

Міністерство освіти і науки України
Криворізький національний університет
Кафедра моделювання та програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА
на здобуття ступеня вищої освіти бакалавра
зі спеціальності 121 – Інженерія програмного забезпечення

На тему: Розробка інформаційної системи аналізу використання апаратних ресурсів комп'ютера в реальному часі

Засвідчую, що в цій
кваліфікаційній роботі немає
запозичень із праць інших
авторів без відповідних
посилань.

Студент гр. ІПЗ–23ск

_____ / Б. Т. Зінов`єв /

Керівник кваліфікаційної
роботи

_____ / Д. В. Швець /

В.о. завідувача кафедри

_____ / А. М. Стрюк /

Кривий Ріг

2026

Криворізький національний університет

Факультет: Інформаційних технологій

Кафедра: Моделювання та програмного забезпечення

Ступінь вищої освіти: бакалавр

Спеціальність: 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

В.о. зав. кафедри

_____ А. М. Стрюк

«_____» _____ 2026 р.

ЗАВДАННЯ

на кваліфікаційну роботу

студенту групи ІПЗ–23ск Зінов`єву Богдану Тимуровичу

1. Тема: «Розробка інформаційної системи аналізу використання апаратних ресурсів комп'ютера в реальному часі» затверджена наказом по КНУ № 283с від «28» травня 2026 р.
2. Термін подання студентом закінченої роботи: «31» травня 2026 р.
3. Вихідні дані по роботі: створений програмний продукт має забезпечити контроль використання апаратних ресурсів ПК.
4. Зміст пояснювальної записки (перелік питань, що їх треба розробити): провести аналіз існуючих на ринку аналогічних програмних продуктів, обґрунтувати функціонал розроблюваної системи, створити програмне забезпечення, здійснити тестування розробленого додатку.
5. Перелік ілюстративного матеріалу: функціональна схема, блок–схема алгоритму, зображення екранних форм додатку.

Календарний план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Розгляд літературних джерел та пошук інтернет-ресурсів з заданої тематики	03.01.26 – 24.01.26
2	Аналіз існуючих методів вирішення проблеми	25.01.26 – 07.02.26
3	Формулювання актуальності роботи і постановка завдань	08.02.26 – 18.02.26
4	Оформлення матеріалів першого розділу роботи	18.02.26 – 02.03.26
5	Створення функціональної системи та алгоритму додатку	03.03.26 – 15.03.26
6	Оформлення матеріалів другого розділу роботи	16.04.26 – 09.04.26
7	Розробка інтерфейсу програмного забезпечення, програмних модулів	10.04.26 – 01.05.26
8	Оформлення додатків	02.05.26 – 07.05.26
9	Тестування розробленої програми	08.05.26 – 15.05.26
10	Оформлення пояснювальної записки	16.05.26 – 30.05.26

Дата видачі завдання: «02» січня 2026 р.

Студент: _____ / Б. Т. Зінов`єв /

Керівник роботи: _____ / Д. В. Швець /

РЕФЕРАТ

ІНФОРМАЦІЙНА СИСТЕМА, МОНІТОРИНГ РЕСУРСІВ, ПРОДУКТИВНІСТЬ, ЦЕНТРАЛЬНИЙ ПРОЦЕСОР, ОПЕРАТИВНА ПАМ'ЯТЬ, ІНТЕГРАЛЬНА ОЦІНКА, ВІЗУАЛІЗАЦІЯ ДАНИХ.

Пояснювальна записка: 56 с., 12 рис., 1 дод., 15 джерел.

Об'єкт дослідження: процеси функціонування обчислювальних комплексів та закономірності зміни показників завантаженості їхніх компонентів під впливом обчислювальних завдань.

Предмет дослідження: алгоритми та програмні інструменти збору, структурування й графічного відображення метрик роботи апаратного забезпечення.

У ході виконання роботи було створено інформаційну систему для спостереження за станом центрального процесора, оперативної пам'яті, дискових накопичувачів та мережевих адаптерів. Усі ресурсомісткі процеси збору даних винесено у площину виконання фонових завдань, що дозволило зберегти високу швидкість реакції програми. Для динамічного оновлення візуальних елементів налаштовано безпечну передачу стану прогресу до головного вікна. Крім того, розроблено алгоритм обчислення загальної інтегральної оцінки продуктивності та індексу стабільності на основі зібраної історії значень. Впроваджено систему текстових рекомендацій, яка допомагає користувачеві виявляти проблеми з перевантаженням системи.

ABSTRACT

INFORMATION SYSTEM, RESOURCE MONITORING, PERFORMANCE, CENTRAL PROCESSING UNIT, RANDOM ACCESS MEMORY, INTEGRAL EVALUATION, DATA VISUALIZATION.

Explanatory note: 56 pages, 12 figures, 1 appendix, 15 sources.

Object of research: the operational processes of computing complexes and the patterns of change in the load indicators of their components under the influence of computational tasks.

Subject of research: algorithms and software tools for collecting, structuring, and graphically displaying hardware operation metrics.

During the execution of the work, an information system was created to monitor the state of the central processing unit, random access memory, disk drives, and network adapters. All resource-intensive data collection processes were delegated to background tasks, which allowed for maintaining high program responsiveness. To dynamically update visual elements, a secure transfer of the progress state to the main window was configured. Furthermore, an algorithm was developed to calculate the overall integral performance evaluation and stability index based on the collected history of values. A system of textual recommendations was implemented to help the user identify problems with system overload.

ЗМІСТ

ВСТУП.....	7
1 ТЕОРЕТИЧНІ ОСНОВИ АНАЛІЗУ ПРОДУКТИВНОСТІ КОМП'ЮТЕРНИХ СИСТЕМ.....	8
1.1. Поняття продуктивності комп'ютерної системи	8
1.2. Основні апаратні ресурси комп'ютера та їх вплив на продуктивність....	9
1.3. Методи моніторингу використання апаратних ресурсів.....	12
1.4. Аналіз існуючих програмних рішень.....	14
1.5 Обґрунтування необхідності розробки власного програмного продукту	20
2 ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ АНАЛІЗУ ВИКОРИСТАННЯ АПАРАТНИХ РЕСУРСІВ КОМП'ЮТЕРА В РЕАЛЬНОМУ ЧАСІ.....	22
2.1 Постановка задачі дослідження.....	22
2.2 Функціональні вимоги	23
2.3 Метрики розрахунку використання системних ресурсів	24
3 РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ АНАЛІЗУ ВИКОРИСТАННЯ АПАРАТНИХ РЕСУРСІВ КОМП'ЮТЕРА В РЕАЛЬНОМУ ЧАСІ.....	28
3.1 Стек технологій для розробки	28
3.2 Функціональна схема програми	30
3.3 Алгоритм роботи	31
3.4 Опис розроблених інструментальних засобів та інструкція	33
3.5 Інструкція до використання.....	36
ВИСНОВКИ.....	38
ПЕРЕЛІК ПОСИЛАНЬ.....	40
Додаток А - Програмний код.....	42

ВСТУП

Розвиток обчислювальних технологій супроводжується збільшенням вимог до продуктивності комп'ютерних систем. Ефективна експлуатація доступного обладнання стає вагомим фактором стабільності як для персональних комп'ютерів, так і для серверних комплексів [1]. У зв'язку з тим, що сучасне програмне забезпечення обробляє величезні потоки інформації, виникає гостра потреба у безперервному відстеженні параметрів роботи заліза. Будь-які непередбачувані затримки в отриманні діагностичних даних можуть призвести до технічних збоїв, через що детальний аналіз процесів стає дедалі затребуванішим, змушуючи розробників шукати нові підходи до оптимізації архітектури моніторингових систем.

Наявні програмні альтернативи для контролю стану апаратного забезпечення часто характеризуються надмірною складністю, розрахованою переважно на вузьких спеціалістів, або, навпаки, – надають надто поверхневі відомості без аналітичного складника. Проектування інформаційної системи, яка гармонійно поєднує простоту сприйняття з високою інформативністю збору даних, дає змогу запобігати аномальній поведінці програмних продуктів. Завдяки розгортанню адаптивних засобів контролю користувачі мають змогу виявляти приховані процеси перевантаження безпосередньо у момент їх появи, що полегшує подальше обслуговування комп'ютера. Це свідчить про доцільність розробки функціональної системи моніторингу в межах кваліфікаційної роботи.

1 ТЕОРЕТИЧНІ ОСНОВИ АНАЛІЗУ ПРОДУКТИВНОСТІ КОМП'ЮТЕРНИХ СИСТЕМ

1.1. Поняття продуктивності комп'ютерної системи

Розгляд проблематики моніторингу апаратних ресурсів доцільно розпочати зі розгляду базових концепцій. Продуктивність комп'ютерної системи є кількісною та якісною мірою її здатності виконувати поставлені обчислювальні завдання за визначений проміжок часу. Ця характеристика не є сталою величиною, оскільки вона динамічно змінюється залежно від поточного навантаження на обладнання. У загальному розумінні, продуктивність відображає обсяг корисної роботи, яку техніка здатна виконати до моменту вичерпання своїх апаратних можливостей. Відповідно, моніторинг цих показників дозволяє розуміти реальний стан машини в будь-який момент часу.

На швидкість обробки даних впливає складна сукупність апаратних та програмних факторів. Серед визначальних елементів варто виділити центральний процесор, де частота генератора тактових імпульсів та кількість фізичних ядер формують основний обчислювальний потенціал, що забезпечує виконання інструкцій. Крім того, обсяг і пропускна здатність оперативної пам'яті напряму впливають на здатність системи працювати з великими масивами даних без необхідності постійного звернення до повільніших накопичувачів. Не менш вагомим фактором є швидкість роботи дискової підсистеми, адже навіть найпотужніший процесор простоюватиме [1], якщо швидкість читання чи запису файлів на носій буде недостатньою. Взаємодія всіх цих компонентів утворює загальну картину ефективності робочої станції.

Для об'єктивного оцінювання стану комп'ютера використовуються специфічні критерії. Найчастіше розробники та системні адміністратори звертають увагу на коефіцієнт завантаження процесора, відсоток використання оперативної пам'яті та інтенсивність операцій введення-виведення, які є найпоказовішими. Час відгуку системи на запит користувача

або зовнішньої програми є ще однією важливою метрикою. Оскільки сучасні операційні системи розподіляють ресурси між десятками фонових процесів, виникає потреба у створенні інструментів, які дозволять спостерігати за цими критеріями у процесі їхньої зміни. Розроблювана в межах цієї роботи інформаційна система має безперервно збирати саме ці метрики, надаючи користувачеві актуальну статистику для своєчасного виявлення "вузьких місць" в апаратній конфігурації.

1.2. Основні апаратні ресурси комп'ютера та їх вплив на продуктивність

Розробка інструментів для відстеження стану комп'ютера вимагає розуміння природи обчислювальних ресурсів. Кожна апаратна складова має свої унікальні технічні характеристики, які визначають її роль у загальному процесі обробки інформації [2]. Щоб майбутня інформаційна система могла надавати релевантні та корисні дані, необхідно спершу проаналізувати найбільш значущі компоненти, оскільки відхилення в їхній роботі найчастіше призводять до зниження загальної швидкодії.

1.2.1 Центральний процесор

Центральний процесор є головним обчислювальним вузлом будь-якої робочої станції, який безпосередньо відповідає за виконання інструкцій програмного забезпечення та керування іншими пристроями. Здатність комп'ютера виконувати велику кількість завдань одночасно базується на постійному перемиканні контексту та розподілі процесорного часу між різними потоками. Коли інтенсивність обчислювальних операцій перевищує фізичні можливості обладнання, виникає ефект перевантаження, внаслідок чого операційна система починає примусово призупиняти обробку менш пріоритетних команд для збереження стабільності. Для користувача це зазвичай проявляється у вигляді системних затримок та зниження чуйності візуальних інтерфейсів.

Зважаючи на цю специфіку, сучасні моніторингові додатки мають розроблятися з максимальною ощадливістю до ресурсів. Система проектується з урахуванням того, щоб процедури збору даних самі по собі не ставали обтяжливим фактором для процесора. Досягти такого балансу дозволяє асинхронне виконання операцій зчитування показників з апаратних датчиків, завдяки чому основний потік програми залишається повністю вільним для відмальовування та оновлення користувацького інтерфейсу.

1.2.2 Оперативна пам'ять

Оперативна пам'ять функціонує як високошвидкісне тимчасове сховище, куди завантажуються необхідні дані та виконуваний код активних процесів перед їхньою передачею на обробку до процесора. Наявність достатнього обсягу вільного простору в оперативній пам'яті є визначальним чинником для плавної роботи системи [3], адже швидкість доступу до її комірок у багато разів перевищує швидкість читання інформації навіть з сучасних твердотільних накопичувачів.

Коли доступна оперативна пам'ять вичерпується, операційна система змушена переносити частину неактивних даних на жорсткий диск (механізм файлу підкачки). Таке перенесення супроводжується падінням швидкодії під час перемикання між програмами.

Тому одним із головних завдань розроблюваної інформаційної системи є не одноразова фіксація зайнятих мегабайтів ОЗП, а й безперервна передача прогресу споживання пам'яті окремими процесами на екран користувача. Аналіз цих показників у динаміці дасть змогу виявляти так звані "витоки пам'яті" у сторонньому програмному забезпеченні, які виникають тоді, коли певна програма резервує ресурси для своїх потреб, але забуває звільнити їх після завершення роботи алгоритму. Впровадження такого моніторингу зробить додаток дієвим інструментом для виявлення прихованих проблем у програмному середовищі комп'ютера.

