Unidade II

3 GERENCIAMENTO DE MEMÓRIA

Diferentemente da realidade atual dos sistemas computacionais, a memória principal era tratada como um recurso escasso e caro. Dessa forma, os projetistas desenvolviam o SO que não usasse muito espaço na memória e realizasse um uso otimizado dos recursos do computador.

Para sistemas monoprogramáveis, a gerência da memória é implementada de forma bastante simplificada. Por sua vez, nos multiprogramáveis, ela se torna crítica, devido à necessidade de se maximizar a quantidade de usuários e processos utilizando o espaço da memória principal de forma otimizada.

Neste tópico, serão apresentados os esquemas básicos de gerência da memória principal, acentuando suas vantagens, desvantagens e como evoluíram historicamente. Também estudaremos o mecanismo de gerência de memória virtual nos SOs modernos.

Mesmo atualmente, com a redução de custo e consequente aumento da capacidade da memória principal, seu gerenciamento é um dos fatores mais importantes no projeto de SOs.

Idealmente, os programadores desejam que a memória de um sistema computacional seja de alta capacidade, de baixo custo, de altíssima velocidade de acesso e com baixo tempo de resposta e não volátil, isto é, que não se apague na ausência de energia elétrica. Apesar da evolução da memória, a tecnologia atual não comporta a construção de tal memória.

3.1 Tipos de memória

Em um sistema computacional, existem diversos tipos de memória com características e particularidades próprias e que variam muito em função do tamanho de memória e do tempo de acesso a ela. O objetivo de todas é o mesmo: armazenar informação.

Os locais de armazenamento internos dos dados em um computador são os registradores do processador, a memória cache, tanto a interna, do processador – chamada de cache L1 –, como o cache externo, da placa-mãe – conhecido como L2. Ainda há a memória principal ou de acesso aleatório, do inglês random access memory (RAM). Adicionalmente, existem discos e unidade de armazenamento externos tais como pen drives, CD-ROMs, DVD-ROMs, fitas magnéticas que possuem a função de armazenar informação de forma não volátil.

Esses dispositivos de hardware são construídos utilizando diversas tecnologias e possuem características distintas, tais como a capacidade de armazenamento, a velocidade de operação, o custo

por tamanho de memória armazenado, o consumo energético e se a memória é volátil ou não volátil. A figura a seguir apresenta a hierarquia de memórias, normalmente representada como uma pirâmide.

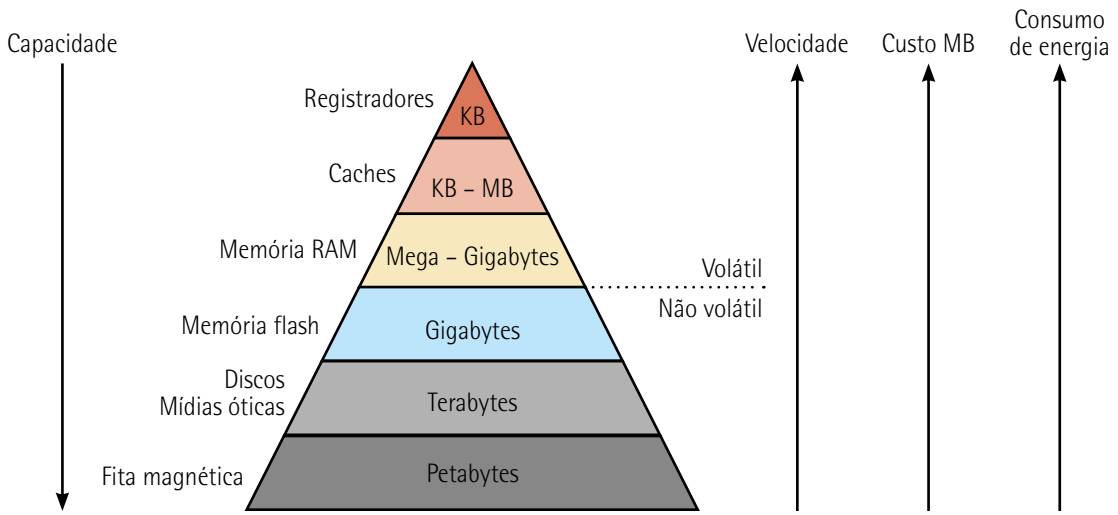

Figura 14 – Hierarquia de memórias

Fonte: Maziero (2019, p. 162).

Em relação à velocidade de uma memória, existem duas medidas mais utilizadas: o tempo de acesso ou latência e a taxa de transferência. Entende-se tempo de acesso como o tempo necessário para começar uma transferência de dados indo ou vindo de um meio de armazenamento. Por sua vez, a taxa de transferência representa a quantidade de bytes transmitidos ou recebidos por segundo que são lidos ou escritos por um tipo de memória após o início da transferência de dados.

A memória principal do computador é composta de uma área de RAM com uma grande sequência de bytes, uma unidade de memória utilizada pelo processador. Cada byte da memória RAM possui um endereço, que é usado para acessá-lo, e o tempo de acesso é praticamente o mesmo para todos os endereços de memória. Um computador moderno padrão possui alguns gigabytes (Gb) de memória RAM, isto é, mais 1 bilhão de posições de memória diferentes. Ela é utilizada para manter o SO e os processos em execução, bem como algumas áreas de finalidades específicas, por exemplo, buffers de dispositivos de E/S. A quantidade de memória RAM disponível em um sistema computacional forma o espaço de memória física.

Observação

O processador obtém acesso à memória principal através de barramentos de dados, de endereços e de controle. O barramento de endereços tem um número fixo de vias, que define a quantidade total de endereços de memória que podem ser gerados pelo processador, e um barramento de dados com n vias consegue gerar 2n endereços. O conjunto de endereços de memória que um processador pode representar é denominado **espaço de endereçamento**.

A memória cache tem apenas uma pequena parte do conteúdo da memória principal. Quando o processador referencia um dado armazenado na memória, busca-se primeiro verificar se ele está salvo na memória cache. Caso o processador encontre o dado, ou seja, ocorra um cache hit, não será realizado o acesso à memória principal, diminuindo assim o tempo de acesso.

De acordo com Machado e Maia (2013), a localidade é a tendência de o processador, durante a execução de um programa, referenciar dados e instruções e registros presentes na memória principal alocados em endereços próximos. Pode-se justificar essa tendência por causa da alta ocorrência de estruturas de repetição, conhecidas como sub-rotinas e acesso a estruturas de dados como vetores e tabelas.

Por meio do **princípio da localidade**, garante-se que após a transferência de um novo bloco da memória principal para a cache existirá uma alta probabilidade de haver cache hits em futuras referências, otimizando, assim, o tempo de acesso ao dado.

3.2 Funções do gerenciamento de memória

Uma tarefa do gerenciador de memória é realizar o gerenciamento da hierarquia de memória, de forma a identificar os espaços livres e ocupados da memória e alocar ou localizar processos e dados na memória. Outra função é controlar as partes que estão em uso e as que estão livres para alocação. Com isso, o gerenciamento de memória pode alocar memória aos processos, quando necessário, e liberar memória no encerramento de um processo, atualizando o status da memória.

Em ambientes de multiprogramação, é preciso que o SO proteja as áreas de memória já usadas por cada processo residente e da área de memória onde está alocado o próprio sistema. Quando um programa realizar uma tentativa de acesso indevido à memória, o SO deve impedir esse acesso não autorizado de alguma forma.

De acordo com Machado e Maia (2013), o papel da gerência de memória é tentar manter na memória principal a maior quantidade de processos alocados e, assim, permitir a maximização do compartilhamento do processador e demais recursos. Mesmo quando não exista mais espaço livre na memória principal, o sistema deve permitir que novos processos sejam aceitos e executados. Para isso ser possível, realiza transferência temporária de processos residentes memória principal para a memória secundária, ou seja, para o disco rígido. Com isso, há a liberação de espaço para novos processos, e tal mecanismo é denominado **swapping**, ou seja, uma troca de memórias.

Saiba mais

Para aprender mais sobre memória, leia:

ANDROID. *Visão geral do gerenciamento de memória*. 2023. Disponível em: https://shre.ink/nYeo. Acesso em: 25 set. 2023.

3.2.1 Swapping

Em algumas situações, não há espaço suficiente na memória principal para conter todos os processos correntemente ativos. A técnica de swapping foi criada para resolver esse problema, fazendo com que os processos excedentes sejam mantidos no disco e trazidos para execução dinamicamente. Assim, há uma troca de memórias entre os programas que são da memória principal e são armazenados temporariamente no disco rígido e, depois, quando houver espaço livre, saem do disco rígido para a memória principal.

A figura a seguir representa o funcionamento do swapping. O sistema escolhe um processo residente na memória, o qual será transferido para a memória secundária, normalmente o disco rígido, e essa etapa é denominada de **swap out**. Quando existe uma partição livre na memória principal, o processo que saiu da memória principal e foi transferido para memória secundária é carregado de volta na memória principal, na etapa denominada **swap in**. Com isso, o processo poderá continuar a ser executado como se nada tivesse acontecido.

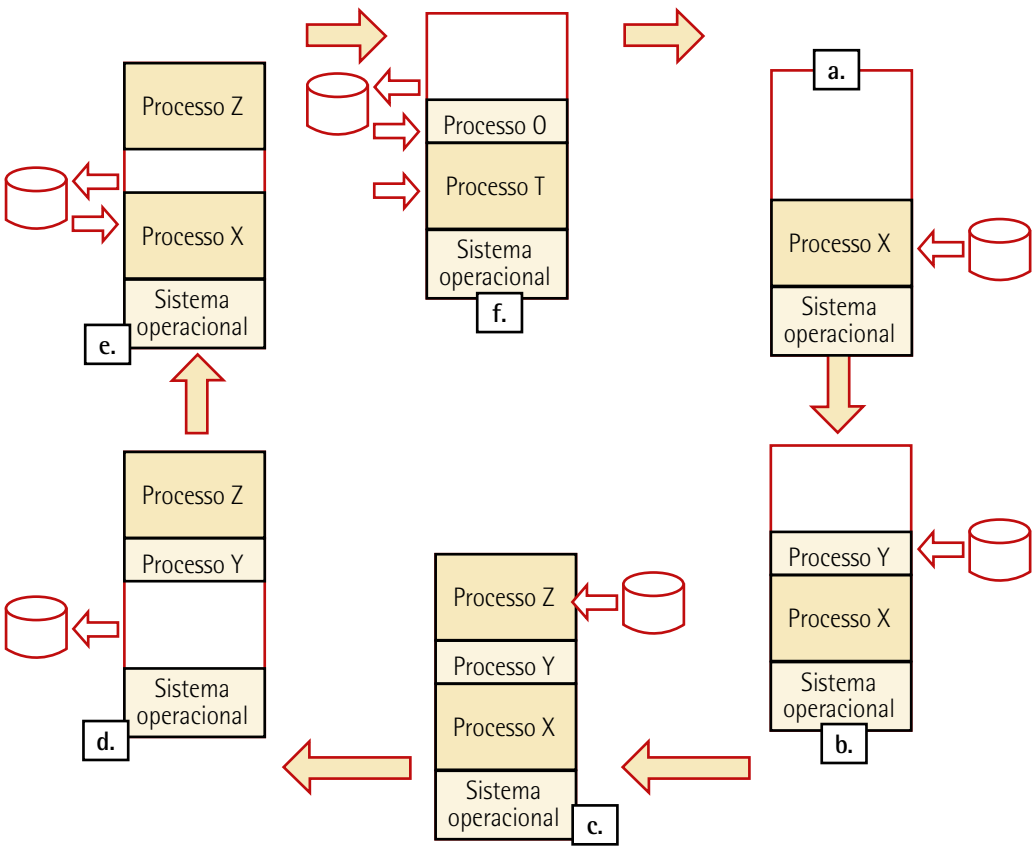

Figura 15 – Ciclo de troca dos processos entre memória principal e secundária

Fonte: Grupo UNIP-Objetivo.

Na parte a da figura anterior, inicialmente, o sistema operacional ocupa a parte mais baixa da memória (permanecerá nessa posição) e, logo em seguida, temos o processo X ocupando uma parte

da memória disponível. Depois, na parte b, um novo processo Y é criado ou trazido do disco rígido e posicionado na memória logo acima do processo X. Em c, um novo processo Z é adicionado. Nesse momento não há mais espaço disponível na memória principal para novos processos. Na parte d, o processo X fica ocioso, então, é enviado para o disco rígido. Nas partes e/f, temos outros processos sendo trocados e o ciclo vai sendo executado até que novos processos entrem e disputem o tempo de CPU e memória e/ou que processos terminados sejam eliminados no ciclo.

