Malloc Lab

20230024 문요준

[define]

<code>

```
#define WORD_SIZE 4                                                                 // Word의 크기(4바이트)를 정의. 헤더 및 풋터의 최소 크기로 사용
#define DOUBLE_WORD_SIZE 8                                                          // Double Word의 크기(8바이트)를 정의. 블록 크기 정렬과 최소 블록 크기로 사용
#define CHUNK_SIZE (1<<12)                                                          // 힙 확장을 위한 기본 크기(4KB). 메모리 할당 요청 시 힙을 이만큼 늘림
#define MAX_VALUE(x,y) ((x) > (y) ? (x) : (y))                                      // 두 값 x, y 중 더 큰 값을 반환. 메모리 관리 시 크기 비교에 사용
#define PACK_BLOCK(size,alloc) ((size) | (alloc))                                   // 블록 크기와 할당 상태를 하나의 값으로 결합
#define GET_WORD(p) (*(unsigned int*)(p))                                           // 포인터 p가 가리키는 메모리 주소에서 4바이트 값을 읽어옴
#define PUT_WORD(p,val) (*(unsigned int*)(p) = (val))                               // 포인터 p가 가리키는 메모리 주소에 4바이트 값을 씀
#define GET_BLOCK_SIZE(p) (GET_WORD(p) & ~0x7)                                      // 포인터 p가 가리키는 헤더 또는 풋터에서 블록 크기를 읽음 / 하위 3비트(~0x7)를 제거하여 순수한 블록 크기만 가져옴
#define GET_ALLOC_STATUS(p) (GET_WORD(p) & 0x1)                                     // 포인터 p가 가리키는 헤더 또는 풋터에서 할당 상태(0 또는 1)를 읽음
#define HEADER_PTR(bp) ((char*)(bp) - WORD_SIZE)                                    // 블록 포인터(bp)에서 헤더의 시작 주소를 계산
#define FOOTER_PTR(bp) ((char*)(bp) + GET_BLOCK_SIZE(HEADER_PTR(bp)) - DOUBLE_WORD_SIZE)  // 블록 포인터(bp)와 헤더 크기, 블록 크기를 이용하여 풋터의 시작 주소를 계산
#define NEXT_BLOCK_PTR(bp) ((char*)(bp) + GET_BLOCK_SIZE(((char*)(bp) - WORD_SIZE)))      // 블록 포인터(bp)에서 현재 블록의 크기를 더해 다음 블록의 시작 주소를 계산
#define PREVIOUS_BLOCK_PTR(bp) ((char*)(bp) - GET_BLOCK_SIZE(((char*)(bp) - DOUBLE_WORD_SIZE))) // 블록 포인터(bp)에서 이전 블록의 크기를 빼서 이전 블록의 시작 주소를 계산
```

1. WORD_SIZE: 시스템에서 가장 작은 메모리 단위인 워드의 크기를 나타내며, 4바이트로 정의된 값이다. 블록의 Header와 Footer의 최소 크기를 결정하며, 메모리 관리 중 데이터의 구조를 유지하는 데 사용된다.

2. DOUBLE_WORD_SIZE: 더블 워드의 크기를 나타내며, 8바이트로 정의된 값이다. 블록 크기와 정렬을 위한 기준 값으로 사용되며, 블록의 최소 크기를 나타내는 역할을 수행하고, 메모리를 할당할 때 블록은 항상 DOUBLE_WORD_SIZE의 배수로 정렬된다.

3. CHUNK_SIZE: heap을 확장할 때 사용하는 기본 크기를 나타내며, 4KB(1 << 12)로 설정된 값이다.

4. MAX_VALUE(x, y): 두 값 x와 y를 비교하여 더 큰 값을 반환한다.

5. PACK_BLOCK(size, alloc): 블록의 크기(size)와 할당 상태(alloc)를 하나의 정수 값으로 결합한다.

6. GET_WORD(p): 포인터 p가 가리키는 메모리 주소에서 4바이트(1워드) 값을 읽어온다.

7. PUT_WORD(p, val): 포인터 p가 가리키는 메모리 주소에 4바이트(1워드) 값을 저장한다.

8. GET_BLOCK_SIZE(p): Header나 Footer에 저장된 블록 크기를 읽어온다.

9. GET_ALLOC_STATUS(p): Header나 Footer에 저장된 블록의 할당 상태를 읽어온다.

10. HEADER_PTR(bp): bp를 기반으로 해당 블록의 헤더(Header) 시작 주소를 계산한다.

11. FOOTER_PTR(bp): bp를 기반으로 해당 블록의 풋터(Footer) 시작 주소를 계산한다.

12. NEXT_BLOCK_PTR(bp): bp를 기반으로 다음 블록의 시작 주소를 계산하는 매크로이다.

13. PREVIOUS_BLOCK_PTR(bp): bp를 기반으로 이전 블록의 시작 주소를 계산한다.

<code>

```
// Pointer
static char *heap_p; // heap의 시작주소 저장
static char *next_p; // Next Fit 탐색에서 블록 탐색을 시작할 위치를 저장

// define functions
static void *heap_extender(size_t size);
static void *coalescer(void* bp);
static void *fit_finder(size_t size);
static void placer(void *bp, size_t asize);
```

