Національний технічний університет України

«Київський політехнічний інститут»

Факультет інформатики і обчислювальної техніки

**Лабораторна робота №2**

**з курсу: «Системне програмування»**

*Виконав:*

студент групи ІС-72

Шумський В.П.

Залікова книжка №7232

*Перевірив:*

Сімоненко А.В.

Київ, 2020р.

#include <stdio.h>

#include "mem\_alloc/mem\_types.h"

#include "mem\_alloc/mem\_alloc.h"

#include <time.h>

#include <stdint.h>

void test1(void)

{

mem\_dump();

void\* mem\_area1 = mem\_alloc(256);

printf("Block Address:\t%p \n \n",mem\_area1);

mem\_dump();

void\* mem\_area2 = mem\_alloc(3000);

printf("Block Address:\t%p \n \n",mem\_area2);

mem\_dump();

mem\_free(mem\_area2);

mem\_dump();

}

int main(int argc, char \*argv[])

{

test1();

return 0;

}

#include "mem\_alloc.h"

#include <stdio.h>

#include <assert.h>

#include <memory.h>

#include <sys/mman.h>

#include <unistd.h>

#include "mem\_types.h"

#include "mem\_rb\_tree.h"

#include <stdint.h>

//static void\* page\_pointers[PAGE\_NUM];

static char\* mem\_start = NULL;

static size\_t formated\_page\_idxs[BLK\_MAX\_POW2+1] = {[0 ... BLK\_MAX\_POW2] = SIZE\_MAX};

static FreeBigBH\* free\_big\_bhs\_root = NULL;

static intptr\_t header\_ptrs[PAGE\_NUM];

u\_int32\_t roundToPow2(u\_int32\_t v)

{

v--;

v |= v >> 1;

v |= v >> 2;

v |= v >> 4;

v |= v >> 8;

v |= v >> 16;

v++;

return v;

}

u\_int32\_t count\_pow2(u\_int32\_t v)

{

uint r = 0;

while (v >>= 1)

{

r++;

}

return r;

}

size\_t setBit(char\* mask, size\_t mask\_len)

{

int i;

for (i = 0; i < mask\_len; ++i) {

if ((mask[i] & 0xFF) != 0xFF) {

break;

}

}

assert(i != mask\_len);

char m;

size\_t res = i \* 8;

if ((mask[i] & 0xF) != 0xF) {

m = 1;

} else {

m = 0x10;

res += 4;

}

while((mask[i] & m) == m){

m <<= 1;

res++;

}

mask[i] |= m;

return res;

}

void clrBit(char\* mask, size\_t bit\_idx)

{

size\_t i = bit\_idx / 8;

int bit\_num = bit\_idx - i \* 8;

char clr\_mask = 1 << bit\_num;

if (mask[i] & clr\_mask) {

mask[i] &= ~clr\_mask;

return;

}

printf("ERROR: Double free. %zu\n", bit\_idx);

mask[i] &= ~clr\_mask;

}

bool isFree32(char\* usage\_mask, const size\_t bits\_num){

char mask = 0;

int i;

for (i = 1; i < bits\_num / 8; ++i) {

mask |= usage\_mask[i];

}

int bits\_left = bits\_num - (bits\_num / 8) \* 8;

if (bits\_left) {

unsigned char mask2 = 0xFF;

unsigned char mask3 = 0xFF;

mask2 <<= bits\_left;

mask3 >>= 8 - bits\_left;

mask |= (usage\_mask[i] | mask2) & mask3;

}

if (mask) {

return false;

}

mask |= usage\_mask[0];

if (mask == 1) {

return true;

}

return false;

}

bool isFree(char\* usage\_mask, const size\_t bits\_num)

{

char mask = 0;

int i;

for (i = 0; i < bits\_num / 8; ++i) {

mask |= usage\_mask[i];

}

int bits\_left = bits\_num - (bits\_num / 8) \* 8;

if (bits\_left) {

char mask2 = 0xFF;

unsigned char mask3 = 0xFF;

mask2 <<= bits\_left;

mask3 >>= 8 - bits\_left;

mask |= (usage\_mask[i] | mask2) & mask3;

}

if (mask) {

return false;

}

return true;

