CSED211: MallocLab Report

20240832 / 채승현

0. 개요

이번 과제에서는 dynamic memory allocation의 성능 지표인 throughput과 utilization을 모두 확보하고자 segregated free list 방식을 사용해 구현했다. 이는 단일 list로 전체 heap을 탐색하는 방식보다 원하는 size의 block을 찾는 속도가 빠르며, 훨씬 효율적으로 best-fit을 찾을 수 있기 때문이다.

1. 구현

0) macros & global variables

우선 포인터 연산의 복잡함과 casting 실수를 줄이기 위해 textbook에 소개된 macro들을 활용했다. 각 macro에 대한 설명은 주석으로 적어 놓았다. 또한 writeup에 적혀 있는 규칙상 global array 선언이 금지되어 있다. 이를 해결하기 위해 void** seg_list라는 포인터 변수 하나만 선언하였고, 실제 list 배열은 mm_init 함수에서 heap 영역에 할당하여 이 포인터가 가리키게 하였다. 이를 통해 seg_list[i] 처럼 직관적으로 segregated free list에 접근할 수 있으면서 규칙을 어기지 않을 수 있었다.

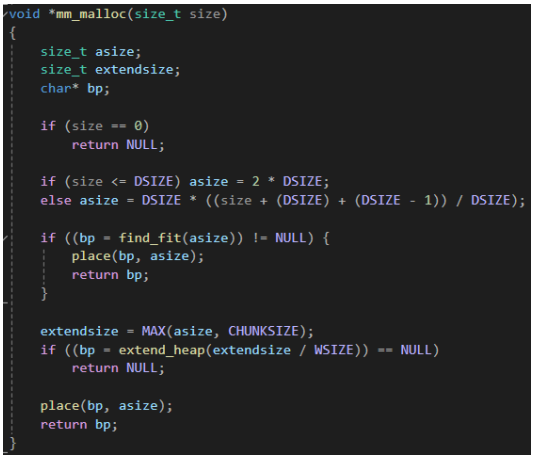
1) mm_init 함수

mm_init은 heap과 segregated list를 초기화하는 함수이다. mem_sbrk를 호출해 LIST_LIMIT (20개) 크기만큼 heap의 가장 앞부분에 포인터 배열 공간을 할당하고 seg_list가 이를 가리키게 했다. 이 공간은 각 size class 별 가용 list의 header pointer 역할을 한다. 이후 정렬을 위한 padding, prologue header/footer, epilogue header를 생성해 heap의 기본 구조를 만들었다. 마지막으로 extend_heap을 호출해 초기 가용 block을 생성했다.

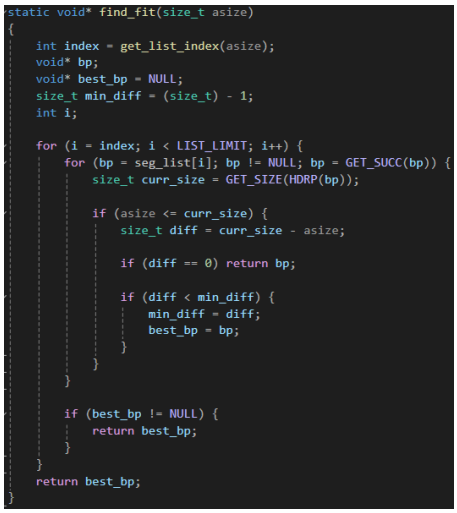
```
int mm_init(void)
{
    int i;
    char* prologue_ptr;

    if ((seg_list = mem_sbrk((LIST_LIMIT * sizeof(void*)) + (4 * WSIZE))) == (void*)-1)
        return -1;

    for (i = 0; i < LIST_LIMIT; i++)
        seg_list[i] = NULL;

    prologue_ptr = (char*)seg_list + (LIST_LIMIT * sizeof(void*));

    PUT(prologue_ptr, 0);
    PUT(prologue_ptr + (1 * WSIZE), PACK(DSIZE, 1));
    PUT(prologue_ptr + (2 * WSIZE), PACK(DSIZE, 1));
    PUT(prologue_ptr + (3 * WSIZE), PACK(0, 1));

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

    return 0;
}
```