heap_p: 힙의 시작 주소를 저장하며 메모리 관리의 기준이 되는 포인터

next_p: Next Fit 탐색에서 블록 탐색을 시작할 위치를 저장하는 포인터

heap_extender(size_t size): 요청된 크기만큼 힙을 확장 및 새로 확보한 메모리를 초기화하는 함수

coalescer(void *bp): 인접한 free 블록을 병합하여 메모리 단편화를 줄이는 함수

fit_finder(size_t size): 요청 크기에 적합한 free 블록을 Next Fit 방식으로 찾아 반환하는 함수

placer(void *bp, size_t asize): 요청된 크기만큼 블록을 할당하고, 필요하면 블록을 분할하는 함수

[mm_init()]

<code>

```
int mm_init(void)
{  %
    // 힙을 초기화하고 초기 메모리를 요청한다.
    void *initial_heap = mem_sbrk(4 * WORD_SIZE);
    if (initial_heap == (void *)-1)
    {
        // 메모리 요청 실패 시 -1 반환
        return -1;
    }

    heap_p = initial_heap; // 힙의 시작 주소 설정

    // 정렬 패딩(Alignment Padding)
    // 첫 번째 워드는 정렬을 위해 비워둔다.
    PUT_WORD(heap_p, 0);

    // Prologue Header: 더블 워드 크기(8바이트) 블록, 할당 상태로 설정
    PUT_WORD(heap_p + (1 * WORD_SIZE), PACK_BLOCK(DOUBLE_WORD_SIZE, 1));

    // Prologue Footer: Header와 동일한 설정
    PUT_WORD(heap_p + (2 * WORD_SIZE), PACK_BLOCK(DOUBLE_WORD_SIZE, 1));

    // Epilogue Header: 크기 0, 할당 상태로 설정
    PUT_WORD(heap_p + (3 * WORD_SIZE), PACK_BLOCK(0, 1));

    // 힙 포인터를 프롤로그 블록의 끝(헤더와 풋터 사이)으로 이동
    heap_p += (2 * WORD_SIZE);

    // next_p는 탐색을 시작할 위치를 설정하기 위해 초기화
    next_p = heap_p;

    // 초기 힙 크기를 확장하여 사용할 수 있는 메모리 공간을 추가
    if (heap_extender(CHUNK_SIZE / WORD_SIZE) == NULL)
    {
        // 힙 확장이 실패하면 -1 반환
        return -1;
    }

    // 초기화 성공 시 0 반환
    return 0;
}
```

mm_init 함수는 동적 메모리 관리 시스템의 초기화를 수행하는 함수다. 힙을 초기화하고, 메모리 할당 및 관리 작업에 필요한 기본 구조를 설정하며, 초기 힙 공간을 확보한다. 먼저 mem_sbrk를 호출하여 힙의 크기를 4 * WORD_SIZE만큼 늘리고, 초기 메모리를 요청한다. 요청된 메모리의 시작 주소는 initial_heap에 저장된다. 만약 메모리 요청이 실패하여 mem_sbrk가 (void *)-1을 반환하면, 함수는 -1을 반환한다. 그리고 요청된 메모리의 시작 주소(initial_heap)를 heap_p에 저장한다. 이후 순차적으로 heap_p에 대해 작업한다. 첫 번째로, 메모리 정렬을 보장하기 위해 첫 번째 워드(4바이트)는 0으로 설정한다. 프롤로그 블록은 더블 워드 크기(8바이트)의 고정된 블록으로, 힙의 시작 부분에 위치한다. 이 블록은 메모리 할당 및 병합 작업에서 경계를 관리하기 위한 목적으로 사용된다. 헤더는 PACK_BLOCK(DOUBLE_WORD_SIZE, 1)를 사용해 크기를 8바이트, 할당 상태를 1(사용 중)으로 설정하고 풋터는 헤더와 동일한 값으로 설정한다. 에필로그 블록은 크기가 0이며, 항상 힙의 끝 부분에 위치한다. 이 블록은 할당 상태가 1(사용 중)으로 설정되어 있어, 힙의 끝임을 나타낸다. 그리고 heap_p를 프롤로그 블록의 끝, 즉 헤더와 풋터 사이로 이동시킨다. 이를 통해 실제 메모리 관리 작업에서 사용할 기준점을 설정할 수 있다. 다음으로 Next Fit 방식에서 사용할 탐색 시작 포인터 next_p를 초기화한다. 초기값은 heap_p로 설정되어, 힙 탐색 시 프롤로그 블록 바로 다음 블록부터 시작한다. 마지막으로 초기화 과정에서 충분한 메모리 공간을 확보하기 위해 heap_extender 함수를 호출한다. 이 함수는 힙의 크기를 CHUNK_SIZE / WORD_SIZE만큼 확장하고, 추가된 메모리를 free 상태로 초기화한다. 만약 힙 확장이 실패하면, 함수는 -1을 반환한다.

