КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра теорії та технологій програмування

Кваліфікаційна робота на здобуття ступеня бакалавра
за спеціальністю 122 «Комп'ютерні науки»
на тему:

РОЗРОБКА БАГАТОПОТОКОВОГО АЛОКАТОРА ПАМ'ЯТІ БЕЗ ЗАСТОСУВАННЯ МЕХАНІЗМУ БЛОКУВАННЯ

Виконав студент 4-го курсу
Кушніренко Олександр Сергійович

Науковий керівник:
доцент, кандидат фіз.-мат. наук
Ставровський Андрій Борисович

Консультант:
доцент, кандидат технічних наук
Ткаченко Олексій Миколайович

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент

Роботу розглянуто й допущено до захисту на засіданні кафедри теорії та технологій програмування
«01» червня 2022 р., протокол № 10
Завідувач кафедри
Микола НІКІТЧЕНКО

Київ — 2022

РЕФЕРАТ

Обсяг роботи 50 сторінок, 15 ілюстрацій, 2 таблиці, 18 джерел посилань.

БАГАТОПОТОКОВИЙ АЛОКАТОР, БЛОКУВАННЯ, РОЗПОДІЛ ПАМ'ЯТІ, ПОДВІЙНЕ ВОЛОДІННЯ, ФРАГМЕНТАЦІЯ, АТОМАРНІ ОПЕРАЦІЇ

Об'єктом роботи є вивчення процесу розподілення пам'яті в сучасних багатопотокових операційних системах та користувацьких програмах. Предметом роботи є програмний засіб для розподілу пам'яті.

Метою роботи є розробка багатопотокового алокатора без застосування блокувань та порівняльний аналіз розробленого алокатора з іншими сучасними розподільниками.

Методи розроблення: принципи багатопотоковості, методи розподілу пам'яті, дизайн сучасних систем розподілу пам'яті. Інструменти розробки: середовище розробки Visual Studio 2019, мова програмування C++ за стандартом C++14.

Результати роботи: виконано загальний огляд існуючих систем розподілу пам'яті та їх архітектури, вивчено методи розподілу пам'яті, принципи багатопотоковості та проблеми, що виникають при багатопотоковому програмуванні. Розроблено багатопотоковий алокатор без блокувань, який можна застосовувати в користувацьких програмах для розподілу пам'яті. Проведено порівняльний аналіз ефективності розробленого алокатора з сучасними комерційними алокаторами.

Розроблений алокатор може застосовуватися для розподілу пам'яті в користувацьких програмах на мові C та C++.

ЗМІСТ

ВСТУП

Оцінка сучасного стану об'єкта розробки. Динамічний алокатор — один із найважливіших компонентів застосунків, які написані на мовах програмування що використовують ручний розподіл пам'яті (C, C++ або Rust). Процес розподілу пам'яті вимагає від розробника явно виділяти та звільняти пам'ять для програми. Протягом виконання програми блоки постійно виділяються та звільняються, тому задача динамічного алокатора пам'яті — надавати блоки на запит, вести їх облік та здійснювати керування. На сьогоднішній день більшість програм є багатопотоковими, що вимагає від алокатора правильно та швидко обробляти запити для різних потоків програми. Іншими словами, сучасний алокатор повинен бути потокобезпечним, що природно ускладнює його дизайн. Для гарантування потокобезпечності багато алокаторів використовують структури даних, які базуються на використанні блокувань, але це впливає на швидкість роботи та гнучкість. Сучасні підходи до дизайну алокаторів пам'яті намагаються мінімізувати кількість блокувань використовуючи часткові блокування, зменшуючи частоту використання блокувань або синхронізуючи виділення пам'яті за допомогою примітивів або використовуючи буфери пам'яті, що призначені до конкретних потоків.

Зазвичай, навіть при використанні часткових блокувань, алокатор все ще має багато недоліків у порівнянні з алокаторами без застосування замикань: існує вразливість до взаємного блокування, змінення стану потоку, інверсії потокових пріоритетів та затримок через випередження під час блокування. Вони також не здатні впоратись з неочікуваним завершенням потоку через неможливість відпустити блокувальний ключ, що може призвести до втрати ресурсу. Альтернативним рішенням для досягнення потокобезпечності є синхронізація потоків без використання блокувань. Відсутність блокувань

гарантує загальне просування процесу хоча б одним з потоків. Це також захищає алокатор від повної зупинки за великої кількості потоків у порівнянні з кількістю машинних ядер.

Актуальність роботи та підстави для її виконання. За означенням, алокатор, який не використовує блокувань, не може отримувати винятковий доступ до ділянки пам'яті або ресурсу, стійкий до взаємних блокувань та зміни стану потоку, не може визвати інверсію пріоритетів через відсутність затримок через інші потоки, стійкий до неочікуваного завершення потоку.

Ще одна перевага неблокуючого алокатора — алгоритми та структури даних які за дизайном не використовують блокувань але потребують динамічно виділеної пам'яті зберігають неблокуючу властивість та можуть використовуватись без втрати швидкості. Неблокуючий аллокатор є фундаментом для продуктів які будуються на принципах відсутності блокувань для структур та алгоритмів.

Для того, щоб неблокуючі аллокатори були актуальними та широко використовувались, їх продуктивність повинна бути не гірша ніж у сучасних блокуючих або частково блокуючих алокаторів пам'яті. На сьогоднішній день, блокуючі алокатори мають швидкість порядку 12 нс, лінійно масштабуються та дають змогу отримувати більшу продуктивність від більшого числа ядер, навіть якщо вони залежать від операцій самого алокатору. Останні дослідження показують, що на сьогоднішній день алокатори займають близько 7 відсотків від усіх циклів процесорів у багатопотокових системах. Тому конкурентоспроможність нових динамічних алокаторів пам'яті напряму впливає на ширину їх використання в корпоративних програмних продуктах.

Мета й завдання роботи. Метою кваліфікаційної роботи є розробка багатопотокового алокатора без застосування блокувань та порівняння роз-

робленого програмного засобу з сучасними багатопотоковими алокаторами. Для досягнення цієї мети поставлено такі завдання:

- Дослідити предметну область: принципи багатопотоковості, методи розподілу пам'яті.

- Дослідити існуючі багатопотокові алокатори та методи їх реалізації.

- Розробити дизайн алокатора.

- Реалізувати програмний засіб за архітектурою.

- Провести порівняльний аналіз застосунку з обраними реалізаціями алокаторів.

Однією із цілей цієї роботи є демонстрація можливостей неблокуючого аллокатора в багатопотокових системах не жертвуючи швидкістю у порівнянні з еталонними алокаторами. Можна ідентифікувати ключові особливості дизайну:

- наявність потокових буферів, які дозволяють виділяти та звільняти блоки пам'яті для багатьох без використання блокування;

- глобальна «купа» — компонент який управляє суперблоками пам'яті з доступних сторінок пам'яті;

- мапа сторінок — компонент який зберігає метадані сторінок.

Результати досліджень показали, що неблокуючий алокатор здатен зрівнятися з алокаторами які використовуються у сучасних розподілених мультипотокових системах.

Об'єкт, методи й засоби розробки. Об'єктом роботи є розробка сучасного багатопотокового алокатора без застосування блокувань. В якості

інструменту було обрано Visual Studio 2019 — інтегроване середовище (IDE), мова програмування — C++ за стандартом 14-го року.

Можливі сфери застосування. Алокатор може використовуватись у програмах написаних на мові C або C++, включаючи комерційні програми та користувацькі застосунки.

РОЗДІЛ 1 ПРИНЦИПИ БАГАТОПОТОКОВОСТІ

З появою багатопроцесорних систем з постійно зростаючою кількістю процесорів, фокус операційних систем перемістився на продуктивність та масштабованість на шкоду використанню пам'яті. Багатопотокове виділення пам'яті ставить на меті синхронізацію процесу розподілу між декількома потоками, при цьому підтримуючи високий рівень ефективності та масштабованості.

Кожен процес в операційній системі може мати декілька одиниць виконання, які називаються потоками. Потоки можуть виконуватись паралельно на різних процесорах чи ядрах, при цьому маючи той самий адресний простір що й батьківський процес. Потоки які працюють над спільними даними, доступними до запису, повинні використовувати синхронізаційні примітиви [2] для того, щоб переконатись в коректності роботи програми. Використання пам'яті, доступної до запису, без використання синхронізації може призвести до проблем, такі як доступ за правом першості (*race condition*)[1]. Надалі зробимо огляд основних принципів та проблем, що виникають при роботі з багатопотоковими середовищами.

1.1 Атомарні операції

Атомарні операції представляють собою інструкції, які не можуть бути перервані планувальником операційної системи та виконуються майже миттєво у баченні інших процесорів. Це означає, що така операція може бути виконана за один процесорний такт. Атомарні операції відрізняються на різних процесорних архітектурах та не все атомарні операції доступні.

Основні атомарні операції на архітектурі x86-64:

- Порівняти та поміняти місцями (Compare and swap, CAS)

- Перевірити та змінити значення (Test and set)

- Витягнути значення та збільшити (Fetch and add)

- Завантажити значення (Load Linked, LL)

- Зберегти за умовою (Store Conditional, SC)

За допомогою атомарних операцій можна створювати примітиви для синхронізації, наприклад блокування.

1.2 Блокування

Блокування (***lock***) — синхронізаційний примітив взаємного виключення, який дозволяє отримати взаємовиключний доступ до ресурсу одним з потоків. Для того, щоб заволодіти ресурсом, потік повинен отримати блокування (*acquire lock*). Якщо ресурс більше не потрібен, потік може відпустити блокування (*release lock*), даючи доступ до ресурсу іншим потокам. Якщо блокування вже отримано, інші процеси не можуть його отримати та не мають доступу до ресурсу.

