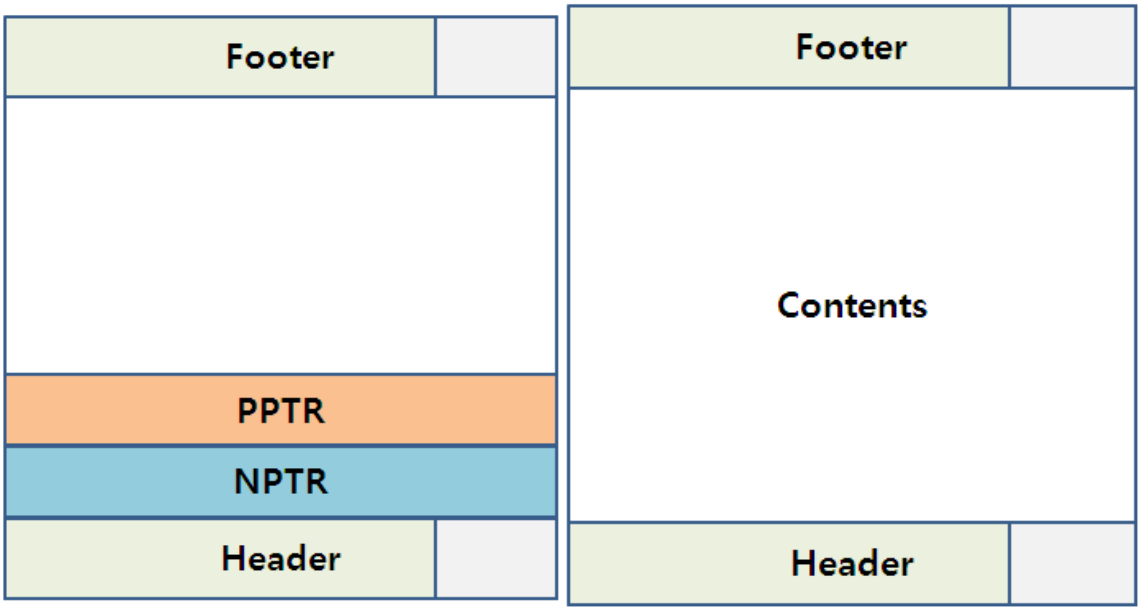
segregated free list를 활용하여 dynamic memory allocator를 구현했다.

본격적으로 구현하기에 앞서 free block과 allocated block의 구조를 아래의 그림과 같이 가정했다.



왼쪽이 free block, 오른쪽이 allocated block이다.

header, footer에 block의 사이즈가 저장되고 header, footer 끝의 1비트는 allocate되었는지를 나타내는 alloc bit이다. 그리고 free block의 PPTR, NPTR의 경우는, segregated free list를 사용할때, 각 list를 doubly linked list 형태로 구현할 계획이므로 이전 free block과 다음 free block의 포인터를 저장하는 용도로 사용된다.

그리고 구현에 필요한 매크로들을 정의했는데 새롭게 정의한 매크로들에 대한 설명은 mm.c 파일에 주석을 달아 모두 설명했으므로 여기서는 넘어가겠다.

또, 구현을 위해 여러 개의 함수들과 2개의 전역변수 heap_listp와 list를 선언해주었다. 일단 전역변수들에 대해 알아보자. 먼저 heap_listp이다. heap_listp는 heap의 prologue block의 footer를 가리키는 포인터 변수이다. 두번째는 list이다. list는 사이즈가 가장 작은, 즉 첫번째 free list의 base pointer를 가리키는 포인터 변수이다. 추후 mm_init함수를 이용하여 list[0]이 0번째 리스트를, list[1]이 1번째 리스트를 ... list[i]가 i번째 리스트를 가리키도록 만들어줄것이다.

mm_init, mm_malloc, mm_free, mm_realloc 함수를 설명하기 전에 구현을 위해 새롭게 만들어준 여러 함수들에 대해 먼저 설명하겠다.

```c
static void *extend_heap(size_t words)
{
    char *bp;
    size_t size;

    size = (words % 2) ? (words+1) * WSIZE : words * WSIZE; //for alignment
    if((long)(bp = mem_sbrk(size)) == -1) return NULL; //fail to expand the heap

    PUT(HDRP(bp), PACK(size, 0)); //make header; save size of block and make alloc bit free
    PUT(FTRP(bp), PACK(size, 0)); //make footer; save size of block and make alloc bit free
    PUT(FTRP(bp)+WSIZE, PACK(0, 1)); //new epilogue header

    return coalesce(bp); //Coalesce if the previous block was free
}
```

