



Метадані

ДОКУМЕНТ

Заголовок

Бак_пояснювальна_Дворник

Автор

Дворник

Науковий керівник / Експерт

ІД документу

334320079

ОРГАНІЗАЦІЯ

Назва організації

Taras Shevchenko National University of Luhansk

підрозділ

Taras Shevchenko National University of Luhansk

ЗВІТ

Дата звіту

6/15/2026

Дата редагування

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.

3.89%

3.89%

КП 1

15919

Кількість слів

5.95%

5.95%

КЦ

130893

Кількість символів

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв



1

Інтервали



0

Мікропробіли



98

Білі знаки



0

Парафрази (SmartMarks)



41

Джерела

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз

Колір тексту

#	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	https://github.com/turinyura/OZPapplications	27 (0.17 %)

2	https://github.com/turinyura/OZPapplications	24 (0.15 %)
3	https://pi.sfu-kras.ru/files/vkr_volhin_ae.pdf	18 (0.11 %)
4	https://blog.csdn.net/m0_74474733/article/details/127729071	15 (0.09 %)
5	https://github.com/turinyura/OZPapplications	14 (0.09 %)
6	https://blog.csdn.net/m0_74474733/article/details/127729071	13 (0.08 %)
7	https://github.com/turinyura/OZPapplications	12 (0.08 %)
8	https://github.com/Nasserahm/UML-3	12 (0.08 %)
9	https://blog.csdn.net/Traveller_man/article/details/140590815	12 (0.08 %)
10	http://moodle2.snu.edu.ua/mod/resource/view.php?id=141345	12 (0.08 %)

База даних RefBooks



#	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
---	-----------	--

Домашня база даних



#	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
---	-----------	--

Програма обміну базами даних (0.54 %)



#	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
---	-----------	--

1	Розробка застосунку для аналізу використання криптографічного методу захисту інформації у різних програмних середовищах 5/31/2026 Odessa National Polytechnic University (ІІБРТ, Каф. кібербезпеки та програмного забезпечення)	20 (4) (0.13 %)
2	IT-M24-11353357 7/7/2025 Bohdan Khmelnytsky National University of Cherkasy (Кафедра прикладної математики та інформатики)	17 (2) (0.11 %)
3	ВКР_Кондратенко 12/13/2025 State University of Trade and Economics (Кафедра комп'ютерних наук та інформаційних систем)	15 (2) (0.09 %)
4	Фесік_Система керування віддаленим сервером з веб-орієнтованим інтерфейсом 6/5/2025 Khmelnytskyi National University (Кафедра комп'ютерної інженерії та інформаційних систем)	13 (2) (0.08 %)
5	Розробка програмного засобу для виявлення атак на промислові системи керування (ICS) з використанням автоенкодерів 6/2/2025 Vinnytsya National Technical University (МБІС)	8 (1) (0.05 %)
6	ФКНТ_2026_СП_Дацюк НД 6/1/2026 Ukrainian national aviation university (ФКНТ Кафедра інтелектуальних кібернетичних систем)	7 (1) (0.04 %)
7	HryshkevychMM_TV32mp_magistr_2024 12/10/2024 National Technical University of Ukraine Igor Sikorskyi Kyiv Politech Institute (ІАТЕ, К-ра інженерії програмного забезпечення в сфері енергетики)	6 (1) (0.04 %)

Інтернет (3.35 %)



#	ДЖЕРЕЛО URL	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
8	https://blog.csdn.net/m0_74474733/article/details/127729071	257 (25) (1.61 %)
9	https://github.com/turinyura/OZPapplications	117 (9) (0.73 %)
10	https://github.com/Nasserahm/UML-3	35 (4) (0.22 %)
11	https://ipi.sfu-kras.ru/files/vkr_volhin_ae.pdf	29 (3) (0.18 %)
12	https://github.com/lanirysss/automat_recall	21 (3) (0.13 %)
13	https://dev.to/devasservice/customtkinter-a-complete-tutorial-4527	20 (2) (0.13 %)
14	http://moodle2.snu.edu.ua/mod/resource/view.php?id=141345	12 (1) (0.08 %)
15	https://blog.csdn.net/Traveller_man/article/details/140590815	12 (1) (0.08 %)
16	https://duikt.edu.ua/repositorii/ipz/2022/%D0%9F%D0%94%2043/%D0%A6%D0%B0%D1%82%D1%83%D1%80%D1%8F%D0%BD%20%D0%95.%D0%A0..pdf	11 (1) (0.07 %)
17	https://blog.csdn.net/xmhj666/article/details/133089294	10 (1) (0.06 %)
18	https://eir.kntu.net.ua/jspui/bitstream/123456789/1215/1/%D0%AF%D1%86%D0%BA%D0%B5%D0%B2%D0%B8%D1%87%20%D0%90.%20%D0%92.%20%D0%9F%D1%80%D0%BE%D1%94%D0%BA%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F%20%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE%20%D1%81%D0%B5%D1%80%D0%B2%D1%96%D1%81%D1%83%20%D1%81%D0%BA%D0%BB%D0%B0%D0%B4%D0%B0%D0%BD%D0%BD%D1%8F%20%D0%BF%D0%BE%D0%B7%D0%BE%D0%B2%D0%BD%D0%B8%D1%85%20%D0%B7%D0%B0%D1%8F%D0%B2%20C2%ABProshuSud%C2%BB.pdf	10 (1) (0.06 %)



Список прийнятих фрагментів

#	ЗМІСТ	КІЛЬКІСТЬ ОДНАКОВИХ СЛІВ (ФРАГМЕНТІВ)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЗ «ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА»

Факультет технологій та інформаційних систем
(назва ¹⁴факультету, інституту).
Кафедра інформаційних технологій та систем
(назва кафедри) Пояснювальна записка до кваліфікаційної роботи
за першим (бакалаврським) рівнем освіти

на тему:
Розробка програмного засобу для моніторингу використання ресурсів комп'ютера

Виконав: здобувач вищої освіти ¹⁸курсу спеціальності
F2 «Інженерія програмного забезпечення».
(шифр і назва напрямку підготовки, спеціальності)

Ростислав ДВОРНИК
¹⁶(прізвище та ініціали) Керівник Микола СЕМЕНОВ
(прізвище та ініціали) Рецензент Владислав ІЩЕНКО
(прізвище та ініціали)

ЗМІСТ

ВСТУП 3

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМИ МОНІТОРИНГУ РЕСУРСІВ КОМП'ЮТЕРНИХ СИСТЕМ 7

- 1.1. Поняття та призначення моніторингу ресурсів комп'ютера 7
- 1.2. Огляд існуючих програмних засобів моніторингу системних ресурсів 12
- 1.3. Аналіз технологій і бібліотек для реалізації системного моніторингу 17
- 1.4. Постановка задачі розробки програмного засобу 22

РОЗДІЛ 2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАСОБУ МОНІТОРИНГУ РЕСУРСІВ 24

- 2.1. Загальна архітектура програмної системи 24
- 2.2. Моделювання структури та компонентів системи 28
- 2.3. Проєктування алгоритмів збору та обробки системної інформації 39

РОЗДІЛ 3 РЕАЛІЗАЦІЯ ТА АПРОБАЦІЯ ПРОГРАМНОГО ЗАСОБУ 48

- 3.1. Реалізація програмного засобу для моніторингу системних ресурсів 48
- 3.2. Реалізація графічного інтерфейсу користувача 59
- 3.3. Тестування та апробація програмного засобу 65

ВИСНОВКИ 68

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 71

Додаток А. Код класу DataCollector з функцією 75

Додаток Б Код для роботи з RAM 78

Додаток В. Код моніторингу дискової активності 81

Додаток Г. Код для роботи з NET 83

Додаток Д. Код графічного інтерфейсу 85

ВСТУП

Актуальність теми розробки програмних засобів для моніторингу використання ресурсів комп'ютера зумовлена постійним зростанням складності сучасних комп'ютерних систем та програмного забезпечення. Сучасні операційні системи одночасно виконують велику кількість процесів і сервісів, що активно використовують процесорний час, оперативну пам'ять, дискові підсистеми та мережеві ресурси. У таких умовах ефективний контроль за станом системних ресурсів стає важливим фактором забезпечення стабільності роботи комп'ютера, своєчасного виявлення переважностей та оптимізації використання апаратних можливостей системи. Особливої актуальності ця проблема набуває під час розробки програмного забезпечення, адміністрування комп'ютерних систем та проведення навчальних або дослідницьких експериментів, коли необхідно отримувати достовірну інформацію про стан ресурсів у реальному часі.

Значну роль у вирішенні таких задач відіграють програмні інструменти моніторингу, які дозволяють автоматично збирати, аналізувати та відображати інформацію про стан комп'ютерної системи. Існує велика кількість готових програм для системного моніторингу, однак вони не завжди забезпечують необхідну гнучкість, можливість модифікації або інтеграції з іншими програмними системами. У цьому контексті особливо перспективним є використання мови програмування Python, яка має розвинену екосистему бібліотек для роботи з операційною системою та апаратними ресурсами комп'ютера. Зокрема, бібліотека psutil дозволяє отримувати детальну інформацію про процеси, використання процесора, пам'яті, дисків та мережевих інтерфейсів, що значно спрощує розробку подібних програмних засобів.

Крім того, Python характеризується простотою синтаксису, високою швидкістю розробки програм та наявністю великої кількості інструментів для створення графічних інтерфейсів і засобів візуалізації даних. Це робить його ефективним інструментом для створення прикладних систем моніторингу, які можуть використовуватися як у навчальному процесі, так і для практичних задач аналізу продуктивності комп'ютерних систем. Розробка власного програмного засобу моніторингу дозволяє глибше зрозуміти принципи функціонування операційних систем, механізми отримання системної інформації та методи обробки і візуалізації даних.

Таким чином, актуальність теми дослідження зумовлена необхідністю створення гнучких та доступних програмних інструментів для аналізу використання системних ресурсів, а також можливістю використання сучасних засобів програмування для реалізації таких систем. Розробка програмного засобу моніторингу ресурсів комп'ютера на основі Python є доцільною з точки зору навчальних, дослідницьких та практичних потреб, що і обґрунтовує вибір теми бакалаврської роботи.

Метою роботи є розробка програмного засобу для моніторингу використання ресурсів комп'ютера з використанням мови програмування Python та спеціалізованих бібліотек для отримання і обробки системної інформації.

Для досягнення поставленої мети необхідно вирішити такі завдання:

1. проаналізувати сучасні підходи та програмні засоби моніторингу системних ресурсів;
2. дослідити можливості мови програмування Python і відповідних бібліотек для отримання інформації про стан комп'ютерної системи;
3. сформулювати функціональні вимоги до програмного засобу; розробити архітектуру та структуру програмної системи; реалізувати модулі збору та обробки інформації про використання процесора, пам'яті, дисків та інших ресурсів;
4. створити інтерфейс користувача для відображення результатів моніторингу; провести тестування та апробацію розробленого програмного засобу.

Об'єктом дослідження є процес моніторингу використання ресурсів комп'ютерної системи.

Предметом дослідження є програмні методи та засоби збору, обробки і відображення інформації про використання апаратних ресурсів комп'ютера.

У процесі виконання роботи застосовуються методи аналізу науково-технічних джерел, порівняльний аналіз існуючих програмних рішень, методи системного аналізу та проєктування програмних систем, а також методи алгоритмізації та програмної реалізації. Як основні інструменти дослідження використано мову програмування Python, бібліотеки psutil та platform для отримання системної інформації, а також додаткові програмні засоби для створення інтерфейсу користувача та візуалізації даних.

У першому розділі роботи проведено аналіз проблеми моніторингу ресурсів комп'ютерних систем, розглянуто основні поняття та принципи системного моніторингу, а також досліджено існуючі програмні інструменти, що використовуються для контролю стану процесора, оперативної пам'яті, дискової підсистеми та мережевих ресурсів. Крім того, проаналізовано можливості сучасних технологій програмування та бібліотек Python, які можуть бути використані для реалізації програмних засобів моніторингу.

У другому розділі роботи розглянуто процес проектування програмного засобу моніторингу використання ресурсів комп'ютера. У цьому розділі обґрунтовано архітектуру системи, визначено основні програмні модулі, описано їх функціональне призначення та взаємодію між ними, а також розроблено алгоритми збору та обробки інформації про стан системних ресурсів. Окрему увагу приділено проектуванню структури інтерфейсу користувача та принципам відображення результатів моніторингу.

У третьому розділі представлено реалізацію розробленого програмного засобу на мові програмування Python із використанням відповідних бібліотек для отримання інформації про стан комп'ютерної системи. Описано структуру програмного коду, реалізацію основних функціональних модулів, а також створення інтерфейсу користувача для відображення результатів моніторингу. Також наведено результати тестування та апробації програмного засобу, що підтверджують його працездатність та можливість використання для аналізу використання ресурсів комп'ютера.

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМИ МОНІТОРИНГУ РЕСУРСІВ КОМП'ЮТЕРНИХ СИСТЕМ

1.1 Поняття та призначення моніторингу ресурсів комп'ютера

Моніторинг ресурсів комп'ютера є важливим елементом забезпечення ефективного функціонування сучасних комп'ютерних систем. У загальному розумінні моніторингом називають процес постійного або періодичного спостереження за станом апаратних і програмних компонентів системи з метою отримання інформації про їх використання, продуктивність та можливі відхилення від нормального режиму роботи. У контексті комп'ютерних систем моніторинг ресурсів передбачає збір, аналіз та відображення інформації про використання центрального процесора, оперативної пам'яті, дискових підсистем, мережевих інтерфейсів та інших апаратних ресурсів. Такий контроль дозволяє отримувати об'єктивну картину роботи системи, своєчасно виявляти перевантаження, помилки або неефективне використання ресурсів, а також приймати рішення щодо оптимізації роботи програмного забезпечення [18].

Рис. 1.1. Процес моніторингу ресурсів комп'ютера

Сучасні операційні системи є складними багатозадачними середовищами, у яких одночасно виконуються десятки або навіть сотні процесів. Кожен із цих процесів використовує певну частину системних ресурсів, що може призводити до конкуренції за процесорний час, пам'ять або інші апаратні можливості. У таких умовах моніторинг ресурсів виконує важливу функцію контролю та аналізу стану системи, забезпечуючи користувачів, системних адміністраторів та розробників програмного забезпечення необхідною інформацією про продуктивність та стабільність роботи комп'ютера. Зокрема, аналіз завантаження центрального процесора дозволяє оцінити рівень використання обчислювальних ресурсів, а контроль використання оперативної пам'яті дає можливість виявити процеси, які споживають надмірну кількість пам'яті або спричиняють її витоки [4].

Особливу роль моніторинг відіграє у процесі адміністрування комп'ютерних систем і мереж. Завдяки систематичному збору даних про стан ресурсів можна вчасно реагувати на зниження продуктивності, виявляти аномальну поведінку програм або апаратних компонентів, а також прогнозувати можливі проблеми у роботі системи. Наприклад, підвищене використання дискової підсистеми або мережевих ресурсів може свідчити про активність певних програм або сервісів, тоді як різке зростання навантаження на процесор може бути ознакою помилок у програмному забезпеченні або виконання ресурсоємних обчислень. Таким чином, моніторинг є одним із ключових інструментів підтримки стабільної роботи комп'ютерних систем та забезпечення їх ефективного використання [11].

