Homework 3 Dynamic Memory Allocator - CSE 320 - Fall 2023

Professor Eugene Stark

Due Date: Friday 10/27/2023 @ 11:59pm

We **HIGHLY** suggest that you read this entire document, the book chapter, and examine the base code prior to beginning. If you do not read the entire document before beginning, you may find yourself doing extra work.

> *:scream: Start early so that you have an adequate amount of time to test your program!*

> *:scream: The functions `malloc`, `free`, `realloc`, `memalign`, `calloc`, etc., are **NOT ALLOWED** in your implementation. If any of these functions, or any other function with similar functionality is found in your program, you **will receive a ZERO**.*

NOTE: In this document, we refer to a word as 2 bytes (16 bits) and a memory row as 4 words (64 bits). We consider a page of memory to be 4096 bytes (4 KB)

Introduction

You must read **Chapter 9.9 Dynamic Memory Allocation Page 839** before starting this assignment. This chapter contains all the theoretical information needed to complete this assignment. Since the textbook has sufficient information about the different design strategies and implementation details of an allocator, this document will not cover this information. Instead, it will refer you to the necessary sections and pages in the textbook.

Takeaways

After completing this assignment, you will have a better understanding of:

- The inner workings of a dynamic memory allocator
- Memory padding and alignment
- Structs and linked lists in C
- errno (https://linux.die.net/man/3/errno) numbers in C
- Unit testing in C

Overview

You will create an allocator for the x86-64 architecture with the following features:

- Free lists segregated by size class, using first-fit policy within each size class.
- Immediate coalescing of large blocks on free with adjacent free blocks.
- Boundary tags to support efficient coalescing.
- Block splitting without creating splinters.
- Allocated blocks aligned to "double memory row" (16-byte) boundaries.

- Free lists maintained using **last in first out (LIFO)** discipline.
- Use of a prologue and epilogue to avoid edge cases at the end of the heap.
- "Wilderness preservation" heuristic, to avoid unnecessary growing of the heap.

You will implement your own versions of the **malloc**, **realloc**, and **free** functions. You will also implement functions to calculate internal fragmentation and heap utilization.

You will use existing Criterion unit tests and write your own to help debug your implementation.

Free List Management Policy

Your allocator **MUST** use the following scheme to manage free blocks: Free blocks will be stored in a fixed array of `NUM_FREE_LISTS` free lists, segregated by size class (see **Chapter 9.9.14 Page 863** for a discussion of segregated free lists). Each individual free list will be organized as a **circular, doubly linked list** (more information below). The size classes are based on a Fibonacci sequence (1, 2, 3, 5, 8, 13, ...), according to the following scheme: The first free list (at index 0) holds blocks of the minimum size `M` (where `M = 32` for this assignment). The second list holds blocks of size `2M`. The third list holds blocks of size `3M`. The fourth list holds blocks whose sizes are in the interval `(3M, 5M]`. The fifth list holds blocks whose size is in the interval `(5M, 8M]`, and so on. This continues up to the list at index `NUM_FREE_LISTS-2` -(i.e. 8 for this assignment), which contains blocks whose size is greater than `34M`. The last list (at index `NUM_FREE_LISTS-1`; *i.e.* 9) is only used to contain the so-called "wilderness block", which is the free block at the end of the heap that will be extended when the heap is grown. Allocation requests will be satisfied by searching the free lists in increasing order of size class; considering the last list with the wilderness block only if no suitable block has been found in the earlier lists.

> *:nerd: This policy means that the wilderness block will effectively be treated as larger than any other free block, regardless of its actual size. This "wilderness preservation" heuristic tends to prevent the heap from growing unnecessarily.*

Allocation requests will be satisfied by searching the free lists in increasing order of size class.

Block Placement Policy

When allocating memory, use a **segregated fits policy**, as follows. First, the smallest size class that is sufficiently large to satisfy the request is determined. The free lists are then searched, starting from the list for the determined size class and continuing in increasing order of size, until a nonempty list is found. The request is then satisfied by the first block in that list that is sufficiently large; *i.e.* a **first-fit policy** (discussed in **Chapter 9.9.7 Page 849**) is applied within each individual free list. The last list, which contains only the wilderness block (if there currently is one), is considered only if no sufficiently large block is found in any of the earlier lists.

If there is no block in the free lists that is large enough to satisfy the allocation request, `sf_mem_grow` should be called to extend the heap by an additional page of memory. After coalescing this page with any free block that immediately precedes it (*i.e.* with the wilderness block), you should attempt to use the resulting block of memory to satisfy the allocation request; splitting it if it is too large and no "splinter" (*i.e.* a remainder smaller than the minimum block size) would result. If the block of memory is still not large enough, another call to `sf_mem_grow` should be made; continuing to grow the heap until either a large enough block is obtained or the return value from `sf_mem_grow` indicates that there is no more memory.

As discussed in the book, segregated free lists allow the allocator to approximate a best-fit policy, with lower overhead than would be the case if an exact best-fit policy were implemented.

Splitting Blocks & Splinters

Your allocator must split blocks at allocation time to reduce the amount of internal fragmentation. Details about this feature can be found in **Chapter 9.9.8 Page 849**. Due to alignment and overhead constraints, there will be a minimum useful block size that the allocator can support. **For this assignment, pointers returned by the allocator in response to allocation requests are required to be aligned to 16-byte boundaries**; *i.e.* the pointers returned will be addresses that are multiples of 2^4. The 16-byte alignment requirement implies that the minimum block size for your allocator

will be 32 bytes. No "splinters" of smaller size than this are ever to be created. If splitting a block to be allocated would result in a splinter, then the block should not be split; rather, the block should be used as-is to satisfy the allocation request (*i.e.*, you will "over-allocate" by issuing a block slightly larger than that required).