2) mm_malloc, find_fit, place 함수

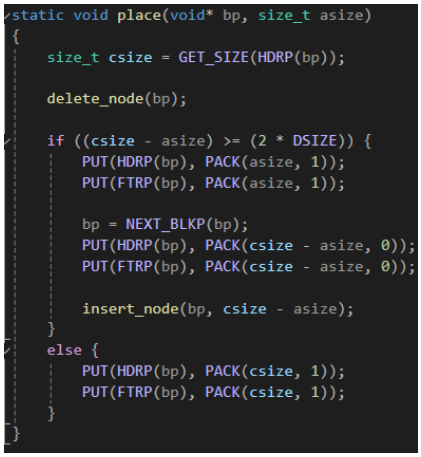
mm_malloc은 메모리를 할당하는 함수다. 요청된 size를 8 bytes ALIGN에 맞춰 조정하고 find_fit 함수를 호출한다. find_fit은 segregated free list를 탐색하는 함수다. get_list_index를 통해 size에 맞는 적절한 index를 찾은 후, 해당 index번째 list부터 상위 list로 이동하며 block을 찾게 된다. 여기서 best-fit 방식을 적용해 list 내에서 요청 크기를 만족하는 block 중 가장 크기 차이가 작은 block을 선택하도록 구현했다. 이는 first-fit에 비해 탐색 시간은 조금 더 걸리겠지만, utilization을 높일 수 있다. 만약 적절한 block을 찾았다면 place 함수가 호출된다. place 함수에서는 block을 allocated 상태로 변경하고 남는 공간이 최소 block 크기 (16 bytes) 이상이면 split하여 남은 부분을 다시 가용 list에 반환하도록 한다. 이는 내부 fragmentation을 줄이기 위함이다.



```
void *mm_malloc(size_t size)
{
    size_t asize;
    size_t extendsize;
    char* bp;

    if (size == 0)
        return NULL;

    if (size <= DSIZE) asize = 2 * DSIZE;
    else asize = DSIZE * ((size + (DSIZE) + (DSIZE - 1)) / DSIZE);

    if ((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        return bp;
    }

    extendsize = MAX(asize, CHUNKSIZE);
    if ((bp = extend_heap(extendsize / WSIZE)) == NULL)
        return NULL;

    place(bp, asize);
    return bp;
}
```



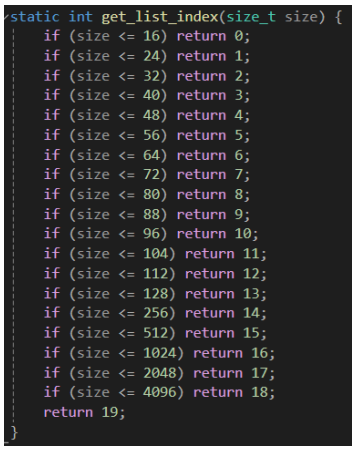
```
static void* find_fit(size_t asize)
{
    int index = get_list_index(asize);
    void* bp;
    void* best_bp = NULL;
    size_t min_diff = (size_t) - 1;
    int i;

    for (i = index; i < LIST_LIMIT; i++) {
        for (bp = seg_list[i]; bp != NULL; bp = GET_SUCC(bp)) {
            size_t curr_size = GET_SIZE(HDRP(bp));

            if (asize <= curr_size) {
                size_t diff = curr_size - asize;

                if (diff == 0) return bp;

                if (diff < min_diff) {
                    min_diff = diff;
                    best_bp = bp;
                }
            }
        }

        if (best_bp != NULL) {
            return best_bp;
        }
    }
    return best_bp;
}
```



```
static void place(void* bp, size_t asize)
{
    size_t csize = GET_SIZE(HDRP(bp));

    delete_node(bp);

    if ((csize - asize) >= (2 * DSIZE)) {
        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_node(bp, csize - asize);
    }
    else {
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }
}
```

