UNIVERSIDADE CESUMAR - UNICESUMAR

MAITÊ FELD, ARTHUR AFONSO PEREIRA MISTURA, BRENDON CESARIO, HENRIQUE CHITTOLINA SILVA, EDUARDO VINICIUS VIANTE VIEIRA, LUCAS CRAIS, PEDRO LEANDRO HACK RUTHES, BRUNO DANKER

GERENCIAMENTO DE MEMÓRIA EM SISTEMAS OPERACIONAIS

CURITIBA

2025

MAITÊ FELD, ARTHUR AFONSO PEREIRA MISTURA, BRENDON CESARIO, HENRIQUE CHITTOLINA SILVA, EDUARDO VINICIUS VIANTE VIEIRA, LUCAS CRAIS, PEDRO LEANDRO HACK RUTHES, BRUNO DANKER

GERENCIAMENTO DE MEMÓRIA EM SISTEMAS OPERACIONAIS

Trabalho acadêmico apresentado ao curso de Engenharia de Software da Universidade Cesumar como atividade avaliativa do semestre, sob orientação do(a) Prof. José Carlos Domingues Flores

CURITIBA

2025

Z2

RESUMO

O presente trabalho tem como objetivo apresentar e aplicar os principais conceitos relacionados ao gerenciamento de memória em sistemas operacionais, enfatizando os mecanismos de alocação, fragmentação e coleta de lixo. Para isso, foram desenvolvidos programas nas linguagens C, Python e Java, com o intuito de demonstrar na prática o funcionamento das estratégias de gerenciamento de memória estudadas. Entre os temas abordados, destacam-se a alocação estática e dinâmica, a fragmentação interna e externa, os algoritmos de substituição de páginas (FIFO e LRU), o mecanismo de Garbage Collection e a comparação de desempenho entre diferentes tipos de alocação. Através das implementações, foi possível compreender como o sistema operacional gerencia os recursos de memória, otimizando o desempenho e garantindo o uso eficiente do hardware.

SUMÁRIO

1. ALOCAÇÃO ESTÁTICA E DINÂMICA

Todo programa, para ser executado, precisa utilizar memória. Quando sua execução se inicia, ele solicita ao sistema operacional uma quantidade de memória necessária para armazenar suas variáveis, instruções e estruturas internas, esse processo é conhecido como alocação de memória. No entanto, nem sempre a quantidade reservada no início da execução é suficiente, e o programa pode precisar solicitar mais memória durante o seu funcionamento.

A alocação de memória estática é um processo no qual a memória para variáveis e estruturas de dados é reservada em tempo de compilação, antes da execução do programa. Isso significa que o tamanho e a localização dos blocos de memória são fixos e não podem ser alterados em tempo de execução. Esse tipo de alocação de memória é rápido e simples, pois não há necessidade de alocar ou desalocar memória durante a execução, e evita a fragmentação da memória, pois os blocos de memória são contíguos e alinhados. Além disso, reduz o risco de vazamentos de memória, pois a memória é liberada automaticamente quando o programa termina. No entanto, a alocação de memória estática é inflexível e desperdiçada, pois você precisa especificar a quantidade máxima de memória necessária, independentemente do uso real. Ele também limita a escalabilidade e a modularidade do seu programa, pois você não pode criar ou destruir dinamicamente objetos ou matrizes, e pode causar estouro de pilha se você alocar muita memória na pilha, que é uma região de memória limitada A alocação dinâmica de memória é um processo que aloca memória para variáveis e estruturas de dados em tempo de execução, quando o programa solicita. Isso permite flexibilidade e eficiência, pois o tamanho e a localização dos blocos de memória podem ser alterados de acordo com a lógica do programa e o tamanho dos dados. Ele também permite a criação e manipulação de estruturas de dados complexas e dinâmicas, como listas vinculadas, árvores, gráficos e tabelas de hash. Além disso, a alocação dinâmica de memória permite que o programa se adapte a diferentes ambientes e entradas do usuário, já que o uso de memória pode ser ajustado em tempo de execução. No entanto, a alocação dinâmica de memória é mais lenta e complexa, pois você mesmo precisa gerenciar a alocação e a desalocação de memória. Também pode causar fragmentação de memória e vazamentos de memória.

