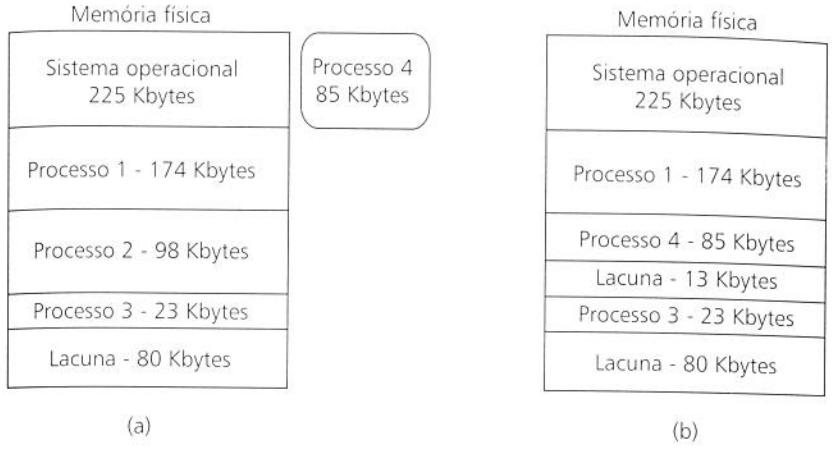

figura 6.5 Memória física dividida em partições variáveis.

Apesar da fragmentação interna introduzida, esse esquema possui várias vantagens. Em algumas arquiteturas, as variáveis do tipo inteiro devem ficar alinhadas corretamente na memória. Por exemplo, a arquitetura exige que inteiros de 4 *bytes* sejam posicionados necessariamente a partir de um *byte* com endereço par. A alocação de memória baseada em parágrafos facilita o atendimento das exigências de arquiteturas desse tipo. Também são necessários menos *bits* para endereçar uma lacuna na memória. Considerando uma memória com 1 *Gbyte*, são necessários 30 *bits* para endereçar um *byte* específico, mas apenas 25 *bits* para endereçar um parágrafo em particular.

Partições variáveis são tipicamente implementadas através de uma lista encadeada de lacunas. Cada lacuna é representada por um descritor de lacuna, que contém, basicamente, o seu endereço, tamanho e apontadores para as lacunas adjacentes. Como cada lacuna tem o tamanho mínimo de um parágrafo, ela pode hospedar o seu próprio descritor. Dessa forma, não é necessário alocar memória especialmente para uma "tabela de lacunas", pois a memória das próprias lacunas é usada para implementar a lista de lacunas. As lacunas são mantidas ordenadas pelo endereço, para facilitar a detecção de lacunas adjacentes, quando a situação acontecer.

Com partições variáveis a fragmentação externa é um problema grave. À medida que áreas de memória são alocadas e liberadas, muitos fragmentos são gerados. Ou seja,

uma grande quantidade de lacunas pequenas demais para serem úteis. A memória adquire uma aparência de "queijo suíço". Não é incomum perder 1/3 da memória devido à fragmentação externa.

É possível tentar usar compactação de memória para eliminar esse problema. Processos são deslocados na memória de forma que as lacunas existentes fiquem adjacentes umas às outras e possam, então, ser unificadas. Se o processo de compactação for completo, ao final teremos uma única lacuna formada por toda a memória livre. Entretanto, deslocar processos na memória gasta muito tempo de processador e também exige um mecanismo de relocação dinâmica, como o implementado pelo registrador de base. Em geral, não é um recurso utilizado.

6.4 →swapping

Existem situações nas quais não é possível manter todos os processos simultaneamente na memória. Por exemplo, considere a figura 6.5a. Suponha que o processo 2 faça uma chamada de sistema, solicitando que sua área de memória seja aumentada. Embora existam áreas de memória ainda livres, nenhuma é contígua à área ocupada pelo processo 2. Outro exemplo é a situação na qual um usuário em terminal solicita o disparo de um programa, e não existe memória disponível no momento, mas é política do sistema disparar imediatamente todos os programas que são solicitados via um terminal.

Uma solução para essas situações é o mecanismo chamado de ***swapping***. A gerência de memória reserva uma área do disco para o seu uso. Em determinadas situações, um processo é completamente copiado da memória para o disco. Sua execução é suspensa, ou seja, seu descritor de processo é removido da fila do processador e colocado em uma fila de processos suspensos. É dito que esse processo sofreu um ***swap-out***. Mais tarde, ele sofrerá um ***swap-in***, ou seja, será copiado novamente para a memória. Seu descritor de processo volta então para a fila do processador, e sua execução será retomada. O resultado desse revezamento no disco é que o sistema operacional consegue executar mais processos do que caberia em um mesmo instante na memória.

Swapping impõe aos programas um grande custo em termos de tempo de execução. Copiar todo o processo da memória para o disco e mais tarde de volta para a memória é uma operação demorada. É necessário deixar o processo um tempo razoável no disco para justificar tal operação. Por exemplo, em torno de alguns segundos.

