Universidade Federal do Amazonas
ICET - INSTITUTO DE CIÊNCIAS EXATAS E TECNOLOGIA
Sistemas de Informação

Felipe Rangel, Nadia Leão, Oliviê Kalil, Marcos Gabriel

Análise de Implementações do Gerenciamento de Memória

Trabalho Prático

Itacoatiara – AM, 2025

Felipe Rangel, Nadia Leão, Oliviê Kalil, Marcos Gabriel

Análise de Implementações do Gerenciamento de Memória

Trabalho Prático apresentado à disciplina de Sistemas Operacionais do Curso de Sistemas de Informação do Instituto de Ciências Exatas e Tecnologia (ICET) da Universidade Federal do Amazonas, como requisito parcial para obtenção de nota.

Universidade Federal do Amazonas

Orientador: Prof. Dr. Rallyson dos Santos Ferreira

Itacoatiara – AM
2025

Sumário

1 Introdução

Este documento apresenta uma análise detalhada das implementações de gerenciamento de memória em sistemas operacionais modernos, abordando tanto a evolução histórica das técnicas quanto as estratégias específicas adotadas por diferentes sistemas operacionais.

O gerenciamento de memória é um dos componentes fundamentais de um sistema operacional, responsável por alocar, gerenciar e liberar memória para processos em execução. A evolução dessas técnicas ao longo das décadas reflete os avanços tecnológicos em hardware e as crescentes demandas por eficiência e desempenho.

O gerenciamento de memória é uma das funções mais críticas de um sistema operacional, responsável por alocar e gerenciar eficientemente a memória principal (RAM) entre múltiplos processos.

2 Conceitos Fundamentais

Para superar as limitações da alocação contígua, notavelmente a fragmentação externa, foram desenvolvidas técnicas de alocação não contígua.

2.1 Hierarquia de Memória

De acordo com Garcez (2023), organiza as memórias existentes em um computador na forma de uma pirâmide. No topo da pirâmide, temos as memórias mais rápidas e mais caras, que consequentemente serão utilizadas em menos quantidade. Já na base da pirâmide temos as memórias mais lentas e mais mais baratas, que costumam ser vistas em maior frequência nos sistemas computacionais.

No topo da pirâmide de memória estão os registradores, que ficam dentro da CPU e possuem altíssima velocidade, mas são extremamente limitados em quantidade. Logo abaixo, encontra-se a memória Cache, responsável por armazenar dados frequentemente utilizados pelo processador, permitindo acessos mais rápidos. Essa memória costuma estar fisicamente próxima da CPU, e em alguns casos, até integrada a ela (GARCEZ, 2023).

Em seguida vem a memória primária, representada pela memória RAM, onde os dados são armazenados temporariamente enquanto o computador está em uso. Logo abaixo na hierarquia está a memória secundária, composta por dispositivos como HDs, SSDs e outros tipos de armazenamento persistente, usados para guardar informações de forma permanente (GARCEZ, 2023).

O design bem-sucedido de uma hierarquia de memórias não apenas melhora o desempenho computacional, mas também influencia a eficiência energética e o custo de produção dos sistemas. O balanceamento entre velocidade e custo é essencial para atender às demandas contemporâneas de aplicações cada vez mais complexas e intensivas

em dados. Além disso, o avanço na tecnologia de memórias, como memórias de acesso aleatório não volátil (NVRAM), traz novos desafios e oportunidades para a organização da hierarquia de memórias (Jacob apud INSTITUTO FEDERAL DE MINAS GERAIS , 2002).

2.2 Alocação Contígua

De acordo com Vasconcellos (2011), Foi implementada nos primeiros sistemas operacionais, dividindo a memória principal em duas partes, uma para o sistema operacional, e outra para o programa do usuário. Não é possui a permissão da utilização eficiente dos recursos do sistema, devido que, um usuário pode dispor destes recursos, e os programas são limitados ao tamanho de memória principal disponível para o usuário.

Nesse método cada arquivo vai ocupar um conjunto de bloco contígos no disco. Desta forma o número de operações acessados nos arquivos alocados contiguamente é mínimo. Sendo a melhor escolha quando; Arquivos não são deletados do disco; O tamanho dos arquivos é fixos (MATTIELLO; SANTOS, 2011).

Para a realizar a alocação de um arquivo, é necessário que o arquivo tenha determinado número de ocupação de bloco, que vá até n blocos:

- Primeiro bloco = 0
- Segundo bloco = b + 1
- Terceiro bloco = b + 2
- até o último = b + (n-1)

Exemplo: O arquivo precisa de 4 blocos, e começa no bloco 10, ele ocuparar os blocos: 10,11,12 e 13.

Segundo UFRGS (

$$s.d$$

), a alocação contígua é um método de armazenamento em que todos os blocos de um arquivo são colocados em sequência no disco, o que permite um acesso mais rápido aos dados. Essa técnica apresenta vantagens tanto no acesso sequencial, já que o sistema pode ler blocos consecutivos sem precisar reposicionar o cabeçote, quanto no acesso direto (randômico), pois qualquer bloco pode ser localizado facilmente pela fórmula (b + i), sendo b o bloco inicial do arquivo.

