Universidade Cesumar do Paraná

Engenharia de Software

Guilherme Henrique Oliveira da Silva **RA: 24224646-2**

Livia dos Reis Gomides **RA: 24121416-2**

Miguel Mendes **RA: 24475690-2**

Vinicius de Souza Fernandes **RA: 24250759-2**

Gerenciamento de Memória em Sistemas Operacionais

Curitiba/Paraná

Novembro/2025

Guilherme Henrique Oliveira da Silva – 24224646-2

Livia dos Reis Gomides – 24121416-2

Miguel Mendes – 24475690-2

Vinicius de Souza Fernandes – 24250759-2

Gerenciamento de Memória em Sistemas Operacionais

Trabalho Bimestral de Gerenciamento de Memória em Sistemas Operacionais apresentado como requisito para obtenção da nota referente a matéria Sistemas Operacionais do curso superior de Engenharia de Software

Orientador(a): José Carlos Domingues Flores

Curitiba/Paraná

Novembro/2025

Lista de Figuras

Sumário

Questão 1 - Alocação Estática vs. Dinâmica

Alocação estática: A memória é reservada **antes do programa começar a rodar** (no momento da compilação). O tamanho é fixo e não pode mudar.

Exemplo: `int numeros[5];`

Vantagens:

- É simples de usar.
- Acesso rápido (memória da pilha).

Desvantagens:

- O tamanho é fixo, não pode crescer.
- Pode causar erro se usar muita memória (Stack Overflow).

Alocação dinâmica:

A memória é reservada **durante a execução do programa** (em tempo real).

Você escolhe quanto quer e pode liberar depois.

Exemplo: `int *numeros = (int *) malloc(10 * sizeof(int));`

Vantagens:

- Pode mudar o tamanho conforme a necessidade.
- Permite trabalhar com grandes quantidades de dados.

Desvantagens:

- O programador precisa liberar a memória com `free()`.
- É um pouco mais lenta e pode causar vazamento de memória se não for bem usada.

Código em C:

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    // 1. Declare um array estático de 5 inteiros e preencha-o com valores de 1 a 5
    int estatico[5] = {1, 2, 3, 4, 5};

    // 2. Aloque dinamicamente um array de 10 inteiros usando malloc
    int n = 10;
    int *dinamico = (int *) malloc(n * sizeof(int));

    // Verifica se a alocação deu certo
    if (dinamico == NULL) {
        printf("Insufficient Memory.\n");
        return 1;
    }

    // 3. Preencha o array dinâmico com valores de 10 a 19
    for (int i = 0; i < n; i++) {
        dinamico[i] = 10 + i;
    }

    // 4. Imprima os endereços de memória de ambos os arrays
    printf("\n------------------Array Estático------------------\n");
    for (int i = 0; i < 5; i++) {
        printf("Estático[%d] = %d\t(Memory Address: %p)\n", i, estatico[i], (void*)&estatico[i]);
    }

    // Exibe valores e endereços do array dinâmico
    printf("\n------------------Array Estático------------------\n");
    for (int i = 0; i < n; i++) {
        printf("Dinâmico[%d] = %d\t(Memory Address: %p)\n", i, dinamico[i], (void*)&dinamico[i]);
    }

    // 5. Calcule e exiba a diferença entre os endereços (para demonstrar que estão em áreas diferentes da memória)
    // (Usando conversão para int apenas para simplificar)
    int diferenca = (int)((long)&dinamico[0] - (long)&estatico[0]);

    printf("\nBase Address Estático: %p\n", (void*)&estatico[0]);
    printf("Base Address Dinâmico: %p\n", (void*)&dinamico[0]);
    printf("Difference between addresses (bytes): %d\n", diferenca);

    printf("\nObservação: Valores muito diferentes indicam que as variáveis estão armazenadas em áreas distintas da memória.\n");

    // 6. Libere a memória alocada dinamicamente
    free(dinamico);
    dinamico = NULL;

    printf("\nMemória dinâmica liberada com sucesso.\n");

    return 0;
}
```

Resultado esperado:

```
------------------Array Estático------------------
Estático[0] = 1  (Memory Address: 0x7ffcfc7efe90)
Estático[1] = 2  (Memory Address: 0x7ffcfc7efe94)
Estático[2] = 3  (Memory Address: 0x7ffcfc7efe98)
Estático[3] = 4  (Memory Address: 0x7ffcfc7efe9c)
Estático[4] = 5  (Memory Address: 0x7ffcfc7efea0)

------------------Array Estático------------------
Dinâmico[0] = 10         (Memory Address: 0x5b3fd983e910)
Dinâmico[1] = 11         (Memory Address: 0x5b3fd983e914)
Dinâmico[2] = 12         (Memory Address: 0x5b3fd983e918)
Dinâmico[3] = 13         (Memory Address: 0x5b3fd983e91c)
Dinâmico[4] = 14         (Memory Address: 0x5b3fd983e920)
Dinâmico[5] = 15         (Memory Address: 0x5b3fd983e924)
Dinâmico[6] = 16         (Memory Address: 0x5b3fd983e928)
Dinâmico[7] = 17         (Memory Address: 0x5b3fd983e92c)
Dinâmico[8] = 18         (Memory Address: 0x5b3fd983e930)
Dinâmico[9] = 19         (Memory Address: 0x5b3fd983e934)

Base Address Estático: 0x7ffcfc7efe90
Base Address Dinâmico: 0x5b3fd983e910
Difference between addresses (bytes): -586880384

Observação: Valores muito diferentes indicam que as variáveis estão armazenadas em áreas distintas da memória.

