CSE 220: Systems Programming

Programming Assignment 4: Dynamic Allocator

Introduction

In this assignment, you will implement a dynamic memory allocator suitable for replacing `malloc()` for heap memory in a Unix process. You will learn about dynamic memory management, pointer manipulation, and pointer casting.

The type of allocator you will implement is a *pool allocator*, using multiple memory pools. Each memory pool consists of allocations of a fixed size. Allocations are served out of the smallest pool that supports the user's request, and allocations too large for any memory pool are requested from the OS directly. The memory pools in this project will contain allocations ranging from 32 bytes to 4096 bytes.

Your allocator is designed in such a way that you will be able to use it to run standard Unix binaries! Because you will not be implementing support for concurrency, some programs may fail through no fault of your own; however, many programs will operate correctly. You may find it interesting and enjoyable to use your allocator implementation to learn about the memory accesses of standard applications.

This document is long, but there is a reason for that. **Please read it carefully and in its entirety before starting your assignment!** There are many subtle requirements in this document that will require *critical thinking about program structure*; for example, the relationship between allocation sizes and the free list table. **Plan to sit down and draw out your data structures and implementation when you begin.** You will not "save time" by skipping this step!

1 Getting Started

You should have received a GitHub Classroom invitation for this assignment. Follow it, then check out your repository.

Chapter 9, Section 9.9 of *Computer Systems: A Programmer's Perspective* describes a dynamic allocator implementation in some detail. You may want to read this section before starting your project, and will certainly want to read it carefully while implementing your project. Note, however, that it is *not* the same allocator that you are required to implement for this project, and that it has some questionable stylistic decisions!

Chapter 8, Section 8.7 of *The C Programming Language* also describes a dynamic allocator of similar structure to that described in CS:APP. This allocator is described in somewhat less detail, but uses techniques (such as declaring structures to hold metadata, rather than using macros and pointer math) that are to be preferred over the approach in CS:APP, so you should read and try to understand it, as well. Again, you may want to read this section before you begin, and will certainly want to read it carefully during implementation. Note again that it is not the same allocator as required for this project, and that in particular you *should not* use any `union` data types in your implementation.

You will absolutely need to read `man 3 malloc`. It is not long. We will not answer questions that are clearly answered in `man 3 malloc`.

Some parts of this handout may not make much sense until you have read the above readings.

2 Pool Allocators

Pool allocators are designed to allow applications to efficiently allocate, free, and reallocate chunks of memory in sizes that are used frequently. This is an effective strategy for many applications because it is common for an application to allocate many objects that are the same size — for example, nodes in a linked list or tree, or buffers for communication. Placing such frequently-used objects in a pool where they can be rapidly reallocated improves application performance.

Allocation pools are maintained by requesting large-ish blocks of memory from the operating system, then breaking those blocks up into the allocation size stored in a pool. These "free" blocks are then used to serve allocations requested by the user. When blocks are freed by the user, they are simply placed back in the pool, preventing future allocations from having to request more memory from the operating system. When an allocation is requested from a pool that does not contain any free blocks, the allocator will request another large chunk from the operating system to be broken up again.

You will be implementing a *multi-pool allocator*, which will maintain pools containing allocation blocks with sizes of powers of two between 2^5 (32) and 2^{12} (4096). These blocks will be used to serve allocations of 24 $(32-8)$ to 4088 $(4096-8)$ bytes, using the remaining 8 bytes between these sizes and the power of two to store metadata. Because the `free()` function does not accept a size argument, this metadata will be used to store the size of the block for use by `free()`. Each allocation request will be served from the *smallest block that will store the allocation*; for example, a 1 byte allocation would come out of a 32 byte block, while a 1000 byte allocation would come out of a 1024 byte block.

3 Requirements

You must implement the following functions in the file `src/mm.c`:

- `void *malloc(size_t size)`
- `void *calloc(size_t nmemb, size_t size)`
- `void *realloc(void *ptr, size_t size)`
- `void free(void *ptr)`

Your implementation must satisfy the standard definition of these functions (see `man 3 malloc`), with the exception that it *need not allow for concurrent access from multiple threads*, as well as the additional implementation requirements listed here.

