Escola Politécnica da Universidade de São Paulo

PCS 3732: Laboratório de Processadores

3º Módulo Acadêmico – 5º Quadrimestre – 2023

Malloc-eiro: Gerenciador de memória dinâmica

Eduardo Hiroshi Ito NºUSP: 11806868

Gabriel Coutinho Ribeiro NºUSP: 11803437

Ricardo Tamay Honda NºUSP: 11803778

SUMÁRIO

1 INTRODUÇÃO

Este projeto consiste na implementação de um gerenciador de memória dinâmica no nível do sistema operacional. Ou seja, ele lida diretamente com os endereços físicos da memória, sem a camada de virtualização e tradução de endereço feita pelo sistema operacional entre os programas usuários e a memória em si.

Como a alocação de memória é feita majoritariamente para endereços localizados na heap, então o alocador desenvolvido gerencia e aloca as posições dessa região de memória.

1.1 Disponibilização

Ambos o repositório do projeto da Prova II e o vídeo no YouTube de uma apresentação mais detalhada podem ser encontrados nos seguintes links:

Repositório do projeto no GitHub;
Vídeo de apresentação;
Backup do vídeo no Google Drive.

1.2 Algoritmo utilizado

O projeto foi desenvolvido com base na implementação de alocação de páginas físicas, mais especificamente na praticada no Linux, que implementa o algoritmo *Binary Buddy Allocator*. Ele pode ser encontrado no seguinte link, sendo o algoritmo desenvolvido por Knowlton e mais profundamente por Knuth. Esse algoritmo é implementado pelo Linux nas funções de alocação de memória internas ao kernel; ou seja, essas não são as funções utilizadas por programas usuário para alocar memória; na realidade, a biblioteca stdlib em C realiza essa função, sendo que a única chamada feita ao sistema operacional é o sbrk, que lida com a expansão e diminuição do heap.

O seu funcionamento pode ser descrito com as seguintes etapas principais, sendo alguns detalhes/etapas presentes na referência sendo omitidas desta explicação:

- Gerenciamento de blocos livres;
- Alocação de páginas;
- Liberação de páginas.

Os passos individuais pertinentes a cada etapa serão descritas mais detalhadamente em seções posteriores do relatório.

Antes disso, no entanto, é necessário definir o que é a ordem de um bloco: é dito que um bloco é de ordem n se esse tem tamanho de 2^n páginas. Sendo assim, esse algoritmo quebra a memória em blocos de tamanho, em páginas, potências de 2. Se um bloco do tamanho/ordem desejada não está disponível, um bloco maior é dividido pela metade e os dois novos blocos são *buddies* (companheiros) um do outro. Um deles é usado para atender ao pedido de alocação e o outro é mantido livre. Os blocos são continuamente divididos até que um bloco do tamanho desejado fique disponível. O contrário deve ocorrer na liberação do bloco: quando um bloco é liberado e seu *buddy* também está livre, ambos devem ser fundidos/juntados.

1.2.1 Gerenciamento de blocos livre

Os blocos livres podem assumir um tamanho de 2^n páginas, sendo *n* a ordem do bloco. A estrutura de gerenciamento dos blocos livres é constituída de listas ligadas que guardam separadamente uma lista de blocos livres de cada ordem, bem como um vetor de tamanho MAX_ORDER (maior ordem implementada) posições para armazenar um ponteiro para cada uma dessas listas ligadas, como mostra a imagem abaixo:

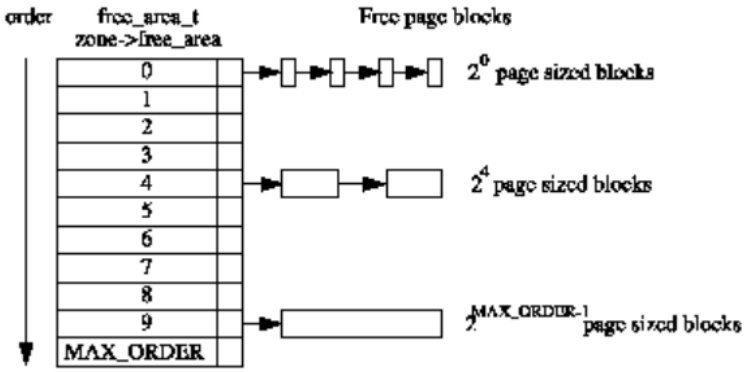

Imagem 1 - Gerenciamento de blocos de páginas livres

Portanto, o elemento 0 do vetor irá apontar para uma lista de blocos livres de tamanho de 2^0 ou 1 pégina, o elemento 1 para uma lista com blocos de 2^1 (2) páginas até blocos de tamanho $2^{MAX\_ORDER-1}$ páginas. Assim, é possível encontrar de maneira simples um bloco de tamanho necessário e suficiente para um pedido de alocação, evitando a utilização de blocos de ordens maiores que o necessário.