3) get_list_index 함수

이 함수에서 일반적인 2^n으로 list를 나누면 작은 크기의 요청이 빈번한 trace에서 효율이 떨어지게 된다. 따라서 16 bytes ~ 128 bytes까지의 작은 구간은 8 bytes 단위로 세밀하게 index를 나누도록 if-else를 이용해 구현했다. 이를 통해 작은 block들이 섞이지 않고 관리되도록 해 내부 fragmentation을 줄일 수 있었다.



```
static int get_list_index(size_t size) {
    if (size <= 16) return 0;
    if (size <= 24) return 1;
    if (size <= 32) return 2;
    if (size <= 40) return 3;
    if (size <= 48) return 4;
    if (size <= 56) return 5;
    if (size <= 64) return 6;
    if (size <= 72) return 7;
    if (size <= 80) return 8;
    if (size <= 88) return 9;
    if (size <= 96) return 10;
    if (size <= 104) return 11;
    if (size <= 112) return 12;
    if (size <= 128) return 13;
    if (size <= 256) return 14;
    if (size <= 512) return 15;
    if (size <= 1024) return 16;
    if (size <= 2048) return 17;
    if (size <= 4096) return 18;
    return 19;
}
```

4) mm_free & coalesce 함수

mm_free는 block을 반환하는 함수다. block의 header와 footer의 allocated bit를 0으로 만들고 coalesce 함수를 호출한다. coalesce는 boundary tag를 이용해 인접한 block이 가용 상태인지 확인하고 병합하는 함수다. 4가지 경우(전후 모두 할당, 앞만 가용, 뒤만 가용, 전후 모두 가용)를 처리하고 병합된 block은 insert_node 함수를 통해 해당 size list의 맨 앞(LIFO)에 삽입한다. LIFO policy는 방금 해제된 block을 다시 사용할 확률이 높다는

점을 이용해 cache 효율을 높인다.

```c
static void* coalesce(void* bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));

    // case 1
    if (prev_alloc && next_alloc) {
        insert_node(bp, size);
        return bp;
    }

    // case 2
    else if (prev_alloc && !next_alloc) {
        delete_node(NEXT_BLKP(bp));
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    }

    // case 3
    else if (!prev_alloc && next_alloc) {
        delete_node(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);
    }

    // case 4
    else {
        delete_node(PREV_BLKP(bp));
        delete_node(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_node(bp, size);
    return bp;
}
```

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

    size = GET_SIZE(HDRP(ptr));

    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));

    coalesce(ptr);
}
```

```c
static void insert_node(void* bp, size_t size)
{
    int index = get_list_index(size);
    void* root = seg_list[index];

    SET_SUCC(bp, root);
    SET_PRED(bp, NULL);

    if (root != NULL)
        SET_PRED(root, bp);

    seg_list[index] = bp;
}
```

5) mm_realloc 함수

이번 과제에서 성능을 결정짓는 아주 중요한 함수로, 메모리를 재할당하는 함수다. 기존의 malloc-memcpy-free 방식은 heap을 계속 확장시키며 외부 fragmentation을 유발할 수 있다. 따라서 이를 해결하기 위해 다음 3가지 방법을 사용했다.

① 요청 size가 기존보다 줄어드는 경우 block을 자르지 않고 그대로 반환한다. 여기서 남는 공간은 당장 쓰이지 않지만 크기가 다시 커질 때 재할당 없이 즉시 사용할 수 있도록 해 heap 확장을 막는다.

② 다음 block이 가용 상태고 합쳤을 때 공간이 충분하다면 즉시 병합한다. 이 경우에도 합친 크기가 요청 크기보다 훨씬 커도 자르지 않고 통째로 사용하게 된다. 이것은 memcpy 비용을 제고하며 미래의 확장을 위한 buffer를 확보하는 효과가 있다.

③ block이 heap 맨 끝(epilogue 앞)에 있다면 mem_sbrk로 부족한 만큼만 heap을 늘려서 사용한다.

