CAPÍTULO III

GERENCIAMENTO DE MEMÓRIA

Trataremos agora da parte do S.O. conhecida como gerenciador de memória, que é o responsável por cuidar de quais partes da memória estão em uso, quais estão livres, alocar memória a processos quando eles precisam, desalocar quando eles não mais necessitarem e gerenciar a troca (swapping) dos processos entre memória principal e disco, quando a memória principal não é suficientemente grande para manter todos os processos.

1. GERENCIAMENTO DE MEMÓRIA SEM TROCA OU PAGINAÇÃO

Troca e paginação são métodos utilizados de movimentação da memória para o disco e vice-versa durante a execução dos processos. Entretanto, antes de estudar esses métodos, estudaremos o caso mais simples em que não é realizada nem troca nem paginação.

2. MONOPROGRAMAÇÃO SEM TROCA OU PAGINAÇÃO

Neste caso, temos um único processo executando por vez, de forma que o mesmo pode utilizar toda a memória disponível, com exceção da parte reservada ao sistema operacional, que permanece constantemente em local pré-determinado da memória. O S.O. carrega um programa do disco para a memória, executa-o e, em seguida, aguarda comandos do usuário para carregar um novo programa, que se sobrepor ao anterior.

3. MULTIPROGRAMAÇÃO

Apesar de largamente utilizada em microcomputadores, a monoprogramação praticamente não é mais utilizada em sistemas grandes. Algumas razões para isso são as seguintes: - Muitas aplicações são mais facilmente programáveis quando as dividimos em dois ou mais processos.

- Os grandes computadores em geral oferecem serviços interativos simultaneamente para diversos usuários. Neste caso, é impossível de se trabalhar com um único processo na memória por vez, pois isso representaria grande sobrecarga devido à constante necessidade de chavear de um processo para outro (o que, com apenas um processo por vez na memória representaria constantemente estar lendo e escrevendo no disco).
- É necessário que diversos processos estejam "simultaneamente" em execução, devido ao fato de que muitos deles estão constantemente realizando operações de E/S, o que implica em grandes esperas, nas quais, por questão de eficiência, a UCP deve ser entregue a outro processo.

3.1 MULTIPROGRAMAÇÃO COM PARTIÇÕES FIXAS

Já que percebemos a importância da existência de diversos processos na memória, devemos agora analisar de que forma este objetivo pode ser conseguido. A forma mais simples é dividir a memória existente em n partições fixas, possivelmente diferentes. Essas partições poderiam ser criadas, por exemplo, pelo operador, ao inicializar o sistema.

Uma forma de tratar com essas partições seria então a seguinte:

i. cria-se uma fila para cada partição existente;
ii. cada vez que um processo é iniciado, ele é colocado na fila da menor partição capaz de o executar;
iii. os processo em cada partição são escolhidos de acordo com alguma forma de política (p.ex.: o primeiro a chegar é atendido antes).

Este método pode ser simbolizado como na Figura 1 :

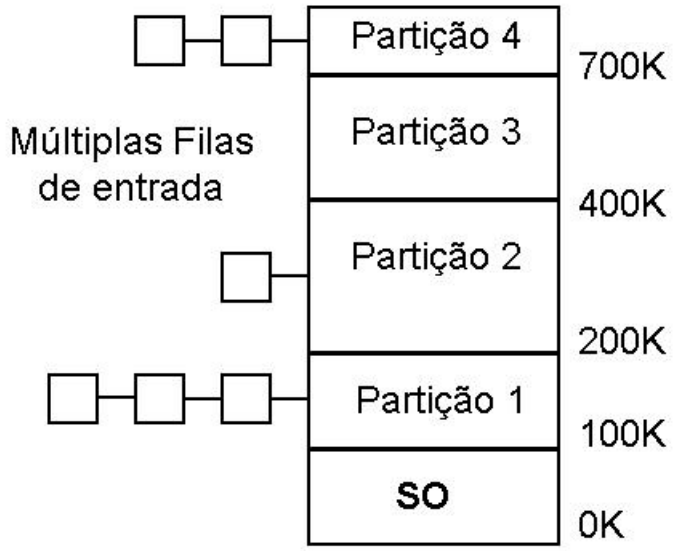

Figura 1. Esquema de filas Múltiplas

Uma desvantagem óbvia desse esquema é a de que pode ocorrer que uma partição grande esteja sem utilização, enquanto que diversos processos estão aguardando para utilizar uma partição menor (p.ex.: na figura acima temos três processos aguardando pela partição 1, enquanto que a partição 3 está desocupada).

