

DCC024 Linguagens de Programação
2022.2

Gerenciamento de memória

Haniel Barbosa



Memória

- ▷ Quais componentes de um programa precisam ser armazenados?
 - ▶ o programa em si
Operações a serem executadas
 - ▶ o estado do programa
Valores manipulados por essas operações
- ▷ A *memória* é onde esses componentes são armazenados
- ▷ Veremos como gerenciar a memória para o estado de um programa

Gerenciamento de memória no interpretador de micro-ML

- ▷ O interpretador de expressões mantinha o estado de variáveis
`[("a", 5), ("f", ("x", Prim2("+", Var "x", Var "a"), [("a", 3)])), ...]`
- ▷ O *armazenamento* de valores se dava pela inserção na cabeça da lista
 - ▶ Definição de um novo escopo
- ▷ A *remoção* de valores se dava pela remoção da cabeça da lista
 - ▶ Saída de um escopo
- ▷ O *acesso* a valores durante a interpretação da expressão se dava por uma busca na lista

Gerenciamento de memória no interpretador de micro-ML

- ▷ O interpretador de expressões mantinha o estado de variáveis
`[("a", 5), ("f", ("x", Prim2("+", Var "x", Var "a"), [("a", 3)])), ...]`
- ▷ O *armazenamento* de valores se dava pela inserção na cabeça da lista
 - ▶ Definição de um novo escopo
- ▷ A *remoção* de valores se dava pela remoção da cabeça da lista
 - ▶ Saída de um escopo
- ▷ O *acesso* a valores durante a interpretação da expressão se dava por uma busca na lista

Possibilitar, de forma eficiente e segura, o **acesso, armazenamento e remoção** de dados é o problema central de gerenciamento de memória.

Gerenciamento de memória em C

```
int global_var = 7;
```

```
void foo(int* ptr)
{
    int auto_var = 9;
    ptr[0] = global_var;
    ptr[1] = auto_var;
}
```

```
int main()
{
    int* ptr = (int*) malloc(5 * sizeof(int));
    foo(ptr);
    ptr[2] = ptr[1] + global_var;
}
```

Tipos de memória

▷ Estática

- ▶ Permanente
- ▶ Variáveis globais

▷ Dinâmica

▶ *Stack*

- Armazenamento feito quando um escopo é criado (*push*)
- Remoção quando o escopo é destruído (*pop*)
- Gerenciamento automático

▶ *Heap*

- Piscina de blocos de memória, virtualmente ilimitada
- Armazenamento/remoção feitos independente de escopos
- Gerenciamento manual

Estática vs Dinâmica

- ⊕ Memória estática não tem custo extra de gerenciamento durante execução
- ⊖ Memória estática não é flexível
 - ▶ Se execução precisar de menos memória, haverá desperdício
 - ▶ Se execução precisar de mais memória, impossibilitará a execução
- ⊕ Memória dinâmica é flexível:
 - ▶ pode-se usar apenas o estritamente necessário para uma execução
 - ▶ Limitada apenas pelo total de memória no dispositivo
- ⊖ Memória dinâmica tem custo de gerenciamento durante a execução

Stack vs Heap

- ⊕ Armazenamento, remoção e acesso na stack são simples
- ⊖ Stack possui tamanho pré-determinado, então pode ocorrer *stack overflow*
 - ▶ Variáveis locais requerendo mais memória do que o limite
 - ▶ Recursão muito profunda levando a mais escopos do que o limite
- ⊕ Heap limitada apenas pela memória do dispositivo
- ⊖ Armazenamento, remoção e acesso são custosos e/ou complexos

Gerenciamento de memória na heap

- ▷ Considere a heap como um conjunto de *unidades de memória* indexado por inteiros (i.e., um vetor)
- ▷ Considere uma *lista de blocos livres* denotando partes desocupadas da heap
 - ▶ Cada elemento da lista contém: a posição de início do bloco livre, seu tamanho, e uma referência ao bloco livre seguinte
 - ▶ Inicialmente a lista contém um bloco livre englobando toda a memória
- ▷ O armazenamento de dados na heap reserva blocos (segmentos contíguos da memória) cuja primeira posição contém o tamanho do bloco

Armazenamento e remoção de dados da heap

Considere os seguintes métodos:

▷ `allocate($n : \text{Int}$) : (a : \text{Int})`

- ▶ Acha o primeiro bloco *ao menos* de tamanho $n + 1$ na lista de blocos livres
- ▶ Cria um bloco de tamanho $n + 1$ começando na posição a do bloco livre. O tamanho do bloco é salvo na posição a
- ▶ O restante do bloco livre se torna um novo bloco livre e é adicionado à cabeça da lista