```c
void *mm_realloc(void *ptr, size_t size)
{
    void* newptr;
    size_t old_size;
    size_t new_size;
    size_t extend_needed;
    size_t combine_size;

    void* next_bp;
    size_t next_alloc;
    size_t next_size;

    if (ptr == NULL) return mm_malloc(size);
    if (size == 0) {
        mm_free(ptr);
        return NULL;
    }

    old_size = GET_SIZE(HDRP(ptr));

    if (size <= DSIZE) new_size = 2 * DSIZE;
    else new_size = DSIZE * ((size + (DSIZE)+(DSIZE - 1)) / DSIZE);

    // policy 1: if new_size is smaller than or equal to old size, return ptr
    // this avoids unecessary copy overhead
    if (new_size <= old_size) {
        return ptr;
    }

    next_bp = NEXT_BLKP(ptr);
    next_alloc = GET_ALLOC(HDRP(next_bp));
    next_size = GET_SIZE(HDRP(next_bp));
```

```c
    // policy 2: if next block is free and combined_size is enough, coalesce and use it
    combine_size = old_size + next_size;
    if (!next_alloc && (combine_size >= new_size)) {
        delete_node(next_bp);
        PUT(HDRP(ptr), PACK(combine_size, 1));
        PUT(FTRP(ptr), PACK(combine_size, 1));
        return ptr;
    }

    // policy 3: if the block is at the end of heap, extend heap directly
    if (next_size == 0) {
        extend_needed = new_size - old_size;
        if ((long)(mem_sbrk(extend_needed)) == -1)
            return NULL;

        PUT(HDRP(ptr), PACK(new_size, 1));
        PUT(FTRP(ptr), PACK(new_size, 1));
        PUT(HDRP(NEXT_BLKP(ptr)), PACK(0, 1)); // restore epilogue header
        return ptr;
    }

    // fallback
    newptr = mm_malloc(size);
    if (newptr == NULL) return NULL;

    memcpy(newptr, ptr, old_size - DSIZE);
    mm_free(ptr);
    return newptr;
}
```

6) mm_check 함수

mm_check, checkblock, printblock 함수를 구현하여 heap의 일관성을 검사해보았다. heap의 시작부터 끝까지 순회하며 block의 정렬 상태, header와 footer의 일치 여부, 가용 block의 coalescing 여부를 검사하도록 했다. 또한 segregated list를 순회하며 pointer가 유효한 heap 범위를 가리키는지, prev와 next가 올바르게 연결되어 있는지, list 상의 가용 block 개수와 heap 전체 탐색 시의 개수가 일치하는 지 까지도 확인하도록 구현했다. 테스트를 할 때 verbose를 0으로 하여 mm_malloc, mm_free의 끝부분에 mm_check를 해보았고, 따로 error가 나지 않은 것을 확인했다. 최종 점수를 낼 때는 mm_check 함수를 제거하고 진행했다.

```c
static void checkblock(void* bp)
{
    if ((size_t)bp % 8) {
        printf("Error: %p is not doubleword aligned\n", bp);
        exit(1);
    }
    if (GET(HDRP(bp)) != GET(FTRP(bp))) {
        printf("Error: Header does not match footer at %p\n", bp);
        exit(1);
    }
}

static void printblock(void* bp)
{
    size_t hsize, halloc, fsize, falloc;

    hsize = GET_SIZE(HDRP(bp));
    halloc = GET_ALLOC(HDRP(bp));
    fsize = GET_SIZE(FTRP(bp));
    falloc = GET_ALLOC(FTRP(bp));

    if (hsize == 0) {
        printf("%p: EOL\n", bp);
        return;
    }
    printf("%p: header: [%d:%c] footer: [%d:%c]\n", bp,
        (int)hsize, (halloc ? 'a' : 'f'),
        (int)fsize, (falloc ? 'a' : 'f'));
}
```

