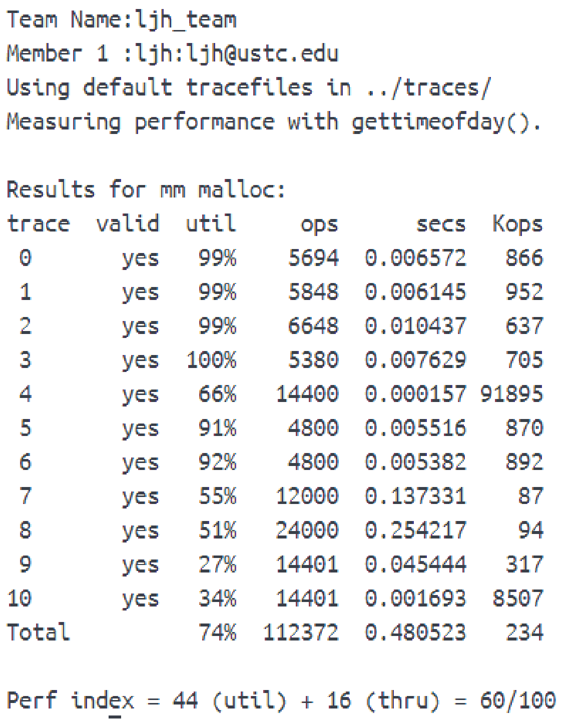
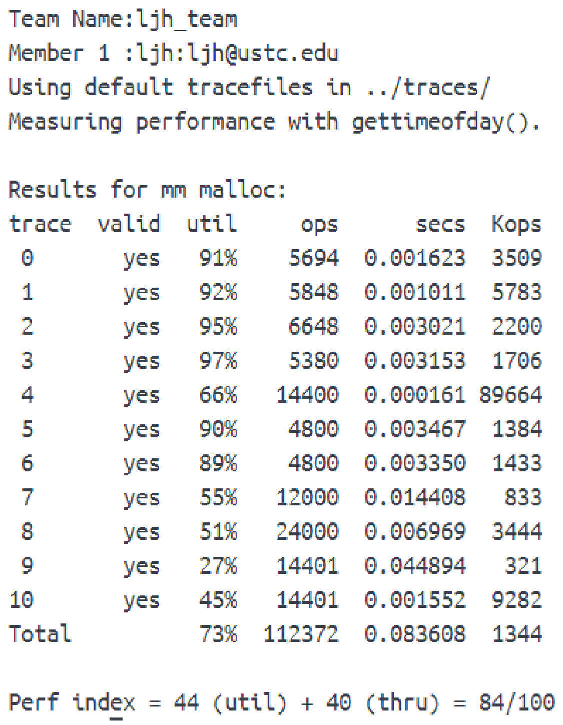
malloc_lab

本实验要求自己实现一个 malloc, free, realloc 的函数. 官网已经给了 implicit list, first fit 和 implicit, next fit 的代码, 直接使用该源代码进行测试可以得到:



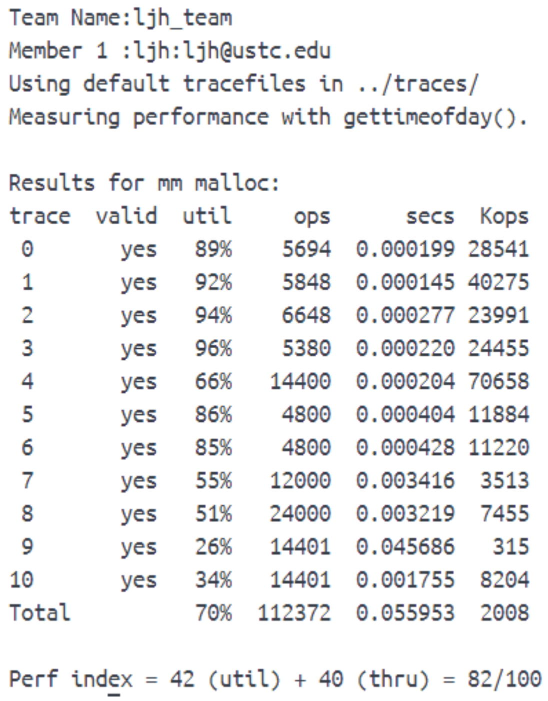
```
Team Name:ljh_team
Member 1 :ljh:ljh@ustc.edu
Using default tracefiles in ../traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.006572   866
 1       yes   99%    5848  0.006145   952
 2       yes   99%    6648  0.010437   637
 3       yes  100%    5380  0.007629   705
 4       yes   66%   14400  0.000157 91895
 5       yes   91%    4800  0.005516   870
 6       yes   92%    4800  0.005382   892
 7       yes   55%   12000  0.137331    87
 8       yes   51%   24000  0.254217    94
 9       yes   27%   14401  0.045444   317
10       yes   34%   14401  0.001693  8507
Total          74%  112372  0.480523   234

Perf index = 44 (util) + 16 (thru) = 60/100
```



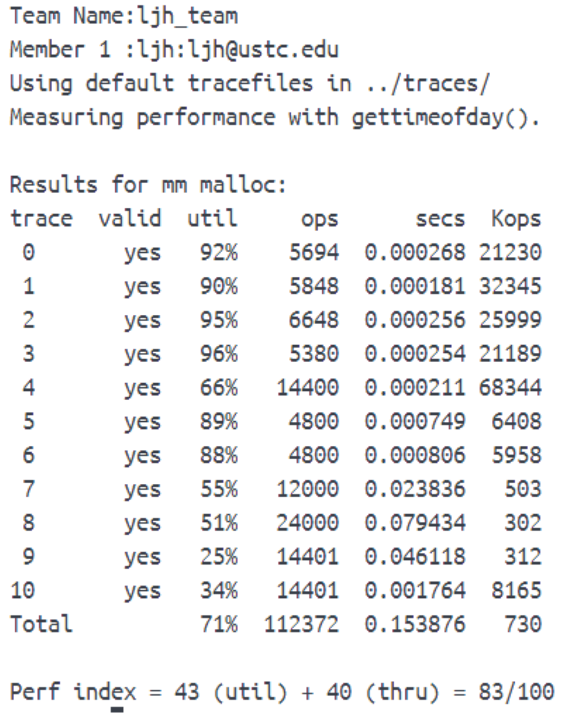
```
Team Name:ljh_team
Member 1 :ljh:ljh@ustc.edu
Using default tracefiles in ../traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   91%    5694  0.001623  3509
 1       yes   92%    5848  0.001011  5783
 2       yes   95%    6648  0.003021  2200
 3       yes   97%    5380  0.003153  1706
 4       yes   66%   14400  0.000161 89664
 5       yes   90%    4800  0.003467  1384
 6       yes   89%    4800  0.003350  1433
 7       yes   55%   12000  0.014408   833
 8       yes   51%   24000  0.006969  3444
 9       yes   27%   14401  0.044894   321
10       yes   45%   14401  0.001552  9282
Total          73%  112372  0.083608  1344

Perf index = 44 (util) + 40 (thru) = 84/100
```

左图为 implicit list, first fit 的运行结果, 可以看出由于每次需要头遍历每一个 block 而且不跳过 allocated block 所以效率非常低下. 右图虽然也是 implicit list, 但是每次寻找 free block 时会从上次找的的 free block 向后寻找, 效率相较于前者要更快一些.

