Multicore Programming Project 3

담당 교수 : 박성용 교수님

이름 : 장준영

학번 : 20211584

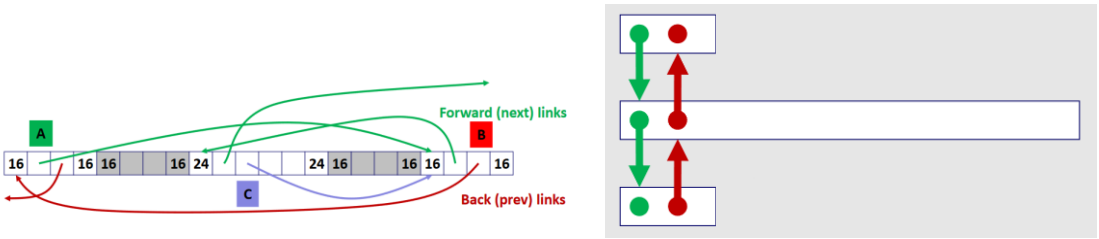
0. Overview

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.003412  1669
 1       yes   99%    5848  0.002463  2375
 2       yes   99%    6648  0.004126  1611
 3       yes   99%    5380  0.003799  1416
 4       yes   99%   14400  0.000401 35919
 5       yes   95%    4800  0.001925  2494
 6       yes   95%    4800  0.001876  2559
 7       yes   95%   12000  0.020781   577
 8       yes   88%   24000  0.015645  1534
 9       yes   99%   14401  0.000210 68544
10       yes   76%   14401  0.000230 62531
Total          95%  112372  0.054868  2048