Memória dinâmica liberada com sucesso.
```

Questão 2 - Simulação de Fragmentação de Memória

Existem diferentes tipos de divisão de memória, utilizadas pelo sistema para proporcionar informações lógicas sobre o endereço de determinado pacote de dados e alocar de forma correta, garantindo a velocidade e integridade das informações. Permitindo assim que múltiplas tarefas possam ser executadas em paralelo e que os processos nela presentes tenham o devido tempo de execução e espaço, sem desperdício.

Primeiramente podemos falar sobre a Alocação Paginada, que trabalha com a criação de "Quadros" dentro de cada página, onde cada página possui um tamanho fixo de Bytes especificados para a realização de um processo, evitando assim o desperdício de memória.

Outra forma muito utilizada é a Alocação Contígua, que diz respeito a "reserva" de um espaço com determinado tamanho para o cumprimento de um processo, porém no momento em que os "visitantes" saírem, podemos entender como o final da execução de um processo, esse espaço ficará livre, porém não necessariamente será alocado para uma mesma quantidade de pessoas, tornando assim a partição com lacunas e "desperdício" de memória

Agora falando sobre Fragmentação Externa, Carlos Maziero diz que a fragmentação externa se dá pela existência de lacunas entre as áreas físicas da memória, deixando assim espaços livres e/ou vazios entre os processos. Conforme explicado anteriormente, isso só pode se dar em estruturar de alocação e gerenciamento com tamanhos variáveis.

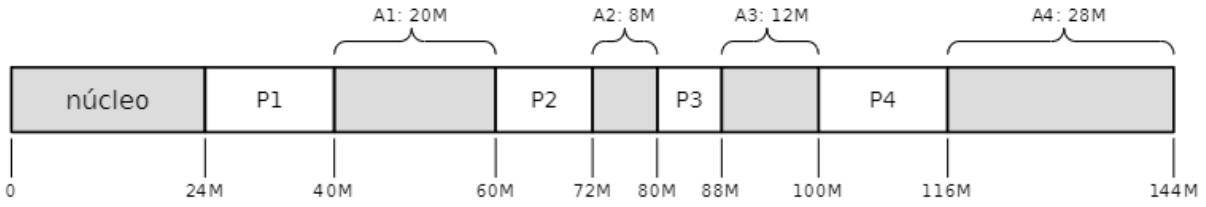

Figura 1 - Memória com Fragmentação Externa

O maior problema gerado pela fragmentação externa é o mal gerenciamento de recursos e maior dificuldade para o alocamento de processos, conforme vemos na imagem acima, onde o tamanho máximo que um processo poderia ser é 28, sendo que realizando a soma de todos esses espaços, o valor dobraria.

Existem formas de diminuir a perda de espaço, com estratégias de alocação, onde o sistema enxerga um espaço disponível e se pergunta, qual a opção menos pior a se seguir? Existem opções como a “Melhor Encaixe” e “Primeiro Encaixe”, onde respectivamente se destacam pela precisão e pela capacidade de ajustar esses espaços, escolhendo sempre a partição com menor tamanho possível e a outra com a rapidez, proporcionando a escolha de partição sempre na primeira opção disponível.

A fragmentação interna é o contrário da primeira, se uma sofre pelo excesso de memória e sobras que geram lacunas no sistema a fragmentação interna afeta o espaço alocado que sobra dentro de uma partição com tamanho fixo, gerando assim um fragmento residual. Isso pode ocorrer quando um processo requisita um número X de bytes em forma paginada por exemplo, porém para suprir essa demanda de tamanho é necessário entregar memória extra, gerando assim maior custo de gestão para as tabelas com páginas e quadros.

Código

```
0 referências
internal class Program
{
    0 referências
    static void Main(string[] args)
    {
        // Cria a memória com partições fixas
        Memoria memoria = new Memoria(new int[] { 100, 150, 200, 250, 300 });

        // Cria os processos
        Particao P1 = new Particao(90, "P1", false);
        Particao P2 = new Particao(140, "P2", false);
        Particao P3 = new Particao(180, "P3", false);
        Particao P4 = new Particao(100, "P4", false);
        Particao P5 = new Particao(350, "P5", false);

        // Executa a sequência de testes
        memoria.AlocarProcesso(P1);
        memoria.AlocarProcesso(P2);
        memoria.AlocarProcesso(P3);
        memoria.liberarProcesso("P2");
        memoria.AlocarProcesso(P4);
        memoria.AlocarProcesso(P5);

