System Programming Project 4

담당 교수 : 김영재

이름 : 김태규

학번 : 20191243

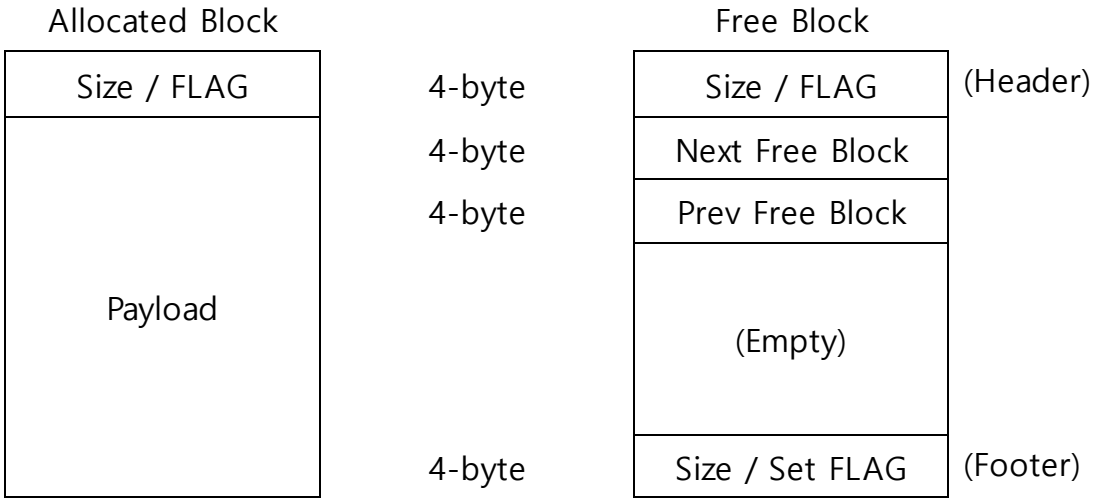
1. 개발 목표

이번 프로젝트의 목표는 libc 라이브러리의 malloc, realloc, free와 동일한 기능을 수행하는 Dynamic Memory Allocator를 구현하는 것에 있다.

본 프로젝트는 implicit list, explicit list, segregated list 기법 중 segregated list 기법을 구현하여 memory allocator 함수를 구현하였다.

2. 구현 방법

이번 프로젝트는 segregated free list를 사용하여 메모리 블록의 할당 및 해제를 구현하였다. 또한 block size와 할당 flag를 나타내는 데 4바이트를 사용한다. 할당된 블록은 4바이트의 header 뒤에 payload가 따르고, footer는 없다. 반면, free block은 header, next free block, prev free block, footer의 4개의 메타데이터를 가진다. header를 제외한 메타데이터는 payload로 할당된 공간을 사용한다. prev_block이라는 함수는 블록의 footer가 유효한지, 즉 블록이 해제된 블록인지 확인한다. 다음은 블록 구조의 예시이다.



Segregated free list는 항상 오름차순으로 정리되며, 할당 과정은 이를 활용하여 새로 할당된 블록에 대해 항상 최적의 적합 블록을 찾는다. 메모리를 해제할 때는 가능한 경우 항상 메모리를 병합한다. malloc 및 realloc 함수는 미래의 external fragmentation를 방지하기 위해 항상 충분한 크기의 메모리를 할당한다.

3. 코드 설명

A. Global Variables and Macros

1) Constant

```
#define ALIGNMENT   8              /* double word (8) alignment */
#define NUM_LISTS   32             /* total number of segregated lists */
#define MALLOCBUF   1 << 12        /* minimum size of allocation */
#define REALLOCBUF  1 << 8         /* minimum additional size during reallocation */
#define HEADER_SIZE 4              /* sizeof(int): size of block header */
```

ALIGNMENT: 더블 워드(8바이트) 정렬.

NUM_LISTS: 총 세분화된 리스트의 수 (32개).

MALLOCBUF: 최소 할당 크기 (4096바이트).

REALLOCBUF: 재할당 시 최소 추가 크기 (256바이트).

HEADER_SIZE: 블록 header의 크기 (4바이트).

2) Alignment Functions

```
#define ALIGN(size)          (((size) + (ALIGNMENT-1)) & ~0x7)  /* align a size */
#define ALIGN_FOR_BLOCK(size) (((size) < ALIGNMENT) ? (ALIGNMENT << 1) : \
                              (((size) + HEADER_SIZE + ALIGNMENT - 1) & ~0x7)) /* align a size with header */
```

ALIGN(size): size를 가장 가까운 8바이트 경계로 정렬.

