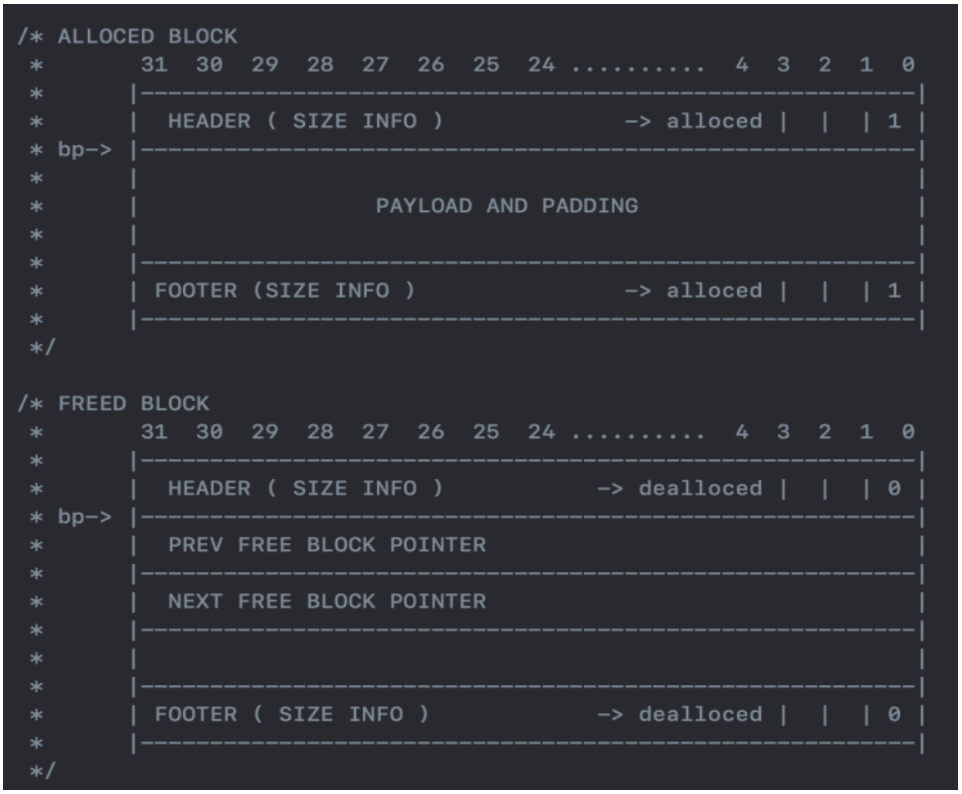
# LAB_03 MALLOC LAB

2014-19498
독어교육과 정은주

1.개요

dynamic memory allocator는 heap이라는 virtual memory를 관리한다. 이 heap은 여러 사이즈의 block으로 이루어져 있고, allocated 혹은 free의 상태이다. Allocator는 explicit, implicit, segregated list 등 여러 방식이 있다.

새로이 구현된 mm.c는 기본적으로 Explicit Free List의 linked list와 더불어서 Segregated List Allocator를 사용한다. 더불어서 Free block이 생길 때마다 Coalescing을 해주고, best fit을 통해서 최적의 free block을 free list에서 찾아내는 방법을 사용한다. 우선, free block들은 seglist에서 보관된다. minimum block size는 16 byte이다. 모든 block은 header, footer 그리고 allocation bit를 마지막에 가진다. allocated block의 경우는 header와 footer만을 가지지만, free list는 prev free block 그리고 next free block에 대한 pointer를 가지고 있다. 따라서 free pointer를 위한 DSIZE 만큼의 공간이 추가로 요구된다. 자세한 Block의 형태는 아래 그림과 같다.




