

CS 252: Systems Programming

Fall 2019

Lab 2: Malloc

Prof. Turkstra

Due September 16th, 2019 11:58 PM

1 Goals

- Learn about how `malloc()` works under the hood
- Get used to systems programming environments

2 Deadlines

Part 1

The Part 1 Checkpoint of this lab is due **Monday September 9th, at 11:59pm**. Part 1 includes: `my_malloc` and `first_fit`. The remaining functions (`my_free`, `next_fit`, `best_fit`, `worst_fit`) are due with the final submission.

There will be no late submissions accepted for Part 1. If you choose not to submit a Part 1, your most recent commit before the deadline will be pulled and whatever grade you receive for that will count as 10% of your final grade.

Final Submission

This lab will be due **Monday September 16th, at 11:59pm**. Each day late will receive a penalty of 10% of the points possible.

3 Code Standard

Part of being a competent computer scientist includes the ability to adhere to established code standards. Almost all (good) software engineering environments have a code standard.

We have one in this course as well. It can be found on the course website. You are expected to follow the code standard. An additional grade out of 10 points will be assessed along these lines.

We have provided a linter that *partially* assesses the code standard. Passing the linter does **not** mean that your code fully adheres to the code standard. There are certain parts that are simply too difficult for a linter to assess. It is still your responsibility to ensure that your code is fully compliant. To use the linter, you can run:

```
$ ~cs252/bin/linter your_file.c
```

Alternatively, you can add `~cs252/bin` to your `PATH` environment variable:

```
$ export PATH=$PATH:~cs252/bin
```

Add it to your `.bashrc` file to make it permanent.

4 Development Environment

You are expected to do your development work on CS Linux systems like `data.cs.purdue.edu`. The `Makefile` for CS 252 is specially crafted to commit updates to a Git repository every time “make” is run. Development on other systems will prevent this mechanism from working.

5 The Big Idea

As you no doubt know by now, a major part of C programming is the management of memory. You have used `malloc()` and its kin before, but now it is time to delve into how `malloc()` works.

In this lab, 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 manage all the bookkeeping and memory efficiently.

6 The Assignment

6.1 Diagram Legend

This handout makes use of numerous diagrams to assist in the understanding of the lab. The key for those diagrams is thus:

Legend:

-  Header
-  Unallocated Block
-  Allocated Block
-  Fencepost
-  Currently Selected Block
-  Random things that don't really need color but why not add colors?

6.2 Background

UNIX provides a rudimentary API for managing heap memory. Using the system call `sbrk(intptr_t increment)`, we can extend (given a positive integer) and shrink (given a negative integer) the size of the heap. Unfortunately, `sbrk()` is a very limited syscall: it can only deallocate memory *in the order it was allocated*. This means that if we used `sbrk()` to manage memory, in order to be able to free any memory we would need to free the memory in the reverse order it was allocated. This implementation is too coarse-grained for our needs. Consider the situation below:

```
// extend the heap to allocate the big array
int *big_array = malloc(BIG_NUMBER);
...
// extend the heap to allocate a string
char *name = malloc(32);
...
// big_array cannot be freed until name is freed if we can only
// deallocate by moving the top of the heap back.
free(big_array);
```

If we managed the memory from the OS ourselves, we could allow the freeing of variables like `big_array`, and then could reallocate them later when a user requests more memory.



Essentially, all you get from the OS is a way to extend and shrink the heap

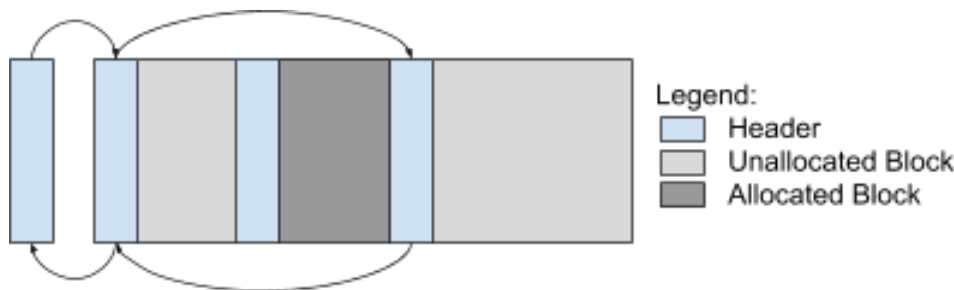
6.2.1 How do we implement a more fine grained allocator?