De acordo com UFRGS (

$$s.d$$

), esse método apresenta problemas significativos. Um dos principais é sua dificuldade em encontrar espaço contíguo suficiente para armazenar novos arquivos, o que leva à necessidade de estratégias como first-fit, best-fit, worst-fit e circular-fit. Além disso, ocorre

a fragmentação externa, que surge quando o espaço livre do disco se divide em pequenas partes não contíguas, exigindo, em muitos casos, a desfragmentação do sistema. Outro desafio está em determinar o espaço necessário para um arquivo no momento da criação, já que o crescimento posterior pode gerar falhas ou desperdício de memória. Para minimizar esse problema, podem ser adotadas soluções como a realocação do arquivo, o superdimensionamento do espaço inicial ou o uso de extensões, que permitem armazenar o arquivo em várias porções contíguas separadas.

2.3 Alocação Não Contígua

A alocação de memória é crucial no domínio dos sistemas operacionais para garantir o uso ideal dos recursos de memória de um computador. "Alocação de Memória Não Contígua"é uma técnica incomum para alocação de memória. Ao contrário da alocação contígua típica, que coloca os processos em blocos consecutivos, essa abordagem permite que os processos sejam armazenados em regiões de memória dispersas ou não adjacentes (GAURAV, 2024).

Segundo Lithmee (2018), embora a alocação de memória não contígua reduza o desperdício de memória, ela apresenta algumas desvantagens. Ela causa sobrecarga devido à tradução de endereços. Além disso, como as diferentes seções do mesmo processo residem em locais de memória diferentes, ela pode minimizar a velocidade geral de execução.

De acordo com UFRGS (

$$s.d$$

), a alocação de arquivos em disco elimina a fragmentação externa e permite que os arquivos cresçam indefinidamente, sendo limitada apenas pela quantidade de blocos livres disponíveis. O desempenho de acesso varia conforme o tipo: no acesso sequencial há custo de reposicionamento do cabeçote, enquanto no acesso direto é necessário ler todos os blocos anteriores até o desejado. Além disso, misturar dados com informações de controle dentro dos blocos exige espaço adicional para ponteiros, o que pode ser otimizado com o uso de clusters. A largura do ponteiro, definida em bits, também limita o tamanho máximo dos arquivos. Em termos de confiabilidade, erros podem causar leitura ou escrita em blocos incorretos, pertencentes a outros arquivos ou a áreas livres do disco.

3 Técnicas de Gerenciamento de Memória

As duas técnicas mais importantes de alocação não contígua são a Paginação e a Segmentação.

3.1 Paginação (Paging)

A paginação é a técnica de gerenciamento de memória dominante na maioria dos sistemas operacionais modernos (como Windows, Linux e macOS). Sua abordagem resolve o problema da fragmentação externa ao dividir a memória em blocos de tamanho fixo.

3.1.1 Conceito Central

A ideia fundamental da paginação é que a memória física e a memória lógica de um processo são divididas em blocos de tamanho igual e fixo.

- **Páginas (Pages):** São os blocos de tamanho fixo que compõem o espaço de endereço lógico de um processo.
- **Quadros (Frames) ou Molduras:** São os blocos de tamanho fixo correspondentes na memória física (RAM).

Quando um processo é carregado, suas páginas podem ser alocadas em quaisquer quadros de memória física que estejam livres. Esses quadros não precisam ser contíguos.

3.1.2 Mecanismo: A Tabela de Páginas (Page Table)

Para que o sistema saiba onde cada página de um processo está localizada na memória física, o sistema operacional mantém uma estrutura de dados chamada Tabela de Páginas (Page Table) para cada processo.

3.1.2.1 Função

A Tabela de Páginas mapeia o endereço de cada página lógica para o endereço do quadro físico onde ela está armazenada.

3.1.2.2 Tradução de Endereço

A CPU (através da Unidade de Gerenciamento de Memória - MMU) traduz automaticamente o endereço lógico (visto pelo processo) em um endereço físico (na RAM). O endereço lógico é dividido em duas partes:

- **Número da Página (p):** Usado como um índice para acessar a Tabela de Páginas.
- **Deslocamento (d) (Offset):** A posição exata do dado dentro daquela página.

A MMU consulta a Tabela de Páginas usando 'p' para encontrar o número do quadro 'f' correspondente. O endereço físico real é calculado combinando o número do quadro com o deslocamento: $EndereçoFísico = (f \times tamanho\_da\_pagina) + d$.

3.1.3 Vantagens

- **Elimina a Fragmentação Externa:** Como todos os blocos (páginas e quadros) têm o mesmo tamanho, qualquer quadro livre pode ser usado por qualquer página. Não há "buracos"de memória inutilizáveis entre os processos.
- **Flexibilidade:** Permite que um processo tenha um espaço de endereço lógico muito maior que a memória física disponível (base da Memória Virtual).
- **Compartilhamento:** Páginas de código (como bibliotecas compartilhadas) podem ser mapeadas para as tabelas de páginas de múltiplos processos, economizando memória.

3.1.4 Desvantagens

- **Fragmentação Interna:** Como um processo raramente tem um tamanho que é um múltiplo exato do tamanho da página, a última página alocada quase sempre terá algum espaço desperdiçado.
- **Overhead de Memória:** A própria Tabela de Páginas consome memória. Em sistemas de 64 bits, essa tabela pode se tornar gigantesca, exigindo técnicas como paginação de múltiplos níveis.
- **Overhead de Desempenho:** Teoricamente, cada acesso à memória pelo programa exigiria dois acessos: um para consultar a Tabela de Páginas e outro para acessar o dado em si. Esse problema é mitigado pelo uso de um cache de hardware especializado chamado TLB (Translation Lookaside Buffer).

