Пловдивски университет „Паисий Хилендарски”

Факултет по математика и информатика

Катедра „Софтуерни технологии“

Дипломна работа

на тема:

Разработка и визуализация на стеков алокатор в съвременни програми

Разработил: Еньо Иванов Енев

Научен ръководител: гл. ас. д-р Христо Христов

Съдържание

5. Съвместимост с досегашни системи

6. Възможности за многоплатформеност

6.1. Линукс и Уиндоус и Мак

7. Основни структури от данни използвани за разработването на стековия алокатор.

7.1. Свързан списък

7.2. Използване на предпроцесора на компилатора за необходимите метаданни чрез макрота.

7.3 Хеш таблица със статичен размер

7.4 Кръгов двойно свързан списък

8. Интеграция

Глава Втора – Проектиране, изграждане и имплементация.

9.1 Анализ на основните етапи на една софтуерна програма

9.2 3-те основни функционалности

9.3 Анализ на "ядрото" на всяка една софтуерна програма

9.4 Защо наричаме този алокатор стеков? Какво извлякохме от анализите?

9.5 Същинската част на стековия алокатор

9.6 Имплементация на методите за временна памет.

9.7 Анализ на бърз и бавен път

Глава Трета - Дизайн и разработване на визуализацията на стековия алокатор

10.1 Дизайн на визуализацията

10.2 Разбор на имплементацията на визуализацията

Глава Четвърта - Бъдещо развитие, възможности за подобрение.

11.1 Възможност за цялостен снапшот (snapshot) на отделен интервал от време (брой кадри) на програмата.

11.2 Последствен инструмент за задълбочен анализ на вече готовия снапшот.

11.3 Възможност за ограничение на фаталните грешки.

11.4 Връщане на повече контекстна информация

Заключение

Речник:

ANSI - AMERICAN NATIONAL STANDARDS INSTITUTE

UI - user interface

imgui - Immediate-mode graphical user interface

Name mangling – Name mangling is the encoding of function and variable names into unique names so that linkers can separate common names in the language. Type names may also be mangled.

Използвани фигури:

Увод

В дипломната работа се разглежда нетрадиционен метод за мениджмънт на памет в съвременни програми, както и неговата визуализация. Поради своята гъвкавост методът подлежи на параметризация при направен съответен подбор на алокационно и блоково подравняване и големина.

Основната цел на дипломната работа е разработването на гъвкав алокатор в едно с неговото визуализиране.

Гъвкавия алокатор и визуализацията служат за подпомагане на разработчика да открива своевременно възникнали проблеми с паметта, като изтичане и фрагментация. В работата се разгледат често срещани ситуации възникващи по време на разработка на софтуерни приложения. Обърнато е внимание на подходите за разрешаване на подобни ситуации чрез разработения метод за заделяне на памет.

За постигането на основната цел на дипломната работа са поставени следните задачи:

1. Анализ на ползите и предимствата от използването на нетрадиционна схема за заделяне на памет.

2. Анализ на основните етапи на една софтуерна програма

3. Групиране на 3-те основи характеристики нужни за разработката на алокатора

4. Разработване на алокатор наподобяващ стек на програма.

5. Анализ на съществуващи визуализации.

6. Избор на подходящ дизайн за презентиране на визуализацията пред разработчици.

7. Събиране на необходимите алокационни данни чрез система за събития.

8. Разработка на системата за визуализация на стеков алокатор.

Настоящата дипломна работа съдържа увод, четири глави и заключение.

В първа глава се разглеждат начините за автоматизиран и ръчен контрол върху заделянето на памет. Обяснени са разликите, предимствата и недостатъците на тези два типа мениджмънт. Създираме необходимостта от различен подход. Определяме ползи от изграждането на нетрадиционен метод за заделяне на памет.

Във втора глава извършваме основен анализ на една софтуерна програма. Определяме 3-те основни функционалности, необходими за изграждането на нетрадиционния метод. Изготвяме анализ на работата на стека на всяка една софтуерна програма. Спрямо него се насочваме към обосновката на основния метод

на стековия алокатор. Изграждаме функциите за временна памет и правим анализ на бързодействието на алокатора в различни ситуации.

В трета глава изработваме дизайн на визуализацията подходящ за потенциалната аудитория използваща този софтуер. Определяме основните функционалности на графичния интерфейс, необходими за изграждането на визуализацията. Имплементираме структури съдържащи обзорни данни за паметта. Създаваме метод за събиране на тези данни и статистики. Обхождаме събитията получени от стековия алокатор. Анализираме цялата информация и имплементираме визуализацията в реално време.

В четвърта глава засягаме възможностите за разширение и подобрение на стековия алокатор и неговата визуализация. Разглеждаме конкретни варианти за изграждане на статичен изглед (снапшот) на програмата. Дефинираме помощни функции, чрез които да улесним употребата на стековия алокатор в повечето програмни ситуации.

Кратко представяне на проблема:

В повечето софтуерни програми са реализиран общи подходи за работа с виртуалната памет, които удовлетворяват само една част от разработчиците. При задълбочен анализ на тези подходи обаче възникват проблеми в ситуации, в които очакваме софтуерът да бъде стабилен и с константно времево поведение. Поради тази причина в настоящата дипломна работа чрез авторска софтуерна разработка, предлагаме нетрадиционен метод за заделяне на памет.

В заключението се обобщават постигането на основната цел, разрешаването на поставените задачи, като се анализират постигнатите резултати и се прави оценка на следващите стъпки за развитие на нова функционалност.

Глава Първа

1. Анализ на паметта на програмата и нейното използване. Необходимост от различен подход

В съвременните програми се използват два основни типа за заделяне на памет. Директен подход върху мениджмънта на паметта чрез указатели и индиректен подход.

Пример е езикът Java, в него индиректно виртуалната машина (JVM) спрямо това колко често се използва определен обект, той попада в различни контейнери и през определен период от време, ако контейнерът трябва да бъде почистен т.е. данните в него вече не се използват, паметта се освобождава.

Втори пример за индиректно освобождаване на памет е изграденият модел в C++ 11. Той се възползва от така наречения механизъм за броене на референции и съответно, когато референциите към даден обект са 0, паметта за този обект се освобождава.

Тези два подхода се използват широко и тяхната употреба, за съжаление, се насърчава. По нататък в дипломната работа ние ще обсъдим защо такива подходи могат да попречат на бързодействието на програмата и на разбираемостта на кода, както и защо в повечето случай те увеличават времето за изчистване на грешки от програмите (debugging), тъй като те взаимодействат с характеристики на езиците C++ и Java, които имат и своите лоши страни. Някои характеристика са:

- капсулация на данни
- конструктори и деструктори
- класове с наследяване
- динамично инициализиране
- принципът resource acquisition is initialization (RAII))

Нека първо разгледаме и втория подход и по-точно ръчното управление на паметта. То от своя страна има доста изградени стратегии през времето. Някои от основните са генералните решения или решенията, които се опитват да разрешат всички проблеми с паметта наведнъж. Това са стандартните алокатори в повечето езици за програмиране, които позволят работа с паметта на ниско ниво. В случая със C/C++ това са malloc()/free() new()/delete(). Естествено те се провалят в някои критични ситуации и ние ще ги анализираме. Отново при тях преобладават скрити проблеми, които ще обсъдим по-обстойно по нататък в дипломната работа.

Съществуват също и няколко различни конкретни подхода към ръчното разпределение на паметта. Някои от тях са използване на един блок или така наречените линейни алокатори, които в последствие се разбиват на множество малки подравнени или не алокации в даденото блоково пространство. Такъв подход крие и

своите недостатъци, пример за това е как бихме разширили един такъв алокатор. Това също е разгледано в тази дипломна работа при анализа в Глава 2.

Следващ тип алокатор, който използва ръчния подход е такъв, който реално е стек. Това позволява да се свива и разширява чрез използване на определени маркери, до които в последствие можем да се върнем. Най-простият пример е този на стека на изпълнение на всяка една софтуерна програма. Той записва последователно в паметта чрез определен регистър, всяко влизане в нова функция, така че при унищожаване на пространството (scope) на функцията да може да се завърне на правилното място и да продължи изпълнението на програмата без да наруши структурата й. Такава стратегия обаче също крие своите рискове и те се състоят най-вече в това как да не препълним дадения стек и как да разширим стековото пространство.

При такъв анализ бихме се зачудили какъв е най-добрият подход към паметта в една съвременна програма. Това ще анализираме и изградим в следващите глави от тази дипломна работа, като използваме някои от методологиите споменати по-горе за изграждане на алокатори на памет. Това ще ни предразположи към набор от страхотни предимства на ръчното управление на паметта.

1.1 Ползи от изграждането на нетрадиционен модел за заделяне на памет

- Възможност за по-голяма гъвкавост при използването на близки по размерност вид данни.
- По-добра видима настройка на подравняването на отделните алокации
- По-добро блоково разпределение.
- По-добро използване на ресурсите на операционната система
- По-голямо бързодействие на програмата
- Възможност за задълбочена оптимизация на програмната структура.
- Експлицитно заделяне и освобождаване на паметта.

1.2 Необходимост от различен подход към структурата на програмата

Различният подход е необходим, защото чрез първоначалния анализ на останалите методи се установи, че те крият недостатъци. Нека първо се спрем на конструкторите и деструкторите. Тази функционалност на пръв поглед изглежда съвсем безупречна. Нека обаче разгледаме следният примерен код.

```
static void MyFunctionInWhichIHaveNoIdeaWhatHappens()
{
    auto WhateverName = WhateverNameSpace::WhateverName(DefaultPropertiesOrEmpty);

    WhateverName. ?? -> ?? Member1 . ?? -> ??
}
```

Фиг. 1.1 Илюстриране на лошите страни на конструкторите и деструкторите.

Тук може да се забележат следните недостатъци:

- Когато използваме auto ние не знаем какъв тип очакваме, освен това не знаем дали всъщност имаме указател към по-сложен тип или реално сложният тип е върнат на стека.
- Също така може да се кажа, че си нямаме и на идея какво точно се случва в конструктора на обекта и също понеже името трябва да е едно и също като това на класа от него не може да се извлече никаква допълнителна информация.
- Следващото нещо е, че зад конструктора най-вероятно стой автоматично генериран код, който на базата на друга основа (мета тагове или друг анализиращ подход), генерира за всеки член (member) на класа по отделно C++ код, който трябва да се изпълни, т.е. ако имаме 3 члена в класа ни и те са от класа string, всеки един от тях ще извърши отделна собствена инициализация, което води до минимум 3 изрични извиквания на алокатора по подразбиране.
- Защо обекта (в случая променливата на стека) трябва да има живот до края на функцията? Това може да не е поведението, което желаем.
- Нека предположим, че извикаме експлицитно деструктора, тогава също си нямаме и на идея какво ще се случи.
- Следващото нещо, което е достатъчно натрапчиво е, че не знаем колко и къде се алокира тази памет. Колко заделяме на стека и колко памет от динамичната (heap memory) реално заделяме при конструирането на целия обект.
- Как е подравнена тази памет? - Също не е ясно.
- Колко памет ще освободим?
- Дали реално всеки един член е бил изчистен с нули (cleared to zero). Или е бил инициализиран по някакъв друг начин. Отново липсва достатъчно информация.

