Центральноукраїнський національний технічний університет
Механіко-технологічний факультет
Кафедра кібербезпеки та програмного забезпечення

”Допущено до захисту”
Завідувач кафедри кібербезпеки
та програмного забезпечення
д.т.н., професор
_____ Олексій СМІРНОВ
“____” _____ 2023 р.

ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА
за першим (бакалаврським) рівнем вищої освіти
на тему
“Програмне забезпечення системи емуляції розподілу динамічної пам’яті”

Виконав здобувач вищої освіти
IV курсу, групи КІ-20-3СК
ОПП «Комп’ютерна інженерія»
спеціальності 123 «Комп’ютерна інженерія»
_____ Дудзінський А.М.
«____» _____ 2023 р.

Керівник проекту
кандидат технічних наук
_____ Буравченко К.О.
«____» _____ 2023 р.
Рецензент _____
_____

м. Кропивницький

Центральноукраїнський національний технічний університет

Факультет *Механіко-технологічний*

Кафедра *Кібербезпеки та програмного забезпечення*

Освітній ступінь *бакалавр*

Галузь знань ._12 _____*"*Інформаційні технології*"*_____

Спеціальність *123 "*Комп'ютерна інженерія*"*

Освітньо-професійна (освітньо-наукова) програма _*"*Комп'ютерна інженерія*"*_

ЗАТВЕРДЖУЮ
Завідувач кафедри
д.т.н., проф.
_____ Олексій СМІРНОВ
« 17 » січня 2023 року

ЗАВДАННЯ НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ РОБОТУ ЗА ПЕРШИМ (БАКАЛАВРСЬКИМ) РІВНЕМ ВИЩОЇ ОСВІТИ ЗДОБУВАЧА ВИЩОЇ ОСВІТИ

Дудзінському Артему Миколайовичу
(прізвище, ім'я, по батькові)

1. Тема роботи *Програмне забезпечення системи емуляції розподілу динамічної пам'яті*

2. Керівник роботи *Буравченко Костянтин Олегович, канд. техн. наук*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу № 8-02 від 5.01.2023 року

3. Строк подання студентом роботи до захисту *23.05.2023 р.*

4. Мета та завдання випускної кваліфікаційної роботи*: Метою роботи є розробка програмного забезпечення системи емуляції розподілу динамічної пам'яті*

5. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Призначення та область використання.
2.Перегляд аналогічних існуючих систем.
3. Опис і обґрунтування проектних рішень.
4. Етапи програмування системи.
5. Впровадження системи в промислову експлуатацію.
6. Висновки

6. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Структурна схема системи	*1 аркуш*
Функціональна схема системи	*1 аркуш*
Діаграма процесів	*1 аркуш*
Блок-схема алгоритму роботи додатку	*2 аркуша*

7. Дата видачі завдання « 17 » січня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти	Строк виконання етапів випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти	Примітка
1.	Аналіз існуючих систем	10.03.2023 р.	
2.	Постановка задачі, оформлення ТЗ	15.03.2023 р.	
3.	Розробка моделі компонента	20.03.2023 р.	
4.	Розробка структур даних	25.03.2023 р.	
5.	Розробка алгоритмів зв’язку та відображення	30.03.2023 р.	
6.	Програмування алгоритмів	10.04.2023 р.	
7.	Оформлення ПЗ	17.04.2023 р.	
8.	Попередній захист роботи	23.05.2023 р.	

Дата видачі завдання
« 17 » січня 2023 р.

Підпис керівника
_____ Буравченко К.О.
(прізвище та ініціали)

Завдання прийнято до виконання
« 17 » січня 2023 р.

Підпис здобувача
_____ Дудзінський А.М.
(прізвище та ініціали)

АНОТАЦІЯ

Дудзінський А.М. Програмне забезпечення системи емуляції розподілу динамічної пам’яті. 123 Комп’ютерна інженерія. Центральноукраїнський національний технічний університет. Кропивницький. 2023.

В даній випускній кваліфікаційній роботі за першим (бакалаврським) рівнем вищої освіти розроблено програмне забезпечення, яке призначено для системи емуляції розподілу динамічної пам’яті.

Метою розробки є програмне забезпечення системи емуляції розподілу динамічної пам’яті.

Результат роботи – програмна реалізація системи емуляції розподілу динамічної пам’яті.

В процесі роботи над програмною моделлю виконано аналіз існуючих апаратних та програмних засобів. В повній мірі описані всі компоненти розробленого програмного забезпечення.

Розроблено зручний інтерфейс користувача. Наведені інструкції по роботі з програмними засобами.

Програма може використовуватися на ПЕОМ архітектури IBM PC з ОС Windows 10/11.

Програму розроблено в середовищі Delphi 10.4 Sydney.

Ключові слова: комп’ютерна інженерія, емуляція, розподіл динамічної пам’яті

ABSTRACT

Dudzinskyi A.M. Dynamic memory allocation emulation system software. 123 Computer engineering. Central Ukrainian National Technical University. Kropyvnytskyi. 2023.

In this final qualification work for the first (bachelor) level of higher education, software is developed, which is intended for the emulation system of dynamic memory allocation.

The purpose of the development is the software of the emulation system of dynamic memory allocation.

The result of the work is a software implementation of the dynamic memory allocation emulation system.

In the process of working on the software model, an analysis of existing hardware and software was performed. All components of the developed software are fully described.

A convenient user interface has been developed. Instructions for working with software tools are provided.

The program can be used on PCs of IBM PC architecture with Windows 10/11 OS.

The program was developed in the Delphi 10.4 Sydney environment.

Keywords: computer engineering, emulation, dynamic memory allocation

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ І ТЕРМІНІВ

ЕОМ	–	Електронно-обчислювальна машина
ЗП	–	Запам'ятовуючий пристрій
ОЗП	–	Оперативний запам'ятовуючий пристрій
ОС	–	Операційна система
ОП	–	Оперативна пам'ять
ПЗ	–	Програмне забезпечення
ПК	–	Персональний комп'ютер
СУБД	–	Система управління базами даних
ЦП	–	Центральний процесор
best fit	–	Найбільш підходящий блок пам'яті
DDR SDRAM	–	Double Data Rate Synchronous Dynamic Random Access Memory
DIMM	–	Dual In line Memory Module
first fit	–	Перший підходящий блок пам'яті
SIMM	–	Single In line Memory Module
SD RAM	–	Synchronous Dynamic RAM
SJ	–	Single-Job, однозадачна система
VCL	–	Бібліотека візуальних компонентів
VMS	–	Virtual Memory System
worst fit	–	Найменш підходящий блок пам'яті

ВСТУП

Актуальність теми. Сучасний світ повністю залежить від наявності справних ІТ-ресурсів. Падіння сервера, або поломка ЕОМ призводить до колосальних фінансових втрат.

Природно, що для того, щоб працювати з ЕОМ, необхідна наявність працівників, які вільно володіють комп'ютером. У самому простому випадку це підготовлені користувачі, які мають уяву про будову ЕОМ, та володіють навичками роботи з тими або іншими програмними продуктами. Але у кожній фірмі або корпорації є людина яка відповідає за праце спроможність наявного парку комп'ютерів. У маленьких фірмах це, як правило інженер програміст, у більш великих фірмах та корпораціях існують цілі відділи, які призначені для підтримки ІТ-структури підприємства.

В зв'язку з усім вищеперерахованим, дуже гостро стоїть питання підготовки висококваліфікаційних кадрів у області ІТ-технологій. У зв'язку з тим, що основа підготовки таких кадрів спирається на знання архітектури ЕОМ, розробка навчальних програмних продуктів які дозволяють вивчити, як функціонування ЕОМ у цілому, так і окремих її частин, є актуальною задачею, яка потребує вирішення у даній бакалаврській роботі.

Мета й завдання дослідження. Метою роботи є програмне забезпечення системи емуляції розподілу динамічної пам'яті.

Для досягнення поставленої мети визначена програма дослідження, що складається з наступних завдань:

– Огляд існуючих систем емуляції розподілу динамічної пам'яті.
– Дослідження системи емуляції розподілу динамічної пам'яті.
– Програмна реалізація системи емуляції розподілу динамічної пам'яті.

Практична цінність отриманих результатів полягає в тому, що розроблені алгоритми дозволяють успішно вирішувати задачі емуляції розподілу динамічної пам’яті.

Таким чином, виходячи з вищеперерахованого, програмне забезпечення системи емуляції розподілу динамічної пам’яті, є актуальною задачею, яка потребує вирішення у даній випускній кваліфікаційній роботі за першим (бакалаврським) рівнем вищої освіти.

1 ПРИЗНАЧЕННЯ ТА ОБЛАСТЬ ВИКОРИСТАННЯ

1.1 Призначення системи

Оперативна пам'ять комп'ютера (RAM) є одним із найважливіших компонентів, що визначають продуктивність вашої системи. Оперативна пам’ять надає програмам місце для короткострокового зберігання та доступу до даних. Він зберігає інформацію, яку активно використовує ваш комп’ютер, щоб отримати до неї швидкий доступ.

Чим більше програм запущено у вашій системі, тим більше вам знадобиться. SSD_(твердотільні диски) також є важливими компонентами та допоможуть вашій системі досягти максимальної продуктивності.

Швидкість і продуктивність вашої системи безпосередньо залежать від обсягу встановленої оперативної пам’яті. Якщо ваша система має занадто мало оперативної пам’яті, вона може працювати повільно та мляво. Але на протилежному кінці ви можете встановити занадто багато з незначними додатковими перевагами. Є способи перевірити, чи ваш комп'ютер потребує більше пам'яті, а також переконатися, що ви купуєте пам’ять, сумісну з іншими компонентами вашої системи. Як правило, компоненти створюються відповідно до найвищих стандартів на момент виробництва, але з розрахунком на те, що технології продовжуватимуть змінюватися.

Щоб користувачі не вставляли несумісну пам’ять, модулі фізично розміщуються різні для кожної технології пам'яті покоління. Ці фізичні відмінності є стандартними для індустрії пам’яті. Однією з причин галузевої стандартизації пам’яті є те, що виробникам комп’ютерів необхідно знати електричні параметри та фізичну форму пам’яті, яку можна встановити в їхні комп’ютери.

Що таке швидкість і затримка оперативної пам'яті?

Продуктивність оперативної пам’яті залежить від співвідношення між швидкістю та затримкою. Хоча вони тісно пов’язані, вони не пов’язані так, як ви думаєте. На базовому рівні затримка означає часову затримку між моментом введення команди та моментом, коли дані стають доступними. Розуміння швидкість і затримка оперативної пам’яті допоможе вам краще вибрати правильну оперативну пам’ять для встановлення у вашій системі відповідно до ваших потреб.

Що робить RAM (пам'ять)?

Оперативна пам’ять дозволяє комп’ютеру виконувати багато повсякденних завдань, наприклад завантажувати програми, переглядати веб-сторінки в Інтернеті, редагувати електронну таблицю або переглядати останню гру. Пам’ять також дозволяє швидко перемикатися між цими завданнями, запам’ятовуючи, де ви перебуваєте в одному завданні, коли ви переходите до іншого. Як правило, чим більше у вас пам'яті, тим краще.

Коли ви вмикаєте комп’ютер і відкриваєте електронну таблицю, щоб редагувати її, але спочатку перевіряєте свою електронну пошту, ви використовуєте пам’ять різними способами. Пам’ять використовується для завантаження та запуску програм, таких як ваша програма для роботи з електронними таблицями, відповіді на команди, наприклад будь-які зміни, які ви внесли в електронну таблицю, або перемикання між кількома програмами, наприклад, коли ви вийшли з електронної таблиці, щоб перевірити електронну пошту. Пам'ять майже завжди активно використовується комп'ютером. Якщо ваша система повільна або не реагує, вам може знадобитися оновлення пам’яті. Якщо ви вважаєте, що вам може знадобитися більше пам’яті, ви легко оновите свій робочий стіл або ноутбук.

У певному сенсі пам'ять схожа на ваш стіл. Це дозволяє вам працювати над різними проектами, і чим більший ваш стіл, тим більше документів, папок і завдань ви можете мати одночасно. Ви можете швидко та легко отримати доступ до інформації, не відвідуючи картотеку (ваш накопичувач). Коли ви закінчите з проектом або залишитеся на день, ви можете помістити деякі або всі проекти в

картотеку для зберігання. Ваш накопичувач (жорсткий диск або твердотільний накопичувач) – це картотека, яка працює разом із вашим столом для відстеження ваших проектів.

Для чого використовується оперативна пам'ять?

Оперативна пам'ять використовується для зберігання інформації, яку потрібно швидко використовувати. Це означає, що відкриття багатьох програм, запуск різних процесів або доступ до кількох файлів одночасно, ймовірно, займе багато оперативної пам’яті. Особливо складні програми, такі як ігри чи програмне забезпечення для проектування, використовуватимуть найбільше оперативної пам’яті.

Вам потрібно оновити оперативну пам'ять?

Незалежно від того, чи є ви геймер, дизайнер, або просто хочете пришвидшити свій персональний комп’ютер, оновлення оперативної пам’яті – це простий і легкий спосіб підвищити продуктивність системи. Щоб визначити правильний тип пам’яті для вашого комп’ютера, скористайтеся Crucial® Advisor™ або Системний сканер. Ці інструменти допоможуть вам визначити, які модулі пам’яті сумісні з вашим комп’ютером, а також варіанти для ваших вимог до швидкості та бюджету.

1.2 Область застосування

Розроблене програмне забезпечення емуляції розподілу динамічної пам’яті дозволить студентам, що навчаються за спеціальністю системне програмування, краще зрозуміти методи розподілу динамічної пам'яті. Даний емулятор можна використовувати для виконання лабораторних робіт з програмування.

Таким чином, виходячи з вищеперерахованого, програмне забезпечення системи емуляції розподілу динамічної пам’яті, є актуальною задачею, яка потребує вирішення у даній випускній кваліфікаційній роботі за першим (бакалаврським) рівнем вищої освіти.

2 ПЕРЕГЛЯД АНАЛОГІЧНИХ ІСНУЮЧИХ СИСТЕМ

2.1 Огляд існуючих систем, технологій, архітектур, програмних рішень за профілем теми випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти

Основний ресурс системи, розподілом якого займається ОС – це оперативна пам'ять. Тому організація пам'яті дуже впливає на структуру й можливості ОС.

RAM (оперативна пам’ять) – це внутрішня пам’ять центрального процесора для зберігання даних, програм і результатів програми. Це пам’ять для читання/запису, яка зберігає дані, поки машина не запрацює. Як тільки апарат вимикається, дані стираються.

Час доступу до оперативної пам’яті не залежить від адреси, тобто кожне місце зберігання всередині пам’яті доступне так само легко, як і інші місця, і займає стільки ж часу. Доступ до даних в оперативній пам’яті можливий випадковим чином, але це дуже дорого.

Оперативна пам’ять є енергонезалежною, тобто дані, що зберігаються в ній, втрачаються, коли ми вимикаємо комп’ютер або в разі збою живлення. Тому резервна система безперебійного живлення (UPS) часто використовується з комп’ютерами. Оперативна пам’ять невелика як за фізичним розміром, так і за обсягом даних, які вона може вмістити.

Оперативна пам'ять буває двох типів:

– Статична оперативна пам'ять (SRAM).

– Динамічна оперативна пам'ять (DRAM).

Статична оперативна пам'ять (SRAM)

Слово **static** вказує на те, що пам'ять зберігає свій вміст, поки подається

живлення. Однак дані втрачаються, коли живлення припиняється через мінливу природу. Мікросхеми SRAM використовують матрицю з 6 транзисторів без конденсаторів. Транзистори не потребують живлення для запобігання витоку, тому SRAM не потрібно регулярно оновлювати.

У матриці є додатковий простір, тому SRAM використовує більше мікросхем, ніж DRAM для того самого обсягу пам’яті, що робить витрати на виробництво вищими. Таким чином, SRAM використовується як кеш-пам'ять і має дуже швидкий доступ.

Характеристика статичної оперативної пам'яті:

– Довге життя.
– Не потрібно оновлювати.
– Швидше.
– Використовується як кеш-пам'ять.
– Великий розмір.
– Дорого.
– Високе енергоспоживання.

Динамічна оперативна пам'ять (DRAM)

DRAM, на відміну від SRAM, необхідно постійно **оновлювати**, щоб підтримувати дані. Це робиться шляхом розміщення пам’яті на схемі оновлення, яка перезаписує дані кілька сотень разів на секунду. DRAM використовується для більшості системної пам’яті, оскільки вона дешева та мала. Усі DRAM складаються з комірок пам’яті, які складаються з одного конденсатора та одного транзистора.

Характеристики динамічної оперативної пам'яті:

– Короткий термін служби даних.
– Необхідно постійно оновлювати.
– Повільніше в порівнянні з SRAM.
– Використовується як оперативна пам'ять.
– Менший за розміром.

– Дешевше.

– Менше енергоспоживання.

Найпростіший випадок керування пам'яттю – ситуація, коли диспетчер пам'яті відсутній, і в системі може бути завантажена тільки одна програма. Саме в такому режимі працюють CP/M і RT-11 SJ (Single-Job, однозадачна).

У цих системах програми завантажуються з фіксованої адреси PROG_START. В CP/M це 0x100; в RT-11 – 01000. В адресах від 0 до початку програми перебувають вектори переривань, а в RT-11 – також і стек програми. Сама система розміщається в старших адресах пам'яті. Адреса SYS_START, з якої вона починається, залежить від кількості пам'яті в машині й від конфігурації самої ОС.

У цьому випадку керування пам'яттю з боку системи полягає в тому, що завантажник перевіряє чи поміститься модуль, що завантажується, у простір від PROG_START до SYS_START. Якщо обсяг пам'яті, що використовує програма, не буде мінятися під час її виконання, то на цьому все керування й закінчується.

Однак програма може використовувати динамічне керування пам'яттю, наприклад функцію malloc() або щось у цьому роді. У цьому випадку вже код malloc() повинен стежити за тим, щоб не залізти в системні адреси. Як правило, динамічна пам'ять починає розміщатися з адреси PROG_END = PROG_START + PROG_SIZE. PROG_SIZE у цьому випадку позначає повний розмір програми, тобто розмір її коду, статичних даних і області, виділеної під стек.

Функція malloc() підтримує деяку структуру даних, що стежить за тим, які блоки пам'яті із вже виділених були звільнені. При кожному новому запиті вона спочатку шукає блок підходящого розміру у своїй структурі даних і, тільки коли цей пошук завершиться невдачею, відкушує новий блок пам'яті в системи. Для цього використовується змінна, котра в бібліотеці мови С називається brklevel. Початково ця змінна дорівнює PROG_END, її значення збільшується при виділенні нових блоків, але в деяких випадках може й зменшуватися. Це відбувається, коли програма звільняє блок, що закінчується на поточному

значенні brklevel.

Основними проблемами багатозадачних систем без диспетчера пам'яті є:

– Проблема виділення додаткової пам'яті програмі, що завантажувалася не останньою.

– Проблема звільнення пам'яті після завершення програми. У системах з монолітним завантажувальним модулем, іноді просто забороняють програмам вивантажуватися. В MS DOS була зроблена спроба заборонити вивантажувати TSR і драйвери, але це привело до пошуку задніх дверей.

– Низька надійність. Помилка в одній із програм може привести до псування коду або даних інших програм або самої системи.

– Проблеми безпеки. У системах з відкритою пам'яттю неможливі ефективні засоби поділу доступу. Будь-яка програмна перевірка прав доступу може бути легко обійдена прямим викликом "захищаємих" модулів ядра. Навіть криптографічні засоби не забезпечують досить ефективного захисту, тому що можна посадити в пам'ять троянську програму, що буде аналізувати код програми шифрування й зчитувати значення ключа.

У системах з динамічною зборкою перші дві проблеми не такі гострі, тому що пам'ять виділяється й звільняється невеликими шматочками, по блоці на кожний об'єктний модуль, тому код програми звичайно не займає безперервного простору. Відповідно, такі системи часто дозволяють і даним програми займати несуміжні області пам'яті.

Такий підхід використовується багатьма системами з відкритою пам'яттю – AmigaDOS, Oberon й т.д. Однак у таких системах дуже гостру форму приймає проблема фрагментації вільної пам'яті.

Кількість фрагментів приблизно пропорційна кількості операцій виділення/звільнення пам'яті. Ймовірність не знайти блок підходящого розміру через фрагментацію оцінити складніше, але вона також росте з кількістю операцій.

Зрозуміло, що якщо ці операції здійснюються відразу багатьма

програмами, то фрагментація пропорційно зростає. Перераховані вище системи не надають ніяких засобів для боротьби із фрагментацією.

Навпроти, у системі MacOS був запропоновано досить оригінальний метод боротьби із фрагментацією.

Керування пам'яттю в MacOS і MS Windows

У цих системах передбачається, що користувальницькі програми не зберігають покажчиків на динамічно виділені блоки пам'яті. Замість цього кожний такий блок ідентифікується цілочисленим дескриптором або "ручкою" (**handle**). Коли програма безпосередньо звертається до даних у блоці, вона виконує системний виклик GlobalLock (замкнути). Цей виклик повертає поточну адресу блоку. Поки програма не виконає виклик GlobalUnlock (відімкнути), система не намагається змінити адресу блоку. Навпроти, якщо блок не замкнений, система вважає себе вправі пересувати його по пам'яті або навіть скидати на диск.

Таким чином, "ручки" являють собою спробу створити програмний аналог апаратних диспетчерів пам'яті. Вони дозволяють вирішити проблему фрагментації й навіть організувати якусь подобу віртуальної пам'яті. Можна розглядати їх як засіб організації оверлейних даних – почергового відображення різних блоків даних на ті самі адреси. Однак за це доводиться платити дуже дорогою ціною.

Використання "ручок" сильно ускладнює програмування взагалі й особливо перенесення ПЗ із систем, що використовують лінійний адресний простір. Всі покажчики на динамічні структури даних у програмі потрібно замінити на "ручки", а кожне звертання до таких структур необхідно оточити викликами GlobalLock/GlobalUnlock. Ці виклики:

– самі по собі збільшують обсяг коду й час виконання;

– заважають компіляторам виконувати оптимізацію, насамперед не дозволяють оптимально використовувати регістри процесора, тому що далеко не всі регістри зберігаються при викликах;

– вимагають розриву конвеєра команд і перезавантаження командного кеша; у сучасних суперскалярних процесорах це може приводити до падіння

продуктивності в багато разів.

Спроби зменшити число блокувань вимагають певних інтелектуальних зусиль. Фактично, до звичайного циклу розробки ПЗ: проектування, вибір алгоритму, написання коду і його налагодження – додається ще дві фази: мікрооптимізація використання "ручок" і налагодження оптимізованого коду. Остання фаза виявляється, мабуть, самою складною й відповідальною.

