Dynamic Allocation

Created	@2023년 6월 23일 오후 7:15
Class	
Type	
Materials	
Reviewed	☐
파일과 미디어	

▼ 프로젝트 목표

- malloc, realloc, free를 직접 구현해봄으로써 Dynamic allcoation의 logic과 동작을 이해한다.
- 직접 구상한 알고리즘을 적용함으로써, throughput과 utilization을 최적화하여 메모리 낭비를 줄이고, 요청에 빠르게 대응할 수 있도록한다.

▼ 전체 로직

- heap에 있는 모든 블록은 할당된 블록, 할당되지 않은 블록으로 나뉜다.
- 모든 블록은 포인터에 저장되어 있어야 관리할 수 있다.
- 할당되어 있는 블록은 caller가 할당받으면서 해당 블록을 기억하고 있다.
- 할당되지 않은 블록, free 블록은 따로 관리해줘야 한다.
- 이를 위하여 segregated_list를 사용한다.
- segregated_list는 연결리스트 배열이다.
- 모든 free block의 주소들은 segreageted_list에 저장되어 있다.
- 각 연결리스트는 해당하는 사이즈들의 free_block들을 연결한다.
- 각 연결리스트 안에 있는 free_block들의 크기는 정렬되어있다.
- free를 할 경우 해당블록은 free되는 동시에 segregated_list에 들어간다.
- free 이후, 연속된 free block은 병합해서 segregated_list에 저장한다.
- malloc 또는 realloc을 할 경우 segregated list를 순회하며 위치할 곳을 찾는다.

- 위치한 곳을 찾은 후, 할당 후 남은 블록은 다시 segregated_list에 들어간다.

▼ macro 설명

```
#define ALIGNMENT 8 // align size
#define WSIZE 4     // word size
#define DSIZE 8     // double word size
```

- 메모리를 할당하기 위한 기본 메모리 단위이다.
- 32-bit 에서는 alignment가 8byte가 된다.
- 4byte는 하나의 word size이고, DSIZE는 double word size이다.

```
/* chunksize */
#define CHUNKSIZE (1 << 12)
#define SMALLCHUNKSIZE (1 << 6)
```

- heap의 크기를 늘릴 때 사용하는 매크로이다.
  - CHUNKSIZE는 heap을 늘릴 크기를 말한다.
  - heap의 크기를 늘리는 것은 소모가 크기 때문에 한 번에 늘릴 때 충분한 양을 늘릴 수 있게 하기 위해 CHUNKSIZE를 2의 12제곱, 4048byt로 하였다.
  - SMALLCHUNKSIZE는 초반에 늘릴 크기를 말하는데 초반부터 너무 크게 heap을 크게 할당할 경우 utilization에 있어 비효율을 야기하기 때문에 2의 6제곱 64byte로 설정하였다.

```
/* max min */
#define MAX(x, y) ((x) > (y) ? (x) : (y))
#define MIN(x, y) ((x) < (y) ? (x) : (y))
```

- 둘 중 더 큰 값이 되는 MAX 매크로와 둘 중 더 작은 값이 되는 MIN 매크로이다.
- 따로 함수를 사용하거나 if문을 사용할 경우 코드의 가독성에 안 좋고, 코드가 길어지기 때문에 삼항연산자를 사용하여 위의 두 기능을 구현하였다.
- MAX는 힙의 크기를 늘릴 때, 해당 힙의 크기를 결정하는데 사용된다.
- 할당해야하는 힙의 크기가 CHUNKSIZE로도 부족할 경우를 위해 MAX를 사용하여 할당해야하는 메모리의 크기와 CHUNKSIZE 중 더 큰 값을 이용한다.
- MIN은 realloc에서 이전에 위치해있던 메모리의 내용을 새로 할당할 곳에 overflow가 발생하지 않으면서 memcpy하기 위해 사용한다.

