CS3103 - Operating Systems

Group Project Report: Dynamic Memory Allocation

Group 02

Group members:

LIU Junqi 54380685

FENG Chuxiao 54780472

TANG Junyi 54380384

CONTENTS

1. Introduction

There are a lot of problems we may encounter when we try to programming. One of those most common problems we discussed in this report is dynamic memory allocation. The basic reason for using dynamic memory allocation is that we don't know the exact sizes of certain data structures until the program actually runs. Dynamic memory allocation is a better approach compared with hard-coded size allocation when facing this kind of circumstance.

This report illustrates a project for dynamic memory allocator design. The report comprises seven parts, which are Introduction, Methodologies, Code implementation, Result and Illustration, Challenges, Teamwork and Reference. Introduction section describes the motivation of our project and the overview of the report. In Methodologies section, we describe the main idea of constructing the dynamic memory allocator and Code Implementation section provide the usage and logic among all the functions. Result section displays the performance of our code in terms of the overall performance index and performance regarding 9 traces. Challenges part are devoted to considering the problems and difficulties we encountered as well as the knowledge we learned throughout the project.

2. Methodologies

In order to maximize throughput and peak memory utilization which always conflict each other, our allocator adopts explicit free list to storage the free blocks and first fit to keep track of the free blocks.

The general procedures and ideas used during implementation are as follows:

a. Create Initial Free list

b. Free and Coalescing Blocks

c. Allocating Blocks

The method we use for free block organization is the explicit free list. Instead of traverse each block in the order in the implicit free list regardless of whether the block is free or not, explicit free list only maintain lists of free blocks, which and modified each block by adding two pointers to make the implicit list a doubly linked list. The improvement free block organization reduced the time for searching goal free block. With respect to placement, First Fit is chosen, which scans the free list in order and selects the first free blocks of the appropriate size that is larger than the required size.
In splitting process, we split the selected blocks when it is larger than the request size. As for coalescing, we use immediate coalescing with the method of boundary tags.

Some details and procedures about explicit free list method are shown here:

Step1: Find_fit

As long as there is any extra memory needs to be assigned, we have to search the whole list and find the appropriate place to assign.

Generally, we have three options which are

a.First Fit

b.Next Fit

c.Best Fit

Among these options, they all get their own advantages and disadvantages. First Fit/Next Fit is nearly the fastest method to find the fit blocks, but it won't offer us the most appropriate block to place so the final throughput may lower than other methods.

The requirement of best fit is (`needsize<=size && min(size-needsize)`), best fit can offer us the most appropriate block to place and the whole throughput is higher while it performs relatively slow when searching some huge free list.

Base on the traces we have, we finally choose to use first fit to implement this project.

Step2: Place && Split

If we already get the appropriate block, we can directly place the extra word in these memories. But if the block we get is much bigger than what we need. We have to do split the block base on the size of the smallest block we need. Then some part is used to place those extra words, the other will be reallocated to the free list.

