Intro. to Computer SW Systems Lab Report

[Malloc Lab]

20220100 박기현

kihyun@postech.ac.kr

명예 서약 (Honor Code)

나는 이 프로그래밍 과제를 다른 사람의 부적절한 도움 없이 완수하였습니다.

I completed this programming task without the improper help of others.

#define

```
1   /* Basic constants and macros */
2   #define WSIZE       4       /* Word and header/footer size (bytes) */
3   #define DSIZE       8       /* Double word size (bytes) */
4   #define CHUNKSIZE  (1<<12)  /* Extend heap by this amount (bytes) */
5
6   #define MAX(x, y) ((x) > (y)? (x) : (y))
7
8   /* Pack a size and allocated bit into a word */
9   #define PACK(size, alloc)  ((size) | (alloc))
10
11  /* Read and write a word at address p */
12  #define GET(p)       (*(unsigned int *)(p))
13  #define PUT(p, val)  (*(unsigned int *)(p) = (val))
14
15  /* Read the size and allocated fields from address p */
16  #define GET_SIZE(p)  (GET(p) & ~0x7)
17  #define GET_ALLOC(p) (GET(p) & 0x1)
18
19  /* Given block ptr bp, compute address of its header and footer */
20  #define HDRP(bp)       ((char *)(bp) - WSIZE)
21  #define FTRP(bp)       ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
22
23  /* Given block ptr bp, compute address of next and previous blocks */
24  #define NEXT_BLKP(bp)  ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
25  #define PREV_BLKP(bp)  ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
```

<CSAPP 교재 참고>

```c
/* Basic constants and macros */
#define WSIZE 4             //Word and header/footer size (bytes)
#define DSIZE 8             //Double word size (bytes)
#define CHUNKSIZE (1 << 12) //Extend heap by this amount (bytes)

#define MAX(x, y) ((x) > (y) ? (x) : (y))

/* Pack a size and allocated bit into a word */
#define PACK(size, alloc) ((size) | (alloc))

/* Read and write a word at address p */
#define GET(p) (*(unsigned int *)(p))
#define PUT(p, val) (*(unsigned int *)(p) = (val))

/* Read the size and allocated fields from address p */
#define GET_SIZE(p) (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)

/* Given block ptr bp, compute address of its header and footer */
#define HDRP(bp) ((char *)(bp) - WSIZE)
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

/* Given block ptr bp, compute address of next and previous blocks */
#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
#define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))

/* Given block ptr bp, compute address of next and previous segregated free list blocks */
/* Read the i-th segregated free list block */
#define NEXT_SEG_BLKP(bp) (*(char **)(bp + WSIZE))
#define PREV_SEG_BLKP(bp) (*(char **)(bp))
#define SEG_POINTER(i) (*((char **)seg_listp + i))
```

Dynamic Storage Allocator를 구현하기 앞서 위와 같이 Constants와 Macros를 정의하였다.

CSAPP 교재는 Implicit List를 기준으로 구현하였으나, Performance를 향상시키기 위해 Segregated free list로 Allocator를 구현하고자 하였다. 따라서 추가적으로 Segregated free list의 pointer macors를 정의하였다.

int mm_init(void)

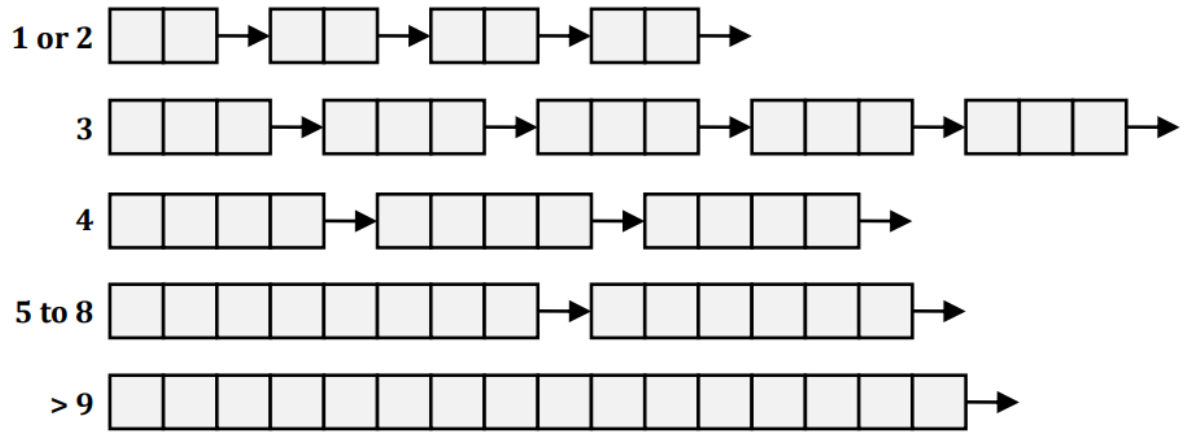
: Initialize the malloc package

```
1   int mm_init(void)
2   {
3       /* Create the initial empty heap */
4       if ((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1)
5           return -1;
6       PUT(heap_listp, 0);                          /* Alignment padding */
7       PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1)); /* Prologue header */
8       PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1)); /* Prologue footer */
9       PUT(heap_listp + (3*WSIZE), PACK(0, 1));     /* Epilogue header */
10      heap_listp += (2*WSIZE);
11
12      /* Extend the empty heap with a free block of CHUNKSIZE bytes */
13      if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
14          return -1;
15      return 0;
16  }
```

