Национальный технический университет Украины

«Киевский политехнический институт»

Факультет информатики и вычислительной техники

Кафедра вычислительной техники

**Лабораторная работа №1**

*по курсу «Организация вычислительных процессов»*

Аллокатор памяти общего назначения

Выполнил

студент группы ИВ-34

Кривоносов Алексей

Киев, 2016 г.

Аллокатор памяти общего назначения должен выполнять, по крайней мере, три задачи: выделение блока памяти заданного размера, освобождение выделенного блока памяти и изменение размера выделенного блока памяти. Эти функции составляют интерфейс аллокатора памяти общего назначения:

1. **public** **int** mem\_alloc(**int** size)

Метод проверяет все свободные блоки памяти . Если метод находит свободный блок необходимого размера, то этот блок помечается как занятый и метод возвращает его адрес. Если метод находит блок большего размера и при этом в него можна еще добавить новый заголовок, то создается новый заголовок после выделяемого в памяти блока, который указывает на оставшийся свободный участок памяти. При этом выделяемый блок помечается в заголовке как занятый и метод возвращает его адрес.

Если метод не может найти соответствующего выше описанным условиям блока памяти, то он возвращает null .

1. **public** **int** mem\_realloc(**int** addr, **int** size)

Если новый размер выделяемой области памяти совпадает со старым, то возвращается адрес данной области. Если размера данного блока достаточно для выделения блока с новым размером (случай когда новый размер меньше старого), то создается новый заголовок и возвращается адрес данного блока (тот же адрес, что передавался в метод). Если новый блок больше старого, то сначала проверяется возможность расширить за счет левого или правого “соседа”. Если не выполняется ни одно из выше перечисленных условий, то данный блок памяти освобождается и выделяется новый блок с помощью метода mem\_alloc(**int** size). Если выделить новый участок памяти не удалось, то метод восстанавливает начальное состояний памяти и возвращает null.

1. **public** **void** mem\_free(**int** addr)

Метод освобождает участок памяти с адресом **int** addr. При этом проверяются блоки памяти, которые находятся до и после данного блока в памяти. Если они свободны (или один из них), то они объединяются с данным блоком в один свободный блок.

1. **void** mem\_dump()

Метод выводит на консоль состояние областей памяти.

**Листинг:**

package spo.lab1;

/\*\*

\* Created by alexey on 09.04.16.

\*/

public class Main {

public static void main(String[] args) {

MemoryController mController = new MemoryController(30);

System.out.println("Begin:");

System.out.println(mController);

System.out.println("mem\_alloc(2)");

mController.mem\_alloc(2);

System.out.println(mController);

System.out.println("mem\_alloc(3)");

Block b3 = mController.mem\_alloc(3);

System.out.println(mController);

System.out.println("mem\_alloc(4)");

Block b4 = mController.mem\_alloc(4);

System.out.println(mController);

System.out.println("mem\_alloc(5)");

Block b5 = mController.mem\_alloc(5);

System.out.println(mController);

System.out.println("mem\_realloc(b5 2)");

b5 = mController.mem\_realloc(b5, 2);

System.out.println(mController);

System.out.println("mem\_free(b3)");

mController.mem\_free(b3);

System.out.println(mController);

System.out.println("mem\_free(b5)");

mController.mem\_free(b5);

System.out.println(mController);

System.out.println("mem\_realloc(b4, 8)");

b4 = mController.mem\_realloc(b4, 8);

System.out.println(mController);

System.out.println(mController.dump());

}

}

package spo.lab1;

/\*\*

\* Created by alexey on 08.04.16.