[mm_malloc()]

<code>

```
void *mm_malloc(size_t size)
{
    // 예외 처리: 요청된 크기가 0이면 NULL 반환
    if (size == 0)
    {
        return NULL;
    }

    // 요청된 크기에 따라 조정된 블록 크기를 계산
    size_t adjusted_size;
    if (size <= DOUBLE_WORD_SIZE)
    {
        // 요청 크기가 최소 블록 크기 이하일 경우, 최소 블록 크기로 조정
        adjusted_size = 2 * DOUBLE_WORD_SIZE;
    }
    else
    {
        // 요청 크기 + 헤더/풋터 크기를 더하고, 8바이트 정렬
        adjusted_size = ALIGN(size + DOUBLE_WORD_SIZE);
    }

    // 조정된 크기를 만족하는 적합한 블록을 찾기
    char *block_pointer = fit_finder(adjusted_size);

    if (block_pointer == NULL)
    {
        // 적합한 블록이 없으면 힙을 확장하여 새 블록 할당
        block_pointer = heap_extender(adjusted_size / WORD_SIZE);

        // 힙 확장이 실패하면 NULL 반환
        if (block_pointer == NULL)
        {
            return NULL;
        }
    }

    // 찾은 블록 또는 확장된 블록에 요청 크기를 배치
    placer(block_pointer, adjusted_size);

    // 사용 가능한 블록의 시작 주소 반환
    return block_pointer;
}
```

mm_malloc 함수는 동적 메모리 할당 요청을 처리하기 위해 설계된 함수이다. 주어진 크기(size)에 적합한 메모리 블록을 할당하여 그 시작 주소를 반환하며, 메모리 공간을 효율적으로 관리하기 위한 작업을 수행한다. 첫 번째로 요청된 크기를 검증한다. 만약 요청된 크기(size)가 0이면, 유효한 메모리 할당이 불필요하므로 NULL을 반환한다. 만약 0보다 크다면 요청된 크기를 기반으로 메모리 블록의 크기를 계산하고 조정한다. 이때 요청 크기가 최소 블록 크기 이하인 경우, 블록을 헤더, 풋터, 정렬을 포함한 블록의 최소 크기를 보장하기 위해 최소 크기인 2 * DOUBLE_WORD_SIZE로 조정한다. 만약 요청 크기가 최소 블록 크기보다 큰 경우 블록 크기를 요청된 크기에 헤더와 풋터의 크기(DOUBLE_WORD_SIZE)를 더한 뒤, 8바이트 정렬 규칙에 따라 조정한다. 이후에 fit_finder 함수를 호출하여 요청된 크기(adjusted_size)를 만족할 수 있는 free 블록을 탐색한다. Next Fit 알고리즘을 사용하며 탐색 시작 위치는 next_p로 설정한다. 적합한 블록이 존재하는 경우 요청된 크기를 만족하는 블록의 시작 주소가 반환되고, 만약 존재하지 않는 경우 heap_extender를 호출하여 힙을 확장하고 추가 메모리를 확보한다. 이때 힙 확장이 실패한 경우 NULL을 반환하고 성공하면 새로 확장된 블록의 시작 주소를 반환받는다. 메모리 블록을 확보했으면 placer 함수를 호출하여 요청된 크기만큼 메모리를 배치한다. 메모리 할당 완료 후 요청된 크기만큼 할당된 블록의 시작 주소(block_pointer)를 반환한다.

[mm_free()]

<code>

```
void mm_free(void *ptr)
{
    // 예외 처리: 유효하지 않은 포인터는 처리하지 않음
    if (ptr == NULL)
    {
        return;
    }

    // 현재 블록의 크기를 가져옴
    size_t size = GET_BLOCK_SIZE(HEADER_PTR(ptr));

    // 블록의 헤더와 풋터를 free 상태로 설정
    PUT_WORD(HEADER_PTR(ptr), PACK_BLOCK(size, 0)); // 헤더에 크기와 할당 상태 저장
    PUT_WORD(FOOTER_PTR(ptr), PACK_BLOCK(size, 0)); // 풋터에 크기와 할당 상태 저장

    // 현재 블록을 병합하여 단편화를 줄임
    coalescer(ptr);
}
```

mm_free 함수는 사용자가 동적 메모리 할당으로 확보한 블록을 해제하는 함수이다. 메모리를 해제하면 해당 블록을 다시 사용 가능한 상태로 전환하고, 추가적으로 인접한 free 블록과 병합하여 단편화를 줄이는 작업을 한다. 첫 번째로 인자로 전달된 포인터(ptr)가 유효한지 확인한다. 만약 ptr이 NULL이라면, 유효하지 않으므로 함수를 종료한다. ptr이 유효하다면 ptr이 가리키는 블록의 크기를 확인하기 위해 해당 블록의 헤더 정보를 읽는다. GET_BLOCK_SIZE 매크로를 사용하여 헤더에서 블록 크기를 추출하며, 할당 상태 정보를 제외한 순수한 블록 크기를 확보한다. 그리고 블록을 free 상태로 전환하기 위해 해당 블록의 헤더와 풋터를 수정한다. 헤더의 경우, PACK_BLOCK 매크로를 사용하여 블록 크기와 할당 상태(0)를 결합한 값을 생성하고, 이를 HEADER_PTR가 가리키는 위치에 기록한다. 풋터 역시 동일한 방식으로 크기와 할당 상태를 기록하며, 풋터의 위치는 블록 크기를 이용해 계산한다. 해제된 블록은 인접한 free 블록과 병합할 수 있고, 병합 과정은 free 블록의 크기를 키워 메모리 단편화를 줄이고 메모리 풀의 연속성을 높이기 때문에 coalescer 함수를 호출하여 단편화를 줄이는 작업을 수행한다.