1.2.3 Дискова підсистема

Дискова підсистема виконує функцію енергонезалежного збереження інформації та історично вважається найповільнішим компонентом серед основних обчислювальних вузлів машини. Навіть за умови масового переходу на швидкісні твердотільні накопичувачі, операції читання та запису файлів об'єктивно вимагають значно більше часу порівняно зі зверненнями до оперативної пам'яті. Коли обсяг даних, які потребують обробки, перевищує пропускну здатність накопичувача, утворюється черга запитів, що фахівці розглядають як [4] один із головних факторів загального гальмування комп'ютера. Оскільки сучасні операційні системи активно використовують дисковий простір для розміщення файлу підкачки при нестачі оперативної пам'яті, затримки на цьому апаратному рівні безпосередньо впливають на загальну плавність роботи прикладного програмного забезпечення.

У контексті розробки інформаційної системи, моніторинг активності дисків вимагає певного архітектурного підходу. Процес опитування апаратних контролерів для зняття показників може тривати непередбачуваний час. Саме тому для збору метрик швидкості читання, запису та довжини черги буде застосовано паралельне виконання відповідних процедур. Цей підхід дозволить винести ресурсомісткі операції опитування обладнання у площину виконання фонових завдань, залишаючи графічний інтерфейс програми повністю вільним для взаємодії з користувачем, що є суттєвою перевагою розробки порівняно з простішими утилітами. Завдяки цьому користувач отримуватиме актуальну інформацію про стан накопичувачів без візуальних затримок у роботі самого додатка.

1.2.4 Мережеві ресурси

Хоча мережеві адаптери не здійснюють прямих обчислень, у середовищі веб-орієнтованих і хмарних технологій їхня пропускну здатність багато в чому визначає швидкість виконання повсякденних завдань. Стан підключення безпосередньо позначається на продуктивності тих додатків, які потребують

постійної синхронізації з віддаленими серверами [5], тому ігнорування мережевого компонента зробило б комплексний аналіз системи неповним. Перевантаження каналу зв'язку прихованими фоновими процесами, такими як оновлення компонентів операційної системи або завантаження об'ємних баз даних, часто помилково сприймається непідготовленими користувачами як збій у роботі центрального процесора чи нестача пам'яті.

Для забезпечення максимально повної картини стану робочої станції, розроблюваний додаток відстежуватиме обсяги вхідного та вихідного трафіку мережевого інтерфейсу. Реалізація цієї функції передбачає передачу стану прогресу обміну пакетами на рівень користувацького інтерфейсу у вигляді динамічних графіків, що полегшує візуальне сприйняття статистики. Водночас важливо враховувати, що мережеві з'єднання можуть бути нестабільними або розриватися з незалежних від комп'ютера причин. З огляду на це, архітектура модулів збору мережевих даних включатиме механізм скасування операцій. Таке рішення гарантуватиме коректне завершення потоків опитування у разі раптового відключення кабелю чи зміни конфігурації маршрутизатора, запобігаючи аварійному завершенню роботи нашої системи моніторингу.

1.3. Методи моніторингу використання апаратних ресурсів

Для побудови ефективної інформаційної системи необхідно проаналізувати наявні підходи до отримання діагностичних даних. Операційні системи сімейства Windows пропонують розгалужений набір інструментів, які дозволяють спостерігати за станом обчислювальної машини з різним рівнем деталізації.

Базовими системними засобами Windows для таких завдань традиційно виступають Диспетчер завдань та Монітор ресурсів [6], які надають загальну статистику щодо завантаженості обладнання. Вони зручні для швидкої оцінки стану комп'ютера, проте їхніх можливостей часто виявляється недостатньо для глибокої аналітики або тривалого відстеження прихованих процесів. Більш

потужним вбудованим рішенням є Системний монітор, який оперує великою кількістю специфічних метрик. Однак його інтерфейс орієнтований переважно на системних адміністраторів, що робить його заскладним для повсякденного використання звичайними користувачами. Зважаючи на це, проєктований додаток має поєднати інформативність професійних утиліт зі зрозумілим та доступним графічним представленням.

Щоб реалізувати власний інструмент збору даних, необхідно звернутися до спеціалізованих програмних інтерфейсів. В екосистемі Windows основним джерелом такої інформації є лічильники продуктивності та інструментарій управління Windows. Лічильники продуктивності забезпечують найшвидший доступ до динамічних показників, таких як відсоток використання процесора чи обсяг зайнятої пам'яті. Інструментарій управління, у свою чергу, краще підходить для отримання статичної інформації про конфігурацію обладнання. Зчитування цих показників вимагає обережного підходу до архітектури майбутньої програми. Щоб уникнути зависання інтерфейсу користувача під час опитування датчиків, буде застосовуватися виконання фонових завдань, що дозволить відокремити ресурсомісткі розрахунки від процесу відтворення графіків на екрані.

Моніторинг у реальному часі в контексті розроблюваного додатка передбачає безперервне опитування апаратних інтерфейсів із заданим інтервалом. Оскільки звернення до системних бібліотек може займати різний час залежно від поточного навантаження на машину, єдиним правильним архітектурним рішенням є розпаралелене виконання всіх процедур збору метрик, що гарантує плавність роботи самої програми. Зібрана статистика обробляється у фоні, після чого відбувається регулярна передача стану прогресу та актуальних обчислених значень до головного вікна для оновлення візуальних елементів. Важливою вимогою до розроблюваної системи є її здатність безпечно зупиняти опитування обладнання під час закриття вікон або зміни налаштувань. Для цього на архітектурному рівні інтегрується надійний механізм скасування операцій [7], який забезпечує своєчасне

вивільнення захоплених системних ресурсів та запобігає виникненню конфліктів у пам'яті.

1.4. Аналіз існуючих програмних рішень

Для визначення архітектурних та функціональних вимог до проектованої системи необхідно проаналізувати поширені на ринку програмні продукти. Найдоступнішими інструментами є вбудовані утиліти операційної системи Windows, зокрема Диспетчер завдань та Монітор ресурсів [8], які надають базову інформацію про загальний стан обчислювальної машини.

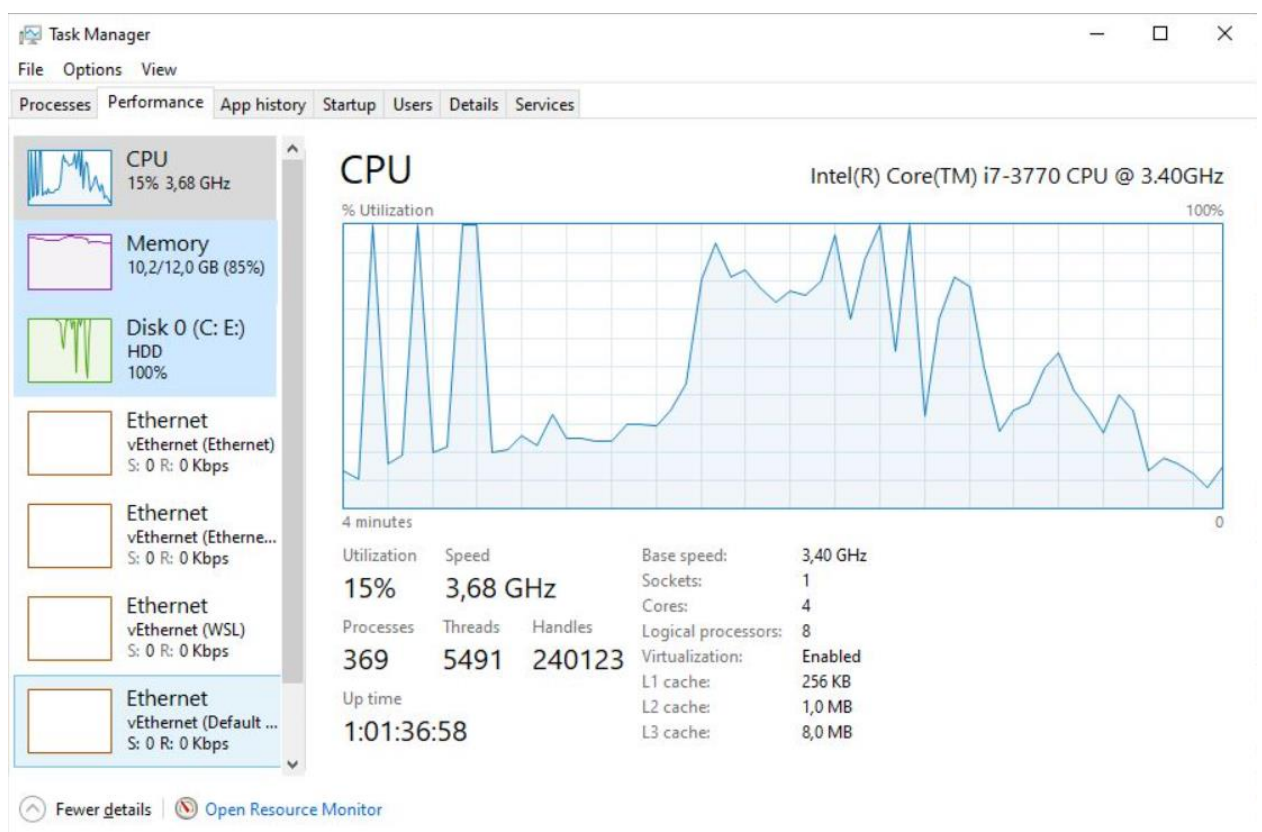


Рисунок 1.1 – Диспетчер задач

Диспетчер завдань є дуже зручним для швидкого перегляду завантаженості центрального процесора та оперативної пам'яті, проте він не зберігає тривалих історичних даних і не здатний зчитувати показники з апаратних датчиків температури. Зі свого боку, Монітор ресурсів пропонує

глибший рівень деталізації щодо дискової активності та мережних з'єднань, але його інтерфейс вимагає від користувача певного часу на пошук необхідних метрик серед великих таблиць. Обидва інструменти добре справляються з базовими завданнями, але їхня закрита архітектура не передбачає гнучкого налаштування віджетів чи тривалого збереження аналітики для подальшого вивчення.

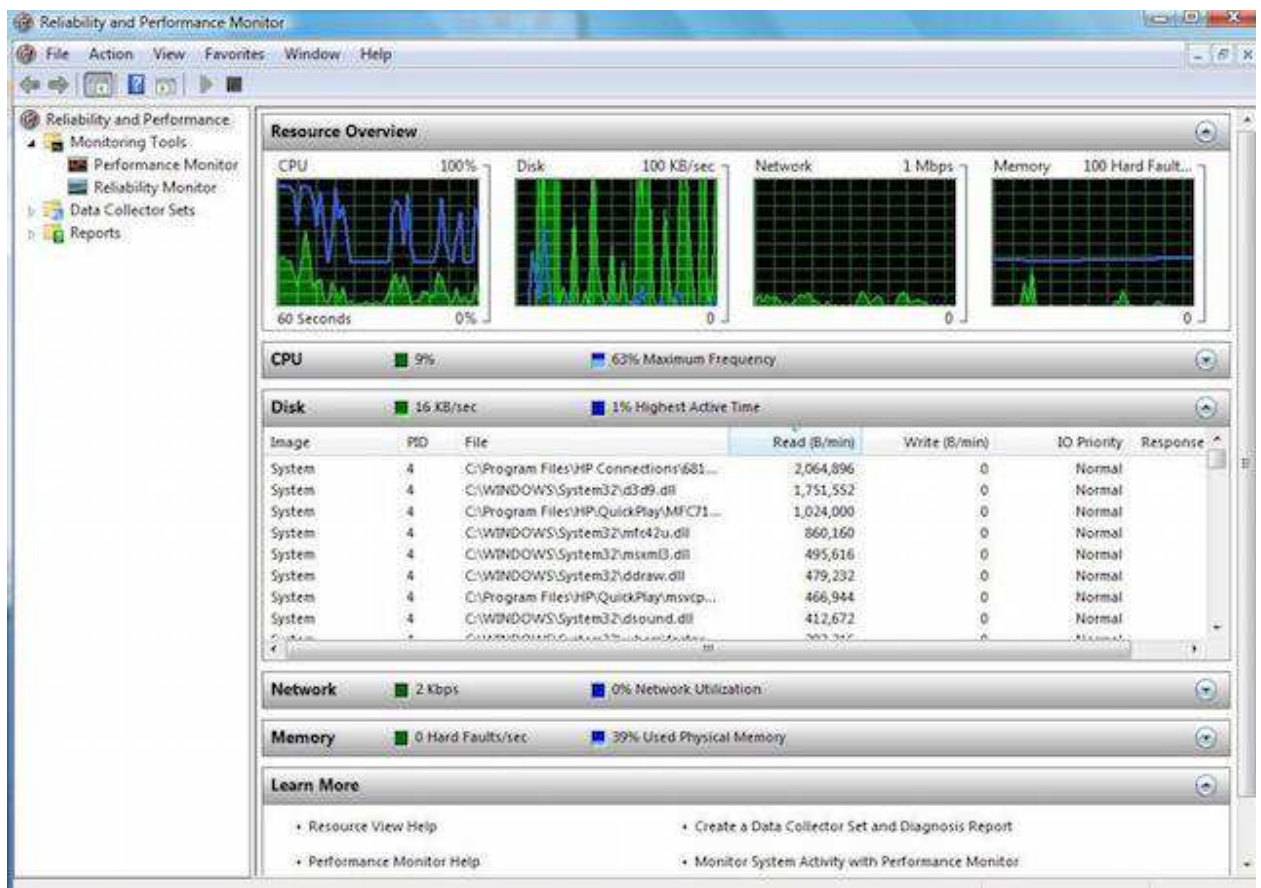
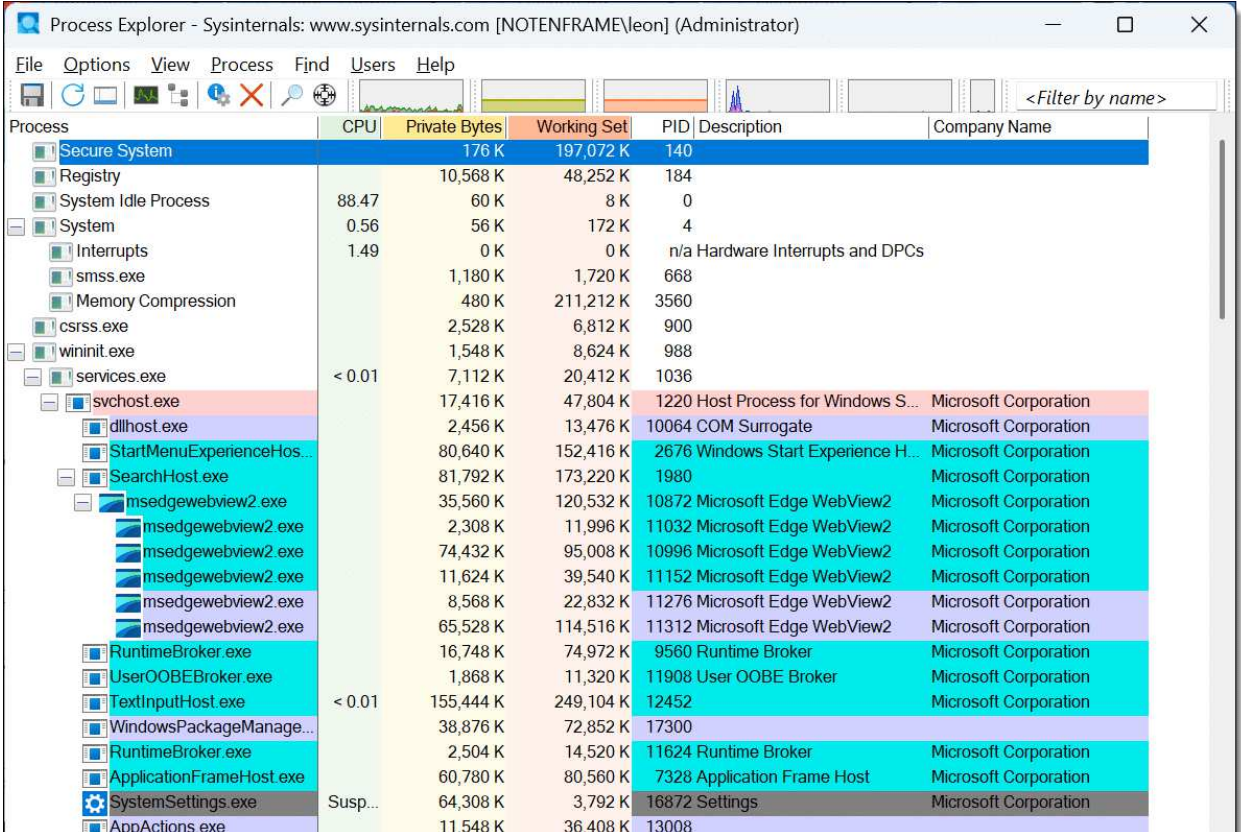


