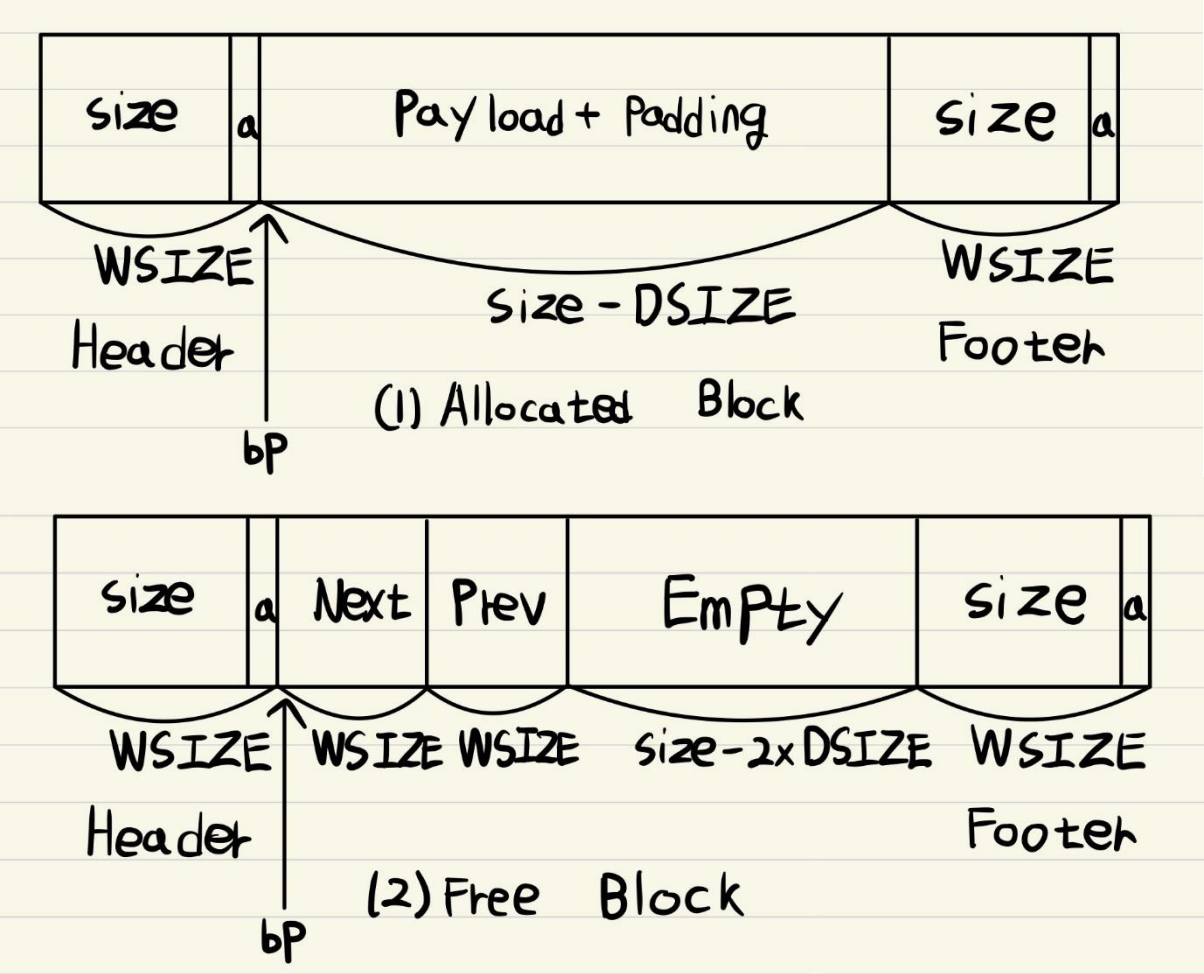
Multicore Programming Project 3

담당 교수 : 박성용

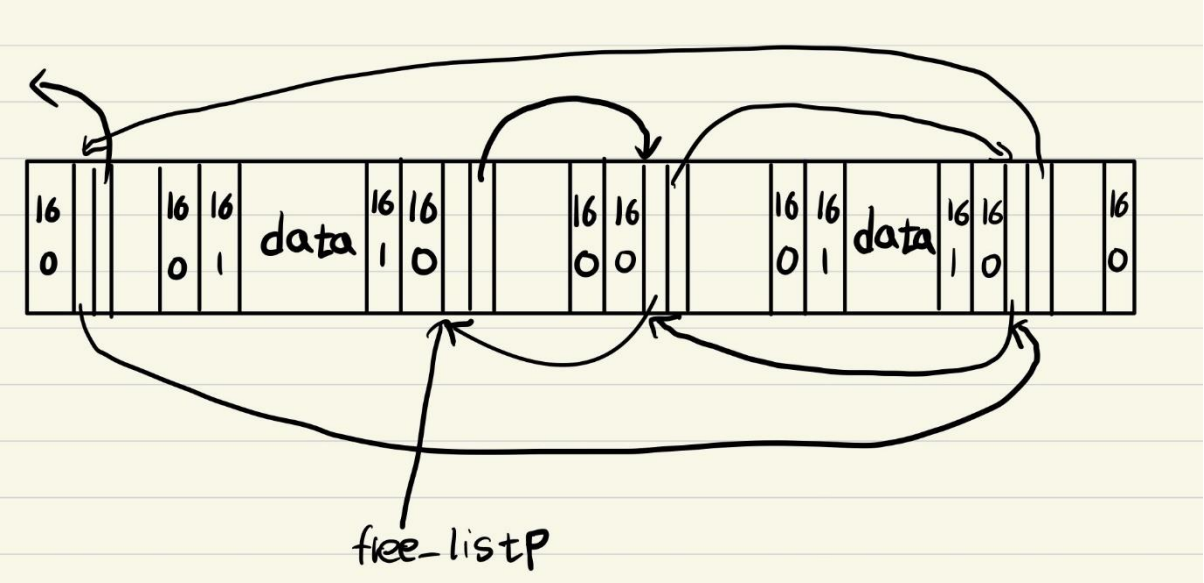
이름 : 엄석훈

학번 : 20181536

이번 프로젝트에서 malloc을 구현하기 위해서는 explicit free lists 방식과 LIFO 방식으로 free list를 관리하였다. 먼저 각 block이 allocated 되었을 때와 free block일 때의 block의 구성은 아래와 같다.



위의 그림처럼 bp를 기준으로 8-byte align이 되도록 구성하고 나머지는 header에 size와 allocated변수를 저장하고 footer에도 동일하게 저장하였다. 다음으로 free block에서는 allocated block과 동일한데 payload와 padding이 들어가는 부분에 첫 두 WSIZE만큼 next free block의 주소와 prev free block의 주소를 저장하였다.



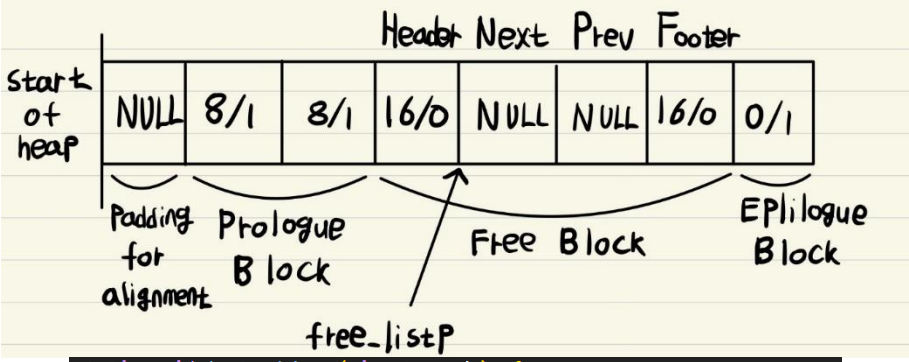
그리고 일반적인 상황에서 위 그림과 같이 block이 구성되는데 free_listp가 free block 리스트를 관리하는 포인터로 해당 포인터에서 출발해서 prev를 타고 링크드리스트 형태로 이전 free block으로 연결되어 있으며 next를 타고 가장 최근에 free된 block 순서로 찾아갈 수 있는 형태로 malloc 구현하였다.

```c
#define NEXT(bp) (*(char **)(bp))
#define PREV(bp) (*(char **)(((char *)(bp) + WSIZE)))

/* Global variables */
static char *free_listp = 0;

/* Function prototypes for internal helper routines */
static void *extend_heap(size_t words);
static void place(void *bp, size_t asize);
static void *find_fit(size_t asize);
static void *coalesce(void *bp);
static void remove_free_list(void *bp);
static void add_free_list(void *bp);
```

