컴퓨터 SW 시스템 개론 Lab 12, 13

20220923 이윤혁

1. 개요

이번 Lab 12, 13에서는 dynamic allocation을 이해하고 malloc(), free(), realloc()을 구현한다. 본 보고서에서는 Explicit free list와 first-fit 방식으로 dynamic allocation을 구현했다.

2. 함수 구성

이번 Lab에서는 주어진 함수 5개를 작성한다.

```
int mm_init(void);

void *mm_malloc(size_t size);

void *mm_free(void *ptr);

void *mm_realloc(void *ptr, size_t size);

int mm_check();
```

위 함수 중 mm_check()를 제외하고 구현하는데 필요한 함수들은 다음과 같다.

```
static void* extend_heap(size_t number_of_words);

static void* coalesce(void* block_ptr);

static void insert_free_block(void* block_ptr);

static void delete_free_block(void* block_ptr);

static void* first_fit(size_t size);

static void allocate(void* block_ptr, size_t size);
```

mm_check()를 구현하는데 필요한 함수들은 다음과 같다.

```
static int is_all_marked_free();
```

```
static int is_contiguous_not_escaped();

static int is_all_free_block_in_list();

static int is_all_valid_free_ptr();

static int is_no_overlap();

static int is_all_valid_allocated_ptr();
```

함수들을 하나씩 설명하면서 함수의 기능과 어떻게 동작하는지 설명할 것이다. 보고서의 가독성을 위해 주석을 제거하였다.

다음은 전역 변수이다

```
static void* heap_root;
static void* free_root;
```

write-up lab에 명시된 것처럼 pointer 처리를 위해 매크로를 선언했다. 매크로는 강의 교재에 있는 것을 참고하여 만들었다.

다음은 매크로이다.

```
#define FREE 0
#define ALLOCATED 1

#define WORDSIZE 4
#define DWORDSIZE 8
#define PAGESIZE (1 << 12)

#define GET(ptr) (*(unsigned int *)(ptr))
#define PUT(ptr, val) (*(unsigned int *)(ptr) = (val))

#define GET_SIZE(ptr) (GET(ptr) & ~0x7)
#define GET_IS_ALLOCATED(ptr) (GET(ptr) & 0x1)

#define HEADER_PTR(block_ptr) ((char *)(block_ptr) - WORDSIZE)
#define FOOTER_PTR(block_ptr) ((char *)(block_ptr) + GET_SIZE(HEADER_PTR(block_ptr)) -
DWORDSIZE)

#define NEXT_BLOCK_PTR(block_ptr) ((char *)(block_ptr) + GET_SIZE(((char *)(block_ptr) -
```

```
WORDSIZE)))
#define PREV_BLOCK_PTR(block_ptr) ((char *)(block_ptr) - GET_SIZE(((char *)(block_ptr) -
DWORDSIZE)))

#define PREV_PTR(ptr) ((void*)(ptr) + WORDSIZE)
#define NEXT_PTR(ptr) ((void*)(ptr))
```

매크로의 역할을 유추하기 쉽도록 기능과 직관된 이름을 설정했다.

2.1. int mm_init(void)

이 함수는 heap을 초기화하는 함수이다. 아래는 mm_init(void)의 전체 코드이다.

```
int mm_init(void)
{

    heap_root = mem_sbrk(6 * WORDSIZE);
    if (heap_root == (void*) -1)
        return -1;

    PUT(heap_root, 0);
    PUT(heap_root + 1 * WORDSIZE, NULL);
    PUT(heap_root + 2 * WORDSIZE, NULL);
    PUT(heap_root + 3 * WORDSIZE, 2 * WORDSIZE | 1);
    PUT(heap_root + 4 * WORDSIZE, 2 * WORDSIZE | 1);
    PUT(heap_root + 5 * WORDSIZE, 0 * WORDSIZE | 1);

    free_root = heap_root + 2 * WORDSIZE;
    heap_root += 4 * WORDSIZE;

    if (extend_heap(PAGESIZE / WORDSIZE) == NULL)
        return -1;

    return 0;
}
```

mm_init()은 mm_malloc()의 초기화를 담당하는 함수이다.

prologue와 epilogue, prev next pointer을 위한 공간을 할당해주고 초기화한다. 할당에 실패한 경우 -1을 반환한다. 초기화 후 heap을 Page size(4096) 만큼 늘려준다. 늘리기에 실패한 경우 -1을 반환한다. 함수가 정상 종료되었으면 0을 반환한다.

