# Unidade IV

## 7 GERENCIAMENTO DE MEMÓRIA

A memória é essencial para a operação de um sistema de computação moderno e consiste em um grande array de bytes, cada um com seu próprio endereço. A CPU extrai instruções da memória, de acordo com o valor do contador do programa. Essas instruções podem causar carga adicional a partir de endereços específicos e armazenamento em endereços específicos da memória.

Um ciclo de execução de instruções típico, por exemplo, primeiro extrai uma instrução da memória. A instrução é então decodificada e pode causar a extração de operandos da memória. Após a instrução ter sido executada sobre os operandos, os resultados podem ser armazenados de volta na memória. A unidade de memória vê apenas um fluxo de endereços de memória; ela não sabe como eles são gerados (pelo contador de instruções, por indexação, indiretamente, como endereços literais, e assim por diante) ou para que eles servem (instruções ou dados). Da mesma forma, podemos ignorar como um programa gera um endereço de memória. Neste livro-texto, estamos interessados apenas na sequência de endereços de memória gerados pelo programa em execução. Assim, teremos como foco questões que permeiam tópicos, tais quais: o hardware básico, a vinculação de endereços simbólicos da memória a endereços físicos reais e a distinção entre endereços lógicos e físicos.

### Hardware básico

A memória principal e os registradores embutidos dentro do próprio processador são o único espaço de armazenamento de uso geral que a CPU pode acessar diretamente. Há instruções de máquina que usam endereços da memória como argumentos, mas nenhuma que use endereços de disco. Portanto, quaisquer instruções em execução e quaisquer dados que estiverem sendo usados pelas instruções, devem estar em um desses dispositivos de armazenamento de acesso direto. Se os dados não estiverem na memória, devem ser transferidos para lá antes que a CPU possa operar sobre eles.

Os registradores que estão embutidos na CPU geralmente podem ser acessados dentro de um ciclo do relógio da CPU. A maioria das CPUs pode decodificar instruções e executar operações simples sobre o conteúdo dos registradores à taxa de uma ou mais operações por tique do relógio. O mesmo não pode ser dito da memória principal, que é acessada por uma transação no bus da memória. Para completar um acesso à memória, podem ser necessários muitos ciclos do relógio da CPU. Nesses casos, o processador normalmente precisa ser interrompido, já que ele não tem os dados requeridos para completar a instrução que está executando. Essa situação é intolerável devido à frequência de acessos à memória. A solução é adicionar uma memória rápida entre a CPU e a memória principal, geralmente no chip da CPU para acesso rápido. Para gerenciar um cache embutido na CPU, o hardware acelera automaticamente o acesso à memória sem nenhum controle do sistema operacional.

Além de estarmos preocupados com a velocidade relativa do acesso à memória física, também devemos assegurar a operação correta. Para a operação apropriada do sistema, devemos proteger o sistema operacional contra o acesso de processos de usuário. Em sistemas multiusuários, devemos adicionalmente proteger os processos de usuário uns dos outros. Essa proteção deve ser fornecida pelo hardware, porque o sistema operacional não costuma intervir entre a CPU e seus acessos à memória, em razão do comprometimento do desempenho. O hardware implementa essa proteção de várias maneiras diferentes. A Figura 91 apresenta uma implementação possível.

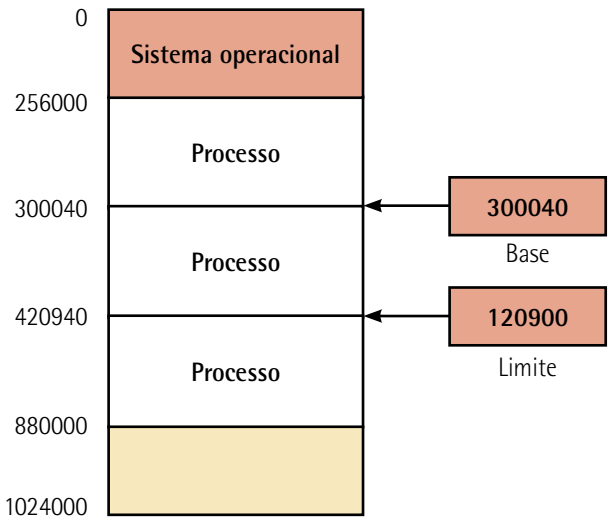



Figura 91 – Um registrador base e um registrador limite definem um espaço de endereçamento lógico

Fonte: Silberschartz (2015, p. 289).

Primeiro, precisamos nos certificar de que cada processo tenha um espaço de memória separado. Esse espaço protege os processos uns contra os outros e é fundamental para que existam múltiplos processos carregados na memória para execução concorrente. Para separar os espaços de memória, precisamos ser capazes de determinar o intervalo de endereços legais que o processo pode acessar e assegurar que ele acesse somente esses endereços. É possível fornecer essa proteção usando dois registradores, usualmente um registrador base e um registrador limite, como ilustrado na figura 91. O registrador base contém o menor endereço físico de memória legal, já o registrador limite especifica o tamanho do intervalo. Por exemplo, se o registrador base contém 300040 e o registrador limite é igual a 120900, o programa pode acessar legalmente todos os endereços de 300040 a 420939.

A proteção do espaço da memória é fornecida pela comparação que o hardware da CPU faz entre cada endereço gerado em modalidade de usuário e os registradores. Qualquer tentativa de um programa, ao executar em modalidade de usuário, de acessar a memória do sistema operacional ou a memória de outros usuários, resulta em uma interceptação para o sistema operacional que trata a tentativa como um erro fatal (figura 92). Esse esquema impede que um programa de usuário modifique, acidental ou deliberadamente, o código ou as estruturas de dados do sistema operacional ou de outros usuários.

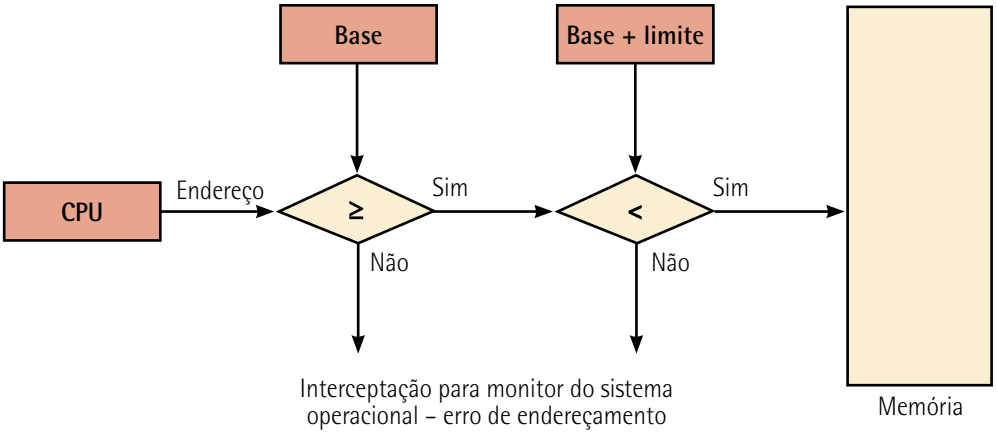



Figura 92 – Proteção de endereço de hardware com registradores base e limite

Fonte: Silberschartz (2015, p. 290).

Os registradores base e limite podem ser carregados apenas pelo sistema operacional que usa uma instrução privilegiada especial. Já que instruções privilegiadas podem ser executadas apenas em modalidade de kernel, e como somente o sistema operacional é executado nessa modalidade, apenas o sistema operacional pode carregar os registradores base e limite. Esse esquema permite que o valor dos registradores sejam alterados pelo sistema operacional, mas impede que programas de usuário alterem seus conteúdos.

O sistema operacional, sendo executado em modalidade de kernel, tem acesso irrestrito tanto à sua memória quanto à memória dos usuários. Essa característica permite que ele carregue programas de usuário na memória dos usuários, os descarregue em caso de erros, acesse e modifique parâmetros de chamadas de sistema, execute I/O a partir e para a memória do usuário, e forneça muitos outros serviços. Consideremos, por exemplo, que o sistema operacional de um sistema com multiprocessamento deve executar mudanças de contexto, transferindo o estado de um processo dos registradores para a memória principal, antes de carregar o contexto do próximo processo da memória principal para os registradores.

### Vinculação de endereços

Usualmente, um programa reside em um disco como um arquivo binário executável. Para ser executado, o programa deve ser trazido para a memória e inserido dentro de um processo. Dependendo do esquema de gerenciamento da memória em uso, o processo pode ser movimentado entre o disco e a memória durante sua execução. Os processos em disco que estão esperando para serem trazidos à memória para execução formam a fila de entrada.

O procedimento normal em ambientes de monotarefa é selecionar um dos processos na fila de entrada e carregá-lo na memória. Enquanto o processo é executado, ele acessa instruções e dados na memória. Em determinado momento, o processo termina, e seu espaço na memória é declarado disponível.

A maioria dos sistemas permite que os processos de usuário residam em qualquer parte da memória física. Portanto, embora o espaço de endereçamento do computador possa começar em 00000, o primeiro endereço do processo de usuário não precisa ser 00000. Veremos, posteriormente, como um programa de usuário coloca realmente um processo na memória física. Na maioria dos casos, um programa de usuário percorre vários passos, alguns dos quais podem ser opcionais, antes de ser executado (figura 93). Assim, os endereços podem ser representados de diferentes maneiras durante esses passos.

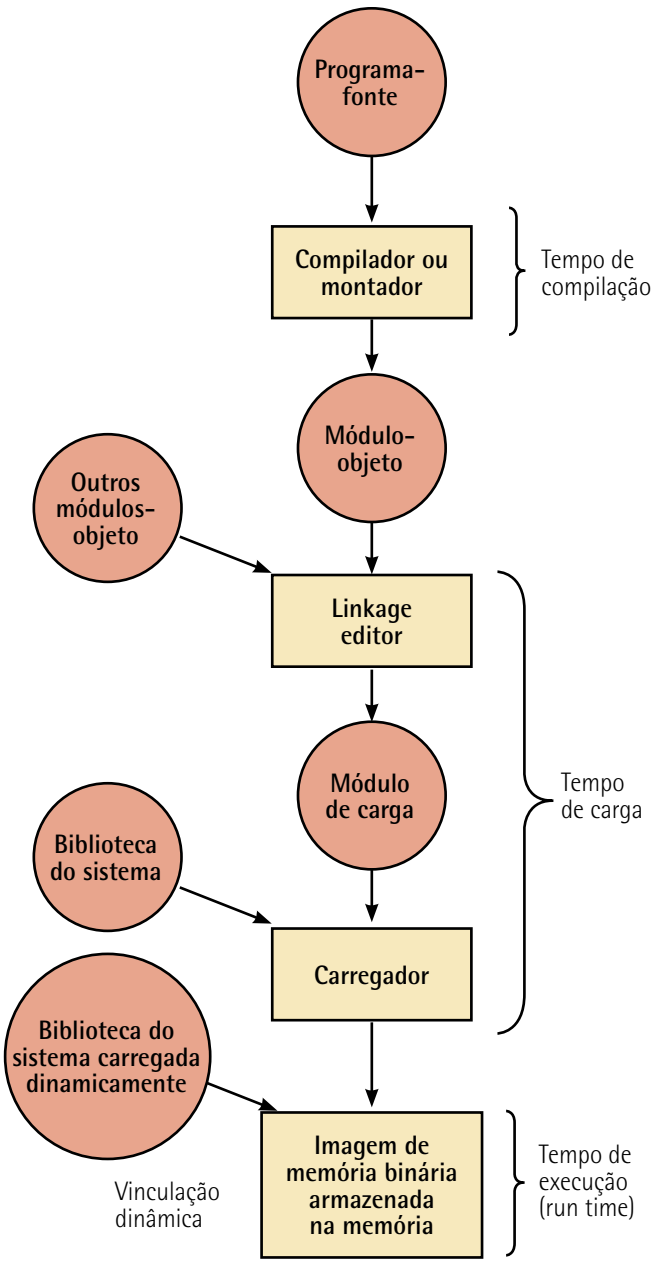



Figura 93 – Processamento de um programa de usuário em vários passos

Fonte: Silberschartz (2015, p. 291).

Os endereços do programa-fonte são, em geral, simbólicos (como a variável count). Normalmente, um compilador vincula esses endereços simbólicos a endereços relocáveis, como "14 bytes a partir do começo desse módulo". Por sua vez, o linkage editor ou carregador vincula os endereços relocáveis a endereços absolutos, como 74014. Cada vinculação é um mapeamento de um espaço de endereçamento para outro. Classicamente, a vinculação de instruções e dados a endereços da memória pode ser feita em qualquer passo ao longo do percurso, considerando:

- **Tempo de compilação**: se você souber onde o processo residirá na memória, um código absoluto pode ser gerado. Por exemplo, se você souber que um processo de usuário residirá a partir da locação *R*, então o código gerado pelo compilador começará nessa locação e se estenderá a partir daí. Se, em algum momento mais tarde, a locação inicial mudar, então será necessário recompilar esse código. Os programas no formato COM do MS-DOS são vinculados em tempo de compilação.

- **Tempo de execução**: se o processo puder ser movido de um segmento de memória para outro durante sua execução, então a vinculação deverá ser adiada até o tempo de execução. A maioria dos sistemas operacionais de uso geral emprega esse método.

**Observação**

Se o tempo de carga não for conhecido em tempo de compilação, onde o processo reside na memória, o compilador deverá gerar um código relocável. Nesse caso, a vinculação final é adiada até o tempo de carga. Se o endereço inicial mudar, precisaremos apenas recarregar o código do usuário para incorporar esse valor alterado.

## 7.1 Conceituação

Memória virtual é uma técnica sofisticada e poderosa de gerência de memória, em que as memórias principal e secundária são combinadas dando ao usuário a ilusão de existir uma memória muito maior que a capacidade real da memória principal. O conceito de memória virtual fundamenta-se em não vincular o endereçamento feito pelo programa dos endereços físicos da memória principal. Desta forma, programas e suas estruturas de dados deixam de estar limitados ao tamanho da memória física disponível, pois podem possuir endereços associados à memória secundária.

Outra vantagem da técnica de memória virtual é permitir um número maior de processos compartilhando a memória principal, já que apenas partes de cada processo estarão residentes. Isto leva a uma utilização mais eficiente também do processador. Além disso, essa técnica possibilita minimizar o problema da fragmentação da memória principal.

A primeira implementação de memória virtual foi realizada no início da década de 1960, no sistema Atlas, desenvolvido na Universidade de Manchester (Kilburn *et al.*, 1961). Posteriormente, a IBM introduziria este conceito comercialmente na família System/370, em 1972. Atualmente, a maioria dos sistemas implementa memória virtual, com exceção de alguns sistemas operacionais de supercomputadores.

Existe um forte relacionamento entre a gerência da memória virtual e a arquitetura de hardware do sistema computacional. Por motivos de desempenho, é comum que algumas funções da gerência de memória virtual sejam implementadas diretamente no hardware. Além disso, o código do sistema operacional deve levar em consideração várias características específicas da arquitetura, especialmente o esquema de endereçamento do processador.

Apresentaremos conceitos relacionados à memória virtual e abordaremos as três técnicas que permitem sua implementação: paginação, segmentação e segmentação com paginação. Apesar da ênfase na gerência de memória virtual por paginação, grande parte dos conceitos apresentados também podem ser aplicados nas técnicas de segmentação.

O conceito de memória virtual se aproxima muito da ideia de um vetor, existente nas linguagens de alto nível. Quando um programa faz referência a um elemento do vetor, não há preocupação em saber a posição de memória daquele dado. O compilador se encarrega de gerar instruções que implementam esse mecanismo, tornando-o totalmente transparente ao programador.

## 7.2 Swapping

Um processo deve estar na memória para ser executado. No entanto, ele pode ser transferido temporariamente da memória principal para uma memória de retaguarda e, então, trazido de volta à memória principal para continuar a execução (figura 94). A permuta torna possível que o espaço de endereçamento físico de todos os processos exceda a memória física real do sistema, aumentando assim o grau de multiprogramação no sistema.

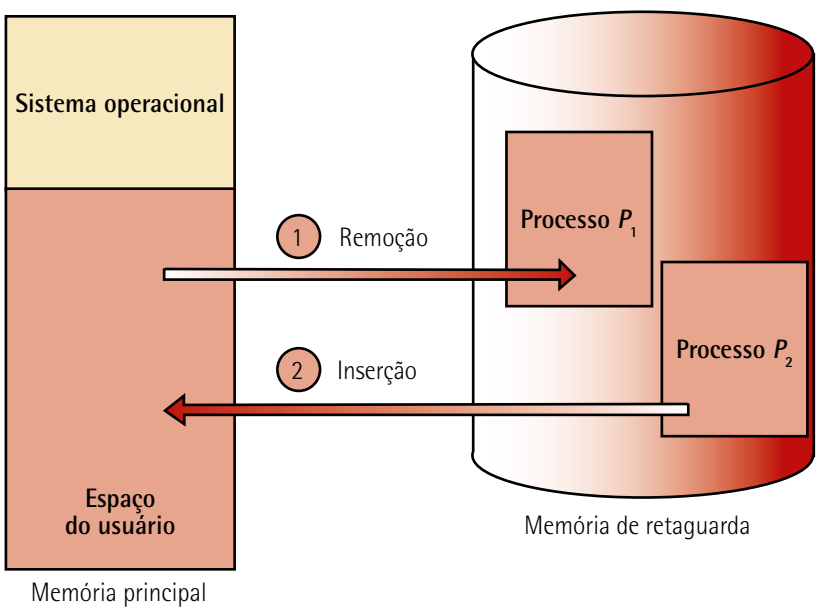



Figura 94 – Permuta de dois processos usando um disco como memória de retaguarda

Fonte: Silberschartz (2015, p. 294).

A paginação por demanda é uma técnica de gerenciamento de memória que permite que um processo carregue apenas as páginas necessárias em memória física, enquanto outras páginas permanecem armazenadas em disco. Isso reduz a quantidade de memória física necessária para executar um processo e aumenta a capacidade multitarefa do sistema. Quando um processo tenta acessar uma página que não está na memória física, ocorre uma falta de página (page fault). O sistema operacional então carrega a página necessária do disco para a memória física, o que pode resultar em uma latência significativa.

A memória de um computador é organizada em uma hierarquia com diferentes níveis de velocidade e custo. O nível mais rápido e caro é o cache, que é usado para armazenar dados e instruções frequentemente acessados. O próximo nível é a memória principal (RAM), que é mais lenta e menos cara do que o cache. O nível mais lento e menos caro é o armazenamento secundário (disco rígido), que é usado para armazenar dados que não estão sendo ativamente usados. A paginação por demanda usa a memória principal e o armazenamento secundário para gerenciar os dados de um processo.

Os padrões de acesso à memória descrevem como um processo acessa os dados em memória. Esses padrões podem variar dependendo do tipo de aplicação, do algoritmo usado e de outros fatores. Existem dois padrões principais de acesso à memória: **localidade espacial** e **localidade temporal**. A localidade espacial ocorre quando um processo acessa dados em locais adjacentes na memória; enquanto a localidade temporal ocorre quando um processo acessa os mesmos dados repetidamente em um curto período de tempo. Esses padrões são importantes para o desempenho da paginação por demanda, pois podem ajudar a minimizar o número de faltas de página.

**Impacto dos padrões de acesso na paginação por demanda**

Se um processo exibe alta localidade espacial ou temporal, a paginação por demanda pode funcionar muito bem, pois o sistema operacional pode manter as páginas necessárias na memória física e evitar a necessidade de carregar páginas do disco. No entanto, se um processo exibe baixa localidade, a paginação por demanda pode resultar em um número significativo de faltas de página, levando a uma latência significativa e um desempenho lento. Isso ocorre porque, com baixa localidade, é mais provável que um processo acesse dados em diferentes partes da memória, exigindo que o sistema operacional carregue muitas páginas diferentes do disco.

Dessa forma, existem várias estratégias que podem ser usadas para otimizar os padrões de acesso à memória, incluindo:

- **Otimização do código**: os programadores podem otimizar seus programas para melhorar os padrões de acesso à memória. Por exemplo, podem usar técnicas como a alocação de memória contínua para reduzir a fragmentação e melhorar a localidade espacial.

- **Algoritmos de alocação de memória**: o sistema operacional pode usar diferentes algoritmos de alocação de memória para otimizar o uso da memória e reduzir o número de faltas de página. Algoritmos como FIFO e Least Recently Used (LRU) podem ajudar a gerenciar a memória de forma mais eficiente, evitando que as páginas mais usadas sejam substituídas prematuramente.