Step3: Coalesce

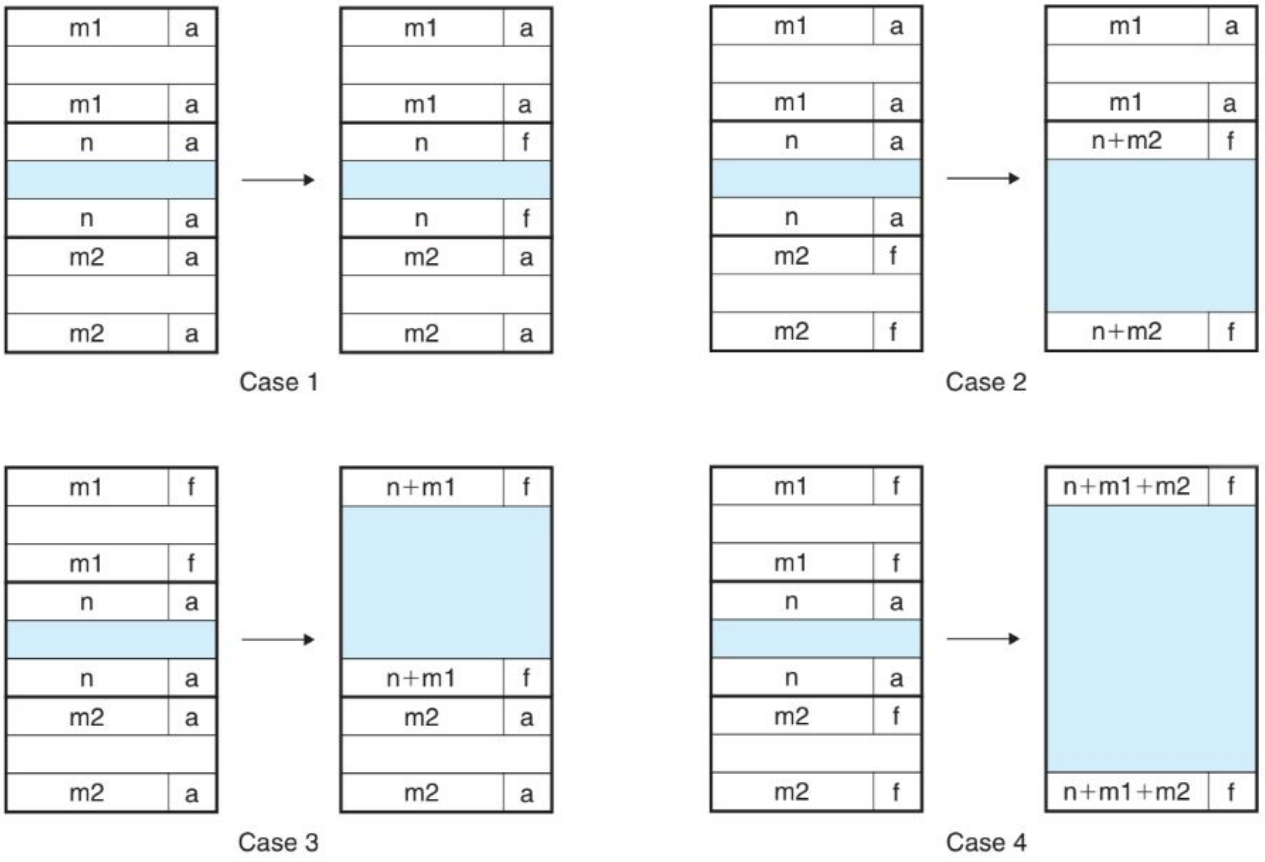
When user free assigned blocks or reallocate split block, we have to do the coalesce. Actually, there are four cases when we do coalesce shown below:

a. The previous and next blocks are both allocated.