Na linguagem C, a alocação dinâmica é feita por meio de quatro funções principais da biblioteca <stdlib.h>: malloc(), calloc(), realloc() e free(). As funções malloc() e free() são as mais utilizadas. A função malloc() serve para reservar um espaço de memória e retorna um ponteiro genérico (void *) que aponta para o início dessa área. Por exemplo:

```
int *vetor;

vetor = (int *) malloc(100 * sizeof(int));
```

Nesse exemplo, o programa está reservando espaço para armazenar 100 números inteiros. Como cada int ocupa 4 bytes, o total de memória alocada é de 400 bytes, geralmente localizados na área heap da memória. É importante sempre verificar se a alocação foi bem-sucedida, pois, se não houver memória disponível, malloc() retorna NULL:

```
if (vetor == NULL) {
   printf("Sem memória suficiente!");
   exit(1);
}
```

Quando o espaço alocado não for mais necessário, ele deve ser liberado com a função free(), permitindo que o sistema reutilize aquela memória. A função calloc() funciona de maneira semelhante à malloc(), mas, além de reservar o espaço, ela inicializa todos os bytes com zero. Já a função realloc() permite redimensionar um bloco de memória previamente alocado, aumentando ou diminuindo seu tamanho conforme a necessidade.

Contudo, a diferença principal entre os dois tipos de alocação está no momento em que o espaço é reservado: na alocação estática, o tamanho é definido na compilação; na alocação dinâmica, ele é determinado durante a execução. A alocação estática é mais direta e segura, porém menos flexível. A alocação dinâmica, por outro lado, oferece mais liberdade e eficiência no uso da memória, mas exige maior cuidado do programador para evitar vazamentos de memória e outros erros relacionados à má gestão de recursos.

2. FRAGMENTAÇÃO DE MEMÓRIA

A fragmentação de memória é um problema comum nos sistemas operacionais que utilizam técnicas de alocação dinâmica de memória. Ela ocorre quando o espaço disponível na memória principal é utilizado de forma ineficiente, resultando em áreas livres que não podem ser aproveitadas adequadamente por novos processos. Esse fenômeno é dividido em dois tipos principais: fragmentação externa e fragmentação interna, que se diferenciam pela forma e pelo local em que o desperdício ocorre.

A fragmentação externa acontece quando a memória livre está dividida em pequenos blocos dispersos, impossibilitando o sistema de encontrar um espaço contínuo suficientemente grande para alocar um novo processo, mesmo que a soma total de memória livre seja suficiente. Esse tipo de fragmentação é típico em sistemas que utilizam partições de tamanho variável, nos quais os processos são carregados e removidos da memória de maneira dinâmica, criando “buracos” entre as áreas ocupadas.

Por exemplo, imagine que um computador possua 1 GB de memória RAM livre, distribuída em blocos de 200 MB, 300 MB e 500 MB. Caso um novo processo necessite de 600 MB contínuos para ser executado, ele não poderá ser alocado, mesmo havendo 1 GB livre no total. Isso ocorre porque o espaço disponível está fragmentado em blocos separados, e o sistema não consegue juntar essas áreas de forma contígua. Esse é o caso clássico da fragmentação externa, onde há memória disponível, mas ela está distribuída de forma desordenada, dificultando novas alocações.

Já a fragmentação interna ocorre quando um processo não utiliza completamente o espaço de memória que lhe foi reservado. Nesse caso, o sistema operacional aloca um bloco maior do que o necessário, e o espaço restante dentro desse bloco permanece ocioso, sem ser utilizado. Essa situação é comum em sistemas que utilizam partições de tamanho fixo.

