Prof. Heonyoung Yeom

System Programming (001)

Lab #3 / Malloc Lab

: Writing a Dynamic Storage Allocator

Report

Dept. of Computer Science and Engineering
2021-13194
Jaejun Ko

2023.4.28

Lab #3 Report

Malloc Lab : Writing a Dynamic Storage Allocator

Introduction

In this lab session, we will implement a dynamic storage allocator program, `mm.c`, which contains `malloc`, `free` and `realloc` routines. We expect to be familiar with memory allocation and managing heap memory.

Specification

The dynamic storage allocator will consist of the following four functions, which are declared in `mm.h` and defined in `mm.c`.

```
int   mm_init(void);
void *mm_malloc(size_t size);
void  mm_free(void *ptr);
void *mm_realloc(void *ptr, size_t size);
```

The `mm.c` file we have given you implements the simplest but still functionally correct malloc package that we could think of. Using this as a starting place, modify these functions (and possibly define other private `static` functions) so that they obey the following semantics:

- `mm_init`: Before calling `mm_malloc`, `mm_realloc`, or `mm_free`, the application program (i.e., the trace-driven driver program that you will use to evaluate your implementation) calls `mm_init` to perform any necessary initializations, such as allocating the initial heap area. The return value should be -1 if there was a problem in performing the initialization, 0 otherwise.
- `mm_malloc`: The `mm_malloc` routine returns a pointer to an allocated block payload of at least size bytes. The entire allocated block should lie within the heap region and should not overlap with any other allocated chunk. We will compare your implementation to the version of malloc supplied in the standard C library (`libc`). Since the `libc` malloc always returns payload pointers that are aligned to 8 bytes, your malloc implementation should do likewise and always return 8-byte aligned pointers.
- `mm_free`: The `mm_free` routine frees the block pointed to by `ptr`. It returns nothing. This routine is only guaranteed to work when the passed pointer (`ptr`) was returned by an earlier call to `mm_malloc` or `mm_realloc` and has not yet been freed.
- `mm_realloc`: The `mm_realloc` routine returns a pointer to an allocated region of at least size bytes with the following constraints.
  - if `ptr` is `NULL`, the call is equivalent to `mm_malloc(size);`
  - if `size` is equal to zero, the call is equivalent to `mm_free(ptr);`
  - if `ptr` is not `NULL`, it must have been returned by an earlier call to `mm_malloc` or `mm_realloc`. The call to `mm_realloc` changes the size of the memory block pointed to by `ptr` (the old block) to

`size` bytes and returns the address of the new block. Notice that the address of the new block might be the same as the old block, or it might be different, depending on your implementation, the amount of internal fragmentation in the old block, and the size of the `realloc` request.

The contents of the new block are the same as those of the old `ptr` block, up to the minimum of the old and new sizes. Everything else is uninitialized. For example, if the old block is 8 bytes and the new block is 12 bytes, then the first 8 bytes of the new block are identical to the first 8 bytes of the old block and the last 4 bytes are uninitialized. Similarly, if the old block is 8 bytes and the new block is 4 bytes, then the contents of the new block are identical to the first 4 bytes of the old block.

These semantics match the semantics of the corresponding `libc malloc`, `realloc`, and `free` routines.

Implementation

0. Macros, Constants, Static Functions

The followings are the macros, constants, and static functions to be used in the implementation of the dynamic memory allocator. Most of the macros, constants, and static functions are in the textbook pp.857-961. The added ones are:

- `SEG_LIST_COUNT`
- `MIN_BLK_SIZE`
- `PUT_PTR(node_ptr, ptr)`
- `PREV_NODEP(node_ptr)`
- `NEXT_NODEP(node_ptr)`
- `PREV_FREE_BLKP(ptr)`
- `NEXT_FREE_BLKP(ptr)`
- `SEG_LIST(list_ptr, index)`
- `static void insert_to_list(void *ptr, size_t size);`
- `static void remove_from_list(void *ptr, size_t size);`
- `static void *seg_list_ptr = NULL;`

```
#define ALIGNMENT               8
#define ALIGN(size)             (((size) + (ALIGNMENT-1)) & ~0x7)
#define WSIZE                   4
#define DSIZE                   8
#define CHUNKSIZE               (1<<12)
#define SEG_LIST_COUNT          12
#define MIN_BLK_SIZE            (2 * DSIZE)

#define MAX(x, y)               ((x) > (y) ? (x) : (y))
#define PACK(size, alloc)       ((size) | (alloc))
#define GET(ptr)                (*(unsigned int *)(ptr))
#define PUT(ptr, val)           (*(unsigned int *)(ptr) = (val))
#define GET_SIZE(ptr)           (GET(ptr) & ~0x7)
#define GET_ALLOC(ptr)          (GET(ptr) & 0x1)
#define HDRP(blk_ptr)           ((char *)(blk_ptr) - WSIZE)
#define FTRP(blk_ptr)           ((char *)(blk_ptr) + GET_SIZE(HDRP(blk_ptr)) -
DSIZE)
#define PREV_BLKP(blk_ptr)      ((char *)(blk_ptr) - GET_SIZE(((char *)(blk_ptr)
- DSIZE)))
#define NEXT_BLKP(blk_ptr)      ((char *)(blk_ptr) + GET_SIZE(((char *)(blk_ptr)
- WSIZE)))
#define PUT_PTR(node_ptr, ptr)  (*(unsigned int *)(node_ptr) = (unsigned
int)(ptr))
#define PREV_NODEP(node_ptr)    ((char *)(node_ptr) + WSIZE)
```

```
#define NEXT_NODEP(node_ptr)     ((char *)(node_ptr))
#define PREV_FREE_BLKP(ptr)      (*(char **)((char *)(ptr) + WSIZE))
#define NEXT_FREE_BLKP(ptr)      (*(char **)(ptr))
#define SEG_LIST(list_ptr, index)    (*((char **)list_ptr + index))
static void *extend_heap(size_t words);
static void *coalesce(void *ptr);
static void *find_fit(size_t size);
static void *place(void *ptr, size_t size);
static void insert_to_list(void *ptr, size_t size);
static void remove_from_list(void *ptr);
int mm_check(void);
static void *seg_list_ptr = NULL;
static char *heap_ptr = NULL;
```

In this implementation, the segregated free list contains doubly linked lists whose elements are blocks with sizes from 2^(index) to 2^(index+1)-1. By maintaining each list is sorted, optimal free blocks can be found readily and quickly. We can retrieve the head node of the list in the list by using the macro `SEG_LIST` and the static pointer `seg_list_ptr`. It will be assigned in `mm_init` to be a pointer for the list. As the blocks are free, we can use block spaces to store information about the next node and previous node of the free block in the list. By using `PUT_PTR`, `PREV_NODEP`, and `NEXT_NODEP`, the pointer for the next and previous blocks can be stored. Also, `PREV_FREE_BLKP` and `NEXT_FREE_BLKP` return pointers for the previous free block and the next free block in the list, respectively. This list will be managed by two static functions, `insert_to_list`, and `remove_from_list`.

The followings are the implementation of two static functions. `insert_to_list` checks if the block is the head or the tail in the list and inserts the block in the list in a sorted way. On the other hand, also checks if the block is the head or the tail in the list and removes it from the list properly.

```
static void insert_to_list(void *blk_ptr, size_t size)
{
    int size_copy = size;
    int list_index;
    for (list_index = 0; (list_index < SEG_LIST_COUNT - 1) && (size_copy > 1);
list_index++)
        size_copy >>= 1;
    void *next_ptr = SEG_LIST(seg_list_ptr, list_index);
    void *prev_ptr = NULL;
    while ((next_ptr != NULL) && (size > GET_SIZE(HDRP(next_ptr)))) {
        prev_ptr = next_ptr;
        next_ptr = NEXT_FREE_BLKP(next_ptr);
    }
    if (next_ptr != NULL) {
        if (prev_ptr != NULL) {
            PUT_PTR(PREV_NODEP(blk_ptr), prev_ptr);
            PUT_PTR(NEXT_NODEP(prev_ptr), blk_ptr);
            PUT_PTR(PREV_NODEP(next_ptr), blk_ptr);
            PUT_PTR(NEXT_NODEP(blk_ptr), next_ptr);
        }
        else {
            PUT_PTR(PREV_NODEP(blk_ptr), NULL);
            PUT_PTR(NEXT_NODEP(blk_ptr), next_ptr);
            PUT_PTR(PREV_NODEP(next_ptr), blk_ptr);
            SEG_LIST(seg_list_ptr, list_index) = blk_ptr;
        }
    }
    else {
        if (prev_ptr != NULL){
            PUT_PTR(PREV_NODEP(blk_ptr), prev_ptr);
            PUT_PTR(NEXT_NODEP(blk_ptr), NULL);
            PUT_PTR(NEXT_NODEP(prev_ptr), blk_ptr);
        }
        else {
            SEG_LIST(seg_list_ptr, list_index) = blk_ptr;
            PUT_PTR(PREV_NODEP(blk_ptr), NULL);
            PUT_PTR(NEXT_NODEP(blk_ptr), NULL);
        }
    }
}
```

```
static void remove_from_list(void *blk_ptr)
{
    size_t size = GET_SIZE(HDRP(blk_ptr));
    if ((PREV_FREE_BLKP(blk_ptr) != NULL)) {
        PUT_PTR(NEXT_NODEP(PREV_FREE_BLKP(blk_ptr)), NEXT_FREE_BLKP(blk_ptr));
        if (NEXT_FREE_BLKP(blk_ptr) != NULL)
            PUT_PTR(PREV_NODEP(NEXT_FREE_BLKP(blk_ptr)),PREV_FREE_BLKP(blk_ptr));
    }
    else {
        int list_index;
        for (list_index = 0; (list_index < SEG_LIST_COUNT - 1) && (size > 1);
list_index++)
            size >>= 1;
        SEG_LIST(seg_list_ptr, list_index) = NEXT_FREE_BLKP(blk_ptr);
        if (SEG_LIST(seg_list_ptr, list_index) != NULL)
            PUT_PTR(PREV_NODEP(SEG_LIST(seg_list_ptr, list_index)), NULL);

    }
}
```

