Alocação de Memória e Sistemas de Arquivos

Prof. Carlos Roberto Moratelli

1 Informações Gerais

- Trabalho Prático 2 de Sistemas Computacionais.
- Semestre 2023/2.
- Entrega: 07/12.
- Equipes de até 3 integrantes.
- O trabalho é dividido em duas partes. Uma envolve alocação de memória e a outra sistema de arquivos.

2 Parte 1 - Alocador de Memória.

Neste exercício simularemos o funcionamento das funções malloc() e free() da linguagem C, através de uma implementação simplificada do algoritmo de alocação de memória. Primeiro, vamos entender com funciona o alocador de memória da GLIBC.

2.1 GLIBC

A GLIBC (GNU C Library), sendo a biblioteca padrão C para muitos sistemas Linux, usa um alocador de memória chamado ptmalloc (por padrão, desde a versão GLIBC 2.3), o qual é uma variante do alocador dlmalloc criado por Doug Lea.

O ptmalloc utiliza uma abordagem de alocação chamada binning. Ele mantém uma lista de bins (basicamente, listas encadeadas) para blocos de diferentes tamanhos. Quando uma alocação é solicitada, ele procura o bin de tamanho apropriado. Se não encontrar um de tamanho adequado, solicitará mais memória ao sistema.

O dlmalloc (e, por extensão, o ptmalloc) utiliza uma combinação de técnicas para a alocação de memória, incluindo:

- **Splitting**: quando um bloco de memória é muito grande para uma solicitação, ele é dividido em dois: um bloco alocado e um bloco livre.
- **Coalescing**: quando blocos de memória adjacentes são liberados, eles são combinados em um único bloco maior para reduzir a fragmentação.
- **Boundary tags**: São utilizadas para armazenar metadados (como o tamanho do bloco) diretamente antes e depois do bloco de memória, facilitando a implementação da coalescência.

- **Bins**: Os blocos livres são organizados em bins com base em seu tamanho. Cada bin pode conter blocos de um determinado intervalo de tamanhos, facilitando a busca por um bloco de tamanho adequado quando uma alocação é solicitada.

É importante notar que os algoritmos de alocação de memória podem variar consideravelmente dependendo do sistema operacional e da biblioteca C específica. O alocador de memória do GLIBC é otimizado para lidar eficientemente com uma ampla variedade de cenários de alocação de memória que podem ocorrer em aplicações reais.

O que foi descrito até aqui, acontece na memória de um processo. Contudo, a GLIBC necessita solicitar memória ao sistema operacional usando o chamadas de sistema. No caso de sistemas Unix-like (como Linux), estas são as chamadas sbrk e mmap:

- **sbrk**: Esta função aumenta o tamanho do segmento de dados do processo pelo valor especificado. Isso é usado para alocar memória no heap do processo. No entanto, o uso de sbrk tem diminuído em favor de mmap por várias razões, incluindo a possibilidade de fragmentação e a falta de compatibilidade entre threads.

- **mmap**: Esta função é usada para mapear um arquivo ou um dispositivo na memória, mas também pode ser usada para alocar memória anônima (ou seja, que não está associada a nenhum arquivo) quando passada a flag MAP_ANONYMOUS. mmap tem várias vantagens sobre sbrk, como a capacidade de alocar e liberar memória em qualquer lugar na memória virtual do processo, sem se preocupar com a fragmentação.

Em geral, a GLIBC usa sbrk para alocar blocos menores de memória e mmap para alocar blocos maiores. A razão para isto é que mmap tem um overhead maior, então não é tão eficiente para blocos pequenos, mas é mais flexível e evita a fragmentação para blocos grandes.

Quando a memória é alocada com mmap, ela volta para o sistema imediatamente quando é liberada. No entanto, a memória alocada com sbrk não volta para o sistema imediatamente quando é liberada — em vez disso, ela é mantida para uso futuro pelo alocador de memória da GLIBC. Isso pode melhorar o desempenho evitando chamadas desnecessárias do sistema, mas também pode resultar em um uso maior de memória.

Por fim, vale lembrar que estes detalhes são bastante específicos para GLIBC e Linux. Outras bibliotecas C ou sistemas operacionais podem usar diferentes mecanismos para gerenciar a memória.