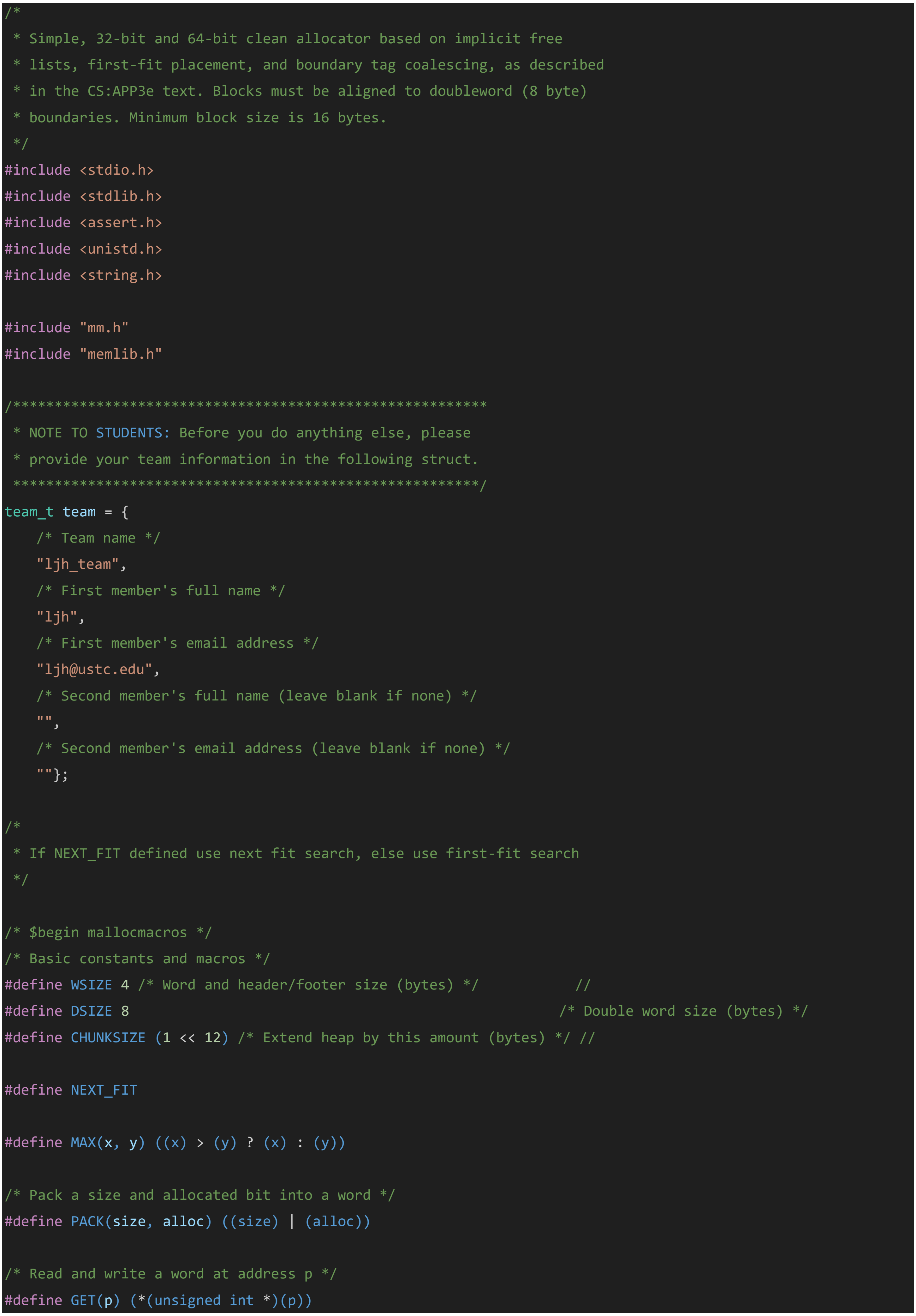
本实验我通过修改教材的 implicit list 实现的源代码, 实现了 explicit list, first fit 和 explicit list, next fit 的实现, 以下是实现的结果和源代码