Observação

Por meio do processo de swapping, quando um processo for maior que a área livre, ou não existir nenhuma área de memória disponível, ele será transferido para o disco e ficará na memória secundária até que seja liberada memória principal suficiente para o carregamento do programa.

Com a introdução da técnica de swapping, é possível um maior compartilhamento da memória principal e, consequentemente, maior utilização dos recursos do sistema computacional. Entretanto, essa técnica traz um elevado custo das operações de E/S (swap in/swap out). Quando existe pouca memória disponível, o sistema pode ficar comprometido com a realização de swapping, deixando de executar outras tarefas e impedindo a operação dos demais processos residentes. Essa situação é considerada como crítica na gerência de memória e é denominada thrashing.

Segundo Machado e Maia (2013), o loader ou carregador é o utilitário do SO com a função de carregar na memória principal um programa para ser executado. O procedimento de carga varia com o código gerado pelo linker e, em função deste, o loader é classificado como de tipo absoluto ou relocável.

No caso de um código executável do tipo absoluto, o loader somente precisa conhecer o endereço de memória inicial e o tamanho do módulo para realizar o carregamento. Desse modo, o loader transfere o programa da memória secundária para a memória principal e inicia sua execução (loader absoluto). Com esse tipo de código, não é possível realizar a operação de swapping.

No caso do código relocável, o programa pode ser alocado em qualquer posição de memória, e o loader é responsável pela relocação no momento do carregamento (loader relocável).

Para a implementação da técnica de swapping, é necessário que o sistema faça a relocação dinâmica, pois o programa pode entrar e sair da memória principal e ser alocado em diferentes endereços de memória. Essa operação é feita por meio de um registro denominado registrador de relocação. Assim que o programa é colocado na memória, o registrador de relocação obtém o endereço inicial da posição de memória a ser ocupada pelo programa. Durante a execução do código, quando existe referência a algum endereço, será adicionada essa referência com o valor de endereço inicial e será determinado o endereço físico, como mostrado na figura a seguir. Com isso, será possível carregar o programa em qualquer posição de memória.

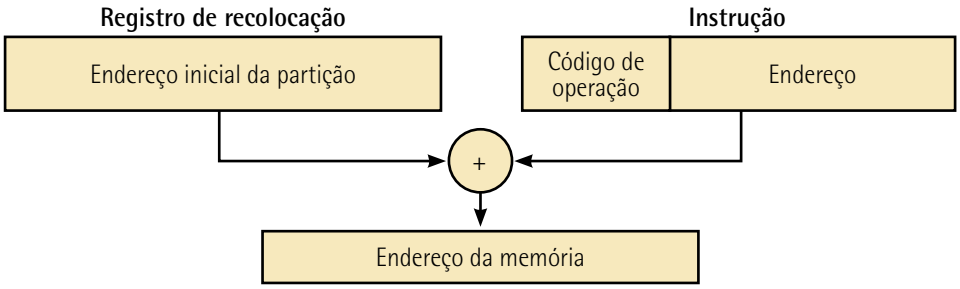

Figura 16 – Relocação dinâmica

Fonte: Machado e Maia (2013, p. 157).

3.3 Gerenciamento básico de memória

De acordo com Tanenbaum e Woodhul (2008), os sistemas de gerenciamento de memória podem ser divididos em duas classes fundamentais: aqueles que utilizam exclusivamente a memória principal para alocar os processos e aqueles que alternam os processos entre a memória principal e o disco durante a execução. Essa troca de memória é conhecida como swapping.

3.3.1 Sistemas monoprogramáveis

A forma de gerenciamento de memória mais simplificada ocorre quando é executado apenas um programa por vez, compartilhando a memória exclusivamente entre esse programa e o SO. Na figura a seguir são apresentadas três formas de organização da memória principal, que será dividida para alocar o SO e o programa do usuário. O SO pode estar na parte inferior da memória RAM, como se vê na parte a da figura a seguir. Pode estar em uma memória somente de leitura, ou seja, Read-Only Memory (ROM), na parte superior da memória, como se observa em b, ou os drivers de dispositivo podem estar na parte superior da memória em uma ROM e o restante do sistema na RAM abaixo dela, como é visto em c.

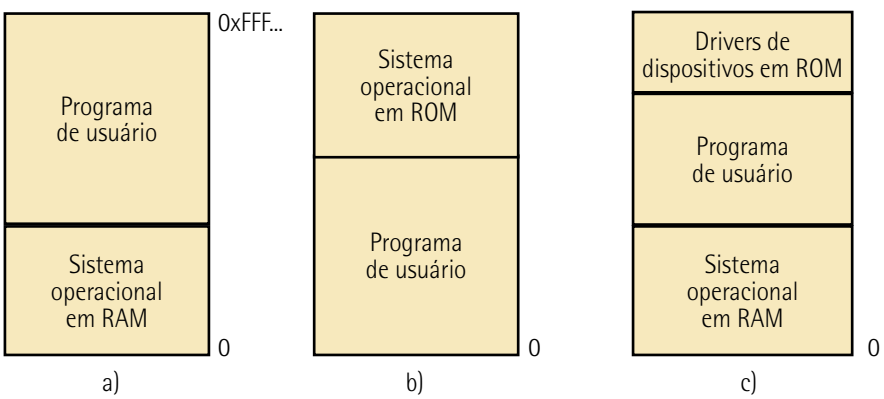

Figura 17 – Formas do gerenciamento de memória para sistemas monoprogramáveis

Fonte: Tanenbaum e Woodhul (2008, p. 347).

O primeiro modelo foi usado inicialmente em computadores de grande porte e em minicomputadores, entretanto hoje não é mais utilizado. O segundo é usado em pequenos computadores de mão ou palmtops e em sistemas embarcados. Já o terceiro foi aplicado pelos primeiros computadores pessoais, por exemplo, no sistema MS-DOS. Neste, parte do sistema que fica na ROM é chamada de BIOS (Basic Input Output System – sistema básico de entrada e saída).

3.3.2 Sistemas multiprogramáveis com partições fixas

Com a evolução dos programas utilitários do SO, o código do programa resultante, em vez de fazer uma referência de endereço absoluto, passou a ser relocável. No código relocável, as referências a endereços existentes no programa são relativas ao início do código, e não estão atreladas a endereços físicos de memória. Com isso, os programas podem ser executados a partir de qualquer partição.

Uma maneira simplificada do gerenciamento de memória para sistemas multiprogramáveis consiste em simplesmente dividir a memória até n partições, que não precisam ter o mesmo tamanho. Esse particionamento pode ser feito manualmente, por exemplo, quando o sistema é inicializado, e esse tipo de gerência de memória é conhecido como alocação particionada estática ou fixa.

Quando ocorre a chegada de um processo, ele pode ser colocado na fila de entrada da menor partição com tamanho suficiente para contê-lo. Como as partições são fixas nesse esquema, todo espaço não utilizado por um job em uma partição é desperdiçado enquanto esse job é executado. A figura a seguir ilustra esse sistema de partições fixas e filas de entrada separadas.

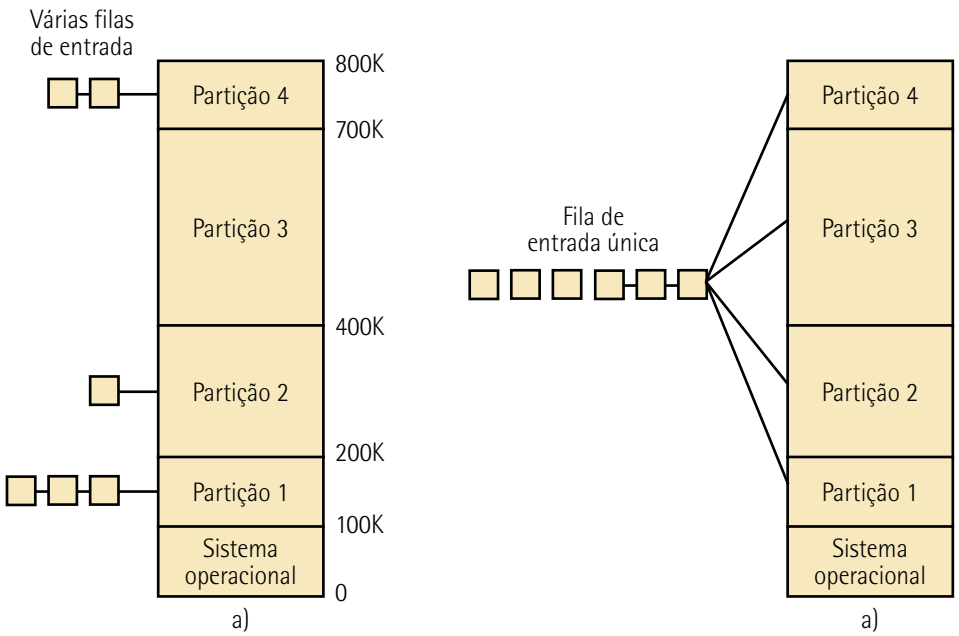

Figura 18 – Multiprogramação com partições fixas de memória: (a) com filas de entrada separadas para cada partição; (b) com uma fila única de entrada

Fonte: Tanenbaum e Woodhul (2008, p. 347).

Uma desvantagem existente na ordenação das tarefas recebidas em filas separadas se torna evidente quando a fila de uma partição grande está vazia, mas a de uma partição pequena está cheia, como acontece nas partições 1 e 3 da parte a da figura anterior. Nessa situação, os jobs pequenos são obrigados a esperar para serem alocados na memória, mesmo com partições de memória livre.

Quando somente há uma fila única, como na parte b da figura anterior, assim que uma partição ficar desocupada, o job mais próximo do começo da fila, e que caiba na partição vazia, poderá ser alocado nessa partição e será executado.

Visando manter o controle sobre quais partições estão alocadas, a gerência de memória elabora uma tabela constando o endereço inicial de cada partição, o tamanho da participação e se está alocada para algum processo ou não, como mostra a figura a seguir.

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

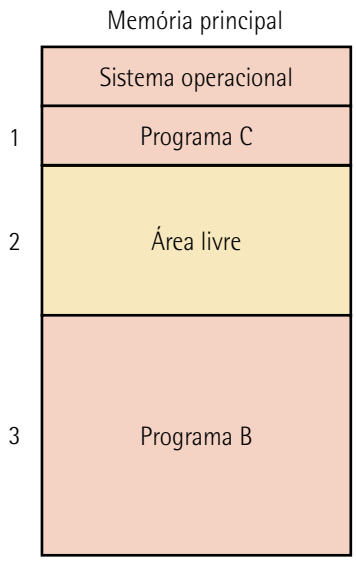

Figura 19 – Tabela de alocação de partições

Fonte: Machado e Maia (2013, p. 151).

Quando um programa é selecionado para ser carregado na memória, a tabela de alocação de partições é percorrida pelo SO na tentativa de localizar uma partição livre onde o programa possa ser carregado.

Com o advento da multiprogramação, dois problemas básicos são introduzidos no gerenciamento de memória e que devem ser resolvidos: realocação e proteção.

Nesse esquema de alocação de memória, existem dois registradores, os quais representam o limite inferior e superior da partição, onde o programa está sendo executado. Quando o programa tenta acessar uma posição de memória fora dos limites definidos pelos registradores, ele é paralisado e o SO envia uma mensagem de violação de acesso.

Quando o tamanho da partição é fixa, dificilmente os programas preenchem totalmente as partições onde são colocados, e esse espaço de memória que sobra será desperdiçado. Esse tipo de problema é denominado como fragmentação interna.

3.3.3 Sistemas multiprogramáveis com partições variáveis

Com o objetivo de reduzir a fragmentação interna, foi eliminado o conceito de partições de tamanho fixo com a introdução da alocação particionada dinâmica ou variável. Nesse esquema, cada programa pode utilizar o espaço que necessita, tornando essa área sua partição. Dado que os programas usam apenas o espaço necessário, o problema da fragmentação interna não acontece.