Нека да обърнем внимание и на капсулацията на данни или по-точно ако имаме един клас, защо повечето негови членове трябва да бъдат частни (private). Не трябва, защото така се усложнява излишно програмата като се пишат безсмислени сет-методи (set) и гет-методи (get). Ако случайно се изисква специална инициализация на даден член, то тогава е най-добре това да става изрично в функция. (която може и да се именува като гет-метод)

Освен това капсулацията на данни, може да причини нечетливост на кода, понеже често може да се обърка дали ако имаме функция принадлежаща на класа и в нея променлива, реално тази променлива принадлежи на стека на функция или е част от обекта и се използва и след това, като се записва евентуален резултат вътре в нея, може променливата да изчезне след изчистване на стека за функцията, а това да не е очакваното поведение.

Наследяване, групирано с капсулацията на данни. То предизвиква умишлено скриване на функционалност като генерална и предизвиква в доста от случаите,

нещо което може да се нарече скрита верига или клас наследяващ от клас от друг и така нататък. В крайна сметка ние не знаем състоянието на общата памет и как точно да инициализираме обекта.

Resource acquisition is initialization (RAII) е принцип, който крие страшно много проблеми най-вече около това, че се фокусира върху състоянията на обектите по отделно, отколкото върху цялостната функционалност на дадена система. Езици, които имат концепции насочени към този принцип също не различават, че има разлика между периода на живот на обекта и това кой ще бъде неговият притежател и дали ще се смени.

След като анализирахме недостатъците на някои функционалности, сега ще се насочим към тяхното елиминиране или използване само при необходимост. По долу илюстрираме как чрез новия подход ние вече знаем, много от детайлите, които липсваха, когато анализирахме случая с конструкторите и деструкторите.

```
enum stack_push_flags : uint32_t
{
    StackFlag_DontClear = 0x0,
    StackFlag_ClearToZero = 0x1
};

enum platform_push_flags : uint32_t
{
    PlatformMemory_NoCheck,
    PlatformMemory_UnderflowCheck,
    PlatformMemory_OverflowCheck
};

struct stack_push_params
{
    stack_push_flags Flags;
    uint32_t Alignment;
};

struct stack_platform_params
{
    platform_push_flags AllocationFlags;
    uintptr_t MinimumBlockSize;
};
```

Фиг. 1.2 Структури описващи параметрите при заделяне на памет

Това са демонстрационни структури, които се използват за разработването на подобния на стек алокатор. Вижда се, че ние знаем дали паметта ще бъде изчистена от флаговете на stack_push_flags enum. Също от следващия enum, може да се разбере дали паметта, заделена от операционната система, ще бъде защитена против overflow и underflow. Това означава дали ще може да се хване, ако някой пише извън границата на заделената си алокация в дебъг режим на алокатора. Разбира се, ясно е и какво ще бъде подравняването на алокацията и какъв да е зададеният оптимален размер на блока на стековия алокатор, за да се намали до минимум фрагментацията.

В Фиг. 1.3 виждаме, че може да се използват помощни структури, които да позволяват по кратък и оптимален запис. Чрез тях ще демонстрираме как по-добре да разпределяме паметта в нашата програма, а не както в случая с Фиг. 1.1

```
static stack_push_params
DefaultStackParameters()
{
    stack_push_params Params;
    Params.Flags = StackFlag_ClearToZero;
    Params.Alignment = Bytes(16);
    return Params;
}

static stack_push_params
NoClear()
{
    stack_push_params Params;
    Params.Flags = StackFlag_DontClear;
    Params.Alignment = Bytes(16);
    return Params;
}

static stack_allocator
DefaultStackAllocator(uintptr_t MinimumBlockSize, platform_push_flags Flags)
{
    stack_allocator StackAllocator = {};
    StackAllocator.AllocationFlags = Flags;
    StackAllocator.MinimumBlockSize = MinimumBlockSize;
    return StackAllocator;
}
```

Фиг. 1.3 Помощни методи илюстриращи по-лесната параметризация на стековия алокатор

```
//NOTE(enev): Now the lines of code are 2,
//but we have a much bigger and significant understanding of what is actually supposed to happen!
static void
MyFunctionInWhichAtleastPartiallyIHaveAnIdeaOfWhatIsHappening()
{
    stack_allocator StackAllocator = DefaultStackAllocator(KiloBytes(64), PlatformMemory_NoCheck);
    my_strict_type* WhateverName = PushStruct(StackAllocator, my_strict_type, DefaultStackParameters());
    //my_strict_type2* WhateverName2 = PushStruct(StackAllocator, my_strict_type2, NoClear());
}
```

Фиг. 1.4. Демонстрационна функция, която показва как се конфигурира разработеният вече алокатор и как се осъществява алокация.

След като демонстрирахме какво е необходимо да знаем, когато работим с памет, нека проверим дали наистина сме се справили.

- Колко голям блок алокация ще извърши операционната система и защитен ли е той? – 64 кб и няма да се правят чекове на всички алокации по-време на изпълнението на програмата за целия период.

- След това, знаем ли при единствената алокация, която правим колко е голяма тя? - Да, защото взимаме размера на зададения тип sizeof(my_strict_type).
- Знаем ли как е подравнена тази алокация? – Да, защото функцията DefaultStackParameters() инициализира и връща резултата със стойност по подразбиране - 16 байта.
- Знаем ли дали паметта ни е нулирана? – Да, чрез флага от stack_push_params StackFlag_ClearToZero.

Трябва да се отбележи, че чрез този подход се фокусираме единствено и само върху паметта, отсяваме всякакви допълнителни странични неща, независимо дали са необходими или не.

Чрез тези два реда ние успяхме да създадем алокатор, да го конфигурираме, както желаем и да го "тестваме" като заделим една алокация, която е с точен очевиден размер и е подравнена както желаем от блок с поискан определен размер.

1.3 Подобряване на бързодействието в определени ситуации

Когато заделяме памет по демонстрирания на Фиг. 1.4 начин, ние можем да групираме лесно няколко алокации в една с общ размер и да елиминираме нуждата от допълнителни ненужни извиквания на метода за заделяне. По този начин избягваме алокирането на памет в много от по-сложните и итериращи части от програма, някои от които могат да достигнат до 200 000 извиквания на един и същи метод на кадър (една итерация на програмата). Допълнително ще се разгледат още няколко сценария в Глава 2, когато изграждаме вече функционалността.

1.4 Линейни подходи към заделянето на памет

Тези подходи се фокусират върху последователното заделяне на памет и имат за цел да ускорят бързината на действие при един и същ блок като използват единствено събиране/изваждане зависимост дали адреса е по-нисък или по-висок, и връщат веднага резултат към новия адрес в паметта. Това става само с една проверка дали не надвишават блоковия размер. Ще се възползваме от този подход по време на разработването на нашия нетрадиционен алокатор.

1.5 Стекът на програмата и анализ на бързодействие

Всяка една софтуерна програма работи чрез програмен стек, ние ще се опитаме да се доближим максимално до тази методология на стека, за да извлечем няколко конкретни ползи, една от които е бързодействието на програмата, а друга е възможността за използване на временната памет, която да бъде освобождавана по желание.

2. Ползи от визуализацията на стековия алокатор

Визуализацията е една от най-важните стъпки в изграждане на цялостната система и среда за разработка на по-високо ниво.

Чрез показване на екрана на всяка една от извършените алокации, ние може да се уверим, че те реално са в желания размер, също броя им е този, който сме задали в програмния код. Уточняваме също, дали съществува подравняване и с какъв размер е било то. Следващото нещо е да се уверим, че няма никаква визуализирана фрагментация. В случай, че има такава или видим в реално време изтичане на памет чрез малки блокове, ние веднага може да се върнем и коригираме дадения програмен код.

Визуализацията е от изключително значение, защото улеснява неимоверно разработчика и позволява да се види в дебъг режим, че реално нетрадиционния алокатор работи и спазва всички подадени изисквания.

Какво се има предвид под дебъг режим?

Този режим казва на алокатора да третира всяка една алокация като отделна такава и да я защити от двете страни със страници памет, върху които не може да се пише. Така при евентуално излизане от границите програмата ще спре мигновено и ние веднага ще открием, че съществува проблем. В обратния случай, ако не е включена тази опция, ние можем да презапишем наша друга последователна алокация в блока и да причиним грешен запис на данни, който на пръв поглед е много трудно да се установи какво точно го е причинило.

2.1. Анализиране на съществуващия софтуер на пазара

На пазара в момента има изключително много инструменти за проверка на паметта и за нейната визуализация, но те се фокусират върху масово използваните технологии, т.е. върху стратегиите за общи алокатори като стандартните в C/C++.

Поради тази причина те няма да ни свършат достатъчно работа, може да използваме такава програма, която следи алокациите на операционната система, но тъй като нашият алокатор се възползва от блокове памет, ние няма да може да разберем какво се случва в самите блокове и ще разполагаме с непълна информация.

Valgrind е един от най-известните инструменти за тази цел, той е с отворен код и е най-широко използван.

2.2. Идентификация на основни необходими параметри, като размер и местоположение, за правилен обзор на състоянието на паметта

Визуализацията трябва да бъде полезна на разработчика, за тази цел е необходима следната информация.

Нека започнем от името и местоположението на метода. То се извлича чрез предпроцесора на езика C/C++, като се използват макрота. Тези макрота позволяват да разберем от коя линия и в кой файл съответно се намира нашият метод, както и негово име (независимо дали използваме шаблони (templates) или не).

Също така се събира информация като уникална идентификация на инстанцията на алокатора, за да знаем в кой поток от визуализацията на всички налични стекове да включим нашата алокация със съответния заделен размер. Събират се също данни за самата алокация като размер (поискан и подравнен), фрагментация и блоков индекс. Трябва да пазим и идентификацията (местоположението) на съответният алокатор и неговата инстанция (първа, втора и т.н.), за да може да съхраним алокацията на правилното място.

Всеки един стеков алокатор се индентифицира уникално чрез местоположение и номер на инстанция. Чрез индентификацията бързо откриваме в хеш таблица с константен размер (за бърз достъп) дали съществува такъв дебъг запис, който реално е различна структура от данни. Отделната структура се прави с цел да се оптимизира нормалното действие на програмата. Опитваме се да ограничим информацията, която трябва да се добави в оригиналната структура, а тя е само идентификационните данни. По този начин не повлияваме на бързодействието на самият метод за заделяне на памет. Чисто програмно се улеснява и разбирането коя част принадлежи на дебъг визуализацията и коя не.

В последвие, когато се итерират всички данни в отделната дебъг структура, тя държи и информацията за всички направени алокации последователно, което улеснява обхождането и в последствие визуализацията.

3. Анализ на необходим графичен потребителски интерфейс

3.1 Основни разлики с други модели за изграждане на потребителски интерфейс.

Съществуват основно няколко начина на програмиране на потребителски интерфейси.

Retained mode graphical user interface – Обикновено в такъв графичен интерфейс се изисква доста голямо количество код, който да поддържа всички характеристики на състоянието. В този тип среда има наличие на дърво, което се състои от различни инструменти на графичния интерфейс, като прозорци, бутони, текстови полета и т.н.. Трябва да се отбележи, че обхождането на един такъв тип дървовидна структура и изключителното много код за следене на състоянието неминуемо водят до замърсяване на кода и обръщане внимание твърде много към системата за изобразяване на графичен потребителски интерфейс. В контраст с това следващият модел за графичен потребителски интерфейс се противопоставя на тези възгледи.