We know that we somehow need to manage the blocks of memory that we are allocating and freeing. A simple way to do this is to add metadata to each block. The metadata will allow us to maintain the size of each block and whether or not it is allocated. We can request a large chunk from the OS and split it up into smaller blocks to service users' requests. When a user frees a block we can mark it as unallocated so that we can reallocate it for another request. This approach is much more versatile than if we were using `sbrk()` alone.



6.2.2 How do we find blocks which have been freed?

We still need a way to find the blocks that are unallocated. To do this, we use a linked list data structure we call a **free list**. We add a next and previous pointer to each block's metadata so that we can iterate over the unallocated blocks. This allows us to perform a simple list search to find a block which can satisfy our requests.



6.2.3 How should we perform the list search?

When searching the free list for a block to satisfy a user's request, there are four main methodologies employed, each of which will be implemented in this lab.

First Fit Under this scheme, the very first block with a size great enough to satisfy the request will be allocated.

Next Fit This scheme implements the same logic as first fit, but begins the search not at the head of the free list, but at the location in the list after the last block was allocated. The free list is treated as though it is circular.

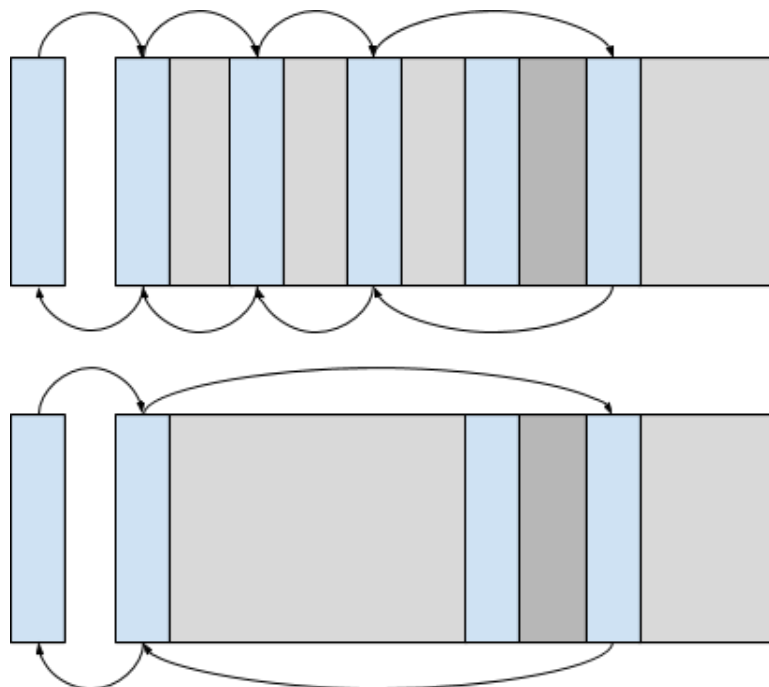
Best Fit This scheme entails searching the entire list for the block with the smallest size still capable of satisfying the user's request.

Worst Fit As suggested by its name, this scheme implements the opposite of best fit, instead searching the free list for the block with the largest size.

Contrary to what the names of these algorithms may suggest, there is no single algorithm that is strictly optimal for all workloads. Each may outperform the others in certain contexts, a fact that will be explored during this lab.

6.2.4 Dealing with memory fragmentation

Now we can allocate memory, and to free it we can return the blocks to the free list. Unfortunately, because we are splitting up large blocks into smaller blocks, when the user requests memory we will end up with many small blocks, but some of those blocks may be able to be merged back into a larger block. To address this issue we could iterate over the free list and try to find if the block we are trying to free is adjacent to another already free block. If the neighboring blocks are free, we can coalesce them into a single larger block.



