/\*

\* mm-implicit.c - Simple allocator based on implicit free lists,

\* first fit placement, and boundary tag coalescing.

\*

\* Each block has header and footer of the form:

\*

\* 31 3 2 1 0

\* -----------------------------------

\* | s s s s ... s s s 0 0 a/f

\* -----------------------------------

\*

\* where s are the meaningful size bits and a/f is set

\* iff the block is allocated. The list has the following form:

\*

\* begin end

\* heap heap

\* -----------------------------------------------------------------

\* | pad | hdr(8:a) | ftr(8:a) | zero or more usr blks | hdr(8:a) |

\* -----------------------------------------------------------------

\* | prologue | | epilogue |

\* | block | | block |

\*

\* The allocated prologue and epilogue blocks are overhead that

\* eliminate edge conditions during coalescing.

\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <memory.h>

#include "mm.h"

#include "memlib.h"

/\*

Basic Constants and Macros from the book

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* NOTE TO STUDENTS: Before you do anything else, please

\* provide your team information in the following struct.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

team\_t team = {

/\* Team name \*/

"Zach's-Team",

/\* First member's full name \*/

"Zachary North",

/\* First member's email address \*/

"zano7488@colorado.edu",

/\* Second member's full name (leave blank if none) \*/

"",

/\* Second member's email address (leave blank if none) \*/

""

};

/////////////////////////////////////////////////////////////////////////////

// Constants and macros

//

// These correspond to the material in Figure 9.43 of the text

// The macros have been turned into C++ inline functions to

// make debugging code easier.

//

/////////////////////////////////////////////////////////////////////////////

#define WSIZE 4 /\* word size (bytes) \*/

#define DSIZE 8 /\* doubleword size (bytes) \*/

#define CHUNKSIZE (1<<12) /\* initial heap size (bytes) \*/

#define OVERHEAD 8 /\* overhead of header and footer (bytes) \*/

static inline int MAX(int x, int y) {

return x > y ? x : y;

}

//

// Pack a size and allocated bit into a word

// We mask of the "alloc" field to insure only

// the lower bit is used

//

static inline uint32\_t PACK(uint32\_t size, int alloc) {

return ((size) | (alloc & 0x1));

}

//

// Read and write a word at address p

//

static inline uint32\_t GET(void \*p)

{

return \*(uint32\_t \*)p;

}

static inline void PUT( void \*p, uint32\_t val)

{

\*((uint32\_t \*)p) = val;

}

//

// Read the size and allocated fields from address p

//

static inline uint32\_t GET\_SIZE( void \*p )

{

return GET(p) & ~0x7;

}

static inline int GET\_ALLOC( void \*p )

{

return GET(p) & 0x1;

}

//

// Given block ptr bp, compute address of its header and footer

//

static inline void \*HDRP(void \*bp)

{

return ( (char \*)bp) - WSIZE;

}

static inline void \*FTRP(void \*bp)

{

return ((char \*)(bp) + GET\_SIZE(HDRP(bp)) - DSIZE);

}

//

// Given block ptr bp, compute address of next and previous blocks

//

static inline void \*NEXT\_BLKP(void \*bp)

{

return ((char \*)(bp) + GET\_SIZE(((char \*)(bp) - WSIZE)));

}

static inline void\* PREV\_BLKP(void \*bp)

{

return ((char \*)(bp) - GET\_SIZE(((char \*)(bp) - DSIZE)));

}

/\* My Fucntions \*/

#define NEXT\_FREEP(bp) (\*(void \*\*)(bp + DSIZE)));

#define PREV\_FREEP(bp) (\*(void \*\*)(bp));

/////////////////////////////////////////////////////////////////////////////

//

// Global Variables

//

static char \*heap\_listp; /\* pointer to first block \*/

static char \*free\_listp = 0;

//

// function prototypes for internal helper routines

//

static void \*extend\_heap(uint32\_t words);

static void place(void \*bp, uint32\_t asize);

static void \*find\_fit(uint32\_t asize);

static void \*next\_fit(uint32\_t asize);

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

static void printblock(void \*bp);

static void checkblock(void \*bp);

//

// mm\_init - Initialize the memory manager

//

int mm\_init(void)

{

//

// You need to provide this

//

/\* New Stuff \*/

int list\_number;

seg\_listp = mem\_sbrk(SEG\_LIST\_COUNT \* WSIZE);

for(list\_number = 0; list\_number < SEG\_LIST\_COUNTl list\_number++)

{

SEG\_LIST(seg\_listp, list\_number) = NULL;

}

/\*Original \*/

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

{

return -1;

}

PUT(heap\_listp, 0); //Alignement 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));

heap\_listp += (2 \* WSIZE);

if(extend\_heap(CHUNKSIZE / WSIZE) == NULL)

{

return -1;

}

return 0;

}

//

// extend\_heap - Extend heap with free block and return its block pointer

//

static void \*extend\_heap(uint32\_t words)

{

//

// You need to provide this

//

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;

}

/\*Initializes free block header/footer and the epilogue header \*/

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

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

PUT(HDRP(NEXT\_BLKP(bp)), PACK(0, 1));

/\*Coalesce if the previous block was free \*/

return coalesce(bp);

}

//

// Practice problem 9.8

//

// find\_fit - Find a fit for a block with asize bytes

//

static void \*find\_fit(uint32\_t asize)