3.2 Segmentação (Segmentation)

A segmentação adota uma abordagem fundamentalmente diferente. Em vez de focar na divisão física da memória (como a paginação), ela foca na divisão lógica do programa, conforme a visão do programador.

3.2.1 Conceito Central

A memória é dividida em blocos de tamanho variável, chamados Segmentos. Cada segmento corresponde a uma unidade lógica do programa. Exemplos de Segmentos:

- Um segmento para o código (instruções do programa).
- Um segmento para os dados globais.
- Um segmento para a pilha (stack).

- Um segmento para a heap (dados dinâmicos).

O programador (ou compilador) vê o programa como uma coleção desses segmentos, e não como um único vetor linear de endereços.

3.2.2 Mecanismo: A Tabela de Segmentos (Segment Table)

Assim como na paginação, o sistema operacional mantém uma Tabela de Segmentos para cada processo, que armazena as informações sobre cada segmento.

3.2.2.1 Função

Cada entrada na tabela de segmentos especifica duas informações cruciais sobre um segmento:

- **Base:** O endereço físico inicial onde o segmento está alocado na RAM.
- **Limite (Limit):** O tamanho (comprimento) daquele segmento.

3.2.2.2 Tradução de Endereço

O endereço lógico gerado pela CPU é um par: (número_do_segmento, deslocamento).

- O sistema usa o número_do_segmento (s) como índice na Tabela de Segmentos para encontrar a Base e o Limite.
- **Verificação de Proteção:** O sistema verifica se o deslocamento (d) é menor que o Limite.
- Se $d \geq Limite$, o processo está tentando acessar memória fora do seu segmento, gerando uma falha de proteção (o famoso "Segmentation Fault").
- Se a verificação for aprovada, o endereço físico é calculado: $EndereçoFísico = Base + d$.

3.2.3 Vantagens

- **Proteção e Compartilhamento:** A segmentação é conceitualmente excelente para proteção. É muito fácil, por exemplo, marcar o segmento de código como "somente leitura"(R-X) e o segmento de dados como "leitura/escrita"(RW-). O compartilhamento de um segmento de código entre processos também é simples.
- **Visão Lógica:** Reflete a estrutura modular de um programa, facilitando o gerenciamento de estruturas de dados que crescem ou diminuem (como a pilha).

- **Sem Fragmentação Interna:** Os segmentos são alocados com seu tamanho exato, eliminando o desperdício dentro de um bloco alocado.

3.2.4 Desvantagens

- **Fragmentação Externa:** Este é o principal problema da segmentação. Como os segmentos têm tamanhos variáveis, à medida que são carregados e descarregados da memória, "buracos"(espaços livres) de diferentes tamanhos aparecem. Com o tempo, a memória fica fragmentada, podendo não haver um bloco contíguo grande o suficiente para um novo segmento, mesmo que a soma dos buracos seja suficiente.
- **Gerenciamento Complexo:** A alocação de memória torna-se um problema complexo. O SO precisa usar algoritmos (como First-fit ou Best-fit) para encontrar um buraco adequado para um novo segmento.

4 Memória Virtual

4.1 Conceito de Memória Virtual

A memória virtual é uma técnica de gerenciamento de memória utilizada pelos sistemas operacionais para simular uma capacidade de memória maior do que a fisicamente disponível na máquina. Essa técnica permite que o sistema operacional utilize parte do armazenamento secundário, como o disco rígido ou SSD, como uma extensão da memória principal (RAM), facilitando a execução de múltiplos processos simultaneamente, mesmo quando a RAM é insuficiente.

Com a memória virtual, os processos têm a ilusão de estarem inteiramente carregados na memória física, quando, na verdade, apenas partes deles estão presentes na RAM em determinado momento. Isso é possível por meio de mecanismos como a paginação, a segmentação e a troca (swapping), coordenados por componentes de hardware (como a Unidade de Gerenciamento de Memória - MMU) e software (o sistema operacional).

Além de expandir virtualmente a capacidade da memória, a memória virtual também proporciona isolamento entre processos e uma melhor organização do espaço de endereçamento, garantindo segurança e estabilidade ao sistema.

Entretanto, o uso intensivo da memória virtual pode acarretar em perda de desempenho, pois o acesso ao disco é consideravelmente mais lento que o acesso à RAM. Em situações extremas, isso pode causar o fenômeno conhecido como thrashing , no qual o sistema operacional passa mais tempo movendo dados entre a memória principal e o disco do que executando de fato os processos (TELES, 2023).

4.2 Swapping

O swapping é uma técnica que faz parte do gerenciamento da memória virtual e consiste na movimentação de processos (ou partes deles) entre a memória principal (RAM) e o disco rígido. O objetivo é liberar espaço na RAM para que outros processos possam ser carregados, especialmente quando a quantidade de memória física disponível se torna insuficiente.