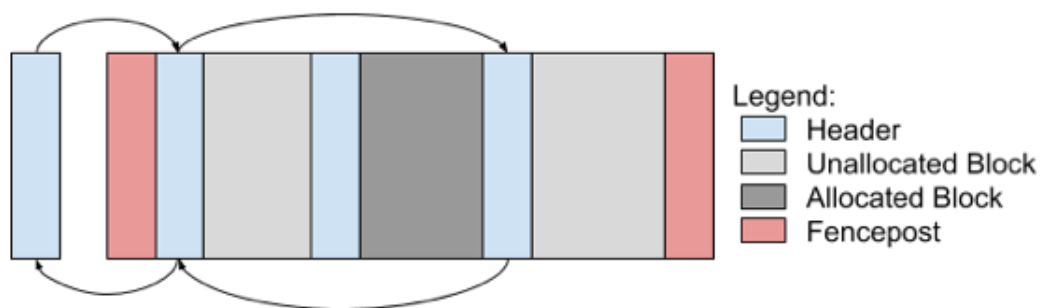
Fragmentation leads to many small chunks in the free list, which cannot satisfy a large request even if the sum of the small chunks could. Coalescing reverses the fragmentation. The fragmented list diagram (top) has a set of three fragmented blocks. The bottom diagram is the same except the fragmented blocks have been coalesced.

6.2.5 Dealing with the edges of the chunks

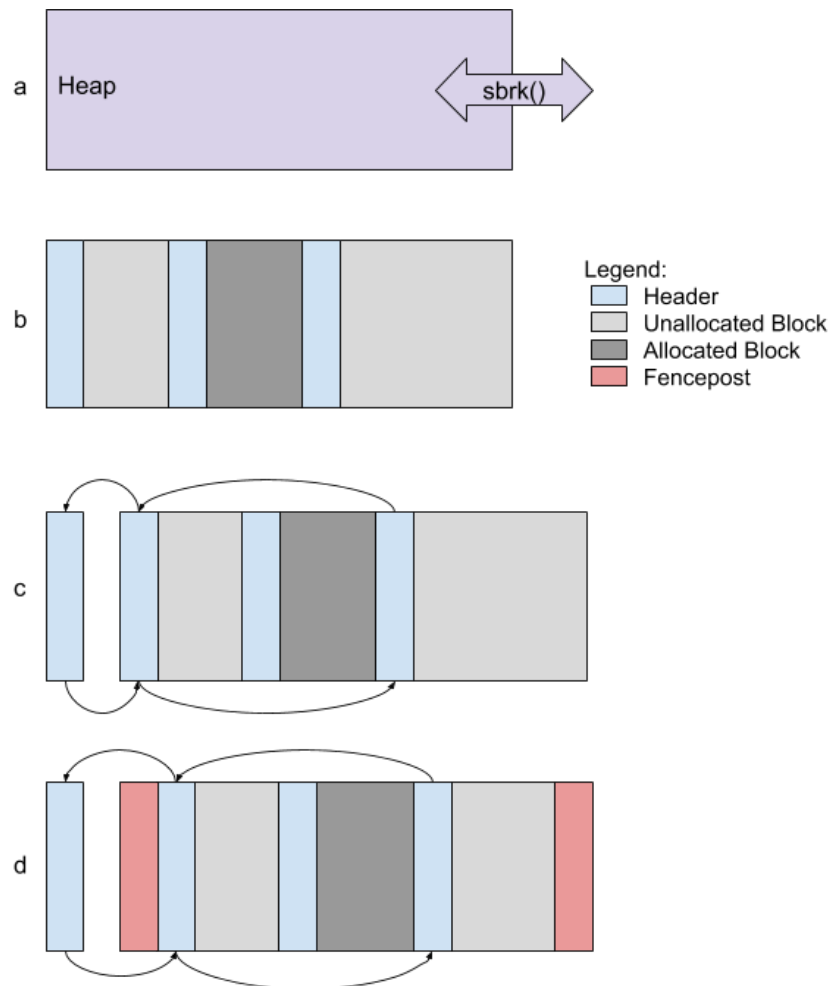
One detail we must consider is how to handle the edges of the chunks from the OS. If we simply start the first allocable block at the beginning of the memory chunk, then we may run into problems when trying to free the block later. This is because a block at the edge of the chunk is missing a neighbor.

