PMR3309 - 2018
Malloc Lab: Construíndo um gerenciador de memória dinâmica
Data final de entrega: 12/12/2018 às 23:59

Adaptado do Material de Bryant e O'Hallaron para o curso 15-213 2016 da CMU por Thiago Martins

1 Introdução

Neste trabalho você escreverá um gerenciador de alocação dinâmica para programas em C, ou seja, a sua própria versão das funções `malloc`, `free` e `realloc`. Você é encorajado a explorar diversas alternativas para implementar um gerenciador que seja rápido e eficiente.

Atenção: Os programas a serem desenvolvidos nesta atividade serão compilados para arquiteturas de *32 bits*!

2 Obtendo seu trabalho

Obtenha o arquivo com os códigos para o trabalho pelo moodle:

`https://edisciplinas.usp.br/mod/resource/view.php?id=2392192`

Copie o arquivo `malloclab-handout.tar` em um diretório protegido no qual você pertende fazer o seu trabalho. Em seguida use o comando: `tar xvf malloclab-handout.tar`. Isso gerará vários arquivos neste diretório. O *único* arquivo que você modificará é `mm.c`. O programa `mdriver.c` é um programa de teste que será empregado para avaliar o desempenho da sua solução. Use o comando `make` para gerar o código deste e executá-lo com o comando `./mdriver -V`. (A opção `-V` mostra informações úteis.)

QUando você completar suas atividades, entregará apenas um arquivo (`mm.c`), com a sua solução.

3 Como trabalhar nesta atividade

O gerenciador de memória dinâmica que você construirá consiste das 4 seguintes funções, que estão declaradas em `mm.h` e definida `mm.c`.

```
int    mm_init(void);
```

```
void *mm_malloc(size_t size);
void  mm_free(void *ptr);
void *mm_realloc(void *ptr, size_t size);
```

O arquivo `mm.c` que foi fornecido implementa um pacote de gerenciamento de memória dinâmica trivial, mas correto. Este gerenciador, como você perceberá, é extremamente veloz (muito mais do que os da biblioteca c) e extremamente ineficiente (tão ineficiente que é incapaz de tratar alguns exemplos). Você pode usar este gerenciador como um base para o seu código, modificando as funções lá e possivelmente criando outras funções privadas `static`.

Lembre-se que as funções que você deve implementar devem obedecer a seguinte semântica:

- `mm_init`: Antes de chamar qualquer outra função do seu gerenciador, o programa (por exemplo o programa avaliador) deve charmar `mm_init` para realizar todas as inicializações necessárias, como alocar a área inicial do *heap*. O valor de retorno deve ser -1 se houver algum problema com a inicialização e 0 caso contrário.

- `mm_malloc`: A função `mm_malloc` retorna um ponteiro para um bloco que comporte ao menos `size` bytes. O bloco inteiro deve estar contido na região do *heap* e não deve se sobrepor a nenhum outro bloco atualmente alocado.

  O comportamento da sua função deve ser similar ao da função `malloc` da biblioteca C. Como em 32 bits a função `malloc` sempre retorna endereços alinhados em 8 bytes, a sua função também deve fazê-lo.

- `mm_free`: A função `mm_free` libera o bloco apontado por `ptr`. Ela não retorna nada. Essa rotina pode pressupor que o ponteiro (`ptr`) foi obtido em uma chamada anterior a `mm_malloc` ou `mm_realloc` e ainda não foi desalocado. Sua função pode ter comportamento indeterminado caso este pressuposto seja violado.

- `mm_realloc`: A função `mm_realloc` retorna um ponteiro para uma região que comporta ao menos `size` bytes com as seguintes restrições:

  - Se `ptr` for NULL, a chamada é equivalente a `mm_malloc(size);`
  - Se `size` for igual a zero, a chamada é equivalente a `mm_free(ptr);`
  - Se `ptr` for não-nulo, você pode pressupor que ele foi retornado anteriormente por uma chamada a `mm_malloc` ou `mm_realloc` e ainda não foi desalocado (sua função pode ter comportamento indeterminado caso contrário). A chamada a `mm_realloc` muda o tamanho do bloco de memória apontado por `ptr` (o bloco antigo) para `size` bytes e retorna o endereço do novo bloco. Note que o endereço do novo bloco pode ser o mesmo do bloco anterior ou pode ser diferente, dependendo da sua implementação, a quantidade de fragmentação interna do bloco antigo e o tamanho da solicitação feita a `realloc`.

    O conteúdo do novo bloco dve ser o mesmo que o do bloco antigo apontado por `ptr`, até o *mínimo* dos tamanhos antigo e novo (pois a chamada a realloc pode diminuir ou aumentar o bloco). Por exemplo, se o bloco antigo tiver 8 bytes e o novo 12, então os 8 primeiros bytes do novo bloco devem ser idênticos aos do bloco antigo. Do mesmo modo, se o bloco antigo for de

8 bytes e o novo bloco de 4 bytes, então o conteúdo dos 4 primeiros bytes do novo bloco devem ser idênticos ao do bloco antigo.

