Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Курсовая работа по курсу**

**«Операционные системы»**

**ПРОЕКТИРОВАНИЕ КОНСОЛЬНОЙ КЛИЕНТ-СЕРВЕРНОЙ ИГРЫ**

Студент: Казарцев Денис Вячеславович

Группа: М8О–210Б–22

Вариант: 21

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2023.

**Постановка задачи**

## Цель работы:

* Приобретение практических навыков в использовании знаний, полученных в течение курса
* Проведение исследования в выбранной предметной области

## Задание

## Необходимо сравнить два алгоритма аллокации: списки свободных блоков (наиболее подходящее) и алгоритм двойников

## Общий метод и алгоритм решения.

Для реализации поставленной задачи необходимо:

1. Изучить аллокации списков свободных переменных и алгоритма двойников.
2. Создать аллокации списков свободных переменных и алгоритма двойников.
3. Создать тесты для сравнения аллокаций.

## Основные файлы программы

**buddySystemAllocator.h:**

#ifndef \_\_BUDDY\_SYSTEM\_ALLOCATOR\_\_

#define \_\_BUDDY\_SYSTEM\_ALLOCATOR\_\_

#include <stdint.h>

#include <stddef.h>

#include "buddySystemBlockInfo.h"

#define MIN\_BLOCK\_SIZE 32

#define MIN\_POWER\_OF\_TWO 5

#define MAX\_POWER\_OF\_TWO 64

typedef struct \_Allocator {

void\* memory;

size\_t memorySize;

} Allocator;

void printBlocks(Allocator\*);

size\_t align(size\_t);

size\_t getPowerOf2(size\_t);

void initArrayOfBlocks();

Allocator\* createMemoryAllocator(size\_t);

void destroyMemoryAllocator(Allocator\*);

void\* allocBlock(Allocator\*, size\_t);

void\* recursiveAlloc(Allocator\*, size\_t, size\_t, size\_t);

void freeBlock(const Allocator\*, void\*);

BlockInfo\* getBuddy(const Allocator\*, BlockInfo\*);

void concatenateBlocks(const Allocator\*, BlockInfo\*);

#endif // \_\_BUDDY\_SYSTEM\_ALLOCATOR\_\_

**buddySystemAllocator:**

#include "../include/buddySystemAllocator.h"

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include "../include/buddySystemLinkedList.h"

BlockInfo\* blocks[MAX\_POWER\_OF\_TWO - MIN\_POWER\_OF\_TWO + 1];

void printBlocks(Allocator\* allocator) {

for (int i = 0; i < 10; ++i) {

printf("I = %d\n", i);

BlockInfo\* currBlock = blocks[i];

printf("Block size = %u\n", currBlock ? currBlock->size : 0);

while (currBlock != NULL) {

printf(" (%zu) ", (uint8\_t\*)currBlock - (uint8\_t\*)allocator->memory);

currBlock = currBlock->next;

}

printf("\n");

}

}

size\_t align(size\_t memorySize) {

size\_t currentSize = 1;

while (currentSize < memorySize) {

currentSize <<= 1;

}

return currentSize;

}

size\_t getPowerOf2(size\_t value) {

size\_t currentPower = 0;

size\_t currentValue = 1;

while (currentValue < value) {

currentValue <<= 1;

++currentPower;

}

return currentPower;

}

void initArrayOfBlocks() {

for (size\_t i = 0; i <= MAX\_POWER\_OF\_TWO - MIN\_POWER\_OF\_TWO; ++i) {

blocks[i] = NULL;

}

}

Allocator\* createMemoryAllocator(size\_t memorySize) {

if (memorySize == 0) {

fprintf(stderr, "Invalid memory size 0\n");

exit(1);

}

Allocator\* allocator = malloc(sizeof(Allocator));

if (allocator == NULL) {

perror("Can't create allocator");

exit(1);

}

memorySize += sizeof(BlockInfo); // Добавляем размер структуры

memorySize = align(memorySize); // Выравниваем саму память

uint8\_t\* memory = malloc(memorySize); // malloc выравнивает память по 8 (или 16 или 32...)

if (memory == NULL) {

fprintf(stderr, "Can't allocate memory\n");

exit(1);