> *:thinking: How do the alignment and overhead requirements constrain the minimum block size? As you read more details about the format of a block header, block footer, and alignment requirements, you should try to answer this question.*

Freeing a Block

When a block is freed, an attempt should first be made to **coalesce** the block with any free block that immediately precedes or follows it in the heap. (See **Chapter 9.9.10 Page 850** for a discussion of the coalescing procedure.) Once the block has been coalesced, it should be inserted at the **front** of the free list for the appropriate size class (based on the size after coalescing). The reason for performing coalescing is to combat the external fragmentation that would otherwise result due to the splitting of blocks upon allocation.

Block Headers & Footers

In **Chapter 9.9.6 Page 847 Figure 9.35**, a block header is defined as 2 words (32 bits) to hold the block size and allocated bit. In this assignment, the header will be 4 words (i.e. 64 bits or 1 memory row). The header fields will be similar to those in the textbook but with some differences. Each free block will also have a footer, which occupies the last memory row of the block. The footer of a free block contains exactly the same information as the header.

Block Header Format:

```
+---------------------------+-------------------------------+--------+---------+---------+ <- header
|       payload_size        |          block_size           | alloc  |prv alloc| unused  |
|          (0/1)            |     (4 LSB's implicitly 0)    | (0/1)  |  (0/1)  |   (0)   |
|        (32 bits)          |          (28 bits)            | 1 bit  |  1 bit  | 2 bits |
+-----------------------------------------------------------+--------+---------+---------+ <- (aligned)
```

- The `payload_size` field, which occupies the four most-significant bytes of the header of an allocated block, will be used to store the payload size that was requested by the client for that block. In a free block this field will be zero. Normally an allocator would not store the payload size, but we are going to use it to enable us to calculate heap utilization.
- The `block_size` field gives the number of bytes for the **entire** block (including header/footer, payload, and padding). It occupies the entire 64 bits of the block header or footer, except that two of the four least-significant bits of the block size, which would normally always be zero due to alignment requirements, are used to store the allocation status (free/allocated) of that block and of the immediately preceding block in the heap. This means that these two bits have to be masked when retrieving the block size from the header and when the block size is stored in the header the previously existing values of these two bits have to be preserved.
- The `alloc` bit (bit 3, mask 0x8) is a boolean. It is 1 if the block is allocated and 0 if it is free.
- The `prev_alloc` (bit 2, mask 0x4) is also a boolean. It is 1 if the **immediately preceding** block in the heap is allocated and 0 if it is not.

> *:thinking: Here is an example of determining the block size required to satisfy a particular requested payload size. Suppose the requested size is 25 bytes. An additional 8 bytes will be required to store the block header, which must always be present. This means that a block of at least 33 bytes must be used, however due to alignment requirements this has to be rounded up to the next multiple of the alignment size. If the alignment size were 16 bytes (which would be just large enough to enable the memory returned by the allocator to store in an aligned fashion any of the basic data types supported by the x86-64 architecture), then a block of at least 48 bytes would have to be used. As a result, there would be 15 bytes of "padding" at the end of the payload area, which contributes to internal fragmentation. Besides the header, it is also necessary to store a footer, as well and next and previous links for the freelist. These will take an additional 24 bytes of space. But the payload area is 40 bytes (25 bytes plus 15 bytes of padding), which is certainly bigger than 24 bytes, so a block of total size 48 would be fine. Note that a block cannot be smaller than 32 bytes, as there there would not then be enough space to store the header, footer, and freelist links.*

Getting Started

Fetch and merge the base code for `hw3` as described in `hw0` from the following link: https://gitlab02.cs.stonybrook.edu/cse320/hw3 (https://gitlab02.cs.stonybrook.edu/cse320/hw3)

Remember to use the `--strategy-option=theirs` flag with the `git merge` command as described in the `hw1` doc to avoid merge conflicts in the Gitlab CI file.

Directory Structure

```
.
├── .gitignore
├── .gitlab-ci.yml
└── hw3
    ├── hw3.sublime-project
    ├── include
    │   ├── debug.h
    │   └── sfmm.h
    ├── lib
    │   └── sfutil.o
    ├── Makefile
    ├── src
    │   ├── main.c
    │   └── sfmm.c
    └── tests
        └── sfmm_tests.c
```

The `lib` folder contains the object file for the `sfutil` library. This library provides you with several functions to aid you with the implementation of your allocator. **Do NOT delete this file as it is an essential part of your homework assignment.**

The provided `Makefile` creates object files from the `.c` files in the `src` directory, places the object files inside the `build` directory, and then links the object files together, including `lib/sfutil.o`, to make executables that are stored to the `bin` directory.

Note: `make clean` will not delete `sfutil.o` or the `lib` folder, but it will delete all other contained `.o` files.

The `sfmm.h` header file contains function prototypes and defines the format of the various data structures that you are to use.

> *:scream:* ***DO NOT modify `sfmm.h` or the Makefile.*** *Both will be replaced when we run tests for grading. If you wish to add things to a header file, please create a new header file in the* `include` *folder*

All functions for your allocator (`sf_malloc`, `sf_realloc`, `sf_free`, `sf_fragmentation`, and `sf_utilization`) **must** be implemented in `src/sfmm.c`.

The program in `src/main.c` contains a basic example of using the allocation functions. Running `make` will create a `sfmm` executable in the `bin` directory. This can be run using the command `bin/sfmm`.

> *Any functions other than* `sf_malloc`*,* `sf_free`*,* `sf_realloc`*,* `sf_fragmentation`*, and* `sf_utilization` ***WILL NOT*** *be graded.*

Allocation Functions

You will implement the following three functions in the file `src/sfmm.c`. The file `include/sfmm.h` contains the prototypes and documentation found here.