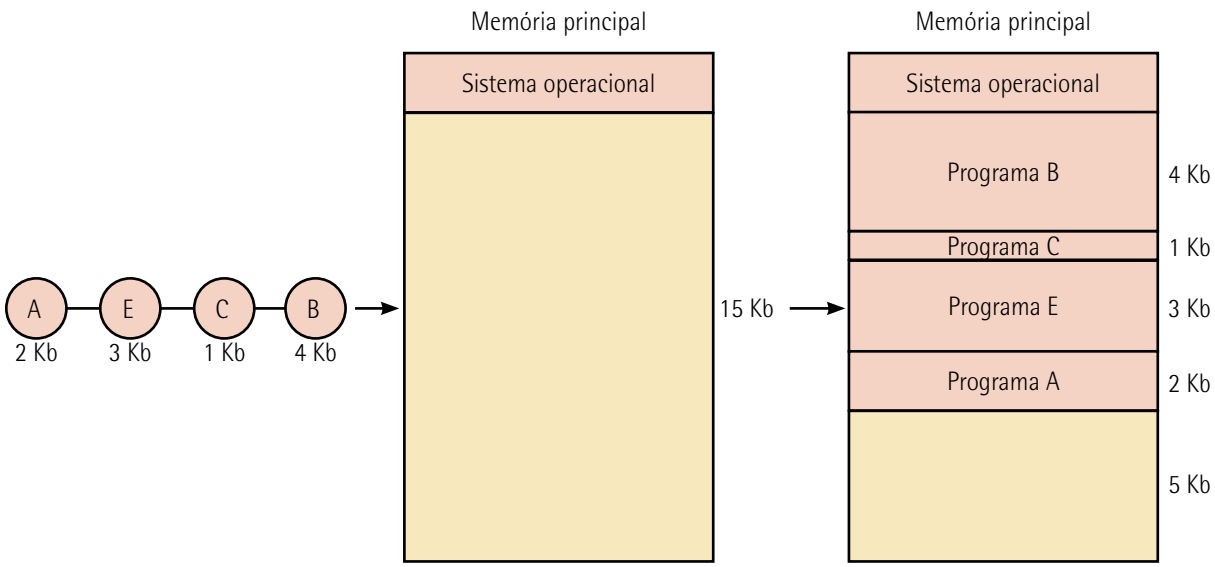

Figura 20 – Alocação particionada dinâmica

Fonte: Machado e Maia (2013, p. 152).

Entretanto, depois de diversas alocações de processos, uma situação que pode acontecer é que os espaços de memória estão se tornando cada vez menores na memória e os novos programas não podem ser alocados nesses espaços. Esse novo problema é denominado fragmentação externa.

3.3.4 Estratégias de alocação

Com a intenção de minimizar a ocorrência de fragmentação externa, cada pedido de alocação pode ser analisado para encontrar a área de memória livre que melhor o atenda. A melhor estratégia a ser escolhida por um sistema depende de alguns fatores, e o tamanho dos programas processados é o mais relevante. Independentemente do algoritmo adotado, o sistema possui uma lista de áreas livres, com o endereço e tamanho de cada área.

A estratégia **First-fit** (o primeiro que couber) consiste em escolher a primeira área livre que possua um espaço livre maior ou igual ao solicitado. A vantagem dessa abordagem é a rapidez, sobretudo se a lista de áreas livres for muito longa.

Já na estratégia **Best-fit** (a que melhor couber), a partição escolhida será aquela em que houver o menor espaço sem utilização após a alocação do programa. Para esse algoritmo, o objetivo é minimizar o desperdício de memória. A listagem de áreas livres está ordenada por tamanho, diminuindo o tempo de busca por uma área desocupada. Com o algoritmo alocado, a partição deixa a menor área livre, e existe a tendência de que a memória fique com pequenas áreas não contiguas e, consequentemente, será agravado o problema da fragmentação externa.

Por sua vez, a estratégia **Worst-fit** (a que pior couber) consiste na escolha da maior área livre possível, de maneira que o excedente não utilizado seja o suficiente para ser usado em futuras alocações.

Uma variação da estratégia First-fit é a **Next-fit** (o próximo que couber). Nela, percorre-se a lista de áreas a partir da última alocada ou liberada para que o uso das áreas livres tenha uma distribuição mais homogênea no espaço de memória.

3.4 Gerenciamento de memória no Linux

É formado por duas partes: a alocação e a liberação de memória física com páginas, grupos de páginas e pequenos blocos de RAM e a manipulação da memória virtual, que será explorada mais adiante.

Por causa de restrições específicas de hardware, o Linux divide a memória física em quatro zonas ou regiões:

- ZONE_DMA.
- ZONE_DMA32.
- ZONE_NORMAL.
- ZONE_HIGHMEM.

Observação

O acesso direto à memória ou direct memory access (DMA) é o método que possibilita que um dispositivo de E/S ou periférico transmita ou receba dados diretamente da memória principal, sem necessidade de participação da CPU, de modo a reduzir o tempo de leitura e gravação de dados.

Com o DMA, é realizado o controle de determinados dispositivos de um computador para acessarem a memória do sistema de forma independente do processador, o que possibilita que o processador seja utilizado para outras tarefas.

De acordo com Silberschatz, Galvin e Gagne (2013), a divisão da memória em cada uma dessas regiões depende da arquitetura. No caso da arquitetura Intel x86-32, certos dispositivos ISA, que é a sigla de Industry Standard Architecture, acessam somente os 16 MB inferiores de memória física usando o DMA. Nesses sistemas, os 16 MB iniciais de memória física irão compor a ZONE_DMA.

Em outros sistemas, certos dispositivos podem acessar apenas os primeiros 4 GB de memória física, apesar de suportarem endereços de 64 bits. Em tais sistemas, os primeiros 4 GB de memória física compõem a ZONE_DMA32. A ZONE_HIGHMEM, cujo nome remete a high memory, refere-se à memória física que não é mapeada para o espaço de endereçamento do kernel. Em especial, na arquitetura Intel de 32 bits, cujo espaço de endereçamento é de 2^{32} endereços ou 4 GB, o kernel é mapeado para os primeiros 896 MB do espaço de endereçamento; a memória restante é chamada memória alta e é alocada a partir da ZONE_HIGHMEM. Para concluir, o restante da memória estará na ZONE_NORMAL, as páginas normais mapeadas regularmente. As restrições de uma arquitetura é que definem se ela tem determinada zona.

O relacionamento entre zonas e endereços físicos na arquitetura do Intel x86-32 é apresentado na tabela a seguir.

Tabela 2

Zona	Memória física
ZONE_DMA	< 16 MB
ZONE_NORMAL	16..896 MB
ZONE_HIHGMEN	> 896 MB

Fonte: Silberschatz, Galvin e Gagne (2015, p. 432).

Nas arquiteturas de 64 bits, como a do Intel x86-64, existe uma pequena ZONE_DMA de 16 MB (para dispositivos legados) e todo o resto de sua memória fica na ZONE_NORMAL, sem "memória alta".

O kernel mantém uma lista de páginas livres para cada zona. Quando chega uma solicitação de memória física, o kernel atende à solicitação usando a zona apropriada.

3.4.1 Sistema de pares (sistema buddy)

O principal gerenciador de memória física no kernel do Linux é o alocador de páginas. Cada zona tem seu próprio alocador, que é responsável por alocar e liberar todas as páginas físicas para a zona e pode alocar intervalos de páginas fisicamente contíguas sob demanda. O alocador usa um sistema de pares ou buddy system para rastrear páginas físicas disponíveis.

O sistema de pares ou buddy system aloca a memória a partir de um segmento de tamanho fixo composto de páginas fisicamente contíguas, isto é, adjacentes. Quando ocorre a liberação de duas regiões parceiras alocadas, estas são combinadas para formar uma região maior, denominada heap de pares ou buddy heap. Essa região maior também possui um parceiro com o qual ela pode se unir para formar uma região livre ainda maior.

Por outro lado, se uma solicitação de pouca memória não puder ser atendida pela alocação de uma pequena região livre existente, então uma região livre maior será subdividida em dois parceiros para atender à solicitação. É utilizado um alocador de potência de 2, que atende às solicitações em unidades dimensionadas como uma potência de 2 (4 KB, 8 KB, 16 KB, 32 KB e assim sucessivamente). Uma solicitação em unidades não apropriadamente dimensionadas é arredondada para a próxima potência de 2 mais alta. Por exemplo, uma solicitação de 21 KB é atendida com um segmento de 32 KB.

Considere um exemplo em que o tamanho de um segmento de memória seja inicialmente de 256 KB e o kernel realize uma solicitação de 21 KB de memória. No primeiro momento, o segmento é dividido em dois pares de potências de 2, os buddies, denominados A_L e A_R, cada um com 128 KB. Um desses pares é dividido mais uma vez em dois pares de 64 KB – B_L e B_R. No entanto, a próxima potência de 2 mais alta após 21 KB é 32 KB e, assim, B_L ou B_R é dividido novamente em dois pares de 32 KB, C_L e C_R. Um desses pares é usado para atender à solicitação de 21 KB. Esse esquema é ilustrado na figura a seguir, em que C_L é o segmento alocado à solicitação de 21 KB.

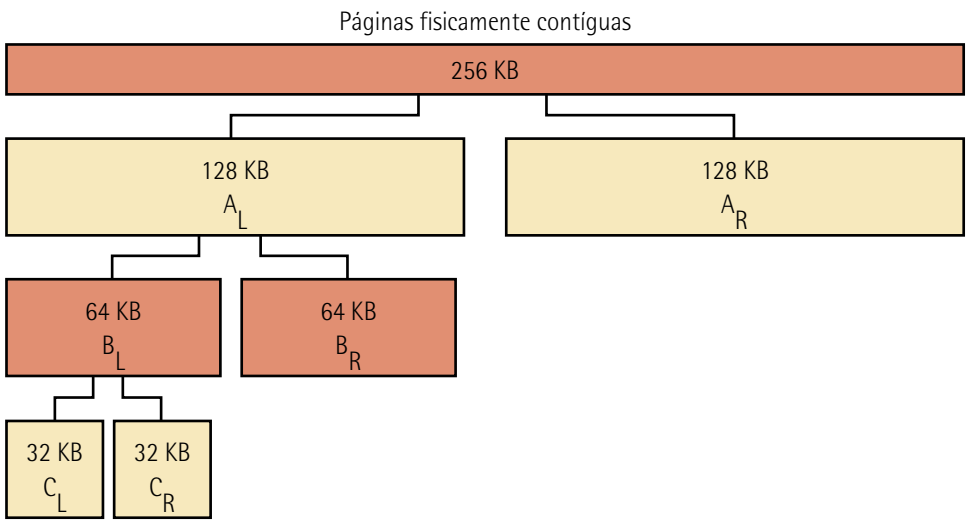

Figura 21 – Alocação com o sistema de pares ou algoritmo buddy

Fonte: Silberschatz, Galvin e Gagne (2015, p. 238).

O controle de áreas de memória livres e alocadas é feito por intermédio de um vetor denominado _área livre ou free_area. Cada elemento desse vetor é, na realidade, uma lista encadeada que possui os dados as páginas livres no sistema organizadas em blocos. O primeiro elemento aponta para blocos que equivalem a uma página física, o segundo, a blocos de duas páginas físicas, o terceiro, a blocos de quatro páginas físicas, assim crescendo sucessivamente em potência de dois.

A figura a seguir ilustra o procedimento de busca por memória, considerando a necessidade de uma alocação de 16 kbytes (KB). Nesse exemplo, considera-se que free_area possui entradas livres apenas para três blocos de 64 KB. O algoritmo de alocação particiona um desses blocos de 64 KB em dois de 32 KB. Um deles é inserido no vetor free_area na entrada correspondente a 32 KB, o outro é subdividido novamente em dois blocos de 16 KB cada. Novamente, um deles é inserido no vetor free_area, e o outro é empregado para satisfazer a alocação solicitada.

