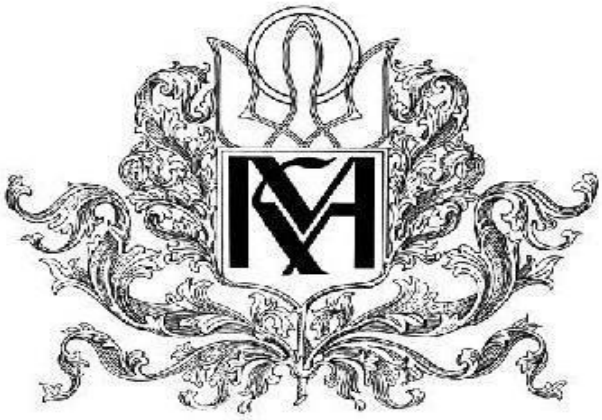
Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики



Створення засобів ефективного управління пам'яттю

Текстова частина до курсової роботи

за спеціальністю „Інженерія програмного забезпечення” 121

Керівник курсової роботидоцент, Бублик В.В.

_____

(підпис)

“_____”_____2022 р.

Виконала студентка Шляхова О. Д.

“_____”_____2022 р.

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри мультимедійних систем,

доцент, к.ф-м.н.

_____О. П. Жежерун (підпис)

„____”_____2022 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Шляховій Олександрі Дмитрівні факультету інформатики 3-го курсу

ТЕМА Створення засобів ефективного управління пам'яттю

Зміст ТЧ до курсової роботи:

Дата видачі „___” _____ 2021 р. Керівник _____ (підпис)

Завдання отримав _____ (підпис)

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ РОБОТИ

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання теми курсової роботи	14.10.2021	
2.	Пошук тематичної літератури	23.10.2021	
3.	Ознайомлення з науковою літературою	01.03.2022	
4.	Визначення структури роботи	01.04.2022	
6.	Написання першої ітерації роботи	15.05.2022	
7.	Доповнення, розширення та редагування стилю роботи	20.05.2022	
8.	Написання висновків	23.05.2022	
9.	Перегляд змісту роботи керівником	03.06.2022	
10.	Корегування роботи згідно із зауваженнями керівника	03.06.2022	
11.	Подача курсової роботи на перевірку	07.06.2022	

Contents

Вступ

> *C++'s has ability to handle hardware well and to provide efficient high-level abstractions.*

У цій роботі мова піде про одну з ability to handle hardware, а саме про способи алокації пам'яті в C++.

Стандартні способи виділення та вивільнення пам’яті

Працюючи на більш високому рівні абстракції, ми використовуємо оператори new та delete для виділення (звільнення) пам’яті та створення (видалення) об’єктів.

Оператор new

```
T* i = new T
```

Це найбільш вживана форма оператора ***new***. Вона виконує декілька функцій. По-перше, виділяє пам’ять для розміщення об’єкта типу T. По-друге, викликає конструктор T, що призводить до створення нового об’єкта в комірці пам’яті, що була повернута на попередньому кроці. Якщо T є примітивним типом(int, float тощо), конструктор викликатись не буде.

Для того, щоб виділити пам’ять достатню для розміщення T, можна використати інше перевантаження оператора new:

```
void* operator new(size_t bytes);
```

А для конструювання об’єкта у новій ділянці пам’яті можна використати третє перевантаження оператора new:

```
T* i = new (memoryAddress) T;
```

Оператор delete

Виклик оператора delete призводить до виклику деструктора об’єкта і подальшого вивільнення пам’яті, яку займав об’єкт.

```
delete i;
```

Також оператор delete можна викликати наступним чином:

```
operator delete(mem);
```

Але у такому разі користувач має викликати деструктор об’єкта самостійно.

Оператори new[] та delete[]

Оператор new[] виділяє пам’ять для розміщення n елементів масиву і викликає конструктор для кожного об’єкта(якщо ж відбувається створення примітивних типів, деструктор викликано не буде). Але насправді пам’ять виділяється не лише для зберігання елементів масиву. Додаткове місце виділяється для зберігання розміру масиву. Тобто наступний код

```
int* i = new int [3];
```

виділить не sizeof(int) * 3, а sizeof(int) * 3 + sizeof(size_t).

На початку виділеної пам’яті буде записаний розмір масиву(3 у цьому випадку), а далі будуть послідовно розміщуватись елементи самого масиву. Користувачу ж повернеться указник ptr + sizeof(size_t).

Така поведінка оператора new[] потрібна для коректного видалення масиву. Під час виклику оператора delete[] користувач передає лише указник на масив, який потрібно видалити. Визначення розміру масиву покладається на оператор delete[]. Тож при видаленні масиву за указником, спочатку будуть прочитані sizeof(size_t) байт перед указником для визначення розміру, потім буде викликана відповідна кількість деструкторів і пам’ять звільниться.

