

# **System Programming Project 3**

담당 교수 : 김영재

이름 : 박준혁

학번 : 20171643

## 1. Purpose

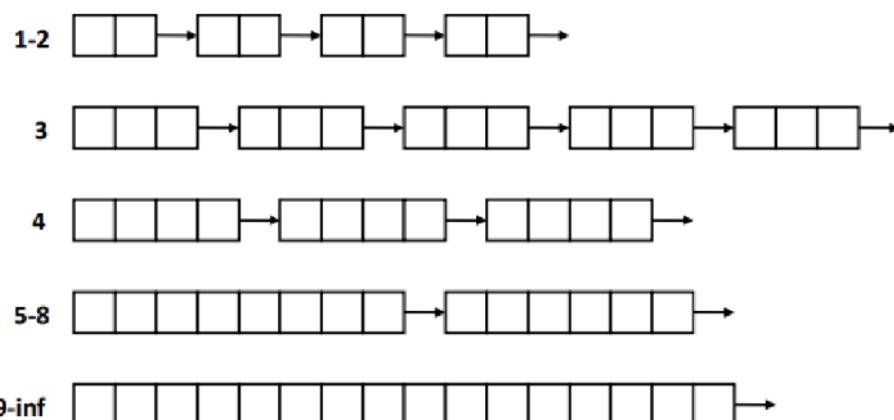
본 프로젝트는 libc 라이브러리에서 제공하는 malloc, realloc, free와 동등한 기능을 수행하는 Dynamic Memory Allocator를 직접 개발해봄으로써 강의 시간에 다룬 'Dynamic Memory Allocation', 'First Fit / Best Fit Search', 'Implicit / Explicit / Segregated Free List', 'Peak Memory Utilization', 'Speedup' 등의 개념을 익히는 것을 목표로 한다.

프로젝트 개발은 점진적으로 수행되었다. 먼저 가장 Naive한 Implicit Free List 기법을 이용해 단순한 Dynamic Memory Allocator를 만든다. 이어서, Boundary Tags 개념을 적용해보고, Explicit Free List 형태로 확장시킨다. 가장 마지막으로는 Explicit List를 기반으로 한 Segregated Free List 기법을 구현한다. 이 과정에서 제공된 mdriver 평가 프로그램과, 개발 과정에서 본인이 직접 만든 'Customized Heap Consistency Checker'를 이용해 mm.c Interface의 범용성과 정확성을 높이고자 노력하였다. 본 Document는 그 중에서 가장 마지막 단계로 개발한 Allocator 이자, 본 프로젝트의 제출물인 'Segregated Free List' 기법 적용 Dynamic Memory Allocator에 대해 다룬다. 실제 제출 코드 역시 마찬가지로 동일한 Allocator의 구현 결과물이다.

## 2. Allocator Design

### A. Why 'Segregated Free List'?

본 프로젝트의 최종 개발 프로그램은 'Segregated Free List' 기법을 채택한다. 'Segregated Free List Method'는 'Free Block의 Size'에 대해 각각의 Class를 마련하는 방식으로, 다음과 같은 간단한 구조로 설명할 수 있다.



특정 Size 혹은 Size Range를 기준으로 Class를 만들어, 해당 Size에 대응하는 Free Block들을 List 형태로 묶어 'List별'로 관리하는 것이다. 강의 시간에 학습한 것처럼, 주로  $2^k$ 을 단위로 클래스를 구분한다. 따라서, 본 프로젝트 개발에서도 마찬가지로 클래스를 구분할 것이다.

이러한 Segregated 방식의 가장 큰 장점은 아무래도 Implicit이나 Explicit Method와 다르게 '전체 Heap 공간 Traversal'이 필요치 않다는 것이다. 요청한 Size에 대응하는 Size Class 위주로 탐색하면 된다. 즉, Allocation을 위한 Free Block 탐색의 시간이 상당히 효율적인 것이다.

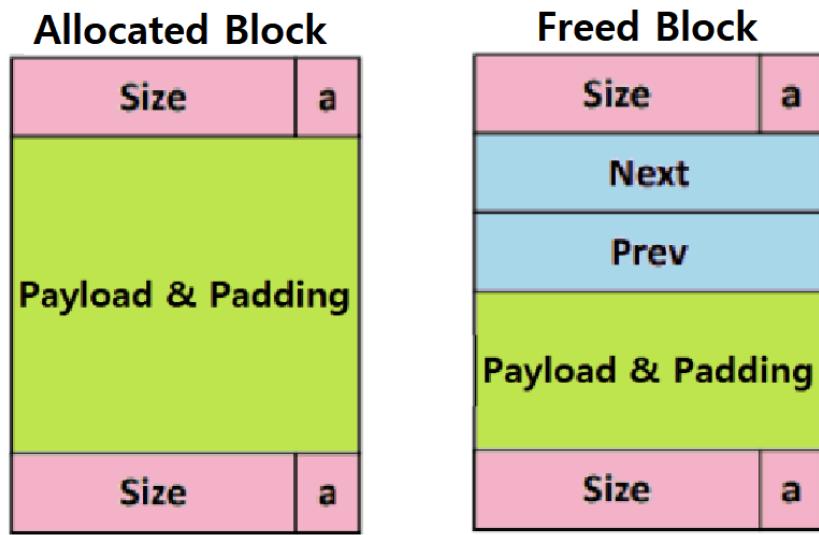
따라서, 본인은 이러한 '시간적 효율'을 최대한 이용하고자 'Segregated Free List' 기법을 적용하기로 결정했다. 단, 각 Class에 해당하는 List는 용이한 탐색 및 삽입/삭제를 위해 Doubly Linked List로 구현한다. 이때, Doubly Linked List란, Explicit Free List를 의미한다. 단순히 'Next Block'만 가리키는 Implicit Free List의 경우, Boundary Tags를 사용하더라도 상대적으로 Explicit에 비해 Block 관리의 용이성이 떨어지기 때문이다. Pointer를 함유하고 있는 Explicit Free List를 채택할 경우, 상당히 효과적으로 Free Block들을 관리할 수 있다.

## B. Block Structure

하지만, 알다시피 Explicit Free List는 Internal Fragmentation이 높다. Header, Footer, Next Block Pointer, Previous Block Pointer, 총 4개의 Extra Words를 필요로 하기 때문이다. 즉, 공간 효율 관점에서 치명적인 단점을 만들 수 있는 것이다. 이에 대해 본인은 장시간 고민 끝에 다음과 같은 아이디어를 떠올렸다.

***"Allocated Block일 때는 2개의 Pointer를 따로 두지 말고, 그냥 Boundary Tags만 유지한다. Block이 Free될 때만 2개의 Pointer를 만들어주는 것이다."***

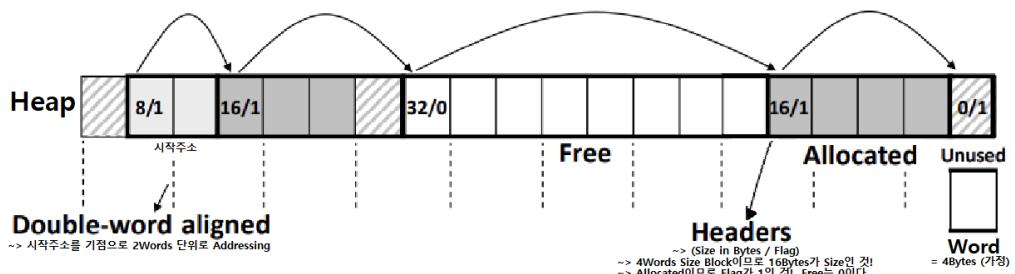
즉, Implicit Free List와 Explicit Free List의 장점을 합치는 것이다. Implicit Free List가 상대적으로 내부 단편화가 적다는 점, 그리고 Explicit Free List가 상대적으로 Block 관리가 용이하다는 점을 동시에 활용하는 것이다. 이를 토대로 본 프로젝트에서 채택한 Block의 구조를 그려보면 다음과 같다. Block의 상태에 따라 구조가 다름에 주목하자.



