/\*

\* mm-naive.c - The fastest, least memory-efficient malloc package.

\*

\* - internal implementation

\*

\* There exist numerous alternative for several options.

\* My 20170736 Hong Jiseung's mm.c uses the format of the followings.

\*

\* 1. Allocation Placement

\*

\* I used first fit. Traversing the free blocks, and then find the

\* first fitted block to allocate.

\*

\* 2. Block Format

\*

\* I used Knuth's boudary tag, which includes header and footer.

\* Not pure.

\*

\* 3. Free List Format

\*

\* I used freeing policy with coalescing right away.

\* It means I do not wait for the next allocation to make coalescing.

\*

\* 4. Splitting Policy

\*

\* As well as Coalescing Policy, I will split the originally freed

\* block to be splitted right away.

\*

\* 5. Re-alloc Implementation

\*

\* Let's say the 'requested size' = new\_size

\* 1) new\_size < old\_size

\* Then another type of coalescing would occur.

\* New header n footer will be made, the size of allocated block will shrunk.

\* 2) new\_size > old\_size

\* Then we free the allocated block, and then do the function 'malloc'.

\* Also another type of coalescing would occur.

\* 3) new\_size = old\_size

\* originally allocated block would be returned.

\*

\* 6. Freed list traversal

\* This code traverse the freed list for operations.

\*

\*/

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <unistd.h>

#include <string.h>

#include "mm.h"

#include "memlib.h"

/\* 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)))

/\* Basic constants and macros \*/

#define WSIZE 4

#define DSIZE 8

#define CHUNKSIZE (1<<12)

/\* Minimum block size \*/

// #define UNIT 24

#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) (\*(size\_t \*)(p))

#define PUT(p, val) (\*(size\_t \*)(p) = val)

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

#define HDRP(bp) ((void\*)(bp) - WSIZE)

#define FTRP(bp) ((void\*)(bp) + GET\_SIZE(HDRP(bp)) - DSIZE)

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

#define NEXT\_BLKP(bp) ((void\*)(bp) + GET\_SIZE(((void\*)(bp) - WSIZE)))

#define PREV\_BLKP(bp) ((void\*)(bp) - GET\_SIZE(((void\*)(bp) - DSIZE)))

/\* Given header or footer ptr p, compute the size of block n allocated state \*/

#define GET\_SIZE(p) (GET(p) & ~0x7)

#define GET\_STATE(p) (GET(p) & 0x1)

/\* Basic pointer to the first block n first freed block \*/

void \*heap\_listp = 0;

/\* Initial prototype of helper functions \*/

void \*extend\_heap(size\_t words);

void place(void \*ptr, size\_t asize);

void \*find\_fit(size\_t asize);

void \*coalesce(void \*ptr);

/\* Extra helper functions \*/

/\*

\* mm\_init - initialize the malloc package.

\*/

int mm\_init(void)

{

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

return -1;

}

/\* Making prologue & epilogue \*/

/\* Alignment paddign n Prologue : [0 | UNIT/1 | UNIT/1] \*/

PUT(heap\_listp, 0);

PUT(heap\_listp + (1 \* WSIZE), PACK(DSIZE, 1));

PUT(heap\_listp + (2 \* WSIZE), PACK(DSIZE, 1));

/\* Epilogue \*/

PUT(heap\_listp + (3 \* WSIZE), PACK(0, 1));

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

/\* Extend the empty heap w/ free block of CHUNKSIZE bytes \*/

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

return -1;

}

return 0;

}

/\*

\* 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)

{

char \*bp;

size\_t asize;

size\_t extend\_size;

if(size == 0){

return NULL;

}

/\* No block can be smaller than UNIT block\*/

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;

}

/\* If no proper fit is found, get more memory \*/

extend\_size = MAX(asize, CHUNKSIZE);

if((bp = extend\_heap(extend\_size/WSIZE)) == NULL)

return NULL;

place(bp, asize);

return bp;

}

/\*

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

\*/

void mm\_free(void \*ptr)

{

if(ptr == NULL){

return;

}

/\* get the block size from ptr \*/

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

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

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

coalesce(ptr);

}

/\*

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

\*/

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

{

size\_t oldsize;

void \*newptr;

if(ptr == NULL){

mm\_malloc(size);

return NULL;

}

if(size == 0){

mm\_free(ptr);

return NULL;

}

// size\_t asize = MAX(ALIGN(size) + DSIZE, DSIZE);

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

// if(oldsize == asize){

// return ptr;

// }

// if(asize <= oldsize){

// size = asize;

// if(oldsize - size <= DSIZE){

// return ptr;

// }

// mm\_free(NEXT\_BLKP(ptr));

// return ptr;

// }

newptr = mm\_malloc(size);

if(newptr == NULL){

return NULL;

}

if(size < oldsize){

oldsize = size;

}

memcpy(newptr, ptr, oldsize);

mm\_free(ptr);

return newptr;

}

/\* reference in book \*/

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));

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

/\* new epilogue \*/

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

return coalesce(bp);

}

/\* reference in book \*/

void \*coalesce(void \*bp){

size\_t prev\_alloc = GET\_STATE(FTRP(PREV\_BLKP(bp)));

size\_t next\_alloc = GET\_STATE(HDRP(NEXT\_BLKP(bp)));

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

if(prev\_alloc && next\_alloc){

return bp;

}

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));

}

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);

}

else{

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

}

/\* first fit \*/

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

void \*bp;

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

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

return bp;

}

}

return NULL;

}

void place(void \*bp, size\_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));

}

}