Національний технічний університет України

«Київський політехнічний інститут»

Факультет інформатики і обчислювальної техніки

Кафедра автоматизованих систем обробки інформації та управління

**Лабораторна робота №2**

**з курсу  «Сучасні операційні сиситеми»**

**«Алокатор пам’яті загального призначення»**

                                                                                           Виконала:

студентка групи  ІС-72,

Дорошенко Карина

Київ 2020

**Завдання**

Разработать аллокатор общего назначения, используя за основу описанный выше базовый вариант алгоритма, принимая во внимания следующие условия:

1. Области памяти можно выделять любым доступным способом.
2. Функции mem\_alloc(), mem\_realloc() и mem\_free() должны соответствовать приведенным выше прототипам.
3. Адреса памяти, возвращаемые функциями mem\_alloc() и mem\_realloc(), должны быть выровнены на границу в 4 байта.
4. Попытаться уменьшить время поиска свободного блока памяти и время освобождения занятого блока.
5. Попытаться уменьшить фрагментацию памяти.

Написать функцию mem\_dump(), которая должна выводить на консоль состояние областей памяти.

**Виконання завдання**

**Лістинг програми:**

**main.cpp**

#include <Windows.h>;

#include <iostream>;

#include "Allocator.h";

using namespace std;

int main() {

size\_t size = 1024 \* 4096;

size\_t ps = 4096;

int n = 300;

Allocator al(size, ps);

size\_t\*\* addrArray = new size\_t \* [n];

cout << "-------- test started --------" << endl;

for (int i = 0; i < n; i++) {

addrArray[i] = (size\_t\*)al.mem\_alloc(rand());

if (addrArray[i] == NULL) {

cout << "error" << endl;

}

}

al.mem\_dump();

for (int i = 0; i < n / 3; i++) {

al.mem\_realloc(addrArray[i], rand());

}

al.mem\_dump();

for (int i = n / 2; i < n; i++) {

al.mem\_free(addrArray[i]);

}

al.mem\_dump();

cout << "--------test\_finished----------" << endl;

getchar();

return 0;

}

**Allocator.cpp**

#include <Windows.h>;

#include <iostream>;

#include "Allocator.h";

using namespace std;

Allocator::Allocator(const size\_t n, const size\_t ps) {

pages = n / ps;

size\_t pds = sizeof(PageDescriptor) / sizeof(size\_t);

cout << pds << endl;

size\_t spaseForPD = pds \* pages;

size\_t maxbs = ps >> 1;

size\_t minbs = 16;

lBlocksLength = 1;

while (maxbs != minbs) {

maxbs >>= 1;

lBlocksLength++;

}

size\_t needControlSpase = lBlocksLength + spaseForPD;

size\_t\* mas = new size\_t[n + needControlSpase];

size = n;

begin = mas;

pagesBegin = mas + needControlSpase;

pageSize = ps;

pageDescriptors = (PageDescriptor\*)begin;

lBlocks = begin + spaseForPD;

for (int i = 0; i < pages; i++) {

pageDescriptors[i].state = 0;

}

for (int i = 0; i < lBlocksLength; i++) {

lBlocks[i] = pages + 1;

}

initPages();

cout << begin << endl;

cout << pagesBegin << endl;

cout << size << endl;

cout << pageSize << endl;

cout << pages << endl;

cout << lBlocksLength << endl;

cout << pageDescriptors << endl;

cout << lBlocks << endl;

}

void\* Allocator::mem\_alloc(size\_t s) {

PageDescriptor def = defineCategory(s);

size\_t\* res;

if (def.state == 1) {

res = getFreeLBlock(def.bsize);

}

else {

res = getFreeMBlock(def.bsize);

}

return res;

}

PageDescriptor Allocator::defineCategory(size\_t s) {

PageDescriptor pd;

if (s > (pageSize >> 1)) {

pd.state = 2;

pd.bsize = s / pageSize;

pd.bsize += ((s % pageSize) > 0) ? 1 : 0;

}

else {

pd.state = 1;

pd.bsize = defineBlockSize(s);

}

return pd;

}

size\_t Allocator::defineBlockSize(size\_t s) {

size\_t res = 16;

while (s > res) {

res <<= 1;

}

return res;

