GERÊNCIA DE MEMÓRIA

1. INTRODUÇÃO

Historicamente, a memória principal sempre foi vista como um recurso escasso e caro. Uma das maiores preocupações dos projetistas foi desenvolver sistemas operacionais que não ocupassem muito espaço de memória e, ao mesmo tempo, otimizassem a utilização dos recursos computacionais. Mesmo atualmente, com a redução de custo e conseqüente aumento da capacidade da memória principal, seu gerenciamento é um dos fatores mais importantes no projeto de sistemas operacionais.

Enquanto nos sistemas monoprogramáveis a gerência da memória não é muito complexa, nos sistemas multiprogramáveis essa gerência se torna crítica, devido à necessidade de se maximizar o número de usuários e aplicações utilizando eficientemente o espaço da memória principal.

2. FUNÇÕS BÁSICAS

Em geral, programas são armazenados em memórias secundárias, como disco ou fita, por ser um meio não-volátil, abundante e de baixo custo. Como o processador somente executa instruções localizadas na memória principal, o sistema operacional deve sempre transferir programas da memória secundária para a memória principal antes de serem executados. Como o tempo de acesso à memória secundária é muito superior ao tempo de acesso à memória principal, o sistema operacional deve buscar reduzir o número de operações de E/S à memória secundária, caso contrário, sérios problemas no desempenho do sistema podem ser ocasionados.

Os sistemas de gerência de memória podem ser divididos em duas grandes categorias: aqueles que movem os processos entre a memória principal e o disco (swapping e paginação), e aqueles que não movimentam os processos entre tais dispositivos de armazenamento. Os últimos são bem mais simples que os primeiros.

A gerência de memória deve tentar manter na memória principal o maior número possível de processos residentes, permitindo maximizar o compartilhamento do processador e demais recursos computacionais. Mesmo na ausência de espaço livre, o sistema deve permitir que novos processos sejam aceitos e executados. Isso é possível através da transferência temporária de processos residentes na memória principal para a memória secundária, liberando espaço para novos processos. Este mecanismo é conhecido como *swapping* e será detalhado posteriormente.

Outra preocupação na gerência de memória é permitir a execução de programas que sejam maiores que a memória física disponível, implementado através de técnicas como *overlay* e *memória virtual*.

Em um ambiente de multiprogramação, o sistema operacional deve proteger as áreas de memória ocupadas por cada processo, além da área onde reside o próprio sistema. Caso um programa tente realizar algum acesso indevido à memória, o sistema de alguma forma deve impedi-lo. Apesar da gerência de memória garantir a proteção de áreas da memória, mecanismos de compartilhamento devem ser oferecidos para que diferentes processos possam trocar dados de forma protegida.

3. ALOCAÇÃO CONTÍGUA SIMPLES

A *alocação contígua simples* foi implementada nos primeiros sistemas operacionais, porém ainda está presente em alguns sistemas monoprogramáveis. Nesse tipo de organização, a memória principal é subdividida em duas áreas: uma para o sistema operacional e outra para o programa do usuário.

Figura 1: Alocação contígua simples

Dessa forma, o programador deve desenvolver suas aplicações, preocupado, apenas, em não ultrapassar o espaço de memória disponível, ou seja, a diferença entre o tamanho total da memória principal e a área ocupada pelo sistema operacional.

Nesse esquema, o usuário tem controle sobre toda a memória principal, podendo ter acesso a qualquer posição de memória, inclusive a área do sistema operacional. Para proteger o sistema desse tipo de acesso, que pode ser intencional ou não, alguns sistemas implementam proteção através de um registrador que delimita as áreas do sistema operacional e do usuário.

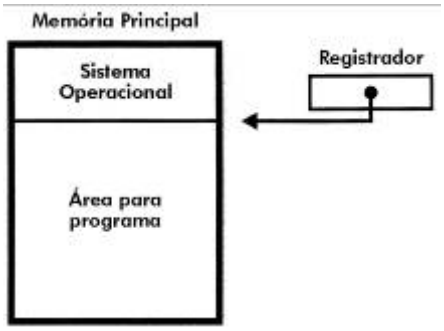

Figura 2: Proteção na alocação contígua simples

Dessa forma, sempre que um programa faz referência a um endereço na memória, o sistema verifica se o endereço está dentro dos limites permitidos. Caso não esteja, o programa é cancelado e uma mensagem de erro é gerada, indicando que houve uma violação no acesso à memória principal.

Apesar de fácil implementação e do código reduzido, a alocação contígua simples não permite a utilização eficiente dos recursos computacionais, pois apenas um usuário pode dispor desses recursos. Em relação à memória principal, caso o programa do usuário não a preencha totalmente, existirá um espaço de memória livre sem utilização.

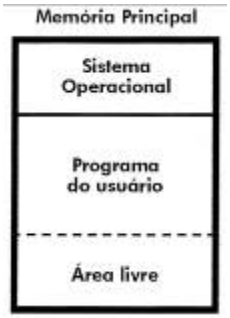

Figura 3: Subutilização da memória principal

4. TÉCNICA DE OVERLAY

Na alocação contígua simples, todos os programas estão limitados ao tamanho da área de memória principal disponível para o usuário. Uma solução encontrada para o problema é dividir o programa em módulos, de forma que seja possível a execução independente de cada módulo, utilizando uma mesma área de memória. Essa técnica é chamada de *overlay*.

Considere um programa que tenha três módulos: um principal, um de cadastramento e outro de impressão, sendo os módulos de cadastramento e de impressão independentes. A independência do código significa que quando um módulo estiver na memória para execução, o outro não precisa necessariamente estar presente. O módulo

principal é comum aos dois módulos; logo, deve permanecer na memória durante todo o tempo da execução do programa.

Como podemos verificar na figura 4, a memória é insuficiente para armazenar todo o programa, que totaliza 9kb. A técnica de overlay utiliza uma área de memória comum, onde os módulos de cadastramento e de impressão poderão compartilhar a mesma área de memória (área de overlay). Sempre que um dos dois módulos for referenciado pelo módulo principal, o módulo será carregado da memória secundária para a área de overlay. No caso de uma referência a um módulo que já esteja na área de overlay, a carga não é realizada; caso contrário, o novo módulo irá sobrepor-se ao que já estiver na memória principal.