        // Exibe o estado final da memória
        memoria.exibir_memoria();
    }
}
```

```
3 referências
public class Memoria
{
    //Realiza a criação de uma lista de partições
    private List<Bloco> blocos = new List<Bloco>();


    1 referência
    public Memoria(int[] tamanhos)
    {
        for (int i = 0; i < tamanhos.Length; i++)
        {
            blocos.Add(new Bloco("A" + (i + 1), tamanhos[i]));
        }
    }


    public void liberarProcesso(String NomePart)
    {
        foreach (var bloco in blocos)
        {
            if(!bloco.Livre && bloco.Processo.nome==NomePart)
            {
                bloco.Livre = true;
                bloco.Processo = null;
                bloco.Fragmentacao = 0;
                Console.WriteLine($"Processo{NomePart} liberado de {bloco.Nome}");
                return;
            }
        }

    }
```

```
public void AlocarProcesso(Particao part)
{
    foreach (var bloco in blocos)
    {
        if (bloco.Livre && bloco.Tamanho >= part.tamanho)
        {
            bloco.Livre = false;
            bloco.Processo = part;
            bloco.Fragmentacao = bloco.Tamanho - part.tamanho;

            Console.WriteLine($"{part.nome} alocado em {bloco.Nome} (tamanho {bloco.Tamanho}). " +
                              $"Fragmentação interna: {bloco.Fragmentacao} unidades.");
            return;
        }
    }

    Console.WriteLine($"Não há partição livre suficiente para {part.nome} ({part.tamanho} unidades).");
}

1 referência
public void exibir_memoria()
{
    Console.WriteLine("\n=== ESTADO ATUAL DA MEMÓRIA ===");
    int totalFragmentacao = 0;

    foreach (var bloco in blocos)
    {
        if (bloco.Livre)
        {
            Console.WriteLine($"{bloco.Nome} ({bloco.Tamanho} unidades): LIVRE");
        }
        else
        {
            Console.WriteLine($"{bloco.Nome} ({bloco.Tamanho} unidades): " +
                              $"{bloco.Processo.nome} ({bloco.Processo.tamanho}) - sobra {bloco.Fragmentacao}");
            totalFragmentacao += bloco.Fragmentacao;
        }
    }

    Console.WriteLine($"\n Fragmentação interna total: {totalFragmentacao} unidades.\n");
}
```

Execução

```
Console de Depuração do Mic
P1 alocado em A1 (tamanho 100). Fragmentação interna: 10 unidades.
P2 alocado em A2 (tamanho 150). Fragmentação interna: 10 unidades.
P3 alocado em A3 (tamanho 200). Fragmentação interna: 20 unidades.
ProcessoP2 liberado de A2
P4 alocado em A2 (tamanho 150). Fragmentação interna: 50 unidades.
Não há partição livre suficiente para P5 (350 unidades).

=== ESTADO ATUAL DA MEMÓRIA ===
A1 (100 unidades): P1 (90) - sobra 10
A2 (150 unidades): P4 (100) - sobra 50
A3 (200 unidades): P3 (180) - sobra 20
A4 (250 unidades): LIVRE
A5 (300 unidades): LIVRE

 Fragmentação interna total: 80 unidades.