Em sistemas nos quais uma pessoa interage com o programa durante a sua execução (chamado antigamente de modo *timesharing*), o mecanismo de *swapping* somente é utilizado em último caso, quando não é possível manter todos os processos na memória. A queda no desempenho do sistema é imediatamente sentida pelo usuário no terminal. Para processos que são executados em *background*, ou seja, desvinculados de um terminal, o mecanismo torna-se mais aceitável.

Swapping pode ser usado tanto com partições fixas como com partições variáveis. Caso o processo, no momento do *swap-in*, volte para uma posição diferente de memória, é necessário corrigir os seus endereços. Isso não é necessário quando se usa um mecanismo baseado em registrador de base, como mostrado na figura 6.3. Nesse caso, basta corrigir o conteúdo do registrador de base, e o programa executará corretamente em sua nova posição na memória física.

6.5 →paginação

A técnica de partições fixas gera muita perda de memória e não é mais utilizada na prática. Embora partições variáveis seja um mecanismo mais flexível, o desperdício de memória em função da fragmentação externa é um grande problema. A origem da fragmentação externa está no fato de cada programa necessitar ocupar apenas uma área contígua de memória. Se essa restrição for eliminada, ou seja, permitir que um programa ocupasse áreas não contíguas de memória, não haveria fragmentação externa. A técnica de **paginação** possibilita exatamente isso.

A figura 6.6 ilustra o funcionamento da técnica de paginação. O exemplo da figura utiliza um tamanho de memória exageradamente pequeno para tornar a figura mais clara e menor. O espaço de endereçamento lógico de um processo é dividido em **páginas lógicas** de tamanho fixo. No exemplo, todos os números mostrados são valores binários. A memória lógica é composta por 12 *bytes*. Ela foi dividida em 3 páginas lógicas de 4 *bytes* cada uma. O endereço lógico também é dividido em duas partes: um **número de página lógica** e um **deslocamento** dentro dessa página. No exemplo, endereços lógicos possuem 5 bits. Considere o *byte* Y2. Ele possui o endereço lógico 00101. Podemos ver esse endereço composto por duas partes: os primeiros 3 *bits* indicam o número da página, isso é, 001; os últimos 2 *bits* indicam a posição de Y2 dentro da página, isso é, 01. Observe que todos os *bytes* pertencentes a uma mesma página lógica apresentam o mesmo número de página. De forma semelhante, todas as páginas possuem *bytes* com deslocamento entre 00 e 11.

A memória física também é dividida em **páginas físicas** com tamanho fixo, idêntico ao tamanho da página lógica. No exemplo, a memória física é composta por 24 *bytes*. A página física tem o mesmo tamanho que a página lógica, ou seja, 4 *bytes*. A memória física foi dividida em 6 páginas físicas de 4 *bytes* cada uma. Os endereços de memória física também podem ser vistos como compostos por duas partes: os 3 primeiros *bits* indicam um número de página física; os 2 últimos *bits* indicam um deslocamento dentro dessa página física.

Um programa é carregado página a página. Cada página lógica do processo ocupa exatamente uma página física da memória física. Entretanto, a área ocupada pelo processo na memória física não precisa ser contígua. Mais do que isso, a ordem em que as páginas lógicas aparecem na memória física pode ser aleatória, não precisa ser a mesma da memória lógica. Observe que, no exemplo, o conteúdo das páginas

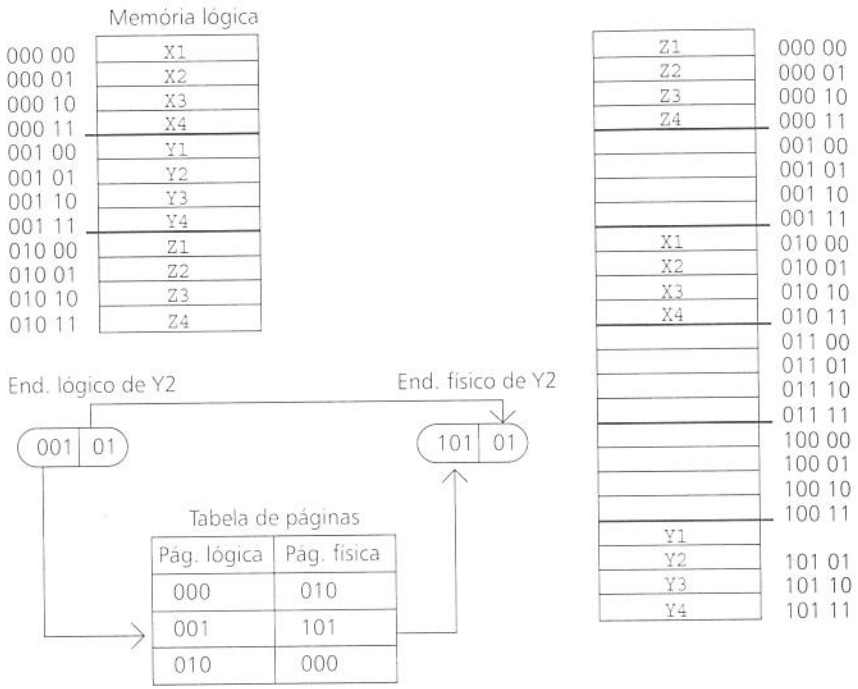

figura 6.6 Mecanismo básico de paginação.

