Compiladores

Aula 21: Gerenciamento de Memória e Ambientes de Runtime

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Maio de 2026

Sumário

1. Objetivos
2. O Layout de Memória do Processo
3. A Pilha (Stack) e Registros de Ativação
4. Escopo e Acesso a Variáveis não Locais
5. O Heap e Alocação Dinâmica
6. Coleta de Lixo (Garbage Collection - GC)
7. Resumo e Referências

Objetivos da Aula

- Compreender como o **compilador e o sistema operacional** colaboram para gerenciar a memória de um programa em execução.
- Detalhar o layout de memória de um processo: **Código, Dados, Heap e Stack**.
- Entender o papel da pilha (**Stack**) na implementação de sub-rotinas, recursão e variáveis locais (**Registros de Ativação**).
- Analisar como linguagens com funções aninhadas acessam variáveis não locais (**Links Estáticos**).
- Compreender os desafios do **Heap** (fragmentação) e os algoritmos clássicos de **Coleta de Lixo** (Garbage Collection).

Introdução ao Ambiente de Runtime

- O compilador traduz código-fonte (nomes lógicos) para código de máquina (endereços físicos/virtuais).
- Mas o compilador **não executa** o programa. Quem gerencia o programa durante a execução é o **Ambiente de Runtime** em conjunto com o Sistema Operacional.
- O compilador deve gerar código assumindo um **modelo de memória** padrão.

Introdução ao Ambiente de Runtime

- O compilador traduz código-fonte (nomes lógicos) para código de máquina (endereços físicos/virtuais).
- Mas o compilador **não executa** o programa. Quem gerencia o programa durante a execução é o **Ambiente de Runtime** em conjunto com o Sistema Operacional.
- O compilador deve gerar código assumindo um **modelo de memória** padrão.

O Desafio

Como mapear variáveis locais, variáveis globais, objetos dinâmicos e o próprio código executável de forma eficiente e segura?

O Layout Típico de um Processo

Divisão Lógica:

O Layout Típico de um Processo

Divisão Lógica:

- **Código (Text):** Instruções do programa. (Apenas leitura).

O Layout Típico de um Processo

Divisão Lógica:

- **Código (Text):** Instruções do programa. (Apenas leitura).
- **Dados Estáticos:** Variáveis globais e constantes. Tamanho fixo em tempo de compilação.

O Layout Típico de um Processo

Divisão Lógica:

- **Código (Text):** Instruções do programa. (Apenas leitura).
- **Dados Estáticos:** Variáveis globais e constantes. Tamanho fixo em tempo de compilação.
- **Heap:** Memória alocada **dinamicamente** (ex: `malloc`, `new`). Cresce para cima.

O Layout Típico de um Processo

Divisão Lógica:

- **Código (Text):** Instruções do programa. (Apenas leitura).
- **Dados Estáticos:** Variáveis globais e constantes. Tamanho fixo em tempo de compilação.
- **Heap:** Memória alocada **dinamicamente** (ex: `malloc`, `new`). Cresce para cima.
- **Stack (Pilha):** Variáveis locais e controle de chamadas de função. Cresce para baixo (na maioria das arquiteturas).

Estratégias de Alocação

1. **Alocação Estática:**
   - Resolvida em tempo de compilação.
   - Endereços fixos na seção de Dados Estáticos.
   - Rápido, mas não suporta recursão ou estruturas dinâmicas. (Ex: Fortran antigo).

Estratégias de Alocação

1. **Alocação Estática:**
   - Resolvida em tempo de compilação.
   - Endereços fixos na seção de Dados Estáticos.
   - Rápido, mas não suporta recursão ou estruturas dinâmicas. (Ex: Fortran antigo).
2. **Alocação Baseada em Pilha (Stack):**
   - Gerenciada automaticamente por chamadas/retornos de função.
   - Suporta recursão.
   - Tempo de vida das variáveis está atrelado ao escopo da função.

