Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

Порівняльний аналіз управління пам’яттю в мовах програмування

Текстова частина до кваліфікаційної роботи

за спеціальністю „Інженерія програмного забезпечення” 121

Керівник кваліфікаційної роботи

канд. фіз-мат. наук, доцент

Бублик В.В.

_____

(підпис)

“____”_____2023 р.

Виконала студентка Шляхова О. Д.

“_____”_____2023 р.

Київ 2023

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри мультимедійних

систем, доцент, канд. ф.-м. наук

_____ О. П. Жежерун

«____» _____2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на кваліфікаційну роботу

студентки 4-го курсу, факультету інформатики
Шляхової Олександри Дмитрівни

Тема: Порівняльний аналіз управління пам’яттю в мовах програмування
Зміст ТЧ до кваліфікаційної роботи:

Індивідуальне завдання
Календарний план
Зміст
Анотація
Перелік термінів та умовних позначень
Вступ
Загальні відомості
Дослідження існуючих методів управління пам’яттю
Характеристичний опис стратегій управління пам’яттю в мовах програмування
Практичне порівняння використання пам’яті у програмах написаних мовами програмування C++ та Java
Висновки
Список використаної літератури

Дата видачі «___» _____ 2023 р.

Керівник _____
(підпис)

Завдання отримав _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ РОБОТИ

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1	Визначення теми	01.10.2023	
2	Дослідження предметної області	01.11.2022	
3	Пошук тематичної літератури	15.11.2022	
4	Ознайомлення зі знайденою літературою	01.02.2023	
5	Написання вступної частини	15.02.2023	
6	Написання теоретичної частини	15.03.2023	
7	Реалізація програм для порівняння	01.04.2023	
8	Аналіз отриманих результатів	15.04.2023	
9	Написання висновків	01.05.2023	
10	Підготовка презентації	10.05.2023	
11	Захист роботи	01.06.2023	

Студентка Шляхова О. Д.

Керівник Бублик В.В.

“____” _____

Зміст

Анотація 6

Перелік прийнятих скорочень 7

Вступ 8

1 Загальні відомості 9

1.1 Опис концепції віртуальної машини 9

1.2 Загальний опис способів управління пам'яттю 9

2 Дослідження існуючих методів управління пам'яттю 10

2.1 Сегменти пам'яті, які використовує програма 10

2.2 Ручне управління пам'яттю 12

2.3 Підрахунок посилань 14

2.4 Автоматичне управління пам'яттю 19

2.4.1 Поняття досяжності об'єкта 20

2.4.2 Прибирання сміття відстеженням 23

2.4.3 Компактувальне прибирання сміття 24

2.4.4 Інкрементний збирач сміття 27

2.4.5 Генераційний збирач сміття 28

3 Характеристичний опис стратегій управління пам'яттю в мовах програмування 29

3.1 C++ 29

3.2 Java 31

4 Практичне порівняння використання пам'яті у програмах написаних мовами C++ та Java 34

4.1 Опис програми 34

4.2 Проста реалізація 36

4.3 Оптимізована реалізація 39

4.4 Порівняння швидкодії регулярних виразів в реалізації бібліотек Boost та STL 41

4.5 Розширення програм графічним інтерфейсом 43

4.6 Порівняння роботи прибиральників сміття, які надає мова програмування Java 48

4.6.1 Serial Garbage Collector 48

4.6.2 Parallel Garbage Collector 51

4.6.3 Garbage First Garbage Collector 53

4.6.1 Висновок після порівнянь роботи прибиральників сміття мови програмування Java 55

4.7 Аналіз ефективності застосунку з графічним інтерфейсом мовою C++ 57

Висновки 58

Лістинги коду 60

Список ілюстрацій 61

Список таблиць 62

Список джерел 63

Анотація

Шляхова О.Д. Порівняльний аналіз управління пам’яттю в мовах програмування.

Проект містить 63 сторінки, 20 ілюстрацій, 9 лістингів коду, 8 таблиць та 10 джерел.

Ключові слова: управління пам’яттю, порівняння, C++, Java, прибирання сміття, JavaFX, Qt.

Об’єктом дослідження є програма для створення інверсного індексу мовами програмування C++ та Java і порівняння ефективності реалізацій з точки зору використання пам’яті.

Мета роботи: дослідити вплив механізмів управління пам’яттю мов програмування C++ та Java на ефективність роботи програми зі створення інверсного індексу.

Кваліфікаційна робота присвячена розробці програми для створення інверсного індексу мовами програмування C++ та Java для порівняння ефективності отриманих рішень з точки зору використання пам’яті. Було досліджено існуючі способи управління пам’яттю, використання цих способів в мовах програмування C++ та Java. Знання, отримані в результаті дослідження, були використані для аналізу ефективності роботи з пам’яттю програми для створення інверсного індексу.

В результаті роботи було написано 2 застосунки мовами C++ та Java, порівняно їх швидкодію та об’єми використаної пам’яті, зроблені заміри службових регіонів пам’яті, які використовує віртуальна машина Java, надані вказівки щодо типу прибиральника сміття, який найкраще використовувати для розв’язання задач побудови інвертованого індексу та використання його в застосунку з графічним інтерфейсом.

Перелік прийнятих скорочень

JVM – віртуальна машина Java;

RAII - «Отримання ресурсу є ініціалізація» (Resource Acquisition Is Initialization) ;

JIT компілятор – Just in Time компілятор;

GC – прибиральник сміття (Garbage Collector);

G1 – Garbage First

Вступ

Сучасні мови програмування надають величезну кількість різних способів управління пам’яттю. Від ручного керування в C та C++ до автоматичного підходу з використанням прибиральника сміття в Java, C# та Python. Кожен з варіантів має свої переваги і недоліки.

Ручне управління пам’яттю полюбляють за його швидкість, але водночас, уникають через складність використання, помилки подвійного видалення та витоку ресурсів.

Мови з автоматичним управлінням пам’яттю обирають за легкість написання коду, але уникають через накладні витрати, які додає реалізація прибиральника сміття.

У цій роботі детально розглядаються техніки управління пам’яттю починаючи від ручної завершуючи автоматичною, надаються описи сучасних алгоритмів, які використовуються в рамках кожної з технік. Також, наявна описова характеристика найпопулярніших способів управління пам’яттю в мовах програмування C++ та Java.

У практичній частині роботи розглянуто задачу побудови інверсного індексу з графічним інтерфейсом та аналіз різних реалізацій з точки зору використання пам’яті. Зроблені заміри роботи програми з використанням різних стратегій прибирання сміття та надані рекомендації щодо найбільш доцільної з них.

1 Загальні відомості

1.1 Опис концепції віртуальної машини

Сучасні мови програмування можна поділити на компільовані та інтерпретовані. Інтерпретовані мови з’явилися для вирішення проблем кросплатформності. У них вводиться поняття байт-коду, який конвертується у машинний код під час виконання програми. Для забезпечення такої конвертації, інтерпретовані мови програмування реалізують віртуальну машину. Зрозуміло, що для підтримки її функціонування потрібна додаткова пам’ять. Тож, до власних витрат програми додаються накладні витрати на підтримку роботи віртуальної машини.

1.2 Загальний опис способів управління пам’яттю

Управління пам’яттю поділяють на ручне і автоматичне. Найпершим з’явилося ручне управління пам’яттю. Воно передбачає, що розробник відповідальний за своєчасне звільнення об’єктів. Інший варіант управління пам’яттю – підрахунок посилань, який спирається на ідіому RAII. Цей метод надає різні види указників, які користувач має обирати в залежності від типу володіння об’єктом і забезпечує автоматичне звільнення пам’яті, коли усі посилання на об’єкт виходять із зони видимості. Останній, найбільш поширений метод управління пам’яттю – автоматичний. Він звільняє користувача від необхідності вручну видаляти створені об’єкти. Мови програмування з таким типом управління пам’яттю вводять нову сутність – прибиральник сміття. Він відповідає за аналіз пам’яті купи, знаходження об’єктів, які більше не використовуються програмою, і видаляє їх. Автоматичне управління пам’яттю було створено у 1959 році для мови програмування Lisp. З того часу з’явилося багато різних алгоритмів прибиральників сміття, як-от: компактувальне, інкрементне, генераційне тощо.

2 Дослідження існуючих методів управління пам’яттю

У цьому розділі буде представлене дослідження концепцій, описаних у розділі загальні відомості.

2.1 Сегменти пам’яті, які використовує програма

Перед тим, як порівнювати управління пам’яттю, потрібно розібратись з визначеннями. По-перше, важливо зрозуміти чим саме відбувається управління. Навіть коли програма не запущена, її виконуваний файл забирає певне місце. У ньому зазвичай зберігається машинний код самої програми(або байт-код для інтерпретованих мов програмування), глобальні змінні, а також інші секції, які мають службовий характер. На рисунку 1 зображена структура виконуваного файлу операційної системи Windows.

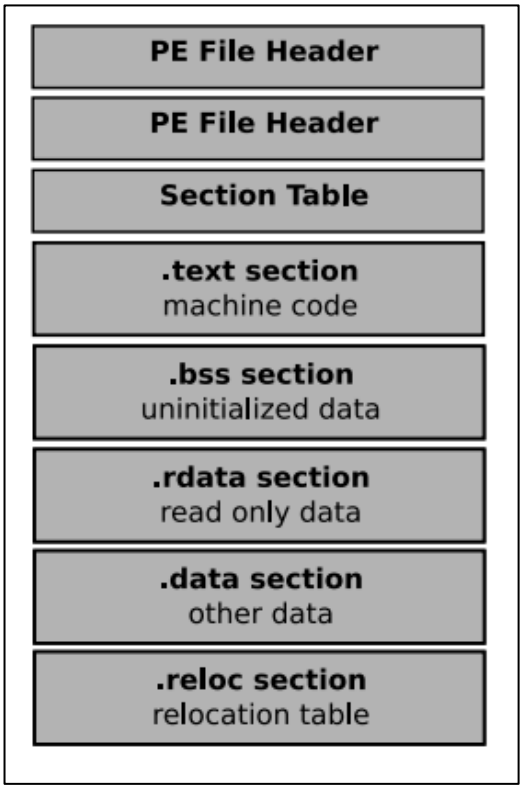


Рисунок 1 - Структура виконуваного файлу операційної системи Windows

Під час виконання частина цих даних буде завантажена у пам’ять, виділену для програми. Програма викликає певний набір функцій, кожна з

яких може мати локальні змінні і викликати інші функції. Ці операції потребують пам’ять. Тож, програми перед початком роботи резервують сегмент пам’яті, який називається стек. Стек складається із фреймів. Локальні змінні, що створюються під час виконання функції, знаходяться у фреймі відповідної функції. Також, фрейм зберігає деяку службову інформацію, як-от: адресу повернення із функції, канарку тощо. Кожен фрейм створюється під час виклику функції і звільняється під час повернення з функції. Оскільки локальні змінні знаходяться у фреймі, час їх життя обмежений часом виконання функції. Також, розмір локальних змінних має бути відомим під час компіляції.

Для того, щоб зберігати змінні, які можуть існувати довше, ніж функція, що їх створила, а також мати можливість визначати розмір змінних під час виконання програми, існує ще один сегмент пам’яті – купа. Попри усі переваги зберігання змінних у купі є і недоліки. По-перше, оскільки життя змінної не обмежується програмою, хтось має вчасно видаляти непотрібні об’єкти. Якщо ж цього не робити, програма швидко використає усі ресурси пам’яті і не зможе більше працювати. По-друге, процес знаходження вільного місця для нової змінної є набагато складнішим і породжує купу нових проблем.

Процес розв’язання питань, пов’язаних з динамічним виділенням пам’яті, називається управлінням пам’яттю. Кожна мова програмування по-різному вирішує ці проблеми. Одні доручають користувачу процес виділення і звільнення пам’яті. Такий спосіб управління називається ручним і використовується у C та C++. Повна протилежність ручному управлінню пам’яттю – автоматичне. Цей підхід зустрічається у більшості сучасних мов програмування як-от: Java, C#, Python, JavaScript, Prolog тощо. Особливістю автоматичного управління пам’яттю є використання прибиральника сміття. Це сутність, що вивільняє пам’ять, яка більше не використовується системою.