▷ `deallocate($a : \text{Int}$)`

- ▶ Remove o bloco na posição a e o adiciona à cabeça da lista de blocos livres

Armazenamento e remoção de dados da heap

Considere os seguintes métodos:

▷ `allocate($n : \text{Int}$) : ($a : \text{Int}$)`

- ▶ Acha o primeiro bloco *ao menos* de tamanho $n + 1$ na lista de blocos livres
- ▶ Cria um bloco de tamanho $n + 1$ começando na posição a do bloco livre. O tamanho do bloco é salvo na posição a
- ▶ O restante do bloco livre se torna um novo bloco livre e é adicionado à cabeça da lista

▷ `deallocate($a : \text{Int}$)`

- ▶ Remove o bloco na posição a e o adiciona à cabeça da lista de blocos livres

▷ Exemplo: considere uma heap m de tamanho 10

```
p1 = m.allocate(4);
```

```
p2 = m.allocate(2);
```

```
m.deallocate(p1);
```

```
p3 = m.allocate(1);
```

Exemplos de uso da heap

▷ E se tivéssemos as seguintes operações?

```
p1 = m.allocate(4);  
p2 = m.allocate(4);  
m.deallocate(p1);  
m.deallocate(p2);  
p3 = m.allocate(7);
```

Exemplos de uso da heap

- ▷ E se tivéssemos as seguintes operações?

```
p1 = m.allocate(4);
```

```
p2 = m.allocate(4);
```

```
m.deallocate(p1);
```

```
m.deallocate(p2);
```

```
[fails] p3 = m.allocate(7);
```

- ▷ Apesar de a memória, de tamanho 10, estar totalmente livre, está dividida em dois blocos livres adjacentes de tamanho 5 cada.

Combinação de blocos livres

- ▷ Combinar blocos livres adjacentes permite o armazenamento no caso anterior
- ▷ Manter a lista de blocos livres ordenada pela posição de início dos blocos permite fácil combinação
- ▷ Basta um novo bloco ser adjacente ao seu antecessor ou sucessor na lista para uma combinação ser possível

Fragmentação de memória

▷ A solução anterior só é suficiente para blocos adjacentes

▷ Exemplo:

```
p1 = m.allocate(3);  
p2 = m.allocate(2);  
m.deallocate(p1);  
p3 = m.allocate(2);  
p4 = m.allocate(3);
```

Fragmentação de memória

▷ A solução anterior só é suficiente para blocos adjacentes

▷ Exemplo:

```
p1 = m.allocate(3);
```

```
p2 = m.allocate(2);
```

```
m.deallocate(p1);
```

```
p3 = m.allocate(2);
```

```
[fails] p4 = m.allocate(3);
```

▷ Apesar de a memória possuir, em princípio, espaço o suficiente, ela está *fragmentada*

Onde armazenar blocos?

▷ Diferentes possibilidades com vantagens e desvantagens. Alguns exemplos:

▷ *First fit*

▶ Armazenamento no bloco compatível com a menor posição na memória

⊕ Simples e eficiente

⊖ Pode separar blocos grandes mais facilmente

▷ *Best fit*

▶ Armazenamento no bloco compatível de menor tamanho

⊕ Maior sinergia entre bloco a ser armazenado e blocos disponíveis

⊖ Pode levar a blocos muito pequenos mais facilmente

⊖ Implementação mais complexa da lista de blocos para acesso eficiente (e.g., árvores rubro-negras)

Evitando fragmentação de memória no exemplo anterior

▷ Com armazenamento por *best fit* o exemplo é bem-sucedido:

▷ Exemplo:

```
p1 = m.allocate(3);
```

```
p2 = m.allocate(2);
```

```
m.deallocate(p1);
```

```
p3 = m.allocate(2);
```

```
[works!] p4 = m.allocate(3);
```

▷ O bloco apontado por p3 não mais é alocado no começo

▷ Um bloco de tamanho 4 está disponível para o armazenamento apontado por p4.