[mm_realloc()]

<code>

```
void *mm_realloc(void *ptr, size_t size)
{
    // 기존 블록의 포인터 저장
    void *old_ptr = ptr;

    // 새로운 블록을 저장할 포인터 선언
    void *new_ptr;

    // 새롭게 필요한 크기를 계산 (요청 크기 + 헤더/푸터 크기)
    size_t new_size = size + DOUBLE_WORD_SIZE;

    // 기존 블록의 크기를 가져옴
    size_t old_size = GET_BLOCK_SIZE(HEADER_PTR(old_ptr));

    // 현재 블록을 포함한 누적 크기 초기화
    size_t total_size = old_size;

    // 블록 병합을 위한 플래그
    int found_sufficient_space = 0;
    int wrap_around_flag = 0;

    // 1. 요청 크기가 기존 블록 크기 이하라면, 기존 블록을 그대로 사용
    if (new_size <= old_size)
    {
        return old_ptr;
    }

    // 2. 병합 가능한 연속된 free 블록을 검색
    void *temp_ptr = old_ptr;
    while (GET_ALLOC_STATUS(HEADER_PTR(NEXT_BLOCK_PTR(temp_ptr))) == 0)
    {
        // 다음 블록의 크기를 누적
        total_size += GET_BLOCK_SIZE(HEADER_PTR(NEXT_BLOCK_PTR(temp_ptr)));

        // 탐색 중 `next_p`를 초과했는지 확인
        if (temp_ptr == next_p)
        {
            wrap_around_flag = 1;
        }
```

```
        // 누적 크기가 요청 크기를 충족하면 탐색 종료
        if (new_size <= total_size)
        {
            found_sufficient_space = 1;
            break;
        }

        // 다음 블록으로 이동
        temp_ptr = NEXT_BLOCK_PTR(temp_ptr);
    }

    // 3. 충분한 크기의 연속된 블록을 찾았을 경우
    if (found_sufficient_space)
    {
        // 기존 블록을 확장하여 헤더와 푸터를 업데이트
        PUT_WORD(HEADER_PTR(old_ptr), PACK_BLOCK(total_size, 1)); // 헤더 갱신
        PUT_WORD(FOOTER_PTR(old_ptr), PACK_BLOCK(total_size, 1)); // 푸터 갱신

        // 탐색 중 `next_p`를 초과했으면 `next_p`를 갱신
        if (wrap_around_flag)
        {
            next_p = old_ptr;
        }

        // 기존 블록 확장 후 반환
        return old_ptr;
    }

    // 4. 충분한 크기의 블록을 찾지 못한 경우
    // 새 블록을 할당
    new_ptr = mm_malloc(new_size);

    // 할당 실패 시 NULL 반환
    if (new_ptr == NULL)
    {
        return NULL;
    }

    // 5. 기존 블록의 데이터를 새 블록으로 복사
    memcpy(new_ptr, old_ptr, old_size);

    // 기존 블록을 free
    mm_free(old_ptr);

    // 새 블록의 포인터 반환
    return new_ptr;
}
```

mm_realloc 함수는 이미 할당된 메모리 블록의 크기를 조정하는 함수이다. 주어진 블록(ptr)을 사용자가 요청한 크기(size)에 맞게 확장하거나 축소하며, 필요 시 새로운 메모리 블록을 할당하고 기존 데이터를 복사한다. 먼저 초기화 및 크기 계산 작업을 수행한다. 기존 블록의 포인터(old_ptr)를 저장하고, 새 블록을 저장할 포인터(new_ptr)를 선언한다. 새롭게 필요한 크기(new_size)는 사