2.2. void mm_malloc(size_t size)

이 함수는 malloc 함수이다. 아래는 mm_malloc(size_t size)의 전체 코드이다.

```
void *mm_malloc(size_t size)
{
	void* block_ptr;
	size_t block_size;
	size_t extension_size;
	int init_success;

	if (size == 0)
		return NULL;

	if (heap_root == NULL)
		init_success = mm_init();
		if (init_success == -1)
			return NULL;
	}

	block_size = ALIGN(size) + 2 * WORDSIZE;

	block_ptr = first_fit(block_size);

	if (block_ptr == NULL) {
		extension_size = block_size > PAGESIZE ? block_size : PAGESIZE;
		block_ptr = extend_heap(extension_size / WORDSIZE);
		if (block_ptr == NULL)
			return NULL;
	}

	allocate(block_ptr, block_size);

	return block_ptr;
}
```

size가 0인 경우 할당이 의미가 없으므로 NULL을 반환하며 함수를 조기 종료한다. heap의 시작점인 heap_root가 NULL인 경우 mm_init()을 호출해 heap을 초기화한다. heap 초기화에 실패한 경우 NULL을 반환하며 함수를 조기종료한다. 8-byte block단위로 allocation을 진행하기 위해 size를 8-byte align한다. 그 후 first_fit 함수를 호출해 free list의 free block 중 적합한 함수가 있는지 탐색하고, 그렇지 않은 경우 heap을 PAGESIZE와 size중 큰 값만큼 늘린다. heap 늘리기에 실패한 경우 NULL을 반환하며 함수를 조기종료한다. allocate 함수를 호출해 주어진 포인터에 align된 사이즈만큼 heap에 할당한다. 할당 후 포인터를 반환한다.

2.3. void mm_free(void *ptr)

이 함수는 free 함수이다. 아래는 mm_free(void *ptr)의 전체 코드이다.

```
void mm_free(void *ptr)
{
	size_t size = GET_SIZE(HEADER_PTR(ptr)); // Size of current block

	PUT(NEXT_PTR(ptr), NULL) ;
	PUT(PREV_PTR(ptr), NULL);
	PUT(HEADER_PTR(ptr), size | FREE);
	PUT(FOOTER_PTR(ptr), size | FREE);

	coalesce(ptr);

	return;
}
```

free 하려는 포인터로부터 size를 읽는다. 포인터의 prev와 next pointer를 NULL로 초기화해 free block으로 초기화하고 Header와 Footer를 동일크기 free block으로 업데이트한다.

그 후 coalesce()를 호출해 coalesce를 진행한다.

2.4. void *mm_realloc(void *ptr, size_t size)

이 함수는 realloc 함수이다. 아래는 realloc(void *ptr, size_t size)의 전체코드이다.

```
void *mm_realloc(void *ptr, size_t size)
{
	void* next_block_ptr = NEXT_BLOCK_PTR(ptr);
	void* newptr;
	void* surplus_block_ptr;
```

```
    size_t is_next_allocated = GET_IS_ALLOCATED(HEADER_PTR(NEXT_BLOCK_PTR(ptr)));
    size_t old_size;
    size_t next_size;
    size_t allocate_size;

    if (ptr == NULL)
        return mm_malloc(size);

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

    size = ALIGN(size) + 2 * WORDSIZE;
    old_size = GET_SIZE(HEADER_PTR(ptr));
    next_size = GET_SIZE(HEADER_PTR(NEXT_BLOCK_PTR(ptr)));

    if (size > old_size) {
        if (next_size >= size - old_size && !is_next_allocated) {
            delete_free_block(next_block_ptr);

            PUT(HEADER_PTR(ptr), old_size + next_size | ALLOCATED);
            PUT(FOOTER_PTR(ptr), old_size + next_size | ALLOCATED);

            return ptr;
        }

        else if (next_size < size - old_size && !is_next_allocated) {

            allocate_size = size - (old_size + next_size) > PAGESIZE ? size - (old_size +
next_size) : PAGESIZE;
            if (extend_heap(allocate_size / PAGESIZE) == NULL) {
                return NULL; // Extend heap failed
            }
            size += allocate_size;

            PUT(HEADER_PTR(ptr), size | ALLOCATED);
            PUT(FOOTER_PTR(ptr), size | ALLOCATED);
```

```
                return ptr;
            }

            else{
                newptr = mm_malloc(size);
                allocate(newptr, size);
                memcpy(newptr, ptr, size);
                mm_free(ptr);

                return newptr;
            }
        }

        return ptr;
}
```

주어진 포인터가 NULL인 경우 mm_malloc으로 기능을 대체하여 반환한다. 주어진 size가 0인 경우 mm_free로 기능을 대체하고 NULL을 반환한다.