- 이전에 위치해있던 메모리의 크기와 새로 할당할 메모리의 크기 중 작은 값을 골라 딱 그 만큼만 복사한다.

```
#define PACK(size, alloc) ((size) | (alloc))
```

- 메모리의 Header와 Footer에 넣을 값을 만들 때 사용된다.
- 메모리는 항상 8byte로 allign되기 때문에 size의 크기는 항상 8의 배수이므로 하위비트 3개는 항상 0이라 하위비트 3개는 자유롭게 사용할 수 있다.
- 메모리의 하위 비트 1개는 allocate 되어있는 지를 기억하는데에 사용된다.
- 메모리가 allocate되었다면 alloc을 1로하여 OR연산을 통해 하위 비트 1개를 1로 설정한다.

```
#define PUT(p, val) (*(unsigned int *)(p) = (unsigned int)(val))
```

- 이번 프로젝트에서는 포인터 연산이 굉장히 어렵고 복잡하기 때문에 모든 포인터 연산을 매크로로 만들어놓았다.
- PUT은 포인터인 p에 val의 값을 삽입하는 기능이다.
- p를 참조할 경우 val에 있던 주소로 참조하게 된다.

```
#define GET_SIZE(p) ((*(unsigned int *)(p)) & ~0x7)
#define GET_ALLOC(p) ((*(unsigned int *)(p)) & 0x1)
```

- 위는 PACK과 연관되어 있는 macro인데 PACK한 데이터에서 size의 정보와 alloc flag를 가져오기 위한 매크로이다.
- pack한 데이터는 size와 alloc 두 개의 정보를 모두 담고 있다.
- 마지막 하위비트 하나는 size속에서 항상 0인 비트 3개에 alloc을 저장해놓은 size와 상관없는 데이터이다.
- size는 항상 allign되어 있으므로 항상 8의 배수이므로 하위비트 3개는 항상 000이다.
- PACK한 데이터를 7(111)과 &~연산을 통해 하위 비트 3개를 원래 사이즈의 정보가 될 수 있도록 000으로 바다.

```
#define HEADER(bp) ((char *)(bp)-WSIZE)
#define FOOTER(bp) ((char *)(bp) + GET_SIZE(HEADER(bp)) - DSIZE)

#define NEXT_BLOCK(bp) ((char *)(bp) + GET_SIZE(HEADER(bp)))
#define PREV_BLOCK(bp) ((char *)(bp)-GET_SIZE(((char *)(bp)-DSIZE)))
```

- HEADER와 FOOTER에 접근하기 위한 매크로이다.
- 다음 인접 블록과 이전 인접 블록에 접근하기 위한 매크로이다.
- HEADER는 메모리의 포인터 앞에 있다. 이를 접근하기 위해 HEADER의 크기인 WSIZE만큼 앞으로 이동하여 HEADER의 포인터로 이동한다.
- HEADER에서 블록의 size를 알아온다.
- 메모리 포인터에 알아온 size만큼 뒤로 이동하면 다음 인접 블록의 메모리 포인터에 위치할 수 있다.
- FOOTER는 할당된 메모리 가장 뒤에 위치해 있다.
- 이를 접근하기 위해 다음 인접 브록의 메모리 포인터로 이동한다.
- 여기서 WSIZE만큼 앞으로 이동하면 다음 인접한 블록의 HEADER에 위치하게 되고 한 번 더 WSIZE만큼 앞으로 이동하면 FOOTER에 위치할 수 있다.
- 이전 인접 블록에 위치하는 방법은 이전 인접 블록의 FOOTER에 접근하여 이전 인접 블록의 size를 알아온 후 현재 블록 메모리 포인터에서 해당 size만큼 앞으로 이동한다.

```
#define NEXT_FREE_PTR(ptr) ((char *)(ptr))
#define NEXT_FREE_REFER(ptr) (*(char **)(ptr))

#define PREV_FREE_PTR(ptr) ((char *)(ptr) + WSIZE)
#define PREV_FREE_REFER(ptr) (*(char **)(PREV_FREE_PTR(ptr)))
```