Estratégias de Alocação

1. **Alocação Estática:**
   - Resolvida em tempo de compilação.
   - Endereços fixos na seção de Dados Estáticos.
   - Rápido, mas não suporta recursão ou estruturas dinâmicas. (Ex: Fortran antigo).
2. **Alocação Baseada em Pilha (Stack):**
   - Gerenciada automaticamente por chamadas/retornos de função.
   - Suporta recursão.
   - Tempo de vida das variáveis está atrelado ao escopo da função.
3. **Alocação Baseada em Heap:**
   - Gerenciada manualmente (C/C++) ou por Garbage Collector (Java, C#, Python).
   - Para dados cujo tamanho não é conhecido na compilação ou que devem sobreviver ao fim da função que os criou.

O Problema das Chamadas de Função

Quando uma função `A` chama uma função `B`:

- A execução de `A` é suspensa.
- Onde guardar o estado de `A` (onde estávamos, valores das variáveis locais)?
- Onde guardar os parâmetros passados para `B`?
- E se `B` chamar `A` novamente (recursão)?

O Problema das Chamadas de Função

Quando uma função `A` chama uma função `B`:

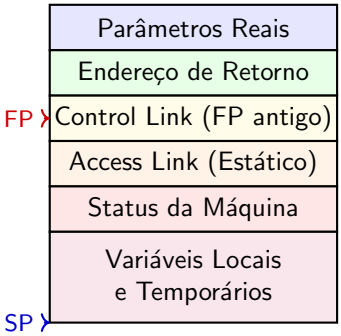
- A execução de `A` é suspensa.
- Onde guardar o estado de `A` (onde estávamos, valores das variáveis locais)?
- Onde guardar os parâmetros passados para `B`?
- E se `B` chamar `A` novamente (recursão)?

Solução: A Pilha de Execução (Call Stack)

Alocamos um bloco de memória, chamado **Registro de Ativação** (Activation Record ou **Stack Frame**), para cada execução de uma função. A estrutura de dados Pilha encaixa perfeitamente pois funções retornam na ordem inversa em que são chamadas (LIFO).

Estrutura do Registro de Ativação (Stack Frame)

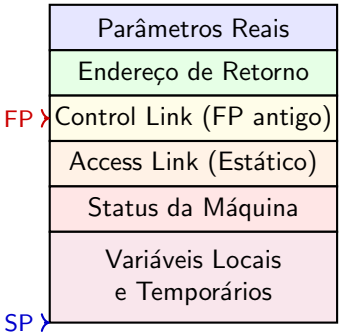
Embora dependa da arquitetura e convenção de chamada (ABI), um frame típico contém:



- **Parâmetros:** Argumentos recebidos do *caller*.

Estrutura do Registro de Ativação (Stack Frame)

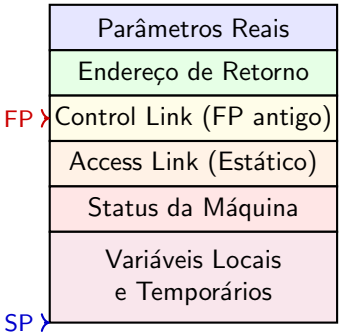
Embora dependa da arquitetura e convenção de chamada (ABI), um frame típico contém:



- **Parâmetros:** Argumentos recebidos do *caller*.
- **End. de Retorno:** Para onde pular quando a função acabar.

Estrutura do Registro de Ativação (Stack Frame)

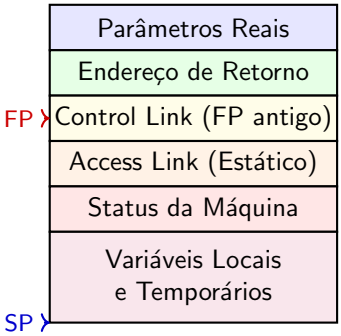
Embora dependa da arquitetura e convenção de chamada (ABI), um frame típico contém:



- **Parâmetros:** Argumentos recebidos do *caller*.
- **End. de Retorno:** Para onde pular quando a função acabar.
- **Control Link:** Ponteiro para o frame da função que chamou (restaura o FP).