C:\Users\Guina\source\repos\GerenciadorMemoria\GerenciadorMemoria\bin\Debug\GerenciadorMemoria.exe (processo 32020) encerrado com o código 0 (0x0).
Para fechar o console automaticamente quando a depuração parar, habilite Ferramentas -> Opções -> Depuração -> Fechar o console automaticamente quando a depuração parar.
Pressione qualquer tecla para fechar esta janela...
```

Questão 3 - Algoritmo de Substituição de Página - FIFO

Também conhecida como Fila, FIFO é uma das mais importantes estruturas de dados, da computação, representa uma das formas que os elementos podem ser processados. Assim como no nosso dia a dia quando vemos pessoas em filas de mercados, bancos e ônibus, todos em fila indiana. Na computação, existe uma estrutura que funciona exatamente da mesma forma: a fila FIFO, que significa First In, First Out, onde como o próprio nome já sugere, o primeiro a entrar é o primeiro a sair.

Ela é importante porque garante que as tarefas e os dados sejam processados na mesma ordem em que chegam, evitando que informações mais novas "furem a fila" e causem erros, atrasos ou perda de sequência.

Esse tipo de controle é essencial em sistemas que lidam com várias requisições ao mesmo tempo.

A fila é uma estrutura de dados linear usada para armazenar elementos em uma sequência ordenada.

Na fila, existem algumas operações principais:

- Enqueue: adiciona um novo elemento no final da fila, mantendo a ordem de chegada.
- Dequeue: remove o primeiro elemento, que está no início da fila.
- Front (ou Peek): permite ver o primeiro elemento sem removê-lo.
- Verificar se está vazia: usada para saber se ainda há elementos na fila.

Essas operações costumam acontecer em tempo constante, o que significa que são bem rápidas e eficientes.

A fila pode ser implementada de diferentes formas, como usando arrays circulares ou listas encadeadas, mas o funcionamento lógico continua o mesmo: entra pelo fim, sai pelo início.

Quando uma fila é criada, o sistema reserva um espaço na memória para armazenar os dados.

Ao adicionar o primeiro elemento, ele se torna o início e o fim da fila ao mesmo tempo.

Cada novo item inserido é colocado no final, e quando um elemento é removido, ele sai do início, mantendo o princípio FIFO.

Isso garante que os dados sejam processados exatamente na ordem em que chegaram, sem confusão ou sobreposição. A fila é considerada um Tipo Abstrato de Dado (TAD), porque define o que pode ser feito com ela, sem precisar mostrar como isso é feito.

Além de ser usada para armazenar dados, a fila também é muito importante dentro dos sistemas operacionais, especialmente no escalonamento de processos.

Quando vários programas estão prontos para serem executados, o sistema operacional precisa decidir qual processo vai usar o processador (CPU) primeiro.

Nesse caso, ele usa uma fila de prontos, que segue exatamente o modelo FIFO.

Funciona assim:

- Quando um processo fica pronto, ele é colocado no final da fila.
- O primeiro da fila é o próximo a ser executado.
- O processo continua rodando até que libere o processador, fique bloqueado ou termine a execução.

O modelo FIFO apresenta várias vantagens importantes. Ele é considerado justo, pois todos os processos ou dados são atendidos na ordem em que chegam, sem privilégios. Também é eficiente em sistemas contínuos, como buffers e redes, já que mantém o fluxo de informações organizado e evita falhas. Além disso, impede a fome (starvation), garantindo que nenhum processo fique esperando indefinidamente, e tem baixo custo de gerenciamento, pois as operações de inserção e remoção são rápidas e leves para o sistema. Apesar de ser simples e justo, o FIFO tem algumas limitações. Ele pode prejudicar processos I/O-bound, que passam mais tempo esperando operações de entrada e saída do que usando a CPU. Isso acontece porque, se um processo muito longo estiver no início da fila, todos os outros precisam esperar, mesmo que sejam rápidos. Esse efeito é conhecido como “efeito comboio” (convoy effect) e pode diminuir o desempenho do sistema.

Além disso, o FIFO não controla bem o tempo de resposta, o que pode causar filas grandes e processos demorados.

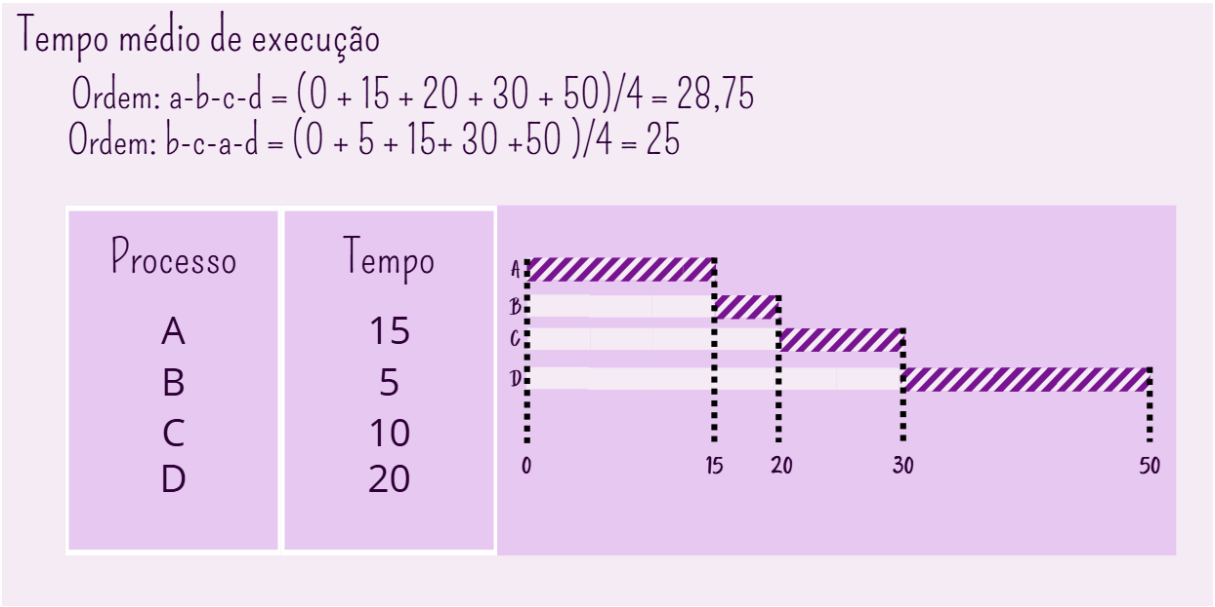

Figura 2 - Figura Exibindo Tempo Médio de Execução

O modelo FIFO é muito utilizado em situações onde é essencial manter a ordem de chegada dos dados, como em filas de impressão, buffers de rede, escalonamento simples de processos e sistemas de atendimento ou mensagens, garantindo organização e previsibilidade. Porém, ele não é indicado para ambientes que exigem prioridade ou respostas rápidas, como sistemas de tempo real, tarefas críticas ou com tempos de execução muito diferentes, pois pode causar atrasos e o chamado “efeito comboio”. Assim, o FIFO funciona muito bem em contextos simples e ordenados, mas não é a melhor escolha para sistemas que precisam de maior flexibilidade ou controle sobre a prioridade das tarefas.

Além dessas aplicações, o conceito de FIFO também é usado dentro do gerenciamento de memória dos sistemas operacionais, por meio do algoritmo de substituição de página FIFO. Ele segue o mesmo princípio, quando a memória está cheia e o sistema precisa abrir espaço para uma nova página, a que está há mais tempo na memória é a primeira a sair.

O sistema mantém uma fila com as páginas carregadas, e toda vez que ocorre uma falta de página (page fault), a mais antiga é removida para dar lugar à nova, que entra no final da fila. Esse método é simples e previsível, o que facilita sua implementação e garante uma certa justiça no gerenciamento da memória.

Porém, o FIFO não analisa quais páginas estão sendo mais usadas. Fazendo com que uma página ainda necessária possa ser retirada apenas por ser mais antiga, o que pode causar lentidão e aumentar o número de trocas de página. Esse problema é conhecido como “efeito de Belady”, que ocorre quando, mesmo aumentando a memória, o desempenho do sistema não melhora, podendo até piorar.

No contexto do gerenciamento de memória, as páginas são como pequenos pedaços de um programa que o sistema operacional carrega na memória RAM para poder funcionar. Em vez de colocar tudo de uma vez , pois seria muito pesado, o sistema divide o programa em partes menores, chamadas páginas. Cada uma delas ocupa um espaço na memória chamado frame. Quando o processador precisa de alguma informação, ele procura a página correspondente na memória: se ela já estiver lá, é um acerto (page hit); se não estiver, acontece uma falta de página (page fault), e o sistema precisa buscá-la no disco. Mas, se a memória já estiver cheia, o sistema precisa decidir qual página vai sair para dar lugar à nova. O algoritmo FIFO faz isso de maneira bem simples: a primeira página que entrou é a primeira que sai.

Código