b. The previous block is allocated and the next block is free

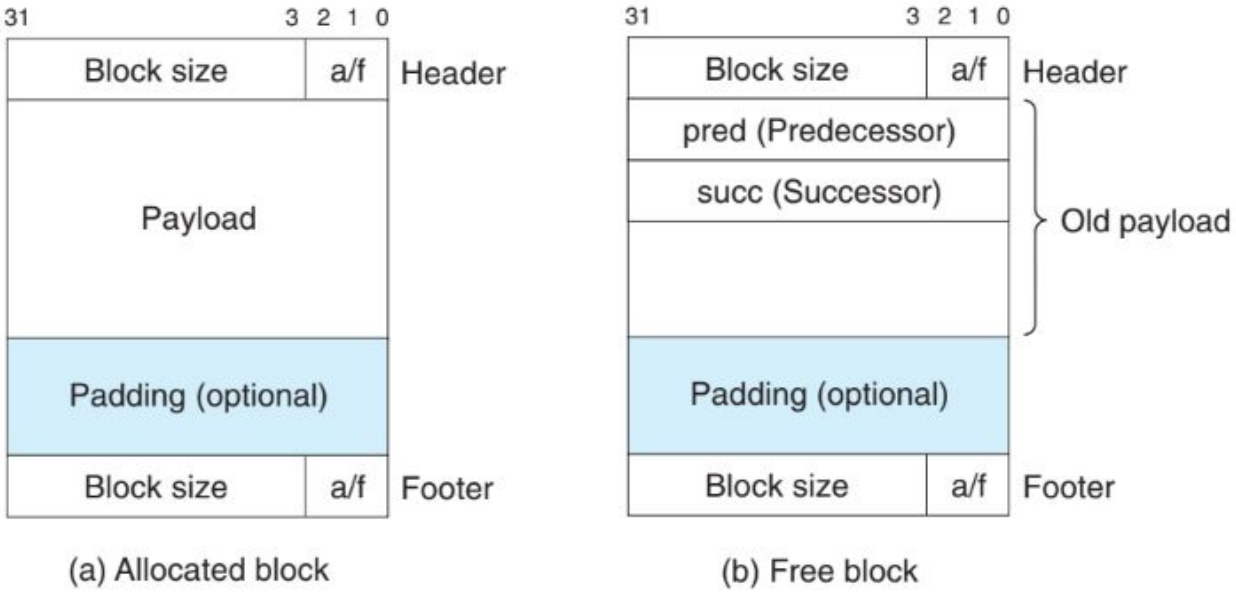
c. The previous block is free and the next block is allocated

d. The previous and next blocks are both free

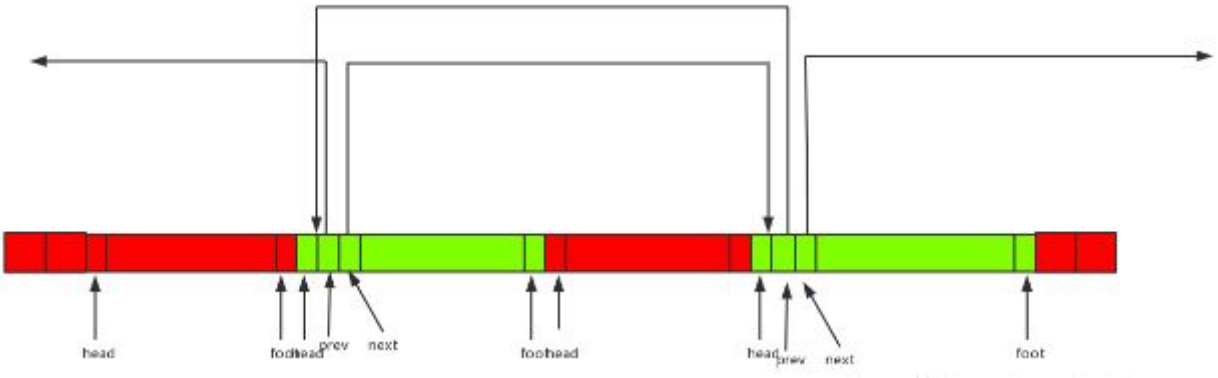


The general idea of explicit free list:

Compared with the implicit free list, the explicit free list is a better way to organize those free blocks. Since the body of free blocks are not needed by the program, the pointers that implement the data structure can be stored within the bodies of free blocks



Using the double linked list instead of an implicit free list reduces the first fit allocation time from linear in the total number of blocks to linear in the number of free blocks.



But the disadvantages of explicit free list is that in general free blocks must be large enough to contain all of the necessary pointers, as well as the header and possibly a footer which leads to a larger minimum block size, and increases the potential for internal fragmentation.