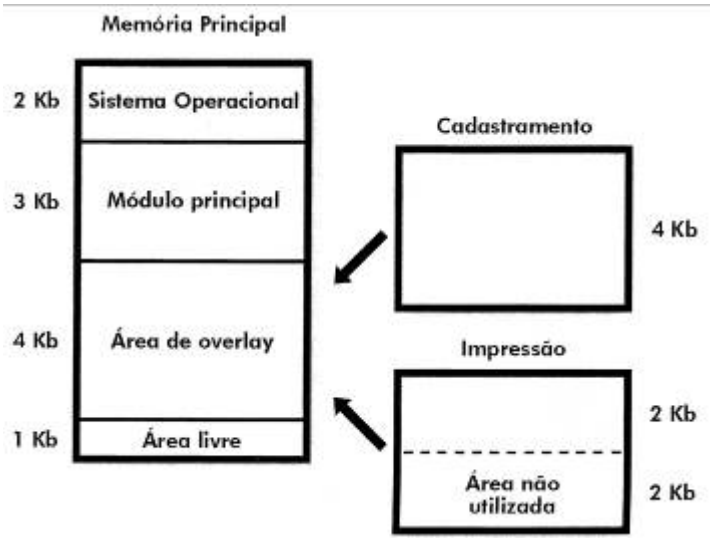

Figura 4: Técnica de overlay

A definição das áreas de overlay é função do programador, através de comandos específicos da linguagem de programação utilizada. O tamanho de uma área de overlay é estabelecido a partir do tamanho do maior módulo.

A técnica de overlay tem a vantagem de permitir ao programador expandir os limites da memória principal. A utilização dessa técnica exige muito cuidado, pois pode trazer implicações tanto na sua manutenção quanto no desempenho das aplicações, devido à possibilidade de transferência excessiva dos módulos entre a memória principal e a secundária.

5. ALOCAÇÃO PARTICIONADA

Os sistemas operacionais evoluíram no sentido de proporcionar melhor aproveitamento dos recursos disponíveis. Nos sistemas monoprogramáveis, o processador permanece grande parte do tempo ocioso e a memória principal é

subutilizada. Os sistemas multiprogramáveis já são muito mais eficientes no uso do processador, necessitando, assim, que diversos programas estejam na memória principal ao mesmo tempo e que novas formas de gerência da memória sejam implementadas.

O Modelo da Multiprogramação

Quando optamos por multiprogramar um processador, sua utilização poderá ser bastante otimizada. Colocando a questão de uma forma bem grosseira, se um processo gasta em média 20% do tempo em que está armazenado na memória principal usando efetivamente o processador, com cinco processos ao mesmo tempo na memória, o processador ficará ocupado o tempo todo. Este modelo é muito otimista e irreal, uma vez que assume que nunca dois ou mais processos estarão esperando por entrada/saída ao mesmo tempo.

Um modelo mais realista procura considerar o uso do processador de forma probabilística. Suponha que determinado processo gaste uma fração *p* de seu tempo, esperando por entrada/saída. Com *n* processos na memória ao mesmo tempo, a probabilidade de todos eles aguardarem por entrada/saída ao mesmo tempo, caso em que o processador estará ocioso, é p^n. Neste caso, o uso do processador será dado pela fórmula:

$$\text{uso do processador} = 1 - p^n$$

A figura 5 mostra a utilização do processador como uma função de *n*, função esta denominada de **grau de multiprogramação**.

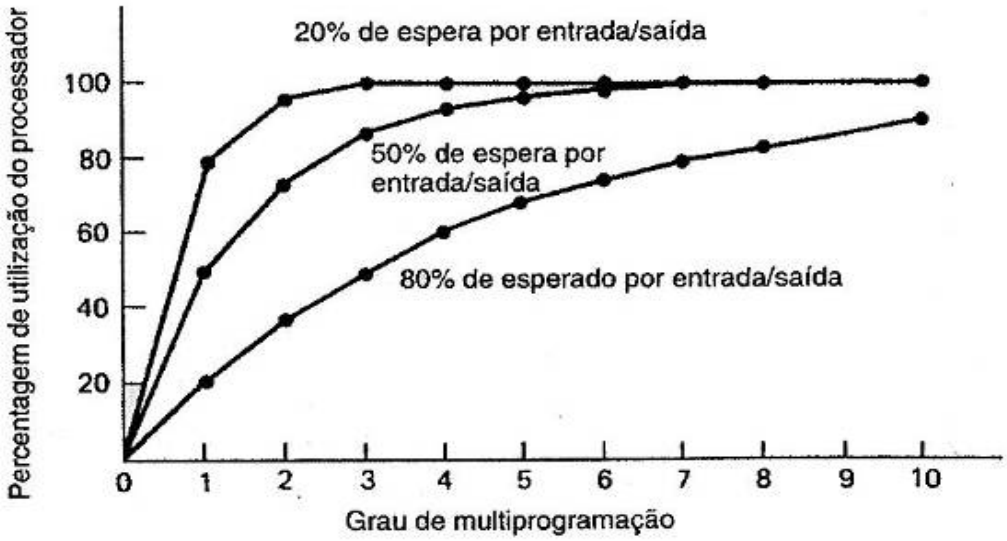

Figura 5: Utilização do processador como função do número de processos na memória principal

A análise da figura 5 mostra que se os processos gastam 80% de seu tempo esperando por entrada/saída, no mínimo 10 processos devem estar ao mesmo tempo na memória, para manter o grau de ociosidade abaixo de 10%. Este modelo probabilístico é apenas uma aproximação, pois assume que todos os n processos são independentes, significando que é aceitável para um sistema com cinco processos na memória ter três deles rodando e dois esperando. Mas, com um único processador, não podemos ter três processos rodando de uma vez, de forma que um processo que fique pronto enquanto o processador está ocupado terá que esperar. Então, tais processos não são independentes. Um modelo mais aperfeiçoado poderia ser construído usando a teoria das filas.