Your implementation `must` be a multi-pool allocator as described in this document. No other type of allocator implementation will receive credit. It must use pool allocation as described here to implement all allocations between 1 and 4088 bytes, and a simple bulk allocator (described below) for allocations larger than 4088 bytes. (The discrepancy between 4088 bytes and the even-power-of-two 4096 bytes, mentioned in Section 2, will be further explained later.)

All memory returned by your allocation functions (`malloc()`, `calloc()`, and `realloc()`) must be *8 byte aligned*. Failure to maintain this requirement will not result in functional failures of your allocator on the x86-64 platform (programs using your allocator would continue to work correctly), but may result in degraded performance for programs using your allocator. On other platforms, failure to maintain this requirement could result in program crashes.

Your implementation must be able to resize an allocation using `realloc`. Under certain circumstances it should do this without performing any new allocation, and under other circumstances it should do this by allocating a new block and copying the user's data from the old block to the new. If it is possible to perform the reallocation without allocating a new block and moving data, your implementation *must do so*. This is always possible for reallocations which *reduce* the size of an allocation, as `realloc()` **must** return the existing block, and the user will simply use less of it. For allocations which *increase* the size of the allocation, it is possible only if the block originally selected to serve the user's allocation is *also* large enough to serve the new allocation. For example, if the user calls `malloc(32)` to request a 32 byte allocation, your allocator will select a 64 byte block containing 56 usable bytes of memory. If the user later wishes to `realloc()` this block to 48 bytes, this does not require any allocation or copying, and your allocator **must** return the same block. This is because while 48 bytes is more than the original 32 bytes requested, it is less than the 56 bytes available in this block.

If `realloc()` cannot resize a block without performing allocation, it **must** use your allocator to acquire a new block of the appropriate size (which is necessarily *larger* than the original block, by these rules), copy the user's data from the old block to the new block, and then *free the old block*.

Requests for an allocation of size 0 to any of the allocator functions may either use the pool allocator to create a minimum-size allocation or return NULL, *as you prefer*. You will probably find that simply returning NULL is easier, but if for some reason your algorithm benefits from the possibility of returning a minimum-size allocation, you may do so.

Your allocator **must** be capable of returning freed memory to the heap and reusing it to serve future allocations. In particular, if a program repeatedly allocates and frees many small objects with a constant upper bound on the number of objects and total space required, your allocator should eventually settle on a heap size that requires no additional requests to the operating system for more memory.

You must *test your own project.* The tests provided in Autograder *will be incomplete*, and you should not rely upon them to give more than a rough estimate of your final score. There is no requirement to submit your tests for this project.

3.1 Metadata and Global Symbols

As described in CS:APP in Section 9.9, a heap allocator *must not* use any global state of non-constant size that cannot be allocated from the heap itself. Therefore, you should declare only *primitive* global (or static) variables for your allocator, and you must use the heap for any other data required. Your implementation will add a few constant bytes to each allocation and store the rest of its variably-sized metadata in free blocks, requiring only a constant allocation at initialization time for its remaining storage.

3.1.1 Static Variables

Any global variables or functions declared by your implementation in `mm.c` **must** include the keyword `static`. This will ensure that they are not visible to any other translation units, and that your allocator's function cannot be disrupted by code that may link to it. (See Section 4.6 of K&R for more discussion of static variables, and the lecture *The Compiler and Toolchain* for a definition of translation units.)

3.1.2 Block Headers

The metadata stored for each block of memory managed by your allocator (allocated or free) should be a 64 bit *header* word **preceding** the allocation (or free block), containing the size of the allocation and some flag bits. Note that the minimum possible block size for the pool-allocated blocks is 2^5, or 32 bytes, and that all pool-allocated blocks have sizes which are powers of two. Recalling the format of integers in memory, this means that the lowest *five bits* of the integer size of a block, whether it is free or allocated, will always be zero. (This is because the binary representation of 32 is 100000_2.) Those bits can therefore be used to store flags, and then masked out when retrieving the size of a block. CS:APP explains this in some detail in Section 9.9.6 (In particular, see Figure 9.35).