A memória, então, é mapeada pelo vetor de struct chamado free_area[MAX_ORDER]. A seguinte struct representa a estrutura de dados para gerenciar a memória:

```
22 typedef struct free_area_struct {
23         struct list_head        free_list;
24         unsigned long           *map;
25 } free_area_t;
```

Imagem 2 - Estrutura de dados para o gerenciamento de memória

Onde:

`free_list` é a lista ligada de blocos livres de uma ordem específica;

`map` é o mapa de bits com o estado dos *buddies.*

No mapa de bits, somente um bit é usado para representar o estado de um par de *buddies*. Cada vez que um *buddy* é alocado ou liberado, o bit representando o par de *buddies* é setado de forma que seja 0 caso ambos os blocos tenham o mesmo estado (ambos alocados ou livres) e 1 caso tenham estados diferentes (somente um deles esteja alocado).

A forma mais simples de calcular o bit correto é utilizando o índice da primeira página do bloco no mapa de memória global. Ao deslocar para a direita esse índice em `1+order` bits, o bit no mapa de bits representando o par é revelado.

1.2.2 Alocação de páginas

As alocações de página são sempre feitas para uma ordem específica, ou seja, sempre é necessário requisitar a ordem de bloco desejada para o algoritmo. Se não houver nenhum bloco livre da ordem desejada, um bloco de maior ordem é dividido em dois *buddies*. Um deles é alocado e o outro é posto na fila de blocos livre para a devida ordem (a ordem dos *buddies*).

A figura a seguir mostra um exemplo em que um bloco de ordem 2 é requisitado, mas não existem blocos de ordem 2 ou 3 livres (listas de livre dessas ordens vazias; sendo assim, um bloco de ordem 4 é dividido e os *buddies* criados são adicionados às listas das menores ordens até que um bloco de ordem 2 esteja disponível.

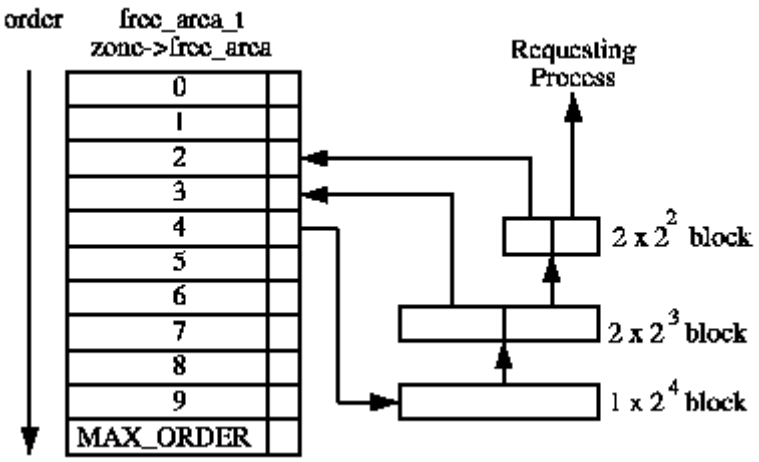

Imagem 3 - Alocação de páginas

Quando o bloco for posteriormente liberado, o status do seu *buddy* será checado. Se ambos estiverem livres, eles são combinados/juntados, formando um novo bloco de maior ordem, e são colocados na lista de livres dessa maior ordem, onde novamente é verificado o status do *buddy* do novo bloco. Esse processo se repete até que uma de duas coisas ocorra: um *buddy* não esteja livre ou o bloco

criado seja da maior ordem implementada. Em ambos os casos, o bloco liberado é adicionado à lista de livres da sua ordem.

1.2.3 Liberação de páginas

Por fim, para a liberação de páginas, é necessário que o alocador tenha conhecimento do tamanho de uma alocação previamente realizada, assim, para a liberação, é essencial relembrar a ordem original dos blocos alocados.

Quando um bloco é liberado, ocorre a tentativa de juntar o bloco e seu *buddy*. Para detectar se esses blocos podem ser combinados, verifica-se o bit referente ao par de *buddies* em `free_area → map`. Como um dos *buddies* acabou de ser liberado pela função, é óbvio que ao menos um *buddy* está livre. Assim, se for verificado o bit no mapa e ele for 0 após inverte-lo, o outro *buddy* também estará livre, já que o bit 0 indica que ambos ou estão alocados, ou estão liberados. Se ambos estão livres, eles podem ser mesclados/combinados.

Para calcular o endereço do *buddy*, como as alocações são sempre em blocos de tamanho 2^k, o endereço do bloco também será uma potência 2^k, ou, ao menos, seu offset (deslocamento) no mapa de memória será uma potência 2^k.

Destarte, sempre haverá ao menos um número k de zeros a direita do endereço, fazendo com que a verificação do k-ésimo bit à direita seja essencial para se obter o endereço do buddy. Caso esse bit seja 0, o bit respectivo do endereço do *buddy* será 1, e vice-versa.