```java
package FIFO;

import java.util.ArrayList; // usado para armazenar as páginas na memória (estrutura de lista dinâmica)
import java.util.Scanner; // usado para ler as entradas digitadas pelo usuário

public class ProgramaFIFO {

    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);  // cria o leitor de entrada do usuário
        ArrayList<Integer> memoria = new ArrayList<>();  // lista que simula os quadros de memória
        int numeroFrames = 0;   // quantidade de frames disponíveis na memória
        int totalFaltas = 0;    // contador de faltas de página (page faults)

        System.out.println(" Substituição de Página FIFO \n");

        // Entrada e validação do número de frames
        while (true) {
            try {
                System.out.print("Digite o número de frames disponíveis (máximo 5 caracteres): ");
                String entrada = input.nextLine(); // lê o texto digitado

                // Verifica se o texto tem mais de 5 caracteres
                if (entrada.length() > 5) {
                    System.out.println("Erro: número de caracteres excedido, máximo 5 caracteres. Tente novamente.\n");
                    continue;
                }

                // Converte o valor digitado para inteiro
                numeroFrames = Integer.parseInt(entrada);

                // Garante que o número seja maior que zero
                if (numeroFrames <= 0) {
                    System.out.println("Erro: o número deve ser maior que zero.\n");
                    continue;
                }

                // Se passou por todas as verificações, sai do loop
                break;
```

```java
            // Se a página já estiver na memória → acerto
            if (memoria.contains(pagina)) {
                resultado = "Acerto (Page Hit)";
            } else {
                // Se não estiver → falta de página
                totalFaltas++;

                // Se ainda houver espaço na memória, adiciona a nova página
                if (memoria.size() < numeroFrames) {
                    memoria.add(pagina);
                    resultado = "Falta (Page Fault)";
                } else {
                    // Se estiver cheia, remove a mais antiga (primeira da fila)
                    int removida = memoria.remove(0);
                    memoria.add(pagina);
                    resultado = "Falta (substituiu " + removida + ")";
                }
            }

            // Exibe o estado da memória e o resultado atual
            System.out.printf("%-12d | %-25s | %-25s | %-15d\n",
```