위와 같은 특수케이스가 아닌 경우 size를 8-byte align하고 Header, Footer를 위한 space를 추가한다. 현재 포인터 블록의 사이즈와 주소상 다음 블록의 사이즈를 매크로를 이용해 추출한다.

입력 사이즈가 현재 포인터 블록의 사이즈보다 큰 경우 케이스를 3가지로 나눌 수 있다.

1. 주소상 다음 블록이 free이고 입력 사이즈 <= 현재 블록 사이즈 + 다음 블록 사이즈
2. 주소상 다음 블록이 free이고 입력 사이즈 > 현재 블록 사이즈 + 다음 블록 사이즈
3. 주소상 다음 블록이 allocated

1번의 경우 next block을 free list에서 삭제하고 현재 블록 사이즈 + 다음 블록 사이즈를 새 사이즈로 Header와 Footer에 입력한다. 그 후 포인터를 반환한다.

2번의 경우 heap을 extend한다. extend하는 사이즈는 입력 사이즈 – (현재 블록 사이즈 + 다음 블록 사이즈) 와 PAGESIZE 중 최댓값이다. 그 후 입력 사이즈를 업데이트한다. 1번과 마찬가지로 next block을 free list에서 삭제하고 새 사이즈를 Header와 Footer에 입력한다. 그후 포인터를 반환한다.

3번의 경우 새로운 공간에 mm_malloc으로 새로운 공간을 할당하고 현재 블록의 내용을 새로 할당한 블록으로 옮긴다.(memcpy) 그 후 예전 공간을 mm_free를 이용해 할당 해제한다.

3번에서 사용하는 memcpy는 시간이 오래 걸리고 메모리를 많이 잡아먹게 만드는 원흉이므로 1번과 2번과 같은 케이스에서 memcpy를 사용하지 않도록 짜는 것이 최적화의 핵심 요인이다.

2.5. static void* extend_heap(size_t number_of_words)

이 함수는 heap을 주어진 word의 수만큼 늘려주는 함수이다. 아래는 이 함수의 전체코드이다.

```
static void* extend_heap(size_t number_of_words) {
	void* block_ptr;
	size_t size;



	size = (number_of_words % 2 == 0) ? number_of_words * WORDSIZE :
(number_of_words + 1) * WORDSIZE;

	block_ptr = mem_sbrk(size);

	if ((long) block_ptr == -1)
		return NULL;

	PUT(NEXT_PTR(block_ptr), NULL);
	PUT(PREV_PTR(block_ptr), NULL);
	PUT(HEADER_PTR(block_ptr), size | FREE);
	PUT(FOOTER_PTR(block_ptr), size | FREE);
	PUT(HEADER_PTR(NEXT_BLOCK_PTR(block_ptr)), 0 | ALLOCATED);

	return coalesce(block_ptr);
}
```

주어진 number_of_words로부터 size를 계산한다. size는 number_of_words가 짝수가 되도록 계산한다. 그 후 mem_sbrk() 를 이용해 heap에 size만큼 공간을 할당한다. 할당에 실패한 경우 NULL을 반환하고 함수를 조기종료한다. 그 후 할당한 공간을 free block으로 만들기 위해 초기화한다. free block의 구성요소인 Header, Footer 및 prev, next pointer를 초기화해준다. 그리고 heap의 끝에 새로운 epilogue를 생성하여 heap의 끝을 명시한다.

새로운 free block이 생겼으므로 coalesce()를 호출해 coalesce를 진행하고 포인터를 반환한다.

