CSAPPLab8 MallocLab

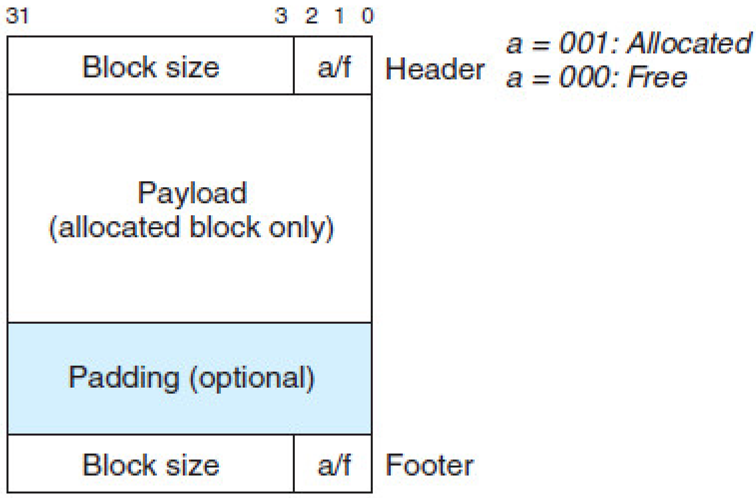
本实验要求实现一个动态内存分配器（*dynamic storage allocator*），主要实现 *malloc*，*free* 和 *realloc* 三个流程，最终结果性能从吞吐量和空间利用率两个方面进行综合评估。

书籍《深入理解计算机系统》第 *9.9.12* 节（综合：实现一个简单的分配器）其实已给出分配器实现的大部分代码，缺少的 *find_fit* 和 *place* 两个函数也在课后习题中要求完成。不断改进程序性能过程中，我实现了如下四个版本，分别为：

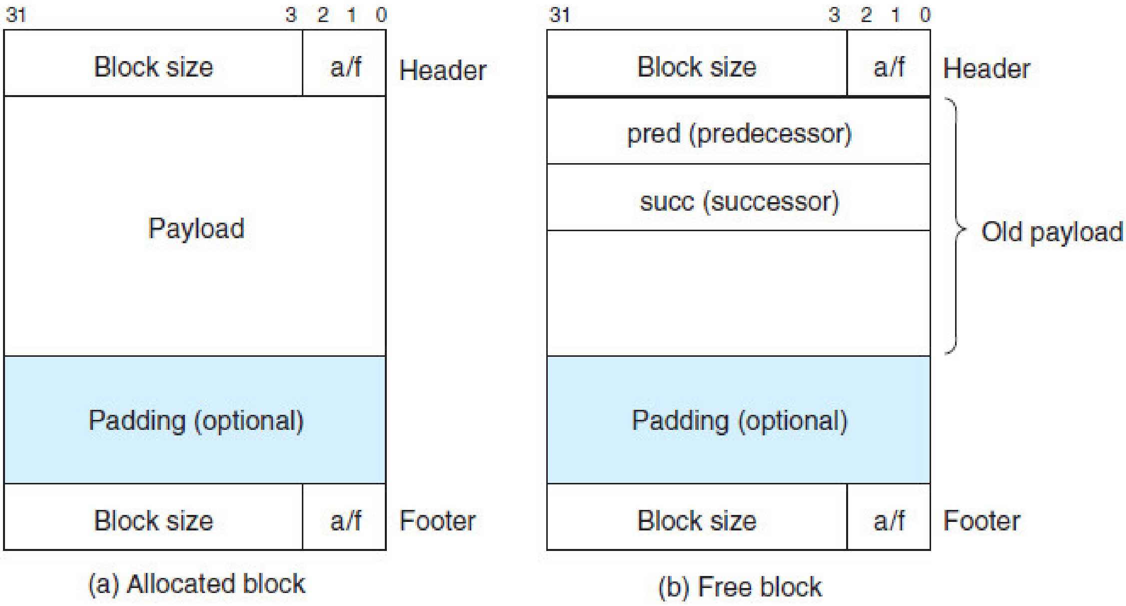
- 隐式空闲链表 + *mm_realloc*函数普通实现
- 隐式空闲链表 + *mm_realloc*函数改进实现
- 显式空闲链表 + *mm_realloc*函数普通实现
- 显式空闲链表 + *mm_realloc*函数改进实现

