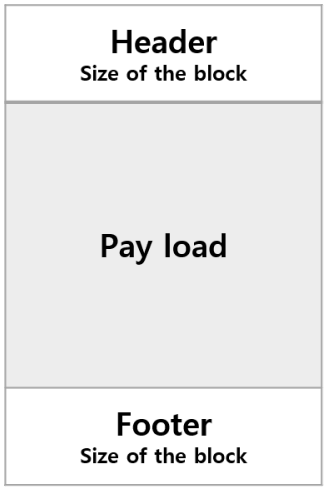
Lab 7

20200437 김채현

본 Lab의 목표는 Dynamic malloc allocator를 구현하여 heap에 공간을 할당해주도록 하는 것이다. mm_init, mm_malloc, mm_free, mm_realloc 이 4개의 함수를 수정하여 높은 performance와 utilization이 나오도록 하면 된다. 구현 과정은 크게 Allocator 부분과 alloc과 free 과정을 진행하고 있는지 확인하는 heap consistency checker 부분으로 나눌 수 있다.

Allocator

구현을 위해서 segregated list를 사용하였다. 16개의 free list가 존재하고, k번째 list는 size가 2^k 이상 2^{k+1}미만인 block을 저장할 수 있다. list는 doubly linked list로 구현되어 있고, block은 내림차순으로 저장되어 있다.



<Allocated Block>

<Free Block>

이 Allocator은 mm_init, mm_malloc, mm_free, mm_realloc을 통해 구현된다.

---

mm_init

```
int mm_init(void) {
    // Store free list pointer on heap
    if ((long)(free_lists = mem_sbrk(((MAX_POWER + 1) & ~1) *sizeof(char
*))) != -1) {

        int i = 0;
        for (i = 0; i <= MAX_POWER; i++) {
```

```
            SET_FREE_LIST_PTR(i, NULL);
        }

        // align to double word
        mem_sbrk(WSIZE);

        if ((long)(heap_ptr = mem_sbrk(4*WSIZE)) != -1) {
            PUT_WORD(heap_ptr, PACK(0, TAKEN)); // Prolog header
            PUT_WORD(FTRP(heap_ptr), PACK(0, TAKEN)); // Prolog footer

            char ** epilog = NEXT_BLOCK_IN_HEAP(heap_ptr);
            PUT_WORD(epilog, PACK(0, TAKEN)); // Epilog header

            heap_ptr += HDR_FTR_SIZE; // Move past prolog

            char **new_block;

            if ((new_block = extend_heap(CHUNK)) != NULL){
                insert_list (new_block);
                return 0;
            }
            else return -1;
        } // 2 for prolog, 2 for epilog
        else  return -1;
    }
    else return -1;
}
```

mm_init 함수에서는 malloc package를 초기화한다. mem_sbrk함수를 이용하여 heap에 영역을 확보하고, free list를 heap의 시작부분에 놓고, 각 i번째 free list이 pointer를 NULL로 초기화해준다. mem_sbrk함수를 이용하여 heap을 doubly word size로 정렬한다. 이후 prologue header, footer 그리고 epilogue header를 할당해준다.

---

mm_malloc

```
void *mm_malloc(size_t size) {
    if (size <= 1<<12) {
        size = round_up(size);
    }

    size_t words = ALIGN(size) / WSIZE;
```

```
    size_t size_extension;
    char **p;

    if (size == 0) {
        return NULL;
    }

    // check block is large enough
    // extend the heap
    if ((p = find_free_block(words)) == NULL) {
        if(words <= CHUNK) size_extension = CHUNK;
        else size_extension = words;

        // do not remove block
        if ((p = extend_heap(size_extension)) != NULL) {
            alloc_free_block(p, words);
            return p + 1;
        }
        else return NULL;
    }

    delete_list (p);
    alloc_free_block(p, words);

    return p + 1;
}
```

