Alocação dinâmica de memória - Ponteiros

Prof. Marcelo Veiga Neves

Motivação

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

```
#include <stdio.h>

#define SIZE 1000000

int main()
{
    double array[SIZE];
    printf("tam. mem: %zu\n", sizeof(double)*SIZE);
    for (int i = 0; i < SIZE; i++)
        array[i] = i;
    ...
}
```

- %zu é o tipo correto para o valor de retorno de *sizeof*
- Funciona?

Motivação

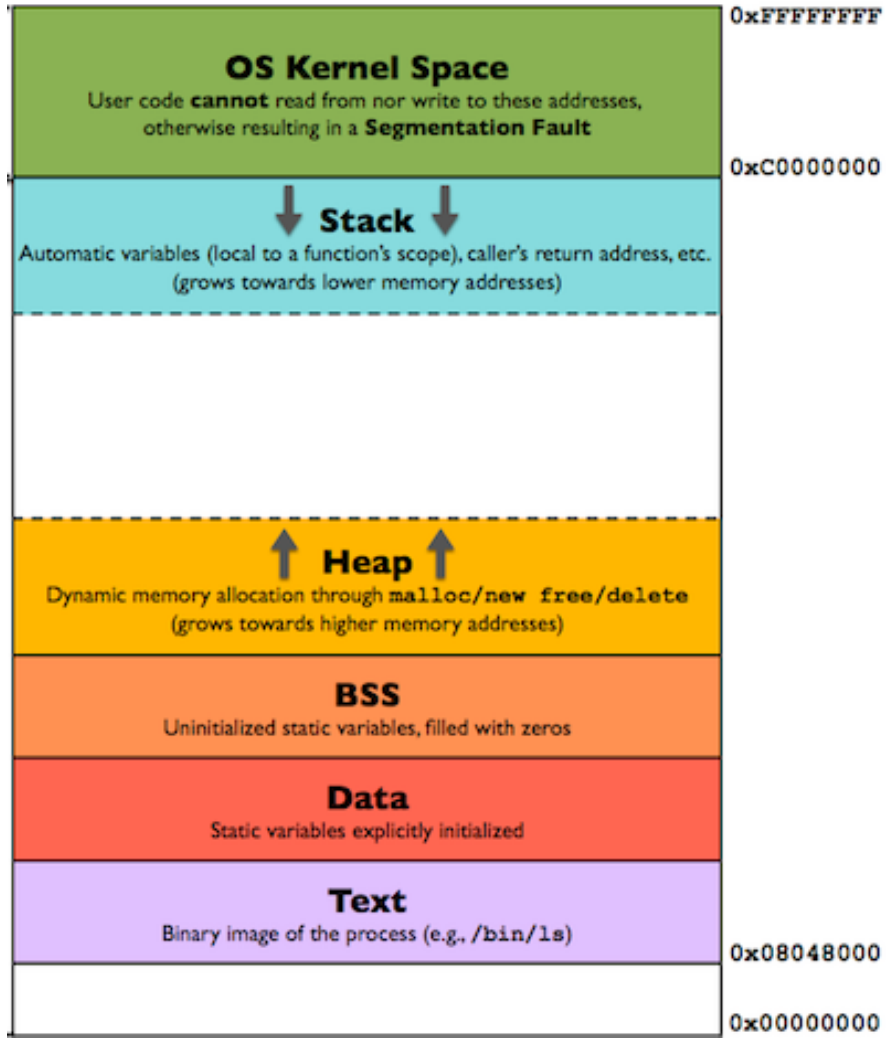
- Talvez funcione, talvez não...
- Vamos tentar outro:

```
#include <stdio.h>
#define SIZE 2000000 // dois milhões
int main()
{
    double array[SIZE];
    printf("tam. mem: %zu\n", sizeof(double)*SIZE);
    for (int i = 0; i < SIZE; i++)
        array[i] = i;
    ...
}
```

- Agora certamente não funciona. Mas por quê?

Entendendo o uso da memória

- Layout de memória de um processo em execução no Linux:



Entendendo o uso da memória

- Layout de memória de um processo em execução no Linux:
- *Stack* (pilha) - armazena todas as variáveis locais de uma função, registradores, endereço de retorno, etc;
- *Heap* (alocação dinâmica) - espaço de memória utilizado para alocação dinâmica;
- *BSS* (segmento não inicializado) - espaço de memória utilizado para alocação estática de memória não inicializada, como variáveis globais;
- *Data* (variáveis inicializadas) - armazena constantes e dados inicializados, como *strings*;
- *Text* (código) - código do programa

Entendendo a memória em um programa C

- Quais são esses limites?
- Podem ser consultados e alterados pela linha de comando com o uso do 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) 63422
max locked memory       (kbytes, -l) unlimited
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) 95
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 63422
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

O 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 (8MB)
- Mas quanta memória precisamos para 2 milhões de *doubles*?
  16 milhões de bytes!