Окрім практичних аспектів експлуатації комп'ютерних систем, моніторинг ресурсів має важливе значення у процесі розробки програмного забезпечення. Під час створення нових програм необхідно оцінювати їх вплив на системні ресурси, визначати ефективність алгоритмів та оптимізувати використання пам'яті або процесорного часу. У цьому контексті інструменти моніторингу дозволяють розробникам аналізувати поведінку програм у реальних умовах виконання, що сприяє підвищенню якості програмних продуктів. Отримані дані можуть використовуватися для профілювання програм, пошуку вузьких місць у коді та покращення загальної продуктивності системи [9].

Системний моніторинг ресурсів комп'ютера є невід'ємною складовою сучасних інформаційних систем, оскільки забезпечує отримання об'єктивної інформації про стан апаратних і програмних компонентів у процесі їх функціонування. Зі зростанням складності програмного забезпечення та збільшенням кількості одночасно виконуваних процесів виникає необхідність у постійному контролі використання обчислювальних ресурсів. Без належного моніторингу система може працювати неефективно, що проявляється у перевантаженні процесора, нестачі оперативної пам'яті, затримках у доступі до дискових або мережевих ресурсів. У таких умовах системний моніторинг виступає як інструмент забезпечення стабільності, продуктивності та надійності функціонування комп'ютерної системи.

Обґрунтування необхідності системного моніторингу базується на тому, що сучасні операційні системи функціонують у режимі багатозадачності, де ресурси розподіляються між великою кількістю процесів і служб. Відсутність контролю за їх використанням ускладнює виявлення причин зниження продуктивності, появи збоїв або неефективного використання ресурсів. Крім того, системний моніторинг є важливим інструментом як для кінцевих користувачів, так і для системних адміністраторів та розробників програмного забезпечення, оскільки дозволяє отримувати актуальні дані про стан системи, аналізувати динаміку змін та приймати обґрунтовані рішення щодо оптимізації її роботи. Таким чином, моніторинг ресурсів виступає не лише засобом спостереження, але й основою для діагностики, прогнозування та управління станом комп'ютерної системи.

У межах системного моніторингу можна виділити низку ключових задач, реалізація яких забезпечує повноцінне функціонування відповідних програмних засобів:

1. збір інформації про використання системних ресурсів, зокрема центрального процесора, оперативної пам'яті, дискових накопичувачів та мережевих інтерфейсів у режимі реального часу або з заданою періодичністю;
2. обробка та агрегація отриманих даних з метою їх подальшого аналізу, включаючи обчислення середніх значень, пікових навантажень та інших показників продуктивності;
3. відображення інформації у зручній для користувача формі, зокрема у вигляді таблиць, діаграм або графіків, що дозволяє швидко оцінити стан системи;
4. виявлення аномалій та відхилень у роботі системи, таких як надмірне використання ресурсів окремими процесами або різкі зміни

навантаження;

5. ведення журналу (логування) змін стану системних ресурсів для подальшого аналізу та дослідження динаміки роботи системи;
6. підтримка прийняття рішень щодо оптимізації роботи системи, зокрема шляхом виявлення «вузьких місць» у використанні ресурсів;
7. забезпечення можливості розширення та інтеграції з іншими програмними системами або інструментами аналізу.

Задачі системного моніторингу охоплюють повний цикл роботи з інформацією про стан комп'ютерної системи - від її збору до інтерпретації та використання для прийняття рішень. Реалізація цих задач у вигляді програмного засобу дозволяє створити ефективний інструмент контролю та аналізу використання ресурсів, що є важливим як у практичній діяльності, так і в навчальному процесі.

Таким чином, моніторинг ресурсів комп'ютера є важливою складовою сучасних інформаційних технологій, що забезпечує можливість контролю, аналізу та оптимізації використання апаратних ресурсів комп'ютерної системи. Його застосування дозволяє підвищити ефективність роботи програмного забезпечення, своєчасно виявляти проблеми у функціонуванні системи та забезпечувати стабільність її роботи. Саме тому розробка програмних засобів моніторингу, зокрема із використанням сучасних мов програмування та бібліотек для отримання системної інформації, є актуальним напрямом досліджень у галузі програмної інженерії.

1.2. Огляд існуючих програмних засобів моніторингу системних ресурсів

Сучасні програмні засоби моніторингу системних ресурсів представлені широким спектром інструментів, які відрізняються за функціональністю, рівнем деталізації даних, інтерфейсом користувача та можливостями інтеграції. Вибір конкретного інструменту залежить від потреб користувача, типу операційної системи, а також завдань, що вирішуються - від базового контролю завантаження ресурсів до глибокого аналізу продуктивності системи. Найбільш поширеними є як вбудовані засоби операційних систем, так і спеціалізовані програмні рішення, що забезпечують розширені можливості моніторингу та аналізу.

Одним із базових інструментів є диспетчер завдань Windows (Windows Task Manager), який входить до складу операційної системи Windows і надає користувачу інформацію про запущені процеси, завантаження центрального процесора, використання оперативної пам'яті, дискову та мережеву активність. Його основною перевагою є доступність та простота використання, що робить його зручним для повсякденного контролю стану системи. Водночас, функціональність цього інструменту обмежена, а можливості детального аналізу та розширення є мінімальними. У контексті моніторингу використання ресурсів центрального процесора (CPU) Windows Task Manager (рис. 1.2) надає дані не лише про загальний відсоток завантаження, а й про логічні процесори, швидкість роботи у ГГц, кількість потоків та дескрипторів, що дозволяє виявити аномальну активність конкретних додатків [19]. Аналіз оперативної пам'яті (RAM) у диспетчері завдань відображає структуру використання: активну пам'ять, зарезервовану апаратними засобами, а також кешовані дані. Важливим аспектом є моніторинг дискових підсистем, де Task Manager показує швидкість читання та запису, а також активний час диска, що критично для виявлення "вузьких місць" під час операцій введення-виведення. Для мережевих інтерфейсів інструмент демонструє пропускну здатність у реальному часі, розділяючи відправлені та отримані дані через Ethernet або Wi-Fi. Окрім основних ресурсів, сучасні версії дозволяють відстежувати стан графічного процесора (GPU), включаючи використання відеопам'яті та рівень кодування/декодування відео. Однак при використанні Windows Task Manager розробники програмних засобів моніторингу слід пам'ятати про "ефект спостерігача": сам диспетчер завдань є активним процесом, який споживає певну частку ресурсів ЦП та пам'яті для оновлення свого графічного інтерфейсу, що особливо помітно на малопотужних системах. Також він пропонує вкладку "Деталі" для поглибленого аналізу пріоритетів процесів та "Монітор ресурсів", який дає ще більш ґранульовану інформацію про використання файлів та мережевих підключень окремими модулями.

Рис. 1.2. Зовнішній вигляд Task Manager.

Аналогами у середовищі Linux є утиліти top та htop (рис. 1.3), які працюють у текстовому режимі та дозволяють відстежувати процеси в реальному часі. Зокрема, htop має більш зручний інтерфейс порівняно з top, підтримує кольорове відображення інформації та інтерактивне керування процесами, що підвищує ефективність роботи користувача.

Рис. 1.3. Утиліти top та htop [16]

Більш розширені можливості надає інструмент Process Explorer (рис. 1.4), розроблений компанією Microsoft як частина пакету Sysinternals. Process Explorer є значно потужнішим наступником стандартного Диспетчера завдань, пропонує глибоку інспекцію процесів у реальному часі [12].

Рис. 1.4. Зовнішній вигляд застосунку Process Explorer

Основна унікальність цього інструменту полягає в його дворівневій структурі відображення: у верхньому вікні відображається деревоподібний список активних процесів (що дозволяє бачити ієрархію «батько-нащадок»), а в нижньому - детальна інформація про дескриптори (handles) або завантажені динамічні бібліотеки (DLL), залежно від обраного режиму.

З точки зору моніторингу ресурсів, Process Explorer надає вичерпні дані про використання системної та відеопам'яті, причому він розрізняє приватну пам'ять процесу (Private Bytes) та віртуальний розмір, що критично для виявлення витоків пам'яті (memory leaks). Графіки продуктивності тут набагато детальніші: навівши курсор на будь-яку точку графіка CPU, можна миттєво побачити, який саме процес спричинив пікове навантаження в цей момент часу. Окрім стандартних метрик, програма дозволяє відстежувати цикли процесора (Cycles), переривання (Interrupts) та контекстні перемикання, що дає розробнику розуміння ефективності коду на низькому рівні.

Завдяки розширеному функціоналу цей інструмент широко використовується системними адміністраторами та розробниками для діагностики проблем у роботі програмного забезпечення.

Окрему категорію становлять сучасні системи моніторингу, орієнтовані на веб-інтерфейси та роботу у мережевому середовищі, такі як Netdata та Grafana. Netdata забезпечує моніторинг у режимі реального часу з високою деталізацією та інтерактивною візуалізацією показників, що дозволяє швидко виявляти проблеми у роботі системи. Grafana, у свою чергу, є потужною платформою для візуалізації даних, яка інтегрується з різними джерелами інформації (наприклад, Prometheus) і дозволяє створювати гнучкі інформаційні панелі (дашборди) для аналізу стану системи. Такі рішення широко використовуються у серверних середовищах та хмарних інфраструктурах, де необхідний централізований моніторинг великої кількості систем.

Порівнюючи зазначені інструменти за функціональністю, можна відзначити, що базові засоби, такі як Task Manager або top, забезпечують лише загальний контроль ресурсів, тоді як Process Explorer, Netdata та Grafana надають розширені можливості аналізу, включаючи деталізацію процесів, апаратні показники та історичні дані. З точки зору інтерфейсу користувача, інструменти можна поділити на консольні (top, htop) та графічні (Task Manager, Process Explorer, Netdata, Grafana), причому останні забезпечують більш наочне представлення інформації. Розширюваність також суттєво відрізняється: більшість вбудованих інструментів мають обмежені можливості розширення, тоді як Grafana та Netdata підтримують інтеграцію з іншими системами та налаштування під конкретні потреби. Щодо платформ, то Task Manager і Process Explorer орієнтовані на Windows, top і htop - на Unix-подібні системи, тоді як Netdata та Grafana є кросплатформними рішеннями. Таким чином, існуючі програмні засоби моніторингу системних ресурсів охоплюють широкий спектр можливостей - від простих інструментів для базового контролю до складних систем для глибокого аналізу та візуалізації даних. Проте жоден із розглянутих інструментів не є універсальним для всіх задач, що обґрунтовує доцільність розробки власного програмного засобу, адаптованого до конкретних вимог і умов використання.

1.3. Аналіз технологій і бібліотек для реалізації системного моніторингу

Реалізація програмних засобів системного моніторингу потребує використання відповідних технологій та інструментів, які забезпечують доступ до інформації про стан апаратних ресурсів і дозволяють ефективно обробляти та відображати ці дані. Одним із найбільш доцільних підходів до розробки таких систем є використання високорівневих мов програмування, які поєднують простоту реалізації з достатніми можливостями взаємодії з операційною системою. У цьому контексті мова програмування Python займає особливе місце завдяки своїй універсальності, великій кількості бібліотек та активній спільноті розробників. Python дозволяє швидко створювати прикладні програми, а також забезпечує доступ до системних ресурсів через спеціалізовані модулі, що робить його ефективним інструментом для задач моніторингу.

Ключову роль у реалізації системного моніторингу на Python відіграють бібліотеки, які надають засоби для отримання інформації про стан системи. Однією з найбільш важливих є бібліотека psutil, яка забезпечує кросплатформний доступ до даних про використання процесора, оперативної пам'яті, дискових накопичувачів, мережевих інтерфейсів та активних процесів. Вона дозволяє отримувати як миттєві значення показників, так і аналізувати їх у динаміці, що є необхідним для побудови систем моніторингу в реальному часі. Бібліотека psutil пропонує уніфікований інтерфейс, який дозволяє отримати дані про систему за допомогою простих викликів функцій, ховаючи складність взаємодії з ядром ОС під капотом.

Ключові можливості psutil [7; 14]:

CPU: завантаження у відсотках (загальне або по ядрах), час роботи в різних режимах, кількість фізичних та логічних ядер.

Memory: обсяг оперативної та віртуальної пам'яті (зайнята, вільна, доступна).

Disks: використання дискового простору, статистика операцій читання/запису (I/O).

Network: лічильники вхідного/вихідного трафіку, активні сокети, системні інтерфейси.

Процеси: повний контроль над запущеними програмами (PID, споживання пам'яті конкретним процесом, завершення процесу, дерево процесів).

У Листингу 1.1. показано простий приклад використання бібліотеки psutil для отримання ключових метрик.

Листинг 1.1

Код отримання ключових метрик моніторингу роботи комп'ютера
import psutil

1. Завантаження процесора за 1 секунду

```
print(f"Завантаження ЦП: {psutil.cpu_percent(interval=1)}%")
```

2. Стан оперативної пам'яті

```
mem = psutil.virtual_memory()
print(f"Використано RAM: {mem.percent}% (Всього: {mem.total / (1024**3):.2f} GB)")
```

3. Список всіх дисків та їх заповненість

```
for partition in psutil.disk_partitions():
    usage = psutil.disk_usage(partition.mountpoint)
    print(f"Диск {partition.device}: {usage.percent}% заповнено")
```

Для демонстрації порівняємо код з використанням psutils та код на C++.

Таблиця 1.1 - порівняння кодів Python та C++.

Python C++

```
import psutil
```

```
memory = psutil.virtual_memory()
print(f"RAM Usage: {memory.percent}%") #include <iostream> #include <windows.h> int main() {
MEMORYSTATUSEX memInfo; memInfo.dwLength = sizeof(MEMORYSTATUSEX); // Викликаємо системну функцію Windows API
if (GlobalMemoryStatusEx(&memInfo)) { std::cout << "RAM Usage: " << memInfo.dwMemoryLoad << " " << memInfo.dwTotal << " GB\n"; }
else { std::cerr << "Помилка отримання даних!\n"; return 0; }
```

Як бачимо з табл. 1.1 для C++ нам потрібно підключати специфічні системні заголовки, оголошувати структури та працювати з типами даних Windows. Цей код працюватиме тільки на Windows. Для Python все лаконічно, кросплатформно і зрозуміло з першого погляду.

Розглянемо бібліотеку platform, вона доповнює можливості psutils, надаючи інформацію про операційну систему, архітектуру процесора та інші характеристики середовища виконання, що може бути використано для адаптації програмного забезпечення до конкретної платформи.

[<https://realpython.com/ref/stdlib/platform/>] У табл. 1.2 описані основні функції модулю для доступу до системних даних.

Таблиця 1.2 - інтерфейс platform для доступу до системних даних

Функція Що повертає Приклад виводу

```
platform.system() Назва ОС 'Windows', 'Linux', 'Darwin' (macOS)
platform.release() Реліз системи '10', '5.15.0-101-generic'
platform.version() Детальна версія ОС '10.0.19045', '#104-Ubuntu SMP'
platform.machine() Тип архітектури 'x86_64', 'AMD64', 'arm64'
platform.processor() Назва процесора 'Intel64 Family 6 Model...'
platform.python_version() Версія Python '3.12.2'
```

Також доцільно використовувати стандартні модулі Python, такі як `os` та `subprocess`, які забезпечують взаємодію з операційною системою та виконання системних команд.

Окрім отримання даних, важливим аспектом є їх обробка та візуалізація. Для цього можуть застосовуватися бібліотеки, призначені для побудови графіків і діаграм, зокрема `matplotlib` або більш сучасні інструменти, такі як `plotly`. Вони дозволяють відображати зміну показників у часі, що є важливим для аналізу навантаження та виявлення тенденцій у роботі системи. Для створення графічного інтерфейсу користувача можуть використовуватися бібліотеки `tkinter` або `PyQt`, які забезпечують побудову інтерактивних віконних додатків. Використання таких засобів дозволяє створити зручний інтерфейс для відображення результатів моніторингу та взаємодії користувача з програмою.