이는 Allocation 시에는 좌측과 같이 Block 내부를 구성하고, Freeing 시에는 우측과 같이 Block 내부를 조정하는 식으로 구현할 수 있을 것이다. 자세한 Code-Level Detail은 아래에서 설명한다.

### C. Heap Structure

위와 같은 Block Structure를 토대로 Heap은 아래와 같이 Block Sequence로 나타낼 수 있다.



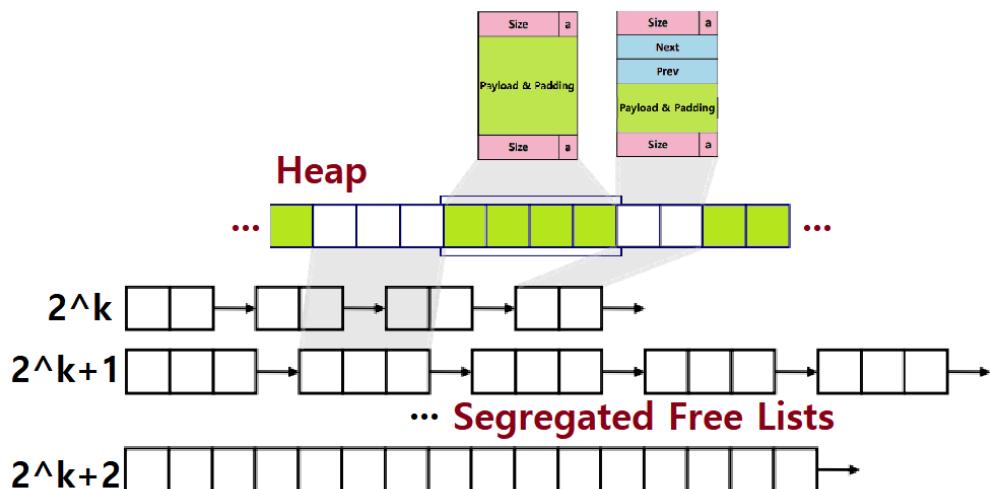
Dynamic Memory Allocator는 항상 실제 Data가 위치한 Payload의 시작점을 사용자에게 반환해주어야 한다. 그래야 Allocation의 의미가 실현되기 때문이다. 이때, Double-Word Alignment System이라면, 위의 그림과 같이 Heap의 시작 주소에 1 Word 크기의 Padding과 2 Words의 '자리만 차지하는 Block'을 두어, Heap에 생성되는 Block들이 모두 8-Bytes Alignment를 이루도록 만들어준다. Heap의 시작 주소에 있는 2 Words의 '자리만 차지하는 Block'들은 다음과 같은 역할도 수행한다.

- malloc, free, realloc 시의 Block 관리 Operation에서 최-좌측 Edge Case를 자동으로 처리할 수 있게 해준다.
  - 어떻게? 2 Words 중 앞의 Word가 Header처럼, 뒤의 Word가 Footer처럼 작동해서 말이다.
  - ◆ 이를 Prologue라고도 부른다고 한다. (외부 자료 인용)

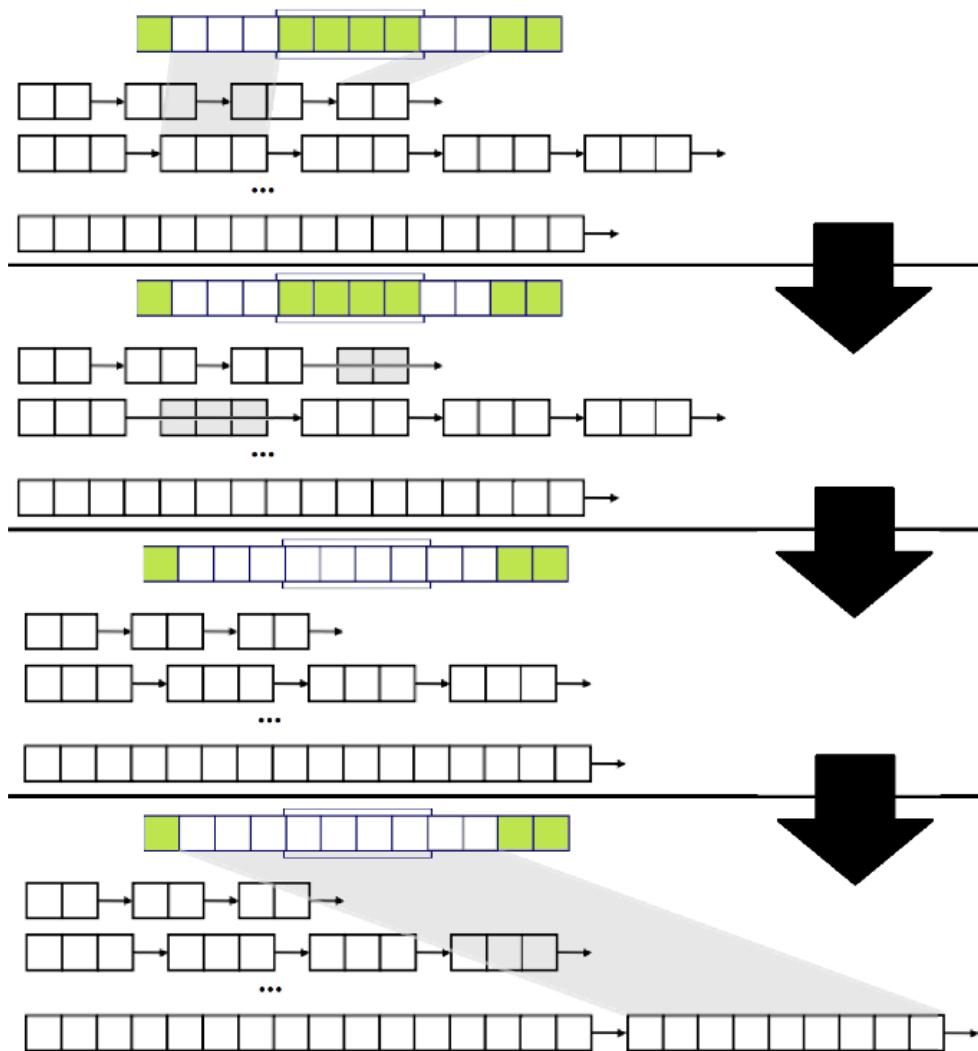
아무튼, 위와 같은 Linear Structure의 형태로 Heap이 Alignment를 지켜가며 구성된다. 각 Block은 복수의 Words로 구성된다. 이때, Heap의 맨 끝을 가리키는 Word도 하나 필요하다. Right-Side Edge Case 처리를 위해 말이다. 따라서, 위의 그림과 같이 'Unused Word' 하나를 더 추가해준다. Heap은 이러한 형태로 구성된다. Free List Method와 관계없이, 기본적으로 위와 같은 구조를 갖추는 것이다.

#### D. Logical View & Reality

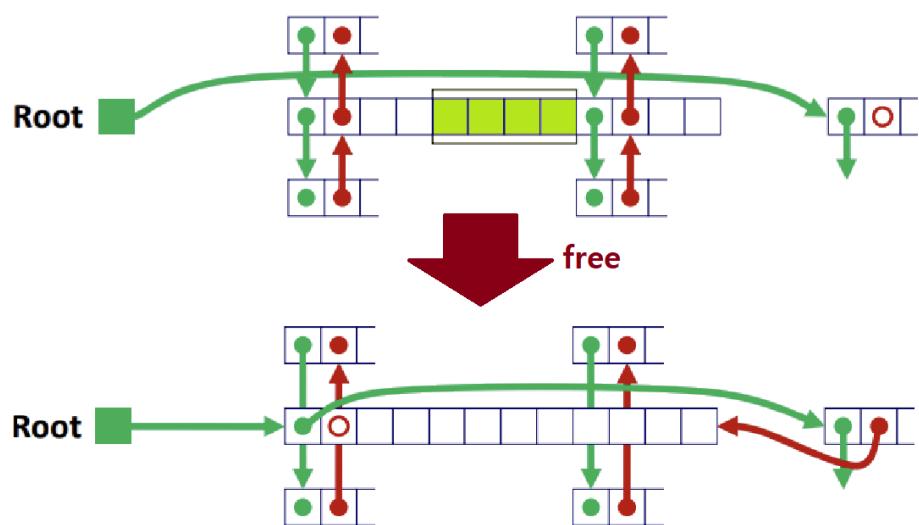
지금까지 언급한 개념들을 하나로 통합하면, 아래와 같은 그림을 생각해볼 수 있다.



