Хмельницький національний університет
Факультет програмування
та комп'ютерних і телекомунікаційних систем
Кафедра комп'ютерної інженерії та системного програмування

# КВАЛІФІКАЦІЙНА РОБОТА

*бакалавр*
Освітній рівень

Засоби аналізу віртуальної пам'яті
Назва теми

*КвРКІ.170134.17.01.02 ПЗ*
Шифр

Галузь знань *12* «*Інформаційні технології*»
Шифр, назва

Спеціальність *123* «*Комп'ютерна інженерія*»
Шифр, назва

Освітня програма «*Комп'ютерна інженерія*»
Назва

Виконав: студент IV курсу, група КІ–17–1 _____ Н.О. Вовк
Підпис Ініціали, прізвище

Керівник: _____ Т.М. Кисіль
Підпис, дата Ініціали, прізвище

Нормоконтролер _____ С.М. Лисенко
Підпис, дата Ініціали, прізвище

До захисту допускаю:
Зав. кафедри комп'ютерної
Інженерії та системного
Програмування _____ Т.О. Говорущенко
Підпис Ініціали, прізвище

«____» _червня_ 2021 р.

Хмельницький 2021

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет ПРОГРАМУВАННЯ ТА КОМП'ЮТЕРНИХ І ТЕЛЕКОМУНІКАЦІЙНИХ СИСТЕМ

Кафедра КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА СИСТЕМНОГО ПРОГРАМУВАННЯ

Освітній рівень БАКАЛАВР

Галузь знань 12 ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ

Спеціальність 123 КОМП'ЮТЕРНА ІНЖЕНЕРІЯ

Освітня програма ОСВІТНЯ ПРОГРАМА «КОМП'ЮТЕРНА ІНЖЕНЕРІЯ»

ЗАТВЕРДЖУЮ

Зав. кафедри Т.О.Говорущенко

" 11 " 01 2021 р.

**ЗАВДАННЯ**
**НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА**

Вовку Назарію Олександровичу

Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Засоби аналізу віртуальної пам'яті

Керівник проекту (роботи) Кисіль Т.М., д.т.н., проф.

Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 05.02.2021 р. № 11

2. Строк подання студентом проекту (роботи) на кафедру 07.06.2021 р.

3. Вихідні дані до проекту (роботи) Завдання на дипломне проектування

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Дослідження предметної області та постановка задачі

Проектування модуля відображення стану пам'яті

Програмна реалізація модуля для відображення стану пам'яті при різних навантаженнях

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

Лістинг модуля відображення стану пам'яті

Алгоритм роботи програми

Лістинг модулів програми

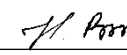
6. Консультанти розділів дипломного проекту (роботи)

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| Нормоконтроль | Лисенко С.М., професор кафедри КІСП |  |  |
| Антиплагіат | Нічепорук А.О., доцент кафедри КІСП |  |  |

7. Дата видачі завдання « 11 » 01 2021 р.

## КАЛЕНДАРНИЙ ПЛАН

| №з/п | Назва етапів (розділів) дипломного проекту (роботи) | Термін виконання етапів проекту (роботи) | Примітка |
|---|---|---|---|
| 1 | Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи з керівником | 11.01.2021 | виконано |
| 2 | Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об’єкта та предмета дослідження | 01.02.2021 | виконано |
| 3 | Робота над розділом 1 – дослідження предметної області та постановка задачі | 01.03.2021 | виконано |
| 4 | Робота над розділом 2 – моделювання та проєктування модуля відображення пам’яті | 01.04.2021 | виконано |
| 5 | Робота над розділом 3 – апаратна реалізація модуля для відображення стану пам’яті | 30.04.2021 | виконано |
| 6 | Оформлення пояснювальної записки згідно вимог | 31.05.2021 | виконано |
| 7 | Попередній захист ВКР | 02.06.2021 | виконано |
| 8 | Захист ВКР на засіданні ЕК | Червень 2021 року | |

**Студент**  Підпис Вовк Н.О. Ініціали, прізвище

**Керівник проекту (роботи)**  Підпис Кисіль Т.М. Ініціали, прізвище

| № рядка | формат | Позначення | Найменування | Кіл. листів | № екз | Примітка |
|---|---|---|---|---|---|---|
| | | | Текстові документи | | | |
| 1 | | КвРКІ 170134.17.01.02 ПЗ | Пояснювальна записка | 66 | | |
| | | | | | | |
| | | | Графічні матеріали | | | |
| 2 | | КвРКІ 170134.17.01.02 Е8 | Модуль відображення | 1 | | |
| | | | стану пам’яті | | | |
| 3 | | КвРКІ 170134.17.01.02 Е8 | Алгоритм роботи | 1 | | |
| | | | програми | | | |
| 4 | | КвРКІ 170134.17.01.02 Е8 | Лістинг модулів програми | 1 | | |

| Зм | Арк | № докум | Підпис | Дата | КвРКІ 170134.17.01.02 ВП | | | |
|---|---|---|---|---|---|---|---|---|
| Розробив | | Вовк | | 24.06.21 | Відомість проекту | Літера | Аркуш | Аркушів |
| Перевір. | | Кисіль | | 24.06.21 | | У | 1 | 1 |
| Н. контр. | | Лисенко | | 24.06.21 | | ХНУ, КІ-17-1 | | |
| Затв. | | Говорущенко | | 24.06.21 | | | | |

# АНОТАЦІЯ

Тема кваліфікаційної роботи: «*Засоби аналізу віртуальної пам'яті*».

Автор роботи: *Вовк Назарій Олександрович*.

Керівник роботи: *Кисіль Тетяна Миколаївна*.

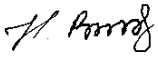
Пояснювальна записка: *66 с., 11 рис., 8 табл., 3 дод., 40 джерел*.

Графічна частина: *8 презентаційних слайдів*.

ВІРТУАЛЬНА ПАМ'ЯТЬ, МЕТОДИ АНАЛІЗУ ВІРТУАЛЬНОЇ ПАМ'ЯТІ, МОДУЛЬ ВІДОБРАЖЕННЯ СТАНУ ПАМ'ЯТІ ПРИ РІЗНИХ НАВАНТАЖЕННЯХ НА СИСТЕМУ.

Метою роботи є розробка модуля моніторингу пам'яті при різних навантаженнях.

У цій роботі розроблений додаток для аналізу пам'яті в середовищі програмування Visual Studio 2019 на мові C++ та з використанням Windows Forms. Розроблений модуль аналізу пам'яті створений на основі функцій та елементів мови програмування C++, дозволяє здійснювати аналіз фізичної та віртуальної пам'яті, також показує вільне місце на диску, та процент зайнятої пам'яті на операційній системі Windows.

Підпис студента  Дата 24.06.2021

**ЗМІСТ**

# СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ОЗУ - оперативний запам'ятовувальний пристрій

TLB – технологія, що оптимізує перетворення віртуальних адрес в фізичні

ПЗ - програмне забезпечення

БД - база даних

БПР - блок прийняття рішень

ГА - генетичний алгоритм

ОС - операційна система

ПЗ - програмне забезпечення

СВВ - система виявлення вторгнень

ЕС - експертна система

DDoS - Distributed Denial of Service (розподілена відмова в обслуговуванні)

IDS - система виявлення вторгнень

DMA – прямий доступ до пам’яті

RAM – оперативна пам’ять

# ВСТУП

Оскільки програми ставали все складніше, а програмуванням займалося все більше людей, виникла необхідність забезпечити автоматичне керування пам'яттю, щоб полегшити тягар програміста. У 1959 році дослідники з Манчестерського університету (Велика Британія) запропонували і створили перший робочий прототип системи віртуальної пам'яті в складі системи Atlas. Вони ввели ключову концепцію віртуальної пам'яті - відмінність між "адресою" і "місцем в пам'яті". Пізніше "адреса" стала більш широко відомою як віртуальна адреса, а "місце в пам'яті" може бути фізичною або реальною адресою в основній пам'яті чи місцем зберігання.

Це дозволило програмістам використовувати інформацію за її віртуальною адресою, в той час як системне програмне забезпечення разом з апаратним забезпеченням покликане динамічно перетворювати віртуальні адреси в їх розташування в основний пам'яті або в сховищі. ОС забезпечувала автоматичне переміщення даних між пам'яттю і сховищем по мірі необхідності.

Ключові концепції і механізм дії віртуальної пам’яті з часом значно покращились в проекті «Multics». Вони надали можливість створити двовимірний віртуальний адресний простір, який забезпечував простоту спільного використання, модульність і захист, а також ефективне управління фізичною пам’яттю шляхом виділення пам’яті фіксованих розмірів (сторінки). Цей адресний простір вимагав два ідентифікатора для виявлення ділянки пам’яті.

Потім була розроблена модель локального доступу до програм Деннінгом, для заповнення критичних компонентів віртуальної пам’яті, таких як виділення обсягу пам’яті для процесів, та розподіл інформації між основною і допоміжною пам’яттю.

Віртуальна пам’ять використовує апаратне та програмне забезпечення для компенсації дефіциту фізичної пам’яті, переносячи дані з оперативної пам’яті на диск. Фактично віртуальна пам’ять дозволяє комп’ютеру обробляти вторинну пам’ять як основну. З її допомогою операційній системі досить зберігати в основній пам’яті тільки використовувані процеси, а решту програм

розташовувати на диску. Вона була розроблена в той час, коли фізичну пам’ять не всі могли собі дозволити. Щороку на ринок випускаються нові покоління оперативної пам'яті, але досі вона має обмежений ресурс через те, що програми постійно збільшуться в розмірах. Оскільки жорсткий диск набагато дешевший за оперативну пам'ять, він є альтернативою збільшення обсягу пам'яті, доступної для програм.

Якщо процес буде використовувати більше пам’яті ніж є в наявності, система вийде з ладу, вирішенням цієї проблеми займається віртуальна пам’ять. Щоразу, коли програма звертається в пам’ять, система віртуальної пам’яті перевіряє, чи знаходиться адреса в пам’яті, чи чекає обміну. Ця перевірка відбувається за допомогою апаратного контролера пам’яті, що збільшує час доступу до пам’яті відносно системи, яка використовує лише фізичну адресу. Якщо адреса знаходиться не в оперативній пам’яті, системі потрібно виконати обмін, що значно збільшує час операції. Доступ до фізичної адреси займає декілька наносекунд, в той час як доступ до диска кілька мілісекунд, що в 1000000 разів довше. Тому варто обмежувати використання віртуальної пам’яті на комп’ютері. Якщо постійно працювати з великим обсягом пам’яті, система буде потребувати більше оперативки.

Механізм віртуальної пам’яті надає операційній системі у використання певну частину вторинної пам’яті. З її допомогою операційна система може завантажувати більші програми та запускати декілька програм одночасно, використовуючи секцію жорсткого диска, для наслідування оперативної пам’яті, що дозволяє процесам працювати ніби з нескінченною пам’яттю.

# 1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

## 1.1 Аналіз задачі, обґрунтування вибору моделі життєвого циклу для реалізації проекту

З перших днів появи електронних обчислювальних машин розробники розуміли, що швидкий доступ до великого об'єму пам'яті ускладнений, тому пам'ять комп'ютера має бути організована ієрархічно. Комп'ютерна пам'ять зазвичай організована як мінімум на двох рівнях - "основна пам'ять" і "допоміжна пам'ять" або сховище. Звертатись до інформації програми можна було тільки тоді, коли вона знаходилася в основній пам'яті. Очевидною проблемою являється постійна перевірка, яка інформація повинна знаходитися в основній пам'яті, а яка в допоміжній. Ця проблема широко відома як проблема розподілу пам'яті.

До кінця 1950-х років будь-яка програма, яка потребувала отримання доступу до великої кількості інформації, ніж могла б поміститися в основну пам’ять, повинна була містити логіку для вирішення проблеми розподілу пам'яті. Крім того, для забезпечення можливості мультипрограмування і багатозадачності ранні системи розділяли фізичну пам'ять спеціальним набором регістрів. Проблеми автоматичного розподілу пам'яті і мультипрограмування не лише ускладнювали завдання написання великих програм, а й ускладнювали ефективне використання основної пам’яті ключового обчислювального ресурсу.

Хоча віртуальна пам'ять спочатку була задумана для того, щоб полегшити програмісту роботу по написанню власних процедур розподілу пам'яті, згодом вона стала використовуватися для різних цілей. Деякі приклади з них перераховані нижче.

Автоматичний розподіл пам'яті: віртуальна пам'ять дозволяє запускати програму, потреби якої в пам'яті перевищують встановлений обсяг основної пам'яті. Це звільняє програміста від необхідності знати обсяг пам'яті системи, в якій буде виконуватися програма. Сучасні ОС досягають цього шляхом переміщення даних між основною пам'яттю (наприклад, DRAM) і допоміжної

пам'яттю (наприклад, диском), щоб створити ілюзію більшого обсягу пам'яті, ніж той, який є насправді. доступною. Цей механізм прийнято називати свопінгом.

Захист: віртуальна пам'ять забезпечує ефективний захист пам'яті. Апаратне забезпечення віртуальної пам'яті може примусово обмежувати доступ до частин пам'яті процесів. Це корисно для запобігання небажаного перезапису інформації, доступної тільки для читання (наприклад, коду). Також дозволяє використовувати оптимізації, такі як копіювання при записі, при якому ділянка пам'яті може бути загальною для декількох процесів до тих пір, поки хоча б один з них не спробує записати ділянку пам'яті. Сучасні системи, також включають захист, який забороняє виконання певних розподілів пам'яті, щоб підвищити ефективність захисту пам'яті від шкідливих фрагментів коду.

Підкачка: віртуальна пам’ять дає змогу переносити інформацію (дані, код…) з допоміжної пам’яті в основну тільки при першому зверненні до них. Підкачка забезпечує швидкий запуск процесів, так як не потребує попереднього завантаження всієї інформації, яка може знадобитись процесу під час її виконання.

Переміщення: полегшує розробку програмного забезпечення, дозволяючи одному і тому ж віртуальному адресу в різних програмах посилатися на різні фізичні частини пам'яті під час виконання.

Збір метаданих: більшість сучасних апаратних засобів віртуальної пам'яті дозволяють ефективно зберігати невелику кількість метаданих з одиницями розподілу віртуальної пам'яті (наприклад біти посилань). Це дозволяє збирати інформацію про використання. Наприклад, ці керовані апаратним забезпеченням метадані часто корисні для багатьох процедур ОС, таких як відновлення сторінкового кадру. або навіть в збірці сміття в багатьох мовах високого рівня.

Спільне використання: віртуальна пам'ять дозволяє ефективно розділяти код і дані між кількома процесами шляхом відображення віртуальних адрес, що генеруються різними процесами, на одну і ту ж комірку фізичної пам'яті. Це особливо корисно при спільному використанні бібліотек кілька процесів.

## 1.2 Механізм переводу віртуальних адресів

Операційна система і апаратне забезпечення координують процес переводу адресів. Існує всього дві техніки перетворення адрес, - сегментація і підкачка. Віртуальна пам’ять з підкачкою, також відомою як сторінкова, переводить лінійну віртуальну адресу в фізичну. Процес перетворення адресів схований від користувацьких програм, за вийнятком його наслідків для продуктивності системи.

Сторінкова віртуальна пам'ять транслює віртуальні адреси в однієї або декількох гранулярностей, визначених ISA (наприклад, 4 КБ). З іншого боку, сегментація дозволяє співставлення між адресними просторами та змінними (часто довільних) гранулярності.

Сегментація часто розкриває користувачам два одновимірних віртуальних адресних простори, де адреса ідентифікується за допомогою ідентифікатора сегменту і зміщенням усередині сегменту. Одною з переваг підкачки перед сегментацією являється те, що пейджинг може допомогти пом'якшити зовнішню фрагментацію.

Зовнішня фрагментація може виникнути, якщо виділення пам’яті відбувається в різних розмірах, залишаючи фізичну пам’ять розкиданою в невеликих комірках пам’яті, непридатних для задоволення подальших запитів пам’яті.