}

allocator->memory = (void\*)memory;

allocator->memorySize = memorySize;

BlockInfo block = {

memorySize,

true,

NULL,

NULL,

};

BlockInfo\* firstBlock = (BlockInfo\*)allocator->memory;

\*firstBlock = block;

// Кладем в массив указателей первый, наибольший блок

initArrayOfBlocks();

blocks[getPowerOf2(memorySize) - MIN\_POWER\_OF\_TWO] = firstBlock;

return allocator;

}

void\* allocBlock(Allocator\* allocator, size\_t blockSize) {

// Число байт, которое необходимо, с учетом того, что входит еще и структура

blockSize += sizeof(BlockInfo);

blockSize = align(blockSize);

size\_t powerOf2 = getPowerOf2(blockSize);

if (powerOf2 > MAX\_POWER\_OF\_TWO) {

fprintf(stderr, "Can't allocate memory\n");

exit(1);

}

// Проверим, есть ли выше блок, из которого можно выделить нужное количество памяти

size\_t firstGoodBlockID = -1;

for (size\_t i = powerOf2 - MIN\_POWER\_OF\_TWO; i <= MAX\_POWER\_OF\_TWO - MIN\_POWER\_OF\_TWO; ++i) {

if (blocks[i] != NULL) {

firstGoodBlockID = i;

break;

}

}

if (firstGoodBlockID == (size\_t)-1) {

#ifdef INFO

for (int i = 0; i <= MAX\_POWER\_OF\_TWO - MIN\_POWER\_OF\_TWO; ++i) {

printf("I = %d: P = %p\n", i, (void\*)blocks[i]);

}

#endif

fprintf(stderr, "Can't allocate memory\n");

exit(1);

}

return recursiveAlloc(allocator, powerOf2, firstGoodBlockID, 1 << (firstGoodBlockID + MIN\_POWER\_OF\_TWO));

}

void\* recursiveAlloc(Allocator\* allocator, size\_t powerOfTwo, size\_t goodBlockId, size\_t blockSize) {

BlockInfo\* block = pop(&blocks[goodBlockId]);

if (powerOfTwo == goodBlockId + MIN\_POWER\_OF\_TWO) {

block->isFree = false;

return (uint8\_t\*)block + sizeof(BlockInfo);

}

size\_t newBlockSize = blockSize / 2;

size\_t newBlockID = goodBlockId - 1;

block->size = newBlockSize;

BlockInfo secondBlock = {

newBlockSize,

true,

NULL,

NULL,

};

// Находим место в память для блока близнеца, кладем туда новый блок

BlockInfo\* secondBlockPtr = (BlockInfo\*)(((uint8\_t\*)block) + newBlockSize);

\*secondBlockPtr = secondBlock;

push(&blocks[newBlockID], secondBlockPtr);

push(&blocks[newBlockID], block);

return recursiveAlloc(allocator, powerOfTwo, newBlockID, newBlockSize);

}

void freeBlock(const Allocator\* allocator, void\* block) {

BlockInfo\* currentBlock = (BlockInfo\*)(((uint8\_t\*)block) - sizeof(BlockInfo));

if (currentBlock->isFree) {

fprintf(stderr, "This block already free, can't free again\n");

exit(1);

}

currentBlock->isFree = true;

concatenateBlocks(allocator, currentBlock);

}

void concatenateBlocks(const Allocator\* allocator, BlockInfo\* currentBlock) {

// printf("Start concat\n");

BlockInfo\* buddy = getBuddy(allocator, currentBlock);

size\_t blockID = getPowerOf2(currentBlock->size) - MIN\_POWER\_OF\_TWO;

if (currentBlock < buddy) {

if (buddy->isFree && buddy->size == currentBlock->size) {

// printf("HERE1\n");

removeBlock(&blocks[blockID], buddy);

currentBlock->size \*= 2;

concatenateBlocks(allocator, currentBlock);

} else {

push(&blocks[blockID], currentBlock);

}

} else if (currentBlock > buddy) {

if (buddy->isFree && buddy->size == currentBlock->size) {

// printf("HERE2\n");

removeBlock(&blocks[blockID], buddy);

// printf("HERE22\n");

buddy->size \*= 2;

concatenateBlocks(allocator, buddy);