Mesmo sendo um modelo muito simples, o modelo da figura acima pode ser usado para fazer previsões a respeito da performance do processador. Suponha, por exemplo, que um computador tenha 1M de memória, com o sistema operacional ocupando 200K, e cada programa do usuário ocupando também 200K. Nestas condições, podemos ter quatro programas de usuário na memória ao mesmo tempo. Com 80% de percentagem de espera por entrada/saída, temos uma utilização do processador em torno de 60%, ignorando o overhead do sistema operacional. A adição de outro megabyte de memória permite que o sistema saia de um grau de multiprogramação 4 para um grau 9, fazendo a percentagem de utilização chegar a 87% aproximadamente. Em outras palavras, o segundo megabyte de memória fará o throughput crescer 45%. Já um terceiro megabyte fará com que a utilização do processador cresça de 87% para 96%, melhorando o throughput em apenas 10% aproximadamente. Usando este modelo, o analista deverá decidir que o segundo megabyte é um ótimo investimento, mas o terceiro não.

Análise da Performance do Sistema sob Multiprogramação

O modelo acima também pode ser usado para analisar sistemas batch. Consideremos um centro de computação onde os jobs esperam em média 80% do tempo por entrada/saída. Quatro jobs são submetidos conforme a tabela abaixo:

Job	Instante da chegada	Minutos de processador necessários
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

O primeiro job precisa de quatro minutos do processador. Com 80% de espera por entrada/saída, tal job gasta apenas 12 segundos de tempo do processador para cada

minuto em que estiver na memória, mesmo que não haja outros jobs competindo com ele pelo uso do processador. Os outros 48 segundos são gastos aguardando que operações de entrada/saída se completem. Desta forma, ele precisará estar na memória por no mínimo 20 minutos, de forma a conseguir quatro minutos de processador, mesmo com ausência de competição.

Das 10:00 às 10:10, o job 1 esteve sozinho na memória e conseguiu 2 minutos de tempo de processador. Com a chegada do job 2 , a utilização do processador cresce de 0,20 para 0,36 devido ao aumento do grau de programação. No entanto, com o escalonamento round robin, cada job obtém 0,18 minuto de trabalho do processador para cada minuto que permanecer na memória. Com a adição do segundo job, custou ao primeiro uma diminuição de 10% de sua performance sem competição.

Ás 10:15 chega o terceiro job. Até este momento os jobs 1 e 2 receberam respectivamente 2,0 e 0,9 minutos de tempo de processador. Com grau de multiprogramação 3, cada job obtém 0,16 minuto do processador para cada minuto decorrido de tempo real, conforme a tabela abaixo:

	1	2	3	4
Processador ocioso	0,80	0,64	0,51	0,41
Processador ocupado	0,20	0,36	0,49	0,59
Processador / processo	0,20	0,18	0,16	0,15

Das 10:15 às 10:20, cada um dos três obtém 0,8 minuto do processador. Às 10:20 chega o quarto job. A figura 6 mostra a seqüência completa dos eventos:

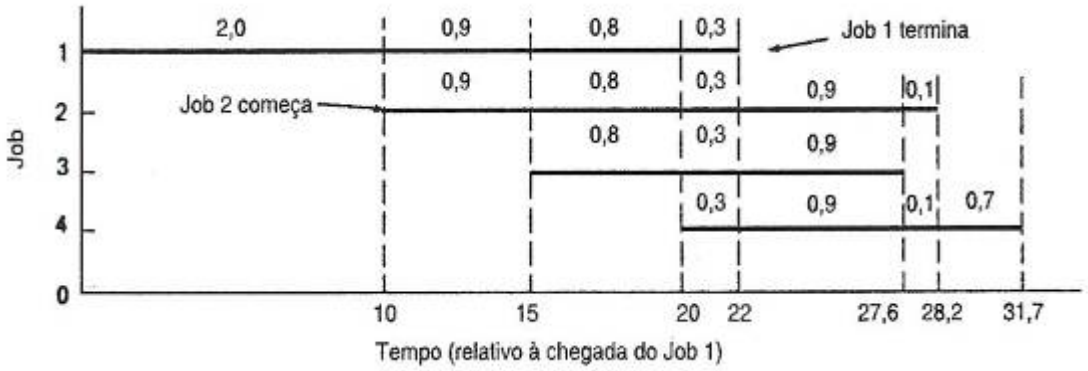

Figura 6:Seqüência de chegadas e término de jobs

Alocação Particionada Estática

Nos primeiros sistemas multiprogramáveis, a memória era dividida em pedaços de tamanho fixo, chamados *partições*. O tamanho das partições, estabelecido na fase de inicialização do sistema, era definido em função do tamanho dos programas que executariam no ambiente. Sempre que fosse necessária a alteração do tamanho de uma

partição, o sistema deveria ser desativado e reinicializado com uma nova configuração. Esse tipo de gerência de memória é conhecido como *alocação particionada estática* ou *fixa*.

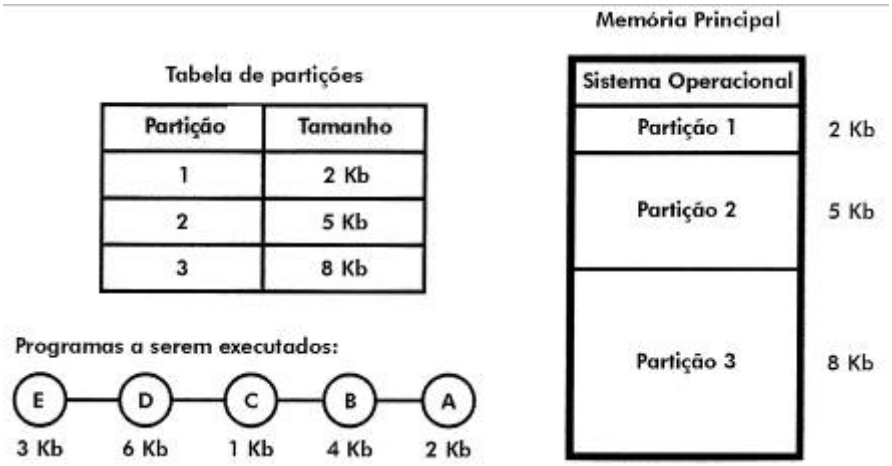

Figura 7: Alocação particionada estática

Inicialmente, os programas só podiam ser carregados e executados em apenas uma partição específica, mesmo se outras estivessem disponíveis. Essa limitação se devia aos compiladores e montadores, que geravam apenas códigos absolutos. No *código absoluto*, todas as referências a endereços no programa são posições físicas na memória principal, ou seja, o programa só poderia ser carregado a partir do endereço de memória especificado no seu próprio código. Se, por exemplo, os programas A e B estivessem sendo executados, e a terceira partição estivesse livre,os programas C e E não poderiam ser processados. A esse tipo de gerência de memória chamou-se *alocação particionada estática absoluta*.