Por exemplo, considere um sistema que divide a memória em blocos de 100 MB. Se um programa necessita apenas de 85 MB, ainda assim ele ocupará uma partição inteira de 100 MB, deixando 15 MB sem uso dentro do bloco. Esse espaço desperdiçado representa a fragmentação interna, pois o desperdício ocorre dentro de uma área já alocada.

Em resumo, tanto a fragmentação interna quanto a externa representam formas de desperdício de memória que impactam diretamente o desempenho e a eficiência do sistema. A

fragmentação interna ocorre dentro dos blocos de memória já alocados, quando o espaço destinado ao processo é maior do que o necessário. Já a fragmentação externa ocorre entre os blocos, quando o espaço livre está espalhado pela memória, impedindo novas alocações contíguas.

A principal diferença entre as duas está no local do desperdício: a fragmentação interna ocorre dentro dos blocos de memória alocados, enquanto a fragmentação externa ocorre fora deles, entre as áreas livres. Para minimizar esses problemas, os sistemas operacionais modernos utilizam técnicas de gerenciamento de memória como paginação, segmentação e compactação, que reorganizam ou distribuem os processos de forma mais eficiente, reduzindo a fragmentação e melhorando o aproveitamento dos recursos disponíveis.

3. ALGORITMO DE SUBSTITUIÇÃO DE PÁGINA - FIFO

O algoritmo de substituição de página FIFO (First-In, First-Out), ou “primeiro a entrar, primeiro a sair”, é um dos métodos mais antigos e simples utilizados pelos sistemas operacionais para o gerenciamento da memória virtual. Sua lógica é intuitiva e semelhante à de uma fila comum: o primeiro elemento que entra é também o primeiro a sair. No contexto da memória, isso significa que, quando todos os quadros disponíveis estão ocupados e uma nova página precisa ser carregada, o sistema remove aquela que está há mais tempo na memória, ou seja, a que foi inserida primeiro.

O funcionamento do algoritmo é bastante direto. À medida que o programa é executado, novas páginas são carregadas na memória física e inseridas em uma estrutura de dados do tipo fila, que mantém a ordem de chegada. Enquanto houver espaço livre, o sistema apenas adiciona as páginas conforme necessário. No entanto, quando a memória atinge sua capacidade máxima, o algoritmo deve decidir qual página será removida para abrir espaço para a nova. No caso do FIFO, essa decisão é simples: ele escolhe a página mais antiga, a primeira a ter sido carregada, e a substitui pela nova página requisitada. Assim, a fila é atualizada continuamente, mantendo sempre a ordem cronológica de inserção das páginas na memória.

Para exemplificar, imagine uma memória física com capacidade para três páginas e uma sequência de acessos composta por 1, 2, 3 e 4. Inicialmente, as páginas 1, 2 e 3 são carregadas sem dificuldade, pois ainda há espaço disponível. No entanto, quando ocorre o acesso à página 4, a memória já está completamente ocupada, e o algoritmo precisa liberar um espaço. Seguindo o princípio do FIFO, a página 1, que foi a primeira a entrar, é removida para dar lugar à página 4. Esse processo se repete continuamente, removendo sempre a página mais antiga e inserindo a nova no final da fila. Embora seja um método de fácil entendimento, o FIFO nem sempre é o mais eficiente, pois não considera o padrão real de utilização das páginas.

Entre as principais vantagens do algoritmo FIFO está a simplicidade de implementação. Ele é de fácil compreensão e exige poucos recursos do sistema, uma vez que utiliza apenas uma estrutura de fila e não depende de cálculos complexos, contadores de tempo ou monitoramento da frequência de uso das páginas. Por esse motivo, o FIFO é amplamente empregado como ferramenta didática para o estudo de algoritmos de substituição de páginas, além de ser útil em sistemas de pequeno porte, onde a demanda por otimização de

desempenho é menor. Outro ponto positivo é o fato de o FIFO seguir uma regra fixa e previsível, apresentando comportamento determinístico que facilita sua análise e teste em ambientes controlados.

