Relatório — Sistemas Operacionais

Eduardo Cabral, Gabriel Perez, Romulo Menezes, and Victor Lopes

Departamento de Ciência da Computação, Universidade Federal Rural do Rio de Janeiro, Rio de Janeiro, Brasil.

Data: 21 de dezembro de 2021

Professora: Juliana Mendes Nascente e Silva Zamith

1 Introdução

A alocação dinâmica é de grande utilidade em diversos projetos, pois ela possibilita a alocação de memória em tempo de execução, não sendo necessário definir uma quantidade de memória a ser alocada na hora de escrever o código. Nas linguagens de programação C e C++, existe a função malloc que é responsável por fazer as alocações de memória de maneira dinâmica, sendo preciso passar um argumento com o tamanho de bytes para ser alocado na memória RAM. No entanto, a implementação do **glibc** do malloc não é muito eficiente, e não é atoa que algumas empresas, como a Google, tem a sua própria implementação do malloc.

Objetivos

Durante o desenvolvimento, tínhamos o objetivo de funções de **alta performance, mas com a menor quantidade de fragmentação possível**. Após pesquisas, descobrimos que existem diversas formas de tornar o acesso à memória mais rápida, como alocar os blocos de tamanhos de forma alinhada, mas isso resultaria em fragmentação interna, pois nem todo o bloco seria ocupado.

Trabalhos Relacionados

Para a realização desse trabalho foi utilizado como base o artigo "A Malloc Tutorial"[1], de Marwan Burelle.

2 Desenvolvimento

Implementação

Para a implementação do nosso alocador de memória dinâmica, foram criadas duas funções principais que são elas:

- malloc: que serve para **alocar** a memória, recebendo um valor referente ao tamanho de bytes a serem alocados e retornando o ponteiro de onde foi reservado esse espaço na memória.
- free: que serve para **desalocar** a memória alocada, sendo preciso passar o ponteiro de onde a memória foi alocada.

A ideia aplicada para essa implementação foi utilizar **listas duplamente encadeadas**, onde elas têm informações úteis com o tamanho do bloco atual, o ponteiro para o próximo bloco e se um bloco esta disponível ou não para fazer uma alocação, por exemplo. Também foi utilizada a estratégia de alocação **First-Fit**, que consiste em alocar um determinado número de bytes no primeiro bloco disponível que ele encontrar na lista que caiba esse número de bytes.

A partir dessa ideia, a função **malloc** verifica através de uma variável global se já existe algum bloco de memória criado previamente. Caso não tenha sido criado, a função malloc chama outra função auxiliar que cria um bloco grande suficiente para armazenar as informações do bloco mais o tamanho requisitado pelo usuário, e retorna um ponteiro do endereço com o "tamanho" que foi requisitado. Caso já existam blocos, é chamada uma outra função auxiliar para verificar se existe algum bloco com o tamanho menor ou igual ao requisitado. Caso exista um bloco livre com tamanho suficiente, é feito uma divisão do bloco que pode resultar em dois blocos, um bloco do tamanho requisitado e outro com o que sobrou. Caso contrário, a heap é expandida e um novo bloco é gerado.

Para realizar as alocações e desalocações de memória, foi utilizado as funções **brk()** e **sbrk()**. A função brk() é responsável por definir o final do segmento de memória

do processo e a sbrk() incrementa o tamanho. Chamar a sbrk() com parâmetro 0, retorna o final do segmento de memória. Esse ato de aumentar e reduzir o segmento de memória do processo tem o efeito de alocar e desalocar memória para o processo.

Já a função **free** funciona basicamente desalocando o bloco de memória. Quando a função recebe o ponteiro como parâmetro, é feito algumas validações para garantir que o ponteiro realmente faz parte de um bloco criado pela biblioteca. Se isso for válido, o bloco é definido como livre e uma tentativa de fusão com os blocos adjacentes é feita. Se essa união resultar em um bloco que se encontra no final da lista de blocos, a heap é reduzida pelo tamanho desse bloco. Essas operações permitem uma otimização e redução da fragmentação da memória.

Apesar de ter sido solicitado a criação de uma função de inicialização (init) que seria responsável por solicitar um espaço de memória ao sistema operacional e que esse espaço fosse utilizado pela implementação, decidimos que essa função seria desnecessária, já que as outras funções seriam responsáveis por isso.

Padding e Alignment

Além disso, como o compilador faz o uso de padding para alinhar os elementos das structs, fizemos o uso da diretiva `__attribute__((packed))` na struct do bloco para permitir que o código funcione corretamente:

```
struct s_block {
    [...]
} __attribute__((packed));
```

Também optamos por não utilizar o alinhamento de memória justamente para evitar a fragmentação que poderia ocorrer entre os blocos. Visto que o endereço do final de um dos blocos poderia não estar alinhado, sendo necessário avançar alguns bytes para ficar de acordo.