- **Gerenciamento de cache**: o sistema operacional pode usar caches para armazenar as páginas mais usadas na memória, evitando que sejam carregadas do disco a cada acesso. Isso pode reduzir significativamente o número de faltas de página.

Imagine um sistema de gerenciamento de banco de dados que processa consultas complexas que requerem acesso a grandes quantidades de dados. Se o sistema for projetado de forma que as consultas acessem os dados de forma aleatória, sem exibir localidade espacial ou temporal, a paginação por demanda pode resultar em um grande número de faltas de página. Isso pode levar a um desempenho lento, impactando a resposta às consultas e a capacidade do sistema de lidar com o volume de transações. No entanto, se o sistema for otimizado para melhorar a localidade espacial, por exemplo, organizando os dados de forma que consultas relacionadas acessem dados adjacentes, o desempenho pode ser significativamente melhorado. As faltas de página serão reduzidas, levando a tempos de resposta mais rápidos e um sistema de gerenciamento de banco de dados mais eficiente.

A paginação por demanda é uma técnica importante para o gerenciamento de memória, mas seu desempenho depende fortemente dos padrões de acesso à memória. A alta localidade espacial e temporal pode levar a um desempenho superior, enquanto a baixa localidade pode resultar em um número excessivo de faltas de página e desempenho lento. Para garantir um bom desempenho com a paginação por demanda, é fundamental otimizar os padrões de acesso à memória usando técnicas como otimização de código, algoritmos de alocação de memória e gerenciamento de cache. O entendimento e a aplicação dessas estratégias são cruciais para a eficiência e o desempenho de sistemas complexos que utilizam paginação por demanda.

**Lembrete**

O swapping ainda se destaca como uma ferramenta versátil e poderosa, utilizada por investidores e instituições financeiras para gerenciar riscos, aproveitar oportunidades de investimento e ajustar seus portfólios de acordo com suas necessidades específicas.

### Permuta-padrão

A permuta-padrão envolve a transferência de processos entre a memória principal e uma memória de retaguarda, que é comumente um disco veloz. Ela deve ser suficientemente grande para acomodar cópias de todas as imagens da memória para todos os usuários, e deve fornecer acesso direto a elas. O sistema mantém uma fila de prontos composta por todos os processos cujas imagens da memória estão na memória de retaguarda ou na memória principal e que estão prontos para serem executados.

Sempre que o scheduler da CPU decide executar um processo, ele chama o despachante. O despachante verifica se o próximo processo na fila está em memória. Caso não esteja, e se não houver uma região de memória livre, o despachante remove um processo correntemente em memória e o permuta com o processo desejado. Em seguida, ele recarrega os registradores e transfere o controle ao processo selecionado.

O tempo de mudança de contexto nesse sistema de permuta é bem alto. Para termos uma ideia, imagine que o processo do usuário tenha um tamanho de 100 MB e a memória de retaguarda seja um disco rígido padrão com taxa de transferência de 50 MB por segundo. A transferência real do processo de 100 MB em uma das transferências (para dentro ou para fora) da memória principal leva:

100 MB / 50 MB por segundo = 2 segundos.

O tempo de 2.000 milissegundos é o tempo de permuta. Já que precisamos executar operações de remoção e inserção do processo, o tempo total da permuta será de aproximadamente 4.000 milissegundos.

Em sua maior parte, o tempo de permuta é o tempo de transferência. O tempo de transferência total é diretamente proporcional ao montante de memória permutada. Se tivermos um sistema de computação com 4 GB de memória principal e um sistema operacional residente ocupando 1 GB, o tamanho máximo para o processo do usuário será de 3 GB. No entanto, muitos processos de usuário podem ser bem menores do que isso, em torno de 100 MB. Um processo de 100 MB poderia ser removido em 2 segundos, comparados aos 60 segundos requeridos para a permuta de 3 GB. É claro que seria útil saber exatamente quanto de memória um processo de usuário **está** usando e não simplesmente quanto ele **pode estar** usando. Assim, teríamos que permutar somente o que estiver sendo realmente usado, reduzindo o tempo de permuta. Para esse método ser eficaz, o usuário deve manter o sistema informado sobre qualquer alteração nos requisitos de memória. Portanto, um processo com requisitos de memória dinâmica terá de emitir chamadas de sistema — request_memory() e release_memory() — para informar ao sistema operacional as mudanças em suas necessidades de memória.

A permuta também é restringida por outros fatores. Se quisermos permutar um processo, devemos estar certos de que ele está totalmente ocioso. Qualquer I/O pendente é particularmente importante. Um processo pode estar esperando por uma operação de I/O quando quisermos removê-lo para liberar memória. No entanto, se a operação de I/O estiver acessando assincronamente a memória do usuário em busca de buffers de I/O, então o processo não poderá ser removido.

Suponha que a operação de I/O esteja enfileirada porque o dispositivo está ocupado. Se removermos o processo P1 da memória e inserirmos o processo P2, a operação de I/O poderia tentar usar a memória que agora pertence ao processo P2. As duas principais soluções para esse problema são: nunca permutar um processo com I/O pendente, ou executar operações de I/O somente em buffers do sistema operacional. Assim, as transferências entre buffers do sistema operacional e a memória do processo ocorrerão apenas quando o processo for inserido na memória. Observe que esse armazenamento duplo em buffer adiciona overhead por si só. Agora, temos de copiar os dados novamente da memória do kernel para a memória do usuário, antes que o processo do usuário possa acessá-los.

A permuta-padrão não é usada nos sistemas operacionais modernos. Ela requer muito tempo de permuta e fornece muito pouco tempo de execução para ser uma solução razoável para o gerenciamento da memória. Versões modificadas de permuta, no entanto, são encontradas em muitos sistemas, inclusive no Unix, no Linux e no Windows. Em uma variação comum, a permuta normalmente é desabilitada, sendo iniciada se o montante de memória livre (memória não utilizada disponível para o sistema operacional ou os processos usarem) cai abaixo de um valor limite. A permuta é interrompida quando

o montante de memória livre aumenta. Outra variação envolve a permuta de partes de processos, em vez de processos inteiros, para diminuir o tempo de permuta. Geralmente, essas formas modificadas de permuta funcionam em conjunto com a memória virtual.

### Permuta em sistemas móveis

Embora a maioria dos sistemas operacionais para PCs e servidores dê suporte a alguma versão modificada de permuta, os sistemas móveis normalmente não suportam forma alguma de permuta. Os dispositivos móveis costumam usar memória flash, em vez dos discos rígidos mais volumosos, como seu espaço de armazenamento persistente. A restrição de espaço resultante é uma razão para os projetistas de sistemas operacionais móveis evitarem a permuta. Outras razões incluem o número limitado de gravações que a memória flash pode tolerar antes de se tornar não confiável e o fraco throughput entre a memória principal e a memória flash nesses dispositivos. Em vez de usar a permuta, quando a memória livre cai abaixo de determinado limite, o iOS da Apple, por exemplo, **solicita** às aplicações que abandonem voluntariamente a memória alocada. Dados somente de leitura, como código, são removidos do sistema e recarregados posteriormente a partir da memória flash, se necessário. Dados que foram modificados, como a pilha, nunca são removidos. No entanto, qualquer aplicação que não consiga liberar memória suficiente pode ser encerrada pelo sistema operacional.

O Android não dá suporte à permuta e adota uma estratégia semelhante à usada pelo iOS. Ele pode encerrar um processo se não houver memória livre suficiente disponível. No entanto, antes de encerrar um processo, o Android grava seu estado da aplicação na memória flash para que ela possa ser rapidamente reiniciada.

Em razão dessas restrições, os desenvolvedores de sistemas móveis devem alocar e liberar memória cuidadosamente para assegurar que suas aplicações não usem memória demais ou sofram de vazamentos de memória. Observe que tanto o iOS quanto o Android dão suporte à paginação, assim, eles têm recursos de gerenciamento da memória.

### Alocação de memória contígua

A memória principal deve acomodar tanto o sistema operacional quanto os diversos processos de usuário. Portanto, precisamos alocar a memória principal da maneira mais eficiente possível. Esta seção explica um método antigo, a alocação de memória contígua.

A memória é usualmente dividida em duas partições: uma para o sistema operacional residente e outra para os processos de usuário. Podemos alocar o sistema operacional tanto na memória baixa quanto na memória alta. O principal fator que afeta essa decisão é a localização do vetor de interrupções, uma vez que ele costuma estar na memória baixa, os programadores também costumam alocar o sistema operacional na memória baixa. Portanto, neste livro-texto, discutimos somente a situação em que o sistema operacional reside na memória baixa; o desenvolvimento da outra opção é semelhante.

É comum esperarmos que vários processos de usuário residam na memória ao mesmo tempo. Logo, precisamos considerar como alocar memória disponível aos processos que estão na fila de entrada esperando para serem trazidos para a memória. Na alocação de memória contígua, cada processo fica contido em uma única seção da memória que é contígua à seção que contém o próximo processo.

### Proteção da memória

O registrador de relocação contém o valor do menor endereço físico; o registrador limite contém o intervalo de endereços lógicos (por exemplo, relocação = 100040 e limite = 74600). Cada endereço lógico deve pertencer ao intervalo especificado pelo registrador limite. A MMU mapeia o endereço lógico dinamicamente adicionando o valor ao registrador de relocação, assim, esse endereço mapeado é enviado à memória (figura 95).

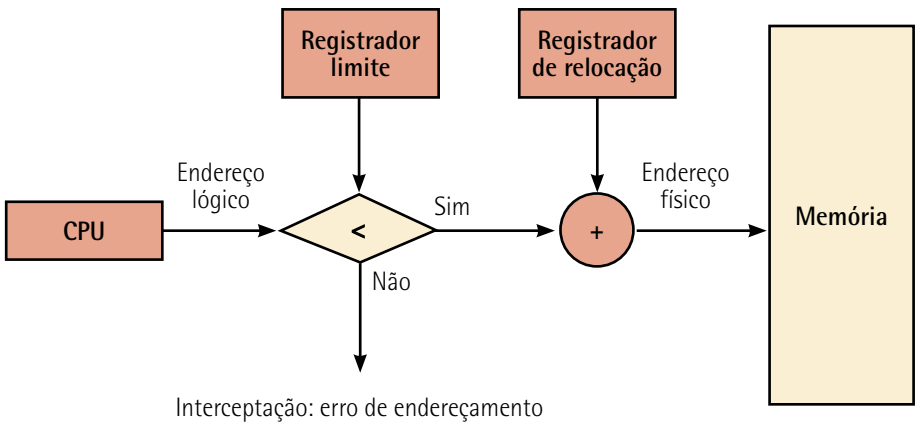



Figura 95 – Suporte de hardware para registradores de relocação e registradores limite

Fonte: Silberschartz (2015, p. 296).

Quando o scheduler da CPU seleciona um processo para execução, o despachante carrega os registradores de relocação e limite com os valores corretos como parte da mudança de contexto. Já que todo endereço gerado por uma CPU é verificado em relação a esses registradores, podemos proteger tanto o sistema operacional quanto os programas e dados de outros usuários para que não sejam modificados pelo processo em execução.

O esquema do registrador de relocação fornece um modo eficaz para permitir que o tamanho do sistema operacional mude dinamicamente. Essa flexibilidade é desejável em muitas situações, por exemplo, quando o sistema operacional contém código e espaço em buffer de drivers de dispositivos. Se um driver de dispositivos (ou outro serviço do sistema operacional) não for comumente usado, não é aconselhável manter o código e os dados na memória, já que podemos usar esse espaço para outras finalidades. Esse código é, às vezes, chamado de código transiente do sistema operacional: ele vem e vai conforme necessário. Portanto, seu uso altera o tamanho do sistema operacional durante a execução de programas.

### Alocação de memória

Um dos métodos mais simples para alocação da memória é dividir a memória em várias partições de tamanho fixo. Cada partição pode conter exatamente um processo, devido ao fato do grau de multiprogramação ser limitado pelo número de partições. Nesse método de partições múltiplas, quando uma delas está livre, um processo é selecionado da fila de entrada e carregado na partição disponível. Quando o processo termina, a partição torna-se disponível para outro processo. Esse método, denominado de MFT, foi originalmente usado pelo sistema operacional IBM OS/360, mas não está mais em uso. O método descrito a seguir, chamado MVT, é uma generalização do esquema de partições fixas, sendo usado principalmente em ambientes batch. Muitas das ideias apresentadas aqui também são aplicáveis a um ambiente de tempo compartilhado que usa a segmentação pura no gerenciamento da memória.

No esquema de partições variáveis, o sistema operacional mantém uma tabela indicando quais partes da memória estão disponíveis e quais estão ocupadas. Inicialmente, toda a memória está disponível para processos de usuário e é considerada um grande bloco de memória disponível, uma brecha. Como você verá, eventualmente a memória contém um conjunto de brechas de vários tamanhos.

Conforme os processos entram no sistema, eles são colocados em uma fila de entrada. O sistema operacional leva em consideração os requisitos de memória de cada um e o montante de espaço disponível nela para determinar que processos devem ter memória alocada. Quando um processo recebe espaço, ele é carregado na memória e pode, então, competir por tempo da CPU. Quando um processo termina, sua memória é liberada para que, então, o sistema operacional possa preencher com outro processo da fila de entrada.

A qualquer tempo, portanto, temos uma lista de tamanhos de blocos disponíveis e uma fila de entrada. O sistema operacional pode ordenar a fila de entrada de acordo com um algoritmo de scheduling. A memória é alocada aos processos até que, finalmente, os requisitos de memória do próximo processo não possam ser atendidos, isto é, nenhum bloco de memória (ou brecha) disponível é suficientemente grande para contê-lo. O sistema operacional pode então esperar até que um bloco suficientemente grande esteja disponível, ou pode percorrer a fila de entrada para ver se os requisitos menores de memória de algum outro processo possam ser atendidos.

Em geral, como mencionado, os blocos de memória disponíveis compõem um conjunto de brechas de vários tamanhos espalhadas pela memória. Quando um processo chega e precisa de memória, o sistema procura no conjunto por uma brecha que seja suficientemente grande para esse processo. Se a brecha for grande demais, ela será dividida em duas partes. Uma parte é alocada ao processo que chegou; a outra é devolvida ao conjunto de brechas. Quando um processo é encerrado, ele libera seu bloco de memória que é, então, colocado novamente no conjunto de brechas. Se a nova brecha for adjacente a outras, elas serão mescladas para formar uma brecha maior. Nesse momento, o sistema pode precisar verificar se existem processos esperando por memória e se essa memória recém-liberada e recombinada poderia atender as demandas de algum desses processos em espera.

Esse procedimento é uma instância específica do problema de alocação de memória dinâmica geral que diz respeito a como satisfazer uma solicitação de tamanho *n* a partir de uma lista de brechas livres. Há muitas soluções para esse problema. As estratégias do primeiro-apto (first-fit), do mais-apto (best-fit) e do menos-apto (worst-fit) são as mais usadas para selecionar uma brecha livre no conjunto de brechas disponíveis.

- **Primeiro-apto**: aloca a primeira brecha que seja suficientemente grande. A busca pode começar tanto no início do conjunto de brechas quanto na locação na qual a busca anterior pelo primeiro-apto terminou. Podemos encerrar a busca, assim que encontrarmos uma brecha livre suficientemente grande.

- **Mais-apto**: aloca a menor brecha que seja suficientemente grande. Devemos pesquisar a lista inteira, a menos que ela seja ordenada por tamanho. Essa estratégia produz a brecha com menos espaço sobrando.

- **Menos-apto**: aloca a maior brecha. Novamente, devemos pesquisar a lista inteira, a menos que ela seja classificada por tamanho. Essa estratégia produz a brecha com mais espaço sobrando, que pode ser mais útil do que a brecha com menos espaço sobrando da abordagem do mais-apto.

Simulações têm mostrado que tanto o primeiro-apto quanto o mais-apto são melhores do que o menos-apto em termos de redução de tempo e utilização de memória. Nem o primeiro-apto, nem o mais-apto é claramente melhor do que o outro em termos de utilização de memória, mas o primeiro-apto geralmente é mais rápido.

**Exemplos de arquiteturas Solaris, Oracle Sparc, Intel 32 e 64 bits**

Nessa etapa de nosso estudo, é importante citarmos sistemas operacionais e CPUs de 64 bits modernos e totalmente integrados que fornecem memória virtual de baixo overhead. O Solaris, sendo executado na CPU Sparc, é um sistema operacional totalmente de 64 bits e, como tal, tem de resolver o problema da memória virtual sem esgotar toda a sua memória física, mantendo múltiplos níveis de tabelas de páginas. Sua abordagem é um pouco complexa, mas resolve o problema eficientemente usando tabelas de páginas com hash.

Há duas tabelas com hash, sendo uma para o kernel e outra para todos os processos de usuário. As duas mapeiam endereços de memória da memória virtual para a memória física. Cada entrada da tabela com hash representa uma área contígua de memória virtual mapeada, o que é mais eficiente do que ter uma entrada da tabela com hash separada para cada página. Cada entrada tem um endereço base e um intervalo indicando o número de páginas que a entrada representa.

A tradução virtual-para-físico demoraria muito, se cada endereço demandasse uma busca na tabela com hash. Assim, a CPU implementa um TLB que contém entradas de uma tabela de tradução, Translation Table Entries (TTEs), para pesquisas de hardware rápidas. Um cache com essas TTEs reside em um buffer de armazenamento de tradução, Translation Storage Buffer (TSB), que inclui uma entrada por página acessada recentemente. Quando ocorre uma referência ao endereço virtual, o hardware pesquisa

o TLB em busca de uma tradução. Se nenhuma é encontrada, o hardware percorre o TSB na memória procurando pela TTE que corresponda ao endereço virtual que causou a pesquisa.

Essa funcionalidade de percurso do TLB é encontrada em muitas CPUs modernas. Se uma ocorrência for encontrada no TSB, a CPU copiará a entrada do TSB no TLB, e a tradução da memória se completará. Se não for encontrada nenhuma ocorrência no TSB, o kernel será interrompido para pesquisar na tabela com hash. O kernel cria então uma TTE, a partir da tabela com hash apropriada, e a armazena no TSB para carga automática no TLB pela unidade de gerenciamento de memória da CPU. Para concluir, o manipulador de interrupções devolve o controle para a MMU, que completa a tradução do endereço e recupera o byte ou a palavra solicitada, na memória principal.

### Arquiteturas Intel de 32 e 64 Bits

A arquitetura de chips Intel tem dominado o panorama dos computadores pessoais há vários anos. O Intel 8086 de 16 bits apareceu no fim dos anos 1970 e foi logo seguido por outro chip de 16 bits, o Intel 8088, que ficou conhecido por ser o chip usado no PC IBM original. Tanto o chip 8086 quanto o chip 8088 foram baseados em uma arquitetura segmentada. Depois, a Intel produziu uma série de chips de 32 bits, o IA-32, que incluía a família de processadores Pentium de 32 bits. A arquitetura IA-32 dava suporte tanto à paginação quanto à segmentação. Mais recentemente, a Intel produziu uma série de chips de 64 bits baseados na arquitetura x86-64. No momento, todos os sistemas operacionais mais populares para PCs são executados em chips Intel, inclusive o Windows, o Mac OS X e o Linux (embora o Linux, é claro, também seja executado em várias outras arquiteturas). Surpreendentemente, no entanto, a predominância da Intel não se estendeu aos sistemas móveis, nos quais a arquitetura ARM desfruta de considerável sucesso atualmente.

### Arquitetura IA-32

O gerenciamento da memória em sistemas IA-32 é dividido em dois componentes, segmentação e paginação, funcionando da seguinte forma: a CPU gera endereços lógicos que são fornecidos à unidade de segmentação; a unidade de segmentação produz um endereço linear para cada endereço lógico; o endereço linear é, então, fornecido à unidade de paginação que, por sua vez, gera o endereço físico na memória principal. Dessa forma, as unidades de segmentação e paginação formam o equivalente à unidade de gerenciamento da memória (MMU), como mostrado na figura 96.

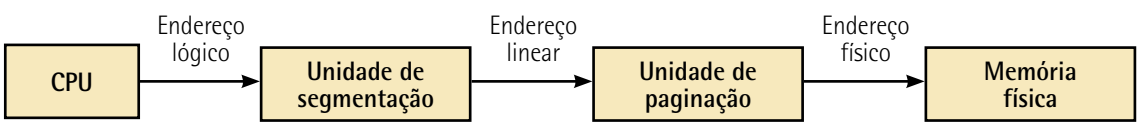



Figura 96 – Tradução de endereço lógico para físico no IA-32

Fonte: Silberschartz (2015, p. 315).