2.6. static void* coalesce(void* block_ptr)

이 함수는 coalesce를 진행하는 함수이다. 아래는 이 함수의 전체 코드이다.

```
static void* coalesce(void* block_ptr) {
    size_t size = GET_SIZE(HEADER_PTR(block_ptr));
    size_t prev_size = GET_SIZE(FOOTER_PTR(PREV_BLOCK_PTR(block_ptr)));
    size_t next_size = GET_SIZE(HEADER_PTR(NEXT_BLOCK_PTR(block_ptr)));
    void* prev_block_ptr = PREV_BLOCK_PTR(block_ptr);
    void* next_block_ptr = NEXT_BLOCK_PTR(block_ptr);
    size_t is_prev_allocated = GET_IS_ALLOCATED(FOOTER_PTR(prev_block_ptr));
    size_t is_next_allocated = GET_IS_ALLOCATED(HEADER_PTR(next_block_ptr));

    if (is_prev_allocated && is_next_allocated) {
        insert_free_block(block_ptr);

        return block_ptr;
    }

    else if (is_prev_allocated && !is_next_allocated) {
        delete_free_block(next_block_ptr);
        size += next_size;

        PUT(HEADER_PTR(block_ptr), size | FREE);
        PUT(FOOTER_PTR(block_ptr), size | FREE);

        insert_free_block(block_ptr);

        return block_ptr;
    }

    else if (!is_prev_allocated && is_next_allocated) {
        delete_free_block(prev_block_ptr);
        size += prev_size;
        PUT(HEADER_PTR(prev_block_ptr), size | FREE);
        PUT(FOOTER_PTR(block_ptr), size | FREE);

        insert_free_block(prev_block_ptr);

        return prev_block_ptr;
```

```
    }

    else if (!is_prev_allocated && !is_next_allocated) {
        delete_free_block(prev_block_ptr);
        size += prev_size;

        delete_free_block(next_block_ptr);
        size += next_size;

        PUT(HEADER_PTR(prev_block_ptr), size | FREE);
        PUT(FOOTER_PTR(next_block_ptr), size | FREE);

        insert_free_block(prev_block_ptr);

        return prev_block_ptr;
    }

    return NULL;
}
```

coalesce는 4가지의 케이스가 있다.

1. 이전 블록이 allocated이고 다음 블록이 allocated
2. 이전 블록이 allocated이고 다음 블록이 free
3. 이전 블록이 free이고 다음 블록이 allocated
4. 이전 블록이 free이고 다음 블록이 free

1번의 경우 coalesce할 수 없으므로 현재 블록을 free list에 추가하고 포인터를 반환하며 함수를 종료한다.

2번의 경우 현재 블록이 다음 블록과 coalesce할 수 있다. 현재 블록과 다음 블록의 사이즈 합을 각각 현재 블록의 Header와 다음 블록의 Footer에 업데이트한다. 그 후 현재 블록을 free list에 추가하고 포인터를 반환하며 함수를 종료한다.

3번의 경우 현재 블록이 이전 블록과 coalesce할 수 있다. 이전 블록과 현재 블록의 사이즈 합을 각각 이전 블록의 Header와 현재 블록의 Footer에 업데이트한다. 그 후 이전 블록을 free list에 추가하고 포인터를 반환하며 함수를 종료한다.

4번의 경우 이전 블록, 현재 블록, 다음 블록 모두 coalesce할 수 있다. 이전 블록과 현재 블록, 다음 블록의 사이즈 합을 각각 이전 블록의 Header와 다음 블록의 Footer에 업데이트한다. 그 후 이전 블록을 free list에 추가하고 포인터를 반환하며 함수를 종료한다.

1, 2, 3, 4 외의 케이스는 없으며 이 케이스들 중 하나라도 해당되어 정상작동했다면 함수는 정상종료한 것이다.

2.7. static void insert_free_block(void* block_ptr)

이 함수는 free list에 free block을 insert하는 함수이다. 아래는 이 함수의 전체 코드이다.

```
static void insert_free_block(void* block_ptr) {
	void* first_block_ptr = GET(free_root);

	if (first_block_ptr != NULL)
		PUT(PREV_PTR(first_block_ptr), block_ptr);

	PUT(NEXT_PTR(block_ptr), first_block_ptr);
	PUT(PREV_PTR(block_ptr), NULL);

	PUT(free_root, block_ptr);

	return;
}
```

free_root(free list의 head)에서 free list의 첫 번째 free block을 읽어온다.