}

bool isFull(char\* usage\_mask, const size\_t bits\_num)

{

char mask = -1;

int i;

for (i = 0; i < bits\_num / 8; ++i) {

mask &= usage\_mask[i];

}

int bits\_left = bits\_num - (bits\_num / 8) \* 8;

if (bits\_left) {

signed char mask2 = 0xFF;

char mask3 = 0xFF;

mask2 >>= 8 - bits\_left;

mask3 <<= bits\_left;

mask &= (usage\_mask[i] & mask2) | mask3;

}

if (mask == -1) {

return true;

}

return false;

}

size\_t roundToPage(size\_t size)

{

return (size + PAGE\_SIZE - 1) / PAGE\_SIZE \* PAGE\_SIZE;

}

void\* get\_page\_start(void\* addr)

{

size\_t bytes\_num = (char\*) addr - mem\_start;

return mem\_start + bytes\_num / PAGE\_SIZE \* PAGE\_SIZE;

}

void\* mem\_init(void)

{

assert(PAGE\_SIZE == sysconf(\_SC\_PAGE\_SIZE));

/\* create mapping \*/

void\* mapping\_area = mmap(NULL, PAGE\_NUM\*PAGE\_SIZE, PROT\_READ|PROT\_WRITE,

MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0);

if (mapping\_area == MAP\_FAILED) {

return NULL;

}

mem\_start = mapping\_area;

FreeBigBH\* bh = mapping\_area;

bh->info.prev\_size = 0;

bh->info.size = PAGE\_NUM\*PAGE\_SIZE;

header\_ptrs[0] = (intptr\_t) bh | FREE\_BIG\_BH;

set\_rb\_root\_var(&free\_big\_bhs\_root);

rbtree\_insert(bh);

return mapping\_area;

}

size\_t page\_idx(void\* page\_addr)

{

return ((char\*) page\_addr - mem\_start) / PAGE\_SIZE;

}

void set\_nxt\_blk\_prev\_size(size\_t size, void\* next\_blk)

{

size\_t pg\_idx = page\_idx(next\_blk);

HeaderType h\_type = header\_ptrs[pg\_idx] & 3;

intptr\_t ptr\_mask = 3;

switch (h\_type) {

case USED\_PH:;

UsedPH\* ph = (UsedPH\*) (header\_ptrs[pg\_idx] & ~ptr\_mask);

ph->prev\_size = size;

break;

case USED\_BIG\_BH:

case FREE\_BIG\_BH:;

FreeBigBH\* fbh = (FreeBigBH\*) (header\_ptrs[pg\_idx] & ~ptr\_mask);

fbh->info.prev\_size = size;

break;

default:

break;

}

}

void\* mem\_alloc(size\_t size)

{

if (mem\_start == NULL) {

if (mem\_init() == NULL) {

return NULL;

}

}

if (size == 0) {

return NULL;

}

// uint pow2 = nextPow2(size);

size\_t block\_len = roundToPow2(size);

uint pow2 = count\_pow2(block\_len);

if (pow2 < BLK\_MIN\_POW2) {

pow2 = BLK\_MIN\_POW2;

block\_len = 1 << BLK\_MIN\_POW2;

}

// size\_t block\_len = 1;

// block\_len <<= pow2;

if (size <= PAGE\_SIZE / 2) {

size\_t ph\_idx = formated\_page\_idxs[pow2];

size\_t blocks\_num = PAGE\_SIZE / block\_len;

size\_t mask\_len = (blocks\_num + 7) / 8;

if (ph\_idx != SIZE\_MAX) {

UsedPH\* ph = (UsedPH\*) (header\_ptrs[ph\_idx] & ~(intptr\_t)3);

size\_t block\_idx = setBit(ph->usage\_mask, mask\_len);

if (isFull(ph->usage\_mask, blocks\_num)){

formated\_page\_idxs[pow2] = ph->next\_ph\_idx;

}

void\* result = mem\_start + ph\_idx \* PAGE\_SIZE + block\_len \* block\_idx;

return result;