```
40
41          } catch (NumberFormatException e) {
42              // Mensagem exibida quando o usuário digita algo que não é número
43              System.out.println("Erro: Número inválido.\n");
44          }
45      }
46
47      // Entrada da sequência de páginas
48      System.out.println("\nDigite as referências de página (separe os números por espaço, ex: 7 0 1 2 0 3):");
49      System.out.print("→ ");
50      String[] paginasTexto = input.nextLine().trim().split("\\s+"); // separa os números digitados pelos espaços
51
52      // Validação do tamanho de cada número digitado
53      for (String ref : paginasTexto) {
54          if (ref.length() > 5) {
55              System.out.println("Erro: número de caracteres excedido, máximo 5 caracteres. Tente novamente.");
56              input.close();
57              return;
58          }
59      }
60
61      // Conversão das páginas digitadas (texto) para números inteiros
62      int[] paginas = new int[paginasTexto.length];
63      try {
64          for (int i = 0; i < paginasTexto.length; i++) {
65              paginas[i] = Integer.parseInt(paginasTexto[i]);
66          }
67      } catch (NumberFormatException e) {
68          System.out.println("Erro: sequência de páginas inválida. Use apenas números.");
69          input.close();
70          return;
71      }
72
73      // Início da simulação FIFO
74      System.out.println("\n===== Iniciando Simulação FIFO =====\n");
75      // Cabeçalho da tabela
76      System.out.printf("%-12s | %-25s | %-25s | %-15s\n", "Referência", "Frames", "Resultado", "Total Faltas");
77      System.out.println("-----------------------------------------------------------------------------------------");
78
79      // Percorre cada página referenciada
80      for (int pagina : paginas) {
81          String resultado; // armazena o que aconteceu: acerto ou falta
```

```
Problems  @ Javadoc  Declaration  Console  Eclipse IDE for Eclipse Committers 2025-09 Release
<terminated> ProgramaFIFO [Java Application] C:\Users\livia\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_
 Substituição de Página FIFO

Digite o número de frames disponíveis (máximo 5 caracteres): 4

Digite as referências de página (separe os números por espaço, ex: 7 0 1 2 0 3):
→ 2 4 5 2 4 2 34

===== Iniciando Simulação FIFO =====

Referência   | Frames                    | Resultado                 | Total Faltas
-------------------------------------------------------------------------------------
2            | [2]                       | Falta (Page Fault)        | 1
4            | [2, 4]                    | Falta (Page Fault)        | 2
5            | [2, 4, 5]                 | Falta (Page Fault)        | 3
2            | [2, 4, 5]                 | Acerto (Page Hit)         | 3
4            | [2, 4, 5]                 | Acerto (Page Hit)         | 3
2            | [2, 4, 5]                 | Acerto (Page Hit)         | 3
34           | [2, 4, 5, 34]             | Falta (Page Fault)        | 4
-------------------------------------------------------------------------------------
Total de referências: 7
Total de faltas de página: 4
Taxa de faltas de página: 57,14%

Liberando memória da fila FIFO...
Memória liberada com sucesso
```

```
Problems  @ Javadoc  Declaration  Console  Eclipse IDE for Eclipse Committers 2025-09
ProgramaFIFO [Java Application] C:\Users\livia\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32
 Substituição de Página FIFO

Digite o número de frames disponíveis (máximo 5 caracteres): 374930
Erro: número de caracteres excedido, máximo 5 caracteres. Tente novamente.

Digite o número de frames disponíveis (máximo 5 caracteres): S
```

Questão 4 - Garbage Collection em Python

O mecanismo de gerenciamento de memória do Python é uma arquitetura híbrida, primariamente impulsionada pela Contagem de Referências (Reference Counting). Este é um sistema determinístico onde cada objeto mantém um contador de quantas referências ativas apontam para ele. Quando o contador de um objeto é decrementado a zero (seja por del, fim de escopo ou reatribuição de variável), seu método __del__ é invocado e a memória é desalocada imediatamente.

Embora determinística e rápida para a maioria dos casos, a Contagem de Referências possui uma falha inerente: ela é incapaz de gerenciar referências circulares. Nesses cenários, dois ou mais objetos referenciam-se mutuamente, garantindo que seus contadores de referência nunca cheguem a zero, mesmo que o grupo de objetos se torne inacessível pelo resto do programa, resultando em vazamento de memória.

Para solucionar isso, o Python complementa esse mecanismo com um Coletor de Lixo Geracional (Generational Garbage Collector). Este é um coletor de rastreamento (tracing collector) que utiliza um algoritmo Mark-and-Sweep (Marcar e Varrer) projetado especificamente para detectar e quebrar esses ciclos. Ele identifica "ilhas" de objetos que são internamente referenciados, mas inacessíveis a partir das "raízes" (roots) do programa (como o escopo global ou a pilha de execução). Como uma otimização de performance, ele emprega uma heurística geracional, dividindo os objetos em três gerações (0, 1 e 2) com base em sua longevidade. O coletor foca suas varreduras mais frequentes na Geração 0 (objetos novos), partindo do princípio de que a maioria dos objetos morre jovem, diminuindo assim a latência (pauses) associada a uma varredura completa da memória.

UniCesumar

Código Completo

```
import gc
import sys

# Classe Objeto para simular uso de memória.
# __del__ avisa quando o objeto é destruído.
class Objeto:
    def __init__(self, nome):
        self.nome = nome
        self.dados = [0] * 100000
        self.ref = None
        print(f"Objeto {self.nome} criado.")

    def __del__(self):
        print(f"Objeto {self.nome} destruído (memória liberada).")

