malloc lab report

2016-12146 이세리
stu224@sp2.snucse.org

0. Intro

At first I tried implementing what was explained in the textbook, which was implicit free list, with first fit placement and boundary tag coalescing. This gave me score of 54.

Then, I tried implementing seglist allocators or segregated free lists in the textbook. Each size class contains explicit free list and there is one class allocated for each two-power size. Since we were not allowed to use any global or static arrays, I allocated a memory space before calling *heap_listp to keep these free lists.

1. 자료구조

```
/*
 *  I implemented
 *  Seglist Allocators described in the ppt
 *
 *  - each size class has its own free list
 *  - one class for each two-power size
 *  - {1},{2},{3,4},{5-8}, ..., {1025-2048}, ...
 *
 *  - It is worth noting the difference between
 *    (predecessor and successor) and (previous and next)
 *    (predecessor and successor) : pred and succ block in free list,
 *    (previous and next) : previous and next block in heap memory
 *
 * - Since we are not allowed to use any global or static arrays,
 *   I allocated a memory space in heap to keep these free lists (Or Seglist Allocators).
 *   This is described in visual text below.
 *
 * -For blocks, I used the boundary tag coalescing technique.
 */
```

2. Allocated Block Structure

```
/* ALLOCATED BLOCK, FREE BLOCK, SEGREGATED FREE LIST, HEAP STRUCTURE

1. Allocated Block

             31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
Header :    |                               size of the block                                   |  |  | 1|
    bp ---> +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
            |                                                                                               |
            |                                                                                               |
            .                               Payload and padding                                             .
            .                                                                                               .
            .                                                                                               .
            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
Footer :    |                               size of the block                                   |     | 1|
            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

3. Free Block Structure

```
2. Free block

           | 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
           +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
Header :   |                     size of the block                                         |  |  | 0|
    bp---> +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
           |                 pointer to pred block in free list                                          |
  bp+4---> +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
           |                 pointer to succ block in free list                                          |
           +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
           .                                                                                             .
           .                                                                                             .
           .                                                                                             .
           +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
Footer :   |                     size of the block                                         |     | 0|
           +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

4. Segregated Free List Structure

```
3. Segregated Free Lists
   (space is allocated in heap to keep this array of pointers)
   (**free_lists points to first pointer in the array)
   ----------------------------------------------------------
   |  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
   |  |size class 0  |size class 1  |       . . .
   |  |(bp of blck 0 |(bp of blck 0 |
   |  | is kept here)|is kept here) |
   |  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+-
   -----------------------------------------------------------
           |               |
     +--+--+--+--+   +--+--+--+--+
     | blck 0    |   | blck 0    |
     +--+--+--+--+   +--+--+--+--+
           |               |
     +--+--+--+--+   +--+--+--+--+
     | blck 1    |   | blck 1    |
     +--+--+--+--+   +--+--+--+--+
```

5. Heap Structure

```
4. Heap Memory After Free Lists
   (prologue and epilogue blocks are tricks that eliminate the edge conditions
   during coalescing)
             31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
             +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
             |                                0 (Padding)                                    |  |  |  |
             +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
             |                                8 (Prologue block)                             |  |  | 1|
*heap_listp ->+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
             |                                8 (Prologue block)                             |  |  | 1|
             +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
             .                                                                                        .
             .                        Both Free and Allocated Blocks inmemory space                   .
             .                                                                                        .
             +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
             |                               0 (Epilogue block hdr)                          |  |  | 1|
             +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
*/
```

6. Implementation

1. mm_init : Before calling mm_malloc mm_realloc or mm_free, the application program calls mm_init to perform any necessary initializations, such as allocating the initial heap area.
2. mm_malloc : The mm_malloc routine returns a pointer to an allocated block payload of at least size bytes.
3. mm_free : The mm_free routine frees the block pointed to by ptr. It returns nothing.
4. mm_realloc : returns a pointer to an allocated region of at least size bytes

these are the functions that I implemented. I think it will be pretty straight-forward since I made very detailed comments.

```c
/*Global variables*/
static char *heap_listp = 0;
static char **free_lists;

/*helper functions*/
static void *extend_heap(size_t size);
static void *coalesce(void *bp);
static void *place(void *bp, size_t asize);
static void add(void *bp, size_t size);
static void delete(void *bp);

void mm_check(int verbose);
```