뼈대 함수의 내용 및 Macro의 경우는 교과서 P893- 897에 제시된 implicit list 구현 code를 참조하여 추가하는 방식으로 작성되었다. array를 사용할 수 없는 제약사항으로 인해서 heap을 위한 pointer와 seg list를 위한 pointer만을 global variable로 사용하였고, 더불어서 함수의 경우에는 free block을 집어넣고 빼는 add_free_block & delete_free_block 외에는 coalesce, extend_heap, find_fit, place의 기본적으로 필요한 함수들만으로 구성하여 mm.c가 최대한 간결하도록 하였다.

```c
/* Segregated list max number */
#define SEG_LIST  24
/* Minimum size for the efficiency */
#define MINSIZE 200

/* Helper macros for the segregated list */
#define GET_VAL(ptr)        ((char **)(ptr))
#define UNSIGN(p)           ((unsigned int)(p))
#define GET_BLK_SIZE(ptr)   (GET_SIZE(HDRP(ptr)))
#define SET_SEG_LIST_PTR(ptr, idx, val)  (*(GET_VAL(ptr) + idx) = val)
#define GET_SEG_LIST_PTR(ptr, idx) (*(GET_VAL(ptr) + idx))

/* each class's address in the seglist */
#define GET_PREV_ADR(bp)    ((char *)(bp))
#define GET_NEXT_ADR(bp)    ((char *)(bp) + WSIZE)

/* each actual address of classes in the seglist */
#define GET_PREV_BLK(bp)    (*(GET_VAL(bp)))
#define GET_NEXT_BLK(bp)    (*(GET_VAL(GET_NEXT_ADR(bp))))
```

## 2. Function Explanation

<Helper Function>

● *extend_heap(size_t words) - 교과서 p894 참조

교과서에 있는 뼈대 코드를 그대로 사용하였으며, heap을 늘려준다는 의미는 아직 allocated 되지 않는 block을 추가하는 것과 같으므로 free list에 더해준다. 따라서 add_free_block을 이용한다.

● *coalesce(void *bp) - 교과서 p896 참조

prologue와 epilogue는 항상 allocated 되었는지 표시를 하고 있기 때문에 어떤 block을 free한 이후에 앞뒤로 만약 free된 block이 있다면, 합쳐주는 과정이 필요하다. 이때 가능한 경우의 수는 4가지뿐이다. 1) 앞뒤 블록이 모두 allocated block인 경우, 2) 뒤 블록만 free block인 경우, 3) 앞의 블록만 free block인 경우, 4) 앞뒤 블록이 모두 free block인 경우. 교과서의 뼈대 코드를 그대로 사용하였고, coalescing이 필요한 경우에는 해당 free block을 seg list에서 delete_free_block 함수를 이용하여 삭제해주어야 한다. 그리고 다시 coalesce 된 block을 add_free_block 함수를 이용하여 segregated free list에 넣어준다.

● *find_fit(size_t asize, size_t csize, int index)

best fit 알고리즘을 사용한다. 모든 free block을 살피면서 가장 알맞은 block을 찾는데, asize에 해당하면서 가장 사이즈가 작은 block을 선택한다. recursive function으로 구현하였으며, 만약 가장 작으면서 해당되는 block을 발견하면 return 하고 그렇지 않다면 seg list의 마지막 class까지 계속해서 찾는 과정을 거친다.

```
static void *find_fit(size_t asize, size_t csize, int index)
{
    /* best-fit search */
    if(index > SEG_LIST)
        return NULL;

    void *seg_ptr = NULL;
    /* set the list ptr for the seg list */
    seg_ptr = GET_SEG_LIST_PTR(seg_listp, index);

    /* search the best block for the asize */
    if(csize <= 1 && seg_ptr != NULL) {
        while(seg_ptr != NULL && (asize > GET_BLK_SIZE(seg_ptr)))
            seg_ptr = GET_PREV_BLK(seg_ptr);
        /* if found then return */
        if(seg_ptr != NULL)
            return seg_ptr;
    }
    csize >>= 1;
    /* if not fount search another class */
    return find_fit(asize, csize, index+1);
}
```

● *place(void *bp, size_t asize) - 교과서 p920

minimum block size가 16byte이기 때문에, 만약 할당해야하는 block size 이외에 남는 부분의 경우에는 header와 footer를 위한 공간이 충분하지 않는 경우에는 free block을 만들기 위해 split을 할 필요가 없고, 만약 2*DSIZE보다 크게 된다면 split을 해서 새로운 free block을 만들어주는 것이 더 효율적이다. 이때 임의로 MINIMUM SIZE 200을 설정하니 preformance 값에서 변화가 생기는 것을 알 수 있었다. 이는 binary.rep와 realloc.rep를 위한 optimized된 값으로 값을 설정하기 전과 후의 performance 차이가 생각보다 크게 났다.

```
    /* the boundary size is set arbitrary fo
    else if(asize <= MINSIZE) {
        PUT(HDRP(bp), PACK(padding, 0));
        PUT(FTRP(bp), PACK(padding, 0));
        nbp = NEXT_BLKP(bp);
        PUT(HDRP(nbp), PACK(asize, 1));
        PUT(FTRP(nbp), PACK(asize, 1));
        add_free_block(bp, padding);
        return nbp;
```