첫번째는 extend_heap 함수이다. 이 함수는 allocation request를 받았을 때 사용할 수 있는 free block을 찾지 못한 경우 사용하는 함수로, heap을 expand하여 새로운 free block을 만들어준다.

size 변수에 words가 짝수냐 홀수냐를 기준으로 나누어 필요한 사이즈를 저장해준다. 이것은 alignment 조건을 지키기 위해서이다. 그리고 mem_sbrk함수를 활용하여 실제로 heap을 expand 해주고, 만약 실패했다면 NULL을 리턴해준다.

그리고 PACK을 이용하여 size와 free를 나타내는 alloc bit인 0을 묶어 새롭게 만들어진 free block의 header에 저장해준다. footer에도 마찬가지로 저장해준다. 또, heap이 expand되었으므로 epilogue block의 header를 새롭게 만들어진 free block 위에 세팅해준다.

그리고 새롭게 만들어진 block의 previous block이 free된 상태라면 새롭게 만들어진 free block과 previous block을 coalesce 함수를 이용하여 합친다.

```
static void *coalesce(void *bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));

    if(prev_alloc && next_alloc) return bp; //prev - alloc, next - alloc
    else if(prev_alloc && !next_alloc) //prev - alloc, next - free
    {
        remove_list(NEXT_BLKP(bp));
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    }
    else if(!prev_alloc && next_alloc) //prev - free, next - alloc
    {
        remove_list(PREV_BLKP(bp));
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    else //prev - free, next - free
    {
        remove_list(PREV_BLKP(bp));
        remove_list(NEXT_BLKP(bp));
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }

    return bp;
}
```

이제 coalesce 함수에 대해 알아보자. coalesce함수는 block pointer를 받아와서 그 block의 prev, next block이 free block인지 체크하여 free block이라면 합쳐주는 함수이다. prev, next block이 각각 free 또는 alloc상태일 수 있으므로 경우의 수는 2*2=4이다.

먼저 prev와 next가 모두 allocate된 케이스는 합칠 수 있는 것이 없기 때문에 아무것도 하지 않고 block pointer를 그대로 리턴해준다.

또, next block만 free이거나 previous block만 free인 상태도 있다. 이 경우는 next block 또는 previous block과 합칠 수 있다. 먼저 prev/next block을 기존에 소속되어 있는 list에서 제거해준다. 그다음 block의 사이즈를 두 블럭의 합으로 바꿔주고, 두 블럭이 하나의 블럭이 되면 header와 footer가 각각 한쪽 블록의 것을 사용하게 된다. 여기에 update된 블럭의 사이즈와 free를 의미하는 alloc bit인 0을 PACK을 이용하여 넣어주면 된다.

마지막으로 previous, next block이 모두 free인 경우가 있다. 이 경우에는 두 블럭을 모두 기존의 list에서 제거해주고 size를 세 블럭의 합으로 업데이트 해준뒤, prev의 헤더와 next의 푸터가 합쳐진 블럭의 헤더와 푸터가 되므로 여기에 PACK을 이용하여 size와 alloc bit을 업데이트 해준다.

또, block pointer는 가정에 의해 block의 헤더 위 NPTR을 가리키고 있어야 하므로 prev가 free여서 prev block과 합쳐진 경우에는 마지막에 block pointer도 업데이트 해주어야하는 것을 잊어서는 안된다.

```
static void *find_fit(size_t asize)
{
    void *bp;
    void *base;
    int i;

    for(i = class_idx(asize); i < CNUM; i++) //Check all possible list in ascending order
    {
        base = BPTR(i);
        for(bp = CPTR(base); bp != NULL; bp = NPTR(bp)) //Search free block
        {
            if(asize <= GET_SIZE(HDRP(bp))) return bp; //suitable free block exists
        }
    } //search free block from possible minimum size list

    return NULL;
}
```