Найбільш неприємною й небезпечною помилкою, що виникає на фазі мікрооптимізації, є винос покажчика на динамічну структуру за межі дужок GlobalLock/GlobalUnlock. Цю помилку дуже складно виявити при тестуванні, тому що вона проявляється тільки якщо система намагалася пересувати блоки в проміжках між обігами. Іншими словами, помилка може проявляти або не проявляти себе залежно від набору додатків, що виконуються в системі й від характеру діяльності цих додатків. У результаті ми одержуємо те, чого найбільше бояться інженери, механіки й програмісти – систему, що працює *іноді*.

Не випадково фірма MicroSoft повністю відмовилася від "ручечного" керування пам'яттю в новій версії *MS Windows – Windows 95*, де реалізована повноцінна віртуальна пам'ять.

Системи з базовою віртуальною адресацією

Як уже говорилося, у системах з відкритою пам'яттю виникають більші складності при організації багатозадачної роботи. Щоб усунути їх, необхідно надавати кожному завданню свій віртуальний адресний простір. Найбільш простим способом організувати різні адресні простори є так звана **базова адресація**. Очевидно, це історично найбільш ранній спосіб.

Цей метод майже повністю у формуванні адреси складається по схемі reg[offset]. Відмінність полягає в тому, що регістр, щодо якого відбувається адресація, не доступний прикладній програмі. Крім того, його значення додається до *всіх* адрес, у тому числі до "абсолютних" адресних посилань або змінних типу покажчик. По суті, така адресація є способом організації віртуального адресного простору. Дійсно, значення покажчика 0x000A25, буде відповідати фізичній

адресі BASE + 0x000A25 і мінятися разом зі значенням регістра BASE.

Як правило, машини, що використовують базову адресацію, мають два регістри. Один з регістрів задає базу для адрес, другий встановлює верхню межу. У системі *ICL1900* ці регістри називаються відповідно BASE і DATUM. Якщо адреса виходить за границю, встановлену значенням DATUM, виникає **виняткова ситуація** (**exception**) помилкової адресації. Як правило, це приводить до того, що система примусово завершує роботу програми.

За допомогою цих двох регістрів відразу вирішуються дві важливі проблеми.

По-перше, програми ізолюються одна від одної – помилки в одній програмі не приведуть до руйнування або ушкодження інших програм або самої системи. Завдяки цьому можна забезпечити захист системи не тільки від помилкових програм, але й від зловмисних дій користувачів по руйнуванню системи або доступу до чужих даних.

По-друге, з'являється можливість пересувати адресні простори задач по фізичній пам'яті так, що сама програма не помічає, що її пересунули. За рахунок цього вирішується проблема фрагментації пам'яті й програмам дається можливість нарощувати свій адресний простір. Дійсно, у системі з відкритою пам'яттю програма може додавати собі пам'ять тільки доти, поки не впреться в початок наступної програми. Після цього ми повинні або говорити, що пам'яті немає, або миритися з тим, що програма може займати несуміжні області фізичного адресного простору. Друге рішення різко ускладнює керування пам'яттю, як з боку системи, так і з боку програми, і часто виявляється неприйнятним. У випадку ж базової адресації можна просто зрушити програму, що заважає, вгору по фізичних адресах.

Часто системи, що працюють на таких архітектурах, уміють скидати на диск ті завдання, які довго не будуть виконуватися. Це найпростіша з форм **свопінга** (**swapping** – обмін).

У сучасних системах базова віртуальна адресація використовується рідко.

Справа не в тому, що вона погана, а в тому, що більш складні методи, такі як сегментна й сторінкова трансляція адрес, виявилися набагато краще. Часто під словами "віртуальна пам'ять" розуміють саме сегментну або сторінкову адресацію.

Сегментна й сторінкова віртуальна пам'ять

У системах із сегментною й сторінковою адресацією віртуальна адреса має складну структуру. Вона розбита на два бітові поля: номер сторінки (сегмента) і зсув у ньому. Відповідно, адресний простір складається з дискретних блоків. Якщо всі ці блоки мають фіксовану довжину й утворюють разом безперервний простір, вони називаються сторінками. Якщо довжина кожного блоку може задаватися довільно, а невикористовуваним частинам блоків відповідають "діри" у віртуальному адресному просторі, вони називаються сегментами. Як правило, один сегмент відповідає коду або даним одного модуля програми. Зі сторінкою або сегментом можуть бути асоційовані права читання, запису й виконання.

Така адресація реалізується апаратно. Процесор, як правило, має спеціальний пристрій, що називається **диспетчером пам'яті**. У деяких процесорах, наприклад в *MC68020* або *MC68030* або в деяких RISC-системах, цей пристрій реалізований на окремому кристалі; в інших, таких як *i386* або *Alpha*, диспетчер пам'яті інтегрований у процесор.

Диспетчер пам'яті містить регістр – покажчик на **таблицю трансляції**. Ця таблиця розміщається десь у фізичній пам'яті. Її елементами є **дескриптори** кожної сторінки/сегмента. Такий дескриптор містить права доступу до сторінки, ознаку присутності цієї сторінки в пам'яті й фізичну адресу сторінки/сегмента. Для сегментів у дескрипторі також зберігається довжина сегмента.

Як правило, диспетчер пам'яті має також **кеш** (**cache**) дескрипторів – швидку пам'ять із асоціативним доступом. У цій пам'яті зберігаються дескриптори часто використовуваних сторінок.

Алгоритм доступу до пам'яті по віртуальній адресі [page:offset]:

– Перевірити, чи існує сторінка page взагалі. Наприклад, у машин

сімейства VAX адресний простір може складатися з 8М сторінок, що відповідає 4 Гбайтам, але реальні програми використовують набагато менше пам'яті, і розміри їхніх таблиць трансляції відповідно зменшуються. Природно, про сторінку, для якої немає відповідного елемента таблиці, можна сказати, що її не існує. Така перевірка здійснюється простим порівнянням номера сторінки з довжиною таблиці трансляції. Якщо сторінки не існує, виникає **особлива ситуація помилки сегментації** (**segmentation violation**)

– Спробувати знайти дескриптор сторінки в кеші.

– Якщо його немає в кеші, завантажити дескриптор з таблиці в пам'яті.

– Перевірити, чи має процес відповідне право доступу до сторінки. Інакше також виникає помилка сегментації.

– Перевірити, чи перебуває сторінка в оперативній пам'яті. Якщо її там немає, виникає **особлива ситуація відсутності сторінки** або **сторінкова відмова** (**page fault**). Як правило, реакція на неї полягає в тому, що викликається спеціальна програма-оброблювач (**trap** – пастка), що підкачує необхідну сторінку з диска. У багатозадачних системах під час такого підкачування може виконуватися інший процес.

– Якщо сторінка є в пам'яті, взяти з її дескриптора фізичну адресу phys_addr.

– Якщо адресація – сегментна, зрівняти зсув у сегменті з довжиною цього сегмента. Якщо зсув виявився більше, також виникає помилка сегментації.

– Зробити доступ до пам'яті за адресою phys_addr[offset].

Видно, що така схема адресації досить складна. Однак у сучасних процесорах все це реалізовано в апаратурі й, завдяки кешу дескрипторів і інших хитрощів, швидкість доступу до пам'яті виходить майже така ж, як і при прямій адресації. Крім того, ця схема має неоціненні переваги при реалізації багатозадачних ОС.

По-перше, з кожною задачею можна зв'язати свою таблицю трансляції, а значить і свій віртуальний адресний простір. Завдяки цьому навіть у

багатозадачних ОС можна користуватися абсолютним завантажником. Крім того, програми виявляються ізольованими одна від одної, що забезпечує їхню безпеку.

По-друге, можна скидати на диск рідко використовувані області віртуальної пам'яті програм – не всю програму цілком, а тільки її частину. Крім того, на відміну від оверлеїв, програма взагалі не зобов'язана знати, яка її частина може бути скинута.

По-третє, програма не зобов'язана займати безперервну область фізичної пам'яті. При цьому вона цілком може бачити безперервний віртуальний адресний простір. Це різко спрощує боротьбу із фрагментацією пам'яті, а в системах зі сторінковою адресацією проблема фрагментації фізичної пам'яті взагалі знімається.

Так, в VAX/VMS вільна пам'ять відслідковується за допомогою бітової маски фізичних сторінок. У цій масці вільній сторінці відповідає 1, а зайнятий – 0. Якщо комусь потрібна сторінка, система просто шукає в цій масці встановлений біт. Якщо врахувати, що VAX має спеціальну команду для пошуку встановленого біта в масиві, все працює дуже швидко й просто.

У результаті віртуальний простір програми може виявитися відображеним на фізичні адреси дуже вигадливим чином, але це нікого не хвилює – швидкість доступу до всіх сторінок однакова.

По-четверте, система може забезпечувати не тільки захист програм одна від одної, але й захист програми від самої себе – наприклад, від помилкового запису даних на місце коду.

По-п'яте, різні задачі можуть використовувати спільні області пам'яті для взаємодії, або просто для того, щоб працювати з однією копією бібліотеки підпрограм.

Перераховані переваги настільки серйозні, що вважається неможливим реалізувати багатозадачну систему загального призначення, таку як *UNIX* або *VMS* (Virtual Memory System) на машинах без диспетчера пам'яті.

Окремою проблемою при розробці системи зі сторінковою або сегментною

адресацією є вибір розміру сторінки або максимального розміру сегмента. Цей розмір визначається шириною відповідного бітового поля адреси й тому повинен бути ступенем двійки. З одного боку, сторінки не повинні бути занадто великими, тому що це може привести до неефективного використання пам'яті і псування великих обсягів даних при скиданні сторінок на диск. З іншого боку, сторінки не повинні бути занадто малими, тому що це приведе до надмірного збільшення таблиць трансляції, необхідного обсягу кеша дескрипторів і т.д.

У реальних системах розмір сторінки міняється від 512 байт у машин сімейства *VAX* до декількох кілобайт. Наприклад, *i3/486* має сторінку розміром 4Кб. Деякі диспетчери пам'яті, наприклад в *MC6801/2/30*, мають змінний розмір сторінки – у тому розумінні, що система при запуску програмує диспетчер і встановлює, крім іншого, цей розмір, і далі працює зі сторінками обраного розміру. У процесора *i860* розмір сторінки перемикається між 4 Кб и 4 Мб.

Керування пам'яттю в Linux

Система керування пам'яттю в Linux здійснює підкачування сторінок по зверненню відповідно до COPY-ON-WRITE стратегії, заснованої на механізмі підкачування, що підтримується 386-тим процесором. Процес одержує свої таблиці сторінок від батьківського (при виконанні fork()) із входами, позначеними як READ-ONLY або як заміщювані. Потім, якщо процес намагається писати в цю область пам'яті, і сторінка є COPY-ON-WRITE сторінкою, вона копіюється й позначається як READ-WRITE. Інструкція exec() приводить до зчитування сторінки або те ж саме відбувається при виконанні програми. Надалі процесу важко одержати доступ до іншої сторінки. Кожний процес має директорію сторінок, що має на увазі можливість доступу до 1Кб таблиць сторінок, що вказують на 1Мб 4-х кілобайтних сторінок, які розміщаються в 4Гб пам'яті. Директорія сторінок процесу ініціалізується при виконанні розпаралелювання за допомогою copy_page_tables(). Директорія сторінок холостого процесу ініціалізується шляхом виконання ініціалізуючої послідовності.

Кожний користувальницький процес має локальну таблицю дескриптора,

що містить сегмент коду й сегмент дані-стек. Сегментам користувача приділяється пам'ять від 0 до 3Гб (0xC0000000). У користувацькому просторі лінійні адреси і логічні адреси ідентичні.

У процесорі 80386 значення лінійної адреси лежить у межах 0Гб – 4Гб. Лінійна адреса вказує на область пам'яті в цьому просторі. Лінійна адреса відмінна від фізичної адреси, він є віртуальним.

Логічна адреса складається із селектора й зсуву. Селектор вказує на сегмент, а зсув говорить про те, де розміщена область всередині сегмента.

Код ядра й сегменти даних є привілейованими сегментами, що визначаються в таблиці глобального дескриптора й розміщаються в області 3Гб – 4Гб. Встановлено директорію сторінок програми підкачування (swapper_pg_dir) і в такий спосіб логічні й фізичні адреси в просторі ядра є ідентичними.

Простір вище 3Гб з'являється в директорії сторінок процесу як покажчики на таблиці сторінок ядра. Ця область прозора для процесів у режимі користувача, однак, планування розподілу пам'яті стає доречним при привілейованому режимі, наприклад, при виконанні системного виклику.

Режим супервізора вводиться всередині контексту поточного процесу, так що трансляція адреси відбувається щодо каталогу сторінок процесу, але використовуючи сегменти ядра. Це ідентично тому, як відбувається керування пам'яттю з використанням swapper_pg_dir і сегменти ядра як і директорії сторінок використовують ті ж таблиці сторінок у цьому просторі.

Користувальницький стек розміщається на вершині сегмента даних користувача й збільшується вниз. Kernel_stack_frame (сторінка) зв'язується з кожним знову створюваним процесом і використовується щоразу, коли ядро виконує дії з контекстом процесом. Неприємності можуть відбутися, якщо стек ядра буде рости нижче його поточного кадру.

Процеси припиняють поділ їхніх сегментів даних і коду (хоча вони мають роздільні локальні таблиці дескрипторів, входи вказують на ті ж сегменти). Сторінки стека й даних будуть скопійовані, коли батько або спадкоємець буде

писати в них (COPY-ON-WRITE).

Програмні й апаратні переривання управляються усередині контексту поточної задачі. Більш детально, директорія сторінок поточного процесу використовується при трансляції адреси. Сегменти, однак, є сегментами ядра й у такий спосіб всі лінійні адреси вказують в область пам'яті ядра. Припустимо, наприклад, що користувальницький процес виконує системний виклик і ядро хоче одержати доступ до змінної за адресою 0x01. Лінійна адреса буде дорівнювати 0xC0000001 (використання сегментів ядра), а фізична адреса – 0x01. Буква тут присутня внаслідок того, що директорія сторінок процесу відображає цей діапазон точно як page_pg_dir.

Область ядра (0xC0000000 + high_memory) відображається через таблиці сторінок ядра, які є частиною RESURVED пам'яті. Тому вони розділяються всіма процесами. Під час розпаралелювання copy_page_tables() прямо звертається до таблиць RESERVED сторінок. Вона встановлює покажчики в директоріях сторінок процесу, щоб вказувати на таблиці сторінок ядра й не розміщає в пам'яті нових таблиць сторінок, як це звичайно робиться. Як приклад kernel_stack_page (який розміщений десь в області ядра) не має потреби у зв'язаній page_table, розміщеній в p_dir процесу, щоб відобразити її. Інструкція переривання встановлює покажчик стека й сегмент стека із привілейованих 0 значень, збережених в tss поточного завдання. Стек ядра в дійсності є фрагментованим об'єктом – це не єдиний об'єкт, а група стекових фреймів, кожний з яких розміщається в пам'яті при створенні процесу й звільняє пам'ять при його закінченні. Стек ядра ніколи не повинен рости всередині контексту процесу занадто швидко, щоб не розширитися за межі поточного фрейму.

2.2 Обґрунтування вибору засобів для побудови системи та мови програмування

Embarcadero Delphi, раніше Borland Delphi і Codegear Delphi, – інтегроване середовище розробки ПЗ для Microsoft Windows, Mac OS, iOS і Android мовою Delphi (що раніше носила назву Object Pascal), створена спочатку фірмою Borland і на даний момент приналежна й розроблювальна Embarcadero Technologies. Embarcadero Delphi є частиною пакета Embarcadero RAD Studio і поставляється в чотирьох редакціях: Community (поширюється безкоштовно й має обмежену ліцензію на використання в комерційних цілях), Professional, Enterprise і Architect.

Delphi 10.4 Sydney

Випущено 26 травня 2020 року. RAD Studio Delphi 10.4 забезпечує значно поліпшену високопродуктивну нативну підтримку Windows, кращу продуктивність розробки, миттєві підказки code completion, прискорення виконання коду із синтаксисом керованих записів, поліпшення виконання паралельних завдань на сучасних багатоядерних CPU, а також містить більш 1000 виправлень багів, поліпшення продуктивності середовища й бібліотек і багато чого крім того.

Основні можливості Delphi 10.4.1:

− Істотні розширення для Windows: поліпшення для застосунків на моніторах 4K High DPI, інтеграція з новим WebView2 на базі Chromium, використання розширених title bars, таких же, як в Office, Explorer, Google Chrome.

− Керування пам'яттю в Delphi тепер стандартизоване на всіх підтримуваних платформах – мобільних, настільних і серверних – використовучи класичну реалізацію керування пам'яттю об'єктів.

− Істотне поліпшення Delphi Code Insight (без можливого блокування IDE – в окремому процесі), що допоможе при роботі з великими проектами.

− Тип даних Delphi «record» тепер підтримуть довільні ініціалізацію, фіналізацію й операції копіювання.

− Розширена підтримка бібліотек C++: ZeroMQ, SDL2, SOCI, libSIMDpp і Nematode.

− Відладник Win 64 (на LLDB) і збирач для C++.

− Поліпшення для C++: Включена велика кількість поліпшень STL з Dinkumware.

− Підтримка Metal Driver GPU для macOS і iOS.

− Вбудований Fmxlinux.

− Компонент Twebbrowser для iOS тепер реалізований на Wkwebview API. Реалізація компонента Media Player для macOS тепер використовує Avfoundation. Реалізований заново стилізуємий FMX компонент TMemo на платформі Windows значно поліпшений і тепер має відмінну підтримку IME.

− Численні поліпшення швидкості й стабільності роботи нашої бібліотеки The Parallel Programming Library (PPL).

− Додані оновлені драйвери для FireBird, PostgreSQL і SQLite.

− Клієнтські бібліотеки HTTP і REST Client розширені застосунковими можливостями роботи з HTTPS. Також були розширені можливості підтримки Amazon AWS services

− У технологію Visual LiveBindings внесена безліч поліпшень, у тому числі швидкодії, що стосуться, застосунків на VCL і FireMonkey

RAD Studio 10.4 Короткий огляд:

– Істотні розширення для Windows. Створення застосунків, що чудово виглядають, із чіткими елементами інтерфейсу на 4k моніторах High DPI за допомогою нової гнучкої підтримки стилів елементів керування на екрані. Інтеграція із сучасними, безпечними web-технологіями від Microsoft – новим WebView2 на базі Chromium. Використання сучасних розширених title bars, таких же, як в Office, Explorer, Google Chrome, у своїх проектах. Істотні поліпшення надійності налагодження в новому відладнику для C++ Windows 64-bit.

– Зросла продуктивність розробки. Ріст продуктивності за рахунок миттєвої реакції підказок code completion у середовищі IDE. Краща сумісність із уже наявною кодовою базою, і спрощення програмування за рахунок уніфікованої архітектури керування пам'яттю. Швидке зв'язування даних і візуальних елементів за допомогою розширеної технології Visual LiveBindings з підвищеною швидкодією. Просте використання розповсюджених бібліотек C++, наприклад, ZeroMQ, SDL2, SOCI, libSIMDpp і Nematode. Оновлена підтримка Amazon AWS cloud.

– Поліпшення швидкодії і якості. Більш 1000 поліпшень швидкодії і якості. Краща ефективність коду за допомогою нового синтаксису custom managed records. Більш швидке виконання паралельних завдань на сучасних багатоядерних CPU. Переконаєтеся в прискоренні відображення на екрані з підтримкою Metal API на macOS і iOS. Краща сумісність із уже наявною кодовою базою й спрощення програмування за рахунок уніфікованої архітектури керування пам'яттю.

Істотне поліпшення Delphi Code Insight

Як найбільше й головне поліпшення інструментів програмування Delphi за багато років, в 10.4 Delphi Code Insight реалізований через Language Server Protocol (LSP). LSP – це технологія генерації результатів для code completion, навігації й інших сервісів в окремому процесі. Це значить, що code completion і Code Insight одержать більш точні результати без блокування IDE. 10.4 забезпечує набагато більш високу продуктивність розроблювачів, які працюють із більшими проектами, що містять мільйони рядків коду.

Delphi Custom Managed Records

Ключове розширення мови Delphi: тип даних Delphi «record» тепер підтримуть довільні ініціалізацію, фіналізацію й операції копіювання. Управляйте тем, як ці структури створюються, копіюються й звільняються з допомогу вашого коду, який буде виконуватися у відповідний момент.

Це розширює потужність конструкцій records в Delphi, які використовуться щоб одержати більшу ефективність у порівнянні із класами.

Єдине керування пам'яттю

Керування пам'яттю в Delphi тепер стандартизоване на всіх підтримуваних платформах – мобільних, настільних і серверних – використовучи класичну реалізацію керування пам'яттю об'єктів.

У порівнянні з Automatic Reference Counting (ARC), це дає кращу сумісність із існучим кодом і спрощує написання компонентів, бібліотек і застосунків.

ARC модель керування пам'яттю model залишилася для керування рядками й посиланнями на тип інтерфейсу на всіх платформах. Для C++ це означає, що при створенні й звільненні Delphi-style класів в C++ використовується звичайне керування пам'яттю, як у будь-якого heap-allocated класу C++, що значно знижує складність коду.

Розширена підтримка бібліотек C++

В 10.4 ми портували багато популярних бібліотек C++ у C++Builder.

Забезпечивши оптимізовану підтримку бібліотек ZeroMQ, SDL2, SOCI, libSIMDpp і Nematode, поряд із уже підтримуваними Boost і Eigen, які можуть бути додані за допомогою менеджера пакетів Getit.

Win 64-відладник і збирач для C++

В 10.4 з'явився новий відладник C++ для Windows 64-bit. Відладник заснований на LLDB і показує значне збільшення стабільності при налагодженні 64-bit застосунків поряд з новими відладочними можливостями, такими як перегляд і інспекція типів начебто рядків C++ і Delphi, а також колекцій STL, включаючи std::vector, std::map і інших. Крім того, згенерована для застосунку відладочна інформація має інший внутрішній формат, сприяючи більш стабільному й багатому на можливості процесу налагодження, більш докладним перегляду й інспекції в debug-time.

Підвищення якості й швидкодії інструментів

– Велика кількість поліпшень STL від Dinkumware.

– Поліпшені деякі найважливіші методи й області RTL, на базі поліпшень сумісності з популярними бібліотеками C++.

– Поліпшена підтримка Cmake.

– Велика кількість виправлень для підвищення стабільності і якості.

– Відновлення Windows API – Обновлено й додали безліч декларацій API щоб добитися ще більшої інтеграції із платформою Windows.

– Загальні вдосконалення в бібліотеці доступу до БД FireDAC, включаючи оновлені драйвера для FireBird, PostgreSQL і SQLite. Вибір статичного або динамічного підключення SQLite до застосунку.

Змінені стилі VCL для High DPI

В 10.4, архітектура стилізації VCL була суттєво розширена для підтримки High DPI і 4K моніторів. Тепер усі елементи UI на формі VCL автоматично масштабуться під відповідне до монітора дозвіл для показу форми. Був оновлений API стилізації для підтримки стилів high DPI.

Кожний графічний елемент UI може бути обраний з наборів різних масштабів і масштабований до потрібного DPI, що дає чітке зображення елементів UI на всіх моніторах.