以上版本程序分配器采用的放置策略均为 **首次适配** 方式。下图分别为隐式空闲链表和显式空闲链表堆块的格式。

- 使用边界标记的堆块的格式（隐式空闲链表）



- 使用双向空链表的堆块的格式（显式空闲链表）



隐式空闲链表 + *"mm_realloc"*函数普通实现

该版本完整代码如下，大部分函数均为书中源代码。

```
/*
 * mm-naive.c - The fastest, least memory-efficient malloc package.
 *
 * In this naive approach, a block is allocated by simply incrementing
 * the brk pointer.  A block is pure payload. There are no headers or
 * footers.  Blocks are never coalesced or reused. Realloc is
 * implemented directly using mm_malloc and mm_free.
 *
 * NOTE TO STUDENTS: Replace this header comment with your own header
 * comment that gives a high level description of your solution.
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <unistd.h>
#include <string.h>

#include "mm.h"
#include "memlib.h"

/*********************************************************
 * NOTE TO STUDENTS: Before you do anything else, please
 * provide your team information in the following struct.
 ********************************************************/
team_t team = {
    /* Team name */
    "ateam",
    /* First member's full name */
    "Harry Bovik",
    /* First member's email address */
    "bovik@cs.cmu.edu",
    /* Second member's full name (leave blank if none) */
    "",
    /* Second member's email address (leave blank if none) */
    ""
```

```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 SIZE_T_SIZE (ALIGN(sizeof(size_t)))

static void *extend_heap(size_t words);
static void *coalesce(void *bp);
static void *find_fit(size_t aszie);
static void place(void *bp, size_t asize);

static char *heap_listp;    /* points to prologue block of heap */

/* Basic constants and macros */
#define WSIZE       4       /* Word and header/footer size (bytes) */
#define DSIZE       8       /* Double word size (bytes) */
#define CHUNKSIZE   (1<<12) /* Extend heap by this amount (bytes) */

#define MAX(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) = (val))

/* Read the size and allocated fields from address p */
#define GET_SIZE(p)     (GET(p) & ~0x7)
#define GET_ALLOC(p)    (GET(p) & 0x1)

/* 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((char *)(bp) - WSIZE))
#define PREV_BLKP(bp)   ((char *)(bp) - GET_SIZE((char *)(bp) - DSIZE))

/*
 * mm_init - initialize the malloc package.
 */
int mm_init(void)
{
    /* Create the initial empty heap */
    if ((heap_listp = mem_sbrk(4*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);

    /* Extend the empty heap with a free block of CHUNKSIZE bytes */
```

```c
    if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
        return -1;
    return 0;
}

/*
 * extend_heap - extends the heap with a new free block.
 *     To maintain alignment, extend_heap rounds up the requested size to
 *     the nearest multiple of 2 words (8 bytes) and then requests the
 *     additional heap space from the memory system.
 */
static void *extend_heap(size_t words)
{
    char *bp;
    size_t size;

    /* Allocate an even number of words to maintain alignment */
    size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
    if ((long)(bp = mem_sbrk(size)) == -1)
        return NULL;

    /* Initialize free block header/footer and the epilogue header */
    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 */

    /* Coalesce if the previous block was free */
    return coalesce(bp);
}

/*
 * mm_malloc - Allocate a block by incrementing the brk pointer.
 *     Always allocate a block whose size is a multiple of the alignment.
 */
void *mm_malloc(size_t size)
{
    size_t asize;       /* Adjusted block size */
    size_t extendsize;  /* Amount to extend heap if no fit */
    char *bp;

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

    /* Adjust block size to include overhead and alignment reqs. */
    if (size <= DSIZE)
        asize = 2*DSIZE;
    else
        asize = DSIZE * ((size + DSIZE + (DSIZE-1)) / DSIZE);

    /* Search the free list for a fit */
    if ((bp = find_fit(asize)) != NULL){        // 找取合适的空闲块
        place(bp, asize);
        return bp;
    }

    /* No fit found. Get more memory and place the block */
    extendsize = MAX(asize, CHUNKSIZE);         // 无合适的空闲块，扩展空间
```

```c
    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
        return NULL;
    place(bp, asize);
    return bp;
}

static void *find_fit(size_t asize){
    char *bp = heap_listp + DSIZE;
    while (GET(HDRP(bp)) != 1){     // 找取合适的空闲块位置，返回其位置指针
        if (!(GET_ALLOC(HDRP(bp))) && (GET_SIZE(HDRP(bp)) >= asize))
            return bp;
        else
            bp = NEXT_BLKP(bp);
    }
    return NULL;
}

static void place(void *bp, size_t asize){
    size_t leftsize = GET_SIZE(HDRP(bp)) - asize;
    if (leftsize >= 2*DSIZE){   // 空闲块的大小大于需求块和最小块的和，对其分割
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        PUT(HDRP(NEXT_BLKP(bp)), PACK(leftsize, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(leftsize, 0));
    } else {                    // 空闲块的大小不大于需求块和最小块的和，直接放置不
分割
        PUT(HDRP(bp), PACK(GET_SIZE(HDRP(bp)), 1));
        PUT(FTRP(bp), PACK(GET_SIZE(HDRP(bp)), 1));
    }
}

/*
 * mm_free - Freeing a block does nothing.
 */
void mm_free(void *bp)
{
    size_t size = GET_SIZE(HDRP(bp));

    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    coalesce(bp);       // 检查释放块前后是否存在空闲块，若存在则与其合并
}

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

    if (prev_alloc && next_alloc){          // 被释放块前面的块和后面的块都是已分
配的
        return bp;
    }
    else if (prev_alloc && !next_alloc){    // 被释放块前面的块是已分配的，后边的
块是空闲的
        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){     // 被释放块前面的块是空闲的，后边的块是已分配的
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        //PUT(FTRP(bp), PACK(size, 0));
        //PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    else {                                   // 被释放块前面的块和后面的块都是空闲的
        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));
        bp = PREV_BLKP(bp);
    }
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    return bp;
}

/*
 * mm_realloc - Implemented simply in terms of mm_malloc and mm_free
 */
void *mm_realloc(void *ptr, size_t size)
{
    size_t cursize;
    char *bp;

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

    if (size == 0){
        mm_free(ptr);
        return NULL;
    }

    cursize = GET_SIZE(HDRP(ptr));
    if (size < cursize) cursize = size;
    bp = mm_malloc(size);
    memcpy(bp, ptr, cursize);
    mm_free(ptr);

    return bp;
}
```