Аналіз доступних технологій показує, що існують також альтернативні підходи до реалізації системного моніторингу, зокрема використання мов програмування нижчого рівня, таких як `C` або `C++`, які забезпечують більш прямий доступ до апаратних ресурсів і можуть бути ефективнішими з точки зору продуктивності. Проте їх використання значно ускладнює процес розробки та потребує глибших знань архітектури операційних систем. Іншим напрямом є застосування готових систем моніторингу з відкритим кодом, які можуть бути адаптовані під конкретні потреби, однак це не завжди дозволяє досягти необхідної гнучкості або зрозуміти внутрішні механізми роботи таких систем. У зв'язку з цим використання Python та відповідних бібліотек є компромісом між простотою реалізації, гнучкістю та функціональністю.

Таким чином, аналіз технологій і бібліотек для реалізації системного моніторингу показує, що мова програмування Python у поєднанні з бібліотеками `psutil`, `platform`, `matplotlib` та засобами створення графічного інтерфейсу є ефективним інструментарієм для розробки програмних засобів моніторингу ресурсів комп'ютера. Такий підхід дозволяє забезпечити кросплатформеність, достатню продуктивність, зручність розробки та можливість подальшого розширення функціоналу системи.

1.4. Постановка задачі розробки програмного засобу

Постановка задачі розробки програмного засобу для моніторингу використання ресурсів комп'ютера ґрунтується на результатах проведеного аналізу предметної області та існуючих рішень. Основною метою створення такого програмного засобу є забезпечення можливості отримання, обробки та відображення актуальної інформації про стан системних ресурсів у зручній та наочній формі. Розроблюваний програмний продукт повинен поєднувати достатню функціональність, простоту використання та можливість розширення, що дозволить застосовувати його як у навчальних цілях, так і для практичного аналізу роботи комп'ютерних систем. У межах постановки задачі необхідно визначити основні вимоги до системи, окреслити сценарії її використання та сформулювати перелік ключових функцій, які повинні бути реалізовані.

Функціональні вимоги до програмного засобу визначають перелік можливостей, які він повинен забезпечувати. Зокрема, система має здійснювати збір інформації про використання центрального процесора, оперативної пам'яті, дискових ресурсів та мережевої активності з використанням відповідних програмних інтерфейсів. Отримані дані повинні оброблятися та відображатися у режимі реального часу або з визначеною періодичністю оновлення. Програмний засіб має забезпечувати візуалізацію інформації у вигляді числових показників, таблиць та графіків, що дозволяє користувачу швидко оцінити стан системи. Крім того, передбачається можливість відображення списку активних процесів із зазначенням їхнього впливу на використання ресурсів, а також збереження статистичних даних для подальшого аналізу. Важливою функцією є забезпечення інтерактивної взаємодії користувача з програмою, зокрема можливість вибору параметрів моніторингу та налаштування інтервалів оновлення інформації.

Нефункціональні вимоги визначають якісні характеристики програмного засобу. До них належать вимоги щодо продуктивності, які передбачають мінімальне навантаження самої програми на систему під час збору даних; вимоги до зручності використання, що включають інтуїтивно зрозумілий інтерфейс користувача та наочність відображення інформації; вимоги до надійності, які передбачають стабільну роботу програми без збоїв упродовж тривалого часу; а також вимоги до переносимості, що забезпечують можливість використання програмного засобу на різних операційних системах. Крім того, важливим є забезпечення модульності структури програмного забезпечення, що дозволяє спростити його супровід та подальше розширення функціоналу.

Сценарії використання програмного засобу охоплюють різні варіанти взаємодії користувача із системою. Основним сценарієм є запуск програми та отримання інформації про поточний стан системних ресурсів у режимі реального часу. Інший сценарій передбачає аналіз роботи окремих процесів з метою виявлення тих, що споживають найбільшу кількість ресурсів. Також можливим є використання програми для спостереження за змінами навантаження системи у часі, що дозволяє оцінити ефективність роботи програм або вплив певних дій користувача на стан системи. У навчальному процесі програмний засіб може використовуватися для демонстрації принципів роботи операційної системи та розподілу ресурсів між процесами.

У результаті постановки задачі визначено основні функції програмного засобу, до яких належать: збір та обробка інформації про використання ресурсів комп'ютера; відображення даних у зручній та наочній формі; моніторинг процесів і їх впливу на систему; візуалізація динаміки зміни показників у часі; збереження результатів спостережень; забезпечення взаємодії користувача з програмою через графічний інтерфейс. Таким чином, сформульована постановка задачі визначає вимоги до структури та функціональності програмного засобу і створює основу для подальшого проєктування та реалізації системи моніторингу ресурсів комп'ютера.

РОЗДІЛ 2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАСОБУ МОНІТОРИНГУ РЕСУРСІВ

2.1. Загальна архітектура програмної системи

Загальна архітектура програмної системи моніторингу використання ресурсів комп'ютера визначається вимогами до функціональності, продуктивності та розширюваності програмного засобу. З урахуванням поставлених задач доцільним є використання модульного підходу до побудови системи, який передбачає поділ програми на окремі логічно завершені компоненти, кожен з яких відповідає за виконання визначеного набору функцій. Такий підхід забезпечує гнучкість розробки, спрощує супровід програмного забезпечення та дозволяє в подальшому

розширювати функціонал без суттєвих змін у вже реалізованих частинах системи. Архітектура програмного засобу може бути охарактеризована як багатокомпонентна, із чітким розподілом відповідальності між модулями, що взаємодіють між собою через визначені інтерфейси. У межах розроблюваної системи доцільно виділити кілька основних модулів, які формують її структуру (див. рис. 2.1). Центральним елементом є модуль збору даних, який відповідає за отримання інформації про стан системних ресурсів із використанням відповідних програмних бібліотек. Саме цей модуль здійснює ⁴взаємодію з операційною системою та ⁴апаратними компонентами, отримуючи дані про завантаження процесора, використання оперативної пам'яті, стан дискових накопичувачів та мережеву активність. Модуль збору даних повинен забезпечувати регулярне оновлення інформації з визначеним інтервалом часу, а також передавати отримані дані для подальшої обробки. Отримана інформація надходить до модуля обробки, який виконує її аналіз, структурування та підготовку до відображення або збереження. У цьому модулі здійснюється обчислення похідних показників, таких як середні значення, максимальні та мінімальні навантаження, а також формування агрегованих даних. Крім того, модуль обробки може реалізовувати механізми виявлення аномалій або відхилень у роботі системи, що підвищує інформативність програмного засобу. Таким чином, цей компонент виступає проміжною ланкою між збором сирих даних та їх подальшим використанням.

Рис. 2.1. Архітектура системи моніторингу

Модуль візуалізації забезпечує представлення отриманої інформації у зручній для користувача формі. Він реалізує графічний інтерфейс користувача, у якому відображаються поточні значення показників, графіки зміни навантаження у часі, а також додаткова інформація про процеси та стан системи. Основним завданням цього модуля є забезпечення наочності та зрозумілості представлення даних, що дозволяє користувачу швидко оцінити стан системи та виявити можливі проблеми. Важливим аспектом є забезпечення динамічного оновлення інформації, що створює ефект роботи в режимі реального часу.

Окремим компонентом системи є модуль збереження статистики, який відповідає за накопичення та зберігання даних про використання ресурсів. Збережена інформація може використовуватися для подальшого аналізу, побудови звітів або дослідження динаміки роботи системи протягом тривалого часу. Реалізація цього модуля може передбачати запис даних у файли або використання простих баз даних. Наявність такого компонента розширює можливості програмного засобу, дозволяючи переходити від миттєвого моніторингу до аналізу історичних даних. Таку архітектуру можна віднести насамперед до модульної багатоваріантної архітектури. У ній чітко виділяються окремі функціональні рівні: отримання даних, їх обробка, подання користувачу та збереження результатів. Кожен компонент має власну відповідальність, а взаємодія між ними є відносно простою та логічною. Саме тому така побудова добре відповідає принципам Layered Architecture, де система поділяється на шари з контрольованими зв'язками між ними.

Також цю побудову частково можна розглядати як спрощений варіант pipeline-архітектури або архітектури типу data flow, оскільки дані проходять послідовний шлях: спочатку збираються, потім обробляються, після чого або візуалізуються, або зберігаються. Водночас вона не є класичним pipeline-рішенням у чистому вигляді, тому що модуль візуалізації та модуль збереження статистики можуть отримувати результати паралельно з одного джерела - модуля обробки.

Крім того, у цій структурі простежуються окремі риси шаблону Model-View-Controller. Модуль візуалізації можна співвіднести з рівнем подання, модуль збору та обробки даних - з логікою застосунку, а модуль збереження статистики - з роботою з даними. Проте повноцінною MVC-архітектурою таку систему назвати не зовсім коректно, якщо в ній явно не виділено контролер керування подіями інтерфейсу. Тому найбільш точним формулюванням буде: модульна багатоваріантна архітектура з елементами потокової обробки даних.

Таким чином, запропонована архітектура програмної системи базується на модульному принципі та включає основні компоненти, що забезпечують повний цикл обробки інформації - від її збору до відображення та збереження. Такий підхід дозволяє створити гнучкий, розширюваний та ефективний програмний засіб для моніторингу використання ресурсів комп'ютера.

2.2. Моделювання структури та компонентів системи

Моделювання структури та компонентів системи є важливим етапом проектування програмного засобу, оскільки воно дозволяє формалізувати внутрішню будову майбутнього застосунку, визначити основні складові, їх призначення та характер взаємодії між ними. Для програмного засобу моніторингу використання ресурсів комп'ютера така формалізація має особливе значення, оскільки система повинна одночасно забезпечувати отримання даних із середовища операційної системи, їх обробку, відображення та, за необхідності, збереження для подальшого аналізу. Саме тому моделювання дає змогу ще до етапу програмної реалізації визначити логічну організацію системи, уникнути дублювання функцій окремих модулів та забезпечити узгодженість між усіма складовими програмного засобу.

Моделювання складної системи почнемо з розробки діаграми UML варіантів використання на основі результатів п.1.4 з першого розділу цієї кваліфікаційної роботи. На рис. 2.2 наведено діаграму яка відображає функціональне призначення системи моніторингу ресурсів комп'ютера та основні сценарії взаємодії користувача із застосунком. Основним актором виступає користувач, який через графічний інтерфейс ініціює запуск програми, переглядає поточний стан системних ресурсів, аналізує процеси, змінює параметри оновлення, зберігає статистику та переглядає історію спостережень. Центральний варіант використання пов'язаний з отриманням інформації про CPU, RAM, диск і мережу, а додаткові функції візуалізації та збереження подано як розширення базового сценарію моніторингу..

Рис. 2.2. Діаграма варіантів використання для системи моніторингу

Як бачимо, рис. 2.2 відображає взаємодію зовнішнього актора, яким є користувач, із програмним засобом моніторингу ресурсів комп'ютера. У межах системи показано основні варіанти використання, що характеризують функціональні можливості застосунку. Користувач може запустити програму, переглянути поточний стан системних ресурсів, отримати доступ до списку активних процесів, побудувати графіки навантаження, змінити інтервал оновлення даних, зберегти статистику у файл та переглянути історію спостережень.

Центральним варіантом використання є перегляд поточного стану ресурсів, оскільки саме він відображає головне призначення розроблюваного програмного засобу. Для реалізації цього варіанта система обов'язково включає підзадачі отримання даних про завантаження центрального процесора, використання оперативної пам'яті, стан дискової підсистеми та мережеву активність. Саме тому між цим варіантом використання та відповідними підфункціями встановлено зв'язки типу `<include>`, які означають, що вони є обов'язковими складовими основного сценарію моніторингу.

Окремі функції, такі як побудова графіків навантаження та збереження статистики у файл, подані як розширення основного варіанта

Діаграма класів (рис. 2.3) відображає внутрішню структуру програмного засобу моніторингу ресурсів комп'ютера та взаємозв'язки між його основними компонентами. Центральним елементом є клас головного вікна, який координує роботу модулів збору, обробки, візуалізації та збереження даних. Клас збору формує знімок поточного стану системи, клас обробки аналізує ці дані, модуль візуалізації відображає результати користувачу, а модуль збереження накопичує статистику для подальшого аналізу.

Наступним кроком була розробка діаграми компонентів яка відображає загальну структурну організацію програмного засобу моніторингу ресурсів комп'ютера. Вона демонструє взаємодію між користувачем, графічним інтерфейсом, модулями збору, обробки, візуалізації та збереження статистики, а також зовнішніми залежностями у вигляді системних бібліотек, операційної системи та локального сховища даних.

Діаграма демонструє, як після запуску моніторингу система отримує показники з операційної системи, обробляє їх, відображає користувачу та, за потреби, зберігає у зовнішнє сховище.

Після завершення збору інформації модуль DataCollector передає сформований знімок стану системи до модуля DataProcessor. На цьому етапі відбувається аналіз, агрегування та підготовка даних до подальшого використання. Оброблені результати передаються як до графічного інтерфейсу, так і до модуля візуалізації, який формує графіки, таблиці та інші засоби подання інформації. Після цього користувач бачить актуальний стан системних ресурсів у наочній формі.

Окремо на діаграмі виділено додатковий сценарій збереження статистики. Якщо користувач ініціює цю дію, інтерфейс звертається до модуля `StatisticsStorage`, який виконує запис оброблених даних у зовнішнє сховище, наприклад у CSV-файл, JSON-файл або базу даних SQLite. Після успішного завершення запису система повертає користувачеві відповідне повідомлення. Такий фрагмент оформлений через конструкцію `opt`, що підкреслює необов'язковий характер цього сценарію.

Крім того, діаграма містить цикл `loop`, який демонструє періодичне оновлення інформації. Це відповідає реальному режиму роботи системи моніторингу, коли через певні інтервали часу знову виконується збір показників, їх обробка та оновлення подання на екрані. Таким чином, діаграма послідовності дозволяє наочно показати часову логіку функціонування системи та взаємодію її компонентів у процесі моніторингу. Діаграма діяльності (рис. 2.6) відображає алгоритм роботи програмного засобу моніторингу ресурсів комп'ютера від моменту запуску програми до завершення її роботи. На діаграмі показано процес ініціалізації компонентів, запуску циклу моніторингу, збору та обробки системних показників, відображення результатів, збереження статистики та перегляду історії спостережень.

Рис. 2.6. Діаграма діяльності для системи моніторингу

Діаграма діяльності на рис. 2.6. відображає загальний алгоритм функціонування системи моніторингу ресурсів комп'ютера як послідовність дій і логічних переходів між ними. Початковими етапами є запуск програми, ініціалізація її основних компонентів та відображення головного вікна. Після цього система переходить у стан очікування дії користувача, який може запустити моніторинг або виконати інші дії, передбачені інтерфейсом.

Основною частиною діаграми є цикл моніторингу, який активується після натискання користувачем команди початку спостереження. У межах цього циклу система послідовно отримує дані про завантаження центрального процесора, використання оперативної пам'яті, дискової підсистеми, мережних ресурсів та активних процесів. Після збору даних формується знімок поточного стану системи, виконується обробка та агрегування показників, а потім результати відображаються користувачу у вигляді таблиць, числових показників і графіків.

Окремим логічним фрагментом на діаграмі представлено перевірку необхідності збереження статистики. Якщо користувач або система ініціює таку дію, результати моніторингу записуються у файл або базу даних. Після цього система переходить до очікування наступного інтервалу оновлення та повторює цикл доти, доки моніторинг залишається активним. Така побудова добре відображає циклічний характер роботи програмного засобу в режимі реального часу.