### x86-64

A Intel tem uma história interessante de desenvolvimento de arquiteturas de 64 bits. Sua estreia foi com a arquitetura IA-64, posteriormente chamada Itanium, mas essa arquitetura não foi amplamente adotada. Enquanto isso, outro fabricante de chips, a AMD, começou a desenvolver uma arquitetura de 64 bits conhecida como x86-64 que se baseava na extensão do conjunto de instruções existente no IA-32. O x86-64 suportava espaços de endereçamento lógico e físico muito maiores, assim como ocorria em outras ferramentas advindas dos avanços feitos na arquitetura. Historicamente, a AMD tem desenvolvido com frequência chips baseados na arquitetura da Intel, mas, agora, os papéis se inverteram, já que a Intel adotou a arquitetura x86-64 da AMD. Ao discutirmos sobre essa arquitetura, em vez de usar os nomes comerciais AMD64 e Intel 64, usaremos o termo mais geral: x86-64.

Normalmente, o suporte a um espaço de endereçamento de 64 bits gera surpreendentes 264 bytes de memória endereçável, um número maior do que 16 quintilhões (ou 16 exabytes). No entanto, ainda que sistemas de 64 bits possam endereçar tanta memória, na prática, bem menos do que 64 bits são usados para a representação de endereços nos projetos atuais. Enquanto isso, atualmente, a arquitetura x86-64 fornece um endereço virtual de 48 bits com suporte a tamanhos de página de 4 kB, 2 MB ou 1 GB, usando quatro níveis de hierarquia de paginação. A representação do endereço linear aparece na figura 97. Esse esquema de endereçamento pode usar a PAE, onde os endereços virtuais têm 48 bits, mas podem suportar endereços físicos de 52 bits (4096 terabytes).

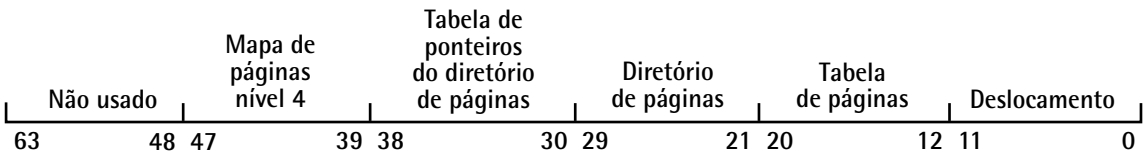



Figura 97 – Endereço linear no x86-64

Fonte: Silberschartz (2015, p. 318).

### Computação de 64 bits

A história nos ensinou que, mesmo que capacidades de memória, velocidades de CPU e recursos de computação semelhantes pareçam suficientemente grandes para satisfazer a demanda no futuro próximo, o crescimento da tecnologia acaba absorvendo os recursos disponíveis, e nos encontramos diante da necessidade de memória ou poder de processamento adicional, frequentemente, antes do que pensávamos. O que será do futuro da tecnologia, onde um espaço de endereçamento de 64 bits será muito pequeno?

Um exemplo sobre o debate do futuro da tecnologia é a arquitetura ARM. Embora os chips da Intel tenham dominado o mercado de computadores pessoais por mais de 30 anos, os chips de dispositivos móveis, como smartphones e computadores tablets, são executados com frequência em processadores ARM de 32 bits. O interessante é que, enquanto a Intel projeta e fabrica os chips, a ARM somente os projeta. Em seguida, ela licencia seus projetos para fabricantes de chips. A Apple obteve licença da ARM para usar seu projeto nos dispositivos móveis iPhone e iPad, e vários smartphones baseados no Android também usam processadores ARM.

A arquitetura ARM de 32 bits suporta os seguintes tamanhos de página: páginas de 4 kB e 16 kB; e páginas de 1 MB e 16 MB (denominadas seções).

O sistema de paginação em uso depende de uma página ou uma seção estar sendo referenciada. A paginação de um nível é usada para seções de 1 MB e 16 MB; a paginação de dois níveis é usada para páginas de 4 kB e 16 kB. A tradução de endereços com a MMU ARM é mostrada na figura 98.

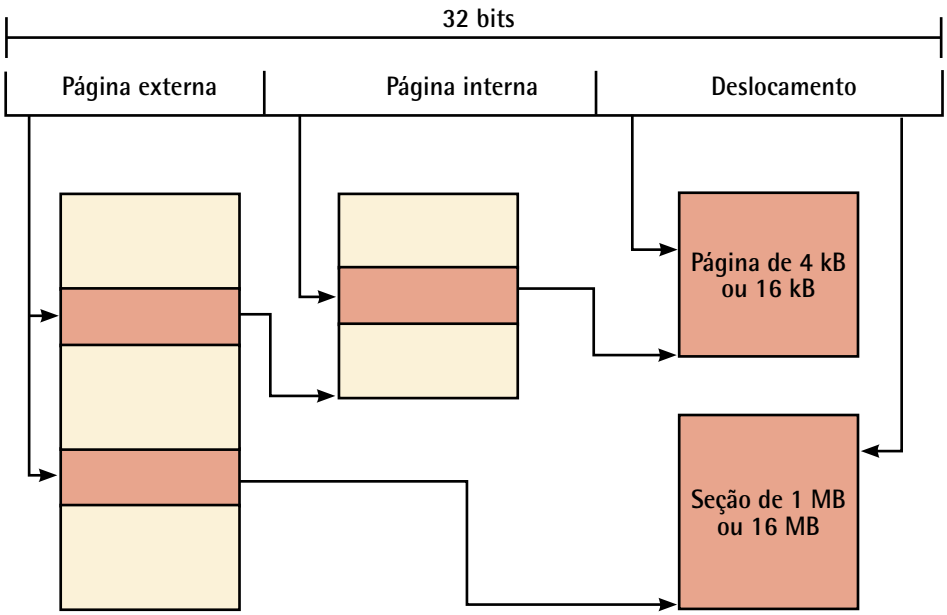



Figura 98 – Tradução de endereço lógico no ARM

Fonte: Silberschartz (2015, p. 319).

A arquitetura ARM também suporta dois níveis de TLBs. No nível mais externo, estão dois micros TLBs: um TLB separado para dados e outro para instruções, onde o micro TLB também dá suporte a ASIDs e, também, onde a tradução de endereços começa. No nível interno, fica um único TLB principal, sendo que, em caso de erro, o TLB principal é verificado. Se os dois TLBs gerarem erros, uma pesquisa de tabela de páginas deve ser executada em hardware.

**Saiba mais**

Para saber mais sobre segmentação no IA-32, leia o capítulo 8 do livro a seguir:

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Fundamentos de sistemas operacionais*. 9. ed. Rio de Janeiro: LTC, 2015.

### Fragmentação

Tanto a estratégia do primeiro-apto quanto a do mais-apto para alocação de memória sofrem de fragmentação externa. À medida que processos são carregados na memória e dela são removidos, o espaço de memória livre é fragmentado em pequenos pedaços. A fragmentação externa ocorre quando há espaço total na memória suficiente para atender a uma solicitação, mas os espaços disponíveis não são contíguos: a memória está fragmentada em um grande número de pequenas brechas. Esse problema de fragmentação pode ser grave, onde, na pior das hipóteses, poderíamos ter um bloco de memória livre (ou desperdiçada) entre cada par de processos. Se, em vez disso, todos esses pequenos fragmentos de memória estivessem em um grande bloco livre, poderíamos ser capazes de executar muito mais processos.

A escolha entre usar a estratégia do primeiro-apto ou a estratégia do mais-apto pode afetar o nível de fragmentação. Dessa forma, o primeiro-apto é melhor para alguns sistemas, enquanto o mais-apto é melhor para outros. Outro fator importante é qual extremidade de um bloco livre está alocada (que parte está sobrando – a inferior ou a superior?). Independentemente do algoritmo usado, a fragmentação externa é um problema.

Dependendo do montante total de espaço na memória e do tamanho médio do processo, a fragmentação externa pode ser um problema maior ou menor. Análises estatísticas do primeiro-apto, por exemplo, revelam que, mesmo com alguma otimização, dados *N* blocos alocados, outros 0,5 *N* blocos serão perdidos por causa da fragmentação. Isto é, um terço da memória pode ficar inutilizável. Essa propriedade é conhecida como a regra dos 50%.

A fragmentação da memória pode ser interna ou externa. Considere um esquema de alocação de partições múltiplas com uma brecha de 18.464 bytes. Suponha que o próximo processo solicite 18.462 bytes. Se alocarmos exatamente o bloco solicitado, ficaremos com uma brecha de 2 bytes. O overhead para administrar o uso dessa brecha será substancialmente maior do que a própria brecha. A abordagem geral para evitar esse problema é particionar a memória física em blocos de tamanho fixo e alocar a memória em unidades com base no tamanho do bloco. Nessa abordagem, a memória alocada a um processo pode ser um pouco maior do que a memória solicitada. A diferença entre esses dois números é a fragmentação interna, a memória não utilizada que pertence a uma partição.

Uma solução para o problema da fragmentação externa é a compactação. O objetivo é mesclar os conteúdos da memória para unir toda a memória disponível em um grande bloco. No entanto, a compactação nem sempre é possível. Se a relocação for estática e feita em tempo de montagem ou de carga, a compactação não poderá ser realizada. Ela é possível apenas se a relocação for dinâmica e feita em tempo de execução. Se os endereços são relocados dinamicamente, a relocação requer somente a transferência do programa e dos dados e, depois, a alteração do registrador base para refletir o novo endereço base. Quando a compactação é possível, devemos determinar seu custo. O algoritmo de compactação mais simples move todos os processos para uma extremidade da memória; todas as brechas são movimentadas para a outra extremidade, produzindo uma grande brecha de memória disponível, fazendo com que esse esquema seja dispendioso.

Outra solução possível para o problema da fragmentação externa é permitir que o espaço de endereçamento lógico dos processos seja não contíguo, possibilitando assim que um processo receba memória física onde quer que essa memória esteja disponível. Duas técnicas complementares proporcionam essa solução: a segmentação e a paginação. Essas técnicas também podem ser combinadas. A fragmentação é um problema geral da computação que pode ocorrer em qualquer local em que tenhamos que gerenciar blocos de dados.

**Segmentação**

A visão que o usuário tem da memória não corresponde à memória física real. Isso é igualmente verdadeiro para a visão que o programador tem dela. Na verdade, lidar com a memória em termos de suas propriedades físicas é inconveniente tanto para o sistema operacional quanto para o programador. Mas, e se o hardware pudesse fornecer um mecanismo de memória que mapeasse a visão do programador para a memória física real? O sistema teria mais liberdade para gerenciar a memória, enquanto o programador teria um ambiente de programação mais natural. É nesse ponto em que temos a segmentação fornecendo esse mecanismo.

Será que os programadores consideram a memória como um array linear de bytes, alguns contendo instruções e outros contendo dados? A maioria dos programadores diria que não. Em vez disso, eles preferem ver a memória como um conjunto de segmentos de tamanho variável, sem que haja necessariamente uma ordem entre eles.

Ao escrever um programa, um programador o vê como um programa principal com um conjunto de métodos, procedimentos ou funções. Ele também pode incluir várias estruturas de dados: objetos, arrays, pilhas, variáveis e assim por diante. Cada um desses módulos ou elementos de dados é referenciado pelo nome. O programador fala sobre "a pilha", "a biblioteca de matemática" e "o programa principal" sem se preocupar com os endereços que esses elementos ocupam na memória. Por exemplo, não é relevante para ele se a pilha está armazenada antes ou depois da função sqrt(). Os segmentos variam em tamanho, e o tamanho de cada um é definido intrinsecamente por sua finalidade no programa. Os elementos dentro de um segmento são identificados por seu deslocamento a partir do início do segmento: o primeiro comando do programa, a sétima entrada do quadro de pilha na pilha, a quinta instrução de sqrt() e assim por diante.

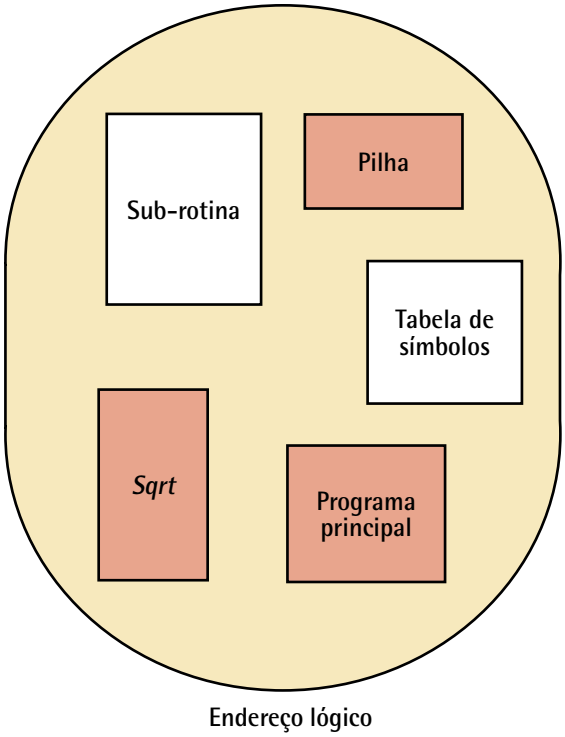



Figura 99 – Visão que um programador tem de um programa

Fonte: Silberschartz (2015, p. 300).

Dessa forma, a segmentação é um esquema de gerenciamento de memória que dá suporte à visão da memória desse programador. Um espaço de endereçamento lógico é um conjunto de segmentos, onde cada um tem um nome e um tamanho. Os endereços especificam tanto o nome do segmento quanto o deslocamento dentro dele. O programador, então, especifica cada endereço com dois valores: um nome de segmento e um deslocamento.

Para simplificar a implementação, os segmentos são numerados e referenciados por um número, não por um nome. Portanto, um endereço lógico é composto por uma dupla de dois elementos:

<número do segmento, deslocamento>

Normalmente, quando um programa é compilado, o compilador constrói automaticamente segmentos que refletem o programa de entrada. Um compilador C pode criar segmentos separados para os elementos, como: código; variáveis globais; heap a partir do qual a memória é alocada; pilhas usadas para cada thread; e biblioteca C padrão.

As bibliotecas, que são vinculadas em tempo de compilação, podem ter segmentos atribuídos separadamente. O carregador toma todos esses segmentos e designa a eles números de segmentos.

## 7.3 Memória virtual

A memória virtual é uma técnica que permite a execução de processos que não estão totalmente nela. Uma grande vantagem desse esquema é que os programas podem ser maiores do que a memória física. Além disso, a memória virtual abstrai a memória principal em um array de armazenamento uniforme extremamente grande, separando a memória lógica, conforme vista pelo usuário, da memória física. Essa técnica deixa os programadores livres de preocupações com as limitações de armazenamento da memória. Ainda, a memória virtual também permite que os processos compartilhem arquivos facilmente e implementem a memória compartilhada. Além disso, ela fornece um mecanismo eficiente para a criação de processos.

Os algoritmos de gerenciamento da memória são necessários, já que o requisito básico deles define que as instruções que estão sendo executadas devem estar na memória física. A primeira abordagem para a execução desse requisito é colocar o espaço de endereçamento lógico inteiro na memória física. A carga dinâmica pode ajudar a atenuar essa restrição, mas, geralmente, ela requer precauções especiais e trabalho adicional do programador.

O requisito de que as instruções devem estar na memória física para serem executadas parece necessário e racional, mas também é inadequado, já que limita o tamanho de um programa ao tamanho da memória física. Na verdade, um exame dos programas reais mostra que, em muitos casos, o programa inteiro não é necessário. Considere os exemplos a seguir:

- Os programas, com frequência, têm um código que manipula condições de erro não usuais. Já que raramente esses erros ocorrem na prática, quando ocorrem, esse código quase nunca é executado.

- Arrays, listas e tabelas recebem em geral mais memória do que realmente precisam. Um array pode ser declarado como tendo 100 por 100 elementos, ainda que raramente tenha mais de 10 por 10 elementos. Uma tabela de símbolos de um montador pode ter espaço para 3.000 símbolos, embora o programa médio tenha menos de 200 símbolos.

- Certas opções e recursos de um programa podem ser usados raramente. Por exemplo, as rotinas dos computadores do governo dos Estados Unidos que equilibram o orçamento não são usadas há muitos anos.

Mesmo nos casos em que o programa inteiro é necessário, ele pode não ser necessário em sua totalidade ao mesmo tempo. A possibilidade de executar um programa que esteja apenas parcialmente na memória traria muitos benefícios, já que o programa não ficaria mais restrito ao montante de memória física disponível e, dessa forma, os usuários poderiam escrever programas para um espaço de endereçamento virtual extremamente grande, simplificando a tarefa de programação. Ainda, cada programa de usuário poderia usar menos memória física, mais programas poderiam ser executados ao mesmo tempo, com um aumento correspondente na utilização da CPU e no throughput, mas sem aumento no tempo de resposta ou no tempo de turnaround. Além do fato de que menos operações de I/O seriam necessárias para carregar ou permutar programas de usuário na memória; portanto, cada programa de usuário seria executado mais rapidamente. Assim, a execução de um programa que não esteja inteiramente na memória beneficiaria tanto o sistema quanto o usuário.

A memória virtual envolve a separação entre a memória lógica como percebida pelos usuários e a memória física. Essa separação permite que uma memória virtual extremamente grande seja fornecida aos programadores quando apenas uma memória física menor está disponível (figura 100). A memória virtual torna a tarefa de programar muito mais fácil porque ele não precisa mais se preocupar com o montante de memória física disponível. Em vez disso, ele pode se concentrar no problema a ser programado.

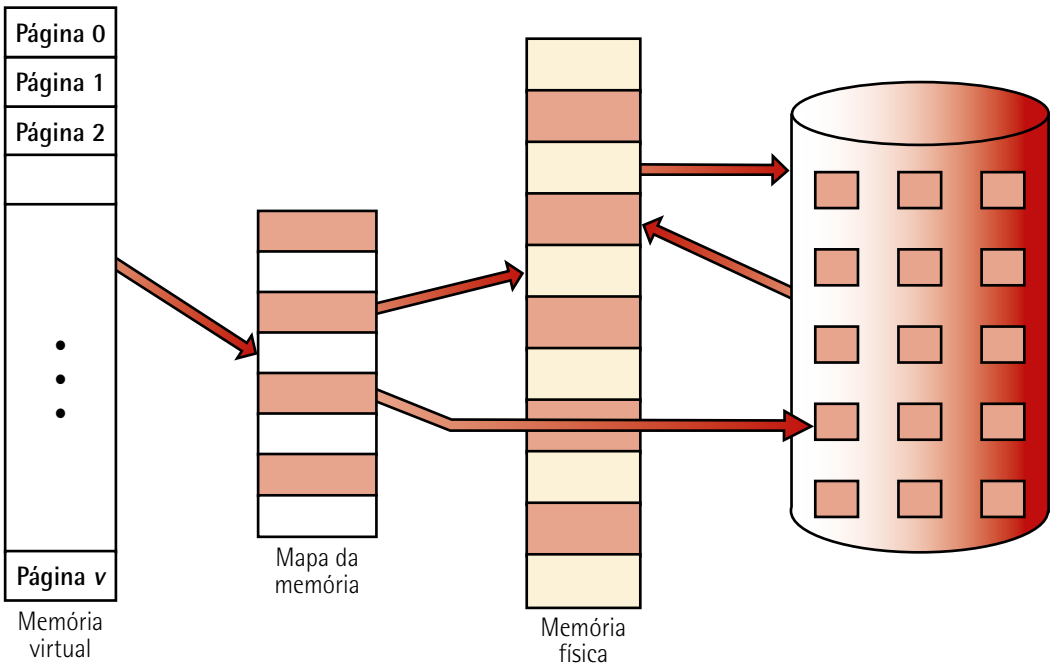



Figura 100 – Diagrama mostrando memória virtual que é maior do que a memória física

Fonte: Silberschartz (2015, p. 326).

O espaço de endereçamento virtual de um processo diz respeito à visão lógica (ou virtual) de como um processo é armazenado na memória. Normalmente, de acordo com essa visão, um processo começa em um determinado endereço lógico — digamos, endereço 0 — e que existe em memória contígua. No entanto, a memória física pode ser organizada em quadros de páginas e os quadros de páginas físicos atribuídos a um processo podem não ser contíguos. Sendo assim, é responsabilidade da unidade de gerenciamento da memória (MMU) mapear páginas lógicas para quadros de páginas físicas na memória.