Розуміючи роботу описаних вище операторів, стає зрозумілим, чому потрібно викликати new лише у парі з delete, а new[] - з delete[]. Якщо об’єкт було створено за допомогою new, а видалено за допомогою delete[], подальша поведінка програми стає непередбачуваною. delete[] інтерпретує перші sizeof(size_t) байт як розмір масиву і спробує очистити таку

ділянку пам’яті. Якщо ж викликати delete для об’єктів, створених за допомогою new[], відбудеться видалення лише першого елементу масиву, що у своєю чергою призведе до витоку пам’яті.

Базове визначення алокатору

Тепер, коли стало зрозуміло, як використовувати new та delete, постає наступне логічне питання: «Як саме ці два оператори роблять свою роботу?». Вище, описуючи роботу оператора new, мимохідь було сказано «виділяє пам'ять, достатню для розміщення об’єкта». Але як саме відбувається процес виділення пам’яті? Що ж, точний алгоритм залежить від операційної системи, але, якщо розглядати цей процес вищому рівні абстракції, в купі потрібно знайти вільну ділянку пам’яті достатнього розміру, помітити її як зайняту і повернути вказівник на адресу користувачу. Найскладнішим етапом у цьому алгоритмі є пошук вільної ділянки достатнього розміру[1].

Цим процесом займаються **алокатори**. Ці сутності отримують у своє розпорядження певний блок вільної пам’яті. Під час запиту на виділення пам’яті алокатор шукає вільну ділянку потрібного розміру у своєму буфері[2].

Важливо зауважити, що алокатори не можуть переміщувати дані з одного місця в інше(з однієї адреси на іншу) для досягнення послідовного розміщення даних. Тому, якщо певний блок було виділено користувачу, він звільниться тільки за запитом користувача і ніяк інакше.

Фрагментація

[1] Насправді у більшості випадків задача алокатора не обмежується пошуком достатньо великого блоку. Знайдений блок має відповідати додатковим критеріям в залежності від обраної стратегії виділення пам’яті. Ця тема буде розглянута більш детально нижче.

[2] Алокатори можуть різнитись за своєю поведінкою у випадку вичерпання буфера. Усім відомий std::bad_alloc це один із варіантів реакції на вичерпання пам’яті – відмова робити наступні алокації. Існує у другий варіант – запит у системи на розширення буфера.

Оскільки користувач звільнює пам’ять у непередбачуваному для алокатора порядку, у буфері з часом з’являються «дірки», які можуть бути заповнені об’єктами обмеженого розміру. Процес утворення таких «дірок» називається фрагментацією. Також до фрагментації відносять ситуацію, коли два вільні блоки розділені невеликою ділянкою зайнятої пам’яті. У такому випадку сумарний розмір вільних ділянок не може бути використаний для запиту на пам’ять, оскільки блоки не є послідовними.

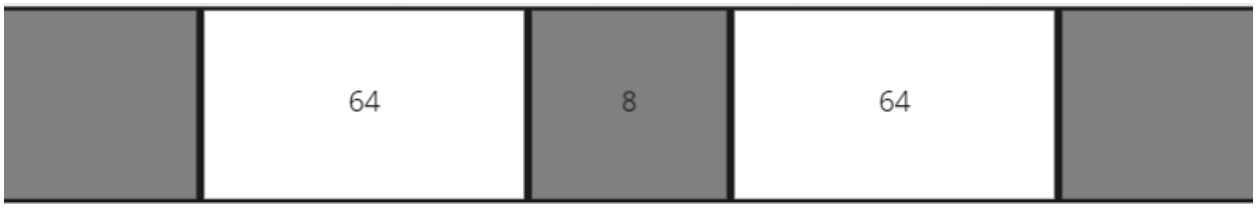


Рисунок 1 Два вільних блоки у 64 байти розділені блоком у 8 байт. Незважаючи на те, що буфер має 128 байт вільної пам'яті, максимальний розмір блоку, який може бути виділений - 64 байти

Узагальнюючи, можна сказати, що фрагментація шкодить програмі тоді, коли кожен вільний блок пам’яті є замалим для того, щоб задовольнити запит користувача, але сумарний розмір вільної пам’яті є достатнім.

Види фрагментації

Фрагментацію поділяють на два види: **внутрішню** і **зовнішню**. Зовнішня фрагментація була описана вище(програма має блоки замалі для розміщення великого об’єкта). Що стосується внутрішньої фрагментації, вона виникає тоді, коли запит користувача задовольняється блоком, розмір якого трохи більше ніж потрібно. Наприклад, для алокації 12 байтів використовується блок у 16 байтів. Зайвих 4 байти є занадто малими для подальшого збереження. Тому такі «шматочки» марно витрачаються, викликаючи внутрішню фрагментацію.