Як саме блокування припиняє роботу потоку залежить від реалізації. Одним з варіантів може бути нескінченний цикл у спробі отримати блокування (*spinlock*). Іноді блокування може явно призупинити (*suspend*) потік.

Використання блокувань є простим за принципом та гарантує ексклюзивне використання ресурсу. Однак, інші потоки що намагаються отримати ресурс повинні чекати. Це може сильно впливати на ефективність та швидкість виконання програми.

Необережне використання блокувань може призводити до взаємного блокування (*deadlock*) або змінення стану потоку після невдачі отримати блокування (*livelock*). Використання блокувань може призводити інверсії пріо-

ритетів потоків, коли потік з більшим пріоритетом чекає на потік з меншим пріоритетом (*lock contention*), та може бути замінений у процесі. Врешті-решт, блокування може зупинити процес виконання всієї програми, якщо процес що отримав блокування завершився в результаті винятку.

РОЗДІЛ 2 КЛАСИЧНІ МЕТОДИ РОЗПОДІЛУ ПАМ'ЯТІ

У цьому розділі ми обговоримо роль та сформулюємо основні принципи розподілу пам'яті. Ці поняття є усталеними в алокаторах [1] й використовуються і по цей день.

2.1 Роль розподільника пам'яті

Роллю динамічного алокатора є відстеження зайнятих та вільних частин пам'яті та можливість виконувати запити керівничої програми. Всього доступно два типи запитів: запит на виділення блоку пам'яті заданого розміру та запит на звільнення виділеного блоку пам'яті. алокатор не знає заздалегідь потік запитів та їх розмір.

Загалом, алокатор гарантує наступні інваріанти для керівничої програми:

1) Результат запиту безповоротний — блок пам'яті, переданий до програми, не може бути змінено чи повернуто.

2) Виділені блоки не можуть бути змінені алокатором.

3) Тільки звільнені блоки можуть бути застосовані чи змінені алокатором.

У свою чергу, керівнича програма або користувач має гарантувати:

1) Виділений блок має бути звільнений тільки один раз.

2) Блоки не можуть бути змінені чи застосовані після їх звільнення.

[1] Тут і надалі терміни алокатор, динамічний алокатор та алокатор пам'яті є взаємозамінними

Недотримання користувачем цих гарантій призводить до загальновідомих проблем — порушення першого пункту призводить до подвійного звільнення [3], порушення другого пункту — до використання блоку після звільнення [4], [7].

2.2 Фрагментація

Фрагментація пам'яті — фундаментальна проблема в області управління пам'яттю. Фрагментація означає, що програма передбачає потребу в більшій кількості пам'яті для задоволення послідовності запитів, ніж сума пам'яті, необхідної для цих запитів поодинці (див. Рис. 1). Була винайдена велика кількість алгоритмів для розміщення необхідних блоків у пам'яті, кожен з яких має різний рівень фрагментації та ефективність. З точки зору використання пам'яті, хороший алокатор — той, який зменшує обсяг пам'яті, необхідний для виконання послідовності запитів, тобто зменшує об'єм фрагментованої пам'яті [5].

Фрагментація буває двох типів — ***внутрішня*** та ***зовнішня***.

2.2.1 Внутрішня фрагментація

Внутрішня фрагментація — це різниця між розміром поверненого блоку пам'яті та розміром запитуваного блоку. В основному внутрішня фрагментація відповідає блокам, розмір яких менший за допустимий розмір відстеження цієї ділянки, наприклад, які менше ніж одне машинне слово. Якщо накладні витрати на відстеження блоку пам'яті перевищують сам блок, його легше віддати разом з запитуваним блоком. Розмір внутрішньої фрагментації варіюється відповідно до розміру блоків, які підтримує алокатор [5].

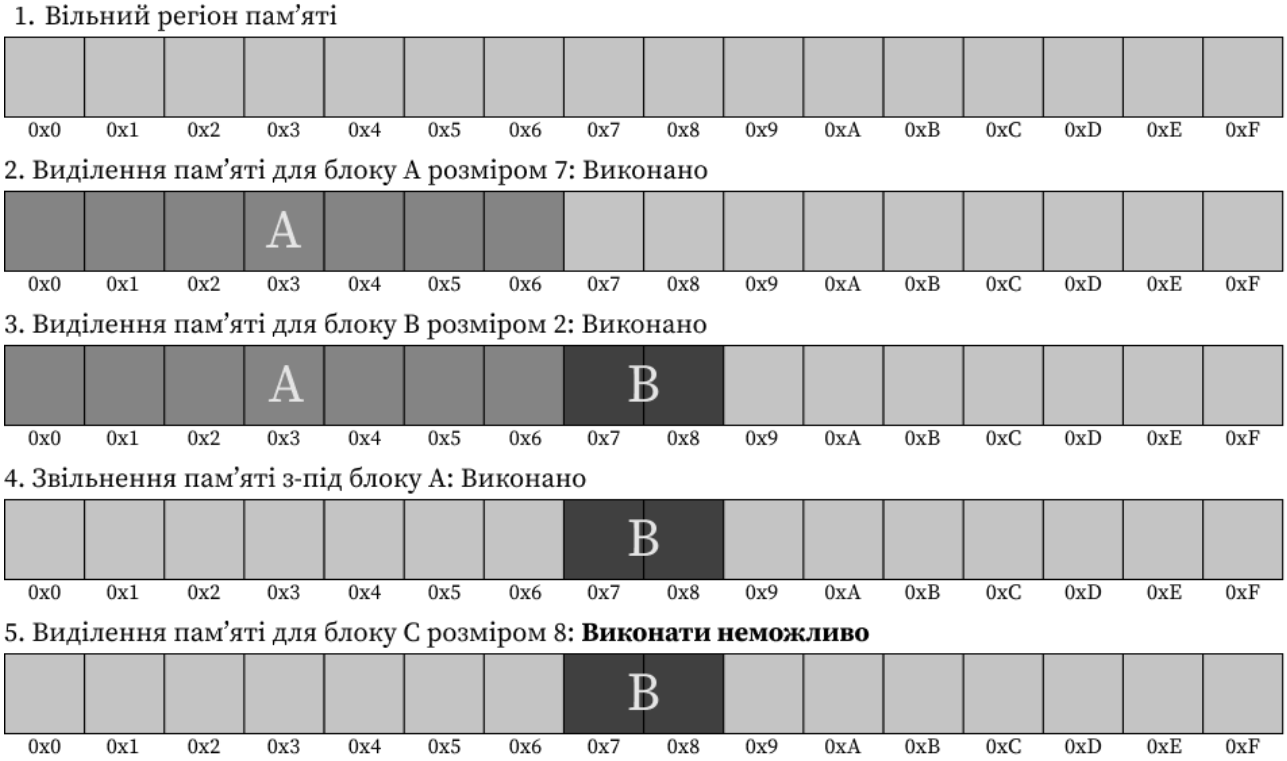


Рисунок 1 — Приклад фрагментації пам'яті

2.2.2 Зовнішня фрагментація

З***овнішня фрагментація*** враховує вільні блоки пам'яті, які загалом не підходять для подальшого використання у виконанні запитів через те, що вони занадто малі або логічно недоступні. На відміну від внутрішньої фрагментації, найгірший випадок зовнішньої фрагментації не можна передбачити, він може бути визначений лише при детальному аналізі алокатора. Robson [7] показує, що в найгіршому випадку обсяг втраченої пам'яті в будь-якому алгоритмі дорівнює $M \log_2 n$, де M — розмір пам'яті у використанні, n — співвідношення між найменшим та найбільшим блоками пам'яті в оброблених запитах.

2.2.3 Накладні витрати

Розподілення пам'яті у загальному виді представляє собою NP-повну комбінаторну проблему [8]. Навіть знаючи потік запитів заздалегідь, алгоритм не може гарантувати мінімальну фрагментацію чи її відсутність. Однак, потік генерується не випадково а є результатом поведінки програми. Таким чином, програма може проявляти закономірності в запитах які можуть бути використані для спрощення задачі та часткового її вирішення:

1) Програми потребують лише кілька розмірів блоків пам'яті [5]. У таблиці нижче наведено яку кількість різних блоків пам'яті використовують програми у відсотковому співвідношенні до всіх запитів.

Програма	**90%**	**99%**	**99,9%**	**100%**
GCC	5	12	254	641
Perl	10	27	60	96
GhostScript	7	85	344	589
MinGW	6	44	82	238
Середнє	7	42	185	391

2) Блоки пам'яті, запити на виділення яких є близькими у часі, зазвичай звільняються в один і той самий час [5].

Алокатор має бути масштабованим, тобто мати лінійне зростання ефективності роботи з точки зору накладних витрат, рівня фрагментації та швидкості зі збільшенням кількості потоків.

2.3 Алгоритми розподілу пам'яті

Алгоритм розміщення блоків у пам'яті є важливою частиною стратегії динамічного алокатора, яка впливає на його ефективність з точки зору

швидкості виконання запитів та розміру фрагментації, викликаної вибраною стратегією.

Деякі з основних алгоритмів, що використовувались в ранніх алокаторах, називаються послідовними підгонками. У них алокатор використовує список чи іншу відповідну структуру зберігання вільних блоків. Під час запиту на виділення, вибирається блок з даного списку за обраною стратегією [11], [12].

Перша підгонка

Під час запиту на виділення пам'яті, алгоритм ***першої підгонки*** повертає перший блок підхожого розміру. Якщо блок більший за запитуваний розмір, він розбивається на два блоки, менший з яких повертається до списку [11].