1. mm_init(void)

In the mm_init function, the segregated free list and heap are initialized. It assigns a pointer for memory whose size is SEG_LIST_COUNT to seg_list_ptr. Also, it makes heap_ptr point between the prologue footer and the epilogue footer. And then extend the heap with a free block of CHUNKSIZE bytes.

```
int mm_init(void)
{
    /* Allocate memory for segregated list and initialize it */
    seg_list_ptr = mem_sbrk(SEG_LIST_COUNT * WSIZE);
    for(int list_index = 0; list_index < SEG_LIST_COUNT; list_index++)
        SEG_LIST(seg_list_ptr, list_index) = NULL;

    /* Allocate memory for heap and initialize it */
    if((heap_ptr = mem_sbrk(4 * WSIZE)) == (void *)-1)
        return -1;
    PUT(heap_ptr, 0);                           /* Alignment padding */
    PUT(heap_ptr + (1*WSIZE), PACK(DSIZE, 1));  /* Prologue header */
    PUT(heap_ptr + (2*WSIZE), PACK(DSIZE, 1));  /* Prologue footer */
    PUT(heap_ptr + (3*WSIZE), PACK(0, 1));      /* Epilogue footer */
    heap_ptr += (2 * WSIZE);

    /* Extend the empty heap with a free block of CHUNKSIZE bytes */
    if(extend_heap(CHUNKSIZE/WSIZE) == NULL)
        return -1;
    return 0;
}
```

The following function is void* extend_heap(size_t words). It extends the heap with a free block containing words words. For the case of that the previous block was a free block, executes coalesce.

```
static void* extend_heap(size_t words)
{
    char *blk_ptr;
    size_t size;
    size = (words % 2) ? (words + 1) * WSIZE : words * WSIZE;
    if ((long)(blk_ptr = mem_sbrk(size)) == -1)
        return NULL;
    PUT(HDRP(blk_ptr), PACK(size, 0));
    PUT(FTRP(blk_ptr), PACK(size, 0));
    PUT(HDRP(NEXT_BLKP(blk_ptr)), PACK(0, 1));
    insert_to_list(blk_ptr, size);
    return coalesce(blk_ptr);
}
```

void* coalesce(void *blk_ptr) function coalesces the adjacent free blocks. The implementations are written below. It considers 4 cases; (1) if the previous and next blocks are allocated, just return blk_ptr; (2) if only the next block is free, then just coalesce both blocks and return `blk_ptr`; (3) if only the previous one is free, then coalesce both blocks, move pointer to the previous block pointer, and return it; (4) if both sides are free, coalesce three blocks and return the previous block pointer.