Immediate mode graphical user interface – при този тип интерфейс, не е нужно от потребителска гледна точка да се следи състояние, това се прави вътрешно библиотечно и чрез минимален обем от данни. Това води до по „чист“ код и възможност да не трябва да менажираме толкова много състояния. Този подход значително опростява кода и дава възможност да се фокусираме върху реалния проблем, които се опитваме да разрешим.

В тази дипломна работа ние ще се сблъскаме основно с Immediate mode graphical user interface поради горе споменатите причини. Той е подходящ за бързо изграждане на инструменти за визуализация, което изцяло подхожда на нашата разработка.

Нека разгледаме и всички необходими средства за разработката на такъв тип стабилна функционалност, на която можем да разчитаме през целия период на изграждане на един софтуерен продукт.

4. Средства за разработка

4.1 Езикът C е разработен от Денис Ричи в периода 1969г. –1973г. в AT&T и Bell Labs. Качествата на C като абстрактност и преносимост бързо го правят популярен и през 1973 г. операционната система UNIX е пренаписана на този език, което значително улеснява пренасянето ѝ на други машини. Тъй като първоначално всяко ядро се е пренаписвало на различен асемблер за нов вид машина. [1]

През 1978 г. Браян Керниган и Денис Ричи написват книгата The C Programming Language. Тази книга е първата по-формална спецификация на езика. Изключителната популярност на езика довежда до разработването на компилатори и разновидности с различни разширения за множество компютри. За да подобри преносимостта на кода и за да осъвремени съществуващия стандарт, през 1983 г. Американският национален институт по стандартите (ANSI) сформира работна група имаща за цел да създаде формално описание на C. През 1989 г. стандартът е завършен и ратифициран под името ANSI X3.159 – 1989 „Programming Language C“. Версията на езика описана в документа често се нарича ANSI C или C89.

C99 е нов стандарт на езика, утвърден през 1999 от ISO/IEC, добавящ някои важни възможности.

Програмите написани на C представляват съвкупност от файлове с изходен код. Чрез компилация всеки файл се превежда до машинно зависим обектен код за определена архитектура, а свързващ редактор (linker) обединява тези относително независими обектни файлове (obj) в цялостен изпълним файл.

C е структурен език, използващ конструкции като условни преходи (if.. else), цикли (while, for) и абстрахиране на кода чрез функции изпълняващи относително самостоятелни задачи.

Някои от най-важните характеристики на езика са:

- Статично типизиран (static typing) – типовете на данните се определят по време на компилирането.
- Позволява работа с паметта на ниско ниво – поддържа указатели към променливи, указатели към функции и адресна аритметика.
- Използва сравнително малък набор от ключови думи, като за сметка на това използва сравнително голямо количество оператори.

- Всеки израз връща стойност – дори присвояването, което прави възможно множествено присвояване от вида x=y=5. Стойностите обаче могат да бъдат игнорирани когато не са нужни.
- Параметрите към функциите винаги се предават по стойност. Така достъпът до външна променлива чрез параметър може да стане само индиректно чрез подаване на указател.
- Изобилстваща употреба на предпроцесор за създаване на макро дефиниции, включване на файлове и условно компилиране.

Основни типове данни:

Типовете данни зависят от конкретния компилатор който се използва.

char	Основната адресируема единица в машината – байт. В C се използва предимно за работа със символи.	от –128 до 127
unsigned char	Основната адресируема единица в машината – байт. В C се използва предимно за работа със символи.	от 0 до 255
short	Цели числа	от –32 768 до 32 767
unsigned short	Естествени числа	0 до 65 535
int, long	Цели числа	от –2 147 483 648 до 2 147 483 647
unsigned int, unsigned long	Естествени числа	от 0 до 4 294 967 295
float	Число с плаваща запетая.	от ± 1,40239846×10−45 до ±3,40282347×1038
double	Число с плаваща запетая с двойна прецизност.	от ±4,94065645841246544×10−324 до ±1,79769313486231570×10308
void	неопределен тип	

Фиг. 1.5 Таблица показваща основни типове при определена процесорна архитектура в C

Запазени думи

Стандартът ANSI C определя 32 ключови думи, които не могат да се използват в имената на функции или променливи. Много компилатори на C добавят други ключови думи. В C главните и малките букви се различават (int, Int и INT са различни).

Запазени думи:

auto, _Bool, break, case, char, _Complex, const, continue, default, do, double, else, enum, extern, float, for, goto, if, _Imaginary, inline, int, long, register, restrict, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while

4.2 C++

C++ (произнася се „си-плюс-плюс“) е неспециализиран език за програмиране от високо ниво. Той е обектно-ориентиран език със статични типове. От 1990-те C++ е един от най-популярните комерсиални езици за програмиране.

Датският програмист Бярне Строуструп разработва C++ през 1983 г. в Лабораториите „Бел“ като разширение на езика C – езикът е базиран на C, но в него са добавени редица допълнителни възможности и са направени няколко промени.

Основната разлика между C и C++ е, че C++ съдържа вградена в езика поддръжка на обектно-ориентирано програмиране. В C++ са добавени класове, множествено наследяване, виртуални функции, overloading, шаблони (templates), обработка на изключения (exceptions) и вградени оператори за работа с динамична памет.

Езиковият стандарт на C++ е ратифициран през 1998 като ISO/IEC 14882:1998, през 2003 година има преразглеждане на стандарта – ISO/IEC 14882:2003, а от 2011 стандартът се обновява на всеки три години. Последната ревизия е от 2020.

Повечето програми на C могат директно или със съвсем малки модификации да бъдат компилирани с компилатор за езика C++. Поради заимстване на множество концепции от C++ езикът Java има много общи черти със C++.

Една голяма част от приложните програми на много операционни системи, както и някои от самите операционни системи, са написани на този език.

4.3 Средства за разработка на визуализацията

OpenGL (Open Graphics Library)

OpenGL (Open Graphics Library) е мощен приложно-програмен интерфейс (API) за реализиране на лесно преносими графични програми. Създаден е през 1992 г., OpenGL бързо става един от най-популярните програмни интерфейси за реализиране на 2D и 3D графика. За това допринасят широката му достъпност, съвместимостта му с различни операционни системи и с различни компютърни платформи.

Тази библиотека е подходяща за приложения изискващи високо качество на изображението, комбинирано с добра производителност, за да бъде възможно генерирането му в реално време.

Основните характеристики на OpenGL

- Индустриален стандарт - развитието на OpenGL се управлява от независим консорциум (OpenGL Architecture Review Board), който има широка поддръжка в индустрията и гарантира, че OpenGL ще остане наистина отворен, неутрален и многоплатформен стандарт.
- Преносимост
- Скалируемост
- Стабилност
- Постоянно развиване и обновление
- Простота на употребата
- Наличие на добра документация

4.4 Избор на минимални зависимости от C++

Един от недостатъците на C++ е, че е сложен и има изключително много различни стилове на писане.

Затова тази част от дипломната работа е отделена, за да се наблегне на нещата, които се използват. Те са следните:

- Предефиниране на оператори (Operator Overloading)

То се състои в това, че произволен оператор като пример оператор за събиране "+" може да изпълнява произволна функция, която дори да извършва обратното действие. Тази функционалност обикновено се използва за по сложни структури от данни като вектори и матрици.

- Предефиниране на функции (Function Overloading)

Функционалността се използва, когато желаем да напишем едно и също име на метод/функция, но имаме различен брой аргументи в декларацията. C++ осъществява това, като използва name mangling. Това е техника, чрез която се генерират уникални имена на функциите, за да може те да бъдат разпознавани.

- Използване на аргументи по-подразбиране в функциите/методите.

Ако имаме функции, които през повечето време използват константно едни и същи стойности може да ги направим да бъдат по-подразбиране, но при условие, че се намират в края на дефиницията на аргументите.

- Елементарни Шаблони (Templates) [5]

Шаблоните в C++ са трудоемки за повечето софтуерни разработчици. Ще се спрем на това какво представлява шаблон и как да го имплементираме в най-общите случай.

Шаблони са въведение в C++, което позволява функции, класове и структури да оперират върху общи типове. Това позволява на функцията или класът да работи с много различни типове без да бъде пренаписвана за всеки един от тях.

Ще разгледаме шаблони само за структури и класове, тъй като това е основата функционалност използвана в тази дипломна работа.

Основните характеристики на шаблоните са следните:

```
template< template-parameter-list >
```

Където template-parameter-list е списък разделен със запетаи на един или повече от следните типове шаблонни параметри:

- тип

- не типов (клас или друг)

- шаблон

Типовете, променливите, константите и обектите в шаблон на клас/структура могат да бъдат декларирани в параметрите на шаблона, както и изрични типове като (int, char).

Също така може да дефинираме шаблони по следния начин.

```
template<class L, class T> class Key;
```

Обекти и член функции на индивидуални шаблони могат да бъдат достъпвани по всички начини, по които могат да се достъпят и обикновените членове на класове обекти и функции.

Следният код е пример за шаблон:

```
template<class T> class Vehicle
{
public:
    Vehicle() { /* ... */ }    // constructor
    ~Vehicle() {};             // destructor
    T kind[16];
    T* drive();
    static void roadmap();
    // ...
};
```

Тук виждаме и как да инстанцираме шаблон от такъв тип.

```
Vehicle<char> bicycle; // instantiates the template
```

4.5 Динамични библиотеки (dll, so)

Първо библиотека е пакетиран код, който може да бъде преизползван от много програми. Обикновено библиотеките се разделят на два типа в C++.

- Статични библиотеки (static libraries)

Статичните библиотеки са такива, които са компилирани и свързани директно в изпълнимия файл на програмата. Те имат .lib/.a разширение. Предимството е, че те идват заедно с изпълнимият файл на програмата. Недостатък е, че ако трябва да се обнови дадена версия, трябва да се разпространи на ново целия изпълним файл на програмата.

- Динамични библиотеки (dynamic libraries)

Динамични библиотеки се състоят от компилиран програмен код, който може да се преизползва от множество програми. Тези библиотеки имат разширение .dll/.so . Предимството им е, че заемат много по-малко място и още по-голям плюс е това, че те могат да бъдат обновявани без да се променя отново цялостния изпълним файл на програмата.

Ще разгледаме динамичните библиотеки и как да представим нашия програмен код за визуализацията на стеков алокатор, а и самият алокатор чрез такава. Необходимо е да се изгради прост интерфейс за комуникация. Това може да означава дори и само експортиране на определени функции и в последствие извикването им от изпълнимия файл на програмата. Като това ще позволи, в зависимост дали разполагаме с тази библиотека да я заредим динамично и след това да използваме визуализацията. Не е необходимо по този начин да представяме на крайния потребител код, който всъщност е необходим само за вътрешна разработка.

4.6 Визуализация чрез immediate-mode graphical user interface.

Фокусирайки се върху изграждането на визуализацията, достигнахме до няколко критерии, които желаем да изпълним.