\*/

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

//import spo.lab1.BlockController.Block;

public class MemoryController {

private int[] memory;

private int memorySize;

private int totalFreeSize;

private Block first;

private Block last;

// Ініціалізація пам'яті

public MemoryController(int memorySize) {

memory = new int[memorySize];

this.memorySize = memorySize;

int offset = 0;

// Початковий блок-заглушка

first = new Block(0, false, 0, offset);

offset += first.record(memory);

Block main = new Block(

(memorySize - (Block.DESCRIPTOR\_SIZE + offset + 2)), true, 0,

offset);

offset += main.record(memory);

// Кінцевий блок-заглушка

last = new Block(0, false, 0, memorySize - 2);

offset += last.record(memory);

totalFreeSize = main.getSize();

}

public String toString() {

StringBuilder sb = new StringBuilder("Memory:\n[\n");

Block b = first;

sb.append("\t" + b + ",");

while (!b.equals(last)) {

b = Block.read(memory, b.getNextOffset());

sb.append("\n\t" + b + ", ");

}

sb.append("\n]");

return sb.toString();

}

public String dump() {

StringBuilder sb = new StringBuilder("Memory:\n");

sb.append(Arrays.toString(memory));

sb.append("\n");

return sb.toString();

}

public List<Block> getAllBlocks() {

ArrayList<Block> blocks = new ArrayList<>();

Block b = first;

do {

blocks.add(b);

b = Block.read(memory, b.getNextOffset());

} while(!b.equals(last));

return blocks;

}

public int alignSize(int size) {

int mod = size % Block.DESCRIPTOR\_SIZE;

return size + (mod == 0 ? 0 : (Block.DESCRIPTOR\_SIZE - mod));

}

public Block mem\_alloc(int size) {

size = alignSize(size);

if (totalFreeSize < size) {

return null;

} else {

// Перегляд всіх блоків з початку

Block block = Block.read(memory, first.getNextOffset());

while (!block.equals(last)) {

// Якщо блок вільний і його розмір не менший за заданий

if (block.isFree() && (block.getSize() >= size)) {

// Виділяємо

Block allocated = block.divide(size);

allocated.makeFree(false);

totalFreeSize -= allocated.record(memory);

// Частина, що залишилася

block.record(memory);

return allocated;

}

block = Block.read(memory, block.getNextOffset());

}

return null;

}

}

private Block divideAndMakeAndRecord(Block block, int size) {

Block reallocated = block.divide(size);

reallocated.makeFree(false);

reallocated.record(memory);

// Перевіряємо чи вільний блок справа від залишку

Block right = Block.read(memory, block.getNextOffset());

if (right.isFree() && !right.equals(last)) {

// Об'єднуємо залищок з правим

block = Block.merge(block, right);

}

block.makeFree(true);

block.record(memory);

return reallocated;

}

public Block mem\_realloc(Block block, int size) {

size = alignSize(size);

Block reallocated = null;

int delta = block.getSize() - size;

if (delta >= 0) {

// Зменшення розміру блока

reallocated = block.divide(size);

reallocated.makeFree(false);

reallocated.record(memory);

totalFreeSize += (delta);

// Перевіряємо чи вільний блок справа від залишку

Block right = Block.read(memory, block.getNextOffset());

if (right.isFree() && !right.equals(last)) {

// Об'єднуємо залищок з правим

block = Block.merge(block, right);

}

block.makeFree(true);

block.record(memory);

return reallocated;

} else {

// Перевіряємо чи вільний блок зліва

Block left = Block.read(memory, block.getPreviousOffset());

if (left.isFree() && !left.equals(first)

&& (left.getSize() + Block.DESCRIPTOR\_SIZE + delta >= 0)) {

System.arraycopy(memory, block.getOffset() + Block.DESCRIPTOR\_SIZE,

memory, left.getOffset() + Block.DESCRIPTOR\_SIZE, block.getSize());

// Об'єднуємо з лівим

left = Block.merge(left, block);

return divideAndMakeAndRecord(left, size);

}

// Перевіряємо чи вільний блок справа

Block right = Block.read(memory, block.getNextOffset());

if (right.isFree() && !right.equals(last)

&& (Block.DESCRIPTOR\_SIZE + right.getSize() + delta >= 0)) {

// Об'єднуємо з правим

block = Block.merge(block, right);

return divideAndMakeAndRecord(block, size);

}

//Перевіряємо чи можливе розширення за рахунок правого і лівого сусіда

if(left.isFree() && !left.equals(first) && right.isFree() && !right.equals(last)

&& (left.getSize() + 2\*Block.DESCRIPTOR\_SIZE + block.getSize() + right.getSize() >= 0)) {