<CSAPP 교재 참고>

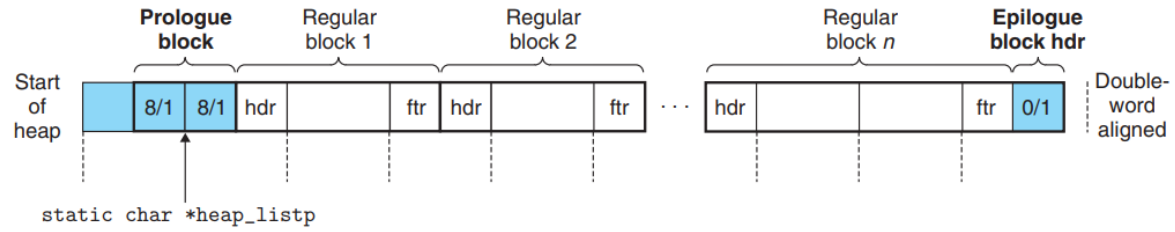
```
int mm_init(void)
{
    int i;

    /* Create the initial segregated free list */
    if ((seg_listp = mem_sbrk(32 * WSIZE)) == (void *) -1) //Expand the heap
        return -1;
    for (i = 0; i < 32; i++)
        SEG_POINTER(i) = NULL;

    /* Create the initial empty heap */
    if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void *) -1) //Expand the heap
        return -1;
    PUT(heap_listp, 0);                                    //Alignment padding
    PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1));         //Prologue header
    PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1));         //Prologue footer
    PUT(heap_listp + (3 * WSIZE), PACK(0, 1));             //Epilogue header
    heap_listp += (2 * WSIZE);

    /* Extend the empty heap with a free block of CHUNKSIZE bytes */
    if (extend_heap(CHUNKSIZE) == NULL)                    //Extend the heap
        return -1;

    return 0;
}
```

가장 먼저 Segregated free list를 생성한다.



Segregated free list는 각 block 크기별로 free list를 갖고 있는 것으로, 큰 사이즈에 대해 2의 n제곱의 크기를 갖는다. Word size가 32-bit임을 고려하여 Segregated free list 역시 2^{32}의 크기의 free list까지 구현하였다.

다음으로 Empty heap을 생성한다.



초기 heap은 총 4 X Word size만큼 확장하는데, 이는 Padding, Prologue block의 header와 footer, Epilogue block의 header를 위한 크기이다. Prologue block의 header와 footer는 모두 크기를 8로 표시하고, Allocated bit를 1로 설정한다. Epilogue block의 header는 구분을 위해 크기를 0으로 표시하고, Allocated bit를 1로 설정한다. 초기 heap을 생성했으면, CHUNCKSIZE(= 2^{12}, page size)만큼 free block을 위한 heap을 확장한다.

void *mm_malloc(size_t size)

: Allocate a block by incrementing the brk pointer

```
void *mm_malloc(size_t size)
{
    size_t asize;      /* Adjusted block size */
    size_t extendsize; /* Amount to extend heap if no fit */
    char *bp;

    /* Ignore spurious requests */
    if (size == 0)
        return NULL;

    /* Adjust block size to include overhead and alignment reqs. */
    if (size <= DSIZE)
        asize = 2*DSIZE;
    else
        asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);

    /* Search the free list for a fit */
    if ((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        return bp;
    }

    /* No fit found. Get more memory and place the block */
    extendsize = MAX(asize,CHUNKSIZE);
    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
        return NULL;
    place(bp, asize);
    return bp;
}
```

