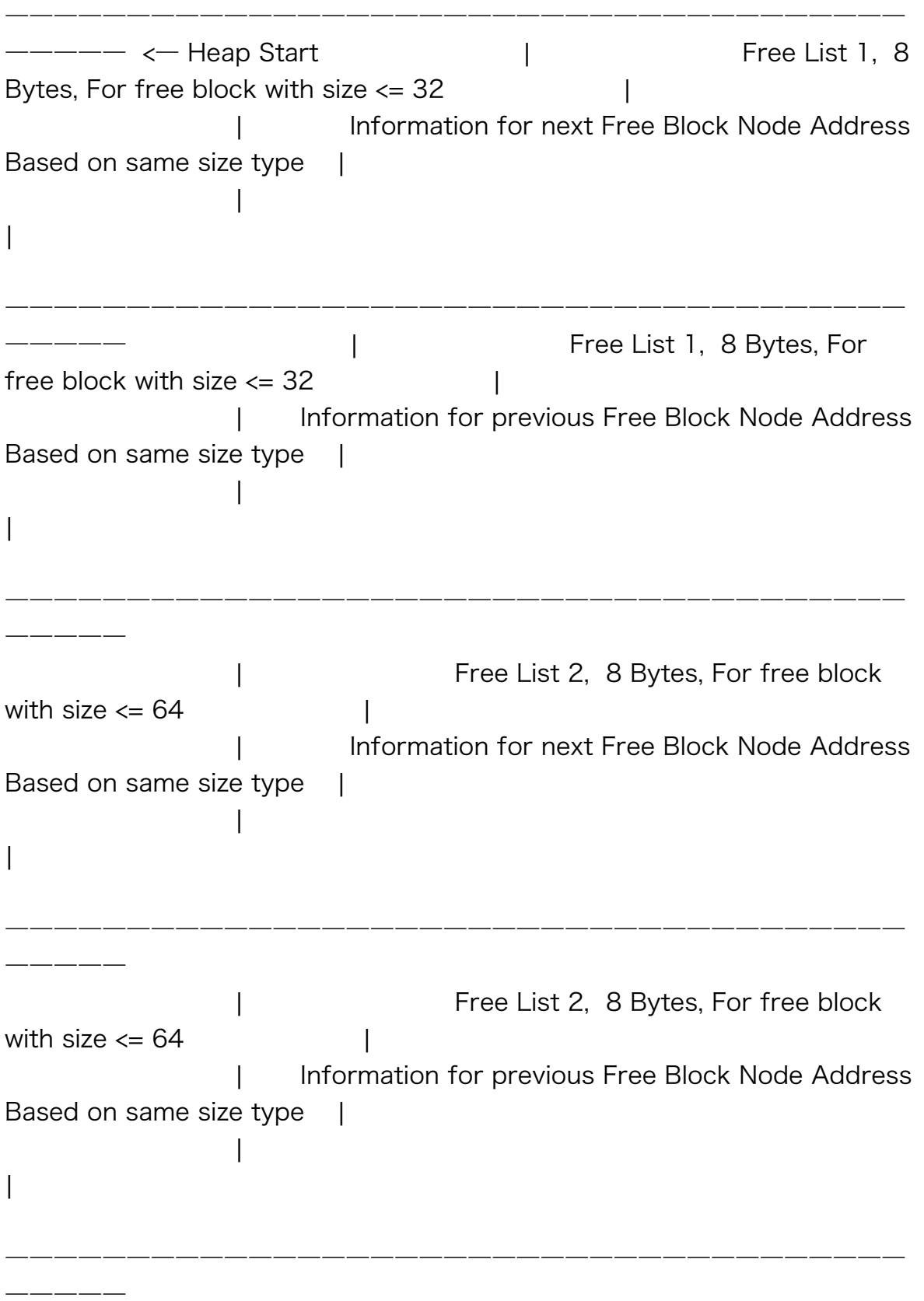
1. Heap structure



```
                        |                                                   ...
|
                        |
|
                        |
|

————————————————————————————————————————————————————————————
                        |                    Free List 11,  8 Bytes, For free block
with size >16384                |
                        |
|
                        |
|

————————————————————————————————————————————————————————————
———————— <— Prologue Block header
                        |                                        Prologue Block Header
|   In Binary: 000…0001001(64 bits)
                        |                                         8 Bytes Size, Size = 8
|
                        |                    Allocation Status: 1 Previous Block
Allocation Status: 1                |

————————————————————————————————————————————————————————————
———————— <- Prologue Block Footer
                        |                                        Prologue Block Footer
|   In Binary: 000…0001000(64 bits)
                        |                                         8 Bytes Size, Size = 8
|
                        |
|

————————————————————————————————————————————————————————————
———————— <— Example Allocated Block Header
                        |                                             Block 1 Header
|   In Binary: 000…0100011(64 bits)
                        |                                        8 Bytes, Size 16n + 16, n
```

