System Programming Lab report
2021-14284 Taehyun Yang
Lab 3- Malloc Lab

1. General implementation

1. In this solution, memory is allocated using the mm_malloc function. The function takes a size_t size argument, which represents the size of the requested memory block.

2. In ALIGN it it used to calculate the adjusted aligned block size (asize): The requested size is aligned to the nearest multiple of alignment to ensure proper calculation and allocations of memory blocks

3. The size of the block is increased by the header and footer size which is 4 in order to account for storing of data

4. Then Find the index of the segregated free list to search for an appropriate block: The code calculates the log2 of the aligned size (asize) and adjusts it to obtain the index of the segregated free list to search for an appropriate block. This is done using bitwise operations and the predefined arrays b and S. This part of the code is referenced from https://en.wikipedia.org/wiki/Binary_logarithm

5. Then the code searches for a fitting block by usin the fitchek function searches for a free block that can accommodate the requested size. If it finds a suitable block, the function returns the pointer to that block else, it returns null

6. If a suitable block is found, the place function is called to allocate memory in the block. The function splits the block if the remaining size is sufficient to create a new block, updates the header and footer of the block with the allocated status, and removes the block from the free list if necessary.

7.Extend the heap: If no suitable block is found, the heap is extended by the maximum of the aligned size or CHUNKSIZE. Keep in mind that instead of what the textbook shows, i automatically selected the byte as 512 to reduce performance speed. the rest is executed using the extend_heap function, which calls mem_sbrk to extend the heap by the requested size which is initialized afterwards to new free block's header and footer, and calls the coalesce function to merge it with any adjacent free blocks.

8. mm_malloc function allocates memory in the most suitable block available in the segregated free lists, or extends the heap if necessary. 2021-14284 Taehyun Yang

2. Macros and helper functions

```c
/* single word (4) or double word (8) alignment */
#define ALIGNMENT 8

/* rounds up to the nearest multiple of ALIGNMENT */
#define ALIGN(size) (((size) + (ALIGNMENT - 1)) & ~0x7)

#define WSIZE 4
#define DSIZE 8
#define CHUNKSIZE 512
#define Byte_LEN 15

#define MAX(x, y) ((x) > (y) ? (x) : (y))
#define MIN(x, y) ((x) < (y) ? (x) : (y))

/* Pack a size and allocated bit into a word */
#define PACK(size, alloc) ((size) | (alloc))

/* Read and write a word at address p */
#define GET(p) (*(unsigned int *)(p))
#define PUT(p, val) (*(unsigned int *)(p) = (unsigned int)val)

/* Read the size and allocated fields from address p */
#define GET_SIZE(p) (GET(p) & ~7)
#define GET_ALLOC(p) (GET(p) & 1)

/* Given block ptr bp, compute address of its header and footer */
#define HDRP(bp) ((char *)(bp)-WSIZE)
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

/* Given block ptr bp, compute address of next and previous blocks */
#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)))
#define PREV_BLKP(bp) ((char *)(bp)-GET_SIZE(((char *)(bp)-DSIZE)))

// get the pred and succ pointers
#define PRED_PTR(bp) ((char *)(bp) + WSIZE)
#define SUCC_PTR(bp) ((char *)bp)

// get the pred and succ blocklist
#define PRED_BLKP(bp) (GET(PRED_PTR(bp)))
#define SUCC_BLKP(bp) (GET(SUCC_PTR(bp)))

// utility functions
static void *extend_heap(size_t size);
static void *coalesce(void *bp);
static void *place(char *bp, size_t size);
static void *fitchek(size_t size, int seg_idx);

const unsigned int b[] = {0x2, 0xC, 0xF0, 0xFF00, 0xFFFF0000}; // list used to find log2 of an nbit int
const unsigned int S[] = {1, 2, 4, 8, 16};                     // another list
char *start_ptr;
char *heap_list;
```

Generally, all the macros are macros I received help from the textbook. However, there are some utility functions that I have added such as fitcheck which checks whether the list is in fit for initialization, place, and some unsigned int arrays used for an extra function I have added along with pred and succ pointers and block sizes to make allocation functions implementations abit easier.