Блок, отриманий через процес розбиття або запит на звільнення повертається до списку, використовуючи одну з стратегій: у початок списку (*LIFO*), в кінець списку (*FIFO*), або за порядком адреси повертаємого блоку. Впорядкований за адресою список гарантує, що суміжні у пам'яті блоки є суміжними у списку. Це може бути використано для швидшого об'єднання. Найгірший та найкращий випадки для першої підгонки добре вивчені [10], [11].

Наступна підгонка

Наступна підгонка виступає як більш ефективний варіант першої підгонки, де пошук починається з елементу, на якому закінчився попередній пошук. Таким чином зменшується середній час пошуку. Загалом, наступна підгонка викликає більше фрагментації, ніж перша підгонка.

Найкраща підгонка

Найкраща підгонка шукає блок мінімального розміру який задовольняє запит на виділення пам'яті. Принципом найкращої підгонки є мінімізація розміру фрагментів отриманих з розбиття. Таким чином, найкраща підгонка зменшує зовнішню фрагментацію, але може збільшувати внутрішню фрагментацію чи створювати занадто малі блоки для подальшого використання. Аналогічно до першої та наступної підгонки, найкраща підгонка має три різні стратегії поверненню блоку пам'яті. Зокрема, варіант упорядкування за адресою показує найкращу ефективність [10], [11].

Найгірша підгонка

Найгірша підгонка є протилежністю до найкращої підгонки — завжди шукається найбільший вільний блок. Зокрема, якщо список блоків впорядкований, такий варіант збільшує швидкість пошуку доступного блока. Однак на практиці найгірша підгонка не використовується, бо показує погану ефективність та зменшує кількість вільних блоків [10].

2.4 Розподілені списки

Розподілені списки — стратегія, яка використовує набір списків з блоками, зазвичай фіксованого розміру. Кожен блок може належати лише одному списку. При запиті на виділення, виділяється блок пам'яті з найбільш підхожого списку блоків. Зазвичай, кількість різних списків визначається кількістю різних розмірів для блоків, або класів розмірів (зазвичай це степені двійки) [5]. Варто зазначити, що блоки пам'яті у різних списках фізично не розділені. При застосуванні політик об'єднання та розбиття можливе об'єднання суміжних блоків пам'яті в один блок. Далі розглядаємо варіації реалізації розподілених списків.

Розподілене сховище

При використанні варіанту розподіленого сховища, усі запити можуть бути оброблені використовуючи один загальний список (див Рис. 2). Алгоритм має наступний вигляд:

- Під час запиту на виділення блоку пам'яті виконується пошук відповідного списку з найменшим розміром, що перевищує наданий запит. Якщо знайдений список пустий, посилається запит до операційної системи на отримання нових блоків у відповідний список.

- Під час запиту на звільнення отриманий блок кладеться у відповідний за розміром список.

Зазначимо, що спроб на об'єднання чи розбиття під час операцій над блоками пам'яті не здійснюється. Це може призводити до важкої фрагментації, коли блоки одних розмірів майже не використовуються, а блоків інших розмірів завжди не вистачає. Досліди [5] показують, що при неправильному підборі класів розміру, внутрішня фрагментація перевищує внутрішні витрати на процес об'єднання та розбиття.

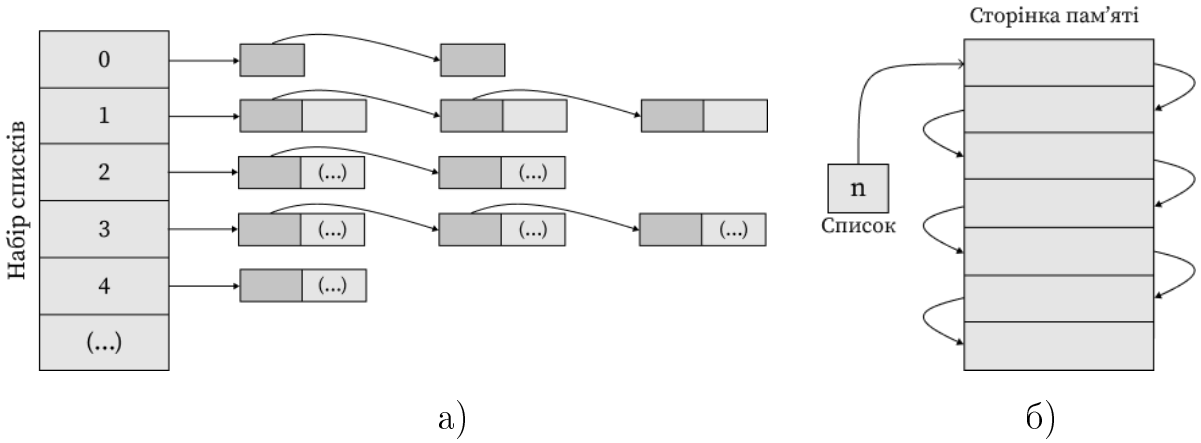
Незважаючи на це, розподілене сховище залишається найбільш використовуваною стратегію зберігання вільних блоків у сучасних динамічних алокаторах. Можна виділити наступні переваги такого підходу:

1) **Локальність** — блоки одного розміру згруповані разом, що може бути узгоджено з періодичностями які зустрічаються у багатьох програмах.

2) **Швидкість** — отримання або звільнення блоку без урахування можливості запиту до операційної системи зводиться до пошуку відповідного списку у сховищі, довжина якого визначається кількістю класів розміру.

3) Має **передбачувану внутрішню фрагментацію**, яка залежить від вибраних класів розміру, відсутня зовнішня фрагментація.

Структура конкретного списку може мати різні представлення. В одній реалізації список представляється вказівником на перший блок, де перше машинне слово зберігає вказівник на наступний блок. В інших реалізаціях використовується масив заданого статичного або динамічного розміру, щоб уникнути виконання записів у самих блоках.



а) б)

Рисунок 2 — Візуалізація розподіленого сховища: а) набір списків; б) розміщення блоків у списку

Розподілені списки з об'єднанням

На відміну від розподіленого сховища ця стратегія дозволяє об'єднання та розбиття блоків пам'яті під час пошуку блока відповідного розміру:

1) Під час запиту на виділення блоку пам'яті виконується пошук відповідного списку з найменшим розміром, що перевищує наданий запит. Якщо список пустий, пошук продовжується до першого непустого списку. Знайдений блок розбивається (за умови, що остача може бути покладена назад до іншого списку), результуючий блок повертається а остача кладеться назад до сховища.

2) Під час запиту на звільнення, отриманий блок кладеться у відповідний за розміром список. Далі виконується спроба об'єднати новий блок з суміжними вільними блоками, якщо такі присутні у сховищі. Іноді об'єднання відкладається до певного моменту для покращення ефективності операції звільнення.

Розподілені списки з об'єднанням використовують найкращу підгонку або першу підгонку для пошуку списку [5].

Розподілені списки за об'єктами

Ця стратегія є частковим випадком розподіленого сховища, яка замість класів розміру використовує типи об'єктів для того, щоб визначити який список використовується. В такому випадку ми маємо один вільний список на один тип об'єкту, таким чином може бути декілька списків з блоками однакового розміру, якщо вони призначені до різних типів. Розподілення за об'єктами найбільш розповсюджене для алокації об'єктів операційної системи [13], однак іноді використовується і для звичайного розподілення пам'яті при наявності статичного аналізу. Варіації алокаторів, глибоко інтегрованих в програмне середовище, можуть пришвидшити процес створення та знищення об'єктів. Загалом, розподілені списки за об'єктами не можуть бути використані як загальний механізм розподілення пам'яті та замінити C інтерфейс, але можуть бути корисними для певних видів програмного забезпечення.

РОЗДІЛ 3 ІНТЕРФЕЙС РОЗПОДІЛУ ПАМ'ЯТІ

Операційна система надає інтерфейс для розподілу віртуальної пам'яті розробникам програмного забезпечення. Однак, різні мови програмування задля уніфікації процесу розподілу та можливості змінити реалізацію впроваждують свої інтерфейси, які вже використовуються користувацькими програмами.

3.1 C Malloc API

Інтерфейс розподілу пам'яті у мові C є загальновживаним та представляє собою набір функцій у стандартній бібліотеці. Він визначає, яким чином програма виділяє та звільняє пам'ять [9].

```
1  // виділяє блок пам'яті розміром не менше size байтів
2  void* malloc(size_t size);
3  // звільняє блок пам'яті, який був попередньо виділений
4  void* free(void* ptr);
5  // виділяє блок пам'яті розміром не менше size * n байтів
6  void* calloc(size_t n, size_t size);
7  // виділяє блок пам'яті не менше size байтів, намагаючись використати при
       цьому попередньо виділений блок ptr
8  void* realloc(void* ptr, size_t size)
9  // виділяє блок пам'яті не менше size, який вирівняний використовуючи
       alignment. alignment повинен бути степенем двійки
10 void* aligned_alloc(size_t alignment, size_t size);
11 // повертає кількість дозволених до використання байтів у блоку ptr
12 size_t malloc_usable_size(void* ptr);
```

З опису параметрів функцій та їх призначень можна бачити, що інтерфейс накладає певні обмеження на реалізацію динамічного алокатора. Наприклад, `free` та `malloc_usable_size` передбачають, що алокатор має змогу визначити розмір виділеного блоку лише за вказівником. Це значить, що алокатор має зберігати інформацію про властивості блоку або в самому блоці,

або во внутрішній пам'яті алокатора, що збільшує внутрішню фрагментацію або витрати пам'яті відповідно.

