**Національний технічний університет України**

**«Київський політехнічний інститут імені Ігоря Сікорського»**

**Факультет інформатики та обчислювальної техніки**

**Кафедра автоматизації систем обробки інформації та управління**

**“Сучасні операційні системи”**

**Лабораторна робота №1**

**Перевірив: Дифучин А. Ю.**

**Виконав: Загинайло Є.О.**

**Студент гр. ІС-73, ФІОТ,**

**3 курс**

Київ

НТУУ «КПІ ім. І. Сікорського»

2020

**Тема**: Алокатор пам’яті загального призначення .

**Мета**: розробити алокатор загального призначення.

**Опис розробленого алгоритму**:

1. Виділення пам’яті через створення масиву типу int.

Кожен блок виділеної пам’яті містить заголовок:

Заголовок займає 12 байт.

Під час ініціалізації виділяється 1 вільний блок, який займає всю доступну пам’ять масиву.

1. Виділення пам’яті заданого розміру mem\_alloc(size\_t size):

Вибирається перший знайдений підходящий блок, тобто такий, розмір якого не менший за size. Якщо розмір більший необхідного, блок розбивається на зайнятий і вільний блоки, і користувачу повертається вказівник на початок зайнятого блоку. Якщо потрібний блок не знайдений, то повертається NULL.

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

Якщо addr = NULL, то виконується виклик mem\_alloc(size). Інакше, відбувається перевірка сусідніх блоків. Якщо хоча б один з них вільний, відбувається об’єднання блоків. Далі перевіряється, чи цей новий блок достатнього розміру для перевизначення. Якщо так, то в ньому створюється 2 блоки: зайнятий і вільний. У зайнятий копіюються дані з колишнього блоку користувача. Якщо ж ні – йде пошук нового вільного блоку і, якщо він знайдений, виділяється блок для користувача і копіюються данні. Інакше – повертається NULL, а данні користувача не змінюютсья.

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**:

O(N), де N – кількість блоків у пам’яті.

**Оцінка часу звільнення занятого блоку**:

O(1).

**Оцінка витрати пам'яті для зберігання службовох інформації:**

12 \* N байт, де N – кількість блоків у пам’яті.

**Переваги аллокатора:**

1. Малий час виконання звільнення блоку.
2. Об’єднання сусідніх вільних блоків в один.

**Недоліки аллокатора:**

1. Великий час пошуку нового блоку( лінійна складність)

Лістинг програми:

#include <Windows.h>

#include <iostream>

#include "Allocator.h"

using namespace std;

Allocator::Allocator(const int n) {

int\* mas = new int[n + 1];

N = n;

bSize = sizeof(BlockHeader) / sizeof(int);

begin = (BlockHeader\*)(&mas[0]);

begin->prevsize = NULL;

begin->size = &mas[n] - &mas[0] - bSize;

begin->state = false;

endOfMemory = &mas[n];

}

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

BlockHeader\* current = begin;

while (current->size < size || current->state) {

if (isLast(current))

return NULL;

current = nextBlockHeader(current);

}

if ((current->size == size) || ((current->size - size) < bSize)) {

current->state = 1;

return getBlock(current);

}

else {

return separateOnUseAndFree(current, size);

}

}

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

if (addr == NULL) {

return mem\_alloc(size);

}

BlockHeader\* current = (BlockHeader\*)addr - 1;

size\_t deltaSize = current->size - size;

if (deltaSize == 0)

return addr;

BlockHeader\* previous = previousBlockHeader(current);

BlockHeader\* next = nextBlockHeader(current);

if (previous == NULL)

if (next == NULL) {

return searchNewBlock(addr, size);

}

else

if (next->state)

return searchNewBlock(addr, size);

else

return expandRight(addr, size);

else

if (previous->state)

if (next == NULL)

return searchNewBlock(addr, size);

else

if (next->state)

return searchNewBlock(addr, size);

else

return expandRight(addr, size);

else

if (next == NULL)

return expandLeft(addr, size);

else

if (next->state)

return expandLeft(addr, size);

else

return expandBoth(addr, size);

return NULL;

}

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

BlockHeader\* current = (BlockHeader\*)addr - 1;

BlockHeader\* next = nextBlockHeader(current);