Estrutura do Registro de Ativação (Stack Frame)

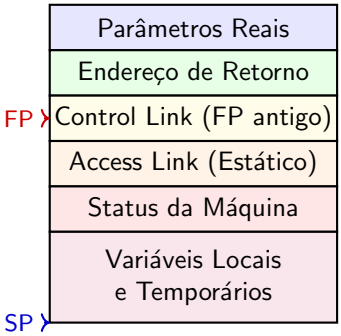
Embora dependa da arquitetura e convenção de chamada (ABI), um frame típico contém:



- **Parâmetros:** Argumentos recebidos do *caller*.
- **End. de Retorno:** Para onde pular quando a função acabar.
- **Control Link:** Ponteiro para o frame da função que chamou (restaura o FP).
- **Status:** Registradores salvos.

Estrutura do Registro de Ativação (Stack Frame)

Embora dependa da arquitetura e convenção de chamada (ABI), um frame típico contém:



- **Parâmetros:** Argumentos recebidos do *caller*.
- **End. de Retorno:** Para onde pular quando a função acabar.
- **Control Link:** Ponteiro para o frame da função que chamou (restaura o FP).
- **Status:** Registradores salvos.
- **Locais/Temporários:** Dados da função.

Por que ter Stack Pointer (SP) e Frame Pointer (FP)?

- O **Stack Pointer (SP)** aponta para o topo atual da pilha. Seu valor **muda** durante a execução da função (ex: se o compilador empilhar argumentos para chamar outra função).

Por que ter Stack Pointer (SP) e Frame Pointer (FP)?

- O **Stack Pointer (SP)** aponta para o topo atual da pilha. Seu valor **muda** durante a execução da função (ex: se o compilador empilhar argumentos para chamar outra função).
- O **Frame Pointer (FP)** (ou Base Pointer - BP) aponta para um local **fixo** no meio do registro de ativação.

Por que ter Stack Pointer (SP) e Frame Pointer (FP)?

- O **Stack Pointer (SP)** aponta para o topo atual da pilha. Seu valor **muda** durante a execução da função (ex: se o compilador empilhar argumentos para chamar outra função).
- O **Frame Pointer (FP)** (ou Base Pointer - BP) aponta para um local **fixo** no meio do registro de ativação.

A Magia do FP

Como o FP não muda, o compilador acessa tudo usando offsets **constantes** em relação a ele!

- Variáveis locais: deslocamento negativo (ex: `FP - 4`)
- Parâmetros: deslocamento positivo (ex: `FP + 8`)

Se usássemos o SP, os offsets mudariam o tempo todo!

Sequência de Chamada (Calling Sequence)

O compilador insere códigos especiais nas extremidades das funções para gerenciar a pilha.

Prologue (Início da Função)

```
// Ao entrar na funcao:
push FP       // Salva o FP antigo (
    Control Link)
FP = SP       // Novo FP e o SP atual
SP = SP - N   // Aloca N bytes p/
    variaveis locais
```

Epilogue (Fim da Função)

```
// Ao sair da funcao:
SP = FP       // Desaloca as
    variaveis locais (SP volta pro FP
    )
FP = pop      // Restaura o FP da
    funcao chamadora
return        // Pula p/ Endereco de
    Retorno
```

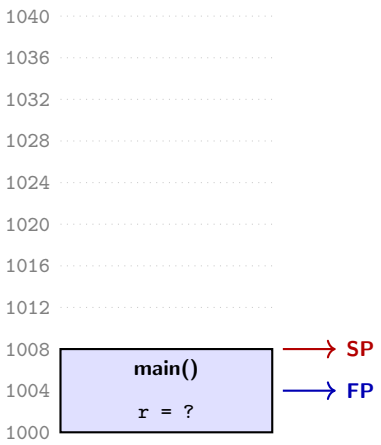
- A divisão exata de responsabilidades (quem empilha argumentos: *caller* ou *callee*?) depende da **Convenção de Chamada** (ex: `cdecl`, `stdcall`, `fastcall`).