Пейджинг пом'якшує цю проблему, виділяючи і видаляючи пам'ять в одному або декількох фіксованих розмірах, що називаються розміром сторінки. Сегментація, з іншого боку, дозволяє легко розділяти семантично пов'язані ділянки пам'яті (сегменти) між декількома процесами.

Методи розподілу пам’яті подано на рисунку 1.1.

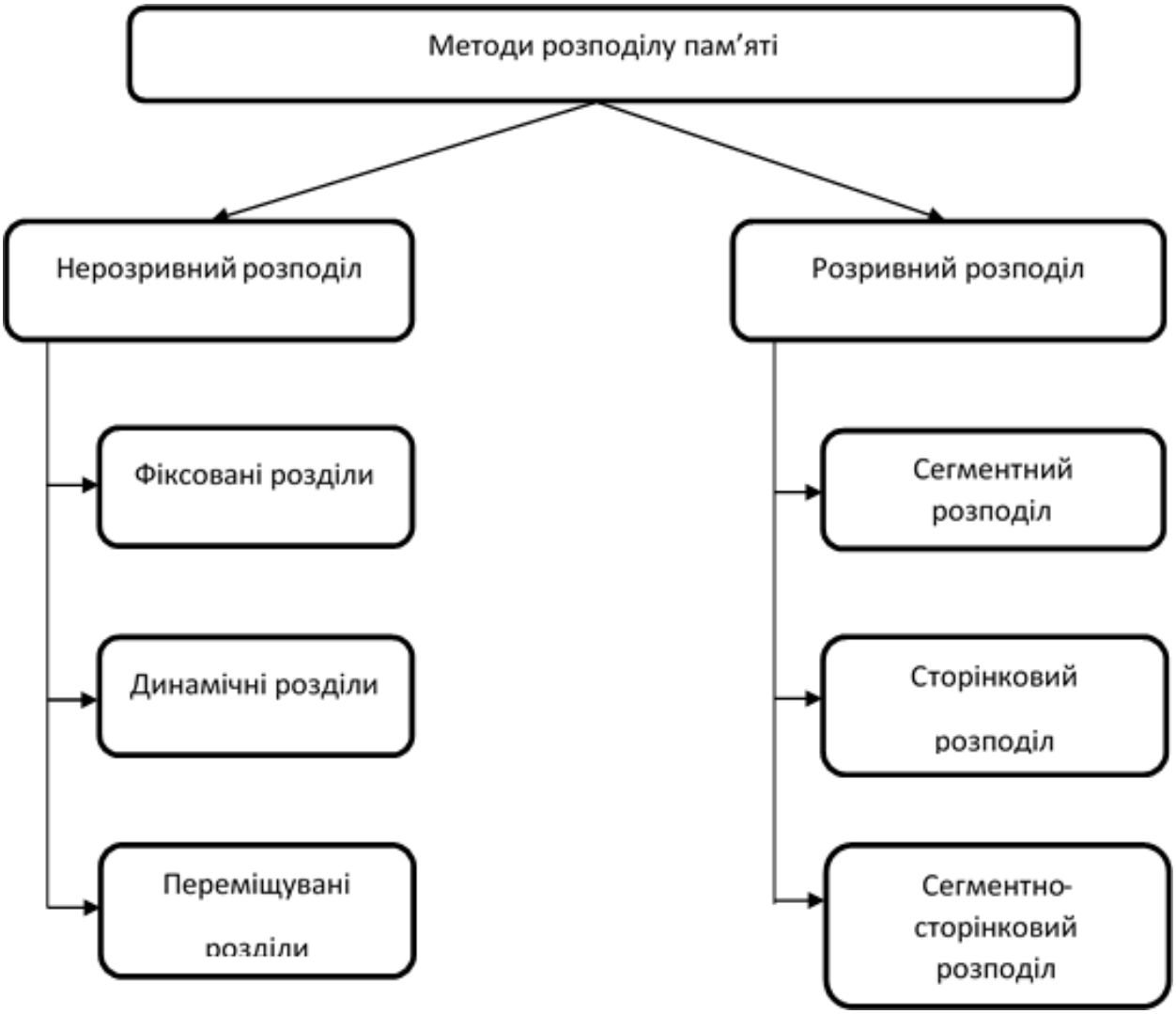



Рисунок 1.1 - Методи розподілу пам'яті

Сегментація і підкачка відрізняються як моделі пам'яті в плані того, як ділиться пам'ять, проте їх також можна комбінувати. Деякі системи віртуальної пам'яті поєднують сегментацію і пейджинг. У цьому випадку пам'ять ділиться на фрейми або сторінки. Сегменти займають кілька сторінок, а віртуальний адреса включає в себе як номер сегмента, так і номер сторінки.

### 1.2.1 Сегментний розподіл пам’яті

Першим з розривних методів розподілу був сегментний розподіл пам’яті. Цей метод являє собою розбиття програми на частини, після кожній частині виділяється фізична пам’ять. Одним з способів є розбиття програми на її логічні

елементи, які називаються сегментами. По своїй суті кожен програмний модуль може сприйматись як окремий сегмент, тоді вся програма буде представленням великої кількості сегментів. Сегменти розміщуються в пам’яті як окремі одиниці. Логічне звернення до цих елементів в цьому випадку буде складатись з імені сегмента і зміщення відносно його початку. Фізично ім’я чи порядковий номер сегмента буде належати якісь адресі, з якої цей сегмент був створений при розміщенні в пам’яті, і зміщення повинне добавлятись до цієї базової адреси.

Перетворенням імені сегмента в його порядковий номер займається система програмування. Для кожного сегменту встановлюється його об’єм. Цей об’єм повинен бути відомим операційній системі, щоб вона мала змогу виділяти необхідний йому об’єм пам’яті. ОС буде розміщувати сегменти в пам’яті і для кожного з них вона має вести облік їх місцезнаходження. Вся інформація про розміщення сегментів задачі зазвичай розміщується в таблиці сегментів, її також називають таблицею дескрипторів сегментів задачі. Усі задачі мають свою таблицю сегментів. Приклад роботи сегментного розподілу наведено на рисунку 1.1.

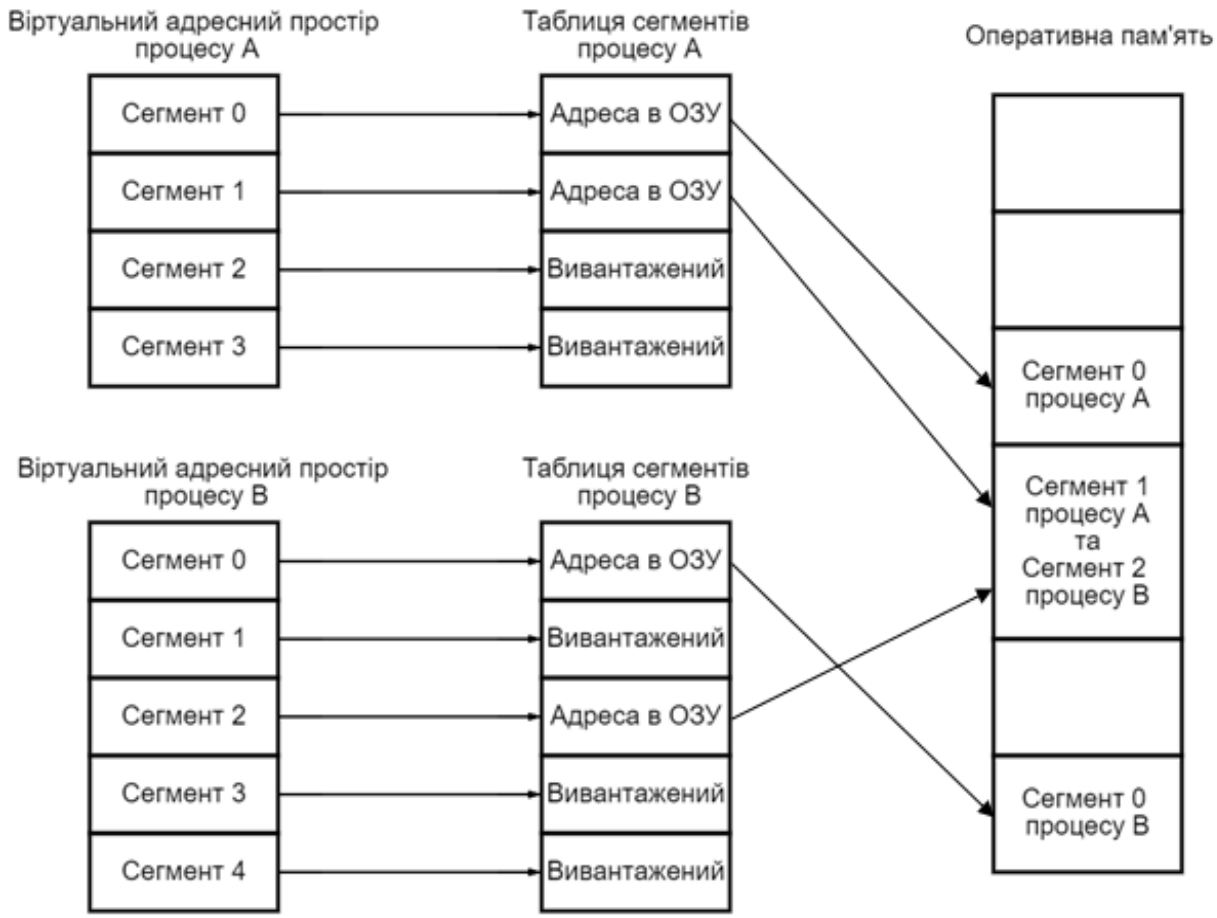



Рисунок 1.2 - Сегментний розподіл пам'яті

Дескриптор таблиці сегментів включає:

1) ознаку присутності;
2) фізичний адрес сегмента в ОЗУ і розмір сегмента;
3) права доступу до сегменту;
4) ознака модифікації сегмента;
5) ознаку звернення до сегмента.

Віртуальний адрес для цього способу буде складатись з двох полів - номеру сегменту і зміщення відносно початку.

Таким чином сегменти, що розміщуються в пам’яті мають відповідну інформаційну структуру, відому як дескриптор сегмента. Для всіх виконуваних процесів ОС створює таблицю дескрипторів сегментів, при розміщенні сегментів в ОЗУ або віртуальній пам’яті помічає в дескрипторі місцезнаходження сегмента.

Якщо сегмент виконуваної задачі в даний момент часу знаходиться в оперативній пам’яті, в дескрипторі ставиться відповідна позначка. Для цього використовується біт присутності P (від слова «present»). В цьому випадку поле адреси диспетчера пам’яті записує адресу фізичної пам’яті, звідки сегмент починається, у поле довжини сегмента («limit») вказується кількість адресних комірок. Це поле використовується не тільки для розміщення сегментів, а й для того, щоб контролювати чи звертається код виконуваної задачі за межі сегмента.

Якщо довжина коду перевищує довжину сегмента внаслідок помилок програмування, це говорить про порушення адресації. За допомогою введення спеціальних апаратних засобів генерується сигнал переривання, який дозволяє знаходити види таких помилок. Коли біт присутності в дескрипторі вказує, що сегмент знаходиться не в ОЗУ а у віртуальній пам’яті, тоді поля адрес і довжини, використовують для зазначення адреси сегменту в координатах зовнішньої пам'яті.

Окрім інформації про місцезнаходження, в дескрипторі містяться дані про його тип (сегмент коду або сегмент даний), права доступу (чи є доступ до його модифікації), мітка про звернення до цього сегменту (як часто цей сегмент використовувався, на цій підставі можна прийняти рішення про те чи надати місце, яке займає поточний сегмент, іншому сегменту).

При передачі управління наступній задачі система заносить у відповідний регістр адресу таблиці дескрипторів сегментної задачі. Сама таблиця дескрипторів в свою чергу являє собою сегмент даних, який обробляється диспетчером пам’яті ОС.

Завдяки такому методу появляється можливість розміщувати в оперативній пам’яті не всі сегменти виконуваної задачі, а тільки ті, які потрібні в даний момент. З однієї сторони, загальний об’єм віртуального простору задачі може бути більшим за об’єм наявної фізичної пам’яті. З іншої сторони в пам’яті можна розміщувати більше задач, оскільки, в більшості випадків, всі сегменти одночасно задачі не потребуються. Коефіцієнт мультипрограмування, як відомо дозволяє збільшити загрузку системи і більш ефективно використовувати ресурси комп’ютера. Очевидно, що збільшувати число задач можна до певної межі, якщо в

пам’яті не вистачатиме місця для сегментів, які часто використовуються, то продуктивність системи різко упаде. З цього випливає, що сегмент, повинен знаходитись в оперативній пам’яті, для участі в обчисленнях. Якщо в пам’яті є вільне місце, необхідно знайти потрібний сегмент у віртуальній пам’яті і загрузити його в ОЗУ. У випадку, коли вільного місця немає, потрібно прийняти рішення, замість якого з наявних сегментів буде завантажуватись необхідний. Переміщення сегментів між оперативною і віртуальною пам’яттю називають свопінгом сегментів.

Якщо потрібний сегмент не наявний в оперативній пам’яті, виникає переривання, і управління передається через диспетчер пам’яті - програмі завантаження сегмента. В той час, коли відбувається пошук сегменту на зовнішньому накопичувачі і завантаження його в ОЗУ, диспетчер визначає підходяще місце для сегмента. Кожного разу при зчитуванні сегментів в оперативну пам’ять, таблиця дескрипторів встановлює адресу початку сегменту і ознаку його присутності.

Для вирішення проблеми заміщення, використовуються такі засоби:

1) FIFO («first in first out») – першим зайшов, першим вийшов;
2) LRU («least recently used») – найменш використовуваний;
3) LFU («least frequently used») – використовується рідше за інших;
4) випадковий вибір сегмента.

Перше і останнє правило є самим простим в реалізації, однак вони не враховують, як часто використовується сегмент, з цього випливає, що диспетчер пам’яті може вивантажити чи розформувати сегмент, до якого буде звернення. Але ймовірність помилки цих двох правил набагато вища, ніж у другого і третього правила, у них враховується інформація про використання сегментів.

В алгоритмі FIFO з кожним сегментом зв’язується черговість його розміщення в пам’яті. Операція заміщення вибирає елемент, що першим попадає в пам’ять. Кожен послідуючий сегмент добавляється в кінець черги. Алгоритм враховує час знаходження сегмента в пам’яті, але не враховує фактичне використання сегментів. Для прикладу, перші завантажені сегменти можуть мати

змінні, які потрібні протягом всієї роботи програми. Це призводить до повернення щойно заміщеного сегменту.

Алгоритм LFU вибирає для заміщення ту сторінку, до якої не було звернення на протязі тривалого періоду часу. Алгоритм LRU асоціює з кожною сторінкою час її останнього використання. Для заміщення вибирається та сторінка, яка найдовше не використовувалася.

Алгоритм LRU співставляє з кожною сторінкою час її кінцевого використання. Щоб виконати заміщення, вибирається сторінка, яка найдовше не використовувалася.

Щоб реалізувати алгоритми LRU і LFU процесору необхідно мати додаткові апаратні засоби. Дескриптору достатньо міняти значення відповідного біта при зверненні, для отримання фізичної адреси, де сегмент розташовується в пам’яті. Диспетчер пам’яті час від часу передивляється таблиці дескрипторів виконуваних завдань та збирає статистичну інформацію про звернення до сегментів. В результаті можна скласти список, впорядкований по тривалості простою для LRU, чи по частоті використання для LFU.

Серйозною проблемою є захист пам’яті, вона виникає при організації мультипрограмного режиму. Щоб запобігти псуванню операційної системи та інших обчислювальних засобів необхідно, щоб доступ до таблиць сегментів з метою їх модифікації був забезпечений тільки для коду самої ОС. Для цього код операційної системи повинен виконуватися з відповідними правами доступу, з яких є можливість здійснювати маніпуляції з дескрипторами сегментів, вихід за межі сегмента в звичайній програмі повинен викликати переривання по захисту пам'яті. Кожна прикладна задача повинна мати можливість звертатися тільки до власних і до загальних сегментам. Всі задачі повинні мати доступ до своїх та загальних сегментів.

