# Project 3: Memory Allocation

Intermediate I Deliverable Due: Mar 26th @ 11:59pm
Intermediate II Deliverable Due: Apr 9th @ 11:59pm
Final Deliverable Due: Apr 14th @ 11:59pm

## 1 Introduction

Dynamically allocated memory, otherwise known as heap space or the heap, serves as a critical component of many modern-day computer programs. Not to be confused with the heap data structure, the heap is simply an area of memory that (1) is set aside for use by the dynamic memory needs of a program during runtime and (2) is separate from the stack or statically-allocated variables. In C++ or Java, have you ever used the `new` keyword? In C, have you ever used `malloc`? These commands allocate memory dynamically, within the heap space. The context of this project is covering a relatively simple implementation of `malloc`, in order to understand the performance and implications of dynamically allocated memory in comparison to stack or static allocation, how an operating system is able to satisfy user requests at runtime, and what happens when these requests cannot be satisfied.

Specifically, you will implement a **best fit** memory allocator that runs within the provided test harness. Your `kmalloc` (for kernel memory allocation), `kfree`, and other supporting routines will allocate memory regions within a provided memory region, which simulates the memory address space of a kernel. A series of tests will be used to evaluate the correctness of your solution and will form a large portion of your grade on the assignment. See the grading section for more details.

This project is a team project. You can choose your team of upto four members. You must send your team name to the grader, Ming Li <mili@mymail.mines.edu> within 24 hours.

One team grade will be assigned, and then your individual grade will be increased/decreased based on a team evaluation form (will be made available on Piazza). Each team member needs to separately complete this form; to complete, send details of the form (in text) to Ming Li. This form MUST be submitted within one week after the project is due via email. *If you do not submit an evaluation form, your team will be penalized 10 points.*

## 2 Theory

### 2.1 Free Lists

The primary concept behind our implementation of dynamic memory is the **free list**. The free list is a list of free memory regions, known as chunks, within the heap. In our context, the heap need not be an actual heap data structure and, indeed, in this case, it is not; the heap simply refers to an area of memory set aside for allocation. In this project, you will maintain these free lists and select the "best" memory area to use for an allocation request. "Best" may refer to returning the smallest available chunk that meets the target size, the chunk that is predicted to cause the least amount of fragmentation, or perhaps the piece of memory that the allocator is able to find fastest.

For this project, we will use the best fit placement algorithm as defined in Chapter 7 of the textbook. Recall that the best fit placement algorithm finds, in the memory space available to it, the smallest section of contiguous memory, called a chunk, that is the same size as or greater than the size of the memory request (plus any overhead incurred by the allocation). That is, this algorithm finds the smallest piece of memory that can service the request.

This algorithm can be implemented in many different ways. One method is via the concept of binning with coalescing. This will be the focus of our implementation.

## 2.2 Binning

Binning is the process by which the memory allocator maintains a large number of fixed-width bins. Each bin represents a memory size, and the bins are spaced logarithmically. That is, suppose we have an array of size 5, as shown in Figure 1 and Figure 2, and each bin holds free chunks between size 2ˆi (inclusive) and 2ˆi+1 (exclusive) in size. Under this system, the bin at index 0 holds a list of all free chunks between size 1 byte and 2 bytes (exclusive); index 1 holds a list of all free chunks between size 2 bytes and 4 bytes (exclusive); index 3 holds a list of all free chunks between size 4 bytes and 8 bytes (exclusive); etc. We note that the first bin and last bin are special; specifically, the first bin holds all sizes smaller than its upper bound and the last bin holds all sizes larger than its lower bound.

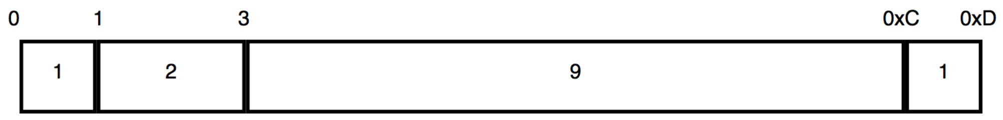

Figure 1: An example memory layout.

