My Malloc Library

Learning Objectives

Upon completion of this assignment, you should be able to:

1. Manipulate C pointers to traverse a process' address space
2. Use pointer arithmetic to adjust pointer references
3. Use casting to dereference memory storage as different types
4. Manually adjust the process heap

New mechanisms you will use include:

- `sbrk()`, enumerated types, type casting and pointer arithmetic

Function Specifications

NAME

`my_malloc(),my_free(), coalesce_free_list(), free_list_begin()`

SYNOPSIS

```
#include "my_malloc.h"

void * my_malloc(uint32_t size);

void my_free(void *ptr);

void coalesce_free_list(void);

FreeListNode free_list_begin( void );

typedef struct freelistnode {
    struct freelistnode *flink;
    uint32_t size;
} * FreeListNode;
```

DESCRIPTION

`my_malloc()`
: allocates `size` bytes of memory

`my_free()`
: deallocates memory referenced by `ptr`, previously allocated by `my_malloc()`

`coalesce_free_list()`
: merges logically adjacent chunks on the free list into single larger chunks

`free_list_begin()`
: retrieves the first node of the free list

RETURN VALUES AND ERRORS

On success, `my_malloc()` returns an 8-byte aligned pointer to the allocated memory.
On failure, `my_malloc()` sets `my_errno` to `MYENOMEM` and returns `NULL`.

On failure, `my_free()` sets `my_errno` to `MYEBADFREEPTR` when passed a `NULL` or non-malloc'd pointer.

`free_list_begin()` returns the first free list node or `NULL` if the list is empty.

Implementation Details

Memory Allocation

We refer to the entire block of memory used to satisfy an allocation request as the "chunk" – the chunk will be bigger than the extent of memory we expect the user to access. The *minimum chunk size* should be the larger of:

16 bytes, OR
the sum of the `struct freelistnode` plus any padding needed to make the chunk size a multiple of 8.

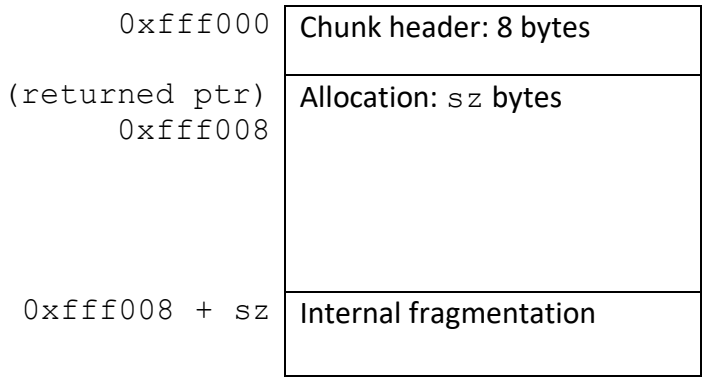
Chunks

Allocated chunks returned by `my_malloc()` will be inflated by:

1. chunk header: 8 bytes
2. internal fragmentation:
    a. any padding necessary to make the chunk size a multiple of 8
    b. potential wastage from using an oversized chunk

Chunk header

Use the 8 bytes just before the address returned by `my_malloc()` for your bookkeeping chunk header. Use the first four bytes for the total chunk size (including header and padding) and the second 4-bytes to designate that the chunk was allocated by `my_malloc()`.



An Example Allocated Chunk

Allocating a chunk

`my_malloc()` first searches the free list for a usable chunk. If no usable chunk is found, call `sbrk()`[1] to extend the heap segment. `my_malloc()` returns a pointer referencing 8 bytes into the chunk, i.e. after the chunk header.

Chunk splitting

Oversized chunks (i.e. larger than needed for the request) must be split into two unless the remainder would be smaller than the *minimum chunk size*. In the former case, the remainder should be added to the free list. In the latter case, `my_malloc()` will return an oversized chunk that suffers small internal fragmentation.

Memory Deallocation

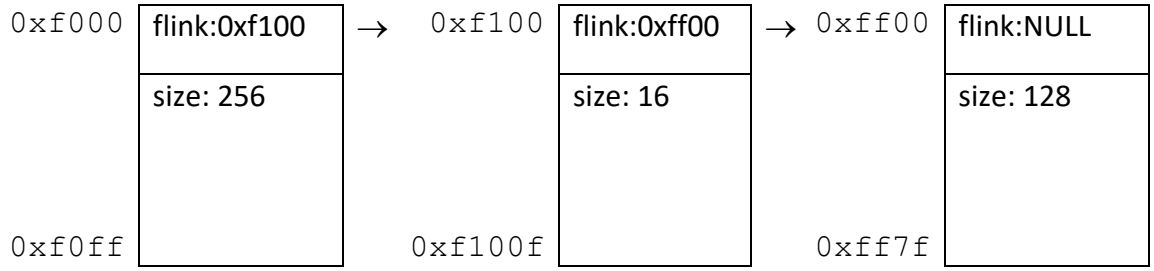
`my_free()` places freed or deallocated chunks of memory onto a free list.

Free List Management

Use `struct freelistnode` in `my_malloc.h` to implement a singly-linked free list to manage free chunks. To add a chunk to the free list, embed a `struct freelistnode` at the beginning of the very same memory chunk[2]. If `ptr` is the address of the chunk:

```
FreeListNode node;
node = (FreeListNode)ptr;
```

Then properly `set node->size` and `node->flink` and insert `node` into the free list. `flink` should be NULL for the last node in the list.



Example Free List

Chunk coalescing

Chunk coalescing is only done when `coalesce_free_list()` is called explicitly: `my_free()` does not coalesce adjacent memory chunks during or after chunk insertion!

[1] For this exercise, we use the simpler yet deprecated sbrk() not the more complex, POSIX-compliant mmap.

[2] This is why minimum chunk size must be the size of struct freelistnode.

Requirements and Constraints

1. **Besides sbrk(), you may not use any other library or system calls.**
2. Always call `sbrk(8192)`[3] except if `my_malloc()` needs more than 8,192 bytes, then call `sbrk()` with the minimum size needed for the new chunk.
3. Assume that other library routines also may make calls to sbrk().
4. You may not use more than 8 bookkeeping bytes per chunk.
5. You may use **one** global variable for the first free list node, not including `my_errno`.
6. Your free list should always be sorted in ascending order by chunk address.
7. Use a *first fit* strategy to search the free list, i.e. return the first usable chunk found.

Submission (via Mimir Classroom)

You must submit only the following files, containing your code solution:

- Sources: `my_malloc.c` and any auxiliary files needed to implement the functions in `my_malloc.h`

Mimir Classroom will immediately build, test and grade your submission. Given you have a limited number of Mimir submissions, we recommend you comprehensively test your solution outside of the Mimir system to identify and fix issues as much as possible. Intermediate testing and debugging is best done on the CS workstations, where you can compare your solution with the instructor's.

Files we provide (that you should not submit):

- `my_malloc.h`: header file containing relevant function and type definitions
- `my_malloc-driver.c`: a sample driver file to write unit tests for your functions
- `Makefile`: a sample makefile that compiles a `my_malloc-driver` executable

[3] You may call `sbrk(0)` to identify the heap's current end.