2.2 Ручне управління пам’яттю

Як зазначалося вище, у системах з ручним управлінням пам’яттю розробник самостійно повертає пам’ять системі. У мові програмування С для виділення пам’яті використовується функція **malloc**, а для вивільнення – **free**.

```
int main(){

    int* array;
    int length;

    scanf("%d", &length);

    //malloc takes amount of needed memory as a parameter
    //size of an array is length times size of one
element
    array = (int*) malloc(length * sizeof(int));

    /** using the array */

    //freeing requested memory explicetely
    free(array);

    return 0;
}
```

Лістинг 1- ручне управління пам'яттю в мові програмування С

У С++ використовуються оператори **new** та **delete**. Аналогічний код буде виглядати наступним чином:

```
int main(){
    int length;
    scanf("%d", &length);

    //creating an array of size length
    int* array = new int[length];

    //allocating an int on the heap
    int* n = new int{5};

    /** ... */

    //freeing an array
    delete[] array;

    //freeing a pointer
    delete n;

    return 0;
}
```

Лістинг 2 - Ручне управління пам'яттю в С++

Ручне управління пам’яттю набагато швидше за автоматичне, адже програмі не потрібно витрачати ресурси на аналіз пам’яті застосунку для знаходження ділянок, що більше не використовуються. Але є і недоліки. По-перше, розробник може забути викликати delete, що призведе до витоку пам'яті. На перший погляд здається, що це зовсім не проблема, просто потрібно писати код більш уважно і не забувати звільняти виділену пам’ять. Але слідкувати стає важче, коли програми стають більшими за декілька стрічок. Також, потрібно завжди пам’ятати про виняткові ситуації, які можуть перервати програму у неочікуваному місці. Більше того, не зовсім зрозуміло як бути із функціями, які створюють об’єкт всередині себе і повертають указник на нього(наприклад, патерн будівник [1]).

По-друге, під час ручного управління пам’яттю можлива проблема подвійного вивільнення. Її можна розв’язати зануленням звільнених указників. Але це покладає додаткову відповідальність на розробника.

```
int main()
{
    /** ... */

    int* a = new int{1};
    int* b = new int{1};

    /** ... */
    //delete is not called at all, a and b leak

    return 0;
}
```

Лістинг 3 - Приклад витоку пам'яті в C++

```
int main()
{
    /** ... */
    int* a = new int{1};
    int* b = new int{1};

    /** delete is called and everything seems fine
        but operator new could throw an exception
        if there is not enought memory for the program
        if that happens for b when a is already created
        a`s memory will leak
    */
    delete a;
    delete b;

    return 0;
}
```

Лістинг 4 - Приклад неочевидного витоку пам'яті

2.3 Підрахунок посилань

Існують альтернативні способи ручного управління пам’яттю. Найбільш популярний варіант пропонує ідіома «Отримання ресурсу є ініціалізація» (Resource Acquisition Is Initialization) або RAII. Як зазначено у [2] RAII – це техніка для управління ресурсами, яка базується на концепції області видимості. У C++ при створенні об’єктів завжди викликається конструктор, а коли їх життєвий цикл завершується – викликається деструктор. Якщо ж об’єкт створено на стеку, його деструктор буде викликано автоматично, коли об’єкт вийде з області видимості. Також деструктор буде викликано у разі аварійного завершення процедури. Такі гарантії мови програмування дозволяють зручно управляти ресурсами. Усе, що потрібно зробити розробнику – виділити ресурс у конструкторі класу-обгортки, і звільнити у деструкторі. Загалом RAII більш загальний термін, який застосовується, коли мова іде про управління будь-якими ресурсами: файлами, потоками і, найголовніше, пам’яттю.

Тож, для управління пам’яттю за допомогою ідіоми RAII потрібно створити клас-обгортку, конструктор якого буде виділяти пам’ять, а деструктор – звільняти. Проста реалізацію могла б виглядати таким чином:

```
template<class T>
class smart_ptr
{

private:
    T* ptr;

public:

    explicit smart_ptr(T* p): ptr(p) { } //constructor caches the pointer to manage

    ~smart_ptr(){ delete ptr; } // destructor frees the memory

    //using operator* and operator-> to make smart_ptr behave as a regular pointer
    T& operator*() const { return *ptr; }
    T* operator->() const { return ptr; }
};

int main()
{
    /** ... */
    int* a = new int{1};

    int* b = new int{1};
    smart_ptr<int> b_ptr{b};

    //can use operator* for smart_ptr just like with plain pointers
    int c = *a + *b;

    //memory for b is automatically deleted as whe exception occurs smart_ptr`s destructor
    //frees the memory, while a`s resources leak
    throw std::exception("unexpected exception");

    //need to manually delete a
    delete a;

    //but no need to manually delete b as it will be deleted when it goes out of scope

    return 0;
}
```

Лістинг 5 - Приклад реалізації розумного указника

Ця реалізація є дуже базовою, але вона показує принцип, за яким будуються подібні класи-обгортки. Стандартна бібліотека C++ надає декілька варіантів шаблонних класів для управління пам’яттю: unique_ptr, shared_ptr та weak_ptr.

unique_ptr надає одноосібний спосіб управління пам’яттю. Такі указники не можна копіювати, можна лише переміщувати пам’ять, якою він управляє, у новий об’єкт, таким чином гарантуючи унікальне володіння ресурсом.

Для випадків, коли потрібно підтримувати декілька посилань на ділянку пам’яті і звільняти її тільки коли останнє посилання вийшло із зони видимості, використовується shared_ptr. Всередині цього розумного указника зберігається лічильник, що збільшується кожного разу, коли створюється нове посилання, і зменшується в деструкторі, коли таке посилання видаляється. Коли ж лічильник доходить до нуля, пам’ять звільняється. Але такий елегантний метод не може вирішити проблему циклічних посилань.

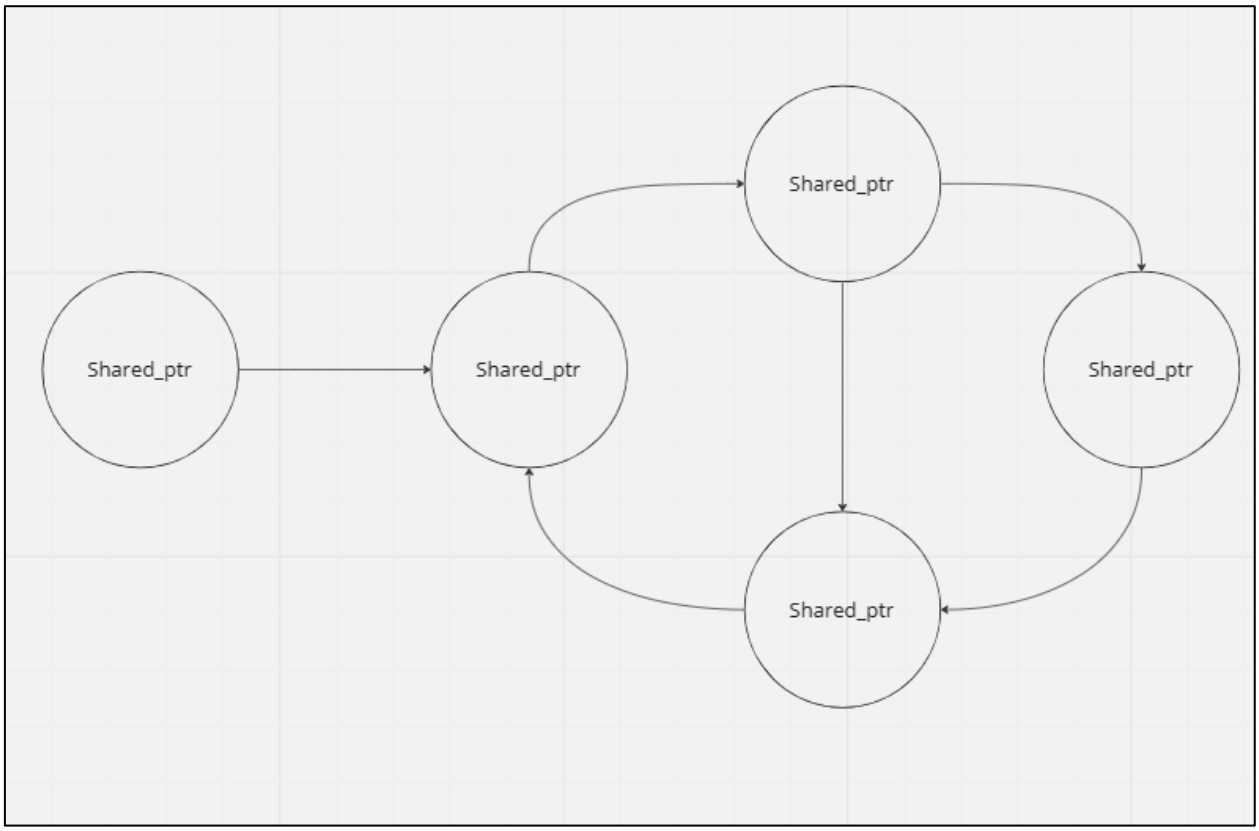


Рисунок 2 - Схема циклічних посилань

У випадку, зображеному на малюнку, жоден з об’єктів не буде видалено, адже їх лічильники ніколи не опустяться до нуля. Якщо така проблема виникає всередині однієї структури даних (наприклад, граф в якому можуть бути

цикли), її можна вирішити за допомогою іншого представлення об’єктів в структурі даних, як показано у відео [3]

Як зазначають [4], у подібного підходу до управління пам’яттю є свої плюси і мінуси. З плюсів можна назвати моментальне звільнення пам’яті, коли закінчуються усі посилання на об’єкт, що призводить до хорошої роботи в умовах, коли купа майже повністю заповнена. Також варто зазначити, що така техніка не потребує знання про внутрішнє влаштування об’єктів або системи рефлексії, що є необхідним для управління пам’яттю в інших мовах програмування. Серед недоліків варто зазначити додаткову пам’ять, яку займають розумні указники. Також у випадку з shared_ptr потрібно додатково зберігати блок із кількістю посилань[1].

Ще одним недоліком є те, що користувацькі потоки виконання отримують певний програш у швидкодії, адже кожна операція копіювання указника тягне за собою додаткову операцію - збільшення лічильника, яка має бути атомарною в багатопотокових застосунках для підтримки коректної роботи і уникнення завчасного видалення пам’яті.

Ще одна проблема, яка може виникнути через звільнення ресурсів у деструкторі об’єкту, це переповнення стеку через рекурсивні виклики. У наведеному нижче лістингу видалення кореневого об’єкта в деструкторі дерева призведе до рекурсивного видалення усіх дочірніх вузлів. Якщо деревовидна структура є великою за розміром, програма завершиться аварійно через переповнення стеку.

Для вирішення перелічених проблем існують розширені варіанти підрахунку посилань. До них можна віднести відкладене, об’єднане та буферне підрахування посилань.

[1] Важко назвати такі накладні витрати великими, порівнюючи їх з накладними витратами, які неодмінно наявні в реалізаціях автоматичного управління пам’яттю. Але, в контексті порівняння підрахунку посилань з ручним управлінням пам’яттю, цей недолік варто згадати

```
template <typename T>
class Tree {
    struct Node {
        template <typename... Args>
        Node(Args... args)
            :data(args...) {}

        std::unique_ptr<Node> left;
        std::unique_ptr<Node> right;
        T data;
    };

public:
    template <typename... Args>
    void insert(Args&&... args) {
        std::unique_ptr<Node>                    newNode                    =
std::make_unique<Node>(std::forward<Args>(args)...);

        if (root == nullptr) {
            root = std::move(newNode);
            return;
        }

        Node* current = root.get();
        while (true) {
            if (newNode->data < current->data)
            {
                if (current->left == nullptr)
                {
                    current->left = std::move(newNode);
                    break;
                }
                else
                {
                    current = current->left.get();
                }
            }
            else
            {
                if (current->right == nullptr)
                {
                    current->right = std::move(newNode);
                    break;
                }
                else
                {
                    current = current->right.get();
                }
            }
        }
    }

private:
    std::unique_ptr<Node> root;

};

int main() {
    Tree<int> tree;
    for (int i = 0; i < 10000; ++i){
        tree.insert(i);
    }

    return 0;
}
```

Лістинг 6 - Приклад деревовидної структури з використанням розумних указників, який може призвести до виняткової ситуації в деструкторі

2.4 Автоматичне управління пам’яттю

Повною протилежністю ручному управлінню пам’яттю є автоматичне. У системах такого типу з’являється нова сутність – прибиральник сміття. Він відповідає за знаходження і вивільнення областей пам’яті, які більше не використовуються програмою. Багато сучасних мов програмування реалізують такий прибиральник сміття як єдиний спосіб управління пам’яттю.

Якщо звернутись до історії, прибиральник сміття винайшов Джон Маккарті під час розробки мови програмування Lisp. У своїй роботі [5] він писав:

This process, because it is entirely automatic, is more convenient for the programmer than a system in which he has to keep track of and erase unwanted lists. Its efficiency depends upon not coming close to exhausting the available memory with accessible lists. This is because the reclamation process requires several seconds to execute, and therefore must result in the addition of at least several thousand registers to the free-storage list if the program is not to spend most of its time in reclamation.

Таким чином він намагався звільнити розробника від необхідності вивільняти пам’ять вручну. На таке рішення можна дивитись з різних боків, на мою думку, це досить добре виражено у цитаті:

> *C programmers think memory management is too important to be left to the computer. Lisp programmers think memory management is too important to be left to the user.*

Існує багато способів реалізації прибиральника сміття. Кожен з них має свої переваги і недоліки. Сучасні мови програмування реалізують різні алгоритми. Тож, для коректного порівняння потрібно розглянути існуючі способи реалізації.