용자가 요청한 크기(size)에 헤더와 풋터의 크기를 더해 계산하며 기존 블록의 크기(old_size)는 해당 블록의 헤더에서 읽어오고, 병합 가능한 인접 블록의 크기를 추적하기 위해 total_size를 기존 블록의 크기로 초기화한다. 그리고 기존 블록 크기와 요청 크기를 비교한다. 요청된 크기(new_size)가 기존 블록 크기(old_size)보다 작거나 같다면, 현재 블록이 이미 충분한 크기를 가지고 있으므로 기존 블록 포인터(old_ptr)를 반환하여 함수를 종료한다. 만약 요청 크기가 기존 블록 크기보다 크다면, 현재 블록 주변에 있는 연속된 free 블록을 검색하여 병합 가능성을 확인한다. 이를 위해, temp_ptr를 시작 블록으로 설정하고 다음 블록(NEXT_BLOCK_PTR)으로 이동하며, free 상태인 블록의 크기를 누적(total_size)한다. 탐색 도중, 현재 위치가 탐색 시작 지점(next_p)을 초과하면 wrap-around 상태로 간주하고 wrap_around_flag를 설정한다. 누적 크기가 요청 크기 이상이면 탐색을 종료한다. 만약 충분한 크기의 연속된 블록을 찾았을 경우, 기존 블록의 헤더와 풋터를 업데이트하여 병합된 블록을 하나의 확장된 블록으로 만든다. 그리고 헤더와 풋터를 갱신하는데, 헤더의 경우 병합된 블록의 전체 크기와 할당 상태(1)를 기록하고 풋터의 경우 병합된 블록의 크기와 동일한 값을 기록한다. 또한, 탐색 도중 wrap-around 상태가 발생했다면(플래그를 설정했었다면) next_p를 확장된 블록의 시작 주소로 갱신한다. 이후 기존 블록 포인터(old_ptr)를 반환하여 함수가 종료된다. 충분한 크기의 블록을 찾지 못한 경우, mm_malloc을 호출하여 요청 크기(new_size)를 가진 새 블록을 할당받는다. 할당이 실패하면 NULL을 반환하고 성공하면 기존 블록(old_ptr)의 데이터를 새로운 블록(new_ptr)로 memcpy 함수를 호출해 복사한다. 데이터를 복사한 후, 기존 블록은 더 이상 필요하지 않기에 mm_free를 호출하여 해당 블록을 해제하고 새 블록의 포인터를 반환하면서 함수를 종료한다.

[heap_extender()]

<code>

```c
static void *heap_extender(size_t words)
{
    // 확장할 힙 영역의 시작 포인터와 크기를 선언
    char *bp;
    size_t size;

    // 요청된 word 수가 짝수인지 확인하여 size를 계산
    if (words % 2 == 0)
    {
        size = WORD_SIZE * words; // 짝수일 경우 그대로 계산
    } else
    {
        size = WORD_SIZE * (words + 1); // 홀수일 경우 한 word 추가
    }

    // 힙을 요청된 크기만큼 확장
    bp = mem_sbrk(size);

    // 힙 확장이 실패한 경우 NULL 반환
    if (bp == (void *)-1)
    {
        return NULL;
    }

    // 새로 확장된 블록의 헤더와 풋터를 설정 (free 상태)
    PUT_WORD(HEADER_PTR(bp), PACK_BLOCK(size, 0));  // 헤더 설정
    PUT_WORD(FOOTER_PTR(bp), PACK_BLOCK(size, 0));  // 풋터 설정

    // 새로운 에필로그 헤더를 생성 (할당된 상태, 크기 0)
    PUT_WORD(HEADER_PTR(NEXT_BLOCK_PTR(bp)), PACK_BLOCK(0, 1));

    // 새로 확장된 블록을 병합하고 병합된 블록의 시작 포인터를 반환
    return coalescer(bp);
}
```

heap_extender 함수는 힙 영역을 확장하여 새로운 메모리 블록을 추가로 확보하는 함수이다. 사용자가 요청한 크기(words)를 기반으로 확장할 메모리 크기를 계산하고, 힙의 끝부분을 확장한 후 이를 초기화한다. 또한, 확장된 블록을 주변 블록과 병합하여 메모리 단편화를 줄이고, 병합된 블록의 시작 주소를 반환한다. 우선 요청된 크기(words)를 기준으로 확장할 메모리 크기(size)를 계산한다. 블록의 크기는 항상 더블 워드(8바이트) 단위로 정렬되어야 하므로, 요청된 words가 홀수인 경우 한 word를 추가하여 짝수 크기로 만든다. 그리고 계산된 크기(size)만큼 힙을 확장하기 위해 mem_sbrk 함수를 호출한다. 확장이 성공하면 새로운 힙 영역의 시작 주소를 반환받고, 확장이 실패할 경우 NULL을 반환 후 종료한다. 힙 확장이 성공하면, 새로 추가된 블록의 헤더와 풋터를 초기화한다.헤더의 경우 블록의 크기(size)와 할당 상태(0, 즉 free)를 기록하고, 풋터의 경우, 헤더와 동일한 정보를 기록하여 블록의 경계를 나타낸다. 그리고 크기가 0이며 항상 할당된 상태로 설정된 에필로그 블록(힙의 끝)을 생성한다. 마지막으로 단편화를 줄이기 위해 coalescer 함수를 호출해 병합 작업을 수행하고 병합된 블록의 시작주소를 반환하여 함수를 종료한다.