O heap crescer para cima na memória, à medida que ele é usado na alocação de memória dinâmica. Da mesma forma, permitimos que a pilha cresça para baixo na memória por meio de chamadas de função sucessivas. O grande espaço vazio (ou brecha) entre o heap e a pilha faz parte do espaço de endereçamento virtual, mas precisará de páginas físicas reais somente se o heap ou a pilha crescerem. Os espaços de endereçamento virtuais que incluem brechas são conhecidos como espaços de endereçamento esparsos.

O uso desses espaços é benéfico, porque as brechas podem ser preenchidas conforme os segmentos da pilha ou do heap que crescem, ou se quisermos vincular bibliotecas dinamicamente (ou possivelmente outros objetos compartilhados) durante a execução do programa, conforme observamos na figura 101.

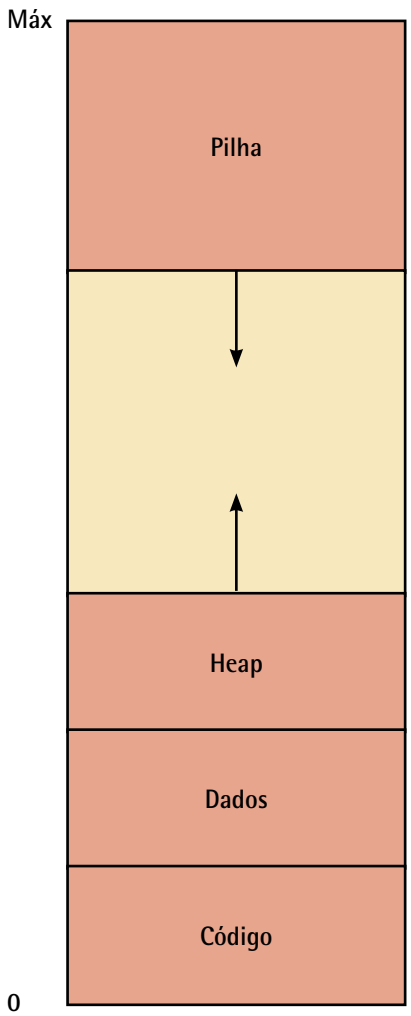



Figura 101 – Espaço de endereçamento virtual

Fonte: Silberschartz (2015, p. 327).

Além de separar a memória lógica da física, a memória virtual permite que arquivos e memória sejam partilhados por dois ou mais processos através do compartilhamento de páginas. Dessa forma, as bibliotecas do sistema podem se dividir por vários processos através do mapeamento do objeto compartilhado para um espaço de endereçamento virtual. Embora cada processo considere as bibliotecas como parte do seu espaço de endereçamento virtual, as páginas reais em que as bibliotecas residem na memória física são compartilhadas por todos os processos. Normalmente, uma biblioteca é mapeada como somente de leitura para o espaço de cada processo que esteja vinculado a ela, o que faz com que os processos possam partilhar memória.

Lembremos de que dois ou mais processos podem se comunicar pelo uso de memória compartilhada. A memória virtual permite que um processo crie uma região de memória que ela possa dividir com outro processo. Os processos que compartilham essa região a consideram como parte de seu espaço de endereçamento virtual, mas as páginas de memória físicas reais são compartilhadas.

**Observação**

As páginas podem ser compartilhadas durante a criação de processos com a chamada de sistema fork(), acelerando a criação deles.

**Paginação por demanda**

Considere que um programa executável pode ser carregado do disco para a memória. Nesse cenário, podemos carregar-lo inteiro na memória física em tempo de execução do programa. No entanto, um problema dessa abordagem é que, inicialmente, podemos não precisar do programa inteiro na memória. Suponha, ainda, que um programa comece com uma lista de opções disponíveis na qual o usuário deve fazer uma seleção; porém, nessa situação, a carga do programa inteiro na memória resultaria na carga do código executável de todas as opções, independentemente de uma opção ter sido ou não selecionada pelo usuário. Uma estratégia alternativa seria carregar páginas somente à medida que elas sejam necessárias. Essa técnica é conhecida como paginação por demanda e é comumente usada em sistemas de memória virtual, onde as páginas são carregadas somente quando são necessárias durante a execução do programa.

Dessa forma, as páginas que nunca são acessadas, nunca são carregadas na memória física, o que faz com que um sistema de paginação por demanda seja semelhante a um sistema de paginação com permuta, em que os processos residem em memória secundária (usualmente um disco). Quando queremos executar um processo, ele é inserido na memória. No entanto, em vez de inserir o processo inteiro na memória, usamos um permutador preguiçoso.

Um permutador preguiçoso nunca insere uma página na memória, a menos que ela seja necessária. No contexto de um sistema de paginação por demanda, o uso do termo permutador é tecnicamente incorreto. Um permutador manipula processos inteiros, enquanto um paginador se preocupa com as páginas individuais de um processo. Portanto, usamos paginador, em vez de permutador, no contexto da paginação por demanda.

Quando um processo está para ser inserido na memória, o paginador avalia as páginas que serão usadas, antes que o processo seja removido novamente. Em vez de inserir um processo inteiro, o paginador traz apenas essas páginas para a memória. Portanto, ele evita que sejam transferidas para a memória páginas que não serão usadas, diminuindo o tempo de permuta e o montante de memória física necessária.

Nesse esquema, precisamos de algum tipo de suporte de hardware para diferenciar as páginas que estão na memória das páginas que estão em disco. O esquema do bit válido-inválido pode ser usado com essa finalidade. Dessa vez, no entanto, quando esse bit é posicionado como válido, a página associada é válida e está na memória. Se o bit estiver posicionado como inválido, a página não é válida (isto é, não faz parte do espaço de endereçamento lógico do processo), ou é válida, mas está correntemente em disco. A entrada na tabela de páginas para uma página que é trazida para a memória é definida como sempre, mas a entrada na tabela de páginas para uma página que não está correntemente em memória, é simplesmente marcada como inválida ou contém o endereço da página em disco (figura 102).

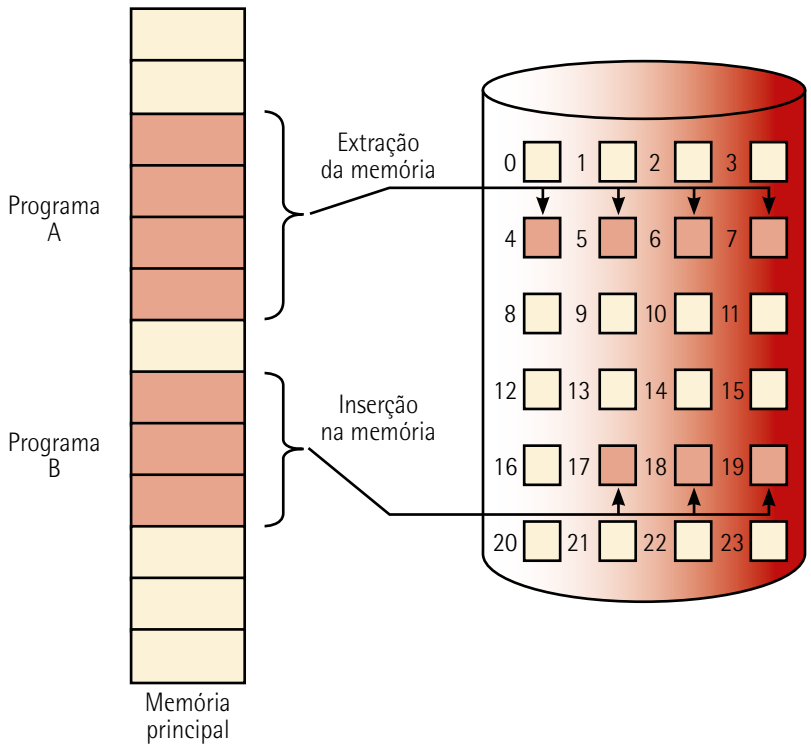



Figura 102 – Transferência de uma memória paginada para espaço de disco contíguo

Fonte: Silberschartz (2015, p. 330).

Observe que a marcação de uma página como inválida não terá efeito se o processo nunca tentar acessar essa página. Logo, se avaliarmos direito e paginarmos para a memória todas as páginas que realmente são necessárias – e somente elas –, o processo será executado exatamente como se tivéssemos trazido todas as páginas. Enquanto o processo for executado acessando páginas residentes na memória, a execução prosseguirá normalmente.

Entretanto, o que acontece se o processo tentar acessar uma página que não foi trazida para a memória? O acesso a uma página marcada como inválida causa um erro de página. Ao traduzir o endereço através da tabela de páginas, o hardware de paginação notará que o bit inválido está posicionado, causando uma interceptação para o sistema operacional. Essa interceptação é resultado da falha do sistema operacional em trazer a página desejada para a memória.

O procedimento para manipulação desse erro de página é simples:

- Verificamos uma tabela interna (usualmente mantida com o bloco de controle de processo) desse processo para determinar se a referência foi um acesso válido ou inválido à memória.
- Se a referência foi inválida, encerramos o processo. Se ela foi válida, mas ainda não trouxemos a página para a memória, a traremos agora.
- Encontramos um quadro livre (usando um da lista de quadros livres, por exemplo).
- Incluímos uma operação de disco no schedule para ler a página desejada para o quadro recém-alocado.
- Quando a leitura em disco é concluída, modificamos a tabela interna mantida com o processo e a tabela de páginas para indicar que agora a página está na memória.
- Reiniciamos a instrução que foi interrompida pela interceptação. Agora, o processo pode acessar a página como se ela sempre tivesse estado na memória.

No caso extremo, podemos iniciar a execução de um processo **sem** páginas na memória. Quando o sistema operacional posiciona o ponteiro de instruções para a primeira instrução do processo que está em uma página não residente na memória, o processo é interrompido imediatamente pelo erro de página. Após essa página ser trazida para a memória, o processo continua a ser executado, sendo interrompido, se necessário, até que cada página que ele precise esteja na memória. Nesse ponto, ele pode ser executado sem mais erros. Esse esquema é a paginação por demanda pura: nunca trazer uma página para a memória antes que ela seja necessária.

Teoricamente, alguns programas poderiam acessar várias páginas novas de memória a cada execução de instrução (uma página para a instrução e muitas para dados), possivelmente causando vários erros de página por instrução. Essa situação resultaria em um desempenho inaceitável do sistema. Felizmente, a análise de processos em execução mostra que esse comportamento é bastante improvável. Os programas tendem a ter uma localidade de referência, o que resulta em um desempenho razoável da paginação por demanda. O hardware que suporta a paginação por demanda é o mesmo da paginação e da permuta, tais quais:

- **Tabela de páginas**: pode marcar uma entrada como inválida por meio de um bit válido-inválido ou de um valor especial de bits de proteção.
- **Memória secundária**: mantém as páginas que não estão presentes na memória principal. A memória secundária é, usualmente, um disco de alta velocidade. Ela é conhecida como dispositivo de permuta, e a seção de disco usada para esse fim é conhecida como espaço de permuta.

Um requisito crucial da paginação por demanda é a capacidade de reiniciar qualquer instrução após um erro de página. Já que salvamos o estado (registradores, código de condição, contador de instruções) do processo interrompido quando o erro de página ocorre, devemos ser capazes de reiniciar o processo exatamente no mesmo local e estado, exceto pelo fato de a página desejada estar agora na memória

e poder ser acessada. Na maioria dos casos, esse requisito é fácil de alcançar. Um erro de página pode ocorrer em qualquer referência à memória. Se ele ocorrer na busca da instrução, poderemos reiniciar procurando a instrução novamente. Se um erro de página ocorrer enquanto estivermos buscando um operando, devemos buscar e decodificar a instrução novamente e, então, procurar o operando.

Como exemplo de um cenário extremo, considere uma instrução de três endereços como ADD (somar) o conteúdo de A e B, colocando o resultado em C. Seguindo os seguintes passos para executar essa instrução:

1) Buscar e decodificar a instrução (ADD).

2) Buscar A.

3) Buscar B.

4) Somar A e B.

5) Armazenar a soma em C.

Se ocorrer um erro ao tentarmos armazenar em C (porque C está em uma página que não se encontra correntemente em memória), teremos que obter a página desejada, trazê-la para a memória, corrigir a tabela de páginas e reiniciar a instrução. O reinício demandará a busca da instrução novamente, uma nova decodificação, a busca dos dois operandos mais uma vez e, então, uma nova soma. No entanto, não há muito trabalho repetido (menos de uma instrução completa) e a repetição é necessária somente quando ocorre um erro de página.

A maior dificuldade surge quando uma instrução pode modificar várias locações diferentes. Por exemplo, considere a instrução MVC (mover caractere) do sistema IBM 360/370 que pode mover até 256 bytes de uma locação para outra (possivelmente com sobreposição). Se um dos blocos (origem ou destino) ultrapassar um limite de página, um erro de página pode ocorrer após a movimentação ser parcialmente executada. Além disso, se os blocos de origem e destino forem sobrepostos, o bloco de origem pode ter sido modificado, caso em que não podemos simplesmente reiniciar a instrução.

Esse problema pode ser resolvido de duas maneiras diferentes. Em uma solução, o microcódigo calcula e tenta acessar as duas extremidades dos dois blocos. Se um erro de página tiver que ocorrer, ele acontecerá nesse passo, antes de algo ser modificado. A movimentação pode, então, ser executada; sabemos que nenhum erro de página pode ocorrer, já que todas as páginas relevantes estão na memória. A outra solução usa registradores temporários para armazenar os valores das locações sobrepostas. Se houver um erro de página, todos os valores anteriores serão gravados de volta na memória antes que a interceptação ocorra. Essa ação restaura a memória ao seu estado anterior ao de início da instrução para que ela possa ser repetida.

Esse não é, de forma alguma, o único problema de arquitetura resultante da adição de páginas a uma arquitetura existente para permitir a paginação por demanda, mas ilustra algumas das dificuldades envolvidas. A paginação é adicionada entre a CPU e a memória em um sistema de computação. Ela deve ser totalmente transparente ao processo do usuário. Portanto, as pessoas assumem, com frequência, que a paginação pode ser adicionada a qualquer sistema. Embora essa suposição seja verdadeira em um ambiente de paginação sem demanda, onde um erro de página representa um erro fatal, ela não é verdadeira onde um erro de página significa apenas que uma página adicional deve ser trazida para a memória e o processo reiniciado.

**Saiba mais**

Para conhecer um pouco mais sobre memória virtual por paginação, leia o capítulo 10 do livro a seguir:

MACHADO, F. B.; MAIA, L. P. *Arquitetura de Sistemas Operacionais*. 5. ed. Rio de Janeiro: LTC, 2013.

## 8 SISTEMAS DE ARQUIVO

O armazenamento e a recuperação de informações são atividades essenciais para qualquer tipo de aplicação. Um processo deve ser capaz de ler e gravar de forma permanente um grande volume de dados em dispositivos como fitas e discos, além de poder compartilhá-los com outros processos. A maneira pela qual o sistema operacional estrutura e organiza estas informações é por intermédio da implementação de arquivos.

Os arquivos são gerenciados pelo sistema operacional de maneira a facilitar o acesso dos usuários ao seu conteúdo. A parte do sistema responsável por essa gerência é denominada **sistema de arquivos**. Ele é a parte mais visível de um sistema operacional, pois a manipulação de arquivos é uma atividade frequentemente realizada pelos usuários, devendo sempre ocorrer de maneira uniforme, independentemente dos diferentes dispositivos de armazenamento.

### 8.1 Conceituação arquivos

Um **arquivo** é constituído por informações logicamente relacionadas, onde tais informações podem representar instruções ou dados. Um arquivo executável, por exemplo, contém instruções compreendidas pelo processador, enquanto um arquivo de dados pode ser estruturado livremente como um arquivo-texto ou, de forma mais rígida, como em um banco de dados relacional. Um arquivo pode ser representado como um conjunto de registros definidos pelo sistema de arquivos, tornando seu conceito abstrato e generalista. A partir dessa definição, o conteúdo do arquivo pode ser manipulado seguindo conceitos preestabelecidos.

Os arquivos são armazenados pelo sistema operacional em diferentes dispositivos físicos, como fitas magnéticas, discos magnéticos e discos ópticos. O tipo de dispositivo no qual o arquivo é armazenado deve ser isolado pelo sistema operacional, de forma que exista uma independência entre os arquivos a serem manipulados e o meio de armazenamento.

Um arquivo é identificado por um nome, composto por uma sequência de caracteres. Em alguns sistemas, é feita uma distinção entre caracteres alfabéticos maiúsculos e minúsculos, regras como extensão máxima do nome e quais são os caracteres válidos também podem variar.

Em alguns sistemas operacionais, a identificação de um arquivo é composta por duas partes separadas com um ponto. A parte após o ponto é denominada extensão do arquivo e tem como finalidade identificar o conteúdo do arquivo. Assim, é possível convencionar que uma extensão TXT identifica um arquivo texto, enquanto EXE indica um arquivo executável. No quadro 7, são apresentadas algumas extensões de arquivos.

**Quadro 7 – Extensão de arquivos**

| Extensão | Descrição |
|---|---|
| ARQUIVO.BAS | Arquivo-fonte em BASIC |
| ARQUIVO.COB | Arquivo-fonte em COBOL |
| ARQUIVO.EXE | Arquivo executável |
| ARQUIVO.OBJ | Arquivo-objeto |
| ARQUIVO.PAS | Arquivo-fonte em Pascal |
| ARQUIVO.TXT | Arquivo-texto |

Fonte: Machado (2013, p. 262).

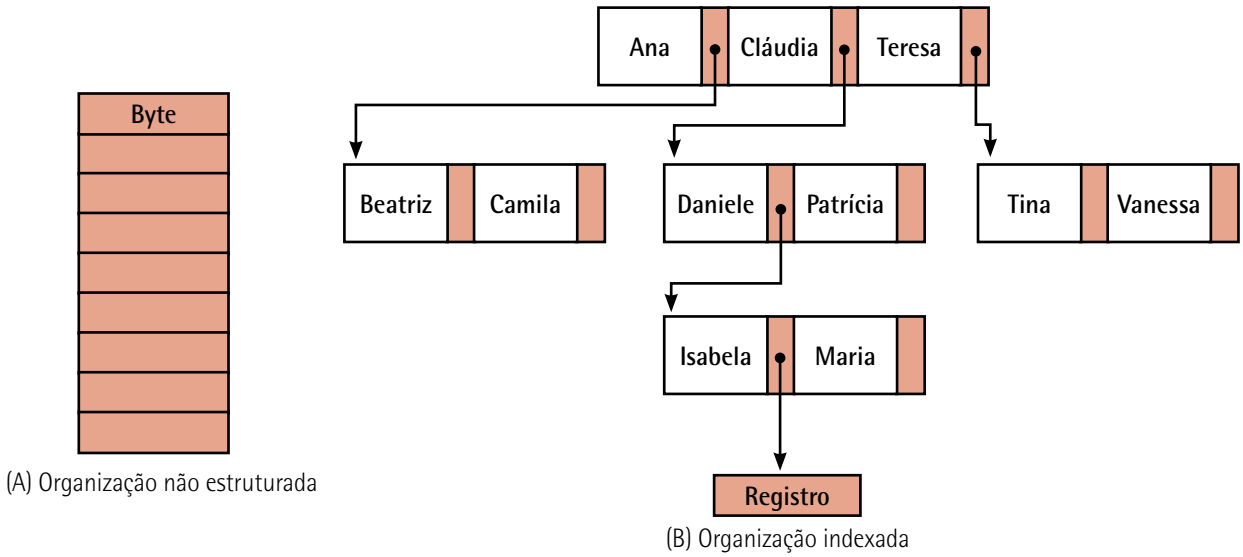
### Organização de arquivos

A **organização de arquivos** consiste em como os seus dados estão internamente armazenados. A estrutura dos dados pode variar em função do tipo de informação contida no arquivo. Arquivos-textos possuem propósitos completamente distintos de arquivos executáveis e, consequentemente, estruturas diferentes podem adequar-se melhor a um tipo do que a outro.

No momento da criação de um arquivo, seu criador pode definir qual a organização adotada. Ela pode ser uma estrutura suportada pelo sistema operacional ou definida pela própria aplicação.

A forma mais simples de organização de arquivos é através de uma sequência não estruturada de bytes. Nesse tipo de organização, o sistema de arquivos não impõe nenhuma estrutura lógica para os dados. A aplicação deve definir toda a organização, estando livre para estabelecer seus próprios critérios.