Библиотеката, която използваме трябва да е лесно преносима. Освен това е необходимо да бъде изключително лесно да се следи имплементацията на визуализацията или програмния поток. За тази цел е подходяща библиотека, която изпълнява инструкциите на място, което означава, че пълни буферите за чертане веднага, а не държи допълнителни помощни структури, които в по следващ етап да бъдат итерирани една по една. Всеки прозорец съхранява свой собствен буфер, който е попълван мигновено щом се извиква функция за визуализация, пример е рендиране на графичен бутон. Такъв тип графичен потребителски интерфейс не трябва да се бърка с рендиране или чертане мигновено (immediate). Всеки от буферите на дадените прозорци в последствие се рендира бързо и цялостно като поток от данни, а не един по един елементи. Друг плюс на такъв тип потребителски интерфейс е, че не е необходимо изрично да се държат различни състояния и да „замърсяваме“ останалия код. Ако се върнем към графичния бутон, той се визуализира изключително лесно и позволява висока четимост на кода. Това се прави чрез само една линия.

```
if(Button("MyButtonName"))
{
    //On true button was pressed, do whatever you want
}
```

Фиг. 1.6 Примерен код за визуализация на бутон

4.7 Насоки и взаимствания на шрифтови библиотеки

За визуализацията е необходимо рендирането на текст. В тази имплементация е използван файловият формат .ttf, като на базата на него е изграден програмен инструмент, който се изпълнява преди старта на нашата основна програма и неговата роля е да разнищи ненужно сложния ttf формат, след това да рендира чрез процесора всяка една буква и да я разположи в колкото се може по-малко текстурно пространство. За английски език се генерират 2 текстури, които съдържат всички символи от определен размер и шрифт. Този подготвящ инструмент е изграден с помощта на библиотека с отворен код stb_truetype.h. [4]

5. Съвместимост с досегашни системи

Визуализацията, както и стековия алокатор са напълно съвместими с досегашни системи. Има се предвид, че стековия алокатор може да се използва съвместно с друг тип общи алокатори. Освен това няма абсолютно никакъв проблем да се използва съвместно с други специфични подходи като линейни и стек алокатори.

6. Възможности за многоплатформеност

6.1. Линукс и Уиндоус и Мак

Многоплатформеността е постигната сравнително лесно, поради факта, че за цялостната имплементация на стековия алокатор се използва една функция принадлежаща на операционната система. Следователно като се направи един слой, който да абстрахира тази функция и спрямо операционната система да се изпълни различен метод, имплементацията от този етап нататък е абсолютно независима стига да имаме операционна система, която да разполага с алокатор за общи цели.

7. Основни структури от данни използвани за разработването на стековия алокатор.

7.1. Свързан списък

Свързаният списък е линейна структура от данни, която съдържа в себе си поредица от елементи. Различава се от масива по това, че може да се оразмерява динамично. Списъците имат свойството дължина (брой елементи) и елементите му са наредени последователно. С помощта на имплементираните към него методи, е

възможно добавянето на нови елементи на която и да е позиция в списъка, махането на такива, обхождането или обръщането на тези елементи и т.н.

7.2. Използване на предпроцесора на компилатора за необходимите метаданни чрез макрота.

Директивата #define се използва за дефиниране на макроси.

Макроса представлява обособена част от програмата, която може да се повтаря многократно. Ние ще се съсредоточим върху макроси с параметри.

```
#define PI 3.14

#define MY_MACRO(Param1, Param2) MyFunction(Param1, Param2)
```

7.3 Хеш таблица със статичен размер

Хеш таблицата е структура от данни, съдържаща ключ и данни, която се характеризира с директен достъп до елементите, независимо от типа им. Елементите ѝ, подобно на тези на други структури от данни използвани за търсене, се състоят от ключ и данни.

Двойно хеширане (double hashing)

При този метод се прави повторно хеширане на вече намерения хеш код, но с друга хеш функция, съвсем различна от първата. Този метод е по-ефективен от линейното и квадратичното пробване, тъй като всяко следващо пробване зависи от стойността на ключа, а не от позицията определена за ключа в таблицата. Това има смисъл, защото позицията за даден ключ зависи от текущия капацитет на таблицата.

7.4. Кръгов двойно свързан списък

Двойно свързаният списък е свързана структура от данни, състояща се от множество последователно прикачени елементи. Всеки един елемент съдържа две полета, наречени връзки, които са указатели към предишния и следващия елемент в поредицата. Връзките на началните и крайните елементи в двойно свързания списък имат по един специален вид разклонение, служещо за прекратяване обхождането на списъка. Този специален вид разклонение обичайно е празен елемент (sentinel node) или null. Ако списъкът има само един празен елемент, то той е кръгообразно свързан чрез него.

8. Интеграция

Интеграцията на имплементацията на стековия алокатор не е проблемна, понеже тя е независима от други библиотеки. Единствената зависимост е от алокатора за общи цели на операционната система, но над операционната система се прави сравнително лесно абстракция описана в Глава 2 на дипломната работа.

Глава Втора

Проектиране, изграждане и имплементация

9.1 Анализ на основните етапи на една софтуерна програма.

Като разработчици знаем, че всяка програма неизбежно трябва да бъде стартирана и да задели определен вид и обем ресурси на куп, това обикновено става рязко при старт като по време на работа на програмата след нейната инициализация обема на заделяни данни спада и поддържа приблизително константно ниво. Също така, огромна част от изчисленията в реално време се постигат чрез временна памет, която в последствие е освободена. След като вече сме направили съответните изчисления и промени настъпва момент, в който желаем да приключим изпълнението на нашия софтуер и да освободим заделените ресурси.

Това са трите основни етапа, които разглеждаме и за тяхното удовлетворение ще търсим подход.

- Възможност за рязко покачване на паметта.

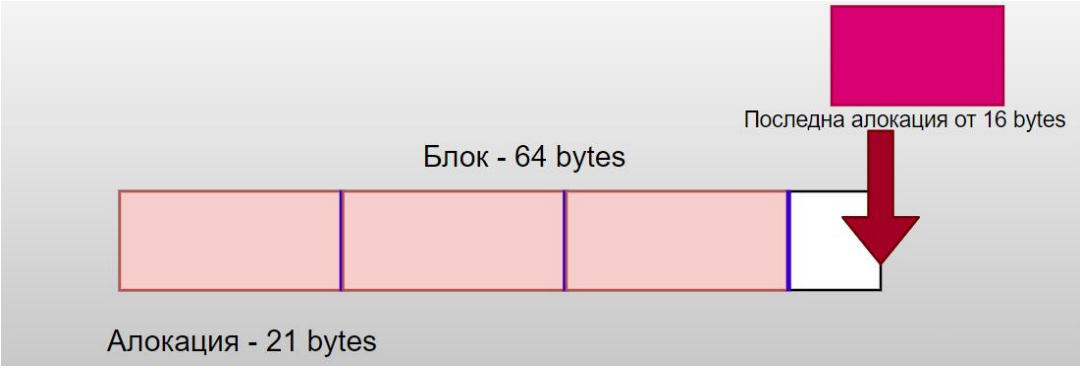
- Временна памет за изчисления.

- Освобождаване на всички ресурси

9.2 3-те основни функционалности

За да постигнем тези 3 основни етапа ние ще разгледаме няколко различни метода за заделяне на памет и как бихме могли да ги комбинираме, така че да постигнем максимална ефективност.

Ако стартираме чрез така наречените линейни алокатори. При тях се заделя подравнен блок памет, който е константен и всяка алокация е подравнена или не (в зависимост от програмните изисквания).



Фиг. 1.7 Илюстрация на линеен алокатор

Възниква въпросът как да разширим съответния блок, за да удовлетворим съответно едно от нашите изисквания за рязко покачване на паметта.

Съществуват два основни варианта единият е да заделим нов цялостен блок да копираме стария в новата памет и след това да го освободим.

Този подход обаче крие доста голям риск, който често е непредвидим и нарушава цялостното бързодействие на програмата. Защото, ако са ни необходими едва няколко десетки байта във функция, може изведнъж да се окаже, че ние трябва да копираме и освобождаваме изключително голям блок и едва след това да продължим нашата простичка и бързодействаща функция. Доста, често това обаче остава скрито от нас, тъй като по-големият обем от работа се извършва от операционната система или надграденият над нея стандартен алокатор използван в повечето C/C++ програми, а и при определени ситуации в зависимост от оптимизациите на операционната система може и да не усетим, че въобще има риск от потенциално забавяне и да разберем, чак когато софтуерът се изпълнява на друга машина и при други обстоятелства.

Фиг. 1.8 Илюстрация на механизъм за разширение на блокова памет

Нека разгледаме и втория основен вариант. Можем да използваме нов блок, но как да ги свържем, съществува и проблем, че разбиваме така нареченото адресно пространство и при последователност на заделянето може да ги разбием по средата и без да разберем да предизвикаме проблеми.

Нека започнем с “разбиването” на адресите това може да разрешим като резервираме памет, но не я поверяваме (commit), т.е. резервираме голяма част от виртуалното адресно пространство, но използваме само при необходимост.

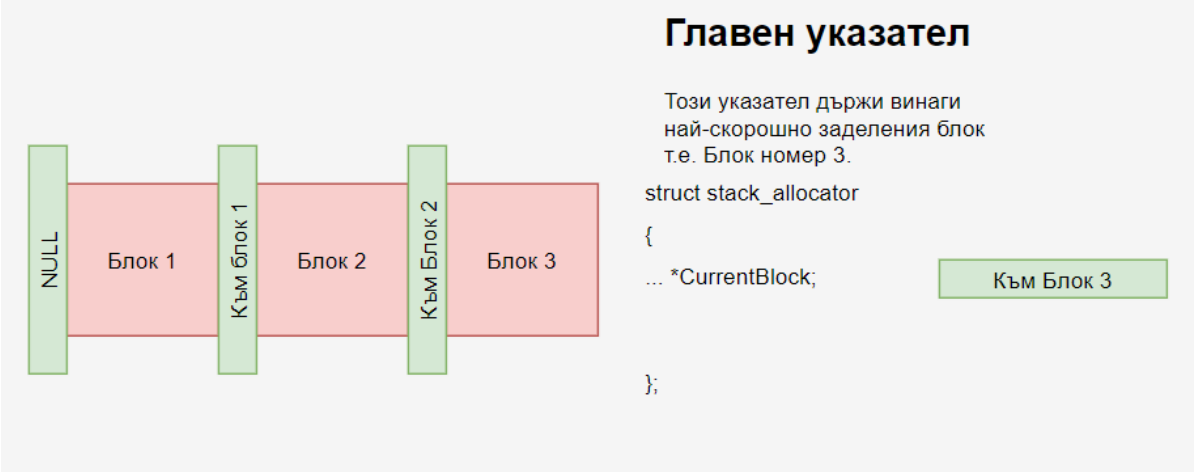
Пример може да бъде изразен чрез код за Windows операционната система.

```
        lpvBase = VirtualAlloc(NULL, MAX_MEMORY_PAGES * dwPageSize,
MEM_RESERVE, PAGE_NOACCESS);
```

В този случай дефинираме NULL като първи аргумент, за да улесним операционната система и тя да избере собствен адрес, в който да резервира паметта. След това MAX_MEMORY_PAGES е дефинирано от максималният размер на памет, който позволяваме в нашата програма и е разделен на най-добрият размер за страница (той може да варира при различни операционни системи).