Perf index = 57 (util) + 40 (thru) = 97/100
cse20211584@cspro:~$
```

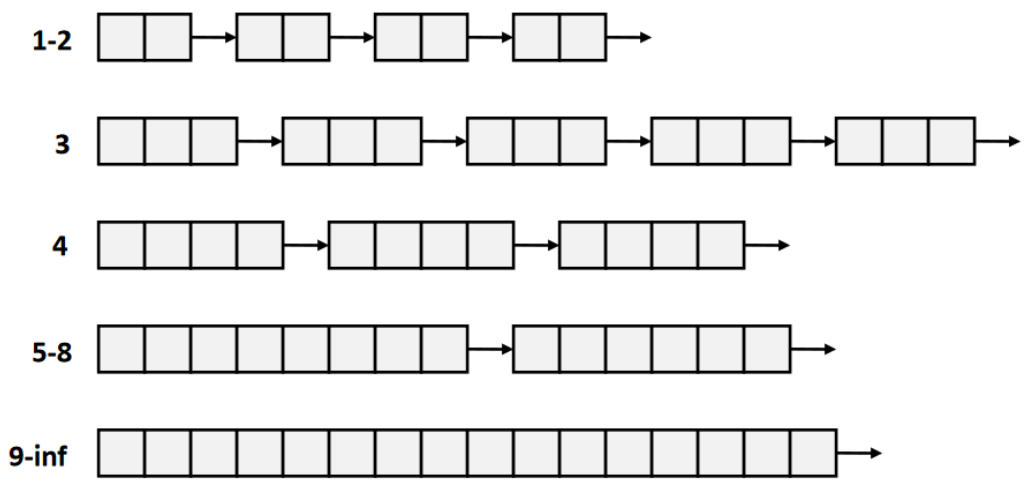
이번 프로젝트에서는 강의 시간에 학습한 정보를 바탕으로 libc가 제공하는 malloc, realloc, free와 같은 Dynamic memory allocator를 직접 구현해보았다. ./mdriver -V의 출력 결과로 97의 Performance index를 받았다. 높은 점수를 위해 배운 방식 중 가장 효율이 좋은 Segregated Free List 방식으로 구현하였다.

1. Depiction of My Allocator (Segregated Free List)

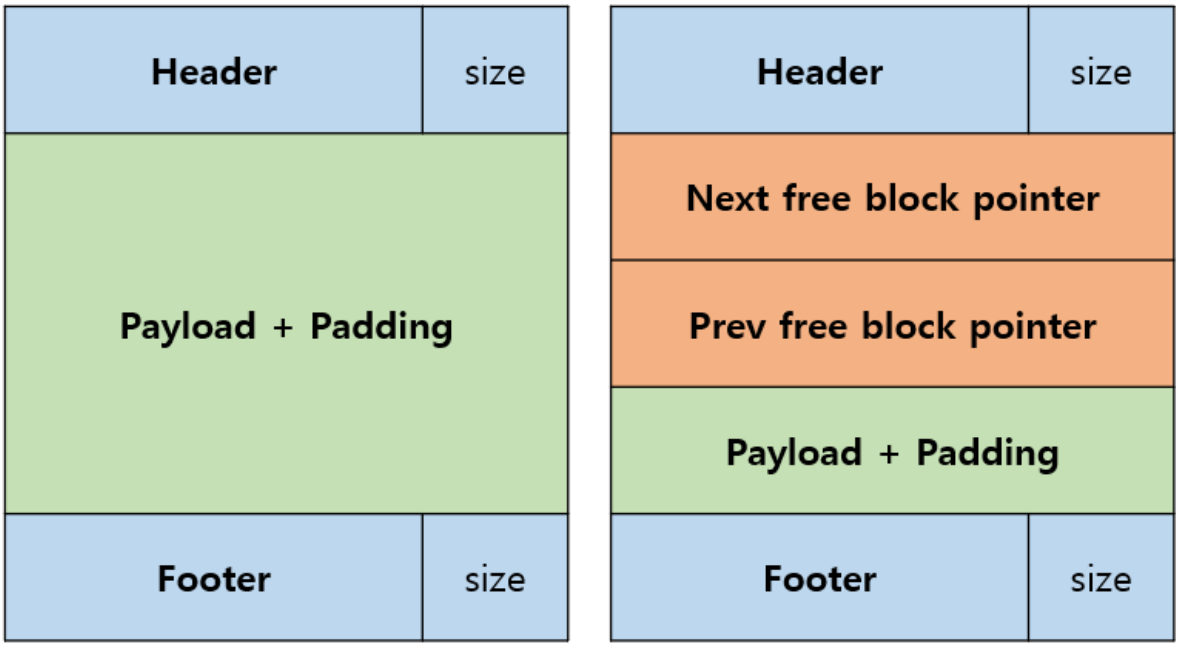
Segragated Free List(이하 seg list)는 앞서 배운 Explicit Free List를 개선한 방식으로, Free block을 크기에 따라 class 별로 나누고, linked list로 연결해둔 것이다.



위 사진은 Explicit Free List의 형태를 표현한 그림이다. 할당되지 않은 free block을 Next, Prev 포인터를 이용해 서로 연결해주는 방식이다. 이렇게 free block을 서로 연결하되, 그 크기에 따라 분류하여 allocation을 용이하게 하면 다음과 같다.



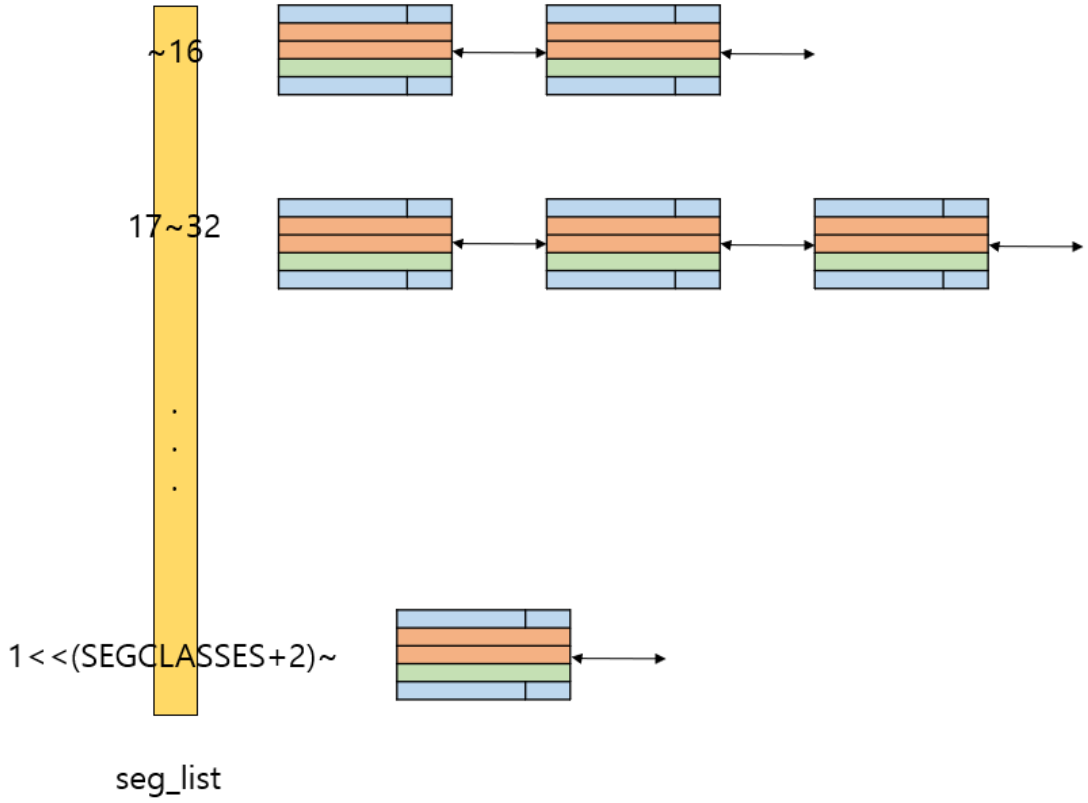
이런 식으로 class를 나누면, 다른 방식과 달리 best fit search를 위해 전체 heap space에 대한 traversal이 필요하지 않다. 원하는 크기를 담을 수 있는 class에 존재하는 free block에 대해서만 탐색하면 되기 때문이다. 만약 해당 이미지의 class에 대해 4 만큼 할당하고 싶다면, 4, 5-8, 9-inf class만 탐색하면 된다. 이렇게 하면 best fit search를 하기 때문에 space utilization도 좋아지고, 탐색 연산량이 줄어들기 때문에 throughput도 높아진다.



따라서 block을 위와 같이 구성할 수 있다. 좌측은 allocated block으로, header와 footer를 포함한 기본적인 block이다. 우측은 free block으로, free 되어서 더 이상 사용하지 않는 내부 공간에 앞 뒤의 free block pointer(1 word)를 포함한다. (4 word의 기본 할당 block이 필수적이다.)

seg list 탐색은 한 방향으로만 진행될텐데, 굳이 pointer를 두 개나 사용할 필요가 있을까?

>> header, footer, payload가 포함된 최소 크기의 allocated block이 어차피 4 words의 크기를 갖기 때문에, pop, push 연산 등의 용이함을 위해 prev pointer를 포함하는게 좋다.



내 코드의 seg list는 다음과 같은 형태이다. 16부터 시작해 2를 곱해 나가며 1<<(크기+2)까지 나누었다. seg list의 포인터는 free block이 연결된 linked list의 처음 노드를 가리킨다.

2. My Code

(※ 수업 시간에 교수님이 libc malloc의 작동 방식에 대해 가볍게 말씀해주신 내용을 떠올렸다. seg list의 모든 class는 2의 제곱수를 기준으로 나뉘어있고, allocate 하는 크기는 block의 절반을 기준으로 채워 split 하거나 하지 않는다는 내용에서 힌트를 얻어 seg list기반 allocator를 구현하였다.)

(0) Macros, Inline functions, Global variables

교재에도 나와있듯, 복잡한 비트 연산과 포인터 연산의 readability를 높이기 위해 macro를 상당히 많이 정의해야 한다. 개인적인 성향으로 macro보단 inline 함수를 사용하는 것을 선호하기 때문에, lvalue 문제가 일어나지 않는 한 대부분의 함수를 inline 함

수로 고쳐서 사용했다.