● add_free_block(void *bp, size_t blk_size)

segregated list를 위해 double pointer를 이용해 각 class의 previous address와 next address뿐만이 아니라 각 block address를 접근할 수 있어야 한다. 각각을 macro로 설정해두고, global variable인 seg_listp를 알맞은 pointer 접근으로 각각의 주소값을 얻고 설정한다. 새로 들어오는 block은 size에 의해 정렬되어 있는 seg list 내에 어디에 들어가야 할지를 정해야 하는데 이는 block 이전의 block 유무 그리고 blcok 이후의 element 유무에 의해서 달라진다.

```
/* the blk_size's class address of segregated list
pos = GET_SEG_LIST_PTR(seg_listp, index);

while(pos != NULL && blk_size > GET_BLK_SIZE(pos))
    prev_pos = pos;
    pos = GET_PREV_BLK(pos);
}

/* bp is between seg list and the next item */
if (pos && prev_pos) {
    PUT(GET_PREV_ADR(prev_pos), UNSIGN(bp));
    PUT(GET_NEXT_ADR(bp), UNSIGN(prev_pos));
    PUT(GET_PREV_ADR(bp),UNSIGN(pos));
    PUT(GET_NEXT_ADR(pos), UNSIGN(bp));
}
```

```
else if (pos && !prev_pos) {
/* bp is the first item in seg list, insert at start */
    PUT(GET_NEXT_ADR(pos), UNSIGN(bp));
    PUT(GET_PREV_ADR(bp), UNSIGN(pos));
    PUT(GET_NEXT_ADR(bp), UNSIGN(NULL));
    SET_SEG_LIST_PTR(seg_listp, index, bp);
}
/* next to bp there is no item  */
else if (!pos && prev_pos) {
    PUT(GET_NEXT_ADR(bp), UNSIGN(prev_pos));
    PUT(GET_PREV_ADR(prev_pos), UNSIGN(bp));
    PUT(GET_PREV_ADR(bp), UNSIGN(NULL));
}
/* bp is the first item and next to bp there is no item */
else {
    PUT(GET_PREV_ADR(bp), UNSIGN(NULL));
    PUT(GET_NEXT_ADR(bp), UNSIGN(NULL));
    SET_SEG_LIST_PTR(seg_listp, index, bp);
```

새로운 free block이 들어오는 경우에 add_free_block function이 사용되는데 이러한 경우는 1) mm_free를 통해서 free block이 생겨나는 경우 2) realloc을 통해서 남는 부분 즉, 새로 free block이 될 수 있는 경우 3) extend_heap을 통해서 heap을 늘리는 경우(거대한 free block이 생긴다) 4) coalesce를 통해서 앞뒤의 free block이 합쳐지는 경우 5) place를 통해서 mm_malloc으로 새로 할당되는 경우 중, 사이즈가 남는 경우(free block으로 만들 수 있는 경우) 등 이 존재한다.

● delete_free_block(void *bp)

segregated list를 위해서 만들어진 function으로 double pointer를 이용해 각 class의 previous address와 next address뿐만이 아니라 각 block address를 접근할 수 있어야 한다. 각각을 macro로 설정해두고, global variable인 seg_listp를 알맞은 pointer 접근으로 각각의 주소 값을 얻고 설정한다. 삭제되어지는 block은 이전의 block 유무 그리고 blcok 이후의 element 유무에 의해서 달라진다. 각각의 위치를 seg list 내에서 다시 재설정해주어야 하기 때문이다.