Следващият проблем е как да свържем блоковете и това ще постигнем чрез най-обикновен свързан списък, представляващ един указател към последния (най-скорошно) резервирания блок. От там нататък всеки блок има свой указател към предишния заделен блок, което позволява лесното последователно обхождане. Има

и още няколко причини, за да се избере вторият подход, но те ще станат ясни по-нататък в нашия анализ.



Фиг. 1.9 Използване на свързан списък за управление на отделните блокове

9.3. Анализ на "ядрото" на всяка една софтуерна програма

Знаем, че всички програми използват програмен стек, няма как да функционират без него.

По време на изпълнение на програмата, динамично или не, памет е заделяна и освобождавана до лимита на съответния стек, който лимит знаем, че може да бъде конфигуриран по време на компилация на програмата чрез съответните флагове и размер.

Съществуват и такива подходи към алокаторите за заделяне на памет, които се възползват от свиването и разширяването на стека, но отново съществува проблема, че паметта е заделена в един единствен блок. От предишния анализ на линейните алокатори избрахме да разширим нашия блок като просто прикачим още един последователен към него. Сега остава да намерим начин как да се възползваме от временната памет подходяща точно при стековите алокатори.

За целта ще използваме функция и променлива принадлежаща на нашия алокатор наподобяващ програмен стек, която да брой блоковете и ще връщаме като резултат, блокът от който сме започнали и колко памет е била заделена до временния момент.

```
temporary_memory Temp = BeginTemporaryMemory(stack_allocator* Stack);

EndTemporaryMemory(Temp);
```

Добър пример за изчисляване с временна памет е следния и по възможност той трябва да бъде консистентен за функцията, т.е. е препоръчително да се извиква в една и съща функция. По този начин не се изисква допълнителен мениджмънт на състояние или какъвто и да било допълнителен код и ресурс.

```
static void
DoTempCompToStoreRes(compute_input* Input, non_persistent_type* Out)
{
    double Tolerance = 0.0000001;
    CompareDoubleLessThan(Input->SomeMember1, 0.7, Tolerance) ? Out->Member1++ : Out->Member1--;
    CompareDoubleLessThan(Input->SomeMember2, 0.8, Tolerance) ? Out->Member2++ : Out->Member2--;
    Out->Member3 = CompareDoubleLessThan(Input->SomeMember3, 0.9, Tolerance) ? Out->Member2 + Out->Member1 : Out->Member1;
}

static void
AssembleToFinalOutput(non_persistent_type* Temp, persistent_memory* Persist)
{
    Persist->TotalDiff1 += Temp->Member1;
    Persist->TotalDiff2 += Temp->Member2;
    Persist->TotalDiff3 += Temp->Member3;
}

static void
DoComplicatedTotalDiffExpensiveCalc(persistent_memory* Result, stack_allocator* StackAllocator, input_data* Input, uint32_t Iterations)
{
    if(Result)
    {
        Result->PersistentMemory = PushStruct(StackAllocator, persistent_memory);
        uint32_t Count = Result->PersistentMemory->Count = Iterations;

        temporary_memory Temp = BeginTemporaryMemory(StackAllocator);

        non_persistent_type* NonPersistentMemory = PushArray(StackAllocator, non_persistent_type, Count);
        for(uint32_t I = 0; I < Count; I++)
        {
            DoTempCompToStoreRes(Input + I, NonPersistentMemory + I);
            AssembleToFinalOutput(NonPersistentMemory + I, Result->PersistentMemory);
        }

        EndTemporaryMemory(Temp);
    }
}
```

Фиг. 2 Примерен код за временни изчисления

Подобни временни изчисления изникват в почти всички сериозни програми, но трябва да се има предвид, че ще възникне самата ситуация и да се разсъждава в нейна полза.

Подобен проблем може да бъде срещнат при всякакви видове изчисления, дали в система за рендиране или в система за разрешаване на колизии при физични обекти, временните данни са навсякъде и няма как да не се възползваме от горе-споменатата техника.

9.4 Защо наричаме този алокатор стеков? Какво извлякохме от анализите?

От анализа до сега знаем, че ни трябва 3 основни характеристики. Те са следните:

- Възможност за рязко покачване на паметта.

- Временна памет за изчисления.

- Освобождаване на всички ресурси

От тези характеристики, временната памет или по точно записването на определени места, за да можем да се върнем в последствие към тях, принадлежи към стековите алокатори. Освобождаването на всички ресурси също е близко до тях свойство.

Освен това ние желаем да сме достатъчно производителни и да можем бързо при наличие на вече съществуващ блок от памет да върнем на потребителя на нашия стеков алокатор адресно пространство. Нашата цел е да постигнем всички тези характеристики и освен това да добавим разширяемост на паметта чрез различни структури от данни.

В следващият параграф ще се запознаем с точните детайли около това как се заделя памет при един такъв алокатор, имащ всички тези характеристики.

9.5 Същинската част на стековия алокатор

uint8_t* Result = PushMemory(StackAllocator, SizeInit, Params); - обосновка на имплементацията

Състои се от една главна функция, която връща като резултат указател към началото на адресното пространство на дадената алокация. Това е така нареченият резултат на функцията и ако съответно е нула това означава, че не сме успели да заделим памет поради причина скрита от нас от операционната система.

Освен резултата, нека разгледаме и входните параметри те са следните:

- адрес към стековият алокатор

- размерът на паметта, който искаме да заделим в uintptr_t

- параметрите за дадената алокация.

След параметрите нека започнем с главната фунцкионалност.

Първото нещо, което правим е да проверим дали вече съществува настоящ (CurrentBlock) блок. Ако имаме такъв ние ще извикаме помощна функция, която да определи от какво подравняване имаме нужда в зависимост от параметрите за алокацията, които сме подали.

1. Подравняваме, ако има блок.

След това излизаме от проверката и се налага да направим още една цялостна проверка. Тя се състои в това отново първо да проверим дали отсъства настоящ блок или сме препълнили чрез новата алокация настоящия и в двата случая ще ни трябва нов блок.

При нов блок взимаме размера само на поисканата алокация (без подравнената (тя е за случай, когато се побираме в настоящия блок и пропускаме този етап)). Следващото нещо е да използваме флаговете на алокатора, за да знаем как да алокираме новият блок от памет. В зависимост дали искаме да направим проверка за надписване или предписване на блока от памет (буфер) или не.

Ако желаем да правим проверки, то ще ни трябва размер на блока идентичен с този на алокацията, затова подравнява алокацията със степен на 2-ката (заради операционната система) и продължаваме към реалното заделяне.

Ако пък не желаем да правим проверки продължаваме по стандартния път стига да сме сложили минимален размер на блока.

Едно последно нещо преди реалната системна функция и нашата абстракция около нея да се изпълнят е, че трябва да проверим чрез вземането на максималният размер между поисканият и този на блока и изберем по-голямото количество.

След като имаме всичко необходимо идва и нашата абстракция над системната функция.

В тази абстракция първо ще добавим размера на една допълнителна структура. Тя се добавя в началото на всеки блок и е подравнена спрямо изискванията на операционната система. (64 байта в случая)

Защо добавяме тази структура към блока?

Тя е с цел да следим дали всички блокове са свързани подред един с друг и не пропускаме някой. Чрез тази структура може да изпълним, ако желаем допълнителна функционалност по време на изпълнението на останалата част от програмата и да видим дали всички блокове са налични. Как ще разберем това? Понеже са свързани в двоен свързан списък взимаме първия блок и започваме да итерираме, ако не стигнем до нашият блок отново (т.е. ако нямаме цикличност) ние сме допуснали грешка в мениджмънта на блокове и трябва да дебъгнем и разберем къде нещата са се объркали. След като сме добавили този малък размер към блока, продължаваме напред с изпълнението и се спираме на разчитането на флаговете.

Отново ако ще правим защита на паметта, ще са ни необходими 2 страници памет (една отпред и една отзад) на подравнената до степен на 2-ка алокация. Вече в зависимост от операционната система размера на страниците памет варира, но той може да бъде взет, когато стартираме даденият изпълним файл и питаме изрично какви са системните особености. След като имаме тази информация ние добавяме този размер 2 пъти към размера на подравнената алокация и заделяме чрез реалната вече системна функция. След като сме заделили, сега трябва да вземем размера на страницата отпред и отзад и отново да извикаме още една системна функция, с която да я защитим (make protected (VirtualProtect())). Сега е време да закачим този блок към двоичния свързан списък, който използва глобална променлива сентинел (Sentinel) и закачаме винаги отзад.

Накрая вече сме свободни да добавим към началото на заделения размер (размера на помощния блок и размера на евентуалната страница, за да го върнем като базов адрес на нашия стеков алокатор.)

Получили вече коректния заделен блок ние го представяме като настоящ, а старият, ако е съществувал такъв, го записваме към пойнтера в блока, който сочи към предишния (PrevBlock). Това се прави с цел да може да итерираме блоковете, когато имаме размерност на временна памет по-голяма от един блок и трябва да се освободим от тях.

2. При отсъствие на блок или не възможност да поберем сегашният размер в настоящия блок. Ние влизаме в етап, в който от нас се изисква да заделим нов потенциален блок. С това приключи втората стъпка тя е и най-трудоемката от целия процес по заделяне на памет.

Вече сме приключили с новият блок, имаме памет, излезли сме от условностите и директно заделяме с подравнения размер на алокацията. Заделяме като вземем базовия адрес на блока към него прибавим използваната досега памет от блока и подравняването на самата алокация. Това се явява и нашият резултат като базов адрес.

Инкрементираме използваното пространство в блока и продължаваме към връщането на резултата.

Последното нещо, което трябва да се има предвид, след като вече сме заделили, е опцията на получателя да поиска да нулира памет. Т.е. проверяваме дали флага за нулиране е включен, ако да това означава, че преди да върнем алокацията ние трябва да я почистим с нули.

След това вече сме готови да върнем базовия адрес на алокацията. С това основната функция приключва.

Можем да добавим и това как се създават помощни макрота, така че още повече да се улесни заделянето на памет. Ето един пример за прости структури

Това се разбива от макро.

fab* Fab = PushStruct(StackAllocator, fab);

Тази линия се доразвива от ново макро като в зависимост дали програмата е в дебъг визуализация или не

(*fab)PushMemory(StackAllocator, sizeof(fab));

има два варианта на макро заместване.

Първият е директно извикване на функцията => (*fab)_PushMemory(StackAllocator, sizeof(fab), DefaultStackParameters());

При което бързодействието на програмата не се забавя.

Вторият вариант по време на дебъг представянето е да се извика още един "слой" т.е. процедура, която освен, че съхранява информацията, като име на функцията и местоположение във файла, тя регистрира алокатора и алокацията, което се извърша в отделна дебъг база.

