CSED211 Lab 12 Report

Malloc Lab: Writing a Dynamic Storage Allocator

20220041 김유진

I. Method

이번 실습은 C언어로 dynamic storage allocator를 구현하는 것을 목표로 한다. 동적 메모리 할당기는 프로세스의 가상 메모리 영역인 heap을 관리하는데, 초기 설정된 heap 영역에서 메모리가 새롭게 할당되거나 해제되기도 한다. 이때 메모리를 효율적으로 사용하기 위해서 메모리 중간 중간에 있는 할당 가능한 블록들, 즉 가용블록들을 찾으며 원하는 크기의 블록을 찾아 할당한다. 만약 가용블록이 없으면 heap을 확장시켜주어 새로운 주소를 할당하게 한다. 이러한 기능들을 할 수 있도록 malloc, free, realloc을 직접 작성해보는 것이 실습 과제이다. 이때, memory utilization과 thoughput을 고려하여 최적화하는 것이 중요하다. 이 utility는 heap 영역을 최소한의 fragmentation이 있도록 데이터를 잘 할당했는지를 나타내는 요소이고 thoughput은 얼마나 빠른 시간동안 주어진 할당 혹은 해제를 해줄 수 있는지를 나타내는 요소이다. utility와 throughput 모두 효율적으로 구현하기 위해서는 가용 블록을 어떻게 관리할 것인지가 중요하며, 경우에 따라 블록을 쪼개거나 합쳐서 가용 블록을 저장하는 것이 필요하다. 본 코드에서는 segregated free list의 방식을 이용하여 구현하였다. 이는 블록의 크기에 따라 각각의 list로 관리해주는 방식이다.

II. Code Explanation

0) 기본 상수 및 매크로

```
/* single word (4) or double word (8) alignment */
#define ALIGNMENT 8

/* rounds up to the nearest multiple of ALIGNMENT */
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)


#define SIZE_T_SIZE (ALIGN(sizeof(size_t)))

/* 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 LISTLIMIT 20 /* Max size of list */
```

```c
#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)

/* Read the size and allocated fields from address p */
#define HDRP(bp) ((char *)(bp) - WSIZE)
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

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

/* Predecessor and successor pointer of free list */
#define PRED_FREE(bp) (*(void**)(bp))
#define SUCC_FREE(bp) (*(void**)(bp + WSIZE))
```

책 교안을 참고하여 작성한 매크로이다. 추가로 segregated free list에서 list를 관리할 수 있도록 PRED_FREE, SUCC_FREE도 정의해주었다.

```c
/* Global variables */
static void *heap_listp;
static void *segreg_list[LISTLIMIT];

/* Additional functions */
static void *extend_heap(size_t words);
static void *coalesce(void *bp);
static void *find_fit(size_t asize);
static void place(void *bp, size_t asize);
static void insert_block(void *bp, size_t size);
static void remove_block(void *bp);

/* mm_check related functions */
static int mm_check(void);
static int check_block(void *ptr);
static int check_segreg_list(void *ptr, size_t size);
```

전역 변수 및 mm_init, mm_malloc, mm_free, mm_realloc, mm_check 구현에 사용되는 함수들이다. mm_check는 구현한 heap공간을 평가하는 함수이다.

1) mm_init

```
/*
 *  * mm_init - initialize the malloc package.
 *   * Explanation:
 *    *  Initializes segregation list and allocate heap.
 *     *  Returns -1 if there is a problem, otherwise 0.
 *      */
int mm_init(void)
{
    /* initialize segregation list */
    int i;
    for(i = 0; i < LISTLIMIT; i++) {
        segreg_list[i] = NULL;
    }
    /* create the initial empty heap */
    if((heap_listp = mem_sbrk(4*WSIZE)) == (void*)-1) {
        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/WSIZE) == NULL) {
        return -1;
    }
    return 0;
}
```

이 함수는 heap 공간과 free block list를 나타낼 segreg_list를 초기화해주는 함수이다. 반복문을 돌며 알맞은 값으로 초기화해주고, header 및 footer도 설정해준다. 만약 초기화 과정에서 에러가 발생하면 -1을 발생시키고 초기화가 잘 마무리되면 0을 리턴한다.