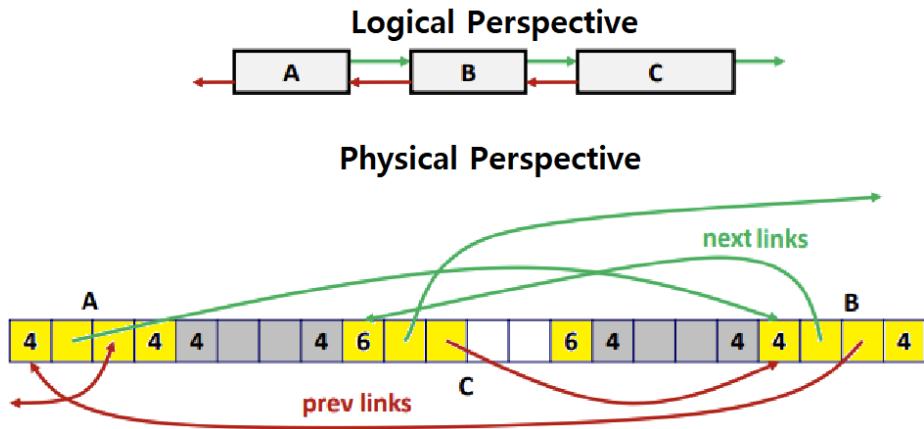
이때, 앞서 언급한 것처럼, 각 Class는 Explicit Free List로 구성한다. 우리가 특정 Allocated Block을 Free하면, Explicit Free List 사용 시 아래와 같은 흐름으로 연산을 수행하면 된다고 배웠다.



이를 Pointer 관점에서 조금 더 구체적으로 표현하면 아래와 같다.



논리적인 관점에선 위와 같이 이해할 수 있는 것이다. 하지만, 역시나 강의 시간에서 배운 것처럼, 실제 Memory 관점에선 아래와 같은 형태로 구성될 것임을 알아야 한다.



단순히 강의 시간에 배운 내용을 언급해 분량을 늘리려는 것이 아니다. 이는 본 프로젝트 구현 상에서 상당히 중요한 부분 중 하나인 다음의 사실을 깨닫는데 매우 중요한 이론적 배경이기 때문이다.

*“Heap에서의 Block 위치’와 ‘Free List에서의 Block 위치’를 구분해야 한다.”*

이는 malloc, realloc, free 구현 시에 반드시 기억해야 할 내용이다. 이제 이론적 배경은 여기까지 하고, 실제 Code-Level Detail을 확인하자.

### 3. Descriptions

#### A. Global Variables & Structs

이제 본격적인 Code-Level Detail을 분석해보자. 단순히 Code 측면에서만이 아니라, 위에서 언급한, 그리고 아직 언급하지 않은 다양한 이론적 배경 및 이유에 대해서도 철저하게 기술할 것이다. 먼저, 본 프로젝트 구현 과정에서 사용된 Global Variables에 대해 알아보자. 선언부는 다음과 같다.

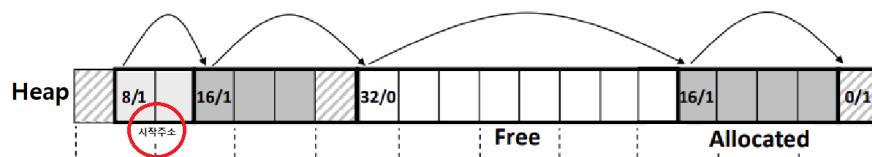
```

/* Developer Information */
team_t team = {
    "20171643",                                // student ID
    "Junhyeok Park",                            // name
    "junttang123@naver.com",                    // email
};

/* Global Scalar Variables */
static char *heap_head;                      // head of heap
static char **seglist;                        // seglist classes

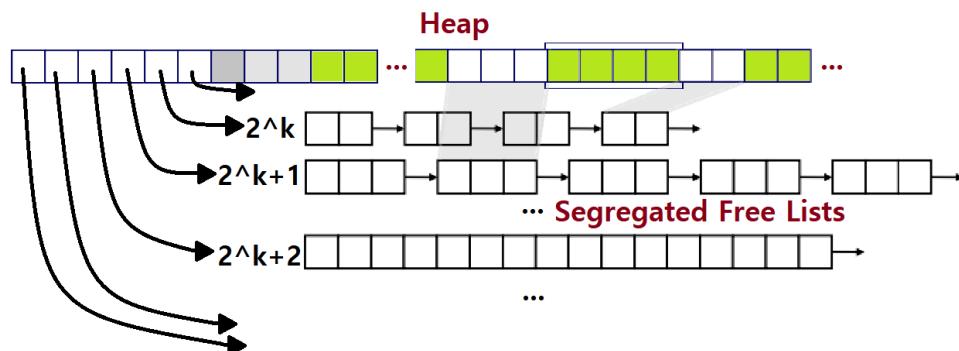
```

- **team Structure** : 본 프로젝트에서 사용하는 Identification이다.
- **heap\_head** : mm.c에서 제공하는 Heap의 시작 주소를 가리키는 포인터 변수이다. 아래 그림에서, 빨간원으로 둘러싼 위치를 가리키는 역할이다.



- **seglist** : Segregated Free List를 운영하기 위한 더블 포인터 변수이다. 최초에 Heap 공간을 초기화할 때, 맨 앞에 N(클래스 개수)개의 Word를 할당해, 이 영역을 seglist라는 더블 포인터 변수로 가리켜서 '각 Seglist의 Header로 이루어진 Array'를 마련하는 것이다.
  - 이는, Global이나 Local-Static 형태로 Data Structure를 구현하는 것을 금지하는, 본 프로젝트의 규칙을 따르기 위해 도입한 방식이다.
  - seglist 변수가 상기한 '공간'의 시작을 가리키고, 거기서 Indexing 등을 통해 각 Class를 접근할 수 있는 것이다. 이때, 각 Class의 Header는 1 Word Size이면 충분하다. 특정 메모리 공간(Heap 내 Block)을 가리키는 Pointer이면 되기 때문이다.

즉, 이를 그림으로 묘사하면 아래와 같다. mm.c에서 관리하는 메모리 공간을 사실 아래와 같은 것이다.



## B. Preprocessor Macros

자세한 Subroutine에 대한 소개에 앞서, 잠시 본 프로젝트에서 사용하는 각 종 Macro Expansion에 대해서도 알아보자. 본 프로젝트 안내 PDF에서도 명시된 것처럼, Dynamic Memory Allocator의 구현엔 필연적으로 Pointer Casting, Pointer Arithmetic이 난잡하게 사용되는데, 이를 매크로 없이 그대로 사용하면, 상당히 가독성이 떨어지는 코드가 나올 수 밖에 없다. 따라서, 본인은 아래와 같은 매크로를 두어 이를 핸들링했다.

```
/* Preprocessor Directives */
#define WORD          4           /* word, doubleword, chunk(4KB) size */
#define DWORD         8           /* to avoid unnecessary indirection,
#define CHUNK        4096        /* just using constants directly!
#define CLASSES       13          /* number of seglist headers */

#define ALIGN(size)    (((size) + (DWORD-1)) & ~0x7) /* doubleword alignment */
#define VALUE(addr)   (*(size_t *)(addr))      /* extract value from address */
#define HEADER(block) ((char *)(block) - WORD)  /* header of the block */
#define SIZE(block)   (VALUE(HEADER(block)) & ~0x7) /* size of the block (not just payload) */
#define FLAG(block)   (VALUE(HEADER(block)) & 0x1)  /* is the block allocated? */
#define FOOTER(block) ((char *)(block) + SIZE(block) - DWORD) /* footer of the block */

#define NEXTB(block)  ((char *)(block) + SIZE(block)) /* next or prev block in heap */
#define PREVB(block)  ((char *)(block) - (VALUE(((char *)(block) - DWORD) & ~0x7))

#define NEXTP(block)  (void *)VALUE(block)           /* next or prev block in seglist */
#define PREVP(block)  (void *)VALUE((char *)(block) + WORD)
#define NEXTP_(block) VALUE(block)                   /* difference between NEXTP and
#define PREVP_(block) VALUE((char *)(block) + WORD)  /* NEXTP_ is only a purpose!
                                         // NEXTP_ for avoiding warnings
#define SBRK_ERR      (void *)-1
#define CHECK_PARAM    0           /* operation code for My Consistency Checker */
```