2.4.1 Поняття досяжності об’єкта

Алгоритми прибирання сміття різняться, але для реалізації кожного з них потрібно знайти об’єкти, що не використовуються, або «сміття». Для цього потрібно ввести формальне означення. Теоретично, об’єкт може вважатися «сміттям» в певний момент часу, якщо до завершення програми його більше не буде використано жодного разу. Але таке формулювання неможливо описати кодом. Для більш чіткого визначення можна ввести поняття досяжності.

Об'єкт N вважається досяжним відносно об’єкту M, якщо за ланцюгом указників об’єкту M можна дістатись до N.

[4]

Червоні об’єкти на схемі вважаються недосяжними, адже до них немає шляху від кореневих об’єктів.

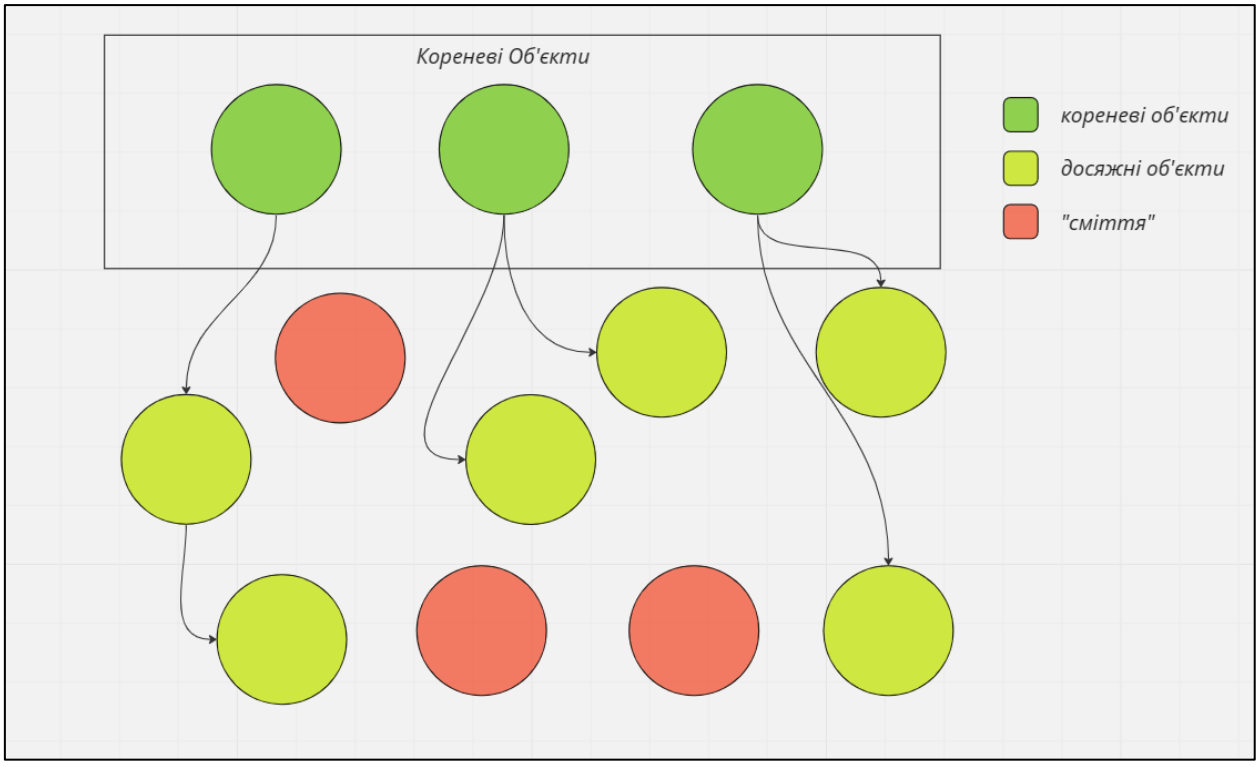


Рисунок 3- Схематичне зображення об'єктів в купі

Тож, у програмі виділяється множина кореневих об’єктів, а усі об’єкти, які є досяжними відносно них, вважаються живими. Якщо з множини усіх

об’єктів прибрати досяжні, отримаємо «сміття», з яким буде працювати система автоматичного управління пам’яттю. Реалізація цієї задачі сильно різниться в залежності від гарантій, які надає мова програмування. Наприклад, Java підтримує систему рефлексії. За допомогою неї можна ітеруватися полями класу для знаходження посилань. Також, можна бути впевненим у тому, що знайдене поле є указником, а не, наприклад, примітивним типом даних.

На противагу, C++ не надає вбудованої системи рефлексії, тому існуючі реалізації прибиральників сміття цією мовою для знаходження указників в об’єктах усі ділянки трактують як потенційний указник. Прибиральники сміття такого типу називаються консервативними.

```
public static Set<Object> GetReachable(List<Object> rootObjects) throws
IllegalAccessException {

    //all reachable objects found from @rootObjects
    Set<Object> reachable = new HashSet<>();

    //stack is used to avoid recursion
    Stack<Object> objectsToProcess = new Stack<>();
    objectsToProcess.addAll(rootObjects);

    while (!objectsToProcess.isEmpty()) {
        Object object = objectsToProcess.pop();
        reachable.add(object);

        Field[] Fields = object.getClass().getDeclaredFields();
        //iterating over references of an object
        for (Field field : Fields) {
            if (!field.getType().isPrimitive()) {
                Object reference = field.get(object);
                if (!reachable.contains(reference)) {
                    //adding objects to process to the stack
                    objectsToProcess.add(reference);
                }
            }
        }
    }

    return reachable;
}
```

Лістинг 7 - Приклад знаходження досяжних об'єктів мовою програмування Java

```
/* Useful structure to store heap allocated objects in a linked list*/
struct header {
    /*size of the object*/
    unsigned int size = 0;

    /*pointer to next used node*/
    header* next = nullptr;

    operator int() const { return (unsigned int)this; }
};

/** @usedList - linked list of object allocated on heap(header goes just
before the object to maintain list structure)
*   @rootObjects - objects collected from the stack, BSS and initialized data
segment
*
*   Finds headers of all heap objects reachable from the root set, they can
be freed further
*/
static std::unordered_set<header*> findReachable(header* usedList,
      const std::unordered_set<header*>& rootObjects)
{
    std::unordered_set<header*> reachable{ rootObjects };
    header* usedObject = usedList;
    do {
        if (reachable.find(usedObject) != reachable.end())
        {
            //searching for references only from roots
            continue;
        }
        //iterating over pointer sized blocks of every used objects
        for (unsigned int* potentialPointer =
            (unsigned int*)(usedObject + 1);
            potentialPointer < (unsigned int*)(usedObject + usedObject->size
     + 1); ++potentialPointer)
        {
            unsigned int pointerValue = *potentialPointer;
            header* otherUsedObject = usedObject->next;
            // iterating over all used headers, if potential pointer points
somewhere inside one of used objects. marking that as reachable
            do {
                if (otherUsedObject != usedObject && // not checking for
references in ourselves
                    (unsigned int)(otherUsedObject + 1) <= pointerValue &&
                    (unsigned int)(otherUsedObject + 1 + otherUsedObject-
>size) > pointerValue)
                {
                    reachable.insert(otherUsedObject);
                    break;
                }
            } while ((otherUsedObject = otherUsedObject->next) !=
usedObject);
        }
        usedObject = usedObject->next;
    } while (usedObject != usedList);
}
```

Лістинг 8 - Приклад знаходження досяжних об'єктів мовою програмування C++

2.4.2 Прибирання сміття відстеженням

Прибирання сміття відстеженням(mark and sweep collector) для виконання своєї задачі помічає усі досяжні об’єкти на фазі знаходження посилань, після завершення цього процесу починається другий етап – очищення. Під час очищення відбувається ітерація по усім об’єктам купи. Ті з них, які не було помічено на першій фазі, очищуються. Зазвичай процес очищення викликається, коли система більше не може задовольнити потреби пам’яті користувача.

Варто зауважити, що прибирання сміття відстеженням вважають непрямим алгоритмом на відміну від підрахунку посилань. Адже ідентифікується не саме сміття, а живі об’єкти. А уже потім сміттям вважаються усі об’єкти, що не було ідентифіковано як живі.

Під час роботи алгоритму виконання основної програми припиняється. З опису алгоритму зрозуміло, що час роботи пропорційний розміру купи. Така складність є не дуже хорошою. Але варто розуміти, що цей алгоритм активно використовувався у 1960 роках, коли об’єми пам’яті, які потрібно проаналізувати не були великими. Зараз же, коли програми потребують значно більше пам’яті, такий колектор буде відпрацьовувати дуже довго.

Окрім того, цей алгоритм посилює фрагментацію. Він залишає невеликі вільні ділянки серед блоків використаної пам’яті. З часом знайти місце для великих об’єктів стає дедалі складніше.

2.4.3 Компактувальне прибирання сміття

Описаний вище алгоритм називають mark-and-sweep(дослівно можна перекласти як помічай і видаляй). Існує інша його варіація, яка називається mark-compact(помічай і стискай). Він намагається розв'язати проблему фрагментації, яка виникає у mark-and-sweep.

Для зменшення фрагментації, під час кожного циклу прибирання сміття, живі об'єкти переміщуються так, щоб розташовуватись послідовно.

Найпростішим способом реалізації цієї ідеї є Two-Finger алгоритм. Після завершення першої фази знаходження досяжних об'єктів, запускається фаза стискання. На ділянку, яку потрібно стиснути, ставлять два вказівника – початок та кінець регіону. Далі указник початку поступово рухається у напрямку кінця, поки не знайде «вакантне місце». Після цього задача другого указника – рухаючись у напрямку початку, знайти досяжний об'єкт, який можна поставити на «вакантне місце». Якщо такий об'єкт буде знайдено – його пересувають на указник, який спершу був на початку регіону, і алгоритм повторюється знову. Стискання вважається завершеним, коли під час пошуку живого об'єкту другий указник зустрівся з першим.

Алгоритм досить легко візуалізувати за допомогою двох пальців, які поступово просуваються на зустріч один одному. Звідси і з'явилася така назва. На останньому етапі роботи потрібно оновити адреси посилань, які змістилися під час зсуву.

Підсумовуючи, такий алгоритм зменшує фрагментацію і є достатньо простим для розуміння. Він добре підходить для очищення регіонів з блоками одного розміру, тому може скласти хорошу пару Buddy system[2] алокатору.

Але є і недоліки. По-перше, внаслідок роботи алгоритму об'єкти, які були розташовані поруч, можуть опинитися досить далеко один від одного. Це може

[2] Алокатори такого типу розділяють отриманий регіон пам'яті на блоки однакового розміру

мати погані наслідки через механізми роботи кеш пам’яті. Зазвичай об’єкти, які часто використовуються разом, зберігають поруч у пам’яті, таким чином, щоб вони розташовувались на дистанції однієї кеш лінії. А описаний вище алгоритм перемістить досяжні об’єкти у випадкові «вакантні» місця.

```
    //to operate on one byte
    using Block = unsigned char;
    bool isReachable(Block* ptr);

    void twoFingerCompacting(Block* start, Block* end, const size_t
blockSize)
    {
        while (end > start)
        {
            if (isReachable(start))
            {
                start += blockSize;
            }
            else
            {
                end -= blockSize;
                if (isReachable(end))
                {
                    memcpy(start, end, blockSize);
                }
            }
        }
    }
```

Лістинг 9 - Приклад реалізації Two-Finger Collector

По-друге, для реалізації цього алгоритму на об’єктах не однакового розміру, структура зберігання блоків має підтримувати ітерацію у зворотному порядку, чого часто немає на практиці.

Варто зазначити, що фрагментацію можна зменшити не тільки за рахунок правильного алгоритму очищення пам’яті, а і за допомогою зміни стратегії її виділення. Недоліки колекторів, які посилюють фрагментацію, можуть бути згладжені стратегією алокатора і навпаки. Більшість колекторів мають працювати у зв’язці з алокатором не тільки для згладження недоліків один одного, а і для отримання службової інформації. Наприклад, колектору часто потрібно знати про усі створені програмою об’єкти та їх розмір. Цю інформацію може зберігати алокатор.

Більш використовуваною стратегією компактування є алгоритм підпросторів. Його було вперше запропоновано у 1970 році як модифікацію mar-and-sweep. Задумка полягає у тому, щоб розділити купу на 2 підпростори і розташовувати об’єкти лише в одній половині. Коли ж настає час «прибирати сміття», живі об’єкти послідовно переміщуються у вільну половину, а на їх попередньому місці записується нова адреса об’єкту. Посилання на об’єкти купи оновлюються після повного переміщення, і та половина, в якій раніше розташовувались живі об’єкти, стає резервною. Вона буде використана знову під час наступного циклу очищення.

З одного боку, такий алгоритм зводить фрагментацію до нуля, усі «живі» об’єкти розташовуються послідовно, що допомагає програмі використовувати переваги кеш пам’яті. З іншого боку, половина купи програми ніколи не буде використано, що є не зовсім ефективним з точки зору використання пам’яті.

У цьому алгоритмі, на відміну від попереднього, послідовність розташування об’єктів зберігається, що покращує швидкодію через меншу кількість промаху кешу.