= 1 |
| Allocation Status: 1 Previous Block
Allocation Status: 1 |

———————————————————————————————————————

| n = 1, 8 bytes
|
|
|
|

———————————————————————————————————————

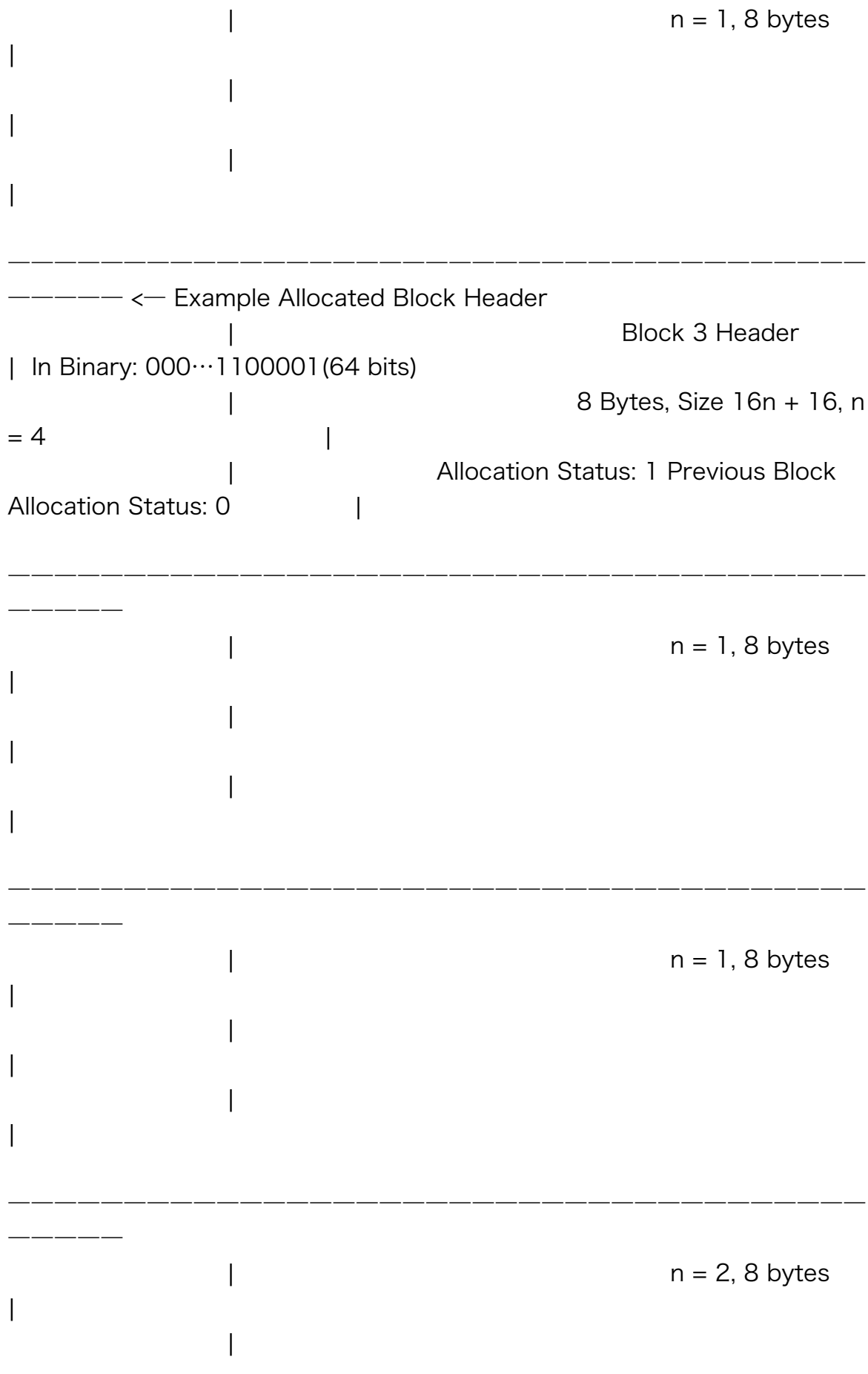
| n = 1, 8 bytes
|
|
|
|