Com o intuito de obter o bit, é possível criar uma máscara para o k-ésimo bit do endereço.

Uma vez tendo os *buddies* mesclados, o *buddy* anteriormente na lista é removido, sendo o par recentemente combinado movido para uma ordem superior com o intuito de verificar se ele pode ser também mesclado com seu *buddy*.

Em suma, a liberação funciona na ordem inversa da alocação.

2 IMPLEMENTAÇÃO

Para implementar um alocador com o algoritmo descrito na seção anterior, então, o desenvolvimento, da mesma forma que a descrição do algoritmo, foi feito em 3 partes principais: inicialização das estruturas necessárias para gerenciar os blocos (vetor de `structs free_area_t`), alocação de blocos e liberação de blocos.

2.1 Inicialização do vetor da struct

Para o gerenciamento da região de memória para alocação dinâmica (heap) foi definida em malloc.h, então, a struct descrita no algoritmo:

```
typedef struct free_area_struct {
    struct list_addr free_list;
    unsigned long    *map;
} free_area_t;
```

Sendo list_addr um simples lista duplamente ligada, definida em malloc.h como segue:

```
struct list_addr {
    struct list_addr *prev, *next;
};
```

Decidiu-se que, como o algoritmo não possui, uma lista de livres ou mapa de bits para os blocos de ordem MAX _ORDER, então, os maiores blocos que se teria seriam de ordem MAX_ORDER-1 e, como descrito no algoritmo, o vetor de estruturas teria tamanho MAX_ORDER (indo de 0 a MAX_ORDER-1).

A função de inicialização foi definida com a seguinte assinatura em malloc.h:

```
void initialize(uint32_t heap_init, uint32_t end_heap, uint8_t
zero);
```

Ela utiliza os endereços inicial e final da heap para calcular o tamanho dessa área da memória e armazena esses valores em variáveis globais para uso posterior. Isso pode ser visto no seguinte trecho da função em malloc.c:

```
uint32_t size_in_bytes = end_heap - heap_init; // heap size
    uint32_t i = 0;
    struct list_addr *current;
    uint8_t order = 0;

    heap_start = heap_init;
    heap_end = end_heap;
    zero_page = zero;
```

O parâmetro zero será explicado posteriormente.

Essa função é chamada no código de inicialização do kernel em boot.s:

```
start:
    ldr r0, =start_heap
    ldr r1, =end_heap
    mov r2, #0
    bl initialize
```

Sendo que os endereços **start_heap** e **endheap** são definidos no linker script kernel.ld:

```
  start_heap = .;
  . = . + HEAP_SIZE;
  . = ALIGN(8);
  end_heap = .;
```

Sendo o tamanho do heap definido, em kernel.ld, como:

```
HEAP_SIZE = 0x4000000;
```

Um problema importante ao inicializar e armazenar o vetor de free_area_t é que esse vetor ocuparia um tamanho que, num programa de usuário, utilizaria alocação dinâmica para essa estrutura. No entanto, isso não é possível no nível de sistema operacional, o que permite que a estrutura seja sobrescrita acidentalmente se o espaço devido não for alocado.

Para solucionar isso, existem duas possibilidades: definir o tamanho das estruturas em tempo de compilação ou utilizar espaços da memória que certamente não estão sendo utilizados por outras funções ou programas.

2.1.1 Mapas de bits

No caso dos mapas de bits de cada ordem, foi decidido por alocar, em tempo de compilação, um espaço para todos os mapas. Sendo assim, decidiu-se que haveria um único mapa de bits contíguo, com todos os mapas de bits de cada ordem justapostos na memória. Dessa forma, nas estruturas de cada uma das ordens, o valor do mapa de bits seria preenchido com o endereço da primeira palavra do mapa de bits da devida ordem.

Então, para calcular o tamanho, em bits, necessário para os mapa de bits combinado, deduziu-se a fórmula para o tamanho de um mapa de bits de ordem **order**:

```
Bitmap_size(order) = HEAP_SIZE/(PAGE_SIZE*(1<<(order+1))
```

Tal fórmula advém do fato de que, ao dividir-se o tamanho do heap pelo tamanho de 1 página, tem-se o tamanho em páginas da heap. Entretanto, para saber quantos blocos de uma certa ordem existem na memória, o número de páginas encontrado deve ser dividido pelo tamanho do bloco em página: 2^{ordem}. Dessa forma, tem-se quantos blocos de certa ordem existem no tamanho de heap em questão. Disso, sabe-se que o mapa de bits possui um bit para um par de blocos, portanto, o tamanho do mapa de bits será o número de blocos dividido por 2. Isso condiz com a fórmula apresentada.