<CSAPP 교재 참고>

```
void *mm_malloc(size_t size)
{
    size_t asize;      //Adjusted block size
    size_t extendsize; //Amount to extend heap if no fit
    char *bp;          //Block pointer

    /* Ignore spurious requests */
    if (size == 0)
        return NULL;

    /* Adjust block size to include overhead and alignment reqs */
    asize = ALIGN(size + DSIZE);

    /* Search the free list for a fit */
    if ((bp = find_fit(asize)) != NULL)
    {
        place(bp, asize);                       //Place the block
        return bp;
    }

    /* No fit found */
    /* Get more memory and place the block */
    extendsize = MAX(asize, CHUNKSIZE);
    if ((bp = extend_heap(extendsize)) == NULL) //Extend the heap
        return NULL;
    place(bp, asize);                           //Place the block
    return bp;
}
```

할당하고자 하는 size가 argument로 주어지면, alignment를 위해 ALIGN macro를 이용하여 크기를 조정한다. 조정한 크기만큼의 block이 들어갈 자리를 탐색하기 위해 find_fit 함수를 이용한다. 적절한 위치를 찾으면 그 위치에 block을 배치하고, 적절한 위치를 찾지 못하면 heap을 확장하여 추가로 확장한 heap에 block을 배치한다.

void mm_free(void *ptr)

: Free a block

```
1   void mm_free(void *bp)
2   {
3       size_t size = GET_SIZE(HDRP(bp));
4
5       PUT(HDRP(bp), PACK(size, 0));
6       PUT(FTRP(bp), PACK(size, 0));
7       coalesce(bp);
8   }
9
```

<CSAPP 교재 참고>

```
void mm_free(void *ptr)
{
    size_t size = GET_SIZE(HDRP(ptr)); //Block size

    PUT(HDRP(ptr), PACK(size, 0));     //Free block header
    PUT(FTRP(ptr), PACK(size, 0));     //Free block footer
    coalesce(ptr);                     //Coalesce the free blocks
}
```

할당 해제하고자 하는 block의 pointer가 주어지면, 해당하는 block의 header와 footer의 Allocated bit를 0으로 표시한다. 할당 해제된 block 주변으로 free block이 존재하면 false fragmentation이 발생하므로 이를 막기 위해 coalesce 함수를 호출하여 연결한다.

void *mm_realloc(void *ptr, size_t size)

: Reallocate a block

```
void *mm_realloc(void *ptr, size_t size)
{
    void *old_ptr = ptr; //Old block pointer
    void *new_ptr = ptr; //New block pointer
    size_t asize;        //Block size
    size_t total_size;   //Total Block size

    /* If ptr is NULL, the call is equivalent to mm_malloc(size); */
    /* If size is equal to zero, the call is equivalent to mm_free(ptr); */
    if (old_ptr == NULL)
    {
        return mm_malloc(size);
    }
    else if (size == 0)
    {
        mm_free(old_ptr);
        return NULL;
    }

    /* If ptr is not NULL and size is not zero */
    /* Adjust block size to include overhead and alignment reqs */
    asize = ALIGN(size + DSIZE);

    if (GET_SIZE(HDRP(old_ptr)) >= asize) //Old block size >= New block size
    {
        return ptr;
    }
    else                                  //Old block size < New block size
    {
        total_size = GET_SIZE(HDRP(old_ptr)) + GET_SIZE(HDRP(NEXT_BLKP(old_ptr))); //Total block size

        if (!GET_ALLOC(HDRP(NEXT_BLKP(old_ptr))) && total_size >= size) //If Next of old block is not allocated and total size >= size
        {
            delete_seg_list_block(NEXT_BLKP(old_ptr)); //Delete the block from the segregated free list
            PUT(HDRP(old_ptr), PACK(total_size, 1));   //Pack a size and allocated bit into a word
            PUT(FTRP(old_ptr), PACK(total_size, 1));   //Pack a size and allocated bit into a word
        }
        else                                                    //Otherwise
        {
            new_ptr = mm_malloc(asize);                     //Allocate a new block
            memcpy(new_ptr, old_ptr, GET_SIZE(HDRP(old_ptr))); //Copy old block to new block
            mm_free(old_ptr);                               //Free the old block
        }
        return new_ptr;
    }
}
```

```
/* If ptr is NULL, the call is equivalent to mm_malloc(size); */
/* If size is equal to zero, the call is equivalent to mm_free(ptr); */
if (old_ptr == NULL)
{
    return mm_malloc(size);
}
else if (size == 0)
{
    mm_free(old_ptr);
    return NULL;
}
```

