Національний технічний університет України

«Київський політехнічний інститут»

Факультет інформатики і обчислювальної техніки

Кафедра АСОІУ

**Лабораторна робота №1**

**з курсу: «Системне програмування»**

*Виконала:*

студент групи ІС-72

Березянко К.

*Перевірив:*

Дифучин А.

2020 р.

**Опис розроблених алгоритмів**:

1. Ініціалізація загального пулу. Пул блоків пам’яті виділяється за допомогою стандартної функції виділення пам’яті malloc:

firstBlock = (el\_t\*)malloc(POOL\_SIZE \* sizeof(el\_t))

де firstBlock – вказівник на початок пулу, el\_t – тип мінімального елемента пулу:

typedef char el\_t;

Кожен блок виділеної пам’яті містить заголовок:

typedef struct {

char busyState;

el\_t checkSum;

size\_t size;

size\_t prevSize;

} Header;

Відповідно до опису заголовок має займати 1 + 1 + 4 + 4 = 10 байт, але враховуючи особливості компіляції, він буде займати 4 \* 3 = 12 байт.

Під час ініціалізації виділяється 3 блоки:

Початковий зайнятий блок-бар’єр нульового розміру.

Кінцевий зайнятий блок-бар’єр нульового розміру.

Проміжний вільний блок.

Звільнення загального пулу. Виконується за допомогою стандартної функції free:

free(firstBlock)

Виділення пам’яті заданого розміру mem\_alloc(size\_t size). Передбачені 2 алгоритми пошуку потрібного блоку для виділення пам’яті:

За необхідний вибирається перший знайдений підходящий блок, тобто такий, розмір якого не менший за size.

За необхідний вибирається підходящий блок з мінімальним розміром.

Якщо блок не знайдений, то повертається NULL, якщо знайдений – то виконується функція виділення пам’яті в заданому блоці.

Функція виділення пам’яті в заданому блоці freeBlockAlloc(Header \*headerPtr, size\_t size).

Якщо розмір блоку дещо більший за size, то він ділиться на 2 блоки.

Якщо розмір блоку майже дорівнює size, то блок залишається неподільним. Повертається вказівник на заголовок блоку.

Перевиділення пам’яті mem\_realloc(void \*addr, size\_t size).

Якщо addr = NULL, то виконується виділення пам’яті заданого розміру mem\_alloc(size).

Якщо розмір блоку, що виділений під данні вказівника addr, більший за size, то перевиділяємо пам’ять блоку за допомогою freeBlockAlloc(headerPtr, size).

У протилежному випадку намагаємося об’єднати даний блок з вільним правим (без перенесення даних) і/або лівим (з перенесенням даних) й перевиділити пам’ять блоку.

Якщо цього виявилося недостатньо, то знаходимо вільне місце для даних з допомогою mem\_alloc(), переносимо дані й звільняємо даний блок.

1. Звільнення пам’яті mem\_free(void \*addr). Помічаємо блок як вільний і об’єднуємо даний блок с сусідніми вільними блоками.

**Оцінка часу пошуку вільного блоку пам’яті**:

* Алгоритм «перший потрібний блок»: O(N).
* Алгоритм «підходящий з мінімальним розміром»: O(2 \* N).

N – кількість блоків в пулі.

**Оцінка часу звільнення занятого блоку**: O(1).

**Оцінка витрати пам'яті для зберігання службовох інформації:** 12 \* N байт.

N – кількість блоків в пулі.

**Переваги аллокатора:**

1. Невеликий об’єм службової інформації.
2. Малий час виконання звільнення блоку.

**Недоліки аллокатора:**

1. Великий час пошуку нового блоку( лінійна складність)
2. При виділенні блоку можливе надмірне виділення пам’яті у розмірі, меншому за розмір заголовку.

**Файл main.сpp:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include "lab1.h"

#define ARRAY\_SIZE 10

#define TEST\_BLOCK\_SIZE 70

#define BLOCK\_SIZE 50

void infoAndWait(char message[]) {

printf("%s\n", message);

mem\_dump();

getchar();

}

void simpleMemoryTest() {

printf("------------ Simple Memory Test ------------\n");

char \*blocksArray[ARRAY\_SIZE];

poolInitialize();

infoAndWait("Initial pool:");

for ( int i = 0; i < ARRAY\_SIZE; i++ ) {

blocksArray[i] = (char\*)mem\_alloc(50);