ALIGN_FOR_BLOCK(size): size를 header와 함께 가장 가까운 8바이트 경계로 정렬하며, 최소 8바이트 이상으로 정렬.

3) Block Header-Related Functions

```
#define BLOCK_SIZE(ptr)             (*((unsigned int*)(ptr)) & ~0x7)
#define PAYLOAD_SIZE(ptr)           (BLOCK_SIZE(ptr) - HEADER_SIZE)
#define PAYLOAD_ADDR(ptr)           ((ptr) + HEADER_SIZE)
#define HEADER_ADDR(ptr)            ((char*)(ptr) - HEADER_SIZE)
#define SET_HEADER(ptr, size)       (*((unsigned int*)(ptr)) = (size) | 1)
#define FREE_HEADER(ptr, size)      do { \
                                        *((unsigned int*)(ptr)) = (size); \
                                        *((unsigned int*)((ptr) + (size) - HEADER_SIZE)) = (size); \
                                    } while(0)
#define NEXT_BLOCK(ptr)             ((ptr) + (*(unsigned int*)(ptr) & ~0x7))
#define NEXT_LIST_HEADER(ptr)       ((ptr) + 2 * HEADER_SIZE)
#define IS_SET(ptr)                 ((*((unsigned int*)(ptr)) & 1) != 0)
```

BLOCK_SIZE(ptr): 블록의 크기(header + payload)를 가져옴.

PAYLOAD_SIZE(ptr): payload의 크기를 가져옴 (블록 크기 - header 크기).

PAYLOAD_ADDR(ptr): 블록 주소에서 payload 주소를 가져옴.

HEADER_ADDR(ptr): payload 주소에서 블록 주소를 가져옴.

SET_HEADER(ptr, size): 블록 header를 설정하고 할당된 상태로 표시.

FREE_HEADER(ptr, size): 블록 header와 푸터를 설정하여 자유 상태로 만듦.

NEXT_BLOCK(ptr): heap에서 다음 블록을 가져옴.

NEXT_LIST_HEADER(ptr): 다음 세분화된 리스트 header를 가져옴.

IS_SET(ptr): 블록이 할당된 상태인지 확인.

4) Free List related functions

```
#define NEXT_FREE_BLOCK_VAL(ptr)            (*(int*)((ptr) + HEADER_SIZE))
#define PREV_FREE_BLOCK_VAL(ptr)            (*(int*)((ptr) + 2 * HEADER_SIZE))
#define SET_NEXT_FREE_BLOCK(ptr, nextptr)   (*(int*)((ptr) + HEADER_SIZE) = (int)((nextptr) - (ptr)))
#define SET_PREV_FREE_BLOCK(ptr, prevptr)   (*(int*)((ptr) + 2 * HEADER_SIZE) = (int)((prevptr) - (ptr)))
#define NEXT_FREE_BLOCK(ptr)                ((ptr) + *(int*)((ptr) + HEADER_SIZE))
#define PREV_FREE_BLOCK(ptr)                ((ptr) + *(int*)((ptr) + 2 * HEADER_SIZE))
#define IS_END(ptr)                         (*(int*)((ptr) + HEADER_SIZE) == 0)
```

NEXT_FREE_BLOCK_VAL(ptr): 다음 free block의 값을 가져옴.

PREV_FREE_BLOCK_VAL(ptr): 이전 free block의 값을 가져옴.

SET_NEXT_FREE_BLOCK(ptr, nextptr): 다음 free block 포인터를 설정.

SET_PREV_FREE_BLOCK(ptr, prevptr): 이전 free block 포인터를 설정.

NEXT_FREE_BLOCK(ptr): 다음 free block의 포인터를 가져옴.

PREV_FREE_BLOCK(ptr): 이전 free block의 포인터를 가져옴.

IS_END(ptr): free list의 끝에 도달했는지 확인.

5) Heap space related functions

```
#define IS_WITHIN_HEAP(ptr)         ((ptr) <= mem_heap_hi())
#define GET_LAST_BLOCK_IF_FREE()    (prev_block(mem_heap_hi() + 1))
```

IS_WITHIN_HEAP(ptr): 포인터가 heap을 초과하지 않는지 확인.