우선, 주어진 재할당하고자 하는 block의 pointer가 NULL이면, mm_malloc(size)와 동일하다. 따라서 mm_malloc(size) 함수 호출을 반환한다. 또한, 주어진 재할당하고자 하는 block의 size가 0이면, mm_free(ptr)과 동일하다. 따라서 mm_free(ptr) 함수를 호출하고, mm_free 함수의 경우 반환을 하지 않기 때문에 NULL을 반환한다.

```
/* If ptr is not NULL and size is not zero */
/* Adjust block size to include overhead and alignment reqs */
asize = ALIGN(size + DSIZE);

if (GET_SIZE(HDRP(old_ptr)) >= asize) //Old block size >= New block size
{
    return ptr;
}
else                                  //Old block size < New block size
{
    total_size = GET_SIZE(HDRP(old_ptr)) + GET_SIZE(HDRP(NEXT_BLKP(old_ptr))); //Total block size

    if (!GET_ALLOC(HDRP(NEXT_BLKP(old_ptr))) && total_size >= size) //If Next of old block is not allocated and total size >= size
    {
        delete_seg_list_block(NEXT_BLKP(old_ptr)); //Delete the block from the segregated free list
        PUT(HDRP(old_ptr), PACK(total_size, 1));   //Pack a size and allocated bit into a word
        PUT(FTRP(old_ptr), PACK(total_size, 1));   //Pack a size and allocated bit into a word
    }
    else                                                        //Otherwise
    {
        new_ptr = mm_malloc(asize);                     //Allocate a new block
        memcpy(new_ptr, old_ptr, GET_SIZE(HDRP(old_ptr))); //Copy old block to new block
        mm_free(old_ptr);                               //Free the old block
    }
    return new_ptr;
}
```

주어진 재할당하고자 하는 block의 pointer가 NULL이 아니거나, size가 0이 아니면, 우선 alignment를 위해 ALIGN macro를 이용하여 크기를 조정한다.

재할당하고자 하는 크기보다 기존 block의 크기가 더 크거나 같은 경우에는 새로 할당을 진행할 필요가 없기 때문에 pointer를 그대로 반환한다.

반면, 그렇지 않은 경우는 두 가지로 나눌 수 있다.

먼저, 기존 block의 다음 block이 free인 경우에서 기존 block의 크기와 다음 block의 크기까지 합친 크기가 재할당하고자 하는 크기보다 더 크거나 같은 경우이다. 이 경우에는 새로 할당을 진행할 필요 없이 다음 block을 Segregated free list에서 삭제하고, 기존 block의 크기를 합친 크기로 바꾸어 표시한다.

다음으로, 기존 block의 다음 block이 할당되어 있거나, free이더라도 합친 크기가 재할당하고자 하는 크기보다 작은 경우이다. 이 경우에는 mm_malloc 함수를 호출하여 새롭게 할당을 진행하고, 기존 block의 내용을 새 block으로 복사한 뒤, 기존 block은 mm_free 함수를 호출하여 할당 해제한다.

static void *extend_heap(size_t words)

: Extend the heap with a new free block

```
1   static void *extend_heap(size_t words)
2   {
3       char *bp;
4       size_t size;
5
6       /* Allocate an even number of words to maintain alignment */
7       size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
8       if ((long)(bp = mem_sbrk(size)) == -1)
9           return NULL;
10
11      /* Initialize free block header/footer and the epilogue header */
12      PUT(HDRP(bp), PACK(size, 0));         /* Free block header */
13      PUT(FTRP(bp), PACK(size, 0));         /* Free block footer */
14      PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* New epilogue header */
15
16      /* Coalesce if the previous block was free */
17      return coalesce(bp);
18  }
```

<CSAPP 교재 참고>

```
static void *extend_heap(size_t words)
{
    char *bp;    //Block pointer
    size_t size; //Block size

    /* Adjust block size to include overhead and alignment reqs */
    size = ALIGN(words);

    if ((long)(bp = mem_sbrk(size)) == -1) //Expand the heap by size bytes
        return NULL;

    /* Initialize free block header/footer and the epilogue header */
    PUT(HDRP(bp), PACK(size, 0));          //Free block header
    PUT(FTRP(bp), PACK(size, 0));          //Free block footer
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));  //New epilogue header

    /* Coalesce if the previous block was free */
    return coalesce(bp);                   //Coalesce the free blocks
}
```

heap을 확장하고자 하는 만큼의 size가 주어지면, 우선 alignment를 위해 ALIGN macro를 이용하여 크기를 조정한다. heap을 확장하여 생성된 free block의 header와 footer는 조정된 크기로 표시하고, Allocated bit를 0으로 설정한다. Epilogue block의 header는 구분을 위해 크기를 0으로 표시하고, Allocated bit를 1로 설정한다. 확장하기 전 heap의 마지막 block이 free인 경우가 있을 수 있으므로 coalesce 함수 호출을 반환하여 fragmentation을 방지한다.

static void *coalesce(void *bp)