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

    if (prev_alloc && next_alloc)
        return blk_ptr;
    else if (prev_alloc && !next_alloc) {
        remove_from_list(blk_ptr);
        remove_from_list(NEXT_BLKP(blk_ptr));
        size += GET_SIZE(HDRP(NEXT_BLKP(blk_ptr)));
        PUT(HDRP(blk_ptr), PACK(size, 0));
        PUT(FTRP(blk_ptr), PACK(size, 0));
    }
    else if (!prev_alloc && next_alloc) {
        remove_from_list(PREV_BLKP(blk_ptr));
        remove_from_list(blk_ptr);
        size += GET_SIZE(HDRP(PREV_BLKP(blk_ptr)));
        PUT(FTRP(blk_ptr), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(blk_ptr)), PACK(size, 0));
        blk_ptr = PREV_BLKP(blk_ptr);
    }
    else {
        remove_from_list(PREV_BLKP(blk_ptr));
        remove_from_list(blk_ptr);
        remove_from_list(NEXT_BLKP(blk_ptr));
        size += GET_SIZE(HDRP(PREV_BLKP(blk_ptr)))
                + GET_SIZE(FTRP(NEXT_BLKP(blk_ptr)));
        PUT(HDRP(PREV_BLKP(blk_ptr)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(blk_ptr)), PACK(size, 0));
        blk_ptr = PREV_BLKP(blk_ptr);
    }

    insert_to_list(blk_ptr, size);
    return blk_ptr;
```

2. **`mm_malloc(size_t size)`**

In the `mm_malloc` function, it adjusts the size to be aligned and finds a free block with the adjusted size that can fit in. If found, allocate on the free block. If there do not exist such blocks, extend the heap and allocate there.

```
void *mm_malloc(size_t size)
{
    size_t adjusted_size;
    char *blk_ptr;
    if(size == 0) return NULL;
    if(size <= DSIZE)
        adjusted_size = MIN_BLK_SIZE;
    else
        adjusted_size = DSIZE * ((size + DSIZE + DSIZE - 1) / DSIZE);
    if((blk_ptr = find_fit(adjusted_size)) != NULL)
        return place(blk_ptr, adjusted_size);
    if((blk_ptr = extend_heap(MAX(adjusted_size, CHUNKSIZE)/ WSIZE)) == NULL)
        return NULL;
    return place(blk_ptr, adjusted_size);
}
```

The following function is `void *place(void *blk_ptr, size_t adjusted_size)`. If the difference between the size of the block and the `size` is larger than `MIN_BLK_SIZE`, it makes the remaining space a free block. If not, it just uses all the space.

```
static void *place(void *blk_ptr, size_t size)
{
    size_t block_size = GET_SIZE(HDRP(blk_ptr));
    void *next_blk_ptr = NULL;
```

```
    remove_from_list(blk_ptr);
    if ((block_size - size) >= (MIN_BLK_SIZE)) {
        PUT(HDRP(blk_ptr), PACK(block_size - size, 0));
        PUT(FTRP(blk_ptr), PACK(block_size - size, 0));
        next_blk_ptr = NEXT_BLKP(blk_ptr);
        PUT(HDRP(next_blk_ptr), PACK(size, 1));
        PUT(FTRP(next_blk_ptr), PACK(size, 1));
        insert_to_list(blk_ptr, block_size - size);
        return next_blk_ptr;
    }
    else {
        PUT(HDRP(blk_ptr), PACK(block_size, 1));
        PUT(FTRP(blk_ptr), PACK(block_size, 1));
    }
    return blk_ptr;
}
```

3. mm_free

In the `mm_free` function, it marks free on the header and footer of the block and inserts it into the list. `coalesce` should be executed.

```
void mm_free(void *blk_ptr)
{
    size_t size = GET_SIZE(HDRP(blk_ptr));
    PUT(HDRP(blk_ptr), PACK(size, 0));
    PUT(FTRP(blk_ptr), PACK(size, 0));
    insert_to_list(blk_ptr, size);
    coalesce(blk_ptr);
}
```

4. mm_realloc

`mm_realloc` function reallocates memory considering several cases:

- Default behavior: If `size = 0`, free it. If `blk_ptr = NULL`, malloc it. If the `size` is the same, just return the original one.
- Case 1: If the new size is less than the old size, allocate to the original block.
- Case 2: If the new size is larger than the old size, the next block is a free block, and the sum of the size of the two blocks is less than the new size, change the original block size to be able to hold the new one.
- Case 3: Otherwise, we should use a completely different address, so malloc new address and copy the original one into the new address, and free the old one.

It does not use the case of that the previous block is free, since copying into the previous block is risky.