Нові High DPI стилі й стилізація окремих VCL компонент

Обновлено велике число вбудованих і преміальних VCL стилів для підтримки нового режиму стилізації High-dpi. Це дозволяє вам створювати застосунку з відмінним дизайном для всіх моніторів.

Розроблювачі VCL застосунків тепер можуть використовувати трохи VCL стилів на різних формах в одному застосунку або в різних компонентів на одній формі. Це також включає стилізацію компонентів загальною темою для платформи. Крім застосункової гнучкості використання стилів, це дозволяє використовувати нестилізуємі компоненти із зовнішніх бібліотек в VCL застосунках, що використовуть стиль.

Поліпшена кроссплатформеність

– Додана підтримка Metal Driver GPU для macOS і iOS.

– Крім підтримки останнього iOS SDK, в RAD Studio 10.4 розроблювачі можуть задовольнити нові вимоги Apple до набору стартових екранів.

– Реалізований заново стилізуємий FMX компонент TMemo на платформі Windows значно поліпшений і тепер має відмінну підтримку IME.

– Користувачам редакцій Enterprise або Architect доступна повна інтеграція Fmxlinux з IDE для створення клієнтських застосунків Linux з GUI.

– Компонент Twebbrowser для iOS тепер реалізований на Wkwebview API.

– Реалізація компонента Media Player для macOS тепер використовує Avfoundation.

Оновлений менеджер пакетів Getit

Менеджер пакетів Getit в IDE був значно вдосконалений.

Дати випуску релізів пакетів тепер видні, і можливе сортування списку по цих датах; відбір тільки встановлених пакетів, контенту, доступного тільки при наявності підписки, багато чого іншого.

Універсальний інсталятор для установки Online і Offline

В 10.4 включений новий універсальний інсталятор, який використовує технологію на базі Getit. Цей інсталятор підтримує як online, так і offline (з ISO) варіанти установки.

Тепер обоє варіанта установки дозволяють вам указати початковий набір можливостей RAD Studio для установки, наприклад, свою комбінацію мов програмування й цільових платформ, мов інтерфейсу, і додавати до нього або видаляти непотрібне в будь-який момент.

2.3 Розгорнута постановка завдання

Згідно з технічним завданням на випускну кваліфікаційну роботу за першим (бакалаврським) рівнем вищої освіти, реалізації підлягає програмне

забезпечення, яке призначено для системи емуляції розподілу динамічної пам’яті.

В процесі розробки випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти необхідно виконати наступний обсяг роботи:

а) провести аналіз існуючих систем-аналогів для виявлення їх позитивних і негативних якостей. Результати аналізу врахувати в подальших розробках;

б) вибрати та обґрунтувати методику побудови системи контролю роботи технологічного обладнання на виробництві в автоматизованому режимі. Розробити функціональну та структурну схеми системи;

в) розробити програмне забезпечення системи, що дозволить реалізувати поставлену технічним завданням задачу. Побудувати блок-схеми алгоритмів програми та підпрограми;

г) організувати інтерфейс користувача з метою формування та виводу на екран ЕОМ повідомлень про некоректні дії користувача та нестандартні ситуації в роботі технологічного обладнання;

д) розробити рекомендації по організаційних та методичних заходах, які забезпечать впровадження системи в промислову експлуатацію та її подальшу успішну експлуатацію;

е) провести розрахунки по визначенню економічної ефективності розробленої системи;

ж) розробити заходи по охороні праці при впровадженні та експлуатації системи, а також розробити заходи з цивільного захисту;

з) сформувати висновки про виконаний обсяг робіт та одержані результати.

3 ОПИС І ОБҐРУНТУВАННЯ ПРОЕКТНИХ РІШЕНЬ

3.1 Опис функціонування системи

RAM (оперативна пам’ять) – це апаратне забезпечення комп’ютерного пристрою, де зберігаються операційна система (ОС), прикладні програми та дані, що використовуються на даний момент, щоб процесор пристрою міг швидко отримати доступ до них. RAM – це основна пам'ять комп'ютера. Читати та записувати з нього набагато швидше, ніж з інших типів накопичувачів, таких як жорсткий диск (HDD), твердотільний накопичувач (SSD) або оптичний привід.

Оперативна пам'ять є енергозалежною. Це означає, що дані зберігаються в оперативній пам’яті, поки комп’ютер увімкнено, але втрачаються, коли комп’ютер вимикається. Коли комп’ютер перезавантажується, ОС та інші файли перезавантажуються в оперативну пам’ять, як правило, з жорсткого диска або SSD.

Функція оперативної пам'яті

Через свою мінливість оперативна пам'ять не може зберігати постійні дані. Оперативну пам'ять можна порівняти з короткочасною пам'яттю людини, а жорсткий диск – з довготривалою пам'яттю людини. Короткочасна пам’ять зосереджена на миттєвій роботі, але вона може одночасно тримати в полі зору лише обмежену кількість фактів. Коли короткочасна пам’ять людини заповнюється, її можна оновити фактами, що зберігаються в довгостроковій пам’яті мозку.

Комп'ютер також працює таким чином. Якщо оперативна пам’ять переповнюється, процесор комп’ютера повинен постійно звертатися до жорсткого диска, щоб накласти старі дані в оперативній пам’яті на нові. Цей процес сповільнює роботу комп'ютера.

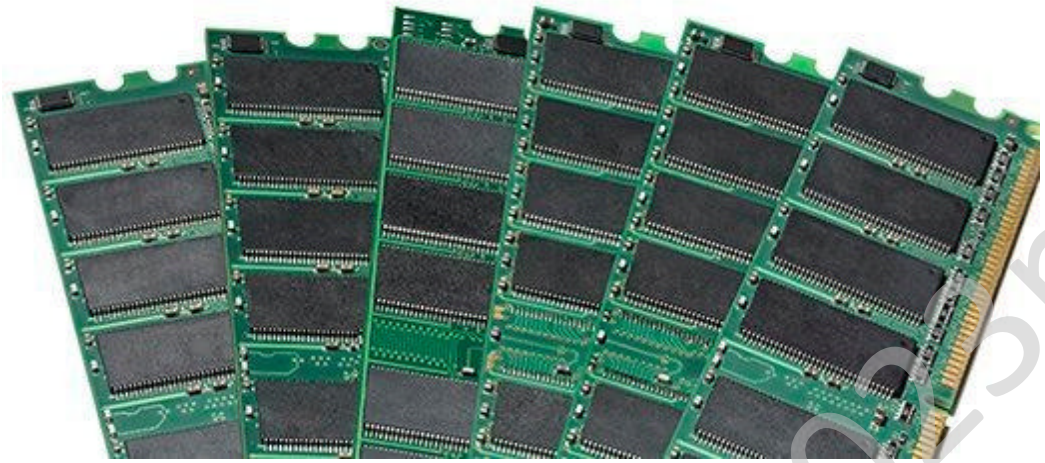

Рисунок 3.1 – Модулі оперативної пам'яті

Жорсткий диск комп’ютера може бути повністю заповнений даними і не зможе більше приймати, але оперативна пам’ять не вичерпається. Однак комбінація оперативної пам’яті та пам’яті може бути повністю використана.

Як працює оперативна пам'ять?

Термін *довільний доступ* у застосуванні до оперативної пам’яті походить від того факту, що до будь-якого місця зберігання, також відомого як будь-яка адреса пам’яті, можна отримати прямий доступ. Спочатку термін *Random Access Memory* використовувався для відмінності звичайної основної пам’яті від автономної пам’яті.

Автономна пам’ять зазвичай відноситься до магнітної стрічки , з якої певний фрагмент даних можна отримати лише шляхом послідовного визначення адреси, починаючи з початку стрічки. Оперативна пам’ять організована та керована таким чином, щоб дані можна було зберігати та отримувати безпосередньо з певних місць.

Інші типи сховищ, такі як жорсткий диск і CD-ROM, також доступні безпосередньо або довільно, але термін *довільний доступ* не використовується

для опису цих інших типів сховищ.

RAM схожа за концепцією на набір ящиків, у кожному з яких може міститися 0 або 1. Кожен ящик має унікальну адресу, яку можна знайти шляхом підрахунку по стовпцях і вниз по рядках. Набір блоків RAM називається масивом , а кожен блок називається коміркою.

Щоб знайти конкретну комірку, контролер оперативної пам’яті надсилає адресу стовпця та рядка по тонкій електричній лінії, викарбуваній на чіпі. Кожний рядок і стовпець в масиві RAM має власний рядок адреси. Будь-які зчитані дані повертаються в окремий рядок даних.

Оперативна пам'ять фізично мала і зберігається в мікросхемах. Він також невеликий з точки зору обсягу даних, який він може вмістити. Типовий портативний комп’ютер може мати 8 гігабайт оперативної пам’яті, тоді як жорсткий диск може вмістити 10 терабайт.

Жорсткий диск, з іншого боку, зберігає дані на намагніченій поверхні чогось схожого на вінілову платівку. Крім того, SSD зберігає дані в мікросхемах пам’яті, які, на відміну від оперативної пам’яті, є енергонезалежними. Вони не залежать від постійного живлення та не втратять дані після вимкнення живлення. Мікросхеми оперативної пам'яті зібрані в модулі пам'яті. Вони вставляються в слоти на материнській платі комп’ютера. Шина, або набір електричних шляхів, використовується для підключення слотів материнської плати до процесора.

Більшість ПК дозволяють користувачам додавати модулі оперативної пам’яті до певної межі. Збільшення оперативної пам’яті в комп’ютері скорочує кількість разів, коли процесор повинен зчитувати дані з жорсткого диска, операція, яка займає більше часу, ніж зчитування даних з оперативної пам’яті. Час доступу до оперативної пам’яті – у наносекундах, а до пам’яті – у мілісекундах.

Скільки оперативної пам'яті вам потрібно?

Необхідний обсяг оперативної пам’яті залежить від того, що робить користувач. Під час редагування відео, наприклад, рекомендується, щоб система

мала принаймні 16 ГБ оперативної пам’яті, хоча бажано більше. Для редагування фотографій за допомогою Photoshop Adobe рекомендує систему мати принаймні 3 ГБ оперативної пам’яті для запуску Photoshop CC на Mac. Однак, якщо користувач одночасно працює з іншими програмами, навіть 8 ГБ оперативної пам’яті можуть уповільнити роботу.

Типи оперативної пам'яті

RAM буває двох основних форм:

– **Динамічна пам'ять з довільним доступом (DRAM)** становить типову оперативну пам'ять обчислювального пристрою, і, як зазначалося раніше, для збереження збережених даних їй потрібно ввімкнути живлення.

Кожна комірка DRAM має заряд або відсутність заряду, що утримується в електричному конденсаторі. Ці дані необхідно постійно оновлювати електронним зарядом кожні кілька мілісекунд, щоб компенсувати витоки з конденсатора. Транзистор служить затвором, який визначає, чи можна прочитати або записати значення конденсатора.

– **Статичній пам’яті з довільним доступом (SRAM)** також потрібна постійна потужність, щоб зберігати дані, але її не потрібно постійно оновлювати, як це робить DRAM.

У SRAM замість конденсатора, що утримує заряд, транзистор діє як перемикач, де одна позиція є 1, а інша позиція – 0. Статичній RAM потрібно кілька транзисторів, щоб зберігати один біт даних, у порівнянні з динамічною RAM, якій потрібен лише один транзистора на біт. Як наслідок, чіпи SRAM набагато більші та дорожчі, ніж еквівалентна кількість DRAM.

Однак SRAM значно швидший і споживає менше енергії, ніж DRAM. Різниця в ціні та швидкості означає, що статична оперативна пам’ять використовується в основному в невеликих кількостях як кеш-пам’ять у процесорі комп’ютера.

Історія RAM: RAM проти SDRAM

Оперативна пам'ять спочатку була асинхронною, оскільки мікросхеми

оперативної пам'яті мали тактову частоту, відмінну від частоти процесора комп'ютера. Це було проблемою, оскільки процесори стали потужнішими, а оперативна пам’ять не могла встигати за запитами процесора щодо даних.

На початку 1990-х тактові частоти були синхронізовані з впровадженням синхронної динамічної оперативної пам’яті або SDRAM. Синхронізуючи пам’ять комп’ютера з вхідними даними процесора, комп’ютери могли швидше виконувати завдання.

Однак початкова стандартна SDRAM (SDR SDRAM) швидко досягла своєї межі. Приблизно в 2000 році була розроблена синхронна оперативна пам'ять з подвійною швидкістю передачі даних (DDR SRAM). Це двічі переміщувало дані за один такт, на початку та в кінці.

DDR SDRAM еволюціонував тричі, з DDR2, DDR3 і DDR4, і кожна ітерація принесла покращену швидкість передачі даних і зменшене енергоспоживання. Однак кожна версія DDR була несумісна з попередніми, оскільки з кожною ітерацією дані обробляються більшими пакетами.

Рисунок 3.2 – Зображення різних форматів DDR

GDDR SDRAM

Графічна подвійна швидкість передачі даних (GDDR) SDRAM використовується в графічних і відеокартах. Подібно до DDR SDRAM, ця технологія дозволяє переміщувати дані в різні моменти циклу ЦП. Однак він працює при вищій напрузі та має менш суворий час, ніж DDR SDRAM.

З паралельними завданнями, такими як візуалізація 2D і 3D відео, стислий час доступу не є необхідним, а GDDR може забезпечити більш високу швидкість і пропускну здатність пам’яті, необхідні для продуктивності GPU.

Подібно до DDR, GDDR пройшла кілька поколінь розробки, кожне з яких забезпечувало більшу продуктивність і менше енергоспоживання. GDDR6 – це останнє покоління графічної пам'яті.

Оперативна пам'ять проти віртуальної пам'яті

Комп’ютеру може не вистачати пам’яті, особливо під час одночасної роботи кількох програм. Операційні системи можуть компенсувати нестачу фізичної пам'яті шляхом створення віртуальної пам'яті.

За допомогою віртуальної пам’яті дані тимчасово переносяться з оперативної пам’яті на дискове сховище, а віртуальний адресний простір збільшується за допомогою активної пам’яті в оперативній пам’яті та неактивної пам’яті на жорсткому диску для формування безперервних адрес, які містять програму та її дані. Використовуючи віртуальну пам’ять, система може завантажувати більші програми або кілька програм, що працюють одночасно, дозволяючи кожній працювати так, ніби вона має нескінченну пам’ять, без необхідності додавати більше оперативної пам’яті.

Віртуальна пам'ять здатна обробляти вдвічі більше адрес, ніж оперативна пам'ять. Інструкції та дані програми спочатку зберігаються за віртуальними адресами, а після виконання програми ці адреси перетворюються на фактичні адреси пам’яті.

Одним із недоліків віртуальної пам’яті є те, що вона може уповільнювати роботу комп’ютера, оскільки дані мають бути зіставлені між віртуальною та

фізичною пам’яттю. За допомогою лише фізичної пам’яті програми працюють безпосередньо з оперативної пам’яті.

RAM проти флеш-пам'яті

Флеш-пам'ять і оперативна пам'ять складаються з твердотільних мікросхем. Однак вони відіграють різні ролі в комп’ютерних системах через відмінності у способах їх виготовлення, характеристиках продуктивності та вартості. Флеш-пам'ять використовується для зберігання даних. RAM використовується як активна пам’ять, яка виконує обчислення даних, отриманих із сховища.

Одна суттєва відмінність між RAM і флеш-пам'яттю полягає в тому, що дані повинні бути видалені з флеш-пам'яті NAND цілими блоками. Це робить його повільнішим, ніж оперативна пам’ять, де дані можна стирати окремими бітами.

Однак флеш-пам’ять NAND дешевша, ніж оперативна пам’ять, і також є енергонезалежною. На відміну від оперативної пам'яті, він може зберігати дані навіть при вимкненому живленні. Через повільнішу швидкість, енергонезалежність і нижчу вартість флеш-пам’ять часто використовують для зберігання на SSD.

RAM проти ROM

Постійна пам’ять, або ROM, – це пам’ять комп’ютера, що містить дані, які можна лише читати, але не записувати. ПЗП містить програму завантаження, яка використовується кожного разу, коли комп’ютер увімкнено. Зазвичай його не можна змінити чи перепрограмувати.

Дані в ПЗУ є енергонезалежними і не втрачаються, коли комп’ютер вимикається. У результаті постійна пам'ять використовується для постійного зберігання даних. З іншого боку, оперативна пам’ять може зберігати дані лише тимчасово. ПЗП зазвичай становить кілька мегабайт пам’яті, тоді як ОЗП – кілька гігабайт.

Тенденції та напрямки майбутнього

Резистивна оперативна пам’ять (RRAM або ReRAM) – це енергонезалежна пам’ять, яка може змінювати опір твердого діелектричного матеріалу, з якого вона складається. Пристрої ReRAM містять мемристор, опір якого змінюється при застосуванні різних напруг.

ReRAM створює кисневі вакансії, які є фізичними дефектами в шарі оксидного матеріалу. Ці вакансії представляють два значення в подвійній системі, подібній до електронів і дірок у напівпровіднику.

ReRAM має вищу швидкість перемикання порівняно з іншими технологіями енергонезалежного зберігання, такими як флеш-пам’ять NAND. Він також обіцяє високу щільність зберігання та менше енергоспоживання, ніж флеш-пам’ять NAND. Це робить ReRAM хорошим варіантом для пам’яті в датчиках, які використовуються в промислових, автомобільних і Інтернет речей.

Постачальники роками боролися за розробку технології ReRAM і запуск чіпів у виробництво. Кілька постачальників зараз їх доставляють.

Технологія 3D XPoint, така як Optane від Intel, може врешті-решт заповнити прогалину між динамічною оперативною пам’яттю та флеш-пам’яттю NAND. 3D XPoint має безтранзисторну точкову архітектуру, у якій селектори та комірки пам’яті знаходяться на перетині перпендикулярних проводів. 3D XPoint не така швидка, як DRAM, але це енергонезалежна пам’ять.

Рисунок 3.3 – SSD Optane на базі 3D XPoint від Intel

З точки зору продуктивності та ціни, технологія 3D XPoint знаходиться між швидкою, але дорогою DRAM і повільнішою, менш дорогою флеш-пам’яттю NAND. З розвитком технології це може стирати різницю між оперативною пам’яттю та пам’яттю.

5G і ринок оперативної пам'яті

У лютому 2022 року Асоціація твердотільних технологій JEDEC опублікувала JESD209-5, Low Power Double Data Rate 5 (LPDDR5). Згодом LPDDR5 працюватиме зі швидкістю введення-виведення 6400 MT/c, що на 50 відсотків вище, ніж у першої версії LPDDR4. Це значно підвищить швидкість пам'яті та ефективність для різноманітних програм. Це стосується мобільних комп’ютерних пристроїв, таких як смартфони, планшети та ультратонкі ноутбуки.

LPDDR5 було опубліковано зі швидкістю передачі даних 6400 MT/c, порівняно з 3200 MT/c для LPDDR4 під час публікації в 2014 році.

У липні 2022 року Samsung Electronics почала масове виробництво першої в галузі 12-гігабітної мобільної DRAM LPDDR5. За словами Samsung, він був оптимізований для включення функцій 5G і штучного інтелекту в майбутніх смартфонах.

Вартість оперативної пам'яті

До літа 2022 року ціни на DRAM залишалися нижчими порівняно з попередніми рівнями, але, тим не менш, нестабільними. Кілька змінних впливають на волатильність, зокрема:

– надлишок пропозиції;

– ринкова напруженість між Південною Кореєю та Японією (де розташовані два найбільших у світі виробники мікросхем пам’яті, Samsung і SK Hynix);

– представлення мобільного чіпа нового покоління LPDDR5;

– збільшення впровадження технології 5G;

– очікуване збільшення попиту на споживчу електроніку в Інтернеті речей (IoT), таку як автомобілі та переносні пристрої, які використовують чіпи.

3.2 Розробка структурної схеми

Структурна схема розробленої системи показана на рисунку 3.4.

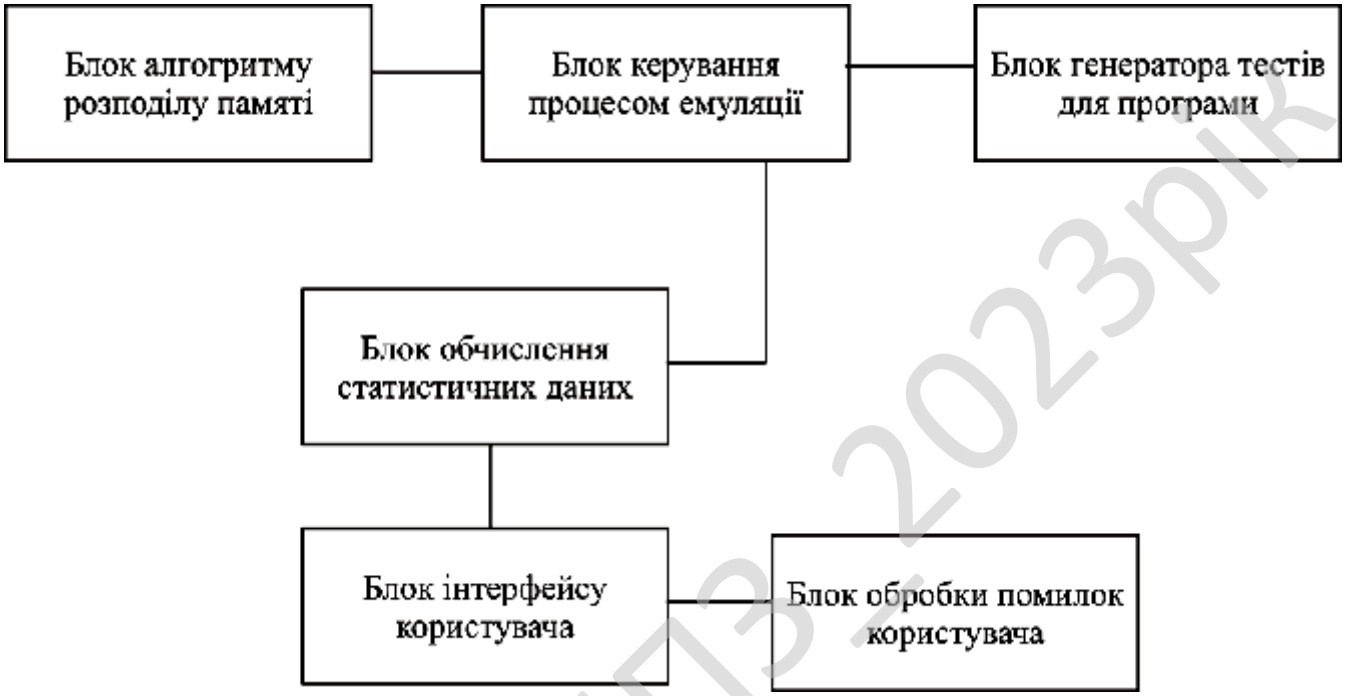

Рисунок 3.4 – Структурна схема системи

Зі схеми видно, що система складається з наступних блоків:

- Блок алгоритму розподілу пам’яті.
- Блок керування процесом емуляції.
- Блок генератора тестів для програми.
- Блок обчислення статистичних даних.
- Блок інтерфейсу користувача.
- Блок обробки помилок користувача.

