

Linguagem C

- Gestão da Memória Dinâmica -

Arquitetura de Computadores 2024/2025

Gestão de Memória em C (1/2)

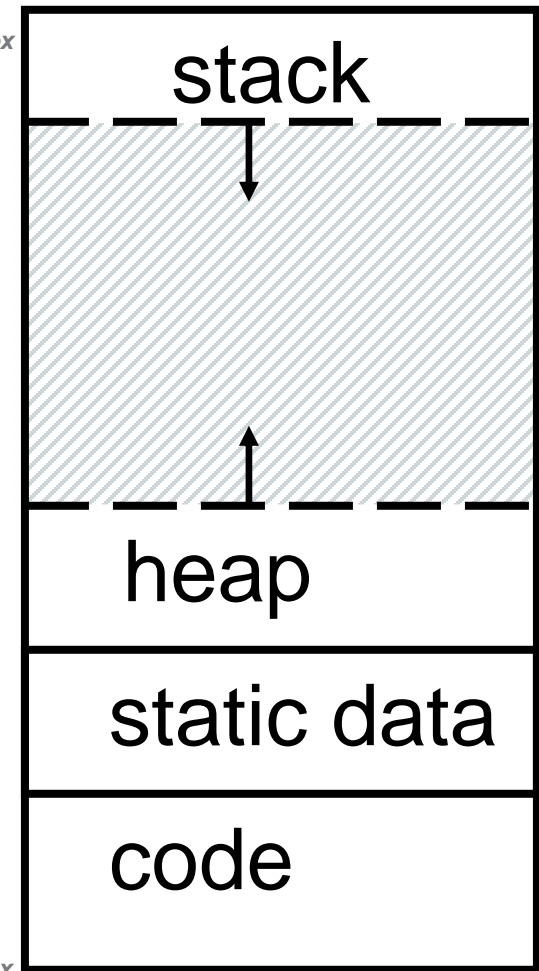
- Um programa em C define três zonas de memória distintas para o armazenamento de dados:
 - Static Storage: onde ficam as variáveis globais que podem ser lidas/escritas por qualquer função do programa. Este espaço está alocado permanentemente durante todo o tempo em que o programa corre (daí o nome estático)
 - A Pilha/Stack: armazenamento de variáveis locais, parâmetros, endereços de retorno, etc.
 - A Heap (dynamic malloc storage): os dados são válidos até ao instante em que o programador faz a desalocação manual com `free()`.
- O C precisa de conhecer a localização dos objectos na memória, senão as coisas não funcionam correctamente.

Gestão de Memória em C (2/2)

- O *espaço de endereçamento* de um programa contém 4 regiões:
 - *stack*: variáveis locais, *cresce para baixo*
 - *heap*: espaço requisitado via `malloc()`; *cresce para cima*.
 - *Dados estáticos*: variáveis globais declaradas fora do `main()`, *tamanho constante durante a execução*.
 - *código*: Carregado quando o programa começa, *o tamanho não se modifica*.

O Sistema Operativo evita a sobreposição da Stack com a Heap

$\sim \text{FFFF FFFF}_{\text{hex}}$



$\sim 0_{\text{hex}}$

A Heap (Memória Dinâmica)

- Grande bloco de memória, onde a alocação não é feita de forma contígua. É uma espécie de "*espaço comunitário*" do programa.
- Em C, é necessário especificar o número exacto de bytes que se pretende alocar:

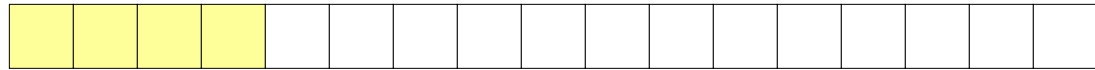
```
int *ptr;  
ptr = (int *) malloc(sizeof(int));  
/* malloc returns type (void *),  
so need to cast to right type */
```