- **WORD, DWORD, CHUNK, CLASSES** 등은 모두 Name과 같은 Constant로 확장된다. 이때, WORD에 대해서 잠시 첨언하자면, sizeof(size\_t)와 같은 형태를 취급하는 것이 사실은 좀 더 정확한 것임은 분명하다. 하지만, 본 프로그램에서 WORD Size는 이미 고정이고, 코드 내에서 WORD에 대한 접근이 상당히 많은 가운데, 만일 sizeof를 사용한 Indirect한 접근을 WORD에 대한 Expansion으로 두면, 경미하더라도 분명한 Overhead가 발생할 것이라 판단해, 위와 같이 그냥 단순히 상수로 둔 것이다. 이는 DWORD, CHUNK 역시 마찬가지이다.
- **ALIGN**은 참조 코드에서도 제공되는 간단한 Alignment 수행 매크로 함수이다. DWORD(8)를 이용해 Alignment를 수행한다. Input으로 'size +

DWORD'를 입력 받아 Alignment를 수행한다.

- **VALUE**는 간단한 Dereferencing으로, 입력 받은 주소에 있는 Value를 size\_t Type에 맞추어 반환한다. 참고로, size\_t는 Doubleword Size이다.
- **HEADER**는 입력 받은 Block의 'Payload 시작 주소 값'을 토대로 해당 Block의 Header 위치를 반환하는 매크로이다. 입력 받은 주소 값을 char Pointer 형으로 Casting 후, Pointer Arithmetic을 수행해 Byte 단위 Referencing을 하고 있음에 주목하자. 이러한 관점에서 **FOOTER**도 어렵지 않게 이해할 수 있다.
- **SIZE**와 **FLAG**는 모두 Header Word에 기록된 Value를 이용해 판단한다. 하위 3개 Bit를 제외한 Value가 Block Size가, LSB와 0x1을 AND 연산한 결과 Value가 Block Flag, 즉, Allocated or Freed 판단 결과가 나타난다. 이들은 모두 malloc, free, realloc에서 요긴하게 사용된다.
- **NEXTB**와 **PREVB**는 Heap Segment 내에서의 '입력 Block'의 'Next Block', 'Previous Block'을 의미한다. 앞서, Heap 공간에서의 Block 위치와 Seglist에서의 Block 위치를 구분해야 한다고 했는데, 이는 'Heap 공간'이 기준이다. 위의 코드를 보면 알 수 있듯, 간단한 **Byte 단위 Addressing**을 통해 해결하고 있음을 알 수 있다.
  - 이때, PREVB에서, Heap 상에서 '이전'에 해당하는 Block의 Footer에 접근 후, 해당 Word 내의 Value를 VALUE로 추출하고, 그 추출 값에서 하위 3개 Bit를 제외해 Size 정보를 얻고 있음에 주목하자.
  - 이어서 해당 Size 정보를 이용해 '이전 Block'의 Payload 시작 주소 값'을 알아내고 있다.
    - ◆ 이 PREVB 매크로 수식 하나로 본 프로젝트를 관통하는 **Block Structure**를 완벽하게 이해할 수 있다. 상당히 중요하다.
- **NEXTP**, **PREVP**는 반대로 Seglist에서의 Block 위치를 기준으로 수행된다. 앞서 소개했던 Block Structure를 기반으로 해석해볼 수 있다.
  - 이때, NEXTP, NEXTP\_와 같이, 같은 기능을 수행하는 매크로가 2개가 정의되어 있는데, 전자는 R-Value로서, 후자는 L-Value로서 사용하기 위함이다. 이를 통해 불필요한 Warnings를 제거할 수 있다.

- **SBRK\_ERROR** : 'mem\_sbrk'의 Error Return 값이다. 가독성을 위해 이 역시 매크로화하였다.
- **CHECK\_PARAM** : 추후 소개할 mm\_check 함수의 Test Type을 결정한다.

## C. Subroutines

이제 본격적으로 본인이 개발한 mm.c Dynamic Memory Allocator를 소개한다. init, malloc, free, realloc을 순서로 소개하되, 각 Interface에서 사용하는 Helper Function들을 그때 그때 소개하는 형태로 글을 진행하겠다.

### (1) mm\_init 함수

```
/* setting the initial states of the heap segment */
int mm_init(void) {
    int i;

    if ((seglist = mem_sbrk((CLASSES + 1) * WORD)) == SBRK_ERR) return -1;
    for (i = 0; i < CLASSES; i++)           // seglists are resident in the heap segment
        seglist[i] = NULL;                  // initialize each header as NULL

    if ((heap_head = mem_sbrk(4 * WORD)) == SBRK_ERR) return -1; // allocate intial area

    VALUE(heap_head) = 0x0;                // 1 word padding for alignment
    VALUE(heap_head + 1 * WORD) = (DWORD | 0x1); // virtual block for handling
    VALUE(heap_head + 2 * WORD) = (DWORD | 0x1); // edge cases of each operation
    VALUE(heap_head + 3 * WORD) = 0x1;         // end of the (virtual) heap segment
    heap_head += DWORD;                   // adjust pointing and alignment

    return 0;
}
```

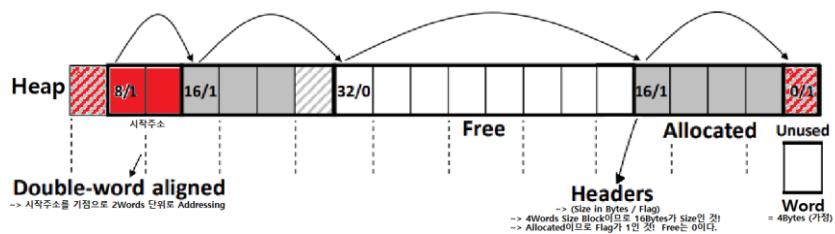
앞서 소개했던 Heap Segment 형태를 '정확히 그대로' 구현한다. 본격적인 Heap Segment 할당에 앞서 Segregated Free List 각 Class들의 Header를 마련한다. 본인은 본 프로젝트에서 Seglist Class를 총 13개 (16Bytes to 131072Bytes)로 두었기 때문에 위와 같이 '+1 Word'의 Padding을 붙이고 있음에 주목하자.

Seglist Header 영역 할당이 끝나면, 이들을 모두 NULL로 초기화해준다. 일반적으로 Linked List의 Header는 NULL로 초기화함에 입각한 것이다.

바로 이어서, 역시나 앞에서 소개했던 'Alignment를 위한 Padding', '자리만 차지하는 초기 Block', 'End of the Heap'을 총 4 Words 크기로 하여

할당한다.

- 이때, '자리만 차지하는 Block'의 경우, malloc, free, realloc 등의 과정에선 '크기가 없는 녀석'으로 취급되어야 하기 때문에 사이즈를 할당하되, DWORD 크기로만 할당함에 주목하자.
- 이 Block이 존재한다는 것을 명확히 하기 위해 Allocated or Freed Flag를 1로 Setting함에도 주목하자.



위와 같이 초기 설정이 마무리 되면, `heap_head`를 2개 Word 크기 만큼 위치를 조정해, Word Alignment를 만족시킬 수 있게 환경을 조성한다. 이렇게 하면, `mm_init` 함수가 할 일을 끝났다.