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.

Conceptual Question: What would happen if we tried to free and coalesce the first block in a chunk without fenceposts? (Hint: the result is nondeterministic)



This design allows us to allocate and free memory without having memory fragmentation problems, but there are some optimizations we can add to improve space and time complexity



Slowly building up the data structures required to handle memory allocation.

a: The heap supporting only an increase decrease size operation.

b: Adding block metadata to maintain information about different blocks of heap memory.

c: Adding linked list pointers to maintain all free nodes in the free list.

d: Adding fencepost tags to mark the beginning and end of the heap chunks retrieved by `sbrk()`.

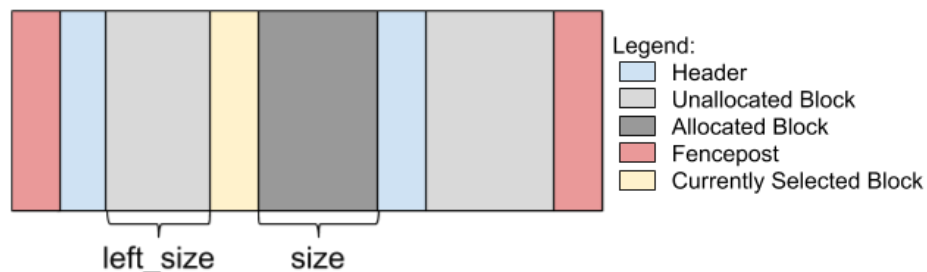
6.3 Optimizations

6.3.1 Constant Time Coalesce

Naive solution Above we mention that we can 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.

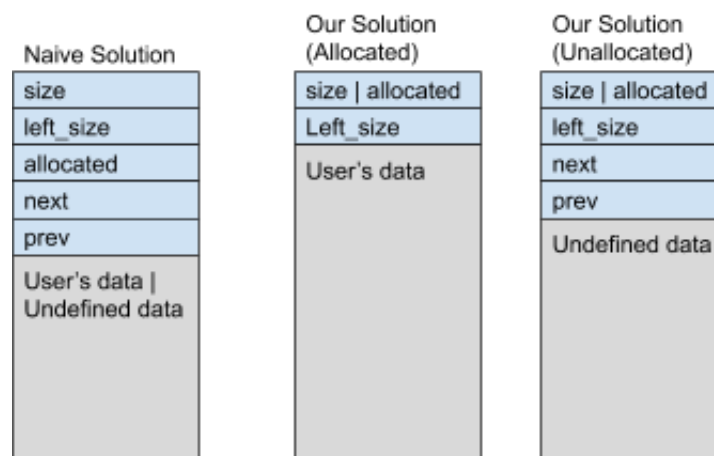
Optimized solution The solution we will use is to add another field to the metadata called `left_size`, which contains the size of the block to the left of each block. Using this value in conjunction with pointer arithmetic will allow us to locate the left neighbor of a given block in $O(1)$ time, rather than $O(n)$.

Conceptual Question: Is `left_size` useful when allocating?



Using the size fields, we can calculate the location of the neighbor blocks without needing the free list and in $O(1)$ time.

6.3.2 Reducing Our Metadata Footprint



Naive Solution For any solution, the user's data must be stored after the metadata of the block. Accordingly, a naive implementation would be to simply use a static header structure with one variable for each metadata component, and place the user's data at the end of the header. However, this simple implementation begets unnecessary space overhead.

In the above diagram, the “allocated” field specifies whether the block is unallocated, allocated, or a fencepost. Given that our solution only allows for the allocation of memory in 8-byte chunks, the 3 least significant bits of the “size” field will never be set. Accordingly, we can use these bits to denote the allocation state, removing the need for an extra field in the header.

