

CS 213, Spring 1999

Lab Assignment L3: Implementing a Dynamic Storage Allocator

Assigned: Feb. 25, Due: Wed., Mar. 10, 11:59PM

Spiros Papadimitriou (spapadim+ta@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 class Alphas. 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-s99/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.

Details

Implementation

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 only 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 your 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.
- *mem_pagesize*: This returns the system's page size in bytes. It may not be necessary to use it, but it might help you in fine-tuning your performance.
- *mem_usage*: This is simply a shorthand that returns the current size of your 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. You should not change this, but 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 particular tracefile for testing; can repeat this option to load multiple tracefiles. If no tracefiles are specified, a list of default tracefiles will be used.
- v Verbose mode; prints out some detailed debugging info (default).
- q Quiet mode.
- t*tolerance* Specify an error tolerance for the time measurements (default: 0.02)

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: memory efficiency and running time. Your implementation will be evaluated both on memory utilization as well as speed. 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 will provide you with some of the traces that we will use for evaluating your allocator. You will find these in `/afs/cs/academic/class/15213-s99/L3/traces/`; please check for possible updates in this directory.

There are several factors that influence memory efficiency. One of the most important is the allocation policy. You should try to pick an allocation policy that minimizes fragmentation. An important step in ensuring that fragmentation does not get out of hand is implementing coalescing of free blocks. You may 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. Finally, memory efficiency is influenced by the amount of overhead.

The memory manager is such a critical part of a runtime system that it is very important to optimize 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.

You should try to do well both in terms of speed and memory efficiency. **Do not optimize speed at the expense of memory overhead, or vice-versa.** One of the most challenging aspects of this assignments will be to achieve a proper balance between those two. Your grade will depend both upon space as well as time overhead of your implementation. We will measure both over a set of traces; some of these traces will be provided to you in order to help you test your implementation. These measurements will be averaged using a set of weights which will reflect the relative significance of each trace; we will determine these weights shortly.

Both space and time overheads will be measured as the difference between the space and time your solution uses versus an optimal estimate. For space, the optimal estimate will be the maximum of total memory allocated at any one instant. No allocator can achieve this bound (at least without foreknowledge of the future, which in reality is impossible), but you should be able to come quite close. For time, the optimal estimate will be an ideal operation that is performed “instantaneously” (ie. a blank function). The performance index will depend on both overheads, so you should try to minimize both. The formula for estimating overall space overhead will be of the following form:

$$\frac{w_1}{o_1} + \frac{w_2}{o_2} + \dots + \frac{w_n}{o_n}$$

where o_i is your space overhead for trace i and w_i is the respective weight (contained in the tracefile). The formula for time overhead is similar. The overall performance index will be a weighted sum of the two overheads.

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-s99/L3/src/` for your final tests.

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; you should definitely do this. 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

We will have an automated testing and grading system. You can submit your `malloc.c` file in a directory by doing “`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.

Hints

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. They are difficult to program efficiently because running time is influenced by a number of factors, including the degree of fragmentation, the behavior of the application, the placement policy of the allocator, and the low-level mechanisms that implement the placement policy.

To help you better understand the behavior of your program, you might find a program called *Atom* to be helpful. Atom provides a simple but extremely powerful and general mechanism for building tools that navigate and instrument executable Alpha object files. You can write your own Atom tools from scratch (not recommended for this Lab), or you can use a wide variety of existing Atom tools. Examples include:

- *gprof*: procedure-level execution time profiling.
- *iprof*: procedure-level instruction profiling.
- *liprof*: basic-block level instruction profiling.
- *syscall*: system call performance summary.
- *3rd*: the *3rd Degree* memory checker and leak finder (similar to Purify).
- *pixie*: basic block profiling (like the `pixie(1)` command).

See `/afs/cs.cmu.edu/class/academic/15213-s99/L3/atom` for documentation and an example of an Atom tool (*ptrace*), which produces an instrumented version of “hello, world” (`hello.ptrace`) that traces every procedure call.