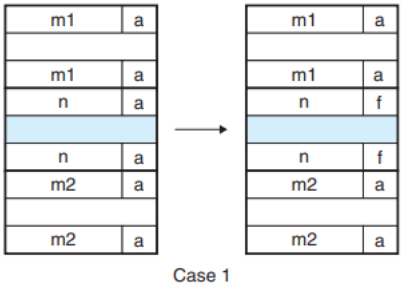
: Coalesce the free blocks

```
10  static void *coalesce(void *bp)
11  {
12      size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
13      size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
14      size_t size = GET_SIZE(HDRP(bp));
15
16      if (prev_alloc && next_alloc) {            /* Case 1 */
17          return bp;
18      }
19
20      else if (prev_alloc && !next_alloc) {      /* Case 2 */
21          size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
22          PUT(HDRP(bp), PACK(size, 0));
23          PUT(FTRP(bp), PACK(size,0));
24      }
25
26      else if (!prev_alloc && next_alloc) {      /* Case 3 */
27          size += GET_SIZE(HDRP(PREV_BLKP(bp)));
28          PUT(FTRP(bp), PACK(size, 0));
29          PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
30          bp = PREV_BLKP(bp);
31      }
32
33      else {                                     /* Case 4 */
34          size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
35              GET_SIZE(FTRP(NEXT_BLKP(bp)));
36          PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
37          PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
38          bp = PREV_BLKP(bp);
39      }
40      return bp;
41  }
```

<CSAPP 교재 참고>

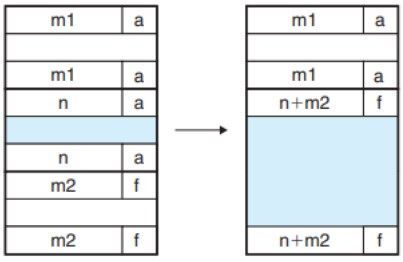
```c
static void *coalesce(void *bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp))); //Allocated bit of previous block
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp))); //Allocated bit of next block
    size_t size = GET_SIZE(HDRP(bp));                   //Block size

    if (prev_alloc && next_alloc)       //Case 1
    {
        add_seg_list_block(bp, size);                                           //Add a block into the segregated free list
        return bp;
    }
    else if (prev_alloc && !next_alloc) //Case 2
    {
        delete_seg_list_block(NEXT_BLKP(bp));                                   //Delete the next block From the segregated free list
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));                                  //Update the next block size
        PUT(HDRP(bp), PACK(size, 0));                                           //Free block header
        PUT(FTRP(bp), PACK(size, 0));                                           //Free block footer
    }
    else if (!prev_alloc && next_alloc) //Case 3
    {
        delete_seg_list_block(PREV_BLKP(bp));                                   //Delete the previous block From the segregated free list
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));                                  //Update the previous block size
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));                                //Free previous block header
        PUT(FTRP(bp), PACK(size, 0));                                           //Free block footer
        bp = PREV_BLKP(bp);                                                     //Previous block pointer
    }
    else                                //Case 4
    {
        delete_seg_list_block(PREV_BLKP(bp));                                   //Delete the previous block From the segregated free list
        delete_seg_list_block(NEXT_BLKP(bp));                                   //Delete the next block From the segregated free list
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp)));  //Update the block size
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));                                //Free previous block header
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));                                //Free next block header
        bp = PREV_BLKP(bp);                                                     //Previous block pointer
    }
    add_seg_list_block(bp, size);                                               //Add a block into the segregated free list

    return bp;
}
```

coalesce 함수는 false fragmentation을 막기 위해 연속된 free block이 있는 경우 하나의 free block으로 연결하는 함수이다.



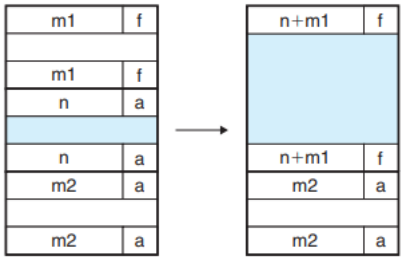
Case 1

첫 번째 경우는 이전 block과 다음 block 모두 할당되어 있는 경우로, 할당 해제된 해당 block만 Segregated free list에 추가하고 pointer를 반환한다.



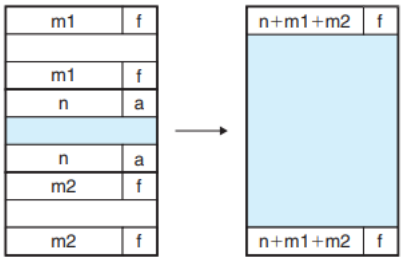
Case 2

두 번째 경우는 이전 block은 할당되어 있으나, 다음 block이 free인 경우로, 우선 다음 block을 Segregated free list에서 제거하고, 해당 block과 다음 block을 합친 크기만큼 해당 block의 header와 footer에 바꾸어 표시한 뒤, 합친 block을 Segregated free list에 새롭게 추가한다.