```
#define GET(p)  (*(size_t *)(p))

static inline void PUT(void *p, size_t val){
    (*(size_t *)p) = val;
}
```

(GET : read a word at address p, PUT : write a word at address p)

```
/*** Global Variables ***/
static char *heap_listp;
static char **seg_list;
```

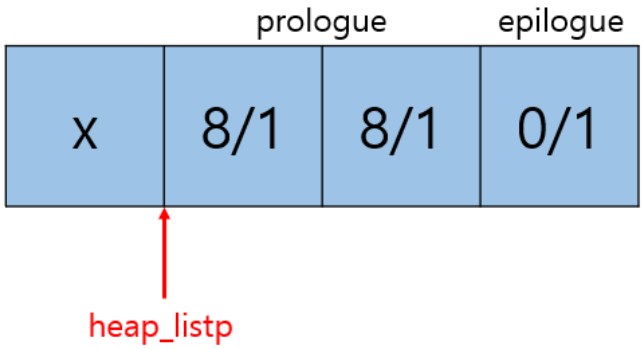
두 개의 Global variable을 선언하였다. heap_listp는 교재와 동일하게 heap의 시작 위치를 가리키는 pointer이고, seg_list는 seg list이다. seg list는 pointer에 대한 배열로 사용할 것이기 때문에 double pointer로 선언한다. 배열로 선언하지 않은 이유는 후술한다.

(1) mm_init

```
int mm_init() {
    /* INIT SEG LIST */
    if ((seg_list = extend_seg(SEGCLASSES * WSIZE)) == (void*)-1)
        return -1;
    for (int i = 0; i < SEGCLASSES; i++)
        *(seg_list + i) = NULL;

    /* INIT HEAP */
    if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void*)-1)
        return -1;

    PUT(heap_listp, 0x0);                               // X
    PUT(heap_listp + 1 * WSIZE, PACK(DSIZE, 0x1));      // 8/1
    PUT(heap_listp + 2 * WSIZE, PACK(DSIZE, 0x1));      // 8/1
    PUT(heap_listp + 3 * WSIZE, PACK(0, 0x1));          // 0/1
    heap_listp += (WSIZE * 2);

    //if (extend_heap(CHUNKSIZE / WSIZE) == NULL)
    //  return -1;

    return 0;
}
```

seg list 초기화 부분을 제외하면 교재의 예시 코드와 전혀 다르지 않은 mm_init 함수

이다. sbrk를 통해 4 words 만큼 힙을 확장하고, 아래 사진처럼 요소를 집어 넣고 heap_listp를 설정한다.



처음 나온 subroutine인 extend_seg 함수는 다음과 같다.