You should use the lowest-order bit (the "ones bit") to indicate whether a block is free or allocated; a one bit will indicate an allocated block, and a zero bit will indicate a free block. While this is not *entirely necessary* to complete this project, it makes the implementation of a heap validator (suggested in Section 5) much more effective. Remember to account for this when allocating and freeing bulk-allocated blocks, if necessary. You will probably find that *bitwise operations* are the best way to maintain the allocated bit and other bits in the size field, rather than using `+ 1` and `- 1` (which has for some reason proven both a popular technique and a popular source of bugs in the past). Think about what this means for bulk allocations, and particular block allocations of odd sizes or *exactly 4089 bytes*, if you choose to do this.

The pointer returned by the allocation functions, and the pointer given to `free()` or `realloc()`, is the first byte of memory *after* the header word. Your implementation will have to use pointer math to find the header word for a block that is passed to `free()` or `realloc()`, and perhaps to calculate the address to be returned from `malloc()` *et al.*

This metadata is the reason that your allocator's effective allocation sizes are 8 bytes smaller than an even power of two. The block from which an allocation is drawn is a power of two in size, but there are 8 bytes of header used from that block for storing metadata.

3.2 The sbrk System Call

The operating system provides two system calls to manipulate the *program break* between the application's heap and the unmapped memory region above the heap: `brk()` and `sbrk()`. The `brk()` system call accepts an *absolute address* and immediately sets the program break to that point, while `sbrk()` accpets a `relative size` and adjusts the current program break by that amount. It is difficult to use `brk()` correctly and safely, so we will use `sbrk()` for this assignment.

Passing a positive value to `sbrk()` causes the system to move the program break the requested number of bytes *upward* in memory, toward larger addresses. It will then return the *old value of the program break*, which is a memory address. The newly allocated memory therefore begins at the address returnd by `sbrk()` and continues for the number of bytes requested by the application.

3.3 The Multi-Pool Allocator

Your allocator **must** use the following pool allocation policy for allocations between 1 and 4088 bytes. All allocations from the pool allocator portion of the heap will be in blocks of size 2^y, $5 \leq y \leq 12$.

All allocations of size 4088 bytes or smaller will be rounded up to the next larger integer size x satisfying the equation $x = 2^y - 8$, for some y between 5 and 12 (inclusive); that is, $24, 56, 120, \ldots, 4088$. These allocations will be served out of blocks of memory 8 bytes larger than x, to allow space for your allocator to store metadata about the allocation. This means that the smallest unit of memory your pool allocator will manage is 32 bytes (including metadata), and the largest unit of memory your pool allocator will manage is 4096 bytes (including metadata).

The provided function `block_index()` will return y (which you may find more convenient than x itself for some operations), given x. In other words, it will return the *log base 2 of the block size required to serve the requested allocation*. You can use this as an index into a table of pools containing free blocks of specific sizes, as discussed in Section 3.5.

Your allocator will maintain a separate free list for each of the block sizes described above. Every block on the free list for a given size will be of that size, and available for allocation. When your allocator attempts to allocate a block of a given size and there are no blocks on the relevant free list, it will request `CHUNK_SIZE` (4096) bytes of memory from the OS using the `sbrk()` system call, break the returned allocation up into blocks of the desired size, and place them on the appropriate free list. By calling `sbrk()` relatively infrequently and serving allocations out of fixed-size blocks in this fashion, you improve the run-time performance of your allocator at the expense of some memory overhead.

Each allocation in your multi-pool allocator should first attempt to find an *existing free allocation block of the desired size*. If such a block exists, your allocator should use it. If it does not, your allocator should request 4096 bytes of memory from the OS as described above, then retry the allocation.

3.4 The Bulk Allocator

For all allocations of larger than 4088 bytes, a bulk allocator must be used that maps a single object into the process's memory space directly, and unmaps it on free. The following provided methods should be used to accomplish this; you will reserve 8 bytes of extra memory for bookkeeping when calling these functions; *i.e.*, if your allocator is serving a request for 4096 bytes of memory, you will request 4104 bytes from the bulk allocator. These extra 8 bytes will be used to provide a header for bulk allocated objects in the same fashion as pool allocated objects, so that they can be freed easily.