Podemos resolver esse problema da seguinte forma:

i. estabelecemos apenas uma fila para todas as partições;
ii. quando uma partição fica livre, um novo processo que caiba na partição livre é escolhido e colocado na mesma, conforme indicado na Figura 2.

O problema que surge aqui é a forma de escolha implícita no item (ii). Se a partição livre for entregue para o primeiro processo da fila, pode ocorrer de que uma partição grande seja entregue a um processo pequeno, o que não é desejável (pois implica desperdício de memória). Por outro lado, se percorremos a fila procurando o maior processo aguardando que caiba na partição, isto representará servir melhor os processos grandes em detrimento dos pequenos.

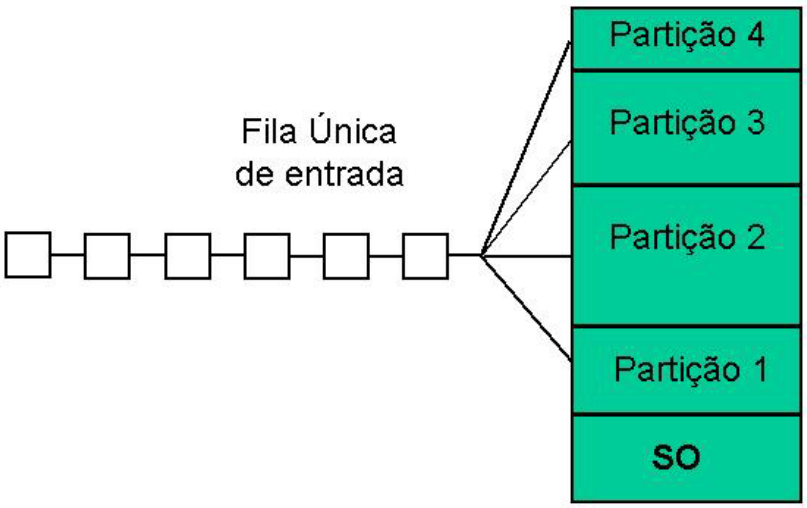

Figura 2. Esquema de filas Única

3.2 RELOCAÇÃO E PROTEÇÃO

Relocação e proteção são dois problemas introduzidos pela multiprogramação, e que precisam ser resolvidos. A necessidade da relocação pode ser vista nas figuras anteriores, onde fica claro que processos diferentes executam em posições de memória diferentes e, portanto, com endereços diferentes.

Suponha uma rotina em um programa que, ao término da ligação dos módulos deste, é colocada na posição 100 em relação ao início do programa. É claro que se esse programa for ser executado na partição 1, todas as chamadas dessa rotina devem ser enviadas para a posição de memória 100k+100. Se o programa for executado na partição 2, as chamadas para essa mesma rotina devem ser enviadas para a posição 200k+100.

Uma possível solução é modificar as instruções conforme o programa é carregado na memória. Desta forma, quando o S.O. carrega o programa, adiciona a todas as instruções que se referenciem a endereços, o valor do ponto inicial de carga do programa (i.e., o início da partição escolhida para o mesmo). Esta solução exige que o ligador (linker) coloque no início do código binário do programa uma tabela (ou uma lista) que apresente as indicações das posições no programa que devem ser modificadas no carregamento.

Esta solução entretanto não resolve o problema da proteção, pois nada impede que um programa errado ou malicioso leia ou altere posições de memória de outros usuários (dado que as referências são sempre as posições absolutas de memória).

Uma solução para isto adotada no IBM 360 foi a de dividir a memória em unidades de 2K bytes e associar um código de proteção de 4 bits a cada uma dessas regiões. Durante a execução de um processo, o PSW contém um código de 4 bits, que é testado com todos os acessos à memória realizados pelo processo, e gera uma interrupção se tentar acessar uma região de código diferente.