Case 3

세 번째 경우는 반대로 다음 block은 할당되어 있으나, 이전 block이 free인 경우로, 두 번째 경우와 마찬가지로 진행한다.



Case 4

네 번째 경우는 이전 block과 다음 block 모두 free인 경우로, 과정은 역시 위와 동일하다.

static void *find_fit(size_t asize)

: Search the free list for a fit

```
1   static void *find_fit(size_t asize)
2   {
3       /* First-fit search */
4       void *bp;
5
6       for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
7           if (!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
8               return bp;
9           }
10      }
11      return NULL; /* No fit */
12  #endif
13  }
```

<CSAPP 교재 참고>

```
static void *find_fit(size_t asize)
{
    int i;
    int index = get_index(asize); //index of free list
    void *ptr;                    //temp pointer
    void *bp = NULL;              //Block pointer

    /* Search the free list for a fit */
    for (i = index; i < 32; i++)
    {
        for(ptr = SEG_POINTER(i); ptr != NULL; ptr = NEXT_SEG_BLKP(ptr))
            if(GET_SIZE(HDRP(ptr)) >= asize)
                if((bp == NULL) || (GET_SIZE(HDRP(ptr)) < GET_SIZE(HDRP(bp))))
                    bp = ptr;
        if(bp != NULL)
            return bp;
    }

    return NULL; //No fit
}
```

free list에서 새롭게 block이 들어갈 적절한 자리가 있는지 탐색하는 함수로, performance 향상을 위해 best fit을 활용한다. 우선, 주어진 크기에 따라 index 시작을 설정한 뒤, Segregated free list에서 block이 들어갈 적절한 자리를 찾아 반환한다. 적절한 자리를 찾는 과정은 새롭게 배치할 block 크기보다 큰 기존 block이 있는 경우 중에서 가장 작은 block을 찾아 pointer를 반환한다.

static void place(void *bp, size_t size)

: Place the block

```
1   static void place(void *bp, size_t asize)
2   {
3       size_t csize = GET_SIZE(HDRP(bp));
4
5       if ((csize - asize) >= (2*DSIZE)) {
6           PUT(HDRP(bp), PACK(asize, 1));
7           PUT(FTRP(bp), PACK(asize, 1));
8           bp = NEXT_BLKP(bp);
9           PUT(HDRP(bp), PACK(csize-asize, 0));
10          PUT(FTRP(bp), PACK(csize-asize, 0));
11      }
12      else {
13          PUT(HDRP(bp), PACK(csize, 1));
14          PUT(FTRP(bp), PACK(csize, 1));
15      }
16  }
```

<CSAPP 교재 참고>

```
static void place(void *bp, size_t asize)
{
    size_t csize = GET_SIZE(HDRP(bp)); //Block size
    delete_seg_list_block(bp);         //Delete the block From the segregated free list

    if ((csize - asize) >= (2 * DSIZE))
    {
        PUT(HDRP(bp), PACK(asize, 1));         //Pack a size and allocated bit into a word
        PUT(FTRP(bp), PACK(asize, 1));         //Pack a size and allocated bit into a word
        bp = NEXT_BLKP(bp);                    //Next block pointer
        PUT(HDRP(bp), PACK(csize - asize, 0)); //Free block header
        PUT(FTRP(bp), PACK(csize - asize, 0)); //Free block footer
        add_seg_list_block(bp, csize - asize); //Add a block into the segregated free list
    }
    else
    {
        PUT(HDRP(bp), PACK(csize, 1));         //Pack a size and allocated bit into a word
        PUT(FTRP(bp), PACK(csize, 1));         //Pack a size and allocated bit into a word
    }
}
```

block이 위치할 자리를 찾았으면, 해당 위치에 block을 배치하는 함수이다. 이때, block을 배치한 뒤 남은 크기가 충분하면, 다른 block을 위해 사용할 수 있도록 Segregated free list에 다시 추가한다. 그렇지 않다면 그 block 전체를 할당하는 데 온전히 사용한다.

static int get_index(size_t size)

: Get the index of the free list

```
static int get_index(size_t size)
{
    int index; //index of free list

    /* Get the index of the free list */
    for (index = 0; index < 32; index++)
        if(size <= (1 << index))
            return index;
    return index;
}
```

Segregated free list에서 block이 위치할 적절한 index를 찾아 반환하는 함수이다. 2의 n제곱 크기별로 list가 있다는 점을 고려하여 index를 설정한다.