Simulação da Pilha: `fat(3)` — Código

```
int fat(int n) {
  if (n <= 1)
    return 1;
  return n * fat(n-1);
}
int main() {
  int r = fat(3);
  return 0;
}
```

Convenção

- Stack **cresce para cima** (endereços maiores no topo)
- Cada célula = 4 bytes
- **SP** = Stack Pointer (topo atual)
- **FP** = Frame Pointer (base do frame atual)



Pilha antes de chamar `fat(3)`

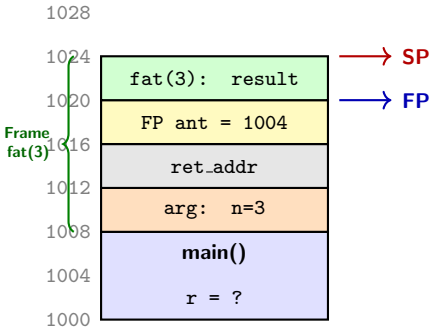
Simulação: Passo 1 — `main` chama `fat(3)`

O que acontece?

1. *Caller* (`main`) empilha o argumento `n = 3`
2. Empilha o endereço de retorno
3. *Callee* (`fat`) executa o **prologue**:
   - Salva o FP antigo (Control Link)
   - `FP = SP`
   - `SP -= 4` (aloca local para retorno parcial)

Registradores agora

`SP = 1020` `FP = 1016`



Simulação: Passo 2 — `fat(3)` chama `fat(2)`

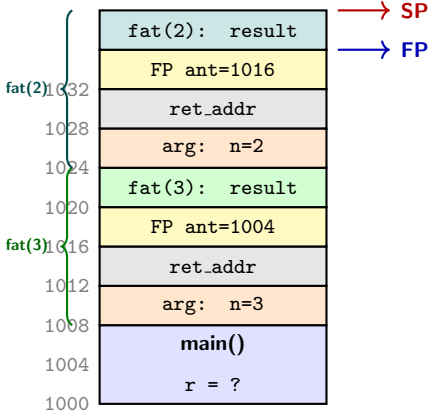
O que acontece?

1. `fat(3)` percebe que `n > 1` — vai chamar `fat(2)`
2. Empilha argumento `n = 2`
3. Empilha endereço de retorno
4. Novo prologue: salva FP de `fat(3)`, cria novo frame

Registradores agora

`SP = 1032  FP = 1028`

A pilha já guarda **duas cópias independentes** de `n`.



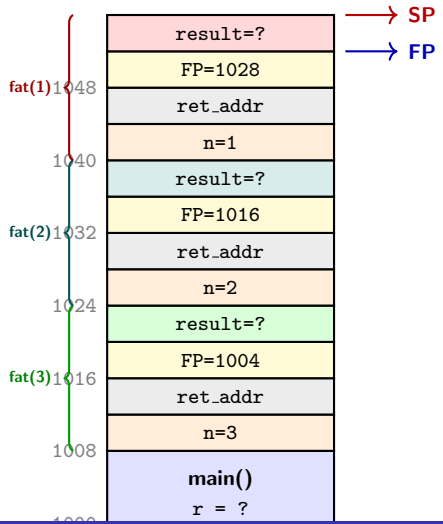
Simulação: Passo 3 — Pico da Pilha (`fat(1)`)

Situação atual

- **3 frames ativos** na pilha simultaneamente
- Cada um tem sua **própria** cópia de `n`
- `fat(1)` detecta `n <= 1` e vai **retornar 1**

Registradores

`SP = 1044  FP = 1040`



Simulação: Passo 4 — `fat(1)` retorna 1

Epilogue de `fat(1)`

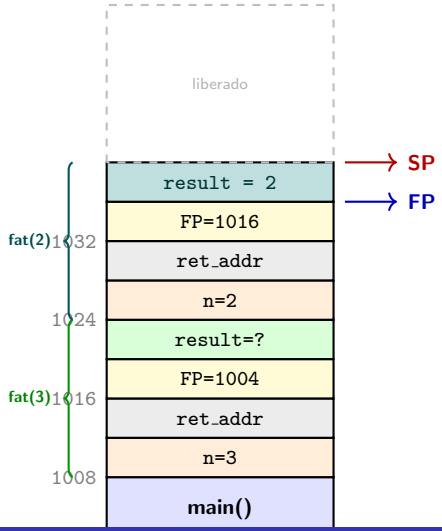
1. Grava `result = 1` no frame
2. `SP = FP` (desaloca locais)
3. `FP = pop` (restaura FP de fat(2))
4. Salta para `ret_addr`

Resultado

Frame de `fat(1)` foi **desempilhado**.
`fat(2)` recebe 1 de volta e calcula `result = 2 * 1 = 2`.