## (2) `mm_malloc` 함수

```
/* main allocating function, just like 'malloc' of libc! */
void *mm_malloc(size_t size) {
    size_t alloc_size, selected_size, surplus;
    void *ptr, *surplus_p;
    int sbrk_flag = 0;

    if (size == 0) return NULL;
    alloc_size = ALIGN(size + WORD); // align!

    if ((ptr = best_fit_search(alloc_size)) == NULL) { // find proper free block!
        if ((ptr = mm_sbrk(alloc_size)) == NULL) // if nothing found, then extend
            mm_error(1);
        sbrk_flag = 1;
    }

    selected_size = SIZE(ptr); // size of selected block
    surplus = selected_size - alloc_size; // surplus area of selected block

    if (sbrk_flag == 0) // if 'selected block' is not newly allocated,
        seglist_pop(ptr); // then, it's freed block that is in seglists!

    /* split routine */
    if (surplus >= 4*WORD) { // 4 words (header, 2 pointers, footer)
        if (alloc_size >= 20*WORD) { // to avoid 'Biased Situation' caused by
            surplus_p = ptr; // alternative allocation (binary-bal.rep),
            ptr = (char *) (ptr) + surplus; // set a threshold and if surplus pass the
        } // threshold, then make the left side of
    }
}
```

```

        else surplus_p = (char*)(ptr) + alloc_size; // selected block as surplus (will be freed)

        VALUE(HEADER(ptr)) = (alloc_size | 0x1);
        VALUE(FOOTER(ptr)) = (alloc_size | 0x1);    // ptr points area that will be allocated

        VALUE(HEADER(surplus_p)) = surplus;          // surplus_p points surplus area
        VALUE(FOOTER(surplus_p)) = surplus;

        mm_free(surplus_p);                         // call mm_free to coalesce!
    }
else {
    VALUE(HEADER(ptr)) = (selected_size | 0x1);
    VALUE(FOOTER(ptr)) = (selected_size | 0x1);
}

return ptr;
}

```

size가 0인 경우, NULL을 반환하도록 초기 예외 처리를 수행한 후, 함수는 본격적으로 일을 시작한다. 가장 먼저 입력 받은 size에 대해 Alignment Rule과 Block Structure를 고려해 '실제 할당할 Size(alloc\_size)' 가 몇인지를 알아낸다. 예를 들어 1 (1 Byte)란 값이 들어오면, 그대로 16 Bytes, 즉, 4 Word에 해당하는 크기가 잡히는 것이다. 앞서 소개한 ALIGN 매크로를 이용한다.

alloc\_size를 알아낸 후, 이를 이용해 best\_fit\_search 함수를 호출한다. 함수 이름 그대로, Size에 알맞은 Segregated Free List를 특정하고, 해당 Class를 Linear하게 순회해, 가장 Fit(입력 Size를 커버할 수 있는 Block 중 크기가 가장 작은 Block)한 Block을 찾아낸다. best\_fit\_search 함수는 아래와 같이 구현되어 있다.

```

/* procedure that find the proper block using 'Best Fit Approach' */
static void *best_fit_search(size_t alloc_size) {
    size_t cur_size, fit_size = (size_t)-1;
    void *iter, *selected = NULL;
    int i, start;

    start = what_class(alloc_size);           // for the size that caller wants,
    for (i = start; i < CLASSES; i++) {      // find the corresponding class by size
        iter = seglist[i];

        while (iter) {                      // traverse the selected class (seglist)
            cur_size = SIZE(iter);

            if (FLAG(iter) == 0)
                if (cur_size >= alloc_size && cur_size < fit_size) {
                    fit_size = cur_size;
                    selected = iter;
                } // find the minimum block that can cover the 'want_size'

            iter = NEXTP(iter);               // move to next block (in free list, not heap)
        }
    }

    return selected;
}

```

간단한 Nested Loops를 이용해 구현하고 있음을 주목하자.

이어서 다시 mm\_malloc을 보자. 만약, Best Fit에서 적절한 Block을 찾는다면, 해당 Block에 대한 포인터를 반환(ptr) 받은 후, 해당 포인터를 이용해 Selected Block의 Size를 알아낸다. 이때, 이 과정에서 만약 Best Fit이 실패해 적절한 Block을 찾지 못했다면, mm\_sbrk라는 Heap Extension 함수를 호출한다.

```
/* extending the size of heap by calling mem_sbrk, with little additioal process */
static void *mm_sbrk(size_t alloc_size) {
    size_t temp, new_size;
    void *ptr;

    temp = ((alloc_size > CHUNK) ? // default size is CHUNK (4K)
            alloc_size : WORD;      // choose bigger one (CHUNK vs input size)
    new_size = (temp % 2 == 0) ? (temp*WORD) : ((temp + 1)*WORD); // align!

    if ((ptr = mem_sbrk(new_size)) == SBRK_ERR) return NULL;

    VALUE(HEADER(ptr)) = new_size;
    VALUE(FOOTER(ptr)) = new_size;
    VALUE((char *)FOOTER(ptr) + WORD) = 0x1; // end of the (virtual) heap segment

    return ptr;
}
```

Caller로부터 Size 정보를 입력 받은 후, 해당 Size와 CHUNK(4096Bytes)를 비교해, 둘 중 더 큰 Size를 채택한다. Size 채택 후, 해당 Size가 Doubleword Alignment를 충족시킬 수 있도록 WORD로 나누고 다시 곱하는 과정을 거친다. 이 작업이 마무리되면 실제 mem\_sbrk함수를 이용해 Heap Segment를 확장한다.

- 이때, 단순히 확장 뿐만 아니라, End of the Heap을 보존하고 있음에 주목하자.

다시 mm\_malloc으로 돌아가자. 앞선 과정에서 sbrk\_flag라는 플래그 변수에 'Selected Block이 어디에서 추출된 것인지'를 기록한다. 이어, 이 플래그를 이용해, Selected Block이 Seglist에서 추출됐을 경우, 이를 실제로 seglist\_pop이란 함수를 이용해 Code-Level로 수행한다. seglist\_push 함수와 seglist\_pop 함수는 Caller로부터 넘겨 받은 Block Size에 대응하는 Seglist Class를 특정한 후, 해당 Class에서 넘겨 받은 Block을 찾아 삽입/삭제를 수행한다. 이 두 Helper Functions의 코드는 아래와 같다.

```

/* insert newly freed block into the corresponding seglist */
static void seglist_push(void *target) {
    size_t target_size = SIZE(target);
    void *former;

    former = seglist[what_class(target_size)];

    NEXTP_(target) = (size_t)former;           // simply insert into the header of list
    PREVP_(target) = (size_t)NULL;

    if (former != NULL) PREVP_(former) = (size_t)target;
    seglist[what_class(target_size)] = target;
}

/* delete the target free block from the corresponding seglist */
static void seglist_pop(void *target) {
    size_t target_size = SIZE(target);
    void *next, *prev;

    next = NEXTP(target);                  // simple 'doubly linked list' node deletion
    prev = PREVP(target);

    if (prev != NULL) NEXTP_(prev) = (size_t)next; // casting for avoiding warnings
    else seglist[what_class(target_size)] = next;

    if (next != NULL) PREVP_(next) = (size_t)prev;
}

```

간단한 논리의 Doubly Linked List 연산임을 어렵지 않게 알아챌 수 있다. 이때, Casting 연산은 Warning을 제거하기 위한 행위로, 프로그램 자체의 컴파일 가능 여부와는 관련이 없다.

- Seglist Insertion 시, List의 Head 방향 가장 맨 앞에 '새 Block'을 삽입하고 있음을 주목하자. Best Fit으로 'Proper Block'을 찾기 때문에, 굳이 삽입 시에는 특별한 루틴을 설치하지 않은 것이다. 가장 빠르고 효율적으로 삽입을 하기 위해서이다.

```

/* return the index of appropriate class, based on 'size' */
static int what_class(size_t size) {
    if (size > 131072) return 12;
    if (size > 32768) return 11;
    if (size > 16384) return 10;
    if (size > 8192) return 9;
    if (size > 4096) return 8;           // 2^k
    if (size > 2048) return 7;
    if (size > 1024) return 6;
    if (size > 512) return 5;
    if (size > 256) return 4;
    if (size > 128) return 3;
    if (size > 64) return 2;
    if (size > 32) return 1;
    return 0;                         // total 13 classes
}

```

참고로, what\_class 함수는 이름 그대로 입력 받은 Size를 이용해 Corresponding Class Index를 반환하는 간단한 함수로, 좌측과 같이 구성되어 있다.

