Assignment 1: Malloc

Implementation of a dynamic memory allocator

On this page

Outline

- **Due date:** 19 September 2022, 23:59
- **Mark weighting:** 25%
- **Submission:** Submit your assignment through GitLab (full instructions below)
- **Policies:** For late policies, plagiarism policies, etc., see the policies page

Deadline passed
0 :00:00:00
Days Hrs Mins Secs

> warn
>
> This assignment builds upon the following labs:
>
> - Lab 5: Building Dynamic Memory Allocators
> - Lab 6: Sanity Checking Implicit Free List Implementation

If you have not completed the tasks in the above labs or do not understand the content, we *strongly* recommend that you first complete the labs and then start the assignment.

Introduction

Managing memory is a major part of programming in C. You have used `malloc()` and `free()` in the recent labs. You have also built a very basic memory allocator, and it is now time to build a more advanced allocator. In this assignment, you will implement a memory allocator, which allows users to malloc() and free() memory as needed. Your allocator will request large chunks of memory from the OS and efficiently manage all the bookkeeping

and memory. The allocator we ask you to implement is inspired by the DLMalloc allocator designed by Doug Lea. The DLMalloc allocator also inspired the PTMalloc allocator, which

GLibC currently uses. Indeed, our allocator is a simplified version of DLMalloc, but you will also notice many similarities.

Background

We hope that the last two labs have motivated the need for dynamic memory allocators. Specifically, we have seen that while it is certainly possible to use the low-level `mmap` and `munmap` functions to manage areas of virtual memory, programmers need the convenience and efficiency of more fine-grained memory allocators. If we managed the memory from the OS ourselves, we could allow allocating and freeing variables in any order, and also reuse memory for other variables.

The last lab taught you how best to build an implicit free list allocator for managing free blocks. In this assignment, we will first build a more efficient free list data structure called an explicit free list, and then perform a number of optimizations.

Explicit free list

The block allocation time with an implicit free list is linear in the total number of heap blocks which is not suitable for a high-performance allocator. We can add a *next* and *previous* pointer to each block's metadata so that we can iterate over the unallocated blocks. The resulting linked list data structure is called an explicit free list. Using a doubly linked list instead of a free list reduces the first-fit allocation time from linear in the total number of blocks to linear in the total number of free blocks.

Dealing with memory fragmentation

Fragmentation occurs when otherwise unused memory is not available to satisfy allocate requests. This phenomenon happens because when we split up large blocks into smaller ones to fulfill user requests for memory, we end up with many small blocks. However, some of those blocks may be able to be merged back into a larger block. To address this issue requires us to iterate over the free list and make an effort to find if the block we are trying to free is adjacent to another already free block. If neighboring blocks are free, we can coalesce them into a single larger block.

Dealing with the edges of chunks

One detail we must consider is how to handle the edges of the chunks from the OS. If we simply start the first allocable block at the beginning of the memory chunk, then we may run into problems when trying to free the block later. This is because a block at the edge of the chunk is missing a neighbor. A simple solution to this is to insert a pair of fenceposts at either end of the chunk. The fencepost is a dummy header containing no allocable memory,

but which serves as a neighbor to the first and last allocable blocks in the chunk. Now we can look up the neighbors of those blocks and don't have to worry about accidentally

coalescing outside of the memory chunk allocated by the OS, because anytime one of the neighbors is a fencepost we cannot coalesce in that direction.

Optimizations

We will also perform the following optimizations as part of the assignment to improve the space and time complexity of our memory allocator.

Reducing the Metadata Footprint

- **Naive solution:** In our description of the explicit free list above, we assume the memory allocated to the user begins after all of the block's metadata. We must maintain the metadata like size and allocation status because we need it in the block's header when we free the object.
- **Optimization 1:** While we need to maintain the size and allocation status, we only use the free list pointers when the object is free. If the object has been allocated, it is no longer in a free list; thus, the memory used to store the pointers can be used for other purposes. By placing the next and previous pointers at the end of the metadata, we can save an additional 2 * sizeof(pointer) bytes and add that to the memory allocated to the user.
- **Optimization 2:** The allocated flag that tells if a block is allocated or not uses only one bit. Since the sizes are rounded up to the next 8 bytes, the last three bits are not used. Instead of using a boolean to store the allocated flag, we can use one of the unused bits in size. That will save an additional 8 bytes.