첫 번째 free block이 있는 경우 첫 번째 free block의 이전 블록 포인터로 현재 free block을 설정한다.

그 후 현재 블록의 다음 블록 포인터로 첫 번째(였던) free block을 설정하고 현재 블록의 이전 블록 포인터를 NULL로 초기화한다.

그 후 free_root의 내용을 현재 블록으로 업데이트하고, 함수를 정상종료한다.

이 과정을 통해 linked list인 free list에 LIFO 방식으로 free block을 업데이트 할 수 있다.

2.8. static void delete_free_block(void* block_ptr)

이 함수는 free list에 있는 free block을 제거하는 함수이다. 아래는 이 함수의 전체 코드이다.

```
static void delete_free_block(void* block_ptr) {
```

```
    void* prev_ptr = GET(PREV_PTR(block_ptr)); // Pointer of previous block
    void* next_ptr = GET(NEXT_PTR(block_ptr)); // Pointer of next block

    if (prev_ptr != NULL && next_ptr != NULL) {
        PUT(PREV_PTR(next_ptr), prev_ptr);
        PUT(NEXT_PTR(prev_ptr), next_ptr);
    }

    else if (prev_ptr != NULL && next_ptr == NULL) {
        PUT(NEXT_PTR(prev_ptr), next_ptr);
    }

    else if (prev_ptr == NULL && next_ptr != NULL) {
        PUT(PREV_PTR(next_ptr), NULL);
        PUT(free_root, next_ptr);
    }

    else if (prev_ptr == NULL && next_ptr == NULL) {
        PUT(free_root, NULL);
    }

    PUT(NEXT_PTR(block_ptr), NULL);
    PUT(PREV_PTR(block_ptr), NULL);

    return;
}
```

free block을 삭제할 때 4가지 케이스가 있다.

1. 이전 블록이 존재하고 다음 블록이 존재
2. 이전 블록이 존재하고 다음 블록이 존재하지 않음
3. 이전 블록이 존재하지 않고 다음 블록이 존재함
4. 이전 블록이 존재하지 않고 다음 블록이 존재하지 않음

1번의 경우 다음 블록의 이전 블록 포인터를 이전 블록으로 대체하고, 이전 블록의 다음 블록 다음 블록으로 대체한다.

2번의 경우 이전 블록의 다음 블록 포인터를 다음 블록으로 대체한다.

3번의 경우 다음 블록의 이전 블록 포인터를 NULL로 초기화하고, free list의 head인 free_root에 다음 블록의 포인터를 집어넣는다.

4번의 경우 free_root에 NULL을 집어넣는다.

1, 2, 3, 4 외의 케이스는 없으며 이 케이스들 중 하나가 완료되면 현재 블록의 다음 블록 포인터와 이전 블록 포인터를 NULL로 초기화한다. 잘못된 접근이 일어났을 때 포인터가 이상한 방향으로 가지 않도록 막기 위함이다.

전술한 과정을 거친 후 함수를 정상종료한다.

2.9. static void* first_fit(size_t size)

이 함수는 first fit 방식으로 size에 맞는 free list를 반환하는 함수이다. 아래는 이 함수의 전체 코드이다.

```
static void* first_fit(size_t size) {
	void* block_ptr;

	for(block_ptr = GET(free_root);
		block_ptr != NULL;
		block_ptr = GET(NEXT_PTR(block_ptr))){
		if(size > GET_SIZE(HEADER_PTR(block_ptr)))
			continue;

		return block_ptr;
	}

	return NULL;
}
```

free list의 첫 번째 free block부터 시작하여, free list의 끝까지 탐색한다. free block의 탐색이 끝나면 다음 free block으로 넘어간다. 탐색 중일 때, 현재 free block의 size가 입력 사이즈보다 작은 경우 continue를 이용해 다음 반복문으로 조기에 넘어간다. 현재 free block의 size가 입력 사이즈보다 큰 경우 그 free block의 포인터를 반환한다.