GET_LAST_BLOCK_IF_FREE(): heap에서 마지막 블록을 가져오되, 블록이 자유 상태인 경우에만 가져옴.

B. Subroutines

1) mm_init

```
int mm_init(void) {
    // init_size - 모든 세분화된 리스트의 header를 포함할 수 있는 크기, 정렬된
상태로 계산
    int init_size = ((2 * HEADER_SIZE * NUM_LISTS + HEADER_SIZE + ALIGNMENT -
1) & ~0x7) - HEADER_SIZE;

    // 메모리를 할당하고 실패 시 -1을 반환
    head = (char*)mem_sbrk(init_size);
    if (head == (char*)-1) {
        return -1;
    }

    // 세분화된 리스트 초기화
    memset(head, 0, mem_heapsize());
    first = (char*)mem_heap_hi() + 1;

    // 초기 heap 할당 및 초기화
    void* initial_alloc = mm_malloc(MALLOCBUF);
    if (initial_alloc != NULL) {
        mm_free(initial_alloc);
    }
    else return -1;

    return 0;
}
```

mm_init 함수는 메모리 관리 시스템을 초기화하고, 메모리 할당 및 해제 프로세스의 기본 흐름을 설정하는 데 필수적이다.

mm_init함수는 init_size 변수를 사용하여 초기 메모리 블록의 크기를 계산한다. 이 크기는 모든 세분화된 리스트의 header를 저장할 수 있을 만큼 충분히 커야 하며, 메모리 정렬 요구 사항을 만족해야 한다. 계산 방법은 다음과 같다:

>> 2 * HEADER_SIZE * NUM_LISTS: 모든 리스트의 header에 필요한 크기를 두 배로 계산한다. 각 리스트에는 두 개의 header 공간이 필요하며, 하나는 시작을, 다른 하나는 끝을 표시한다.

>> HEADER_SIZE + ALIGNMENT - 1: 정렬을 보장하기 위해 추가적인 크기

를 더하고, 이 값을 다음 가장 큰 정렬 단위로 반올림한다.

>> & ~0x7: 8바이트 경계에 맞춰 크기를 정렬한다.

>> - HEADER_SIZE: 마지막에 header 크기를 빼주어 최종 크기를 조정한다.

그 후, mem_sbrk(init_size)를 호출하여 요청한 크기만큼의 메모리를 할당하고, head 포인터에 이 메모리의 시작 주소를 저장한다. 할당에 실패하면 -1을 반환하여 초기화 실패를 나타낸다. 할당된 메모리 영역을 모두 0으로 초기화(memset)하여, 세분화된 리스트를 사용할 준비를 한다. first 포인터는 할당된 메모리의 맨 끝을 가리키게 설정된다. (mem_heap_hi() + 1).

2) mm_malloc

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

    /* if less than MALLOCBUF, round to nearest 2's power */
    if (size < MALLOCBUF) {
       size = round_up_to_power_of_two(size);
    }

    char *list, *block_ptr;
    int newsize = ALIGN_FOR_BLOCK(size), bufsize;