Registradores

`SP = 1032` `FP = 1028`



Simulação: Passo 5 — `fat(2)` retorna 2

Epilogue de `fat(2)`

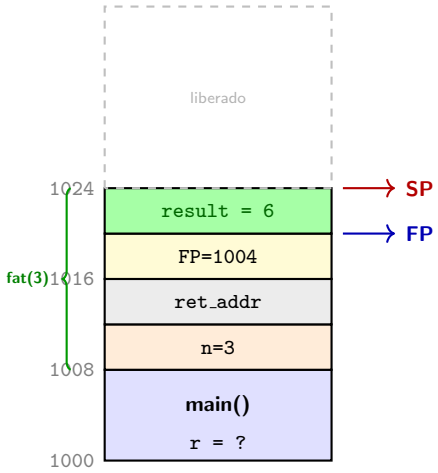
1. `result = 2 * fat(1) = 2 * 1 = 2` ✓
2. Epilogue: `SP = FP`, restaura FP de `fat(3)`
3. Retorna 2 para `fat(3)`

Resultado

`fat(3)` recebe 2 e calcula `result = 3 * 2 = 6`.

Registradores

`SP = 1020` `FP = 1016`



Simulação: Passo 6 — `fat(3)` retorna `6` para `main`

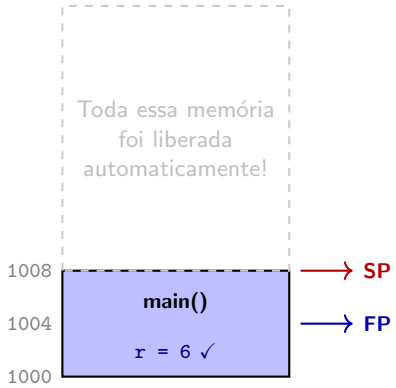
Epilogue de `fat(3)`

1. `result = 3 * fat(2) = 3 * 2 = 6` ✓
2. Epilogue: restaura FP e SP de `main`
3. `main` recebe 6 e grava em `r`

Estado Final

Todos os frames de `fat` foram desempilhados.
Apenas `main` permanece na pilha com `r = 6`.

$$\texttt{fat(3)} = 3 \times 2 \times 1 = \mathbf{6}$$



Funções Aninhadas e Escopo Léxico

Linguagens como C não permitem declarar funções dentro de funções. Mas linguagens como Pascal, ML, Python e JavaScript (Closures) **permitem**.

```
procedure A();
var x: integer;
  procedure B();
  var y: integer;
  begin
    y := x + 1; // x de A!
  end;
begin
  x := 10;
  B();
end;
```

Problema: Em tempo de execução, quando B acessa x, onde x está na pilha?

Não podemos usar um offset fixo em relação ao FP de B, pois B pode ter sido chamada de diversos lugares e a distância na pilha varia.

O Link Estático (Access Link)

- Solução: adicionar um ponteiro extra no Registro de Ativação, chamado **Link Estático** (Static/Access Link).
- O Link Estático aponta para o Registro de Ativação da função que engloba **lexicamente** a função atual no código-fonte.
- O **Control Link (FP antigo)** aponta para quem a chamou no tempo de execução (*dynamic scoping*). O **Static Link** aponta para quem a envolve no código-fonte (*lexical scoping*).

Cálculo do Acesso

Para acessar a variável `x` definida k níveis lexicais acima:

1. Siga o Link Estático k vezes.
2. Adicione o offset conhecido de `x`.

Por que precisamos do Heap?