}

size\_t\* Allocator::getFreeMBlock(size\_t ps) {

PageDescriptor pd = pageDescriptors[firstFreePage];

size\_t firstPage = firstFreePage;

size\_t thisPage = firstFreePage;

size\_t nextPage;

size\_t counter = 0;

while (pd.next <= pages && counter < ps) {

nextPage = pd.next;

if (nextPage - 1 == thisPage) {

counter++;

}

else {

counter = 0;

firstPage = nextPage;

}

thisPage = nextPage;

pd = pageDescriptors[thisPage];

nextPage = pd.next;

}

if (counter == ps) {

firstFreePage = nextPage;

return getAbsolutePageAddr(firstFreePage);

}

else {

return NULL;

}

}

bool Allocator::freeLBlockIsLast(PageDescriptor pd) {

LBlockDescriptor\* desc = pd.firstFree;

if (desc -> nextFreeBlock) {

return false;

}

else {

return true;

}

}

size\_t\* Allocator::getFreeLBlock(size\_t bs) {

size\_t numberOfPage = lBlocks[getIndex(bs)];

size\_t index;

if (numberOfPage > pages) {

index = createLBlockPage(bs);

}

else {

index = numberOfPage;

}

PageDescriptor pd = pageDescriptors[index];

LBlockDescriptor\* numberOfBlock = pd.firstFree;

if (numberOfBlock -> nextFreeBlock == 0) {

pd.firstFree = NULL;

lBlocks[index] = pages + 1;

}

else {

pd.firstFree = (LBlockDescriptor\*)((size\_t\*)(numberOfBlock)+numberOfBlock -> nextFreeBlock);

}

return (size\_t\*)pd.firstFree;

}

size\_t Allocator::createLBlockPage(size\_t bs) {

size\_t index = getFreePage();

if (index <= pages) {

PageDescriptor pd = pageDescriptors[index];

pd.state = 1;

pd.bsize = bs;

size\_t\* addr = getAbsolutePageAddr(index);

for (int i = 0; i < pages / bs; i++) {

LBlockDescriptor\* des = (LBlockDescriptor\*)addr[i \* pd.bsize];

des -> nextFreeBlock = 1;

if (i == pages / bs - 1)

des -> nextFreeBlock = 0;

}

pd.firstFree = 0;

pd.next = pages + 1;

lBlocks[getIndex(bs)] = index;

}

return index;

}

size\_t\* Allocator::getAbsolutePageAddr(size\_t index) {

size\_t\* res = NULL;

if (index <= pages) {

res = &(pagesBegin[pageSize \* index]);

}

return res;

}

size\_t Allocator::getFreePage() {

PageDescriptor pd = pageDescriptors[firstFreePage];

firstFreePage = pd.next;

return pages + 1;

}

void Allocator::setAllFree(PageDescriptor pd) {

size\_t bs = pd.bsize;

size\_t blocks = pageSize / bs;

for (int i = 0; i < blocks; i++) {

6

}

}

void Allocator::initPages() {

for (int i = 0; i < pages; i++) {

pageDescriptors[i].state = 0;

pageDescriptors[i].next = i + 1;

}

}

size\_t Allocator::getIndex(size\_t s) {

size\_t counter = 0;

while (s > 1) {

counter++;

s >>= s;

}

counter -= 4;

return counter;

}

void\* Allocator::mem\_realloc(void\* addr, size\_t size) {

if (addr == NULL) {

return mem\_alloc(size);

}

size\_t\* beg = (size\_t\*)addr;

size\_t pageNumber = findPageByAddress(beg);

PageDescriptor pd = pageDescriptors[pageNumber];

size\_t usefulMem;

if (pd.state == 1) {

usefulMem = pd.bsize;

}

else {

usefulMem = pd.bsize \* pageSize;

}

if (size == usefulMem) {

return addr;

}

mem\_free(addr);

size\_t\* newAddr = (size\_t\*)mem\_alloc(size);

size\_t length = min(size, usefulMem);

copyData(beg, newAddr, length);

return newAddr;

}

void Allocator::copyData(size\_t\* from, size\_t\* to, size\_t length) {

if (from > to) {

for (int i = 0; i < length; i++) {

to[i] = from[i];

}

}

else {

for (int i = length - 1; i <= 0; i++) {

to[i] = from[i];