static void add_seg_list_block(void *bp, size_t size)

: Add a block into the segregated free list

```
static void add_seg_list_block(void *bp, size_t size)
{
    int index = get_index(size); //index of free list

    /* Add a block into the segregated free list */
    PREV_SEG_BLKP(bp) = NULL;
    NEXT_SEG_BLKP(bp) = SEG_POINTER(index);
    if (SEG_POINTER(index) != NULL)
        PREV_SEG_BLKP(SEG_POINTER(index)) = bp;
    SEG_POINTER(index) = bp;
}
```

새롭게 할당 해제된 block을 Segregated free list에 추가하는 함수이다. 이때 각 list의 가장 앞부분에 block을 배치한다.

static void delete_seg_list_block(void *bp)

: Delete the block from the segregated free list

```
static void delete_seg_list_block(void *bp)
{
    int size = GET_SIZE(HDRP(bp)); //Block size
    int index = get_index(size);   //index of free list

    /* Delete the block from the segregated free list */
    if (SEG_POINTER(index) != bp)
    {
        if (NEXT_SEG_BLKP(bp) != NULL)
            PREV_SEG_BLKP(NEXT_SEG_BLKP(bp)) = PREV_SEG_BLKP(bp);
        NEXT_SEG_BLKP(PREV_SEG_BLKP(bp)) = NEXT_SEG_BLKP(bp);
    }
    else
    {
        if (NEXT_SEG_BLKP(bp) != NULL)
            PREV_SEG_BLKP(NEXT_SEG_BLKP(bp)) = NULL;
        SEG_POINTER(index) = NEXT_SEG_BLKP(bp);
    }
}
```

새롭게 할당된 block을 Segregated free list에서 제거하는 함수이다.

Heap Consistency Checker

int mm_check(void)

```
int mm_check(void)
{
    if (mark_or_not() == 0)
        return 0;
    if (coalesce_or_not() == 0)
        return 0;
    if (freeinlist_or_not() == 0)
        return 0;
    if (overlap_or_not() == 0)
        return 0;
    if (valid_or_not() == 0)
        return 0;

    return 1;
}
```

각 함수별로 테스트를 설정하여 에러 메시지 출력과 함께 0 을 반환받은 경우 0 을 반환하여 Heap 이 Consistent 하지 않다는 것을 알려준다.

static int mark_or_not(void)

: Is every block in the free list marked as free?

```
static int mark_or_not(void)
{
    int i;
    void* ptr;

    for (i = 0; i < 32; i++)
    {
        for (ptr = SEG_POINTER(i); ptr != NULL; ptr = NEXT_SEG_BLKP(ptr))
        {
            if (GET_ALLOC(HDRP(ptr)) || GET_ALLOC(FTRP(ptr)))
            {
                printf("Error: Exist block in the free list marked as allocated\n");
                return 0;
            }
        }
    }
    return 1;
}
```

위 테스트 함수는 free list 에 있는 모든 block 이 free 로 표시되어 있는가를 확인하는 함수이다. 이를 확인하기 위해 Segregated free list 를 처음부터 모두 탐색하여 block 의 header 혹은 footer 의 Allocated bit 가 1 로 표시된 block 이 있는지 확인한다. 만약 있는 경우 에러 메시지를 출력하고 0 을 반환한다. 그렇지 않으면 1 을 반환하여 heap 이 consistent 함을 알려준다.

static int coalesce_or_not(void)

: Are there any contiguous free blocks that somehow escaped coalescing?

```
static int coalesce_or_not(void)
{
    char *ptr = heap_listp;

    for (ptr = NEXT_BLKP(ptr); GET_SIZE(ptr) != 0; ptr = NEXT_BLKP(ptr))
    {
        if ((GET_ALLOC(HDRP(ptr)) == 0) && (GET_ALLOC(HDRP(PREV_BLKP(ptr))) == 0))
        {
            printf("Error: Exist any contiguous free blocks that somehow escaped coalescing\n");
            return 0;
        }
    }
    return 1;
}
```

위 테스트 함수는 연결되어 있지 않은 연속된 free block 이 있는가, 즉 false fragmentation 이 있는가를 확인하는 함수이다. 이를 확인하기 위해 block 을 처음부터 탐색하여 연속적으로 Allocated bit 가 0 으로 표시된 경우가 있는지 확인한다. 만약 있는 경우 에러 메시지를 출력하고 0 을 반환한다. 그렇지 않으면 1 을 반환하여 heap 이 consistent 함을 알려준다.

static int freeinlist_or_not(void)