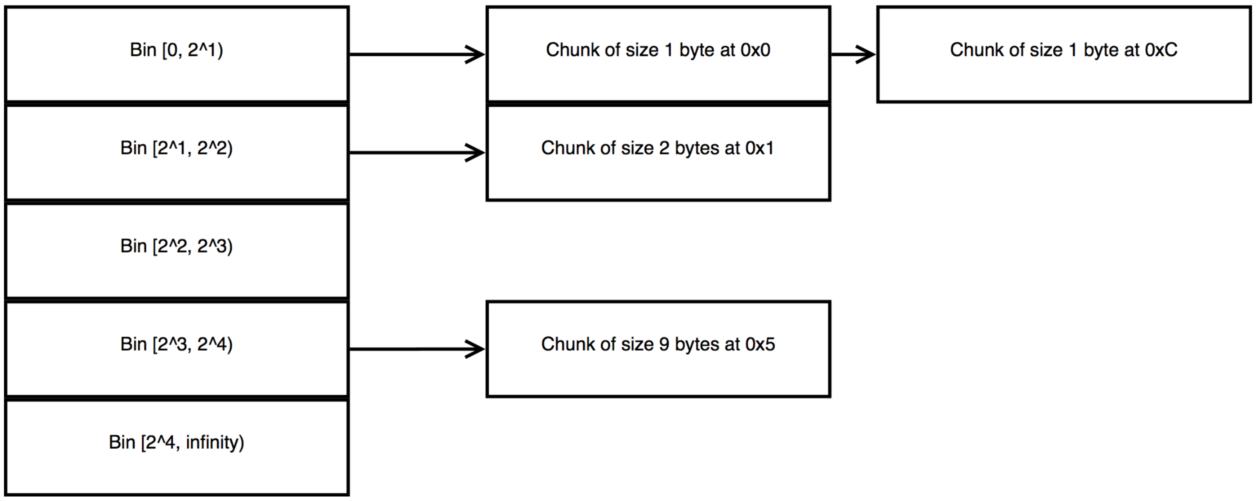
Bin [0, 2^1)
Bin [2^1, 2^2)
Bin [2^2, 2^3)
Bin [2^3, 2^4)
Bin [2^4, infinity)
Chunk of size 1 byte at 0x0
Chunk of size 1 byte at 0xC
Chunk of size 2 bytes at 0x1
Chunk of size 9 bytes at 0x5

Figure 2: Bins for the example memory layout.

Within each bin, the list of free chunks is sorted in order of the size of the region. This way, when a request for a memory allocation of size n is made, the correct bin is first located, and then the list is walked until a chunk with a size greater than n is found. If a chunk is not found, then the search continues into the next bin.

Once a chunk of adequate size is found, the chunk is removed from the bin. The allocated region is returned to the caller, and any leftover space is placed back into a free list in the proper bin.

When the memory is freed, the chunk is again returned to a free list, in the proper bin.

In order to start the process, the entire chunk of the heap, which is empty at the start of the allocation process, is considered to be one large chunk, which is split as needed by the first request.

## 2.3 Coalescing

The previous description has a problem. Consider the following scenario in a heap space of 16 bytes:

1. Allocate chunk of size 4 bytes.
2. 16 byte chunk is split into two chunks, of size 4 and 12. The size 12 chunk is returned to the free list. The free list now has a single 12 byte chunk.
3. Allocate chunk of size 6 bytes.
4. The 12 byte chunk is split into two chunks, where each is 6 bytes. The 6 byte chunk is returned to the free list. The free list now has a single 6 byte chunk.
5. Free 4 byte region.
6. 4 byte chunk returned to free list. The free list now has a 4 byte chunk and a 6 byte chunk.
7. Free 6 byte region.
8. 6 byte chunk returned to free list. The free list now has a 4 byte chunk and two 6 byte chunks.
9. Attempt to allocate chunk of size 14 bytes.
10. Uh oh! All of the chunks are less than size 14. Even though the total size of available memory is 16 bytes (4+6+6) and this memory is contiguous, the algorithm cannot return a chunk that is at least 14 bytes.

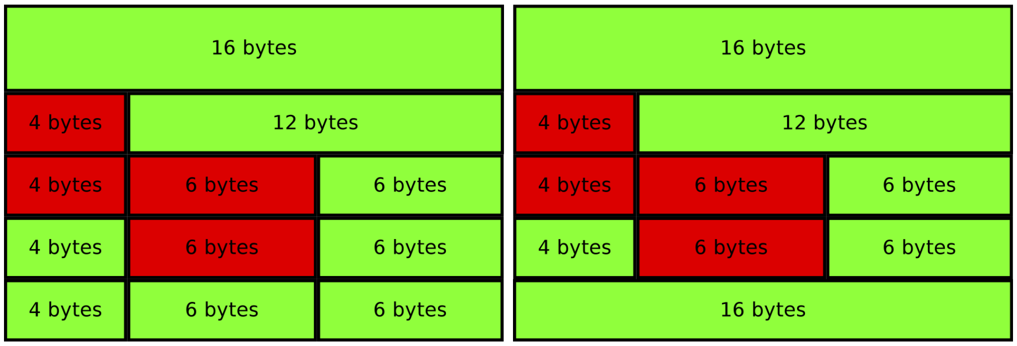

Figure 3: An example of memory allocation without (LHS) and with (RHS) coalescing.