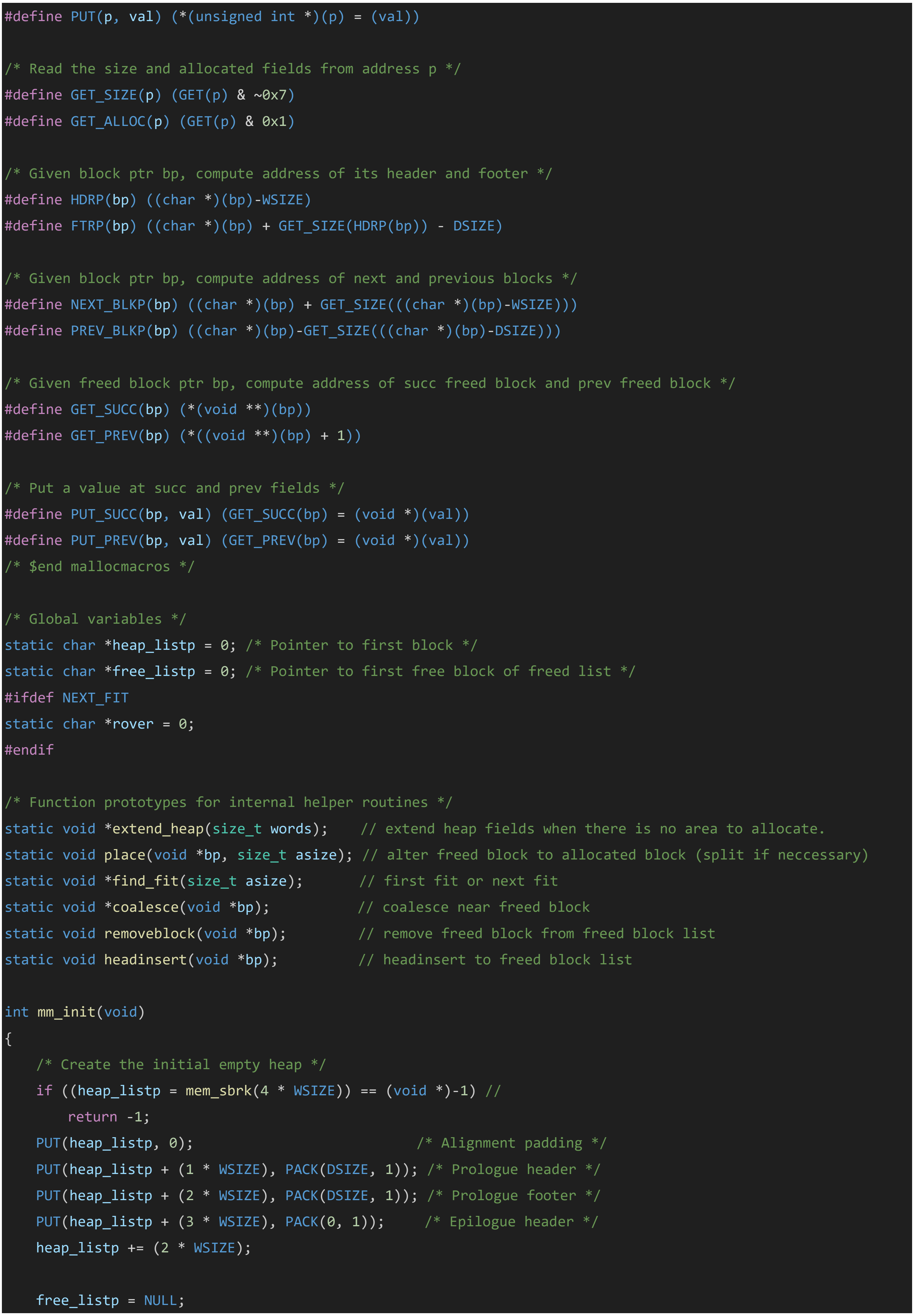
```
Team Name:ljh_team
Member 1 :ljh:ljh@ustc.edu
Using default tracefiles in ../traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   89%    5694  0.000199 28541
 1       yes   92%    5848  0.000145 40275
 2       yes   94%    6648  0.000277 23991
 3       yes   96%    5380  0.000220 24455
 4       yes   66%   14400  0.000204 70658
 5       yes   86%    4800  0.000404 11884
 6       yes   85%    4800  0.000428 11220
 7       yes   55%   12000  0.003416  3513
 8       yes   51%   24000  0.003219  7455
 9       yes   26%   14401  0.045686   315
10       yes   34%   14401  0.001755  8204
Total          70%  112372  0.055953  2008

Perf index = 42 (util) + 40 (thru) = 82/100
```