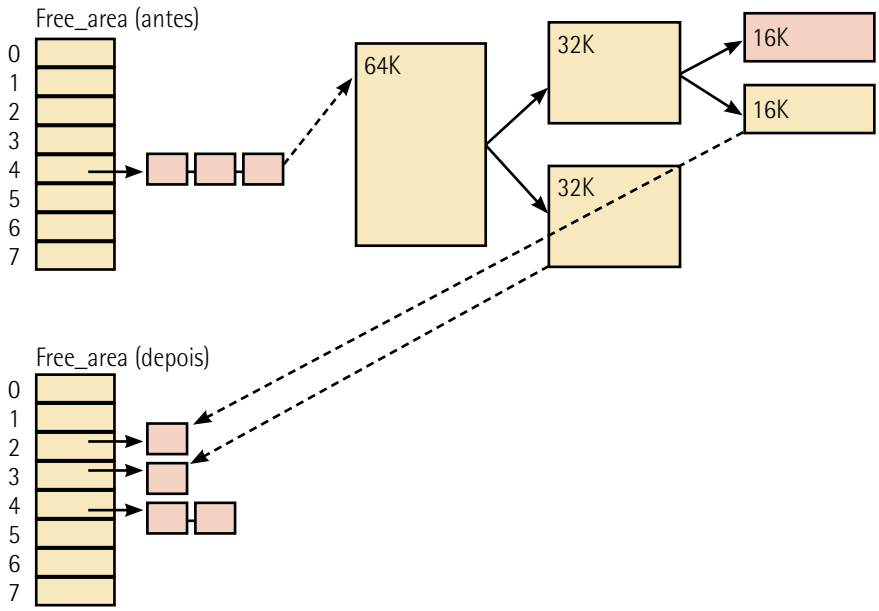

Figura 22 – Alocação de memória física no Linux

Fonte: Oliveira *et al.* (2009, p. 259).

Quando ocorre a liberação de um bloco de páginas, o algoritmo verifica se existe um bloco livre de mesmo tamanho adjacente ao bloco, ou buddy block, que será desalocado. Caso exista, os dois serão fundidos e criarão um único bloco com um tamanho igual a duas vezes o da área liberada. Esse procedimento de obtenção de blocos maiores é realizado recursivamente até que os blocos não possam mais formar um único bloco maior.

Um ponto favorável do sistema de pares é a combinação de pares adjacentes para formar segmentos maiores utilizando a técnica denominada fusão. No exemplo apresentado na figura anterior, quando o kernel liberar a unidade C_L à qual ele foi alocado, o sistema poderá fundir C_L e C_R em um segmento de 64 KB. Por sua vez, esse segmento, B_L, poderá ser fundido com seu par B_R para então formar um segmento de 128 KB. Por fim, voltaremos a ter o segmento original de 256 KB de tamanho.

Uma deficiência de sistema de pares é que o arredondamento para a próxima potência de 2 é mais alta e tende a transformar os grandes blocos de memória em blocos menores, causando eventualmente a fragmentação de memória dentro dos segmentos alocados. Por exemplo, uma solicitação de 34 KB pode ser atendida somente com um segmento de 64 KB. Na verdade, não podemos garantir que menos de 50% da unidade alocada serão desperdiçados em razão da fragmentação interna.

No Linux, o menor tamanho locável por meio desse mecanismo é uma única página física ou frame. As alocações de memória no kernel do Linux são feitas estaticamente por drivers que reservam uma área contígua de memória durante o tempo de inicialização do sistema, ou dinamicamente, pelo alocador de páginas. Entretanto, as funções do kernel não precisam usar o alocador básico para reservar memória. Vários subsistemas especializados de gerenciamento da memória usam o alocador de páginas subjacente para gerenciar seus próprios pools de memória.

De forma geral, todas as alocações de memória no kernel do Linux são feitas estaticamente por drivers que reservam uma área contígua de memória durante o tempo de inicialização do sistema, ou dinamicamente, pelo alocador de páginas. No entanto, as funções do kernel não precisam usar o alocador básico para reservar memória. Vários subsistemas especializados de gerenciamento da memória usam o alocador de páginas subjacente para gerenciar seus próprios pools de memória.

4 MEMÓRIA VIRTUAL

A memória virtual representa uma técnica sofisticada e poderosa de gerência de memória, na qual tanto a memória principal quanto a memória secundária são agrupadas de modo a oferecer ao usuário a impressão de existir uma memória principal de maior capacidade que a capacidade real.

Nessa técnica, a memória secundária é utilizada como uma memória "cache" para partes do espaço de endereçamento dos processos. É usada porque o tamanho de memória ocupado pelo software é crescente, para permitir que programas maiores que a memória de acesso aleatório (RAM) sejam executados.

Por exemplo, o gerenciamento da memória no Linux possui duas partes. A primeira está ligada à alocação e a liberação de memória física, páginas, grupos de páginas e pequenos blocos de RAM e foi apresentada no tópico anterior. A segunda parte é a manipulação da memória virtual, que é a memória mapeada para o espaço de endereçamento de processos em execução.

4.1 Unidade de gerenciamento de memória (MMU)

O espaço de endereçamento lógico é o conjunto de todos os endereços lógicos gerados por um programa. Analogamente, espaço de endereço físico inclui todos os endereços físicos correspondentes a esses endereços lógicos. O MMU é um dispositivo de hardware que transforma endereços virtuais em endereços físicos.

Endereços físicos ou reais são os endereços dos bytes de memória física do computador. Eles são definidos pela quantidade de memória disponível na máquina. Já os **endereços lógicos** ou virtuais são os endereços de memória usados pelos processos e pelo SO e, portanto, empregados pelo processador durante a execução. Esses endereços são definidos de acordo com o espaço de endereçamento do processador.

O espaço de endereço do processo representa o conjunto de endereços lógicos a que um processo faz referência em seu código-fonte. Há três tipos de endereços usados em um programa, antes e depois da memória ser alocada: endereços simbólicos, endereços relativos e endereços físicos. Os endereços

simbólicos são aqueles utilizados em um código-fonte, por exemplo, nomes das variáveis, constantes e rótulos de instrução.

Para os endereços relativos, no momento da compilação, um compilador converte endereços simbólicos em endereços relativos. Já para os endereços físicos, o carregador produz esses endereços no momento em que um programa é carregado na memória principal.

Os programas manipulam endereços lógicos e nunca conhecem o endereço físico real. A MMU suporta o SO na realização do mapeamento dos endereços da memória física e endereços da memória virtual, permitindo, assim, a movimentação eficaz das partes dos programas da memória virtual para o disco ou vice-versa.

A figura a seguir apresenta a localização da MMU como parte do chip da CPU, como é na maior parte dos computadores atuais.

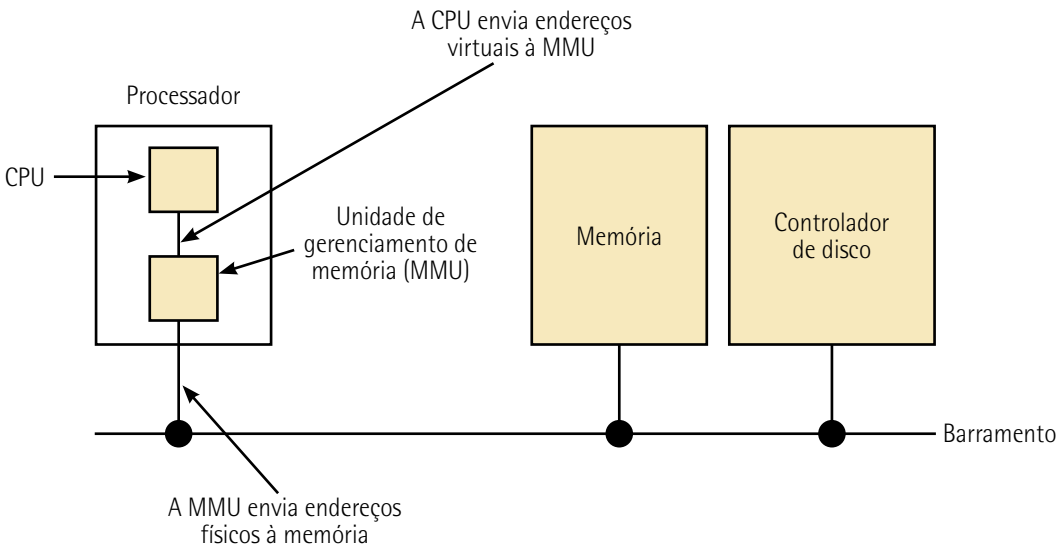

Figura 23 – Posição e função da MMU

Fonte: Tanenbaum e Bos (2016, p. 135).

Na memória virtual são utilizados dois tipos de endereços físicos e endereços lógicos. A conversão de endereços virtuais em físicos é realizada pela MMU, que é o elemento de hardware que realiza essa função.

Um conceito fundamental de memória virtual consiste em não vincular o endereçamento feito pelo programa dos endereços físicos da memória principal. Dessa forma, não existe mais o limite da memória física disponível para programas e suas estruturas de dados, já que eles podem possuir endereços associados à memória secundária.

Adicionalmente, a memória virtual possibilita maior quantidade de processos compartilhando a memória principal, pois apenas partes de cada processo estarão residentes na memória. Outros benefícios

dessa técnica são aumentar o grau de multiprogramação e minimizar o problema da fragmentação da memória principal.

Em sistemas que implementam a técnica de memória virtual, o espaço de endereçamento do processo é conhecido como espaço de endereçamento virtual e identifica o conjunto de endereços virtuais que o processo pode endereçar. De forma análoga, o conjunto de endereços físicos ou reais que o processador pode referenciar é chamado de espaço de endereçamento real.

O mecanismo de tradução do endereço virtual para endereço físico é conhecido como mapeamento. O espaço de endereçamento virtual não possui relação direta com os endereços no espaço real, posto que um programa pode referenciar endereços virtuais que estejam fora dos limites da memória principal. Para que isso ocorra, a memória secundária é utilizada como extensão da memória principal. Na inicialização de um processo, apenas o programa fica parcialmente residente na memória principal e o restante permanece na memória secundária até que seja referenciado. A figura a seguir ilustra ambos os espaços de endereçamento.

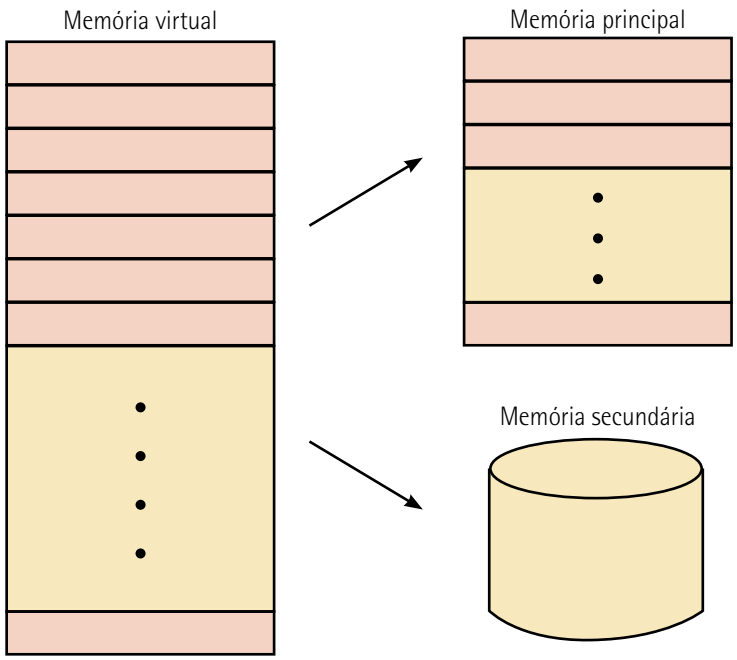

Figura 24 – Espaço de endereçamento virtual

Fonte: Machado e Maia (2013, p. 160).

Conforme Oliveira, Carissimi e Toscani (2010), a memória virtual aumenta a memória total disponível em um sistema adicionando à memória real um espaço em disco através de uma memória secundária. Essa área em disco corresponde ao que se denomina área de swap. Tal área é implementada de forma diferente dependendo do SO. Nos SOs da Microsoft, é executada por intermédio de um arquivo específico, enquanto em sistemas da Unix ocorre através de partições do disco Unix. Além da memória principal, há o espaço da área de swap.

Exemplo de aplicação

Para entender as vantagens da utilização de gerência de memória virtual, pense em uma planilha eletrônica como o Excel. Geralmente ela oferece aos usuários uma variedade de funções, sendo muitas delas raramente utilizadas. As rotinas que implementam tais funções não precisam estar na memória principal na maior parte do tempo e somente serão usadas nos raros instantes em que são necessárias.

Outro exemplo de aplicação são os componentes de um programa que tratam exceções, como erro no acesso aos arquivos. Essa parte de um código apenas será necessária quando efetivamente ocorrer um erro no acesso ao arquivo, fato que é pouco frequente.

Com o gerenciamento da memória virtual e quando cada processo ocupa exclusivamente a memória física necessária para sua execução em cada instante, ocorrerá a redução significativa do espaço ocupado na memória principal. O espaço disponível possibilitará o carregamento e a execução simultâneos de mais processos ou programas maiores poderiam ser executados. Com isso, os programas carregados na memória principal poderiam até ser maiores que a própria memória física, já que somente uma parte deles precisaria estar na memória a cada instante.