Рисунок 1.2 – Монітор ресурсів

Серед більш спеціалізованих безкоштовних утиліт виділяється Process Explorer від підрозділу Sysinternals [9]. Ця програма надає розгорнуту інформацію про дерево активних процесів, реальне споживання пам'яті та використання системних дескрипторів. Вона є корисною для пошуку проблемного програмного забезпечення, оскільки забезпечує постійну

передачу стану прогресу використання адресного простору кожним окремим потоком.

Водночас Process Explorer зосереджений переважно на програмному рівні функціонування системи, залишаючи поза увагою фізичні апаратні датчики (контролери напруги на материнській платі чи показники швидкості обертання вентиляторів охолодження).



Process	CPU	Private Bytes	Working Set	PID	Description	Company Name
Secure System		176 K	197,072 K	140		
Registry		10,568 K	48,252 K	184		
System Idle Process	88.47	60 K	8 K	0		
System	0.56	56 K	172 K	4		
Interrupts	1.49	0 K	0 K	n/a	Hardware Interrupts and DPCs	
smss.exe		1,180 K	1,720 K	668		
Memory Compression		480 K	211,212 K	3560		
csrss.exe		2,528 K	6,812 K	900		
wininit.exe		1,548 K	8,624 K	988		
services.exe	< 0.01	7,112 K	20,412 K	1036		
svchost.exe		17,416 K	47,804 K	1220	Host Process for Windows S...	Microsoft Corporation
dllhost.exe		2,456 K	13,476 K	10064	COM Surrogate	Microsoft Corporation
StartMenuExperienceHos...		80,640 K	152,416 K	2676	Windows Start Experience H...	Microsoft Corporation
SearchHost.exe		81,792 K	173,220 K	1980		Microsoft Corporation
msedgeview2.exe		35,560 K	120,532 K	10872	Microsoft Edge WebView2	Microsoft Corporation
msedgeview2.exe		2,308 K	11,996 K	11032	Microsoft Edge WebView2	Microsoft Corporation
msedgeview2.exe		74,432 K	95,008 K	10996	Microsoft Edge WebView2	Microsoft Corporation
msedgeview2.exe		11,624 K	39,540 K	11152	Microsoft Edge WebView2	Microsoft Corporation
msedgeview2.exe		8,568 K	22,832 K	11276	Microsoft Edge WebView2	Microsoft Corporation
msedgeview2.exe		65,528 K	114,516 K	11312	Microsoft Edge WebView2	Microsoft Corporation
RuntimeBroker.exe		16,748 K	74,972 K	9560	Runtime Broker	Microsoft Corporation
UserOOBEBroker.exe		1,868 K	11,320 K	11908	User OOBEBroker	Microsoft Corporation
TextInputHost.exe	< 0.01	155,444 K	249,104 K	12452		Microsoft Corporation
WindowsPackageManage...		38,876 K	72,852 K	17300		
RuntimeBroker.exe		2,504 K	14,520 K	11624	Runtime Broker	Microsoft Corporation
ApplicationFrameHost.exe		60,780 K	80,560 K	7328	Application Frame Host	Microsoft Corporation
SystemSettings.exe	Susp...	64,308 K	3,792 K	16872	Settings	Microsoft Corporation
AppActions.exe		11,548 K	36,408 K	13008		

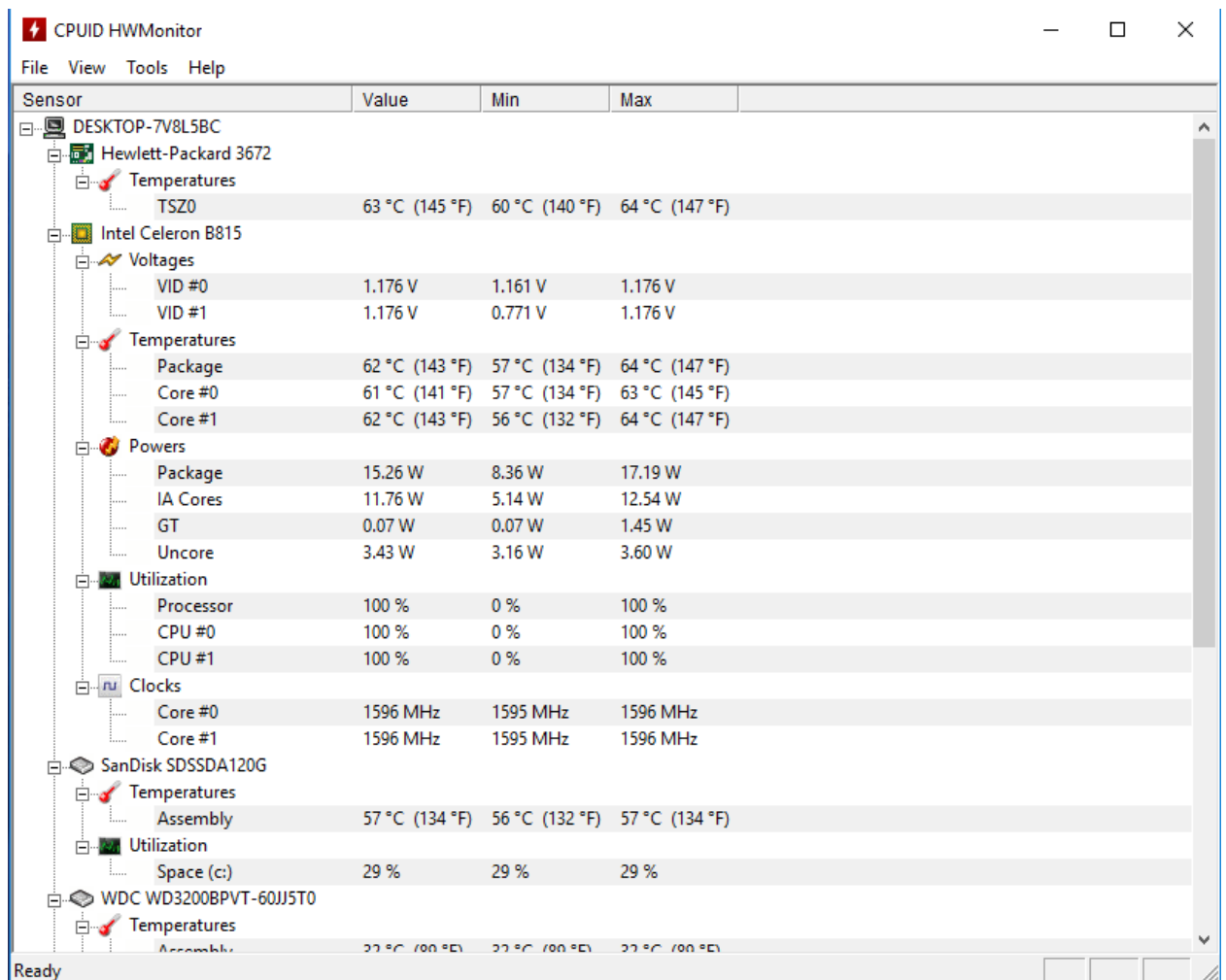
Рисунок 1.3 – Process Explorer

Для комплексного збору даних безпосередньо з апаратних компонентів користувачі найчастіше звертаються до сторонніх рішень, таких як HWMonitor та AIDA64.

Утиліта HWMonitor вирізняється легковагістю та здатністю взаємодіяти майже з усіма відомими мікросхемами моніторингу [10], відображаючи температури, напруги та рівні енергоспоживання у форматі простого ієрархічного дерева.

З іншого боку, програмний комплекс AIDA64 є одним із найпотужніших комерційних продуктів на ринку, який як збирає детальну телеметрію, так і дозволяє проводити стрес-тести обладнання [11].

Попри високу інформативність, такі системи мають певні експлуатаційні недоліки: AIDA64 є значно перевантаженою функціоналом та вимагає платної ліцензії, тоді як HWMonitor пропонує досить застарілий текстовий інтерфейс без можливості побудови наочних графіків у динаміці.



The screenshot shows the CPUID HWMonitor application window. The title bar reads 'CPUID HWMonitor'. The menu bar includes 'File', 'View', 'Tools', and 'Help'. The main content area displays a tree view of system components with corresponding sensor data in a table format. The table has columns for 'Sensor', 'Value', 'Min', and 'Max'. The components listed include the motherboard (DESKTOP-7V8L5BC), system information (Hewlett-Packard 3672), temperatures (TSZ0), CPU (Intel Celeron B815), voltages (VID #0, VID #1), CPU temperatures (Package, Core #0, Core #1), powers (Package, IA Cores, GT, Uncore), utilization (Processor, CPU #0, CPU #1), clocks (Core #0, Core #1), storage (SanDisk SDSSDA120G, WDC WD3200BPVT-60JJ5T0), and their respective temperatures and utilization percentages.

Sensor	Value	Min	Max
DESKTOP-7V8L5BC			
Hewlett-Packard 3672			
Temperatures			
TSZ0	63 °C (145 °F)	60 °C (140 °F)	64 °C (147 °F)
Intel Celeron B815			
Voltages			
VID #0	1.176 V	1.161 V	1.176 V
VID #1	1.176 V	0.771 V	1.176 V
Temperatures			
Package	62 °C (143 °F)	57 °C (134 °F)	64 °C (147 °F)
Core #0	61 °C (141 °F)	57 °C (134 °F)	63 °C (145 °F)
Core #1	62 °C (143 °F)	56 °C (132 °F)	64 °C (147 °F)
Powers			
Package	15.26 W	8.36 W	17.19 W
IA Cores	11.76 W	5.14 W	12.54 W
GT	0.07 W	0.07 W	1.45 W
Uncore	3.43 W	3.16 W	3.60 W
Utilization			
Processor	100 %	0 %	100 %
CPU #0	100 %	0 %	100 %
CPU #1	100 %	0 %	100 %
Clocks			
Core #0	1596 MHz	1595 MHz	1596 MHz
Core #1	1596 MHz	1595 MHz	1596 MHz
SanDisk SDSSDA120G			
Temperatures			
Assembly	57 °C (134 °F)	56 °C (132 °F)	57 °C (134 °F)
Utilization			
Space (c:)	29 %	29 %	29 %
WDC WD3200BPVT-60JJ5T0			
Temperatures			
Assembly	27 °C (80 °F)	27 °C (80 °F)	27 °C (80 °F)

Рисунок 1.4 – HWMonitor

Проведений огляд показує, що наявні рішення здебільшого розділяються на дві полярні категорії: занадто прості системні утиліти або надмірно складні професійні комплекси.