Варто також зазначити, що такий алгоритм прибирання сміття суттєво знижує час на виділення нової пам’яті. Адже через те, що усі об’єкти розташовуються послідовно, для знаходження нового блоку пам’яті потрібно просто повернути ділянку пам’яті відповідного розміру за останнім використаним об’єктом.

Поділення купи над підпростори використовується у сучасних мовах програмування з прибиральниками сміття, наприклад у Java та JavaScript.

2.4.4 Інкрементний збирач сміття

Інкрементний алгоритм прибирання сміття - це спроба розділити процес прибирання сміття на певну кількість ітерацій. Генрі Бейкер у своїй статті [6] запропонував такий алгоритм для зменшення часу роботи прибиральника сміття у Lisp.

Тож, прибиральник сміття спочатку проводить звичайну фазу маркування живих об’єктів і переносить певну кількість живих об’єктів, але коли певний ліміт часу перевищується, цей процес зупиняється. Відновлюється ж він частково кожного разу, коли програма використовує указник, який не було перенесено. В такому разі основна програма зупиняється на деякий обмежений час і деяка кількість об’єктів переноситься у правильний підпростір.

Таким чином гарантується, що зупинки програми не будуть перевищувати певний ліміт часу. Але недоліком такої реалізації є додаткові витрати на перевірку валідності указника кожного разу при роботі з ними.

Інкрементною можна зробити і фазу маркування. В такому разі процес знаходження недосяжних об’єктів проводиться невеликими за часом ітераціями. Це також допомагає зменшити затримки, які неодмінно присутні у всіх реалізаціях прибиральників сміття. Таку техніку використовує прибиральник сміття мови Ruby.

2.4.5 Генераційний збирач сміття

Згідно з гіпотезою слабкої генераційності [4], більшість об’єктів мають невеликий життєвий цикл і стають сміттям досить швидко. Також існує множина об’єктів з довгим життєвим циклом. Вважають, якщо об’єкт пережив декілька циклів прибирання сміття, його життєвий цикл буде довгим. На цій гіпотезі базується алгоритм генераційного збирача сміття.

Для його реалізації купу поділяють на декілька регіонів(зазвичай два або три), кожен регіон зберігає об’єкти різних поколінь. На початку усі об’єкти розташовуються у регіоні молодого покоління. Після декількох циклів прибирання сміття об’єкти, які залишились живими, переміщаються у регіон наступного покоління.

Наступні цикли прибирання перевіряють лише регіон молодого покоління, адже за гіпотезою описаною вище існує велика вірогідність, що більшість об’єктів старших регіонів ще живі. Старші ж регіони також очищуються, але з набагато меншою частотою.

Але при такому підході треба окремо обробляти указники із старших регіонів в молодші. Без урахування цього об’єкти можуть бути завчасно видалені.

Генераційний прибиральник сміття є дуже популярним, він використовується в мовах C#, Java, JavaScript та Python.

3 Характеристичний опис стратегій управління пам’яттю в мовах програмування

В цьому розділі на основі знань, описаних вище, надається характеристичний опис способів управління пам’яттю в мовах програмування C++ та Java. Вони є досить популярними, але мають різні підходи до управління пам'яттю. C++ є репрезентативним представником мови з ручним управлінням пам'яттю, який також підтримує підрахунок посилань. А Java є класичним представником мови з автоматичним управлінням пам'яттю, яка на додаток до цього надає користувачу вибір між декількома різними прибиральниками сміття. Тож на її прикладі можна розглянути і порівняти різні алгоритми

3.1 C++

У мові програмування C++ основним способом управління пам’яттю є ручне. Розробнику надається декілька альтернатив.

По-перше, є можливість використання операторів **new** та **delete**. Але, як зазначалося раніше, пряме використання цих операторів може призводити до помилок подвійного видалення та витоку ресурсів. Ось що Б'ярн Страуструп говорив про це:

When I see a delete in application code I think there must be a bug somewhere.
Why? Because if there is one delete there`s probably should be more.

[2]

Тому стандартна бібліотека C++ надає декілька розумних указників, які працюють по принципу RAII.

- unique_ptr надає одноосібний спосіб володіння об’єктом, підтримує лише «перенесення» об’єкта зі старого указника в новий

- shared_ptr надає спільний доступ до об’єкту, кожна копія такого посилання збільшує внутрішній лічильник, об’єкт буде видалено тоді, коли усі посилання вийдуть з області бачення
- weak_ptr надає змогу посилатися на об’єкт, не впливаючи на його довжину життя, та перевіряти чи не видалений іще об’єкт за посиланням.

Використовуючи розумні указники зі стандартної бібліотеки розробник захищає себе від помилок завчасного або подвійного видалення об’єктів та витоку пам’яті.

Як зазначалося вище, підрахування посилань не може вивільнити пам’ять для структур, які містять циклічні посилання. Цю проблему можна розв’язувати різними способами. Наприклад, Герб Саттер запропонував новий тип розумного указника deferred_ptr, який можна використовувати для реалізації структур з підтримкою циклічних посилань [3].

Альтернативним способом управління пам’яттю в C++ є використання прибиральників сміття зі сторонніх бібліотек. Найпопулярнішим прикладом є прибиральник сміття Ганса Бема [7].

3.2 Java

Почати опис управління пам’яттю потрібно із організації пам’яті віртуальної машини Java або JVM. [8]

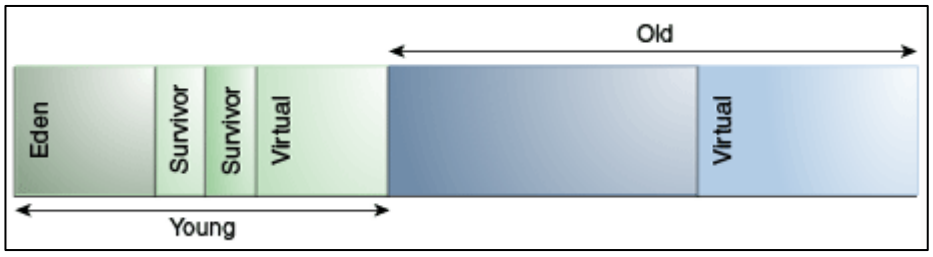


Рисунок 4 - Представлення купи у мові програмування Java

Як видно за схеми, купу поділено на регіони. Перше поділення відбувається на молоде і старе покоління. Молоде покоління в свою чергу поділяється на Edem space та Survivor space. При чому Survivor space складається з двох підпросторів однакового розміру.

Стратегія очищення пам’яті є комбінованою. По-перше, використовується генераційний колектор. Об’єкти створюються в Edem space, ті з них, які пережили перше прибирання переходять в один з просторів Survivor space. З часом з Survivor space об’єкти буде перенесено в старий регіон.

Під час перенесення об’єктів між регіонами відбувається компактування. А в Survivor space використовується алгоритм поділу на підпростори.

За замовченням прибиральник сміття працює в окремих потоках, але можна перенести його роботу в один потік з основною програмою. Такі налаштування можна робити окремо для молодого і старого покоління.

Для покращення роботи програми, прибиральника сміття можна налаштовувати. Розмір купи можна змінити використовуючи прапорці Xms та Xmx.

```
    #increases initial size of the heap to 4096MB and maximum heap size to
6144MB
    java -jar -Xms4096M -Xmx6144M myJar.jar
```

Але ці прапорці не впливають на розмір молодого регіону. Збільшення пам'яті впливає лише на старий регіон. Для того, що змінити розмір молодого покоління використовується прапорець -XX:NewRatio.

```
    # memory ratio between old and young generation is set to 2:1
    java -jar -XX:NewRatio=2:1 pathToJarFile.jar
```

На основі описаного вище алгоритму поділу купи на регіони, мова програмування Java надає декілька алгоритмів прибирання сміття, які можна задати під час запуску програми.

По-перше, прибирання може виконуватись в одному потоці, для цього використовується Serial GC. Однопотоковий алгоритм є більш повільним, але для невеликих програм накладні витрати, пов'язані зі створенням потоків можуть виявитись надлишковими. В такому разі варто використати однопотоковий алгоритм. Для цього під час запуску програми потрібно використати -XX:+UseSerialGC.

```
    java -jar -XX:+UseSerialGC pathToJarFile.jar
```

По-друге, існує багатопоточний алгоритм, який називається Parallel GC. Він розподіляє роботу маркування та прибирання на декілька потоків для пришвидшення процесу прибирання. Характеризується невеликими паузами основного потоку та кращими загальними часовими характеристиками. Для використання багатопоточного прибирання сміття потрібно додати

-XX:+UseParallelGC.

```
    java -jar -XX:+UseParallelGC pathToJarFile.jar
```

По-третє, з JDK 7 версії з’явився доступ до Garbage First GC. Він був створений щоб надавати баланс між загальною швидкістю роботи і невеликими паузами основного потоку. Фаза маркування відбувається в окремих потоках, на цій фазі визначається в яких регіонах знаходиться найбільша кількість недосяжних об’єктів. Першими прибираються саме ці регіони, звідси і назва – Garbage First. Цей алгоритм варто застосовувати на купах середнього і великого розміру. Для його використання потрібно додати -XX:+UseG1GC.

```
java -jar -XX:+UseG1GC pathToJarFile.jar
```

4 Практичне порівняння використання пам’яті у програмах написаних мовами C++ та Java

4.1 Опис програми

Розглянемо використання пам’яті у програмі написаній мовами C++ та Java. Для розгляду вирішено було реалізувати побудову інверсного індексу на колекції файлів великого розміру. Для цього програма ітерується по файлам у директорії, розбиває вміст файлів на окремі слова і зберігає у внутрішній структурі відповідність слово – список файлів та позиці, в яких воно зустрічалося. Розмір колекції складає 755 МБ.

Програми мовами C++ та Java було розширено графічним інтерфейсом з використанням Qt та JavaFX відповідно. Таке рішення обумовлене не тільки бажанням спростити використання, але і особливостями роботи з пам’яттю, які з’являються під час роботи з графічним інтерфейсом.

Для аналізу використаної пам’яті у реалізації мовою C++ було використано вбудовану систему профілювання Microsoft Visual Studio, а для аналогічної програми мовою Java – Jet Brain`s IntelliJ IDEA з системою для моніторингу використання пам’яті IntelliJ Profiler та можливості Java Development Kit для отримання даних щодо роботи прибиральника сміття, розміру купи та нативних регіонів. Для аналізу пам’яті, яку займає процес, було використано Моніторинг ресурсів операційної системи Windows.

Спочатку було розроблено консольний застосунок, який проводив індексацію колекції файлів. Було проведено аналіз використання ресурсів цієї реалізації. Ця робота виділена в окремий етап з декількох причин. По-перше, консольний застосунок має специфічні вимоги до роботи з пам’яттю в порівнянні з реалізацією, що включає графічний інтерфейс. По-друге, модуль, який реалізовано в консольному застосунку є найбільш ресурсоємною частиною в версії з графічним інтерфейсом. Для консольного застосунку наведено аналіз двох реалізацій – простої та оптимізованої.

Наступним етапом консольний застосунок розширився графічним інтерфейсом. Для порівняння використання ресурсів в мовах з ручним і автоматичним керуванням пам’яттю, графічний інтерфейс додано в реалізації обидвома мовами. Проведено аналіз впливу роботи прибиральника сміття на чутливість (responsiveness) інтерфейсу.

4.2 Проста реалізація

Проста реалізація зберігала у словнику відповідність слово – список назв файлів, в яких воно зустрічається. Такі відповідності зберігалися у std::unordered_map в C++, та HashMap в Java. Це рішення було прийняте оскільки найчастішими операціями є вставка нового елементу та пошук за ключем.

Така реалізація виявилась дуже поганою з точки зору використання пам’яті в C++.

