

Alocador de memória

Este projeto faz parte do [PPOS v2](#).

Este projeto visa construir um alocador de memória *heap* para o PPOS. Com ele funcionando, não precisaremos mais usar o alocador da biblioteca C (`malloc/free`).

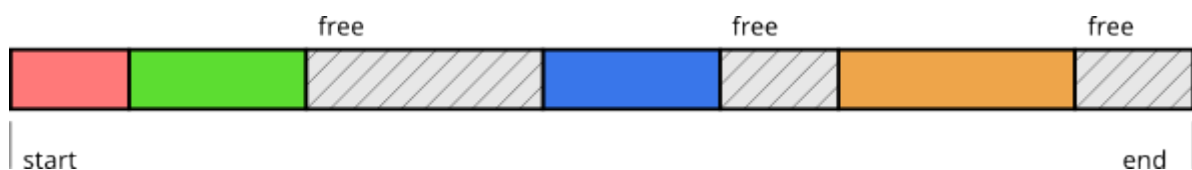
Estrutura geral

A área de memória *heap*, usada para alocação dinâmica, será representada em nosso SO por um grande vetor estático de bytes, a ser definido em `memory.c`:

```
// tamanho total da heap: 64 MB
#define HEAP_SIZE 64*1024*1024

static char heap[HEAP_SIZE];
```

Essa área de memória poderá ser alocada pelas aplicações em blocos de tamanho variável, usando o algoritmo de alocação *first-fit*, que aloca sempre o primeiro bloco de memória livre com tamanho suficiente. A figura abaixo mostra a organização da memória *heap*, com blocos alocados e blocos livres:



Para gerenciar os blocos de memória (livres ou alocados), sugere-se definir um vetor de *descritores de bloco* contendo, para cada bloco:

- endereço inicial do bloco
- tamanho do bloco em bytes
- estado do bloco (livre ou alocado)
- número do bloco vizinho da esquerda
- número do bloco vizinho da direita



Observe que neste projeto **não podemos usar** uma lista de descritores dinâmica, nem a biblioteca de filas genéricas desenvolvida anteriormente. Ambas precisam da alocação dinâmica de memória *heap*, que é justamente o objetivo deste projeto. A abordagem mais simples neste caso é definir um vetor estático de descritores de blocos.