Дослідження поведінки програми

На перший погляд здається, що уникнути фрагментації неможливо. Але Детальний аналіз поведінки програми може показати у реальних застосунках запити на пам’ять не є хаотичними. особливості таких запитів.

У такому дослідженні варто звернути увагу на розміри блоків, які виділяються найчастіше, тривалість життя блоків різного розміру, частоту зміни розмірів алокації (запити на пам’ять одного розміру є послідовними чи за виділенням малої ділянки пам’яті часто слідує запит на великий блок) тощо. Зібравши статистку такого роду, можна сконструювати досить ефективний алокатор. Але варто розуміти, що алокатор, побудований з урахуванням статистики про роботу певної програми, буде дуже специфічним і майже ніколи не задовольнить потреби іншої програми.

Варто зауважити, що поведінка однієї програми може змінюватись час від часу. Наприклад, на початку роботи програма створює багато невеликих об’єктів з коротким життєвим циклом, а в середині своєї роботи робить багато запитів на великі об’єкти, які «живуть» до кінця роботи програми. У такому випадку алокатор, який буде мати хороші показники на першому етапі, може показувати погані результати під час другої фази роботи.

Алокатори загального призначення

Прямою протилежністю алокаторів, побудованих на основі статистики про роботу програми(алокаторів спеціального призначення), є алокатори загального призначення, які сконструйовано таким чином, щоб задовольнити більшість програм.

Алокатор Дугласа Лі

Прикладом алокатора загального призначення є загальновідомий алокатор Дугласа Лі або, як його ще називають, dlmalloc. Він досить гнучко налаштовується за допомогою директив препроцесора. Наприклад, користувач може змінити вирівнювання, увімкнути використання футерів, задати поведінку програми після виявлення порушень у доступі до пам'яті тощо. Повний список усіх налаштувань можна переглянути у коментарях до вихідного коду алокатора. Dlmalloc використовується оператором new за замовчуванням в деяких версіях Linux. Але цей алокатор призначений для використання в однопотокових застосунках[3].

Алокатор Вольфрама Глогера

Для багатопотокових систем можна використати ptmalloc. Це розширена версія алокатора Дугласа Лі, яка була написана Вольфрамом Глогером у 1999 році. Цей алокатор набагато краще пристосований для використання в багатопотоковому середовищі.

> *Велика різниця між ptmalloc і dlmalloc полягає в підтримці багатопоточних виділень пам'яті. ptmalloc підтримує кілька* ***арен****. Кожна арена являє собою незалежний екземпляр алокатора, захищеного одним замком. Коли потік робить запит на виділення пам'яті, він шукає незаблоковану арену,*

[3] dlmalloc можна налаштувати для роботи у багатопоточному середовищі визначивши директиву препроцесора USE_LOCKS, але, за словами автора, це буде не дуже ефективним

починаючи з тієї, що він використовував останньою. Якщо потоку вдалося заблокувати арену, алокація відбувається на ній, в іншому разі відбувається алокація нової арени.

Щоб знайти арену за указником на блок, використовується проста арифметика вказівників. Усі арени, крім однієї, починаються з адрес, кратних максимальному розміру арени, за замовчуванням один мегабайт. Таким чином, маскування бітів указівника блоку, що адресують в арені, повертає початковий указник арени. Одна арена призначена як ***головна*** *і не підлягає обмеженням у розмірах. Головні арени визначаються бітовими заголовками.*

(1)

Інші алокатори загального призначення

Серед інших відомих алокаторів загального призначення можна згадати про tcmalloc, який використовує компанія Google, а також Hoard розроблений для багатопотокових програм.

Механізми виділення пам’яті

Алокатори розрізняють за механізмами, які вони використовують для виділення і звільнення пам’яті. Тож у цій частині будуть розглянуті популярні механізми виділення пам’яті.

Нульовий алокатор (NullAllocator)

Найпростіший алокатор - це алокатор, який нічого не робить. Він кидає помилку при будь-якій спробі виділення пам’яті. Такий простий алокатор насправді є дуже важливим. Його найголовніша цінність полягає у тому, що він є дуже зручним структурним елементом. Під час конструювання складніших алокаторів(які є композитом) такому простому структурному елементу можна делегувати роботу у випадку нестачі пам’яті у буфері іншого алокатору.