3.2 C++ Memory API

Інтерфейс розподілу пам'яті у мові C++ представляється операторами `new` та `delete`, які зазвичай використовують стандартну реалізацію `malloc` та `free`. Оператору виділення та звільнення можна перевизначити для застосунку або конкретного класу, тим самим змінивши механізм розподілу пам'яті.

```
// виділяє блок пам'яті розміром не менше count байтів. При невдачі, кидає виняток std::bad_alloc.
void* operator new(std::size_t count)
// Виділяє пам'ять для масиву розміром не менше count байтів. При невдачі, кидає виняток std::bad_alloc.
void* operator new[](std::size_t count);
// Звільняє блок пам'яті ptr, який був попередньо виділений за допомогою new.
void* operator delete(void* ptr) noexcept;
// Звільняє блок пам'яті ptr, який був попередньо виділений за допомогою new[]
void* operator delete[](void* ptr);
```

C++ API розділяє керування пам'яті для об'єктів та масивів об'єктів. Воно також дозволяє самостійно визначати механізм керування пам'яттю.

3.3 Windows Memory API

Для того, щоб отримати блок пам'яті з операційної системи, програма має виконати системний виклик [9]. Інтерфейс для розподілу пам'яті на платформі Windows знаходиться в заголовку `<memoryapi.h>`, та нараховує багато функцій. Далі опишемо лише ті системні виклики, які будуть використовуватись для розподілу пам'яті сторінок для алокатора.

```
// Виділяє або змінює стан віртуальної пам'яті розміру dwSize для викликаючого процесу. Під час виділення пам'ять автоматично занулюється.
LPVOID VirtualAlloc(LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect)
// Виділяє або змінює стан віртуальної пам'яті розміру dwSize для вказаного процесу hProcess, Під час виділення пам'ять автоматично занулюється.
LPVOID VirtualAllocEx(HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect)
// Звільняє регіон пам'яті lpAddress розміру dwSize для викликаючого процесу.
BOOL VirtualFree(LPVOID lpAddress, SIZE_T dwSize, DWORD dwFreeType)
// Звільняє регіон пам'яті lpAddress розміру dwSize для вказаного процесу hProcess
BOOL VirtualFreeEx(HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize, DWORD dwFreeType)
```

3.4 Linux Memory API

Операційна система Linux має свій інтерфейс для розподілу пам'яті, який знаходиться в заголовку `<sys/mman.h>`

```
// Виділяє пам'ять розміру length, починаючи з offset байтів з початку файлу.
void* mmap(void* start, size_t length, int prot, int flags, int fd, off_t offset)
// Повертає блок пам'яті start розміру length до операційної системи.
int munmap(void* start, size_t length)
```

Системні виклики до операційної системи є повільними у порівнянні з роботою програмного динамічного алокатору, теж не можуть бути використані напряму. Однак, операційна система більш ефективно виділяє та звільняє пам'ять для великих блоків (більше розміру сторінки).

РОЗДІЛ 4 РОЗПОДІЛ ПАМ'ЯТІ В МУЛЬТИПОТОКОВОМУ СЕРЕДОВИЩІ

Розподіл пам'яті в багатопотоковому середовищі стикається з труднощами, які обумовлені одночасними запитами на виділення та звільнення блоків пам'яті. Сучасні алокатори пам'яті повинні мати можливість обслуговувати багато потоків, не збільшуючи при цьому зайві витрати на обслуговування та багато часу на синхронізацію.

4.1 Подвійне володіння

Подвійне володіння (***false sharing***) — феномен, коли два або більше процесів мають блоки пам'яті, доступні для запису, на одній кеш лінії (див. Рис. 3). Це призводить до погіршення ефективності, тому що кеш лінія потребує оновлення на всіх процесорах, що використовуються для виконання цих потоків.

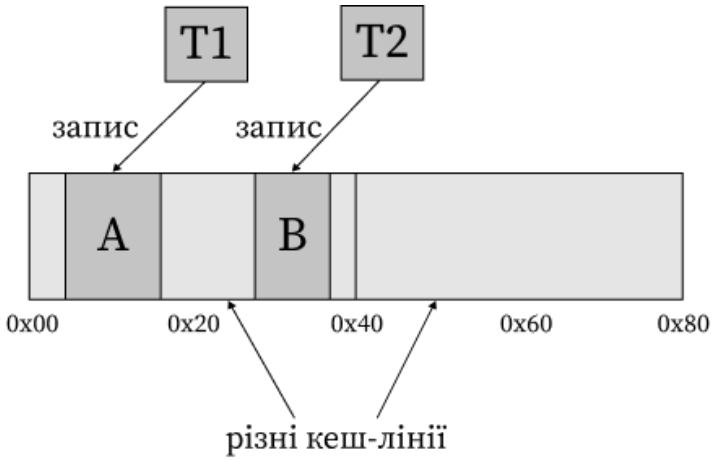


Рисунок 3 — Візуалізація феномену подвійного володіння

Варто зазначити, що феномен подвійного володіння не виникає з пам'яттю, яка доступна тільки для читання, така як код програми або стек.

алокатор може впливати на частоту виникнення подвійного володіння активно або пасивно. Активний вплив відбувається за умови, що алокатор

може виділити пам'ять з однієї кеш лінії багатьом процесорам [15]. Пасивний вплив може бути за моделі виробник-споживач, коли один потік виділяє пам'ять та передає її іншому потоку, який в свою чергу її звільняє, даючи можливість отримати блоки пам'яті з однієї кеш лінії під час наступного виділення [14], [15].

алокатор може збільшити мінімальний розмір блоку до розміру кеш лінії, але зазвичай її розмір набагато більше ніж довжина машинного слова (наприклад, 8 байтів та 64 байти), що призведе до різкого збільшення використання пам'яті.

4.2 Надмірне використання пам'яті (blowup)

Надмірне використання пам'яті — окремий вид фрагментації, який виникає при розподілі пам'яті без синхронізації між процесами, та найбільш поширений у випадку, коли кожен потік має приватний алокатор. Він визначається збільшенням споживання пам'яті потоком, коли алокатор, що отримує запит на звільнення пам'яті не може використати її у майбутніх запитах. [15] показує, що в ранніх версіях алокаторів це могло призводити до необмеженого використання пам'яті. Модель виробник-споживач сприяє надмірному використанню пам'яті, коли один потік надсилає запити на виділення пам'яті та передає її іншому потоку, який, в свою чергу, її звільняє. Якщо синхронізація між алокаторами не відбувається або відбувається з великою затримкою.

Сучасні алокатори запобігають необмеженому використанню пам'яті встановивши ліміт на максимальну кількість блоків. Після досягнення цього ліміту, алокатор віддає блоки у операційну систему або глобальну «купу».

4.3 Арени

Арени — техніка для зменшення випадків подвійного володіння за рахунок збільшення накладних витрат на пам'ять та збільшення фрагментації [15], [16]. За принципом, арена — повністю самостійний алокатор який підтримує багатопотокову алокацію та деалокацію за запитом від декількох потоків. Так як кількість потоків в процесі не відома завчасно, кількість потоків що використовують одну арену може збільшуватись чи зменшуватись з часом (див. Рис. 4). Головний алокатор повинен назначати арени до потоків з ціллю мінімізувати ефект подвійного володіння.

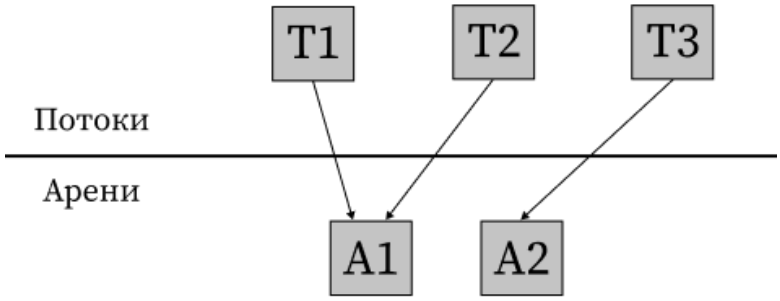


Рисунок 4 — Арени

Виділяють декілька стратегій для вибору арени для нового потоку:

- арена знаходиться за хешованим індексом потоку [15];

- вибирається арена з найменшою кількістю потоків [16];

- вибирається не зайнята чи не заблокована арена [17].

Використання арен вимагає від алокатора зберігати інформацію про арену в кожному виділеному блоку пам'яті для того, щоб коректно його звільнити (уникнувши пасивного подвійного володіння).

4.4 Потокові буфери

Потокові буфери дозволяють виконувати запити на виділення та звільнення пам'яті майже без синхронізації. За принципом, кожен потік має приватний список блоків пам'яті, який має виконувати його запити. Якщо кількість блоків впала нижче певного значення, до операційної системи або загального механізму подається запит на більшу кількість блоків. Якщо кількість блоків перевищила певне значення, частина блоків вивільняються чи передаються іншому буферу (див. Рис. 5).

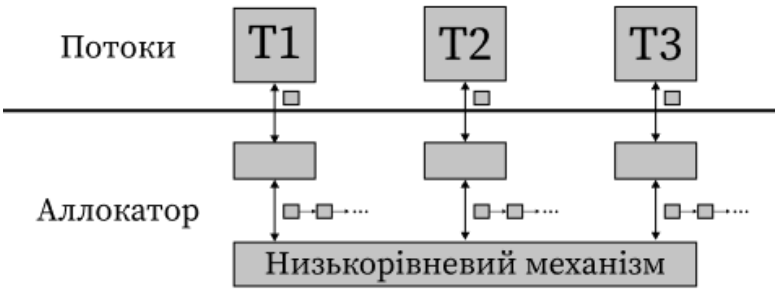


Рисунок 5 — Потокові буфери

Застосовуючи потокові буфери, потік взаємодіє лише з власним буфером а не з усім алокатором. Таким чином, алокатор може зменшити кількість синхронізацій до операцій між алокатором та буферами.