Блок алгоритму розподілу пам’яті містить функції та процедури, що управляють розподілом пам'яті. Для розгляду було вирішено обрати алгоритм "близнюків" (описаний у четвертому розділі даної бакалаврської роботи), але його легко можна замінити на інший, не втручаючись в інші блоки програми.

Блок керування процесом емуляції містить функції, що дозволяють змінити швидкість емуляції, призупинити або припинити її та вивести на екран чи прибрати статистичну інформацію. Всі ці функції створені винятково для більш зручної роботи з програмою. Уповільнення, наприклад, дозволяє детальніше роздивитися зміни, що відбуваються з картою пам'яті та чергою задач, а прискорення доречне якщо список задач дуже довгий.

Блок генератора тестів для програми дозволяє автоматично створити чергу задач без нудного та довгого створення окремо кожної задачі вручну.

Блок обчислення статистичних даних містить функції, що обчислюють скільки процентів динамічної пам'яті зайнято у кожний момент часу виконання проекту, а також скільки програм отримали відмови у поточний момент часу. За допомогою даного модуля можна оцінити ефективність використаного алгоритму розподілу пам'яті.

Блок інтерфейсу користувача містить процедури обробки натиснень кнопок та меню користувача, графічне відображення карти пам'яті, черги задач, інформаційної панелі, що містить статистику, список вільних блоків пам'яті тощо.

Блок обробки помилок користувача включає функції перевірки допустимих значень при створенні користувачем проекту і черги задач та при наявності помилок виведення інформаційних повідомлень, що пояснюють суть помилки. Наприклад перевіряється введений розмір задачі, він не повинен перевищувати розміру наявної динамічної пам'яті. Якщо користувач допустив помилку, виводиться повідомлення про перевищення допустимого ліміту та пропонується ввести менше значення.

3.3 Розробка функціональної схеми

Розглянемо функціональну схему системи, що зображена на рисунку 3.5. Умовно програму можна розділити на блок управління та користувацький інтерфейс.

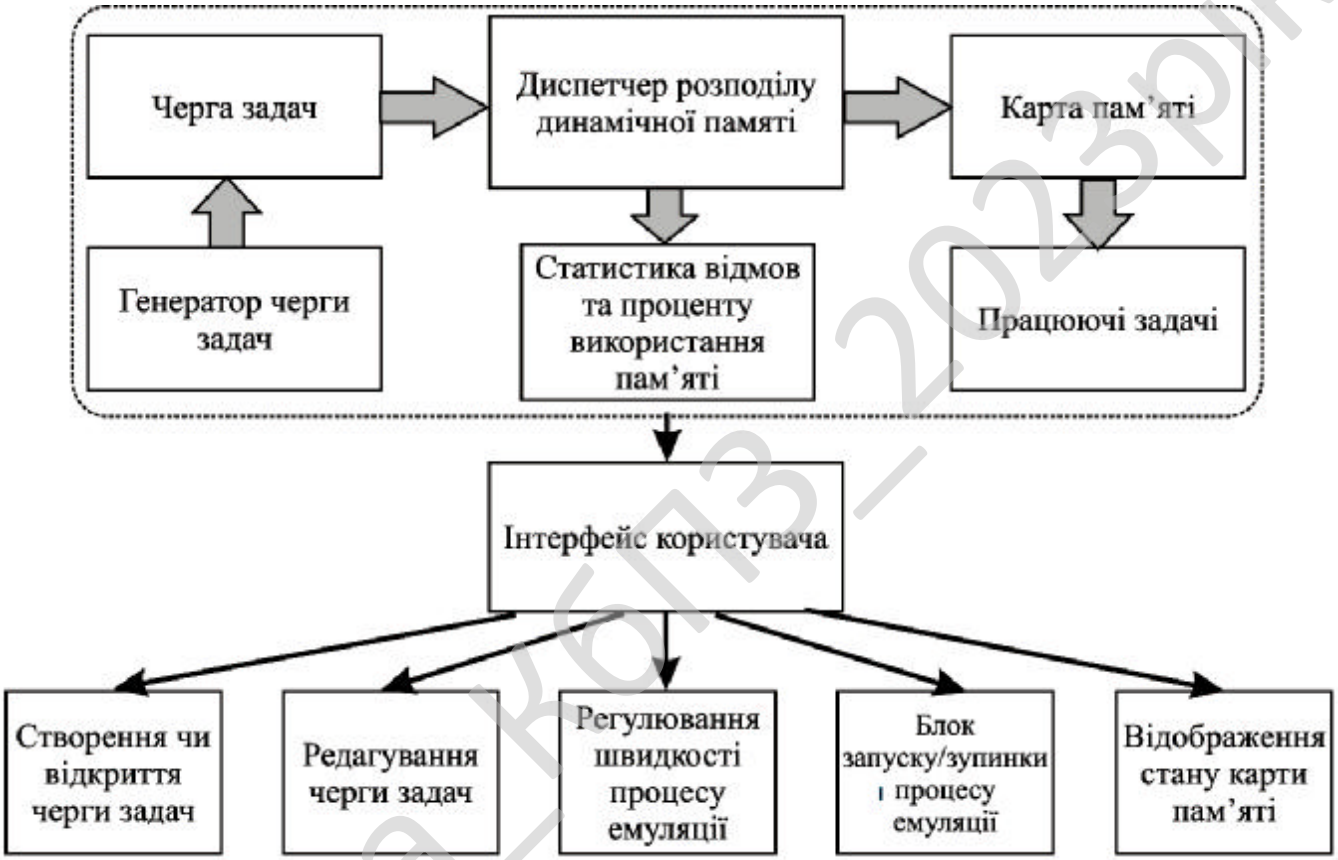

Рисунок 3.5 – Функціональна схема системи

Блок управління складається з наступних елементів:

- Генератор черги задач.
- Черга задач.
- Диспетчер розподілу динамічної пам'яті.
- Карта пам'яті.
- Працюючі задачі.
- Статистика відмов та проценту використання пам'яті.

Генератор черги використовується для автоматичного створення черги задач. Сама черга містить список задач у наступному форматі:

– Назва задачі.
– Розмір задачі.
– Початок виконання.
– Тривалість.

Диспетчер розподілу динамічної пам’яті містить алгоритм управління пам'яттю та здійснює розміщення задач у динамічній пам'яті, а також їх видалення по завершенню виконання.

Карта пам'яті зображена у вигляді сукупності комірок, що мають різний колір залежно від свого стану (зайнята чи вільна). Також окремо відображається таблиця, що містить список вільних блоків пам'яті, а саме їх адресу та розмір.

Працюючі задачі відображаються в окремій таблиці, що містить їх назву, розмір, адресу у пам'яті та час завершення.

Статистика відмов та проценту використання пам’яті відображається в окремому вікні та містить кількість відмов та завантаженість пам'яті.

Користувацький інтерфейс містить наступні функції:

– Створення чи відкриття черги задач.
– Редагування черги задач.
– Регулювання швидкості процесу емуляції.
– Блок запуску/зупинки процесу емуляції.
– Відображення стану карти пам’яті.

З функціональної схеми видно, що спочатку відбувається генерація черги задач, потім створена черга обробляється диспетчером розподілу динамічної пам'яті. Диспетчер бере з черги задачу та шукає для неї вільне місце у динамічній пам'яті використовуючи для цього алгоритм близнюків. Якщо місце знайдене задача займає його, якщо ні, то вона отримує відмову та залишається у черзі. При цьому ведеться статистика відмов та проценту використання пам'яті.

Інтерфейс користувача отримує дані з блоку управління та відображає їх

на екрані у наглядному та зручному для розуміння вигляді.

Всі функції інтерфейсу користувача викликаються натисненням відповідної кнопки на панелі інструментів або вибором потрібного пункту меню. Відображення стану карти пам'яті відбувається постійно після створення проекту.

3.6 Розробка діаграми процесів

Діаграма процесів розробленої системи зображена на рисунку 3.6.

Рисунок 3.6 – Діаграма процесів системи

Розглянемо її складові. Після вмикання програми користувач запускає процес створення проекту та генерацію черги задач. Потім список задач відкривається та редагується. Редагування складається з двох процесів: створення нової задачі і додавання її у чергу та видалення задачі. Після редагування списку задач, користувач може запустити безпосередньо процес емуляції.

Після запуску емуляції стають доступними процеси редагування швидкості емуляції, ввімкненя інформаційної панелі, що містять статистичні дані та призупинення процесу емуляції. Перед виходом з програми можна скористатися процесом збереження списку задач.

Таким чином, розглянувши опис системи, структурну, функціональну схеми системи, та діаграму взаємодії процесів перейдемо до опису блок-схем основної програми, та підпрограм, які використовуються, для реалізації системи.

4 РЕАЛІЗАЦІЯ РОБОТИ. РОЗРАХУНКИ І ЕКСПЕРИМЕНТАЛЬНІ ДАНІ, ЩО ПІДТВЕРДЖУЮТЬ ВІРНІСТЬ ПРОЕКТНИХ ТА ПРОГРАМНИХ РІШЕНЬ

4.1 Блок-схеми та опис алгоритмів функціонування системи

Блок-схема основної програми наведена на рисунку 4.1. З рисунку видно, що робота програми починається зі створення нового проекту та введення користувачем розміру динамічної пам'яті. Після цього програма виводить на екран карту пам’яті вказаного розміру та з порожніми комірками.

Потім відбувається створення черги задач. Для цього користувач може використати генератор черги або кожну задачу додати вручну вказуючи при цьому назви задач, їх розміри, початок запуску та час виконання. Створену чергу можна редагувати. Після створення проекту та черги задач запускається процес емуляції. Він складається з циклу в якому задачі беруться з черги та розміщуються у пам'яті поки черга не спорожніє. На кожній ітерації циклу береться чергова задача та викликається для неї підпрограма пошуку вільного місця в пам’яті за допомогою алгоритму близнюків. Якщо місце знайдене, то задача запускається. Якщо ж ні, то вона отримує відмову та повертається до черги задач. Комірки пам'яті зайняті працюючою задачею зафарбовуються у рожевий колір у карті пам'яті. Якщо одна з запущених задач завершила своє виконання, то вона видаляється з пам'яті, а зафарбовані комірки знову стають білими. А звільнений блок пам'яті додається до списку вільних. Після завершення циклу відбувається очищення черги задач, очищення карти пам’яті та призупинення процесу емуляції. Після цього користувач може або вийти з програми, або створити новий проект та знову запустити процес емуляції. Також існує можливість перед виходом з програми зберегти чергу задач у текстовому файлі для повторного використання.

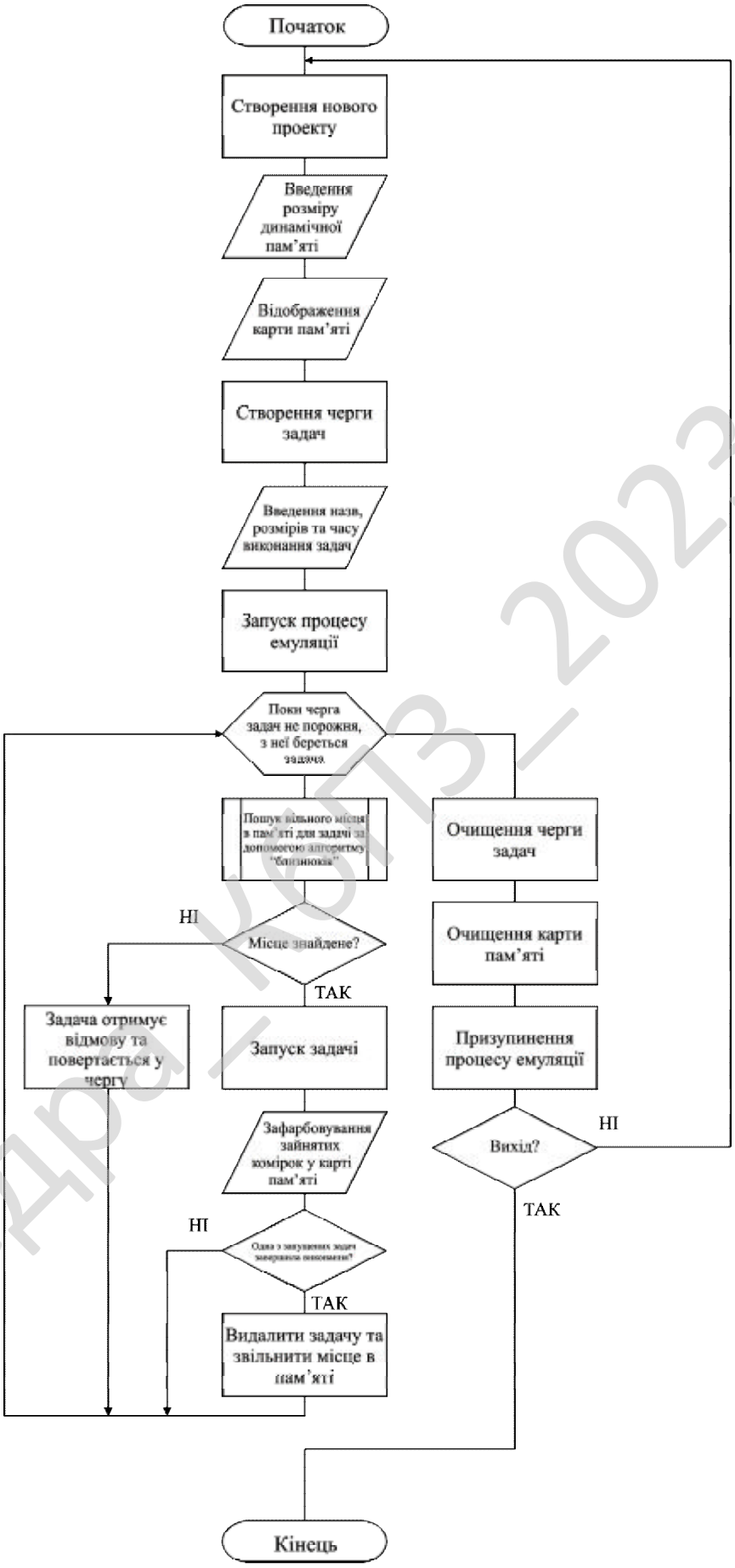

Рисунок 4.1 – Блок-схема основної програми

Розглянемо підпрограму пошуку вільного місця в пам’яті для задачі за допомогою алгоритму близнюків та обґрунтуємо чому був використаний саме цей алгоритм.

Пошук у списку може вестися двома способами: до знаходження першого підходящого (first fit) блоку або до блоку, розмір якого ближче всього до заданого – найбільш підходящого (best fit). Для знаходження найбільш підходящого треба переглядати весь список, у той час як перший підходящий може виявитися в будь-якому місці, і середній час пошуку буде менше. Наскільки менше – залежить від відношення кількості підходящих блоків до загальної кількості.

Крім того, у загальному випадку best fit збільшує фрагментацію пам'яті. Дійсно, якщо ми знайшли блок з розміром більше заданого, то повинні відокремити "хвіст" і позначити його як новий вільний блок. Зрозуміло, що у випадку best fit середній розмір цього хвоста буде маленьким, і ми в підсумку одержимо велику кількість дрібних блоків, які неможливо об'єднати, тому що простір між ними зайнято.

При використанні first fit з лінійним двонаправленим списком виникає специфічна проблема. Якщо щораз переглядати список з того самого місця, то більші блоки, розташовані ближче до початку, будуть частіше віддалятися. Відповідно, дрібні блоки будуть мати тенденцію накопичуватися на початку списку, що збільшить середній час пошуку. Простий спосіб боротьби із цим явищем полягає в тому, щоб переглядати список то в одному напрямку, то в іншому. Більш радикальний і ще більш простий метод полягає в тому, що список робиться кільцевим, і пошук кожний починається з того місця, де ми зупинилися минулого разу. У це ж місце додаються блоки, що звільнилися. У результаті список дуже ефективно перемішується й ніякий "антисортування" не виникає.

У ситуаціях, коли ми розміщаємо блоки декількох фіксованих розмірів, алгоритми best fit виявляються краще. Однак бібліотеки розподілу пам'яті розраховують на гірший випадок, і в них звичайно використовуються алгоритми

first fit. У випадку роботи із блоками декількох фіксованих розмірів напрошується таке рішення: створити для кожного типорозміру свій список. Це рятує програміста від необхідності вибирати між first і best fit, усуває пошук у списках і як наслідок вирішує відразу багато проблем. Цікавий варіант цього підходу для випадку, коли різні розміри є ступенями числа 2, як 512 байт, 1Кбайт, 2Кбайта й т.д., називається алгоритмом близнюків (рисунок 4.2).

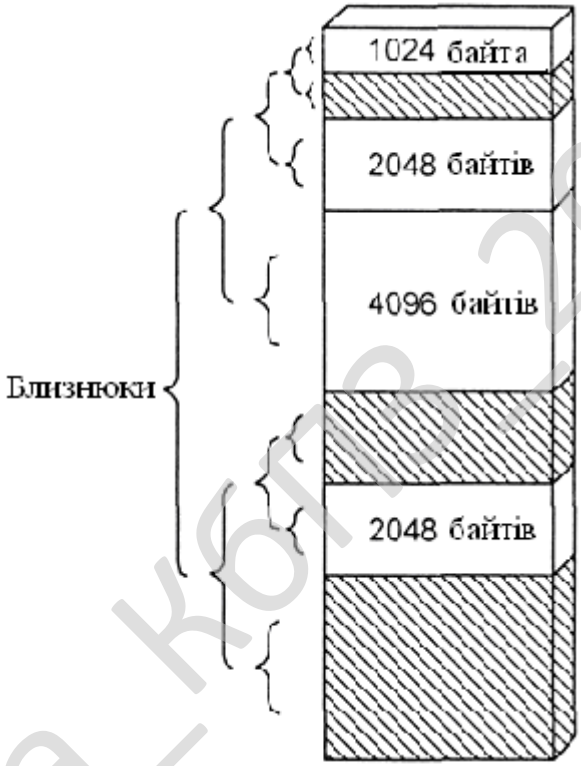

Рисунок 4.2 – Алгоритм близнюків

Він полягає в тому, що шукають блок необхідного розміру у відповідному списку. Якщо цей список порожній, береться список блоків удвічі більшого розміру. Одержавши блок удвічі більшого розміру, ділимо його навпіл. Непотрібну половину поміщаємо у відповідний список вільних блоків. Цікаво, що зовсім неважливо, чи одержаний цей блок просто з відповідного списку, або ж розподілом навпіл учетверо більшого блоку, і далі по рекурсії.

Блок схема алгоритму близнюків наведена на рисунку 4.3.

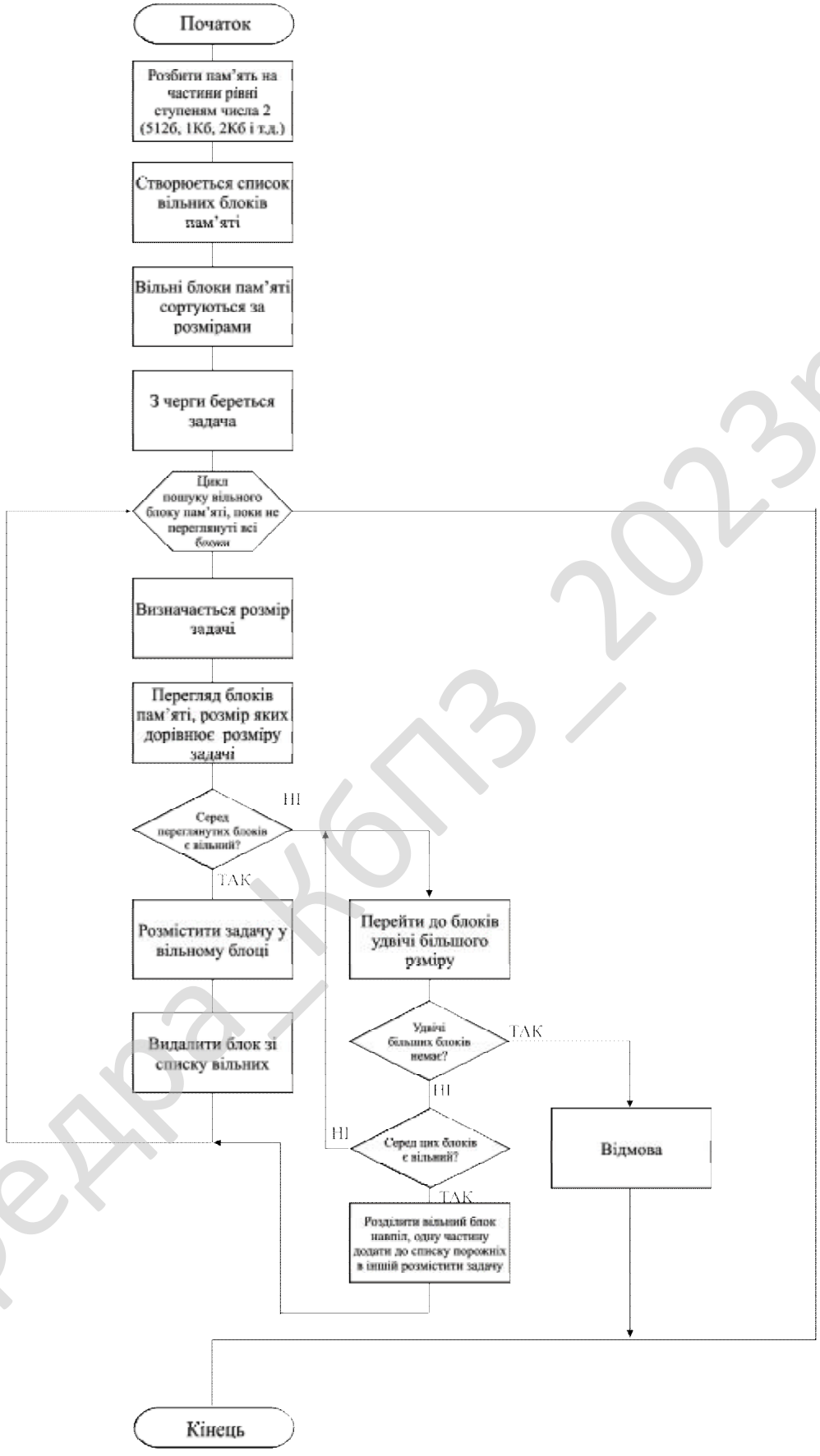

Рисунок 4.3 – Алгоритм роботи підпрограми пошуку вільного місця в пам'яті для задачі за допомогою алгоритму близнюків

Одна з переваг цього методу складається в простоті об'єднання блоків при їхньому звільненні.