Standard C library functions set `errno` when there is an error. To avoid conflicts with these functions, your allocation functions will set `sf_errno`, a variable declared as `extern` in `sfmm.h`.

```
/*
 * This is your implementation of sf_malloc. It acquires uninitialized memory that
 * is aligned and padded properly for the underlying system.
 *
 * @param size The number of bytes requested to be allocated.
 *
 * @return If size is 0, then NULL is returned without setting sf_errno.
 * If size is nonzero, then if the allocation is successful a pointer to a valid region of
 * memory of the requested size is returned.  If the allocation is not successful, then
 * NULL is returned and sf_errno is set to ENOMEM.
 */
void *sf_malloc(size_t size);

/*
 * Resizes the memory pointed to by ptr to size bytes.
 *
 * @param ptr Address of the memory region to resize.
 * @param size The minimum size to resize the memory to.
 *
 * @return If successful, the pointer to a valid region of memory is
 * returned, else NULL is returned and sf_errno is set appropriately.
 *
 *   If sf_realloc is called with an invalid pointer sf_errno should be set to EINVAL.
 *   If there is no memory available sf_realloc should set sf_errno to ENOMEM.
 *
 * If sf_realloc is called with a valid pointer and a size of 0 it should free
 * the allocated block and return NULL without setting sf_errno.
 */
void* sf_realloc(void *ptr, size_t size);

/*
 * Marks a dynamically allocated region as no longer in use.
 * Adds the newly freed block to the free list.
 *
 * @param ptr Address of memory returned by the function sf_malloc.
 *
 * If ptr is invalid, the function calls abort() to exit the program.
 */
void sf_free(void *ptr);
```

> *:scream: Make sure these functions have these exact names and arguments. They must also appear in the correct file. If you do not name the functions correctly with the correct arguments, your program will not compile when we test it.* ***YOU WILL GET A ZERO***

Statistics Functions

Besides the allocation functions discussed above, you are to implement the following two functions that return statistics about the memory utilization of the allocator:

```
/*
 * Get the current amount of internal fragmentation of the heap.
 *
 * @return  the current amount of internal fragmentation, defined to be the
 * ratio of the total amount of payload to the total size of allocated blocks.
 * If there are no allocated blocks, then the returned value should be 0.0.
 */
double sf_fragmentation();

/*
 * Get the peak memory utilization for the heap.
 *
 * @return  the peak memory utilization over the interval starting from the
 * time the heap was initialized, up to the current time.  The peak memory
 * utilization at a given time, as defined in the lecture and textbook,
 * is the ratio of the maximum aggregate payload up to that time, divided
 * by the current heap size.  If the heap has not yet been initialized,
 * this function should return 0.0.
 */
double sf_utilization();
```

These functions are also to be implemented in `sfmm.c`.

> *Any functions other than `sf_malloc`, `sf_free`, `sf_realloc`, `sf_fragmentation`, and `sf_utilization` **WILL NOT** be graded.*

Initialization Functions

In the `lib` directory, we have provided you with the `sfutil.o` object file. When linked with your program, this object file allows you to access the `sfutil` library, which contains the following functions:

```
/*
 * @return The starting address of the heap for your allocator.
 */
void *sf_mem_start();

/*
 * @return The ending address of the heap for your allocator.
 */
void *sf_mem_end();

/*
 * This function increases the size of your heap by adding one page of
 * memory to the end.
 *
 * @return On success, this function returns a pointer to the start of the
 * additional page, which is the same as the value that would have been returned
 * by sf_mem_end() before the size increase.  On error, NULL is returned
 * and sf_errno is set to ENOMEM.
 */
void *sf_mem_grow();

/* The size of a page of memory returned by sf_mem_grow(). */
#define PAGE_SZ ((size_t)4096)
```

> :scream: As these functions are provided in a pre-built .o file, the source
> is not available to you. You will not be able to debug these using gdb.
> You must treat them as black boxes.

sf_mem_grow

The function `sf_mem_grow` is to be invoked by `sf_malloc`, at the time of the
first allocation request to obtain an initial free block, and on subsequent allocations
when a large enough block to satisfy the request is not found.
For this assignment, your implementation **MUST ONLY** use `sf_mem_grow` to
extend the heap. **DO NOT** use any system calls such as **brk** or **sbrk**
to do this.

Function `sf_mem_grow` returns memory to your allocator in pages.
Each page is 4096 bytes (4 KB) and there are a limited, small number of pages
available (the actual number may vary, so do not hard-code any particular limit
into your program). Each call to `sf_mem_grow` extends the heap by one page and
returns a pointer to the new page (this will be the same pointer as would have
been obtained from `sf_mem_end` before the call to `sf_mem_grow`.

The `sf_mem_grow` function also keeps track of the starting and ending addresses
of the heap for you. You can get these addresses through the `sf_mem_start` and
`sf_mem_end` functions.

> :smile: A real allocator would typically use the **brk**/**sbrk** system calls
> calls for small memory allocations and the **mmap**/**munmap** system calls
> for large allocations. To allow your program to use other functions provided by
> glibc, which rely on glibc's allocator (*i.e.* `malloc`), we have provided
> `sf_mem_grow` as a safe wrapper around **sbrk**. This makes it so your heap and
> the one managed by glibc do not interfere with each other.

