Dynamic Memory Allocator Project - Writeup and Documentation

Team info: Aditya Joglekar 201401086

Rajdeep Pinge 201401103

Project Intro: We have implemented a dynamic memory allocation package(based on the standard libc malloc package implementation). We had received a code which had an implicit free list strategy for management of free blocks and first fit. The task was to find a better implementation to improve throughput as well as memory utilization.

How can we improve the performance?

We first implemented the explicit free list paradigm which is a slight improvement over the naive starting algorithm. While searching for an appropriate free block, instead of going through the entire heap, we maintain an 'explicit' list of free blocks. Thus in the worst case the search for a suitable block will be of the order of the number of free blocks instead of the total number of blocks. We need to modify the mm_malloc and mm_free functions and we also need to store the pointers to the next and previous free blocks.

Where do we store the pointers to the free list?

Once a block is freed, in principle it is free to be reused. So its payload is no longer needed and hence we decide to store the pointers here in the block itself to avoid wasting any extra space.

We further extended this design to the 'segregated free list' design. The functions used in these two designs are similar and hence will be discussed below.

Segregated free list design

The worst case performance for the explicit free list is O(number of free blocks). To improve on this, we decided to divide all free blocks into size based classes. We then made 'separate' lists for each class size. Basically when we need to search, we can simply ignore the lists which have class sizes less than the required size since no free block in these lists is big enough for the requested size. This, significantly improves our search time.

The above was the basic principle used in the segregated list paradigm. Other optimizations made are as follows:

Changes made to important functions for optimization or to handle segregated lists:

1. Proper number of lists. We found that having 10 to 20 lists gives a good enough performance. If we increase the number of lists too much then the search time within a list on an average goes down.
2. We defined some new macros to efficiently handle the increased complexity of code and to give clarity to code. New macros are for total number of list-classes in the segregated

free list, minimum size of a free block and to get and set predecessor and successor blocks of a given free block.

3. **mm_init**: It was specified that the pointers to the free lists **must be in the heap itself**. Therefore we simply shifted the pointer pointing to the start of the heap(heap_listp) and used the initial space to store these pointers to segregated free list classes. We also initialised these pointers to NULL.
4. **mm_malloc**: The only change is to make sure that the allocated block is inserted into the appropriate list. We call find_free_block to find appropriate free block from segregated free lists. We have modified the implementation of place function slightly.

   find_free_block(): As the name suggests, this function finds appropriate free block. We have used first fit approach to find free block. We start our search ignoring the lists where we know that a large enough block will not be found and run through the subsequent lists. Care has to be taken that the allocated block is removed from the appropriate list.

   place(): In this function we remove the block, that is to be allocated, from the free list. If the block is sufficiently larger than the required memory, we split the block and put the smaller free extra block back into the appropriate free list.

5. **mm_free, coalesce** : These are the functions where the state of the blocks will change. We needed to make sure that a block when allocated is removed from the free list and is inserted into a proper list on being freed. mm_free is same but we have slightly changed coalesce. We delete the appropriate free block (next or previous or both or none), according to the conditions, from the segregated free list. Merge it with current block and then put back the larger block in the correct list.

   insert_block_in_seg_list: This function inserts the block in segregated free list according to LIFO implementation. We also tried insertion by ordering blocks according to their addresses (address ordered implementation). But if when we ran the programme with the address ordered implementation, the utilisation actually reduced by 2 (changed from 31 to 29). Hence we haven't used this implementation.

   delete_block_from_seg_list: Helper function to delete given block from appropriate free list. A block may have to be deleted if it has been allocated or has been coalesced into a larger block.

   find_list_class: It returns the appropriate list-class from a predefined number of list-classes when the size of the block is given

Documentation of checkheap and other heap checker functions:

We used these functions to check if heap consistency was being maintained. The function had to be written keeping the ‘segregated list’ design in mind.

1. Is every block in the free list marked as free?

   We used a loop to run through every list. In each list we verified if no block has been marked as allocated.

2. Are all sizes valid?

   In the same loop, we verified if invalid sizes for eg negative sizes have not been packed, or if any of the blocks is packed with size 0, which is the size of the epilogue.

3. Since we have used immediate coalescing, the adjacent free blocks are immediately coalesced. In that case, at any instant we should not find such blocks in our heap. It is a necessary condition to check if our coalescing function is correct or not. We simply run through each list and check that neighbouring blocks of a free block are not marked as free.
4. Is every free block actually in the free list?

   To check this, we run through the entire heap. If we find a free block, we note its address. Then we search for this address in the appropriate free list. If everything is fine, some pointer in the free list must equal this address.

5. Do the pointers in a heap block point to valid heap addresses?

   Note that only the pointers of the segregated free list and free blocks contain pointers to other blocks in the heap. So we run through the heap, searching for free blocks. We examine the pointers pointing to this blocks neighbours. A pointer is valid if it is pointing to a block within the heap i.e. it is between the start and the end of the heap. So we simply put a condition that both pointers must be >=heap_start and <=heap_end.

The above work is done by these checker functions:

static int checkblock(void *bp)	basic checker which was already implemented.
static int checkheap();	
static void printblock(void *bp);	visualize the block for debugging.
static int check_seg_list_ptrs();	are all pointers in segregated list valid?
static int check_size();	do all blocks have a positive size?
static int check_free_block_in_heap();	are all free blocks actually present in the free lists?