lógicas foi espalhado pela memória física, mantendo o critério de que cada página lógica é carregada em exatamente uma página física. A página lógica 000 (X1, X2, X3 e X4) foi carregada na página física 010. A página lógica 001 (Y1, Y2, Y3 e Y4) foi carregada na página física 101. A página lógica 010 (Z1, Z2, Z3 e Z4) foi carregada na página física 000.

Durante a carga, é montada uma **tabela de páginas** para o processo. Essa tabela informa, para cada página lógica, qual a página física correspondente. No exemplo, a tabela é formada por 3 entradas, uma vez que o processo possui 3 páginas lógicas.

Quando um processo executa, ele manipula endereços lógicos. O programa é escrito com a suposição que ele vai ocupar uma área contígua de memória que inicia no endereço zero, ou seja, vai ocupar a memória lógica do processo. Para que o programa execute corretamente, é necessário transformar o endereço lógico especificado em cada instrução executada no endereço físico correspondente. Isso é feito com o auxílio da tabela de páginas.

O endereço lógico gerado é inicialmente dividido em duas partes: um número de página lógica e um deslocamento dentro da página. O número da página lógica é

usado como índice no acesso à tabela de páginas. Cada entrada da tabela de páginas possui o mapeamento de página lógica para página física. Dessa forma, é obtido o número da página física correspondente. Já o deslocamento do *byte* dentro da página física será o mesmo deslocamento desse *byte* dentro da página lógica, pois cada página lógica é carregada exatamente em uma página física. Basta juntar o número de página física obtido na tabela de páginas com o deslocamento já presente no endereço lógico para obter-se o endereço físico do *byte* em questão. A figura 6.6 mostra como o endereço lógico do *byte* Y2 é transformado no endereço físico correspondente. O endereço lógico 00101 é dividido em número de página lógica 001 e deslocamento 01. A entrada 001 da tabela de páginas indica que essa página lógica foi carregada na página física 101. Finalmente, as duas partes são unidas, formando o endereço físico 10101.

Na prática, os tamanhos de página variam entre 1 *Kbytes* e 8 *Kbytes*. Espaços de endereçamento lógico variam de 64 *Kbytes* para sistemas antigos até muitos *Gbytes* para máquinas atuais. Espaços de endereçamento físico também ficam, em geral, na ordem de *Gbytes*. Note que o espaço de endereçamento físico denota a capacidade de endereçamento do processador, e não a quantidade de memória realmente instalada na máquina. Embora o processador de um computador doméstico atual possa endereçar 32 *Gbytes*, tipicamente a memória física realmente instalada é menor que 1 *Gbyte*.

Na paginação, uma página lógica pode ser carregada em qualquer página física que esteja livre. Dessa forma, não existe fragmentação externa. Como a unidade de alocação é a página, e o tamanho de um processo raramente é igual a um múltiplo do tamanho de página, isso introduz uma fragmentação interna. Suponha que um sistema no qual as páginas são de 4 *Kbytes*, e um programa necessita 201 *Kbytes* para executar. Serão alocadas para ele 51 páginas, totalizando 204 *Kbytes*. Isso resultará em uma fragmentação interna de 3 *Kbytes*. Em média, podemos esperar uma fragmentação interna de meia página por processo.

Existem vantagens e desvantagens em utilizar páginas grandes. Páginas maiores significam que um processo terá menos páginas, a tabela de páginas será menor e a leitura do disco será mais eficiente. Em geral, páginas maiores resultam em um custo menor imposto pelo mecanismo de gerência de memória, ou seja, um ***overhead*** menor. Por outro lado, páginas maiores resultam em uma fragmentação interna maior. Normalmente não é o sistema operacional que escolhe o tamanho das páginas. Esse valor é fixado pelo *hardware* que suporta a gerência de memória, ou seja, pela MMU do computador em questão.

A gerência de memória deve manter controle das áreas ainda livres na memória. Isso significa manter uma lista de páginas físicas livres. Uma forma de implementar esse controle é por meio de um mapa de *bits*. Cada *bit* representa o estado de uma página física em particular (por exemplo, 0 indica página livre e 1 indica página ocupada). Para localizar uma página física livre, basta percorrer o mapa de *bits* até encontrar

um *bit* 0. A posição do *bit* no mapa indica o número da página física livre. Esse mecanismo poderá ficar muito lento quando a memória física for grande e estiver praticamente toda ocupada. Uma alternativa é manter uma lista encadeada com os números das páginas físicas livres. Para localizar uma página física livre basta pegar o primeiro elemento dessa lista. As próprias páginas livres da memória podem ser utilizadas para armazenar a lista.