A grande vantagem desse modelo (figura 103) é a flexibilidade para criar diferentes estruturas de dados, porém todo o controle de acesso ao arquivo é de inteira responsabilidade da aplicação.




(A) Organização não estruturada

(B) Organização indexada

Figura 103 – Organização de arquivos

Fonte: Machado (2013, p. 264).

Alguns sistemas operacionais possuem diferentes organizações de arquivos. Nesse caso, cada arquivo criado deve seguir um modelo suportado pelo sistema de arquivos. As organizações mais conhecidas e implementadas são a sequencial, a relativa e a indexada. Nesses tipos de organização, podemos visualizar um arquivo como um conjunto de registros, que podem ser classificados em registros de tamanho fixo, quando possuírem sempre o mesmo tamanho, ou registros de tamanho variável.

### Métodos de acesso

Em função de como o arquivo está organizado, o sistema de arquivos pode recuperar registros de diferentes maneiras. Inicialmente, os primeiros sistemas operacionais só armazenavam arquivos em fitas magnéticas. Com isso, o acesso era restrito à leitura dos registros na ordem em que eram gravados, e a gravação de novos registros só era possível no final do arquivo. Esse tipo de acesso, chamado de acesso sequencial, era próprio da fita magnética, que, como meio de armazenamento, possuía esta limitação.

Com o desenvolvimento dos discos magnéticos, foi possível a introdução de métodos de acesso mais eficientes. O primeiro a surgir foi o acesso direto, que permite a leitura/gravação de um registro diretamente na sua posição. Esse método é realizado através do número do registro, que é a sua posição relativa ao início do arquivo. No acesso direto não existe restrição à ordem em que os registros são lidos ou gravados, sendo sempre necessária a especificação do número do registro. É importante notar que o acesso direto somente é possível quando o arquivo é definido com registros de tamanho fixo, onde

esse acesso pode ser combinado com o acesso sequencial. Com isso, é possível acessar diretamente um registro qualquer de um arquivo e, a partir deste, acessar sequencialmente os demais.

Um método de acesso mais sofisticado, que tem como base o acesso direto, é o chamado acesso indexado ou acesso por chave. Nesse caso, o arquivo deve possuir uma área de índice onde existam ponteiros para os diversos registros. Sempre que a aplicação desejar acessar um registro, deverá ser especificada uma chave através da qual o sistema pesquisará na área de índice o ponteiro correspondente. A partir dessa informação é realizado um acesso direto ao registro desejado.

### Operações de entrada/saída

O sistema de arquivos disponibiliza um conjunto de rotinas que permite às aplicações realizarem operações de E/S, como tradução de nomes em endereços, leitura e gravação de dados e criação/eliminação de arquivos. Na realidade, as rotinas de E/S têm como função disponibilizar uma interface simples e uniforme entre a aplicação e os diversos dispositivos. A figura 104 ilustra a comunicação entre aplicação e dispositivos de maneira simplificada.

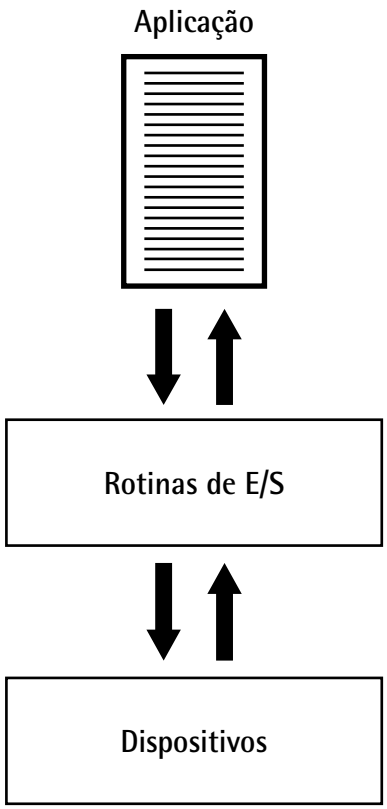



Figura 104 – Operações de entrada/saída

Fonte: Machado (2013, p. 264).

### Atributos

Cada arquivo possui informações de controle denominadas atributos. Eles variam dependendo do sistema de arquivos, porém, alguns, como tamanho do arquivo, proteção, identificação do criador e data de criação, estão presentes em quase todos os sistemas.

Alguns atributos especificados na criação do arquivo não podem ser modificados em função de sua própria natureza, como organização e data/hora de criação. Outros são alterados pelo próprio sistema operacional, como tamanho e data/hora do último backup realizado. Existem, ainda, atributos que podem ser modificados pelo próprio usuário, como proteção do arquivo, tamanho máximo e senha de acesso.

## 8.2 Diretórios

A estrutura de **diretórios** é como o sistema que organiza logicamente os diversos arquivos contidos em um disco, ou seja, é uma estrutura de dados que contém entradas associadas aos arquivos em que cada entrada armazena informações como localização física, nome, organização e demais atributos.

**Quadro 8 – Extensão de arquivos**

| Atributo | Descrição |
|---|---|
| Tamanho | Especifica o tamanho do arquivo |
| Proteção | Código de proteção de acesso |
| Dono | Identifica o criador do arquivo |
| Criação | Data e hora de criação do arquivo |
| Backup | Data e hora do último backup realizado |
| Organização | Indica a organização lógica dos registros |
| Senha | Senha necessária para acessar o arquivo |

Fonte: Machado (2013, p. 265).

Quando um arquivo é aberto, o sistema operacional procura a sua entrada na estrutura de diretórios, armazenando as informações sobre atributos e localização do arquivo em uma tabela mantida na memória principal. Essa tabela contém todos os arquivos abertos, sendo fundamental para aumentar o desempenho das operações com arquivos. É importante que ao término do uso de arquivos, eles sejam fechados, ou seja, que se libere o espaço na tabela de arquivos abertos.

A implementação mais simples de uma estrutura de diretórios é chamada de nível único (em inglês, single-level directory). Nesse caso, existe somente um único diretório contendo todos os arquivos do disco (figura 105). Esse modelo é bastante limitado, já que não permite que usuários criem arquivos com o mesmo nome, o que ocasionaria um conflito no acesso aos arquivos.

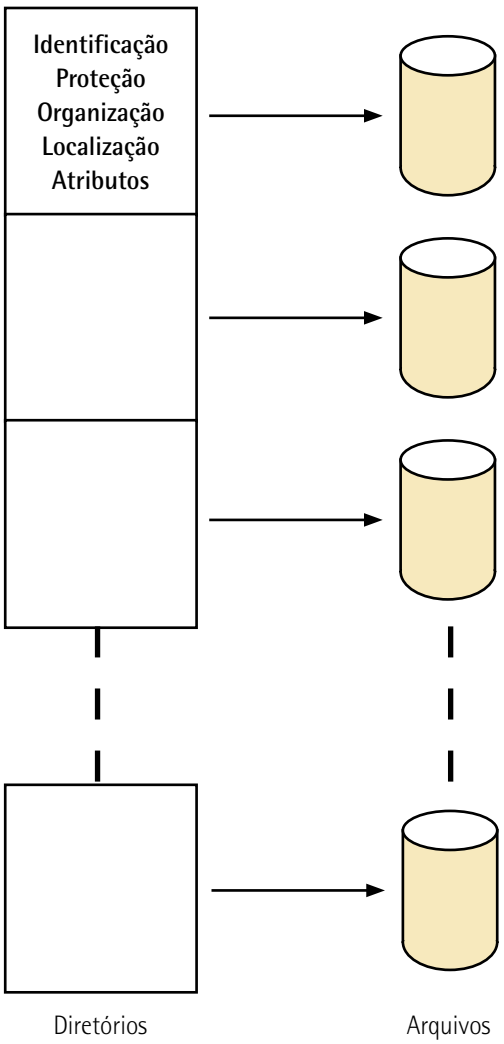



Figura 105 – Estrutura de diretórios de nível único

Fonte: Machado (2013, p. 266).

Como o sistema de nível único é bastante limitado, uma evolução do modelo foi a implementação de uma estrutura em que, para cada usuário, existiria um diretório particular denominado User File Directory (UFD). Com esta implementação, cada usuário passa a poder criar arquivos com qualquer nome, sem a preocupação de conhecer os demais arquivos do disco.

Para que o sistema possa localizar arquivos nessa estrutura, deve haver um nível de diretório adicional para controlar os diretórios individuais dos usuários. Esse nível, denominado Master File Directory (MFD), é indexado pelo nome do usuário e, nele, cada entrada aponta para o diretório pessoal. A figura 106 ilustra este modelo de estrutura de diretórios com dois níveis (em inglês, two-level directory).

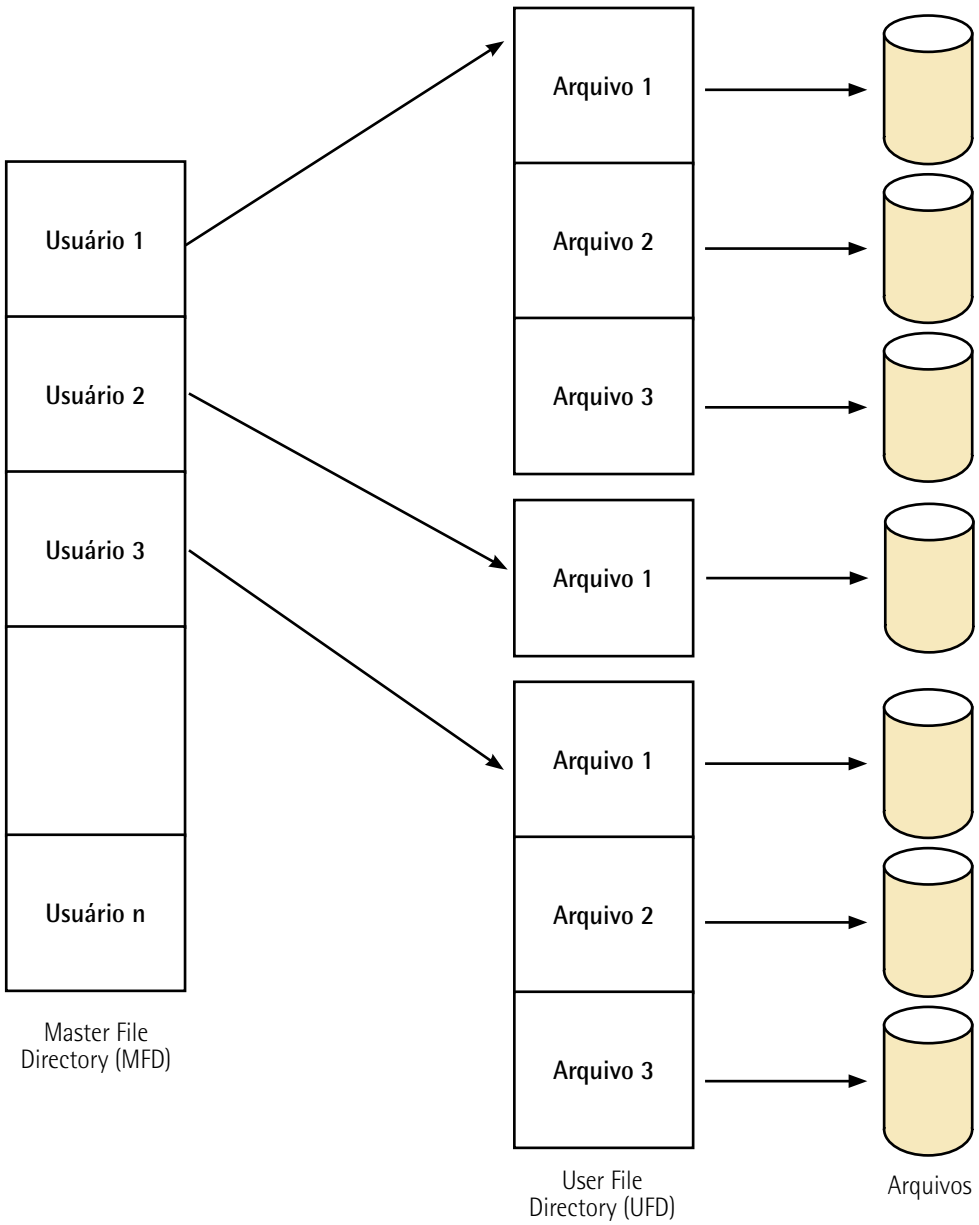



Figura 106 – Estrutura de diretórios com dois níveis

Fonte: Machado (2013, p. 270).

A estrutura de diretórios com dois níveis é análoga a uma estrutura de dados em árvore, em que o MFD é a raiz, os galhos são os UFD e os arquivos são as folhas. Nesse tipo de estrutura, quando se referencia um arquivo, é necessário especificar, além do seu nome, o diretório onde ele se localiza. Essa referência é chamada de path (caminho). Como exemplo, caso o usuário CARLOS necessite acessar um arquivo próprio

chamado DOCUMENTO.TXT, este pode ser referenciado como /CARLOS/DOCUMENTO.TXT. Cada sistema de arquivos possui sua própria sintaxe para especificação de diretórios e arquivos.

Sob o ponto de vista do usuário, a organização dos seus arquivos em um único diretório não permite uma organização adequada. Porém, a extensão do modelo de dois níveis para um de múltiplos níveis permite que os arquivos sejam logicamente mais bem organizados. Este novo modelo, chamado de estrutura de diretórios em árvore (tree-structured directory), é adotado pela maioria dos sistemas.

Na estrutura em árvore, cada usuário pode criar diversos níveis de diretórios, também chamados subdiretórios. Cada diretório pode conter arquivos ou outros diretórios. O número de níveis de uma estrutura em árvore é dependente do sistema de arquivos de cada sistema operacional. No Open VMS são possíveis oito níveis de diretórios.

Um arquivo, nesta estrutura em árvore, pode ser especificado unicamente por meio de um path absoluto, descrevendo todos os diretórios percorridos a partir da raiz (MFD) até o diretório no qual o arquivo está ligado. Na maioria dos sistemas, os diretórios também são tratados como arquivos, possuindo identificação e atributos, como proteção, identificador do criador e data de criação.

**Lembrete**

Os arquivos ainda podem ser encontrados em tipos, como:

- **Estruturados**: seguem um formato predefinido e rígido, com regras claras para organização dos dados. Por exemplo: planilhas Excel, bancos de dados relacionais, arquivos XML etc.
- **Semiestruturados**: possuem alguma estrutura, mas não são tão rígidos quanto os estruturados. Por exemplo: JSON, CSV, XML (com estrutura menos definida), documentos HTML etc.
- **Não estruturados**: não possuem uma estrutura predefinida. Por exemplo: documentos de texto, imagens, áudio, vídeos, código-fonte de software etc.

## 8.3 Compartilhamento

O compartilhamento de arquivos é desejável para usuários que querem colaborar e reduzir o esforço requerido para alcançar um objetivo computacional. Assim, sistemas operacionais orientados ao usuário devem acomodar a necessidade de compartilhar arquivos, apesar das dificuldades inerentes.

Ao adentrarmos o tópico de compartilhamento de arquivos, é importante discutirmos questões gerais que surgem quando múltiplos usuários compartilham arquivos. Uma vez que múltiplos usuários possuem permissão para partilhar arquivos, o desafio é estender o compartilhamento a múltiplos sistemas de arquivos, incluindo sistemas de arquivos remotos. Dessa forma, devemos considerar o que

deve ser feito em relação às ações conflitantes que ocorrem em arquivos compartilhados. Por exemplo, se múltiplos usuários estão gravando em um arquivo, todas as gravações devem ser permitidas, ou o sistema operacional deve proteger os usuários das ações uns dos outros?

### Múltiplos usuários

Quando um sistema operacional acomoda múltiplos usuários, as questões de compartilhamento, nomeação e proteção de arquivos tornam-se proeminentes. Dada uma estrutura de diretório que permita o compartilhamento de arquivos pelos usuários, o sistema deve mediar esse compartilhamento. O sistema pode permitir que um usuário acesse os arquivos de outros usuários por default, ou pode requerer que um usuário conceda especificamente acesso aos arquivos.

Para implementar o compartilhamento e a proteção, o sistema deve manter mais atributos de arquivo e diretório do que os necessários em um sistema monousuário. Embora muitas abordagens tenham sido adotadas para atender a esse requisito, a maioria dos sistemas evoluiu para o uso dos conceitos de proprietário (ou usuário) e grupo de arquivos (ou diretórios), sendo que o proprietário é o usuário que pode alterar atributos e conceder acesso e é quem tem mais controle sobre o arquivo. O atributo de grupo define um subconjunto de usuários que podem compartilhar o acesso ao arquivo. Por exemplo, o proprietário de um arquivo em um sistema Unix pode executar todas as operações sobre ele, enquanto os membros do grupo do arquivo podem executar um subconjunto dessas operações e todos os outros usuários podem executar outro subconjunto de operações. O proprietário do arquivo é quem define exatamente que operações podem ser executadas por membros do grupo e por outros usuários.

Os IDs de proprietário e de grupo de determinado arquivo (ou diretório) são armazenados com os outros atributos do arquivo. Quando um usuário solicita uma operação sobre um arquivo, seu ID é comparado com o atributo de proprietário para determinar se o usuário solicitante é o proprietário do arquivo. Da mesma forma, os IDs de grupo podem ser comparados. O resultado indica que permissões são aplicáveis. O sistema aplica tais permissões à operação solicitada e ela é autorizada ou negada.

Muitos sistemas têm vários sistemas de arquivos locais, incluindo volumes em um único disco ou múltiplos volumes em múltiplos discos. Nesses casos, a verificação do ID e das permissões é simples, contanto que os sistemas de arquivos estejam montados.

### Sistemas de arquivos remotos

A comunicação entre computadores remotos tornou-se possível com o desenvolvimentos das redes. A comunicação em rede permite o compartilhamento de recursos espalhados por um campus ou até mesmo ao redor do globo. Um recurso óbvio a ser compartilhado são os dados em forma de arquivos.

Com a evolução da tecnologia de redes e arquivos, os métodos de compartilhamento remoto de arquivos mudaram. O primeiro método implementado envolve a transferência manual de arquivos entre máquinas por meio de programas como o File Transfer Protocol (FTP). O segundo grande método usa um sistema de arquivos distribuído, Distributed File System (DFS), em que diretórios remotos são visíveis a partir de um computador local. Em alguns aspectos, o terceiro método, a World Wide Web, é uma volta

ao primeiro. É necessário um navegador para a obtenção de acesso aos arquivos remotos e são usadas operações separadas (essencialmente um encapsulador do FTP) para transferir arquivos. Cada vez mais, a computação em nuvem vem sendo usada para o compartilhamento de arquivos.

O FTP é usado tanto para acesso anônimo quanto para acesso autenticado, já que o acesso anônimo permite que o usuário transfira arquivos sem possuir uma conta no sistema remoto. A World Wide Web usa, quase exclusivamente, a troca anônima de arquivos; enquanto que o DFS envolve uma integração maior entre a máquina que está acessando os arquivos remotos e a máquina que os fornece.

**Sistemas de informação distribuídos**

Para tornar os sistemas cliente-servidor mais fáceis de gerenciar, os sistemas de informação distribuídos, também conhecidos como serviços de nomeação distribuídos, fornecem acesso unificado às informações requeridas pela computação remota. O sistema de nomes de domínio, Domain Name System (DNS), fornece traduções do nome de hospedeiro para o endereço-de-rede em toda a Internet. Antes de o DNS se tornar comum, arquivos contendo as mesmas informações eram enviados, por e-mail ou por FTP, para todos os hospedeiros da rede. No entanto, essa metodologia não era escalável.

Outros sistemas de informação distribuídos fornecem um espaço de nome de usuário/senha/ID de usuário e de grupo para um recurso distribuído. Os sistemas Unix têm empregado uma grande variedade de métodos de informações distribuídas. A Sun Microsystems (agora parte da Oracle Corporation) introduziu as páginas amarelas (depois renomeadas como serviço de informações de rede – NIS) e a maior parte da indústria adotou seu uso. O NIS centraliza o armazenamento de nomes de usuário, nomes de hospedeiro, informações de impressora e assim por diante.

Infelizmente, ele usa métodos de autenticação inseguros, incluindo o envio de senhas de usuário não criptografadas (em texto literal) e a identificação de hospedeiros por endereços IP. O NIS+ da Sun é um substituto muito mais seguro do NIS, mas é muito mais complicado e não foi amplamente adotado.