}

infoAndWait("After memory allocation:");

mem\_free(blocksArray[8]);

infoAndWait("After blocksArray[8] memory free:");

blocksArray[4] = (char\*)mem\_realloc(blocksArray[4], 20);

infoAndWait("After blocksArray[4] memory reallocation to size 20:");

mem\_free(blocksArray[5]);

infoAndWait("After blocksArray[5] memory free:");

blocksArray[6] = (char\*)mem\_realloc(blocksArray[6], 200);

infoAndWait("After blocksArray[6] memory reallocation to size 200:");

blocksArray[4] = (char\*)mem\_realloc(blocksArray[4], 100);

infoAndWait("After blocksArray[4] memory reallocation to 100:");

blocksArray[5] = (char\*)mem\_alloc(70);

infoAndWait("After blocksArray[5] memory allocation to 70:");

for ( int i = 0; i <= 7; i++ ) {

mem\_free(blocksArray[i]);

}

mem\_free(blocksArray[9]);

infoAndWait("After all memory free:");

poolFree();

printf("Free pool\n");

getchar();

}

void checkSumTest() {

printf("------------ Check Sum Test ------------\n");

el\_t \*testBlock;

el\_t \*blocksArray[ARRAY\_SIZE];

el\_t checkSum = 0;

srand((unsigned int)time(NULL));

poolInitialize();

infoAndWait("Initial pool:");

testBlock = (el\_t\*)mem\_alloc(TEST\_BLOCK\_SIZE \* sizeof(char));

infoAndWait("After testBlock allocation:");

for ( int i = 0; i < TEST\_BLOCK\_SIZE; i++ ) {

testBlock[i] = rand() % 256;

checkSum ^= testBlock[i];

}

updateCheckSums();

infoAndWait("After check sums of testBlock updating:");

for ( int i = 0; i < ARRAY\_SIZE; i++ ) {

blocksArray[i] = (char\*)mem\_alloc(BLOCK\_SIZE);

for ( int j = 0; j < BLOCK\_SIZE; j++ ) {

blocksArray[i][j] = rand() % 256;

}

}

updateCheckSums();

infoAndWait("After memory allocation:");

while ( 1 ) {

size\_t size = rand() % (2 \* BLOCK\_SIZE) + BLOCK\_SIZE / 2;

int index = rand() % ARRAY\_SIZE;

blocksArray[index] = (char\*)mem\_realloc(blocksArray[index], size);

if ( blocksArray[index] == NULL ) {

break;

}

for ( size\_t j = 0; j < size; j++ ) {

blocksArray[index][j] = rand() % 256;

}

}

updateCheckSums();

infoAndWait("After memory realocating many times:\n");

printf("Previous check sum = %u\n\n", (unsigned char) checkSum);

poolFree();

printf("Free pool\n");

getchar();

}

int main() {

simpleMemoryTest();

checkSumTest();

return 0;

}

**Файл lab1.h:**

#define POOL\_SIZE 1024

#define MIN\_BLOCK\_SIZE 4

typedef char el\_t;

typedef enum {

FIRST\_FOUND\_ALGORITHM,

MIN\_SIZE\_ALGORITHM

} SearchAlgorithm;

void poolInitialize();

void poolFree();

void setSearchAlgorithm(SearchAlgorithm algorithm);

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

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

void mem\_free(void \*addr);

void mem\_dump();

void updateCheckSums();

**Файл lab1сpp:**

#include "lab.h"

#include <malloc.h>

#include <stdio.h>

#define FREE\_STATE 0

#define BUSY\_STATE 1

#define DEFAULT\_FILLER 0

#define DEFAULT\_CHECK\_SUM 0

typedef struct {

char busyState;

el\_t checkSum;

size\_t size;

size\_t prevSize;

} Header;

typedef Header\* (\*SearchAlgorithmFun)(size\_t size);

el\_t \*firstBlock;

int blocksQuantity = 3;

SearchAlgorithm sAlgorithm = FIRST\_FOUND\_ALGORITHM;

Header \*firstFoundSAlgorithm(size\_t size);

Header \*minSizeSAlgorithm(size\_t size);

SearchAlgorithmFun sAlgFunArray[] = {

firstFoundSAlgorithm,

minSizeSAlgorithm

};

size\_t headerSize = sizeof(Header) / sizeof(el\_t);

Header \*fillHeader(Header \*headerPtr, size\_t size, size\_t prevSize, char busyState) {

headerPtr->busyState = busyState;

headerPtr->checkSum = DEFAULT\_CHECK\_SUM;

headerPtr->size = size;

headerPtr->prevSize = prevSize;

return headerPtr;

