CS201
REPORT: MALLOC LAB

1551020 - Vo Tran Thanh Luong - 1551031 - Dinh Dat Thanh

December 24, 2016

Contents

1 Theory and Method Chosen **2**

2 Our Mission **2**

3 Preparation **3**

4 MMinit **6**

5 MMMaloc **8**

6 MMFree **10**

7 MMRealloc **11**

8 Team Credit **13**

9 Thank you note **13**

Warning : the codes I input in this report are for descriptive reason only. I keep them tiny because normal size would make them spam in the report. They are all extracted from the main source code file. Although they are still readable here, I highly recommend viewing them in the main source code file.

1 Theory and Method Chosen

There are many ways to solve this lab assignment as we have learnt about ways to keep track of free blocks. There are 3 ways mentioned : implicit free list, explicit free list, and segregated free list, but for the sake of higher throughput and better memory optimization, we opt for segregated free list. (space efficiency and speed efficency)

2 Our Mission

At first glance, we can obviously expect what we need to do : malloc(): reserve a block of memory, free(): release a chunk of memory, realloc(): resize a reserved chunk. In details :

int mm_init(void)

Will be called by the trace-driven program in order to perform necessary initializations

ie. allocating initial heap area

Return -1 if an error occurs and 0 if otherwise

void mm_malloc(size_t size)

Returns pointer to an allocated block payload of at least size bytes

Block must lie within heap area

Block cannot overlap with another allocated chunk

Must return 8-byte aligned pointer

void mm_free(void ptr)

Frees the block pointed to by ptr

Returns nothing

void mm_realloc(void ptr, size_t size)

If ptr is NULL, call mm_malloc(size)

If size==0, call mm_free(ptr)

If otherwise, change the size of the block of memory pointed to by ptr to size bytes and returns the address of this new block

Possible for this address to remain the same as for the old block ie: if a consecutive block of memory is free and satisfies the given size requirement

Contents of memory block remain the same (as much as possible if allocating to a smaller size block)

3 Preparation

The memlib.c package simulates the memory system for your dynamic memory allocator. You can invoke the following functions in memlib.c:

void mem sbrk(int incr): Expands the heap by incr bytes, where incr is a positive non-zero integer and returns a generic pointer to the first byte of the newly allocated heap area. The semantics are identical to the Unix sbrk function, except that mem sbrk accepts only a positive non-zero integer argument.

void mem heap lo(void): Returns a generic pointer to the first byte in the heap.

void mem heap hi(void): Returns a generic pointer to the last byte in the heap.

size t mem heapsize(void): Returns the current size of the heap in bytes.

size t mem pagesize(void): Returns the system's page size in bytes (4K on Linux systems)

We will use sbrk as our main tool to implement malloc. All we have to do is to acquire more space (if needed) to fulfil the query.

First question that comes to our mind is that how do we represent the block information. As we are using segregated freelist, it is crucial that we need to include a small block at the beginning of each data chunk for the metadata

(header, footer, pointer). Translating this idea into code, however, is a very hard problem. Initially, we tried with struct as each struct will represent a block of memory for metadata. Problem comes when we realized that create a struct without a working malloc is too hard so we selected another approach : define a bunch of macros.

Our macro comprises of :

1. **Alignment 8**. *It is often required that pointers be aligned to the integer size. We are doing this lab on a 64 bit machine so our pointer must be a multiple of 8. Therefore, the alignment macro here is equal to 8, as the size of data block.*

2. **ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)**. *The alignment size that malloc need to create may not be a multiple of 8. However, the way blocks of memory work requires us to make it an multiple of 8 (even if it is bigger than the size we need, size + padding), thus our job is to find the nearest multiple of 8 of the size we need to align. This is simple and can be done by a bitwise trick. Let X be an integer such that X = 8 * p + q (with q from 0 to 7). We have the formula (X+7) - ((X+7) - (X+7)/8 * 8) which always results in the nearest mulitple of 8 of X. Expressing it in bitwise is as the above definition.*

3. **SIZE_T_SIZE (ALIGN(sizeof(size_t)))** :*define size_t size parameter.*

4. **WSIZE 4** .*Word size equals to 4 (4 bytes for header and footer)*

5. **DSIZE 8** .*Double word size equals to 8.*

6. **INITCHUNKSIZE (1 <<6)** *(2 to the power of 6 is 64) the size of the initial free block.*

7. **CHUNKSIZE (1 <<12)** *(2 to the power of 12) the default size for expanding the heap.*