Entretanto, o FIFO também apresenta limitações importantes quando comparado a algoritmos mais modernos. A principal desvantagem é o fato de ele ignorar o padrão de acesso às páginas. Assim, uma página frequentemente utilizada pode ser removida apenas por ter sido carregada há mais tempo, resultando em um aumento no número de faltas de página, situações em que o sistema precisa buscar uma página no disco, o que é um processo lento e prejudica o desempenho geral do sistema. Além disso, o FIFO pode sofrer com o chamado efeito de Belady, um fenômeno no qual o aumento da quantidade de quadros de memória, em vez de reduzir as faltas de página, pode paradoxalmente aumentá-las. Esse comportamento contraditório demonstra que o FIFO nem sempre apresenta um desempenho lógico ou eficiente.

Em termos práticos, o algoritmo FIFO é considerado justo, pois todas as páginas eventualmente serão substituídas, uma vez que a ordem de remoção é puramente cronológica. Contudo, ele não é necessariamente inteligente, já que ignora o princípio da localidade de referência (conceito segundo o qual os programas tendem a acessar repetidamente um conjunto limitado de páginas em curtos intervalos de tempo). Por essa razão, em sistemas que exigem maior eficiência e desempenho no uso da memória, o FIFO costuma ser substituído por algoritmos mais avançados, como o LRU (Least Recently Used), que considera o tempo de uso mais recente das páginas, ou o LFU (Least Frequently Used), que leva em conta a frequência de utilização.

Dessa forma, embora o FIFO continue sendo um método fundamental para fins didáticos e para aplicações simples, ele é limitado em ambientes complexos, nos quais a otimização do desempenho da memória virtual é essencial. O estudo desse algoritmo, contudo, permanece importante, pois serve como base para a compreensão e o aprimoramento de técnicas mais sofisticadas de gerenciamento de memória nos sistemas operacionais modernos.

4. GARBAGE COLLECTION

Em Python, o programador não precisa se preocupar em liberar manualmente a memória utilizada pelos objetos durante a execução de um programa. Essa tarefa é realizada automaticamente por um mecanismo interno conhecido como Garbage Collector, ou coletor de lixo. O principal objetivo desse sistema é identificar objetos que não estão mais sendo utilizados pelo código e removê-los da memória, liberando espaço e evitando o desperdício de recursos computacionais.

O método primário utilizado pelo Python para gerenciar a memória é a contagem de referências (reference counting). Esse mecanismo funciona de forma simples e eficiente: ao criar um objeto, o interpretador associa a ele um contador que registra quantas variáveis ou estruturas de dados estão apontando para aquele objeto. Sempre que uma nova referência é criada (por exemplo, ao atribuir o mesmo objeto a uma segunda variável) esse contador é incrementado. Quando uma referência é removida, seja por exclusão explícita com o comando del, seja pela saída de escopo de uma função, o contador é decrementado.

Quando o contador atinge o valor zero, significa que o objeto não é mais acessível por nenhuma parte do programa, e nesse momento o Python o remove imediatamente da memória. Por exemplo, ao criar a lista a = [1, 2, 3] e em seguida atribuir b = a, ambas as variáveis passam a referenciar o mesmo objeto, elevando o contador de referências para dois. Caso a seja removida com o comando del a, o contador é reduzido para um, mas o objeto ainda permanece na memória, pois b continua apontando para ele. Somente quando b também for removida o contador chegará a zero, e o objeto será definitivamente destruído. Esse processo é rápido, determinístico e resolve a maior parte das situações de liberação de memória em programas escritos em Python.