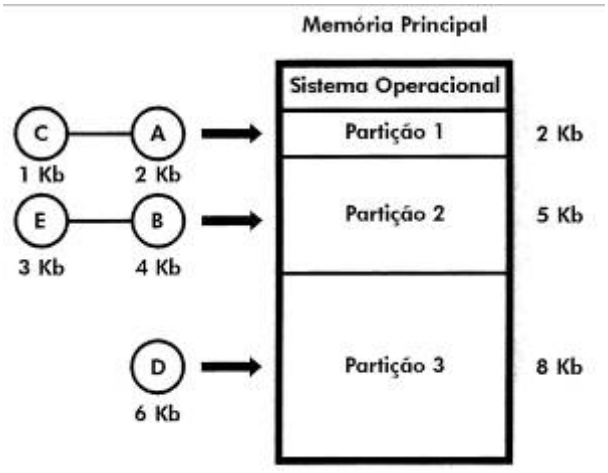

Figura 8: Alocação particionada estática absoluta

Com a evolução dos compiladores, montadores, linkers e loaders, o código gerado deixou de ser absoluto e passa a ser relocável. No *código relocável*, todas as referências a endereços no programa são relativas ao início do código e não a endereços físicos de memória. Desta forma, os programas puderam ser executados a partir de qualquer partição. Quando o programa é carregado, o loader calcula todos os endereços a partir da posição inicial onde o programa foi alocado. Caso os programas A e B terminassem, o programa E poderia ser executado em qualquer uma das duas partições. Esse tipo de gerência de memória é denominado *alocação particionada estática relocável*.

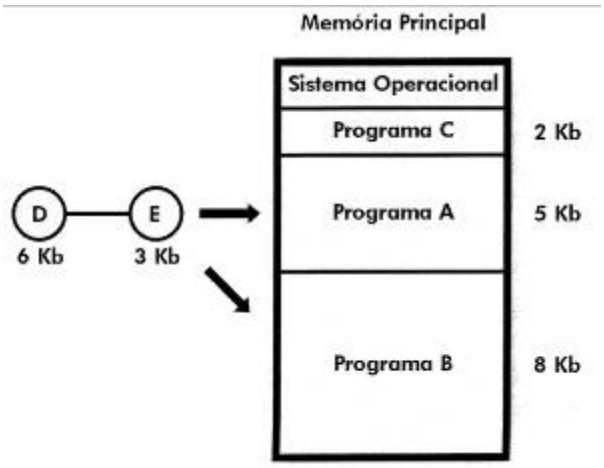

Figura 9: Alocação particionada estática relocável

Para manter o controle sobre quais partições estão alocadas, a gerência de memória mantém uma tabela com o endereço inicial de cada partição, seu tamanho, e se está em uso. Sempre que um programa é carregado para a memória, o sistema percorre a tabela, na tentativa de localizar uma partição livre, onde o programa possa ser carregado.

Partição	Tamanho	Livre
1	2 Kb	Não
2	5 Kb	Sim
3	8 Kb	Não

Figura 10: Tabela de alocação de partições

Nesse esquema de alocação de memória, a proteção baseia-se em dois registradores, que indicam os limites inferior e superior da partição onde o programa está sendo executado. Caso o programa tente acessar uma posição de memória fora dos limites definidos pelos registradores, ele é interrompido e uma mensagem de violação de acesso é gerada pelo sistema operacional.

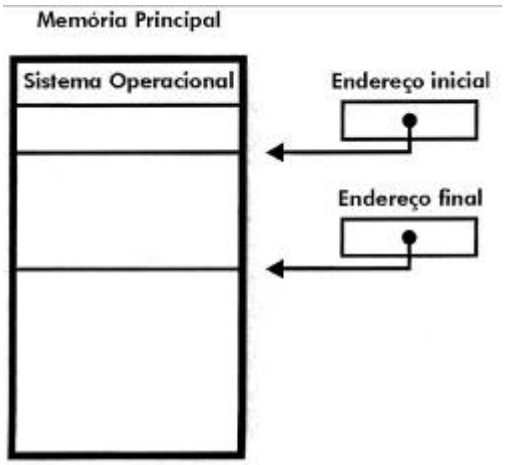

Figura 11: Proteção na alocação particionada

Tanto nos sistemas de alocação absoluta quanto nos de alocação relocável, os programas, normalmente, não preenchem totalmente as partições onde são carregados. Por exemplo, os programas C, A e E não ocupam integralmente o espaço das partições onde estão alocados, deixando 1kb, 3kb e 5kb de áreas livres, respectivamente, conforme a figura 10. Este tipo de problema, decorrente da alocação fixa das partições, é conhecido como *fragmentação interna*.

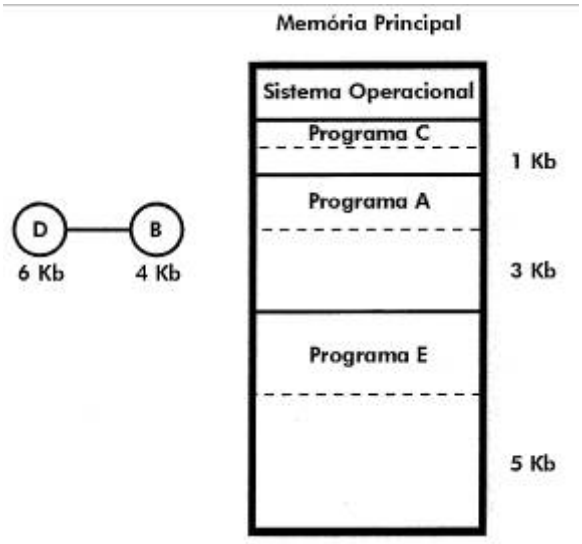

Figura 12: Fragmentação interna

Um exemplo de sistema operacional que implementou esse tipo de gerência de memória é o OS/MFT da IBM.