Continuando, para saber o tamanho total, em bits, do mapa de bits combinado, deduziu-se a fórmula para o essa tamanho a a partir da soma do tamanho de cada ordem, de 0 até MAX_ORDER-1:

$$\sum_{0}^{MAX\_ORDER-1}(HEAP\_SIZE/(PAGE\_SIZE*(1<<(order+1))) =$$

$$= HEAP\_SIZE/(PAGE\_SIZE*2) * \sum_{0}^{MAX\_ORDER-1}(1/2^{order}) =$$

$$= HEAP\_SIZE/(PAGE\_SIZE*2) * 2*(1-1/2^{MAX\_ORDER}) =$$

$$= HEAP\_SIZE/PAGE\_SIZE*(1-1/(1 << MAX\_ORDER))$$

Ainda, observa-se que ambas as fórmulas calculam o tamanho em bits dos mapas de bits. Então, para calcular o tamanho total em longs, é necessário dividir os valores encontrados pelo tamanho, em bits, de uma palavra long.

Disso, foram feitas as seguintes definições em size.h e malloc.h:

```
#define WORD_SIZE (8*sizeof(unsigned long))

#define BITS_PER_LONG WORD_SIZE

#define MAX_PAGES (HEAP_SIZE/PAGE_SIZE)
```

```
#define first_index_in_bitmap_of_order(order)
(MAX_PAGES-MAX_PAGES/(1UL << (order)))
#define first_word_in_bitmap_of_order(order)
(first_index_in_bitmap_of_order(order)/BITS_PER_LONG)
```

Das definições acima, vê-se que foi assumido um tamanho padrão para o heap, o que, num código de um sistema operacional, seria incorreto e seria previamente definido e configurado. Mas, para fins do projeto, esse tamanho foi assumido como 64MB, com um tamanho de página de 4096 B.

Dessa forma, foi possível declarar um mapa de bits global, em malloc.c, como segue:

```
unsigned long bitmaps[first_word_in_bitmap_of_order(MAX_ORDER) +
1];
```

Esse mapa tem o tamanho da fórmula calculada somado de 1 porque blocos de ordens maiores que 8 utilizam menos de 32 bits (menos que um long) no caso do heap de tamanho escolhido, então, é necessário ao menos mais uma palavra para o mapa.

Então, para inicializar propriamente os mapas de bits com os endereços corretos no mapa de bits global, o seguinte trecho de código, em malloc.c:

```
for (order = 0; order < MAX_ORDER; order++) {
```

```
            // Initialize the bitmap for this order
            uint32_t index = first_word_in_bitmap_of_order(order);
            if (index == MAX_ORDER-1) index++; // last one needs only
half a long (32 bits), for the 64 MB heap
            free_area[order].map = &bitmaps[index]; // set the right
addres for each one

            uint32_t map_size = bitmap_size_for_order(order);
            map_size = map_size == 0 ? 1 : map_size;
            for (i = 0; i < map_size; i++) {
                free_area[order].map[i] = 0;
            }
```

Ou seja, para cada ordem é calculado o primeiro índice no mapa de bits global **bitmaps** e o endereço desse índice é salvo no correspondente endereço do mapa para aquela ordem. Para calcular o primeiro índice, é necessário somente calcular a soma acumulada dos tamanhos dos mapas de ordens menores, como visto no código.

Ainda, da fórmula inicial mostrada, o tamanho em palavras de um mapa de bits foi definido, em malloc.h, como:

```
    #define bitmap_size_for_order(order) ((MAX_PAGES/(1UL <<
(order+1)))/BITS_PER_LONG)
```

Esse tamanho é utilizado para que, a partir do endereço obtido, o mapa daquela ordem seja inteiramente limpo, com todas as palavras sendo sentadas em 0.

2.1.2 Lista de blocos livres

No caso da lista de blocos livres, decidiu-se por posicionar cada nó das listas nas primeiras duas palavras de cada bloco. Isso seria possível porque um bloco só estaria em uma lista se estivesse livre e, portanto, suas posições de memória não estariam sendo usadas por nenhum outro processo; assim seria possível utilizá-las sem que fossem sobrescritas. Além disso, um bloco nunca está em mais de uma lista simultaneamente; sendo assim, um bloco conterá somente o seu nó, que estará em somente uma das listas dentre as várias ordens.