Дебъг базата от своя страна следи как се развива стековата памет и дали тази алокация е направена между обозначения за временна памет.

uint8_t Result = _PushMemoryDebug(StackAllocator, sizeof(fab), Params, __FILE__, __LINE__, __FUNCTION__);

9.6 Имплементация на методите за временна памет

- temporary_memory Temp =

BeginTemporaryMemory (StackAllocator); - обосновка на имплементацията

Функцията за започване на временна памет, която вече споменахме, приема един аргумент и това е стековият алокатор. Тя връща като резултат структура от данни, която съдържа важна информация за текущото състояние на паметта на стековия алокатор, за да може да се върнем към него.

От какво е изградена тази структура?

Тя включва 3 члена и те са съответно:

1. Адресът на алокатор (указателят), за да може да го достъпим в последствие.

2. Адрес към настоящият блок, за да можем да го разпознаем, когато итерираме обратно и искаме да се върнем към него.

3. Ако съществува настоящ блок, взимаме му размера иначе е просто нула

Чрез тези три данни в нашата структура, ние успешно ще можем да се върнем към коректното място. Освен това обаче, след като сме я попълнили, държим един брояч, който индикира колко „навътре“ сме навлезли като се инкрементира всеки път, когато започнем нова временна памет. Това ни позволя да следим колко нива сме след първата заявка за временна памет.

Това, което накрая ни остава е да върнем вече попълнената структура.

- **EndTemporaryMemory(Temp);** - обосновка на имплементацията.

Нека сега обърнем внимание на функцията за приключване на временна памет. Тя не връща стойност, но приема един аргумент, който е познатата ни вече структура от функцията за започване на временна памет.

Първото нещо, на което ще се фокусираме е while цикълът, който започва от настоящият блок и итерира докато не срещне блокът запазен в структурата.

В тялото на цикъла се изпълнява функция за освобождаване на блок. Какво представлява тази малка помощна функция? Тя има ролята да „откачи“ блока, като вземе неговият указател към предишния и го запише на веригата вместо него. След това извиква абстрактната функция за деалокиране на блок памет или освобождаване (тя е подобна на тази за алокиране). Абстракцията е нужна, защото за всяка операционна система функцията е различна. Какво се случва в тази абстрактна функция? Тя не връща нищо като резултат, но поема като аргумент указател към блока, който желаем да освободим.

Първо проверяваме дали въобще ни е подаден указател, различен от нула, тъй като понякога е нормално да се опитаме да освободим повторно така, че трябва да имаме предвид този случай.

След тази проверка навлизаме в частта, в която трябва да откачим от двойният свързан списък от нашия блок.

След това се насочваме към намирането на базовия адрес. Първо от адреса на блока, който получихме трябва да извадим стойността на помощната структура, която използвахме за двойния свързан списък, за да стигнем реално почти до базовия адрес на блока. След това трябва да проверим дали флаговете за протекции са включени и ако са трябва също да извадим стойността на една страница, за да стигнем до базовия адрес на блока, който сме заделили от операционната система и да го освободим. След освобождаването (VirtualFree()) функцията приключва успешно.

Clear(); - обосновка на имплементацията

Има и още една съществена функция, за която до момента не сме споменали нищо и това е как да подходим, когато искаме да изчистим всичката памет на стековия алокатор.

Тази функция работи по следния начин. Взима само указател към стековия алокатор като аргумент, но не връща нищо. Чрез указателя достъпваме алокатора и взимаме като локална променлива указателя на настоящия блок, за да може да изчистим указателя от структурата на стековия алокатор. Продължаваме като обхождаме всички блокове по-подобие на функцията за временна памет. Отново с цикъл while(CurrentBlock), докато съществува блок ние искаме да проверим дали блока е последен, като ако е последен, то указателят към предишния ще бъде 0(NULL). След като установим това изчистваме блока и чак след това излизаме от цикъла, ако блока е последен. В случай, че не е такъв се въртим в едно-свързания списък до неговия край.

9.7 Анализ на бърз и бавен път

Бърз път

Според имплементационната обосновка на функцията за заделяне на памет PushMemory(), би трябвало да имаме огромно предимство, когато сме на така наречения бърз път. Нека анализираме дали това е така.

Ще започнем от важността на това да заделим правилен размер на блока. При вече зададен такъв, може да се предвиди, че всяка следваща алокация ще бъде на бързия път в началото на имплементацията на трудоемка функция.

Ще разгледаме следната ситуация:

```
static void
IncreasingMemorySituation(stack_allocator* StackAllocator, uint32_t Iterations, uint32_t ExpectedMemorySize, do_stuff* DoStuff)
{
    uint32_t Size = ExpectedMemorySize;
    uint64_t TotalSize = Size;
    uint32_t AdditionalSpaceNeeded = 0;
    for(uint32_t I = 0; I < Iterations; I++)
    {
        //NOTE(enev): Here the default stack parameters are accepted!
        uint_8* Memory = PushMemory(StackAllocator, Size);
        DoStuffOnMemory(DoStuff, Memory, &AdditionalSpaceNeeded);
        Size = AdditionalSpaceNeeded;
        TotalSize += AdditionalSpaceNeeded;
    }
}

static void
AvoidMultipleAllocations(stack_allocator* StackAllocator, uint32_t Iterations, do_stuff* DoStuff)
{
    uint64_t TotalSize = 0;
    for(uint32_t I = 0; I < Iterations; I++)
    {
        uint32_t Size = DeriveNeededSizeForEachElement(DoStuff, I);
        TotalSize += Size;
    }

    //NOTE(enev): Here the default stack parameters are accepted!
    uint_8* Memory = PushMemory(StackAllocator, TotalSize);
    DoStuffOnTheWholeMemory(DoStuff, Memory);
}
```

Фиг. 2.1 Демонстриране на бързия път на програмата и как да избегнем много алокации

На горната фигурата е показан демонстрационен код, в който всички обхождания на цикъла освен тези, които заделят блокове, ще са изключително бързи. При допълнително знание за средата и кодът, ние може да пренапишем процедурата DoStuffOnMemory().

Нека оставим пренаписването настрана засега и да се фокусираме върху първата процедура, когато имаме заделен блок. Единствената проверка, която се прави, е дали той съществува и дали имам достатъчно място. След първата алокация знаем, че такъв има и то с подходящ размер. Следователно няма да се наложи да разширяваме прекомерно.

Остават следващите инструкции, през които ще трябва да преминем. И те са:

- извличане на базовия адрес на блока

- събиране на използваната досега памет в блока

- подравняване на стартовия адрес на алокацията

Тези три променливи се събират и единствено остава една помощна функция с минимален брой инструкции. Тя реално е извличането на подравняване.

При условие, че всички данни са налични освен подравняването, изискващо само маски и събиране, целият процес по алокиране се извършва в рамките на няколко цикъла. Имайки предвид архитектурата на модерните процесори, които позволят изпълнението на много операции на различни портове, тази операция е почти безплатна.

В сравнение с всички останали алокатори за общи цели е видно, че тук имаме предимство. При една обща стратегия на алокатор, когато се заделя памет, в зависимост от размера и ситуацията, която имаме, се итерира лист (в най-елементарния вариант на имплементация) от предишни подредени свободни блокове. Това се прави докато не намерим такъв, който да е точен по размер или по-голям. В случай на по-голям, блокът се разбива на две и едната част ни се предоставя, а останалите байтове ще се разположат отново на свързания списък. Също трябва да се обърне внимание, че има период, в който трябва да се намеси и хийп мениджъра. Той от своя страна използва стратегия за обединяване, защото имаме създадена фрагментация. Колкото и бърза да е тази имплементация, все пак е код, който трябва да се изпълни. Най-добре е да бъде елиминиран.

Освен това, при отсъствие на блок, ще се сблъскаме с операционната система, но по-лошата новина е, че не сме сигурни, кога точно ще стане това. Нямаме така демонстриран изричен достъп до размера и степента на запълване на блоковете и разпръснатостта на фрагментацията, както при стековия алокатор.

От всичко това можем да кажем, че дори да се налага да обработваме големи количества памет, стига да сме предвидили правилно размера на блока, ние няма да имаме затруднение, в код, който е ориентиран към производителност. Разбира се, тук говорим за нормална производителност.

Трябва да се има предвид, че ако се налага по-сериозна оптимизация, тогава е необходимо да се правят минимален брой заделяния дори от този така наречен бърз път. Ако попаднем в подобен тип ситуация, е добре да се разбие цикълът на два и да итерираме първо размера на общата памет. Следващата стъпка е да направим една единствена алокация и тогава вече започваме с натоварващите портовете на процесора изчисления. Това е демонстрирано в метода AvoidMultipleAllocations().

В такива ситуации е добре винаги първо да се профилира, преди да се променя каквото и да било, за да не се окаже, че имаме проблем с кеш паметта при изчисленията. Съвременните компилатори разполагат с много задълбочена оптимизационна фаза спрямо това каква процесорна архитектура за компилация е зададена.

Бавен път

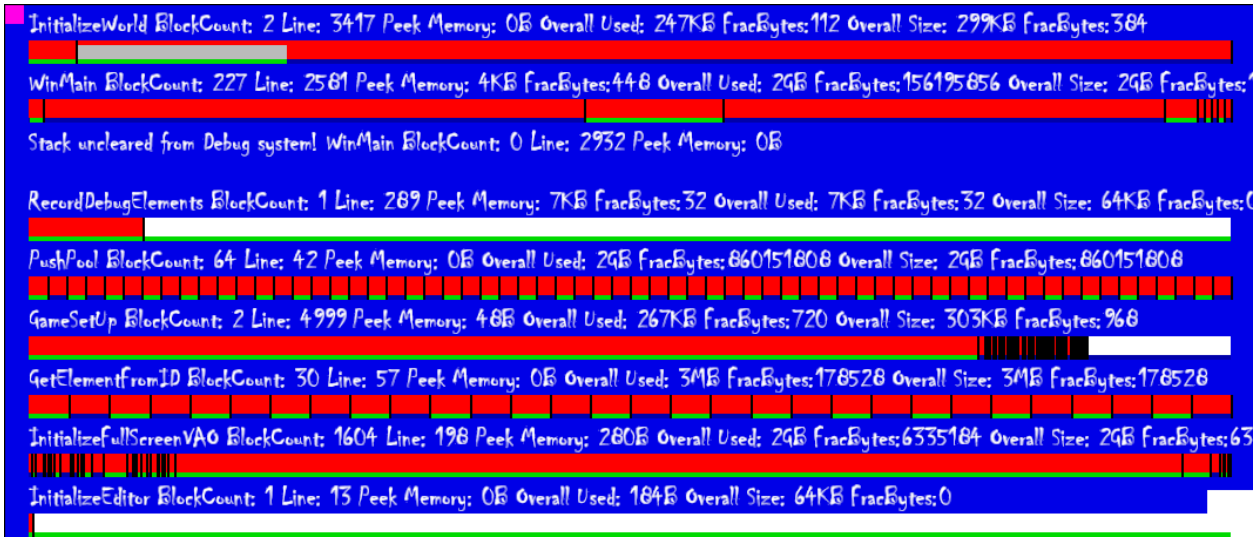
Можем да обърнем внимание и на случая, в който ще ни е необходим нов блок. Вече разяснихме, в какво се състои същността му. Нека да споменем кои части са най-податливи на проблеми с производителността. Това обикновено е частта между нашата софтуерната програма и комуникацията с операционната система. Необходимостта от валидации и така наречена смяна на контекст между процесите е нещо, което неизменно трябва да се има предвид, когато се комуникира с операционната система.