The solution to this is coalescing. When freeing memory, rather than returning the freed region directly to the free list, the freed chunks are coalesced with neighboring chunks. That is, when a memory chunk is about to be returned to the free list, the system first checks the chunks directly before and after the chunk being freed. If either or both of the chunks are also free, then the free region is combined into one larger chunk, rather than smaller separate chunks. The neighboring free regions are removed from the free lists, and the larger region is reinserted. For example, in Figure 3, when the 6 byte chunk is freed and returned to the free list, the separate chunks are coalesced into a 16 byte chunk. This way, when a request is made for a chunk of size 14 bytes, the allocator is able to service the request.

This process is similar to the buddy system that is described in Chapter 7 of the textbook. However, this system differs in that the chunks need not be of a size that is a power of two.

### 2.4 Page Alignment

Most modern operating systems use a concept called **paging**. For the purpose of the project, your memory allocator must support simulated page alignment for returned memory addresses, if requested by the caller of the `kalloc` function.

This is accomplished by using the `PAGE_SIZE` constant, as defined in the skeleton code (in `common.h`). If page alignment is requested, then the following must be true of the returned pointer p: `p & PAGE_SIZE == p`. That is, the pointer address returned to the user must be divisible by the page size.

## 3 Implementation

### 3.1 Data Structures

#### 3.1.1 `struct sorted_array`

A sorted array, provided in `sorted_array.h` and `sorted_array.c`, is the basis of the free list. The array uses insertion sort for each insertion in order to maintain a list of items in ascending order. The comparison function used for this project, `header_less_than`, in `heap.c`, sorts by the size of the chunk. These data structures and assorted functions should not require modification.

#### 3.1.2 `struct heap`

Defined in `kheap.h`, this structure is the basic heap structure, containing the start and end addresses, as well as the max address. The difference between the end address and the max address is that the end address represents the **current** end of the heap, while the max address represents the **maximum** end of the heap. For this project, the difference is not substantial; however, within the kernel, this allows the number of pages used by the heap to grow and shrink with usage.

#### 3.1.3 `struct header` and `struct footer`

Each chunk requires a header and a footer, to maintain sanity checks, record the size of each chunk, record the allocation status of each chunk, and discover neighboring chunks. The footer refers to the header so that data, such as the allocation status, is only recorded in one location. The magic variable is stored in two locations, to ensure the integrity of both the header and footer. A magic variable is simply a relatively unique number that is used to detect corruption. If a magic number does not exist at a location at which it is expected, then it is likely that something is corrupting the memory space.

## 3.2 Functions

The required functions below are those that are necessary in order to pass the project. The recommended functions are just that: recommended not required. It is **strongly** recommended to follow the outline of the sample code, as provided with the project, for both sets of functions. However, should you choose to implement your memory allocator differently, then you need only implement the prototypes for the required functions. NOTE: if you choose this route, you may eliminate your chance for partial credit; see the grading section for more details. See the sample code for more in-depth comments about each function.

### 3.2.1 Required

- `void *kalloc_heap(size_t size, u8int page_align, struct heap *heap)`
  allocates memory on the specified heap; if `page_align` is non-zero, then page alignment is requested.
- `void kfree_heap(void *p, struct heap *heap)`
  frees memory on the specified heap
- `void *memset(void *s, int c, size_t n)`
  fill memory with a constant value

### 3.2.2 Recommended

- `void add_hole(void *start, void *end, struct heap *heap)`
  creates and writes a hole that spans [start,end)
- `void coalesce(struct header *chunk)`
  coalesces the provided chunk
- `ssize_t find_smallest_hole(size_t size, u8int page_align, struct heap *heap)`
  finds the chunk that will be used to service the allocation request

## 3.3 Other Notes

Your final deliverable should **not** rely upon the C standard library. As such, certain necessary functions will be provided as a part of the initial project. These functions are documented in comments within the source code and deal primarily with data structures such as sorted arrays as well as the test harness and testing code. However, it is possible to use the C standard library during development, to aid in debugging.

### 3.4 Suggested Implementation Order

It is suggested that you implement your code in the following order:

1. `memset`
2. `add_hole` (do not worry about coalescing yet)
3. `find_smallest_hole`
4. `kalloc_heap` (do not worry about page alignment or coalescing yet)
5. `kfree_heap`
6. page alignment in `kalloc_heap`
7. coalescing

## 4 Getting, Building, Running, Testing, and Debugging the Project

### 4.1 Checking Out the Sample Code

Sample code is provided on Piazza to get you started. You need to download the sample code from Piazza and then upload this into your remote repo.