Assim, a lista na estrutura free_area seria a cabeça da lista para aquela ordem. Como a lista é duplamente ligada, cada cabeça foi inicializada para uma lista vazia, em que a cabeça aponta para si mesma:

```
    current = &free_area[order].free_list;
            current->prev = current;
            current->next = current;
```

Então, para que o algoritmo funcione corretamente, é necessário que somente a lista da maior ordem implementada (MAX_ORDER-1) esteja inicialmente preenchida, para que, a partir dela, os blocos possam ser subdivididos se necessário. Por isso, tal lista é inicializada da seguinte forma:

- A partir do tamanho máximo do bloco, dado por 2^{MAX_ODER-1} * PAGE_SIZE, é calculado o número de blocos dessa ordem que existem no tamanho de heap dado;
- Com isso, a partir do endereço inicial do heap, é calculado o endereço de cada bloco e, a cada endereço calculado, a lista é formada
- O endereço de cada bloco é calculado em passos de tamanho máximo do bloco, calculado anteriormente

Isso é implementado então, em malloc.c, da seguinte forma:

```
for (i = 0, current = &free_area[MAX_ORDER-1].free_list; i <
size_in_bytes/MAX_BLOCK_SIZE; i++, current = current->next) { //
filling only the max order array (max block size)
        if (i == 0) {
            current->prev = current; // If current->prev == current,
its the head of the list
        }
        else if (i == 1) {
            current->prev = &free_area[MAX_ORDER-1].free_list; // head
        }
        else {
            current->prev = (struct list_addr *) (heap_init + (i-2) *
MAX_BLOCK_SIZE);
        }
        current->next = (struct list_addr *) (heap_init + i *
MAX_BLOCK_SIZE);
```

```
        if (i == size_in_bytes/MAX_BLOCK_SIZE-1) { // last node
            current = current->next;
            current->prev = (struct list_addr *) (heap_init + (i-1) *
MAX_BLOCK_SIZE);
            current->next = current; // End of list, points to itself
        }
    }
```

Do código, vê-se que, em cada bloco calculado, um nó da lista é criado de forma que o nó no bloco de índice i-1 aponte para os blocos de índice i-2 e i, formando assim a lista para os blocos de maior ordem.

2.2 Alocação de um bloco

A assinatura da função de alocação é a seguinte:

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

Por conta disso, a ordem do bloco não é imediatamente conhecida e deve ser calculada para que o algoritmo de *buddies* possa ser utilizado.

Para isso, primeiramente verifica-se se o tamanho desejado, em bytes, é menor que o tamanho de uma página. Caso seja, o bloco a ser alocado é de ordem 0. Caso o bloco seja maior que o tamanho máximo de páginas, não é possível fazer a alocação e é retornado um ponteiro nulo.

No caso de serem necessárias 2 ou mais páginas para comportar o tamanho requisitado, sem ultrapassar o tamanho máximo de página, é feita a verificação da ordem necessária para tal.

Assim, o código de cálculo da ordem é o seguinte:

```
    if (size <= PAGE_SIZE) {
            order = 0;
            p = (void *) get_free_page();
        }
        else if (size > MAX_BLOCK_SIZE) { // can't give one
MAX_ORDER-1 block
            return (void *)0UL;
        }
        else {
            order = 1;
```

```
            block_size = (PAGE_SIZE * (1UL << order));
            // calculate the minimum order necessary
            while (order < MAX_ORDER-1 && (size / block_size > 1 ||
(size / block_size == 1 && size % block_size > 0)))
            {
                order++;
                block_size = (PAGE_SIZE * (1UL << order));
            }
            p = (void *) get_free_pages(order);
        }
```

Esse código visa calcular a menor ordem necessária para um tamanho requisitado; sendo assim, são considerados os casos em que exatamente 2^k páginas são necessárias e, então, um bloco de ordem k é desejado.

Do código acima, vê-se que existem duas funções diferentes para requisitar páginas. Entretanto, ambas fazem uma chamada à função **allocate_pages**, com a diferença que uma delas requisita blocos de ordem 0 e a outra não.

A função **allocate_pages**, então, é responsável por verificar qual lista da menor ordem possível, a partir da ordem desejada, possui ao menos um bloco livre. Quando encontrado, se esse bloco não é da ordem desejada, então a função **get_block_from_order** é chamada, para que os blocos sejam divididos e um bloco da ordem desejada seja retornado. Ainda, essa função inverte o bit no mapa de bits correspondente ao bloco recebido, que tem a ordem requisitada. O trecho de código que realiza essa função é disposto a seguir:

```
    for (current_order = order; current_order < MAX_ORDER;
current_order++) {
            current_list = &free_area[current_order].free_list;
            if (!empty_list(current_list)) {
                block_addr = (unsigned long)
pop_first_element(current_list); // found a block
                if (current_order != order) { // Not in the wanted
order
                    block_addr = get_block_from_order(order,
current_order, block_addr); // divide it until the wanted order
                }
                MARK_USED((block_addr-heap_start)/PAGE_SIZE, order,
&free_area[order]); // Invert bit for the actual allocation
                return block_addr;
```

```
            }
        }
```

