Lab 2: Memory Allocator
Group Report

Mariia Zameshina, Do Son Tung

1 Implemented features

1. Fast pools
2. Standard pools
3. Print function for state of memory in fast pools and standard pools
4. Next fit policy
5. Dealing with alignment constrains : in full version (for all powers of 2)
6. Safety checks: freeing all the memory at the end of the pool usage

2 A short description of your main design choice

2.1 The fast pools

2.2 The standard pool (first version)

In case of the pool P_3 when we have to allocate a new block and space which is left from free space after allocating a block is not enough to form a new free block, we increase the size of block we want to allocate on size which is left from free block. (More details on fig. 1)

For numeration of free blocks in standard pool we consider different cases such as previous block is NULL or not NULL, next block is NULL or not NULL.

2.3 Displaying the state of the heap

For printing a memory state of fast pools we use a flag array initialized as "all blocks are allocated" and update it according to information in free list.

For printing a memory state of the standard pool we use information stored in headers and footers and move through the pool. (because in this case it would be take too much memory to store an array of flags).

In order to improve a visualization of the memory state and use it for debugging the program we use new symbols :

1. [– shows beginning of a block
2.] – shows end of a block
3. ‖ – shows border between header/footer and rest of the memory inside block

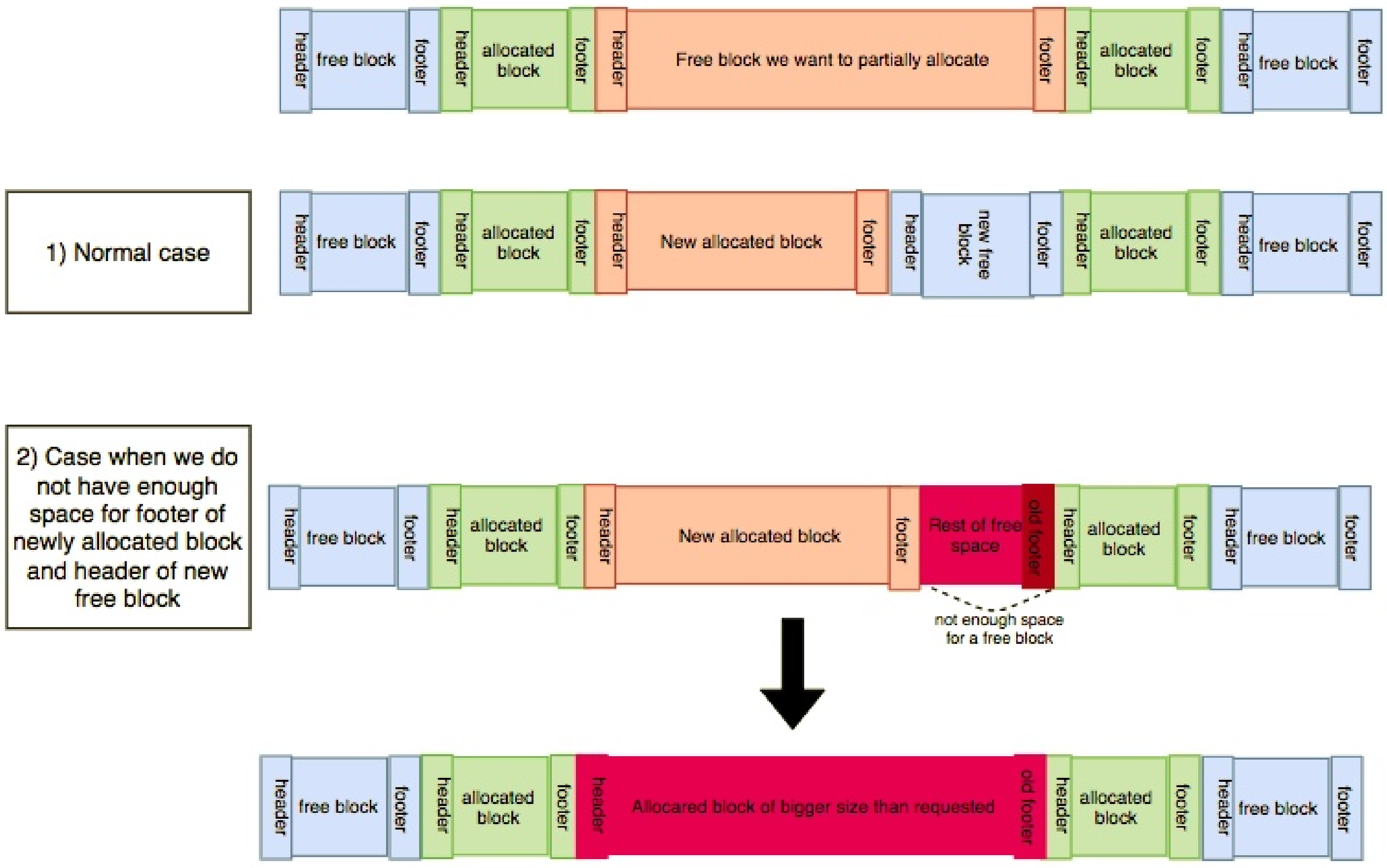

header
free block
footer
header
allocated block
footer
header
Free block we want to partially allocate
footer
header
allocated block
footer
header
free block
footer
1) Normal case
New allocated block
new free block
2) Case when we do not have enough space for footer of newly allocated block and header of new free block
Rest of free space
old footer
not enough space for a free block
Allocared block of bigger size than requested

Fig. 1: *Allocation of a block in case if we do not have enough memory to split a block on new allocated and new free one*

2.4 Placement policies for the standard pool

2.5 Alignment constraints

We used the following idea: we changed size of both all the headers and footers and size of the memory we allocate such that it gives residue 0 by mod of $MEM\_ALIGNMENT$.

So, for that purpose, if user requests memory size S, for which $S\ \%\ MEM\_ALIGNMENT = residue \neq 0$ we will allocate memory with size $S + (MEM\_ALIGNMENT - residue)$, so that new size of block is divisible by $MEM\_ALIGNMENT$. Else, it is already divisible by $MEM\_ALIGNMENT$ and we do not have to change size of a block.

For header and footer if $MEM\_ALIGNMENT > 8$ we add some empty space so that size of header and footer are divisible by $MEM\_ALIGNMENT$.

2.6 Safety checks