```
static void *extend_seg(size_t asize) {
    size_t incr;
    void *ptr;

    if(asize > SEGCHUNKSIZE)
        asize /= WSIZE;
    else
        asize = SEGCHUNKSIZE / WSIZE;
    incr = (asize + (asize % 2)) * WSIZE;

    if ((ptr = mem_sbrk(incr)) == (void*)-1)
        return NULL;

    PUT(HDRP(ptr), PACK(incr, 0x0));
    PUT(FTRP(ptr), PACK(incr, 0x0));
    PUT(HDRP(NEXT_BLKP(ptr)), PACK(0, 0x1));

    return ptr;
}
```

```
static void *extend_heap(size_t asize) {
    size_t incr;
    void *ptr;

    if(asize > CHUNKSIZE)
        asize /= WSIZE;
    else
        asize = CHUNKSIZE / WSIZE;
    incr = (asize + (asize % 2)) * WSIZE;

    if ((ptr = mem_sbrk(incr)) == (void*)-1)
        return NULL;

    PUT(HDRP(ptr), PACK(incr, 0x0));
    PUT(FTRP(ptr), PACK(incr, 0x0));
    PUT(HDRP(NEXT_BLKP(ptr)), PACK(0, 0x1));

    return ptr;
}
```

extend_seg는 mm_malloc에서 heap의 크기가 모자랄 때 확장하기 위해 사용하는 extend_heap과 완전히 동일하지만, 크기가 쓸 데 없이 커지는 것을 방지하기 위해 CHUNKSIZE를 SEGCHUNKSIZE로 변경하였다. 이를 통해 seg list를 배열로 할당하지 않고, heap 영역에 할당하여 사용할 수 있다. 프로젝트 규칙 상 global array를 사용해선 안되므로, 다음과 같은 방식을 선택하였다. (extend_heap의 자세한 설명은 후술한다.) 해당 함수를 통해 heap 영역에 seg list pointer를 위한 공간을 마련하고, 모두 NULL로 초기화 한다.

(2) mm_malloc

mm_malloc은 크기를 할당하는 allocation, 남은 공간을 처리하는 split의 과정을 거친다.

```
void *mm_malloc(size_t size) {
    if (size == 0)
        return NULL;

    /* ALLOCATION */
    size_t asize;
    void *ptr;

    asize = ALIGN(size + DSIZE);

    if ((ptr = best_fit(asize)))                    // Best Fit Search
        seg_pop(ptr);
    else{
        if ((ptr = extend_heap(asize)) == NULL)     // No left heap space >> Increment
            return -1;
    }
```

할당하고자 하는 크기로 입력받은 size를 double word alignment에 맞춘 것이 asize이다. asize를 cover 할 수 있는 free block이 있는지 seg list에서 찾는 best_fit 함수를 호출한다. 만약 존재한다면 seg_pop을 통해 seg list에서 해당 block을 제거하고, 존재하지 않는다면 extend_heap을 통해 heap의 크기를 asize만큼 늘린다. 만약 heap을 더 이상 늘릴 수 없다면 종료한다.

```
static void *best_fit(size_t asize) {
    void *best = NULL;
    size_t best_size = ULLONG_MAX;

    for (int i = class_num(asize); i < SEGCLASSES; i++) {                                                   // Check every possible classes
        for (void *node = seg_list[i]; node; node = SEG_NEXT(node)) {                                       // Check every possible nodes
            if (!GET_ALLOC(HDRP(node)) && GET_SIZE(HDRP(node)) >= asize && GET_SIZE(HDRP(node)) < best_size) {  // Find minimal cover for asize
                best = node;
                best_size = GET_SIZE(HDRP(node));
            }
        }
    }

    return best;
}
```

best_fit 함수에선 asize가 포함되는 class 중 가장 작은 부분에서 시작해, 가장 큰 부분까지 탐색한다. class의 모든 node를 탐색하면서, [1]block이 free 상태이고, [2]크기가 asize보다 크거나 같고, [3]현재의 best_size보다 작을 경우 갱신하는 형식으로 가장 작게 cover 할 수 있는 node를 찾아내 return 한다. 이렇게 하면 다른 방식보다 훨씬 적은 탐색으로 최고의 block을 찾아낼 수 있다.

