

# CS 213, Fall 1999

## Lab Assignment L3: Implementing a Dynamic Storage Allocator

Assigned: Thur., Oct. 7, Due: Wed., Oct. 20, 11:59PM

Khalil Amiri ([amiri+@cs.cmu.edu](mailto:amiri+@cs.cmu.edu)) is the lead person for this lab.

### Introduction

In this lab you will be writing a dynamic storage allocator for C programs, i.e., your own version of the `malloc` and `free` routines from the standard C library. Your task is to develop an allocator that is correct, memory efficient, and fast. All implementation decisions are up to you! You will need to be very creative to write a good allocator.

### Logistics

As usual, you may work in a group of up to 2 people.

Any clarifications and revisions to the assignment will be posted on the class bboard and WWW page.

Your programs will be evaluated by their correctness and performance on the “fish” cluster. However, you can do your code development on any machine (you may need to edit the Makefile if you change platforms).

The tarfile

```
/afs/cs.cmu.edu/class/academic/15213-f99/L3/L3.tar
```

contains the files you’ll need for this assignment. You will be turning in the file `malloc.c`, after filling in the empty functions with your implementation.

You can hand in the assignment via “`gmake handin NAME=username`”, with “`VERSION=version_num`” if necessary for hand-ins after the initial one.

## Specification

Your dynamic storage allocator consists of the following three functions, which are declared in `malloc.h` and defined in `malloc.c` with empty function bodies.

```
int    mm_init(void)
char *mm_malloc(size_t size);
void   mm_free(void *block);
```

You will fill in these empty function bodies (and possibly define other private functions) as your solution to this lab assignment. **You are not allowed to change the interfaces, nor to call any system routines that manage dynamic storage** (e.g. `malloc`, `free`, `sbrk`, etc.) You are also not allowed to declare other variables to hold the control data for your allocator; you should store these in the heap area.

Your dynamic storage allocator interacts with an arbitrary application program in the following way: As part of its initialization phase, the application calls your `mm_init` function to perform initialization of the heap. You must allocate the necessary initial heap area and initialize all structures you need. The application then makes a series of calls to `mm_malloc` and `mm_free`.

You are allowed to use the following functions from `memlib.c` in your allocator:

- `mem_sbrk`: You use this function to expand the heap area. The lower and upper boundaries of the heap area are contained in `dseg_lo` and `dseg_hi` respectively. You are allowed to read these variables, but you should not modify them in any way. You must call `mem_sbrk` in order to change the upper bound. This function accepts a positive integer argument, which is the amount of bytes by which the upper bound should be expanded. The return value is the beginning of the newly allocated heap area, or `NULL` if there wasn't any memory left. The interface of `mem_sbrk` is very similar to that of the `sbrk` system call, but you should use `mem_sbrk`. You cannot decrease the heap area in size, only increase it, so be careful how you call `mem_sbrk`. In effect, each time you call `mem_sbrk`, the value of `dseg_hi` is incremented by the amount you request, but the actual memory allocated is always in multiples of the system page size, so it might be a good idea to call `mem_sbrk` with an increment that is a multiple of the page size. You can call `mem_pagesize()` to find out the system page size.
- `mem_usage`: This is simply a shorthand that returns the current size of the heap in bytes.

The functions you need to implement are the following:

- `mm_init`: Before calling `mm_malloc` or `mm_free`, the application program calls `mm_init` to perform any necessary initializations, including the allocation of the initial heap area. The return value should be `-1` if there was a problem in performing the initialization, `0` otherwise. We will grade your implementation in several phases. To facilitate our testing, write `mm_init` such that it reinitializes *all* state when it is called. We will use it to reinitialize your dynamic storage allocator between each test phase.
- `mm_malloc`: The `mm_malloc` routine returns a pointer to an allocated region of at least `size` bytes. The pointer must be aligned to 8 bytes, and the entire allocated region should lie within the memory region from `dseg_lo` to `dseg_hi`.
- `mm_free`: The `mm_free` routine is only guaranteed to work when it is passed pointers to allocated blocks that were returned by previous calls to `mm_malloc`. The `mm_free` routine should add the block to the pool of unallocated blocks, making the memory available to future `mm_malloc` calls.