Після завершення моніторингу користувач може звернутися до функції перегляду історії спостережень. У цьому випадку система зчитує раніше збережені дані, формує на їх основі історичні звіти або графіки та відображає результати на екрані. Завершальним етапом є закриття програми.

Отже, діаграма діяльності дає змогу наочно подати загальну логіку функціонування системи моніторингу, підкреслюючи послідовність дій, наявність циклів і умовних переходів.

2.3. Проектування алгоритмів збору та обробки системної інформації

У цьому параграфі розглядаються алгоритми збору та обробки основних типів системної інформації, що характеризують стан комп'ютерної системи. Зокрема, буде детально описано алгоритми отримання завантаження центрального процесора, моніторингу використання оперативної пам'яті, аналізу дискової активності та збору мережної статистики. Для кожного з цих напрямів буде визначено особливості отримання даних, принципи їх обробки та можливості використання у межах програмного засобу моніторингу. Такий підхід дозволяє сформувати цілісну систему алгоритмів, яка забезпечує повний контроль за використанням основних ресурсів комп'ютера.

Проектування алгоритмів збору та обробки системної інформації є одним із ключових етапів розробки програмного засобу моніторингу ресурсів комп'ютера, оскільки саме від якості реалізації цих алгоритмів залежить точність отриманих даних, ефективність роботи системи та зручність їх подальшого використання. У контексті розроблюваного застосунку алгоритми повинні забезпечувати регулярне отримання актуальних показників стану комп'ютерної системи, їх коректну інтерпретацію, а також підготовку до відображення або збереження. Враховуючи, що система працює у режимі, близькому до реального часу, особливого значення набувають питання продуктивності, мінімізації затримок та оптимального використання обчислювальних ресурсів під час виконання процедур збору та обробки даних.

Алгоритми збору інформації повинні враховувати специфіку функціонування операційної системи та особливості доступу до апаратних ресурсів. Зокрема, отримання показників завантаження центрального процесора, використання оперативної пам'яті, дискової активності та мережної статистики базується на використанні системних викликів або спеціалізованих бібліотек, таких як `psutil`. При цьому важливо забезпечити коректність вимірювань, оскільки деякі показники мають динамічний характер і потребують обчислення на основі різниці значень за певний проміжок часу. Таким чином, алгоритми повинні передбачати не лише отримання миттєвих значень, але й можливість аналізу їх змін у часовій динаміці.

Не менш важливим є проектування алгоритмів обробки отриманих даних. Сирі показники, отримані з операційної системи, часто мають різний формат представлення та потребують нормалізації, агрегування та структуризації. У межах програмного засобу ці алгоритми повинні забезпечувати приведення даних до єдиного вигляду, обчислення додаткових характеристик (наприклад, середніх значень, відсоткового використання ресурсів, пікових навантажень), а також підготовку структур даних, зручних для подальшої візуалізації. Крім того, обробка повинна бути організована таким чином, щоб мінімізувати вплив на загальну продуктивність системи та не створювати додаткового навантаження на ресурси, які вже є об'єктом моніторингу.

Алгоритм отримання завантаження центрального процесора (рис. 2.7) в системі моніторингу ресурсів комп'ютера призначений для визначення поточного рівня використання обчислювальних можливостей системи у відсотковому вираженні. Його основне призначення полягає в тому, щоб із заданою періодичністю отримувати актуальні показники навантаження на процесор, перетворювати їх у придатну для подальшого використання форму та передавати до модулів обробки, візуалізації або збереження статистики. У загальному випадку такий алгоритм повинен працювати циклічно, оскільки завантаження процесора є динамічним параметром, що постійно змінюється залежно від активності операційної системи, запущених процесів і фонових служб.

Рис. 2.7. Узагальнена схема алгоритму отримання завантаження центрального процесора

На початковому етапі алгоритм ініціалізує процедуру зчитування даних про центральний процесор. Для цього програмний засіб звертається до системної бібліотеки, наприклад `psutil`, яка надає доступ до відповідних показників операційної системи. Далі виконується запит на отримання значення поточного завантаження CPU. Залежно від обраного способу реалізації може використовуватися або миттєве значення, або

вимірювання за певний короткий інтервал часу, що дозволяє отримати більш коректну оцінку фактичного навантаження. На цьому етапі також може бути передбачено отримання як загального показника завантаження процесора, так і окремих значень для кожного ядра, якщо це потрібно для розширеного аналізу.

Після отримання первинного значення алгоритм переходить до його перевірки та первинної обробки. Зокрема, виконується контроль коректності отриманих даних, приведення їх до потрібного числового формату та, за необхідності, округлення до визначеної кількості знаків після коми. Якщо система підтримує накопичення історії спостережень, отриманий показник також може бути збережений у внутрішній структурі даних разом із часовою позначкою. Це дозволяє в подальшому будувати графіки зміни навантаження процесора у часі, визначати середні значення або виявляти пікові навантаження.

На завершальному етапі алгоритм передає підготовлене значення до інших компонентів системи. Якщо моніторинг здійснюється у режимі реального часу, показник одразу надходить до модуля візуалізації, де відображається у вигляді числового значення, індикатора або графіка. Якщо увімкнено режим збереження статистики, це саме значення також передається до модуля запису у файл або базу даних. Після цього алгоритм переходить у стан очікування до настання наступного інтервалу оновлення, після чого цикл повторюється. Таким чином, загальна логіка алгоритму отримання завантаження CPU полягає у послідовному виконанні дій зі зчитування, перевірки, обробки, передачі та, за потреби, збереження показників завантаження центрального процесора.

Рис. 2.8. Узагальнена схема алгоритму моніторингу використання оперативної пам'яті

Алгоритм моніторингу використання оперативної пам'яті (рис. 2.8) призначений для отримання та аналізу показників, що характеризують поточний стан пам'яті комп'ютерної системи. Його робота починається з ініціалізації процедури звернення до системних засобів отримання інформації, після чого виконується зчитування даних про загальний обсяг оперативної пам'яті, обсяг зайнятої та вільної пам'яті, а також, за потреби, доступний обсяг пам'яті для нових процесів. Далі отримані дані перевіряються на коректність, приводяться до єдиного формату та використовуються для обчислення відсоткового співвідношення використаних і вільних ресурсів. Після обробки результати передаються до модуля візуалізації для відображення у вигляді числових значень або графічних елементів, а за необхідності - до модуля збереження статистики. Такий алгоритм дозволяє забезпечити безперервний контроль за станом оперативної пам'яті та своєчасно виявляти ситуації, пов'язані з її надмірним використанням.

Рис. 2.9. Узагальнена схема алгоритму аналізу дискової активності

Алгоритм аналізу дискової активності (рис. 2.9) орієнтований на отримання інформації про стан дискової підсистеми та інтенсивність операцій читання й запису. На початковому етапі виконується ініціалізація процедури збирання дискових показників, після чого система звертається до відповідних бібліотек або системних інтерфейсів для отримання даних про загальний обсяг накопичувача, зайнятий і вільний простір, а також параметри вводу-виводу. Після перевірки коректності зчитаних даних алгоритм здійснює їх узагальнення та формує показники, що характеризують як використання дискового простору, так і активність роботи з даними. Отримані результати можуть бути використані для поточного відображення стану дискової системи, а також для накопичення статистики з метою подальшого аналізу. Реалізація такого алгоритму дозволяє не лише оцінювати заповненість накопичувача, але й виявляти ознаки підвищеного навантаження на дискову підсистему.

Алгоритм збору мережевої статистики (рис. 2.10) забезпечує отримання показників, що характеризують мережеву активність комп'ютера у процесі його роботи. Його функціонування починається зі зчитування системних лічильників, які містять інформацію про кількість переданих і прийнятих байтів, пакетів, а також інші параметри мережевих інтерфейсів. Оскільки мережеві показники мають динамічний характер, алгоритм, як правило, передбачає порівняння значень, отриманих у різні моменти часу, що дає змогу визначити інтенсивність мережевого обміну за заданий інтервал.

Після цього виконується обробка та форматування результатів, які передаються до модулів візуалізації та збереження. Використання такого алгоритму дає можливість оцінювати мережеве навантаження системи, виявляти періоди підвищеної активності та аналізувати загальну динаміку обміну даними в мережі.

Рис. 2.10. Узагальнена схема алгоритму збору мережевої статистики

У межах підрозділу 2.3 було розроблено сукупність алгоритмів збору та обробки системної інформації, що охоплюють основні аспекти функціонування комп'ютерної системи, а саме: завантаження центрального процесора, використання оперативної пам'яті, дискову активність та мережеву взаємодію. Усі розглянуті алгоритми орієнтовані на роботу в режимі реального часу та забезпечують періодичне отримання актуальних показників із використанням системних засобів операційної системи або спеціалізованих бібліотек. Отримані дані проходять етапи перевірки, нормалізації та форматування, що дозволяє уніфікувати їх подальше використання незалежно від джерела походження. Таким чином, сформовано цілісну алгоритмічну основу для функціонування системи моніторингу.

Спільною рисою всіх алгоритмів є їх циклічна структура, яка передбачає послідовне виконання етапів ініціалізації, збору даних, перевірки коректності, обробки, передачі результатів до модулів візуалізації та, за потреби, збереження статистики. Така уніфікована логіка забезпечує модульність і масштабованість програмної системи, спрощує її розширення новими компонентами моніторингу та підвищує надійність обробки інформації. Водночас специфіка кожного алгоритму визначається природою відповідного ресурсу: для процесора важливою є миттєва оцінка навантаження, для оперативної пам'яті - співвідношення використаних і доступних ресурсів, для дискової підсистеми - як обсяг зайнятого простору, так і інтенсивність операцій вводу-виводу, а для мережі - динаміка передавання даних у часі.

Розроблені алгоритми становлять основу практичної реалізації програмного застосунку моніторингу та безпосередньо інтегруються у відповідні модулі системи. Зокрема, результати їх виконання використовуються модулем обробки для узагальнення показників, модулем візуалізації - для відображення поточного стану системи у зручній для користувача формі, а модулем збереження статистики - для накопичення історичних даних та подальшого аналізу. Завдяки цьому забезпечується комплексний підхід до моніторингу, який поєднує оперативний контроль стану ресурсів із можливістю довгострокового аналізу та виявлення тенденцій функціонування комп'ютерної системи.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ ТА АПРОБАЦІЯ ПРОГРАМНОГО ЗАСОБУ

3.1. Реалізація програмного засобу для моніторингу системних ресурсів

Програмна реалізація застосунку для моніторингу системних ресурсів базується на використанні мови програмування Python, що зумовлено її універсальністю, простотою синтаксису та наявністю широкого спектра бібліотек для роботи з операційною системою. Вибір саме цієї мови обґрунтовується тим, що Python забезпечує ефективний баланс між швидкістю розробки та функціональними можливостями, що є особливо важливим при створенні прикладних програм системного моніторингу. Крім того, Python є кросплатформним інструментом, що дозволяє створювати програмні засоби, здатні працювати на різних операційних системах без суттєвих змін у коді. Значну роль у виборі відіграє також активна спільнота розробників та велика кількість відкритих бібліотек, які значно спрощують реалізацію складних функціональних можливостей. Для реалізації функцій моніторингу використано бібліотеку `psutil`, яка надає зручний інтерфейс доступу до інформації про використання центрального процесора, оперативної пам'яті, дискових ресурсів, мережевої активності та списку активних процесів. Додатково застосовано модуль `platform`, який дозволяє отримувати інформацію про операційну систему та апаратне середовище виконання програми. Для створення графічного інтерфейсу користувача можуть використовуватися стандартні засоби Python, зокрема бібліотека `tkinter`, або більш розширені фреймворки, такі як `PyQt`, що забезпечують створення сучасного інтерфейсу з підтримкою графіків та інтерактивних елементів. Для візуалізації даних доцільно використовувати бібліотеку `matplotlib`, яка дозволяє будувати графіки зміни показників у часі. У якості середовища розробки обрано інтегроване середовище PyCharm, що забезпечує зручність написання коду, налагодження та тестування програмного застосунку. Реалізація програмного засобу здійснюється відповідно до розробленої архітектури, яка передбачає поділ системи на окремі функціональні модулі. На початковому етапі створюється базова структура застосунку, яка включає головний модуль керування, що відповідає за запуск програми, ініціалізацію компонентів та організацію циклу моніторингу. Далі реалізується модуль збору даних, який забезпечує взаємодію з операційною системою та отримання показників використання ресурсів. Наступним кроком є реалізація модуля обробки даних, у якому здійснюється аналіз отриманої інформації, її нормалізація та підготовка до подальшого використання.

Після цього реалізується модуль візуалізації, який відповідає за відображення результатів моніторингу у вигляді числових значень, таблиць та графіків. Особлива увага приділяється забезпеченню динамічного оновлення інформації, що дозволяє відображати стан системи у режимі реального часу. Паралельно розробляється модуль збереження статистики, який забезпечує накопичення даних про використання ресурсів у зовнішньому сховищі, що дає змогу виконувати подальший аналіз і формувати звіти.

Загалом процес програмної реалізації передбачає послідовне впровадження кожного з модулів із подальшою їх інтеграцією в єдину систему. Такий підхід дозволяє забезпечити узгодженість роботи всіх компонентів, а також спрощує тестування та вдосконалення програмного засобу. У результаті створюється цілісний застосунок, який реалізує функції збору, обробки, відображення та збереження інформації про стан системних ресурсів комп'ютера.

Почнемо з модуля збору даних так, щоб його надалі було зручно підключити до графічного інтерфейсу на tkinter. Для цього доцільно одразу реалізувати окремий клас, який не залежить від GUI, а лише повертає підготовлені дані. Такий підхід спрощує тестування, повторне використання коду та інтеграцію в головний застосунок. Повний текст коду розробленого класу DataCollector наведено в додатку А. Клас DataCollector, який відповідає за збір системної інформації. На даному етапі він містить функції для роботи з процесором, але його структура одразу підготовлена до подальшого розширення. Це означає, що пізніше до цього самого класу буде додано методи для отримання інформації про оперативну пам'ять, дискову активність і мережеву статистику. Така організація є зручною для інтеграції в tkinter, оскільки графічний інтерфейс зможе просто викликати потрібні методи класу й отримувати готові словники з даними.

Основною є функція `get_cpu_info()` (дивись листинг 3.1). Вона виконує повніше опитування стану процесора. Спочатку формується мітка часу `timestamp`, щоб зафіксувати момент отримання даних. Потім за допомогою `psutil.cpu_percent(interval=interval)` визначається загальне завантаження процесора у відсотках. Параметр `interval` задає тривалість вимірювання: якщо він дорівнює 1.0, функція фактично аналізує навантаження протягом однієї секунди. Далі за допомогою `psutil.cpu_percent(interval=None, percpu=True)` отримується список значень завантаження для кожного ядра окремо. Після цього збирається додаткова інформація: кількість фізичних ядер, кількість логічних потоків, поточна, мінімальна та максимальна частота процесора, а також статистика перемикачів контексту, переривань і системних викликів. Усі ці дані формуються у словник `cpu_data`, який і повертається з функції.