A função que implementa a divisão de um bloco de ordem em pares de *buddies* até que um bloco da ordem desejada seja obtido, de acordo com o algoritmo descrito na seção 1.2.2, é:

```
    unsigned long get_block_from_order(size_t wanted_order, size_t
found_block_order,unsigned long block_addr) {
        unsigned long return_addr;
        uint32_t i;

        for (i = 0; i < found_block_order-wanted_order; i++) {
            return_addr = block_addr +
BLOCK_SIZE(found_block_order-i)/2; // calculates the buddy for this
block in the next low order
            if (found_block_order-i != MAX_ORDER-1) { // invert the
bit when my block is divided
                MARK_USED((block_addr-heap_start)/PAGE_SIZE,
found_block_order-i, &free_area[found_block_order-i]);
            }
            put_element(&free_area[found_block_order-i-1].free_list,
(struct list_addr *)block_addr); // puts the block in the next low
order free_list
            block_addr = return_addr;
        }

        return return_addr;
    }
```

Nesse código, vê-se que, a cada iteração, o bloco de certa ordem é dividido pela metade em dois blocos de menor ordem, seu bit no mapa de sua ordem original é invertido, indicando que ele foi alocado, o primeiro dos dois *buddies* criados é posto na lista de livres de sua ordem, menor do que a original e, por último, o outro *buddy* fica disponível para ser alocado e/ou separado novamente. Se o novo *buddy* for da ordem desejada, a função retorna seu endereço, efetivamente alocando-o, caso contrário, o processo se repete para o novo *buddy*, assim como determina o algoritmo.

Observa-se que, se o bloco encontrado for da ordem MAX_ORDER-1, não é necessário manipular seu mapa de bits pois blocos dessa ordem não são combinados, então, não é necessário gerenciar os *buddies* nessa ordem.

Em suma, a partir dos códigos de alocação mostrados, vê-se que o algoritmo de alocação de *buddies* está implementado: quando um bloco de certa ordem é requisitado, procura-se pelo bloco de ordem superior mais próxima da desejada e, caso não seja dessa ordem, o bloco é dividido e um dos *buddies* é retornado à respectiva lista de livres até que um bloco da ordem desejada seja alocado.

A definição de MARK_USED, usada para inverter um bit em um mapa de bits com base no endereço relativo do bloco no heap e sua ordem, será explicado numa seção posterior.

2.3 Liberação de um bloco

Como dito anteriormente, na liberação de um bloco, somente o endereço desse bloco é passado à função de liberação. Sendo assim, o alocador deve, de alguma forma, ter um registro da ordem do bloco quando foi alocado. No caso do projeto, decidiu-se armazenar, na primeira palavra do bloco, a ordem do bloco alocado. Para isso, foi necessário retornar, na função malloc, não o endereço do bloco, mas sim o endereço somado do tamanho do header. Assim, o retorno da função malloc foi alterado para o seguinte:

```
          block_header = p;
          block_header->order = order;
          return p + sizeof(struct block_order) + (4 - sizeof(struct
block_order) % 4);
```

Ou seja, a ordem é salva no bloco alocado e seu endereço somado ao tamanho do header é retornado.

Assim, na função de liberação, é possível recuperar o endereço e ordem originais do bloco:

```
          block_addr = addr - sizeof(struct block_order) -
(sizeof(struct block_order) % 4);
```

```
        order = ((struct block_order *)block_addr)->order;
```

A partir disso, a função que implementa o algoritmo do *buddy allocator* quanto à liberação é **free_pages_ok**, que recebe o endereço do bloco e sua ordem. Com tais informações, a função libera o bloco, invertendo seu bit no respectivo mapa de bits, verifica se o buddy desse bloco está livre (com a definição auxiliar BUDDY_IS_FREE, que será posteriormente explicada) e, se estiver, chama a função que junta os *buddies* e retorna o endereço do novo bloco; caso contrário, ou seja, se o *buddy* do bloco liberado não estiver livre, ele é colocado na lista de livres de sua ordem.

Se houver a junção do bloco com o seu *buddy*, o processo se repete utilizando o novo bloco de ordem maior até que o *buddy* do bloco resultante não esteja livre e, portanto, ele seja posicionado na lista de livres respectiva. Outra forma do algoritmo ser terminado é o novo bloco criado após a junção ser de ordem MAX_ORDER-1 que, como já explicado, é a ordem inicialmente preenchida, que não possui *buddies*; portanto, nesse caso, o bloco é posicionado na lista dessa ordem e o processo se encerra.

Tais comportamentos podem ser acompanhados no código a seguir:

```
    for (current_order = order;;current_order++) {
            if (current_order < MAX_ORDER - 1) {
                MARK_USED((addr-heap_start)/PAGE_SIZE, current_order,
&free_area[current_order]); // when freed, the bit for the block and
its buddy is inverted
            }
            if (BUDDY_IS_FREE((addr-heap_start)/PAGE_SIZE,
current_order, &free_area[current_order]) && current_order < MAX_ORDER
- 1) { // for the MAX_ORDER-1, it is always marked as free
                addr = merge(addr, current_order); // merges the
block with its buddy, as it is also free
            }
            else {
                put_element(&free_area[current_order].free_list,
(struct list_addr *)addr); // puts the block in free_list
                break;
            }
        }
```