mm_malloc 함수는 parameter로 allocation 해야하는 size를 받아서 얼마나 할당을 해야하는지 판단한다. size를 round up하여 어디에 allocation 해야하는지 결정하여야 한다. size에 맞는 free block이 존재하는지 체크하고, block을 할당하고 delete_list 함수를 호출하여 free list에서 삭제한다. 이후 header size만큼 더한 값을 반환한다. 반대로 size에 맞는 free block이 없다면 size를 얼마나 extension 해야하는지 정한다. extend_heap 함수를 호출하여 heap을 extension 해주고 성공하면 제대로 할당해준 후 위와 같이 header size 만큼 더한 값을 반환하면 되고, 실패하면 NULL를 반환해주면 된다. 알다시피 이 경우에서는 free list의 영역 외에 영역을 확장하고 그곳에 할당한 것이므로 delete_list 함수를 호출하여 free list에서 삭제하거나 그럴 필요가 없다.

---

mm_free

```
void mm_free(void *ptr) {
    ptr -= WSIZE;

    size_t size = GET_SIZE(ptr);
```

```
    PUT_WORD(ptr, PACK(size, FREE));
    PUT_WORD(FTRP(ptr), PACK(size, FREE));

    ptr = coalesce(ptr);
    insert_list (ptr);
}
```

mm_free 함수는 굉장히 짧다. 먼저 함수의 parameter로서 block의 pointer를 저장한 ptr을 받는다. block의 상태를 free로 변환하고, coalesce 함수를 호출한다. 이후 insert_list 함수를 호출하여 free list에 block을 추가해준다.

---

mm_realloc

```
void *mm_realloc(void *ptr, size_t size) {
    static int pre_size;
    int buffer_size;
    int dif = abs(size - pre_size);

    if (!(dif < 1<<12 && dif % round_up(dif))) {
        buffer_size = size % 1000 >= 500 ? size + 1000 - size % 1000 : size
- size % 1000; //rounding to thousand
    } else {
        buffer_size = round_up(dif);
    }

    if (ptr == NULL) {
        return mm_malloc(ptr);
    }

    // start of block
    char **old = (char **)ptr - 1;
    char **p = (char **)ptr - 1;

    // get intended and current size
    size_t old_size = GET_SIZE(p); // words
    size_t new_size = ALIGN(size) / WSIZE; // words
    size_t total_buffer = new_size + buffer_size;

    if (total_buffer == old_size && new_size <= total_buffer) {
        return p + 1;
    }

    if (new_size == 0) {
```

```
        mm_free(ptr);
        return NULL;
    }

    if (new_size > old_size) {
        // checks if possible to merge with both prev and next block in
memory
        if (GET_SIZE(PREV_BLOCK_IN_HEAP(p)) +
GET_SIZE(NEXT_BLOCK_IN_HEAP(p)) + old_size + 4 >= total_buffer &&
GET_STATUS(PREV_BLOCK_IN_HEAP(p)) == FREE &&
GET_STATUS(NEXT_BLOCK_IN_HEAP(p)) == FREE) {
            PUT_WORD(p, PACK(old_size, FREE));
            PUT_WORD(FTRP(p), PACK(old_size, FREE));

            p = coalesce(p);
            memmove(p + 1, old + 1, old_size * WSIZE);
            alloc_free_block(p, total_buffer);
        }
        // checks if possible to merge with next block in memory
        else if (GET_SIZE(PREV_BLOCK_IN_HEAP(p)) + old_size + 2 >=
total_buffer && GET_STATUS(PREV_BLOCK_IN_HEAP(p)) == FREE &&
GET_STATUS(NEXT_BLOCK_IN_HEAP(p)) == TAKEN) {
            PUT_WORD(p, PACK(old_size, FREE));
            PUT_WORD(FTRP(p), PACK(old_size, FREE));

            p = coalesce(p);

            memmove(p + 1, old + 1, old_size * WSIZE);
            alloc_free_block(p, total_buffer);
        }
        // checks if possible to merge with previous block in memory
        else if (GET_SIZE(NEXT_BLOCK_IN_HEAP(p)) + old_size + 2 >=
total_buffer && GET_STATUS(PREV_BLOCK_IN_HEAP(p)) == TAKEN &&
GET_STATUS(NEXT_BLOCK_IN_HEAP(p)) == FREE) {
            PUT_WORD(p, PACK(old_size, FREE));
            PUT_WORD(FTRP(p), PACK(old_size, FREE));

            p = coalesce(p);
            alloc_free_block(p, total_buffer);
        }
        else { // end case: if no optimization possible, just do brute force
realloc
            p = (char **)mm_malloc(total_buffer*WSIZE + WSIZE) - 1;

            if (p == NULL) {
                return NULL;
            }
```

```
            memcpy(p + 1, old + 1, old_size * WSIZE);
            mm_free(old + 1);
        }
    }

    pre_size = size;
    return p + 1;
}
```