Uma solução alternativa tanto para o problema da relocação como para o da proteção é a utilização de registradores de base e limite. Sempre que um processo é carregado na memória, o S.O. ajusta o valor do registrador de base de acordo com a disponibilidade de memória. Este registrador é utilizado de forma que, sempre que um acesso é realizado na memória pelo processo, o valor do registrador de base é automaticamente somado, o que faz com que o código do programa em si não precise ser modificado durante o

carregamento. O registrador de limite é utilizado para determinar o espaço de memória que o processo pode executar. Todo acesso realizado pelo processo à memória é testado com o valor do registrador limite, para verificar se esse acesso está se realizando dentro do espaço reservado ao processo (caso em que ele é válido) ou fora do seu espaço (acesso inválido). Uma vantagem adicional do método dos registradores base e limite é o de permitir que um programa seja movido na memória, mesmo após já estar em execução. No método anterior isto não era possível sem realizar todo o processo de alteração dos endereços novamente.

3.3 TROCA (swapping)

Num sistema em batch, desde que se mantenha a UCP ocupada o máximo de tempo possível, não existe razão para se complicar o método de gerenciamento da memória. Entretanto, num sistema de time-sharing, onde muitas vezes existe menos memória do que o necessário para manter todos os processos de usuário, então é preciso que uma parte dos processos sejam temporariamente mantidos em disco. Para executar processos que estão no disco, eles devem ser enviados para a memória, o que significa retirar algum que lá estava.
Este processo é denominado troca (swapping).

3.4 MULTIPROGRAMAÇÃO COM PARTIÇÕES VARIÁVEIS

A utilização de partições fixas para um sistema com swapping é ineficiente, pois implicaria em que grande parte do tempo as partições estariam sendo utilizadas por processos muito pequenos para as mesmas.
A solução para isso é tratarmos com partições variáveis, isto é, partições que variam conforme as necessidades dos processos, tanto em tamanho, como em localização, como em número de partições. Isto melhora a utilização de memória mas também complica a alocação e desalocação de memória.
É possível combinar todos os buracos formados na memória em um único, o que é conhecido como compactação de memória, mas isto é raramente utilizado, devido à grande utilização de UCP requerida.
Um ponto importante a considerar é quanta memória deve ser alocada a um processo, quando o mesmo é iniciado. Se os processos necessitam de uma certa quantidade pré-fixada e invariante de memória, então basta alocar a quantidade necessária para cada processo ativo e o problema está resolvido. Entretanto, muitas vezes os processos necessitam de mais memória durante o processamento (alocação dinâmica de memória). Neste caso, quando um processo necessita de mais memória, podem ocorrer dois casos:

a. existe um buraco (região não ocupada) de memória próximo ao processo: neste caso, basta alocar memória desse buraco;
b. o processo está cercado por outros processos, ou o buraco que existe não é suficiente, neste caso, podemos tomar alguma das seguintes ações:

- mover o processo para um buraco de memória maior (grande o suficiente para as novas exigências do processo);
- se não houver tal espaço, alguns processos devem ser retirados da memória, para deixar espaço para esse processo;
- se não houver espaço no disco para outros processos, o processo que pediu mais espaço na memória deve ser morto (**kill**).

Quando se espera que diversos processos cresçam durante a execução, pode ser uma boa idéia reservar espaço extra para esses processos quando os mesmos são criados, para eliminar a sobrecarga de lidar com movimentação ou troca de processos.
Trataremos agora de duas das formas principais de cuidar da utilização de memória: mapas de bits, listas e sistemas de desabrochamento.

3.5 GERENCIAMENTO DE ESPAÇO

3.5.1 GERENCIAMENTO COM MAPA DE BITS

Este método consiste em formar um mapa de bits. A memória é subdividida em unidades de um certo tamanho. A cada uma dessas unidades é associado um bit que, se for 0 indica que essa parte da memória está livre, e se for 1, indica que ela está ocupada. O tamanho da unidade que é associada com um bit deve ser cuidadosamente escolhida, entretanto, mesmo com a associação de apenas 4 bytes de memória para cada bit no mapa, a parte da memória gasta com esse mapa é de apenas 3%. Se escolhemos mais de 4 bytes, o espaço ocupado pela tabela será menor, mas o desperdício de memória crescerá . A Figura 3 mostra um trecho de memória e o mapa de bits associado.