Глава Трета

Дизайн и разработка на визуализацията на стековия алокатор

10.1 Дизайн на визуализацията

Визуализацията е подготвена в отделен прозорец, в който се представят в реално време всички алокации налични към дадения момент (кадър) на изпълнение на програмата.



Фиг. 2.2 Цялостен изглед върху визуализацията

Всеки отделен стеков алокатор наличен в програмата се визуализира по следният начин:

Разделя се на два информационни реда.

В първия имплицитно се задава име, което е името на функцията, която първа е заделила памет. След името на стековия алокатор се представят и следните няколко характеристики:

- Blockcount (броя на блоковете)
- Line (линията на която е направена първата алокация)
- Peek memory (това е най-много заделената памет в кадър)
- Overall used (размерът на използваната до сега памет в алокатора)
- FragBytes (размерът на фрагментация на паметта на първата функция)
- Overall size (размерът на цялостно заделената памет от алокатора)
- FragBytes (размерът на цялостната фрагментация на алокатора)

Фиг. 2.3 Илюстриране на информационните редове във визуализацията.

Във вторият информационен ред се съдържа визуалната репрезентация на стека като правоъгълник(червено/бял). Под него се намира линия в различни цветове (син и зелен). Тази линия обозначава размера и местоположението на всеки един блок от паметта на алокатора.

Пример: Ако сме заделили три блока ще видим (зелен, син, зелен) цвят под правоъгълника обозначаващи точния размер и местоположение на блока. Правоъгълникът от своя страна ще бъде запълнен в червено в частта, която е зает, а останалата ще бъде в бяло (незаета). Върху правоъгълника са наложени черни линии, които обозначават краят на всяка една алокация, извършена в дадения блок.

Фиг. 2.4 Демонстрация на визуализацията на блокове.

Съществува още една особеност на визуализацията, която е представяне на фрагментацията на всяка една алокация. Тя също се представя като полупрозрачен правоъгълник, който е малко по-малък от основния правоъгълник. Фрагментацията се намира в края на всяка една алокация.

Фиг. 2.5 Примерна визуализация на фрагментация

На Фиг. 2.3 Ясно може да се види, че при заделяне втората алокация, се заделя в чисто нов блок, което означава, че мястото в предишният е било недостатъчно, което е илюстрирано с полупрозрачен правоъгълник, също на горният ред е изписано колко точно в байтове е фрагментацията. Тази фрагментация може да се елиминира, като се настрой размерът на блока, а не бъде оставен този по подразбиране. Тук се вижда също и колко блока са заделени. В случая са два, илюстрирани чрез зелен и син правоъгълник.

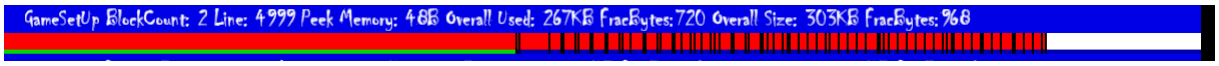
Също така визуализацията следи дали сме затворили всички временни блокове с памет, но ако искаме нашата временна памет да е постоянна (persistent) през кадри, или какъвто и да е интервал от време, визуализацията ще покаже като цифра разликата между изпълнените BeginTemporaryMemory(....) направените временни алокации и EndTemporaryMemory(....) затворените временни блокове.

Допълнителни визуализационни функции за улеснение на прегледа на състоянието на стековия алокатор

Приближение (Zoom-in and out on specific allocator) със скрол бутона на мишката.

Имаме полза и от функции като приближаване (zoom in/out), ако разполагаме със голям брой малки алокации може да приближим в определен участък и да видим как точно се случват и дали съществува фрагментация.

Фиг. 2.6 Илюстрираме zoom способностите на визуализацията.



Фиг. 2.7 Ясно е видимо, че чрез zoom може отчетливо да се различат всички алокации.

Вижда се, че вторият блок (синият) е пълен с различни малки алокации. Понякога в по комплексни програми може да не знаем какво причинява толкова много малки различни алокации. Затова визуализацията разполага с още един скрит инструмент. Това е покриване с мишката върху определения регион, за който не сме сигурни. От тази великолепна функция, ако не сме сигурни коя точно алокация се извиква десет пъти във втория блок от дясно, може да приближим и след това да отидем с мишката върху точното място, което е притеснително. Така ще посочим конкретна алокация и ще можем да видим дали е същата с предходната, какъв е нейният размер, къде се намира и от кой метод идва.

Фиг. 2.8 Специфична информация за всяка алокация, когато я покрием с мишката.

От фигурата става ясно, че метода AddWidget(), е основният виновник за тези малки алокации. Също така разбираме, че тази функционалност се намира в fromnothings_widget.cpp на линия 137 и освен това една от най-важните части е размерът, който сме поискали и подравнения размер, които сме получили. В случая, SizeAligned: 488 bytes и SizeInit: 480 bytes.

10.2. Разбор на имплементацията на визуализацията

Имаме основна част, която е и най-много обхождана и ще се спрем главно на нея, и ще избегнем подробности около изграждането на рендирането и потребителския интерфейс.

Първо да се спрем на основните структури използвани в имплементацията. Те са хеш таблици за по бърз запис на самите данни, и в последствие при подходящ размер лесно итериране. Първо как записваме самите данни. Използваме огромен

масив от така наречените събития. Събитие представлява едно от четири неща. Те са следните:

- Заделили сме памет.
- Започнали сме временна такава.
- Приключваме временна памет
- Изчистваме целия алокатор

Тези събития се записват с помощта на рапъри около самите функции на алокатора и когато бъдат повикани те отчитат съответното събитие с прикрепената към него информация като ред, файл, име на функция, идентификация на алокатор (от неговото местоположение и инстанция). Всяко събитие е важно да се отбележи, че съдържа и тип на събитието, за да може да бъде разпозната информацията, която носи.

Освен тази информация в събитието се записва и информацията за самата алокация. (поискан размер, получен размер, индекс на блока).

Цялото това записване започва от самия старт на програма и в последствие се обновява на всеки цикъл (кадър).

След като сме събрали огромния обем от информация, е време да започнем с анализа и чисто програмното изграждане на визуализацията.

Обхождаме всички събития като спрямо типа се насочваме каква информация да запазим. Като получим събитие за заделяне на памет, ние търсим дебъг структурата на алокатора в хеш таблицата. След като я извлечем, добавяме записа към останалите в масив с разширяване. Пазим и стек от събитията, които са започване на временна памет и приключването й, за да може да следим колко блока и какъв размер от последния блок от алокации се освобождават. Чрез това следене ние свиваме и разширяваме масива от алокации и когато настъпи частта с визуализацията информацията ще бъде прецизно подготвена.

Последното събитие, което ще разгледаме е събитието за цялостно изчистване на стековия алокатор. Към него се подхожда малко по внимателно, защото трябва да уточним, че ние сме откъсни от реалната структура на алокатора. Следователно не трябва да се забравя възможността да се случи изчистване. В такъв случай ние не трябва все още да изтриваме напълно дебъг структурата, защото ако в същия поток от събития отново пристигне ново заделяне, то тогава, ако прибързано изтрием структура, няма вече да разполагаме с данните за стековия алокатор, а само с неговата идентификация от последното събитие. Чрез нея няма да можем да открием нищо и не ще да отразим данните. Затова всички алокатори, които са били изчиствани се запазват в масив от указатели, който в последствие се итерира, за да се види дали няма добавена информация, ако има такава, то ще запазим дебъг структурата на алокатора, иначе ще я изтрием.

```
for(u32 Index = 0; Index < DebugSystem->EventCountForFrame; Index++)
{
    debug_event* Event = Events + Index;

    switch(Event->Type)
    {
        .................. Other types of events ..............

        case PushMemory:
        {
            debug_stack_recording* Recording = GetStackRecording(Event->StringID, Event->Allocation.CurrentStackInstance);
            //NOTE(enev): In here you can also do the non-debug version of the PushMemory to get memory for the record
            //This code is illustrative
            Record->Filename = Event->Filename;
            Record->FunctionName = Event->FunctionName;
            Record->LineNumber = Event->LineNumber;
            Record->StringID = Event->StringID;
            Record->Stats.SizeInit = Event->Allocation.SizeInit;
            Record->Stats.SizeAligned = Event->Allocation.SizeAligned;
            Record->Stats.CurrentStackInstance = Event->Allocation.CurrentStackInstance;
            //NOTE(enev): Also make sure you hold the current block and calcute necessary fragmentation
        }
        break;
        case BeginTempMem:
        {
            debug_stack_recording* Recording = GetStackRecording(Event->StringID, Event->Allocation.CurrentStackInstance);
            debug_trace Trace = {Recording->AllocationCount, Recording->BlockCount};
            AddToTempStack(&Recording->TempStack, Trace);
            TempCount++;
        }break;
        case EndTempMem:
        {
            debug_stack_recording* Recording = GetStackRecording(Event->StringID, Event->Allocation.CurrentStackInstance);
            debug_trace Trace = RemoveFromTempStack(&Recording->TempStack);

            Assert(Trace.AllocationCount <= Recording->AllocationCount);
            Assert(Trace.BlockCount <= Recording->BlockCount);
            //NOTE(enev): In here you can "free" the memory on a linked list or any other approach that you might want
            Recording->AllocationCount = Trace.AllocationCount;
            --TempCount;
        }break;
        case ClearStack:
        {
            debug_stack_recording* Recording = GetStackRecording(Event->StringID, Event->Allocation.CurrentStackInstance);
            //NOTE(enev): Attach on the waiting free list
        }break;

        .................. Other types of events ..............
    };

    //NOTE(enev): If new activity arrised after the free iterate to keep them, free the rest!
}
```

Фиг. 2.9 Илюстративен код на начина, по който се обхождат и обработват събития.

От кода по-горе се вижда най-важната част от имплементацията, засягаща управлението на събития в дебъг визуализациите. Обхождаме всички събития за даден цикъл (кадър) и съответно усвояваме получената информация.