```
void *mm_realloc(void *blk_ptr, size_t size)
{
    if (size == 0) {
        mm_free(blk_ptr);
        return NULL;
    }
    if (blk_ptr == NULL)
        return mm_malloc(size);
    size_t align_size = ALIGN(size);
    size_t org_size = GET_SIZE(HDRP(blk_ptr)) - DSIZE;
    if (align_size == org_size)
        return blk_ptr;
    else if (align_size < org_size) {
        if (org_size - align_size < MIN_BLK_SIZE)
            return blk_ptr;
        PUT(HDRP(blk_ptr), PACK(align_size + DSIZE, 1));
        PUT(FTRP(blk_ptr), PACK(align_size + DSIZE, 1));
        void *next_blk_ptr = NEXT_BLKP(blk_ptr);
        PUT(HDRP(next_blk_ptr), PACK(org_size - align_size, 0));
        PUT(FTRP(next_blk_ptr), PACK(org_size - align_size, 0));
        insert_to_list(next_blk_ptr, GET_SIZE(HDRP(next_blk_ptr)));
```

```
        coalesce(next_blk_ptr);
        return blk_ptr;
    }
    else if ((NEXT_BLKP(blk_ptr) != NULL)
&& !GET_ALLOC(HDRP(NEXT_BLKP(blk_ptr)))) {
        size_t next_size = GET_SIZE(HDRP(NEXT_BLKP(blk_ptr)));
        if (next_size + org_size >= align_size) {
            remove_from_list(NEXT_BLKP(blk_ptr));
            if (next_size + org_size - align_size < MIN_BLK_SIZE) {
                PUT(HDRP(blk_ptr), PACK(org_size + DSIZE + next_size, 1));
                PUT(FTRP(blk_ptr), PACK(org_size + DSIZE + next_size, 1));
                return blk_ptr;
            }
            else {
                PUT(HDRP(blk_ptr), PACK(align_size + DSIZE, 1));
                PUT(FTRP(blk_ptr), PACK(align_size + DSIZE, 1));
                void *next_ptr = NEXT_BLKP(blk_ptr);
                PUT(HDRP(next_ptr), PACK(org_size + next_size - align_size, 0));
                PUT(FTRP(next_ptr), PACK(org_size + next_size - align_size, 0));
                insert_to_list(next_ptr, GET_SIZE(HDRP(next_ptr)));
                return blk_ptr;
            }
        }
    }
    void *new_blk_ptr = mm_malloc(size);
    if (new_blk_ptr == NULL)
        return NULL;
    memcpy(new_blk_ptr, blk_ptr, org_size);
    mm_free(blk_ptr);
    return new_blk_ptr;
}
```

Discussion

What was Difficult?

The most difficult part when implementing a dynamic memory allocator was how to implement the segregated free list. The idea to use non-header/footer space was difficult to apply easily. Understanding each macro was also a challenge. By reading the textbook, we were able to get a clue to understanding each macro.

Something New and Surprising

In the implementation of `place`, the space was divided so that the front part was free space, and the rear part was allocated space. The method of dividing the space reversely was also considered, but in that case, it was observed that the score decreased in terms of space utilization.

Conclusion

In this lab session, we implemented a malloc package using the segregated free list. In this approach, a block is allocated by finding the optimal free block in the segregated free list. the segregated free list contains doubly linked lists each block in the list has a size from 2^(index) to 2^(index+1)-1. By maintaining each list is sorted, finding optimal free blocks can be conducted readily and quickly. Once the block is found, mark the block as allocated by changing the alloc info of the header and footer of the block. If such a free block is not found, extend heap memory to allocate.

Freeing the allocated memory is conducted by changing the alloc info in the header and footer of the block. After adding the block to the list, coalesce adjacent free blocks to manage the free block efficiently.

Reallocation is implemented considering three cases excluding trivial cases:

(1) If the new size is less than the old size, update the header and footer and make the remaining space a free block, and coalesce.

(2) If the new size is larger than the old size and the next block is free, then check if the next block is large enough for the remainder of the new size. If it is, coalesce two blocks and allocate a new block of the size.

(3) Otherwise, just malloc for a completely new address copy the data, and free the original one.

By implementing this, fragmentation can be reduced, and throughput is extremely improved.