—`malloc()`: aloca memória não inicializada na área da *heap*



Exemplos de Alocação de Memória

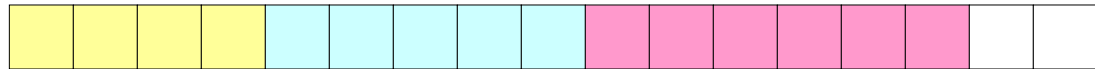
`p1 = malloc(4)`



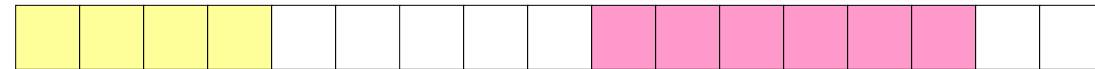
`p2 = malloc(5)`



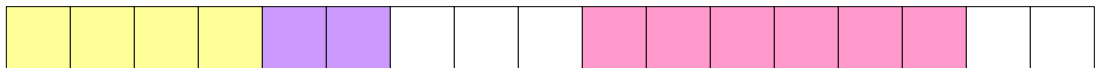
`p3 = malloc(6)`



`free(p2)`

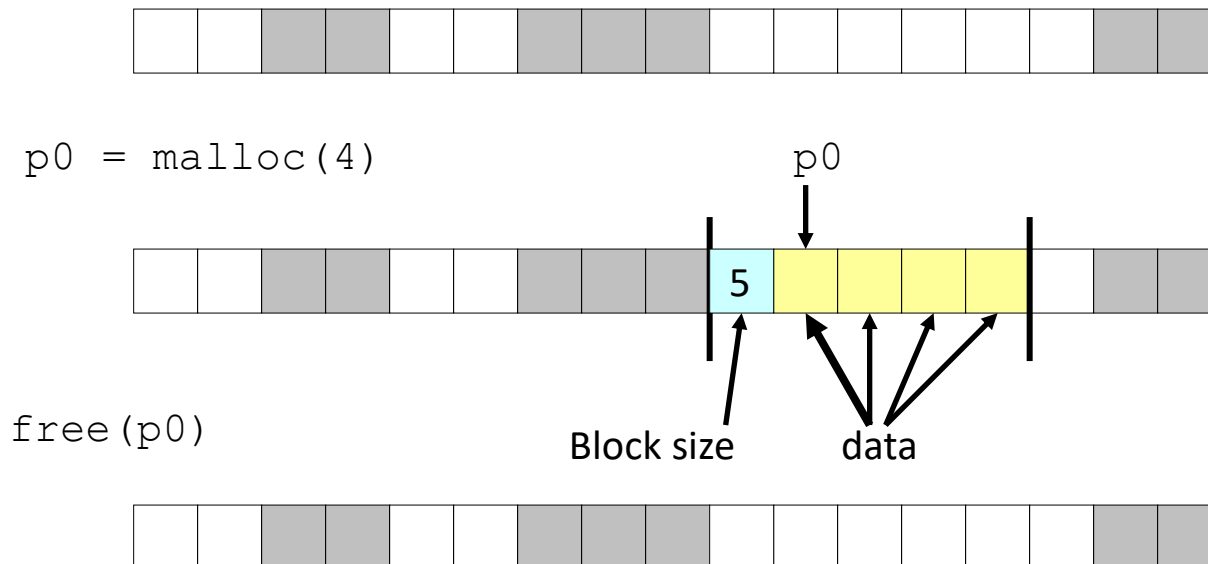


`p4 = malloc(2)`



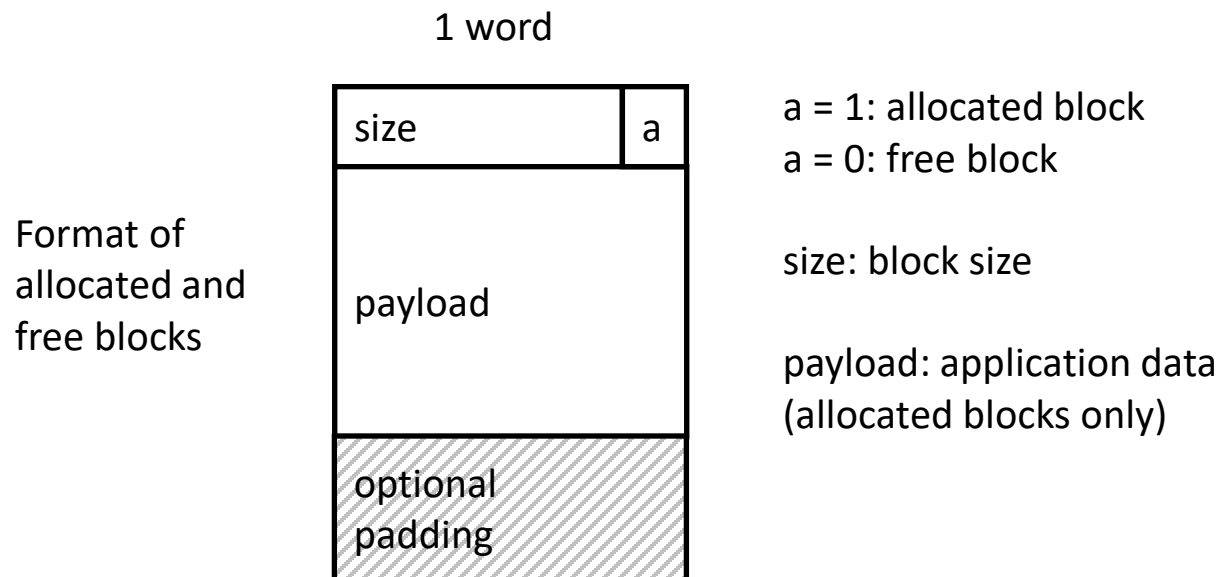
Como saber o que libertar?