In our implementation in order to test that after making a safety check and freeing all the used memory we definitely get a pool with all the arrays freed we print our pool state.

3 Questions in the assignment

3.1 The fast pools

Questions

1. Metadata is data about available free blocks and we will store it in a free list which is usually stored in the free space.

2. For the pools P_0, P_1, P_2 size of blocks is defined. So even if we allocate block of the smaller size it still will be stored in same memory space. (For the fisrt pool it is 64 bytes, for the second it is 256 bytes, for the third one it is 1024 bytes) For the pool P_3 minimum size of a free block and allocated block is 1025 bytes.

3. No, we do not have to keep a list of allocated blocks because we have a list of free blocks and we will not allocate a block which has been allocated.

4. Metadata of a free block in pools P_0, P_1, P_2 should contain a link to the next free block, and in block P_3 is should contain a link to the next and previous free blocks.
   For allocated block it should contain only the size of this block to be sure that we do not use memory we did not ask to allocate.

5. For a block in pools P_0, P_1, P_2 it should be inserted at the first position in the single linked list because we do not care about a position of free blocks since they all have the same size. For a block in pool P_3 we should search for its' position in the linked list and have the previous free block point to the recently freed block and the recently freed block points to the next free block.

Q1 Using the LIFO policy for free block is good in the context of fast pools because you can ignore completely the external fragmentation and because it's fast so we don't have to searching often for free block, we can give them all the same size for each request. And we don't need to search where to put the recently freed block in the free list.

3.2 The standard pool (first version)

Questions

1. For the pool P_4 size for header is 64 bits, for footer 64 bits, plus 1025 bytes (minimum size of memory we can try to allocate in this block). So in total, $1025 + 8 + 8 = 1041$ byte.

2. No, because we already have a list of free blocks, so that we know what memory can not be used and also for each allocated block we have its' address in memory and its' size (which is stored in header and footer).

3. Address of the first byte in block which can be use (not metadata stored in header and footer).

4. Address of the first byte in footer, because we need to free full block.

5. Newly freed block should be inserted between blocks which are stored in memory before and after it and also should be coalesced with them if they (or one of them) are free.

6. We should split the block and leave the remaining part to free list.

7. Yes, we can have issues with allocating a new block. To solve this issue, we should move existing allocated blocks and free the place for a new block.