Quando um processo em execução não está utilizando ativamente a CPU ou permanece inativo por um tempo, o sistema pode transferi-lo para uma área específica do disco chamada espaço de swap. Esse processo é denominado swap-out. Posteriormente, se esse processo voltar a ser requisitado, ele é recarregado na RAM por meio do swap-in.

Esse procedimento é gerenciado pelo sistema operacional, que decide dinamicamente quais processos devem ser trocados com base na disponibilidade de memória e na prioridade dos processos em execução. Embora eficaz, o swapping pode causar degradação no desempenho quando realizado com muita frequência, pois o acesso ao disco é significativamente mais lento do que o acesso direto à RAM (GEKSFORGEEKS, 2023).

4.3 Algoritmos de Substituição de Páginas (FIFO, LRU, etc.)

Para que a memória virtual funcione corretamente, o sistema operacional precisa decidir quais páginas de memória devem ser removidas da RAM quando não há espaço disponível para novas páginas. Essa decisão é tomada com base em algoritmos de substituição de páginas. Os mais comuns são:

a) **FIFO (First In, First Out)**
   O algoritmo FIFO substitui a página que está há mais tempo na memória. Ele utiliza uma estrutura de fila simples, onde a página que foi carregada primeiro será a primeira a ser removida. Apesar de sua simplicidade, o FIFO pode ter um desempenho inferior, pois não considera a frequência ou a recência de uso das páginas, podendo remover páginas que ainda são importantes para o sistema. Esse algoritmo também pode sofrer da anomalia de Belady, em que o aumento da memória física paradoxalmente aumenta o número de falhas de página (SILBERSCHATZ; GALVIN; GAGNE, 2013).

b) **LRU (Least Recently Used)**
   O algoritmo LRU remove a página que foi usada há mais tempo. Ele parte do princípio de que páginas utilizadas recentemente tendem a ser reutilizadas em breve, e aquelas que não foram acessadas há muito tempo provavelmente não serão necessárias novamente em breve. É um dos algoritmos mais eficazes e não sofre da anomalia de Belady. No entanto, requer maior complexidade na implementação, pois é ne-

cessário registrar o histórico de uso de cada página (SILBERSCHATZ; GALVIN; GAGNE, 2013).

c) **Ótimo (Optimal)**
O algoritmo Ótimo substitui a página que será utilizada mais tardiamente no futuro. Em termos teóricos, ele apresenta o melhor desempenho possível, pois sempre escolhe a página que causará o menor impacto. Contudo, por exigir o conhecimento futuro do comportamento dos processos, ele é impraticável em sistemas reais. Seu principal uso é como referência para comparação de desempenho com outros algoritmos (TANENBAUM; BOS, 2015).

A eficiência dos algoritmos de substituição de páginas é crucial para o desempenho do sistema operacional. Um algoritmo mal escolhido pode causar excesso de page faults (faltas de página), aumentando a quantidade de acessos ao disco e reduzindo significativamente a velocidade de execução dos processos.

Algoritmos como LRU são eficazes para minimizar essas faltas, pois conseguem manter na memória as páginas com maior probabilidade de serem reutilizadas. Já algoritmos mais simples, como o FIFO, embora mais fáceis de implementar, podem comprometer o desempenho ao remover páginas úteis.

A escolha do algoritmo adequado também influencia na prevenção do thrashing, garantindo que o sistema mantenha o equilíbrio entre a utilização eficiente da RAM e a necessidade de trocas com o disco. Assim, algoritmos de substituição bem projetados são essenciais para garantir a estabilidade, desempenho e responsividade de sistemas que utilizam memória virtual.

5 Evolução Histórica das Técnicas de Gerenciamento de Memória

5.1 Anos 1950-1960: Primórdios da Computação

5.1.1 Contexto Tecnológico

Nos primórdios da computação, os computadores mainframe operavam com memória extremamente limitada, geralmente medida em quilobytes. Os programas eram executados em modo batch (lote), e a memória era um recurso extremamente caro e escasso.

5.1.2 Alocação Contígua Simples

A técnica mais primitiva consistia em:

- Carregamento completo do programa na memória

- Execução de apenas um programa por vez
- Sistema operacional ocupando área fixa da memória
- **Limitação Principal:** Subutilização da CPU durante operações de I/O

Listing 1 – Estrutura de Memória Simples

```
Mem ria Total: 64 KB
          Sistema Operacional: 8 KB (fixo)
          Programa do Usu rio: 56 KB (m ximo)
```

5.1.3 Multiprogramação com Partições Fixas

Com o advento da multiprogramação, surgiu a divisão da memória em partições de tamanho fixo, permitindo múltiplos programas residentes simultaneamente. Isso introduziu o conceito de fila de processos, mas trouxe o problema da fragmentação interna severa.

5.2 Anos 1960-1970: Revolução da Memória Virtual

5.2.1 Marco Histórico: Atlas Computer

O Atlas Computer, desenvolvido na Universidade de Manchester em 1962, foi o primeiro sistema a implementar memória virtual em hardware. Este marco histórico introduziu o conceito revolucionário de paginação por demanda.