- segregated free list를 관리하기 위해 사용하는 포인터이다.
- 고난도의 포인터 연산을 필요로 하기 때문에 실수할 일이 없게 매크로로 설정해 두었다.
- NEXT_FREE_PTR은 free block에서 header 바로 다음에 있다.
- 즉 메모리 포인터 자리에 다음 free 블록을 가리키는 주소가 저장되어 있다.
- NEXT_FREE_REFER은 free list를 순회하기 위해 사용된다.
- 한 번 접근하고 도착한 next free block의 포인터 또한 참조할 수 있는 포인터여야 하므로 해당 포인터를 참조할 때는 이중 포인터로 type casting을 하여서 접근해야한다.
- PREV_FREE_PTR은 NEXT_FREE_PTR 다음에 위치해있는데 이전 free_block의 주소를 저장하고 있다.
- PREV_FREE_REFER을 사용하여 PREV_FREE_PTR 참조할 때 또한 이중 포인터 type casting을 하여 참조후에도 다시 참조할 수 있게 한다.

▼ 코드 설명

▼ 초기화

```
// mm_init set
static void init_free_list();
static void set_logue();
int mm_init(void);

void **free_lists;
void *logue;
#define LIST 20
static void init_free_list()
{
    // printf("init_free_list\n");
    int index = 0;
    free_lists = (void **)malloc(sizeof(void *) * LIST);
    for (index = 0; index < LIST; index++)
    {
        free_lists[index] = NULL;
    }
}

static void set_logue()
{
    // printf("set_logue\n");
    //   cannot allocated the heap
    if ((long)(logue = mem_sbrk(4 * WSIZE)) == -1)
        return;
    // padding
    PUT(logue, 0);
    // input the prologue header
    PUT(logue + 1 * WSIZE, PACK(DSIZE, 1));
    // prologue footer
    PUT(logue + 2 * WSIZE, PACK(DSIZE, 1));
    // epilogue header
    PUT(logue + 3 * WSIZE, PACK(0, 1));
}

int mm_init(void)
{
    // printf("mm_init\n");
    init_free_list();
    set_logue();
    if (extend_heap(SMALLCHUNKSIZE) == NULL)
    {
        return -1;
    }
    return 0;
}
```

- mm_init에서 set_logue와 init_free_list를 호출하여 초기화를 한다.
- free_lists는 segregated list를 관리하기 위해 사용하는 변수이다.
- free_lists는 포인터배열이다. 연결리스트 배열로 이해할 수 있다.