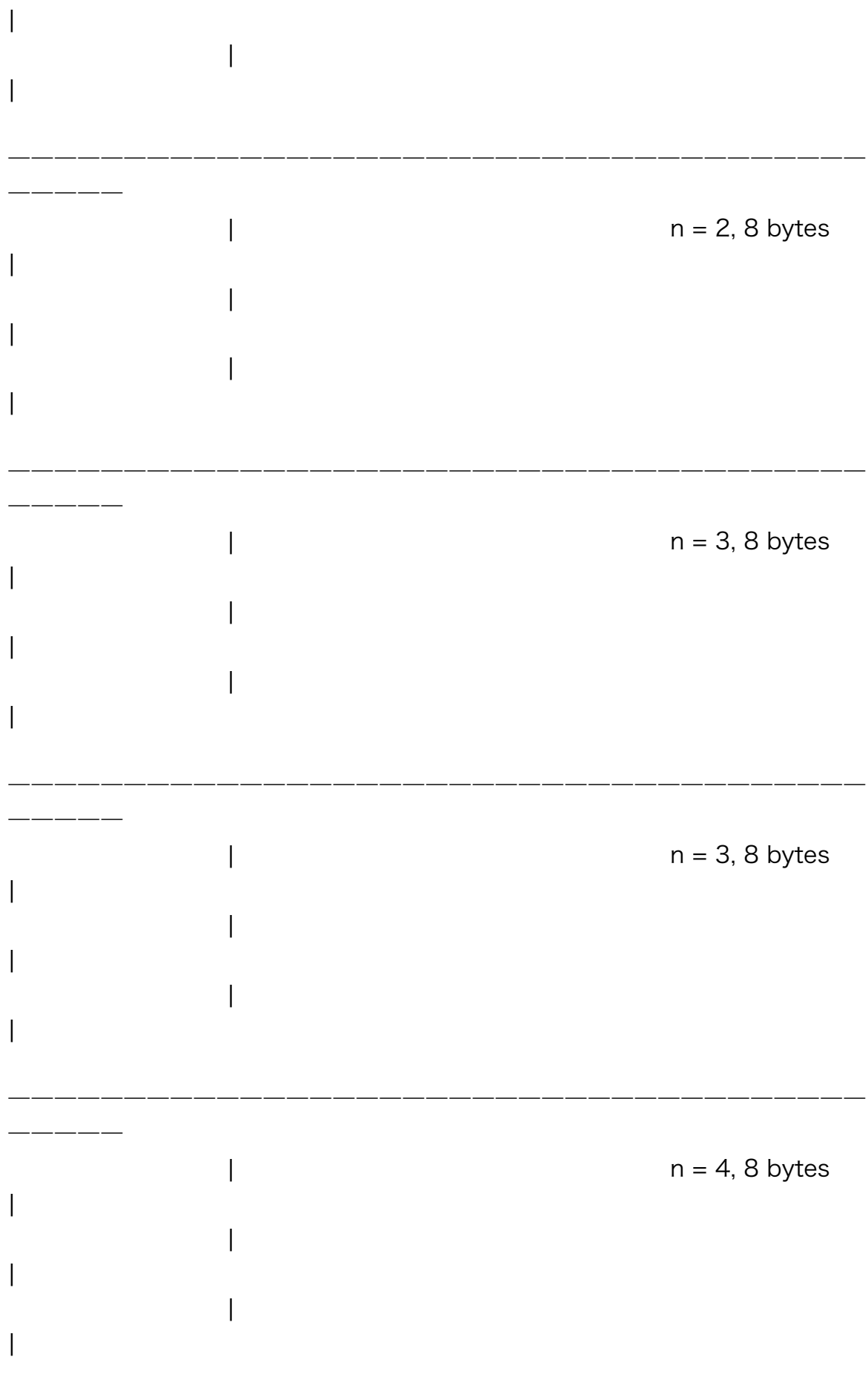
———————————————————————————————————————
——— <— Example Free Block Header
| Block 2 Header
| In Binary: 000···0110010(64 bits)
| 8 Bytes, Size 16n + 16, n
= 2 |
| Allocation Status: 0 Previous Block
Allocation Status: 1 |