Um aspecto importante da paginação é a forma como a tabela de páginas é implementada. Observe que ela deve ser consultada a cada acesso à memória. Os próximos parágrafos discutem três formas usadas para implementar a tabela de páginas, sendo a última delas a mais usada atualmente.

Quando a tabela de páginas é pequena, ela pode ser completamente colocada em registradores de acesso rápido. Por exemplo, um computador no qual a memória lógica dos processos seja de apenas 64 *Kbytes* e cada página ocupe 8 *Kbytes*, terá apenas 8 entradas nas tabelas de páginas. Registradores apresentam a vantagem de serem rápidos e, portanto, não degradam o tempo de acesso à memória. Quando ocorre um chaveamento de processos, a tabela de páginas do processo que recebe o processador deve ser copiada do descritor de processo (DP) para os registradores.

Quando a tabela de páginas é muito grande, não é possível mantê-la em registradores. Outra solução é manter a tabela de páginas na própria memória. A MMU possui então dois registradores para localizar a tabela na memória. O **registrador de base da tabela de páginas** (*page table base register*, PTBR) indica o endereço físico de memória onde a tabela está colocada. O **registrador de limite da tabela de páginas** (*page table limit register*, PTLR) indica o número de entradas da tabela. O problema desse mecanismo é que agora cada acesso que um processo faz à memória lógica transforma-se em dois acessos à memória física. No primeiro acesso, a tabela de páginas é consultada, e o endereço lógico é transformado em endereço físico. No segundo acesso, a memória do processo é lida ou escrita. Quando ocorre um chaveamento de processos, os valores do PTBR e do PTLR para a tabela de páginas do processo que recebe o processador devem ser copiados do DP para os registradores na MMU.

Uma forma de reduzir o tempo de acesso à memória no esquema anterior é adicionar uma memória *cache* especial que vai manter as entradas da tabela de páginas mais recentemente utilizadas. Essa memória *cache* interna à MMU é chamada normalmente de ***translation lookaside buffer*** (**TLB**). O acesso a essa *cache* é rápido e não degrada o tempo de acesso à memória como um todo. Quando a entrada requerida da tabela de páginas está na TLB, o acesso à memória lógica do processo é feito com um único acesso à memória física, como quando registradores são utilizados. Nesse caso, é dito que tivemos um acerto (*hit*). Quando a entrada da tabela de páginas associada com a página lógica acessada não está na TLB (*miss*), é necessário um duplo acesso à memória física, como no esquema anterior. Nesse caso, a entrada referente a página lógica acessada é incluída na TLB, na suposição (correta na maioria das vezes) de que o processo acessará essa página mais vezes logo em seguida. Os próximos acessos já encontrarão essa entrada da tabela de páginas na TLB e o acesso será rápido.

Normalmente, a memória *cache* é implementada por um componente de *hardware* conhecido como memória associativa. A memória associativa inclui não somente algumas células de memória, mas também toda a eletrônica necessária para fazer uma pesquisa paralela, incluindo todas as células. O resultado é um *hardware* caro, o que limita o tamanho das memórias associativas utilizadas na prática. Entretanto, mesmo memórias associativas de 16 a 32 entradas permitem taxas de acerto de 80% a 90%. O resultado final é um tempo de acesso à memória com paginação que é apenas 20% a 30% maior do que seria caso não houvesse paginação.

Quando a memória *cache* é usada e ocorre um chaveamento de processos, novamente os valores do PTBR e do PTLR para a tabela de páginas do processo que recebe o processador devem ser copiados do DP para os registradores na MMU. Além disso, a memória *cache* deve ser esvaziada (*flushed*). Isso é necessário, pois ela ainda contém entradas da tabela de página do processo que executou antes e agora perdeu o processador. Esquemas alternativos incluem o número do processo em cada entrada da TLB para resolver esse problema.

Normalmente, cada entrada da tabela de páginas possui, além do número da página física correspondente, uma série de *bits* que auxiliam na gerência da memória. É comum a inclusão de um *bit* de válido/inválido. Quando o processo tenta acessar uma página que está marcada como inválida, a MMU gera uma interrupção de proteção, e o sistema operacional é acionado. Em geral, isso representa um erro de programação, pois o processo está tentando acessar uma página que não faz parte da sua memória lógica. Entretanto, existem também outros usos para esse *bit*, como será visto no capítulo sobre memória virtual.

Também é comum a inclusão de *bits* que indicam como o conteúdo daquela página pode ser usado. Tipicamente, uma página pode ser para apenas leitura (*read-only*, RO), para apenas execução (*execute-only*, XO) ou para leitura e escrita (*read-write*, RW). Caso o processo tente acessar a página de uma maneira diferente daquela determinada pelos *bits* de proteção, a MMU gera uma interrupção de proteção e aciona o sistema operacional. Esse mecanismo permite, entre outras coisas, que erros de programação sejam detectados.