- 배열 하나 하나의 연결리스트는 해당 인덱스에 해당하는 크기의 free_block을 연결리스트로 하여 저장하고 있다.
- init_free_list에서 연결리스트 배열을 초기화한다.
- LIST의 크기는 20개로 한다.
- index i일 경우 2의 i제곱보다 크고 2의 i + 1제곱보다 작은 free_block들을 연결하고 있다.
- set_logue()는 heap의 프롤로그와 에필로그를 설정한다.
- logue가 할당되어있다면 heap이 시작되어있음을 판단할 수 있다.

```
static void insert_free_block(void *ptr, size_t size);
static void delete_free_block(void *ptr);
static int which_index(size_t size);
static int which_index(size_t size)
{
    // printf("index_from_size\nsize : %d, ", size);
    int index = 0;
    while (size > 1)
    {
        size = size >> 1;
        index++;
        if (index >= LIST - 1)
        {
            break;
        }
    }
    return index;
}
static void insert_free_block(void *ptr, size_t size)
{
    // printf("insert node\n");
    int index = 0;
    void *cur;
    void *prev = NULL;

    index = which_index(size);
    cur = free_lists[index];
    //find a position
    while (cur != NULL && size > GET_SIZE(HEADER(cur)))
    {
        prev = cur;
        cur = NEXT_FREE_REFER(cur);
    }
    if (cur == NULL)
    {
        // empty
        if (prev == NULL)
        {
            PUT(NEXT_FREE_PTR(ptr), NULL);
            PUT(PREV_FREE_PTR(ptr), NULL);
            free_lists[index] = ptr;
        }
        // end of the list
        else
```

```
        {
            PUT(NEXT_FREE_PTR(ptr), NULL);
            PUT(PREV_FREE_PTR(ptr), prev);
            PUT(NEXT_FREE_PTR(prev), ptr);
        }
    }
    else
    {
        // beginning
        if (prev == NULL)
        {
            PUT(NEXT_FREE_PTR(ptr), cur);
            PUT(PREV_FREE_PTR(ptr), NULL);
            PUT(PREV_FREE_PTR(cur), ptr);
            free_lists[index] = ptr;
        }
        //beteween
        else
        {
            PUT(NEXT_FREE_PTR(ptr), cur);
            PUT(PREV_FREE_PTR(cur), ptr);

            PUT(NEXT_FREE_PTR(prev), ptr);
            PUT(PREV_FREE_PTR(ptr), prev);
        }
    }
    return;
}

static void delete_free_block(void *ptr)
{
    // // printf("delete node\n");
    int index = 0;
    int size = GET_SIZE(HEADER(ptr));
    index = which_index(size);
    if (NEXT_FREE_REFER(ptr) != NULL)
    {
        // between
        if (PREV_FREE_REFER(ptr) != NULL)
        {
            PUT(PREV_FREE_PTR(NEXT_FREE_REFER(ptr)), PREV_FREE_REFER(ptr));
            PUT(NEXT_FREE_PTR(PREV_FREE_REFER(ptr)), NEXT_FREE_REFER(ptr));
        }
        // beginning
        else
        {
            PUT(PREV_FREE_PTR(NEXT_FREE_REFER(ptr)), NULL);
            free_lists[index] = NEXT_FREE_REFER(ptr);
        }
    }
    else
    {
        // end
        if (PREV_FREE_REFER(ptr) != NULL)
        {
            PUT(NEXT_FREE_PTR(PREV_FREE_REFER(ptr)), NULL);
        }
        // alone
        else
        {
            free_lists[index] = NULL;
```

```
        }
    }
    return;
}
```

- insert_free_block은 새로운 free block을 segregated list에 삽입하는 기능을 한다.
- delete_free_block은 연결리스트에서 주어진 노드를 제거하는 기능을 한다.
- which_index는 size가 주어졌을 때 해당 size에 free_list 배열의 인덱스를 찾아준다. 예를 들어 168인 경우 2^7보다 크로 2^8보다 작으므로 index는 7이다.
- insert할 때는 which index를 사용하여 우선 주어진 size가 들어갈 free_list의 인덱스를 찾는다.
- 해당 연결리스트 안에서 해당 size가 위치할 수 있는 곳을 찾는다.
- 네 가지 경우에 따라서 노드를 추가한다.
- 비어있을 경우
  - 추가할 노드의 NEXT_FREE_PTR, PREV_FREE_PTR 은 NULL이 된다.
  - free_lists는 해당 노드를 가리킨다.
- 연결리스트의 끝에 위치할 경우
  - 이전 free block의 NEXT_FREE_PTR는 현재 추가된 노드를 가리킨다.
  - 현재 추가된 노드의 PREV_FREE_PTR은 이전 free block을 가리킨다.
  - 현재 추가된 노드의 NEXT_FREE_PTR에는 NULL이 들어간다.
- 가장 앞에 위치할 경우
  - 현재 추가할 노드의 NEXT_FREE_PTR은 가장 앞에 있는 노드를 가리킨다.
  - 현재 추가할 노드의 PREV_FREE_PTR은 NULL이 들어간다.
  - 원래 가장 앞이었던 노드의 PREV_FREE_PTR은 현재 추가할 노드를 가리킨다.
  - free_lists[index]는 현재 추가한 노드를 가리킨다.
- 사이에 위치할 경우
  - 현재 추가할 노드의 NEXT_FREE_PTR은 다음 노드를 가리킨다.
  - 다음 노드의 PREV_FREE_PTR은 현재 추가할 노드를 가리킨다.
  - 현재 추가할 노드의 PREV_FREE_PTR은 이전 노드를 가리킨다.
  - 이전 노드의 NEXT_FREE_PTR은 현재 추가할 노드를 가리킨다.
- delete_free_block은 주어진 free_block을 free_list에서 해제하기 위해 사용한다.