Переваги сегментного способу організації віртуальної пам’яті:

По-перше, при запуску програми є можливість використання не всієї пам’яті, а її частин, по мірі необхідності. Оскільки в переважній більшості випадків алгоритм, за яким працює код програми, є розгалуженим, а не лінійним, звідси випливає, що деякі частини програми, розташовані в окремих сегментах,

можуть бути не задіяні, це означає, що завантаження їх в оперативну пам’ять має не обов’язковий характер.

По-друге, частина програмних модулів роздільна. Оскільки ці модулі являють собою сегменти, значно легше організувати доступ до їх спільних сегментів. Сегмент з роздільним кодом розміщується в пам’яті в єдиному екземплярі, в деяких таблицях дескриптора сегментів буде знаходитись мітка на їх роздільні сегменти.

Однак у цього способу розподілу пам’яті є свої недоліки. Доступ до комірок пам’яті витрачає багато часу. Для початку потрібно знайти і прочитати дескриптор сегмента, після цього, вичислити кінцеву фізичну адресу, використовуючи отримані дані про його місцезнаходження. Щоб зменшити витрачений час, використовуються кеш-функції – це дескриптори, які розміщенні в частинах оперативної пам’яті, у них знаходяться вміст і адреси часто використовуваних даних.

Не дивлячись на те, що сегментний спосіб розподілу пам’яті зменшує фрагментацію пам’яті, ніж способи з нерозривним розподілом, фрагментація все одно залишається. Також велика кількість пам’яті витрачається на розміщення і обробку дескрипторних таблиць. Адже кожній задачі повинна відповідати власна таблиця дескрипторів сегментів. До того ж, при визначенні фізичних адрес доводиться виконувати математичні операції, що займає додаткові витрати часу.

### 1.2.2 Сторінковий розподіл пам’яті

При сторінковому методі розподілу пам’яті всі фрагменти програми, при розбитті виходять однаковими. Також однаковими є одиниці пам’яті, які доступні для розміщення частин програми. Ці частини також відомі, як сторінки, оперативна пам’ять розбивається на фізичні сторінки, програма в свою чергу на віртуальні сторінки. Частини віртуальних сторінок розміщуються в оперативній пам'яті, інша частина в зовнішній. Місце в зовнішній пам’яті, в якості якої виступають магнітні накопичувачі, відоме як файл підкачки, чи сторінковий файл. Приклад роботи показано на рисунку 1.3.

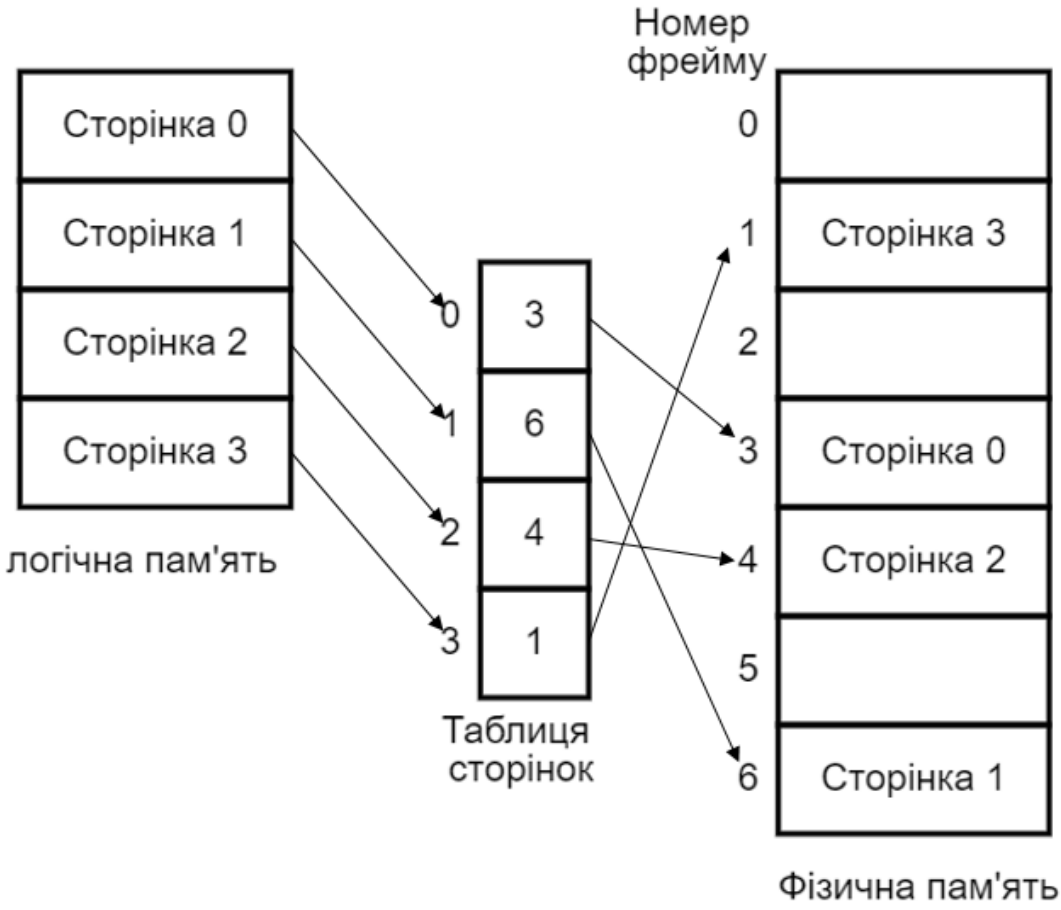



Рисунок 1.3 - Сторінковий метод організації пам'яті

Завдяки розбиттю оперативної пам’яті на однакові сторінки, які кратні двійці, замість одновимірного адресного простору ми отримуємо двовимірний. Перша координата адресного простору - це номер сторінки, друга координата - номер комірки всередині вибраної сторінки (також відома як індекс). Кількість бітів, які виділяються для індексу визначаються розміром сторінки, в той же час, обсяг бітів, які відводяться під номер віртуальної сторінки відповідають розміру необхідної віртуальної пам’яті.

Відображення системи під час виконання зводиться до відображення Pv (номер віртуальної сторінки) у величину Pp (номер фізичної сторінки). Немає потреби обмежувати віртуальні сторінки числом фізичних, сторінки, що не поміщаються можна розміщувати у зовнішній пам'яті, яка в даному випадку служить розширенням ОЗУ.

Для відображення віртуального адресу в фізичній пам'яті, задачі повинні мати таблицю сторінок, для трансляції адресного простору. Для опису сторінки диспетчер пам'яті ОС звертається до потрібного дескриптору, який на відміну від сегментного немає довжини, тому що всі сторінки мають однаковий розмір.

Ці дескриптори знаходяться за номером віртуальної сторінки в таблиці дескрипторів. За умови одиничного значення біта присутності, дана сторінка розміщується в оперативній пам'яті, і в дескрипторі знаходиться номер фізичної сторінки, який виділений для віртуальної адреси.

У випадку, коли біт присутності дорівнює нулю, в дескрипторі буде наявна адреса віртуальної сторінки, що розміщується в оперативній пам'яті. Таким чином відбувається трансляція віртуального адресного простору в фізичну пам'ять.

Захист сторінкової інформації, як у випадку сегментного розподілу, грунтується на контролі прав доступу до сторінок.

Усі сторінки без винятку оснащуються відповідним рівнем доступу. При перетворенні логічної адреси в фізичну, зрівнюється значення коду певного рівня доступу. Якщо рівні доступу не співпадають їх робота завершується. Рівні доступу подані на рисунку 1.4.


Рисунок 1.4 - Рівні доступу

Якщо відбувається звернення до віртуальної сторінки, яка в даний момент не знаходиться в оперативній пам’яті, виникає переривання, і управління передається диспетчеру пам’яті, що займається пошуком вільного місця. Зазвичай появляється перша вільна сторінка. Коли фізичної адреси немає, диспетчер пам’яті за допомогою процесу заміщення визначить сторінку, що підлягає

розформуванню або збереженню у зовнішній пам'яті. На її місці він розмістить нову віртуальну сторінку, якої не виявилося в оперативній пам'яті при зверненні.

При невеликому об’ємі фізичної пам’яті, коли не вдається розмістити сторінки в оперативній пам’яті, виникає «пробуксовка». Це ситуація, при якій комп’ютер замість вирішення поставленої задачі, займається циклічним переміщенням пам’яті з ОЗУ на диск, і назад. Щоб не допустити виконання цього процесу, бажано збільшити обсяг оперативної пам’яті, зменшити кількість запущених завдань, або використати більш ефективні методи заміщення.

Для більшості операційних систем метод заміщення сторінок LRU є найефективнішим. Він використовується в операційній системі Linux. Натомість у Windows розробники відмовились від цього методу і використовують правило FIFO, пояснюючи це бажанням, не бути залежними від апаратних можливостей процесора. З метою компенсування неефективності правила, була введена «буферизація» сторінок, які записуються в файл підкачки на диск, або розформовуються. Принцип буферизації – сторінка, перед появленням в зовнішній пам’яті помічається як кандидат на вивантаження. При наступному зверненні до сторінки, яка знаходиться в буфері, вивантаження не відбувається, і вона стає в кінець черги. У противному випадку сторінка вивантажується, а на її місце в буфер стає наступний кандидат. Довжина буфера не може бути більшою, звідси випливає, що ефективність сторінкової реалізації в Windows набагато менша, ніж у інших ОС. Пробуксовка починається навіть тоді, коли оперативна пам'ять має використовує значно більший обсяг оперативки.

Операційні системи, які мають пакетні послуги, для боротьби з пробуксовкою використовують метод «робочої множини». Робоча множина представляє собою перелік сторінок, до яких частіше звертались. Для коректної роботи процесу потрібно, щоб робоча множина постійно знаходилася в оперативній пам'яті та не підлягала вивантаженню.

Суттєвою перевагою сторінкового методу розподілу є мінімальна фрагментація. Використання пам’яті є достатньо ефективним, оскільки на кожну задачу може припадати одна незаповнена сторінка.

Сторінковий метод міг бути одним з найефективнішим, якщо б не його два суттєвих недоліка:

Перше, сторінкова трансляція віртуальної пам’яті вимагає істотних ресурсних витрат. Таблиці сторінок також розміщуються в пам’яті, крім того, їх потрібно постійно обробляти, чим займається диспетчер пам’яті.

Другий недоліком є розбиття програм на сторінки випадковим чином, без врахування логічних взаємозв’язків, які складають код. Це призводить до того, що міжсторінкові переходи, зазвичай, виконуються частіше, аніж міжсегментні, стає важко організувати розподіл програмних модулів між процесами.

### 1.2.3 Сторінково-сегментний метод розподілу пам’яті

Сторінково-сегментний розподіл являється комбінацією двох методів, які направлені на реалізацію переваг обох підходів. Віртуальний простір виконуваної задачі ділиться сегментами, що дає змогу встановлювати різні права доступу до частин коду та даних виконуваного процесу. Перенесення даних між диском та пам’яттю відбувається за допомогою сторінок. Для виконання функції перенесення сегменти віртуальної і фізичної пам’яті діляться на сторінки однакового розміру. З цього виходить, що у віртуальній адресі міститься показник на номер відповідного сегмента і зміщення відносно його початку, яке в свою чергу може складається з двох полів – сторінки та індексу. З вищесказаного можна зробити висновок, що віртуальна адреса сторінково-сегментного методу складається з трьох компонентів: сторінки, сегменту та індексу (рисунок 1.5).

Нумерація фізичних сторінок відбувається в межах оперативної пам’яті. Коли відбувається процес створення, в пам’ять завантажується частина сторінок, інші при необхідності додаються. З ходом часу система звільнює пам’ять для нових сторінок, вивантажуючи непотрібні. ОС веде таблицю сторінок для кожного процесу, у ній вказується відповідність між віртуальними на фізичними сторінками. Частиною її контексту є таблиці початкових адресів сегментів та сторінок процесу. При роботі процесу, ці адреси завантажуються в регістри процесора, для використання механізму перетворення адрес.

Перетворення віртуальної адреси в фізичну складається з:

Механізм сегментації – початкова віртуальна адреса перетворюється в лінійну. Для виконання цього завдання вираховується адреса дескриптора сегмента, на підставі номера сегмента та базової адреси таблиці сегментів. Відбувається аналіз полів дескриптора, після перевіряється можливість виконання створеної операції. Якщо є дозвіл на виконання, тоді відбувається додавання початкової адреси дескриптора і зміщення, яке задається початковою віртуальною адресою.

Сторінковий механізм – отримана віртуальна адреса перетворюється в фізичну.

Також відомий механізм, при якому віртуальний простір ділиться на сегменти, а сегмент в свою чергу на сторінки. Сторінки нумеруються в межах сегменту, і віртуальна адреса складається з трьох елементів: номер сегмента, номер сторінки та зміщення сторінки. ОС завантажує виконувану задачу посторінково, одна частина поміщається в ОЗУ, інша на диск. Коли потрібний процес активізується, у таблицю сегментів завантажується спеціальний регістр процесора. Ця таблиця складається з дескрипторів сегментів, вони містять опис розташування таблиць.


Рисунок 1.5 – Сегментно-сторінкова організація пам'яті

Переваги сегментно-сторінкового методу розподілу. Спосіб розбиття сегментів (рисунок 1.5) розміщує їх в пам’яті цілими частинами. Сегменти розбиваються на сторінки, всі сторінки сегмента завантажуються в пам’ять. Що дає можливість зменшити кількість звернень до відсутніх сторінок, тому що вихід за межі сегмента має меншу ймовірність ніж вихід за межі сторінки. Сторінки сегмента, які знаходяться в пам’яті, можуть бути в різних частинах пам’яті, через маніпуляцію диспетчеру пам’яті. Наявність сегментів полегшує розподіл програмних модулів паралельних процесів. Також зменшується фрагментація, завдяки виділенню пам’яті сторінок.

Недоліком цього методу являється використання великої кількості апаратних можливостей. Його застосунок можливий лише в дорогих обчислювальних системах.

## 1.3 Технологія перетворення фізичних адрес у віртуальні

Віртуальна пам'ять сильно вплинула б на продуктивність, якби вона вимагала читання сторінокових таблиць при кожному завантаженні або збереженні, подвоюючи затримку. На щастя, доступ до таблиці сторінок має велику часову локальність. Завдяки тимчасовій і просторовій локальності доступу до даних і великого розміру сторінки, більшість послідовних завантажень і збережень, швидше за все, посилатимуться на одну і ту ж сторінку. Тому, якщо процесор пам'ятає останній прочитаний запис таблиці сторінок, він, ймовірно, зможе повторно використати цей переклад без повторного читання таблиці сторінок.

Загалом, процесор може зберігати останні записи таблиці на декількох сторінках у невеликому кеші, який називається буфером перетворення (TLB). Процесор "дивиться убік", щоб знайти переклад в TLB, перш ніж звернутися до таблиці сторінок у фізичній пам'яті. У реальних програмах переважна більшість звернень потрапляє в буфером перетворення, уникаючи трудомісткого читання таблиці сторінок з фізичної пам'яті.

TLB організований як повністю асоціативний кеш і зазвичай містить від 16 до 512 записів. Кожен запис TLB містить номер віртуальної сторінки і відповідний їй номер фізичної сторінки. Доступ до TLB здійснюється за номером віртуальної сторінки. Якщо TLB звертається до віртуальної сторінки, вона повертає відповідний номер фізичної сторінки. В іншому випадку процесор повинен прочитати таблицю сторінок в фізичній пам'яті. TLB спроектований настільки компактно, що доступ до нього можна отримати менш ніж за один цикл. Незважаючи на це, TLB зазвичай має коефіцієнт попадання більше 99%. TLB зменшує кількість звернень до пам'яті, необхідних для більшості інструкцій завантаження або зберігання, з двох до одного.