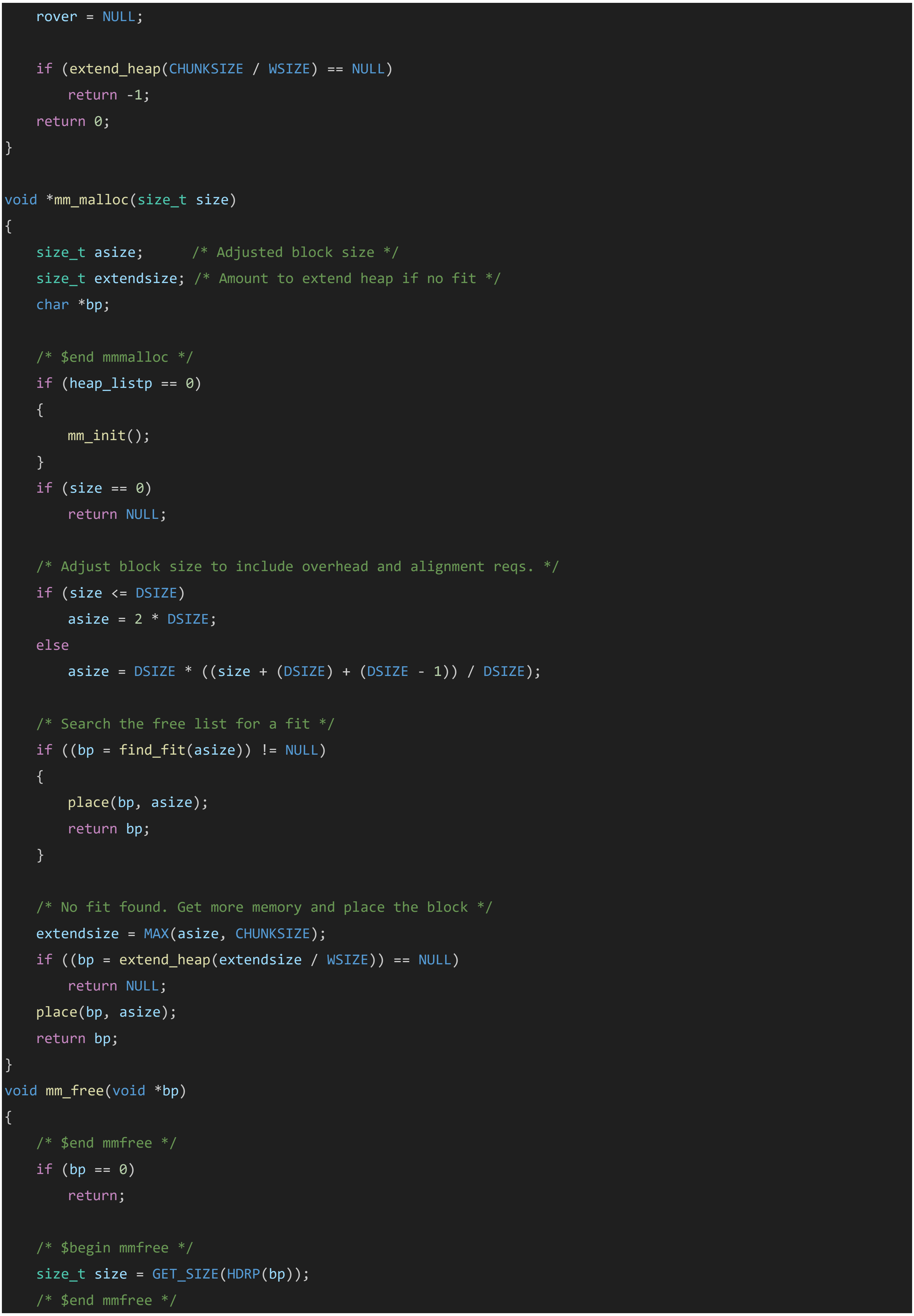
```
Team Name:ljh_team
Member 1 :ljh:ljh@ustc.edu
Using default tracefiles in ../traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   92%    5694  0.000268 21230
 1       yes   90%    5848  0.000181 32345
 2       yes   95%    6648  0.000256 25999
 3       yes   96%    5380  0.000254 21189
 4       yes   66%   14400  0.000211 68344
 5       yes   89%    4800  0.000749  6408
 6       yes   88%    4800  0.000806  5958
 7       yes   55%   12000  0.023836   503
 8       yes   51%   24000  0.079434   302
 9       yes   25%   14401  0.046118   312
10       yes   34%   14401  0.001764  8165
Total          71%  112372  0.153876   730

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