Additionally, when a block is allocated, it is definitionally no longer part of the free list. As such, there is no need to maintain “next” and “prev” pointers in allocated blocks, allowing an additional $2 * \text{sizeof}(\text{pointer})$ bytes of storage to be allocated to the user (provided we place said pointers at the end of the header).

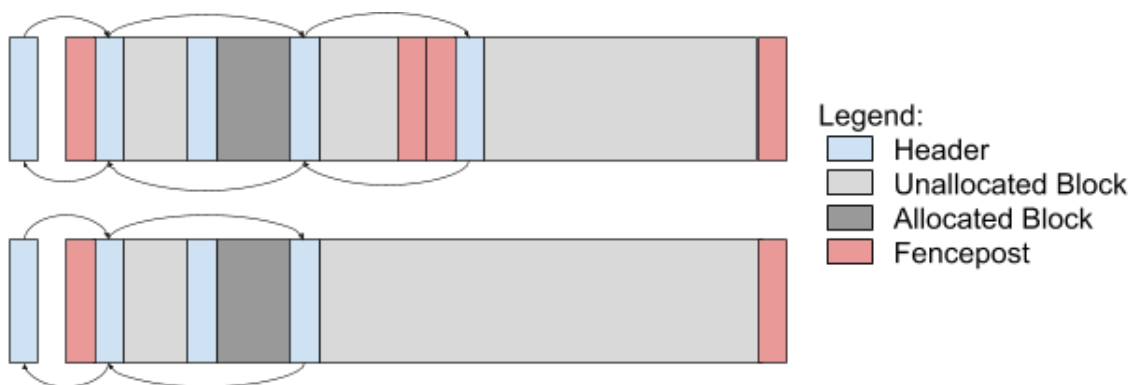
Conceptual Question: What is the minimum number of bytes an allocation can take up with this solution, including both the metadata and the memory requested? (Assume 64-bit architecture)

6.3.3 Coalescing chunks from the OS

Naive Solution When additional memory is required from the OS we can simply add the new chunk to the appropriate free list, but this limits our allocator to only service allocation requests which are less than the arena size.

Optimized Solution Recall that when calls are made to `sbrk()`, they are retrieved in ascending order. As long as the user program does not call `sbrk()` in between calls we make to `sbrk()`, we can coalesce the new chunk with the most recently allocated chunk. This allows us to form larger chunks which can service requests larger than a single arena’s size.

Conceptual Question: Is it possible to coalesce chunks that are not contiguous?



When chunks from the OS are adjacent, we should coalesce them with the last block in the previous chunk.

6.4 Lab Specification

6.4.1 General Malloc Spec

The malloc interface is specified in the malloc man page, but leaves many details up to the library’s authors. Consider the numerous optimizations mentioned above; all versions of malloc would be

correct by the specification in the man page, but some are better than others, as we've shown.

6.4.2 Our Implementation Design Spec

In the background and optimization sections above, we have described the basic implementation we will follow with a set of optimizations. Here we will cover the technical specification of the required design. Some of the requirements are in place to enforce conformance to the design, but others are purely to guarantee determinism between our reference allocator and your allocator for the purposes of testing. The specification below should contain all of the details necessary to make sure your implementation is consistent with the reference implementation. You should read the specification, but it is more for reference when working on the lab and below the specification is a more detailed description of the various tasks to be completed in the lab.