Великі сторінки є одним з найбільш широко застосовуваних механізмів для зменшення промахів TLB в сучасних процесорах. Великий розмір сторінки відображає більшу кількість пов'язаних і розташованих на одній лінії віртуальних адрес, в пов'язані і впорядковані фізичні адреси. Таким чином, певна кількість

записів TLB може потенційно відображати набагато більший обсяг пам'яті в порівнянні з розміром базової сторінки (наприклад, в 512 разів більше для сторінки розміром 2 МБ в порівнянні з 2 КБ). в 512 разів більше для сторінки розміром 2 МБ в порівнянні з 4 КБ). Це потенційно може допомогти зменшити кількість промахів TLB. Таким чином, задану кількість записів TLB може потенційно відображати набагато більший обсяг пам'яті в порівнянні з розміром базової сторінки (наприклад, в 512 разів більше для сторінки розміром 2 МБ в порівнянні з 2 КБ). в 512 разів більше для сторінки розміром 2 МБ в порівнянні з 4 КБ). Це потенційно може допомогти зменшити кількість промахів TLB. Сьогодні майже всі процесори підтримують безліч великих розмірів сторінок. Наприклад, процесори x86-64 підтримують великі розміри сторінок 2 МБ і 1 ГБ понад базового розміру сторінки 4 КБ. ARM підтримує розміри сторінок 4 КБ і 64 КБ. PowerPC підтримує розміри сторінок 4 КБ, 64 КБ, 16 МБ, 16 ГБ. UltraSPARC підтримує сторінки розміром 8 КБ, 64 КБ, 4 МБ і 256 МБ.

Однак використання великих сторінок створює свій власний набір труднощів - як програмних, так і апаратних. Використання великих сторінок впливає на конструкцію TLB. Більший розмір сторінки вносить додаткову невідому в процес трансляції адрес - розмір сторінки трансляції. Відсутність знань про розмір сторінки ускладнює визначення номера віртуальної сторінки для пошуку в TLB по заданій віртуальній адресі. Цей в свою чергу, створює компроміс між складністю конструкції, продуктивністю і потужністю при проектуванні TLB.

Комерційні процесори використовують два підходи для підтримки декількох розмірів сторінок в TLB. Один з підходів, що дозволяє використовувати кілька розмірів сторінок в TLB, полягає в використанні повністю асоціативної TLB, як це часто робиться в багатьох процесорах AMD і IB. Мітка в кожного запису TLB містить інформацію про розмір сторінки на додачу до номера віртуальної сторінки. Логіка попадання / промаху використовує інформацію про розмір сторінки для вибору фактичного номера сторінки для порівняння тега віртуальної адреси. Повністю асоціативна конструкція переглядає всі свої записи і, як наслідок, збіг буде відбуватися, якщо в TLB присутній потрібний запис

трансляції адреси, незалежно від розміру сторінки. Коротше кажучи, асоціативна TLB спрощує підтримку декількох розмірів сторінок, оскільки вимагає лише мінімальної зміни записів в TLB для включення інформації про розмір сторінки. Записи TLB включають інформацію про розмір сторінки і вимагають, щоб логіка відповідала інформації. Однак повністю асоціативна конструкція часто працює повільніше, споживає більше енергії і займає більше простору на кристалі в порівнянні з набірний-асоціативної конструкцією з однаковою кількістю записів.

Крім того, повністю асоціативна конструкція вимагає стільки ж порівняльних пристроїв, скільки записів, щоб знайти збіг тегів. Також необхідний обширний (логічний) мультиплексор (вхід дорівнює кількості записів) для вибору співпадаючого запису. Висока затримка і накладні витрати живлення ускладнюють масштабування повністю асоціативного TLB на більшу кількість записів. Наскільки мені відомо, не існує комерційного процесора з більш ніж 64 записами в повністю асоціативної TLB.

Другий підхід для підтримки декількох розмірів сторінок використовує набірні асоціативні TLB. Набірно-асоціативні TLB можуть вирішити проблему масштабованості повністю асоціативної конструкції. У комерційних рішеннях Intel, використовуються набірний-асоціативні TLB. Сет-асоціативні конструкції більш масштабовані в порівнянні з повністю асоціативними, оскільки кількість компараторів не росте зі збільшенням кількості записів, а залишається рівним асоціативності. Однак набірно-асоціативна конструкція для TLB має свої недоліки. В звичайній системі віртуальної пам'яті розмір сторінки для відображення заданого адресного поля невідомий доти, поки не завершиться перетворення адреси. Це ускладнює вибір бітів адреси, які необхідно індексувати в набірно-асоціативній TLB. Частина номеру віртуальної сторінки повинна бути використана для індексації TLB із заданою асоціацією. Однак в залежності від розміру сторінки розташування номера віртуальної сторінки в адресних бітах може змінюватися.

## 1.4 Висновки

Переваги віртуальної пам’яті:

1) більше процесів може зберігатися в основній пам’яті, оскільки ми завантажуємо лише деякі сторінки якогось процесу, є місце для більшої кількості процесів. Це дає змогу більш ефективно використовувати процесор, оскільки, швидше за все один з процес з великої кількості в будь-який час буде напоготові;

2) скасовано одне з найбільш фундаментальних обмежень у програмуванні «процес може бути більшим за всю основну пам’ять». Цей процес може виконуватись через підкачку. Операційна система завантажує сторінки процесу в оперативну пам’ять за необхідності;

3) це дозволяє збільшити рівні мультипрограмування, використовуючи менше доступної пам'яті для кожного процесу.

Віртуальна пам’ять застосовується для покращення продуктивості системи, підвищення багатозадачності, використання великих програм. Коли використовується програма, її дані зберігаються в фізичній адресі. Віртуальна пам’ять відобразить цю адресу за допомогою блоку управління пам’яттю. Операційна система буде створювати і керувати відображенням пам’яті за допомогою таблиць сторінок та інших структурних даних. Блок управління виконує функцію обладнання для перекладу адрес, автоматично перекладає їх.

Якщо місце в оперативній пам’яті буде потрібно для більш приорітетної програми, дані можуть бути перемістити в віртуальну пам’ять. Якщо ці дані будуть потрібні знову, контекстний перемикач відновить потрібну задачу.

Під час копіювання віртуальної пам’яті у фізичну система ділить їх на файли сторінок або обмінюється файлами з фіксованою кількістю адрес. Сторінки зберігаються на диску, коли до них звертаються, операційна система копіює їх з диска в основну пам’ять і перетворює віртуальні адреси в реальні.

Однак процес заміни пам’яті відбувається повільно, що спричиняє помітне зниження продуктивності. Через обмін комп’ютери з більшою оперативною пам’яттю показують кращу продуктивність.

Недоліком віртуальної пам’яті являється високий ступінь мультипрограмування. Якщо кількість процесів в пам'яті постійно зростає, кількість кадрів, виділених для кожного з них, буде зменшуватися. Отже, для кожного процесу буде доступна менша кількість кадрів. У зв’язку з цим помилки сторінок траплятимуться частіше, процесор буде витрачати більше часу просто на заміну сторінок і виведення їх, а використання буде продовжувати зменшуватися. Ще одним недоліком є розмір віртуального сховища, який обмежується обсягом вторинної пам’яті, через що може статися збій, який призведе до сповільнення роботи системи.

Програми якими ми користуємось не знають про те, що вони використовують віртуальну пам’ять. Програмне забезпечення визначає де знаходиться адреса в оперативній пам’яті чи у файлі підкачки, і перетворює віртуальну адресу на фізичну. Якщо цієї адреси немає у оперативній пам’яті перетворити адресу не вдасться. Цю помилку також іменують як помилка сторінки. При виникненні цієї помилки система турбується про перенесення адреси в фізичну пам’ять. Як правило це передбачає обмін блоками адрес, тому що, це більш ефективний спосіб доступу до диска.

Задачі віртуальної пам’яті:

1) ізоляція та захист пам’яті шляхом створення власних віртуальних адресних комірок для процесів;

2) перетворення фізичних адресів у віртуальні;

3) розміщення даних в запам’ятовуючих пристроях;

4) транспортування даних між пам’яттю та диском;

5) вибір частин процесів чи їх образів для переміщення з фізичної пам’яті на диск і назад;

6) вивантаження частин пам’яті, яка не задіяна в область файлу підкачки.

# 2 ПРОЄКТУВАННЯ ПРОГРАМНО-ТЕХНІЧНОГО ЗАСОБУ

## 2.1 Методи аналізу віртуальної пам’яті

Щоб отримати перемінливу віртуальну пам'ять і проаналізувати її, спочатку потрібно володіти технікою отримання пам'яті. Існує два методи отримання віртуальної пам'яті: апаратне і програмне. Обидва методи, включаючи їх плюси і мінуси, будуть описані в цьому підрозділі. В цілому, краще використовувати апаратне метод, оскільки він більш надійний і хакеру важко його зіпсувати, але в даний час програмний є більш популярним методом через його економність і доступність.

Апаратний метод отримання пам’яті передбачає припинення роботи процесора комп'ютера і використання прямого доступу до пам'яті (DMA) для отримання копії. Цей метод вважається більш надійним, оскільки навіть, якщо операційна система і програмне забезпечення системи були зламані або пошкоджені зловмисником, ми все одно отримаємо точний образ пам'яті, оскільки ми не покладаємося на ці компоненти системи.

Недоліком цього методу є вартість необхідно придбати спеціальне обладнання, щоб виконати отримання пам'яті таким чином. Одним із спеціалізованих апаратних засобів, розроблених для цієї мети, є карта Tribble, яка являє собою PCI карту, яка повинна бути встановлена в систему до того, як відбудеться компрометація, щоб дозволити юзеру захопити пам'ять більш точним і надійним способом.

Збір даних на основі програмного забезпечення найчастіше здійснюється з використанням довіреної набору інструментів, який використовується програмістом, але також можливо зібрати перемінливу пам'яті за допомогою інструментів, вбудованих в операційну систему (наприклад, «memdump» або «dd» в системах Unix). Незалежно від того, чи є ці інструменти підключеними чи вже знаходяться в системі, зловмисники можуть легко обійти цей метод, якщо вони зламали операційну систему комп'ютера, який аналізується, оскільки можна приховати дані, змінити системні виклики і внутрішні структури системи.

### 2.1.1 Список запущених процесів

Один з перших методів аналізу віртуальної пам’яті, яку користувач може використати для відображення віртуальних процесів, - це список запущених процесів. Структурне подання процесу фактично аналогічно в більшості операційних системах, і загальна методологія відновлення списку запущених процесів з пам'яті також практично однакова. У цьому підрозділі розповідається про системи Linux, та Windows, щоб дати загальне уявлення про методи, які використовуються в більш поширених операційних системах.

У більшості версій Linux дескриптор процесу використовується для зберігання інформації про поточний стан кожного запущеного процесу і служить представленням цього процесу. Ця структура називається «task_struct», і використовується для представлення всіх типів процесів, починаючи з процесів, що викликаються користувачем, і закінчуючи потоками ядра.

У Windows структури процесів виглядають приблизно так само, як і в операційних системах Linux - є базова структура, в якій зберігається вся специфічна для процесу інформація, а список цих структур з подвійним зв'язком використовується для відстеження всіх запущених процесів в пам'яті. Пошук запущених процесів в Windows не такий простий, як в Linux, на жаль, тому що у аналітиків немає карти пам'яті, з якої можна почати пошук пов'язаного списку процесів. Замість цього аналітик зазвичай повинен знайти відправну точку, звернувшись до глобальних змінних ядра, які вказують на початок списку процесів.

### 2.1.2 Відновлення файлів, що відображаються в пам'яті

У Windows структури даних, що зберігають схеми відображення файлів, виділяються ядром з пулу пам'яті (динамічної області зберігання в пам'яті, що виділяється ядром). Існує кілька різних типів структур, які використовуються для опису файлового об'єкта. часто, ці структури знаходяться шляхом перегляду структури даних процесу і пошуку файлів, зіставлених з цим процесом. Найкраще

почати з кореня дерева VAD, яке представляє собою набір структур, який описує діапазони пам'яті процесів. Одна зі структур в дереві VAD називається таблиця об'єктів, в якій перераховані приватні об'єкти, що використовуються процесом - це можуть бути файли, ключі реєстру і події. Зіставлені з пам'яттю файли, пов'язані з кожним процесом, можна відновити, пройшовши по дереву VAD і витягнувши потрібні об'єкти - в даному випадку файли, але потенційно можуть бути і інші об'єкти.

Як і у випадку з завершеними процесами, файли, які були закриті, часто залишаються в пам'яті, але більше не пов'язані зі списками, які веде операційна система, і їх доводиться шукати альтернативними методами. Процес відновлення таких файлів схожий на відновлення файлів на жорсткому диску, які були видалені, хоча той факт, що пам'ять зазвичай набагато більш фрагментована, ніж жорсткий диск, робить цей процес більш складним. Часто перегляд таблиці сторінок дозволяє відновити файли, навіть якщо вони більше не активні в пам'яті. Існує також область пам'яті під назвою "Область управління", яка підтримує зв'язки між іменами файлів і файловими даними, що зберігаються в сторінках.

У Linux файли, які відображаються в пам'яті, описуються структурами «inode», які представляють собою ту ж структуру, яка використовується для опису файлів, що зберігаються на жорсткому диску. Використовуючи цей об'єкт, можна отримати інформацію про каталог, з якого запускався файл. Існує безліч ресурсів, що описують детальну структуру «inode» і перераховують всю інформацію, яку можна з неї витягнути. Щоб знайти файли, які відображаються в пам'яті, аналітик може спочатку перерахувати процеси і для кожного з них переглянути файлові структури, пов'язані з цим процесом. Таким чином можна знайти всі файли, пов'язані з процесом.

### 2.1.3 Пошук підпису файлу

Різні типи файлів (наприклад, документи word або файли, стислі в форматі zip) мають різні сигнатури. В даному випадку сигнатура ставиться до певних шаблонних значень, які є унікальними для конкретного типу файлу. Вирізання файлів може здійснюватися лінійно, що означає, що цілісні файли можуть бути відновлені, а фрагментованих - немає. Існують також більш складні алгоритми вирізання файлів, які здатні відновлювати фрагментовані файли шляхом більш глибокого вивчення структур даних, які їх описують.

Вирізання файлів дуже корисно при аналізі диска, оскільки більшість операційних систем прагнуть не фрагментувати файли, що дозволяє легко відновити їх за допомогою простих лінійних методів. На жаль, часто дуже важко використовувати вирізання файлів для відновлення даних з дампа пам'яті, оскільки ймовірність того, що файли будуть безперервними в пам'яті, набагато нижче, ніж на диску. Навіть якщо використовується більш розумний алгоритм, який може обробляти фрагментацію, зазвичай в пам'ять завантажуються тільки частини файлів, що може збити інструмент з пантелику і перешкодити йому знайти потрібні частини файлу і відновити їх.

Хоча вирізання файлів є хорошою технікою і дуже корисною, яку необхідно мати в інструментах кожного аналітика, вона не є оптимальною, і для досягнення найкращих результатів слід використовувати методи, описані в попередньому підрозділі, коли це можливо. Існує безліч інструментів, які можуть виконувати вирізання файлів, наприклад PTFinder.

### 2.1.4 Виявлення та відновлення прихованих даних

У підрозділах 2.2 і 2.3 ми говорили про відновлення процесів, потоків і файлів, що відображаються в пам'яті, працюючи зі списками цих об'єктів, які веде операційна система. Процедури, обговорювані в цих підрозділах, корисні і важливі для розуміння, але аналітики повинні також розуміти, що ці методи не зможуть знайти всі процеси або потоки. Це відбувається тому, що після

завершення процесу або якщо операційна система отримує вказівку приховати процес, структура даних, яка визначає процес, більше не буде членом структури даних пов'язаного списку, яку операційна система підтримує для відстеження, що виконується в даний момент.