No caso do Common Internet File System (CIFS) da Microsoft, informações de rede são usadas em conjunção com a autenticação do usuário (nome de usuário e senha) para criar um login de rede, esse login é usado pelo servidor que decide se deve permitir ou negar acesso ao sistema de arquivos solicitado. Para que essa autenticação seja válida, os nomes de usuário devem coincidir com os nomes de um computador para outro (como no NFS). A Microsoft usa o diretório ativo como uma estrutura de nomeação distribuída, a fim de fornecer um único espaço de nomes para os usuários. Uma vez estabelecido, o recurso de nomeação distribuída é usado por todos os clientes e servidores para autenticar usuários.

A indústria está se movendo em direção ao protocolo peso-leve de acesso a diretórios, Lightweight Directory Access Protocol (LDAP), sendo ele um mecanismo seguro de nomeação distribuída e um diretório ativo baseado no LDAP. O Solaris, da Oracle, assim como a maioria dos outros grandes sistemas operacionais, inclui o LDAP em sua estrutura e, com isso, permite que ele seja empregado para autenticação de usuários, assim como para recuperação de informações em todo o sistema. Dessa forma, se faz compreensível como um diretório LDAP distribuído poderia ser usado por uma empresa para armazenar todas as informações dela, principalmente aquelas sobre usuários e recursos dos computadores da organização. O resultado do uso de LDAP é uma assinatura única e segura para

os usuários que inseririam suas informações de autenticação apenas uma vez para acessar todos computadores da empresa. Além disso, o uso também facilitaria os esforços de administração do sistema ao combinar, na mesma locação, informações correntemente espalhadas em vários arquivos em cada sistema ou em diferentes serviços de informações distribuídos.

## 8.4 Implementação

Várias estruturas em disco e em memória são usadas para implementar um sistema de arquivos. Elas variam dependendo do sistema operacional e do sistema de arquivos, mas alguns princípios gerais são aplicáveis. Em disco, o sistema de arquivos pode conter informações sobre como inicializar um sistema operacional, já que, nele, há estruturas como:

- **Bloco de controle de inicialização (por volume)**: pode conter as informações requeridas pelo sistema para inicializar um sistema operacional a partir desse volume. Se o disco não contiver um sistema operacional, esse bloco pode estar vazio. Ele é normalmente o primeiro bloco de um volume, sendo que, no UFS, ele é chamado de bloco de inicialização. Já no NTFS, ele pertence ao setor de inicialização de partição.

- **Bloco de controle de volume (por volume)**: contém detalhes do volume (ou partição), tais como o número de blocos na partição, o tamanho dos blocos, uma contagem e ponteiros de blocos livres, e uma contagem de FCBs livres e ponteiros para FCBs. No UFS, esse bloco é chamado de superbloco; no NTFS, ele é armazenado na tabela de arquivos mestre.

- **Estrutura de diretório (por sistema de arquivos)**: é usada para organizar arquivos. No UFS, ela inclui os nomes de arquivo e os números de inode associados; já no NTFS, a estrutura é armazenada na tabela de arquivos mestre.

- **Arquivos individuais**: um FCB por arquivo contém muitos detalhes sobre o arquivo. O FCB possui um número identificador exclusivo para permitir a associação a uma entrada do diretório. No NTFS, essas informações são realmente armazenadas dentro da tabela de arquivos mestre que usa uma estrutura de banco de dados relacional, com uma linha por arquivo.

As informações em memória são usadas tanto no gerenciamento do sistema de arquivos quanto na melhoria do desempenho por meio do armazenamento em cache. Os dados são carregados em tempo de montagem, atualizados durante operações sobre o sistema de arquivos e descartados na desmontagem. Vários tipos de estruturas podem ser incluídos, como:

- Uma tabela de montagens em memória contém informações sobre cada volume montado.

- Um cache em memória da estrutura de diretórios mantém as informações referentes aos diretórios acessados recentemente. Para diretórios nos quais há volumes montados, ele pode conter um ponteiro para a tabela de volumes.

- A tabela de arquivos abertos em todo o sistema contém uma cópia do FCB de cada arquivo aberto, assim como outras informações.

- A tabela de arquivos abertos por processo contém um ponteiro para a entrada apropriada na tabela de arquivos abertos em todo o sistema, assim como outras informações.

- Buffers mantêm blocos do sistema de arquivos quando eles estão sendo lidos do disco ou gravados em disco.

Para criar um novo arquivo, um programa de aplicação chama o sistema de arquivos lógico, onde ele conhece o formato das estruturas de diretório, alocando um novo FCB para criar um novo arquivo. Alternativamente, se a implementação do sistema de arquivos gera todos os FBCs em tempo de criação do sistema de arquivos, um FCB é alocado a partir do conjunto de FCBs livres. O sistema, então, lê o diretório apropriado para a memória, atualiza-o com o nome e o FCB do novo arquivo e grava-o de volta em disco. Alguns sistemas operacionais, inclusive o Unix, tratam um diretório exatamente como um arquivo, com um campo "tipo" indicando que é um diretório. Outros sistemas operacionais, por exemplo o Windows, implementam chamadas de sistema separadas para arquivos e diretórios, tratando este último como entidades separadas dos arquivos.

Independentemente das questões estruturais de maior abrangência, o sistema de arquivos lógico pode chamar o módulo de organização de arquivos para mapear o I/O de diretório, onde números de blocos de disco são passados ao sistema de arquivos básico e ao sistema de controle de I/O, mapeando os arquivos de acordo com: permissões; datas (criação, acesso, gravação etc.); proprietário do arquivo, grupo, ACL; tamanho do arquivo; e blocos de dados do arquivo ou ponteiros para os blocos de dados do arquivo.

Agora que um arquivo foi criado, ele pode ser usado para I/O, mas, primeiro, ele deve ser aberto. A chamada open() passa um nome de arquivo para o sistema de arquivos lógico. Assim, a chamada de sistema open() pesquisa, em primeiro lugar, a tabela de arquivos abertos em todo o sistema para ver se o arquivo já está sendo usado por outro processo. Se estiver, é criada uma entrada na tabela de arquivos abertos por processo, apontando para a tabela de arquivos abertos em todo o sistema existente, onde esse algoritmo poderá evitar um overhead significativo. Se o arquivo ainda não estiver aberto, a estrutura do diretório será pesquisada em busca do nome de arquivo fornecido. Partes da estrutura do diretório são usualmente armazenadas no cache em memória para acelerar as operações de diretório. Uma vez que o arquivo seja encontrado, o FCB é copiado em uma tabela de arquivos abertos em todo o sistema, em memória.

Essa tabela não apenas armazena o FCB, mas também registra o número de processos que estão com o arquivo aberto. Em seguida, é criada uma entrada na tabela de arquivos abertos por processo, com um ponteiro para a entrada na tabela de arquivos abertos em todo o sistema e alguns outros campos. Esses outros campos podem incluir um ponteiro para a locação corrente no arquivo – para a próxima operação read() ou write() – e a modalidade de acesso para a qual o arquivo foi aberto. A chamada open() retorna um ponteiro para a entrada apropriada na tabela de sistemas de arquivos por processo. Todas as operações de arquivo são, então, executadas através desse ponteiro. O nome do arquivo pode não fazer parte da tabela de arquivos abertos, já que o sistema não precisa usá-lo, uma vez que o FCB apropriado seja localizado em disco. Ele poderia ser armazenado em cache, no entanto, para economizar tempo em aberturas subsequentes do mesmo arquivo. Devido a isso, o nome dado à entrada varia. Sistemas Unix referem-se a ela como descritor de arquivos; já o Windows a chama de manipulador de arquivos.

Quando um processo fecha o arquivo, a entrada na tabela por processo é removida e a contagem de aberturas da entrada na tabela, em todo o sistema, é decrementada. Quando todos os usuários que tenham aberto o arquivo o fecharem, qualquer metadado atualizado será copiado de volta na estrutura do diretório baseada em disco, e a entrada na tabela de arquivos abertos em todo o sistema será removida.

Alguns sistemas complicam ainda mais esse esquema usando o sistema de arquivos como uma interface para outros aspectos do sistema, tal como a conexão em rede. Por exemplo, no UFS, a tabela de arquivos abertos em todo o sistema mantém os inodes e outras informações de arquivos e diretórios. Além disso, o sistema também mantém informações semelhantes sobre conexões e dispositivos de rede. Dessa forma, um mecanismo pode ser usado para múltiplas finalidades.

Os aspectos do armazenamento em cache das estruturas dos sistemas de arquivos não devem ser ignorados. A maioria dos sistemas mantém em memória todas as informações sobre um arquivo aberto, exceto seus blocos de dados reais. O sistema BSD Unix é típico em seu uso de caches onde quer que o I/O de disco possa ser reduzido. Sua taxa média de acessos ao cache, da ordem de 85%, mostra que vale a pena implementar essas técnicas.

## 8.5 Métodos de alocação e gerenciamento de espaço

A natureza de acesso direto dos discos oferece-nos flexibilidade na implementação de arquivos. Em quase todos os casos, muitos arquivos são armazenados no mesmo disco. O principal problema é como alocar espaço para esses arquivos, de modo que o espaço em disco seja eficientemente utilizado e os arquivos possam ser rapidamente acessados. Três métodos principais de alocação de espaço em disco são muito usados: contíguo, encadeado e indexado. Cada método apresenta vantagens e desvantagens, sendo que, embora alguns sistemas suportem todos os três, o mais comum é que um sistema utilize um método para todos os arquivos de um tipo de sistema de arquivos.

### Alocação contígua

A alocação contígua requer que cada arquivo ocupe um conjunto de blocos contíguos em disco. Os endereços de disco definem uma ordenação linear dele. Com essa ordenação, supondo que apenas um job esteja acessando o disco, o acesso ao bloco $b + 1$ após o bloco $b$, normalmente, não requer movimentação do cabeçote. Quando tal movimentação é necessária (do último setor de um cilindro para o primeiro setor do próximo cilindro), o cabeçote precisa apenas mover-se de uma trilha para a seguinte. Portanto, o número de buscas em disco requeridas para acessar arquivos alocados contiguamente é mínimo, assim como o tempo de busca quando ela é finalmente necessária.

A alocação contígua de um arquivo é definida pelo endereço de disco do primeiro bloco e tamanho do arquivo (em unidades do bloco). Se o arquivo tem $n$ blocos de tamanho e começa na locação $b$, então ele ocupa os blocos $b$, $b + 1$, $b + 2$, ..., $b + n - 1$. A entrada no diretório para cada arquivo indica o endereço do bloco inicial e o tamanho da área alocada para esse arquivo.

É fácil acessar um arquivo que tenha sido alocado contiguamente. No acesso sequencial, o sistema de arquivos lembra o endereço em disco do último bloco referenciado e, quando necessário, lê o

próximo bloco. No acesso direto ao bloco *i* de um arquivo que começa no bloco *b*, podemos acessar imediatamente o bloco *b* + *i*. Assim, tanto o acesso sequencial quanto o acesso direto podem ser suportados na alocação contígua.

No entanto, a alocação contígua apresenta alguns problemas. Uma das dificuldades é encontrar espaço para um novo arquivo, já que o sistema selecionado é o responsável por gerenciar o espaço livre que determinará como essa tarefa é executada. Qualquer sistema de gerenciamento pode ser usado, mas alguns são mais lentos do que outros.

O problema da alocação contígua pode ser visto como uma aplicação particular do problema geral de alocação de memória dinâmica, que diz respeito a como atender a uma solicitação de tamanho *n* a partir de uma lista de brechas livres. O primeiro-apto e o mais-apto são as estratégias mais comuns usadas para selecionar uma brecha livre no conjunto de brechas disponíveis. Simulações têm mostrado que tanto o primeiro-apto quanto o mais-apto são mais eficientes do que o menos-apto em termos de tempo e utilização da memória. Nem o primeiro-apto, nem o mais-apto é claramente melhor em termos de utilização da memória, mas o primeiro-apto é geralmente mais rápido.

Todos esses algoritmos sofrem do problema de fragmentação externa e, conforme os arquivos são alocados e excluídos, o espaço livre em disco é quebrado em pequenas partes. A fragmentação externa ocorre sempre que esse espaço é quebrado em porções, se tornando um problema quando a maior porção contígua é insuficiente para uma solicitação. Isso faz  com que o armazenamento seja fragmentado em várias brechas, quando nenhuma delas é suficientemente grande para armazenar os dados.

**Observação**

Dependendo do montante total de memória em disco e do tamanho médio do arquivo, a fragmentação externa pode ser um problema maior ou menor.

Uma estratégia para evitar a perda de montantes significativos de espaço em disco, em razão da fragmentação externa, é copiar um sistema de arquivos inteiro em outro disco. Assim, o disco original é totalmente liberado, criando um grande espaço livre contíguo. Após isso, copiamos os arquivos de volta no disco original alocando espaço contíguo a partir dessa grande brecha. Esse esquema compacta efetivamente todo o espaço livre em um espaço contíguo, resolvendo o problema de fragmentação. No entanto, o custo dessa compactação é refletido no tempo e pode ser particularmente alto para grandes discos rígidos. A compactação desses discos pode levar horas e pode ser necessária semanalmente. Alguns sistemas requerem que essa função seja executada off-line, com o sistema de arquivos desmontado.

Durante esse tempo de baixa, a operação normal do sistema geralmente não pode ser permitida e, portanto, tal compactação é evitada a todo custo em máquinas de produção. A maioria dos sistemas modernos que precisa de desfragmentação pode executá-la on-line durante as operações normais do sistema, mas a perda de desempenho pode ser significativa.

Outro problema da alocação contígua é determinar quanto espaço é necessário para um arquivo. Quando o arquivo é criado, o montante total de espaço de que ele precisará deve ser encontrado e alocado. Como o criador (programa ou pessoa) sabe o tamanho do arquivo a ser criado? Em alguns casos, essa determinação pode ser bem simples (ao copiar um arquivo existente, por exemplo). Em geral, no entanto, pode ser difícil estimar o tamanho de um arquivo de saída.

Se alocarmos um espaço muito pequeno para um arquivo, podemos acabar descobrindo que o arquivo não pode ser estendido. Principalmente em uma estratégia de alocação do mais-apto, o espaço nos dois lados do arquivo pode estar em uso. Portanto, não podemos tornar o arquivo maior *in loco*, podendo haver duas possibilidades. Na primeira, o programa do usuário pode ser encerrado, apresentando uma mensagem de erro apropriada; então, o usuário deve alocar mais espaço e executar o programa novamente. Essas execuções repetidas podem ser dispendiosas e, para evitá-las, o usuário normalmente superestima o montante de espaço necessário, o que resulta em considerável desperdício de espaço. A outra possibilidade é encontrar uma brecha maior, copiar o conteúdo do arquivo para o novo espaço e liberar o espaço anterior.

Essa série de ações pode ser repetida enquanto houver espaço, mas pode ser demorada. No entanto, o usuário nunca precisa ser informado explicitamente sobre o que está ocorrendo, a execução do sistema continua apesar do problema, embora cada vez mais lenta.

Mesmo se o montante total de espaço necessário para um arquivo for conhecido antecipadamente, a pré-alocação pode ser ineficiente. Um arquivo que cresce lentamente durante um longo período (meses ou anos), deve receber espaço suficiente para seu tamanho final, ainda que grande parte desse espaço não seja usado por um longo tempo. O arquivo terá, então, um grande montante de fragmentação interna.

Para minimizar essas inconveniências, alguns sistemas operacionais usam um esquema modificado de alocação contígua. Aqui, uma porção de espaço contíguo é alocada inicialmente. Então, se esse montante de espaço não se mostrar suficientemente grande, outra porção de espaço contíguo, conhecida como extensão, é adicionada. A locação dos blocos de um arquivo é, então, registrada como uma locação e uma contagem de blocos, com a adição de um link para o primeiro bloco da próxima extensão. Em alguns sistemas, o proprietário do arquivo pode definir o tamanho da extensão, mas essa definição resulta em ineficiências se o proprietário estiver errado. A fragmentação interna ainda pode ser um problema se as extensões forem grandes demais, e a fragmentação externa pode se tornar um problema à medida que extensões de vários tamanhos sejam alocadas e desalocadas. O sistema de arquivos comercial Veritas usa extensões para otimizar o desempenho, além de ser um substituto de alto desempenho para o UFS padrão do Unix.

### Alocação encadeada

A alocação encadeada resolve todos os problemas da alocação contígua. Nela, cada arquivo é uma lista encadeada de blocos de disco, podendo estar espalhados em qualquer local nele. O diretório contém um ponteiro para o primeiro e o último bloco do arquivo. Por exemplo, um arquivo de cinco blocos pode começar no bloco 9 e continuar no bloco 16, passar ao bloco 1, depois ao bloco 10 e, finalmente,

ao bloco 25. Cada bloco contém um ponteiro para o próximo bloco, sendo que esses ponteiros não se tornam disponíveis para o usuário. Assim, se cada bloco tem 512 bytes e um endereço de disco (o ponteiro) requer 4 bytes, então, o usuário enxerga blocos de 508 bytes.

Para criar um novo arquivo, simplesmente criamos uma nova entrada no diretório. Na alocação encadeada, cada entrada no diretório tem um ponteiro para o primeiro bloco de disco do arquivo. Esse ponteiro é inicializado com null (o valor do ponteiro para o fim da lista) para indicar um arquivo vazio. O campo de tamanho também é posicionado em 0. Uma gravação no arquivo obriga o sistema de gerenciamento de espaços livres a encontrar um bloco livre, e esse novo bloco recebe a gravação, sendo encadeado ao fim do arquivo, assim, para ler um arquivo, simplesmente lemos os blocos seguindo os ponteiros de um bloco a outro. Não há fragmentação externa na alocação encadeada, qualquer bloco livre na lista de espaços livres pode ser usado para atender a uma solicitação. Além disso, o tamanho de um arquivo não precisa ser declarado quando o arquivo é criado, podendo continuar a crescer enquanto existam blocos livres disponíveis e, consequentemente, nunca é necessário compactar o espaço em disco.

A solução usual para esse problema é reunir blocos em múltiplos, chamados clusters, e aloca-los em vez de blocos. Por exemplo, o sistema de arquivos pode definir um cluster como quatro blocos e operar sobre o disco apenas em unidades de cluster. Assim, os ponteiros usarão um percentual muito menor do espaço do arquivo em disco. Esse método permite que o mapeamento de blocos lógicos para físicos permaneça simples, além de melhorar o throughput do disco (porque são requeridas menos buscas do cabeçote do disco) ao diminuir o espaço necessário para a alocação de blocos e o gerenciamento da lista de blocos livres. O custo dessa abordagem é um aumento na fragmentação interna, devido ao fato de que mais espaço é desperdiçado quando um cluster está parcialmente cheio do que quando um bloco está parcialmente cheio. Os clusters também podem ser usados para melhorar o tempo de acesso ao disco em muitos outros algoritmos, logo, eles são usados na maioria dos sistemas de arquivos.

Ainda, outro problema da alocação encadeada é a confiabilidade. Como já tratamos neste livro-texto, os arquivos são encadeados por ponteiros espalhados por todo o disco e, ao termos um ponteiro perdido ou danificado, é importante considerar o que ocorreria nesse cenário, já que um bug no software do sistema operacional ou uma falha no hardware do disco podem resultar na seleção do ponteiro errado. Por sua vez, esse erro poderia resultar no encadeamento à lista de espaços livres ou a outro arquivo. Uma solução parcial seria usar listas duplamente encadeadas; a outra seria armazenar o nome do arquivo e o número de bloco relativo em cada bloco. No entanto, esses esquemas requerem ainda mais overhead para cada arquivo.

Uma variação importante da alocação encadeada é o uso de uma tabela de alocação de arquivos, Fat Allocation Table (FAT). Esse método simples, mas eficiente, de alocação de espaço em disco foi utilizado pelo sistema operacional MS-DOS. Uma seção de disco no começo de cada volume é reservada para conter a tabela que contém uma entrada para cada bloco do disco e é indexada pelo número do bloco. A FAT é usada de maneira semelhante a uma lista encadeada.

A entrada no diretório contém o número do primeiro bloco do arquivo; por conseguinte, a entrada na tabela indexada por esse número de bloco contém o número do próximo bloco do arquivo. Essa cadeia continua até alcançar o último bloco, que possui um valor especial de fim de arquivo como entrada na tabela, onde um bloco não usado é indicado por um valor de tabela igual a 0.

A alocação de um novo bloco a um arquivo requere a simples tarefa de encontrar a primeira entrada na tabela com valor 0 e substituir o valor anterior de fim de arquivo pelo endereço do novo bloco. O 0 é, então, substituído pelo valor de fim de arquivo.

Um exemplo ilustrativo é a estrutura da FAT mostrada na figura 107, onde temos um arquivo composto pelos blocos de disco 217, 618 e 339.