Започваме с цикъл for(), обхождайки цялата хеш таблица от записи с указатели към алокатори, докато не намерим пълен слот. При пълен запис, започваме още един цикъл отново for(), но този път итерираме върху инстанциите породени от този слот. След като идентифицираме не празен слот отново, ние вече разполагаме с основната дебъг структура, в която държим всички записи за дадения алокатор. Започваме обхождането последователно. Първо от всички блокове и след това през всички алокации. Така, чрез предварително обработените данни, обхождането на всеки стек е по-бързо и улеснява изключително много рендирането, както и изобразяването на графичния интерфейс. Освен това линейната итерация позволява събирането на обзорни данни, като цялостно използвана памет, цялостно заделена и общата фрагментация.

```
static void
DebugShowStackAllocs(window_properties* Window, game_controller* Controller, layout* Layout, gpu_program GPU_PROGRAM, game_mouse* Mouse)
{
    if(!(Window->WFlags & WINDOW_HIDDEN))
    {
        stream WarningsStream = {};
        BeginCharacterStream(&WarningsStream);
        for(uint32_t Index = 0; Index < STACK_HASH_LENGTH; Index++)
        {
            for(stack_hash* Chain = StackHash[Index]; Chain; Chain = Chain->NextInHash)
            {
                Assert(Chain->CurrentCount != 0);
                for(uint32_t I = 0; I < INSTANCE_HASH_LENGTH; I++)
                {
                    for(stack_instance* InnerChain = Chain->InstanceHash[I]; InnerChain; InnerChain = InnerChain->Next)
                    {
                        //NOTE(enev): Here we have found a recording!
                        debug_stack_recording* Recording = &InnerChain->RecordedStack;

                        ......... Actual Visualization ......................

                        //NOTE(enev): Produce the warning stream in the most inner loop if any
                        //NOTE(enev): Do correct eventual warnings display
                    }
                }
            }
        }
        EndCharacterStream(&WarningsStream);
    }
}
```

Фиг. 3 Илюстрираме как да обходим двете хеш таблици така, че да намерим стековия алокатор

След като вече сме намерили стековия алокатор, трябва сега да се итерират всички блокове и алокации. При тази част възникват няколко отделни етапа. Първият е да осигурим структура, в която да се съхраняват общите статистики. След това е необходимо да подготвим позиционните данни на всеки малък провоъгълник на блок или алокация спрямо базовия на стековия алокатор. Свършили това, изчисляваме позицията и размера на фрагментацията като правоъгълник, разбира се, ако съществува такава. Налага се да направи и линейна интерполация върху тези данни за възможно най-добро позициониране. След което идва и един от последните етапи, който е да изпратим данните за рендиране в буферите на специфичната графична програма.

```
//NOTE(enev): On top calculate total statistics
for(uint32_t B = 0; B < BlockCount; B++)
{
    //NOTE(enev): Calculate position on base rect
    //NOTE(enev): Figure out position and alignment
    //NOTE(enev): Bunch of lerping!
    //NOTE(enev): Write render data
    for(uint32_t S = 0; S < AllocationCount; S++)
    {
        //NOTE(enev): Calculate position on base rect
        //NOTE(enev): Figure out position and alignment
        //NOTE(enev): Bunch of lerping!
        //NOTE(enev): Write render data
    }
}
```

Фиг. 3.1 Показваме какво е необходимо да се направи след като вече имаме дебъг структурата на стековия алокатор.

```
if(OnToolTip(Mouse, Rect, &WrapTagsAt.Rect, Window, Layout, GPU_PROGRAM, v4{0.0f, 0.0f, 0.0f, 1.0f}))
{
    //NOTE(enev): Get the correct language properties!
    float SpaceCharWidth = GetSetAttributes(ENGLISH, REGULAR, &Layout->TextAtlas->FontData)->SpaceCharWidth * UniformScale;
    //NOTE(enev): Display LineNumber, SizeInit, SizeAligned, Filename and Function Name
}
```

Фиг. 3.2 Илюстрираме предимствата на Immediate-mode graphical user interface

На фигурата е показано, че чрез ред ние успяваме да проверим дали сме в указания правоъгълник и чак след това създаваме прозорец, в който да покажем цялата необходима информация като местоположение, поискан и получен размер на алокация. Освен тези детайли вземаме предвид и отделните езици и шрифтове отново чрез само ред.

```
if(!MiddleDown) //NOTE(enev): Otherwise we scroll the window!
{
    if(IsMouseInRect(Mouse, StackRect))
    {
        Recording->ZoomInPX = MouseP.x - CurrentStackPos.x;

        //NOTE(enev): I want to zoom one forth per tick!
        float ScrollAmountHoriz = ScrollAmount * 0.25f;
        Recording->ZoomAmount += ScrollAmountHoriz;
        if(Recording->ZoomAmount > MaxZoom)
        {
            Recording->ZoomAmount = MaxZoom;
        }
        if(Recording->ZoomAmount < MinZoom)
        {
            Recording->ZoomAmount = MinZoom;
        }
    }
}
```

Фиг. 3.3 Базова имплементация на приближаване чрез мишка

От фигурата става ясно как имплементацията за приближение работи, макар не много добра, виждаме, че първо проверяваме дали не сме натиснали средния бутон. След това проверяваме дали попадаме върху базовия правоъгълник. Той илюстрира цялостното пространство на стековия алокатор. В случай, че сме върху него, ако имаме стойност в променливата ScrollAmount, т.е. сме завъртели средния бутон на мишката, приближаваме правоъгълника към нас с определеното количество. Тази имплементация може да се подобри значително.

Глава Четвърта

Бъдещо развитие, възможности за подобрение.

11.1 Възможност за цялостен снапшот (snapshot) на отделен интервал от време (брой кадри) на програмата.

В подобни софтуери за анализиране на паметта чрез визуализация има възможност и за покритие на определен период от време на цялостния цикъл на програмата. Този обем от информация е изключително голям и е много трудно да се държи в паметта на програмата. За тази цел този така наречен снапшот (snapshot) се записва в постоянната памет и след това се дава възможност за анализиране на тези данни, посредством друг софтуер, който може да използва подобни механизми, но вече върху определени постоянни статични данни. По този начин има възможност за анализиране и на временната памет по обстоен начин. Тя може да бъде прихванатата за всички стекови алокатори през всички кадри и след това да бъде прегледана алокация по алокация.

11.2 Последствен инструмент за задълбочен анализ на вече готовия снапшот

Този инструмент е подходящо да бъде отделен изпълним файл. И той да разглежда само и единствено снапшот-а. Може да се добави и опцията да се разглежда няколко снапшот-а, за да се открие разликата между двата и съответно, ако има изтичане то да бъде хванато.

11.3 Възможност за ограничение на фаталните грешки

Под това се има предвид, че ако нарочно изтича памет и то в големи количества това няма да може да се визуализира, и програмата ще спре с фатален край понеже са препълнени буферите за рендиране. Това може да се избегне като просто се изпише предупреждение за недостатъчно място и предупреждение за нужно разучаване на проблема, който е възникнал. По който и да е от двата начина ще може да се открие сравнително бързо, защото в дебъг режим на софтуера буфера все още се пази в паметта и ще може да го разгледаме със записаните данни и да установим, кой метод изразходва 5 пъти на кадър 128 байта памет.

11.4 Връщане на повече контекстна информация

Възможност за връщане на така наречените „тежки указатели“ (fat-pointers), които носят повече информация. Можем да дадем за пример обикновения низ, в повечето случай при него е задължително, той да идва с дължина, ако включим език и шрифт, това може да се окаже полезна информация.

При определени условия може да се направи архитектура, при която функцията за заделяне да връща по-голяма структура от информация. При заделяне на масив от някакъв тип може да се върне начален и краен адрес и да се прави проверка при достъп до такава структура чрез предефиниран оператор [] в дебъг

версията на софтуерната програма. При нормално изпълнение ще използваме само указателя от цялата структура.

```
//NOTE(enev): Now we can overload the [] operator on Arr and use it to check in debug mode the array boundaries.
array_structure* Arr = PushArray(StackAllocator, type, Count);

template<typename T>
struct array_structure
{
    uint8_t* BaseAddress;
    uintptr_t TypeSizeInBytes;
    uintptr_t LastUsedIndex;
    uint64_t Count;

    array_structure& operator[](uintptr_t Index)
    {
        LastUsedIndex = Index;
        uintptr_t BytesToJump = LastUsedIndex * TypeSizeInBytes;
        uint8_t* EndAddress = BaseAddress + BytesToJump;
        if(EndAddress < BaseAddress ||
                EndAddress > (BaseAddress + Count * TypeSizeInBytes))
        {
            Assert(!"Invalid access");
        }
        return (T*)EndAddress;
    }
};
```

Фиг. 3.3 Илюстрация на пробна имплементация за проверка на валидност на масив.

Тук ясно е показана необходимата информация за достъп. Това са 4 променливи.

- Базов адрес
- Размер на типа
- Последно използван индекс
- Брой елементи в масива

Първо изчисляваме колко памет трябва да „прескочим“, след което намираме крайния адрес и го валидираме срещу базовия, който сме запазили още при заделяне. Ако имаме неуспешна валидация, съответно ще хвърли грешка, което е най-желателното нещо. Странно, но това е правилният подход. При успешна проверка просто ще върнем крайния адрес под типа, който сме получили в началото.

Заключение:

Основната цел на дипломната работа се състоеше в това да изградим стабилна имплементация на стеков алокатор. Освен това ние се насочихме и към имплементирането на система за неговата визуализация. Тя ни е необходима, за да бъдем абсолютно прецизни при отстраняването на възникнали програмни грешки, като изтичане на памет и фрагментация. Смятам, че след като успешно се запознахме с разработката, ние вече може да изградим такъв тип алокатор и знаем как да го използваме в почти всички ситуации. Успешно предоставихме необходимите методи и структури за безотказната работа на неговата визуализация. Чрез нея можем да разгледаме цялостната памет на програмата и да забележим съответните проблеми, като изтичане на памет и фрагментация, ако въобще те са налични. Чрез използването на immediate-mode графичен интерфейс ние успяхме по лесен и достатъчно бърз начин да покажем необходимия интерфейс. Във визуализацията бяха разположени всички налични стекови алокатори изпълняващи се по време на работата на една софтуерна програма. Спряхме се на рендиране чрез програмния интерфейс на OpenGL, поради неговата многоплатформеност. Както стековият алокатор, така и визуализацията са успешно вградени в собствен софтуерен продукт, работещ безотказно. Основните задачи на дипломната работа са разрешени посредством няколко анализа. Успяхме да усвоим 3-те основни характеристики за изграждането на такъв тип алокатор. Имплементирахме методи по начин позволяващ многоплатформеност и широка реализация. Достигайки до крайният етап, ние покрихме основните аспекти на алокатора, като работа с временна памет, разширяемост и заделяне на памет.

Благодарности:

Изказвам сърдечна благодарност на научния си ръководител гл. ас. д-р Христо Христов за проявената подкрепата по време на разработката на дипломната работа. Също така, особено съм благодарен на Casey Muratori за огромния принос за осъществяването на имплементацията на стеков алокатор чрез проекта Handmade Hero.

В учебния проект Handmade Hero може да намерите не само неговата добра имплементация (в случая, той го нарича арена вместо стек (Arena)), но и много други различни теми като рендиране, 3D графики, осветяване на сцени, детекции на колизии.

Използвана литература:

[1] Kernighan и др. The C Programming Language. 2nd. Englewood Cliffs, NJ, Prentice Hall, March 1988. ISBN 0-13-110362-8. Архив на оригинала от 2008-11-06 в Wayback Machine.

[2] Handmade Hero - https://handmadehero.org/ (2014 – 2021)

[3] Casey Muratori - https://caseymuratori.com/blog_0001 - Immediate-mode graphical user interfaces (2005)

[4] Sean Barret - https://nothings.org/ - Creator of stb_libraries

[5] IBM C++ Template documentation https://www.ibm.com/docs/en/zos/2.4.0?topic=only-class-templates-c

Референции:

Valgrind – съществуващ инструмент за анализиране на паметта на програмата https://www.valgrind.org/

stb_libraries – съществуващи библиотеки за различни цели, една от които е справянето с извличането на информация и рендирането на шрифтове в текстура. https://github.com/nothings/stb

OpenGL – графична библиотека за изпълнение на сложни графични програми. https://www.khronos.org/opengl/