Para fazer a junção de um bloco com seu *buddy*, a função **merge** é utilizada. Essa função, primeiramente, calcula o endereço do *buddy* do bloco e, então, remove esse *buddy* da lista de blocos livre da ordem desses blocos. Então, é retornado o endereço do novo bloco, que corresponde ao menor dos dois endereços entre o bloco e seu *buddy*. Isso pode ser visto a seguir:

```
     buddy_addr = ((block_addr-heap_start) ^ (1 <<
(order+PAGE_SHIFT))) + heap_start;
          remove_from_list(&free_area[order].free_list, (struct
list_addr *)buddy_addr);

          return buddy_addr < block_addr ? buddy_addr : block_addr;
```

Para calcular o endereço do *buddy*, como ambos estão no mesmo alinhamento de página relativamente ao endereço inicial do heap, então é necessário somente inverter o bit (ordem + $\log_2$(PAGE_SIZE)), já que blocos de maior ordem terão alinhamento relativo de 2^{ordem} páginas.

2.4 Funções auxiliares

Além das funções auxiliares para lidar com uma lista duplamente ligada, destacam-se duas definições principais: MARK_USED e BUDDY_IS_FREE.

2.4.1 MARK_USED

MARK_USED é definida em malloc.h, como segue:

```
     #define MARK_USED(index, order, area)  change_bit((index) >>
(1+(order)), (area)->map)
```

Ou seja, ela utiliza a função change_bit e passa como parâmetro o índice da primeira página do bloco na memória dividido por $2^{ordem+1}$, além do mapa de bits da estrutura de área livre recebida. No caso, o índice é dividido pela potência de 2 de ordem mais 1 porque, justamente, ao dividir por 2^{ordem}, é revelado o índice dos blocos naquela ordem na memória, e o 1 somado a ordem se dá porque o mapa de bits mapeia pares de blocos, como explicado na seção 1.2.3.

A função **change_bit**, então, inverte o bit de um mapa de bits de acordo com o índice do bit recebido e o endereço do mapa recebido:

```
	static inline void change_bit(int nr, volatile unsigned long
*addr)
	{
		unsigned long *p = ((unsigned long *)addr) + (nr) /
BITS_PER_LONG;

		bit_inv((*p), (nr % BITS_PER_LONG));
	}
```

Portanto, nas 3 ocasiões em que MARK_USED é usada no código, ocorre a mesma chamada:

```
	MARK_USED((block_addr-heap_start)/PAGE_SIZE, order,
&free_area[order]);
```

O primeiro parâmetro é o índice da primeira página de um certo bloco, utilizando o endereço relativo ao heap desse bloco para calcular o índice no mapa de memória. Também são passadas a ordem e a struct dessa ordem, para que seu mapa de bits possa ser acessado.

2.4.2 BUDDY_IS_FREE

BUDDY_IS_FREE é definido como segue:

```
	#define BUDDY_IS_FREE(index, order, area)   (!(check_bit((index)
>> (1+(order)), (area)->map)))
```

Então, ela utiliza a função **check_bit** e recebe os mesmos parâmetros que MARK_USED, utilizando-os também da mesma forma, a saber, passando o mapa de bits da ordem recebida e o índice do bit a ser checado. Também, **check_bit** é definida como segue:

```
	static inline unsigned long check_bit(int nr, volatile unsigned
long *addr)
	{
		unsigned long *p = ((unsigned long *)addr) + (nr) /
BITS_PER_LONG;

```

```
        return bit_is_set((*p), (nr % BITS_PER_LONG));
    }
```

De forma semelhante a **change_bit**, essa função acessa a palavra desejada do mapa de bits do endereço em questão, mas, ao invés de inverter o bit desejado, ela verifica se o bit em questão é 1.

Como, no caso do algoritmo *buddy allocator*, o *buddy* está livre se, após liberar o bloco e inverter o seu bit de *buddy*, esse bit tem valor 0, como já explicado na seção 1.2.3, então BUDDY_IS_FREE inverte o resultado retornado por essa função. Ou seja, se o bit é zero, o *buddy* está livre e BUDDY_IS_FREE é verdadeiro; caso contrário, o valor é falso (0).