2) mm_malloc

```
/*
 *  * mm_malloc - Allocate a block by incrementing the brk pointer.
 *   *     Always allocate a block whose size is a multiple of the alignment.
 *    * Explanation:
 *     *  Allocates a block by searching for appropriate free block in
segreg_list.
 *      *  If fit is found, it places the block. Otherwise extends the heap.
 *       *  Returns a pointer to allocated memory.
 *        */
```

```
void *mm_malloc(size_t size)
{
    int newsize = ALIGN(size + SIZE_T_SIZE);
    char *bp;
    size_t extendsize;
    /* search the free list */
    if((bp = find_fit(newsize)) != NULL) {
        place(bp, newsize);
        return bp;
    }
    /* no fit found, get more memory and place the block */
    extendsize = MAX(newsize, CHUNKSIZE);
    if((bp = extend_heap(extendsize/WSIZE)) == NULL) {
        return NULL;
    }
    place(bp, newsize);
    return bp;
}
```

newsize 변수에 ALIGN 매크로를 사용하여 요청된 크기를 알맞게 align해주고 segreg_list를 돌며 적절한 free 블록을 찾아서 할당해준다. 이때 place 함수는 할당된 블록에 메모리를 배치하는 작업을 해준다. 만약 적절한 free 블록이 없는 경우에는 heap을 확장하고 블록을 할당해주어야 하므로 extend_heap 함수를 호출한다. 할당된 블록에 메모리를 배치한 후 해당 포인터를 리턴한다.

3) mm_free

```
/*
 *  * mm_free - Freeing a block does nothing.
 *   * Explanation:
 *    *  Free the given block by updating header and footer, then coalesce.
 *     */
void mm_free(void *ptr)
{
    if(ptr == NULL) {
        return;
    }
    size_t size = GET_SIZE(HDRP(ptr));
    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));
    coalesce(ptr);
}
```

이 코드는 동적 메모리 할당 해제를 하는 함수이다. 먼저 인자로 받은 포인터가 NULL이면 아무 작업도 수행하지 않고 반환한다. 그렇지 않으면, GET_SIZE 매—로를 사용하여 주어진 블록의 크기를 가져오고 해당 크기와 상태를 header와 footer에 업데이트해준다. 그리고 coalesce 함수를 호출하여 할당 해제된 블록 주위의 블록들과 병합하여 빈 공간을 효과적으로 활용할 수 있게 한다.

4) mm_realloc

```
/*
 *  * mm_realloc - Implemented simply in terms of mm_malloc and mm_free
 *   * Explanation:
 *    *  Reallocate a block by either extending, splitting, or allocating a
new block.
 *     *  Copy old data if needed.
 *      */
void *mm_realloc(void *ptr, size_t size)
{
    size_t oldsize;
    void *newptr;
    if(size == 0) {
        mm_free(ptr);
        return NULL;
    }
    if(ptr == NULL) {
        return mm_malloc(size);
    }
    size_t curr_size = GET_SIZE(HDRP(ptr));
    size_t total_size = size + 2*WSIZE; /* include header and footer */
    if(size < curr_size - 2*WSIZE) { /* no need to reallocate */
        return ptr;
    }
    void *next = NEXT_BLKP(ptr);
    int next_alloc = GET_ALLOC(HDRP(next));
    size_t next_size = GET_SIZE(HDRP(next));
    /* check if coalescing is possible */
    if(!next_alloc && total_size <= (curr_size + next_size - 2*WSIZE)) {
        size_t copySize = curr_size + next_size;
        remove_block(next);
        PUT(HDRP(ptr), PACK(copySize, 1));
        PUT(FTRP(ptr), PACK(copySize, 1));
        return ptr;
    }
    /* if next block is free, but not enough to coalesce, allocate new block
*/
    if (!next_alloc && next_size == 0) {
        size_t remainder = curr_size + next_size - total_size;
        size_t extendSize = MAX(-remainder, CHUNKSIZE);
        if (remainder < 0) {
            if (extend_heap(extendSize) == NULL) {
                return NULL;
            }
            remainder += extendSize;
        }
        remove_block(next);
```

```
        PUT(HDRP(ptr), PACK(curr_size + remainder, 1));
        PUT(FTRP(ptr), PACK(curr_size + remainder, 1));
        return ptr;
    }
    /* allocate new block */
    newptr = mm_malloc(size);
    if(newptr == NULL) {/* allocation fail */
        return NULL;
    }
    /* copy old data */
    oldsize = curr_size;
    if (size < oldsize) {
        oldsize = size;
    }
    memcpy(newptr, ptr, oldsize);
    mm_free(ptr);
    return newptr;
}
```