- `void *bulk_alloc(size_t size)`
- `void bulk_free(void *ptr, size_t size)`

The `bulk_alloc()` function returns a pointer to a contiguous block of memory of at least `size` bytes, similar to `malloc()`. However, unlike `malloc()`, this allocation cannot be freed with `free()`. Instead, `bulk_free()` will free this block when provided with *both its address and its size in bytes*. You may view the implementations of these functions in the provided file `src/bulk.c`, although you should not modify them and you do not need to understand them. They use the system calls `mmap()` and `munmap()` to request specific modifications to the process's memory map.

Note that *you must provide the size of the allocation being freed to the bulk allocator.* It cannot determine this itself! This is the purpose of the extra 8 bytes of overhead requested by your allocator.

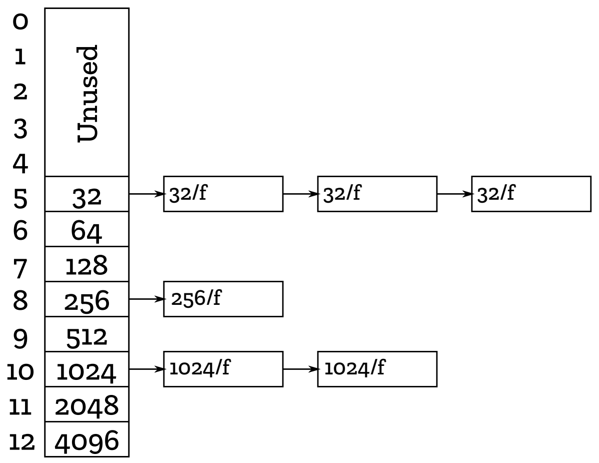

Figure 1: A free list table containing six free blocks of memory. Each block is labeled with its size and free flag.

3.5 Free List Table

You must store the free blocks in your allocator in a set of *explicit free lists* (as opposed to the implicit free list described in CS:APP). For an explicit free list, the allocation space inside a free block (which is otherwise unused) is used to store pointers to the next and previous free blocks on the list. You can define a structure, similar to the `ListNode` structure you used for PA2, that includes the free block header, a next pointer, and a previous pointer, and then use pointer type casting to map it to the address of a free block on the heap. You may find that your PA2 code can be used almost directly for this project!

Note that this project requires only that you are able to insert and remove from the head of a list. You should *never* walk a list. It is important that `malloc()` and `free()` are constant-time operations, and for this reason, their run time should not depend on the length of free lists. You can likely therefore use singly-linked lists for this project. All of the allocation sizes are large enough for doubly-linked lists if you wish to use your PA2 code, however.

As explained in CS:APP, Section 9.9, this sort of free list requires *no extra space*. The pointers that make up the free list use the space inside the free blocks that would otherwise be used by the application when the blocks are allocated. *Make sure that you understand this subtle point.* It is critical that you do *NOT* manipulate these pointers on an allocated block!

By creating a table of these lists indexed by the log base 2 of the size of objects at each entry, you can use `block_index()` to very quickly determine whether a free block of a given size is available. The block sizes for this project were carefully chosen to be precise powers of two for this very reason. Figure 1 shows an example depiction of such a table.

You must allocate the free list table itself on the heap. To avoid `malloc()` depending on `malloc()`, you may wish to allocate it directly from memory acquired with `sbrk()`; because it will never be freed, this is not a problem, and it does not require a header or any other allocation structure. Note, however, that this is *very inefficient* if you do not *reuse the rest of the memory allocated using sbrk()* as blocks that can be allocated by the user. The free list table is about 100 bytes, and you must call `sbrk()` for `CHUNK_SIZE` bytes!

4 Guidelines

There are many aspects of this project that you may choose to implement in a variety of ways. You have examples of some portions of the project in the form of Section 9.9 of CS:APP and Section 8.7 of K&R. This section contains some guidelines on how you might tackle those aspects, as well as general advice to help you succeed in this project.