6.4.3 Terminology

Term	Definition
chunk	A region of memory sized as a multiple of <code>ARENA_SIZE</code> retrieved from the OS using the <code>sbrk()</code> call. The size of a chunk can be further increased by coalescing two chunks together
block	Piece of memory containing both metadata and allocable memory. Freed blocks make up all the nodes of the free list
metadata	A general term for pieces of information malloc uses to manage blocks
allocated metadata	Metadata maintained when a block is allocated. This excludes metadata that concerns the free list. It only includes: <ul style="list-style-type: none"> - size - left_size
unallocated metadata	Metadata maintained when a block is not allocated. This includes all the allocated metadata and the free list pointers: <ul style="list-style-type: none"> - next - prev
header	An implementation of metadata. Can either include the free list pointers or not (see the header struct definition in <code>my_malloc.h</code>)
fencepost	A header with only allocated metadata that denotes the edges of a chunk. When coalescing chunks, the two fence posts between the chunks must be removed
<code>ARENA_SIZE</code>	The number of bytes of memory requested of the OS using the <code>sbrk()</code> call should be a multiple of this value. This is a parameterized value which is set for each test case individually (see the tests Makefile: <code>-D ARENA_SIZE={size}</code>)
request size	The size the user program has requested.

6.4.4 Data Structures

- The `size` and `left_size` fields refer to the amount of space that can be allocated to a user. This does not include the size of the metadata, with the exception of the “next” and “prev” pointers.
- All insertions into the free list are made to the **beginning** of the list.

6.4.5 Allocation

- An allocation of 0 bytes should return the NULL pointer for determinism
- All chunks requested from the OS should be the minimum multiple of `ARENA_SIZE` defined in `my_malloc.h` necessary to satisfy the request (remember that some of the allocated space will go toward metadata).
- All requests from the user are rounded up to the nearest multiple of `MIN_ALLOCATION` bytes
- When allocating a block there are a few cases to consider:
 - If the block’s size is exactly the request size the block is simply removed from the free list.
 - If the block’s size 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’s size is larger than the request size and the remainder is large enough to be allocated on its own (note that the remainder must be large enough to contain an entire unallocated header), the block is split into two smaller blocks. The block that is lower in memory should have its size set to exactly the size needed to satisfy the user’s request, and should be allocated and returned to the user. The block that is higher in memory should have its size set to the remaining space, and should be inserted into the 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. No chunks should be requested from the OS until the first call to malloc is made.
 - When allocating another chunk there are two cases we must consider:
 - * If the new chunk from the OS is adjacent to the last chunk allocated from the OS, the chunks should be coalesced into a single chunk.
 - * If the new chunk from the OS is not adjacent to the last chunk allocated from the OS, the chunk should simply be inserted into the free list.
 - In operating systems, you can never simply expect a call to the OS to work all the time. If allocating a new chunk from the OS fails, the NULL pointer should be returned and `errno` should be set appropriately (see the man page.)
 - New chunks should be allocated lazily. That means that more memory is requested only when servicing a request that cannot be satisfied by any of the available free blocks.

6.4.6 Deallocation

- Freeing a NULL pointer is a no-op (don't do anything).
- Freeing an un-allocated block is an error (you should `assert(false)`).
- When freeing a block, there are a few cases to consider:
 - Neither the right nor the left blocks are unallocated. In this case, simply insert the block into the free list
 - Only the right block is unallocated. Then coalesce the current and right blocks together. The newly coalesced block should remain where the current 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. 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

6.5 Tasks

- Allocation
- Deallocation
- Managing multiple chunks

6.5.1 Task 1: Allocation

1. Calculate the required block size (actual request size) by rounding the allocation size up to the next `MIN_ALLOCATION`-byte modulo.
2. Find the appropriate block to allocate based on the allocation algorithm (controlled by `FIT_ALGORITHM`)
3. Depending on the size of the block, either allocate the full block or split the block and allocate the left (lower in memory) portion to the user.
 - (a) When splitting, remember to update the `size` and `left_size` fields
4. When allocating a block, update its allocation status to `ALLOCATED`
5. Finally, return to the user a pointer to the `data` field of the header.

6.5.2 Task 2: 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 header of the block being freed we must calculate the locations of its right and left neighbors, also 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 with one or both of the neighboring blocks

6.5.3 Task 3: Managing Additional Chunks

Above we don't specify how to handle the case where the user's request cannot be fulfilled by any of the available blocks.