4.2 Memória virtual por paginação

Nesta técnica de gerência de memória, o espaço de endereçamento virtual e real é dividido em blocos de tamanho fixo que são denominados página. As páginas no espaço real são denominadas páginas reais ou frames, enquanto aquelas no espaço real são conhecidas como páginas virtuais.

A fim de realizar o mapeamento da memória virtual são utilizadas as tabelas de páginas, para cada processo há sua própria tabela de páginas, e cada página virtual do processo gera uma entrada na tabela de páginas (ETP), incorporando informações de mapeamento que possibilitem a localização da página real correspondente. A figura a seguir apresenta a tabela de páginas.

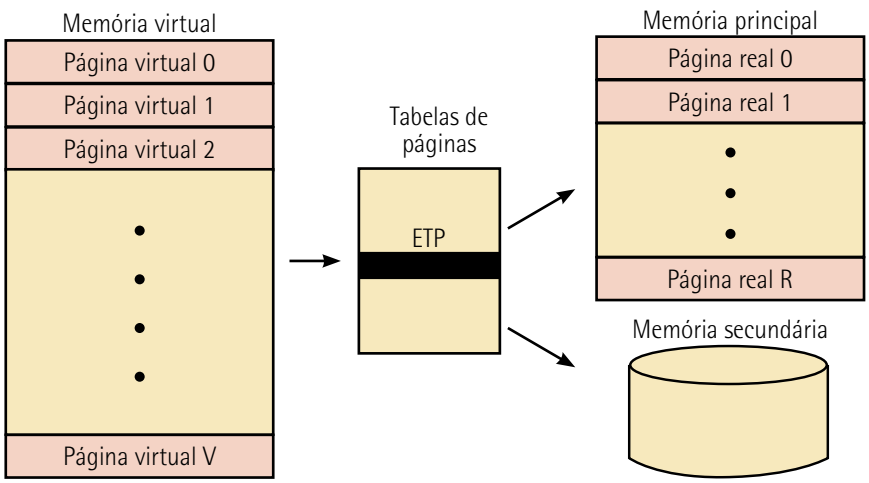

Figura 25 – Tabela de páginas

Fonte: Machado e Maia (2013, p. 164).

Por exemplo, os blocos de tamanho fixo, como 4 KB, são denominados página. O espaço de endereçamento virtual é dividido em páginas virtuais, misturando tipos de dados.

Um dos campos principais da ETP é o bit de validade ou valid bit, que indica se uma página está ou não na memória principal. Se o bit for 0, não está carregada e é gerada uma interrupção de falta de página ou page fault que desvia a execução para o SO. Neste caso, o sistema transfere a página da memória secundária para a memória principal, realizando page in, ou paginação. Se o bit for igual a 1, a página já estará carregada na memória principal.

Outros componentes da tabela são o número da página real ou page frame number, os bits de proteção (leitura, execução ou leitura/escrita), o bit de modificação, o bit de cache e o bit de referência.

A quantidade de page faults gerados por cada processo durante um intervalo é definida como taxa de paginação do processo. Caso a taxa de paginação seja muito elevada, o excesso de operações de transferência de memória ocasionará degradação no desempenho do sistema. Quando um processo em execução faz referência a um endereço e resulta em um page fault, ele irá para o estado de espera, enquanto a página não é transferida do disco rígido para a memória principal.

Durante a troca de contexto, é necessário que o SO salve as informações da tabela de mapeamento e sejam restauradas as informações para o processo escolhido no escalamento. Depois da transferência da página solicitada para a memória principal, o processo será colocado na lista de processos prontos, aguardando ser escolhido pelo escalonador para voltar à execução.

O endereço virtual é composto do número da página virtual (NPV) e de deslocamento. O NPV identifica unicamente a página virtual que contém o endereço, funcionando como um índice na tabela de páginas. Por sua vez, o deslocamento representa a posição do endereço virtual frente ao início da página na qual está armazenado.

Para a obtenção do endereço físico, combina-se o endereço do frame, localizado na tabela de páginas, com o deslocamento, presente no endereço virtual.

4.2.1 Flags de status e controle

Além do número do quadro correspondente na memória física, cada entrada de tabela de páginas possui um conjunto de flags (bits) de status ou de controle relativos à página e com finalidades específicas (Maziero, 2019). Os bits mais comuns são:

- **Valid**: indica se a página é válida, isto é, se existe no espaço de endereçamento daquele processo. Caso este bit seja 0, tentativas de acesso à página gerarão uma interrupção de falta de página ou page fault.

- **Writable**: aponta se a página pode ser acessada em leitura e escrita (1) ou somente em leitura (0).

- **User**: quando está ativo (1), o código será executado em modo usuário para acessar a página; caso contrário, a página ela somente será acessível ao núcleo do sistema.

- **Present**: indica se a página está presente na memória RAM ou se foi transferida para um armazenamento secundário, como ocorre nos sistemas com paginação em disco.

4.2.2 Política de alocação de páginas

A política de alocação de páginas define quantos frames cada processo poderá manter na memória principal. Há duas alternativas para essa política: alocação fixa ou variável. Na alocação fixa, cada processo possui um número máximo de frames que pode ser utilizado na execução do programa. Caso o número de páginas reais seja insuficiente, uma página do processo em questão é descartada para que uma nova seja carregada. Na alocação variável, o número máximo de páginas alocadas varia dependendo da taxa de paginação e da ocupação da memória principal.

4.2.3 Política de busca de páginas

Por meio de técnicas de memória virtual, é possível a execução de um código sem que este esteja carregado completamente na memória principal. Através da política de busca de páginas, determina-se quando uma página deve ser carregada na memória principal. De forma simplificada, existem duas estratégias: paginação por demanda ou paginação antecipada.

Na paginação por demanda ou demand paging, as páginas dos processos são transferidas da memória secundária para a memória principal somente quando são referenciadas. Com isso, serão levadas apenas aquelas necessárias à execução do programa. Alguns fragmentos do código como rotinas para tratamento de erros podem nunca ser carregados na memória.

Por meio da paginação antecipada ou anticipatory paging, além da página referenciada, outras páginas que podem ou não ser necessárias ao processo ao longo do processamento são carregadas na memória principal. As páginas adicionais podem não ser utilizadas durante a execução do processo.

Normalmente, a paginação antecipada é utilizada na criação de um processo. Nesse momento, diversas páginas do código na memória secundária serão armazenadas na memória principal, gerando muitos page faults e operações de leitura em disco. Conforme as páginas são carregadas para a memória principal, a taxa de paginação tem uma tendência de redução. Caso o sistema carregue apenas uma, mas um conjunto de páginas, a taxa de paginação do processo deverá cair imediatamente e se estabilizará posteriormente.

Analogamente, a paginação por antecipação é empregada em casos de ocorrência de page fault. Quando isso acontece, o sistema poderá carregar a página referenciada e páginas adicionais, visando evitar novos page faults e sucessivas operações de leitura em disco.

4.2.4 Política de substituição de páginas

A política de substituição de páginas determina como o SO seleciona qual página será liberada da memória principal para que outra a substitua. Existem duas políticas principais: política de substituição de páginas local e política de substituição de páginas global.

Na política de substituição de páginas local, sempre que um processo necessitar de uma nova página, o sistema selecionará uma página do processo em questão a ser substituída. Já na política de substituição de páginas global, todas as páginas alocadas na memória principal são candidatas à substituição, independentemente do processo que a gerou.

4.2.5 Algoritmos de substituição de páginas

Quando é utilizada a paginação no gerenciamento de memória virtual, o algoritmo de substituição de página é responsável por decidir qual página necessita ser substituída quando uma nova página chegar. Quando uma nova página é referenciada e não está presente na memória, ocorre uma falha de página e o SO substitui uma das páginas existentes pela nova necessária. Apesar de os algoritmos de substituição de página adotarem diferentes formas de decidir qual página substituir, todos os algoritmos visam à redução do número de falhas de página. Dessa forma, o algoritmo que otimiza a substituição é aquele que escolhe uma página lógica utilizada anteriormente por um programa e que não será mais necessária ou que somente será usada no futuro mais distante.

De acordo com Maziero (2019), no mundo ideal, a melhor página a ser removida da memória em um dado instante é aquela que ficará mais tempo sem ser utilizada pelos processos. Essa ideia simples define o algoritmo ótimo (OPT). Entretanto, não é viável implementar o algoritmo ótimo, posto que isso exige o conhecimento antecipado do comportamento dos processos, isto é, saber previamente o fluxo de controle para determinar o grau de necessidade das páginas. Dada essa impossibilidade, os algoritmos de substituição de páginas realizam estimativas na forma de previsão do futuro com base no passado.

Esse algoritmo nos ajuda na criação de um limite mínimo conceitual no qual para cada cadeia de referências, quando o algoritmo ótimo gerar N faltas de página, nenhum outro algoritmo resultará em menos de N faltas de página no tratamento da mesma cadeia. Dessa forma, o resultado do algoritmo ótimo é utilizado com um parâmetro de avaliação de desempenho dos demais algoritmos.

A escolha da página a ser substituída é implementada pelo algoritmo de substituição de páginas que pode ser classificado em duas classes genéricas possíveis: classe global e classe local. A classe global considera que as páginas físicas do sistema formam um conjunto único compartilhado pela totalidade dos processos. O fato que caracteriza a política como global é que a página a ser retirada da memória principal pode estar vinculada a qualquer um dos processos ativos no sistema.

Para os algoritmos locais, é realizada a análise de dois comportamentos apresentados durante a execução de processos. No primeiro, cada processo necessita de um conjunto mínimo de páginas para

executar de forma eficiente, isso é, sem provocar alta taxa de falta de página. No segundo, esse conjunto de páginas é dinâmico, isso é, evolui com a execução do processo.

O algoritmo First-In-First-Out (FIFO), traduzindo "o primeiro que entra é o primeiro que sai", mantém no SO uma fila para rastrear todas as páginas na memória, com a chegada mais recente na parte de trás (cauda da fila) e a chegada mais antiga na frente (cabeça da fila). Depois que a memória principal está sem espaço, uma página será substituída para entrada de uma nova página. Com o algoritmo FIFO, a página mais antiga estará na frente da fila e será selecionada para substituição. Segundo Córdova Jr., Ledur e Morais (2018), uma desvantagem desse algoritmo é que o aumento do número de quadros de página resulta em um incremento no número de falhas de página para um determinado padrão de acesso à memória. Este problema é conhecido como anomalia de Belady.

O algoritmo menos recentemente usado, em inglês least recently used (LRU), controla a utilização de páginas em uma janela de tempo definida. Quando é o momento de substituir uma página, ela substitui a página menos usada recentemente e que está na memória há mais tempo sem ser acessada. A ideia é que páginas mais frequentes nas últimas instruções serão mais prováveis a serem utilizadas nas próximas execuções, isto é, por um processo permanecerão a ser necessárias futuramente.

Assim, páginas antigas que continuarem em desuso pelo maior tempo são as escolhas preferenciais para a troca de páginas. Esse algoritmo possui um alto custo computacional, pois é necessário manter uma lista encadeada com todas as páginas existentes na memória, ordenadas pelo instante de referência à memória, com as mais recentes no início e as usadas há mais tempo no fim. Para cada referência à memória, é preciso atualizar a lista encadeada. O algoritmo LRU pode ser implementado tanto em software quanto em hardware. Na realização via hardware, a unidade MMU deve suportar a execução do LRU.

Outro algoritmo utilizado consiste em uma escolha aleatória das páginas a serem retiradas da memória e é denominado Random. Esse algoritmo aleatório pode ser útil em situações onde as abordagens LRU e FIFO possuam desempenho ruim, sendo aplicado em SOs antigos.

Um aperfeiçoamento do algoritmo FIFO analisa o bit de referência R de cada página candidata para verificar se esta foi acessada recentemente e é denominada algoritmo da segunda chance. Se a página não tiver sido referenciada recentemente e for a página mais antiga, o bit R será 0 e ela será trocada. Se a página tiver sido referenciada recentemente, isto é, o bit R for igual a 1, a página voltará para o fim da fila, com ajuste do bit de referência para zero. O tempo de carga é alterado, para mostrar que a página é recém-chegada à memória e que recebeu uma segunda chance. Assim, a busca irá continuar e páginas antigas que são muito acessadas não são substituídas, como aconteceria no algoritmo FIFO.