}

if (free\_big\_bhs\_root == NULL) {

return NULL; // no memory left

}

/\* Format new page \*/

UsedPH\* new\_ph;

void\* first\_block;

if (pow2 != 5) {

new\_ph = mem\_alloc(sizeof(UsedPH) + mask\_len);

if (new\_ph == NULL) {

return NULL; // no memory left for the header

}

new\_ph->usage\_mask[0] = 1;

first\_block = free\_big\_bhs\_root;

}

FreeBigBH\* fbh = free\_big\_bhs\_root;

rbtree\_delete(fbh);

if (fbh->info.size > PAGE\_SIZE) {

FreeBigBH\* rem\_fbh = (FreeBigBH\*)((char\*) fbh + PAGE\_SIZE);

rem\_fbh->info.prev\_size = PAGE\_SIZE;//fbh->info.prev\_size;

rem\_fbh->info.size = fbh->info.size - PAGE\_SIZE;

void\* next\_block = (char\*) rem\_fbh + rem\_fbh->info.size;

if ((char\*) next\_block < mem\_start + PAGE\_NUM\*PAGE\_SIZE) {

set\_nxt\_blk\_prev\_size(rem\_fbh->info.size, next\_block);

}

rbtree\_insert(rem\_fbh);

size\_t block\_idx = page\_idx(rem\_fbh);

header\_ptrs[block\_idx] = (intptr\_t) rem\_fbh | FREE\_BIG\_BH;

}

size\_t prev\_sz = fbh->info.prev\_size;

if (pow2 == 5) {

new\_ph = (UsedPH\*)fbh;

new\_ph->usage\_mask[0] = 3;

first\_block = (char\*) fbh + block\_len;

}

size\_t pg\_idx = page\_idx(fbh);

header\_ptrs[pg\_idx] = (intptr\_t) new\_ph | USED\_PH;

new\_ph->blk\_size\_pow2 = pow2;

new\_ph->prev\_size = prev\_sz;

if (prev\_sz == 0) {

printf("Memory init\n");

}

new\_ph->next\_ph\_idx = formated\_page\_idxs[pow2];

memset(&new\_ph->usage\_mask[1], 0, mask\_len-1);

formated\_page\_idxs[pow2] = pg\_idx;

return first\_block;

}

/\* if big block \*/

if (free\_big\_bhs\_root == NULL) {

return NULL; // no memory left

}

FreeBigBH\* fbh = rbtree\_lookup(size + sizeof(UsedBigBH));

if (fbh == NULL) {

return NULL; // no suitable free block left

}

rbtree\_delete(fbh);

size\_t block\_size = roundToPage(size + sizeof(UsedBigBH));

if (block\_size < fbh->info.size) {

FreeBigBH\* rem\_bh = (FreeBigBH\*)((char\*) fbh + block\_size);

rem\_bh->info.prev\_size = block\_size;

rem\_bh->info.size = fbh->info.size - block\_size;

void\* next\_block = (char\*) rem\_bh + rem\_bh->info.size;

if ((char\*) next\_block < mem\_start + PAGE\_NUM\*PAGE\_SIZE) {

set\_nxt\_blk\_prev\_size(rem\_bh->info.size, next\_block);

}

rbtree\_insert(rem\_bh);

size\_t block\_idx = page\_idx(rem\_bh);

header\_ptrs[block\_idx] = (intptr\_t) rem\_bh | FREE\_BIG\_BH;

fbh->info.size = block\_size;

}

header\_ptrs[page\_idx(fbh)] = (intptr\_t) fbh | USED\_BIG\_BH;

return (UsedBigBH\*) fbh + 1;

}

void\* mem\_realloc(void \*addr, size\_t size)

{

if (addr == NULL) {

return mem\_alloc(size);

}

if (size == 0) {

return NULL;

}

void\* pg\_start = get\_page\_start(addr);

size\_t pg\_idx = page\_idx(pg\_start);

HeaderType h\_type = header\_ptrs[pg\_idx] & 3;

void\* new\_addr;

switch (h\_type) {

case USED\_BIG\_BH:;

UsedBigBH\* bh = pg\_start;

if (bh->size >= size + sizeof(UsedBigBH) + PAGE\_SIZE) {

size\_t free\_bh\_sz = (bh->size - size - sizeof(UsedBigBH)) / PAGE\_SIZE \* PAGE\_SIZE; // rounding

bh->size -= free\_bh\_sz;