性能评估结果如下：

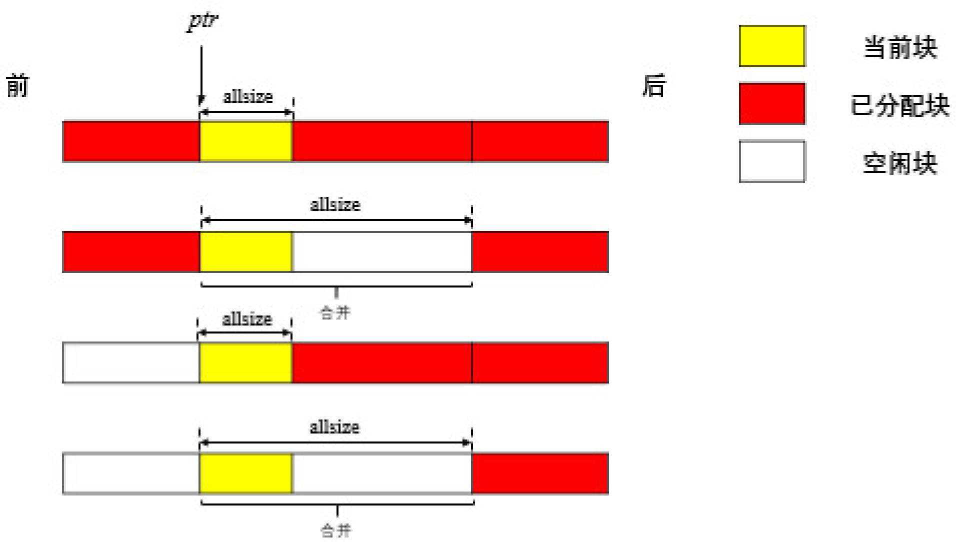
```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.008607   662
 1       yes   99%    5848  0.008104   722
 2       yes   99%    6648  0.013062   509
 3       yes  100%    5380  0.009626   559
 4       yes   66%   14400  0.000251 57348
 5       yes   92%    4800  0.007983   601
 6       yes   92%    4800  0.007509   639
 7       yes   55%   12000  0.140739    85
 8       yes   51%   24000  0.329627    73
 9       yes   27%   14401  0.052593   274
10       yes   34%   14401  0.002100  6859
Total          74%  112372  0.580200   194

Perf index = 44 (util) + 13 (thru) = 57/100
```

隐式空闲链表 + *mm_realloc*函数改进实现

该版本除对 *mm_realloc* 函数进行改动外，其余代码均与前者一致。实现该函数的思路为，当需要重新为已分配空间改变大小时，只需判断要重新分配空间与原始空间的大小关系，若小于原始空间，可进行原地分配，若原始空间不足，再申请另外的空间进行分配，而不必和前面版本一样每次都申请另外的空间进行分配。

其实可以不仅只是将重新分配空间大小与原始空间比较，还可以将原始空间与相邻前面和后面的空闲空间合并后的大小与要重新分配的空间大小进行比较。以下实现了两种合并方案：**方案一**为只将原始空间和后面的空闲空间合并，而不考虑前面的空间是否空闲，该种方案可以减少空间合并后数据拷贝对资源的消耗；**方案二**为将原始空间与前面和后面的空闲空间均进行合并，该种方案在一定程度上可以减少碎片空间，且大概率保证有足够的空间大小被分配，而不必另外申请空间。

- **方案一：原始空间只与后面的空闲空间合并**




```c
/*
 * mm_realloc - Implemented simply in terms of mm_malloc and mm_free
 */
void *mm_realloc(void *ptr, size_t size)
{
    size_t asize, cursize, nextsize, allsize;
    unsigned int nextalloc;
```