이제 find_fit 함수에 대해 알아보자. 이 함수는 allocation을 위해 free block을 탐색한다. 주어진 크기인 asize에 맞는 free list부터 탐색을 시작한다. 만약 여기서 asize에 맞는 free block을 못찾았으면, 더 큰 size를 가진 리스트에서 다시 탐색하고, 이 과정을 반복하여 끝까지 free block을 못찾은 경우 NULL을 리턴해준다. 이 함수에는 asize에 맞는 free list부터 first fit 기법을 이용하여 적합한 free block을 탐색하는 segregated free list의 핵심 원리가 담겨있다.

```
static void place(void *bp, size_t asize)
{
    size_t csize = GET_SIZE(HDRP(bp));

    remove_list(bp);

    if((csize - asize) >= 3*DSIZE) //rest part is enough to use later
    {
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        bp = NEXT_BLKP(bp);
        PUT(HDRP(bp), PACK(csize-asize, 0));
        PUT(FTRP(bp), PACK(csize-asize, 0));

        add_list(coalesce(bp));
    }
    else //rest part is too small to use later
    {
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }
}
```

이제 place 함수에 대해 알아보자. 이 함수는 어떤 block을 새롭게 allocate할때 사용한다. 먼저 csize에 bp, 즉 block pointer가 가리키는 block의 사이즈를 저장한다. 그리고 이 블럭을 할당할 것이므로 list에서 제거해준다.

이제 block의 사이즈와 할당요청이 들어온 사이즈의 차, 즉 csize-asize가 3*DSIZE 이상인지 아닌지를 기준으로 케이스를 나눈다. 이것은 free block의 일부를 할당했을때, 남은 파트가 사용가능한지를 기준으로 케이스를 나눈 것과 같다. 왜냐하면 free block은 헤더, 푸터, 그리고 previous/next block의 포인터인 PPTR, NPTR을 반드시 가지고 있어야한다. 헤더와 푸터의 사이즈는 word size와 같고, PPTR, NPTR의 사이즈는 8 byte이므로 double word size와 같다. 따라서 다 더했을때 3*DSIZE와 같으므로 3*DSIZE보다 적게 남게 되면, 이 부분은 사용할 수 없을 것이다.

그래서 3*DSIZE보다 남은 부분이 큰 경우는 free block에서 asize만큼만 할당해주고, 남은 부분으로 free block을 만들어준다. 그리고 새롭게 만들어진 이 free block을 list에 추가해준다.

3*DSIZE보다 남은 부분이 작은 경우는 어차피 free block으로 만들 수 없으므로 그냥 free block 전체를 할당해준다. 주의해야할 점은 전체를 할당했기 때문에, 헤더와 푸터에는 asize가 들어가지 않고 csize가 들어간다는 것이다.

```c
static void *BPTR(int idx)
{
    return ((char *)list + idx*DSIZE);
}
```

이제 BPTR 함수에 대해 알아보자. 이 함수는 인덱스 번호에 따라 해당 리스트의 베이스 포인터를 리턴해준다. list + idx*DSIZE로 idx번째의 리스트에 접근할 수 있는 이유는 mm_init에서 i번째 리스트를 가리키는 포인터와 i+1번째 리스트를 가리키는 포인터를 heap에 연속적으로 저장할 예정이기 때문이다.

```c
static int class_idx(unsigned int asize)
{
    int idx;

    if(asize == 24) idx = 0;
    else if(asize <= 32) idx = 1;
    else if(asize <= 64) idx = 2;
    else if(asize <= 128) idx = 3;
    else if(asize <= 256) idx = 4;
    else if(asize <= 512) idx = 5;
    else if(asize <= 1024) idx = 6;
    else if(asize <= 2048) idx = 7;
    else if(asize <= 4096) idx = 8;
    else if(asize <= 8192) idx = 9;
    else if(asize <= 16384) idx = 10;
    else if(asize <= 32768) idx = 11;
    else if(asize <= 65536) idx = 12;
    else if(asize <= 131072) idx = 13;
    else if(asize <= 262144) idx = 14;
    else if(asize <= 524288) idx = 15;
    else idx = 16;

    return idx;
}
```