FreeBigBH\* free\_bh = (FreeBigBH\*) ((char\*) bh + bh->size);

free\_bh->info.prev\_size = bh->size;

free\_bh->info.size = free\_bh\_sz;

void\* next\_addr = (char\*) free\_bh + free\_bh\_sz;

bool hasNext = (char\*) next\_addr < mem\_start + PAGE\_NUM\*PAGE\_SIZE;

if (hasNext) {

size\_t next\_idx = page\_idx(next\_addr);

HeaderType next\_h\_type = header\_ptrs[next\_idx] & 3;

switch (next\_h\_type) {

case FREE\_BIG\_BH:

rbtree\_delete(next\_addr);

free\_bh->info.size += ((FreeBigBH\*) next\_addr)->info.size;

char\* next\_next\_blk = (char\*) next\_addr + ((FreeBigBH\*) next\_addr)->info.size;

if (next\_next\_blk < mem\_start + PAGE\_NUM\*PAGE\_SIZE) {

set\_nxt\_blk\_prev\_size(free\_bh->info.size, next\_next\_blk);

}

break;

case USED\_BIG\_BH:

case USED\_PH:

set\_nxt\_blk\_prev\_size(free\_bh->info.size, next\_addr);

break;

default:

break;

}

}

rbtree\_insert(free\_bh);

size\_t free\_bh\_idx = page\_idx(free\_bh);

header\_ptrs[free\_bh\_idx] = (intptr\_t) free\_bh | FREE\_BIG\_BH;

return addr; // TODO add call to free\_block mb.

}

if (bh->size >= size + sizeof(UsedBigBH)) {

return addr;

}

void\* next\_addr = (char\*) bh + bh->size;

if ((char\*) next\_addr < mem\_start + PAGE\_NUM \* PAGE\_SIZE) {

size\_t next\_idx = page\_idx(next\_addr);

HeaderType next\_h\_type = header\_ptrs[next\_idx] & 3;

if (next\_h\_type == FREE\_BIG\_BH) {

FreeBigBH\* next\_bh = next\_addr;

if (next\_bh->info.size + bh->size >= size + sizeof(UsedBigBH) + PAGE\_SIZE) { // was <=

rbtree\_delete(next\_bh);

size\_t block\_size = roundToPage(size + sizeof(UsedBigBH));

FreeBigBH\* rem\_bh = (FreeBigBH\*)((char\*) bh + block\_size);

rem\_bh->info.prev\_size = block\_size;

rem\_bh->info.size = next\_bh->info.size + bh->size - block\_size;

void\* next\_block = (char\*) rem\_bh + rem\_bh->info.size;

if ((char\*) next\_block < mem\_start + PAGE\_NUM\*PAGE\_SIZE) {

set\_nxt\_blk\_prev\_size(rem\_bh->info.size, next\_block);

}

rbtree\_insert(rem\_bh);

size\_t block\_idx = page\_idx(rem\_bh);

header\_ptrs[block\_idx] = (intptr\_t) rem\_bh | FREE\_BIG\_BH;

bh->size = block\_size;

return addr;

}

if (next\_bh->info.size + bh->size >= size + sizeof(UsedBigBH)) {

rbtree\_delete(next\_bh);

bh->size += next\_bh->info.size;

void\* next\_next\_addr = (char\*) next\_bh + next\_bh->info.size;

if ((char\*) next\_next\_addr < mem\_start + PAGE\_NUM \* PAGE\_SIZE) {

set\_nxt\_blk\_prev\_size(bh->size, next\_next\_addr);

}

return addr;

}

}

}

new\_addr = mem\_alloc(size);

if (new\_addr == NULL) {

return NULL;

}

memcpy(new\_addr, addr, bh->size - sizeof(UsedBigBH));

mem\_free(addr);

return new\_addr;

case USED\_PH:;

UsedPH\* old\_ph = (UsedPH\*) (header\_ptrs[pg\_idx] & ~(intptr\_t)3);

size\_t old\_ph\_blk\_sz = 1 << old\_ph->blk\_size\_pow2;

if (old\_ph\_blk\_sz >= size) {

return addr;