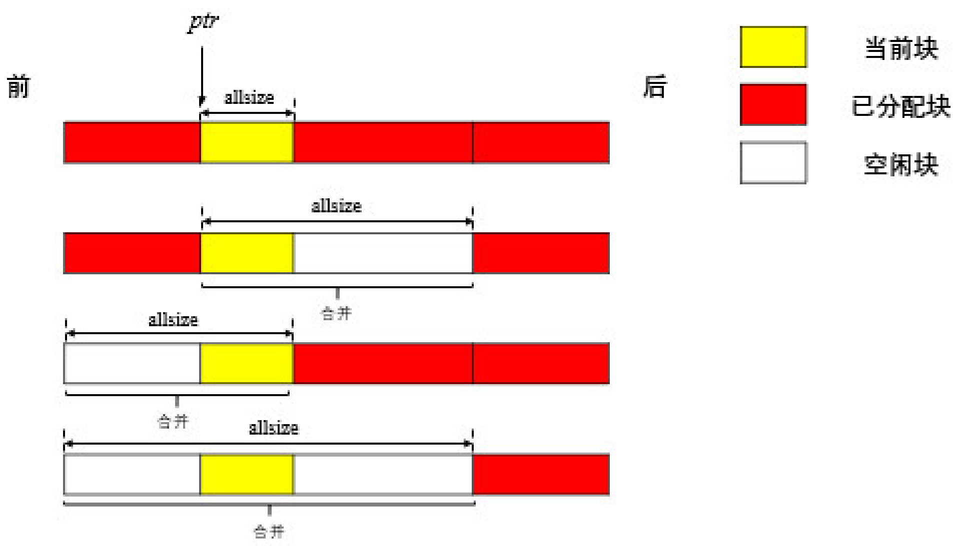
```
    char *bp;

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

    if (size == 0){
        mm_free(ptr);
        return NULL;
    }

    if (size <= DSIZE)
        asize = 2*DSIZE;
    else
        asize = DSIZE * ((size + DSIZE + (DSIZE-1)) / DSIZE);

    cursize = GET_SIZE(HDRP(ptr));
    if (cursize == asize)
        return ptr;

    nextalloc = GET_ALLOC(HDRP(NEXT_BLKP(ptr)));
    nextsize = GET_SIZE(HDRP(NEXT_BLKP(ptr)));

    if (cursize < asize){   // 当前块空间不足
        if (!nextalloc && ((cursize + nextsize) >= asize)) {    // 后面的块是空闲的，且两者空间之和足够分配
            allsize = cursize + nextsize;   // 两者合并的总空间
            if ((allsize - asize) >= 2 * DSIZE) {   // 分配后剩余空间大于最小块空间，进行分割
                PUT(HDRP(ptr), PACK(asize, 1));
                PUT(FTRP(ptr), PACK(asize, 1));
                PUT(HDRP(NEXT_BLKP(ptr)), PACK(allsize - asize, 0));
                PUT(FTRP(NEXT_BLKP(ptr)), PACK(allsize - asize, 0));
            }else {                                 // 分配后剩余空间不足以分配最小块空间，直接分配总的空间
                PUT(HDRP(ptr), PACK(allsize, 1));
                PUT(FTRP(ptr), PACK(allsize, 1));
            }
        }else {     // 后面的块是已分配的。原地分配空间不足，需另外申请空间，且注意数据的拷贝
            bp = mm_malloc(asize);
            memcpy(bp, ptr, cursize);
            mm_free(ptr);
            return bp;
        }
        }else {         // 当前块空间足够
        if(!nextalloc){         // 后面的块是空闲的，合并两个空间，并在分配后再进行分割
            allsize = cursize + nextsize;
            PUT(HDRP(ptr), PACK(asize, 1));
            PUT(FTRP(ptr), PACK(asize, 1));
            PUT(HDRP(NEXT_BLKP(ptr)), PACK(allsize - asize, 0));
            PUT(FTRP(NEXT_BLKP(ptr)), PACK(allsize - asize, 0));
        }else {
            if ((cursize-asize) >= 2*DSIZE){    // 分配后剩余空间大于最小块空间，进行分割
                PUT(HDRP(ptr), PACK(asize, 1));
                PUT(FTRP(ptr), PACK(asize, 1));
                PUT(HDRP(NEXT_BLKP(ptr)), PACK(cursize-asize, 0));
```

```
                PUT(FTRP(NEXT_BLKP(ptr)), PACK(cursize-asize, 0));
            }
        }
    }

    return ptr;
}
```

性能评估结果如下:

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.008584   663
 1       yes   99%    5848  0.008056   726
 2       yes   99%    6648  0.012952   513
 3       yes  100%    5380  0.009685   555
 4       yes   66%   14400  0.000249 57855
 5       yes   92%    4800  0.007985   601
 6       yes   92%    4800  0.007538   637
 7       yes   55%   12000  0.141856    85
 8       yes   51%   24000  0.332280    72
 9       yes   92%   14401  0.000272 52984
10       yes   86%   14401  0.000220 65429
Total          85%  112372  0.529679   212

Perf index = 51 (util) + 14 (thru) = 65/100
```

- **方案二：原始空间与后面的和前面的空闲空间合并**




```
/*
 * mm_realloc - Implemented simply in terms of mm_malloc and mm_free
 */
void *mm_realloc(void *ptr, size_t size)
{
    char *bp;
    size_t asize, cursize, presize, nextsize, allsize;
    unsigned int prealloc, nextalloc, flag;

