Alocação Dinâmica de Memória em C

Roland Teodorowitsch

Programação de Software Básico - Curso de Engenharia de Software - PUCRS

18 de maio de 2023

Motivação

Motivação (1)

- Por que precisamos de alocação dinâmica?
- Um experimento simples:

```c
#include <stdio.h>
#define TAM 1000000 // um milhao
int main() {
    double vetor[TAM];
    printf("tamanho de memoria: %zu\n", sizeof(vetor)); // sizeof(double)*TAM
    for (int i=0; i<TAM; i++)
        vetor[i] = i;
    return 0;
}
```

- %zu é o tipo correto para o valor de retorno de `sizeof()`
- Funciona?

Motivação (2)

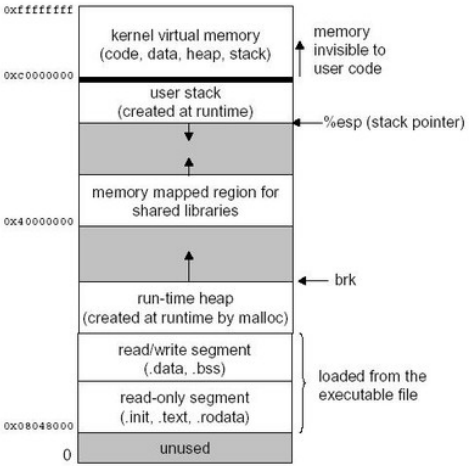
- Talvez funcione, talvez não...
- Vamos tentar outro:

```c
#include <stdio.h>
#define TAM 2000000 // dois milhoes
int main() {
    double vetor[TAM];
    printf("tamanho de memoria: %zu\n", sizeof(vetor)); // sizeof(double)*TAM
    for (int i=0; i<TAM; i++)
        vetor[i] = i;
    return 0;
}
```

- Agora certamente **NÃO funciona** – mas por quê?

Entendendo o Uso da Memória

- Veja o *layout* de memória de um processo em execução no Linux
- *User stack*: armazena todas as variáveis locais de uma função, registradores, endereço de retorno, etc.
- *Runtime heap*: compartilhado por todos os processos em execução no sistema, usado para alocação dinâmica
- *Read-write segment*: as variáveis globais de um programa
- *Read-only segment*: o código do programa (pode ser compartilhado entre instâncias)



Entendendo a memória em um programa C

- Quais são esses limites?
- Podem ser consultados e alterados pela linha de comando: o comando `ulimit`

```
ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 31119
max locked memory       (kbytes, -l) 65536
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 31119
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

Tamanho da pilha é limitado!

- A parte importante é destacada abaixo:

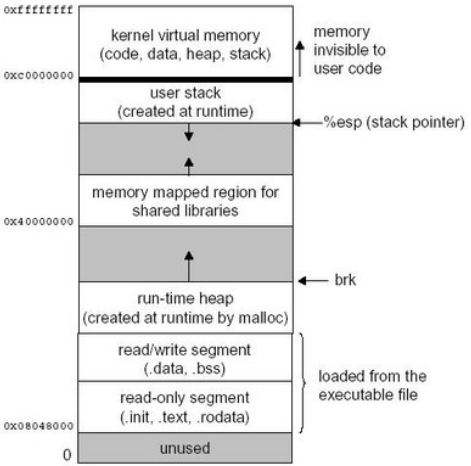
```
...
stack size              (kbytes, -s) 8192
...
```

- O tamanho da pilha para qualquer processo é 8192KB, isto é, **8MB**
- Mas quanta memória precisamos para 2 milhões de variáveis do tipo `double`?
- 16 milhões de bytes!

Alocação Dinâmica

Alocação dinâmica em C

- Resolve o problema de armazenar grandes quantidades de dados na pilha
- Também possibilita a alocação de memória que irá persistir além do escopo de uma função
  - Por exemplo, para estruturas encadeadas (listas, etc.)
- Lembre-se do *layout* de memória: alocação dinâmica usa memória do ***runtime heap***
- É preciso usar **funções** específicas para gerenciar memória dinamicamente



Funções para gerência de memória

- Geralmente presentes no arquivo de cabeçalho `stdlib.h`
- Para **alocar memória**, usa-se `malloc()`, `calloc()` ou `realloc()`:

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void* ptr, size_t size);
```