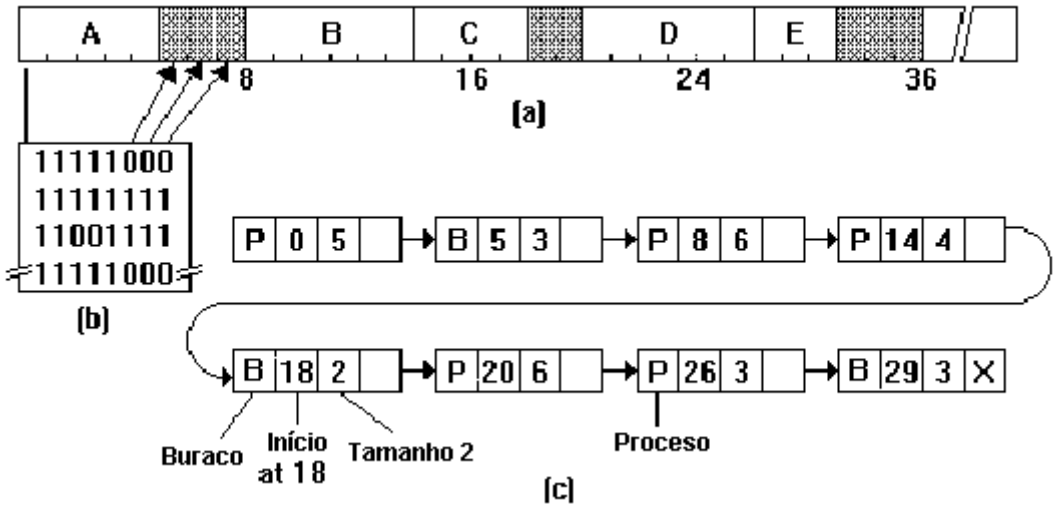

Figura 3. Mapa de bits associado.

A desvantagem desse método, é que, quando um novo processo que ocupa k unidades de memória deve ser carregado na memória, o gerenciador deve percorrer o mapa de bits para encontrar k bits iguais a zero consecutivos, o que não é um processo simples.

3.5.2 GERENCIAMENTO COM LISTAS ENCADEADAS

Neste caso, mantemos uma lista encadeada de segmentos alocados e livres, sendo que cada segmento é um processo ou um buraco entre dois processos. Na Figura 5, anteriormente apresentada, temos também a lista associada ao trecho de memória indicado. B indica um buraco e P um processo. A lista apresenta-se em ordem de endereços, de forma que quando um processo termina ou é enviado para o disco, a atualização da lista é simples: cada processo, desde que não seja nem o primeiro nem o último da lista, apresenta-se cercado por dois segmentos, que podem ser buracos ou outros processos, o que nos dá as quatro possibilidades mostradas na Figura 4:

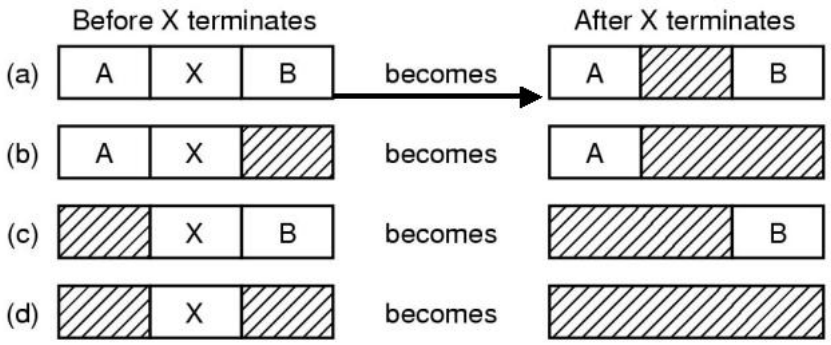

Figura 4. As quatro combinações para terminar a conclusão de um processo.