Це підтверджує доцільність розробки власної інформаційної системи, яка зможе поєднати зручний графічний інтерфейс із достатнім рівнем системної аналітики.

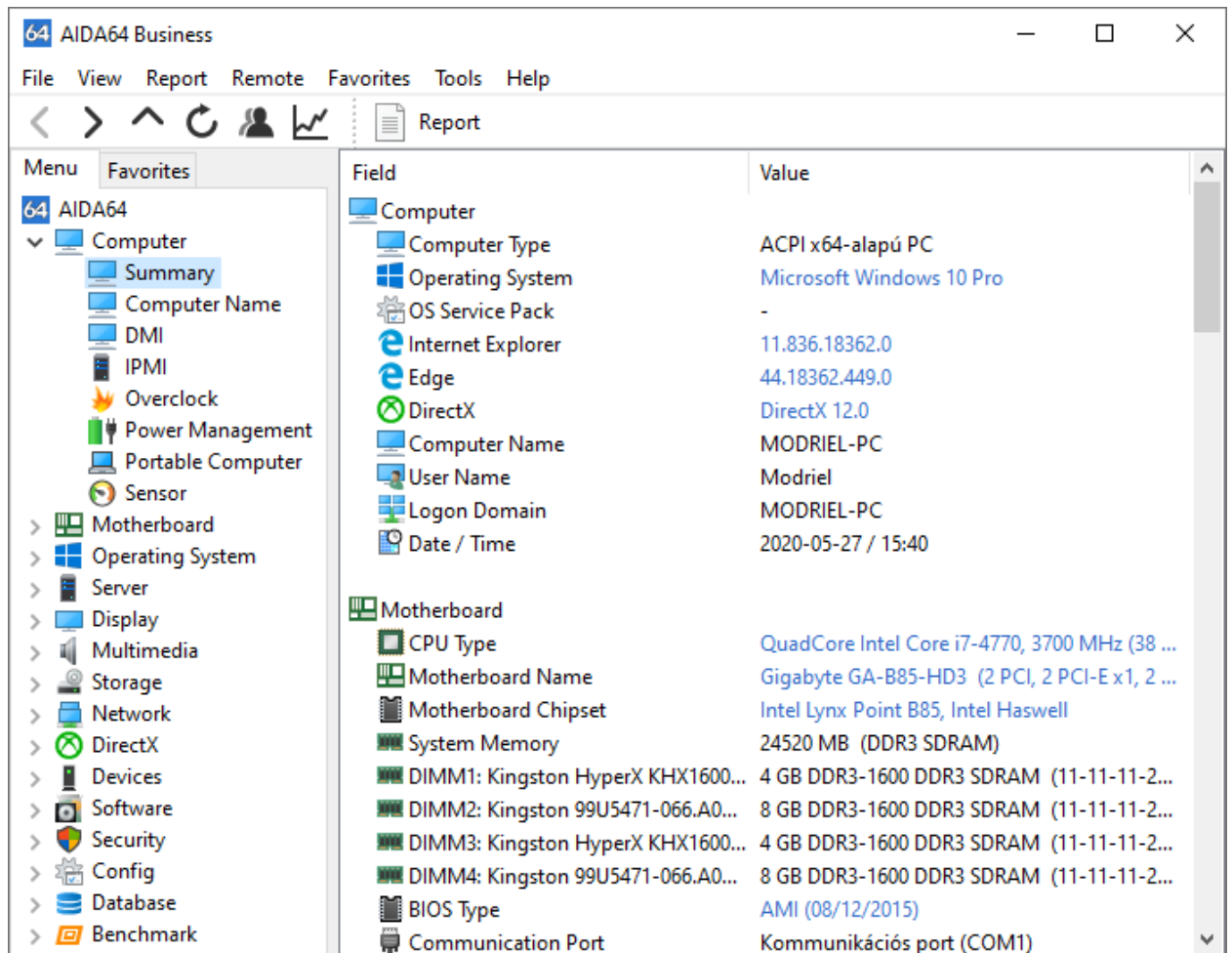


Рисунок 1.5 – AIDA64

У процесі проектування майбутнього додатка буде враховано недоліки існуючих аналогів.

Архітектура програми, своєю чергою, базуватиметься на виконанні фонових завдань із надійним механізмом скасування операцій, що гарантуватиме безпечне завершення роботи потоків моніторингу та мінімальний вплив самого додатка на продуктивність досліджуваного комп'ютера.

1.4.1 Переваги існуючих систем

Наявні на ринку програмні рішення для відстеження апаратних метрик мають низку переваг, які формують певний стандарт очікувань користувачів. Однією з найвагоміших сильних сторін таких комерційних комплексів, як AIDA64, є їхня здатність взаємодіяти з величезною кількістю різноманітного обладнання.

Завдяки багаторічній історії розвитку, розробники накопили великі бази драйверів, що дозволяє програмам отримувати достовірну телеметрію [12] навіть від рідкісних або застарілих материнських плат та процесорів. Крім того, професійні утиліти часто пропонують вбудовані інструменти для проведення навантажувального тестування компонентів та генерації розгорнутих звітів про технічний стан робочої станції.

Якщо ж розглядати базові системні утиліти сімейства Windows, то їхньою головною перевагою є глибока інтеграція в середовище операційної системи [13], яка гарантує високу стабільність роботи та практично виключає виникнення конфліктів доступу до системних ресурсів. Важливим позитивним аспектом багатьох сучасних аналогів також є якісна візуалізація зібраних даних. Використання динамічних графіків, діаграм та ієрархічних списків значно полегшує сприйняття обчислювальної статистики порівняно з переглядом звичайних текстових логів. Проектована інформаційна система запозичить цей підхід, забезпечуючи передачу стану прогресу використання пам'яті та процесора на інтерфейсний рівень у наочному графічному форматі.

1.4.2 Недоліки існуючих систем

Незважаючи на описані сильні сторони, існуючі програмні продукти мають певні обмеження та вразливості, які виправдовують доцільність створення нового, більш оптимізованого додатка.

Одним із найбільш відчутних недоліків багатофункціональних комплексів є їхня надмірність. Вони перевантажені специфічними функціями, якими пересічний користувач майже ніколи не користується, що призводить

до підвищеного споживання оперативної пам'яті самими засобами моніторингу. Виникає парадоксальна ситуація, коли програма, покликана оптимізувати роботу системи, сама стає причиною уповільнення її реакції. З іншого боку, легковагі безкоштовні альтернативи часто мають застарілий та незручний дизайн, який важко назвати орієнтованим на сучасного користувача.

Ще однією поширеною архітектурною проблемою багатьох утиліт є синхронне блокування графічної оболонки під час опитування обладнання. Нерідко трапляється так, що під час звернення до повільного апаратного датчика (наприклад, жорсткого диска, що перейшов у режим сну) інтерфейс програми тимчасово завмирає, що значно погіршує загальне враження від використання та створює ілюзію зависання комп'ютера.

Щоб уникнути цього недоліку в розробці, архітектура додатку базуватиметься на концепції виконання фонових завдань. Усі операції зчитування метрик відбуватимуться ізольовано від головного вікна програми, а вбудований механізм скасування операцій дозволить коректно та безпечно переривати тривалі процеси опитування при зміні налаштувань або закритті додатка. Такий підхід забезпечить високу чуйність розроблюваної системи за будь-яких умов навантаження.

1.5 Обґрунтування необхідності розробки власного програмного продукту

На основі проведеного огляду наявних інструментів можна зробити висновок, що сучасний ринок програмного забезпечення пропонує користувачам переважно дві протилежні ситуації. З одного боку, доступні базові системні утиліти, які не здатні зберігати історичні дані та мають досить обмежений аналітичний потенціал. З іншого боку, існують перевантажені комерційні комплекси [14], які через свою багатофункціональність самі споживають забагато обчислювальних потужностей.

Для звичайного користувача, студента чи системного адміністратора, якому потрібно контролювати завантаженість обладнання без заглиблення у надлишкові налаштування, більшість із цих рішень є неоптимальними. Тому розробка легкого, вузькоспрямованого додатка для відстеження апаратних метрик є обґрунтованим і актуальним завданням для кваліфікаційної роботи.

Розроблювана інформаційна система покликана заповнити цю прогалину, пропонуючи збалансований підхід до моніторингу. Головною ідеєю є створення оптимізованої програмної архітектури, у якій процес зчитування показників з апаратних датчиків жодним чином не впливатиме на плавність відображення графічного інтерфейсу. Для досягнення цієї мети буде застосовано паралельне виконання всіх ресурсомістких операцій, завдяки чому програма залишатиметься чуйною навіть під час тривалого опитування повільних дискових накопичувачів або мережових адаптерів. Уся важка логіка збору та обробки телеметрії виноситься у площину виконання фонових завдань, тоді як до головного вікна додатка здійснюватиметься лише регулярна передача стану прогресу [15] у вигляді оновлених значень для побудови наочних графіків.

Окрім продуктивності, проектування власного програмного продукту дозволяє гарантувати безпеку використання системних ресурсів. У підсумку, створення такої системи дозволить отримати зручний інструмент, який поєднує простоту візуального сприйняття з технічною аналітикою. Розроблений додаток відповідатиме вимогам до прикладного програмного забезпечення та вирішуватиме завдання безперервного відстеження стану комп'ютера.

2 ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ АНАЛІЗУ ВИКОРИСТАННЯ АПАРАТНИХ РЕСУРСІВ КОМП'ЮТЕРА В РЕАЛЬНОМУ ЧАСІ

2.1 Постановка задачі дослідження

На основі попереднього аналізу предметної області та виявлених недоліків наявних аналогів виникає потреба у формулюванні конкретних вимог до майбутнього програмного продукту. Основне завдання цього етапу полягає у переході від загальних теоретичних концепцій до визначення інженерних рішень, які дозволять побудувати надійний інструмент моніторингу, орієнтований на повсякденне використання. Розроблюваний додаток має збирати актуальну статистику про стан обчислювальної машини, обробляти її та виводити на екран, не створюючи при цьому додаткового навантаження на операційну систему.

Відповідно, постановка задачі зводиться до проектування такої архітектури, яка забезпечить повністю незалежне функціонування модулів збору даних від графічного інтерфейсу користувача.

З погляду функціональних можливостей, інформаційна система повинна опитувати апаратні датчики для отримання інформації про завантаженість центрального процесора, обсяг вільної та зайнятої оперативної пам'яті, інтенсивність операцій з дисковими накопичувачами та активність мережевих підключень.

Оскільки ці показники постійно змінюються, необхідно організувати безперервний цикл їх оновлення, де регулярна передача стану прогресу з обчислювальних модулів до візуальних компонентів відбуватиметься без затримок, забезпечуючи плавну побудову аналітичних графіків. Окремим завданням є структурування цих даних у зручному для сприйняття вигляді, щоб користувач міг швидко оцінити загальну ситуацію на робочій станції.

Для забезпечення високої швидкодії та уникнення зависань графічної оболонки під час опитування повільного обладнання, архітектура програми має базуватися на принципі розподілення обов'язків. Усі операції зчитування телеметрії повинні бути винесені у площину виконання фонових завдань, що дозволить звільнити основний потік програми виключно для взаємодії з користувачем.

Таке виконання гарантує, що навіть у випадку тимчасової недоступності певного апаратного компонента (наприклад, зупинки жорсткого диска для економії енергії), додаток залишатиметься повністю працездатним і не блокуватиме систему.

Крім того, вагомою вимогою до проекрованої системи є безпечне управління системними ресурсами під час її тривалої роботи. Оскільки процес моніторингу передбачає створення багатьох паралельних потоків для збору метрик, існує ризик виникнення конфліктів або витоків пам'яті при раптовому закритті програми чи зміні її конфігурації.

Щоб запобігти цьому, до логіки роботи кожного модуля збору даних необхідно інтегрувати надійний механізм скасування операцій, який дозволить коректно та безпечно зупиняти опитування обладнання за першої необхідності.

Вирішення цих архітектурних завдань створить базу для подальшої програмної реалізації інформаційної системи та гарантуватиме її стабільність.

2.2 Функціональні вимоги

Функціональні вимоги описують специфічну поведінку розроблюваної інформаційної системи та деталізують перелік завдань, які вона повинна виконувати під час взаємодії з користувачем та операційним середовищем. Формування цього переліку спирається на мету роботи та необхідність створення оптимізованого інструменту для повсякденного використання.

Основне призначення додатка полягає у безперервному отриманні діагностичної інформації від різних вузлів комп'ютера. З огляду на це, інформаційна система повинна забезпечувати виконання таких функцій:

- а) збір даних про поточне навантаження центрального процесора;
- б) аналіз стану оперативної пам'яті з визначенням загального, доступного та фактично використаного обсягу;
- в) відстеження активності дискової підсистеми;
- г) моніторинг стану мережевих адаптерів із урахуванням обсягів завантаження даних для активних з'єднань.

Окрім безпосереднього зчитування метрик, система повинна мати зручну візуальну складову. Зібрана статистика має фіксуватися та відображатися у вигляді динамічних графіків та шкал завантаженості.

Окремою групою функціональних вимог є забезпечення стабільності та чуйності інтерфейсу під час роботи. Оскільки процес опитування обладнання може займати різний час, програма не повинна зависати. Для цього логіка збору телеметрії має бути ізольованою.