Листинг 3.1 - Скорочений код `get_cpu_info` (без додаткових параметрів)

```
def get_cpu_info(self, interval: float = 1.0) -> Dict[str, Any]:
    try:
        timestamp: str = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        total_percent: float = psutil.cpu_percent(interval=interval)
        per_core_percent: List[float] = psutil.cpu_percent(interval=None, percpu=True)
        physical_cores: int | None = psutil.cpu_count(logical=False)
        logical_cores: int | None = psutil.cpu_count(logical=True)
        freq = psutil.cpu_freq()
        if freq is not None:
            current_frequency: float = round(freq.current, 2)
            min_frequency: float = round(freq.min, 2)
            max_frequency: float = round(freq.max, 2)
        else:
            current_frequency = 0.0
            min_frequency = 0.0
            max_frequency = 0.0
        return cpu_data
    except Exception as error:
        return {
            "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
            "error": True,
            "message": f"Помилка отримання параметрів CPU: {error}"
        }
```

Допоміжною є функція `get_cpu_load_fast()`. Її призначення полягає у швидкому отриманні поточного завантаження CPU без затримки на інтервал вимірювання. Вона буде особливо корисною при роботі з `tkinter`, коли потрібно часто оновлювати показники у вікні без блокування інтерфейсу. Ця функція повертає спрощений набір даних: мітку часу, загальне завантаження CPU та список завантаження по ядрах. Вона працює швидше, але точність оцінки може бути дещо нижчою, ніж у `get_cpu_info()`.

Функція `get_cpu_info(interval: float = 1.0)` (Листинг 3.1) є головною для повного збору параметрів процесора. Її аргумент `interval` задає часовий інтервал вимірювання. Якщо встановити значення 1.0, то функція чекатиме одну секунду та на основі цього проміжку обчислить завантаження. У середині функції використовується змінна `timestamp`, у якій зберігається дата і час вимірювання. Змінна `total_percent` містить загальне завантаження CPU у відсотках. Змінна `per_core_percent` є списком, де кожен елемент відповідає завантаженню окремого ядра. `physical_cores` і `logical_cores` відображають апаратну структуру процесора. Змінна `freq` містить об'єкт з частотними характеристиками процесора, з якого виділяються `current_frequency`, `min_frequency` і `max_frequency`. Далі змінна `cpu_stats` містить службову статистику роботи процесора: кількість перемикачів контексту, переривань, м'яких переривань і системних викликів. Після цього всі значення збираються в один словник. Якщо під час виконання виникає помилка, вона перехоплюється блоком `except`, і функція повертає словник з ознакою помилки та пояснювальним повідомленням.

Функція `get_cpu_load_fast()` має спрощений алгоритм. Спочатку також фіксується поточний час у змінній `timestamp`. Потім викликається `psutil.cpu_percent(interval=None)` для швидкого отримання загального завантаження та `psutil.cpu_percent(interval=None, percpu=True)` для окремих ядер. Отримані значення округлюються до двох знаків після коми та повертаються у словнику. Перевага цього методу полягає у швидкості виконання, що робить його зручним для постійного оновлення віджетів інтерфейсу. Недолік полягає в тому, що він не виконує повноцінне вимірювання за часовий інтервал, а тому є більше придатним для оперативного відображення, ніж для точного аналізу.

У межах модуля збору даних було реалізовано функції отримання параметрів центрального процесора, які забезпечують як повне, так і швидке опитування стану CPU. Основна функція виконує збір комплексної інформації про завантаження процесора, кількість ядер, частотні характеристики та службову статистику, тоді як допоміжна функція орієнтована на оперативне оновлення показників у графічному інтерфейсі. Такий підхід забезпечує гнучкість використання модуля та створює основу для його подальшої інтеграції у застосунок моніторингу, побудований із використанням `tkinter`.

Продовжимо реалізацію модуля збору даних, доповнивши його функціями для моніторингу оперативної пам'яті. Як і у випадку з CPU, функції проєктуються таким чином, щоб їх було легко інтегрувати у графічний інтерфейс на `tkinter`, а також використовувати як для швидкого оновлення показників, так і для більш детального аналізу.

У додатку Б наведено повний код для роботи з оперативною пам'яттю (як частина вже існуючого класу `DataCollector`).

Цей код розширює модуль збору даних функціями для отримання інформації про оперативну пам'ять. Як і у випадку з процесором, реалізація базується на використанні бібліотеки `psutil`, яка надає зручний доступ до системних параметрів. Основною перевагою такого підходу є уніфікованість: всі функції модуля повертають дані у вигляді словників, що значно спрощує їх подальшу передачу до модулів обробки, візуалізації або збереження.

Основною є функція `get_memory_info()`. Вона виконує повне зчитування параметрів оперативної пам'яті. На початку формується мітка часу `timestamp`, яка дозволяє прив'язати дані до конкретного моменту. Далі викликається `psutil.virtual_memory()`, що повертає об'єкт із повною інформацією про пам'ять. З цього об'єкта витягуються ключові параметри: загальний обсяг пам'яті (`total`), доступний обсяг (`available`), використаний обсяг (`used`) та відсоток використання (`percent`). Значення пам'яті переводяться з байтів у гігабайти шляхом ділення на 1024×1024 та округлюються до двох знаків після коми. Усі ці значення зберігаються у словнику `memory_data`, який і повертається з функції.

Допоміжною є функція `get_memory_fast()`, яка призначена для швидкого отримання основних показників. Вона використовує той самий виклик `psutil.virtual_memory()`, але повертає лише ключові параметри - відсоток використання пам'яті та обсяг зайнятої пам'яті. Такий підхід дозволяє мінімізувати обсяг обчислень і використовувати функцію для частого оновлення даних у графічному інтерфейсі без помітного навантаження на систему.

Розглянемо детальніше функцію `get_memory_info()`. Основною змінною є `mem`, яка містить результат виклику `psutil.virtual_memory()` і включає всі необхідні параметри пам'яті. Змінні `total_memory`, `available_memory` та `used_memory` відповідають за обсяги пам'яті у гігабайтах. Змінна `percent_used` відображає частку використаної пам'яті у відсотках. Формування словника результатів дозволяє стандартизувати формат даних і використовувати його як універсальний інтерфейс між різними модулями системи. У разі виникнення помилки використовується блок `try-except`, що забезпечує стабільність роботи програми навіть у випадку збоїв при отриманні системної інформації.

Функція `get_memory_fast()` має спрощений алгоритм і використовує лише необхідний мінімум даних. Вона формує компактний словник, який містить лише найважливіші показники для відображення у GUI. Це робить її оптимальною для використання в циклі оновлення `tkinter`, наприклад через метод `after()`.

У контексті розроблюваного застосунку ці функції будуть використовуватися таким чином: функція `get_memory_fast()` застосовуватиметься для оперативного оновлення інтерфейсу користувача, наприклад відображення відсотка використання пам'яті у вигляді індикатора або графіка, тоді як функція `get_memory_info()` - для періодичного збереження повної інформації у модулі статистики або для більш детального аналізу. Такий підхід дозволяє поєднати швидкодію та інформативність, що є важливим для систем моніторингу, які працюють у режимі реального часу. Таким чином, у межах модуля збору даних реалізовано функції отримання параметрів оперативної пам'яті, які забезпечують як детальний аналіз її використання, так і швидке отримання основних показників для відображення у графічному інтерфейсі. Основна функція формує повний набір характеристик пам'яті, тоді як допоміжна орієнтована на оперативне оновлення даних.

Продовжимо реалізацію модуля збору даних, додавши функції для аналізу дискової активності. У додатку В наведено повний код функцій для роботи з дисковою підсистемою (як частина класу `DataCollector`). Цей код реалізує функціональність збору даних про дискову підсистему.

Основна функція `get_disk_info()` забезпечує отримання повної інформації про стан диска та інтенсивність операцій введення-виведення, тоді як допоміжна функція `get_disk_io_fast()` орієнтована на швидке оновлення показників у графічному інтерфейсі.

Функція `get_disk_info()` виконує комплексний збір даних. На першому етапі формується мітка часу `timestamp`. Далі викликається `psutil.disk_usage("/")`, що дозволяє отримати інформацію про використання дискового простору: загальний обсяг (`total`), зайнятий (`used`), вільний (`free`) та відсоток використання (`percent`). Паралельно викликається `psutil.disk_io_counters()`, який повертає статистику операцій введення-виведення. З цього об'єкта отримуються такі параметри, як кількість прочитаних і записаних байтів (`read_bytes`, `write_bytes`), а також кількість операцій читання та запису (`read_count`, `write_count`). Значення обсягів переводяться у мегабайти або гігабайти для зручності подальшого відображення. Усі дані формуються у словник `disk_data`, який повертається як результат роботи функції.

Функція `get_disk_io_fast()` має спрощений характер і використовується для швидкого отримання основних показників активності диска. Вона повертає лише обсяг прочитаних і записаних даних у мегабайтах, що дозволяє використовувати її для частого оновлення графічних елементів інтерфейсу без значного навантаження на систему. Такий підхід є важливим при реалізації динамічних графіків у `tkinter`.

Розглянемо детальніше основну функцію `get_disk_info()`. Ключовими змінними є `disk_usage`, що містить інформацію про використання дискового простору, та `disk_io`, яка містить статистику операцій введення-виведення. Змінні `total_disk`, `used_disk` та `free_disk` представляють обсяги пам'яті у гігабайтах, тоді як `percent_used` відображає ступінь заповненості диска. Змінні `read_bytes` і `write_bytes` характеризують обсяг даних, що були прочитані або записані, а `read_count` і `write_count` - кількість відповідних операцій. Така структура дозволяє отримати як статичну інформацію про стан диска, так і динамічні показники його використання.

У контексті розроблюваного застосунку ці функції будуть використовуватися таким чином: функція `get_disk_io_fast()` застосовуватиметься для відображення поточної активності диска у вигляді графіків або індикаторів у реальному часі, тоді як `get_disk_info()` - для періодичного збору повної інформації, яка може бути зведена тут рівняння.бережена та використана для подальшого аналізу. Такий підхід забезпечує баланс між продуктивністю та інформативністю системи моніторингу.

Завершальним етапом реалізації модуля збору даних є розробка функцій для отримання та аналізу мережевої статистики (код дивись в додатку Г). Особливістю моніторингу мережевої активності є те, що більшість показників, які надає операційна система, мають накопичувальний характер. Це означає, що значення кількості переданих і прийнятих байтів постійно зростають з моменту запуску системи, а тому для визначення реальної інтенсивності мережевого обміну необхідно використовувати не абсолютні значення, а їх зміну за певний проміжок часу, тобто так звану дельту.

Реалізація алгоритму збору мережевої статистики базується на використанні функції `psutil.net_io_counters()`, яка повертає накопичувальні значення переданих і прийнятих байтів. Однак ці значення самі по собі не дають уявлення про поточну швидкість мережі, тому в системі реалізовано підхід обчислення дельти між двома послідовними вимірюваннями. Для цього у класі використовується змінна `_prev_net`, яка зберігає попередній стан лічильників.

Алгоритм роботи функції `get_network_speed()` полягає у наступному: при кожному виклику зчитуються поточні значення байтів, після чого обчислюється різниця між поточним і попереднім значенням. Ця різниця відображає кількість даних, переданих або отриманих за інтервал часу між двома викликами функції. Далі ця величина нормується на інтервал часу, що дозволяє отримати швидкість передачі даних.

Математично обчислення швидкості можна представити у вигляді формули:

$$V = \frac{\Delta t}{\Delta t} \quad (3.1)$$

де - поточне значення кількості переданих або отриманих байтів,

- попереднє значення,

Δt - інтервал часу між вимірюваннями,

V - швидкість передачі даних.

Для зручності відображення результат переводиться у кілобайти за секунду:

$$= \frac{\Delta t}{\Delta t} \quad (3.2)$$

У функції `get_network_speed()` використовуються змінні `delta_sent` і `delta_recv`, які представляють зміну кількості переданих і отриманих байтів відповідно. Значення `upload_speed` і `download_speed` визначають швидкість передачі та прийому даних у КБ/с. Після кожного обчислення попередні значення оновлюються, що забезпечує коректність наступного вимірювання.

У контексті розроблюваного застосунку функція `get_network_speed()` є ключовою для відображення мережевої активності у режимі реального часу, наприклад у вигляді графіків або індикаторів швидкості. Функція `get_network_info()` може використовуватися для отримання загальної статистики або збереження історичних даних. Такий підхід дозволяє поєднати аналіз накопичувальних показників і динамічну оцінку мережевої активності, що є важливим для повноцінного моніторингу системи.

Рис. 3.1. Встановлення пакета `psutil`

Для підготовки середовища `PyCharm` для коректного виконання коду виконаємо декілька дій. Перше, встановимо необхідні пакети `psutil` та `plt` (рис.3.1 та рис.3.2).

Рис. 3.2. Встановлення пакета `plt`

3.2. Реалізація графічного інтерфейсу користувача

Побудова графічного інтерфейсу користувача є важливим етапом реалізації програмного засобу моніторингу системних ресурсів, оскільки саме через інтерфейс забезпечується взаємодія користувача з функціональними можливостями застосунку. У межах даної роботи для створення інтерфейсу обрано бібліотеку `'tkinter'`, яка входить до стандартного складу Python та надає достатній набір засобів для розробки віконних прикладних програм. Вибір `'tkinter'` обґрунтовується її доступністю, простотою використання, відсутністю потреби у встановленні додаткових компонентів та можливістю створення функціонального графічного середовища, достатнього для реалізації поставлених задач. Крім того, ця бібліотека добре інтегрується з раніше розробленими модулями збору та обробки системної інформації, що дозволяє побудувати цілісний програмний застосунок у межах єдиного програмного середовища.

Графічний інтерфейс (рис.3.3) розроблюваного застосунку забезпечує не лише відображення результатів моніторингу, а й надає користувачеві можливість керувати основними параметрами роботи системи. Одним із таких параметрів є інтервал оновлення даних, який визначає частоту повторного зчитування системних показників. У зв'язку з цим в інтерфейсі передбачено окремий елемент керування, наприклад поле введення або список вибору, за допомогою якого користувач може встановити бажаний часовий інтервал у секундах. Наявність такого механізму дає змогу адаптувати режим роботи програми до конкретних умов використання: у разі потреби користувач може обрати часте оновлення для оперативного спостереження або, навпаки, збільшити інтервал з метою зменшення навантаження на систему.

Іншою важливою функціональною особливістю інтерфейсу є забезпечення вибору типів системних ресурсів, моніторинг яких буде здійснюватися у конкретний момент часу. Для цього використовуємо прапорці (`'Checkbutton'`), що дозволяють користувачеві самостійно визначати, які саме показники необхідно відстежувати: завантаження центрального процесора, використання оперативної пам'яті, дискову активність або мережеву статистику. Застосування такого підходу є доцільним, оскільки воно надає можливість комбінованого вибору, тобто одночасного моніторингу кількох типів ресурсів. Це підвищує гнучкість програмного засобу, дозволяє зосередитися лише на релевантних показниках та спрощує

інтерпретацію результатів.

Рис. 3.3. Інтерфейс застосунку моніторингу

На рис. 3.4 наведено вигляд вікна інтерфейсу з результатами моніторингу. Окрему роль у побудові інтерфейсу відіграє організація механізмів збереження та подальшого аналізу зібраних даних. У межах розроблюваного застосунку передбачається збереження результатів моніторингу у форматі CSV, що забезпечує простоту реалізації, зручність перегляду та можливість подальшого використання даних у табличних процесорах або інших аналітичних засобах. У зв'язку з цим в інтерфейсі необхідно передбачити кнопку для ініціювання збереження поточних або накопичених результатів до CSV-файлу. Крім того, для підтримки аналітичної складової реалізовано окрему кнопку побудови графіків, що дозволяє відображати зміну обраних показників за певний проміжок часу. Така функція забезпечує перехід від простого числового контролю до візуального аналізу динаміки використання системних ресурсів.