```c
int mm_check(void)
{
    char* heap_start = (char*)seg_list + (LIST_LIMIT * sizeof(void*)) + (2 * WSIZE);

    char* bp = heap_start;
    int i;
    int free_count_by_heap = 0;
    int free_count_by_list = 0;
    int verbose = 0;

    if (verbose) printf("Heap Start (%p):\n", heap_start);

    if ((GET_SIZE(HDRP(heap_start)) != DSIZE) || !GET_ALLOC(HDRP(heap_start))) {
        printf("Error: Bad prologue header\n");
        exit(1);
    }

    for (bp = heap_start; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
        if (verbose) printblock(bp);
        checkblock(bp);

        if (!GET_ALLOC(HDRP(bp)) && !GET_ALLOC(HDRP(NEXT_BLKP(bp)))) {
            printf("Error: Contiguous free blocks not coalesced at %p\n", bp);
            exit(1);
        }

        if (!GET_ALLOC(HDRP(bp))) {
            free_count_by_heap++;
        }
    }

    if ((GET_SIZE(HDRP(bp)) != 0) || !(GET_ALLOC(HDRP(bp)))) {
        printf("Error: Bad epilogue header\n");
        exit(1);
    }

    for (i = 0; i < LIST_LIMIT; i++) {
        void* curr = seg_list[i];
        void* prev = NULL;

        while (curr != NULL) {
            if ((char*)curr < (char*)mem_heap_lo() || (char*)curr >(char*)mem_heap_hi()) {
                printf("Error: Free list pointer %p out of bounds in list %d\n", curr, i);
                exit(1);
            }
            if (GET_ALLOC(HDRP(curr))) {
                printf("Error: Allocated block %p in free list %d\n", curr, i);
                exit(1);
            }
            if (prev != NULL && GET_PRED(curr) != prev) {
                printf("Error: Prev pointer inconsistency at %p\n", curr);
                exit(1);
            }
            free_count_by_list++;
            prev = curr;
            curr = GET_SUCC(curr);
        }
    }

    if (free_count_by_heap != free_count_by_list) {
        printf("Error: Free block count mismatch. Heap: %d, List: %d\n",
            free_count_by_heap, free_count_by_list);
        exit(1);
    }

    return 1;
}
```

2. 결과

이렇게 구현된 mm.c를 writeup을 참고하여 ./mdriver -v 를 통해 테스트했다. 그 결과 perf index에서 총 90/100 (util: 50/60, thru 40/40) 만큼의 성능을 달성한 것을 확인했다. mm_realloc과 get_index_list는 writeup에 적혀 있는 ./mdriver 기능들을 활용해 테스트해 보며, 실험적으로 구현했다.

```
[cotmdgus@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  0.001503  3789
 1       yes   99%    4805  0.000172 27952
 2       yes   55%   12000  0.000319 37641
 3       yes   55%    8000  0.000211 37879
 4       yes   51%   24000  0.001515 15838
 5       yes   51%   16000  0.000407 39283
 6       yes   99%    5848  0.000205 28485
 7       yes   99%    5032  0.000178 28238
 8       yes   66%   14400  0.000329 43729
 9       yes   66%   14400  0.001629  8839
10       yes   99%    6648  0.000243 27358
11       yes   99%    5683  0.000209 27204
12       yes  100%    5380  0.001507  3569
13       yes  100%    4537  0.000179 25375
14       yes   96%    4800  0.000735  6530
15       yes   96%    4800  0.002035  2359
16       yes   95%    4800  0.002031  2363
17       yes   95%    4800  0.000733  6549
18       yes  100%   14401  0.000206 69772
19       yes  100%   14401  0.000207 69738
20       yes   67%   14401  0.000197 73288
21       yes   67%   14401  0.000198 72659
22       yes   66%      12  0.000000 24000
23       yes   66%      12  0.000000 24000
24       yes   89%      12  0.000000 24000
25       yes   89%      12  0.000000 24000
Total          83%  209279  0.014951 13998

Perf index = 50 (util) + 40 (thru) = 90/100
```