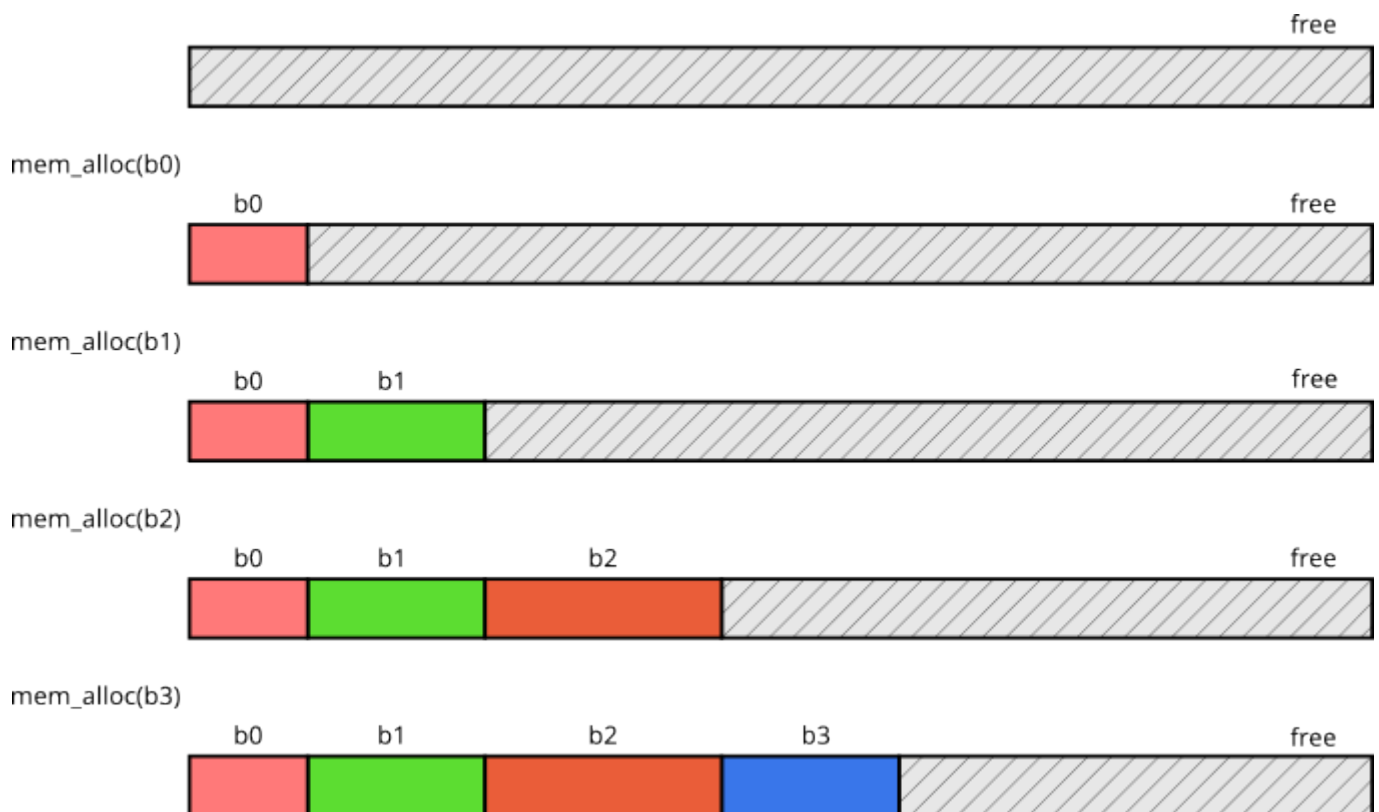
Algoritmo de alocação

A **alocação** de um novo bloco usa a seguinte estratégia:

1. ajusta o tamanho solicitado para um múltiplo de 16 bytes
2. na lista de blocos livres, procura um bloco com tamanho suficiente (*first-fit*)
3. se não encontrar, retorna NULL (alocação falhou)
4. se o tamanho do bloco livre encontrado for maior que o necessário, divide-o em dois blocos:
 - um bloco a ser alocado, com o tamanho solicitado

- um bloco livre, com o restante do tamanho anterior
- 5. marca o bloco como ocupado
- 6. retorna um ponteiro para o início do bloco

A figura a seguir ilustra a alocação consecutiva de 4 blocos de memória, em um *heap* inicialmente livre:

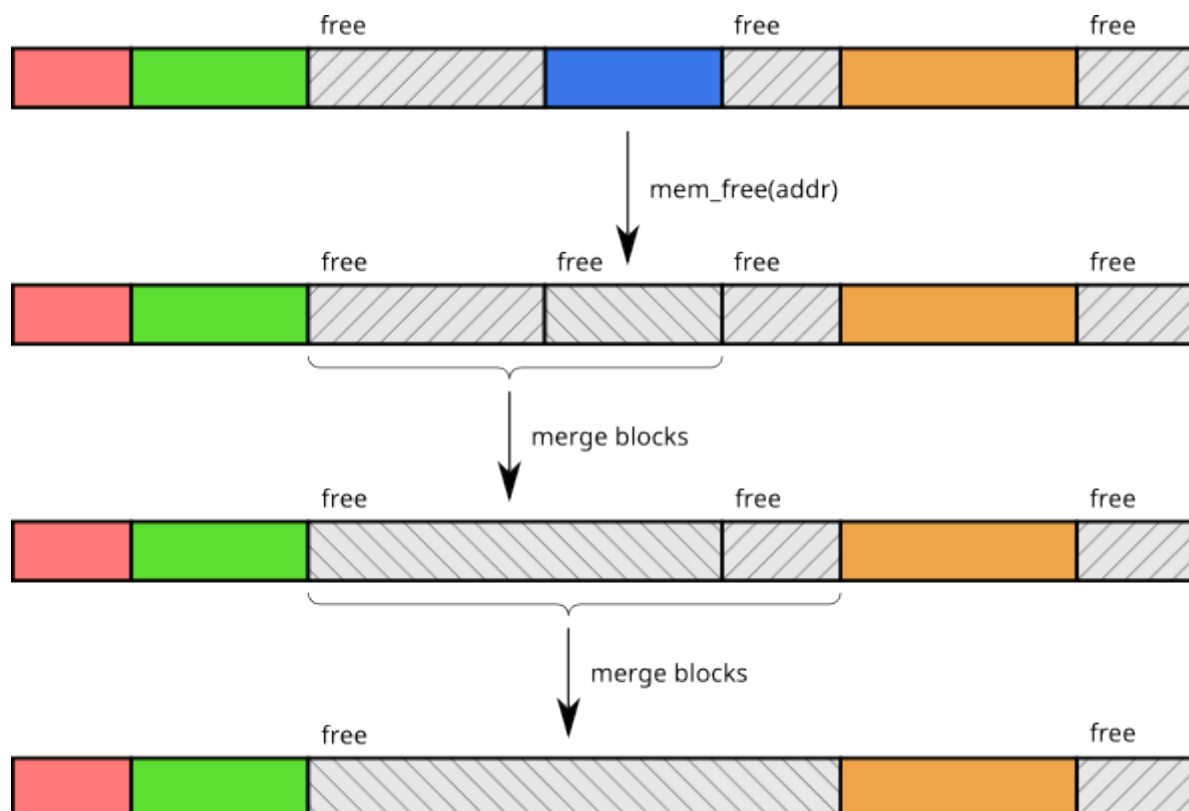


A **liberação** de um bloco usa a seguinte estratégia:

1. localiza o número do bloco alocado a partir do endereço fornecido
2. se não encontrar o bloco, retorna erro
3. se o bloco já estiver livre, retorna erro
4. marca o bloco como livre
5. se o bloco à sua esquerda também estiver livre, funde-os em um só bloco
6. se o bloco à sua direita também estiver livre, funde-os em um só bloco

Ao liberar um bloco, deve-se fazer a **coalescência** dos blocos livres: se um dos vizinhos estiver livre, o bloco recém-liberado e seu vizinho livre devem ser fundidos em um bloco livre maior, **recursivamente**. Esse procedimento permite obter blocos livres maiores, diminuindo a fragmentação da memória.

A coalescência é ilustrada no exemplo abaixo:



Funções a implementar

As seguintes funções, com suas respectivas estruturas de dados de apoio, devem ser implementadas em `memory.c`:

Inicia a memória

```
void mem_init()
```

Inicia o subsistema de memória, preparando as estruturas de dados de controle do mesmo (tabelas/listas de descritores, etc.).

Aloca memória

```
void *mem_alloc(int size)
```

Aloca um bloco de memória com o tamanho indicado em bytes; retorna um ponteiro para a área alocada ou NULL se houver erro.

Cada bloco alocado deve ter um tamanho mínimo de 16 bytes e deve ser alinhado em 16 bytes, ou seja, seu tamanho e endereço inicial devem ser múltiplos de 16. O ajuste de um valor para o próximo múltiplo de 16 pode ser feito assim:

```
// versão mais rápida
value = ((value - 1) | 0x00000F) + 1;

// versão mais simples
while (value % 16)
    value++ ;
```

Libera memória

```
int mem_free(void *addr)
```

Libera um bloco de memória previamente alocado, indicado pelo endereço `addr`; retorna 0 se executar sem problemas ou -1 se o ponteiro for NULL ou indicar uma área não-alocada.

Memória total

```
int mem_size()
```

Informa a quantidade de memória total, em bytes.

Memória livre

```
int mem_avail()
```

Informa a quantidade total de memória disponível, em bytes. Não se preocupa se essa memória livre está fragmentada ou não.