Também, porque **check_bit** acessa o mapa de bits da mesma forma que **change_bit**, então são passados os mesmos parâmetros tanto para MARK_USED quanto para BUDDY_IS_FREE:

```
    BUDDY_IS_FREE((addr-heap_start)/PAGE_SIZE, current_order,
&free_area[current_order])
```

3 RESULTADOS FINAIS

3.1 Alguns testes

Alguns testes realizados estão nesse link. A função utilizada para testar o alocador foi a seguinte:

```
    int main(void) {
        uint32_t pages = 3;
        void *p, *q, *r, *s, *t, *a, *b, *c;

        a = malloc(2048*sizeof(int)); // 8192 bytes -> 2 pages ->
order 1
        b = malloc(4096*sizeof(int)); // 16384 bytes -> 4 pages ->
order 2
        c = malloc(4097*sizeof(int)); // 16388 bytes -> at least 5
pages -> order 3 (8 pages)
        p = malloc(pages); // 3 bytes -> 1 page -> order 0
        q = malloc(pages*sizeof(int)); // 12 bytes -> 1 page -> order
0
        free(p);
```

```
        r = malloc(pages*sizeof(int)); // 12 bytes -> 1 page -> order
0
        free(c);
        s = malloc(1030*sizeof(int)); // 4120 bytes -> 2 pages ->
order 1
        free(a);
        free(q);
        free(r);
        free(b);
        t = malloc(1024*sizeof(int)); // 4096 -> 1 page -> order 0
        free(s);
        free(t);
    }
```

3.2 Observações

Apesar do algoritmo estar funcionando corretamente, o algoritmo em si possui algumas falhas que diminuem sua eficiência e podem levar a problemas de fragmentação externa e interna, além da possibilidade de blocos serem unidos e, na requisição seguinte, novamente divididos.

3.2.1 Problemas de fragmentação interna

Este algoritmo, apesar de intencionar alocar o menor espaço de memória necessário a uma requisição, não o faz da forma ótima; isso porque, o menor espaço de memória que pode ser alocado é um bloco de ordem 0, ou seja, uma página inteira. Dessa forma, por existirem inúmeros programas que requisitam alocações muito menores que uma única página, o índice de fragmentação interna para um alocador que utilize somente o algoritmo de alocação *buddy* será significativamente alto.

Um contorno para esse problema, usado , por exemplo, no Linux, é o alocador *slab*, que subdivide páginas em blocos de memória menores que 1 página para atender requisições de poucos bytes. Com isso, é possível minimizar a fragmentação externa inerente ao *buddy allocator*.

3.2.2 Problemas de fragmentação externa

Tal problema advém do fato de que, se não existem blocos de uma certa ordem disponíveis porque todos estão alocados, então os blocos de ordem superior são divididos. Entretanto, pode ocorrer de, em vários pares de *buddies*, somente um deles ter sido liberado e estar disponível, enquanto o outro está alocado; com isso, ocorre a fragmentação externa, pois esses blocos, se não houvesse o sistema de buddies, poderiam ser, em alguns casos, combinados para formar um bloco de ordem maior e atender requisições dessa ordem.

Apesar disso, esse não é um problema tão grave e, no Linux por exemplo, é solucionado utilizando a alocação não-contígua de memória, que faz uso da virtualização da memória. Se utiliza então, da camada de abstração de endereços virtuais.

3.2.3 Problema da repetição de trabalho

Esse problema decorre do comportamento do algoritmo na liberação de blocos.

Por exemplo, inicialmente, todos os blocos são da ordem MAX_ORDER-1, como já explicado. Então, uma primeira alocação, de ordem 0, é requisitada. Com isso, o primeiro bloco de ordem MAX_ORDER-1 é dividido até que se obtenha um bloco de ordem 0. Após isso, esse bloco é liberado e, porque nenhum outro bloco foi alocado, todos os *buddies* até a penúltima ordem serão unidos, e o bloco de ordem MAX_ORDE-1 retornará à lista após todas as junções. Por último, ocorre uma outra requisição para um bloco de ordem 0 e, novamente, o bloco de ordem MAX_ORDER-1 será dividido até que um de ordem 0 esteja disponível.

Em suma, o comportamento da liberação de unir blocos até que não seja mais possível fazê-lo não é ótimo, já que no pior caso, descrito acima, o algoritmo terá várias junções seguidas imediatamente pela divisão dos mesmos blocos.

3.3 Memória Virtual

Para operar em um sistema operacional que não permite que programas de modo usuário acessem endereços físicos diretamente, mas sim endereços virtuais que, a cada acesso à memória, são traduzidos para endereços físicos, então, é necessário que, no caso de um alocador em nível de sistema operacional, esse retorne endereços virtuais ao programa requisitante.

Mais especificamente, para o projeto em questão, seria necessário, após a alocação do bloco e da determinação de seu endereço, traduzir o endereço do bloco para um endereço virtual e, somente após isso, retorná-lo ao programa usuário. No caso da liberação, o endereço recebido, que seria virtual, seria inicialmente traduzida e, após isso, a liberação efetivamente ocorreria.

Ou seja, nesse caso, todas as estruturas de gerenciamento da memória continuariam a utilizar endereços físicos e somente os endereços retornados pelo malloc e recebidos pelo free seriam virtuais. Isso seria possível porque, programas em nível do sistema operacional, como o alocador em questão, podem acessar diretamente a memória física.

4 REFERÊNCIAS

https://www.kernel.org/doc/gorman/html/understand/understand009.html