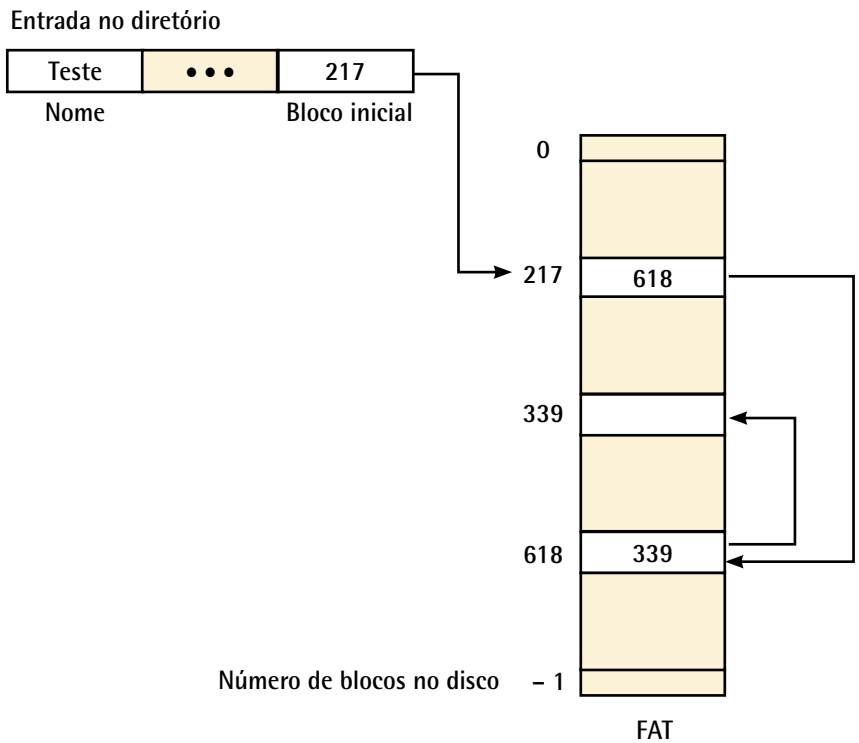



Figura 107 – Tabela de alocação de arquivo

Fonte: Silberschartz (2015, p. 453).

O esquema de alocação por FAT pode resultar em um número significativo de buscas do cabeçote do disco, a menos que a FAT seja armazenada em cache. O cabeçote do disco deve movimentar-se até o início do volume para ler a FAT, encontrar a locação do bloco em questão e, por fim, movimentar-se para a locação do próprio bloco. No pior caso, as duas movimentações ocorrerão para cada um dos blocos. Uma vantagem é que o tempo de acesso randômico é melhorado porque o cabeçote do disco pode encontrar a locação de qualquer bloco lendo as informações na FAT.

### Desempenho

Os métodos de alocação que discutimos variam em sua eficiência de armazenamento e nos tempos de acesso aos blocos de dados. Os dois são critérios importantes na seleção do(s) método(s) apropriado(s) para serem implementados por um sistema operacional. Antes de selecionar um método de alocação, precisamos determinar como os sistemas serão usados, já que um sistema com acesso, principalmente sequencial, não deve usar o mesmo método de um sistema com acesso predominantemente randômico.

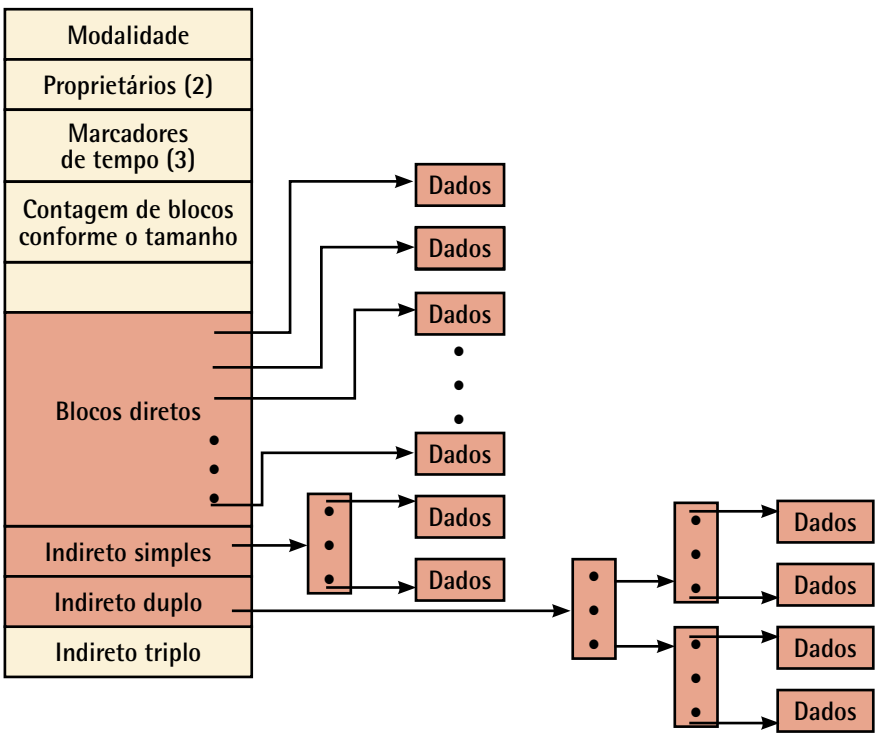



Figura 108 – O inode do Unix

Fonte: Silberschartz (2015, p. 455).

Para qualquer tipo de acesso, a alocação contígua requer apenas um acesso para alcançar um bloco de disco. Desse modo, podemos manter facilmente o endereço inicial do arquivo na memória, podendo calcular imediatamente o endereço de disco do *i*-ésimo bloco (ou do próximo bloco) e lê-lo diretamente.

Na alocação encadeada, também podemos manter o endereço do próximo bloco na memória e lê-lo diretamente. Esse método é adequado ao acesso sequencial; para o acesso direto, no entanto, um acesso ao *i*-ésimo bloco pode requerer *i* leituras em disco. Esse problema mostra por que a alocação encadeada não deve ser usada para uma aplicação que requeira acesso direto.

Como resultado, alguns sistemas suportam arquivos de acesso direto usando a alocação contígua, enquanto os arquivos de acesso sequencial usam a alocação encadeada. Nesses sistemas, o tipo de acesso a ser feito deve ser declarado quando o arquivo é criado. No caso de um arquivo criado para acesso sequencial, este será encadeado e não poderá ser usado para acesso direto.

Um arquivo criado para acesso direto será contíguo e poderá suportar tanto o acesso direto quanto o sequencial, mas seu tamanho máximo deve ser declarado quando ele é criado. Nesse caso, o sistema operacional deve ter estruturas de dados e algoritmos apropriados para suportar os dois métodos de alocação. Os arquivos podem ser convertidos de um tipo para outro por meio da criação de um novo arquivo do tipo desejado, para o qual o conteúdo do arquivo antigo é copiado. O arquivo antigo pode, então, ser excluído e o novo arquivo renomeado.

A alocação indexada é mais complexa. Se o bloco de índices já estiver na memória, o acesso poderá ser feito diretamente. No entanto, manter o bloco de índices na memória requer um espaço considerável, sendo que, se esse espaço não estiver disponível, podemos precisar ler primeiro o bloco de índices e, depois, o bloco de dados desejado.

Em um índice de dois níveis, podem ser necessárias duas leituras no bloco de índices. Em um arquivo extremamente grande, o acesso a um bloco perto do fim do arquivo demandará a leitura de todos os blocos de índices antes que o bloco de dados requerido possa finalmente ser lido. Assim, o desempenho da alocação indexada depende da estrutura do índice, do tamanho do arquivo e da posição do bloco desejado.

Alguns sistemas combinam a alocação contígua com a alocação indexada, usando a alocação contígua para arquivos pequenos (até três ou quatro blocos), e permutando automaticamente para uma alocação indexada se o arquivo ficar maior. Como a maior parte dos arquivos é pequena e a alocação contígua é eficiente para arquivos pequenos, o desempenho médio pode ser bastante adequado.

Em ocorrências assim, muitas outras otimizações estão em uso. Dada a disparidade entre a velocidade da CPU e a velocidade do disco, não é irracional adicionar milhares de instruções extras ao sistema operacional para economizar alguns movimentos do cabeçote do disco. Além disso, essa disparidade está aumentando com o tempo, ao ponto de centenas de milhares de instruções poderem ser racionalmente usadas para otimizar movimentos do cabeçote.

**Gerenciamento do espaço livre**

Como o espaço em disco é limitado, precisamos reutilizar para novos arquivos o espaço decorrente de arquivos excluídos, se possível. Discos óticos de gravação única permitem apenas uma gravação em qualquer setor determinado, logo, essa reutilização não é fisicamente possível. Para controlar o espaço livre em disco, o sistema mantém uma lista de espaços livres, que registra todos os blocos de disco livres, incluindo aqueles não alocados a algum arquivo ou diretório. Para criar um arquivo, pesquisamos a lista de espaços livres em busca do montante de espaço requerido, alocamos esse espaço ao novo arquivo e, então, ele é removido da lista de espaços livres. Quando um arquivo é excluído, seu espaço em disco é adicionado à lista de espaços livres – apesar de sua denominação, a lista de espaços livres pode não ser implementada como uma lista, conforme discutiremos a seguir.

### Vetor de bits

Frequentemente, a lista de espaços livres é implementada como um mapa de bits ou vetor de bits. Cada bloco é representado por 1 bit. Se o bloco está livre, o bit é 1; se o bloco está alocado, o bit é 0. Por exemplo, considere um disco em que os blocos 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 e 27 estejam livres e o resto dos blocos esteja alocado. O mapa de bits de espaços livres seria:

0011110011111100011000000011100000...

A principal vantagem dessa abordagem é sua relativa simplicidade e sua eficiência em encontrar o primeiro bloco livre ou *n* blocos livres consecutivos em disco. Na verdade, muitos computadores fornecem instruções de manipulação de bits que podem ser usadas efetivamente para essa finalidade. Uma técnica, para encontrar o primeiro bloco livre em um sistema que use um vetor de bits para alocar espaço em disco, é verificar sequencialmente cada palavra do mapa de bits, a fim de se verificar se esse valor não é 0, já que uma palavra com valor 0 contém somente bits 0 e representa um conjunto de blocos alocados. A primeira palavra diferente de 0 é varrida em busca do primeiro bit 1, que é a locação do primeiro bloco livre. O cálculo do número do bloco é (número de bits por palavra) × (número de palavras de valor 0) + deslocamento do primeiro bit 1.

Novamente, vemos recursos de hardware dirigindo funcionalidade de software. Infelizmente, os vetores de bits são ineficientes, a menos que o vetor inteiro seja mantido na memória principal (e seja gravado em disco ocasionalmente para fins de recuperação). É possível mantê-lo na memória principal para discos menores, mas não necessariamente para os maiores. Um disco de 1.3 GB com blocos de 512 bytes precisaria de um mapa de bits de mais de 332 kB para rastrear seus blocos livres, embora o agrupamento dos blocos em clusters de quatro reduza esse número para cerca de 83 kB por disco. Um disco de 1 TB com blocos de 4 kB requer 32 MB para armazenar seu mapa de bits, assim, dado que o tamanho dos discos aumenta constantemente, o problema dos vetores de bits também continuará a aumentar.

## 8.6 Segurança

Dizemos que um sistema é seguro quando seus recursos são usados e acessados como esperado sob todas as circunstâncias. Infelizmente, a segurança total não pode ser atingida, mas, mesmo assim, devemos possuir mecanismos que tornem as brechas de segurança uma ocorrência rara e não a normal.

Violações (ou má utilização) da segurança do sistema podem ser categorizadas como intencionais (maliciosas) ou acidentais. É mais fácil se proteger contra a má utilização acidental do que contra a maliciosa. Geralmente, os mecanismos de proteção são a base da proteção contra acidentes. Antes de adentrarmos o tópico, devemos observar que, em nossa discussão sobre segurança, usamos os termos **invasor** e **cracker** para quem tenta violar a segurança. Além disso, **ameaça** é a possibilidade de uma violação de segurança, tal como a descoberta de uma vulnerabilidade. Enquanto que **ataque** é a tentativa de violar a segurança.

A lista a seguir inclui vários tipos de violações acidentais e maliciosas de segurança:

- **Brecha de sigilo**: esse tipo de violação envolve a leitura não autorizada de dados (ou roubo de informações). Normalmente, uma brecha de sigilo é o objetivo de um invasor que captura dados secretos de um sistema ou fluxo de dados, tais como informações de cartões de crédito ou informações de credenciais para roubo de identidades, o que pode resultar diretamente em dinheiro para o intruso.

- **Brecha de integridade**: essa violação envolve a modificação não autorizada de dados. Esses ataques podem, por exemplo, resultar na transferência de responsabilidade para terceiros inocentes ou na modificação do código-fonte de uma aplicação comercial importante.

- **Brecha de disponibilidade**: essa violação envolve a destruição não autorizada de dados. Alguns crackers preferem provocar destruição e ganhar status ou se vangloriar de direitos, em vez de obter ganhos financeiros. A desfiguração de websites é um exemplo comum desse tipo de brecha de segurança.

- **Roubo de serviço**: essa violação envolve o uso não autorizado de recursos. Por exemplo, um invasor (ou programa invasor) pode instalar um daemon em um sistema que atue como servidor de arquivos.

- **Recusa de serviço**: essa violação envolve o impedimento do uso legítimo do sistema. Ataques de recusa de serviço, Denial-of-Service (DOS), são algumas vezes acidentais. O verme original da Internet transforma-se em um ataque DOS quando um bug não conseguiu retardar sua rápida disseminação.

Os agressores usam vários métodos-padrão em suas tentativas de violar a segurança. O mais comum é o mascaramento, em que um participante de uma comunicação finge ser alguém que não é (outro hospedeiro ou outra pessoa). Por meio do mascaramento, os agressores violam a autenticação ou a precisão da identidade. Eles podem obter acesso que, normalmente, não receberiam ou aumentar seus privilégios que, normalmente, não lhes seriam atribuídos.

Outro ataque comum é a reexecução de uma troca de dados capturada, que consiste na repetição maliciosa ou fraudulenta de uma transmissão de dados válida. Às vezes, a reexecução compõe o ataque inteiro, por exemplo, na repetição de uma solicitação para transferência de dinheiro, mas, frequentemente, ela é feita junto com a modificação de mensagens, novamente para aumentar privilégios. Considere o dano que poderia ser feito se uma solicitação de autenticação tivesse as informações de um usuário legítimo substituídas pelas de um usuário não autorizado. Outro tipo de ataque é o intermediário, em que um invasor se instala no fluxo de dados de uma comunicação, mascarando-se como o emissor para o receptor e vice-versa.

Em uma comunicação de rede, um ataque do intermediário pode ser precedido de um sequestro de sessão, em que uma sessão de comunicação ativa é interceptada (figura 109).

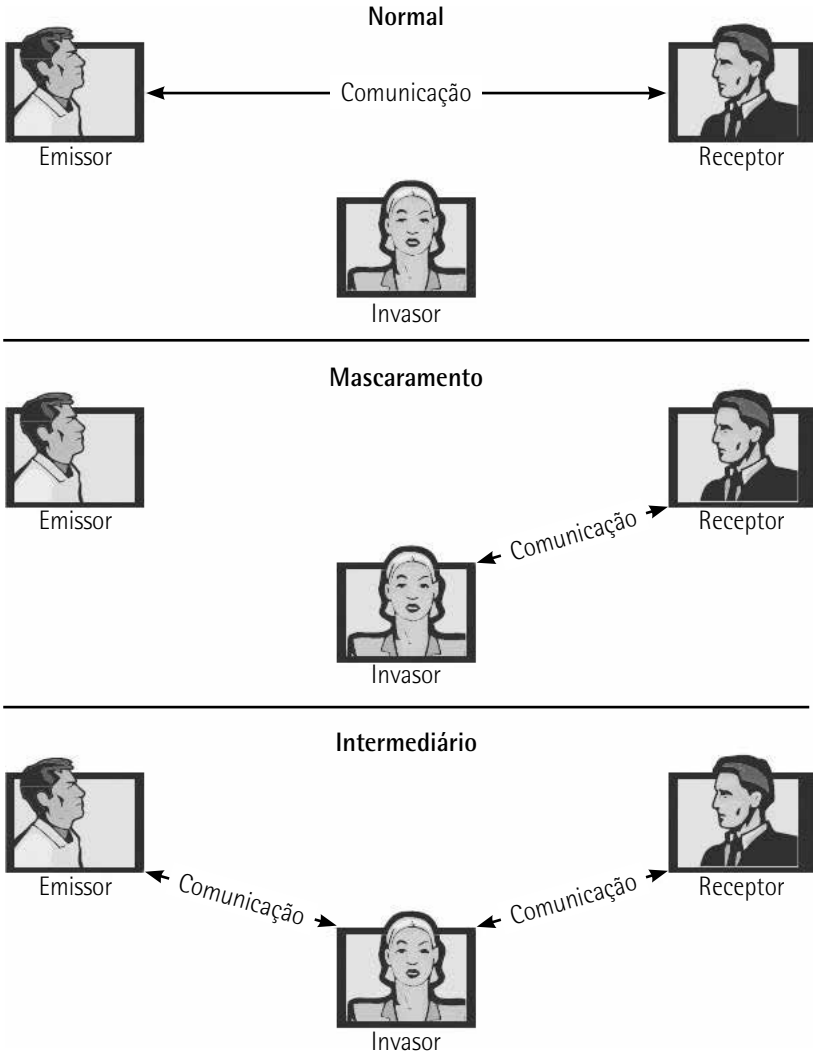



Figura 109 – Ataques-padrão à segurança

Fonte: Silberschartz (2015, p. 530).

Como já mencionado, não é possível proteger completamente o sistema de abusos maliciosos, mas o custo para o infrator é suficientemente alto para deter a maioria dos invasores. Em alguns casos, tal como em um ataque de recusa de serviço, é preferível impedir o ataque, mas detectá-lo, muitas vezes, já é o suficiente para que medidas defensivas possam ser tomadas. Para proteger um sistema, devemos tomar medidas de segurança em quatro níveis:

- **Físico**: os sítios que contêm os sistemas de computação devem ser fisicamente protegidos contra a entrada forçada ou furtiva de intrusos, assim como, tanto as salas das máquinas quanto os terminais ou estações de trabalho que têm acesso às máquinas devem ser protegidos.

- **Humano**: a autorização deve ser feita cuidadosamente para assegurar que apenas usuários apropriados tenham acesso ao sistema. Até mesmo usuários autorizados, no entanto, podem ser encorajados a deixar outras pessoas utilizarem seu acesso (mediante suborno, por exemplo), podendo, também, ser levados a permitir o acesso pela engenharia social. Um tipo de ataque de engenharia social é o phishing. Nesse caso, um e-mail ou página da web de aparência legítima engana um usuário, levando-o a inserir informações confidenciais. Outra técnica é o dumpster diving, um termo geral para a tentativa de coletar informações de modo a obter acesso não autorizado ao computador (examinando o conteúdo de lixeiras, bisbilhotando agendas telefônicas ou examinando lembretes contendo senhas, por exemplo). Esses problemas de segurança são de esfera pessoal e de gerenciamento, e não problemas relacionados aos sistemas operacionais.

- **Sistema operacional**: o sistema deve proteger a si próprio contra brechas de segurança acidentais ou propositais. Um processo fora de controle poderia constituir um ataque acidental de recusa de serviço; uma consulta a um serviço poderia revelar senhas; um estouro de pilha poderia permitir o acionamento de um processo não autorizado etc. A lista de brechas possíveis é quase infinita.

- **Rede**: muitos dados nos sistemas modernos viajam por linhas privadas dedicadas, linhas compartilhadas como a Internet, conexões sem fio ou linhas dial-up.

A interceptação desses dados poderia ser tão danosa quanto uma invasão em um computador e a interrupção de comunicações poderia constituir um ataque remoto de recusa de serviço, diminuindo o uso do sistema e a confiança dos usuários.

A segurança nos dois primeiros níveis deve ser mantida para que a segurança do sistema operacional seja assegurada. Uma vulnerabilidade em um nível alto de segurança (físico ou humano) permite que medidas de segurança estritamente de baixo nível (sistema operacional) sejam burladas. Portanto, o antigo provérbio de que uma corrente é tão forte quanto seu elo mais fraco, é particularmente verdadeiro quando se trata da segurança de sistemas.