Дійсно, адреса блоку-близнюка обчислюється простим інвертуванням відповідного біта в адресі блоку. Потрібно тільки перевірити, чи вільний цей близнюк. Якщо він вільний, то треба поєднати братів у блок удвічі більшого розміру, і т.д.

Навіть у найгіршому випадку час пошуку не перевищує $O(\log(S_{max})-\log(S_{min}))$, де S_{max} і S_{min} позначають, відповідно, максимальний і мінімальний розміри використовуваних блоків.

Це робить алгоритм близнюків тяжко замінюваним для ситуацій, у яких необхідний гарантований час реакції – наприклад, для завдань реального часу. Часто цей алгоритм або його варіанти використовуються для виділення пам'яті усередині ядра ОС.

Для роботи програми перш за все треба створити чергу задач. Для цього було розроблено спеціальний генератор, що використовує псевдовипадкові числа. Алгоритм створення черги представлений на рисунку 4.4.

Спочатку відбувається вивід вікна генератора черги задач. Потім ініціалізація генератора випадкових чисел. Запуск процесу генерації здійснює користувач, натиснувши на відповідну кнопку у вікні генератора.

Після цього створюється та відкривається текстовий файл. Назву файлу та його місцезнаходження на диску вказує користувач.

Потім здійснюється генерація розміру динамічної пам’яті. Далі генерується число N, що означає кількість задач у черзі. Далі починається цикл створення N задач. Задачі дається назва, що складається зі слова *задача* та номера ітерації *i*. Потім генерується її розмір, тривалість та час запуску. І задача додається до черги.

По завершенню циклу здійснюється сортування черги по часу запуску задач. Потім у файл записується розмір динамічної пам'яті, кількість задач і утворена та відсортована черга.

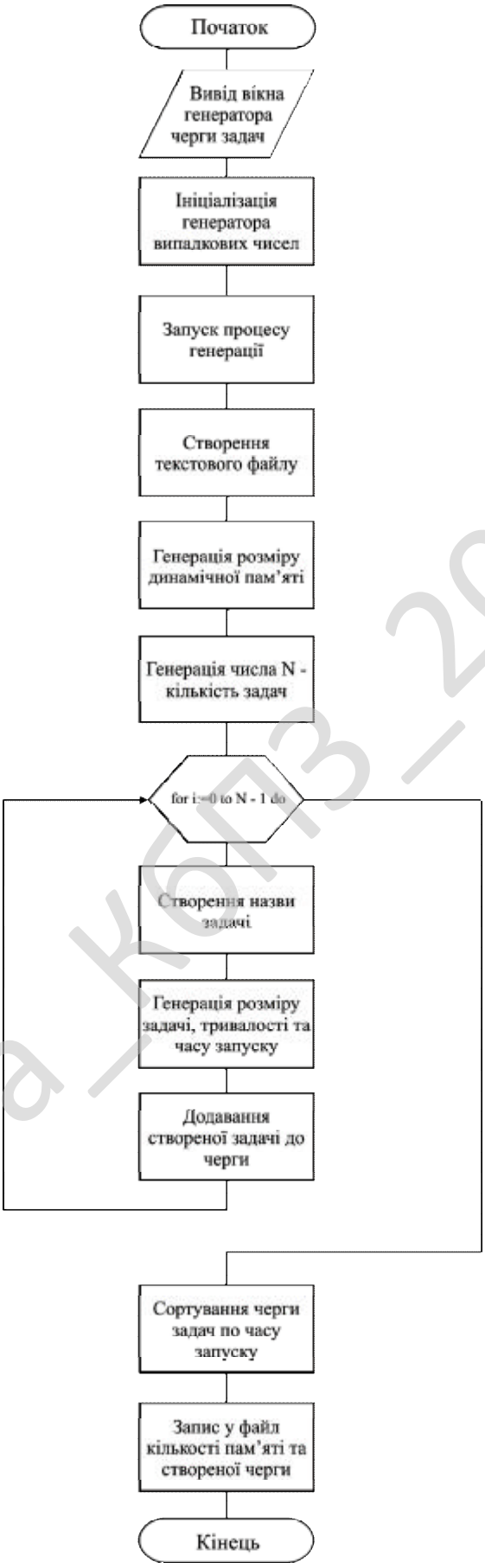

Рисунок 4.4 – Алгоритм роботи генератора черги задач

4.2 Захист розробленого програмного забезпечення

Захист розробленого програмного забезпечення буде відбуватися за допомогою алгоритму DES. DES є класичною мережею Фейштеля із двома гілками. Дані шифруються 64-бітними блоками, використовуючи 56-бітний ключ. Алгоритм перетворить за кілька раундів 64-бітний вхід в 64-бітний вихід. Довжина ключа дорівнює 56 бітам. Процес шифрування складається із чотирьох етапів. На першому з них виконується початкова перестановка (IP) 64-бітного вихідного тексту (забілювання), під час якої біти переставляються у відповідності зі стандартною таблицею. Наступний етап складається з 16 раундів однієї й тої ж функції, що використовує операції зрушення й підстановки. На третьому етапі ліва й права половини виходу останньої (16-й) ітерації міняються місцями. Нарешті, на четвертому етапі виконується перестановка IP^{-1} результату, отриманого на третьому етапі. Перестановка IP^{-1} інверсна початковій перестановці. Праворуч на рисунку показаний спосіб, яким використовується 56-бітний ключ. Спочатку ключ подається на вхід функції перестановки. Потім для кожного з 16 раундів підключ K_i є комбінацією лівого циклічного зрушення й перестановки. Функція перестановки та сама для кожного раунду, але підключи K_i для кожного раунду виходять різні внаслідок повторюваного зрушення біт ключа.

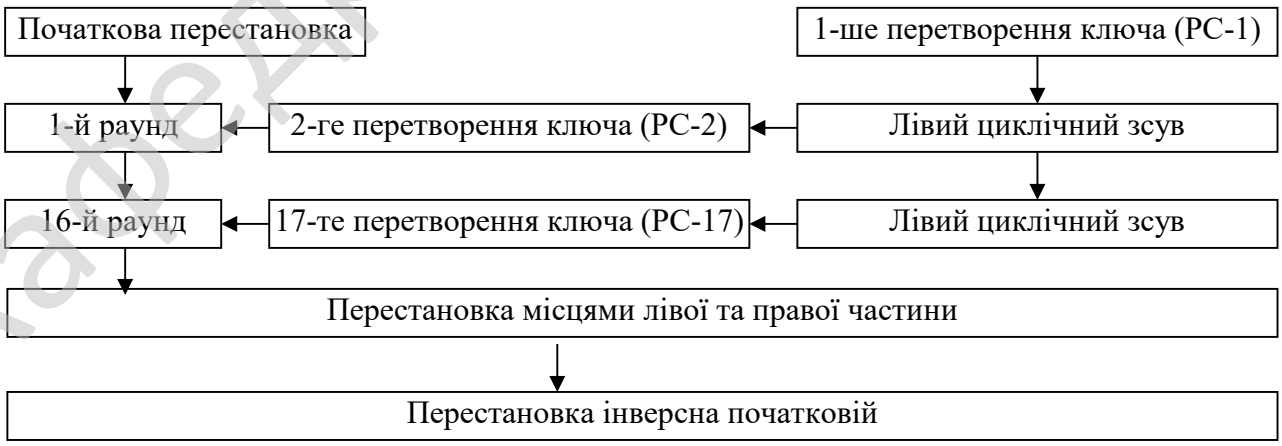

Рисунок 4.5 – Загальна схема DES

5 МЕТОДИКА ВПРОВАДЖЕННЯ СИСТЕМИ В ПРОМИСЛОВУ ЕКСПЛУАТАЦІЮ

Програма має простий та інтуїтивно зрозумілий інтерфейс, який зображений на рисунку 5.1.

Програма реалізує процес емуляції розподілу динамічної пам'яті трьома методами: "першого підходящого", "найбільш підходящого", по системі "близнюків".

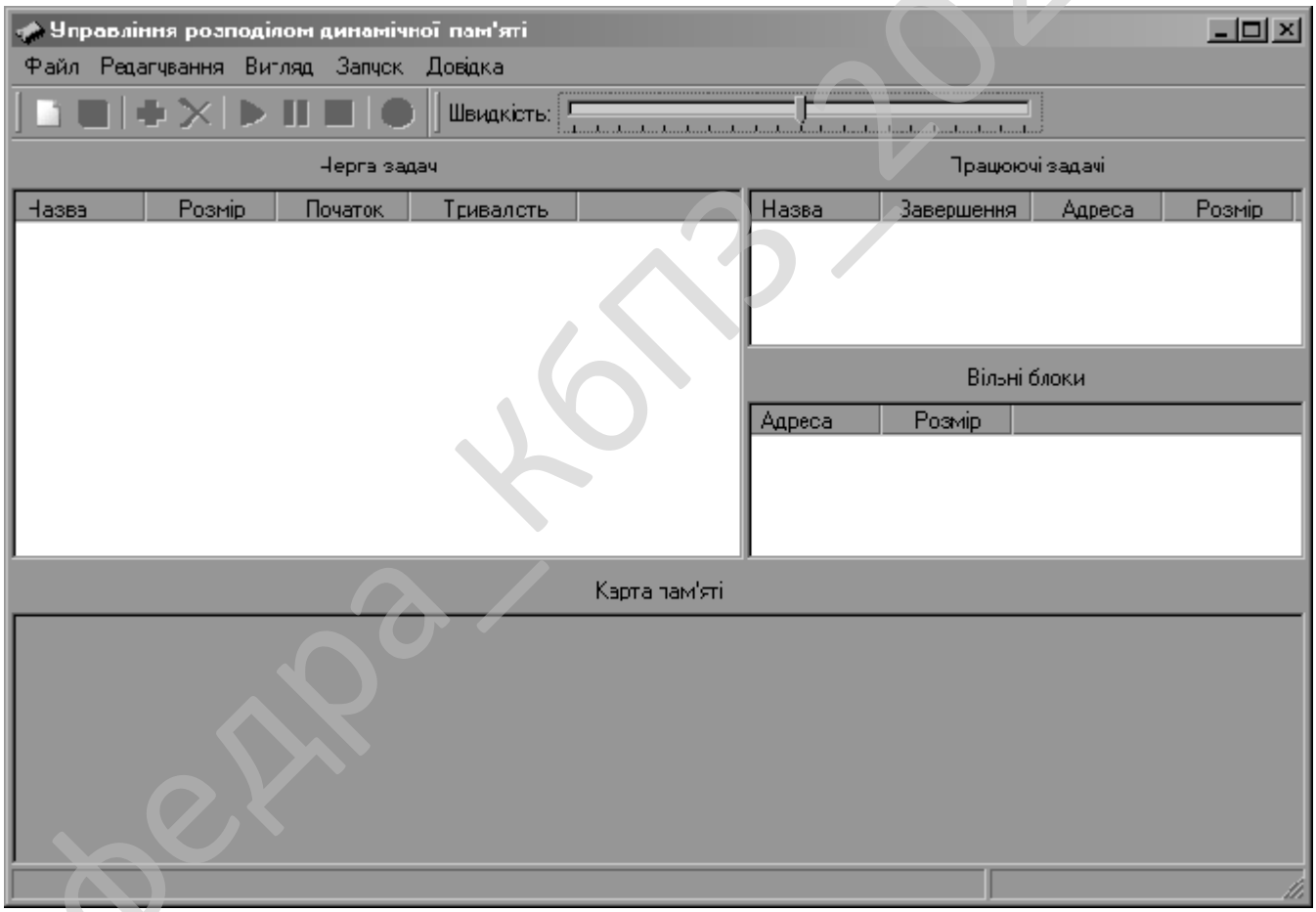

Рисунок 5.1 – Головне вікно програми

Для запуску процесу емуляції необхідно спочатку створити проект, в якому вказати кількість наявної динамічної пам'яті, шляхом введення найменшої

та найбільшої адреси (рисунок 5.2). Створення проекту можливе за допомогою пункту меню **Файл->Створити**, або ж за допомогою відповідної кнопки на панелі інструментів, що зображена у вигляді наступного символу – .

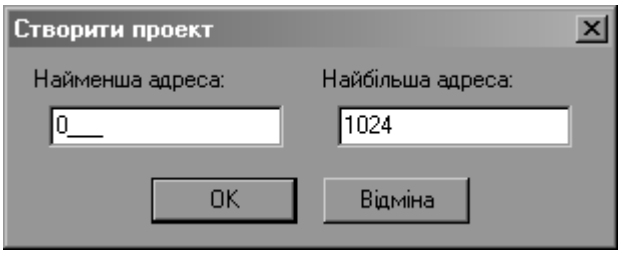

Рисунок 5.2 – Діалогове вікно створення проекту

Після створення проекту в головному вікні програми з'явиться карта пам'яті, що буде відображати усі наявні комірки пам'яті у вигляді невеликих білих квадратиків (рисунок 5.3).

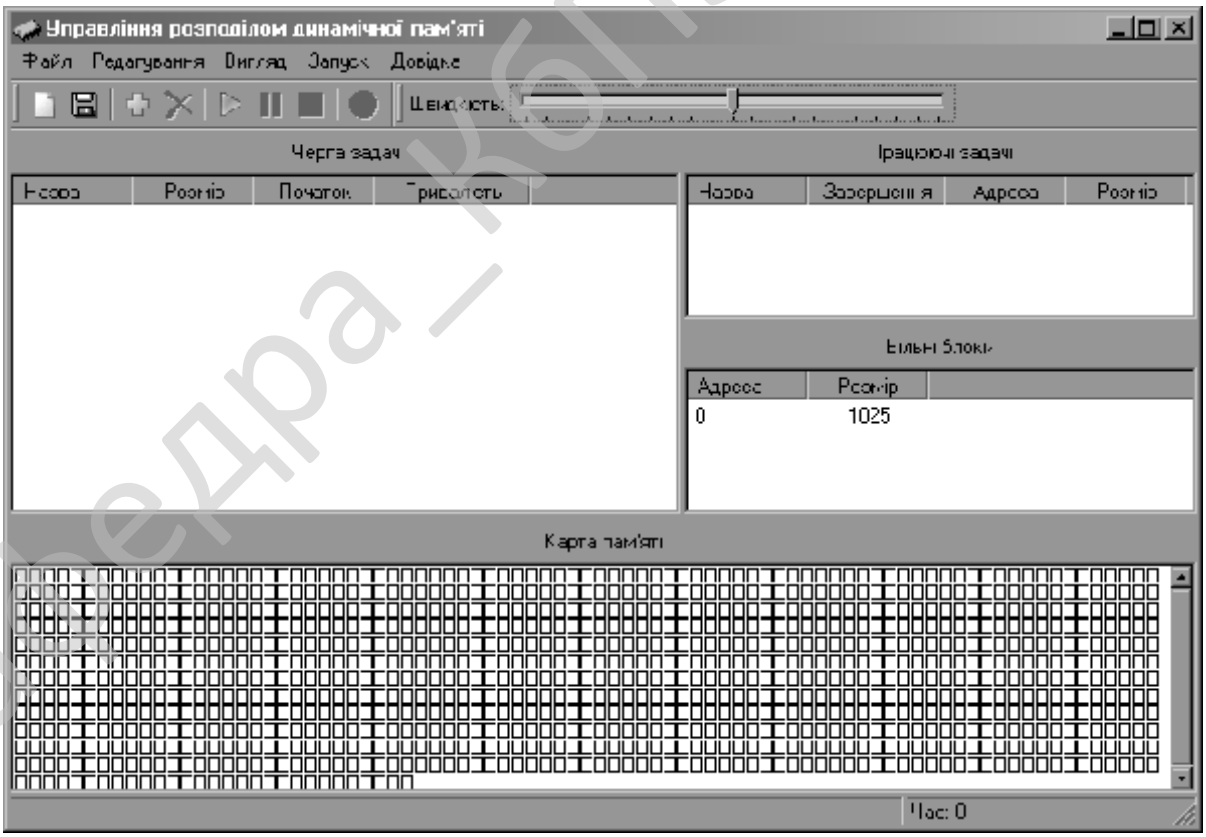

Рисунок 5.3 – Головне вікно програми після створення проекту

Потім треба створити задачі, які потребують доступу до пам'яті, за домомогою пункту меню **Редагувати->Додати задачу**, або за допомогою кнопки, що изображена у вигляді плюса – .

В діалоговому вікні, що з'явиться на екрані, необхідно вкказати назву задачі, її розмір, час виконання, та час запуску (рисунок 5.4).

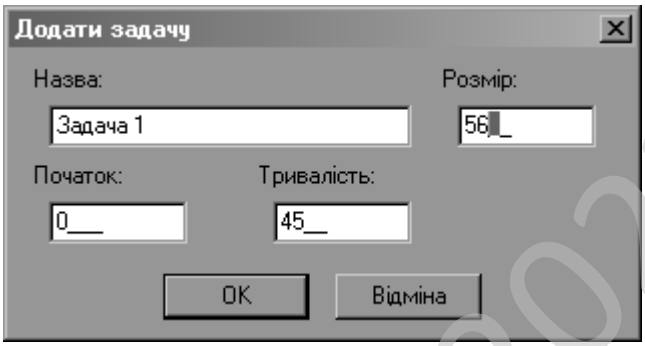

Рисунок 5.4 – Діалогове вікно додавання задач

Тепер після додавання задач у головному вікні програми всі задачі будуть відображені у черзі (рисунок 5.5).

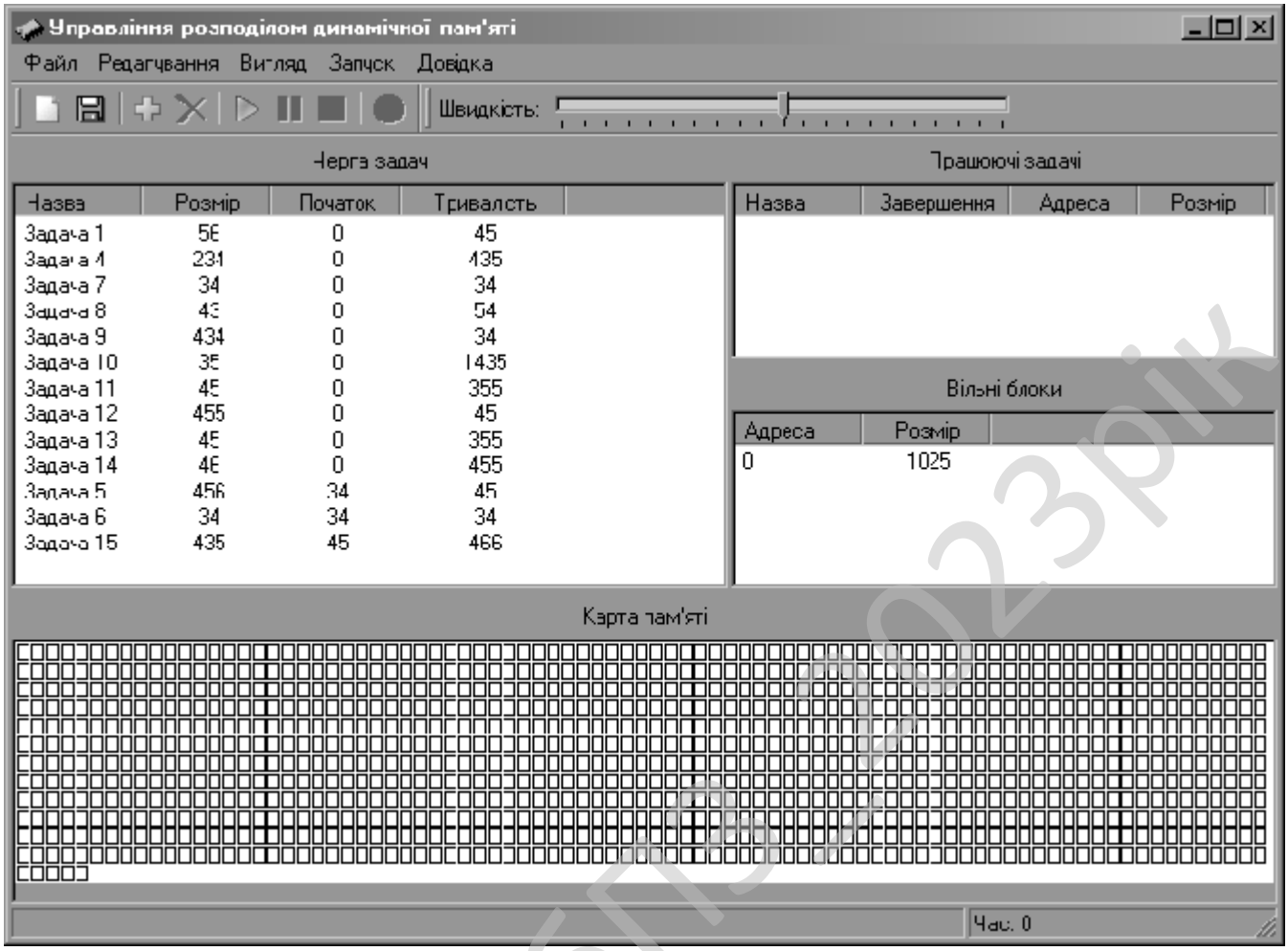

Рисунок 5.5 – Головне вікно програми після заповнення черги задач

Таким чином можна заповнити чергу задач, будь-якою кількістю задач, та з будь-якими параметрами часу та розміру, що не перевищують обмеження накладене розміром наявної пам'яті.

На даному етапі все готове для запуску процесу емуляції, достатньо лише вибрати пункт меню **Запуск->Старт**, або натиснути кнопку ▶.

Стан системи буде відображатися у вигляді карти пам'яті, що показує розташування вільних і зайнятих блоків пам'яті. Вільні блоки – білого кольру, зайняті – зафарбовані рожевим (рисунок 5.6).

Також програма веде статистику по загальній кількості відмов по пам'яті й відображає відсоток використання пам'яті.

В будь-який момент можна зупинити процес, або поставити на паузу.

