МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КПІ»

Кафедра

обчислювальної техніки

**ЛАБОРАТОРНА РОБОТА 1**

з дисципліни

“Операційні Системи”

на тему

"Аллокатор пам'яті загального призначення"

Студента 3 курсу

групи ІП-53

Сулима О.С.

Перевірив: Сімоненко А.В.

Київ 2017

**Завдання:**

Розробити аллокатор загального призначення, використовуючи за основу описаний вище

базовий варіант алгоритму, беручи до уваги наступні умови:

1. Області пам'яті можна виділяти будь-яким доступним способом.

2. Функції mem\_alloc (), mem\_realloc () і mem\_free () повинні відповідати

наведеним вище прототипам.

3. Адреси пам'яті, які повертаються функціями mem\_alloc () і mem\_realloc (),

повинні бути вирівняні на кордон в 4 байтах.

4. Спробувати зменшити час пошуку вільного блоку пам'яті і час

звільнення зайнятого блоку.

5. Спробувати зменшити фрагментацію пам'яті.

6. Написати функцію mem\_dump (), яка повинна виводити на консоль

стан областей пам'яті.

**Алгоритм відповідає даним в описі лабораторної роботи:**

Під час ініціалізації аллокатора запитується деяка область пам'яті у ОС. При запиті на виділення блоку пам'яті, в цій області пам'яті шукається вільний блок пам'яті потрібного розміру, позначається як зайнятий і його адреса повертається з додатком. При запиті на звільнення блоку пам'яті, цей блок позначається як вільний. аналогічно

реалізується функція зміни розміру блоку. Якщо виділеної області пам'яті недостатньо, то виділяється ще одна область пам'яті.

Кожен блок пам'яті складається з заголовка фіксованого розміру і даних. В

заголовку вказується розмір поточного блоку, розмір попереднього блоку і

ознака зайнятості блоку. Вибирається область пам'яті, яка містить хоча б

один вільний блок. Далі в знайденої області пам'яті скануються всі блоки

по порядку, і вибирається перший відповідний вільний блок. знайдений блок

розділяється на два, один позначається як вільний, а другий позначається як

зайнятий і адреса його даних повертається з додатком. Якщо в знайденої

області пам'яті не вдалося знайти вільний блок достатнього розміру, то

проводиться пошук в іншій області пам'яті. При звільненні блоку

виробляється склейка з лівим і правим вільними блоками.

**Оцінка часу для вільного блоку:**

Θ (n) де n поточну кількість блоків в аллокаторе так як пошук лінійний, отже в залежності від стану аллокатора дана оцінка буде сильно мінятися.

**Оцінка часу для зайнятого блоку:**

Θ (1) так як для звільнення блоку необхідно лише виставити прапор зайнятості а також провести необхідну дефрагментацію блоків (склейку даного з сусідніми по можливості)

**Оцінка для збереження системних даних(трата пам’яті):**

На 64 бітної системі необхідні службові заголовки повинні займати по 12 байт (3 int) і їх буде 2 (отже 24 байта) а так само по 12 байт на кожен новий виділений блок.

Переваги:

* простота
* легкість реалізації
* універсальність

Недоліки:

* неоптимальна робота для конкретних завдань
* відсутність оптимізацій
* великий обсяг службової інформації

**Приклад:**

alloc::Allocator allocator(300);

allocator.mem\_dump();

void \*mem1 = allocator.mem\_alloc(25);

allocator.mem\_dump();

void \*mem2 = allocator.mem\_alloc(25);

allocator.mem\_dump();

allocator.mem\_free(mem2);

allocator.mem\_dump();

void \*mem3 = allocator.mem\_alloc(50);

allocator.mem\_dump();

void \*mem1\_realloc = allocator.mem\_realloc(mem1, 50);

allocator.mem\_dump();

**Лістинг:**

==> main.cpp <==

#include <iostream>

#include <tuple>

#include <vector>

#include <random>

#include <climits>

#include <algorithm>

#include <functional>

#include "alloc.h"

#include "debug.h"

#include "auto\_test.h"

int main() {

autotest::test(100, 200, 10000);

std::cout << "Info chunk size (per each allocation): " << sizeof(alloc::Chunk) << std::endl;

std::cout << "Permanent service data (2 chunks): " << 2 \* sizeof(alloc::Chunk) << std::endl;

return 0;

}

==> alloc.cpp <==

#include <cstring>

#include <algorithm>

#include "alloc.h"

#include "debug.h"

using alloc::Chunk;

size\_t align4(size\_t size) {

return size % 4 ? size + (4 - size % 4) : size;

}

void \*assign\_chunk(void \*ptr, const Chunk &chunk) {

memcpy(ptr, &chunk, sizeof(Chunk));

return ptr;