BlockHeader\* previous = previousBlockHeader(current);

if (previous == NULL)

if (next == NULL)

current->state = false;

else

if (next->state)

current->state = false;

else

mergeWithNext(current, next);

else

if (previous->state)

if (next == NULL)

current->state = false;

else

if (next->state)

current->state = false;

else

mergeWithNext(current, next);

else

if (next == NULL)

mergeWithPrevious(previous, current);

else

if (next->state)

mergeWithPrevious(previous, current, next);

else

mergeBoth(previous, current, next);

}

BlockHeader\* Allocator::nextBlockHeader(BlockHeader\* current) {

if (isLast(current))

return NULL;

return (BlockHeader\*)((int\*)((char\*)current + sizeof(BlockHeader)) + current->size);

}

BlockHeader\* Allocator::previousBlockHeader(BlockHeader\* current) {

if (current->prevsize == NULL)

return NULL;

return (BlockHeader\*)((int\*)((char\*)current - sizeof(BlockHeader)) - current->prevsize);

}

bool Allocator::isLast(BlockHeader\* h) {

if (((int\*)((char\*)h + sizeof(BlockHeader)) + h->size) == endOfMemory)

return true;

else

return false;

}

void Allocator::mergeWithNext(BlockHeader\* current, BlockHeader\* next) {

current->size += next->size + sizeof(BlockHeader) / sizeof(int);

current->state = false;

BlockHeader\* next2 = nextBlockHeader(next);

if (next2 != NULL)

next2->prevsize = current->size;

}

void Allocator::mergeWithPrevious(BlockHeader\* previous, BlockHeader\* current) {

previous->size += current->size + sizeof(BlockHeader) / sizeof(int);

}

void Allocator::mergeWithPrevious(BlockHeader\* previous, BlockHeader\* current, BlockHeader\* next) {

previous->size += current->size + sizeof(BlockHeader) / sizeof(int);

next->prevsize = previous->size;

}

void Allocator::mergeBoth(BlockHeader\* previous, BlockHeader\* current, BlockHeader\* next) {

previous->size += current->size + next->size + 2 \* sizeof(BlockHeader) / sizeof(int);

BlockHeader\* next2 = nextBlockHeader(next);

if (next2 != NULL)

next2->prevsize = previous->size;

}

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

int\* f = (int\*)from;

int\* t = (int\*)to;

for (unsigned int i = 0; i < quantity; i++)

t[i] = f[i];

}

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

BlockHeader\* current = (BlockHeader\*)addr - 1;

size\_t deltaSize = current->size - size;

if ((deltaSize > 0) && (deltaSize < 3)) {

return addr;

}

if (deltaSize > 0) {

return separateOnUseAndFree((BlockHeader\*)addr - 1, size);

}

else {

void\* nBlock = mem\_alloc(size);

if (nBlock != NULL) {

copyData(addr, nBlock, current->size);

mem\_free(addr);

return nBlock;

}

return NULL;

}

}

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

BlockHeader\* current = (BlockHeader\*)addr - 1;

BlockHeader\* previous = previousBlockHeader(current);

size\_t area = current->size + current->prevsize + sizeof(BlockHeader) / sizeof(int);

size\_t deltaSize = current->size - size;

if (deltaSize > 0) {

int\* first = (int\*)addr;

for (int i = size - 1; i >= 0; i--) {

first[i + deltaSize] = first[i];

}

previous->size += deltaSize;

current = nextBlockHeader(previous);

initBlockHeader(current, true, previous->size, size, 7);

BlockHeader\* next = nextBlockHeader(current);

if (next != NULL)

next->prevsize = current->size;

return getBlock(current);

}

else {

if (area >= size)

if ((area == size) || ((area - size) < 3)) {

previous->size = area;

BlockHeader\* next = nextBlockHeader(current);

if (next != NULL)

next->prevsize = previous->size;

previous->state = true;

copyData(addr, getBlock(previous), current->size);

return getBlock(previous);

}

else {

previous->size = size;

previous->state = true;

copyData(addr, getBlock(previous), current->size);

current = nextBlockHeader(previous);

current->size = area - size - sizeof(BlockHeader) / sizeof(int);

current->state = false;

current->prevsize = previous->size;

BlockHeader\* next = nextBlockHeader(current);

if (next != NULL)

next->prevsize = current->size;

return getBlock(previous);