Лінійний алокатор

Наступний тип алокатора – **лінійний**. Він ініціалізується буфером певного розміру і розташовує об’єкти послідовно один за одним(між об’єктами можуть бути лише проміжки, пов’язані з вирівнюванням). Такий алокатор не реалізовує можливості вивільнення окремих ділянок пам’яті з буфера. Натомість він дає можливість вивільнити увесь буфер повністю. Такий алокатор є дуже специфічним, але дуже швидким. Якщо в програмі є певна кількість об’єктів, які мають однаковий життєвий цикл(«помирають» одночасно), такий алокатор буде дуже зручним. Вивільнення відбудеться моментально. Також у таких випадках часто можна вивільнити пам’ять навіть без виклику деструкторів об’єктів. Якщо це можливо, це точно потрібно зробити. Лінійний алокатор має ще одну перевагу за рахунок того, що вивільнення відбувається лише для усього буфера. Річ у тому, що більшість алокаторів зберігають додаткову інформацію перед виділеною ділянкою як мінімум

для збереження довжини зайнятої пам’яті. Довжина потрібна для реалізації оператора вивільнення оскільки він приймає лише указник як параметр, розмір має зберігатись в іншому місці. Тож розмір кожної алокації збільшується принаймні на одне машинне слово. Наприклад, в алокаторі Дугласа Лі такі накладні витрати:

> *Мінімальні накладні витрати на виділений фрагмент: 4 або 8 байт (якщо розміри машинного слова 4 байти) 8 або 16 байт (якщо розміри машинного слова 8 байт)*
>
> *Кожний алокований фрагмент доповнюється прихованим машинним словом, в якому зберігається інформація про стан. За умови визначення макросу FOOTER, ще одне слово виділяється для додаткової перевірки.*

(2)

> *Заголовок і футер, ймовірно, значно вплинуть на розміри алокації— із середнім розміром об’єкта близько десяти слів, наприклад, заголовок із одного слова тягне за собою 10% накладних витрат, а футер із одним словом несе ще 10%.*

(3)

У лінійному алокаторі це не потрібно, тож можна розмістити більшу кількість об’єктів. Ще одне важливе зауваження, при такому способі роботи з пам’яттю фрагментація повністю відсутня.

Також варто звернути увагу на те, що об’єкти розташовуються близько один до одного у пам’яті. Це дозволяє використовувати кеш, що значно пришвидшує зчитування інформації.

Підсумовуючи, лінійний алокатор є дуже швидким(операції виділення пам’яті відбуваються за константний час), він розташовує об’єкти близько один до одного, а також не

використовує додаткової пам’яті для кожної алокації, але лінійний алокатор не підтримує вивільнення окремих об’єктів, що значно зменшує область його застосування.

Алокатор-стек

Лінійний алокатор може бути доповнений можливістю видалення ділянки, що була зайнята останньою. Такий тип алокатора називається **стек**. Він має майже усі плюси та мінуси лінійного алокатору, окрім економії пам’яті. Дозволяючи користувачу вивільняти деякі ділянки пам’яті, алокатор накладає на себе зобов’язання зберігати розмір кожного зайнятого блоку.

Приклад реалізації стекового алокатора:

```
template <int aligment>
class StackAllocator final
{
public:
    StackAllocator(size_t bufferSize);
    ~StackAllocator();

    void* allocate(size_t size);
    void free(void*);
    void reset();
private:
    //where the buffer initially starts
    char* _bufferStart; //char* is for convenience of pointer addition
    //where the buffer initially ends
    char* _bufferEnd;
    //pointer to first free byte in buffer
    char* _currPosition;
};

template <int aligment>
StackAllocator<aligment>::StackAllocator(size_t bufferSize)
    :_bufferStart(reinterpret_cast<char*>(malloc(bufferSize))), _bufferEnd(_bufferStart
+ bufferSize),
    _currPosition(_bufferStart)
{

}

template <int aligment>
StackAllocator<aligment>::~StackAllocator()
{
    ::free(_bufferStart);
}

template <int aligment>
void StackAllocator<aligment>::free(void* ptrToFree)
{
    if (ptrToFree == nullptr)
    {
        return;
    }
    union
    {
        void* asVoid;
        size_t* asSize;
        char* asChar;
    };
    asVoid = ptrToFree;
    asSize -= 1;
    const bool isTheLastAllocation = asChar + sizeof(size_t) + *asSize ==
_currPosition;
    //only last allocated chunk can be freed
    if (isTheLastAllocation)
    {
        _currPosition = asChar;
    }
    else
    {
        assert(false && "Deallocation is done not in LIFO fashion");
    }
}
```

```cpp
template <int aligment>
void* StackAllocator<aligment>::allocate(size_t size)
{
    const size_t allignedSize = Utils::getAligned(size, aligment);
    /*
    every allocation has overhead of aligned sizeof(size_t) to support free operation
    free has only pointer as a parameter, size has to be deducted from elsewhere(in our case from allocation header )
    */
    static constexpr size_t headerSize = sizeof(size_t);
    const size_t allocationSize = allignedSize + headerSize;
    if (_currPosition + allocationSize > _bufferEnd)
    {
        //buffer is exhausted
        return nullptr;
    }

    union
    {
        void* asVoid;
        size_t* asSize;
        char* asChar;
    };
    asVoid = _currPosition;
    *asSize = allignedSize;//saving
    _currPosition = asChar + allocationSize;
    asChar += headerSize;//returning address after header to user(he doesn`t need to know about header existence)
    return asVoid;
}

template <int aligment>
void StackAllocator<aligment>::reset()
{
    _currPosition = _bufferStart;
}
```