É importante observar que a proteção entre processos é facilmente conseguida com uma MMU que suporte paginação. Em primeiro lugar, o mecanismo de paginação garante que cada processo somente tenha acesso às páginas físicas que constam em sua tabela de páginas. Essa tabela de páginas é construída pelo sistema operacional e fica em uma região da memória à qual apenas o sistema operacional tem acesso. O acesso aos registradores PTBR e PTLR é privilegiado, isso é, restrito ao código do sistema operacional, que executa em modo supervisor. Finalmente, o sistema operacional ainda dispõe dos *bits* de proteção na tabela de páginas para controlar como cada

página é acessada e para marcar como inválidas as entradas da tabela de páginas que o processo não pode usar.

Em sistemas atuais, a tabela de páginas pode ser muito grande. Considere como exemplo o processador 80386 da Intel que trabalha com endereços de 32 *bits*. O espaço de endereçamento lógico pode ser de até 4 *Gbytes* (1 *Gbyte* equivale a 1024×1024×1024 *bytes*, ou ainda, 2 elevado a 30 *bits*). Cada página ocupa 4 *Kbytes*. Isso significa que um processo pode ter até 1.048.576 entradas na tabela de páginas, o que representaria uma tabela de páginas ocupando 4 *Mbytes* de memória (cada entrada são 4 *bytes*).

Em arquiteturas atuais, uma tabela de páginas completa raramente é utilizada. Na prática, as tabelas de páginas possuem um tamanho variável, ajustado à necessidade de cada processo. Ocorre que, se elas puderem ter qualquer tamanho, então teremos fragmentação externa novamente (a maior razão para usar paginação foi a eliminação da fragmentação externa).

Para evitar isso, são usadas tabelas de páginas com dois níveis. As tabelas de páginas crescem de pedaço em pedaço, e uma tabela auxiliar chamada "diretório" mantém o endereço de cada pedaço. Para evitar a fragmentação externa, cada pedaço da tabela de páginas deve ter um número inteiro de páginas físicas, mantendo assim toda a alocação de memória física em termos de páginas, não importando a sua finalidade. Entradas desnecessárias em cada pedaço são marcadas como inválidas.

No caso do Intel 386, a tabela de páginas é dividida em dois níveis. No primeiro nível, existe um diretório de tabelas de páginas. No segundo nível, estão os pedaços, chamados de tabelas de páginas na documentação da Intel. O endereço lógico é formado por 32 *bits*, dividido em número da entrada no diretório de tabelas de páginas (10 *bits*), número da página na tabela de páginas indicada (10 *bits*) e deslocamento dentro da página (12 *bits*). A figura 6.7 ilustra como um endereço lógico é transformado em endereço físico.

Uma alternativa seria ter toda a tabela de páginas ocupando uma área contígua de memória, dispensando o diretório, mas isso significaria usar alocação contígua de memória, o que traria problemas para o crescimento dinâmico da memória dos processos. Uma outra vantagem do esquema em dois níveis é a existência de regiões não contínuas de espaço lógico. Por exemplo, de 00000000H até 00010000H fica o programa, e de FFFF0000H até FFFFFFFFH fica a pilha desse programa. Nesse caso, uma tabela de páginas completa e contígua teria um enorme espaço inútil entre os dois extremos ocupados.

Em resumo, a implementação da tabela de páginas em dois níveis traz as seguintes vantagens:

- A tabela cresce de pedaço em pedaço, à medida que a alocação de páginas físicas acontece;

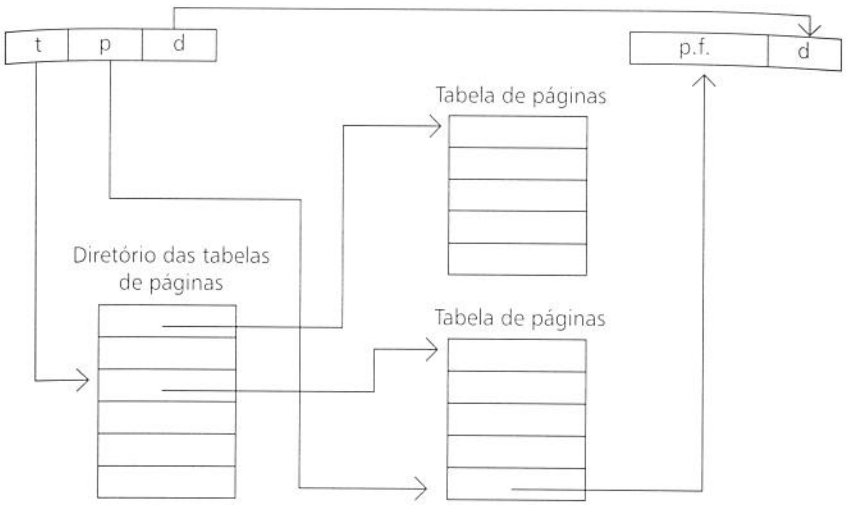

figura 6.7 Tabela de páginas organizada em dois níveis.

