System Programming - Malloc Lab Report

2019 Spring, 2017-18570 Sungchan Yi

After looking at the code from the book, I decided to implement a segregated list. I had to define a bunch of macros, I mostly got the names from the book. Most of the explanation is in the comments.

```c
#define ALIGNMENT 8
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)

// Macros from code in textbook + My Macros
#define SIZEW 4
#define SIZEQ 8
#define INITCHUNKSIZE (1 << 6)
#define CHUNKSIZE (1 << 12)
#define LIST_N 20

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

// Read from ptr
#define GET(ptr) (*(unsigned int *)(ptr))
// write val to p
#define PUT(p, val) (*(unsigned int *)(p) = (val))

// size, alloc bit, tag bit at ptr
#define GET_SIZE(ptr) (GET(ptr) & ~0x7)
#define GET_ALLOC(ptr) (GET(ptr) & 1)

// Address of block header / footer
#define HDRP(ptr) ((char *)(ptr) - SIZEW)
#define FTRP(ptr) ((char *)(ptr) + GET_SIZE(HDRP(ptr)) - SIZEQ)

// next block, prev block
#define NEXT_BLKP(ptr) ((char *)(ptr) + GET_SIZE((char *)(ptr) - SIZEW))
#define PREV_BLKP(ptr) ((char *)(ptr) - GET_SIZE((char *)(ptr) - SIZEQ))

// Next, prev free block entry
#define PRED_PTR(ptr) ((char *)(ptr))
#define SUCC_PTR(ptr) ((char *)(ptr) + SIZEW)

// Next, prev of free block on seg list
#define PRED(ptr) (*(char **)(ptr))
#define SUCC(ptr) (*(char **)(SUCC_PTR(ptr)))

// Set Pointer
```

```
#define SET_PTR(p, ptr) (*(unsigned int *)(p) = (unsigned int)(ptr))
```

There were so many macros, but because macros are usually a great source of bugs, I tried not to use them very much... I decided to have 20 range of sizes for the segregated list, so I used a double pointer.

```
void *l0 = 0, *l1 = 0, *l2 = 0, *l3 = 0, *l4 = 0, *l5 = 0, *l6 = 0, *l7 = 0, *l8 = 0, *l9 = 0, *l10 = 0, *l11 = 0, *l12 = 0, *l13 = 0, *l14 = 0, *l15 = 0, *l16 = 0, *l17 = 0, *l18 = 0, *l19 = 0;
void **seg_list = &l19; // Now works as a pointer to a pointer array
```

`seg_list` will point to the list. `*l`n can be referenced by `seg_list[n]`.
Initialize the procedure, allocating the heap, setting an initial block etc.

```
int mm_init(void) {
    int i = 0;
    char *heap_st; // track the start of the heap

    // Initialize segregated list
    while(i < LIST_N) seg_list[i++] = NULL;

    // Allocate memory on heap
    if((long) (heap_st = mem_sbrk(4 * SIZEW)) == -1) return -1;

    // Initial block
    PUT(heap_st , 0);
    PUT(heap_st + (1 * SIZEW), PACK(SIZEQ , 1));
    PUT(heap_st + (2 * SIZEW), PACK(SIZEQ , 1));
    PUT(heap_st + (3 * SIZEW), PACK(0, 1));

    // extend heap
    if(!extend_heap(INITCHUNKSIZE)) return -1;
    return 0;
}
```

And here was the fun part, extending the heap and inserting, deleting elements to the segregated list. For extending the heap, I needed to consider the alignment requirements. After extending the heap, consider it as a free block and insert it into the list and coalesce it.
The insertion and deletion procedure was quite similar to the normal insertion/deletion procedures for linked lists. Furthermore, the segregation list range was decided by 2^k. To decide which list the block should go into, call a subroutine `select_list`.

```
static void *extend_heap(size_t size) {
    void *ptr;
```

```
    size_t nsize = ALIGN(size); // alignment of new size
    // In case it fails
    if((ptr = mem_sbrk(nsize)) == (void *) -1) return NULL;

    PUT(HDRP(ptr), PACK(nsize, 0));
    PUT(FTRP(ptr), PACK(nsize, 0)); // Update header and footer
    PUT(HDRP(NEXT_BLKP(ptr)), PACK(0, 1));
    insert(ptr, nsize); // insert newly extended area to the list
    return coalesce(ptr); // coalesce if necessary
}

// Select from segregated list
static int select_list(int size) {
    int idx = 0;
    while(idx < LIST_N - 1 && size > 1) {
        size >>= 1;
        idx++;
    }
    return idx;
}

// Insert into segregated list
static void insert(void *ptr, size_t size) {
    void *search_p = ptr;
    void *insert_p = NULL;
    int idx = select_list(size); // Select from segregated list

    // Search for a place to fit
    search_p = seg_list[idx];
    while(search_p && (size > GET_SIZE(HDRP(search_p)))) {
        insert_p = search_p;
        search_p = PRED(search_p);
    }

    // Set next, prev
    if(search_p) {
        SET_PTR(PRED_PTR(ptr), search_p);
        SET_PTR(SUCC_PTR(search_p), ptr);
        if(insert_p) {
            SET_PTR(SUCC_PTR(ptr), insert_p);
            SET_PTR(PRED_PTR(insert_p), ptr);
        } else {
            SET_PTR(SUCC_PTR(ptr), NULL);
            seg_list[idx] = ptr;
        }
    } else {
```

```
        SET_PTR(PRED_PTR(ptr), NULL);
        if(insert_p) {
            SET_PTR(SUCC_PTR(ptr), insert_p);
            SET_PTR(PRED_PTR(insert_p), ptr);
        } else {
            SET_PTR(SUCC_PTR(ptr), NULL);
            seg_list[idx] = ptr;
        }
    }

    return;
}

// Delete from segregated list
static void delete(void *ptr) {
    size_t size = GET_SIZE(HDRP(ptr));
    int idx = select_list(size); // Select from seg list

    // Deletion works just like linked lists
    if(PRED(ptr)) {
        if(SUCC(ptr)) {
            // set succ of pred to the succ of current
            SET_PTR(SUCC_PTR(PRED(ptr)), SUCC(ptr));
            // set pred of succ to the pred of current
            SET_PTR(PRED_PTR(SUCC(ptr)), PRED(ptr));
        } else {
            SET_PTR(SUCC_PTR(PRED(ptr)), NULL);
            seg_list[idx] = PRED(ptr);
        }
    } else {
        if(SUCC(ptr)) SET_PTR(PRED_PTR(SUCC(ptr)), NULL);
        else seg_list[idx] = NULL;
    }
    return;
}
```