3. Main implementation brief explanation

```
int mm_init(void)
{

    if ((heap_list = mem_sbrk((Byte_LEN + 3) * WSIZE)) == (void *)-1)
        return -1;

    for (int i = 0; i < Byte_LEN; ++i)
        PUT(heap_list + i * WSIZE, NULL);
    // alignment padding
    PUT(heap_list + (Byte_LEN)*WSIZE, PACK(DSIZE, 1));       // prologue header
    PUT(heap_list + (Byte_LEN + 1) * WSIZE, PACK(DSIZE, 1)); // prologue footer
    PUT(heap_list + (Byte_LEN + 2) * WSIZE, PACK(0, 1));     // epilogue header

    start_ptr = heap_list;
    heap_list += (Byte_LEN)*WSIZE;
    // extend the empty heap with a free block of chunksize bytes
    if (extend_heap(CHUNKSIZE) == NULL)
        return -1;

    return 0;
}
```

Almost exactly same as the textbook with the initialization of prologue header, footer, and epilogue header along with alignment padding.

```
void *mm_malloc(size_t size)    // mm_malloc allocates a block from the free list.
{                               // mostly copied from the textbook
    size_t asize = ALIGN(size); /* Adjusted aligned block size */
    size_t extendsize;          /* Amount to extend heap if no fit */
    char *bp;

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

    register unsigned int r = 0; // faster method to search the free lists by finding the log 2 of the list
    for (int i = 4; i >= 0; i--)
    {
        if (asize & b[i])
        {
            asize >>= S[i];
            r |= S[i];
        }
    }
    int size_index = (int)r - 4;
    if (size_index < 0)
        size_index = 0;
    if (size_index >= 15)
        size_index = 14;

    asize = DSIZE * ((size + (DSIZE) + (DSIZE - 1)) / DSIZE);

    if ((bp = fitchek(asize, size_index)) != NULL) // search the free list for a fit
        return place(bp, asize);

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

    bp = place(bp, asize);
    return bp;
}
```

This mm_malloc function is responsible for allocating memory from the free list of a memory management system. It takes a size_t size argument representing the requested size of

the memory block. The function is mostly copied from the textbook and has been slightly modified for this implementation.

```
static void free_block(char *bp) // empties a block
{

    if (PRED_BLKP(bp) && !SUCC_BLKP(bp)) // if the block is the last in the list
    {

        PUT(SUCC_PTR(PRED_BLKP(bp)), NULL);
    }

    if (SUCC_BLKP(bp) && PRED_BLKP(bp)) // if the block is in the middle of the list

    {
        PUT(SUCC_PTR(PRED_BLKP(bp)), SUCC_BLKP(bp));
        PUT(PRED_PTR(SUCC_BLKP(bp)), PRED_BLKP(bp));
    }
    PUT(SUCC_PTR(bp), NULL); // if the block is the first in the list
    PUT(PRED_PTR(bp), NULL);
}
```