- Apenas o necessário é usado, com uma pequena **fragmentação interna** na própria tabela, quando não são necessárias todas as entradas do pedaço alocado;
- O processo pode "povoar" diferentes regiões do espaço lógico sem dificuldades.

É importante destacar que a implementação de paginação descrita aqui não é a única possível. Por exemplo, um esquema chamado **tabela de páginas invertida** (*inverted page table*) é usado em alguns processadores, como o PowerPC (Jacob; Mudge, 1998a).

6.6 → segmentação

O conceito de página, fundamental para a paginação, é uma criação do sistema operacional para facilitar a gerência da memória. Programadores e compiladores não enxergam a memória lógica dividida em páginas, mas sim em **segmentos**. Uma divisão típica descreve um programa em termos de quatro segmentos: código, dados alocados estaticamente, dados alocados dinamicamente e pilha de execução. Outros sistemas trabalham com uma granularidade menor. Por exemplo, cada objeto ou módulo corresponde a um segmento. Em geral, o programador atribui nomes aos segmentos, e o compilador transforma esses nomes em números. Por exemplo, o segmento "Sub-rotinas da biblioteca gráfica" passa a ser conhecido como segmento número 5.

É possível orientar a gerência de memória para suportar diretamente o conceito de segmento. Nesse caso, a memória lógica do processo passa a ser organizada em termos de segmentos. Uma posição da memória lógica passa a ser endereçada por um número

de segmento e um deslocamento em relação ao início do seu segmento. Em tempo de carga, cada segmento é copiado para a memória física, e uma **tabela de segmentos** é construída. Essa tabela informa, para cada segmento, qual o endereço da memória física onde ele foi colocado e qual o seu tamanho. A figura 6.8 ilustra essa técnica de gerência de memória. Todos os números mostrados são valores binários.

Os processos geram endereços lógicos compostos por um número de segmento e um deslocamento dentro do segmento. A MMU inicialmente utiliza o número de segmento fornecido para indexar a tabela de segmentos. Caso esse segmento não exista, é gerada uma interrupção de proteção. Uma vez localizada a entrada na tabela de segmentos, o deslocamento fornecido é comparado com o limite do segmento. Um deslocamento maior que o limite significa que o processo está tentando endereçar além do final do segmento, fora da sua memória lógica, e uma interrupção de proteção é gerada. Finalmente, o deslocamento fornecido é somado ao valor de base do segmento, resultando no endereço físico correspondente ao endereço lógico fornecido.

Considerando a figura 6.8, suponha que o processo gere o endereço lógico do *byte* D3, ou seja, segmento 01 e deslocamento 00010. A tabela de segmentos informa

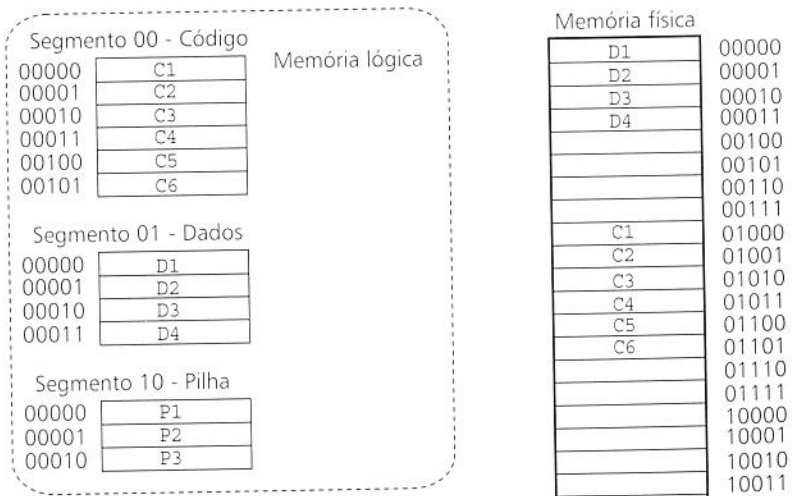

Tabela de segmentos

Segmento	Base	Limite
00	01000	0110
01	00000	0100
10	10100	0011

figura 6.8 Gerência de memória baseada em segmentos.

que o segmento 01 possui base 00000 e limite 0100. O deslocamento de 00010 é válido, pois é menor que o limite 0100. Somando a base do segmento 00000 com o deslocamento 00010, temos o endereço físico 00010. Esse é o endereço do *byte* D3 na memória física.

A tabela de segmentos pode ser implementada por meio das mesmas três formas básicas que foram apresentadas na seção sobre paginação. Os *bits* de proteção existentes nas tabelas de páginas também são usados em tabelas de segmentos. Uma diferença importante é que a segmentação não apresenta fragmentação interna, visto que a quantidade exata de memória necessária é alocada para cada segmento. Entretanto, como áreas contíguas de diferentes tamanhos devem ser alocadas, temos a ocorrência de fragmentação externa.