```
static void seg_push(void *node) {              // LIFO (Push at first of linked list)
    void *first_node = seg_list[class_num(GET_SIZE(HDRP(node)))];

    GET_SEG_PREV(node) = (size_t)NULL;
    GET_SEG_NEXT(node) = (size_t)first_node;

    if (first_node)
        GET_SEG_PREV(first_node) = (size_t)node;

    seg_list[class_num(GET_SIZE(HDRP(node)))] = node;
}

static void seg_pop(void *node) {               // LIFO
    if (SEG_PREV(node))
        GET_SEG_NEXT(SEG_PREV(node)) = (size_t)SEG_NEXT(node);
    else
        seg_list[class_num(GET_SIZE(HDRP(node)))] = SEG_NEXT(node);

    if (SEG_NEXT(node) != NULL)
        GET_SEG_PREV(SEG_NEXT(node)) = (size_t)SEG_PREV(node);
}
```

seg_push, seg_pop 함수는 linked list에 새로운 노드를 삽입하는 아주 기초적인 함수이다. push를 할 땐 삽입하고자 하는 노드의 앞은 NULL, 뒤는 원래의 첫 노드로 설정한다. 이후 첫 노드가 NULL이 아니라면 앞을 삽입하고자 하는 노드로 변경한다. 이후 맨 앞에 삽입하고자 하는 노드를 넣으면 LIFO 방식으로 push를 구현할 수 있다. pop도 마찬가지이다. pop 하고자 하는 노드 앞 뒤를 연결하기만 하면 된다.

왜 LIFO로 하는 것이 좋을까?

>> 여러 경우에서 최근 free 된 block을 사용하는 것이 좋은 성능을 보인다고 한다. 하지만 나의 코드에선 best fit을 사용하기 때문에 큰 이점은 없고, 가장 적은 pointer와 traversal로 노드 삽입이 가능하기 때문에 사용하였다.

```
static int class_num(size_t size) {
    int classNum = 0;
    int thresh = (1 << (SEGCLASSES + 2));

    while (classNum < SEGCLASSES - 1 && size > thresh) {
        classNum++;
        thresh >>= 2;
    }

    return classNum;
}
```

class_num 함수는 class 중 size를 cover 할 수 있는 최소를 return 하는 함수이다. 1<<(SEGCLLASSES+2)부터 최대 SEGCLASSES-1번의 과정을 거쳐 0부터 SEGCLASSES-1까지의 class 중 하나가 return된다.

```c
static void *extend_heap(size_t asize) {
    size_t incr;
    void *ptr;

    if(asize > CHUNKSIZE)
        asize /= WSIZE;
    else
        asize = CHUNKSIZE / WSIZE;
    incr = (asize + (asize % 2)) * WSIZE;

    if ((ptr = mem_sbrk(incr)) == (void*)-1)
        return NULL;

    PUT(HDRP(ptr), PACK(incr, 0x0));
    PUT(FTRP(ptr), PACK(incr, 0x0));
    PUT(HDRP(NEXT_BLKP(ptr)), PACK(0, 0x1));

    return ptr;
}
```

앞서 extend_seg같이 heap 공간을 키우는 mem_sbrk의 일종의 wrapper 함수이다. 기본적으로 CHUNKSIZE 이상을 double word alignment에 맞게 확장한다. 이후, 기존에 없던 확장된 block의 맨 앞과 뒤에 size를 담은 header, footer를 넣고, epilogue block을 설정한다.

```c
    /* SPLIT */
    size_t fsize, rsize;
    void *free_ptr;

    fsize = GET_SIZE(HDRP(ptr));
    rsize = fsize - asize;

    if (rsize >= 4 * WSIZE) {                       // | H | N | P | F | Pointers (min)
        if (asize < 100)                            // Optimal Threshold (100) (Trick!)
            free_ptr = &(*(ptr + asize));
        else {
            free_ptr = ptr;
            ptr = &(*(ptr + rsize));
        }

        PUT(HDRP(ptr), PACK(asize, 0x1));
        PUT(FTRP(ptr), PACK(asize, 0x1));

        PUT(HDRP(free_ptr), PACK(rsize, 0x0));
        PUT(FTRP(free_ptr), PACK(rsize, 0x0));
        mm_free(free_ptr);
    }
    else {
        PUT(HDRP(ptr), PACK(fsize, 0x1));
        PUT(FTRP(ptr), PACK(fsize, 0x1));
    }