8. **LIST 20** .*size of segregated list = 20.*

9. **REALLOC_BUFFER (1 <<7)** .*size of realloc buffer : 2 to the power of 7.*

10. **MAX(x, y) ((x) >(y) ? (x) : (y))** *A macro to decide the bigger number between 2 numbers.*

11. **MIN(x, y) ((x) <(y) ? (x) : (y))** *A macro to decide the smaller number between 2 numbers.*

12. **PACK(size, alloc) ((size) |(alloc))** *Pack a size and allocate it into a word.*

13. **GET, PUT, PUTNOTAG** *are all actions to read , return or write a word at address p. PUT differs from PUTNOTAG that PUT also takes the tag (which is the second bit of the last 2 bits).*

14. **SET_PTR(p, ptr) ((unsigned int)(p) = (unsigned int)(ptr))** *Store predecessor or successor pointer for free blocks.*

15. **GET_SIZE(p) (GET(p) & ~0x7)** *because except the last 3 tags, the remainder is the size.*

16. **GET_ALLOC(p) (GET(p) & 0x1)** *in the last 2 bits, the outmost bit is the bit that show whether the block is allocated or not.*

17. **GET_TAG(p) (GET(p) & 0x2)** *gets the tag from the last 2 bits.*

18. **SET_RATAG(p) (GET(p) |= 0x2)** *set reallocated tag.*

19. **REMOVE_RATAG(p) (GET(p) & = ~0x2)** *remove reallocated tag.*

20. **HDRP(ptr) ((char)(ptr) - WSIZE) and FTRP(ptr) ((char ×)(ptr) + GET_SIZE(HDRP(ptr)) - DSIZE)** *returns the address of header and footer when given a pointer that points to a block- or a pointer that points to the first payload.*

21.**NEXT_BLKP(ptr) ((char)(ptr) + GET_SIZE((char)(ptr) - WSIZE)) and PREV_BLKP(ptr) ((char)(ptr) - GET_SIZE((char)(ptr) - DSIZE))** *returns the address of next and previous blocks when given a pointer that points to the block.*

22.**PRED_PTR(ptr) ((char)(ptr) and SUCC_PTR(ptr) ((char ×)(ptr) + WSIZE)** *returns the address of next and previous FREE BLOCKS when given a pointer that points to the block.*

23. **PRED(ptr) ((char)(ptr)) and SUCC(ptr) ((char)(SUCC_PTR(ptr)))**

returns the address of the next and previous block on the segregated list.

4 MMinit

First of all, we need to create a heap with an initial free block using mm_init.

```
int mm_init(void)
{
        int index;
        //initialize the segregated free list NULL
        for(index=0;index< LIST; index++)
                free_lists[index] = NULL;

        char * heap;
        //cannot allocated the heap
        if((long)(heap = mem_sbrk(4 * WSIZE)) == -1)
                return  -1;
        //padding
        PUT_NOTAG(heap, 0);
        //input the prologue header
        PUT_NOTAG(heap + 1* WSIZE, PACK(DSIZE,1));
        //prologue footer
        PUT_NOTAG(heap + 2* WSIZE, PACK(DSIZE,1));
        //epilogue header
        PUT_NOTAG(heap + 3* WSIZE, PACK(0,1));

        if(extend_heap(INITCHUNKSIZE)==NULL)
                return -1;
    return 0;
}
```

Because we are using segregated free list, we need to initialize the segregated free list (with all value NULL). Then, we allocate the heap. There are cases when initialization problems happen and it should return -1. If everything is okay, the mm_init function gets four words from the memory and initialize them to create the empty free list. After that it calls the extend heap function, which extends the heap by the CHUNKSIZE bytes and creates the initial free block. There are two cases when the extend_heap function is invoked one is when the heap is initialized, and another is when mm_malloc is unable to find a suitable fit. The purpose of extend_heap is to round up the requested size to the nearest multiple of 8 (which is 2 words) then request additional heap space from the memory if neccessary. We need to notice that the heap begins on a double-word aligned boundary, and every call to extend_heap returns a block whose size is an integral number of double words. Thus, every call to mem_sbrk returns a double-word aligned chunk of memory immediately following the header of the epilogue block. This header becomes the header of the new free block, and the last word of the chunk becomes the new epilogue block header. The extend heap function should also insert free node at the pointer position into the segregated list.

