Mehmet Furkan Şahin 21201385
Zübeyr Furkan Eryılmaz 21202676

Project 4 - Section 2
Ibrahim Körpeoğlu

# 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

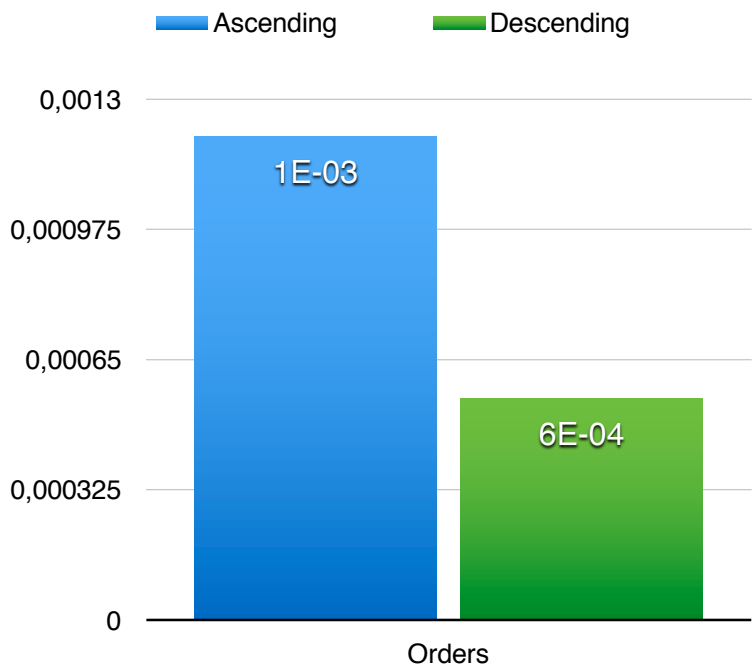
we need 16 bytes more to keep the mapping info (this makes the needed space 65 bytes) and we have at most 64 bytes as a whole. (Scenario 3)

## 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 allocation  starting from index 0,
  - index of the allocation % 2 == 0 => allocation size = 100
  - other wise allocation size = index * 50.
- Then, we freed the allocations 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 allocations,
  - for the second allocation request, it again starts from the head and goes to the next space by going through newly made allocation 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 made allocation 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 made allocation 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 allocation starting from index 0,
  - index of the allocation % 2 == 0 => allocation size = 100
  - other wise allocation size = index * 50.
- Then, we freed allocations starting from the first block linearly and measured the time it takes.
- After that, we filled in the allocation again by following the same filling algorithm above.
- Then, we freed allocations 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 allocations and simply jump over the free parts.
- Thus, when we start to free from the first allocation, we simply shorten the way that we should follow for the next free requests.
- However, when we start to free form the last allocation, we go through the allocations that we will free in the future anyways.

**SCENARIO 3**

- We created 64KB of segment.
- Then we made allocations in a format as following;
- **128B** – 128B – **128B** – 128B – **128B** – 128B – **128B** – …. – **256B**
- Namely, we filled the segment with 455 units 128B allocations, then we freed the last two allocation and made a 256B allocation.
- Since for each allocation there is a 16B structure that holds next allocation pointer and allocation size, for each 128B allocation it takes 144B from our memory and for 256B allocation it takes 272B.
- Since our segment is 64*1024B = 65536B, total allocated part is 453*144B + 272B + 16B = 65520B and minimum request is 64B, we encountered with external fragmentation. The unusable memory size here is 65536B – 65520B = 16B

```
Alloc = from:0x1ff31f0   to:0x1ff3270    size:144
Alloc = from:0x1ff3280   to:0x1ff3300    size:144
Alloc = from:0x1ff3310   to:0x1ff3390    size:144
Alloc = from:0x1ff33a0   to:0x1ff3420    size:144
Alloc = from:0x1ff3430   to:0x1ff34b0    size:144
Alloc = from:0x1ff34c0   to:0x1ff3540    size:144
Alloc = from:0x1ff3550   to:0x1ff35d0    size:144
Alloc = from:0x1ff35e0   to:0x1ff3660    size:144
Alloc = from:0x1ff3670   to:0x1ff36f0    size:144
Alloc = from:0x1ff3700   to:0x1ff3780    size:144
Alloc = from:0x1ff3790   to:0x1ff3810    size:144
Alloc = from:0x1ff3820   to:0x1ff38a0    size:144
Alloc = from:0x1ff38b0   to:0x1ff3930    size:144
Alloc = from:0x1ff3940   to:0x1ff39c0    size:144
Alloc = from:0x1ff39d0   to:0x1ff3a50    size:144
Alloc = from:0x1ff3a60   to:0x1ff3ae0    size:144
Alloc = from:0x1ff3af0   to:0x1ff3b70    size:144
Alloc = from:0x1ff3b80   to:0x1ff3c00    size:144
Alloc = from:0x1ff3c10   to:0x1ff3c90    size:144
Alloc = from:0x1ff3ca0   to:0x1ff3d20    size:144
Alloc = from:0x1ff3d30   to:0x1ff3db0    size:144
Alloc = from:0x1ff3dc0   to:0x1ff3e40    size:144
Alloc = from:0x1ff3e50   to:0x1ff3ed0    size:144
Alloc = from:0x1ff3ee0   to:0x1ff3fe0    size:272
Free = from:0x1ff3ff0    to:0x1ff4000    size:16
```

**Figure 3: External Fragmentation**