    return ptr;
}
```

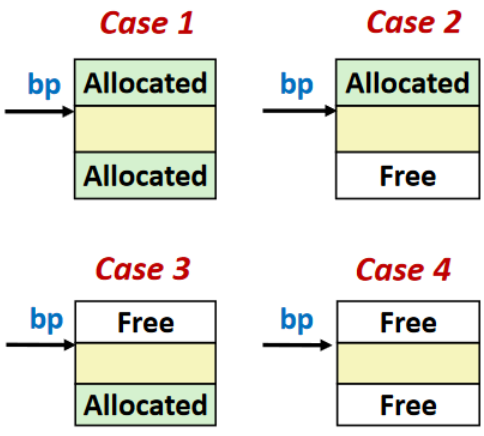
다음으로 split 과정이다. allocation 이후, rsize 만큼의 남는 block이 생긴다. 그 rsize가 Header, Next, Prev, Footer 네 개의 pointer를 담을 수 있는 최소 크기를 만족한다면 split을 진행한다. 만약 만족하지 않는다면 그냥 남은 block까지 padding으로 할당한다. 만족하는 경우에도 두 가지 경우가 나뉘는데, allocate 하는 asize가 100 이상인 경우와 미만인 경우이다. 굳이 이렇게 경우를 나누는 이유는 이후 **3.** 에서 후술하겠다. split할 남은 block의 시작점을 free_ptr로 시작하고, 나뉜 두 블록에 boundary tag를 넣어 split을 진행한다. 마지막으로, free를 통해 split으로 발생한 free block이 다른 free block과 coalesce 할 수 있도록 mm_free를 호출한다.

(3) mm_free

mm_free는 free, 공간 최적화를 위해 free block을 합치는 coalesce, 처리된 free block을 seg list에 넣는 seg push 과정을 거친다.

```
void mm_free(void *ptr) {
    if (ptr == NULL)
        return;

    size_t size = GET_SIZE(HDRP(ptr));

    /* FREE */
    PUT(HDRP(ptr), PACK(size, 0x0));
    PUT(FTRP(ptr), PACK(size, 0x0));

    /* COALESCE */
    int condition;
    if (GET_ALLOC(HDRP(NEXT_BLKP(ptr))) == 1 && GET_ALLOC(HDRP(PREV_BLKP(ptr))) == 1) condition = 0;          // Case 1: 1 1
    else if (GET_ALLOC(HDRP(NEXT_BLKP(ptr))) == 0 && GET_ALLOC(HDRP(PREV_BLKP(ptr))) == 1) condition = 1;     // Case 2: 0 1
    else if (GET_ALLOC(HDRP(NEXT_BLKP(ptr))) == 1 && GET_ALLOC(HDRP(PREV_BLKP(ptr))) == 0) condition = 2;     // Case 3: 1 0
    else if (GET_ALLOC(HDRP(NEXT_BLKP(ptr))) == 0 && GET_ALLOC(HDRP(PREV_BLKP(ptr))) == 0) condition = 3;     // Case 4: 0 0
    ptr = coalesce(ptr, size, condition);

    /* SEG PUSH */
    seg_push(ptr);
}
```

free는 간단하다. header과 footer의 allocation bit만 0으로 바꿔주면 된다. coalesce에는 네 가지 경우가 있는데, 아래 사진과 같다.



Case 1의 경우 합치지 않고, 2에는 다음 block과 합치고, 3에는 이전 block과 합치고, 4에는 앞 뒤로 합쳐야 한다. mm_free에서 앞 뒤 block의 alloc bit를 확인하여 condition(case)를 정하고, coalesce 함수에 pointer, size, condition을 보내준다.

```
static void *coalesce(void *ptr, size_t size, int condition) {
    if (condition == 0)                                     // case 1
        return ptr;

    void *front, *rear;
    void *next = NEXT_BLKP(ptr), *prev = PREV_BLKP(ptr);
    size_t merged_size, next_size = GET_SIZE(HDRP(next)), prev_size = GET_SIZE(HDRP(prev));

    if (condition == 1) {                                   // case 2
        front = ptr;
        rear = next;
        seg_pop(next);
        merged_size = size + next_size;
    }
    else if (condition == 2) {                              // case 3
        front = prev;
        rear = ptr;
        seg_pop(prev);
        merged_size = prev_size + size;
    }
    else if (condition == 3) {                              // case 4
        front = prev;
        seg_pop(prev);
        rear = next;
        seg_pop(next);
        merged_size = prev_size + size + next_size;
    }
    else
        return -1;

    PUT(HDRP(front), PACK(merged_size, 0x0));
    PUT(FTRP(rear), PACK(merged_size, 0x0));

    return front;
}
```

coalesce 함수는 교재의 예제 코드와 동일한 알고리즘을 수행한다. 단순하게 condition에 따라 경우를 나눠 합쳐진 크기를 구하고, 합쳐진 block의 맨 앞이 되는 block에 합쳐진 크기만큼 size 설정을 한다. coalesce까지 마친 완성된 free block을 다시 seg push 하면서 free를 마친다.

(4) mm_realloc

realloc은 세 가지 경우가 있다. [1]0만큼 realloc 하는 경우, [2]앞 뒤의 free block을 통해 cover가 가능한 경우, [3]앞 뒤의 free block을 통해 cover가 불가능한 경우이다.