    /* search segregated free list for available free blocks, starting at appropriate list, in ascending order */
    list = search_list(newsize);
    while (list <= first - ALIGNMENT) {
        block_ptr = NEXT_FREE_BLOCK(list);
        while (block_ptr) { // block_ptr이 NULL이 아닐 때까지, 또는 IS_END로 끝을 확인할 때까지
            if (BLOCK_SIZE(block_ptr) >= newsize) {  // 적합한 크기의 블록 찾기
                remove_block(block_ptr);            // 자유 리스트에서 해당 블록 제거
                split_block(block_ptr, newsize);        // 필요하다면 블록 분할
                return (void *)PAYLOAD_ADDR(block_ptr); // 할당된 블록의 사용자 데이터 주소 반환
            }
            if (IS_END(block_ptr)) {          // 블록이 리스트의 끝을 나타내면
                break; // 내부 루프 탈출
            }
            block_ptr = NEXT_FREE_BLOCK(block_ptr); // 다음 블록으로 이동
        }
        list = NEXT_LIST_HEADER(list); // 다음 크기 범주의 리스트로 이동
    }
    /* there is no suitable free block; assign a new size that is at least MALLOCBUF */
    bufsize = newsize > MALLOCBUF ? newsize : MALLOCBUF;
    if ((block_ptr = allocate_new_block(newsize)) == NULL)
        return NULL;
    return block_ptr;
}
```

mm_malloc 함수는 요청한 크기(size)의 메모리 블록을 할당하는 함수이

다. 이 함수는 메모리 할당 요청을 처리하고 적절한 크기의 메모리 블록을 사용자에게 반환한다.

먼저, 요청된 size가 MALLOCBUF(1<<12) 미만일 경우, 크기를 가장 가까운 2의 거듭제곱으로 반올림한다. 이는 메모리의 효율적 사용을 위해 fragmentation을 최소화하기 위해 round_up_to_power_of_two 함수를 호출하여 반올림을 수행합니다.

반올림된 크기를 정렬된 크기(newsize)로 조정한다(ALIGN_FOR_BLOCK). 이후, 세분화된 free 리스트에서 적합한 크기의 블록을 검색하기 시작한다(search_list).

그 다음 각 리스트를 순회하며 적절한 크기의 블록을 찾는다. 리스트 내의 각 블록(block_ptr)을 확인하고, 요구된 크기 이상의 블록을 찾을 때까지 리스트를 탐색한다. 적합한 블록을 찾으면, 그 블록을 free list에서 제거한다(remove_block). 필요한 경우에는, 블록을 더 작은 크기로 분할한다(split_block).

할당된 블록의 PAYLOAD_ADDR를 계산하여 반환한다. 이 주소는 실제 데이터를 저장할 수 있는 메모리 영역의 start point이다.

적절한 크기의 블록을 찾지 못한 경우, 요구된 크기 이상으로 새로운 메모리 블록을 할당한다(allocate_new_block). 그리고 나서 새로 할당된 블록의 주소를 반환한다. 새 블록 할당에 실패할 경우 NULL을 반환한다.

이처럼 first-fit 방법으로 malloc 함수를 구현했다. 블록을 찾을 때 더 작은 블록을 찾기 위해 리스트를 더 깊이 순회하지 않고, 첫 번째 적합한 블록을 찾으면 그것을 할당한다. 이에 따라 utilization이 낮아질 수 있지만 throughput을 위해 best fit이 아닌 first fit 방법으로 구현하였다.

3) mm_free

```
void mm_free(void *ptr) {
    if (!ptr) return;

    char *block_ptr = HEADER_ADDR(ptr);
    int current_size = BLOCK_SIZE(block_ptr);

    // 이전 블록과 다음 블록을 확인하여 병합 가능한지 판단
    char *prev_block_ptr = prev_block(block_ptr); // 이전 블록의 헤더 주소
    char *next_block_ptr = NEXT_BLOCK(block_ptr); // 현재 블록 다음의 블록 주소

    // 이전 블록이 free인 경우 병합
    if (prev_block_ptr && !IS_SET(prev_block_ptr)) {
        remove_block(prev_block_ptr);
        current_size += BLOCK_SIZE(prev_block_ptr);
        block_ptr = prev_block_ptr;
    }

    // 다음 블록이 free이고 heap 범위 내에 있는 경우 병합
    bool can_coalesce_next = IS_WITHIN_HEAP(next_block_ptr) && !IS_SET(next_block_ptr);
    if (can_coalesce_next) {
        remove_block(next_block_ptr);
        current_size += BLOCK_SIZE(next_block_ptr);
    }

    // 현재 블록을 자유 블록으로 설정하고 segregated list에 삽입
    coalesce_free_block(block_ptr, current_size);

    return;
}
```

mm_free 함수는 메모리를 해제할 때 단순히 메모리를 free 리스트에 반환하는 것뿐만 아니라, 주변 블록과의 병합을 통해 큰 free 블록을 만들어 내어 메모리 사용 효율을 높인다.

먼저, 주어진 포인터로부터 해당 메모리 블록의 header 주소를 계산한다. HEADER_ADDR 매크로를 사용하여 이를 수행한다.

이전 블록의 header 주소를 찾고, 해당 블록이 free인지 확인한다. Free block인 경우, 해당 블록을 free list에서 제거하고, 현재 블록과 이전 블록의 크기를 합쳐 하나의 큰 free block을 형성한다.

현재 블록 바로 다음에 위치한 블록을 확인하고, 이 블록이 free 상태이며 heap의 범위 내에 있는지 확인한다. 조건이 충족될 경우, 다음 블록도

free 리스트에서 제거하고 현재 블록과 병합한다.

병합된 새로운 크기로 현재 블록을 free block으로 재설정하고, 적절한 segregated list에 삽입한다. 이 과정은 coalesce_free_block 함수를 통해 수행된다.

4) mm_realloc

mm_realloc 함수는 할당된 메모리 블록의 크기를 변경하는 기능을 수행한다. 이 함수는 메모리 크기를 조정하거나 새로운 위치로 메모리를 이동할 필요가 있을 때 사용된다.