5.2.2 Paginação (1960s)

A paginação representou um salto qualitativo ao dividir a memória em páginas de tamanho fixo:

- Mapeamento através de tabelas de páginas
- Eliminação da fragmentação externa
- **Inovação:** Processos não precisavam estar contíguos na memória

5.2.3 Segmentação (1965)

Proposta no sistema Multics, a segmentação refletia a estrutura lógica dos programas:

- Proteção e compartilhamento facilitados
- Cada segmento com base e limite próprios

- Correspondência com unidades lógicas (código, dados, pilha)

5.2.4 Memória Virtual com Paginação por Demanda

No final dos anos 1960, surgiu a técnica de carregar páginas apenas quando necessárias:

- Algoritmo FIFO inicial para substituição
- Swapping automático entre memória e disco
- **Impacto:** Programas maiores que a memória física podiam executar

5.3 Anos 1970-1980: Refinamento e Otimização

5.3.1 LRU (Least Recently Used) - 1970

Baseado no princípio de localidade temporal, o LRU apresentava melhor desempenho que FIFO em muitos casos, porém sua implementação em hardware era custosa.

5.3.2 Working Set Model (Peter Denning, 1968-1970)

O conceito de conjunto de trabalho de um processo forneceu a base teórica para prevenção de thrashing e influenciou políticas de substituição modernas.

5.3.3 Algoritmo do Relógio (Clock/Second Chance) - 1970s

Uma aproximação eficiente do LRU que utilizava bit de referência, apresentando menor overhead computacional. Foi amplamente adotado em sistemas Unix.

5.4 Anos 1980-1990: Era dos Microcomputadores

5.4.1 Contexto

O surgimento de PCs com memória limitada (640 KB no MS-DOS) demandou técnicas criativas de gerenciamento.

5.4.2 Memória Expandida (EMS) e Estendida (XMS)

Soluções para ultrapassar o limite de 640 KB do MS-DOS através de bank switching, embora com programação complexa e não transparente.

5.4.3 Introdução de MMU em Microprocessadores

- **Intel 80286 (1982):** Modo protegido e segmentação
- **Intel 80386 (1985):** Paginação por hardware
- **Impacto:** Memória virtual viável em PCs

5.5 Anos 1990-2000: Consolidação de Padrões Modernos

5.5.1 TLB (Translation Lookaside Buffer)

Cache para traduções de endereços virtuais físicos, com implementação em hardware, reduzindo drasticamente o overhead de tradução.

5.5.2 Huge Pages / Large Pages

Páginas maiores (2 MB, 1 GB) além das tradicionais 4 KB, reduzindo entradas na TLB e melhorando o desempenho para aplicações com grandes conjuntos de dados.

5.5.3 Copy-on-Write (COW)

Otimização para fork() em sistemas Unix, onde páginas são compartilhadas até modificação, economizando significativamente memória.

5.6 Anos 2000-Presente: Era da Virtualização e Big Data

5.6.1 NUMA (Non-Uniform Memory Access)

Arquiteturas multiprocessador com memórias locais, otimizações para localidade de acesso e desafios no balanceamento automático de páginas.

5.6.2 Transparent Huge Pages (Linux, 2011)

Gerenciamento automático de páginas grandes, melhorando desempenho sem modificação de aplicações.

5.6.3 Memory Deduplication

- KSM (Kernel Same-page Merging) no Linux
- Compartilhamento de páginas idênticas entre processos
- Crucial para ambientes virtualizados

5.6.4 Compressão de Memória

- zRAM (Linux): Swap em memória comprimida
- Memory Compression (macOS): Compressão antes de swap
- **Benefício:** Redução de I/O em disco

5.6.5 PMEM (Persistent Memory)

Intel Optane: Memória não-volátil com velocidade próxima à RAM, introduzindo novos paradigmas de gerenciamento e desafios de integração com modelos tradicionais.

6 Implementações em Sistemas Operacionais Modernos

6.1 Linux (Kernel 6.x)

6.1.1 Arquitetura de Gerenciamento de Memória

O Linux implementa uma estrutura em camadas para gerenciamento de memória:

Listing 2 – Estrutura em Camadas do Linux

```
|     Espaco do Usuario (Processos)     |
+---------------------------------------+
                          V syscalls
+---------------------------------------+
|    Virtual Memory Subsystem (VMA)   |
| - mmap, brk, malloc                   |
+---------------------------------------+
                          V
+---------------------------------------+
|    Page Cache & Buffer Cache        |
+---------------------------------------+
                          V
+---------------------------------------+
|    Page Allocator (Buddy System)    |
+---------------------------------------+
                          V
+---------------------------------------+
|    Physical Memory (RAM)            |
+---------------------------------------+
```

6.1.2 Características Principais

6.1.2.1 Paginação por Demanda

- Páginas alocadas apenas quando acessadas (lazy allocation)
- Page fault handler carrega páginas sob demanda
- COW (Copy-on-Write) para fork()

6.1.2.2 Algoritmo de Substituição: Aproximação do LRU

- Duas listas: Active e Inactive
- Páginas movem entre listas baseado em acesso
- Política LRU-2Q (Two Queue)

6.1.2.3 Transparent Huge Pages (THP)

- Páginas de 2 MB automaticamente
- Fragmentação e desfragmentação automáticas
- Configurável por processo via madvise ()

6.1.2.4 Memory Zones

- **ZONE_DMA:** Para dispositivos DMA legacy (< 16 MB)
- **ZONE_NORMAL:** Área principal (16 MB - 896 MB em 32-bit)
- **ZONE_HIGHMEM:** Acima de 896 MB (apenas 32-bit)
- **ZONE_MOVABLE:** Páginas migráveis