4.5 Сучасні алокатори пам'яті

Цей підрозділ робить огляд існуючих та розповсюджених алокаторів та наводить ряд особливостей кожного з них.

Hoard

Hoard — багатопотоковий алокатор, запропонований Berger [15]. Він

адресує проблеми подвійного володіння та необмеженого використання пам'яті, тим самим збільшуючи швидкість виконання запитів та роботи програми.

Hoard використовує набір «куп», по одній «купі» на кожен процесор, та глобальну «купу», як основне сховище пам'яті. Пам'ять зберігається у вигляді суперблоків — послідовного набору сторінок з якого відокремлюються блоки відповідного розміру.

Виділення відбувається через запит до відповідної «купи» та виділення блоку з відповідного суперблоку. Звільнення потребує пошуку суперблоку з якого був вирізаний блок з подальшим їх об'єднанням, якщо це можливо.

Ptmalloc2

Ptmalloc2 — стандартизований алокатор, який використовується разом з бібліотекою `glibc`, яка є найбільш розповсюдженою на Linux системах [17].

Ptmalloc2 використовує теги на межі блоку пам'яті, зберігаючи метадані, внаслідок чого кожне виділення пам'яті коштує одне додаткове машинне слово. алокатор використовує арени або потокові буфери, в залежності від налаштувань компіляції. У випадку арен, алокатор притримується новітньої стратегії — вибирається не заблокована арена, або створюється нова. Таким чином, кількість арен змінюється з часом.

Jemalloc

Jemalloc — сучасний алокатор, який намагається уникати фрагментації та фокусується на підтримці масштабування при збільшенні кількості потоків [16]. Jemalloc може використовувати як арени так і потокові буфери.

При використанні арен, на початку роботи програми Jemalloc створює в чотири рази більше арен ніж кількість ядер. Потоки призначаються за круго-

вим принципом, тобто потік отримує арену з найменшою кількістю активних потоків.

При використанні потокових буферів, Jemalloc використовує розподілене сховище для кожного буферу, де список — це масив динамічного розміру кожний елемент якого посилається на вільний блок пам'яті. Таким чином алокатор не зберігає дані у самих блоках, що дозволяє розмічати виділені блоки заздалегідь.

На сьогоднішній час, Jemalloc показує одне з найкращих співвідношень ефективності до використання пам'яті, що робить його ідеальним алокатором для багатопотокових програм.

NBMalloc

NBMalloc — академічний алокатор без блокувань, який базується на архитектурі алокатора Hoard. NBMalloc використовує набори суперблоків, з яких відокремлюються блоки певного класу розміру.

Щоб керувати суперблоками, для кожного потока створюється буфер, який взаємодіє з глобальною купою. Коли надходить запит на виділення блоку, вибирається потоковий буфер, який відокремлює блок необхідного розміру з суперблоку. При надходженні запиту на звільнення, блок повертається до відповідного суперблоку.

NBMalloc використовує велику кількість операцій при кожному запиті, що робить його повільним у порівнянні з сучасними алокаторами.

РОЗДІЛ 5 ДИЗАЙН ТА РЕАЛІЗАЦІЯ ВЛАСНОГО АЛОКАТОРА

Цей розділ описує дизайн та деталі реалізації багатопотокового алокатора без застосування блокувань. Алокатор передбачає використання 64-х бітну архітектуру і використовує мову C++.

Перерахуємо основні вимоги, які повинен виконувати алокатор:

1) **Швидкість** — алокатор повинен виконувати запити на виділення та звільнення пам'яті на рівні сучасних розповсюджених алокаторів.

2) **Масштабованість** — ефективність алокатора повинна збільшуватись лінійно при зростанні кількості одночасних потоків.

3) **Відсутність блокувань** — алокатор не має використовувати блокування, обмежившись атомарними операціями.

4) **Низький рівень фрагментації** — алокатор використовує мінімум пам'яті необхідної для виконання своєї роботи та має низьку внутрішню фрагментацію.

5) **Уникнення подвійного володіння** — алокатор не повинен активно чи пасивно виділяти пам'ять під різні потоки на одній кеш лінії.

5.1 Огляд структури

На верхньому рівні алокатор складається з трьох основних компонентів (див. Рис. 6):

1) **Потокові буфери** — компонент, який дозволяє звести запити потоку до взаємодії з потоковим буфером, уникаючи необхідності у синхронізації. Під кожен потік створюється свій потоковий буфер.

2) **«Купа»** — загальне сховище суперблоків, з яких потім відокремлюються блоки відповідних розмірів. З одного суперблоку можна відокремити блок лише певного класу розміру. Потокові буфери взаємодіють з «купою», передаючи блоки до «купи» чи забираючи блоки для заповнення. «Купа» може бути використана одночасно декількома потоками.

3) **Мапа сторінок** — структура, яка зберігає метадані про кожну сторінку, яка використовується суперблоками. Мапа дозволяє зменшити кількість пам'яті, яка використовується для зберігання про блоки.

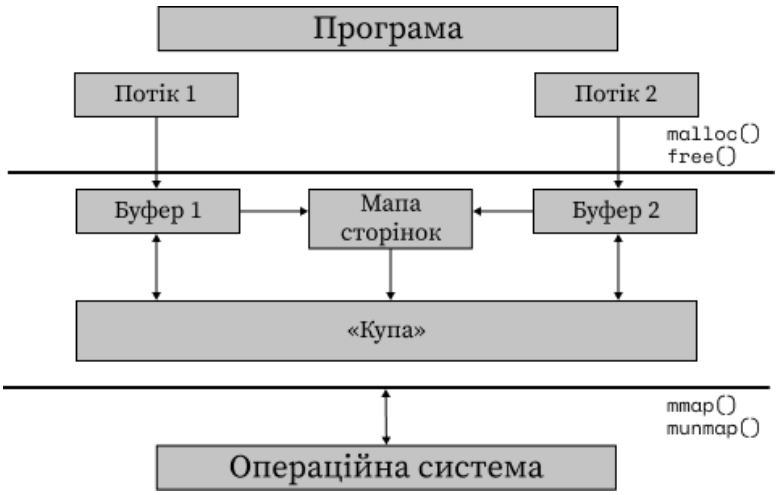


Рисунок 6 — Огляд структури

На рисунку 7 зображені основні сутності програми та зв'язки між ними та зовнішніми акторами — користувацькою програмою та операційною системою.

Кожен потік при створені неявно ініціалізує потоковий буфер в області статичної пам'яті. Потоки працюють зі спільною захищеною від запису ділянкою пам'яті, де й розташований програмний код алокатора.

Потоки взаємодіють лише з інтерфейсом розподілу пам'яті мови C, який перевизначає алокатор.

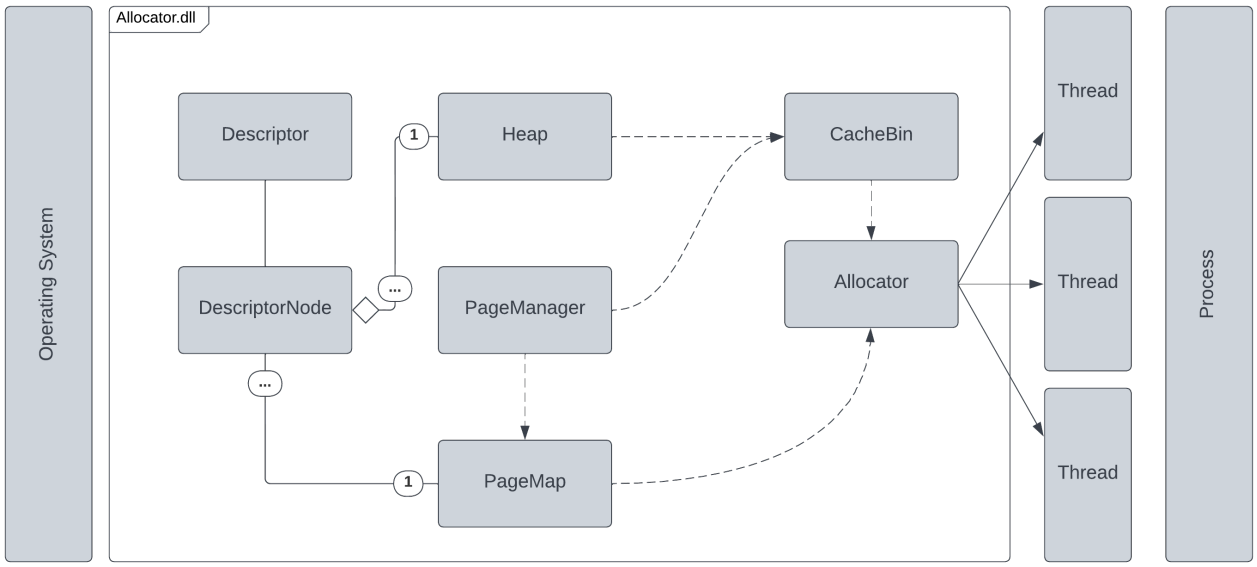


Рисунок 7

Наступний псевдокод демонструє реалізацію основних функцій виділення `malloc()` та звільнення `free()`:

```
void* malloc(size_t size)
{
  if (size > MAX_SIZECLASS)
  {
    void* SuperBlock = AllocateLargeBlock(size);
    Status status = CreateStatus(SuperBlock, FULL);
    Descriptor* desc = CreateDescriptor(status);
    return SuperBlock;
  }
  size_t Index = GetSizeClass(size);
  CacheBin* cache = GetCache(Index);
  if (cache->IsEmpty())
  {
    cache->Fill(Index);
  }
  return cache->PopBlock();
}
void free(void* ptr)
{
```

```
20   if (IsLargeBlock(ptr))
21   {
22     DeallocateLargeBlock(ptr);
23     return;
24   }
25   size_t index = GetSizeClass(ptr);
26   Cache* cache = GetCache(index);
27   if (cache->IsFull())
28   {
29     cache->Flush(index);
30   }
31   cache->PushBlock(ptr);
32 }
```

5.2 Класи розміру

Багато сучасних алокаторів використовують ***класи розміру*** (степені двійки або сума степенів двійки, див. Табл. 1), що дозволяє зменшити зовнішню фрагментацію та обмежити кількість списків у розподіленому сховищі. Велика кількість класів зменшить внутрішню фрагментацію за рахунок невеликого збільшення накладних витрат алокатора.