seg list 내에서 free block으로 설정되어 있던 block이 allocated 되는 경우에 delete_free_block function을 통해서 seg list에서 삭제가 되는데, 1) realloc function에서 previous 또는 next block과 합쳐져 realloc 가능한 경우에는 이 block들이 삭제 될 때에 delete_free_block( prev_ptr) 혹은 delete_free_block(next_ptr)를 사용한다. 2) coalesce를 하는 경우에 previous block 혹은 next block과 합쳐질 수 있는 경우에는 delete_free_block(bp) 혹은 delete_free_block(PREV_BLKP(bp)) 혹은 delete_free_block(NEXT_BLKP(bp))를 사용해서 free block을 삭제하고, 다시 하나로 합쳐서 add_free_block을 사용한다. 3) place를 통해서 mm_malloc으로 새로 할당되는 경우 free block을 삭제해야 새로 block을 할당할 수 있다.

```
/* removing the block depends on the next_adr and prev_adr */
if(next_adr == NULL) {

    SET_SEG_LIST_PTR(seg_listp, index, prev_adr);
    list_ptr = GET_SEG_LIST_PTR(seg_listp, index);
    /* !next_adr && list_ptr */
    if(list_ptr != NULL)
        PUT(GET_NEXT_ADR(list_ptr), UNSIGN(NULL));

} else {
    PUT(GET_PREV_ADR(next_adr), UNSIGN(prev_adr));
    /* next_adr && prev_adr */
    if(prev_adr != NULL)
        PUT(GET_NEXT_ADR(prev_adr), UNSIGN(next_adr));
}
```

● int mm_check(void)

기본적으로 heap의 consistency를 체크한다. allocation이 제대로 됐는지 혹은 header와 footer의 consistency가 제대로 되어 있는지, 사이즈는 제대로 설정되어 있는지, 그리고 8 byte alignment와 heap에서 잘못된 주소를 참조하고 있지 않은지, coalesce가 제대로 이루어져서 free block이 연속되어 있는 것은 아닌지 역시 확인한다.

1. Free List

1) Checking if the allocated block is in free list

```
if(GET_ALLOC(blkp)) {
    printf("FREE BLOCK %p MARKED ALLOC\n", blkp);
    errnum = -1;
}
```

2) Checking if the free block address is invalid

```
if(blkp < mem_heap_lo() || blkp > mem_heap_hi()) {
    printf("FREE BLOCK %p INVALID\n", blkp);
    errnum = -1;
}
```

3) Checking for the header and footer consistency

```
if(GET_SIZE(HDRP(blkp)) != GET_SIZE(FTRP(blkp))) {
    printf("FREE BLOCK %p HEADER AND FOOTER HAS DIFFERENT
        SIZE\n", blkp);
    errnum = -1;
}
```

4) Checking for the alignment rule – 8 byte

```
if(UNSIGN(blkp) % DSIZE != 0) {
    printf("FREE BLOCK %p SHOULD BE 8 BYTE ALIGNED\n",
        blkp);
    errnum = -1;
}
```

5) Checking if the free blocks are appropriately coalesced

```
if(nblkp != NULL && HDRP(blkp) - FTRP(blkp) == DSIZE) {
    printf("FREE BLOCK %p SHOULD BE COALESCED\n", blkp);
    errnum = -1;
}
```

2 Heap

1) Checking for the header and footer (SIZE and ALLOC)

```
if(GET_ALLOC(HDRP(list_ptr)) != GET_ALLOC(FTRP(list_ptr))) {
    printf("BLOCK %p HEADER AND FOOTER HAS DIFFERENT
        ALLOCATION BIT\n", blkp);
    errnum = -1;
}
```

```
if(GET_SIZE(HDRP(list_ptr)) != GET_SIZE(FTRP(list_ptr))) {
    printf("BLOCK %p HEADER AND FOOTER HAS DIFFERENT SIZE\n",
        blkp);
    errnum = -1;
}
```

2) Checking if the block address is invalid

```
if(list_ptr < mem_heap_lo() || list_ptr > mem_heap_hi()) {
    printf("BLOCK %p INVALID\n", list_ptr);
}
```

*mem_heap_lo() 그리고 mem_heap_hi()를 이용해서 Heap List를 쭉 훑도록 한다.