1. If no available block can satisfy an allocation request then we must allocate more memory from the OS. To do this we must call `sbrk()`.
2. It is possible that, between two calls to `sbrk()`, the user program itself may have called `sbrk()`, in which case a new chunk should be allocated normally. However, if the two calls to `sbrk()` return consecutive memory chunks, they should be combined together with no fenceposts between them.

6.6 Implementation Notes

- Whenever referencing the size of a variable or structure, you must use `sizeof` instead of hard coding a constant. One of the tests uses the `-m32` flag in gcc to compile the test as a 32-bit binary. For instance, use:

```
x = sizeof(size_t);  
DO NOT USE: x = 8;
```

6.7 Coding

Skeleton Code

Log into a lab machine and type:

```
mkdir cs252  
cd cs252  
git clone ~cs252/repos/$USER/lab2.git  
cd lab2
```

Note: Before you start working on the lab read through the provided code. Make sure you understand how the data structures work and how the initialization code works. Taking time to understand the skeleton will save you a lot of time in the end.

my_malloc.c

The main implementation file for the project. Provided is an initialization function which performs some necessary setup for you prior to the `main` function running. Feel free to add additional code to this function if you desire, but do not remove any of the existing functionality.

The functions `calloc` and `realloc` have been added for you as wrappers for your `malloc` function, which you must implement. You must also implement `free`, which, along with `malloc`, must be guaranteed to be thread safe (hint: `pthread_mutex_lock`).

In addition, you must implement each of the 4 functions that implement a mechanism for selecting blocks from the freelist: `first_fit`, `next_fit`, `best_fit`, and `worst_fit`.

It is recommended that you implement your solution in a modular way, for both your clarity and that of your TAs. Try to piece out individual steps or procedures into smaller functions, so that it is easier to pinpoint where issues may occur, and understand the flow of your code. Additionally, modularity lends itself very well to reducing repetitive code, of which there can be a lot in this project, if implemented poorly.

A small number of additional short functions have been implemented for your benefit. It is recommended that you examine these functions to make sure you understand both how they function, and what their implications are for the overall structure of the data in this assignment.

printing.c

Helper functions for printing the free list and arbitrary blocks

tests/test*.c

Test source files

6.8 Using “make” and Testing

To make your code type

```
cd lab2
make
```

The make command will build your malloc sources and push the latest changes into your git repository. It is important that you use “make” instead of directly typing the compiler command to make sure that a git log is built for your project. Projects without a git log will not be graded.

All tests may be run from the “tests” directory. To run all of the tests, type

```
make testall
```

Of the 100 points possible for this lab, 80 of them are proportional to the number of these test cases that your program passes. There will be an additional 10 points of tests used during the final grading. The remaining 10 points on the final grade are from the checkpoint score (proportional to the number of test cases you pass then).

To run only the tests that will be included in the Part 1 Checkpoint, you can run

```
make testPart1
```

To run only a single test, type

```
make test<number>
```

The “expected” directory contains the output your program should match (WITH THE EXCEPTION of non-deterministic quantities such as raw memory addresses. Such values are not shown in the expected output files; do not worry about them, nor about differences in the number of empty lines) for tests that produce output. The output of your programs may be found in the “output” directory.

For debugging purposes, output is simple text by default, but color printing can be enabled by setting the MALLOC_DEBUG_COLOR environment variable to 1337_CoLoRs

To enable: export MALLOC_DEBUG_COLOR=1337_CoLoRs

To disable: export MALLOC_DEBUG_COLOR=false

NOTE THAT ENABLING COLOR WILL CAUSE TESTS WITH EXPECTED OUTPUT TO FAIL. Make sure MALLOC_DEBUG_COLOR is set to false before running the test suite.

7 Turning in Your Work

Checkpoint 1 Turnin (Monday September 9)

1. Login to a CS department machine
2. Navigate to your lab2 directory
3. Run `make clean`
4. Run `make` and check that the tests look correct
5. Run `make submit_part1`

Final Turnin (Monday September 16)

1. Login to a CS department machine
2. Navigate to your lab2 directory
3. Run `make clean`
4. Run `make` and check that the tests look correct
5. Run `make submit_final`