- 우선 해당 free_block이 위치해있는 연결리스트의 인덱스를 which_index를 사용하여 찾아낸다.
- 해당 노드의 위치에 따라 네 가지 경우로 나눠서 처리한다.
- 혼자 있을 경우
  - free_lists[index]는 NULL이 된다
- 가장 끝에 있을 경우
  - 삭제할 노드의 이전 노드의 NEXT_FREE_PTR은 NULL이 된다.
- 시작에 있을 경우
  - 삭제할 노드의 다음 노드의 PREV_FREE_PTR은 NULL이 된다..
  - free_lists[index]는 삭제할 노드의 다음노를 가리킨다.
- 사이에 있을 경우
  - 삭제할 노드의 다음 노드의 PREV_FREE_PTR은 삭제할 노드의 이전노드를 가리킨다.
  - 삭제할 노드의 이전 노드의 NEXT_FREE_PTR은 삭제할 노드의 다음노드를 가리킨다.

```
static void *coalesce(void *ptr)
{
    // printf("coalesce\n");
    size_t prev_all = GET_ALLOC(HEADER(PREV_BLOCK(ptr)));
    size_t next_all = GET_ALLOC(HEADER(NEXT_BLOCK(ptr)));
    size_t size = GET_SIZE(HEADER(ptr));

    if (prev_all == 1 && next_all == 1)
    {
        return ptr;
    }
    else if (prev_all == 1 && next_all == 0)
    {
        delete_free_block(ptr);
        delete_free_block(NEXT_BLOCK(ptr));
        size += GET_SIZE(HEADER(NEXT_BLOCK(ptr)));
        PUT(HEADER(ptr), PACK(size, 0));
        PUT(FOOTER(ptr), PACK(size, 0));
    }
    else if (prev_all == 0 && next_all == 1)
    {
        delete_free_block(ptr);
        delete_free_block(PREV_BLOCK(ptr));
        size += GET_SIZE(HEADER(PREV_BLOCK(ptr)));
        ptr = PREV_BLOCK(ptr);
        PUT(HEADER(ptr), PACK(size, 0));
        PUT(FOOTER(ptr), PACK(size, 0));
    }
    else
```

```
    {
        delete_free_block(ptr);
        delete_free_block(PREV_BLOCK(ptr));
        delete_free_block(NEXT_BLOCK(ptr));
        size += GET_SIZE(HEADER(PREV_BLOCK(ptr))) + GET_SIZE(HEADER(NEXT_BLOCK(ptr)));
        ptr = PREV_BLOCK(ptr);
        PUT(HEADER(ptr), PACK(size, 0));
        PUT(FOOTER(ptr), PACK(size, 0));
    }
    insert_free_block(ptr, size);
    return ptr;
}

static void *place(void *ptr, size_t asize)
{
    // printf("place\n");
    size_t size = GET_SIZE(HEADER(ptr));
    size_t free_size = size - asize;

    delete_free_block(ptr);

    if (free_size <= DSIZE * 2)
    {
        PUT(HEADER(ptr), PACK(size, 1));
        PUT(FOOTER(ptr), PACK(size, 1));
    }
    // small one into the front
    else if (asize > free_size)
    {
        PUT(HEADER(ptr), PACK(free_size, 0));
        PUT(FOOTER(ptr), PACK(free_size, 0));

        PUT(HEADER(NEXT_BLOCK(ptr)), PACK(asize, 1));
        PUT(FOOTER(NEXT_BLOCK(ptr)), PACK(asize, 1));
        insert_free_block(ptr, free_size);
        return NEXT_BLOCK(ptr);
    }
    else
    {
        PUT(HEADER(ptr), PACK(asize, 1));
        PUT(FOOTER(ptr), PACK(asize, 1));
        PUT(HEADER(NEXT_BLOCK(ptr)), PACK(free_size, 0));
        PUT(FOOTER(NEXT_BLOCK(ptr)), PACK(free_size, 0));
        insert_free_block(NEXT_BLOCK(ptr), free_size);
    }
    return ptr;
}

static void *find_place(size_t asize)
{
    void *p;
    for (int index = which_index(asize); index < LIST; index++)
    {
        if (index == LIST - 1 || free_lists[index])
        {
            p = free_lists[index];
            while (p != NULL && ((asize > GET_SIZE(HEADER(p)))))
            {
                p = NEXT_FREE_REFER(p);
            }
            if (p != NULL)
```

```
                break;
            }
        }
        return p;
    }
```