Стек, своєю чергою, можна розширити можливістю виділяти пам’ять з обох кінців. Такий двосторонній стек також буде слідувати принципу LIFO(останнім прийшов — першим пішов), але користувач зможе використовувати обидва кінці стека. Такий варіант може бути корисним, якщо під час формування стеку потрібно зробити тимчасові алокації для створення «довгоживучого» об’єкта.

Наприклад, для створення об’єкта A потрібно зчитати вміст JSON файлу. Після створення об’єкта A вміст JSON файлу більше не потрібен. У класичному варіанті реалізації стека неможливо зробити щось подібне. JSON файл потрібно або розміщувати у додатковому буфері, або залишати до видалення об’єкта A (або до звільнення стека). Двосторонній стек дозволяє розмістити додатковий ресурс з одного кінця буфера, а залежний об’єкт – з іншого.

Список(Freelist)

Багато класичних алгоритмів алокації зберігають один зв’язний список усіх вільних ділянок пам’яті[4]. Кожен вільний блок починається зі стандартного заголовка. У ньому зберігаються адреси попереднього і наступного вільного блоку, а також розмір вільної ділянки. Окрім того, в заголовку може зберігатись додаткова інформація для відстеження помилок у використанні пам’яті. Наприклад, якщо оператор new повернув указник на ділянку розміром у 50 байтів, а користувач записав 60 байтів у цю ділянку. При будь-яких розкладах така ситуація є катастрофічною.

Якщо наступна ділянка була зайнятою, зайві 10 байтів перезапишуть збережену там інформацію. Окрім того, що

[4] Такий список зазвичай є двозв’язним та циклічним

інформація, що знаходиться за цією адресою стане не валідною. Ще більша катастрофа відбудеться при спробі видалити такий об’єкт. Як відомо, перед кожним зайнятим блоком зберігається інформація про його розмір. Якщо на це місце записати нову інформацію, видалення звільнить ділянку невідомого розміру.

Якщо ж наступна ділянка була вільною, заголовок блоку також перепишеться, що зламає структуру зв’язного списку.

Для боротьби з такою поведінкою у заголовки вузлів списку додають магічне число. Таким чином, під час обходу списку, алокатор перевірить чи наявне правильне магічне число. Якщо це число перезаписане, можна зробити висновок про некоректність роботи програми й відреагувати відповідним чином.

Таким чином, розуміючи що зберігається у заголовках вільних блоків, можна перейти до того, як відбувається пошук вільної ділянки пам’яті. Зазвичай відбувається проста ітерація по списку, доки не буде знайдений блок підхожого розміру. Такий блок вилучається зі списку, а його адреса повертається користувачу.

Такий простий алгоритм може мати погані показники часової складності для буферів великого розміру. Для більшості програм складність O(N) є неприпустимою. Тому для зберігання вільних блоків в реальних алокаторах зазвичай використовують більш ефективні структури(наприклад, дерево).

Абстрагуючись від проблеми зберігання вільних блоків, звернімо увагу на пошук вільного блока. Для розв'язання цієї задачі є декілька підходів.

Найвдаліше влучання(Best Fit)

Найвдаліше влучання шукає найменший вільний блок, що є достатньо великим, щоб задовольнити потребу користувача. Таке правило пошуку мінімізує об’єми змарнованого простору. Але така стратегія може показувати погані результати, якщо знайдені блоки є зовсім трохи більшими за потрібний розмір. У такому разі у пам’яті залишиться багато замалих вільних ділянок, що навряд можна буде використати для нових запитів. Але зрозуміло, що такий алгоритм недоцільно використовувати, якщо вільні блоки зберігаються у вигляді зв’язного списку.

Перше влучання(First Fit)

Перше влучання шукає з самого початку списку доки не знайде перший блок, що буде достатньо великим. Якщо знайдений блок виявився завеликим, він буде розділений і залишок повернеться у список вільних блоків.

Такий алгоритм може викликати проблеми, оскільки він породжує велику кількість маленьких блоків на початку списку. Ці невеличкі блоки збільшують час пошуку вільної ділянки для середніх і великих об’єктів.