2.10. static void allocate(void* block_ptr, size_t size)

이 함수는 heap에 주어진 포인터에 사이즈만큼 공간을 할당하는 함수이다. 아래는 이 함수의

전체 코드이다.

```
static void allocate(void* block_ptr, size_t size) {

    size_t free_block_size = GET_SIZE(HEADER_PTR(block_ptr));
    size_t surplus_size = free_block_size - size;
    void* surplus_block_ptr;

    delete_free_block(block_ptr);

    if (surplus_size <= 4 * DWORDSIZE){
        PUT(HEADER_PTR(block_ptr), free_block_size | ALLOCATED);
        PUT(FOOTER_PTR(block_ptr), free_block_size | ALLOCATED);
        return;
    }

    PUT(HEADER_PTR(block_ptr), size | ALLOCATED);
    PUT(FOOTER_PTR(block_ptr), size | ALLOCATED);

    surplus_block_ptr = NEXT_BLOCK_PTR(block_ptr);
    PUT(NEXT_PTR(surplus_block_ptr), NULL);
    PUT(PREV_PTR(surplus_block_ptr), NULL);
    PUT(HEADER_PTR(surplus_block_ptr), surplus_size | FREE);
    PUT(FOOTER_PTR(surplus_block_ptr), surplus_size | FREE);

    coalesce(surplus_block_ptr);

    return;
}
```

먼저 주어진 포인터의 블록을 free list에서 제거한다. 현재 블록의 size와 입력 size가 다를 수 있는데, 잉여 사이즈(surplus_size)를 현재 블록의 size – 입력 size로 계산한다. 잉여 사이즈가 4 * DOUBLE WORD SIZE보다 크지 않은 현재 free block에 HEADER와 FOOTER를 free block의 size로 업데이트한다. 잉여 사이즈가 4 * DOUBLE WORD SIZE보다 큰 경우 free block 전체에 그대로 할당 했다간 fragmentaion이 심해진다. 그러므로 현재 free block에 입력 size 만큼 allocation을 진행하고, 남는 잉여 블록(surplus block)을 free block으로 초기화해주고 coalesce를 진행해준다. first fit은 fragments들이 크게 생성될 수 있는 확률이 높으므로, fragment들의 사이즈를 줄이도록 allocation시 surplus block을 생성해주면 first fit의 문제 점을 어느정도 해

소할 수 있다. 이 부분 또한 최적화의 핵심 요인이다.

2.11. int mm_check()

이 함수는 디버깅 함수로, Heap에 allocation과 free가 제대로 일어나고 있는지 확인해주는 역할을 가진다. 아래는 이 함수의 전체 코드이다.

```
int mm_check(){
    int status = is_all_marked_free() | is_contiguous_not_escaped() | is_all_free_block_in_list()
| is_all_valid_free_ptr() | is_no_overlap() | is_all_valid_allocated_ptr();

    printf("heap check status: %d₩n", status);
    printf("Is every block in the free list marked as free: %d₩n", status & (1 << 0));
    printf("Are there any contiguous free blocks that somehow escaped
coalescing?: %d₩n", status & (1 << 1));
    printf("Is every free block actually in the free list?: %d₩n", status & (1 << 2));
    printf("Do the pointers in the free list point to valid free blocks?: %d₩n", status & (1
<< 3));
    printf("Do any allocated blocks overlap?: %d₩n", status & (1 << 4));
    printf("Do the pointers in a heap block point to valid heap addresses?: %d₩n", status
& (1 << 5));

    return status == 0x3f;
}
```

status에 bit 단위로 write-up lab에 적힌 6가지 조건을 저장한다. 그 후 status 자체와 각 조건마다 heap allocation과 free가 잘 진행되었는지 한 줄씩 출력한다.

모든 출력 이후 6가지 조건이 모두 만족되었다면 1, 그렇지 안하면 0을 반환하고 함수를 정상 종료한다.

2.12. int mm_check()에 사용된 함수들

이 함수는 write-up lab의 "Is every block in the free list marked as free?"를 검사하는 함수이다. 아래는 이 함수의 전체 코드이다.

```
static int is_all_marked_free() {

```

```
    int flag = 1 << 0;
    void* block_ptr;

    for(block_ptr = GET(free_root); block_ptr != NULL; block_ptr =
GET(NEXT_PTR(block_ptr))){
 if(GET_IS_ALLOCATED(HEADER_PTR(block_ptr)) == FREE &&
GET_IS_ALLOCATED(FOOTER_PTR(block_ptr)) == FREE)
            continue;
        return 0;
    }

    return flag;
}
```

free list을 순차적으로 훑으면서 현재 free block의 header와 footer에 allocation bit가 0인지 검사한다. 하나라도 0이 아닌 블록이 존재시 0을 반환하고, 모두 0인 경우 1을 반환한다.