Informa sobre o uso da memória

```
void mem_report()
```

Imprime um relatório sobre o uso atual da memória, com o seguinte formato:

```
heap: 435 KB allocated (16 blocks), 65100 KB free (1 blocks)
heap: block    0: 0x0000565154c56340 - 0x0000565154c5644f alloc prev    16 next
1 size 272
heap: block    1: 0x0000565154c56450 - 0x0000565154c5646f alloc prev     0 next
2 size 32
heap: block    2: 0x0000565154c56470 - 0x0000565154c5648f alloc prev     1 next
3 size 32
heap: block    3: 0x0000565154c56490 - 0x0000565154c5659f alloc prev     2 next
4 size 272
heap: block    4: 0x0000565154c565a0 - 0x0000565154c5e59f alloc prev     3 next
5 size 32768
heap: block    5: 0x0000565154c5e5a0 - 0x0000565154c5e5bf alloc prev     4 next
6 size 32
heap: block    6: 0x0000565154c5e5c0 - 0x0000565154c6d91f alloc prev     5 next
7 size 62304
heap: block    7: 0x0000565154c6d920 - 0x0000565154c7025f alloc prev     6 next
8 size 10560
heap: block    8: 0x0000565154c70260 - 0x0000565154c7f94f alloc prev     7 next
9 size 63216
heap: block    9: 0x0000565154c7f950 - 0x0000565154c84c8f alloc prev     8 next
10 size 21312
heap: block   10: 0x0000565154c84c90 - 0x0000565154c9319f alloc prev     9 next
11 size 58640
heap: block   11: 0x0000565154c931a0 - 0x0000565154c9f82f alloc prev    10 next
12 size 50832
```

```
heap: block 12: 0x0000565154c9f830 - 0x0000565154ca645f alloc prev 11 next
13 size 27696
heap: block 13: 0x0000565154ca6460 - 0x0000565154cac40f alloc prev 12 next
14 size 24496
heap: block 14: 0x0000565154cac410 - 0x0000565154cb82ff alloc prev 13 next
15 size 48880
heap: block 15: 0x0000565154cb8300 - 0x0000565154cc321f alloc prev 14 next
16 size 44832
heap: block 16: 0x0000565154cc3220 - 0x0000565158c5633f FREE prev 15 next
0 size 66662688
```

Cada entrada do relatório deve informar:

- número do bloco
- endereços inicial e final
- status (livre ou ocupado)
- vizinho à esquerda e à direita
- tamanho em bytes

Observações:

Após a implementação do alocador, você deve usá-lo em todo o seu projeto. Nos arquivos `kernel/*.c` e `lib/*.c` você deve:

- trocar todas as ocorrências de `malloc` por `mem_alloc`
- trocar todas as ocorrências de `free` por `mem_free`
- remover todas as inclusões de `stdlib.h`

Sua implementação deve funcionar corretamente com o programa de teste `test/pingpong-mqueue.c`. Esse programa vai usar intensivamente a alocação de memória na criação e destruição de filas genéricas, tarefas, semáforos e filas de mensagens.

Arquivos

Os seguintes arquivos são relevantes para este projeto:

- `kernel/memory.h`: interface da gestão de memória
- `kernel/memory.c`: implementação gestão de memória
- `test/pingpong-memory.c`: programa de teste simples
- `test/pingpong-memory.txt`: saída esperada do teste
- `test/pingpong-mqueue.c`: teste mais complexo

Outras informações

- Duração estimada: 6 horas.
- Dependências:
 - [Tarefas cooperativas](#)
 - [Despachante de tarefas](#)
 - [Preempção por Tempo](#)
 - [Tarefas que esperam](#)
 - [Tarefas que dormem](#)
 - [Spinlocks e Semáforos](#)

From:

<https://wiki.inf.ufpr.br/maziero/> - **Prof. Carlos Maziero**

Permanent link:

https://wiki.inf.ufpr.br/maziero/doku.php?id=ppos-v2:alocador_de_memoria

Last update: **2026/07/31 11:12**