[coalescer()]

<code>

```
static void *coalescer(void *bp)
{
    // 이전 블록과 다음 블록의 할당 상태를 확인
    size_t prev_alloc = GET_ALLOC_STATUS(FOOTER_PTR(PREVIOUS_BLOCK_PTR(bp)));
    size_t next_alloc = GET_ALLOC_STATUS(HEADER_PTR(NEXT_BLOCK_PTR(bp)));

    // 현재 블록의 크기를 가져옴
    size_t size = GET_BLOCK_SIZE(HEADER_PTR(bp));

    // Case 1: 이전 블록과 다음 블록 모두 할당된 상태
    if (prev_alloc && next_alloc)
    {
        next_p = bp;  // `next_p`를 현재 블록으로 설정
        return bp;    // 병합할 필요가 없으므로 그대로 반환
    }

    // Case 2: 이전 블록은 할당되고 다음 블록은 free 상태
    if (prev_alloc && !next_alloc)
    {
        size += GET_BLOCK_SIZE(HEADER_PTR(NEXT_BLOCK_PTR(bp))); // 다음 블록 크기 추가
        PUT_WORD(HEADER_PTR(bp), PACK_BLOCK(size, 0));         // 헤더 갱신
        PUT_WORD(FOOTER_PTR(bp), PACK_BLOCK(size, 0));         // 풋터 갱신
    }

    // Case 3: 이전 블록은 free 상태이고 다음 블록은 할당된 상태
    else if (!prev_alloc && next_alloc)
    {
        size += GET_BLOCK_SIZE(HEADER_PTR(PREVIOUS_BLOCK_PTR(bp))); // 이전 블록 크기 추가
        PUT_WORD(FOOTER_PTR(bp), PACK_BLOCK(size, 0));             // 풋터 갱신
        PUT_WORD(HEADER_PTR(PREVIOUS_BLOCK_PTR(bp)), PACK_BLOCK(size, 0)); // 이전 블록 헤더 갱신
        bp = PREVIOUS_BLOCK_PTR(bp);                               // 블록 포인터를 이전 블록으로 이동
    }

    // Case 4: 이전 블록과 다음 블록 모두 free 상태
    else
    {
        size += GET_BLOCK_SIZE(HEADER_PTR(PREVIOUS_BLOCK_PTR(bp))) +
                GET_BLOCK_SIZE(FOOTER_PTR(NEXT_BLOCK_PTR(bp))); // 이전 및 다음 블록 크기 추가
        PUT_WORD(HEADER_PTR(PREVIOUS_BLOCK_PTR(bp)), PACK_BLOCK(size, 0)); // 이전 블록 헤더 갱신
        PUT_WORD(FOOTER_PTR(NEXT_BLOCK_PTR(bp)), PACK_BLOCK(size, 0));     // 다음 블록 풋터 갱신
        bp = PREVIOUS_BLOCK_PTR(bp);                                       // 블록 포인터를 이전 블록으로 이동
    }

    // `next_p`를 병합된 블록의 시작 포인터로 업데이트
    next_p = bp;

    // 병합된 블록의 시작 포인터 반환
    return bp;
}
```

coalescer 함수는 동적 메모리 할당 시스템에서 병합을 통해 메모리 단편화를 줄이기 위해 사용되는 함수이다. 전달받은 블록 포인터 bp를 기준으로, 이전 블록과 다음 블록의 상태(할당 여부)를 확인하고, 필요에 따라 블록을 병합한다. 우선 전달받은 블록 bp 주변의 메모리 상태를 확인하기 위해, 이전 블록과 다음 블록의 할당 여부를 조사하여 prev_alloc과 next_alloc에 저장한다. 그리고 현재 블록의 크기를 헤더에서 읽어와서 size에 저장하며, 이 값은 병합 시 새로운 블록 크기 계산에 사용된다. 함수는 이전 블록과 다음 블록의 상태에 따라 4가지 경우로 나누어 동작을 수행한다.

Case 1: 이전 블록과 다음 블록 모두 할당된 상태

- 현재 블록 주변에 free 상태의 블록이 없으므로 병합이 필요하지 않다. 따라서 next_p를 현재 블록 bp로 설정하고 그대로 반환한다.

Case 2: 이전 블록은 할당되고 다음 블록은 free 상태

- 현재 블록과 다음 블록을 병합한다.

- 다음 블록의 크기를 현재 블록 크기에 더한 후, 새로운 크기를 헤더와 풋터에 기록한다.

Case 3: 이전 블록은 free 상태이고 다음 블록은 할당된 상태

- 현재 블록과 이전 블록을 병합한다.

- 이전 블록의 크기를 현재 블록 크기에 더한 후, 새로운 크기를 풋터와 헤더에 기록한다.

- 병합이 완료되었으므로 블록 포인터 bp를 이전 블록의 시작 주소로 업데이트한다.

Case 4: 이전 블록과 다음 블록 모두 free 상태

- 현재 블록과 이전 블록, 그리고 다음 블록을 모두 병합한다.

- 이전 블록, 현재 블록, 다음 블록의 크기를 합산하여 새로운 크기를 계산한다.

- 이전 블록의 헤더와 다음 블록의 풋터를 갱신한다.

- 블록 포인터 bp를 이전 블록의 시작 주소로 업데이트한다.

병합을 완료하면, 다음 메모리 탐색 시 시작 위치를 최적화하기 위해 next_p를 병합된 블록의 시작 주소로 업데이트한다. 그리고 병합 작업이 완료된 블록의 시작 포인터를 반환해 함수를 종료한다.