Для відображення актуальних даних на графіках має бути налаштована регулярна передача стану прогресу з обчислювальних потоків до інтерфейсних елементів. Також, враховуючи динамічну природу операційної системи (наприклад, раптове відключення мережевого кабелю або перехід диска в режим сну), програма повинна безпечно реагувати на непередбачувані зміни.

2.3 Метрики розрахунку використання системних ресурсів

Для забезпечення повноцінного моніторингу недостатньо просто виводити показники з апаратних датчиків. Користувач повинен розуміти загальний стан робочої станції, що вимагає впровадження аналітичного складника. Тому архітектура розроблюваної інформаційної системи передбачає збір вихідних даних, їхню статистичну обробку та обчислення єдиного інтегрального показника продуктивності.

Отримання інформації про стан обладнання здійснюється через системні лічильники продуктивності та мережеві інтерфейси операційної системи. Кожен компонент має свою специфіку опитування:

а) Центральний процесор та дискова підсистема. Оскільки ці вузли динамічно змінюють свій стан, перше звернення до їхніх лічильників завжди повертає нульове значення. Тому логіка програми передбачає виконання початкового ініціалізаційного запиту для встановлення базової точки відліку, після чого починається збір реальних відсоткових значень у діапазоні від 0 до 100.

б) Оперативна пам'ять. Зчитується відсоток фактично використаного адресного простору (зафіксованих байтів) відносно загального доступного обсягу.

в) Мережа. Система автоматично обирає перший активний фізичний інтерфейс, ігноруючи віртуальні адаптери, тунелі та локальні петлі (Loopback). Розрахунок швидкості передачі даних базується на обчисленні різниці отриманих та відправлених байтів між двома вимірюваннями, поділений на час, що минув. Отримана швидкість (у байтах за секунду) нормалізується у відсоткове значення відносно пропускної здатності у 100 Мбіт/с. Для наочності, якщо мережева активність присутня, але становить менше одного відсотка, на графіку відображається мінімальне значення в 1%.

г) Інтегральна оцінка продуктивності. Щоб звести розрізнені показники до єдиного зрозумілого індексу, застосовано алгоритм зваженого оцінювання. Чим менше навантаження на систему, тим більше вільних ресурсів залишається для нових завдань, і, відповідно, тим вищим є загальний бал (100 балів означає повну відсутність навантаження). Кожному апаратному компоненту призначено ваговий коефіцієнт, що відображає його вплив на загальну чуйність системи: процесор – 40% (0.4), оперативна пам'ять – 30% (0.3), диск – 20% (0.2), мережа – 10% (0.1). Формула розрахунку має такий вигляд:

$$Score_{total} = (100 - U_{cpu}) \cdot 0.4 + (100 - U_{ram}) \cdot 0.3 + (100 - U_{disk}) \cdot 0.2 + (100 - U_{net}) \cdot 0.1 \quad (2.1)$$

де U – поточний відсоток завантаження відповідного вузла.

Для аналізу динаміки роботи програма формує історію даних. У пам'яті зберігається список останніх вимірювань для кожного ресурсу. Під час додавання нового запису найстаріший запис автоматично видаляється. На основі цього масиву безперервно розраховуються середнє, мінімальне та максимальне значення. Окремим аналітичним інструментом є індекс стабільності, який оцінює рівномірність роботи комп'ютера на основі останніх 50 значень інтегральної оцінки. Алгоритм враховує два типи аномалій: поступові відхилення від середнього рівня та різкі стрибки (зміна показника більш ніж на 20% між двома сусідніми вимірюваннями). Розрахунок штрафних балів за відхилення та стрибки формалізується так:

$$Penalty_{dev} = \left(\frac{\Delta_{avg}}{50} \right) \cdot 30 \quad (2.2)$$

$$Penalty_{jump} = \left(\frac{N_{jumps}}{C_{total}} \right) \cdot 70 \quad (2.3)$$

де Δ_{avg} – середнє відхилення між сусідніми показниками,

N_{jumps} – кількість зафіксованих стрибків,

C_{total} – загальна кількість записів в історії.

Фінальний індекс розраховується відніманням штрафів від ідеальних 100 балів:

$$Index = 100 - Penalty_{dev} - Penalty_{jump} \quad (2.4)$$

Таким чином, максимальний штраф за нестабільність графіка обмежується 30 балами, а часті різкі перепади навантаження можуть знизити індекс ще на 70 балів.

На основі зібраної телеметрії інформаційна система формує текстові поради для користувача. Логіка побудована на системі порогових значень:

а) Процесор: понад 80% – високе завантаження (рекомендується закрити ресурсомісткі процеси); понад 60% – помірне навантаження.

б) Оперативна пам'ять: понад 85% – критичне споживання простору (потрібне закриття зайвих програм або розширення пам'яті); понад 70% – помірне.

в) Диск та Мережа: понад 80% – висока активність (рекомендується зачекати завершення операцій читання/запису чи передачі даних); понад 60% – підвищена активність.

Якщо всі метрики знаходяться в межах норми ($\leq 60\%$ або $\leq 70\%$ для пам'яті), додаток повідомляє про нормальний режим роботи без додаткових рекомендацій.

Для уникнення надмірного споживання обчислювальних потужностей самою програмою, архітектура опитування розділена на різні часові інтервали. Поточні показники та дані для відмальовування графіків оновлюються кожену секунду, формуючи вікно перегляду тривалістю 1 хвилина (60 точок). Щоб запобігти дублюванню системних викликів, значення з лічильників зчитуються лише один раз на початку циклу, зберігаються у внутрішніх змінних і далі використовуються для всіх паралельних обчислень (оновлення інтерфейсу, графіків, інтегральної оцінки). Уся логіка збору даних обгорнута в блоки перехоплення помилок, що гарантує стабільну передачу стану прогресу: якщо певний датчик виходить з ладу або тимчасово недоступний, система безпечно реєструє нульовий рівень навантаження, не перериваючи загальне виконання фонових завдань.

3 РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ АНАЛІЗУ ВИКОРИСТАННЯ АПАРАТНИХ РЕСУРСІВ КОМП'ЮТЕРА В РЕАЛЬНОМУ ЧАСІ

3.1 Стек технологій для розробки

Для реалізації програмного продукту необхідно обрати оптимальний технологічний стек, який поєднував би надійність, простоту розгортання та широку підтримку можливостей операційної системи Windows. Зважаючи на вимоги до продуктивності та стабільності, основою для створення інформаційної системи була обрана платформа .NET Framework 4.8. Її використання гарантує сумісність із більшістю сучасних версій операційних систем від Windows 7 до Windows 11 без необхідності встановлення додаткових громіздких середовищ виконання. Важливою перевагою цієї платформи є наявність вбудованої бібліотеки для побудови графіків, що дозволяє уникнути залежності від сторонніх компонентів.

Основною мовою програмування виступає C#. Вона пропонує інструменти для роботи з об'єктами та колекціями даних. Завдяки вбудованій технології LINQ значно спрощується процес статистичної обробки зібраної телеметрії, оскільки пошук мінімальних, максимальних та середніх значень в історії вимірювань відбувається за допомогою коротких та оптимізованих запитів. Графічний інтерфейс користувача побудовано на базі технології Windows Forms. Цей фреймворк забезпечує швидке створення віконних додатків завдяки набору готових візуальних елементів. Для організації робочого простору використовуються панелі вкладок, а для відображення рівня завантаженості – індикатори прогресу. Головним візуальним інструментом є вбудований компонент Chart, який бере на себе завдання динамічної відмальовки лінійних графіків споживання ресурсів у часі. Інтерфейс проектується таким чином, щоб розділити логіку обробки подій та

автоматично згенерований код дизайну, що значно полегшує подальшу підтримку коду.

Для безпосереднього опитування апаратних датчиків використовуються системні інтерфейси взаємодії. Отримання метрик центрального процесора, оперативної пам'яті та дискової підсистеми здійснюється через класи `PerformanceCounter`. Це стандартний та найнадійніший спосіб зчитування показників активності операційної системи. Своєю чергою, мережева статистика збирається через спеціалізований простір імен `NetworkInformation`, який дозволяє ідентифікувати активні фізичні адаптери та підраховувати обсяги переданих байтів.

Щоб інформаційна система працювала стабільно та не блокувала графічну оболонку під час збору даних, застосовується архітектурний патерн MVC (Модель-Вигляд-Контролер). Усі процедури отримання метрик та математичні розрахунки винесені в окремі класи-аналізатори. Регулярна передача стану прогресу з цих аналізаторів до користувацького інтерфейсу ініціюється за допомогою таймера. Таймер бере на себе роль координатора, який щосекунди запускає виконання фонових завдань зі збору нових значень та оновлення графіків.

Також реалізовано безпечне управління ресурсами комп'ютера. Оскільки підключення до системних лічильників вимагає виділення некерованої пам'яті, у програмі реалізовано надійний механізм скасування операцій на базі інтерфейсу `IDisposable`. Цей підхід гарантує, що при завершенні роботи програми або зміні параметрів моніторингу всі утримувані системні ресурси будуть коректно звільнені.

Основним інструментом розробки, написання коду та проектування інтерфейсу виступає середовище `Visual Studio 2022`, яке забезпечує всі необхідні засоби для компіляції та налагодження десктопних додатків без залучення зовнішніх систем управління залежностями.

3.2 Функціональна схема програми

На рисунку 3.1 наведено функціональну схему інформаційної системи аналізу використання апаратних ресурсів комп'ютера в реальному часі.

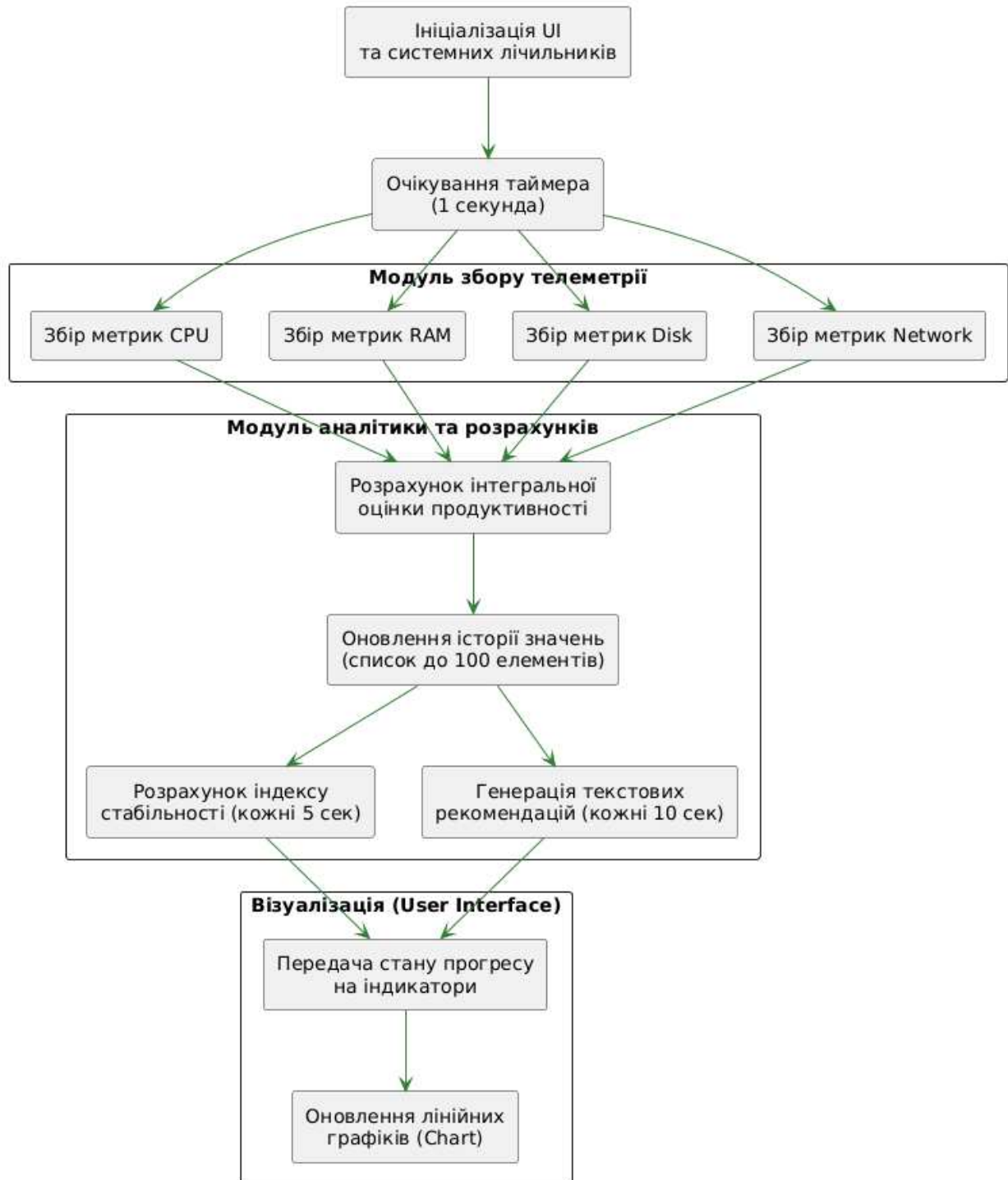


Рисунок 3.1 – Функціональна схема інформаційної системи аналізу використання апаратних ресурсів комп'ютера в реальному часі

Основою робочого процесу є періодичний цикл, керований системним таймером.

Після запуску програми відбувається базова ініціалізація візуальних елементів та системних лічильників, після чого додаток переходить у режим очікування.