Uma melhoria ao algoritmo da segunda chance é aplicada no algoritmo do relógio ou clock. Nesse caso, existe uma lista circular com a seta do ponteiro apontado para a página mais antiga, na forma de um relógio. Quando há uma falha de página, inspeciona-se a cabeça da lista e seu bit de referência. Caso R seja igual a 0, a página da cabeça será substituída pela nova. A cabeça da seta será avançada para a próxima posição.

Caso o bit de referência R seja igual a 1, avança-se a cabeça uma posição à frente e o processo será repetido até que se encontre uma página com R igual a 0, e depois é realizado o mesmo procedimento de R igual a 0.

O algoritmo de mudança de página não usada recentemente, em inglês not recently used (NRU), apresenta melhoria em relação ao algoritmo da segunda chance, ao considerar também o bit de modificação ou dirty bit, que indica se o conteúdo de uma página foi modificado após ela ter sido carregada na memória.

Combinando os bits R (referência) e M (modificação), é possível classificar as páginas em memória em quatro níveis de importância:

- **00 (R = 0; M = 0)**: as páginas sem referências recentes e com conteúdo não modificado são as melhores candidatas à substituição, já que podem ser simplesmente retiradas da memória.

- **01 (R = 0; M = 1)**: as páginas que não foram referenciadas recentemente, mas cujo conteúdo já foi modificado não são as escolhas ideais, porque terão de ser gravadas na área de troca antes de serem substituídas.

- **10 (R = 1; M = 0)**: as páginas referenciadas recentemente e com conteúdo inalterado são provavelmente páginas de código que estão sendo usadas ativamente e têm alta probabilidade de serem referenciadas novamente no futuro.

- **11 (R = 1; M = 1)**: as páginas referenciadas recentemente e com conteúdo modificado representam a pior escolha de páginas, pois terão de ser gravadas na área de troca e provavelmente serão necessárias em breve.

O algoritmo NRU busca substituir primeiramente páginas do nível 0; caso não as encontre, procura candidatas no nível 1 e assim sucessivamente. Eventualmente, é necessário percorrer várias vezes a lista circular até encontrar uma página adequada para substituição.

4.2.6 Tamanho de página

Uma importante decisão para a técnica de paginação é a definição do tamanho de página. Ele impacta diretamente o número de entradas na tabela de páginas e, por consequência, o tamanho da tabela e o espaço ocupado na memória principal. O tamanho da página está relacionado com a arquitetura do hardware e depende do processador e normalmente está entre 512 e 16 milhões de endereços. Por exemplo, em uma arquitetura de 32 bits para endereçamento e páginas de 4 K endereços, as tabelas de páginas teriam até 220 entradas. Se cada entrada ocupasse 4 bytes, as tabelas de páginas teriam 4 MB por processo.

Assim, páginas pequenas necessitam de tabelas de mapeamento maiores, gerando maior taxa de paginação e aumentando a quantidade de acessos à memória secundária. Por outro lado, páginas grandes tornam menor o tamanho das tabelas de páginas, mas resultam no problema da fragmentação

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

O principal argumento favorável à utilização de páginas de menor tamanho é a otimização do uso da memória principal. Com páginas pequenas, a tendência é que a memória possua apenas as partes dos programas com maiores chances de serem executadas. Quanto maior o tamanho de página, as chances de ter na memória código pouco referenciado aumentam, gerando ocupação desnecessária de memória. Adicionalmente, páginas pequenas reduzem o problema da fragmentação interna.

A definição do tamanho da página também está relacionada aos tempos de leitura e gravação na memória secundária. Dado o modo de funcionamento dos discos rígidos, o tempo de operações de E/S com duas páginas de 512 bytes é muito maior do que em uma página de 1024 bytes.

De acordo com Machado e Maia (2013), a evolução dos sistemas computacionais acarretou o aumento do espaço de endereçamento e velocidade de acesso à memória principal e a tendência no projeto de SOs com memória virtual por paginação é a adoção de páginas maiores, mesmo com as dificuldades apontadas anteriormente.

4.2.7 Paginação em múltiplos níveis

Um problema crítico para sistemas que implementam apenas um nível de paginação é o tamanho das tabelas de páginas. Como apresentado anteriormente, em uma arquitetura de 32 bits para endereçamento e páginas com 4 K endereços por processo, em que cada entrada na tabela de páginas ocupe 4 bytes, a tabela de páginas poderia ter mais de 1 milhão de entradas e ocuparia 4 MB de espaço. Na figura a seguir consta a apresentação da paginação com um único nível, cuja sigla NPV significa o número da página virtual.

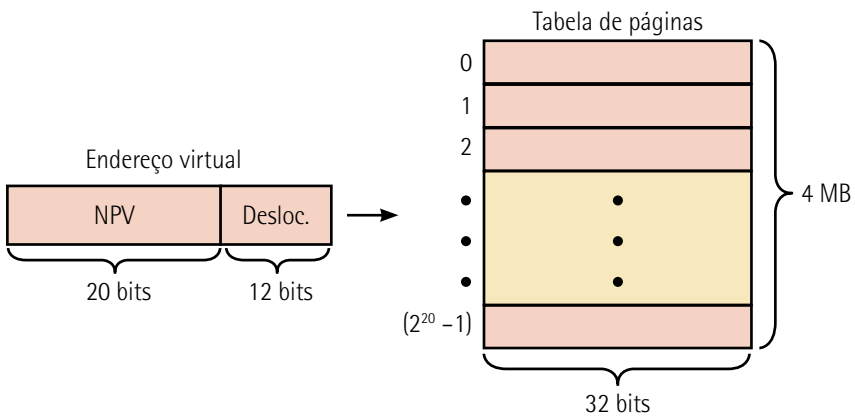

Figura 26 – Paginação em um nível

Fonte: Machado e Maia (2013, p. 176).

Reflita sobre a situação de diversos processos residentes na memória principal, como seria possível manter e gerenciar tabelas desse tamanho para cada processo?

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

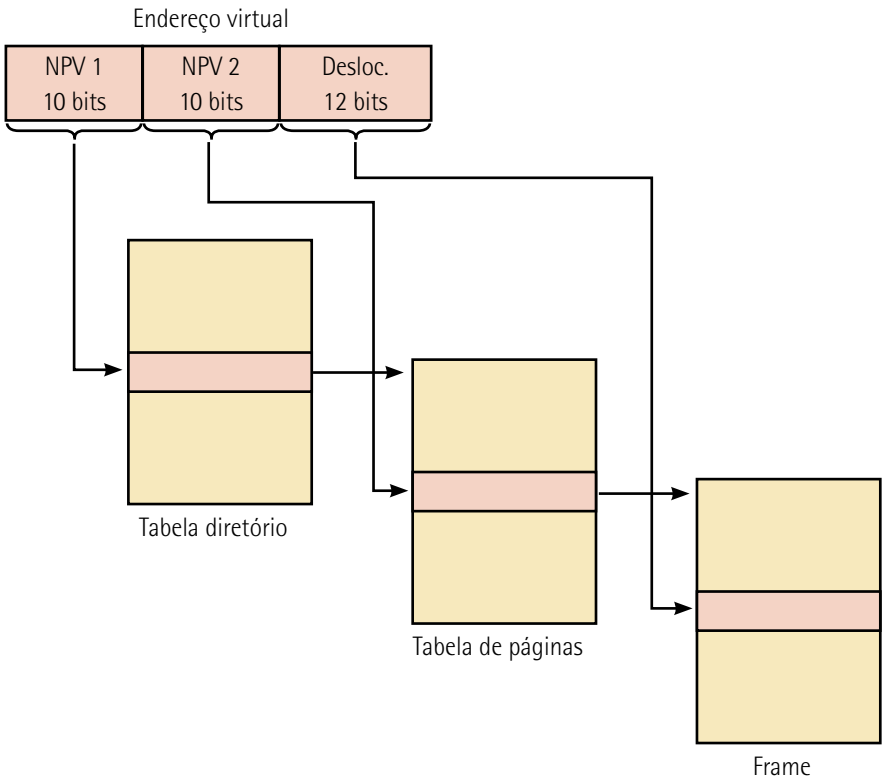

Figura 27 – Endereço virtual em dois níveis

Fonte: Machado e Maia (2013, p. 176).

O maior benefício da paginação em múltiplos níveis é a redução do espaço ocupado na memória, pois somente estarão residentes na memória principal as tabelas realmente necessárias aos processos. A técnica de paginação em múltiplos níveis pode ser estendida para três, quatro, cinco ou mais níveis.

4.3 Memória virtual por segmentação

Nesta técnica de gerência de memória virtual, o espaço de endereçamento virtual é dividido em blocos de tamanhos diferentes e arbitrários chamados de segmentos. Um programa será dividido logicamente em sub-rotinas e estruturas de dados, que são alocadas em segmentos na memória principal.

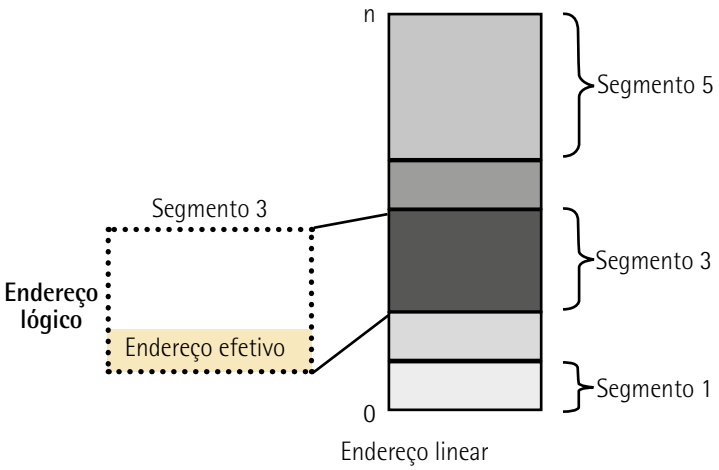

Figura 28 – Segmentação

Na figura anterior, foram apresentados cinco segmentos de tamanhos e tipos diferenciados, com destaque para o lado esquerdo, por meio da demonstração do segmento 3, que está parcialmente usado. O espaço de endereçamento virtual de um processo possui um número máximo de segmentos, e cada um deles pode possuir diferentes tamanhos com um limite definido.

Uma vantagem dessa técnica é a flexibilidade que permite que o tamanho do segmento seja alterável. Entretanto, a segmentação causa fragmentação externa, posto que há várias áreas livres na memória principal, mas nenhuma é suficiente para alocar um novo segmento.

4.3.1 Memória virtual por paginação *versus* segmentação

As técnicas de paginação e segmentação possuem diferentes características. A tabela a seguir as comparará.

Quadro 3 – Características das técnicas de paginação e segmentação

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

Adaptado de: Machado e Maia (2013, p. 184).

Visando extrair as vantagens de ambas as técnicas, foi desenvolvida a técnica de memória virtual por segmento com paginação. Nela, o espaço de endereçamento é dividido em segmentos e cada um deles será dividido em páginas.

Assim, a aplicação será mapeada em segmento de tamanhos diferentes que dependem das sub-rotinas e estrutura de dados definidas no programa. O sistema, por sua vez, tratará cada segmento como um conjunto de páginas, que possuem o mesmo tamanho. Cada segmento terá o mapeamento das páginas que está associado.

Saiba mais

Existem alguns simuladores de SO nos quais é possível entender os conceitos da gerência de memória virtual. O software SOsim, simulador para ensino de SOs, foi desenvolvido pelo Prof. Luiz Paulo Maia, da Universidade Federal do Rio do Janeiro (UFRJ), e está disponível para download no site a seguir:

MAIA, L. P. *SOsim*: Simulador para o Ensino de Sistemas Operacionais. Versão 2.0. 2007. Disponível em: https://tinyurl.com/3ps5mv53. Acesso em: 26 set. 2023.

Embora alguns processadores da família Intel suportem a segmentação, o Linux a utiliza muito pouco. Entre as razões pelas quais o Linux não explora a segmentação, estão: a gerência da paginação é mais simples que a da segmentação; nem sempre o hardware dos processadores oferece bom suporte à segmentação. Adicionalmente, em muitos casos é possível transformar a segmentação paginada em paginação pura, mapeando-se todo o espaço virtual em um único segmento.