- Método mais comum:
 - Armazenar o comprimento do bloco na palavra que precede o bloco.
 - Esta palavra é denominada *header field* ou *header*
 - Requer uma palavra extra por cada bloco alocado



Lista Implícita

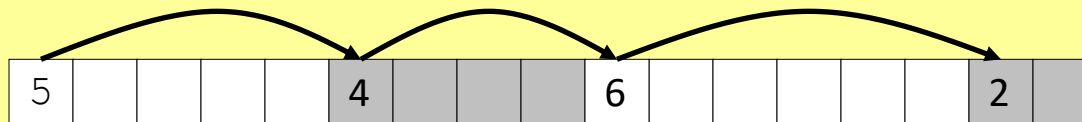
- Necessita de identificar se o bloco está livre ou não
 - Utiliza um *bit* extra
 - Este bit pode ser colocado no tamanho do bloco admitindo que este tem sempre um tamanho múltiplo de 2 (utiliza-se uma máscara para filtrar o bit menos significativo quando se quer ler o tamanho).



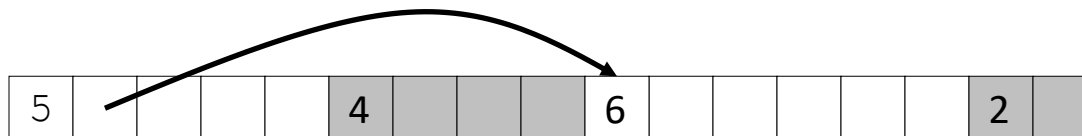


Lista Implícita

- Método 1: *Lista Implícita* utilizando os comprimentos dos blocos – liga todos os blocos:



- Método 2: *Lista Explícita* nos blocos livres utilizar ponteiros para os próximos blocos livres:



- Método 3: *Listas Explícitas por classes*
 - Diferentes listas para diferentes classes de tamanhos de blocos
- Método 4: Blocos ordenados por tamanho

Mecanismos de Gestão da Heap

- Alocação Dinâmica "Manual" - Caso do C, em que o programador é responsável por alocar e libertar os blocos de memória
 - malloc()/free() implementação do K&R Sec 8.7 (ler só introdução)
 - Slab Allocators
 - Buddy System
- Alocação "Automática" / Garbage Collectors - O sistema mantém registo de forma automática das zonas da *heap* que estão alocadas e em uso, reclamando todas as restantes*
- Contagem de referências
- Mark and Sweep
- Copying Garbage Collection

* O overhead com Garbage Collectors é obviamente maior

Implementação do malloc/free (K&R Sec. 8.7)

- Cada bloco de memória na *heap* tem um cabeçalho com dois campos:
 - tamanho do bloco e
 - um ponteiro para o bloco livre seguinte
- Todos os blocos livres são mantidos numa lista ligada circular (a "*free list*").
- Normalmente os blocos da "*free list*" estão por ordem crescente de endereços no espaço de endereçamento
- No caso de um bloco ser alocado, o seu ponteiro fica NULL.