이 함수는 인자로 받은 포인터가 가리키는 블록의 크기를 조절하여 realloc을 수행하는데, 이를 위해 기존 데이터를 새로운 메모리에 복사하고, 어떤 경우에는 새로운 블록을 할당해주거나 기존 블록을 확장하는 작업이 필요하다. 먼저 크기가 0이면 메모리를 해제하고 NULL을 반환하고 포인터가 NULL이면 새로운 메모리를 할당해준다. 그 외의 경우들을 처리해주기 위해서 현재 블록의 크기와 header와 footer를 포함한 총 크기를 계산하는데, 재할당이 필요하지 않은 경우에는 현재 포인터를 그대로 반환한다. 그리고 재할당을 해주기 위해서 현재 블록 옆에 있는 다음 블록의 정보를 가져온다. 그리고 이를 조건문으로 coalescing이 가능한지 확인한다. 만약 가능한 경우에는 블록을 병합하고 해당 블록 포인터를 리턴한다. 다음 블록이 할당되어있지 않지만 충분한 크기가 아닌 경우에는 새로운 블록을 할딩해준다. remainder라는 변수를 통해 추가로 할당해주어야 하는 크기를 계산하고 블록을 합쳐준다. 위의 모든 조건문에 해당하지 않은 경우에는 새로운 블록을 할당해주고 이전에 있던 데이터를 복사해주고, 기존 메모리를 할당해준 후 새로운 포인터를 반환한다.

6) extend_heap

```
/*
 *  * extend_heap
 *   * Explanation:
 *    *  Extend the heap by specified number of words, creating a new free
block.
 *     *  Return block pointer of new free block.
 *      *
 *      */
static void *extend_heap(size_t words)
{
```

```
    char *bp;
    size_t size = ALIGN(words * WSIZE);

    /* allocate an even number of words to maintain alignment */
    if((long)(bp = mem_sbrk(size)) == -1) {
        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);
}
```

인자로 받은 words를 통해 heap을 얼만큼 확장할 지 정보를 받는다. 이를 토대로 확장될 heap의 크기를 align해준 값으로 조정하고 mem_sbrk 함수를 사용하여 heap을 확장한다. 이때 bp는 새로 할당된 블록을 가리키는 블록 포인터이다. 그리고 새로 할당된 블록의 header와 footer, epilogue header를 초기화해준 후, 만약에 coalesce가 가능한 경우 coalesce 함수를 호출하여 새로운 블록과 이전 블록이 merge될 수 있도록 한다.

7) coalesce

```
/*
 *  * coalesce
 *   * Explanation:
 *    *  Coalesce a free block with its adjacent free blocks to create larger
free blocks.
 *     *  Return block pointer of coalesced block.
 *      *
 *      */
static void *coalesce(void *bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp))); /* Previous block
footer */
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp))); /* Next block header
*/
    size_t size = GET_SIZE(HDRP(bp));

    if(prev_alloc && next_alloc) { /* prev: alloc, next: alloc */
        insert_block(bp, size);
        return bp;
    }
    else if(prev_alloc && !next_alloc) { /* prev: alloc, next: free */
        remove_block(NEXT_BLKP(bp));
```

```
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    }
    else if(!prev_alloc && next_alloc) { /* prev: free, next: alloc */
        remove_block(PREV_BLKP(bp));
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    else { /* prev: free, next: free */
        remove_block(PREV_BLKP(bp));
        remove_block(NEXT_BLKP(bp));
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    insert_block(bp, size);
    return bp;
}
```