Implementation Details

Memory Row Size

The table below lists the sizes of data types (following Intel standard terminlogy)
on x86-64 Linux Mint:

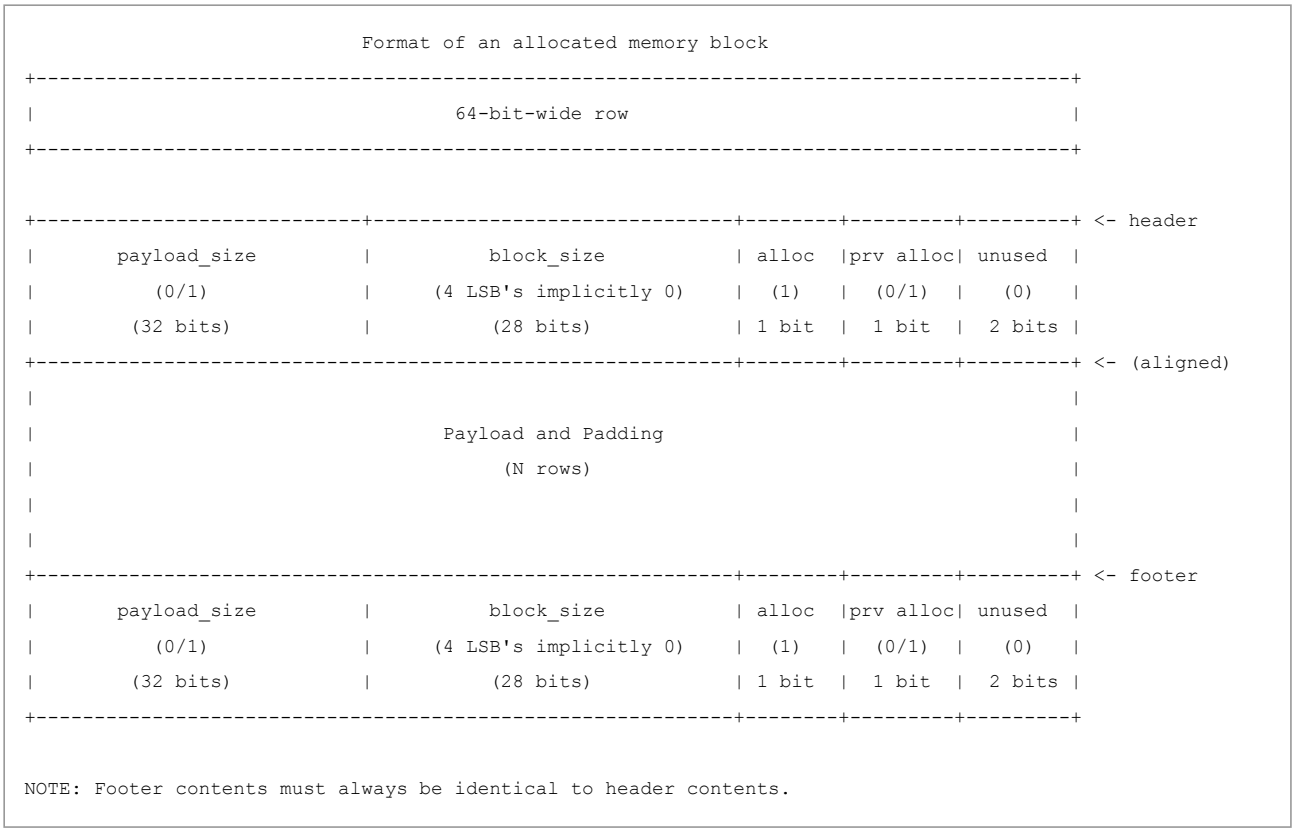
C declaration	Data type	x86-64 Size (Bytes)
char	Byte	1
short	Word	2
int	Double word	4
long int	Quadword	8
unsigned long	Quadword	8
pointer	Quadword	8
float	Single precision	4
double	Double precision	8
long double	Extended precision	16

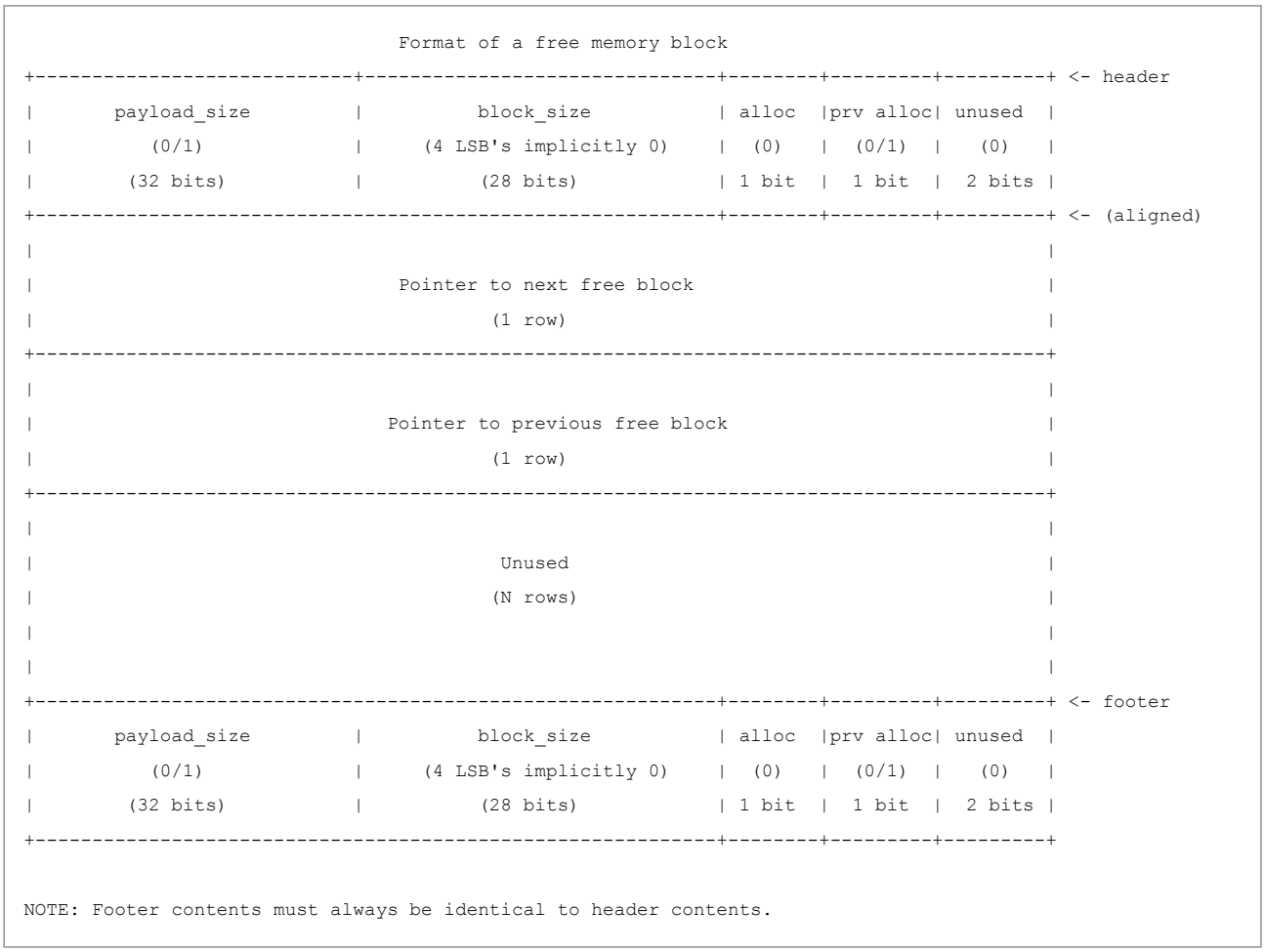
> :nerd: You can find these sizes yourself using the sizeof operator.
> For example, `printf("%lu\n", sizeof(int))` prints 4.