Estas convenções seguem as das funções correspondentes da biblioteca c `malloc`, `realloc` e `free`. Digite `man malloc` para a documentação completa destas funções.

4 Verificação de consistência do *heap*

Gerenciadores de memória dinâmica são notoriamente difíceis de se programar corretamente. Isso acontece por que eles usam diversas manipulações de ponteiros sem verificação em tempo de compilação de tipagem. Um verificador de consistência do seu *heap* pode ajudá-lo nesta tarefa.

Algumas funções úteis de um verificador de consistência são:

- Todos os blocos armazenados na sua estrutura de dados de blocos livres são realmente blocos livres?
- Há blocos livres contíguos que deveriam ter sido coalescidos?
- Todos os blocos livres estão presentes na sua estrutura de dados de blocos livres?
- Há sobreposição entre blocos alocados?

Você pode escrever um verificador de consistência adicionando uma função `int mm_check(void)` em `mm.c`. Ela verificará quaisquer invariantes e condições de consistência que você achar prudente. Invoque esta função ao final de cada chamada das suas funções `mm_malloc`, `mm_realloc` e `mm_free`. Sugere-se que você imprima mensagens de erro nesta função caso algum problema seja encontrado.

5 Rotinas de Suporte

O pacote memlib.c simula o sistema de memória para o seu gerenciador. Você pode usar as seguintes funções em `memlib.c`:

- `void *mem_sbrk(int incr)`: Expande o seu *heap* em `incr` bytes, onde `incr` é um inteiro positivo não-nulo. A função retorna um ponteiro para o primeiro byte da nova região alocada. Você pode assumir que esta nova área é *contígua* à area anterior. A semântica é simular à função Unix `sbrk`, exceto que `mem_sbrk` aceita somente um número positivo não-nulo.
- `void *mem_heap_lo(void)`: Retorna um ponteiro para o primeiro byte no *heap*.
- `void *mem_heap_hi(void)`: Retorna um ponteiro para o último byte válido do *heap*.
- `size_t mem_heapsize(void)`: Retorna o tamanho atual do *heap* em bytes.
- `size_t mem_pagesize(void)`: Retorna o tamanho da página do sistema (4K em sistemas Linux x86).

6 O programa de teste

O programa de teste `mdriver.c` testa o seu pacote `mm.c` verificando correção, utilização de espaço e velocidade. O programa é controlado por um conjunto de arquivos de rastreamento que estão no subdiretório `traces`. Cada arquivo contém uma sequência de alocações realocações e deasalocações que instruem o programa a chamar suas funções `mm_malloc`, `mm_realloc`, e `mm_free` em ordem apropriada. O programa de teste e os arquivos de rastreamento são os mesmos que serão empregados para avaliar o seu trabalho.

O programa `mdriver.c` aceita os seguintes parâmetros de linha de comando:

- `-t <tracedir>`: Porcura por arquivos de rasteamento no diretório `tracedir` no lugar do diretório padrão definido em `config.h` (o subdiretório `traces`).
- `-f <tracefile>`: Usa um arquivo particular `tracefile` para testes no lugar do conjunto padrão.
- `-h`: Mostra um sumário dos parâmetros de linha de comando.
- `-l`: Além de avaliar o pacote do aluno, avalia também a velocidade do pacote malloc da bilbioteca c (mas não avalia a sua utilização de espaço).
- `-v`: Mostra o desempenho individualizado para cada arquivo de rastramento em uma tablea compacta.
- `-V`: Saída mais detalhada. Mostra informação adicional de diagnóstico a medida que cada arquivo de rastreamento é processado. Útil para determinar qual arquivo (e em que ponto) está fazendo seu pacote falhar..

7 Regras de Programação

- Você não deve alterar as interfaces emn `mm.c`.
- Você não deve invocar nenhuma rotina de gerenciamento de memória, nem tampouco *syscalls*. Isso naturalmente exclui o uso de `malloc`, `calloc`, `free`, `realloc`, `sbrk`, `brk` e quaisquer variantes no seu código. Outras funções da biblioteca c são permitidas (`memcpy` é particularmente útil na implementação de `mm_realloc`).
- Você não pode definir nenhum dado composto global nem `static`, como arrays, structs, árvores e listas no seu `mm.c`. Você pode no entanto declarar variáveis escalares globais tais como inteiros, floats e ponteiros.
- Para consistência com o pacote `malloc` em 32 bits, você deve sempre retornar ponteiros alinhados em 8 bytes. O programa de testes verificará este alinhamento.

8 Avaliação

Você receberá nota *zero* se você violar alguma regra, se seu código tiver bugs que levarem a uma falha no programa teste ou se o seu gerenciador não for capaz de tratar todos os arquivos de rastreamento (note que o alocador fornecido *não* é capaz de tratar todos os casos).