이 함수는 주어진 블록과 인접한 블록들을 병합하여 효율적으로 메모리 공간을 구성할 수 있게 해주는 함수이다. prev_alloc, next_alloc 변수를 통해 주어진 블록의 앞 뒤 블록이 할당되어 있는지 여부를 조사하고 size 변수에는 현재 블록의 크기를 저장한다. 이후로는 크게 4가지 경우를 나누어 블록을 알맞게 조정해주는 작업을 해준다. 첫번째 if문은 주변 블록이 모두 할당된 경우에는 현재 블록을 free block list(segreg_list)에 추가하고 리턴한다. 두번째 else if문은 이전 블록은 할당되어 있고 다음 블록은 비어 있는 경우로 list에 다음 블록을 제거하고 현재 블록을 확장시켜준다. 세번째 else if문은 이전 블록은 비어 있고 다음 블록은 할당되어 있는 경우로 list에 이전 블록을 제거하고 현재 블록을 확장해준다. 마지막 else문은 주변 블록이 모두 비어있는 경우로 list에 주변 블록을 모두 제거하고 현재 블록을 확장해준다. 각 경우에 따라 블록을 확장하고 필요한 경우에는 이전 블록의 포인터를 업데이트해준다. 그리고 최종적으로 만들어진 블록을 list에 추가한 후 해당 블록 포인터를 리턴한다.

8) find_fit

```
/*
 *  * find_fit
 *   * Explanation:
 *    *  Find a free block in the segregated free list that is sufficient to
admit the given size.
 *     *  Return block pointer of the fit if it is found, otherwise NULL.
 *      *
```

```
 *      */
static void *find_fit(size_t asize)
{
    /* first-fit search */
    void *bp;
    int i;
    for (i = 0; i < LISTLIMIT; i++) {
        bp = segreg_list[i];
        while (bp != NULL) {
            size_t block_size = GET_SIZE(HDRP(bp));
            if (asize <= block_size) {
                return bp;
            }
            bp = SUCC_FREE(bp);
        }
    }
    return NULL;
}
```

이 함수는 주어진 크기를 할당할 수 있는 free block을 찾는 함수로 first-fit search를 적용하였다. segreg_list 배열을 탐색하여 적절한 크기의 free block을 찾고, 만약 찾은 블록이 주어진 크기 이상이라면 해당 블록을 반환하고 아니라면 list의 다음 블록으로 이동하여 list를 계속해서 탐색할 수 있게 한다. 모든 segreg_list를 다 탐색한 후에도 찾지 못했다면 NULL을 반환한다.

9) place

```
/*
 *  * place
 *   * Explanation:
 *    *  Place the allocated block in the heap after finding a fit in
segreg_list.
 *     *  Split the block if it is larger than needed.
 *      *
 *      */
static void place(void *bp, size_t asize)
{
    size_t csize = GET_SIZE(HDRP(bp));
    /* rearrange free list */
    remove_block(bp);
    if((csize - asize) >= 2*(SIZE_T_SIZE)) { /* split the block */
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        bp = NEXT_BLKP(bp);
        PUT(HDRP(bp), PACK(csize-asize, 0));
        PUT(FTRP(bp), PACK(csize-asize, 0));
        insert_block(bp, csize - asize);
```

```
    }
    else {
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }
}
```

이 함수는 적절한 블록에 할당된 블록을 배치하는 기능을 하는 함수이다. 먼저 할당된 블록을 가용 블록 리스트에서 제거해준다. 그리고 조건문을 통해 블록의 분할 여부를 결정하는데, 만약 현재 블록의 나머지 부분이 충분히 크면 블록을 분할한다. 할당된 부분을 위한 header와 footer를 설정해주고 남은 부분을 위한 header와 footer도 설정해준다. 분할된 나머지 블록은 free list에 추가해준다. 만약 분할이 필요없는 경우에는 전체 블록을 할당된 블록으로 설정해준다.