# --- Cenário 1: Contagem de Referências (Destruição Imediata) ---
print("\n=== CENÁRIO 1: DESTRUIÇÃO AUTOMÁTICA POR CONTAGEM DE REFERÊNCIAS ===")
obj1 = Objeto("A")

# sys.getrefcount(obj): Retorna o número de referências ativas para o objeto.
print(f"Referências ativas de obj1: {sys.getrefcount(obj1)}")
del obj1  # A contagem zera, __del__ é chamado imediatamente.

# --- Cenário 2: Referências Circulares (O Problema) ---
# O ciclo (B <-> C) impede a contagem de zerar.
# Os objetos NÃO são destruídos pelo 'del' (vazamento de memória).
print("\n=== CENÁRIO 2: REFERÊNCIAS CIRCULARES ===")
obj2 = Objeto("B")
obj3 = Objeto("C")
obj2.ref = obj3
obj3.ref = obj2
del obj2
del obj3
print("Objetos B e C não destruídos imediatamente (referência circular impede GC automático).")
# --- Coleta Forçada (A Solução) ---
print("\nForçando coleta de lixo...")

# gc.collect(): Força o coletor geracional (inteligente) a rodar.
# Ele é capaz de encontrar e destruir ciclos inacessíveis.
gc.collect()

# --- Cenário 3: Coleta em Massa e Estatísticas ---
print("\n=== CENÁRIO 3: COLETOR GERACIONAL ===")
objetos = [Objeto(f"Temp_{i}") for i in range(5)]
print(f"Referências do primeiro objeto: {sys.getrefcount(objetos[0])}")
objetos.clear() # O 'clear' zera a contagem e já destrói os objetos 'Temp'
coletados = gc.collect()
print(f"GC coletou {coletados} objetos órfãos.")

print("\nEstatísticas do coletor geracional:")

# gc.get_stats(): Retorna o relatório de quantas coletas/objetos
# foram limpos em cada uma das 3 "gerações" de objetos.
for i, stats in enumerate(gc.get_stats()):
    print(f"Geração {i}: {stats}")

print("\nDemonstração concluída.")S
```

Resultado

```
=== CENÁRIO 1: DESTRUIÇÃO AUTOMÁTICA POR CONTAGEM DE REFERÊNCIAS ===
Objeto A criado.
Referências ativas de obj1: 2
Objeto A destruído (memória liberada).

=== CENÁRIO 2: REFERÊNCIAS CIRCULARES ===
Objeto B criado.
Objeto C criado.
Objetos B e C não destruídos imediatamente (referência circular impede GC automático).

Forçando coleta de lixo...
Objeto B destruído (memória liberada).
Objeto C destruído (memória liberada).

=== CENÁRIO 3: COLETOR GERACIONAL ===
Objeto Temp_0 criado.
Objeto Temp_1 criado.
Objeto Temp_2 criado.
Objeto Temp_3 criado.
Objeto Temp_4 criado.
Referências do primeiro objeto: 2
Objeto Temp_4 destruído (memória liberada).
Objeto Temp_3 destruído (memória liberada).
Objeto Temp_2 destruído (memória liberada).
Objeto Temp_1 destruído (memória liberada).
Objeto Temp_0 destruído (memória liberada).
GC coletou 0 objetos órfãos.

Estatísticas do coletor geracional:
Geração 0: {'collections': 3, 'collected': 25, 'uncollectable': 0}
Geração 1: {'collections': 0, 'collected': 0, 'uncollectable': 0}
Geração 2: {'collections': 2, 'collected': 4, 'uncollectable': 0}

Demonstração concluída.