Alocação Particionada Dinâmica

A alocação particionada estática, analisada anteriormente, deixou evidente a necessidade de uma nova forma de gerência da memória principal, onde o problema da fragmentação interna fosse reduzido e, conseqüentemente, o grau de compartilhamento da memória aumentado.

Na *alocação particionada dinâmica* ou *variável*, foi eliminado o conceito de partições de tamanho fixo. Nesse esquema, cada programa utilizaria o espaço necessário, tornando essa área sua partição. Como os programas utilizam apenas o espaço de que necessitam, no esquema de alocação particionada dinâmica o problema da fragmentação interna não ocorre.

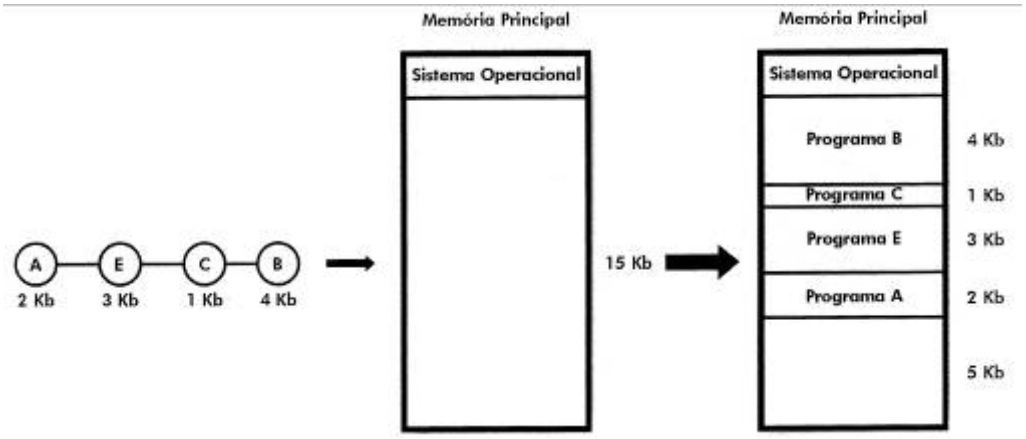

Figura 13: Alocação particionada dinâmica

A princípio, o problema da fragmentação interna está resolvido, porém, nesse caso, existe um problema que não é tão óbvio quanto no esquema anterior. Um tipo diferente de fragmentação começará a ocorrer, quando os programas forem terminando e deixando espaços cada vez menores na memória, não permitindo o ingresso de novos programas. No caso da figura 14, mesmo existindo 12kb livres de memória principal, o programa D, que necessita de 6kb de espaço, não poderá ser carregado para execução, pois este espaço não está disposto contiguamente. Esse tipo de problema é chamado de *fragmentação externa*.

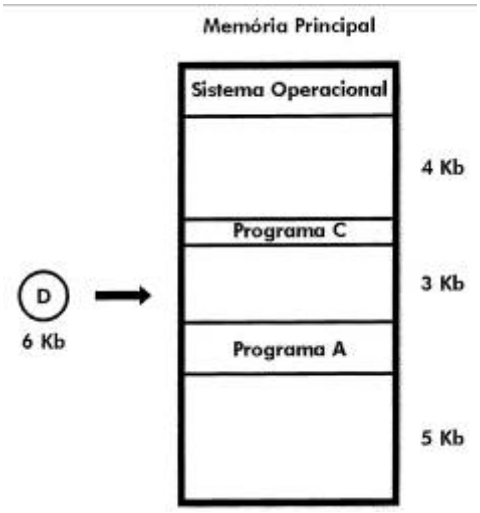

Figura 14: Fragmentação externa

Existem duas soluções para o problema da fragmentação externa da memória principal. Na primeira solução, conforme os programas terminam, apenas os espaços livres adjacentes são reunidos, produzindo áreas livres de tamanho maior. Na figura 15, caso o programa C termine, uma área de 8kb será criada.

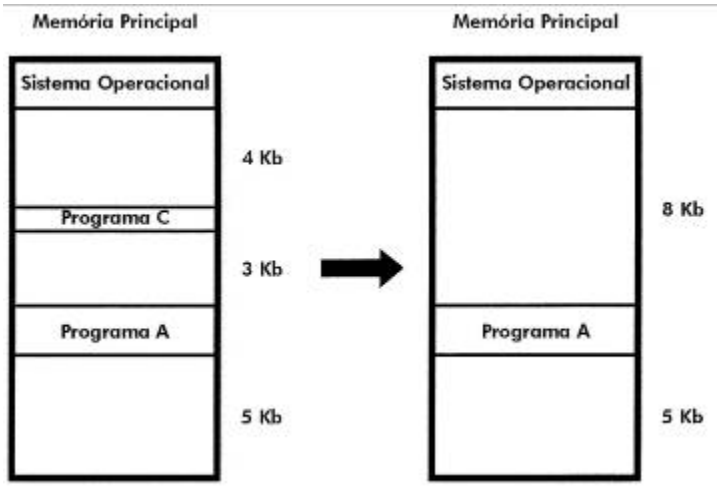

Figura 15: Solução para a fragmentação externa (a)

A segunda solução envolve a relocação de todas as partições ocupadas, eliminado todos os espaços entre elas e criando uma única área livre contígua (Fig. 16). Para que esse processo seja possível, é necessário que o sistema tenha a capacidade de mover os diversos programas na memória principal, ou seja, realizar *relocação dinâmica*. Esse mecanismo de compactação, também conhecido como *alocação particionada dinâmica com relocação*, reduz em muito o problema da fragmentação, porém a complexidade do seu algoritmo e o consumo de recursos do sistema, como processador e área em disco, podem torná-lo inviável.