4.1 Structures and Types

Computer Systems: A Programmer's Perspective presents an allocator implementation that uses preprocessor macros and raw pointer accesses to extract essentially all of the information from allocation blocks. **I do not recommend this approach.** The approach in *The C Programming Language* is more appropriate, although it uses a union in a way that I do not think is necessary (for portability reasons that we are not currently concerned with), and stores its fields *in a different order than is required in this project*.

Consider declaring a structure that encompasses the header word and linked list pointer(s) for the free list, and using that structure to access the free list. You can probably use some of your PA2 implementation's code for this purpose. Note that you can use the same structure for *both free* and *allocated blocks*, provided that you simply *do not access* the parts of the structure that are inside the user's memory in allocated blocks.

The reason for this recommendation is multi-fold, but the two most important considerations are:

- The use of macros to implement program functionality can make debugging *very difficult*.
- Limiting usage of void pointers and pointer casting as much as possible will help the compiler flag errors, by allowing *type errors* to indicate *logic errors*.

4.2 Macros and Functions

The previous notwithstanding, you will probably still need or want some macros or functions to perform type casting and certain pointer manipulations reliably and consistently. You *absolutely should not* have random additions and subtractions of fixed values or `sizeof()` calculations sprinkled all throughout your code! This makes it very easy to miss a location where you should have made such a calculation, and did not; a macro or function to perform this cast and change is a better solution. In general, prefer functions over macros (because of type checking), but in some places macros may be appropriate.

4.3 Task Decomposition

You may find that it improves the clarity and flow of your code by breaking the various tasks down into smaller functions that may even be used by more than one portion of your implementation. You should absolutely do so! Declare `static` functions in `mm.c` to encapsulate such functionality.

In particular, the logic for requesting more heap space using `sbrk()` when necessary and breaking up free blocks to serve smaller allocations will likely be shared between several functions.

Remember also that some operations can be delegated to other required functions — in particular, `malloc()`.

4.4 Pointer Math

You will do a lot of pointer math in this project! You may find it easiest to do this math by first casting the pointer to `void *`, then manipulating it using simple arithmetic and `sizeof()`. Remember that while arithmetic on `void *` behaves "normally", arithmetic on other types operates in increments of the size of the type. This means, for example, that `(void *)0 + 1` is 1, while `(int *)0 + 1` is 4. Therefore, you may find it less confusing to simply cast any value to `void *` before manipulation.

4.5 Frequently Asked Questions

These are questions that will be asked dozens of times in office hours and Piazza, even though they're right here in the document. You'll be sent back to this section when you ask them.

How do I test against Unix commands? Read the comments in the `Makefile`. They tell you how.

Unix commands are running fine, but my allocator fails on Autograder; why? Check to make sure that you don't see an error like this when you run your command: `ERROR: ld.so: object './libcsemalloc.so' from LD_PRELOAD cannot be preloaded (cannot open shared object file): ignored`. If you do, make sure you've run `make`, that your library built properly, and that you're in your project build directory.

Why is `ls` crashing? It's a problem with your `realloc()`. We promise that `ls` works, but for *some reason* it uses `realloc()` for basically everything. Read the man page for the allocation functions very carefully, and implement your `realloc()` completely, and it will run.

How do I run `gdb` with Unix commands? This is pretty difficult to do, because of the way your allocator has to be loaded. A better approach is to run `ulimit -c unlimited` in your terminal before running the Unix command, and then run `gdb` against the resulting core file. For example (`$` is the shell prompt, you wouldn't type it):

```
$ ulimit -c unlimited
$ LD_PRELOAD=./libcsemalloc.so ls
Segmentation fault (core dumped)
$ gdb ls core
Lots of gdb boilerplate here
(gdb) backtrace
```

Everything works, I can run Emacs, but some fancypants program is crashing. Is Unix broken? It probably uses threads. If GUI applications (in particular) are crashing for you, it *might* be your allocator, but don't sweat it terribly.

A program says "Memory exhausted", what does that mean? It *probably* means that your `malloc()`, `calloc()`, or `realloc()` returned `NULL`, so the program thinks that there is no memory left in the computer.