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

```
        //not enough space
        if(ptr == (void * ) -1)
                return NULL;

        //set header and footer infomation
        //header
        PUT_NOTAG(HDRP(ptr),PACK(tempsize,0));
        //footer
        PUT_NOTAG(FTRP(ptr),PACK(tempsize,0));
        PUT_NOTAG(HDRP(NEXT_BLKP(ptr)),PACK(0,1));
        //insert free node
        insert_node(ptr,tempsize);

        return coalesce(ptr);
}
```

Inside the insert node function we find a good position by traversing the list. There are 3 cases :insert between the list, insert at the beginning at the list or insert at the end of the list. We uitilizes all the SETPTR macro that has been predefined. This operations is very simple and similar to a linked list.

```
static void insert_node(void * ptr, size_t size)
{
        int index;
        void *next = ptr;
        void *before = NULL;

        for(index=0;index < LIST -1; index++ )
        {
                if(size > 1)
                {
                        size = size >> 1;
                }
                else break;
        }
        next = free_lists[index];
        //traverse the free list to find a position to input the node
        while( next !=NULL && size < GET_SIZE(HDRP(next)))
        {
                before = next;
                next = PRED(next);
        }
        if(next != NULL)
        {
                //insert between the list
                if(before!= NULL)
                {
                        SET_PTR(PRED_PTR(ptr),next);
                        SET_PTR(SUCC_PTR(next), ptr);
                        SET_PTR(PRED_PTR(before), ptr);
                        SET_PTR(SUCC_PTR(ptr), before);
                }
                //insert at the begining of the list
                else
                {
                        SET_PTR(PRED_PTR(ptr), next);
                        SET_PTR(SUCC_PTR(next), ptr);
                        SET_PTR(SUCC_PTR(ptr), NULL);
                        //update the root of the free list
                        free_lists[index]= ptr;
                }
        }
        //at the end of the list
        else
        {
                //at the end of the list
                if(before!=NULL)
                {
                        SET_PTR(PRED_PTR(ptr),NULL);
                        SET_PTR(SUCC_PTR(ptr), before);
                        SET_PTR(PRED_PTR(before),ptr);
                }
                //the list is empty initially at that index
                else
                {
```

```
                        SET_PTR(PRED_PTR(ptr),NULL);
                        SET_PTR(SUCC_PTR(ptr),NULL);
                        //update the root of free list at the index
                        free_lists[index]=ptr;
                }
        }
        return;

}
```

5 MMMaloc

Now we do our malloc function. In our malloc function, we define the block size that need to be alligned. Then we traverse the segregated free list and find the appropriate one, which is the smallest different size block (because the list is from large to small size, after everyloop we drop one bit in the search size) . If the appropriate block is found(fit), we place the block to the memory. If no appropriate block is found (not fit), we extend the heap and place the requested block in the new free block. If the heap cannot be extended, then we return null. If everything is successful, we return the pointer that points to the allocated block. Also note that we use the helper function place to help place the block to the memory. In this function, we took a free block from the list then bascially use it. But the problem here is to decide whethere to split or not to prevent losing a lot of spaces. When a chunk is wide enough to held the asked size plus a new chunk, we split it and put the allocated block at the end of the free block then insert the remaining free block back to the segregated free list. These are all done thanks to the PUT macro that we predefinde to write a word at a address (either with tag or without tag). You may have cared what the delete node function do. This function delete the node in the segregrated free list to input. It operates just like when we delete a node from a linked list, we modify the pointers to that node. Here we modify the next and the prev pointers, which mean we need to know where the current node is.

```
// helper function place
static void * place(void * ptr, size_t asize)
{
        size_t size = GET_SIZE(HDRP(ptr));
        size_t remain = size - asize;

        delete_node(ptr);

        if(remain <= DSIZE*2)
        {
                //do not split
                PUT(HDRP(ptr), PACK(size,1));
                PUT(FTRP(ptr), PACK(size,1));
        }

        else if(asize >= 100)
        {
                //split block
```

```
                PUT(HDRP(ptr), PACK(remain,0));
                PUT(FTRP(ptr), PACK(remain,0));
                //put the allocated block at the end of the free block
                PUT_NOTAG(HDRP(NEXT_BLKP(ptr)), PACK(asize,1));
                PUT_NOTAG(FTRP(NEXT_BLKP(ptr)), PACK(asize,1));
                //insert the remainder free block to segregated free list
                insert_node(ptr,remain);
                return NEXT_BLKP(ptr);
        }
        //put the allocated block at the beginning of the free block
        else
        {
                //split block
                PUT(HDRP(ptr), PACK(asize,1));
                PUT(FTRP(ptr), PACK(asize,1));
                PUT_NOTAG(HDRP(NEXT_BLKP(ptr)), PACK(remain,0));
                PUT_NOTAG(FTRP(NEXT_BLKP(ptr)), PACK(remain,0));
                insert_node(NEXT_BLKP(ptr),remain);
        }
        return ptr;
}


// malloc function
void *mm_malloc(size_t size)
{
    if(size==0)
        return NULL;

    size_t asize ; //adjust size
    size_t extend; //extend heap if neccessary
    void * ptr = NULL;

    //align block size
    if( size <= DSIZE)
    {
        asize = 2*DSIZE;
    }
    else
        {
        asize =ALIGN(size + DSIZE);
        }

        int index=0;
        size_t search =asize;
        //traverse the segregated free list
        while(index < LIST)
        {
                //find the appropriate free list
                if((index == LIST -1) || (search <= 1 && free_lists[index] != NULL))
                {
                        ptr = free_lists[index];
                        //ignore the block with reallcation bit and find the
                             smallest different size block
                        while(ptr !=NULL  && ((asize > GET_SIZE(HDRP(ptr)) ||
                             GET_TAG(ptr))))
                        {
                                ptr = PRED(ptr);
                        }
                        //can find the free block
                        if(ptr != NULL)
                                break;
                }
                search = search >>1;
                index ++;
        }
        //expand the heap to allocate
        if(ptr == NULL)
        {
                extend = MAX(asize,CHUNKSIZE);
                //cannot extend the heap
                ptr = extend_heap(extend);
                if(ptr == NULL)
                        return NULL;
        }
        //place and divide block to the memory
        ptr = place(ptr,asize);

        //return pointer to the allocated block
```

```
        return ptr;
}
```