Правила генерації класів розміру була адаптована з алокатора Jemalloc [16]:

1) 2^X

2) $2^X + 2^{(X-2)}$

3) $2^X + 2^{(X-1)}$

4) $2^X + 2^{(X-1)} + 2^{(X-2)}$

Варто зазначити, що деякі розміри не підійшли через вирівнювання по машинному слову.

Таблиця 1 — Класи розміру

Клас розміру	**Формула**	**Розмір блоку (байт)**
1	2^3	8
2	2^4	16
3	$2^4 + 2^3$	24
4	2^5	32
5	$2^5 + 2^3$	40
6	$2^5 + 2^4$	48
7	$2^5 + 2^4 + 2^3$	56
...	...	...
37	2^{13}	8192
38	$2^{13} + 2^{11}$	10240
39	$2^{13} + 2^{12}$	12288
40	$2^{13} + 2^{12} + 2^{11}$	14336
41	2^{14}	16384

Найбільший клас розміру дорівнює 16 кілобайтам. Великі за розміром запити на виділення, алокатор буде напряму передавати операційній системі, тоді як невеликі запити буде виконувати за допомогою потокових буферів.

Щоб швидко отримати розмір відповідного класу розміру, при першому зверненні ініціалізується таблиця пошуку.

Класи розміру визначаються статично за допомогою макросу:

```
// номер класу, група, дельта, кількість сторінок
#define CALCULATE_SIZE_CLASS(index, group, delta, deltanumber, pages) \
  {(1U << group) + (deltanumber << delta), pages },

SizeClass SizeClasses[MAX_SIZECLASS_INDEX] =
{
  {0, 0}, // розмір. збережений для визначення суперблоків
  CALCULATE_SIZE_CLASS(1, 3, 3, 0, 1)   \
  CALCULATE_SIZE_CLASS(2, 3, 3, 1, 1)   \
  CALCULATE_SIZE_CLASS(3, 3, 3, 2, 3)   \
  CALCULATE_SIZE_CLASS(4, 3, 3, 3, 1)   \
  CALCULATE_SIZE_CLASS(5, 5, 3, 1, 5)   \
```

```
  CALCULATE_SIZE_CLASS(6, 5, 3, 2, 3)   \
  CALCULATE_SIZE_CLASS(7, 5, 3, 3, 7)   \
  CALCULATE_SIZE_CLASS(8, 5, 3, 4, 1)   \
};
```

5.3 Потокові буфери

Потокові буфери — компоненти, які дозволяють виконувати запити на виділення та звільнення пам'яті без синхронізаційних примітивів. Реалізацією буферу є розподілене сховище, яке використовує набір списків, кількість яких дорівнює кількості класів розміру (див. Рис. 8). Для реалізації у рамках цієї роботи було прийнято рішення використовувати однозв'язний список, де перше слово блоку посилається на наступний блок. Це співпадає з реалізацією купи, яка використовує однозв'язний список блоків у суперблоку.

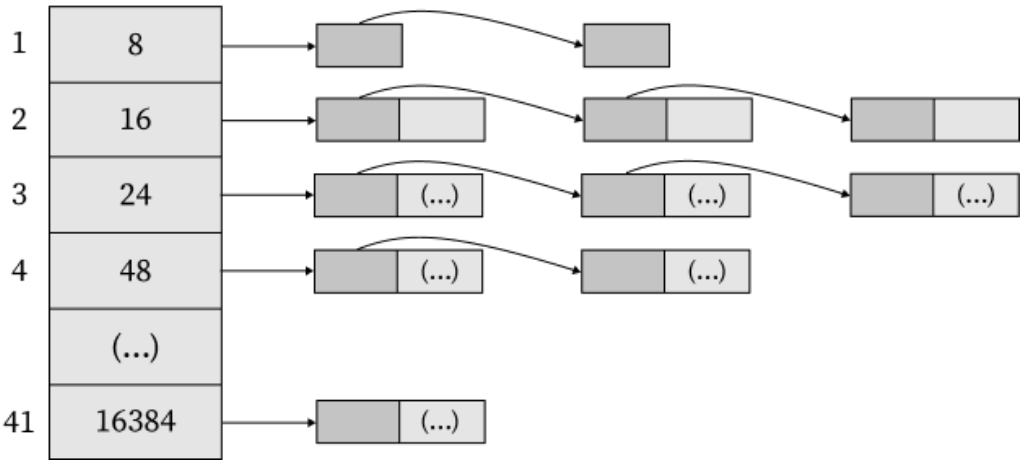


Рисунок 8 — Потокові буфери

Потоковий буфер має наступний зовнішній інтерфейс:

```
class CacheBin
{
public:
  // звільнити блок пам'яті
  void PushBlock(char* block);
  // виділити блок пам'яті
```

```
  char* PopBlock();
  // звільнити певну кількість блоків пам'яті,
  // використовується для звільнення суперблоків
  void PushList(char* block, uint32_t length);
  // виділити певну кількість блоків пам'яті,
  // використовується для звільнення суперблоків
  void PopList(char* block, uint32_t length);
  // заповнити відповідний клас розміру блоками пам'яті
  void Fill(size_t sizeClassIndex);
  // звільнити відповідний клас розміру
  void Flush(size_t sizeClassIndex);

  FORCEINLINE char* GetBlock() const { return block; }
  FORCEINLINE uint32_t GetBlockNum() const { return blockNum; }
};
```

5.4 «Купа»

«Купа» — компонент без блокувань, який керує суперблоками через застосування дескрипторів. Купа відповідає за взаємодію з операційною системою та заповнення або звільнення потокових буферів.

5.4.1 Дескриптори

Дескриптор — повторно використовуваний об'єкт, який описує властивості суперблоку та відстежує його статус. Дескриптор також містить статус, який займає одне машинне слово (64 біт), що дозволяє виконувати з ним атомарні операції CAS та SC.

Оголошення структури дескриптору та статусу:

```
struct Status
{
  size_t State: 2;     // FULL = 0, PARTIAL = 1, EMPTY = 2
  size_t Avail: 31;    // Індекс першого вільного блоку
  size_T Count: 31;    // Кількість вільних блоків
```

```
};
struct Descriptor
{
  Status status;    // статус
  char* SuperBlock;   // вказівник на суперблок
  size_t Size;      // розмір кожного блоку у суперблоці
  size_t MaxCount;    // максимальна кількість блоків
  size_t SizeClassIndex;  // індекс, який вказує на розмір у масиві
   розмірів.
};
```

5.4.2 Суперблоки

Суперблоки — послідовний набір сторінок, які розбиваються на блоки пам'яті однакового розміру. Суперблок може перебувати в трьох різних станах:

1) **Повний** — кожен блок з суперблоку знаходиться в буферах або використовується програмою.

2) **Частковий** — частина блоків суперблоку є вільною.

3) **Вільний** — суперблок складається лише з вільних блоків та може бути повернутий до операційної системи.

Рисунок 9 демонструє структуру суперблоку та його дескриптор.

Діаграма на рисунку 10 показує життєвий цикл блоку, отриманого з операційної системи.

5.5 Мапа сторінок

Мапа сторінок складається з метаданих про використовувані суперблоки. Зберігання метаданих про сторінку є більш ефективним з точки зору

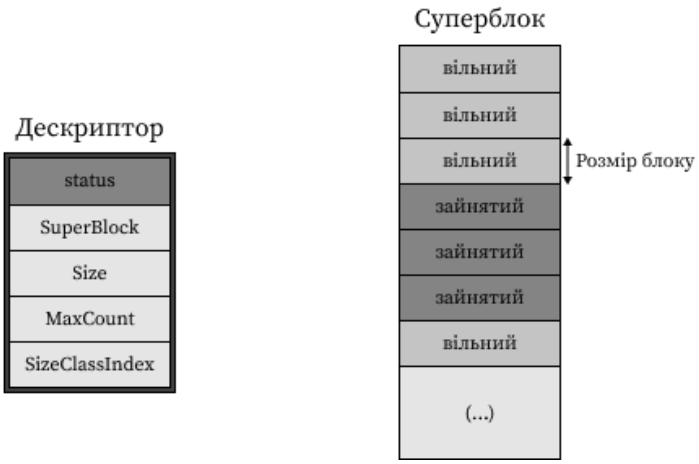


Рисунок 9

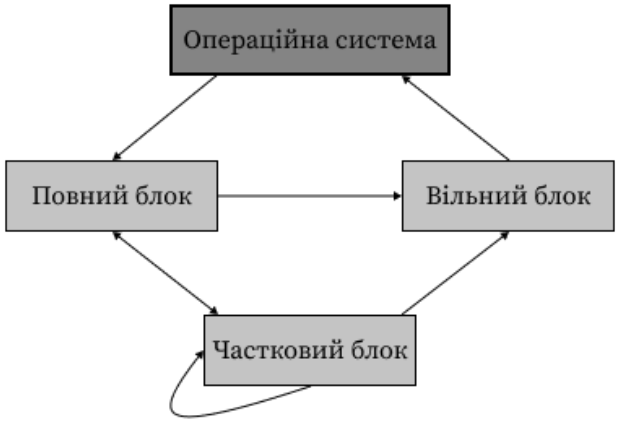


Рисунок 10

витрат на пам'ять ніж зберігання в кожному виділеному блоку пам'яті. Також це відділяє пам'ять, яку застосовує алокатор від пам'яті, яку використовує програма, що збільшує локальність даних.