These are the helper functions I implemented. Most of them are pretty straight-forward but there is one note to make about static void *place(void *bp, size_t asize);

7. static void *place(void *bp, size_t asize)

```
/*place - place block of asize bytes at free block bp
          split if remainder would be at least minimum block size*/
static void *place(void *bp, size_t asize)
{
    size_t bp_size = GET_SIZE(HDRP(bp));
    size_t remainder = bp_size - asize;

    delete(bp);


    if (remainder <= DSIZE * 2) {
        // Do not split block
        PUT(HDRP(bp), PACK(bp_size, 1));
        PUT(FTRP(bp), PACK(bp_size, 1));
    }
    /*
     ususally we just do else here and split the block.
     But there are cases like in binary-bal.rep, binary2-bal.rep

    case :
     allocated <blocks> - (small size block or big size bock)
     small - big - small - big - small - big

     if we free this blocks in order small-big-small-big-small-bg
     it causes no problems.
```

This function places block of size bytes at free block bp and splits if remainder would be at least minimum block size. But I also divided the case where asize is bigger than 96 for the following reasons explained in the comment.

```
case :
allocated <blocks> - (small size block or big size bock)
small - big - small - big - small - big

if we free this blocks in order small-big-small-big-small-bg
it causes no problems.

However, if we free only the big blocks like below,
small - big(freed) - small - big(freed) - small-

Even if we want to use the big freed blocks together, we can't because
there are small allocated blocks in between.

If there is a allocate call to size bigger than big block,
- we have to find another free block.

So what I am trying to do here is put allocated blocks
in continuous places so we can use the freed space

small-small-small-small-big(freed)-big(freed)-

I set the size 96 to get the best result for binary-bal.rep
and binary2-bal.rep
*/
```

```
    else if (asize >= 96) {
        PUT(HDRP(bp), PACK(remainder, 0));
        PUT(FTRP(bp), PACK(remainder, 0));
        PUT(HDRP(NEXT_BLKP(bp)), PACK(asize, 1));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(asize, 1));
        add(bp, remainder);
        return NEXT_BLKP(bp);

    }

    else {
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        PUT(HDRP(NEXT_BLKP(bp)), PACK(remainder, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(remainder, 0));
        add(NEXT_BLKP(bp), remainder);
    }
    return bp;
}
```

8. void mm_check(int verbose) and debugging
I did minimal check for the heap, to check if prologue and epilogue header is bad or not.
Also I created helper functions static void checkblock(void *bp), static void printblock(void *bp) to check if block is double word aligned, if header matches the footer, and print information about the block.
These are some of the debugging processes that I went through.
I commented all the functions that I used to debug to increase efficiency.

```
0xf6a85208: header: [4080:a] footer: [4080:a]
0xf6a861f8: header: [3240:f] footer: [3240:f]
0xf6a86ea0: header: [168:a] footer: [168:a]
0xf6a86f48: header: [168:a] footer: [168:a]
0xf6a86ff0: header: [168:a] footer: [168:a]
0xf6a87098: header: [168:a] footer: [168:a]
0xf6a87140: header: [4080:a] footer: [4080:a]
0xf6a88130: header: [4080:a] footer: [4080:a]
0xf6a89120: header: [4080:a] footer: [4080:a]
```

information about the heap, and allocated block. Size and see if header matches the header.
If there is any problem with the block or the heap(ex. if is not aligned), it will print an error.
Thankfully, it does not print any error in ./traces/amptjp-bal.rep.

9. Results

```
Results for mm malloc:
trace  valid  util     ops      secs   Kops
 0       yes   98%    5694  0.000764   7453
 1       yes   99%    5848  0.000745   7854
 2       yes   99%    6648  0.000839   7927
 3       yes   99%    5380  0.000677   7943
 4       yes   98%   14400  0.001143  12594
 5       yes   94%    4800  0.000970   4951
 6       yes   91%    4800  0.000989   4853
 7       yes   95%   12000  0.001348   8901
 8       yes   88%   24000  0.001836  13070
 9       yes   99%   14401  0.000309  46681
10       yes   98%   14401  0.000185  77885
Total          96%  112372  0.009804  11461

Perf index = 58 (util) + 40 (thru) = 98/100
```