이제 class_idx 함수에 대해 알아보자. 이 함수는 할당해야할 사이즈에 따라 해당되는 free block list의 인덱스 번호를 리턴해준다. free block의 최소 사이즈인 3*DSIZE, 즉 24는 0번 리스트에 넣고, 나머지는 2의 배수를 기준으로 리스트를 분류했다. 따라서 i번째 리스트에는 2^(i+3)+1 이상 2^(i+4) 이하의 free block들이 담기게 된다.

```c
static void add_list(void *bp)
{
    size_t size = GET_SIZE(HDRP(bp));
    int i = class_idx(size);
    void *base = BPTR(i);

    if(CPTR(base) == NULL) //list[i] is empty
    {
        PUT_NPTR(bp, NULL);
        PUT_PPTR(bp, NULL);
        PUT_CPTR(base, bp);
    }
    else //list[i] is not empty
    {
        PUT_PPTR(bp, NULL);
        PUT_NPTR(bp, CPTR(base));
        PUT_PPTR(CPTR(base), bp);
        PUT_CPTR(base, bp);
    }
}
```

이제 add_list 함수에 대해 알아보자. 먼저 헤더로부터 블럭의 사이즈를 받아와서 size 변수에 저장한다. 그리고 이 사이즈를 가지고 list index를 얻어 변수 i에 저장해주고, list[i]의 base pointer를 base 변수에 저장해준다.

CPTR(base) 즉, list의 head가 NULL이라면 list[i]는 빈 리스트라는 것이므로, 블럭의 NPTR, PPTR을 NULL로 설정해주고(prev/next block이 없으므로) list의 head를 bp로 바꿔준다.

CPTR(base)가 NULL이 아닌 경우, 즉 list[i]가 빈 리스트가 아닌 경우에는, bp의 NPTR이 기존의 head block의 포인터가 되고, 기존의 head block의 PPTR이 bp가 되며, list의 head가 bp를 가리키게 된다. LIFO policy를 사용할 것이므로 항상 head에 블럭을 삽입하도록 구현했다.

```
static void remove_list(void *bp)
{
    size_t size = GET_SIZE(HDRP(bp));
    int i = class_idx(size);
    void *base = BPTR(i);

    if(NPTR(bp) == NULL && PPTR(bp) == NULL) PUT_CPTR(base, NULL); //list[i].size == 1
    else if(NPTR(bp) != NULL && PPTR(bp) == NULL) //first element of list[i]
    {
        PUT_PPTR(NPTR(bp), NULL);
        PUT_CPTR(base, NPTR(bp));
        PUT_NPTR(bp, NULL);
        PUT_PPTR(bp, NULL);
    }
    else if(NPTR(bp) == NULL && PPTR(bp) != NULL) //last element of list[i]
    {
        PUT_NPTR(PPTR(bp), NULL);
        PUT_NPTR(bp, NULL);
        PUT_PPTR(bp, NULL);
    }
    else //middle of list[i]
    {
        PUT_NPTR(PPTR(bp), NPTR(bp));
        PUT_PPTR(NPTR(bp), PPTR(bp));
        PUT_NPTR(bp, NULL);
        PUT_PPTR(bp, NULL);
    }
}
```

이제 remove_list 함수에 대해 알아보자. 이 함수는 블럭을 free block list에서 제거해준다.

리스트에 이 블럭만 존재하는 경우, 리스트의 첫번째 블럭인 경우, 리스트의 마지막 블럭인 경우, 나머지 경우 총 4가지로 케이스를 분류하였다.

리스트에 이 블럭만 존재하는 경우는 list head를 NULL로 바꿔주는 것으로 끝이다. 두번째는 리스트의 첫번째 블럭인 경우인데, 이때는 블럭의 다음 블럭이 첫번째 블럭이 되어야 하므로 먼저 다음 블럭의 PPTR을 NULL로 바꿔주고, base가 다음블럭을 가리키도록 만들어준다. 그리고 bp의 NPTR, PPTR을 NULL로 바꿈으로써 연결을 끊어준다. 세번째 경우는 리스트의 마지막 블럭인 경우인데, 이때는 블럭의 이전 블럭이 마지막 블럭이 되어야 하므로 이전 블럭의 NPTR을 NULL로 바꿔주고, bp의 NPTR, PPTR을 NULL로 바꿈으로써 연결을 끊어준다. 마지막 케이스는 블럭의 이전 블럭과 다음 블럭을 연결해주고, 블럭의 연결을 끊어주면 끝이다.