Загалом графічний інтерфейс застосунку організовано таким чином, щоб поєднувати простоту користування, інформативність і функціональну повноту. Для цього передбачено структурований поділ вікна на логічні зони: область налаштувань параметрів моніторингу, область вибору ресурсів, область відображення поточних результатів та область керування збереженням і побудовою графіків. Такий підхід сприяє зручності сприйняття інформації, забезпечує інтуїтивність роботи з програмою та створює основу для подальшої практичної реалізації графічної оболонки засобами `tkinter`.

Рис. 3.4 Результати моніторингу у текстовому вигляді на екрані комп'ютера

У додатку Г наведено код повної програмної реалізації графічного інтерфейсу на tkinter, який звертається до класу DataCollector, дозволяє обирати інтервал оновлення, вибирати типи ресурсів для моніторингу за допомогою прапорців, зберігати результати у CSV та будувати графіки за зібраними даними.

Цей код реалізує графічний інтерфейс застосунку моніторингу системних ресурсів засобами бібліотеки tkinter. Його побудовано так, щоб інтерфейсна частина не містила логіки збору системної інформації безпосередньо, а зверталася до окремого класу DataCollector. Такий підхід відповідає раніше визначеній модульній архітектурі, у якій модулі збору даних, обробки, візуалізації та збереження розглядаються як відносно самостійні компоненти. У межах наведеного прикладу клас MonitoringApp виконує роль інтерфейсного шару, тоді як DataCollector забезпечує доступ до системних показників CPU, оперативної пам'яті, диска та мережі.

Структурно інтерфейс складається з кількох логічних блоків. У верхній частині розміщено область налаштування параметрів моніторингу, де користувач може вибрати інтервал оновлення даних за допомогою списку Combobox. Це реалізує вимогу щодо керування частотою зчитування системної інформації. Далі розташовано блок вибору ресурсів для моніторингу, де використано прапорці Checkbutton. Така форма подання дозволяє обирати довільну комбінацію ресурсів: лише CPU, лише оперативну пам'ять, або будь-яке інше поєднання CPU, RAM, диска та мережі. Саме такий підхід раніше був визначений як доцільний для забезпечення гнучкості застосунку.

Блок кнопок забезпечує основні дії користувача: запуск моніторингу, його зупинку, збереження результатів у CSV, побудову графіків та очищення накопичених даних. Після натискання кнопки запуску активується метод `start_monitoring()`, який перевіряє, чи обрано хоча б один ресурс, а потім запускає періодичне оновлення через метод `update_monitoring()`. У цьому методі дані збираються функцією `collect_selected_data()`, що звертається лише до тих методів DataCollector, які відповідають встановленим прапорцям. Для циклічного виклику використовується метод `root.after(...)`, що є стандартним засобом tkinter для реалізації періодичного оновлення без блокування головного потоку інтерфейсу.

Метод `display_data()` відповідає за відображення поточних результатів у текстовому полі. Він аналізує наявні у словнику ключі та формує зрозуміле представлення для користувача. Якщо були зібрані дані про CPU, відображається завантаження процесора, кількість ядер та поточна частота. Якщо моніториться пам'ять, виводиться відсоток використання та зайнятий обсяг. Аналогічно подаються результати для диска та мережі. Це означає, що інтерфейс динамічно адаптується до вибраної конфігурації моніторингу і не перевантажує користувача зайвою інформацією.

Метод `save_to_csv()` реалізує збереження результатів у форматі CSV, що відповідає вимозі накопичення статистики для подальшого аналізу.

Програма автоматично формує повний перелік полів на основі всіх зібраних записів і записує кожен запис окремим рядком таблиці. Якщо певне значення є списком, наприклад завантаження окремих ядер CPU, воно перетворюється на текстовий рядок. Така організація забезпечує сумісність з електронними таблицями та подальшу зручність аналізу збережених даних.

Метод `plot_graphs()` реалізує побудову графіків за накопиченими результатами. Для кожного обраного типу ресурсу формується окремий графік: завантаження CPU, використання оперативної пам'яті, використання диска або швидкість мережевого обміну. Тут використовується бібліотека `matplotlib`, а побудова виконується лише за тими даними, які дійсно були зібрані. Такий підхід дозволяє поєднати поточний моніторинг у графічному вікні з подальшим аналізом динаміки за певний проміжок часу, що є важливим для дослідницького та прикладного використання системи.

Запропонований інтерфейс повністю відповідає результатам, сформованим у першому та другому розділах роботи. По-перше, у підпункті 1.4 було визначено, що система повинна забезпечувати збір інформації про використання центрального процесора, оперативної пам'яті, дискових ресурсів та мережевої активності, а також візуалізацію цих даних, моніторинг у реальному часі та збереження статистики. Саме ці вимоги реалізовано в наведеному інтерфейсі через систему прапорців, текстове відображення поточних результатів, збереження у CSV та побудову графіків. По-друге, у підпункті 2.1 було виділено модулі збору даних, обробки, візуалізації та збереження статистики. У наведеному коді ця логіка збережена: DataCollector виконує роль модуля збору, методи відображення і графічне вікно реалізують модуль візуалізації, а збереження у CSV – функції підсистеми накопичення статистики.

Крім того, інтерфейс узгоджується з висновками підрозділу 2.3, де було зазначено спільну логіку алгоритмів: ініціалізація, отримання даних, перевірка коректності, обробка, передача на відображення та, за потреби, збереження. У межах програми саме ця логіка і реалізується: користувач задає параметри, програма циклічно викликає потрібні функції збору даних, відображає результати та за командою виконує їх запис або візуальний аналіз у вигляді графіків. Також інтерфейс безпосередньо відповідає положенням підпункту 2.4, де передбачалося надання користувачеві засобів для вибору параметрів моніторингу, відображення поточних даних та керування процесом збереження й аналізу результатів. Отже, запропоноване рішення є не ізольованим фрагментом коду, а практичною реалізацією вимог і проектних рішень, обґрунтованих у попередніх розділах роботи.

3.3. Тестування та апробація програмного засобу

У межах роботи було проведено тестування та апробацію розробленого програмного засобу для моніторингу використання системних ресурсів комп'ютера. Основною метою тестування було перевірити коректність роботи функціональних модулів, стабільність роботи застосунку в режимі реального часу, а також відповідність отриманих результатів фактичному стану системи. Тестування виконувалося на персональному комп'ютері з операційною системою Windows, у середовищі розробки PyCharm, із використанням бібліотеки `psutil` для збору системних показників. Особлива увага приділялася перевірці роботи основних компонентів системи: модуля збору даних, модуля обробки, графічного інтерфейсу користувача та механізмів збереження результатів у форматі CSV.

У процесі тестування було перевірено коректність отримання показників ⁶ центрального процесора, оперативної пам'яті, дискової підсистеми та мережевої активності (рис. 3.5, рис. 3.6, рис. 3.7 та рис. 3.8). Зокрема, для CPU здійснювалася перевірка відповідності значень завантаження процесора при виконанні ресурсомістких задач (наприклад, запуск декількох паралельних процесів). Для оперативної пам'яті аналізувалася зміна обсягу використаної пам'яті під час відкриття та закриття програм. Аналогічно перевірялася реакція системи на зміну дискової активності (копіювання файлів) та мережевого навантаження (завантаження даних з Інтернету). Результати тестування показали, що програмний засіб адекватно реагує на зміну стану системи та забезпечує актуальне відображення показників у режимі реального часу.

Окремим етапом апробації стало порівняння результатів, отриманих розробленим застосунком, із даними стандартних системних засобів моніторингу, зокрема диспетчера задач операційної системи Windows (Task Manager). Для цього одночасно запускалися обидва інструменти, після чого фіксувалися значення завантаження CPU та використання оперативної пам'яті. У результаті було встановлено, що відхилення між показниками не перевищує 1-3%, що пояснюється різницею в алгоритмах вимірювання та часових інтервалах оновлення даних. Таким чином, можна зробити висновок про достатню точність розробленого програмного засобу для практичного використання.

Рис. 3.5. Графік моніторингу CPU

Рис. 3.6. Графік моніторингу RAM

Також було перевірено роботу функції збереження результатів у форматі CSV. У ході тестування підтверджено, що всі зібрані дані коректно записуються у файл, структура якого відповідає табличному поданню, а значення можуть бути легко імпортовані у табличні процесори, такі як Microsoft Excel. Додатково було протестовано функцію побудови графіків, яка дозволяє відобразити зміну показників у часі. Після усунення проблем із backend бібліотеки matplotlib (зокрема, переходу до використання TkAgg) графіки коректно відображаються та відповідають накопиченим даним.

Рис. 3.7. Графік моніторингу NET

Рис. 3.8. Графік моніторингу HDD

Узагальнюючи результати тестування та апробації, можна зробити висновок, що розроблений програмний засіб відповідає поставленим вимогам, сформульованим у першому та другому розділах. Застосунок демонструє стабільну роботу, коректність обробки системної інформації та достатню точність отриманих результатів. Проведена апробація підтверджує можливість практичного використання розробленого програмного засобу.

ВИСНОВКИ

У сучасних умовах цифровізації та зростання вимог до ефективності використання обчислювальних ресурсів актуальним є створення програмних засобів, що забезпечують оперативний контроль стану комп'ютерної системи. Особливої значущості такі засоби набувають у навчальному процесі та практичній діяльності, де необхідно не лише отримувати поточні показники, а й аналізувати динаміку використання ресурсів. У межах даної бакалаврської роботи було поставлено за мету розробити програмний засіб для моніторингу використання ресурсів комп'ютера із використанням мови програмування Python та сучасних бібліотек доступу до системної інформації. Для досягнення цієї мети було сформульовано ряд завдань, що охоплюють аналіз предметної області, проектування архітектури системи, розробку алгоритмів, реалізацію програмного застосунку та його тестування.

У ході виконання першого завдання було здійснено аналіз поняття та призначення моніторингу системних ресурсів, а також розглянуто існуючі програмні засоби, такі як диспетчер задач операційної системи, спеціалізовані утиліти та системи візуалізації. Проведений аналіз дозволив визначити основні функціональні можливості таких систем, їх переваги та обмеження, а також сформулювати вимоги до розроблюваного програмного засобу. Крім того, було досліджено сучасні технології та бібліотеки, що використовуються для реалізації системного моніторингу, зокрема бібліотеку `psutil`, що стала основою для збору системної інформації.

У межах другого завдання було виконано проектування програмного засобу, зокрема визначено його загальну архітектуру, що передбачає поділ на модулі збору даних, обробки, візуалізації та збереження інформації. Було розроблено моделі системи, включаючи структуру компонентів, діаграми взаємодії та алгоритми збору та обробки системних показників. Особливу увагу приділено розробці уніфікованих алгоритмів, які мають спільну структуру та відрізняються специфікою оброблюваних даних. Це забезпечило цілісність архітектурного рішення та спростило подальшу реалізацію програмного засобу.

У межах третього завдання було здійснено програмну реалізацію розробленої системи із використанням мови Python. Було створено модуль збору даних, що забезпечує отримання інформації про використання центрального процесора, оперативної пам'яті, дискової підсистеми та мережевої активності. Реалізовано графічний інтерфейс користувача засобами бібліотеки `tkinter`, який дозволяє керувати параметрами моніторингу, обирати ресурси для спостереження, відображати поточні результати, зберігати дані у форматі CSV та будувати графіки змін показників у часі. Така реалізація відповідає принципам модульності та забезпечує зручність використання програмного засобу.

У процесі виконання завершального завдання було проведено тестування та апробацію розробленого програмного засобу. Було перевірено коректність роботи всіх функціональних компонентів, стабільність роботи в режимі реального часу, а також відповідність отриманих результатів фактичним показникам системи. Проведене порівняння із вбудованими системними засобами показало достатню точність отриманих даних, що підтверджує правильність обраних алгоритмів і підходів до реалізації. Додатково було перевірено коректність збереження результатів та побудови графіків.

Отже, у результаті виконання бакалаврської роботи було досягнуто поставленої мети та вирішено всі сформульовані завдання. Розроблений

програмний засіб забезпечує ефективний моніторинг використання системних ресурсів комп'ютера, має зручний інтерфейс користувача та може бути використаний як у навчальному процесі, так і для практичних задач аналізу стану комп'ютерних систем. Отримані результати можуть слугувати основою для подальшого розвитку програмного засобу, зокрема розширення функціональності, додавання нових методів аналізу та інтеграції з іншими інформаційними системами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Караблін П.В. Розробка сайту "Порівняння утиліт для моніторингу стану апаратних засобів ПК": Електронний архів Полтавського університету економіки і торгівлі. Електронний архів Полтавського університету економіки і торгівлі. URL: <http://dspace.puet.edu.ua/handle/123456789/15433>.
2. 12 НАЙКРАЩИХ програмних засобів моніторингу системи (2026). Guru99. URL: <https://www.guru99.com/uk/best-system-monitoring-software.html>.
3. A computer network monitoring system / D.E. Morgan et al. IEEE Transactions on Software Engineering. 1975. SE-1, no. 3. P. 299-311. URL: <https://doi.org/10.1109/tse.1975.6312855>.
4. Abraham Silberschatz; Greg Gagne; Peter B. Galvin. Operating System Concepts. Wiley Global Education US, 2018. 1536 p.
5. Aldea C.L., Bocu R., Solca R.N. Real-Time Monitoring and Management of Hardware and Software Resources in Heterogeneous Computer Networks through an Integrated System Architecture. Symmetry. 2023. Vol. 15, no. 6. P. 1134. URL: <https://doi.org/10.3390/sym15061134>.
6. Bhosale P. Task Manager. INTERNATIONAL JOURNAL OF SCIENTIFIC RESEARCH IN ENGINEERING AND MANAGEMENT. 2024. Vol. 08, no. 04. P. 1-5. URL: <https://doi.org/10.55041/ijjsrem32239>.
7. <https://psutil.readthedocs.io/en/latest/>
8. INVESTIGATING THE HARDWARE STATUS OF THE UNIVERSITY'S LOCAL NETWORK USING PYTHON / Kandyba et al. Scientific notes of Taurida National V.I. Vernadsky University. Series: Technical Sciences. 2022. No. 3. P. 44-49. URL: <https://doi.org/10.32838/2663-5941/2022.3/07>.
9. Jones B.K. Python Cookbook. 3rd ed. O'Reilly Media, 2013. 706 p.
10. Jones B.K. Python Cookbook. 3rd ed. O'Reilly Media, 2013. 706 p.
11. Love R. Linux System Programming. Shroff Publishers & Distributors Pvt Ltd, 2014. 480 p.
12. Process Explorer - Sysinternals. Microsoft Learn: Build skills that open doors in your career. URL: <https://learn.microsoft.com/en-us/sysinternals/downloads/process-explorer>.
13. Professional System Information and Diagnostics Comprehensive Hardware Analysis, Monitoring and Reporting for Windows and DOS. HWINFO. URL: <https://www.hwinfo.com/about-software/>.
14. psutil. PyPI. URL: <https://pypi.org/project/psutil>.
15. Rashmi M. Computer Lab Monitoring System. International Journal on Recent and Innovation Trends in Computing and Communication. 2015. Vol. 3, no. 3. P. 1652-1656. URL: <https://doi.org/10.17762/ijritcc2321-8169.1503165>.
16. Sharma S. top vs htop: What's the Difference. Linux Handbook. URL: <https://linuxhandbook.com/top-vs-htop> (date of access: 23.03.2026).
17. Stanojevic M. 13 Best System Resource Monitoring Tools for Windows PCs. WindowsReport.com. URL: <https://windowsreport.com/monitor-system-resources-windows-10/>.
18. Tanenbaum A.S., Bos H. Modern Operating Systems: Global Edition. Pearson Education, Limited, 2015. 1138 p.
19. Task Manager Android App. International Research Journal of Modernization in Engineering Technology & Science. 2026. URL: <https://doi.org/10.56726/irjmets91061>.
20. The Research of Computer Monitoring System / X. Li et al. Applied Mechanics and Materials. 2014. Vol. 635-637. P. 1648-1653. URL: <https://doi.org/10.4028/www.scientific.net/amm.635-637.1648>.
21. Troubleshoot processes by using Task Manager - Windows Server. Microsoft Learn: Build skills that open doors in your career. URL: <https://learn.microsoft.com/en-us/troubleshoot/windows-server/support-tools/support-tools-task-manager>.
22. Weik M.H. monitoring system. Computer Science and Communications Dictionary. Boston, MA, 2000. P. 1042. URL: https://doi.org/10.1007/1-4020-0613-6_11778.
23. Weik M.H. system resources manager. Computer Science and Communications Dictionary. Boston, MA, 2000. P. 1724. URL: https://doi.org/10.1007/1-4020-0613-6_18931.