Implementação do malloc/free (K&R Sec. 8.7)

- `malloc()` procura na "free list" um bloco que seja suficientemente grande para satisfazer o pedido.
 - Se existir, então bloco é partido de forma a satisfazer o pedido, e a "sobra" é mantida na lista.
 - Se não existir então é feito um pedido ao sistema operativo de mais áreas de memória.
- `free()` verifica se os blocos adjacentes ao bloco liberto também estão livres.
 - Se sim, então os blocos adjacentes são concatenados (*coalesced*) num único bloco de maiores dimensões (para evitar fragmentação)
 - Se não, o bloco é simplesmente adicionado à "free list".

Qual é o bloco que o `malloc()` escolhe?

- Se existirem vários blocos na "*free list*" que satisfaçam os requisitos, qual deles é que é escolhido?
 - *best-fit*: escolhe o bloco mais pequeno que satisfaça os requisitos de espaço
 - *first-fit*: Escolhe o primeiro bloco que satisfaça os requisitos
 - *next-fit*: semelhante ao *first-fit*, mas lembra-se onde terminou a pesquisa da última vez, e retoma-a a partir desse ponto (não volta ao início)



QUIZ - Prós e Contras dos "Fit"

- A. Um contra do **first-fit** é que resulta em vários **pequenos blocos** no início da free list
- B. Um contra do **next-fit** é que é **mais lento do que o first-fit**, dado que demora mais tempo à procura de um bloco adequado
- C. Um contra do **best-fit** é que **gera muitos blocos de pequenas dimensões na free list**


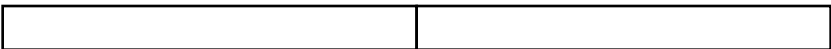
| | ABC |
|----|-----|
| 0: | FFF |
| 1: | FFT |
| 2: | FTF |
| 3: | FTT |
| 4: | TEF |
| 5: | TFT |
| 6: | TTF |
| 7: | TTT |

Slab Allocator (1/2)

- Um sistema alternativo utilizado na GNU `libc`
- Divide os blocos que formam a *heap* em "grandes" e "pequenos". Os "grandes" são geridos através de uma *freelist* como anteriormente.
- Para blocos pequenos, a alocação é feita em blocos que são múltiplos de potências de 2
 - e.g., se o programa quiser alocar 17 bytes, dá-se-lhe 32 bytes.

Slab Allocator (2/2)

- A gestão dos pequenos blocos é fácil; basta usar um *bitmap* para cada gama de blocos do mesmo tamanho

| | | |
|-----------------|--|--------------------------------|
| 16 byte blocks: |  | 16 byte block bitmap: 11011000 |
| 32 byte blocks: |  | 32 byte block bitmap: 0111 |
| 64 byte blocks: |  | 64 byte block bitmap: 00 |

- Os bitmaps permitem minimizar os *overheads* na alocação de blocos pequenos (mais frequentes)
- As desvantagens do esquema são
 - Existem zonas alocadas que não são utilizadas (caso dos 32 bytes para 17 bytes pedidos)
 - A alocação de blocos grandes é lenta

Fragmentação Externa vs Interna

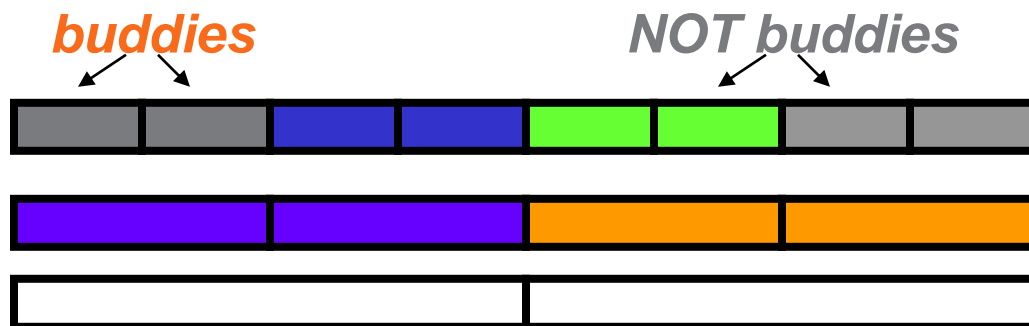
- Com o *slab allocator*, a diferença entre o tamanho requisitado e a potência de 2 mais próxima faz com que se desperdice muito espaço
 - e.g., se o programa quer alocar 17 bytes e nós damos 32 bytes, então há 15 bytes que não são utilizados
- Repare que isto não é *fragmentação externa*. A fragmentação externa refere-se ao espaço desperdiçado entre blocos alocados.
- Este problema é conhecido por *fragmentação interna*. Trata-se de espaço desperdiçado dentro de um bloco já alocado.