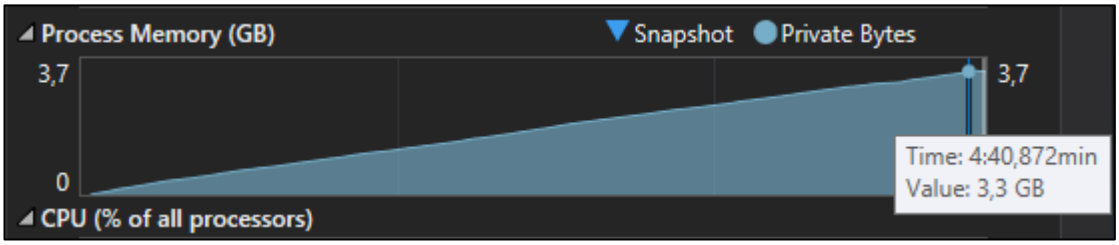


Рисунок 5 - Використання пам'яті першої реалізації мовою C++

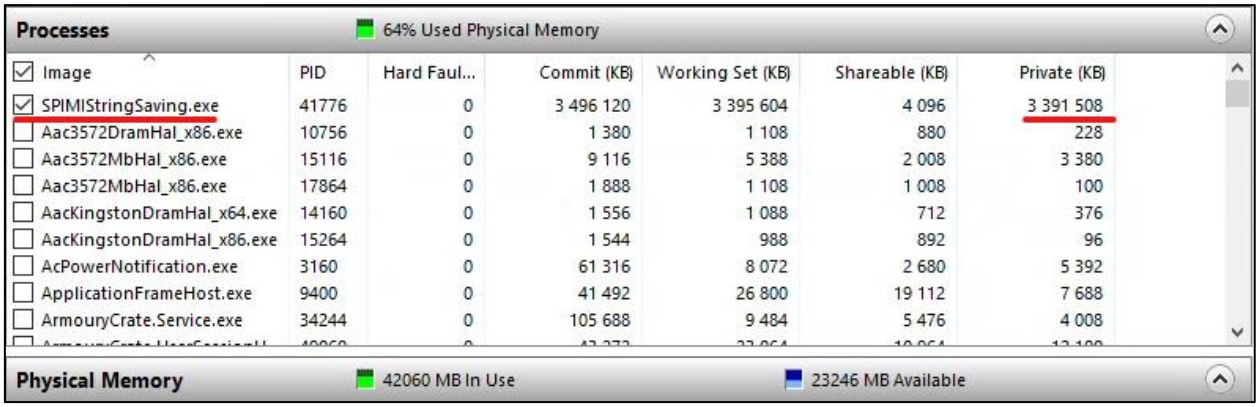


Рисунок 6 - Моніторинг ресурсів першої реалізації мовою C++

Загалом вона зайняла 3,3 Гб у купі. Моніторинг ресурсів показує, що в кінці виконання програми процес займав 3 391 508 Кб. Такий результат є очікуваним, адже файли зберігалися у вигляді стрічки із назвою. Такі назви дублювалися для кожного нового слова. Тому цей алгоритм є неефективним з точки зору використання пам’яті.

Рисунок 7 - Використання пам'яті першою реалізації мовою Java

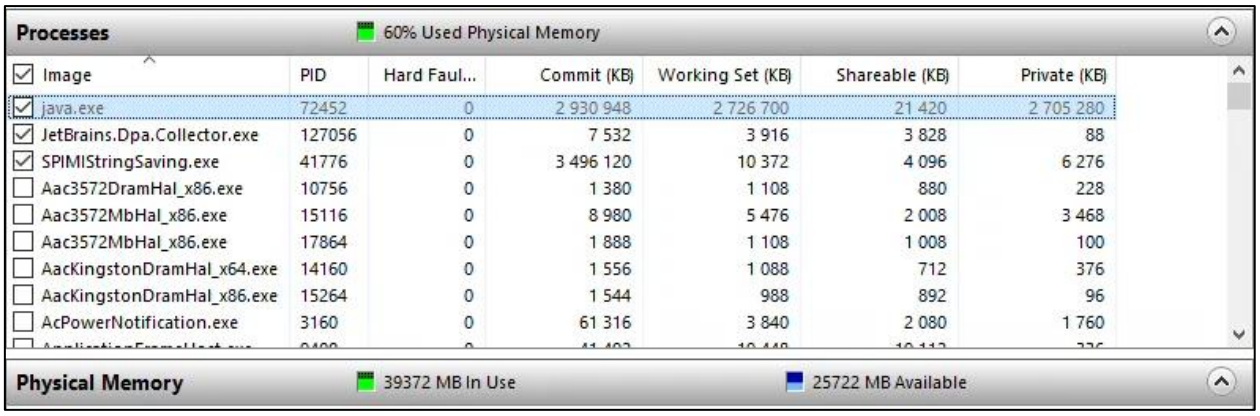


Рисунок 8 - Моніторинг ресурсів першої реалізації мовою Java

У Java використання пам’яті купи виглядало наступним чином. На графіку видно зубці. Найбільший з них сягає 2267Мб. Моніторинг ресурсів показує, що в кінці роботи програми було використано 2 705 280 Кб пам’яті.

З таких даних можна зробити декілька висновків. По-перше, неефективно написані програми мовою C++ цілком можуть займати стільки ж або навіть більше пам’яті, ніж програми на Java. По-друге, можна помітити, що кількість використаної пам’яті процесом у Java дещо перебільшує розмір купи. Різниця пов’язана з наявністю у Java віртуальної машини, яка займає додаткове місце. Для того, щоб отримати статистику використання пам’яті віртуальною машиною, було використано прапорець XX:NativeMemoryTracking= summary під час запуску програми. В результаті було отриману інформацію, наведену у лістингу.

Важливо зауважити, що дані з моніторингу процесів і лістинг використання пам’яті зібрані під час одного запуску програми. Таким чином видно, що розмір купи складає 2 695 Мб, загальний розмір – 2 913 Мб. Розгляньмо, для чого використовувалась різниця.

- Class використано 348 Кб для збереження даних про 1162 класи. Ця пам’ять потрібна для підтримки системи рефлексії
- Code використано 8 372КБ, ця пам'ять використовувалась для підтримки роботи JIT(Just in Time) компілятора
- GC використано 189 515Кб, цю пам’ять витрачено для підтримки внутрішніх структур прибиральника сміття
- GCCardSet використано 1 099Кб. Тут зберігається інформація про ділянки пам’яті, які були модифіковані після останнього прибирання сміття для оптимізації цього процесу.
- Symbol використано 1 297Кб. Тут зберігаються строкові літерали для оптимізації роботи зі стрічками.

4.3 Оптимізована реалізація

Для оптимізації використання пам’яті було вирішено зберігати назви файлів в окремому масиві, а у відповідність словам ставити індекс файлу в масиві. Така реалізація мовою C++ значно покращила показники використання пам’яті. Тепер піковим розміром купи стало 983 Мб.

Рисунок 9 - Використання пам'яті другої реалізації мовою C++

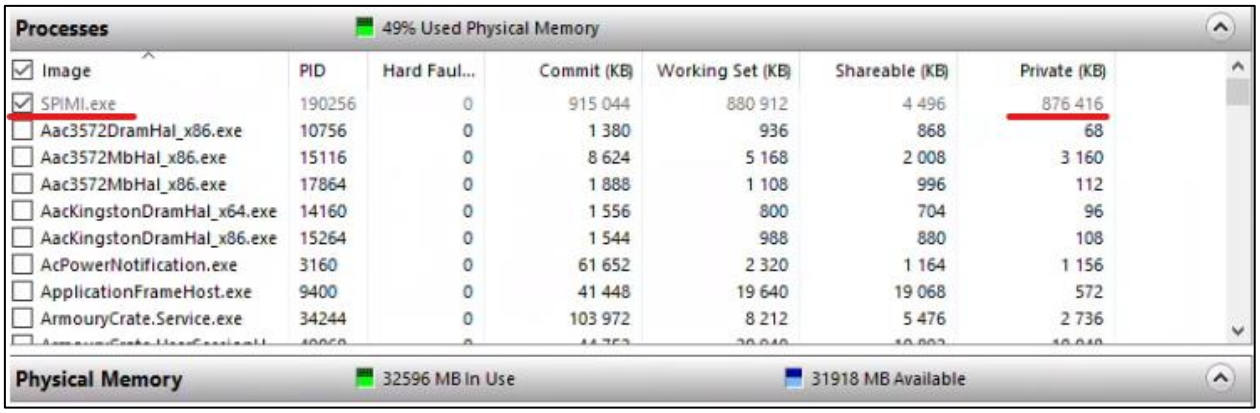


Рисунок 10 - Моніторинг ресурсів другої реалізації мовою C++

Натомість у Java така реалізація дещо погіршила показники. Піковим розміром купи стало значення 2294 Мб. Такий результат може бути пов’язаним із тим, що віртуальна машина підтримує внутрішнє сховище використаних стрічок. Тому оптимізація, зроблена на цьому кроці, уже працювала на попередньому кроці для Java. Зберігаючи колекцію Integer замість колекції String ми лише збільшили використання пам’яті.

Рисунок 11 - Використання пам'яті другої реалізації мовою Java

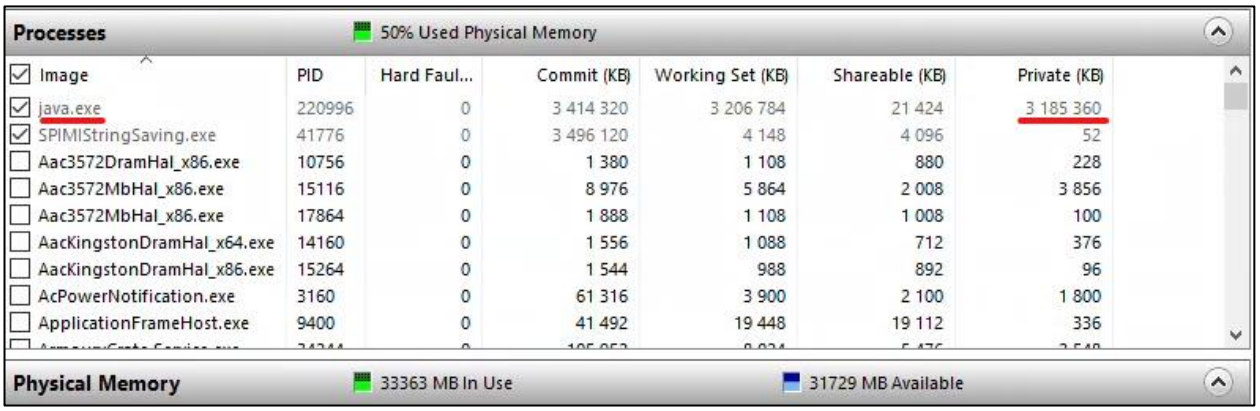


Рисунок 12 - Моніторинг ресурсів другої реалізації мовою Java

Розмір купи виріс з 2 695Кб до 3 162Кб. Порівняння характеристик можна переглянути у таблиці 1.

	First Run	Second Run
Total	2 913 274	3 396 216
Java Heap	2 695 168	3 162 112
Class	348	346
Code	8 372	8 365
GC	189 515	205 575
GCCardSet	1 099	128
Symbol	1 297	1 297

Таблиця 1- Використання пам'яті програмою написаною мовою Java

4.4 Порівняння швидкодії регулярних виразів в реалізації бібліотек Boost та STL

Під час порівняння першої реалізації програми різними мовами, можна помітити вразливу різницю у часі виконання. Реалізація мовою C++ займала 272 секунди, а аналогічний варіант Java 40 секунд. Оптимізована реалізація теж показала значну різницю – 36 секунд проти 230 секунд.

Після отримання таких результатів був проведений аналіз часу витраченого на основні операції програми. Так, ними можна назвати читання файлів, розбиття їх вмісту на слова і збереження пар слово – файл у внутрішню структуру даних. За результатами виявилось що найбільше часу займає розбиття тексту на слова.

Для реалізації цієї задачі було використано регулярні вирази з бібліотеки STL. У спробі зменшити кількість часу на розбиття тексту реалізація зі стандартної бібліотеки була замінена на Boost. Це призвело до вражаючих результатів.

	File reading	Text Parsing	Saving Words
STL	5 560 мс	194 148 мс	31 377 мс
Boost	6 329 мс	27 051 мс	26 782 мс

Таблиця 2 - Заміри часу, витраченого на основні операції оптимізованої реалізації мовою C++

Реалізація Boost дала прискорення приблизно в 7 разів. Така ситуація трапилась через те, що стандартна бібліотека намагається підтримувати стабільність ABI [9]. Натомість Boost не надає таких гарантій, тому він може впроваджувати оптимізації.

Треба також зауважити, що для отримання даних з таблиці код був розширений системою логування та збереження часу виконання, яка збільшує

загальний час виконання. Тому сума цього часу перевищує заміри наведені вище.

Тож, після заміни регулярних виразів з STL на Boost час виконання зменшився до 50 секунд.

4.5 Розширення програм графічним інтерфейсом

Наступним етапом роботи було додавання графічного інтерфейсу. Він підтримує створення індексу колекції, яку задає користувач, пошук файлі колекції на основі створеного індексу. Для обробки файлів використовувалась оптимізована версія, описана вище. Її роботу винесено в окремий потік для того, щоб не блокувати інтерфейс програми під час створення індексу, оскільки індексація великих колекцій може займати десятки секунд.

Програма складається з головного екрану.