Також перше влучання може викликати внутрішню фрагментацію, коли розмір знайденого блоку перевищує потрібний розмір лише на декілька машинних слів. Такі «залишки» не можуть бути повернені у зв’язний список, оскільки їх розмір недостатній навіть для зберігання заголовка.

Але на цей алгоритм можна дивитись зовсім з іншої точки зору, якщо замінити зв’язний список на складнішу структуру. Тоді постає логічне питання: який порядок задати вільним блокам, щоб покращити роботу цього алгоритму.

Найпростіший варіант – ставити нещодавно звільнені блоки на початок списку. У такий спосіб ми отримаємо повторне використання пам’яті. Загалом така поведінка описується правилом LIFO(останнім прийшов — першим пішов).

Прямою протилежністю попереднього варіанту упорядкування блоків є алгоритм, при якому звільнені блоки записуються у кінець зв’язного списку. Таким чином блоки будуть опрацьовуватись за принципом FIFO(першим прийшов, перший вийшов). Очевидно, що тут ми отримуємо пряму протилежність повторного використання. Користувачу постійно буде повертатись «стара» пам’ять, якою довго ніхто не користувався.

Інший варіант впорядкування вільних блоків – сортування за адресою. Це, очевидно, сповільнює операцію вивільнення пам’яті, оскільки ділянка має зайняти правильне місце у списку. Але у такого підходу є декілька важливих переваг. По-перше, об’єкти розташовуються близько один до одного, що дозволяє використовувати кеш для пришвидшення роботи з даними. По-друге, той факт, що усі блоки відсортовані за адресою, дає можливість скоротити заголовок блоків. Тепер інформація про розмір блоку є достатньою для знаходження наступного вільного блоку. Тож немає потреби зберігати адресу наступного вільного блоку.

Наступне влучання(Next Fit)

Наступне влучання можна розглядати як оптимізацію першого влучання. Замість того, щоб для кожної алокації повертати указник на початок списку, зберігається позиція останньої алокації й пошук нової ділянки відбувається з того самого місця. Такий невинний, на перший погляд, алгоритм може призвести до посилення фрагментації.

Річ у тому, що об’єкти, які алокуються послідовно, можуть мати однакову або схожу тривалість життя. Виходячи з такої

логіки, раціонально було б розміщувати об’єкти послідовно запитам на їх алокацію і сподіватись на те, що вони будуть «жити» однаково довго. У такому випадку, під час звільнення пам’яті обома об’єктами, звільнена пам’ять може бути об’єднана в один великий блок.

Якщо стратегія першого влучання розташовує об’єкти так близько один до одного у порядку запиту на алокацію, то стратегія наступного влучання буде алокувати об’єкти в різних частинах купи.

Приклад реалізації FreeList:

```
struct FLNode
{
    FLNode* _next;
};


class FreeListFixedSize
{
public:
    FreeListFixedSize(size_t chunkSize, size_t numOfElements)
        :_bufferSize(numOfElements* chunkSize), _bufferStart(malloc(numOfElements *
chunkSize)), _firstFreeNode(nullptr)
#ifdef DEBUG
        , _chunkSize(chunkSize)
#endif // DEBUG

    {
        assert(numOfElements > 0); // if number of elements is 0 malloced memory leaks
        union // as it is useful only in context of this function made inner
        {
            char* asChar;
            FLNode* asNode;
        };

        asChar = static_cast<char*>(_bufferStart); //initializing with the start of the
buffer
        FLNode* currentChunk = _firstFreeNode = asNode;
        for (int i = 0; i < numOfElements; ++i)
        {
            currentChunk->_next = asNode;
            currentChunk = currentChunk->_next;
            asChar += chunkSize;//shifting to next free chunk
        }
        currentChunk->_next = nullptr; //last has no next, points to nullptr
    }
    ~FreeListFixedSize()
    {
        ::free(_bufferStart);
    }

    //allocators can not be copied
    FreeListFixedSize(const FreeListFixedSize& other) = delete;
    FreeListFixedSize& operator=(const FreeListFixedSize& other) = delete;

    //but can be moved
    FreeListFixedSize(FreeListFixedSize&& other)
        :_bufferSize(other._bufferSize), _bufferStart(other._bufferStart),
_firstFreeNode(other._firstFreeNode)
    {
        other._bufferStart = nullptr;
        other._firstFreeNode = nullptr;
    }
    FreeListFixedSize& operator=(FreeListFixedSize&& other)
    {
        _bufferSize = other._bufferSize;
        _bufferStart = other._bufferStart;
        _firstFreeNode = other._firstFreeNode;

        other._bufferStart = nullptr;
        other._firstFreeNode = nullptr;
    }
```

```
   //as simple as returning next pointer to user, time complexity is O(1)(really fast!)
    void* allocate(size_t size)//size is added only for interface consistency
    {
#ifdef DEBUG
        assert(size == _chunkSize);
#endif // DEBUG
        if (_firstFreeNode == nullptr)
        {
            //buffer is exhausted
            return nullptr;
        }
        FLNode* freeChunk = _firstFreeNode; //extracting free chunk for user
        /*
          as next chunk will be returned to user(and it is not free anymore)
          redirecting next pointer to chunk after the one we will give to user to
support linked list structure
        */
        _firstFreeNode = freeChunk->_next;
        return freeChunk;
    }

    //freeing is also really fast
    void free(void* ptr)
    {
        FLNode* newFreeBlock = static_cast<FLNode*>(ptr);
        newFreeBlock->_next = _firstFreeNode;//inserting new block just after the start
        _firstFreeNode = newFreeBlock;
    }

    //checks if allocation of a particular block was made by this allocator
    bool owns(void* ptr)
    {
        char* startAsChar = static_cast<char*>(_bufferStart);
        return ptr >= startAsChar && ptr <= startAsChar + _bufferSize;
    }

private:
    size_t _bufferSize; //stores only for being able to implement owns function
    void* _bufferStart;
    FLNode* _firstFreeNode;
#ifdef DEBUG
    size_t _chunkSize;
#endif // DEBUG

};
```