Why do I keep getting the message "munmap failed in bulk_free(); you probably passed invalid arguments"? It means exactly what it says. This *usually* happens because of one of two things: *a)* you passed a pointer to the middle of some bulk allocated object (possibly after the header?) to `bulk_free()`, or *b)* you passed a multipool-allocated object to `bulk_free()`.

Why is my allocator crashing? I can't find the bug! Write a validator. For real. Right now. *It will save you time.*

What is the flag for in the block header? Your validator. You can complete this project without it, but it can make your validator *much* more robust. Consider including it *and* checking it in your validator!

5 Testing

You *should plan to write a heap validator for your project*. This is a function that you can call while debugging to crawl through your heap structure and verify its correctness. You should be able to use your `list_validate()` from PA2 as part of this process, but will also want to check other things that are allocator-specific. You may wish to use debug macros such as discussed in class when talking about the C preprocessor to include extra information for validation, routinely validate the heap, *etc.*, when a debug symbol is defined. **Like drawing your data structures, you will *absolutely* not save time by skipping this step. You should consider developing it along side, or even before, your allocator implementation!**

Your project will be built into a *shared object* (library) when you run `make` in the top-level source directory. You can use this library to run standard Unix system utilities using your allocator, or you can link `mm.o` and `bulk.o` against tests that you write. The project `Makefile` is extensively commented and provides information about how to do both of these things.

Liberally sprinkling your project with debugging statements and correctness assertions that can be turned on and off with the preprocessor will make your job easier. When printing debugging statements, you will want to use `fprintf(stderr, ...)` for a variety of reasons; in addition to the previously-mentioned benefits of this approach (such as unbuffered output), this is *guaranteed not to allocate memory*. When debugging an allocator, that is an important property!

The debugger will be indispensable for this project. Plan to use it often to examine pointer values. Remember that the gdb `print` command will print the value of *any expression*, including pointer arithmetic and casting.

Some example simple programs to test your library against include `ls`, `w`, `echo`, and `ps`. These and many other Unix commands should run under your allocator once it is complete. You can look at their man pages for more information, and TAs may be able to help you find other commands to test. If your allocator becomes sufficiently robust, large applications such as Emacs will even run! Note that, because your allocator need not support multiple threads (we haven't yet talked about how to do this), some applications will *not* run and it is not your fault. If simple command-line applications fail to run, you should suspect your allocator. If large GUI applications fail to run, it is probably that they are trying to use threads. (Emacs 25, which is installed on the course VM, happens to be single-threaded, so it is an exception to this rule.)

TAs and instructors will expect you to have implemented a heap validator and know how to use it on this project. Bugs that appear to be easily identified by a working validator will meet a response of "what does your validator say?" If you need help strengthening your validator, ask!

6 Submission

Use `make submission` to build a submission tarball, `malloc.tar`, that you will upload to Autograder. The Autograder tests for this project will be incomplete, but give an indication of the correctness of your project.

This is a *three week project*. It is a three week project because it will take more time and care to implement than previous projects! Implementing an allocator is a tricky and subtle problem, and you can lose a lot of time to debugging. Plan to start early and work consistently; you will have much more success with this approach than you will with sitting down for a few marathon coding sessions late in the assignment.

7 Grading

The handout quiz for this project will be worth 0.5% of your final course grade, and the rubric below will be worth 9.5% of your final course grade, for a total of 10% of your grade. The handout quiz score will not be reflected on Autograder for this project.

Your project will be evaluated as follows:

Description	**Points**
Handout quiz	1
All allocations are 8 byte aligned	1
All heap data structures are allocated on the heap	1
`sbrk()` is called for *only* `CHUNK_SIZE`	1
Pool allocations are for powers of 2	2
Bulk allocations are used for large allocations	1
`malloc()`, `calloc()`, and `realloc()` use pool allocation	3
`calloc()` properly clears memory	1
`realloc()` can extend (and reduce) allocations	1
`realloc()` copies memory when necessary	1
`free()` returns memory to the heap	2
Functions sufficiently to run simple applications	5