System.arraycopy(memory, block.getOffset() + Block.DESCRIPTOR\_SIZE,

memory, left.getOffset() + Block.DESCRIPTOR\_SIZE, block.getSize());

// Об'єднуємо з лівим

block = Block.merge(left, block);

// Об'єднуємо з правим

block = Block.merge(block, right);

return divideAndMakeAndRecord(block, size);

}

//Розширити без перенесення не вдалося

reallocated = mem\_alloc(size);

if(reallocated != null) {

System.arraycopy(memory, block.getOffset() + Block.DESCRIPTOR\_SIZE,

memory, reallocated.getOffset() + Block.DESCRIPTOR\_SIZE, reallocated.getSize());

}

}

return reallocated;

}

public void mem\_free(Block start) {

// Перевіряємо чи вільний блок зліва

Block left = Block.read(memory, start.getPreviousOffset());

if (left.isFree() && !left.equals(first)) {

// Об'єднуємо з лівим

start = Block.merge(left, start);

}

// Перевіряємо чи вільний блок справа

Block right = Block.read(memory, start.getNextOffset());

if (right.isFree() && !right.equals(last)) {

// Об'єднуємо з правим

start = Block.merge(start, right);

}

start.makeFree(true);

start.record(memory);

totalFreeSize += start.getSize() + Block.DESCRIPTOR\_SIZE;

}

}

package spo.lab1;

/\*\*

\* Created by alexey on 07.04.16.

\*/

public class Block {

public static final int DESCRIPTOR\_SIZE = 2;

private int header;

private int prevSize;

//"Вказівник" = зміщення дескриптора (заголовку) в області пам'яті

private int offset;

private Block(int header, int prevSize, int offset) {

this.header = header;

this.prevSize = prevSize;

this.offset = offset;

}

public Block(int size, boolean free, int prevSize, int offset) {

if(size < 0) {

size = 0;

}

this.header = size;

this.prevSize = prevSize;

this.offset = offset;

makeFree(free);

}

public boolean isFree() {

return (header >= 0);

}

public void makeFree(boolean free) {

if(free) {

header = getSize();

} else {

header = -getSize();

}

}

/\*\*

\* @return розмір тіла блоку (без урахування розміру дескриптора)

\*/

public int getSize() {

return Math.abs(header);

}

public int getPrevSize() {

return prevSize;

}

//

public int getOffset() {

return offset;

}

public int getBegin() {

return (offset + DESCRIPTOR\_SIZE);

}

public int getNextOffset() {

return (offset + DESCRIPTOR\_SIZE + this.getSize());

}

public int getPreviousOffset() {

return (offset - (DESCRIPTOR\_SIZE + prevSize));

}

/\*\*

\* @param memory - масив, який представляє пам'ять

\* @param offset - "вказівник" на блок = зміщення дескриптора (заголовку) в області пам'яті

\* @return блок по його "вказівнику"

\*/

public static Block read(int[] memory, int offset) {

return new Block(memory[offset], memory[offset + 1], offset);

}

public static boolean rewrite = true;

public int record(int[] memory) {

memory[offset] = header;

memory[offset + 1] = prevSize;

if (rewrite) {

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

memory[offset + 2 + i] = (i + 1) \* 100;

}

}

return (DESCRIPTOR\_SIZE + this.getSize());

}

public Block divide(int size) {

Block left = new Block(size, prevSize, offset);

//Встановлюємо новий дескриптор правої частини

header = getSize() - (DESCRIPTOR\_SIZE + size);

prevSize = size;

offset += DESCRIPTOR\_SIZE + size;

return left;

}

public static Block merge(Block left, Block right) {

return new Block(left.getSize() + DESCRIPTOR\_SIZE + right.getSize(), left.isFree(), left.prevSize, left.offset);

}

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

Block other = (Block) obj;

if (header != other.header)

return false;

if (offset != other.offset)

return false;

if (prevSize != other.prevSize)

return false;

return true;

}

@Override

public String toString() {

return "{offset = " + offset + ", size = " + (Math.abs(header) + DESCRIPTOR\_SIZE) + "(" + header + "), prevSize = " + prevSize + "}";

}

}