}

Chunk \*move\_chunk(Chunk \*chunk, size\_t amount = sizeof(Chunk)) {

return (Chunk \*) ((void \*) chunk + amount);

}

Chunk \*get\_next\_chunk(Chunk \*chunk) {

return move\_chunk(chunk, chunk->next\_chunk + sizeof(Chunk));

}

Chunk \*get\_prev\_chunk(Chunk \*chunk) {

return move\_chunk(chunk, -chunk->prev\_chunk - sizeof(Chunk));

}

void merge\_right(Chunk \*left, Chunk \*right) {

left->next\_chunk += right->next\_chunk + sizeof(Chunk);

get\_next\_chunk(right)->prev\_chunk = left->next\_chunk;

if (!right->free) {

memmove(move\_chunk(left), move\_chunk(right), right->next\_chunk);

left->free = false;

}

}

void check\_split\_chunk(Chunk\* chunk, size\_t size) {

if (chunk->next\_chunk <= size + sizeof(Chunk)) {

return;

}

auto \*next\_chunk = get\_next\_chunk(chunk);

size\_t splitted\_chunk\_size = chunk->next\_chunk - size - sizeof(Chunk);

chunk->next\_chunk = size;

auto \*splitted\_chunk = get\_next\_chunk(chunk);

assign\_chunk(splitted\_chunk, {splitted\_chunk\_size, size, true});

next\_chunk->prev\_chunk = splitted\_chunk\_size;

if (next\_chunk->free) merge\_right(splitted\_chunk, next\_chunk);

}

alloc::Allocator::Allocator(size\_t size) {

size = align4(size);

void\* mem = new char[size];

size\_t main\_chunk = size - 3 \* sizeof(Chunk);

assign\_chunk(mem, {0, 0, false});

assign\_chunk(mem + size - sizeof(Chunk), {0, main\_chunk, false});

assign\_chunk(mem + sizeof(Chunk), {main\_chunk, 0, true});

pool = mem + sizeof(Chunk);

}

alloc::Allocator::~Allocator() {

delete[] (pool - sizeof(Chunk));

}

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

size = align4(size);

auto \*chunk = (Chunk \*) pool;

while (chunk->next\_chunk > 0) {

if (chunk->free && chunk->next\_chunk >= size) {

check\_split\_chunk(chunk, size);

chunk->free = false;

return move\_chunk(chunk);

}

chunk = get\_next\_chunk(chunk);

}

return nullptr;

}

void alloc::Allocator::mem\_free(void \*ptr) {

auto \*chunk = (Chunk \*) (ptr - sizeof(Chunk));

chunk->free = true;

auto \*prev\_chunk = get\_prev\_chunk(chunk);

auto \*next\_chunk = get\_next\_chunk(chunk);

if (next\_chunk->free) merge\_right(chunk, next\_chunk);

if (prev\_chunk->free) merge\_right(prev\_chunk, chunk);

}

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

size = align4(size);

auto check\_merge = [size](Chunk \*adjacent, Chunk \*main) {

return adjacent->free

&& adjacent->next\_chunk + main->next\_chunk >= size + sizeof(Chunk);

};

if (!ptr) return mem\_alloc(size);

auto \*chunk = (Chunk \*) (ptr - sizeof(Chunk));

auto \*prev\_chunk = get\_prev\_chunk(chunk);

auto \*next\_chunk = get\_next\_chunk(chunk);

if (chunk->next\_chunk > size + sizeof(Chunk)) {

check\_split\_chunk(chunk, size);

return ptr;

} else if (check\_merge(prev\_chunk, chunk)) {

merge\_right(prev\_chunk, chunk);

check\_split\_chunk(prev\_chunk, size);

return move\_chunk(prev\_chunk);

} else if (check\_merge(next\_chunk, chunk)) {

merge\_right(chunk, next\_chunk);

check\_split\_chunk(chunk, size);

return move\_chunk(chunk);

} else {

auto \*new\_ptr = mem\_alloc(size);

if (new\_ptr) {

memcpy(new\_ptr, ptr, std::min(chunk->next\_chunk, size));

mem\_free(ptr);

return new\_ptr;

}

}

return nullptr;

}

void alloc::Allocator::mem\_dump() {

print\_chunks(pool);

}

==> alloc.h <==

#ifndef LAB1\_ALLOCATOR\_ALLOC\_H

#define LAB1\_ALLOCATOR\_ALLOC\_H

#include <cstddef>

namespace alloc {

struct Chunk {

size\_t next\_chunk;

size\_t prev\_chunk;

bool free;

Chunk(size\_t next\_chunk, size\_t prev\_chunk, bool free)

: next\_chunk(next\_chunk), prev\_chunk(prev\_chunk), free(free) {}

};

class Allocator {

void\* pool;

public:

Allocator(size\_t size);

~Allocator();

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

void mem\_free(void \*ptr);