```
void *mm_realloc(void *ptr, size_t size) {
    if (ptr == NULL) return mm_malloc(size);
    if (!size) {
        mm_free(ptr);
        return NULL;
    }

    /* if less than REALLOCBUF, round to nearest 2's power */
    if (size < REALLOCBUF) {
        size = round_up_to_power_of_two(size);
    }

    char *blkptr = HEADER_ADDR(ptr);
    char *nextblkptr = NEXT_BLOCK(blkptr), *prevblkptr = prev_block(blkptr);
    int oldsize = BLOCK_SIZE(blkptr), newsize = ALIGN_FOR_BLOCK(size), combsize;

    // Check if the existing block size is sufficient.
    if (newsize <= PAYLOAD_SIZE(blkptr))
        return ptr;
```

ptr이 NULL이면, 이 함수는 단순히 mm_malloc(size)를 호출하여 새 메모리 블록을 할당한다. 요청된 크기가 0이면, mm_free(ptr)를 호출하여 현재 블록을 해제하고 NULL을 반환한다.

요청된 size가 REALLOCBUF(1<<8)보다 작은 경우, 크기를 가장 가까운 2의 거듭제곱으로 반올림한다. 이는 메모리 할당 효율성을 높이고 fragmentation을 최소화하는데 도움이 된다.

현재 블록의 payload 크기가 새로운 요구 크기보다 크거나 같은지 확인한다. 충분하면 기존의 포인터를 그대로 반환한다.

```
// Coalesce with adjacent free blocks if possible and needed.
if (prevblkptr && !IS_SET(prevblkptr) && BLOCK_SIZE(prevblkptr) > REALLOCBUF) {
    combsize = BLOCK_SIZE(prevblkptr) + oldsize;
    if (IS_WITHIN_HEAP(nextblkptr) && !IS_SET(nextblkptr)) {
        combsize += BLOCK_SIZE(nextblkptr);
        remove_block(nextblkptr);
    }
    if (combsize >= newsize) {
        remove_block(prevblkptr);
        memmove(prevblkptr, blkptr, oldsize);
        blkptr = prevblkptr;
        SET_HEADER(blkptr, combsize);
        return (void *)PAYLOAD_ADDR(blkptr);
    }
}
else if (IS_WITHIN_HEAP(nextblkptr) && !IS_SET(nextblkptr)) {
    combsize = BLOCK_SIZE(nextblkptr) + oldsize;
    if (combsize >= newsize) {
        remove_block(nextblkptr);
        SET_HEADER(blkptr, combsize);
        return (void *)PAYLOAD_ADDR(blkptr);
    }
}
```

현재 블록의 이전 및 다음 블록을 검사하여 free인지 확인하고, 가능한 경우 현재 블록과 병합하여 크기를 조정한다. 이전 블록과 다음 블록 모두를 병합할 수 있는 상황에서 충분한 공간이 확보된다면, 병합을 수행하고 주소를 조정한다.

```
    // If extending is needed, attempt to extend the heap.
    if (!IS_WITHIN_HEAP(nextblkptr)) {
        int bufsize = ALIGN_FOR_BLOCK(newsize - oldsize);
        if (mem_sbrk(bufsize) == (void *)-1)
            return NULL;
        SET_HEADER(blkptr, oldsize + bufsize);
        return (void *)PAYLOAD_ADDR(blkptr);
    }

    // Allocate a new block if no coalescing or extension is possible.
    char *newblkptr = (char*)mm_malloc(newsize);
    if (!newblkptr)
        return NULL;
    memcpy(newblkptr, ptr, PAYLOAD_SIZE(blkptr));
    mm_free(ptr);
    return newblkptr;
}
```

필요한 크기가 인접 블록 병합으로 충족되지 않고, 현재 블록이 heap의 끝에 위치한다면, mem_sbrk를 통해 heap을 확장하여 추가 공간을 확보한다.

위의 모든 시도가 실패하거나 적용할 수 없는 경우, 새로운 메모리 블록을 할당하고, 기존 블록의 데이터를 새 블록으로 복사한 후, 기존 블록을 해제한다.

위의 과정들을 모두 거치고 나면, 조정된 또는 새로 할당된 메모리 블록의 주소를 반환한다.

5) split_block