- Para **liberar memória**, usa-se `free()`:

```
void free(void* ptr);
```

- `size_t` é um tipo sem sinal de pelo menos 16 bits (geralmente é um `unsigned int`)

Alocando memória

- O **ponteiro** `void` é um ponteiro para uma área de memória cujo tipo não é definido (pode apontar para qualquer coisa – vetor de `int`, `char`, ...)
- A função `malloc()` é a mais simples – tenta alocar `size` *bytes* de memória, e retorna um ponteiro para o bloco alocado (ou `NULL` se falhar)

```
void *malloc(size_t size);
```

- A quantidade exata de *bytes* pode ser obtida com o uso de `sizeof`:

```
// Aloca um vetor de 10 int
int *numeros = malloc( 10 * sizeof(int));
// ou:
int *numeros = malloc( 10 * sizeof *numeros );
```

Alocando memória (2)

- A função `calloc()` aloca memória para um vetor de `nmemb` itens, onde cada um requer `size` *bytes*
- Também inicializa a memória com **zeros** (`malloc()` não faz isso)

```
void *calloc(size_t nmemb, size_t size);
```

- O exemplo anterior com `calloc()`:

```
// Aloca e zera um vetor de 10 int
int *numeros = calloc(10, sizeof(int)));
```

Alocando memória (3)

- A função `realloc()` altera o tamanho de um bloco já alocado (`ptr`)
- Se é maior e não cabe, a função irá movê-lo para outro lugar e liberar o ponteiro original
- Retorna um ponteiro para o novo bloco alocado

```
void *realloc(void *ptr, size_t size);
```

- Exemplo: criando espaço para mais 20 `int` no vetor

```
int *mais_numeros = realloc(numeros, 30);
```

Liberando memória

- Para liberar memória alocada dinamicamente, chama-se a função `free()`, passando o ponteio do bloco de memória para ela:

```
void free(void *ptr);
```

- Exemplo: liberando o vetor de inteiros

```
free(mais_numeros);
```

- **Observação importante**: não há ***garbage collection*** em C, portanto, lembre-se de liberar **toda** a memória alocada dinamicamente

Voltando ao exemplo...

- Agora é possível voltar e alterar o exemplo anterior para usar alocação dinâmica:
- Vamos tentar outro:

```c
#include <stdio.h>
#include <stdlib.h>
#define TAM 2000000 // dois milhoes
int main() {
    double *vetor = malloc( TAM * sizeof(double) );
    printf("tamanho de memoria: %zu\n", TAM * sizeof(double));
    for (int i=0; i<TAM; i++)
        vetor[i] = i;
    free(vetor);
    return 0;
}
```

- Agora funciona?

Alocação de Matrizes

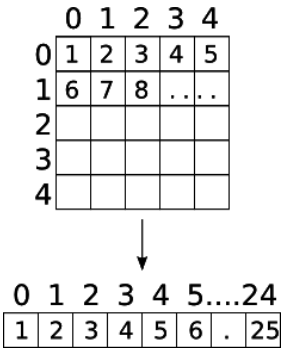
Alocando uma matriz

- Uma matriz pode ser vista como memória linear, ou como uma estrutura realmente bidimensional
- O primeiro caso é simples – para uma matriz 5x5 de `int`, basta fazer:

```
int *mat = malloc( 5 * 5 * sizeof(int) );
```

- Mas agora o problema é acessar através de um só índice:

```
mat[0] = 1; // armazena 1 na pos (0,0)
mat[1] = 2; // armazena 2 na pos (0,1)
...
mat[4] = 5; // armazena 5 na pos (0,4)
mat[5] = 6; // armazena 6 na pos (1,0)
...
// genericamente: mat[5 * linha + coluna] = valor;
```



Alocando dinamicamente uma matriz (2)