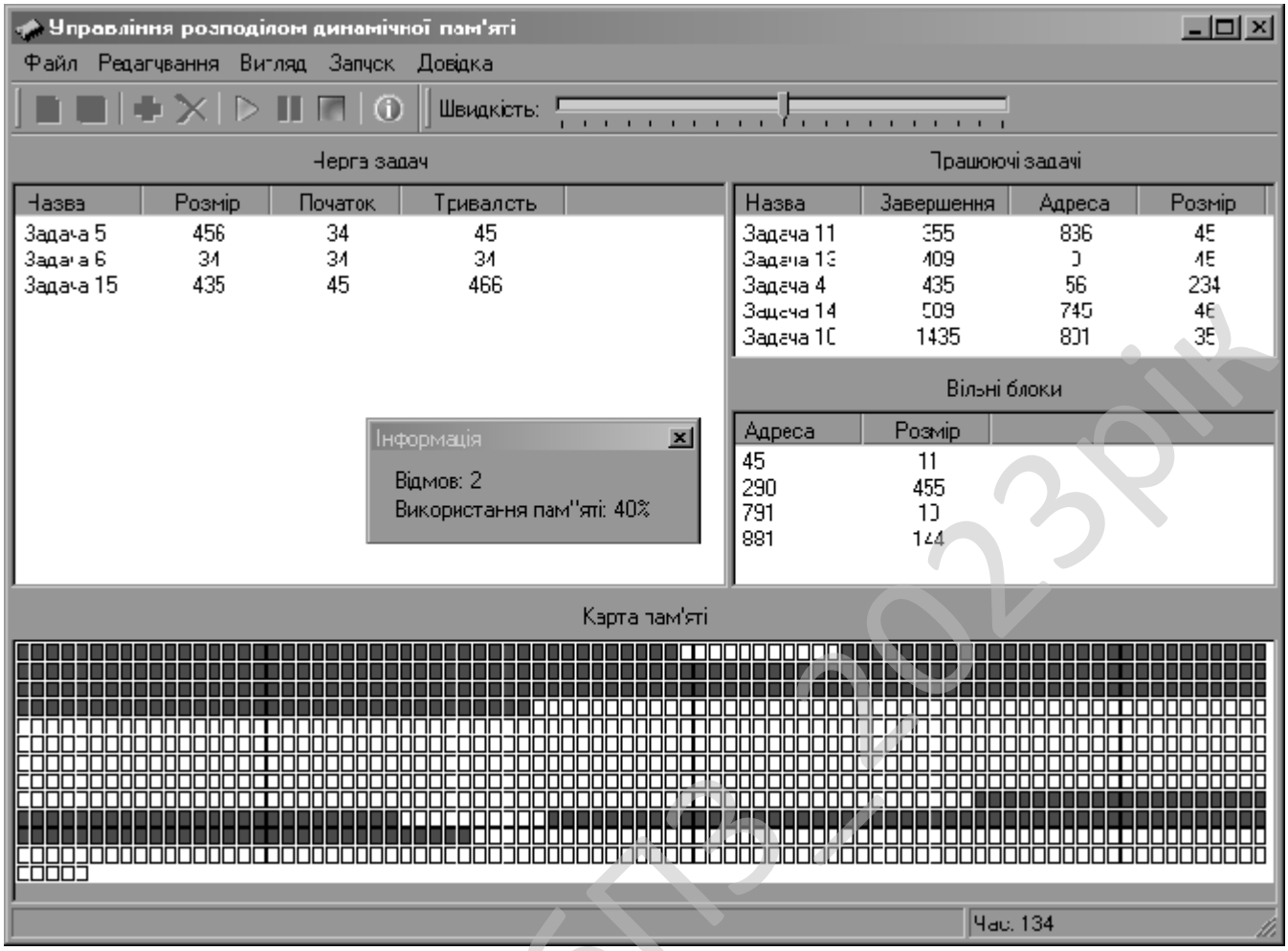

Рисунок 5.6 – Головне вікно програми під час процесу емуляції розподілу динамічної пам'яті

Є можливість регулювання швидкості перемикання станів системи а також редагування списку задач із можливістю їхнього наступного збереження у текстовий файл і відкриття вже підготовленого заздалегідь набору (рисунок 5.7).

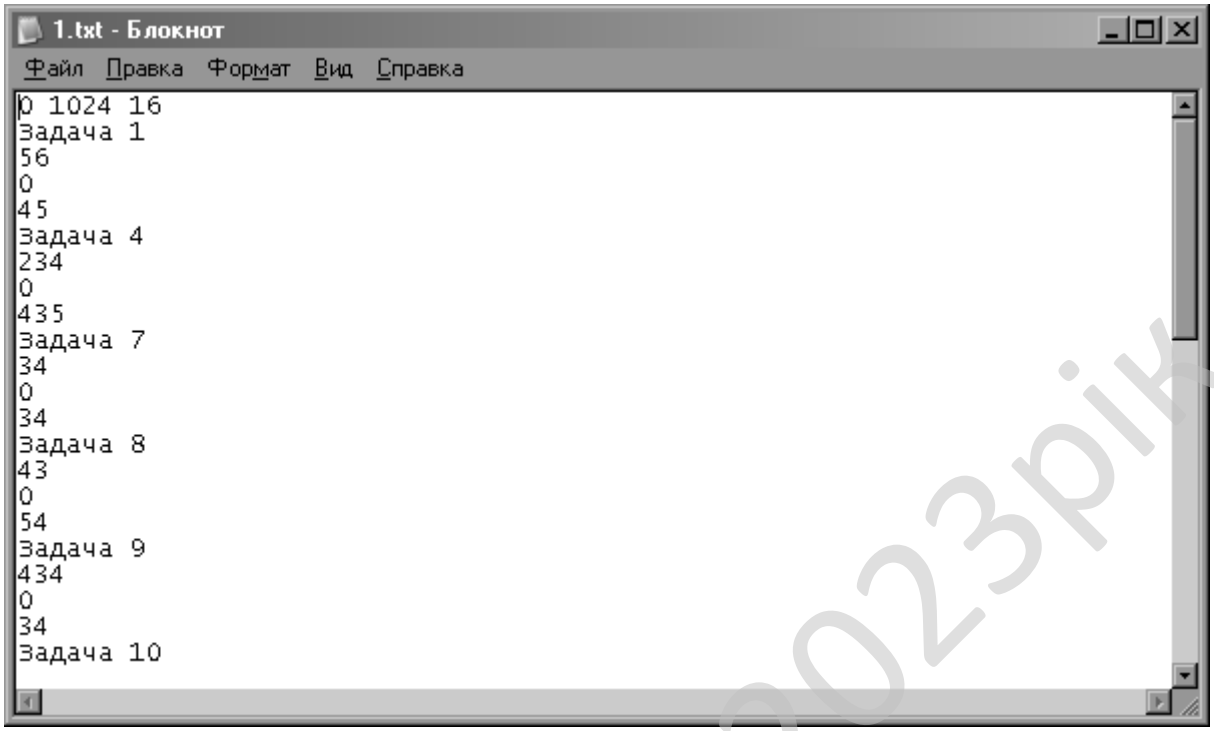

Рисунок 5.7 – Формат збереження черги задач

Також для полегшення роботи з програмою, створено генератор тестів, що сам заповнює .txt файл чергою задач випадковим чином (рисунок 5.8). Це дозволяє значно скоротити час створення проекту та безпосередньо перейти до процесу емуляції.

Рисунок 5.8 – Генератор тестів для основної програми

Вибрав пункт меню **Довідка->Про програму…** , можна переглянути інформацію про призначення програми та ім’я автора (рисунок 5.9).

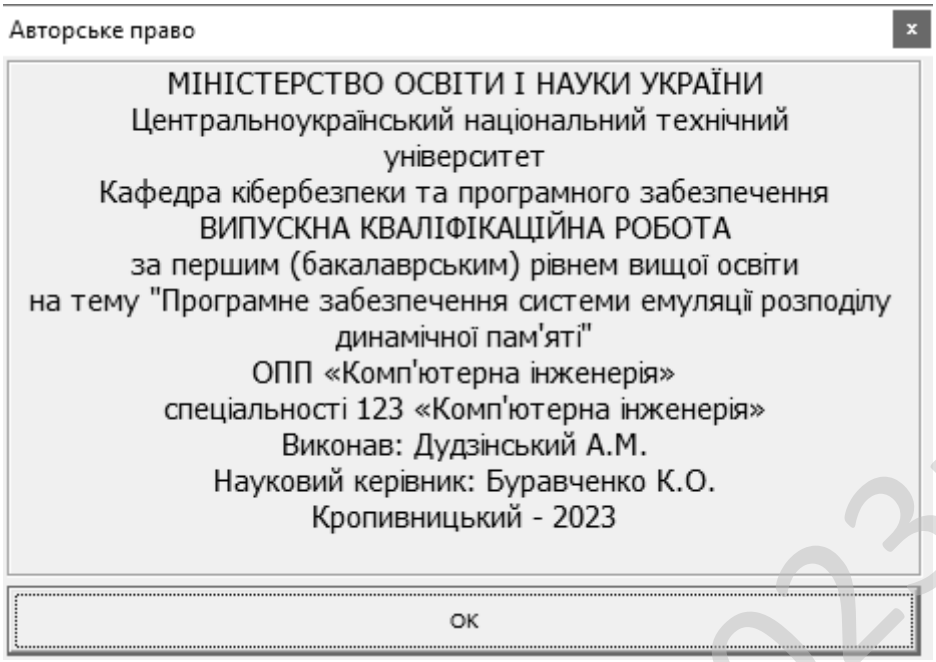

Рисунок 5.9 – Вікно "Про програму…"

6 ОСНОВНІ ВИСНОВКИ

Програмне забезпечення, створене в результаті виконання випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти, призначено для системи емуляції розподілу динамічної пам’яті.

В межах України в недостатній мірі представлені вітчизняні розробки в цій області.

Рішення завдання полягало у вирішенні наступних задач:

– Був проведений огляд існуючих систем емуляції розподілу динамічної пам’яті.

– Досліджена система емуляції розподілу динамічної пам’яті.

– На основі отриманих результатів досліджень створена програмна реалізація системи емуляції розподілу динамічної пам’яті.

Розроблені під час виконання випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти алгоритми дозволяють успішно вирішувати завдання емуляції розподілу динамічної пам’яті.

Розроблене програмне забезпечення має простий, дружній та зручний інтерфейс користувача, що забезпечує легкість у освоєнні роботи програмного продукту, зручність у використанні, і не потребує особливих спеціальних знань.

При створені програмного забезпечення було використано об’єктно-орієнтований підхід, що відповідає сучасним тенденціям у галузі розробки комерційних програмних систем.

Програма реалізована на мові високого рівня Delphi 10.4 Sydney. Дана мова програмування дозволяє найбільш ефективно обробляти дані призначені для системи емуляції розподілу динамічної пам’яті. Це дозволило мінімізувати строк розробки програмного забезпечення, і, як слід, зменшити витрати на його розробку. Запропоноване програмне забезпечення ділиться на загальне програмне забезпечення, що поставляється із засобами обчислювальної техніки й спеціальне

програмне забезпечення, що спеціально розроблене для даної конкретної системи й включає програми, що реалізують її функції.

Програма призначена для виконання під управлінням багатозадачної операційної системи Windows 10/11.

Даються необхідні рекомендації з установки розробленого програмного забезпечення.

Для підвищення рівня безпеки запропоновано застосовувати алгоритм DES.

В цілому створене програмне забезпечення підтверджує правильність використаних проектних рішень та повністю відповідає вимогам технічного завдання. Створене програмне забезпечення має потенційну можливість для подальшого вдосконалення і застосування у різних галузях.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Андреева А. Ю. Кошелев К.Б. Операционные системы. Учебное пособие по курсу СУСПО. – Барнаул, АлтГТУ, 2002. (10 экз)

2. Барский А.Б. Параллельные процессы в вычислительных системах. Планирование и организация. – М.: Радио и связь, 1983.

3. Вильям Столлингс. Операционные системы: Внутренняя организация и принципы проектирования. — М.: «Вильямс», 2004. — 848 с.

4. Головкин Б.А. Параллельные вычислительные системы. М.: Наука, 1980. – 520 с.

5. Гук М. Аппаратные средства IBM PC Издательство «ПитерКом» С.-П. 1999г.

6. Джесхоуп Х.Л. Параллельные ЭВМ: Архитектура, программирование и алгоритмы.– М.: Радио и связь, 1986.

7. Жедицький В.Ц. Охорона праці користувачів комп'ютерів: Навч. посібник.-2-ге вид. поп.– Львів: Афіша, 2000.-176 с.

8. Корнеев В.В. Параллельные вычислительные системы. М.: Москва, 1999.

9. Михайлов Н.Л. Основы построения операционных систем. Учебное пособие. Рыбинск, РГАТА, 2000

10. Столлингс В. Операционные системы. Внутреннее устройство и принципы проектирования.-М.Ж Вильямс, 2002. – 844 с. (50 экз.)

11. Таненбаум Э. Современные операционные системы. 2-е изд. – СПб.: Питер, 2002. – 1040 с.

12. Харт Д. Системное программирование в Win32. М., Диалектика, 2003

13. Яковлев Ю.С., Тихонов Б.М. О распределении памяти в компьютерных системах // Управляющие системы и машины. – 2006. – № 5. – С. 40–47.

14. Kovalenko O., Popereshnyak S., Grinenko S., Grinenko O., Radivilova T. «Methods for Assessing the Maturity Levels of Software Ecosystems». *CEUR Workshop Proceedings* Volume 2654, 2019, Pages 251-261. Режим доступу: https://www.scopus.com/record/display.uri?eid=2-s2.0-85083214331&origin=AuthorNamesList&txGid=5633fba897776a6e0f3d5633fbcd3fbe **(Scopus)**.

15. Kovalenko O., Drieieva H., Simakhin V., Bondar S., Drieiev O., Zhumadilova M. «Multifractal Properties of Traffic Generator Based on Markov Chains ». *CEUR Workshop Proceedings* Volume 2588, 2019, Pages 567-579. Режим доступу: https://www.scopus.com/record/display.uri?eid=2-s2.0-85083214331&origin=AuthorNamesList&txGid=176e2cada8976a6e0f3d5633fbcd3fbe **(Scopus)**.

16. Kovalenko Oleksandr Qualitative risk analysis of software development / Oleksandr Kovalenko, Jamil Al-Azzeh, Oleksii Smirnov, Anna Kovalenko, Serhii Smirnov // Asian Journal of Information Technology. – Volume 17 Issue 3. – Medwell Journals. – 2018. – P. 218-230. ISSN: 1682-3915. URL: http://medwelljournals.com/abstract/?doi=ajit.2018.218.230 Doi: ajit.2018.218.230

17. Kovalenko Oleksandr, The mathematical model of the testing technology for DOM XSS vulnerabilities / O. Kovalenko, O. Smirnov, A.Kovalenko, S. Smirnov, V. Vialkova // Scientific & practical cyber security journal (SPCSJ) Volume 2 Issue 1, P. 22-28. Georgia. Tbilisi. Scientific Cyber Security Association (SCSA), 2018 ISSN: 2587-4667. URL: https://journal.scsa.ge/wp-content/uploads/2018/12/04-3-o.kovalenko-a.kovalenko-o.smirnov-s.smirnov-v.vialkova.pdf

18. Коваленко А.В. Технология тестирования DOM XSS уязвимости / А.В. Коваленко, А.С. Коваленко, А.А. Смирнов, С.А. Смирнов // Scientific & practical cyber security journal (SPCSJ) Volume 1. Issue 1. P. 38-45 Georgia. Tbilisi. Scientific Cyber Security Association (SCSA), 2017 ISSN: 2587-4667. URL: https://journal.scsa.ge/wp-content/uploads/2018/12/8-dom-xss-testing-technology-vulnerabilities.pdf

19. Коваленко О.В. Моделі та методи розроблення програмного

забезпечення комп'ютерних систем для підвищення безпеки даних: монографія / О.В. Коваленко // К.: Вид. «КОД» – 2019. – 305 с.

20. Коваленко А.В. Методы качественного анализа и количественной оценки рисков разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов // Информационные технологии в управлении, образовании, науке и промышленности: монография / Под редакцией профессора В.С. Пономаренко. – Х.: Видавець Рожко С.Г., 2016. – 566 с.

21. Коваленко А.В. Разработка метода управления рисками разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов // Інформаційні технології: проблеми та перспективи: монографія / За загальною редакцією В.С. Пономаренка. – Х.: Видавець Рожко С.Г., 2017. – 447 с.

22. Коваленко А.В. Комплекс математических моделей технологии тестирования web-приложений / А.В. Коваленко, А.А. Смирнов // Інформаційні технології: сучасний стан та перспективи: монографія / За загальною редакцією В.С. Пономаренка. – Х.: ТОВ «ДІСА ПЛЮС», 2018. – 461 с.

23. Коваленко А.В. Задачи распознавания ситуаций в ERP системах / А.В. Коваленко, А.А. Смирнов, А.С. Коваленко // Збірник наукових праць "Системи обробки інформації". – Випуск 4(120). – Х.: ХУПС – 2014. – С. 161-164.

24. Коваленко А.В. Методы качественного анализа и количественной оценки рисков разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов // Збірник наукових праць "Системи обробки інформації". – Випуск 5(142). – Х.: ХУПС – 2016. – С. 153-157.

25. Коваленко А.В. Проблемы анализа и оценки рисков информационной деятельности / А.В. Коваленко, А.А. Смирнов, Н.Н. Якименко, А.П. Доренский // Збірник наукових праць "Системи обробки інформації". – Випуск 3(140). – Х.: ХУПС – 2016. – С. 40-42.

26. Коваленко А.В. Метод качественного анализа рисков разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов, Н.Н. Якименко, А.П.

Доренский // Наука і техніка Повітряних Сил Збройних Сил України. – Випуск 2(23). – Харків: ХУПС. – 2016. – С. 150-158.

27. Коваленко А.В. Метод количественной оценки рисков разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов, Н.Н. Якименко, А.П. Доренский // Збірник наукових праць Харківського університету Повітряних Сил. Випуск 2 (47). – Харків: ХУПС. – 2016. – С. 128-133.

28. Коваленко А.В. Использование псевдобулевых методов бивалентного программирования для управления рисками разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов // Системи управління, навігації та зв'язку. – Випуск 1 (37). – Полтава: ПолтНТУ. – 2016. – С. 98-103.

29. Коваленко А.В. Метод управления рисками разработки программного обеспечения / А.В. Коваленко // Системи управління, навігації та зв'язку. – Випуск 2 (38). – Полтава: ПолтНТУ. – 2016. – С. 93-100.

30. Коваленко А.В. Технология тестирования уязвимости к SQL инъекциям / А.В. Коваленко // Системи управління, навігації та зв'язку. – Випуск 5 (45). – Полтава: ПолтНТУ. – 2017. – С. 66-71.

31. Коваленко А.В. Масштабирование имитационной модели технологии тестирования безопасности / А.В. Коваленко // Системи управління, навігації та зв'язку. – Випуск 6 (46). – Полтава: ПолтНТУ. – 2017. – С. 181-184.

32. Коваленко А.В. Имитационная модель технологии тестирования безопасности Web-приложений / А.В. Коваленко // Системи управління, навігації та зв'язку. – Випуск 1 (47). – Полтава: ПолтНТУ. – 2018. – С. 114-123.

33. Коваленко О.В. Методи якісного аналізу та кількісної оцінки ризиків розроблення програмного забезпечення/ О.В. Коваленко // Системи управління, навігації та зв'язку. – Випуск 3 (49). – Полтава: ПолтНТУ. – 2018. – С. 116-125.

34. Коваленко О.В. Управління ризиками розроблення програмного забезпечення за умови обмеженості коштів виділених на усунення помилок безпеки/ О.В. Коваленко // Техніка в сільськогосподарському виробництві, галузеве машинобудування, автоматизація. – Випуск 31. – Кропивницький:

ЦНТУ. – 2018. – С. 128-140.

35. Коваленко О.В. GERT-мережевий синтез технології тестування на вразливість WEB-додатків/ О.В. Коваленко // Захист інформації. – Випуск 20(2) – К.: НАУ. – 2018. – С. 89-94.

36. Коваленко О.В. Імітаційна модель технології тестування безпеки на основі положень теорії масштабування / О.В. Коваленко // Безпека інформації. – Випуск 24 (2). – К.: НАУ. – 2018. – С. 110-117.

37. Коваленко О.В. Оцінка ефективності технології тестування безпеки / О.В. Коваленко // Вчені записки Таврійського національного університету імені В.І. Вернадського. Серія: Технічні науки. Том 29 (68) № 2, 2018. – С. 137-141

38. Коваленко О.В. Методи та засоби управління безпекою додатків / О.В. Коваленко // Інформаційно-керуючі системи на залізничному транспорті. №4, 2018. – С. 41-44.

39. Коваленко О.В. Розробка інформаційної технології передтестової компіляції та розподілу доступу / О.В. Коваленко // Системи управління, навігації та зв'язку. – Випуск 4 (50). – Полтава: ПолтНТУ. – 2018. – С. 115-119.

40. Коваленко О.В. Аналіз та дослідження інформаційних технологій розробки програмного забезпечення/ О.В. Коваленко // Вчені записки Таврійського національного університету імені В.І. Вернадського. Серія: Технічні науки. Том 29 (68) № 5, 2018. – С. 131-137.

41. Коваленко О.В. Удосконалений метод управління ризиками розроблення програмного забезпечення на основі напівмарковської моделі прийняття рішень/ О.В. Коваленко // Сучасні інформаційні системи. – Випуск 2(3). – Харків. – 2018. – С. 41-48.

42. Коваленко О.В. Математичні моделі технології тестування DOM XSS вразливості та вразливості до SQL ін'єкцій / О.В. Коваленко // Вісник Черкаського державного технологічного університету. Серія : Технічні науки №4, 2018. – С. 29-36.

43. Коваленко О.В. Математична модель технології тестування вразливості до SQL ін'єкцій / О.В. Коваленко // Системи управління, навігації та зв'язку. – Випуск 6 (58). – Полтава: ПолтНТУ. – 2019. – С. 43-47.

44. Коваленко О.В. Математична модель технології тестування комплексу DOM XSS вразливостей для аналітичної оцінки часових витрат / О.В. Коваленко // Центральноукраїнський науковий вісник. Технічні науки. № 2(33). с. 173-180, 2019.

45. Коваленко А.В. Проблемы анализа и оценки рисков информационной деятельности / А.В. Коваленко, А.А. Смирнов // Збірник наукових праць II міжнародної науково-практичної конференції «Актуальні питання забезпечення кібернетичної безпеки та захисту інформації». м. Київ. 24-27 лютого 2016 р. – Київ: Європейський університет. – 2016. – С. 138-139.

46. Коваленко А.В. Анализ и оценка рисков разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов // Збірник тез «Securitea informationala 2015-2016». Conferenta internationala (editia a XII-a). Chisinau. Moldova. 3 martie 2016. – Chisinau: ADSEM. – 2016. – P. 96-102.

47. Коваленко А.В. Исследование источников и причин риска разработки программного обеспечения, этапов и работ, при выполнении которых возникает риск / А.В. Коваленко, А.А. Смирнов // Збірник тез VII всеукраїнської науково-практичної конференції "Інформатика та системні науки (ІСН-2016)". м. Полтава. 10-12 березня 2016 р. – Полтава.: ПУЕТ – 2016. – С. 264-266.

48. Коваленко А.В. Оценка показателя чистой приведенной стоимости для количественной оценки рисков проекта разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов // Збірник тез науково-практичної конференції "Проблеми кібербезпеки інформаційно-телекомунікаційних систем". м. Київ. 10-11 березня 2016 р. – Київ: КНУ ім. Тараса Шевченко – 2016. – С. 81-82.

49. Коваленко А.В. Методика структурной идентификации рисков

разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов // Збірник тез Міжнародної науково-практичної конференції «Інформаційна безпека та комп'ютерні технології» (IS&CT). м. Кіровоград. 24-25 березня 2016 р. – Кіровоград: КНТУ. – 2016. – С. 71-72.