6 MMFree

MMfree free a block then connect it with other free blocks. Here we are applying segregated free list so we will implement the way that segregated free list coalesce free block. Freeing a block is simple as we remove the reallocated tag from it, put new info about header and footer in it to confirm it is now free, then put the free block into the segregated free list. The coalesce function check if the next and the previous block is reallocated. If the prev block is reallocated do not coalese. If both the prev and the next are not free, do not coalesce. We can only coalesce in three case : either the next or the prev is not allocated or both of them are free. Simple put the pointers to their right places and define the new free size. With this new size, we also need to find a proper place to put on the segregated free list.

```
// helper function coalesce
 static void * coalesce(void * ptr)
{
        //check if the prevrious block is allocated
        size_t prev_all =GET_ALLOC(HDRP(PREV_BLKP(ptr)));
        //check if the next block is allocated
        size_t next_all =GET_ALLOC(HDRP(NEXT_BLKP(ptr)));
        size_t size =GET_SIZE(HDRP(ptr));

        //if the previous is reallocated, do not coalesce
        if(GET_TAG(HDRP(PREV_BLKP(ptr))) == 1)
                prev_all = 1;

        //cannot coalesce with previous and the next block
        if(prev_all == 1 && next_all ==1)
                return ptr;

        //can coalesce with the next block
        if(prev_all == 1 && next_all == 0)
        {
                delete_node(ptr);
                delete_node(NEXT_BLKP(ptr));
                //the new size of the coalesce free block
                size += GET_SIZE(HDRP(NEXT_BLKP(ptr)));
                //update the info at the header and the footer of the new free block
                        at the pointer
                PUT(HDRP(ptr), PACK(size,0));
                PUT(FTRP(ptr), PACK(size,0));
        }
        //coalesce with the previous block
        else if(prev_all == 0 && next_all == 1)
        {
                delete_node(ptr);
                delete_node(PREV_BLKP(ptr));
                size+= GET_SIZE(HDRP(PREV_BLKP(ptr)));
                ptr= PREV_BLKP(ptr);
                PUT(HDRP(ptr), PACK(size,0));
                PUT(FTRP(ptr), PACK(size,0));
        }
        //coalesce with both previous and next block
        else if (prev_all ==0 && next_all ==0)
        {
                delete_node(ptr);
```

```
                delete_node(PREV_BLKP(ptr));
                delete_node(NEXT_BLKP(ptr));

                size+= GET_SIZE(HDRP(PREV_BLKP(ptr))) + GET_SIZE(HDRP(NEXT_BLKP(ptr)
                    ));

                ptr = PREV_BLKP(ptr);
                PUT(HDRP(ptr), PACK(size,0));
                PUT(FTRP(ptr), PACK(size,0));
        }
        //insert the new free list to the segregated free list
        insert_node(ptr,size);
        return ptr;
}

// mmfree
void mm_free(void *ptr)
{
        size_t size= GET_SIZE(HDRP(ptr));
        //remove the reallcoted tag
        REMOVE_RATAG(HDRP(NEXT_BLKP(ptr)));
        //input the new info in to the block
        PUT(HDRP(ptr), PACK(size,0));
        PUT(FTRP(ptr), PACK(size,0));
        //insert the node to the segregated free list
        insert_node(ptr,size);
        //expand the free block if possible
        coalesce(ptr);

}
```