For coalescing, identify all 4 cases and handle them separately.

```
static void *coalesce(void *ptr) {
    // allocation status of prev, next
    size_t prev = GET_ALLOC(HDRP(PREV_BLKP(ptr)));
    size_t next = GET_ALLOC(HDRP(NEXT_BLKP(ptr)));
    size_t size = GET_SIZE(HDRP(ptr));

    if(prev && next) return ptr; // Case 1
    delete(ptr);
```

```
    if(prev && !next) { // Case 2
        delete(NEXT_BLKP(ptr)); // delete next block
        size += GET_SIZE(HDRP(NEXT_BLKP(ptr))); // add size
        PUT(HDRP(ptr), PACK(size, 0)); // update header
        PUT(FTRP(ptr), PACK(size, 0));
    } else if(!prev && next) { // Case 3
        delete(PREV_BLKP(ptr)); // delete prev block
        size += GET_SIZE(HDRP(PREV_BLKP(ptr))); // add size
        PUT(FTRP(ptr), PACK(size, 0)); // update footer
        PUT(HDRP(PREV_BLKP(ptr)), PACK(size, 0)); // update header
        ptr = PREV_BLKP(ptr); // set to prev block
    } else { // Case 4
        delete(PREV_BLKP(ptr)); // delete prev block
        delete(NEXT_BLKP(ptr)); // delete next block
        size += GET_SIZE(HDRP(NEXT_BLKP(ptr)));
        size += GET_SIZE(HDRP(PREV_BLKP(ptr))); // add size
        PUT(HDRP(PREV_BLKP(ptr)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(ptr)), PACK(size, 0));
        ptr = PREV_BLKP(ptr); // set to prev block
    }
    insert(ptr, size); // insert newly coalesced block
    return ptr;
}
```

And the important part was actually implementing malloc. To find a free block in the segregated list, call a subroutine `search_block`.

```
static void *search_block(void* ptr, int asize) {
    size_t ssize = asize; // search size
    int idx = 0;
    for(; idx < LIST_N; ++idx, ssize >>= 1) {
        if(idx == LIST_N - 1 || ((ssize <= 1) && seg_list[idx])) {
            ptr = seg_list[idx];
            // ignore small blocks
            while(ptr && asize > GET_SIZE(HDRP(ptr))) {
                ptr = PRED(ptr);
            }
            if(ptr) break;
        }
    }
    return ptr;
}

void *mm_malloc(size_t size) {
    size_t asize; // adjust size
    size_t ext_size; // extended size
```

```
    void *ptr = NULL;

    // size 0
    if(!size) return NULL;
    if(size <= SIZEQ) asize = 2 * SIZEQ;
    else asize = ALIGN(size + SIZEQ);

    ptr = search_block(ptr, asize); // Search for free block in seglist

    // if not found, extend heap
    if(!ptr) {
        ext_size = asize > CHUNKSIZE ? asize : CHUNKSIZE;
        if(!(ptr = extend_heap(ext_size))) return NULL;
    }

    // Place block, split if necessary
    // splitting will be taken care of by the place function
    ptr = place(ptr, asize);
    return ptr;
}
```

Free was quite simple at first thought, but I had a hard time debugging because I forgot to put `HDRP(ptr)` in the second line...

```
void mm_free(void *ptr) {
    size_t size = GET_SIZE(HDRP(ptr)); // HDRP(ptr) !!!
    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0)); // Update header / footer
    insert(ptr, size); // insert to seg_list
    coalesce(ptr); // coalesce free blocks
    return;
}
```

Now for the reallocation, use the implemented `mm_malloc` and `mm_free`. Simply follow the C standards.

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

    if(!size) { // realloc(ptr, 0) is equal to free
        mm_free(ptr);
        return 0;
    }
    size += 1 << 7; // add size for future reallocation
```

```c
    if(!ptr) return mm_malloc(size); // realloc(NULL, size) is malloc(size)
    newptr = mm_malloc(size);
    if(!newptr) return 0; // if fail, return 0

    oldsize = size < GET_SIZE(HDRP(ptr)) ? size : GET_SIZE(HDRP(ptr));
    memcpy(newptr, ptr, oldsize); // copy old data
    mm_free(ptr); // free the old block
    return newptr;
}
```

This lab was particularly hard due to so many macros that I had to use. I even had to waste hours due to a mistake in the macro. But after writing the code, and when I reviewed it, it seems to me that macros are really necessary. The code looks much cleaner thanks to the macro. I should really be careful about them. Moreover, simulating malloc was a great experience, manipulating data at the kernel level.[1] I kept having the urge to use malloc whenever I tried to create a new node, but since it's already free memory, it didn't matter!

[1] I'm not sure if I'm allowed to say this...