먼저 구현을 위해서 기본적인 macro에 추가적으로 next free block와 prev free block을 찾아갈 수 있도록 NEXT와 PREV라는 macro를 추가해주었다. 또한 free block list를 관리하기 위한 포인터로 free_listp를 선언하고 malloc 구현을 위해서 여러가지 helper 함수를 선언해주었다.

```c
int mm_init(void) {
    if((free_listp = mem_sbrk(8 * WSIZE)) == (void *)-1) {
        return -1;
    }
    PUT(free_listp, 0);
    PUT(free_listp + (1 * WSIZE), PACK(DSIZE, 1));
    PUT(free_listp + (2 * WSIZE), PACK(DSIZE, 1));
    PUT(free_listp + (3 * WSIZE), PACK(4 * WSIZE, 0));
    PUT(free_listp + (4 * WSIZE), NULL);
    PUT(free_listp + (5 * WSIZE), NULL);
    PUT(free_listp + (6 * WSIZE), PACK(4 * WSIZE, 0));
    PUT(free_listp + (7 * WSIZE), PACK(0, 1));
    free_listp += (4 * WSIZE);

    if(extend_heap(CHUNKSIZE / WSIZE) == NULL) {
        return -1;
    }
    return 0;
}
```

mm_init은 가장 기본적인 heap구조를 만들어 주는 함수이다. 아래 그림과 같이 가장 초기에 heap을 구성하기 위해서 위와 같이 코드를 작성하였다. Heap의 경계를 나타내기 위해 prologue block과 eplilogue block이 있고 그 사이에 첫번째 free block이 기본적으로 하나 들어있다. 그리고 기본적은 extend_heap함수를 통해서 heap을 기본 사이즈로 늘려주었다.




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

    /* Allocate an even number of words to maintain alignment */
    size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
    //size = ALIGN(size);
    if((long)(bp = mem_sbrk(size)) == -1) {
        return NULL;
    }

    /* Initialize free block header/footer and the epilogue header */
    PUT(HDRP(bp), PACK(size, 0));         /* Free block header */
    PUT(FTRP(bp), PACK(size, 0));         /* Free block footer */
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* New epilogue header */
    add_free_list(bp);

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

extend_heap 함수는 heap의 공간이 부족할 때 words의 개수에 해당하는 크기만큼 늘려주는 함수이다. 따라서 words를 바탕으로 늘려줄 사이즈를 구하고 mem_sbrk를 통해서 heap의 크기를 늘려준다. 그리고 늘어난 heap에서 header와 footer를 넣어주고 eplilogue block을 넣어준다. 그리고 새로 생신 block을 free list에 추가해준다. 그리고 coalesce 함수를 통해 이전 free block과 합쳐준다.

```
void *mm_malloc(size_t size)  {
    size_t asize;      /* Adjusted block size */
    size_t extendsize; /* Amount to extend heap if no fit */
    char *bp;

    /* Ignore spurious requests */
    if(size == 0) {
        return NULL;
    }

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

    /* Search the free list for a fit */
    if((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        return bp;
    }

    /* No fit found. Get more memory and place the block */
    extendsize = MAX(asize, CHUNKSIZE);
    if((bp = extend_heap(extendsize / WSIZE)) == NULL) {
        return NULL;
    }
    place(bp, asize);
    return bp;
}
```

mm_malloc 함수는 교재에 나와있는 함수를 활용했다. 기본적인 흐름은 header와 footer, padding을 모두 고려한 asize를 구한 뒤 find_fit 함수를 통해 공간을 할당받아 place함수를 통해 할당해준다. 이때 공간이 부족해 fine_fit 함수가 null로 return되면 extend_heap함수로 공간을 늘려주고 place함수를 통해 공간을 할당해준다.

```
static void *find_fit(size_t asize) {
    void *bp = free_listp;

    while(bp != NULL) {
        if ((asize <= GET_SIZE(HDRP(bp))))
            return bp;
        bp = PREV(bp);
    }

    return NULL; /* No fit */
}
```

find_fit 함수는 들어온 asize를 바탕으로 free_listp를 순회하며 가장 먼저 발견되는 asize보다 큰 크기를 가진 free block을 발견하면 return 해주는 함수이다. 만약 free block을 발견하지 못하면 NULL을 return 해준다.

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

    remove_free_list(bp);

    if ((csize - asize) >= (2 * DSIZE)) {
        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_free_list(bp);
    }
    else {
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }
}
```

이 함수도 교재의 함수와 매우 유사하다. 먼저 들어온 block을 free block에서 삭제한다. 그리고 free block에서 할당하는 block의 크기를 빼고 남는 공간이 2 * DSIZE보다 커서 다른 block이 들어갈 여유가 있다면 free block을 잘라서 새로운 free block을 free list에 추가하고 아니라면 그냥 할당해준다.

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

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

    add_free_list(bp);
    coalesce(bp);
}
```