6.1.2.5 OOM Killer (Out-of-Memory Killer)

- Mata processos quando memória esgota
- Score baseado em uso de memória e importância
- Configurável via /proc/[pid]/oom_score_adj

6.1.3 Comandos de Análise

Listing 3 – Comandos para Análise de Memória no Linux

```
# Visualizar uso de mem ria
free -h
cat /proc/meminfo

# Estat sticas de pagina  o
vmstat 1

# Informa  es sobre huge pages
cat /proc/sys/vm/nr_hugepages

# OOM score de um processo
cat /proc/[pid]/oom_score
```

6.1.4 Parâmetros Configuráveis (sysctl)

Listing 4 – Configurações via sysctl

```
# Swappiness (0-100): prefer ncia por swap
vm.swappiness = 60

# Dirty ratio: % de mem ria suja antes de writeback
vm.dirty_ratio = 20

# Overcommit memory
vm.overcommit_memory = 0  # heur stica (padr o)
```

6.2 Windows (Windows 10/11 e Server)

6.2.1 Arquitetura do Memory Manager

6.2.1.1 Virtual Address Descriptors (VADs)

- Estrutura em árvore para rastreamento de regiões virtuais
- Informações de proteção e mapeamento
- Suporte a Copy-on-Write

6.2.1.2 Working Set Manager

- Controla conjunto de páginas residentes de cada processo
- Working Set Minimum e Maximum configuráveis
- Trimming automático sob pressão de memória

6.2.1.3 Memory Compression

- Introduzido no Windows 10 (2015)
- Comprime páginas antes de swap
- Processo "System"mostra memória comprimida
- Redução de I/O em 50-70

6.2.1.4 SuperFetch / Prefetch

- Pré-carregamento inteligente de páginas
- Análise de padrões de uso
- Renomeado para "SysMain"no Windows 10

6.2.2 Ferramentas de Análise

- **Task Manager:** Visão geral de memória
- **Resource Monitor:** Detalhamento por processo
- **Performance Monitor:** Contadores detalhados
- **RAMMap (Sysinternals):** Análise profunda de uso

6.3 macOS (Darwin/XNU Kernel)

6.3.1 Unified Buffer Cache

6.3.1.1 Memória Comprimida

Desde OS X Mavericks (2013):

- Compressão de páginas inativas antes de swap
- Algoritmo WKdm (Wilson-Kaplan direct-mapped)

- Taxa de compressão típica: 2:1 a 3:1
- **Vantagem:** Mantém mais dados em RAM

6.3.1.2 Páginas Purgeable (Expurgáveis)

- API para marcar memória como descartável
- Sistema pode liberar sem swap
- Útil para caches de aplicativos
- Recarregável sob demanda

6.3.1.3 Memory Pressure

- Sistema de sinais para aplicativos
- Níveis: Normal, Warning, Critical
- Apps recebem notificações para liberar memória
- Resposta cooperativa em vez de forçada

6.3.2 Estados de Memória no macOS

- **Wired (Travada):** Não pode ser paginada (kernel, drivers)
- **Active (Ativa):** Recentemente usada
- **Inactive (Inativa):** Não usada recentemente, candidata à liberação
- **Compressed (Comprimida):** Compactada mas ainda em RAM
- **Free (Livre):** Disponível

6.4 FreeBSD

6.4.1 Características do VM System

6.4.1.1 Design Modular

- Baseado no Mach VM original
- Objetos VM (vm_object) para abstração
- Suporte a diferentes backends (swap, vnode, etc.)

6.4.1.2 Page Queue System

- Múltiplas filas: Active, Inactive, Cache, Free
- Clock algorithm para movimentação entre filas

6.4.1.3 Superpages

- Suporte a páginas de 2 MB e 1 GB
- Promoção automática de páginas contíguas
- Redução de pressure na TLB

6.5 Android (Linux Kernel Modificado)

6.5.1 Adaptações para Dispositivos Móveis

6.5.1.1 Low Memory Killer (LMK)

- Substituto do OOM Killer padrão
- Mata processos baseado em prioridade (oom_adj)
- Limiares configuráveis por nível de memória
- **Objetivo:** Manter sistema responsivo

6.5.1.2 zRAM (Compressed Swap in RAM)

- Swap em memória comprimida
- Evita I/O lento em flash
- Compressão LZ4 (rápida)
- Típico: 25-50

6.5.1.3 Ciclo de Vida de Processos

Foreground -> Visible -> Service -> Background -> Empty -> [Killed por LMK]

Tabela 1 – Comparação de Implementações de Gerenciamento de Memória

Aspecto	**Linux**	**Windows**	**macOS**	**FreeBSD**	**Android**
Algoritmo Base	LRU-2Q	Clock	LRU + Compress	Clock PQ	LRU + zRAM
Huge Pages	THP (2 MB)	Large Pages	Superpages	Superpages	Limitado
Compressão	zRAM	Nativa (2015+)	Nativa (2013+)	Não nativa	zRAM
Overcommit	Configurável	Sim (padrão)	Limitado	Não (padrão)	Sim
Swap	Partição/Arq	Arquivo	Arquivo	Partição	zRAM
Memory Killer	OOM Killer	Trimming	Memory Press.	*N/A*	LMK
NUMA	Sim	Sim	Sim (Mac Pro)	Sim	Não
Page Size	4 KB	4 KB	4 KB / 16 KB	4 KB	4 KB