Três principais aspectos do gerenciamento da heap

- ▷ *Placement*: dentre vários blocos livres, em qual realizar o armazenamento
 - ▶ First fit, best fit, ...
- ▷ *Splitting*: quando e como separar grandes blocos livres de memória
 - ▶ pode ser melhor não gerar blocos livres de tamanhos “incomuns” durante o armazenamento
- ▷ *Coalescing*: quando e como recombinar blocos de memória
 - ▶ em casos com frequente armazenamento de pequenos blocos pode ser melhor não recombinar blocos adjacentes, ou fazê-lo só quando necessário

Erros devido ao mau uso de memória

- ▷ O gerenciamento manual de memória pode levar a erros
- ▷ Um erro comum é não remover blocos de memória armazenados
 - ▶ Vazamentos de memória (*Memory leaks*)
 - ▶ Pode levar a um consumo excessivo de memória
- ▷ Um erro ainda mais perigoso é o de manter referências a blocos de memória que foram removidos
 - ▶ Ponteiros cegos (*Dangling pointers*)
 - ▶ Pode levar a comportamentos inesperados

Exemplo de vazamento de memória

```
void leak()  
{  
    int* i = (int*) malloc(sizeof(int));  
    *i = 3;  
    printf("%d\n", *i);  
}
```

```
int main()  
{  
    leak();  
}
```

Exemplo de ponteiro cego

```
void dangling()  
{  
    int* i = (int*) malloc(sizeof(int));  
    int* j;  
    *i = 3;  
    free(i);  
    j = (int*) malloc(sizeof(int));  
    *j = 8;  
    printf("%d\n", *i);  
}
```

```
int main()  
{  
    dangling();  
}
```

Valgrind for fun and profit

▷ Pode detectar vazamentos de memória

```
$ valgrind ./11-mem-leak
...
==70445== LEAK SUMMARY:
==70445==    definitely lost: 4 bytes in 1 blocks
==70445==    indirectly lost: 0 bytes in 0 blocks
==70445==    possibly lost: 0 bytes in 0 blocks
==70445==    still reachable: 0 bytes in 0 blocks
==70445==    suppressed: 0 bytes in 0 blocks
```

▷ Pode detectar acessos inválidos de memória

```
$ valgrind ./11-mem-dangling
...
==71000== Invalid read of size 4
==71000==    at 0x1091D5: dangling (in /home/hbarbosa/teaching/lp/codes/11-mem-dangling)
==71000==    by 0x1091FE: main (in /home/hbarbosa/teaching/lp/codes/11-mem-dangling)
==71000== Address 0x4a5a040 is 0 bytes inside a block of size 4 free'd
==71000==    at 0x483CA3F: free (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-
linux.so)
==71000==    by 0x1091B8: dangling (in /home/hbarbosa/teaching/lp/codes/11-mem-dangling)
==71000==    by 0x1091FE: main (in /home/hbarbosa/teaching/lp/codes/11-mem-dangling)
==71000== Block was alloc'd at
==71000==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-
linux.so)
==71000==    by 0x10919E: dangling (in /home/hbarbosa/teaching/lp/codes/11-mem-dangling)
==71000==    by 0x1091FE: main (in /home/hbarbosa/teaching/lp/codes/11-mem-dangling)
```

Coleta de lixo (*Garbage collection*)

- ▷ Mau uso de memória pode ser evitado com abordagens automáticas de remoção de blocos armazenados na heap
- ▷ Essas abordagens são conhecidas como coleta de lixo
 - ▶ Remoção de blocos que não são mais utilizados (lixo)
- ▷ Geralmente requer determinar quais blocos da memória são relevantes:
 - ▶ Dadas um conjunto inicial de interesse, adicionar todas as posições de memória alcançáveis a partir daquele conjunto e as adicionar ao conjunto
 - ▶ Repetir até um ponto fixo
- ▷ Diferentes abordagens com vantagens e desvantagens

Exemplos de coleta de lixo: *mark and sweep*

- ▶ Marcação: marcar todos os blocos de memória relevantes
- ▶ Varredura: remoção de todos os blocos armazenados que não são relevantes
- ⊕ Não é necessário mover blocos armazenados relevantes
- ⊖ Não afeta fragmentação da heap
- ⊖ Requer uma extensa análise da memória armazenada

Exemplos de coleta de lixo: *copying collector*

- ▶ Heap é dividida em duas partes, com uma parte usada de cada vez
- ▶ Quando a parte usada enche, copia-se *apenas* os blocos relevantes para a outra parte e remove a primeira.
- ⊕ Elimina fragmentação
- ⊖ Requer copiar todos os blocos relevantes
- ⊖ Requer uma extensa análise da memória armazenada

Exemplos de coleta de lixo: *reference counting*

- ▷ Todo bloco armazenado é associado a um contador
- ▷ Contador incrementado (resp. decrementado) quando referência ao bloco é copiada (resp. descartada)
- ▷ Quando o contador atinge zero, o bloco é removido
- ⊕ Não é preciso analisar a memória para determinar blocos relevantes
- ⊕ Custo da coleta é incremental, sem uma grande pausa
- ⊖ Manutenção dos contadores adiciona custo extra a operações
- ⊖ Contagem errada pode levar a ponteiros cegos e vazamentos de memória