OS
Lab 2: Memory Allocator

Authors: Arieta Shabani, Mirjeta Shabani

October 2018

1 Implemented Features

- Fast pools
- Standard pool
- Displaying State of the Heap
- Placement policies: First Fit and Next Fit
- Alignment: with 2, 4, 8, 16, 32, 64
- Safety Checks: memory leak warning, ignore call to free if given block is already free.

2 Limitations of our solution

In our solution we do not ensure that the metadata for each block is not corrupted. When printing the standard pool we do not show it's state based on the sizes of the block therefore one cannot know the block sizes when displaying the standard pool. However, we have implemented an additional function for a more informative view of the free blocks in the standard pool. This function is better explained in section 6 of this report.

3 Fast Pools

All features regarding fast pools have been implemented and all tests have passed.
Q1. Why a LIFO policy for free block allocation is interesting in the context of fast pools?

Since we use a singly linked list to keep the free blocks, using a LIFO policy will make sure that we will always have the free list pointer at the beginning of the list. When allocating we always give blocks from the beginning of the list, moving the pointer of the list to the next free block. When freeing we always insert the freed block in the beginning of the list, thus the last freed block will be given when an allocate request is preformed maintaining a LIFO policy. Hence, no traversing of free list is needed when allocating of freeing blocks.

4 Allocating in standard pool

The difference in these two policies is how we start the free list traversal when a request to allocate is invoked. Thus, depending on the chosen policy, when a request to allocate is invoked we either call the *get_first_fit* or the *get_next_fit* to find the block in which to allocate the memory.

4.1 First Fit

In first fit we always start from the beginning of the list and return the first free block we find that fits the requested size.

4.2 Next Fit

In next fit we keep a pivot (a pointer to a free block) which points to the next of the last allocated block. So, next time an allocation is started the search for the free block starts from the pivot. To ensure that the pivot points to a valid block, after coalescing we always check if we need to update the pivot pointer using the function *update_pointer* and if so we update its address.

4.3 Comparison of Policies

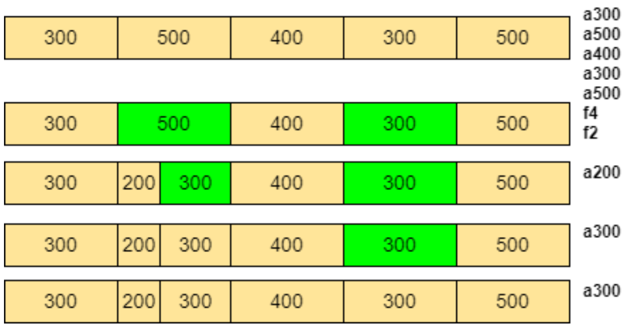


Figure: 1 First Fit Scenario

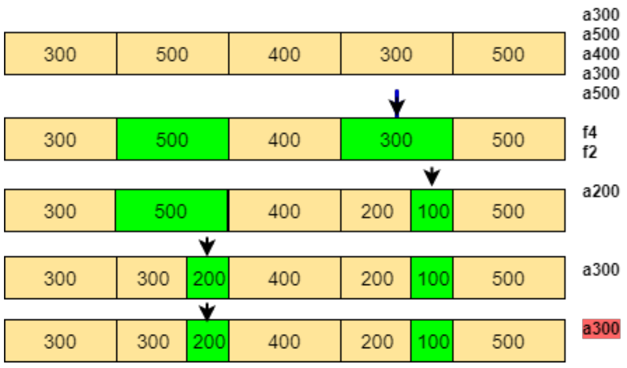


Figure: 2 Next Fit Scenario

Figure 1 and Figure 2 show the same scenario, a sequence of allocate and free request, where First Fit manages to fulfill all the requests, whereas Next Fit fails in the last allocation request. The arrow indicates the pivot used to start searching in Next Fit.

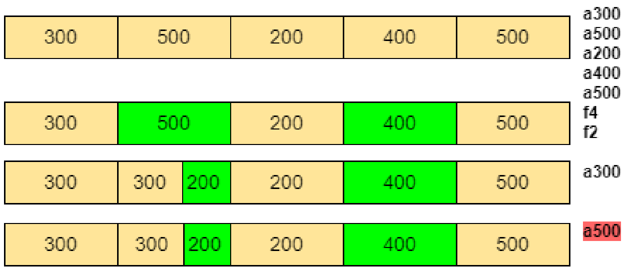


Figure: 3 First Fit Scenario

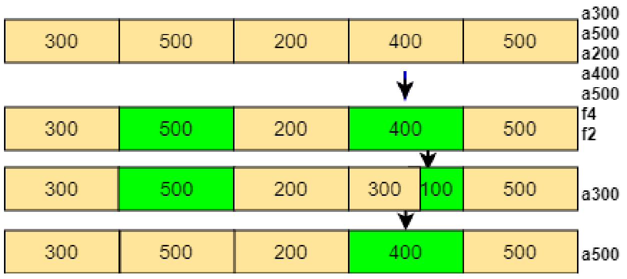


Figure: 4 Next Fit Scenario

Figure 3 and Figure 4 show another scenario, a sequence of allocate and free request, where Next Fit manages to fulfill all the requests, whereas First Fit fails in the last allocation request.

5 Freeing a block in standard pool

Q1. Describe the main principles of the algorithm you use to insert a freed block into the free list, and to apply coalescing if need be.

To keep the linked list sorted by addresses, whenever a call to free is made we traverse the list of free blocks until we find a block with a greater address than the one to be freed and return the address of that block so we know that the block to be freed will be inserted just before this. This method is *find_address_to_insert_free_block*.