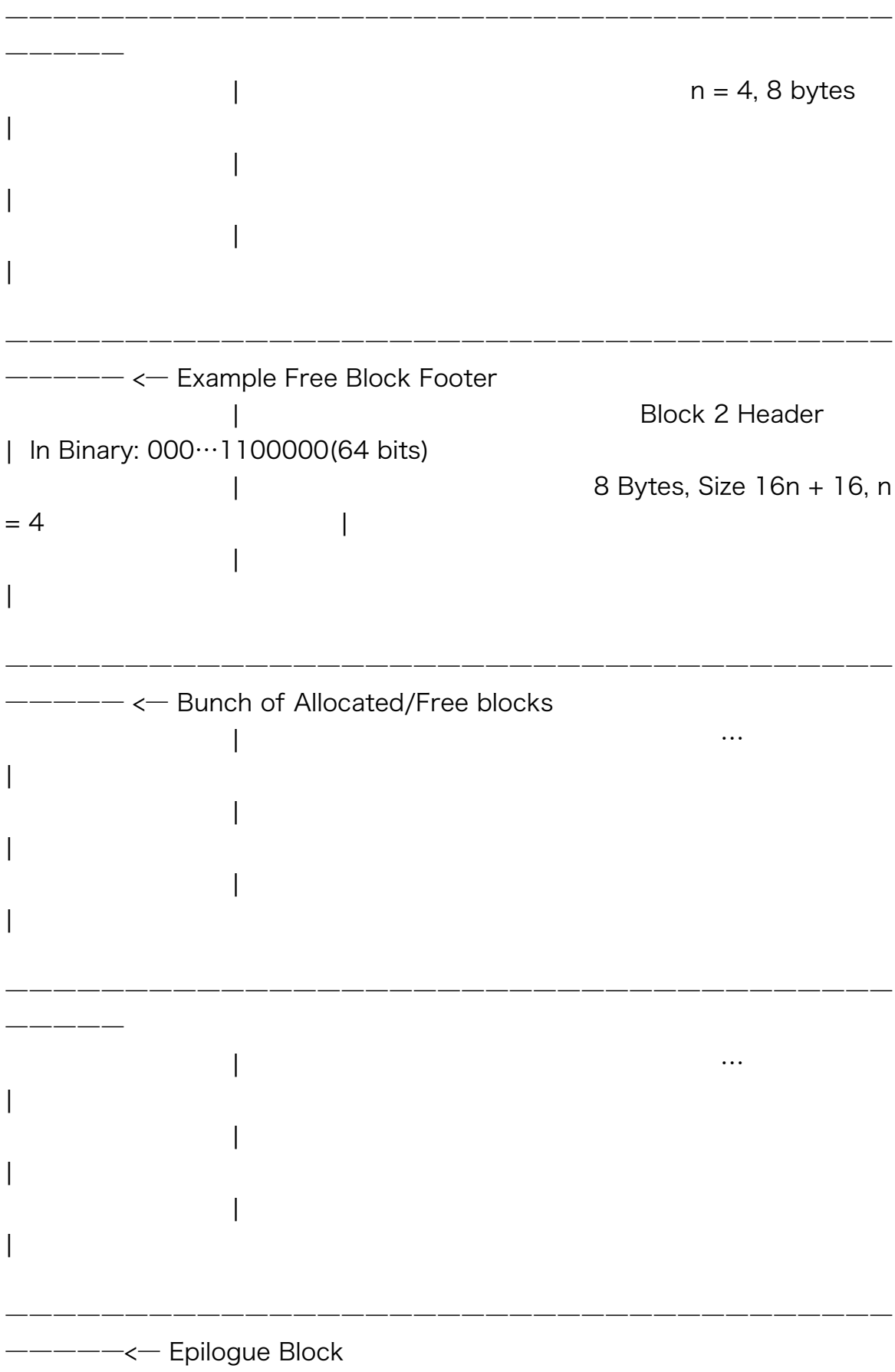
———————————————————————————————————————

| n = 1, 8 bytes
|
| Information for next Free Block Node Address
Based on same size type |
|
|

——
| n = 1, 8 bytes |
| Information for previous Free Block Node Address Based on same size type |
| |
——
| n = 2, 8 bytes |
| |
| |
——
| n = 2, 8 bytes |
| |
| |
—— <— Example Free Block Footer
| Block 2 Header | In Binary: 000…0110010(64 bits)
| 8 Bytes, Size 16n + 16, n = 4 |
| |
——


n = 1, 8 bytes
n = 1, 8 bytes
n = 1, 8 bytes
n = 2, 8 bytes
Block 3 Header
In Binary: 000···1100001(64 bits)
8 Bytes, Size 16n + 16, n = 4
Allocation Status: 1 Previous Block Allocation Status: 0
<— Example Allocated Block Header


n = 2, 8 bytes
n = 3, 8 bytes
n = 3, 8 bytes
n = 4, 8 bytes


n = 4, 8 bytes
Block 2 Header
8 Bytes, Size 16n + 16, n = 4
In Binary: 000…1100000(64 bits)
Example Free Block Footer
Bunch of Allocated/Free blocks
…
…
Epilogue Block

| Epilogue Block
| In Binary: 000···0000001(64 bits)
| 8 Bytes Size, Size = 0
|
| Allocation Status: 1 Previous Block
Allocation Status: 1 or 0 |

———————————————————————————————————————

| Padding
|
| 8 Bytes
|
|
|

———————————————————————————————————————

2. Information of blocks

a. Free list 11 blocks: 16*8 = 88 bytes. Contains information address that points to next and previous free block according to the size type. For example, Free list 1 will only link to free blocks that are less or equal to 32 bytes.

b. Prologue Block: Start of the main allocation blocks. 8 bytes for header and 8 bytes for footer. Contains 64 bit information of size, allocation status, and previous allocation status. Both allocation and previous allocation status will set to 1.

c. Allocated blocks: 8 bytes for header and aligned size in multiple of 16. For example, to malloc a 65 byte block, the aligned size will be 8(Header) + 8*10(Main block) = 88 bytes as multiple of 16 bytes to satisfy the alignment. The allocated block header will contain 64 bit information of size , allocation status, and previous allocation status. Starting from 4th bit will be the size. The allocation status is 1. Depending on the status of previous allocation status, the 2nd LSB bit will be 1 if allocated and 0 if unallocated.