}

Header \*getNextHeader(Header \*lastHeaderPtr) {

return (Header\*)((el\_t\*)lastHeaderPtr + headerSize + lastHeaderPtr->size);

}

Header \*getPrevHeader(Header \*lastHeaderPtr) {

return (Header\*)((el\_t\*)lastHeaderPtr - lastHeaderPtr->prevSize - headerSize);

}

int isAppropriateBlock(Header \*headerPtr, size\_t size) {

return headerPtr->busyState == FREE\_STATE && headerPtr->size >= size;

}

void poolInitialize() {

firstBlock = (el\_t\*)malloc(POOL\_SIZE \* sizeof(el\_t));

for ( el\_t \*ptr = firstBlock, \*lastPtr = ptr + POOL\_SIZE - 1;

ptr <= lastPtr; ptr++ ) {

\*ptr = DEFAULT\_FILLER;

}

size\_t freeSpace = POOL\_SIZE - 3 \* headerSize;

el\_t \*ptr = firstBlock;

fillHeader((Header\*) ptr, 0, 0, BUSY\_STATE);

ptr += headerSize;

fillHeader((Header\*) ptr, freeSpace, 0, FREE\_STATE);

ptr += headerSize + freeSpace;

fillHeader((Header\*) ptr, 0, freeSpace, BUSY\_STATE);

}

void poolFree() {

free(firstBlock);

}

void setSearchAlgorithm(SearchAlgorithm algorithm) {

sAlgorithm = algorithm;

}

Header \*firstFoundSAlgorithm(size\_t size) {

Header \*headerPtr = (Header\*)firstBlock + 1;

for ( int i = 1, last = blocksQuantity - 2; i <= last; i++ ) {

if ( isAppropriateBlock(headerPtr, size) ) {

return headerPtr;

}

headerPtr = getNextHeader(headerPtr);

}

return NULL;

}

Header \*minSizeSAlgorithm(size\_t size) {

Header \*headerPtr = (Header\*)firstBlock + 1;

size\_t minBlockSize = POOL\_SIZE;

int minBlockIndex = -1;

for ( int i = 1, last = blocksQuantity - 2; i <= last; i++ ) {

if ( isAppropriateBlock(headerPtr, size) && minBlockSize > headerPtr->size ) {

minBlockSize = headerPtr->size;

minBlockIndex = i;

}

headerPtr = getNextHeader(headerPtr);

}

if ( minBlockIndex == -1 ) {

return NULL;

}

headerPtr = (Header\*)firstBlock + 1;

for ( int i = 1; i < minBlockIndex; i++ ) {

headerPtr = getNextHeader(headerPtr);

}

return headerPtr;

}

void \*freeBlockAlloc(Header \*headerPtr, size\_t size) {

size\_t restSpace = headerPtr->size - size;

if ( restSpace > headerSize ) {

Header \*newHeaderPtr = (Header\*)((el\_t\*)headerPtr + headerSize + size);

fillHeader(newHeaderPtr, restSpace - headerSize, size, FREE\_STATE);

getNextHeader(newHeaderPtr)->prevSize = newHeaderPtr->size;

headerPtr->size = size;

blocksQuantity++;

}

headerPtr->busyState = BUSY\_STATE;

return (void\*)(headerPtr + 1);

}

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

Header \*headerPtr = (\*sAlgFunArray[sAlgorithm])(size);

if ( headerPtr == NULL ) {

return NULL;

}

return freeBlockAlloc(headerPtr, size);

}

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

if ( addr == NULL ) {

return mem\_alloc(size);

}

Header \*headerPtr = (Header\*)addr - 1;

size\_t freeSpace = headerPtr->size;

if ( freeSpace > size ) {

return freeBlockAlloc(headerPtr, size);

}

Header \*nextHeaderPtr = getNextHeader(headerPtr);

if ( nextHeaderPtr->busyState == FREE\_STATE ) {

freeSpace += nextHeaderPtr->size + headerSize;

if ( freeSpace > size ) {

headerPtr->size = freeSpace;

blocksQuantity--;

return freeBlockAlloc(headerPtr, size);

}

}

el\_t \*oldPtr = (el\_t\*)addr;

el\_t \*newPtr;