7 Comparativo Entre Sistemas Operacionais

7.1 Tabela Comparativa

7.2 Filosofias de Design

7.2.1 Linux

- Flexibilidade e configurabilidade máximas
- Políticas agressivas de cache ("unused RAM is wasted RAM")
- Overcommit por padrão

7.2.2 Windows

- Foco em responsividade e experiência do usuário
- Working sets bem definidos por processo
- Compressão para evitar swap quando possível

7.2.3 macOS

- Balanceamento entre performance e eficiência energética
- Integração profunda com framework de aplicativos
- Notificações proativas para aplicações

7.2.4 Android

- Agressividade na liberação de memória
- Priorização de app foreground
- Otimização para hardware limitado

8 Tendências e Direções Futuras

8.1 Tecnologias Emergentes

8.1.1 Persistent Memory (PMEM)

- Intel Optane DC Persistent Memory
- Velocidade próxima à DRAM, persistência como SSD
- Desafio: Novos modelos de programação
- Suporte inicial em Linux 4.x+

8.1.2 CXL (Compute Express Link)

- Protocolo para memória compartilhada entre CPU e aceleradores
- Pooling de memória em datacenters
- Potencial para mudança de paradigma em gerenciamento

8.1.3 Machine Learning em Gerenciamento de Memória

- Predição de page faults
- Pré-carregamento inteligente
- Otimização dinâmica de working sets
- Pesquisas em andamento (Google, Microsoft Research)

8.2 Desafios Atuais

8.2.1 Security vs Performance

- Mitigações Spectre/Meltdown impactam TLB
- Kernel Page Table Isolation (KPTI)
- Trade-off entre segurança e velocidade

8.2.2 Heterogeneous Memory

- Combinação DRAM + PMEM + HBM
- Placement automático de dados
- Gestão de latências variáveis

9 Aplicações Práticas e Casos de Uso

9.1 Servidores Web (Nginx, Apache)

9.1.1 Desafio

- Alto número de conexões simultâneas
- Cache de conteúdo estático
- Minimizar page faults durante picos

9.1.2 Soluções Implementadas

- Huge pages para tabelas de conexão
- madvise (MADV_WILLNEED) para pré-carregamento
- Ajuste de vm.swappiness para evitar swap

9.2 Bancos de Dados (PostgreSQL, MySQL)

9.2.1 Implementações

- Buffer pools próprios (ignoram page cache)
- Uso de huge pages para shared memory
- Direct I/O para evitar double caching

9.3 Ambientes de Virtualização

9.3.1 Técnicas Utilizadas

9.3.1.1 Linux KVM

- KSM (Kernel Same-page Merging): Deduplicação
- Ballooning: VM devolve memória sob demanda
- Transparent Huge Pages para VMs

9.3.1.2 Hyper-V

- Dynamic Memory: Ajuste automático por VM
- Smart Paging: Swap temporário para startup
- NUMA spanning configurável

10 Conclusão

10.1 Evolução Consolidada

A evolução do gerenciamento de memória demonstra uma trajetória clara desde a simplicidade das implementações dos anos 1950 até as sofisticadas técnicas modernas que incorporam compressão, deduplicação e até inteligência artificial.

10.2 Convergências e Divergências

10.2.1 Convergências

- Todos os SOs modernos usam paginação por demanda
- Aproximações do LRU são universais
- Compressão antes de swap está se tornando padrão
- Huge pages amplamente adotadas

10.2.2 Divergências

- Filosofias de overcommit (Linux agressivo vs FreeBSD conservador)
- Handling de pressão de memória (killing vs compression vs notification)
- Defaults e tunables (Linux configurável vs macOS opaco)

10.3 Lições Aprendidas

1. **Não existe solução única:** Cada SO otimiza para seu contexto
2. **Hardware direciona software:** Avanços como TLB e MMU moldam implementações
3. **Trade-offs são inevitáveis:** Performance vs segurança, flexibilidade vs simplicidade
4. **Monitoramento é essencial:** Todos os SOs fornecem ferramentas robustas

Referências

[1] Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.

[2] Tanenbaum, A. S., & Bos, H. (2014). *Modern Operating Systems* (4th ed.). Pearson.

[3] Love, R. (2010). *Linux Kernel Development* (3rd ed.). Addison-Wesley.

[4] Russinovich, M. E., Solomon, D. A., & Ionescu, A. (2017). *Windows Internals* (7th ed.). Microsoft Press.

[5] Linux Kernel Documentation: Memory Management. https://www.kernel.org/doc/html/latest/admin-guide/mm/

[6] FreeBSD Architecture Handbook: VM System. https://docs.freebsd.org/en/books/arch-handbook/vm/

[7] Denning, P. J. (1970). Virtual Memory. *ACM Computing Surveys*, 2(3), 153-189.

[8] Waldspurger, C. A. (2002). Memory Resource Management in VMware ESX Server. OSDI.

[9] BLOG GRAN CURSOS ONLINE. (2024). *Sistemas Operacionais Gerência de Memória*. Disponível em: https://blog.grancursosonline.com.br/sistemas-operacionais-gerencia-de-memoria/. Acesso em: 22 out. 2025.