4.4 Estrutura de tabela de páginas

Na técnica de paginação do gerenciamento de memória, existe um problema crônico com o tamanho da tabela de páginas, que será crescente em função da capacidade de memória. Dado o tamanho da memória RAM atual, há um relevante consumo de memória somente para armazenar as informações das tabelas de páginas e precisa estar na memória principal. Para melhorar a organização, foram criadas três estruturas da tabela de páginas:

- Paginação hierárquica ou multinível.
- Tabela de página em hash.
- Tabela de página invertida.

4.4.1 Paginação hierárquica

Nesta solução, a tabela de páginas será quebrada em múltiplas tabelas de páginas, com partes mais internas e outras mais externas. Dessa forma, é criada uma estrutura de tabelas de páginas multinível. Uma técnica simples é realizar a tabela de páginas em dois níveis, mantendo somente a parte da tabela necessária na memória principal.

Como apenas uma parte das páginas será armazenada na memória, haverá um consumo menor de memória principal. O endereçamento é composto de número de página PT1, da tabela mais externa, o número de página PT2, da tabela mais interna, e o de offset.

4.4.2 Tabela de página em hash

O número de página virtual é usado para função hash. Cada entrada na tabela contém uma lista ligada de elementos composta de: número da página virtual, número da moldura da página ou frame e um ponteiro para o próximo item. Esta técnica funciona bem com 32 bits de endereçamento, mas não tem um bom desempenho com 64 bits.

4.4.3 Tabela de página invertida

Nesta técnica, há a ocorrência apenas de páginas na memória física, em vez de uma entrada por página no espaço virtual. A entrada inclui tanto o processo quanto a página virtual. Existe uma economia de espaço quando a memória física é menor que a memória virtual. Utiliza-se um dispositivo de hardware para representar uma memória cache da tabela de páginas invertidas que é denominado TLB, do inglês translation lookaside buffer.

4.5 Thrashing ou atividade improdutiva

Thrashing representa uma situação do SO resultante da excessiva transferência de páginas ou segmentos entre a memória principal e a memória secundária. Essa alta atividade de paginação é denominada **atividade improdutiva** e resulta em sérios problemas de desempenho. Esse problema é decorrente da implementação de paginação ou segmentação no SO. Um processo em atividade improdutiva gasta mais tempo paginando do que em execução.

Na memória virtual por paginação, o thrashing acontece em dois níveis: no do próprio processo e no nível do SO. Analisando o nível do processo, a excessiva paginação ocorre em razão da elevada quantidade de page faults gerada pelo processo em execução. Por causa desse problema, o processo passa mais tempo esperando por páginas do que realmente sendo executado. Duas causas geram esse comportamento em um processo: dimensionamento incorreto do limite máximo de páginas do processo, que não poderá ser acomodado em seu working set, e a ausência do princípio da localidade.

O conceito de working set surgiu com o objetivo de reduzir o problema do thrashing e está ligado ao princípio da localidade. Existem dois tipos de localidade, a localidade espacial, que tem a tendência de que após uma referência a uma posição de memória sejam realizadas novas referências a endereços próximos, e a localidade temporal é a tendência de que após a referência a uma posição de memória, esta mesma posição seja novamente referenciada em um curto intervalo.

Considere o cenário a seguir, baseado no comportamento real dos primeiros sistemas de paginação. O SO monitora constantemente a utilização da CPU. Quando ela está muita baixa, aumentamos o grau de multiprogramação introduzindo um novo processo no sistema. Um algoritmo global de substituição de páginas é usado; ele substitui páginas sem se preocupar com o processo ao qual elas pertencem.

Agora suponha que um processo entre em uma nova fase de sua execução e precise de mais quadros. Ele começa a falhar e a retirar quadros de outros processos.

Porém, esses processos precisam dessas páginas e, assim, eles também falham, retirando quadros de outros processos. Esses processos com falhas precisam usar o dispositivo de paginação para inserir e remover páginas da memória. À medida que os processos se enfileiram em espera pelo dispositivo de paginação, a fila de prontos se esvazia. Enquanto os processos esperam pelo dispositivo de paginação, a utilização da CPU diminui.

O escalonador de processo ou scheduler percebe a diminuição na utilização da CPU e aumenta o grau de multiprogramação como consequência. O novo processo tenta ser executado retirando quadros de processos em execução, gerando mais erros de página e uma fila mais longa de espera pelo dispositivo de paginação. Consequentemente, a utilização da CPU cai ainda mais, e o scheduler da CPU tenta aumentar ainda mais o grau de multiprogramação. A atividade improdutiva ocorreu e o throughput do sistema despenca. A taxa de erros de página aumenta enormemente. Como resultado, o tempo de acesso efetivo à memória sobe. Nenhum trabalho está sendo executado porque os processos gastam todo o seu tempo paginando.

Esse fenômeno é ilustrado na figura a seguir, em que a utilização da CPU é representada graficamente em relação ao grau de multiprogramação. Conforme o grau de multiprogramação aumenta, a utilização da CPU também cresce, embora mais lentamente, até que um nível máximo seja alcançado. Se o grau de multiprogramação aumentar ainda mais, a atividade improdutiva se instala e a utilização da CPU cai significativamente. Nesse ponto, para aumentar o uso da CPU e interromper a atividade improdutiva, devemos diminuir o grau de multiprogramação.

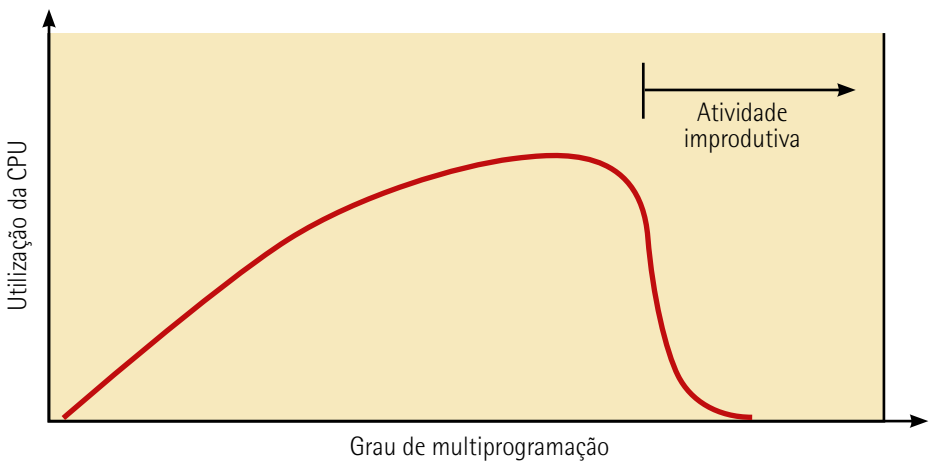

Figura 29 – Atividade improdutiva ou thrashing

Fonte: Silberschatz, Galvin e Gagne (2015, p. 231).

É possível limitar os efeitos da atividade improdutiva através de um algoritmo de substituição local, também conhecido como algoritmo de substituição por prioridades. Com a substituição local, se um processo começa a sofrer de atividade improdutiva, ele não pode roubar quadros de outro processo,

causando também a atividade improdutiva desse outro processo. No entanto, o problema não é totalmente resolvido. Se os processos estão em atividade improdutiva, eles ficam a maior parte do tempo na fila do dispositivo de paginação. O tempo médio de manipulação de um erro de página aumentará em razão do crescimento na fila média de espera pelo dispositivo de paginação. Portanto, o tempo de acesso efetivo aumentará até mesmo para um processo que não esteja em atividade improdutiva.

Para impedir a ocorrência de atividade improdutiva, devemos fornecer ao processo quantos quadros ele precisar.

Lembrete

Thrashing representa uma situação do SO resultante da excessiva transferência de páginas ou segmentos entre a memória principal e a memória secundária.

4.6 Paginação em sistemas Linux

O Linux emprega um sistema de paginação em três níveis para traduzir endereços virtuais presentes nas páginas lógicas para endereços reais, que são encontrados nas páginas físicas, ou frames. Nesse modelo, podem ser utilizados três tipos de tabelas de páginas: diretório global de páginas, diretório intermediário de páginas e a própria tabela de páginas. Cada uma delas possui entradas que indicam para a tabela o próximo nível hierárquico, quando uma entrada do diretório global de páginas aponta para um diretório intermediário de páginas, e, por sua vez, as entradas desse apontam para diferentes tabelas de páginas. O endereço da tabela que compõe o diretório global de páginas é fornecido por um apontador à parte, normalmente implementado por um registrador especial do processador.

Um motivo pelo qual o Linux emprega um esquema de paginação em três níveis é seu objetivo de portabilidade a diferentes plataformas. Processadores de 64 bits, por exemplo, os Alpha, normalmente são projetados para realizar uma tradução de endereço lógico em endereço físico empregando três níveis de indireção. Nos processadores de 32 bits, como a família Intel 80x86, o mecanismo de tradução implementado pelo hardware considera apenas dois níveis. Visando fornecer suporte a ambas as arquiteturas de processadores, o Linux considera sempre uma paginação em três níveis, adaptando-a logicamente para as arquiteturas de processadores que empregam somente dois níveis.

Na paginação em três níveis, inicialmente, realiza-se a leitura do conteúdo de uma entrada do diretório global de páginas. A entrada a ser acessada é obtida, indexando-se essa tabela com o valor do campo Nível_1. O conteúdo dessa entrada fornece um apontador para uma tabela do segundo nível hierárquico, isto é, um repositório intermediário de páginas. De forma semelhante, é utilizado o valor do campo Nível_2 para fornecer um índice que é empregado para fornecer o endereço inicial de uma tabela de páginas. Após isso, o campo Nível_3 será aplicado para selecionar uma entrada da tabela de páginas. Entre as informações mantidas na tabela de páginas, está o endereço da página física equivalente a esse endereço virtual, que, somado ao valor correspondente ao campo deslocamento, resultará na posição de memória referenciada pelo endereço virtual.

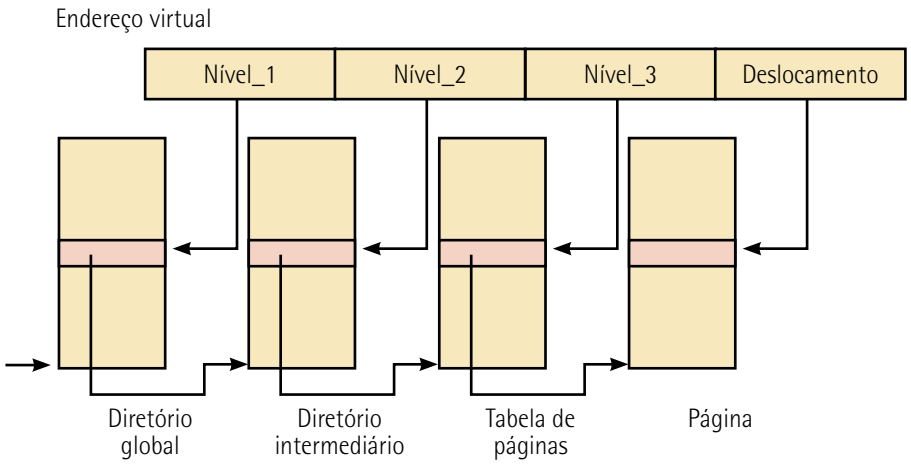

Figura 30 – Paginação em três níveis

Fonte: Oliveira, Carissimi e Toscani (2010, p. 258).

Um processo Linux possui um modelo de memória organizado em quatro partes: texto, dados não inicializados, dados inicializados e pilha. A área de texto corresponde ao código do programa. A área de dados, sejam inicializados ou não, é composta do espaço de armazenamento necessário às variáveis alocadas estaticamente no programa. Na área de pilha, é fornecido o espaço de memória necessário às variáveis automáticas (locais), para a passagem de parâmetros, e ainda para armazenar e recuperar endereços de retorno dando suporte a sub-rotinas.

As áreas de texto e de dados são armazenadas na parte baixa de memória, e a pilha na parte superior, e a pilha cresce em direção aos endereços inferiores. O espaço existente entre a pilha e os dados é denominado heap. A partir dessa área, os espaços de memória são alocados dinamicamente.

Lembrete

A quantidade de espaço de endereçamento está relacionada com o número de bits que endereçará a memória. Considerando n como o número de bits de endereçamento, o espaço total da memória é dado por:

Espaço de endereçamento = 2n.

Nesse modelo, existem 32 bits para seleção do endereço de memória, assim, o espaço de endereçamento de um processo qualquer é de 4 gigabytes ou (2^{32}), o que corresponde à capacidade total de endereçamento dos processadores de 32 bits.