Щойно минає заданий інтервал часу, ініціюється виконання фонових завдань зі збору метрик. Дані паралельно зчитуються з центрального процесора, оперативної пам'яті, дискової підсистеми та мережевих адаптерів. У разі закриття програми на цьому або будь-якому іншому етапі спрацьовує механізм скасування операцій, який розриває зв'язок з обладнанням та звільняє виділену оперативну пам'ять.

Зібрані показники надходять до аналітичного модуля. Спочатку обчислюється загальна інтегральна оцінка продуктивності з урахуванням вагових коефіцієнтів кожного ресурсу. Далі ці показники додаються до історії вимірювань для подальшого аналізу. Залежно від внутрішніх лічильників, система також може ініціювати розрахунок індексу стабільності (кожні п'ять секунд) та формування текстових порад для користувача (кожні десять секунд).

Завершальним етапом циклу є передача стану прогресу та всіх обчислених даних до головного вікна програми. Відбувається оновлення текстових написів, індикаторів завантаженості та повна перемальовка динамічних лінійних графіків. Після оновлення інтерфейсу поточна ітерація алгоритму закінчується, і система повертається до очікування наступного такту таймера.

3.3 Алгоритм роботи

На рисунку 3.2 наведено блок-схему алгоритму роботи інформаційної системи аналізу використання апаратних ресурсів комп'ютера в реальному часі.



Рисунок 3.2 – Блок-схема алгоритма

Блок-схема дає змогу наочно зобразити послідовність операцій, які виконуються системою від моменту запуску до її завершення.

Оскільки додаток має графічний інтерфейс, його життєвий цикл є подієво-орієнтованим. Після проходження етапу ініціалізації компонентів та первинного налаштування системних лічильників, програма переходить у стан очікування. Основний робочий цикл прив'язаний до спрацьовування таймера. При настанні чергового такту починається виконання фонових завдань із читання даних апаратних датчиків.

Зібрані показники проходять через математичний блок, де обчислюється загальна оцінка продуктивності, після чого ці дані додаються до внутрішньої історії. Далі алгоритм містить розгалуження, які відповідають за періодичні розрахунки: перевіряється, чи настав час для оновлення індексу стабільності (кожні 5 секунд) та генерації текстових порад (кожні 10 секунд). Після виконання всіх необхідних обчислень здійснюється передача стану прогресу на візуальні елементи головного вікна.

Якщо користувач вирішує завершити роботу з додатком, алгоритм переходить до процедури безпечного виходу. На цьому етапі спрацьовує надійний механізм скасування операцій, який розриває з'єднання зі службами моніторингу, звільняє виділену пам'ять і лише після цього передає керування фінальному блоку завершення програми.

3.4 Опис розроблених інструментальних засобів та інструкція

Результатом виконання практичної частини кваліфікаційної роботи є створення інформаційної системи аналізу використання апаратних ресурсів комп'ютера в реальному часі. Система реалізована для збору діагностичної інформації про стан операційної системи. Головне призначення розробленого інструменту – надати користувачеві зручний засіб для спостереження за апаратними ресурсами без створення додаткового навантаження на обчислювальну машину.

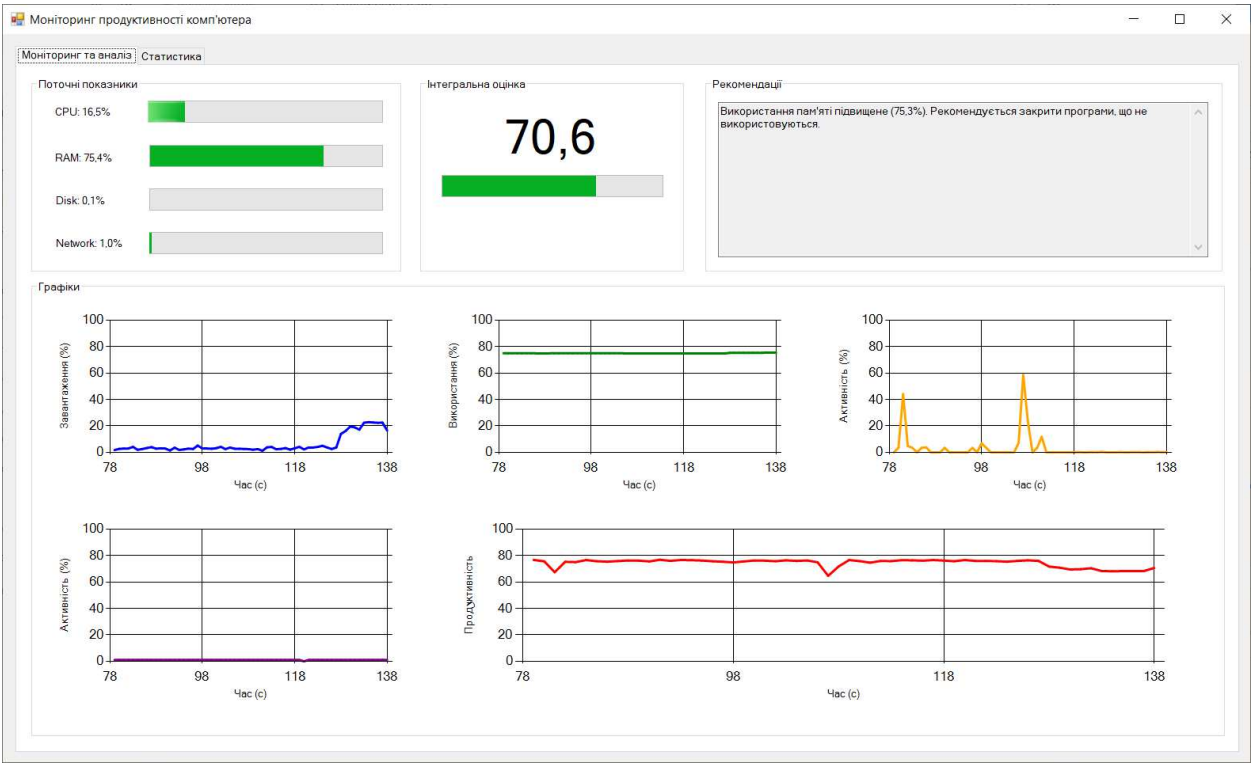


Рисунок 3.3 – Вікно інформаційної системи (вкладка «Моніторинг та аналіз»)

Інтерфейс програми розділений на дві логічні вкладки. Перша з них, «Моніторинг та аналіз», виконує роль головної панелі інструментів. Тут відображаються поточні показники чотирьох основних вузлів: центрального процесора, оперативної пам'яті, дискового накопичувача та мережевих адаптерів. Для зручності сприйняття інформації кожне отримане значення виводиться у вигляді текстового напису з точністю до одного знака після коми, а також дублюється графічним індикатором завантаженості зі шкалою від 0 до 100%.

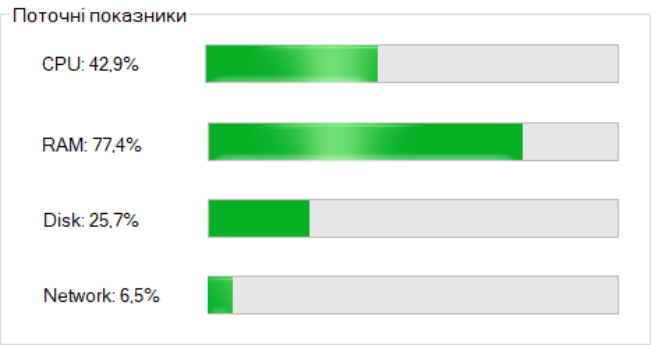


Рисунок 3.4 – Поточні показники

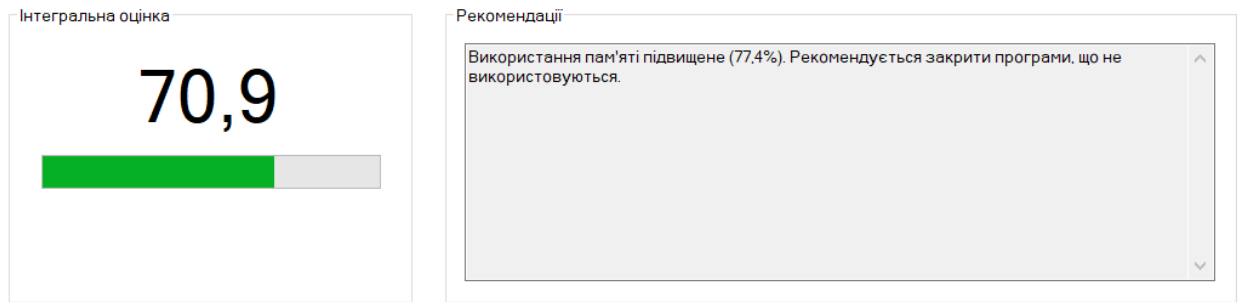


Рисунок 3.5 – Інтегральна оцінка продуктивності та рекомендації

На цій самій вкладці розташований блок інтегральної оцінки продуктивності. Вона розраховується як зважена сума зібраних показників (процесор має вагу 40%, пам'ять – 30%, диск – 20%, мережа – 10%) і дозволяє користувачеві одним поглядом оцінити загальний стан комп'ютера. Візуалізація динаміки навантаження реалізована через п'ять лінійних графіків: по одному для кожного апаратного ресурсу та один для загальної інтегральної оцінки.

Графіки вміщують історію за останню хвилину роботи (60 точок). Кожну секунду відбувається передача стану прогресу на інтерфейсний рівень, завдяки чому графіки плавно оновлюються.

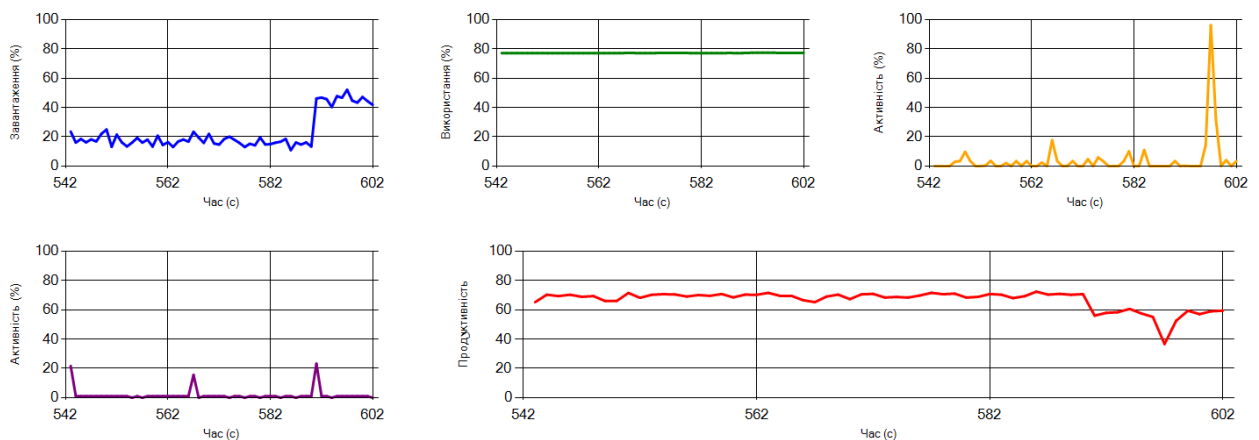


Рисунок 3.6 – Графіки використання системних ресурсів

Також на першій вкладці працює вбудована система автоматичних рекомендацій. Кожні 10 секунд алгоритм перевіряє поточні показники. Якщо

навантаження на певний компонент перевищує помірні або високі пороги, програма генерує відповідну текстову пораду щодо можливих шляхів оптимізації роботи операційної системи.

Друга вкладка має назву «Статистика» і призначена для глибшого аналізу зібраних даних. Тут формуються звіти щодо середнього, мінімального та максимального значень завантаженості процесора, використання оперативної пам'яті та загальної продуктивності.

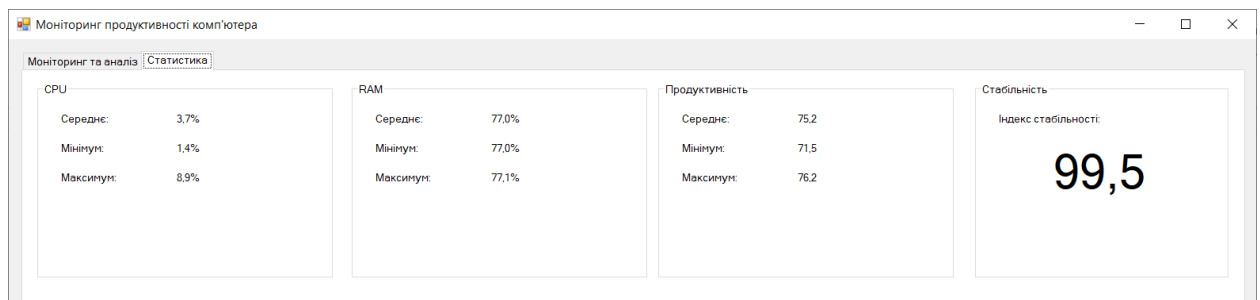


Рисунок 3.7 – Графіки використання системних ресурсів

Окремою важливою метрикою є індекс стабільності системи. Цей показник розраховується кожні 5 секунд на основі останніх 50 вимірювань. Алгоритм відстежує середні відхилення та штрафує загальний бал за різкі стрибки навантаження (понад 20%). Чим вищий індекс (наближений до 100 балів), тим стабільніше працює досліджуваний комп'ютер.