Buddy System (1/2)

- Outro sistema de gestão de memória usado no *kernel* do Linux.
- É semelhante ao “*slab allocator*”, mas só aloca blocos em tamanhos que são potência 2 (fragmentação interna é ainda possível)
- Mantém *free-lists* separadas para cada tamanho
 - e.g., listas separadas para 16 byte, 32 byte, 64 byte, etc.

Buddy System (2/2)

- Se não há um bloco de tamanho n disponível, então procura um bloco de tamanho $2n$ e divide-o em dois blocos de tamanho n
- Quando o bloco de tamanho n é libertado, então, se o vizinho (buddy) estiver também livre, os dois são combinados num bloco de $2n$



- Tem as mesmas vantagens de velocidade que o *slab*

Esquemas de alocação

- Qual destes sistemas é o melhor?
 - Não existe um esquema que seja melhor para toda e qualquer aplicação
 - As aplicações têm diferentes padrões de alocação/libertação de memória
 - Um esquema que funcione bem para uma aplicação, poderá não funcionar bem para outra

Gestão automática de memória

- É difícil gerir e manter registos das alocações/libertações de memória – porque não tentar fazê-lo de forma **automática**?
- Se conseguirmos saber em cada instante de *runtime* os blocos da *heap* que estão a ser usados, então todo o espaço restante está livre para alocação.
 - A memória que não está a ser apontada chama-se **garbage** (é impossível aceder-lhe). O processo de a recuperar chama-se **garbage collection**. No C a recuperação/libertação de memória tem de ser feita manualmente
- Como conseguimos saber o que está a ser usado?

Manter registo da memória utilizada

- As técnicas dependem da linguagem de programação utilizada e precisam da ajuda do compilador.
- Pode começar-se por manter registo de todos os ponteiros, definidos tanto como variáveis globais ou locais (root set). (para isto o compilador tem de colaborar)
- **Ideia Chave:** Durante o *runtime* mantém-se o registo dos objectos dinâmicos apontados por esses ponteiros.
 - À partida, um objecto que não seja apontado por ninguém é *garbage* e pode ser desalocado.

Manter registo da memória utilizada

- Mas o problema não é assim tão simples ...
 - O que é que acontece se houver um type cast daquilo que é apontado pelo ponteiro? (permitido pelo C)
 - O que acontece se são definidas variáveis ponteiro na zona alocada?
- A pesquisa de garbage tem de ser sempre feita de forma recursiva.
- Não é um mecanismo simples e envolve sempre maiores overheads do que a gestão manual
- Os "Garbage Collectors" estão fora do nosso programa, mas os alunos interessados poderão consultar material suplementar.

Concluindo ...

- O C tem 3 zonas de memória
 - Armazenamento estático: variáveis globais
 - A Pilha: variáveis locais, parâmetros, etc
 - A heap (alocação dinâmica): `malloc()` aloca espaço, `free()` liberta espaço.
- Várias técnicas para gerir a heap via `malloc` e `free`: best-, first-, next-fit
 - 2 tipos de fragmentação de memória: interna e externa; todas as técnicas sofrem com pelo menos uma delas
 - Cada técnica tem pontos fortes e fracos, e nenhuma é melhor para todos os casos
- A gestão automática de memória liberta o programador da responsabilidade de gerir a memória. O preço é um maior *overhead* durante a execução.

Para saber mais ...

- Artigo a explicar a divisão de memória no C (dividem a zona estática em inicializada e não inicializada)
 - <http://www.informit.com/articles/article.aspx?p=173438>
- A Wikipedia ao nosso serviço
 - http://en.wikipedia.org/wiki/Dynamic_memory_allocation
 - [http://en.wikipedia.org/wiki/Garbage_collection_\(computer_science\)](http://en.wikipedia.org/wiki/Garbage_collection_(computer_science))
 - http://en.wikipedia.org/wiki/Memory_management