<Dynamic Storage Allocator Function>

● int mm_init(void) - 교과서 p894 참조

mm_malloc, mm_realloc, mm_free를 부르기 전에 필요한 setting을 위해서 mm_init을 부르게 되는데, 초기의 heap을 할당하는 등의 일을 한다. 만약 extend_heap을 통해서 힙 사이즈를 할당하려는데 문제가 생기게 되면, -1을 return하고 그렇지 않다면 0을 return한다. seg list의 경우에도 heap에 seg list 수만큼 space를 할당해주고, 각각 class에 해당하는 부분을 NULL 값으로 초기화해준다.

```
/* initialize the seg_lists */
for(int i = 0; i<SEG_LIST; i++) {
    SET_SEG_LIST_PTR(seg_listp, i, NULL);
}
```

● mm_malloc(size_t size) - 교과서 p897 참조

mm_malloc은 brk pointer를 증가하면서 block을 allocate한다. 항상 8 byte 만큼 align되어야 하고, 이 값을 return 해야 한다. find_fit 함수를 통해서 free list에서 가장 알맞은 공간을 찾고 거기에 값을 할당하는데, 만약 알맞은 block이 존재하지 않는다면, heap 공간이 부족한 것이므로, extend_heap을 통해서 공간을 확보한 뒤에 place 함수를 통해서 block을 할당한다. 교과서에 나온 뼈대 코드를 그대로 사용하였다.

● mm_free(void *bp) - 교과서 p896 참조

mm_malloc 그리고 mm_realloc에 의해서 allocated된 block을 free하기 위해서 사용된다. 함수는 교과서에 나온 뼈대를 사용했으며, free된 block을 기록하기 위해서 add_free_block 함수를만 새로 더해주었다.

● *mm_realloc(void *ptr, sie_t size)

이미 allocated 된 block의 size를 재조정해주는 함수로 ptr이 NULL인지의 여부, size가 0인지의 여부 그리고 사이즈가 줄어드는지 늘어나는지에 의해서 free block을 만들거나 혹은 앞뒤의 free block과 합쳐서 하나의 free block으로 만들거나 하는 등 여러 가지 경우의 수가 있다.

1) ptr == NULL
    => mm_malloc 함수를 부르면 된다.
2) size == 0
    => mm_free 함수를 부르고 return NULL을 한다.
3) 만약 이전 size와 재할당하는 size가 같은 경우
    => 이전 ptr을 그대로 return 한다.

4) 사이즈가 줄어드는 경우

만약 재할당 시에 Header와 Footer를 DSIZE의 공간이 없는 경우에는 Heap 영역이 overwrap 되지 않도록 해야 하므로, 재할당을 할 수 없다. 충분한 공간이 있는 경우 재할당을 위한 size + DSIZE 만큼을 allocate해준다. 그리고 realloc되었다는 표시를 해준다.

```
if(asize < csize) {
    padding = csize - asize;
    if(padding <= DSIZE) {
        return oldptr;
    }
    PUT(HDRP(oldptr), PACK(asize+DSIZE, 1));
    PUT(FTRP(oldptr), PACK(asize+DSIZE, 1));
    newptr = oldptr;
    realloced = 1;
}
```

5) 사이즈가 커지는 경우

realloc을 할 수 있는 범위는 3가지로 나눌 수 있다. 1) current block + next block 2) previous block + current block 3) previous block + current block + next blcok. 이 3가지 범위에 할당하기 위해서는 또 조건이 필요하다. 먼저 next 혹은 previous block의 해당 pointer가 유효해야 하고, allocated되지 않은 free block이어야 메모리를 재할당할 수 있다. 더불어서 가능한 block 조합의 사이즈가 realloc하고 싶은 사이즈와 같거나 크다면 메모리를 realloc 할 수 있다. delete_free_block을 이용해서 alloc할 수 있는 부분의 free block을 삭제하고, 새로운 size를 구한다음 PUT을 이용해서 realloc해준다.

# previous + currernt block에 realloc 가능

```
/* if previous block can be merged */
if(prev_ptr!= NULL && !GET_ALLOC(HDRP(prev_ptr))) {
    if(psize + csize >= asize)
    {
        padding = csize + psize - asize;
        delete_free_block(prev_ptr);
        new_size = (padding <= DSIZE) ?
                    csize+psize+DSIZE : asize+DSIZE;
        newptr = prev_ptr;
        /* realloc with new size */
        PUT(HDRP(newptr), PACK(new_size, 1));
        memmove(newptr, oldptr, csize+DSIZE);
        PUT(FTRP(newptr), PACK(new_size, 1));
        realloced = 1;
    }
}
```