Q2 We need to find a place for the newly freed block, and there are two possibilities:

- **Left or right neighbour block of newly freed block is free :**
  To check that this it the case, we will read a footer of block before our block and a header of a block after ours. If it is the case, we will merge our newly freed block with left or right or both free blocks. More details on this are shown on fig. 2

- **Both of the previous and next blocks are allocated:** In this case, after checking that it is true, we will take a free list and go through it in order to find last block which has address of first bit smaller than index of our newly freed block and first block which has address bigger than address of newly freed block. Like this, we will get the place where our newly freed block should be stored. More details on this are shown on fig.3

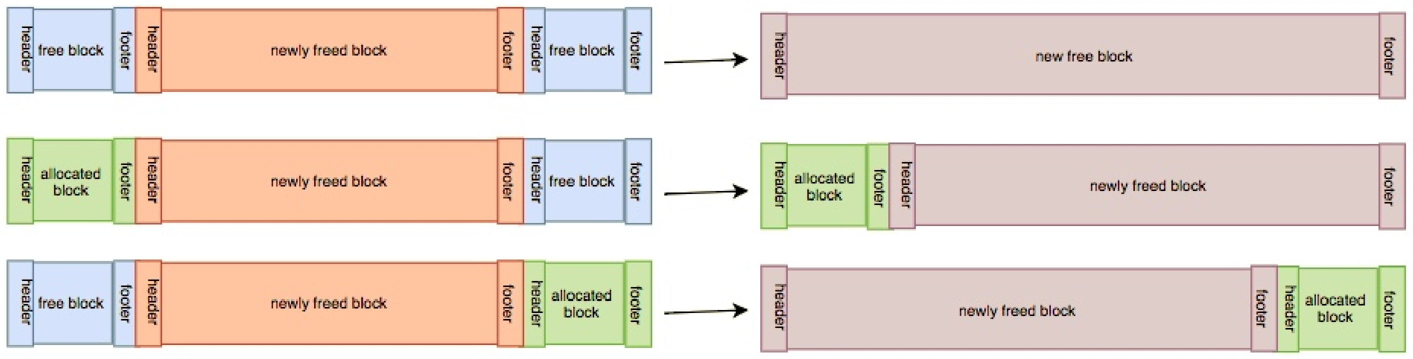


Fig. 2: *Different possibilities for newly freed block which occure in case when left ot right neighbour of newly freed block are free*

