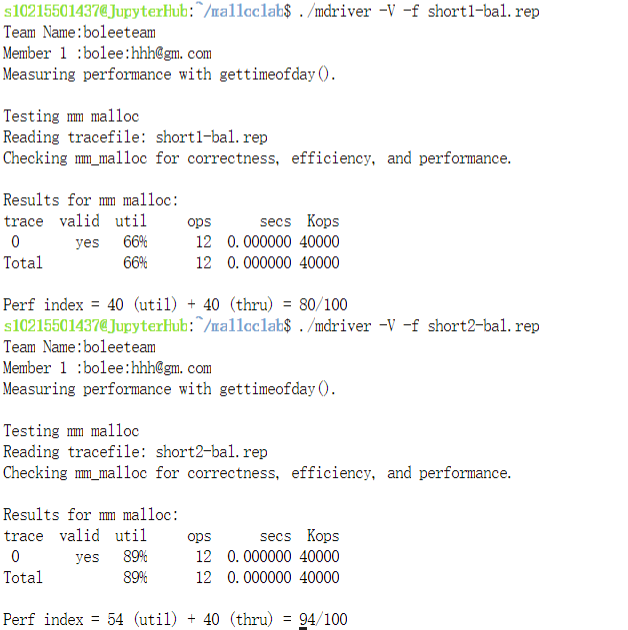
结果：



函数的作用：

int mm\_init(void)：初始化对齐块，序言块，结尾块，并扩充堆块大小。

void \*mm\_malloc(size\_t size)：寻求合适的空闲块并分配，其间调用其他函数完成。

void mm\_free(void \*prt)：修改内存块的分配位并合并内存块。

void \*mm\_realloc(void \*ptr, size\_t size)：按要求分配内存块。

static void \*extend\_heap(size\_t size)：扩充堆空间，并返回指向结尾扩充后的空闲块的指针

static void \*find\_fit(size\_t size)：首次适配（first-fit）

static void place(char \*bp, size\_t size)：负责在空闲块内放置数据，使其分割为分配块和空闲块，注意最小块的限制。

static void \*coalesce(void \*bp)：合并空闲块，四种情况，返回指向该空闲块的指针。

构造函数的思想：

空闲块组织：隐式空闲链表

放置：首次适配（first fit） / 下一次适配（next fit）

分割：分割空闲块的前部使用。

合并：立即合并

#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) \*/

""

};

#define VERBOSE 0

#ifdef DEBUG

#define VERBOSE 1

#endif

/\* single word (4) or double word (8) alignment \*/

#define ALIGNMENT 8

/\* rounds up to the nearest multiple of ALIGNMENT \*/ // 对 ALIGNMENT 倍数上取整的计算

#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)

#define SIZE\_T\_SIZE (ALIGN(sizeof(size\_t)))

/\* 自定义的宏，有便于操作常量和指针运算 \*/

#define WSIZE 4 //字、脚部或头部的大小（字节）

#define DSIZE 8 //双字大小（字节）

#define CHUNKSIZE (1<<12) //扩展堆时的默认大小

#define MINBLOCK (DSIZE + 2\*WSIZE)

#define MAX(x, y) ((x) > (y) ? (x) : (y))

#define PACK(size, alloc) ((size) | (alloc)) //将 size 和 allocated bit 合并为一个字

#define GET(p) (\*(unsigned int \*)(p)) //读地址p处的一个字

#define PUT(p, val) (\*(unsigned int \*)(p) = (val)) //向地址p处写一个字

#define GET\_SIZE(p) (GET(p) & ~0x07) //得到地址p处的 size

#define GET\_ALLOC(p) (GET(p) & 0x1) //得到地址p处的 allocated bit

//block point --> bp指向有效载荷块指针

#define HDRP(bp) ((char\*)(bp) - WSIZE) //获得头部的地址

#define FTRP(bp) ((char\*)(bp) + GET\_SIZE(HDRP(bp)) - DSIZE) //获得脚部的地址, 与宏定义HDRP有耦合

#define NEXT\_BLKP(bp) ((char\*)(bp) + GET\_SIZE((char\*)(bp) - WSIZE)) //计算后块的地址

#define PREV\_BLKP(bp) ((char\*)(bp) - GET\_SIZE((char\*)(bp) - DSIZE)) //计算前块的地址

static void\* heap\_listp; //指向序言块

/\* private functions \*/

static void \*extend\_heap(size\_t size); //拓展堆块

static void \*find\_fit(size\_t size); //寻找空闲块

static void place(char \*bp, size\_t size); //分割空闲块

static void \*coalesce(void \*bp); //合并空闲块

//check

/\*

static void mm\_check(int verbose, const char\* func); //heap consistency checker

static void mm\_checkblock(int verbose, const char\* func, void \*bp);

static int mm\_checkheap(int verbose, const char\* func);

\*/

static void mm\_printblock(int verbose, const char\* func);

/\*

\* mm\_init - initialize the malloc package.