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

```
    if (size == 0){
        mm_free(ptr);
        return NULL;
    }

    if (size <= DSIZE)
        asize = 2*DSIZE;
    else
        asize = DSIZE * ((size + DSIZE + (DSIZE-1)) / DSIZE);

    cursize = GET_SIZE(HDRP(ptr));
    if (cursize == asize)
        return ptr;

    prealloc = GET_ALLOC(HDRP(PREV_BLKP(ptr)));
    nextalloc = GET_ALLOC(HDRP(NEXT_BLKP(ptr)));
    presize = GET_SIZE(HDRP(PREV_BLKP(ptr)));
    nextsize = GET_SIZE(HDRP(NEXT_BLKP(ptr)));
    flag = (prealloc << 1) | nextalloc;

    if (cursize > asize){           // 当前块空间足够
        switch(flag){
            case 0:     // 前面的块和后面的块都是空闲的
                allsize = presize + cursize + nextsize;
                bp = PREV_BLKP(ptr);
                memcpy(bp, ptr, asize);
                break;
            case 1:     // 前面的块是空闲的，后面的块是已分配的
                allsize = presize + cursize;
                bp = PREV_BLKP(ptr);
                memcpy(bp, ptr, asize);
                break;
            case 2:     // 前面的块已分配的，后面的块是空闲的
                allsize = cursize + nextsize;
                bp = ptr;
                break;
            case 3:     // 前面的块和后面的块都是已分配的
                allsize = cursize;
                if ((cursize - asize) < 2*DSIZE){
                    PUT(HDRP(ptr), PACK(allsize, 1));
                    PUT(FTRP(ptr), PACK(allsize, 1));
                    return ptr;
                }
                break;
            default:
                printf("mm_realloc error!\n");
        }
        // 总空间足够大，分割块
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        PUT(HDRP(NEXT_BLKP(bp)), PACK(allsize-asize, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(allsize-asize, 0));
    } else {                        // 当前块空间不足
        switch(flag){
            case 0:     // 前面的块和后面的块都是空闲的
                allsize = presize + cursize + nextsize;
                bp = PREV_BLKP(ptr);
                break;
```

```
            case 1:     // 前面的块是空闲的，后面的块是已分配的
                allsize = presize + cursize;
                bp = PREV_BLKP(ptr);
                break;
            case 2:     // 前面的块已分配的，后面的块是空闲的
                allsize = cursize + nextsize;
                bp = ptr;
                break;
            case 3:     // 前面的块和后面的块都是已分配的
                allsize = cursize;
                bp = ptr;
                break;
            default:
            printf("mm_realloc error!\n");
        }
        if (allsize < asize){   // 总空间不足，不能原地分配，需另外申请空间
            bp = mm_malloc(size);
            memcpy(bp, ptr, cursize);
            mm_free(ptr);
        } else {                // 总空间足够分配
            //memcpy(bp, ptr, cursize);     // !!! 特别注意：此处拷贝空间应使用
memmove函数，而不能使用memcpy函数，
            memmove(bp, ptr, cursize);      // 因拷贝原数据地址可能和目的地址相重
叠，使用memcpy函数可能不能正确拷贝原始数据。
            if ((allsize - asize) >= 2*DSIZE){  // 剩余空间大于最小块空间，分割
块
                PUT(HDRP(bp), PACK(asize, 1));
                PUT(FTRP(bp), PACK(asize, 1));
                PUT(HDRP(NEXT_BLKP(bp)), PACK(allsize-asize, 0));
                PUT(FTRP(NEXT_BLKP(bp)), PACK(allsize-asize, 0));
            } else {                        // 剩余空间小于最小块空间，不分
割
                PUT(HDRP(bp), PACK(allsize, 1));
                PUT(FTRP(bp), PACK(allsize, 1));
            }
        }
    }

    return bp;
}
```

性能评估结果如下：

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.008611   661
 1       yes   99%    5848  0.008075   724
 2       yes   99%    6648  0.012987   512
 3       yes  100%    5380  0.009707   554
 4       yes   66%   14400  0.000249 57855
 5       yes   92%    4800  0.008388   572
 6       yes   92%    4800  0.007949   604
 7       yes   55%   12000  0.146758    82
 8       yes   51%   24000  0.333182    72
 9       yes   44%   14401  0.037300   386
10       yes   45%   14401  0.001157 12447
Total          77%  112372  0.574363   196

Perf index = 46 (util) + 13 (thru) = 59/100
```

从上结果可以看出，方案一较方案二占优，且实现代码更为简单。

显式空闲链表 + *mm_realloc*函数普通实现

该版本中采用双向链表的结构连接空闲块，固定链表头的位置（程序中为指针 *heap_listp*），以某个节点的后继值为零代表指向链表尾。任何链表插入删除等操作中都应注意，很容易出错，特别是应注意判断当前操作节点后继是否为空，以执行不同的指针连接操作。还有在设置堆块大小及标志位和设置其前后指针时注意相应的操作顺序，当心原堆块数据被覆盖。

完整代码如下：

```c
/*
 * mm-naive.c - The fastest, least memory-efficient malloc package.
 *
 * In this naive approach, a block is allocated by simply incrementing
 * the brk pointer.  A block is pure payload. There are no headers or
 * footers.  Blocks are never coalesced or reused. Realloc is
 * implemented directly using mm_malloc and mm_free.
 *
 * NOTE TO STUDENTS: Replace this header comment with your own header
 * comment that gives a high level description of your solution.
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <unistd.h>
#include <string.h>

#include "mm.h"
#include "memlib.h"

/*********************************************************
 * NOTE TO STUDENTS: Before you do anything else, please
 * provide your team information in the following struct.
 ********************************************************/
team_t team = {
    /* Team name */
    "ateam",
    /* First member's full name */
    "Harry Bovik",
```

```c
    /* First member's email address */
    "bovik@cs.cmu.edu",
    /* Second member's full name (leave blank if none) */
    "",
    /* Second member's email address (leave blank if none) */
    ""
};

/* 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 SIZE_T_SIZE (ALIGN(sizeof(size_t)))


static void *extend_heap(size_t words);
static void *coalesce(void *bp);
static void *find_fit(size_t aszie);
static void place(void *bp, size_t asize);

static char *heap_listp;    /* points to prologue block of heap */

/* Basic constants and macros */
#define WSIZE       4       /* Word and header/footer size (bytes) */
#define DSIZE       8       /* Double word size (bytes) */
#define CHUNKSIZE   (1<<12) /* Extend heap by this amount (bytes) */

#define MAX(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) = (val))

/* Read the size and allocated fields from address p */
#define GET_SIZE(p)     (GET(p) & ~0x7)
#define GET_ALLOC(p)    (GET(p) & 0x1)

/* 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((char *)(bp) - WSIZE))
#define PREV_BLKP(bp)   ((char *)(bp) - GET_SIZE((char *)(bp) - DSIZE))

#define GET_PRED(p)         (GET(p))
#define PUT_PRED(p, val)    (PUT(p, val))
#define GET_SUCC(p)         (*(unsigned int *)((char *)(p) + WSIZE))
#define PUT_SUCC(p, val)    (*(unsigned int *)((char *)(p) + WSIZE) =
(val))