Entretanto, a contagem de referências apresenta uma limitação importante: ela não é capaz de detectar referências circulares. Esse problema ocorre quando dois ou mais objetos se referenciam mutuamente, formando um ciclo. Imagine, por exemplo, dois objetos a e b, em que a.ref = b e b.ref = a. Mesmo que todas as variáveis externas que apontavam para a e b sejam removidas, cada objeto ainda possui uma referência, um apontando para o outro. Como consequência, o contador de referências de ambos nunca atinge zero, impedindo que sejam liberados da memória. Caso o Python dependesse exclusivamente desse método, tais ciclos causariam vazamentos de memória.

Para contornar essa limitação, o Python utiliza um mecanismo complementar conhecido como coletor de lixo geracional (generational garbage collector), disponível no módulo gc. Esse sistema trabalha em conjunto com a contagem de referências, sendo responsável por identificar e eliminar objetos envolvidos em ciclos de referência que não são mais acessíveis. O termo “geracional” refere-se à organização dos objetos em três gerações (0, 1 e 2), de acordo com o tempo de vida de cada um na memória. A geração 0 contém os objetos recém-criados, a geração 1 agrupa os objetos que sobreviveram a uma ou mais coletas da geração 0, e a geração 2 abriga objetos de longa duração, que resistiram a várias varreduras.

Essa estratégia baseia-se na observação empírica de que a maioria dos objetos têm vida curta, ou seja, é criada e destruída rapidamente, enquanto apenas uma pequena parte deles permanece ativa por longos períodos. Por isso, o coletor realiza verificações mais frequentes na geração 0, onde há maior probabilidade de encontrar objetos descartáveis. Quando um objeto sobrevive a uma coleta, ele é promovido para uma geração superior, reduzindo o custo de análise sobre objetos antigos. Periodicamente, o coletor executa varreduras em todas as gerações, identificando grupos de objetos que se referenciam mutuamente, mas que não possuem nenhuma referência externa acessível a partir do programa. Esses objetos são então reconhecidos como “lixo cíclico” e removidos da memória.

Em síntese, o Garbage Collector do Python combina dois mecanismos complementares que garantem uma gestão eficiente e segura da memória: a contagem de referências, que remove imediatamente objetos sem referências ativas, e o coletor de lixo geracional, que identifica e elimina ciclos de objetos inacessíveis. Juntos, esses sistemas permitem que o programador concentre seus esforços na lógica do programa, sem se preocupar com o gerenciamento manual da memória. Dessa forma, o Python mantém um uso de recursos otimizado, evitando vazamentos e garantindo o bom desempenho das aplicações.

5. COMPARAÇÃO DE DESEMPENHO DE ALOCAÇÃO

A diferença entre Stack (Pilha) e Heap (Monte) está na forma como cada uma gerencia a memória e no tipo de dados que armazena. A Stack é mais rápida e previsível, enquanto o Heap é mais flexível, porém mais lento. Essa distinção é resultado direto das estruturas de organização e dos mecanismos de gerenciamento utilizados por cada uma dessas áreas de memória. Ambas desempenham papéis fundamentais na execução de programas, mas com propósitos e comportamentos distintos.

A Stack (Pilha) é uma área de memória otimizada para velocidade e simplicidade, sendo utilizada principalmente para o armazenamento de variáveis locais e dados temporários, ou seja, informações que existem apenas durante a execução de uma função. Sua principal característica é seguir a estrutura LIFO (Last-In, First-Out) — o último dado inserido é o primeiro a ser removido. Esse modelo garante que a alocação e a liberação de memória ocorram em uma ordem previsível e extremamente rápida. Quando uma função é chamada, as variáveis locais e parâmetros são automaticamente empilhados na Stack, e, ao final da execução, toda essa área é liberada de forma imediata, sem necessidade de código adicional.