In this assignment we will assume that each "memory row" is 8 bytes (64 bits) in size.
All pointers returned by your `sf_malloc` are to be 16-byte aligned; that is, they will be
addresses that are multiples of 16. This requirement permits such pointers to be used to
store any of the basic machine data types in a "naturally aligned" fashion.
A value stored in memory is said to be *naturally aligned* if the address at which it
is stored is a multiple of the size of the value. For example, an `int` value is
naturally aligned when stored at an address that is a multiple of 4. A `long double` value
is naturally aligned when stored at an address that is a multiple of 16.
Keeping values naturally aligned in memory is a hardware-imposed requirement for some
architectures, and improves the efficiency of memory access in other architectures.

Block Header & Footer Fields

The various header and footer formats are specified in `include/sfmm.h`:


Format of an allocated memory block
64-bit-wide row
payload_size
(0/1)
(32 bits)
block_size
(4 LSB's implicitly 0)
(28 bits)
alloc
(1)
1 bit
prv alloc
(0/1)
1 bit
unused
(0)
2 bits
<- header
<- (aligned)
Payload and Padding
(N rows)
<- footer
NOTE: Footer contents must always be identical to header contents.


Format of a free memory block
payload_size
(0/1)
(32 bits)
block_size
(4 LSB's implicitly 0)
(28 bits)
alloc
(0)
1 bit
prv alloc
(0/1)
1 bit
unused
(0)
2 bits
<- header
<- (aligned)
Pointer to next free block
(1 row)
Pointer to previous free block
(1 row)
Unused
(N rows)
<- footer
NOTE: Footer contents must always be identical to header contents.

The `sfmm.h` header file contains C structure definitions corresponding to the above diagrams:

```c
typedef size_t sf_header;
typedef size_t sf_footer;

/* Structure of a block. */
typedef struct sf_block {
    sf_footer prev_footer;  // NOTE: This actually belongs to the *previous* block.
    sf_header header;       // This is where the current block really starts.
    union {
        /* A free block contains links to other blocks in a free list. */
        struct {
            struct sf_block *next;
            struct sf_block *prev;
        } links;
        /* An allocated block contains a payload (aligned), starting here. */
        char payload[0];   // Length varies according to block size.
    } body;
} sf_block;
```

For `sf_block`, the `body` field is a `union`, which has been used to emphasize the difference between the information contained in a free block and that contained in an allocated block. If the block is free, then its `body` has a `links` field, which is a `struct` containing `next` and `prev` pointers. If the block is allocated, then its `body` does not have a `links` field, but rather has a `payload`, which starts at the same address that the `links` field would have started if the block were free. The size of the `payload` is obviously not zero, but as it is variable and only determined at run time, the `payload` field has been declared to be an array of length 0 just to enable the use of `bp->body.payload` to obtain a pointer to the payload area, if `bp` is a pointer to `sf_block`.

> *:thumbsup: You can use casts to convert a generic pointer value to one of type `sf_block *` or `sf_header *`, in order to make use of the above structure definitions to easily access the various fields. You can even cast an integer value to these pointer types; this is sometimes required when calculating the locations of blocks in the heap.*

Each block must have a valid footer whose contents are identical to the header contents.

> *:scream: Note that the `prev_footer` field in the `sf_block` structure is actually part of the **previous** block in the heap. In order to initialize an `sf_block` pointer to correctly access the fields of a block, it is necessary to compute the address of the footer of the immediately preceding block in the heap and then cast that address to type `sf_block *`. The footer of a particular block can be obtained by first getting an `sf_block *` pointer for that block and then using the contained information (i.e. the block size) to obtain the `prev_footer` field of the **next** block in the heap. The `sf_block` structure has been specified this way so as to permit it to be defined with a fixed size, even though the payload size is unknown and will vary.*

Free List Heads

In the file `include/sfmm.h`, you will see the following declaration:

```
#define NUM_FREE_LISTS 10
struct sf_block sf_free_list_heads[NUM_FREE_LISTS];
```