10) insert_block

```
/*
 *  * insert_block
 *   * Explanation:
 *    *  Insert free block into segreg_list based on its size.
 *     *
 *     */
static void insert_block(void *bp, size_t size)
{
    int i = 0;
    size_t target_size = size;
    void *target_ptr = NULL;
    void *insert_ptr = NULL;
    /* find the appropriate list in the segregated free list */
    while((i < LISTLIMIT - 1) && (target_size > 1)) {
        target_size >>= 1;
        i++;
    }
    /* find the correct poisition in the list to insert the block */
    target_ptr = segreg_list[i];
    while((target_ptr != NULL) && (size > GET_SIZE(HDRP(target_ptr)))) {
        insert_ptr = target_ptr;
        target_ptr = SUCC_FREE(target_ptr);
    }
    /* update pointers to insert the block in the list */
    if(target_ptr != NULL) {
        if(insert_ptr != NULL) { /* insert btw insert_ptr and target_ptr */
            PRED_FREE(bp) = insert_ptr;
            SUCC_FREE(bp) = target_ptr;
            PRED_FREE(target_ptr) = bp;
            SUCC_FREE(insert_ptr) = bp;
        }
```

```
        else { /* insert at the front of the list */
            PRED_FREE(bp) = NULL;
            SUCC_FREE(bp) = target_ptr;
            PRED_FREE(target_ptr) = bp;
            segreg_list[i] = bp;
        }
    }
    else {
        if(insert_ptr != NULL) { /* insert at the end of the list */
            PRED_FREE(bp) = insert_ptr;
            SUCC_FREE(bp) = NULL;
            SUCC_FREE(insert_ptr) = bp;
        }
        else { /* first block in the list */
            PRED_FREE(bp) = NULL;
            SUCC_FREE(bp) = NULL;
            segreg_list[i] = bp;
        }
    }
    return;
}
```

이 함수는 주어진 크기의 free block을 segreg_list의 적절한 위치에 삽입하는 기능을 수행한다. 먼저 크기에 따라서 어느 list에 넣을지를 판단하기 위해 반복문을 돈다. list 내의 올바른 위치를 찾으면 블록을 리스트에 삽입하기 위해 포인터를 업데이트해준다. target_ptr는 리스트 내에서 적절한 위치를 찾기 위한 포인터이고 insert_ptr는 블록을 삽입할 위치 이전 블록이다. 블록을 삽입할 때에는 포인터들을 업데이트하는데 삽입되는 위치가 어디냐에 따라 알맞게 포인터를 업데이트해주어야 한다.

11) remove_block

```
/*
 *  * remove_block
 *   * Explanation:
 *    *  Remove block from segreg_list.
 *     *
 *     */
static void remove_block(void *bp)
{
    int i = 0;
    size_t size = GET_SIZE(HDRP(bp));
    /* find the appropriate list in the segregated free list */
    while((i < LISTLIMIT - 1) && (size > 1)) {
        size >>= 1;
        i++;
```

```
    }
    if(SUCC_FREE(bp) != NULL) {
        if(PRED_FREE(bp) != NULL) { /* remove block in the middle */
            PRED_FREE(SUCC_FREE(bp)) = PRED_FREE(bp);
            SUCC_FREE(PRED_FREE(bp)) = SUCC_FREE(bp);
        }
        else { /* remove block at the front of the list */
            PRED_FREE(SUCC_FREE(bp)) = NULL;
            segreg_list[i] = SUCC_FREE(bp);
        }
    }
    else {
        if(PRED_FREE(bp) != NULL) { /* remvove block at the end of the list */
            SUCC_FREE(PRED_FREE(bp)) = NULL;
        }
        else { /* when it was the only block in the list, nullify */
            segreg_list[i] = NULL;
        }
    }
    return;
}
```

이 함수는 insert_block과 반대로 주어진 블록을 segreg_list에서 제거하는 역할을 한다. GET_SIZE를 통해 크기를 조사한 후 어느 리스트에 존재하는지를 확인하기 위해 반복문을 돈다. 그리고는 블록을 제거하는 데에 필요한 포인터들을 업데이트하는데 몇가지 경우에 따라 구분된다. 먼저 다음 블록이 존재하는 경우에는 리스트 중간에 있는 블록을 제거하는 경우와 리스트 맨 앞에 있는 블록을 제거하는 경우로 세분화될 수 있다. 만약에 블록의 다음 블록이 없는 경우에는 리스트 맨 뒤의 블록을 제거해야하는 경우와 리스트에 블록이 오직 하나뿐인 경우로 구분된다. 후자는 해당 리스트를 비워주면 된다.