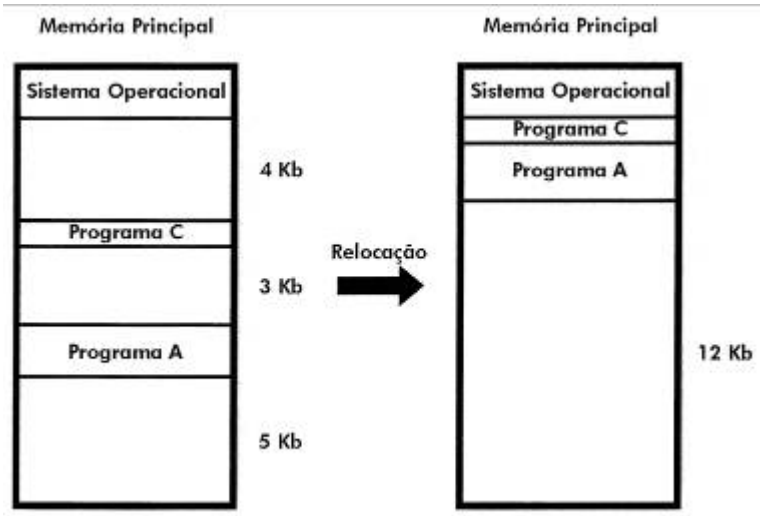

Figura 16: Solução para a fragmentação externa (b)

Um exemplo de sistema operacional que implementou esse tipo de gerência de memória é o OS/MVT da IBM.

Estratégias de Alocação de Partição

Os sistemas operacionais implementam, basicamente, três estratégias para determinar em qual área livre um programa será carregado para execução. Essas estratégias tentam evitar ou diminuir o problema da fragmentação externa.

A melhor estratégia a ser adotada por um sistema depende de uma série de fatores, sendo o mais importante o tamanho dos programas processados no ambiente.

Gerência de Memória com Mapeamento de Bits

Com o emprego do mapeamento de bits, a memória é dividida em unidades de alocação, tão pequenas quanto poucas palavras ou tão grandes quanto vários quilobits. Correspondendo a cada unidade de alocação definida, há um bit do mapa de bits, que é 0 se a unidade estiver livre e 1 se estiver ocupada.

O tamanho da unidade de alocação é um ponto importante do projeto. Quanto menor a unidade de alocação, maior o mapa de bits. No entanto, mesmo com uma unidade de alocação tão pequena quanto quatro bytes, 32 bits de memória necessitarão de somente um bit do mapa. Se a unidade de alocação escolhida for muito grande, o mapa de bits será pequeno, mas uma parcela considerável da memória poderá ser desperdiçada na última unidade, se o tamanho do processo não for um múltiplo exato do tamanho da unidade de alocação.

O mapa de bits é uma forma simples de controlar a alocação da memória, pois seu tamanho só depende do tamanho da memória da unidade de alocação. O principal problema deste método ocorre quando for necessário trazer para a memória um processo que ocupa k unidades de alocação. O gerente de memória deve procurar por k bits 0

consecutivos no mapa. Esta procura é excessivamente lenta, de forma que, na prática, os mapas de bit raramente são usados.

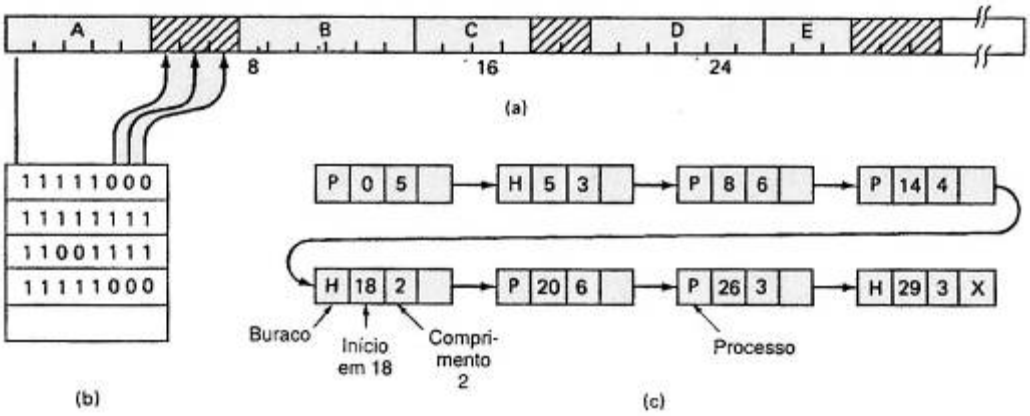

Figura 17: Mapa de bits e Listas ligadas

Gerência da Memória com Listas Ligadas

Uma outra forma de controlar a alocação da memória é mantendo uma lista ligada dos segmentos livres e ocupados da memória. Entende-se por segmento um processo ou um buraco entre dois processos. Cada entrada da lista especifica um buraco (H) ou um processo (P), o endereço no qual o segmento começa e um ponteiro para a próxima entrada.

A lista de segmentos poderá ser ordenada pelos endereços. Esta ordenação tem a vantagem de permitir uma rápida atualização da lista sempre que um processo terminar ou for removido da memória. Um processo que deixa a memória por qualquer motivo sempre tem dois vizinhos, exceto quando estiver no início ou no fim da fila. Tais vizinhos podem ser processos ou buracos, levando a quatro possíveis combinações.

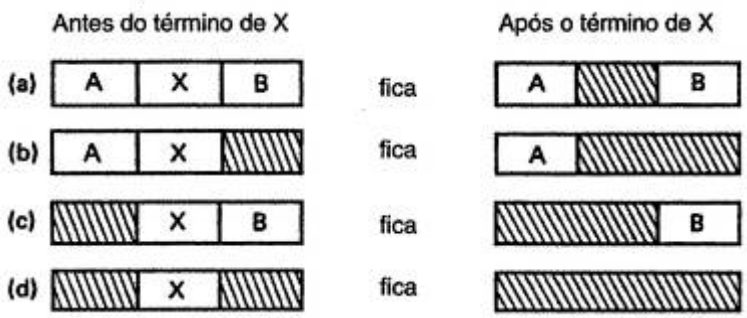

Figura 18: Combinações de vizinhos possíveis para um processo X que está terminando seu processamento

Quando os processos e os buracos são mantidos numa lista ordenada por endereços, vários algoritmos podem ser usados para alocar memória para um novo processo ou para um processo que precise ser transferido do disco para a memória.