## Test driver

The file `driver.c` contains the actual driver program we will use to test your allocator. Feel free to use any other testing method you wish while developing your code. The test driver should provide you with some useful information for debugging your program. The command line options it accepts are as follows:

- `-f tracefile ...` Use a particular tracefile for testing; can repeat this option to load multiple tracefiles.  
If no tracefiles are specified, the default set of tracefiles is used.
- `-v` Verbose mode; print out some detailed debugging info (default).
- `-q` Quiet mode.
- `-h` Print a help (usage) message.
- `-c` Run the system (`libc`) malloc in addition to yours and report the throughput stats for both.
- `-C` Run the system malloc by itself and report its throughput stats.
- `-d` Generate a dump of your allocator's internal operation ( e.g. the pointers it returns and the `mem_sbrk` calls it initiates) into a text file.
- `-t tolerance` Specify an error tolerance for the time measurements (default: 0.05)

Please avoid excessive use of the `-c` and `-C` flags especially when a lot of users are sharing the same machine for development. These options are implemented to provide you with a basis for comparison and should be avoided under high load. Because they cause the program to consume a significant amount of memory, their excessive use can make machine sharing inconvenient when the number of concurrent users is high.

## Grading Criteria

Your dynamic storage allocator will be evaluated in four areas: correctness, memory efficiency, running time, and style. There are a total of 60 points.

### Correctness (20 points)

To be correct, your `mm_malloc` routine must return `NULL` if it cannot find a sufficiently large free block. Otherwise, it must return a pointer, aligned to 8 bytes, to an allocated block of at least the requested size (the block might be larger because of alignment constraints or placement policies in your allocator). The block must be located within the allocated heap (between `dseg_lo` and `dseg_hi`), and no part of the block may be returned by subsequent calls to `mm_malloc` until it has been released by a call to `mm_free`.

The correctness criteria are all or nothing. If your implementation is correct by this definition, you will receive all 20 points, otherwise you will receive 0 points.

### Performance (35 points)

There are two main aspects to the performance of the memory allocator: space utilization and running time. Your implementation will be evaluated both on space utilization as well as throughput (operations completed per unit time). A number of traces will be used to test your allocator; some are artificially generated for the purpose of testing the behaviour of your code in various situations and others have been obtained from real-world applications. We are providing you with all the traces that we will use to evaluate your allocator. These can be found at `/afs/cs/academic/class/15213-f99/L3/traces/`.

We will use two performance metrics to evaluate your allocator. The first metric is *space utilization* and the second is *throughput*.

Space utilization is defined as the aggregate amount of memory requested by the driver (via `mm_malloc`) and not yet freed (via `mm_free`) to the size of the heap used by your allocator (`dseg_hi-dseg_lo`). Space utilization fluctuates during the execution of the tests as the heap is expanded and as memory is allocated and freed. The performance metric we will use for space utilization is the peak utilization,  $U$ , defined as the maximum amount of memory allocated (and not yet freed) by the application at any point in time during its execution to the final heap size.

The optimal space utilization is, of course,  $U_{opt} = 1$ , which corresponds to 0% space lost to overhead and fragmentation. Although  $U_{opt}$  is unachievable, you should come quite close to it. There are several factors that influence space utilization, the most important of which is the allocation policy. Your allocation policy should minimize fragmentation. One way to ensure that fragmentation does not get out of hand is to coalesce adjacent free blocks. You can either do immediate coalescing in `mm_free`, or do lazy coalescing - as long as `mm_malloc` never fails when enough memory is available in consecutive free blocks. Space utilization is also influenced by the amount of overhead, the space used by your allocator for its internal housekeeping.

The second metric is throughput,  $T$ , defined as the total number of operations per second. To do well on this metric, you have to expend a minimal number of instructions when allocating and freeing memory in the common case. The memory manager is a critical part of the runtime system. Therefore, it is very important to optimize it in every way possible. We will be measuring the speed of your implementation using a number of different workloads. You should think about how to write the code in such a way as to minimize the number of instructions required for the common case. When designing your implementation, try to make choices that simplify the code, e.g. that result in fewer instructions, need fewer conditionals, etc.

