Arquitetura de Computadores

LEI 2022/2023

TPC 3

Entrega: 23h59 de 15 de maio de 2023

Este trabalho de casa é para ser realizado em grupos de 2 estudantes. A entrega será feita por metodologia a ser definida por mensagem de e-mail enviada via CLIP
Os casos de plágio serão punidos de acordo com os regulamentos em vigor.

Malloc e chamada ao SO *brk*

Durante a execução de um programa, a criação/libertação de espaço para dados é, em C, normalmente baseada nas funções de biblioteca, *malloc* e *free*. O espaço de memória usado para tal é chamado *heap*, sendo este um espaço contínuo de memória (isto é, contínuo em termos de endereços) com dois limites:

- um endereço de início;
- um endereço limite corrente (*break*);

O limite corrente marca o fim do espaço de memória de dados atribuído pelo SO e acessível ao processo num dado momento. O valor deste limite pode ser alterado com as chamadas ao sistema *brk* e *sbrk*.

`int brk(void *addr)` – altera, se possível, o limite do segmento de dados para o valor especificado pelo *addr*; retorna 0 no sucesso e -1 no erro.

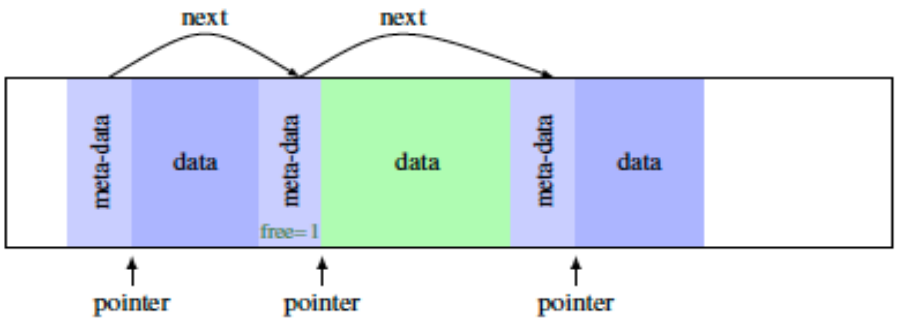
`void *sbrk(intptr_t incr)` – incrementa o segmento de dados em *incr* bytes. Se correr bem, devolve o limite anterior, o que também é o endereço do início da nova zona de memória. Em caso de erro devolve `(void *)-1`. Consulte as páginas dos manuais das chamadas *brk* e *sbrk*.

Malloc/Free[1]

Considere a seguinte versão de uma função de *malloc*:

```
void *myMalloc(size_t size){
    void *p = sbrk(size);
    /* If sbrk fails, we return NULL */
    if ( p == (void*)-1)
        return NULL;
    return p;
}
```

O código acima não deve ser utilizado porque não é possível libertar/reutilizar a memória uma vez alocada. Para construir versões utilizáveis de *malloc/free,* o *heap* deve ser gerido, como por exemplo na figura seguinte (extraído do relatório citado na nota de rodapé):



Nesta figura, cada bloco de memória atribuído é gerido numa lista ligada, tendo por isso duas partes:

- metadados - elementos para a gestão do espaço, incluindo um *pointer* para o bloco seguinte;

[1] A partir de agora, este guia usa partes do relatório "A Malloc Tutorial" por Marwan Burelle, Laboratoire Système et Sécurité, École Pour l'Informatique et les Techniques Avancées (EPITA), 2009.

- o espaço para os dados do processo, devolvido por *malloc*. Note que "pointer" na imagem corresponde ao valor efetivamente devolvido ao processo por *malloc*.

Em C, podemos ter uma implementação desta lista com estes metadados, usando a seguinte declaração:

```
typedef struct s_block *t_block;

struct s_block {
  size_t size;  // size of current block
  t_block next; // pointer to next block
  long long int free;     // flag indicating that the block is free or occupied, 1 or 0
};  // all the structure fields have 64 bits
```

Quando libertado um bloco pode ser marcado livre colocando 1 no campo `free` e mais tarde reutilizado.

Encontrar um bloco de memória para reutilizar usando o algoritmo First Fit

A função *malloc* deve primeiro tentar reutilizar um bloco livre já disponível antes de pedir mais memória ao SO. Suponha que mantemos os seguintes dois *pointers* globais:

- *head* que aponta para o início da lista de blocos;
- *tail* que aponta para o último elemento da lista (isto é útil quando não há nenhum bloco disponível e é necessário adicionar um novo bloco à lista).

A procura do primeiro bloco disponível é feita percorrendo a lista até que um bloco livre com tamanho igual ou maior ao tamanho necessário seja encontrado. Se não houver tal bloco, é devolvido `NULL`.

Aumentando o heap

Quando não há um bloco disponível, é necessário pedir mais memória com a função *sbrk()*, sendo esse espaço então adicionado ao fim da nossa lista de blocos:

```
#define BLOCK_SIZE sizeof(struct s_block)  // metadata size

t_block extend_heap(size_t s){
    t_block b = (t_block) sbrk(BLOCK_SIZE + s);
    if ( b == (void*)-1)  // if sbrk fails, return NULL pointer
         return NULL;
    b->size = s;
    b->next = NULL;
    b->free = 1;
    if (head==NULL) head=b;
    else tail->next = b;
    tail = b;
    return b;
}
```

Trabalho a realizar

Neste trabalho queremos implementar, em assembly *x86_64* as funções

```
void *myMalloc(unsigned int bytesToAllocate);

int myFree(void *address);
```

cujo comportamento é o mesmo das funções da biblioteca do C *malloc* e *free*.

Estão disponíveis no CLIP os seguintes ficheiros:

- um programa principal em C *main.c* completo
- um ficheiro com um esqueleto do programa em assembly chamado *mf.s*. Terá de completar este ficheiro escrevendo o código das funções *myMalloc*, *myFree* e *find_block*, Esta última função tem o comportamento descrito anteriormente em *Encontrar um bloco de memória para reutilizar usando o algoritmo First Fit*
- um *makefile* que gera um ficheiro executável

```
CC=gcc

CFLAGS=-g

all: main
```

```
main: main.o mf.o
	$(CC) $(CFLAGS) -static -o main *.o
main.o: main.c
	$(CC) $(CFLAGS) -c -o main.o main.c
mf.o: mf.s
	as -gstabs -o m.o m.s
clean:
	rm -f *.o  main
```

Como exemplo, uma implementação correta do programa *main* deve produzir na saída um resultado semelhante ao seguinte, quando executada com "**`main 10`**" (note que os endereços podem variar):

```
current block list:
address: 22245376 - 1 bytes (free)
address: 22245401 - 2 bytes (free)
address: 22245427 - 4 bytes (free)
address: 22245455 - 8 bytes (free)
current block list:
address: 22245376 - 1 bytes (in use)
address: 22245401 - 2 bytes (in use)
address: 22245427 - 4 bytes (free)
address: 22245455 - 8 bytes (in use)
address: 22245487 - 10 bytes (in use)
```

Entrega

As instruções para entrega serão divulgadas oportunamente. Deve entregar apenas o ficheiro *mf.s.*