Основна передумова для пошуку об'єктів в пам'яті, які недоступні через активні в даний момент списки об'єктів, які використовуються ядром, досить проста. Всі типи об'єктів, такі як процеси або файли, мають свої шаблони - наприклад, заголовок кожного об'єкта процесу буде містити деякі константи, які будуть однаковими для кожного процесу в пам'яті. Щоб знайти процеси, яких немає в списку з подвійним зв'язком, використовуваному для посилань на процеси, які запущені в даний момент, аналітик повинен пройти через весь образ пам'яті і знайти ці постійні значення і використовувати їх в якості орієнтира, щоб вказати на об'єкти процесів, які в іншому випадку були б пропущені.

Це, звичайно, дуже виснажливий процес, який потрібно виконувати вручну. На щастя, існує безліч інструментів, написаних для автоматизації пошуку таких поширених об'єктів, як процеси і файли. Хоча вмотивованим аналітикам може знадобитися написати власний сценарій для пошуку чогось незвичайного, для чого ніхто не створив відкритої або комерційної програми, в цілому, особливо для поширених операційних систем, таких як Linux і Windows, можна знайти програми, які розберуть цікаві елементи.

Зараз ми детально розглянемо деякі приклади характеристик, які використовуються наявними автоматизовані інструменти для пошуку прихованих об'єктів процесів. Процеси представлені в Windows структурою EPROCESS, яка містить інші значення і структури, що описують важливу інформацію, необхідну для запущеного процесу. Покажчик DirectoryTableBase вказує на початок відповідних структур і може бути використаний для перегляду інформації, щоб знайти відповідні частини, та звірити їх із загальним процесом, яка потрібна, щоб допомогти виявити приховані процеси. Однією з перших перевірок є значення параметра PageDirectoryTable. Воно не повинно дорівню нулю, а якщо взяти по модулю 4096, то значення має дорівнювати нулю. Крім того, кожному процесу

потрібен хоча б один потік; потоки зберігаються в структурі, яка, по суті, є ще одним двічі пов'язаним списком.

Два покажчика (звані ThreadListHead.Flink і ThreadListHead.Blink) потрібні, щоб переконатися, що вони вказують на адресу, що перевищує 0x7fffffff. Це означає, що обидва покажчика повинні вказувати на простір ядра (верхня частина адресного простору пам'яті), де зберігається структура. Існують і інші важливі перевірки, які повинні бути виконані, і які, в разі виявлення збігу, збільшують ймовірність того, що об'єкт в пам'яті дійсно є прихованим процесом. Таким чином, дотримуючись цих правил для розбору пам'яті, можна знайти приховані процеси в пам'яті.

## 2.2 Існуючі рішення

У цьому підрозділі описуються і аналізуються деякі з сучасних інструментів, доступних аналітикам, зацікавленим у проведенні аналізу пам'яті. Цей аналіз зосереджений в основному на доступних інструментах, які може отримати і використовувати будь-який аналітик. Існують також комерційні інструменти, які коротко розглядаються в підрозділі 2.2.9. Це лише підмножина доступних в даний час інструментів, і його не слід вважати вичерпним списком. Крім того, розгляд повного спектра можливостей кожного інструмента виходить за рамки даної роботи.

Також хотів би відзначити, що при випробуванні нових інструментів або спробі визначити точність інструменту, вкрай важливо проводити тестування на достовірній базовій лінії. Наприклад, якщо аналітик пробує інструмент, який аналізує приховані процеси з пам'яті, і у нього мало фактичної інформації про роботу інструменту, йому слід протестувати інструмент на відомих образах пам'яті. Іншим способом тестування нових інструментів є використання аналогічного інструменту, робота якого була доведена, в якості основи для порівняння, щоб аналітик міг визначити, пропускає інструмент критичну інформацію або дає зворотні помилкові спрацьовування. Ця робота дуже важлива, оскільки якщо не вдається довести, що інструмент працює правильно в будь-якій

ситуації, докази і аналіз, проведені за допомогою цього інструменту, не працюватимуть.

Тут важливо відзначити, що багато з інструментів, які обговорюються в даному розділі, мають дуже мало документованої і доступної інформації. У наступних розділах зроблена спроба зібрати наявну інформацію, а в деяких випадках протестувати інструмент для визначення додаткову функціональність і простоту використання.

### 2.2.1 Базові інструменти

Існують деякі інструменти загального призначення, які можуть бути корисні при аналізі віртуальної пам'яті. Прикладом може служити інструмент netstat, який перераховує різноманітну інформацію про активні мережеві з'єднання, lsof - список файлів, відкритих на даній машині, ps - список запущених процесів і пов'язаної з ними інформації, ifconfig - список деталей конфігурації мережевого інтерфейсу.

Інші інструменти, які не часто зустрічаються в системах, але можуть бути встановлені системним адміністратором, включають пакет Sysinternals, підтримуваний Microsoft, інструменти Foundstone і набори ресурсів для Windows. Ці інструменти можуть запропонувати засоби для вилучення специфічної інформації про процеси, відкриті файли і т.д. з пам'яті, в залежності від того, що шукає користувач. Жоден з цих інструментів не був спеціально розроблений для аналізу, але вони все ж можуть запропонувати деякі цінні можливості.

Як штатні засоби системного адміністрування, так і додаткові набори інструментів, що пропонуються різними фірмами, корисні і часто використовуються системними адміністраторами для перевірки стану системи. Вони також використовуються деякими фахівцями для збору інформації з віртуальної пам'яті, щоб аналітик міг використовувати її в ході дослідження. Однак існує кілька проблем з використанням цих інструментів для аналізу пам’яті.

Аналітик ніколи не повинен використовувати інструменти на самій системі, оскільки, якщо зловмисник зламав систему, виконувані файли могли бути змінені, щоб повернути неправдиву інформацію або приховати що-небудь. Більшість аналітиків беруть з собою копію власних інструментів на CD або USB-накопичувачі, щоб знизити цей ризик. Однак навіть при такому підході важливо відзначити, що запуск інструментів може призвести до перезапису інформації, пов'язаної з аналітикою. Цього ризику важко уникнути, якщо у фахівця, немає апаратного пристрою збору даних.

### 2.2.2 Memdump, KnTTools

Memdump – це безкоштовний інструмент, який працює на більшості поширених операційних системах, включаючи Windows та Linux. Він легко компілюється, та простий у використанні, по суті вся його функціональність зводиться до створення побітової копії віртуальної пам’яті. В ідеалі цей інструмент повинен записуватись з системи де відбувається аналіз. Один зі способів це виконати, - використати netcat наступним чином: «memdump | nc host port», де host це IP-адреса системи, на якій працює netcat на порту вказаному в якості останнього аргументу.

KnTTools – це інструмент для збору і аналізу пам’яті, створений для використання в операційних системах Windows. Компонент збору даних KnTDD може захоплювати фізичну пам'ять і зберігати її на знімному диску або передавати по мережі для архівування на окремій машині. Отримані дані можуть бути стиснуті в кілька різних форматів. Він також здатний конвертувати бінарний фрагмент в форматі Microsoft для аварійних дампів, що може бути корисно для аналітиків, що вважають за краще формат аварійних дампів Microsoft. Аналітичний компонент пакета KnTTools називається KnTList і витягує докази з захопленої пам'яті шляхом реконструкції відповідних структур даних операційної системи Windows. Він може виводити звітну інформацію в форматі XML, щоб полегшити аналіз отриманих даних.

### 2.2.3 FATKit

FATKit - популярний інструмент для аналізу віртуальної пам'яті, який автоматизує процес вилучення цікавих даних з вторинної пам'яті. Після отримання даних FATKit також має можливість візуалізувати знайдені об'єкти, щоб допомогти аналітику зрозуміти дані, які вдалося знайти. Інструмент здатний аналізувати структури, характерні для ядер Linux і Windows. Оскільки інструмент є модульним, він легко розширюється користувачем, який хоче отримати додаткову підтримку різних операційних або файлових систем.

На нижчому рівні FATKit використовує кілька різних методів для надання корисної інформації аналітику. Він здатний реконструювати віртуальні адресні простори, використовувані процесами, і переводити віртуальні та фізичні адреси для отримання точної картини того, де насправді перебували дані в пам'яті під час роботи машини. Ще однією корисною функцією FATKit є здатність виявляти шкідливий код, що знаходиться в віртуальній пам'яті.

### 2.2.4 WMFT

Інструментарій Windows Memory Forensic Toolkit (WMFT) підтримує аналіз образів пам'яті машин під управлінням Windows 7, Windows 8 і Windows 10. Існує також версія для Linux, але її функціональність в даний час дещо обмежена в порівнянні з версією для Windows.

Для того щоб використовувати WMFT, аналітик повинен спочатку знайти символи, які вказують на важливі об'єкти і структури в пам'яті. Знаходження символів - це найскладніша частина, і вона описана в розповсюджуваній статті «Введення в експертизу пам'яті Windows». Після того як символи знайдені, аналітик може вставити значення (відповідні місцях в пам'яті) в WMFT і проаналізувати структури даних, щоб відновити процеси та інші об'єкти, що зберігаються в пам'яті.

Так як WMFT використовує самі структури для обходу пам'яті і вилучення відповідну інформацію, він вразливий для просунутих атак, коли зловмисник

приховує процеси та інші об'єкти, залишаючи їх поза структурами даних, таких як пов'язані списки, які використовуються ядром для відстеження таких даних.

### 2.2.5 Procenum

Перерахування процесів користувацького режиму є однією з основних можливостей, необхідних для проведення будь-якого аналізу вторинної пам'яті. Procenum представляє собою утиліту, яка буде виконувати цю дію, переглядаючи всі дескриптори сторінок, виділені операційною системою.

Техніка, яка використовується procenum, є дуже загальною, що робить її широко ефективної проти процесів, які зловмисник приховав за допомогою виправлення коду, модифікації покажчиків функцій, або за допомогою прямого маніпулювання об'єктами ядра.

### 2.2.6 Idetect

Idetect - це інструмент тільки для linux, який переглядає образ пам'яті і намагається витягти детальну інформацію про активні процеси. Він також може дозволити дізнатися вміст файлу, якщо цей файл був відображений в пам'яті раніше. Будь-яка структура, яка відноситься до процесу, який цікавить користувача, може бути перевірена за допомогою idetect. Цей інструмент можна використовувати як проти живих систем, так і проти образів пам'яті, створених під час ліквідації наслідків інциденту, що, хоча і не ідеально для більшості робіт, але забезпечує певну гнучкість в разі, якщо якісь обставини не дозволяють вивести систему в автономний режим.

### 2.2.7 Volatility Framework

Volatility - це набір інструментів, призначених для використання в рамках реагування на інциденти, де необхідний або бажаний аналіз віртуальної пам'яті. Він є безкоштовним, з відкритим вихідним кодом і написаний на мові сценаріїв

Python. Він надає платформу для аналізу і вилучення об'єктів з дампов пам'яті і підтримує кілька операційних систем, включаючи Linux, OSX 10.5 і Windows.

Система Volatility підтримує широкий спектр команд, включаючи команди, які виводять список відкритих мережевих з'єднань, список відкритих DLL-файлів, карту пам'яті, пов'язану з аналізованим дампом пам'яті, список відкритих файлів, пов'язаних з процесом, і багато іншого. Однією з цікавих функцій програми (особливо для аналітиків шкідливих програм) є її здатність відновлювати і виписувати зразок виконуваного файлу з пов'язаного ним процесу.

### 2.2.8 VAD Tools

Структури Virtual Address Descriptor містять багато цінної інформації про процеси і виділених ними структурах (таких як зіставлені файли). VAD Tools – це набір скриптів, написаних на мові python, які здатні аналізувати цю інформацію в пошуках того, що становить інтерес для аналітика.

Використовується п'ять скриптів: «vadwalk.py, vadinfo.py, vaddump.py, procdump.py і listdll.py». Скрипт vadwalk.py переглядає дерево VAD і виводить його у вигляді таблиці або ASCII-дерева; він також може створити файл GraphViz, який можна завантажити в візуалізатор для відображення дерева. Сценарій vadinfo.py, як випливає з назви, виводить детальну інформацію з VAD, включаючи файли, відображені в адресний простір процесу, такі як DLL. Сценарії vaddump.py і procdump.py витягають області пам'яті з дерева VAD - procdump.py специфічний для виконуваних файлів, таких як файли .dll і .exe, які зберігаються в образі пам'яті. Нарешті, listdll.py роздруковує список всіх модулів (DLL), які завантажені для певного процесу.

Інструменти VAD є просунутими і низькорівневими, але надзвичайно корисними для розбору структур даних в пам'яті і вилучення інформації. Зокрема, можливість відновити exe-файл або dll-файл є корисною можливістю.

### 2.2.9 Доступні комерційні інструменти

Існує безліч комерційно доступних інструментів, які виконують велику частину функцій, і можуть використовуватися замість безкоштовних інструментів з відкритим вихідним кодом, описаних раніше в цьому розділі. У цьому підрозділі коротко описані кілька особливо відомих продуктів.

Encase Enterprise, один з найбільш широко використовуваних інструментів віртуального аналізу, має утиліту «Snapshot», яка захоплює віртуальні дані, що існують в оперативній пам'яті на певних типах систем, і розбирає ці дані, щоб вони були організовані і більш значущі для аналітика. В даний час Encase не надає аналітику необроблений знімок даних, тому аналітики повинні покладатися на програмне забезпечення, щоб правильно інтерпретувати пам'ять і знайти все, що представляє інтерес.

F-Response - це ще один інструмент, який дозволяє отримати віддалений доступ тільки для читання фізичної пам'яті машини, що цікавить аналітика. Він не здійснює фактичне захоплення даних (для цього аналітику потрібно інша програма).

HBGary - це дуже потужний інструмент, який виконує аналіз пам'яті. Аналіз пам'яті і зворотне проектування шкідливого ПО, яке витягується з пам'яті. Шкідливе ПО являє особливий інтерес - він дозволяє отримати підозрілий виконуваний файл, знайдений в пам'яті, витягнути та аналізувати код і просканувати на предмет підозрілого коду і функціональності. HBGary також має службову програму FastDump, яка доступна безкоштовно і може бути використана для захоплення фізичної пам'яті.

Ці комерційні інструменти корисні, і в деяких випадках їх функціональність легше використовувати, ніж інструменти з відкритим вихідним кодом, або вони володіють можливостями, яких не вистачає інструментам з відкритим вихідним кодом. Тим не менш, важливо відзначити, що для проведення ретельного та ефективного аналізу віртуальної пам'яті не обов'язково мати дорогі комерційні інструменти.

# 3 ПРОГРАМНО-АПАРАТНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНО-ТЕХНІЧНОГО ЗАСОБУ

## 3.1 Необхідність в управлінні пам'яттю

Управління пам'яттю одна з найбільш фундаментальних областей в програмуванні. У більшості скриптованих мов, процес управління пам’яттю виконується автоматично, але це не робить цей механізм менш значущим. Знання про можливості менеджера пам'яті (memory manager) і тонкощах його роботи, є запорукою ефективного програмування. У більшості системних мов, наприклад таких як C / C ++, розробнику необхідно самому стежити за кількістю використаної пам'ятті. У третьому розділі йде мова про ручні, напівавтоматичні і автоматичні методи управління пам'яттю.

Були часи, коли управління пам'яттю не було великою проблемою. Як приклад можна згадати часи розробки Apple II на асемблері. В основному програми запускалися не окремо від ОС, а разом з нею. Будь-яка ділянка пам'яті могла використовуватися як системою, так і розробником. Не було необхідності в розрахунку загального обсягу пам'яті, тому що вона була однакова для всіх комп'ютерів. Так що вимоги до пам'яті були досить статичні - необхідно було просто вибрати ділянку пам'яті і використовувати її.

Однак навіть в такій простій обчислювальній машині були свої проблеми, особливо, якщо ви не знали скільки пам'яті може знадобитися окремій взятій ділянці програми.