- Para trabalhar com dois índices, primeiro é preciso alocar um vetor de linhas:

```
int **mat = malloc ( 5 * sizeof(int *) );
```

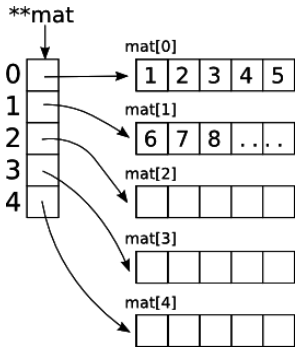
- Então, para cada linha, aloca-se o espaço para as suas respectivas colunas:

```
for (int i=0; i<5; i++)
    mat[i] = malloc( 5 * sizeof(int) );
```

- Agora basta usar o operador de matriz:

```
// Armazena 6 na linha 1, coluna 0
mat[1][0] = 6;
```

- **Observação**: isto provavelmente impedirá a alocação contígua das linhas



Alocando dinamicamente uma matriz (3)

- Para liberar a matriz, basta reverter o processo
- Primeiro libera-se cada linha, e depois libera-se o vetor de linhas:

```
for (int i=0; i<5; i++)
    free(mat[i]);
free(mat);
```

Alocando dinamicamente uma matriz (4)

- Outra opção, se o compilador C suportar o padrão C99 (*variable-length arrays*)
- Alocar uma matriz inteira **em uma única chamada**:

```
int n = 5;
int (*mat)[n] = malloc(n * sizeof *mat);
```

- Usar normalmente:

```
mat[1][0] = 6; // armazena 6 na linha 1, coluna 0
```

- E liberar no final:

```
free(mat);
```

Estruturas Encadeadas

Estruturas Encadeadas

- Alocação dinâmica é **essencial** para a criação de estruturas dinâmicas usuais, como listas encadeadas, pilhas, filas, árvores e grafos.

Criando uma lista encadeada

- Por exemplo, o nodo de uma lista poderia ser declarado assim:

```
typedef struct {
    int      dado;
    nodo_t *prox;
} nodo_t;
```

- Porém, isso não funciona: C não permite uma auto-referência para um tipo que **está sendo definido neste momento**.

Criando uma lista encadeada (2)

- A solução é definir um nome temporário para a definição da `struct`:

```
typedef struct nodo {
    int           dado;
    struct nodo *next;
} nodo_t;
```

- Uma estrutura de lista então poderia ser definida assim:

```
typedef struct {
    nodo_t* head;
    nodo_t* tail;
    int tamanho;   // conveniente sempre saber o tamanho
} lista_t;
```

- Basta criar agora um conjunto de funções para manipular `nodo_t` e `lista_t`

Exercícios

Exercício 1

1. Escreva um programa em C que lê as dimensões (linhas x colunas) de uma matriz de `char` e aloca memória para ela. Depois o programa deve fazer o seguinte:
   - Inicialize todas as posições com espaços em branco
   - Escolha aleatoriamente 5 posições na matriz e preencha-as com o caractere `'.'` (ponto) – estes chamamos de sementes
   - Para cada posição ainda vazia m_{ij}, encontre a distância até a semente mais próxima (S_{kl}). Para isto, você pode usar a fórmula da distância Euclidiana:

     $$d(m_{ij}, S_{kl}) = \sqrt{(i-k)^2 + (j-l)^2}$$

     ou mais simples:

     $$d(m_{ij}, S_{kl}) = \max(|i-k|, |j-l|)$$

   - Agora armazene nesta célula o número da semente mais próxima (de '1' a '5' ou use outros símbolos, se quiser)
   - No final, exiba a matriz resultante na tela e não se esqueça de liberar a memória alocada!
   - A figura resultante é chamada de Diagrama de Voronoi. Descubra mais em: `http://en.wikipedia.org/wiki/Voronoi_diagram`

Créditos

Créditos

- Figura das lâminas 5 e 9: `http://www.linuxjournal.com/article/6701`
- Estas lâminas contêm trechos de materiais criados e disponibilizados pelo professor Marcelo Cohen.