3. Code Implementation

3.1 Initialization: Firstly, initializing the memory manager. We create the initial empty heap with a prologue header, a prologue footer, and an epilogue header, which is the same as the the provided implicit free list. The extension of the empty heap is finished by enlarging a free block of CHUNKSIZE bytes. Finally, the mm_init function will return 0 if the memory manager is successfully initialized and -1 otherwise.

```
int mm_init(void)
{
   if ((heap_listp = mem_sbrk(8*WSIZE)) == (void *)-1)
            return -1;
   PUT(heap_listp, 0);                          /* Alignment padding */
   PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1)); /* Prologue header */
   PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1)); /* Prologue footer */
   PUT(heap_listp + (3*WSIZE), PACK(0, 1));     /* Epilogue header */
   heap_listp += (2*WSIZE);
   free_listp = heap_listp;
   if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
            return -1;
   return 0;
}
```

3.2 Allocation: The mm_malloc function aims to allocate a block with at least size bytes of payload unless the size is 0. We should adjust the block size if the space of the heap is not sufficient. Then, search the free list for a fit. It will return the address of the specific block if the allocation is successful and NULL otherwise.

```
void *mm_malloc(size_t size)
{
   size_t asize;      /* Adjusted block size */
```

```
    size_t extendsize; /* Amount to extend heap if no fit */
    char *bp;
    if (heap_listp == 0){
            mm_init();
    }
    if (size == 0)
            return NULL;
    if (size <= DSIZE)
            asize = 2*DSIZE;
    else
            asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);
    if ((bp = find_fit(asize)) != NULL) {
            place(bp, asize);
            return bp;
    }
    extendsize = MAX(asize,CHUNKSIZE);
    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
            return NULL;
    place(bp, asize);
    return bp;
}
```

3.3 Free a block: Explicit allocators require the application to explicitly free any allocated blocks.

```
void mm_free(void *bp)
{
    if(bp == 0)
            return;
    size_t size = GET_SIZE(HDRP(bp));
```

```
    if (heap_listp == 0){
        mm_init();
    }
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    coalesce(bp);
}
```

3.4 Coalescing: In this function, we successfully realize performing boundary tag coalescing. Finally, it will return the address of the coalesced block. There are four cases need to be considered. Noticed that "bp" is the address of a newly freed block.

```
static void *coalesce(void *bp)
{
    bool prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp))) || PREV_BLKP(bp) == bp;
    bool next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));
    if (prev_alloc && next_alloc) {
        freeList_push(bp);
        return bp;
    }
    else if (prev_alloc && !next_alloc) {
        freeList_pop(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) {
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0));
```

```
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        freeList_pop(PREV_BLKP(bp));
        bp = PREV_BLKP(bp);
    }

    else {
        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));
        freeList_pop(PREV_BLKP(bp));
        freeList_pop(NEXT_BLKP(bp));
        bp = PREV_BLKP(bp);
    }
    freeList_push(bp);
    return bp;
}
```

3.5 Newly added methods: "freeList_push" adds free block pointed by ptr to the free_list and "freeList_pop" deletes free block pointed by ptr to the free_list.

```
static void freeList_push(void* ptr){
    NEXT_PTR(ptr) = free_listp;
    PREV_PTR(free_listp) = ptr;
    PREV_PTR(ptr) = NULL;
    free_listp = ptr;
}
static void freeList_pop(void* ptr){
    if(PREV_PTR(ptr) == NULL)
        free_listp = NEXT_PTR(ptr);
```

```
    else
        NEXT_PTR(PREV_PTR(ptr)) = NEXT_PTR(ptr);
    PREV_PTR(NEXT_PTR(ptr)) = PREV_PTR(ptr);
}
```

3.6 Extension of the heap: Extend heap with the free block and return its block pointer. It is important to allocate an even number of words to maintain alignment.

```
static void *extend_heap(size_t words)
{
    char *bp;
    size_t size;
    size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
    if ((long)(bp = mem_sbrk(size)) == -1)
            return NULL;
    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 */
    return coalesce(bp);
}
```

3.6 Placement: Place the block of “asize” bytes at the start of free block “bp” and split if the remainder is at least minimum block size.

```
static void place(void *bp, size_t asize)
{
    size_t csize = GET_SIZE(HDRP(bp));
    if ((csize - asize) >= (2*DSIZE)) {
        freeList_pop(bp);
            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));
        coalesce(bp);
    }
    else {
        freeList_pop(bp);
            PUT(HDRP(bp), PACK(csize, 1));
            PUT(FTRP(bp), PACK(csize, 1));
    }
}
```

