Summary

In this project we worked on implementing a serial dynamic storage allocation system, that provides the following three interfaces for user programs to utilize:

- *my_malloc(size_t size):* allocates a memory of size at least size and returns a pointer of this memory block to the user.
- *my_free(void *ptr)*: deallocates the memory block pointed to by ptr. The memory block pointed to by ptr needs to have been been previously returned by malloc or realloc and not freed.
- my_*realloc(void *ptr, size_t size):* deallocates the memory space pointed to by ptr, allocates a new memory block of size at least size, copies the contents of the old memory block to the new memory block and returns a pointer to the new memory block. The memory block pointed to by ptr needs to have been been previously returned by malloc or realloc and not deallocated..

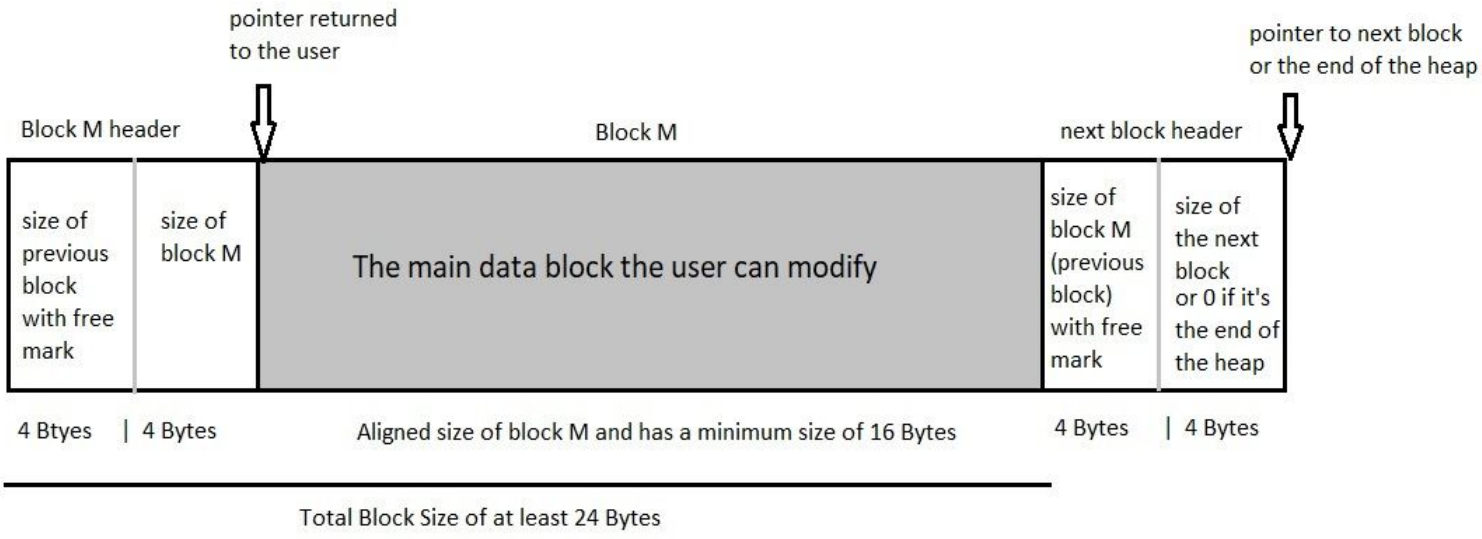
The initial implementation had the following simple functionalities: my_malloc(size) always extended the heap by size; my_free(ptr) did nothing; my_realloc(ptr, size) deallocated the memory block pointed to by ptr via my_free(ptr), allocated a new memory block via my_malloc(size), and copied over data from the old block to the new block via a call to the inbuilt memcpy. This implementation achieved a very high throughput. However, the code did not reuse deallocated memory blocks. That resulted in a critically low level of space utilization. With this implementation, the storage allocator was likely to crush early due to running out of heap space.

The major optimization we made was to use an array of free-lists to store deallocated memory blocks. We reuse these blocks for future allocations as long as the requested size is greater than all the blocks present in the array of free-lists. This allowed us to significantly improve our space utilization while, after making many small optimizations, not hurting our throughput significantly. This resulted in better overall performance.