다음으로 mm_free함수는 블록의 header와 footer의 정보를 변경해주고 free_list에 추가해주고 coalesce함수를 통해서 앞뒤의 free block과 합쳐준다.

```c
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) {           /* Case 1 */
        return bp;
    }

    else if (prev_alloc && !next_alloc) {     /* Case 2 */
        remove_free_list(bp);
        remove_free_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) {     /* Case 3 */
        remove_free_list(bp);
        remove_free_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 {                                    /* Case 4 */
        remove_free_list(bp);
        remove_free_list(NEXT_BLKP(bp));
        remove_free_list(PREV_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);
    }
    add_free_list(bp);

    return bp;
}
```

coalesce 함수도 교재와 비슷한 흐름으로 작성하였다. 총 4가지 경우로 나눠서 앞

뒤에 free block이 없는 경우는 그냥 return해주고 next block만 free block인 경우 bp와 next bp를 free block list에서 제거하고 header와 footer를 업데이트 하고 bp를 free list에 추가한다. 비슷하게 prev block만 free block인 경우 bp와 prev bp를 free block list에서 제거하고 header와 footer를 업데이트하고 prev bp를 free list에 추가한다. 마지막으로 next block, prev block 모두 free block인 경우 모두 free block list에서 제거하고 footer와 header를 업데이트 하고 prev bp를 free block list에 추가한다.

```
void *mm_realloc(void *ptr, size_t size) {
    size_t oldsize, asize;
    void *newptr;
    int newsize;

    /* If size == 0 then this is just free, and we return NULL. */
    if(size == 0) {
        mm_free(ptr);
        return 0;
    }

    /* If oldptr is NULL, then this is just malloc. */
    if(ptr == NULL) {
        return mm_malloc(size);
    }

    oldsize = GET_SIZE(HDRP(ptr)) - DSIZE;
    if (size <= DSIZE) {
        asize = 2 * DSIZE;
    } else {
        asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);
    }

    if(GET_ALLOC(HDRP(NEXT_BLKP(ptr))) == 0) {
        newsize = GET_SIZE(HDRP(ptr)) + GET_SIZE(HDRP(NEXT_BLKP(ptr))) - asize;
        if(newsize >= 2 * DSIZE) {
            remove_free_list(NEXT_BLKP(ptr));
            PUT(HDRP(ptr), PACK(asize, 1));
            PUT(FTRP(ptr), PACK(asize, 1));
            PUT(HDRP(NEXT_BLKP(ptr)), PACK(newsize, 0));
            PUT(FTRP(NEXT_BLKP(ptr)), PACK(newsize, 0));
            add_free_list(NEXT_BLKP(ptr));
            return ptr;
        }
    }
    newptr = mm_malloc(size);

    /* If realloc() fails the original block is left untouched  */
    if(!newptr) {
        return 0;
    }

    /* Copy the old data. */
    if(size < oldsize) {
        oldsize = size;
    }
    memcpy(newptr, ptr, oldsize);

    /* Free the old block. */
    mm_free(ptr);

    return newptr;
}
```

다음으로 mm_realloc 함수도 교재의 함수를 바탕으로 작성하였다. 먼저 size가 0 이거나 ptr이 null인 경우에 대한 예외처리를 해준다. 그리고 기존 ptr block의 header와 footer를 제거한 사이즈 oldsize로 구해준다. 그리고 asize에는 새로운 size를 align하여 저장한다. 그 상태에서 만약 현재 block의 next block이 free block이고 asize를 할당하고도 공간이 남는다면 현재의 ptr에서 뒤로 공간만 늘리고 header와 footer를 업데이트해서 현재의 ptr을 그대로 사용하도록 해준다. 하지만 공간이 부족하면 새로운 공간을 malloc을 통해 할당받고 데이터를 복사한 뒤 기존의 공간을 free해준다.

```
static void remove_free_list(void *bp) {
    if (bp == free_listp) {
        free_listp = PREV(free_listp);
        return;
    }

    PREV(NEXT(bp)) = PREV(bp);

    if (PREV(bp) != NULL) {
        NEXT(PREV(bp)) = NEXT(bp);
    }
}

static void add_free_list(void *bp) {
    PREV(bp) = free_listp;

    if (free_listp != NULL) {
        NEXT(free_listp) = bp;
    }

    free_listp = bp;
}
```

마지막으로 free_listp를 관리하기 위한 remove_free_list, add_free_list 함수는 위와 같이 작성하였다. remove_free_list 함수의 경우는 먼저 삭제하려는 bp가 free_listp의 첫번째라면 free_listp가 PREV를 가리키도록 하고 return 한다. 아니라면 NEXT의 PREV에 bp의 PREV를 저장시키고 bp의 PREV가 있다면 PREV의 NEXT에 현재 bp의 NEXT를 연결해준다. add_free_list 함수는 더 간단하게 free_listp를 PREV에 저장해주고 기존 free_listp가 있었다면 NEXT에 저장해준다. 그리고 free_listp가 새로 들어온 bp를 가리키도록 해준다. 즉 free_listp는 가장 최근에 들어온 free block을 가리키게 된다.