\*/

//设立序言块、结尾块，以及序言块前的对齐块（4B），总共需要4个4B的空间

int mm\_init(void)

{

if ((heap\_listp = mem\_sbrk(4\*WSIZE)) == (void\*)-1)

return -1;

PUT(heap\_listp, 0); //堆起绐位置的对齐块，使bp对齐8字节

PUT(heap\_listp + 1\*WSIZE, PACK(8, 1)); //序言块

PUT(heap\_listp + 2\*WSIZE, PACK(8, 1)); //序言块

PUT(heap\_listp, PACK(0, 1)); //结尾块

heap\_listp += (2\*WSIZE); //小技巧：使heap\_listp指向下一块, 即两个序主块中间

if (extend\_heap(CHUNKSIZE) == NULL) //拓展堆块

return -1;

mm\_printblock(VERBOSE, \_\_func\_\_);

return 0;

}

static void \*extend\_heap(size\_t size) {

size\_t asize;

void \*bp;

asize = ALIGN(size);

//printf("extend %d\n", asize);

if ((long)(bp = mem\_sbrk(asize)) == -1)

return NULL;

PUT(HDRP(bp), PACK(asize, 0)); //HDRP(bp)指向原结尾块

PUT(FTRP(bp), PACK(asize, 0));

PUT(HDRP(NEXT\_BLKP(bp)), PACK(0, 1)); //新结尾块

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; //ajusted size

size\_t extendsize; //若无适配块则拓展堆的大小

void \*bp = NULL;

if (size == 0) //无效的申请

return NULL;

asize = ALIGN(size + 2\*WSIZE);

if ((bp = find\_fit(asize)) != NULL) {

place((char \*)bp, asize);

mm\_printblock(VERBOSE, \_\_func\_\_);

return bp;

}

//无足够空间的空闲块用来分配

extendsize = MAX(asize, CHUNKSIZE);

if ((bp = extend\_heap(extendsize)) == NULL) {

return NULL;

}

place(bp, asize);

mm\_printblock(VERBOSE, \_\_func\_\_);

return bp;

}

//放置策略搜索 首次适配搜索

static void \*find\_fit(size\_t size) {

void \*curbp;

for (curbp = heap\_listp; GET\_SIZE(HDRP(curbp)) > 0; curbp = NEXT\_BLKP(curbp)) {

if (!GET\_ALLOC(HDRP(curbp)) && (GET\_SIZE(HDRP(curbp)) >= size)) return curbp;

}

return NULL; //未适配

}

//分割空闲块

static void place(char \*bp, size\_t asize) { //注意最小块的限制（16B == DSIZE + 2\*WSIZE == MINBLOCK）

size\_t total\_size = GET\_SIZE(HDRP(bp));

size\_t remainder\_size = total\_size - asize;

if (remainder\_size >= MINBLOCK) {

PUT(HDRP(bp), PACK(asize, 1));

PUT(FTRP(bp), PACK(asize, 1));

bp = NEXT\_BLKP(bp);

PUT(HDRP(bp), PACK(remainder\_size, 0));

PUT(FTRP(bp), PACK(remainder\_size, 0));

} else { //没有已分配块或空闲块可以比最小块更小

PUT(HDRP(bp), PACK(total\_size, 1));

PUT(FTRP(bp), PACK(total\_size, 1));

}

}

/\*

\* mm\_free - Freeing a block does nothing.

\*/

void mm\_free(void \*ptr)

{

size\_t size = GET\_SIZE(HDRP(ptr));

PUT(HDRP(ptr), PACK(size, 0));

PUT(FTRP(ptr), PACK(size, 0));

coalesce(ptr);

mm\_printblock(VERBOSE, \_\_func\_\_);

}

/\*

\* coalesce - 合并内存块

\*/

static void \*coalesce(void \*bp) {

int pre\_alloc = GET\_ALLOC(HDRP(PREV\_BLKP(bp)));

int post\_alloc = GET\_ALLOC(HDRP(NEXT\_BLKP(bp)));

size\_t size = GET\_SIZE(HDRP(bp));

if (pre\_alloc && post\_alloc) {

return bp;

} else if (pre\_alloc && !post\_alloc) { //与后块合并

size += GET\_SIZE(HDRP(NEXT\_BLKP(bp)));

} else if (!pre\_alloc && post\_alloc) { //与前块合并

size += GET\_SIZE(HDRP(PREV\_BLKP(bp)));

bp = PREV\_BLKP(bp);

} else { //前后块都合并

size += GET\_SIZE(HDRP(PREV\_BLKP(bp))) + GET\_SIZE(FTRP(NEXT\_BLKP(bp)));

bp = PREV\_BLKP(bp);

}

PUT(HDRP(bp), PACK(size, 0));

PUT(FTRP(bp), PACK(size, 0)); //FTRP()与GET\_SIZE()有耦合，故此时所用的SIZE已经改变

return bp;

}

/\*

\* mm\_realloc - Implemented simply in terms of mm\_malloc and mm\_free

\*/

void \*mm\_realloc(void \*ptr, size\_t size)

{

size\_t old\_size, new\_size, extendsize;

void \*old\_ptr, \*new\_ptr;

if (ptr == NULL) {

return mm\_malloc(size);

}

if (size == 0) {

mm\_free(ptr);

return NULL;

}

new\_size = ALIGN(size + 2\*WSIZE);

old\_size = GET\_SIZE(HDRP(ptr));

old\_ptr = ptr;

if (old\_size >= new\_size) {

if (old\_size - new\_size >= MINBLOCK) { //分割内存块

place(old\_ptr, new\_size);

mm\_printblock(VERBOSE, \_\_func\_\_);

return old\_ptr;

} else { //剩余块小于最小块大小，不分割

mm\_printblock(VERBOSE, \_\_func\_\_);

return old\_ptr;

}

} else { //释放原内存块，寻找新内存块

if ((new\_ptr = find\_fit(new\_size)) == NULL) { //无合适内存块

extendsize = MAX(new\_size, CHUNKSIZE);

if ((new\_ptr = extend\_heap(extendsize)) == NULL) //拓展堆空间

return NULL;

}

place(new\_ptr, new\_size);

memcpy(new\_ptr, old\_ptr, old\_size - 2\*WSIZE);

mm\_free(old\_ptr);

mm\_printblock(VERBOSE, \_\_func\_\_);

return new\_ptr;

}

}