Note que os buracos adjacentes devem ser combinados num único.
Vejamos agora alguns algoritmos que podem ser utilizados para escolher o ponto em que deve ser carregado um processo recém criado ou que veio do disco por uma troca (foi swapped in). Assumimos que o gerenciador de memória sabe quanto espaço alocar ao processo:

1. **First Fit (primeiro encaixe)**: neste caso, o algoritmo consiste em percorrer a fila até encontrar o primeiro espaço em que caiba o processo. É um algoritmo rápido;
2. **Next Fit (próximo encaixe)**: o mesmo que o algoritmo anterior, só que ao invés de procurar sempre a partir do início da lista, procura a partir do último ponto em que encontrou. Simulações mostraram que esse algoritmo apresenta um desempenho ligeiramente melhor que o anterior;
3. **Best Fit (melhor encaixe)**: consiste em verificar toda a lista e procurar o buraco que tiver espaço mais próximo das necessidades do processo. É um algoritmo mais lento e, além disso, simulações demonstram que tem um desempenho pior que o First Fit, devido ao fato de tender a encher a memória com pequenos buracos sem nenhuma utilidade;
4. **Worst Fit (pior ajuste)**: sugerido pelo fracasso do algoritmo anterior, consiste em pegar sempre o maior buraco disponível. Simulações mostraram que também seu desempenho é ruim.

Os quatro algoritmos podem ter sua velocidade aumentada pela manutenção de duas listas separadas, uma para processos e outra para buracos. Ainda outro algoritmo possível, quando temos duas listas separadas, é o Quick Fit (ajuste rápido), que consiste em manter listas separadas para alguns dos tamanhos mais comuns especificados (por exemplo, uma fila para 2k, outra para 4k, outra para 8k, etc.). Neste caso, a busca de um buraco com o tamanho requerido é extremamente rápido, entretanto, quando um processo termina, a liberação de seu espaço é complicada, devido à necessidade de reagrupar os buracos e modificá-los de fila.

3.6 ALOCAÇÃO DE ESPAÇO DE TROCA (SWAP)

Chamamos de espaço de troca ao espaço ocupado no disco pelos processos que aí estão guardados pelo fato de que foram retirados da memória devido a uma troca (swap). Os algoritmos para gerenciar o espaço alocado em disco para swap são os mesmos apresentados para o gerenciamento da memória, com a diferença de que em alguns

sistemas, cada processo tem no disco um espaço reservado para o mesmo, de forma que não está, como na memória, sendo constantemente mudado de lugar.
Um fator adicional de diferença é o fato de que, pelos discos serem dispositivos de bloco, a quantidade de espaço reservado para os processos no disco deverão ser múltiplas do tamanho do bloco.

3.7 MEMÓRIA VIRTUAL

Quando os programas começaram a ficar grandes demais para a quantidade de memória necessária, a primeira solução adotada foi a de utilização de overlay. Nesta técnica, o programa era subdividido em partes menores, chamadas overlays, e que podiam ser rodadas separadamente. Quando um dos overlays terminava a execução, um outro poderia ser carregado na mesma posição de memória utilizada pelo anterior (isto existe ainda em linguagens como o Turbo-Pascal e o Aztec C).
O problema com este método é que todo o trabalho de divisão de programas em overlays, que aliás não é simples, deve ser realizado pelo programador.
A técnica de memória virtual é uma outra forma de executar um programa que não cabe na memória existente, mas que não apresenta os inconvenientes dos overlays, por ser realizada de forma automática pelo próprio computador. Neste caso, partes do programa, dos dados, e da pilha são mantidas no disco, sendo que existe uma forma de decisão cuidadosa de quais devem permanecer no disco e quais na memória. Da mesma forma, podemos alocar diversos processos na memória virtual, de forma que cada um pensa ter uma quantidade de memória que somadas ultrapassam a quantidade real de memória.