/*
```

```
 * mm_init - initialize the malloc package.
 */
int mm_init(void)
{
    /* Create the initial empty heap */
    if ((heap_listp = mem_sbrk(8*WSIZE)) == (void *)-1)
        return -1;
    PUT(heap_listp, 0);                              /* ALignment padding */
    PUT(heap_listp + (1*WSIZE), PACK(2*DSIZE, 1));  /* Explicit Free Lists
header */
    PUT(heap_listp + (2*WSIZE), 0);                 /* Explicit Free Lists
pred pointer */
    PUT(heap_listp + (3*WSIZE), 0);                 /* Explicit Free Lists
succ pointer */
    PUT(heap_listp + (4*WSIZE), PACK(2*DSIZE, 1));  /* Explicit Free Lists
footer */
    PUT(heap_listp + (5*WSIZE), PACK(DSIZE, 1));    /* Epilogue header */
    PUT(heap_listp + (6*WSIZE), PACK(DSIZE, 1));    /* Prologue footer */
    PUT(heap_listp + (7*WSIZE), PACK(0, 1));        /* Epilogue header */
    heap_listp += (2*WSIZE);

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

/*
 * extend_heap - extends the heap with a new free block.
 *     To maintain alignment, extend_heap rounds up the requested size to
 *     the nearest multiple of 2 words (8 bytes) and then requests the
 *     additional heap space from the memory system.
 */
static void *extend_heap(size_t words)
{
    char *bp;
    size_t size;

    /* Allocate an even number of words to maintain alignment */
    size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
    if ((long)(bp = mem_sbrk(size)) == -1)
        return NULL;

    //printf("  --- extend_heap: %d\n", size);
    /* Initialize free block header/footer and the epilogue header */
    PUT(HDRP(bp), PACK(size, 0));        /* Free block header */
    PUT(FTRP(bp), PACK(size, 0));        /* Free block footer */
    PUT_PRED(bp, 0);                     /* Free block pred pointer */
    PUT_SUCC(bp, 0);                     /* Free block succ pointer */
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));    /* New epilogue header */

    /* Coalesce if the previous block was free */
    if (!GET_ALLOC(HDRP(PREV_BLKP(bp)))){        // 若前面的块是空闲的，当前分配
块与前面的块合并，该空闲块在空闲链表上的位置不变，大小改变
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(PREV_BLKP(bp)), PACK(size, 0));
        return PREV_BLKP(bp);
```

```
    }
    // 若前面的块是已分配的，将当前分配块插入在空闲链表首位，注意判断空闲链表是否为空
    if (GET_SUCC(heap_listp) == 0){
        PUT_PRED(bp, heap_listp);
        PUT_SUCC(bp, 0);
        PUT_SUCC(heap_listp, bp);
    } else {
        PUT_PRED(bp, heap_listp);
        PUT_SUCC(bp, GET_SUCC(heap_listp));
        PUT_PRED(GET_SUCC(heap_listp), bp);
        PUT_SUCC(heap_listp, bp);
    }
    return bp;
}

/*
 * mm_malloc - Allocate a block by incrementing the brk pointer.
 *     Always allocate a block whose size is a multiple of the alignment.
 */
void *mm_malloc(size_t size)
{
    size_t asize;   /* Adjusted block size */
    size_t extendsize;  /* Amount to extend heap if no fit */
    char *bp;

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

    /* Adjust block size to include overhead and alignment reqs. */
    if (size <= DSIZE)
        asize = 2*DSIZE;
    else
        asize = DSIZE * ((size + DSIZE + (DSIZE-1)) / DSIZE);

    //displayEmptylist();
    //printf("\n>>> mm_malloc size: %d\n", asize);
    /* Search the free list for a fit */
    if ((bp = find_fit(asize)) != NULL){
        //printf("  --- find_fit: %p\n", bp);
        place(bp, asize);
        return bp;
    }

    /* No fit found. Get more memory and place the block */
    extendsize = MAX(asize, CHUNKSIZE);
    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
        return NULL;
    //printf("  --- extend_heap: %p\n", bp);
    place(bp, asize);
    return bp;
}

static void *find_fit(size_t asize){
    //printf("+ find_fit\n");
    char *bp = GET_SUCC(heap_listp);
    while (bp != 0){         // 搜索空闲链表，首次适配方式
        if (GET_SIZE(HDRP(bp)) >= asize)
```

```
            return bp;
        else
            bp = GET_SUCC(bp);
    }
    return NULL;
}

static void place(void *bp, size_t asize){
    //printf("+ place: %p\n", bp);
    size_t leftsize = GET_SIZE(HDRP(bp)) - asize;
    if (leftsize >= 2*DSIZE){   // 剩余空间大于最小块空间，分割空闲块；剩余空闲块替
换原空闲块在空闲链表中的位置
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        PUT(HDRP(NEXT_BLKP(bp)), PACK(leftsize, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(leftsize, 0));
        PUT_PRED(NEXT_BLKP(bp), GET_PRED(bp));
        PUT_SUCC(NEXT_BLKP(bp), GET_SUCC(bp));
        PUT_SUCC(GET_PRED(bp), NEXT_BLKP(bp));
        if (GET_SUCC(bp) != 0)
            PUT_PRED(GET_SUCC(bp), NEXT_BLKP(bp));
    } else {                    // 剩余空间小于最小块空间，不进行分割；将原空闲块在
空闲链表中删除
        PUT(HDRP(bp), PACK(GET_SIZE(HDRP(bp)), 1));
        PUT(FTRP(bp), PACK(GET_SIZE(HDRP(bp)), 1));
        PUT_SUCC(GET_PRED(bp), GET_SUCC(bp));
        if (GET_SUCC(bp) != 0)
            PUT_PRED(GET_SUCC(bp), GET_PRED(bp));
    }
}