3.7 First fit implementation: Find out a fit for a block with "asize" bytes

```
static void *find_fit(size_t asize)
{
    void *bp;
    for (bp = free_listp; GET_ALLOC(HDRP(bp)) == 0; bp = NEXT_PTR(bp)) {
            if (asize <= GET_SIZE(HDRP(bp)))
                return bp;
    }
    return NULL; }
```

3.8 Checking: "checkheap" is the minimal check of the heap for consistency.

```
static void checkblock(void *bp)
{
    if (GET(HDRP(bp)) != GET(FTRP(bp)))
            printf("Error: The header does not meet with the footer\n");
    if ((size_t)bp % 8)
            printf("Error: %p is not doubleword aligned\n", bp);
```

```
}
void checkheap(int verbose)
{
    void*bp = free_listp;
    while (NEXT_PTR(bp)!=NULL) {
      if (GET_ALLOC(HDRP(bp)) == 1 || GET_ALLOC(FTRP(bp)) == 1) {
          printf("Come across an allocated block in the free list.\n");
          return;
       }
          bp = NEXT_PTR(bp);
    }
    if (verbose == 1)
            printf("Heap (%p): ", heap_listp);
    if (GET_SIZE(HDRP(heap_listp)) != DSIZE ||
        !GET_ALLOC(HDRP(heap_listp)))
            printf("Bad p-header\n");
            checkblock(heap_listp);
    for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
            if (verbose == 1)
                printblock(bp);
            checkblock(bp);
    }

    if (verbose == 1)
            printblock(bp);
    if (GET_SIZE(HDRP(bp)) != 0 || !GET_ALLOC(HDRP(bp)))
            printf("Bad e-header\n");
}
```

4. Result and Illustration

Our allocator achieces a overall 88 performance index, with 48 unit for memory utilization and 40 n term of throughput. The testing result assessed by 9 test traces and performance summary are displayed below:

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   89%      12  0.000000 40000
 1       yes   88%    5694  0.000185 30778
 2       yes   91%    5848  0.000126 46266
 3       yes   94%    6648  0.000258 25807
 4       yes   96%    5380  0.000168 31986
 5       yes   66%   14400  0.000263 54857
 6       yes   88%    4800  0.000396 12133
 7       yes   85%    4800  0.000453 10596
 8       yes   54%   12000  0.003935  3050
 9       yes   47%   24000  0.002668  8995
Total          80%   83582  0.008452  9889
Perf index = 48 (util) + 40 (thru) = 88/100
```

5. Challenges

In this section, we describe the problems we encountered in the process of building the allocator. The first difficulty was encountered after the completion of the first version of modified allocator using segregated free list. This version of the code is correct to run but achieved extremely limit improvement (performance index of 58/100) in comparison with the original version provided

(performance index of 52/100). In the process of modifying, the second problem occurred having known the information that it is easy to get 80+ for the explicit free list, and 90+ for the segregated free list, we continued to modify our code. After many failed attempts, we decided to restart with an explicit free list instead of directly doing modifications in the segregated free list version.

6. Teamwork

	LIU Junqi	FENG Chuxiao	TANG Junyi
Department	Electronic Engineering	Computing Mathematics	Electronic Engineering
Programming	70%	15%	15%
Report	20%	40%	40%

7. References

[1]Randal E. Bryant and David R. O'Hallaron, Pearson, "Computer Systems: A Programmer's Perspective, Second Edition".

[2] Remzi H. Arpaci-Dusseau and Andrea C, "Operating Systems: Three Easy Pieces", March 2015 (Version 1.00).

[3] Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles, "Dynamic Storage Allocation: A Survey and Critical Review". International Workshop on Memory Management, Scotland, UK, September 1995.