}

new\_addr = mem\_alloc(size);

if (new\_addr == NULL) {

return NULL;

}

memcpy(new\_addr, addr, old\_ph\_blk\_sz);

mem\_free(addr);

return new\_addr;

default:

printf("ERROR: in mem\_realloc()\n");

fflush(stdout);

return NULL;

}

static void free\_block(void\* addr, size\_t block\_size, size\_t prev\_size)

{

size\_t blk\_idx = page\_idx(addr);

void\* prev\_addr = (char\*) addr - prev\_size;

size\_t prev\_idx;

HeaderType prev\_h\_type;

bool hasPrev = prev\_size != 0;

if (hasPrev) {

prev\_idx = page\_idx(prev\_addr);

prev\_h\_type = header\_ptrs[prev\_idx] & 3;

}

void\* next\_addr = (char\*) addr + block\_size;

bool hasNext = (char\*) next\_addr < mem\_start + PAGE\_NUM\*PAGE\_SIZE;

HeaderType next\_h\_type;

if (hasNext) {

size\_t next\_idx = page\_idx(next\_addr);

next\_h\_type = header\_ptrs[next\_idx] & 3;

}

size\_t new\_size = block\_size;

FreeBigBH\* new\_blk = addr;

if (hasPrev) {

switch (prev\_h\_type) {

case FREE\_BIG\_BH:

rbtree\_delete(prev\_addr);

new\_size += prev\_size;

new\_blk = prev\_addr;

header\_ptrs[prev\_idx] = (intptr\_t) new\_blk | FREE\_BIG\_BH;

break;

case USED\_BIG\_BH:

case USED\_PH:

new\_blk->info.prev\_size = prev\_size;

header\_ptrs[blk\_idx] = (intptr\_t) new\_blk | FREE\_BIG\_BH;

break;

default:

break;

}

} else {

header\_ptrs[blk\_idx] = (intptr\_t) new\_blk | FREE\_BIG\_BH;

}

new\_blk->info.size = new\_size;

if (hasNext) {

switch (next\_h\_type) {

case FREE\_BIG\_BH:

rbtree\_delete(next\_addr);

new\_size += ((FreeBigBH\*) next\_addr)->info.size;

char\* next\_next\_blk = (char\*) next\_addr + ((FreeBigBH\*) next\_addr)->info.size;

new\_blk->info.size = new\_size;

if (next\_next\_blk >= mem\_start + PAGE\_NUM\*PAGE\_SIZE) {

break;

}

set\_nxt\_blk\_prev\_size(new\_size, next\_next\_blk);

break;

case USED\_BIG\_BH:

case USED\_PH:

set\_nxt\_blk\_prev\_size(new\_size, next\_addr);

break;

default:

break;

}

}

rbtree\_insert(new\_blk);

}

size\_t formatted\_prev\_idx(int pow2, size\_t pg\_idx)

{

size\_t prev\_idx = SIZE\_MAX;

for (size\_t i = formated\_page\_idxs[pow2]; i != SIZE\_MAX;

i = ((UsedPH\*) (header\_ptrs[i] & ~(intptr\_t)3))->next\_ph\_idx) {

if (i >= PAGE\_NUM) {

fflush(stdout);

printf("Prev idx!\n");

}

if (pg\_idx == i) {

return prev\_idx;

}

prev\_idx = i;

}

assert(prev\_idx == SIZE\_MAX);

printf("ERROR? at formatted\_prev\_idx, line: %i\n", \_\_LINE\_\_);

mem\_dump();

fflush(stdout);

return SIZE\_MAX-1;

}

void mem\_free(void \*addr)

{

if (addr == NULL) {

return;

}

intptr\_t test = (intptr\_t) addr & 3;

if (test != 0) {

fprintf(stderr, "ERROR: Incorrect address for mem\_free()!\n");

}

void\* page\_start = get\_page\_start(addr);

size\_t pg\_idx = page\_idx(page\_start);

HeaderType h\_type = header\_ptrs[pg\_idx] & 3;

switch (h\_type) {

case USED\_PH:;