O grande atrativo da segmentação está na facilidade para compartilhar memória. Cada segmento representa uma parte específica do programa, podendo ou não ser compartilhado. Segmentos tendem a ser homogêneos nesse sentido. Isso é, todo o segmento pode ser compartilhado, ou nenhuma parte do segmento pode ser compartilhada. Podemos citar como exemplos o código de uma sub-rotina e uma pilha, respectivamente.

Por exemplo, suponha que o código das rotinas de biblioteca de uma linguagem de programação é compilado como sendo um segmento único. Esse segmento é marcado como "para apenas execução", ou seja, não pode ser lido nem escrito. Todos os programas escritos nessa linguagem de programação utilizam esse segmento, entretanto, apenas uma cópia dele é necessária na memória física. Todos os processos executando programas escritos nessa linguagem terão em sua respectiva tabela de segmentos uma referência à posição desse segmento na memória física. Como ele nunca é alterado, uma única cópia na memória física é suficiente para atender a todos os processos. Nesse caso, o sistema operacional deve manter uma tabela na memória, indicando quais segmentos estão na memória principal e qual a sua localização. Um segmento compartilhado somente é removido da memória principal quando nenhum processo o estiver usando, isso é, quando ele não constar mais em nenhuma tabela de segmentos em uso.

Se considerarmos um sistema com dezenas de programas executando, e a maioria deles compartilhando as mesmas bibliotecas, poderemos perceber a importância do compartilhamento de código. Também é necessário salientar a importância dos *bits* de proteção no momento de restringir o acesso dos processos aos segmentos compartilhados. Obviamente, um segmento contendo dados de um programa é do tipo "para leitura e escrita" e não pode ser compartilhado.

6.7 → segmentação paginada

Com a segmentação, o problema da fragmentação externa retorna. A sucessiva alocação e liberação de segmentos com diferentes tamanhos gera lacunas pequenas demais para serem úteis. Uma solução possível é paginar cada segmento.

Na **segmentação paginada** o espaço lógico é formado por segmentos e cada segmento é dividido em páginas lógicas. Cada segmento possui uma tabela de páginas associada. No momento de endereçar a memória, a tabela de segmentos indica, para cada segmento, onde a respectiva tabela de páginas está. Essa tabela de páginas é usada para transformar o endereço de página lógica de determinado segmento em endereço de página física, como é feito normalmente na paginação. A figura 6.9 ilustra essa configuração básica. É importante salientar que essa não é a única maneira de construir uma solução para a segmentação paginada.

A alocação de espaço em memória é feita na base de página física, eliminando completamente o problema de fragmentação externa. Entretanto, o sistema passa a ter fragmentação interna. Em média, teremos meia página de fragmentação interna por segmento de processo.

6.8 → exercícios

exercício 1 Considere um sistema cuja gerência de memória é feita por meio de partições variáveis. Nesse momento, existem as seguintes lacunas (áreas livres): 10K, 4K, 20K, 18K, 7K, 9K, 12K e 13K, nessa ordem. Quais espaços serão ocupados pelas solicitações: 5K, 10K e 6K, nessa ordem, se: (seção 6.3)

a *First-fit* for utilizado?
b *Best-fit* for utilizado?

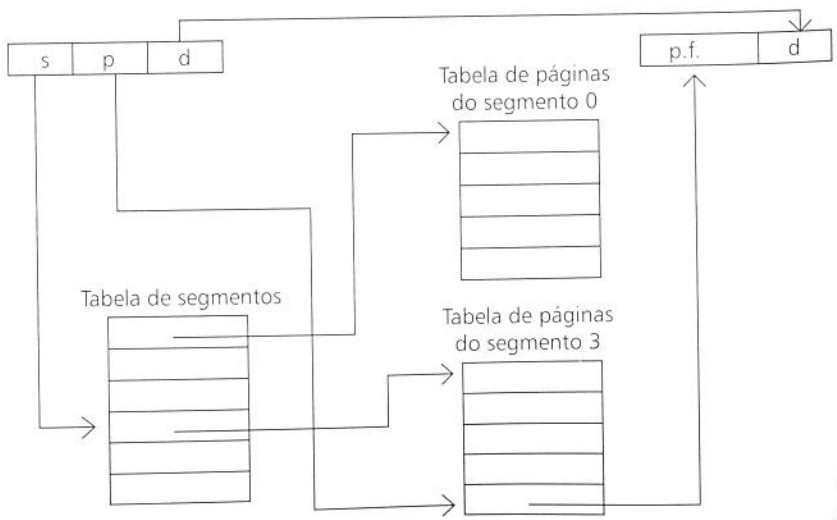

figura 6.9 Esquema clássico para segmentação paginada.

c *Worst-fit* for utilizado?
d *Circular-fit* for utilizado?

exercício 2 Considere novamente um sistema cuja gerência de memória utiliza partições variáveis. Nesse momento, existem as seguintes lacunas (áreas livres): 10K, 4K, 20K, 18K, 7K, 9K, 12K e 13K, nessa ordem. Quais espaços serão ocupados pelas solicitações: 15K, 4K e 8K, nessa ordem, se: (seção 6.3)

a *First-fit* for utilizado?
b *Best-fit* for utilizado?
c *Worst-fit* for utilizado?
d *Circular-fit* for utilizado?