3.7.1 PAGINAÇÃO

Paginação é uma técnica muito utilizada em sistemas com memória virtual. Antes, estabeleçamos o conceito de espaço virtual.
Chamamos de espaço virtual ao espaço de memória que pode ser referenciado por um programa qualquer em dado processador. Por exemplo, um processador com endereçamento de 16 bits possui um espaço virtual de 64k bytes (se o endereçamento for em bytes). Quando uma instrução como:
LD A,(1000h); carrega acumulador com conteúdo do endereço 1000 (em hexadecimal) é apresentada no Z80, o 1000h corresponde a um endereço virtual, de um espaço de endereçamento virtual de 64k bytes.
Em um computador sem memória virtual, o endereço virtual corresponde ao endereço efetivamente colocado no duto de endereçamento da memória. No exemplo acima, seria colocado no duto de endereços o valor binário correspondente a 1000h.
Quando o computador possui memória virtual, esse endereço virtual é enviado para uma unidade de gerenciamento de memória, MMU (memory management unit), que corresponde a um chip ou um conjunto de chips que translada esse endereço virtual em um endereço físico, de acordo com uma tabela.
A operação da MMU pode ser explicada conforme apresentado na Figura 5:

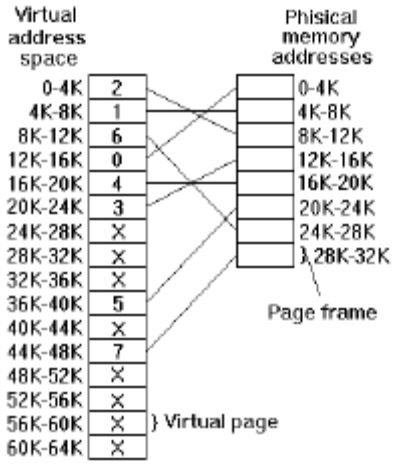

Figura 5. Relação entre o endereçamento da memória virtual e o endereçamento da memória física dado pela tabela de páginas.

O espaço de endereços virtuais é dividido em unidades chamadas páginas e o espaço de memória física é dividido em unidades chamadas quadros de página, de mesmo tamanho das páginas. A MMU tem uma tabela que indica, para cada página, qual o quadro de página que corresponde à mesma. No exemplo da Figura 5, temos um espaço virtual de 64k bytes, uma memória física de 32k bytes, e páginas de 4k bytes. Se o processador tenta acessar o endereço 0, a MMU verifica que isto corresponde ao primeiro endereço da primeira página,
verifica então que essa primeira página está alocada no terceiro quadro de página (i.e. o de número 2). Converte então esse endereço para 8192 (decimal) e envia esse endereço convertido para a memória. Note que nem o processador nem a memória precisaram ficar sabendo da existência de paginação.
No entanto, como nem todas as páginas do espaço virtual podem estar residentes na memória simultaneamente, ocorrer o caso de que um acesso seja realizado para uma página que não está na memória. Para saber isto, a MMU mantém na tabela de translação um bit para cada página que indica se a mesma está presente na memória ou não. Se um acesso for realizado a uma página ausente, é gerada uma falta de página (page fault), o que chama uma rotina de tratamento de interrupção específica para o S.O., que então se encarrega do carregamento da página faltante e o ajuste correspondente na tabela de translação.
A forma mais comum de implementação da MMU, é escolher alguns dos bits mais significativos do endereço virtual como indicadores do número de página e o restante dos bits como um deslocamento dentro dessa página. Por exemplo, na Figura 10, apresentada acima, de 16 bits do endereço virtual, 12 serão utilizados para o deslocamento (pois são necessários 12 bits para endereçar os 4k bytes de uma página), sendo os 4 restantes utilizados como um índice para qual das 16 páginas está sendo referenciada. A MMU portanto, pega os 4 bits do índice de página, acessa a posição correspondente da tabela de translação, verifica se a página está presente na memória, se não estiver, gera uma interrupção para carregamento e, depois, verifica o valor colocado nessa entrada da tabela de translação e os junta aos 12 bits de deslocamento dentro da página. A Figura 6 mostra a operação da MMU.