/*
 * mm_free - Freeing a block does nothing.
 */
void mm_free(void *bp)
{
    size_t size = GET_SIZE(HDRP(bp));

    //displayEmptylist();
    //printf("\n>>> mm_free: %p\n", bp);
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    coalesce(bp);
}

static void *coalesce(void *bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));
    //printf(" > coalesce\n");

    if (prev_alloc && next_alloc){          // 前面的块和后面的块都是已分配的；将
该空闲块插入到空闲链表首位
        //printf("  --- coalesce: %p\n", bp);
        if (GET_SUCC(heap_listp) == 0){
            PUT_PRED(bp, heap_listp);
```

```
            PUT_SUCC(bp, 0);
            PUT_SUCC(heap_listp, bp);
        } else {
            PUT_PRED(bp, heap_listp);
            PUT_SUCC(bp, GET_SUCC(heap_listp));
            PUT_PRED(GET_SUCC(heap_listp), bp);
            PUT_SUCC(heap_listp, bp);
        }
        return bp;
    }
    else if (prev_alloc && !next_alloc){    // 前面的块已分配的，后面的块是空闲
的；将该空闲块替换其后面的空闲块在空闲链表中的位置，并将两者合并
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        if (GET_SUCC(NEXT_BLKP(bp)) != 0)
            PUT_PRED(GET_SUCC(NEXT_BLKP(bp)), bp);
        PUT_SUCC(GET_PRED(NEXT_BLKP(bp)), bp);
        PUT_PRED(bp, GET_PRED(NEXT_BLKP(bp)));
        PUT_SUCC(bp, GET_SUCC(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    }
    else if (!prev_alloc && next_alloc){    // 前面的块是空闲的，后面的块是已分配
的；前面的块在空闲链表的位置不变，并将两者合并
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    else {                                  // 前面的块和后面的块都是空闲的；前面
的块在空闲链表的位置不变，将后面的块从空闲链表中删除，并合并这三个空闲块
        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));
        if (GET_SUCC(NEXT_BLKP(bp)) != 0)
            PUT_PRED(GET_SUCC(NEXT_BLKP(bp)), GET_PRED(NEXT_BLKP(bp)));
        PUT_SUCC(GET_PRED(NEXT_BLKP(bp)), GET_SUCC(NEXT_BLKP(bp)));
        bp = PREV_BLKP(bp);
    }

    //printf("  --- coalesce: %p\n", bp);
    return bp;
}

/*
 * mm_realloc - Implemented simply in terms of mm_malloc and mm_free
 */
void *mm_realloc(void *ptr, size_t size)
{
    size_t oldsize;
    char *bp;
    //printf("\n>>> mm_realloc: %p\n", ptr);

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

    if (size == 0){
        mm_free(ptr);
```

```
        return NULL;
    }

    oldsize = GET_SIZE(HDRP(ptr));
    if (size < oldsize) oldsize = size;
    bp = mm_malloc(size);
    memcpy(bp, ptr, oldsize);
    mm_free(ptr);

    return bp;
}

// 打印空闲链表，验证程序用
void displayEmptylist(){
    printf(" > displayEmptylist\n");
    char *bp = GET_SUCC(heap_listp);
    printf("heaplist");
    while (bp != 0){
    printf(" -> %p", bp);
    bp = GET_SUCC(bp);
    }
    printf("\n");
}
```

性能评估结果如下：

Results for mm malloc:

trace	valid	util	ops	secs	Kops
0	yes	94%	5694	0.000243	23432
1	yes	95%	5848	0.000200	29182
2	yes	96%	6648	0.000300	22160
3	yes	98%	5380	0.000218	24713
4	yes	66%	14400	0.000311	46332
5	yes	92%	4800	0.000810	5928
6	yes	88%	4800	0.000807	5947
7	yes	55%	12000	0.008466	1418
8	yes	51%	24000	0.004719	5086
9	yes	27%	14401	0.055633	259
10	yes	34%	14401	0.002155	6683
Total		72%	112372	0.073861	1521

Perf index = 43 (util) + 40 (thru) = 83/100

显式空闲链表 + *mm_realloc*函数改进实现

该版本处 *mm_realloc* 函数外其余代码与前者相同，从版本二中可以看出，*mm_realloc* 函数方案一的实现方式较优，所以该版本 *mm_realloc* 函数的实现思路与之相同，代码如下：

```
/*
 * mm_realloc - Implemented simply in terms of mm_malloc and mm_free
 */
void *mm_realloc(void *ptr, size_t size)
{
    size_t asize, cursize, nextsize, allsize;
    unsigned int nextalloc;
    char *bp;
    //displayEmptylist();
```

```c
    //printf("\n>>> mm_realloc: %p\n", ptr);

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

    if (size == 0){
        mm_free(ptr);
        return NULL;
    }

    if (size <= DSIZE)
        asize = 2*DSIZE;
    else
        asize = DSIZE * ((size + DSIZE + (DSIZE-1)) / DSIZE);