```c
/*
 * Simple, 32-bit and 64-bit clean allocator based on implicit free
 * lists, first-fit placement, and boundary tag coalescing, as described
 * in the CS:APP3e text. Blocks must be aligned to doubleword (8 byte)
 * boundaries. Minimum block size is 16 bytes.
 */
#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 */
    "ljh_team",
    /* First member's full name */
    "ljh",
    /* First member's email address */
    "ljh@ustc.edu",
    /* Second member's full name (leave blank if none) */
    "",
    /* Second member's email address (leave blank if none) */
    ""};

/*
 * If NEXT_FIT defined use next fit search, else use first-fit search
 */

/* $begin mallocmacros */
/* 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 NEXT_FIT

#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))
```



```c
#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)))

/* Given freed block ptr bp, compute address of succ freed block and prev freed block */
#define GET_SUCC(bp) (*(void **)(bp))
#define GET_PREV(bp) (*((void **)(bp) + 1))

/* Put a value at succ and prev fields */
#define PUT_SUCC(bp, val) (GET_SUCC(bp) = (void *)(val))
#define PUT_PREV(bp, val) (GET_PREV(bp) = (void *)(val))
/* $end mallocmacros */

/* Global variables */
static char *heap_listp = 0; /* Pointer to first block */
static char *free_listp = 0; /* Pointer to first free block of freed list */
#ifdef NEXT_FIT
static char *rover = 0;
#endif

/* Function prototypes for internal helper routines */
static void *extend_heap(size_t words);    // extend heap fields when there is no area to allocate.
static void place(void *bp, size_t asize); // alter freed block to allocated block (split if neccessary)
static void *find_fit(size_t asize);       // first fit or next fit
static void *coalesce(void *bp);           // coalesce near freed block
static void removeblock(void *bp);         // remove freed block from freed block list
static void headinsert(void *bp);          // headinsert to freed block list

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);

    free_listp = NULL;
```



```
    rover = NULL;

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

void *mm_malloc(size_t size)
{
    size_t asize;      /* Adjusted block size */
    size_t extendsize; /* Amount to extend heap if no fit */
    char *bp;

    /* $end mmmalloc */
    if (heap_listp == 0)
    {
        mm_init();
    }
    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)
    {
        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;
    place(bp, asize);
    return bp;
}
void mm_free(void *bp)
{
    /* $end mmfree */
    if (bp == 0)
        return;

    /* $begin mmfree */
    size_t size = GET_SIZE(HDRP(bp));
    /* $end mmfree */
    if (heap_listp == 0)
    {
        mm_init();
    }
    /* $begin mmfree */