# next + current block에 realloc 가능

```
/* if next block can be merged */
else if(next_ptr != NULL && !GET_ALLOC(HDRP(next_ptr))) {
    if(nsize + csize >= asize) {
        padding = csize + nsize - asize;
        delete_free_block(next_ptr);
        new_size = (padding <= DSIZE) ?
                    csize+nsize+DSIZE : asize+DSIZE;
        /* realloc with new size */
        PUT(HDRP(oldptr), PACK(new_size, 1));
        PUT(FTRP(oldptr), PACK(new_size, 1));
        newptr = oldptr;
        realloced = 1;
    }
}
```

# previous + current + next blcok에 realloc 가능

```
/* if the sum of previous, next and current block size satisfies
 * the new size, then they can be merged and realloced */
if(!GET_ALLOC(HDRP(prev_ptr)) && !GET_ALLOC(HDRP(next_ptr))) {
    if(psize + nsize + csize >= asize)
    {
        padding = csize + psize + nsize - asize;
        delete_free_block(next_ptr);
        delete_free_block(prev_ptr);
        new_size = (padding<=DSIZE) ?
                    csize+psize+nsize+DSIZE : asize+DSIZE;
        newptr = prev_ptr;
        /* realloc with new size */
        PUT(HDRP(newptr), PACK(new_size, 1));
        memmove(newptr, oldptr, csize+DSIZE);
        PUT(FTRP(newptr), PACK(new_size, 1));
        realloced = 1;
    }
}
```

current block에서 previous block을 합치는 경우에는 시작점이 달라지므로, mem_move(newptr, oldptr, csize+DSIZE)를 통해서 oldptr의 값을 newptr로 oldptr의 사이즈만큼 옮겨주는 과정이 필요하다. memory overwrapping을 막기 위해서 메모리를 다 옮긴 후에 footter를 할당하는 과정이 필요하다.

* 메모리를 재할당한 이후에는 padding에 해당하는 free block을 add_free_block을 통해 처리해주어야 한다. padding을 만드는 이유는 메모리를 좀 더 절약해서 사용하기 위해서이고, 더 효율적으로 heap에 메모리를 할당하고 해제 할 수 있다.

```
/* For the case of success of REALLOC */
if(realloced) {
    void * ret = NULL;
    /* if the remaining size after the realloc is bigger t
     * size then the remaining block can be free block */
    if(padding > DSIZE) {
        ret = NEXT_BLKP(newptr);
        PUT(HDRP(ret), PACK(padding, 0));
        PUT(FTRP(ret), PACK(padding, 0));
        add_free_block(ret, GET_BLK_SIZE(ret));
        coalesce(ret);
    }
}
```

6) 메모리를 할당하지 못한 경우 == 완전히 새로운 곳에 메모리를 realloc 해야하는 경우

mm_malloc과 mm_free를 이용해서 완전히 새로운 곳에 block을 할당하는 과정이 필요하다. 이때, 할당에 실패하면 NULL을 return하게 되고, 성공하면 이전 값을 복사하는 과정이 필요하다.

```
/* For the case of Failure of REALLOC */
else {
    newptr = mm_malloc(size);
    if(newptr == NULL)
        return NULL;
    memcpy(newptr, oldptr, csize+DSIZE);
    mm_free(oldptr);
}
```

## 3. Difficulties

기본적으로 이번 lab은 상당한 난이도로 인해 시간이 오래 소요되었다. 우선 과제를 이해하기 위해 교과서와 ppt를 꼼꼼히 공부했음에도 불구하고, 직접 구현하는 것은 많은 노력이 필요했다. 더불어서 각 implicit allocator와 explicit allocator 그리고 segregated list 까지 다양한 방법이 존재하기 때문에, 여러 방식으로 시도해 보며 Space Utilization과 Throughput 사이에서 어떤 방법이 더 중요한지를 생각해보았다. 다행스럽게도, 교재에 기본적인 skeleton code가 주어져 있었기 때문에 이를 기반으로 먼저 코드의 흐름을 생각해볼 수 있었고, 마지막으로 어떤 방식으로 짜야하는지 그 흐름을 정리하고 코드를 짰기 때문에 보다 시간을 절약할 수 있었다. 그럼에도 realloc 파트 그리고 performance efficiency를 위한 place 함수의 수정 등에서 많은 시간을 소요했고, 가장 어려운 부분이었다.