50. Коваленко А.В. Методы качественного анализа рисков разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов // Збірник тез першої міжнародної науково-практичної конференції «Проблеми науково-технічного та правового забезпечення кібербезпеки у сучасному світі» (ПНПЗК-2016). м. Харків. 30 березня – 1 квітня 2016 р. – Харків: НТУ «ХПІ». – 2016. – С. 6-7.

51. Коваленко А.В. Структурная идентификация рисков разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов // Збірник тез XVIII міжнародного науково-практичного семінару «Комбінаторні конфігурації та їх застосування». м. Кіровоград. 15-16 квітня 2016 р. – Кіровоград: КНТУ. – 2016. – С. 175-182.

52. Коваленко А.В. Исследование разработанной методики структурной идентификации рисков разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов // Збірник тез VIII міжнародної науково-практичної конференції “Проблеми і перспективи розвитку IT-індустрії”. м. Харків. 28-29 квітня 2016 р. – Харків: ХНЕУ. – 2016. – С. 49.

53. Коваленко А.В. Исследование дерева рисков разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов // Збірник тез III міжнародної науково-практичної конференції «Інформаційна та економічна безпека» (INFECO-2016)». м. Харків. 28-30 квітня 2016 р. – Харків: ХНМІ ДВНЗ «УБС». – 2016. – С. 174-178.

54. Коваленко А.В. Методы качественного анализа и количественной оценки рисков разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов, А.С. Коваленко // Сборник тезисов XII международной конференции "Стратегия качества в промышленности и образовании". г. Варна.

Болгария. 30 мая – 02 июня 2016 г – Варна. ТУВ. – 2016. – С. 585-589.

55. Коваленко А.В. Разработка метода управления рисками разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов, А.С. Коваленко // Матеріали Всеукраїнської науково-практичної конференції «Кібербезпека в Україні: правові та організаційні питання». м. Одеса, 21 жовтня 2016 р. – Одеса : ОДУВС, 2016. – С.146-148.

56. Коваленко А.В. Метод управления рисками разработки программного обеспечения с использованием псевдобулевых методов бивалентного программирования / А.В. Коваленко, А.А. Смирнов // Матеріали Всеукраїнської науково-практичної конференції «Актуальні задачі та досягнення у галузі кібербезпеки». м. Кропивницький, 23-25 листопада 2016 року – Кропивницький: ЦНТУ, 2016. – С. 162.

57. Коваленко А.В. Псевдобулевые методы бивалентного программирования для управления рисками разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов, А.С. Коваленко, С.А. Смирнов // Збірник наукових праць III міжнародної науково-практичної конференції «Актуальні питання забезпечення кібернетичної безпеки та захисту інформації». м. Київ. 22-25 лютого 2017 р. – Київ: Європейський університет. – 2017. – С. 158-162.

58. Коваленко А.В. Метод управления рисками разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов // Збірник тез II науково-практичної конференції “Проблеми кібербезпеки інформаційно-телекомунікаційних систем”. м. Київ. 23-24 березня 2017 р. – Київ: КНУ ім. Тараса Шевченко – 2017. – С. 203-205.

59. Коваленко А.В. Алгоритмы анализа уязвимостей при управлении рисками разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов, А.С. Коваленко // Conferenta internationala (editia a XIII-a). «Securitea informationala 2017». Chisinau. Republic of Moldova. 4-5 aprilie 2017. – Chisinau: ADSEM. – 2017. – P. 19-22.

60. Коваленко А.В. Алгоритм анализа DOM XSS уязвимости при управлении рисками разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов, А.С. Коваленко // Збірник тез дев'ятнадцятого міжнародного науково-практичного семінару «Комбінаторні конфігурації та їх застосування». м. Кропивницький 7-8 квітня 2017 р. – Кропивницький: ГЛА НАУ. – 2017. – С. 125-127.

61. Коваленко А.В. Алгоритм анализа уязвимости SQL Injection для управления рисками разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов, А.С. Коваленко // Збірник тез другої міжнародної науково-технічної конференції «Проблеми науково-технічного та правового забезпечення кібербезпеки у сучасному світі» (ПНПЗК-2017). м. Харків.10-12 квітня 2017 р. – Харків: НТУ «ХПІ». – 2017. – С. 27.

62. Коваленко А.В. Метод управления рисками разработки программного обеспечения на основе алгоритмов анализа уязвимостей / А.В. Коваленко, А.А. Смирнов, А.С. Коваленко // Збірник тез Міжнародної науково-практичної конференції «Інформаційна безпека та комп'ютерні технології» (IS&CT). м. Кіровоград. 20-22 квітня 2017 р. Кіровоград: КНТУ. – 2017. – С. 92.

63. Коваленко А.В. Алгоритмы анализа DOM XSS уязвимости и уязвимости SQL Injection при управлении рисками разработки программного обеспечения / А.В. Коваленко, А.А. Смирнов, А.С. Коваленко // Збірник тез IX міжнародної науково-практичної конференції "Проблеми і перспективи розвитку IT-індустрії". м. Харків. 20-21 квітня 2017 р. – Харків: ХНЕУ. – 2017. – С. 61.

64. Kovalenko O.V. Method of testing the dom xss vulnerability / Kovalenko Oleksandr, Kovalenko Anna, Smirnov Oleksii, Smirnov Serhii // International Conference «information technologies, systems and networks ITSN-2017». Chisinau, Republic of Moldova. 17 – 18 October 2017. – Chisinau: Academy of Sciences of Moldova, Military Academy of Armed Forces "Alexandru cel Bun". – 2017. – P. 7.

65. Коваленко О.В. Метод тестування DOM XSS уразливості / О.В. Коваленко, О.А. Смірнов, А.С. Коваленко, С.А. Смірнов // Збірник тез всеукраїнської науково-практичної інтернет-конференції «Автоматика та комп'ютерно-інтегровані технології у промисловості, телекомунікаціях, енергетиці та транспорті». м. Кропивницький. 16-17 листопада 2017 р. – Кропивницький: ЦНТУ. – 2017. – С. 198-199.

66. Коваленко О.В. GERT-модель технології тестування DOM XSS уразливості / О.В. Коваленко, А.С. Коваленко, О.А. Смірнов, С.А. Смірнов // Збірник наукових праць IV міжнародної науково-практичної конференції «Актуальні питання забезпечення кібернетичної безпеки та захисту інформації». м. Київ. 21-24 лютого 2018 р. – Київ: Європейський університет. – 2018. – С. 65-70.

67. Коваленко О.В. Технології тестування уразливостей Web-застосунків з використанням GERT-моделі / О.В. Коваленко, А.С. Коваленко, О.А. Смірнов, С.А. Смірнов // Збірник тез всеукраїнської науково-практичної конференції "Комп'ютерні інтелектуальні системи та мережі (КІСМ-2018)". м. Кривий Ріг. 21-23 березня 2018 р. – Кривий Ріг.: ДВНЗ КНУ – 2018. – С. 227-230.

68. Коваленко А.В. Тестирование уязвимости Web-приложений к атаке вида межсайтовый скриптинг / А.В. Коваленко, А.С. Коваленко, А.А. Смирнов, С.А. Смирнов // Збірник тез «Securitea informationala 2018». Conferenta internationala (editia a XIV-a). Chisinau. Moldova. 20-21 martie 2018. – Chisinau: ADSEM. – 2018. – P. 54-56.

69. Коваленко А.В. Комплекс математических моделей технологии тестирования web-приложений / А.В. Коваленко, А.С. Коваленко, А.А. Смирнов, С.А. Смирнов // Збірник тез X міжнародної науково-практичної конференції "Проблеми і перспективи розвитку IT-індустрії". м. Харків. 19-20 квітня 2018 р. – Харків: ХНЕУ. – 2018. – С. 38.

70. Коваленко О.В. Розробка методу передтестової компіляції й розподілу доступу / О.В. Коваленко, А.С. Коваленко, О.А. Смірнов,

С.А. Смірнов // Збірник наукових праць III міжнародної науково-практичної конференції “Інформаційна безпека та комп’ютерні технології”, м. Кропивницькй. 19-20 квітня 2018 р. – Кропивницький: ЦНТУ. – 2018. – С. 214-215.

71. Коваленко О.В. Оцінка ефективності технологій тестування безпеки уразливостей DOM XSS й SQL-ін'єкцій / О.В. Коваленко, А.С. Коваленко, О.А. Смірнов, С.А. Смірнов // Сборник тезисов XIV международной конференции "Стратегия качества в промышленности и образовании", Варна, Болгария. 04-07 июня 2018 г – Варна. ТУВ. – 2018. – С. 271-274.

72. Коваленко О.В. Аналіз основних підходів математичного моделювання та методологій для забезпечення максимальних показників безпеки програмного забезпечення / О.В. Коваленко, А.С. Коваленко // Збірник наукових праць всеукр. наук.-практ. конф. здобувачів вищої освіти й молодих учених «Комп’ютерна інженерія і кібербезпека : досягнення та інновації, м. Кропивницькй. 27-29 листопада

Додаток А
(обов'язковий)

Технічне завдання

Зміст

1 Найменування та область застосування

Це технічне завдання розповсюджується на розробку системи емуляції розподілу динамічної пам’яті.

2 Підстава для розробки

Підставою для розробки служить завдання на випускну кваліфікаційну роботу за першим (бакалаврським) рівнем вищої освіти, видане на кафедрі кібербезпеки та програмного забезпечення (нак. № 8-02 від 5.01.2023 року).

3 Мета та призначення розробки

Метою випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти є розробка програмного забезпечення системи емуляції розподілу динамічної пам’яті.

4 Джерела розробки

Джерелом цієї випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти є стосовна до теми література і існуючі аналоги.

5 Технічні вимоги

5.1 Склад продукції

Складниками розробки є:

– вибір і обґрунтування методів реалізації проекту;

– розробка програмної частин системи, а також розробка взаємодії системи з ОС та з користувачем;

– розробка програми, що реалізує спроектовані алгоритми роботи системи.

5.2 Показники призначення

Система повинна забезпечувати:

– системи емуляції розподілу динамічної пам’яті;

– цілісність даних у процесі роботи та при зберіганні;

– простий, інтуїтивно зрозумілий інтерфейс.

5.3 Вимоги до функціональних характеристик

Розроблене програмне забезпечення не повинно мати обмежень на версію драйверів та операційної системи.

5.4 Вимоги до архітектури

Компонент, що розробляється повинен використовувати системні засоби та апаратні засоби, що на даному етапі розвитку обчислювальної техніки найбільше поширені.

5.5 Вимоги до надійності

Програмні модулі написані по всім правилам, які стосуються стандартних викликів процедур, функцій, методів і форм, визначених технічною документацією на середовище розробки.

5.6 Умови експлуатації

Робочі місця користувачів ПЗ повинні задовольняти наступним умовам експлуатації:

– температура повітря: 19-20 град. по Цельсію;

– відносна вологість повітря до 80%;

– атмосферний тиск 107 кПа.

5.7 Вимоги до складу та параметрів технічних засобів

Програмне забезпечення повинно бути реалізоване на ПЕОМ архітектури IBM PC, працювати в ОС Windows 10/11 і з сумісними з цією платформою пристроями і прикладним програмним забезпеченням.

5.8 Вимоги до інформаційної і програмної сумісності

Переносність програмного забезпечення повинна бути забезпечена за рахунок його реалізації стандартного інтерфейсу взаємодії з ОС, що працюють під управлінням ОС Windows 10/11.

5.8.1 Обладнання

Комп'ютер Intel® Celeron/8 Mb/1.2 Gb/SVGA 14" 1Mb або сумісні з ним.

5.8.2 Мова програмування

Середовище Delphi 10.4 Sydney.

5.8.3 Вхідні дані

Опис алгоритму роботи запропонованої системи.

5.8.4 Вихідні дані

Робоча програма.

6 Вимоги до програмної документації

Програмна продукція повинна бути представлена у виді опису структури даних, схем та опису алгоритму, а також текстів вихідних модулів програмного забезпечення згідно ЄСПД .

7 Перелік документів, що розробляються

– Структурна схема системи – 1 аркуш.
– Функціональна схема системи – 1 аркуш.
– Діаграма процесів – 1 аркуш.
– Блок-схема алгоритму роботи програми – 2 аркуша.
– Пояснювальна записка – 70 аркушів.

8 Етапи розробки

8.1 Збір і обробка інформації по темі випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти. Постановка задачі на виконання випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти (складання ТЗ).

8.2 Проведення досліджень або експериментальних робіт для уточнення основних положень випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти.

8.3 Розробка функціональних схем, блок схем алгоритмів роботи програмного забезпечення.

8.4 Побудова схем взаємодії даних.

8.5 Створення прототипу ПЗ.

8.6 Віднаходження ПЗ, аналіз отриманих результатів.

8.7 Оформлення пояснювальної записки і виконання робіт по графічній частині.

9 Порядок контролю та приймання

9.1 Подання випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти на попередній захист 23.05.2023 р.

9.2 Подання випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти на захист 8.06.2023 р.

Додаток Б

(обов’язковий)