7 MMRealloc

The most naiive approach is to reallocate a new block of the given size using malloc, copy data from the old one to the new one and free the old block. But here we do something different a little bit. Instead of purely copy the data, we extend the current data size to fit the new requirements. First we check if the size of the new block is smaller than a double word size. If yes then the new size needed will fit in a space that is double the old size. If not, we need to add the double word size to the old size and then round up to the multiple of 8 (by using ALGIN). After that we need to add overhead requirement for block size and calculate the block buffer. When there is not enough space for the new block size, we check if the next block is free or the epilogue block fo the next block. If the next block is free, we check if the next block free size + the current block size is enough to make up space for the new block size. IF there is still not enough space, we need to extend the heap. IF there is now enough space, we delete the next free block from the segregated list and put value in. In case there the next block is not free, it is fine for us now to alloc a new size and copy the old data into new place using memcpy (as the naiive approach). After all of this, it is also imperative that we set the reallocation tag and then return the reallocation block.

```
// helper function void * memcpy(void * destination, const void * source, size_t num
    );
```

```
void *mm_realloc(void *ptr, size_t size)
{
        if(size == 0 )
                return NULL;
    void * oldptr = ptr;
    size_t newsize =size;       //size of the new block
    int remain; //the remain size after allocation
    int extend; //size of heap extension
    int blockbuff;

    //align block size
    if( size <= DSIZE)
    {
        newsize = 2*DSIZE;
    }
    else
        {
        newsize =ALIGN(size + DSIZE);
        }
        //add overhead requirment for block size
        newsize += REALLOC_BUFFER;
        //calculate the block buffer
        blockbuff = GET_SIZE(HDRP(ptr)) - newsize;
        //not enough space
        if(blockbuff < 0)
        {
                //check if the next block is free or the epilogue block
                if(GET_ALLOC(HDRP(NEXT_BLKP(ptr)))==0 || GET_SIZE(HDRP(NEXT_BLKP(ptr
                        ))) == 0)
                {
                        //calculate the space missing
                        remain = GET_SIZE(HDRP(ptr)) + GET_SIZE(HDRP(NEXT_BLKP(ptr))
                              ) - newsize;
                        //not enough space
                        if(remain < 0)
                        {
                                extend = MAX ( -remain, CHUNKSIZE);
                                //can not extend the heap
                                if(extend_heap(extend) == NULL)
                                        return NULL;
                                remain += extend;
                        }

                        delete_node(NEXT_BLKP(ptr));

                        //do not split block, update the info of the current block
                        PUT_NOTAG(HDRP(ptr), PACK(newsize + remain,1));
                        PUT_NOTAG(FTRP(ptr), PACK(newsize + remain,1));

                }
                else
                {
                        //enough space to allocated into the block
                        oldptr = mm_malloc(newsize -DSIZE);
                        memcpy( oldptr, ptr, MIN(size,newsize));
                        mm_free(ptr);
                }
                blockbuff = GET_SIZE(HDRP(oldptr)) - newsize;
        }

        if(blockbuff < 2 * REALLOC_BUFFER)
        {
                SET_RATAG(HDRP(NEXT_BLKP(oldptr)));
        }
        //return the reallocation block
        return oldptr;
}
```

8 Team Credit

Dinh Dat Thanh :

- Set basic idea and theory (segregated free list)
- MM Alloc & helper functions
- MM Free & helper functions
- Debug files

Vo Tran Thanh Luong :

- Enhance idea (segregated free fits list)
- MM Realloc
- Report Writing

Both:

- Cross code checking
- Mutual explaination

9 Thank you note

Thank you Prof Nghiem Quoc Minh and our TAs for having introduced the basis of memmory allocation.