Додатки

Додаток А. Код класу DataCollector з функцією

```
import psutil
```

```
import time
```

```
from datetime import datetime
```

```
from typing import Any, Dict, List
```

```
class DataCollector:
```

```
    """
```

```
    Клас для збору системної інформації.
```

```
    Поточна реалізація містить функції для отримання параметрів CPU.
```

```
    """
```

```
    def __init__(self) -> None:
```

```
        """
```

```
        Ініціалізація об'єкта збору даних.
```

```
        """
```

```
        pass
```

```
    def get_cpu_info(self, interval: float = 1.0) -> Dict[str, Any]:
```

```
        """
```

Отримує основні параметри CPU.

:param interval: інтервал у секундах для вимірювання завантаження CPU.
Чим більше значення, тим точніше оцінка поточного навантаження,
але тим довше виконується функція.

:return: словник з основними параметрами CPU.

"""

try:

```
timestamp: str = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
```

```
# Загальне завантаження CPU за інтервал
```

```
total_percent: float = psutil.cpu_percent(interval=interval)
```

```
# Завантаження кожного ядра окремо
```

```
per_core_percent: List[float] = psutil.cpu_percent(interval=None, percpu=True)
```

```
# Кількість фізичних ядер
```

```
physical_cores: int | None = psutil.cpu_count(logical=False)
```

```
# Кількість логічних ядер (потоків)
```

```
logical_cores: int | None = psutil.cpu_count(logical=True)
```

```
# Частота CPU
```

```
freq = psutil.cpu_freq()
```

```
if freq is not None:
```

```
    current_frequency: float = round(freq.current, 2)
```

```
    min_frequency: float = round(freq.min, 2)
```

```
    max_frequency: float = round(freq.max, 2)
```

```
else:
```

```
    current_frequency = 0.0
```

```
    min_frequency = 0.0
```

```
    max_frequency = 0.0
```

```
# Додаткові статистичні показники
```

```
cpu_stats = psutil.cpu_stats()
```

```
cpu_data: Dict[str, Any] = {
```

```
    "timestamp": timestamp,
```

```
    "total_percent": round(total_percent, 2),
```

```
    "per_core_percent": [round(value, 2) for value in per_core_percent],
```

```
    "physical_cores": physical_cores,
```

```
    "logical_cores": logical_cores,
```

```
    "current_frequency_mhz": current_frequency,
```

```
    "min_frequency_mhz": min_frequency,
```

```
    "max_frequency_mhz": max_frequency,
```

```
    "context_switches": cpu_stats.ctx_switches,
```

```
    "interrupts": cpu_stats.interrupts,
```

```
    "soft_interrupts": getattr(cpu_stats, "soft_interrupts", 0),
```

```
    "system_calls": getattr(cpu_stats, "syscalls", 0),
```

```
}
```

```
return cpu_data
```

```
except Exception as error:
```

```
    return {
```

```
        "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
```

```
        "error": True,
```

```
        "message": f"Помилка отримання параметрів CPU: {error}"
```

```
    }
```

```
def get_cpu_load_fast(self) -> Dict[str, Any]:
```

```
    """
```

Швидке отримання поточного завантаження CPU без очікування інтервалу.

Підходить для швидкого оновлення в інтерфейсі.

:return: словник з короткою інформацією про завантаження CPU.

```
    """
```

```
try:
    timestamp: str = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    total_percent: float = psutil.cpu_percent(interval=None)
    per_core_percent: List[float] = psutil.cpu_percent(interval=None, percpu=True)
```

```
return {
    "timestamp": timestamp,
    "total_percent": round(total_percent, 2),
    "per_core_percent": [round(value, 2) for value in per_core_percent]
}
```

except Exception as error:

```
return {
    "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
    "error": True,
    "message": f"Помилка швидкого отримання навантаження CPU: {error}"
}
```

```
if __name__ == "__main__":
```

```
    collector = DataCollector()
```

```
    print("Повні параметри CPU:")
```

```
    cpu_info = collector.get_cpu_info(interval=1.0)
```

```
    print(cpu_info)
```

```
    print("\nШвидке опитування CPU:")
```

```
    fast_cpu = collector.get_cpu_load_fast()
```

```
    print(fast_cpu)
```

Додаток Б Код для роботи з RAM

```
import psutil
```

```
import time
```

```
from datetime import datetime from typing import Any, Dict, List
```

```
class DataCollector:
```

```
    def __init__(self) -> None:
```

```
        """
```

```
        Ініціалізація об'єкта збору даних.
```

```
        """
```

```
        pass
```

```
import psutil
```

```
from datetime import datetime from typing import Any, Dict
```

```
class DataCollector:
```

```
    """
```

```
    Клас для збору системної інформації.
```

```
    Містить функції для отримання параметрів CPU та оперативної пам'яті.
```

```
    """
```

```
    def __init__(self) -> None:
```

```
        pass
```

```
    def get_memory_info(self) -> Dict[str, Any]:
```

```
        """
```

```
        Отримує детальну інформацію про використання оперативної пам'яті.
```

```
        :return: словник з параметрами пам'яті
```

```
        """
```

```
        try:
```

```
            timestamp: str = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
```

```
            mem = psutil.virtual_memory()
```

```
            total_memory: float = round(mem.total / (1024 ** 3), 2) # GB
```

```

available_memory: float = round(mem.available / (1024 ** 3), 2)
used_memory: float = round(mem.used / (1024 ** 3), 2)
percent_used: float = round(mem.percent, 2)

```

```

memory_data: Dict[str, Any] = {
    "timestamp": timestamp,
    "total_gb": total_memory,
    "available_gb": available_memory,
    "used_gb": used_memory,
    "percent_used": percent_used
}

```

```

return memory_data

```

except Exception as error:

```

return {
    "timestamp": datetime.now().strftime("%Y-%m-%d.%H:%M:%S"),
    "error": True,
    "message": f"Помилка отримання даних пам'яті: {error}"
}

```

def get_memory_fast(self) -> Dict[str, Any]:

"""

Швидке отримання основних параметрів пам'яті.
Використовується для оновлення GUI.

:return: словник з базовими параметрами

"""

try:

```

timestamp: str = datetime.now().strftime("%Y-%m-%d.%H:%M:%S")

```

```

mem = psutil.virtual_memory()

```

```

return {
    "timestamp": timestamp,
    "percent_used": round(mem.percent, 2),
    "used_gb": round(mem.used / (1024 ** 3), 2)
}

```

except Exception as error:

```

return {
    "timestamp": datetime.now().strftime("%Y-%m-%d.%H:%M:%S"),
    "error": True,
    "message": f"Помилка швидкого отримання пам'яті: {error}"
}

```

if __name__ == "__main__":

```

collector = DataCollector()

```

```

print("Детальна інформація про пам'ять:")
print(collector.get_memory_info())

```

```

print("\nШвидкий режим:")
print(collector.get_memory_fast())

```

def get_cpu_load_fast(self) -> Dict[str, Any]:

"""

Швидке отримання поточного завантаження CPU без очікування інтервалу.
Підходить для швидкого оновлення в інтерфейсі.

:return: словник з короткою інформацією про завантаження CPU.

"""

try:

```

timestamp: str = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
total_percent: float = psutil.cpu_percent(interval=None)
per_core_percent: List[float] = psutil.cpu_percent(interval=None, percpu=True)

```

```

return {
    "timestamp": timestamp,
    "total_percent": round(total_percent, 2),
    "per_core_percent": [round(value, 2) for value in per_core_percent]
}

```

except Exception as error:

```

return {
    8 "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
    "error": True,
    "message": f"Помилка швидкого отримання навантаження CPU: {error}"
}

```

if __name__ == "__main__":

collector = DataCollector()

print("Повні параметри CPU:")

cpu_info = collector.get_cpu_info(interval=1.0)

print(cpu_info)

print("\nШвидке опитування CPU:")

fast_cpu = collector.get_cpu_load_fast()

print(fast_cpu)

Додаток В. Код моніторингу дискової активності

import psutil

from datetime import datetime

from typing import Any, Dict

class DataCollector:

"""

Клас для збору системної інформації.

Містить функції для моніторингу дискової активності.

"""

def __init__(self) -> None:

pass

def get_disk_info(self) -> Dict[str, Any]:

"""

Отримує детальну інформацію про дискову підсистему.

:return: словник з параметрами диска

"""

try:

timestamp: str = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

disk_usage = psutil.disk_usage('/')

disk_io = psutil.disk_io_counters()

total_disk: float = round(disk_usage.total / (1024 ** 3), 2)

used_disk: float = round(disk_usage.used / (1024 ** 3), 2)

free_disk: float = round(disk_usage.free / (1024 ** 3), 2)

percent_used: float = round(disk_usage.percent, 2)

read_bytes: float = round(disk_io.read_bytes / (1024 ** 2), 2) # MB

write_bytes: float = round(disk_io.write_bytes / (1024 ** 2), 2)

read_count: int = disk_io.read_count

write_count: int = disk_io.write_count

disk_data: Dict[str, Any] = {

"timestamp": timestamp,

"total_gb": total_disk,

"used_gb": used_disk,

"free_gb": free_disk,

```

        "percent_used": percent_used,
        "read_mb": read_bytes,
        "write_mb": write_bytes,
        "read_count": read_count,
        "write_count": write_count
    }

```

```

    return disk_data

```

```

except Exception as error:

```

```

    return {
        "timestamp": datetime.now().strftime("%Y-%m-%d.%H:%M:%S"),
        "error": True,
        "message": f"Помилка отримання даних диска: {error}"
    }

```

```

def get_disk_io_fast(self) -> Dict[str, Any]:

```

```

    """

```

```

    Швидке отримання даних про дискову активність.

```

```

    :return: словник з основними показниками

```

```

    """

```

```

    try:

```

```

        timestamp: str = datetime.now().strftime("%Y-%m-%d.%H:%M:%S")

```

```

        disk_io = psutil.disk_io_counters()

```

```

    return {
        "timestamp": timestamp,
        "read_mb": round(disk_io.read_bytes / (1024 ** 2), 2),
        "write_mb": round(disk_io.write_bytes / (1024 ** 2), 2)
    }

```

```

except Exception as error:

```

```

    return {
        "timestamp": datetime.now().strftime("%Y-%m-%d.%H:%M:%S"),
        "error": True,
        "message": f"Помилка швидкого отримання IO: {error}"
    }

```

```

if __name__ == "__main__":

```

```

    collector = DataCollector()

```

```

    print("Детальна інформація про диск:")

```

```

    print(collector.get_disk_info())

```

```

    print("\nШвидкий режим IO:")

```

```

    print(collector.get_disk_io_fast())

```

Додаток Г. Код для роботи з NET

```

import psutil

```

```

from datetime import datetime

```

```

from typing import Any, Dict, Optional

```

```

class DataCollector:

```

```

    """

```

```

    Клас для збору системної інформації.

```

```

    Містить функції для моніторингу мережевої активності.

```

```

    """

```

```

    def __init__(self) -> None:

```

```

        # Збереження попередніх значень для обчислення дельти

```

```

        self._prev_net: Optional[Dict[str, float]] = None

```

```

    def get_network_info(self) -> Dict[str, Any]:

```

```

    """

```

Отримує накопичувальні показники мережі.

:return: словник з базовою мережею статистикою

"""

try:

```
8 timestamp = datetime.now().strftime("%Y-%m-%d.%H:%M:%S")
```

```
net = psutil.net_io_counters()
```

```
return {
```

```
    "timestamp": timestamp,
    "bytes_sent_mb": round(net.bytes_sent / (1024 ** 2), 2),
    "bytes_recv_mb": round(net.bytes_recv / (1024 ** 2), 2),
    "packets_sent": net.packets_sent,
    "packets_recv": net.packets_recv
}
```

except Exception as error:

```
return {
```

```
8 timestamp": datetime.now().strftime("%Y-%m-%d.%H:%M:%S"),
    "error": True,
    "message": f"Помилка отримання мережевих даних: {error}"
}
```

def get_network_speed(self, interval: float = 1.0) -> Dict[str, Any]:

"""

Обчислює швидкість передачі даних (дельту).

:param interval: інтервал часу між вимірюваннями (сек)

:return: словник зі швидкістю передачі даних

"""

try:

```
8 timestamp = datetime.now().strftime("%Y-%m-%d.%H:%M:%S")
```

```
net = psutil.net_io_counters()
```

```
current = {
```

```
    "bytes_sent": net.bytes_sent,
    "bytes_recv": net.bytes_recv
}
```

```
if self._prev_net is None:
```

```
    self._prev_net = current
```

```
    return {
```

```
        "timestamp": timestamp,
        "upload_kb_s": 0.0,
        "download_kb_s": 0.0
    }
```

```
# Обчислення дельти
```

```
delta_sent = current["bytes_sent"] - self._prev_net["bytes_sent"]
```

```
delta_recv = current["bytes_recv"] - self._prev_net["bytes_recv"]
```

```
# Переведення у швидкість (КБ/с)
```

```
upload_speed = round(delta_sent / 1024 / interval, 2)
```

```
download_speed = round(delta_recv / 1024 / interval, 2)
```

```
# Оновлення попередніх значень
```

```
self._prev_net = current
```

```
return {
```

```
    "timestamp": timestamp,
    "upload_kb_s": upload_speed,
    "download_kb_s": download_speed
}
```

except Exception as error:

```

return {
    8 timestamp: datetime.now().strftime("%Y-%m-%d.%H:%M:%S"),
    "error": True,
    "message": "Помилка обчислення швидкості: {error}"
}

```

```

if __name__ == "__main__":
    import time

```

```

collector = DataCollector()

```

```

print("Тест швидкості мережі:")
for _ in range(5):
    print(collector.get_network_speed(interval=1.0))
    time.sleep(1)

```

Додаток Д. Код графічного інтерфейсу

```

import csv
import os
import 3 time from datetime import datetime from typing import Any, Dict, List, Optional

```

```

12 import psutil
import tkinter as tk from tkinter import ttk, messagebox, filedialog

```

```

12 import matplotlib.pyplot as plt

```

```

class DataCollector:

```

```

    """
    Клас збору системної інформації.
    Повертає дані у вигляді словників, що спрощує інтеграцію з GUI.
    """

```

```

def __init__(self) -> None:
    self._prev_net: Optional[Dict[str, float]] = None

```

```

def get_cpu_info(self, interval: float = 0.5) -> Dict[str, Any]:

```

```

    try:
        13 timestamp = datetime.now().strftime("%Y-%m-%d.%H:%M:%S")

```

```

        total_percent = psutil.cpu_percent(interval=interval)
        per_core_percent = psutil.cpu_percent(interval=None, percpu=True)
        physical_cores = psutil.cpu_count(logical=False)
        logical_cores = psutil.cpu_count(logical=True)