```
static void split_block(char *blkptr, int newsize) {
    char *newblkptr;
    int oldsize = BLOCK_SIZE(blkptr);
    if (oldsize <= newsize + ALIGNMENT) { /* there is enough space to split the block */
        SET_HEADER(blkptr, oldsize);
    }
    else {                              /* this block cannot be split */
        SET_HEADER(blkptr, newsize);
        newblkptr = NEXT_BLOCK(blkptr);
        FREE_HEADER(newblkptr, oldsize - newsize); /* set header for split block */
        insert_list(newblkptr);     /* insert split block into segregated list */
    }
}
```

split_block 함수는 특정 메모리 블록을 주어진 크기(newsize)에 맞게 조정하고, 필요하다면 남는 공간을 free block으로 분할하는 역할을 수행한다.

함수는 먼저 주어진 블록(blkptr)의 현재 크기(oldsize)를 확인한다. 함수는 newsize에 ALIGNMENT를 더한 값과 oldsize를 비교하여, 현재 블록이 주어진 newsize로 분할 가능한지 판단한다. 만약 oldsize가 newsize + ALIGNMENT보다 작거나 같다면, 충분한 공간이 없어 블록을 분할할 수 없다.

분할할 수 없는 경우, 현재 블록의 header는 oldsize로 설정되어 블록의 크기가 변경되지 않는다. 이는 블록이 요청된 크기로 조정될 수 없을 때 기존 크기를 유지한다는 것을 의미한다.

분할이 가능한 경우, 먼저 현재 블록의 header를 newsize로 설정한다. 이는 블록이 새로운 크기로 사용될 준비가 되었음을 나타낸다.

이후 newblkptr를 계산하여 현재 블록의 끝 부분에서 새로운 free block을 시작한다. 이 새 블록의 크기는 oldsize - newsize로, 현재 블록에서 새 크기를 제외한 나머지 부분이다. 새 free block의 header를 설정하고(FREE_HEADER), 이 블록을 segregated free list에 삽입합니다(insert_list).

6) prev_block

```
static char *prev_block(char *blkptr) {
    char *prevblk_footer = blkptr - HEADER_SIZE;
    char *prevblk_header = blkptr - BLOCK_SIZE(prevblk_footer);

    if (!is_prev_block_valid(prevblk_header, prevblk_footer)) {
        return NULL;
    }

    /* finally, check if the block is officially a free block by looking into segregated list */
    if (is_block_in_free_list(prevblk_header)) {
        return prevblk_header;
    }
    else return NULL;
}
```

prev_block 함수는 주어진 메모리 블록(blkptr)의 직전 블록을 찾고, 해당 블록이 free list에 포함되어 있는지 확인하는 역할을 수행한다. 이 함수의 목적은 메모리 할당 및 해제 작업 중 블록 병합을 효율적으로 수행하기 위해 필요한 이전 블록의 정보를 제공하는 것이다.

함수는 먼저 blkptr에서 HEADER_SIZE를 빼서 이전 블록의 footer 주소(prevblk_footer)를 계산한다. 이 footer 정보를 바탕으로, 이전 블록의 크기(BLOCK_SIZE(prevblk_footer))를 이용하여 header 주소(prevblk_header)를 찾는다.

이전 블록이 유효한지 확인하기 위해 is_prev_block_valid 함수를 호출한다. 이 함수는 header와 footer의 정보가 일치하는지, header가 heap의 범위 내에 있는지, 그리고 블록이 할당된 상태인지 등을 검사한다. 유효성 검사에서 문제가 발견되면 함수는 NULL을 반환하여 이전 블록이 사용할 수 없음을 나타낸다.

이전 블록의 header가 유효하다면, is_block_in_free_list 함수를 호출하여 해당 블록이 free list에 포함되어 있는지 확인한다. 이 함수는 세분화된 free list를 순회하면서 블록의 주소를 리스트 내의 주소와 비교한다.

만약 이전 블록이 free list에 있다면, 그 블록의 header 주소를 반환한다. 이는 메모리 해제 또는 재할당 과정에서 이 블록과 현재 블록을 병합할 수 있음을 의미한다.

free list에서 이전 블록을 찾지 못하면, 함수는 NULL을 반환하여 병합할 수 있는 이전 블록이 없음을 나타낸다.

7) search_list