다시 mm\_malloc으로 돌아가자. sbrk\_flag를 이용해 Selected Block이 Seglist에서 왔는지, mem\_sbrk에서 왔는지를 확인 후, 이를 토대로 Seglist Deletion 연산을 수행할지 말지를 결정한다.

이어서, Split Routine이 구현되어 있다. Selected Block의 크기에서 실제 할당하고자 하는 크기를 뺀을 때, 그 결과가 4 Words 이상일 경우, 여유분(Surplus)을 따로 Free Block으로 만들고, 실제로 mm\_free 함수를 호출해 Freed화시킨다. 만약, 여유분 사이즈가 작을 경우, 그냥 여유분까지 포함해서 Selected Block 전체를 반환 Block으로 설정한다.

- 4 Words인 이유는 무엇일까? 그렇다. Freed Block의 경우, Header, Footer와 2개의 Pointer가 필요하기 때문이다.
- 또한, 코드를 보면, alloc\_size가 20 Words 이상의 큰 사이즈일 경우 Surplus를 우측이 아니라 좌측에 두어 Split을 진행하고 있음을 알 수 있다.
  - ◆ 이러한 처리는 왜 있는 것일까? 우리가 Surplus를 계속 우측에 다만 두고 Split을 진행하면, Tracefiles의 binary-bal.rep 파일 입력과 같은, 크기 합이  $2^k$ 이지만, 이 둘을 분리해 같은 크기로 교대로 할당이 반복적으로 이뤄지는 경우, Fragmentation이 매우 커지는 Biased Situation을 맞이하게 된다.
  - ◆ 두 교대 할당의 Payload 합은  $2^k$ 이지만, 실제 Allocated Block Size는  $2^k$ 에 Overhead가 포함되는 크기이기 때문에, Split 시에 계속 빈 공간을 아깝게 만들어내는 형태가 되는 것이다.
  - 이는 위와 같이, Word의 Multiple인 특정 Threshold를 설정 후, 해당 Threshold를 넘어가면 Split을 하되, Surplus를 좌측에 두게 함으로써, 교대 입력 상황에서 Surplus가 서로 마주보도록 유도해 Fragmentation을 낮출 수 있다.
- 이는 본인이 프로젝트를 수행하면서 발견한 가장 주요한 Skill 중 하나이다.

아무튼, 위와 같은 Split Routine까지 마무리 되면, Newly Allocate한 Block의 주소를 반환함으로써 mm\_malloc은 수행을 마무리 한다.

### (3) mm\_free 함수

```

/* main freeing function, just like 'free' of libc! */
void mm_free(void *ptr) {
    size_t size, prev_size, next_size;
    void *prev_block, *next_block;
    int prev_alloc, next_alloc, option;

    if (ptr == NULL) return;

    size = SIZE(ptr);
    VALUE(HEADER(ptr)) = size;
    VALUE(FOOTER(ptr)) = size;

    prev_block = PREVB(ptr);           // with previous and next block,
    prev_alloc = FLAG(prev_block);     // determine whether to coalesce or not
    prev_size = SIZE(prev_block);

    next_block = NEXTB(ptr);
    next_alloc = FLAG(next_block);
    next_size = SIZE(next_block);

    if (next_alloc == 1 && prev_alloc == 1) option = 1;      // NOT coalesce
    if (next_alloc == 0 && prev_alloc == 1) option = 2;      // coalesce with next block
    if (next_alloc == 1 && prev_alloc == 0) option = 3;      // coalesce with prev block
    if (next_alloc == 0 && prev_alloc == 0) option = 4;      // coalesce with both blocks

    ptr = coalesce(option, size, ptr, next_size, next_block, prev_size, prev_block);
    seglist_push(ptr);                  // insert (possibly) coalesced block into seglist
}

```

Free Routine은 Coalescing만 고려하면 된다. 간단하다. 입력 받은 포인터가 가리키는 Block의 'Heap 내' Previous Block과 Next Block을 알아낸다. 이들이 Freed인지를 FLAG Macro를 이용해 확인 후, Freed인 Block들을 특정해 합체시키면 된다. 합체 후, Seglist에 Push하는 것을 주목하자.

```

/* procedure that coalesce input block with prev or next block (in heap) */
static void *coalesce(
    int option,                      // what case?
    size_t size, void *p,             // current block
    size_t nszie, void *np,          // next block of curr
    size_t psize, void *pp           // prev block of curr
) {
    size_t new_size;
    void *front_ptr, *rear_ptr;

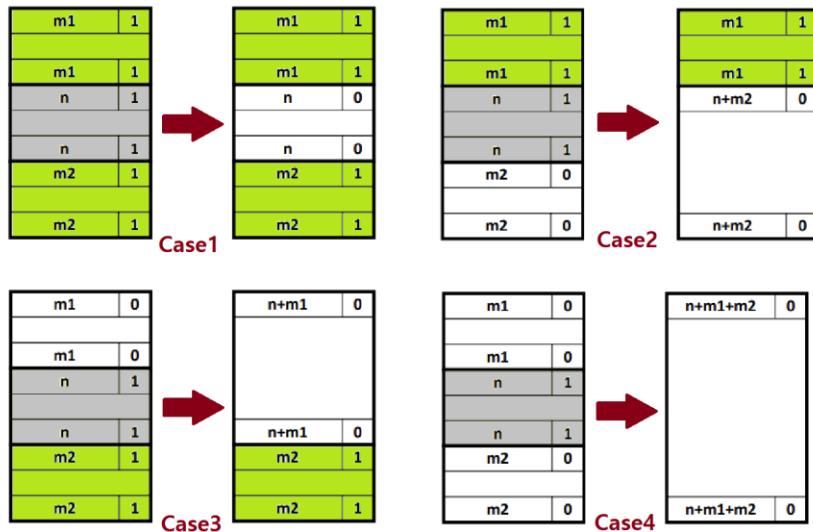
    if (option == 1) return p;        // case1
    switch (option) {
        case 2: new_size = size + nszie; // case2
            front_ptr = p; rear_ptr = np;
            seglist_pop(np);
            break;
        case 3: new_size = psize + size; // case3
            front_ptr = pp; rear_ptr = p;
            seglist_pop(pp);
            break;
        case 4: new_size = psize + size + nszie; // case4
            front_ptr = pp; rear_ptr = np;
            seglist_pop(np);
            seglist_pop(pp);
            break;
        default: break;
    }

    VALUE(HEADER(front_ptr)) = (new_size & ~0x7); // set boundary tags
    VALUE(FOOTER(rear_ptr)) = (new_size & ~0x7); // as 'freed' with size info

    return front_ptr;
}

```

Immediate Coalescing을 수행하는 coalesce 함수 역시 간단하다. mm\_free 함수에서 Flag들을 통해 판단한 Coalescing Case를 토대로, 간단히 합치면 된다. 합체하기 전, 기-Freed Block의 경우, 먼저 Seglist에서 Delete하고 나서, 합체 후 다시 Insert하고 있음에 주목하자. Segregated Free List의 Consistency를 유지하기 위해 반드시 수행해야 하는 루틴이다.