이것으로 mm_init, mm_malloc, mm_free, mm_realloc 함수를 설명하기 위한 모든 함수들을 설명했다. 이제 이 4가지 함수를 설명하겠다.

```c
int mm_init(void)
{
    void *ptr;
    void *base;
    int i;

    //Init heap
    if ((list = mem_sbrk((CNUM+2) * DSIZE)) == (void *)-1) return -1;

    for(i = 0; i < CNUM; i++) //Make head of free list NULL
    {
        base = BPTR(i);
        PUT_CPTR(base, NULL);
    }

    heap_listp = list + CNUM*DSIZE;
    PUT(heap_listp, 0); //Alignment padding
    PUT(heap_listp + (1*WSIZE), PACK(DSIZE,1)); //Set prologue header
    PUT(heap_listp + (2*WSIZE), PACK(DSIZE,1)); //Set prologue footer
    PUT(heap_listp + (3*WSIZE), PACK(0,1)); //Set epliogue header
    heap_listp += (2*WSIZE); //heap_listp == prologue block footer

    //Extend heap
    if((ptr = extend_heap(CHUNKSIZE/WSIZE) == NULL)) return -1;
    base = BPTR(8); //(1<<12) / 4 * 4 = 2^12 = 4096 -> idx 8

    //Set initial free block
    PUT_CPTR(base, ptr);
    PUT_NPTR(base, NULL);
    PUT_PPTR(base, NULL);

    return 0;
}
```

먼저 mm_init 함수에 대해 알아보자. 먼저, mem_뉴가 함수를 사용하여 heap을 expand해준다. 이때 (CNUM+2)*DSIZE가 의미하는 것은 다음과 같다. 먼저 리스트의 개수만큼 리스트의 base pointer를 저장할 공간이 필요하므로 CNUM개의 DSIZE만큼의 공간이 필요하다. 또, alignment를 위해서 WSIZE만큼의 padding할 공간이 필요하고 또, prologue block의 헤더와 푸터, 그리고 epilogue block의 헤더를 위해 총 WSIZE*3만큼의 공간이 필요하기 때문이다.

그리고 list부터 CNUM번 DSIZE단위로 list들의 base pointer를 NULL로 바꿔준다. 그리고 prologue header, footer, epilogue header를 만들어준다.

그다음 heap을 CHUNKSIZE만큼 extend 해준다. CHUNKSIZE가 1<<12 = 2^12 = 4096이므로 이 free block은 list[8]에 해당한다. list[8]에 이 블럭을 추가해준다.

```
void *mm_malloc(size_t size)
{
    size_t asize;
    size_t extendsize;
    char *bp;

    if(size == 0) return NULL; //malloc returns NULL when size is 0

    //Adjust block size to include overhead and alignment reqs
    if(size <= DSIZE) asize = 2*DSIZE;
    else asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);

    if((bp = find_fit(asize)) != NULL) //search free block
    {
        place(bp, asize); //allocate
        return bp;
    }

    extendsize = MAX(asize, CHUNKSIZE); //need more memory
    if((bp = extend_heap(extendsize/WSIZE)) == NULL) return NULL;

    add_list(bp); //add bp to appropriate list
    place(bp, asize); //allocate

    return bp;
}
```

이제 mm_malloc 함수에 대해 알아보자. 먼저 size가 0인 경우는 바로 NULL을 리턴해준다. 그리고 asize를 overhead와 alignment 조건을 포함하여 조절해준다. 그다음 find_fit 함수를 이용하여 적합한 free block을 찾아준다. 만약 찾았다면, 즉 bp가 NULL이 아니라면 allocate해주고 bp를 리턴해준다. 못찾았다면, 더 많은 메모리 공간이 필요하다는 것이므로 extendsize만큼 heap을 extend해주어 새로운 큰 free block을 만들어준다. 그리고 이 블럭을 적절한 리스트에 넣어준 후 allocate해준뒤 bp를 리턴해준다.

```c
void mm_free(void *ptr)
{
    size_t size = GET_SIZE(HDRP(ptr));

    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));

    PUT_NPTR(ptr, NULL);
    PUT_PPTR(ptr, NULL);

    add_list(coalesce(ptr));
}
```