O conceito de programa executável é utilizado para alocação de processos na memória. Um executável, similarmente ao modelo de processo, é composto de uma área de código, uma área de dados e um conjunto de informações necessárias para carregar esse código executável na memória. O executável possui um formato interno que descreve os tamanhos de cada parte do processo, não

sendo necessário um espaço de armazenamento idêntico ao tamanho do processo, isso é, 4 gigabytes. O problema agora consiste em mapear o executável no modelo de memória de processo, e esse, por sua vez, na memória física da máquina.

Outro ponto relevante é que o processo não necessita acessar todo o seu código nem todos os seus dados durante todo o período em que está em execução ou em espera. Consequentemente, ocorre perda de espaço de memória se todo o código e todos os dados forem mantidos na memória. Se lembramos que o Linux é um sistema multiprogramado e possibilita que múltiplos processos sejam executados praticamente de forma simultânea, tal desperdício fica multiplicado pelo número de processos em execução. Como a gerência de memória do Linux é baseada em paginação, a memória virtual de um processo é dividida em páginas, e esse problema de desperdício é resolvido com a técnica de paginação por demanda.

Através da paginação por demanda, o gerenciamento de memória do SO busca a manutenção em memória somente dos pedaços de código, dados e pilha que um processo qualquer utiliza em determinado período. Assim, em vez de carregar toda a área de código e de dados de um processo, o núcleo do Linux mantém apenas uma estrutura interna que descreve a memória virtual do processo, indicando quais partes estão carregadas na memória e quais estão armazenadas em disco. Quando um processo necessita acessar uma parte que não está na memória, é gerada uma exceção, denominada page fault, que é tratada pelo SO. Esse tratamento consiste basicamente em carregar as páginas necessárias na memória. Quando não há mais espaço suficiente na memória de trabalho, o sistema de gerenciamento de memória realiza um procedimento de substituição de páginas, isto é, o swapping.

4.6.1 Swapping no Linux

Imagine que durante a execução de um processo seja necessário carregar uma nova página em memória e não haja mais espaço disponível na memória. O SO realizará um procedimento de troca de memórias denominado swapping, retirando uma página carregada na memória principal, e substituindo por uma página que precisa ser carregada.

No Linux, o termo correto a ser aplicado é pagging, e o nome das movimentações de E/S de páginas são page in e page out, respectivamente. Daemon page é o nome do procedimento realizado por um processo interno do Linux. O algoritmo utilizado para selecionar a página a ser substituída é o LRU, combinado com o algoritmo de relógio ou clock. Para agilizar esse procedimento, algumas otimizações são consideradas. Por exemplo, se a página a ser substituída não foi alterada, ela não é reescrita em disco, pois ele já possui uma cópia atualizada. Caso a página tenha sido alterada, então ela será reescrita em disco, mas em um arquivo especial: a área de swap.

4.7 Gerenciamento de memória no Android

Em Android, utilizam-se a paginação e o mapeamento em memória ou mapping para gerenciamento da memória, tanto no Android ART quanto na máquina virtual Dalvik. Conforme Android (2023), qualquer memória que um app modifique, através de alocação de novos objetos ou de acesso às páginas mapeadas, será armazenada na RAM e não poderá ser excluída da memória. Para desalocar a memória

de um aplicativo, é necessária a eliminação das referências de objetos gerados pelo aplicativo, de forma a disponibilizar a memória para o coletor de lixo.

Um ambiente de memória gerenciado no Android monitora cada alocação de memória. Depois que a escolha do espaço da memória não está mais sendo usada pelo programa, ela é liberada de volta para o heap, sem a necessidade de intervenção do programador. Para liberação de memória não utilizada em um ambiente de memória gerenciada, é realizada a coleta de lixo ou garbage collection. Tal processo possui a função de encontrar objetos de dados em um programa que não pode ser acessado posteriormente e liberar os recursos utilizados por esses objetos.

O coletor de lixo divide o espaço com base na vida útil esperada e no tamanho de um objeto alocado, separando em gerações. Por exemplo, objetos alocados recentemente são classificados como geração jovem. Quando um objeto permanece ativo por tempo suficiente, ele pode ser promovido a uma geração mais antiga, seguida por uma geração permanente. Cada geração de heap possui um limite máximo dedicado de memória que seus objetos podem ocupar.

Quando uma geração começa a ser preenchida, o sistema executa um evento de coleta de lixo para tentar liberar memória. A duração da coleta de lixo varia em função da geração de objetos que está sendo coletado e da quantidade de objetos ativos em cada geração.

Mesmo que a coleta de lixo seja realizada rapidamente, esta pode afetar a performance do app, já que toma uma proporção significativa do tempo total de processamento do programa. Em geral, o programador não consegue controlar manualmente, com o código do programa, quando um evento de coleta de lixo ocorre, uma vez que o Android realizará esse gerenciamento.

O Android utiliza um conjunto de critérios para determinar quando a coleta de lixo deve ser feita. Quando os critérios são atendidos, ele interrompe a execução do processo e inicia a coleta de lixo. A coleta de lixo pode ocorrer durante um processamento intensivo, como uma animação ou durante a reprodução de uma música, e com isso o tempo de processamento da coleta de lixo pode aumentar.

Uma característica fundamental para o Android é possibilitar a realização de múltiplas tarefas simultaneamente, isto é, ser um sistema multitarefa. Com objetivo de manter o ambiente multitarefa, ele atribui um limite rígido para o tamanho do heap de cada app, que varia entre dispositivos com base na quantidade de RAM disponível para cada aparelho. Quando o aplicativo alcança a capacidade de heap e busca alocar mais memória, ele pode encontrar um OutOfMemoryError, um erro que se dá quando a máquina virtual Java não pode alocar um objeto pois está com falta de memória e nenhuma memória está disponível pelo coletor de lixo.

No ambiente multitarefa, os usuários podem alternar a utilização dos aplicativos abertos e, quando isso ocorre, o Android mantém em um cache os apps que não estão em primeiro plano, ou seja, não estão visíveis para o usuário ou estão executando serviços de primeiro plano, como reprodução de músicas. Por exemplo, quando o usuário executar um aplicativo pela primeira vez, ocorre a criação de um processo para este app. Enquanto o usuário não finalizar a execução, o processo não é encerrado,

permanecendo na memória do sistema. Quando o usuário retomar o uso do aplicativo, o sistema reutilizará o processo, resultando em alternância de apps mais rápida.

Quando uma aplicação tiver um processo armazenado na memória e detiver recursos desnecessários no momento, mesmo que o usuário não os utilize, o desempenho geral do sistema será afetado. Quando o sistema estiver sem recursos, como memória, ele encerrará processos no cache, considerando quais processos estão alocando a maior parte da memória para liberar a memória RAM.

4.8 Gerenciamento de memória no iOS

O sistema iOS é multitarefa e permite alternar entre as aplicações em execução sem a necessidade de seu fechamento. Com isso, os dados das aplicações permanecem em memória até que exista a necessidade de liberação de recursos ou ocorra o fechamento explícito por parte do usuário.

Caso não exista espaço de memória suficiente para a execução do sistema e das aplicações em execução, o SO procura recursos que já não estão sendo utilizados e tenta recuperá-los. Quando essa ação não for suficiente, é realizada uma varredura nas aplicações em segundo plano e aquelas que ocupam mais recursos e não são usadas pelo usuário há mais tempo são finalizadas. Esse processo continua enquanto a quantidade necessária de memória não for suficiente.

Para os dispositivos que utilizam esse SO, a memória total é dividida de maneira que permita ao sistema o carregamento acelerado de novas aplicações e restauração da execução de uma aplicação em segundo plano de maneira otimizada.

A fim de possibilitar essas atividades, são criadas algumas áreas em memória que realizam um link lógico entre a memória física e o código carregado. O processo de inclusão das informações na tabela de páginas ou page table é denominado page out, e acontece quando os dados não existirem na tabela. O processo de recuperação dessas informações é conhecido como page in, no qual o gerenciador da memória virtual verifica o endereço correspondente na tabela e retorna essa informação para o processo que a solicitou.

Resumo

Esta unidade apresentou conceitos de gerenciamento de memória aplicados aos sistemas operacionais e a utilização das técnicas de gerenciamento de memória em sistemas operacionais, como técnicas de memória virtual.

Apesar da quantidade de memória dos dispositivos ter aumentado de forma relevante, o gerenciamento dela é um elemento essencial para que um sistema multitarefa funcione de forma a não acarretar lentidão em função de transferências entre memórias.

Observamos que as técnicas de memória virtual são fundamentais para os SOs, e a paginação e segmentação são exemplos de técnicas frequentes.

Na paginação, o mapeamento da memória ocorre com blocos de memória de tamanhos fixos, conhecidos como páginas. São aplicadas diversas políticas para essa técnica a fim de alocar, buscar e substituir páginas.

Exercícios

Questão 1. O gerenciamento de memória é uma das funções críticas desempenhadas por um SO. Esse recurso é essencial para otimizar o desempenho do sistema e evitar problemas, como vazamento de memória e falhas de alocação. Entre as principais tarefas realizadas pelo SO no gerenciamento de memória, estão a alocação de memória, o controle de acesso e o swapping. Nesse contexto, avalie as asserções e a relação proposta entre elas.

I – A técnica de swapping permite ao sistema operacional enviar um processo para a memória secundária.

porque

II – O loader é o utilitário do SO com a função de carregar, na memória principal (RAM), um programa para ser executado.

Assinale a alternativa correta.

A) As asserções I e II são verdadeiras, e a asserção II justifica a I.

B) As asserções I e II são verdadeiras, e a asserção II não justifica a I.

C) A asserção I é verdadeira, e a II é falsa.

D) A asserção I é falsa, e a II é verdadeira.

E) As asserções I e II são falsas.

Resposta correta: alternativa B.

Análise das asserções

I – Asserção verdadeira.

Justificativa: a técnica de swapping envolve a transferência de processos ou de partes de processos da memória principal para um dispositivo de armazenamento secundário, como um disco rígido.

II – Asserção verdadeira.

Justificativa: o loader (ou carregador) é um utilitário do SO responsável por carregar programas da memória secundária para a memória principal (RAM) antes de serem executados.

Embora ambas as asserções estejam no contexto de gerenciamento de memória e de carregamento de programas na memória principal, não há uma relação de causa e efeito entre elas. A asserção I descreve corretamente a técnica de swapping, enquanto a II acentua o papel do loader do SO.

Questão 2. (Instituto AOCP/2018, adaptada) A memória virtual confere capacidades de execução de programas maiores do que a memória física da máquina, utilizando o disco rígido para essa extensão de memória e movendo peças entre memória RAM e disco. Sobre memória virtual, avalie as afirmativas.

I – Uma técnica de memória virtual é o defrag do sistema de arquivos.

II – A paginação é uma técnica de memória virtual. Em qualquer computador existe um conjunto de endereços de memória que os programas podem gerar ao serem executados. Esses endereços são denominados endereços virtuais e constituem o espaço de endereçamento virtual.

III – A memória virtual não tem utilidade em sistemas com multiprogramação.

É correto o que se afirma em:

A) II, apenas.

B) III, apenas.

C) I e III, apenas.

D) II e III, apenas.

E) I, II e III.

Resposta correta: alternativa A.

Análise das afirmativas

I – Afirmativa incorreta.

Justificativa: defrag (ou desfragmentação) do sistema de arquivos é uma técnica utilizada para otimizar o acesso a arquivos no disco rígido. A desfragmentação consiste em reorganizar os dados de um disco para que os arquivos sejam armazenados de forma contígua, em vez de ficarem fragmentados em vários pedaços dispersos. Essa técnica não está diretamente relacionada à memória virtual.

II – Afirmativa correta.

Justificativa: a paginação é, de fato, uma técnica de memória virtual que envolve dividir a memória em pequenas unidades, chamadas de páginas, e mover essas páginas entre a memória RAM e o

armazenamento secundário, conforme necessário. O uso de endereços virtuais permite que os programas acessem uma quantidade maior de memória primária do que está fisicamente disponível no hardware.

III – Afirmativa incorreta.

Justificativa: a memória virtual é útil em sistemas com multiprogramação, pois permite que vários programas sejam executados de maneira eficiente, compartilhando a memória primária física disponível e alocando dinamicamente espaço em disco, conforme a necessidade de acomodar os processos em execução. Desse modo, a memória virtual ajuda a otimizar o uso da memória em sistemas multiprogramados. O Linux, que é multiprogramado, emprega um sistema de paginação em três níveis.