: Is every free block actually in the free list?

```
static int freeinlist_or_not(void)
{
    char *ptr;
    char *_ptr;
    int i;
    int flag;

    for (ptr = heap_listp; GET_SIZE(ptr) != 0; ptr = NEXT_BLKP(ptr))
    {
        if ((GET_ALLOC(HDRP(ptr)) == 0))
        {
            flag = 0;
            for (i = 0; i < 32; i++)
            {
                for (_ptr = SEG_POINTER(i); _ptr != NULL; _ptr = NEXT_SEG_BLKP(_ptr))
                {
                    if (ptr == _ptr)
                    {
                        flag = 1;
                        break;
                    }
                }
            }
            if (flag == 0)
            {
                printf("Error: Exist Free block that is not in free list");
            }
        }
    }
    return 1;
}
```

위 테스트 함수는 모든 free block 이 free list 에 있는가를 확인하는 함수이다. 이를 확인하기 위해 block 을 처음부터 탐색하여 Allocated bit 가 0 으로 설정되어 있는 경우, 즉 free block 인 경우 Segregated free list 에 있는지 확인한다. 만약 없는 경우 에러 메시지를 출력하고 0 을 반환한다. 그렇지 않으면 1 을 반환하여 heap 이 consistent 함을 알려준다.

static int overlap_or_not(void)

: Do any allocated blocks overlap?

```
static int overlap_or_not(void)
{
    char *ptr;

    for (ptr = heap_listp; GET_SIZE(ptr) != 0; ptr = NEXT_BLKP(ptr))
    {
        if ((GET_ALLOC(HDRP(ptr)) == 1))
        {
            if(ptr + GET_SIZE(HDRP(ptr)) - WSIZE >= NEXT_BLKP(ptr))
            {
                printf("Error: Exist allocated blocks overlapped\n");
                return 0;
            }
        }
    }
    return 1;
}
```

위 테스트 함수는 할당된 block 이 overlap 되어 있는가를 확인하는 함수이다. 이를 확인하기 위해 block 을 처음부터 탐색하여 각 block 의 크기만큼 건너뛴 위치보다 다음 block 의 시작 위치가 앞에 있는지 확인한다. 만약 있는 경우 에러 메시지를 출력하고 0 을 반환한다. 그렇지 않으면 1 을 반환하여 heap 이 consistent 함을 알려준다.

static int valid_or_not(void)

: Do the pointers in a heap block point to valid heap address?

```
static int valid_or_not(void)
{
    char *ptr;

    for(ptr = NEXT_BLKP(heap_listp); GET_SIZE(ptr) != 0; ptr = NEXT_BLKP(ptr))
    {
        if(HDRP(ptr) < HDRP(NEXT_BLKP(heap_listp)))
        {
            printf("Error: Exist not valid pointer\n");
            return 0;
        }
    }
    return 1;
}
```

위 테스트 함수는 heap block 에 있는 pointer 가 유효한 heap 주소를 가리키고 있는가를 확인하는 함수이다. 이를 확인하기 위해 block 을 처음부터 탐색하여 block 의 pointer 가 heap 시작보다 앞에 있는지 확인한다. 만약 있는 경우 에러 메시지를 출력하고 0 을 반환한다. 그렇지 않으면 1 을 반환하여 heap 이 consistent 함을 알려준다.

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.000300 18993
 1       yes   99%    4805  0.000239 20121
 2       yes   55%   12000  0.000412 29162
 3       yes   55%    8000  0.000255 31422
 4       yes   51%   24000  0.000759 31641
 5       yes   51%   16000  0.000468 34166
 6       yes   99%    5848  0.000275 21235
 7       yes   99%    5032  0.000232 21718
 8       yes   66%   14400  0.000433 33256
 9       yes   66%   14400  0.000461 31243
10       yes   99%    6648  0.000333 19964
11       yes   99%    5683  0.000278 20435
12       yes  100%    5380  0.000276 19521
13       yes  100%    4537  0.000232 19581
14       yes   96%    4800  0.000495  9695
15       yes   96%    4800  0.000494  9715
16       yes   95%    4800  0.000486  9877
17       yes   95%    4800  0.000487  9864
18       yes   87%   14401  0.000275 52310
19       yes   87%   14401  0.000277 51989
20       yes   67%   14401  0.000215 66857
21       yes   67%   14401  0.000213 67484
22       yes   66%      12  0.000001 20000
23       yes   66%      12  0.000000 24000
24       yes   89%      12  0.000001 17143
25       yes   89%      12  0.000001 20000
Total          82%  209279  0.007896 26504

Perf index = 49 (util) + 40 (thru) = 89/100
```

Dynamic Storage Allocator의 Performance는 위와 같다.