```
static char *search_list(int size) {
    char *list = head;
    int threshold = 16;  // 초기 threshold 값 설정
    int list_num = 1;    // 리스트 번호 초기화

    // 주어진 크기가 threshold를 초과하지 않고, 리스트의 최대 수에 도달하지 않을 때까지 반복
    while (size > threshold && list_num < NUM_LISTS) {
        threshold <<= 1;  // threshold 값을 배로 증가
        list_num++;
        list = NEXT_LIST_HEADER(list);  // 다음 리스트 헤더로 이동
    }
    return list;
}
```

search_list 함수는 주어진 크기(size)에 따라 적절한 세분화된 free list를 찾는 역할을 수행한다. 이 함수는 메모리 할당 시스템에서 특정 크기의 메모리 블록을 요청할 때, 그 요청을 가장 잘 처리할 수 있는 리스트를 선택하는 과정을 담당한다.

함수는 list 포인터를 메모리 관리 시스템의 헤드(head)로 초기화하여 첫 번째 seg_list에서 시작한다. threshold는 16으로 설정되며, 이는 리스트를 구분하는 크기의 기준점이 된다. list_num은 리스트의 순서를 추적하며, 1에서 시작한다.

주어진 size가 현재 threshold보다 크고 아직 모든 리스트를 확인하지 않았다면, 리스트를 순차적으로 탐색한다. threshold는 매 iteration마다 두 배로 증가하며, 이는 각 리스트가 처리할 수 있는 메모리 블록의 최대 크기를 의미한다. 예를 들어, 첫 번째 리스트는 최대 16 바이트 크기의 블록을 처리하고, 다음 리스트는 32 바이트를 처리하는 식이다. list_num은 리스트의 순서를 나타내며, NUM_LISTS에 도달할 때까지 증가한다.

각 반복에서 list 포인터는 NEXT_LIST_HEADER(list)를 사용하여 다음 리

스트의 header로 이동한다. 이 매크로는 현재 리스트의 header에서 다음 리스트의 header로 포인터를 옮기는 데 사용된다.

주어진 size에 적합한 리스트를 찾으면, 그 리스트의 포인터(list)를 반환한다. 만약 모든 리스트를 검토해도 적합한 리스트를 찾지 못하면, 가장 큰 크기를 다루는 마지막 리스트의 포인터를 반환한다.

8) insert_list

```
static void insert_list(char *blkptr)
{
    /* first, find the list that best-fits block's size */
    char *list = search_list(BLOCK_SIZE(blkptr));

    if (IS_END(list)) {
        SET_NEXT_FREE_BLOCK(blkptr, blkptr);
        link_blocks(list, blkptr);
        return;
    }
    else {
        while (1) {
            /* insert when the block not greater than next block, or */
            /* there is no more block in the list                    */
            if (BLOCK_SIZE(NEXT_FREE_BLOCK(list)) >= BLOCK_SIZE(blkptr)) {
                /* found the appropriate position */
                link_blocks(blkptr, NEXT_FREE_BLOCK(list));
                link_blocks(list, blkptr);
                return;
            }
            else if (IS_END(NEXT_FREE_BLOCK(list))) {
                /* there is no more block to compare to */
                SET_NEXT_FREE_BLOCK(blkptr, blkptr);
                link_blocks(NEXT_FREE_BLOCK(list), blkptr);
                return;
            }
            /* none of above condition holds; go to next item in the list */
            list = NEXT_FREE_BLOCK(list);
        }
    }
}
```

insert_list 함수는 주어진 메모리 블록(blkptr)을 적절한 크기의 free list에 삽입하는 역할을 수행한다. 이 함수는 메모리 해제 과정에서 반환된 블록

을 free list에 효율적으로 추가하여 재사용을 용이하게 하며, 메모리 fragmentation를 최소화한다.

함수는 먼저 search_list 함수를 호출하여 주어진 블록의 크기(BLOCK_SIZE(blkptr))에 가장 적합한 free list를 찾는다. 이 리스트는 블록 크기에 기반하여 세분화되어 있다.

만약 찾은 리스트가 리스트의 끝(IS_END(list))을 나타내면, 현재 블록을 리스트의 첫 번째 블록으로 설정하고 자기 자신을 가리키도록 한다. 그 다음, link_blocks 함수를 사용하여 블록을 리스트에 연결한다. 리스트의 끝이 아닌 경우, 적절한 삽입 위치를 찾기 위해 리스트를 순회한다.

각 블록을 검사하여 현재 블록의 크기가 다음 블록(NEXT_FREE_BLOCK(list))의 크기보다 작거나 같은 위치를 찾는다. 적절한 위치를 찾으면, link_blocks 함수를 사용하여 현재 블록을 리스트에 연결하고, 삽입 과정을 종료한다.

리스트를 순회하는 동안 적절한 삽입 위치를 찾지 못하고 리스트의 끝에 도달하면(IS_END(NEXT_FREE_BLOCK(list))), 현재 블록을 리스트의 마지막으로 추가한다. 이 때, SET_NEXT_FREE_BLOCK을 사용하여 현재 블록이 자기 자신을 가리키도록 설정하고, link_blocks를 사용하여 리스트에 연결한다.

9) remove_block