d. Free blocks: 8 bytes of header, 8 bytes for footer, and size by information of allocated block - 16. Since it is being freed from already malloced block, no need to do alignment to 16 bytes anymore. The header of free block will contain 64 bit information of size, allocation status, and previous allocation status. The bit construction is as same as allocated block, but with allocation status of 0. The 2nd and 3rd block will contain next free block address and previous free block address. If this block is 1st free block, then the previous address will be free list address on top of the head in respective size type. Finally, the footer will contain 64 bit information of size only.

e. Epilogue Block: Similar to prologue block, but it lays at the end part of the heap to state the end of dynamic allocation. The block contains 64 bit information of size, allocation status, and previous allocation status. The size will always be 0 to let the program know if reached to epilogue since epilogue is the only block that is allowed to have size 0. The allocation status is always 1, and previous allocation status depends on previous block's status.

3. Managing Free Blocks: Segregated Linked List.

A looping double linked list is used and list is separated by size types from 32, 64, 128, ⋯ , and blocks that are larger than 16384. In the start of the eleven 16 bytes free list information, those are set as the starting point of each respective size type. Then, by adding new free block, the block will be linked with its respective head. In those free blocks, 2nd and 3rd blocks contains the information of previous and next address of the free block so no free block will be missed out. If the free block is the first one in list, then the previous address of the free block should be the address of the list head in one of those eleven 16 bytes blocks. Using a segregated list to manage free blocks can greatly reduce traversing search time when a lot of block is being freed. The first fit strategy is being used to optimize the throughput.

4. Functions

mm_init(): Call mem_sbrk() with size of 16*11(Free List head) + 8(Prologue Block Header) + (Prologue Block Footer) + 8(Epilogue Block) + 8(Empty Padding) = 208 Bytes which is a multiple of 16 to satisfy the alignment requirement.

Then, initialize those Free List Head (22 Words = 11

16Bytes blocks) and set their previous and next value(16 bytes each) to itself to each form a small linked list pointing toward itself.

Next, set prologue block's header and footer in 23rd and 24th words.

Finally set epilogue block in the 25th word and padding in 26th word.

In addition, set the starting address located at the start epilogue's address.

malloc(): 1. Find existing free block in the list by input size. For example, malloc(65) will be looking at 2nd free list and if not found, traverse next list until 11th list. If none of the free block is found, go to 3.

2. If a free block in free list is found, look at the free block's size, decide if there is still empty space after allocation, if yes, then split the free block and put the free block in correct list type. For the allocated block, set the header value to allocated 1 bit. The found free block must be deleted from current position in linked list to serve updating free block's purpose. Then, the updated free block can be linked to any satisfactory free list.

3. If the free block is not found, call mem_sbrk(input size) and create header for newly allocated block with size and allocation status.

4. After doing either expanding heap by mem_sbrk() or allocating from existing heap, the header should be created or update to store size and allocation status information. Footer is not needed since the 2nd bit of information contains allocation status of the previous block. If previous block is allocated, nothing happens. If previous block is free, then there is footer; just move left 8 byte to fetch its size and do pointer arithmetic to take control of previous blocks.

free(): 1. If pointer is NULL, then return NULL. If not, do pointer arithmetic to locate its header since input pointer is payload pointer.

2. If the neighboring blocks are not free, just free the block. If either previous, next, or both neighboring blocks are free. The coalesce() function should be called to combine contiguous free blocks.

3. Depending on coalescing or simply free, the respective header should be updated and a footer should be added since it is a free block.

realloc(): If the old block is bigger than the reallocation's new size, split the block to one free and one allocated. If not, do malloc() to look for a better fit free block to re-fit the block that satisfies the size or increase the heap.