[10] CENTRO DE INFORMÁTICA (CIn) UFPE. *Gerenciamento de Me- mória*. Disponível em: https://www.cin.ufpe.br/~can/Arquivos/gerenciamento-de-memoria.htm. Acesso em: 22 out. 2025.

[11] GRUPO DE TELEINFORMÁTICA E AUTOMAÇÃO (GTA) UFRJ. *Segmen- tação de Memória - Sistemas Operacionais*. Disponível em: https://www.gta.ufrj.br/~cruz/courses/eel770/slides/12_segmentacao.pdf. Acesso em: 22 out. 2025.

[12] INSTITUTO DE MATEMÁTICA E ESTÁTISTICA (IME) USP. *Memória virtual paginação e segmentação*. Disponível em: https://www.ime.usp.br/~song/mac344/slides07-virtual-memory.pdf. Acesso em: 22 out. 2025.

[13] LUCASGABRIEL. (2023). *Como funciona o processo de Segmentação de memória?* Medium. Disponível em: https://medium.com/@snowden5958/como-funciona-o-processo-de-segmenta%C3%A7%C3%A3o-de-mem%C3%B3ria-897776625f4d. Acesso em: 22 out. 2025.

[14] RIBAS, Bruno César. (2012). *Gerenciamento de Memória*. Disponível em: https://www.brunoribas.com.br/so/2012-1/a.gerencia-de-memoria.pdf. Acesso em: 22 out. 2025.

[15] UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO (UFES). *Gerência de Memória Segmentação*. Disponível em: https://www.inf.ufes.br/~rgomes/so_fichiers/aula21.pdf. Acesso em: 22 out. 2025.

[16] UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL (UFRGS). *Pagi- nação Como alocar memória?* Disponível em: https://www.inf.ufrgs.br/~johann/sisop1/aula14b.paginas.pdf. Acesso em: 22 out. 2025.

[17] WIKIPÉDIA. *Paginação de memória*. Wikipédia, a enciclopédia livre. Dispo- nível em: https://pt.wikipedia.org/wiki/Pagina%C3%A7%C3%A3o_de_mem%C3%B3ria. Acesso em: 22 out. 2025.

[18] WIKIPÉDIA. *Segmentação (memória)*. Wikipédia, a enciclopLdia livre. Disponível em: https://pt.wikipedia.org/wiki/Segmenta%C3%A7%C3%A3o_(mem%C3%B3ria). Acesso em: 22 out. 2025.

[19] VASCONCELLOS, J. (2011). *Gerenciamento de Memória*. Disponível em: https://jvasconcellos.com.br/wp-content/uploads/2011/04/ger_memoria.pdf. Acesso em: 20 out. 2025.

[20] MATTIELLO, Caciano D.; SANTOS, Eduardo dos. (2011). *Métodos de Alocação*. Sistemas Operacionais – slides. Disponível em: https://www.brunoribas.com.br/so/slides-fs/FS-metodos-alocacao-caciano-eduardo.pdf. Acesso em: 20 out. 2025.

[21] UF RGS. (

s.d.

). *Curso de Sistemas Operacionais — Aula 11*. Disponível em: http://www.inf.ufrgs.br/~asc/sisop/pdf/aula11.pdf. Acesso em: 20 out. 2025.

[22] GAURAV, Sushant. (2024). *Difference Between Contiguous and Non-Contiguous Memory Allocation in OS*. Scaler Topics. Disponível em: https://www.scaler.com/topics/contiguous-and-non-contiguous-memory-allocation-in-os/. Acesso em: 22 out. 2025.

[23] LITHMEE. (2018). *Difference Between Contiguous and Non-contiguous Memory Allocation*. Pediaa – Technology / IT / Systems. Disponível em: https://pediaa.com/difference-between-contiguous-and-noncontiguous-memory-allocation/. Acesso em: 22 out. 2025.

[24] GARCEZ, Letícia. (2023). *O que é hierarquia de memória*. Blog Formação DEV. Disponível em: https://blog.formacao.dev/o-que-e-hierarquia-de-memoria/. Acesso em: 22 out. 2025.

[25] Jacob, B., Ng, S., & Wang, D. (2002). *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers.

[26] INSTITUTO FEDERAL DE MINAS GAERIS. (

s.d.

). *Hierarquia das Memórias*. Campus Betim. Disponível em: `https://wiki.betim.ifmg.edu.br/docs/organizacao_computadores/hierarquia_memorias`. Acesso em: 22 out. 2025.

[27] TELES, Linda. (2023). *Virtual memory (memória virtual)*. TechTarget – SearchStorage. Disponível em: `https://www.techtarget.com/searchstorage/definition/virtual-memory`. Acesso em: 17 out. 2025.

[28] GEKSFORGEEKS. (2023). *Swapping in Operating System*. Disponível em: `https://www.geeksforgeeks.org/swapping-in-operating-system/`. Acesso em: 17 out. 2025.

[29] SILBERSCHATZ, Abraham; GALVIN, Peter B.; GAGNE, Greg. (2013). *Fundamentos de Sistemas Operacionais* (9. ed.). Rio de Janeiro: LTC.

[30] TANENBAUM, Andrew S.; BOS, Herbert. (2015). *Modern Operating Systems* (4th ed.). Pearson.