Process finished with exit code 0
```

Questão 5 - Comparação de Desempenho de Alocação

A alocação de memória a pilha(*Stack*) é uma função reconhecidamente mais rápida que a alocação em *heap* por conta da forma que cada uma dessas regiões é gerenciada. Podemos entender a pilha como uma estrutura contínua de memória pois é organizada de maneira sequencial, onde a alocação e a desalocação ocorrem através do ajuste do ponteiro de pilha (do inglês padrão, *stack pointer*). A partir do momento em que uma função é chamada, o ponteiro incrementado para reservar espaço necessário para as variáveis locais, ao retornar, ele é imediatamente decrementado para liberar toda esta memória. Este mecanismo, além de extremamente simples, executa operações de tempo constante e é diretamente suportado pelo hardware da CPU, o que, por fim, acaba tornando o acesso à pilha altamente eficiente e previsível. Somado a isto, por se tratar de uma contígua e próxima aos registradores e ao cache da CPU, a pilha se beneficia de excelente localidade de referência, reduzindo ainda mais o tempo de acesso.

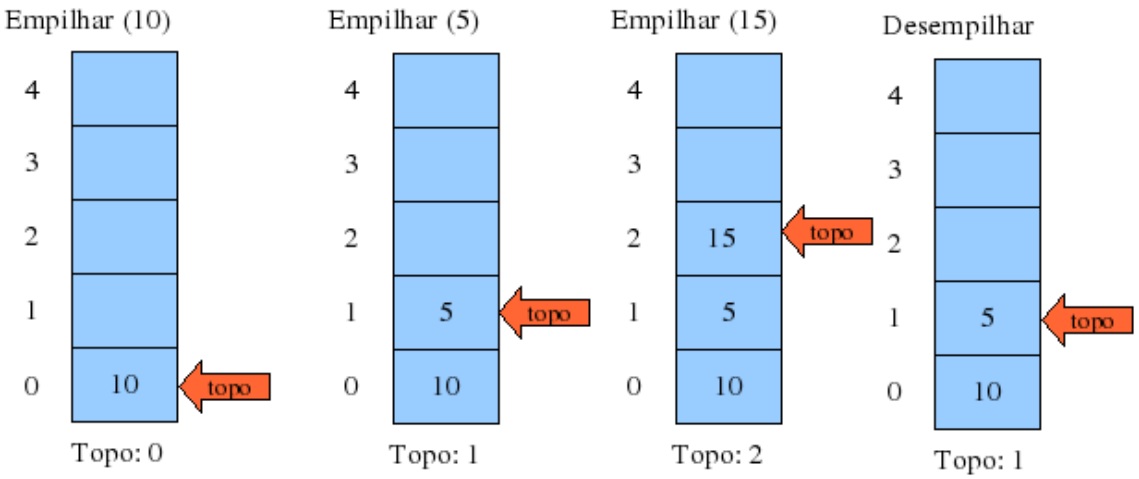

Figura 3 - Figura Exemplificando o Funcionamento da Alocação em Pilha (Stack)

A memória *heap* é gerida de forma dinâmica e mais complexa pois contém blocos de memória que podem estar livres ou ocupados, desta forma, eles acabam por exigir algoritmos para localizar espaços adequados para alocação, controlar metadados e lidar com fragmentação. Uma representação deste fenômeno é a função *malloc()*, ao ser utilizada na linguagem C, ela precisa percorrer listas ou estruturas internas para encontrar um bloco compatível, o que ocasiona um processo mais custoso em termos de tempo e não oferece o desempenho ideal. Em alguns outros casos, a expansão do *heap* pode até demandar chamadas ao sistema operacional, como *mmap()* ou *sbrk()*, que são operações comparativamente lentas. Além disso, por ter uma gerencia manual em linguagens como C e C++, o *heap* está sujeito a problemas como vazamento de memória e uso incorreto de ponteiros, o que não ocorre na pilha.

Código:

```java
public class Main {
    // Número de repetições para tirar média
    private static final int REPETICOES = 10;  4 usages

    // Quantidade de operações feitas em cada teste
    private static final int OPERACOES = 1_000_000;  2 usages

    public static void main(String[] args) {

        long tempoMedioStack = testarStack();
        long tempoMedioHeap = testarHeap();

        System.out.println("======= RESULTADOS =======");
        System.out.println("Média - Alocação na PILHA: " + tempoMedioStack + " ns");
        System.out.println("Média - Alocação no HEAP:  " + tempoMedioHeap + " ns");

        double diferenca = ((double) tempoMedioHeap / tempoMedioStack - 1) * 100;
        System.out.printf("Diferença percentual (Heap vs Stack): %.2f%%\n", diferenca);
    }

    // Teste de alocação na pilha
    private static long testarStack() {  1 usage
        long soma = 0;

        for (int r = 0; r < REPETICOES; r++) {

            long inicio = System.nanoTime();

            for (int i = 0; i < OPERACOES; i++) {
                int x = i; // variável local, vive na stack, criação e destruição são baratas
            }

            long fim = System.nanoTime();
            soma += (fim - inicio);
```

```
        return soma / REPETICOES;
    }

    // Teste de alocação no heap
    private static long testarHeap() {  1 usage
        long soma = 0;

        for (int r = 0; r < REPETICOES; r++) {

            long inicio = System.nanoTime();

            for (int i = 0; i < OPERACOES; i++) {
                new ObjetoExemplo(i);
                // criação no heap; GC decide quando destruir mais tarde
            }

            long fim = System.nanoTime();
            soma += (fim - inicio);

            System.gc(); // pequena ajuda para minimizar interferência
        }

        return soma / REPETICOES;
    }
    // Classe simples para instanciar no heap
    static class ObjetoExemplo {  1 usage
        int valor;  1 usage

        public ObjetoExemplo(int v) {  1 usage
            this.valor = v;
        }
    }
```

Resultado:

```
======= RESULTADOS =======
Média - Alocação na PILHA: 931800 ns
Média - Alocação no HEAP:  3730140 ns
Diferença percentual (Heap vs Stack): 300,32%

Process finished with exit code 0
```