    //printf("\n>>> mm_realloc asize: %d\n", asize);
    cursize = GET_SIZE(HDRP(ptr));
    if (cursize == asize)
        return ptr;

    nextalloc = GET_ALLOC(HDRP(NEXT_BLKP(ptr)));
    nextsize = GET_SIZE(HDRP(NEXT_BLKP(ptr)));

    if (cursize < asize){    // 当前块空间不足
        if (!nextalloc && ((cursize + nextsize) >= asize)){ // 后面的块是空闲的，且两者空间之和足够分配
            allsize = cursize + nextsize;    // 两者合并的总空间
            if ((allsize - asize) >= 2*DSIZE){  // 分配后剩余空间大于最小块空间，进行分割；剩余空闲块替换原后面的空闲块在空闲链表中的位置（此处应注意设置好相关块的前后指针，且注意堆块大小及分配标志位和相关块的前后指针设置顺序，防止取的覆盖原数据后的值）
                bp = NEXT_BLKP(ptr);
                PUT(HDRP(ptr), PACK(asize, 1)); // 注：代码顺序很重要！！
                PUT_PRED(NEXT_BLKP(ptr), GET_PRED(bp));
                PUT_SUCC(NEXT_BLKP(ptr), GET_SUCC(bp));
                if (GET_SUCC(bp) != 0)
                    PUT_PRED(GET_SUCC(bp), NEXT_BLKP(ptr));
                PUT_SUCC(GET_PRED(bp), NEXT_BLKP(ptr));
                PUT(FTRP(ptr), PACK(asize, 1));
                PUT(HDRP(NEXT_BLKP(ptr)), PACK(allsize-asize, 0));
                PUT(FTRP(NEXT_BLKP(ptr)), PACK(allsize-asize, 0));
            } else {     // 分配后剩余空间不足以分配最小块空间，直接分配总的空间；从空闲链表中删除原后面的空闲块
                bp = NEXT_BLKP(ptr);
                if (GET_SUCC(bp) != 0)
                    PUT_PRED(GET_SUCC(bp), GET_PRED(bp));
                PUT_SUCC(GET_PRED(bp), GET_SUCC(bp));
                PUT(HDRP(ptr), PACK(allsize, 1));
                PUT(FTRP(ptr), PACK(allsize, 1));
            }
        } else {        // 后面的块是已分配的。原地分配空间不足，需另外申请空间，且注意数据的拷贝
            bp = mm_malloc(asize);
            memcpy(bp, ptr, cursize);
            mm_free(ptr);
            //printf("  --- mm_realloc: %p\n", bp);
            return bp;
        }
    } else {    // 当前块空间足够
```

```
        if (!nextalloc){    // 后面的块是空闲的，合并两个空间，并在分配后再进行分割；剩余空闲块取代原后面的空闲块在空闲链表中的位置
            bp = NEXT_BLKP(ptr);
            allsize = cursize + nextsize;
            PUT(HDRP(ptr), PACK(asize, 1));
            PUT(FTRP(ptr), PACK(asize, 1));
            PUT(HDRP(NEXT_BLKP(ptr)), PACK(allsize-asize, 0));
            PUT(FTRP(NEXT_BLKP(ptr)), PACK(allsize-asize, 0));
            PUT_PRED(NEXT_BLKP(ptr), GET_PRED(bp));
            PUT_SUCC(NEXT_BLKP(ptr), GET_SUCC(bp));
            if (GET_SUCC(bp) != 0)
                PUT_PRED(GET_SUCC(bp), NEXT_BLKP(ptr));
            PUT_SUCC(GET_PRED(bp), NEXT_BLKP(ptr));
        } else {         // 后面的块是已分配的
            allsize = cursize;
            if ((cursize - asize) >= 2*DSIZE){  // 分配后剩余空间大于最小块空间，进行分割；剩余空闲块插入到空闲链表头部
                PUT(HDRP(ptr), PACK(asize, 1));
                PUT(FTRP(ptr), PACK(asize, 1));
                PUT(HDRP(NEXT_BLKP(ptr)), PACK(allsize-asize, 0));
                PUT(FTRP(NEXT_BLKP(ptr)), PACK(allsize-asize, 0));
                bp = NEXT_BLKP(ptr);
                if (GET_SUCC(heap_listp) == 0){
                    PUT_PRED(bp, heap_listp);
                    PUT_SUCC(bp, 0);
                    PUT_SUCC(heap_listp, bp);
                } else {
                    PUT_PRED(bp, heap_listp);
                    PUT_SUCC(bp, GET_SUCC(heap_listp));
                    PUT_PRED(GET_SUCC(heap_listp), bp);
                    PUT_SUCC(heap_listp, bp);
                }
            }
        }
    }

    //printf("  --- mm_realloc: %p\n", ptr);
    return ptr;
}
```

性能评估结果如下：

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   94%    5694  0.000244 23365
 1       yes   95%    5848  0.000203 28808
 2       yes   96%    6648  0.000286 23253
 3       yes   98%    5380  0.000211 25486
 4       yes   66%   14400  0.000310 46452
 5       yes   92%    4800  0.000788  6093
 6       yes   88%    4800  0.000794  6047
 7       yes   55%   12000  0.008444  1421
 8       yes   51%   24000  0.004742  5061
 9       yes   92%   14401  0.000318 45329
10       yes   86%   14401  0.000274 52654
Total          83%  112372  0.016613  6764

Perf index = 50 (util) + 40 (thru) = 90/100
```