mm_realloc 함수에서 이전의 size와 현재 alloc 하려는 size의 차를 계산하여 넣을 수 있으면 round up 함수를 호출하여 round up to the next highest power of 2 방법을 사용한다. 반대로 넣을 수 없다면 rounding to thousand 방식을 이용하여 buffer의 크기를 결정한다. round up 함수는 Round up to the next highest power of 2를 하는 함수이고, rounding to thousand 함수는 따로 구현하지 않았다. pointer ptr이 NULL이면 그냥 malloc을 해주면 된다. size가 0이면 그냥 free 해주면 된다. new_size인지 old_size인지에 따라서 이웃한 block을 사용하거나 coalesce를 사용해야 하는지 체크한다. 위의 방법이 안된다면 alloc과 free를 한다.

Heap consistency checker

Heap consistency checker에서는 mm_check와 같은 함수를 구현하고, 그 외의 check 함수를 정의하여 이용한다.

mm_check는 check_free_blocks_marked_free(), check_contiguous_free_block_coalesced(), check_all_free_blocks_in_free_list(), check_all_free_blocks_valid(), check_ptrs_valid_heap_address() 이 5가지 함수로 이루어져 있다.

---

check_free_blocks_marked_free

```
static void check_free_blocks_marked_free() {
    char ** p;

    int i;
    for (i=0; i <= MAX_POWER; i++) {
        if (p = GET_FREE_LIST_PTR(i)) {
            while (p) {
                if (GET_STATUS(p) == TAKEN) {
                    printf("There are free blocks that are marked as taken");
                    assert(0);
                }
            p = GET_SUCC(p);
            }
```

```
        }
    }

    printf("check_free_blocks_marked_free passed.\n");
}
```

먼저 free marking이 되어 있는 free block을 확인하는 과정이 필요한데 이를 위해 check_free_blocks_marked_free() 함수를 정의하고 사용한다. 이는 각 size 별로 나눈 free list의 pointer를 가져오고 각 free list에 있는 block들에 대해 어떤 상태인지 확인한다. 상태가 TAKEN 되었다면 "There are free blocks that are marked as taken"이라는 메시지를 출력하고 assert(0)을 호출한다.

---

check_contiguous_free_block_coalesced()

```
static void check_contiguous_free_block_coalesced() {
    char ** p = heap_ptr;

    while (GET_STATUS(p) != TAKEN && GET_SIZE(p) != 0) {

        char ** next_block = NEXT_BLOCK_IN_HEAP(p);

        if (GET_STATUS(p) == FREE && GET_STATUS(next_block) == FREE) {
        printf("Block %p should coalesce with block %p", p, next_block);
        assert(0);
        }

        p = NEXT_BLOCK_IN_HEAP(p);
    }

    printf("check_contiguous_free_block_coalesced passed.\n");
}
```

free block들이 연속적일 때 coalesce가 성공적으로 되어 있는지 체크하기 위해 check_contiguous_free_block_coalesced()를 구현한다. 어떤 block의 상태가 FREE일 때 다음 next block 상태도 FREE이면 두 block 사이에 coalesced가 되지 않았다는 뜻이므로 "Block _ should coalesce with block _" 라는 에러 메세지를 출력해주고 assert(0)을 호출한다.