Header \*prevHeaderPtr = getPrevHeader(headerPtr);

if ( prevHeaderPtr->busyState == FREE\_STATE ) {

freeSpace += prevHeaderPtr->size + headerSize;

if ( freeSpace > size) {

prevHeaderPtr->size = freeSpace;

blocksQuantity--;

newPtr = (el\_t\*)(prevHeaderPtr + 1);

for ( size\_t i = 0; i < headerPtr->size; i++ ) {

newPtr[i] = oldPtr[i];

}

return freeBlockAlloc(prevHeaderPtr, size);

}

}

newPtr = (el\_t\*)mem\_alloc(size);

if ( newPtr == NULL ) {

return NULL;

}

for ( size\_t i = 0; i < headerPtr->size; i++ ) {

newPtr[i] = oldPtr[i];

}

mem\_free((void\*)oldPtr);

return newPtr;

}

void mem\_free(void \*addr) {

Header \*headerPtr = (Header\*)addr - 1;

Header \*nextHeaderPtr = getNextHeader(headerPtr);

Header \*prevHeaderPtr = getPrevHeader(headerPtr);

headerPtr->busyState = FREE\_STATE;

if ( nextHeaderPtr->busyState == FREE\_STATE ) {

headerPtr->size += nextHeaderPtr->size + headerSize;

getNextHeader(headerPtr)->prevSize = headerPtr->size;

blocksQuantity--;

}

if ( prevHeaderPtr->busyState == FREE\_STATE ) {

prevHeaderPtr->size += headerPtr->size + headerSize;

getNextHeader(prevHeaderPtr)->prevSize = prevHeaderPtr->size;

blocksQuantity--;

}

}

void mem\_dump() {

Header \*headerPtr = (Header\*) firstBlock;

printf("Blocks quantity: %d\n", blocksQuantity);

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

printf("Block %d:\n", i);

printf("\tBusy state: %s\n", headerPtr->busyState == BUSY\_STATE ? "busy" : "free");

printf("\tCheck sum: %d\n", (unsigned char)headerPtr->checkSum);

printf("\tBlock size: %u\n", headerPtr->size);

printf("\tPrevious block size: %d\n", headerPtr->prevSize);

headerPtr = getNextHeader(headerPtr);

}

printf("----------------------------------------\n");

}

void updateCheckSums() {

Header \*headerPtr = (Header\*)firstBlock + 1;

for ( int i = 1, last = blocksQuantity - 2; i <= last; i++ ) {

el\_t checkSum = 0;

for ( el\_t \*ptr = (el\_t\*)(headerPtr + 1), \*lastPtr = ptr + headerPtr->size - 1;

ptr <= lastPtr; ptr++ ) {

checkSum ^= \*ptr;

}

headerPtr->checkSum = checkSum;

headerPtr = getNextHeader(headerPtr);

}

}

**Приклад роботи аллокатора:**

Blocks quantity: 19

Block 0:

Busy state: busy

Check sum: 0

Block size: 0

Previous block size: 0

Block 1:

Busy state: busy

Check sum: 44

Block size: 70

Previous block size: 0

Block 2:

Busy state: busy

Check sum: 254

Block size: 50

Previous block size: 70

Block 3:

Busy state: free

Check sum: 14

Block size: 50

Previous block size: 50

Block 4:

Busy state: busy

Check sum: 121

Block size: 50

Previous block size: 50

Block 5:

Busy state: busy

Check sum: 169

Block size: 50

Previous block size: 50

Block 6:

Busy state: free

Check sum: 69

Block size: 22

Previous block size: 50

Block 7:

Busy state: free

Check sum: 110

Block size: 16

Previous block size: 22

Block 8:

Busy state: busy

Check sum: 73

Block size: 50

Previous block size: 16

Block 9:

Busy state: busy

Check sum: 189

Block size: 50

Previous block size: 50

Block 10:

Busy state: busy

Check sum: 14

Block size: 50

Previous block size: 50

Block 11:

Busy state: busy

Check sum: 197

Block size: 50

Previous block size: 50

Block 12:

Busy state: busy

Check sum: 233

Block size: 106

Previous block size: 50

Block 13:

Busy state: free

Check sum: 193

Block size: 19

Previous block size: 106

Block 14:

Busy state: busy

Check sum: 115

Block size: 31

Previous block size: 19

Block 15:

Busy state: free

Check sum: 126

Block size: 20

Previous block size: 31

Block 16:

Busy state: busy

Check sum: 62

Block size: 104

Previous block size: 20

Block 17:

Busy state: free

Check sum: 0

Block size: 8

Previous block size: 104

Block 18:

Busy state: busy

Check sum: 0

Block size: 0

Previous block size: 8