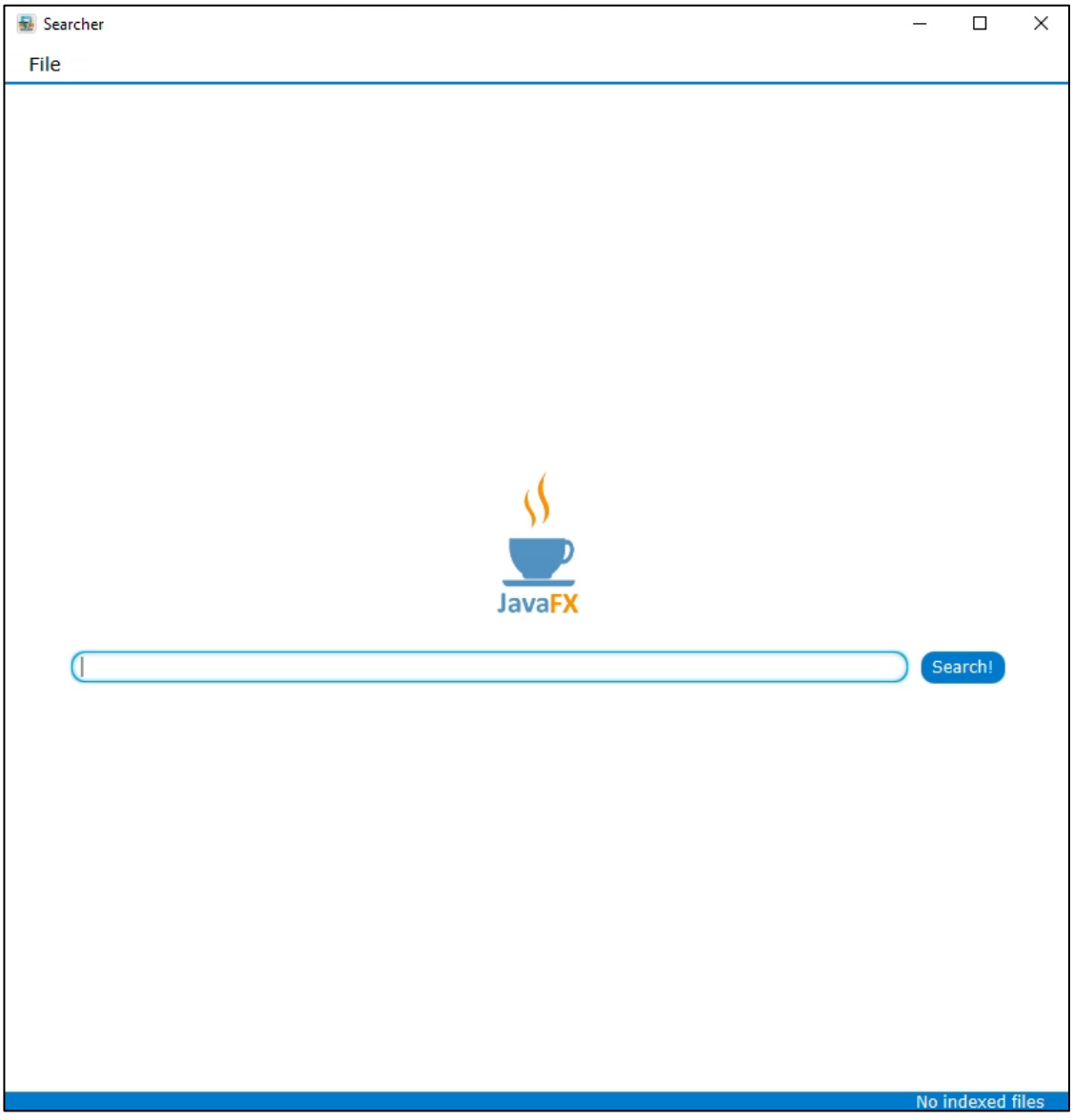


Рисунок 13 - Основний екран програми мовою Java

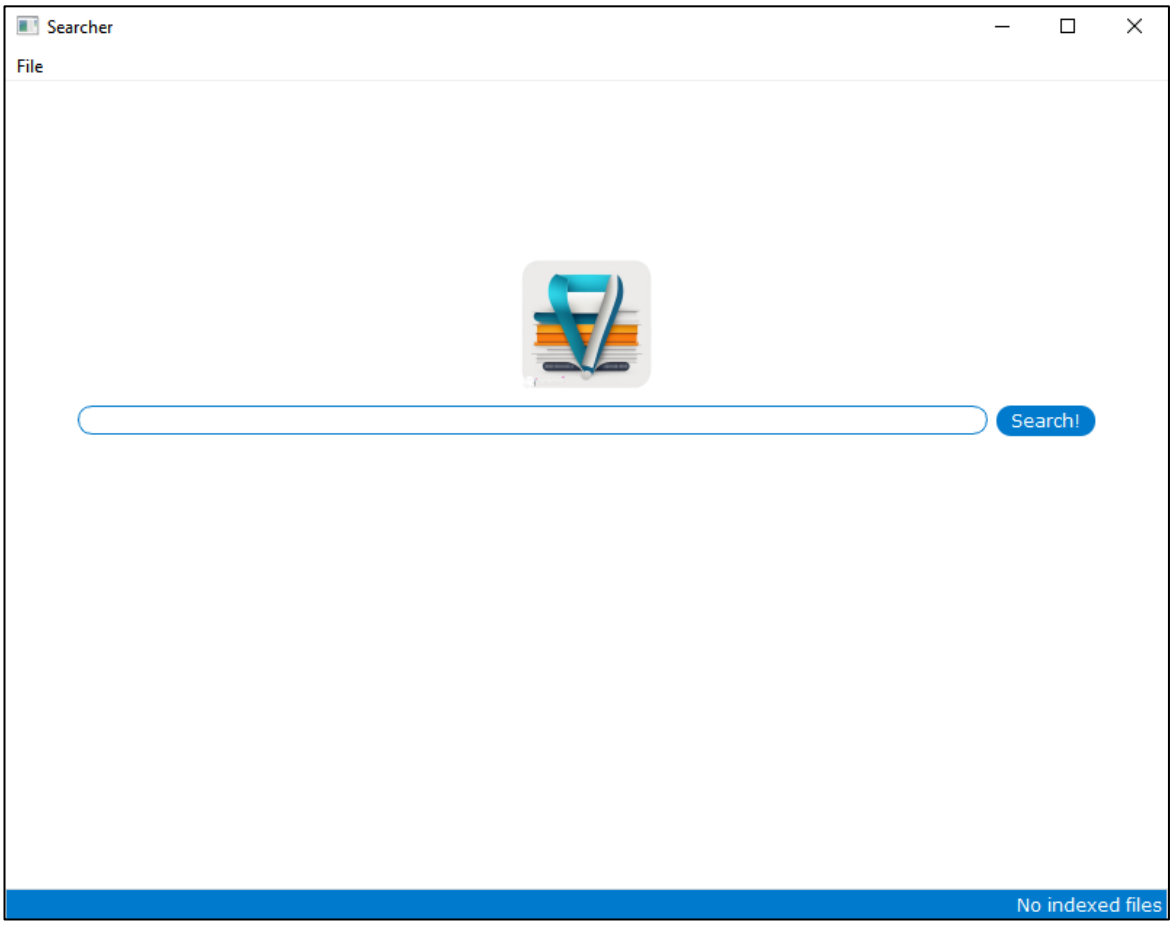


Рисунок 14 - Основний екран програми мовою C++

Для створення колекції необхідно перейти в меню File, обрати опцію Index Collection та вибрати папку, яку потрібно проіндексувати.

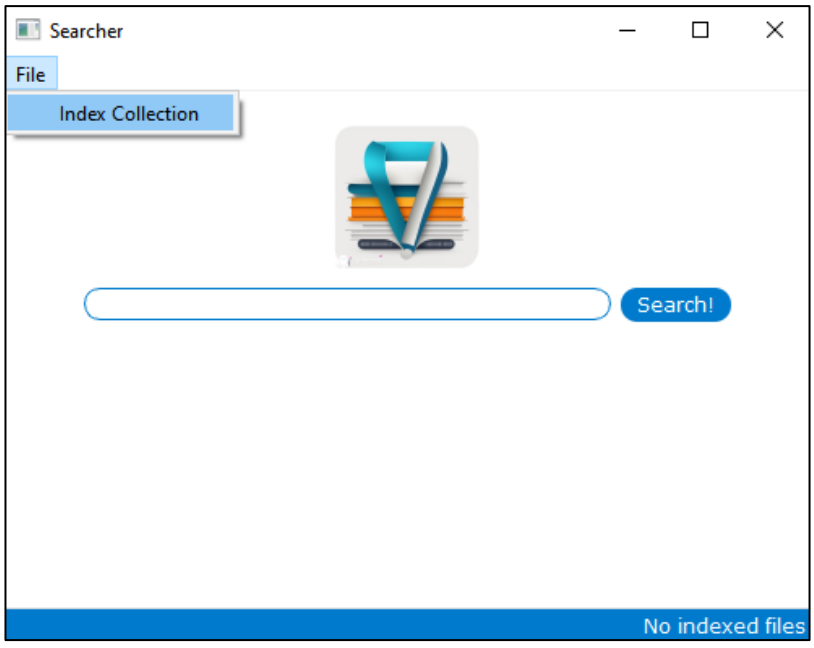


Рисунок 15 - Опція індексації колекції мовою C++

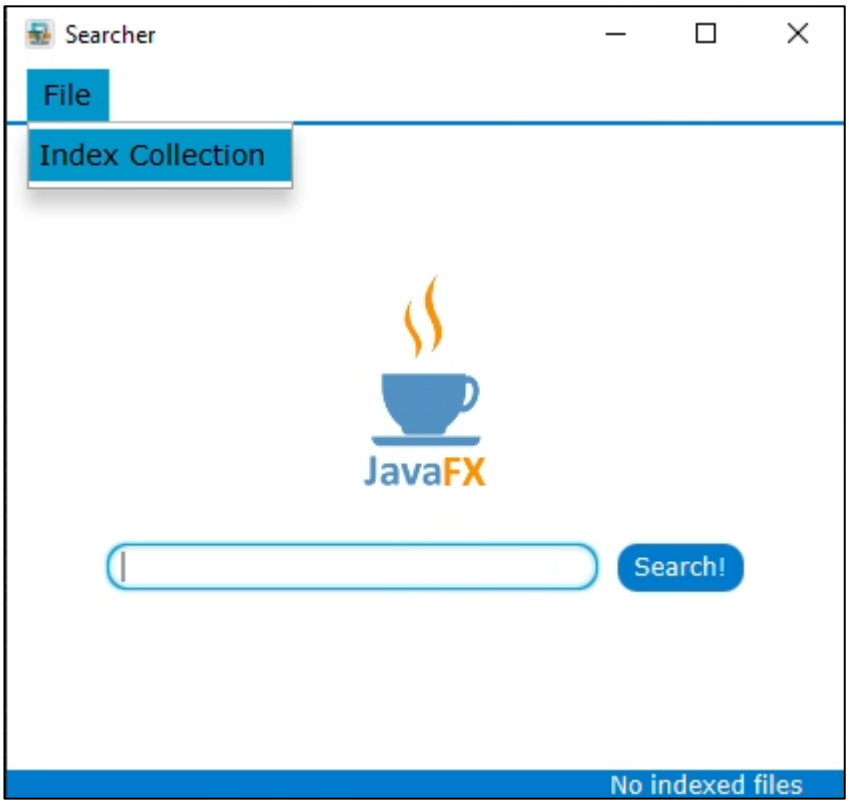


Рисунок 16 - Опція індексації колекції мовою Java

Під час створення індексу в графічному інтерфейсі з’являється індикатор завантаження. Після закінчення цього процесу, користувач бачить яку саму колекцію було створено в нижній частині екрану. На цьому етапі можливо робити пошук документів.