Vamos assumir que o gerente de memória conheça a quantidade de memória a ser alocada:

- Primeira alocação: o gerente de memória procura ao longo da lista de segmentos até encontrar um buraco que seja suficiente grande para abrigar o processo. O buraco é então quebrado em dois pedaços, um para o processo e outro para o espaço não-utilizado. A primeira alocação é um algoritmo extremamente rápido, pois ele gasta em procura o mínimo de tempo possível.
- Próxima alocação: que funciona exatamente igual ao primeiro, exceto pelo fato de guardar a posição onde ele encontra um buraco conveniente. Da próxima vez que o algoritmo for chamado, ele inicia sua busca deste ponto, em vez de começar de novo no início da lista. A performance deste algoritmo é um pouco pior que o anterior.
- Melhor alocação: busca na lista inteira a melhor posição para armazenar o processo que está precisando de espaço. Em vez de parar ao encontrar um buraco grande, que poderá ser necessário mais tarde, ele busca um buraco cujo tamanho seja o mais próximo possível do tamanho do processo. Este algoritmo é o mais lento, pois precisa pesquisar toda a lista cada vez que for chamado. Ele também resulta em maior desperdício de memória que o da primeira locação, pois ele tende a dividir a memória em buracos muito pequenos, que tornam-se difíceis de usar em função do tamanho reduzido.
- Pior alocação: a pior partição é escolhida, ou seja, aquela em que o programa deixa o maior espaço sem utilização. Apesar de utilizar as maiores partições, a técnica pior alocação deixa espaços livres maiores que permitem a um maior número de programas utilizar a memória, diminuindo o problema da fragmentação. As simulações mostram que este algoritmo não apresenta bons resultados práticos.

Todos os quatro algoritmos poderão ter seus tempos de execução reduzidos mantendo-se as listas separadas para processos e buracos. Desta forma, em todos eles se gasta toda a energia na pesquisa de buracos e não de processos. O preço pago por isto é a complexidade adicional introduzida no algoritmo e o maior tempo gasto na liberação da memória, uma vez que um segmento que foi liberado deve ser removido da lista de processos e inserido na de buracos.

Se mantivermos listas distintas para processos e buracos, a de buracos deve ser ordenada em função do tamanho dos mesmos, para fazer com que a melhor alocação fique mais rápida. Quando este algoritmo pesquisa a lista de buracos do menor para o maior, tão logo ele encontre um buraco adequado, poderá abandonar a busca, pois certamente este será o melhor buraco. Com a lista ordenada por tamanho, as performances da primeira alocação e da melhor alocação se equivalem.

Gerência da Memória Usando o Sistema Buddy

Na seção anterior vimos que mantendo todos os buracos em uma ou mais listas, ordenadas pelo tamanho do buraco, conseguimos um tempo de alocação bem razoável, porém um tempo de liberação extremamente ruim, pelo fato de todas as listas terem que ser pesquisadas para encontrar os vizinhos dos segmentos liberados. O sistema buddy é um algoritmo para gerenciamento de memória que tira vantagem do fato de os computadores usarem números binários para o endereçamento, como uma forma de acelerar a junção dos buracos adjacentes quando um processo termina ou quando é retirado da memória.

Tal algoritmo funciona da seguinte forma. O gerente de memória mantém uma lista de blocos livres de tamanho 1, 2, 4, 8, 16, etc bytes, até o tamanho da memória. Com uma memória de 1M, serão necessárias 21 destas listas, desde o tamanho de 1 byte até 1M byte. As demais listas estão vazias. A configuração inicial da memória é mostrada na figura abaixo:

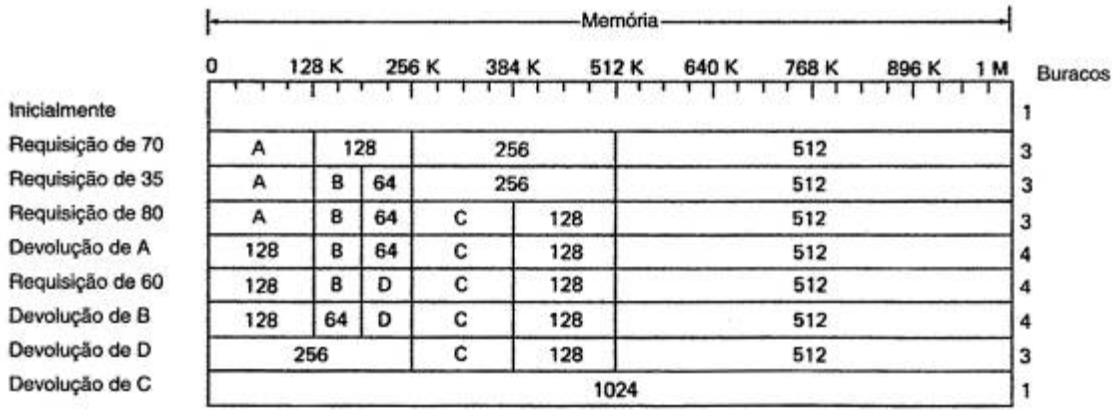

Figura 19: Sistema Buddy

Inicialmente um processo A de 70k é trazido para a memória. Como a lista de buracos só tem potências inteiras de 2, o processo requisitará 128k por ser a potência de 2 mais próxima da sua necessidade. Não há nenhum bloco de 128k disponível e, portanto, o bloco de 1M vai ser dividido em dois blocos de 512k, chamados buddies, um começando no endereço 0 e outro no 512k. Um deles, o de endereço 0, é então dividido em dois outros de 256k, um no endereço 0 e outro no 256k. Destes dois, o que estiver no endereço mais baixo vai ser novamente dividido em dois de 128k, e o do endereço 0 é alocado ao processo A.

A seguir um processo B de 35k deve ser escrito na memória. Neste momento devemos arredondar 35k para a potência 2 imediatamente superior, descobrindo que não há nenhum bloco de 64k disponível. Desta forma, o outro buddie de 128k é dividido em dois, um no endereço 128k e outro no 192k. O bloco de endereço 128k é alocado ao

processo B. A terceira requisição é para um processo C de 80k, atendida como um bloco de 128k com início em 256k.

Imaginemos agora que o processo A termine seu processamento e o bloco de 128k deve ser devolvido. Ele simplesmente deve ser acrescentado à lista dos blocos livres de 128k. Agora, o processo D requisita um bloco de 60k. O bloco de 64k do endereço 192k é alocado ao processo.

O processo B termina e devolve o bloco. Somente quando D devolve o bloco é que pode ser feito a junção em um bloco de 256k. Quando C finaliza, voltaremos à configuração inicial de 1M em 0.