```

```

        freq = psutil.cpu_freq()
        if freq is not None:
            current_frequency = round(freq.current, 2)
            min_frequency = round(freq.min, 2)
            max_frequency = round(freq.max, 2)

```

```

        else:
            current_frequency = 0.0
            min_frequency = 0.0
            max_frequency = 0.0

```

```

        cpu_stats = psutil.cpu_stats()

```

```

    return {
        "timestamp": timestamp,
        "cpu_total_percent": round(total_percent, 2),
        "cpu_per_core_percent": [round(v, 2) for v in per_core_percent],
        "cpu_physical_cores": physical_cores,
        "cpu_logical_cores": logical_cores,
        "cpu_current_frequency_mhz": current_frequency,
        "cpu_min_frequency_mhz": min_frequency,
        "cpu_max_frequency_mhz": max_frequency,
        "cpu_context_switches": cpu_stats.ctx_switches,
    }

```

```

"cpu_interrupts": cpu_stats.interrupts,
"cpu_soft_interrupts": getattr(cpu_stats, "soft_interrupts", 0),
"cpu_system_calls": getattr(cpu_stats, "syscalls", 0),
}

```

except Exception as error:

```

return {
    "timestamp": datetime.now().strftime("%Y-%m-%d.%H:%M:%S"),
    "error": True,
    "message": f"Помилка отримання параметрів CPU: {error}",
}

```

def get_memory_info(self) -> Dict[str, Any]:

```

try:
    timestamp = datetime.now().strftime("%Y-%m-%d.%H:%M:%S")
    mem = psutil.virtual_memory()

```

```

return {
    "timestamp": timestamp,
    "memory_total_gb": round(mem.total / (1024 ** 3), 2),
    "memory_available_gb": round(mem.available / (1024 ** 3), 2),
    "memory_used_gb": round(mem.used / (1024 ** 3), 2),
    "memory_percent_used": round(mem.percent, 2),
}

```

except Exception as error:

```

return {
    "timestamp": datetime.now().strftime("%Y-%m-%d.%H:%M:%S"),
    "error": True,
    "message": f"Помилка отримання даних пам'яті: {error}",
}

```

def get_disk_info(self, path: str = "/") -> Dict[str, Any]:

```

try:
    timestamp = datetime.now().strftime("%Y-%m-%d.%H:%M:%S")

```

```

if os.name == "nt":
    path = "C:\\"

```

```

disk_usage = psutil.disk_usage(path)
disk_io = psutil.disk_io_counters()

```

```

return {
    "timestamp": timestamp,
    "disk_total_gb": round(disk_usage.total / (1024 ** 3), 2),
    "disk_used_gb": round(disk_usage.used / (1024 ** 3), 2),
    "disk_free_gb": round(disk_usage.free / (1024 ** 3), 2),
    "disk_percent_used": round(disk_usage.percent, 2),
    "disk_read_mb": round(disk_io.read_bytes / (1024 ** 2), 2) if disk_io else 0.0,
    "disk_write_mb": round(disk_io.write_bytes / (1024 ** 2), 2) if disk_io else 0.0,
    "disk_read_count": disk_io.read_count if disk_io else 0,
    "disk_write_count": disk_io.write_count if disk_io else 0,
}

```

except Exception as error:

```

return {
    "timestamp": datetime.now().strftime("%Y-%m-%d.%H:%M:%S"),
    "error": True,
    "message": f"Помилка отримання даних диска: {error}",
}

```

def get_network_info(self) -> Dict[str, Any]:

```

try:
    timestamp = datetime.now().strftime("%Y-%m-%d.%H:%M:%S")
    net = psutil.net_io_counters()

```

```

return {
    "timestamp": timestamp,
    "net_bytes_sent_mb": round(net.bytes_sent / (1024 ** 2), 2),
    "net_bytes_recv_mb": round(net.bytes_recv / (1024 ** 2), 2),
}

```

```

        "net_packets_sent": net.packets_sent,
        "net_packets_rcv": net.packets_rcv,
    }

```

except Exception as error:

```

    return {
        "timestamp": datetime.now().strftime("%Y-%m-%d.%H:%M:%S"),
        "error": True,
        "message": f"Помилка отримання мережевих даних: {error}",
    }

```

def get_network_speed(self, interval: float = 1.0) -> Dict[str, Any]:

```

    try:
        timestamp = datetime.now().strftime("%Y-%m-%d.%H:%M:%S")
        net = psutil.net_io_counters()

        current = {
            "bytes_sent": net.bytes_sent,
            "bytes_rcv": net.bytes_rcv,
        }

        if self._prev_net is None:
            self._prev_net = current
            return {
                "timestamp": timestamp,
                "net_upload_kb_s": 0.0,
                "net_download_kb_s": 0.0,
            }

        delta_sent = current["bytes_sent"] - self._prev_net["bytes_sent"]
        delta_rcv = current["bytes_rcv"] - self._prev_net["bytes_rcv"]

        upload_speed = round(delta_sent / 1024 / interval, 2)
        download_speed = round(delta_rcv / 1024 / interval, 2)

        self._prev_net = current

        return {
            "timestamp": timestamp,
            "net_upload_kb_s": upload_speed,
            "net_download_kb_s": download_speed,
        }
    except Exception as error:
        return {
            "timestamp": datetime.now().strftime("%Y-%m-%d.%H:%M:%S"),
            "error": True,
            "message": f"Помилка обчислення швидкості мережі: {error}",
        }

```

class MonitoringApp:

```

    def __init__(self, root: tk.Tk) -> None:
        self.root = root
        self.root.title("Моніторинг системних ресурсів")
        self.root.geometry("950x700")

        self.collector = DataCollector()
        self.monitoring = False
        self.records: List[Dict[str, Any]] = []

        self.interval_var = tk.StringVar(value="1")
        self.cpu_var = tk.BooleanVar(value=True)
        self.memory_var = tk.BooleanVar(value=True)
        self.disk_var = tk.BooleanVar(value=False)
        self.network_var = tk.BooleanVar(value=False)

        self._build_interface()

    def _build_interface(self) -> None:
        top_frame = ttk.LabelFrame(self.root, text="Параметри моніторингу")
        top_frame.pack(fill="x", padx=10, pady=10)

```

```

9 tk.Label( top_frame, text="Інтервал оновлення (сек):", grid(row=0, column=0, padx=5, pady=5, sticky="w"))
interval_box = tk.Combobox(
    top_frame,
    textvariable=self.interval_var,
    values=["1", "2", "5", "10"],
    width=10,
    state="readonly"
)
interval_box.grid(row=0, column=1, padx=5, pady=5, sticky="w")

resource_frame = ttk.LabelFrame(self.root, text="Ресурси для моніторингу")
resource_frame.pack(fill="x", padx=10, pady=5)
9 tk.Checkbutton(resource_frame, text="CPU", variable=self.cpu_var, grid(row=0, column=0,
padx=10, pady=5, sticky="w"))
17 tk.Checkbutton(resource_frame, text="RAM", variable=self.memory_var, grid(row=0, column=1, padx=10, pady=5, sticky="w"))
9 tk.Checkbutton(resource_frame, text="Диск", variable=self.disk_var, grid(row=0, column=2, padx=10, pady=5, sticky="w"))
9 tk.Checkbutton(resource_frame, text="Мережа", variable=self.network_var, grid(row=0, column=3, padx=10, pady=5, sticky="w"))

10 button_frame = tk.Frame(self.root)
button_frame.pack(fill="x", padx=10, pady=10)

9 tk.Button(button_frame, text="Почати моніторинг", command=self.start_monitoring).pack(side="left", padx=5)
tk.Button(button_frame, text="Зупинити моніторинг", command=self.stop_monitoring).pack(side="left", padx=5)
9 tk.Button(button_frame, text="Зберегти CSV", command=self.save_to_csv).pack(side="left", padx=5)
tk.Button(button_frame, text="Побудувати графіки", command=self.plot_graphs).pack(side="left", padx=5)
tk.Button(button_frame, text="Очистити дані", command=self.clear_data).pack(side="left", padx=5)

output_frame = ttk.LabelFrame(self.root, text="Поточні результати")
11 output_frame.pack(fill="both", expand=True, padx=10, pady=10)
self.text_output = tk.Text(output_frame, wrap="word", font=("Consolas", 10))
self.text_output.pack(fill="both", expand=True, padx=5, pady=5)
11 def start_monitoring(self) -> None:
    if not any([self.cpu_var.get(), self.memory_var.get(), self.disk_var.get(), self.network_var.get()]):
        messagebox.showwarning("Попередження", "Оберіть хоча б один ресурс для моніторингу.")
        return

    self.monitoring = True
    self.text_output.insert(tk.END, "Моніторинг запущено.\n")
    self.text_output.see(tk.END)
    self.update_monitoring()

def stop_monitoring(self) -> None:
    self.monitoring = False
    self.text_output.insert(tk.END, "Моніторинг зупинено.\n")
    self.text_output.see(tk.END)

def clear_data(self) -> None:
    self.records.clear()
    self.text_output.delete(1.0, tk.END)
    messagebox.showinfo("Інформація", "Накопичені дані очищено.")

def collect_selected_data(self) -> Dict[str, Any]:
    interval = float(self.interval_var.get())
    8 timestamp = datetime.now().strftime("%Y-%m-%d.%H:%M:%S")

    result: Dict[str, Any] = {"timestamp": timestamp}

    if self.cpu_var.get():
        result.update(self.collector.get_cpu_info(interval=0.1))

    if self.memory_var.get():
        result.update(self.collector.get_memory_info())

    if self.disk_var.get():
        result.update(self.collector.get_disk_info())

    if self.network_var.get():
        result.update(self.collector.get_network_info())
        result.update(self.collector.get_network_speed(interval=interval))

    return result

```

```

def update_monitoring(self) -> None:
    if not self.monitoring:
        return

    try:
        interval_ms = int(float(self.interval_var.get()) * 1000)
    except ValueError:
        messagebox.showerror("Помилка", "Некоректний інтервал оновлення.")
        self.monitoring = False
        return

    data = self.collect_selected_data()
    self.records.append(data)
    self.display_data(data)

    self.root.after(interval_ms, self.update_monitoring)

def display_data(self, data: Dict[str, Any]) -> None:
    lines = [f"\nЧас: {data.get('timestamp', '-')}\n"]

    if "cpu_total_percent" in data:
        lines.append(f"CPU: {data['cpu_total_percent']} %")
        lines.append(f"Ядер (фіз./лог.): {data.get('cpu_physical_cores', '-')} / {data.get('cpu_logical_cores', '-')}")
        lines.append(f"Частота CPU: {data.get('cpu_current_frequency_mhz', '-')} МГц")

    if "memory_percent_used" in data:
        lines.append(
            f"RAM: {data['memory_percent_used']} % "
            f"(використано {data.get('memory_used_gb', '-')} ГБ із {data.get('memory_total_gb', '-')} ГБ)"
        )

    if "disk_percent_used" in data:
        lines.append(
            f"Диск: {data['disk_percent_used']} % "
            f"(використано {data.get('disk_used_gb', '-')} ГБ із {data.get('disk_total_gb', '-')} ГБ)"
        )
        lines.append(
            f"ІО диск: читання {data.get('disk_read_mb', '-')} МБ, "
            f"запис {data.get('disk_write_mb', '-')} МБ"
        )

    if "net_upload_kb_s" in data or "net_download_kb_s" in data:
        lines.append(
            f"Мережа: Upload {data.get('net_upload_kb_s', '-')} КБ/с, "
            f"Download {data.get('net_download_kb_s', '-')} КБ/с"
        )

    self.text_output.insert(tk.END, "\n".join(lines) + "\n" + ("-" * 70) + "\n")
    self.text_output.see(tk.END)

def save_to_csv(self) -> None:
    if not self.records:
        messagebox.showwarning("Попередження", "Немає даних для збереження.")
        return

    filepath = filedialog.asksaveasfilename(
        defaultextension=".csv",
        filetypes=[("CSV files", "*.csv")],
        title="Зберегти результати"
    )

    if not filepath:
        return

    all_keys = set()
    for record in self.records:
        all_keys.update(record.keys())

```

```
fieldnames = sorted(all_keys)
```

```
try:
```

```
15 with open(filepath, mode="w", newline="", encoding="utf-8-sig") as file:  
    writer = csv.DictWriter( file, fieldnames=fieldnames)  
    writer.writeheader()  
    for record in self.records:  
        row = {}  
        for key in fieldnames:  
            value = record.get(key, "")  
            if isinstance(value, list):  
                row[key] = ";".join(map(str, value))  
            else:  
                row[key] = value  
2        writer.writerow(row)
```

```
    messagebox.showinfo("Успіх", f"Дані збережено у файл:\n(filepath)")
```

```
except Exception as error:  
    messagebox.showerror("Помилка", f"Не вдалося зберегти CSV:\n{error}")
```

```
def plot_graphs(self) -> None:
```

```
    if not self.records:  
        messagebox.showwarning("Попередження", "Немає даних для побудови графіків.")  
    return
```

```
    times = [record.get("timestamp", "") for record in self.records]
```

```
    plotted = False
```

```
    if self.cpu_var.get():
```

```
        cpu_values = [record.get("cpu_total_percent") for record in self.records if "cpu_total_percent" in record]
```

```
        cpu_times = [record.get("timestamp") for record in self.records if "cpu_total_percent" in record]
```

```
        if cpu_values:
```

```
            plt.figure(figsize=(10, 4))  
            plt.plot(cpu_times, cpu_values, marker="o")  
            plt.title("Завантаження CPU")  
            plt.xlabel("Час")  
            plt.ylabel("%")  
            plt.xticks(rotation=45)  
            plt.tight_layout()  
            plotted = True
```

```
    if self.memory_var.get():
```

```
        mem_values = [record.get("memory_percent_used") for record in self.records if "memory_percent_used" in record]
```

```
        mem_times = [record.get("timestamp") for record in self.records if "memory_percent_used" in record]
```

```
        if mem_values:
```

```
            plt.figure(figsize=(10, 4))  
            plt.plot(mem_times, mem_values, marker="o")  
            plt.title("Використання оперативної пам'яті")  
            plt.xlabel("Час")  
            plt.ylabel("%")  
            plt.xticks(rotation=45)  
            plt.tight_layout()  
            plotted = True
```

```
    if self.disk_var.get():
```

```
        disk_values = [record.get("disk_percent_used") for record in self.records if "disk_percent_used" in record]
```

```
        disk_times = [record.get("timestamp") for record in self.records if "disk_percent_used" in record]
```

```
        if disk_values:
```

```
            plt.figure(figsize=(10, 4))  
            plt.plot(disk_times, disk_values, marker="o")  
            plt.title("Використання диска")  
            plt.xlabel("Час")  
            plt.ylabel("%")  
            plt.xticks(rotation=45)  
            plt.tight_layout()  
            plotted = True
```

```

if self.network_var.get():
    net_values = [record.get("net_download_kb_s") for record in self.records if "net_download_kb_s" in record]
    net_times = [record.get("timestamp") for record in self.records if "net_download_kb_s" in record]
    if net_values:
        plt.figure(figsize=(10, 4))
        plt.plot(net_times, net_values, marker="o")
        plt.title("Швидкість прийому даних")
        plt.xlabel("Час")
        plt.ylabel("КБ/с")
        plt.xticks(rotation=45)
        plt.tight_layout()
        plotted = True

    if plotted:
        plt.show()
    else:
        messagebox.showwarning("Попередження", "Немає придатних даних для побудови графіків.")

```

```

5 if __name__ == "__main__":
    root = tk.Tk() app = MonitoringApp( root)
    root.mainloop()

```