Міністерство освіти і науки України

Центральноукраїнський національний технічний університет

ЗАТВЕРДЖУЮ

Керівник випускної кваліфікаційної роботи за

першим (бакалаврським) рівнем вищої освіти

_____ Буравченко К.О.

Програмне забезпечення системи емуляції розподілу динамічної пам’яті

Лістинг програми

Код документу 12

Носій: CD/DVD-диск / USB-флеш-накопичувач

Загальна кількість аркушів: 20

Літера: РП

Кропивницький – 2023 року

Основна програма

Файл uMain.pas основної програми

```
unit uMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Menus, ComCtrls, ToolWin, ExtCtrls, StdCtrls, ImgList, ActnList;

type
  TMemUnit = LongWord;

  PFreeMemRec = ^TFreeMemRec;
  TFreeMemRec = record
    Next: PFreeMemRec;
    Address, Size: TMemUnit;
  end;

  TTaskQueueRec = record
    Name, Size, Beginning, Duration: String;
  end;

  TfmMain = class(TForm)
    mmMain: TMainMenu;
    miFile: TMenuItem;
    miExit: TMenuItem;
    stbMain: TStatusBar;
    miHelp: TMenuItem;
    miAbout: TMenuItem;
    cbrMain: TCoolBar;
    tbMain: TToolBar;
    tbtCreate: TToolButton;
    pnMemoryMap: TPanel;
    pnMemoryMapHeader: TPanel;
    scbMemoryMap: TScrollBox;
    Splitter1: TSplitter;
    pnSpeed: TPanel;
    laSpeed: TLabel;
    trbSpeed: TTrackBar;
    pnTopRight: TPanel;
    Splitter2: TSplitter;
    pnTaskQueue: TPanel;
    pnTaskQueueHeader: TPanel;
    lvTaskQueue: TListView;
    pnFreeMemList: TPanel;
    Splitter3: TSplitter;
    pnFreeMemListHeader: TPanel;
    lvFreeMemList: TListView;
    pnWorkingTasks: TPanel;
    pnWorkingTasksHeader: TPanel;
    lvWorkingTasks: TListView;
    imgMemoryMap: TImage;
    ilMain: TImageList;
    alMain: TActionList;
    actCreate: TAction;
    miSeparator1: TMenuItem;
    miCreate: TMenuItem;
    tbtSave: TToolButton;
    tbtSeparator1: TToolButton;
    actOpen: TAction;
    actSave: TAction;
    miEdit: TMenuItem;
    actAddTask: TAction;
```

```
    actDeleteTask: TAction;
    miAddTask: TMenuItem;
    miDeleteTask: TMenuItem;
    tbtAddTask: TToolButton;
    tbtDeleteTask: TToolButton;
    tbtSeparator2: TToolButton;
    miRun: TMenuItem;
    actStart: TAction;
    tbtStart: TToolButton;
    actPause: TAction;
    actStop: TAction;
    tbtPause: TToolButton;
    tbtStop: TToolButton;
    miStart: TMenuItem;
    miPause: TMenuItem;
    miStop: TMenuItem;
    sdMain: TSaveDialog;
    odMain: TOpenDialog;
    miSave: TMenuItem;
    miOpen: TMenuItem;
    tbtSeparator3: TToolButton;
    miView: TMenuItem;
    tbtInformation: TToolButton;
    actInformation: TAction;
    miInformation: TMenuItem;
    procedure FormDestroy(Sender: TObject);
    procedure miExitClick(Sender: TObject);
    procedure miAboutClick(Sender: TObject);
    procedure pnTopRightResize(Sender: TObject);
    procedure actCreateExecute(Sender: TObject);
    procedure stbMainResize(Sender: TObject);
    procedure scbMemoryMapResize(Sender: TObject);
    procedure actOpenExecute(Sender: TObject);
    procedure actSaveExecute(Sender: TObject);
    procedure actAddTaskExecute(Sender: TObject);
    procedure actDeleteTaskExecute(Sender: TObject);
    procedure lvTaskQueueSelectItem(Sender: TObject; Item: TListItem;
      Selected: Boolean);
    procedure actStartExecute(Sender: TObject);
    procedure actPauseExecute(Sender: TObject);
    procedure actStopExecute(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure actInformationExecute(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    procedure Delay;
    procedure FilllvFreeMemList(FreeMemList: PFreeMemRec);
    procedure ResetControls;
    procedure SetControlsEnabled(bEnabled: boolean);
    procedure SetStatusOnOpenOrCreateProject;
    procedure SetStatusOnStop;
    procedure RefreshInformation;
    procedure SimulationLoop;
    procedure SavelvTaskQueueItems;
    procedure RestorelvTaskQueueItems;
  public
  end;

{Ініціалізація змінних}

var
  fmMain: TfmMain;

  MinAddress, MaxAddress, TotalMemorySize: TMemUnit;
  FreeMemList: PFreeMemRec;

  TasksCounter: Integer = 1;

  CurTime: DWORD;
```

```
  TotalRefusals: DWORD = 0;
  bWasAccounted: boolean = false;

  bRunning: boolean = false;
  bStopped: boolean = true;

  TaskQueueArr: array of TTaskQueueRec;

implementation

{$R *.DFM}

uses
   uCreate, uFuncs, uAddTask, uInformation, uAbout;

const
  sCurTimeName='Час: %d';

{Процедура відображення карти пам'яті}

procedure DrawMemoryStatus(FreeMemList: PFreeMemRec; Image: TImage;
PaintAreaWidth, PaintAreaHeight: Integer; ScrollBarVisible: boolean;
FreeMemColor, AllocatedMemColor: TColor);
var
  Bitmap: TBitmap;
  CellsPerLine: Integer;
  R: TRect;
  P: PFreeMemRec;
  FromAddress, ToAddress: TMemUnit;
  i: Integer;

  procedure DrawMemoryCell(Address: TMemUnit; Color: TColor);
  begin
    Address:=(Address - MinAddress){ + 1};

    Bitmap.Canvas.Brush.Color:=clBlack;
    R.TopLeft.x:=1 + ((Address mod CellsPerLine) * 8);
    R.TopLeft.y:=1 + ((Address div CellsPerLine) * 10);
    R.BottomRight.x:=R.TopLeft.x + 7;
    R.BottomRight.y:=R.TopLeft.y + 9;
    Bitmap.Canvas.FillRect(R);

    Bitmap.Canvas.Brush.Color:=Color;
    Inc(R.TopLeft.x);
    Inc(R.TopLeft.y);
    Dec(R.BottomRight.x);
    Dec(R.BottomRight.y);
    Bitmap.Canvas.FillRect(R);
  end;

begin
  Bitmap:=TBitmap.Create;
  try
    if (PaintAreaWidth div 8) * ((PaintAreaHeight - 1) div 10) < TotalMemorySize
then
      if not ScrollBarVisible then
        Dec(PaintAreaWidth, GetSystemMetrics(SM_CYVSCROLL));
    CellsPerLine:=PaintAreaWidth div 8;
    PaintAreaHeight:=(TotalMemorySize div (CellsPerLine)) * 10;
    Inc(PaintAreaHeight);
    if TotalMemorySize mod CellsPerLine <> 0 then
      Inc(PaintAreaHeight, 10);
    Bitmap.Width:=PaintAreaWidth;
    Bitmap.Height:=PaintAreaHeight;

    Bitmap.Canvas.Brush.Color:=clWhite;
    R.TopLeft.x:=0;
    R.TopLeft.y:=0;
    R.BottomRight.x:=PaintAreaWidth;
```

```
    R.BottomRight.y:=PaintAreaWidth;
    Bitmap.Canvas.FillRect(R);

    FromAddress:=MinAddress;

    P:=FreeMemList;
    while P <> nil do
      begin
        ToAddress:=P^.Address - 1;
        for i:=FromAddress to ToAddress do
          DrawMemoryCell(i, AllocatedMemColor);
        FromAddress:=P^.Address;
        ToAddress:=P^.Address + P^.Size - 1;
        for i:=FromAddress to ToAddress do
          DrawMemoryCell(i, FreeMemColor);
        FromAddress:=ToAddress + 1;

        P:=P^.Next;
      end;

    for i:=FromAddress to MaxAddress do
      DrawMemoryCell(i, AllocatedMemColor);

    Image.Picture.Assign(Bitmap);
  finally
    Bitmap.Free;
  end;
end;

{Процедура відображення списку вільних блоків пам'яті}

procedure FreeList(var FirstItem: PFreeMemRec);
var
  NextItem: PFreeMemRec;
begin
  while FirstItem <> nil do
    begin
      NextItem:=FirstItem^.Next;
      Dispose(FirstItem);
      FirstItem:=NextItem;
    end;
end;

{Процедура очищення списку вільних блоків пам'яті}

procedure ResetFreeMemList(var FreeMemList: PFreeMemRec);
begin
  FreeList(FreeMemList);
  New(FreeMemList);
  FreeMemList^.Address:=MinAddress;
  FreeMemList^.Size:=TotalMemorySize;
  FreeMemList^.Next:=nil;
end;

{Процедура вибору швидкості процесу емуляції}

procedure TfmMain.Delay;
var
  StartTime: DWORD;
begin
  StartTime:=GetTickCount;
  while (GetTickCount - StartTime < 1000 div (trbSpeed.Position + 1)) and
(bRunning) do
    Application.ProcessMessages;
end;

{Функція знаходження і розподілу вільних блоків пам'яті}
```

```
function FindAndAllocateBlock(var AvailFreeMemList: PFreeMemRec; Size: TMemUnit;
var Address: TMemUnit): boolean;
var
  P, Q, L, QP: PFreeMemRec;

  procedure AllocateBlock(var Q: PFreeMemRec);
  var
    K: TMemUnit;
    P: PFreeMemRec;
  begin
    K:=Q^.Size - Size;

    Address:=Q^.Address;
    if K = 0 then
      begin
        P:=Q^.Next;
        Dispose(Q);
        Q:=P;
      end
    else
      begin
        Q^.Address:=Q^.Address+Size;
        Q^.Size:=K;
      end;

    Result:=true;
  end;

begin
  Q:=nil;

  P:=FreeMemList;
  while (P <> nil) do
    begin
      if (P^.Size >= Size) then
        if Q = nil then
          begin
            Q:=P;
            QP:=L;
          end
        else
          if (P^.Size < Q^.Size) then
            begin
              Q:=P;
              QP:=L;
            end;

      L:=P;
      P:=P^.Next;
    end;

  if Q = nil then
    Result:=false
  else
    if Q = FreeMemList then
      AllocateBlock(FreeMemList)
    else
      AllocateBlock(QP^.Next);
end;

{Функція, що повертає адреси та розміри вільних блоків пам'яті}

function FreeBlock(var AvailFreeMemList: PFreeMemRec; P0Address, P0Size:
TMemUnit): PFreeMemRec;
var
  P, Q: PFreeMemRec;
begin
  New(Result);
  Result^.Address:=P0Address;
```

```
  Result^.Size:=P0Size;

  Q:=nil;
  P:=AvailFreeMemList;

  while (P <> nil) and (P^.Address <= Result^.Address) do
    begin
      Q:=P;
      P:=P^.Next;
    end;
  if (P <> nil) and (Result^.Address + Result^.Size = P^.Address) then
    begin
      Result^.Size:=Result^.Size + P^.Size;
      Result^.Next:=P^.Next;
      Dispose(P);
    end
  else
    Result^.Next:=P;

  if (Q <> nil) and (Q^.Address + Q^.Size = Result^.Address) then
    begin
      Q^.Size:=Q^.Size + Result^.Size;
      Q^.Next:=Result^.Next;

      Dispose(Result);
      Result:=Q;
    end
  else
    if Q = nil then
      AvailFreeMemList:=Result
    else
      Q^.Next:=Result;
end;

{Функція, що обчислює скільки процентів пам'яті використовується у поточний
момент часу}

function GetMemoryUsedPercents: String;
var
  P: PFreeMemRec;
  TotalMemoryUsed: TMemUnit;
begin
  P:=FreeMemList;

  TotalMemoryUsed:=TotalMemorySize;

  while P <> nil do
    begin
      Dec(TotalMemoryUsed, P^.Size);

      P:=P^.Next;
    end;
  Result:=IntToStr(Round(TotalMemoryUsed * 100 / TotalMemorySize))+'%';
end;

{Процедура заповнення вільних блоків пам'яті}

procedure TfmMain.FilllvFreeMemList(FreeMemList: PFreeMemRec);
begin
  lvFreeMemList.Items.Clear;
  while FreeMemList <> nil do
    begin
      with lvFreeMemList.Items.Add do
        begin
          Caption:=IntToStr(FreeMemList^.Address);
          SubItems.Add(IntToStr(FreeMemList^.Size));
        end;

      FreeMemList:=FreeMemList^.Next;
```

```
    end;
end;

{Процедура знищення карти пам'яті}

procedure TfmMain.FormDestroy(Sender: TObject);
begin
  FreeList(FreeMemList);
end;

{Процедура закриття головної програми}

procedure TfmMain.miExitClick(Sender: TObject);
begin
  Close;
end;

{Процедура відкриття вікна "Про програму..."}

procedure TfmMain.miAboutClick(Sender: TObject);
begin
  fmAbout:=TfmAbout.Create(Self);
  try
       fmAbout.ShowModal;
  finally
    fmAbout.Free;
  end;
end;

{Процедура зміни розмірів головного вікна програми}

procedure TfmMain.pnTopRightResize(Sender: TObject);
begin
  pnFreeMemList.Height:=(pnTopRight.ClientHeight - Splitter3.Height) div 2;
end;


{Процедура очищення змінних}

procedure TfmMain.ResetControls;
begin
  imgMemoryMap.Picture.Graphic:=nil;
  imgMemoryMap.Width:=0;
  imgMemoryMap.Height:=0;
  lvTaskQueue.Items.Clear;
  lvWorkingTasks.Items.Clear;
  lvFreeMemList.Items.Clear;

  stbMain.Panels[1].Text:=Format(sCurTimeName, [0]);
end;

{Процедура розблокування кнопок на панелі задач, після створення проекту}

procedure TfmMain.SetControlsEnabled(bEnabled: boolean);
begin
  actAddTask.Enabled:=bEnabled;
  actOpen.Enabled:=bEnabled;
  actSave.Enabled:=bEnabled;
end;

{Процедура відображення стану проекту та відкритої або створеної вручну черги
задач}

procedure TfmMain.SetStatusOnOpenOrCreateProject;
begin
  ResetControls;
  SetControlsEnabled(true);
  actStart.Enabled:=true;
```

```
  TotalMemorySize:=MaxAddress-MinAddress+1;
  ResetFreeMemList(FreeMemList);

  FilllvFreeMemList(FreeMemList);
  DrawMemoryStatus(FreeMemList, imgMemoryMap, scbMemoryMap.ClientWidth,
scbMemoryMap.ClientHeight, scbMemoryMap.VertScrollBar.IsScrollBarVisible,
clWhite, clFuchsia);
end;

{Процедура створення проекту}

procedure TfmMain.actCreateExecute(Sender: TObject);
begin
  fmCreate:=TfmCreate.Create(Self);
  try
    fmCreate.ShowModal;
    if fmCreate.ModalResult=mrOk then
      begin
        sdMain.FileName:='';

        TasksCounter:=1;
        MinAddress:=StrToInt(ProcessMaskEditString(fmCreate.meMinAddress.Text));
        MaxAddress:=StrToInt(ProcessMaskEditString(fmCreate.meMaxAddress.Text));

        SetStatusOnOpenOrCreateProject;
      end;
  finally
    fmCreate.Free;
  end;
end;

{Процедура зміни розміру панелі для карти пам'яті в залежності від вказаної
користувачем кількості динамічної пам'ті}

procedure TfmMain.stbMainResize(Sender: TObject);
begin
  stbMain.Panels[0].Width:=stbMain.ClientWidth-170;
end;

{Процедура зміни розміру карти пам'яті в залежності від вказаної користувачем
кількості динамічної пам'ті}

procedure TfmMain.scbMemoryMapResize(Sender: TObject);
begin
  if imgMemoryMap.Picture.Graphic <> nil then
    DrawMemoryStatus(FreeMemList, imgMemoryMap, scbMemoryMap.ClientWidth,
scbMemoryMap.ClientHeight, scbMemoryMap.VertScrollBar.IsScrollBarVisible,
clWhite, clFuchsia);
end;

{Процедура відкриття готової черги задач}

procedure TfmMain.actOpenExecute(Sender: TObject);
var
  s: String;
begin
  if odMain.Execute then
    begin
      sdMain.FileName:=odMain.FileName;

      AssignFile(Input, odMain.FileName);
      Reset(Input);

      ReadLn(MinAddress, MaxAddress, TasksCounter);

      SetStatusOnOpenOrCreateProject;

      while not EOF do
        with lvTaskQueue.Items.Add do
```

```
          begin
            ReadLn(s);
            Caption:=s;
            ReadLn(s);
            SubItems.Add(s);
            ReadLn(s);
            SubItems.Add(s);
            ReadLn(s);
            SubItems.Add(s);
          end;

      CloseFile(Input);
    end;
end;

{Процедура збереження у текстовому файлі створеної черги задач}

procedure TfmMain.actSaveExecute(Sender: TObject);

  procedure DoSaveProject;
  var
    i: Integer;
  begin
    AssignFile(Output, sdMain.FileName);
    Rewrite(Output);

    WriteLn(MinAddress, ' ', MaxAddress, ' ', TasksCounter);

    for i:=0 to lvTaskQueue.Items.Count - 1 do
      with lvTaskQueue.Items[i] do
        begin
          WriteLn(Caption);
          WriteLn(SubItems[0]);
          WriteLn(SubItems[1]);
          WriteLn(SubItems[2]);
        end;

    CloseFile(Output);
  end;

begin
  if sdMain.FileName <> '' then
    DoSaveProject
  else
    if sdMain.Execute then
      DoSaveProject;
end;

{Процедура додавання задачі у чергу}

procedure TfmMain.actAddTaskExecute(Sender: TObject);
var
  TaskBeginning: LongWord;
  i: Integer;
begin
  fmAddTask:=TfmAddTask.Create(Self);
  try
    fmAddTask.edName.Text:='Задача '+IntToStr(TasksCounter);
    fmAddTask.ShowModal;
    if fmAddTask.ModalResult=mrOk then
      begin

TaskBeginning:=StrToInt(ProcessMaskEditString(fmAddTask.meBeginning.Text));

        i:=0;
        while (i <= lvTaskQueue.Items.Count - 1) and (TaskBeginning >=
StrToInt(lvTaskQueue.Items[i].SubItems[1])) do
          Inc(i);
        with lvTaskQueue.Items.Insert(i) do
```

```
          begin
            Caption:=fmAddTask.edName.Text;
            SubItems.Add(ProcessMaskEditString(fmAddTask.meSize.Text));
            SubItems.Add(ProcessMaskEditString(fmAddTask.meBeginning.Text));
            SubItems.Add(ProcessMaskEditString(fmAddTask.meDuration.Text));
          end;

        Inc(TasksCounter);
      end;
  finally
    fmAddTask.Free;
  end;
end;

{Процедура видалення задачі}

procedure TfmMain.actDeleteTaskExecute(Sender: TObject);
begin
  if lvTaskQueue.Selected <> nil then
    lvTaskQueue.Selected.Delete;
end;

procedure TfmMain.lvTaskQueueSelectItem(Sender: TObject; Item: TListItem;
  Selected: Boolean);
begin
  if actAddTask.Enabled then
    actDeleteTask.Enabled:=lvTaskQueue.Selected <> nil;
end;

{Процедура припинення процесу емуляції користувачем}

procedure TfmMain.SetStatusOnStop;
begin
  actCreate.Enabled:=true;
  SetControlsEnabled(true);
  actStart.Enabled:=true;
  actPause.Enabled:=false;
  actStop.Enabled:=false;

  actInformation.Enabled:=false;
  fmInformation.Visible:=false;

  CurTime:=0;
  TotalRefusals:=0;
  bWasAccounted:=false;

  ResetControls;

  ResetFreeMemList(FreeMemList);

  FilllvFreeMemList(FreeMemList);
  DrawMemoryStatus(FreeMemList, imgMemoryMap, scbMemoryMap.ClientWidth,
scbMemoryMap.ClientHeight, scbMemoryMap.VertScrollBar.IsScrollBarVisible,
clWhite, clFuchsia);

  RestorelvTaskQueueItems;
end;

{Процедура відображення інформаційного вікна, що містить кількість відмав та
процент використаної пам'яті}

procedure TfmMain.RefreshInformation;
begin
  fmInformation.laRefusals.Caption:='Відмов: '+IntToStr(TotalRefusals);
  fmInformation.laMemoryUsed.Caption:='Використання пам"яті:
'+GetMemoryUsedPercents;
end;

{Процедура емуляції розподілу динамічної пам'яті}
```

```
procedure TfmMain.SimulationLoop;
var
  Address: LongWord;
  i: Integer;
  bWasChanges: boolean;

  procedure MakeTimeStep;
  begin
    Inc(CurTime);
    stbMain.Panels[1].Text:=Format(sCurTimeName, [CurTime]);
  end;

begin
  while bRunning do
    begin
      //Створення черги задач
      bWasChanges:=false;

      while (lvWorkingTasks.Items.Count > 0) and
(StrToInt(lvWorkingTasks.Items[0].SubItems[0]) <= CurTime) do
        begin
          FreeBlock(FreeMemList, StrToInt(lvWorkingTasks.Items[0].SubItems[1]),
StrToInt(lvWorkingTasks.Items[0].SubItems[2]));

          lvWorkingTasks.Items[0].Delete;

          bWasChanges:=true;
        end;

      while (lvTaskQueue.Items.Count > 0) and
(StrToInt(lvTaskQueue.Items[0].SubItems[1]) <= CurTime) do
        if FindAndAllocateBlock(FreeMemList,
StrToInt(lvTaskQueue.Items[0].SubItems[0]), Address) then
          begin
            i:=0;
            while (i <= lvWorkingTasks.Items.Count - 1) and
(CurTime+StrToInt(lvTaskQueue.Items[0].SubItems[2]) >=
StrToInt(lvWorkingTasks.Items[i].SubItems[0])) do
              Inc(i);

            with lvWorkingTasks.Items.Insert(i) do
              begin
                Caption:=lvTaskQueue.Items[0].Caption;

SubItems.Add(IntToStr(CurTime+StrToInt(lvTaskQueue.Items[0].SubItems[2])));
                SubItems.Add(IntToStr(Address));
                SubItems.Add(lvTaskQueue.Items[0].SubItems[0]);
              end;
            lvTaskQueue.Items[0].Delete;

            bWasChanges:=true;

            bWasAccounted:=false;
          end
        else
          begin
            if not bWasAccounted then
              begin
                Inc(TotalRefusals);
                bWasAccounted:=true;
              end;
            Break;
          end;

      if bWasChanges then
        begin
          FilllvFreeMemList(FreeMemList);
```

```
          DrawMemoryStatus(FreeMemList, imgMemoryMap, scbMemoryMap.ClientWidth,
scbMemoryMap.ClientHeight, scbMemoryMap.VertScrollBar.IsScrollBarVisible,
clWhite, clFuchsia);
        end;

      if fmInformation.Visible then
        RefreshInformation;

      Delay;

      if bRunning then
        MakeTimeStep
      else
        if bStopped then
          SetStatusOnStop
        else
          begin
            actPause.Enabled:=false;
            actStart.Enabled:=true;

            MakeTimeStep;
          end;
    end;
end;

{Процедура запуску емуляції}

procedure TfmMain.actStartExecute(Sender: TObject);
begin
  if CurTime = 0 then
    begin
      SavelvTaskQueueItems;

      actCreate.Enabled:=false;
      SetControlsEnabled(false);
      actStop.Enabled:=true;

      actInformation.Enabled:=true;
      actInformation.Execute;

      bStopped:=false;
    end;
  actStart.Enabled:=false;
  actPause.Enabled:=true;

  bRunning:=true;

  SimulationLoop;
end;

{Процедура призупинення процесу емуляції}

procedure TfmMain.actPauseExecute(Sender: TObject);
begin
  bRunning:=false;
end;

procedure TfmMain.actStopExecute(Sender: TObject);
begin
  if bRunning then
    begin
      bRunning:=false;
      bStopped:=true;
    end
  else
    begin
      bStopped:=true;
      SetStatusOnStop;
    end;
```

```
end;

{Процедура закінчення процесу емуляції}

procedure TfmMain.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  if bRunning then
    actStop.Execute;
end;

{Процедура зміни формату черги задач перед записом у файл}

procedure TfmMain.SavelvTaskQueueItems;
var
  i: Integer;
begin
  SetLength(TaskQueueArr, lvTaskQueue.Items.Count);
  for i:=0 to lvTaskQueue.Items.Count - 1 do
    with TaskQueueArr[i], lvTaskQueue.Items[i] do
      begin
        Name:=Caption;
        Size:=SubItems[0];
        Beginning:=SubItems[1];
        Duration:=SubItems[2];
      end;
end;

{Процедура відновлення черги задач із файлу}

procedure TfmMain.RestorelvTaskQueueItems;
var
  i: Integer;
begin
  for i:=0 to Length(TaskQueueArr) - 1 do
    with lvTaskQueue.Items.Add, TaskQueueArr[i] do
      begin
        Caption:=Name;
        SubItems.Add(Size);
        SubItems.Add(Beginning);
        SubItems.Add(Duration);
      end;
  TaskQueueArr:=nil;
end;

{Процедура пропорційної зміни розмірів панелей при зміні розміру вікна}

procedure TfmMain.actInformationExecute(Sender: TObject);
begin
  if not fmInformation.Visible then
    begin


      RefreshInformation;

      fmInformation.Left:=fmMain.Left + fmMain.Width - fmInformation.Width - 3;
      fmInformation.Top:=fmMain.Top + ((fmMain.Height - fmInformation.Height)
div 2);
      fmInformation.Visible:=true;
    end;
end;

{Процедура відкриття файлу довідки}

procedure TfmMain.FormCreate(Sender: TObject);
begin
  Application.HelpFile:=ChangeFileExt(Application.ExeName, '.hlp');
end;

end.
```

Файл uCreate.pas – створення проекту

```
unit uCreate;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Mask;

type
  TfmCreate = class(TForm)
    meMinAddress: TMaskEdit;
    laMinAddress: TLabel;
    meMaxAddress: TMaskEdit;
    laMaxAddress: TLabel;
    btOK: TButton;
    btCancel: TButton;
    procedure btOKClick(Sender: TObject);
  private
  public
  end;

var
  fmCreate: TfmCreate;

implementation

{$R *.DFM}

uses
  uFuncs, uMsgs;

procedure InformationMaxCanNotBeLessThatMin;
begin
  MessageBox(0, 'Максимальна адреса не може'#13#10'бути менше мінімальної.',
PChar(sInformationName), MB_ICONASTERISK or MB_TASKMODAL or MB_OK);
end;

procedure TfmCreate.btOKClick(Sender: TObject);
var
  Value1, Value2: LongWord;
begin
  Value1:=StrToInt(ProcessMaskEditString(meMinAddress.Text));
  Value2:=StrToInt(ProcessMaskEditString(meMaxAddress.Text));
  if Value2 < Value1 then
    begin
      InformationMaxCanNotBeLessThatMin;
      fmCreate.ModalResult:=mrNone;
    end;
end;

end.
```

Файл uAddTask.pas – обробка помилок користувача

```
unit uAddTask;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Mask;

type
  TfmAddTask = class(TForm)
    laName: TLabel;
    edName: TEdit;
    laSize: TLabel;
    meSize: TMaskEdit;
    laBeginning: TLabel;
    meBeginning: TMaskEdit;
    laDuration: TLabel;
    meDuration: TMaskEdit;
    btOK: TButton;
    btCancel: TButton;
    procedure btOKClick(Sender: TObject);
  private
  public
  end;

var
  fmAddTask: TfmAddTask;

implementation

{$R *.DFM}

uses
  uFuncs, uMsgs, uMain;

procedure InformationTaskSizeCanNotBeZero;
begin
  MessageBox(0, 'Розмір займаної задачею пам"яті'#13#10'не може бути рівний 0.',
PChar(sInformationName), MB_ICONASTERISK or MB_TASKMODAL or MB_OK);
end;

procedure InformationTaskSizeCanNotBeMoreThatTotalMemory;
begin
  MessageBox(0, PChar('Розмір займаної задачею пам"яті не може'#13#10'бути
більше загального об"єму пам"яті ('+IntToStr(TotalMemorySize)+').'),
PChar(sInformationName), MB_ICONASTERISK or MB_TASKMODAL or MB_OK);
end;

procedure InformationTaskDurationCanNotBeZero;
begin
  MessageBox(0, 'Час роботи задачі не'#13#10'може бути рівний 0.',
PChar(sInformationName), MB_ICONASTERISK or MB_TASKMODAL or MB_OK);
end;

procedure TfmAddTask.btOKClick(Sender: TObject);
var
  TaskSize, TaskDuration: LongWord;
begin
  TaskSize:=StrToInt(ProcessMaskEditString(meSize.Text));
  if TaskSize = 0 then
    begin
      InformationTaskSizeCanNotBeZero;
      fmAddTask.ModalResult:=mrNone;
    end
  else
```

```
    if TaskSize > TotalMemorySize then
      begin
        InformationTaskSizeCanNotBeMoreThatTotalMemory;
        fmAddTask.ModalResult:=mrNone;
      end
    else
      begin
        TaskDuration:=StrToInt(ProcessMaskEditString(meDuration.Text));
        if TaskDuration = 0 then
          begin
            InformationTaskDurationCanNotBeZero;
            fmAddTask.ModalResult:=mrNone;
          end;
      end;
end;

end.
```

Файл uAbout.pas довідки

```
unit uAbout;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, StdCtrls, ShellAPI;

type
  TfmAbout = class(TForm)
    btOk: TButton;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;

  private
  public
  end;

var
  fmAbout: TfmAbout;

implementation

{$R *.dfm}

end.
```

Файл uMain2.pas генератора черги задач

```
unit uMain2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, jpeg, ExtCtrls;

type
  TTaskQueueRec = record
    Name, Size, Beginning, Duration: String;
  end;


  TfmMain = class(TForm)
    btGenerateAndSave: TButton;
    sdMain: TSaveDialog;
    Image1: TImage;
    procedure btGenerateAndSaveClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
  public
  end;

var
  fmMain: TfmMain;

implementation

{$R *.DFM}

{Процедура сортування черги задач по часу запуску}

procedure Sort(var A: array of TTaskQueueRec);

  procedure QuickSort(var A: array of TTaskQueueRec; iLo, iHi: Integer);
  var
    Lo, Hi: Integer;
    Mid, T: TTaskQueueRec;
  begin
    Lo := iLo;
    Hi := iHi;
    Mid := A[(Lo + Hi) div 2];
    repeat
      while StrToInt(A[Lo].Beginning) < StrToInt(Mid.Beginning) do Inc(Lo);
      while StrToInt(A[Hi].Beginning) > StrToInt(Mid.Beginning) do Dec(Hi);
      if Lo <= Hi then
      begin
        T := A[Lo];
        A[Lo] := A[Hi];
        A[Hi] := T;
        Inc(Lo);
        Dec(Hi);
      end;
    until Lo > Hi;
    if Hi > iLo then QuickSort(A, iLo, Hi);
    if Lo < iHi then QuickSort(A, Lo, iHi);
  end;

begin
  QuickSort(A, Low(A), High(A));
end;

{Процедура генерації та збереження у текстовому файлі черги задач}

procedure TfmMain.btGenerateAndSaveClick(Sender: TObject);
```

```
var
  TotalTasks, i: Integer;
  TaskQueueArr: array of TTaskQueueRec;
begin
  if sdMain.Execute then
    begin
      AssignFile(Output, sdMain.FileName);
      Rewrite(Output);

      TotalTasks:=Random(20) + 30;

      WriteLn(5, ' ', 132, ' ', TotalTasks + 1);

      SetLength(TaskQueueArr, TotalTasks);

      for i:=0 to TotalTasks - 1 do
        begin
          TaskQueueArr[i].Name:='Задача '+IntToStr(i);
          TaskQueueArr[i].Size:=IntToStr(Random(30)+1);
          TaskQueueArr[i].Beginning:=IntToStr(Random(140)+10);
          TaskQueueArr[i].Duration:=IntToStr(Random(50)+1);
        end;

      Sort(TaskQueueArr);

      for i:=0 to TotalTasks - 1 do
        begin
          WriteLn(TaskQueueArr[i].Name);
          WriteLn(TaskQueueArr[i].Size);
          WriteLn(TaskQueueArr[i].Beginning);
          WriteLn(TaskQueueArr[i].Duration);
        end;


      TaskQueueArr:=nil;

      CloseFile(Output);
    end;
end;



{Процедура ініціалізації}

procedure TfmMain.FormCreate(Sender: TObject);
begin
  Randomize;
end;

end.
```