[fit_finder()]

<code>

```
static void *fit_finder(size_t size)
{
    // 힙을 순회하기 위한 포인터
    void *bp;

    // 적합한 블록을 찾았을 경우 해당 블록의 포인터를 저장할 변수
    void *next_fit_block = NULL;

    // next_p가 NULL일 경우 힙의 시작 지점으로 초기화
    if (next_p == NULL)
    {
        next_p = heap_p;
    }

    // 힙의 끝까지 순회하며 적합한 free 블록을 찾음
    bp = next_p;
    while (GET_BLOCK_SIZE(HEADER_PTR(bp)) > 0)
    {
        // 현재 블록이 할당된 상태면 다음 블록으로 이동
        if (GET_ALLOC_STATUS(HEADER_PTR(bp)))
        {
            bp = NEXT_BLOCK_PTR(bp);
            continue;
        }

        // 요청된 크기보다 크거나 같은 free 블록을 찾았을 경우
        if (size <= GET_BLOCK_SIZE(HEADER_PTR(bp)))
        {
            next_fit_block = bp; // 해당 블록을 적합한 블록으로 설정
            next_p = bp;         // next_p를 현재 블록으로 갱신
            break;               // 탐색 종료
        }

        // 다음 블록으로 이동
        bp = NEXT_BLOCK_PTR(bp);
    }

    // 적합한 블록의 포인터를 반환 (없으면 NULL 반환)
    return next_fit_block;
}
```

fit_finder 함수는 요청된 크기를 만족하는 적합한 블록을 힙에서 찾는 함수이다. 이는 동적 메모리 할당 시 적합한 free 블록을 탐색하여 반환하는 과정으로, next_fit 탐색 전략을 사용하여 이전 탐색 위치를 기준으로 이후 블록부터 탐색을 시작하며, 힙의 끝까지 순회하면서 적합한 블록을 찾는다. 만약 힙 끝까지 적합한 블록을 찾지 못하면, NULL을 반환하여 요청된 크기의 블록을 찾을 수 없음을 나타낸다. 변수 선언 부분에서 블록 포인터 bp는 탐색을 시작할 위치를 나타내고 next_fit_block는 적합한 블록을 찾았을 때 그 블록의 포인터를 저장하는 변수이다. 첫 탐색이거나 탐색 위치가 초기화된 경우에 대해서 next_p는 NULL일 경우 힙의 시작 포인터 heap_p로 초기화한다. 그리고 힙 순회 작업을 시작한다. bp를 next_p로 설정하여 이전 탐색에서 마지막으로 탐색된 위치 이후부터 탐색을 시작한다. 각 블록의 크기를 확인하기 위해 HEADER_PTR(bp)에서 헤더 정보를 읽어오고 GET_BLOCK_SIZE(HEADER_PTR(bp)) > 0 조건을 통해 블록 크기가 0이 아닌 동안 (즉, 에필로그 블록에 도달하기 전) 순회한다. GET_ALLOC_STATUS(HEADER_PTR(bp))를 사용해 현재 블록의 할당 여부를 확인하고 만약 블록이 이미 할당된 상태라면, 현재 블록을 건너뛰고 다음

블록(NEXT_BLOCK_PTR(bp))으로 이동한다. 이후 요청된 크기 size와 현재 블록 크기를 비교해서 현재 블록의 크기가 요청된 크기보다 크거나 같으면 적합한 블록으로 판단, next_fit_block에 현재 블록 포인터 bp를 저장한다. 그리고 next_p를 현재 블록 포인터로 업데이트하여 다음 탐색 시 이 블록 이후부터 시작하도록 한 후 탐색을 종료한다. 탐색이 종료된 후 블록이 적합한 경우 그 블록의 포인터를 반환하고 그렇지 않은 경우 NULL 상태일 것이기에 NULL을 반환한다.

[placer()]

<code>

```
static void placer(void *bp, size_t size)
{
    // 현재 블록의 크기를 가져옴
    size_t current_size = GET_BLOCK_SIZE(HEADER_PTR(bp));

    // Case 1: 블록을 분할할 수 있을 만큼 충분히 큰 경우
    if ((current_size - size) >= (2 * DOUBLE_WORD_SIZE))
    {
        // 현재 블록을 요청된 크기로 설정 (헤더와 푸터 갱신)
        PUT_WORD(HEADER_PTR(bp), PACK_BLOCK(size, 1));  // 헤더 설정
        PUT_WORD(FOOTER_PTR(bp), PACK_BLOCK(size, 1));  // 푸터 설정

        // 분할 후 남은 공간을 새로운 free 블록으로 설정
        void *next_bp = NEXT_BLOCK_PTR(bp);             // 다음 블록의 시작 주소 계산
        PUT_WORD(HEADER_PTR(next_bp), PACK_BLOCK(current_size - size, 0)); // 헤더 갱신
        PUT_WORD(FOOTER_PTR(next_bp), PACK_BLOCK(current_size - size, 0)); // 푸터 갱신

        // `next_p`를 새로 생성된 free 블록으로 업데이트
        next_p = next_bp;

    // Case 2: 블록을 분할할 수 없을 만큼 작은 경우
    } else
    {
        // 현재 블록 전체를 할당 상태로 설정 (헤더와 푸터 갱신)
        PUT_WORD(HEADER_PTR(bp), PACK_BLOCK(current_size, 1)); // 헤더 설정
        PUT_WORD(FOOTER_PTR(bp), PACK_BLOCK(current_size, 1)); // 푸터 설정

        // `next_p`를 현재 블록으로 업데이트
        next_p = bp;
    }
}
```