Якщо є обмеження, пов'язані з пам'яттю, то необхідний підхід, який буде займатися вирішенням таких завдань до вимог пам’яті, як:

1) визначення достатнього об’єму пам’яті;
2) отримання секції з доступної пам’яті;
3) повернути секцію назад в масив, щоб була можливість використовувати її в інших програмах, чи їх частин.

Бібліотеки, які займаються пошуком, виділенням, звільненням пам'яті називаються allocator-ми. З ростом складності програми, підвищується складність

управління пам'яттю і тим самим підвищується роль allocator-а в такій програмі. В наступних підрозділах будуть показаьт різні методи управління пам'яттю, розглянуто їх переваги та недоліки, а також ситуації, де вони найбільш ефективні.

Allocator – мова програмування С має підтримку двох функцій, які займаються вирішенням завдань вимог пам’яті:

1) malloc – виділяє задану кількість біт і повертає покажчик. Якщо пам’яті не вистачає, повертається покажчик *NULL*;

2) free - приймає на вході покажчик області пам'яті, виділеної за допомогою malloc і повертає її для подальшого використання в програмі або операційній системі (насправді, деякі malloc-функції повертають пам'ять для подальшого використання тільки програмою, але не ОС).

## 3.2 Ієрархія пам’яті

Сучасна система пам'яті утворює ієрархію від швидких типів пам'яті маленького розміру до повільних типів пам'яті великого розміру. Ми говоримо, що конкретний рівень ієрархії є кешем для даних, розташованих на більш низькому рівні. Це означає, що він містить копії даних з більш низького рівня. Коли процесор хоче отримати якісь дані, він їх спершу шукає на найшвидших високих рівнях. І спускається на більш низькі, якщо не може знайти.

На вершині ієрархії знаходяться регістри процесора. Доступ до них займає 0 тактів, але їх всього кілька штук. Далі йде кілька кілобайт кеш-пам'яті першого рівня, доступ до якої займає приблизно 4 такта. Потім йде пара сотень кілобайт повільнішої кеш-пам'яті другого рівня. Потім кілька мегабайт кеш-пам'яті третього рівня. Вона набагато повільніша, але все одно швидше оперативної пам'яті. Далі розташована відносно повільна оперативна пам'ять. Приклад ієрархії наведено в таблиці 3.1.

Оперативну пам'ять можна розглядати як кеш для локального диска. Диски це робочі конячки серед пристроїв зберігання. Вони великі, повільні і коштують дешево. Комп'ютер завантажує файли з диска в оперативну пам'ять, коли збирається над ними працювати. Розрив у часі доступу між оперативною пам'яттю

і диском колосальний. Диск повільніше оперативної пам'яті в десятки тисяч разів, і повільніше кеша першого рівня в мільйони разів. Вигідніше звернутися тисячу раз до оперативної пам'яті, ніж один раз до диска. На це спираються такі структури даних, як *B*-дерева, які намагаються розмістити більше інформації в оперативній пам'яті, намагаючись уникнути звернення до диска за всяку ціну.

Локальний диск сам може розглядатися як кеш для даних, розташованих на віддалених серверах. Коли ви відвідуєте веб-сайт, ваш браузер зберігає зображення з веб-сторінки на диску, щоб при повторному відвідуванні їх не потрібно було качати. Існують і більш низькі ієрархії пам'яті. Великі датацентри, типу Google, зберігають великі обсяги даних на дискових носіях, які зберігаються десь на складах, і коли знадобляться, повинні бути приєднані вручну або роботом.

Таблиця 3.1 - Сучасна ієрархічна система

| Тип кеша | Час доступу | Розмір кеша |
|---|---|---|
| Регістри | 0 | Десятки штук |
| L1 кеш | 4 | 32 KB |
| L2 кеш | 10 | 256 KB |
| L3 кеш | 50 | 8 MB |
| Оперативна пам’ять | 200 | 8 GB |
| Буфер диска | 100'000 | 64 MB |
| Локальний диск | 10'000'000 | 1000 GB |
| Видалені сервера | 1'000'000'000 | ∞ |

Швидка пам'ять коштує дуже дорого, а повільна дуже дешево. Це велика ідея архітекторів систем поєднати великі розміри повільної і дешевої пам'яті з маленькими розмірами швидкої і дорогої. Таким чином система може працювати на швидшій пам'яті і коштувати як повільна.

Наприклад, ваш комп'ютер має 8 ГБ оперативної пам'яті і диск розміром 1000 ГБ. Але подумайте, що ви не працюєте з усіма даними на диску в один момент. Ви завантажуєте операційну систему, відкриваєте веб-браузер, текстовий редактор, пару-трійку інших додатків і працюєте з ними кілька годин. Всі ці програми поміщаються в оперативній пам'яті, тому вашій системі не потрібно звертатися до диска. Потім, звичайно, ви закриваєте один додаток і відкриваєте інший, яке потрібно завантажити з диска в оперативну пам'ять. Але це займає пару секунд, після чого ви кілька годин працюєте з цим додатком, не звертаючись до диска. Ви не особливо помічаєте низьку швидкодію диска, тому що в один момент ви працюєте тільки з невеликим об’ємом даних, які кешуються в оперативній пам'яті. Вам немає сенсу витрачати величезні гроші на установку 1024 ГБ оперативної пам'яті, в яку можна було б завантажити вміст всього диска. Якби ви це зробили, ви б майже не помітили ніякої різниці в роботі, це були б «гроші на вітер».

## 3.3 Дослідження віртуальної пам’яті

Програма завантажується під конкретну ОС, саме вона вирішує, скільки пам'яті виділити під ту чи іншу програму. Кожен запущений процес вважає, що має доступ до всієї фізичної пам'яті комп'ютера. Очевидним є той факт, що одночасно працює безліч процесів, і кожен з них не може мати доступ до всієї пам'яті. Отже, як змінюється виділення пам’яті завдяки використанню віртуальної пам'яті.

Як приклад, припустимо програма звертається до 629-ої ділянки в пам'яті. Система віртуальної пам'яті не гарантує, що дані зберігаються в RAM за адресою 629. Фактично, це може бути навіть не RAM, - дані могли бути перенесені на диск, якщо RAM не залишилось місця.

Тобто в віртуальній пам'яті можуть зберігатися адреси, які відповідають фізичному пристрою. ОС зберігає таблицю відповідностей віртуальних адрес до фізичних, щоб комп'ютер міг правильно реагувати на запит за адресою. Якщо RAM зберігає фізичні адреси, то ОС буде змушена тимчасово призупинити процес, вивантажити частину даних на диск (з RAM), довантажити необхідні дані для роботи процесу з диску і перезапустити процес. Таким чином, кожен процес отримує свій адресний простір, яким може оперувати і отримувати ще більше пам'яті, ніж йому було виділила ОС.

У 32-х бітних додатках (архітектура x86), кожен процес може працювати з 4 гігабайтами пам'яті. На даний момент більшість користувачів не володіють таким обсягом. Навіть якщо використовується підкачка (swap), все одно повинно вийти менше 4 Гб на процес. Таким чином, коли процес вивантажується в пам'ять, йому виділяється певний простір. Кінець цієї ділянки пам'яті іменується як system break.

За цією межею знаходиться нерозмічена пам'ять, тобто без проекції на диск або RAM. Тому коли у процесу закінчується пам'ять (з тієї, що йому була виділена при завантаженні) він повинен запитати у ОС більший шматок пам'яті. Mapping - це математичний термін, що означає відповідність один до одного, тобто коли по віртуальній адресі зберігається адреса на диску, за яким вже зберігаються реальні дані.

### 3.3.1 Системна інформація

У попередньому розділі було з’ясовано, як система керує віртуальною пам'яттю, як процес отримує свій адресний простір і що воно собою являє. Розглянемо деякі Windows-функції, які повідомляють про стан системної пам'яті і віртуального адресного простору в тому чи іншому процесі.

Багато параметрів операційної системи (розмір сторінки, гранулярність, виділення пам’яті та інші) залежать від процесора, який використовує обчислювальна машина. Тому не можна «зашивати» їх значення в вихідний код

программ. Цю інформацію треба зчитувати в момент ініціалізації процесу за допомогою функції GetSysteinfo.

Потрібно передати в GetSystemInfo адресу структури SYSTEM_INFO, і функція ініціалізує елементи цієї структури:

```
typedef struct _SYSTEM_INFO
{
        union
    {
        struct
    {
                    WORD wProcessorArchitecture;
                WORD wRescrved;
                };
        };
        DWORD dwPageSize;
    LPVOID lpMinimurnAppliration Address;
    LPVOID lpMaximumApplicationAddress;
    DWORD_PTR dwActiveProcessorMask;
    DWORD dwNumberOfProcessors;
    DWORD dwProcessorType;
    DWORD dwAllocationGranularity;
    WORD wProcessorLevel;
    WORD wProcessorRpvi4inn;
    } SYSTEM INFO *LPSYSlEM_INFO;
```

При завантаженні система визначає значення елементів цієї структури, для конкретної системи, їх значення постійні. Функція GetSystemInfo передбачена спеціально для того, щоб і додатки могли отримувати цю інформацію. З усіх елементів структури лише чотири мають відношення до пам'яті. Вони описані в таблиці 3.2.

Таблиця 3.2 - Системна інформація

| № п.п | Назва функції | Опис функції |
|---|---|---|
| 1 | dwPageStze | Розмір сторінки пам'яті. На процесорах x86 це значення дорівнює 4096, а на процесорах Alpha - 8192 байт. |
| 2 | IpMinimumApplicationAddress | Мінімальна адреса пам'яті доступного адресного простору для кожного процесу. У Windows 98 це значення дорівнює 4 194 304, або 0x00400000, оскільки нижні 4 Мб адресного простору кожного процесу недоступні. У Windows 2000 це значення дорівнює 65 536, або 0x00010000, так як в цій системі резервуються лише перші 64 Кб адресного простору кожного процесу. |
| 3 | lpMaximwnApplicationAddress | Максимальна адреса пам'яті доступного адресного простору, відведеного в "особисте користування" кожному процесу. У Windows 2000 ця адреса відповідає початку розділу для коду та даних режиму ядра за вирахуванням 64 Кб. |
| 4 | dwAllocationGranularity | Гранулярність резервування адресного простору. |

Кінець таблиці 3.2 - Системна інформація

| | | |
|---|---|---|
| 5 | dwNumberOfProccecors | Кількість процесорів на компютері |
| 6 | dwActiveProcessorMask | Бітова маска, яка повідомляє, які процесори активні (виконують потоки) |

3.3.2 Статус віртуальної пам’яті

Функція GlobalMemoryStatus дозволяє відстежувати поточний стан пам’яті. Перед викликом GlobalMemoryStatus треба записати в елемент dwLength розмір структури в байтах. Такий принцип виклику функції дає можливість Microsoft розширювати цю структуру в майбутніх версіях Windows, не порушуючи роботу існуючих додатків. Після виклику GlobalMemoryStatus встановлює інші елементи структури і повертає управління:

```
typedef struct _MEMORYSTATUS
{
    DWORD dwLength;
    DWORD dwMemoryLoad;
    SIZE_T dwTotalPhys;
    SIZE_T dwAvailPhys;
    SIZE_T dwTotalPageFile;
    SIZE_T dwAvailPageFile;
    SIZE_T dwTotalVirtual;
    SIZE_T dwAvailVirtual;
}
MEMORYSTATUS, *LPMEMORYSTATUS;
```

Якщо потрібно, щоб виконувана програма працювала на системі з об'ємом оперативної пам'яті понад 4 Гб або файлом підкачки більше 4 Гб, доцільно використовувати нову функцію GlobalMemoryStatusEx:

```
typedef struct _MEMORYSTATUSEX
{
    DWORD dwLength;
    DWORD dwMemoryLoad;
    DWORDLONG ullTotalPhys;
    DWORDLONG ullAvailPhys;
    DWORDLONG ullTotalPageFile;
    DWORDLONG ullAvaiIPageFile;
    DWORDLONG ullTotalVirtual;
    DWORDLONfi ullAvailVirtual;
    DWOHDLONG ullAvailExtendedVirtual;
    }
    MEMORYSTATUSEX, *LPMEMORYSTATUSEX;
```

Ця структура ідентична первісній структурі MEMORYSTATUS з одним винятком всі її елементи мають розмір по 64 біта, що дозволяє оперувати зі значеннями, що перевищують 4 Гб. Останній елемент, ullAvailExtendedVirtual, вказує розмір незарезервірованої пам'яті в найбільшій області пам’яті віртуального адресного простору в якому відбувається виклик процесу. Цей елемент слід використовувати для процесорів певних архітектур при певних конфігураціях.

### 3.4 Створення консольного модуля для відображення стану пам’яті

#### 3.4.1 Підключення бібліотек

Для правильного функціонування коду підключаю необхідні біліотеки, наведенні в таблиці 3.3.

Таблиця 3.3 - Бібліотеки

| № п.п | Бібліотека | Використання |
|---|---|---|
| 1 | #include <windows.h> | Файл заголовків, який містить реалізацію для всіх функцій в Windows API, загальні макроси, типи даних, що використовуються різними функціями і підсистемами. |
| 2 | #include <chrono> | Визначення класів і функцій, які представляють і управляють тривалістю часу і моментами часу. |
| 3 | #include <iostream> | Оголошує об'єкти, керуючи читанням зі стандартних потоків і записом в них. Це включення часто є єдиним заголовком, який необхідний для виконання введення та виведення з програми на C ++. |
| 4 | #include <thread> | Визначає об'єкт, який використовується для спостереження та управління потоком виконання в додатку |

### 3.4.2 Оголошення констант

Оголошення змінних, які використовуються в програмному додатку наведені в таблиці 3.4.

Таблиця 3.4 - Константи

| № п.п | Константа | Використання |
|---|---|---|
| 1 | #define DIV 1024 | Використовується для перетворення байтів у КБ |
| 2 | #define MESSAGE_WIDTH 30<br>#define NUMERIC_WIDTH 10 | Вказують ширину поля, в якому будуть друкуватися числа. |

### 3.4.3 Функції

Створення власних функцій для відображення стану пам’яті, подані в таблиці 3.5.