Constant Time Coalesce

- **Naive solution:** We mentioned above that we could iterate over the free list to find blocks that are next to each other, but unfortunately, that makes the free operation O(n), where n is the number of blocks in the list.
- **Optimized solution:** The solution we will use is to add another data structure called Boundary Tags, which allows us to calculate the location of the right and left blocks in memory. To calculate the location of the block to the right, all we need to know is the size of the current block. To calculate the location of the block to the left, we must also maintain the size of the block to the left in each block's metadata. Now we can find the neighboring blocks in O(1) time instead of O(n).

Multiple Free Lists

- **Naive solution:** So far, we have assumed a single free list containing all free blocks. To find a block large enough to satisfy a request, we must iterate over all the blocks to find a block large enough to fulfill the request.

- **Optimized solution:** We can use multiple free lists. We create `n` free lists, one for each allocation size (8, 16, ..., 8*(n-1), 8*n bytes.) That way, when a user requests

  memory, we can jump directly to the list representing blocks that are the correct size instead of looking through a general list. If that list is empty, the next non-empty list will contain the block best fitting the allocation request. However, we only have `n` lists, so if the user requests `8*n` bytes of memory or more, we fall back to the naive approach and scan the final list for blocks that can satisfy the request. This optimization cannot guarantee an O(1) allocation time for all allocations. Still, for any allocation under 8*n, the allocation time is O(number of free lists) as opposed to O(length of the free list).

Getting additional chunks from the OS

The allocator may be unable to find a fit for the requested block. If the free blocks are already maximally coalesced, then the allocator asks the kernel for additional heap memory by calling `mmap`. The allocator transforms the additional memory into one large free block, inserts the block in the free list, and then places the requested block in this new free block.

Placement policies

As we know already, when an application requests a block of `k` bytes, the allocator searches the free list for a free block that is large enough to hold the requested block. Placement policy dictates the manner in which the allocator performs this search. There are three popular policies.

- **First fit:** Search the free list from the beginning and choose the first free block that fits.
- **Next fit:** Similar to first fit, but start each search where the previous one left off.
- **Best fit:** Examine every free block and choose the free block with the smallest size that fits.

> info
>
> You only need to implement first fit unless you are attempting *one of the two optional tasks* in the HD category.

Lab Specification

Malloc spec

You can read the malloc interface on the malloc man page. Many details are left up to the

library's authors. For instance, consider the many optimizations we mention above. All versions of malloc would be correct by the specification on the man page, but some are more efficient than others.

Our implementation spec

We have described the basic implementation we want you to follow with optimizations in the background and optimization sections above. We now provide the technical specification of the required design. Some of the requirements are in place to enforce conformance to the design, and others guarantee determinism between our reference allocator and your allocator for testing. The specification below should contain all the details necessary to ensure your implementation is consistent with the reference implementation.

Data structures and constants

We provide certain constants namely:

1. `ARENA_SIZE` : We always get a constant chunk of 4 MB from the OS for allocation. For objects larger than 4 MB, you may have to use a multiple of 4 MB.
2. `kAlignment` : We require word-aligned addresses from our allocations.
3. `kMinAllocationSize` : We set the minimum allocation size for our allocator to be 1 word.
4. `kMaxAllocationSize` : We set the maximum allocation size for our allocator to be 16 MB - size of your meta-data. Note you will have to define this constant yourself in your `mymalloc.c` file (like from labs).

Allocation

- An allocation of 0 bytes should return the NULL pointer for determinism.
- All chunks requested from the OS should be of size ARENA_SIZE defined in mymalloc.h.
- All requests from the user are rounded up to the nearest multiple of 8 bytes.
- The minimum request size is the size of the full header struct. Even though the pointer fields at the end of the header are not used when the block is allocated, they are necessary when the block is free, and if space is not reserved for them, it could lead to memory corruption when freeing the block.
- When allocating from the final free list (N_LISTS - 1), the blocks are allocated in first-fit order: you will iterate the list and look for the first block large enough to satisfy the request size. Given that all other lists are multiples of 8, and all blocks in each list are the same size, this is not an issue with the other lists.
- When allocating a block, there are a few cases to consider:
    - If the block is exactly the request size, the block is simply removed from the free

list.

    - If the block is larger than the request size, but the remainder is too small to be allocated on its own, the extra memory is included in the memory allocated to the user and the full block is still allocated just as if it had been exactly the right size.
    - If the block is larger than the request size and the remainder is large enough to be allocated on its own, the block is split into two smaller blocks. We could allocate either of the blocks to the user, but for determinism, the user is allocated the block which is higher in memory (the rightmost block).
    - When splitting a block, if the size of the remaining block is no longer appropriate for the current list, the remainder block should be removed and inserted into the appropriate free list.
- When no available block can satisfy the user's request, we must request another chunk of memory from the OS and retry the allocation. On initialization of the library, the allocator obtains a chunk from the OS and inserts it into the free list.
- In operating systems, you can never expect a call to the OS to work all the time. If allocating a new chunk from the OS fails, your code should return the NULL pointer, and `errno` should be set appropriately (check the man page).
- The allocator should allocate new chunks lazily. Specifically, the allocator requests more memory only when servicing a request that cannot be satisfied by any available free blocks.