- free한 이후 앞 뒤 block이 free되어 있는지에 따라 병합해주는 기능이다.
- prev_all과 next_all은 각각 이전 인접 블록과 다음 인접 블록이 할당되어있는지를 저장하고 있다.
- 이전블록과 다음블록이 할당되어있다면 병합을 진행할 수 없다.
- 이전블록이나 다음블록만 할당되어있다면 현재 free블록과 인접 free블록을 segregated list에서 해제하고 둘을 합친 후 다시 segregated list에 넣는다.
- 이전블록과 다음블록이 둘다 free block이라면 이전 블록과 다음블록 모두 segregated list에서 제거하고 블록 세개를 합친후 segregate list에 넣는다.
- find_place는 할당할 메모리가 어디에 위치해야할지 판단해주는 기능을 한다.
- 해당 size의 크기에 해당하는 인덱스를 찾은 후 해당 리스트 부터 순회하며 들어갈 수 있는 free_block을 탐색한다.
- 만약 해당 인덱스에서 찾지 못했을 경우 다음 인덱스에서 찾는다.
- 이와 같은 방법으로 찾다가 찾지 못했을 경우는 heap을 늘린다.
- place는 찾은 free_block에 해당 메모리를 할당하는 기능을 한다.
- 만약 할당하고 남은 크기가 크지 않다면 남은크기까지 할당하려고 했던 크기에 추가하여 할당한다.
- 할당하고 남은 블록의 크기와 할당한 블록의 크기를 비교하여 둘 중 더 작은 값을 앞에 둘 수 있도록 한다.
- 이와 같은 질서를 부여하면 Utilization이 좋아진다.
- free_block에 block을 할당한 후 할당한 메모리의 위치를 반환한다.

```
static void *extend_heap(size_t size)
{
    // printf("extend heap\n");
    void *ptr;
    if ((ptr = mem_sbrk(size)) == (void *)-1)
        return NULL;
    // header
    PUT(HEADER(ptr), PACK(size, 0));
    // footer
    PUT(FOOTER(ptr), PACK(size, 0));
    // epilogue
    PUT(HEADER(NEXT_BLOCK(ptr)), PACK(0, 1));
    // insert_free_block
    insert_free_block(ptr, size);
```

```
    return coalesce(ptr);
}

void mm_free(void *ptr)
{
    // printf("mm_free\n");
    size_t size = GET_SIZE(HEADER(ptr));
    PUT(HEADER(ptr), PACK(size, 0));
    PUT(FOOTER(ptr), PACK(size, 0));
    insert_free_block(ptr, size);
    coalesce(ptr);
}
```