Програмний код:

```
void printMessageLine(LPCSTR msg, DWORD value)
{
   printMessage(msg, true);
   cout.width(NUMERIC_WIDTH);
   cout << right << value << " %" << endl;
}

void printMessageLine(LPCSTR msg, DWORDLONG value)
{
   printMessage(msg, true);
   cout.width(NUMERIC_WIDTH);
   cout << right << value << " Kb" << endl;
}

void checkInput(HANDLE exitEvent)
{
   for (;;)
   {
```

```
        char character;
        cin.get(character);
        if (character == 'x')
        {
           ::SetEvent(exitEvent);
           break;
        }
     }
  }
```

Таблиця 3.5 - Власні функції

| № п.п | Функції | Опис |
|---|---|---|
| 1 | printMessageLine | Функції виведення в консоль, в змінну передається довжина рядка та змінна. |
| 2 | checkInput | Функція зупинки моніторингу пам’яті, при нажатті на «x» завершує моніторинг |

3.4.4 Ініціалізація змінних та виведення в консоль

Для роботи з пам’яттю використанно функції наведенні в таблиці 3.6.

```
int main(int argc, char** argv) {
setlocale(LC_ALL, "Ukrainian"); // підключення української мови
MEMORYSTATUSEX statex; // ініціалізація змінної
statex.dwLength = sizeof(statex); // встановлення довжини
BOOL success = ::GlobalMemoryStatusEx(&statex); // булева функція

if (!success) // якщо не вдалось ініціалізувати монітор, вивести помилку
        {
           DWORD error = GetLastError();
           printMessageLine("Помилка отримання інформації", error);
        }
 else // запустити процес моніторингу
```

```
        {
            DWORD load = statex.dwMemoryLoad;
            DWORDLONG physKb = statex.ullTotalPhys / DIV;
            DWORDLONG freePhysKb = statex.ullAvailPhys / DIV;
            DWORDLONG pageKb = statex.ullTotalPageFile / DIV;
            DWORDLONG freePageKb = statex.ullAvailPageFile / DIV;
            DWORDLONG virtualKb = statex.ullTotalVirtual / DIV;
            DWORDLONG freeVirtualKb = statex.ullAvailVirtual / DIV;
            DWORDLONG freeExtKb = statex.ullAvailExtendedVirtual / DIV;

        }
    }
```

Таблиця 3.6 - Функції роботи з пам'яттю

| № п.п | Назва функції | Опис функції |
|---|---|---|
| 1 | dwMemoryLoad | Число від 0 до 100, що визначає приблизний відсоток використовуваної фізичної пам'яті |
| 2 | ullTotalPhys | Обсяг фактичної фізичної пам'яті, в байтах. |
| 3 | ullAvailPhys | Обсяг фізичної пам'яті, доступної в даний час, в байтах. Це обсяг фізичної пам'яті, який може бути негайно використаний повторно без попереднього запису його вмісту на диск. Це сума розмірів резервного, вільного і нульового списків. |
| 4 | ullTotalPageFile | Поточний розмір встановленого ліміту пам'яті, в байтах. Це фізична пам'ять плюс розмір файлу сторінок, мінус невеликі накладні витрати. |

Кінець таблиці 3.6 - Функції роботи з пам'яттю

| | | |
|---|---|---|
| 5 | ullAvailPageFile | Максимальний обсяг пам'яті, який може зайняти поточний процес, в байтах. Це значення має бути менше, ніж загальносистемний доступний обсяг фіксації. |
| 6 | ullTotalVirtual | Розмір частини віртуального адресного простору викликає процес в режимі користувача, в байтах. Це значення залежить від типу процесу, типу процесора і конфігурації операційної системи. |
| 7 | ullAvailVirtual | Обсяг незарезервованої і нефіксованої пам'яті, що знаходиться в призначеному для користувача режимі віртуального адресного простору |

Результат виконання програми показано на рис. 3.1-3.2.

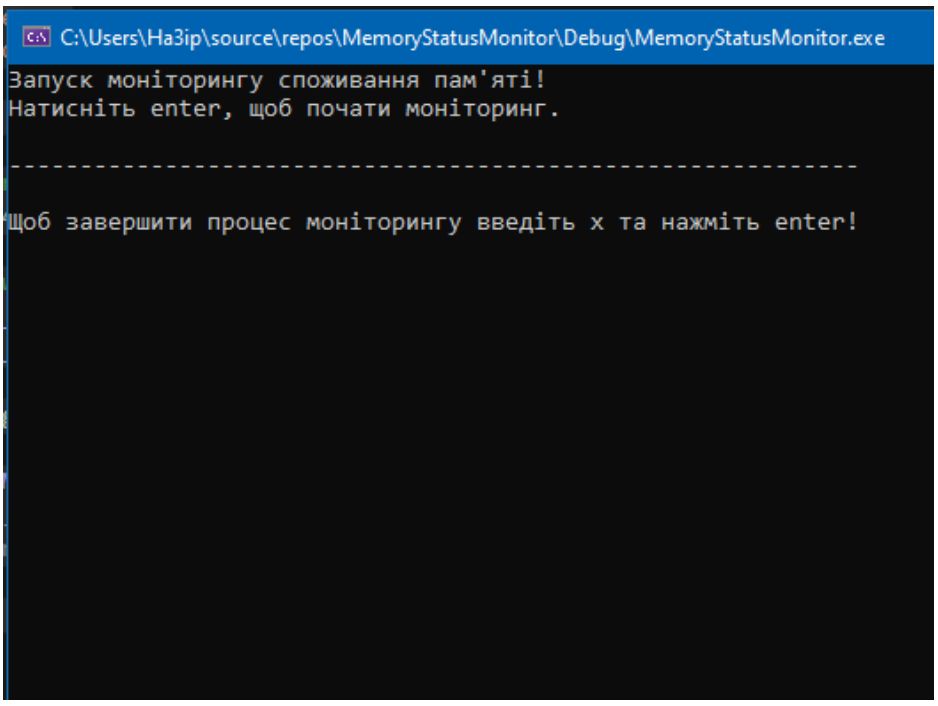



Рисунок 3.1 - Початок роботи програми

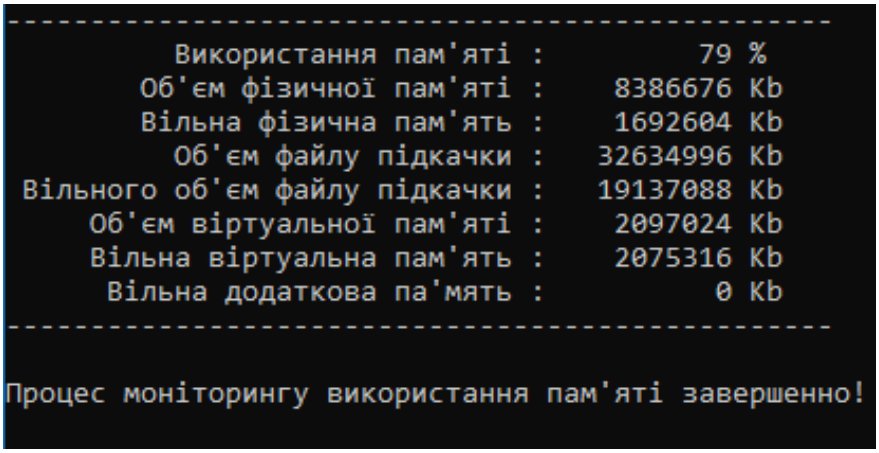



Рисунок 3.2 - Результат моніторингу пам'яті

## 3.5 Створення додатку для моніторингу

Для зручного використання буде створенно додаток, розроблений у Windows Forms, лістинг модулів наведено в додатку А, функції взаємодії наведені в таблиці 3.7.

1) створення форми для взаємодії з інтерфейсом (рис. 3.3);

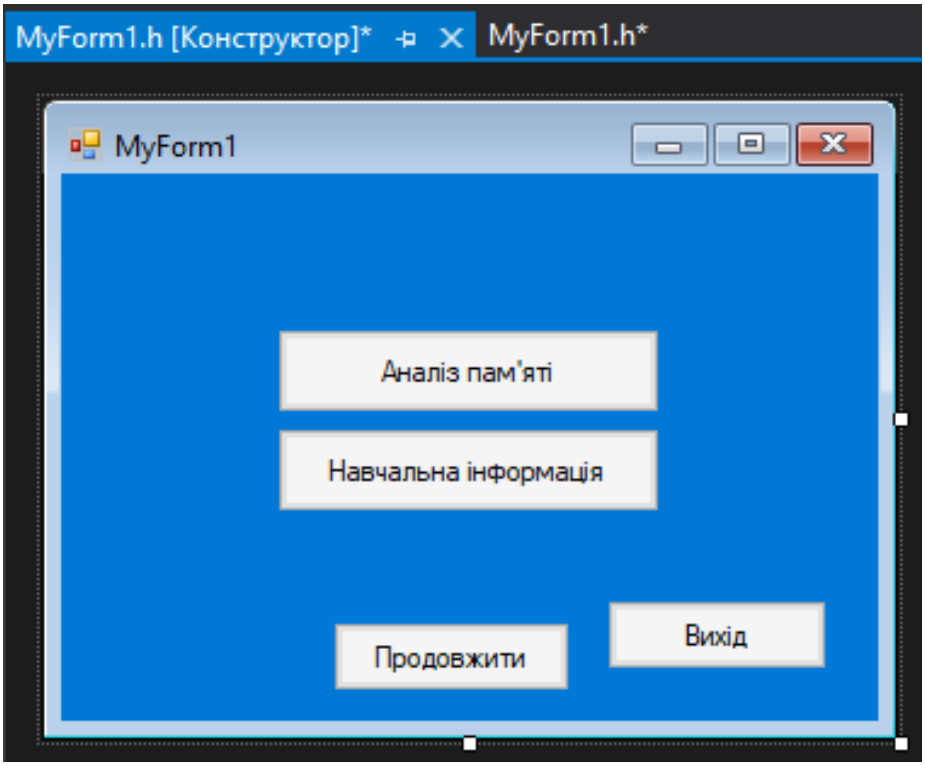



Рисунок 3.3 - Форма вибору взаємодії з інтерфейсом

Таблиця 3.7 - Функції інтерфейсного модуля

| № п.п | Назва функції | Опис функції |
|---|---|---|
| 1 | button1_Click | Функція здійснює виклик форми з навчальною інформацією |
| 2 | button2_Click | При нажатті на елемент «Аналіз пам’яті» відкривається форма моніторингу пам’яті при різних навантаженнях |
| 3 | richTextBox1->LoadFile("vstup.txt", RichTextBoxStreamType::PlainText); | При завантаженні програми відкривається інформаційна панель |
| 4 | button5_Click | Функція, яка закриває виконання програми |

2) створення модуля моніторингу пам’яті (рис. 3.4), функції наведені в таблиці 3.8;

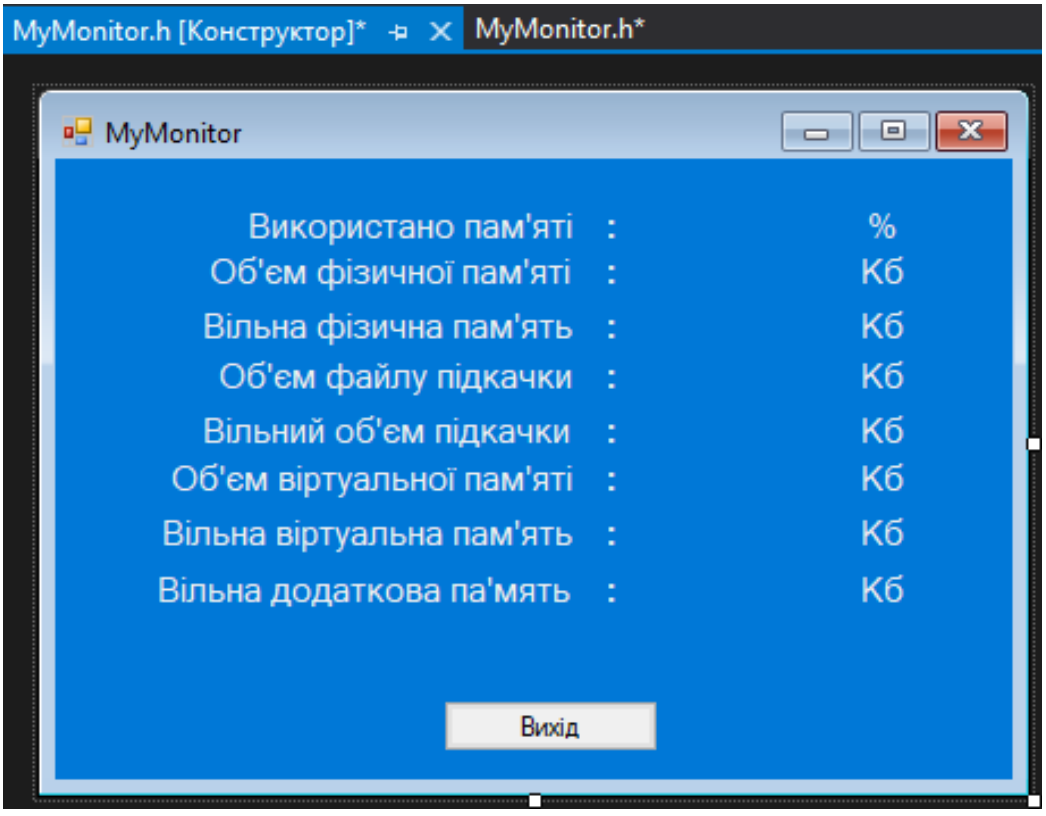



Рисунок 3.4 - Модуль моніторингу пам'яті, при різних навантаженнях

Таблиця 3.8 - Модуль моніторингу пам'яті

| № п.п | Назва функції | Опис функції |
|---|---|---|
| 1 | button1_Click | Функція здійснює виклик форми з навчальною інформацією |
| 2 | button2_Click | При нажатті на елемент відкривається повідомлення про завершення моніторингу пам’яті |
| 3 | GetLastError() | Функція, яка отримує значення останньої помилки при невдалому завантаженні модуля |
| 4 | MessageBox::Show() | Функція, виводить інформаційне повідомлення |
| 5 | GlobalMemoryStatusEx() | Функція, отримує інформацію про поточне використання системою фізичної і віртуальної пам'яті. |
| 6 | sizeof() | Оператор, видає обсяг пам'яті в байтах, необхідний для зберігання об'єкта типу операнда. |
| 7 | Close() | Функція закриття форми |
| 8 | Convert.ToString() | Перетворює вказане значення в його еквівалентне рядкове представлення |

3.6 Робота модуля моніторингу пам’яті, при різних навантаженнях

На рисунку 3.5-3.6 показана робота модуля моніторингу пам’яті при середньому навантаженні, на компютері в момент сканування було запущенно 3 програми. Використання пам’яті було здійснене на 52 відсотка, доступної вільної пам’яті 3942 мегабайта, всього на ОС виділенно 8 гб оперативної пам’яті, доступний об’єм файлу підкачки 32634 мб, використаного простору файлу підкачки 23368 мб. Об’єм віртуальної пам’яті 2 гб, вільної 1818 мб. Додаткової пам’яті немає.