Мапа сторінок реалізована у вигляді масиву, розмір якого залежить від розміру адресного простору, тобто 2^{36} на 64-бітній архітектурі.

5.6 Взаємодія з операційною системою

Взаємодія з операційною системою зводиться до запитів на виділення або звільнення суперблоків, з яких купа відокремлює списки блоків певного розміру. Зазначимо, що системні виклики дуже повільні, тому їх кількість

потрібно звести до мінімуму. Частота запитів напряму залежить від розміру суперблоків. Чим більший розмір, тим менше системних викликів та більші витрати пам'яті. Згідно досліджень [15], [17] оптимальним розміром суперблоку є 2 МБ.

API операційної системи Windows відрізняється від API Linux системи. Для того, щоб зробити алокатор кросплатформенним необхідно визначати тип платформи на етапі статичного або динамічного лінкування до програмного застосунку. На допомогу прийдуть платформо-специфічні символи, або макроси:

1) `_WIN32 || _WIN64` — операційна система Windows.

2) `__linux__` — операційна система Linux.

3) `__APPLE__` — операційна система IOS або OS X.

За допомогою наступного коду визначається платформа, на якій запускається програма на етапі лінковки:

```
#if defined(_WIN32) | defined(_WIN64)
  #define PLATFORM_WINDOWS
#elif defined(__linux__)
  #define PLATFORM_LINUX
#elif defined(__unix__)
  #define PLATFORM_UNIX
#elif defined(__APPLE__)
  #define PLATFORM_APPLE
#endif
```

Приклад запитів виділення та звільнення суперблоку на платформах Windows та Linux:

```
// виділення суперблоку
void* Page::Allocate(size_t size)
```

```
{
#ifdef PLATFORM_WINDOWS
  // код на платформі Windows
  void* ptr = VirtualAlloc(nullptr, size, MEM_COMMIT | MEM_RESERVE,
   PAGE_READWRITE);
  return ptr;
#elif PLATFORM_LINUX
  // код на платформі Linux
  void* ptr = mmap(nullptr, size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
   MAP_ANON, -1, 0);
  if (ptr == MAP_FAILED)
  {
    return nullptr;
  }
  return ptr;
#endif
}
// звільнення суперблоку
void Page::Free(void* ptr, size_t size)
{
#ifdef PLATFORM_WINDOWS
  // код на платформі Windows
  bool bResult = VirtualFree(ptr, size, MEM_DECOMMIT);
  ASSERT(bResult != 0);
#elif PLATFORM_LINUX
// код на платформі Linux
  int retval = munmap(ptr, size);
  ASSERT(retval == 0);
#endif
}
```

РОЗДІЛ 6 ПОРІВНЯЛЬНИЙ АНАЛІЗ

В цьому розділі демонструється порівняльний аналіз розробленого алокатора з іншими сучасними алокаторами, описаними в попередніх розділах. Аналіз був зроблений по таким показникам як: *швидкість*, *масштабованість*, *рівень фрагментації*, *кількість використаної пам'яті*, *подвійне володіння* та *надмірне використання пам'яті*.

В якості контрольного показника для швидкості та масштабованості була використана програма ***Threadtest***. Для оцінки подвійного володіння були використані застосунки ***cache-thrash*** та ***cache-scratch***. Кожен застосунок приймає кількість потоків в якості аргументу разом з іншими аргументами та повертає час виконання або пропускну здатність на одиницю часу. Варто зазначити, що бенчмарки статично зв'язуються з алокаторами на етапі компіляції.

Далі наведено список алокаторів з якими проводились порівняння. З них можна виділити дві групи з точки зору реалізації: алокатори без блокувань, такий як NBMalloc; сучасні алокатори які застосовують блокування, такі як Hoard, Ptmalloc2 та Jemalloc.

Середовище на якому проводився аналіз — багатопотоковий процессор архітектури x86-64, який має 24 логічних ядра, 64 ГБ оперативної пам'яті, операційна система Windows 10 Pro.

6.1 Опис роботи бенчмарків

Threadtest запускає вказану кількість незалежних потоків, кожен з яких виконує групу з 100 тисяч запитів на виділення та 100 тисяч запитів на звільнення, блоки звільняються в тому ж порядку що й виділяються. Кожен потік виконує 100 груп запитів. На вихід threadtest подає статистику за вико-

ристанням пам'яті, включаючи фрагментацію та потреби алокатора, середню кількість виконаних викликів за одиницю часу та ефективність масштабованості.

Cache-thrash вимірює наскільки алокатор сприяє активному подвійному володінню. Він створює задану кількість незалежних потоків, кожен з яких виділяє блоки пам'яті розміром від 16 до 32 байтів. Далі виконується 100 тисяч операцій запису до виділеної пам'яті та звільнення. Кожен потік виконує від 500 до 1000 виділень.

Cache-scratch вимірює наскільки алокатор сприяє пасивному подвійному володінню. Він починає з виділення фіксованої кількості блоків розміру 16 байтів в основній програмі та передає їх у користування іншим потокам. Далі він створює задану кількість потоків, які звільняють блоки, передані від основної програми. Далі кожен потік слідує алгоритму роботи ***cache-thrash***, виконуючи від 500 до 1000 виділень.

6.2 Ефективність та масштабованість

Рис. 11 демонструє швидкість на запусках від 1 до 24 потоків. Усі представлені нижче результати є середнім значенням з п'яти запусків.

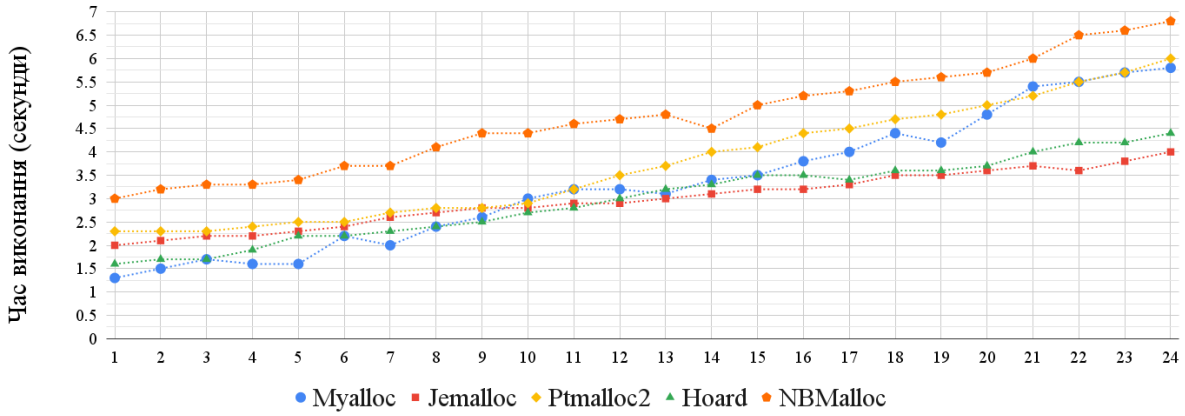


Рисунок 11 — Швидкість

Графік показує, що алокатор трохи швидший за сучасні аналоги Jemalloc та Hoard на малій кількості потоків, але зі збільшенням їх кількості має більш зростаючу тенденцію, ніж Jemalloc, та наближається до стандартного Ptmalloc2. Варто зазначити, що алокатор NBMalloc, який не використовує блокувань, показує значно гіршу швидкість на будь якій кількості потоків.

Рис. 12 демонструє масштабованість на запусках від 1 до 24 потоків. Алокатор демонструє схожу ефективність що й сучасні алокатори.

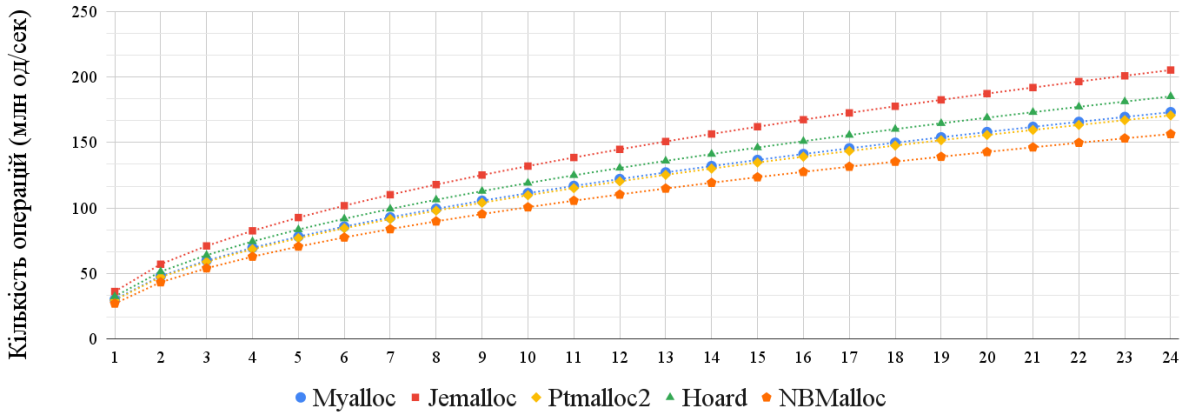


Рисунок 12 — Масштабованість

Рис. 13 демонструє кількість використаної пам'яті на запусках від 1 до 24 потоків.