---

check_all_free_blocks_in_free_list()

```
static void check_all_free_blocks_in_free_list() {
```

```
    char ** p = heap_ptr;

    while (GET_STATUS(p) != TAKEN && GET_SIZE(p) != 0) { // Haven't hit
epilog
        if (GET_STATUS(p) == FREE) {
            int size = GET_SIZE(p);
            int index = find_free_index (size);

            if (GET_FREE_LIST_PTR(index) == p)
                return; // beginning of free list

            char **prev_block = GET_PRED(p);
            char **next_block = GET_SUCC(p);

            if (!prev_block && !next_block) {
                printf("Free block %p not in free list", p);
                assert(0);
            }
        }
        p = NEXT_BLOCK_IN_HEAP(p);
    }

    printf("check_all_free_blocks_in_free_list passed.\n");
}
```

모든 free block들이 free list에 잘 있는지 확인하기 위해서 check_all_free_blocks_in_free_list 함수를 구현한다. 먼저 어떤 block의 상태가 free일 때 free list에 존재하는지 체크해야 한다. 모든 block에 대해서 이를 확인하, free block이 있는 클래스를 찾아내 그 index의 free list pointer가 그 block과 일치할 경우 이 block들이 free list에 있는 첫 block이란 뜻이므로 return 해준다. 아닌 경우 previous block과 next block이 존재할텐데 만약에 둘 다 존재하지 않는다면 "Free block _ not in free list" 라는 에러 메시지를 출력하고 assert(0)을 호출해준다.

---

check_all_free_blocks_valid()

```
static void check_all_free_blocks_valid_ftr_hdr() {
    char ** p = heap_ptr;
    size_t size_in_hdr;
    size_t size_in_ftr;

    while (GET_STATUS(p) != TAKEN && GET_SIZE(p) != 0) {
        if (GET_STATUS(p) == FREE) {
            // is valid free block
            size_in_hdr = GET_SIZE(p);
```

```
            size_in_ftr = GET_SIZE(FTRP(p));

            // Check size in hdr == size in ftr
            if (size_in_hdr != size_in_ftr) {
                printf("Free block %p has different sizes in hdr and ftr",
p);
                assert(0);
            }

            // Check status in hdr and ftr
            if (GET_STATUS(p) == TAKEN) {
                printf("Free block %p has status as taken in header", p);
                assert(0);
            }

            if (GET_STATUS(FTRP(p)) == TAKEN) {
                printf("Free block %p has status as taken in footer", p);
                assert(0);
            }
        }
        p = NEXT_BLOCK_IN_HEAP(p);
    }

    printf("check_all_free_blocks_valid_ftr_hdr passed.\n");
}
```

check_all_free_blocks_valid()을 구현하여 free block이 valid한지 체크한다. 어떤 block이valid하기 위해서는 block header과 footer의 size가 같아야하고, 상태가 free여야 한다. 만약 header의 상태가 TAKEN이거나 footer의 상태가 TAKEN이면 에러 메시지를 출력하고 assert(0)을 호출한다.

---

check_ptrs_valid_heap_address()

```
static void check_ptrs_valid_heap_address() {
    void * heap_lo = mem_heap_lo();
    void * heap_hi = mem_heap_hi();

    char ** p = heap_ptr;

    do {
        if (!(heap_lo <= p <= heap_hi)) {
            printf("%x not in heap range", p);
            assert(0);
        }
        p = NEXT_BLOCK_IN_HEAP(p);
```

```
    } while (GET_STATUS(p) != TAKEN && GET_SIZE(p) != 0); // Haven't hit
epilog

    printf("check_ptrs_valid_heap_address passed.\n");

}
    printf("check_contiguous_free_block_coalesced passed.\n");
}
```

TAKEN 상태의 block이 valid한 heap address를 가지고 있는지 확인하기 위해 check_ptrs_valid_heap_address 함수를 구현한다. TAKEN 상태의 block이 valid한 heap address를 가르키기 위해서는 heap의 가장 첫 번째 byte와 heap의 가장 마지막 byte사이에 p의 address가 있어야 한다. heap의 가장 첫 번째 byte를 heap_lo에 저장하고, 가장 마지막 byte를 heap_hi에 저장하여 이를 확인한다. 만약 그 사이에 있지 않는다면 "- not in heap range"라는 에러 메시지를 추력하고 assert(0)을 호출한다.

결과적으로 mdriver를 실행시켜보면 until 57, performance 40으로 97/100점을 얻는 것을 볼 수 있다.