Testes

Para garantir que o código se comportava corretamente durante o desenvolvimento, os seguintes testes foram elaborados:

⇒ Teste 1: verificar se os valores armazenados nos ponteiros alocados não foram alterados durante o processo;

⇒ Teste 2: verificar se os valores armazenados em uma matriz n x m foram alterados durante o processo;

⇒ Teste 3: verificar se os valores armazenados em uma matriz n x m x p foram alterados durante o processo;

⇒ Teste 4: verificar se os valores armazenados em uma matriz n x m x p vetorizada foram alterados durante o processo;

⇒ Teste 5: verificar se os ponteiros da heap retornados pela função sbrk(0) após as alocações e liberações estão coesos. Isso é, se o tamanho da heap é o mesmo após alocar e desalocar os blocos.

Benchmarks

Para saber como o código performava, alguns benchmarks foram elaborados.

⇒ Benchmark 1: Compara malloc e malloqueiro com x bytes e informa o tempo execução de ambos;

⇒ Benchmark 2: Avalia se ao alocar sem desalocar muitos blocos ocorrerá algum erro;

⇒ Benchmark 3: Tempo de alocação do malloqueiro de 1024 até 1024000 bytes a cada 1024 bytes de forma sequencial;

⇒ Benchmark 4: Tempo de alocação do malloqueiro de 1024 até 1024000 bytes a cada 1024 bytes de forma paralela;

⇒ Benchmark 5: Tempo de alocação do malloc do glibc de 1024 até 1024000 bytes a cada 1024 bytes de forma paralela.

⇒ Benchmark 6: Tempo de alocação do malloc do glibc de 1024 até 1024000 bytes a cada 1024 bytes de forma sequencial.

Testando em paralelo

Também foram feitos alguns testes a fim de tentar entender um pouco sobre como seria a paralelização dos testes (não da implementação em si), e para fazer esses testes paralelizamos os testes do nosso malloc e do malloc padrão e como resultado obtivemos que o nosso malloc ficou com mesmo desempenho porém o glibc ficou cerca de 13% mais lento. Porém, mesmo após fazer pesquisas e pensar sobre, não conseguimos entender o motivo disso acontecer.

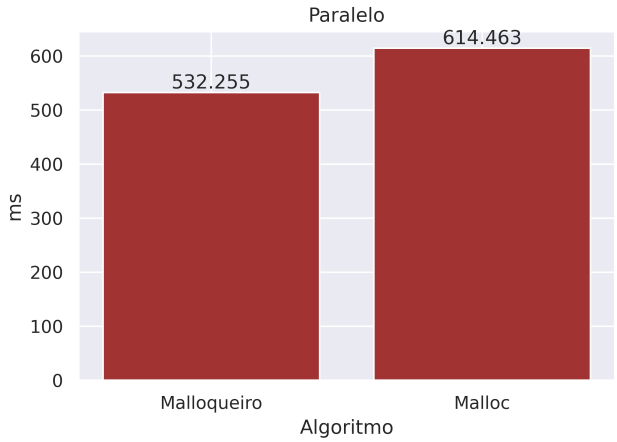

Figura 1: Comparação de tempo de alocação e desalocação das implementações em paralelo

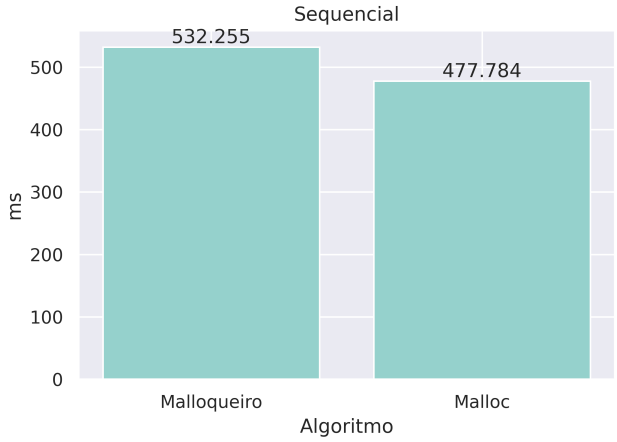

Figura 2: Comparação de tempo de alocação e desalocação das implementações sequencialmente

3 Resultados

Para todos os testes apresentados anteriormente, a nossa implementação conseguiu passar em todos eles, sem apresentar problema algum. No entanto, ao fazer uma comparação com o malloc do glibc, foi possível observar algumas coisas:

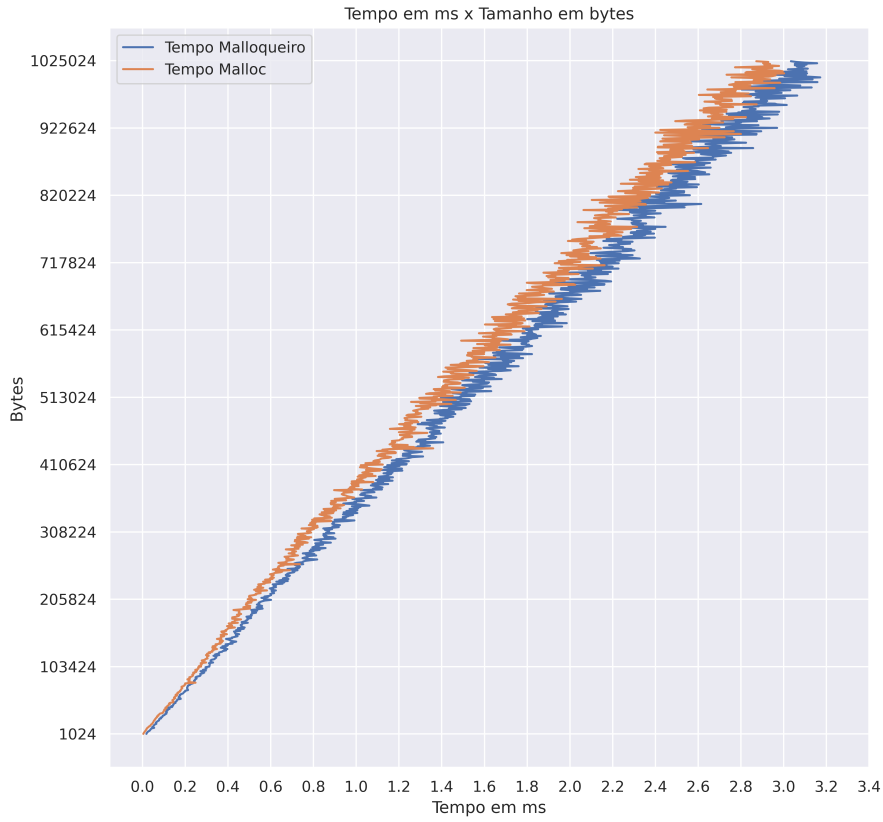

Figura 3: Comparação da média do tempo de alocação e desalocação das implementações

O gráfico acima foi elaborado a partir da média dos tempos de 50 benchmarks. Os valores de alocação iniciaram em 2 bytes e foram incrementados de 1024 bytes enquanto o valor fosse menor ou igual a 1.024.000 bytes (1,024 MB).

Podemos observar que ambas implementações resultados bem semelhantes, até mesmo em grandes alocações.

4 Conclusão

Como é possível observar pelos resultados, embora a nossa implementação tenha funcionado para os testes feitos, ela não apresentou resultados melhores que o malloc do glibc.

Uma solução para tentar melhorar esse desempenho seria implementar uma segunda lista com os blocos de memória livres para serem alocados e a sua referência para a lista completa. Fazendo isso, o tempo de busca por blocos livres seria reduzido, porém, continuaria ocorrendo problemas de fragmentação após as divisões dos blocos.

Outra solução viável poderia ser utilizar tanto o sbrk quanto o mmap. Apesar o sbrk servir bem para a maioria dos casos, o mmap tem um desempenho maior em grandes alocações. O malloc do glibc tem um comportamento semelhante, se o limite da constante **MMAP_THRESHOLD** for atingido, o mmap é utilizado em vez do sbrk.

Pensamos em utilizar outro tipo de estrutura de dados, mas a lista é a única que conseguimos pensar ser possível ocorrer a fusão de blocos adjacentes para reduzir a fragmentação. Logo, árvores e heaps estariam descartadas e só sobrariam as listas (encadeadas ou duplamente encadeadas). Por eliminação sobraria a lista duplamente encadeada.

Todos os dados, algoritmos e gráficos informados no relatório podem ser consultados no repositório Malloqueiro[2] do GitHub.

Lista de Figuras

Referências

[1] Marwan Burelle. «A Malloc Tutorial». Em: (2009).

[2] Romulo Morais Menezes Eduardo de Almeida Ferro Cabral Gabriel Perez Vargas De Vasconcelos e Victor Lopes Machado. *Malloqueiro*. URL: `https://github.com/Softawii/Malloqueiro`.