} else {

push(&blocks[blockID], currentBlock);

}

} else {

fprintf(stderr, "Something went wrong ...\n");

exit(1);

}

// printf("End concat\n");

}

BlockInfo\* getBuddy(const Allocator\* allocator, BlockInfo\* currentBlock) {

return (BlockInfo\*)((uint8\_t\*)allocator->memory + (((size\_t)(((uint8\_t\*)currentBlock) - ((uint8\_t\*)allocator->memory))) ^ ((size\_t)currentBlock->size)));

}

void destroyMemoryAllocator(Allocator\* allocator) {

free(allocator->memory);

free(allocator);

}

**resourceMapAllocator.cpp:**

#include "../include/resourceMapAllocator.h"

#include <stdlib.h>

#include <stdio.h>

#include <stdint.h>

#include "../include/resourceMapBlockInfo.h"

size\_t align(size\_t memorySize) {

return memorySize + (ALIGN\_BY - (memorySize % ALIGN\_BY)) % ALIGN\_BY;

}

Allocator\* createMemoryAllocator(size\_t memorySize) {

if (memorySize == 0) {

fprintf(stderr, "Invalid memory size 0\n");

exit(1);

}

Allocator\* allocator = malloc(sizeof(Allocator));

if (allocator == NULL) {

perror("Can't create allocator");

exit(1);

}

// Добавляем размер структуры

memorySize += ALIGN\_BY;

memorySize = align(memorySize); // Выравниваем саму память

uint8\_t\* memory = malloc(memorySize); // malloc выравнивает память по 8 (или 16 или 32...)

#ifdef INFO

printf("ALLOCATED TOTAL %zu\n", memorySize);

#endif

if (memory == NULL) {

fprintf(stderr, "Can't allocate memory\n");

exit(1);

}

allocator->memory = (void\*)memory;

allocator->memorySize = memorySize;

// В начало ставим блок, который указывает на всю память

BlockInfo block = {

NULL,

memorySize - ALIGN\_BY,

};

BlockInfo\* firstBlock = (BlockInfo\*)allocator->memory;

\*firstBlock = block;

setBlockFree(firstBlock);

allocator->firstFreeBLock = firstBlock;

return allocator;

}

size\_t getBlockLengthByGivenMemory(void\* memory) {

return ((BlockInfo\*)((uint8\_t\*)memory - ALIGN\_BY))->blockSize;

}

// Сдвиг начала блока относительно начала выделенной памяти

size\_t getOffset(Allocator\* allocator, BlockInfo\* block) {

return (size\_t)((uint8\_t\*)block - (uint8\_t\*)allocator->memory);

}

void\* allocBlock(Allocator\* allocator, size\_t requestedMemory) {

requestedMemory = align(requestedMemory);

BlockInfo\* currentBlock = allocator->firstFreeBLock;

BlockInfo\* prevBlock = NULL;

BlockInfo\* bestBlock = NULL;

BlockInfo\* bestPrevBlock = NULL;

size\_t minLength = -1;

while (currentBlock != NULL) {

size\_t currentBlockSize = currentBlock->blockSize;

if (currentBlockBetter(currentBlockSize, requestedMemory, minLength)) {

bestBlock = currentBlock;

bestPrevBlock = prevBlock;

minLength = currentBlockSize;

}

prevBlock = currentBlock;

currentBlock = resetToNormalPointer(currentBlock->nextBlock);

}

if (bestBlock == NULL) {

fprintf(stderr, "Can't allocate new memory\n");

exit(1);

}

// Можно выделить память еще для следующего блока, он будет свободным

if (requestedMemory + sizeof(BlockInfo) <= bestBlock->blockSize) {

// Расположение нового блока

BlockInfo\* newBlock = (BlockInfo\*)((uint8\_t\*)bestBlock + sizeof(BlockInfo) + requestedMemory);

newBlock->nextBlock = bestBlock->nextBlock;

newBlock->blockSize = bestBlock->blockSize - requestedMemory - sizeof(BlockInfo);

setBlockFree(newBlock);

// Меняем блок, из которого была выделена памяти

setBlockOccupied(bestBlock);

bestBlock->blockSize -= (sizeof(BlockInfo) + newBlock->blockSize);

