Mehmet Furkan Sahin
21201385

Zubeyr Furkan Eryilmaz
21202676

Project 4 - Section 2
Ibrahim Korpeoglu

REPORT

- The project is implemented by following the First Fit algorithm.
- Mapping is done by adding 16 bytes long struct to the head of each allocated segment.
- This struct keeps two important data;
  - One indicates the length of the segment that it is kept, (integer is 8 bytes in 64 bits systems)
  - The other one keeps the address of the next segment. (addresses are 8 bytes in 64 bits systems)
- One additional struct is added to the head of the whole memory block when first it is created to be used for mapping. This struct has two attributes and one is NULL (address of the next allocated block) the other one is 0 (length of the current allocated block), initially. By the help of this struct we understand that there is no allocation made, yet.
- Since we followed first fit, we are simply using linear search for an appropriate place. Thus, there might happen some performance differences for the same set that will be allocated according to the order that we allocate them. (Scenario 1)
- To be able to protect the mapping structure through the whole memory block, we are following a linear search for s_free method too.Thus, again, there might be some performance differences for the same set that will be freed according to the order that we free them. (Scenario 2)
- This library is open for some external fragmentation cases. For example, if there are 512 bytes of free space in total but they are spread among the memory between different allocated segments by 64 bytes per free space. We cannot even allocate 49 bytes, since

we need 16 bytes more to keep the mapping info (this makes the needed space 65 bytes) and we have at most 64 bytes as a whole. (Scenario 3)

SCENARIO 1

- We created 1 MB of segment.
- Then we made allocations in a format as following;
  - **100** - 50 - **100** - 150 - **100** - 250 - **100** - 350 - **100** - 450 - **100** - …
- Thus for each segment starting from index 0,
  - index of the segment % 2 == 0 => segment size = 100
  - other wise segment size = index * 50.
- Then, we freed the segments with odd index numbers to create space to make new allocations.
- Now, we have specific locations for allocation requests with different sizes. For example, allocation request with 350 bytes cannot fit in to first 3 free blocks.
- Now we applied two different orders of requests to fill in the segment with the same sets of requests.
  - First we made requests with 50, 150, 250, 350, 450, … in ascending order.
  - Then we made requests with the same set but in descending order.
- We measured the time it takes for both request sets and it can be seen in the figure 1.

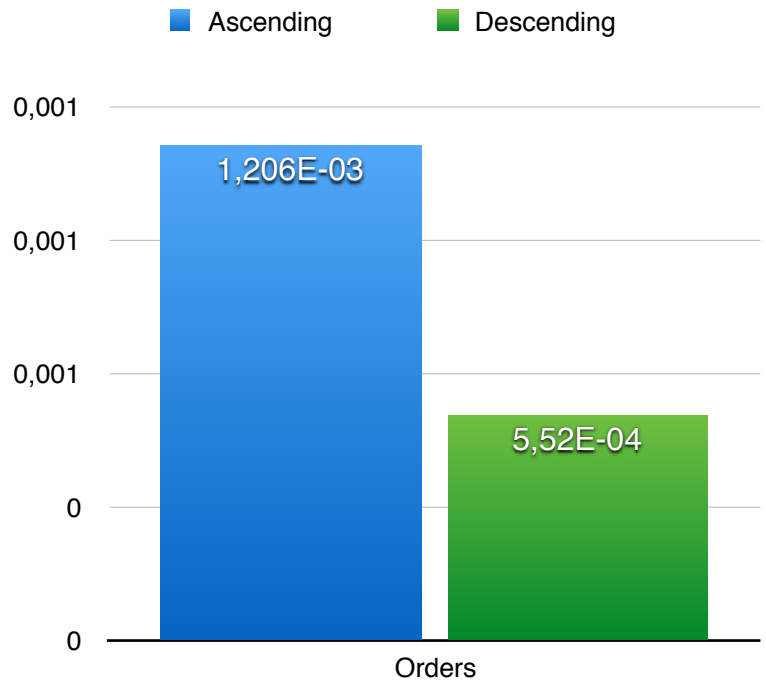

Figure 1: Ascending vs. Descending allocation

Result: As we can see from the chart, ascending approximately doubles the descending.

The reason of this is the following;

- For ascending,
  - algorithm finds the first place and links it to the next of one previous allocated segment,
  - for the second allocation request, it again starts from the head and goes to the next space by going through newly allocated segment for the first request
  - For the third allocation requests, it again starts from the head and goes to the next space by going through newly allocated segment for the first request and second request
  - …
  - Thus, How much we make request, the way to find the next free space stretches.
- For descending, since we do not go through newly allocated segments more than once, the time decreases radically.
- If we were keeping the data for free blocks and go through them in case of allocation, the result would be the reversed of the above.

SCENARIO 2

- We created 1 MB of segment.
- Then we made allocations in a format as following;
  - **100** - 50 - **100** - 150 - **100** - 250 - **100** - 350 - **100** - 450 - **100** - …
- Thus for each segment starting from index 0,
  - index of the segment % 2 == 0 => segment size = 100
  - other wise segment size = index * 50.
- Then, we freed segments starting from the first block linearly and measured the time it takes.
- After that, we filled in the segment again by following the same filling algorithm above.
- Then, we freed segments starting from the last block linearly and measured the time it takes.
- The result can be seen from the Figure 2.

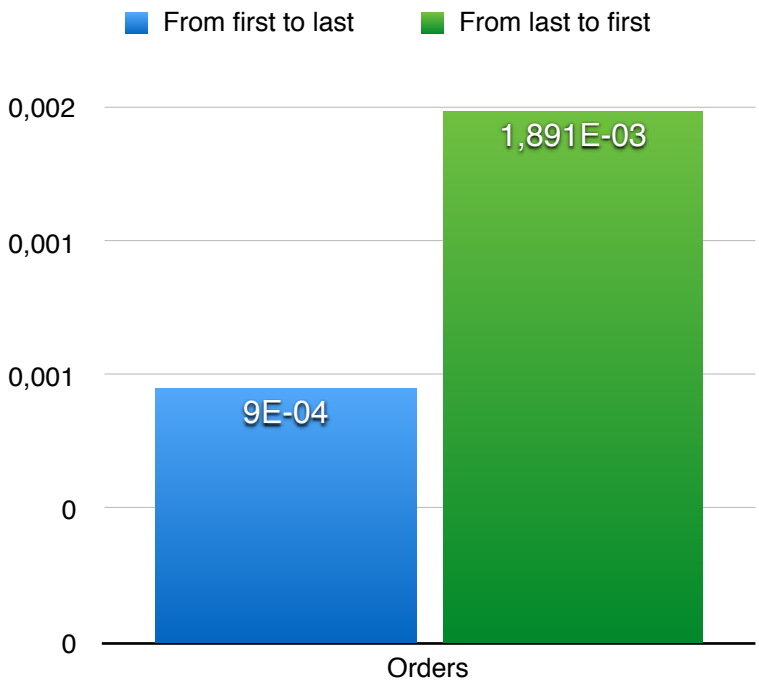

Figure 2: Order effect on free operation

- The reason is very similar to the Scenario 1.

- Because we keep the information for allocated segments and simply jump over the free parts.
- Thus, when we start to free from the first segment, we simply shorten the way that we should follow for the next free request.
- However, when we start to free form the last segment, we go through the segments that we will free in the future anyways.