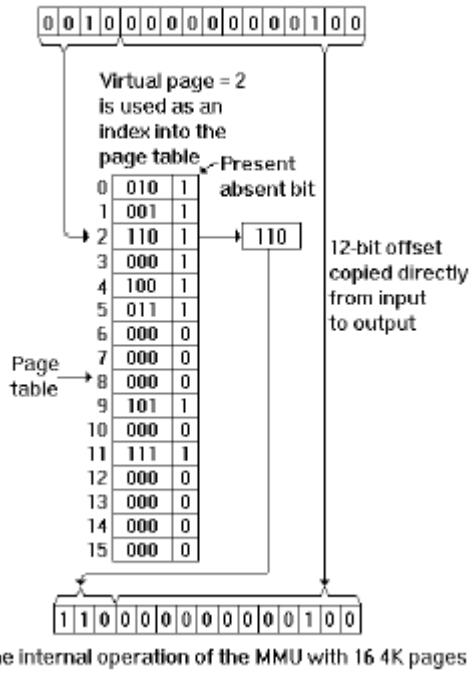

Figura 6. Operação interna da MMU com 16 4K páginas.

3.7.2 SEGMENTAÇÃO

A paginação fornece uma forma de se conseguir grandes espaços de endereçamento lineares em uma quantidade finita de memória física. Em algumas aplicações, no entanto, é preferível ter um espaço bidimensional. O espaço é bidimensional no sentido de que é dividido em um certo número de segmentos, cada um com dado número de bytes. Dessa forma, um endereçamento é sempre expresso da forma (segmento, deslocamento). Os diferentes segmentos são associados a diversos programas ou mesmo arquivos, de forma que neste caso, os arquivos podem ser acessados como se fossem posições de memória.

Veja que, diferentemente da paginação, na segmentação os programadores (ou os compiladores) levavam cuidadosamente em conta a segmentação, tentando colocar entidades diferentes em segmentos diferentes. Essa estratégia facilitou o compartilhamento de objetos entre processos diferentes.

A Figura 7 mostra uma implementação comum de segmentação em microcomputadores com o processador 68000.

Note que o hardware suporta até 16 processos, cada um com 1024 páginas de 4k bytes cada (isto é, cada processo com um endereço virtual de 4M bytes). A cada um dos 16 processos, a MMU associa uma seção com 64 descritores de segmento, sendo que cada descritor contém até 16 páginas. O descritor do segmento contém o tamanho do segmento (1 a 16 páginas), bits que indicam a proteção associada com o segmento e um ponteiro para uma tabela de páginas.

Note também que para trocar de processo, a única coisa que o S.O. precisa fazer é alterar um registrador de 4 bits e isso automaticamente muda os acessos às tabelas.

Deve-se enfatizar que este é apenas um exemplo de esquema de segmentação e não o único possível.

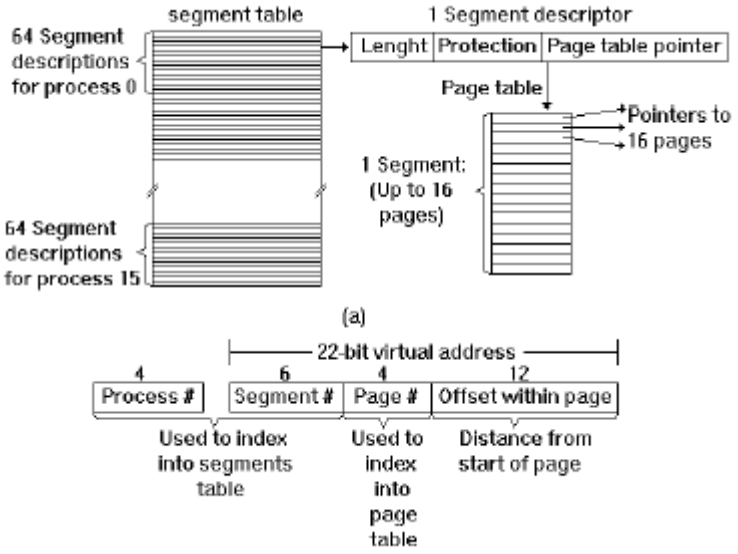

Figura 7. Implementação comum de segmentação em microcomputadores com o processador 68000.