### 4.2 Building and Running

To build, use the `make` command from within the `src` directory. The Makefile has already been configured for the project, so long as your headers end in `.h` and your C files end in `.c`.

Running a memory allocator is not very exciting. However, there is a simple test program that uses the allocator for a single allocation request. In order to evaluate your project, run the tests as described in the next section.

### 4.3 Testing

Tests are included with the project in order to evaluate your code. The following test commands are available (run all from the `src` directory):

- `make test` - run all tests
- `make test1` - run all tests for the first intermediate deliverable
- `make test2` - run all tests for the first and second intermediate deliverables

The test scripts will output information about each test while running. To hide the output from make, use -s. Example: `make -s test`.

See the tests subsection, under grading, for more information.

## 4.4 Debugging

### 4.4.1 Valgrind

Valgrind (http://valgrind.org) includes a tool called Memcheck that is able to check for invalid memory accesses. This tool is used when running `make test`, as described above, in order to show memory errors. Should you wish to run Memcheck on your main executable, use `valgrind ./main` in order to run the `main` program under Valgrind. Ensure that you have compiled your source with the `-g` option for debugging symbols (which is already done if you use the `make` command). If your test program requires a large amount of stack space, then you must use Valgrind's `--max-stackframe` option to set the maximum size of the stack. If this is necessary, then the Valgrind output will indicate a minimum value.

### 4.4.2 gdb

The GNU Debugger (gdb) is another debugging tool to aid in developing your project. In order to use gdb to test the main program, run `gdb ./main`. At the prompt, type `run`. If the program crashes, then a message will be printed indicating the error. To get a backtrace, which shows the calls that lead up to the crash, type `bt`.

## 5 Grading

The project grade will be assigned as follows.

- 75% - testing your code (at least 40% of tests are provided with initial source code)
- 15% - correctness of your algorithm
- 10% - code clarity, documentation quality (write comments! indent properly! leave blank space where appropriate!)

The tests, which form a large majority of your grade, cover scenarios such as integrity of the heap, handling of out of memory exceptions, correctness of the algorithm, and other details.

There are two kinds of test cases: required and recommended; these cases correspond roughly to black box and white box tests. If your code passes all of the required tests, then you will receive 100% for testing. If your code does not pass all of the required tests, but passes some of the recommended tests, then you will receive partial credit. Note: you are allowed to implement your system however you wish, so long as the required tests pass; however, if you wish to receive partial credit should your code fail, then it is suggested to follow the recommended function guidelines in order to receive the maximum amount of partial credit. The grader will have more complex test cases than the basic test cases given to you; your code must pass some of these more complex test cases to receive full credit.

### 5.1 Extra Credit Tests

There is a potential for up to 2% of extra credit for providing additional tests that may break your classmates' algorithms. Each additional test may earn up to 1%, depending on the complexity of the test, and there is a maximum of submitting two tests. To earn credit, each test **must** be accompanied by **detailed** comments explaining the test case as well as what is being tested, and the test must test something that is not already covered by the basic test suite.

### 5.2 Binning

An additional 3% can be earned by implementing binning, as described in the theory section. You must have at least ten bins, and you must submit at least one test (including the **detailed** description of the test) that shows your bins are working properly. If you implement binning, your `struct heap` should look exactly as follows, where `NUM_BINS` is defined in `kheap.h`:

```
struct heap
{
   struct sorted_array  free_lists[NUM_BINS];
   void    *start_address;
   void    *end_address;
   void    *max_address;
};
```

## 6 Deliverables

- Intermediate I (Mar 26th at 11:59pm): Implementation of the `memset` function.
- Intermediate II (Apr 9th at 11:59pm): Implementation of memory allocation without coalescing.
- Final (Apr 14th at 11:59pm): Final implementation due.

## 7 Submission

Submission for this project will be via Blackboard. Below is the directory structure that your submission should follow:

```
project3/
    README // See section below
    src/ // Source code
        Makefile // Provided - no modifications needed
        common.h // Provided - no modifications needed
        kheap.h // Provided - no modifications needed
        kheap.c // Complete implementation
        memset.h // Provided - no modifications needed
        memset.c // Complete implementation
```

```
sorted_array.h // Provided - no modifications needed
sorted_array.c // Provided - no modifications needed
```

You must provide a README file in the root of your project directory that contains the following:

1. Your team name
2. Each team member's name
3. Approximate number of hours spent on each phase of the project: intermediate I, intermediate II, and final submission
4. Any unusual/interesting features
5. Hardest part of the assignment
6. Any additional comments regarding the assignment