placer 함수는 요청된 크기의 메모리를 적합한 블록에 배치하는 함수이다. 우선 현재 블록의 크기를 확인하는데, GET_BLOCK_SIZE 매크로를 사용하여 블록 포인터 bp가 가리키는 블록의 크기를 가져온다. 그리고 블록을 분할할 수 있을 만큼 여유 공간이 있는지 확인하기 위해 현재 블록 크기와 요청된 크기의 차이를 계산하여 여유 공간이 2 * DOUBLE_WORD_SIZE 이상인 경우(즉, 헤더와 풋터를 포함한 최소 블록 크기를 초과하는 경우) 블록 분할이 가능하므로 PUT_WORD 매크로를 사용하여 블록의 헤더와 풋터를 요청된 크기와 할당 상태(1)로 설정한다. 그리고 요청된 크기만큼 현재 블록을 이동하여 새로운 블록의 시작 주소를 계산 후 새로운 free 블록의 헤더와 풋터를 설정하여 크기와 할당 상태(0)를 저장, 이후 메모리 탐색 시 효율성을 높이기 위해 next_p를 새로 생성된 free 블록의 시작 주소로 업데이트한다. 만약 블록 분할이 불가능한 경우 PUT_WORD를 사용하여 블록의 헤더와 풋터를 현재 블록 크기와 할당 상태(1)로 설정하고 탐색 위치를 최신 상태로 유지하기 위해서 next_p를 현재 블록의 시작 주소로 설정한다.

[test]

```
[yojun313@programming2 7_MallocLab]$ ./mdriver -V
Using default tracefiles in ./traces/
Measuring performance with gettimeofday().

Testing mm malloc
Reading tracefile: amptjp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: amptjp.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary2.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cccp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cccp.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: coalescing-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: coalescing.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cp-decl-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cp-decl.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: expr-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: expr.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random2.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc2.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: short1-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: short1.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: short2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: short2.rep
Checking mm_malloc for correctness, efficiency, and performance.
```

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   86%    5694  0.000071 80197
 1       yes   86%    4805  0.000060 79817
 2       yes   55%   12000  0.000144 83507
 3       yes   55%    8000  0.000102 78431
 4       yes   51%   24000  0.000257 93276
 5       yes   51%   16000  0.000188 85288
 6       yes   90%    5848  0.000076 76947
 7       yes   90%    5032  0.000065 77895
 8       yes   66%   14400  0.000081178439
 9       yes   66%   14400  0.000081177778
10       yes   94%    6648  0.000093 71407
11       yes   94%    5683  0.000079 72211
12       yes   95%    5380  0.000075 72118
13       yes   95%    4537  0.000071 64173
14       yes   84%    4800  0.000798  6015
15       yes   84%    4800  0.023421   205
16       yes   82%    4800  0.000806  5959
17       yes   82%    4800  0.000768  6248
18       yes   45%   14401  0.000246 58493
19       yes   45%   14401  0.000243 59263
20       yes   53%   14401  0.000070207209
21       yes   53%   14401  0.000069207507
22       yes   66%      12  0.000000 40000
23       yes   66%      12  0.000000 60000
24       yes   90%      12  0.000000 60000
25       yes   90%      12  0.000000 60000
Total          74%  209279  0.027863  7511

Perf index = 44 (util) + 40 (thru) = 84/100
```

Correctness 평가 항목은 모든 valid가 yes가 나오면서 만점을 받았음을 알 수 있다. 성능은 메모리 활용도(util)에서 44%, 처리량(Thru)에서 40%가 나왔는데 메모리 활용도 부분에서 낭비가 있었음을 알 수 있다. Util의 경우, 블록 분할 및 병합이 이루어지지 않아 작은 요청에 큰 블록이 낭비되었거나, 요청된 크기보다 큰 블록 할당으로 인해 내부 단편화가 일어났을 수 있다는 생각이 들었다. 그리고 처리량의 경우 fit_finder에서 Next-fit을 수행하면서 힙을 순차적으로 탐색할 때 시간이 소요되고 병합 과정에서 조건 확인 중 처리 속도를 저하시켰을 가능성이 있다는 생각이 들었다. 다만 블록 병합과 적절한 탐색 전략 선택, mm_realloc()을 통해 기존 블록 크기가 충분하면 데이터 복사 없이 그대로 재사용하는 방법으로 속도를 최적화함으로써 이정도 점수가 나온 것에 만족한다.