Сегрегатор

Ще одна стратегія виділення пам’яті має назву сегрегатор. Під цим поняттям мається на увазі масив зі зв’язних списків вільних блоків пам’яті. Кожен зі зв’язних списків має свій унікальний фіксований розмір блоків. Часто розміри блоків є або степенем двійки, або дільником двійки(одиниці вимірювання – машинні слова). Для того, щоб задовольнити запит користувача на виділення пам’яті, достатньо взяти перший блок зі списку відповідного розміру. Додатковим плюсом такого підходу є економія пам’яті на заголовках, адже розмір блоків однаковий у межах одного зв’язного списку, тож розмір блоку можна не зберігати. Це може стати у пригоді для зберігання малих ділянок пам’яті. Адже зазвичай для ділянок маленького розміру заголовок може складати більш ніж половину розміру «корисної» пам’яті.

Потрібно зауважити, що після звільнення блоку певної величини, він не обов’язково повернеться у зв’язний список свого розміру. Спочатку перевіряють чи не є вільними сусідні блоки. Якщо хоча б один із сусідів виявився вільним, блоки зливаються в один для забезпечення можливості алокувати великі ділянки пам’яті.

Але існує і варіант сегрегатора, коли злиття сусідніх вільних блоків не відбувається. Вивільнена пам’ять завжди повертається у зв’язний список свого розміру. Коли ж відбувається запит на блок пам’яті такого розміру, якого немає у сегрегаторі, додаткова пам’ять виділяється для створення зв’язного списку з блоків такого розміру. Такий варіант є зручним і ефективним коли програма часто робить запити на об’єкти однакового розміру. Тож можна бути впевненим, якщо

ділянка певного розміру стала потрібною одного разу, то той самий розмір ще знадобиться програмі у майбутньому.

Але така стратегія буде показувати погані результати, якщо програма робить велику кількість алокацій одного розміру, видаляє їх, а потім повторює цей процес для інших розмірів.

Інший варіант реалізації сегрегатора – зберігати масив із зв’язних списків в середині яких зберігаються блоки приблизно одного розміру. Тобто визначається фіксований розмір блоку і припустиме відхилення.

Приклад реалізації сегрегатора:

```
//composite of freelists
template<size_t initialSize, size_t step, size_t numberOfSteps>
class Segregator final
{
public:
    Segregator(const size_t chunksPerList)
    {
        for (size_t i = 0; i < numberOfSteps; ++i)
        {
            _segregatedFreeLists[i] = new FreeListFixedSize(initialSize + i * step,
chunksPerList);
        }
    }
    ~Segregator()
    {
        for (size_t i = 0; i < numberOfSteps; ++i)
        {
            delete _segregatedFreeLists[i];
        }
    }

    //allocates if and only if there is a free list of this size, returns nullptr in
other cases
    void* allocate(size_t size)
    {
        if ((size - initialSize) % step != 0 || size - initialSize < 0 || (size -
initialSize) / step >= numberOfSteps)
        {
            //there is no appropriate size in segregator
            return nullptr;
        }

        const size_t appropriateSizeListIndex = (size - initialSize) / step;
        if (size < initialSize || appropriateSizeListIndex >= numberOfSteps)
        {
            return nullptr;
        }
        return _segregatedFreeLists[appropriateSizeListIndex]->allocate(size);
    }

    void free(void* ptr)
    {
        for (auto& freeList : _segregatedFreeLists)
        {
            if (freeList->owns(ptr))
            {
                freeList->free(ptr);
                return;
            }
        }
        assert(false && "Trying to free memory that wasn`t allocated with this
segregator");
    }


private:
    //sorted array of free lists
    FreeListFixedSize* _segregatedFreeLists[numberOfSteps];
    //std::array<FreeListFixedSize, numberOfSteps> _segregatedFreeLists;
};
```