이 함수는 write-up lab의 "Are there any contiguous free blocks that somehow escaped coalescing?" 조건을 검사하는 함수이다. 아래는 이 함수의 전체 코드이다.

```
static int is_contiguous_not_escaped() {
    int flag = 1 << 1;
    void* block_ptr;

    for(block_ptr = GET(free_root); block_ptr != NULL; block_ptr =
GET(NEXT_BLOCK_PTR(block_ptr))) {
        if(NEXT_BLOCK_PTR(block_ptr) != NULL) {
            if(GET_IS_ALLOCATED(HEADER_PTR(NEXT_PTR(block_ptr))) == FREE)
                return 0;
        }

        if(PREV_BLOCK_PTR(block_ptr) != NULL) {
            if(GET_IS_ALLOCATED(HEADER_PTR(PREV_BLOCK_PTR(block_ptr))) == FREE)
                return 0;
        }
    }

    return flag;
}
```

free list을 순차적으로 훑으면서 현재 free block에 주소상 인접한 이전 블록과 다음 블록이 free인지 확인한다. free인 블록이 존재 시 0을 반환하고, 모든 free block에 대해서 이전 블록과 다음 블록이 모두 allocated 블록이면 1을 반환한다.

이 함수는 write-up lab의 "Is every free block actually in the free list?"의 조건을 검사하는 함수이다. 아래는 이 함수의 전체 코드이다.

```
static int is_all_free_block_in_list() {
	int flag = 1 << 2;
	void* block_ptr;
	void* temp_ptr;

	for (block_ptr = heap_root; block_ptr != NULL; block_ptr =
GET(NEXT_BLOCK_PTR(block_ptr))) {
		if (GET_IS_ALLOCATED(HEADER_PTR(block_ptr)) == FREE) {
			temp_ptr = GET(free_root);
			while(temp_ptr != NULL){
				if(temp_ptr == block_ptr)
					break;
				temp_ptr = GET(NEXT_PTR(temp_ptr));
			}

			if (temp_ptr == NULL)
				return 0;
		}
	}

	return flag;
}
```

Heap을 순차적으로 훑으면서 free block을 탐색한다. free block이 탐색되면 그 free block이 free list 안에 있는지 free list를 순차적으로 훑으며 탐색한다. free list안 에서 현재 free block이 탐색되지 않는 경우 0을 반환하고, 모든 free block이 free list안에서 발견되면 1을 반환한다.

이 함수는 write-up lab의 "Do the pointers in the free list point to valid free blocks?" 조건을 검사하는 함수이다. 아래는 이 함수의 전체 코드이다.

```
static int is_all_valid_free_ptr() {
	int flag = 1 << 3;
	void* block_ptr;

	for (block_ptr = NEXT_BLOCK_PTR(heap_root); block_ptr < mem_heap_hi(); block_ptr = NEXT_BLOCK_PTR(block_ptr)) {
		if (GET_IS_ALLOCATED(HEADER_PTR(block_ptr)) == FREE){
			if(!(mem_heap_lo() <= HEADER_PTR(block_ptr) && FOOTER_PTR(block_ptr) <= mem_heap_hi()) || (GET(HEADER_PTR(block_ptr)) & 0x6 != 0))
				return 0;
		}
	}

	return flag;
}
```

heap을 순차적으로 훑으면서 현재 block이 free라면, 현재 block의 header와 footer과 heap의 시작과 끝 사이에 있는지 검사한다. heap의 시작과 끝 사이에 없는 경우 0을 반환한다. 또한 block의 size를 검사하여, size가 8의 배수가 아니라면 0을 반환한다. 모든 free block에 대해 0을 반환하지 않은 경우, 1을 반환한다.