UsedPH\* ph = (UsedPH\*) (header\_ptrs[pg\_idx] & ~(intptr\_t)3);

size\_t blk\_size = 1;

blk\_size <<= ph->blk\_size\_pow2;

int block\_idx = ((char\*) addr - (char\*) page\_start) / blk\_size;

size\_t blocks\_num = PAGE\_SIZE / blk\_size;

bool was\_full = isFull(ph->usage\_mask, blocks\_num);

clrBit(ph->usage\_mask, block\_idx);

if ((blk\_size == 32 && isFree32(ph->usage\_mask, blocks\_num))

|| (blk\_size != 32 && isFree(ph->usage\_mask, blocks\_num))) {

/\* remove from formatted pages list \*/

size\_t f\_prev\_idx = formatted\_prev\_idx(ph->blk\_size\_pow2, pg\_idx);

if (f\_prev\_idx == SIZE\_MAX - 1) {

formated\_page\_idxs[ph->blk\_size\_pow2] = SIZE\_MAX;

} else if (f\_prev\_idx != SIZE\_MAX) {

UsedPH\* list\_prev\_ph = (UsedPH\*) (header\_ptrs[f\_prev\_idx] & ~(intptr\_t)3);

list\_prev\_ph->next\_ph\_idx = ph->next\_ph\_idx;

} else { // if it's the head of the list

formated\_page\_idxs[ph->blk\_size\_pow2] = ph->next\_ph\_idx;

}

free\_block(page\_start, PAGE\_SIZE, ph->prev\_size);

if (blk\_size != 32) {

mem\_free(ph);

}

} else if (was\_full) {

ph->next\_ph\_idx = formated\_page\_idxs[ph->blk\_size\_pow2];

formated\_page\_idxs[ph->blk\_size\_pow2] = pg\_idx;

}

break;

case USED\_BIG\_BH:;

UsedBigBH\* bh = (UsedBigBH\*) (header\_ptrs[pg\_idx] & ~(intptr\_t)3);

free\_block(bh, bh->size, bh->prev\_size);

break;

case FREE\_BIG\_BH:

printf("ERROR: double free!\n");

fflush(stdout);

break;

default:

break;

}

}

void mem\_dump(void)

{

size\_t i = 0;

char\* addr = mem\_start;

if (mem\_start == NULL) {

printf("Memory is not initialized\n");

return;

}

bool prev\_was\_free = false;

while (true) {

HeaderType h\_type = header\_ptrs[i] & 3;

switch (h\_type) {

case USED\_PH:;

UsedPH\* ph = (UsedPH\*) (header\_ptrs[i] & ~(intptr\_t)3);

char\* used\_ph\_addr = mem\_start+i\*PAGE\_SIZE;

size\_t block\_size = 1 << ph->blk\_size\_pow2;

printf("UsedPH:\t%p block\_size: %zu : ", used\_ph\_addr, block\_size);

printf("\n");

size\_t ph\_size = PAGE\_SIZE;

addr += ph\_size;

prev\_was\_free = false;

break;

case USED\_BIG\_BH:;

UsedBigBH\* big\_bh = (UsedBigBH\*) (header\_ptrs[i] & ~(intptr\_t)3);

printf("UsedBigBH:\t%p, size: %zu\n", big\_bh, big\_bh->size);

addr += big\_bh->size;

assert(big\_bh->size != 0);

prev\_was\_free = false;

break;

case FREE\_BIG\_BH:;

if (prev\_was\_free) {

printf("prev was free!\n");

}

prev\_was\_free = true;

FreeBigBH\* free\_bh = (FreeBigBH\*) (header\_ptrs[i] & ~(intptr\_t)3);

printf("FreeBigBH:\t%p, size: %zu\n", free\_bh, free\_bh->info.size);

addr += free\_bh->info.size;

assert(free\_bh->info.size != 0);

break;

default:

break;

}

if (addr >= mem\_start + PAGE\_NUM \* PAGE\_SIZE) {

return;

}

i = page\_idx(addr);

fflush(stdout);

}

}

HeaderType mem\_get\_type(void\* addr)

{

void\* page\_start = get\_page\_start(addr);

size\_t pg\_idx = page\_idx(page\_start);

HeaderType h\_type = header\_ptrs[pg\_idx] & 3;

return h\_type;

}