The free_block function is a static function that removes a block from the free list, essentially marking it as empty or no longer free. It takes a single argument, char *bp, which is a pointer to the block that will be removed from the free list. If the block is the last in the list: If the block has a predecessor (PRED_BLKP(bp)) and no successor (!SUCC_BLKP(bp)), the function updates the successor pointer of the predecessor block to NULL, essentially removing the current block from the list. If the block is in the middle of the list: If the block has both a predecessor (PRED_BLKP(bp)) and a successor (SUCC_BLKP(bp)), the function updates the successor pointer of the predecessor block to point to the successor block, and updates the predecessor pointer of the successor block to point to the predecessor block. This effectively removes the current block from the list. If the block is the first in the list: The function sets both the successor and predecessor poi nters of the block to NULL.

```c
static void insert_block(char *bp)
{
    int seg_index = GET_SIZE(HDRP(bp));
    register unsigned int r = 0; // result of log2(v) will go here
    for (int i = 4; i >= 0; i--) // unroll for speed...
    {
        if (seg_index & b[i])
        {
            seg_index >>= S[i];
            r |= S[i];
        }
    }
    int x = (int)r - 4;
    if (x < 0)
        x = 0;
    if (x >= 15)
        x = 14;

    char *start = start_ptr + x * WSIZE; // use the index found to quickly find the starting pointer
    void *succ = start;

    while (SUCC_BLKP(succ))
    {
        succ = (char *)SUCC_BLKP(succ);
        if ((unsigned int)succ >= (unsigned int)bp)
        {

            char *tmp = succ;
            succ = (char *)PRED_BLKP(succ);
            PUT(SUCC_PTR(succ), bp);
            PUT(PRED_PTR(bp), succ);
            PUT(SUCC_PTR(bp), tmp);
            PUT(PRED_PTR(tmp), bp);

            return;
        }
    }

    PUT(SUCC_PTR(succ), bp);
    PUT(PRED_PTR(bp), succ);
    PUT(SUCC_PTR(bp), NULL);
}
```

The insert_block function is a static function that inserts a block into the appropriate free list based on its size. It takes a single argument, char *bp, which is a pointer to the block that will be inserted into the free list. The function first calculates the index of the appropriate free list by using bitwise operations and log2 calculation. The size of the block header (HDRP(bp)) is used for this calculation. The index is clamped to be within the bounds of 0 and 14.

```c
void *mm_realloc(void *ptr, size_t size)
{

    if (ptr == NULL)
        return mm_malloc(size);

    else if (size == 0)
    {
        mm_free(ptr);
        return NULL;
    }
    size_t asize = DSIZE * ((size + (DSIZE) + (DSIZE - 1)) / DSIZE);
    size_t old_size = GET_SIZE(HDRP(ptr));
    void *newptr;

    if (old_size == asize)
        return ptr;

    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(ptr)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(ptr)));
    size_t next_size = GET_SIZE(HDRP(NEXT_BLKP(ptr)));
    char *next_bp = NEXT_BLKP(ptr);
    size_t total_size = old_size;

    if (prev_alloc && !next_alloc && (old_size + next_size >= asize))
    {
        total_size += next_size;
        free_block(next_bp);
        PUT(HDRP(ptr), PACK(total_size, 1));
        PUT(FTRP(ptr), PACK(total_size, 1));
        place(ptr, total_size);
    }
    else if (!next_size && asize >= old_size)
    {
        size_t extend_size = asize - old_size;
        if ((long)(mem_sbrk(extend_size)) == -1)
            return NULL;

        PUT(HDRP(ptr), PACK(total_size + extend_size, 1));
        PUT(FTRP(ptr), PACK(total_size + extend_size, 1));
        PUT(HDRP(NEXT_BLKP(ptr)), PACK(0, 1));
        place(ptr, asize);
    }
    else
    {
        newptr = mm_malloc(asize);
        if (newptr == NULL)
            return NULL;
        memcpy(newptr, ptr, MIN(old_size, size));
        mm_free(ptr);
        return newptr;
    }
    return ptr;
}
```

The mm_realloc function is responsible for resizing an existing memory block in a dynamic memory allocator. It takes two arguments: a pointer void *ptr to the memory block that needs to be resized, and the new size size_t size in bytes. The function first checks if ptr is NULL, in which case it behaves like mm_malloc and allocates a new block of memory with the given size. If size is 0, it behaves like mm_free, deallocating the memory block pointed to by ptr and returning NULL.

‘

4. What I have learned

First I have learned more about dynamic memory allocation. I learned how to understand the fundamentals of allocating and deallocating memory at runtime, including the use of functions like malloc, free, and realloc.

I also got to learn more about memory alignment. I have learned the importance of aligning memory for efficient access and how to calculate aligned block sizes.

Memory block headers and footers are also another thing I heav learned. I Gained insight into how memory blocks are organized in memory, including the use of headers and footers to store metadata about the block (size and allocation status).

Ultimately the textbook and the hints given in class was a big help and I hope to learn more about memory allocation