Використання алокаторів для збору статистичної інформації

Вище були описані алокатори, які допомагають оптимізувати процес виділення пам’яті. Але алокатори також можна використовувати для фіксації статистичної інформації й для перевірки на коректність роботи з пам’яттю. Наприклад, до кожної виділеної ділянки пам’яті можна додавати префікс і суфікс, а під час звільнення блоку перевіряти чи збігаються дані. Один із цікавих прикладів використання алокаторів для збору статистичної інформації - це збереження в окремому місці назви файлу і номеру стрічки, де відбувся запит на виділення пам’яті. Також може бути корисним збереження адреси й розміру блоку пам’яті. Таким чином, можна відстежити які частини програми роблять найбільшу кількість запитів на виділення пам’яті. Більше того, на основі такої інформації стає можливим повністю відтворити роботу з пам’яттю певної сесії програми. Для цього потрібно створити окрему програму, яка буде робити запити на роботу з пам’яттю із записаної у файл статистики. Така програма не буде нічого робити із виділеною пам’яттю, окрім її видалення у певний момент часу. Але це може бути корисним для тестування і порівняння різних стратегій алокації. Звичайно це можна зробити й перезапускаючи програму кожного разу з новим алокатором, але це займе більше часу і порівняння буде не зовсім точне.

Що стосується збору статистики, алокатор як сутність зі станом може зберігати кількість запитів на виділення та звільнення пам’яті, середній розмір блоків, що виділяються.

Приклад використання алокатора для збору статистичної інформації

```
template <class Allocator>
class StatisticsAllocator final
{
public:

    StatisticsAllocator(Allocator& mainAllocator)
        :_mainAllocator(mainAllocator) {}

    void* allocate(size_t size)
    {
        ++_numberOfAllocations;
        if (_allocationSizeToQuantity.contains(size))
        {
            ++_allocationSizeToQuantity[size];
        }
        else
        {
            _allocationSizeToQuantity.emplace(size, 1);
        }
        void* chunk = _mainAllocator.allocate(size);
        if (chunk != nullptr)
        {
            ++_numberOfSuccesfulAllocations;
        }
        return chunk;
    }

    void free(void* ptr)
    {
        ++_numberOfDeallocations;
        return _mainAllocator.free(ptr);
    }

    void printStatistics() const
    {
        std::printf("numberOfAllocations: %i\nnumberOfSuccesfulAllocations:
%i\nnumberOfDeallocations: %i\n ", _numberOfAllocations, _numberOfSuccesfulAllocations,
_numberOfDeallocations);
    }

private:
    int _numberOfAllocations = 0;
    int _numberOfSuccesfulAllocations = 0;
    int _numberOfDeallocations = 0;
    std::unordered_map<size_t, int> _allocationSizeToQuantity;

    Allocator& _mainAllocator;
};
```

Висновки

Підсумовуючи, хочеться сказати, що алокатори спеціального призначення є дуже потужним інструментом оптимізації. Але його варто використовувати тільки там, де він справді потрібен. Для більшості простих програм вбудований алокатор загального призначення ідеально зробить свою роботу. Якщо ж в програмі наявні видимі проблеми у роботі операторів new та delete, варто розглянути написання алокатора спеціального призначення. Адже використані у правильному місці алокатори дають значний приріст продуктивності.

Список джерел

1. **Vlastimil Babka, Lubomír Bulej, Vojtěch Horký, Petr Tůma.** *Operating Systems.* 2018.

2. **Lea, Doug.** Code Browser by Woboq for C & C++. *dlmalloc.* [Онлайновий] 22 09 2005 р. https://code.woboq.org/gcc/libffi/src/dlmalloc.c.html.

3. **Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles.** *Dynamic Storage Allocation: A Survey and Critical Review.* 1989.

4. **Reinalter, Stefan.** Memory system – Part 1. *Molecular Musings.* [Онлайновий] 5 7 2011 р. https://blog.molecular-matters.com/2011/07/05/memory-system-part-1/.

5. **Lea, Doug.** *A Memory Allocator.* 1996.

6. **Development blog of the Molecule Engine. *Molecular Musings.* [Онлайновий] https://blog.molecular-matters.com/2011/08/03/memory-system-part-5/.**

7. Mazières, David. *My Rant on C++'s operator new.*

8. Knuth, Donald. *The Art of Computer Programming.* 1968. Т. 1.

9. Alexandrescu, Andrei. CppCon 2015: Andrei Alexandrescu “std::allocator...”.