#### (4) mm\_realloc 함수

```
/* re-allocating function, just like 'realloc' of libc! */
void *mm_realloc(void *ptr, size_t size) {
    size_t cur_size, want_size, next_size, prev_size, new_size, temp1, temp2;
    void *next_block, *prev_block, *new_block, *NorP_block;
    int next_alloc, prev_alloc, NorP_alloc;

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

    cur_size = SIZE(ptr); // size of input block
    want_size = ALIGN(size + DWORD); // size that caller wants
    if (want_size <= cur_size) return ptr; // if want is smaller, then just return

    /* select the bigger one between previous block and next block */
    next_block = NEXTB(ptr); //***** prev_block = PREVB(ptr);
    next_size = SIZE(next_block); //***** prev_size = SIZE(prev_block);
    temp1 = next_size + cur_size; //***** temp2 = prev_size + cur_size;

    new_size = (temp1 > temp2) ? temp1 : temp2;
    new_block = (temp1 > temp2) ? ptr : prev_block; // selection
    NorP_block = (temp1 > temp2) ? next_block : prev_block;
    NorP_alloc = (temp1 > temp2) ? next_alloc : prev_alloc;

    if (size < cur_size) cur_size = size; // adjust cur_size
```

```

if (NorP_alloc == 0 && (new_size - want_size) >= DWORD) {    // if (prev/next+curr) can cover
    seglist_pop(NorP_block);                                // the want_size, then coalesce,
    memmove(new_block, ptr, cur_size);

    VALUE(HEADER(new_block)) = (new_size | 0x1);           // adjust the block,
    VALUE(FOOTER(new_block)) = (new_size | 0x1);           // and DO NOT split !!!
    return new_block;                                     // return the coalesced block
}

/* if 'selected block with prev/next' cannot cover 'want', then just new alloc! */
if ((new_block = mm_malloc(size)) == NULL) mm_error(1);
memcpy(new_block, ptr, cur_size);
mm_free(ptr);

return new_block;
}

```

mm.c에서 가장 Tricky한 mm\_realloc 함수이다. 우선, NULL Pointer 입력과 0 Size 입력과 같은 예외 상황을 간단히 처리하며 함수는 본격적인 일을 시작한다. 입력 받은 ptr Pointer가 가리키는 Allocated Block의 Size를 SIZE Macro를 이용해 추출해 cur\_size에 저장한다. 입력 받은 size 값을 토대로 Caller가 원하는 Size를 위해 필요한 최소한의 크기도 ALIGN을 통해 알아내 want\_size에 저장한다.

이때, want\_size가 cur\_size보다 작으면, 그대로 ptr가 가리키는 Block 그대로 반환한다. 왜냐? 굳이 처리할 필요가 없기 때문이다. 실제 libc의 realloc도 기존 ptr 블록 크기보다 작은 Size 입력 시 기존 공간에 대한 보호가 특별히 이뤄지지 않는데, 아마 이러한 처리가 실제로도 이루어질 것임을 예측할 수 있다.

한편, 아무런 예외에도 해당하지 않고, want\_size가 cur\_size보다 큰 상황이라 하자. 이때는, 'Heap 내'에서 ptr Block의 앞, 뒤 Block이 Freed인지를 확인한다. 만일, 이들 중 하나가 Freed이면, 해당 Block과 ptr Block을 Coalescing 하면 want\_size를 커버할 수도 있기 때문이다. 코드 중반부의 처리는 이러한 처리가 반영된 것이다. temp1에는 cur\_size와 Next Block Size를, temp2에는 cur\_size와 Previous Block Size를 더해 저장한다.

- 이때, 통념과 다르게, temp1과 temp2 중 더 큰 Case에 해당하는 Block이 선택되고 있다(코드를 보면 알 수 있다). 또한, 더 큰 Block을 선택해놓고, Split하지도 않는다. 이전의 malloc Routine과는 상이한 처리가 이뤄지고 있는 것이다.

◆ 왜 이렇게 처리한 것일까?

- **External Fragmentation**을 낮춰 궁극적으로 더 많은 공간을 **Programmer**가 **Utilize**할 수 있기 때문이다. 즉, 쉽게 말해 '작은 것을 희생해 큰 것을 취하는' 기법인 것이다. 이를 통해 Space Utilization을 높일 수 있다. 특히나 realloc-bal.rep 케이스에 대해서 그렇다. 더 큰 Block을 선택하고, Split을 하지 않음으로써, realloc의 대상이 되는 Block이 큰 공간을 활용하게 하고, **자잘한 크기의 Freed Block의 양산**을 막아 **External Fragmentation**을 높이고자 하는 것이다.
- 물론, 이는 **realloc-bar.rep**의 패턴과 **mdriver**의 평가 방법에 **종속적인 Skill**이다. 따라서 언제든지 반박 당할 수 있는 Skill이라고 자평한다. 그러나, 본 프로젝트 상황에서는 유효한 처리이다.
  - 이 외에의 여러 Optimization 시도를 해보았지만, 본 프로젝트 mdriver 평가 상황에서는 유의미한 차이를 만들지 못했다. 따라서, 이 이상의 최적화는 그만하였다. **총 97점의 Score**를 받았는데, 그럼에도 아쉽다. 분명 더 좋은 최적화 방법이 존재할 것 같은데, 본인은 아쉽게도 더 이상 시간 내에선 찾지 못하였다. 이는 학기 종료 후 자가 학습을 통해 좀 더 알아보도록 하겠다.

한편, 위의 Case에도 해당하지 않는다면, mm\_malloc을 mm\_realloc 내에서 호출해 별도의 Heap 공간에 아예 새롭게 할당하여 Re-Allocation을 수행한다.

이 밖에 mm\_error라는 Error Reporting Function도 존재한다. 다만, 이는 mm\_check 구현을 위한 간단한 프린팅 함수에 불과하기 때문에 코드는 생략한다. 첨부 코드에서 확인할 수 있다.

이제, 마지막 Heap Consistency Checker에 대해 소개한다.

## 4. Heap Consistency Checker

본인이 작성한 Heap Consistency Checker mm\_check 함수는 CHECK\_PARAM이라는 Macro 변수를 통해 동작한다. 이 값을 통해 '어떤 Test를 진행할지'를 결정한다. Test는 총 7개가 존재한다. 각 Test별로 mm\_check가 어떤 역할을 수행하는지 소개한다.

### (1) Test1: Free List의 모든 Block이 Freed Flag인지 확인

```
switch (CHECK_PARAM) { // programmer select the type of test by changing CHECK_PARAM
case 1: /* Test1: Is every block in the free list marked as free? */
    for (i = 0; i < CLASSES; i++)
        if (seglist[i] != NULL) {           // iterate all non-empty classes
            iter = seglist[i];

            while (iter) {
                if (FLAG(iter) == 1)      // simply check the flag of each block
                    mm_error(2);          // if it's 'allocated', then error!
                iter = NEXTP(iter);
            }
        }
    break;
```

모든 Non-Empty Seglist Class를 일일이 순회하여, 소속 Block 중 Flag 가 1, 즉, Allocated Flag인 Block이 있는지를 확인한다. 지금까지의 설명을 토대로 간단히 이해할 수 있다.

### (2) Test2: Heap 내에 '연속된 Free Blocks'가 있는지를 확인

```
case 2: /* Test2: Are there any contiguous free blocks that somehow escaped coalescing? */
    for (iter = heap_head; SIZE(iter) > 0; iter = NEXTB(iter)) {
        //printf("%d(%d) ", (VALUE(HEADER(iter))), (VALUE(HEADER(iter))) & 0x1);
        if (first_flag == 1) {           // only for the first step of iteration
            if (FLAG(iter) == 0)
                prev_f_flag = 1;          // intermediate state for error situation
            else prev_f_flag = 0;

            first_flag = 0;
        }
        else {
            if (FLAG(iter) == 0) {
                if (prev_f_flag == 0)
                    prev_f_flag = 1;
                else mm_error(3);          // when two consecutive blocks are freed!
            }
            else prev_f_flag = 0;
        }
    }
break;
```

첫 순회를 first\_flag를 통해 식별 후, Heap의 첫 번째 '유효 Block'이 Allocated인지, Freed인지를 확인한다. 이후, 이어지는 모든 Block에 대해 Linear Traversal을 이어가면서, Intermediate State를 두어 '연속 Free Block 여부'를 확인한다. 코드가 아주 간단한 FSM, 유한 상태 기계로 동작하는 것이다. 역시나 코드 자체는 아주 간단히 이해할 수 있다.

### (3) Test3: Heap의 모든 Free Block이 실제로 Free List에 포함되는지 확인

```
case 3: /* Test3: Is every free block actually in the free list? */
    for (iter = heap_head; SIZE(iter) > 0; iter = NEXTB(iter)) {
        if (FLAG(iter) == 0) { // for every free blocks in heap,
            index = what_class(SIZE(iter)); // specify the corresponding class,
            //printf("[%d] %d(%d) : ", index, (VALUE(HEADER(iter))), (VALUE(HEADER(iter))) & 0x1);
            jter = seglist[index];
            found = 0;

            while (jter) { // and then traverse that class,
                if (iter == jter)
                    found++;
                // count whenever u find the same address of block
                //printf("%d(%d), ", (VALUE(HEADER(jter))), (VALUE(HEADER(jter))) & 0x1);
                jter = NEXTP(jter);
            }

            if (found != 1) mm_error(4); // count must be NOT ONLY nonzero, but also must be 1
        }
    }
    break;
```

본 Test 중 가장 시간이 오래 걸리는 Test이다. 중첩 꼴로 Naive하게 판단하기 때문이다.