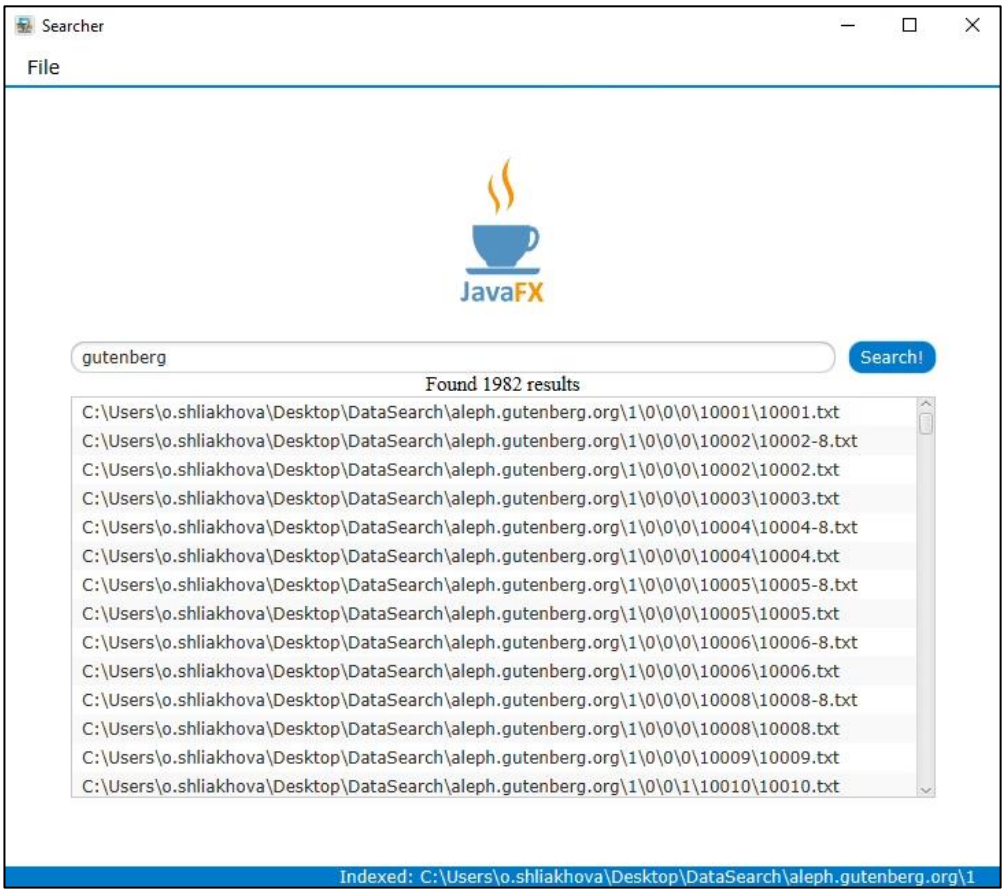


Рисунок 17 - Відображення результатів пошуку в програмі мовою Java

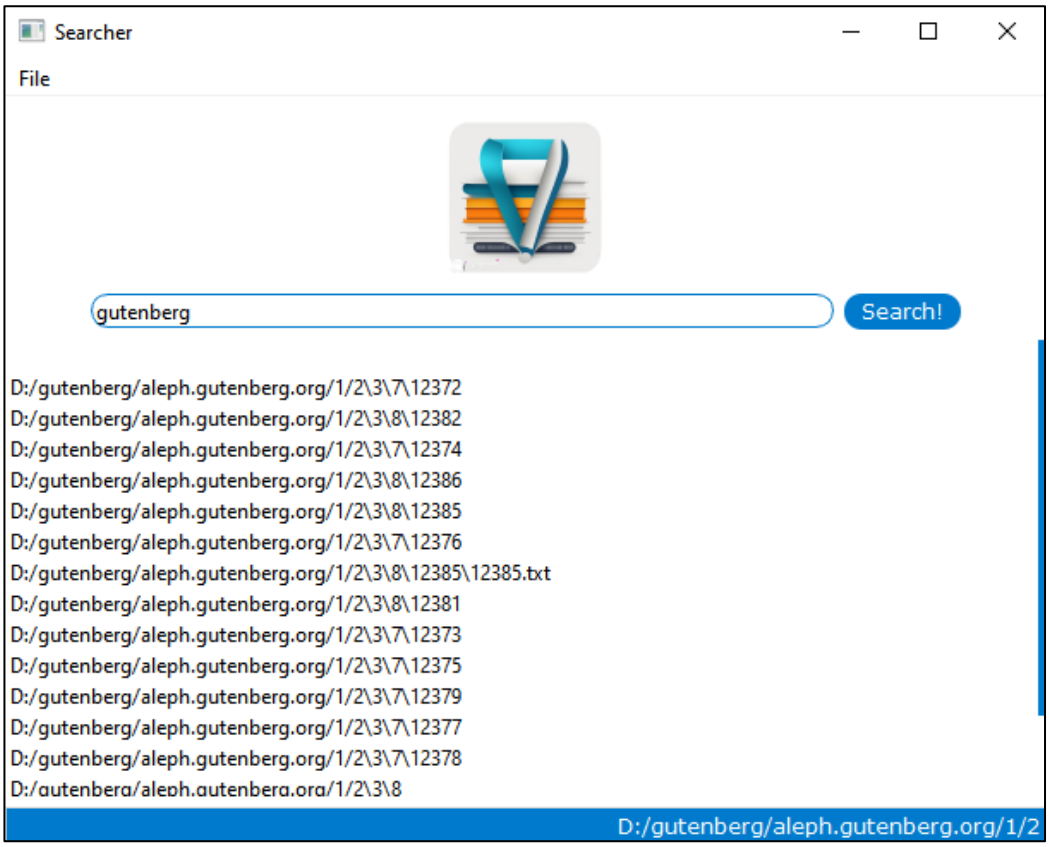


Рисунок 18 - Відображення результатів пошуку в програмі мовою C++

Під час роботи з графічним інтерфейсом важливо зберігати чутливість (responsiveness) інтерфейсу. Робота прибиральника сміття додає до роботи застосунку паузи, необхідні для підрахунку посилань, видалення недосяжних об’єктів, компактування (в разі використовування компактувального прибиральника сміття). Це негативно впливає на досвід користувача. Тож, під час вибору прибиральника сміття основним критерієм мають бути якомога менші паузи основного потоку.

4.6 Порівняння роботи прибиральників сміття, які надає мова програмування Java

Як зазначалося вище, мова програмування Java надає користувачу можливість обирати різні стратегії прибирання сміття. Серед них Serial GC, Parallel GC та G1 GC [10]. У цьому розділі будуть наведені статистичні дані, зібрані у ході виконання програми з використанням кожного з наведених прибиральників сміття.

4.6.1 Serial Garbage Collector

Однопотоковий алгоритм прибирання сміття зазвичай використовують для програм, в яких накладні витрати на створення потоків, є завеликими відносно роботи основної програми. Під таку характеристику підпадають невеликі консольні застосунки.

По-перше, потрібно зазначити загальний час індексації колекцій різних розмірів, затримки основного потоку, пов’язані з роботою прибиральника сміття, а також об’єм використаної їм пам’яті, загальний об’єм пам’яті та «корисний» об’єм(що використовувався власне програмою). Для перегляду статистики використано утиліту jstat та систему профілювання Inteliji Profiler.

Розмір колекції	Кількість прибирань в молодому поколінні	Кількість прибирань в старому поколінні	Найдовша затримка основного потоку	Загальна затримка основного потоку через прибирання сміття	Час виконання програми
755 Мб	50	5	5095 мс	32424 мс (48%)	67348 мс
511 Мб	44	4	3301 мс	20044 мс (42%)	47584 мс
235 Мб	30	3	1632 мс	9155 мс (42%)	21406 мс
73 Мб	12	1	194 мс	1828 мс (33%)	5498 мс
41 Мб	7	1	172 мс	887 мс (29%)	3045 мс
8 Мб	1	1	132 мс	165 мс (27%)	607 мс

Таблиця 3 - Часові характеристики побудови індексу з використанням Serial GC

Наведемо також статистику використання пам’яті застосунком загалом і прибиральником сміття. Для збору статистики використано утиліту native memory tracking та Inteliji Profiler.

Розмір колекції	Розмір купи	Розмір структур GC	Розмір процесу
755 Мб	8804280 Кб	28742 Кб (0,3%)	8930031 Кб
511 Мб	5232896 Кб	17098 Кб(0,3%)	5311404 Кб
235 Мб	2962284 Кб	9722 Кб(0,3%)	3033632 Кб
73 Мб	1048640 Кб	3486 Кб (0,3%)	1150650 Кб
41 Мб	1048640 Кб	3486 Кб (0,3%)	1153857 Кб
8 Мб	1048640 Кб	3486 Кб (0,3%)	1161429 Кб

Таблиця 4 - Використання пам'яті програми для побудови індексу із застосуванням Serial GC

4.6.2 Parallel Garbage Collector

Цей прибиральник сміття використовує багатопотокові системи для пришвидшення роботи. Нижче наведено статистику швидкодії та використання пам’яті застосунку, який використовує Parallel Garbage Collector.

Розмір колекції	Кількість прибирань в молодому поколінні	Кількість прибирань в старому поколінні	Найдовша затримка основного потоку	Загальна затримка основного потоку через прибирання сміття	Час виконання програми
755 Мб	18	6	956 мс	5 564 мс (12%)	44135 мс
511 Мб	14	4	574 мс	2 626 мс (9%)	28204 мс
235 Мб	11	3	372 мс	1389 мс (10%)	13287 мс
73 Мб	8	1	76 мс	212 мс (5%)	3738 мс
41 Мб	6	1	37 мс	143 мс (6%)	2130 мс
8 Мб	2	1	12 мс	32 мс (6%)	487 мс

Таблиця 5 - Часові характеристики побудови індексу з використанням Parallel GC

Розмір колекції	Розмір купи	Розмір структур GC	Розмір процесу
755 Мб	9938432 Кб	659387 Кб (6%)	10690986 Кб
511 Мб	8477184 Кб	653683 Кб(7%)	9222769 Кб
235 Мб	7091712 Кб	649015 Кб(8%)	7834181 Кб
73 Мб	3254272 Кб	639071 Кб (16%)	3989796 Кб
41 Мб	2071552 Кб	636759 Кб (22%)	2820015 Кб
8 Мб	1048576 Кб	634763 Кб (35%)	1799336 Кб

Таблиця 6 - Використання пам'яті програми для побудови індексу із застосуванням Parallel GC

У порівнянні з попереднім алгоритмом, затримки головного потоку значно зменшилися. Загальний час виконання також є меншим під час використання паралельного алгоритму навіть в колекціях малого розміру.

Але однопотоковий алгоритм показав кращі результати у використанні пам’яті, адже для підтримки прибиральника сміття було використано 0,4%-0,5% пам’яті, відносно паралельного. Також, накладні витрати залишалися стабільними на рівні 0,3% в однопотоковому алгоритмі, в багатопотоковому ж вони поступово зростали від 6% від загального розміру в колекції 755 Мб до 35% в колекції 8 Мб. Загальний розмір програми також є кращім у варіанті з однопотоковим прибиральником. Найбільшою є різниця для колекції 73 Мб де програми відрізнялись за розміром у 3,4 рази. Загалом можна сказати, що однопотоковий алгоритм є більш ефективним в контексті використання пам’яті.

4.6.3 Garbage First Garbage Collector

Останнім прибиральником сміття, який буде розглянуто, є Garbage First. Він гарантує невеликі зупинки основного потоку а також аналіз поведінки застосунку для очищення найбільш «засмічених» регіонів пам’яті.

Розмір колекції	Кількість прибирань в молодому поколінні	Кількість прибирань в старому поколінні	Найдовша затримка основного потоку	Загальна затримка основного потоку через прибирання сміття	Час виконання програми
755 Мб	41	4	746 мс	2166 мс (4%)	45380 мс
511 Мб	37	4	725 мс	1405 мс (4%)	31595 мс
235 Мб	25	1	76 мс	606 мс (4%)	13576 мс
73 Мб	16	1	31 мс	231 мс (5%)	4279 мс
41 Мб	13	1	27 мс	114 мс (4%)	2403 мс
8 Мб	8	1	8 мс	41 мс (7%)	536 мс

Таблиця 7 - Часові характеристики побудови індексу з використанням G1 GC

Розмір колекції	Розмір купи	Розмір структур GC	Розмір процесу
755 Мб	9510912 Кб	444592 Кб(4%)	10055796 Кб
511 Мб	8413184 Кб	402217 Кб (4%)	8924842 Кб
235 Мб	3629056 Кб	220636 Кб(5%)	3946648 Кб
73 Мб	1728512 Кб	148442 Кб (7%)	1983853 Кб
41 Мб	1130496 Кб	126109 Кб (9%)	1357615 Кб
8 Мб	1048576 Кб	122223 Кб(9%)	1251778 Кб

Таблиця 8 - Використання пам'яті програми для побудови індексу із застосуванням G1 GC

Цей алгоритм робить більшу кількість пауз в порівнянні з багатопоточним алгоритмом, але за рахунок цього зменшується затримка основного потоку. Загалом, Garbage First сприяв найнижчим паузам основного потоку з усіх трьох алгоритмів. При цьому використання пам’яті також є нижчим в порівнянні з багатопоточним алгоритмом, але все ще менш ефективне за однопоточний алгоритм.

4.6.1 Висновок після порівнянь роботи прибиральників сміття мови програмування Java

У цій частині роботи було наведення порівняння ефективності роботи програми з використанням трьох прибиральників сміття: Serial GC, Parallel GC та G1 GC.

Найефективнішим з точки зору використання пам’яті виявився Serial GC. Пам’ять, яка пішла на підтримку прибирання сміття складала 0,3% від загального розміру програми. Тож, для створення індексу колекції, яка за розміром наближена або перевищує оперативну пам’ять пристрою, краще використовувати саме цей алгоритм.

Але неможливо не звернути увагу на часові показники, продемонстровані роботою цього колектора. Вони виявилися найгіршими з усіх трьох. Великі паузи, які додаються до роботи основного потоку, не дозволяють використовувати цей алгоритм для застосунків з графічним інтерфейсом, адже користувачу будуть помітні паузи, що буде негативно впливати на враження від роботи з програмою. Під час спроби взаємодії з програмою під час індексації, яка створює велику кількість тимчасових об’єктів, з якими працює прибиральник, помітні зависання основного потоку під час циклів прибирання(сама індексація відбувається в окремому потоці). Особливо помітними є зависання, які відбуваються під час основних циклів прибирання, які зачіпають молоде і старе покоління.

Тож, Serial GC може бути використаним в консольному варіанті застосунку, але дуже не бажано використовувати його в програмі з графічним інтерфейсом.