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

void mem\_dump();

};

}

#endif //LAB1\_ALLOCATOR\_ALLOC\_H

==> debug.cpp <==

#include <iostream>

#include "debug.h"

#include "alloc.h"

using alloc::Chunk;

void print\_chunk(const Chunk &chunk) {

std::cout << "{ "

<< chunk.next\_chunk

<< ", " << chunk.prev\_chunk

<< ", " << chunk.free

<< " }\n";

}

void print\_chunk(void \*ptr) {

print\_chunk(\*(Chunk \*) (ptr - sizeof(Chunk)));

}

void print\_chunks(void \*pool) {

auto \*chunk = (Chunk \*) pool;

print\_chunk(\*(Chunk \*) (pool - sizeof(Chunk)));

while (chunk->next\_chunk) {

print\_chunk(\*chunk);

chunk = (Chunk \*) (((void \*) chunk) + chunk->next\_chunk + sizeof(Chunk));

}

print\_chunk(\*chunk);

std::cout << std::endl;

}

==> debug.h <==

#ifndef LAB1\_ALLOCATOR\_DEBUG\_H

#define LAB1\_ALLOCATOR\_DEBUG\_H

#include "alloc.h"

void print\_chunk(const alloc::Chunk &chunk);

void print\_chunk(void \*ptr);

void print\_chunks(void \*pool);

#endif //LAB1\_ALLOCATOR\_DEBUG\_H

==> auto\_test.cpp <==

#include <cstddef>

#include <climits>

#include <random>

#include <iostream>

#include "auto\_test.h"

#include "debug.h"

#include "alloc.h"

using namespace autotest;

struct Test {

void \*ptr = nullptr;

int size = 0;

long hash = 0;

};

long autotest::hash(void \*curr, size\_t size) {

void \*end = curr + size;

long hash = 0;

while (curr + sizeof(long) < end) {

hash ^= \*((long \*) curr);

curr += sizeof(long);

}

if (curr + sizeof(long) != end) {

long last = 0;

while (curr < end) {

last <<= sizeof(char);

last |= \*((char \*) curr);

curr++;

}

hash ^= last;

}

return hash;

}

void autotest::fill\_random(void \*ptr, size\_t size) {

std::independent\_bits\_engine<std::default\_random\_engine, CHAR\_BIT, unsigned char> engine{};

auto \*data = (unsigned char \*) ptr;

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

\*data++ = engine();

}

}

void update\_test(Test &test, int size, void \*ptr) {

test.ptr = ptr;

test.size = size;

fill\_random(test.ptr, test.size);

test.hash = hash(test.ptr, test.size);

}

void autotest::test(int elements, int rounds, int pool\_size) {

std::vector<Test> test\_data{elements, Test()};

alloc::Allocator allocator{pool\_size};

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> coin\_dis(0, 1);

std::uniform\_int\_distribution<> test\_dis(0, elements - 1);

std::uniform\_int\_distribution<> size\_dis(2, pool\_size / 10);

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

Test &test = test\_data[test\_dis(gen)];

if (!test.ptr) {

int size = size\_dis(gen);

void\* ptr = allocator.mem\_alloc(size);

if (ptr) update\_test(test, size, ptr);

} else {

if (hash(test.ptr, test.size) != test.hash) {

std::cout << "Hash mismatch for:" << std::endl;

print\_chunk(test.ptr);

std::cout << std::endl;

std::cout << std::endl << "All chunks:" << std::endl;

allocator.mem\_dump();

return;

}

if (coin\_dis(gen)) {

int size = size\_dis(gen);

void \*ptr = allocator.mem\_realloc(test.ptr, size);

if (ptr) {

if (size > test.size && hash(ptr, test.size) != test.hash) {

std::cout << "Realloc integrity failed" << std::endl;

std::cout << "Realloc (" << size << "):";

print\_chunk(ptr);

std::cout << std::endl;

return;

}

update\_test(test, size, ptr);

}

} else if (test.ptr) {

allocator.mem\_free(test.ptr);

test.ptr = nullptr;

}

}

}

std::cout << std::endl << "All chunks at the end of test:" << std::endl;

allocator.mem\_dump();

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

Test &test = test\_data[i];

if (test.ptr) allocator.mem\_free(test.ptr);

}

std::cout << std::endl << "All chunks after cleanup:" << std::endl;

allocator.mem\_dump();

}

==> auto\_test.h <==

#ifndef LAB1\_ALLOCATOR\_AUTO\_TEST\_H

#define LAB1\_ALLOCATOR\_AUTO\_TEST\_H

namespace autotest {

void test(int elements, int rounds, int pool\_size);

long hash(void \*curr, size\_t size);

void fill\_random(void \*ptr, size\_t size);

}

#endif //LAB1\_ALLOCATOR\_AUTO\_TEST\_H