{

/\*First-Fit search \*/

void \*bp;

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

{

if (!GET\_ALLOC(HDRP(bp)) && (asize <= GET\_SIZE(HDRP(bp))))

{

return bp;

}

}

return NULL; /\* no fit \*/

//#endif

}

static void \*next\_fit(uint32\_t asize)

{

size\_t size = asize;

int list\_number = 0;

void \*list\_ptr = NULL;

while (list\_number < SEG\_LIST\_COUNT)

{

if((list\_number == SEG\_LIST\_COUNT - 1) || ((sise <= 1) && (SEG\_LIST(seg\_listp, list\_number) != NULL)))

{

list\_ptr = SEG\_LIST(seg\_listp, list\_number);

while((list\_ptr != NULL) && (asize > GET\_SIZE(HDRP(list\_ptr))))

{

list\_ptr = GET\_PREV\_BLK(list\_ptr);

}

if(list\_ptr !- NULL)

{

break;

}

}

list\_number++;

size = size >> 1;

}

return list\_ptr;

}

//

// mm\_free - Free a block

//

void mm\_free(void \*bp)

{

//

// You need to provide this

//

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

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

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

coalesce(bp);

}

//

// coalesce - boundary tag coalescing. Return ptr to coalesced block

//

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));

/\*Case 1 \*/

if(prev\_alloc && next\_alloc)

{

return bp;

}

/\*Case 2 \*/

else if(prev\_alloc && !next\_alloc)

{

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

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

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

}

/\*Case 3 \*/

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));

bp = PREV\_BLKP(bp);

}

/\*Case 4 \*/

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));

bp = PREV\_BLKP(bp);

}

return bp;

}

//

// mm\_malloc - Allocate a block with at least size bytes of payload

//

void \*mm\_malloc(uint32\_t size)

{

//

// You need to provide this

//

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;

}

/\*Adjsut 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;

}

//

//

// Practice problem 9.9

//

// place - Place block of asize bytes at start of free block bp

// and split if remainder would be at least minimum block size

//

static void place(void \*bp, uint32\_t asize)

{

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

}

else

{

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

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

}

}

//

// mm\_realloc -- implemented for you

//

void \*mm\_realloc(void \*ptr, uint32\_t size)

{

void \*newp;

uint32\_t copySize;

newp = mm\_malloc(size);

if (newp == NULL) {

printf("ERROR: mm\_malloc failed in mm\_realloc\n");

exit(1);

}

copySize = GET\_SIZE(HDRP(ptr));

if (size < copySize) {

copySize = size;

}

memcpy(newp, ptr, copySize);

mm\_free(ptr);

return newp;

}

/\*Inserting a Free Block \*/

static void insert\_free\_block(void \*bp, size\_t block\_size)

{

void \*list\_ptr = NULL;

void \*insert\_loc = NULL;

int list\_number = 0;

while((list\_number < (SEG\_LIST\_COUNT - 1)) && (block\_size > 1))

{

block\_size = block\_size >> 1; //Reducing size by powers of 2

list\_number++;

}

list\_ptr = SEG\_LIST(seg\_listp, list\_number);

while((list\_ptr != NULL) && (block\_size > GET\_SIZE(HDRP(list\_ptr))))

{

insert\_loc = list\_ptr;

list\_ptr = GET\_PREV\_BLK(list\_ptr);

}

if(list\_ptr)

{

if(insert\_loc)

{

PUT\_PTR(GET\_PREV)

}

}

}

//

// mm\_checkheap - Check the heap for consistency

//

void mm\_checkheap(int verbose)

{

//

// This provided implementation assumes you're using the structure

// of the sample solution in the text. If not, omit this code

// and provide your own mm\_checkheap

//

void \*bp = heap\_listp;

if (verbose) {

printf("Heap (%p):\n", heap\_listp);

}

if ((GET\_SIZE(HDRP(heap\_listp)) != DSIZE) || !GET\_ALLOC(HDRP(heap\_listp))) {

printf("Bad prologue header\n");

}

checkblock(heap\_listp);

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

if (verbose) {

printblock(bp);

}

checkblock(bp);

}

if (verbose) {

printblock(bp);

}

if ((GET\_SIZE(HDRP(bp)) != 0) || !(GET\_ALLOC(HDRP(bp)))) {

printf("Bad epilogue header\n");

}

}

static void printblock(void \*bp)

{

uint32\_t hsize, halloc, fsize, falloc;

hsize = GET\_SIZE(HDRP(bp));

halloc = GET\_ALLOC(HDRP(bp));

fsize = GET\_SIZE(FTRP(bp));

falloc = GET\_ALLOC(FTRP(bp));

if (hsize == 0) {

printf("%p: EOL\n", bp);

return;

}

printf("%p: header: [%d:%c] footer: [%d:%c]\n",

bp,

(int) hsize, (halloc ? 'a' : 'f'),

(int) fsize, (falloc ? 'a' : 'f'));

}

static void checkblock(void \*bp)

{

if ((uintptr\_t)bp % 8) {

printf("Error: %p is not doubleword aligned\n", bp);

}

if (GET(HDRP(bp)) != GET(FTRP(bp))) {

printf("Error: header does not match footer\n");

}

}