O sistema buddy tem uma vantagem sobre os algoritmos que ordenam os blocos pelo tamanho, mas não necessariamente forçando-os a serem potências de 2. A vantagem é que quando um bloco de tamanho 2^k for liberado, o gerente de memória só deve procurar na lista dos blocos de 2^k para ver se há possibilidade de junção. Com os outros algoritmos todas as listas deverão ser verificadas. O resultado é que o sistema buddy é bem mais rápido que os demais.

Infelizmente, o sistema buddy não é eficiente em termos de utilização da memória. O problema decorre obviamente da necessidade de se arredondar a requisição feita pelo processo para a próxima potência de 2. A um processo de 35k deve ser alocado 64k. Os 29k excedentes são perdidos.

6. SWAPPING

Mesmo com o aumento da eficiência da multiprogramação e, particularmente, da gerência de memória, muitas vezes um programa não podia ser executado por falta de uma partição livre disponível. A técnica de *swapping* foi introduzida para contornar o problema da insuficiência de memória principal.

Em todos os esquemas apresentados anteriormente, um processo permanecia na memória principal até o fim da sua execução, inclusive nos momentos em que esperava por um evento, como uma operação de leitura ou gravação. O swapping é uma técnica aplicada à gerência de memória para programas que esperam por memória livre para serem executados. Nesta situação, o sistema escolhe um processo residente, que é transferido da memória principal para a memória secundária (swap out), geralmente disco. Posteriormente, o processo é carregado de volta da memória secundária para a memória principal (swap in) e pode continuar sua execução como se nada tivesse ocorrido.

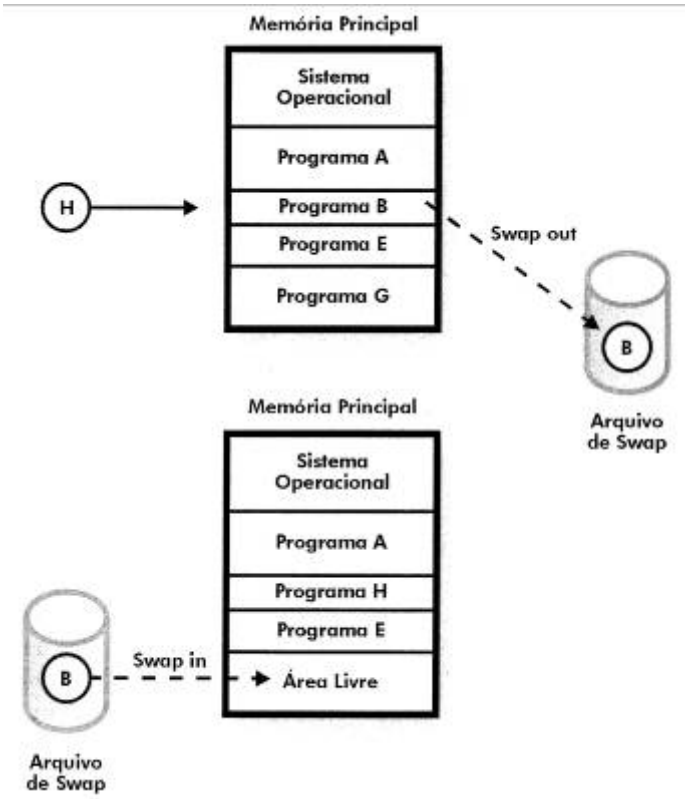

Figura 20: Swapping

O algoritmo de escolha do processo a ser retirado da memória principal deve priorizar aquele com menores chances de ser executado, para evitar o swapping desnecessário de um processo que será executado logo em seguida. Os processos retirados da memória estão geralmente no estado de espera, mas existe a possibilidade de um processo no estado de pronto também ser selecionado. No primeiro caso, o processo é dito no estado de espera outswapped e no segundo caso no estado de pronto outswapped.

Para que a técnica de swapping seja implementada, é essencial que o sistema ofereça um loader que implemente a relocação dinâmica de programas. Um loader relocável que não ofereça esta facilidade permite que um programa seja colocado em qualquer posição de memória, porém a relocação é apenas realizada no momento do carregamento. No caso do swapping, um programa pode sair e voltar diversas vezes para a memória, sendo necessário que a relocação seja realizada pelo loader a cada carregamento.

A relocação dinâmica é realizada através de um registrador especial denominado *registrador de relocação*. No momento em que o programa é carregado na memória, o registrador recebe o endereço inicial da posição de memória que o programa irá ocupar. Toda vez que ocorrer uma referência a algum endereço, o endereço contido na instrução será somado ao conteúdo do registrador, gerando, assim, o endereço físico. Dessa forma, um programa pode ser carregado em qualquer posição de memória.

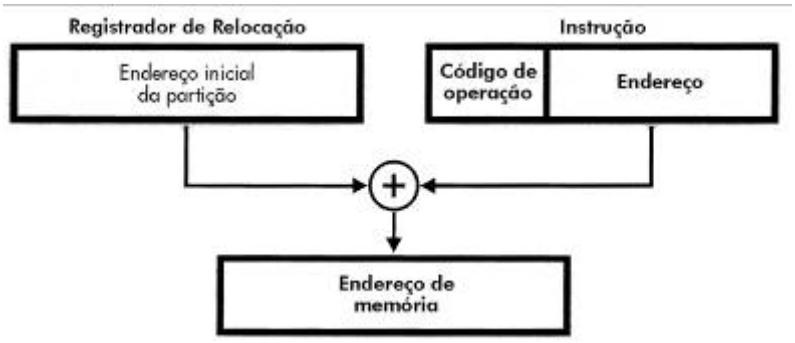

Figura 21: Relocação dinâmica

O conceito de swapping permite maior compartilhamento da memória principal e, conseqüentemente, maior utilização dos recursos do sistema operacional. Seu maior problema é o elevado custo das operações de entrada/saída (swap in/out). Em situações críticas, quando há pouca memória disponível, o sistema pode ficar quase que dedicado à execução de swapping, deixando de realizar outras tarefas e impedindo a execução dos processos residentes.

Os primeiros sistemas operacionais que implementaram esta técnica surgiram na década de 1960, como o CTSS do MIT e OS/360 da IBM. Com a evolução dos sistemas operacionais, novos esquemas de gerência de memória passaram a incorporar a técnica de swapping, como a gerência de memória virtual.