<Pointer>

pointer에 대한 어려움이 사실 가장 컸다. pointer의 경우 잘못 사용할 때, segmentation fault가 발생하게 때문에 error가 발생하는 경우 이를 수정하기 위해 시간을 많이 소요했다. 따라서 중반에 포인터를 다시 제대로 이해하고 코드를 뒤엎는 경우도 있었다. 하지만 이러한 과정 덕분에 조금은 포인터를 잘 이해하게 되었다.

<Realloc>

realloc의 경우에는 고려해야 하는 경우가 많았다. 단순히 malloc과 free를 하고, memcpy를 하는 경우에는 performance 측면에서 비효율적이었기 때문에 우선적으로 아낄 수 있는 block 즉, free block으로 split 할 수 있는지를 고려해야했다. 더불어서 만약 새로운 size가 원래 size보다 큰 경우에는 다시 find_fit을 통해서 알맞은 block을 찾아야 하지만, 일단 근접해 있는 previous block과 next block이 freed block인지 alloced block인지에 따라서 가능한 경우의 수가 다르기 때문에 그리고 합칠 수 있는 부분과 free list에 넣고 빼고 하는 과정에서 코드를 짜면서도 많이 헷갈렸다. 이후 논리적으로 정리를 하고 순차적으로 구현했다.

<Performance Efficiency>

각 trace 별로 어떻게 구현하는지에 따라서 performance 값이 변화했다. 따라서 Space Utilization과 Throughput을 적절히 balance를 맞춰서 구현하는 것이 이번 과제에서 가장 어려웠다. 단적으로 realloc을 구현할 때 만약 next block이랑만 합치는 경우를 고려한다면, realloc trace 파일에서 현저히 낮은 Space Utilization 값을 가졌으나 만약, previous block과 함께 합쳐질 수 있는 가능성을 염두에 두는 경우 높은 Space Utilization 값을 가지는 것을 알 수 있었다. 더불어 CHUNCKSIZE의 값에 의해서도 binary trace 파일과 realloc trace 파일의 performance 값이 영향을 받는 것을 알 수 있었다. 또한, Throughput에 비해서 Space Utilization의 경우를 고려하는 것이 더 어려웠는데, place function에서 split 하여 alloc 그리고 dealloc을 하는 과정에서 if, else if, 그리고 else로 나누는 그 과정에서도 performance 값이 천차만별로 달라지기 때문에 binary, realloc 중에 어느 파일을 고려했는지에 의해서도 각각 utility percent가 달라지는 것을 알 수 있었다. 결국 마지막까지 binary2.rep의 경우에는 utility가 53%가 나왔으나 더 이상의 optimization을 이끌어내지 못했고, 이를 제외한 다른 파일들의 경우에는 utility가 90%이상이 나오도록 구현했다.

4. Performance Results => TOTAL 95 / 100

```
user23@SystemProgramming:~/malloclab-handout/src$ ./mdriver -l -v
Using default tracefiles in ./traces/
Measuring performance with gettimeofday().

Results for libc malloc:
trace  valid  util     ops      secs  Kops
 0       yes    0%    5694  0.001958  2908
 1       yes    0%    5848  0.001749  3343
 2       yes    0%    6648  0.002811  2365
 3       yes    0%    5380  0.002888  1863
 4       yes    0%   14400  0.000824 17478
 5       yes    0%    4800  0.004917   976
 6       yes    0%    4800  0.004792  1002
 7       yes    0%   12000  0.002195  5467
 8       yes    0%   24000  0.002149 11170
 9       yes    0%   14401  0.000649 22179
10       yes    0%   14401  0.000334 43104
Total           0%  112372  0.025266  4448

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.000424 13417
 1       yes   98%    5848  0.000434 13468
 2       yes   99%    6648  0.000506 13136
 3       yes   99%    5380  0.000410 13132
 4       yes   98%   14400  0.000674 21368
 5       yes   93%    4800  0.000546  8790
 6       yes   90%    4800  0.000575  8351
 7       yes   96%   12000  0.000603 19900
 8       yes   53%   24000  0.006725  3569
 9       yes   81%   14401  0.000702 20500
10       yes   97%   14401  0.000878 16404
Total           91%  112372  0.012477  9006

Perf index = 55 (util) + 40 (thru) = 95/100
user23@SystemProgramming:~/malloclab-handout/src$
```