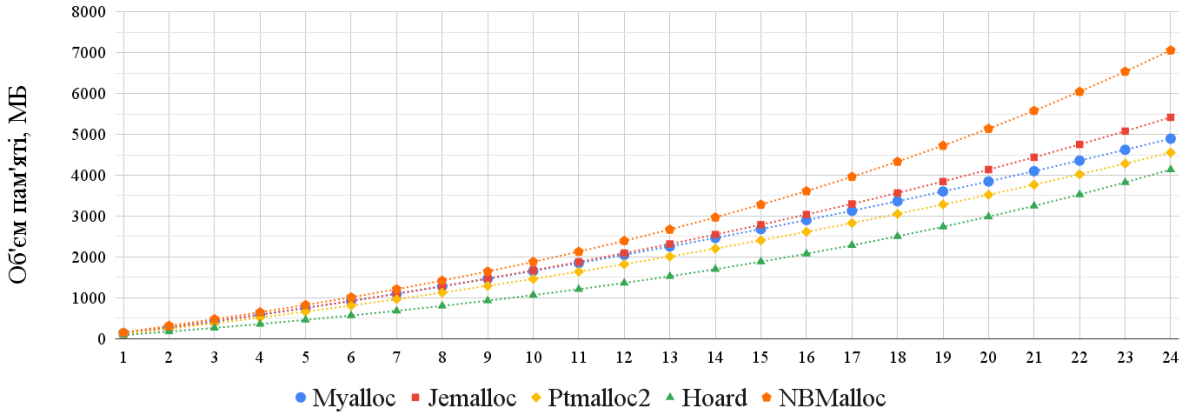


Рисунок 13 — Загальна кількість використаної пам'яті

Алокатор показує середній об'єм пам'яті та має лінійне зростання у порівнянні з сучасними алокаторами Hoard або Jemalloc. Це може бути пов'язано з використанням мапи сторінок, яка дозволяє зафіксувати об'єм пам'яті, який потрібен для коректної роботи алокатора.

6.3 Подвійне володіння

Рисунок 14 демонструє час виконання на запусках від 1 до 24 потоків. Представлені результати є середнім значенням з п'яти запусків.

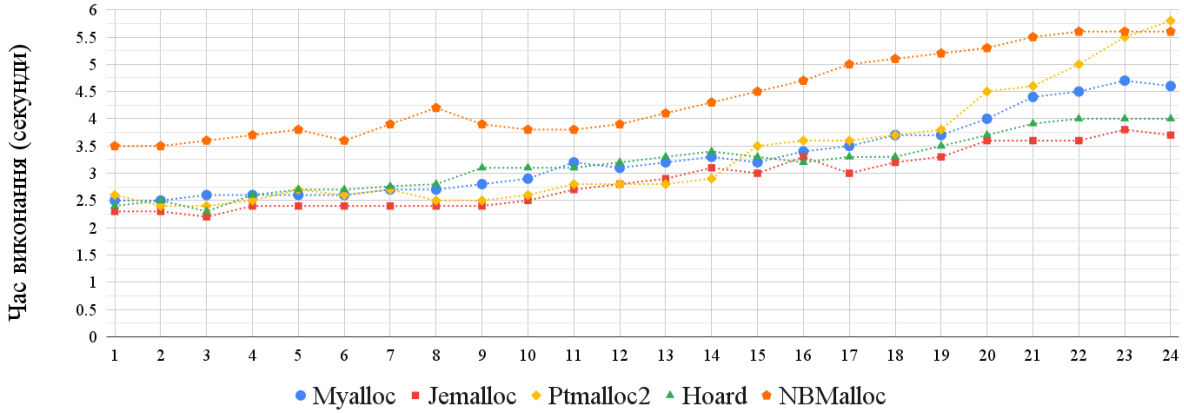


Рисунок 14 — Активне подвійне володіння

Алокатор показує середній час виконання, але загалом адресує проблему активного подвійного володіння, як і більшість інших алокаторів. Несподівано, академічний алокатор NBMalloc, що розроблений на базі Hoard, показує гірший час виконання ніж алокатори, які явно не адресують подвійне володіння, такий як PtMalloc2.

Рисунок 15 демонструє час виконання на запусках від 1 до 24 потоків. Представлені результати є середнім значенням з п'яти запусків.

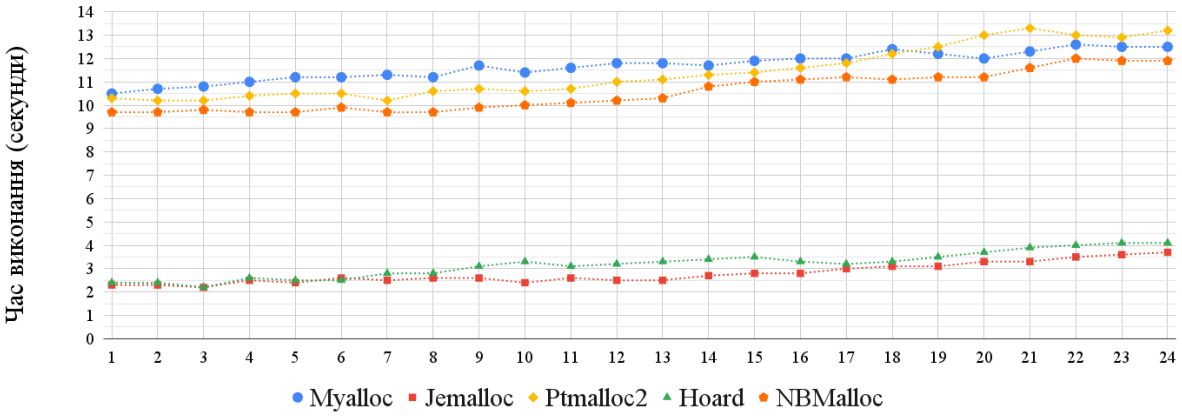


Рисунок 15 — Пасивне подвійне володіння

На графіку видно, що алокатор разом з NBMalloc та Ptmalloc2 має значно гірший час виконання у порівнянні з сучасними алокаторами Hoard та Jemalloc, тому погано адресує проблему пасивного подвійного володіння.

ВИСНОВКИ

Результатом роботи є багатопотоковий алокатор, який виконує поставлені вимоги — потокобезпечність без застосування блокувань, масштабованість, імунітет до проблем взаємного блокування та зміни стану потоку, захищеність від інверсії пріоритетів та раптового закінчення роботи.

Експериментальний аналіз показав, що алокатор демонструє хорошу ефективність за більшістю показників на рівні з іншими сучасними рішеннями.

Алокатор показав ефективність у вирішенні проблеми активного подвійного володіння. Аналіз використаної пам'яті показує, що алокатор добре адресує проблему внутрішньої фрагментації правильним підбором класів розміру та має фіксовані витрати пам'яті на внутрішню реалізацію завдяки реалізації мапи сторінок. Аналіз ефективності та масштабованості показав, що алокатор не поступається сучасним алокаторам.

Отже, алокатор створений у рамках цієї роботи може використовуватись у сучасних програмах для розподілу пам'яті.

Код алокатора та приклади його застосування можна знайти за посиланням: `https://github.com/sancheg31/Allocator`.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

[1] Robert HB Netzer and Barton P Miller. What are Race Conditions? - Some Issues and Formalizations ACM Letters on Programming Languages and Systems 74-88, 1992.

[2] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. Journal of Parallel and Distributed Computing, 67(12): 1270–1285, 2007.

[3] Електронний ресурс: `https://owasp.org/www-community/vulnerabilities/Doubly_freeing_memory`

[4] Електронний ресурс: `https://pure.security/introduction-to-use-after-free-vulnerabilities/`

[5] Mark S Johnstone and Paul R Wilson. The Memory Fragmentation Problem: Solved? In ACM SIGPLAN Notices, volume 34, pages 26–36. ACM, 1998.

[6] Jonathan Afek and Adi Sharabani. Dangling Pointer: Smashing the Pointer for Fun and Profit. 2007.

[7] John M Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. The Computer Journal, 20(3):242–244, 1977

[8] Leah Epstein and Rob van Stee. Improved results for a memory allocation problem. Theory of Computing Systems, 48(1):79–92, 2011.

[9] C++ Standard Committee. Information Technology – Programming Languages – C. Standard, International Organization for Standardization, 2011.

[10] John M Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. The Computer Journal, 20(3):242–244, 1977.

[11] M. Tufegdzic, A. Miskovic. Sequential fit algorithms evaluation using object-oriented programming. 12th International Conference Science and Higher Education in Function of Sustainable Development - SED 2021

[12] Електронний ресурс: `https://research.cs.vt.edu/AVresearch/MMtutorial/sequential.php`

[13] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In USENIX summer, volume 16. Boston, MA, USA, 1994

[14] Josep Torrellas, HS Lam, and John L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. IEEE Transactions on Computers, 43(6):651–663, 1994.

[15] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In ACM SIGARCH Computer Architecture News, volume 28, pages 117–128. ACM, 2000.

[16] Jason Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In BSDCan Conference, 2006.

[17] Wolfram Gloger. Ptmalloc, 2006.

[18] Sanjay Ghemawat, Paul Menage. TCMalloc: Thread-caching malloc, 2009.