Project description

The general goal of our project is to maximize space utilization while maintaining high throughput. To do this we use a doubly linked list to store pointers to deallocated memory blocks. We create 27 of such linked lists, stored in an array *bin* where *bin[0]* contains a doubly linked list of blocks of size smaller than 64 bytes and bigger than or equal to 24 bytes. Each remaining linked list at *bin[i]* contains blocks of size bigger or equal to 2^(5 + i) and smaller than

2^(6 + i) bytes. We use the memory blocks themselves to store the free-list structure. Our free-list structure simply contains two pointers, one to the previous block and one to the next. Thus one node of the free-list occupies 16 bytes of memory in the architecture we work with. Also to allow easy access to the previous and forward blocks we added a header before each block containing the size of the block itself and the size of the previous block. Since the maximum size of a block in this project is not going to be bigger than 2^32, we decided to store these sizes as integers. Thus each header occupies 8 bytes of memory, abstracted as a simple struct containing *int size* and *int prev_size* fields. This results in a minimum overall block size of 24 bytes. Besides the sizes of the previous and current blocks, the header also stores information indicating whether the previous block is free. Since we align the sizes of the blocks to 8, we know the *prev_size* field is not using the last three bits. We set the least significant bit of this field to 1 when the previous block is freed and to 0 otherwise. The structure of a block M can be visualized as in the following figure.



Performance bottlenecks

The original implementation behaviour was as follows:

- The malloc function extends the heap each time it's called by the required aligned size and returns a pointer to the newly created block.
- The free function does nothing and assumes that the block was deallocated.
- The realloc function extends the heap by the new size and copies the old data to the new allocated size. Then returns a pointer to the beginning of the new allocated block.

As can be seen, the provided implementation wastes a lot of space as it extends the heap each time we call malloc. The space utilization is in face 0% because none of the previously freed blocks gets ever used in the next allocations. Also, the realloc function is very slow as it has to copy all the data each time it is called and it wastes a lot of space since each realloc requires as to extend the memory be the new requested size which wastes the previous block space. The provided realloc was actually running out of heap space and failing due to the amount of wasted space.

Coding and Optimizations

Fortunately in this project, we struck on the right idea before we began coding. We layed out the exact optimization we were going to make to the finest detail, and we needed to adjust it very little when we started coding. We have summarized our optimizations in the following overview section and we explain in detail how each function works below.

Optimizations Overview:

The first and major optimization was using binned linked lists to store deallocated blocks. We added header sections to each block to assist in calculating the locations of the next and previous blocks, and to know whether the next or previous blocks were free. These headers allowed us to easily implement coalescing. When we free a block, we check if the next and previous blocks are free and coalesce with either or both if they are free. When we allocate a block from the free-list, we check if the block size is bigger than that requested and split it if the extra size is bigger than the minimum block size we store.

The second optimization we made was on realloc. When the requested size is less than the size of the memory block we are freeing, we simply reuse the passed block's first size bytes and return it back to the user. This improved throughput, as it eliminates the need to copy data and walk the free-find to find a free block of the right size.

Detailed explanation of our code:

The major functions in our code are *void* *my*_malloc(size_t size), void my_free(void *ptr), void *my_realloc(void *ptr, size_t size), void split_free_list(int alinged_size, free_list_t *free_list, int free_list_size), void *malloc_from_free_list(size_t size), void delete_node(free_list_t *free_list, int bin_index) and void *coalesce(void *ptr)*.

Helper functions:

*split_free_list(int alinged_size, free_list_t *free_list, int free_list_size):*
Splits the passed free_list into two blocks of size aligned_size and free_list_size - aligned_size

malloc_from_free_list(size_t size) :
Looks in the bin to find a block of size at least size and returns a pointer to it

*coalesce(void *ptr):*
Checks if the neighbors of the block pointed to by ptr are free, and coalesces the block with either or both and returns a pointer to the new block.

*void delete_node(free_list_t *free_list, int bin_index):*
Deletes the free-list node pointed to by free_list, that must be contained in the linked list at bin_index.

my_malloc:

When malloc is called we follow the following procedure:

1. We align the requested size + size of header to the nearest 8 bytes alignment and check if the aligned size is smaller than the minimum block size. If it is then we change the aligned size to be the minimum block size.
2. We first check the freelist bin that corresponds to the aligned size. This can be computed easily by taking max between 0 and (27 - number of leading zeros in the aligned size). We iterate over the linked list and we check if there is any block in the freelist of size bigger or equal to the aligned size. If we don't find any match, we check the first element of the bins bigger than the aligned size bin and if there is any free block we just use that block as it's guaranteed to be bigger than the aligned size. Moreover, to minimize space wasting, we check if the free list block has more than 24 bytes(minimum block size) extra space above the aligned size and if it does we add the extra space to the corresponding free list and we return a pointer to the first half of the block.
3. If the previous attempt doesn't work, we check if the last block in the heap is empty. If it is, we extend the heap by (requested size - the last block size) and we return a pointer to the last block in the heap.
4. If the previous attempts doesn't work we extend the heap size by the aligned size and we return the pointer of the new block.
5. Before returning any pointer to the user we store the size in the header. We also store the size in the next block header as the previous size of the next block and we change the least significant bit to be 1 to mark the block as free. Since the

size is aligned to 8 bytes we know that the least significant bit is 0 so changing it to one doesn't result in any information loss.
6. We return the pointer of the block.