Deallocation

- Freeing a NULL pointer is a no-op (don't do anything).
- When freeing a block, you need to consider a few cases:
    - Neither the right nor the left blocks are unallocated. In this case, simply insert the block into the appropriate free list
    - Only the right block is unallocated. Then coalesce the current and right blocks together. The newly coalesced block should remain where the right block was in the free list
    - Only the left block is unallocated. Then coalesce the current and left blocks, and the newly coalesced block should remain where the left block was in the free list.
    - Both the right and left blocks are unallocated, and we must coalesce with both neighbors. In this case, the coalesced block should remain where the left block (lower in memory) was in the free list.
- When coalescing a block, if the size of the coalesced block is no longer appropriate for the current list, the newly formed block should be removed and inserted into the appropriate free list. (Note: This applies even to cases above where it is mentioned to leave the block where it was in the free list.)

Tasks

Your task is to implement `malloc` (memory allocator) and include in your implementation the various requirements and optimizations discussed above. Broadly, your coding tasks are three-fold.

Allocation

1. Calculate the required block size.
2. Find the appropriate free list to look for a block to allocate.
3. Depending on the size of the block, either allocate the full block or split the block and allocate the right (higher in memory) portion to the user.
4. When allocating a block, update its allocation status.
5. Finally, return the user a pointer to the data field of the header.

Deallocation (Freeing)

1. Free is called on the same pointer that `malloc` returned, which means we must calculate the location of the header by pointer arithmetic.
2. Once we have the block's header freed, we must calculate the locations of its right and left neighbors, using pointer arithmetic and the block's size fields.
3. Based on the allocation status of the neighboring blocks, we must either insert the block or coalesce it with one or both of the neighboring blocks.

Managing additional chunks

Handle the case where the user's request cannot be fulfilled by any of the available blocks.

> warn
>
> Note that the tests we provide will succeed even if you submit an `mmap` or an implicit free list allocator. The success of these provided tests on a non-explicit free list allocator does not mean you are done. Do not submit code files with allocators from a previous lab. We have tests to ensure compliance with the assignment specification.

Report

You must submit a report (maximum of two pages) along with your malloc implementation. The report consists of the following sections.

- Describe your implementation of explicit free list, fence posts, and constant time coalescing. Briefly mention key data structures and function names.

- Describe the optimizations you have attempted in your implementation of malloc.

- If you have done quantitatively analyzed the placement policies, include any graphs and tables.
- Discuss two implementation challenges you encountered in your implementation of malloc.
- Discuss two key observations from testing and benchmarking your malloc implementation. Did something break? Did you end up fixing some stuff after testing and benchmarking? What did not work?

Marking

The code is worth 60% of your grade (in your specific category). The report is worth 40% of the grade.

Coding and Implementation

Fork the Assignment 1 repo and then clone it locally.

mymalloc.h

This file contains the type signatures of `my_malloc` and `my_free` and some pre-defined constants. Do ***not*** change this file.

mymalloc.c

This file will contain your implementation of the `my_malloc` and `my_free` functions. We only provide some constants to help with your implementation. Your task will be to implement an explicit free-list allocator. We recommend using a modular approach with judicious use of helper functions as well as explanatory comments. You can insert logging calls with the `LOG()` macro we provide. Its use is the same as `printf` except it will print the logs to `stderr` and will not print logs unless you build with logging enabled.

> warn
>
> We recommended using `exit(1)` instead of `abort()` if you want to stop the execution of the program. We advise to do so because some tests will compare the output directly and using `abort()` may change the output.

test.py

Script for testing your implementation.

```
./test.py -h
usage: test.py [-h] [-t TEST] [--release] [--log] [-m MALLOC]

options:
  -h, --help            show this help message and exit
  -t TEST, --test TEST  test name to run
  --release             build in release mode
  --log                 build with logging
  -m MALLOC, --malloc MALLOC
                        allocator name, default to "mymalloc"
```

The most important option is `-t <TEST>` which allows you to test your implementation with a single test.

tests/

Directory with test source files and built executables.

bench.py

Script for benchmarking your implementation. The script uses a simple benchmark from the glibc library which stresses your implementation.

```
usage: bench.py [-h] [-m MALLOC] [-i INVOCATIONS]

options:
  -h, --help            show this help message and exit
  -m MALLOC, --malloc MALLOC
                        allocator name, default to "mymalloc"
  -i INVOCATIONS, --invocations INVOCATIONS
                        number of invocations of the benchmark
```

The default number of invocations for the benchmark is 10. If you want to perform quick benchmark runs, then you can change the number of invocations to 3 using `-i 3`. It is recommended to use at least 10 invocations if you are reporting results, however.

bench/

Directory with benchmark source files and built executables.