O Stack é perfeito para variáveis atreladas à vida da função. Mas falha quando:

1. Uma função cria uma estrutura de dados e quer retorná-la (o frame morre, os dados seriam perdidos - *dangling pointer*).
2. O tamanho dos dados não é conhecido nem na entrada da função (ex: carregar um arquivo inteiro, árvores dinâmicas).

O Heap

Uma área de memória global onde blocos são alocados e desalocados em ordem arbitrária.

O Gerenciador de Memória (Memory Manager)

Quando você chama `malloc(size)` ou `new Object()`:

- O Memory Manager do Runtime busca um bloco contíguo de bytes no Heap.
- **Desafio Crítico: Fragmentação**. Com repetidas alocações e liberações, a memória vira um queijo suíço.

Mesmo havendo 25B livres no total, se alguém pedir 20B, a alocação falha! (Fragmentação Externa).

Algoritmos de Alocação Clássicos

O runtime mantém uma lista de blocos livres (*free list*).

- **First-Fit:** Procura a partir do início e pega o **primeiro** bloco livre que for suficientemente grande. (Rápido, mas gera mais fragmentação com o tempo).

Algoritmos de Alocação Clássicos

O runtime mantém uma lista de blocos livres (*free list*).

- **First-Fit:** Procura a partir do início e pega o **primeiro** bloco livre que for suficientemente grande. (Rápido, mas gera mais fragmentação com o tempo).
- **Best-Fit:** Procura a lista inteira e pega o **menor** bloco que satisfaça o pedido. (Tenta evitar dividir blocos grandes, mas deixa "lascas" microscópicas difíceis de usar).

Algoritmos de Alocação Clássicos

O runtime mantém uma lista de blocos livres (*free list*).

- **First-Fit:** Procura a partir do início e pega o **primeiro** bloco livre que for suficientemente grande. (Rápido, mas gera mais fragmentação com o tempo).
- **Best-Fit:** Procura a lista inteira e pega o **menor** bloco que satisfaça o pedido. (Tenta evitar dividir blocos grandes, mas deixa "lascas" microscópicas difíceis de usar).
- **Buddy System:** A memória é dividida em blocos de potências de 2 (2, 4, 8, 16...). Quando se pede 5, um bloco de 8 é retornado. Blocos "irmãos" (buddies) adjacentes livres são fundidos automaticamente para formar blocos maiores.

A Dificuldade do Descarte

Em C/C++, a desalocação no Heap é manual (`free`, `delete`). Erros comuns:

- **Memory Leaks:** Esquecer de liberar (a memória acaba).
- **Dangling Pointers:** Liberar o objeto, mas continuar usando o ponteiro (Corrupção de memória / Falhas de segurança).
- **Double Free:** Liberar o mesmo ponteiro duas vezes.

A Dificuldade do Descarte

Em C/C++, a desalocação no Heap é manual (`free`, `delete`). Erros comuns:

- **Memory Leaks:** Esquecer de liberar (a memória acaba).
- **Dangling Pointers:** Liberar o objeto, mas continuar usando o ponteiro (Corrupção de memória / Falhas de segurança).
- **Double Free:** Liberar o mesmo ponteiro duas vezes.

Garbage Collection (Lisp - 1959)

O Ambiente de Runtime automatiza a desalocação. Ele descobre quais objetos não podem mais ser acessados pelo programa e os recicla. Presente em Java, C#, Python, Go, JS...

O que é "Lixo"? O Conceito de Alcançabilidade

Um objeto está **vivo** (não é lixo) se ele for **alcançável**.

1. **Root Set:** Qualquer objeto referenciado diretamente por uma variável na Pilha (Stack), nos registradores ou nas variáveis Globais está vivo.
2. **Transição:** Qualquer objeto referenciado por um objeto vivo, também está vivo.

Todo o resto é lixo e pode ser varrido! O programa **nunca** conseguirá acessar objetos inalcançáveis novamente.

Estratégia 1: Reference Counting (Contagem de Referências)

- Cada objeto no Heap guarda um contador inteiro.
- Sempre que um ponteiro para o objeto é criado (atribuição), o contador aumenta.
- Quando um ponteiro é sobrescrito ou morre, o contador diminui.
- Se o contador chegar a 0, o objeto é liberado imediatamente. (Usado por Python, C++ `std::shared_ptr`).