```
static void remove_block(char *block_ptr)
{
    char *prevblkptr = PREV_FREE_BLOCK(block_ptr);
    char *nextblkptr = NEXT_FREE_BLOCK(block_ptr);
    if (IS_END(block_ptr)) /* if at the end of free list, just fix the previous block */
        SET_NEXT_FREE_BLOCK(prevblkptr, prevblkptr);
    else { /* if in the middle of free list, connect previous and next free blocks */
        link_blocks(prevblkptr, nextblkptr);
    }
    return;
}
```

remove_block 함수는 free list 내의 특정 메모리 블록을 제거하는 역할을 수행한다. 이 함수는 주로 메모리 블록이 할당되거나 블록 병합 과정에서 필요할 때 호출된다.

remove_block 함수는 먼저 제거하려는 블록(block_ptr)의 이전(prevblkptr) 및 다음(nextblkptr) 블록 포인터를 구한다. 이 포인터들은 블록의 header 정보에서 얻어진다.

함수는 제거하려는 블록이 free list의 끝에 위치하는지 확인한다(IS_END(block_ptr)). 만약 블록이 리스트의 끝이라면, 이전 블록의 다음 블록 포인터를 자기 자신을 가리키도록 설정하여 리스트의 끝을 나타낸다. 이는 리스트에서 해당 블록을 제거하는 과정이다.

제거하려는 블록이 리스트의 중간에 위치하는 경우, link_blocks 함수를 호출하여 이전 블록의 다음 블록 포인터를 제거된 블록의 다음 블록에 연결한다. 동시에, 다음 블록의 이전 블록 포인터를 제거된 블록의 이전 블록에 연결한다.

4. 구현 결과

위와 같이 segregated free list 기법을 사용하여 malloc, realloc, free 함수를 구현하였다. 구현 결과는 아래와 같다.

```
cse20191243@cspro:~/SP/4/prj4-malloc$ ./mdriver -v
[20191243]::NAME: Taekyu Kim, Email Address: taekyu.s.kim@gmail.com
Using default tracefiles in ./tracefiles/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   97%    5694  0.000434 13111
 1       yes   98%    5848  0.003616  1617
 2       yes   98%    6648  0.004552  1460
 3       yes   99%    5380  0.001154  4662
 4       yes   49%   14400  0.000568 25361
 5       yes   94%    4800  0.008577   560
 6       yes   92%    4800  0.004861   988
 7       yes   97%   12000  0.019469   616
 8       yes   90%   24000  0.064831   370
 9       yes  100%   14401  0.004038  3566
10       yes   77%   14401  0.004289  3358
Total          90%  112372  0.116388   965

Perf index = 54 (util) + 40 (thru) = 94/100
```

이번 프로젝트에서 best-fit이 아닌 first-fit 방법으로 malloc을 구현하였는데 이로 인해 utilization이 조금 낮게 나온 것 같다. Best fit 방법으로 구현하였다면 utilization 수치가 조금 더 높게 나왔겠지만 그렇게 되면 크기가 딱 맞는 block을 찾기 위해 list를 순회하는 시간이 길어져 throughput과 trade-off가 발생할 수밖에 없다. Segregated list 기법은 이미 크기별로 free block들을 분류하므로 어느 정도 best-fit의 특성을 지니므로 first-fit으로 구현하였다.