이제 mm_free 함수에 대해 알아보자. 먼저 헤더로부터 블럭의 사이즈를 얻는다. 그다음 블럭의 헤더와 푸터의 alloc bit을 0, 즉 free로 바꿔준다. 그리고 블럭의 NPTR, PPTR을 NULL로 reset해주고 coalesce 함수를 사용하여 prev/next block이 free block이라면 합쳐준 뒤 list에 넣어준다.

```c
void *mm_realloc(void *ptr, size_t size)
{
    void *oldptr = ptr;
    void *newptr;
    size_t copySize, nextSize, asize;

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

    copySize = GET_SIZE(HDRP(oldptr));
    if(size == copySize) return ptr; //requested same size

    nextSize = GET_SIZE(HDRP(NEXT_BLKP(oldptr)));
    if(!GET_ALLOC(HDRP(NEXT_BLKP(oldptr))) && copySize + nextSize >= size) //possible to coalesce with next block
    {
        //Adjust block size to include overhead and alignment reqs
        if(size <= DSIZE) asize = 2*DSIZE;
        else asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);

        //coalesce
        remove_list(NEXT_BLKP(oldptr));
        copySize += nextSize;
        PUT(HDRP(oldptr), PACK(copySize, 0));
        PUT(FTRP(oldptr), PACK(copySize, 0));

        place(oldptr, asize); //reallocate
        newptr = oldptr;
    }
    else
    {
        newptr = mm_malloc(size);
        if(size < copySize) copySize = size;
        memcpy(newptr, oldptr, copySize);
        mm_free(oldptr);
    }

    return newptr;
}
```

이제 mm_realloc 함수에 대해 알아보자. 먼저 write_up대로 ptr이 NULL인 경우, size가 0인 경우를 예외처리해준다. 그다음 copySize에 현재 블럭의 사이즈를 저장해주고, copySize와 size가 같다면 같은 사이즈의 reallocation을 요청한 것이므로 따로 뭔가를 해줄 필요가 없다. 그래서 그냥 ptr을 바로 리턴해준다.

realloc을 제외한 나머지를 구현한 뒤 mdriver를 돌렸을 때, realloc의 성능이 매우 나빴다. 이것은 제공된 realloc의 새롭게 할당하고 copy한뒤 기존 블럭을 free해주는 방식이 느리다는 것을 의미한다. 나는 느린 원인이 memcpy때문일 것이라고 추정했다.

그래서 memcpy를 사용하지 않고 size를 바꿀 방법을 생각해보았고, 그것은 가능하다면 다음 블럭(리스트의 다음 블럭이 아닌 주소상 다음 블럭을 의미함)과 합치는 것이다. 다음 블럭의 사이즈를 nextSize 변수에 저장한다. 그리고 다음 블럭이 free block이고, 현재 블럭과 다음 블럭의 사이즈 합이 새롭게 할당해야할 사이즈보다 크다면 다음블럭과 합쳐주는 것으로 문제가 해결된다.

따라서 다음 블럭과 합치는 것이 가능한 경우는 overhead와 alignment 조건을 포함하도록 블럭 사이즈를 조정하여 asize에 저장하고, 다음 블럭과 합친뒤, asize만큼 할당하고 리턴하도록 했다.

그리고 불가능한 경우는 기존에 제공된 realloc 함수의 방법대로 malloc함수를 이용하여 새롭게 size만큼 할당한뒤 기존 블럭에서 새로운 블럭으로 내부 값들을 카피하고, 기존 블럭을 free해준뒤 새 블럭의 포인터를 리턴하는 방식을 사용했다.

```
[greenidea@programming2 malloclab-handout]$ make
gcc -Wall -O2 -m32   -c -o mm.o mm.c
mm.c: In function 'mm_init':
mm.c:98:13: warning: assignment makes pointer from integer without a cast [enabled by default]
     if((ptr = extend_heap(CHUNKSIZE/WSIZE) == NULL)) return -1;
             ^
gcc -Wall -O2 -m32 -o mdriver mdriver.o mm.o memlib.o fsecs.o fcyc.o clock.o ftimer.o
[greenidea@programming2 malloclab-handout]$ ./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: cccp-bal.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: expr-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: coalescing-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   98%    5694  0.000183 31132
 1       yes   97%    5848  0.000197 29715
 2       yes   98%    6648  0.000229 29069
 3       yes   99%    5380  0.041285   130
 4       yes   66%   14400  0.000253 56940
 5       yes   93%    4800  0.000300 15984
 6       yes   90%    4800  0.000314 15282
 7       yes   55%   12000  0.000268 44793
 8       yes   51%   24000  0.000495 48495
 9       yes   32%   14401  0.000605 23819
10       yes   45%   14401  0.000698 20638
Total          75%  112372  0.044826  2507

Perf index = 45 (util) + 40 (thru) = 85/100
[greenidea@programming2 malloclab-handout]$
```

그래서 최종적으로 위와 같은 결과를 얻을 수 있었다.