Heap 내의 모든 Block을 반복문을 통해 순회한다. 이때, Free Block에 대해, 해당 Block의 Size 정보를 읽은 후, 이를 토대로 대응하는 Seglist Class를 특정해, 해당 List를 순회한다. 이때, found라는 Count 변수가 도입되는데, 코드를 보면 알 수 있듯, 이 변수는 단순히 'Free Block이 List에 있음'만을 알리는 것이 아니라, '단 하나만 있음'도 확인하기 위해 존재한다.

- Free Block은 반드시 단 하나의 Free List에 단 하나의 Instance 형태로 존재해야 한다. 만일, 서로 다른 위치에 같은 Freed Block이 기억될 경우, Data Inconsistency가 발생할 수 있기 때문이다. 이는 나아가 Heap Inconsistency로 심화될 것이다.

#### (4) Test4: Free List의 모든 Pointer들은 유효한 Free Block을 가리키는가?

```
case 4: /* Test4: Do the pointers in the free list point to valid free blocks? */
for (i = 0; i < CLASSES; i++) {
    if (seglist[i] != NULL) {
        iter = seglist[i];

        while (iter) {           // simply check the connection state!
            if ((NEXTP(iter) != NULL) && (iter != PREVP(NEXTP(iter))))
                mm_error(5);
            if ((PREVP(iter) != NULL) && (iter != NEXTP(PREVp(iter))))
                mm_error(5);

            iter = NEXTP(iter);
        }
    }
    break;
}
```

단순한 Seglist Traversal을 이용하고 있다. 각 순회에서, 'Free List 내' 이전 Block의 Next Pointer가 '현재 조회 Block'인지, 반대로, 'Free List 내' 이후 Block의 Previous Pointer가 '현재 조회 Block'인지를 체크한다. 이것이 곧 유효 Pointing 관계 체크이기 때문이다.

#### (5) Test5: Heap 내의 Allocated Block들이 서로 Overlap되는 경우가 있는가?

```
case 5: /* Test5: Do any allocated blocks overlap? */
for (iter = NEXTB(heap_head); SIZE(iter) > 0; iter = NEXTB(iter)) {
    if (FLAG(iter) == 1) {           // for every allocated blocks,
        next = NEXTB(iter);

        if (SIZE(next) > 0 && FLAG(next) == 1) {      // if next is also allocated,
            size = SIZE(iter);

            jter = HEADER(iter);
            byte_cnt = 0;
            while (jter != HEADER(next)) {           // then check the overlap!
                jter = (char *)jter + 1;             // go forward by 'BYTE'
                byte_cnt++;                         // count bytes number
            }

            if (byte_cnt != size)           // check if bytes number matches cur_size
                mm_error(6);
        }
    }
}
break;
```

Heap 내의 각 Block을 Linear하게 순회하되, 매 Block에 대해 Nested Loop를 통해 Byte Counting을 수행한다. Counting은 메모리 주소 값을

기준으로 횟수가 정해지기 때문에 정확하다. Counting이 완료되면, 총 카운트된 횟수와 Size 값을 비교해, Overlap 여부를 확인한다. Byte Counting을 위해 Byte 단위 Addressing이 Inner Loop에서 수행되고 있음을 주목하자.

(6) Test6: Heap 내에서 Block들이 유효한 곳을 Pointing하는가?

```
case 6: /* Test6: Do the pointers in a heap block point to valid heap addresses? */
    for (iter = NEXTB(heap_head); (VALUE(HEADER(iter)) & ~0x7) > 0; iter = NEXTB(iter)) {
        if ((NEXTB(iter) != NULL) && (iter != PREVB(NEXTB(iter)))) {
            mm_error(7);
        }
        if ((PREVB(iter) != NULL) && (iter != NEXTB(PREVb(iter)))) {
            mm_error(7);
        }
        // simply check the connection state!
    }
    break;
```

아주 간단한 Check이기 때문에 Code로 대체한다.

(7) Test7: 모든 Block의 주소 값이 Doubleword Aligned인지 확인하고, 해당 시점에서의 Heap 구성과 Segregated Free Lists 구성을 모두 출력한다.

```
case 7: /* Test7(self): Print the heap word by word, and the whole free lists!
           with checking whether each blocks in the heap is aligned by doublewords. */
    iter = NEXTB(heap_head);

    while (SIZE(iter) > 0) {      // for every blocks in heap (regardless of flag)
        if ((size_t)iter % DWORD != 0)          // check the alignment
            mm_error(8);

        size = SIZE(iter) / WORD;
        total_size += (size * 4);
        byte_cnt = 0;

        printf("| ");
        for (jter = HEADER(iter), i = 0; i < size; // iterate the inside of block
             jter = (char *)jter + WORD, i++) {
            if (FLAG(iter) == 1) {                  // if block is allocated,
                if (i == 0) printf("ah(4) ");
                else if (i == size - 1) printf("p(%d) af(4) ", byte_cnt * 4);
                else byte_cnt++;
            }
            else {
                if (i == 0) printf("fh(4) ");
                else if (i == 1 || i == 2) printf("fp(4) ");
                else if (i == size - 1) printf("p(%d) ff(4) ", byte_cnt * 4);
                else byte_cnt++;
            }
        }
        iter = NEXTB(iter);
    }
    printf("\nTotal: %d Bytes\n", total_size);      // total bytes of heap
```

```

for (i = 0; i < CLASSES; i++) {
    if (seglist[i] != NULL) {
        iter = seglist[i];           // print all the non-empty seglist classes
        printf("Class %d: ", i);

        while (iter) {
            printf("%d ", SIZE(iter));

            iter = NEXTP(iter);
        }
        printf("\n");
    }
    printf("\n");
    break;
}

```

코드 자체는 위의 Test들과 거의 동일한 코드를 토대로 작성되었기 때문에 별 다른 설명이 없어도 이해할 수 있다. 각 Heap Block 순회에서, iterator에 대해 DWORD Division을 수행해 '실제 Virtual Memory Address'의 Alignment 여부를 일일이 체크하고 있음에 주목하자. 이 Test7의 실제 수행 결과는 아래와 같다. short1-bal.rep에 대한 결과 일부이다.

```

| ah(4) p(2040) af(4) | ah(4) p(2040) af(4) |
Total: 4096 Bytes

| ah(4) p(48) af(4) | fh(4) fp(4) fp(4) p(1976) ff(4) | ah(4) p(2040) af(4) |
Total: 4096 Bytes
Class 6: 1992

| ah(4) p(48) af(4) | fh(4) fp(4) fp(4) p(1976) ff(4) | ah(4) p(2040) af(4) | fh(4) fp(4) fp(4) p(0) ff(4) | ah(4) p(4072) af(4) |
Total: 8192 Bytes
Class 0: 16
Class 6: 1992

| ah(4) p(48) af(4) | fh(4) fp(4) fp(4) p(1976) ff(4) | ah(4) p(2040) af(4) | fh(4) fp(4) fp(4) p(0) ff(4) | ah(4) p(4072) af(4) |
Total: 8192 Bytes
Class 0: 16
Class 6: 1992

| fh(4) fp(4) fp(4) p(16) ff(4) | ah(4) p(4072) af(4) | ah(4) p(4072) af(4) |
Total: 8192 Bytes
Class 0: 32

Perf index = 59 (util) + 6 (thru) = 65/100

```

위와 같이 Heap 내부와 Class 상태를 출력한다. 이러한 mm\_check 함수에 대해, 각 mm\_malloc, mm\_free, mm\_realloc 함수 끝에 호출 명령을 설치하면 위와 같은 각 종 Test 및 디버깅을 수행할 수 있다.

이렇게, 2022년 봄학기 System Programming Project 3를 마무리한다. 감사합니다.