3.5 Інструкція до використання

Процес експлуатації розробленого програмного продукту є простим і складається з таких кроків:

а) Запуск програми. Після відкриття виконуваного файлу додаток автоматично ініціалізує всі необхідні служби та починає збір телеметрії. Надання прав адміністратора або виконання складних налаштувань не вимагається.

б) Спостереження за станом. На головній вкладці «Моніторинг та аналіз» користувач може стежити за поточним станом системи. Якщо графіки

починають стрімко підійматися до максимальних позначок, варто звернути увагу на текстове поле зверху вікна, де з'являться згенеровані рекомендації.

в) Аналіз динаміки. Для оцінки роботи комп'ютера на довшому проміжку часу необхідно перейти на вкладку «Статистика». Там можна переглянути індекс стабільності та середні показники навантаження, щоб зробити висновки про загальну ефективність роботи машини.

г) Завершення роботи. Для зупинки моніторингу достатньо закрити головне вікно програми стандартним способом. Внутрішні алгоритми самостійно виконують безпечне відключення від системних інтерфейсів та вивільняють оперативну пам'ять.

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальне завдання – спроектовано та програмно реалізовано інформаційну систему для безперервного аналізу використання апаратних ресурсів комп'ютера. Зростання вимог до обчислювальної техніки та складність програмного забезпечення зумовлюють необхідність постійного спостереження за станом обладнання. Розроблений додаток успішно справляється з цією задачею, надаючи користувачеві розгорнуту аналітику без створення додаткового навантаження на саму операційну систему.

На першому етапі дослідження було детально вивчено предметну область та проаналізовано існуючі інструменти моніторингу. Огляд показав, що наявні рішення здебільшого є або занадто простими, або надмірно перевантаженими (споживають багато пам'яті та вимагають платної ліцензії). Це обґрунтувало доцільність розробки власного оптимізованого програмного продукту, орієнтованого на повсякденне використання.

Під час архітектурного проектування було сформовано логічну структуру майбутнього додатка. Всі обчислення та звернення до обладнання винесені у площину виконання фонових завдань. Також на цьому етапі було розроблено алгоритми для розрахунку єдиної інтегральної оцінки продуктивності та унікального індексу стабільності.

Практичну реалізацію проекту виконано мовою C# з використанням платформи .NET Framework 4.8 та технології Windows Forms. Створена програма успішно збирає телеметрію з центрального процесора, оперативної пам'яті, дискової підсистеми та мережевих адаптерів. Завдяки правильно побудованій взаємодії компонентів, відбувається регулярна передача стану прогресу з аналітичних модулів до головного вікна, що гарантує плавне відмальовування динамічних графіків та індикаторів.

Окрему увагу в процесі розробки було приділено стабільності роботи та безпечному поводженню із системними ресурсами. Завдяки тому, що в

алгоритм інтегровано надійний механізм скасування операцій, система здатна коректно переривати процеси опитування та повністю вивільняти зайняту оперативну пам'ять під час закриття вікна, запобігаючи програмним збоям.

У підсумку, всі поставлені завдання виконано в повному обсязі, а мету кваліфікаційної роботи досягнуто. Створена інформаційна система є повністю працездатним інструментом, який як відображає базові показники, так і формує текстові рекомендації для оптимізації роботи комп'ютера. Розроблений додаток може бути корисним як для звичайних користувачів, так і для системних адміністраторів під час діагностики технічного стану робочих станцій.

ПЕРЕЛІК ПОСИЛАНЬ

1. How to Measure Computer Performance / Intel. 2024. URL: <https://www.intel.com/content/www/us/en/tech-tips-and-tricks/how-to-measure-computer-performance.html>
2. Олійник В. В. Архітектура та організація комп'ютерних систем : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2022.
3. Memory Management Architecture Guide / Microsoft Learn. 2024. URL: <https://learn.microsoft.com/en-us/sql/relational-databases/memory-management-architecture-guide>
4. Storage I/O Performance Tuning / Microsoft Learn. 2023. URL: <https://learn.microsoft.com/en-us/windows-server/administration/performance-tuning/subsystem/storage-io-performance>
5. Guide to Network Monitoring: Basics, Best Practices & Tools / Paessler. 2025. URL: <https://www.paessler.com/it-explained/network-monitoring>
6. About Performance Counters / Microsoft Learn. 2024. URL: <https://learn.microsoft.com/en-us/windows/win32/perfctr/about-performance-counters>
7. Monitoring System Resources with Code / InfoQ. 2022. URL: <https://www.infoq.com/articles/monitoring-system-resources/>
8. Task Manager: Live performance metrics / Microsoft Learn. 2024. URL: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/taskmgr>
9. Process Explorer - Sysinternals / Microsoft Learn. 2024. URL: <https://learn.microsoft.com/en-us/sysinternals/downloads/process-explorer>
10. HWMonitor hardware monitoring program / CPUID. 2025. URL: <https://www.cpubid.com/software/hwmonitor.html>
11. AIDA64 System Diagnostics / FinalWire. 2025. URL: <https://www.aida64.com/products/aida64-extreme>

12. Hardware Monitoring: Best Practices and Tools / SolarWinds. 2024. URL: <https://www.solarwinds.com/resources/it-glossary/hardware-monitoring>
13. Best hardware monitoring tools for proactive management / ITPro. 2023. URL: <https://www.itpro.com/infrastructure/server-storage/356133/best-hardware-monitoring-tools>
14. System monitoring tools and software / TechTarget. 2024. URL: <https://www.techtarget.com/searchitoperations/definition/system-monitoring>
15. Keep the UI thread responsive / Microsoft Learn. 2024. URL: <https://learn.microsoft.com/en-us/windows/uwp/debug-test-perf/keep-the-ui-thread-responsive>

Додаток А - Програмний код

Form1.cs

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
using System.Windows.Forms.DataVisualization.Charting;

namespace PerformanceMonitor
{
    // Головна форма програми для моніторингу продуктивності
    комп'ютера
    // Містить два вкладки: "Моніторинг та аналіз" та
    "Статистика"
    public partial class Form1 : Form
    {
        // Колектор даних про CPU
        private CpuDataCollector cpuCollector;

        // Колектор даних про RAM
        private RamDataCollector ramCollector;

        // Колектор даних про Disk
        private DiskDataCollector diskCollector;

        // Колектор даних про Network
        private NetworkDataCollector networkCollector;

        // Аналізатор продуктивності
        private PerformanceAnalyzer analyzer;

        // Система рекомендацій
        private RecommendationSystem recommendationSystem;

        // Менеджер статистики
        private StatisticsManager statisticsManager;

        // Калькулятор індексу стабільності
        private StabilityIndexCalculator stabilityCalculator;

        // Таймер для періодичного оновлення даних
        private System.Windows.Forms.Timer updateTimer;

        // Списки для зберігання історії показників для графіків
        private List<double> cpuHistory;
        private List<double> ramHistory;
        private List<double> diskHistory;
        private List<double> networkHistory;
        private List<double> performanceHistory;
```

```

// Лічильник часу для осі X графіків
private int timeCounter;

// Лічильник для періодичного оновлення статистики
private int statisticsUpdateCounter;

// Лічильник для періодичного оновлення рекомендацій
private int recommendationsUpdateCounter;

// Максимальна кількість точок на графіку
private const int MAX_CHART_POINTS = 60;

// Конструктор форми
public Form1()
{
    // Стандартна ініціалізація компонентів
    InitializeComponent();

    // Ініціалізація об'єктів класів
    InitializeComponents();

    // Запускаємо моніторинг при завантаженні форми
    StartMonitoring();
}

// Метод для ініціалізації компонентів програми
private void InitializeComponents()
{
    // Створення об'єктів колекторів даних
    cpuCollector = new CpuDataCollector();
    ramCollector = new RamDataCollector();
    diskCollector = new DiskDataCollector();
    networkCollector = new NetworkDataCollector();

    // Створення об'єктів аналізу та статистики
    analyzer = new PerformanceAnalyzer();
    recommendationSystem = new RecommendationSystem();
    statisticsManager = new StatisticsManager();
    stabilityCalculator = new
StabilityIndexCalculator();

    // Ініціалізація списків історії для графіків
    cpuHistory = new List<double>();
    ramHistory = new List<double>();
    diskHistory = new List<double>();
    networkHistory = new List<double>();
    performanceHistory = new List<double>();

    // Ініціалізація лічильника часу
    timeCounter = 0;
    statisticsUpdateCounter = 0;
}

```

```

        recommendationsUpdateCounter = 0;

        // Налаштування таймера для оновлення даних кожному
секунду
        updateTimer = new System.Windows.Forms.Timer();
        updateTimer.Interval = 1000; // 1000 мс = 1 секунда
        updateTimer.Tick += UpdateTimer_Tick;

        // Ініціалізація графіків
        InitializeCharts();
    }

    // Метод для запуску моніторингу
private void StartMonitoring()
{
    // Ініціалізуємо колектор CPU
    cpuCollector.Initialize();

    // Ініціалізуємо колектор RAM
    ramCollector.Initialize();

    // Ініціалізуємо колектор Disk
    diskCollector.Initialize();

    // Ініціалізуємо колектор Network
    networkCollector.Initialize();

    // Запускаємо таймер для періодичного оновлення
даних
    updateTimer.Start();
}

// Метод для ініціалізації графіків
private void InitializeCharts()
{
    // Налаштування графіка CPU
    if (chartCpu != null)
    {
        chartCpu.Series.Clear();
        chartCpu.Series.Add("CPU");
        chartCpu.Series["CPU"].ChartType =
SeriesChartType.Line;
        chartCpu.Series["CPU"].Color =
System.Drawing.Color.Blue;
        chartCpu.Series["CPU"].BorderWidth = 3;
        chartCpu.ChartAreas[0].AxisY.Minimum = 0;
        chartCpu.ChartAreas[0].AxisY.Maximum = 100;
        chartCpu.ChartAreas[0].AxisX.Title = "Час (с)";
        chartCpu.ChartAreas[0].AxisY.Title =
"Завантаження (%)";
    }
}

```



```

// Налаштування графіка RAM
if (chartRam != null)
{
    chartRam.Series.Clear();
    chartRam.Series.Add("RAM");
    chartRam.Series["RAM"].ChartType =
SeriesChartType.Line;
    chartRam.Series["RAM"].Color =
System.Drawing.Color.Green;
    chartRam.Series["RAM"].BorderWidth = 3;
    chartRam.ChartAreas[0].AxisY.Minimum = 0;
    chartRam.ChartAreas[0].AxisY.Maximum = 100;
    chartRam.ChartAreas[0].AxisX.Title = "Час (с)";
    chartRam.ChartAreas[0].AxisY.Title =
"Використання (%)";
}

// Налаштування графіка Disk
if (chartDisk != null)
{
    chartDisk.Series.Clear();
    chartDisk.Series.Add("Disk");
    chartDisk.Series["Disk"].ChartType =
SeriesChartType.Line;
    chartDisk.Series["Disk"].Color =
System.Drawing.Color.Orange;
    chartDisk.Series["Disk"].BorderWidth = 3;
    chartDisk.ChartAreas[0].AxisY.Minimum = 0;
    chartDisk.ChartAreas[0].AxisY.Maximum = 100;
    chartDisk.ChartAreas[0].AxisX.Title = "Час (с)";
    chartDisk.ChartAreas[0].AxisY.Title =
"Активність (%)";
}

// Налаштування графіка Network
if (chartNetwork != null)
{
    chartNetwork.Series.Clear();
    chartNetwork.Series.Add("Network");
    chartNetwork.Series["Network"].ChartType =
SeriesChartType.Line;
    chartNetwork.Series["Network"].Color =
System.Drawing.Color.Purple;
    chartNetwork.Series["Network"].BorderWidth = 3;
    chartNetwork.ChartAreas[0].AxisY.Minimum = 0;
    chartNetwork.ChartAreas[0].AxisY.Maximum = 100;
    chartNetwork.ChartAreas[0].AxisX.Title = "Час
(с)";
    chartNetwork.ChartAreas[0].AxisY.Title =
"Активність (%)";
}

```

```

        // Налаштування графіка інтегрального показника
        if (chartIntegrated != null)
        {
            chartIntegrated.Series.Clear();
            chartIntegrated.Series.Add("Performance");
            chartIntegrated.Series["Performance"].ChartType
= SeriesChartType.Line;
            chartIntegrated.Series["Performance"].Color =
System.Drawing.Color.Red;

            chartIntegrated.Series["Performance"].BorderWidth = 3;
            chartIntegrated.ChartAreas[0].AxisY.Minimum = 0;
            chartIntegrated.ChartAreas[0].AxisY.Maximum =
100;
            chartIntegrated.ChartAreas[0].AxisX.Title = "Час
(c)";
            chartIntegrated.ChartAreas[0].AxisY.Title =
"Продуктивність";
        }
    }

    // Метод для зупинки моніторингу
    private void StopMonitoring()
    {
        // Реалізація буде виконана на наступному етапі
    }

    // Обробник події таймера для оновлення даних
    private void UpdateTimer_Tick(object sender, EventArgs
e)
    {
        // Оновлюємо відображення поточних показників
        UpdateCurrentIndicators();

        // Оновлюємо графіки
        UpdateCharts();

        // Оновлюємо інтегральну оцінку
        UpdateIntegratedScore();

        // Оновлюємо статистику кожні 5 секунд
        statisticsUpdateCounter++;
        if (statisticsUpdateCounter >= 5)
        {
            UpdateStatistics();
            statisticsUpdateCounter = 0;
        }

        // Оновлюємо рекомендації кожні 10 секунд
        recommendationsUpdateCounter++;
        if (recommendationsUpdateCounter >= 10)
        {

```