A alocação de memória na Stack é praticamente instantânea, pois o processador precisa apenas mover o ponteiro da pilha (Stack Pointer) para reservar o espaço necessário, o que pode ser realizado com uma única instrução de CPU. Além disso, como o crescimento e a redução da pilha ocorrem de maneira sequencial, não há ocorrência de fragmentação, e os dados armazenados possuem alta localidade de cache, o que melhora o desempenho do sistema. Um exemplo prático pode ser visto em uma função simples em linguagem C:

```
void exemplo() {

   int x = 10;  // variável armazenada na Stack

}
```

Nesse caso, a variável x é criada na pilha quando a função é executada e é automaticamente destruída ao término da execução, liberando o espaço ocupado. Apesar de sua eficiência, a Stack possui tamanho limitado e fixo; caso o programa utilize mais memória do que o permitido, ocorre o erro conhecido como Stack Overflow.

Por outro lado, o Heap (Monte) é utilizado para o armazenamento de dados dinâmicos, que precisam persistir por mais tempo ou cujo tamanho só é conhecido em tempo de execução. Sua principal vantagem é a flexibilidade, permitindo que o programa solicite e libere memória conforme a necessidade. No entanto, essa flexibilidade traz um custo: o gerenciamento do Heap é mais lento e complexo. A alocação é realizada por meio de funções como malloc() em C ou new em C++, que solicitam ao sistema um bloco de memória de tamanho apropriado. Esse processo demanda tempo, pois o sistema precisa procurar um espaço livre suficientemente grande para atender à solicitação.

O Heap está sujeito a fragmentação, já que blocos de tamanhos variados podem ser alocados e liberados em ordens diferentes, gerando espaços vazios dispersos na memória. Além disso, o gerenciamento exige estruturas auxiliares chamadas metadados, que armazenam informações sobre o tamanho e o estado de cada bloco, aumentando a sobrecarga do sistema. Outro ponto importante é que a desalocação da memória no Heap não é automática. Em linguagens como C e C++, o programador deve liberar manualmente o espaço utilizado por meio de funções como free() ou delete. Já em linguagens com coletor de lixo (Garbage Collector), como Java, C# ou Python, o sistema realiza essa tarefa de forma automática, embora isso possa causar pequenas pausas na execução do programa.

Um exemplo simples em C demonstra esse comportamento:

```
int* p = malloc(sizeof(int));  // alocação no Heap

*p = 10;

free(p);  // liberação manual
```

Nesse exemplo, a variável p é armazenada na Stack, mas o valor apontado por ela (no caso, o inteiro 10) está localizado no Heap. Esse modelo ilustra bem como as duas áreas interagem: a Stack armazena o ponteiro, e o Heap armazena o dado dinâmico.

Em síntese, a principal diferença estrutural entre Stack e Heap está na forma como cada uma organiza e gerencia a memória. A Stack segue um modelo linear e ordenado, controlado automaticamente pelo compilador e pela CPU, proporcionando alocação rápida, previsível e livre de fragmentação. É ideal para dados temporários e variáveis locais de curta

duração. Já o Heap adota uma estrutura dinâmica e desordenada, permitindo alocação flexível e persistente, mas com maior custo computacional, risco de fragmentação e necessidade de controle mais cuidadoso.

Por fim, o tempo de vida dos dados também é um fator de distinção importante. Enquanto as variáveis armazenadas na Stack existem apenas dentro do escopo da função em que foram criadas, as alocações realizadas no Heap permanecem na memória até que sejam explicitamente liberadas ou que o coletor de lixo as remova. Assim, a Stack prioriza desempenho e simplicidade, enquanto o Heap privilegia flexibilidade e persistência. Ambos os mecanismos são essenciais para o funcionamento eficiente de programas, e compreender suas diferenças é fundamental para a escrita de código otimizado e seguro em qualquer linguagem de programação.