이 함수는 write-up lab의 "Do any allocated blocks overlap" 조건을 검사하는 코드이다. 아래는 이 함수의 전체 코드이다.

```
static int is_no_overlap() {
	int flag = 1 << 4;
	void* block_ptr;

	for(block_ptr = heap_root; block_ptr != NULL && GET_SIZE(HEADER_PTR(block_ptr)) != 0; block_ptr = NEXT_BLOCK_PTR(block_ptr)) {
		if(GET_IS_ALLOCATED(HEADER_PTR(block_ptr)) == FREE)
			continue;

		if(NEXT_BLOCK_PTR(block_ptr) != NULL){
			if(GET_IS_ALLOCATED(HEADER_PTR(NEXT_BLOCK_PTR(block_ptr))) == FREE)
				continue;

			if(FOOTER_PTR(block_ptr) > HEADER_PTR(NEXT_BLOCK_PTR(block_ptr)))
				return 0;
```

```
        }
    }

    return flag;
}
```

heap을 순차적으로 훑으며, 현재 블록이 allocated block이고, 다음 블록이 allocated block이면 현재 블록의 footer와 다음 블록의 header의 크기를 검사한다. 현재 블록의 footer가 다음 블록의 header보다 크면, allocated block끼리 겹치는 것이므로 0을 반환한다. 모든 allocated block이 겹치지 않는 경우 1을 반환한다.

이 함수는 write-up lab의 "Do the pointers in a heap block point to valid heap addresses?"의 조건을 검사하는 함수로, 아래는 이 함수의 전체 코드이다.

```
static int is_all_valid_allocated_ptr(){
    int flag = 1 << 5;
    void* block_ptr;

    for (block_ptr=NEXT_BLOCK_PTR(heap_root); block_ptr < mem_heap_hi(); block_ptr = NEXT_BLOCK_PTR(block_ptr)) {
        if (GET_IS_ALLOCATED(HEADER_PTR(block_ptr)) == ALLOCATED){
            if(!(mem_heap_lo() <= HEADER_PTR(block_ptr) && FOOTER_PTR(block_ptr) <= mem_heap_hi()) || (GET(HEADER_PTR(block_ptr)) & 0x6 != 0))
                return 0;
        }
    }

    return flag;
}
```

heap을 순차적으로 훑으면서 현재 block이 allocated라면, 현재 block의 header와 footer과 heap의 시작과 끝 사이에 있는지 검사한다. heap의 시작과 끝 사이에 없는 경우 0을 반환한다. 또한 block의 size를 검사하여, size가 8의 배수가 아니라면 0을 반환한다. 모든 allocated block에 대해 0을 반환하지 않은 경우, 1을 반환한다.

3. 결과

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   89%    5694  0.000125 45588
 1       yes   89%    4805  0.000114 42335
 2       yes   97%   12000  0.000104115830
 3       yes   97%    8000  0.000066121396
 4       yes   90%   24000  0.000185129660
 5       yes   90%   16000  0.000118135939
 6       yes   92%    5848  0.000082 71404
 7       yes   92%    5032  0.000071 70973
 8       yes   66%   14400  0.000130110769
 9       yes   66%   14400  0.000130110855
10       yes   94%    6648  0.000188 35324
11       yes   94%    5683  0.000174 32623
12       yes   96%    5380  0.000162 33189
13       yes   96%    4537  0.000155 29252
14       yes   88%    4800  0.000282 16997
15       yes   88%    4800  0.000283 16979
16       yes   85%    4800  0.000307 15635
17       yes   85%    4800  0.000307 15645
18       yes   97%   14401  0.000100144299
19       yes   97%   14401  0.000101142867
20       yes   38%   14401  0.000099146203
21       yes   38%   14401  0.000098146501
22       yes   66%      12  0.000000 60000
23       yes   66%      12  0.000000 60000
24       yes   89%      12  0.000000 60000
25       yes   89%      12  0.000000120000
Total          84%  209279  0.003380 61917

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

./mdriver 명령어를 이용해 90점이라는 성능 점수를 얻을 수 있었다. 점수가 나오는 것으로 보아 정확성에는 문제가 없다고 판단된다.

4. 결론

이번 Lab을 통해 Dynamic allocation의 원리에 대해서 이해할 수 있었다. gprofile을 이용해 어느 곳에서 작동 시간이 오래 걸리는지 개략적으로 파악할 수 있었다. 최종적으로 Explicit Free list와 First-fit 방식을 이용했다. 보고서와 코드에는 드러나지 않으나, fit 방식은 다양한 방식으로 테스트했다. 테스트 중 의문이 들었던 점은 Best fit 방식을 이용해 memory utill을 높이려 했으나 오히려 떨어지는 결과를 얻은 것이다. Best fit으로 인해 작은 memory fragment들이 쌓여서 결과적으로 memory util이 낮아진 것으로 추정된다.