- extend_heap은 heap의 크기를 늘리는 기능을 한다.
- mem_sbrk에 size를 요청하면 해당 크기만큼 heap의 크기를 늘리고 늘린 메모리의 시작주소를 return한다.
- 에필로그를 수정한다.
- 늘린 크기는 free_block으로 만들어서 segregated_list에 넣는다.
- 만약 늘린 heap의 앞에 free_block이 있을 경우 병합하고 병합한 free_block의 시작주소를 return한다.
- mm_free는 free하려고하는 block을 free한 후 segregated_list에 넣는다.
- segregated_list에 넣은 후 병합한다.

```
void *mm_malloc(size_t size)
{
    // printf("mm_malloc\n");
    if (size == 0)
        return NULL;

    size_t asize;  // adjust size
    size_t extend; // extend heap if neccessary
    void *p;
    // align block size
    if (size <= DSIZE)     // line:vm:mm:sizeadjust1
        asize = 2 * DSIZE; // line:vm:mm:sizeadjust2
    else
        asize = DSIZE * ((size + (DSIZE) + (DSIZE - 1)) / DSIZE);

    p = find_place(asize);
    // printf("find_place\n");
    if (p == NULL)
    {
        extend = MAX(asize, CHUNKSIZE);
        // cannot extend the heap
        p = extend_heap(extend);
        if (p == (void *)-1)
            return NULL;
    }
    p = place(p, asize);
    return p;
```

```
}

void *mm_realloc(void *ptr, size_t size)
{ // printf("mm_realloc\n");
    if (size == 0)
        return NULL;

    void *newptr;
    size_t asize = size;

    newptr = ptr;
    // align block size
    if (asize <= DSIZE)    // line:vm:mm:sizeadjust1
        asize = 2 * DSIZE; // line:vm:mm:sizeadjust2
    else
        asize = DSIZE * ((asize + (DSIZE) + (DSIZE - 1)) / DSIZE);

    // no space
    if (GET_SIZE(HEADER(ptr)) < asize)
    {
        // extendable
        if (!GET_ALLOC(HEADER(NEXT_BLOCK(ptr))) || !GET_SIZE(HEADER(NEXT_BLOCK(ptr))))
        {
            int size_sum = GET_SIZE(HEADER(ptr)) + GET_SIZE(HEADER(NEXT_BLOCK(ptr)));
            //but still no space
            if (size_sum < asize)
            {
                newptr = mm_malloc(asize);
                memcpy(newptr, ptr, MIN(size, asize));
                mm_free(ptr);
            }
            //yes space
            else
            {
                delete_free_block(NEXT_BLOCK(ptr));
                PUT(HEADER(ptr), PACK(size_sum, 1));
                PUT(FOOTER(ptr), PACK(size_sum, 1));
            }
        }
        // not extendable
        else
        {
            newptr = mm_malloc(asize);
            memcpy(newptr, ptr, MIN(size, asize));
            mm_free(ptr);
        }
    }

    return newptr;
}
```

- mm_malloc은 malloc 기능을 담당하고 있다.
- 요청된 사이즈를 allign에 맞게 재조정하여 asize를 만든다.
- asize가 들어갈 수 있는 위치를 find_place를 통해 찾고 찾지 못했을 경우 heap의 크기를 늘린 후, 늘린 힙의 시작하는 부분에 해당 ptr를 위치시킨 후 해당 위치를 return한다.

- mm_realloc은 realloc기능을 담당한다.
- 요청된 사이즈를 allignment 에 맞게 재조정하여 asize를 만든다.
- 요청된 사이즈가 이미 할당된 사이즈보다 작다면 따로 위치를 옮겨줄 필요가 없다.
- 요청된 사이즈가 할당된 사이즈보다 크다면 우선 뒤로 확장할 수 있는지 판단한다.
- 뒤로 확장할 수 있다면 확장한 크기가 충분히 큰 지 판단한다.
- 만약 충분히 크다면 따로 재조정 없이 free_block과 할당된 블록만 최신화한다.
- 확장이 불가능하거나 확장한 크기가 충분히 크지 않다면 무조건 재조정을 해야한다.
- 요청된 size만큼 새로 할당하고 원래의 내용을 복사한다. 그리고 현재 할당되어있던 block은 free한다.