```c
void *mm_realloc(void *ptr, size_t size) {
    /* 1. ZERO REALLOC == FREE BLOCK */
    if (size == 0) {
        mm_free(ptr);
        return NULL;
    }

    /** REALLOCATION **/
    size_t cur_size, realloc_size;

    cur_size = GET_SIZE(HDRP(ptr));
    realloc_size = ALIGN(size + DSIZE);

    if (realloc_size <= cur_size)                    // Smaller reallocation >> Do nothing!
        return ptr;
```

우선 0만큼 realloc 하는 것은 free와 같다. 따라서, free를 호출하고 NULL을 return 한다. 본격적인 realloc을 위해 현재 block의 크기와 realloc 하고싶은 크기를 저장해둔다. 만약 현재 block이 더 크다면 이미 충분하므로 따로 처리하지 않는다.

```c
    /* 2. COVER WITH PREV OR NEXT BLOCK */
    size_t new_size, merge_prev_size, merge_next_size;
    void *new_block, *bigger_block;
    int bigger_alloc;

    merge_prev_size = GET_SIZE(HDRP(PREV_BLKP(ptr))) + cur_size;
    merge_next_size = GET_SIZE(HDRP(NEXT_BLKP(ptr))) + cur_size;
    cur_size = (cur_size < size) ? cur_size : size;

    new_size = merge_prev_size;                      // Prev + (1)
    new_block = PREV_BLKP(ptr);
    bigger_block = PREV_BLKP(ptr);
    bigger_alloc = GET_ALLOC(HDRP(PREV_BLKP(ptr)));

    if (bigger_alloc == 0 && (new_size - realloc_size) >= DSIZE) {
        seg_pop(bigger_block);
        memmove(new_block, ptr, cur_size);

        PUT(HDRP(new_block), PACK(new_size, 0x1));   // No split!
        PUT(FTRP(new_block), PACK(new_size, 0x1));   // for better utilization

        return new_block;
    }

    new_size = merge_next_size;                      // + Next (2)
    new_block = ptr;
    bigger_block = NEXT_BLKP(ptr);
    bigger_alloc = GET_ALLOC(HDRP(NEXT_BLKP(ptr)));

    if (bigger_alloc == 0 && (new_size - realloc_size) >= DSIZE) {
        seg_pop(bigger_block);
        memmove(new_block, ptr, cur_size);

        PUT(HDRP(new_block), PACK(new_size, 0x1));   // No split!
        PUT(FTRP(new_block), PACK(new_size, 0x1));   // for better utilization

        return new_block;
    }
```

그 다음엔 앞 뒤 block을 통해 cover 할 수 있는지 확인한다. (이는 구현의 문제인데, 이

부분을 구현하는 방법은 다양하다. 앞 뒤 중 더 큰 block 만을 확인하거나, coalesce와 같이 모든 경우의 수를 확인하거나, 다른 빈 block을 연결해주는 등 다양한 방법으로 구현할 수 있다. 하지만, 해당 project 기준으로 방법에 따른 차이가 크지 않았기 때문에, 나의 코드에서는 앞 뒤 block만 확인하는 방식을 사용하였다.) 먼저 앞의 block의 alloc bit가 0이고 합쳤을 때 크기가 realloc size를 cover 할 수 있다면 둘을 합쳐 realloc 한다. 그 다음도 마찬가지로 뒤의 block이 조건을 만족한다면 realloc 한다. 이때, 공간이 남더라도 split 하지 않는다. 이유는 **3.** 에서 후술한다.

```
    /* 3. CANNOT COVER WITH PREV OR NEXT BLOCK >> NEW BLOCK MALLOC */
    if ((new_block = mm_malloc(size)) == NULL)
        return -1;
    memmove(new_block, ptr, cur_size);
    mm_free(ptr);

    return new_block;
}
```

만약 모든 경우에서 재할당이 불가하여 새로운 공간을 할당해야 한다면, malloc으로 새로운 block을 할당받은 후 내용을 memmove한다. 이후 이전 ptr은 free한다. 이렇게 하면 realloc 과정을 마칠 수 있다.

3. Efforts to/Skills for Maximize Score

(1) mm_realloc에서 split을 하지 않는 이유

- 이 프로젝트의 경우 realloc 이후 남은 block을 free하면 할당에 충분하지 않은 작은 free block이 많이 생겨나 오히려 external fragmentation이 늘어나는 경우가 생긴다.

- 또, 보통 상황의 realloc의 경우 한 번 호출한 이후에도 realloc을 다시 호출하는 상황이 많다. 따라서, 빈 공간을 유지해두면 나중에 호출된 realloc에서도 이용할 수 있게 된다. 이 방식을 위해, offset으로 인해 생기는 3 bit 중 alloc bit를 뺀 2 bit를 realloc tag로 사용하기도 한다. 나도 이 방식을 시도해보았는데, 구현이 많이 달라지고 복잡해지지만 성능 향상에 큰 도움이 되지 않아 다시 원래 방식을 사용하기로 하였다.