my_free:

When free is called we follow the following procedure:

1. We first try to coalesce the pointer by checking if the block in front of it is free. If it is, then we add it to the current block.
2. We then check if the block before the pointer is free and if it is then we add the current block to the free block and our new pointer now points to the beginning of the pervious block.
3. When coalescing any two blocks we have to remove the free block from it's free list and we can do that easily because we have stored pointers to the previous and next node inside our free block. We also update the size of the new block to include the added block and header.
4. Then, we add the block to the corresponding freelist bin and we update the pointers accordingly.
5. Finally, we update the header of the block with the new block size. We also update the header of the next block and we make sure to mark the block as free by setting the last bit to 0.

my_realloc:

When free is called we follow the following procedure:

1. We first check if the required size is equal to 0. If it is, we free the block and return a NULL pointer.
2. We then check if the current block size is already bigger than the requested size. If it is, we free the remaining of the block if it has more than 24 bytes extra space and then we return the same pointer of current block.
3. After that, we check if the block in front of the current block is empty and if adding that block will result in a size at least as big as the requested size. If this is true then we add the block to our current block and we free any extra space as described previously. We return the same provided pointer as block starting point didn't change.
4. Then, We check if the current block is the last block in the heap. If it is, we extend the heap by the required size and we return the same provided pointer.
5. Finally, If none of the above attempts succeeds we extend the heap by the requested size and then copy the data in the old block to the new one. We then free the old block and return the pointer of the new block.
6. Before returning any of the previous pointers we make sure to have the appropriate sizes and free marks in the headers as described previously.

Testing, and debugging

Our testing and debugging code is divided into two main sections. The first one is the code contained in valdiator.h and the other is inside the allocator.c file.

validator.h

Checks that
• my_malloc() and my_realloc() don’t return NULL.
• Allocated ranges returned by the allocator are aligned to 8 bytes.
• Allocated ranges returned by the allocator are within the heap.
• Allocated ranges returned by the allocator do not overlap.
• my_realloc() copies existing data to the new block an existing allocation

allocator.c

The *check_coalesce()* function checks if there are blocks in the free-lists that should be coalesced but are not.

The *check_all_free()* function checks if there are blocks in the free-lists that are not marked free.

The *my_check()* checks that all the invariants are preserved by making calls to the above two functions and making sure our header file contains the right sizes for the blocks.

War stories

One of the greatest difficulties of this project was finding the bugs, as problems in the code usually carry over from previous parts and show themselves in unrelated parts of the code. Also, since malloc and free depend on each other we had to implement both free and malloc before being able to run the code. This resulted in a very buggy code which made us stay awake for many hours just reading the code over and over again. One of the bugs that took us a long time to discover was a very simple casting of a pointer that didn’t give any warning but it made the math we did on that pointer completely wrong. Fortunately, we were able to get our code running by sunrise.

Code summary

Summary for Yosef:

2019-10-29: +55 -12
2019-10-30: +312 -147
2019-10-31: +52 -38
2019-11-01: +307 -116

Summary for tammam:

2019-10-30: +329 -103
2019-10-31: +493495 -262
2019-11-01: +14 -7
2019-11-02: +18 -5

Total summary:
2019-10-27 thru 2019-11-03: +494582 -690

Time sheet

#	Date	Start Time	Duration		Task
			Yosef	Tammam	
1	Tue, Oct 29	5:00 PM	3	3	Read through handout, explored the codebase and discussed different approaches that we can take for the implementation.
2	Tue, Oct 29	8:00 PM	2		Implemented validator according to the provided instructions.
3	Tue, Oct 29	10:00 PM	4	4	Pair programmed the first draft of malloc and free.
4	Wed, Oct 30	1:00 AM	6	6	Worked on debugging malloc and free. Was able to get them working and had a great utilization and throughput..
5	Wed, Oct 30	8:00 AM		7	Worked on optimizing malloc, free and realloc by adding some more minor changes.
6	Wed, Oct 30	5:00 PM	3		Worked on optimizing the realloc.
7	Thu, Oct 31	6:00 PM		5	Worked on tuning parameters to get the best performance and some minor restructuring.

8	Fri, Nov 1	12:00 pm	5		.Worked on improving the test suite and restructuring code
9	Fri, Nov 1	9:00 - 10:00 PM;		1	Added minor optimization malloc.
10	Sat, Nov 2	8:00 pm	2	5	Worked on report.
Total			**25**	**31**	