The array `sf_free_list_heads` contains the heads of the free lists, which are maintained as **circular, doubly linked lists**. Each node in a free list contains a `next` pointer that points to the next node in the list, and a `prev` pointer that points the previous node. For each index `i` with `0 <= i < NUM_FREE_LISTS` the variable `sf_free_list_head[i]` is a dummy, "sentinel" node, which is used to connect the beginning and the end of the list at index `i`. This sentinel node is always present and (aside from its `next` and `free` pointers) does **not** contain any other data. If the list is empty, then the fields `sf_freelist_heads[i].body.links.next` and `sf_freelist_heads[i].body.links.prev` both contain `&sf_freelist_heads[i]` (*i.e.* the sentinel node points back to itself). If the list is nonempty, then `sf_freelist_heads[i].body.links.next` points to the first node in the list and `sf_freelist_heads[i].body.links.prev` points to the last node in the list. Inserting into and deleting from a circular doubly linked list is done in the usual way, except that, owing to the use of the sentinel, there are no edge cases for inserting or removing at the beginning or the end of the list. If you need a further introduction to this data structure, you can readily find information on it by googling ("circular doubly linked lists with sentinel").

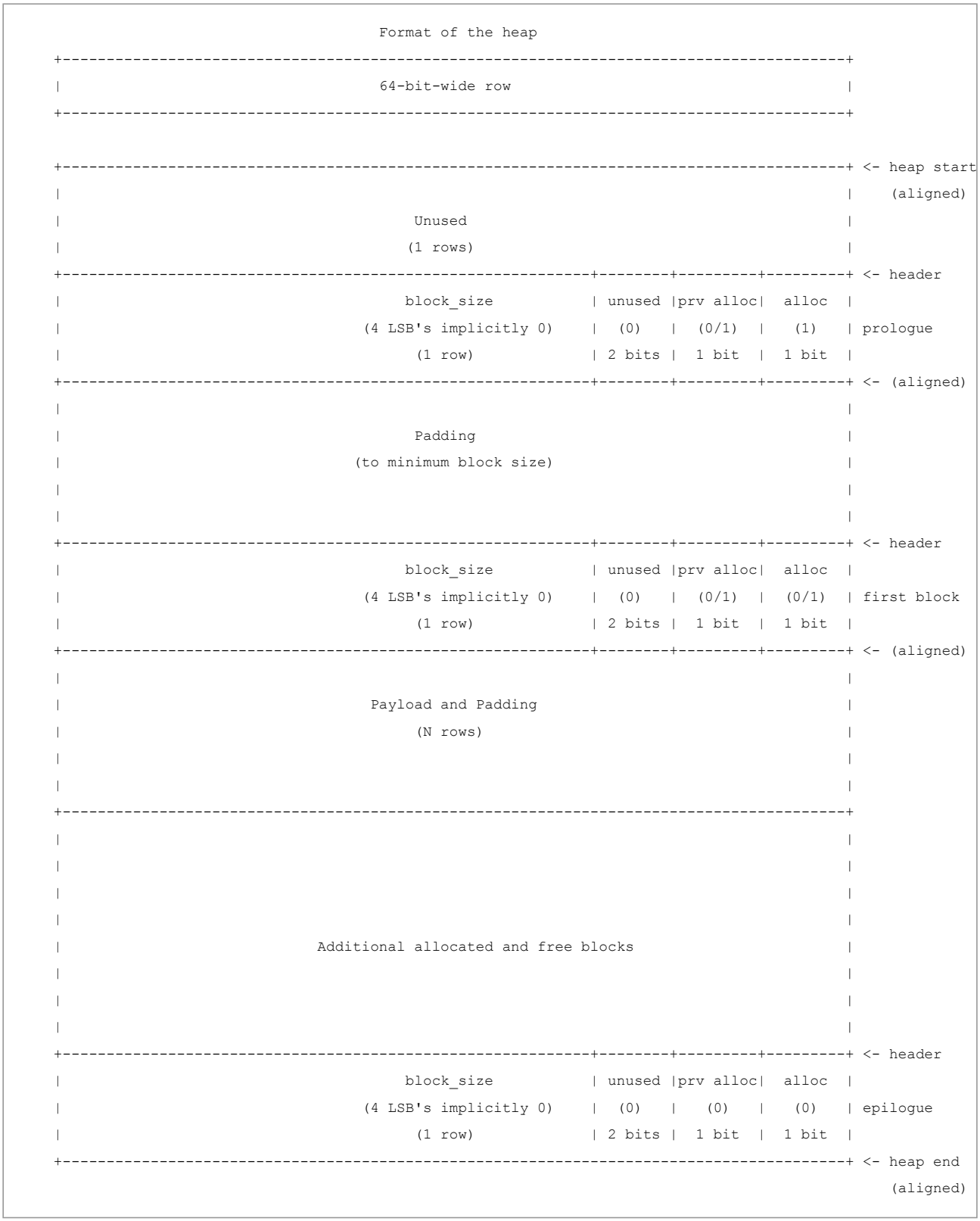
> *:scream: You **MUST** use the `sf_free_list_heads` array for the heads of your free lists and you **MUST** maintain these lists as circular, doubly linked lists. The helper functions discussed later, as well as the unit tests, will assume that you have done this when accessing your free lists.*

> *:scream: Note that the head of a freelist must be initialized before the list can be used. The initialization is accomplished by setting the `next` and `prev` pointers of the sentinel node to point back to the node itself.*

Overall Structure of the Heap

The overall structure of the allocatable area of your heap will be a sequence of allocated and free blocks. Your heap should also contain a **prologue** and an **epilogue** (as described in the book, **page 855**) to avoid edge cases when coalescing blocks. You may assume that the pointer returned by `sf_mem_start` is 16-byte aligned. In order for the first heap block to have its payload area also 16-byte aligned, it is necessary for the header of the first heap block to be stored at an offset of 8 bytes from the start of the heap. Thus, the first 8 bytes of the heap will be unused. The last 8 bytes of the heap will constitute the epilogue. These 8 bytes are reserved to become the header of the next block when the heap is extended using `sf_mem_grow`. The payload area of this new block will thus be aligned on a 16-byte boundary and the heap can be grown seamlessly without any wasted space in between successive pages.