```
/* Adjust reallocation tag */
static inline size_t REMOVE_RATAG(void* p) {
    return GET(p) & 0x2;
}
static inline size_t SET_RATAG(void* p) {
    return GET(p) | 0x2;
}
```

참고한 다른 코드에선 realloc bit를 위해 다음과 같은 macro를 사용한다. alloc bit는 0x1, realloc bit는 0x2로 사용하는데, 남는 칸을 사용한다고 보면 된다.

(2) global array에서 global pointer로의 전환

```
void *seg_list[SEGCLASSES];
```

처음에는 global pointer array가 global pointer를 array의 크기만큼 선언하는 것이기 때문에 프로젝트 규칙을 위배하지 않는다고 생각했지만, 배열을 선언함으로써 메모리 영역을 할당받는 것이기 때문에 수정해야 된다는 것을 깨달았다. 방법을 고민하다가, 구현하고 있는 heap 영역에 class 들을 최소한의 크기로 cover 할 수 있도록 할당해주기로 하였다. 이를 위해 extend_seg 함수, seg list traversal을 위한 macro 등을 정의하였다.

```
#define SEG_NEXT(bp)        (void *)GET(bp)                         // Next block (seg)
#define SEG_PREV(bp)        (void *)GET((char *)(bp) + WSIZE)       // Prev block (seg)
#define GET_SEG_NEXT(bp)    GET(bp)                                 // GET Next block (seg)
#define GET_SEG_PREV(bp)    GET((char *)(bp) + WSIZE)               // GET Prev block (seg)
                                                                    // Macros (lvalue problem)
```

```
#define WSIZE           4
#define DSIZE           8
#define CHUNKSIZE       (1<<12)
#define SEGCLASSES      8
#define SEGCHUNKSIZE    (DSIZE * SEGCLASSES)
```

seg_list traversal을 위해 정의한 macro로, SEG_*는 포인터를, GET_SEG_*는 값을 의미한다. //

seg list와 관련된 함수의 일관성을 유지하기 위해 SEGCLASSES(CLASS의 개수), SEGCHUNKSIZE(extend_seg에서 사용할 기본 chunk size) 등을 미리 정의하였다.

```
Total           96%  112372  0.006167 18220

Perf index = 58 (util) + 40 (thru) = 98/100
cse20211584@cspro:~$
```

seg list를 배열로 사용하였을 땐 performance index가 98이 나왔었다. 개선 이후 점수가 줄어드는 것을 최소화하기 위해 여러 방법을 사용하였고, 1점만 떨어지도록 구현할 수 있었다.

(3) split 시 biased allocation으로 인한 fragmentation 해소

90점 대 초반에서 점수가 도무지 오르지 않아서 여러 코드를 찾아보다가, split 부분에

서 alloc size에 따라 branching을 하는 코드를 발견하였다. alloc size가 100 이상이 되면 전체 block의 오른쪽에 할당하고 앞 부분을 free block으로 정하고, 그렇지 않으면 일반적으로 왼쪽부터 할당하는 방식이다. 도무지 이렇게 하는 이유를 알 수 없어서 검색을 해보니, biased situation에 의한 fragmentation의 증가를 막기 위한 것임을 알게 되었다.

```
else if (asize >= 100) {
    /* split block */
    PUT(HDRP(ptr), PACK(remainder, 0)); /* Block header */
    PUT(FTRP(ptr), PACK(remainder, 0)); /* Block footer */
    PUT(HDRP(NEXT_BLKP(ptr)), PACK(asize, 1)); /* Next header */
    PUT(FTRP(NEXT_BLKP(ptr)), PACK(asize, 1)); /* Next footer */
    insert_node(ptr, remainder);
    return NEXT_BLKP(ptr);

}

else {
    /* Split block */
    PUT(HDRP(ptr), PACK(asize, 1)); /* Block header */
    PUT(FTRP(ptr), PACK(asize, 1)); /* Block footer */
    PUT(HDRP(NEXT_BLKP(ptr)), PACK(remainder, 0)); /* Next header */
    PUT(FTRP(NEXT_BLKP(ptr)), PACK(remainder, 0)); /* Next footer */
    insert_node(NEXT_BLKP(ptr), remainder);
}
return ptr;
```

참고했던 코드

우리가 사용하는 malloc-lab의 tracefile은 여러 값을 계속해서 malloc 하는데, 작은 값을 계속 나눠가며 한 쪽 방향으로 담다보면 payload에 비해 garbage가 많아져 fragmentation이 증가한다고 한다. 이를 막기 위해 threshold 값을 정해 이를 넘어가는 allocation의 경우 오른쪽부터 채우는 것이다. threshold 값을 조금씩 바꿔가며 테스트를 진행했는데, 100의 값에서 가장 좋은 효율을 보이는 것을 확인하고 100으로 설정했다. (보통 WSIZE의 배수로 threshold를 설정한다고 한다. 어떻게 하나 작동에 문제는 없을 것으로 보이지만 우선 그렇게 했다.)

*** 한 학기 동안 수고 많으셨습니다. ***