12) check_block

```
/* Heap Consistency Checker */
/*
 *  * check_block
 *   * Explanation:
 *    *  Check alignment and consistency of a single block.
 *     *  Return 1 if the block is consistent, otherwise 0.
 *      *
 *      */
static int check_block(void *bp)
{
    /* Check if the block is properly aligned */
    if((size_t)bp % ALIGNMENT) {
        printf("Error: Block at address %p is not properly aligned\n", bp);
```

```
        return 0;
    }
    /* Check if the header and footer sizes match */
    size_t header_size = GET_SIZE(HDRP(bp));
    size_t footer_size = GET_SIZE(FTRP(bp));
    if (header_size != footer_size) {
        printf("Error: Header and footer size mismatch for block at
address %p\n", bp);
        return 0;
    }
    return 1;
}
```

check_block, check_segreg_list는 heap의 일관성을 확인하기 위한 함수인 mm_ccheck에서 사용될 함수들이다. check_block은 heap 내의 block alignment와 크기에 대한 일관성을 평가한다. 먼저 블록이 올바르게 align되었는지를 확인하고 그렇지 않으면 오류 메시지를 출력하고 0을 반환한다. 그리고 header와 footer의 크기가 일치하는지를 확인하여 만약 그렇지 않다면 마찬가지로 오류메시지를 출력하고 0을 반환한다. 결론적으로 이 함수의 리턴값이 1이면 일관성이 있다라는 지표이고 0이면 그렇지 않다는 것이다.

13) check_segreg_list

```
/*
 *  * check_segreg_list
 *   * Explanation:
 *    *  Check consistency of segreg_list.
 *     *  Return 1 if the list is consistent, otherwise 0.
 *      *
 *      */
static int check_segreg_list(void *ptr, size_t size)
{
    void *prev = NULL;
    for (; ptr != NULL; ptr = SUCC_FREE(ptr)) {
        if (PRED_FREE(ptr) != prev) {
            printf("Error: Predecessor pointer mismatch in segregated list of
size %zu\n", size);
            return 0;
        }
        if (prev != NULL && SUCC_FREE(prev) != ptr) {
            printf("Error: Successor pointer mismatch in segregated list of
size %zu\n", size);
            return 0;
        }
        /* check if block size is within the range */
        size_t block_size = GET_SIZE(HDRP(ptr));
```

```
        if (block_size < size || (size != LISTLIMIT && block_size > (size <<
1) - 1)) {
            printf("Error: Block size in segregated list of size %zu is
incorrect\n", size);
            return 0;
        }
        prev = ptr;
    }
    return 1;
}
```

이 함수는 segreg_list의 일관성을 확인하는 기능을 수행한다. 반복문을 돌면서 list의 블록의 일관성을 확인한다. 오류를 출력하고 0을 반환하는 경우로는 현재 블록의 PRED 블록이 이전 블록과 같지 않거나 이전 블록의 SUCC 블록이 현재 블록이 아닌 경우이다. 이는 list의 free block들이 제대로 구성되어 있는지를 확인한다. 또한 블록 크기가 올바른 범위 내에 있는지를 확인하여 그렇지 않으면 오류를 출력하고 0을 리턴한다. prev를 현재 블록으로 설정해주어 반복문을 돌 수 있게 한다.