Що стосується Parallel GC, було продемонстровано непогані показники по часу виконання, вони значно перевищували Serial GC у більшості випадків, та були на одному рівні з G1 GC. Паузи основного потоку були значно кращими за Serial GC, але гіршими за G1 GC. Використання пам’яті в цьому

алгоритмі виявилося найменш ефективним. Але показники все ще є непоганими, цей алгоритм можна використати як для консольного варіанту, так і для застосунку з графічним інтерфейсом. Невеликі паузи основного потоку під час індексації були помітними, але не відчувалися критичними, особливо в порівнянні з Serial GC.

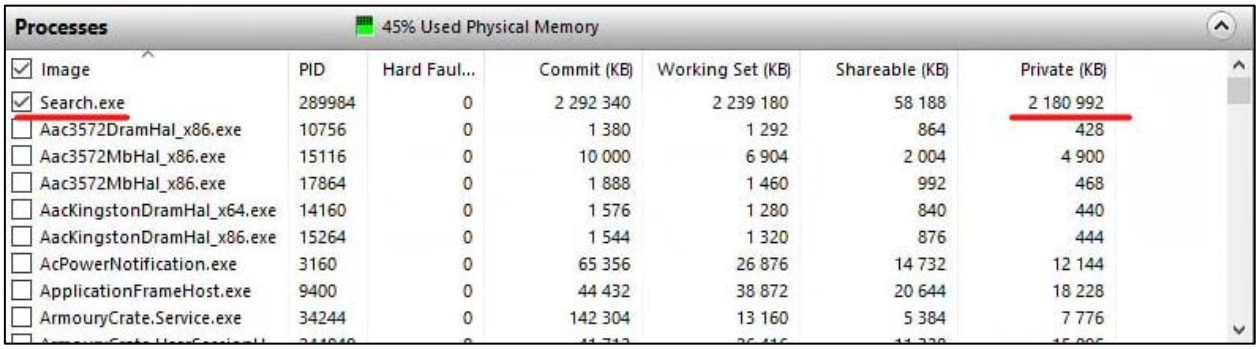
Останнім у порівнянні був G1 GC. Він продемонстрував найнижчі паузи основного потоку, що робить його найкращим алгоритмом для використання у застосунках з графічним інтерфейсом. Використання пам’яті є нижчим у порівнянні з Serial GC, тож алгоритм також може бути використаним для консольних застосунків.

Підсумовуючи, для застосунку з графічним інтерфейсом варто обрати G1 GC, а для консольного застосунку можна обрати між G1 GC та Serial GC в залежності від потреб. Для більш швидкої обробки за рахунок вищого використання пам’яті варто використовувати G1 GC, для економії пам’яті краще застосувати Serial GC, але виконання буде займати більшу кількість часу.

4.7 Аналіз ефективності застосунку з графічним інтерфейсом мовою C++

Для написання застосунку з графічним інтерфейсом мовою C++ було використано фреймворк Qt. Він використовує ручне управління пам’яттю. Перший висновок, який можна зробити під час роботи з програмою, це відсутність пауз, як в випадку з реалізацією мовою Java навіть під час «важкої» операції створення індексу.

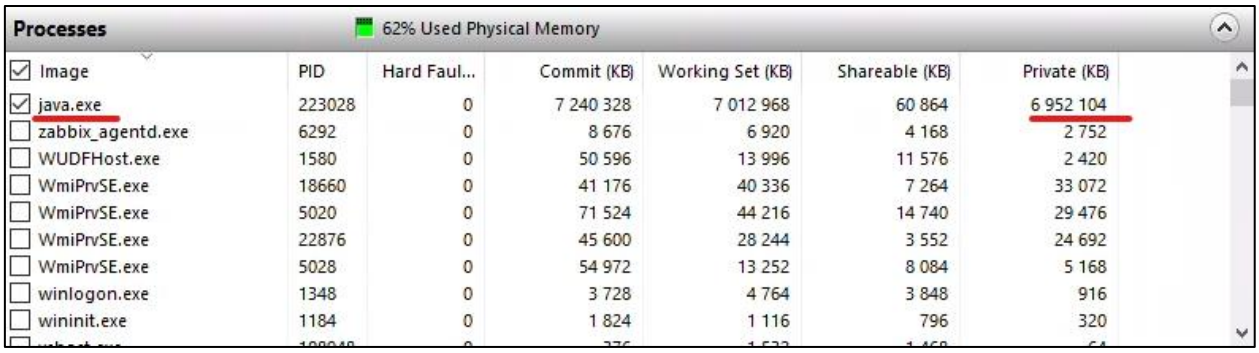
Як видно з малюнку, програма займає 2 180 992 Кб після індексації колекції розміром 755 Мб. Для порівняння, найнижчим показником з використання пам’яті у Java після індексації колекції того самого розміру було 6 952 104 Кб.



Processes — 45% Used Physical Memory

Image	PID	Hard Faul...	Commit (KB)	Working Set (KB)	Shareable (KB)	Private (KB)
Search.exe	289984	0	2 292 340	2 239 180	58 188	2 180 992
Aac3572DramHal_x86.exe	10756	0	1 380	1 292	864	428
Aac3572MbHal_x86.exe	15116	0	10 000	6 904	2 004	4 900
Aac3572MbHal_x86.exe	17864	0	1 888	1 460	992	468
AacKingstonDramHal_x64.exe	14160	0	1 576	1 280	840	440
AacKingstonDramHal_x86.exe	15264	0	1 544	1 320	876	444
AcPowerNotification.exe	3160	0	65 356	26 876	14 732	12 144
ApplicationFrameHost.exe	9400	0	44 432	38 872	20 644	18 228
ArmouryCrate.Service.exe	34244	0	142 304	13 160	5 384	7 776

Рисунок 19 - Використання ресурсів застосунку з графічним інтерфейсом мовою C++



Processes — 62% Used Physical Memory

Image	PID	Hard Faul...	Commit (KB)	Working Set (KB)	Shareable (KB)	Private (KB)
java.exe	223028	0	7 240 328	7 012 968	60 864	6 952 104
zabbix_agentd.exe	6292	0	8 676	6 920	4 168	2 752
WUDFHost.exe	1580	0	50 596	13 996	11 576	2 420
WmiPrvSE.exe	18660	0	41 176	40 336	7 264	33 072
WmiPrvSE.exe	5020	0	71 524	44 216	14 740	29 476
WmiPrvSE.exe	22876	0	45 600	28 244	3 552	24 692
WmiPrvSE.exe	5028	0	54 972	13 252	8 084	5 168
winlogon.exe	1348	0	3 728	4 764	3 848	916
wininit.exe	1184	0	1 824	1 116	796	320

Рисунок 20 - Використання ресурсів застосунку з графічним інтерфейсом мовою Java

Висновки

У результаті роботи було проведено дослідження використання пам’яті двох реалізацій алгоритмів для побудови інверсного індексу мовами C++ та Java. Найбільш ефективну з двох реалізацій було розширено графічним інтерфейсом. Отриману програму було досліджено з точки зору ефективності використання прибиральників сміття, які надає мова програмування Java. Після аналізу результатів можна зробити декілька висновків.

Результати роботи полягають у формулюванні наведених нижче рекомендацій, стосовних використання мов програмування і наявних бібліотек для створення програм, ємних у використанні пам’яті, зокрема при графічному інтерфейсі.

По-перше, оптимізації, які проводить мова програмування Java виявились ефективними для задач зберігання стрічкових літералів, що дублюються. Цей факт допоміг простій реалізації бути більш ефективною з точки зору використання пам’яті за аналогічну реалізацію мовою C++.

По-друге, оптимізації аналогічні тим, що є у Java, написані мовою C++ для вирішення конкретної задачі виявились більш ефективними. В оптимізованій версії програми самостійне кешування стрічок значно знизило розмір програми та зробило реалізацію мовою C++ більш ефективною з точки зору використання пам’яті.

По-третє, пам’ять, яку займає віртуальна машина Java, складала 8% у першій реалізації та 6% у другій реалізації. Тож, для задач, які оперують на купах великого розміру(у випадку конкретної задачі 2 695 168 KB та 3 162 112Кб) цими витратами можна нехтувати.

По-четверте, для реалізації частини програми, яка відповідає за розбиття тексту на слова мовою C++ краще використовувати регулярні вирази з

бібліотеки Boost, ніж з STL. Різниця у часі роботи цих двох алгоритмів склала 86%.

По-п’яте, для застосунків з графічним інтерфейсом, написаних мовою Java, краще використовувати G1 Garbage Collector, оскільки він додає найкоротші паузи у роботу застосунку в порівнянні з Serial GC та Parallel GC.

По-шосте, використання мов програмування з ручним управлінням пам’яттю для розробки програм з графічним інтерфейсом має перевагу відсутності пауз основного потоку, які викликає робота прибиральника сміття в мовах з автоматичним управлінням пам’яттю.

Лістинги коду

Лістинг 1- ручне управління пам'яттю в мові програмування C 12

Лістинг 2 - Ручне управління пам'яттю в C++ 12

Лістинг 3 - Приклад витоку пам'яті в C++ 13

Лістинг 4 - Приклад неочевидного витоку пам'яті 14

Лістинг 5 - Приклад реалізації розумного указника 15

Лістинг 6 - Приклад деревовидної структури з використанням розумних указників, який може призвести до виняткової ситуації в деструкторі 18

Лістинг 7 - Приклад знаходження досяжних об'єктів мовою програмування Java 21

Лістинг 8 - Приклад знаходження досяжних об'єктів мовою програмування C++ 22

Лістинг 9 - Приклад реалізації Two-Finger Collector 25

Список ілюстрацій

Рисунок 1 - Структура виконуваного файлу операційної системи Windows 10
Рисунок 2 - Схема циклічних посилань 16
Рисунок 3- Схематичне зображення об'єктів в купі 20
Рисунок 4 - Представлення купи у мові програмування Java 31
Рисунок 5 - Використання пам'яті першої реалізації мовою C++ 36
Рисунок 6 - Моніторинг ресурсів першої реалізації мовою C++ 36
Рисунок 7 - Використання пам'яті першою реалізації мовою Java 37
Рисунок 8 - Моніторинг ресурсів першої реалізації мовою Java 37
Рисунок 9 - Використання пам'яті другої реалізації мовою C++ 39
Рисунок 10 - Моніторинг ресурсів другої реалізації мовою C++ 39
Рисунок 11 - Використання пам'яті другої реалізації мовою Java 40
Рисунок 12 - Моніторинг ресурсів другої реалізації мовою Java 40
Рисунок 13 - Основний екран програми мовою Java 43
Рисунок 14 - Основний екран програми мовою C++ 44
Рисунок 15 - Опція індексації колекції мовою C++ 44
Рисунок 16 - Опція індексації колекції мовою Java 45
Рисунок 17 - Відображення результатів пошуку в програмі мовою Java 46
Рисунок 18 - Відображення результатів пошуку в програмі мовою C++ 46
Рисунок 19 - Використання ресурсів застосунку з графічним інтерфейсом мовою C++ 57
Рисунок 20 - Використання ресурсів застосунку з графічним інтерфейсом мовою Java 57

Список таблиць

Таблиця 1- Використання пам'яті програмою написаною мовою Java .. 40

Таблиця 2 - Заміри часу, витраченого на основні операції оптимізованої реалізації мовою C++ 41

Таблиця 3 - Часові характеристики побудови індексу з використанням Serial GC 49

Таблиця 4 - Використання пам'яті програми для побудови індексу із застосуванням Serial GC 50

Таблиця 5 - Часові характеристики побудови індексу з використанням Parallel GC 51

Таблиця 6 - Використання пам'яті програми для побудови індексу із застосуванням Parallel GC 52

Таблиця 7 - Часові характеристики побудови індексу з використанням G1 GC 53

Таблиця 8 - Використання пам'яті програми для побудови індексу із застосуванням G1 GC 54

Список джерел

[1] R. H. R. J. J. M. V. Erich Gamma, «Design Patterns: Elements of Reusable Object-Oriented Software,» в *Design Patterns: Elements of Reusable Object-Oriented Software*, 1994.

[2] B. Stroustrup, Programming: Principles and Practice Using C++, 2008 .

[3] H. Sutter, «Leak-Freedom in C++... By Default.,» 2016.

[4] A. H. M. Richard Jones, The Garbage Collection Handbook, Chapman & Hall/CRC, 2011.

[5] J. McCarthy, «Recursive Functions of Symbolic Expressions and Their Computation by Machine,» 1960.

[6] H. G. B. Jr, «List Processing in Real Time on a Serial Computer,» 1978.

[7] H.-J. Boehm, «Conservative GC Algorithmic Overview,» HP Labs, [Онлайновий]. Available: https://hboehm.info/gc/gcdescr.html.

[8] Oracle, «HotSpot Virtual Machine Garbage Collection Tuning Guide,» [Онлайновий]. Available: https://docs.oracle.com/en/java/javase/20/gctuning/garbage-collector-implementation.html#GUID-23844E39-7499-400C-A579-032B68E53073.

[9] CppCast, *The C++ ABI*, 2019.

[10] M. J. Williams, «Java Garbage Collection Basics,» Oracle Corporation, [Онлайновий]. Available: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html.