```
        UpdateRecommendations();
        recommendationsUpdateCounter = 0;
    }
}

// Змінні для зберігання поточних значень
private double currentCpuUsage;
private double currentRamUsage;
private double currentDiskUsage;
private double currentNetworkUsage;

// Метод для оновлення відображення поточних показників
private void UpdateCurrentIndicators()
{
    // Отримуємо поточне завантаження CPU
    currentCpuUsage = cpuCollector.GetCurrentUsage();

    // Оновлюємо текст та прогрес-бар для CPU
    // Перевіряємо, чи існують елементи управління (для
безпеки)
    if (lblCpu != null)
    {
        lblCpu.Text = $"CPU: {currentCpuUsage:F1}%";
    }

    if (progressBarCpu != null)
    {
        progressBarCpu.Value = (int)currentCpuUsage;
    }

    // Отримуємо поточне використання RAM
    currentRamUsage = ramCollector.GetCurrentUsage();

    // Оновлюємо текст та прогрес-бар для RAM
    if (lblRam != null)
    {
        lblRam.Text = $"RAM: {currentRamUsage:F1}%";
    }

    if (progressBarRam != null)
    {
        progressBarRam.Value = (int)currentRamUsage;
    }

    // Отримуємо поточну активність диска
    currentDiskUsage = diskCollector.GetCurrentUsage();

    // Оновлюємо текст та прогрес-бар для Disk
    if (lblDisk != null)
    {
        lblDisk.Text = $"Disk: {currentDiskUsage:F1}%";
    }
}
```

```

        if (progressBarDisk != null)
        {
            progressBarDisk.Value = (int)currentDiskUsage;
        }

        // Отримуємо поточну мережеву активність
        currentNetworkUsage =
networkCollector.GetCurrentUsage();

        // Оновлюємо текст та прогрес-бар для Network
        if (lblNetwork != null)
        {
            lblNetwork.Text = $"Network:
{currentNetworkUsage:F1}%";
        }

        if (progressBarNetwork != null)
        {
            progressBarNetwork.Value =
(int)currentNetworkUsage;
        }
    }

    // Метод для оновлення інтегральної оцінки
продуктивності
    private void UpdateIntegratedScore()
    {
        // Використовуємо вже отримані значення
        double performanceScore =
analyzer.CalculateIntegratedPerformance(currentCpuUsage,
currentRamUsage, currentDiskUsage, currentNetworkUsage);

        // Оновлюємо текст та прогрес-бар для інтегральної
оцінки
        if (lblIntegratedValue != null)
        {
            lblIntegratedValue.Text =
$"{{performanceScore:F1}}";
        }

        if (progressBarIntegrated != null)
        {
            progressBarIntegrated.Value =
(int)performanceScore;
        }
    }

    // Метод для оновлення графіків
    private void UpdateCharts()
    {
        // Використовуємо вже отримані значення

```

```

        double cpuUsage = currentCpuUsage;
        double ramUsage = currentRamUsage;
        double diskUsage = currentDiskUsage;
        double networkUsage = currentNetworkUsage;

        // Розраховуємо інтегральний показник продуктивності
        double performanceScore =
analyzer.CalculateIntegratedPerformance(cpuUsage, ramUsage,
diskUsage, networkUsage);

        // Додаємо поточні значення в історію для графіків
        cpuHistory.Add(cpuUsage);
        ramHistory.Add(ramUsage);
        diskHistory.Add(diskUsage);
        networkHistory.Add(networkUsage);
        performanceHistory.Add(performanceScore);

        // Додаємо вимірювання в менеджер статистики для
подальшого аналізу
        statisticsManager.AddCpuMeasurement(cpuUsage);
        statisticsManager.AddRamMeasurement(ramUsage);

statisticsManager.AddPerformanceMeasurement(performanceScore);

        // Додаємо показник продуктивності в калькулятор
стабільності
        stabilityCalculator.AddValue(performanceScore);

        // Збільшуємо лічильник часу
        timeCounter++;

        // Якщо кількість точок перевищує максимум,
видаляємо найстаріші
        if (cpuHistory.Count > MAX_CHART_POINTS)
        {
            cpuHistory.RemoveAt(0);
            ramHistory.RemoveAt(0);
            diskHistory.RemoveAt(0);
            networkHistory.RemoveAt(0);
            performanceHistory.RemoveAt(0);
        }

        // Оновлюємо графік CPU
        if (chartCpu != null)
        {
            chartCpu.Series["CPU"].Points.Clear();
            for (int i = 0; i < cpuHistory.Count; i++)
            {

chartCpu.Series["CPU"].Points.AddXY(timeCounter -
cpuHistory.Count + i, cpuHistory[i]);
            }

```

```

    }

    // Оновлюємо графік RAM
    if (chartRam != null)
    {
        chartRam.Series["RAM"].Points.Clear();
        for (int i = 0; i < ramHistory.Count; i++)
        {

chartRam.Series["RAM"].Points.AddXY(timeCounter -
ramHistory.Count + i, ramHistory[i]);
        }
    }

    // Оновлюємо графік Disk
    if (chartDisk != null)
    {
        chartDisk.Series["Disk"].Points.Clear();
        for (int i = 0; i < diskHistory.Count; i++)
        {

chartDisk.Series["Disk"].Points.AddXY(timeCounter -
diskHistory.Count + i, diskHistory[i]);
        }
    }

    // Оновлюємо графік Network
    if (chartNetwork != null)
    {
        chartNetwork.Series["Network"].Points.Clear();
        for (int i = 0; i < networkHistory.Count; i++)
        {

chartNetwork.Series["Network"].Points.AddXY(timeCounter -
networkHistory.Count + i, networkHistory[i]);
        }
    }

    // Оновлюємо графік інтегрального показника
    if (chartIntegrated != null)
    {

chartIntegrated.Series["Performance"].Points.Clear();
        for (int i = 0; i < performanceHistory.Count;
i++)
        {

chartIntegrated.Series["Performance"].Points.AddXY(timeCounter -
performanceHistory.Count + i, performanceHistory[i]);
        }
    }
}

```

```

// Метод для оновлення статистики
private void UpdateStatistics()
{
    // Отримуємо статистичні дані для CPU
    double cpuAvg = statisticsManager.GetCpuAverage();
    double cpuMin = statisticsManager.GetCpuMinimum();
    double cpuMax = statisticsManager.GetCpuMaximum();

    // Отримуємо статистичні дані для RAM
    double ramAvg = statisticsManager.GetRamAverage();
    double ramMin = statisticsManager.GetRamMinimum();
    double ramMax = statisticsManager.GetRamMaximum();

    // Отримуємо статистичні дані для продуктивності
    double perfAvg =
statisticsManager.GetPerformanceAverage();
    double perfMin =
statisticsManager.GetPerformanceMinimum();
    double perfMax =
statisticsManager.GetPerformanceMaximum();

    // Отримуємо індекс стабільності
    double stabilityIndex =
stabilityCalculator.CalculateStabilityIndex();

    // Оновлюємо відображення статистики CPU
    if (lblCpuAverage != null)
    {
        lblCpuAverage.Text = $"{cpuAvg:F1}%";
    }
    if (lblCpuMinimum != null)
    {
        lblCpuMinimum.Text = $"{cpuMin:F1}%";
    }
    if (lblCpuMaximum != null)
    {
        lblCpuMaximum.Text = $"{cpuMax:F1}%";
    }

    // Оновлюємо відображення статистики RAM
    if (lblRamAverage != null)
    {
        lblRamAverage.Text = $"{ramAvg:F1}%";
    }
    if (lblRamMinimum != null)
    {
        lblRamMinimum.Text = $"{ramMin:F1}%";
    }
    if (lblRamMaximum != null)
    {
        lblRamMaximum.Text = $"{ramMax:F1}%";
    }
}

```

```

    }

    // Оновлюємо відображення статистики продуктивності
    if (lblPerfAverage != null)
    {
        lblPerfAverage.Text = $"{perfAvg:F1}";
    }
    if (lblPerfMinimum != null)
    {
        lblPerfMinimum.Text = $"{perfMin:F1}";
    }
    if (lblPerfMaximum != null)
    {
        lblPerfMaximum.Text = $"{perfMax:F1}";
    }

    // Оновлюємо відображення індексу стабільності
    if (lblStabilityIndex != null)
    {
        lblStabilityIndex.Text = $"{stabilityIndex:F1}";
    }
}

// Метод для оновлення рекомендацій
private void UpdateRecommendations()
{
    // Отримуємо поточні показники
    double cpuUsage = cpuCollector.GetCurrentUsage();
    double ramUsage = ramCollector.GetCurrentUsage();
    double diskUsage = diskCollector.GetCurrentUsage();
    double networkUsage =
networkCollector.GetCurrentUsage();

    // Отримуємо рекомендації від системи рекомендацій
    List<string> recommendations =
recommendationSystem.GetRecommendations(cpuUsage, ramUsage,
diskUsage, networkUsage);

    // Оновлюємо відображення рекомендацій
    if (txtRecommendations != null)
    {
        // Об'єднуємо всі рекомендації в один текст з
новими рядками
        txtRecommendations.Text =
string.Join(Environment.NewLine, recommendations);
    }
}

// Метод для очищення ресурсів при закритті форми
protected override void
OnFormClosing(FormClosingEventArgs e)
{

```



```

        // Зупинка таймера
        updateTimer?.Stop();

        // Очищення ресурсів колекторів
        cpuCollector?.Dispose();
        ramCollector?.Dispose();
        diskCollector?.Dispose();
        networkCollector?.Dispose();

        base.OnFormClosing(e);
    }
}

```

StabilityIndexCalculator.cs

```

using System;
using System.Collections.Generic;
using System.Linq;

namespace PerformanceMonitor
{
    // Клас для розрахунку індексу стабільності системи
    // Аналізує плавність зміни показників та виявляє різкі
    скачки
    public class StabilityIndexCalculator
    {
        // Список для зберігання історії показників для
        розрахунку стабільності
        private List<double> valueHistory;

        // Максимальна кількість значень для розрахунку
        стабільності
        private const int MAX_HISTORY_SIZE = 50;

        // Попіг для виявлення різких скачків (у відсотках)
        private const double JUMP_THRESHOLD = 20.0;

        // Конструктор класу
        public StabilityIndexCalculator()
        {
            // Ініціалізуємо список історії
            valueHistory = new List<double>();
        }

        // Метод для додавання нового значення в історію
        public void AddValue(double value)
        {
            // Додаємо нове значення в історію
            valueHistory.Add(value);
        }
    }
}

```

```

        // Якщо кількість значень перевищує максимум,
        видаляємо найстаріше
        if (valueHistory.Count > MAX_HISTORY_SIZE)
        {
            valueHistory.RemoveAt(0);
        }
    }

    // Метод для розрахунку індексу стабільності
    // Повертає значення від 0 до 100, де 100 - максимальна
    стабільність
    public double CalculateStabilityIndex()
    {
        // Якщо замало даних для розрахунку, повертаємо 100
        (максимальна стабільність)
        if (valueHistory.Count < 2)
        {
            return 100;
        }

        // Розраховуємо середнє відхилення між сусідніми
        значеннями
        double avgDeviation = CalculateAverageDeviation();

        // Підраховуємо кількість різких скачків
        int jumpCount = CountJumps();

        // Розраховуємо індекс стабільності
        // Чим менше відхилення та скачків, тим вищий індекс
        // Базовий індекс - 100
        // Зменшуємо за середнє відхилення (максимум - 30
        балів)
        // Зменшуємо за кількість скачків (максимум - 70
        балів)
        double deviationPenalty = (avgDeviation / 50.0) *
        30.0; // Максимальне відхилення 50% дає 30 балів штрафу
        double jumpPenalty = ((double)jumpCount /
        valueHistory.Count) * 70.0; // Кожен скачок дає штраф

        double stabilityIndex = 100 - deviationPenalty -
        jumpPenalty;

        // Обмежуємо результат діапазоном від 0 до 100
        if (stabilityIndex < 0)
        {
            return 0;
        }
        if (stabilityIndex > 100)
        {
            return 100;
        }
    }

```

```

        return stabilityIndex;
    }

    // Метод для розрахунку середнього відхилення між
    // сусідніми значеннями
    private double CalculateAverageDeviation()
    {
        // Якщо замало даних, повертаємо 0
        if (valueHistory.Count < 2)
        {
            return 0;
        }

        // Розраховуємо відхилення між кожною парою сусідніх
        // значень
        double totalDeviation = 0;
        for (int i = 1; i < valueHistory.Count; i++)
        {
            double deviation = Math.Abs(valueHistory[i] -
            valueHistory[i - 1]);
            totalDeviation += deviation;
        }

        // Повертаємо середнє відхилення
        return totalDeviation / (valueHistory.Count - 1);
    }

    // Метод для підрахунку кількості різких скачків
    private int CountJumps()
    {
        // Якщо замало даних, повертаємо 0
        if (valueHistory.Count < 2)
        {
            return 0;
        }

        int jumpCount = 0;

        // Перевіряємо кожну пару сусідніх значень
        for (int i = 1; i < valueHistory.Count; i++)
        {
            double deviation = Math.Abs(valueHistory[i] -
            valueHistory[i - 1]);

            // Якщо відхилення перевищує поріг, вважаємо це
            // скачком
            if (deviation > JUMP_THRESHOLD)
            {
                jumpCount++;
            }
        }
    }

```

```
        return jumpCount;
    }

    // Метод для очищення історії
    public void ClearHistory()
    {
        // Очищаємо список історії
        valueHistory.Clear();
    }
}
}
```