exercício 3 Compare partições fixas, partições variáveis, paginação simples e segmentação simples com respeito ao compartilhamento de memória (código). Para cada uma, mostre como é possível implementar ou por que é impossível implementar. Lembre-se de que é sempre necessário manter a proteção entre usuários, e que os programas ocupam um espaço lógico contíguo.

exercício 4 Qual a fragmentação apresentada pelos métodos de gerência de memória baseados em partições fixas, partições variáveis, paginação simples e segmentação simples? Justifique.

exercício 5 Considere um sistema operacional que trabalha com paginação simples. As páginas são de *1Kbyte*. O endereço lógico é formado por 16 *bits*. O endereço físico é formado por 20 *bits*. Qual o tamanho do: (seção 6.5)

a Espaço de endereçamento lógico (maior programa possível)?
b Espaço de endereçamento físico (memória principal)?
c Entrada da tabela de páginas, sem considerar *bits* de proteção?
d Tabela de páginas (número de entradas necessárias no pior caso)?

exercício 6 Partições fixas e partições variáveis podem ser implementadas com dois tipos de *hardware*: baseado em registradores de limite ou baseado em registrador de base. Pode-se também usar *swapping* associado ao mecanismo de partições. Discuta a viabilidade de usar um ou outro *hardware*, quando a gerência de memória for: (seções 6.2/6.3)

a Partições fixas com *swapping*;
b Partições variáveis com *swapping*.

exercício 7 O professor de sistemas operacionais solicitou como trabalho aos seus alunos a construção de um ligador para um novo sistema que será desenvolvido no próximo semestre. O computador para o qual será construído o novo sistema ope-

racional foi encomendado junto ao IPIAL (Instituto de Pesquisas em Informática de Algum Lugar), mas ainda não chegou. O *hardware* para acesso à memória ainda não é conhecido. Ele deverá ser um dos seguintes:

H1 – Apenas dois registradores de limite para proteção da memória;

H2 – Um registrador de base mais um de limite;

H3 – Hardware suficiente para implementar paginação.

Em função disso, a gerência de memória a ser utilizada no novo sistema ainda não está definida. Ela poderá ser baseada em partições fixas, em partições variáveis ou em paginação.

Isso significa que ainda não está claro que tipo de arquivo executável deverá ser gerado para o novo sistema. Existem 3 possibilidades:

X1 – Código gerado para o endereço 0, acompanhado de mapa para relocação;

X2 – Código gerado para o endereço 0, sem mapa de relocação;

X3 – Código gerado para o endereço de carga correto.

Analise a viabilidade de cada tipo de arquivo executável para cada uma das gerências de memória listadas abaixo (são 27 combinações ao todo):

a Partições fixas com registradores de limite;

b Partições variáveis com registrador de base e limite;

c Paginação com hardware para paginação, obviamente.

exercício 8 O sistema operacional XYZ utiliza paginação como mecanismo de gerência de memória. São utilizadas páginas de *1Kbyte*. Um endereço lógico ocupa 20 *bits*. Um endereço físico ocupa 24 *bits*. Cada entrada na tabela de páginas contém, além do número da página física, um *bit* de válido/inválido e um bit que indica apenas leitura (*read-only*). Mostre como podem ser calculados os seguintes valores: (seção 6.5)

a Qual o tamanho máximo para a memória física;

b Qual o maior programa que o sistema suporta;

c Quantas entradas possui a tabela de páginas;

d Quantos *bits* serão necessários para a tabela de páginas (cálculo exato).

exercício 9 Mostre a diferença entre fragmentação interna e fragmentação externa. Em que situação poderá ocorrer fragmentação interna quando partições variáveis são utilizadas? (Seção 6.3.)

exercício 10 Considere um sistema que utiliza partições variáveis na gerência da memória. Para os itens: (seção 6.3)

a Arquitetura emprega apenas registradores de limite;

b Arquitetura utiliza registradores de base e de limite.

Analise a viabilidade de ser implementado *swapping*. Defina o tipo de carregador a ser utilizado.

exercício 11 Uma vez que paginação não apresenta fragmentação externa enquanto segmentação apresenta fragmentação externa, que motivo existe para justificar a ideia de segmentação? Que vantagem a segmentação pura apresenta sobre a paginação pura? (Seções 6.5/6.6.)

exercício 12 Em um sistema usando segmentação paginada, o espaço de endereçamento lógico de cada processo consiste de no máximo 16 segmentos, cada um deles podendo ter até 64 *Kbytes* de tamanho. As páginas físicas são de 512 *bytes*. Diga quantos *bits* são necessários para especificar cada uma das grandezas abaixo, explicando de onde veio cada número (seção 6.7).

a Número do segmento;

b Número de uma página lógica dentro do segmento;

c Deslocamento dentro de uma página;

d Endereço lógico completo.