Desvantagem Fatal: Ciclos

Se o Objeto A aponta para B, e B aponta para A, os contadores nunca chegam a 0, mesmo se ambos perderem ligação com a Pilha. Ocorre vazamento!

Estratégia 2: Mark-and-Sweep (Marcação e Varredura)

O programa para temporariamente (*Stop the World*). O GC executa duas fases:

1. **Mark:** Começa do *Root Set* e percorre o grafo de objetos marcando um bit (*marked*) em todos os objetos visitados.
2. **Sweep:** Varre o Heap inteiro de ponta a ponta. Objetos não marcados são devolvidos para a lista livre. Os bits de marcação são zerados para a próxima rodada.

Estratégia 2: Mark-and-Sweep (Marcação e Varredura)

O programa para temporariamente (*Stop the World*). O GC executa duas fases:

1. **Mark:** Começa do *Root Set* e percorre o grafo de objetos marcando um bit (*marked*) em todos os objetos visitados.
2. **Sweep:** Varre o Heap inteiro de ponta a ponta. Objetos não marcados são devolvidos para a lista livre. Os bits de marcação são zerados para a próxima rodada.

Vantagem: Resolve o problema de ciclos perfeitamente.
Desvantagem: O programa "congela". Causa pausas (latência) imprevisíveis. E a fragmentação da memória ainda existe.

Estratégia 3: Stop-and-Copy (Copiador)

Divide o Heap ao meio em dois "semiespaços": *From-Space* e *To-Space*.

- As alocações ocorrem apenas no *From-Space*.
- Quando lota, o GC é acionado. Ele copia todos os objetos **vivos** do *From-Space* compactados um do lado do outro no *To-Space*.
- O resto é ignorado (lixo morre rapidamente).
- Os papéis dos semiespaços se invertem.

Vantagem: Zero fragmentação (os objetos são compactados na cópia). Alocação nova é ultrarrápida (só incrementar um ponteiro).
Desvantagem: Desperdiça 50% da memória disponível.

Estratégia 4: Generational GC (GC Geracional)

Baseado em um fenômeno empírico chamado **Hipótese Geracional**:

"A maioria dos objetos morre jovem."

- O Heap é dividido em "Gerações" (ex: *Young* e *Old*).
- Objetos novos nascem no espaço *Young*. O GC roda frequentemente apenas nesta área (que é pequena), usando algoritmos copiadores rápidos.
- Objetos que sobrevivem a várias rodadas do GC são promovidos para o espaço *Old*.
- O espaço *Old* (gigante e cheio de coisas persistentes) é varrido raramente (com Mark-and-Sweep).

(Esta é a base do GC de alta performance da JVM V8 JavaScript moderno).

Resumo da Aula

- **Layout:** Código, Dados, Heap e Stack.
- **Stack (Pilha):** Organiza a execução procedural via **Registros de Ativação**. Controlada por convenções (*prologue/epilogue*) usando FP e SP.
- **Escopo Léxico:** Resolvido estruturalmente via **Links Estáticos** (Access Links).
- **Heap:** Para dados dinâmicos. Problema central é a **Fragmentação**.
- **Garbage Collection:** Algoritmos para reciclagem segura da memória, variando desde simples *Reference Counting* a *Mark-and-Sweep*, Copiadores e a moderna abordagem *Geracional*.

Referências

- AHO, A. V. et al. *Compiladores: Princípios, Técnicas e Ferramentas* (Livro do Dragão). 2ª Edição. Cap. 7 (Ambientes de Tempo de Execução).
- COOPER, K.; TORCZON, L. *Engineering a Compiler*. 2ª Edição. Cap. 6 (The Procedure Abstraction).
- APPEL, A. W. *Modern Compiler Implementation in Java*. Cap. 13 (Garbage Collection).

Dúvidas?

Obrigado!

Email: alessio@cefetmg.br
Web: alessiojr.com