The overall organization of the heap is as shown below:



The heap begins with unused "padding", so that the header of each block will start `sizeof(sf_header)` bytes before an alignment boundary. After the padding is the "prologue", which is an allocated block of minimum size with an unused payload area. The prologue is never used to satisfy allocation requests and it is never freed. At the end of the heap is an "epilogue", which consists only of an allocated header, with block size set to 0. The epilogue is never used to satisfy an allocation request and it is never freed. Whenever the heap is extended, a new epilogue is created at the end of the newly added region and the old epilogue becomes the header of the new block. This is as described in the book. We do not make any separate C structure definitions for the prologue or epilogue. They can be manipulated using the existing `sf_block` structure, though care must be taken not to access fields that are not valid for this special block (*i.e.* anything other than `header` and `prev_footer` for the epilogue).

As your heap is initially empty, at the time of the first call to `sf_malloc` you will need to make one call to `sf_mem_grow` to obtain a page of memory within which to set up the prologue and initial epilogue. The remainder of the memory in this first page should then be inserted into the free list as a single block.

Notes on sf_malloc

When implementing your `sf_malloc` function, first determine if the request size is 0. If so, then return `NULL` without setting `sf_errno`. If the request size is non-zero, then you should determine the size of the block to be allocated by adding the header size and the size of any necessary padding to reach a size that is a multiple of 16 to maintain proper alignment. Remember also that the block has to be big enough to store the footer as well as the `next` and `prev` pointers when the block is free. As these fields are not present in an allocated block this space can (and should) be overlapped with the payload area. As has already been discussed, the above constraints lead to a minimum block size of 32 bytes, so you should not attempt to allocate any block smaller than this. After having determined the required block size, you should determine the index of the first free list that would be able to satisfy a request of that size. Search that free list from the beginning until the first sufficiently large block is found. If there is no such block, continue with the next larger size class. If a big enough block is found, then after splitting it (if it will not leave a splinter), you should insert the remainder part back into the appropriate freelist. When splitting a block, the "lower part" (i.e. locations with lower-numbered addresses) should be used to satisfy the allocation request and the "upper part" (i.e. locations with higher-numbered addresses) should become the remainder.

If a big enough block is not found in any of the freelists, then you must use `sf_mem_grow` to request more memory (for requests larger than a page, more than one such call might be required). If your allocator ultimately cannot satisfy the request, your `sf_malloc` function must set `sf_errno` to `ENOMEM` and return `NULL`.

Notes on sf_mem_grow

After each call to `sf_mem_grow`, you must attempt to coalesce the newly allocated page with any free block immediately preceding it, in order to build blocks larger than one page. Insert the new block at the beginning of the appropriate freelist.

Note: Do not coalesce past the beginning or end of the heap.

Notes on sf_free

When implementing `sf_free`, you must first verify that the pointer being passed to your function belongs to an allocated block. This can be done by examining the fields in the block header. In this assignment, we will consider the following cases to be invalid pointers:

- The pointer is `NULL`.
- The pointer is not 16-byte aligned.
- The block size is less than the minimum block size of 32.
- The block size is not a multiple of 16
- The header of the block is before the start of the first block of the heap, or the footer of the block is after the end of the last block in the heap.
- The `allocated` bit in the header is 0.
- The `prev_alloc` field in the header is 0, indicating that the previous block is free, but the `alloc` field of the previous block header is not 0.

If an invalid pointer is passed to your function, you must call `abort` to exit the program. Use the man page for the `abort` function to learn more about this.

After confirming that a valid pointer was given, you must free the block. Insert the block at the *front* of the appropriate free list, after coalescing with any adjacent free block.

Note that blocks in a free list must **not** be marked as allocated, and they must have a valid footer with contents identical to the block header.

Notes on sf_realloc

When implementing your `sf_realloc` function, you must first verify that the pointer passed to your function is valid. The criteria for pointer validity are the same as those described in the 'Notes on sf_free' section above. If the pointer is valid but the size parameter is 0, free the block and return `NULL`.

After verifying the parameters, consider the cases described below. Note that in some cases, `sf_realloc` is more complicated than calling `sf_malloc` to allocate more memory, `memcpy` to move the old memory to the new memory, and `sf_free` to free the old memory.

Reallocating to a Larger Size

When reallocating to a larger size, always follow these three steps:

1. Call `sf_malloc` to obtain a larger block.
2. Call `memcpy` to copy the data in the block given by the client to the block returned by `sf_malloc`. Be sure to copy the entire payload area, but no more.
3. Call `sf_free` on the block given by the client (inserting into a freelist and coalescing if required).
4. Return the block given to you by `sf_malloc` to the client.

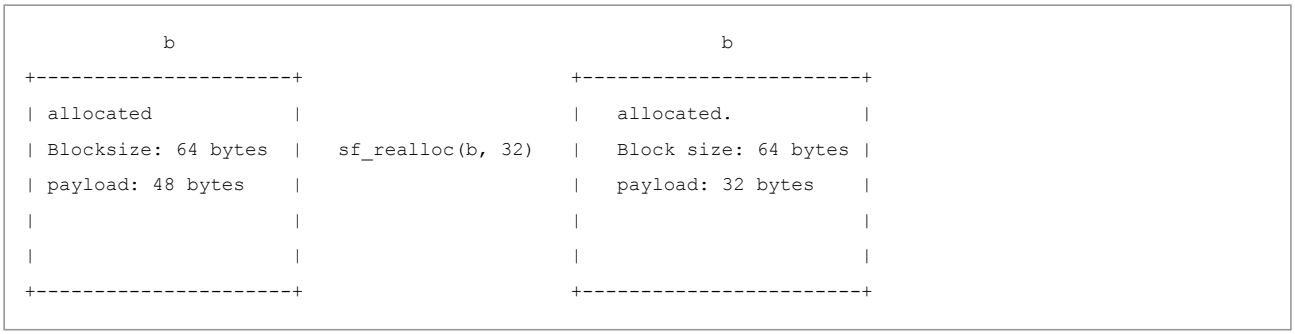
If `sf_malloc` returns `NULL`, `sf_realloc` must also return `NULL`. Note that you do not need to set `sf_errno` in `sf_realloc` because `sf_malloc` should take care of this.