// Удаляем этот блок из списка

if (bestPrevBlock == NULL) {

allocator->firstFreeBLock = newBlock;

} else {

bestPrevBlock->nextBlock = newBlock;

}

} else { // не хватило памяти под следующий блок

setBlockOccupied(bestBlock);

if (bestPrevBlock == NULL) {

allocator->firstFreeBLock = bestBlock->nextBlock;

} else {

bestPrevBlock->nextBlock = bestBlock->nextBlock;

}

}

return (void\*)(((uint8\_t\*)bestBlock) + ALIGN\_BY);

}

void freeBlock(Allocator\* allocator, void\* memoryBlock) {

BlockInfo\* blockForFree = (BlockInfo\*)((uint8\_t\*)memoryBlock - ALIGN\_BY);

if (!(allocator->memory <= memoryBlock && memoryBlock <= (void\*)((uint8\_t\*)allocator->memory + allocator->memorySize)) || isBlockFree(blockForFree)) {

fprintf(stderr, "Can't free this block\n");

exit(1);

}

// Будем проходится по все блокам, до тех пор, пока не найдем, куда поставить нвоый блок

BlockInfo\* currentBlock = allocator->firstFreeBLock;

if (currentBlock == blockForFree) {

fprintf(stderr, "Can't free this block\n");

exit(1);

}

// Надо добавить блок в самое начало

if (blockForFree < currentBlock || currentBlock == NULL) {

blockForFree->nextBlock = currentBlock;

setBlockFree(blockForFree);

allocator->firstFreeBLock = blockForFree;

} else { // Блок будет находиться не в начале

while (currentBlock->nextBlock < blockForFree && resetToNormalPointer(currentBlock->nextBlock) != NULL) {

currentBlock = resetToNormalPointer(currentBlock->nextBlock);

}

blockForFree->nextBlock = resetToNormalPointer(currentBlock->nextBlock);

currentBlock->nextBlock = blockForFree;

setBlockFree(blockForFree);

setBlockFree(currentBlock);

}

currentBlock = allocator->firstFreeBLock;

while (currentBlock != NULL) {

if (canConcatenate(currentBlock)) {

currentBlock->blockSize += ALIGN\_BY + resetToNormalPointer(currentBlock->nextBlock)->blockSize;

currentBlock->nextBlock = resetToNormalPointer(currentBlock->nextBlock)->nextBlock;

continue;

}

currentBlock = resetToNormalPointer(currentBlock->nextBlock);

}

}

bool canConcatenate(BlockInfo\* currentBlock) {

return resetToNormalPointer(currentBlock->nextBlock) == (BlockInfo\*)((uint8\_t\*)currentBlock + ALIGN\_BY + currentBlock->blockSize);

}

bool currentBlockBetter(size\_t currentBlockLength, size\_t requestedMemory, size\_t minLength) {

return currentBlockLength >= requestedMemory && currentBlockLength < minLength;

}

void destroyMemoryAllocator(Allocator\* allocator) {

free(allocator->memory);

free(allocator);

}

**resourceMapAllocator.h:**

#ifndef \_\_RESOURCE\_MAP\_ALLOCATOR\_\_

#define \_\_RESOURCE\_MAP\_ALLOCATOR\_\_

#include <stddef.h>

#include "resourceMapBlockInfo.h"

#define ALIGN\_BY (sizeof(BlockInfo))

typedef struct \_Allocator {

void\* memory;

size\_t memorySize;

BlockInfo\* firstFreeBLock;

} Allocator;

size\_t align(size\_t);

Allocator\* createMemoryAllocator(size\_t);

void destroyMemoryAllocator(Allocator\*);

void\* allocBlock(Allocator\*, size\_t);

void freeBlock(Allocator\*, void\*);

size\_t getBlockLengthByGivenMemory(void\*);

void concatenateBlocks(Allocator\*);

size\_t getOffset(Allocator\*, BlockInfo\*);

#endif // \_\_RESOURCE\_MAP\_ALLOCATOR\_\_

## Вывод

В ходе лабораторной работы я изучил аллокаторы, а также сравнил работу двух различных аллокаторов. Программа показала, что аллокаторы различаются не только теоретически, но и практически(программно).