Coalescing is done immediately upon requesting to free a block. To coalesce given the address of the block to be freed we extract the footer of the previous block and the header of the next one by moving the given pointer. This is done in order to check if those blocks are free. If either of them is free we coalesce with it or with both in case both are free, we update the header and footer of the block and return the updated pointer. In our implementation this function is *coalesce*. The returned pointer will be used to check where we need to update the previous and next pointers of the free list once the block has been freed and coalesced if possible.

6 Printing memory state

Fast Pools To print the state of the fast pools we have assigned the symbol 'X' for allocated and '-' for free. One such symbol represents one block in the specific pool. So, for the first fast pool a symbol represents the maximum requested size (64 bytes). For the second fast pool a symbol represents a block of size 256 bytes, while for the third fast pool a symbol represents a block of size 1024 bytes. Name of the function in the code: *print_fast_pool*

Standard Pool To print the state of the standard pool we will put the symbol '-' for the free blocks and 'X' for the allocated ones. Since the size of the block is not fixed, a symbol may represent different number of bytes. Initially, when no blocks have been allocated the standard pool will contain only one block and the printing function would show one dash only: '-' meaning there is one free block in the standard pool. When allocating one block the standard pool would look like 'X-', showing an allocated block at the start and a free one after it (the rest of the free pool). Name of the function in the code: *print_standard_pool*.

Examples of the standard pool printing scenarios:
'XXX-' : 3 consecutive allocated blocks and one free block after
'X-XX-' : 1 allocated then 1 free then 2 allocated then 1 free block

We have implemented another method *print_free* which prints detailed information about the state of the free list in the standard pool, including the start address of block, block sizes, end address and if the block is actually free. We have utilized this function to test our implementation and through our tests we have not found any bugs in this part. To invoke this method press 's' while running mem_shell interactively.

7 Alignment

To ensure appropriate alignment the MEM_ALIGNMENT value is used to set the header size and in each allocation request the requested size is resized in order to be a multiple of MEM_ALIGNMENT. Specifically, for each requested size that is not a multiple of MEM_ALIGNMENT value we add the least needed size to reach a multiple value. Thus, this results in internal fragmentation which is a trade-of with aligned addresses. The padding of block sizes together with the header size ensures that all addresses are multiples of MEM_ALIGNMENT.

```
size_t modulo = size % MEM_ALIGNMENT;
    blueif(modulo != 0){
        size = size - modulo + MEM_ALIGNMENT;
    }
```

Considering that the provided solution follows a generic approach by relying the header size and the block size in multiples of MEM_ALIGNMENT, it supports alignments of all required values (2, 4, 8, 16, 32 and 64).

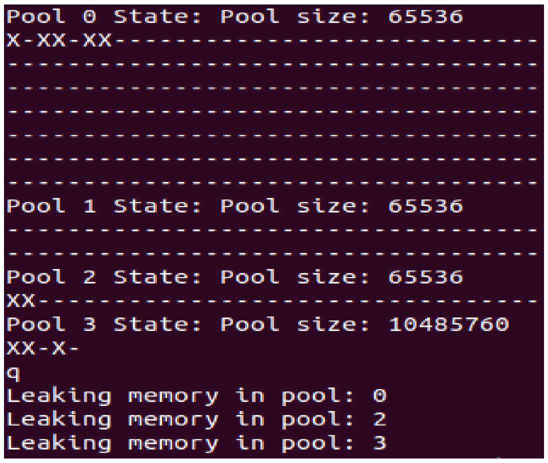
8 Tests

Upon implementing all the above mentioned features we have thoroughly tested them using the provided tests as well as the mem_shell program. Our solution passes all the tests (alloc1 - alloc8) provided in all alignment values (1, 2, 4, 8, 16, 32, 64) as well as in both policies (First Fit and Next Fit).

9 Safety Checks

9.1 Forgetting to free memory (Memory Leaks)

We have implemented the warning message when there are memory leaks upon exiting the program. To test our implementation we run mem_shell and request a series of allocations and do not free them. When we quit the program we print the pools that have memory leaks. One such case is represented in the image below where one can see the state of the heap before quitting the program and see that in pool 0 we have 5X symbols (allocated blocks), pool 1 has no allocated blocks, pool 2 has 2 allocated blocks and pool 3 (standard pool) has 3 allocated blocks. Thus, the heap has memory leaks in pools 0, 2, and 3.



```
Pool 0 State: Pool size: 65536
X-XX-XX----------------------------
-----------------------------------
-----------------------------------
-----------------------------------
-----------------------------------
-----------------------------------
-----------------------------------
Pool 1 State: Pool size: 65536
-----------------------------------
-----------------------------------
Pool 2 State: Pool size: 65536
XX---------------------------------
Pool 3 State: Pool size: 10485760
XX-X-
q
Leaking memory in pool: 0
Leaking memory in pool: 2
Leaking memory in pool: 3
```

Figure: 5. Safety check: Memory Leaks

9.2 Calling *free()* incorrectly

For this safety check we have implemented only the part where we ignore a request to free on an already freed block. This is the first check that is done in the beginning of *mem_free_standard_pool* function. This is done by checking if the header at that address is free or not. If it is a free block address it simply ignores the requests and does not exit.

10 Submitted files

On the archive submitted there are two folders provided:
src: contains the basic version of the solution,
src_alignment: contains the advanced version of the solution For a better understanding of the code please refer to the comments added mainly in the advanced version of the code.

*** The code provided is written by people who have never coded in C before, so please accept our apologies for any novice mistakes***