Reallocating to a Smaller Size

When reallocating to a smaller size, your allocator must use the block that was passed by the caller. You must attempt to split the returned block. There are two cases for splitting:

- Splitting the returned block results in a splinter. In this case, do not split the block. Leave the splinter in the block, update the header field if necessary, and return the same block back to the caller.

Example:

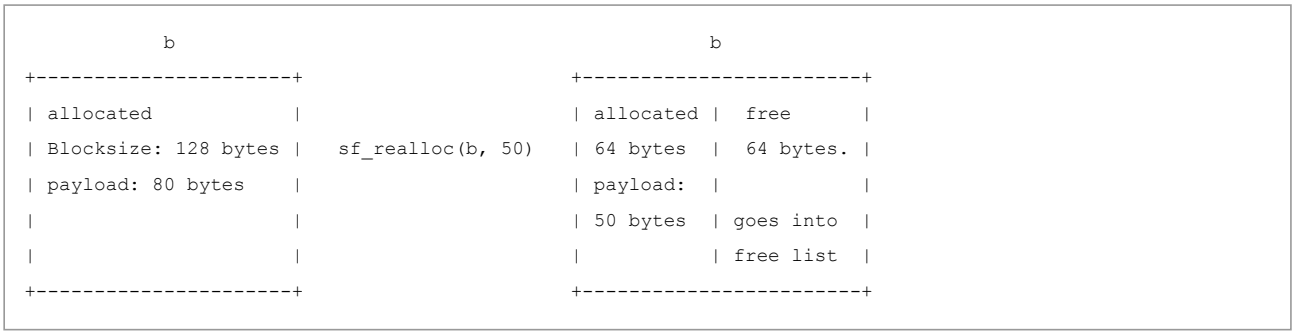


In the example above, splitting the block would have caused a 24-byte splinter. Therefore, the block is not split.

- The block can be split without creating a splinter. In this case, split the block and update the block size fields in both headers. Free the remainder block by inserting it into the appropriate free list (after coalescing, if possible). Return a pointer to the payload of the now-smaller block to the caller.

Note that in both of these sub-cases, you return a pointer to the same block that was given to you.

Example:



Helper Functions

The `sfutil` library additionally contains the following helper functions, which should be self explanatory. They all output to `stderr`.

```
void sf_show_block(sf_block *bp);
void sf_show_free_list(int index);
void sf_show_free_lists();
void sf_show_heap();
```

We have provided these functions to help you visualize your free lists and allocated blocks.

Unit Testing

For this assignment, we will use Criterion to test your allocator. We have provided a basic set of test cases and you will have to write your own as well.

You will use the Criterion framework alongside the provided helper functions to ensure your allocator works exactly as specified.

In the `tests/sfmm_tests.c` file, there are nine unit test examples. These tests check for the correctness of `sf_malloc`, `sf_realloc`, and `sf_free`. We provide some basic assertions, but by no means are they exhaustive. It is your job to ensure that your header/footer bits are set correctly and that blocks are allocated/freed as specified.

Compiling and Running Tests

When you compile your program with `make`, a `sfmm_tests` executable will be created in the `bin` folder alongside the `main` executable. This can be run with `bin/sfmm_tests`. To obtain more information about each test run, you can use the verbose print option: `bin/sfmm_tests --verbose`. You might also find it helpful to suppress the running of tests concurrently by giving the `--j1` option. It is also possible to restrict the set of tests that are run. For example, using `--filter suite_name/test_name` will only run the test named `test_name` in test suite `suite_name` (if there is such a test, otherwise it will run no tests).

Writing Criterion Tests

The first test `malloc_an_int` tests `sf_malloc`. It allocates space for an integer and assigns a value to that space. It then runs an assertion to make sure that the space returned by `sf_malloc` was properly assigned.

```
cr_assert(*x == 4, "sf_malloc failed to give proper space for an int!");
```

The string after the assertion only gets printed to the screen if the assertion failed (i.e. `*x != 4`). However, if there is a problem before the assertion, such as a SEGFAULT, the unit test will print the error to the screen and continue to run the rest of the unit tests.

For this assignment **you must write 5 additional unit tests which test new functionality and add them to `sfmm_tests.c` below the following comment:**

```
//############################################
//STUDENT UNIT TESTS SHOULD BE WRITTEN BELOW
//DO NOT DELETE THESE COMMENTS
//############################################
```

> *For additional information on Criterion library, take a look at the official documentation located here (http://criterion.readthedocs.io/en/master/)! This documentation is VERY GOOD.*

Hand-in instructions

Make sure your directory tree looks like it did originally after merging the basecode, and and that your homework compiles.

This homework's tag is: `hw3`

```
$ git submit hw3
```

A Word to the Wise

This program will be very difficult to get working unless you are extremely disciplined about your coding style. Think carefully about how to modularize your code in a way that makes it easier to understand and avoid mistakes. Verbose, repetitive code is error-prone and **evil!** When writing your program try to comment as much as possible. Format the code consistently. It is much easier for your TA and the professor to help you if we can quickly figure out what your code does.