The performance of your allocator is summarized by a performance index,  $P$ , which is a linear sum of the utilization and throughput metrics. The index is slightly biased towards the first (default  $w = .65$ ):

$$P = w \frac{U}{U_{opt}} + (1 - w) \text{Min}(1, \frac{T}{T_{libc}}) = wU + (1 - w) \text{Min}(1, \frac{T}{T_{libc}})$$

The first part of the performance index,  $w \cdot U$ , is the contribution of the space utilization metric. At best, this term is equal to  $w$ , which occurs when your space utilization  $U$  is 1.

The second part of the index,  $(1 - w) \cdot \text{Min}(1, \frac{T}{T_{libc}})$  is contributed by the throughput metric.  $T_{libc}$  is the throughput of `libc` malloc. At best, when your throughput equals or exceeds that of `libc`,  $T_{libc}$ , this part is equal to  $1 - w$ . Thus, ideally the performance index is  $P = w + (1 - w) = 1$  or 100%.

The reason for the *Min* in the second term is to make sure that a very fast solution that exceeds `libc`'s throughput while doing poorly on space utilization does not get a high performance index. The motivation is that space and time are both expensive resources and your solution should be balanced in optimizing for both. Because the summary performance index depends both on the space and time performance, **you should not optimize speed at the expense of memory overhead, or vice-versa**. One of the challenging aspects of this assignment will be to achieve a proper balance between those two.

The driver program reports the performance index,  $P$ , of your allocator as a percentage.

Over the entire set of default traces, your throughput should approximate  $T_{libc}$ . The `libc` allocator incurs substantially more overhead in the operating system than your allocator. Your allocator uses simulated `sbrk` calls which are much faster than the real calls invoked by `libc` malloc. Moreover, the `libc` allocator is allocating the memory used by the driver program. Therefore, a smart implementation should be able to

come close to  $T_{libc}$  and even supercede it.

The final grading will be done on a curve, after we have reviewed the performance results from all your implementations. You can see how good your implementation is by checking the statistics web page. Note that if you fail the correctness tests, you will not get any points for performance.

### Style (5 points)

Your code should be readable and commented. Define macros or subroutines as necessary to make the code more understandable. Keep in mind that when your code gets more and more complicated, your performance is likely to suffer. Smart design decisions and optimizations will tend to make your code smaller.

### Automated Testing/Grading System

Feel free to modify `driver.c` to do your own testing and debugging, but it would be a good idea if you used `driver.c` located in `/afs/cs/academic/class/15213-f99/L3/src/` for your final tests.

You can evaluate the performance of your allocator by running the tests locally. Note that due to variability in load on the cluster machines, the performance reported by the local driver may be inaccurate (usually slightly slower) than what it would be when your solution is graded.

We will be using a Web-based automated testing and grading system. You can submit your `malloc.c` to this testing and grading system at any time, and as many times as you wish, by doing a “`gmake update NAME=username`”. Your code will be tested for the above criteria, and the results will be posted to a web page every few minutes. This will allow you to check your implementation for correctness and to gauge the performance of your implementation against those of other groups.

You can hand in your assignment by doing “`gmake handin user=USERNAME`”, with `version=VERSION`” if necessary for hand-ins after the initial one.

### Hints

- Debugging: Dynamic memory allocators are notoriously tricky beasts to program correctly and efficiently. They are difficult to program correctly because they involve a lot of `(void *)` pointer references. It may be helpful to write functions that print the state of your allocator’s data structures which you can use when debugging your program.
- Dumps: For debugging purposes, you may find the `-d` option helpful. Feel free to modify the dump routines to report more information if you like.
- Traces: During initial development, using shorter traces may simplify debugging and testing. We have placed two short trace files in `L3/traces/short/short{1,2}.rep` that you can use by invoking the `-f` option of the driver.
- Performance: When optimizing performance, you may find the `gprof` tool helpful. This tool produces an execution profile of your program. It calculates the amount of time spent in each routine. To use `gprof`, you will need to turn on the `gprof` flags when compiling your program:

```
bass>gmake clean
bass>gmake GPROF=-pg
bass>gmake GPROF=-pg
```

When you run your executable, say `malloc`, a file named `gmon.out` is created in your current working current directory. To view the profile information in this file, you can invoke `gprof` as follows:

```
bass>gprof malloc gmon.out
```