Testing

You can test your implementation (assuming you have implemented your allocator in the “mymalloc.c” file) by simply running:

```
./test.py
```

The above command will clean previous outputs, compile your implementation, and then run all the provided tests against your implementation. If you want to run a single test (such as `align`) then you can run the test script like so:

```
./test.py -t align
```

If you have another implementation in a different file named “different_malloc.c” (for example), then you can run tests using this implementation with:

```
./test.py -m different_malloc
```

This may be useful if you want to test a different implementation strategy or want to benchmark two different implementations (Note: `bench.py` has this same flag).

If you have inserted logging calls using `LOG()`, then you can compile and run tests with logging enabled like so:

```
./test.py --log
```

Note that some tests which compare output may fail if logging is enabled!

If you want to run a single test directly (for example `align`) then you can run it like so:

```
./tests/align
```

If you have logging enabled and want to save the log for a particular test (for example `align`) to a file then you can run the following:

```
./tests/align &> align.log
```

Make sure you don’t accidentally add the log file to your git repo as these can get quite large in size!

You will almost certainly require using `gdb` at some point to debug your implementation and failing tests. You can run `gdb` directly on a test (for example `align`) like so:

```
gdb ./tests/align
```

By default the tests and your library are built with debug symbols enabled so you don’t have to fiddle with enabling debug symbols to aid your `gdb` debugging.

Benchmarking

Benchmarking

If you want to benchmark your code, then you will have to install some python libraries, namely `numpy` and `scipy`. This can be achieved by using `pip3`, python's package manager:

```
pip3 install numpy scipy
```

This will install the two libraries to your local user (as opposed to system-wide). This is the recommended method for installing per-user packages for python.

Benchmarking your code works in a similar way as testing:

```
./bench.py
```

This will run the provided benchmark (`glibc-malloc-bench-simple`) from glibc 10 times and provide you the average. If you are benchmarking, it is ***highly*** recommended to close all other intensive applications on your machine as you may get random interference otherwise. If you want to report your benchmark numbers, it is important to note what CPU and memory speed you were using in your report.

Just like the test script, you can switch the malloc implementation using the `-m <MALLOC>` flag. This is useful as you may want to have two different implementations that you want to compare performance on.

Submitting your work

Submit your work through `Gitlab` by pushing changes to your fork of the assignment repository. A marker account should automatically have been added to your fork of the Assignment 1 repo (if it isn't there under "Members" then let one of your tutors know).

We recommend maintaining good `git` hygiene by having descriptive commit messages and committing and pushing your work regularly. We will not accept late submissions.

Submission checklist

- The code with your implementation of malloc in `mymalloc.c`.
- The `report.pdf` is in the top-level directory.
- (Optional) Any optional tests and benchmrks you want us to look at.

Grading

The following description of grading categories assumes you submit both the code for your malloc implementation and the report.

malloc implementation and the report.

P

You will be rewarded a maximum grade of `P` if you complete the following tasks.

- Implement a single explicit free list
- Linear time coalescing
- Fence posts

CR

You will be rewarded a maximum grade of `CR` if you complete the following tasks.

- All tasks in the `P` category
- Metadata reduction
- Constant time coalescing with boundary tags

D

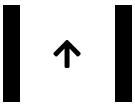
You will be rewarded a maximum grade of `D` if you complete the following tasks.

- All tasks in the `P` and `CR` categories
- Multiple free lists

HD

You will be rewarded a maximum grade of `HD` if you complete the following tasks.

- All tasks in the `P` and `CR` and `D` categories
- Requesting additional chunks from the OS
- Quantitative analysis measuring and reporting external fragmentation with three different placement policies. We leave it to you to compute the degree of fragmentation. **If you do quantitative analysis, please use the single explicit free list.**



Acknowledgement of Country

The Australian National University acknowledges, celebrates and pays our respects to the Ngunnawal and Ngambri people of the Canberra region and to all First Nations Australians on whose traditional lands we meet and work, and whose cultures are among the oldest continuing cultures in human history.

Updated: 06 Oct 2022

Responsible Officer: School Director

Page Contact: COMP2310 Course Convener