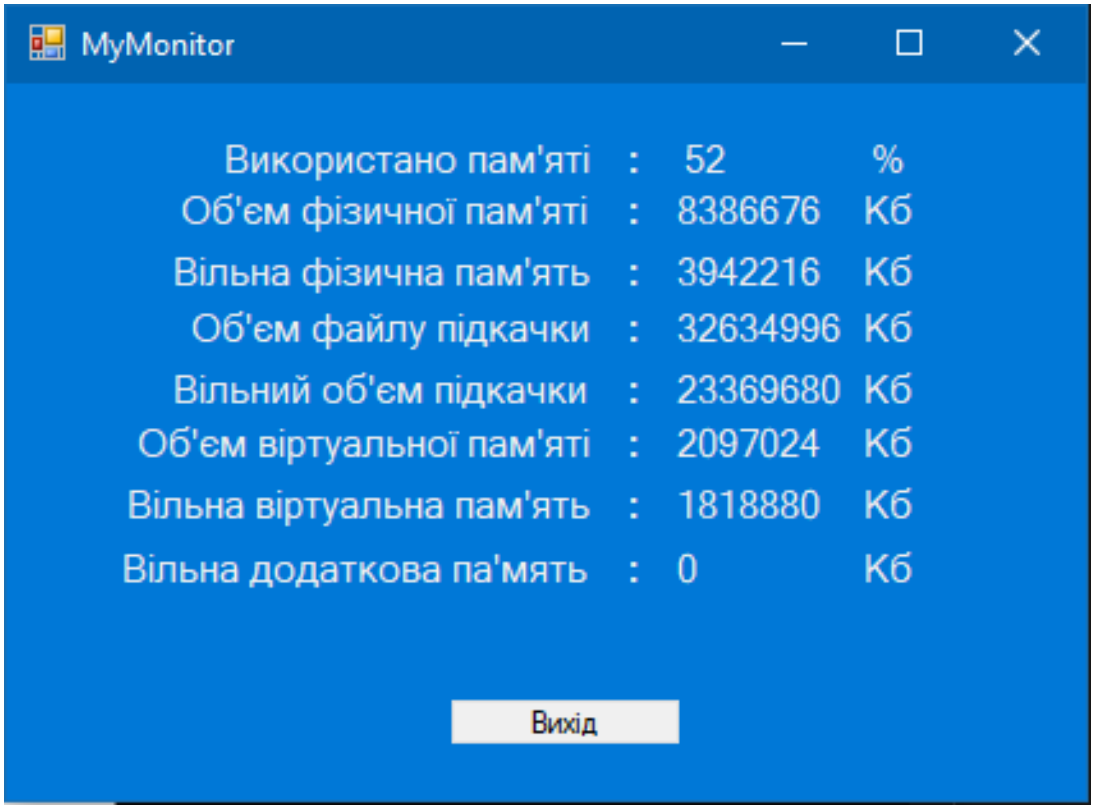



Рисунок 3.5 - Робота модуля

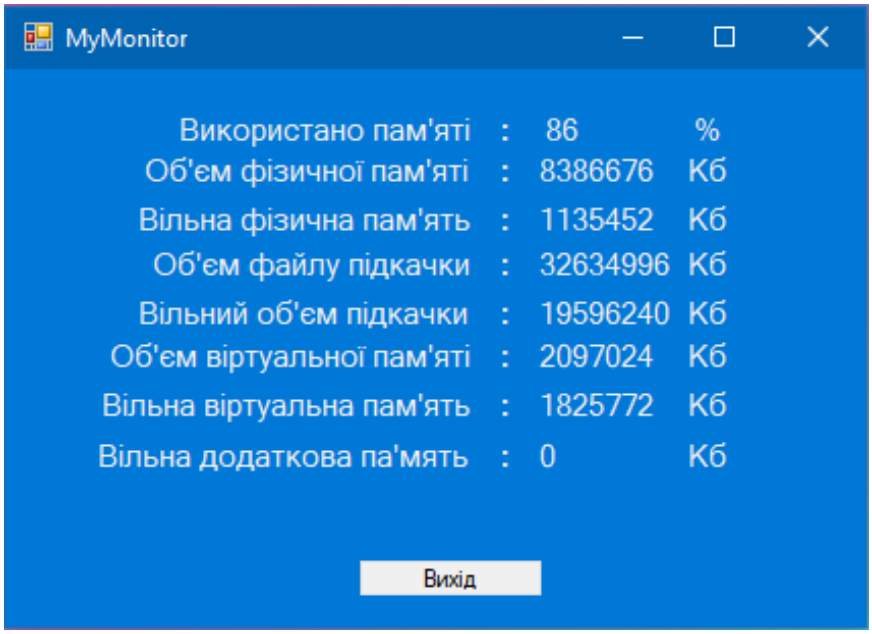



Рисунок 3.6 - Завантаженність модуля

Пам’ять вдалось завантажити на 86 відсотків, було запущенно 6 процесів, які використовували пам’ять, вільної фізичної пам’яті 1135 мб, віртуальна пам’ять майже не використовувалась вільно 1825 мб з 2 гб , всього використано файлу підкачки 13 гб. Цей результат задовольняє потреби ОС.

## 3.7 Висновки

Створенний модуль має лаконічний, простий у використанні дизайн, є можливість подальшого дослідження, та подальшого удосконалення, дозволяє користувачу дізнатись інформацію про загруженість пам’яті, при різноманітних навантаженнях на систему. У підрозділі 3.6 реалізовано порівняння додатку, за умови різних навантажень на ОС.

# ВИСНОВКИ

В ході роботи було проаналізовано існуючі методи аналізу віртуальної пам’яті, комерційні програми моніторингу, аналізу, та використання віртуального адресного простору.

Під час виконання кваліфікаційної роботи було здійснено теоретичний аналіз, вивчено недоліки існуючих рішень та сформовано задачу для дослідження шляхів їх вирішення.

Віртуальна пам'ять ефективно працює в системах з багатозадачною роботою, в яких пам'ять містить частини багатьох програм одночасно, в очікуванні читання своєї частини, процесор може бути зайнятий іншим процесом.

На першому етапі роботи відбулось ознайомлення з засобами аналізу віртуальної пам’яті. При цьому було охарактеризовано структуру предметної області та базову модель організації операційного пристрою.

Після опису існуючих рішень було створено модуль на мові програмування C++, для відображення стану пам’яті при різних навантаженнях на систему. Для створення модуля було використано середовище Visual Studio 2019. В результаті виконання дисппломного проекту було розроблено рекомендації по управлінню пам’яттю.

## ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ


1. Пахмурин Д. О. Операционные системы ЭВМ. Учебное пособие. Томск: Томский государственный университет систем управления и радиоэлектроники, 2013.
2. Карачка А. Ф., Дудко О. І. Архітектура комп'ютерів: Навч. посіб. За ред. А. О. Саченка. Тернопіль: Економічна думка, 2010. 176 с.
3. Х. М. Дейтел, П. Дж. Дейтел, Д. Р. Чофнес. Операционные системы. Часть1. Основы и принципы. М. Бином-Пресс. 2011. 1024с.
4. Операционные системы. Часть 2. Распределенные системы, сети, безопасность Х. М. Дейтел, П. Дж. Дейтел, Д. Р. Чофнес, М. Бином Пресс. 2011. 704 с.
5. Орлов С. А., Цилькер Б. Я., Организация ЭВМ и систем. СПб.: Питер, 2004. 667 с.
6. Клименко М. І., Панасенко Є. В., Стреляєв Ю. М., Ткаченко І. Г. Варіаційне числення та методи оптимізації : навч. посіб. Запоріжжя : ЗНУ, 2015. 84 с.
7. ДСТУ 3582:2013. Бібліографічний опис. Скорочення слів і словосполучень українською мовою. Загальні вимоги та правила(ISO 4:1984, NEQ; ISO 832:1994, NEQ). Вид. офіц. Київ : Мінекономрозвитку України, 2014. 15 с. (Інформація та документація).
8. Reiss, C., Tumanov, A., Ganger, G.R., Katz, R.H., and Kozuch, M.A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. Proceedings of the 3rd ACM Symposium on Cloud Computing, ACM (2012).
9. Про затвердження Вимог до оформлення дисертації: наказ Міністерства освіти і науки від 12.01.2017 р. № 40. Офіційний вісник України. 2017. № 20. С. 136–141.
10. Коноваленко І. В., Федорів П. С. Системне програмування у Windows з прикладами на Delphi, 2012.

11. Чистяков В. Д. Анатомия ПК. Все о компьютерном железе. – М.: НТ Пресс, 2011. 160 с.

12. Ahn, J., Jin, S., and Huh, J. Revisiting Hardware-Assisted Page Walks for Virtualized Systems. Proceedings of the 39th Annual International Symposium on Computer Architecture, 2012.

13. Операционные системы, С. В. Синицын, А. В. Батаев, Н. Ю. Налютин, 2010. 304с.

14. Современные операционные системы, Э. Таненбаум, СПб, Питер, 2010. 1120 с.

15. Иртегов Д. В. Введение в операционные системы. Д. В. Иртегов. 2-е изд. БХВ-Петербург, 2012. 1040 с.

16. Kusswurm Daniel. Modern X86 Assembly Language Programming. Daniel Kusswurm. - Apress, 2019. 604 p.

17. William Stallings. Operating Systems: Internals and Design Principles., 2018.

18. Stallings, William. Operating systems: internals and design principles William Stallings. 7 th ed. Prentice Hall, New Jersey, 2012, p.769.

19. Куклин В. М. ред. Введение в методы программных решений: учебное пособие: ХНУ им. В.Н. Каразина, 2011.

20. Текстові документи. Загальні вимоги. Красильникова, Л. Першина, Т. Касянчук., Хмельницький : ХНУ, 2017. 45 с.

21. Микитишин А.Г., Чихіра І.В. Операційні системи: конспект лекцій Тернопіль: ТНТУ, 2016. 104 с.

22. Tulloch M. Optimizing and Troubleshooting Hyper-V Networking Microsoft Press, 2013.

23. Л. Аврамов. Центры обработки данных на основе политик и ACI: структура, концепции и методология, 2016. 384 с.

24. Christos Kozyrakis, A.K. and Vaid, K. Server Engineering Insights for Large-Scale Online Services. IEEE Micro, 2010.

25. Авраменко В. С., Салапатов В. І. Вступ до програмної інженерії. Том 1. Історія розвитку. Основні поняття. Навчальний посібник. Черкаси: Черкаський національний університет ім. Б Хмельницького, 2015. 500 с.

26. Lustig, D., Bhattacharje, A., and Martonosi, M. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. ACM Transactions on Architecture and Code Optimization, 2013.

27. Назаров С.В., Широков А.И. Современные операционные системы: учебное пособие. М.: Национальный Открытый Университет «ИНТУИТ», 2012. 367 с.

28. Srikantaiah, S. and Kandemir, M. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. Proceedings of 43rd Annual IEEE/ACM International Symposium on Microarchitecture, (2010).

29. Вычислительные системы, сети и телекоммуникаци: учебное пособие, А.П. Пятибратов, Л.П. Гудыно, А.А. Кириченко., 2013. 376 с.

30. Таненбаум Э., Бос Х. Современные операционные системы. 4-е изд. СПб.: Питер, 2015. 1120 с.

31. Volos, H., Tack, A.J., and Swift, M.M. Mnemosyne: Lightweight Persistent Memory. Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, (2011).

32. Логічні елементи то основні операційні вузли: Лабораторний практикум. В. В. Булатецький, Л. В. Булатецька. Луцьк. Вежа, 2007. 68 с.

33. Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach TLBs. In MICRO, pages 258–269. IEEE Computer Society, 2012.

34. Устройство и функционирование ОС Windows. Национальный Открытый Университет «ИНТУИТ», 2016. 240 с.

35. Востокин С. В. Операционные системы: Учебник. Самара: Изд-во Самар, гос. аэрокосм, ун-та, 2012. 120 с.

36. Фридланд А. Я., Ханамирова Л. С., Фридланд И. А. Информатика и компьютерные технологии: Основные термины: Толковый словарь. 3-е изд., испр. и доп. М.: ООО «Издательство Астрель»: ООО «Издательство АСТ», 2013.

37. Рамбо Джеймс, Якобсон Айвар, Буч Грэди. Специальный справочник администратора. Питер, 2012. 656с.

38. Фролов А.В., Фролов Г.В. Аппаратное обеспечение "ДИАЛОГ-МИФИ", 2012. 208 с.

39. Lustig, D., Bhattacharje, A., and Martonosi, M. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. ACM Transactions on Architecture and Code Optimization, 2013.

40. Рассел С., Норвиг П. Искусственный интеллект: современный подход. 2-е изд. М. Вильямс, 2006. 1408 с.

UNICHECK

Ім'я користувача:
Кафедра КІ

ID перевірки:
1008347370

Дата перевірки:
22.06.2021 17:17:05 EEST

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
22.06.2021 17:17:23 EEST

ID користувача:
100005591

Назва документа: Вовк_Засоби аналізу віртуальної пам'яті

Кількість сторінок: 70 Кількість слів: 13821 Кількість символів: 108941 Розмір файлу: 2.30 MB ID файлу: 1008417485

Виявлено модифікації тексту (можуть впливати на відсоток схожості)

## 15%
## Схожість

Найбільша схожість: 6.84% з джерелом з Бібліотеки (ID файлу: 1008384073)

11.2% Джерела з Інтернету | 380 ..... Сторінка 72

3.57% Джерела з Бібліотеки | 128 ..... Сторінка 74

## 0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

## 0%
## Вилучень

Немає вилучених джерел

## Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи | 12

Підозріле форматування | 12 сторінок

Thu Jun 24 09:28:16 EEST 2021, Медзатий Дмитро Миколайович, Хмельницький національний університет, ХНУ

# Anti-Plagiarism v-15.257

**Максимальное совпадение с одним документом 0.0%**

Словари проверки: en_US, ru_RU, ua_UA. **Ошибок в документах: 11%**

| | Документ | | Суммарное совпадение по Базе Данных | |
|---|---|---|---|---|
| ID: 95304<br>Название: Засоби аналізу віртуальної пам’яті<br>Добавлено в БД: 2021-06-24<br>Авторы: Н.О. Вовк<br>Руководители: Т.М. Кисіль<br>Консультанты:<br>Опоненты: | Символы | Лексемы | Символы | Лексемы |
| | 82424 | 730 | 1029 (1%) | 19 (3%) |

Источник плагиата

| ID | Описание | Наличие плагиата в документе | |
|---|---|---|---|
| | | Символы | Лексемы |

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

# РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дипломник Вовк Назарій Олександрович

Тема Засоби аналізу віртуальної пам'яті

Спеціальність 123 Комп'ютерна інженерія

**Обсяг дипломної роботи:**

Кількість листів креслень _____; кількість сторінок записки 66

1.Короткий зміст ДР та прийнятих рішень В рамках кваліфікаційної роботи було розроблено модуль аналізу пам'яті, що дозволяє здійснювати аналіз фізичної та віртуальної пам'яті, також показує вільне місце на диску, та процент зайнятої пам'яті на операційній системі Windows.

2. Висновок про відповідність ДР поставленому завданню: Кваліфікаційна робота у повній мірі відповідає поставленому завданню як в теоретичній так і в практичній частині

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У першому розділі наведені переваги і недоліки різних методів поділу пам'яті та технології перетворення фізичних адрес у віртуальні. У другому розділі описуються і аналізуються деякі з сучасних інструментів, доступних аналітикам, зацікавленим у проведенні аналізу пам'яті, також зроблена спроба зібрати наявну інформацію, а в деяких випадках протестувати інструмент. У частині проектної реалізації було створено модуль на мові програмування C++, для відображення стану пам'яті при різних навантаженнях на систему.

4. Позитивні сторони роботи: Кваліфікаційна робота відповідає сучасним вимогам

5. Негативні сторони роботи Розроблений в роботі модуль має досить вузький функціонал, але це зумовлено специфікою роботи

6. Оцінка графічного оформлення та пояснювальної записки роботи: Графічне оформлення виконане на належному рівні

7. Відгук про роботу в цілому: В загальному кваліфікаційна робота відповідає вимогам. Весь матеріал кваліфікаційної роботи структурований, чіткий та послідовний. Усі розділи йдуть і вірній послідовності, що дозволяє чітко розуміти викладений матеріал в рамках даної роботи. Графічний матеріал дозволяє наочно побачити доцільність та ефективність рішень, прийнятих в даній роботі.

8. Інші зауваження немає

9. Оцінка дипломної роботи. Розглянувши позитивні та негативні сторони даної роботи, можна зробити висновок, що вона заслуговує оцінку «Добре»

РЕЦЕНЗЕНТ (прізвище, ім'я, по-батькові, посада, місце роботи) Мартинюк Валерій Володимирович, завідувач кафедри АКІТ і ТК, доктор технічних наук, професор

"22" 06 2021 р.

(підпис )

Завідувачу кафедри КІСП
д-ру техн.наук, проф. Говорущенко Т. О.

Вовка Н.О.

ПІБ здобувача вищої освіти

ФПКТС, 4 курсу, групи КІ-17-1

ЗАЯВА

З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 26.09.2020 (зі змінами від 26.11.2020), згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений (а). Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність плагіату ознайомлений(а) та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

22.01.2021

дата

підпис

# РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ

## КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА СИСТЕМНОГО ПРОГРАМУВАННЯ

## ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: Засоби аналізу віртуальної пам'яті

Автор: Вовк Назарій Олександрович

Спеціальність: 123 – Компютерна інженерія

Освітня програма: освітньо-професійна

Науковий керівник: Кисіль Тетяна Миколаївна, к.ф.-м.н, доцент

Після аналізу звіту подібності зроблено такий висновок:

| № | Висновок | Позначка про відповідність |
|---|---|---|
| 1 | Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту. | відповідає |
| 2 | Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи | |
| 3 | Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат. | |
| 4 | Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту. | |

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) запозичення розміщені в розділах аналізу існуючих аналогів та прототипів, які не описують безпосередньо авторське дослідження і не стосуються результатів роботи;
2) усі запозичення фрагментарні, або мають належним чином оформленні посилання;
3) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з 10-40 джерелами на один фрагмент речення;

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості, складає 15% і адресується до 380 першоджерела, що, з урахуванням наведених обгрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

Керівник роботи _____ Т.М. Кисіль

Гарант ОП _____ С. М. Лисенко

Завідувач кафедри КІСП _____ Т. О. Говорущенко