Caso contrário sua nota será calculada da seguinte maneira:

- Desempenho (100 points). Duas métricas de desempenho serão empregadas na sua avaliação:
  - *Utilização de espaço*: O *pico* da razão entre a memória total usada pelo gerenciador (ou seja, atualmente alocada) e o tamanho do *heap* empregado pelo seu alocador. A utilização ótima é, naturalmente, 1, mas uma utilização de 95% já garante nota máxima neste quesito (para fins de cálculo da nota, sua utilização é dividida por 0,95 e limitada superiormente em 1).
  - *Velocidade*: O número médio de operações completadas por segundo na máquina de testes do *labgeocomp*. A sua velocidade será comparada com uma "velocidade padrão" de 15000 Kops (mil operações por segundo). A velocidade do seu programa - quando executado na máquina de testes do laboratório - será dividida pela velocidade padrão para obter o índice de velocidade. Caso você supere a velocidade padrão (o que é na verdade bem fácil, note como o gerenciador fornecido já faz isso), seu índice de velocidade será 1.

    Atenção! Naturalmente os testes que você executar em qualquer outro computador que não a máquina de testes padrão da disciplina irá apresentar índices de velocidade distintos. Para referência, a máquina de testes executa os testes da biblioteca padrão malloc a 18700 Kops. Você pode usar este índice para estimar uma constante de proporcionalidade entre o desempenho da sua máquina e a de testes.

  O programa de testes calcula o seu desempenho somando o índice de desempenho multiplicado por 60 e o índice de velocidade por 40. Note que esta distribuição favorece ligeiramente a utilização. De fato, você descobrirá que é relativamente fácil escrever um gerenciador rápido. No entanto, você não deve sacrificar excessivamente o desempenho pela utilização.

  É possível atingir um índice de 100% de desempenho nesta atividade, mas você deverá empregar estruturas de dados complexas (possivelmente mais do que as vistas em aula).

- Estilo (20 pontos).
  - Seu código deve ser dividido em funções e empregar o mínimo de variáveis globais possível.
  - Seu código deve iniciar-se por um cabeçalho de comentário que descreve a estrutura dos seus blocos livres e alocados, a organização da estrutura de dados de blocos livres e alocados (se ouver) e como você a manipula. Todas as funções devem ser precedidas de um cabeçalho que descreve o seu comportamento..

9 Instruções para entrega

Submeta o seu arquivo `mm.c` pelo moodle:

https://edisciplinas.usp.br/mod/assign/view.php?id=2392193

10 Sugestões

- *Use a opção* `mdriver -f`. Durante o desenvolvimento inicial, use rastreamentos pequenos para simplificar a depuração e testes. Dois rastreamentos pequenos foram incluídos, (`short1,2-bal.rep`) que você pode usar para testes iniciais.
- *Use as opções* `mdriver -v` *e* `-V`. A opção `-v` dará um sumário detalhado de cada arquivo. A opção `-V` irá além disso indicar quando cada arquivo é lido, o que o ajudará a isolar erros.
- *Compile com a opção* `gcc -g -m32` *e use um debugger.* Um debugger irá ajudá-lo a isolar e identificar problemas de referência a memória (lembre-se que a opção `-m32` é necessária para compilar seu código em 32 bits!).
- *Compreenda a implementação do alocador no livro texto.* O livro texto tem um exemplo detalhado de um alocador simples baseado em lista implícita. Use este alocador como ponto de partida.
- *Encapsule a sua aritmética de ponteiros em macros.* Aritmética de ponteiros é confusa e sujeita a erros por causa de todas as conversões de tipos necessárias. Você pode reduzir a complexidade escrevendo macros para suas operações de ponteiros. Veja livro-texto para exemplos.
- *Faça sua implementação em etapas.* Os 9 primeiros testes contém chamadas somente a `malloc` e `free`. Os 2 últimos contém também chamadas a `realloc`. Sugere-se que você comece por fazer implementações corretas e eficientes de `malloc` e `free` nos primeiros 9 testes. Apenas aí você deve cuidar da implementação de `realloc`. Você pode inicialmente construir uma implementação de `realloc` usando as funções `malloc` e `free` implementadas. Mas um desempenho realmente bom só será obtido com uma implementação independente de `realloc`.

11 Lembretes de aritmética de ponteiros em 32 bits

Lembre-se, o código gerado será de 32 bits. Isso tem algumas implicações:

- O tamanho de `int`s e `long`s é de 4 bytes.
- O tamanho de todos os ponteiros é de 4 bytes.

A aritmética de ponteiros em C é tal que um ponteiro é sempre modificado em múltiplos do tipo de dado a ele associado. Assim, em 32 bits, tem-se

```
int a;
int *b;
b = &a + 2;
```

Neste código, o ponteiro `b` contém o endereço de `a` mais *oito* bytes (duas vezes o tamanho de um inteiro).

Uma maneira de se incrementar um endereço de um ponteiro em bytes é convertê-lo temporariamente em um ponteiro para `char`. Assim, este código:

```
void *mudaponteiro(void *p, int offset) {
    return (char *)p + offset;
}
```

Retorna o endereço apontado por `p` modificado de `offset` bytes.