14) mm_check

```
/*
 *  * mm_check
 *   * Explanation:
 *    *  Perform a consistency check on entire heap and segreg_list.
 *     *  Return 1 if the heap is consistent, otherwise 0.
 *      *
 *      */
static int mm_check(void)
{
    char *bp = heap_listp;
    /* prologue header consistency */
    if (GET_SIZE(HDRP(heap_listp)) != DSIZE || !GET_ALLOC(HDRP(heap_listp))) {
        printf("Error: Prologue header is inconsistent\n");
        return 0;
    }
    /* first block consistency */
    if (!check_block(heap_listp)) {
        return 0;
    }
    /* check consistency of each blocks in heap */
    int prev_list = 0;
    for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
        if (!check_block(bp)) {
            return 0;
        }
```

```
        int curr_list = check_block(bp);
        /* not coalesced */
        if (prev_list && curr_list) {
            printf("Error: Uncoalesced free blocks\n");
            return 0;
        }
    }
    /* check consistency of each segreg_list */
    int i;
    size_t size = 1;
    for (i = 0; i < LISTLIMIT; i++) {
        if (!check_segreg_list(segreg_list[i], size)) {
            return 0;
        }
        size <<= 1;
    }
    /* epilogue header consistency */
    if (GET_SIZE(HDRP(bp)) != 0 || !GET_ALLOC(HDRP(bp))) {
        printf("Error: Epilogue header is inconsistent\n");
        return 0;
    }
    return 1;
}
```

이 함수는 check_block, check_segreg_list 함수를 이용하여 heap 전체 및 free block list의 일관성을 확인하는 함수이다. bp를 heap의 시작지점으로 설정해주고 prologue header의 일관성을 확인하고, 첫번째 블록의 일관성을 확인한다. 그다음으로는 heap 내의 다른 블록들의 일관성을 확인하고 free block이 알맞게 coalesce되었는지도 확인하여 이 과정에서 그렇지 않은 것이 있다면 0을 리턴하게 한다. 그리고 segreg_list의 일관성을 확인해주기 위해 반복문을 돌며 check_segreg_list 함수를 통해 이를 확인한다. 마지막으로 epilogue header의 일관성도 확인하여 결론적으로 heap이 consistent하면 1을, 그렇지 않으면 0을 리턴하게 한다.

III. Result

```
[kimyujin1224@programming2 malloclab-handout]$ ./mdriver -v
Using default tracefiles in ./traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.000281 20235
 1       yes   99%    4805  0.000231 20774
 2       yes   55%   12000  0.016750   716
 3       yes   55%    8000  0.016447   486
 4       yes   51%   24000  0.059367   404
 5       yes   51%   16000  0.058604   273
 6       yes   99%    5848  0.000262 22295
 7       yes   99%    5032  0.000220 22862
 8       yes   66%   14400  0.000334 43101
 9       yes   66%   14400  0.000312 46183
10       yes   99%    6648  0.000318 20939
11       yes   99%    5683  0.000264 21510
12       yes  100%    5380  0.000256 20991
13       yes  100%    4537  0.000222 20428
14       yes   96%    4800  0.001832  2621
15       yes   96%    4800  0.001918  2503
16       yes   95%    4800  0.001791  2680
17       yes   95%    4800  0.001803  2662
18       yes   87%   14401  0.000209 69036
19       yes   87%   14401  0.000219 65788
20       yes   52%   14401  0.000165 87226
21       yes   52%   14401  0.000165 87544
22       yes   66%      12  0.000000 24000
23       yes   66%      12  0.000000 30000
24       yes   89%      12  0.000000 24000
25       yes   89%      12  0.000000 24000
Total          81%  209279  0.161970  1292

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

목표 점수까지 도달하지 못했는데 binary 및 binary2 파일 등의 trace에서 약 50%의 utility를 보이고 있음을 확인할 수 있었다. 실제로 이를 개선할 방법을 찾지는 못했지만 추측컨대 fit을 찾는 방법을 조정하거나 realloc하는 과정에서 블록 크기 조절 방식을 더 개선하는 방법이 있을 것이라고 예상한다. 또한 주어진 trace의 특성을 반영하지 않았는데, 각 trace의 특성에 알맞게 경우를 더 세분화하여 알고리즘을 최적화했다면 utility를 향상시킬 수 있었을 것이다.