    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    PUT_PREV(bp, NULL); // ex added.
    PUT_SUCC(bp, NULL);

    bp = coalesce(bp);
    headinsert(bp);
    return 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));

    if (prev_alloc && next_alloc)
    {
    }

    else if (prev_alloc && !next_alloc)
    {
        removeblock(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)
    {
        removeblock(PREV_BLKP(bp));
        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
    {
        removeblock(PREV_BLKP(bp));
        removeblock(NEXT_BLKP(bp));
        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);
    }
#ifdef NEXT_FIT
    if ((rover > (char *)bp) && (rover < NEXT_BLKP(bp))) // adjust where rover point
        rover = bp;
#endif
    return bp;
}

/*
 * mm_realloc - Naive implementation of realloc
 */
void *mm_realloc(void *ptr, size_t newsize)
{
    size_t oldsize;
    void *newptr;

    if (newsize == 0)
    {
        mm_free(ptr);
        return 0;
    }

    if (ptr == NULL)
    {
        return mm_malloc(newsize);
    }

    newptr = mm_malloc(newsize);

    if (!newptr)
    {
        return 0;
    }

    /* Copy the old data. */
    oldsize = GET_SIZE(HDRP(ptr));
    if (newsize < oldsize)
        oldsize = newsize;
    memcpy(newptr, ptr, oldsize);

    /* Free the old block. */
    mm_free(ptr);

    return newptr;
}

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;

    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));

    PUT_PREV(bp, NULL);
    PUT_SUCC(bp, NULL);

    bp = coalesce(bp);
    headinsert(bp);
    return bp;
}

static void place(void *bp, size_t asize)
{
    size_t csize = GET_SIZE(HDRP(bp));
    removeblock(bp);

    if ((csize - asize) >= (2 * DSIZE))
    {
        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));
        PUT_SUCC(bp, NULL);
        PUT_PREV(bp, NULL);
        coalesce(bp);
        headinsert(bp);
    }
    else
    {
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }
}

#ifndef NEXT_FIT
static void *find_fit(size_t asize)
{
    void *bp;

    for (bp = free_listp; bp != NULL; bp = GET_SUCC(bp))
    {
        if (asize <= GET_SIZE(HDRP(bp)))
        {
            return bp;
        }
    }
    return NULL;
}
#endif

#ifdef NEXT_FIT
static void *find_fit(size_t asize)
{
    void *ret = NULL;
    if (rover == NULL)
        rover = free_listp;
    for (; rover != NULL; rover = GET_SUCC(rover))
    {
        if (asize <= GET_SIZE(HDRP(rover)))
        {
            ret = rover;
            rover = GET_SUCC(rover);
            return ret;
        }
    }
    for (rover = free_listp; rover != NULL; rover = GET_SUCC(rover))
    {
        if (asize <= GET_SIZE(HDRP(rover)))
        {
            ret = rover;
            rover = GET_SUCC(rover);
            return ret;
        }
    }
    return ret;
}
#endif

static void removeblock(void *bp)
{
    void *prev = GET_PREV(bp);
    void *succ = GET_SUCC(bp);
    PUT_PREV(bp, NULL);
    PUT_SUCC(bp, NULL);
    if (!prev && !succ)
    {
        free_listp = NULL;
        return;
    }
    if (!prev && succ)
    {
        PUT_PREV(succ, NULL);
        free_listp = succ;
        return;
    }
    if (prev && !succ)
    {
        PUT_SUCC(prev, NULL);
        return;
    }
    if (prev && succ)
    {
        PUT_SUCC(prev, succ);
        PUT_PREV(succ, prev);
    }
}
static void headinsert(void *bp)
{
    if (!free_listp)
    {
        free_listp = bp;
        return;
    }
    void *q = free_listp;
    PUT_SUCC(bp, q);
    PUT_PREV(bp, NULL);
    PUT_PREV(q, bp);
    free_listp = bp;
}
```