}

}

}

void Allocator::mem\_free(void\* addr) {

size\_t pageNumber = findPageByAddress((size\_t\*)addr);

PageDescriptor pd = pageDescriptors[pageNumber];

if (pd.state == 1) {

size\_t block = findBlockByAddress((size\_t\*)addr, pd.bsize);

if (pd.firstFree == NULL) {

pd.firstFree = (LBlockDescriptor\*)(addr);

7

pd.firstFree -> nextFreeBlock = 0;

}

else {

size\_t shift = (size\_t\*)addr - (size\_t\*)(pd.firstFree);

pd.firstFree = (LBlockDescriptor\*)(addr);

pd.firstFree -> nextFreeBlock = shift;

}

}

else {

size\_t pgs = pd.bsize;

for (int i = 0; i < pgs; i++) {

pd = pageDescriptors[pageNumber + i];

pd.state = 0;

pd.bsize = 0;

pd.firstFree = NULL;

pd.next = firstFreePage;

firstFreePage = pageNumber + i;

}

}

}

size\_t Allocator::findPageByAddress(size\_t\* addr) {

size\_t shiftFromBegin = addr - pagesBegin;

size\_t pageNumber = shiftFromBegin / pageSize;

return pageNumber;

}

size\_t Allocator::findBlockByAddress(size\_t\* addr, size\_t bs) {

size\_t shiftFromBegin = addr - pagesBegin;

size\_t shiftFromPageBegin = shiftFromBegin % pageSize;

size\_t number = shiftFromPageBegin / bs;

return number;

}

void Allocator::mem\_dump() {

cout << "begin: " << endl;

cout << "begin of pages :" << endl;

cout << "number of pages :" << endl;

cout << "page size :" << endl;

cout << "page descriptors state :" << endl;

PageDescriptor pd;

cout << "(state, size, next, firstFree)" << endl;

for (int i = 0; i < pages; i++) {

pd = pageDescriptors[i];

cout << "[" << i << "]" << pd.state << " " pd.bsize << " " << pd.next << " " pd.firstFree << endl;

}

}

**Allocator.h**

#pragma once

struct LBlockDescriptor {

size\_t nextFreeBlock;//shift

};

struct PageDescriptor {

size\_t next; //list of pages with the same state

LBlockDescriptor\* firstFree; //free block in state 1

size\_t bsize; // size of lBlock in state 1 or number of blocks in state 2

char state; //0-free, 1-lblock, 2-mblock

};

class Allocator {

public:

Allocator(const size\_t ms, const size\_t ps);

//return addr on begin of allocated block or NULL

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

//return addr on begin of reallocated block or NULL

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

//free block by this address

void mem\_free(void\* addr);

//out blocks characteristic in table on console

void mem\_dump();

private:

//begin of control information

size\_t\* begin;

//begin of pages in memory

size\_t\* pagesBegin;

size\_t size;

size\_t pages;

size\_t pageSize;

//array of all page descriptors

PageDescriptor\* pageDescriptors;

//array of pages with state 1

size\_t\* lBlocks;

size\_t lBlocksLength;

size\_t firstFreePage;

//define what type of state rigth for this size

PageDescriptor defineCategory(size\_t s);

//round s to minimal need size power of 2

size\_t defineBlockSize(size\_t s);

//initial all pages as free

void initPages();

// return free block with size bs

size\_t\* getFreeLBlock(size\_t bs);

//return index for lBlocks array

size\_t getIndex(size\_t s);

size\_t createLBlockPage(size\_t bs);

//size\_t getLBlockPage(size\_t bs);

//not use :/

size\_t getFreePage();

//return page`s address from index of pages array

size\_t\* getAbsolutePageAddr(size\_t index);

//set all blocks in page to free state

void setAllFree(PageDescriptor pd);

//return big block with length ps\*pageSize

size\_t\* getFreeMBlock(size\_t ps);

//check is this block alst free in this page

bool freeLBlockIsLast(PageDescriptor pd);

//return number of page from her addr

size\_t findPageByAddress(size\_t\* addr);

//return number of block from his addr

size\_t findBlockByAddress(size\_t\* addr, size\_t bs);

//copy data from old pos to new

void copyData(size\_t\* from, size\_t\* to, size\_t length);

};