}

else {

void\* p = mem\_alloc(size);

if (p != NULL) {

copyData(addr, p, current->size);

BlockHeader\* next = nextBlockHeader(current);

if (next == NULL)

mergeWithPrevious(previous, current);

else

mergeWithPrevious(previous, current, next);

return p;

}

}

}

return NULL;

}

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

BlockHeader\* current = (BlockHeader\*)addr - 1;

BlockHeader\* next = nextBlockHeader(current);

size\_t area = current->size + next->size + sizeof(BlockHeader) / sizeof(int);

int deltaSize = (int)(current->size) - (int)(size);

if (deltaSize > 0) {

current->size = size;

next = nextBlockHeader(current);

initBlockHeader(next, false, current->size, area - size - sizeof(BlockHeader) / sizeof(int), 7);

BlockHeader\* next2 = nextBlockHeader(next);

if (next2 != NULL)

next2->prevsize = next->size;

return getBlock(current);

}

else {

if (area >= size)

if ((area - size) < 3) {

current->size = area;

next = nextBlockHeader(current);

if (next != NULL)

next->prevsize = current->size;

return getBlock(current);

}

else {

current->size = size;

next = nextBlockHeader(current);

next->size = area - size - sizeof(BlockHeader) / sizeof(int);

next->prevsize = size;

next->state = false;

BlockHeader\* next2 = nextBlockHeader(next);

if (next2 != NULL)

next2->prevsize = next->size;

return getBlock(current);

}

else {

void\* p = mem\_alloc(size);

if (p != NULL) {

copyData(addr, p, current->size);

next = nextBlockHeader(current);

mergeWithNext(current, next);

return p;

}

}

}

return NULL;

}

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

BlockHeader\* current = (BlockHeader\*)addr - 1;

BlockHeader\* previous = previousBlockHeader(current);

BlockHeader\* next = nextBlockHeader(current);

size\_t area = current->size + previous->size + next->size + 2 \* sizeof(BlockHeader) / sizeof(int);

int deltaSize = (int)(current->size) - (int)(size);

if (deltaSize > 0) {

mergeBoth(previous, current, next);

copyData(addr, getBlock(previous), size);

return separateOnUseAndFree(previous, size);

}

else {

if (area >= size)

if ((area == size) || ((area - size) < 3)) {

previous->size = area;

BlockHeader\* next2 = nextBlockHeader(next);

if (next2 != NULL)

next2->prevsize = previous->size;

previous->state = true;

copyData(addr, getBlock(previous), current->size);

return getBlock(previous);

}

else {

mergeBoth(previous, current, next);

copyData(addr, getBlock(previous), current->size);

return separateOnUseAndFree(previous, size);

}

else {

void\* p = mem\_alloc(size);

if (p != NULL) {

copyData(addr, p, current->size);

mergeBoth(previous, current, next);

return p;

}

}

}

return NULL;

}

void Allocator::initBlockHeader(BlockHeader\* bh, bool state, size\_t previous, size\_t size, int mask) {

if (bh == NULL)

return;

if (mask & 4) {

bh->state = state;

}

if (mask & 2) {

bh->prevsize = previous;

}

if (mask & 1)

bh->size = size;

}

void\* Allocator::separateOnUseAndFree(BlockHeader\* current, size\_t size) {

size\_t curSize = current->size;

BlockHeader\* next = nextBlockHeader(current);

if (next == NULL) {

initBlockHeader(current, true, NULL, size, 5);

BlockHeader\* next1 = nextBlockHeader(current);

initBlockHeader(next1, false, size, (curSize - size - bSize), 7);

return getBlock(current);

}

else {

initBlockHeader(current, true, NULL, size, 5);

BlockHeader\* next1 = nextBlockHeader(current);

initBlockHeader(next1, false, size, (curSize - size - bSize), 7);

next->prevsize = next1->size;

return getBlock(current);

}

}

void\* Allocator::getBlock(BlockHeader\* h) {

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

}

void Allocator::mem\_dump() {

BlockHeader\* current = begin;

int i = 0;

while (current != NULL) {

cout << i << ". " << current << endl;

i++;

current = nextBlockHeader(current);

}

}

**Результати виконання програми:**