Alocação dinâmica em C

- Resolve o problema de armazenar grandes quantidades de dados na pilha
- Também nos permite alocar 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 *heap*
- Precisamos usar funções para gerenciar memória dinamicamente (manualmente)

Funções para gerência de memória

- Geralmente presentes no header *stdlib.h*
- Para alocar memória, usamos *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, usamos 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 - array de *int*, *char*, ...)
- A função *malloc()* é a mais simples - tenta alocar size bytes de memória, retorna ponteiro para o bloco (NULL se falhar)

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

- Usar quantidade desejada de bytes, ou seja, usando *sizeof* :

```
// Aloca um array de 10 int
int *numbers = malloc(sizeof(int) * 10);
// Faz o mesmo, e é considerado mais seguro
int *numbers = malloc(10 * sizeof *numbers);
```

Alocando memória

- A função *calloc()* aloca memória para um array 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 array de 10 int
int *numbers = calloc(10, sizeof *numbers));
```

Alocando memória

- 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 array

```
int *newarray = realloc(numbers, 30);
```

Liberando memória

- Para liberar memória alocada dinamicamente, precisamos chamar a função *free()*, passando o ponteio do bloco de memória para ela:
- 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 free(void *ptr);
```

- Exemplo: liberando o *array* de inteiros

```
free(numbers);
```

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

Voltando ao exemplo

- Agora podemos voltar e alterar nosso exemplo para usar alocação dinâmica:

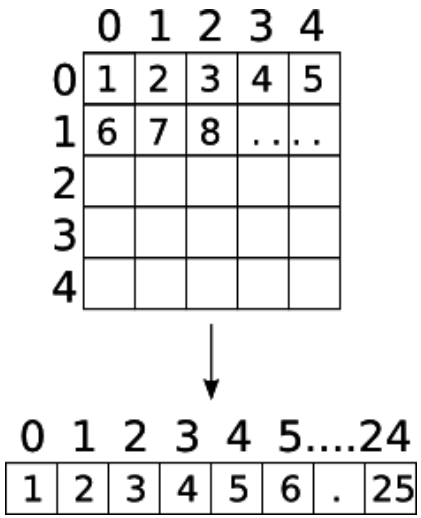
```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 2000000

int main()
{
    double *array = malloc(sizeof *array * SIZE);
    printf("tam. mem: %zu\n", sizeof *array * SIZE);
    for (int i = 0; i < SIZE; i++)
        array[i] = i;
    free(array);
    ...
}
```

- Agora funciona?

Alocando uma matriz

- Uma matriz pode ser vista como memória linear, ou como uma estrutura realmente bidimensional
- O primeiro caso é simples - se desejamos uma matriz 5x5 de *int*, podemos fazer:

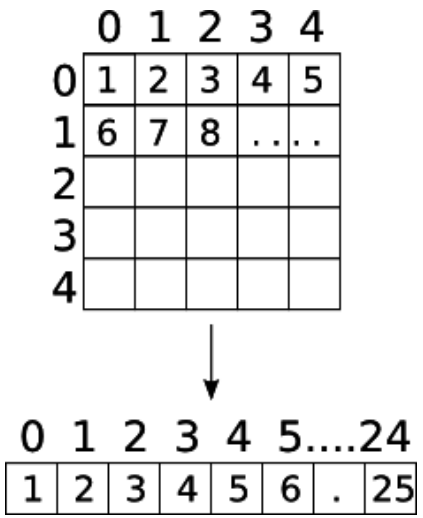
```
int *mat = malloc(5 * 5 * sizeof *mat);
```



Alocando uma matriz

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

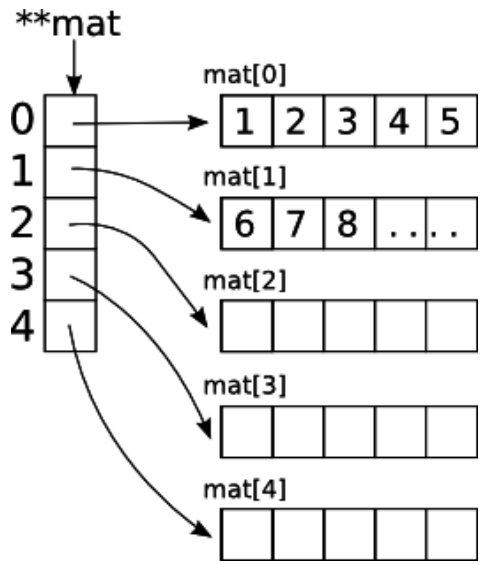
```
mat[0] = 1; // armazena 1 na pos (0,0)
mat[1] = 2; // armazena 2 na pos (0,1)
...
mat[4] = 5; // armazena 5 na pos (0,4)
mat[5] = 6; // armazena 6 na pos (1,0)
...
// genericamente: mat[5 * linha + coluna] = valor;
```



Alocando dinamicamente uma matriz

- Para trabalhar com dois índices, precisamos primeiro alocar um array de linhas:

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



- Então, para cada linha, alocamos espaço para as colunas:

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

- Agora basta usar o operador de matriz:

```
mat[1][0] = 6;
```

Alocando dinamicamente uma matriz

- A matriz anterior não será alocada contiguamente na memória!
- Para liberar a matriz, basta reverter o processo
- Primeiro liberamos cada linha, e depois liberamos o array de linhas:

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

Alocando dinamicamente uma matriz

- 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;
```

- E liberar no final:

```
free(mat);
```

Exercícios

1. Escreva um programa que lê as dimensões de uma matriz (linhas x colunas) de *char* e aloca memória para ela. Depois o programa deve fazer o seguinte:
   - Inicialize todas as posições com espaços em branco
   - Gere randomicamente 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 Euclideana:
   - $d(m_{ij}, S_{kl}) = \sqrt{(i-k)^2 + (j-l)^2}$
   - 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

   Para os que estiverem curiosos, a figura resultante é chamada de Diagrama de Voronoi

Referências

- Este material foi construído com base nos slides de aula do prof. Marcelo Cohen