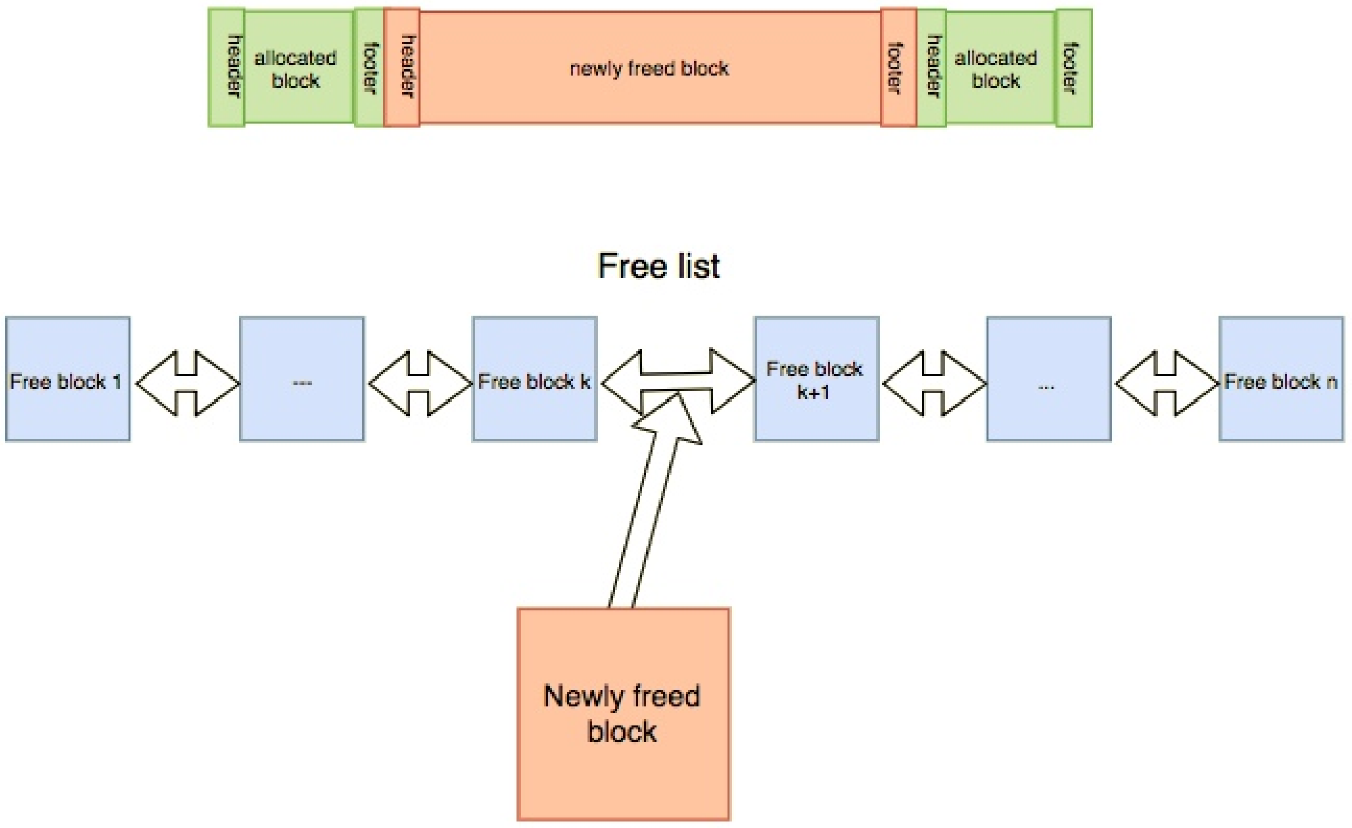


Fig. 3: *Case when both left and right neighbours of newly freed block are allocated*

This solution is good in the case if we have free blocks around newly freed block (which is usually the case) and in this case time our program will need to put this block into a free list is $O(1)$. But in case if we have many allocated blocks and both neighbour blocks are allocated we will have to go through the free list which will work at O(n) time where n is number of free blocks.

3.3 Displaying the state of the heap

→ Described in section "Implemented features".

3.4 Placement policies for the standard pool

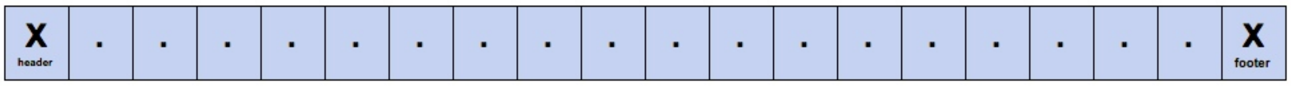
(a) Implemented.

(b) **Comparison between "first fit" policy and "next fit" policy:**
Next fit policy allows us to start allocating memory at the point where we found space for last process (and we are more likely to find space for another one there too). For example, if we have 10 allocated processes in the beginning, we will not go through them over and over again in order to find a free space.
Examples:
Lets assume that standard pools size is just 20 bytes. Then we have the following sequence of allocations and frees.

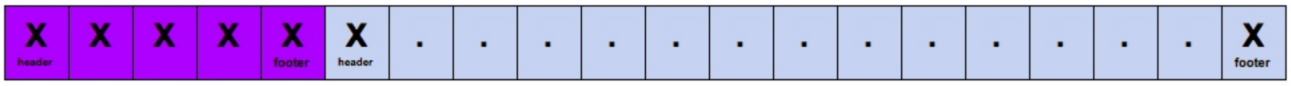
1. allocate a block with size 3 bytes
2. allocate a block with size 3 bytes
3. free the first block allocate a block with size 5 bytes
4. allocate a block with size 1 bytes
5. allocate a block with size 3 bytes

For the case of the first free policy we will get the following sequence of prints and will not succeed to allocate all the blocks we want:

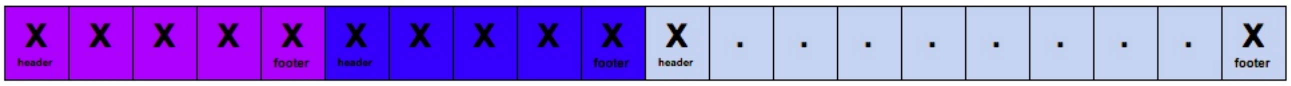
Initial state:



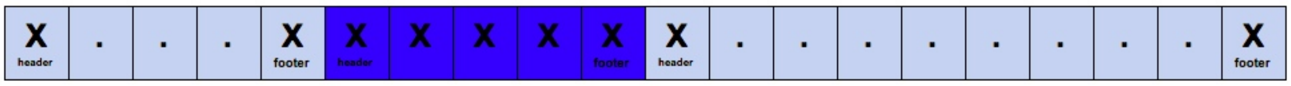
1st step:



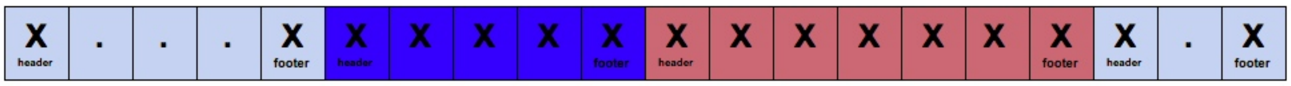
2nd step:



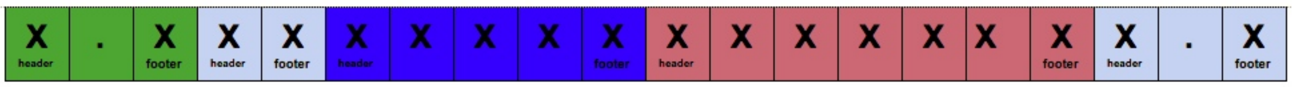
3rd step:



4th step:



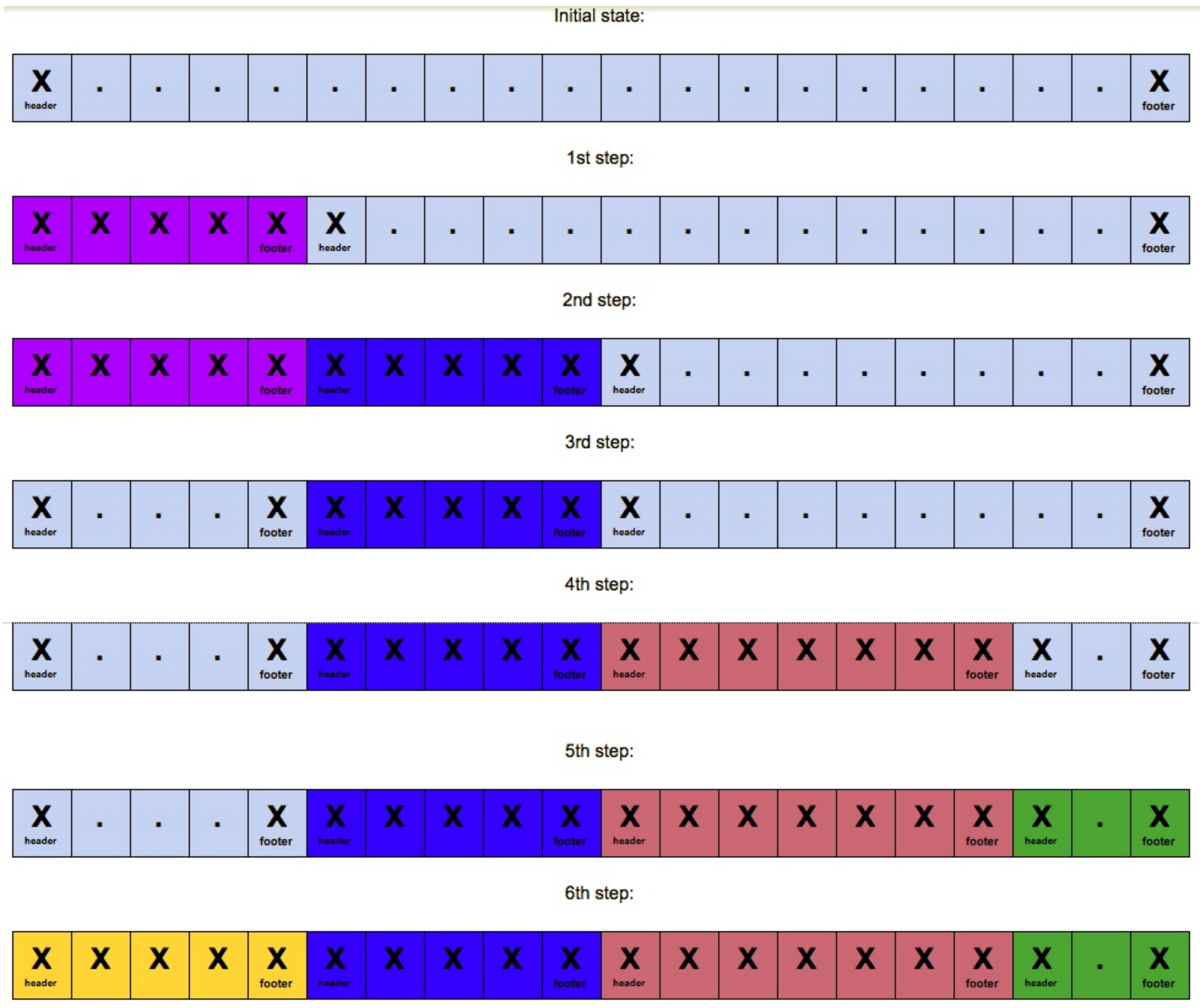
5th step:



5th step:

We do not have space to allocate this block!

For the case of the next free policy we will get the following sequence of prints and will succeed to allocate all the blocks we want:



We succeed to allocate all the blocks we wanted.

3.5 Alignment constraints

3.6 Safety checks