Além disso, o sistema deve fornecer proteção para permitir a implementação de recursos de segurança. Sem a capacidade de autorizar usuários e processos, controlar seu acesso e registrar suas atividades, seria impossível para um sistema operacional implementar medidas de segurança ou ser executado de forma segura. Recursos de proteção de hardware são necessários para suportar um esquema geral de proteção. Por exemplo, um sistema sem proteção de memória não pode ser seguro.

Desse modo, novos recursos de hardware estão permitindo que os sistemas sejam mais seguros. No entanto, infelizmente, pouca coisa em segurança é direta e, à medida que invasores exploram vulnerabilidades de segurança, medidas defensivas são criadas e implantadas, fazendo com que os invasores se tornem mais sofisticados em seus ataques. Em incidentes recentes de segurança, detectou-se o uso de spyware para fornecer um canal de introdução de spam por intermédio de sistemas inocentes. Assim, esse jogo de gato e rato deve continuar, onde serão necessárias mais ferramentas de segurança para bloquear o número cada vez maior de técnicas e atividades de invasores.

### Ameaças de programas

Os processos, junto com o kernel, são o único meio de execução de tarefas em um computador. Logo, escrever um programa que crie uma brecha de segurança, ou faça um processo normal mudar seu comportamento e criar uma brecha, é um objetivo comum dos crackers. Na verdade, até mesmo a maioria dos eventos de segurança não relacionados com programas têm como objetivo causar uma ameaça de programa. Por exemplo, embora seja útil fazer login em um sistema sem autorização, é muito mais útil deixar para trás um daemon de porta dos fundos que forneça informações ou permita o acesso fácil, mesmo se o ataque original for bloqueado.

**Saiba mais**

Para conhecer um pouco mais sobre bomba lógica, leia no capítulo 15 do livro a seguir:

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Fundamentos de sistemas operacionais*. 9. ed. Rio de Janeiro: LTC, 2015.

## Resumo

Na unidade IV, exploramos os conceitos fundamentais de memória virtual e swapping, desvendando como os sistemas operacionais gerenciam a memória física e virtual de um computador. Abordamos desde a alocação de memória, passando pelas técnicas de paginação e segmentação, até o impacto da memória virtual no desempenho do sistema.

Explicamos como o sistema operacional desempenha um papel crucial no gerenciamento da memória, garantindo que os recursos sejam alocados de forma eficiente e os programas em execução tenham acesso aos dados necessários. O sistema operacional utiliza algoritmos complexos para gerenciar a memória, como o algoritmo First In, First Out (FIFO), Last In, First Out (LIFO) e Least Recently Used (LRU). Esses algoritmos determinam qual página da memória deve ser transferida para o disco rígido quando o espaço na RAM estiver limitado, otimizando o uso da memória física e maximizando o desempenho do sistema. O sistema operacional também é responsável por proteger a memória, impedindo que os programas acessem áreas de memória que não lhes foram atribuídas, garantindo a integridade e segurança dos dados.

Vimos, ainda, como funciona os sistemas de arquivos, abordando a estrutura, métodos de acesso, compartilhamento e gerenciamento de espaço, além de práticas de segurança.

## Exercícios

**Questão 1**. Os sistemas modernos costumam adotar esquemas de memória virtual, o que permite que o espaço de endereçamento lógico dos processos seja bem maior do que a memória física. Além disso, a paginação por demanda, em que páginas são carregadas apenas quando realmente são necessárias, amplia a eficiência no uso da memória e reduz o tempo de carregamento. Entretanto, a adoção desses mecanismos depende de um suporte adequado de hardware, como bits de válido-inválido em cada entrada de página, e a capacidade de gerar erros de página ao acessar áreas não presentes em memória.

A depender do cenário de execução, um processo pode, em determinado momento, apresentar um número excessivo de falhas de página (chamado de page thrashing), o que prejudica drasticamente o desempenho do sistema. Nesse contexto, considere as particularidades da paginação por demanda e assinale a alternativa que melhor representa como essa técnica contribui para a execução de programas maiores do que a memória física, ao mesmo tempo em que reconhece os riscos e os desafios de implementação.

A) A paginação por demanda implica, necessariamente, a carga de todas as páginas do processo antes da sua execução. No instante em que o processo inicia, o sistema garante que cada página esteja devidamente carregada no espaço físico, o que evita erros de página durante a execução. Dessa forma, o benefício de não precisar de alocação contígua é mantido, mas não há economia no tempo de carregamento, pois é preciso trazer cada bloco. Esse modelo é o mais comum nos sistemas operacionais modernos e permite que as páginas adicionais sejam somente trocadas se a memória estiver cheia.

B) Graças à paginação por demanda, o processo pode começar a ser executado mesmo que apenas algumas de suas páginas estejam carregadas na memória. As páginas são trazidas sob demanda quando o processo tenta acessá-las. Essa abordagem economiza I/O e espaço físico, pois somente as páginas realmente usadas são carregadas. Porém, caso o processo precise subitamente de muitas páginas em sequência, o sistema pode sofrer sucessivas falhas de página, o que leva a um comportamento chamado de page thrashing, em que o tempo gasto em trocas de página supera o tempo efetivo de CPU. Ainda assim, o suporte de hardware é imprescindível, como o bit válido-inválido e a lógica que dispara interrupções ao acessar páginas não carregadas.

C) Em um esquema de paginação por demanda, não existe a possibilidade de erros de página. Se um processo referenciar uma página ainda não trazida para a memória, essa página é simplesmente ignorada, e o valor 0 é retornado como se fosse o conteúdo da célula de memória acessada. Isso garante execução rápida, pois não há interrupções. No entanto, cada processo deve prever quais páginas não estão em memória e ajustar sua lógica para tratar de forma correta os dados que não existem no array físico.

D) A paginação por demanda é incompatível com a memória virtual. Isso ocorre porque, na definição clássica de memória virtual, todos os blocos do programa (ou segmentos, dependendo da arquitetura) precisam estar simultaneamente em memória para que as instruções sejam executadas sem erros. Logo, se um processo tentar acessar uma instrução em disco, essa operação não será interceptada pelo sistema operacional, mas sim pelo hardware, que imediatamente encerra o processo por violação de acesso. Dessa forma, cada processo deve caber inteiramente na memória principal antes de iniciar sua execução.

E) Embora a paginação por demanda dispense a alocação integral do programa na memória, ela obriga cada página a ser carregada sequencialmente em ordem de acesso, sem permitir acessos simultâneos a páginas contíguas. Assim, é inviável acessar páginas vizinhas paralelamente ou compartilhar páginas entre processos. Também não existe a possibilidade de mapeamento de bibliotecas em áreas de memória compartilhada, pois cada processo permanece isolado nos seus frames específicos, e cada página só pode ser carregada por um processo de cada vez.

Resposta correta: alternativa B.

**Análise da questão**

A alternativa B descreve com precisão o princípio da paginação por demanda: carregar as páginas apenas quando o processo de fato as acessa, o que evita o overhead inicial e reduz a necessidade de espaço físico. Entretanto, ela também aponta o risco de page thrashing — quando muitas falhas de página ocorrem em sequência, o que impacta o desempenho. Esse modelo depende de um hardware que diferencie páginas na memória das que estão em disco, via bit válido-inválido e interrupções de erro de página, coerentemente com a teoria de memória virtual.

**Questão 2**. A estrutura de diretórios de um sistema de arquivos serve como mapa para localizar os arquivos no disco e registrar dados como nome, localização física e atributos (tamanho, datas, permissões etc.). Várias estratégias de organização de diretórios podem ser adotadas, desde o modelo de nível único, em que todos os arquivos residem em um único local, até o modelo de árvore de múltiplos níveis, o que permite subdiretórios recursivos. Além disso, o sistema precisa lidar com compartilhamento entre usuários, o que envolve definir permissões de acesso e associar cada arquivo a um dono e/ou grupo.

Considere, então, um ambiente multiusuário em que cada usuário tem seu "diretório pessoal" e pode criar subdiretórios livremente. A partir disso, assinale a alternativa que descreve corretamente como esse sistema gerencia a localização e o acesso aos arquivos em múltiplos níveis de diretórios.

A) Em uma estrutura de nível único, basta o sistema de arquivos manter um único mapeamento (nome do arquivo → localização física em disco). Assim, cada usuário precisa dar nomes exclusivos aos seus arquivos, pois não há subdiretórios nem possibilidade de caminhos diferenciados. Para resolver isso em um ambiente multiusuário, é comum o sistema renomear automaticamente os arquivos para assegurar unicidade, adicionando sufixos como "_Usuario".

B) No modelo de dois níveis, existe um diretório global (MFD) indexado pelo nome de cada usuário, que aponta para seu User File Directory (UFD) particular. Cada UFD pode armazenar uma lista de arquivos sem interferir nos nomes escolhidos por outros usuários, pois cada subdiretório pessoal atua como um espaço isolado de nomes. No entanto, os usuários não podem criar outros subdiretórios dentro do seu UFD. Essa solução resolve o problema de nomes de arquivos colidindo entre usuários, mas não oferece a granularidade de organização que subdiretórios recursivos permitem.

C) Em uma hierarquia de diretórios em árvore (vários níveis), cada diretório pode conter arquivos ou subdiretórios; portanto, o sistema de arquivos "caminha" (path) da raiz (MFD) até chegar ao subdiretório ou arquivo desejado. Com isso, mesmo que dois usuários criem arquivos com o mesmo nome, não há conflito, pois cada arquivo está em um caminho distinto. Para compartilhamento, o dono pode ajustar permissões, definindo quem pode ler ou gravar, e o kernel verifica esses atributos ao abrir ou modificar o arquivo.

D) Em arquiteturas de árvore de múltiplos níveis, o sistema não necessita de nenhuma estrutura de diretório adicional além da raiz. Por convenção, os nomes completos (caminhos) são ignorados quando um programa acessa um arquivo, pois o sistema simplesmente faz uma busca linear em todos os diretórios até encontrar uma correspondência. Apesar de intuitivo, esse esquema inviabiliza a proteção por permissões, uma vez que não há registro de qual usuário tem qual arquivo. Desse modo, basta que um arquivo seja encontrado por busca para que ele se torne acessível.

E) Normalmente, em sistemas que suportam multiusuário, cada nome de arquivo é convertido internamente em identificadores globais únicos (GUID) pelo sistema de arquivos. Esses GUIDs são iguais em todo o disco, mas não são ligados a nenhum diretório. Por isso, quando se abre um arquivo, o usuário informa apenas o GUID; não há conceito de "caminho" ou "subdiretório", pois a localização em disco é definida no momento da formatação e permanece fixa, independentemente de reorganizações lógicas.

Resposta correta: alternativa C.

**Análise da questão**

A descrição de uma árvore de múltiplos níveis – cada diretório podendo conter arquivos ou subdiretórios – corresponde ao modelo prevalecente em sistemas como Unix, Windows e outros. Cada arquivo é, então, endereçado por um path a partir da raiz, podendo existir nomes iguais, contanto que se encontrem em caminhos diferentes. O compartilhamento e a proteção são resolvidos por permissões (ou ACLs) associadas ao arquivo que o kernel verifica ao abrir ou manipular.

As outras alternativas descrevem estruturas menos flexíveis (A e B); omitem a existência de permissões ou paths (D); e inventam um esquema de GUID que ignora completamente a organização de diretórios (E).

## REFERÊNCIAS


AFIFI-SABET, K. What is cache memory? *ITPro*, 26 set. 2023. Disponível em: https://tinyurl.com/3ph2kmdm. Acesso em: 19 out. 2024.

ALECRIM, E. O que é HD (disco rígido): características e como funciona. *Infowester*, 23 ago. 2020. Disponível em: https://tinyurl.com/2pnn4zkk. Acesso em: 10 ago. 2024.

ALECRIM, E. O que é memória RAM? Conceito, características e tipos. *Infowester*, 20 set. 2024. Disponível em: https://tinyurl.com/3rfa5zye. Acesso em: 11 mar. 2025.

ARTHUR, C. Apple faces its 'Nike moment' as ABC Nightline goes inside Foxconn. *The Guardian*, 20 fev. 2012. Disponível em: https://tinyurl.com/yhkb5sj2. Acesso em: 17 out. 2024.

ASTH, R. C. Sistema de numeração decimal. *Toda Matéria*, [s.d.]. Disponível em: https://tinyurl.com/4ccwn4rj. Acesso em: 7 set. 2024.

BAPTISTA, J. Evolução dos micro processadores. *Timetoast*, [s.d.]. Disponível em: https://tinyurl.com/nhefjpp3. Acesso em: 14 set. 2024.

BHUNJE, S. USB. *Theegeek*, 18 jun. 2013. Disponível em: https://tinyurl.com/3cxd56te. Acesso em: 10 out. 2024.

BLAKEMORE, E. The U.S. nuclear program still uses eight-inch floppy disks. *Smithsonian Magazine*, 26 maio 2016. Disponível em: https://tinyurl.com/2aufra3c. Acesso em: 11 out. 2024.

BOLTON, D. Definition of ROM. *ThoughtCo.*, 1º jun. 2017. Disponível em: https://tinyurl.com/2zkskbpa. Acesso em: 20 out. 2024.

CARTER, N. *Arquitetura de computadores*. Porto Alegre: Bookman, 2002.

CLEMENTS, A. *Computer organization and architecture*: themes and variations. Stamford: Cengage, 2014.

COLEMAN, D. Fastest SD cards: real speed test results/2025. *Have Camera Will Travel*, 16 jan. 2025. Disponível em: https://tinyurl.com/msjs7dku. Acesso em: 11 mar. 2025.

DELGADO, J.; RIBEIRO, C. *Arquitetura de computadores*. Rio de Janeiro: LTC, 2014.

FISHER, T. What is an IDE cable? *Lifewire*, 22 jun. 2023. Disponível em: https://tinyurl.com/362fdk9c. Acesso em: 27 set. 2024.

GLOSSARY: types of CD, DVD and Blu-ray. *Disc Wizards*, [s.d.]. Disponível em: https://tinyurl.com/5n8pvxwb. Acesso em: 22 out. 2024.

HISTORY of computers: the Z1: the first programmable computer. *Sutori*, [s.d.]a. Disponível em: https://tinyurl.com/4cxzsuuy. Acesso em: 22 set. 2024.

HISTORY of silicon wafers used in the semiconductor industry. *Chips etc.*, [s.d.]b. Disponível em: https://tinyurl.com/kchpckr5. Acesso em: 10 out. 2024.

IDENTIFYING expansion slots: ISA vs. PCI vs. AGP. *Computers by Campus*, [s.d.]. Disponível em: https://tinyurl.com/2dvc7bva. Acesso em: 22 set. 2024.

IDOETA, I. V.; CAPUANO, F. G. *Elementos de eletrônica digital*. 40. ed. São Paulo: Érica, 2008.

INTEL 8008. *Chipdb*, [s.d.]. Disponível em: https://tinyurl.com/5fj9mk44. Acesso em: 10 out. 2024.

INTEL 80486 microprocessor family. *CPU-World*, 23 jan. 2023a. Disponível em: https://tinyurl.com/3dddecwh. Acesso em: 16 out. 2024.

INTEL 8080 family. *CPU-World*, 24 mar. 2023b. Disponível em: https://tinyurl.com/53hd3848. Acesso em: 14 out. 2024.

INTEL Desktop Pentium microprocessor family. *CPU-World*, 1º jul. 2024. Disponível em: https://tinyurl.com/4yva22x8. Acesso em: 17 out. 2024.

A INTEL lança processadores Intel Core de 14ª geração para desktop. *Intel Newsroom*, 16 out. 2023. Disponível em: https://tinyurl.com/3xjm8um7. Acesso em: 11 mar. 2025.

JANSEN, D. CPUs: Intel 80286. *Low End Mac*, 6 out. 2014. Disponível em: https://tinyurl.com/mr3sh5nz. Acesso em: 14 out. 2024.

JOWITT, T. Tales in tech history: the floppy disk. *Silicon*, 7 abr. 2017. Disponível em: https://tinyurl.com/3b5xu3kb. Acesso em: 14 ago. 2024.

KARASINSKI, V. Placas de vídeo: a versão e a banda do PCIe interferem no desempenho? *Tecmundo*, 25 fev. 2014. Disponível em: https://tinyurl.com/3mv4xsed. Acesso em: 22 set. 2024.

KIDS ENCYCLOPEDIA FACTS. Vacuum tube facts for kids. *Kiddle*, 11 jul. 2024. Disponível em: https://tinyurl.com/2yfuzmun. Acesso em: 22 set. 2024.

KILBURN, T. *et al.* The Manchester University Atlas Operating System. Part I: Internal Organization. *Computer Journal*, v. 4, n. 3, out. 1961.

LANTZ, M. Why the future of data storage is (still) magnetic tape. *IEEE Spectrum*, 28 ago. 2018. Disponível em: https://tinyurl.com/3vc8r733. Acesso em: 11 set. 2024.

MACHADO, F. B.; MAIA, L. P. *Arquitetura de sistemas operacionais*. 5. ed. Rio de Janeiro: LTC, 2013.

MEMORY. *Imperial College London*, 2 nov. 2004. Disponível em: https://tinyurl.com/ymn78t7t. Acesso em: 10 set. 2024.

MICROPROCESSEUR Intel 4004. *Histoire de l'Informatique*, [s.d.]. Disponível em: https://tinyurl.com/58mpubrj. Acesso em: 10 out. 2024.

MILLS, M. Almost all Sata cables are flat, why does it have this shape? *ITIGIC*, 5 ago. 2020. Disponível em: https://tinyurl.com/bd2cf3xp. Acesso em: 25 set. 2024.

MONTEIRO, M. A. *Introdução à organização de computadores*. Rio de Janeiro: LTC, 2019.

NULL, L.; LOBUR, J. *Princípios básicos de arquitetura e organização de computadores*. Porto Alegre: Bookman, 2010.

PANNAIN, R.; BEHRENS, F. H.; PIVA JUNIOR, D. *Organização básica de computadores e linguagem de montagem*. São Paulo: Elsevier, 2012.

PURE silicon stock photos, images & pictures. *Dreamstime*, [s.d.]. Disponível em: https://tinyurl.com/2d5kp5se. Acesso em: 15 out. 2024.

REPLICA of the first transistor. *Science Museum Group*, [s.d.]. Disponível em: https://tinyurl.com/zjpye2x2. Acesso em: 22 set. 2024.

ROTMAN, D. We're not prepared for the end of Moore's Law. *MIT Technology Review*, 24 fev. 2020. Disponível em: https://tinyurl.com/4e2pu2zd. Acesso em: 12 dez. 2024.

SHILOV, A. Intel's Core i7-4790K 'Devil's Canyon' may not be available at launch. *KitGuru*, 27 maio 2014. Disponível em: https://tinyurl.com/4fzpx7vp. Acesso em: 15 out. 2024.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Fundamentos de sistemas operacionais*. 9. ed. Rio de Janeiro: LTC, 2015.

SILICON powder (Si powder). *Edgetech Industries LLC.*, [s.d.]. Disponível em: https://tinyurl.com/53ahhnv8. Acesso em: 10 out. 2024.

STALLINGS, W. *Arquitetura e organização de computadores*. São Paulo: Pearson, 2010.

TANENBAUM, A. S.; AUSTIN, T. *Organização estruturada de computadores*. São Paulo: Pearson, 2013.

TANENBAUM, A. S.; AUSTIN, T. *Sistemas operacionais modernos*. 5. ed. São Paulo: Pearson, 2024.

THE FIRST disk drive: RAMAC 350. *Computer History Museum*, [s.d.]. Disponível em: https://tinyurl.com/yf6mutcz. Acesso em: 10 ago. 2024.

TOCCI, R. J.; WIDMER, N. S.; MOSS, G. L. *Sistemas digitais*: princípios e aplicações. São Paulo: Pearson, 2011.

TRIOLET, D. *et al.* Intel Core i7 et Core i5 LGA 1155 Sandy Bridge. *Hardware.fr*, 3 jan. 2011. Disponível em: https://tinyurl.com/bdcnz4er. Acesso em: 10 de out. 2024.

UNDERSTANDING flash memory and how it works. *Electronics-Lab*, 5 fev. 2018. Disponível em: https://tinyurl.com/mv9mm69k. Acesso em: 20 out. 2024.

WILLIAMS, R. How do I type accents on my computer keyboard? *The iPaper*, 1º mar. 2019. Disponível em: https://tinyurl.com/d27xwt6j. Acesso em: 12 out. 2024.


Informações:
www.sepi.unip.br ou 0800 010 9000