Quadro 1 – Comparação entre Stack e Heap

Aspecto	**Stack (Pilha)**	**Heap (Monte)**
Tipo de alocação	Automática	Dinâmica
Controle de memória	Feito automaticamente pelo compilador e CPU	Feito manualmente pelo programador (C/C++) ou automaticamente pelo Garbage Collector (Java, C#, Python)
Velocidade	Muito rápida (operações de alocação e liberação instantâneas)	Mais lenta, devido à busca e gerenciamento de blocos livres
Estrutura de dados	LIFO	Estrutura não linear e não ordenada
Uso típico	Variáveis locais, parâmetros de funções e dados temporários	Objetos e estruturas de dados que precisam persistir após o término de uma função
Gerenciamento	Automático: a memória é liberada quando a função termina	Manual ou automático via Garbage Collector

Fragmentação	Não sofre fragmentação	Pode sofrer fragmentação interna e externa
Tempo de vida dos dados	Limitado ao escopo da função	Permanece enquanto houver referência ativa
Tamanho da memória	Limitado e fixo (definido em tempo de compilação)	Amplo e variável (depende da disponibilidade de memória)
Localidade de cache	Alta	Baixa – os dados podem estar espalhados na memória
Risco comum	Stack Overflow (estouro da pilha)	Vazamento de memória (Memory Leak)
Vantagem principal	Rapidez e previsibilidade	Flexibilidade e persistência
Desvantagem principal	Tamanho limitado e inflexível	Gerenciamento mais lento e complexo

6. REFERÊNCIAS BIBLIOGRÁFICAS

CREASY, R. J. The origin of the VM/370 time-sharing system.1981. Disponível em: https://doi.org/10.1147/rd.255.0483. Acesso em: 12 set. 2025

UNIVERSIDADE FEDERAL DO PARANÁ. Alocação de memória estática e dinâmica. Curitiba: UFPR, 2018. Disponível em: https://www.inf.ufpr.br/andrey/ci067/alocacao_memoria.pdf Acesso em: 16 set. 2025.

UNIVERSIDADE FEDERAL DO CEARÁ. Alocação dinâmica de memória em C. Fortaleza: UFC, 2019. Disponível em: https://www.lia.ufc.br/~valeriab/disciplinas/ED/aulas/alocacao-dinamica.pdf . Acesso em: 16 set. 2025.

UNIVERSIDADE ESTADUAL DE CAMPINAS. Alocação dinâmica de memória. Campinas: UNICAMP, 2020. Disponível em: https://ic.unicamp.br/~ra100721/mc102/aulas/alocacao-dinamica.html. Acesso em: 16 set. 2025.

UNIVERSIDADE DE SÃO PAULO. Funções malloc(), calloc(), realloc() e free() na linguagem C. São Carlos: USP, 2015. Disponível em: https://www.icmc.usp.br/pessoas/andre/cursos/2015/cc/ponteiros4.html . Acesso em: 16 set. 2025.

TANENBAUM, Andrew S.; BOS, Herbert. Sistemas Operacionais Modernos. 4. ed. São Paulo: Pearson, 2016. Disponível em: https://books.google.com.br/books?id=KxqCDwAAQBAJ . Acesso em: 16 set. 2025.

BECKENKAMP, Gerson Miguel. Gerenciamento de memória em linguagens de programação. Universidade de Santa Cruz do Sul – UNISC, 2004. Disponível em: https://repositorio.unisc.br/jspui/bitstream/11624/1034/1/Gerson%20Miguel%20Beckenkamp .pdf. Acesso em: 20 set. 2025.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL. Gerência de memória. Porto Alegre: UFRGS, 2011. Disponível em: https://lume.ufrgs.br/bitstream/handle/10183/36924/000819136.pdf . Acesso em: 20 set. 2025.

UNIVERSIDADE FEDERAL DE SÃO CARLOS. Alocação dinâmica de memória. São Carlos: UFSCar, 2015. Disponível em: http://livresaber.sead.ufscar.br:8080/jspui/bitstream/123456789/1505/1/alocacao_dinamica.pdf . Acesso em: 20 set. 2025.