2.2 smalloc - Simplified Malloc

Implementaremos as funções smalloc() e sfree() que serão versões simplificadas de seus originais. A simulação começa com um bloco de memória de tamanho fixo, alocado com o uso de sbrk() ou malloc(), representado por um vetor bytes. Criaremos, também, uma estrutura mem_block que representa os blocos de memória que estão sendo gerenciados. Cada bloco mantém as seguintes informações:

```c
#pragma pack(push, 1)
struct mem_block {
    int is_free;    // Se o bloco esta livre 1 senao 0.
    size_t size;    // Tamanho do bloco de memoria.
    void* mem_ptr; // Ponteiro para o bloco de memoria.
    struct mem_block* next; // Ponteiro para o proximo bloco na lista.
};
#pragma pack(pop)
```

No início, temos apenas um bloco que representa toda a memória, que está livre.

Quando uma alocação de memória é solicitada (usando a função smalloc()), percorremos a lista de blocos até encontrar o primeiro bloco livre que é grande o suficiente para acomodar a quantidade de memória solicitada (este é o método *first-fit*).

Se o bloco é maior que *sizeof(struct mem_block)* mais a quantidade solicitada, então o bloco é dividido em dois: um para a memória alocada e outro para a memória restante. Para isso, ajustamos o tamanho do bloco alocado para corresponder à quantidade solicitada e criamos um novo bloco para a memória restante. Veja o exemplo:

[Bloco livre (1024 bytes)]

Após alocar 200 bytes:

[Bloco alocado (200 bytes)][Bloco livre (824 bytes)]

Quando um bloco de memória é liberado (usando a função sfree()), simplesmente marcamos o bloco correspondente como livre. Contudo, se houver dois blocos livres adjacentes, devemos executar a coalescência, ou seja, combinar esses três blocos em um único bloco maior. Por exemplo, o diagrama a seguir ilustra esse processo.

[Bloco livre (300 bytes)][Bloco alocado (200 bytes)][Bloco livre (524 bytes)]

Depois da coalescência:

[Bloco livre (1024 bytes)]

2.3 Simulação

O programa principal deve simular uma série de chamadas smalloc()/sfree(). O professor irá disponibilizar o programa de testes com as rotinas necessárias para validar a implementação. Caso não haja memória suficiente, smalloc() deve mostrar uma mensagem adequada e retornar NULL. Quando a memória é alocada com sucesso, smalloc() retorna um ponteiro para a memória alocada.

3 Parte 2 - Arquivador

O objetivo deste exercício é implementar um sistema simples de arquivamento de arquivos, que permite a concatenação de vários arquivos em um único arquivo em disco. O sistema de arquivamento deve permitir que os arquivos individuais sejam posteriormente recuperados.

O programa deverá implementar as seguintes funcionalidades:

- Arquivamento de arquivos: esta operação deve receber uma lista de caminhos de arquivos e um caminho de destino para o arquivo de saída. Cada arquivo deve ser lido do disco, e seu conteúdo deve ser anexado ao arquivo de saída. Antes do conteúdo de cada arquivo, um cabeçalho deve ser escrito contendo o nome do arquivo e seu tamanho.

- Recuperação de arquivo: esta operação deve receber um caminho para o arquivo de arquivamento e um nome de arquivo. O programa deve procurar o arquivo com esse nome e, se encontrado, escrever seu conteúdo para um novo arquivo no disco. Se o arquivo não for encontrado, deve emitir uma mensagem de erro adequada.

- Listagem de todos os arquivos e tamanhos incluídos no arquivamento.

A estrutura do arquivo de arquivamento deve ser lida como uma lista encadeada de arquivos, onde cada nó na lista é um cabeçalho de arquivo seguido pelo conteúdo do arquivo. Para encontrar um arquivo, você deve ler cada cabeçalho de arquivo até encontrar o arquivo desejado. O cabeçalho pode ter o seguinte formato:

```
1 #define MAX_NAME 256
2
3 #pragma pack(push, 1)
4 struct file_header{
5     char name[MAX_NAME];              // Nome do arquivo
6     unsigned int file_size;           // Tamanho do arquivo em bytes
7 };
8 #pragma pack(pop)
```

Ou seja, o cabeçalho do primeiro arquivo é encontrado na posição 0 do arquivo. O próximo cabeçalho estará na posição sizeof(struct file_header) + file_size;

Para que todas as implementações sigam os mesmos padrões de entrada e saída, use o esqueleto de código fornecido pelo professor.

4 Entregáveis

1. Código-fonte dos dois exercícios.

2. A equipe deve mostrar em funcionamento e explicar para o professor a implementação.